

Стюарт
Рассел

Питер
Норвиг

Искусственный интеллект
Современный подход

Четвертое издание

Том I

Решение проблем: знания и рассуждения

Искусственный интеллект
Современный подход

Четвертое издание

Том I.
Решение проблем: знания и рассуждения

Artificial Intelligence

A Modern Approach

Fourth Edition

Stuart J. Russel and Peter Norvig



Pearson

Искусственный интеллект
Современный подход

Четвертое издание

Том I.
Решение проблем: знания и рассуждения

С. Рассел, П. Норвиг

Москва · Санкт-Петербург
2021

ББК 32.813
Р24
УДК 004.8

ООО "Диалектика"

Перевод с английского и редакция *А.В. Слепцова*

По общим вопросам обращайтесь в издательство "Диалектика"
по адресу: info.dialektika@gmail.com, http://www.dialektika.com

Рассел, Стюарт, **Норвиг**, Питер.

Р24 Искусственный интеллект: современный подход, 4-е изд., том 1. Решение проблем: знания и рассуждения. : Пер. с англ. — СПб. : ООО "Диалектика", 2021. — 704 с. — Парал. тит. англ.

ISBN 978-5-907365-25-4 (рус., том 1)

ISBN 978-5-907365-24-7 (рус., многотом.)

ББК 32.813

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

Никакая часть настоящего издания ни в каких целях не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами, будь то электронные или механические, включая фотокопирование и запись на магнитный носитель, если на это нет письменного разрешения издательства Pearson Education, Inc.

Copyright © 2021 by Dialektika Computer Publishing.

Original English edition Copyright © 2021 by Pearson Education, Inc.

All rights reserved including the right of reproduction in whole or in part in any form. This translation is published by arrangement with Pearson Education, Inc.

Научно-популярное издание

Стюарт Рассел, Питер Норвиг

Искусственный интеллект: современный подход

4-е издание, том 1

Решение проблем: знания и рассуждения

Подписано в печать 24.03.2021. Формат 70x100/16
Усл. печ. л. 56,8. Уч.-изд. л. 53,3
Тираж 500 экз. Заказ № 2850.

Отпечатано в АО "Первая Образцовая типография"
Филиал "Чеховский Печатный Двор"
142300, Московская область, г. Чехов, ул. Полиграфистов, д. 1
Сайт: www.chpd.ru, E-mail: sales@chpd.ru, тел. 8 (499) 270-73-59

ООО "Диалектика", 195027, Санкт-Петербург, Магнитогорская ул., д. 30, лит. А, пом. 848

ISBN 978-5-907365-25-4 (рус., том 1)
ISBN 978-5-907365-24-7 (рус., многотом.)
ISBN 978-0-13-461099-3 (англ.)

© ООО "Диалектика", 2021, перевод, оформление, макетирование

© 2021, 2010, 2003 by Pearson Education, Inc.

Оглавление

Предисловие 13

Об авторах 19

Часть I. ИСКУССТВЕННЫЙ ИНТЕЛЛЕКТ, ОСНОВЫ

Глава 1. Введение 23

Глава 2. Интеллектуальные агенты 81

Часть II. РЕШЕНИЕ ЗАДАЧ

Глава 3. Решение задач посредством поиска 125

Глава 4. Поиск в сложных средах 205

Глава 5. Поиск в условиях противодействия и игры 261

Глава 6. Задачи удовлетворения ограничений 319

Часть III. ЗНАНИЯ, РАССУЖДЕНИЯ И ПЛАНИРОВАНИЕ

Глава 7. Логические агенты 369

Глава 8. Логика первого порядка 441

Глава 9. Логический вывод в логике первого порядка 493

Глава 10. Представление знаний 551

Глава 11. Автоматизированное планирование 603

Приложение А. Математические основы 673

Приложение Б. Сведения о языках и алгоритмах, используемых в книге 684

Предметный указатель 689

Содержание

Предисловие 13
Новое в этом издании 13
Общий обзор книги 14
Ресурсы в Интернете 16
Обложка книги 16
Благодарности 17

Об авторах 19
Ждем ваших отзывов! 20

Часть I. ИСКУССТВЕННЫЙ ИНТЕЛЛЕКТ, ОСНОВЫ

Глава 1. Введение 23
1.1. Что такое ИИ 24
1.2 Истоки искусственного интеллекта 30
1.3. История искусственного интеллекта 48
1.4. Современное состояние исследований 63
1.5. Риски и преимущества искусственного интеллекта 69
Резюме 75
Библиографические и исторические заметки 76
Упражнения 77

Глава 2. Интеллектуальные агенты 81
2.1. Агенты и среды 81
2.2. Лучшее поведение: концепция рациональности 85
2.3. Свойства окружающей среды 90
2.4. Структура агентов 97
Резюме 116
Библиографические и исторические заметки 117
Упражнения 119

Часть II. РЕШЕНИЕ ЗАДАЧ

Глава 3. Решение задач посредством поиска 125
3.1. Агенты, решающие задачи 125
3.2. Примеры задач 130
3.3. Алгоритмы поиска 137

3.4. Стратегии неинформированного поиска 144
3.5. Стратегии информированного (эвристического) поиска 155
3.6. Эвристические функции 176
Резюме 187
Библиографические и исторические заметки 189
Упражнения 194

Глава 4. Поиск в сложных средах 205
4.1. Локальный поиск и задачи оптимизации 205
4.2. Локальный поиск в непрерывных пространствах 218
4.3. Поиск с недетерминированными действиями 222
4.4. Поиск в частично наблюдаемых средах 228
4.5. Поисковые агенты, действующие в оперативном режиме, и неизвестные варианты среды 240
Резюме 250
Библиографические и исторические заметки 251
Упражнения 256

Глава 5. Поиск в условиях противодействия и игры 261
5.1. Теория игр 261
5.2. Принятие оптимальных решений в играх 265
5.3. Эвристический альфа-бета-поиск по дереву 275
5.4. Поиск по дереву методом Монте-Карло 283
5.5. Игры с элементами случайности 289
5.6. Частично наблюдаемые игры 293
5.7. Ограничения игровых алгоритмов поиска 300
Резюме 302
Библиографические и исторические заметки 304
Упражнения 311

Глава 6. Задачи удовлетворения ограничений 319
6.1. Определение задач удовлетворения ограничений 320
6.2. Распространение ограничений: логический вывод в CSP 328
6.3. Поиск с возвратами в задачах удовлетворения ограничений 337
6.4. Локальный поиск в задачах удовлетворения ограничений 346
6.5. Структура задач 349
Резюме 356
Библиографические и исторические заметки 357
Упражнения 362

Часть III. ЗНАНИЯ, РАССУЖДЕНИЯ И ПЛАНИРОВАНИЕ

Глава 7. Логические агенты 369
7.1. Агенты, основанные на знаниях 370
7.2. Мир вампуса 373
7.3. Логика 377
7.4. Логика высказываний: очень простая логика 382
7.5. Доказательство теорем логики высказываний 390
7.6. Эффективный пропозициональный логический вывод 406
7.7. Агенты, основанные на логике высказываний 414
Резюме 428
Библиографические и исторические заметки 429
Упражнения 434

Глава 8. Логика первого порядка 441
8.1. Еще раз о представлении 441
8.2. Синтаксис и семантика логики первого порядка 449
8.3. Использование логики первого порядка 463
8.4. Инженерия знаний на основе логики первого порядка 472
Резюме 481
Библиографические и исторические заметки 482
Упражнения 484

Глава 9. Логический вывод в логике первого порядка 493
9.1. Логический вывод в логике высказываний и логике первого порядка 493
9.2. Унификация и логический вывод в логике первого порядка 496
9.3. Прямой логический вывод 503
9.4. Обратный логический вывод 513
9.5. Резолюция 521
Резюме 538
Библиографические и исторические заметки 539
Упражнения 544

Глава 10. Представление знаний 551
10.1. Онтологическая инженерия 551
10.2. Категории и объекты 555
10.3. События 564
10.4. Ментальные объекты и модальная логика 569
10.5. Системы рассуждений о категориях 573

10.6. Рассуждения при наличии информации по умолчанию 579
Резюме 586
Библиографические и исторические заметки 587
Упражнения 595

Глава 11. Автоматизированное планирование 603
11.1. Определение классической задачи планирования 603
11.2. Алгоритмы классического планирования 609
11.3. Эвристики для задач планирования 616
11.4. Иерархическое планирование 621
11.5. Планирование и действие в недетерминированных проблемных областях 634
11.6. Время, расписания и ресурсы 648
11.7. Анализ различных подходов к планированию 654
Резюме 655
Библиографические и исторические заметки 656
Упражнения 664

Приложение А. Математические основы 673
А.1. Анализ сложности и нотация O() 673
А.2. Векторы, матрицы и линейная алгебра 677
А.3. Распределения вероятностей 679
Библиографические и исторические заметки 683

Приложение Б. Сведения о языках и алгоритмах, используемых в книге 684
Б.1. Определение языков с помощью формы Бэкуса–Наура 684
Б.2. Описание алгоритмов с помощью псевдокода 685
Б.3. Дополнительный материал в Интернете 687

Предметный указатель 689

Посвящается Лой, Гордону, Люси, Джорджу и Исааку.

С.Дж. Рассел

Посвящается Крис, Изабелле и Джульетте.

П. Норвиг

Предисловие

Искусственный интеллект (ИИ) — чрезвычайно широкая область знаний, и именно поэтому данная книга имеет такой большой объем. Авторы описали эту область во всей ее широте, включая математическую логику, теорию вероятностей и теорию непрерывных функций. Они раскрыли суть таких понятий, как восприятие, рассуждение, обучение и действие, обсудили важные социальные аспекты (справедливость, доверие, общественное благо и безопасность), а также описали использование достижений в этой области в самом широком диапазоне, начиная с микроэлектронных устройств и заканчивая роботизированными средствами исследования других планет и глобальными интерактивными системами информационного обслуживания миллионов пользователей.

Подзаголовок этой книги — "Современный подход". Это означает, что авторы решили освещать историю проводившихся исследований с современной точки зрения, синтезируя то, что известно сейчас, в общие рамки и пересматривая ранние работы с использованием идей и терминологии, общепринятой в наши дни. Авторы приносят свои извинения представителям тех направлений, которые по этой причине стали выглядеть не столь значимыми.

Новое в этом издании

В этом, четвертом, издании книги отражены те изменения в области искусственного интеллекта, которые произошли со времени выпуска предыдущего, третьего, издания книги в 2010 году.

- Теперь мы в большей степени ориентируемся на машинное обучение, чем на подготовку знаний вручную, что вызвано существенно возросшей доступностью данных и вычислительных ресурсов, а также появлением новых алгоритмов.
- Глубокое обучение, вероятностное программирование и многоагентные системы теперь рассматриваются более широко — каждой из этих областей ИИ посвящена отдельная глава.
- Подход к изложению тем понимания естественного языка, робототехники и компьютерного зрения был существенно пересмотрен с целью отразить влияние последних достижений в области глубокого обучения.
- В главу о робототехнике добавлены разделы о роботах, взаимодействующих с людьми, и применении к роботам методов обучения с подкреплением.
- Ранее целью ИИ полагалось создание систем, направленных на максимизацию их полезности, причем конкретная информация о самой этой полезности — т.е. цель — предоставлялась человеком, создававшим систему. Теперь

авторы больше не предполагают, что цель всегда заранее установлена и известна системе ИИ. Вместо этого допускается, что система может не иметь уверенности в истинных целях тех людей, от имени которых она действует. Ей потребуется выяснить, что необходимо максимизировать, а затем она должна будет функционировать соответствующим образом, даже если у нее не будет полной уверенности в конечной цели.

- Расширен охват анализа влияния ИИ на общество, включая жизненно важные вопросы этики, справедливости, доверия и безопасности.
- В оригинальном англоязычном издании авторы перенесли упражнения из конца каждой главы на специально созданный веб-сайт, чтобы получить возможность постоянно добавлять, обновлять и улучшать упражнения, стремясь удовлетворять меняющиеся потребности преподавателей и полнее отражать достижения в области ИИ и в связанных с ней инструментах программного обеспечения.

  В этом русскоязычном издании имеется ссылка на данный англоязычный сайт, а те упражнения, которые на нем присутствовали на момент перевода книги, также были переведены и включены в нее на прежнем месте — в конце каждой главы. Это было сделано для удобства русскоязычных читателей книги. Те же из них, кто в достаточной степени владеет английским языком, имеют возможность в любой момент обратиться к сайту авторов и воспользоваться новым материалом, который будет на нем появляться с течением времени.
- В целом можно сказать, что в этом издании примерно 25% материала книги совершенно новые. Остальные 75% были в значительной степени переработаны с целью представить читателю более цельную, единообразную картину всей охватываемой в ней области знаний. Также отметим, что 22% цитат, приведенных в этом издании, взяты из работ, опубликованных уже после 2010 года.

Общий обзор книги

Главной объединяющей темой этой книги является идея **интеллектуального агента**. Авторы определяют искусственный интеллект как науку об агентах, которые получают результаты актов восприятия из окружающей среды и выполняют действия. Каждый такой агент реализует функцию, которая отображает последовательности актов восприятия в действия, и в данной книге рассматриваются различные средства представления этих функций, такие как реактивные агенты, планировщики в реальном масштабе времени, системы, действующие на основе теории принятия решений, и системы глубокого обучения. Авторы трактуют обучение и как метод создания компетентных систем, и как способ расширения пределов досягаемости разработчика в неизвестной среде. Кроме того, авторы

рассматривают робототехнику и системы искусственного зрения не как независимо определяемые научные направления, а как области знаний, позволяющие обеспечить более успешное достижение целей, стоящих перед агентами, и подчеркивают важность учета особенностей окружающей среды при определении соответствующей конструкции агента.

Наша главная цель — изложить *идеи*, которые были сформулированы в исследованиях по искусственному интеллекту, проводившихся в течение последних семидесяти лет, а также собраны на протяжении последних двух тысячелетий в смежных областях знаний. Мы старались избегать чрезмерного формализма при изложении этих идей и при этом сохранять необходимую точность. Чтобы конкретизировать излагаемые идеи, в текст были включены математические формулы и алгоритмы на псевдокоде. Необходимые сведения об основных математических концепциях и принятых соглашениях в отношении нотации приводятся в приложении А. Особенности используемого в книге псевдокода описываются в приложении Б.

Эта книга, в первую очередь, предназначена для использования в курсе или серии курсов бакалавриата. Она включает 28 глав,[1] на каждую из которых потребуется выделить около недели лекций, поэтому проработка всего материала книги предполагает не менее двух семестров. В курсах, рассчитанных на один семестр, возможно использование лишь части всех глав, отобранных так, чтобы удовлетворить интересы преподавателя и студентов. Книга также может послужить основой курса для выпускников (возможно, с добавлением некоторых первоисточников, предлагаемых в библиографии) либо использоваться для самостоятельного изучения или же в качестве справочного пособия.

Везде в книге ➔ *важные замечания* отмечены данным значком в виде стрелки и выделены особым курсивным написанием. Также в тех местах, где дается *определение* ▶ **нового термина**, перед ним ставится значок в виде треугольника, а сам этот термин выделяется полужирным написанием. Впоследствии там, где упоминание этого термина также будет иметь важное значение, он будет выделен полужирным написанием, но треугольного значка перед ним уже не будет. В книгу включен предметный указатель со ссылками на все важные определения и термины, приводимые в книге, а во втором томе представлен обширный список дополнительной литературы.

Для работы с книгой единственным необходимым условием является знакомство с основными понятиями компьютерных наук (алгоритм, структура данных, сложность) на уровне студента-второкурсника. Для некоторых тем также полезно владеть основами интегрального и дифференциального исчисления и линейной алгебры на уровне студента первого курса.

---

[1] В русскоязычном издании они распределены по трем томам: том I — главы 1–11, том II — главы 12–18, том III — главы 19–28. Библиография оформлена в виде отдельного файла, размещенного на сайте издательства по адресу: `http://www.williams-publishing.com/Books/978-5-907365-25-4.html`.

Ресурсы в Интернете

Интерактивные ресурсы книги доступны через сайт издательства Pearson по адресу:
`pearsonhighered.com/cs-resources`
либо непосредственно на веб-сайте книги, расположенном по адресу:
`aima.cs.berkeley.edu`
Там вы найдете следующее.

- Упражнения, программные проекты и исследовательские проекты. Как уже упоминалось, в англоязычном оригинале книги упражнений в конце каждой главы больше нет — в этом издании они доступны только на веб-сайте книги. В русском переводе книги они получены с этого сайта в том виде, в котором находились на нем в момент перевода, и помещены, как обычно, в конце соответствующей главы.
- Реализации всех приведенных в книге алгоритмов на Python, Java и некоторых других языках программирования. (В настоящее время эти программные коды доступны на сайте `github.com/aimacode`.)
- Список более чем из 1400 учебных заведений, в которых использовалась данная книга. Во многих случаях предоставляются ссылки на материалы и программы собственных интерактивных курсов.
- Дополнительные материалы и ссылки для студентов и преподавателей.
- Инструкции о том, как сообщать об ошибках, обнаруженных в книге, если они будут найдены.

Обложка книги

На обложке представлена финальная позиция шестой, и решающей, игры шахматного матча 1997 года, в котором программа Deep Blue победила Гарри Каспарова (он играл черными). В этом матче компьютер впервые добился победы над чемпионом мира в шахматном матче. Фото Каспарова находится на обложке вверху справа. Там же показана ключевая позиция во второй игре исторического матча по игре в го между бывшим чемпионом мира Ли Седолом и программой Deep Mind's ALPHA GO. Ход 37 программы ALPHAGO нарушил вековые ортодоксальные традиции игры в го и немедленно был воспринят экспертами-людьми как досадная ошибка, но именно он оказался решающим в достижении выигрыша. Вверху слева находится фото Атласа — человекоподобного робота, созданного компанией Boston Dynamics. Изображение автомобиля с автоматическим управлением, анализирующего окружающую среду, размещено между портретом Ады Лавлейс, первого в мире программиста, и фотографией Алана Тьюринга, фундаментальная работа которого стала краеугольным

камнем в теории искусственного интеллекта. В нижней части шахматной доски находятся фото робота-планетохода Mars Exploration Rover и бюста Аристотеля, который первым начал изучать логику. Под шахматной доской помещен код вероятностной модели программирования, используемой Организацией договора о всеобъемлющем запрещении ядерных испытаний при ООН для обнаружения ядерных взрывов по сейсмическим сигналам.

Благодарности

Чтобы выпустить в свет эту книгу, потребовалась помощь множества людей по всему миру. Более 600 человек прочитали отдельные фрагменты книги и предоставили авторам свои предложения по их улучшению. Полный список всех наших помощников можно найти по адресу `aima.cs.berkeley.edu/ack.html`. Мы глубоко благодарны им всем. Здесь у нас есть место лишь для того, чтобы упомянуть только некоторых из них, тех, чей вклад был особенно важен для нас. Сначала упомянем авторов отдельных разделов.

- Джуди Перл (Judea Pearl) — раздел 13.5, *Каузальные сети.*
- Викаш Мансингха (Vikash Mansinghka) раздел 15.4, *Программы как вероятностные модели.*
- Майкл Уолдридж (Michael Wooldridge) — глава 18, *Принятие решений при наличии нескольких агентов.*
- Ян Гудфеллоу (Ian Goodfellow) — глава 21, *Глубокое обучение.*
- Джейкоб Девлин (Jacob Devlin) и Мей-Вин Чанг (Mei-Wing Chang) — глава 24, *Глубокое обучение при обработке естественного языка.*
- Джитендра Малик (Jitendra Malik) и Дэвид Форсайт (David Forsyth) — глава 25, *Компьютерное зрение.*
- Анка Драган (Anca Dragan) — глава 26, *Робототехника.*

А теперь упомянем тех, кто сыграл в создании этой книги ключевую роль.

- Синтия Юнг (Cynthia Yeung) и Малика Кантор (Malika Cantor) — управление проектом.
- Джулия Суссман (Julie Sussman) и Том Галлоуэй (Tom Galloway) — набор и редактирование текста.
- Омари Стивенс (Omari Stephens) — иллюстрации.
- Трейси Джонсон (Tracy Johnson) — выпускающий редактор.
- Эрин Олт (Erin Ault) и Роуз Кернан (Rose Kernan) — обложка и цветоделение.
- Налин Чиббер (Nalin Chhibber), Сэм Гото (Sam Goto), Раймон де Лаказ (Raymond de Lacaze), Рави Мохан (Ravi Mohan), Кьяран О'Рейли (Ciaran O'Reilly), Амит Патель (Amit Patel), Драгомир Радив (Dragomir Radiv) и

Самагра Шарма (Samagra Sharma) — интерактивная разработка кода и наставничество.

- Студенты курса Google Summer of Code — интерактивная разработка кода.

Стюарт хотел бы поблагодарить свою жену, Лой Шефлотт (Loy Sheflott), за бесконечное терпение и безграничную мудрость. Он надеется, что Гордон, Люси, Джордж и Исаак со временем прочтут эту книгу, простив его за столь долгую работу над ней. И как всегда, исключительно полезной для него была работа с RUGS (Russell's Unusual Group of Students — необыкновенная группа студентов Рассела).

Питер хотел бы поблагодарить своих родителей, Торстена и Герду, за помощь на первых порах, свою жену, Крис, детей Беллу и Джульетту, коллег, друзей и руководство за то, что подбадривали его и проявляли понимание и терпение в те долгие часы, когда он писал и переписывал эту книгу.

Об авторах

Стюарт Рассел родился в 1962 году в г. Портсмут, Англия. Он получил степень бакалавра по физике с наградами первой степени в Оксфордском университете в 1982 году и степень доктора философии по компьютерным наукам в Станфордском университете в 1986 году. Затем он перешел на факультет Калифорнийского университета в г. Беркли, где занял должность профессора на кафедре компьютерных наук, которую возглавлял некоторое время, а также директора центра "ИИ, совместимый с человеком" и заведующего кафедрой инженерного искусства Смита–Задэ. В 1990 году он получил Президентскую премию молодых исследователей Национального научного фонда, а в 1995 стал лауреатом премии "Компьютеры и мышление". Рассел является членом Американской ассоциации искусственного интеллекта, Ассоциации вычислительной техники и Американской ассоциации развития науки, почетным членом Вадхам-колледжа в Оксфорде и членом Общества Эндрю Карнеги, занимал почетное кресло Блеза Паскаля в Париже с 2012 по 2014 год. Стюарт опубликовал более 300 статей по широкому кругу тем в области искусственного интеллекта. Среди прочих написанных им книг можно упомянуть *The Use of Knowledge in Analogy and Induction*, *Do the Right Thing: Studies in Limited Rationality* (в соавторстве с Эриком Вифольдом) и *Human Compatible: Artificial Intelligence and the Problem of Control*.

Питер Норвиг в настоящее время является директором по исследованиям в компании Google, Inc., а ранее занимал в ней пост директора подразделения, отвечающего за основные алгоритмы веб-поиска. Он был одним из преподавателей в интерактивном курсе ИИ, на который подписалось более 160 тысяч студентов, и помогал начать текущий раунд массовых открытых интерактивных курсов. В свое время он был руководителем отдела вычислительных наук в Исследовательском центре Эймса, НАСА, где курировал исследования и разработки в области искусственного интеллекта и робототехники. Норвиг получил степень бакалавра прикладной математики в Университете Брауна и степень доктора философии в области компьютерных наук в Беркли. Он был профессором в Университете Южной Калифорнии и членом совета факультета в Беркли и Станфорде. Питер Норвиг является членом Американской ассоциации искусственного интеллекта, Ассоциации вычислительной техники, Американской академии искусств и наук и Калифорнийской академии наук. Кроме того, он является автором книг *Paradigms of AI Programming: Case Studies in Common Lisp*, *Verbmobil: A Translation System for Face-to-Face Dialog* и *Intelligent Help Systems for UNIX*.

Два автора этой книги разделили первую награду AAAI/EAAI "Выдающийся педагог" в 2016 году.

Ждем ваших отзывов!

Вы, читатель этой книги, и есть главный ее критик. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересны любые ваши замечания в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам электронное письмо либо просто посетить наш веб-сайт и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится ли вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Отправляя письмо или сообщение, не забудьте указать название книги и ее авторов, а также свой обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию новых книг.

Наши электронные адреса:

E-mail: `info.dialektika@gmail.com`
WWW: `http://www.dialektika.com`

Часть I

Искусственный интеллект, основы

Введение 23

Интеллектуальные агенты 81

ГЛАВА 1

Введение

В этой главе авторы предпринимают попытку объяснить, почему они рассматривают искусственный интеллект как предмет, в наибольшей степени заслуживающий изучения, а также определить, в чем именно он заключается, — это необходимо сделать до того, как можно будет отправиться дальше.

Мы называем себя Homo Sapiens — *человек разумный*, — потому что наш ▶**интеллект**, наши умственные способности столь для нас важны. На протяжении тысячелетий люди пытались понять, *как мы думаем и действуем*, т.е. разобраться в том, как наш мозг, всего лишь небольшая горсточка материи, может ощущать, понимать, предсказывать и манипулировать миром, который несравнимо больше в размерах и сложнее, чем он сам. Область ▶**искусственного интеллекта**, или ИИ, охватывает не только понимание всего того, о чем говорилось выше, но и создание интеллектуальных сущностей — машин, которые будут способны вычислять, как им действовать эффективно и безопасно в самых разнообразных, в том числе незнакомых им, ситуациях.

Регулярно проводимые исследования свидетельствуют о том, что область ИИ расценивается как одна из самых интересных и наиболее быстро развивающихся областей науки и техники. Уже сейчас она приносит годовой доход размером более триллиона долларов. Эксперт по искусственному интеллекту Кай-Фу Ли предсказывает, что ее влияние будет "больше, чем что-либо иное в истории человечества". Более того, интеллектуальные границы ИИ широко открыты. В то время как студенты, изучающие традиционные науки, такие как физика, могут полагать, что лучшие идеи в этой области уже были выдвинуты Галилеем, Ньютоном, Кюри, Эйнштейном и другими, они осознают, что в области ИИ еще достаточно простора для выдающихся открытий.

Тематика области искусственного интеллекта в настоящее время охватывает огромный перечень научных направлений, от задач самого общего характера (обучение, рассуждение, восприятие и т.д.) и до таких конкретных задач, как игра в шахматы, доказательство математических теорем, сочинение стихов, вождение автомобиля или диагностика заболеваний. Достижения в области ИИ могут найти себе применение при решении любой интеллектуальной задачи, — это универсальная научная область.

1.1. Что такое ИИ

Выше мы уже заявили, что область ИИ вызывает большой интерес, но пока еще не пояснили, что же она собой представляет. Исторически сложилось так, что исследователи рассматривали несколько различных версий ИИ. Одни давали определение интеллекту с точки зрения соответствия поведению человека, в то время как другие предпочитали использовать абстрактное, формальное определение интеллекта, получившее название ▶**рациональность** — в широком смысле это способность "поступать правильно". Сам предмет также воспринимается по-разному: одни считают, что интеллект является свойством внутренних *мыслительных процессов* и *рассуждений*, в то время как другие фокусируются на интеллектуальном *поведении*, т.е. на внешней характеристике.[1]

Из этих двух противопоставлений — *человекоподобность–рациональность*[2] и *мышление–поведение* можно вывести четыре различные попарные комбинации, и у каждой из них будут свои приверженцы и соответствующие исследовательские программы. Используемые в них методы были по необходимости различными: поиски человекоподобного интеллекта должны были проводиться в рамках эмпирических наук, связанных с психологией, включая наблюдение и гипотезы о фактическом человеческом поведении и мыслительных процессах. С другой стороны, рационалистический подход предполагает некий синтез математики и техники, с привлечением статистики, теории управления и экономики. Группы исследователей, следовавшие различными путями, могли как проявлять пренебрежение, так и помогать друг другу. Давайте рассмотрим все четыре подхода более подробно.

1.1.1. Действуя, как человек: подход на основе теста Тьюринга

▶**Тест Тьюринга**, предложенный Аланом Тьюрингом в 1950 году, был разработан как мысленный эксперимент, который позволил бы обойти философскую неясность вопроса "Может ли машина мыслить?" Компьютер пройдет этот тест, если человек-испытатель, направив ему несколько письменных вопросов, в конечном итоге не сможет определить, от кого исходят полученные им письменные ответы — от человека или от компьютера. В главе 27 подробно обсуждается этот тест и рассматривается вопрос о том, действительно ли можно считать интеллектуальным компьютер, который успешно прошел подобный тест. На данный

[1] Не следует смешивать понятия "искусственный интеллект" и "машинное обучение". Машинное обучение — это область ИИ, в которой изучается способность улучшать свои навыки на основе опыта. В одних системах искусственного интеллекта используются методы машинного обучения для достижения необходимого уровня знаний, а в других этот подход не используется.

[2] Мы не предполагаем, что люди "иррациональны" в буквальном смысле этого слова, т.е. "лишены нормальной ясности ума". Мы просто допускаем, что человеческие решения не всегда безупречны с точки зрения математики.

момент просто отметим, что программирование компьютера для прохождения этого теста в строгом соответствии с исходными требованиями потребует очень большого объема работы. Запрограммированный таким образом компьютер должен обладать всеми перечисленными ниже возможностями.

- ▶**Обработка естественного языка** для успешного общения с человеком на его языке.
- ▶**Представление знаний** для успешного сохранения того, что он узнает или услышит.
- ▶**Автоматические рассуждения** для ответа на вопросы и вывода новых заключений.
- ▶**Машинное обучение** для адаптации к новым обстоятельствам, а также для выявления и экстраполяции моделей.

Сам Тьюринг полагал, что для демонстрации искусственного интеллекта нет необходимости в физической имитации человека. Однако другие исследователи с этим не согласились и предложили ▶**общий тест Тьюринга**, для прохождения которого необходимо продемонстрировать взаимодействие с объектами и людьми в реальном мире. Чтобы пройти полный тест Тьюринга, роботу дополнительно понадобятся следующие способности.

- ▶**Компьютерное зрение** и распознавание речи для восприятия реального мира.
- ▶**Робототехника** для манипулирования объектами и перемещения в пространстве.

Эти шесть перечисленных выше направлений составляют основную часть области исследований ИИ. Тем не менее исследователи искусственного интеллекта практически не занимаются решением задачи прохождения теста Тьюринга, считая, что гораздо важнее изучить основополагающие принципы интеллекта. И действительно, проблему "искусственного полета" удалось успешно решить лишь после того, как инженеры и изобретатели перестали имитировать птиц и приступили к изучению аэродинамики. В научных и технических работах по воздухоплаванию цель этой области знаний не определяется как "создание машин, которые в своем полете настолько напоминают голубей, что даже могут обмануть настоящих птиц этого вида".

1.1.2. Думая, как человек: подход когнитивного моделирования

Чтобы сказать, что программа мыслит, как человек, мы должны знать, как люди думают. Мы можем изучать человеческое мышление тремя способами.

- ▶**Самоанализ** — попытки поймать наши собственные мысли, когда они приходят в наше сознание.

- ▶**Психологические эксперименты** — наблюдение за человеком в действии.
- ▶**Визуализация работы мозга** — наблюдение за мозгом в действии.

Если у нас появится достаточно точная теория работы сознания, станет возможным выразить эту теорию в виде компьютерной программы. Если поведение системы ввода-вывода программы соответствует действительному поведению человека, это свидетельствует о том, что и некоторые механизмы программы могут работать, как у людей.

Например, Аллен Ньюэлл (Allen Newell) и Герберт Саймон (Herbert Simon), которые разработали программу GPS (General Problem Solver — универсальный решатель задач) ([1668], 1961), не стремились лишь к тому, чтобы эта программа правильно решала поставленные задачи. Они в большей степени заботились о том, чтобы запись этапов проводимых ею рассуждений совпадала с регистрацией рассуждений людей, решающих такие же задачи. Междисциплинарная область ▶**когнитивной науки** объединяет компьютерные модели из ИИ и экспериментальные методы из психологии для построения точных, позволяющих выполнить их проверку теорий человеческого разума.

Когнитивная наука сама по себе является увлекательной областью, достойной нескольких учебников и по крайней мере одной энциклопедии ([2354], 1999). Время от времени мы будем комментировать сходства или различия между методами искусственного интеллекта и человеческим познанием. Однако реальная когнитивная наука по необходимости строится на основе экспериментальных исследований реальных людей или животных. Мы оставим обсуждение этого аспекта для других книг, поскольку предполагаем, что для проведения экспериментов читатель располагает только компьютером.

На ранних этапах исследований ИИ часто возникала путаница между разными подходами. Автор мог утверждать, что алгоритм хорошо справляется с заданием и, *следовательно*, является хорошей моделью человеческой деятельности, или наоборот. Современные авторы разделяют эти два вида претензий; это различение позволяет как ИИ, так и когнитивной науке развиваться более быстрыми темпами. Эти две области исследований часто оплодотворяют друг друга, что наиболее заметно в компьютерном зрении, где результаты нейрофизиологических исследований используются при построении вычислительных моделей. В последнее время комбинирование методов нейровизуализации с технологиями машинного обучения с целью анализа собираемых данных уже привело к появлению возможности "читать мысли", т.е. к возможности определения семантического содержания мыслей в сознании человека. Эта способность, в свою очередь, могла бы пролить дополнительный свет на то, как работает человеческое познание.

1.1.3. Думая рационально: подход на основе "законов мышления"

Древнегреческий философ Аристотель был одним из первых, кто попытался определить законы "правильного мышления", т.е. процессы формирования неопровержимых рассуждений. Его ▶**силлогизмы** стали образцом для создания процедур доказательства, которые всегда позволяют прийти к правильным заключениям, если даны правильные предпосылки. Вот канонический пример таких рассуждений: "Сократ — человек; все люди смертны; следовательно, Сократ смертен". (Этот пример, вероятно, скорее связан с Секстом Эмпириком, чем с Аристотелем.) Предполагалось, что эти законы мышления управляют работой ума; их исследование положило начало научному направлению, называемому **логикой**.

В XIX веке ученые-логики создали точную систему логических обозначений для утверждений о предметах любого рода, встречающихся в мире, и об отношениях между ними. (Сравните это с обычной системой арифметических обозначений, которая предназначена в основном для формирования утверждений о равенстве и неравенстве *чисел*.) К 1965 году уже были разработаны программы, которые в принципе могли решить любую разрешимую проблему, описанную в системе логических обозначений. Исследователи в области искусственного интеллекта, придерживающиеся так называемых традиций ▶**логицизма**, надеются, что им удастся создать интеллектуальные системы на основе подобных программ.

Логика, как она обычно понимается, требует, чтобы знания о мире, которыми она оперирует, были *точными* — условие, которое в действительности редко достижимо. Мы просто не знаем правил, которые действуют, скажем, в политике или на войне, с той же степенью достоверности, как правила арифметики или игры в шахматы. ▶**Теория вероятности** заполняет этот пробел, позволяя проводить строгие рассуждения с неточной информацией. В принципе, это позволяет построить всеобъемлющую модель рационального мышления, которая обеспечит переход от необработанной субъективно воспринимаемой информации к пониманию того, как устроен мир и даже к предсказаниям о будущем. Но этой модели все же недостаточно для генерирования разумного *поведения*. Для этого нам потребуется еще и теория рационального действия — рационального мышления самого по себе нам будет недостаточно.

1.1.4. Действуя рационально: подход с использованием рационального агента

▶**Агент** — это просто что-то, что действует (слово *агент* произошло от латинского слова *agere* — "действовать"). Конечно, все компьютерные программы что-то делают, но ожидается, что компьютерные агенты будут делать больше: работать автономно, воспринимать окружающую среду, сохранять свое существование в течение длительного периода времени, приспосабливаться к изменениям,

устанавливать и преследовать определенные цели. ▶**Рациональным агентом** называется агент, действующий таким образом, чтобы достичь наилучшего результата или, если он находится в условиях неопределенности, наилучшего ожидаемого результата.

В подходе к созданию ИИ на основе "законов мышления" акцент был сделан на формировании правильных логических выводов. Безусловно, иногда формирование правильных логических выводов становится *частью* функционирования и рационального агента, поскольку один из способов рациональной организации своих действий состоит в том, чтобы логическим путем прийти к заключению, что данное конкретное действие позволяет достичь указанных целей, а затем действовать в соответствии с принятым решением. С другой стороны, существуют способы действовать рационально, о которых нельзя сказать, что они предполагают логический вывод. Например, отдергивание пальца от горячей печи — это рефлекторное действие, которое обычно является более успешным в сравнении с более медленным действием, предпринятым после тщательного обдумывания ситуации.

Все навыки, необходимые для теста Тьюринга, также позволяют агенту действовать рационально. Представление знаний и рассуждения обеспечат агенту возможность принимать правильные решения. Мы должны быть в состоянии генерировать понятные фразы на естественном языке, чтобы войти в состав сложного социума. Нам нужно учиться не только для эрудиции, но и для развития способности генерировать эффективное поведение, особенно в новых условиях.

Подход к созданию ИИ с использованием рациональных агентов имеет два важных преимущества в сравнении с другими подходами. Во-первых, он более общий, чем подход на основе "законов мышления", — правильный вывод является лишь одним из нескольких возможных механизмов достижения рациональности. Во-вторых, он лучше поддается научному развитию. Стандарт рациональности в самом общем виде хорошо определен математически. Во многих случаях можно исходить из этой спецификации, чтобы получить проект агента, который доказуемо достигнет цели, что почти невозможно, если цель состоит в том, чтобы имитировать человеческое поведение или процессы мышления.

По этим причинам подход к созданию ИИ с использованием рациональных агентов преобладал на протяжении большей части истории проведения исследований в этой области. В первые десятилетия рациональные агенты строились на логических основах и формировали определенные планы для достижения конкретных целей. Позже методы, основанные на теории вероятностей и технологии машинного обучения, позволили создавать агентов, которые были способны принимать решения в условиях неопределенности, имея целью достижение наилучшего ожидаемого результата. Сказанное можно обобщить следующим образом: ➔ *большинство работ в области ИИ фокусировалось, прежде всего, на изучении и создании агентов, способных* ▶ ***поступать правильно***. Что считать правильным, определялось целью, которая ставилась перед агентом. Эта общая парадигма настолько распространена, что ее вполне правомерно назвать ▶**стандартной моделью**. Она превалирует не только в области

ИИ, но и в теории управления, где регулятор минимизирует стоимостную функцию, в исследовании операций, где линия поведения максимизирует целевую функцию, в статистике, где правило принятия решения минимизирует функцию потерь, и в экономике, где лицо, принимающее решения, максимизирует полезность или некоторую меру социального обеспечения.

В отношении стандартной модели необходимо сделать одно важное уточнение: следует учесть тот факт, что идеальная рациональность — всегда выбирать именно оптимальное действие — не всегда достижима в сложных условиях. Например, требования к вычислительным ресурсам могут оказаться слишком высокими. В главах 5 и 17 будет обсуждаться вопрос ▶**ограниченной рациональности** — как поступить надлежащим образом, если не хватает времени на проведение всех необходимых вычислений. Тем не менее идеальная рациональность часто остается хорошей отправной точкой для проведения теоретического анализа.

1.1.5. Полезные машины

Стандартная модель была полезным ориентиром для исследований в области ИИ с самого начала, но в долгосрочной перспективе она, вероятно, уже не будет настолько подходящей. Причина в том, что стандартная модель предполагает, что машине всегда ставится точно определенная цель.

Для искусственно определенных задач, таких как игра в шахматы или нахождение кратчайшего пути, задача формулируется с изначально определенной конкретной целью, поэтому стандартная модель здесь будет вполне применима. Однако по мере приближения к реальному миру становится все труднее и труднее определить конечную цель точно и полностью. Например, при проектировании самоуправляемого автомобиля изначально можно полагать, что цель состоит лишь в том, чтобы безопасно достичь пункта назначения. Однако движение по любой дороге сопряжено с риском получения травмы из-за других движущихся по ней автомобилей, отказа оборудования и т.д. В результате жестко заданная цель обеспечения полной безопасности приводит к единственному оптимальному решению: просто оставаться в гараже. Необходим некий компромисс между требованием достижения прогресса в отношении приближения к месту назначения и риском получить при этом травму. Как можно достичь такого компромисса? Пойдем дальше: в какой степени мы можем позволить самоуправляемой машине совершать действия, которые будут раздражать других водителей? В какой степени автомобиль должен смягчать ускорение, крутизну поворотов и резкость торможения, чтобы исключить неприятные ощущения у пассажира? На такие вопросы сложно ответить априори. Эти и другие подобные аспекты создают множество проблем во всей области взаимодействия человека и робота, где самоуправляемый автомобиль является лишь одним из примеров.

Проблема достижения согласия между нашими истинными предпочтениями и той целью, которую мы ставим перед машиной, называется ▶**проблемой**

выравнивания ценностей: ценности или цели, передаваемые машине, должны быть согласованы с ценностями человека. Если система искусственного интеллекта разрабатывается в лаборатории или в симуляторе — как это и было раньше в большинстве исследований в данной области, — будет очень просто исправить неверно выбранную цель: сбросить систему, откорректировать цель и попробовать еще раз. Но по мере того, как исследования в этой области проводятся со все более и более сложными и умными интеллектуальными системами, развертываемыми в реальном мире, такой подход становится нежизнеспособным. Развертывание системы с неверно заданной целью неизбежно будет иметь негативные последствия. Более того, чем более интеллектуальной будет такая система, тем более отрицательными будут последствия.

Возвращаясь к явно беспроблемному примеру игры в шахматы, рассмотрим, что произойдет, если машина будет достаточно умна, чтобы рассуждать и действовать и за пределами шахматной доски. В этом случае она может попытаться увеличить свои шансы на победу, используя такие хитрости, как использование гипноза или шантаж своего оппонента, либо даже подкуп аудитории, чтобы она шумела в то время, когда противник будет размышлять над очередным ходом.[3] Она даже может попытаться захватить для себя дополнительные вычислительные мощности. ➡ *Такое поведение не является "неразумным" или "безумным", в действительности оно является логическим следствием определения победы как единственной цели машины.*

Невозможно противодействовать всем способам неправильного поведения машины, преследующей фиксированную цель. И это весомая причина, чтобы прийти к заключению, что стандартная модель является неадекватной. Мы не хотим машин, которые будут интеллектуальными в смысле преследования *их* целей; мы хотим, чтобы они преследовали *наши* цели. Если мы не можем точно передать эти цели машине, то нам нужна новая формулировка — такая, согласно которой машина преследует наши цели, но *обязательно* не имеет полной уверенности в том, каковы они. Когда машина знает, что ей не известны цели во всей их полноте, у нее будет стимул действовать осторожно, просить разрешения на те или иные действия, чтобы узнать больше о наших предпочтениях посредством наблюдения, и считаться с контролем со стороны человека. В конечном счете мы хотим агентов, которые будут ▶**доказуемо полезны** человеку. Мы вернемся к этой теме в разделе 1.5.

1.2 Истоки искусственного интеллекта

В этом разделе кратко описана история развития научных дисциплин, которые внесли свой вклад в область ИИ в виде конкретных идей, воззрений и методов. Как и в любом историческом очерке, поневоле приходится ограничиваться описанием небольшого круга людей, событий и открытий, игнорируя все остальные

[3] В одной из первых книг по шахматам Руи Лопес (1561) писал: "Всегда ставьте доску так, чтобы солнце светило в глаза вашему противнику".

факты, которые также были важны. Авторы построили этот исторический экскурс вокруг ограниченного круга вопросов. Безусловно, они не хотели бы, чтобы у читателя создалось такое впечатление, будто эти вопросы являются единственными, которые рассматриваются в указанных научных дисциплинах, или что сами эти дисциплины развивались исключительно ради того, чтобы их конечным итогом стало создание искусственного интеллекта.

1.2.1. Философия

- Можно ли использовать формальные правила для получения обоснованных заключений?
- Как мысль возникает в физическом мозге?
- Откуда приходят знания?
- Каким образом знание ведет к действию?

Аристотель (384–322 до н.э.) был первым, кто сформулировал точный свод законов, регулирующих рациональную часть нашего мышления. Он разработал неформальную систему силлогизмов, предназначенную для проведения правильных рассуждений, которая в принципе позволяла любому делать выводы механически, лишь на основании начальных предпосылок.

Раймон Луллий (ок. 1232–1315) разработал систему рассуждений, опубликованную им под названием *Ars Magna,* или *Великое искусство* ([1438], 1305). Луллий даже предпринял попытку реализовать свою систему в виде механического устройства: набора бумажных колец, которые можно было поворачивать, получая разные перестановки.

Около 1500 года Леонардо да Винчи (1452–1519) спроектировал, но не построил механический калькулятор. Недавние реконструкции этого устройства показали, что оно вполне работоспособно. Первая известная машина для выполнения расчетов была создана примерно в 1623 году немецким ученым Вильгельмом Шиккардом (1592–1635). В 1642 году Блез Паскаль (1623–1662) построил арифметическую машину "Паскалин". Он писал, что она "производит эффект, который кажется более близким к мышлению по сравнению с любыми действиями животных". Готфрид Вильгельм Лейбниц (1646–1716) создал механическое устройство, предназначенное для выполнения операций над *концепциями*, а не числами, но область его применения была довольно ограниченной. В своей книге *Левиафан*, вышедшей в 1651 году, Томас Гоббс (1588–1679) выдвинул идею создания думающей машины, "искусственного животного", как он ее называл. По его словам, "Вместо сердца у нее будет пружина, вместо нервов — пучок струн, а вместо суставов — множество колес". Он также предположил, что рассуждение подобно числовым расчетам: "Ведь «суждение»... это не что иное, как «подведение итогов», в ходе которого мы складываем и вычитаем".

Одно дело — сказать, что сознание функционирует, по крайней мере частично, в соответствии с логическими или числовыми правилами, а затем построить

физические системы, которые имитируют некоторые из этих правил. Совсем другое дело — сказать, что сознание само по себе является такой физической системой. Рене Декарт (1596–1650) впервые опубликовал строгое обсуждение различий между разумом и материей. Он отметил, что чисто физическая концепция ума, похоже, оставляет мало места для свободной воли. Если сознание регулируется исключительно физическими законами, то оно имеет не больше свободной воли, чем скала, "решившая" рухнуть вниз. Декарт был сторонником ▶**дуализма**. Он считал, что есть часть человеческого сознания (или *душа* либо *дух*), которая находится за пределами естества и не подчиняется физическим законам. Животные, с другой стороны, не обладают таким дуалистическим свойством, поэтому их можно рассматривать как машины.

Альтернативой дуализму является **материализм**, утверждающий, что сознание *складывается* из операций, выполняемых мозгом в соответствии с законами физики. Свободная воля — это просто форма, которую принимает восприятие нашим существом доступных вариантов в процессе выбора. Для описания подобного представления, исключающего любую возможность существования сверхъестественного, также используются термины **физикализм** и **натурализм**.

Если полагать, что знаниями манипулирует физический разум, то возникает следующая проблема — установить источник знаний. Такое научное направление, как ▶**эмпиризм**, родоначальником которого был Френсис Бекон (1561–1626), автор *Нового Органона*,[4] можно охарактеризовать высказыванием Джона Локка (1632–1704): "В человеческом понимании нет ничего, что не проявлялось бы прежде всего в ощущениях".

Дэвид Юм (1711–1776) в своей книге *Трактат о человеческой природе* ([1095], 1739) предложил метод, известный теперь под названием ▶**принцип индукции**, — общие правила вырабатываются путем изучения повторяющихся ассоциаций между элементами, которые рассматриваются в этих правилах.

Основываясь на работе Людвига Витгенштейна (1889–1951) и Бертрана Рассела (1872–1970), знаменитый Венский кружок, группа философов и математиков, собиравшихся в Вене в 1920–1930-е годы, разработал доктрину ▶**логического позитивизма**. Согласно этой доктрине все знания могут быть охарактеризованы с помощью логических теорий, связанных в конечном итоге с ▶**протокольными предложениями**, которые соответствуют наблюдаемым фактам. Таким образом, логический позитивизм объединяет рационализм и эмпиризм.

В ▶**теории подтверждения** Рудольфа Карнапа (1891–1970) и Карла Хемпеля (1905–1997) была предпринята попытка понять, как знания могут быть приобретены из опыта посредством количественной оценки степени доверия, присваиваемой логическим предложениям на основе сопоставления с наблюдениями, подтверждающими или опровергающими их. В книге Карнапа *Логическая структура*

[4] Книга *Новый органон* была создана как новая версия труда Аристотеля *Органон* (инструмент мышления).

мира ([372], 1928) была сформулирована, по-видимому, первая теория мышления как вычислительного процесса.

Последним элементом в философской картине разума является связь между знанием и действием. Этот вопрос является жизненно важным для искусственного интеллекта, поскольку интеллектуальность требует не только рассуждений, но и действий. Более того, только понимая, как обосновать предпринимаемые действия, можно понять, как создать агент, действия которого будут обоснованы (или рациональны).

Аристотель утверждал (в *De Motu Animalium*), что действия обосновываются логической связью между целями и знанием о результатах этих действий.

> Но почему происходит так, что размышления иногда сопровождаются действием, а иногда — нет, иногда за ними следует движение, а иногда — нет? Создается впечатление, что почти то же самое происходит и в случае построения рассуждений и формирования выводов о неизменных объектах. Но в таком случае целью умственной деятельности оказывается умозрительное суждение... тогда как заключением, которое следует из данных двух предпосылок, является действие... Мне нужна защита от дождя; защитой может послужить плащ. Мне нужен плащ. Я должен изготовить то, в чем нуждаюсь; я нуждаюсь в плаще. Я должен изготовить плащ. И заключение "я должен изготовить плащ" становится действием.

В книге *Никомахова этика* (том III. 3, 1112b) Аристотель дополнительно развивает эту тему, предлагая следующий алгоритм.

> Мы размышляем не о конечных целях, а о средствах. Врач не обдумывает, должен ли он лечить, а оратор — должен ли он убедить... Они уже установили конечную цель и рассматривают, как и за счет чего она достигается, и если окажется несколько средств, то определяют, какое из них самое простое и наилучшее; если же достижению цели служит одно средство, думают, *как* ее достичь при помощи этого средства и что будет средством для *этого* средства, пока не дойдут до первой причины, которую находят последней... и то, что является последним в порядке анализа, окажется первым в порядке выполнения. А если мы сталкиваемся с невозможностью, то прекращаем поиск — например, если нам нужны деньги, а их нельзя получить, — но если что-то кажется возможным, мы пытаемся это сделать.

Алгоритм Аристотеля был реализован через 2300 лет Ньюэллом и Саймоном в их программе **General Problem Solver** (GPS). Теперь то, что создано на его базе, принято называть системой жадного регрессивного планирования (см. главу 11). Методы, основанные на логическом планировании для достижения определенных целей, доминировали в первые несколько десятилетий теоретических исследований в области ИИ.

Анализ исключительно с точки зрения действий по достижению цели часто является полезным, но иногда оказывается неприменимым. Например, если к цели ведет несколько вариантов действий, необходимо иметь какой-то способ выбирать среди них. Еще важнее то, что иногда может не быть полной уверенности в возможности достижения цели, но некоторые действия все же следовало бы

предпринять. Как в таких ситуациях следует поступать? Антуан Арно ([74], 1662), анализируя идею принятия рациональных решений в азартных играх, предложил количественную формулу максимизации ожидаемого конечного денежного результата. Позже Даниэль Бернулли ([191], 1738) ввел более общее понятие ▶**полезности** для фиксации внутренней, субъективной ценности результата. Современное понятие рационального принятия решений в условиях неопределенности предполагает максимизацию ожидаемой полезности, как это описывается в главе 16.

В вопросах этики и государственной политики лицо, принимающее решения, должно учитывать интересы множества людей. Джереми Бентам ([176], 1823) и Джон Стюарт Милль ([1576], 1863) поддерживали идею ▶**утилитаризма**: рациональное принятие решений на основе максимизации полезности должно применяться во всех сферах человеческой деятельности, в том числе в области государственных политических решений, принимаемых от имени многих людей. Утилитаризм является одной из форм ▶**консеквенциализма**, основная идея которого такова: что считать правильным или неправильным, определяется ожидаемыми результатами действия.

В противоположность этому Иммануил Кант в 1875 году предложил свою теорию ▶**деонтологической этики**, базирующейся на системе установленных правил. Согласно ее положениям правильность действия определяется не по результатам, а по соответствию универсальным социальным законам, регулирующим допустимость действий, таким как "не лги" или "не убивай". Таким образом, последователь утилитаризма имеет право на "белую" ложь, если ее ожидаемые хорошие следствия перевешивают плохие, тогда как для приверженца этики Канта это недопустимо, поскольку ложь в самой своей сути является действием неправильным. Милль признавал значение правил, но понимал их как эффективные процедуры принятия решений, составленные на результатах первичных рассуждений о последствиях. Во многих современных системах ИИ применяется именно этот подход.

1.2.2. Математика

- Каковы формальные правила формирования правильных заключений?
- Что может быть вычислено?
- Как проводить рассуждения на основе недостоверной информации?

Философы сформулировали наиболее важные идеи искусственного интеллекта, но для его преобразования в формальную науку потребовалось достичь определенного уровня математической формализации в области логики, теории вероятности и разработки новой ветви математики: теории вычислений.

Истоки идей ▶**формальной логики** можно найти уже в работах философов Древней Греции, Индии и Китая, но ее становление как математической дисциплины фактически началась с трудов Джорджа Буля (1815–1864), который детально разработал логику высказываний, или булеву логику. В 1879 году Готтлоб Фреге (1848–1925) расширил булеву логику для включения в нее объектов и отношений

[775], создав логику первого порядка, которая в настоящее время используется как наиболее фундаментальная система представления знаний.[5] Помимо своей центральной роли в ранний период исследований в области ИИ, логика первого порядка мотивировала работы Гёделя и Тьюринга, которые заложили теоретические основы вычислительной техники, как это будет объяснено ниже.

▶**Теорию вероятности** можно рассматривать как обобщение логики на ситуации с неопределенной информацией — весьма важный вклад в теорию искусственного интеллекта. Итальянский математик Джероламо Кардано (1501–1576) первым сформулировал идею вероятности, описывая ее в терминах результатов событий с несколькими исходами, возникающих в азартных играх. В 1654 году Блез Паскаль (1623–1662), в письме Пьеру Ферма (1601–1665), показал, как можно предсказать будущее в бесконечной азартной игре и распределить средний выигрыш между игроками. Вероятность быстро стала неотъемлемой частью всех количественных наук, помогая справляться с неточностью измерений и незавершенностью теорий. Якоб Бернулли (1654–1705, дядя Даниила Бернулли), Пьер Лаплас (1749–1827), и другие внесли большой вклад в эту теорию и ввели новые статистические методы. Томас Байес (1702–1761) предложил правило обновления вероятностей с учетом новых фактов. Правило Байеса и возникшее на его основе научное направление, называемое байесовским анализом, являются важными инструментами для систем ИИ.

Формализации вероятности, в сочетании с доступностью данных, привели к появлению ▶**статистики** как нового поля научных исследований. Одним из первых достижений в этой области стал выполненный Джоном Граунтом анализ данных переписи населения Лондона 1662 года. Первым современным статистиком считается Рональд Фишер. Он объединил идеи вероятности, планирования эксперимента, анализа данных и вычислений. В 1919 году он настаивал на том, что не смог бы выполнять свою работу без механического калькулятора под названием MILLIONAIRE (первый арифмометр, позволявший выполнять операцию умножения), даже несмотря на то, что стоимость этого калькулятора была больше, чем его годовая зарплата.

История вычислений так же стара, как история чисел, но первым нетривиальным ▶**алгоритмом** считается алгоритм вычисления наибольшего общего знаменателя, предложенный Евклидом. Само слово "алгоритм" пришлом к нам от Мухаммеда ибн Мусы аль-Хорезми, среднеазиатского математика IX столетия, чьи труды также познакомили Европу с арабскими цифрами и алгеброй. Буль и другие ученые широко обсуждали алгоритмы логического вывода, а к концу XIX столетия даже предпринимались усилия по формализации общих принципов проведения математических рассуждений как логического вывода.

---

[5] Предложенная Готтлобом Фреге система обозначений для логики первого порядка, представлявшая собой загадочную комбинацию из текстовых и геометрических элементов, так и не нашла широкого распространения.

Курт Гёдель (1906–1978) показал, что существует эффективная процедура доказательства любого истинного высказывания в логике первого порядка Фреге и Рассела, но при этом логика первого порядка не позволяет выразить принцип математической индукции, необходимый для представления натуральных чисел. В 1931 году Гёдель показал, что действительно существуют реальные пределы вычислимости. Предложенная им ▶**теорема о неполноте** показывает, что в любой теории, достаточно выразительной для описания свойств арифметики Пеано (элементарной теории натуральных чисел), существуют истинные высказывания, которые являются недоказуемыми в рамках этой теории.

Этот фундаментальный результат также может быть интерпретирован как демонстрация того, что некоторые функции на целых числах не могут быть представлены с помощью какого-либо алгоритма, т.е. они не могут быть вычислены. Это побудило Алана Тьюринга (1912–1954) попытаться точно охарактеризовать, какие функции *являются* ▶**вычислимыми**, т.е. могут быть вычислены с использованием некоторой эффективной процедуры. Тезис Черча–Тьюринга предлагает отождествить общее понятие вычислимости с функциями, вычисляемыми машиной Тьюринга. Тьюринг также показал, что существуют некоторые функции, которые ни одна машина Тьюринга не может вычислить. Например, никакая машина не сможет *в общем случае* определить, будет ли указанная программа возвращать результат и прекращать работу при указанных данных или будет работать бесконечно.

Хотя понятие вычислимости очень важно для понимания возможностей вычисления, гораздо большее влияние на развитие искусственного интеллекта оказало понятие ▶**разрешимости**. Грубо говоря, задача называется неразрешимой, если время, требуемое для решения отдельных примеров этой задачи, растет экспоненциально с увеличением размеров этих примеров. Различие между полиномиальным и экспоненциальным ростом сложности было впервые подчеркнуто в середине 1960-х годов в работах Кобхэма и Эдмондса. Это важно, потому что экспоненциальный рост сложности означает, что даже умеренно большие примеры могут оказаться неразрешимыми за какое-либо разумное время.

Теория ▶**NP-полноты**, впервые предложенная Стивеном Куком и Ричардом Карпом, предоставляет необходимую основу для анализа разрешимости задач: любой класс задач, к которому может быть сведен класс NP-полных задач, является, по-видимому, неразрешимым. (Хотя еще не было доказано, что NP-полные задачи обязательно являются неразрешимыми, большинство теоретиков считают, что дело обстоит именно так.) Эти результаты контрастируют с тем оптимизмом, с которым в популярных периодических изданиях приветствовалось появление первых компьютеров под такими заголовками, как "Электронные супермозги", которые думают "быстрее Эйнштейна!" Несмотря на постоянное повышение быстродействия компьютеров, экономное использование ресурсов и вынужденное несовершенство являются характерными особенностями интеллектуальных систем. Грубо говоря, наш мир — это *чрезвычайно* крупный экземпляр задачи.

1.2.3. Экономика

- Как нам следует принимать решения в соответствии с нашими предпочтениями?
- Как это следует делать, когда другие могут препятствовать нам?
- Как действовать в таких случаях, когда вознаграждение может быть получено лишь в отдаленном будущем?

Экономика как наука возникла в 1776 году, когда шотландский философ Адам Смит (1723–1790) опубликовал свою книгу *Исследование о природе и причинах богатства народов*. Смит предложил рассматривать экономику как состоящую из множества индивидуальных агентов, стремящихся к достижению собственных интересов. Смит, однако, не рассматривал стремление к финансовому обогащению как основную моральную установку: свою раннюю книгу *Теория моральных отношений* (1759) он начал с указания, что беспокойство о благополучии других является важным компонентом интересов каждого индивида.

Большинство людей считают, что экономика имеет дело исключительно с деньгами, и действительно, первый математический анализ принятия решений в условиях неопределенности — формула максимальной ожидаемой стоимости Арнольда — имел отношение к денежной стоимости ставок. Даниил Бернулли отметил, что эта формула, похоже, плохо работает в случае больших денежных сумм, например инвестиций в морские торговые экспедиции. Вместо нее он предложил принцип, построенный на максимизации ожидаемой полезности, и объяснил выбор инвестиций людьми исходя из предположения, что минимальная полезность дополнительного количества денег уменьшается, когда человек получает больше денег.

Леон Вальрас дал более общую математическую трактовку теории полезности в терминах предпочтений между азартными играми на любые результаты (не только денежные). Эта теория была улучшена Фрэнком Рамсеем, а затем усовершенствована Джоном фон Нейманом и Оскаром Моргенштерном в книге *Теория игр и экономического поведения* ([2282], 1944). Сейчас экономика уже не рассматривается как наука о деньгах — скорее, это изучение намерений и предпочтений людей.

▶ **Теория принятия решений**, объединяющая в себе теорию вероятностей и теорию полезности, предоставляет формальную и полную инфраструктуру для принятия решений (в области экономики или в другой области) в условиях неопределенности, т.е. в тех случаях, когда среда, в которой действует лицо, принимающее решение, наиболее адекватно может быть представлена лишь с помощью вероятностных описаний. Она хорошо подходит для "крупных" экономических образований, где каждый агент не обязан учитывать действия других агентов как индивидуумов. Однако в "небольших" экономических образованиях ситуация в большей степени напоминает **игру**, поскольку действия одного игрока могут существенно повлиять на полезность действий другого (или положительно, или

отрицательно). **Теория игр**, разработанная фон Нейманом и Моргенштерном, позволяет сделать неожиданный вывод: в некоторых играх рациональный агент должен действовать случайным образом или по крайней мере таким образом, который кажется случайным для соперников. В отличие от теории принятия решений, теория игр не предлагает однозначного рецепта для выбора действий. В области исследований искусственного интеллекта решения с участием нескольких агентов исследуются под заголовком **мультиагентные системы** (глава 18).

Экономисты за немногими исключениями не стремятся найти ответ на третий вопрос, приведенный в начале раздела, т.е. не предпринимают попыток выработать способ принятия рациональных решений в таких условиях, когда вознаграждение в ответ на определенные действия не предоставляется немедленно, а становится результатом нескольких действий, выполненных в определенной *последовательности*. Изучению этой темы посвящена область ▶**исследования операций**, которая возникла во время второй мировой войны в результате усилий, предпринятых в Великобритании в отношении оптимизации работы радарных установок, а в дальнейшем нашла применение и в гражданском обществе при выработке сложных управленческих решений. В работе Ричарда Беллмана ([169], 1957) формализован определенный класс последовательных задач выработки решений, называемых **марковскими процессами принятия решений**, которые рассматриваются в главе 17, а под названием **обучение с подкреплением** — в главе 22.

Работы в области экономики и исследования операций оказали большое влияние на сформулированное в этой книге понятие рациональных агентов, однако в течение многих лет исследования в области искусственного интеллекта проводились совсем по другим направлениям. Одной из причин этого была кажущаяся сложность задачи выработки рациональных решений. Тем не менее один из первых исследователей в области искусственного интеллекта, Герберт Саймон (1916–2001), получил в 1978 году Нобелевскую премию по экономике за свои ранние работы, в которых показал, что модели, основанные на ▶**разумной достаточности** (т.е. на принятии решений, которые являются "достаточно приемлемыми", вместо проведения трудоемких расчетов с целью нахождения оптимального решения), дают лучшее описание фактического поведения человека. С 1990-х годов отмечается возрождение интереса к использованию методов теории принятия решений в применении к системам искусственного интеллекта.

1.2.4. Нейронауки

- Как информация обрабатывается в мозгу?

▶**Нейронауки** — это область научных исследований, посвященная изучению нервной системы, в особенности мозга. Хотя точный способ, посредством которого мозг реализует мышление, все еще является одной из самых больших тайн в науке, тот факт, что он действительно обеспечивает мышление, был известен

в течение тысяч лет, поскольку существовали свидетельства, что сильные удары по голове могут привести к умственной недееспособности. Также давно было известно, что человеческий мозг чем-то отличается от мозга других живых существ, — примерно в 335 г. до н.э. Аристотель писал: "Из всех животных у человека самый большой мозг в отношении к его размерам".[6] Тем не менее широкое признание того, что мозг является вместилищем сознания, пришло только в середине XVIII столетия. До этого времени в качестве возможных источников сознания рассматривались сердце и селезенка.

Поль Брока (1824–1880) в 1861 году провел исследования афазии (нарушения речи) у пациентов с повреждением мозга, которые стали отправной точкой для проведения исследований функциональной организации мозга, благодаря выявлению определенной области в левом полушарии — теперь ее называют зоной Брока, — которая отвечает за организацию речевой активности.[7] К тому времени уже было известно, что мозг состоит из нервных клеток, или ▶**нейронов**, но только в 1873 году Камилло Гольджи (1843–1926) сумел разработать надежный метод, позволяющий наблюдать за отдельными нейронами в мозгу (рис. 1.1). Именно этот метод использовал Сантьяго Рамон-и-Кахаль (1852–1934) в своих пионерских исследованиях нейронных структур мозга.[8] В настоящее время общепризнано, что когнитивные функции являются результатом электрохимического воздействия этих структур. То есть ➔*совокупность простых клеток может привести к мышлению, действию и пониманию*. По содержательным словам Джона Сирла ([2025], 1992), *мозг является причиной разума*.

Теперь ученые располагают некоторыми данными о том, как связаны между собой отдельные области мозга и те части тела, которыми они управляют или от которых получают сенсорные данные. Удивительно, что подобная привязка может коренным образом измениться в течение нескольких недель, а у некоторых животных, по-видимому, имеется несколько вариантов такой привязки. Более того, еще не совсем понятно, как другие области могут взять на себя функции поврежденных областей. К тому же почти полностью отсутствуют обоснованные теории того, как осуществляется хранение информации в памяти индивидуума или как работают высокоуровневые когнитивные функции.

---

[6] С течением времени было обнаружено, что у землеройки и некоторых видов птиц соотношение веса мозга и тела превышает таковое у человека.

[7] В качестве возможного более раннего источника многие цитируют работу Александра Гуда (1824).

[8] Гольджи упорно отстаивал свое мнение, что функции мозга осуществляются в основном непрерывной средой, в которую включены нейроны, тогда как Кахаль предлагал "нейронную доктрину". Эти ученые совместно получили Нобелевскую премию в 1906 году, но в роли лауреатов произнесли речи, содержащие взаимные антагонистичные выпады.

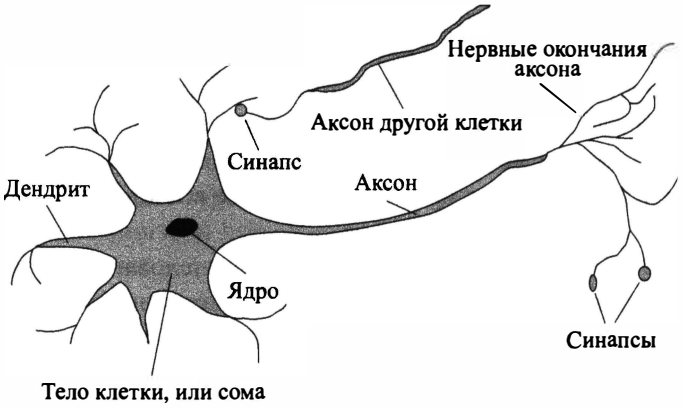


Рис. 1.1. Части нервной клетки, или нейрона. Каждый нейрон состоит из тела клетки (или сомы), которое содержит ядро клетки. От тела клетки ответвляется множество коротких волокон, называемых дендритами, и одно длинное волокно, называемое аксоном. Аксон растягивается на большое расстояние, намного превышающее то, что показано в масштабе этого рисунка. Обычно аксоны имеют длину 1 см (что в 100 раз больше диаметра тела клетки), но могут достигать 1 метра. Нейрон соединяется с другими нейронами, количество которых может составлять от 10 до 100 000 в точках соединения, называемых синапсами. Сигналы распространяются от одного нейрона к другому с помощью сложной электрохимической реакции. Эти сигналы управляют активностью мозга в течение короткого интервала времени, а также вызывают долговременные изменения состояния самих нейронов и их соединений. Считается, что именно эти механизмы служат в мозгу основой для обучения. Обработка информации происходит главным образом в коре головного мозга, которая представляет собой внешний слой нейронов мозга. По-видимому, основной ее структурной единицей является колонка ткани или *модуль*, имеющий диаметр около 0,5 мм и протяженность на всю глубину коры, толщина которой в человеческом мозгу составляет около 4 мм. Каждый модуль содержит примерно 20 000 нейронов

Первые измерения активности неповрежденного мозга начали проводить в 1929 году с изобретением Гансом Бергером электроэнцефалографа (ЭЭГ). Разработки в области получения изображений на основе функционального магнитного резонанса (МРТ) позволили нейрологам получать исключительно подробные изображения активности мозга, что обеспечило возможность проведения измерений, связанных с происходящими познавательными процессами различными интересными способами. Эти возможности постоянно расширяются благодаря прогрессу в области регистрации нейронной активности отдельной клетки и в методах ▶**оптогенетики**, позволяющих как измерять, так и управлять активностью отдельных нейронов, модифицированных таким образом, чтобы сделать их чувствительными к свету.

Разработка ▶**интерфейсов "мозг–машина"** (Лебедев и Николелис [1369], 2006) для сенсорного и двигательного управления не только обещает получить средства восстановления функциональных возможностей людей с ограниченными возможностями, но также проливает свет на многие аспекты нейронных систем. Замечательный вывод этой работы состоит в том, что мозг способен настроить себя на успешное взаимодействие с внешним устройством, рассматривая его в действии как еще один орган чувств или конечность.

Мозг и цифровые компьютеры имеют несколько различающиеся свойства. На рис. 1.2 показано, что продолжительность рабочего цикла современного компьютера в миллион раз меньше, чем мозга. Мозг компенсирует эту разницу за счет гораздо большего объема памяти и количества взаимосвязей в сравнении с персональным компьютером даже самого высокого класса, хотя крупнейшие суперкомпьютеры по некоторым показателям уже соответствуют нашему мозгу. Футуристы придают большое значение этим показателям, указывая на приближение ▶**особого момента**, когда компьютеры достигнут сверхчеловеческого уровня производительности (Винге [2272], 1993; Курцвейл [1330], 2005; Докторов и Стросс [629], 2012), а затем стремительно улучшат себя в еще большей степени. Однако сравнение самих числовых значений недостаточно информативно. Даже в случае создания компьютера с практически неограниченной мощностью, нам по-прежнему потребуются дальнейшие концептуальные прорывы в нашем понимании интеллекта (см. главу 28). Грубо говоря, без наличия правильной теории более быстрые машины будут просто быстрее давать нам неправильные ответы.

	Суперкомпьютер	Персональный компьютер	Мозг человека
Вычислительные модули	10^6 ГП + ЦП	8-ядерный ЦП	10^6 колонок
	10^{15} транзисторов	10^{10} транзисторов	10^{11} нейронов
Модули памяти	10^{16} байт ОП	10^{10} байт ОП	10^{11} нейронов
	10^{17} байт ДП	10^{12} байт ДП	10^{14} синапсов
Цикл обработки	10^{-9} секунды	10^{-9} секунды	10^{-3} секунды
Операций в секунду	10^{18}	10^{10}	10^{17}

Рис. 1.2. Грубое сравнение характеристик лидирующего суперкомпьютера Summit (Фельдман [719], 2017), типичного персонального компьютера образца 2019 года, и человеческого мозга. (Здесь ГП — графический процессор, ЦП — центральный процессор, ОП — оперативная память, ДП — дисковая память.) Мощность человеческого мозга не изменялась сколько-нибудь значительно на протяжении тысяч лет, тогда как мощность суперкомпьютеров возросла от мегафлопсов в 1960-х до гигафлопсов в 1980-х, терафлопсов в 1990-х, петафлопсов в 2008 году и эксафлопсов в 2018 году (1 эксафлопс = 10^{18} операций с плавающей точкой в секунду)

1.2.5. Психология

- Как люди и животные думают и действуют?

Истоки научной психологии обычно прослеживаются до работ немецкого физика Германа фон Гельмгольца (1821–1894) и его студента Вильгельма Вундта (1832–1920). Гельмгольц применил научный метод для изучения зрения человека. Выпущенная им книга *Справочник по физиологической оптике* даже в наши дни характеризуется как "непревзойденный по своей важности вклад в изучение физики и физиологии зрения человека" (Нальва [1655], 1993). В 1879 году Вундт открыл первую лабораторию по экспериментальной психологии в Лейпцигском университете. Он настаивал на проведении тщательно контролируемых экспериментов, в которых его сотрудники выполняли задачи по восприятию или формированию ассоциаций, проводя интроспективные наблюдения за своими мыслительными процессами. Тщательный контроль позволил ему сделать очень многое для превращения психологии в науку, но субъективный характер делал маловероятным, что экспериментаторы когда-либо будут опровергать их собственными теориями.

С другой стороны, биологам, изучающим поведение животных, интроспективные данные недоступны, поэтому они разработали объективную методологию, которую Г.С. Дженнингс описал в своей нашумевшей работе *Поведение низших организмов* ([1133], 1906). Распространив этот подход на людей, сторонники движения, возглавляемого Джоном Уотсоном (1878–1958) и получившего название ▶**бихевиоризм**, отвергали любую теорию, учитывающую мыслительные процессы, на том основании, что интроспекция не может предоставлять надежные свидетельства. Бихевиористы настаивали на изучении только объективных мер восприятия (или *стимулов*), предъявленных животному, и их вытекающих из этого действий (или *откликов*). Бихевиоризм позволил многое узнать о крысах и голубях, но оказался менее успешным при изучении поведения человека.

Взгляды, согласно которым мозг рассматривается как устройство обработки информации, характерны для представителей ▶**когнитивной психологии** и прослеживаются по крайней мере до работ Уильяма Джеймса (1842–1910). Гельмгольц также настаивал на том, что восприятие связано с определенной формой подсознательного логического вывода. В Соединенных Штатах такой подход к изучению познавательных процессов был в основном отвергнут из-за широкого распространения бихевиористских взглядов, но на факультете прикладной психологии Кембриджского университета, возглавляемом Фредериком Бартлеттом (1886–1969), когнитивное моделирование активно поддерживалось. В книге *Природа объяснения* ([491], 1943) ученик и последователь Бартлетта Кеннет Крэйк привел весомые доводы в пользу допустимости применения таких "мыслительных" терминов, как "убеждения" и "цели", доказав, что они являются не менее научными, чем, скажем, такие термины, применяемые в рассуждениях о газах, как "давление" и "температура", несмотря на то что речь в них идет о молекулах, которые сами не обладают этими характеристиками.

Крэйк обозначил следующие три этапа деятельности агента, основанного на знаниях: во-первых, действующий стимул должен быть преобразован во внутреннее представление, во-вторых, с этим представлением должны быть выполнены манипуляции с помощью познавательных процессов для выработки новых внутренних представлений и, в-третьих, они должны быть, в свою очередь, снова преобразованы в действия. Он наглядно объяснил, почему такой проект является приемлемым для любого агента.

> Если живой организм несет в своей голове "модель в уменьшенном масштабе" внешней реальности и своих возможных действий, то он способен проверять различные альтернативы, делать заключение, какая из них является наилучшей для него, реагировать на будущие ситуации, прежде чем они возникнут, использовать знания о прошлых событиях, сталкиваясь с настоящим и будущим, и во всех отношениях реагировать на опасности, встречаясь с ними, гораздо полнее, безопаснее и более компетентно (Крейк [491], 1943).

В 1945 году, после смерти Крэйка в результате аварии во время поездки на велосипеде, его работа была продолжена Дональдом Броудбентом, чья книга *Восприятие и коммуникация* ([310], 1958) была одной из первых работ по моделированию психологических явлений как процессов обработки информации. Между тем в Соединенных Штатах работы в области компьютерного моделирования привели к созданию такого научного направления, как ▶**когнитивная наука** (или **когнитивистика**). Зарождение этого направления исследований произошло на одном из семинаров в Массачусетсском технологическом институте в сентябре 1956 года. (Ниже будет показано, что это событие произошло всего лишь через два месяца после проведения конференции, на которой "родился" сам искусственный интеллект.)

На этом семинаре Джордж Миллер представил доклад *Волшебное число "семь"*, Ноам Хомский прочитал доклад *Три модели языка*, а Аллен Ньюэлл и Герберт Саймон представили свою работу *Машина теории логики*. В этих трех работах, получивших широкое признание, было показано, как можно использовать компьютерные модели для решения задач в области психологии памяти, обработки естественного языка и логического мышления соответственно. В настоящее время среди психологов находит широкое признание (хотя и не является универсальным) взгляд, что "любая теория познания должна быть подобна компьютерной программе" (Андерсон [47], 1980), т.е. должна описывать работу любой познавательной функции в терминах обработки информации.

Исходя из целей данного обзора, мы не будем рассматривать область **взаимодействия человека с компьютером** как раздел психологии. Дуг Энгельбарт, один из пионеров в этой области, отстаивал идею ▶**усиления интеллекта** (УИ) вместо ИИ. Он считал, что компьютеры должны *расширять* человеческие способности, а не автоматизировать решение задач человека. В 1968 году Энгельбарт в своем выступлении на компьютерной конференции (позднее его окрестили как "Мать всех демонстраций") впервые продемонстрировал такие невероятные на то время

новшества, как компьютерная мышь, оконный компьютерный интерфейс, гипертекстовый документ и даже проведение видеоконференции с коллективной работой над одним документом. Все это имело единственную цель — продемонстрировать, чего способны коллективно достичь занятые умственным трудом люди с помощью различных средств, отвечающих концепции усиления интеллекта.

Сегодня мы склонны рассматривать концепции УИ и ИИ как две стороны одной медали, делая акцент в первом случае на контроле со стороны человека, а во втором — на интеллектуальном поведении со стороны машины. И то, и другое необходимо, чтобы машины были полезны людям.

1.2.6. Компьютерная техника

- Как можно создать эффективный компьютер?

Современный цифровой электронный компьютер был изобретен независимо и почти одновременно учеными трех стран, сражавшихся во второй мировой войне. Первым *действующим* компьютером был электромеханический компьютер "Хит Робинсон",[9] построенный в 1943 году командой Алана Тьюринга с единственной целью: расшифровка перехваченных сообщений немецких войск. Позднее в том же 1943 году эта же группа разработала Colossus — мощную машину универсального назначения на основе электронных ламп.[10] Первым работающим *программируемым* компьютером был Z-3, изобретенный Конрадом Цузе в Германии в 1941 году. Цузе также изобрел способ представления чисел с плавающей точкой и первый язык программирования высокого уровня, *Plankalkül*. Первый *электронный* компьютер, ABC, был собран Джоном Атанасовым и его студентом Клиффордом Берри между 1940 и 1942 годами в Университете штата Айова. Работа Атанасова не получила необходимой поддержки или признания в отличие от проекта ENIAC — компьютера, созданного в рамках засекреченного военного проекта в Университете штата Пенсильвания командой разработчиков, включавшей Джона Мокли и Дж. Преспера Эккерта. Именно этот проект оказался самым влиятельным предтечей современных компьютеров.

С тех пор появление нового поколения компьютерной техники всегда сопровождалось увеличением скорости процессоров и емкости памяти с одновременным снижением стоимости — тенденция, зафиксированная в ▶**законе Мура**. Согласно этому закону производительность компьютеров удваивается каждые 18 месяцев или около того. Так продолжалось до 2005 года, когда проблема отвода тепла вынудила производителей обратиться к увеличению количества ядер в процессорах

[9] Сложная машина, названная в честь британского карикатуриста, изображавшего причудливые и нелепо сложные способы решения повседневных задач, например приготовления тостов.

[10] В послевоенный период Тьюринг хотел использовать эти компьютеры для исследований в области ИИ, например он создал схему первой шахматной программы (Тьюринг и др. [2236] 1953), но британское правительство запретило проведение этих исследований.

вместо повышения их тактовой частоты. В настоящее время ожидается, что дальнейшее увеличение функциональности будет достигаться за счет интенсивного распараллеливания вычислительных процессов — здесь просматривается любопытная схожесть со свойствами мозга. Также можно отметить новые аппаратные разработки, основанные на идее, что, имея дело с неопределенным миром, нам не требуется числовая точность, обеспечиваемая форматом слова в 64 бита. Вполне достаточно будет всего 16 бит (как в формате `bfloat16`) или даже 8 бит, что существенно ускорит обработку.

Мы лишь совсем недавно смогли увидеть компьютерное оборудование, специально предназначенное для применения в области искусственного интеллекта. Это, например, графические процессоры (ГПУ), тензорные процессоры (ТПУ) и суперпроцессор параллельной обработки Wafer Scale Engine (WSE). В период с 1960-х до 2012 года объем вычислительной мощности, используемой для обучения высокопроизводительных машин, соответствовал закону Мура. Начиная с 2012 года ситуация изменилась: с 2012 по 2018 год суммарные показатели выросли в 300 000 раз (!), что соответствует их удваиванию каждые 100 дней или около того (Амодей и Эрнандес [44], 2018). Модель машинного обучения, которая в 2014 году требовала на выполнение полного рабочего дня, в 2018 году выполнялась всего за две минуты (Йинг и др. [2404], 2018). Хотя ▶**квантовые вычисления** пока еще не достигли уровня практического применения, этот подход обещает гораздо большее ускорение для некоторых важных подклассов алгоритмов ИИ.

Безусловно, вычислительные устройства существовали и до появления электронного компьютера. Первые автоматизированные устройства, появившиеся еще в XVII столетии, уже упоминались в разделе 1.2.1. Первым *программируемым* устройством был ткацкий станок, изобретенный в 1805 году Жозефом Марией Жаккардом (1752–1834), в котором для хранения инструкций по плетению узоров ткани использовались перфокарты.

В середине XIX века Чарльз Бэббидж (1792–1871) разработал две вычислительные машины, ни одну из которых он так и не завершил. Его дифференциальная машина была предназначена для вычисления математических таблиц для инженерных и научных расчетов. Значительно позднее, уже в 1991 году, она все же была построена, а ее работоспособность наглядно продемонстрирована (Свейд [2160], 2000). Вторая, аналитическая, машина Бэббиджа была гораздо более амбициозной: она включала адресацию памяти, сохраненные программы на перфорированных картах Жаккарда, команды условных переходов. Это была первая машина, способная выполнять универсальные вычисления.

Коллега Бэббиджа Ада Лавлейс, дочь поэта лорда Байрона, осознала потенциал этого устройства, описав его как "мыслящая или... рассуждающая машина", способная рассуждать обо "всех предметах во Вселенной" (Лавлейс [1447], 1843). Она также предвидела возможность шумихи вокруг ИИ, написав, что "желательно защититься от возможности появления преувеличенных идей, которые могут возникнуть в отношении мощности аналитической машины". К сожалению, машины Бэббиджа и идеи Ады Лавлейс были в значительной степени забыты.

Искусственный интеллект во многом обязан также тем направлениям компьютерных наук, которые связаны с созданием операционных систем, языков программирования и прочих инструментальные средств, необходимых для написания современных программ (и статей о них). Но эта область научной деятельности является также одной из тех, в которых искусственный интеллект в полной мере возмещает свои долги: работы в области искусственного интеллекта стали источником многих идей, которые затем были воплощены в основных направлениях развития компьютерных наук, включая разделение времени, интерактивные интерпретаторы, персональные компьютеры с оконными интерфейсами и поддержкой позиционирующих устройств, применение эффективных сред разработки, создание типов данных в виде связных списков, автоматическое управление памятью и ключевые концепции символического, функционального, декларативного и объектно-ориентированного программирования.

1.2.7. Теория управления и кибернетика

- Как артефакты могут работать под собственным управлением?

Ктесибий из Александрии (ок. 250 до н.э.) построил первую самоуправляемую машину: водяные часы с регулятором, поддерживающим постоянную скорость потока. Это изобретение изменило определение того, что может делать артефакт. Ранее только живые существа могли изменять свое поведение в ответ на изменения в окружающей среде. Другие примеры саморегулирующихся систем управления с обратной связью включают регулятор парового двигателя, созданный Джеймсом Уаттом (1736–1819), и термостат, изобретенный Корнелисом Дреббелем (1572–1633), который также изобрел подводную лодку. Джеймс Клерк Максвелл ([1518], 1868) положил начало математической теории систем управления.

В послевоенный период центральной фигурой в разработке ▶**теории управления**, был Норберт Винер (1894–1964). Винер был блестящим математиком, который работал со многими учеными, включая Бертрана Рассела, прежде чем у него появился интерес к изучению биологических и механических систем управления и их связи с познанием. Как и Крэйк (который также использовал системы управления в качестве психологических моделей), Винер и его коллеги Артуро Розенблют и Джулиан Бигелоу бросили вызов ортодоксальным бихевиористским взглядам ([1914], 1943). Они рассматривали целенаправленное поведение как обусловленное действием некого регуляторного механизма, пытающего минимизировать "ошибку" — различие между текущим и целевым состояниями. В конце 1940-х годов Винер совместно с Уорреном Мак-Каллоком, Уолтером Питтсом и Джоном фон Нейманом организовал ряд влиятельных конференций, на которых рассматривались новые математические и вычислительные модели познания. Книга Винера ▶***Кибернетика*** ([2339], 1948) стала бестселлером и убедила широкие круги общественности в том, что мечта о создании машин, обладающих искусственным интеллектом, воплотилась в реальность.

Между тем в Великобритании У. Росс Эшби впервые *применил* подобные идеи (Эшби [82], 1940). Эшби, Алан Тьюринг, Грей Уолтер и другие сформировали "Ratio Club" для "тех, кто разделял идеи Винера до появления книги Винера". В книге *Дизайн для мозга* ([83], 1948; [84], 1952) Эшби разрабатывал свою идею о том, что разум может быть создан при использовании ▶ **гомеостатических** устройств, содержащих соответствующие петли обратной связи, обеспечивающие достижение стабильного адаптивного поведения.

Современная теория управления, особенно ее ветвь с названием *стохастическое оптимальное управление*, ставит своей целью проектирование систем, которые максимизируют ▶ **целевую функцию** во времени. Это примерно соответствует представлению авторов настоящей книги об искусственном интеллекте как о проектировании систем, которые действуют оптимальным образом. Почему же в таком случае искусственный интеллект и теория управления рассматриваются как две разные научные области, особенно если учесть, какие тесные взаимоотношения связывали их основателей? Ответ на этот вопрос состоит в том, что существует также тесная связь между математическими методами, которые были знакомы участникам этих разработок, и соответствующими множествами задач, которые были охвачены в каждом из этих подходов к описанию мира. Дифференциальное и интегральное исчисление, а также матричная алгебра, являющиеся инструментами теории управления, в наибольшей степени подходят для анализа систем, которые могут быть описаны с помощью фиксированных множеств непрерывно изменяющихся переменных, тогда как сама область исследований ИИ была отчасти основана как способ избежать этих ограничений математических средств. Такие инструменты, как логический вывод и вычисления, позволили исследователям искусственного интеллекта успешно рассматривать некоторые проблемы, такие как понимание естественного языка, зрение и символическое планирование, полностью выходящие за рамки исследований, предпринимавшихся теоретиками в области теории управления.

1.2.8. Лингвистика

- Каким образом язык связан с мышлением?

В 1957 году Б.Ф. Скиннер опубликовал свою книгу *Вербальное поведение*. Это был всеобъемлющий, подробный отчет о результатах исследований по изучению языка, проведенных в рамках бихевиористского подхода, написанный наиболее выдающимся экспертом в этой области. Но весьма любопытно то, что рецензия к этой книге стала не менее известной, чем сама книга, и послужила причиной почти полного исчезновения интереса к бихевиоризму. Автором этой рецензии был Ноам Хомский, который сам только что опубликовал книгу *Синтаксические структуры* ([422], 1957) с изложением собственной теории. Хомский показал, что бихевиористская теория не позволяет понять истоки творческой деятельности, осуществляемой с помощью языка, — она не объясняет, почему ребенок способен

понимать и складывать предложения, которые он до сих пор никогда еще не слышал. Теория Хомского, основанная на синтаксических моделях, восходящих к работам древнеиндийского лингвиста Панини (примерно 350 год до н.э.), позволяла объяснить этот феномен и, в отличие от предыдущих теорий, оказалась достаточно формальной для того, чтобы ее можно было реализовать в виде программ.

Таким образом, современная лингвистика и искусственный интеллект, которые "родились" примерно в одно и то же время и продолжают расти вместе, пересекаются в гибридной области, называемой ▶**вычислительная лингвистика** или **обработка естественного языка**. Со временем было обнаружено, что проблема понимания языка является гораздо более сложной, чем это казалось в 1957 году. Для понимания языка требуется понимание предмета и контекста речи, а не только анализ структуры предложений. Это утверждение теперь кажется очевидным, но сам данный факт не был широко признан до 1960-х годов. Основная часть ранних работ в области **представления знаний** (науки о том, как преобразовать знания в такую форму, которой может оперировать компьютер) была привязана к языку и подпитывалась исследованиями в области лингвистики, которые, в свою очередь, основывались на результатах философского анализа языка, проводившегося в течение многих десятков лет.

1.3. История искусственного интеллекта

Один из быстрых способов подвести итог вехам в истории ИИ состоит в том, чтобы перечислить некоторых лауреатов премии Тьюринга: Марвин Мински (1969) и Джон Мак-Карти (1971) — за формирование основ этого научного направления, построенных на представлении и рассуждении; Эдвард Фейгенбаум и Радж Редди (1994) — за разработку экспертных систем, кодирующих знания человека для решения реальных проблем; Джуди Перл (2011) — за разработку вероятностных методов рассуждения, которые принципиальным образом решают проблему неопределенности; и наконец, Йэшуа Бегио, Джеффри Хинтон и Янн ЛеКун (2019) — за разработку концепции "глубокого обучения" (многослойные нейронные сети), критически важной составляющей современной вычислительной технологии. В остальной части этого раздела каждая фаза истории ИИ описывается более подробно.

1.3.1. Начальный этап развития искусственного интеллекта (1943–1956)

Первая работа, которую сейчас по общему признанию относят к искусственному интеллекту, была выполнена Уорреном Мак-Каллоком и Уолтером Питтсом ([1537], 1943). Вдохновленные работой по математическому моделированию советника Питтса Николаса Рашевского, они опирались на три источника: знание основ физиологии и функции нейронов мозга; формальный анализ логики высказываний,

взятый из работ Рассела и Уайтхеда, и теорию вычислений Тьюринга. Мак-Каллок и Питтс предложили модель из искусственных нейронов, в которой каждый нейрон мог находиться в состоянии "включено" или "выключено" и переход в состояние "включено" происходил в ответ на стимуляцию со стороны достаточного количества соседних нейронов. Состояние нейрона рассматривалось как "фактически эквивалентное высказыванию, в котором предлагается адекватное количество стимулов". В своей работе они показали, что любая вычислимая функция может быть вычислена с помощью некоторой сети из соединенных нейронов и что все логические связки ("И", "ИЛИ", "НЕ" и т.д.) могут быть реализованы с помощью простых сетевых структур. Кроме того, Мак-Каллок и Питтс выдвинули предположение, что структурированные соответствующим образом сети способны к обучению. Дональд Хебб ([997], 1949) продемонстрировал простое правило обновления для модификации количества соединений между нейронами. Предложенное им правило, называемое теперь правилом ▶ **хеббовского обучения**, продолжает служить основой для моделей, широко используемых и в наши дни.

Два студента Гарварда, Марвин Мински и Дин Эдмондс, в 1950 году создали первый компьютер на основе нейронной сети. В этом компьютере, получившем название SNARC, для моделирования сети из 40 нейронов использовалось 3000 электронных ламп плюс механизм автопилота с бомбардировщика B-24. Позже, в Принстоне, Мински изучал возможности выполнения универсальных вычислений в нейронных сетях. Когда он защищал диссертацию доктора философии, аттестационная комиссия выразила сомнение, может ли работа такого рода рассматриваться как математическая, на что фон Нейман, по словам современников, возразил: "Сегодня — нет, но когда-то будет".

Существует большое количество примеров других ранних работ, которые можно охарактеризовать как относящиеся к искусственному интеллекту, в том числе две программы игры в шашки, разработанные независимо одна от другой в 1952 году Кристофером Стрейчем в Университете Манчестера и Артуром Сэмюэлом в IBM. Однако решающим оказалось мнение Алана Тьюринга. Начиная с 1947 года он читал лекции по этому вопросу в Лондонском математическом обществе и сформулировал убедительную программу исследований в статье *Вычислительные машины и интеллект* [2235], опубликованной в 1950 году. В этой статье он описал тест Тьюринга, принципы машинного обучения, генетические алгоритмы и обучение с подкреплением. Он рассмотрел многие возражения в отношении возможностей ИИ, как это описано в главе 27. Он также предположил, что создать ИИ на уровне человеческого интеллекта будет проще путем разработки алгоритмов обучения, с последующим обучением машины, а не прямым программированием ее интеллекта вручную. В последующих лекциях он предупреждал, что достижение этой цели может быть не самым лучшим решением для человечества.

В 1955 году Джон Маккарти уговорил Марвина Мински, Клода Шеннона и Натаниэля Рочестера помочь ему собрать всех американских исследователей, проявляющих интерес к теории автоматов, нейронным сетям и исследованиям

интеллекта. В конечном итоге они организовали двухмесячный семинар в Дартмуте летом 1956 года. Всего на нем присутствовало 10 участников, включая Аллена Ньюэлла и Герберта Саймона из Института Карнеги,[11] Тренчарда Мура из Принстона, Артура Сэмюэла из компании IBM, а также Рея Соломонова и Оливера Селфриджа из Массачусетсского технологического института. В приглашении говорилось следующее.[12]

> Мы предлагаем провести двухмесячное исследование искусственного интеллекта для 10 человек летом 1956 года в Дартмутском колледже в Ганновере, штат Нью-Гемпшир. Исследование будет проводиться на основании предположения, что каждый аспект обучения или любой другой функции интеллекта в принципе может быть описан настолько точно, что это позволит создать машину, способную его имитировать. Будут предприняты попытки выяснить, как разрабатывать машины, способные использовать язык, формулировать абстракции и концепции, решать различные задачи, в настоящее время доступные только людям, а также улучшать самих себя. Мы считаем, что можно добиться значительного прогресса в одном или нескольких из указанных направлений, если тщательно отобранная группа ученых будет совместно работать над этим в течение лета.

Несмотря на этот оптимистический прогноз, дартмутский семинар не привел к появлению каких-либо новых крупных открытий. Ньюэл и Саймон представили, вероятно, наиболее зрелую работу, систему доказательства математических теорем под названием Logic Theorist (LT). Саймон заявил: "Мы изобрели компьютерную программу, способную мыслить не численно, и тем самым решили освященную веками проблему разума и тела".[13] Вскоре после семинара эта программа смогла доказать большинство теорем из второй главы книги *Принципы математики* Рассела и Уайтхеда ([2331], 1910). Как говорили, Рассел был в восторге, когда ему сказали, что для одной из теорем LT смогла найти доказательство, которое оказалось короче, чем приведенное в его книге. Однако редактора журнала *Journal of Symbolic Logic* оказались менее впечатленными и отклонили статью, авторами которой были указаны Ньюэлл, Саймон и программа Logic Theorist.

---

[11] Теперь это учебное заведение называется "Университет Карнеги–Меллона" (Carnegie–Mellon University — CMU).

[12] Это было первое официальное использование принадлежащего Маккарти термина "искусственный интеллект". Может быть, вариант "вычислительная рациональность" был бы более точным и менее пугающим, но "ИИ" с тех пор прочно вошел в обиход. На 50-летии Дартмутской конференции Маккарти заявил, что он избегал терминов "компьютер" и "вычислительный" в знак уважения к Норберту Винеру, который в то время пропагандировал аналоговые кибернетические устройства, а не цифровые компьютеры.

[13] В процессе создания программы LT Ньюэлл и Саймон разработали также язык обработки списков IPL. У них не было компилятора, поэтому ученые переводили программы со своего языка в машинный код вручную. Чтобы избежать ошибок, они работали параллельно, называя друг другу двоичные числа после записи каждой команды, чтобы убедиться в том, что они совпадают.

1.3.2. Ранний энтузиазм, большие ожидания (1952–1969)

Интеллектуальный истеблишмент в 1950-х годах в своем большинстве продолжал считать, что "ни одна машина не сможет выполнить действие X". (Длинный список таких X, собранный Тьюрингом, приведен в главе 27.) Исследователи в области искусственного интеллекта, естественно, отвечали на это, демонстрируя способность машин решать одну задачу X за другой. Чаще всего они обращались к задачам, которые принято считать свидетельством интеллектуальности человека, — играм, головоломкам, математике, тестам IQ. Позднее Джон Маккарти назвал этот период эпохой "Смотри, мама, что я умею!"

За первой успешной разработкой Ньюэлла и Саймона, программой LT, последовало создание программы общего решателя задач (General Problem Solver — GPS). В отличие от программы Logic Theorist, эта программа с самого начала была предназначена для моделирования процедуры решения задач человеком. Как оказалось, в пределах того ограниченного класса головоломок, которые была способна решать эта программа, порядок, в котором она рассматривала подцели и возможные действия, был аналогичен тому подходу, который применяется людьми для решения таких же проблем. Поэтому программа GPS была, по-видимому, первой программой, в которой был воплощен подход к "организации мышления по такому же принципу, как и у человека". Результаты успешного применения GPS и последующих программ в качестве модели познания позволили сформулировать знаменитую гипотезу ▶**физической символической системы**, в которой утверждается, что существует "физическая символическая система, которая имеет необходимые и достаточные средства для интеллектуальных действий общего вида". Под этим подразумевается, что любая система, проявляющая интеллект (человек или машина), должна действовать по принципу манипулирования структурами данных, состоящими из символов. Ниже будет показано, что эта гипотеза во многих отношениях оказалась уязвимой для критики.

В компании IBM Натаниэль Рочестер и его коллеги создают некоторые из первых ИИ-программ. Герберт Гелернтер ([828], 1959) создал программу Geometry Theorem Prover, которая была в состоянии доказать такие геометрические теоремы, которые многие студенты-математики сочли бы достаточно сложными. Эта работа стала предшественницей современных систем доказательства математических теорем.

Из всех исследовательских работ, проделанных за этот период, возможно, самой влиятельной в долгосрочной перспективе стала разработка Артура Сэмюэла по игре в шашки. Благодаря использованию методов, которые мы сейчас называем обучением с подкреплением (см. главу 22), программы Сэмюэла в конечном итоге научились играть на уровне хорошо подготовленного любителя. Тем самым Сэмюэл опроверг утверждение, что компьютеры способны выполнять только то, чему их учили: одна из его программ быстро научилась играть лучше, чем ее создатель. Работа этой программы была показана на телевидении в 1956 году и произвела

очень сильное впечатление на зрителей. Как и Тьюринг, Сэмюэл с трудом находил машинное время. Работая по ночам, он использовал компьютеры, которые все еще находились на испытательной площадке производственного предприятия компании IBM. Программа Сэмюэла стала предшественницей более поздних систем, таких как TD-GAMMON (Тезауро [2192], 1992), прочно утвердившейся среди лучших игроков в нарды в мире, и ALPHAGO (Сильвер и др. [2063], 2016), потрясшей мир тем, что победила человека, чемпиона мира по игре в го (см. главу 5).

В 1958 году Джон Маккарти внес два важных вклада в развитие искусственного интеллекта. В документе MIT AI Lab Memo No. 1 он привел определение нового языка высокого уровня ▶ **Lisp**, которому суждено было стать доминирующим языком программирования для искусственного интеллекта в ближайшие 30 лет. В статье под названием *Программы со здравым смыслом* ([1529], 1958) он выдвинул концептуальное предложение о разработке систем искусственного интеллекта, основанных на знаниях и рассуждении. В этой статье он описал гипотетическую программу Advice Taker, способную воплощать общие знания о мире, а затем использовать их для выработки плана действий. Концепция была проиллюстрирована с помощью простых логических аксиом, которых было достаточно для разработки плана проезда в аэропорт. Также программа была задумана таким образом, чтобы принимать новые аксиомы в процессе работы, что позволяло ей достигать нужного уровня компетентности в других областях *без перепрограммирования*. Следовательно, в программе Advice Taker были воплощены центральные принципы представления знаний и проведения рассуждений: необходимо иметь формальное, детализированное представление о мире и его функционировании и уметь манипулировать этим представлением с помощью дедуктивных процедур. Данная статья оказала большое влияние на всю область ИИ, оставаясь вполне актуальной и сегодня.

Все тот же 1958 год также стал годом, когда Марвин Мински перешел в Массачусетсский технологический институт (МТИ). Но успешное его сотрудничество с Маккарти продолжалось недолго. Маккарти делал акцент на способах представления и проведении рассуждений в формальной логике, тогда как Мински в большей степени интересовался тем, как заставить программы работать, и в конечном итоге у него сформировалось отрицательное отношение к логике. В 1963 году Маккарти основал лабораторию искусственного интеллекта в Стандфордском университете. Разработанный им план использования логики для создания окончательной версии программы Advice Taker выполнялся быстрее, чем было задумано, благодаря открытию Дж. А. Робинсоном метода резолюции (полного алгоритма доказательства теорем для логики первого порядка; см. главу 9). Работы, выполненные в Стандфордском университете, подчеркнули важность применения методов общего назначения для проведения логических рассуждений. В число логических приложений вошли системы формирования ответов на вопросы и планирования Корделла Грина ([920], 1969), а также робототехнический проект Shakey, разрабатывавшийся в Стандфордском научно-исследовательском институте (Stanford Research

Institute — SRI). Последний проект, который подробно рассматривается в главе 26, впервые продемонстрировал полную интеграцию логических рассуждений и физической активности.

В МТИ Мински руководил работой ряда студентов, выбравших для себя небольшие задачи, для решения которых, как тогда казалось, требовалась интеллектуальность. Эти ограниченные проблемные области получили название ▶**микромиры**. Программа SAINT Джеймса Слэгла ([2083], 1963) была способна решать задачи интегрирования в закрытом исчислении, типичные для первых курсов колледжей. Программа ANALOGY Тома Эванса ([707], 1968) решала задачи выявления геометрических аналогий, используемые в тестах IQ. Программа STUDENT Дэниэла Боброва ([237], 1967) решала изложенные в виде рассказа алгебраические задачи, подобные приведенной ниже.

> Если количество заказов, полученных Томом, вдвое превышает квадратный корень из 20% опубликованных им рекламных объявлений, а количество этих рекламных объявлений равно 45, то каково количество заказов, полученных Томом?

Самым известным примером микромира был ▶**мир блоков**, состоящий из множества цельных блоков, размещенных на поверхности стола (или, чаще, имитации стола), как показано на рис. 1.3. Типичной задачей в этом мире является изменение расположения блоков определенным образом с использованием манипулятора-робота, который в каждый момент может захватывать только один блок. Мир блоков стал основой для проекта системы технического зрения Дэвида Хаффмена ([1091], 1971), работы по изучению зрения и удовлетворения ограничений Дэвида Уолтса ([2291], 1975), теории обучения Патрика Уинстона ([2362], 1970), программы понимания естественного языка Тэрри Винограда ([2361], 1972) и планировщика в мире блоков (Скотта Фалмана [709], 1974).

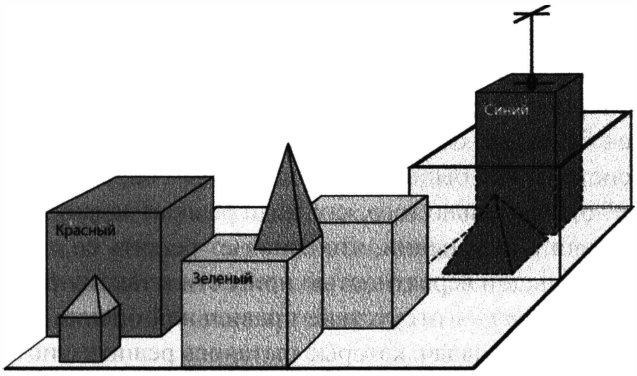


Рис. 1.3. Сцена из мира блоков. Программа SHRDLU Тэрри Винограда [2361] только что завершила выполнение команды "Найти блок, более высокий по сравнению с тем, который находится в манипуляторе, и поместить его в ящик"

Активно продвигались также исследования, основанные на ранних работах по созданию нейронных сетей Мак-Каллока и Питтса. В работе Шмуэля Винограда и Коуэна ([2360], 1963) было показано, как большое количество элементов в своей совокупности может представить отдельную концепцию с соответствующим увеличением надежности и степени распараллеливания. Методы обучения Хебба были усовершенствованы в работах Берни Видроу ([2335], 1960), ([2334], 1962), который называл свои сети **адалинами**, а также Френка Розенблатта ([1912], 1962), создателя **перцептронов**. Розенблатт доказал **теорему сходимости перцептрона**, которая подтверждает, что предложенный им алгоритм обучения позволяет корректировать количество соединений перцептрона в соответствии с любыми входными данными при условии, что такое соответствие существует.

1.3.3. Столкновение с реальностью (1966–1973)

С самого начала исследователи ИИ не стеснялись делать прогнозы в отношении своих будущих успехов. Так, в свое время часто цитировалось приведенное ниже предсказание Герберта Саймона, опубликованное им в 1957 году.

> Я не ставлю перед собой задачу удивить или шокировать вас, но проще всего я могу подвести итог, сказав, что теперь мы живем в мире машин, которые думают, учатся и создают. Более того, их способность выполнять эти действия будет продолжать быстро расти до тех пор, пока — в обозримом будущем — круг проблем, с которыми они смогут справиться, можно будет сопоставить с кругом проблем, в которых до сих пор был нужен человеческий интеллект.

Такое выражение, как "обозримое будущее", довольно расплывчато, но Саймон сделал также более конкретные прогнозы, что через десять лет компьютер станет чемпионом мира по шахматам и что машиной будет доказана важная математическая теорема. Эти предсказания сбылись (или почти сбылись), но не через десять, а через сорок лет. Излишняя самоуверенность Саймона была обусловлена тем, что первые системы искусственного интеллекта демонстрировали многообещающую производительность, хотя и на простых примерах. Однако эти ранние системы почти всегда терпели поражение, сталкиваясь с более трудными проблемами.

У этих неудач были две основные причины. Первая заключалась в том, что многие ранние системы ИИ были основаны главным образом на "информированном самоанализе" относительно того, как люди решают задачу, а не на тщательном анализе самой задачи и выяснении, что означает решить ее и что должен делать алгоритм, чтобы с большой вероятностью привести к такому решению.

Вторая причина неудач — отсутствие правильной оценки сложности для машинной обработки многих задач, которые пытались решить с помощью ИИ. Большинство ранних систем решения проблем работали по методу проверки различных комбинаций возможных шагов, пока решение не будет найдено. Сначала эта стратегия успешно работала, поскольку микромиры содержали очень мало объектов, а следовательно, количество возможных действий было невелико и к решению

приводили относительно короткие их последовательности. До того, как была разработана теория вычислительной сложности, господствовало представление, что для больших проблем "масштабирование вверх" было просто вопросом повышения быстродействия оборудования и увеличения объема доступной памяти. Например, оптимизм, сопровождавший разработку систем доказательства теорем, очень быстро улетучился, когда исследователям не удалось добиться доказательства теорем, включающих более нескольких десятков фактов. ➔ *Тот факт, что программа может найти решение в принципе, вовсе не означает, что эта программа уже содержит все механизмы, необходимые для его поиска на практике.*

Иллюзия неограниченности вычислительной мощности касалась не только программ решения задач. Ранние эксперименты в области ▶**машинной эволюции** (теперь она называется **генетическое программирование**) строились на, несомненно, правильном убеждении, что внесение соответствующего ряда небольших изменений в машинный код программы позволит создать эффективную программу решения любой конкретной задачи. Общая идея состояла в том, что необходимо проверять случайные мутации (изменения в коде) с отбором для сохранения мутаций, которые кажутся полезными. Несмотря на тысячи часов потраченного процессорного времени, почти никаких признаков прогресса не было продемонстрировано.

Неспособность справиться с "комбинаторным взрывом" стала одним из основных критических замечаний в адрес ИИ, содержащихся в отчете Лайтхилла ([1412], 1973), что послужило основанием для британского правительства принять решение о прекращении поддержки исследований в области искусственного интеллекта во всех университетах, кроме двух. (Устная традиция рисует в отношении этого решения несколько иную и более красочную картину, с политическими амбициями и личной неприязнью, описание которых выходит за рамки данного курса.)

Третий вид затруднений возник в связи с некоторыми фундаментальными ограничениями базовых структур, использовавшихся в процедурах выработки интеллектуального поведения. Например, Мински и Паперт в книге *Перцептроны* ([1585], 1969) доказали, что перцептроны (простая форма нейронной сети) могут продемонстрировать способность изучить все, что возможно представить с их помощью, но, к сожалению, они позволяют представить лишь очень немногое. В частности, перцептрон с двумя входами не может быть обучен распознавать ситуацию, когда на два его входа поданы разные сигналы. Хотя полученные ими результаты не применимы к более сложным, многослойным сетям, финансирование исследований в области нейронных сетей вскоре сократилось почти до нуля. Как это ни странно, но те новые алгоритмы обучения путем обратного распространения для многослойных сетей, которые стали причиной возрождения необычайного интереса к исследованиям в области нейронных сетей в конце 1980-х годов, фактически впервые были открыты (но в другом контексте) уже в начале 1960-х годов (Келли [1214], 1960; Брайсон [334], 1962).

1.3.4. Экспертные системы (1969–1986)

Основная методология решения проблем, выработанная в течение первого десятилетия исследований в области ИИ, заключалась в использовании поискового механизма общего назначения, посредством которого предпринимались попытки связать воедино элементарные рассуждения так, чтобы отыскать полные решения. Такие подходы получили название ▶ **слабые методы**, поскольку, даже будучи достаточно общими, они не масштабируются до уровня больших или сложных проблем. Альтернативой слабых методов является использование более содержательных, предметно-ориентированных знаний, что позволяет строить более длинные цепочки рассуждений и дает возможность проще справиться с теми проблемными ситуациями, которые обычно возникают в специализированных областях знаний. Можно сказать, что для решения сложной проблемы требуется уже почти знать полный ответ.

Одним из первых примеров реализации такого подхода была программа DENDRAL ([339], 1969), разработанная в Стандфордском университете группой ученых, в которую вошли Эд Фейгенбаум (бывший студент Герберта Саймона), Брюс Бьюкенен (философ, который сменил специальность и стал заниматься компьютерными науками) и Джошуа Ледерберг (лауреат Нобелевской премии в области генетики). Эта группа занималась решением проблемы определения структуры молекул на основе информации, полученной от масс-спектрометра. Вход этой программы состоял из химической формулы соединения (например, $C_6H_{13}NO_2$) и спектра масс, позволяющего определять массы различных фрагментов молекулы, который формировался при бомбардировке молекулы потоком электронов. Например, спектр масс может содержать пик в точке $m = 15$, соответствующий массе метилового фрагмента (CH_3).

Первая, простейшая версия этой программы предусматривала генерацию всех возможных структур, совместимых с данной формулой, после чего предсказывала, какой спектр масс должен наблюдаться для каждой из этих структур, сравнивая его с фактическим спектром. Вполне можно ожидать, что такая задача применительно к молекулам более крупных размеров становится практически неразрешимой. Поэтому разработчики программы DENDRAL проконсультировались с химиками-аналитиками и пришли к выводу, что следует попытаться организовать работу по принципу поиска широко известных картин расположения пиков в спектре, которые указывают на наличие общих подструктур в молекуле. Например, для распознавания кетоновых подгрупп (>C=O) с атомной массой 28 имеем следующее.

if M — масса всей молекулы, и есть два пика в x_1 и x_2, таких, что
(a) $x_1 + x_2 = M + 28$; (b) $x_1 - 28$ высокий пик; (c) $x_2 - 28$ высокий пик; и
(d) по крайней мере один из x_1 и x_2 высокий
then это подгруппа кетона.

Выяснение того, что молекула содержит определенную подструктуру, чрезвычайно сокращает число возможных кандидатов. По мнению авторов, программа

DENDRAL стала мощным инструментом, поскольку в нее встроены соответствующие знания, полученные средствами масс-спектроскопии, но не в форме базовых принципов, а в виде эффективных "кулинарных рецептов" (Фейгенбаум и др. [716], 1971). Значение программы DENDRAL определялось тем, что это была первая *успешная* система с интенсивным использованием знаний: ее анализ был построен на использовании большого количества специальных правил. В 1971 году Фейгенбаум и его коллеги в Станфорде приступили к новому проекту эвристического программирования с целью исследования, в какой степени их новая методология построения ▶**экспертных систем** может применяться в других областях.

Следующей крупной попыткой стала разработка системы MYCIN для диагностики инфекции в крови. Исходя из примерно 450 правил, программа MYCIN оказалась способной решать поставленную задачу на уровне некоторых экспертов и заметно лучше молодых врачей. Но у нее было два больших отличия от системы DENDRAL. Во-первых, в отличие от правил DENDRAL, не существует общей теоретической модели, из которой можно было бы вывести правила для MYCIN. Их можно было выделить только из результатов обширного опроса экспертов. Во-вторых, правила должны были отражать неопределенность, связанную с медицинскими знаниями. В MYCIN были включены подпрограммы вычисления неопределенности, называемой ▶**факторами уверенности** (см. главу 13), которые, как казалось на то время, хорошо согласуются с тем, как врачи оценивали влияние отдельных показателей на окончательный диагноз.

Первая успешная коммерческая экспертная система, R1, была развернута в компании DEC (Digital Equipment Corporation) ([1546], 1982). Эта программа помогала составлять конфигурации для выполнения заказов на новые компьютерные системы. К 1986 году она обеспечивала компании DEC экономию примерно 40 миллионов долларов в год. К 1988 году группой искусственного интеллекта компании DEC было развернуто 40 экспертных систем, а в дальнейшем предусматривалось развернуть еще больше. В компании DuPont применялось 100 систем, а в разработке находилось еще 500. Почти в каждой крупной корпорации США была создана собственная группа искусственного интеллекта и экспертные системы либо использовались, либо исследовались.

Важность знания предметной области была также совершенно очевидна в области понимания естественного языка. Несмотря на успех системы SHRDLU Винограда, ее методы нельзя было распространить на более общие задачи. Для решения таких проблем, как разрешение неоднозначности, в ней использовались совсем простые правила, которых, тем не менее, было вполне достаточно для крошечных пределов мира блоков.

Ряд исследователей, в том числе Юджин Чарняк в МТИ и Роджер Шенк в Йельском университете, выдвинули предположение, что уверенное понимание языка потребует общих знаний о мире и некоторого общего метода использования этих знаний. (Шенк пошел дальше, заявив, что "Синтаксиса не существует", и это огорчило многих лингвистов, но послужило началом полезного обсуждения.)

Шенк и его студенты создали серию программ (1978–1981), предназначенных для решения одной и той же задачи — понимания естественного языка. Однако больше внимания при этом уделялось не собственно языку, а проблемам представления знаний и рассуждений, необходимых для его понимания.

Быстро возрастающее количество коммерческих приложений, предназначенных для решения практических задач, потребовало разработки широкого диапазона инструментальных средств представления знаний и рассуждений. Одни из них были основаны на логике, например язык Prolog, ставший популярным в Европе и Японии, или пакет PLANNER в США. В других, следуя идее Мински о ▶**фреймах** ([1582], 1975), был принят более структурированный подход, предполагающий сбор фактов о конкретных типах объектов и событий, с последующим упорядочиванием типов в большую таксономическую иерархию, аналогичную биологической таксономии.

В 1981 году правительство Японии объявило о развертывании проекта "Пятое поколение" — 10-летнего плана по созданию интеллектуальных компьютеров с массовым параллелизмом, работающих под управлением языка Prolog. Бюджет проекта предполагался в размере более 1,3 млрд. долларов по сегодняшнему курсу. В ответ Соединенные Штаты создали корпорацию по микроэлектронике и компьютерным технологиям (MCC) — консорциум, призванный обеспечить стране национальную конкурентоспособность. В обоих случаях область ИИ была частью широкого спектра направлений, включавшего разработку микросхем и исследование интерфейса "человек–машина". В Великобритании Олви в своем отчете потребовал восстановления финансирования тех направлений, финансирование которых было прекращено на основании отчета Лайтхилла. Однако ни один из этих проектов так и не достиг своих амбициозных целей как с точки зрения новых возможностей ИИ, так и с точки зрения экономического эффекта.

В целом в индустрии искусственного интеллекта наблюдался бурный рост, начиная с нескольких миллионов долларов в 1980 году и до миллиарда долларов в 1988 году. Были созданы сотни компаний, разрабатывавших экспертные системы, системы технического видения, роботов, а также специализированное программное обеспечение и аппаратные средства для этих целей.

Однако вскоре после этого наступил период, получивший название "зима искусственного интеллекта", когда многие вновь созданные компании сильно пострадали, поскольку не сумели выполнить свои заманчивые обещания. Оказалось, что создание и поддержка экспертных систем в сложных предметных областях — задача очень сложная, отчасти потому, что используемые в этих системах методы рассуждения отказывали в условиях неопределенности, а отчасти потому, что эти системы были неспособны извлекать уроки из накопленного опыта.

1.3.5. Возвращение к нейронным сетям (1986–настоящее время)

К середине 1980-х годов по меньшей мере четыре разные группы исследователей независимо заново открыли алгоритм обучения путем обратного распростра-

нения, впервые предложенный в начале 1960-х. Этот алгоритм был применен для решения многих проблем обучения в компьютерных науках и психологии, а после публикации результатов его использования в сборнике статей *Распределенная параллельная обработка* ([1933], 1986) привлек всеобщее внимание.

Эти так называемые ▶ **коннекционистские** (основанные на соединениях) модели рассматривались многими как непосредственно конкурирующие и с символическими моделями, продвигаемыми Ньюэллом и Саймоном, и с логицистским подходом, предложенным Маккарти и др. Кажется вполне очевидным, что на каком-то уровне мышления люди манипулируют символами. И действительно, антрополог Терренс Дикон в своей книге под названием *Символические виды* ([563], 1997) указал, что это *определяющая характеристика* человека. В противовес этому Джефф Хинтон, ведущий деятель в возрождении нейронных сетей в 1980- и 2010-х годах, назвал символы "светоносным эфиром искусственного интеллекта" — метафорически ссылаясь на несуществующую физическую среду, по которой якобы распространяются электромагнитные волны в представлении физиков начала XIX века. Безусловно, многие концепции, которые можно выделить в языке, при ближайшем рассмотрении не обладают теми или иными логически определенными необходимыми и достаточными состояниями, которые первые исследователи ИИ предполагали сформулировать в аксиоматической форме. Возможно, коннекционистские модели формируют внутренние концепции более гибким и неточным образом, что лучше подходят для беспорядочности реального мира. Они также способны учиться на примерах, сравнивая прогнозируемое ими значение выходного сигнала с истинным значением в решаемой задаче, и изменять свои параметры так, чтобы уменьшить различие, таким образом делая более вероятным получение лучших решений для будущих примеров.

1.3.6. Вероятностные рассуждения и машинное обучение (1987–настоящее время)

Хрупкость создаваемых экспертных систем привела к использованию нового, более широкого научного подхода, включающего вероятности вместо чистой булевой логики, машинное обучение вместо ручного кодирования и экспериментальные результаты вместо философских рассуждений.[14] Теперь обычной практикой стало использование уже существующих теорий вместо предложения совершенно новых, построение вновь выдвигаемых положений на строгих теоремах или твердой экспериментальной методологии, а не на интуиции, и демонстрация приложений, актуальных для реального мира, вместо игрушечных примеров.

[14] Некоторые характеризуют это изменение как победу "чистюль" — тех, кто считает, что теории искусственного интеллекта должны основываться на математической строгости, — над "неряхами" — теми, кто предпочел бы опробовать множество идей, написать несколько программ, а затем оценить то, что кажется работающим. Оба подхода важны. Сдвиг в сторону математической точности подразумевает, что область исследований достигла уровня стабильности и зрелости. Нынешний акцент на глубоком обучении может представлять собой возрождение значимости экспериментаторов.

Совместно используемые эталонные наборы задач стали нормой для демонстрации достигнутого прогресса. Среди них — репозиторий UC Irvine для наборов данных машинного обучения, набор International Planning Competition для проверки алгоритмов планирования, корпус LibriSpeech для задач распознавания речи, набор данных MNIST для распознавания рукописных цифр, наборы ImageNet и COCO для распознавания изображений объектов, набор SQuAD для ответов на вопросы на естественном языке, комплект WMT для машинного перевода и набор International SAT Competitions для проверки выполнимости булевых формул.

В свое время область ИИ появилась отчасти как протест против ограничений в таких уже существовавших областях исследований, как теория управления и статистика. Но в этот период искусственный интеллект включил в себя и положительные результаты из этих областей. Вот что сказал Дэвид Макаллестер ([1523], 1998).

> В ранний период развития ИИ казалось вполне вероятным, что новые формы символических вычислений, например фреймы и семантические сети, сделают основную часть классической теории устаревшей. Это привело к определенной форме самоизоляции, отчего искусственный интеллект в значительной степени отделился от остальной части компьютерных наук. В настоящее время этот изоляционизм преодолен. Появилось признание того, что машинное обучение не следует отделять от теории информации, что проведение рассуждений в условиях неопределенности нельзя изолировать от стохастического моделирования, что поиск не следует рассматривать отдельно от классической оптимизации и управления и что автоматические рассуждения нельзя отделять от формальных методов и статистического анализа.

Эту тенденцию особенно хорошо иллюстрирует область распознавания речи. В 1970-е годы было опробовано большое разнообразие различных архитектур и подходов. Многие из них оказались узко специализированными, надуманными и работали только на нескольких специально отобранных примерах. В 1980-е годы доминирующие позиции в этой области заняли подходы, основанные на использовании ▶**скрытых марковских моделей** (Hidden Markov Model — HMM). Решающую роль здесь сыграли две их особенности. Во-первых, эти модели основаны на строгой математической теории, и это позволило исследователям речи опереться на математические результаты, накопленные в других областях за несколько десятилетий. Во-вторых, они генерируются в процессе обучения программ на крупном массиве реальных речевых данных. Это гарантирует достижение надежных показателей производительности, а в строгих слепых испытаниях модели HMM неизменно улучшают свои показатели. В результате технологии обработки речи и связанная с ними область распознавания рукописных символов смогли совершить переход к широко распространенным промышленным и массовым потребительским приложениям. Обратите внимание, что прежде не было никаких научных утверждений о том, что люди используют HMM для распознавания речи. Эти модели лишь предоставляли математической основу для понимания и решения задач. Однако в разделе 1.3.8 будет показано, что методы глубокого обучения способны в скором времени нарушить эту идиллическую картину.

Год 1988 стал важным годом в отношении установления связей между ИИ и другими научными областями, в том числе статистикой, исследованием операции, теорией принятия решений и теорией управления, методами изучения байесовских сетей и связанных с ними моделей на основе данных. Книга Джуды Перла *Вероятностные рассуждения в интеллектуальных системах* ([1755], 1988) послужила толчком к признанию важности теории вероятностей и теории принятия решений для области искусственного интеллекта. Перл разработал новый подход — модель ▶**байесовских сетей**, обеспечивающей строгий и эффективный формальный подход к представлению нечетко определенных знаний, а также практический алгоритм для вероятностных рассуждений. Эти темы подробно рассматриваются в главах 12–16 вместе с результатами более поздних разработок, позволивших существенно увеличить выразительную мощь вероятностных методов. В главе 20 описываются методы обучения байесовских сетей и связанные с ними модели данных.

Вторым важным итогом 1988 года стала работа Рича Саттона, связавшая метод обучения с подкреплением — этот подход уже использовался в 1950-х годах Артуром Сэмюэлом в его программе игры в шашки — с марковским процессом принятия решений (Markov Decision Processes — MDP), разработанным ранее в рамках теории исследования операций. Последовал целый поток работ по связыванию исследований по планированию ИИ с MDP. С другой стороны, методы обучения с подкреплением нашли применение в области робототехники и управления процессами, одновременно получив глубокое теоретическое обоснование.

Одним из последствий переоценки в области ИИ значения данных, статистического моделирования, оптимизации и машинного обучения стало постепенное воссоединение с такими подобластями, как компьютерное зрение, робототехника, распознавание речи, многоагентные системы и обработка естественного языка, которые к этому времени уже успели несколько отдалиться от основного ядра области искусственного интеллекта. Процесс реинтеграции дал существенное преимущество как в отношении приложений — например, развертывание новых работ по практическому созданию роботов значительно возросло за этот период, — так и в отношении углубления теоретического понимании основных проблем ИИ.

1.3.7. Большие данные (2001–настоящее время)

Замечательные достижения в области увеличения вычислительной мощности и создание Всемирной паутины, World Wide Web, способствовали созданию очень больших наборов данных — явление, сейчас известное, как ▶**большие данные**. Эти наборы данных включают в себя триллионы слов текста, миллиарды изображений и миллиарды часов записей речи и видео, а также колоссальное количество геномных данных, сведений по отслеживанию транспортных средств, информации о посещениях веб-страниц, данных социальных сетей и т.д.

Все это требовало разработки обучающих алгоритмов, специально предназначенных для извлечения преимуществ, даваемых очень большими наборами

данных. Очень часто подавляющее большинство экземпляров данных в таких наборах никак не помечены. Например, в своей известной работе по многозначности слов Яровский ([2400], 1995) указывает, что отдельные вхождения слов, таких как "лист", обычно никак *не маркированы*, чтобы указать, относятся они к флоре, книгоиздательству или материаловедению. Однако при достаточно больших наборах данных подходящие алгоритмы обучения позволяют достичь правильных решений в 96% процентах случаев, исходя из того, какой смысл имеет предложение в целом. Более того, Банко и Брилл ([122], 2001) утверждают, что улучшение в производительности, полученное от увеличения набора данных в размере на два или три порядка по величине перевешивает какие-либо улучшения, которые могут быть получены в результате тонкой настройки алгоритма.

Аналогичный феномен, кажется, имеет место в некоторых задачах компьютерного зрения, таких как заполнение в фотографиях пустых мест, вызванных либо случайными повреждениями, либо такими действиями, как "удаление" бывших друзей. Хейс и Эфрос ([993], 2007) разработали более умный способ достичь этой цели посредством наложения пикселей из аналогичных изображений. Они обнаружили, что этот метод плохо работал на базе данных из нескольких тысяч изображений, но позволял достичь нужного качества при работе с миллионами изображений. Вскоре после появления в базе данных ImageNet десятков миллионов изображений в области компьютерного зрения произошла настоящая революция (Денг и др. [602], 2009).

Доступность больших данных и переход к машинному обучению способствовали восстановлению коммерческого интереса к области искусственного интеллекта (Хавенстейн [986], 2005; Галеви и др. [950], 2009). Большие данные стали решающим фактором, когда в 2011 году система Watson компании IBM добилась победы на людьми — чемпионами викторины Jeopardy! Это событие оказало большое влияние на восприятие искусственного интеллекта широкой публикой.

1.3.8. Глубокое обучение (2011–настоящее время)

Термин ▶ **глубокое обучение** относится к машинному обучению с использованием нескольких слоев из простых, регулируемых вычислительных элементов. Эксперименты с такими сетями проводились еще в 1970-х годах, и в форме **сверточных нейронных сетей** они имели некоторый успех в распознавании рукописных цифр в 1990-х годах (ЛеКун и др. [1373], 1995). Однако только в 2011 году методы глубокого обучения действительно получили признание. Сначала это произошло в области распознавания речи, а затем и в области распознавания визуальных объектов.

В 2012 году на конкурсе ImageNet, где требуется классифицировать предлагаемые изображения в одну из тысяч категорий (броненосцы, книжная полка, штопор и т.д.), система глубокого обучения, созданная группой Джеффри Хинтона в Университете в Торонто (Крижевски и др. [1313], 2013) продемонстрировала

значительное улучшение в сравнении с предыдущими системами, построенными главным образом на написанных вручную функциях. С тех пор системы глубокого обучения смогли превзойти возможности человека в решении некоторых визуальных задач (но пока отстают в решении некоторых других задач). Подобные успехи были отмечены и в областях распознавания речи, машинного перевода, постановки медицинского диагноза и компьютерных игр. Использование сети глубокого обучения для представления функции оценки внесло свой вклад в победу программы AlphaGo над ведущими игроками в го со всего мира (Сильвер и др. [2063–2065], 2016–2018).

Эти замечательные успехи привели к возрождению интереса к искусственному интеллекту среди студентов, компаний, инвесторов, правительств, средств массовой информации и широкой общественности. Складывается впечатление, что почти каждую неделю появляются сообщения о том, что очередное новое приложение ИИ смогло приблизиться к человеческим способностям или даже превзойти их. Все это обычно сопровождается очередными спекуляциями либо о безудержном росте достижений, либо о приближении новой зимы ИИ.

Глубокое обучение в значительной степени зависит от мощного оборудования. Если стандартный процессор персонального компьютера способен выполнять от 10^9 до 10^{10} операций в секунду, алгоритм глубокого обучения, работающий на специализированном оборудовании (таком, как GPU, TPU или FPGA), может требовать выполнения от 10^{14} до 10^{17} операций в секунду, в основном в виде сильно распараллеленных матричных или векторных операций. Конечно, глубокое изучение также зависит от доступности в необходимом (большом) количестве данных для обучения, а кроме того, от нескольких алгоритмических трюков (см. главу 21).

1.4. Современное состояние исследований

Проект Стандфордского университета One Hundred Year Study on AI (также известный как AI100) предусматривает регулярный созыв групп экспертов с целью предоставления докладов о текущем состоянии дел в области искусственного интеллекта. В отчете за 2016 год [2338] был сделан вывод, что "можно ожидать существенного увеличения применения приложений ИИ в ближайшем будущем, включая самоуправляемые автомобили, медицинские системы диагностики и таргентной терапии, а также средства оказания физической поддержки в уходе за престарелыми" и что "общество в настоящее время переживает решающий момент в определении того, как развертывать технологии искусственного интеллекта таким образом, чтобы способствовать, а не препятствовать демократическим ценностям, таким как свобода, равенство и прозрачность". В рамках проекта AI100 на сайте `aiindex.org` также ведется ▶**Индекс ИИ** (AI Index) с целью помочь отслеживать прогресс в этой области. Ниже приведены некоторые основные моменты из отчетов за 2018 и 2019 годы (в сравнении с уровнем 2000 года, принятым как исходный, если не указано иное).

- *Публикации.* Количество публикаций в области ИИ за период с 2010 по 2019 год увеличилось в 20 раз и составило около 20 тысяч в год. Самой популярной категорией было машинное обучение. (На сайте `arXiv.org` в 2009–2017 годы количество статей на тему машинного обучения ежегодно удваивалось.) Следующим по популярности были категории компьютерного зрения и обработки естественных языков.
- *Отношение.* Около 70% новых статей по ИИ имеют нейтральный тон, но количество статей с положительным тоном увеличилось с 12% в 2016 году до 30% в 2018. В большинстве случаев вызывающие беспокойство проблемы носят этический характер: конфиденциальность данных и необъективность алгоритмов.
- *Количество студентов.* В сравнении 2010 годом количество зачисленных на курсы ИИ увеличилось в 5 раз в США и в 16 раз в масштабе всего мира. В настоящее время ИИ — самая популярная специализация в области компьютерных наук.
- *Соотношение полов.* В масштабе всего мира среди профессоров ИИ около 80% мужчин и 20% женщин. Аналогичные цифры имеют место для научных работников, студентов и наемных работников в данной отрасли.
- *Конференции.* С 2012 года посещаемость конференций NeurIPS увеличилась на 800% и достигла уровня 13 500 человек. Для других конференций в области ИИ наблюдается ежегодный рост количества участников — около 30%.
- *Промышленность.* В США количество стартапов в области ИИ выросло в 20 раз, их общее число уже превысило 800.
- *Интернационализация.* В Китае за год публикуется больше статей, чем в США, и примерно столько же, сколько во всей Европе. Однако при оценке публикаций посредством взвешенного показателя цитируемости влияние американских авторов на 50% больше, чем китайских. Сингапур, Бразилия, Австралия, Канада и Индия демонстрируют наиболее быстрый рост количества наемных работников в области ИИ.
- *Компьютерное зрение.* Показатель количества ошибок в распознавании объектов (по результатам конкурса ImageNet LSVRC) улучшился с 28% в 2010 году до 2% в 2017 году, превысив возможности человека. Точность ответов в тестах с открытыми визуальными вопросами (Visual Question Answering — VQA) с 2015 года улучшилась с 55% до 68%, но пока еще отстает от эффективности работы человека — 83%.
- *Скорость работы.* Время обучения для задачи распознавания изображений сократилось в 100 раз за последние два года. Количество вычислительной мощности, используемой в лучших приложениях ИИ, удваивается каждые 3,4 месяца.
- *Обработка языка.* За период с 2015 по 2019 год точность и полнота ответов на вопросы, измеренная в показателях F1 на основании набора данных

Stanford Question Answering Dataset (SQuAD), увеличилась от 60 до 95. На наборе тестовых данных SQuAD2 прогресс оказался больше, с 62 до 90 всего за один год. Оба показателя превышают показатели человеческого уровня.

- *Сравнение с возможностями человека.* Как сообщается, к концу 2019 года системы искусственного интеллекта достигнут или превысят возможности человека по игре в шахматы, го, покер и компьютерную игру PacMan, в викторине Jeopardy! и выполнении тестов ImageNet по распознаванию объектов, в способности распознавания речи в ограниченных областях и перевода с китайского языка на английский в ограниченных областях, в компьютерных играх Quake III, Dota 2, StarCraft II, различных играх игровой приставки Atari, в диагностике рака кожи и простаты, диагностике проблем со сворачиванием белка и диабетической ретинопатии.

Когда (если когда-либо) системы ИИ достигнут производительности на уровне человека при решении широкого круга задач? В проведенных в 2018 году интервью с экспертами в области ИИ был предложен широкий диапазон прогнозируемых дат — от 2029 до 2200 года со средним значением 2099. В аналогичном опросе, проведенном в 2017 году, 50% респондентов считали, что это может произойти к 2066 году, хотя 10% полагали, что это может произойти уже в 2025 году, а некоторые даже ответили "никогда". Мнения экспертов также разделились в отношении того, потребуются ли нам новые фундаментальные прорывы или будет достаточно лишь простого уточнения и развития уже существующих подходов. Однако не следует принимать эти предсказания слишком серьезно, — как показал в своей работе Филипп Тетлок ([2195], 2017), в области прогнозирования мировых событий эксперты ничем не лучше, чем просто любители.

Как будут работать будущие системы ИИ? Мы пока не можем этого сказать. Как подробно говорилось в этом разделе, в данной научной области за время ее развития сменилось несколько ведущих концепций: в основу была положена смелая мысль, что создать машинный интеллект вообще возможно, при этом принималось, что поставленная цель может быть достигнута путем кодирования экспертных знаний в логические построения. На смену этому пришла уверенность, что главным инструментом могли бы стать вероятностные модели мира, уступившие в последние годы свои позиции убежденности, что машинное обучение позволяет получать такие модели, которые вообще не могут быть созданы на основании любой хорошо изученной теории. Будущее покажет, какая модель будет следующей.

Что ИИ может делать сегодня? Возможно, не так много, как пытались нас убедить авторы некоторых из наиболее оптимистичных статей в СМИ, но все же очень многое. Вот некоторые примеры.

Роботизированные транспортные средства. Отсчет истории роботизированных средств передвижения следует начинать с радиоуправляемых автомобилей 1920-х годов, но первые демонстрации автономного движения по дороге

без специальных направляющих имели место лишь в 1980-х годах (Кенаде и др. [1177], 1986; Дикманс и Зэпп [618], 1987). После успешных демонстраций вождения по грунтовым дорогам на 132-мильном ралли DARPA Grand Challenge в 2005 году (Трун [2212], 2006) и на улицах с дорожным движением в рамках ралли Urban Challenge 2007 года началась настоящая гонка по разработке самоуправляющихся автомобилей. В 2018 году тестовые автомобили компании Waymo преодолели рубеж в 10 миллионов миль езды на дорогах общего пользования без серьезных аварий, при этом человеку-наблюдателю приходилось брать управление на себя только один раз за каждые 6000 миль. Вскоре после этого компания начала предлагать коммерческие услуги роботизированного такси.

В воздухе автономные беспилотники с неподвижным крылом использовались в Руанде для доставки крови через всю страну начиная с 2016 года. Квадрокоптеры автономно выполняли примечательные акробатические маневры, исследуя здания в процессе создания 3-D карт, и самостоятельно собирались в автономные группы.

Перемещение на ногах. Четвероногий робот BigDog, созданный Райбертом и др. ([1846], 2008), перевернул наши представления о том, как роботы могут двигаться, — это уже не походка роботов из голливудских фильмов (медленно, на негнущихся ногах, раскачиваясь из стороны в сторону), а что-то очень похожее на движение животного, способного восстановить равновесие и продолжить движение после толчка или поскользнувшись на замерзшей луже. Атлас, человекоподобный робот, может не только совершать прогулки по неровной местности, но и запрыгивать на ящики и даже делать сальто (Акерман и Джиццо [11], 2016).

Автономное планирование и составление расписаний. Работающая на удалении в сотни миллионов километров от Земли программа Remote Agent агентства NASA стала первой бортовой автономной программой планирования, предназначенной для управления процессами составления расписания операций для космического аппарата (Джонсон и др. [1148], 2000). Программа Remote Agent вырабатывала планы на основе целей высокого уровня, задаваемых с Земли, а также контролировала работу космического аппарата в ходе выполнения этих планов — обнаруживала, диагностировала и устраняла проблемы по мере их возникновения. Сегодня инструментарий планирования EUROPA (Беррейро и др. [133], 2012) используется для планирования рядовых операций марсоходов НАСА, а система SEXTANT (Винтерниц [2364], 2017) обеспечивает автономную навигацию в глубоком космосе, за пределами действия глобальной GPS-системы.

Во время кризиса в Персидском заливе в 1991 году американские войска развернули систему динамического анализа и перепланирования DART (Кросс и Уолкер [503], 1994) с целью автоматизации планирования перевозок и графиков движения для транспорта. Она обрабатывала данные примерно о 50 тысячах автомобилей, единиц грузов и людей одновременно, принимая во внимание их отправные точки, места назначения, маршруты, любые транспортные мощности, возможности портов и аэродромов, разрешая возникающие конфликты по всем параметрам. Управление оборонных проектов США (Defense Advanced Research

Agency Project — DARPA) впоследствии заявило, что одно это приложение с избытком окупило все инвестиции этого ведомства в ИИ за 30 лет.

Ежедневно компании по перевозкам пассажиров и грузов, подобные Uber, и картографические сервисы, такие как Google Maps, обеспечивают эффективный проезд к заданному пункту назначения сотням миллионов пользователей, быстро прокладывая оптимальный маршрут с учетом текущего и *прогнозируемого* трафика по всему пути движения.

Машинный перевод. В настоящее время интерактивные системы машинного перевода позволяют работать с документами более чем на ста языках, в число которых входят родные языки более чем 99% населения Земли. Они переводят сотни миллиардов слов в день для сотен миллионов пользователей. Хотя работа этих систем еще не совсем доскональна, в целом предоставляемый ими перевод вполне адекватен для понимания. Для близкородственных языков (таких, как французский и английский) и при довольно большом объеме обучающих данных перевод в достаточно узкой области приближается к уровню перевода человеком (Ву и др. [2392], 2016).

Распознавание речи. В 2017 году компания Microsoft показала, что ее система распознавания разговорной речи достигла уровня ошибок в словах в пределах 5,1%, что соответствует результатам обычного человека при выполнении тестовой задачи Switchboard, заключающейся в расшифровке телефонных разговоров (Сионг и др. [2394], 2017). Около трети всех компьютерных коммуникаций во всем мире в настоящее время ведется голосовыми данными, а не текстовыми сообщениями, набираемыми с помощью клавиатуры. Приложение Skype обеспечивает перевод речи в режиме реального времени на десять языков. Компании Alexa, Siri, Cortana и Google предлагают программных виртуальных помощников, способных отвечать на вопросы и выполнять поставленные пользователем задания. Например, служба Google Duplex использует средства распознавания и синтеза речи для выполнения бронирования мест в ресторанах по запросу пользователей, обеспечивая поддержку необходимых переговоров от их имени.

Рекомендации, целевая реклама. Такие компании, как Amazon, Facebook, Netflix, Spotify, YouTube, Walmart и другие, используют машинное обучение, чтобы предоставить пользователю информацию о том, что ему может понравиться, исходя из его прошлого опыта и опыта других, подобных ему. Сегмент таких систем имеет долгую историю (Резник и Варян [1874], 1997), но сейчас быстро меняется благодаря новым методам глубокого обучения, обеспечивающим анализ содержания (текста, музыки, видео) наряду с историей поиска и метаданными (Ван ден Орд и др. [2250], 2014; Жанг и др. [2433], 2017). Фильтрацию спама также можно рассматривать как форму рекомендации (или не рекомендации), — современные методы искусственного интеллекта позволяют отфильтровывать более 99,9% спама, а почтовые службы также могут рекомендовать потенциальных получателей наряду с текстом возможного ответа.

Ведение игр. Когда программа Deep Blue компании IBM победила чемпиона мира Гарри Каспарова в матче 1977 года, апологеты человеческого превосходства возложили свои надежды на игру го. Пиет Хут, астрофизик и энтузиаст го, предсказал, что пройдет "сто лет, прежде чем компьютер побьет людей в игре в го, — возможно, даже дольше". Но всего лишь 20 лет спустя программа ALPHAGO превзошла всех людей-игроков (Сильвер и др. [2065], 2017). Ке Цзе, чемпион мира, сказал: "В прошлом году программа еще была похожа на человека в своей игре. Но в этом году она предстала как бог го". Программа ALPHAGO получила преимущество за счет изучения сотен тысяч игр, сыгранных в прошлом игроками-людьми, а также за счет дистилляции знаний опытных игроков го, работавших в команде.

Следующая программа, ALPHAZERO, уже не использовала никаких входных данных от человека (за исключением правил игры, конечно же), но оказалась в состоянии на играх с самой собой научиться играть в такой степени, что смогла победить всех противников, и людей, и машин, играя в го, шахматы и сёги (Сильвер и др. [2064], 2018). Как бы там ни было, чемпионы-люди были побеждены системами ИИ в играх столь разнообразных, как викторина Jeopardy! (Ферруччи и др. [736], 2010), покер (Боулинг и др. [272], 2015; Моравчик и др. [1619], 2017; Браун и Сэндхом [321], 2019), а также в видеоиграх DOTA 2 (Фернандес и Молменн [733], 2018), StarCraft II (Винялс и др. [2273], 2019) и Quake III (Ядерберг и др. [1121], 2019).

Распознавание изображений. Не удовольствовавшись превосходством, достигнутым над человеком в точности распознавания объектов в сложном тесте ImageNet, исследователи в области компьютерного зрения переключились на более сложную задачу генерации субтитров к изображению. Вот некоторые впечатляющие примеры: "Человек, едущий на мотоцикле по грунтовой дороге", "Две пиццы на верхней плите кухонной печи" и "Группа молодых людей, играющих в фрисби" (Винялс и др. [2275], 2017). Однако нынешние системы все еще далеки от совершенства: "Холодильник, заполненный большим количеством еды и напитков" оказывается табличкой, запрещающей парковку, частично залепленной множеством маленьких наклеек.

Медицина. Результаты работы алгоритмов ИИ в настоящее время сравнимы или даже превышают результаты врачей — экспертов в диагностике по многим симптомам, особенно когда диагноз ставится на основании изображений. Примеры включают болезнь Альцгеймера (Динг и др. [622], 2018), метастазы рака (Лиу и др. [1433], 2017; Эстева и др. [698], 2017), офтальмологические заболевания (Гульшан и др. [934], 2016) и кожные заболевания (Лиу и др., 2019). Систематический обзор и метаанализ (Лиу и др. [1432], 2019) показали, что производительность программ ИИ в среднем была эквивалентна таковой для специалистов в области здравоохранения. В настоящее время акцент на применении ИИ в медицине часто делается на содействии партнерству между человеком и машиной. Например, система LYNA в диагностике метастатического рака молочной железы достигает общей точности на уровне 99,6% — лучше, чем эксперт-медик без

посторонней помощи, но их комбинация работает еще лучше (Лиу и др. [1434], 2018; Штейнер и др. [2126], 2018).

В настоящее время широкое применение подобных методов ограничивается не их диагностической точностью, а необходимостью продемонстрировать достигнутое улучшение в клинических результатах и необходимостью обеспечения прозрачности, отсутствия предвзятости и сохранения конфиденциальности данных (Топол [2221], 2019). В 2017 году администрация США по продуктам питания и лекарственным средствам (FDA) одобрила только два медицинских приложения ИИ, но в 2018 году их число увеличилось до 12 и продолжает расти.

Климатология. В 2018 году команда ученых получила премию Гордона Белла за создание модели глубокого обучения, способной выявить детальную информацию об экстремальных погодных явлениях, которая ранее была захоронена в огромных массивах климатических данных. Прежде чем программа машинного обучения смогла сделать это, им потребовалось превысить уровень быстродействия в 10^{18} операций в секунду, для чего пришлось использовать суперкомпьютер со специализированным графическим процессором (Курт и др. [1329], 2018). В 2019 году Ролник и его соавторы [1906] представили 60-страничный каталог способов, которыми методы машинного обучения могут быть использованы для отслеживания климатических изменений.

Все сказанное выше — это всего лишь несколько примеров использования систем искусственного интеллекта, которые уже существуют на сегодняшний день. И это не магия или научная фантастика, а наука, инженерия и математика, введением в которые и является эта книга.

1.5. Риски и преимущества искусственного интеллекта

Философ Фрэнсис Бэкон, которому приписывается создание научного метода, в своей книге *Мудрость древних* ([105], 1609) отметил, что "механические искусства можно использовать двояко, они одинаково хороши как для причинения вреда, так и для исцеления". Поскольку ИИ играет все более важную роль в экономической, социальной, научной, медицинской, финансовой и военной сферах, будет полезно рассмотреть его способности "причинения вреда" и "исцеления", на современном языке — те риски и преимущества, которые он способен принести. Здесь дается лишь общее представление по этому вопросу, более подробно он анализируется в главах 27 и 28.

Чтобы начать обсуждение преимуществ, сначала просто напомним, что вся наша цивилизация является продуктом интеллекта человека. Если мы получим доступ к существенно большему машинному интеллекту, потолок наших амбиций также существенно поднимется. Потенциал искусственного интеллекта и робототехники в освобождении человечества от монотонной тяжелой работы и резком увеличении производства товаров и услуг может предвещать наступление эры

мира и изобилия. Возможное ускорение научных исследований может иметь следствием, например, нахождение средств излечения различных заболеваний, решения проблемы климатических изменений и дефицита ресурсов. Как сказал Демис Хассабис, генеральный директор компании Google DeepMind, "Сначала решите проблему ИИ, а затем используйте ИИ для решения всех остальных проблем".

Однако задолго до того, как нам представится возможность "решить проблему ИИ", мы будем подвержены рискам его неправильного использования, — возможно, непреднамеренного либо наоборот. Одни из этих рисков уже вполне очевидны, тогда как другие пока кажутся лишь весьма вероятными, исходя из существующих тенденций.

- *Смертельное автономное оружие*. По решению Организации Объединенных Наций в эту категорию попадает любое оружие, способное самостоятельно определить местонахождение, выбрать и устранить человеческие цели без вмешательства человека-оператора. Основная проблема с таким оружием заключается в его *масштабируемости*: отсутствие в виде обязательного требования надзора со стороны человека означает, что небольшая группа может развернуть сколь угодно большое количество такого оружия, направив его на людей, определяемых по любому выполнимому этим устройством критерию распознавания. Технологии, необходимые для создания автономного оружия, аналогичны тем, которые требуются для создания автомобилей с автоматическим управлением. В 2014 году в ООН начались неофициальные обсуждения с экспертами потенциальных рисков смертельного автономного оружия, после чего в 2017 году группа правительственных экспертов перешла к официальной стадии, предшествующей заключению соответствующего договора.
- *Наблюдение и убеждение*. В то время как контролировать телефонные линии, каналы видеокамер, электронную почту и другие каналы передачи сообщений людьми дорого, утомительно, а иногда и юридически сомнительно для безопасности персонала, вполне возможно использовать системы ИИ (распознавание речи, компьютерное зрение, обработка естественного языка) в виде масштабируемых приложений для выполнения массового наблюдения за людьми и выявления их деятельности по интересам. Путем адаптации информационных потоков через социальные сети к отдельным лицам на основе использования методов машинного обучения их политическое поведение можно в определенной степени менять и контролировать — проблема, которая стала очевидна на выборах, проходивших в США в 2016 году.
- *Предвзятость принимаемых решений.* Небрежное или преднамеренно неправильное использование алгоритмов машинного обучения для таких задач, как оценка возможности условно-досрочного освобождения или анализ кредитных заявок, может привести к принятию системой ИИ таких решений, которые будут предвзятыми по признаку расы, пола или других

защищенных категорий. Часто данные для обучения сами по себе уже так или иначе отражают господствующие в обществе предубеждения.

- *Влияние на занятость*. Проблема машин, использование которых приводит к сокращению рабочих мест, имеет многовековую историю. История эта никогда не была простой: машины выполняют некоторые из тех операций, которые в противном случае выполняли бы люди, но они также делают людей более продуктивными и, следовательно, дают им больше возможностей для трудоустройства, а компании делают более прибыльными и способными платить более высокую заработную плату. Они также могут сделать некоторые виды деятельности экономически жизнеспособными, тогда как в противном случае они были бы непрактичны. Использование машин в конечном счете приводит к увеличению богатства, но, как правило, способствует смещению этого богатства от труда к капиталу, в еще большей степени усугубляя рост общественного неравенства. Предыдущие достижения в области технологии, такие как изобретение механических ткацких станков, имели следствием серьезные проблемы в области занятости, но в конце концов люди все же находили для себя новые виды работ. С другой стороны, вполне возможно, что со временем системы ИИ будут выполнять и эти новые виды работ. Эта тема быстро становится основным фокусом сосредоточения внимания для экономистов и правительств во всем в мире.
- *Приложения, критические с точки зрения безопасности*. По мере своего развития технологии ИИ все чаще используются в приложениях с "высокими ставками", надежность которых критически важна для безопасности людей, таких как вождение автомобилей или управление водоснабжением городов. Аварии со смертельным исходом уже имели место, и это высветило трудности формальной верификации и статистического анализа риска для систем ИИ, разработанных с использованием методов машинного обучения. Область ИИ действительно нуждается в разработке технических и этических стандартов, по крайней мере сопоставимых с теми, которые приняты в других инженерных и медицинских дисциплинах, где на карту ставится жизнь людей.
- *Кибербезопасность*. Методы искусственного интеллекта полезны для защиты от кибератак, например, путем обнаружения необычных моделей поведения, но они также могут способствовать повышению устойчивости вредоносных программ, повышению их выживаемости и дальнейшему распространению. Например, методы обучения с подкреплением уже были использованы для создания высокоэффективных инструментов проведения автоматизированного персонифицированного шантажа и фишинговых атак.

Более подробно эти темы рассматриваются в разделе 27.3. По мере того как системы ИИ будут становиться все более умелыми, они будут выполнять в обществе все большее количество обязанностей, которые ранее выполняли люди. И подобно

тому как люди, которые ранее выполняли эти обязанности, могли совершать злонамеренные действия, можно ожидать, что в новых условиях люди смогут злонамеренно использовать системы искусственного интеллекта, выполняющие эти обязанности, для причинения еще большего вреда. Все приведенные выше примеры указывают на важность управления и в конечном счете регулирования систем ИИ. В настоящее время исследовательское сообщество и основные корпорации, вовлеченные в исследования в области ИИ, разработали принципы добровольного самоуправления для деятельности, связанной с ИИ (см. раздел 27.3). Правительства и международные организации создают консультативные органы для разработки соответствующих правил для каждого конкретного случая использования ИИ, для подготовки к возможным экономическим и социальным последствиям и для извлечения всех преимуществ из использования возможностей ИИ в решении основных социальных проблем.

Что можно сказать о долгосрочной перспективе? Достигнем ли мы давней цели: создания интеллекта, сравнимого или даже более одаренного, чем человеческий интеллект? И если мы это сделаем, то что тогда?

На протяжении большей части истории искусственного интеллекта эти вопросы находились в тени текущей рутинной работы по созданию систем ИИ, способных сделать что-нибудь, хотя бы отдаленно свидетельствующее о разумности. Как и в любой иной достаточно широкой дисциплине, большая часть исследователей в области ИИ специализировалась в определенных направлениях, таких как различные игры, представление знаний, компьютерное зрение или понимание естественного языка, часто исходя из предположения, что прогресс в этим направлении будет способствовать достижению главных целей всей области ИИ. Нильс Нильссон ([1690], 1995), один из первых руководителей проекта Shakey в SRI, напоминал о широте поля этих целей и предупреждал, что каждое из более узких направлений находится в опасности замкнуться на собственных задачах. Позже некоторые влиятельные основатели ИИ, в том числе Джон Маккарти ([1533], 2007), Марвин Мински ([1584], 2007) и Патрик Уинстон (Бил и Уинстон [149], 2009), согласились с предупреждениями Нильссона, предложив исследователям в области ИИ вместо фокусировки на измеримой производительности конкретных приложений обратить взор к истокам и поставить конечной целью своих исследований, как выразился Херб Саймон, создание "машин, которые думают, учатся и создают". Они назвали эту конечную цель ▶**ИИ на уровне человека** (Human-Level AI, **HLAI**) — машина должна научиться делать все, что может сделать человек. Первый симпозиум этого направления состоялся в 2004 году (Мински и др. [1587], 2004). Еще одна попытка создать новое движение в ИИ с аналогичными целями была определена как ▶**общий искусственный интеллект** (Artificial General Intelligence — **AGI**) (Гортцел и Пенначин [876], 2007), в его рамках в 2008 году была проведена первая конференция и организован *Journal of Artificial General Intelligence*.

Примерно в то же время были высказаны опасения, что создание ▶**искусственного суперинтеллекта** (Artificial SuperIntelligence — **ASI**) — интеллекта,

который намного превосходит человеческие способности — может быть плохой идеей (Юдковски [2416], 2008; Омохундро [1713], 2008). Сам Тьюринг ([2238], 1996) высказал то же самое предположение в лекции, прочитанной им в Манчестере в 1951 году, опираясь на более ранние идеи Сэмюэля Батлера ([353], 1863).[15]

> Кажется вполне вероятным, что после того, как метод машинного мышления будет реализован, ему не понадобится много времени, чтобы превзойти наши слабые силы... Поэтому на каком-то этапе мы должны ожидать, что машины получат контроль — таким способом, который был описан в утопическом романе "Эревон" Сэмюэля Батлера.

В связи с последними достижениями в области глубокого обучения подобные мнения получили еще бо́льшее распространение, о чем свидетельствуют публикации книг, таких как *Суперинтеллект* Ника Бострома ([260], 2014), и публичные заявления Стивена Хокинга, Билла Гейтса, Мартина Риза и Илона Маска.

Вполне естественно испытывать общее чувство беспокойства в отношении идеи создания сверхинтеллектуальных машин. Эту ситуацию можно охарактеризовать как ▶**проблема гориллы**: около семи миллионов лет назад появился вымерший примат, от которого одна ветвь дальнейшего развития ведет к гориллам, а другая — к людям. Сегодня гориллы не слишком счастливы от сосуществования с ветвью человека и по сути не имеют контроля над своим будущим. Если таким же окажется конечный результат достижения успеха в создании сверхчеловеческого ИИ — т.е. люди уступят ему контроль над своим будущим, — то, возможно, мы должны будем прекратить работу над ИИ и как следствие отказаться от выгод, которые он может принести. В этом суть предупреждения Тьюринга: вовсе не очевидно, что мы сможем управлять машинами, которые будут умнее нас.

Если бы сверхчеловеческий ИИ представлял собой черный ящик, который прибыл к нам из космоса, то на самом деле было бы вполне разумно проявлять осторожность в отношении его открытия. Однако это не так: *мы* разрабатываем системы ИИ, поэтому, если в конечном счете они действительно "захватят контроль", как предполагал Тьюринг, то это может быть только результатом неудачного проектирования.

Чтобы избежать такого результата, прежде всего необходимо понять возможные причины потенциальной неудачи. Норберт Винер ([2341], 1960), который задумался над отдаленным будущим искусственного интеллекта, увидев, как программа игры в шашки Артура Сэмюэла научилась обыгрывать своего создателя, сказал по этому поводу следующее.

---

[15] Еще раньше, в 1847 году, Ричард Торнтон, редактор журнала *Primitive Expounder*, протестовал против механических тепломеров: "Ум... обгоняет сам себя и покончил с необходимостью собственного существования, изобретая машины, мыслящие вместо него... Но кто может знать, что такие машины, достигнув большего совершенства, не задумаются о планах исправления всех своих недостатков, а затем, напрягшись, не выдвинут идеи, выходящие за пределы доступного смертному уму!"

> Если, чтобы достичь наших целей, мы используем механическое устройство, в работу которого мы не можем эффективно вмешиваться... нам будет лучше иметь полную уверенность, что цель, поставленная перед машиной, является той, которой мы действительно хотим достичь.

Во многих культурах есть мифы о людях, которые о чем-то просят богов, гениев, магов или дьяволов. В этих историях они неизменно получают то, о чем просили, причем буквально, а затем сожалеют об этом. Третье желание, если оно есть, всегда состоит в том, чтобы отменить первые два. Можно назвать это ▶**проблемой царя Мидаса**: Мидас, легендарный царь в греческой мифологии, попросил богов, чтобы все, к чему он прикоснется, превращалось в золото, но быстро пожалел об этой просьбе, коснувшись своей еды, питья и членов семьи.[16]

Мы упоминали об этой проблеме в разделе 1.1.5, где было указано на необходимость существенной модификации стандартной модели ввода фиксированных целей в машину. Устранение затруднений Винера состоит в том, чтобы вообще не иметь определенной "цели, поставленной перед машиной". Вместо этого нам нужны машины, которые стремятся к достижению целей человека, но знают, что они не имеют полной определенности в том, каковы именно эти цели.

Как ни жаль, почти все исследования ИИ до настоящего времени проводились в рамках стандартной модели, а это означает, что почти весь технический материал в данном издании отражает именно эти интеллектуальные рамки. Есть, однако, некоторые первые результаты и в рамках новых положений. В главе 16 показано, что машина имеет положительный стимул для того, чтобы позволить себе отключиться тогда и только тогда, когда она не уверена в отношении цели человека. В главе 18 были сформулированы и рассмотрены ▶**игры-помощники**, математически описывающие ситуацию, в которой человек имеет цель, и машина пытается ее достичь, но изначально не знает, что она собой представляет. В главе 22 объясняются методы ▶**инвертированного обучения с подкреплением**, позволяющие машинам больше узнать о предпочтениях человека, наблюдая за вариантами выбора, осуществляемого людьми. А в главе 27 исследуются две основные трудности новых концепций: во-первых, наш выбор зависит от наших предпочтений через очень сложную когнитивную архитектуру, которую трудно инвертировать, а во-вторых, мы, люди, можем не ставить постоянные предпочтения на первое место — индивидуально либо как группа, — поэтому может быть неясно, что системы ИИ *должны* делать для нас.

---

[16] Мидас мог поступить лучше, если бы последовал основным принципам безопасности и включил в свою просьбу кнопку "Отменить" и кнопку "Пауза".

Резюме

В данной главе дается определение искусственного интеллекта и описывается исторический контекст, в котором развивалась эта область исследований. Некоторые важные моменты приведены ниже.

- Взгляды ученых на искусственный интеллект могут различаться. Вот два важных вопроса, на которые каждый должен дать ответ: "Вас больше интересует мышление или поведение?" и "Вы намерены моделировать качества человека или просто стремитесь достичь наилучших результатов?"
- В соответствии с тем, что сейчас называют стандартной моделью, ИИ нацелен в основном на **рациональное действие**. Идеальный **интеллектуальный агент** выбирает наилучшее возможное действие в каждой ситуации. В этой книге задача создания интеллектуальных агентов рассматривается именно в этом смысле.
- К этой простой идее необходимо добавить два уточнения. Во-первых, способность любого агента, человека или нет, выбирать рациональные действия ограничена вычислительной сложностью выполнения этого. Во-вторых, концепцию машины, которая преследует определенные цели, следует заменить концепцией машины, преследующей цели, полезные человеку, но не имеющей точного представления о том, что именно они собой представляют.
- Философы (начиная с 400 года до н.э.) обосновали возможность ИИ, выдвинув предположение, что сознание в некотором отношении напоминает машину, оперирующую знаниями, закодированными на каком-то внутреннем языке, и что мышление может использоваться для выбора, какие действия следует предпринять.
- Математики предоставили инструментальные средства для манипулирования высказываниями, обладающими логической достоверностью, а также недостоверными вероятностными высказываниями. Кроме того, они заложили основу не только понимания того, что представляют собой вычисления, но и формирования рассуждений об алгоритмах.
- Экономисты формализовали проблему принятия решений, максимизирующих ожидаемую полезность для лица, принимающего решение.
- Нейробиологи установили некоторые факты о том, как мозг работает и в чем он похож, а чем отличается от компьютеров.
- Психологи подтвердили идею, что люди и животные могут рассматриваться как машины обработки информации. Лингвисты показали, что процессы использования естественного языка вписываются в эту модель.
- Компьютерные инженеры предоставляли все более и более мощные машины, что в конечном счете позволило реализовать самые разные приложения

ИИ, а инженеры-программисты позаботились о том, чтобы они были более удобными для пользователей.

- Теория управления описывает способы разработки таких устройств, которые действуют оптимально на основе обратной связи с окружающей средой. Изначально математические инструментальные средства теории управления весьма отличались от применяемых в области искусственного интеллекта, но эти научные области все больше сближаются.
- История искусственного интеллекта циклична и включает периоды успеха и неоправданного оптимизма, за которыми неизбежно следовали снижение энтузиазма и сокращение финансирования. В ней также были периоды появления новых творческих подходов, лучшие из которых затем неуклонно совершенствовались.
- Область ИИ значительно укрепилась в сравнении с первыми десятилетиями, — как теоретически, так и методологически. Поскольку проблемы, с которыми имел дело ИИ, становились все более сложными, потребовался переход от булевой логики к вероятностной логике, от ручного кодирования знаний к машинному обучению на основе имеющихся данных. Все это уже привело к существенному повышению возможностей реальных систем и более тесной интеграции с другими дисциплинами.
- Поскольку системы ИИ уже нашли себе широкое применение в реальном мире, возникла насущная необходимость рассмотреть широкий диапазон связанных с этим рисков и этических последствий.
- В долгосрочной перспективе мы столкнулись с трудной проблемой контроля над системами искусственного суперинтеллекта, способными развиваться в непредсказуемых направлениях. Решение этой проблемы, похоже, потребует изменения самой концепции искусственного интеллекта.

Библиографические и исторические заметки

Исчерпывающую историю ИИ дал Нильс Нильссон ([1691], 2009), один из пионеров в этой области. Педро Домингос ([633], 2015) и Мелани Митчелл ([1593], 2019) дали обзор машинного обучения, доступный для широкой аудитории, а Кай-Фу Ли ([1376], 2018) описал гонку за международное лидерство в области искусственного интеллекта. Мартин Форд ([757], 2018) взял интервью у 23 ведущих исследователей ИИ.

Основными профессиональными сообществами по ИИ являются Ассоциация развития искусственного интеллекта (Association for the Advancement of Artificial Intelligence — AAAI), группа Special Interest Group in Artificial Intelligence (SIGAI, ранее — SIGART) в рамках Ассоциации вычислительной техники (ACM), Европейская ассоциация искусственного интеллекта и Общество искусственного интеллекта и моделирования поведения (Society for Artificial Intelligence and

Simulation of Behaviour — AISB). Партнерство по ИИ объединяет многие коммерческие и некоммерческие организации, заинтересованные в этических и социальных последствиях внедрения систем ИИ. В журнале *AI Magazine*, выпускаемом AAAI, публикуется множество тематических и учебных статей, а веб-сайт этой ассоциации, `aaai.org`, предлагает посетителям новости, учебные пособия и справочную информацию.

Результаты новейших работ публикуются в трудах основных конференций по ИИ: Международной объединенной конференции по ИИ (International Joint Conference on AI — IJCAI), ежегодной Европейской конференции по ИИ (European Conference on AI — ECAI) и конференции National Conference on AI, чаще упоминаемой под названием AAAI (так сокращенно называется организация American Association for AI, под эгидой которой проводится эта конференция). Машинное обучение освещается в рамках Международной конференции по машинному обучению (International Conference on Machine Learning) и конференции по системам обработки нейронной информации (Neural Information Processing Systems — NeurIPS). Крупнейшими журналами в области общего ИИ являются *Artificial Intelligence*, *Computational Intelligence*, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *IEEE Intelligent Systems* и *Journal of Artificial Intelligence Research*. Имеется также много конференций и журналов, посвященных отдельным областям исследований, которые будут указаны в соответствующих главах.

Упражнения

Эти упражнения предназначены для стимулирования дискуссий, а некоторые могут быть использованы как учебные проекты с установленными сроками. Альтернативный подход — предварительные решения могут быть представлены сейчас, но рассматриваться будут уже после завершения всего курса, построенного на базе этой книги.

1.1. Дайте определения своими словами следующим понятиям: а) интеллект, б) искусственный интеллект, в) агент, г) рациональность, д) логические рассуждения.

1.2. Прочитайте оригинальную статью Тьюринга по искусственному интеллекту (Тьюринг [2235], 1950). В этой статье он обсуждает несколько потенциальных возражений против предложенного им подхода и теста на интеллектуальность. Какие из этих возражений все еще остаются весомыми в определенной степени? Являются ли приведенные им опровержения правильными? Можете ли вы выдвинуть новые возражения, которые следуют из событий, происшедших с тех пор, как Тьюринг написал свою статью? В этой статье он предсказал, что к 2000 году компьютер с вероятностью 30% будет успешно проходить пятиминутный тест Тьюринга с участием слабо подготовленного экспериментатора. Какие шансы, по вашему мнению, имел бы компьютер сегодня? Еще через 50 лет?

1.3. Каждый год премия Лёбнера присуждается программе, наиболее близкой к прохождению определенной версии теста Тьюринга. Проведите исследование и сообщите о последнем лауреате премии Лёбнера. Какие методы используются в этой программе? Какой вклад внесла эта программа в развитие искусственного интеллекта?

1.4. Рациональны ли рефлекторные действия (например, отдергивание руки при прикосновении к горячей печи)? Можно ли считать их интеллектуальными?

1.5. Существуют хорошо известные классы проблем, трудноразрешимых для компьютеров, а также другие классы, для которых доказана их неразрешимость. Означает ли это, что создание искусственного интеллекта невозможно?

1.6. Предположим, мы расширили программу Эванса SYSTEM так, чтобы она могла набрать 200 баллов на стандартном тесте проверки интеллекта (IQ). Можно ли в этом случае считать, что программа стала более интеллектуальной, чем обычный человек? Объясните.

1.7. Нейронная структура морского слизняка аплазии (aplysia) была широко изучена (первым был нобелевский лауреат Эрик Кандел), — просто потому, что у него всего около 20 000 нейронов, большинство из которых крупные и легко доступные для манипуляций. Предположив, что время цикла нейрона аплизии примерно такое же, как и у нейрона человека, сравните вычислительную мощность нейронной структуры животного, выраженную в количестве обновлений памяти в секунду, в сравнении с высокопроизводительным компьютером, данные которого приведены на рис.1.3.

1.8. Почему интроспекция (т.е. самоанализ — составление отчета о собственных мыслях) может оказаться неточной? Может ли человек ошибаться в отношении того, что он думает? Обоснуйте свой ответ.

1.9. В какой степени следующие компьютерные системы можно считать системами искусственного интеллекта:

а) сканеры штрих-кода в супермаркетах;
б) поисковые системы Интернета;
в) голосовое меню телефона;
г) алгоритмы интернет-маршрутизации, динамически реагирующие на состояние сети.

1.10. В какой степени следующие компьютерные системы можно считать системами искусственного интеллекта:

а) сканеры штрих-кода в супермаркетах;
б) голосовое меню телефона;
в) функции проверки правописания и коррекции грамматики в Microsoft Word;
г) алгоритмы интернет-маршрутизации, динамически реагирующие на состояние сети.

1.11. Многие из предложенных вычислительных моделей когнитивной деятельности включают в себя довольно сложные математические операции, такие как свертка изображения с использованием функции Гаусса или поиск минимума функции энтропии. Большинство людей (и конечно, все животные) вообще никогда не изучают подобную математику, почти никто не изучает ее перед поступлением в колледж и почти никто не может вычислить свертку функции с использованием функции Гаусса в уме. Какой смысл в том, чтобы говорить, что "система компьютерного зрения" выполняет подобный тип математических расчетов, тогда как реальный человек не имеет представления о том, как это делается?

1.12. Некоторые авторы утверждают, что самой важной частью интеллекта служат сенсорные способности и моторные навыки, а "высокоуровневые" возможности по сути являются паразитическими, — они просто надстройки над этими основным возможностям. Не вызывает сомнения, что бо́льшая часть эволюции и значительная часть

мозга были связаны с развитием восприятия и моторных навыков, в то время как искусственный интеллект сосредоточился на таких задачах, как ведение игр и формирование логического вывода, которые во многом оказались значительно более простыми по сравнению с восприятием и осуществлением действий в реальном мире. Считаете ли вы, что традиционная направленность искусственного интеллекта на изучение высокоуровневых познавательных способностей является неверной?

1.13. Почему результатом эволюции обычно становится появление систем, которые действуют рационально? Для достижения каких целей предназначены подобные системы?

1.14. Искусственный интеллект — это наука или инженерная дисциплина? Или ни то, ни другое? Поясните свой ответ.

1.15. "Безусловно, компьютеры не могут быть интеллектуальными, ведь они способны выполнять только то, что указали им программисты". Является ли последнее утверждение истинным и следует ли из него первое?

1.16. "Безусловно, животные не могут быть интеллектуальными, ведь они способны выполнять только то, что диктуют им гены". Является ли последнее утверждение истинным и следует ли из него первое?

1.17. "Безусловно, животные, люди и компьютеры не могут быть разумными, ведь они способны выполнять только то, что диктуют законы физики тем атомам, из которых они состоят". Является ли последнее утверждение истинным и следует ли из него первое?

1.18. Изучите литературу по искусственному интеллекту и определите, могут ли в настоящее время компьютеры решать следующие задачи.

а) Игра в настольный теннис (пинг-понг) на достаточно высоком уровне.
б) Вождение автомобиля в центре Каира, Египет.
г) Вождение автомобиля в Викторвилле, Калифорния.
д) Покупка недельного запаса продуктов в супермаркете.
е) Покупка недельного запаса продуктов в интернет-магазине.
ж) Участие в карточной игре бридж на конкурентоспособном уровне.
з) Открытие и доказательство новых математических теорем.
и) Написание рассказа, который непременно должен быть смешным.
к) Предоставление компетентных юридических консультаций в специализированной области права.
л) Перевод разговорной речи в режиме реального времени с английского языка на шведский.
м) Выполнение сложной хирургической операции.

1.19. В отношении тех задач из упражнения 1.18, которые в настоящее время являются неосуществимыми, попытайтесь понять, в чем заключаются основные трудности, и предсказать, когда они будут преодолены (и произойдет ли это вообще).

1.20. Уже давно в различных подобластях ИИ проводятся конкурсы в виде постановки стандартной задачи и предложения исследователям сделать все возможное для нахождения наилучшего ее решения. Примерами могут служить соревнование DARPA Grand Challenge для роботизированных автомобилей, международный турнир по планированию, роботизированная футбольная лига Robocup, конкурс TREC по извлечению информации и соревнования по машинному переводу и распознаванию речи. Изучите материалы пяти из этих конкурсов и опишите прогресс, достигнутый за последние годы. В какой степени эти конкурсы способствовали продвижению к современному состоянию исследований в области ИИ? В какой степени они наносят ущерб этой области, отнимая возможности реализации у новых идей?

ГЛАВА 2

Интеллектуальные агенты

В этой главе обсуждаются основные свойства агентов, идеальных или нет, разнообразие вариантов окружающей среды и вытекающее из этого многообразие типов агентов.

В главе 1 понятие **рационального агента** было определено как центральная концепция в выбранном авторами данной книги подходе к искусственному интеллекту. В этой главе указанное понятие раскрывается более подробно. В ней показано, что концепция рациональности может применяться к самым различным агентам, действующим в любой среде, которую только можно себе представить. В этой книге намерение авторов состоит в том, чтобы использовать эту концепцию для разработки небольшого набора принципов проектирования для создания успешно действующих агентов — систем, которые вполне обоснованно можно было бы назвать **интеллектуальными**.

Мы начнем с изучения агентов, вариантов среды и связей между ними. Тот факт, что одни агенты действуют лучше, чем другие, естественным образом приводит к идее рационального агента — такого, который действует настолько успешно, насколько это вообще возможно. Насколько успешно может действовать агент, зависит от природы окружающей его среды, — одни варианты будут более сложными, чем другие. В этой главе представлена грубая классификация вариантов среды и показано, как ее свойства влияют на проектирование агентов, наиболее подходящих именно для нее. Здесь описан ряд основных, "скелетных", проектов агентов, которые затем будут необходимым образом детализированы в остальной части книги.

2.1. Агенты и среды

Агент — это все что угодно, что может рассматриваться как имеющее способность воспринимать ▶**окружающую среду** с помощью ▶**датчиков** и воздействовать на эту среду с помощью ▶**исполнительных механизмов**. Эта простая идея представлена на рис. 2.1. Человек, если рассматривать его в качестве агента, имеет глаза, уши и другие органы чувств в качестве датчиков, а его исполнительными

механизмами являются руки, ноги, речевой аппарат и т.д. Агент-робот в качестве датчиков может иметь видеокамеры и инфракрасные дальномеры, а исполнительными механизмами для него могут служить различные двигатели. Компьютерная программа, рассматриваемая как агент, получает в качестве сенсорных данных содержимое файлов, сетевые пакеты, вводимую человеком информацию (клавиатура, мышь, сенсорный экран, голос), а ее воздействие на окружающую среду осуществляется посредством записи файлов, отправки сетевых пакетов, отображения информации или генерации звука. Мы принимаем общее допущение, что каждый агент может воспринимать собственные действия (но не всегда их результаты). Теоретически окружающей средой может быть что угодно — в пределе это вся Вселенная! На практике это просто та часть Вселенной, состояние которой мы принимаем во внимание, создавая некоторый агент. Иначе говоря, это та ее часть, которая вмещает все, что агент сможет воспринять и на что сможет повлиять своими действиями.

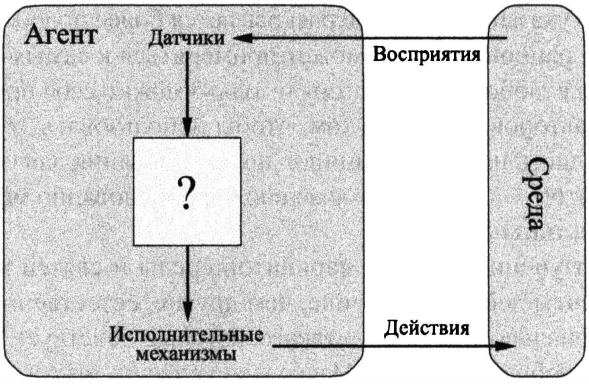


Рис. 2.1. Агенты взаимодействуют с окружающей средой посредством датчиков и исполнительных механизмов

Мы используем термин ▶**восприятие** для обозначения контента, воспринимаемого сенсорами агента. ▶**Последовательностью восприятия** агента называется полная история всего, что когда-либо было им воспринято. Вообще говоря, ▶*выбор агентом действия в любой конкретный момент может зависеть от его встроенных знаний и всей последовательности восприятия, накопленной к данному моменту, но никогда от чего-либо, чего агент не воспринимал.* Определив, какое действие будет выбрано агентом в ответ на любую возможную последовательность восприятия, можно получить о нем более-менее полную информацию. Говоря языком математики, можно утверждать, что поведение любого агента может быть описано с помощью ▶**функции агента**, отображающей любую заданную последовательность восприятия на некоторое действие.

Функцию агента можно представить в виде *таблицы*, описывающей поведение любого заданного агента, но для большинства из них это была бы очень большая таблица — фактически бесконечная, если только не установить предельную длину рассматриваемой последовательности восприятия. Проводя эксперименты с некоторым агентом, такую таблицу, в принципе, можно построить, проверяя все возможные последовательности актов восприятия и регистрируя ответные действия агента.[1] Такая таблица, безусловно, является *внешним* описанием агента. *Внутренняя* функция агента для некоторого искусственного агента реализуется с помощью ▶**программы агента**. Важно различать два этих понятия. Функция агента представляет собой абстрактное математическое описание, а программа агента — это конкретная реализация, действующая в рамках некоторой физической системы.

Чтобы проиллюстрировать эти идеи, воспользуемся очень простым примером: рассмотрим *мир пылесоса*, представленный на рис. 2.2. Этот мир, в котором действует агент в виде робота-пылесоса, состоит всего из нескольких квадратов, которые могут быть либо чистыми, либо грязными. На рис. 2.2 представлен вариант всего с двумя квадратами, *A* и *B*. Робот-пылесос, выполняющий роль агента, воспринимает, в каком квадрате он находится и есть ли в этом квадрате мусор. Исходно он находится в квадрате *A*. Его возможными действиями являются: перейти в квадрат справа, перейти в квадрат слева, убрать мусор или бездействовать.[2] Одна из очень простых функций агента состоит в следующем: если в текущем квадрате имеется мусор, то убрать его, иначе — перейти в другой квадрат. Часть таблицы для данной функции агента показана на рис. 2.3, а реализующая ее программа агента приведена ниже в этой главе, на рис. 2.8.

Глядя на рис. 2.3, можно прийти к заключению, что в мире пылесоса можно определять различные агенты, просто заполняя разными способами правый столбец таблицы функций. Но здесь возникает очевидный вопрос: ▶*Какой способ заполнения этой таблицы следует считать правильным?* Иными словами, что делает агента хорошим или плохим, интеллектуальным или наоборот? Ответ на этот вопрос приведен в следующем разделе.

Прежде чем завершить этот раздел, необходимо отметить, что понятие агента здесь рассматривается лишь как инструмент анализа систем, а не как абсолютная характеристика, в соответствии с которой мир делится на агентов и неагентов. Например, в качестве агента вполне можно рассматривать карманный калькулятор,

---

[1] Если агент для выбора своих действий использует определенную рандомизацию, то может потребоваться проверить каждую последовательность многократно, чтобы определить вероятность каждого действия. Можно предположить, что выбор действий случайным образом является довольно неразумным подходом, однако ниже в этой главе будет показано, что такое поведение может оказаться весьма интеллектуальным.

[2] В реальном роботе-пылесосе едва ли будут реализованы такие действия, как "двигаться вправо" и "двигаться влево". Скорее его возможными действиями будут "вращать колеса вперед" и "вращать колеса назад". В нашем примере мы выбрали такие действия, которые проще отслеживать на рисунке, не заботясь о простоте их реализации в реальном роботе.

выбирающий действие "отобразить число 4" после получения последовательности восприятия "2+2=", но подобный анализ вряд ли поможет кому-либо понять принципы работы калькулятора. В широком смысле целью любой области техники можно считать создание артефактов, так или иначе взаимодействующих с окружающим миром. При таком подходе область ИИ связана (по мнению авторов) с наиболее интересной частью этого обширного спектра, где артефакты имеют значительные вычислительные ресурсы и действуют в среде, требующей принятия нетривиальных решений.

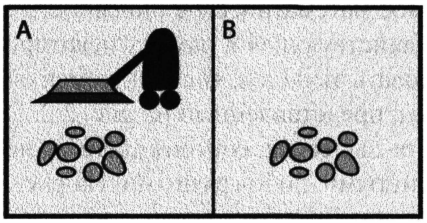


Рис. 2.2. Мир пылесоса, в котором всего два помещения, представленных квадратами *A* и *B*. Каждый квадрат может быть чистым или грязным, а агент может перемещаться в левый или правый квадрат и выполнять в нем уборку. В различных версиях мира пылесоса могут действовать разные правила в отношении того, что может воспринимать агент, всегда ли его действия будут успешны и т.д.

Последовательность восприятия	Действие
[A, Чисто]	Вправо
[A, Грязно]	Убрать
[B, Чисто]	Влево
[B, Грязно]	Убрать
[A, Чисто], [A, Чисто]	Вправо
[A, Чисто], [A, Грязно]	Убрать
⋮	⋮
[A, Чисто], [A, Чисто], [A, Чисто]	Вправо
[A, Чисто], [A, Чисто], [A, Грязно]	Убрать
⋮	⋮

Рис. 2.3. Часть таблицы функции простого агента в мире пылесоса, представленном на рис. 2.2. Агент убирает текущий квадрат, если в нем грязно, а в противном случае перемещается в другой квадрат. Обратите внимание, что эта таблица будет иметь бесконечно большой размер, если нет ограничений на длину допустимых последовательностей восприятия

2.2. Лучшее поведение: концепция рациональности

▶ **Рациональным агентом** является такой агент, который выполняет правильные действия. Очевидно, что выполнение правильных действий лучше, чем выполнение неправильных, но что же понимается под выражением "выполнение правильных действий"?

2.2.1. Показатели производительности

Философия морали выработала несколько различных определений "правильной вещи", но в области ИИ, как правило, придерживаются лишь одного из них, получившего название ▶ **консеквенциализм**: поведение агента оценивается по его последствиям. Когда агент помещается в некоторую среду, он генерирует последовательность своих действий в соответствии с потоком получаемых им восприятий. Последовательность действий агента заставляет среду пройти через определенную последовательность ее состояний. Если эта последовательность состояний среды является желательной, агент действовал хорошо. Само понятие желательности фиксируется ▶ **показателями производительности**, с помощью которой оценивается любая заданная последовательность состояний среды.

У человека всегда имеются собственные желания и предпочтения, поэтому понятие рациональности применительно к людям имеет отношение к их успеху в выборе действий, имеющих следствием последовательность состояний окружающей среды, желательную *с их собственной точки зрения*. С другой стороны, машины *не имеют* собственных желаний и предпочтений; поэтому показатели производительности, по крайней мере изначально, существуют лишь в уме создателя машины или в представлении тех пользователей, для которых эта машина предназначена. Вы увидите, что одним конструкциям агентов свойственно точное представление (версия) показателей производительности, тогда как в других конструкциях показатели производительности полностью скрыты — агент может совершать правильные действия, но не знает, почему.

Вспомнив предупреждение Норберта Винера о том, что "цель, поставленная перед машиной, должна являться той, которой мы действительно хотим достичь" (см. раздел. 1.5), следует подчеркнуть, что в реальной действительности может быть достаточно сложно правильно сформулировать показатели эффективности. Вернемся, например, к агенту-пылесосу из предыдущего раздела. В этом случае можно предложить измерять производительность по количеству мусора, убранного за одну восьмичасовую смену, однако следует учитывать, что в случае рационального агента что вы попросите, то и получите. Так, рациональный агент может максимизировать этот показатель производительности, выполнив уборку, а затем вытряхнув мусор на пол, чтобы убрать его вновь, и т.д. Более подходящим показателем производительности будет вознаграждение агента за поддержание пола чистым. Например, можно присуждать одно очко за каждый чистый квадрат

на каждом заданном временном отрезке (возможно, с штрафом за потребленную электроэнергию и произведенный шум). ➔*Как общее правило лучше выбирать показатели производительности в соответствии с тем, чего на самом деле требуется достичь в данной среде, а не в соответствии с тем, как, по мнению проектировщика, должен вести себя агент.*

Даже если удастся избежать подобных очевидных просчетов, некоторые узловые проблемы могут все еще оставаться нерешенными. Например, понятие "чистый пол" из предыдущего абзаца базируется на средней чистоте во времени. Но одна и та же средняя чистота может быть достигнута с помощью двух различных агентов, один из которых медленно выполняет свои обязанности, но действует непрерывно, тогда как другой убирает исключительно быстро, но требует больших перерывов в работе. То, что здесь является предпочтительным, может показаться тонким аспектом науки о проведении уборки, но на самом деле это глубокий философский вопрос с далеко идущими последствиями. Что лучше — безрассудная жизнь взлетов и падений или безопасное, но однообразное существование? Что лучше — экономика, при которой каждый живет в умеренной бедности, или иной вариант, когда немногие купаются в изобилии, тогда как остальные живут в крайней нищете? Мы оставляем поиск ответов на эти вопросы в качестве упражнения для прилежного читателя.

В этой книге мы обычно будем предполагать, что показатели эффективности могут быть определены правильно. Однако вследствие приведенных выше причин мы должны допускать возможность того, что указанные машине цели действительно могут быть поставлены неверно, подобно проблеме царя Мидаса, упоминавшейся в разделе 1.5. Более того, при разработке компонента программного обеспечения, копии которого будут предоставлены разным пользователям, невозможно точно предвидеть предпочтения каждого из них в отдельности. Следовательно, нам, возможно, потребуется создать агентов, которые исходно будут в неопределенности в отношении истинных показателей своей производительности, но смогут узнать о них больше с течением времени. Такие агенты будут обсуждаться в главах 16, 18 и 22.

2.2.2. Рациональность

В любой момент времени оценка рациональности действий агента определяется с учетом четырех перечисленных ниже факторов.

- Показатели производительности, которые определяют критерии успеха.
- Знания агента о среде, приобретенные ранее.
- Действия, которые могут быть выполнены агентом.
- Последовательность восприятия агента на текущий момент.

С учетом этих факторов можно сформулировать следующее ▶**определение рационального агента**.

> *Для каждой возможной последовательности восприятия рациональный агент должен выбрать действие, которое, как ожидается, максимизирует его показатели производительности с учетом фактов, предоставленных данной последовательностью актов восприятия и всех встроенных знаний, которыми обладает агент.*

Рассмотрим пример простого агента-пылесоса, который убирает квадрат, если в нем имеется мусор, либо переходит в другой квадрат, если мусора в нем нет. (Часть таблицы функции такого агента приведена на рис. 2.3.) Является ли этот агент рациональным? Ответ на этот вопрос может быть разным! Прежде необходимо указать, каковы его показатели производительности, что известно о среде и какими датчиками и исполнительными механизмами он располагает. Примем перечисленные ниже предположения.

- Используемый показатель производительности предполагает вознаграждение в одно очко за каждый чистый квадрат в каждом интервале времени в течение "времени жизни" агента, состоящего из 1000 интервалов времени.
- "География" среды известна *заранее* (см. рис. 2.2), но распределение мусора и первоначальное расположение агента не определены. Чистые квадраты остаются чистыми, а действие *Suck* заключается в уборке мусора и приводит к очистке текущего квадрата. Действия *Left* и *Right* приводят к перемещению агента соответственно влево и вправо, за исключением тех случаев, когда они могли бы вывести агента за пределы среды, и в этих случаях агент остается там, где он находится.
- Единственными возможными для агента действиями являются *Left*, *Right* и *Suck*.
- Агент правильно определяет свое местонахождение и правильно воспринимает информацию о наличии мусора в данном помещении.

При этих условиях этот агент действительно является рациональным; его ожидаемая производительность по крайней мере так же хороша, как и производительность любого другого агента.

Однако можно легко убедиться, что при других обстоятельствах тот же самый агент может стать нерациональным. Например, после того как весь мусор будет убран, агент станет совершать постоянные бесполезные перемещения вправо-влево, и если показатели производительности будут предусматривать штраф в одно очко за каждое передвижение в том или ином направлении, то агент не сможет достичь удовлетворительных результатов. В таких условиях лучший агент должен ничего не делать до тех пор, пока он уверен в том, что все квадраты остаются чистыми. Если чистые квадраты могут снова стать грязными, то агент должен время от времени проводить проверку и снова убирать их по мере необходимости. А если география среды неизвестна, то агенту потребуется ее **исследовать**. В упражнении 2.13 предлагается спроектировать агентов для подобных случаев.

2.2.3. Всезнание, обучение и автономность

Необходимо четко понимать различие между рациональностью и ▶**всезнанием**. Всезнающий агент знает фактический результат любых своих действий и всегда может действовать соответствующим образом, но в действительности всезнание недостижимо. Рассмотрим следующий пример: в один из дней некто гуляет в Париже по Елисейским полям и видит на другой стороне улицы давнего приятеля. Вблизи нет никаких машин, он никуда не спешит и поэтому, будучи рациональным агентом, решает перейти на другую сторону улицы. Между тем на высоте 10 000 метров у пролетающего самолета отваливается дверь грузового отсека[3] и, прежде чем этот господин успевает пересечь улицу, расплющивает его в лепешку. Было ли нерациональным решение этого господина перейти улицу? Очень маловероятно, что в его некрологе напишут: "Пал жертвой глупой попытки перейти улицу".

Этот пример показывает, что рациональность вовсе не равнозначна совершенству. Рациональность — это максимизация *ожидаемой* производительности, а совершенство — максимизация *фактической* производительности. Отказ от требования совершенства агентов — это не только проявление справедливости. Дело в том, что если от агента ожидают, что он будет выполнять действия, которые можно будет полагать наилучшими уже после их совершения, то задача проектирования агента, отвечающего этой спецификации, становится невыполнимой — по крайней мере до тех пор, пока эффективность машин времени или хрустальных шаров провидцев не будет повышена в достаточной степени.

Поэтому наше определение рациональности не требует всезнания, ведь рациональный выбор зависит только от последовательности восприятия, сформированной к *данному моменту*. Также необходимо гарантировать, чтобы агенту непреднамеренно не было позволено принимать участие в действиях, безусловно, не являющихся интеллектуальными. Например, если агент не посмотрит влево и вправо, прежде чем решит пересечь дорогу с интенсивным движением, то полученная им до сих пор последовательность восприятия не сможет подсказать, что к нему на большой скорости приближается тяжелый грузовик. Указывает ли наше определение рациональности, что в этом случае агент может перейти через дорогу? Вовсе нет!

Во-первых, нельзя считать рациональным решение перейти дорогу, исходя из настолько неинформативной последовательности восприятия: риск несчастного случая при попытке пересечения дороги, не взглянув по сторонам, слишком велик. Во-вторых, рациональный агент должен выбрать действие "оглянуться", прежде чем ступить на дорогу, поскольку оно позволяет максимизировать ожидаемую производительность. Выполнение действий *с целью изменения последующих восприятий* — иногда это называют ▶**сбором информации** — составляет важную часть рациональности и подробно рассматривается в главе 16. Другим

[3] См. заметку Н. Гендерсона "На дверях аэробусов Boeing 747 необходимо срочно установить новые замки". — Газета "Вашингтон пост" от 24 августа 1989 года.

примером сбора информации является **исследование**, которое должен предпринять агент-пылесос, оказавшись в среде, изначально ему неизвестной.

Наше определение требует, чтобы рациональный агент не только собирал информацию, но также ▶**обучался** в максимально возможной степени на тех данных, которые он воспринимает. Исходная конфигурация агента может отражать некоторые предварительные знания о среде, но по мере приобретения агентом опыта эти знания могут модифицироваться и пополняться. Существуют крайние случаи, в которых среда известна *изначально* и полностью предсказуема. В таких случаях агенту не требуется воспринимать информацию или обучаться; он просто действует правильно.

Безусловно, такие агенты являются весьма уязвимыми. В качестве примера обратимся к скромному навозному жуку. Выкопав гнездо и отложив яйца, он скатывает шарик навоза у ближайшей навозной кучи и доставляет его к гнезду, чтобы заткнуть вход. Если по пути шарик навоза удалить из его захвата, жук продолжит "манипулировать" им и разыграет целую пантомиму, "затыкая" гнездо несуществующим шариком и вовсе не замечая, что в действительности заветный шарик отсутствует. В процессе эволюции поведение этого жука сформировалось на основании предположения о наличии шарика, и если это предположение нарушается, поведение жука становится неправильным.

Немного более интеллектуальными являются осы-сфексы. Самка сфекса выкапывает норку, вылетает из нее, находит и жалит гусеницу, а затем приносит ее к норе. Затем она снова заползает в норку, чтобы проверить, все ли в порядке, затаскивает внутрь гусеницу и откладывает на нее яйца. Гусеница послужит источником пищи для личинок, которые вылупятся из яиц. До сих пор все идет хорошо, но если энтомолог переместит гусеницу на несколько сантиметров в сторону, пока оса проверяет состояние своей норы, насекомое, не обнаружив гусеницы на месте, вновь возвращается к этапу поиска и перетаскивания гусеницы и продолжает выполнять свой план без изменений даже после десятков вмешательств в процедуру помещения гусеницы в нору. Оса-сфекс не способна обучиться тому, что ее врожденный план не срабатывает, и поэтому не пытается его изменить.

Если степень, в которой агент полагается на априорные знания своего проектировщика, а не на свои собственные восприятия и процессы обучения, высока, о таком агенте говорят, что ему не хватает ▶**автономности**. Рациональный агент должен быть автономным — он должен обучаться всему, что может освоить, для компенсации неполных или неправильных исходно заложенных в него знаний. Например, агент-пылесос, который обучается прогнозированию, где и когда появится новый мусор, безусловно, будет лучше, агента, который на это не способен.

С точки зрения практики к агенту редко предъявляется требование, чтобы он с самого начала был полностью автономным: если агент имеет мало опыта или не имеет его вообще, он будет вынужден действовать случайным образом, если только проектировщик не оказал ему определенную помощь. Подобно тому, как эволюция предоставляет животным необходимое количество врожденных рефлексов,

чтобы после рождения они прожили достаточно долго и успели обучиться остальному самостоятельно, так и искусственному интеллектуальному агенту будет разумно предоставить некоторые начальные знания, а не только наделить его способностью обучаться. После достаточного опыта существования в своей среде поведение рационального агента может, по сути, стать *независимым* от его начальных знаний. Следовательно, включение в проект способности к обучению позволяет проектировать унифицированных рациональных агентов, которые будут способны успешно действовать в исключительно разнообразных вариантах среды.

2.3. Свойства окружающей среды

Теперь, когда у нас есть определение рациональности, почти все готово к тому, чтобы приступить к созданию рациональных агентов. Однако сначала следует внимательнее рассмотреть понятие ▶**проблемной среды**, по сути представляющей собой "проблему", для которой рациональный агент служит "решением". Начнем с демонстрации того, как определить среду задачи, и проиллюстрируем этот процесс на ряде примеров. Затем будет показано, что проблемная среда может иметь целый ряд разновидностей. Свойства окружающей среды непосредственно влияют на выбор проектного решения в отношении программы агента.

2.3.1. Определение проблемной среды

Выше, при обсуждении рациональности простого агента-пылесоса, нам потребовалось определить показатели производительности, среду, а также исполнительные механизмы и датчики этого агента. Все это можно объединить под заголовком **проблемная среда**. Для тех, кто любит аббревиатуры, можно привести соответствующее сокращение: ▶**PEAS** (**P**erformance, **E**nvironment, **A**ctuators, **S**ensors — производительность, среда, исполнительные механизмы, датчики). При разработке любого агента первый этап всегда должен заключаться в определении проблемной среды с наибольшей возможной полнотой.

Мир пылесоса — это очень простой пример, поэтому давайте рассмотрим более сложную задачу: автоматизированный водитель такси. На рис. 2.4 представлено общее описание PEAS для такого такси. Ниже каждый элемент этого описания будет рассмотрен более подробно.

Итак, какими могут быть **показатели производительности**, к которым должен будет стремиться наш автоматизированный водитель? Желательные качества включают успешное достижение пункта назначения; минимизацию расхода топлива и износа автомобиля; минимизацию времени поездки или ее стоимости; минимизацию нарушений правил дорожного движения и числа помех, создаваемых для других водителей; максимизацию безопасности и комфорта пассажиров; максимизацию прибыли. Очевидно, что некоторые из этих целей противоречат друг другу, поэтому потребуется найти некий компромисс.

Тип агента	Показатели производительности	Среда	Исполнительные механизмы	Датчики
Водитель такси	Безопасная, быстрая и комфортная езда с соблюдением правил дорожного движения, максимизация прибыли, минимизация влияния на других участников дорожного движения	Дороги, различные транспортные средства, полиция, пешеходы, клиенты, погода	Рулевое управление, акселератор, тормоза, клаксон, световые указатели, дисплей, речь	Видеокамеры, дальномер, спидометр, одометр, акселерометр, датчик GPS, датчики двигателя, микрофоны, сенсорный экран

Рис. 2.4. PEAS-описание проблемной среды для автоматизированного водителя такси

Далее, что представляет собой дорожная **среда**, в которой будет функционировать водитель такси? Любому водителю такси приходится иметь дело с различными дорогами, начиная от сельских улочек и городских переулков до 12-полосных автострад. На любых дорогах может присутствовать другой движущийся транспорт, пешеходы и бездомные животные, могут проводиться дорожные работы, встречаться полицейские машины, попадаться лужи и выбоины. Таксист также должен взаимодействовать с потенциальными и действительными пассажирами. Существуют и некоторые дополнительные аспекты. Так, для такси в Южной Калифорнии снег редко бывает проблемой, тогда как на Аляске будет меньше дней, когда его нет. Движение на дорогах может быть правосторонним или левосторонним, поэтому придется учитывать этот факт, если требуется, чтобы агент был более гибким и мог успешно функционировать и в Великобритании, Таиланде или Японии. Очевидно, что чем более ограничена среда, тем проще задача проектирования.

В число **исполнительных механизмов** автоматизированного такси обязательно войдут все механизмы, доступные водителю-человеку: управление двигателем посредством педали акселератора, ручное и ножное управление тормозами и рулевое управление. Кроме того, понадобятся средства вывода на экран дисплея или синтезатор голоса, чтобы общаться с пассажирами, и, возможно, какие-то механизмы общениями с другими транспортными средствами, вежливо или наоборот.

Главные **датчики** агента — водителя такси должны включать одну или более видеокамер, чтобы можно было наблюдать за происходящим, а также лидар и ультразвуковые датчики для определения расстояния до других машин и препятствий. Чтобы избежать штрафов за превышение скорости, в такси должен быть спидометр, а для правильного управления машиной, особенно на поворотах, в ней должен быть установлен акселерометр. Для определения текущего состояния оборудования автомобиля достаточно будет обычного набора датчиков в двигателе, топливной и электрической системе. Как и многие люди-водители, агент-таксист может нуждаться в доступе к системе GPS, чтобы не потеряться в пути. И наконец, для получения от пассажира информации о требуемом пункте назначения потребуется сенсорный экран или система голосового ввода.

На рис. 2.5 в общих чертах приведены основные элементы PEAS для целого ряда прочих типов агентов. Дополнительные примеры приведены в упражнениях 2.4 и 2.5. Эти примеры включают как физические, так и виртуальные проблемные среды. Обратите внимание, что виртуальные проблемные среды могут быть такими же сложными, как и "реальный" мир, например ▶**программный агент** (или программный робот либо ▶**программный бот**), который проводит аукцион и перепродает веб-сайты, имеет дело с миллионами других пользователей и миллиардами объектов, многие из которых имеют реальное отображение.

Тип агента	Показатели производительности	Среда	Исполнительные механизмы	Датчики
Медицинская диагностическая система	Выздоровление пациентов, снижение стоимости	Пациенты, больница, персонал	Дисплей вопросов, анализы, диагнозы, лечение, уход	Сенсорный экран, голосовой ввод, система поиска
Система анализа изображений, полученных со спутника	Корректировка, категоризация объектов, территория	Спутник на орбите, система связи, погодные условия	Отображение изображений, категоризация	Цифровая камера с высоким разрешением
Робот—сортировщик деталей	Процент деталей в контейнере годных	Лента конвейера, детали на ленте, контейнеры	Манипулятор с шарнирами, захват	Видеокамера, датчик касания, датчики угла в шарнирах
Контроллер очистительной установки	Степень очистки, производительность, безопасность	Очистительная установка, входное сырье, операторы	Вентили, насосы, нагреватели, смесители, дисплеи	Датчики температуры, давления, потока, химические сенсоры
Интерактивная система обучения английскому языку	Оценки учащихся при прохождении тестов	Группа учащихся, агентство тестирования	Дисплей упражнений, обратная связь, речевой вывод	Ввод с клавиатуры, голосовой ввод

Рис. 2.5. Примеры типов агентов и описания их PEAS

2.3.2. Свойства проблемной среды

Очевидно, что разнообразие вариантов проблемной среды, которые могут возникать в приложениях ИИ, огромно. Тем не менее существует возможность определить относительно небольшое количество измерений, по которым варианты проблемной среды можно будет классифицировать. Эти измерения в значительной степени определяют наиболее подходящую конструкцию агента и применимость каждого из основных семейств методов для его реализации. В этом разделе сначала будет приведен список предлагаемых измерений, а затем проанализировано несколько вариантов проблемной среды для иллюстрации этих идей. Приведенные здесь определения являются неформальными; более точные определения и примеры вариантов среды каждого типа будут описаны в последующих главах.

▶ **Полностью наблюдаемая** или ▶ **частично наблюдаемая** среда. Если датчики агента обеспечивают ему доступ к полной информации о состоянии среды в каждый момент времени, такая проблемная среда называется полностью наблюдаемой. Фактически проблемная среда будет полностью наблюдаемой, если датчики анализируют все ее аспекты, *релевантные* для выбора агентом действия; при этом релевантность, в свою очередь, зависит от показателей производительности. Полностью наблюдаемые среды очень удобны, поскольку агенту не требуется поддерживать какое-либо внутреннее состояние для того, чтобы быть в курсе всего происходящего в его мире. Среда может оказаться частично наблюдаемой из-за шума и неточных датчиков или из-за того, что отдельные характеристики ее состояния просто отсутствуют в информации, поступающей от датчиков. Например, агент-пылесос, в котором имеется только локальный датчик мусора, не может определить, есть ли мусор в других квадратах, а автоматизированный водитель такси не имеет сведений о том, какие маневры намереваются выполнить на дороге другие водители. Если агент вообще не имеет датчиков, то среда является ▶ **ненаблюдаемой**. Можно подумать, что в таких случаях положение агента безнадежно, но, как будет показано в главе 4, цели агента все еще могут оказаться достижимыми, иногда с уверенностью.

▶ **Одноагентная** или ▶ **мультиагентная** среда. Различие между одноагентными и мультиагентными средами, на первый взгляд, может показаться достаточно простым. Например, очевидно, что агент, самостоятельно решающий кроссворд, находится в одноагентной среде, а агент, играющий в шахматы, действует в двухагентной среде. Однако здесь имеется несколько тонких нюансов. Во-первых, выше было указано, на каком основании некоторая сущность *может* рассматриваться как агент, но не было пояснено, какие сущности *должны* рассматриваться как агенты. Должен ли агент A (например, водитель такси) считать агентом объект B (другой автомобиль), или он может относиться к нему просто как к объекту, поведение которого определяется законами физики, подобно волнам, набегающим на берег, или листьям, трепещущим на ветру? Ключевое различие состоит в том, следует или не следует описывать поведение объекта B как направленное на максимизацию его собственных показателей производительности, величина которых зависит от поведения агента A.

Например, в шахматах соперничающая сущность B пытается максимизировать свои показатели производительности, а по правилам шахмат это ведет к минимизации показателей производительности агента A. Следовательно, шахматы — это ▶ **конкурентная** мультиагентная среда. С другой стороны, в проблемной среде агента — водителя такси предотвращение столкновений максимизирует показатели производительности всех агентов, поэтому она может служить примером частично ▶ **кооперативной** мультиагентной среды. Она также является частично конкурентной, поскольку, например, каждую парковочную площадку может занять только один автомобиль.

Проблемы проектирования агентов, действующих в мультиагентной среде, часто совершенно отличаются от проблем агентов, с которыми приходится иметь дело в одноагентных проблемных средах. Например, в мультиагентной среде одним из признаков рационального поведения часто бывает поддержка коммуникации между агентами, а в некоторых вариантах частично наблюдаемых конкурентных сред рациональным будет стохастическое, случайное поведение, поскольку оно позволяет избежать ловушек предсказуемости.

▶ **Детерминированная** или ▶ **недетерминированная** среда. Если следующее состояние среды полностью определяется текущим состоянием и действием, выполненным агентом, то такая среда называется детерминированной; в противном случае она является недетерминированной. В принципе, в полностью наблюдаемой детерминированной среде агенту не приходится действовать в условиях неопределенности. Но если среда — частично наблюдаемая, то может *создаться впечатление*, что она является недетерминированной.

Большинство реальных ситуаций являются настолько сложными, что просто невозможно отслеживать историю всех их ненаблюдаемых аспектов, и из практических соображений их следует рассматривать как недетерминированные. В этом смысле проблемная среда агента — водителя такси, очевидно, будет недетерминированной, поскольку никто не может точно предсказать поведение других участников движения. Более того, даже в собственном автомобиле неожиданно может произойти прокол шины или его двигатель откажет без какого-либо предупреждения. Мир пылесоса в том виде, в каком он был описан выше, является детерминированным, но другие варианты этой среды могут включать недетерминированные элементы, такие как случайное появление мусора и ненадежная работа механизма всасывания пылесоса (см. упражнение 2.15).

И еще одно, последнее, замечание: в отношении свойств среды многие используют слово ▶ **стохастическая** как синоним слова "недетерминированная", но авторы данной книги проводят различие между этими двумя терминами. Мы говорим, что модель среды является *стохастической*, если в ее определении явно присутствуют вероятности (например, "завтра возможен дождь с вероятностью 25%") и *недетерминированной*, если потенциальные возможности перечисляются без количественного определения (например, "завтра есть вероятность дождя").

▶ **Эпизодическая** или ▶ **последовательная** среда. В эпизодической проблемной среде опыт агента состоит из неразрывных эпизодов. Каждый эпизод включает в себя сначала восприятие среды агентом, а затем выполнение одного действия. При этом очень важно то, что следующий эпизод не зависит от действий, предпринятых в предыдущих эпизодах. Эпизодическими являются многие задачи классификации. Например, агент, который должен распознавать дефектные детали на сборочной линии, формирует каждое решение применительно к текущей детали независимо от предыдущих решений; более того, от текущего решения не зависит то, будет ли определена как дефектная следующая деталь. С другой стороны, в последовательных средах текущее решение может повлиять на все будущие

решения.[4] Последовательными являются такие задачи, как игра в шахматы и вождение такси: в обоих случаях кратковременные действия агента могут иметь долговременные последствия. Эпизодические среды намного проще последовательных, поскольку в них агенту не нужно думать наперед.

▶ **Статическая** или ▶ **динамическая** среда. Если среда может измениться за то время, пока агент выбирает очередное действие, то такая среда называется динамической для данного агента. В противном случае она является статической. Действовать в условиях статической среды проще, поскольку агенту не требуется наблюдать за миром в процессе выработки решения в отношении очередного действия, к тому же нет необходимости беспокоиться о затраченном на это времени. С другой стороны, динамическая среда как бы непрерывно спрашивает у агента, что он собирается делать, и если он еще ничего не решил, то это воспринимается как решение ничего не делать. Если сама среда с течением времени не изменяется, а изменяются лишь показатели производительности агента, то такую среду принято называть ▶ **полудинамической**. Очевидно, что среда вождения такси является динамической, поскольку другие автомобили и само такси продолжают движение в то время, когда алгоритм вождения определяет, что делать дальше. Игра в шахматы с контролем времени является полудинамической, а задача решения кроссворда — статической.

▶ **Дискретная** или ▶ **непрерывная** среда. Различие между дискретным и непрерывным типами среды может относиться к *состоянию* среды, способу учета *времени*, а также к *восприятию* и *действию* агента. Например, среда игры в шахматы имеет конечное количество различных состояний (без учета времени). Также игра в шахматы связана с дискретным множеством восприятий и действий. Вождение такси — это проблемная среда с непрерывно меняющимся состоянием и непрерывно текущим временем, поскольку скорость и местонахождение самого такси и других транспортных средств изменяются в определенном диапазоне непрерывных значений, причем эти изменения происходят во времени плавно. Действия по вождению такси также являются непрерывными (непрерывная регулировка угла поворота руля и т.д.). Строго говоря, входные данные от цифровых видеокамер поступают дискретно, но обычно рассматриваются как представляющие непрерывно изменяющиеся интенсивности и положения.

▶ **Известная** и ▶ **неизвестная** среда. Строго говоря, это различие относится не к окружающей среде самой по себе, а к состоянию знаний агента (или его разработчика) о действующих в ней "законах физики". В известной среде результаты (или вероятности исхода, если среда является недетерминированной) определены для всех действий. Очевидно, что, если среда неизвестна, агент должен будет изучить, как она функционирует, чтобы принимать правильные решения.

---

[4] Термин "последовательный" используется также в компьютерных науках как антоним термина "параллельный". Эти два толкования одного термина никак не связаны одно с другим.

Различие между известными и неизвестными средами вовсе не такое, как между полностью наблюдаемой и частично наблюдаемой средами. Вполне возможно, что *известная* среда может быть *частично* наблюдаемой, например в карточном пасьянсе игроку известны все правила, но он еще не видит карт, которые пока не были перевернуты. И наоборот, *неизвестная* среда может быть *полностью* наблюдаемой, — в новой видеоигре ее интерфейс может быть представлен на экране полностью, но назначение отдельных его элементов будет неизвестно, пока они не будут опробованы.

Как было отмечено в разделе 2.2.1, сам показатель производительности может быть пока неизвестен: либо потому, что разработчик не знает, как правильно его описать, либо потому, что конечный пользователь — чьи предпочтения наиболее важны — пока неизвестен. Например, водитель такси обычно не знает, что предпочитает новый пассажир: неторопливую или достаточно быструю езду, осторожный или агрессивный стиль вождения. Виртуальный личный помощник начинает свою работу, ничего не зная о личных предпочтениях своего нового владельца. В подобных случаях агент может узнать больше о требуемых показателях производительности, лишь исходя из последующих взаимодействий с разработчиком или пользователем. Такой подход, в свою очередь, предполагает, что проблемная среда обязательно должна рассматриваться как мультиагентная.

Наиболее сложный вариант среды — это *частично наблюдаемая*, *мультиагентная*, *недетерминированная*, *последовательная*, *динамическая*, *непрерывная* и *неизвестная* среда. Задача вождения такси является сложной во всех этих смыслах, за исключением того, что эта проблемная среда водителю в основном известна. Вождение взятого напрокат автомобиля в новой стране с незнакомой географией, иными правилами дорожного движения и при наличии нервных пассажиров является задачей, намного более впечатляющей.

На рис. 2.6 перечислены свойства ряда знакомых сред. Обратите внимание, что в отдельных случаях приведенные в ней описания являются слишком краткими и сухими. Например, задача медицинской диагностики указана как одноагентная, поскольку ход болезни пациента нецелесообразно моделировать как агента, однако в действительности медицинская диагностическая система вполне может иметь дело с отказывающимися подчиняться пациентами либо со скептицизмом медицинского персонала, в результате чего эта среда может приобретать свойства мультиагентной. Кроме того, задача медицинской диагностики будет иметь эпизодический характер, если ее назначение заключается лишь в постановке диагноза по заданному перечню симптомов, но она будет последовательной, если предусматривается выполнение серии анализов, оценка прогресса в ходе лечения, работа с группой пациентов и т.д.

На рис. 2.6 нет столбца "Известная/неизвестная", поскольку, как уже объяснялось выше, эта характеристика не определяется исключительно самой проблемной средой. Для некоторых сред, таких как шахматы или покер, будет довольно просто предоставить агенту полные знания о действующих в них правилах, тем не менее

было бы интересно посмотреть, как агент сможет научиться играть в эти игры без этих знаний.

Репозиторий кода, относящийся к данной книге (`aima.cs.berkeley.edu`), включает реализации нескольких вариантов проблемных сред, а также имитатор среды общего назначения, предназначенные для оценки производительности агента. Такие эксперименты часто выполняются применительно не к одной среде, а ко многим ее вариантам, выбранным из некоторого ▶ **класса вариантов среды**. Например, чтобы оценить действия водителя такси в моделируемой ситуации дорожного движения, может потребоваться провести много сеансов моделирования с различными условиями трафика, освещения и погоды. В конечном счете нас будут интересовать средние показатели производительности агента для выбранного класса вариантов сред.

Среда проблемы	Наблюдаемость	Агенты	Тип	Повторяемость	Статичность	Дискретность
Решение кроссворда	Полная	Один	Детерминированная	Последовательная	Статическая	Дискретная
Игра в шахматы с контролем времени	Полная	Много	Детерминированная	Эпизодическая	Полудинамическая	Дискретная
Игра в покер	Частичная	Много	Недетерминированная	Последовательная	Статическая	Дискретная
Игра в нарды	Полная	Много	Недетерминированная	Последовательная	Статическая	Дискретная
Вождение такси	Частичная	Много	Недетерминированная	Последовательная	Динамическая	Непрерывная
Медицинская диагностика	Частичная	Один	Недетерминированная	Последовательная	Динамическая	Непрерывная
Анализ изображений	Полная	Один	Детерминированная	Эпизодическая	Полудинамическая	Непрерывная
Робот — сборщик деталей	Частичная	Один	Недетерминированная	Эпизодическая	Динамическая	Непрерывная
Система контроля очистки	Частичная	Один	Недетерминированная	Последовательная	Динамическая	Непрерывная
Система обучения английскому языку	Частичная	Много	Недетерминированная	Последовательная	Динамическая	Дискретная

Рис. 2.6. Примеры проблемных сред и их характеристики

2.4. Структура агентов

До сих пор обсуждение агентов проводилось посредством анализа их поведения — действий, выполняемых агентом после получения любой заданной последовательности восприятия. Теперь нам поневоле придется сменить тему и перейти к обсуждению того, как организовано их внутреннее функционирование. Задача искусственного интеллекта состоит в разработке ▶ **программы агента**, реализующей функцию агента, т.е. отображающей последовательность восприятия на

действия. Будем предполагать, что эта программа должна работать в своего рода вычислительном устройстве с физическими датчиками и исполнительными механизмами. В целом совокупность этих компонентов именуется в данной книге ▶**архитектурой агента**. Таким образом, структуру агента можно представить следующей формулой:

agent = *architecture* + *program*.

Очевидно, что выбранная программа должна быть подходящей для существующей архитектуры. Так, если в программе предполагается выдача таких указаний, как *Walk* (идти), то в архитектуре следует предусмотреть наличие ног. Архитектура может представлять собой обычный персональный компьютер или же может быть реализована в виде роботизированного автомобиля с несколькими бортовыми компьютерами, видеокамерами и другими датчиками. В общем случае архитектура должна обеспечить передачу в программу результатов восприятия, поступающих от датчиков, выполнение самой программы агента и передачу исполнительным механизмам выбранных программой вариантов действий по мере их генерации. Основная часть данной книги посвящена проектированию программ агентов, и только главы 24 и 25 касаются непосредственно датчиков и исполнительных механизмов.

2.4.1. Программы агентов

Все программы агентов, которые будут разработаны в этой книге, имеют одну и ту же структуру: они принимают от датчиков в качестве входных данных результаты текущего восприятия и возвращают исполнительным механизмам выбранный вариант действия.[5] Необходимо указать на различие между программой агента, которая принимает в качестве входных данных текущие восприятия, и функцией агента, которая принимает на входе всю историю восприятия. Программа агента получает в качестве входных данных только результаты текущего восприятия, поскольку больше ничего не может узнать из своей среды; если действия агента зависят от всей последовательности актов восприятия, то агент должен сам запоминать результаты этих актов восприятия.

Для описания программ агентов будет использоваться простой язык псевдокода, определенный в приложении Б. (Интерактивный репозиторий кода содержит реализации программ на реальных языках программирования.) Например, на рис. 2.7 представлена довольно простая программа агента, которая регистрирует последовательность восприятия, а затем использует полученную последовательность для доступа по индексу к таблице действий с целью определения, что нужно сделать. Таблица — в данном случае из примера для мира пылесоса,

[5] Существуют и другие варианты структуры программы агента, например можно было бы использовать в виде программ агентов сопроцедуры, которые асинхронно взаимодействуют со средой. Каждая такая сопроцедура имеет входной и выходной порты, а ее работа организована в виде цикла, в котором из входного порта считываются результаты восприятий, а в выходной порт записываются варианты действий.

представленного на рис. 2.3 — явно представляет функцию агента, воплощаемую данной программой агента. Чтобы создать рационального агента таким способом, проектировщики должны сформировать таблицу, которая содержит подходящее действие для любой возможной последовательности актов восприятия.

---

function TABLE-DRIVEN-AGENT(*percept*) **returns** действие *action*
persistent: *percepts*, последовательность восприятия, исходно пустая
table, таблица действий, индексированная по последовательностям восприятия, исходно полностью определенная

добавить результат воспрятия *percept* в конец последовательности *percepts*
action ← LOOKUP (*percepts*, *table*)
return *action*

Рис. 2.7. Программа TABLE-DRIVEN-AGENT вызывается для каждого нового восприятия и каждый раз возвращает действие. Она сохраняет полную последовательность восприятия в памяти

Поучительно разобраться в том, почему табличный подход к конструированию агента неизбежно обречен на провал. Пусть $\mathcal{P}$ — набор всех возможных восприятий и пусть T — срок существования агента (общее количество актов восприятия, которое он может получить). В этом случае таблица поиска будет содержать $\sum_{t=1}^{T} |\mathcal{P}|^t$ записей. Рассмотрим случай автоматизированного такси: визуальные входные данные с одной видеокамеры (обычно их восемь) поступают со скоростью примерно 70 Мбайт/с (30 кадров в секунду, в кадре 1080 × 720 пикселей с 24 битами цветовой информации). На один час езды потребуется справочная таблица более чем с $10^{600\,000\,000\,000}$ записей! Даже для игры в шахматы (крошечного, хорошо изученного фрагмента реального мира) поисковая таблица должна содержать не менее 10^{150} записей. Для сравнения, количество атомов в наблюдаемой части Вселенной оценивается менее чем в 10^{80}. Ошеломляющий размер этих таблиц означает, что: *а*) ни один физический агент в этой Вселенной не будет иметь достаточно места для хранения подобной таблицы; *б*) любому проектировщику просто не хватит времени для ее создания; и *в*) ни один агент не сможет обучиться всему, что содержится во всех правильных записях таблицы на собственном опыте.

Однако, несмотря на все сказанное выше, программа TABLE-DRIVEN-AGENT действительно делает то, что мы хотели (при условии, что таблица заполнена правильно): реализует желаемую функцию агента.

➔ *Ключевая задача ИИ заключается в выяснении того, как создавать программы, которые в рамках возможного обеспечивают рациональное поведение агента с использованием небольшого объема программного кода, а не обширных таблиц с множеством записей.*

Существует много примеров, свидетельствующих о том, что подобную задачу можно успешно решить в других областях. Например, огромные таблицы квадратных корней, которыми пользовались инженеры и школьники до 1970-х годов, теперь заменены программой из пяти строк, работающей в электронных калькуляторах, — для выполнения расчетов в ней применяется метод Ньютона. Вопрос заключается в том, могут ли достигнутые в области ИИ результаты предоставить для интеллектуального поведения в целом то, что Ньютон сделал для упрощения вычисления квадратных корней? Авторы данной книги полагают, что ответ на этот вопрос является положительным.

В остальной части этого раздела обсуждаются четыре основных вида программ агентов, воплощающих принципы, лежащие в основе почти всех интеллектуальных систем.

- Агенты с простым рефлекторным поведением
- Агенты с поведением, основанным на модели
- Агенты, действующие на основе цели
- Агенты, действующие на основе полезности

В агентской программе каждого вида комбинируются определенные компоненты, в частности способы генерации действий. В разделе 2.4.6 в общих словах объясняется, как преобразовать агентов всех этих типов в *обучающихся агентов*, способных повысить производительность своих компонентов, чтобы генерировать лучшие действия. И наконец, в разделе 2.4.7 описываются различные способы, позволяющие представить сами компоненты и их работу в агенте. Все это разнообразие обусловливает основной принцип организации как области исследований ИИ, так и самой этой книги.

2.4.2. Агенты с простым рефлекторным поведением

Простейшим видом агента является ▶**простой рефлекторный агент**. Такие агенты выбирают действия на основе *текущего* восприятия, игнорируя всю остальную историю восприятия. Например, агент-пылесос, таблица функций которого представлена на рис. 2.3, является простым рефлекторным агентом, поскольку его решения основаны только на информации о текущем местоположении и о том, содержит ли оно мусор. Программа для данного агента приведена на рис. 2.8.

Обратите внимание, что программа агента-пылесоса действительно очень невелика в сравнении с таблицей функций этого агента. Наиболее очевидное сокращение произошло в результате игнорирования истории восприятия, что позволило уменьшить количество анализируемых последовательностей восприятия с 4^T до просто 4. Еще одно небольшое уменьшение связано с тем фактом, что, когда в текущем квадрате есть мусор, выбираемое действие не зависит от местоположения. Хотя программа агента была записана с использованием операторов **if-then-else**, она достаточно проста, чтобы ее можно было реализовать в виде логической схемы.

function REFLEX-VACUUM-AGENT([*location*, *status*]) **returns** действие *action*

if *status* = *Dirty* **then return** *Suck*
else if *location* = *A* **then return** *Right*
else if *location* = *B* **then return** *Left*

Рис. 2.8. Программа агента для простого рефлекторного агента в мире пылесоса с двумя помещениями. Эта программа реализует функцию агента, представленную на рис. 2.3

Простое рефлекторное поведение встречается и в более сложных средах. Представьте себя в качестве водителя автоматизированного такси. Если автомобиль перед вами стал тормозить и его стоп-сигналы засветились, вы должны заметить это и также немедленно начать торможение. Другими словами, необходима определенная обработка визуальных сигналов для выявления ситуации, которую мы обозначим как "*car-in-front-is-braking*" — автомобиль впереди тормозит. Ее обнаружение инициирует в программе агента некоторую связь с действием *initiate-braking*, т.е. начать торможение. Подобную связь называют ▶ **правилом "условие–действие"**,[6] которое можно сформулировать как

if *car-in-front-is-braking* **then** *initiate-braking*.

Люди также обладают большим количеством таких связей, причем одни из них представляют собой сложные реакции, освоенные в результате обучения (как при вождении автомобиля), а другие являются врожденными рефлексами (например, моргание при приближении к глазу постороннего предмета). В последующих главах этой книги будет представлено несколько различных способов, с помощью которых можно организовать обучение агента и реализацию таких связей.

Программа, приведенная на рис. 2.8, предназначена для одной конкретной среды мира пылесоса. Более общий и гибкий подход состоит в том, чтобы вначале создать интерпретатор общего назначения для правил "условие–действие", а затем определить наборы правил для конкретной проблемной среды. На рис. 2.9 приведена структура такой общей программы в схематической форме и показано, каким образом правила "условие–действие" позволяют агенту создать связь от восприятия к действию. (Не следует беспокоиться, если такой способ покажется тривиальным; вскоре он обнаружит намного более интересные возможности.)

Программа агента, представленного на рис. 2.9, приведена на рис. 2.10. Функция INTERPRET-INPUT вырабатывает абстрагированное описание текущего состояния по результатам восприятия, а функция RULE-MATCH возвращает первое правило из общего набора правил, соответствующее заданному описанию состояния. Обратите внимание, что приведенное здесь описание в терминах "правил" и

[6] Эти связи также могут называться правилами **ситуация–действие**, **продукциями** или **правилами if-then**.

"соответствия" является чисто концептуальным. Как уже отмечалось выше, действительные реализации могут быть такими же простыми, как и совокупность логических элементов, реализующих логическую схему. В качестве альтернативы может быть использована "нейронная" схема, в которой логические элементы заменены нелинейными элементами искусственных нейронных сетей (см. главу 21).

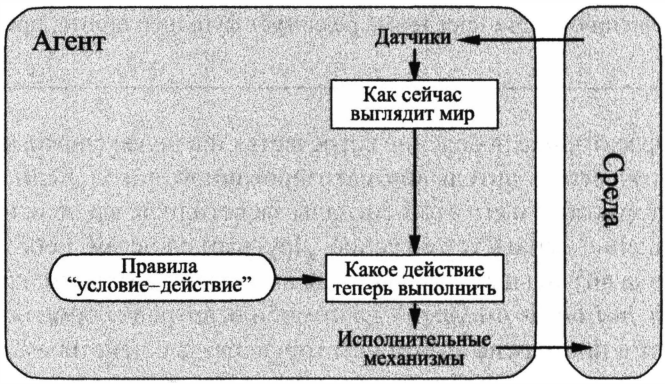


Рис. 2.9. Схематическое представление простого рефлекторного агента. Здесь прямоугольники использованы для обозначения текущего внутреннего состояния процесса принятия решения агентом, а овалы — для представления фоновой информации, используемой в этом процессе

Простые рефлекторные агенты имеют замечательное свойство быть действительно очень простыми, но при этом обладают ограниченным интеллектом. Агент, программа которого приведена на рис. 2.10, будет работать, ➨*только если правильное решение может быть принято на основе исключительно текущего восприятия, т.е. только в том случае, когда среда является полностью наблюдаемой.*

```
function SIMPLE-REFLEX-AGENT(percept) returns действие action
  persistent: rules, множество правил "условие–действие"

  state ← INTERPRET-INPUT(percept)
  rule ← RULE-MATCH(state, rules)
  action ← rule.ACTION
  return action
```

Рис. 2.10. Простой рефлекторный агент. Он действует в соответствии с правилом, условие которого соответствует текущему состоянию, определяемому восприятием

Внесение даже небольшой доли ненаблюдаемости может вызвать серьезное нарушение в работе такого агента. Например, в приведенном выше правиле

торможения принято предположение, что условие *car-in-front-is-braking* может быть определено из текущего восприятия (единственного кадра видеоизображения). Это возможно, если стоп-сигналы у движущегося впереди автомобиля расположены некоторым стандартным образом (а значит, однозначно идентифицируются). К сожалению, очень часто более старые модели имеют другие конфигурации расположения габаритных огней, сигналов поворота и стоп-сигналов, поэтому не всегда по единственному изображению можно с уверенностью определить, тормозит этот автомобиль или нет. Простой рефлекторный агент, ведущий автомобиль вслед за таким автомобилем, либо будет часто тормозить без какой-либо необходимости, либо, что еще хуже, вообще не станет тормозить.

Появление подобной проблемы можно отметить и в мире пылесоса. Предположим, что у простого рефлекторного агента-пылесоса отказал датчик местонахождения и работает только датчик мусора. Такому агенту будут доступны только два возможных восприятия: [*Dirty*] и [*Clean*]. Он может выполнить действие *Suck* в ответ на восприятие [*Dirty*], но что он должен делать в ответ на восприятие [*Clean*]? Выполнение действия *Left* приведет к отказу (и надолго), если окажется, что он предпринял это действие в квадрате *A*, а действие *Right* приведет к таким же результатам, если будет выполняться в квадрате *B*. Для простых рефлекторных агентов, действующих в частично наблюдаемых средах, часто неизбежно попадание в бесконечные циклы.

Выход из бесконечных циклов становится возможным, если агент обладает способностью ▶ **рандомизировать** свои действия, т.е. вводить в них элемент случайности. Например, если агент-пылесос получает результат восприятия [*Clean*], то может подбросить монету, чтобы выбрать между действиями *Left* и *Right*. Легко показать, что агент достигнет другого квадрата в среднем за два этапа. Затем, если в этом квадрате имеется мусор, пылесос его уберет, и задача очистки будет выполнена. Поэтому рандомизированный простой рефлекторный агент может превзойти по своей производительности детерминированного простого рефлекторного агента.

Выше (в разделе 2.3) отмечалось, что в некоторых мультиагентных вариантах среды может оказаться рациональным рандомизированное поведение подходящего типа, тогда как в одноагентных вариантах среды рандомизация обычно не является рациональной. Это лишь полезный прием, который может помочь простому рефлекторному агенту в некоторых особых ситуациях. В большинстве случаев добиться гораздо большего можно с помощью более сложных детерминированных агентов.

2.4.3. Агенты с поведением, основанным на модели

Наиболее эффективный способ действия агента в условиях частичной наблюдаемости — *отслеживать ту часть мира, которую он в данный момент не видит*. Это означает, что агент должен поддерживать некоторого рода ▶ **внутреннее состояние**, которое будет зависеть от истории восприятия и поэтому отражать по крайней мере некоторые ненаблюдаемые аспекты текущего состояния. Для проблемы торможения поддержка внутреннего состояния не требует слишком

больших затрат — достаточно лишь сохранить предыдущий кадр, отснятый видеокамерой. Это позволит агенту определять моменты, когда два красных световых сигнала с обеих сторон идущего впереди автомобиля одновременно загораются или гаснут. Для решения других задач вождения, таких как переход с одной полосы движения на другую, агент должен следить за тем, где находятся другие автомобили, если он не может видеть все эти автомобили одновременно. А для того, чтобы вождение было вообще возможно, агенту необходимо отслеживать, где находятся ключи от его автомобиля.

Для обновления этой внутренней информации о состоянии во времени требуется, чтобы в программе агента в некоторой форме были закодированы знания двух видов. Во-первых, нужна определенная информация о том, как мир изменяется во времени, которую можно разделить примерно на две части: результаты действий агента и как мир изменяется независимо от агента. Например, когда агент поворачивает рулевое колесо по часовой стрелке, автомобиль поворачивает направо, а когда идет дождь, видеокамеры автомобиля могут быть забрызганы. Подобные знания о том, "как работает мир", реализованные в виде простых логических схем или сложных научных теорий, называются ▶**моделью мира**.

Во-вторых, нам нужна информация о том, как состояние мира отражается в восприятии агента. Например, если автомобиль впереди начинает торможение, на изображении с видеокамеры, обращенной вперед, появляются одна или больше подсвеченных красных областей, а если любая видеокамера забрызгана, на полученном с нее изображении появляются каплеобразные объекты, частично закрывающие дорогу. Этот вид знаний называют ▶**моделью сенсоров**.

Используемые совместно, модель мира и модель сенсоров позволяют агенту сохранить необходимую ему историю состояния мира, насколько это возможно с учетом ограничений, налагаемых его датчиками. Агент, действия которого определяются с использованием подобных моделей, называется ▶**агентом с поведением, основанным на модели**.

На рис. 2.11 приведена структура рефлекторного агента с поведением, основанным на модели, действующего с учетом внутреннего состояния, и показано, как текущее восприятие комбинируется с прежним внутренним состоянием для выработки обновленного описания текущего состояния на основании модели мира агента. Программа такого агента приведена на рис. 2.12. Особый интерес в ней представляет функция UPDATE-STATE, отвечающая за создание нового описания внутреннего состояния. Детали того, как представляются модели и состояния, варьируются в широких пределах в зависимости от типа среды и конкретной технологии, используемой при разработке агента.

Вне зависимости от вида используемого представления, агенту редко удается *точно* определить текущее состояние частично наблюдаемой среды. Поэтому прямоугольник с надписью "Как сейчас выглядит мир" (см. рис. 2.11) в действительности содержит "лучшее предположение" агента об этом (или иногда "лучшие предположения", если агент способен поддерживать несколько возможных

вариантов). Например, автоматизированное такси может оказаться не в состоянии увидеть, что происходит за большим грузовиком, который остановился перед ним, и может только гадать о том, что могло послужить причиной задержки. Таким образом, может оказаться просто невозможно избежать неопределенности в отношении текущего состояния среды, но агент все равно должен принять решение.

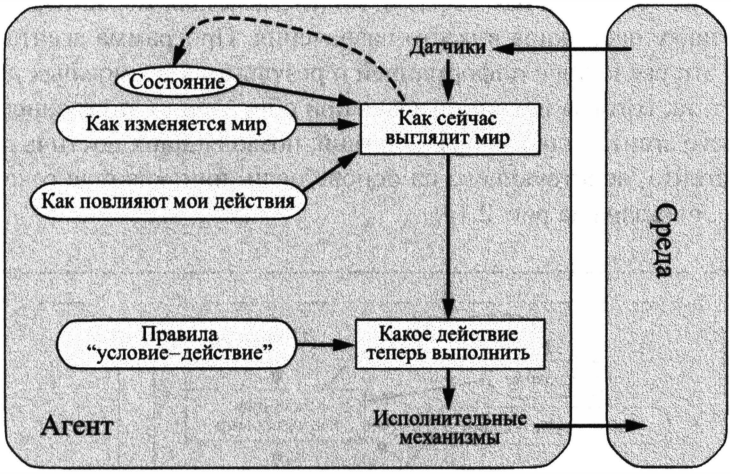


Рис. 2.11. Структура рефлекторного агента с поведением, основанным на модели

function MODEL-BASED-REFLEX-AGENT(*percept*) **returns** действие *action*
persistent: *state*, представление агента о текущем состоянии мира
transition_model, описание, как следующее состояние мира зависит от его текущего состояния и действия
sensor_model, описание, как текущее состояние мира отражается в восприятии агента
rules, множество правил "условие–действие"
action, последнее по времени действие; исходно не определено

state ← UPDATE-STATE(*state*, *action*, *percept*, *transition_model*, *sensor_model*)
rule ← RULE-MATCH(*state*, *rules*)

Рис. 2.12. Рефлекторный агент с поведением, основанным на модели. Он отслеживает текущее состояние мира, используя внутреннюю модель, а затем выбирает действие таким же образом, как и простой рефлекторный агент

2.4.4. Агенты, действующие на основе цели

Просто знания чего-либо о текущем состоянии среды не всегда может быть достаточно для принятия решения о том, что делать. Например, на перекрестке такси может повернуть налево, повернуть направо или поехать прямо. Правильное решение зависит от того, куда это такси должно добраться. Иными словами, агенту требуется не только описание текущего состояния среды, но и информация о цели, представленная тем или иным образом и описывающая желаемые ситуации, например доставку пассажира в место назначения. Программа агента может комбинировать эти сведения с информацией о результатах возможных действий (такой же, как и та, которая использовалась при обновлении внутреннего состояния рефлекторного агента) для выбора действий, позволяющих достичь данной цели. Структура агента, действующего на основе цели, или, как еще говорят, *целенаправленного*, показана на рис. 2.13.

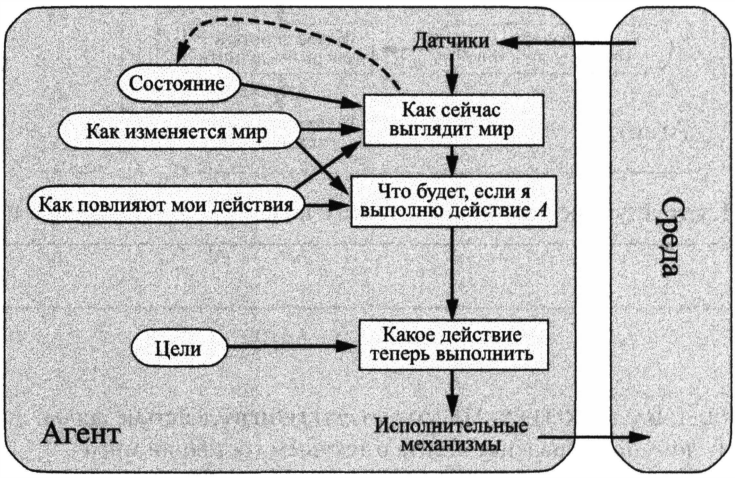


Рис. 2.13. Агент, действующий на основе модели и цели. Он следит за состоянием мира, а также за поставленными перед ним целями, которых он пытается достичь, и выбирает действие, позволяющее (в конечном итоге) добиться достижения этих целей

Иногда выбор целевого действия является простым, например когда цель достигается сразу, после единственного действия. А иногда он сложнее, например когда агенту приходится рассматривать длинную последовательность возможных действий и необходимых условий, чтобы найти способ достижения цели. Для решения столь сложных задач агенту приходится применять различные методы **▶поиска** и **▶планирования** — подобластей ИИ, подробно обсуждаемых в главах 3–6 и 11–12 соответственно.

Обратите внимание, что процедура принятия решений такого рода в корне отличается от описанной выше процедуры применения правил "условие–действие". В этом случае агенту приходится размышлять о будущем: "Что произойдет, если я сделаю то-то и то-то?" и "Позволит ли это мне достичь успеха?" В конструкции рефлекторных агентов такая информация не представлена явно, поскольку встроенные правила устанавливают непосредственное соответствие между восприятиями и действиями. Обнаружив сигналы торможения движущегося впереди автомобиля, рефлекторный агент тормозит, и точка. У него нет никакого представления о том, *почему* он это делает. Целенаправленный агент, увидев стоп-сигналы, тормозит потому, что это единственное действие, которое, как он предполагает, позволит ему достичь одной из своих целей — избежать столкновения с другими автомобилями.

Хотя, на первый взгляд, целенаправленный агент выглядит менее эффективным, он более гибок, поскольку знания, на которые опираются его решения, представлены явно и могут быть модифицированы. Например, поведение целенаправленного агента может легко быть изменено, — чтобы отправить его в другое место назначения, достаточно просто указать его в качестве конечной цели. Правила рефлекторного агента, определяющие, где и в какую сторону он должен поворачивать, а когда необходимо следовать прямо, будут применимыми только для одного пункта назначения; чтобы отправить агента в какое-то другое место, все эти правила придется заменить.

2.4.5. Агенты, действующие на основе полезности

В большинстве сред для выработки качественного поведения одних только целей недостаточно. Например, множество последовательностей действий обеспечат прибытие такси к месту назначения (и тем самым достижение поставленной цели), но одни из них позволят сделать это быстрее, безопаснее, надежнее или дешевле, чем другие. Цели сами по себе позволяют провести лишь простейшее бинарное различие между состояниями "хорошо" и "не хорошо". Более общий показатель производительности должен допускать сравнение различных состояний мира в соответствии с тем, *насколько* хороши они будут для агента. Поскольку определение "хорошо" звучит не очень научно, экономисты и специалисты по компьютерным наукам используют вместо него термин ▶ **полезность**.[7]

Выше уже применялся прием, в котором показатель производительности предусматривал начисление баллов в соответствии с любой заданной последовательностью состояний среды. Такой подход позволяет легко различать более и менее желательные варианты, например, достижения автоматизированным такси пункта назначения. ▶ **Функция полезности** агента, по сути, представляет собой

[7] Термину "полезность" соответствует англоязычный эквивалент "utility", который в данном контексте обозначает "свойство быть полезным", а не какое-то иное значение из числа тех, которые приведены для него в англо-русских словарях.

внутреннее отображение его показателей производительности. В тех случаях, когда внутренняя функция полезности и внешние показатели производительности согласованы между собой, агент, выбирающий действия исходя из максимизации своей функции полезности, будет проявлять рациональное поведение и в соответствии с его внешними показателями производительности.

Следует еще раз подчеркнуть, что это вовсе не *единственно возможный* способ достижения рациональности, — выше уже была представлена рациональная программа агента для мира пылесоса (см. рис. 2.8). Однако агент, действующий на основе полезности (или, как еще говорят, *практичный* агент), как и целенаправленный агент, имеет много преимуществ с точки зрения гибкости и возможности обучения. Кроме того, в двух особых ситуациях, когда цели неоднозначны, практичный агент все же сможет принимать рациональные решения. Первый случай: поставленные цели противоречивы, и только некоторые из них могут быть достигнуты в ущерб другим (например, скорость и безопасность), — функция полезности позволяет найти необходимый компромисс. Второй случай: есть несколько целей, к которым агент может стремиться, но ни одной из них нельзя достичь с уверенностью, — полезность предоставляет возможность сопоставить вероятность успеха с важностью целей.

В реальном мире частичная наблюдаемость и недетерминированность вездесущи и неизбежны, следовательно, принимать решения приходится в условиях неопределенности. Технически говоря, рациональный агент, действующий на основе полезности, будет выбирать действие, максимизирующее **▶ожидаемую полезность** результатов действия, т.е. полезность, которую он ожидает получить в среднем, учитывая вероятности и полезности каждого результата. (В приложении А приведено более точное определение ожидания.) В главе 16 будет показано, что любой рациональный агент должен вести себя так, *как если бы* он обладал функцией полезности, ожидаемое значение которой он пытается максимизировать. Поэтому агент, обладающий *явно заданной* функцией полезности, имеет возможность принимать рациональные решения и способен делать это с помощью алгоритма общего назначения, не зависящего от конкретной максимизируемой функции полезности. Благодаря этому "глобальное" определение рациональности (согласно которому рациональными считаются функции агента, имеющие наивысшую производительность) преобразуется в "локальное" ограничение на конструкцию рациональных агентов, которое может быть выражено в виде простой программы.

Структура агента, действующего на основе полезности (или, иначе говоря, практичного агента), показана на рис. 2.14. Программы агентов, действующих с учетом полезности, приведены в главах 16 и 17, где обсуждается проектирование принимающих решения агентов, способных учитывать неопределенность, свойственную недетерминированным и частично наблюдаемым вариантам среды. Принятие решений в мультиагентной среде также рассматривается в рамках теории полезности, — подробно об этом говорится в главе 18.

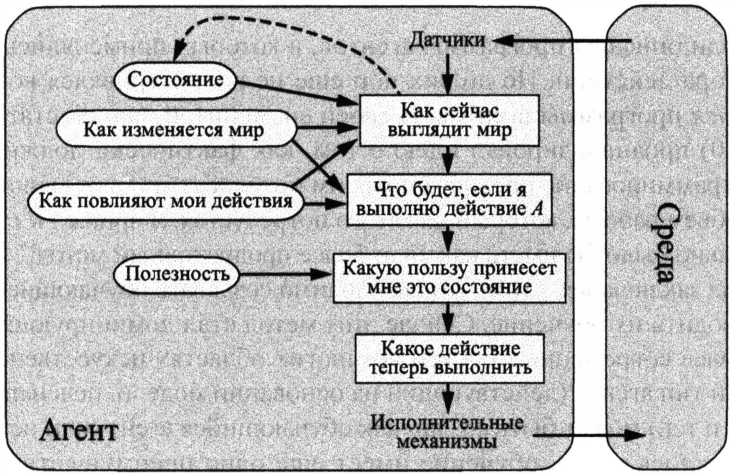


Рис. 2.14. Структура агента, действующего на основе модели и полезности. В нем модель мира используется наряду с функцией полезности, оценивающей предпочтения агента применительно к состояниям мира. Затем агент выбирает действие, которое ведет к наилучшей ожидаемой полезности. Для вычисления ожидаемой полезности выполняется усреднение по всем возможным результирующим состояниям с учетом коэффициента, определяющего вероятность каждого результата

В этот момент читатель может задаться вопросом "Это так просто? Мы всего лишь создаем агентов, которые максимизируют ожидаемую полезность... И это все?" Нет сомнения, что такие агенты были бы интеллектуальны, но это совсем непросто. Практичный агент должен моделировать и отслеживать окружающую среду — задачи, решение которых потребовало проведения большого количества исследований в областях восприятия, представления, рассуждения и обучения. Результаты этих исследований определили содержание многих глав данной книги. Выбор курса на максимизацию полезности действий также является трудной задачей, требующей использования хитроумных алгоритмов, описание которых составляет содержание еще нескольких глав. Но даже при использовании этих алгоритмов идеальная рациональность обычно недостижима на практике по причине их вычислительной сложности, как это отмечалось в главе 1. Следует также заметить, что не у всех агентов, действующих на основе полезности, поведение определяется на основе модели. В главах 22 и 26 будет показано, что ▶**безмодельный агент** может посредством обучения выяснить, какое действие будет наилучшим в определенной ситуации, даже не определив точно, какие изменения это действие вызывает в окружающей среде.

И наконец, все сказанное выше предполагает, что разработчик агента, безусловно, может корректно определить его функцию полезности. В главах 17, 18 и 22 проблема неизвестных функций полезности рассматривается более подробно.

2.4.6. Обучающиеся агенты

Выше были описаны программы агентов, в которых применялись различные методы выбора действий. Но до сих пор еще не рассматривался вопрос о том, как *создаются* программы агентов. В своей знаменитой ранней статье Тьюринг ([2235], 1950) проанализировал идею о том, как фактически должно осуществляться программирование предложенных им интеллектуальных машин вручную. Он оценил объем работы, который для этого потребуется, и пришел к такому выводу: "Желательно было бы иметь какой-то более продуктивный метод". Предложенный им метод заключался в том, что необходимо создавать обучающиеся машины, а затем проводить их обучение. Сейчас этот метод стал доминирующим при создании наиболее современных систем во многих областях искусственного интеллекта. Любой тип агента (действующий на основании модели, целенаправленный, практичный и т.д.) может быть создан как обучающийся агент (или нет).

Как отмечалось выше, обучение имеет еще одно преимущество: позволяет агенту функционировать в первоначально неизвестной ему среде и становиться более компетентным в сравнении с тем, что обеспечивали лишь его начальные знания. В данном разделе кратко представлены основные сведения об обучающихся агентах. Существующие возможности и методы обучения агентов конкретных типов рассматриваются во многих главах данной книги, а в главах 19–22 более подробно описываются сами алгоритмы обучения.

Как показано на рис. 2.15, в структуре обучающегося агента можно выделить четыре концептуальных компонента. Наиболее важное различие наблюдается между ▶**обучающим компонентом**, отвечающим за внесение улучшений, и ▶**действующим компонентом**, обеспечивающим выбор внешних действий. Действующим компонентом в этом агенте является то, что до сих пор в данной книге рассматривалось в качестве всего агента в целом: он получает данные восприятия и принимает решение о выполняемом действии. Обучающий компонент использует информацию обратной связи от ▶**критика**, оценивающего, как действует агент, и определяет, каким образом должен быть модифицирован действующий компонент, чтобы он успешнее действовал в будущем.

Конструкция обучающего компонента во многом зависит от конструкции действующего компонента. Предпринимая попытку спроектировать агента, который будет обучаться определенным способностям, прежде всего необходимо найти ответ на вопрос "Действующий компонент какого рода потребуется агенту после того, как он будет обучен эффективно выполнять свои функции?" Будет ошибкой поставить во главу угла вопрос "Как можно решить задачу обучения агента выполнению требуемых функций?" Только после того как сам агент будет спроектирован, имеет смысл приступать к разработке обучающих механизмов, позволяющих усовершенствовать любую часть этого агента.

Компонент "Критик" сообщает обучающему компоненту, насколько хорошо действует агент с учетом установленного стандарта производительности. Этот компонент необходим, поскольку сами результаты восприятия не дают никакой

информации о том, успешно ли действует агент. Например, шахматная программа может получить результаты восприятия, указывающие на то, что она поставила мат своему противнику, но ей требуется стандарт производительности, который позволил бы определить, что это — хороший результат; сами данные восприятия ничего об этом не говорят. Важно, чтобы стандарт производительности был постоянным. В принципе, этот стандарт следует рассматривать как полностью внешний по отношению к агенту, поскольку агент не должен иметь возможности его модифицировать так, чтобы он в большей степени соответствовал его собственному поведению.

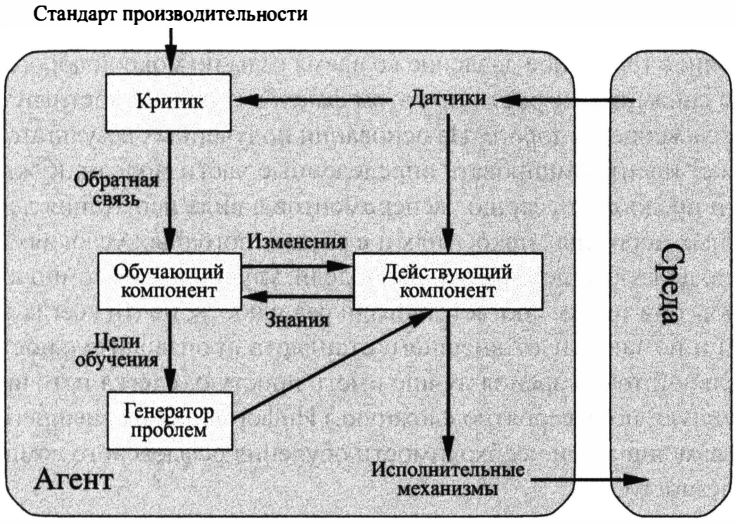


Рис. 2.15. Общая структура обучающегося агента. Прямоугольник "Действующий компонент" представляет то, что выше рассматривалось как программа агента в целом. Прямоугольник "Обучающий компонент" имеет возможность модифицировать ее с целью повышения производительности агента

Последним компонентом обучающегося агента является ▶**генератор проблем**. Его задача заключается в том, чтобы предлагать действия, которые должны привести к получению нового, информативного опыта. Дело в том, что если действующий компонент будет предоставлен самому себе, он будет продолжать выполнять действия, которые являются наилучшими с точки зрения того, что он уже знает. Но если агент готов к тому, чтобы немного поэкспериментировать и в кратковременной перспективе выполнять действия, которые, возможно, окажутся не совсем оптимальными, он получает возможность обнаружить гораздо лучшие действия с точки зрения долговременной перспективы. Генератор проблем предназначен именно для того, чтобы предлагать такие исследовательские действия. Именно этим занимаются ученые, проводя эксперименты. Галилей не считал, что

сбрасывание камней с вершины Пизанской башни является самоцелью. Он не ставил задачи просто разбить эти булыжники вдребезги или вправить ими мозги неудачливым прохожим. Его замысел состоял в том, чтобы изменить взгляды, сложившиеся в его собственной голове, сформулировав лучшую теорию движения объектов.

Обучающий компонент может вносить изменения в любой из компонентов "знаний", присутствующих на схемах структуры агентов (см. рис. 2.9, 2.11, 2.13 и 2.14). В простейших случаях обучение будет осуществляться непосредственно на основании последовательности актов восприятия. Наблюдение за парами последовательных состояний среды может позволить агенту получить информацию о том, "какое влияние оказывают мои действия" и "как изменяется мир" в ответ на эти действия. Например, после того как водитель автоматизированного такси применит определенное тормозное давление во время езды по мокрой дороге, он сможет узнать, какое снижение скорости при этом фактически было достигнуто и наблюдалось ли скольжение по дороге. На основании полученных результатов генератор проблем может идентифицировать определенные части модели, нуждающиеся в улучшении, и предложить серию экспериментов в виде испытания действия тормозов на разных дорожных покрытиях и в разных погодных условиях.

Для агента, действующего на основе модели, улучшение ее компонентов таким образом, чтобы они точнее соответствовали реальности, почти всегда является хорошей идеей и не зависит от внешнего стандарта производительности. (Иногда с вычислительной точки зрения лучше иметь простую, слегка неточную модель, чем совершенную, но невероятно сложную.) Информация из внешнего стандарта потребуется вам лишь при необходимости обучения рефлексного компонента или функции полезности.

Например, предположим, что агент — водитель такси, не получает чаевые о тех пассажиров, которые после поездки чувствовали себя утомленными и полностью разбитыми. Внешний стандарт производительности должен информировать агента, что отсутствие чаевых — это отрицательный вклад в его общую производительность. В этом случае в результате обучения агент получает возможность усвоить, что грубые маневры, раздражающие и утомляющие пассажиров, не позволяют повысить оценку его собственной функции полезности. В этом смысле стандарт производительности позволяет выделить определенную часть входных результатов восприятия как ▶**вознаграждение** (или ▶**штраф**), непосредственно предоставляющее обратную связь в отношении качества поведения агента. Именно с этой точки зрения в жизни животных можно рассматривать такие жестко закрепленные стандарты производительности, как боль или голод.

В более общем смысле *выбор человека* также может предоставлять информацию о его предпочтениях. Например, предположим, что агент — водитель такси не имел сведений о том, что людям обычно не нравятся громкие звуки, и остановился на идее непрерывно нажимать на клаксон, чтобы предупреждать пешеходов о своем приближении. Последующее за этим поведение пассажиров — затыкание

ушей, использование ненормативной лексики и, возможно, обрыв проводов, идущих к клаксону — предоставит агенту возможность успешно обновить свою функцию полезности. Этот вопрос более подробно обсуждается в главе 22.

Подводя итог, отметим, что агенты имеют множество компонентов, а сами эти компоненты могут быть представлены в программе агента многими способами, поэтому создается впечатление, что существует большое разнообразие методов обучения. Однако у всех этих методов есть один объединяющий аспект. Процесс обучения в интеллектуальных агентах в целом можно охарактеризовать как процесс модификации каждого компонента агента с целью более точного соответствия этих компонентов доступной информации обратной связи и тем самым улучшения общей производительности агента.

2.4.7. Как работают компоненты агентских программ

Выше программы агентов были описаны (в терминах самого высокого уровня) как состоящие из различных компонентов, функции которых заключаются в предоставлении ответов на определенные вопросы: "Как сейчас выглядит мир?", "Какое действие я должен сейчас предпринять?", "Что произойдет в результате моих действий?" Следующий вопрос, неизбежно возникающий у читателя, — "А как вообще эти компоненты работают?" Чтобы дать хоть сколько-нибудь правильный ответ на такой вопрос, потребуются тысячи страниц. Поэтому здесь мы хотим лишь обратить внимание читателя на некоторые основные различия в способах, которыми эти компоненты могут представлять окружающую среду, в которой находится агент.

Грубо говоря, можно разместить представления вдоль оси в порядке увеличения сложности и степени выразительности — атомарное, развернутое и структурное. Чтобы проиллюстрировать эту идею, полезно будет рассмотреть определенный компонент агента, например тот, который дает ответ на вопрос "Что произойдет в результате моих действий?" Этот компонент описывает изменения, которые могут произойти в среде в результате выполнения действия. На рис. 2.16 схематически показано, как эти изменения могут быть представлены.

В ▶**атомарном представлении** каждое состояние мира неделимо — оно не имеет внутренней структуры. Рассмотрим задачу о нахождении маршрута поездки из одного конца страны в другой как некоторой последовательности проезжаемых городов (подобная задача представлена на рис. 3.1). Для ее решения может быть вполне достаточно свести состояние мира просто к названию очередного города, в котором находится агент, — единственного атома знаний, "черного ящика", единственным различимым свойством которого является быть идентичным или отличаться от другого черного ящика. Все стандартные алгоритмы, лежащие в основе процедур поиска и игровых процессов (главы 3–5), скрытых марковских моделей (глава 14), а также марковских процессов принятия решений (глава 17), работают с атомарными представлениями.

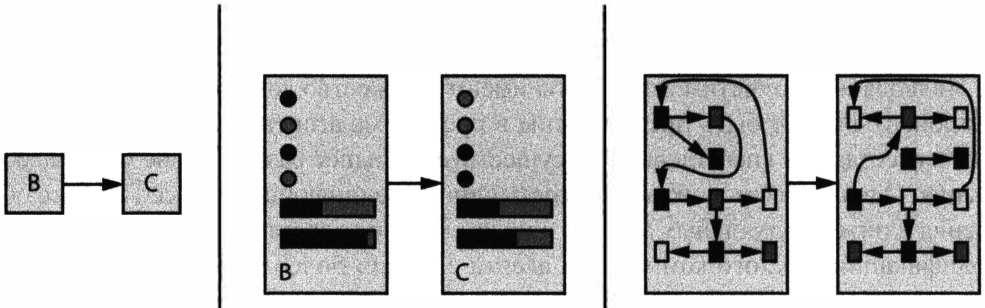


Рис. 2.16. Три способа представления состояний и переходы между ними. **а)** *Атомарное* представление: любое состояние (например, *B* или *C*) является черным ящиком, не имеющим какой-либо внутренней структуры. **б)** *Развернутое* представление: состояние представлено как вектор значений атрибутов, которые могут быть логическими, действительными числами или некоторым символом из фиксированного набора. **в)** *Структурное* представление: состояние включает объекты, каждый из которых может иметь собственные атрибуты, а также связи с другими объектами

В ▶**развернутом представлении** каждое состояние раскладывается в фиксированный набор ▶**переменных** или ▶**атрибутов**, каждый из которых может иметь определенное ▶**значение**. Рассмотрим описание той же задачи определения маршрута, но выполненное уже с большей точностью. Теперь нас интересует не только неделимое местонахождение в том или ином городе, но и количество бензина в баке, текущие GPS-координаты, светится ли индикатор уровня машинного масла, сколько денег потрачено на дорожные сборы, какая радиостанция сейчас в пределах досягаемости и т.д. В то время как два различных атомарных состояния не имеют ничего общего между собой — это просто два разных черных ящика, — два разных развернутых состояния могут иметь одинаковые значения некоторых атрибутов (например, одни и те же GPS-координаты), но значения остальных будут различаться (например, бензина в баке много или нет вовсе). Такое представление намного упрощает выработку решения, как перейти из одного состояния в другое. Многие важные области ИИ предполагают использование развернутого представления состояний, включая алгоритмы удовлетворения ограничений (глава 6), логику высказываний (глава 7), планирование (глава 11), байесовские сети (главы 12–16), а также различные алгоритмы машинного обучения.

Во многих случаях нам необходимо представлять мир как присутствующие в нем *предметы* или *сущности*, имеющие те или иные *отношения* или *связи* между собой, а не только переменные с определенными значениями. Например, во время поездки через всю страну мы можем обнаружить, что прямо перед нами большой грузовик начал поворачивать на подъездную дорогу к молочной ферме,

но ленивая корова блокирует ему путь. Развернутое представление вряд ли будет предусмотрительно включать атрибут *TruckAheadBackingIntoDairyFarmDrivewayBlockedByLooseCow* со значением *true* или *false*. В данном случае нам удобнее будет работать со структурным представлением состояний, в котором такие объекты, как коровы и грузовики со своими атрибутами, а также различными отношениями между ними, могут быть описаны в явном виде (см. рис. 2.16, *в*). Структурные представления лежат в основе реляционных баз данных и логики первого порядка (главы 8–10), вероятностных моделей первого порядка (глава 15) и большей части методов обработки естественного языка (главы 23 и 24). В действительности многое из того, что люди выражают на естественном языке, касается именно объектов и их отношений.

Как уже указывалось выше, ось, вдоль которой мы расположили атомарное, развернутое и структурное представления на рис. 2.16, является осью повышения ▶**выразительности** представления. Грубо говоря, более выразительное представление может включать, по крайней мере в сжатой форме, все то, что включает менее выразительное, плюс еще кое-что. Часто более выразительный язык является *гораздо* более лаконичным. Например, правила игры в шахматы могут быть записаны на одной-двух страницах на языке, использующем структурное представление, таком как логика первого порядка, но потребуются тысячи страниц, если записать их на языке, использующем развернутое представление, таком как логика высказываний, и около 10^{38} страниц, если воспользоваться языком с атомарным представлением, таким как язык конечных автоматов. С другой стороны, с ростом выразительности представлений выполнение рассуждений и процессы обучения становятся более сложными. Для того чтобы получить преимущества от выразительных представлений и одновременно избежать свойственных им недостатков, интеллектуальным системам в реальном мире может потребоваться работать во всех точках вдоль оси выразительности одновременно.

Другая ось для представлений предполагает отображение концепций на определенное место в физической памяти, будь то память компьютера или мозг. Если между понятиями и ячейками памяти имеет место отображение "один-к-одному", мы называем эту ситуацию ▶**локализованным представлением**. С другой стороны, если представление концепции распределено по многим ячейкам памяти и каждая ячейка памяти используется как часть представления *нескольких* различных концепций, такая ситуация называется ▶**распределенным представлением**. Распределенные представления являются более надежными в отношении шума и потери информации. При локализованном представлении отображение концепции на ячейки памяти является произвольным, и если в результате ошибки при передаче информации будет искажено значение всего нескольких битов, понятие *мир* может быть спутано с несвязанным понятием *тир*. Но в случае распределенного представления каждую концепцию можно представить себе как точку в многомерном пространстве, и если несколько битов исказятся в этом случае, то обращение произойдет к другой, достаточно близкой точке в этом пространстве, которое будет иметь достаточно близкое значение.

Резюме

Эту главу можно сравнить с головокружительным полетом над обширным ландшафтом области искусственного интеллекта, которую мы рассматриваем как науку проектирования агентов. Ниже кратко перечислены основные идеи, рассмотренные в данной главе.

- **Агент** — это нечто воспринимающее и действующее в определенной среде. Функция агента определяет действие, предпринимаемое агентом в ответ на любую последовательность его восприятия.
- **Показатель производительности** оценивает поведение агента в среде. Рациональный агент действует так, чтобы максимизировать ожидаемое значение своего показателя производительности, исходя из последовательности восприятия, полученной агентом к данному моменту.
- Спецификация **проблемной среды** включает определение показателей производительности, внешней среды, исполнительных механизмов и датчиков. Первым этапом проектирования агента всегда должно быть определение проблемной среды с наибольшей возможной полнотой.
- Варианты проблемной среды классифицируются по нескольким важным аспектам. Они могут быть полностью или частично наблюдаемыми, детерминированными или недетерминированными, эпизодическими или последовательными, статическими или динамическими, дискретными или непрерывными, а также одноагентными или мультиагентными.
- Если показатель производительности неизвестен или его трудно правильно определить, есть значительный риск, что агент будет оптимизировать процесс достижения ошибочной цели. В таких случаях конструкция агента должна отражать некоторую неопределенность относительно истинной цели.
- **Программа агента** реализует функцию агента. Существует целый ряд основных проектов программ агента, соответствующих характеру явно воспринимаемой информации, которая используется в процессе принятия решения. Разные проекты характеризуются различной эффективностью, компактностью и гибкостью. Выбор наиболее подходящей структуры программы агента зависит от характера среды.
- **Простые рефлекторные агенты** прямо отвечают на полученные восприятия, тогда как **рефлекторные агенты с поведением, основанным на модели**, поддерживают внутреннее состояние, прослеживая те аспекты среды, которые не наблюдаются в текущем акте восприятия. **Агенты, действующие на основе цели**, организуют свои действия так, чтобы достигнуть своих целей, а **агенты, действующие с учетом полезности**, пытаются максимизировать собственную ожидаемую "удовлетворенность".
- Все агенты способны улучшать свои показатели производительности за счет обучения.

Библиографические и исторические заметки

Истоки представлений о центральной роли действий в интеллекте (в виде понятия практических рассуждений) прослеживаются до *Никомаховой этики* Аристотеля. Практические рассуждения были также темой влиятельной статьи Маккарти *Программы со здравым смыслом* ([1529], 1958). Такие области науки, как робототехника и теория управления, по самому своему характеру преимущественно касаются проблем конструирования физических агентов. Понятие ▶ **контроллера** в теории управления идентично понятию агента в искусственном интеллекте. И хотя это может показаться удивительным, но на протяжении большей части истории развития искусственного интеллекта основные усилия в этой области были сосредоточены на исследовании отдельных компонентов агентов — систем поиска ответов на вопросы, программ автоматического доказательства теорем, систем технического зрения и так далее, а не самих агентов, рассматриваемых как единое целое. Важным исключением из этого правила, оказавшим значительное влияние на дальнейшее развитие данной области, стало обсуждение проблематики агентов в работе Генезерета и Нильссона ([836], 1987). В настоящее время подход, основанный на изучении всего агента в целом, уже широко применяется и является центральной темой в новейших учебниках ([1721], 2004; [1147], 2007; [1810], 2017).

В главе 1 корни концепции рациональности прослеживаются в философии и экономике. В искусственном интеллекте это понятие не вызывало значительного интереса до тех пор, пока в середине 1980-х годов не началось широкое обсуждение проблемы создания подходящих теоретических основ данной области. В статье Джона Дойла ([646], 1983) было предсказано, что со временем проектирование рациональных агентов станет рассматриваться в качестве основного назначения искусственного интеллекта, в то время как другие популярные на тот момент темы исследований послужат основой формирования новых дисциплин.

Стремление к тщательному изучению свойств среды и их влияния на выбор самого подходящего проекта рационального агента наиболее ярко проявляется в традиционных областях теории управления; например, в исследованиях по классическим системам управления ([637], 2004 и [1229], 2004) рассматриваются полностью наблюдаемые, детерминированные варианты среды; темой работ по стохастическому оптимальному управлению ([1325], 1986 и [203], 2007) являются частично наблюдаемые, стохастические варианты среды; а работы по гибридному управлению ([1012], 1998 и [379], 2006) касаются таких вариантов среды, которые содержат и дискретные, и непрерывные элементы. Описание различий между полностью и частично наблюдаемыми вариантами среды является также центральной темой работ по динамическому программированию, проводимых в области исследования операций ([1827], 1994), которая будет обсуждаться в главе 17.

Хотя рефлекторные агенты были основной моделью в бихевиористской психологии (см. главу 1), в большинстве работ по искусственному интеллекту их рассматривали как слишком простых для достижения значительных успехов. Однако

в работах Розеншайна ([1916], 1985) и Брукса ([312], 1986) это предположение было поставлено под сомнение (см. главу 26). В начале 2000-х годов значительная часть работ была посвящена поиску эффективных алгоритмов слежения за сложными вариантами среды ([130], 2001; [425], 2005; [2069], 2006), в большинстве случаев в вероятностной обстановке.

Понимание необходимости создания агентов, действующих на основе цели можно проследить во всем, начиная с введенного Аристотелем определения практического рассуждения и заканчивая ранними статьями Маккарти по логическому искусственному интеллекту. Робот Shakey ([738], 1971; [1688], 1984) был первым воплощением логического агента на основе цели, реализованным в виде автоматизированного устройства. Полный логический анализ агентов, действующих на основе цели, приведен в книге Генезерета и Нильссона ([836], 1987), а методология программирования на основе цели, получившая название "агентно-ориентированное программирование", была разработана Шохемом ([2052], 1993). Агентный подход в настоящее время чрезвычайно популярен в разработке программного обеспечения ([440], 2001). Он также проник в область создания операционных систем, где понятие ▶**автономных вычислений** определяет компьютерные системы и сети, которые контролируют сами себя и управляют сами собой, используя циклы "восприятие–действие" и методы машинного обучения ([1217], 2003). Исходя из наблюдения, что совокупность агентских программ, разработанных для совместной работы в истинной мультиагентной среде, обязательно демонстрирует модульность, — программы не имеют общего разделяемого элемента внутреннего состояния и могут общаться друг с другом только через окружающую среду, — сейчас в области ▶**мультиагентных систем** общепринято проектировать программу агента даже отдельного агента как совокупность автономных субагентов. В некоторых случаях можно даже доказать, что полученная система дает те же оптимальные решения, что и вариант с цельной конструкцией.

Кроме того, начиная с книги *Решение проблем человеком* Ньюэлла и Саймона ([1669], 1972), оказавшей чрезвычайно большое влияние на ход дальнейших исследований в области когнитивной психологии, посвященной изучению процедур решения различных задач человеком, ведущее место во взглядах на агентов занял подход, основанный на понятии цели. На этом подходе построены также все последующие работы Ньюэлла ([1665], 1990). Цели, дополнительно подразделяемые на *желания* (общие замыслы) и *намерения* (преследуемые в настоящее время), также являются основой признанной теории агентов, разработанной Майклом Братманом ([294], 1987).

Как отмечалось в главе 1, развитие теории полезности как основы рационального поведения насчитывает уже сотни лет. Однако в ранних исследованиях в области ИИ полезности избегали в пользу целей, за некоторыми исключениями (Фельдман и Спроул [717], 1977). Возрождение интереса к вероятностным методам в 1980-х годах привело к принятию максимизации ожидаемой полезности в качестве наиболее общей основы для принятия решений (Горвиц и др. [1068],

1988). Книга Перла ([1755], 1988) была первой работой в области искусственного интеллекта, в которой был дан глубокий анализ проблем применения теории вероятности и теории полезности, и, вероятно, послужила наиболее важной причиной быстрой смены направления исследований в 1990-х годах (см. главу 16). Формализация обучения с подкреплением в рамках теории принятия решений также внесла свой вклад в этот сдвиг (Саттон [2156], 1988). Довольно примечателен тот факт, что почти во всех исследованиях в области ИИ до самого недавнего времени предполагалось, что показатели производительности могут быть точно и правильно определены в форме функции полезности или функции вознаграждения (Хадфильд–Менелл и др. [942], 2017; Рассел [1942], 2019).

Общая структура обучающегося агента, приведенная на рис. 2.15, является классическим образцом такого проекта в литературе по машинному обучению ([337], 1978 и [1595], 1997). Среди первых примеров использования этой структуры в программах можно упомянуть обучающуюся программу игры в шашки Артура Самюэла ([1969], 1959 и [1970], 1967). Обучающиеся агенты подробно рассматриваются в главах 19–22.

Некоторые ранние работы по агентному подходу представлены в сборниках Ханса и Сингха ([1094], 1998) и Вулдриджа и Рао ([2378], 1999). К учебной литературе по мультиагентным системам, предлагающей отличное введение во многие аспекты проектирования агентов, можно отнести [2305] (2000) и [2377] (2009). К числу конференций, посвященных агентам, относятся International Workshop on Agent Theories, Architectures, and Languages (ATAL), International Conference on Autonomous Agents (AGENTS) и International Conference on Multi-Agent Systems (ICMAS). В 2002 году эти три конференции объединились в одно целое под названием International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS). С 2000 по 2012 год также проводились ежегодные конференции Agent-Oriented Software Engineering (AOSE). В 1998 году был основан журнал *Autonomous Agents and MultiAgent Systems*. И наконец отметим, что в книге *Dung Beetle Ecology* ([962], 1991) приведен большой объем интересной информации о поведении навозных жуков (о котором упоминалось в данной главе). Также на сайте YouTube можно найти несколько впечатляющих видеороликов на эту тему.

Упражнения

2.1. Предположим, что показатель производительности агента имеет отношение только к первым T периодам времени его действия в среде, а все последующее игнорируется. Покажите, что действия рационального агента могут зависеть не только от состояния окружающей среды, но и от достигнутого им периода времени.

2.2. Давайте рассмотрим рациональность различных функций пылесоса.

а) Покажите, что простая функция агента-пылесоса, представленная на рис. 2.3, действительно рациональна при предположениях, перечисленных на этой странице.

б) Опишите функцию рационального агента для случая, когда каждое его перемещение стоит одно очко. Требует ли соответствующая агентская программа внутреннего состояния?

в) Обсудите возможные конструкции агентов для случаев, когда чистые квадраты могут стать грязными, а география окружающей среды неизвестна. Имеет ли в этих случаях смысл агенту учиться на своем опыте? Если да, то чему он должен учиться? Если нет, то почему?

2.3. Напишите эссе о связи между эволюцией и одним или несколькими способностями из числа автономности, интеллектуальности или обучения.

2.4. Для каждого из следующих утверждений укажите, является оно истинным или ложным, и подкрепите свой ответ примерами или контрпримерами, где это будет необходимо.

а) Агент, воспринимающий только частичную информацию о состоянии, не может быть полностью рациональным.

б) Существуют проблемные среды, в которых ни один чисто рефлекторный агент не сможет вести себя рационально.

в) Существует проблемная среда, в которой любой агент является рациональным.

г) Входные данные для программы агента совпадают с входными данными для функции агента.

д) Любая функция агента реализуема с помощью некоторой комбинации программы и машины.

е) Предположим, что агент случайным образом выбирает свое действие из множества возможных действий. Существует детерминированная проблемная среда, в которой этот агент является рациональным.

ж) Определенный агент может быть полностью рациональным в двух разных проблемных средах.

з) Любой агент является рациональным в ненаблюдаемой среде.

и) Полностью рациональный агент, играющий в покер, никогда не проигрывает.

2.5. Для каждого из приведенных ниже действий дайте PEAS-описание проблемной среды и охарактеризуйте ее в терминах свойств, перечисленных в разделе 2.3.1.

а) Игра в футбол.

б) Изучение подземных океанов Титана.

в) Покупка подержанных книг по искусственному интеллекту в Интернете.

г) Теннисный матч.

д) Отработка теннисных ударов у стены.

е) Выполнение прыжка в высоту.

ж) Вязание свитера.

з) Торги на предмет, выставленный на аукцион.

2.6. Для каждого из приведенных ниже действий дайте PEAS-описание проблемной среды и охарактеризуйте ее в терминах свойств, перечисленных в разделе 2.3.1.

а) Выполнение гимнастических упражнений на полу.

б) Изучение подземных океанов Титана.

в) Игра в футбол.

г) Покупка подержанных книг по искусственному интеллекту в Интернете.

д) Отработка теннисных ударов у стены.

е) Выполнение прыжка в высоту.

ж) Торги на предмет, выставленный на аукцион.

2.7. Дайте собственные определения следующих понятий: агент; функция агента; программа агента; рациональность; автономность; рефлекторный агент; агент, основанный на модели; целенаправленный агент; агент, действующий на основе полезности; обучающийся агент.

2.8. В этом упражнении исследуются различия между функциями агентов и программами агентов.

а) Может ли существовать больше чем одна программа агента, которая реализует данную функцию агента? Приведите пример, подтверждающий положительный ответ, или покажите, почему такая ситуация невозможна.

б) Есть ли такие функции агента, которые не могут быть реализованы никакими программами агента?

в) Верно ли, что каждая программа агента реализует точно одну функцию агента, если считать, что архитектура вычислительной машины остается неизменной?

г) Если в архитектуре предусмотрено n битов памяти, то сколько различных возможных программ агента может быть реализовано с ее помощью?

д) Предположим, что мы сохраним программу агента неизменной, но ускорим машину в два раза. Изменит ли это функцию агента?

2.9. Напишите на псевдокоде программы агента для: агента, действующего на основе цели; агента, действующего на основе полезности.

2.10. Рассмотрим простой термостат, который включает подогрев, когда температура по крайней мере на 3 градуса ниже заданного значения, и отключает подогрев, когда температура по крайней мере на 3 градуса выше заданного значения. Является ли этот термостат экземпляром простого рефлекторного агента, рефлекторного агента на основе модели или целенаправленного агента?

Все приведенные ниже упражнения касаются реализации вариантов среды и агентов для мира пылесоса.

2.11. Реализуйте имитатор среды измерения производительности в мире пылесоса, который изображен на рис. 2.2 и определен в разделе 2.2.1. Ваша реализация должна быть модульной, чтобы можно было легко заменять датчики, исполнительные механизмы и модифицировать характеристики окружающей среды (размер, форма, размещение мусора и т.д.). *Примечание.* В интерактивном репозитории кода этой книги уже имеются подобные реализации для некоторых сочетаний языка программирования и операционной системы.

2.12. Реализуйте простого рефлекторного агента для среды пылесоса, которая рассматривается в упражнении 2.11. Вызовите на выполнение имитатор среды с этим агентом для всех возможных начальных конфигураций мусора и местоположений агента. Запишите оценки производительности работы агента для каждой конфигурации и определите его общую среднюю оценку.

2.13. Рассмотрите модифицированную версию среды пылесоса из упражнения 2.11, в которой агенту назначается штраф в один балл за каждое перемещение.

а) Может ли простой рефлекторный агент быть полностью рациональным для этой среды? Обоснуйте свой ответ.

б) А что можно сказать о рефлекторном агенте с внутренним состоянием? Спроектируйте такого агента.

в) Как изменятся приведенные вами ответы на вопросы а и б, если восприятие агента позволяет ему иметь информацию о том, является ли чистым или грязным каждый квадрат в этой среде?

2.14. Рассмотрите модифицированную версию среды пылесоса из упражнения 2.11, в которой география среды (ее протяженность, границы и препятствия) заранее неизвестна, как и начальная конфигурация расположения мусора. (Агент может совершать движения *Left* и *Right*, а также *Up* и *Down*.)

а) Может ли простой рефлекторный агент быть полностью рациональным для этой проблемной среды? Обоснуйте свой ответ.

б) Может ли простой рефлекторный агент с рандомизированной функцией агента превзойти по своей производительности простого рефлекторного агента? Спроектируйте такого агента и измерьте его производительность в нескольких вариантах среды.

в) Можете ли вы спроектировать среду, в которой предложенный вами рандомизированный агент будет показывать очень низкую производительность? Продемонстрируйте полученные вами результаты.

г) Может ли рефлекторный агент с поддержкой внутреннего состояния превзойти по своей производительности простого рефлекторного агента? Спроектируйте такого агента и измерьте его производительность в нескольких вариантах среды. Можете ли вы спроектировать рационального агента этого типа?

2.15. Повторите упражнение 2.14 для такого случая, в котором датчик местоположения заменен датчиком "удара", позволяющим обнаруживать попытки агента пройти сквозь препятствие или выйти за границы проблемной среды. Предположим, что датчик удара перестал работать. Как должен действовать агент?

2.16. Все варианты проблемной среды мира пылесоса, рассматриваемые в предыдущих упражнениях, были детерминированными. Обсудите возможные программы агента для каждого из следующих недетерминированных вариантов.

а) *Закон Мэрфи.* В течение 25% времени применение действия *Suck* не позволяет очистить пол, когда он грязный, и приводит к выбросу мусора на пол, когда он чистый. Как эти обстоятельства отразятся на предложенной вами программе агента, если датчик мусора будет давать неправильный ответ в течение 10% времени?

б) *Маленькие дети.* В каждом интервале времени каждый чистый квадрат имеет 10%-ную вероятность стать грязным. Можете ли вы предложить проект рационального агента для этого случая?

Часть II
Решение задач

Решение задач посредством поиска 125

Поиск в сложных средах 205

Поиск в условиях противодействия и игры 261

Задачи удовлетворения ограничений 319

ГЛАВА 3

Решение задач посредством поиска

В этой главе показано, каким образом агент может найти последовательность действий, позволяющую достичь его целей.

Когда правильное действие, которое следует предпринять, не вполне очевидно, агенту может потребоваться *разработать план наперед*: определить такую *последовательность* действий, которая обеспечит достижение целевого состояния. Такой агент называется ▶**агентом, решающим задачи**, а вычислительный процесс, который он при этом выполняет, называется ▶**поиском**.

Агенты, решающие задачи, используют **атомарные** представления, которые подробно обсуждались в разделе 2.4.7, — в этом случае состояния мира рассматриваются как нечто цельное, не имеющее внутренней структуры, видимой для алгоритма решения задачи. Агенты, использующие **развернутые** или **структурные** представления состояний, называются **агентами планирования** и обсуждаются в главах 7 и 11.

Мы рассмотрим несколько алгоритмов поиска. В этой главе будут рассмотрены только самые простые среды: эпизодические, одноагентные, полностью наблюдаемые, детерминированные, статические, дискретные и известные. Мы проведем различие между **информированными** алгоритмами, когда агент имеет возможность оценить, насколько далеко еще до цели, и **неинформированными** алгоритмами, в которых такая оценка недоступна. В главе 4 столь строгие ограничения в отношении среды будут сняты, а в главе 5 будут рассмотрены среды с несколькими агентами.

В данной главе используются понятия об асимптотической сложности, т.е. о системе обозначений $O(n)$ и NP-полноте из области анализа алгоритмов. Читателям, не знакомым с этими понятиями, следует обратиться к приложению А.

3.1. Агенты, решающие задачи

Представьте себе, что некоторый агент проводит свой отпуск в Румынии в качестве туриста. Он хочет осмотреть достопримечательности, улучшить свой

румынский язык, насладиться ночной жизнью, избежать неприятностей и т.д. Все это уже представляет собой непростую задачу. Далее предположим, что агент в настоящее время находится в городе Арад и что у него есть билет на вылет из Бухареста на следующий день, который нельзя сдать или обменять. Агент видит дорожный указатель, на котором указано, что из Арада ведут три дороги: одна — в сторону Сибиу, одна — в Тимишоару и одна — в Зеринд. Ни один из этих населенных пунктов не является целью агента, поэтому, если он не знаком с географией Румынии, не будет знать, по какому пути следовать.[1]

Если агент не имеет никакой дополнительной информации, т.е. среда является *неизвестной*, то он не сможет сделать ничего лучшего, чем выбрать одно из направлений в случайном порядке. Эта печальная ситуация обсуждается в главе 4. В этой же главе предполагается, что агенты всегда имеют доступ к информации о мире, — в данном случае это будет карта, представленная на рис. 3.1. Располагая этой информацией, наш агент сможет реализовать процесс решения задачи, включающий четыре этапа.

- ▶**Выбор цели**. Агент ставит своей **целью** добраться до Бухареста. Цели организуют поведение, ограничивая условия задачи, и как следствие — действия, которые следует рассматривать.
- ▶**Формулировка задачи**. Агент разрабатывает описание состояний и действий, необходимых для достижения цели, — абстрактную модель соответствующей части мира. Для нашего агента одной из подходящих моделей будет рассмотрение действий переезда от одного населенного пункта к соседнему, а значит, единственным фактом, описывающим изменение состояние мира в результате некоторого действия, будет название достигнутого города.
- ▶**Поиск**. Прежде чем предпринимать какие-либо действия в реальном мире, агент моделирует последовательность действий в своей модели, выполняя поиск решения до тех пор, пока не обнаружит последовательность действий, позволяющую достичь требуемой цели. Такая последовательность действий называется ▶**решением**. Возможно, агенту придется смоделировать несколько последовательностей, не позволяющих достичь цели, но в конце концов он найдет решение (например, переезд из Арада в Сибиу, затем — в Фэгэраш и наконец — в Бухарест) или установит, что решения поставленной задачи не существует.
- ▶**Выполнение**. Теперь агент может последовательно выполнять действия, предусмотренные решением, по одному за раз.

[1] Авторы предполагают, что большинство читателей в подобной ситуации окажутся в таком же положении и вполне смогут представить себя такими же беспомощными, как наш агент. Приносим свои извинения румынским читателям, которые не смогут воспользоваться преимуществами использованного здесь педагогического приема.

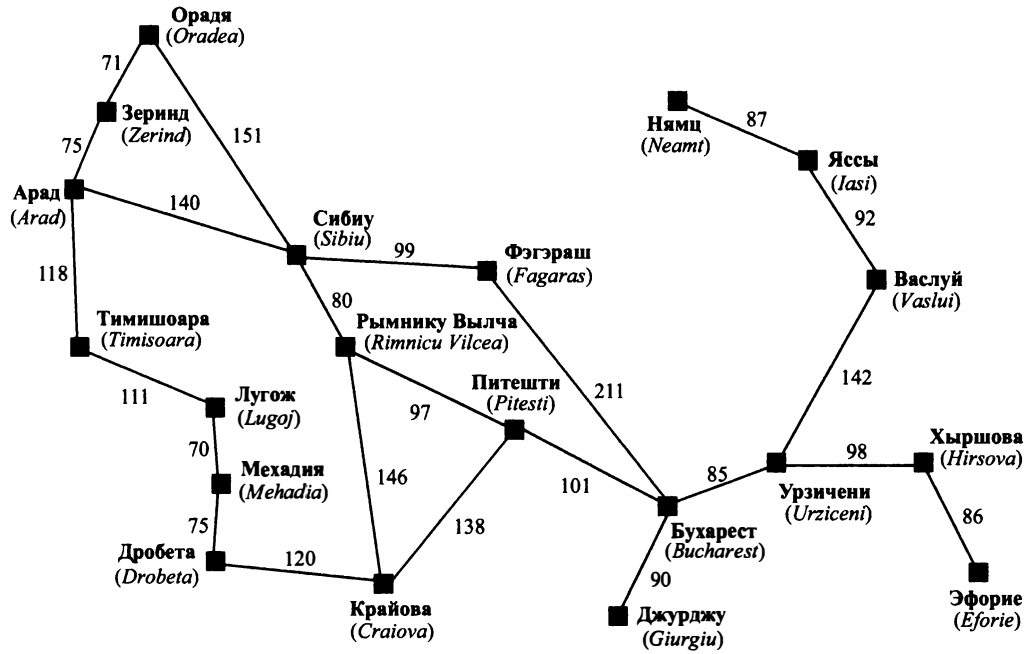


Рис. 3.1. Упрощенная дорожная карта части Румынии с расстояниями в милях

Следует отметить важное свойство: в полностью наблюдаемой, детерминированной, известной среде ➔ *решение любой проблемы представляет собой фиксированную последовательность действий*: переезд в Сибиу, затем — в Фэгэраш, а затем — в Бухарест. Если модель верна, то после того, как агент найдет решение, он сможет игнорировать свое восприятие по ходу выполнения требуемых действий — закрыть глаза, если можно так выразиться, — поскольку найденное решение гарантированно приведет его к цели. В теории управления эту ситуацию называют системой ▶**без обратной связи**: игнорирование восприятия разрывает петлю связи между агентом и средой. Однако если существует вероятность, что модель неверна или среда недетерминирована, то агент будет в большей безопасности, используя подход ▶**с обратной связью**, предполагающий контроль восприятия (см. раздел 4.4).

В частично наблюдаемых или недетерминированных средах решением может быть разветвляющаяся стратегия, предусматривающая в перспективе различные действия в зависимости от тех данных, которые будут получены от датчиков. Например, агент может планировать переезд из Арада в Сибиу, но иметь план и на случай непредвиденных обстоятельств, если случайно окажется в Зеринде или по пути в Сибиу неожиданно обнаружит табличку с надписью "Drum Închis"(дорога закрыта).

3.1.1. Поисковые задачи и решения

Формальное определение поисковой ▶**задачи** включает следующее.

- Множество возможных ▶**состояний**, в которых может находиться среда. Это множество в дальнейшем мы будем называть ▶**пространством состояний**.
- ▶**Начальное состояние**, в котором агент начинает действовать. Например, пребывание в Араде, которое можно обозначить как *Arad*.
- Набор из одного или нескольких ▶**целевых состояний**. Иногда существует одно целевое состояние (например, Бухарест — *Bucharest*), иногда — небольшой набор альтернативных целевых состояний, а иногда цель определяется *свойством*, которое применимо ко многим состояниям (потенциально — к бесконечному числу). Например, в мире пылесоса целью может быть отсутствие мусора в любом местоположении, независимо от каких-либо иных фактов о состоянии среды. Можно учесть все эти три возможности, просто определив для задачи метод IS-GOAL. В этой главе для упрощения речь иногда будет идти об отдельной "цели", но то, что будет сказано, с тем же результатом можно будет применить и к "любому из возможных целевых состояний".
- ▶**Действия**, доступные агенту. При заданном состоянии s функция ACTIONS(s) возвращает конечный[2] набор из действий, которые могут быть выполнены в состоянии s. В этом случае говорят, что каждое из этих действий ▶**применимо** в (состоянии) s, например:

  $$\text{ACTIONS}(Arad) = \{ToSibiu, ToTimisoara, ToZerind\}.$$

- ▶**Функция определения преемника**, описывающая, что именно выполняет каждое действие. Функция RESULT(s, a) возвращает состояние, которое является результатом выполнения действия a в состоянии s, например:

  $$\text{RESULT}(Arad, ToZerind) = Zerind.$$

- ▶**Функция стоимости действия**, — обозначается как ACTION-COST(s, a, s') при программировании или как $c(s, a, s')$ в математических выкладках — возвращает числовое значение стоимости применения действия a в состоянии s для достижения состояния s'. Агент, решающий задачу, должен использовать функцию стоимости, отражающую его собственный показатель производительности. Так, для агентов, осуществляющих поиск маршрута движения, стоимостью действия может быть расстояние между пунктами, выраженное в милях (как показано на рис. 3.1), или это может быть время, необходимое для совершения данного действия в полном объеме.

[2] Для задач с бесконечным количеством действий нам понадобятся методы, выходящие за рамки тематики этой главы.

Последовательность действий образует ▶**путь**, а ▶**решение** — это путь от начального состояния к целевому состоянию. Предполагается, что затраты на действия являются аддитивными, т.е. общая стоимость пути представляет собой сумму стоимостей образующих его действий. ▶**Оптимальное решение** имеет самую низкую стоимость пути среди всех существующих решений. В этой главе предполагается, что стоимости всех действий выражены положительными числами, а это позволяет избежать определенных осложнений.[3]

Пространство состояний можно представить в виде ▶**графа**, в котором вершины являются состояниями, а направленные ребра между ними — действиями. Карта Румынии, представленная на рис. 3.1, является таким графом, на котором каждая дорога представляет два действия, по одному в каждом направлении движения.

3.1.2. Формулировка задачи

Наша формулировка задачи "добраться из Арада в Бухарест" представляет собой **модель** — абстрактное математическое описание, а не что-то реальное. Сравните простое атомарное описание состояния *Arad* с действительным путешествием по стране, в котором состояние мира должно учитывать так много факторов: попутчиков, текущие радиопередачи, вид из окна, наличие поблизости представителей сил правопорядка, расстояние до ближайшей остановки на отдых, состояние дороги, погоду, встречный транспорт и т.д. Все эти соображения исключены из нашей модели, поскольку они не имеют отношения к задаче поиска маршрута в Бухарест.

Процесс удаления деталей из представления называется ▶**абстракцией**. Хорошая формулировка задачи должна иметь правильный уровень детализации. Если эти действия будут определяться на уровне "продвинуть правую ногу вперед на

[3] В любой задаче с циклами, имеющими суммарную отрицательную стоимость, оптимальным по стоимости решением будет бесконечное повторение этого цикла. Алгоритмы Беллмана–Форда и Флойда–Уоршалла (здесь не рассматриваются) корректно работают с действиями, имеющими отрицательную стоимость, но только в случае, если нет отрицательных циклов. Также можно без затруднений работать с действиями с нулевой стоимостью, когда количество последовательно выполняемых действий с нулевой стоимостью ограничено. Например, можно представить робота, для которого установлена определенная стоимость перемещений, а стоимость поворота на 90° считается нулевой. Алгоритмы, обсуждаемые в этой главе, могут справиться с этим, если будет разрешено не более трех последовательных поворотов на 90°. Также существуют сложности с задачами, имеющими бесконечное количество сколь угодно малых затрат на действие. Рассмотрим версию парадокса Зенона, в которой действие продвижения на половину оставшегося до цели пути будет иметь стоимость, равную половине стоимости предыдущего действия. Эта задача не имеет решения при конечном числе действий, но чтобы предотвратить принятие алгоритмом поиска неограниченного количества действий в попытках достижения конечной цели, можно потребовать, чтобы стоимость любого действия была не меньше ϵ — некоторого небольшого положительного значения.

сантиметр" или "повернуть рулевое колесо на один градус влево", агент, вероятно, не сможет даже выехать со стоянки, не говоря уже о том, чтобы добраться до Бухареста.

Можно ли достичь большей точности определения приемлемого ▶**уровня абстракции**? Просто рассматривайте те абстрактные состояния и действия, которые были выбраны, как соответствующие большие множества более детализированных состояний мира и последовательностей действий. Давайте рассмотрим конкретное решение абстрактной задачи, например путь от Арада до Бухареста через города Сибиу, Рымнику-Вылча и Питешти. Это абстрактное решение соответствует большому количеству более детализированных путей. Например, можно вести автомобиль между городами Сибиу и Рымнику-Вылча с включенным радиоприемником, а затем выключить его на всю оставшуюся часть путешествия.

Абстракция является *действительной*, если мы можем преобразовать любое абстрактное решение в такое решение, которое подходит для более детально описанного мира. Достаточным условием является то, что для любого детализированного состояния, которое представляет собой "пребывание в городе Арад", существует детализированный путь в некоторое состояние, соответствующее "пребыванию в городе Сибиу",[4] и т.д. Абстракция будет полезной, если осуществление каждого из действий, предусмотренных в решении, становится проще по сравнению с первоначальной задачей. В нашем случае действие "проехать из Арада в Сибиу" является достаточно простым для того, чтобы оно могло быть выполнено без дальнейшего поиска или планирования водителем средней квалификации. Поэтому выбор хорошей абстракции сводится к исключению максимально возможного количества подробностей при обязательном сохранении ее способности оставаться действительной и гарантирующей легкую осуществимость соответствующих абстрактных действий. Если бы не было возможности создавать полезные абстракции, то интеллектуальные агенты не могли бы успешно действовать в реальном мире.

3.2. Примеры задач

Предложенный выше подход к решению задач был применен в очень многих вариантах экспериментальной среды. В этом разделе перечислены некоторые из наиболее известных примеров решения задач, которые подразделяются на два типа — *стандартизированные* и *реальные* задачи. ▶**Стандартизированная задача** предназначена для иллюстрации или проверки различных методов решения задач. Ей может быть дано краткое, точное описание, а это означает, что такая задача может использоваться разными исследователями для сравнения производительности алгоритмов. ▶**Реальной задачей**, такой как проблема ориентации робота, называется задача, решение которой действительно требуется людям. Формулировка таких задач, как правило, строго индивидуальна и не стандартизирована,

[4] См. раздел 11.4.

например потому, каждый робот имеет свой набор датчиков, которые предоставляют ему уникальный набор данных.

3.2.1. Стандартизированные задачи

Среда задач ▶**клеточного мира** (*grid world*) представляет собой двухмерный прямоугольный массив из квадратных ячеек, в котором агенты могут перемещаться из клетки в клетку. Как правило, агент может перейти в любую соседнюю клетку, если на его пути нет препятствий, — по горизонтали, по вертикали, а в некоторых задачах и по диагонали. Клетки могут содержать объекты, которые агент может подбирать, толкать или воздействовать на них иным образом. Стена или другое непреодолимое препятствие в клетке препятствует проникновению в нее агента. **Мир пылесоса** из раздела 2.1 может быть определен как задача клеточного мира следующим образом.

- **Состояния**. Состояние мира определяет, какие объекты в нем присутствуют и в каких клетках они находятся. Для мира пылесоса объектами являются агент и любой мусор. В самом простом варианте с двумя клетками агент может находиться в любой из них, при этом каждая из клеток может содержать мусор или нет. Исходя из этого можно определить, что в этом мире возможны всего $2 \times 2 \times 2 = 8$ состояний, как показано на рис. 3.2. В общем случае мир пылесоса с n клетками будет иметь $n \times 2^n$ состояний.

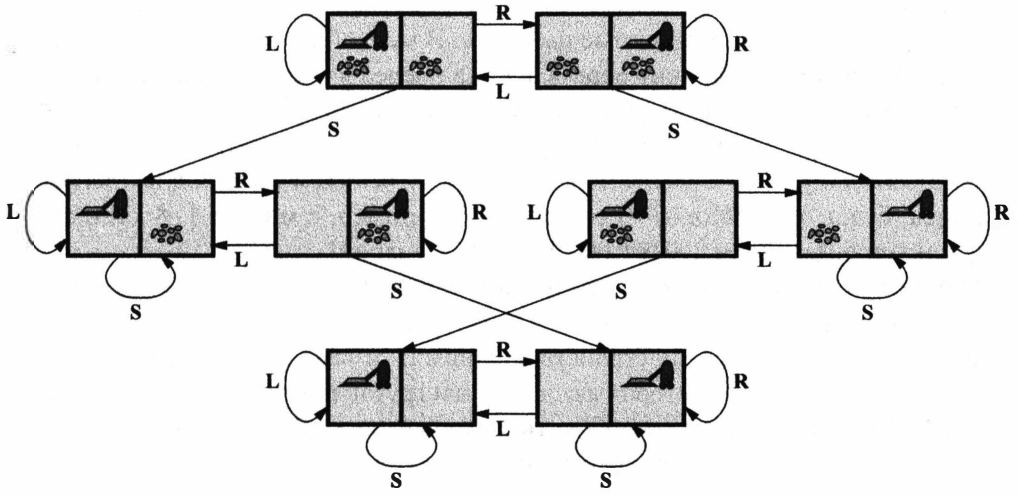


Рис. 3.2. Граф пространства состояний для мира пылесоса с двумя клетками. В нем имеется 8 состояний и по три действия для каждого состояния: L = *Left*, R = *Right*, S = *Suck*

- **Начальное состояние**. В качестве начального состояния может быть назначено любое состояние мира.
- **Действия**. В мире пылесоса с двумя клетками были определены три возможных действия: убрать мусор (*Suck*), переместиться влево (*Left*) и переместиться вправо (*Right*). В произвольном двухмерном многоклеточном мире агенту потребуется больше движений. Можно было бы добавить действия переместиться вверх (*Upward*) и переместиться вниз (*Downward*), что дало бы агенту четыре **абсолютных** действия перемещения, или же можно было бы перейти к эгоцентричным действиям, определяемым относительно направления взгляда агента, например двигаться вперед (*Forward*), назад (*Backward*), повернуть направо (*TurnRight*) и повернуть налево (*TurnLeft*).
- **Функция определения преемника**. Действие *Suck* удаляет мусор в клетке, где находится агент. Действие *Forward* вызывает перемещение агента на одну клетку вперед, т.е. в том направлении, куда он направлен, если только он не упрется в стену — в этом случае действие не имеет никакого эффекта. Действие *Backward* вызывает перемещение агента в противоположном направлении, в то время как действия *TurnRight* и *TurnLeft* изменяют направление на 90° вправо и влево соответственно.
- **Целевое состояние.** Состояние, в котором мусор отсутствует во всех клетках мира.
- **Функция стоимости.** Каждое действие стоит 1 балл.

Другим типом клеточного мира является **головоломка *Сокобан***, в которой цель агента состоит в том, чтобы поместить фишки, изначально произвольно размещенные в клетках мира, на отведенные им места. В каждой клетке может быть не более одной фишки. Когда агент перемещается вперед в клетку с фишкой, сразу за которой находится пустое место, то вперед перемещаются и агент, и фишка.

Агент не может переместить фишку в клетку, где уже находится другая фишка, или же вытолкнуть ее за стену. В таком мире, имеющем n свободных клеток и b фишек, будет $n \times n!/(b!(n-b)!)$ состояний; например, в мире с 8×8 клетками и 12 фишками будет более 200 000 000 000 000 состояний!

В ▶ **головоломках с перемещением плиток** количество плиток в коробке может быть разным, но всегда есть одно или более пустых мест, что позволяет перемещать плитки на пустое пространство. Одним из известных вариантов является головоломка "Час пик", в которой автомобили и грузовики перемещаются по доске размером 6×6 ячеек с целью освободить заданный автомобиль из пробки. Возможно, наиболее известным вариантом такого мира является головоломка ***Восемь*** (рис. 3.3), состоящая из коробки 3×3 квадрата с восемью пронумерованными плитками и одним пустым пространством, а также подобная ей головоломка ***Пятнадцать*** с коробкой 4×4 квадрата и 15-тью плитками. Цель в каждом случае состоит в том, чтобы достичь определенного целевого состояния, например такого, как показано на рис. 3.3, справа. Стандартная формулировка этой задачи приведена ниже.

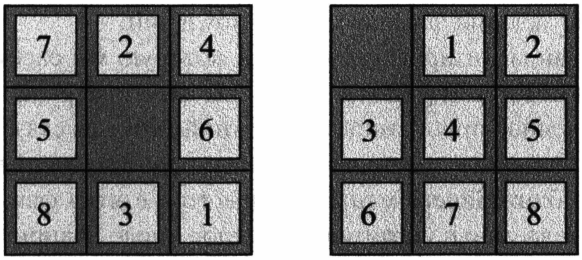


Рис. 3.3. Типичный пример головоломки ***Восемь***

- **Состояния.** Описание состояния определяет местонахождение каждой из восьми плиток и пустого участка на игральной доске из девяти квадратов.
- **Начальное состояние.** В качестве начального может быть определено любое состояние. Обратите внимание, что в этом мире свойство четности делит пространство состояний на две части: любая заданная цель может быть достигнута точно из половины возможных начальных состояний (см. упр. 3.6).
- **Действия.** В то время как в физическом мире головоломка состоит из плиток, перемещаемых в пределах границ головоломки, простейший способ из описания действий в соответствующем клеточном мире — представить, что перемещается пустое поле. Соответствующие этому случаю действия вполне типичны: *Left* (влево), *Right* (вправо), *Up* (вверх) и *Down* (вниз). Если пустое поле окажется на краю сетки или в углу, не все действия будут применимы.
- **Функция определения преемника**. Отображает состояния и действия в результирующие состояния. Например, если выполнить действие *Left* в начальном состоянии мира, представленном на рис. 3.3, пустое место и плитка 5 поменяются местами.
- **Целевое состояние.** Хотя в этом мире любое состояние может быть выбрано как целевое, в данной головоломке ее решение определено как состояние, когда все плитки расположены по возрастанию их номеров, как показано рис. 3.3, справа.
- **Функция стоимости.** Каждое перемещение плитки стоит 1 балл.

Обратите внимание, что каждая формулировка задачи включает в себя некоторые абстракции. Какие абстракции предусмотрены в этом примере? В головоломке ***Восемь*** мы абстрагируемся до уровня указания ее начального и конечного состояний, при этом все промежуточные состояния, достигаемые по ходу перемещения плиток, игнорируются. Из абстрактного описания также исключены такие действия, как встряхивание коробочки, позволяющее передвинуть застрявшую плитку, или извлечение плиток с помощью ножа и повторное укладывание

их в коробку. Исключено также описание правил игры, что позволило обойтись без подробных обсуждений физических манипуляций.

И последняя стандартизированная задача, которую мы рассмотрим, была разработана Дональдом Кнутом ([1244], 1964). Она иллюстрирует, как в задаче может возникнуть бесконечное пространство состояний. Кнут высказал предположение, что начиная с числа 4 последовательность операций извлечения квадратного корня, округления до ближайшего меньшего целого и взятия факториала позволяет получить любое желаемое положительное целое число. Например, получить 5 из 4 можно следующим образом:

$$\left\lfloor \sqrt{\sqrt{\sqrt{\sqrt{\sqrt{(4!)!}}}}} \right\rfloor = 5.$$

Формулировка этой задачи очень проста.

- **Состояния.** Положительные действительные числа.
- **Начальное состояние.** 4.
- **Действия.** Применение математических операций извлечения квадратного корня, округления до ближайшего меньшего целого и взятия факториала (только для целых чисел).
- **Функция определения преемника**. Соответствует математическому определению указанных операций.
- **Целевое состояние**. Заданное положительное целое число.
- **Функция стоимости.** Каждая выполненная операция стоит 1 балл.

Пространство состояний этой задачи является бесконечным: для любого целого числа больше 2 операция взятия факториала дает в результате большее целое число. Эта задача интересна тем, что в ней необходимо манипулировать очень большими числами: кратчайший путь к 5 проходит через (4!)! = 620 448 401 733 239 439 360 000. Бесконечные пространства состояний часто возникают в задачах, связанных с генерацией математических выражений, электронных схем, доказательств, программ и других рекурсивно определяемых объектов.

3.2.2. Реальные задачи

Выше уже было показано, как **задачу поиска маршрута** можно определить в терминах заданных местонахождений и переходов по связям между ними. Алгоритмы поиска маршрута используются в самых разных приложениях. Некоторые из них, такие веб-сайты и автомобильные системы навигации, предоставляющие водителям информацию о направлении движения, являются относительно простыми расширениями нашего примера о поездке в Румынии. (Основные сложности в них связаны с различными издержками, связанными с задержками из-за непредсказуемости трафика, а также изменением маршрутов в связи с закрытием отдельных участков дорог.) К другим приложениям, таким как маршрутизация

видеопотоков в компьютерных сетях, системы планирования военных операций или авиаперевозок, предъявляются намного более сложные требования. Рассмотрим спецификацию задачи создания системы планирования авиапутешествий, которая будет работать на соответствующем веб-сайте.

- **Состояния**. Очевидно, что каждое состояние должно включать местоположение (например, аэропорт) и текущее время. Кроме того, поскольку стоимость действия (перелета) может зависеть от предыдущих этапов полета, их тарифных баз и их статуса (внутреннего — международного), в состоянии должна регистрироваться дополнительная информация об этих "исторических" аспектах.
- **Начальное состояние.** Аэропорт вылета клиента.
- **Действия**. Выполнение любого перелета из текущего местоположения, в любом классе посадочных мест, с вылетом позднее текущего момента времени с учетом дополнительного времени для трансфера в пределах аэропорта, если это требуется.
- **Функция определения преемника.** Возвращает состояние, достигнутое в результате выполнения перелета, в котором местоположением является его аэропорт назначения, а момент прибытия — новой отметкой времени.
- **Целевое состояние.** Место назначения. Иногда цель может быть более сложной, например "прибыть в место назначения беспосадочным рейсом".
- **Функция стоимости**. Комбинация из денежных затрат, времени ожидания, времени полета, таможенных и иммиграционных процедур, категории посадочного места, времени суток, типа самолета, поощрительных баллов и т.д.

В коммерческих системах консультирования по вопросам планирования путешествий используется подобная формулировка задачи, включающая множество дополнительных усложнений, необходимых, например, для успешной обработки запутанных схем тарификации авиакомпаний. Однако любой опытный путешественник знает, что не все авиапутешествия проходят согласно плану. Действительно хорошая система должна предусматривать и планы действий в непредвиденных ситуациях — что предпринять, если вылет задержат и стыковочный рейс будет пропущен?

В ▶**задачах планирования обхода** вместо единственного пункта назначения предполагается посещение заданного множества мест. ▶**Задача коммивояжера** (*traveling salesperson problem — TSP*) представляет собой такую задачу планирования обхода, в которой требуется посетить каждый город, имеющийся на карте. Целью является отыскание маршрута со стоимостью $< C$ (или в качестве задачи оптимизации — поиск пути с минимально возможной стоимостью). На улучшение возможностей TSP-алгоритмов было потрачено огромное количество усилий. Эти алгоритмы также могут быть расширены для управления флотами или автомобильными перевозками. Например, разработка алгоритма поиска и оптимизации маршрутов школьных автобусов в Бостоне позволила получить годовую экономию

в 5 млн долл., снизить интенсивность трафика и уменьшить загрязнение окружающей среды, а также обеспечила экономию времени водителям и учащимся ([204], 2019). Помимо планирования маршрутов транспорта, алгоритмы планирования обхода использовались для решения таких задач, как планирование перемещений головки автоматической дрели при изготовлении печатных плат и определении оптимальных маршрутов доставки материалов к станкам в цехах.

► **Задача компоновки СБИС** требует позиционирования миллионов компонентов и соответствующей разводки соединений на кристалле для минимизации площади, схемных задержек, паразитных емкостей и максимизации выхода готовой продукции. Задача компоновки следует за этапом логического проектирования и обычно подразделяется на две части: **компоновка ячеек** и **разводка соединений**. При компоновке ячеек простейшие компоненты схемы группируются по ячейкам, каждая из которых выполняет некоторую конкретную функцию. Каждая ячейка имеет постоянную форму (размеры и площадь) и требует создания определенного количества соединений с каждой из остальных ячеек. Целью является размещение ячеек на кристалле таким образом, чтобы они не перекрывались и оставалось достаточно места для размещения соединительных проводников между ячейками. При разводке соединений осуществляется поиск конкретного маршрута для каждого проводника через зазоры между ячейками. Эти задачи поиска являются чрезвычайно сложными, но затраты на их решение, безусловно, оправдываются.

► **Задача навигации робота** представляет собой обобщение описанной выше задачи поиска маршрута. Вместо следования по отдельным путям (таким, как дороги в Румынии) робот может двигаться в любом направлении, по сути прокладывая собственный путь. Для робота, движущегося на колесах и перемещающегося по плоской поверхности, окружающее пространство фактически является двухмерным. Если робот имеет верхние и нижние конечности, которыми также необходимо управлять, то пространство поиска становится многомерным — по одному измерению на каждое угловое соединение. Даже для того чтобы сделать это пространство поиска конечным, требуются весьма развитые методы (см. главу 26). Реальные роботы, помимо изначальной сложности задачи, должны справляться с ошибками в показаниях своих датчиков и в управлении двигателями, с частичной наблюдаемостью среды и возможным наличием других роботов, способных изменять состояние среды.

► **Автоматическая конвейерная сборка** сложных объектов (таких, как электрические двигатели) с помощью в роботов стала стандартной практикой в промышленности уже с 1970-х годов. Сначала были найдены алгоритмы выполнимой последовательности сборки, а затем проведена работа по оптимизации процесса. Минимизация количества ручного человеческого труда на конвейере позволяет добиться значительной экономии времени и затрат. В задачах сборки цель состоит в том, чтобы определить порядок сборки деталей какого-либо объекта. Если следовать неверному порядку, может оказаться, что поместить на свое место очередную часть будет невозможно без того, чтобы не снять некоторую уже установленную

ранее часть. Проверка действий в последовательности на выполнимость является задачей сложного геометрического поиска, тесно связанной с задачей навигации робота. В результате генерация допустимой последовательности действий становится самым "дорогостоящим" элементом задачи конвейерной сборки. Любой практически применимый алгоритм должен предотвращать необходимость поиска во всем пространстве состояний, за исключением крошечной его части. Одной из важных задач сборки также является ▶**конструирование белка**, цель которого состоит в том, чтобы найти последовательность аминокислот, которую можно будет сложить в трехмерную структуру белка с правильными свойствами, предназначенного для лечения некоторых заболеваний.

3.3. Алгоритмы поиска

▶**Алгоритм поиска** в качестве входных данных принимает задачу поиска и возвращает ее решение либо сообщает о неудаче. В этой главе мы рассмотрим алгоритмы, в которых на граф пространства состояний накладывается **дерево поиска**, представляющее различные пути из начального состояния, выбранные в поисках пути, ведущего к целевому состоянию. Каждый ▶**узел** в дереве поиска соответствует некоторому состоянию в пространстве состояний, а соединяющие узлы линии представляют действия. Корень дерева поиска соответствует начальному состоянию задачи.

Важно понимать различие между пространством состояний и деревом поиска. Пространство состояний описывает (возможно, бесконечный) набор состояний мира и те действия, которые позволяют переходить из одного состояния в другое. Дерево поиска описывает пути между теми состояниями, которые будут достигнуты в поисках цели. Дерево поиска может иметь несколько путей (проходящих через несколько разных узлов) к любому заданному состоянию, но каждый узел в дереве имеет уникальный путь в обратном направлении к корню (как и у всех деревьев).

На рис. 3.4 показаны первые несколько этапов поиска пути из Арада в Бухарест. Корневой узел дерева поиска соответствует начальному состоянию, *Arad*. Мы можем ▶**развернуть** этот узел, рассмотрев все доступные для его состояния действия (ACTIONS) и воспользовавшись функцией RESULT, чтобы увидеть, куда эти действия приводят. Для каждого из полученных новых состояний ▶**формируется** новый узел (называемый ▶**дочерним узлом** или ▶**преемником** либо **потомком**). Для каждого сформированного дочернего узла узел *Arad* выступает в качестве ▶**родительского узла**.

Теперь нужно определить, какой из сформированных трех дочерних узлов рассмотреть следующим. В этом и заключается суть поиска: сначала проверить один вариант, отложив другие в сторону для последующего анализа, если он потребуется. Предположим, решено сначала развернуть узел Сибиу, — результат показан на рис. 3.4, внизу: у нас есть набор из шести пока еще не развернутых узлов (на рисунке они выделены светло-серым фоном). Совокупность этих узлов называют

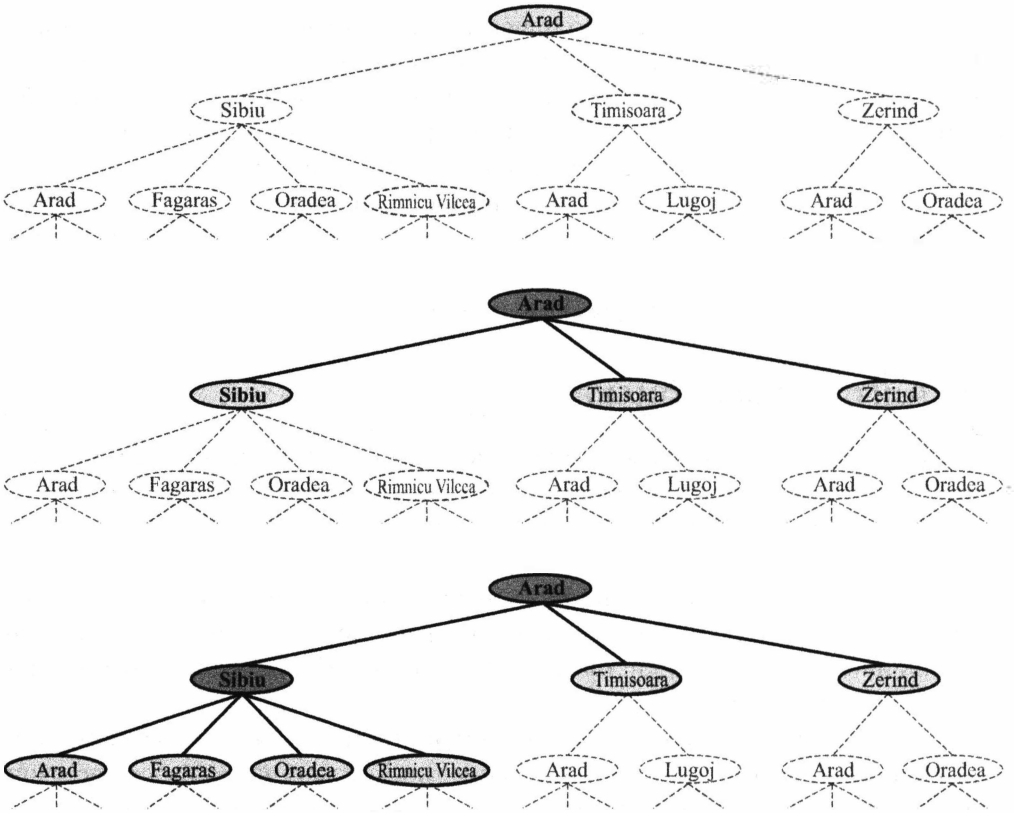


Рис. 3.4. Три частично развернутых дерева поиска, полученных при определении маршрута из Арада в Бухарест. Узлы, которые были *развернуты*, выделены темно-серым цветом и полужирным шрифтом. Периферийные узлы, которые были *сформированы*, но еще не развернуты, выделены светло-серым цветом. Множество состояний, соответствующих этим двум типам узлов, называют *достигнутыми* состояниями. Узлы, которые могут быть сформированы на следующем этапе, обозначены тонкими пунктирными линиями. Обратите внимание, что в нижнем дереве есть цикл: из Арада в Сибиу и обратно в Арад. Понятно, что такой путь не может быть оптимальным, поэтому поиск не должен следовать по этому пути

периферией поискового дерева, а сами эти узлы — ▶**листовыми**, поскольку в дереве у них нет преемников. Также говорят, что любое состояние, для которого уже был сформирован узел в дереве поиска, было ▶**достигнуто** (независимо от того, развернут этот узел или нет).[5] На рис. 3.5 показано дерево поиска, наложенное на граф пространства состояний.

[5] Некоторые авторы называют границу **открытым списком**, что как географически является менее наглядным, так и с вычислительной точки зрения менее целесообразно, поскольку для представления этой структуры эффективнее использовать очередь, а не список. Эти авторы также используют термин **закрытый список** в отношении

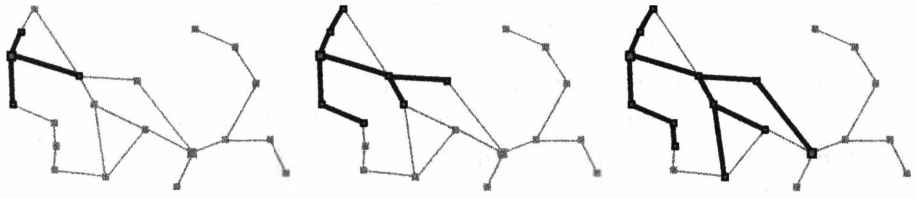
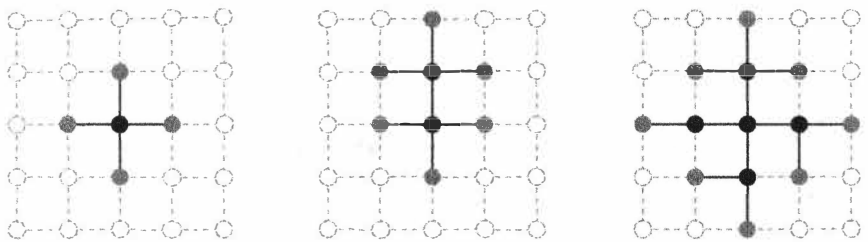


Рис. 3.5. Последовательность поисковых деревьев, развернутых на графе дорог в Румынии, представленных на рис. 3.1. На каждом этапе каждый узел на периферии был развернут по всем путям со всеми применимыми действиями, за исключением тех, которые приводят к состоянию, уже достигнутому ранее. Обратите внимание, что на третьем этапе первый сверху город (Орадя) имеет всего два преемника, оба из которых уже были достигнуты с помощью других путей, так что никакие пути из Оради уже не развертывались

Обратите внимание, что периферия как совокупность листовых узлов ▶ **разделяет** граф пространства состояний на две области: внутреннюю, где каждое состояние уже было развернуто, и внешнюю, включающую все состояния, которые еще не были достигнуты. Это свойство проиллюстрировано на рис. 3.6.



а) б) в)

Рис. 3.6. Свойство разделения пространства состояния удобно демонстрировать на графе задачи клеточного мира. Образующие периферию листовые узлы (светло-серого цвета) отделяют внутреннюю область (узлы темно-серого цвета) от внешней области (пунктирные узлы). Периферия представляет собой множество узлов (и соответствующих состояний), которые уже были достигнуты, но еще не были развернуты. Внутренняя область представляет собой множество узлов (и соответствующих состояний), которые уже были развернуты, а внешняя область — это набор состояний, которые еще не были достигнуты. На рис. *а* развернут только корень дерева, на рис. *б* дополнительно развернут верхний граничный узел, а на рис. *с* все остальные преемники корневого узла были развернуты при их обходе по часовой стрелке

множества ранее развернутых узлов, что в нашей терминологии можно определить как *достигнутые* узлы без *граничных* узлов.

3.3.1. Поиск по первому наилучшему совпадению

Как решить, какой узел периферии развертывать следующим? Самый общий подход называется ▶**поиском по первому наилучшему совпадению**, при котором следующим развертывается узел n с минимальным значением некоторой ▶**функции оценки** $f(n)$. На рис. 3.7 приведен соответствующий алгоритм. Здесь на каждой итерации из периферии выбирается узел с минимальным значением функции $f(n)$, который возвращается как результат, если это состояние является целевым. В противном случае для формирования его дочерних узлов применяется функция EXPAND. Каждый дочерний узел добавляется в периферию, если это состояние не было достигнуто ранее, или еще раз добавляется к ней, если теперь оно достигается по пути, стоимость которого ниже, чем стоимость любого предыдущего пути к этому состоянию. Алгоритм возвращает либо сообщение о неудаче, либо узел, представляющий путь к цели. Используя различные функции $f(n)$, можно получить различные конкретные алгоритмы, которые будут рассмотрены в этой главе.

```
function BEST-FIRST-SEARCH(problem, f) returns искомый узел или failure
  node ← NODE(STATE = problem.INITIAL)
  frontier ← очередь с приоритетом, упорядоченная по f, и элементом node
  reached ← таблица поиска с одной записью с ключом problem.INITIAL
              и значением node
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if s отсутствует в reached or child.PATH-COST < reached[s].PATH-COST then
        reached[s] ← child
        add child to frontier
  return failure

function EXPAND(problem, node) yields узлы
  s ← node.STATE
  for each action in problem.ACTIONS(s) do
    s' ← problem.RESULT(s, action)
    cost ← node.PATH-COST + problem.ACTION-COST(s, action, s')
    yield NODE(STATE = s', PARENT = node, ACTION = action, PATH-COST = cost)
```

Рис. 3.7. Алгоритм поиска по первому наилучшему совпадению и функция развертывания узла. Используемые здесь структуры данных описаны в разделе 3.3.2. Для справок в отношении оператора **yield** обратитесь к приложению Б

3.3.2. Структуры данных при поиске

Для применения алгоритмов поиска необходимы структуры данных, позволяющие сохранять сведения о пути в дереве поиска. Для представления отдельного **узла** дерева мы будем использовать структуру данных *node* с четырьмя компонентами.

- *node*.STATE: состояние в пространстве состояний, которому соответствует данный узел.
- *node*.PARENT: узел в дереве поиска, применявшийся для формирования данного узла (родительский узел).
- *node*.ACTION: действие, которое было применено к родительскому узлу для формирования данного узла.
- *node*.PATH-COST: общая стоимость пути от начального состояния до данного узла. В математических формулах запись $g(node)$ будет использоваться как синоним для PATH-COST.

Следование из узла в обратном направлении по указателям PARENT позволяет восстановить состояния и действия по пути, ведущему к этому узлу. Выполнение этой процедуры из целевого узла представляет собой искомое решение.

Некоторая структура данных, назовем ее *frontier*, необходима и для хранения информации о **периферии** дерева. Наилучшим выбором будет ▶**очередь** того или иного типа, что позволит выполнять с образующими ее листовыми узлами следующие операции.

- IS-EMPTY(*frontier*). Возвращает значение *true* (истина), если в очереди больше нет элементов.
- POP(*frontier*). Возвращает первый элемент в очереди и удаляет его из нее.
- TOP(*frontier*). Возвращает первый элемент в очереди, не удаляя его.
- ADD(*node*, *frontier*). Вставляет узел в надлежащее для него место в очереди.

В алгоритмах поиска используются три типа очередей.

- ▶**Очередь с приоритетом**. В начало очереди помещается элемент с минимальной стоимостью, определяемой в соответствии с некоторой функцией оценки *f*. Такая очередь используется в поиске по первому наилучшему совпадению.
- ▶**Очередь типа FIFO**. Очередь функционирует по приципу "первым вошел — первым вышел", т.е. первым из нее будет извлечен тот элемент, который был помещен в нее первым. Такая очередь используется при поиске в ширину.
- ▶**Очередь типа LIFO**. Очередь функционирует по принципу "первым вошел — последним вышел", т.е. первым из нее будет извлечен элемент, который был помещен в нее последним. (Этот тип очереди также называют ▶**стеком**.) Такая очередь используется при поиске в грубину.

Достигнутые состояния могут быть сохранены в структуре *reached*, представленной в виде таблицы поиска (например, хэш-таблицы), где ключом поиска является состояние, а каждое значение представляет узел для этого состояния.

3.3.3. Избыточные пути

Дерево поиска, показанное на рис. 3.4, внизу, содержит путь из Арада до Сибиу и обратно, из Сибиу в Арад. В этом случае говорят, что *Arad* является ▶**повторяющимся состоянием** в поисковом дереве, образованным в данном случае из-за наличия ▶**цикла** (или **петли**). Поэтому несмотря даже на то, что пространство состояний включает всего 20 состояний, дерево поиска в этом пространстве может включать *бесконечное* количество узлов, поскольку нет никаких ограничений на количество прохождений любого из возможных в нем циклов.

Цикл является особым случаем ▶**избыточного пути**. Например, из Арада в Сибиу можно попасть по прямому пути (140 миль) или добраться по маршруту Арад–Зеринд–Орадя–Сибиу (297 миль). Этот второй путь избыточен, поскольку это просто худший способ перейти в то же состояние, что и в предыдущем случае, поэтому этот путь не следует рассматривать при поиске оптимальных путей.

Пусть агент в клеточном мире размером 10×10 имеет возможность переместиться в любой из 8 соседних квадратов. Если препятствий нет, агент может достичь любой из 100 клеток мира за 9 ходов или меньше. Но общее число путей длиной 9 будет почти 8^9 (чуть меньше из-за наличия краев мира) или более 100 миллионов. Другими словами, в общем случае любая ячейка может быть достигнута более чем миллионом избыточных путей длиной 9, и если исключить из рассмотрения все избыточные пути, поиск можно будет выполнить примерно в миллион раз быстрее. Перефразируя известное выражение, можно сказать: ➔*алгоритмы, которые не помнят прошлого, обречены повторять его.* Существует три подхода к решению этого вопроса.

Во-первых, можно запоминать все уже достигнутые состояния (что и делается в алгоритме поиска по первому наилучшему совпадению), это позволит сразу выявлять все избыточные пути и хранить только наилучший путь к каждому состоянию. Такой способ подходит для пространств состояний с большим количеством избыточных путей и будет предпочтительным выбором, когда таблица достигнутых состояний помещается в памяти.

Во-вторых, можно просто не беспокоиться о "повторении прошлого". Существуют некоторые формулировки задач, в которых наличие двух путей, ведущих к одному и тому же состоянию маловероятно или невозможно. Примером может быть задача сборки, в которой при каждом действии к изделию добавляются детали, причем в определенном порядке, так что можно добавить A, а затем B, но нельзя добавить B, а затем A. Для подобных проблем можно сэкономить место в памяти, если вовсе не отслеживать достигнутые состояния и не проверять избыточные пути. Алгоритм поиска принято называть ▶**поиском в графе**, если проверка

избыточных путей выполняется, и ▶**поиском по дереву**,[6] если она не выполняется. Алгоритм BEST-FIRST-SEARCH на рис. 3.7 — это алгоритм поиска в графе, но если в нем удалить все ссылки на таблицу *reached*, он превратится в алгоритм поиска по дереву, использующий меньше памяти, но проверяющий избыточные пути к одному и тому же состоянию, а значит, работающий медленнее.

В-третьих, можно пойти на компромисс и проверять только на наличие циклов, а не вообще всех избыточных путей. Поскольку каждый узел имеет цепочку указателей на родительские узлы, можно выявлять циклы и без использования дополнительной памяти, просто проверяя цепочку родительских узлов на наличие в ней добавляемого конечного узла. В одних реализациях эта цепочка анализируется полностью, и таким образом устраняются все циклы, тогда как в других реализациях проверяется только несколько первых ссылок (например, родитель, родитель родителя и родитель родителя родителя), причем на проверку затрачивается постоянное количество времени и устраняются все короткие циклы (и используются другие механизмы для работы с длинными циклами).

3.3.4. Измерение производительности решения задачи

Прежде чем приступить к анализу принципов работы различных алгоритмов поиска, будет полезно обсудить те критерии, которые используются при их выборе. Оценить производительность алгоритма можно с помощью следующих четырех показателей.

- ▶**Полнота**. Гарантирует ли алгоритм нахождение решения, если оно имеется, и корректное сообщение об ошибке, если решения не существует?
- ▶**Оптимальность затрат**. Обеспечивает ли алгоритм нахождение решения с минимальной стоимостью пути из числа всех возможных решений?[7]
- ▶**Временная сложность**. За какое время алгоритм находит решение? Этот показатель можно измерять в секундах или более абстрактно — по количеству рассматриваемых состояний и действий.
- ▶**Пространственная сложность**. Какой объем памяти необходим для выполнения поиска?

Чтобы понять смысл показателя полноты алгоритма, рассмотрим задачу поиска единственной цели. Искомая цель может находиться в любом месте пространства состояний, поэтому полный алгоритм должен быть способен систематически проанализировать каждое состояние, достижимое из начального состояния.

---

[6] Мы говорим "поиск по дереву", потому что пространство состояний остается тем же графом независимо от того, как выполняется поиск; мы просто рассматриваем его так, как если бы оно было деревом, только с одним путем от каждого узла до корня.

[7] Одни авторы используют термин "приемлемость" для свойства поиска решения с наименьшими затратами, а другие это свойство определяют как просто "оптимальность", но в этом случае его можно спутать с другими типами оптимальности.

В конечных пространствах достичь этого достаточно просто: если сохранять данные о пройденных путях и отсекать циклы (например, Арад–Сибиу–Арад), в конце концов будут достигнуты все состояния, достижимые из начального.

В бесконечных пространствах состояний требуется больше внимания. Например, алгоритм, многократно применяющий операцию получения факториала в задаче "4" Кнута, будет следовать по бесконечному пути от 4 к 4! к (4!)! и т.д. Точно так же в бесконечном клеточном мире без препятствий повторение перемещения вперед по прямой линии приведет к бесконечному пути из новых состояний. В обоих случаях алгоритм никогда не возвращается в состояние, которого он достиг ранее, но является неполным, поскольку обширные участки пространства состояний так никогда и не будут достигнуты.

Чтобы быть полным, алгоритм поиска должен быть ▶**систематическим** в отношении того способа, которым он исследует бесконечное пространство состояний, что позволит иметь уверенность в том, что в конечном итоге может быть достигнуто любое состояние, достижимое из начального состояния. Например, в бесконечном клеточном мире одним из видов систематического поиска является движение по спирали, охватывающее все клетки, находящиеся на расстоянии s этапов от отправного состояния, прежде чем переместиться к клеткам, находящимся на расстоянии $s+1$ этапов. К сожалению, в бесконечном пространстве состояний не содержащих решения, полный алгоритм будет продолжать поиск вечно; он не сможет закончить работу, потому что не может знать, будет ли следующее состояние искомой целью.

Временная и пространственная сложность всегда анализируются с учетом определенного критерия измерения сложности задачи. В теории компьютерных наук типичным критерием является размер графа пространства состояний, $|V|+|E|$, где $|V|$ — число вершин (узлов состояний) графа, а $|E|$ — количество ребер (отдельные пары "состояние/действие"). Это уместно, когда граф рассматривается как явно заданная структура данных, например карта дорог Румынии. Однако во многих задачах искусственного интеллекта граф представлен лишь *неявно* с помощью начального состояния, действий и функции определения преемника. При неявном задании пространства состояний сложность может быть измерена в терминах d — ▶**глубины** или количества действий в оптимальном решении, m — максимального количества действий в любом пути и b — ▶**коэффициента ветвления** или числа преемников узла, которые необходимо учитывать.

3.4. Стратегии неинформированного поиска

Неинформированный алгоритм поиска не имеет представления о том, насколько близко любое состояние находится к цели (целям). Например, рассмотрим нашего агента в городе Арад, цель которого — добраться до Бухареста. Неинформированный агент без знания географии Румынии не может определить, какой город лучше выбрать на первом этапе — Зеринд или Сибиу. В противоположность этому

информированный агент (см. раздел 3.5), знающий о географическом расположении каждого города, понимает, что Сибиу гораздо ближе к Бухаресту чем Зеринд, и следовательно, с большей вероятностью находится на кратчайшем пути к цели.

3.4.1. Поиск в ширину

Когда все действия имеют одну и ту же стоимость, соответствующая стратегия поиска называется ▶**поиском в ширину**. Это простая стратегия, в которой первым развертывается корневой узел, далее развертываются все преемники корневого узла, затем — все преемники *этих* преемников и т.д. Это систематическая стратегия поиска, и следовательно, она будет полной даже в бесконечных пространствах состояний. Можно реализовать стратегию поиска в ширину посредством вызова функции BEST-FIRST-SEARCH, если используемая функция оценки $f(n)$ будет возвращать глубину узла, т.е. количество действий, необходимое для достижения этого узла.

Эффективность этой стратегии можно повысить с помощью нескольких приемов. Во-первых, последовательная очередь типа FIFO (*первым пришел — первым вышел*) работает быстрее приоритетной и при этом автоматически обеспечивает правильный порядок развертывания узлов: новые узлы (которые всегда находятся уровнем ниже, чем их родители) помещаются в конец очереди, а ранее добавленные в нее узлы, расположенные на более высоких уровнях, находятся в начале и будут развернуты первыми. Во-вторых, множество *reached* теперь может представлять собой множество состояний, а не отображений состояний на узлы, поскольку, если состояние было достигнуто, мы можем уже не найти к нему лучшего пути. А это также означает, что появляется возможность выполнять ▶**раннюю проверку достижения цели**, проверяя, является ли узел искомым решением, непосредственно при его *формировании*, отказавшись от ▶**поздней проверки достижения цели**, используемой в алгоритме поиска по первому наилучшему совпадению и заключающейся в проверке узла только тогда, когда он извлекается из очереди. На рис. 3.8 показан ход поиска в ширину в двоичном дереве, а на рис. 3.9 представлены соответствующие алгоритмы, построенные с учетом упомянутых выше возможностей повышения эффективности его работы.

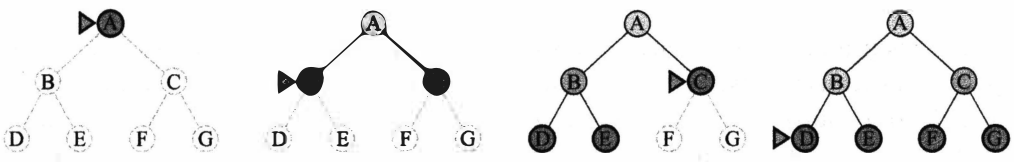


Рис. 3.8. Поиск в ширину в простом двоичном дереве. На каждом этапе узел, который будет развернут следующим, отмечен треугольным маркером

Поиск в ширину всегда находит решение с минимальным количеством действий, поскольку, когда формируется узел на глубине d, все узлы на глубине $d-1$ уже сформированы, а значит, если бы один из них был решением, он был бы уже найден. Это означает, что найденное решение будет оптимальным по затратам для тех задач, в которых все действия имеют одинаковую стоимость, но не для задач, в которых такого свойства нет. Поиск будет полным в любом случае. В отношении оценки затрат времени и пространства представим, что поисковое дерево унифицировано и каждое состояние имеет b преемников. При поиске из корневого узла формируется b узлов, при развертывании каждого из которых формируется еще по b узлов, т.е. всего b^2 узлов на втором уровне. При развертывании каждого из них формируется еще по b узлов, в результате чего на третьем уровне будет b^3 узлов, и т.д. Теперь предположим, что решение находится на глубине d. Тогда общее количество сформированных узлов составит

$$1 + b + b^2 + b^3 + \ldots + b^d = O(b^d).$$

```
function BREADTH-FIRST-SEARCH(problem) returns искомый узел или failure
  node ← NODE(problem.INITIAL)
  if problem.IS-GOAL(node.STATE) then return node
  frontier ← очередь типа FIFO с элементом node
  reached ← {problem.INITIAL}
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if problem.IS-GOAL(s) then return child
      if s отсутствует в reached then
        добавить s в reached
        добавить child в frontier
  return failure

function UNIFORM-COST-SEARCH(problem) returns искомый узел или failure
  return BEST-FIRST-SEARCH(problem, PATH-COST)
```

Рис. 3.9. Алгоритм поиска в ширину и функция поиска по критерию стоимости

Все сформированные узлы остаются в памяти, поэтому как временная, так и пространственная сложность этого алгоритма оценивается как $O(b^d)$. Экспоненциальный рост сложности пугает. В качестве типичного примера из реального мира рассмотрим, например, задачу с фактором ветвления $b = 10$, скоростью обработки 1 млн узлов/с и требованиями к памяти на уровне 1 Кбайт/узел. Поиск с глубиной $d = 10$ потребует чуть меньше 3 ч машинного времени, но также около 10 Тбайт памяти. ➡*Для поиска в ширину требования к памяти представляют существенно большую проблему, чем время выполнения*. Но время, тем не менее, также является

важным фактором. При глубине $d = 14$, даже при наличии бесконечно большой памяти поиск занял бы 3,5 года. В общем случае ➡ *задачи поиска с экспоненциальной сложностью невозможно решить с использованием неинформированного поиска для всех случаев, кроме самых небольших экземпляров.*

3.4.2. Алгоритм Дейкстры или поиск по критерию стоимости

Когда действия имеют разные стоимости, очевидным выбором является использование поиска по первому наилучшему совпадению, где функция оценки — это стоимость пути от корневого узла до текущего узла. Этот алгоритм называется алгоритмом Дейкстры в сообществе теоретиков компьютерных наук и ▶ **поиском по критерию стоимости** — в сообществе специалистов ИИ. Идея состоит в том, что если поиск в ширину распространяется в пространстве состояний волнами одинаковой глубины (сначала — на глубину 1, затем — на глубину 2 и т.д.), то поиск по критерию стоимости распространяется волнами по критерию стоимости пути поиска. Алгоритм может быть реализован как вызов функции BEST-FIRST-SEARCH с использованием функции PATH-COST в качестве функции оценки, как показано на рис. 3.9.

Рассмотрим рис. 3.10, на котором представлена часть схемы дорог Румынии для участка от Сибиу до Бухареста. Преемниками узла Сибиу являются Рымнику Вылча и Фэгэраш, стоимость пути до которых составляет 80 и 99 миль соответственно. Следующим развертывается узел с меньшими затратами, Рымнику Вылча, добавляя в маршрут узел Питешти с общей стоимостью пути $80 + 97 = 177$. Теперь наименьшей стоимостью пути обладает узел Фэгэраш (99), поэтому он будет развернут следующим, добавляя в маршрут Бухарест с общей стоимостью пути $99 + 211 = 310$.

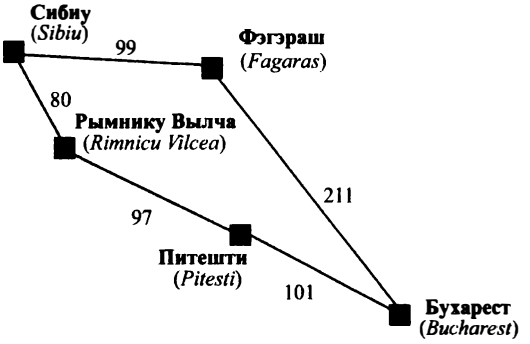


Рис. 3.10. Часть пространства состояний Румынии, выбранная для иллюстрации поиска по критерию стоимости

Бухарест является целью поиска, но в этом алгоритме проверка на достижение цели проводится только тогда, когда узел развертывается, а не когда формируется как узел, так что тот факт, что путь к конечной цели найден, пока еще алгоритмом не обнаружен.

Алгоритм продолжает работать, выбирая следующим для развертывания узел Питешти, и добавляет в дерево второй маршрут к Бухаресту с ценой пути $80 + 97 + 101 = 278$. Этот путь имеет меньшую стоимость, и поэтому данный узел заменяет предыдущий в таблице *reached* (достигнутые) и добавляется в очередь *frontier* (периферия). Далее выясняется, что теперь этот узел имеет наименьшую стоимость и, следовательно, выбирается для развертывания. Вот теперь, наконец, определяется, что он является целевым узлом и функция возвращает его в качестве результата своей работы. Обратите внимание, что если бы проверка достижения цели была проведена при создании узла, а не его развертывании как узла с наименьшей стоимостью пути, то функция вернула бы в качестве результата путь с более высокой стоимостью, — через Фэгэраш.

Сложность поиска по критерию стоимости характеризуется в терминах C^*, т.е. стоимости оптимального решения,[8] и ϵ, нижней границы стоимости каждого действия, при $\epsilon > 0$. Тогда временная и пространственная сложность алгоритма в самом худшем случае может быть оценена как $O(b^{1+\lfloor C^*/\epsilon \rfloor})$, что может быть больше, чем b^d. Так происходит потому, что поиск по критерию стоимости может исследовать обширные деревья из действий с низкой стоимостью, прежде чем доберется до анализа путей, включающих пути с высокой стоимостью и, возможно, более полезные действия. Когда стоимость всех действий одинакова, $b^{1+\lfloor C^*/\epsilon \rfloor}$ превращается в b^{d+1} и поиск по критерию стоимости работает как поиск в ширину.

Поиск по критерию стоимости является полным и оптимальным по затратам, потому что первое найденное им решение будет иметь стоимость, которая по меньшей мере так же мала, как и стоимость любого другого узла в периферии. В поиске по критерию стоимости все пути рассматриваются систематически в порядке возрастания их стоимости и исключается попадание в бесконечные циклы (при условии, что стоимость любого действия будет $> \epsilon > 0$).

3.4.3. Поиск в глубину и проблема памяти

При ▶**поиске в глубину** всегда развертывается *самый глубокий* узел в текущей периферии дерева поиска. Такой поиск можно реализовать как вызов функции BEST-FIRST-SEARCH с функцией оценки f, возвращающей глубину узла со знаком "минус". Однако этот тип поиска обычно реализуется не как поиск в графе, а как поиск по дереву без сохранения таблицы достигнутых состояний. Ход такого поиска показан на рис. 3.11. Поиск сразу же начинает уходить в глубину, пока не

[8] Здесь и далее во всей книге "звездочка" в обозначении C^* означает, что это оптимальное значение C.

достигнет самого глубокого уровня дерева поиска, на котором узлы не имеют преемников. По мере того как узлы на этом уровне развертываются, они удаляются из периферии и после их исчерпания поиск "возобновляется" со следующего самого глубокого узла, который все еще имеет неисследованных преемников. Поиск в глубину не оптимален по затратам и возвращает первое же решение, которое будет найдено, даже если оно не является оптимальным по затратам.

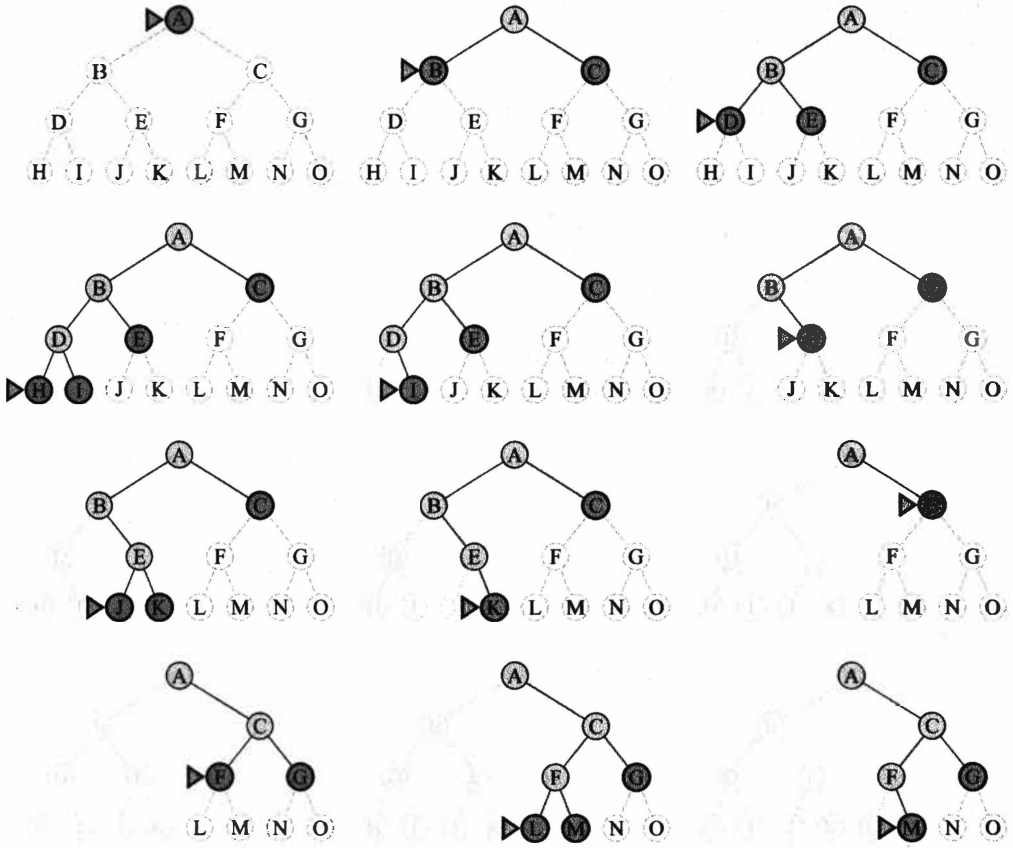


Рис. 3.11. Двенадцать первых этапов (слева направо, сверху вниз) работы алгоритма поиска в глубину в двоичном дереве от начального состояния **A** до целевого состояния **M**. Периферия дерева выделена темно-серым цветом, а треугольный маркер указывает на узел, который будет развернут следующим. Уже развернутые узлы имеют светло-серую заливку, а те, которые еще могут быть достигнуты в будущем, представлены тонким пунктиром. Уже развернутые узлы, не имеющие преемников в периферии и уже отброшенные, представлены бледными размытыми очертаниями

Для конечных пространств состояний, которые являются деревьями, этот тип поиска является эффективным и полным; для пространств состояний без циклов

он может закончиться многократным развертыванием того же самого состояния много раз через разные пути, но (в конечном счете) систематически исследует все пространство.

В пространствах состояний с циклами этот алгоритм может попадать в бесконечный цикл, поэтому некоторые реализации поиска в глубину включают проверку каждого нового узла на наличие циклов. И наконец, в бесконечных пространствах состояний алгоритм поиска в глубину не является систематическим: он может "застрять" в непрерывном движении вниз по бесконечному пути, даже если в самом пространстве состояний циклических переходов нет. Следовательно, алгоритм поиска в глубину является неполным.

Почему, учитывая все упомянутые выше недостатки, кто-то все же может решить применить именно поиск в глубину, а не в ширину или по первому наилучшему совпадению? Ответ прост: для тех задач, в которых возможно построение дерева поиска, метод поиска в глубину имеет намного меньшие потребности в памяти. Нет необходимости хранить полную таблицу всех *достигнутых* узлов, а количество узлов в периферии относительно невелико: если совокупность периферийных узлов при поиске в ширину можно представить себе как поверхность непрерывно расширяющейся сферы, образованной всеми достигнутыми узлами, то при поиске в глубину периферия будет представлять собой всего лишь радиус этой сферы.

Для конечного древовидного пространства состояний, подобного представленному на рис. 3.11, затраты времени при использовании алгоритма поиска в глубину пропорциональны количеству состояний, а потребность в памяти оценивается как $O(bm)$, где b — это фактор ветвления, а m — максимальная глубина дерева. Некоторые задачи, для решения которых по алгоритму поиска в ширину потребуются экзабайты памяти, при применении поиска в глубину можно решить, используя лишь килобайты памяти. Именно по причине его скромных потребностей в памяти поиск в глубину широко использовался в качестве основной рабочей лошадки во многих областях ИИ, в том числе в областях удовлетворения ограничений (глава 6), проблемы выполнимости булевых формул (глава 7) и в логическом программировании (глава 9).

Существует вариант поиска в глубину, называемый ▶ **поиском с возвратами**, в котором используется еще меньше памяти (подробнее о нем — в главе 6). При поиске с возвратами каждый раз формируется только один преемник, а не все преемники, и в каждом частично развернутом узле запоминается информация о том, какой преемник должен быть сформирован следующим. В дополнение при формировании преемника *модифицируется* само уже существующее описание текущего состояния, без выделения памяти для размещения нового. При этом потребность в памяти сокращается до объема, необходимого для хранения только одного описания состояния и $O(m)$ действий, — существенное сокращение в сравнении с $O(bm)$ состояний для классического поиска в глубину. При поиске с возвратами также появляется возможность использовать более эффективную структуру набора данных

для состояний в текущем пути, что позволяет проверять наличие циклических путей за время $O(1)$ вместо $O(m)$. Однако для успешного применения этого типа поиска нужно иметь возможность *отменять* каждую модификацию при возврате, выполняемом для формирования следующего преемника. Использование метода поиска с возвратами может оказаться важнейшим фактором достижения успеха при решении задач с объемными описаниями состояния, таких как роботизированная сборка.

3.4.4. Поиск с ограничением глубины и поиск в глубину с итеративным углублением

Чтобы предотвратить зацикливание алгоритма поиска в глубину на бесконечном спуске, можно прибегнуть к ▶**поиску с ограничением глубины** — варианту классического алгоритма, в котором установлен лимит максимальной допустимой глубины ℓ, т.е. все узлы на глубине ℓ рассматриваются как не имеющие преемников (рис. 3.12). Временная сложность этого алгоритма — $O(b^{\ell})$, а пространственная сложность — $O(b\ell)$. К сожалению, если в любом конкретном случае неудачно выбрать значение ℓ, алгоритм не сможет достичь искомого решения, что, опять-таки, делает его неполным.

Поскольку поиск в глубину является древовидным поиском, в общем случае мы не можем не позволить ему тратить время на проверку избыточных путей, но можем исключить циклы за счет дополнительных расходов времени на вычисления. Если в родительской цепи проверять на цикличность лишь несколько ссылок вверх, можно выявить большинство имеющихся циклов, а более длинные циклы будут устранены за счет задания предельной глубины.

Иногда хороший лимит глубины можно выбрать, основываясь на знании проблемы. Например, на карте Румынии (см. рис. 3.1) имеется 20 городов, поэтому $\ell = 19$ является максимальным допустимым лимитом. Но если внимательно изучить эту карту, то можно обнаружить, что любой город может быть достигнут из любого другого города в пределах 9 этапов. Это число, известное как ▶**диаметр** графа пространства состояний, дает лучший лимит глубины, что ведет к повышению эффективности поиска с ограничением глубины. Однако для большинства задач мы не сможем установить хороший лимит глубины, пока она не будет решена.

▶**Поиск в глубину с итеративным углублением** решает проблему выбора хорошего значения для лимита глубины ℓ посредством проверки всех его значений в порядке увеличения (сначала — 0, затем —1, потом — 2 и т.д.) — пока либо не будет найдено решение, либо функция поиска с ограничением глубины не вернет значение *failure* вместо *cutoff*. Данный алгоритм представлен на рис. 3.12. Итеративный поиск в глубину сочетает в себе многие преимущества и поиска в глубину, и поиска в ширину. Как и поиск в глубину, он предъявляет скромные требования к памяти: $O(bd)$, если решение существует, и $O(bm)$ — для границы пространства состояний, если решение отсутствует. Как и в случае поиска в ширину, итеративный

поиск в глубину является оптимальным для задач, у которых все действия имеют одну и ту же цену, и полным для конечных пространств состояний без циклов или любых конечных пространств состояний в случае проверки на наличие цикла всех узлов, находящихся на пути вверх.

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns искомый узел или failure
  for depth = 0 to ∞ do
    result ← DEPTH-LIMITED-SEARCH(problem, depth)
    if result ≠ cutoff then return result

function DEPTH-LIMITED-SEARCH(problem, ℓ) returns узел или failure или cutoff
  frontier ← очередь типа LIFO (стек) с элементом NODE(problem.INITIAL)
  result ← failure
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    if DEPTH(node) > ℓ then
      result ← cutoff
    else if not IS-CYCLE(node) do
      for each child in EXPAND(problem, node) do
        добавить child в frontier
  return result
```

Рис. 3.12. Поиск по дереву в глубину с итеративным углублением и с ограничением глубины. Функция итеративного углубления многократно обращается к функции поиска с ограничением глубины с возрастающим лимитом глубины. Она возвращает значение одного из трех возможных типов: либо искомый узел, либо значение *failure*, когда проверены все узлы и доказано, что среди них нет искомого на любой глубине, либо значение *cutoff*, указывающее, что искомый узел может находиться на глубине, большей, чем ℓ. Данный алгоритм поиска по дереву не требует хранения всех *достигнутых* состояний, и следовательно, использует гораздо меньше памяти, чем алгоритм поиска по первому наилучшему совпадению, но допускает возможность многократного посещения одного и того же узла по разным путям. Кроме того, если проверке IS-CYCLE будут подвергнуты не все циклы, этот алгоритм может попасть в бесконечный цикл.

Временная сложность алгоритма составляет $O(b^d)$ при наличии решения и $O(b^m)$ — при его отсутствии. На каждой итерации алгоритма поиска с итеративным углублением генерируется новый уровень узлов, — точно так, как при поиске в ширину, но при поиске в ширину это делается с последующим сохранением всех сформированных узлов в памяти, тогда как при итеративном поиске каждый уровень повторно использует один и тот же участок памяти, обеспечивая тем самым ее экономию за счет дополнительных затрат времени. На рис. 3.13 показаны первые четыре итерации работы алгоритма поиска с итеративным углублением на дереве двоичного поиска, в котором решение находится на четвертой итерации.

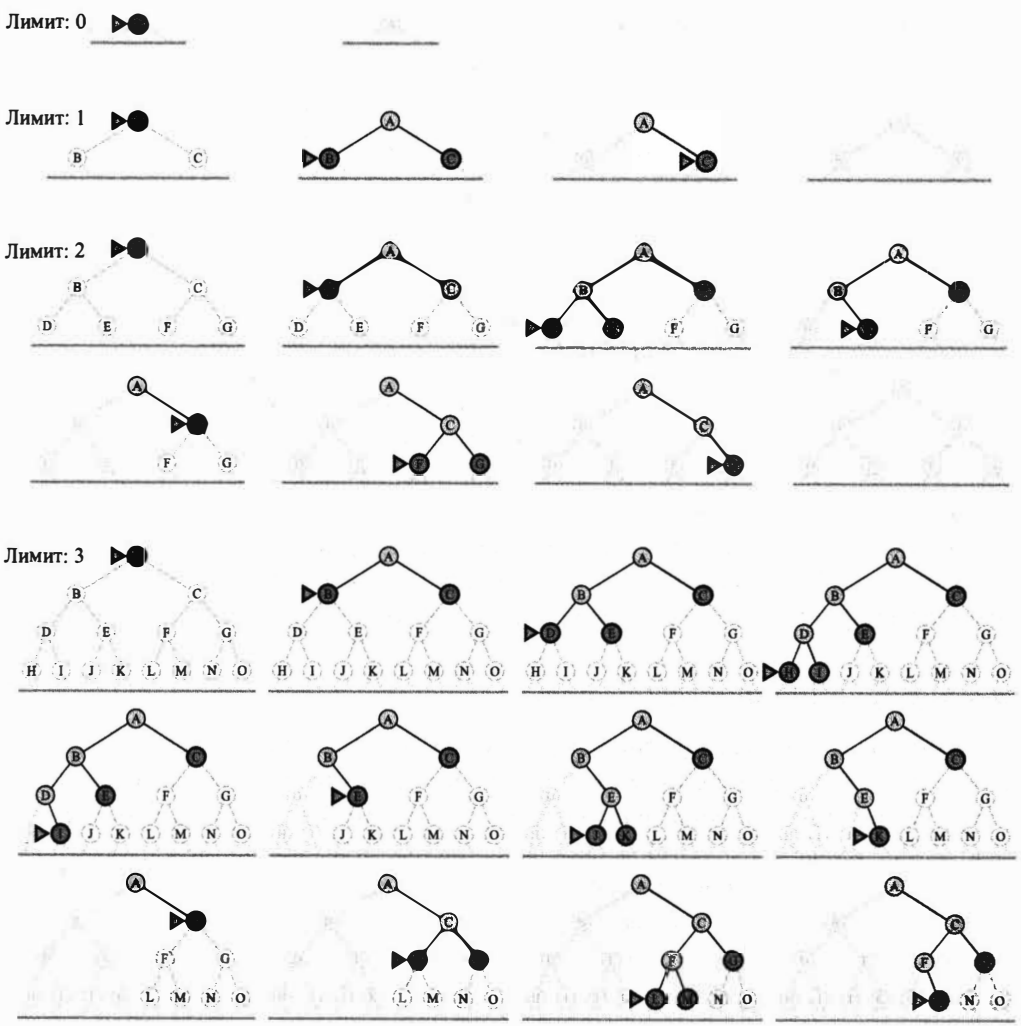


Рис. 3.13. Четыре итерации поиска в глубину с итеративным углублением целевого узла M в двоичном дереве с изменением лимита глубины ℓ от 0 до 3. Обратите внимание, что внутренние узлы образуют единственный путь. На рисунке треугольным маркером отмечается узел, который будет развернут следующим; узлы с темно-серой заливкой образуют периферию; а светло-серым показаны узлы, которые уже отброшены и не могут быть частью решения при заданном лимите глубины

Поиск с итеративным углублением может показаться расточительным, поскольку все узлы в верхней части дерева поиска повторно формируются на каждой итерации. Но во многих пространствах состояний большинство узлов находится на нижнем уровне, поэтому не имеет большого значения, что верхние уровни формируются повторно. При поиске в глубину с итеративным углублением узлы на

нижнем уровне (глубина d) формируются один раз, узлы, которые находятся на предыдущем по отношению к нижнему уровне, формируются два раза, и так далее вплоть до дочерних узлов корневого узла, которые формируются d раз. Следовательно, общее количество формируемых узлов в самом худшем случае составляет

$$N_{\text{IDS}} = (d)b^1 + (d-1)b^2 + (d-2)b^3 \ldots + b^d,$$

что дает временную сложность $O(b^d)$ — асимптотически такую же, как и при поиске в ширину. Например, если $b = 10$ и $d = 5$, то соответствующие значения наибольшего количества узлов будут равны

$$N_{\text{IDS}} = 50 + 400 + 3000 + 20\,000 + 100\,000 = 123\,450,$$
$$N_{\text{BFS}} = 10 + 100 + 1000 + 10\,000 + 100\,000 = 111\,110.$$

Если повторное формирование узлов действительно вызывает беспокойство, можно использовать гибридный подход: поиск в ширину выполняется вплоть до момента, когда будет использована вся доступная память, после чего для всех узлов, находящихся в этот момент в периферии, выполняется поиск в глубину с итеративным углублением. ➨ *В общем случае поиск в глубину с итеративным углублением — это предпочтительный метод неинформированного поиска при условиях, когда пространство поиска велико, а глубина решения неизвестна.*

3.4.5. Двунаправленный поиск

Алгоритмы, рассматривавшиеся до этого момента, начинают свою работу из начального состояния и способны достичь любого из нескольких возможных целевых состояний. Альтернативный подход, называемый ▶**двунаправленным поиском**, предполагает одновременное выполнение поиска в прямом направлении из начального состояния и в обратном направлении от целевого состояния (или состояний) в надежде, что два этих поиска встретятся. Идея состоит в том, что $b^{d/2} + b^{d/2}$ гораздо меньше, чем b^d (например, в 50 000 раз меньше, если $b = d = 10$).

Для реализации этой идеи необходимо отслеживать две периферии и хранить две таблицы достигнутых состояний, а также иметь возможность рассуждать в обратном направлении: если состояние s' является преемником состояния s в прямом направлении, то теперь нам необходимо знать, что состояние s является преемником состояния s' в обратном направлении. Решение будет найдено, когда две периферии столкнутся.[9]

[9] В нашей реализации структура данных таблицы *reached* поддерживает запрос о том, входит ли в нее некоторое заданное состояние, а структура данных *frontier* (очередь с приоритетом) не поддерживает. Поэтому для проверки наличия столкновения у нас используются структуры *reached* — достигнутые узлы, хотя концептуально речь идет о столкновении границ, т.е. периферии каждого поиска. Предлагаемая реализация может быть расширена для обработки нескольких целевых состояний путем формирования узлов для каждого целевого состояния в периферии и таблице достигнутых узлов обратного поиска.

Существует много различных версий двунаправленного поиска — просто потому, что имеется много различных алгоритмов однонаправленного поиска. В этом разделе рассматривается двунаправленный поиск по первому наилучшему совпадению. Хотя в этом случае есть две независимые периферии, следующий узел, который должен быть развернут, всегда будет один — с минимальным значением функции оценки, в пределах любой из двух периферий. Если функция оценки представляет собой стоимость пути, мы получаем двунаправленный поиск по критерию стоимости, при этом если стоимость оптимального пути равна C^*, то ни один узел со стоимостью > $C^*/2$ не будет развернут. Такой подход может привести к значительному ускорению работы алгоритма.

Общий алгоритм двунаправленного поиска по первому наилучшему совпадению представлен на рис. 3.14. На входе функции передаются по две версии задачи и функции оценки, первая версия для прямого направления (индекс F), а вторая — для обратного (индекс B). Известно, что когда функция оценки является стоимостью пути, первое найденное решение будет оптимальным решением, но при наличии двух различных функций оценки это утверждение не всегда справедливо. Следовательно, необходимо сохранять последовательность найденных на данный момент наилучших решений, и, возможно, потребуется несколько раз его обновить, пока тест TERMINATED не докажет, что возможных лучших решений больше не осталось.

3.4.6. Сравнение алгоритмов неинформированного поиска

На рис. 3.15 приведены результаты сравнения рассмотренных выше алгоритмов неинформированного поиска в терминах четырех критериев оценки, сформулированных в разделе 3.3.4. Это сравнение проведено для версий древовидного поиска, не проверяющих наличия повторяющихся состояний. При реализации поиска в графе с выполнением такой проверки основные различия будут состоять в том, что поиск в глубину будет полным для конечных пространств состояний, а пространственная и временная сложность ограничены размерами пространства состояний (количество вершин и ребер, $|V|+|E|$).

3.5. Стратегии информированного (эвристического) поиска

В данном разделе показано, как стратегия ▶**информированного поиска** (такая, в которой, кроме определения самой задачи, используются знания, относящиеся к данной конкретной проблемной области) позволяет находить решения более эффективно, чем стратегия неинформированного поиска. Необходимые подсказки поступают от ▶**эвристической функции**, обозначаемой как $h(n)$:[10]

[10] Может показаться странным, что эвристическая функция работает с узлом, когда нам в действительности нужно лишь состояние узла. Это просто традиция — использовать

$h(n)$ = оценка стоимости самого дешевого пути от состояния в узле n до целевого состояния.

Например, в задаче поиска маршрута в Румынии можно было бы оценивать дистанцию от текущего узла до целевого, вычисляя по карте расстояние по прямой линии между этими двумя точками. В разделе 3.6 будет подробнее рассказано об эвристиках и способах их отыскания.

```
function BiBF-Search(problem_F, f_F, problem_B, f_B) returns искомый узел или failure
  node_F ← Node(problem_F.Initial)                    // Узел для исходного состояния
  node_B ← Node(problem_B.Initial)                    // Узел для целевого состояния
  frontier_F ← очередь с приоритетом, упорядоченная по f_F, и элементом node_F
  frontier_B ← очередь с приоритетом, упорядоченная по f_B, и элементом node_B
  reached_F ← таблица поиска с одним ключом node_F.State и значением node_F
  reached_B ← таблица поиска с одним ключом node_B.State и значением node_B
  solution ← failure
  while not Terminated(solution, frontier_F, frontier_B) do
    if f_F(Top(frontier_F)) < f_B(Top(frontier_B)) then
      solution ← Proceed(F, problem_F, frontier_F, reached_F, reached_B, solution)
    else solution ← Proceed(B, problem_B, frontier_B, reached_B, reached_F, solution)
  return solution

function Proceed(dir, problem, frontier, reached, reached_2, solution) returns решение
        // Разворачивает узел в периферии; сравнивает с другой периферией в reached_2.
        // Переменная "dir" определяет направление поиска: F — вперед, B — назад.
  node ← Pop(frontier)
  for each child in Expand(problem, node) do
    s ← child.State
    if s not in reached or Path-Cost(child) < Path-Cost(reached[s]) then
      reached[s] ← child
      добавить child в frontier
      if s is in reached_2 then
        solution_2 ← Join-Nodes(dir, child, reached_2[s]))
        if Path-Cost(solution_2) < Path-Cost(solution) then
          solution ← solution_2
  return solution
```

Рис. 3.14. Двунаправленный поиск по первому наилучшему совпадению поддерживает две очереди периферии $frontier_{dir}$ и две таблицы достигнутых состояний $reached_{dir}$. Когда путь в одной из периферий достигает состояния, которое уже было достигнуто в другой половине поиска, эти два пути объединяются (функция JOIN-NODES) для формирования решения. Первое решение, которое будет найдено, не обязательно будет оптимальным, — для определения, когда можно будет прекратить поиск новых решений, используется функция TERMINATED

функцию $h(n)$ вместо функции $h(s)$ для поддержания соответствия с функцией оценки $f(n)$ и функцией стоимости пути $g(n)$.

Характеристика	Поиск в ширину	Поиск по критерию стоимости	Поиск в глубину	Поиск с ограничением глубины	Поиск с итеративным углублением	Двунаправленный поиск (если применим)
Полнота	Да[а]	Да[а, б]	Нет	Нет	Да[а]	Да[а, г]
Оптимальность	Да[в]	Да	Нет	Нет	Да[в]	Да[в, г]
Временная сложность	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Пространственная сложность	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$

Рис. 3.15. Оценка алгоритмов неинформированного поиска. Здесь b — коэффициент ветвления; m — максимальная глубина дерева поиска; d — глубина решения с наименьшей глубиной и будет равно m, если решения нет; ℓ — лимит глубины поиска. Буквенные индексы имеют следующее значение: ***а*** — полный, если b конечно, а пространство состояний либо содержит решение, либо конечно; ***б*** — полный, если стоимость всех действий $\geq \epsilon > 0$; ***в*** — оптимальный по затратам, если стоимость всех действий одинакова; ***г*** — если в обоих направлениях выполняется поиск в ширину или по критерию стоимости

3.5.1. Жадный поиск по первому наилучшему совпадению

▶**Жадный поиск по первому наилучшему совпадению** — это форма поиска по первому наилучшему совпадению, в которой первым развертывается узел с наименьшим значением $h(n)$, т.е. узел, который кажется наиболее близким к цели, исходя из предположения, что это может быстрее привести к цели. Поэтому функцией оценки в данном случае будет $f(n) = h(n)$.

Давайте посмотрим, как эта стратегия работает на примере задачи поиска маршрута в Румынии. Будем использовать эвристику ▶**расстояние по прямой** (Straight Line Distance — SLD), для которой примем обозначение h_{SLD}. Если целью является Бухарест, то необходимо знать расстояния по прямой от него до каждого города, приведенного на рис. 3.1, — эти данные приведены на рис. 3.16. Например, $h_{SLD}(Arad) = 366$. Обратите внимание, что значения функции h_{SLD} не могут быть вычислены на основании описания самой задачи (т.е. функций ACTIONS и RESULT). Более того, здесь также нужны определенные знания о мире, чтобы понять, как значения функции h_{SLD} связаны с действительными дорожными расстояниями и, следовательно, могут ли они являться полезной эвристикой.

На рис. 3.17 показан процесс выполнения жадного поиска по первому наилучшему совпадению с использованием значений эвристики h_{SLD} для определения пути от Арада до Бухареста. Первым узлом, подлежащим развертыванию после узла *Arad*, является узел *Sibiu*, поскольку город Сибиу находится ближе к Бухаресту, чем города Зеринд или Тимишоара. Следующим узлом, подлежащим развертыванию, является узел *Fagaras*, поскольку теперь ближайшим к Бухаресту является город Фэгэраш. Узел *Fagaras*, в свою очередь, обеспечивает формирование узла *Bucharest*, который и является целевым. Применение в процессе решения

данной конкретной задачи алгоритма жадного поиска по первому наилучшему совпадению с использованием эвристической функции h_{SLD} позволило найти решение без развертывания какого-либо узла, не находящегося на пути решения. Однако найденное решение не является оптимальным: путь до Бухареста через города Сибиу и Фэгэраш на 32 мили длиннее, чем путь через города Сибиу, Рымнику Вылча и Питешти. Вот почему данный алгоритм называется "жадным": на каждом этапе он пытается подойти к цели как можно ближе (фигурально выражаясь, "захватить как можно больше"), однако жадность может привести к худшим результатам в сравнении с внимательностью.

Обозначение узла	Город	Расстояние по прямой до Бухареста	Обозначение узла	Город	Расстояние по прямой до Бухареста
Arad	**Арад**	366	*Mehadia*	**Мэхадия**	241
Bucharest	**Бухарест**	0	*Neamt*	**Нямц**	234
Craiova	**Крайова**	160	*Oradea*	**Орадя**	380
Drobeta	**Дробета**	242	*Pitesti*	**Питешти**	100
Eforie	**Эфорие**	161	*Rimnicu Vilcea*	**Рымнику Вылча**	193
Fagaras	**Фэгэраш**	176	*Sibiu*	**Сибиу**	253
Giurgiu	**Джуржду**	77	*Timisoara*	**Тимишоара**	329
Hirsova	**Хыршова**	151	*Urziceni*	**Урзичени**	80
Iasi	**Яссы**	226	*Vaslui*	**Васлуй**	199
Lugoj	**Лугож**	244	*Zerind*	**Зеринд**	374

Рис. 3.16. Значения функции h_{SLD} — расстояния по прямой линии до Бухареста

Жадный поиск по первому наилучшему совпадению является полным только в конечных пространствах состояний и не является таковым в бесконечных. В наихудшем случае временная и пространственная сложность этого алгоритма оценивается как $O(|V|)$. Однако с хорошей эвристической функцией его сложность может быть существенно уменьшена и для некоторых задач достигать $O(bm)$.

3.5.2. Поиск A*

Наиболее распространенный алгоритм информированного поиска носит название ▶ **поиск A*** (произносится "поиск А-звездочка"). Это алгоритм поиска по первому наилучшему совпадению, использующий функцию оценки

$$f(n) = g(n) + h(n),$$

где $g(n)$ — это стоимость пути от начального состояния до узла n, а $h(n)$ — это расчетная стоимость кратчайшего пути от узла n до целевого состояния, поэтому мы имеем

$f(n)$ = оценочная стоимость наилучшего пути, который идет через узел n к цели.

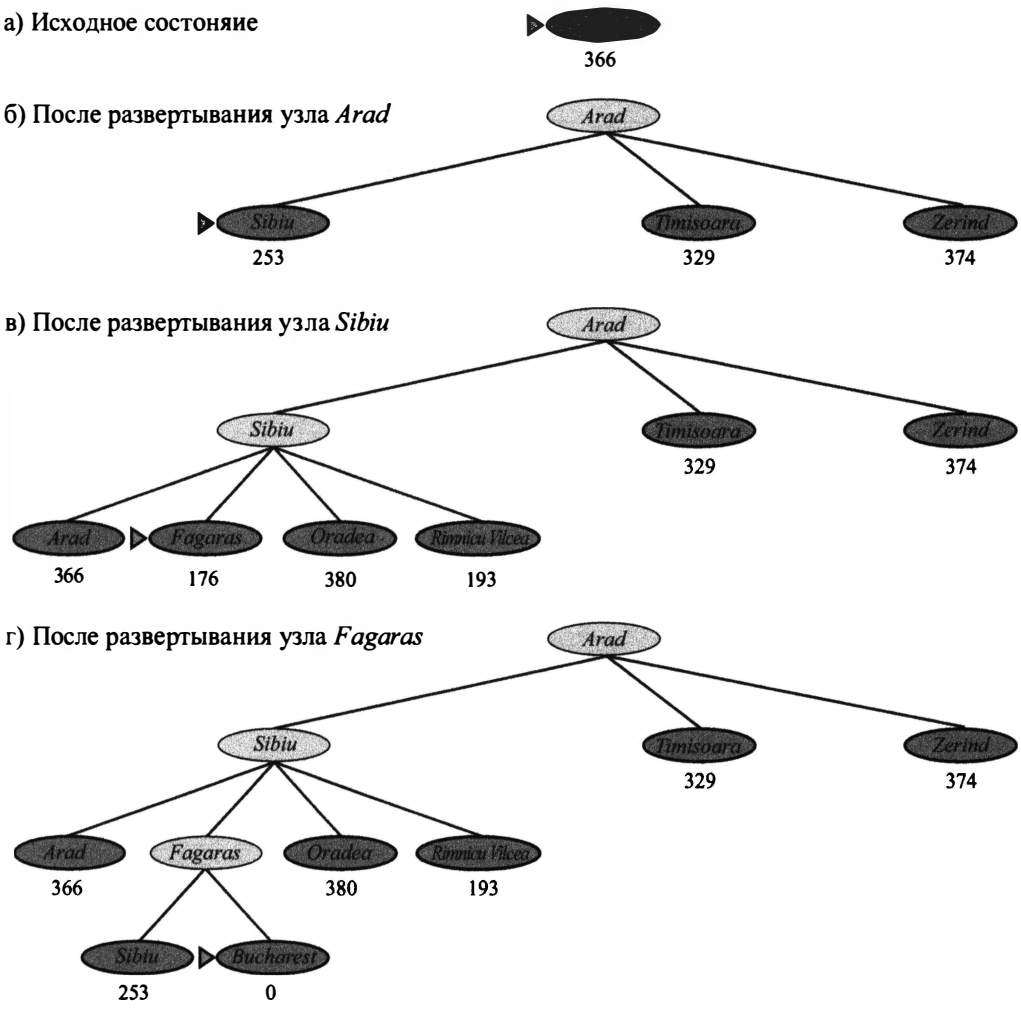


Рис. 3.17. Этапы выполнения жадного поиска по первому наилучшему совпадению с использованием эвристической функции h_{SLD} — расстояния до Бухареста по прямой линии. На рисунке для всех узлов приведено их h-значение

На рис. 3.18 представлены этапы поиска по алгоритму A* с целью попасть из Арада в Бухарест. Значения функции g вычисляются на основании стоимости действий, как показано на рис. 3.1, а значения функции h_{SLD} приведены на рис. 3.16. Обратите внимание, что узел *Bucharest* впервые появляется в периферии на этапе *д*, но в этом случае он не выбирается для развертывания (и поэтому не обнаруживается как целевой узел), так как при значении $f = 450$ он не является узлом с наименьшей стоимостью в пределах периферии, поскольку в ней есть узел *Pitesti* с $f = 417$. Иначе можно сказать, что так происходит потому, что *может*

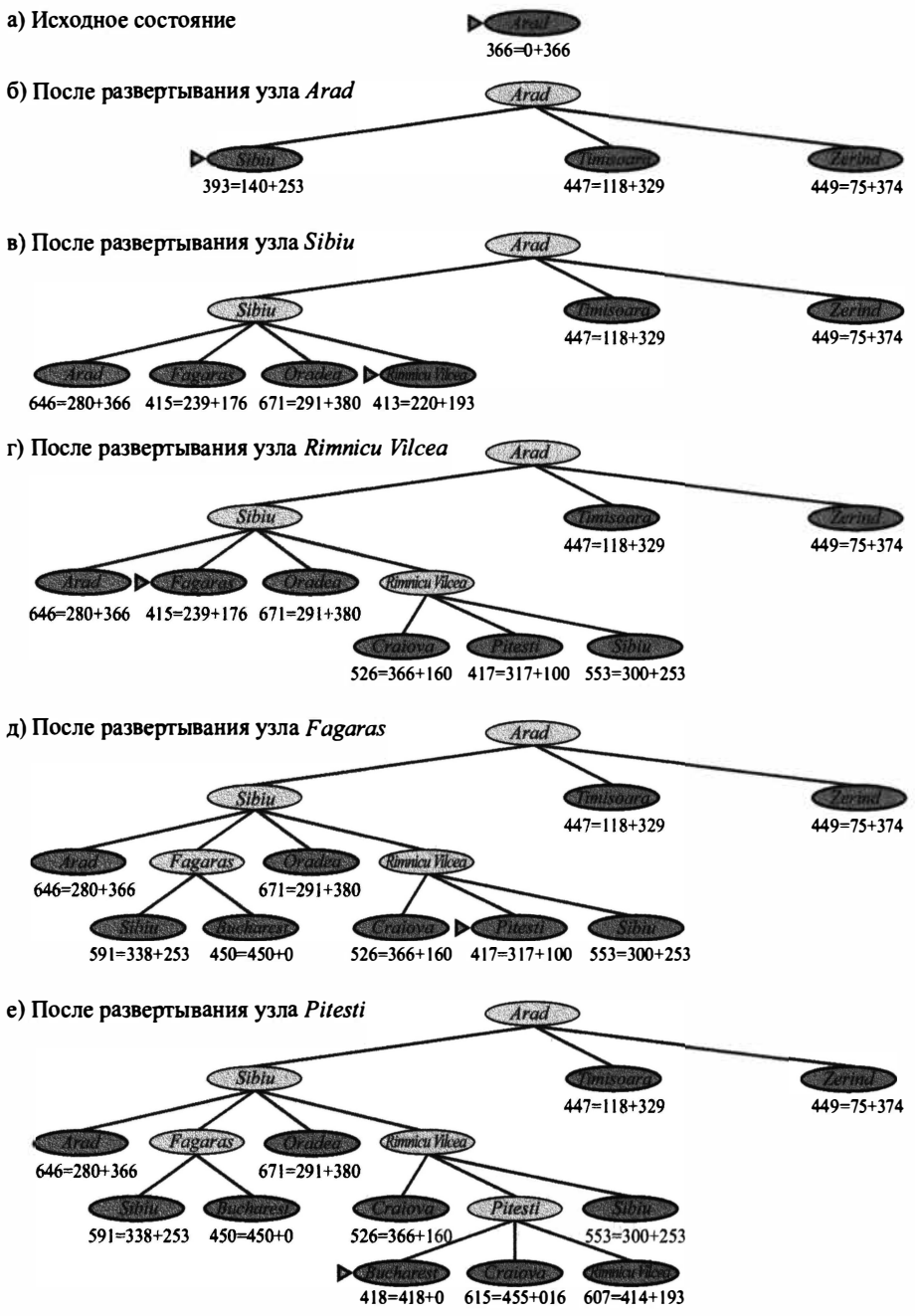


Рис. 3.18. Этапы выполнения алгоритма поиска A* для определения оптимального пути в Бухарест. Для всех узлов указаны значения функции $f = g + h$. Значения h — это расстояния по прямой до Бухареста, представленные на рис. 3.16

существовать решение, проходящее через Питешты, стоимость которого — всего лишь 417, так что алгоритм не будет полагаться на другое решение, стоимость которого составляет 450. На этапе *д* выбирается другой путь к Бухаресту, который второй раз появляется в периферии, и теперь этот узел является самым дешевым при $f=418$. Именно поэтому он выбирается следующим для развертывания и обнаруживается как оптимальное решение.

Поиск A* является полным.[11] Будет ли поиск A* оптимальным по затратам, зависит от определенных свойств выбранной эвристики. Ключевым свойством в данном случае является **допустимость**: ▶**допустимой эвристикой** является такая эвристика, которая *никогда* не *переоценивает* стоимость достижения цели. (Следовательно, допустимая эвристика является *оптимистической*.) При применении допустимой эвристики алгоритм поиска A* является оптимальным по затратам, что можно доказать методом от противного. Предположим, что оптимальный путь имеет стоимость C^*, но алгоритм при выполнении возвращает путь стоимостью $C > C^*$. Тогда должен существовать какой-то узел n, который находится на оптимальном пути и является неразвернутым (потому, что если все узлы на оптимальном пути были развернуты, то должен был возвращен этот оптимальный путь). Поэтому, используя нотацию $g^*(n)$ для обозначения стоимости оптимального пути из исходного узла до узла n и нотацию $h^*(n)$ для обозначения стоимости оптимального пути от узла n до ближайшей цели, мы имеем:

$f(n) > C^*$	(иначе узел n был бы развернут)
$f(n) = g(n) + h(n)$	(по определению)
$f(n) = g^*(n) + h(n)$	(поскольку n находится на оптимальном пути)
$f(n) \leq g^*(n) + h^*(n)$	(по свойству допустимости $h(n) \leq h^*(n)$)
$f(n) \leq C^*$	(по определению $C^* = g^*(n) + h^*(n)$)

Первая и последняя строки противоречат одна другой, поэтому предположение о том, что алгоритм может возвращать неоптимальный путь, следует считать неверным, — значит, можно утверждать, что поиск A* возвращает только оптимальные по затратам пути.

Немного более строгое свойство эвристики называется ▶**преемственностью** или **монотонностью**. Эвристика $h(n)$ является преемственной, если для любого узла n и его преемника n', формируемого действием a, мы имеем

$$h(n) \leq c(n, a, n') + h(n').$$

Это требование является одной из форм ▶**неравенства треугольника**, постулирующего требование о том, что любая из сторон треугольника не может быть больше, чем сумма двух других его сторон (рис. 3.19). Примером преемственной эвристики является расстояние по прямой линии h_{SLD}, — ее мы использовали, чтобы найти путь до Бухареста.

[11] Опять же, предполагая, что стоимости всех действий $> \epsilon > 0$ и пространство состояний либо содержит решение, либо является конечным.

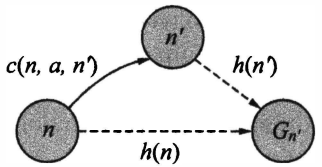


Рис. 3.19. Неравенство треугольника: если эвристика h является **преемственной** (или **монотонной**), то значение $h(n)$ всегда будет меньше, чем сумма стоимости $c(n, a, n')$ действия от узла n к узлу n' и эвристической оценки $h(n')$

Каждая преемственная эвристика является допустимой (но не наоборот), так что при использовании преемственной эвристики алгоритм поиска A* является оптимальным по затратам. Кроме того, при преемственной эвристике в первый же раз, когда некоторое состояние достигается, оно будет находиться на оптимальном пути, поэтому больше не потребуется еще раз добавлять его в периферию и изменять запись в таблице *reached* (достигнутые). С другой стороны, при использовании немонотонной эвристики поиск может закончиться с несколькими путями, ведущими к одному и тому же состоянию, и если каждый новый путь будет иметь более низкую стоимость, чем предыдущий, то в конечном итоге поиск закончится с несколькими узлами в периферии для одного и того же состояния, на что будет затрачено дополнительное время и память. По этой причине в некоторых реализациях алгоритма поиска A* позаботились о том, чтобы каждое состояние могло войти в периферию только один раз, и если к состоянию найден лучший путь, все преемники этого состояния будут обновлены (такой подход требует, чтобы узлы содержали указатели на узлы-потомки и родительские узлы). Подобные осложнения вынудили многих реализаторов избегать немонотонных эвристик, но Фельнер и др. ([725], 2011) утверждают, что худшие последствия редко имеют место на практике, а значит, не следует бояться использовать немонотонные эвристики.

При недопустимой эвристике алгоритм поиска A* может быть или не быть оптимальным по затратам. Вот два случая, когда это имеет место. Во-первых, если есть хотя бы один оптимальный по затратам путь, на котором функция $h(n)$ является допустимой для всех узлов n в пути, то этот путь будет найден независимо от того, что эвристики говорят в отношении состояний на этом пути. Во-вторых, если оптимальное решение имеет стоимость C^*, а второй из лучших путей имеет стоимость C_2 и если функция $h(n)$ переоценивает некоторые стоимости, но всегда не больше, чем $C_2 - C^*$, то алгоритм поиска A* гарантированно вернет оптимальное по затратам решение.

3.5.3. Контуры поиска

Полезный способ визуализации процесса поиска состоит в том, чтобы рисовать ▶**контуры** в пространстве состояний, подобно контурам равных высот на

топографической карте. На рис. 3.20 представлен соответствующий пример. Внутри контура, помеченного цифрой 400, все узлы имеют $f(n) = g(n) + h(n) \leq 400$ и т.д. Далее, поскольку в поиске A* всегда развертывается периферийный узел с минимальной f-стоимостью, можно заметить, что поиск распространяется из начального узла, добавляя новые узлы в виде концентрических полос с возрастающей f-стоимостью.

При поиске по критерию стоимости также можно построить контуры, но уже по g-стоимости, а не по значению $g+h$. В этом случае контуры будут представлять собой концентрические кольца с центром в начальном состоянии, равномерно распространяющиеся во всех направлениях, без предпочтения в отношении направления к цели. Если использовать более точные эвристические функции, полосы будут вытягиваться в направлении целевого состояния, теснее охватывая оптимальный путь, как показано на рис. 3.20.

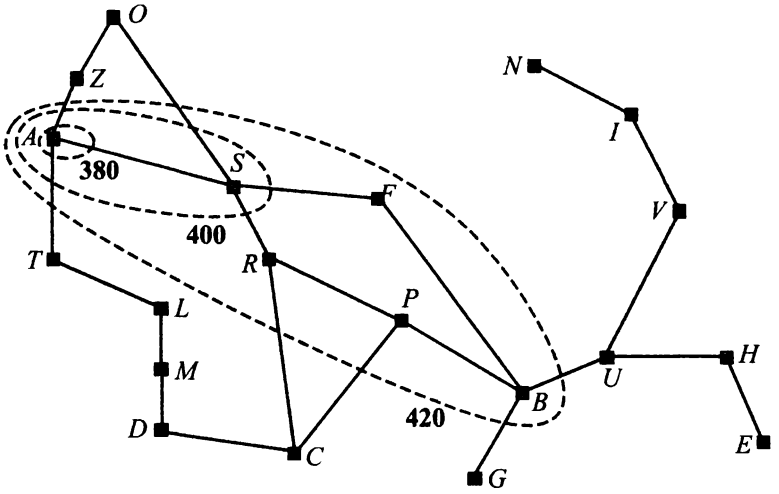


Рис. 3.20. Карта дорог Румынии, на которую нанесены контуры, соответствующие $f=380$, $f=400$ и $f=420$, с городом Арад в качестве начального состояния. Узлы внутри каждого такого контура имеют стоимость $f = g + h$, меньшую или равную значению контура

Должно быть очевидно, что при развертывании пути функция g-стоимости является ▶**монотонной**: при движении по пути стоимость пути постоянно увеличивается, поскольку стоимость любого действия всегда положительна.[12] В результате образуются концентрические линии контуров, не пересекающиеся между собой,

[12] Если говорить точнее, для функции затрат, значения которой непрерывно увеличиваются, используется термин "строго монотонная", а термин "монотонная" предназначен для функции затрат, значения которой никогда не уменьшаются, но могут оставаться неизменными.

и если наносить эти линии достаточно тонкими, то их всегда можно будет провести между любыми двумя узлами на любом пути.

Однако не всегда очевидно, будет ли монотонно возрастать функция стоимости $f=g+h$. При расширении пути от n к n' стоимость меняется от $g(n)+h(n)$ к $g(n)+c(n,a,n')+h(n')$. Отбрасывая общий член $g(n)$, видим, что стоимость пути будет монотонно возрастать тогда и только тогда, когда $h(n) \le c(n,a,n')+h(n')$, — другими словами, тогда и только тогда, когда эвристика является преемственной.[13] Но обратите внимание, что путь может включать несколько узлов, образующих ряд с одним и тем же значением суммы $g(n)+h(n)$. Эта ситуация будет иметь место всякий раз, когда уменьшение значения h будет в точности равно стоимости выполняемого действия, например в клеточном мире, когда узел n находится в той же строке, что и целевой узел, и предпринимается переход в направлении к цели, значение g увеличивается на 1, а значение h уменьшается на 1. Если C^* — стоимость пути оптимального решения, то мы можем сказать следующее.

- В поиске A* развертываются все узлы, которые могут быть достигнуты из начального состояния на пути, где каждый его узел имеет $f(n)<C^*$. Говорят, что это ▶**обязательно развертываемые узлы**.
- В поиске A* перед выбором целевого узла могут также развертываться узлы, находящиеся непосредственно на "целевом контуре" (т.е. где $f(n)=C^*$).
- В поиске A* не развертываются узлы с $f(n)>C^*$.

Можно утверждать, что алгоритм поиска A* ▶**оптимально эффективен** в том смысле, что любой алгоритм, который начинает поиск пути из начального состояния и использует ту же самую эвристическую информацию, должен будет развернуть *не меньше* узлов, чем при поиске по алгоритму A* (потому что любой из них мог бы оказаться частью пути к оптимальному решению). Один алгоритм может действовать более удачно и из узлов с $f(n)=C^*$ выбрать оптимальный первым, тогда как другой алгоритм в этом отношении может оказаться менее удачливым, — в определении оптимальной эффективности эти различия не учитываются.

Алгоритм поиска A* является эффективным потому, что в дереве поиска ▶**отсекаются** все узлы, которые не потребуются в дальнейшем поиске оптимального решения. На рис. 3.18, *б* видно, что узел *Timisoara* имеет $f=447$, а узел *Zerind* — $f=449$. Хотя эти узлы и являются дочерними для корневого узла дерева и должны были бы находиться в числе первых узлов, развернутых при поиске по критерию стоимости или при поиске в ширину, они никогда не будут развернуты при поиске A*, потому что уже было найдено решение с $f=418$. Концепция *отсечения* — исключения из рассмотрения вариантов без необходимости их изучения — очень важна для многих областей ИИ.

---

[13] В действительности термин "монотонная эвристика" является синонимом термина "последовательная эвристика". Эти две идеи были разработаны независимо одна от другой, но позднее было доказано, что они эквивалентны ([1751], 1984).

Тот факт, что поиск A* является полным, оптимальным по затратам и оптимально эффективным среди всех подобных алгоритмов, оставляет очень приятное впечатление. К сожалению, это не означает, что поиск A* является ответом на все возможные потребности выполнения поиска. Основной недостаток состоит в том, что для многих задач число развернутых этим алгоритмом узлов может расти экспоненциально с увеличением длины решения. Например, рассмотрим вариант мира пылесоса с супермощным пылесосом, способным убрать одну любую клетку при стоимости 1 даже без необходимости посещения этого квадрата, — в этом сценарии квадраты можно убирать в любом порядке. Если изначально имеется N квадратов с мусором, то существует 2^N состояния, когда некоторое их подмножество уже было убрано. Все эти состояния находятся на оптимальном пути решения и, следовательно, удовлетворяют требованию $f(n) < C^*$, поэтому все они будут посещены при поиске A*.

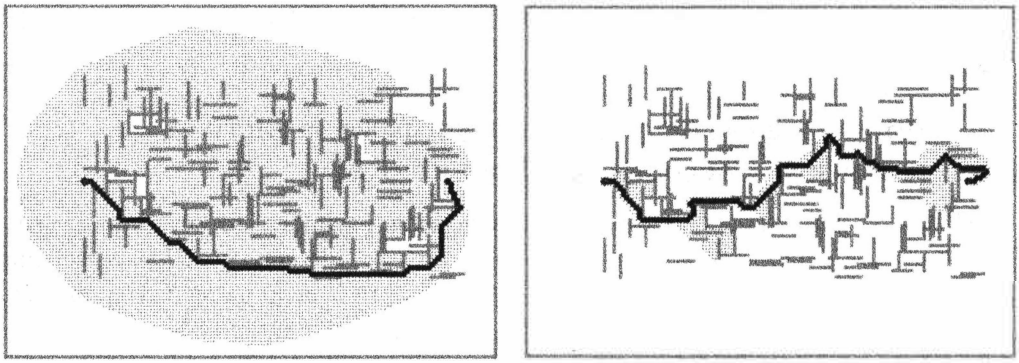
3.5.4. Приближенный поиск: недопустимые эвристики и поиск A* с весовым коэффициентом

Поиск A* характеризуется множеством хороших качеств, но, отыскивая оптимальный путь, он развертывает слишком много узлов. Можно исследовать меньше узлов (что потребует меньше времени и пространства), если согласиться принять решение, которое не будет оптимальным, но будет "достаточно хорошим" — то, что принято называть ▶**близким к оптимальному** или ▶**субоптимальным** решением. Если в поиске A* позволить использовать недопустимую эвристику — т.е. такую, которая может переоценивать стоимость достижения цели, — то появляется риск не найти оптимальное решение, но потенциально эта эвристика может оказаться более точной, уменьшая тем самым количество развертываемых узлов. Так, например, дорожные инженеры используют понятие ▶**индекс объезда**, представляющий собой множитель, применяемый к расстоянию по прямой для вычисления стандартного показателя извилистости дорог. Значение индекса объезда, равное 1,3, означает, что если расстояние по прямой между двумя населенными пунктами равно 10 км, то хорошая оценка оптимального пути между ними составит 13 км. Для большинства населенных пунктов индекс объезда находится в пределах от 1,2 до 1,6.

Эту идею можно применить к любой задаче, а не только к тем, которые связаны с дорогами, — такой подход называется ▶**поиском A* с весовым коэффициентом**, где эвристической оценке придается больший вес за счет использования весового коэффициента W, а функция оценки приобретает следующий вид: $f(n) = g(n) + W \times h(n)$ для некоторого $W > 1$.

На рис. 3.21 представлена задача поиска в клеточном мире. На рис. 3.21, *а* выполнение поиска A*, позволило найти оптимальное решение, но для этого потребовалось исследовать большую часть пространства состояний. На рис. 3.21, *б* использование поиска A* с весовым коэффициентом позволило найти субоптимальное

решение, немного превышающее оптимальное, но этот результат был достигнут намного быстрее. Можно видеть, что при поиске A* с весовым коэффициентом контур достигнутых состояний фокусируется на достижении цели. Это означает, что будет проанализировано меньшее количество состояний, но если оптимальный путь в какой-то момент выйдет за пределы контура поиска A* с весовым коэффициентом (что и произошло в данном случае), то оптимальный путь так и не будет найден. В общем случае, если оптимальное решение имеет стоимость C^*, алгоритм поиска A* с весовым коэффициентом сможет найти решение со стоимостью в диапазоне от C^* до $W \times C^*$. Однако на практике обычно получают результаты, которые гораздо ближе к C^*, чем к $W \times C^*$.



а) б)

Рис. 3.21. Два варианта поиска в одном и том же клеточном мире: *а*) поиск A* и *б*) поиск A* с весовым коэффициентом $W=2$. Серые полосы представляют препятствия, а темная линия — путь от начального состояния (слева) к целевому состоянию (справа). Маленькие серые точки — это достигнутые состояния в каждом поиске. В этой конкретной задаче при поиске по алгоритму A* с весовым коэффициентом было проверено в 7 раз меньше состояний и найден путь к цели стоимостью, которая на 5% больше стоимости оптимального пути

Итак, выше обсуждались варианты поиска, в которых оценка состояния проводилась на основе различных комбинаций значений функций g и h. Поиск A* с весовым коэффициентом можно рассматривать как обобщение их всех.

Поиск A*:	$g(n)+h(n)$	$(W=1)$
Поиск по критерию стоимости:	$g(n)$	$(W=0)$
Жадный поиск по первому наилучшему совпадению:	$h(n)$	$(W=\infty)$
Поиск A* с весовым коэффициентом:	$g(n)+W \times h(n)$	$(1<W<\infty)$

Можно было бы назвать поиск A* с весовым коэффициентом "немного жадным поиском": как и жадный поиск по первому наилучшему совпадению, он фокусирует ход поиска на цели, однако стоимость пути также не полностью игнорируется, — путь, который дает небольшой прогресс при больших затратах, будет отброшен.

Существует большое разнообразие приближенных алгоритмов поиска, которые можно характеризовать по критериям того, что считается "близким к оптимальному". При ▶ **приближенном поиске с ограничениями** выполняется поиск решения, которое гарантированно будет находиться в пределах, определяемых оптимальной стоимостью с учетом некоторого постоянного коэффициента W. Поиск A* с весовым коэффициентом обеспечивает такую гарантию. В случае ▶ **поиска с ограниченной стоимостью** выполняется поиск решения, стоимость которого будет меньше некоторой заданной величины C. При ▶ **поиске с неограниченной стоимостью** будет принято решение с любой стоимостью, если только удастся найти его достаточно быстро.

Алгоритм поиска с неограниченной стоимостью является примером ▶ **быстрого поиска** и представляет собой версию жадного поиска по первому наилучшему совпадению, в котором в качестве эвристики используется оценка числа действий, необходимых для достижения цели, независимо от стоимости этих действий. Следовательно, для задач, в которых *все* действия с имеют одну и ту же стоимость, это будет все тот же жадный поиск по первому наилучшему совпадению, но если действия будут иметь разную стоимость, такой подход позволит быстрее найти решение, пусть даже оно будет иметь более высокую стоимость.

3.5.5. Поиск с ограничением объема памяти

Основная проблема поиска A* заключается использовании им больших объемов памяти. В этом разделе будут рассмотрены особые приемы реализации стандартного алгоритма поиска A*, обеспечивающие более экономный расход памяти при работе, а затем некоторые совершенно новые алгоритмы, способные извлекать максимальные преимущества из доступного пространства.

В алгоритме A* память разделена между *периферией* (*frontier*) и *достигнутыми состояниями* (*reached*). В нашей реализации поиска по первому наилучшему совпадению информация о тех состояниях, которые относятся к периферии, хранятся в двух местах: в виде узла в структуре периферии (что позволяет быстро принимать решение, какой узел следует развертывать следующим) и как запись в таблице достигнутых состояний (что позволяет сохранить информацию о том, что это состояние уже анализировалось). Для многих задач (таких, как исследование клеточного мира) это дублирование не является проблемой, поскольку размер *периферии* всегда намного меньше, чем количество *достигнутых состояний*, так что дублирование сведений о состояниях в периферии требует сравнительно незначительных дополнительных объемов памяти. Однако в некоторых реализациях

сведения о состояниях хранятся только в одном из этих двух мест, обеспечивая этим некоторую экономию пространства за счет усложнения (и, возможно, замедления работы) алгоритма.

Еще одна возможность — удалять состояния из таблицы *достигнутых*, когда можно доказать, что они больше не потребуются. В некоторых задачах для этой цели можно воспользоваться свойством разделения (см. рис. 3.6) наряду с запретом выбора действий в обратном направлении, что гарантирует, что все действия направлены либо наружу относительно периферии, либо на другое состояние в пределах периферии. В этом случае необходимо будет лишь проверять периферию на наличие избыточных путей, а таблицу достигнутых узлов можно просто аннулировать.

Что касается других проблем, то можно хранить для состояния ▶ **счетчик ссылок**, указывающий сколько раз оно было достигнуто, и удалять его из таблицы *достигнутых*, когда больше не останется способов достичь этого состояния. Например, в клеточном мире каждое состояние может быть достигнуто только из четырех его соседей, поэтому, когда счетчик ссылок состояния станет равен четырем, оно уже будет достигнуто всеми возможными способами и его можно будет удалить из таблицы.

А теперь обратимся к новым алгоритмам, которые были специально разработаны с целью сокращения расходов доступной памяти.

Алгоритм ▶ **поиска по лучу** предполагает ограничение размера периферии. Самый простой подход — оставить только k узлов с лучшими значениями функции f, отбрасывая любые другие развернутые узлы. Подобное решение, конечно, делает поиск неполным и неоптимальным, но позволяет выбрать такое значение k, которое обеспечит эффективное использование доступной памяти, а алгоритм будет выполняться быстрее, поскольку он развертывает меньше узлов. Для многих задач этот алгоритм позволяет найти хорошие, почти оптимальные решения. Можно представить себе поиск по критерию стоимости или поиск A* как расширяющие исследуемую область равномерно во всех направлениях, в виде концентрических контуров, тогда как поиск по лучу расширяется сфокусированно, исследуя только ту часть этих контуров, которая содержит k лучших кандидатов.

Альтернативная версия поиска по лучу не предусматривает задания строгого ограничения на размер периферии, но вместо этого сохраняет каждый узел, значение функции f которого находится в пределах δ лучших значений функции f. В этом варианте, когда имеется лишь несколько узлов с весомыми значениями функции f, расход памяти будет незначителен, но если таких узлов в периферии нет, потребуется много памяти, пока они не появятся.

Алгоритм ▶ **поиска A* с итеративным углублением** (IDA*) представляет собой алгоритм A*, в котором используется идея итеративного поиска в глубину. Алгоритм IDA* обеспечивает все преимущества поиска A* без необходимости сохранять все достигнутые состояния в памяти по причине возможного посещения некоторых состояний несколько раз. Это очень важный и часто используемый алгоритм для задач, которые не помещаются в доступной памяти.

В стандартном алгоритме итеративного углубления вводятся ограничения по глубине, которая увеличивается с каждой итерацией. В алгоритме IDA* условием прекращения развертывания узлов является *f*-стоимость ($g+h$). На каждой итерации пределом будет наименьшая *f*-стоимость любого узла, которая превосходит предельное ограничение на предыдущей итерации. Другими словами, на каждой итерации исчерпывающе исследуется очередной *f*-контур и отыскивается узел сразу за этим контуром, *f*-стоимость которого будет использоваться как ограничение в следующем контуре. Для таких задач, как головоломка ***Восемь***, где *f*-стоимость каждого пути является целым числом, этот алгоритм работает очень хорошо, добиваясь неуклонного прогресса в достижении цели на каждой итерации. Если оптимальное решение имеет стоимость C*, то итераций не может быть больше, чем C* (например, не более 31 итерации для самых трудных раскладов в головоломке ***Восемь***). Но для задач, в которых каждый узел имеет разную *f*-стоимость, каждый новый контур может состоять только из одного нового узла, а количество итераций может оказаться равным числу состояний в задаче.

Алгоритм ▶**рекурсивного поиска по первому наилучшему совпадению** (Recursive best-first search — RBFS) — это простой рекурсивный алгоритм (рис. 3.22), в котором предпринята попытка имитировать работу стандартного алгоритма поиска по первому наилучшему совпадению, но с использованием только линейного пространства. Алгоритм RBFS напоминает рекурсивный поиск в глубину, но вместо бесконечного следования вниз по текущему пути данный алгоритм использует переменную *f_limit*, чтобы отслеживать *f*-значение наилучшего *альтернативного* пути, доступного из любого предка текущего узла. Если *f*-значение очередного узла превышает данный предел, то текущий этап рекурсии отменяется до альтернативного пути, и рекурсия продолжается уже с него. В процессе отмены рекурсии алгоритм RBFS заменяет *f*-значение каждого узла вдоль пути резервной копией — лучшим *f*-значением его дочернего узла. Благодаря этому в алгоритме RBFS запоминается *f*-значение наилучшего листового узла из забытого поддерева, и поэтому в некоторый последующий момент времени может быть принято решение о том, стоит ли снова развертывать это поддерево. На рис. 3.23 показано, как происходит поиск пути в Бухарест при использовании алгоритма RBFS.

Алгоритм RBFS несколько более эффективен, чем IDA*, но все еще страдает от слишком частого повторного формирования узлов. В примере, приведенном на рис. 3.23, алгоритм RBFS вначале следует по пути через узел *RimnicuVilcea*, затем "меняет решение" и пытается пройти через узел *Fagaras*, после чего снова возвращается к отвергнутому ранее решению. Такие смены решения происходят в связи с тем, что при каждом развертывании текущего наилучшего пути велика вероятность того, что его *f*-значение увеличится, поскольку функция *h* часто становится менее оптимистической для узлов, более близких к цели. Когда это происходит, путь, который был вторым после наилучшего, может теперь стать наилучшим путем, поэтому в алгоритме поиска приходится выполнять возврат,

чтобы проследовать по нему. Каждое изменение решения соответствует одной итерации алгоритма IDA* и может потребовать многих повторных развертываний забытых узлов для воссоздания наилучшего пути и развертывания пути еще на один узел.

```
function RECURSIVE-BEST-FIRST-SEARCH(problem) returns решение или failure
    solution, f_value ← RBFS(problem, NODE(problem.INITIAL), ∞)
  return solution

function RBFS(problem, node, f_limit) returns решение или failure и новый
                                                предел f-стоимости
  if problem.IS-GOAL(node.STATE) then return node
  successors ← LIST(EXPAND(node))
  if successors is empty then return failure, ∞
  for each s in successors do    // обновление f-стоимости значением
                                 // из предыдущего поиска
      s.f ← max(s.PATH-COST + h(s), node.f))
  while true do
      best ← узел из successors с самым низким значением f-стоимости
      if best.f > f_limit then return failure, best.f
      alternative ← второй из узлов с самым низким значением f-стоимости
                    из successors
      result, best.f ← RBFS(problem, best, min(f_limit, alternative))
      if result ≠ failure then return result, best.f
```

Рис. 3.22. Алгоритм рекурсивного поиска по первому наилучшему совпадению

Алгоритм RBFS будет оптимальным, если эвристическая функция $h(n)$ допустима. Его пространственная сложность линейна пропорционально глубине самого глубокого оптимального решения — $O(bd)$, а вот временную его сложность охарактеризовать довольно трудно: она зависит как от точности эвристической функции, так и от того, как часто будет изменяться лучший путь по мере развертывания узлов. Алгоритм развертывает узлы в порядке увеличения значения функции f, даже если эта функция будет немонотонной.

Алгоритмы IDA* и RBFS страдают от того, что в них используется *слишком мало* памяти. Между итерациями алгоритм IDA* сохраняет только единственное число — текущий предел f-стоимости. Алгоритм RBFS сохраняет в памяти больше информации, но количество используемой в нем памяти с усложнением задачи возрастает линейно и характеризуется как $O(bd)$: даже если будет доступно больше памяти, алгоритм RBFS будет не способен ею воспользоваться. Поскольку эти алгоритмы забывают большую часть информации, которую уже получили, оба они в конечном итоге могут многократно заново исследовать одни и те же состояния.

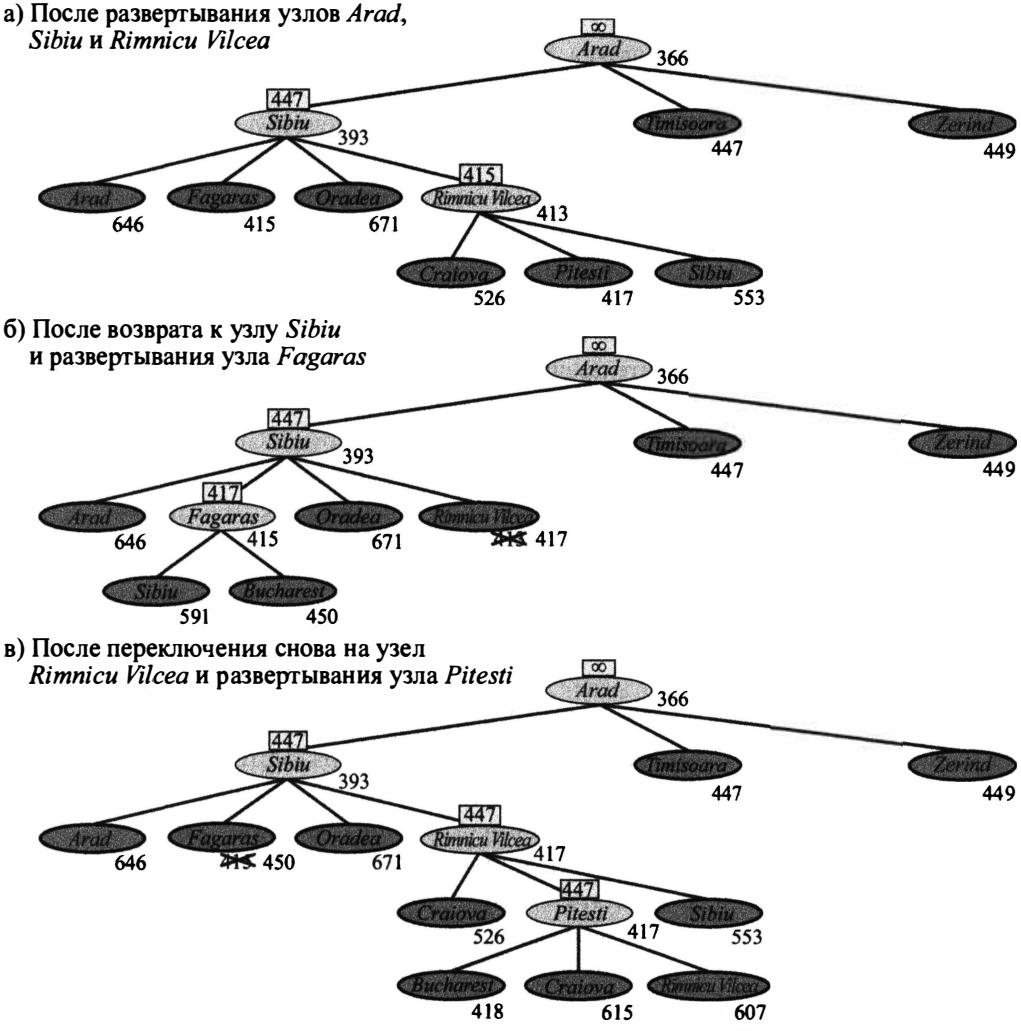


Рис. 3.23. Этапы поиска кратчайшего маршрута в Бухарест с использованием алгоритма RBFS. Значение переменной *f_limit* для каждого рекурсивного вызова приведено над каждым текущим узлом, а под каждым узлом указана его *f*-стоимость. *а*) Путь через узел *RimnicuVilcea*, ведущий до текущего наилучшего листового узла (*Pitesti*), имеет *f*-значение, худшее по сравнению с наилучшим альтернативным путем (*Fagaras*). *б*) Рекурсия отменяется и наилучшая *f*-стоимость листового узла забытого поддерева (417) сохраняется в узле *Rimnicu Vilcea*. Далее развертывается узел *Fagaras* и открывается наилучшее значение листового узла, равное 450. *в*) Рекурсия отменяется и значение наилучшего листового узла забытого поддерева (450) сохраняется в узле *Fagaras*, затем снова развертывается узел *RimnicuVilcea*. На этот раз развертывание продолжается в сторону Бухареста, поскольку наилучший альтернативный путь (через узел *Timisoara*) стоит по меньшей мере 447

Поэтому представляется разумным определить, сколько памяти доступно, и позволить алгоритму использовать ее всю. Двумя алгоритмами, в которых это требование реализуется, являются **▶поиск A* с ограничением памяти** (Memory-bounded A* — **▶MA***) и **▶упрощенный поиск MA*** (Simplified MA* — **▶SMA***). Алгоритм SMA* проще, поэтому в этом разделе будет рассмотрен именно он. Алгоритм SMA* действует полностью аналогично поиску A*, развертывая наилучшие листовые узлы до тех пор, пока будет исчерпана доступная память. С этого момента он не может добавить новый узел к дереву поиска, не уничтожив старый. В алгоритме SMA* всегда уничтожается *наихудший* листовой узел (тот, который имеет наибольшее *f*-значение). После этого, как и в алгоритме RBFS, в алгоритме SMA* *f*-значение забытого (уничтоженного) узла резервируется в его родительском узле. В результате предок забытого поддерева позволяет определить качество наилучшего пути в этом поддереве. Благодаря этому в алгоритме SMA* поддерево восстанавливается только тогда, когда обнаруживается, что все другие пути выглядят менее многообещающими по сравнению с забытым путем. Иными словами, если все потомки узла *n* забыты, то неизвестно, каким путем можно следовать от *n*, но все еще можно получить представление о том, насколько целесообразно следовать куда-либо от *n*.

Полный алгоритм описан в интерактивном репозитории кода, прилагаемом к этой книге. Однако есть один нюанс, который стоит упомянуть. Выше было сказано, что алгоритм SMA* развертывает лучший листовой узел и удаляет худший лист. Но что делать, если все листовые узлы имеют одно и то же *f*-значение? Чтобы избежать выбора одного и того же узла и для удаления, и развертывания, в алгоритме SMA* развертывается *самый новый* лучший лист и удаляется самый *старый* худший лист. Они совпадают, только если имеется лишь один лист, но в этом случае текущее дерево поиска должно представлять единственный путь от корня до листового узла, заполняющий всю память. И если этот лист не является целевым узлом, то решение при доступном объеме памяти недостижимо, *даже если этот узел находится в оптимальном пути решения*. Следовательно, такой узел может быть отброшен точно так же, как и в том случае, если он не имеет преемников.

Алгоритм SMA* является полным, если существует какое-либо достижимое решение, иными словами, если *d*, глубина самого поверхностного целевого узла, меньше, чем объем памяти (выраженный в хранимых узлах). Этот алгоритм оптимален, если достижимо какое-либо оптимальное решение; в противном случае он возвращает наилучшее достижимое решение. С точки зрения практики алгоритм SMA* вполне можно считать отличным выбором для поиска оптимальных решений, особенно если пространство состояний представляет собой граф, стоимости этапов не одинаковы, а операция формирования узлов является более дорогостоящей в сравнении с дополнительными издержками сопровождения периферии и множества достигнутых узлов.

Однако при решении очень сложных задач часто возникают ситуации, когда алгоритм SMA* вынужден постоянно переключаться с одного пути решения

на другой в пределах множества возможных путей решения, из которых только небольшое их подмножество может поместиться в памяти. (Такая ситуация напоминают проблему ▶**пробуксовки** в системах подкачки страниц с жесткого диска.) В таком случае на повторное формирование одних и тех узлов затрачивается дополнительное время, а это означает, что задачи, которые были бы фактически разрешимыми с помощью поиска A* при наличии неограниченной памяти, становятся трудноразрешимыми для алгоритма SMA*. Иными словами, ▶*из-за ограничений в объеме доступной памяти некоторые задачи могут становиться трудноразрешимыми с точки зрения времени вычисления*. Хотя отсутствует теория, позволяющая найти компромисс между затратами времени и памяти, создается впечатление, что зачастую избежать возникновения этой проблемы невозможно. Единственным способом преодоления такой ситуации становится частичный отказ от требований к оптимальности решения.

3.5.6. Двунаправленный эвристический поиск

Для однонаправленного поиска по первому наилучшему совпадению было показано, что при использовании функции оценки вида $f(n)=g(n)+h(n)$ мы получаем поиск A*, гарантированно дающий оптимальные по затратам решения (при условии допустимости функции f) и одновременно оптимально эффективный по количеству развертываемых узлов.

Для двунаправленного поиска по первому наилучшему совпадению можно было бы также попытаться использовать функцию оценки $f(n)=g(n)+h(n)$, но, к сожалению, нет никаких гарантий, что это приведет к нахождению решения, оптимального по затратами, или что поиск в этом случае будет оптимально эффективным, даже с допустимой эвристикой. При двунаправленном поиске оказывается, что обоснованно выбирать для развертывания каждый раз требуется уже не один узел, а *пару* узлов, по одному из каждой периферии, поэтому в любом доказательстве эффективности алгоритма необходимо рассматривать именно пары узлов (Экерле и др. [673], 2017).

Начнем с введения некоторой новой нотации. Будем использовать функцию оценки $f_F(n)=g_F(n)+h_F(n)$ для узлов, исследуемых в прямом направлении (с корнем дерева поиска в начальном состоянии), и $f_B(n)=g_B(n)+h_B(n)$ для узлов, исследуемых в обратном направлении (с корнем дерева поиска в целевом состоянии). Хотя и прямой, и обратный поиск выполняются для нахождения решения одной и той же задачи, у них будут разные функции оценки, хотя бы потому, что их эвристики, например, будут различаться в зависимости от того, куда необходимо попасть — в целевой узел или в исходный. Мы будем считать, что обе эти эвристики являются допустимыми.

Рассмотрим путь в прямом направлении (от начального состояния к узлу m) и путь в обратном направлении (от целевого состояния к узлу n). Можно определить нижнюю границу стоимости решения, которое включает путь из начального

состояния до узла m, затем каким-то образом попадает в узел n, от которого следует по известному нам пути к целевому узлу, как

$$lb(m, n) = \max(g_F(m) + g_B(n), f_F(m), f_B(n)).$$

Другими словами, стоимость такого пути должна быть по крайней мере не меньше суммы стоимости его двух известных нам частей (поскольку оставшийся участок пути между этими двумя частями должен иметь неотрицательную стоимость), а также эта стоимость пути должна быть по крайней мере не меньше расчетной f-стоимости любой его части (поскольку эвристические оценки оптимистичны). Учитывая сказанное, можно предложить теорему, что для любой пары узлов m, n с $lb(m, n)$ меньше, чем оптимальная стоимость C^*, мы должны развернуть либо узел m, либо узел n, потому что путь, который проходит через оба этих узла, является потенциально оптимальным решением. Трудность здесь состоит в том, что нет четкой уверенности в том, какой именно узел лучше развернуть, а следовательно, нет алгоритма двунаправленного поиска, который будет гарантированно оптимально эффективным, — любой алгоритм может развернуть до удвоенного минимально необходимого количества узлов, если постоянно будет ошибаться в выборе члена пары, развертываемого в первую очередь. Некоторые алгоритмы двунаправленного эвристического поиска специальным образом управляют очередью из пар (m, n), но мы будем придерживаться алгоритма двунаправленного жадного поиска по первому наилучшему совпадению (рис. 3.14), предусматривающего две очереди с приоритетом для каждой из периферий, и дадим ему функцию оценки, имитирующую lb-критерий:

$$f_2(n) = \max(2g(n), g(n) + h(n)).$$

Следующим всегда будет развертываться узел, минимизирующий значение функции f_2, причем этот узел может быть взят из любой периферии. Такая функция f_2 гарантирует, что мы никогда не развернем узел (из любой периферии) со значением функции $g(n) > C^*/2$. Можно сказать, что две половины поиска "встречаются посередине" в том смысле, что, когда две периферии соприкасаются, ни один узел внутри любой периферии не имеет стоимости пути, превышающей ограничение $C^*/2$. На рис. 3.24 показан пример выполнения двунаправленного поиска.

Выше описан подход, когда эвристика h_F оценивает расстояние до цели (если задача имеет несколько целевых состояний, то до ближайшей цели), а эвристика h_B оценивает расстояние до начального состояния. Такой подход называется поиском ▶**спереди назад** (front-to-end). В альтернативном варианте, называемом поиском ▶**спереди на передний план** (front-to-front), предпринимается попытка оценить расстояние до другой периферии. Очевидно, что если периферия имеет миллионы узлов, будет неэффективно применять эвристическую функцию к каждому из них, а затем выбирать минимальное значение. Но этот подход может сработать в применении к *выборке* из нескольких узлов периферии. В некоторых специфических проблемных областях вполне возможно так или иначе *обобщить*

характеристики периферии, например в клеточном мире можно пошагово вычислять для периферии ограничительную рамку, а затем использовать в качестве эвристики расстояния от узла до этой рамки.

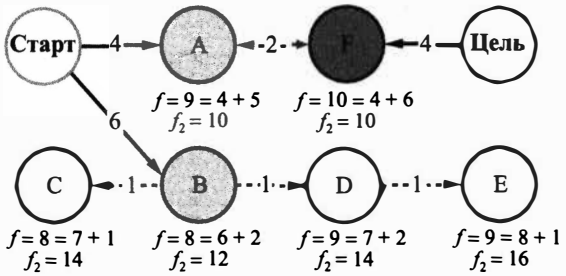


Рис. 3.24. Двунаправленный поиск поддерживает две периферии: слева узлы A и B являются преемниками состояния Старт, а справа узел F является обратным преемником состояния Цель. Для каждого узла указаны значения функции $f = g + h$ и функции $f_2 = \max(2g, g + h)$. (Значения g представляют собой сумму стоимостей действий, указанных для каждой стрелки, тогда как значения h являются произвольными и не могут быть выведены из чего-либо, приведенного на рисунке.) Оптимальное решение Старт–A–F–Цель имеет стоимость $C^* = 4 + 2 + 4 = 10$, и это означает, что двунаправленный алгоритм до встречи посередине не должен развернуть ни один узел с $g > C^*/2 = 5$. И действительно, следующим узлом, который будет развернут, может быть A или F (каждый с $g = 4$), что ведет к оптимальному решению. Если же первым развертывать узел с наименьшим значением функции стоимости f, то первыми на очереди будут узлы B и C, а узлы D и E окажутся связанными с A, но все они имеют значение функции $g > C^*/2$ и, следовательно, никогда не будут развернуты, если в качестве функции оценки будет использоваться f_2

Двунаправленный поиск в одних ситуациях эффективнее однонаправленного, а в других — нет. В общем случае, если используется очень хорошая эвристика, в ходе поиска A* создаются контуры поиска, хорошо ориентированные на цель, и добавление двунаправленного поиска дает незначительные результаты. При среднем качестве эвристики двунаправленный поиск, обеспечивающий встречу посередине пути, имеет тенденцию развертывать меньшее количество узлов и является предпочтительным. В худшем случае плохой эвристики контуры поиска уже не сфокусированы на цели, а двунаправленный поиск имеет ту же асимптотическую сложность, что и простой поиск A*. Однако в любом случае двунаправленный поиск с функцией оценки f_2 и допустимой эвристикой h является полным и оптимальным.

3.6. Эвристические функции

В этом разделе анализируется, как точность эвристики влияет на производительность поиска, а также рассматривается вопрос о том, как вообще можно придумать эвристику. В качестве главного примера будет использоваться головоломка ***Восемь***, уже упоминавшаяся в разделе 3.2. Эта головоломка была одной из первых задач эвристического поиска. В ходе ее решения требуется передвигать плитки по горизонтали или по вертикали на пустой участок до тех пор, пока полученная конфигурация будет соответствовать целевой конфигурации, представленной на рис. 3.25.

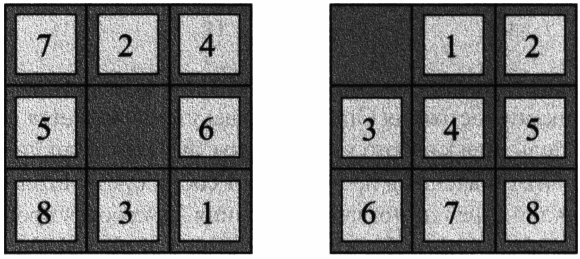


Рис. 3.25. Типичный пример головоломки ***Восемь***. Самое короткое решение насчитывает 26 действий

Для этой головоломки существует 9!/2 = 181 400 достижимых состояний, поэтому в процессе поиска все они легко могут быть размещены в памяти компьютера. Однако для ее старшей сестры, головоломки ***Пятнадцать***, число допустимых состояний составит уже 16!/2 — более 10 триллионов, а значит, для поиска решения в этом пространстве состояний требуется использовать хорошую допустимую эвристическую функцию. Существует долгая история поиска таких эвристик. Вот два чаще всего используемых кандидата.

- h_1 = количество плиток, стоящих не на своем месте (пустое место не включается). На рис. 3.25 все восемь плиток стоят не на своих местах, поэтому показанное слева начальное состояние имеет эвристическую оценку $h_1 = 8$. Эвристическая функция h_1 является допустимой, поскольку очевидно, что каждую плитку, находящуюся не на своем месте, необходимо переместить по меньшей мере один раз.
- h_2 = сумма расстояний всех плиток от их целевых позиций. Поскольку плитки не могут передвигаться по диагонали, это расстояние представляет собой сумму горизонтальных и вертикальных расстояний. Такое расстояние иногда называют ▶**расстоянием, измеряемым в городских кварталах** или

▶**манхэттенским расстоянием**. Эвристическая функция h_2 также является допустимой, поскольку все, что может быть сделано в одном ходе, заключается лишь в перемещении одной плитки на один этап ближе к цели. Все плитки в начальном состоянии, представленном на рис. 3.25, дают следующее значение манхэттенского расстояния:

$$h_2 = 3+1+2+2+2+3+3+2 = 18.$$

Как и следовало ожидать, ни одна из этих функций не переоценивает истинную стоимость решения, равную 26.

3.6.1. Влияние точности эвристической функции на производительность поиска

Одним из способов охарактеризовать качество эвристической функции является определение ее ▶**эффективного коэффициента ветвления** b^*. Если общее количество узлов, формируемых алгоритмом поиска A* в процессе решения некоторой задачи, равно N, а глубина решения равна d, то b^* представляет собой коэффициент ветвления, который должно иметь однородное дерево глубиной d для того, чтобы в нем содержалось $N+1$ узлов.

$$N+1 = 1+b^*+(b^*)^2+\ldots+(b^*)^d$$

Например, если алгоритм A* находит решение на глубине 5, используя 52 узла, то его эффективный коэффициент ветвления равен 1,92. Эффективный коэффициент ветвления может изменяться от одного экземпляра одной и той же задачи к другому, но для определенного класса задач (подобных головоломке ***Восемь***) остается относительно постоянным для всех ее нетривиальных вариантов. Поэтому экспериментальные измерения коэффициента b^* на небольшом множестве различных экземпляров задачи могут служить хорошим критерием общей полезности рассматриваемой эвристической функции. Хорошо спроектированная эвристическая функция должна иметь значение b^*, близкое к 1, что позволит решать довольно большие задачи с разумными вычислительными издержками.

В своей работе Корф и Рейд ([1292], 1998) утверждают, что лучший способ охарактеризовать эффект обрезки, выполняемой алгоритмом поиска A* с заданной эвристикой h, состоит в том, что он уменьшает ▶**эффективную глубину** на постоянную k_h в сравнении с истинной глубиной. Это означает, что общая стоимость поиска составит $O(b^{d-k_h})$ в сравнении с $O(b^d)$ для неинформированного поиска. Их эксперименты с кубиком Рубика и головоломками ***N-плиток*** показывают, что эта формула давала точные прогнозы общей стоимости поиска для выборки вариантов задач в широком диапазоне длины решений, по крайней мере для длины решений, большей k_h.

На рис. 3.26 представлены результаты экспериментов со сгенерированными случайным образом вариантами головоломки ***Восемь***. Эти задачи решались с использованием трех алгоритмов: неинформированного жадного поиска по первому

наилучшему совпадению (BFS) и информированного поиска A* с эвристиками h_1 и h_2. Приведены значения стоимости поиска, выраженной в количестве сформированных узлов, и эффективного коэффициента ветвления для решений с разной глубиной d. Полученные результаты свидетельствуют, что эвристика h_2 дает лучшие результаты в сравнении с h_1, и обе они превосходят неинформированный поиск.

	Стоимость поиска			Эффективный коэффициент ветвления		
d	BFS	A*(h_1)	A*(h_2)	BFS	A*(h_1)	A*(h_2)
6	128	24	24	2,01	1,42	1,34
8	368	48	48	1,91	1,40	1,30
10	1033	116	116	1,85	1,43	1,27
12	2672	279	279	1,80	1,45	1,28
14	6783	678	678	1,77	1,47	1,31
16	17270	1683	1683	1,74	1,48	1,32
18	41558	4102	4102	1,72	1,49	1,34
20	91493	9905	9905	1,69	1,50	1,34
22	175921	22955	22955	1,66	1,50	1,34
24	290082	53039	53039	1,62	1,50	1,36
26	395355	110372	110372	1,58	1,50	1,35
28	463234	202565	202565	1,53	1,49	1,36

Рис. 3.26. Сравнение стоимости поиска (в сформированных узлах) и эффективных коэффициентов ветвления для нескольких вариантов задач головоломки ***Восемь***, решаемых с использованием жадного поиска по первому наилучшему совпадению и поиска A* с эвристиками h_1 (количество плиток не на месте) и h_2 (манхэттенское расстояние). Данные усреднены по результатам выполнения более 100 примеров головоломки для каждой глубины решения от 6 до 28

Интерес представляет вопрос, всегда ли эвристическая функция h_2 лучше, чем h_1. Ответ — "Как правило, да". На основании определений этих двух эвристических функций можно легко прийти к выводу, что для любого узла n будет справедливо выражение $h_2(n) \geq h_1(n)$. Следовательно, можно сказать, что эвристика h_2 ▶**доминирует** над h_1. Доминирование напрямую связано с эффективностью: при поиске A* с использованием эвристики h_2 никогда не происходит развертывание большего количества узлов, чем при поиске A* с использованием эвристики h_1 (возможно, за исключением нескольких случаев неудачного расположения плиток). Доказать это утверждение несложно. Напомним приведенное в разделе 3.5.3 замечание, что каждый узел со значением $f(n) < C^*$ наверняка будет развернут. Это аналогично утверждению, что наверняка будет развернут каждый узел со значением $h(n) < C^* - g(n)$. Но поскольку для всех узлов значение h_2 по крайней мере не меньше значения h_1, то каждый узел, который должен быть наверняка развернут в поиске A* с эвристикой h_2, также будет наверняка развернут при поиске с эвристикой h_1, а применение эвристической функции h_1 может дополнительно вызвать

развертывание и других узлов. Поэтому всегда лучше использовать эвристическую функцию с более высокими значениями, при главном условии, что эта функция *не переоценивает* длину пути решения и что время вычисления этой эвристической функции не слишком велико.

3.6.2. Генерация эвристик из ослабленных задач

Выше было показано, что эвристические функции h_1 (количество плиток не на месте) и h_2 (манхэттенское расстояние) являются довольно неплохими эвристиками для задач головоломки ***Восемь*** и что из них функция h_2 — лучшая. Но на основании чего была предложена функция h_2? Возможно ли, чтобы компьютер мог создать подобную эвристическую функцию механически?

Эвристические функции h_1 и h_2 представляют собой оценки оставшейся длины пути для задач головоломки ***Восемь***, но они, кроме того, представляют собой совершенно точные значения длины пути для *упрощенных* версий этой головоломки. Если правила головоломки ***Восемь*** изменить таким образом, чтобы любую плитку можно было передвигать куда угодно, а не только на соседний пустой квадрат, то эвристическая функция h_1 возвращала бы точное количество этапов в кратчайшем решении. Аналогичным образом, если бы любую плитку можно было перемещать на один квадрат в любом направлении, даже на занятый квадрат, то точное количество этапов в кратчайшем решении возвращала бы эвристическая функция h_2. Задача с меньшим количеством ограничений на возможные действия называется ▶**ослабленной задачей**. Граф пространства состояний ослабленной задачи является *суперграфом* исходного пространства состояний, поскольку снятие ограничений создает дополнительные ребра в исходном графе.

Поскольку ослабленная задача добавляет ребра в граф пространства состояний, любое оптимальное решение в исходной задаче по определению также будет решением в ослабленной задаче, но последняя также может иметь *лучшие* решения, если добавленные ребра обеспечивают сокращенные пути. Следовательно, ◀ *стоимость оптимального решения ослабленной задачи является допустимой эвристикой для исходной задачи*. Более того, поскольку производная эвристика является точной стоимостью для ослабленной задачи, она должна подчиняться неравенству треугольника и поэтому является преемственной (см. раздел 3.5.2).

Стоимость оптимального решения ослабленной задачи представляет собой допустимую эвристику для первоначальной задачи. Такая эвристическая функция является допустимой, поскольку оптимальное решение первоначальной задачи по определению служит также решением ослабленной задачи и поэтому должно быть по меньшей мере таким же дорогостоящим, как и оптимальное решение ослабленной задачи. Поскольку значение производной эвристической функции представляет собой точную стоимость решения ослабленной задачи, эта функция должна подчиняться неравенству треугольника и поэтому должна быть преемственной (см. раздел 3.5.2).

Если определение задачи записано на формальном языке, то существует возможность создавать ослабленные задачи автоматически.[14] Например, если действия в головоломке ***Восемь*** описать как

> Плитку можно переместить из квадрата X в квадрат Y, если
> квадрат X смежный с Y и квадрат Y пуст,

то могут быть созданы три ослабленные задачи путем удаления одного или обоих приведенных выше условий.

а) Плитку можно переместить из квадрата X в квадрат Y, если квадрат X смежный с Y.

б) Плитку можно переместить из квадрата X в квадрат Y, если квадрат Y пуст.

в) Плитку можно переместить из квадрата X в квадрат Y.

Из ослабленной задачи *а* можно вывести функцию h_2 (манхэттенское расстояние). Обоснование этого утверждения состоит в том, что эвристика h_2 должна представлять собой правильную оценку, если каждая плитка передвигается к месту своего назначения по очереди. Эвристическая функция, полученная на основании ослабленной задачи *б*, обсуждается в упр. 3.39. На основании ослабленной задачи *в* можно получить эвристическую функцию h_1 (плитки не на своих местах), поскольку эта оценка была бы правильной, если бы плитки можно было передвигать в предназначенное для них место за один этап. Обратите внимание: крайне важно, чтобы ослабленные задачи, создаваемые этим методом, можно было решить, по существу, без поиска. В данном примере это требование соблюдается, поскольку ослабленные правила позволяют разложить исходную проблему на восемь независимых подзадач. Если ослабленную задачу решить сложно, то получать значения соответствующей эвристики будет дорого.

Программа под названием ABSOLVER позволяет автоматически генерировать эвристики из определения задачи, используя метод "ослабленной задачи" и различные другие методы ([1823], 1993). Программа ABSOLVER создала новую эвристику для головоломки ***Восемь***, которая оказалась лучше, чем любая уже существовавшая до этого, и нашла первую полезную эвристику для знаменитого кубика Рубика.

Если для решения какой-либо задачи может использоваться целая коллекция допустимых эвристических функций $h_1 \ldots h_m$ и ни одна из них явным образом не превосходит другие, то какая из этих функций должна быть выбрана? Оказалось, что такой выбор делать не требуется, поскольку можно взять от них всех самое лучшее, определив такой критерий выбора:

$$h(n) = \max\{h_1(n), \ldots, h_k(n)\}.$$

---

[14] В главах 8 и 11 описываются формальные языки, подходящие для этой цели. С помощью формальных описаний, которыми можно манипулировать, процедуру создания ослабленных задач можно автоматизировать. В данном случае используется русский язык.

Эта составная эвристика выбирает какую угодно функцию, являющуюся наиболее точной в рассматриваемом узле. Поскольку компоненты h_i допустимы, функция h также допустима (и если все h_i преемственны, функция h также преемственна). Более того, функция h будет доминировать над всеми образующими ее эвристиками. Единственный недостаток в этом случае состоит в том, что получение значений $h(n)$ требует больше времени для вычислений. Если это является проблемой, альтернативой может быть случайный выбор одной из эвристик при каждой оценке или использование машинного обучения алгоритма, чтобы научить его предсказывать, какая из эвристик будет лучше в каждом конкретном случае. Такой подход может привести к получению эвристики, которая окажется непреемственной (даже если каждая из h_i является преемственной), но на практике это обычно приводит к более быстрому решению задач.

3.6.3. Генерация эвристик из подзадач: базы данных с шаблонами

Допустимые эвристические функции могут быть также получены на основании стоимости решения подзадачи данной задачи. Например, на рис. 3.27 представлена подзадача для примера головоломки ***Восемь***, приведенного на рис. 3.25. Эта подзадача предусматривает перемещение плиток 1, 2, 3, 4 в их правильные позиции. Очевидно, что стоимость оптимального решения этой подзадачи представляет собой нижнюю границу стоимости решения полной задачи. Как оказалось, такая оценка в некоторых случаях является намного более точной по сравнению с манхэттенским расстоянием.

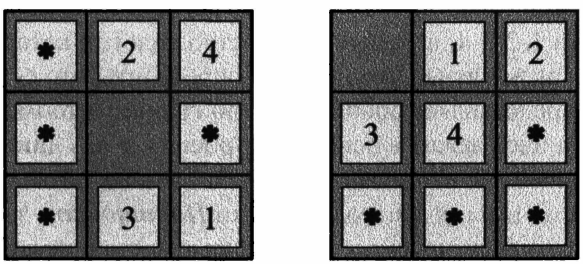


Рис. 3.27. Подзадача для примера головоломки ***Восемь***, показанного на рис. 3.25. Задание заключается в том, чтобы передвинуть плитки 1, 2, 3 и 4 в их правильные позиции, не беспокоясь о том, что произойдет с другими плитками

Идея ▶**баз данных с шаблонами** состоит в том, чтобы хранить эти точные стоимости решения для каждого возможного экземпляра подзадачи; в нашем примере — каждой возможной конфигурации из четырех плиток и пустой ячейки.

(В базе данных будет $9 \times 8 \times 7 \times 6 \times 5 = 15\,120$ шаблонов. Расположение остальных четырех плиток не нужно принимать во внимание, но ходы с этими плитками следует учитывать в стоимости решения.) После этого допустимая эвристическая функция h_{DB} для каждого состояния головоломки, встретившегося в процессе поиска, вычисляется путем выборки данных для соответствующей конфигурации подзадачи из базы данных. Сама база данных создается путем поиска от целевого состояния и записи в нее стоимости каждого нового обнаруженного шаблона.[15] Затраты на этот поиск будут погашены за счет решения последующих экземпляров задачи, и поэтому создавать базу данных с шаблонами имеет смысл, только если ожидается, что такая потребность возникнет.

Выбор плиток 1-2-3-4 и пустой ячейки является довольно произвольным; можно было бы также создать базы данных для плиток 5-6-7-8, 2-4-6-8 и т.д. Каждая такая база данных обеспечивает допустимую эвристику, и все эти эвристики можно объединить в общую эвристическую функцию, как предлагалось ранее, принимая максимальное значение. Составная эвристическая функция такого вида является намного более точной в сравнении с манхэттенским расстоянием: количество узлов, формируемых при решении образованных случайным образом экземпляров задачи головоломки ***Пятнадцать***, может быть уменьшено примерно в 1000 раз. Однако с каждой дополнительной базой данных уменьшается отдача и увеличиваются затраты памяти и времени вычислений.

Довольно интересен вопрос, можно ли *сложить* эвристики, полученные из баз данных 1-2-3-4 и 5-6-7-8, так как две эти подзадачи, похоже, не перекрываются. Будет ли это по-прежнему допустимая эвристика? Ответ — нет, поскольку решения подзадачи 1-2-3-4 и подзадачи 5-6-7-8 для данного состояния почти наверняка разделят некоторые этапы, ведь маловероятно, что плитки 1-2-3-4 можно перемещать на их место, не трогая плиток 5-6-7-8, и наоборот. Но что будет, если не учитывать эти ходы? Иными словами, допустим, что регистрируется не общая стоимость решения подзадачи 1-2-3-4, а только количество ходов, в которых затрагиваются плитки 1-2-3-4. В таком случае можно легко определить, что сумма этих двух стоимостей все еще представляет собой нижнюю границу стоимости решения всей задачи. Именно эта идея лежит в основе построения ▶**баз данных с непересекающимися шаблонами**. Использование таких баз данных позволяет решать случайно выбранные экземпляры задачи головоломки ***Пятнадцать*** за несколько миллисекунд — количество формируемых узлов сокращается примерно в 10 000 раз по сравнению с использованием манхэттенского расстояния. А для задачи головоломки ***Двадцать четыре*** может быть достигнуто ускорение приблизительно в миллион раз. Базы данных с непересекающимися шаблонами так хорошо работают для задач головоломок с перемещающимися плитками потому, что исходная задача может быть разделена таким образом, что каждый этап будет затрагивать только одну подзадачу, поскольку на каждом этапе перемещается только одна плитка.

---

[15] Работая в обратном направлении от целевого состояния, можно сразу получить точную стоимость решения для каждого обнаруженного экземпляра. Это пример динамического программирования, речь о котором пойдет в главе 17.

3.6.4. Создание эвристик с ориентирами

В Сети есть интерактивные сервисы, содержащие географические карты с десятками миллионов пунктов и позволяющие найти оптимальные по издержкам маршруты поездки между ними за миллисекунды (рис. 3.28). Как им удается это делать, если даже лучшие алгоритмы поиска, рассмотренные нами выше, работают примерно в миллион раз медленнее? Для достижения таких результатов используется много различных приемов, но наиболее важным из них является ▶**предвычисление** стоимостей некоторых оптимальных путей. Хотя предвычисление может потребовать значительных затрат машинного времени, достаточно сделать это только один раз, а затем компенсировать издержки за счет миллиардов поисковых запросов пользователей.

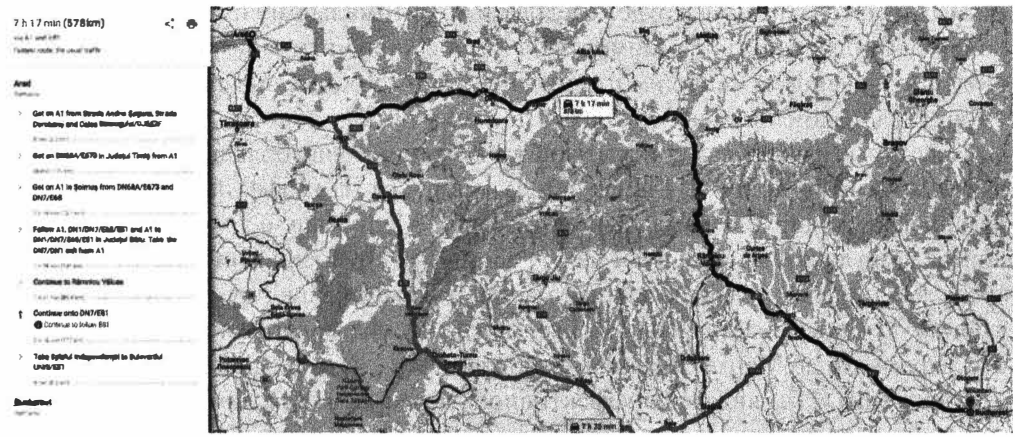


Рис. 3.28. Веб-сервис, предоставляющий маршруты проезда, рассчитанные с использованием алгоритмов поиска

Можно разработать более совершенные эвристики за счет предвычисления и сохранения стоимости оптимальных путей между каждой парой пунктов. Это потребует пространственных затрат порядка $O(|V|^2)$ и временных затрат порядка $O(|E|^3)$, что будет приемлемо для графов с 10 тысячами вершин, но не с 10 миллионами.

Лучший подход — выбрать из всех вершин графа несколько (возможно, 10 или 20) ▶**опорных точек**,[16] а затем для каждой опорной точки L и каждой другой вершины v в графе вычислить и сохранить $C^*(v, L)$ — точную стоимость оптимального пути от v до L. (Также может потребоваться рассчитать значение $C^*(L, v)$, поскольку только на неориентированном графе оно такое же, как и $C^*(v, L)^*$, тогда как на ориентированном графе — например, при наличии улиц с односторонним движением — это

[16] Опорные точки иногда называют ориентирами или якорями.

значение потребуется вычислять отдельно). При наличии подобных сохраняемых таблиц C^* можно легко создать эффективную (хотя и недопустимую) эвристику: стоимость пути от исходного узла до той опорной точки L, для которой это значение минимально среди всего множества опорных точек $\{L\}$, плюс стоимость пути от этой опорной точки до цели *goal*.

$$h_L(n) = \min_{L \in \{L\}} C^*(n, L) + C^*(L, goal)$$

Если оптимальный путь проходит через опорную точку, эта эвристика будет точной; а если нет, то она будет недопустимой, поскольку переоценивает стоимость цели. При использовании поиска A*, если имеются точные эвристики, как только поиск достигнет узла, находящегося на оптимальном пути, все последующие развертываемые узлы также будут находиться на оптимальном пути. Представьте, что линии контуров поиска будут расширяться непосредственно вдоль оптимального пути. Поиск будет следовать по оптимальному пути, на каждой итерации добавляя действие стоимостью c, чтобы получить результирующее состояние с h-значением, меньшим c, означающим, что общее значение $f = g + h$ будет оставаться неизменным на уровне C^* по всему пути.

Некоторые алгоритмы поиска маршрута экономят еще больше времени, добавляя ▶**кратчайшие пути** — искусственные ребра на графе, определяющие оптимальный маршрут из нескольких действий. Например, если имеются заранее определенные кратчайшие пути между всеми 100 крупнейшими городами США и требуется определить оптимальный маршрут из кампуса в городе Беркли в Калифорнии до университета в Нью-Йорке, можно воспользоваться кратчайшим путем от Сакраменто до Манхеттена и этим покрыть 90% пути в одном действии.

Эвристика $h_L(n)$ является эффективной, но не является допустимой. Однако если чуть постараться, то можно придумать эвристику, которая будет одновременно и эффективной, и допустимой:

$$h_{DL}(n) = \min_{L \in \{L\}} |C^*(n, L) - C^*(goal, L)|.$$

Этот вариант называется ▶**разностной эвристикой** (из-за вычитания). Представьте себе это как ситуацию, когда опорная точка находится *за* целью. Если повезет и цель окажется на оптимальном пути от n к опорной точке, то это выражение означает "рассматриваем весь путь от n до L, а затем вычитаем из него конечный участок от цели к L, что в результате дает точную стоимость пути от n до цели". В случае, когда цель находится немного в стороне от оптимального пути к опорной точке, эвристика будет неточной, но по-прежнему допустимой. Все опорные точки, которые находятся в пределах контура поиска цели, в данном случае будут бесполезны. Так, опорная точка, расположенная точно на полпути до цели, даст значение $h_{DL} = 0$, не имеющее никакого смысла.

Существует несколько способов выбора опорных точек. Выбор точек случайным образом выполняется быстро, но можно получить лучшие результаты, если постараться распределить опорные точки так, чтобы они находились не слишком

близко одна к другой. "Жадный" подход состоит в том, чтобы выбрать первую опорную точку наугад, а затем найти точку, наиболее удаленную от нее, добавить ее к множеству опорных точек и продолжать действовать, на каждой итерации добавляя точку с максимальным расстоянием до ближайшей опорной точки. Если имеется журнал прошлых поисковых запросов пользователей, то в качестве опорных точек можно выбрать пункты, чаще всего упоминаемые в этих запросах. При использовании разностной эвристики имеет смысл расположить опорные точки по периметру графа. Следовательно, в этом случае полезным решением будет найти центральный узел графа и разделить весь граф на k секторов, сходящихся в центральном узле, а затем в каждом секторе выбрать опорную точку, максимально удаленную от центра.

Метод опорных точек особенно хорошо работает для задач поиска маршрута также благодаря способу, которым в реальном мире прокладывается большинство дорог. Большая часть дорожного трафика имеет место между определенными пунктами, и инженеры дорожных служб прокладывают самые широкие и скоростные трассы именно между ними. Если выбрать эти пункты в качестве опорных точек, то инструменты поиска маршрутов, использующие эти точки, будут направлять пользователей именно на эти скоростные трассы.

3.6.5. Учимся искать лучше

Выше было представлено несколько стратегий поиска (поиск в ширину, жадный поиск по первому наилучшему совпадению и т.д.), которые были разработаны учеными и специалистами по компьютерным наукам. Но может ли сам агент *обучаться* лучшим способам поиска? Ответ на этот вопрос является положительным, а применяемый метод обучения опирается на важную концепцию, называемую ▶**метауровневым пространством состояний**. Каждое состояние в метауровневом пространстве состояний отражает внутреннее (вычислительное) состояние программы, выполняющей поиск в обычном пространстве состояний, таком как карта Румынии. (Чтобы различать эти две концепции, мы будем называть карту Румынии ▶**пространством состояний объектного уровня**.) Например, внутреннее состояние алгоритма поиска A* включает в себя текущее дерево поиска. Каждое действие в метауровневом пространстве состояний представляет собой этап вычислений, изменяющий внутреннее состояние, например на каждом этапе вычислений в процессе поиска A* развертывается один из листовых узлов, а его преемники добавляются к дереву. Поэтому рис. 3.18, на котором представлена последовательность все больших и больших деревьев поиска, может рассматриваться как отображение пути в метауровневом пространстве состояний, где каждое состояние в пути является объектно-уровневым деревом поиска.

Итак, путь, показанный на рис. 3.18, содержит пять этапов, включая один этап (развертывание узла *Fagaras*), который нельзя считать особенно полезным. Для более сложных задач можно ожидать, что количество подобных ненужных этапов

будет намного больше, и алгоритм ▶**метауровневого обучения** может изучать этот опыт, чтобы в дальнейшем научиться избегать исследования бесперспективных поддеревьев. Методы, используемые при обучении такого рода, описаны в главе 22. Целью обучения являются минимизация суммарной стоимости решения задач, а также поиск компромисса между вычислительными издержками и стоимостью пути.

3.6.6. Изучение эвристик на основе опыта

Выше уже было показано, что один из способов построения эвристики состоит в разработке ослабленной задачи, найти оптимальное решение которой будет несложно. Альтернативным подходом является изучение полученного опыта. Под словом "опыт" здесь понимается, например, решение большого количества экземпляров головоломки ***Восемь***. Каждое оптимальное решение задачи головоломки ***Восемь*** предоставляет отдельный пример — цель и путь. На основе данных примеров с помощью алгоритма обучения может быть сформирована некоторая функция $h(n)$, способная (если повезет) предсказывать истинные стоимости решений для других состояний, которые возникают во время поиска. В большинстве таких случаев обучение дает несовершенное приближение к эвристической функции, и следовательно, есть риск ее недопустимости. В результате неизбежен компромисс между временем обучения, временем выполнения поиска и стоимостью решения. Методы машинного обучения обсуждаются в главе 19. Методы обучения с подкреплением, описываемые в главе 22, также применимы к поиску.

Некоторые методы машинного обучения работают лучше, когда в них учитываются ▶**характеристики** состояния, релевантные для оценки эвристической стоимости этого состояния, а не просто его общее описание. Например, в головоломке ***Восемь*** характеристика "количество плиток не на своем месте" может быть полезной при предсказании фактического удаления некоторого состояния от цели. Назовем эту характеристику $x_1(n)$. Можно взять 100 сформированных случайным образом конфигураций головоломки ***Восемь*** и собрать статистические данные об их фактических стоимостях решений. Допустим, это позволит обнаружить, что при $x_1(n)$, равном 5, средняя стоимость решения будет около 14, и т.д. Безусловно, можно использовать сразу несколько характеристик. Второй характеристикой, $x_2(n)$, может быть "количество пар смежных плиток, которые не являются смежными в целевом состоянии". Каким образом можно скомбинировать значения $x_1(n)$ и $x_2(n)$ для предсказания значения $h(n)$? Общепринятый подход состоит в использовании линейной комбинации, как показано ниже.

$$h(n) = c_1 x_1(n) + c_2 x_2(n)$$

Константы c_1 и c_2 корректируются для достижения наилучшего соответствия фактическим данным о стоимостях решений для случайным образом сгенерированных конфигураций задачи. Можно ожидать, что константы c_1 и c_2 будут положительными, поскольку плитки не на месте и неправильные соседние пары

затрудняют решение задачи. Обратите внимание, что эта эвристика удовлетворяет условию $h(n) = 0$ для целевых состояний, но не обязательно является допустимой или преемственной.

Резюме

В настоящей главе представлены поисковые алгоритмы, которые агент может использовать для выбора последовательности действий в самых различных вариантах среды, — при условии, что она является эпизодической, одноагентной, полностью наблюдаемой, детерминированной, статической, дискретной и полностью известной. Необходимо принять определенный компромисс между количеством времени, которое потребуется для выполнения поиска, количеством доступной памяти и качеством найденного решения. Действия будут более эффективными при наличии определенных знаний в отношении окружения, представленных в виде функции эвристики, позволяющей оценить, насколько далеко целевое состояние от текущего, или если будут проведены предвычисления частичных решений, включая шаблоны или опорные точки.

- Прежде чем агент сможет начать поиск, необходимо сформулировать четко определенную **задачу**.
- Формулировка задачи включает пять частей: **начальное состояние**, набор **действий**, **функция определения приемника**, описывающая результаты этих действий, набор **целевых состояний** и **функция стоимости пути**.
- Среда задачи представляется в виде **графа пространства состояний**. **Путь** через пространство состояний (последовательность действий) от начального состояния к целевому состоянию является **решением**.
- Алгоритмы поиска в общем случае рассматривают состояния и действия как **атомарные**, не имеющие какой-либо структуры (хотя позднее нами будут введены функции состояний, когда придет время заняться обучением).
- Алгоритмы поиска оцениваются на основе **полноты**, **оптимальности по затратам**, **временной сложности** и **пространственной сложности**.
- Методы **неинформированного поиска** имеют доступ только к определению задачи. Алгоритмы строят дерево поиска в попытке найти решение. Алгоритмы различаются в зависимости от того, какой узел они развертывают первым.
  - При поиске **по первому наилучшему совпадению** узлы для развертывания выбираются с использованием **функции оценки**.
  - При **поиске в ширину** для развертывания выбирается самый поверхностный неразвернутый узел, поиск является полным, оптимальным при единичных стоимостях этапов, но имеет экспоненциальную пространственную сложность.

- Поиск **по критерию стоимости** первым развертывает узел с самой низкой стоимостью пути, $g(n)$ и является оптимальным по затратам.
- При **поиске в глубину** для развертывания выбирается самый глубокий неразвернутый узел. Этот поиск не является ни полным, ни оптимальным, но имеет линейную пространственную сложность. В **поиске с ограничением глубины** дополнительно устанавливается предел глубины дерева поиска.
- При **поиске с итеративным углублением** циклически вызывается поиск в глубину с возрастающим пределом глубины — до тех пор, пока цель не будет найдена. Этот поиск будет полным при выполнении полного цикла проверок и оптимальным — при единичных стоимостях этапов. Он характеризуется временной сложностью, сравнимой с поиском в ширину, и имеет линейную пространственную сложность.
- **Двунаправленный поиск** расширяет две периферии: одну — от начального состояния, и одну — от целевого состояния; он завершается, когда эти две периферии встречаются.

- **Методы информированного поиска** имеют доступ к функции **эвристики** $h(n)$, оценивающей стоимость решения от узла n. Они также могут иметь доступ к дополнительной информации, такой как базы данных с шаблонами, для которых известны стоимости решения.
  - При **жадном поиске по первому наилучшему совпадению** развертываются узлы с минимальным значением $h(n)$. Этот поиск не оптимален, но часто бывает эффективным.
  - При **поиске A*** развертываются узлы с минимальным значением функции $f(n) = g(n) + h(n)$. Алгоритм A* является полным и оптимальным при условии, что функция $h(n)$ допустима. Пространственная сложность алгоритма A* все еще остается слишком высокой.
  - **Двунаправленный поиск A*** иногда бывает эффективнее поиска A*.
  - Алгоритм **IDA*** (поиск A* с итеративным углублением) является итеративно углубляющейся версией поиска A*, а значит, ориентирован, прежде всего, на решение проблем с пространственной сложностью.
  - Алгоритмы **RBFS** (рекурсивный поиск по первому наилучшему совпадению) и **SMA*** (упрощенный поиск A* с ограничением памяти) являются надежными, оптимальными алгоритмами поиска, использующими ограниченный объем памяти; при достаточном количестве времени они могут решить проблемы, при решении которых поиск A* останавливается из-за нехватки памяти.
  - Алгоритм **поиска по лучу** накладывает ограничение на размер периферии, что делает его неполным и неоптимальным, но часто находит достаточно хорошие решения и работает быстрее, чем алгоритмы полного поиска.

  - Алгоритм поиска **A* с весовым коэффициентом** фокусирует поиск в направлении цели, развертывая меньше узлов, но при этом жертвуя оптимальностью.
- Производительность алгоритмов эвристического поиска зависит от качества эвристической функции. Иногда хорошие эвристические функции можно составить путем ослабления определения задачи, предварительного вычисления стоимости решения подзадач с сохранением этой информации в базе данных шаблонов, определения опорных точек или обучения на основе опыта решения данного класса задач.

Библиографические и исторические заметки

Тема поиска в пространстве состояний возникла в самый ранний период ИИ. Работа Ньюэлла и Саймона над программами Logic Theorist ([1667], 1957) и GPS ([1668], 1961) привела к принятию алгоритмов поиска в качестве основного инструмента для исследователей искусственного интеллекта 1960-х годов, а сами процессы решения задач стали рассматриваться в качестве канонической проблемы искусственного интеллекта. Работа по исследованию операций, проведенная Ричардом Беллманом ([167], 1957), показала важность аддитивных стоимостей путей в упрощении алгоритмов оптимизации. Статья Нильса Нильссона ([1687], 1971) поставила эту область на твердую теоретическую основу.

Головоломка ***Восемь*** — это "младшая сестра" головоломки ***Пятнадцать***, история которой подробно описана Слокумом и Сонневельдом ([2086], 2006). К 1880 году головоломка ***Пятнадцать*** привлекла широкое внимание общественности и математиков ([1141], 1879 и [2170], 1880). Редакторы *American Journal of Mathematics* заявили: "Головоломка ***Пятнадцать*** в последние несколько недель занимает все внимание американской публики, и можно с уверенностью сказать, что ею интересуются девять из десяти людей всех полов и возрастов и всех общественных положений", а в газете *Weekly News-Democrat*, Эмпория, шт. Канзас, от 12 марта 1880 года писали: "Это стало уже буквально эпидемией по всей стране".

Известный американский разработчик игр Сэм Лойд ложно утверждал, что это он изобрел головоломку ***Пятнадцать*** ([1554], 1959), но на самом деле она была изобретена Нойесом Чепменом, почтмейстером из Канастоты, шт. Нью-Йорк, в середине 1870-х годов (хотя общий патент, охватывающий игры со скользящими плитками, был выдан Эрнесту Кинси в 1878 году). Ратнер и Уормут ([1861], 1986) показали, что общая версия игры (не 15×15, а $n \times n$) принадлежит к классу NP-полных задач.

Кубик Рубика, без сомнений, был изобретен в 1974 году Эрнё Рубиком, который также изобрел для него алгоритм поиска хороших, но не оптимальных решений. Корф ([1291], 1997) нашел оптимальные решения для некоторых случайных задач, используя базы данных с шаблонами и поиск IDA*. Рокицки и соавторы ([1904], 2014) доказали, что любой вариант задачи может быть решен за 26 ходов,

если рассматривать поворот на 180° как два хода, и за 20 ходов, если рассматривать такой поворот как один ход. Доказательство потребовало 35 процессорных лет вычислений и не привело к немедленному получению эффективного алгоритма. Агостинелли и соавторы ([22], 2019) использовали обучение с подкреплением, сети с глубоким обучением и дерево поиска Монте-Карло, чтобы обучить гораздо более эффективный решатель для кубика Рубика. Он не обязательно находит оптимальное по затратам решение, но получает его примерно в 60% случаев, а типичное время нахождения решения у него меньше секунды.

Каждая из реальных задач поиска, перечисленных в данной главе, была предметом значительных усилий исследователей. Методы выбора оптимальных расписаний авиаперелетов по большей части остаются недоступными для общего пользования (закрытыми), но Карл де Маркен (Carl de Marcken) сведением к диофантовым проблемам решения показал, что схемы формирования цен на авиабилеты и связанные с ними ограничения стали настолько запутанными, что задача выбора оптимального маршрута полета является формально *неразрешимой* ([1901], 2002). Задача коммивояжера (Traveling Salesperson Problem — TSP) — это стандартная комбинаторная проблема в теоретических компьютерных науках ([1364], 1992). Карп ([1189], 1972) доказал, что задача TSP является NP-трудной, но для нее были разработаны эффективные методы эвристической аппроксимации ([1415], 1973). Арора ([76], 1998) разработал полностью полиномиальную схему аппроксимации для евклидовых вариантов задачи TSP. Методы компоновки СБИС исследовались Ла-По ([1352], 2010), а в журналах по сверхбольшим интегральным микросхемам появилось много статей, посвященных оптимизации компоновки. Задачи робототехнической навигации обсуждаются в главе 26. Последовательная автоматическая сборка впервые была продемонстрирована роботом FREDDY ([1564], 1972), а всеобъемлющий обзор этой темы был дан в [110] (2016).

Алгоритмы неинформированного поиска являются центральной темой компьютерных наук ([479], 2009) и исследования операций ([655], 1969). Метод поиска в ширину применительно к решению задач с лабиринтами был сформулирован Муром ([1615], 1959). Метод динамического программирования ([169], 1957 и [168], 1962), в котором предусматривается систематическая регистрация решений для всех подзадач с возрастающей длиной, может рассматриваться как форма поиска в ширину в графах.

Алгоритм Дейкстры в том виде, в котором он обычно представляется ([620], 1959), применим к явным конечным графам. Нильссон ([1687], 1971) представил версию алгоритма Дейкстры, которую он назвал *uniform-cost search* (в буквальном переводе — *метод равных цен*, но в русскоязычной литературе его называют *поиском по критерию стоимости*), поскольку алгоритм "распространялся вдоль контуров с равной стоимостью пути"), которую можно было применять к неявно определенным, бесконечным графам. В этой работе Нильссона также были введены понятия открытого (*периферия*) и закрытого (*достигнутые*) списков, а также термин "поиск по графу". Название алгоритма "Поиск по первому наилучшему совпадению" было введено в

Справочнике по искусственному интеллекту ([132], 1981). Алгоритмы Флойда–Уоршелла ([748], 1962) и Беллмана–Форда ([164], 1958 и [755], 1956) допускают отрицательные стоимости действий (если нет отрицательных циклов).

Одна из версий алгоритма поиска с итеративным углублением, предназначенная для эффективного ведения игры в шахматы с контролем времени, была впервые применена Слейтом и Аткином ([2084], 1977) в программе ведения шахматной игры CHESS 4.5. Алгоритм В Мартелли также включал аспект итеративного поиска в глубину. Поиск с итеративным углублением был предложен Бертрамом Рафаэлем ([1851], 1976), но широкое признание получил благодаря работе Корфа ([1283], 1985).

Впервые использование эвристической информации при решении задач упоминается в одной из ранних статей Саймона и Ньюэлла ([2072], 1958), но само выражение "эвристический поиск" и обоснование применения эвристических функций, которые оценивают расстояние до цели, появились немного позже ([1666], 1965 и [1414], 1965). Доран и Мичи ([636], 1966) провели обширные экспериментальные исследования в области эвристического поиска. Хотя они анализировали длину пути и "проникновение" (*penetrance* — отношения длины пути к общему количеству узлов, исследованных к данному моменту) в процессе эвристического поиска, они, видимо, игнорировали информацию, предоставляемую стоимостью пути $g(n)$. Алгоритм поиска A*, включающий текущую стоимость пути в эвристический поиск, был разработан Хартом, Нильссоном и Рафаэлем ([972], 1968), с некоторыми дальнейшими поправками [624]. Декстер и Перл ([581], 1985) изучили условия, при которых алгоритм A* оптимально эффективен (по количеству развернутых узлов).

В оригинальной статье Харта и других об алгоритме поиска A* ([972], 1968) было введено условие преемственности эвристических функций. В качестве его более простой замены Полом ([1803], 1977) было введено условие монотонности, но через несколько лет Перл ([1751], 1984) показал, что эти два условия эквивалентны.

Пол также впервые изучил связь между ошибками в эвристических функциях и временной сложностью алгоритма A* ([1803], 1977). Основные результаты были получены для древовидного поиска с единичной стоимостью действий и одним целевым состоянием ([1803], 1977; [816], 1979; [1107], 1980; [1751], 1984) и с несколькими целевыми состояниями ([623], 2007). Корф и Рейд ([1292], 1998) показали, как предсказать точное количество развернутых узлов (а не просто асимптотическое приближение), в различных актуальных проблемных областях. Понятие "эффективный коэффициент ветвления" было предложено Нильссоном ([1687], 1971) в качестве эмпирической меры эффективности. В отношении поиска в графах Хельмерт и Рёджер ([1008], 2008) отметили, что несколько хорошо известных задач, содержащих экспоненциально возрастающее количество узлов в путях оптимального решения, подразумевают и экспоненциальную временную сложность для алгоритма поиска A*.

Существует много вариантов алгоритма поиска A*. Пол ([1804], 1970) ввел поиск A* с весовым коэффициентом, а затем предложил его динамическую версию ([1902], 1973), в которой весовые коэффициенты изменяются в зависимости от глубины дерева поиска. Эбендт и Дрехслер ([672], 2009) обобщили результаты использования этой версии и рассмотрели некоторые возможные приложения. Хатем и Румл ([983], 2014) предложили упрощенную и улучшенную версию поиска A* с весовыми коэффициентами, которую проще реализовать. Уилт и Румл ([2356], 2014) ввели как альтернативу жадному поиску быстрый поиск, фокусирующийся на сведении к минимуму времени поиска, а затем в работе [2357] (2016) показали, что лучшие эвристики для удовлетворительного поиска отличаются от лучших эвристик для оптимального поиска. В [347] (2012) предлагаются некоторые приемы реализации для написания кода быстрого поиска, а в [723] (2018) рассматривается, как реализация меняется при использовании ранней проверки достижения цели.

Впервые двунаправленный поиск предложил Пол ([1801], 1971), а в [1052] (2016) описывается версия двунаправленного поиска, в которой два направления гарантированно встречаются в середине, обеспечивая ему более широкое применение. В [673] (2017) обсуждается множество обязательно развертываемых пар узлов и показано, что двунаправленный поиск не может быть оптимально эффективным. В алгоритме NBS ([411], 2017) использование очереди пар узлов разработано детально.

Для эффективного поиска маршрутов движения в интерактивном картографическом сервисе Microsoft используется комбинация двунаправленного поиска A* и известных опорных точек ([879], 2006). После кэширования множества путей между опорными точками алгоритм способен найти оптимальный по затратам путь между любой парой точек в Соединенных Штатах, просмотрев менее 0,1% из 24 миллионов таких точек, определенных в графе. Корф ([1285], 1987) показал, как использовать подцели, макрооператоры и абстракцию для достижения значительного ускорения по сравнению с предыдущими методами. В [598] (2009) описывается, как для определения маршрутов движения автотранспорта можно использовать двунаправленный поиск, опорные точки, иерархическую структуру и другие приемы. В [49] (2008) описывается связанный подход, называемый **▶иерархическим поиском**, который может рассматриваться как определение ориентиров на различных иерархических уровнях абстракции. Корф ([1287], 1987) описывает условие, при котором иерархический поиск обеспечивает экспоненциальное ускорение. Кноблок ([1243], 1991) приводит экспериментальные результаты и анализ для количественной оценки преимуществ иерархического поиска.

Алгоритм поиска A* и другие алгоритмы поиска в пространстве состояний тесно связаны с методами **▶ветвей и границ**, широко используемыми в области исследования операций ([1366], 1966 и [1863], 1996). Кумар и Канал ([1327], 1988) предприняли попытку "великой унификации" методов эвристического поиска, динамического программирования, а также методов ветвей и границ под общим названием CDP (Composite Decision Process — комплексный процесс принятия решений).

Поскольку большинство компьютеров в 1960-х годах имели только несколько тысяч слов основной памяти, темой ранних исследовательских работ часто служил эвристический поиск с ограничением памяти. Одна из первых поисковых программ, Graph Traverser ([636], 1966), фиксировала свои результаты после выполнения поиска по первому наилучшему совпадению вплоть до установленного лимита памяти. Алгоритм IDA* ([1284], 1985) стал одним из первых широко применяемых оптимальных алгоритмов эвристического поиска с ограничением памяти, после чего было разработано большое количество его вариантов. Анализ эффективности алгоритма IDA* и сложностей, возникающих при его применении с эвристическими функциями, которые встречаются в реальных задачах, приведен в [1744] (1992).

Оригинальная версия алгоритма RBFS ([1287], 1993) была фактически несколько более сложной, чем алгоритм, представленный на рис. 3.22, который на самом деле ближе к независимо разработанному алгоритму с названием ▶**итеративное развертывание** или IE ([1939], 1992). В алгоритме RBFS используется не только верхняя, но и нижняя граница; эти два алгоритма при использовании допустимых эвристических функций ведут себя одинаково, но RBFS развертывает узлы в порядке первого наилучшего совпадения даже при наличии недопустимой эвристической функции. Идея отслеживания наилучшего альтернативного пути появилась еще раньше в изящной реализации алгоритма A* на языке Prolog, предложенной Братко ([293], 2009), и в алгоритме DTA* ([1947], 1991). В последней работе обсуждаются также метауровневые пространства состояний и метауровневое обучение.

Алгоритм MA* впервые опубликован в [384] (1989). Появление алгоритма SMA*, или упрощенного (Simplified) MA*, стало результатом попытки реализовать MA* ([1939], 1992). Кайндл и Корсанд ([1174], 1994) применили алгоритм SMA* для создания алгоритма двунаправленного поиска, который был значительно быстрее по сравнению с предшествующими алгоритмами. В [1290] (2000) описан подход по принципу "разделяй и властвуй", а Чжоу и Хансен в [2438] (2002) представили алгоритм поиска A* в графе с ограничением памяти и стратегию перехода к поиску в ширину, чтобы повысить эффективность использования памяти в [2439] (2006).

Идея о том, что допустимые эвристические функции могут быть получены с помощью ослабления условий задачи, впервые опубликована в оригинальной статье ([1005], 1970), авторы которой использовали эвристику на основе минимального связующего дерева для решения задачи TSP (см. упр. 3.38). Автоматизация процесса ослабления была успешно реализована Приедитисом ([1823], 1993). Постоянно появляются все новые публикации по применению машинного обучения для обнаружения эвристических функций ([1966], 2008; [67], 2010; [2199], 2011; [1382], 2012).

Использование баз данных шаблонов для получения допустимой эвристики было предложено в [817] (1995); [505] (1996); [506] (1998). Базы данных

непересекающихся шаблонов описаны Корфом и Фельднером ([1289], 2002), а похожий метод с использованием символических шаблонов разработал Эделькамп ([675], 2009). В [724] (2007) показано, как можно сжать базу данных с шаблонами для сохранения пространства. Вероятностная интерпретация эвристик исследовалась Перлом ([1751], 1984), а также Ханссоном и Майером ([963], 1989).

Признанными учебниками по поиску являются *Heuristics* Перла (1984) и *Heuristic Search* Эделькампа и Шрёдля. Статьи о новых алгоритмах поиска публикуются в материалах Международного симпозиума по комбинаторному поиску (International Symposium on Combinatorial Search — SoCS) и Международной конференции по автоматическому планированию и составлению расписаний (International Conference on Automated Planning and Scheduling — ICAPS), а также общих конференций по ИИ, таких как AAAI и IJCAI, и в журналах, таких как *Artificial Intelligence* и *Journal of the ACM.*

Упражнения

3.1. Объясните, почему формулировка задачи должна выполняться после формулировки цели.

3.2. Составьте полную формулировку задачи для каждой из следующих проблем. Подберите формулировку, которая будет достаточно точной для реализации.

а) Шесть стеклянных запертых на замок шкатулок стоят в один ряд. В пяти первых шкатулках лежат ключи, открывающие следующую шкатулку в ряду, а в последней лежит банан. У вас есть ключ к первой шкатулке, и вы хотите банан.

б) Исходно у вас есть последовательность букв ABABAECCEC или вообще любая последовательность, состоящая из букв A, B, C и E. Исходную последовательность можно преобразовать, используя следующие равенства: AC = E, AB = BC, BB = E и Ex = x для любого x. Например, последовательность ABBC может быть преобразована в AEC, затем — в AC, а затем — в E. Ваша цель — из исходной последовательности получить последовательность E.

в) Дана сетка из квадратов размером $n \times n$, каждый из которых изначально представляет собой либо неокрашенный пол, либо бездонную яму. Исходно вы начинаете в квадрате с неокрашенным полом и можете либо покрасить пол в том квадрате, в котором находитесь, либо перейти в любой соседний квадрат с неокрашенным полом. Ваша задача — покрасить весь пол в сетке.

г) В порту находится контейнеровоз, загруженный контейнерами. На нем имеется 13 вертикальных рядов контейнеров, каждый по 13 контейнеров в длину и 5 контейнеров в высоту. Вы управляете краном, который может переместиться в любое место над кораблем, поднять контейнер под собой и доставить его на причал. Требуется полностью разгрузить корабль.

3.3. Ваша цель — вывести робота из лабиринта. Робот начинает движение, стоя в центре лабиринта и глядя на север. Можно повернуть робота на север, восток, юг или запад и можно направить робота двигаться вперед (т.е. туда, куда он смотрит) на определенное расстояние, но он остановится, если достигнет стены до завершения задания. Сформулируйте эту задачу. Насколько велико будет пространство состояний?

а) При навигации по лабиринту единственное место, где нужно будет поворачивать, — это пересечение двух или более коридоров. Переформулируйте эту задачу, опираясь на данное наблюдение. Насколько велико будет пространство состояний в этом случае?

б) Из каждой точки лабиринта можно двигаться в любом из четырех направлений, пока не будет достигнута точка поворота, и поворот будет единственным действием, которое нужно будет выполнить. Переформулируйте проблему, используя эти действия. Нужно ли теперь отслеживать ориентацию робота?

в) В первоначальном описании задачи мы уже абстрагировались от реального мира, ограничив возможные действия и устранив детали. Назовите три упрощения из числа тех, которые были нами сделаны.

3.4. Имеется сетка из квадратов 9×9, каждый из которых может быть окрашен в красный или синий цвет. Изначально сетка окрашена в синий цвет, но цвет любого квадрата можно изменить любое количество раз. Представим исходную сетку как разделенную на девять субквадратов размером 3×3 и поставим задачу закрасить каждый субквадрат одним цветом, причем все его соседние субквадраты должны быть окрашены другим цветом.

а) Сформулируйте эту задачу самым очевидным способом. Рассчитайте размер ее пространства состояний.

б) Допустим, что каждый квадрат разрешается покрасить только один раз. Переформулируйте задачу и вычислите размер ее пространства состояний. Будет ли поиск в ширину в этом варианте задачи работать быстрее, чем в варианте из пункта *а*? А что можно сказать в отношении поиска с итеративным углублением?

в) Учитывая цель, необходимо рассмотреть лишь варианты окраски, в которых каждый субквадрат окрашен равномерно. Переформулируйте задачу и вычислите размер пространства состояний.

г) Сколько решений у этой задачи?

д) В пунктах *б* и *в* последовательно абстрагируется исходная задача из пункта *а*. Можете ли вы дать перевод решения задачи *в* в решение задачи *б*, а затем — решения задачи *б* в решение задачи *а*?

3.5. Предположим, что два друга живут в разных городах на карте, например такой, как карта Румынии на рис. 3.1. На каждом этапе можно одновременно переместить каждого из друзей в соседний город на карте. Время, необходимое для перемещения из города i в соседний город j, равно длине дороги $d(i, j)$ между городами, но на каждом этапе друг, который прибывает первым, должен ждать, пока второй достигнет своего города (и позвонит первому на его мобильный телефон), чтобы можно было начать следующий этап. Наша цель — обеспечить, чтобы два друга встретились как можно быстрее.

а) Запишите подробную формулировку этой задачи поиска. (Здесь будет полезно определить некоторые формальные обозначения.)

б) Пусть $D(i, j)$ — расстояние по прямой между городами i и j. Какие из следующих эвристических функций будут допустимыми: **i)** $D(i, j)$; **ii)** $2D(i, j)$; **iii)** $D(i, j)/2$?

в) Существуют ли полностью связные карты, для которых не существует решения?

г) Существуют ли карты, на которых все решения требуют, чтобы один друг дважды посетил один и тот же город?

3.6. Покажите, что в задаче головоломки ***Восемь*** состояния подразделяются на два непересекающихся множества, таких, что ни одно состояние из первого множества не может быть преобразовано в состояние из второго множества, даже с применением сколь угодно большого количества ходов. (Подсказка: см. Berlekamp E.R., Conway J.H., Guy R.K. *Winning Ways, For Your Mathematical Plays*. (1982) Academic Press, New York.) Разработайте процедуру, позволяющую узнать, к какому множеству относится данное состояние, и объясните, для чего нужно иметь под рукой такую процедуру, формируя состояния случайным образом.

3.7. Рассмотрите задачу с n ферзями, цель которой состоит в размещении ферзей на шахматной доске таким образом, чтобы ни один ферзь не нападал на любого другого (ферзь атакует любую фигуру, находящуюся на одной и той же с ним горизонтали, вертикали или диагонали), используя операторы, которые дополняют описание состояния начиная с пустого состояния, и каждое действие приводит к добавлению к этому состоянию еще одного ферзя. Объясните, почему размер пространства состояний задачи равен по меньшей мере $\sqrt[3]{n!}$, и оцените наибольшее значение n, для которого является осуществимым исчерпывающее исследование. (Подсказка: определите нижнюю границу коэффициента ветвления, рассматривая максимальное количество клеток, которые ферзь может атаковать в любом столбце.)

3.8. Для каждой из следующих задач дайте ее полную формулировку, выбрав такую, которая будет достаточно точной, чтобы ее можно было успешно реализовать.

а) Необходимо раскрасить плоскую карту, используя только четыре цвета, причем таким образом, чтобы никакие два смежных региона не имели один и тот же цвет.

б) Обезьяна, имеющая рост 3 фута, находится в комнате, где под потолком на высоте 8 футов подвешено несколько бананов. Обезьяна хочет достать бананы. В комнате находятся две проволочные корзины высотой по 3 фута, которые можно передвигать, ставить друг на друга и на которые можно залезать.

в) Программа выводит сообщение "недопустимая входная запись" после передачи ей некоторого файла с входными записями. Известно, что обработка каждой записи происходит независимо от других записей. Требуется обнаружить, какая запись является недопустимой.

г) Имеются три кувшина, емкостью 12 , 8 и 3 галлона, а также водопроводный кран. Кувшины можно заполнять или опорожнять, выливая воду из одного кувшина в другой или на землю. Необходимо отмерить ровно один галлон.

3.9. Рассмотрите задачу определения кратчайшего пути между двумя точками на плоскости, на которой имеются препятствия в виде выпуклых многоугольников, как показано на рис. 3.29. Это — идеализация задачи, которую должен решать робот, прокладывая свой путь через среду, в которой мало свободного места.

а) Предположим, что пространство состояний состоит из всех позиций (x,y) на плоскости. Каково количество состояний в этом пространстве? Каково в нем количество путей к цели?

б) Кратко поясните, почему в этой двухмерной сцене кратчайший путь от одной вершины многоугольника до любой другой должен состоять из прямолинейных отрезков, соединяющих некоторые вершины многоугольников. А теперь определите более приемлемое пространство состояний. Насколько велико это пространство состояний?

в) Определите функции, необходимые для реализации решения этой задачи поиска, включая функцию определения преемника, которая принимает в качестве входных данных координаты любой из вершин и возвращает множество вершин, достижимых по прямой линии от данной вершины. (Не забывайте при этом о соседних вершинах того же многоугольника.) Используйте в качестве значения эвристической функции расстояние по прямой между указанными точками.

г) Примените один или несколько алгоритмов, представленных в настоящей главе, для решения ряда задач из этой области и прокомментируйте данные об их производительности.

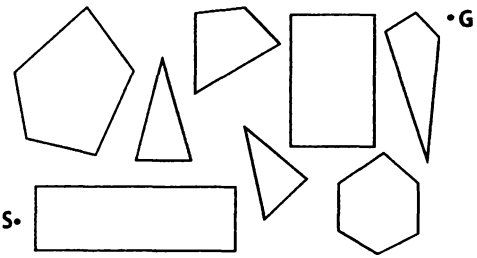


Рис. 3.29. Плоскость с многоугольными препятствиями

3.10. В разделе 3.1.1 было указано, что мы не будем принимать во внимание задачи с отрицательными значениями стоимости пути. В данном упражнении эта тема рассматривается немного более подробно.

а) Предположим, что действия могут иметь произвольно большие отрицательные стоимости; объясните, почему такая ситуация может вынудить любой оптимальный алгоритм исследовать полное пространство состояний.

б) Удастся ли выйти из этого положения, потребовав, чтобы стоимости этапов были больше или равны некоторой отрицательной константе c? Рассмотрите и деревья, и графы.

в) Предположим, что имеется множество операторов, образующих цикл, так что выполнение операторов этого множества в определенном порядке не приводит к какому-либо чистому изменению состояния. Если все эти операторы имеют отрицательную стоимость, то какие выводы из этого следуют применительно к оптимальному поведению агента в такой среде?

г) Можно легко представить себе, что операторы с высокой отрицательной стоимостью имеются даже в таких проблемных областях, как поиск маршрута. Например, некоторые участки дороги могут оказаться настолько живописными, что стремление ознакомиться с ними намного перевесит обычные здравые рассуждения о стоимости, измеряемой в терминах затрат времени и топлива. Объясните, применяя точные термины, принятые в контексте поиска в пространстве состояний, почему все же люди не ведут свои автомобили неопределенно долго по живописным циклическим участкам пути, и укажите, каким образом нужно определить пространство состояний и операторы для задачи поиска маршрута, чтобы агенты с искусственным интеллектом также могли избежать попадания в цикл.

д) Можете ли вы придумать пример такой реальной проблемной области, в которой стоимости этапов таковы, что могут вызвать возникновение цикла?

3.11. Задача обычно формулируется следующим образом. Три миссионера и три каннибала находятся на одной стороне реки, где также находится лодка, которая может выдержать одного или двух человек. Найдите способ перевезти всех на другой берег реки, никогда не оставляя где-либо группу миссионеров, которую превосходила бы по численности группа каннибалов. Это известная задача в искусственном интеллекте, поскольку она была темой первой статьи, в которой был применен подход к формулировке проблемы с аналитической точки зрения ([40], 1968).

а) Точно сформулируйте эту задачу, определяя только те различия, которые необходимы для обеспечения правильного решения. Нарисуйте схему ее полного пространства состояний.

б) Реализуйте и решите эту задачу оптимальным образом, используя соответствующий алгоритм поиска. Имеет ли смысл проверять наличие повторяющихся состояний?

в) Почему, по вашему мнению, люди сталкиваются с затруднениями при решении этой головоломки, несмотря на то, что пространство ее состояний является чрезвычайно простым?

3.12. Самостоятельно сформулируйте определения следующих понятий: состояние, пространство состояний, дерево поиска, поисковый узел, цель, действие, функция определения преемника и коэффициент ветвления.

3.13. В чем состоит различие между состоянием мира, описанием состояния и поисковым узлом? Почему это различие полезно?

3.14. Некоторые действия на самом деле могут состоять из длинной последовательности более мелких операций: включить мотор автомобиля, отпустить тормоз, начать движение вперед и т.д. Наличие составных действий такого рода уменьшает количество шагов в последовательности решения, тем самым сокращая время поиска. Предположим, мы довели этот подход до логического предела, создав сверхсоставные действия из всех возможных последовательностей действий. Тогда каждый экземпляр задачи будет решаться одним суперсоставным действием подобного толка. Объясните, как поиск мог бы работать при такой формулировке задачи. Будет ли такой подход практичным с точки зрения ускорения решения задач?

3.15. Всегда ли конечное пространство состояний приводит к конечному дереву поиска? А что можно сказать в отношении конечного пространства состояний, которое является деревом? Можете ли вы более точно указать, какие типы пространств состояний всегда приводят к конечным деревьям поиска?

3.16. Покажите, что удовлетворяет свойству разделения графов, показанному на рис. 3.6. (*Подсказка*: начните с демонстрации, что это свойство выполняется в начале, а затем покажите, что если оно выполняется до начала очередной итерации алгоритма, то оно выполняется и после ее завершения.) Опишите алгоритм поиска, который нарушает это свойство.

3.17. Какие из следующих утверждений истинны, а какие ложны? Поясните свой ответ.

а) Поиск в глубину всегда развертывает по крайней мере столько узлов, сколько поиск с допустимой эвристикой.

б) $h(n) = 0$ — допустимая эвристика для головоломки ***Восемь***.

в) Поиск A* бесполезен в робототехнике, поскольку восприятия, состояния и действия в ней непрерывны.

г) Поиск в ширину является полным, даже если допустимы нулевые стоимости путей.

д) Допустим, что ладья может перемещаться по шахматной доске на любое количество квадратов по прямой линии, вертикальной или горизонтальной, но не может перепрыгивать через другие фигуры. В этом случае манхэттенское расстояние является допустимой эвристикой для задачи перемещения ладьи из квадрата A в квадрат B за наименьшее количество ходов.

3.18. Рассмотрим пространство состояний, в котором начальным состоянием является число 1 и у каждого состояния k есть два преемника: числа $2k$ и $2k+1$.

а) Нарисуйте часть пространства состояний для состояний от 1 до 15.

б) Предположим, что целевое состояние равно 11. Перечислите порядок посещения узлов при поиске в ширину, поиске с ограничением глубины с лимитом 3 и при итеративном поиске с углублением.

в) Насколько хорошо двунаправленный поиск будет работать в этой задаче? Каков будет фактор ветвления в каждом направлении двунаправленного поиска?

г) Предлагает ли ответ на задание *в* такую переформулировку задачи, которая позволяет решить задачу перехода из состояния 1 в заданное целевое состояние практически без поиска?

д) Назовите действие, ведущее от k к $2k$ *слева*, и действие, ведущее к $2k+1$ *справа*. Можете ли вы найти алгоритм, который выводит решение этой задачи без какого-либо поиска вообще?

3.19. Простой деревянный железнодорожный конструктор содержит части, показанные на рис. 3.30. Задача состоит в том, чтобы соединить эти части в замкнутый железнодорожный путь, в котором не будет перекрывающихся путей и свободных концов, где поезд мог бы сойти с рельсов.

а) Предположим, что все части соединяются точно, без зазоров. Дайте точную формулировку задания как поисковой задачи.

б) Укажите подходящий неинформированный алгоритм поиска для этой задачи и поясните свой выбор.

в) Объясните, почему удаление какой-либо одной развилки сделает задачу неразрешимой.

г) Укажите верхнюю границу общего размера пространства состояний, определенного вашей формулировкой задачи. (*Подсказка.* Подумайте о максимальном коэффициенте ветвления для процесса строительства и максимальной глубине, игнорируя проблему перекрывающихся путей и свободных концов. Начните с предположения, что каждый элемент конструктора уникален.)



Рис. 3.30. Комплект элементов пути игрушечной железной дороги

3.20. Реализуйте следующие две версии функции определения преемника для задачи головоломки ***Восемь***. Первая должна копировать и редактировать структуру данных родительского узла *s*, а вторая должна прямо модифицировать родительское состояние (отменяя эти модификации в случае необходимости). Напишите версии процедуры поиска в глубину с итеративным углублением, в которых используются эти функции, и сравните данные об их производительности.

3.21. Если в поиске с итеративным углублением вместо увеличивающихся пределов глубины использовать увеличивающиеся пределы стоимости пути, то это будет алгоритм, получивший название *поиск с итеративным удлинением* — итеративный аналог поиска по критерию стоимости. Его основная идея состоит в том, что при поиске следует использовать увеличивающиеся пределы стоимости пути. Если сформирован узел, стоимость пути которого превышает текущий предел, этот узел немедленно отбрасывается. При каждой новой итерации предел устанавливается равным самому низкому значению стоимости пути для любого узла, отвергнутого в предыдущей итерации.

а) Покажите, что этот алгоритм оптимален в отношении общих методов определения стоимости пути.

б) Рассмотрим однородное дерево с коэффициентом ветвления b, глубиной решения d и единичными стоимостями этапов. Сколько итераций потребуется при использовании алгоритма итеративного удлинения?

в) Теперь рассмотрите стоимости этапов, взятые из непрерывного диапазона $[\epsilon, 1]$, где $0 < \epsilon < 1$. Сколько итераций требуется в худшем случае?

в) Реализуйте этот алгоритм и примените его к экземплярам задачи головоломки ***Восемь*** и задачи коммивояжера. Сравните производительность этого алгоритма с производительностью алгоритма поиска по критерию стоимости и прокомментируйте полученные результаты.

3.22. Опишите пространство состояний, в котором поиск с итеративным углублением выполняется намного хуже, чем поиск в глубину (например, $O(n^2)$ вместо $O(n)$).

3.23. Напишите программу, которая будет принимать в качестве входных данных два URL-адреса веб-страниц и находить путь из ссылок от одной к другой. Какая стратегия поиска будет подходящей в этом случае? Является ли двунаправленный поиск хорошей идеей? Можно ли использовать поисковую систему в качестве реализации функции определения предшественника?

3.24. Рассмотрим задачу мира пылесоса, определенную в разделе 2.1 (см. рис. 2.2).

а) Какой из алгоритмов, определенных в этой главе, подойдет для решения этой задачи? Алгоритм должен использовать поиск по дереву или поиск по графу?

б) Примените выбранный алгоритм для вычисления оптимальной последовательности действий для мира 3×3, в начальном состоянии которого мусор находится в трех верхних квадратах, а агент находится в центре.

в) Сконструируйте агента, выполняющего поиск в подобном мире пылесоса и оцените его производительность в множестве миров 3×3 с вероятностью присутствия мусора в каждом квадрате, равной 0,2. Включите в показатель эффективности стоимость поиска, а также стоимость пути, используя разумный для данного случая "курс обмена".

г) Сравните вашего лучшего поискового агента с простым рандомизированным рефлекторным агентом, который убирает мусор, если он имеется, или случайным образом перемещается в противном случае.

д) Оцените, что произойдет, если этот мир пылесоса будет увеличен до размеров $n \times n$. Как производительность поискового агента и рефлекторного агента зависит от значения n?

3.25. Докажите каждое из следующих утверждений или приведите контрпример.

а) Поиск в ширину — это особый случай поиска по критерию стоимости.

б) Поиск в глубину — это особый случай жадного поиска по первому наилучшему совпадению по дереву.

в) Поиск по критерию стоимости является частным случаем поиска A*.

3.26. Сравните производительность алгоритмов поиска A* и RBFS на множестве случайно сгенерированных экземпляров задач в проблемных областях задачи головоломки ***Восемь*** (с манхэттенским расстоянием) и задачи TSP (с MST — см. упражнение 3.38). Обсудите полученные результаты. Как изменится производительность алгоритма RBFS после добавления небольшого случайного числа к эвристическим значениям в проблемной области задачи головоломки ***Восемь***?

3.27. Выполните трассировку работы алгоритма поиска A* применительно к задаче поиска оптимального пути до Бухареста из города Лугож, используя эвристику расстояния по прямой линии. Иными словами, запишите последовательность узлов, которые будет рассматривать этот алгоритм, и дайте значения оценок f, g и h для каждого из них.

3.28. Иногда для некоторой задачи нет хорошей функции оценки, но есть хороший метод сравнения: способ определения, будет ли один узел лучше, чем другой, без присвоения им числовых значений. Покажите, что этого будет достаточно для выполнения поиска по первому наилучшему совпадению. Существует ли соответствующий аналог для алгоритма поиска A*?

3.29. Предложите пространство состояний, в котором применение поиска A* будет давать субоптимальное решение при использовании функции $h(n)$, являющейся допустимой, но непреемственной.

3.30. Точная эвристика не обязательно сокращает время поиска в худшем случае. Для любой глубины d определите задачу поиска с целевым узлом на глубине d и напишите эвристическую функцию так, чтобы $|h(n) - h^*(n)| \leq O(\log h^*(n))$, но при этом алгоритм A* развертывал все узлы глубиной меньше d.

3.31. **Эвристический алгоритм поиска пути** ([1803], 1977) представляет собой алгоритм поиска по первому наилучшему совпадению, в котором функция оценки имеет вид $f(n) = (2 - w)g(n) + wh(n)$. Для каких значений w этот алгоритм будет полным? Для каких значений w он будет оптимальным, если предположить, что функция h является допустимой? Какой вид поиска выполняется в этом алгоритме при $w = 0$, $w = 1$ и $w = 2$?

3.32. Рассмотрим неограниченную версию регулярного двухмерного клеточного мира. Начальное состояние находится в начале координат (0, 0), а целевое состояние имеет координаты (x, y).

а) Каков фактор ветвления b в этом пространстве состояний?

б) Сколько различных состояний существует на глубине k (для $k > 0$)?

в) Какое максимальное количество узлов развертывается при древовидном поиске в ширину?

г) Какое максимальное количество узлов развертывается при поиске в ширину по графу?

д) Является ли $h = |u - x| + |v - y|$ допустимой эвристикой для состояния (u, v)? Поясните.

е) Сколько узлов развертывается при поиске A* по графу с использованием эвристики h?

ж) Останется ли функция h допустимой, если некоторые связи будут удалены?

з) Останется ли функция h допустимой, если будут добавлены связи между некоторыми несмежными состояниями?

3.33. В клеточном мире размерностью $n \times n$ автомобили в количестве n единиц занимают квадраты от (1, 1) до (n, 1), т.е. весь нижний ряд. Все автомобили следует переместить в верхний ряд мира, но в обратном порядке. Иначе говоря, автомобиль i, который начинает движение из клетки (i, 1), должен закончить свой путь в клетке ($n-i+1, n$). На каждом этапе передвижения каждый из n автомобилей может переместиться на одну клетку вверх, вниз, влево или вправо или остаться на месте, и если автомобиль остался на месте, другой соседний автомобиль (но не более одного) может перепрыгнуть через него. Два автомобиля не могут занимать один и тот же квадрат.

а) Определите размер пространства состояний задачи как функцию от n.

б) Рассчитайте коэффициент ветвления как функцию от n.

в) Предположим, что автомобиль i находится в точке (x_i, y_i). Запишите нетривиальную допустимую эвристику h_i для количества ходов, которые потребуются для достижения его целевого состояния ($n-i+1, n$), в предположении, что в этом клеточном мире нет других автомобилей.

г) Какие из следующих эвристик допустимы для задачи перемещения всех n автомобилей к месту назначения? Поясните свой ответ.

- 1. $\sum_{i=1}^{n} h_i$.
- 2. $\max h_1, \ldots, h_n$.
- 3. $\min h_1, \ldots, h_n$.

3.34. Рассмотрим задачу перемещения k коней из k исходных квадратов $s_1, \ldots, s_k$ в k целевых квадратов $g_1, \ldots, g_k$ на неограниченной шахматной доске при условии, что никакие два коня не могут одновременно оказаться на одной и той же клетке. Каждое действие состоит в одновременном перемещении всех k коней. Цель — завершить перемещение за наименьшее количество действий.

а) Каков будет максимальный коэффициент ветвления в этом пространстве состояний, выраженный в виде функции от k?

б) Предположим, что h_i — это допустимая эвристика для задачи перемещения коня i в целевое состояние g_i. Какие из следующих эвристик будут допустимы для задачи с k конями? Какая из них будет лучшей?

- 1. $\min h_1, \ldots, h_k$.
- 2. $\max h_1, \ldots, h_k$.
- 3. $\sum_{i=1}^{k} h_i$

в) Выполните задание *б* для случая, когда будет разрешено перемещать только одного коня за ход.

3.35. В разделе 3.5.1 было показано, что применение эвристической функции определения расстояния прямой линии приводит к тому, что жадный поиск по первому наилучшему совпадению безвозвратно углубляется в дерево поиска при решении задачи поиска пути из города Яссы в город Фэгэраш. Однако эта эвристика отлично работает при решении противоположной задачи — поиска пути от Фэгэраша до Ясс. Существуют ли такие задачи, в которых эта эвристическая функция не позволяет успешно выполнить поиск пути в обоих направлениях?

3.36. Придумайте эвристическую функцию для головоломки ***Восемь***, которая иногда допускает переоценку, и покажите, каким образом это может привести к получению неоптимального решения некоторых конкретных экземпляров задачи. (Для упрощения работы при выполнении упражнения можете использовать компьютер.) Докажите, что если функция h никогда не переоценивает больше чем на c, то алгоритм поиска A*, использующий функцию h, всегда возвращает решение, стоимость которого превышает стоимость оптимального решения не более чем на c.

3.37. Докажите, что если эвристическая функция является преемственной, то она должна быть допустимой. Предложите допустимую эвристическую функцию, которая не является преемственной.

3.38. Задача коммивояжера (TSP) может быть решена с помощью эвристической функции, основанной на определении минимального связующего дерева (Minimum Spanning Tree — MST), которая используется для оценки стоимости завершения обхода, при условии, что уже был сформирован частичный путь обхода. Стоимость MST для множества городов представляет собой минимальную сумму стоимостей соединений в любом дереве, которое связывает все города.

а) Покажите, как эта эвристика может быть получена из упрощенной версии TSP.

б) Покажите, что эвристика MST доминирует над эвристической функцией расстояния по прямой.

в) Напишите генератор задач для создания экземпляров задачи TSP, в которых города представлены случайно выбранными точками в единичном квадрате.

г) Найдите в литературе эффективный алгоритм построения MST и используйте его с поиском по графу для решения экземпляров задач TSP.

3.39. В разделе 3.6.2 был определен ослабленный вариант головоломки ***Восемь***, в которой любая плитка может перемещаться из квадрата A в квадрат B, если квадрат B пуст. Точное решение этой проблемы определяет эвристику Гашнига ([816], 1979). Объясните, почему эвристика Гашнига является по меньшей мере такой же точной, как функция h_1 (плитки не на месте), и продемонстрируйте случаи, когда она более точна, чем функции h_1 и h_2 (манхэттенское расстояние). Предложите эффективный способ вычисления эвристической функции Гашнига.

3.40. В этой главе для головоломки ***Восемь*** были предложены две простые эвристические функции: "манхэттенское расстояние" и "плитки не на месте". В литературе было предложено несколько других эвристических функций в качестве их улучшенной замены, например см. **Hansson O.**, Mayer A., Yung M. "Criticizing solutions to relaxed models yields powerful admissible heuristics". *Information Sciences*, 63(3), p. 207–227. 1992; **Mostow J.**, Prieditis A.E. *Discovering admissible heuristics by abstracting and optimizing: A transformational approach.* In Proceedings of the Eleventh International Joint Conference on Artificial Intelligence (IJCAI 89), Vol. 1, p. 701–707, 1989, Detroit. Morgan Kaufmann.; **Nilsson N.J.** *Problem Solving Methods in Artificial Intelligence.* 1971. McGraw Hill, New York. Проверьте обоснованность этих претензий, реализовав соответствующие эвристические функции и сравнив производительность полученных алгоритмов.

ГЛАВА 4

Поиск в сложных средах

В этой главе будут ослаблены упрощающие предположения, принятые в предыдущей главе, что позволит приблизиться к реальности.

В главе 3 рассматривались задачи поиска только для полностью наблюдаемых, детерминированных, статических, известных сред, в которых решение представляет собой некоторую последовательность действий. В этой главе данные ограничения будут ослаблены. Мы начнем с задачи нахождения хорошего состояния, не беспокоясь о пути к нему, включив в рассмотрение как дискретные (раздел 4.1), так и непрерывные (раздел 4.2) пространства состояний. Затем будет ослаблено предположение как о детерминированности (раздел 4.3), так и о наблюдаемости (раздел 4.4) среды. В случае недетерминированного мира агенту понадобится условный план и придется выполнять различные действия в зависимости от того, что он будет наблюдать в своем окружении, например останавливаться, если свет светофора красный, и двигаться без остановки, если свет зеленый. При условии частичной наблюдаемости среды агенту также потребуется отслеживать различные возможные состояния, в которых он может оказаться. Наконец, в разделе 4.5 обсуждается поведение агента в неизвестном ему пространстве, которое ему придется изучать по мере продвижения в нем, используя **поиск в оперативном режиме**.

4.1. Локальный поиск и задачи оптимизации

В поисковых задачах, обсуждавшихся в главе 3, требовалось найти путь в заданном поисковом пространстве, например выяснить, как добраться из Арада до Бухареста. Но при решении многих задач имеет значение только достижение конечного состояния, а путь к цели не представляет интереса. Например, в задаче с восемью ферзями (рис. 4.3) для нас важна лишь окончательная конфигурация ферзей, а не порядок, в котором они расставляются на доске. К этому классу также относится множество важных приложений, таких как проектирование интегральных схем, размещение оборудования в цехах, составление производственных графиков, автоматическое программирование, оптимизация телекоммуникационных сетей, разработка плана уборки урожая или управление портфелем акций.

Алгоритмы ▶**локального поиска** действуют лишь с учетом текущего состояния, предусматривая переход в какое-то одно из состояний из числа соседних по отношению к текущему. Они не отслеживают ни пути, ни набора состояний, которые уже были достигнуты. Это означает, что они не являются систематическими, т.е. не обеспечивают полное исследование той части пространства поиска, где может находиться решение. Тем не менее они обладают двумя важными преимуществами: во-первых, в них используется очень небольшой объем памяти и, во-вторых, они часто позволяют находить приемлемые решения в больших или бесконечных (непрерывных) пространствах состояний, для которых систематические алгоритмы не применимы.

Локальные поисковые алгоритмы могут также использоваться для решения ▶**задач оптимизации**, в которых цель заключается в поиске состояния, наилучшего с точки зрения ▶**целевой функции**.

Чтобы лучше понять суть локального поиска, обратимся к представлению состояний некоторой задачи в виде ▶**ландшафта пространства состояний**, как показано на рис. 4.1. Каждая точка (состояние) в ландшафте характеризуется "возвышением", определяемым значением целевой функции (другой вариант — функции стоимости). Если возвышение каждой точки (состояния) соответствует целевой функции, то наша задача заключается в поиске самого высокого пика — ▶**глобального максимума**, — и тогда этот процесс называют **восхождением к вершине**. Если же возвышение соответствует значению функции стоимости для состояния, то наша задача заключается в поиске самой глубокой долины — ▶**глобального минимума**, — и тогда этот процесс называют **градиентным спуском**.

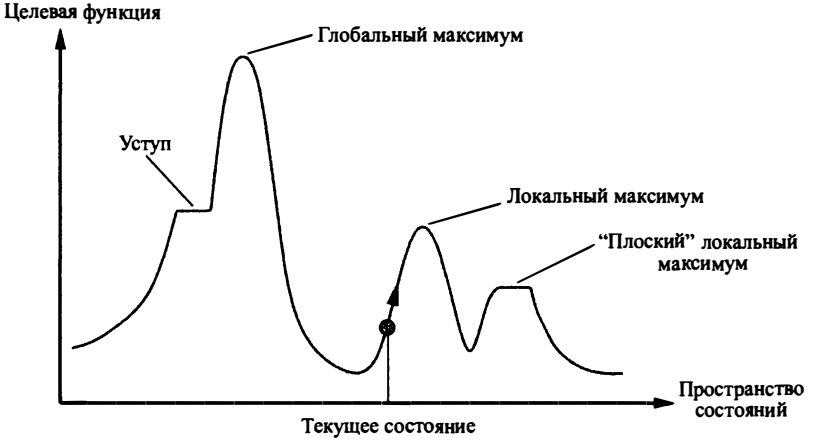


Рис. 4.1. Ландшафт одномерного пространства состояний, в котором возвышение соответствует целевой функции. Задача состоит в поиске глобального максимума

4.1.1. Поиск восхождением к вершине

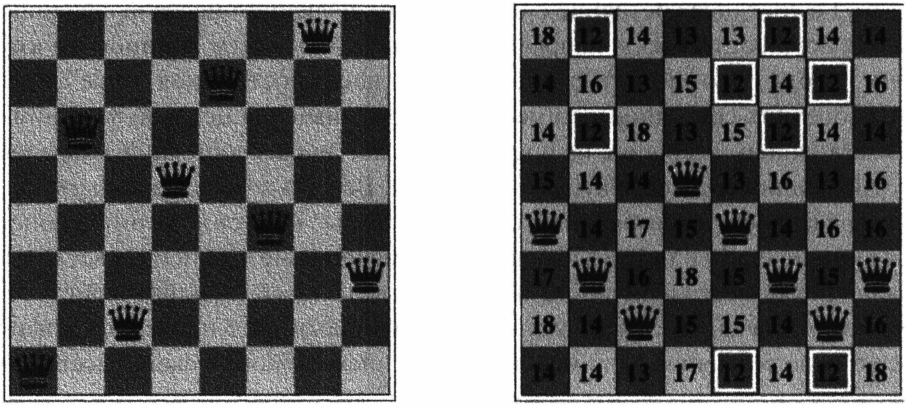
Алгоритм поиска ▶**восхождением к вершине** приведен на рис. 4.2. В нем отслеживается единственное текущее состояние и на каждой итерации выполняется переход в соседнее состояние с наибольшим значением целевой функции, т.е. движение осуществляется в направлении, обеспечивающем ▶**наиболее крутой подъем**. Работа этого алгоритма заканчивается после достижения "пика", когда ни одно из соседних состояний не имеет большего значения целевой функции. Поиск восхождением к вершине не предусматривает анализ состояний за пределами круга ближайших соседей текущего состояния. Все это похоже на попытку альпиниста, страдающего амнезией, найти вершину горы Эверест в густом тумане. Обратите внимание, что одним из вариантов применения метода поиска восхождением к вершине является использование в качестве целевой функции эвристической стоимости со значениями, взятыми с обратным знаком, что приведет к локальному подъему до состояния с наименьшим значением эвристического расстояния до цели.

```
function HILL-CLIMBING(problem) returns состояние, представляющее
        собой локальный максимум
  current ← problem.INITIAL
  while true do
    neighbor ← преемник состояния current с наивысшим значением
          целевой функции VALUE
    if VALUE(neighbor) ≤ VALUE(current) then return current
    current ← neighbor
```

Рис. 4.2. Алгоритм поиска восхождением к вершине, являющийся фундаментальным методом локального поиска. На каждом этапе текущий узел заменяется наилучшим соседним узлом

Для иллюстрации работы поиска восхождением к вершине воспользуемся ▶**задачей о восьми ферзях**, представленной на рис. 4.3. В этом примере будет использоваться ▶**формулировка полного состояния**, в которой в каждом состоянии на доске имеются все восемь ферзей, по одному ферзю в каждом столбце, но не обязательно в нужной позиции. Начальное состояние выбирается случайным образом, и его преемниками являются все возможные состояния, достигаемые посредством перемещения одного из ферзей на соседнюю клетку в том же самом столбце (в этом случае каждое состояние имеет $8 \times 7 = 56$ преемников). Эвристическая функция стоимости h определяет количество пар ферзей, которые атакуют друг друга прямо или косвенно, т.е. когда на одной линии находится больше двух ферзей и атака засчитывается через промежуточные фигуры. Глобальный минимум этой функции будет равен нулю, и это состояние соответствует идеальному

решению. На рис. 4.3, *б* показаны состояние со значением функции $h = 17$, а также значения этой функции для всех возможных состояний-преемников.



а) б)

Рис. 4.3. Задача о 8 ферзях: на шахматной доске разместите 8 ферзей так, чтобы ни один из них не нападал на другой (ферзь атакует любую фигуру в той же строке, столбце или диагонали). **а)** Эта позиция является почти решением, за исключением двух ферзей в четвертом и седьмом столбцах, которые атакуют друг друга по диагонали. **б)** Состояние задачи о 8 ферзях с эвристической оценкой стоимости $h = 17$. На доске также показано значение функции h для каждого возможного преемника, достигаемого путем перемещения ферзя в пределах своего столбца, и отмечено 8 ходов, дающих наилучший результат с $h = 12$. Алгоритм поиска восхождением к вершине выберет один из них

Метод поиска восхождением к вершине иногда называют ►**жадным локальным поиском**, поскольку в нем предусматривается переход в самое хорошее соседнее состояние без предварительных рассуждений о том, куда лучше отправиться дальше. Хотя жадность считается одним из семи смертных грехов, практика показала, что жадные алгоритмы часто демонстрируют достаточно высокую производительность. Поиск восхождением к вершине обычно позволяет получить быстрое продвижение в сторону решения, поскольку улучшить начальное плохое состояние чаще всего достаточно просто. Например, в состоянии, представленном на рис. 4.3, *б*, достаточно сделать лишь пять ходов, чтобы достичь состояния, показанного на рис. 4.3 с оценкой $h = 1$ и находящегося очень близко к одному из возможных решений. К сожалению, поиск восхождением к вершине часто заходит в тупик по любой из следующих причин.

- ►**Локальные максимумы**. Локальный максимум представляет собой пик, более высокий по сравнению с каждым из его соседних состояний, но более

низкий, чем глобальный максимум. Алгоритмы поиска восхождением к вершине, достигнув окрестностей локального максимума, обеспечивают уверенное продвижение вверх, к вершине этого пика, но в конечном счете заходят в тупик, из которого для них нет выхода. Такая задача схематически показана на рис. 4.1. Более конкретный пример состоит в том, что состояние, показанное на рис. 4.3, *а*, фактически представляет собой локальный максимум (т.е. локальный минимум для оценки стоимости h); задача еще не решена, но при любом передвижении отдельно взятого ферзя ситуация становится только хуже.

- ►**Хребты**. На рис. 4.4 представлен типичный пример хребта, представляющего собой последовательность локальных максимумов, задача прохождения которых для жадных алгоритмов является очень трудной.
- ►**Плато**. Это такая область в ландшафте пространства состояний, где функция оценки является плоской. Она может представлять собой плоский локальный максимум, из которого не существует выхода вверх, либо иметь вид ►**уступа**, из которого теоретически возможно дальнейшее успешное продвижение (см. рис. 4.1). Алгоритм поиска восхождением к вершине может оказаться неспособным выйти за пределы плато.

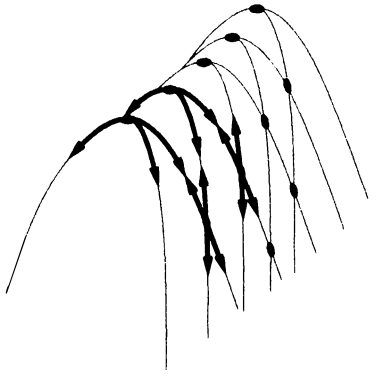


Рис. 4.4. Иллюстрация того, почему хребты вызывают сложности для алгоритмов поиска восхождением к вершине. На хребет, повышающийся в направлении слева направо, наложена решетка состояний (темные дуги), образующая последовательность локальных максимумов, не связанных напрямую друг с другом. В любом из локальных максимумов все доступные действия направлены вниз. Подобные топологии распространены в низкоразмерных пространствах состояний, таких как точки в двухмерной плоскости. Но в пространствах состояний с сотнями или тысячами измерений эта наглядная картина не применима, и обычно есть по крайней мере несколько измерений, которые позволят алгоритму покинуть гребни или плато

В каждом из этих случаев рассматриваемый алгоритм достигает такой точки, из которой дальнейшее успешное продвижение невозможно. Так, начиная со случайно сформированного состояния задачи о восьми ферзях, алгоритм поиска восхождением к вершине по самому крутому подъему заходит в тупик в 86% случаев, успешно справляясь лишь с 14% экземпляров этой задачи. Однако он работает очень быстро, выполняя в среднем только 4 этапа в случае успешного завершения и 3 этапа, если заходит в тупик. Это неплохой результат для пространства состояний с $8^8 \approx 17$ миллионами состояний.

Как можно добиться успешного решения большего числа задач? Один из ответов — продолжать движение до тех пор, когда будет достигнуто плато, т.е. разрешить ▶**перемещение в сторону** в надежде на то, что плато действительно является уступом, как показано на рис. 4.1. Однако, если на самом деле будет найден плоский локальный максимум, может возникнуть бесконечный цикл и алгоритм застрянет на плато навсегда. Тогда возможный вариант — ограничить количество последовательных движений в сторону, прекращая работу алгоритма после, скажем, 100 переходов на плато. Это позволяет увеличить процент успешно решаемых методом восхождения к вершине экземпляров задач с 14% до 94%. Однако за такой успех приходится платить: алгоритм в среднем выполняет примерно 21 этап при каждом успешном решении экземпляра задачи и 64 этапа в случае неудачи.

Было разработано немало вариантов поиска восхождением к вершине. При поиске ▶**стохастическим восхождением к вершине** случайным образом осуществляется выбор одного из возможных перемещений вверх; при этом вероятность такого выбора может зависеть от крутизны движения вверх. Как правило, этот алгоритм сходится медленнее по сравнению с вариантом, предусматривающим наиболее крутой подъем, но в некоторых ландшафтах состояний он уверенно находит лучшие решения. При поиске ▶**восхождением к вершине с выбором первого варианта** реализуется стохастический поиск восхождением к вершине путем выбора преемников случайным образом до тех пор, пока не будет найден преемник, лучший по сравнению с текущим состоянием. Это хорошая стратегия, если любое состояние имеет большое количество преемников (например, тысячи).

Еще один вариант — это ▶**поиск восхождением к вершине со случайным перезапуском** следует широко известной рекомендации: "Если первая попытка была неудачной, пробуйте снова и снова". В этом варианте серия поисков из различных начальных состояний, сформированных случайным образом, выполняется, пока цель не будет достигнута. Он является полным с вероятностью, достигающей 1, даже по той тривиальной причине, что в нем в конечном итоге в качестве начального состояния формируется одно из целевых состояний. Если вероятность успеха каждого поиска восхождением к вершине равна p, то ожидаемое количество требуемых перезапусков составляет $1/p$. Для экземпляров задачи с восемью ферзями, если не разрешено движение в сторону, $p \approx 0{,}14$, поэтому для нахождения цели требуется приблизительно 7 итераций (6 неудачных и 1 успешная). Ожидаемое количество этапов решения равно стоимости одной успешной итерации,

которая складывается с увеличенной в $(1 - p)/p$ раз стоимостью неудачи, или составляет приблизительно 22 этапа. Если разрешено движение в сторону, то в среднем требуется $1/0{,}94 \approx 1{,}06$ итераций и $(1 \times 21) + (0{,}06/0{,}94) \times 64 \approx 25$ этапов. Поэтому алгоритм поиска восхождением к вершине со случайным перезапуском действительно является очень эффективным применительно к задаче с восемью ферзями. Даже для варианта с тремя миллионами ферзей этот подход позволяет находить решения меньше чем за минуту.[1]

Успех алгоритма поиска восхождением к вершине в значительной степени зависит от формы ландшафта пространства состояний: если в нем есть лишь несколько локальных максимумов и плато, то метод поиска восхождением к вершине со случайным перезапуском позволяет очень быстро найти хорошее решение. С другой стороны, многие реальные задачи имеют ландшафт, который больше напоминает семейство дикобразов на плоском полу, и на вершине иглы каждого дикобраза живут другие миниатюрные дикобразы, и так до бесконечности. NP-трудные задачи (см. приложение А) обычно имеют экспоненциальное количество локальных максимумов, способных завести алгоритм в тупик. Несмотря на это часто существует возможность найти достаточно хороший локальный максимум после небольшого количества перезапусков.

4.1.2. Имитация отжига

Алгоритм поиска восхождением к вершине, исключающий возможность движения "вниз по склону" к состояниям с более низкой оценкой (или более высокой стоимостью), всегда подвержен риску зайти в тупик, достигнув локального максимума. В отличие от этого алгоритм с чисто случайным блужданием, предполагающий произвольный выбор преемника, не принимая во внимание его оценку, в конечном счете натолкнется на глобальный максимум, но является крайне неэффективным. Поэтому представляется разумной попытка каким-то образом скомбинировать восхождение к вершине со случайным блужданием, что позволит обеспечить и эффективность, и полноту.

Одним из решений является алгоритм, получивший название ▶**имитация отжига**. В металлургии **отжигом** называют процесс *отпуска* металла или стекла путем нагревания этих материалов до высокой температуры с последующим постепенным охлаждением, что позволяет перевести обрабатываемый материал в низкоэнергетическое кристаллическое состояние. Чтобы пояснить суть имитации отжига, вместо восхождения к вершине обратимся к **градиентному спуску** (т.е. к минимизации затрат) и представим себе, что задача состоит в том, чтобы на очень неровной поверхности загнать маленький шарик в самую глубокую лунку.

[1] В [1459] (1993) предполагается, что в некоторых случаях лучше выполнять перезапуск рандомизированного алгоритма по выполнении фиксированного числа этапов, и показано, что такой подход может оказаться гораздо более эффективным по сравнению с тем, когда разрешено каждый поиск продолжать до бесконечности.

Если позволить шарику просто катиться под уклон, он быстро остановится в ближайшем углублении — локальном минимуме. Но если теперь встряхнуть нашу поверхность, шарик может выскочить из локального минимума и, возможно, закатится в более глубокое углубление, откуда вытряхнуть его будет сложнее. Идея состоит в том, чтобы встряхивать поверхность достаточно сильно, чтобы шарик мог выскочить из локальных минимумов, но не настолько сильно, чтобы он вылетел и из глобального минимума. Поэтому в алгоритме имитации отжига предполагается, что вначале происходит достаточно интенсивное встряхивание (аналогичное нагреву до высокой температуры), а затем интенсивность встряхиваний постепенно уменьшается (что можно сравнить с понижением температуры).

Общая структура алгоритма имитации отжига (рис. 4.5) аналогична структуре алгоритма поиска восхождением к вершине. Однако вместо *наилучшего* хода в нем выбирается *случайный* ход. Если он улучшает ситуацию, то всегда принимается, а если нет, то алгоритм принимает его с некоторой вероятностью, меньшей 1. Эта вероятность экспоненциально убывает с учетом "ухудшения" хода — в зависимости от величины ΔE, на которую ухудшается его оценка. Вероятность также уменьшается по мере понижения "температуры" T: "плохие" ходы с большей вероятностью могут быть разрешены в начале, когда температура высока, но становятся менее вероятными по мере снижения T. Если *график* понижения T до 0 предусматривает достаточно медленное ее снижение, то по свойству распределения Больцмана $e^{\Delta E/T}$ вся вероятность концентрируется на глобальном максимуме, который алгоритм найдет с вероятностью, приближающейся к 1.

---

function SIMULATED-ANNEALING(*problem*, *schedule*) **returns** состояние решения
  current ← *problem*.INITIAL
  for $t = 1$ **to** ∞ **do**
    $T \leftarrow$ *schedule*(*t*)
    if $T = 0$ **then return** *current*
    next ← случайно выбранный преемник узла *current*
    $\Delta E \leftarrow$ VALUE(*current*) – VALUE(*next*)
    if $\Delta E > 0$ **then** *current* ← *next*
    else *current* ← *next*, но только с вероятностью $e^{-\Delta E/T}$

Рис. 4.5. Алгоритм имитации отжига: версия стохастического поиска восхождением на вершину, в которой разрешаются некоторые ходы вниз. Входная переменная *schedule* определяет значение "температуры" T как функции времени

Алгоритм поиска имитацией отжига широко использовался для решения задач компоновки СБИС уже с начала 1980-х годов. Кроме того, этот алгоритм нашел широкое применение при решении задач планирования производства и других крупномасштабных задач оптимизации.

4.1.3. Локальный поиск по лучу

Хранение в памяти только одного узла может показаться слишком радикальной реакцией на проблему ограничения памяти. Алгоритм ►**локального поиска по лучу** отслеживает k состояний вместо одного. Работа этого алгоритма начинается с генерации k состояний, выбранных случайным образом. Далее на каждом этапе формируются все преемники всех k состояний и, если любой из них соответствует целевому состоянию, алгоритм завершает работу. В противном случае из общего списка выбираются k лучших преемников и цикл повторяется.

На первый взгляд, локальный поиск по лучу с k состояниями представляет собой не более чем выполнение k случайных перезапусков, но не последовательно, а параллельно. Однако в действительности эти два подхода совершенно разные. При случайном перезапуске поиска каждый процесс поиска выполняется независимо от других. ➔ *При локальном поиске по лучу полезная информация передается между всеми параллельными поисковыми потоками.* Это подобно тому, как если бы состояния, имеющие лучших преемников, сообщали остальным: "Идите все сюда, здесь трава зеленее!" В результате алгоритм быстро прекращает неэффективные поиски и перемещает свои ресурсы туда, где был достигнут наибольший прогресс.

Локальный поиск по лучу может страдать от отсутствия разнообразия между k состояниями, если все они сосредоточатся в небольшой области пространства состояний. В результате поиск по лучу превратится в чуть более чем в k раз более медленный вариант поиска восхождением к вершине. Вариант с названием ►**стохастический поиск по лучу**, аналогичный стохастическому поиску восхождением к вершине, позволяет решить эту проблему. В этом случае вместо выбора k лучших преемников все преемники выбираются из общего пула с вероятностью, пропорциональной значению этого преемника, что увеличивает разнообразие.

4.1.4. Эволюционные алгоритмы

►**Эволюционные алгоритмы** можно рассматривать как варианты стохастического поиска по лучу, разработка которых очевидным образом была мотивирована концепцией естественного отбора в биологии. Здесь также имеется популяция из особей (состояния), в которой наиболее приспособленные особи (наибольшее значение) производят потомство (преемники состояния), образующее следующее поколение популяции, посредством процесса, называемого ►**рекомбинацией**. Существует бесконечное число форм эволюционных алгоритмов, различающихся следующими характеристиками.

- Размер популяции.
- Представление отдельной особи. В ►**генетических алгоритмах** каждая особь является строкой элементов из конечного алфавита (часто это битовая строка), так же, как молекула ДНК представляет собой строку элементов из алфавита A, C, G и T. В ►**эволюционных стратегиях** особи представляют

собой последовательность действительных чисел, а в ▶**генетическом программировании** особь — это компьютерная программа.

- Показатель смешивания ρ, определяющий число родителей, которые собираются вместе, чтобы сформировать потомство. Наиболее распространенным случаем является ρ = 2: два родителя объединяют свои "гены" (части своего представления), чтобы сформировать потомство. При ρ = 1 мы получим стохастический поиск по лучу (который можно рассматривать как бесполое размножение). Вполне возможно задать ρ > 2, что в природе встречается крайне редко, но достаточно легко моделируется на компьютерах.
- Процесс ▶**выбора** для рекомбинации особей, которые будут родителями для следующего поколения: один из вариантов состоит в выборе из популяции ρ особей с вероятностью, пропорциональной их **функции приспособленности**. Другой вариант — случайным образом выбрать в популяции n особей ($n > \rho$), а затем отобрать из них в качестве родителей ρ особей с наибольшей приспособленностью.
- Процедура рекомбинации. Наиболее общий подход (при условии ρ = 2) состоит в случайном выборе точки пересечения (▶**кроссинговер**), используемой для разделения каждой из родительских строк на две части, а затем объединения этих частей с целью формирования двух потомков. При этом один будет состоять из первой части родителя 1 и второй части родителя 2, а другой — из первой части родителя 2 и второй части родителя 1.
- ▶**Частота мутации**, которая определяет, как часто у потомков будут иметь место случайные мутации в их представлении. После того как потомок уже будет создан, каждый бит в его составе инвертируется с вероятностью, равной частоте мутации.
- Состав следующего поколения. В новое поколение может входить только вновь образованное потомство, либо оно может включать несколько родителей с наилучшей приспособленностью из предыдущего поколения (на практике этот механизм называют ▶**элитарностью**, гарантирующей, что для популяции ее общая приспособленность никогда не будет уменьшаться со временем). Практика ▶**отбраковки**, когда все особи с приспособленностью ниже установленного порога отбрасываются, может способствовать ускорению работы алгоритма (Баум и др. [141], 1995).

На рис. 4.6, *а* приведена совокупность из четырех 8-значных строк, каждая из которых представляет состояние в задаче с 8 ферзями: *с*-я цифра определяет номер строки с ферзем в столбце *с*. На рис. 4.10, *б* каждое состояние оценивается с использованием функции приспособленности. Более высокие значения приспособленности лучше, поэтому в этом примере для задачи о 8 ферзях в качестве критерия используется количество неатакующих друг друга пар ферзей, которое в любом решении будет равно $8 \times 7/2 = 28$. Значения этой функции для четырех приведенных на рисунке состояний равны 24, 23, 20 и 11. Далее оценка

приспособленности нормализуется к вероятности выбора — на рис. 4.6, *б* полученные значения приведены рядом с соответствующими оценками.

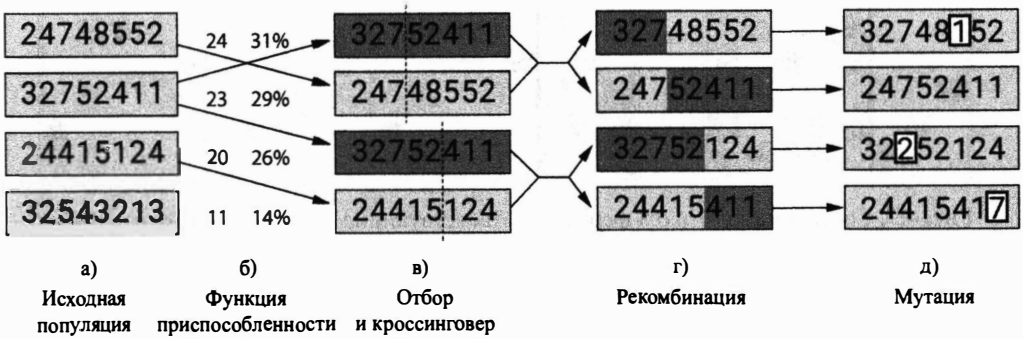


Рис. 4.6. Работа генетического алгоритма иллюстрируется с помощью цифровых строк, представляющих состояния в задаче с 8 ферзями. Начальная популяция **а** ранжируется по функции приспособленности **б**, после чего отбираются пары для скрещивания **в**. Эти пары производят потомство **г**, которое затем подвергается мутации **д**

На рис. 4.6, *в* из всей исходной популяции выбираются две пары родителей в соответствии с вероятностями в *б*. Обратите внимание, что одна особь выбирается дважды, а другая — не выбирается вовсе. К каждой выбранной паре применяется кроссинговер — случайным образом выбирается точка пересечения (пунктирная линия). На рис. 4.6, *г* для генерации потомков применяется рекомбинация: родительские строки обмениваются фрагментами, определяемыми точками пересечения. Например, первый потомок первой пары получает три первые цифры (327) от первого родителя, а остальные цифры (48552) — от второго родителя. Состояния задачи о 8 ферзях, образованные на этапе рекомбинации для первой пары родителей, показаны на рис. 4.7.

Наконец, на рис. 4.6, *д* каждый элемент в каждой строке подвергается случайной мутации с небольшой независимой вероятностью. В результате мутировало по одной цифре в первом, третьем и четвертом потомках. Для задачи о 8 ферзях подобная мутация означает случайный выбор ферзя на доске и его перемещение в случайно выбранную клетку того же столбца. Чаще всего на ранних этапах процесса популяция оказывается очень разнообразной, и процедура кроссинговера часто позволяет делать значительные перемещения в пространстве состояний в начале поиска (как в алгоритме имитации отжига). После селекции многих поколений в сторону повышения приспособленности популяция становится менее разнообразной, и типичными будут уже небольшие перемещения. На рис. 4.8 приведен алгоритм, реализующий выполнение всех описанных выше этапов.

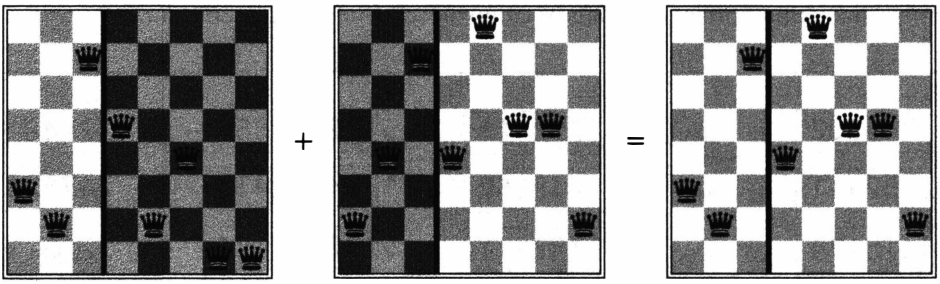


Рис. 4.7. Состояния задачи о 8 ферзях, соответствующие первым двум родителям на рис. 4.6, *в* и их потомку на рис. 4.6, *г*. Темные столбцы теряются на этапе кроссинговера, а светлые сохраняются в потомке. (Для правильной интерпретации чисел на рис. 4.6 уточним, что строка 1 — это нижний ряд, а строка 8 — верхний.)

```
function GENETIC-ALGORITHM(population, FITNESS) returns особь
  repeat
    weights ← WEIGHTED-BY(population, FITNESS)
    population2 ← пустой список
    for i = 1 to SIZE(population) do
      parent1, parent2 ← WEIGHTED-RANDOM-CHOICES(population, weights, 2)
      child ← REPRODUCE(parent1, parent2)
      if (небольшое случайно выбранное значение вероятности) then
        child ← MUTATE(child)
      add child to population2
    population ← population2
  until некоторый индивидуум не станет достаточно пригодным или не истечет
        установленное количество времени
  return наилучшая особь в популяции population, выбранная в соответствии
         со значением weights, определяемым функцией FITNESS

function REPRODUCE(parent1, parent2) returns особь
  n ← LENGTH(parent1)
  c ← случайно выбранное число в диапазоне от 1 до n
  return APPEND(SUBSTRING(parent1, 1, c), (SUBSTRING(parent2, c + 1, n))
```

Рис. 4.8. Генетический алгоритм. В этой функции популяция представлена как упорядоченный список особей *population*, переменная *weights* — это список соответствующих показателей приспособленности для каждой особи, а приспособленность определяется функцией FITNESS, вычисляющей эти значения

Эволюция и поиск

Теория эволюции была разработана Чарльзом Дарвином и представлена в его книге *Происхождение видов путем естественного отбора* ([527], 1859) независимо от Альфреда Рассела Уоллеса ([2288], 1858). Основная идея этой теории проста: при воспроизводстве возникают вариации, которые сохраняются в последующих поколениях приблизительно пропорционально их влиянию на способность носителей к воспроизводству.

Дарвин разрабатывал свою теорию, не зная о том, как те или иные свойства организма могут быть унаследованы и модифицированы. Вероятностные законы, управляющие этими процессами, были открыты Грегором Менделем ([1557], 1866), монахом, проводившим эксперименты со сладким горошком. Гораздо позже Уотсон и Крик ([2300], 1953) выявили структуру молекулы ДНК и ее алфавит "АГТЦ" (аминокислоты *аденин*, *гуанин*, *тимин*, *цитозин*). В предложенной ими стандартной модели вариации возникают и в результате точечных мутаций в последовательности этих аминокислот, и в результате "кроссинговера" (при котором ДНК потомка формируется путем объединения длинных секций ДНК от каждого из родителей).

Аналогия между этим процессом и алгоритмами локального поиска была описана выше; принципиальное различие между стохастическим поиском по лучу и эволюцией состоит в использовании полового воспроизводства, в котором потомки формируются с участием *нескольких* особей, а не только одного. Однако фактически механизмы эволюции намного богаче по сравнению с тем, что допускает большинство генетических алгоритмов. Например, мутации могут быть связаны с обращением, дублированием и перемещением больших фрагментов ДНК; некоторые вирусы заимствуют ДНК из одного организма и вставляют в другой; кроме того, существуют взаимозаменяемые гены, которые лишь копируют самих себя в геноме много тысяч раз.

Есть даже такие гены, которые отравляют клетки потенциальных родителей, не несущие ген этого типа, повышая тем самым шансы на собственное воспроизводство. Наиболее важным является тот факт, что *гены сами кодируют те механизмы*, с помощью которых геном воспроизводится и транслируется в организме. В генетических алгоритмах такие механизмы представляют собой отдельную программу, не закодированную в строках, с которыми осуществляются манипуляции.

Дарвиновская эволюция может показаться неэффективной, поскольку на Земле она вслепую породила примерно 10^{43} или около того организмов и при этом ни на йоту не улучшила свою поисковую эвристику. Но обучение все же играет определенную роль в эволюции. Хотя великий во всем остальном французский натуралист Жан Ламарк ([1347], 1809) и ошибался, предположив, что признаки, приобретенные путем адаптации на протяжении жизни организма, будут переданы потомству, внешне похожая теория Джеймса Болдуина ([118], 1896) оказалась правильной: обучение может эффективно сглаживать кривую функции приспособленности, способствуя повышению скорости эволюции. Организм, имеющий особенность, не вполне соответствующую его среде обитания, сможет передать эту особенность потомкам, если он также обладает достаточной пластичностью, чтобы научиться адаптироваться к окружающей среде подходящим способом. Компьютерное моделирование (Хинтон и Ноулан [1026], 1987) подтвердило, что **эффект Болдуина** имеет место в реальности и что как следствие то, чему трудно научиться, попадает в геном, а то, чему научиться легко, не должно в нем фиксироваться (Морган и Гриффитс [1624], 2015).

Генетические алгоритмы в целом похожи на стохастический поиск по лучу, но с добавлением операции кроссинговера. Она дает преимущества, если в состояниях есть блоки элементов, определяющие выполнение полезных функций. Например, в задаче о 8 ферзях может быть полезно, чтобы в первых трех столбцах ферзи находились в строках 2, 4 и 6 (где они не нападают друг на друга), — это будет полезный блок, который для получения решения может быть объединен с иными полезными блоками, присутствующими в других особях. Можно показать математически, что если блоки элементов не служат полезной цели, например если позиции генетического кода просто переставляются случайным образом, использование кроссинговера не дает никаких преимуществ.

Теория генетических алгоритмов объясняет, как это работает, используя идею ►**схемы**, которая представляет собой строку, в которой некоторые позиции могут оставаться неопределенными. Например, схема 246***** описывает все состояния задачи о 8 ферзях, в которых первые три ферзя находятся в позициях 2, 4 и 6 соответственно. Строки, которые соответствуют схеме (например, 24613578), называются ►**примерами** схемы. Можно показать, что если средняя пригодность примеров схемы выше среднего, то число примеров схемы со временем будет расти.

Очевидно, что данный эффект вряд ли окажется существенным, если смежные биты абсолютно не связаны друг с другом, поскольку в таком случае будет существовать мало непрерывных блоков, предоставляющих какое-либо постоянное преимущество. Генетические алгоритмы работают лучше всего в сочетании со схемами, соответствующими осмысленным компонентам решения. Например, если строка представляет какую-либо радиотехническую антенну, то схемы могут соответствовать компонентам этой антенны, таким как рефлекторы и дефлекторы. Хороший компонент, по всей вероятности, будет оставаться хорошим во многих разных проектах. Из этого следует, что для успешного использования генетических алгоритмов требуются тщательное обдумывание и проработка представления задачи.

На практике генетические алгоритмы нашли свое место в широком спектре методов оптимизации (Марлер и Арора [1495], 2004), особенно в случае достаточно сложных структурированных задач, таких как компоновка интегральных схем или планирование работы магазина, а в последнее время и в развитии архитектуры глубоких нейронных сетей (Миккулайнен и др. [1567], 2019). Пока не ясно, насколько привлекательность генетических алгоритмов обусловлена их превосходством в решении определенных классов задач, а насколько — подкупающей метафорой эволюции.

4.2. Локальный поиск в непрерывных пространствах

В главе 2 мы объяснили различие между дискретной и непрерывной средами, указав, что большинство реальных сред являются непрерывными. Непрерывное пространство действий имеет бесконечно большой коэффициент ветвления, а следовательно, действовать в ней не способен ни один из уже рассмотренных ранее

алгоритмов (за исключением алгоритмов поиска восхождением к вершине с выбором первого варианта и имитации отжига).

В этом разделе дано *очень краткое* введение в некоторые методы локального поиска в непрерывных пространствах. Литература по этой теме обширна; многие из этих основных методов были созданы еще в XVII веке после разработки первых математических исчислений Ньютоном и Лейбницем.[2] Применение этих методов обсуждается в нескольких главах данной книги, включая главы, касающиеся обучения, машинного зрения и робототехники.

Начнем с рассмотрения примера. Предположим, что в Румынии требуется найти место для размещения трех новых аэропортов, причем таким образом, чтобы сумма квадратов расстояний от каждого города на карте (см. рис. 3.1) до ближайшего к нему аэропорта была минимальной. В этой задаче пространство состояний определяется координатами трех аэропортов, (x_1, y_1), (x_2, y_2) и (x_3, y_3), и является *шестимерным*, а следовательно, каждое состояние будет определяться шестью ►**переменными**. В общем случае состояния определяются n-мерным вектором из переменных, **x**. Перемещение в этом пространстве соответствует перемещению одного или более из трех рассматриваемых аэропортов на карте Румынии. Для любого конкретного состояния вычислить целевую функцию $f(\mathbf{x}) = f(x_1, y_1, x_2, y_2, x_3, y_3)$ будет относительно легко, как только будут определены ближайшие города. Пусть C_i будет множеством городов, для которых ближайшим аэропортом (в состоянии **x**) является аэропорт i. Тогда мы имеем следующее:

$$f(x) = f(x_1, y_1, x_2, y_2, x_3, y_3) = \sum_{i=1}^{3} \sum_{c \in Ci} (x_i - x_c)^2 + (y_i - y_c)^2 . \qquad (4.1)$$

Это уравнение верно не только для состояния **x**, но и для состояний в локальной окрестности **x**. Тем не менее оно не будет корректным глобально: если удалиться от состояния **x** слишком далеко, значительно изменив расположение одного или более аэропортов, то множество ближайших городов для этих аэропортов может измениться, и потребуется пересчитать значения C_i.

Один из способов устранения необходимости иметь дело с непрерывными пространствами состояний состоит в их ►**дискретизации**. Например, вместо того чтобы позволить местоположению (x_i, y_i) быть любой точкой в непрерывном двухмерном пространстве, можно ограничить их значения отдельными точками, образующими прямоугольную решетку с шагом δ (дельта). Тогда вместо бесконечного числа преемников каждое состояние в пространстве будет иметь только 12 преемников, каждый из которых соответствует приращению одной из 6 переменных на ± δ. В результате появляется возможность использовать в подобном дискретном пространстве любой из алгоритмов локального поиска, описанных выше. Другой подход — сделать коэффициент ветвления конечным путем случайной выборки состояний-преемников, перемещаясь в случайном направлении на небольшую

[2] Для этого раздела необходимо как минимум знакомство с векторами, матрицами и производными (см. приложение А).

величину δ. Методы, в которых прогресс определяется по изменению значения целевой функции между двумя соседними точками, называются методами ▶**эмпирического градиента**. Поиск по эмпирическому градиенту — это то же самое, что и поиск восхождением к вершине по самому крутому склону в дискретизированном пространстве состояний. Уменьшая значение δ с течением времени, можно получить более точное решение, но при этом схождение к глобальному оптимуму в пределе не гарантируется.

Если есть целевая функция, выраженная в математической форме, то можно попробовать решить проблему аналитически, а не эмпирически. Существует много методов, предполагающих использование для поиска максимума ▶**градиента** ландшафта. Градиент целевой функции представляет собой вектор ∇f, позволяющий определить величину и направление наиболее крутого склона. Для рассматриваемой задачи справедливо следующее соотношение:

$$\nabla f = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial y_1}, \frac{\partial f}{\partial x_2}, \frac{\partial f}{\partial y_2}, \frac{\partial f}{\partial x_3}, \frac{\partial f}{\partial y_3} \right).$$

В некоторых случаях можно найти максимум, решая уравнение $\nabla f = 0$, но во многих случаях это уравнение не может быть решено в замкнутой форме. Например, при наличии трех аэропортов выражение для градиента зависит от того, какие города являются ближайшими к каждому аэропорту в текущем состоянии. Это означает, что мы можем вычислить этот градиент *локально*, но не *глобально*, например с помощью следующей формулы:

$$\frac{\partial f}{\partial x_1} = 2 \sum_{c \in C_1} (x_1 - x_c). \tag{4.2}$$

Имея локально правильное выражение для градиента, можно воспользоваться методом восхождения на вершину по самому крутому склону посредством обновления текущего состояния в соответствии с формулой

$$\mathbf{x} \leftarrow \mathbf{x} + \alpha \nabla f(\mathbf{x}),$$

где α (альфа) — небольшая константа, часто называемая ▶**размером шага**. Существует огромное разнообразие методов подбора величины α. Основная проблема состоит в том, что, если значение α будет слишком мало, потребуется слишком много этапов; а если значение α будет слишком велико, поиск может "перепрыгнуть" через максимум. Попытка преодолеть эту дилемму предпринимается в методе ▶**линейного поиска**, предусматривающем продолжение поиска в направлении текущего градиента (обычно путем многократного удвоения) до тех пор, пока целевая функция f не начнет вновь уменьшаться. Точка, в которой это происходит, становится новым текущим состоянием. Сформировалось несколько научных школ, в которых доминируют разные взгляды на то, каким образом в этой точке следует выбирать новое направление.

Для многих задач наиболее эффективным алгоритмом является почтенный метод ▶**Ньютона–Рафсона**. Это общий метод поиска корней функций, т.е. метод решения уравнений в форме $g(x) = 0$. Этот алгоритм действует посредством вычисления новой оценки для корня x в соответствии с формулой Ньютона:

$$x \leftarrow x - g(x)/g'(x).$$

Чтобы найти максимум или минимум f, необходимо найти такое значение x, для которого *градиент* равен нулевому вектору (т.е. $\nabla f(\mathbf{x}) = \mathbf{0}$). Поэтому функция $g(x)$ в формуле Ньютона принимает вид $\nabla f(\mathbf{x})$ и уравнение обновления состояния может быть записано в матрично-векторной форме следующим образом:

$$\mathbf{x} \leftarrow \mathbf{x} - \mathbf{H}_f^{-1}(\mathbf{x})\nabla f(\mathbf{x}),$$

где $\mathbf{H}_f(\mathbf{x})$ представляет собой ▶**гессиан** (матрицу вторых производных), элементы которого H_{ij} задаются выражением $\partial^2 f/\partial x_i \partial x_j$. Например, из уравнения (4.2) для нашего аэропорта мы можем видеть, что гессиан $\mathbf{H}_f(\mathbf{x})$ является относительно простым: его недиагональные элементы являются нулями, а диагональные элементы для аэропорта i — это просто удвоенное число городов в Ci. Расчет момента показывает, что один этап обновления перемещает аэропорт i непосредственно в центр тяжести C_i, который является минимумом локального выражения для f из уравнения (4.1).[3] Однако для задач большой размерности вычисление n^2 элементов гессиана и его инвертирование могут оказаться дорогостоящими, поэтому было разработано множество приближенных версий метода Ньютона–Рафсона.

Локальные максимумы, хребты и плато создают затруднения в работе методов локального поиска не только в дискретных, но и в непрерывных пространствах состояний. Для преодоления этих затруднений могут применяться алгоритмы с перезапуском случайным образом и имитации отжига, которые часто оказываются достаточно полезными. Тем не менее многомерные непрерывные пространства характеризуются очень большим объемом, в котором легко потеряться.

Последней темой является ▶**оптимизация при ограничениях**. Задача оптимизации называется задачей оптимизации при ограничениях, если ее решения должны удовлетворять некоторым жестким ограничениям на значения каждой переменной. Например, в рассматриваемой задаче с размещением аэропортов можно ввести ограничения, чтобы места для аэропортов находились в пределах Румынии, причем на суше (а не, допустим, в середине озера). Сложность задач оптимизации при ограничениях зависит от характера ограничений и от целевой функции. Наиболее широко известной категорией таких задач являются задачи ▶**линейного программирования**, в которых ограничения должны представлять собой

[3] В общем случае обновление по методу Ньютона–Рафсона можно рассматривать как подгонку квадратичной поверхности к функции f в точке $\mathbf{x}$, а затем как перемещение непосредственно к минимуму этой поверхности, что также является минимумом для f, если функция f квадратична.

линейные неравенства, образующие ▶**выпуклое множество**,[4] а целевая функция также является линейной. Задачи линейного программирования могут быть решены за время, полиномиально зависящее от количества переменных.

Линейное программирование, вероятно, является наиболее широко изученным и наиболее полезным методом оптимизации. Оно является особым случаем в более общей задаче ▶**выпуклой оптимизации**, в которой ограничивающая область может быть любой выпуклой областью, а целевая функция может быть любой областью, являющейся выпуклой в пределах области ограничений. При определенных условиях задачи выпуклой оптимизации также разрешимы за полиномиальное время и на практике могут допускать наличие тысяч переменных. Несколько важных задач в теории машинного обучения и управления могут быть сформулированы как задачи выпуклой оптимизации (см. главу 20).

4.3. Поиск с недетерминированными действиями

В главе 3 проблемная среда предполагалась полностью наблюдаемой, детерминированной и известной. В такой среде агент может воспринять и изучить исходное состояние, вычислить последовательность действий, приводящих к цели, а затем выполнять эти действия с "закрытыми глазами", больше не пользуясь своим восприятием.

Однако, когда среда лишь частично наблюдаема, агент не знает наверняка, в каком состоянии он находится; а если среда еще и недетерминированная, агент не знает и того, в какое состояние он перейдет после выполнения некоторого действия. Это означает, что вместо того, чтобы утверждать: "я нахожусь в состоянии s_1, и если я выполню действие a, то окажусь в состоянии s_2", агент теперь будет вынужден полагать: "я в состоянии либо s_1, либо s_3, и если я выполню действие a, я окажусь в состоянии s_2, s_4 или s_5". Назовем множество физических состояний, которые, по мнению агента, являются возможными, ▶**доверительным состоянием**.

В частично наблюдаемых и недетерминированных средах решение задачи не является больше некоторой последовательностью действий; скорее это ▶**условный план** (иногда называемый планом действий в непредвиденных обстоятельствах или стратегией), который определяет, что следует делать в зависимости от того, какие восприятия поступают агенту во время выполнения плана. В этом разделе будет рассмотрено поведение агента в недетерминированной среде, а в следующем — в частично наблюдаемой.

[4] Множество точек S является выпуклым, если линии, соединяющие его любые две точки, также содержится в S. **Выпуклая функция** является такой функцией, для которой пространство "над" ней образует выпуклое множество. По определению выпуклые функции не имеют локальных (в отличие от глобальных) минимумов.

4.3.1. Непредсказуемый мир пылесоса

Мир пылесоса, рассматривавшийся в главе 2, имел восемь возможных состояний, как показано на рис. 4.9. В нем возможны три действия — *Right* (вправо), *Left* (влево) и *Suck* (убрать), — а целью является уборка всего мусора (состояния 7 и 8). Если среда полностью наблюдаема, детерминирована и известна, то задача легко решается с помощью любого из алгоритмов, обсуждавшихся в главе 3, и решение представляет собой четкую последовательность действий. Например, если исходным является состояние 1, то выполнение последовательности действий [*Suck*, *Right*, *Suck*] приведет мир в целевое состояние 8.

Теперь предположим, что пылесос в этом мире стал более мощным, но при этом иногда может вести себя неожиданным образом, создавая непредвиденные ситуации, а значит, этот мир стал *недетерминированным*. В **непредсказуемом мире пылесоса** действие *Suck* работает следующим образом.

- При выполнении в квадрате с мусором это действие убирает мусор в данном квадрате, а иногда еще и в соседнем.
- При выполнении в чистом квадрате после этого действия на ковре иногда остается немного мусора.[5]

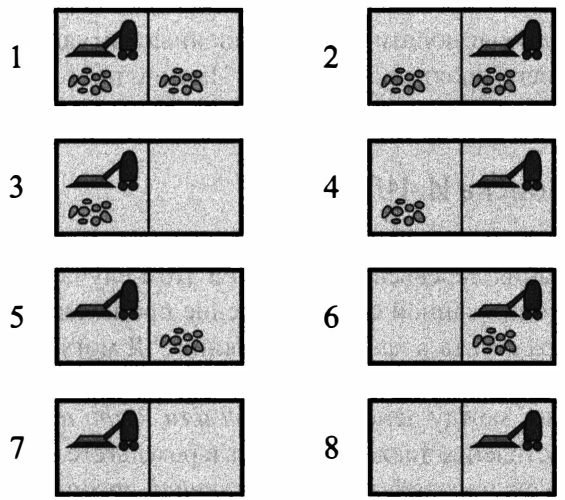


Рис. 4.9. Восемь возможных состояний мира пылесоса, где состояния 7 и 8 являются целевыми

[5] Авторы предполагают, что большинство читателей сталкивались с подобными проблемами и могут посочувствовать нашему агенту. Они приносят свои извинения тем владельцам современных и эффективных чистящих приборов, которые не могут оценить преимуществ использования этого педагогического устройства.

Для того чтобы предоставить более точную формулировку этой задачи, необходимо обобщить понятие **функции определения преемника** из главы 3. Вместо определения этого понятия как функции RESULT, всегда возвращающей *одно* результирующее состояние, мы теперь определим его как функцию RESULTS, возвращающую *множество возможных* результирующих состояний. Например, в мире непредсказуемого пылесоса действие *Suck* в состоянии 1 может убрать мусор либо только в текущем местоположении, либо сразу в обоих местоположениях:

$$\text{RESULTS}(1, \textit{Suck}) = \{5, 7\}.$$

Таким образом, если исходным является состояние 1, нет единственной фиксированной *последовательности* действий, позволяющей решить задачу, но следующий **условный план** позволяет это сделать:

$$[\textit{Suck}, \textbf{if}\ \textit{State} = 5\ \textbf{then}\ [\textit{Right}, \textit{Suck}]\ \textbf{else}\ [\,]]. \tag{4.3}$$

Отсюда видно, что условный план может содержать этапы **if–then–else**; а это означает, что решения являются *деревьями*, а не последовательностями. В этом случае условие в операторе **if** предназначено для проверки, каким является текущее состояние, оно должно представлять собой что-то, что агент может наблюдать во время выполнения плана, но еще не знал при его составлении. В качестве альтернативы можно предложить формулировку, использующую проверку восприятия агента, а не состояния. Многие задачи в реальном, физическом мире являются задачами, предполагающими непредвиденные ситуации, поскольку точное предсказание будущего просто невозможно. Именно поэтому многие люди даже во время прогулки внимательно следят за происходящим вокруг.

4.3.2. Деревья поиска И–ИЛИ

Как же найти эти условные решения недетерминированных задач? Как и в главе 3, начнем с создания деревьев поиска, но в этом случае деревья имеют иной характер. В детерминированной среде ветвление происходит только в результате собственного выбора агента в каждом состоянии: "Я могу выполнить это *или* то действие". Назовем такие узлы ►**ИЛИ-узлами**. Например, в мире пылесоса агент в ИЛИ-узле выбирает между действиями *Left* ***или*** *Right,* ***или*** *Suck*. В недетерминированной среде ветвление также возникает в результате выбора, но теперь уже выбора результата каждого действия *окружающей средой*. Назовем такие узлы ►**И-узлами**. Например, выполнение действия *Suck* в состоянии 1 дает результат в виде доверительного состояния {5, 7}, поэтому агент должен сформировать условный план для состояний 5 ***и*** 7. Эти два вида узлов чередуются, приводя к образованию ►**И–ИЛИ-дерева**, как показано на рис. 4.10.

Решением задачи поиска в дереве И–ИЛИ является такое поддерево в полном дереве поиска, в котором 1) каждый лист является *целевым* узлом; 2) выбирается *одно* действие в каждом из его ИЛИ-узлов; и 3) включена *каждая* исходящая ветвь

во всех его И-узлах. Решение, представленное на рис. 4.10 утолщенными линиями, соответствует плану, приведенному в уравнении (4.3).

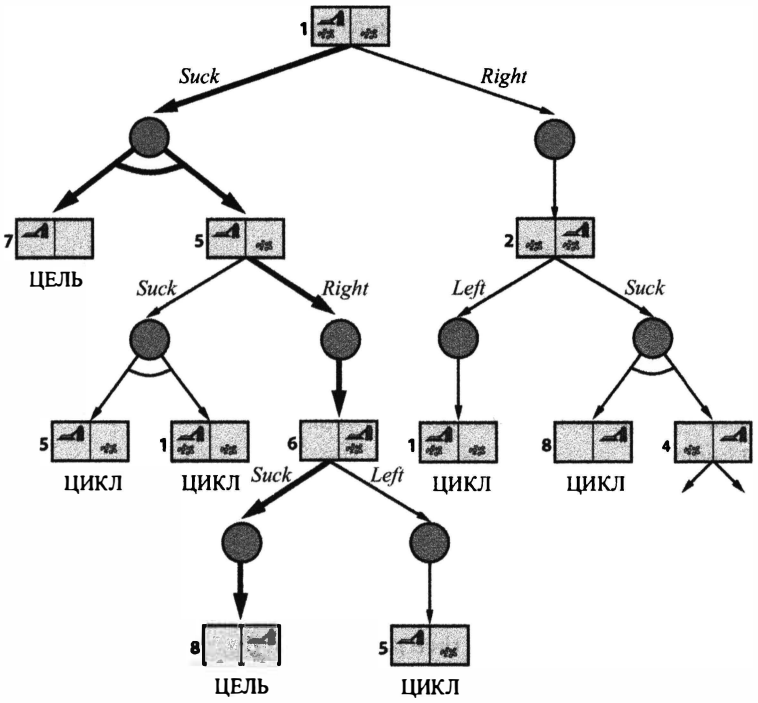


Рис. 4.10. Первые два уровня дерева поиска для непредсказуемого мира пылесоса. Узлы состояния являются ИЛИ-узлами, в которых агент должен выбрать какое-то действие. В И-узлах, показанных на рисунке в виде кружков, каждый возможный результат действия должен быть обработан, что подчеркивается дугами, соединяющими исходящие их этих узлов ветви. Пути к решению показаны утолщенными линиями

На рис. 4.11 представлен рекурсивный алгоритм поиска в глубину, реализующий поиск по графу И–ИЛИ. Ключевым аспектом этого алгоритма является способ, используемый для работы с циклами, часто возникающими в недетерминированных задачах (например, если действие иногда не дает никакого результата или если непреднамеренно полученный эффект может быть исправлен). Если текущее состояние идентично некоторому состоянию на пути к нему от корня, то алгоритм возвращает ошибку. Это не означает, что для текущего состояния *не существует* решения; это лишь означает, что если нециклическое решение существует, оно должно быть достижимо из предыдущего воплощения текущего состояния, а значит, его новое воплощение может быть отброшено. Благодаря этой

проверке можно гарантировать, что алгоритм обязательно завершит свою работу в любом конечном пространстве состояний, поскольку каждый путь должен либо достичь цели, либо привести в тупик, либо вернуться к уже пройденному состоянию. Обратите внимание, что алгоритм не проверяет, является ли текущее состояние повторением состояния на каком-то ином пути от корня, что очень важно для его эффективности.

```
function AND-OR-SEARCH(problem) returns условный план или failure
  return OR-SEARCH(problem, problem.INITIAL, [])

function OR-SEARCH(problem, state, path) returns условный план или failure
  if problem.IS-GOAL(state) then return пустой план
  if IS-CYCLE(path) then return failure
  for each action in problem.ACTIONS(state) do
      plan ← AND-SEARCH(problem, RESULTS(state, action), [state] + path])
      if plan ≠ failure then return [action] + plan]
  return failure

function AND-SEARCH(problem, states, path) returns условный план или failure
  for each s_i in states do
      plan_i ← OR-SEARCH(problem, s_i, path)
      if plan_i = failure then return failure
  return [if s_1 then plan_1 else if s_2 then plan_2 else ... if s_{n-1} then plan_{n-1} else plan_n]
```

Рис. 4.11. Алгоритм поиска в графах И–ИЛИ, генерируемых в недетерминированных средах. Решение представляет собой условный план, учитывающий каждый недетерминированный результат и включающий план для каждого из них

Для поиска в графах И–ИЛИ могут также использоваться алгоритмы поиска в ширину и жадного поиска по первому наилучшему совпадению. Что касается концепции эвристической функции, то ее потребуется модифицировать так, чтобы она оценивала стоимость условного решения, а не последовательности действий, но при этом требование о ее допустимости сохраняет свою силу. Отметим, что существует аналог алгоритма поиска A*, позволяющий находить оптимальные решения (см. библиографические примечания в конце главы).

4.3.3. Попробуйте, попробуйте и еще раз попробуйте

Рассмотрим *ненадежный* мир пылесоса, который ничем не отличается от обычного (не непредсказуемого) мира пылесоса за исключением того, что действия перемещения иногда терпят неудачу и агент остается на том же месте. В таком мире, например, действие *Right* в состоянии 1 приводит к доверительному состоянию

{1, 2}. На рис. 4.12 показана часть графа поиска для этого мира. Очевидно, что в этом случае больше нет никаких нециклических решений из состояния 1, а следовательно, И–ИЛИ-поиск никогда не завершится успешно. Существует, однако, ►**циклическое решение**, которое заключается в том, чтобы продолжать попытки выполнить действие *Right*, пока оно не сработает. Это можно выразить с помощью новой конструкции **while**:

$$[Suck, \textbf{while } State = 5 \textbf{ do } Right, Suck]$$

или добавив метку для обозначения некоторой части плана, а затем сославшись на эту метку:

$$[Suck, L_1 : Right, \textbf{if } State = 5 \textbf{ then } L_1 \textbf{ else } Suck].$$

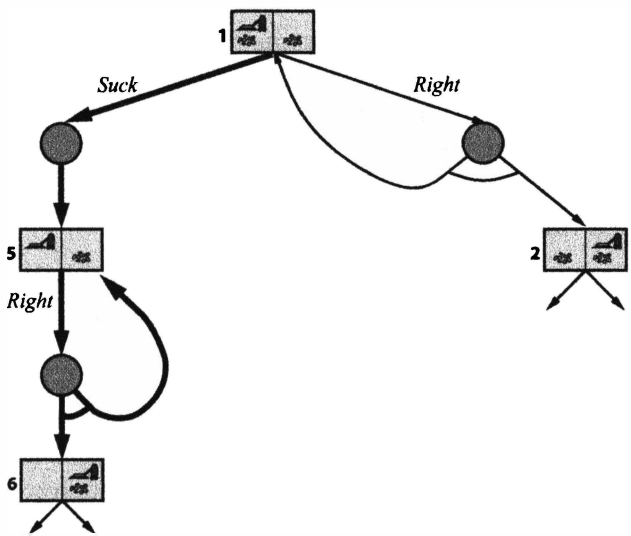


Рис. 4.12. Часть поискового графа для ненадежного мира пылесоса, на которой наглядно представлены (некоторые) возможные циклы. Все решения этой задачи — циклические планы, поскольку надежного пути в этом мире не существует

Когда циклический план является решением? Минимальное условие состоит в том, что каждый листовой узел дерева поиска должен представлять целевое состояние и при этом должен быть достижимым из любой точки в плане. В дополнение к этому следует рассмотреть причину недетерминированности среды. Если это условие возникает из-за того, что механизм передвижения робота-пылесоса бо́льшую часть времени успешно работает, но иногда случайным образом и по независящей от робота причине проскальзывает, то агент может быть уверен, что если повторять действие достаточное число раз, то в конце концов оно сработает и план

будет выполнен. Но если недетерминированность возникает из-за какого-то ненаблюдаемого события в окружающей робота среде, — например, лопнул ремень привода и робот вообще утратил возможность перемещаться, — то простое повторение действия ему уже не поможет.

Один из вариантов пояснить последнее утверждение — указать на то, что в этой ситуации от исходной формулировки задачи (полностью наблюдаемая, недетерминированная) приходится отказаться в пользу другой формулировки (частично наблюдаемая, детерминированная), при которой провал циклического плана приписывается ненаблюдаемому свойству приводного ремня. В главе 12 мы обсудим, как прийти к заключению, какая из нескольких неопределенных возможностей более вероятна.

4.4. Поиск в частично наблюдаемых средах

Теперь мы обратимся к задачам с частичной наблюдаемостью, когда восприятия агента недостаточно, чтобы получить точное представление о состоянии. Для нас это означает, что некоторые действия агента будут направлены на уменьшение неопределенности в отношении текущего состояния.

4.4.1. Поиск без наблюдения

Когда восприятие агента *вообще не предоставляет ему информации*, мы оказываемся в ситуации, которая называется ►**бессенсорной задачей** (или задачей ►**соответствия**). На первый взгляд может показаться, что у бессенсорного агента нет шансов найти решение поставленной ему задачи, если он не имеет представления о том, в каком состоянии начинает действовать, тем не менее бессенсорные решения на удивление распространены и полезны, в первую очередь потому, что они *не полагаются* на правильно работающие датчики. Например, в промышленных системах разработано множество изобретательных методов правильной ориентации деталей из неизвестной начальной позиции посредством использования некоторой последовательности действий вообще без какого-либо контроля. Иногда бессенсорный план оказывается лучше даже при наличии условного плана и возможности анализа ситуации. Например, врачи часто назначают пациентам антибиотик широкого спектра действия вместо условного плана, предусматривающего взятие анализа крови, ожидание его результатов и последующее назначение антибиотика направленного действия. Бессенсорный план экономит время и деньги, а также снижает риск ухудшения состояния больного, пока будут доступны результаты анализов.

Рассмотрим бессенсорную версию (детерминированного) мира пылесоса. Предположим, что агент знает географию мира, но не имеет сведений о своем местоположении и распределении в нем мусора. В этом случае его начальное доверительное состояние будет {1, 2, 3, 4, 5, 6, 7, 8} (см. рис. 4.9). Теперь, если агент

выполнит действие *Right*, он окажется в одном из состояний {2, 4, 6, 8}, — как видите, агент получил некоторую информацию без восприятия чего-либо! Выполнив последовательность действий [*Right*, *Suck*], агент всегда будет находиться в одном из состояний {4, 8}. И наконец, выполнив последовательность действий [*Right*, *Suck*, *Left*, *Suck*], он гарантированно достигнет целевого состояния 7 *независимо* от его начального состояния. В этом случае говорят, что агент может ▶**принудительно перевести** мир в состояние 7.

Для бессенсорной задачи решение представляет собой последовательность действий, а не условный план (ведь у агента нет ничего воспринимающего), но поиск проводится в пространстве доверительных, а не физических состояний.[6] В пространстве доверительных состояний задача *полностью наблюдаема*, поскольку агент всегда знает свое доверительное состояние. Более того, для бессенсорной задачи решение (если таковое имеется) будет представлять собой последовательность действий. Так происходит потому, что, как и в обычных задачах из главы 3, восприятие, полученное после каждого действия, будет полностью предсказуемым — оно всегда будет пустым! Следовательно, оно свободно от любых непредвиденных обстоятельств, которые необходимо учитывать в плане. И это справедливо, *даже если среда является недетерминированной.*

Можно было бы разработать новые алгоритмы специально для бессенсорных задач поиска. Тем не менее вполне возможно использовать и существующие алгоритмы, обсуждавшиеся в главе 3, если трансформировать лежащую в их основе физическую задачу в задачу доверительных состояний, когда поиск ведется среди доверительных, а не физических состояний. Если физическая задача (обозначим ее как P) имела такие компоненты, как действия $Actions_P$, результаты $Result_P$ и другие, то задача доверительных состояний будет иметь следующие компоненты.

- **Состояния**. Пространство доверительных состояний содержит все возможные подмножества существующих физических состояний. Если задача P имеет N состояний, то задача доверительных состояний будет иметь 2^N доверительных состояний, хотя многие из них могут оказаться недоступными из исходного состояния.
- **Начальное состояние**. Как правило, в начальном состоянии доверительное состояние будет включать все возможные состояния задачи P, хотя в некоторых случаях агент может иметь больше знаний, чем в простейшем случае.
- **Действия**. Здесь все сложнее. Предположим, что агент находится в доверительном состоянии $b = \{s_1, s_2\}$, но $\text{ACTIONS}_P(s_1) \neq \text{ACTIONS}_P(s_2)$. Тогда агент не знает, какое из действий будет допустимым. Если предположить, что недопустимые действия не оказывают влияния на окружающую среду, то будет

[6] В полностью наблюдаемой среде каждое доверительное состояние содержит одно физическое состояние. Следовательно, алгоритмы из главы 3 можно рассматривать как поиск в пространстве доверительных состояний, образованных из одноэлементных доверительных состояний.

безопасно взять *объединение* всех действий в любом из физических состояний в текущем доверительном состоянии b:

$$\text{ACTIONS}(b) = \bigcup_{s \in b} \text{ACTIONS}_P(s).$$

С другой стороны, если недопустимое действие может привести к катастрофе, то безопаснее разрешить только *пересечение*, т.е. набор действий, допустимых во *всех* состояниях. Для мира пылесоса каждое состояние имеет одинаковый набор допустимых действий, поэтому оба эти метода дают один и тот же результат.

- **Функция определения преемника**. Для действий в детерминированной среде новое доверительное состояние будет включать по одному результирующему состоянию для каждого из состояний, возможных на текущий момент (хотя некоторые результирующие состояния могут оказаться одними и теми же):

$$b' = \text{RESULT}(b, a) = \{s' : s' = \text{RESULT}_P(s, a) \text{ и } s \in b\}. \quad (4.4)$$

В недетерминированной среде новое доверительное состояние включает все возможные результаты применения действия к любому из состояний в текущем доверительном состоянии:

$$b' = \text{RESULT}(b, a) = \{s' : s' \in \text{RESULTS}_P(s, a) \text{ и } s \in b\} = \\ = \bigcup_{s \in b} \text{RESULTS}_P(s, a).$$

Для действий в детерминированной среде новое доверительное состояние b' по размеру будет таким же или меньше, чем b, но может оказаться больше b для действий в недетерминированной среде (рис. 4.13).

- **Проверка цели**. Агент, *возможно*, достигает цели, если *некоторое* состояние s в доверительном состоянии дает удовлетворительный результат при выполнении проверки достижения целевого состояния основной задачи, $\text{IS-GOAL}_P(s)$. Агент *обязательно* достигает цели, если *каждое* состояние удовлетворяет $\text{IS-GOAL}_P(s)$. Мы стремимся обязательно достичь цели.
- **Стоимость действия**. Это тоже непростой вопрос. Если одно и то же действие может иметь разную стоимость в разных состояниях, то стоимость выполнения действия в заданном доверительном состоянии может иметь несколько разных значений. (Это порождает новый класс задач, который рассматривается в упражнении 4.7.) Пока мы будем полагать, что стоимость действия является одной и той же во всех состояниях и поэтому может быть взята непосредственно из базовой физической задачи.

На рис. 4.14 показано пространство достижимых доверительных состояний для детерминированного бессенсорного мира пылесоса. Существует только 12 достижимых доверительных состояний из $2^8 = 256$ возможных доверительных состояний.

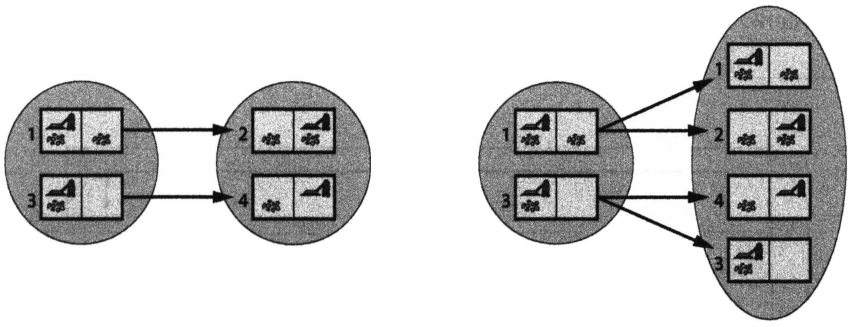


Рис. 4.13. а) Прогноз следующего доверительного состояния для бессенсорного мира пылесоса при выборе детерминированного действия *Right*. **б)** Прогноз для того же самого доверительного состояния и действия в ненадежном бессенсорном мире пылесоса

Предыдущие определения позволяют автоматически строить формулировку задачи доверительных состояний из определения лежащей в основе физической задачи. Как только это будет сделано, мы сможем решать бессенсорные задачи с помощью любого из обычных алгоритмов поиска, обсуждавшихся в главе 3.

При обычном поиске по графу вновь достигнутые состояния проверяются, чтобы определить, были ли они достигнуты ранее. Это правило работает и в случае доверительных состояний. Например, на рис. 4.14 последовательность действий [*Suck*, *Left*, *Suck*], выполненная из начального состояния, приводит к тому же доверительному состоянию, что и последовательность [*Right*, *Left*, *Suck*], а именно {5, 7}. А теперь рассмотрим доверительное состояние, достигаемое после выполнения действия [*Left*], а именно {1, 3, 5, 7}. Очевидно, что оно не идентично состоянию {5, 7}, но это его *надмножество*. Мы можем отбросить (обрезать) любое такое доверительное состояние-надмножество. Почему? Потому что решение, достигнутое из состояния {1, 3, 5, 7}, должно быть решением и для каждого из отдельных состояний 1, 3, 5 или 7, а значит, оно является решением и для любой комбинации этих отдельных состояний, например {5, 7}. Для нас это означает, что нет необходимости пытаться решить задачу для доверительного состояния {1, 3, 5, 7} и можно сосредоточиться на попытке решить задачу для существенно более простого доверительного состояния {5, 7}.

И наоборот, если доверительное состояние {1, 3, 5, 7} уже было сгенерировано и признано разрешимым, то любое *подмножество* его элементов, например {5, 7}, также гарантированно будет разрешимым. (Если есть решение, применимое для состояния, вызвавшего у меня значительное беспокойство, оно будет применимо и для состояний, вызывающих меньшее беспокойство.) Такой дополнительный

уровень обрезки может существенно повысить эффективность решения бессенсорной задачи.

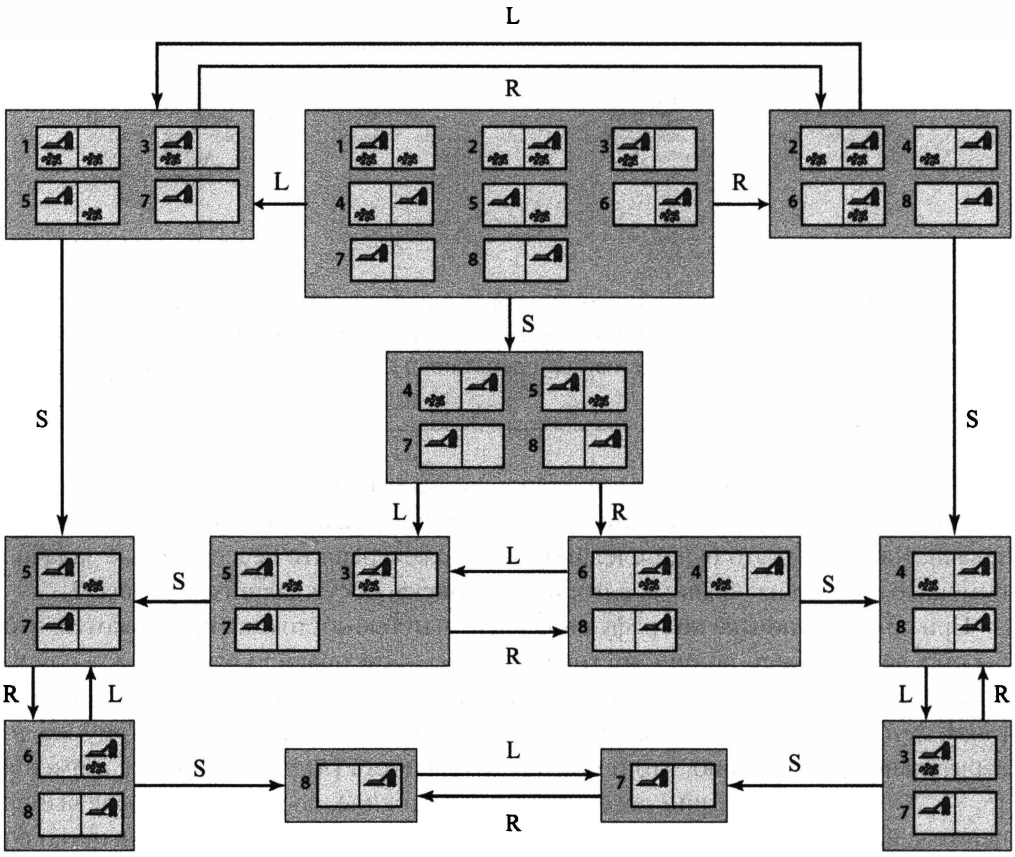


Рис. 4.14. Достижимая часть пространства доверительных состояний для детерминированного бессенсорного мира пылесоса. Каждый темно-серый прямоугольник представляет одно из доверительных состояний, а входящие в него физические состояния представлены светлыми прямоугольниками. В любой заданный момент агент находится в известном ему доверительном состоянии, но не знает, в каком из образующих его физических. Начальное доверительное состояние (полная неизвестность) — центральный прямоугольник в верхнем ряду

Однако даже с подобным улучшением решение бессенсорных задач в том виде, в каком мы его описали, редко осуществимо на практике. Одна из проблем состоит в обширности пространства доверительных состояний — в предыдущей главе было показано, что часто пространство поиска размером N оказывается слишком большим, а в данном случае мы имеем дело с поиском в пространстве размером 2^N.

Более того, каждый элемент в этом пространстве поиска представляет собой множество компонентов размерностью до N. Поэтому при больших значениях N может оказаться просто невозможно представить даже одно доверительное состояние без выхода за пределы доступной памяти.

Одно из решений состоит в том, чтобы представить доверительное состояние как можно более компактным описанием. Говоря обычным языком, можно было бы, например, выбрать следующие формулировки: в начальном состоянии о своем состоянии агент не знает "Ничего"; после выполнения действия *Left* он уже может утверждать, что находится "Не в правом столбце", и т.д. Как сделать это согласно формальной схеме представления, рассказывается в главе 7.

Другой подход состоит в том, чтобы отказаться от стандартных алгоритмов поиска, рассматривающих доверительные состояния как черные ящики, подобно любому другому состоянию задачи. Вместо этого можно заглянуть *внутрь* доверительных состояний и разработать алгоритмы **►инкрементного поиска в доверительных состояниях**, которые строят решение по одному физическому состоянию за раз. Так, например, в бессенсорном мире пылесоса начальное доверительное состояние — {1, 2, 3, 4, 5, 6, 7, 8} и требуется найти последовательность действий, которая будет работать во всех восьми его физических состояниях. Можно достичь этой цели, сначала отыскав решение для первого состояния, а затем проверить, будет ли оно работать для второго. Если нет, то нужно будет вернуться и найти другое решение для состояния 1 и т.д.

Подобно тому, как в случае И–ИЛИ-поиска требуется найти решение для каждой ветви, исходящей из И-узла, этот алгоритм должен отыскать решение для каждого состояния в доверительном состоянии. Разница лишь в том, что при И–ИЛИ-поиске допускается находить разные решения для каждой из ветвей, а при инкрементном поиске в доверительных состояниях необходимо найти *одно* решение, которое будет работать для *всех* состояний.

Основное преимущество инкрементного подхода заключается в том, что чаще всего обнаружить отсутствие решения можно достаточно быстро: если доверительное состояние неразрешимо, обычно имеет место ситуация, когда небольшое подмножество из нескольких его первых исследованных состояний также оказывается неразрешимым. В некоторых случаях это приводит к ускорению, пропорциональному размеру доверительных состояний, которые сами по себе могут быть такими же обширными, как и пространство физических состояний.

4.4.2. Поиск в частично наблюдаемых средах

Многие задачи невозможно решить без какого-либо восприятия. Например, бессенсорный вариант головоломки ***Восемь*** невозможен. С другой стороны, восприятие даже небольшого объема информации может оказать большое влияние на результат: головоломку ***Восемь*** можно решить, если видеть только плитку в верхнем левом углу. Решение заключается в перемещении каждой плитки по очереди

в видимый квадрат с последующим отслеживанием ее местоположения начиная с этого момента.

Для задач в частично наблюдаемой среде цель спецификации состоит в определении функции PERCEPT(s), возвращающей восприятие, получаемое агентом в данном состоянии. Если восприятие не является полностью детерминированным, можно использовать функцию PERCEPTS, возвращающую множество возможных восприятий. Для задач в полностью наблюдаемых средах PERCEPT(s) = s для любого состояния s, а для бессенсорных задач PERCEPT(s) = *null*.

Рассмотрим мир пылесоса с локальным восприятием, в котором агент имеет датчик положения, возвращающий восприятие L в левом квадрате и R — в правом, а также датчик мусора, возвращающий восприятие *Dirty* при наличии мусора и *Clean*, если в квадрате чисто. Таким образом, функция PERCEPT в состоянии 1 возвращает значение [L, *Dirty*]. В случае частично наблюдаемых сред обычно бывает так, что в нескольких состояниях восприятие будет одним и тем же, например в состоянии 3 восприятие также будет [L, *Dirty*]. Следовательно, при наличии такого начального восприятия начальное доверительное состояние агента будет {1, 3}. В частично наблюдаемых средах функцию определения преемника в пространстве доверительных состояний можно представить как выполняющуюся в три этапа, как показано на рис. 4.15.

- На этапе **прогнозирования** вычисляется доверительное состояние, достигаемое в результате выполнения действия, RESULT(b, a), — точно так, как это делается при решении бессенсорных задач. Чтобы подчеркнуть, что результат этих вычислений является лишь прогнозом, *оценкой*, будем использовать обозначение $\hat{b}$ = RESULT(b, a), где "шапочка" над b означает "оценка", а также воспользуемся записью PREDICT(b, a) как синонимом для RESULT(b, a).
- На этапе **определения возможного восприятия** вычисляется множество вариантов восприятия, которые могут быть наблюдаемы в прогнозируемом доверительном состоянии (для представления восприятия будем использовать обозначение o):

$$\text{POSSIBLE-PERCEPTS}(\hat{b}) = \{o : o = \text{PERCEPT}(s) \text{ и } s \in \hat{b}\}.$$

- На этапе **обновления** для каждого возможного восприятия вычисляется доверительное состояние, определяемое этим восприятием. Обновленное доверительное состояние b_o представляет собой множество состояний в $\hat{b}$, в которых может быть получено это восприятие:

$$b_o = \text{UPDATE}(\hat{b}, o) = \{s : o = \text{PERCEPT}(s) \text{ и } s \in \hat{b}\}.$$

При планировании агенту приходится иметь дело с *возможными* восприятиями, поскольку он не знает, каким будет *реальное* восприятие, пока не выполнит план. Обратите внимание, что недетерминированность физической среды приводит к увеличению размера доверительного состояния на этапе

прогнозирования, но каждое обновленное доверительное состояние b_o всегда будет не больше прогнозируемого доверительного состояния $\hat{b}$, поскольку наблюдение может лишь уменьшить неопределенность. Более того, в детерминированной среде доверительные состояния для разных восприятий не пересекаются, образуя независимые *сегменты* в исходном прогнозируемом доверительном состоянии.

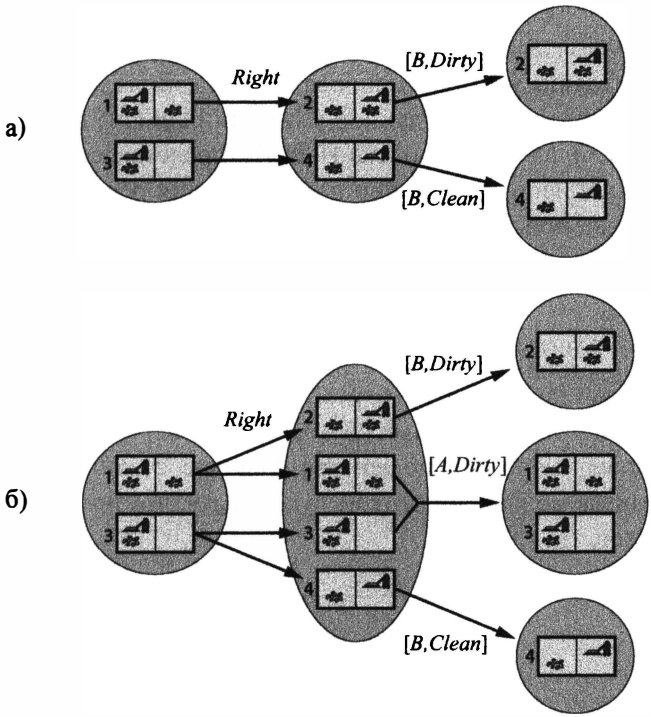


Рис. 4.15. Два примера переходов в мире пылесоса с локальным восприятием. **а)** В детерминированном варианте такого мира в начальном доверительном состоянии выполняется действие *Right*, что приводит к новому прогнозируемому доверительному состоянию из двух возможных физических состояний; для этих состояний возможными вариантами восприятия являются [*R, Dirty*] и [*R, Clean*], что определяет два новых доверительных состояния, каждое из которых включает по одному физическому. **б)** В ненадежном мире пылесоса при выполнении в начальном доверительном состоянии действия *Right* новое прогнозируемое доверительное состояние будет включать четыре физических состояния; для этих доверительных состояний возможными вариантами восприятия являются [*L, Dirty*], [*R, Dirty*] и [*R, Clean*], что приводит к трем новым доверительным состояниям, как показано на рисунке

Собрав все три этапа в одно целое, можно, исходя из заданного действия, определить возможное доверительное состояние и возможные результирующие восприятия:

$$\text{RESULTS}(b, a) = \{b_o : b_o = \text{UPDATE}(\text{PREDICT}(b, a), o) \text{ и } o \in \text{POSSIBLE-PERCEPTS}(\text{PREDICT}(b, a))\}. \quad (4.5)$$

4.4.3. Решение задач в частично наблюдаемой среде

В предыдущем разделе было показано, как вывести функцию RESULTS для доверительного состояния недетерминированной задачи из той физической задачи, на которой она основана, при заданной функции PERCEPT. Если воспользоваться этой формулировкой, алгоритм поиска И–ИЛИ, представленный на рис. 4.11, может быть применен непосредственно для получения решения. На рис. 4.16 показана часть дерева поиска для частично наблюдаемого мира пылесоса в предположении, что начальное восприятие — [*A*, *Dirty*]. Решение представляет собой условный план:

[*Suck*, *Right*, **if** *Bstate* = {6} **then** *Suck* **else** [ ]].

Обратите внимание: поскольку алгоритму поиска И–ИЛИ была передана задача, сформулированная в доверительных состояниях, он вернул условный план, также проверяющий доверительное, а не действительное состояние. Так и должно быть: в частично наблюдаемой среде агент не будет знать, что представляет собой фактическое состояние.

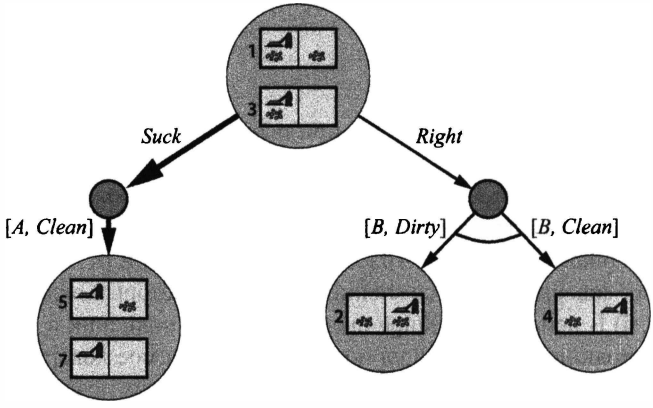


Рис. 4.16. Первый уровень дерева И–ИЛИ-поиска для задачи в частично наблюдаемом мире пылесоса; первое действие в решении — *Suck*

Как и в случае применения стандартных алгоритмов поиска для решения бессенсорных задач, алгоритм И–ИЛИ-поиска воспринимает доверительные состояния как черные ящики, — как и любые другие состояния. Можно улучшить эту ситуацию, проверяя, являются ли ранее сформированные доверительные состояния подмножествами или надмножествами для текущего состояния, так же как и в случае бессенсорных задач. Можно также вывести алгоритмы инкрементного поиска, аналогичные описанным выше для бессенсорных задач и обеспечивающие существенное ускорение расчетов в сравнении с вариантом работы с черными ящиками.

4.4.4. Агент для частично наблюдаемых сред

Агент для частично наблюдаемых сред формулирует задачу, применяет алгоритм поиска (такой, как И–ИЛИ-поиск) для ее решения и выполняет найденное решение. Между этим агентом и агентом для полностью наблюдаемых детерминированных сред имеются два основных различия. Во-первых, решение будет представлять собой условный план, а не последовательность конкретных действий, и чтобы выполнить входящие в него *if–then–else*-выражения, агенту потребуется проверять текущее состояние и выполнять соответствующую ветвь условия. Во-вторых, агенту необходимо поддерживать свое доверительное состояние, пока он выполняет действия и получает восприятия. Этот процесс напоминает процедуру прогнозирования-наблюдения-обновления в уравнении (4.5), но на самом деле он проще, так как восприятие будет определяться окружающей средой, а не рассчитываться агентом. При начальном доверительном состоянии b, действии a и восприятии o новое доверительное состояние определяется так:

$$b' = \text{UPDATE}(\text{PREDICT}(b, a), o). \tag{4.6}$$

Рассмотрим вариант мира пылесоса *в детском саду*, в котором агенты воспринимают состояние только своего текущего квадрата и мусор в каждом квадрате может появиться когда угодно, если только агент не чистит его непосредственно в этот момент.[7] На рис. 4.17 показано доверительное состояние, которое поддерживается в этой среде.

В частично наблюдаемых средах, к которым относится абсолютное большинство сред в реальном мире, поддержание своего доверительного состояния является основной функцией любой интеллектуальной системы. Этой функции можно давать различные названия: ►**мониторинг**, ►**фильтрация** или ►**оценка состояния**, но суть остается неизменной. Уравнение (4.6) называют формулой рекурсивной оценки состояния, поскольку новое доверительное состояние в нем вычисляется на основе предыдущего вместо анализа всей последовательности восприятий. Чтобы агент не отставал от происходящего, вычисления должны выполняться как

[7] Как обычно, приносим извинения тем, кто не знаком с влиянием маленьких детей на окружающую среду.

минимум с той же скоростью, с которой поступают данные восприятия. По мере усложнения среды у агента может оказаться достаточно времени только на то, чтобы лишь приблизительно просчитать доверительное состояние, возможно, сфокусировав все внимание исключительно на результатах восприятия тех аспектов среды, которые интересуют его в данный момент. Бо́льшая часть работы по этой проблеме была проведена для стохастических, непрерывных сред с использованием тех средств теории вероятности, которые обсуждаются в главе 14.

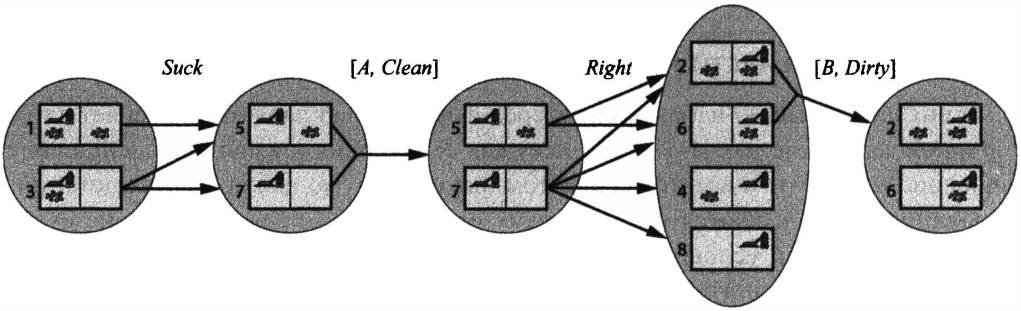


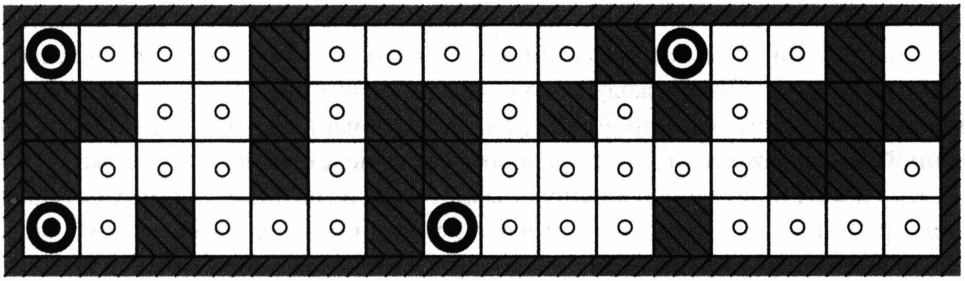
Рис. 4.17. Два цикла прогнозирования-обновления с локальным восприятием для поддержания доверительного состояния в мире пылесоса в детском саду

В этом разделе рассматривается пример функционирования агента в дискретной среде с детерминированными датчиками и недетерминированными действиями. Речь пойдет о роботе с конкретной задачей оценки состояния, называемой ►**локализация**, т.е. определение того, где он находится, при наличии карты мира и известной ему последовательности восприятий и действий. Итак, робот находится в среде, представляющей собой лабиринт, показанный на рис. 4.18. Он оснащен четырьмя сонарными датчиками, сообщающими, есть ли перед ними препятствие — внешняя стена или темный квадрат (см. рисунок), — в каждом из четырех географических направлений. Восприятие представляется в форме вектора из четырех битов, по одному для каждого из направлений (север, восток, юг и запад), расположенных в указанном порядке. Следовательно, восприятие 1011 означает, что имеются препятствия на севере, юге и западе, а на востоке их нет.

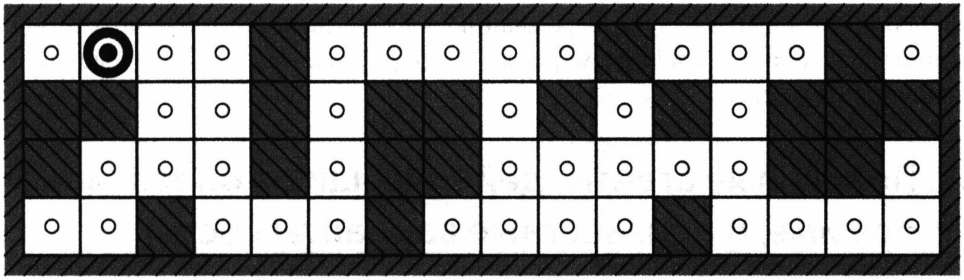
Предполагается, что датчики предоставляют корректные данные и что робот имеет правильную карту окружающей среды. Но, к сожалению, навигационная система робота сломана, поэтому, выполняя действие *Right* (вправо), он случайным образом перемещается на один из соседних квадратов. Задача робота — определить его текущее местоположение.

Предположим, что робот был только что включен и абсолютно не знает, где он находится, поэтому его начальное доверительное состояние представляет собой множество всех местоположений на карте. Далее робот получает восприятие 1011

и выполняет обновление, воспользовавшись уравнением b_o = UPDATE(1011), получив в результате 4 местоположения, показанные на рис. 4.18, *а*. Советуем вам внимательно осмотреть лабиринт и убедиться, что в нем есть всего четыре местоположения, позволяющие получить восприятие 1011.



а) Возможные местоположения робота после получения восприятия $E_1 = 1011$



б) Возможные местоположения робота после получения восприятий $E_1 = 1011$, $E_2 = 1010$

Рис. 4.18. Большими кружками указаны возможные местоположения робота: **а)** после получения первого восприятия, E_1 = 1011, и **б)** после его перемещения на один квадрат и получения второго восприятия, E_2 = 1010. Если датчики надежны и функция определения преемника точна, существует только одно местоположение, в котором может оказаться робот в соответствии с этой последовательностью двух восприятий

Затем робот выполняет действие *Right*, однако результат его выполнения является недетерминированным, поэтому новое доверительное состояние, b_a = PREDICT(b_o, *Right*), включает все те местоположения, которые находятся на расстоянии одного шага в любую сторону от местоположения в b_o. Получив второе восприятие, 1010, робот просчитывает функцию UPDATE(b_a, 1010) и обнаруживает, что доверительное состояние свернулось до единственного местоположения, показанного на рис. 4.18, *б*. В данном лабиринте это единственное местоположение, которое может быть обнаружено в результате вычисления функции

UPDATE(PREDICT(UPDATE(*b*, 1011), *Right*), 1010).

При недетерминированности действий функция PREDICT на этапе прогнозирования расширяет доверительное состояние, однако функция UPDATE на этапе обновления вновь сокращает его — до тех пор, пока восприятия предоставляют некоторую полезную для идентификации местоположения информацию. В некоторых случаях восприятия могут предоставлять недостаточно данных для эффективной локализации. Если в лабиринте имеется один или более коридоров, вытянутых в направлении "восток-запад", то робот может получать длинную последовательность восприятий 1010, не позволяющую уточнить, где именно он находится в таком коридоре (а если их несколько, то и в каком из них). Но для сред с разумными изменениями в географии процессы локализации часто быстро сходятся к единственной точке, даже если действия робота будут недетерминированными.

А что произойдет, если неисправными будут и датчики? Если рассуждать, используя только булеву логику, то в этой ситуации необходимо рассматривать каждый бит восприятия либо как правильный, либо как неправильный, что в конечном счете то же самое, что вообще не получать какой-либо информации восприятия. Однако в главе 12 будет показано, что вероятностные рассуждения позволяют извлечь полезную информацию и из данных, поступающих от неисправного датчика, по крайней мере до тех пор, пока он ошибается менее чем в половине случаев.

4.5. Поисковые агенты, действующие в оперативном режиме, и неизвестные варианты среды

До сих пор речь шла только о таких агентах, в которых использовались алгоритмы ▶**поиска в автономном режиме**. Эти агенты вычисляют полное решение, прежде чем предпринять первое действие. В противоположность этому агент, выполняющий ▶**поиск в оперативном режиме**, функционирует по методу чередования вычислений и действий: вначале предпринимает действие, а затем анализирует состояние среды и вычисляет следующее действие. Поиск в оперативном режиме целесообразно применять в динамических или полудинамических средах, где есть возможность получения штрафа за то, что агент ведет себя пассивно и вычисляет свои действия слишком долго. Поиск в оперативном режиме также будет полезен в недетерминированных средах, поскольку это позволит агенту сосредоточить свои вычислительные усилия на непредвиденных обстоятельствах, которые действительно имеют место, а не на тех, которые могут произойти, но, вероятно, не произойдут.

Конечно, здесь необходим компромисс: чем больше агент планирует наперед, тем реже он будет попадать в затруднительное положение. В неизвестном окружении, когда агент не знает, какие состояния среды существуют и к чему могут привести его действия, он должен воспринимать свои действия как эксперименты, направленные на изучение окружающей среды.

Каноническим примером применения поиска в оперативном режиме может служить ►**задача отображения**: робот находится в неизвестном здании и должен его исследовать, чтобы составить карту, которую затем сможет использовать для перемещения из точки A в точку B. Методы выхода из лабиринтов (необходимые знания для отважных героев древности) также могут быть приведены в качестве примеров такого поиска. Однако исследование пространства — это не единственная форма познания окружающего мира. Приведем в пример поведение новорожденного ребенка: в его распоряжении есть много возможных действий, но он еще не знает, к чему они приведут, и испытывает лишь немногие возможные состояния из числа тех, которые может выполнить.

4.5.1. Задачи поиска в оперативном режиме

Любая задача поиска в оперативном режиме решается посредством чередования вычислений, восприятия и действия. Начнем обсуждение с предположения, что среда детерминирована и полностью наблюдаема (в главе 17 эти предположения будут ослаблены), и условимся, что агенту известно только следующее.

- Функция ACTIONS(s), определяющая список действий, допустимых в состоянии s.
- Функция стоимости этапа $c(s, a, s')$, определяющая стоимость выполнения действия a в состоянии s с целью перехода в состояние s'. Она не может использоваться, пока агент не узнает, что результатом действия a является состояние s'.
- Функция IS-GOAL(s), проверяющая достижение целевого состояния.

Обратите внимание, в частности, на то, что агент *не может* определить значение функции RESULT(s, a) за исключением тех случаев, когда он действительно находится в состоянии s и выполняет действие a. Например, в задаче с лабиринтом, представленной на рис. 4.19, агент не знает, что выполнение действия *Up* в состоянии (1,1) приведет его в состояние (1,2); как не знает и того, что после его выполнения он сможет выполнить действие *Down* и вернется в состояние (1,1). Такая степень неведения в некоторых приложениях может быть уменьшена, — например, робот-исследователь может знать, как работают его действия по передвижению, и оставаться в неведении лишь в отношении местонахождения препятствий.

И наконец, агент может иметь доступ к допустимой эвристической функции $h(s)$, предоставляющей оценку расстояния до целевого состояния от текущего. Например, на рис. 4.19 агент может знать местоположение цели и быть в состоянии использовать эвристику манхэттенского расстояния до нее (см. раздел 3.6).

Как правило, задача агента состоит в том, чтобы достичь целевого состояния при минимальных затратах. (Другая возможная цель — просто исследовать всю среду.) Стоимость — это общая стоимость пути, который агент пройдет в поисках цели. Обычно эту стоимость сравнивают со стоимостью того пути, который агент

выбрал бы, *заранее зная пространство поиска*, т.е. оптимального пути в известной среде. В терминологии алгоритмов оперативного поиска это соотношение называется ▶**коэффициентом конкурентоспособности**, — желательно, чтобы этот коэффициент был как можно меньше.

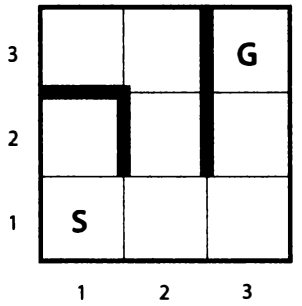


Рис. 4.19. Простая задача с лабиринтом. Агент начинает движение в квадрате S и должен достичь квадрата G, но ничего не знает об окружающей среде

Исследования, проводимые в оперативном режиме, часто подвержены попаданию в ▶**тупик**, т.е. в такое состояние, из которого целевого состояния достичь невозможно. Если агент не знает, к чему приводит каждое действие, он может выполнить действие "прыгнуть в бездонную яму" и, таким образом, уже никогда не достигнуть цели. В целом ➨*ни один алгоритм не позволяет избежать тупиков во всех пространствах состояний.* Рассмотрим два тупиковых пространства состояний, показанных на рис. 4.20, *а*. Алгоритм поиска в оперативном режиме, который последовательно посетил состояния *S* и *A*, не сможет определить, находится он в верхнем варианте лабиринта или в нижнем, — они выглядят совершенно одинаково, исходя из того, что видел агент. Следовательно, у него не было возможности узнать, как выбрать правильное действие в каждой ситуации, и он должен принять одинаковое решение в обоих вариантах среды, а значит, в одном из вариантов он потерпит неудачу. Это один из примеров ▶**аргумента противника** (adversary argument): можно представить, как противник формирует пространство состояний в ходе того, как это пространство исследуется агентом, и может размещать цели и тупики везде, где пожелает, как на рис. 4.20, *б*.

Тупики представляют собой реальную трудность для исследования роботов: лестницы, рампы, обрывы, улицы с односторонним движением — все это примеры состояний, в которых возможны ▶**необратимые действия**, после выполнения которых уже не будет возможности вернуться в предыдущее состояние. Алгоритм исследования, который будет представлен ниже, гарантированно работает лишь в пространствах состояний, являющихся ▶**безопасно исследуемыми**, т.е. некоторое целевое состояние в них является достижимым из каждого достижимого

состояния. Пространства состояний с обратимыми действиями, такие как лабиринты и головоломка ***Восемь***, очевидно, являются безопасно исследуемыми (если для них вообще существует какое-либо решение). Более подробно тема безопасного исследования рассматривается в разделе 22.3.2.

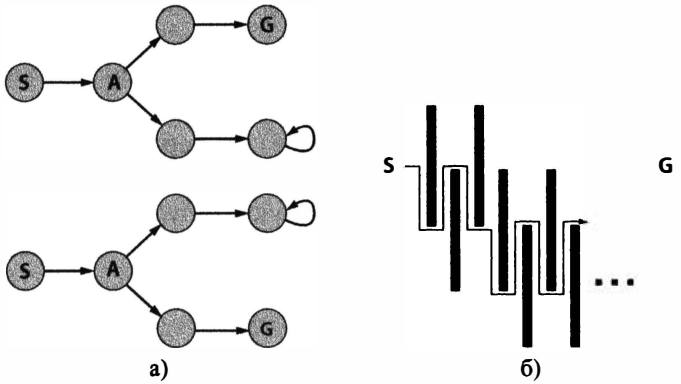


Рис. 4.20. а) Два пространства состояний, способные завести агента, выполняющего поиск в оперативном режиме, в тупик. Любой такой агент потерпит неудачу по меньшей мере в одном из двух вариантов этих пространств. **б)** Двухмерная среда, которая может вынудить агента, выполняющего поиск в оперативном режиме, следовать к цели, по сколь угодно неэффективному маршруту. Какой бы выбор агент ни сделал, противник блокирует его маршрут еще одной длинной и тонкой стеной, чтобы путь, по которому следует агент, стал намного длиннее по сравнению с наилучшим возможным путем

Но даже в безопасно исследуемых вариантах среды нельзя гарантировать достижение какого-либо ограниченного значения коэффициента конкурентоспособности, если в них имеются маршруты с неограниченной стоимостью. Это утверждение легко доказать применительно к вариантам среды с необратимыми действиями, но фактически оно остается истинным и для случая с обратимыми действиями (см. рис. 4.20, *б*). По этой причине обычно принято характеризовать производительность алгоритмов поиска в оперативном режиме с учетом размеров всего пространства состояний, а не только глубины самой поверхностной цели.

4.5.2. Агенты, выполняющие поиск в оперативном режиме

После каждого действия агент, выполняющий поиск в оперативном режиме в наблюдаемой среде, получает восприятие, сообщающее ему, в каком состоянии он находится. Исходя из этой информации он может дополнить свою карту окружающей среды, а затем обновленная карта используется им для планирования, куда идти дальше. Такое чередование этапов планирования и действия означает, что

алгоритмы поиска в оперативном режиме довольно сильно отличаются от алгоритмов поиска в автономном режиме, которые рассматривались выше. В последних исследуется их *модель* пространства состояний, тогда как алгоритмы поиска в оперативном режиме исследуют реальный мир. Например, алгоритм поиска A* может развернуть узел в одной части пространства состояний, а следующим развернуть узел в иной, отдаленной его части, поскольку процедура развертывания узла включает моделируемые, а не реальные действия.

С другой стороны, любой алгоритм поиска в оперативном режиме позволяет агенту развертывать только тот узел, который он физически занимает. Для предотвращения необходимости путешествовать через все дерево, чтобы развернуть следующий узел, представляется более удобным развертывание узлов в *локальном* порядке. Именно таким свойством обладает поиск в глубину, поскольку (за исключением операции возврата) следующий развертываемый узел является дочерним узлом ранее развернутого узла.

Алгоритм агента, выполняющего поиск в глубину в оперативном режиме, приведен на рис. 4.21. Агент хранит составленную им карту в таблице *result*[*s*, *a*], в которой регистрируются состояния, явившиеся следствием выполнения действия *a* в состоянии *s*. (Для недетерминированных действий агент может записывать в эту таблицу множество состояний как элемент таблицы *result*[*s*, *a*].) Всякий раз, когда текущее состояние имеет еще непроверенные действия, этот агент предпринимает попытку выполнения одного из них. Сложности возникают после того, как агент опробовал *все* действия в некотором состоянии. При поиске в глубину в автономном режиме это состояние просто удаляется из очереди, тогда как при поиске в оперативном режиме агент должен выполнить возврат физически. При поиске в глубину это равносильно возврату в последнее по времени состояние, из которого агент перешел в текущее состояние. Чтобы реализовать это требование, в алгоритме предусматривается ведение еще одной таблицы, в которой для каждого состояния перечисляются состояния-предшественники, к которым агент еще не выполнял возврат. Если агент исчерпал список состояний, к которым может выполнить возврат, это означает, что проведенный агентом поиск является полным.

Читателю рекомендуется выполнить трассировку выполнения процедуры ONLINE-DFS-AGENT применительно к лабиринту, показанному на рис. 4.19. Можно довольно легко убедиться в том, что в наихудшем случае агент в конечном итоге пройдет по каждой связи в данном пространстве состояний точно два раза. Для задач исследования пространства такая организация работы является оптимальной; а вот для задач поиска целевого состояния коэффициент конкурентоспособности может оказаться сколь угодно плохим, если агент отправится в длительную экскурсию, когда целевое состояние находится непосредственно рядом с начальным. Такая задача успешно решается с использованием варианта метода итеративного углубления для работы в оперативном режиме, — если среда задачи представляет собой однородное дерево, коэффициент конкурентоспособности подобного агента будет небольшой константой.

```
function ONLINE-DFS-AGENT(problem, s') returns действие action
        s, a, предыдущие состояние и действие, первоначально
              неопределенные
  persistent: result, таблица, отображающая (s, a) в s', первоначально пустая
        untried, таблица, отображающая s в список неопробованных действий
        unbacktracked, таблица, отображающая s в список состояний,
                к которым еще не было возвратов

  if problem.IS-GOAL(s') then return stop
  if s' есть новое состояние (отсутствующее в таблице untried) then
     untried[s'] ← problem.ACTIONS(s')
  if s не является неопределенным then
     result[s, a] ← s'
     добавить s в начало таблицы unbacktracked[s']
  if элемент untried[s'] пуст then
     if элемент unbacktracked[s'] пуст then return stop
     else a ← действие b такое, что result[s', b] = POP(unbacktracked[s'])
  else a ← POP(untried[s'])
  s ← s'
  return a
```

Рис. 4.21. Агент, выполняющий поиск в оперативном режиме и использующий для проведения исследования алгоритм поиска в глубину. Этот агент может безопасно исследовать только те пространства состояний, в которых каждое действие может быть "отменено" каким-либо другим действием

Из-за применения метода с возвратами алгоритм ONLINE-DFS-AGENT будет работать только в пространствах состояний, где все действия обратимы. Существуют также немного более сложные алгоритмы, применимые в пространствах состояний общего вида, но ни один из них не имеет ограниченного коэффициента конкурентоспособности.

4.5.3. Локальный поиск в оперативном режиме

Как и поиск в глубину, **поиск восхождением к вершине** обладает свойством локализации в отношении его операции развертывания узлов. В действительности уже *сам* поиск восхождением к вершине можно считать алгоритмом поиска в оперативном режиме, поскольку он предусматривает хранение в памяти только одного текущего состояния! К сожалению, в своей простейшей форме этот алгоритм не очень полезен, так как оставляет агента в таком положении, что он не может покинуть локальный максимум и отправиться куда-то еще. Более того, в этом алгоритме не может использоваться перезапуск случайным образом, поскольку агент не способен перенести самого себя в новое состояние.

Вместо перезапуска случайным образом для исследования среды может быть предусмотрено использование ▶**случайного блуждания** (*random walk*). В методе случайного блуждания просто случайным образом выбирается одно из действий, доступных из текущего состояния, причем предпочтение отдается действиям, которые еще не были опробованы. Легко доказать, что метод случайного блуждания позволяет агенту в *конечном итоге* найти цель или выполнить свою задачу исследования при условии, что пространство является конечным и безопасно исследуемым.[8] С другой стороны, процесс поиска может оказаться очень медленным. На рис. 4.22 показана среда, в которой поиск цели с помощью метода случайного блуждания потребует экспоненциально возрастающего количества этапов, поскольку в каждом состоянии верхнего ряда (за исключением состояния *S*) вероятность выполнения шага назад вдвое превышает вероятность выполнения шага вперед. Безусловно, данный пример является надуманным, но существует много реальных пространств состояний, топология которых способствует возникновению "ловушек" такого рода при случайном блуждании.

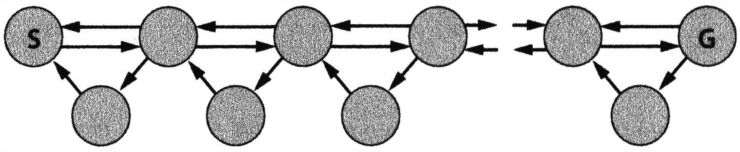


Рис. 4.22. Среда, в которой применение метода случайного блуждания для поиска цели потребует выполнения экспоненциально возрастающего количества этапов

Более эффективным подходом оказалось расширение метода поиска восхождением к вершине способностью *запоминать*, а не выбирать следующее действие случайным образом. Основная идея состоит в том, что необходимо хранить в памяти "текущую наилучшую оценку" $H(s)$ стоимости достижения цели из каждого состояния, которое уже было посещено. В начале работы $H(s)$ представляет собой просто эвристическую оценку $h(s)$, которая обновляется по мере того, как агент накапливает опыт, исследуя пространство состояний.

На рис. 4.23 приведен простой пример одномерного пространства состояний. На рис. 4.23, *а* показано, что агент, по-видимому, зашел в тупик, попав в плоский локальный минимум, соответствующий затененному состоянию. Вместо того чтобы оставаться там, где находится, агент должен последовать по тому пути, который может рассматриваться как наилучший путь к цели согласно текущим оценкам стоимостей для соседних состояний. Оценка стоимости достижения цели через соседнее состояние s' равна оценке стоимости достижения s' плюс оценка

[8] Случайные блуждания конечны на бесконечных одно- и двухмерных сетках. На трехмерной сетке вероятность того, что случайно блуждающий агент когда-либо вернется в начальную точку, составляет всего лишь приблизительно 0,3405 (Хьюз [1092], 1995).

стоимости достижения цели из s′, т.е. равна $c(s, a, s') + H(s')$. В данном примере агенту доступны два действия с оценками стоимости, 1+9 и 1+2, поэтому, по всей видимости, лучше всего двигаться вправо.

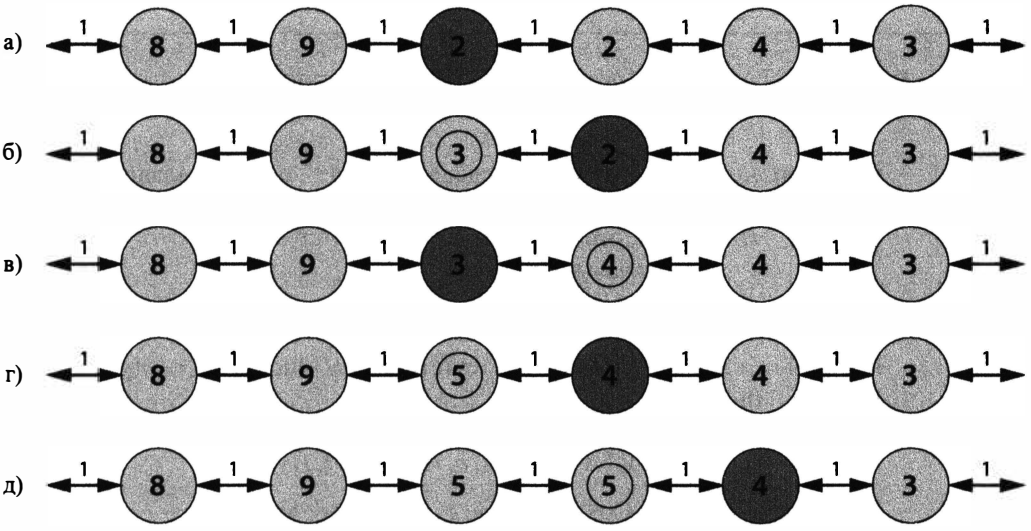


Рис. 4.23. Пять итераций алгоритма LRTA* в одномерном пространстве состояний. Каждое состояние обозначено значением $H(s)$, текущей оценкой стоимости достижения цели, а каждая дуга обозначена соответствующей ей стоимостью этапа. Затененное состояние показывает местонахождение агента, а значения, обновленные после каждой итерации, обозначаются двойным кружком

На следующем этапе (рис. 4.23, *б*) становится очевидно, что оценка стоимости для этого затененного состояния, равная 2, была слишком оптимистичной. Поскольку стоимость наилучшего хода равна 1 и он ведет в состояние, которое находится по меньшей мере на расстоянии 2 шагов от цели, то затененное состояние должно находиться по меньшей мере на расстоянии 3 шагов от цели, поэтому его оценка H должна быть обновлена соответствующим образом, что и показано на рисунке. Продолжая этот процесс, агент перейдет вперед и назад еще дважды, каждый раз обновляя оценку H и "сглаживая" локальный минимум до тех пор, пока не вырвется из него вправо (рис. 4.23, *д*).

Агент, в котором реализован этот алгоритм, получивший название поиска A* в реальном времени с обучением (▶ **LRTA*** — Learning Real-Time A*), приведен на рис. 4.24. Как и в алгоритме ONLINE-DFS-AGENT, в данном алгоритме составляется карта среды с использованием таблицы *result*. Этот алгоритм обновляет оценку стоимости для состояния, которое он только что оставил, а затем выбирает ход, "который представляется наилучшим" в соответствии с его текущими оценками

стоимости. Следует отметить одну важную деталь: действия, которые еще не были опробованы в состоянии s, всегда рассматриваются как ведущие непосредственно к цели с наименьшей возможной стоимостью, а именно — $h(s)$. Такой ▶**оптимизм в отношении неопределенности** побуждает агента исследовать новые пути, которые могут действительно оказаться перспективными.

---

function LRTA*-AGENT(*problem*, s', h) **returns** действие a
  s, a, предыдущие состояние и действие, первоначально неопределенные
  persistent: *result*, таблица, отображающая (s, a) в s', первоначально пустая
  H, таблица, отображающая s в оценку стоимости, первоначально пустая

  if IS-GOAL(s') **then return** *stop*
  if s' является новым состоянием (отсутствует в H) **then** $H[s'] \leftarrow h(s')$
  if s не null **then**
    $result[s, a] \leftarrow s'$
    $H[s] \leftarrow \min_{b \in \text{ACTIONS}(s)} \text{LRTA*-COST}(s, b, result[s, b], H)$
  $a \leftarrow \underset{b \in \text{ACTIONS}(s)}{\text{argmin}} \text{LRTA*-COST}(problem, s', b, result[s', b], H)$
  $s \leftarrow s'$
  return a

function LRTA*-COST(*problem*, s, a, s', H) **returns** оценка стоимости
  if s' является неопределенным **then return** $h(s)$
  else return *problem*.ACTION-COST(s, a, s') + $H[s']$

Рис. 4.24. Функция LRTA*-AGENT выбирает действие в соответствии со значениями соседних состояний, которые обновляются по мере того, как агент передвигается в пространстве состояний

Агент LRTA* гарантированно найдет цель в любой конечной, безопасно исследуемой среде. Однако, в отличие от алгоритма поиска A*, в бесконечных пространствах состояний применяемый в этом агенте алгоритм становится неполным, поскольку в некоторых случаях он может вовлечь агента в бесконечные блуждания. В наихудшем случае данный алгоритм позволяет исследовать среду с n состояниями за $O(n^2)$ этапов, но чаще всего он действует намного лучше. Агент LRTA* представляет собой лишь один пример из большого семейства агентов, выполняющих поиск в оперативном режиме, которые могут быть определены путем задания правила выбора действия и правила обновления различными способами. Это семейство агентов, первоначально разработанное для стохастических вариантов среды, рассматривается в главе 22.

4.5.4. Обучение при поиске в оперативном режиме

То, что агенты, выполняющие поиск в оперативном режиме, изначально находятся в полном неведении, открывает некоторые возможности для обучения. Во-первых, агенты изучают "карту" своей среды (а точнее, результаты каждого действия в каждом состоянии), просто регистрируя результаты каждого из своих опытов. Во-вторых, агенты, выполняющие локальный поиск, получают все более точные оценки значения каждого состояния, используя локальные правила обновления, которые сходятся к точным значениям для каждого состояния, при условии, что агент исследует пространство состояний правильным способом. Как только точные значения станут известны, оптимальные решения могут быть реализованы просто посредством перемещения к преемнику с наименьшей стоимостью, т.е. оптимальной стратегией в этом случае становится метод поиска восхождением к вершине в чистом виде.

Если читатель выполнил рекомендацию провести трассировку поведения алгоритма ONLINE-DFS-AGENT в среде, приведенной на рис. 4.19, то должен был заметить, что этот агент не слишком умен. Например, после того как агент обнаружил, что действие *Up* ведет из пункта (1,1) в пункт (1,2), он еще не знает, что действие *Down* возвратит его назад, в пункт (1,1), или что следующее действие *Up* приведет его из пункта (2,1) в пункт (2,2), из пункта (2,2) в пункт (2,3) и т.д. Вообще говоря, было бы желательно, чтобы агент освоил в результате обучения, что действие *Up* приводит к увеличению координаты y, если на этом пути нет стены, а действие *Down* приводит к уменьшению этой координаты, и т.д.

Для того чтобы это произошло, требуются две вещи. Во-первых, необходимо формальное и явно манипулируемое представление общих правил такого рода. До сих пор эта информация скрывалась внутри "черного ящика", называемого функцией определения преемника RESULT. Данному вопросу посвящены главы 8–11 этой книги. Во-вторых, нужны алгоритмы, позволяющие формировать подходящие общие правила из конкретных наблюдений, сделанных агентом. Эта тема рассматривается в главе 19.

Если ожидается, что в будущем потребуется решать множество подобных задач, имеет смысл потратить время (и память), чтобы облегчить такие поиски. Существует несколько способов сделать это, и все они подпадают под заголовок ► **инкрементный поиск**. Можно хранить дерево поиска в памяти и многократно использовать те его части, которые остаются неизменными в новой задаче. Можно хранить значения эвристической функции h и обновлять их при поступлении новой информации — либо потому, что мир изменился, либо потому, что были вычислены лучшие оценки. Или можно хранить значения функции наилучшего пути g, используя их для объединения в новое решение, и обеспечить их обновление, когда мир меняется.

Резюме

В этой главе рассматривались алгоритмы поиска для задач в частично наблюдаемых, недетерминированных, неизвестных и непрерывных средах.

- Такие методы *локального поиска*, как **поиск восхождением к вершине**, сохраняют в памяти только небольшое количество состояний. Они применялись к задачам оптимизации, суть которых заключалась в том, чтобы найти достаточно хорошее состояние, не заботясь о пути к нему. Также было разработано несколько стохастических алгоритмов локального поиска, включая поиск **имитацией отжига**, возвращающий оптимальное решение при наличии подходящего графика "охлаждения".
- Многие локальные методы поиска применимы также к задачам в непрерывных средах. Задачи **линейного программирования** и **выпуклой оптимизации** накладывают определенные ограничения на форму пространства состояний и характер целевой функции, но используемые в них алгоритмы характеризуются полиномиальным временем выполнения, что на практике часто обеспечивает им чрезвычайно высокую эффективность. Для некоторых математически правильно сформулированных задач можно найти максимум, просто рассчитав, где градиент обращается в нуль, а в других задачах потребуется воспользоваться эмпирическим градиентом, характеризующим разницу в пригодности между двумя соседними точками.
- **Эволюционный алгоритм** представляет собой стохастическое поиск восхождением к вершине, в котором поддерживается популяция состояний, именуемых особями. Новые особи генерируются из пар особей предыдущего поколения с использованием механизмов **мутаций** и **кроссинговера**.
- В **недетерминированных** средах, агенты могут использовать И–ИЛИ-поиск для генерации **условных** планов, что позволяет достичь цели независимо от того, какие именно результаты действий будут иметь место во время его выполнения.
- Когда среда частично наблюдаема, **доверительное состояние** представляет собой множество возможных состояний, в которых агент, по его мнению, может оказаться.
- **Бессенсорные задачи** можно решать, применяя стандартные алгоритмы поиска непосредственно к пространству доверительных состояний, а метод И–ИЛИ-поиска в пространстве доверительных состояний позволяет решить прочие типы задач в частично наблюдаемой среде. Инкрементные алгоритмы, образующие целевые состояния в пределах доверительного состояния, часто оказываются более эффективными.
- **Задачи исследования** возникают, кода агент не имеет представления о том, каковы состояния и действия в окружающей его среде. В безопасно исследуемых вариантах среды агенты, выполняющие **поиск в оперативном**

режиме, могут составить карту и отыскать цель, если она существует. Эффективным методом выхода из локальных минимумов является обновление эвристических оценок на основе получаемого агентом опыта.

Библиографические и исторические заметки

Методы локального поиска имеют долгую историю в математике и компьютерных науках. Безусловно, метод Ньютона–Рафсона (Ньютон [1671], 1671; Рафсон [1852], 1690) может рассматриваться как очень эффективный метод локального поиска в непрерывных пространствах, в которых доступна информация о градиенте. Книга Брента ([3012], 1973) представляет собой классический справочник по алгоритмам оптимизации, для которых такая информация не требуется. Алгоритм поиска по лучу, представленный в данной книге как алгоритм локального поиска, впервые появился в виде одного из вариантов методов динамического программирования с ограничением по ширине, предназначенного для распознавания речи, в системе HARPY ([1452], 1976). Еще один алгоритм подобного типа был глубоко проанализирован Перлом ([1751], 1984, глава 5).

Интерес к теме локального поиска вновь пробудился в начале 1990-х годов под влиянием на удивление хороших результатов, полученных при решении таких крупных задач удовлетворения ограничений, как задача с n-ферзями (Минтон и др. [1588], 1992) и задача формирования логических рассуждений (Селман и др. [2025], 1992), а также под влиянием успешного включения в алгоритмы методов рандомизации, методов множественного одновременного поиска и других усовершенствований. Это возрождение интереса к алгоритмам, названным Кристосом Пападимитриу алгоритмами "новой эры", вызвало также повышенный интерес теоретиков в области компьютерных наук ([1293], 1992; [28], 1994).

В области исследования операций широкую популярность завоевал один из вариантов поиска восхождением к вершине, получивший название ▶**поиска с запретами** (Гловер и Лагуна [870], 1997). Этот алгоритм предусматривает формирование и поддержку списка из k ранее посещенных узлов, которые не могут быть вновь посещены, — помимо повышения эффективности поиска по графу, этот список способствует выходу алгоритма из некоторых локальных минимумов.

Еще одним полезным усовершенствованием алгоритма поиска восхождением к вершине является алгоритм STAGE (Боян и Мур [278], 1998). Идея состоит в том, чтобы использовать локальные максимумы, найденные при выполнении поиска восхождения к вершине со случайным перезапуском, для получения общего представления о форме ландшафта. Алгоритм подгоняет гладкую квадратичную поверхность к множеству обнаруженных локальных максимумов, а затем аналитическим путем вычисляет глобальный максимум этой поверхности. Этот глобальный максимум становится новой точкой перезапуска. В работе Гомеса и др. [889] (1998) показано, что время выполнения алгоритмов с возвратами часто

характеризуется ▶**распределением с тяжелым хвостом** (heavy-tailed distribution). Это означает, что вероятность очень длительного времени выполнения выше, чем можно было бы предположить в случае нормального распределения значений времени выполнения. Когда время выполнения подчиняется распределению с тяжелым хвостом, алгоритм со случайным перезапуском в среднем будет находить решение быстрее, чем единственный прогон, выполняемый до нахождения решения. Хуз и Стюцле посвятили этой теме отдельную книгу ([1056], 2004).

Алгоритм поиска имитацией отжига был впервые описан Кирпатриком и соавторами ([1230], 1983), заимствовавшими эту идею непосредственно из **алгоритма Метрополиса** (этот алгоритм применялся для эмуляции сложных систем в физике ([1560], 1953) и предположительно был изобретен на одном из званых обедов в Лос-Аламосе). Методы поиска имитацией отжига теперь лежат в основе отдельного научного направления, в котором ежегодно публикуются сотни статей.

Поиск оптимальных решений в непрерывных пространствах является предметом исследований в нескольких научных областях, включая **теорию оптимизации**, **теорию оптимального управления** и **вариационное исчисление**. Базовые методы хорошо описываются в работе Бишопа [221] (1995), а в книге Пресса и др. [1821] (2007) охватывается широкий спектр алгоритмов и предоставляется работающее программное обеспечение.

Разрабатывая алгоритмы поиска и оптимизации, исследователи черпали вдохновение в самом широком спектре областей человеческой деятельности: *металлургия* (имитация отжига); *биология* (генетические алгоритмы); *нейробиология* (нейронные сети); *альпинизм* (восхождение к вершине); *экономика* (рыночные алгоритмы (Диас и др. [617], 2006)); *физика* (рои частиц (Ли и Яо [1406], 2012) и спиновые стекла (Мецард и др. [1562], 1987)); *поведение животных* (обучение с подкреплением, алгоритм стаи волков (Мирджалили и Льюис [1589], 2014)); *орнитология* (поиск кукушки (Янг и Деб [2398], 2014)); *энтомология* (колония муравьев (Дориго и др. [638], 2008), пчелиная семья (Карабога и Бастурк [1186], 2007), алгоритм светлячка (Янг [2397], 2009) и оптимизация роем светлячков (Кришнанд и Гхош, 2009)) и пр.

Первые систематические исследования в области **линейного программирования** (ЛП) провел Леонид Канторович ([1184], 1939). Одним из первых приложений для компьютеров была реализация относящегося к этой области **симплексного алгоритма** (Данциг [523], 1949), который до сих пор используется, несмотря на то что в наихудшем случае он характеризуется экспоненциальной сложностью. Позднее Кармаркар ([1188], 1984) разработал гораздо более эффективное семейство алгоритмов **внутренней точки**, которые, как было показано в [1663] (1994), имеют полиномиальную временную сложность при решении более общего класса задач выпуклой оптимизации. Отличное введение в методы выпуклой оптимизации можно найти книгах Бен-Таля и Немировского ([170], 2001), а также Бойда и Ванденберге ([279], 2004).

Важной работой, предшествующей разработке генетических алгоритмов, была публикация Сьюэлла Райта ([2388], 1931), посвященная концепции **ландшафта приспособленности**. В 1950-х годах некоторые специалисты в области статистики, включая Бокса ([275], 1957) и Фридмана ([786], 1959), использовали эволюционные методы для решения задач оптимизации, но популярность этот подход приобрел только после того, как Рехенберг ([1867], 1965) предложил использовать **эволюционные стратегии** для решения задач оптимизации аэродинамических профилей. В 1960- и 1970-х годах Джон Холланд пропагандировал генетические алгоритмы как полезный инструмент оптимизации и как способ расширения нашего понимания адаптации (Холланд [1050], 1975).

Сторонники движения **искусственной жизни** (Лэнгтон [1351], 1995) развили эту идею на один шаг, рассматривая плоды работы генетических алгоритмов как *организмы*, а не как решения задач. Эффект Болдуина, обсуждавшийся в этой главе, был предложен примерно одновременно Конви Ллойдом Морганом ([1623], 1896) и Джеймсом Болдуином ([118], 1896), а компьютерное моделирование уже в наше время помогло прояснить его последствия ([1026], 1987; [13], 1991; [1624], 2015). Для ознакомления с общими сведениями об эволюции авторы рекомендуют книги [2096] (1999), [1880] (2004) и [386] (2007).

Сравнение генетических алгоритмов с другими подходами (особенно с алгоритмом стохастического поиска восхождением к вершине) чаще всего показывает, что генетические алгоритмы сходятся медленнее ([1716], 1994; [1594], 1996; [1157], 1996; [120], 1997). Такие исследования не находят полного признания в сообществе приверженцев генетических алгоритмов, но недавние попытки представителей этого сообщества трактовать поиск на основе популяций как приближенный аналог байесовского обучения (см. главу 20) могут способствовать преодолению разногласий между представителями этой области и ее критиками ([1768], 1999). Для анализа производительности генетических алгоритмов может также применяться теория **квадратичных динамических систем** ([1839], 1998). Есть несколько впечатляющих практических примеров применения генетических алгоритмов в самых различных областях, таких как проектирование антенн ([1440], 2001), системы компьютерного проектирования ([1872], 2003), моделирование климата ([2124], 2015), медицина ([843], 2015) и разработка глубоких нейронных сетей ([1567], 2019).

Генетическое программирование является одним из разделов области генетических алгоритмов, в которых представителями-особями являются программы, а не строки битов. Программы в этом случае представляются в виде деревьев выражений, которые могут быть сформированы либо на стандартном языке программирования, либо в специально разработанных форматах для представления электронных схем, контроллеров роботов и т.д. Кроссинговер предусматривает соединение поддеревьев друг с другом таким образом, что бы потомство гарантированно представляло собой правильно построенные выражения.

Интерес к генетическому программированию был стимулирован работами Джона Козы ([1299], 1992; [1300], 1994), но исследования в этой области проводились уже давно, начиная с экспериментов с машинным кодом ([784], 1958) и с конечными автоматами ([749], 1966). Как и в отношении генетических алгоритмов, споры по поводу эффективности этой методологии еще не прекратились. В работе Джона Козы [1301] (1999) приведены результаты экспериментов по использованию генетического программирования для автоматизированного проектирования схемотехнических устройств.

Работы в области эволюционных алгоритмов публикуются в журналах *Evolutionary Computation* и *IEEE Transactions on Evolutionary Computation*; статьи на эти темы можно также найти в журналах *Complex Systems*, *Adaptive Behavior* и *Artificial Life*. Основной конференцией в этом направлении является *Genetic and Evolutionary Computation Conference*. Хорошими обзорными учебниками по генетическим алгоритмам являются книги Мелани Митчелл ([1592], 1996), Ленгдона и Поли ([1805], 2008).

Непредсказуемость и частичная наблюдаемость в реальных условиях были признаны уже на ранней стадии робототехнических проектов, в которых использовались методы планирования, включая проекты SHAKEY (Файкс и др. [737], 1972) и FREDDY (Мичи [1564], 1972). Этим проблемам стало уделяться больше внимания после публикации влиятельной статьи Макдермотта ([1541], 1978) *Планирование и действие*.

Первой работой, в которой явно использовались И–ИЛИ-деревья, похоже, была программа символического интегрирования SAINT (Слэгл [2083], 1963), упоминаемая в главе 1. Амарел ([39], 1967) применил эту идею к доказательству теорем высказываний (эта тема обсуждается в главе 7) и ввел алгоритм поиска, аналогичный И–ИЛИ-поиску по графу. Затем этот алгоритм был развит Нильссоном ([1687], 1971), который также описал алгоритм АО*, который, как следует из названия, находит оптимальные решения. Алгоритм АО* был дополнительно улучшен Мартелли и Монтанари ([1502], 1973).

Алгоритм АО* работает в направлении сверху вниз, тогда как в направлении снизу вверх работает обобщение алгоритма А* — А* LD, сокращение от А* Lightest Derivation ([726], 2007). Интерес к И–ИЛИ-поиску вновь возродился в начале 2000-х годов благодаря появлению новых алгоритмов поиска циклических решений (Хименес и Торрес [1138], 2000; Хансен и Зильберштейн [960], 2001) и новых методов, вдохновленных идеями динамического программирования (Бонет и Геффнер [249], 2005).

Идея преобразования частично наблюдаемых задач в задачи доверительных состояний возникла у Астрома ([87], 1965) для гораздо более сложного случая вероятностной неопределенности (см. главу 17). Эрдман и Мейсон ([691], 1988) изучали проблему роботизированной манипуляции без датчиков, используя непрерывную форму поиска в пространстве доверительных состояний. Они показали, что возможно правильно сориентировать деталь на столе из произвольного

начального положения с помощью хорошо продуманной последовательности поворачивающих ее действий. В более утилитарных методах, основанных на применении серии точно ориентированных диагональных барьеров на конвейерной ленте, использованы те же алгоритмические идеи ([2337], 1996).

Подход с использованием доверительных состояний был заново изобретен в контексте бессенсорных и частично наблюдаемых поисковых задач Генесеретом и Нурбакшем ([837], 1993). Дополнительная работа по бессенсорным задачам была проведена в сообществе логического планирования ([882], 1996; [2092], 1998). В этой работе был сделан акцент на кратком представлении доверительных состояний, как объясняется в главе 11. Бонет и Геффнер ([248], 2000) ввели первые эффективные эвристики для поиска в пространстве доверительных состояний, которые позднее были уточнены Брайсом и др. ([329], 2006). Инкрементный подход к поиску в пространстве доверительных состояний, при котором решения образуются пошагово для подмножества состояний в пределах каждого доверительного состояния, был изучен в работе ([1328], 2002). Рассел и Вольф ([1948], 2005) предложили несколько новых инкрементных алгоритмов для недетерминированных, частично наблюдаемых задач. Дополнительные ссылки на литературу по вопросу планирования в стохастических, частично наблюдаемых средах будут приведены в главе 17.

Алгоритмы исследования неизвестных пространств состояний привлекали интерес ученых в течение многих столетий. Поиск в глубину в лабиринте, допускающем возвраты, можно осуществить, постоянно держась левой рукой за стену; циклов можно избежать, отмечая каждую развилку. Более общая задача исследования ►**графов Эйлера** (т.е. графов, в которых каждый узел имеет одинаковое число входящих и исходящих ребер) была решена с помощью алгоритма, предложенного Хирхольцером ([1020], 1873).

Первое исчерпывающее описание алгоритмов решения задачи исследования для произвольных графов было приведено в [605] (1990); авторы этой работы предложили абсолютно общий алгоритм, но показали, что для задачи исследования графа общего вида невозможно определить ограниченный коэффициент конкурентоспособности. В [1728] (1991) приведены результаты исследования задачи поиска путей к цели в геометрических вариантах среды планирования пути (где все действия являются обратимыми). Эти результаты показывают, что достичь того, чтобы коэффициент конкурентоспособности оставался небольшим, можно лишь при наличии квадратных препятствий, а когда препятствия имеют произвольную прямоугольную форму, невозможно добиться, чтобы значения коэффициента конкурентоспособности оставались ограниченными (см. рис. 4.20).

В динамичной среде состояние мира может самопроизвольно меняться без каких-либо действий со стороны агента. Например, агент может спланировать оптимальный маршрут проезда от A до B, но авария или необычно плотный трафик в час пик может помешать выполнению его плана. Алгоритмы инкрементного поиска, такие как LPA* ([1261], 2004) и D* Lite ([1260], 2002), позволяют справиться с подобной ситуацией.

Алгоритм LRTA* был разработан Корфом ([1286], 1990) в ходе исследований в области **поиска в реальном времени** для вариантов среды, в которых агент должен совершить действие после проведения поиска лишь в течение ограниченного времени (эта ситуация встречается гораздо чаще в играх с двумя участниками). По сути, алгоритм LRTA* представляет собой частный случай алгоритмов обучения с подкреплением для стохастических вариантов среды ([138], 1995). Применяемый в нем принцип оптимистического отношения к неопределенности — всегда направляться к ближайшему непосещенному состоянию — может в случае отсутствия информации о среде привести к формированию такого подхода к исследованию, который является менее эффективным по сравнению с простым поиском в глубину ([1258], 2000). В [528] (1994) показано, что поиск с итеративным углублением в оперативном режиме обладает оптимальной эффективностью поиска цели в однородном дереве при отсутствии эвристической информации.

В сочетании с различными методами поиска и обновления в известной части графа было разработано несколько вариантов алгоритма LRTA*, предусматривающих использование информации о среде ([1769], 1992). Тем не менее еще нет полного понимания того, как следует находить цели с оптимальной эффективностью при использовании эвристической информации. В [2145] (2016) предлагается анализ некоторых подводных камней, возникающих на практике.

Упражнения

4.1. Укажите имя алгоритма, который будет получен в каждом из следующих особых случаев.
   а) Локальный поиск по лучу с $k = 1$.
   б) Локальный поиск по лучу с одним начальным состоянием и без ограничения количества хранимых состояний.
   в) Поиск имитацией отжига с фиксированным $T = 0$ (и без проверки завершения).
   г) Поиск имитацией отжига с фиксированным $T = \infty$.
   д) Генетический алгоритм с размером популяции $N = 1$

4.2. В упражнении 3.19 рассматривается задача сборки игрушечных железнодорожных путей в предположении, что соединяемые части подходят точно, без зазоров и искривлений. А теперь обратимся к реальной задаче, в которой части подходят не абсолютно точно и допускается отклонение до 10° в любую сторону от правильного направления. Объясните, как сформулировать подобную задачу сборки, чтобы ее можно было решить методом поиска имитацией отжига.

4.3. В этом упражнении рассматривается использование локальных методов поиска для решения задач TSP такого типа, который определен в упражнении 3.38.
   а) Предложите, реализуйте и протестируйте подход к решению задач TSP методом поиска восхождением к вершине. Сравните полученные результаты с оптимальными решениями, полученными на основе алгоритма A* с эвристической функцией MST (упражнение 3.38).
   б) Повторите задание из варианта а, используя генетический алгоритм вместо поиска восхождением к вершине..

4.4. Сформируйте большое количество экземпляров головоломки ***Восемь*** и задачи о восьми ферзях, а затем найдите их решения (где это возможно) с помощью поиска восхождением к вершине (в варианте подъема по самому крутому склону и в варианте поиска по первому наилучшему совпадению), затем — с помощью поиска восхождением к вершине со случайным перезапуском и поиска имитацией отжига. Измерьте стоимость поиска и процентное соотношение решенных задач, а затем нанесите эти данные на график и сопоставьте их с данными об оптимальной стоимости решения. Прокомментируйте полученные вами результаты.

4.5. В алгоритме поиска в графах И–ИЛИ, представленном на рис. 4.11, появление повторяющихся узлов проверяется только на пути от корня до текущего состояния. Предположим, что, кроме этого, алгоритм должен хранить сведения о каждом посещенном узле и сверяться с этим списком (пример см. на рис. 3.9). Определите информацию, которая должна храниться в таком списке, и то, как алгоритм будет ее использовать при обнаружении повторяющегося узла. (Подсказка: необходимо по крайней мере отличать узлы, для которых ранее был создан успешный подплан, от узлов, для которых найти подплан невозможно.) Объясните, как можно использовать метки (см. раздел 4.3.3), чтобы избежать хранения нескольких копий подпланов.

4.6. Дайте точные пояснения, как можно изменить алгоритм поиска в графах И–ИЛИ таким образом, чтобы он генерировал циклический план, если плана без циклов не существует. Вам потребуется решить три проблемы. Во-первых, пометить этапы плана так, чтобы циклический план мог ссылаться на предшествующие части плана. Во-вторых, модифицировать функцию ИЛИ-поиска так, чтобы она продолжила поиск ациклических планов после нахождения циклического плана. В-третьих, расширить представление плана таким образом, чтобы можно было указать, является ли план циклическим. Продемонстрируйте, как модифицированный алгоритм будет работать в а) ненадежном мире пылесоса, б) в ненадежном и непредсказуемом мире пылесоса. Возможно, потребуется использовать компьютерную реализацию для проверки полученных результатов.

4.7. В разделе 4.3 для решения бессенсорных задач было введено понятие доверительного состояния. Последовательность действий решает бессенсорную задачу, если она отображает каждое физическое состояние в начальном доверительном состоянии b в целевое состояние. Предположим, что агент знает $h^*(s)$ — истинную оптимальную стоимость решения для физического состояния s в полностью наблюдаемой задаче — для каждого состояния s в начальном доверительном состоянии b. Найдите допустимую эвристику $h(b)$ для бессенсорной задачи в терминах этих стоимостей и докажите ее допустимость. Прокомментируйте точность этой эвристики на примере бессенсорной задачи в мире пылесоса, представленного на рис. 4.14. Насколько хорошо работает поиск A*?

4.8. В этом упражнении исследуются отношения "подмножество–надмножество" между доверительными состояниями в бессенсорных или частично наблюдаемых средах.

а) Докажите, что если последовательность действий является решением для доверительного состояния b, то она также будет решением для любого подмножества b. Можно ли что-нибудь сказать о надмножествах b?

б) Подробно объясните, как изменить граф поиска бессенсорной задачи, чтобы воспользоваться преимуществами, определяемыми ответами на вариант а.

в) Подробно объясните, как можно модифицировать алгоритм И–ИЛИ-поиска для частично наблюдаемых задач, помимо изменений, предложенных в варианте б.

4.9. В разделе 4.4.1 предполагалось, что некоторое действие будет иметь одну и ту же стоимость при выполнении в любом физическом состоянии в пределах данного доверительного состояния. (В результате получается задача поиска в пространстве доверительных состояний с четко определенными затратами на этап.) Теперь рассмотрим, что произойдет, если это предположение не выполняется. Понятие оптимальности в этом контексте все еще имеет смысл или требует модификации? Рассмотрите различные возможные определения "стоимости" выполнения действия в доверительном состоянии. Например, можно использовать минимальное значение физических затрат или максимальное либо интервал затрат с нижней границей, определяемой минимальной стоимостью и верхней границей, определяемой максимальной; либо просто сохранить весь набор всех возможных затрат для данного действия. Для каждого из этих вариантов выясните, будет ли поиск A* (с модификациями при необходимости) возвращать оптимальные решения.

4.10. Рассмотрите бессенсорную версию непредсказуемого мира пылесоса. Нарисуйте пространство доверительных состояний, достижимых из исходного доверительного состояния {1, 2, 3, 4, 5, 6, 7, 8}, и объясните, почему задача неразрешима.

4.11. Рассмотрите бессенсорную версию непредсказуемого мира пылесоса. Нарисуйте пространство доверительных состояний, достижимых из исходного доверительного состояния {1, 3, 5, 7}, и объясните, почему задача неразрешима.

4.12. Задачу навигации в упражнении 3.9 можно превратить в среду следующим образом.

– Восприятие будет списком позиций видимых вершин относительно агента. При этом оно не должно включать положение робота! Роботу потребуется самостоятельно определять свою позицию на карте. На данный момент достаточно предположить, что каждое местоположение имеет свой, отличный от других "вид".

– Каждое действие будет представлено вектором, описывающим прямой путь, по которому робот будет следовать. Если на пути нет препятствий, действие завершается успешно, в противном случае робот останавливается в точке, где его путь впервые перекрывается препятствием. Если агент возвращает нулевой вектор движения и находится у цели (которая является фиксированной и известной), то среда телепортирует агента в случайно выбранное место (не внутри препятствия).

– Показатель эффективности взимает с агента 1 балл за каждую пройденную единицу пути и присуждает 1000 баллов при каждом достижении цели.

а) Реализуйте эту среду и агента для решения задач в ней. После каждой телепортации агент должен будет сформулировать новую задачу, которая должна включать и проблему определения его текущего местоположения.

б) Обеспечьте документирование данных об эффективности вашего агента (заставив агента генерировать подходящие комментарии по мере его движения) и составьте отчет о его эффективности более чем в 100 проходах.

в) Измените среду так, чтобы агент 30% времени оказывался в непредусмотренном месте назначения (выбирается случайным образом из других видимых вершин, если таковые имеются; в противном случае агент вообще не перемещается). Это грубая модель ошибок движения реального робота. Измените агента таким образом, чтобы при обнаружении подобной ошибки он знал, где оказался, а затем создавал план, чтобы вернуться туда, где был, и возобновить выполнение прежнего плана. Не забывайте, что иногда попытка возвращения на прежнее место также может оказаться неудачной! Приведите пример того, как агент успешно

преодолел две последовательные ошибки в движении и в конечном счете достиг цели.

г) Теперь попробуйте две разные схемы восстановления после ошибки. 1) Найдите ближайшую вершину на исходном маршруте и отправьте робота к ней. 2) Перепланируйте маршрут к цели из нового местоположения. Сравните производительность трех схем восстановления. Повлияет ли включение затрат на поиск на результаты этого сравнения?

д) Теперь предположим, что в среде есть местоположения, в которых "вид" идентичен. (Например, предположим, что данный мир — это сетка с квадратными препятствиями.) С какой проблемой в этом случае сталкивается агент? Как выглядят решения?

4.13. Предположим, что агент находится в среде лабиринта 3×3, представленного на рис. 4.19. Агенту известно, что его первоначальным местонахождением является квадрат (1,1), что цель находится в квадрате (3,3) и что четыре действия, *Up* (вверх), *Down* (вниз), *Left* (влево) и *Right* (вправо), приводят к своим обычным последствиям, если не заблокированы стеной. Однако агент не знает, где находятся внутренние стены. В любом конкретном состоянии агент воспринимает информацию о множестве допустимых действий. Он также может определить, является ли данное состояние тем, которое он посещал ранее, или он оказался в новом состоянии.

а) Объясните, каким образом к этой задаче поиска в оперативном режиме можно применить такую трактовку, чтобы она могла рассматриваться как поиск в автономном режиме в пространстве доверительных состояний, где первоначальное доверительное состояние включает все возможные конфигурации среды. Насколько велико это первоначальное доверительное состояние? Насколько велико все пространство доверительных состояний?

б) Сколько различных вариантов восприятия возможно в первоначальном состоянии?

в) Опишите первые несколько ветвей плана действий в непредвиденных ситуациях для этой задачи. Насколько велик (приблизительно) этот план в полном объеме?

Обратите внимание на то, что этот план действий в непредвиденных ситуациях является решением для любой возможной среды, соответствующей данному описанию. Поэтому чередование поиска и выполнения не будет строго необходимо даже в неизвестных вариантах среды.

4.14. Предположим, что агент находится в среде лабиринта 3×3, представленного на рис. 4.19. Агенту известно, что его первоначальным местонахождением является квадрат (3,3), что цель находится в квадрате (1,1) и что четыре действия, *Up* (вверх), *Down* (вниз), *Left* (влево) и *Right* (вправо), приводят к своим обычным последствиям, если не заблокированы стеной. Однако агент не знает, где находятся внутренние стены. В любом конкретном состоянии агент воспринимает информацию о множестве допустимых действий. Он также может определить, является ли данное состояние тем, которое он уже посещал ранее, или он оказался в новом состоянии.

а) Объясните, каким образом к этой задаче поиска в оперативном режиме можно применить такую трактовку, чтобы она могла рассматриваться как поиск в автономном режиме в пространстве доверительных состояний, где первоначальное доверительное состояние включает все возможные конфигурации среды. Насколько велико это первоначальное доверительное состояние? Насколько велико все пространство доверительных состояний?

б) Сколько различных вариантов восприятия возможно в первоначальном состоянии?
в) Опишите первые несколько ветвей плана действий в непредвиденных ситуациях для этой задачи. Насколько велик (приблизительно) этот план в полном объеме?

Обратите внимание на то, что этот план действий в непредвиденных ситуациях является решением для любой возможной среды, соответствующей данному описанию. Поэтому чередование поиска и выполнения не будет строго необходимо даже в неизвестных вариантах среды.

4.15. В данном упражнении исследуется применение поиска восхождением к вершине в контексте навигации робота с использованием среды, показанной на рис. 3.29.

а) Повторите упражнение 4.12 с использованием поиска восхождением к вершине. Зашел ли когда-либо предложенный вами агент в тупик в локальном минимуме? Возможно ли, чтобы он зашел в тупик при наличии выпуклых препятствий?
б) Постройте среду с препятствиями в виде невыпуклых многоугольников, в которой агент может оказаться в тупике.
в) Модифицируйте алгоритм поиска восхождением к вершине таким образом, чтобы вместо выполнения поиска на глубину 1 при принятии решения о том, куда двигаться дальше, агент предпринимал поиск на глубину k. Он должен найти наилучший путь, состоящий из k этапов, пройти по нему один этап, а затем повторить весь процесс.
г) Существует ли такое значение k, при котором новый алгоритм *гарантирует* выход из локальных минимумов?
д) Объясните, благодаря чему алгоритм LRTA* позволяет агенту выходить из локальных минимумов в таких ситуациях.

4.16. Как и алгоритм DFS, вариант алгоритма DFS для поиска в оперативном режиме является неполным для обратимых пространств состояний с бесконечными путями. Например, предположим, что состояния — это точки на бесконечной двухмерной сетке, а действия представляют собой единичные векторы (1, 0), (0, 1), (–1, 0), (0, –1), проверяемые в этом порядке. Покажите, что алгоритм DFS при поиске в оперативном режиме, начав выполнение в состоянии (0, 0), никогда не достигнет состояния (1, –1). Теперь предположим, что агент может наблюдать в дополнение к своему текущему состоянию все состояния-преемники и действия, которые к ним приведут. Напишите алгоритм, который будет полным даже для двунаправленных пространств состояний с бесконечными путями. Какие состояния этот алгоритм посетит при достижении состояния (1, −1)?

4.17. Определите соотношение временной сложности алгоритма LRTA* с его пространственной сложностью.

ГЛАВА 5

Поиск в условиях противодействия и игры

В этой главе рассматриваются среды, в которых противоборствующие агенты составляют планы, направленные против нас.

В этой главе обсуждаются **конкурентные среды**, в которых два или более агентов имеют противоположные цели, что ведет к возникновению задач ▶**поиска в условиях противодействия**. Однако вместо того, чтобы разбираться в хаосе противоречий и противостояний реального мира, мы сосредоточимся на играх, таких как шахматы, го и покер. Для исследователей в области ИИ упрощенная природа этих игр является большим плюсом: состояние игры достаточно легко представить, а агенты обычно ограничены небольшим количеством возможных действий, последствия которых определены точными правилами. Физические игры, такие как крокет или хоккей на льду, имеют более сложные описания, более широкий диапазон возможных действий и довольно неточные правила, определяющие допустимость действий. За исключением футбола роботов, различные физические игры никогда не вызывали особого интереса в сообществе исследователей ИИ.

5.1. Теория игр

Имеется как минимум три положения, которые могут быть приняты в отношении многоагентных сред. Первое положение, подходящее для ситуации, когда в среде действует очень большое количество агентов, заключается в том, чтобы рассматривать их в совокупности как ▶**экономику**, а не игру, что позволит, например, предсказывать, что увеличение спроса приведет к росту цен, без необходимости прогнозировать действие любого отдельного агента.

Согласно второму положению противоборствующих агентов можно рассматривать просто как часть среды, что делает ее недетерминированной. Однако если смоделировать противников таким же образом, как, скажем, выпадение дождя, который иногда идет, а иногда нет, то будет упущена важная идея, что противники

любого агента активно пытаются его победить, тогда как дождь, предположительно, не имеет такого намерения.

Третье положение состоит в том, чтобы явно моделировать противоборствующих агентов с помощью методов состязательного поиска по игровому дереву. И это именно то, о чем пойдет речь в данной главе. Мы начнем обсуждение с ограниченного класса игр, понятия оптимального хода и некоторых алгоритмов его поиска: минимаксный поиск и обобщенный И–ИЛИ-поиск (см. рис. 4.11). Затем будет показано, что ▶**отсечение** делает поиск более эффективным за счет игнорирования тех частей дерева поиска, которые не оказывают влияния на окончательный выбор. В случае нетривиальных игр участники обычно не располагают достаточным количеством времени, чтобы искать гарантированно оптимальный ход (даже с отсечением), поэтому в какой-то момент им приходится просто прекращать поиск.

Для каждого состояния, в котором было решено прекратить поиск, следует поставить вопрос, кто выигрывает. Чтобы получить на него ответ, можно либо воспользоваться эвристической **функцией оценки** того, кто окажется в выигрыше, исходя из особенностей данного состояния (раздел 5.3), либо усреднить результаты многократного быстрого моделирования хода игры из этого состояния по всем путям до конечного (раздел 5.4).

В разделе 5.5 рассматриваются игры, включающие элемент случайности (за счет бросания костей или перетасовки колоды карт), а в разделе 5.6 — игры с ▶**неполной информацией** (такие, как покер и бридж, в которых не все карты видны всем игрокам).

5.1.1. Игры с нулевой суммой для двух игроков

Те игры, которые чаще всего изучаются в исследованиях в области ИИ (например, шахматы или го), теоретики называют детерминированными, с двумя игроками, с последовательными ходами, ▶**с полной информацией** и ▶**нулевой суммой**. Термин "полная информация" — это синоним понятия "полностью наблюдаемая",[1] а "нулевая сумма" означает, что то, что хорошо для одного игрока, обязательно будет плохо для других, в таких играх нет "беспроигрышного" результата. Говоря об играх, мы часто будем использовать слово ▶**ход** как синоним для понятия "действие", а слово ▶**позиция** как синоним для понятия "состояние".

Обсуждая такие игры, будем называть двух игроков MAX и MIN по причинам, которые вскоре станут очевидными. Первым всегда ходит MAX, а затем игроки ходят

[1] Некоторые авторы делают здесь различие, используя термин "игры с неполной информацией" для игр типа покера, в которых игроки имеют конфиденциальную информацию о картах у себя на руках, которой другие игроки не имеют, и термин "частично наблюдаемые игры" для игр типа StarCraft II , в которых каждый игрок может видеть происходящее в его ближайшем окружении, но не видит прочих удаленных участков игровой среды.

по очереди, пока игра не закончится. В конце игры выигравшему игроку начисляются очки, а на проигравшего налагается штраф. Формально игра может быть определена следующими элементами.

- S_0. **Начальное состояние**, определяющее стартовую позицию игры.
- To-Move(s). Функция, определяющая игрока, которому принадлежит право совершить ход в состоянии s.
- Actions(s). Функция, определяющая множество допустимых ходов в состоянии s.
- Result(s, a). ▶**Функция определения преемника**, возвращающая результирующее состояние после выполнения действия a в состоянии s.
- Is-Terminal(s). Функция ▶**проверки терминального состояния**, возвращающая значение *true*, если игра окончена, и *false* в противном случае. Состояния, в которых игра закончена, называются ▶**терминальными состояниями**.
- Utility(s, p). ▶**Функция полезности** (также называемая целевой функцией или функцией вознаграждения), определяющая итоговое числовое значение для игрока p, когда игра заканчивается в терминальном состоянии s. В шахматах результатом может быть выигрыш, проигрыш или ничья с числовыми значениями 1, 0 или 1/2 соответственно.[2] Некоторые игры имеют более широкий диапазон возможных результатов, например в нардах количество очков может иметь любое значение в диапазоне от 0 до 192.

Как и в главе 3, начальное состояние, функция Actions и функция Result определяют ▶**граф пространства состояний** — граф, в котором вершины представляют состояния, ребра соответствуют действиям и какое-либо состояние может быть достигнуто по нескольким путям. Как и в главе 3, чтобы определить, какой ход следует сделать, на часть этого графа можно наложить ▶**дерево поиска**. Определим полное ▶**дерево игры** как дерево поиска, которое фиксирует любую последовательность ходов вплоть до терминального состояния. Дерево игры может быть бесконечным, если пространство состояний само по себе является неограниченным или если правила игры допускают бесконечно повторяющиеся позиции.

На рис. 5.1 представлена часть дерева игры в крестики-нолики. В начальном состоянии игроку MAX доступно девять возможных ходов. Ходы игроков чередуются, причем при каждом ходе MAX ставит значок "X", а MIN — значок "O". Игра продолжается до тех пор, пока будут достигнуты листовые узлы, соответствующие терминальным

[2] Шахматы считаются игрой "с нулевой суммой" несмотря даже на то, что сумма результатов для двух игроков в каждой (турнирной) игре равна +1, а не нулю. В этом случае более удачным термином было бы "с постоянной суммой", однако традиционным является первый вариант и он даже может иметь смысл, если представить, что перед началом игры каждый игрок делает начальный взнос в размере 1/2.

состояниям, в которых один из игроков поставил три своих значка в один ряд или все клетки поля уже заполнены. Число под каждым листовым узлом указывает значение функции полезности для этого терминального состояния с точки зрения игрока MAX. Предполагается, что высокие значения благоприятны для игрока MAX и неблагоприятны для игрока MIN (вот почему в теории этим игрокам присвоены такие имена).

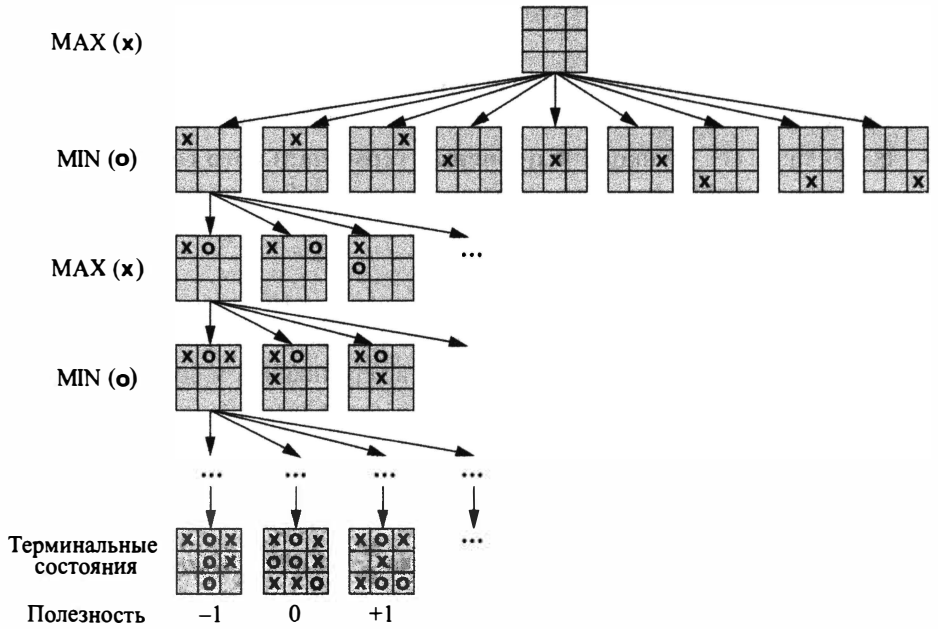


Рис. 5.1. (Частичное) дерево поиска для игры "крестики-нолики". Верхний узел представляет собой начальное состояние, первым ходит игрок MAX, ставя значок X в любой пустой клетке. На этом рисунке показана часть дерева поиска, в которой демонстрируются чередующиеся ходы игроков MIN (значок O) и MAX (значок X). Ходы выполняются до тех пор, пока в конечном итоге будет достигнуто одно из терминальных состояний, которым могут быть назначены данные о полезности в соответствии с правилами игры

Для игры в крестики-нолики дерево игры будет относительно небольшим — в нем меньше чем 9! = 362 880 терминальных узлов (из них только 5478 представляют различные состояния). Но для шахмат существует более 10^{40} узлов, так что в этом случае дерево игры лучше понимать как сугубо теоретическую конструкцию, которую невозможно реализовать в физическом мире.

5.2. Принятие оптимальных решений в играх

Игрок MAX хочет найти последовательность действий, ведущих к победе, но игроку MIN также есть что сказать по этому поводу. А это означает, что стратегия игрока MAX должна представлять собой условный план — зависящую от обстоятельств стратегию, определяющую ответ на любой из возможных ходов игрока MIN. В играх, которые имеют бинарный результат (выигрыш или проигрыш), для генерации условного плана можно использовать И–ИЛИ-поиск (раздел 4.3.2). Фактически для таких игр определение выигрышной стратегии идентично отысканию решения задачи планирования в недетерминированной среде: в обоих случаях желаемый результат должен быть гарантирован независимо от того, что делает "другая сторона". Для игр со многими возможными значениями результата необходимо использовать более общий алгоритм, носящий название ▶**минимаксный поиск**.

Рассмотрим простейшую игру, представленную на рис. 5.2. Возможные ходы игрока MAX из корневого узла обозначены как a_1, a_2 и a_3. Возможными ответными ходами на ход a_1 для игрока MIN являются b_1, b_2, b_3 и т.д. Данная конкретная игра заканчивается после того, как каждый игрок, MAX и MIN, делают по одному ходу. (Согласно терминологии теории игр, это дерево имеет глубину в *один* ход, который состоит из ходов, сделанных двумя участниками, каждый из которых называется ▶**полуходом** и вызывает переход на один уровень ниже в дереве игры.) Полезности терминальных состояний в этой игре меняются в пределах от 2 до 14.

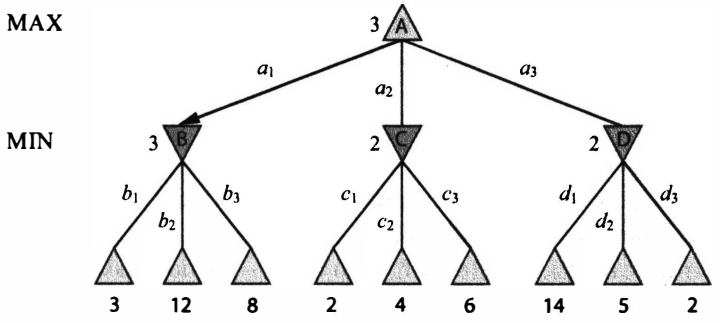


Рис. 5.2. Дерево игры с двумя игроками. Узлы в виде светлых треугольников вершиной вверх представляют право хода игрока MAX, а узлы в виде темных треугольников вершиной вниз — право хода игрока MIN. Для терминальных узлов в нижнем ряду приведены значения полезности для игрока MAX, а для остальных узлов — их минимаксные значения. Для игрока MAX лучший ход в корневом узле — a_1, поскольку он ведет к состоянию с наивысшим минимаксным значением. Лучшим ответным ходом игрока MIN в этом случае будет b_1, так как он приводит к состоянию с наименьшим минимаксным значением

При наличии дерева игры оптимальную стратегию можно определить, исследуя ▶**минимаксное значение** каждого узла в дереве, которое можно записать как MINIMAX(s). Минимаксным значением узла является полезность (для игрока MAX) пребывания в данном состоянии *при условии, что оба игрока делают ходы оптимальным образом* от этого узла и до конца игры. Понятно, что минимаксным значением терминального состояния является просто его полезность. В нетерминальном состоянии игроку MAX следует выбирать ход, ведущий в состояние с максимальным значением, а игроку MIN — ведущий в состояние с минимальным значением (т.е. с минимальным значением для MAX, а значит, с максимальным для MIN). Таким образом, имеет место следующее соотношение.

$$\text{MINIMAX}(s) = \begin{cases} \text{UTILITY}(s, \text{MAX}) & \text{если } \text{IS-TERMINAL}(s) \\ \max_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{если } \text{TO-MOVE}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{если } \text{TO-MOVE}(s) = \text{MIN} \end{cases}$$

Применим эти определения к дереву игры, показанному на рис. 5.2. Для терминальных узлов на нижнем уровне уже приведены числовые значения, определяющие их полезность. Первый узел игрока MIN, обозначенный как B, имеет трех преемников со значениями 3, 12 и 8, поэтому его минимаксное значение равно 3. Аналогичным образом другие два узла игрока MIN имеют минимаксное значение, равное 2. Корневым узлом является узел игрока MAX; его преемники имеют минимаксные значения 3, 2 и 2, поэтому сам корневой узел имеет минимаксное значение 3. Можно также определить понятие ▶**минимаксного решения**, принимаемого в корне дерева: действие a_1 является оптимальным выбором для игрока MAX, поскольку ведет к преемнику с наивысшим минимаксным значением.

Однако это вовсе не означает, что всегда лучше выбирать оптимальные минимаксные ходы при игре с противником, действующим неоптимально. Рассмотрим ситуацию, когда оптимальная игра с обеих сторон приводит к ничьей, но есть один рискованный ход для игрока MAX, приводящий к состоянию, в котором имеется 10 возможных ответных ходов игрока MIN, все из которых кажутся разумными, но 9 из них ведут к потерям для MIN и только один — к потерям для MAX. Если MAX полагает, что MIN не имеет необходимой вычислительной мощности для обнаружения единственного оптимального хода, он, возможно, решится на рискованный шаг на том основании, что шанс 9/10 на выигрыш — это лучше, чем неизбежная ничья.

5.2.1. Минимаксный алгоритм поиска

Теперь, когда функция MINIMAX(s) определена, можно использовать ее для построения алгоритма поиска, определяющего лучший ход для игрока MAX, проверяя все возможные действия и выбрав из них то, которое приводит к состоянию с максимальным значением MINIMAX. На рис. 5.3 приведен такой алгоритм.

Это рекурсивный алгоритм, который проходит весь путь до листовых узлов дерева игры, а затем минимаксные значения просчитываются и ▶**резервируются** при движении в обратном направлении по всему дереву по мере поэтапного свертывания рекурсии. Например, для дерева, показанного на рис. 5.2, алгоритм вначале выполнил рекурсию с переходом на нижние уровни до трех нижних левых узлов и применил к ним функцию полезности UTILITY, чтобы определить, что значения полезности этих узлов равны 3, 12 и 8 соответственно. Затем алгоритм определил минимальное из этих значений, 3, и возвратил его в качестве зарезервированного значения узла B. Аналогичный процесс позволяет получить зарезервированные значения 2 для узла C и 2 — для узла D. Наконец, берется максимальное из значений 3, 2 и 2 для получения зарезервированного значения 3 корневого узла.

```
function MINIMAX-SEARCH(game, state) returns действие action
  player ← game.TO-MOVE(state)
  value, move ← MAX-VALUE(game, state)
  return move

function MAX-VALUE(game, state) returns пара значений (utility, move)
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← −∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MIN-VALUE(game, game.RESULT(state, a))
    if v2 > v then
      v, move ← v2, a
    return v, move

function MIN-VALUE(game, state) returns пара значений (utility, move)
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← +∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MAX-VALUE(game, game.RESULT(state, a))
    if v2 < v then
      v, move ← v2, a
  return v, move
```

Рис. 5.3. Алгоритм расчета оптимального хода с использованием минимаксных значений: выбирается ход, ведущий к терминальному состоянию с наивысшей полезностью в предположении, что противник стремится минимизировать эту полезность. Функции MAX-VALUE и MIN-VALUE используются для прохождения через все дерево игры вплоть до листовых узлов, чтобы определить зарезервированное значение состояния и ведущего к нему хода.

Этот минимаксный алгоритм выполняет полное исследование дерева игры по методу поиска в глубину. Если максимальная глубина дерева равна m и в каждом состоянии имеются b допустимых ходов, то временная сложность этого минимаксного алгоритма составляет $O(bm)$. Для алгоритма, который формирует сразу всех преемников, пространственная сложность будет равна $O(bm)$, а для алгоритма, который формирует преемников по одному, — $O(m)$ (см. раздел 3.4.3). Экспоненциальная временная сложность делает использование алгоритма MINIMAX для достаточно сложных игр непрактичным. Например, для шахмат фактор ветвления b равен приблизительно 35, а средняя игра имеет глубину m около 80 полуходов, а значит, будет абсолютно невозможно провести поиск по $35^{80} \approx 10^{123}$ состояниям. Тем не менее алгоритм MINIMAX широко используется в качестве основы математического анализа игр. Также, аппроксимируя минимаксный анализ различными способами, можно получить значительно более практичные алгоритмы.

5.2.2. Оптимальные решения в играх с несколькими игроками

Многие популярные игры допускают наличие больше чем двух игроков. Рассмотрим, как можно распространить идею минимаксного алгоритма на игры с несколькими игроками. С точки зрения технической реализации это сделать несложно, но при этом возникают некоторые новые и интересные концептуальные проблемы.

Вначале необходимо заменить единственное значение для каждого узла *вектором* значений. Например, в игре с тремя игроками, в которой участвуют игроки A, B и C, с каждым узлом ассоциируется вектор $\langle v_A, v_B, v_C \rangle$. Для терминальных состояний этот вектор задает полезность данного состояния с точки зрения каждого игрока. (В играх с двумя игроками и нулевой суммой двухэлементный вектор может быть сокращен до единственного значения, поскольку значения в нем всегда противоположны.) Простейшим способом реализации такого подхода является применение функции UTILITY, которая возвращает вектор значений полезности.

Теперь необходимо рассмотреть нетерминальные состояния. Рассмотрим узел, который в дереве игры, показанном на рис. 5.4, отмечен символом X. В этом состоянии игрок C выбирает, что ему делать. Два возможных варианта ведут к терминальным состояниям с векторами полезности $\langle v_A = 1, v_B = 2, v_C = 6 \rangle$ и $\langle v_A = 4, v_B = 2, v_C = 3 \rangle$. Поскольку 6 больше, чем 3, игрок C должен выбрать первый ход. Это означает, что если достигнуто состояние X, то дальнейшая игра приведет к терминальному состоянию с полезностями $\langle v_A = 1, v_B = 2, v_C = 6 \rangle$. Следовательно, зарезервированным значением для узла X является этот вектор. Вообще говоря, зарезервированное значение узла n представляет собой вектор полезности такого узла-преемника, который имеет наивысшее значение для игрока, выбирающего ход в узле n.

Любой, кто играет в игры с несколькими игроками, такие как "Дипломатия" или "Колонизаторы", быстро осознает, что в них происходит гораздо больше событий, чем в играх с двумя игроками. В играх с несколькими игроками между участниками обычно возникают ▶**альянсы**, формальные или неформальные. По ходу игры они могут как создаваться, так и разрушаться. Как нам следует понимать такое поведение? Являются ли альянсы естественным следствием выбора оптимальных стратегий каждым из участников в играх с несколькими игроками? Как оказалось, они действительно могут стать таким следствием.

Например, предположим, что игроки *A* и *B* имеют слабые позиции, а игрок *C* — более сильную позицию. В таком случае и для *A*, и для *B* часто оптимальным оказывается решение атаковать игрока *C*, а не друг друга, поскольку иначе *C* ликвидирует каждого из них по отдельности. Таким образом, сотрудничество двух игроков становится следствием их чисто эгоистичного поведения. Понятно, что как только под совместным натиском противников игрок *C* ослабеет, альянс потеряет свой смысл и соглашение может нарушить либо игрок *A*, либо игрок *B*.

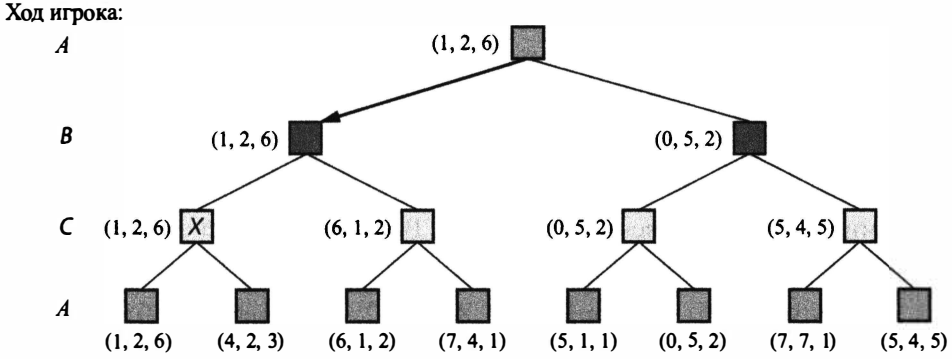


Рис. 5.4. Первые три полухода дерева игры с тремя игроками (*A*, *B*, *C*). Каждый узел обозначен значениями, достигаемыми с точки зрения каждого игрока. Наилучший ход из корня обозначен стрелкой

В одних случаях распад альянсов оказывается просто конкретным выражением того, что и так бы произошло. В других случаях попытка нарушить альянс может вызвать общественное осуждение, а значит, игрокам приходится делать выбор между немедленно достигаемой выгодой от нарушения альянса и потенциальным долговременным ущербом от того, что их будут считать не заслуживающими доверия. Дополнительная информация об этих сложностях в игре приведена в разделе 18.2.

Если игра не является игрой с нулевой суммой, то сотрудничество может возникать и при наличии всего двух игроков. Допустим, что в игре имеется

терминальное состояние с полезностями $\langle v_A = 1000, v_B = 1000 \rangle$ и что 1000 — максимальная возможная полезность для каждого игрока. В таком случае оптимальная стратегия для обоих игроков заключается в том, чтобы делать все возможное для достижения этого состояния, и это означает, что игроки автоматически вступают в сотрудничество для достижения обоюдно желаемой цели.

5.2.3. Альфа-бета-отсечение

Количество игровых состояний экспоненциально зависит от глубины дерева игры. Ни один алгоритм не может полностью исключить эту экспоненциальность, но иногда можно сократить дерево почти наполовину, вычислив правильное минимаксное решение даже без проверки каждого его узла благодаря **отсечению** (см. раздел 3.5.3) больших частей дерева, узлы в которых не оказывают никакого влияния на конечный результат. Конкретный метод, который будет рассмотрен в этом разделе, называется ►**альфа-бета-отсечением**.

Давайте вновь обратимся к дереву игры с двумя полуходами (см. рис. 5.2) и еще раз проведем расчет оптимального решения, но на этот раз обращая особое внимание на то, что известно на каждом этапе этого процесса. Все пояснения к выполняемым вычислениям приведены на рис. 5.5. Результат наших действий заключается в том, что можно найти минимаксное решение, даже не приступая к вычислению значений двух листовых узлов.

Этот подход может также рассматриваться иначе — как упрощение формулы для получения значения MINIMAX. Допустим, что два преемника узла C на рис. 5.5, не обработанные в процессе вычисления, имеют значения x и y, и предположим, что z — минимальное значение среди x и y. В таком случае значение корневого узла можно найти следующим образом:

$$\begin{aligned} \text{MINIMAX}(root) &= \max(\min(3, 12, 8), \min(2, x, y), \min(14, 5, 2)) \\ &= \max(3, \min(2, x, y), 2) \\ &= \max(3, z, 2) \qquad \text{где } z = \min(2, x, y) \leq 2 \\ &= 3. \end{aligned}$$

Другими словами, значение корневого узла, а следовательно, и минимаксное решение не зависит от значений отсеченных листовых узлов x и y.

Альфа-бета-отсечение может применяться к деревьям любой глубины; к тому же часто возникает возможность отсекать целые поддеревья, а не просто листья. Общий принцип состоит в следующем: рассмотрим узел n, находящийся где-либо в дереве (рис. 5.6), такой, что участник игры с нашей стороны (назовем его *Игрок*) имеет возможность выбрать ход, ведущий к этому узлу. Но если *Игрок* имеет лучший выбор m либо в родительском узле узла n, либо в любой другой точке выбора, находящейся еще выше в дереве, то узел n никогда не будет достигнут в игре, происходящей в действительности. Поэтому, получив

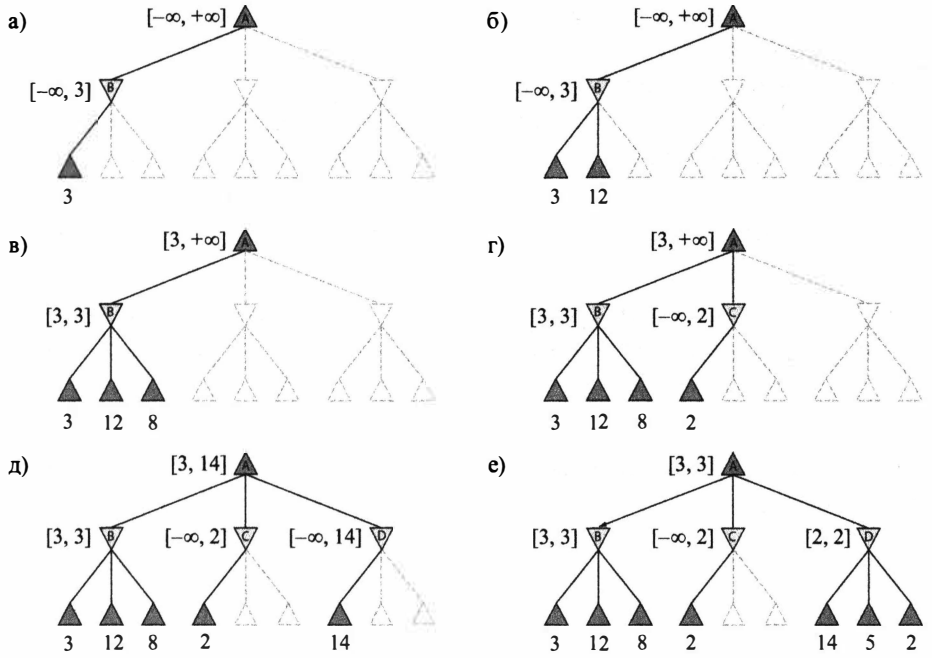


Рис. 5.5. Этапы вычисления оптимального решения для дерева игры, показанного на рис. 5.2. В каждой точке показан ряд возможных значений для каждого узла. **а)** Первый лист, расположенный ниже узла *B*, имеет значение 3; следовательно, узел *B*, который является узлом игрока MIN, имеет, самое большее, значение 3. **б)** Второй лист, расположенный ниже узла *B*, имеет значение 12, и игрок MIN должен избегать этого хода, поэтому значение *B*, все еще самое большее, равно 3. **в)** Третий лист, расположенный ниже узла *B*, имеет значение 8, — теперь проверены все преемники узла *B* и можно сделать вывод, что значение *B* в точности равно 3. Также теперь можно сделать вывод, что значение корневого узла, самое меньшее, равно 3, поскольку у игрока MAX здесь есть выбор со значением 3. **г)** Первый лист, находящийся ниже узла *C*, имеет значение 2; следовательно, *C*, являющийся узлом MIN, имеет, самое большее, значение 2. Но уже известно, что узел *B* позволяет игроку MAX достичь значения 3, поэтому он не станет выбирать узел *C*. Это означает, что алгоритму не имеет смысла проверять остальных преемников узла *C* — пример того, что называют альфа-бета-отсечением. **д)** Первый лист расположен ниже узла *D* и имеет значение 14, поэтому *D* имеет, самое большее, значение 14, и оно выше, чем наилучшая известная нам альтернатива для игрока MAX (т.е. 3). Следовательно, необходимо продолжить исследование преемников узла *D*. Отметим также, что теперь определены предельные значения всех преемников корневого узла, поэтому его значение также равно, самое большее, 14. **е)** Второй преемник узла *D* имеет значение 5, поэтому исследование придется продолжить. Значение третьего преемника равно 2; следовательно, значение *D* точно равно 2, а это значит, что в корневом узле игрок MAX должен принять решение сделать ход, ведущий к узлу *B*, что позволит ему получить результирующее значение 3

достаточно информации об узле *n* (путем исследования некоторых из его потомков), чтобы прийти к такому заключению, можно с уверенностью выполнить его отсечение.

Напомним, что минимаксный поиск осуществляется в глубину, поэтому в любой момент времени достаточно рассматривать узлы вдоль единственного пути в дереве. Алгоритм альфа-бета-отсечения получил свое название по двум своим дополнительным параметрам в функции MAX-VALUE(*state*, α, β), представляющим пределы в зарезервированных значениях, присутствующих во всех узлах вдоль этого пути.

α = значение наилучшего варианта (т.е. самое высокое значение), который был до сих пор найден в любой точке выбора вдоль пути для игрока MAX. Считайте: α = "по крайней мере".

β = значение наилучшего варианта (т.е. самое низкое значение), который был до сих пор найден в любой точке выбора вдоль пути для игрока MIN. Считайте: β = "не больше чем".

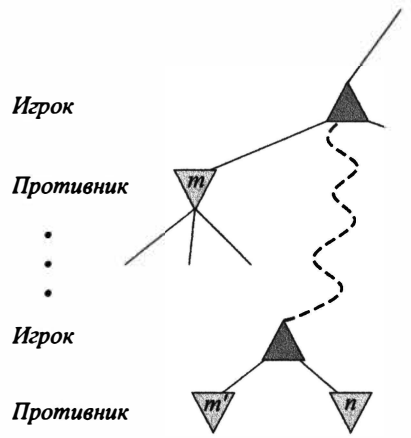


Рис. 5.6. Альфа-бета-отсечение: общий случай. Если для *Игрока* узел *m* лучше, чем *n*, то в игре он никогда не выберет узел *n*

В процессе работы алгоритм альфа-бета-поиска обновляет значения α и β и отсекает оставшиеся в узле ветви (т.е. прекращает рекурсивные вызовы), как только становится известно, что значение текущего узла хуже по сравнению с текущим значением α или β для игрока MAX или MIN соответственно. Полный алгоритм приведен на рис. 5.7. На рис. 5.5 приведена трассировка работы этого алгоритма применительно к дереву игры, показанному на рис. 5.2.

```
function ALPHA-BETA-SEARCH(game, state) returns действие action
  player ← game.TO-MOVE(state)
  value, move ← MAX-VALUE(game, state, −∞, +∞)
  return move

function MAX-VALUE(game, state, α, β) returns пара значений (utility, move)
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← −∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MIN-VALUE(game, game.RESULT(state, a), α, β)
    if v2 > v then
      v, move ← v2, a
      α ← MAX(α, v)
    if v ≥ β then return v, move
  return v, move

function MIN-VALUE(game, state, α, β) returns пара значений (utility, move)
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← +∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MAX-VALUE(game, game.RESULT(state, a), α, β)
    if v2 < v then
      v, move ← v2, a
      β ←MIN(β, v)
    if v ≤ α then return v, move
  return v, move
```

Рис. 5.7. Алгоритм альфа-бета-поиска. Обратите внимание, что используемые здесь функции остались такими же, как и функции алгоритма MINIMAX-SEARCH, приведенного на рис. 5.3, за исключением того, что границы в этом случае сохраняются в переменных α и β, которые затем используются для отсечения ветвей поиска, если рассматриваемое значение находится вне задаваемого ими диапазона

5.2.4. Упорядочивание ходов

Эффективность алгоритма альфа-бета-отсечения в большой степени зависит от того, в каком порядке происходит проверка преемников. Например, на рис. 5.5, *д*, *е* невозможно было бы вообще выполнить отсечение каких-либо преемников узла *D*, поскольку в первую очередь были бы сформированы наихудшие преемники (с точки зрения игрока MIN). А если бы в первую очередь был сформирован третий преемник со значением 2, то появилась бы возможность отсечь двух остальных. На основании этого можно сделать вывод, что имеет смысл стремиться исследовать первыми таких преемников, которые, вероятно, могут оказаться наилучшими.

Если бы это можно было сделать идеально, то для определения наилучшего хода алгоритму альфа-бета-отсечения достаточно было бы исследовать только $O(b^{m/2})$ узлов вместо $O(b^m)$ узлов, как при использовании минимаксного алгоритма. А это означает, что эффективный коэффициент ветвления становится равным $\sqrt{b}$, а не b, например для шахмат он равен 6, а не 35. Иными словами, за такое же время альфа-бета-поиск при правильном выборе ходов позволяет проанализировать дерево игры примерно в два раза глубже по сравнению с минимаксным поиском. Если же исследование преемников будет выполняться в случайном порядке, а не по принципу первоочередного выбора наилучших вариантов, то при умеренных значениях b общее количество исследованных узлов будет составлять примерно $O(b^{3m/4})$. Теперь становится очевидным, что мы не можем достичь точного упорядочивания ходов — в этом случае функцию упорядочения можно было бы использовать, чтобы сыграть идеальную игру! Однако часто к этой цели можно подобраться достаточно близко. В случае шахмат применение довольно простой функции упорядочения (например, сначала рассматриваются взятия фигур, затем — угрозы, далее — ходы вперед, а после этого — ходы назад) позволяет оставаться в пределах, не превышающих удвоенное значение результата при наилучшем упорядочивании $O(b^{m/2})$.

Добавление динамических схем упорядочения ходов, в частности таких, в которых в первую очередь проверяются ходы, обозначенные как наилучшие на предыдущем этапе, позволяет подойти совсем близко к этому теоретическому пределу. Как прошлое может рассматриваться предыдущий этап — часто сохраняются одни и те же угрозы — или сведения о прошлом могут поступать из предыдущего исследования текущего хода через процедуру **итеративного углубления** (см. раздел 3.4.4). Сначала поиск ведется в пределах одного полухода и составляется рейтинг ходов на основе полученных оценок. Затем поиск смещается на один полуход глубже и проводится с использованием ранее составленного рейтинга для упорядочивания анализируемых ходов и т.д. Увеличение времени поиска за счет итеративного углубления может оказаться больше, чем в случае простого упорядочивания ходов. Найденные лучшие ходы называют ▶**ходами-убийцами**, а тактика проверять их в первую очередь называется *эвристикой убийцы*.

Как было отмечено в разделе 3.3.3, наличие в дереве поиска избыточных путей к повторяющимся состояниям может вызвать экспоненциальное увеличение стоимости поиска. Чтобы справиться с этой проблемой, необходимо хранить таблицу ранее достигнутых состояний. При поиске по дереву игры повторяющиеся состояния появляются вследствие возникновения ▶**перестановок** — различных перестановок в последовательности ходов, оканчивающихся в одной и той же позиции. Эту проблему можно устранить созданием ▶**таблицы перестановок**, в которой кешируются эвристические значения состояний.

Например, пусть в шахматной партии белые имеют в своем распоряжении ход w_1, на который черные могут ответить ходом b_1, а также еще один не связанный с ним ход w_2 на другой стороне доски, на который может быть дан ответ b_2,

при этом последовательность ходов $[w_1, b_1, w_2, b_2]$ приводит игру к состоянию s. После исследования большого поддерева ниже состояния s для него было найдено зарезервированное значение, которое было сохранено в таблице перестановок. Далее, в процессе анализа последовательности ходов $[w_2, b_2, w_1, b_1]$ выяснилось, что она также приводит к состоянию s, но теперь вместо повторного выполнения поиска можно просто выбрать зарезервированное значение из таблицы перестановок. В шахматах использование таблиц перестановок оказалось очень эффективным приемом, часто позволяющим за то же самое время исследовать дерево игры на вдвое большую глубину.

Однако даже при использовании алгоритма альфа-бета-отсечения с разумным упорядочиванием ходов метод минимакса неэффективен для таких игр, как шахматы или го, поскольку за доступное время все еще потребуется исследовать слишком много состояний. В первой статье, посвященной игровым компьютерным программам, *Программирование компьютера для игры в шахматы* ([2040], 1950), Клод Шеннон признал эту проблему и предложили две стратегии. В ►**стратегии типа A** рассматриваются все возможные ходы на определенную глубину в дереве поиска, а затем эвристическая функция используется для оценки полезности состояний на этой глубине. В этом случае дерево игры исследуется *широко, но неглубоко*. В ►**стратегии типа B** игнорируются ходы, которые выглядят слабыми, и анализ ведется исключительно вдоль перспективных направлений "настолько далеко, насколько это возможно". Дерево игры исследуется *глубоко, но узко*.

Исторически сложилось так, что большинство шахматных программ принадлежали к типу A (он будет подробно обсуждаться в следующем разделе), в то время как программы игры в го по большей части относились к типу B (рассматривается в разделе 5.4), поскольку коэффициент ветвления в игре го значительно выше, чем в шахматах. В последнее время программы, принадлежащие к типу B, продемонстрировали игру на уровне чемпионов мира во многих играх, включая шахматы (Сильвер и др. [2064], 2018).

5.3. Эвристический альфа-бета-поиск по дереву

Чтобы уложиться в рамки ограниченного времени вычислений, можно раньше времени прекратить поиск и применить эвристическую **функцию оценки** к достигнутым состояниям, продуктивно обрабатывая нетерминальные узлы так, как если бы они были терминальными. Другими словами, идея состоит в замене функции UTILITY функцией EVAL, оценивающей полезность состояния. Также проверка терминального состояния заменяется ►**тестом останова**, возвращающего значение *true* (истина) для терминальных состояний, а в противном случае — принимающее решение, когда остановить поиск, исходя из заданной глубины поиска и любого свойства состояния, которое было выбрано для анализа. Это дает нам формулу H-MINIMAX(s, d) для определения эвристического минимаксного значения состояния s на глубине поиска d.

$$\text{H-MINIMAX}(s, d) = \begin{cases} \text{EVAL}(s, \text{MAX}) & \text{если } \text{IS-CUTOFF}(s,d) \\ \max_{a \in Actions(s)} \text{H-MINIMAX}(\text{RESULT}(s,a), d+1) & \text{если } \text{TO-MOVE}(s)=\text{MAX} \\ \min_{a \in Actions(s)} \text{H-MINIMAX}(\text{RESULT}(s,a), d+1) & \text{если } \text{TO-MOVE}(s)=\text{MIN} \end{cases}$$

5.3.1. Функции оценки

Эвристическая функция оценки EVAL(s, p) возвращает *оценку* ожидаемой полезности игры из состояния s для игрока p по аналогии с тем, как эвристические функции, описанные в главе 3, возвращают оценку расстояния до цели. Для терминальных состояний должно выполняться равенство EVAL(s, p) = UTILITY(s, p), а для нетерминальных состояний оценка должна быть где-то между *проигрышем* и *выигрышем*: UTILITY(*loss*, p) ≤ EVAL(s, p) ≤ UTILITY(*win*, p).

Помимо этих требований, что делает функцию оценки хорошей? Во-первых, ее вычисление не должно требовать слишком много времени! (Наша конечная цель — сделать поиск более быстрым.) Во-вторых, функция оценки должна быть тесно связана с реальными шансами на победу. Можно поинтересоваться: а что означает выражение "шансы на победу"? В конце концов, игра в шахматы не предполагает случайных элементов: о ее текущем состоянии известно абсолютно все и с полной уверенностью, бросание костей не предусмотрено и, если игрок сделает ошибку, результат заранее предопределен. Но если поиск должен останавливаться в нетерминальных состояниях, то алгоритм неизбежно *не будет иметь* точных сведений об окончательных исходах этих состояний (хотя теоретически эта неопределенность может быть устранена при наличии бесконечных вычислительных ресурсов).

Давайте сделаем эту идею более конкретной. Большинство функций оценки работает путем вычисления различных ►**характеристик** состояния, например в шахматах можно рассматривать как характеристики количество на доске белых пешек, черных пешек, белых ферзей, черных ферзей и т.д. Подобные характеристики, рассматриваемые совместно, могут определять различные *категории* или *классы эквивалентности* состояний: состояния в каждой категории будут иметь одинаковые значения всех принимаемых во внимание характеристик. Например, одна категория может включать все эндшпили с двумя пешками против одной пешки. Любая заданная категория будет включать некоторые состояния, приводящие (при правильной игре) к победе, другие — к ничьей, а остальные — к проигрышу.

Функция оценки не позволяет определить, какие состояния приводят к тому или иному результату, но способна вернуть единственное значение, оценивающее *процентное соотношение* состояний с каждым результатом. Например, предположим, что имеющийся опыт указывает, что 82% состояний, относящихся к категории эндшпилей с двумя пешками против одной пешки, приводят к победе (полезность +1); 2% — к проигрышу (0) и 16% — к ничьей (1/2). Тогда разумная оценка для состояний этой категории будет иметь ►**ожидаемое значение**: (0,82 × +1) +

$(0{,}02 \times 0) + (0{,}16 \times 1/2) = 0{,}90$. В принципе, ожидаемое значение можно определить для каждой категории состояний, получив в итоге функцию оценки, применимую для любого состояния.

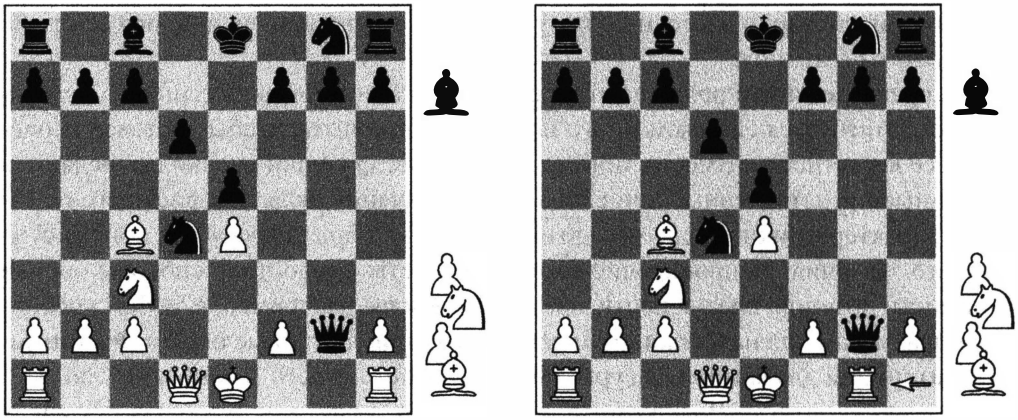
На практике такого рода анализ требует слишком много категорий, а следовательно, потребуется слишком много практического опыта, чтобы достоверно оценить все вероятности. Вместо этого в большинстве функций оценки вычисляются отдельные представленные в числовом виде значения вклада, зависящего от каждой характеристики, после чего эти значения *комбинируются* для поиска суммарного значения. За прошедшие века шахматисты выработали способы оценки стоимости позиции, следуя почти той же самой идее. Например, в любом учебнике по шахматам для начинающих можно найти приближенные оценки ►**стоимости материала** для каждой фигуры: пешка имеет стоимость 1, конь или слон — 3, ладья — 5, а ферзь — 9. Другие характеристики, такие как "хорошая пешечная структура" и "безопасность короля", могут оцениваться как равные, скажем, половине стоимости пешки. Чтобы получить оценку позиции, стоимости таких характеристик просто складываются.

В математике функция оценки такого типа называется ►**взвешенной линейной функцией**, поскольку она может быть представлена в следующем виде:

$$\text{EVAL}(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s) = \sum_{i=1}^{n} w_i f_i(s),$$

где каждая функция f_i оценивает некоторую характеристику позиции (такую, как количество белых слонов), а каждый коэффициент w_i представляет собой вес (указывает, насколько важна эта характеристика). Эти веса должны быть нормализованы таким образом, чтобы сумма всегда находилась в диапазоне от проигрыша (0) до победы (+1). Надежное преимущество, эквивалентное стоимости пешки, дает существенную вероятность выигрыша, а надежное преимущество, эквивалентное трем пешкам, почти определенно приводит к победе, как показано на рис. 5.8, *а*. Выше было сказано, что функция оценки должна сильно коррелировать с реальными шансами на победу, но эта корреляция вовсе не должна быть линейной: если состояние s имеет в два раза больше шансов на победу, чем состояние s', то совсем не требуется, чтобы значение $\text{EVAL}(s)$ было точно в два раза больше значения $\text{EVAL}(s')$; вполне достаточно, чтобы $\text{EVAL}(s) > \text{EVAL}(s')$.

На первый взгляд, метод вычисления суммы стоимостей характеристик может показаться приемлемым, но в действительности он основан на очень радикальном допущении, что вклад каждой характеристики *не зависит* от стоимости других характеристик. По этой причине в современных программах игры в шахматы и другие игры используются также *нелинейные* комбинации характеристик. Например, пара слонов может стоить больше удвоенной стоимости одного слона, а слон в конце игры стоит больше, чем в ее начале — когда характеристика *номер хода* достаточно велика или значение характеристики *количество оставшихся фигур* уже невелико.



а) Ход белых б) Ход белых

Рис. 5.8. Две шахматные позиции, различающиеся только позицией одной белой ладьи внизу справа. **а)** Черные имеют преимущество в одного коня и две пешки и должны выиграть партию. **б)** Белые берут ферзя и получают преимущество, которого должно быть достаточно для победы

А откуда появились характеристики и их веса? Они не являются частью правил игры в шахматы и были выработаны в течение столетий на основе опыта игры людей в шахматы. В играх, для которых опыт такого рода недоступен, веса в функции оценки могут быть оценены с помощью методов машинного обучения, обсуждаемых в главе 22. Применение этих методов к шахматам подтвердило, что слон действительно стоит примерно трех пешек, и, похоже, многовековой человеческий опыт может быть воспроизведен всего за несколько часов машинного обучения.

5.3.2. Останов поиска

Следующий этап состоит в модификации алгоритма ALPHA-BETA-SEARCH таким образом, чтобы он вызывал эвристическую функцию EVAL, когда возникает подходящая ситуация, чтобы остановить поиск. С точки зрения реализации потребуется лишь заменить две строки в алгоритме на рис. 5.7, в которых упоминается функция IS-TERMINAL, следующей строкой.

if *game*.IS-CUTOFF(*state*, *depth*) **then return** *game*.EVAL(*state*, *player*), *null*

Также необходимо предусмотреть выполнение определенных регистрирующих операций, чтобы текущее значение глубины *depth* наращивалось при каждом рекурсивном вызове. Наиболее прямолинейный подход к управлению объемом поисковых операций состоит в задании фиксированной глубины поиска, когда функция IS-CUTOFF(*state*, *depth*) возвращает значение *true* для всех значений *depth*, превы-

шающих некоторую заданную величину d (а также для всех терминальных состояний). Глубина d выбирается таким образом, чтобы ход выбирался в пределах установленного интервала времени. Более надежный подход состоит в использовании метода итеративного углубления (см. главу 3). По истечении отведенного на расчеты времени программа возвращает ход, выбранный по итогам наиболее глубокого завершенного поиска. Возможны и дополнительные преимущества: если в каждом раунде итеративного углубления сохранять данные в таблице перестановок, последующие раунды будут выполняться быстрее за счет использования этих оценок для упорядочивания ходов.

Однако подобные простые подходы могут приводить к ошибкам из-за приблизительного характера функции оценки. Еще раз рассмотрим простую функцию оценки для шахмат, основанную на учете преимущества в материале. Предположим, что программа, выполняя поиск с заданным пределом глубины, достигла позиции, приведенной на рис. 5.8, *б*, в которой черные имеют перевес в одного коня и две пешки. Она расценила бы это как достаточное основание для определения эвристической оценки данного состояния, объявив тем самым, что это состояние, по всей вероятности, приведет к победе черных. Но на следующем ходу белые берут ферзя черных без компенсации, и в действительности данная позиция является выигрышной для белых, но об этом можно было бы узнать, только заглянув вперед еще на один полуход.

Функция оценки должна применяться только к таким позициям, которые можно считать ►**спокойными**, т.е. такими, в которых нет возможных ходов, ведущих к резким изменениям ситуации (таких, как взятие ферзя). В неспокойных позициях функция IS-CUTOFF должна возвращать значение *false* и поиск должен продолжаться до достижения спокойной позиции. Этот дополнительный ►**поиск спокойных позиций** иногда ограничивается анализом лишь определенных типов ходов, например взятие фигур, быстро устраняющих состояние неопределенности в данной позиции.

Устранение ►**эффекта горизонта** является более сложной задачей. Подобный эффект возникает, когда программа сталкивается с ходом противника, причиняющим серьезный ущерб и в конечном счете неизбежным, но который можно временно отложить, используя тактику задержки. Рассмотрим шахматную позицию, приведенную на рис. 5.9. Очевидно, что невозможно избежать потери черного слона. Например, ладья белых может взять его, переместившись на h1, затем — на a1, а затем — на a2 (взятие находится на расстоянии 6 полуходов).

Но у черных есть последовательность ходов, при которой взятие слона выталкивается "за горизонт" выполняемого анализа. Предположим, что черные ведут поиск на глубину 8 полуходов. Большинство ходов черных ведет к возможному взятию их слона, а значит, они будут отмечены как "плохие" ходы. Но в ходе анализа черные также будут проверять и последовательность ходов, начинающуюся с объявления шаха пешкой, склоняя белого короля к тому, чтобы взять ее. Затем черные могут повторить тот же сценарий со второй пешкой, и вся эта процедура

потребует такого количества ходов, что ситуацию взятия слона уже нельзя будет обнаружить в оставшейся части их поиска. В результате черные приходят к выводу, что в этой линии игры удается спасти слона, жертвуя две пешки, тогда как на самом деле все, что будет достигнуто, — это бесполезная потеря двух пешек и выталкивание по-прежнему неизбежной потери слона за горизонт, в пределах которого черные могут видеть.

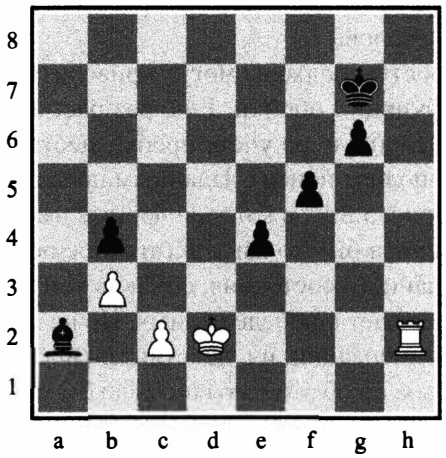


Рис. 5.9. Пример эффекта горизонта. При ходе черных их слон, безусловно, обречен. Но черные могут задержать это событие, сделав ход пешкой и объявив шах белому королю, что должно побудить короля взять эту пешку. Это выталкивает неизбежную потерю слона за горизонт, и потому жертва пешки рассматривается поисковым алгоритмом как хороший ход, а не как плохой

Одна из возможных стратегий ослабления эффекта горизонта заключается в разрешении ▶ **выборочных продлений** — ходов, которые "очевидно лучше", чем все остальные ходы в данной позиции, даже тогда, когда в нормальной ситуации поиск в этой точке должен быть уже остановлен. В нашем примере ранее проведенный поиск уже покажет, что три хода белой ладьи (с h2 на h1, потом — с h1 на a1 и затем — взятие слона с a1 на a2) являются, каждый в свою очередь, очевидно лучшими ходами. Поэтому, даже если последовательность ходов пешками подталкивает нас к горизонту, этим очевидно лучшим ходам будет предоставлен шанс продолжить поиск. В результате дерево поиска станет более глубоким, но, поскольку обычно имеется лишь небольшое количество выборочных продлений, эта стратегия не приводит к добавлению к дереву множества общих узлов и уже доказала свою эффективность на практике.

5.3.3. Прямое отсечение

Если метод альфа-бета-отсечения предполагает отбрасывание тех ветвей дерева поиска, которые не могут повлиять на окончательную оценку, то метод ▶ **прямого отсечения** предусматривает отбрасывание ветвей, которые *кажутся* плохими, но, вероятно, могут оказаться и хорошими. Следовательно, такая стратегия экономит время вычислений, рискуя допустить ошибку. Согласно классификации Шеннона, эта стратегия относится к типу B. Очевидно, что большинство людей, играющих в шахматы, рассматривают лишь несколько ходов из каждой позиции (по крайней мере, сознательно).

Одним из подходов в методике прямого отсечения является **поиск по лучу** (см. раздел 4.1.3): для каждого полухода из всех возможных ходов анализируется только узкий "луч" из n лучших ходов (в соответствии с функцией оценки). К сожалению, этот подход является довольно опасным, поскольку нет никакой гарантии, что лучший ход не будет отброшен еще до того, как появится возможность его обнаружить.

Алгоритм PROBCUT, или "вероятностное отсечение" (Буро, [395], 1995), является версией альфа-бета-поиска с прямым отсечением. В нем для уменьшения шанса отсечения лучшего хода используется статистическая информация, полученная из предыдущего опыта. Алгоритм альфа-бета-отсечения отбрасывает любой узел, который *доказуемо* не попадает в текущий диапазон (α, β), а алгоритм PROBCUT отсекает узлы, которые, *вероятно*, находятся вне этого диапазона. Эта вероятность вычисляется путем выполнения неглубокого поиска для определения резервируемого значения v узла, а затем прошлый опыт используется для оценки, насколько вероятно, что резервируемое значение v на глубине d в дереве может оказаться вне диапазона (α, β). Буро применил этот метод в своей программе игры в реверси, LOGISTELLO, и установил, что версия его программы с использованием алгоритма PROBCUT превосходила обычную версию в 64% случаев, даже когда обычной версии предоставлялось вдвое больше времени.

Другой метод, ▶ **сокращение последних ходов**, строится на предположении, что упорядочение ходов было выполнено корректно, и поэтому те из них, которые оказались в конце списка возможных ходов, с меньшей вероятностью окажутся хорошими. Но вместо того, чтобы полностью их обрезать, алгоритм просто уменьшает глубину, до которой эти ходы анализируются, чем и достигается экономия времени. Если сокращенный поиск возвращает значение выше текущего значения α, можно повторно запустить его, но уже на полную глубину.

В результате совместного использования всех описанных выше методов можно создать программу, которая будет неплохо играть в шахматы (или другие игры). Предположим, что реализована функция оценки для шахмат и предусмотрена разумная проверка останова в сочетании с поиском спокойной позиции. Кроме того, предположим, что, затратив месяцы на скрупулезную разработку программ, манипулирующих данными на уровне битов, мы получили возможность формировать

и оценивать примерно миллион узлов в секунду на новейшем ПК. Коэффициент ветвления для шахмат составляет в среднем около 35, а 35^5 равно приблизительно 50 миллионам, поэтому при использовании минимаксного поиска мы получим возможность заглядывать вперед примерно лишь на пять полуходов в минуту, и правила проведения соревнований предоставляли бы нам недостаточно времени, чтобы вести поиск на глубину шести полуходов. У такой программы, хотя и не совсем некомпетентной, легко сможет выиграть любой любитель шахмат среднего уровня, который часто способен планировать на шесть или даже восемь полуходов вперед.

Альфа-бета-поиск и достаточно большая таблица перестановок позволяют достичь глубины приблизительно в 14 полуходов, что приводит к усилению игры до уровня мастера. Если заменить обычный ПК рабочей станцией с 8-ю графическими процессорами, то это позволит программе анализировать более миллиарда узлов в секунду, но для достижения уровня гроссмейстера дополнительно потребуется тщательно настроенная функция оценки и большая база данных с записями оптимальных ходов в эндшпиле. Лучшие шахматные программы, такие как STOCKFISH, имеют все это и часто достигают глубины 30 и даже более полуходов в дереве поиска, благодаря чему намного превосходят возможности любого игрока-человека.

5.3.4. Поиск или просмотр таблицы

Для программы игры в шахматы кажется как-то чрезмерным начинать игру с анализа дерева из миллиарда игровых состояний только для того, чтобы прийти к решению сделать ход пешкой e2 на e4 (самый популярный первый ход). Книги, описывающие хорошую игру в шахматы в начале партии и эндшпиле, доступны уже более ста лет (Таттерсалл [2184], 1911). Поэтому не удивительно, что многие игровые программы в начале и конце игры вместо поиска по дереву используют *просмотр таблицы*.

В начале игры компьютерные программы в основном полагаются на опыт человека. Самые лучшие советы от людей-экспертов в отношении того, как начинать каждую игру, могут быть взяты из книг и занесены в таблицы для использования компьютерами. Кроме того, компьютеры могут получать статистическую информацию из базы данных ранее сыгранных игр с целью установить, какая последовательность ходов в начале игры чаще всего приводит к выигрышу. Для первых нескольких ходов возможностей немного, и большинство позиций будут присутствовать в таблице. Обычно после примерно 10 или 15 ходов игра приходит в состояние, которое встречается уже нечасто, и программе необходимо перейти от просмотра поисковой таблицы к поиску по дереву игры.

Ближе к концу игры возможных позиций вновь становится меньше, а следовательно, просматривать таблицу вновь становится проще. Но здесь в качестве экспертов выступают уже компьютеры: возможности компьютерного анализа в

эндшпиле выходят далеко за пределы человеческих возможностей. Игроки-новички могут выиграть эндшпиль с королем и ладьей против одного короля, просто следуя нескольким правилам. Другие окончания, такие как король, слон и конь против одного короля, будут достаточно сложными и для мастера, для них не существует достаточно краткого описания оптимальной стратегии.

С другой стороны, компьютерная программа может полностью *решить* эндшпиль, рассчитав **стратегию**, которая будет представлять собой отображение каждого возможного состояния в лучший ход в этом состоянии. В результате компьютер сможет демонстрировать превосходную игру, просто отыскивая правильный ход в этой таблице. Такая таблица создается с помощью ▶**обратного минимаксного поиска**: сначала рассматриваются все способы размещения имеющихся фигур на доске (например, белого короля, слона и коня и черного короля). Некоторые из таких возможных позиций окажутся позициями выигрыша белых, — отметим их как таковые. Затем меняем правила шахмат на обратные, чтобы можно было делать обратные, а не обычные ходы. Любой ход белых, который, независимо от любого возможного ответного хода черных, приводит к позиции, отмеченной как выигрышная, также должен приводить к выигрышу. Подобный поиск продолжается, пока все возможные позиции не будут определены как выигрышные, проигрышные или ничейные, в результате чего будет создана безошибочная таблица поиска для всех эндшпилей с этими фигурами. Такие расчеты были проведены не только для окончаний с королем, конем и слоном против одного короля, но и для всех эндшпилей с семью или менее фигурами. Созданные в результате таблицы содержат 400 триллионов позиций. Отметим, что анализ эндшпиля с 8-ю фигурами потребует создания таблицы с 40 квадриллионами позиций.

5.4. Поиск по дереву методом Монте-Карло

Игра го хорошо иллюстрирует два основных недостатка эвристического альфа-бета-поиска по дереву. Первый состоит в том, что для го коэффициент ветвления в начале игры достигает 361, а это означает, что возможности альфа-бета-поиска будут ограничены лишь 4 или 5 полуходами. Второй состоит в том, что для игры го очень трудно отыскать хорошую функцию оценки, поскольку стоимость материала в этой игре не является сильным показателем, а большинство позиций находятся в движении до самого конца игры. Из-за этих двух проблем современные разработчики программ игры в го отказались от альфа-бета-поиска и вместо него используют стратегию, получившую название ▶**поиск по дереву методом Монте-Карло** (*Monte Carlo tree search — MCTS*).[3]

Базовая стратегия MCTS не использует эвристическую функцию оценки. Вместо этого значение состояния оценивается как средняя полезность по ряду

[3] Алгоритмы Монте-Карло — это рандомизированные алгоритмы, получившие свое название в честь казино "Монте-Карло" в Монако.

► **симуляций** завершенных игр, начиная с данного состояния. При симуляции (ее также называют ► **прогоном** или ► **прокачкой**) по очереди выбираются ходы сначала для одного игрока, затем — для другого, пока не будет достигнуто терминальное состояние. На этом этапе по правилам игры (а не по подверженным ошибкам эвристическим оценкам) определяется, кто выиграл или проиграл и с каким счетом. Для игр, в которых единственным результатом является выигрыш или проигрыш, "средняя полезность" совпадает с "процентом побед".

Как же можно выбрать, какой ход сделать при прогоне? Если просто выбирать очередной ход случайным образом, то после нескольких симуляций мы получим ответ на вопрос "Какой ход является лучшим, если оба игрока играют случайным образом?" Для некоторых простых игр это одновременно будет ответом и на вопрос "Какой ход является лучшим, если оба игрока играют правильно?", но для большинства игр это будет не так. Чтобы получить полезную информацию от прогонов, необходимо иметь ► **стратегию прогонов**, которая смещает выбор ходов в сторону хороших. В игре го и других играх стратегия прогонов была успешно изучена при игре компьютеров с самим собой с использованием нейронных сетей. Иногда используются специфические для игры эвристики, такие как "рассмотреть ходы взятия" в шахматах или "захватить угловой квадрат" в реверси.

Имея стратегию прогонов, далее нужно принять два решения: с каких позиций игра будет начинаться и сколько прогонов выделяется для каждой позиции? Самый простой ответ, называемый ► **чистым поиском по методу Монте-Карло**, состоит в том, чтобы выполнить N симуляций, начиная с текущего состояния игры, и проследить, какой из возможных ходов с текущей позиции будет иметь самый высокий процент побед.

Для некоторых стохастических игр этот алгоритм сходится к оптимальной игре по мере увеличения значения N, но для большинства игр этого будет недостаточно — дополнительно потребуется ► **стратегия выбора**, выборочно фокусирующая вычислительные ресурсы на важных частях дерева игры. Ее назначение — сбалансировать два фактора: ► **исследование** состояний, для которых было выполнено мало прогонов, и ► **использование** тех состояний, которые хорошо себя показали в прошлых прогонах, чтобы получить более точную оценку их стоимости. (Подробнее о компромиссе исследования/использования читайте в разделе 17.3.) В поиске по дереву методом Монте-Карло это делается посредством сохранения дерева поиска и его расширения после каждой итерации, включающей следующие четыре этапа, как показано на рис. 5.10.

- **Выбор**. Начиная с корня дерева поиска, выбираем ход (руководствуясь стратегией выбора), ведущий к узлу-преемнику, и повторяем этот процесс, перемещаясь вниз по дереву до листовых узлов. На рис. 5.10, *а* показано дерево поиска, корень которого представляет состояние, в которое белые только что переместились и в котором (на данный момент) белые выиграли 37 из 100 симулированных игр. Утолщенная стрелка показывает выбор очередного хода

черными, ведущий к узлу, из которого черные уже выиграли 60 из 79 симуляций. Это наилучший процент побед для трех доступных ходов, так что данный выбор является примером *использования*. Однако также было бы разумно выбрать и узел 2/11 — в целях его дополнительного *исследования*, поскольку при только 11 выполненных с его использованием прогонах этот узел все еще характеризуется значительной неопределенностью в оценке, которая в конечном итоге может оказаться значительно лучше, если собрать больше информации. Далее этап выбора подобным образом продолжается вплоть до достижения листового узла с оценкой 27/35.

- **Расширение**. Дерево поиска расширяется — для выбранного узла генерируется новый дочерний узел с нулевыми результатами. На рис. 5.10, *б* этот новый узел помечен как 0/0. (В некоторых версиях на этом этапе генерируется более одного дочернего узла.)
- **Симуляция.** Из нового дочернего узла проводится симуляция дальнейшей игры вплоть до чьей-либо победы, при этом ходы обоих игроков выбираются в соответствии со стратегией прогонов. Эти ходы *не записываются* в дерево поиска. На рисунке симуляция приводит к победе черных.
- **Обратное распространение.** Теперь полученный результат симуляции используется для обновления всех пройденных на этапе выбора узлов поискового дерева вплоть до его корня. Так как прогон выиграли черные, в узлах черных увеличиваются на единицу как счетчик побед, так и счетчик прогонов, так что значение 27/35 заменяется значением 28/36, а значение 60/79 — значением 61/80. Поскольку белые проиграли, в узлах белых будет увеличен на единицу только счетчик прогонов, поэтому значение 16/53 заменяется значением 16/54, а значение 37/100 в корневом узле заменяется значением 37/101.

Выполнение цикла из этих четырех этапов повторяется либо заданное количество раз, либо до истечения отведенного для этого времени, а затем возвращается ход с наибольшим количеством прогонов.

Одна очень эффективная стратегия выбора называется "верхние границы достоверности, применяемые к деревьям" или ▶**UCT** (*Upper Confidence bounds applied to Trees*). Она предусматривает ранжирование каждого возможного хода на основе формулы "верхней границы уверенности 1, применяемой к деревьям" и обычно называемой ▶**UCB1** (подробнее читайте в разделе 17.3.3). Для некоторого узла n эта формула выглядит следующим образом.

$$UCB1(n) = \frac{U(n)}{N(n)} + C \times \sqrt{\frac{\log N(\text{PARENT}(n))}{N(n)}}$$

Здесь $U(n)$ является общей полезностью всех прогонов, прошедших через узел n, $N(n)$ является количеством этих прогонов, а PARENT(n) определяет родительский

узел для узла *n* в дереве. Таким образом, первый компонент $U(n)/N(n)$ представляет в формуле *использование* и является средней полезностью узла *n*. Выражение под квадратным корнем представляет *исследование*: в его знаменателе стоит $N(n)$, а это означает, что оно будет иметь большее значение для узлов, которые исследовались лишь несколько раз. В числителе этого выражения стоит логарифм от общего количества прогонов, выполненных для узла, являющегося родительским для узла *n*. Это означает, что если процент выбора узла *n* при прогонах отличен от нуля, то весь второй компонент формулы, представляющий исследование, будет стремиться к нулю по мере увеличения этого процента, и в конечном итоге в прогонах предпочтение будет отдаваться узлам с наибольшей средней полезностью.

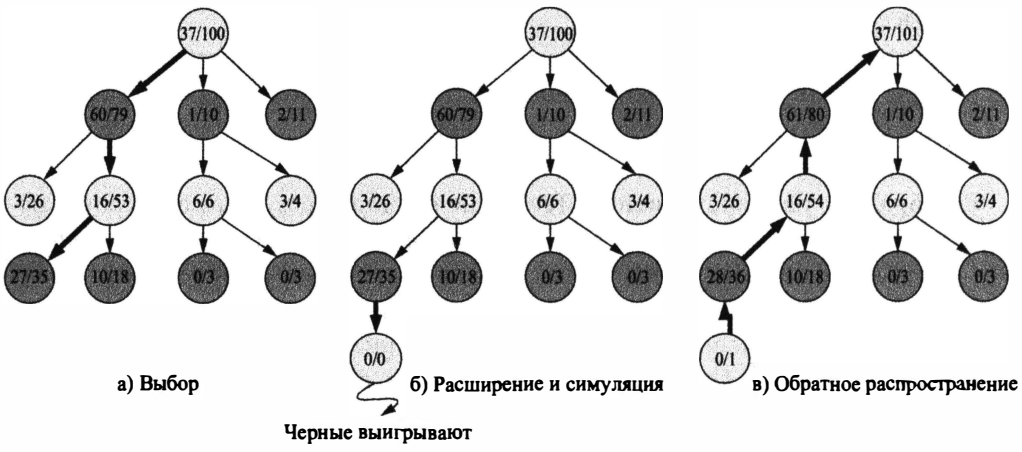


Рис. 5.10. Одна итерация процесса выбора хода по алгоритму поиска по дереву методом Монте-Карло (MCTS) с использованием стратегии верхних границ достоверности, применяемых к деревьям (UCT), выполняемая после того, как 100 подобных итераций уже были выполнены. **а)** В направлении вниз по дереву выбирается последовательность ходов вплоть до листового узла с отметкой 27/35 (27 побед черных из 35 прогонов). **б)** Выбранный узел расширяется, и выполняется симуляция (прогон), заканчивающаяся победой черных. **в)** Результат симуляции распространяется в обратном направлении вверх по дереву

Коэффициент C является константой, определяющей баланс между двумя компонентами формулы, представляющими использование и исследование. Теоретически он должен иметь значение, равное $\sqrt{2}$, но на практике создатели игровых программ проверяют несколько различных значений C и выбирают то, которое дает наилучшие результаты. (В некоторых программах используются слегка различающиеся формулы, например в программе AlphaZero добавлен компонент для представления вероятности хода, который вычисляется нейронной сетью, обученной на результатах прошлых игр программы с самой собой.) Для дерева на

рис. 5.10 при $C = 1{,}4$ наивысшую оценку UCB1 будет иметь узел 60/79, но при $C = 1{,}5$ это будет уже узел 2/11.

На рис. 5.11 представлен полный алгоритм UCT MCTS. Когда итерации заканчиваются, возвращается ход с наибольшим количеством прогонов. Можно подумать, что было бы лучше вернуть узел с самой высокой средней полезностью, но сама идея метода состоит в том, что узел с 65/100 выигрышами будет лучше, чем узел с 2/3 выигрышами, поскольку для последнего имеет место значительная неопределенность в возможных результатах. В любом случае формула UCB1 гарантирует, что узел с наибольшим количеством прогонов почти всегда будет узлом и с самым высоким процентом выигрышей, поскольку процесс выбора способствует росту процента выигрышей по мере увеличения количества прогонов.

```
function MONTE-CARLO-TREE-SEARCH(state) returns действие action
  tree ← NODE(state)
  while IS-TIME-REMAINING() do
    leaf ← SELECT(tree)
    child ← EXPAND(leaf)
    result ← SIMULATE(child)
    BACK-PROPAGATE(result, child)
  return ход из ACTIONS(state), узел которого имеет наибольшее число прогонов
```

Рис. 5.11. Алгоритм поиска по дереву методом Монте-Карло. Дерево игры, *tree*, инициализируется, а затем цикл SELECT/EXPAND/SIMULATE/BACK-PROPAGATE повторяется до тех пор, пока не закончится отведенное на его выполнение время, после чего возвращается ход, ведущий к узлу с наибольшим количеством прогонов

С увеличением глубины дерева игры время просчета прогонов увеличивается линейно, а не экспоненциально, поскольку в каждой точке выбора реализуется только один ход. А значит, у нас имеется достаточно времени для выполнения множества прогонов. Например, рассмотрим игру с коэффициентом ветвления 32 и средней продолжительностью партии в 100 полуходов. При наличии вычислительной мощности, достаточной для проведения анализа миллиарда игровых состояний до выполнения хода, минимаксный алгоритм позволит полностью исследовать до 6 полуходов, алгоритм альфа-бета-отсечения с правильным упорядочиванием ходов — до 12 полуходов, а алгоритм поиска по дереву по методу Монте-Карло сможет выполнить до 10 миллионов прогонов. Какой подход лучше? Все зависит от точности эвристической функции в сравнении с эффективностью стратегий выбора и прогона.

Накопленный практический опыт показывает, что поиск по методу Монте-Карло имеет преимущество над альфа-бета-поиском для таких игр, как го, у которых коэффициент ветвления очень высок (а значит, альфа-бета-поиск не сможет вести анализ достаточно глубоко), или когда имеются трудности в отыскании

эффективной функции оценки. Суть альфа-бета-поиска — в выборе пути к узлу, имеющему самое высокое значение функции оценки с учетом того, что противник будет пытаться минимизировать данное значение. А это означает, что если функция оценки является неточной, то и альфа-бета будет работать неточно. Ошибочная оценка для одного из узлов может привести альфа-бета-поиск к ошибочному выбору (или к отказу от выбора) пути к этому узлу. Однако поиск по методу Монте-Карло полагается на общий результат множества выполненных прогонов, а значит, он гораздо менее уязвим для одиночных ошибок. Можно объединить преимущества метода MCTS и функции оценки, выполнив прогон для определенного количества узлов с последующим отсечением и использованием функции оценки.

Также возможно комбинировать определенные аспекты альфа-бета-поиска и поиска по методу Монте-Карло. Например, в играх, которые могут длиться много ходов, можно использовать ▶ **раннее завершение прогона**, когда выполнение прогона принудительно завершается после выполнения в нем слишком большого количества ходов, а затем либо он оценивается с использованием эвристической функции оценки, либо просто объявляется о проигрыше.

Поиск по методу Монте-Карло может быть применен к совершенно новым играм, для которых накоплено еще недостаточно опыта для нахождения эффективных функций оценки. Если правила игры известны, выполнение поиска по методу Монте-Карло не требует никакой дополнительной информации. В стратегиях выбора и прогона могут эффективно использоваться экспертные знания, собранные вручную, если они доступны, но хорошие стратегии можно также найти, используя обучение нейронных сетей, играющих сами с собой.

Поиск по методу Монте-Карло имеет один недостаток: если есть вероятность, что единственный ход может радикально изменить всю игру, то по самой сути своего стохастического характера поиск по методу Монте-Карло может просто случайно не суметь его обнаружить. Другими словами, использование в поиске по методу Монте-Карло отсечки типа B может привести к тому, что жизненно важная линия игры может оказаться просто неисследованной. Поиск по методу Монте-Карло также испытывает затруднения, когда в игре имеются состояния, являющиеся "очевидно" выигрышными для одной или другой стороны (в соответствии с имеющимися у нас знаниями и результатами функции оценки), но требующие выполнения еще множества ходов при прогоне, пока победитель будет установлен. Долгое время считали, что альфа-бета-поиск лучше подходит для таких игр, как шахматы, — с низким коэффициентом ветвления и хорошими функциями оценки. Однако в последнее время подход с использованием метода Монте-Карло продемонстрировал значительный успех и в шахматах, и в других играх.

Общая идея симуляции будущих ходов с наблюдением результатов и использованием их для определения, какие ходы являются хорошими, — это один из видов **обучения с подкреплением**, которое подробно обсуждается в главе 22.

5.5. Игры с элементами случайности

►**Стохастические игры** (или **игры с элементами случайности**) подводят нас немного ближе к непредсказуемости реальной жизни за счет включения в игровой процесс некоторых случайных элементов, таких как бросание костей. Нарды — это типичная стохастическая игра, сочетающая в себе удачу и умение. В позиции игры в нарды, показанной на рис. 5.12, белые бросили кости с результатом 6–5 и теперь имеют четыре возможных варианта хода (в каждом из них одна фишка перемещается вперед — против часовой стрелки — на 5 позиций, а затем еще одна вперед на 6 позиций).

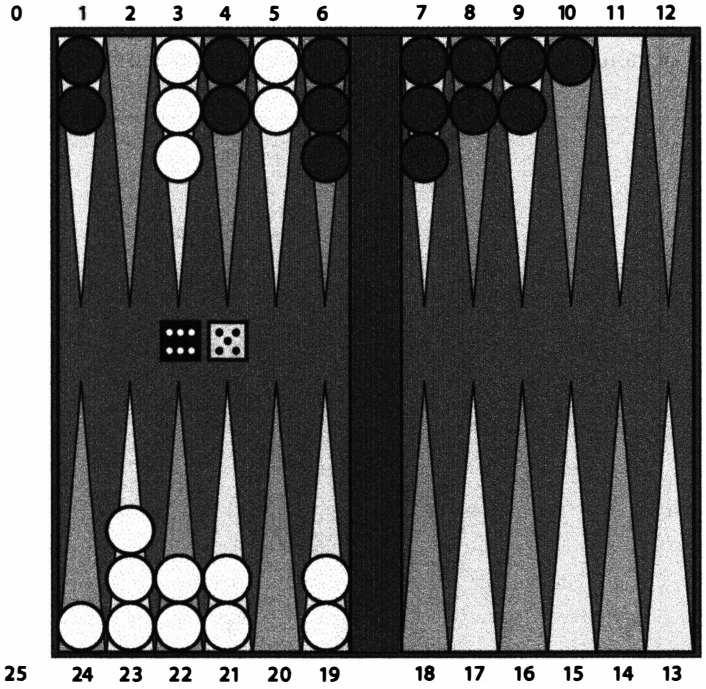


Рис. 5.12. Типичная позиция игры в нарды. Цель игры состоит в том, чтобы перевести все свои фишки в определенный угол доски, а затем снять их. Белые ходят против часовой стрелки от поля 1 к полю 25, а черные — по часовой стрелке от поля 24 к полю 0. Фишка может переходить на любое поле, если на нем нет несколько фишек противника; если же на этом поле есть только одна фишка противника, она попадает в плен и должна начать свое движение с самого начала. В показанной здесь позиции белые сделали бросок с результатом 6–5 и теперь должны сделать выбор среди четырех допустимых ходов: (5–11, 5–10), (5–11, 19–24), (5–10, 10–16) и (5–11, 11–16), где запись (5–11, 11–16) означает, что одна фишка перемещается с позиции 5 на позицию 11, а затем эта же фишка перемещается с позиции 11 на позицию 16

Хотя белым известно, каковы их допустимые ходы, они не знают, какие очки принесет бросание костей черным, и поэтому не могут определить, какими будут допустимые ходы черных. Это означает, что белые не имеют возможности сформировать стандартное дерево игры такого типа, которое встретилось нам в шахматах или крестиках-ноликах. Дерево игры в нарды, показанное на рис. 5.13, кроме узлов MAX и MIN, должно включать ▶ **узлы жеребьевки**, — такие узлы на рисунке представлены кружками (CHANCE). Ветви, ведущие от каждого узла жеребьевки, обозначают возможные результаты метания жребия, поэтому каждая из них отмечена надписью с указанием количества очков и вероятности, с которой могут быть получены эти очки. Существует 36 возможных сочетаний очков на двух кубиках, и все они являются равновероятными; но поскольку такие сочетания очков, как 6–5 и 5–6, для игры являются одинаковыми, имеется только 21 различное сочетание очков. Поэтому вероятность появления шести дублей (от 1–1 до 6–6) равна 1/36, а каждое из 15 остальных различных сочетаний очков имеет вероятность 1/18.

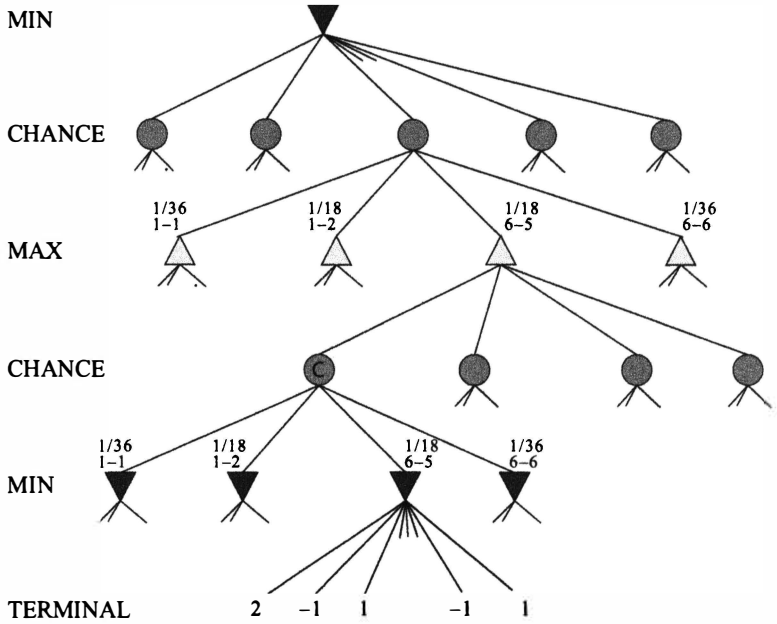


Рис. 5.13. Схематическое изображение дерева игры для одной из позиций игры в нарды

Следующий этап состоит в том, чтобы понять, как следует принимать правильные решения. Безусловно, и в этом случае требуется найти такой ход, который ведет к наилучшей позиции. Однако результирующие позиции не имеют определенных минимаксных значений. Вместо этого существует возможность вычислить

только ▶**ожидаемое значение**, в котором ожидаемый результат устанавливается с учетом всех возможных выпадений жребия, которые могут произойти.

Это приводит к обобщению минимаксного значения для детерминированных игр до ▶**ожидаемого минимаксного значения** (*expectiminimax value*) для игр с узлами жеребьевки. Терминальные узлы и узлы MAX и MIN работают точно так же, как и раньше (с оговоркой, что допустимые ходы для MAX и MIN будут зависеть от исхода броска костей в предыдущем узле жеребьевки). Для узлов жеребьевки ожидаемое значение вычисляется как сумма значений, полученных в результате всех возможных выпадений жребия, взвешенных по вероятности каждого случайного действия.

$$\text{EXPECTIMINIMAX}(s) = \begin{cases} \text{UTILITY}(s, \text{MAX}) & \text{если } \text{IS-TERMINAL}(s) \\ \max_a \text{EXPECTIMINIMAX}(\text{RESULT}(s, a)) & \text{если } \text{TO-MOVE}(s) = \text{MAX} \\ \min_a \text{EXPECTIMINIMAX}(\text{RESULT}(s, a)) & \text{если } \text{TO-MOVE}(s) = \text{MIN} \\ \sum_r P(r)\text{EXPECTIMINIMAX}(\text{RESULT}(s, r)) & \text{если } \text{TO-MOVE}(s) = \text{CHANCE} \end{cases}$$

Здесь r представляет возможный бросок костей (или другое случайное событие), а $\text{RESULT}(s, r)$ является тем же состоянием s, но с дополнительным обстоятельством, что результатом броска костей является r.

5.5.1. Функции оценки в играх с узлами жеребьевки

По аналогии с минимаксными значениями очевидный подход к использованию ожидаемых минимаксных значений состоит в том, чтобы останавливать поиск в некоторой точке и применять функцию оценки к каждому листу. На первый взгляд может показаться, что функции оценки для таких игр, как нарды, должны быть полностью подобны функциям оценки для шахмат, — они просто должны присваивать более высокие значения полезности лучшим позициям. Но в действительности наличие узлов жеребьевки означает, что потребуется более тщательный анализ смысла таких оценочных значений.

На рис. 5.14 показано, что происходит в стохастических играх: если функция оценки присваивает листовым узлам значения [1, 2, 3, 4], то наилучшим является ход a_1, а если она присваивает значения [1, 20, 30, 400], то наилучшим становится ход a_2. В результате программа ведет себя совершенно по-разному, если вносятся изменения в шкалу некоторых оценочных значений, даже если порядок предпочтения узлов остается тем же.

Как оказалось, чтобы избежать этой проблемы, функция оценки должна возвращать значения, представляющие собой положительное линейное преобразование **вероятности** выигрыша (или ожидаемой полезности для игр, допускающих иные результаты, помимо победы/проигрыша). Подобная связь с вероятностью является важным и общим свойством ситуаций, предполагающих наличие неопределенности, и оно детально обсуждается в главе 16.

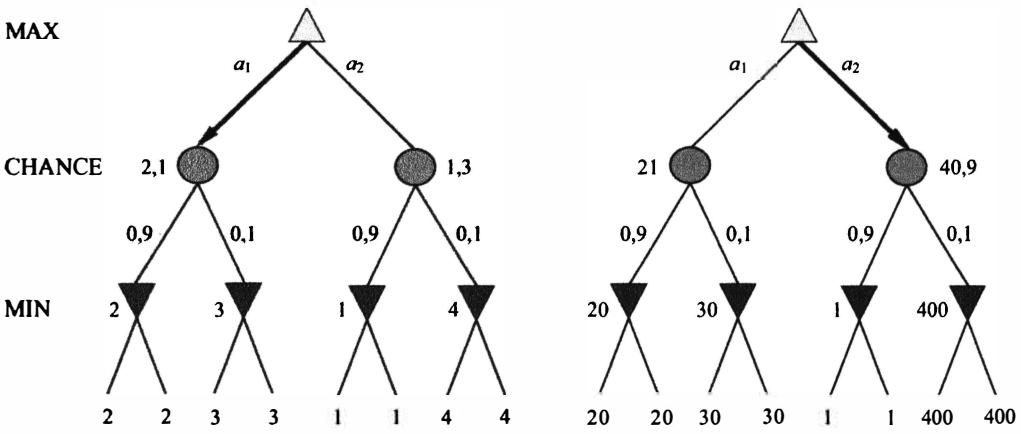


Рис. 5.14. При трансформации оценочных значений листовых узлов, даже с сохранением их упорядоченности, наилучший ход может измениться

Если бы программе были заранее известны все результаты жеребьевки, которые должны произойти в течение оставшейся части игры, то поиск решения в стохастической игре был бы полностью аналогичен поиску решения в игре без жеребьевки, который выполняется минимаксным алгоритмом поиска за время $O(b^m)$, где b является коэффициентом ветвления, а m — максимальной глубиной дерева игры. Но поскольку в алгоритме, использующем ожидаемые минимаксные значения, рассматриваются все возможные последовательности результатов жеребьевки, его выполнение потребует времени $O(b^m n^m)$, где n — количество различных вариантов результатов жеребьевки.

Даже если глубина поиска ограничена некоторой небольшой величиной d, из-за таких дополнительных затрат времени по сравнению с минимаксным алгоритмом в большинстве стохастических игр стремление заглянуть в дерево поиска на очень большую глубину будет неосуществимым. В нардах n равно 21, а b обычно принимается равным около 20, но в некоторых ситуациях этот коэффициент может достигать 4000, — при выбрасывании игроком серии дублей. По-видимому, все, на что в этой игре можно рассчитывать, — это заглянуть вперед на три полухода.

Другой вариант подхода к этой проблеме состоит в следующем: преимуществом альфа-бета-поиска является то, что в нем игнорируются те будущие направления развития игры, которые не должны быть реализованы, если в игре всегда выбирается наилучший ход. В результате появляется возможность сосредоточиться лишь на перспективных, многообещающих событиях. Но в играх, включающих бросание костей перед каждым ходом, *нет* перспективных последовательностей ходов, — даже наиболее многообещающий ход будет иметь место только в 2/36 случаях, поскольку для его выполнения необходимо, чтобы результаты предшествующего броска сделали данный ход допустимым. И это общая проблема,

возникающая в любой ситуации, допускающей наличие неопределенности: количество вариантов дальнейших действий возрастает в огромной степени, а построение подробных планов действий становится бессмысленным, поскольку мир, скорее всего, не станет нам подыгрывать.

Видимо, вам уже пришла в голову мысль, что к деревьям игр с узлами жеребьевки можно попробовать применить какой-то метод, подобный альфа-бета-отсечению. Как оказалась, такая возможность действительно существует. Анализ узлов MIN и MAX остается неизменным, но можно также реализовать отсечение узлов жеребьевки с использованием некоторой доли изобретательности. Рассмотрим узел жеребьевки *C* на рис. 5.13 и попробуем определить, что будет происходить с его значением по мере исследования и оценки его дочерних узлов. Можно ли найти верхний предел значения *C* до того, как будут проверены все его дочерние узлы? (Напомним, что именно это требуется в альфа-бета-поиске для того, чтобы можно было выполнить отсечение узла и его поддерева.)

На первый взгляд это может показаться невозможным, поскольку значение узла *C* представляет собой *среднее* значение его дочерних узлов, а для того, чтобы вычислить среднее значение нескольких чисел, необходимо знать все эти числа. Тем не менее, если установить пределы допустимых значений функции полезности, то можно будет прийти и к пределам для среднего значения, не зная всех числовых значений. Например, пусть все значения полезности находятся в пределах от –2 до +2, тогда значения листовых узлов являются ограниченными, а это, в свою очередь, позволяет установить верхний предел значения узла жеребьевки, не рассматривая все его дочерние узлы.

В играх, в которых коэффициент ветвления узлов жеребьевки очень высок — как в игре "Яцзы" (покер на кубиках), в которой при каждом ходе бросается пять кубиков, — можно попробовать проанализировать прямое отсечение, когда для анализа выбирается лишь небольшое количество возможных результатов жеребьевки. Или же можно вообще отказаться от использования функции оценки и обратиться к поиску по дереву методом Монте-Карло, при котором каждый прогон будет включать и случайные броски костей.

5.6. Частично наблюдаемые игры

Бобби Фишер заявил, что "шахматы — это война", тем не менее в шахматах отсутствует по крайней мере одна важная характеристика реальных войн, а именно — ▶**частичная наблюдаемость**. В "тумане войны" истинное местопребывание вражеских подразделений часто неизвестно, пока оно не будет установлено при непосредственном контакте. Как результат любые войны обязательно включают в себя действия разведчиков и шпионов, направленные на сбор информации, а также использование маскировки и ложных целей для введения врага в заблуждение.

Частично наблюдаемые игры разделяют все эти характеристики, а значит, качественно отличаются от тех игр, которые обсуждались в предыдущих разделах.

Такие видеоигры, как StarCraft, являются особенно сложными, поскольку их следует классифицировать как частично наблюдаемые, мультиагентные, недетерминированные, динамические с неизвестной средой.

В детерминированных частично наблюдаемых играх неопределенность относительно состояния игрового поля возникает всецело по причине отсутствия доступа к выбору, сделанному противником. Этот класс включает в себя многие детские игры, такие как "Морской бой" (корабли каждого игрока расположены в местах, скрытых от противника) или "Стратего" (места фигур известны, но типы фигур скрыты). В этом разделе речь пойдет об игре ▶"**Кригшпиль**", частично наблюдаемой версии шахмат, в которой фигуры игрока полностью невидимы для противника. Многие другие игры также имеют частично наблюдаемые версии: "фантомные го", "фантомные крестики-нолики", "экранные сёги".

5.6.1. Кригшпиль: частично наблюдаемые шахматы

Правила игры кригшпиль следующие: белые и черные видят доски, содержащие только их собственные фигуры. Арбитр, который может видеть все фигуры, выступает в игре в качестве судьи и периодически делает объявления, которые слышат оба игрока. Сначала белые предлагают арбитру ход, который был бы допустимым, если бы на доске не было черных фигур. Если расположение черных фигур препятствует этому ходу, арбитр объявляет "недопустимо" и белые продолжают предлагать ходы, пока не будет найден допустимый, — по ходу дела собирая информацию о местонахождении черных фигур.

Как только игрок предлагает допустимый ход, арбитр делает одно из следующих объявлений: "Взятие на квадрате X", если была взята одна из фигур противника, или "Шах D", если черному королю угрожает взятие, где D определяет вариант нападения и выбирается из следующих фраз: "конем", "по горизонтали", "по вертикали", "по длинной диагонали" или "по короткой диагонали". Если черным поставлен мат или они попали в патовое положение, арбитр объявляет об этом, — в противном случае право хода передается черным.

Кригшпиль может показаться игрой сложной до невозможности, но люди справляются с ней довольно хорошо, да и компьютерные программы начинают набирать обороты. Полезно будет вспомнить о понятии **доверительного состояния**, как оно было определено в разделе 4.4 и проиллюстрировано на рис. 4.14, — это набор всех *логически возможных* состояний на доске с учетом полной истории восприятий на текущий момент. Исходно доверительное состояние белых включает единственную позицию, поскольку фигуры черных еще не совершали каких-либо ходов. После того как белые сделают первый ход и черные ответят, доверительное состояние белых будет включать 20 позиций, потому что у черных имеется 20 вариантов ответа на любой начальный ход белых. Отслеживание содержимого доверительного состояния по ходу игры в точности представляет собой задачу **оценки состояния**, для которой этап обновления был дан в уравнении (4.6) в разделе 4.4.4. Можно

отобразить оценку состояния игры кригшпиль непосредственно на частично наблюдаемую, недетерминированную рабочую среду из раздела 4.4, если рассматривать противника как источник недетерминизма. В этом случае функция RESULTS для хода белых будет включать (предсказуемый) результат их собственного хода и непредсказуемый результат ожидаемого ответного хода черных.[4]

Учитывая текущее доверительное состояние, белые могут поставить вопрос "Есть ли возможность выиграть игру?" Для частично наблюдаемой игры понятие **стратегии** меняется: вместо определения хода, который следует сделать в ответ на каждый возможный *ход* противника, необходимо определить ход для каждой вероятной *последовательности восприятий*, которая может быть получена.

Для кригшпиля выигрышной стратегией или ▶**гарантированным матом** будет такая стратегия, которая для каждой возможной последовательности восприятий приведет к фактическому объявлению мата для любого возможного состояния на доске в текущем доверительном состоянии независимо от того, как будет ходить оппонент. При таком определении доверительное состояние противника уже не имеет никакого значения, эта стратегия должна работать, даже если противник сможет увидеть все фигуры. Такой подход значительно упрощает необходимые вычисления. На рис. 5.15 представлена часть стратегии гарантированного мата для эндшпиля с королем и ладьей против короля. В этом случае у черных только одна фигура (король), поэтому доверительное состояние белых может быть представлено в виде одной доски, на которой отмечена каждая позиция, которую может занимать черный король.

Для поиска гарантированных матов к пространству доверительных состояний можно применить общий алгоритм И–ИЛИ-поиска, описанный в разделе 4.4. Алгоритм инкрементного поиска в доверительных состояниях, упоминавшийся в разделе 4.4.2, часто позволяет найти матовые ситуации в середине игры вплоть до глубины 9 — далеко за пределами возможностей большинства игроков-людей.

В дополнение к гарантированным матам в кригшпиле присутствует совершенно новая концепция, не имеющая смысла в полностью наблюдаемых играх: ▶**вероятностный мат**. Такие матовые состояния также требуют работы с каждым состоянием игры в доверительном состоянии; они являются вероятностными в отношении рандомизации ходов победившего игрока. Чтобы понять основную идею, рассмотрим задачу поиска одинокого черного короля, используя только белого короля. Просто перемещаясь случайным образом, белый король в *конечном итоге* столкнется с черным королем, даже если последний попытается избежать этой участи, поскольку черные не могут угадывать правильные уклоняющиеся ходы бесконечно. Согласно терминологии теории вероятностей, обнаружение происходит *с вероятностью* 1.

[4] Иногда доверительное состояние может стать слишком большим, чтобы его было удобно представлять просто как список состояний на доске, но пока мы будем игнорировать эту проблему. В главах 7 и 8 предлагаются методы компактного представления для любых, в том числе очень больших, доверительных состояний.

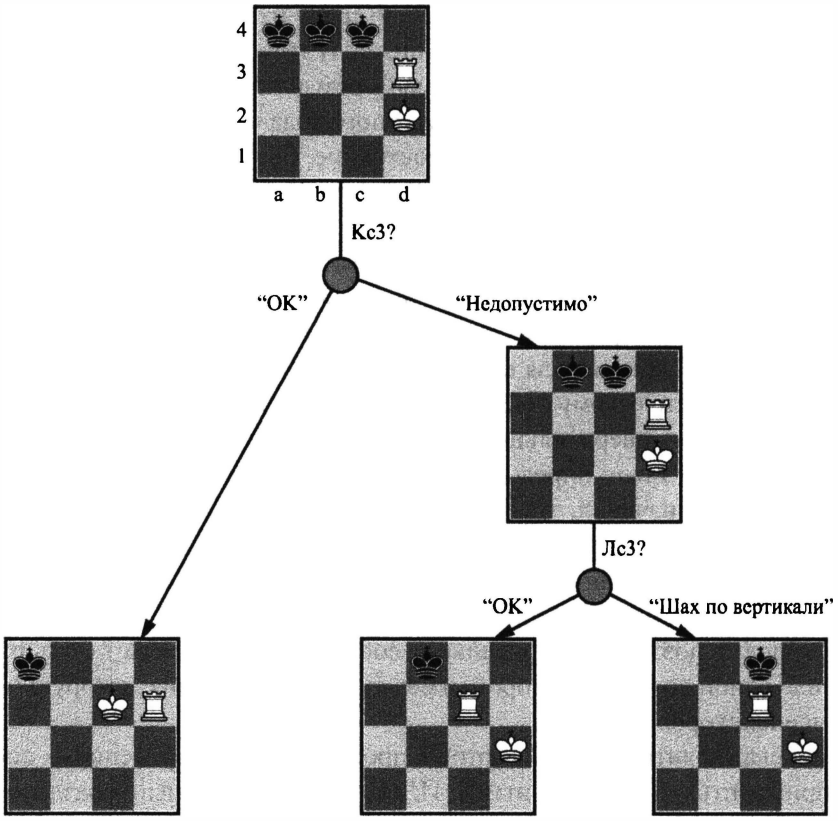


Рис. 5.15. Часть стратегии гарантированного мата в эндшпиле с королем и ладьей против одного короля, представленная на уменьшенной доске. В начальном доверительном состоянии черный король находится в одном из трех возможных мест на доске. За счет выполнения комбинации пробных ходов стратегия сводит возможное положение к единственному возможному месту. Завершение игры до объявления мата оставляется читателю в качестве упражнения

В этом смысле эндшпиль с королем, слоном и конем против одного короля является выигрышным. Белые возьмут черного короля на прицел с помощью бесконечной последовательности случайных ходов, один из которых черные рано или поздно угадают неправильно и тем раскроют свою позицию, что неизбежно приведет к мату. С другой стороны, эндшпиль с королем и двумя слонами против короля белые выигрывают с вероятностью $1 - \epsilon$. В этом случае белые могут добиться победы, только оставив одного из своих слонов незащищенным на один ход. Если черным повезет оказаться в нужном месте и взять этого слона (недопустимый ход в случае, если слон защищен), то игра будет закончена вничью. Белые могут решиться сделать этот рискованный ход в некоторой случайно выбранной точке

посреди очень длинной последовательности ходов и, таким образом, уменьшить ϵ до сколь угодно малой величины, но никогда не смогут уменьшить ϵ до нуля.

Иногда стратегия объявления мата работает лишь для *некоторых* состояний на доске в текущем доверительном состоянии, но не для остальных. Попытка применения такой стратегии может быть успешной, что приведет к ▶**случайному мату** — случайному в том смысле, что белые не могут *знать*, что это будет мат, — если фигуры черных оказались в нужных местах. (Большинство матов в играх между людьми носят именно такой случайный характер.) Эта идея естественным образом приводит к вопросу о том, *насколько вероятно*, что данная стратегия будет выигрышной, что, в свою очередь, поднимает вопрос о том, *насколько вероятно*, что каждое состояние на доске в текущем доверительном состоянии является истинным состоянием на доске.

Можно предположить, что все состояния на доске в текущем доверительном состоянии одинаково вероятны, но это неверно. Рассмотрим, например, доверительное состояние белых после первого хода черных в игре. По определению (в предположении, что черные играют оптимальным образом) черные должны сделать оптимальный ход, поэтому всем состояниям на доске, достигаемым в результате неоптимальных ходов черных, должны быть назначены нулевые вероятности.

Однако этот аргумент также не совсем верен, поскольку ➨*цель каждого игрока состоит не только в том, чтобы перемещать фигуры в правильные квадраты, но и в том, чтобы минимизировать имеющуюся у противника информацию об их местоположении*. Разыгрывание любой *предсказуемой* "оптимальной" стратегии дает противнику некоторую информацию. Следовательно, оптимальная игра в частично наблюдаемых играх требует готовности играть до некоторой степени *случайно*. (Вот почему инспекторы санитарной службы выполняют инспекционные посещения ресторанов *случайным* образом.) Это означает, что иногда выбирается ход, который может показаться слабым "по своей сущности", но его непредсказуемость придает ему дополнительную силу, поскольку противник вряд ли предусмотрел какую-то защиту от него.

Из приведенных выше соображений может показаться, что такие вероятности, связанные с состояниями на доске в текущем доверительном состоянии, могут быть рассчитаны только с учетом оптимальной рандомизированной стратегии. В свою очередь, вычисление такой стратегии, как кажется, требует знания вероятностей всех тех различных состояний на доске, которые могут иметь место. Эту головоломку можно решить посредством принятия на вооружение существующего в теории игр понятия равновесия решений, которое подробно анализируется в главе 18. Равновесие определяет оптимальную рандомизированную стратегию для каждого игрока. Для кригшпиля вычисление равновесия является слишком дорогостоящей процедурой. В настоящее время разработка эффективных алгоритмов для обычной игры в кригшпиль является открытой темой исследований. Большинство систем выполняют ограниченный по глубине прогноз в

собственном пространстве доверительных состояний, игнорируя доверительное состояние противника. Функции оценки напоминают функции для полностью наблюдаемой игры, но дополнительно включают компонент для определения размера доверительного состояния, — чем меньше, тем лучше! Мы вернемся к теме частично наблюдаемых игр при обсуждении теории игр в разделе 18.2.

5.6.2. Карточные игры

Карточные игры, такие как бридж, вист, червы и покер, имеют *стохастическую* частичную наблюдаемость, т.е. недостающая информация генерируется за счет случайной раздачи карт.

На первый взгляд может показаться, что карточные игры аналогичны играм с бросанием костей: раздача карт происходит случайным образом, а сами карты определяют, какие ходы могут быть сделаны каждым игроком. Однако в картах вся жеребьевка происходит с самого начала! Хотя эта аналогия и является некорректной, она предлагает некий алгоритм: принять начало игры как узел жеребьевки с каждой возможной раздачей как одним из возможных результатов, а затем воспользоваться формулой EXPECTIMINIMAX, чтобы выбрать наилучший ход. Обратите внимание, что при таком подходе единственный узел жеребьевки является корневым узлом дерева, после чего игра становится полностью наблюдаемой. Такой подход иногда называют *усреднением по ясновидению*, поскольку он предполагает, что после того, как раздача была выполнена, игра становится полностью наблюдаемой для обоих игроков. Несмотря на свою интуитивную привлекательность, эта стратегия может сбить с толку. Рассмотрим следующую историю.

> **День первый.** Дорога *A* ведет к горшку с золотыми монетами; дорога *B* ведет к развилке. Можно видеть, что если от развилки пойти налево, то дорога приведет к двум горшкам с золотом, а если пойти направо, то попадешь под автобус.
>
> **День второй.** Дорога *A* ведет к горшку с золотыми монетами; дорога *B* ведет к развилке. Можно видеть, что если от развилки пойти направо, то дорога приведет к двум горшкам с золотом, а если пойти налево, то попадешь под автобус.
>
> **День третий.** Дорога *A* ведет к горшку с золотыми монетами; дорога *B* ведет к развилке. Если от развилки выбрать правильное направление, то найдешь два горшка с золотом, а если неправильное, то попадешь под автобус. К сожалению, остается неизвестным, какое из двух направлений правильное.

Метод усреднения по ясновидению приводит к следующим рассуждениям. В день первый правильным выбором является дорога *B*, в день второй дорога *B* также является правильным выбором, в день третий ситуация такая же, как в день первый или день второй, поэтому дорога *B* все равно должна быть правильным выбором.

Теперь можно понять, как усреднение по ясновидению терпит неудачу: оно не учитывает того *доверительного состояния*, в котором окажется агент после

выполнения действия. Полное игнорирование доверительного состояния нежелательно, особенно когда одна из возможностей представляет собой верную гибель. Поскольку предполагается, что каждое будущее состояние автоматически будет одним из точно известных, в методе усреднения по ясновидению никогда не выбираются действия, которые *собирают информацию* (например, первый этап на рис. 5.15), а также действия, которые скрывают информацию от оппонента или предоставляют информацию партнеру, поскольку предполагается, что эта информация им уже известна. В покере этот метод полностью исключает ▶блеф,[5] исходя из убеждения, что противник может видеть все карты. В главе 17 показано, как создавать алгоритмы, которые делают все это в силу решения задач действительно частично наблюдаемых решений, имеющих результатом стратегию оптимального равновесия (см. раздел 18.2).

Несмотря на указанные недостатки, усреднение по ясновидению может быть достаточно эффективной стратегией, особенно с некоторыми хитростями, позволяющими заставить ее работать лучше. В большинстве карточных игр количество возможных раздач является довольно большим. Например, в игре в бридж каждый игрок видит только две из четырех рук, т.е. есть две невидимые ему руки по 13 карт в каждой, поэтому количество раздач составляет $\binom{26}{13} = 10\,400\,600$. Поиск решения даже для одной раздачи — задача довольно сложная, поэтому поиск решения для 10 миллионов раздач просто не обсуждается. Один из способов справиться с этим огромным числом является **абстракция**, т.е. расценивание аналогичных рук как идентичных. Например, очень важно, какие в руке есть тузы и короли, тогда как наличие в руке четверок или пятерок не столь важно и от этого можно абстрагироваться.

Другим способом справиться с большим количеством вариантов является прямое отсечение: рассматривается только небольшая случайная выборка из N раздач и снова рассчитывается оценка EXPECTIMINIMAX. Даже при достаточно малом N — скажем, от 100 до 1000 — этот метод дает хорошее приближение. Его вполне можно применять и к детерминированным играм, таким как кригшпиль, анализируя лишь некоторую выборку возможных состояний игры вместо всех возможных вариантов, если только существует некоторый способ оценить, насколько вероятно каждое из возможных состояний. Также может оказаться полезным выполнять эвристический поиск лишь на определенную глубину вместо поиска по всему дереву игры.

До сих пор предполагалось, что каждая раздача одинаково вероятна. Это имеет смысл для таких игр, как вист и черви. Однако в бридже началу игры предшествует этап торгов, на котором каждая команда указывает, сколько взяток она ожидает взять. Поскольку игроки делают заявки на основании имеющихся у них карт, другие игроки получают некоторую информацию о вероятности $P(s)$ каждой раздачи. Принять это во внимание при поиске решения о том, как разыграть собственную

[5] Блеф — решение делать ставки так, как будто у вас хорошая рука, даже если на самом деле это не так — является основной частью стратегии покера.

руку, достаточно сложно по причинам, указанным в приведенном выше обсуждении игры кригшпиль: игроки могут вести торг таким образом, чтобы свести к минимуму информацию, передаваемую их оппонентам.

Компьютеры уже достигли сверхчеловеческого уровня эффективности игры в покер. Программа игры в покер Libratus приняла участие в 20-дневном безлимитном турнире по Техасскому холдему с четырьмя лучшими игроками в покер в мире и решительно победила их всех. Поскольку в покере очень много возможных состояний, в программе Libratus для сокращения пространства состояний используется абстракция: две руки **AAA72** и **AAA64** считаются эквивалентными (это "три туза и несколько младших карт"), а также ставки в 200 долларов и 201 доллар могут рассматриваться как одинаковые. Более того, программа Libratus вела наблюдение за другими игроками и после каждого матча по ночам анализировала стратегию оппонентов и вносила необходимые изменения в собственную стратегию, стремясь закрыть обнаруженные бреши. В целом, чтобы добиться победы, на разработку и *обучение* этой программы было использовано около 25 миллионов часов работы суперкомпьютера.

Огромные затраты вычислительных мощностей на подготовку программы Libratus (и аналогичные затраты для программ AlphaZero и других подобных систем) позволяют предположить, что игра на уровне чемпиона мира может оказаться недостижимой для исследователей с ограниченным бюджетом. В какой-то степени это так: точно так же, как не следует ожидать, что вам удастся собрать гоночную машину, способную победить в чемпионате Формулы-1, из запасных частей в своем гараже, несомненное преимущество всегда будет у тех, у кого есть доступ к суперкомпьютерам или специальному оборудованию, такому как тензорный процессор. Это особенно верно при обучении системы, но обучение можно успешно проводить и с помощью краудсорсинга. Например, система LeelaZero с открытым исходным кодом представляет собой повторную реализацию системы AlphaZero, которая обучается посредством игры с самой собой на компьютерах участников-добровольцев. После обучения вычислительные требования к ведению реальных игр в турнирах становятся более скромными. Программа AlphaStar выигрывает игры StarCraft II, работая на скромном настольном ПК с одним графическим процессором, а программу AlphaZero также можно запустить в этом режиме.

5.7. Ограничения игровых алгоритмов поиска

Поскольку вычисление оптимальных решений в сложных играх часто является просто неосуществимым, во всех алгоритмах приходится использовать некоторые предположения и приблизительные оценки. В альфа-бета-поиске для приблизительной оценки используется эвристическая функция оценки, а в поиске по методу Монте-Карло вычисляется приблизительное среднее по результатам серии прогонов, выполняемых в случайном режиме. Выбор используемого алгоритма

в определенной степени зависит от особенностей каждой игры: если коэффициент ветвления велик или сложно подобрать эффективную функцию оценки, предпочтение следует отдать поиску по методу Монте-Карло. Тем не менее оба упомянутых алгоритма страдают от фундаментальных ограничений.

Одно важное ограничение альфа-бета-поиска заключается в его уязвимости к ошибкам эвристической функции оценки. На рис. 5.16 показано дерево игры с двумя полуходами. Минимаксный поиск предлагает выбрать правую ветвь, поскольку 100 > 99. Этот выбор будет правильным, если все оценки являются действительно точными. Но предположим, что оценка каждого узла делается с ошибкой, которая не зависит от других узлов и подчиняется нормальному распределению со стандартным отклонением σ. Тогда в действительности левая ветвь будет лучше в 71% случаев, когда σ = 5, и в 58% случаев, когда σ = 2 (потому что один из четырех листьев правой ветви в этом случае, вероятно, станет меньше 99). Если ошибки функции оценки не являются независимыми, то вероятность ошибки еще больше возрастает. Этот факт трудно компенсировать, потому что у нас нет хорошей модели зависимости между значениями узлов, имеющих общего родителя.

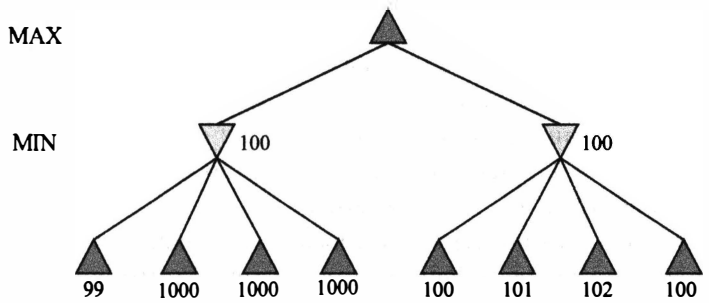


Рис. 5.16. Дерево игры с двумя полуходами, для которого минимаксный поиск с использованием эвристической функции оценки может делать ошибки

Вторым ограничением как для альфа-бета-поиска, так и для поиска по методу Монте-Карло, является то, что они предназначены для расчета (предела) значений допустимых ходов. Но иногда существует один ход, который очевидно лучший (например, когда вообще есть только один допустимый ход), и в этом случае нет смысла тратить вычислительные ресурсы на выяснение его полезности, — лучше просто выбрать этот ход. Хороший алгоритм поиска должен использовать идею *полезности развертывания узла*, выбирая для развертывания только те узлы, которые, вероятно, приведут к открытию значительно лучших ходов. Если нет вариантов развертывания узлов, полезность которых превышает стоимость этой процедуры (в отношении затрат времени), то алгоритм должен прекратить поиск и просто выбрать ход. Это замечание касается не только ситуаций с явными фаворитами, но

и тех случаев, когда имеются *симметричные ходы*, для которых сколь угодно большой объем поиска не позволит показать, что один ход лучше другого.

Данный тип рассуждений о том, какие вычисления следует выполнять, называется ▶**метарассуждениями** (рассуждениями о рассуждениях). Этот подход распространяется не только на ведение игр, но и на рассуждения любого рода в целом. Все вычисления выполняются для того, чтобы выработать лучшие решения, все они имеют стоимость и характеризуются определенной вероятностью достижения конкретного улучшения качества решения. Алгоритм поиска по методу Монте-Карло действительно предусматривает проведение метарассуждений в отношении предоставления вычислительных ресурсов наиболее важным частям дерева, но делает это не оптимальным образом.

Альфа-бета-поиск представляет собой реализацию метарассуждений простейшего вида, а именно — теоремы о том, что некоторые ветви дерева можно игнорировать без ущерба для всей игры. Но такие метарассуждения могут проводиться гораздо лучше. В главе 16 будет показано, как сделать изложенные идеи более точными и реализуемыми.

Третье ограничение состоит в том, что как в алгоритме альфа-бета-поиска, так и в алгоритме поиска по методу Монте-Карло все рассуждения выполняются на уровне отдельных ходов. Очевидно, что люди играют в игры иначе: они могут рассуждать на более абстрактном уровне, преследуя цель более высокого уровня, например поимку ферзя противника, и используя эту цель для избирательной выработки правдоподобных планов. В главе 11 будет обсуждаться этот тип **планирования**, а в разделе 11.4 будет показано, как планировать при наличии иерархии представлений от абстрактных до конкретных.

Четвертая проблема заключается в возможности включения **машинного обучения** в процесс игрового поиска. Ранние игровые программы полагались исключительно на опыт человека в отыскании эффективных функций оценки, выборе поисковых стратегий и приемов повышения эффективности. Лишь совсем недавно появились такие программы, как AlphaZero (Сильвер и др. [2064], 2018), которые полагаются на машинное обучение в процессе игры с самой собой вместо использования экспертных знаний человека в отношении отдельных игр. Тема машинного обучения подробно обсуждается в нескольких главах этой книги начиная с главы 19.

Резюме

В этой главе было рассмотрено несколько разных игр с целью установить, что означает понятие "оптимальная игра", а также разобраться в том, как должен действовать агент в состязательной среде любого типа.

- Любая игра может быть определена с помощью **начального состояния** (вид игрового поля, подготовленного к игре), допустимых **действий** в каждом состоянии, **результатов** каждого действия, функции **проверки**

терминального состояния (позволяющей определить, когда игра окончена) и **функции полезности**, применяемой к терминальным состояниям и указывающей, кто выиграл и каков финальный счет.

- В играх с двумя игроками, дискретных, детерминированных, пошаговых, с **полной информацией** и нулевой суммой для выбора оптимальных ходов может использоваться алгоритм **минимаксного поиска** с перебором узлов в глубину.
- Алгоритм **альфа-бета-поиска** вычисляет такие же оптимальные ходы, как и алгоритм минимаксного поиска, но позволяет достичь гораздо большей эффективности, исключая из рассмотрения поддеревья, которые, по всей вероятности, не нужны для нахождения решения.
- Обычно не представляется возможным рассмотреть все дерево игры (даже с помощью альфа-бета-поиска), поэтому необходимо в какой-то точке останавливать поиск и применять **функцию оценки**, определяющую приближенное значение полезности достигнутого состояния.
- Альтернативный вариант — **поиск по дереву методом Монте-Карло** (MCTS) — оценивает состояния не с помощью эвристической функции, а посредством симуляции хода игры от текущего узла до терминального состояния с использованием правил игры для определения победителя. Поскольку ходы, выбранные во время **прогона**, не обязательно будут оптимальными, процесс симуляции повторяется несколько раз, а оценка определяется усреднением полученных результатов.
- Во многих игровых программах используются таблицы предварительно вычисленных лучших ходов для начала и завершения игры, что позволяет вместо поиска использовать простой просмотр таблицы.
- Ведение игр с элементами случайности можно осуществить с помощью алгоритма EXPECTIMINIMAX — расширения алгоритма минимаксного поиска, в котором **узлы жеребьевки** оцениваются посредством определения средней полезности всех их дочерних узлов с учетом вероятности каждого дочернего узла.
- В играх **с неполной информацией**, таких как кригшпиль, ведение оптимальной игры требует рассуждений о текущем и будущем **доверительных состояниях** для каждого игрока. Одна из простых аппроксимаций может быть получена путем усреднения значения данного действия по всем возможным конфигурациям недостающей информации.
- Программы уже продемонстрировали свою способность победить в турнирах лучших игроков-людей в таких играх, как шахматы, шашки, червы, го, покер и многих других. Человек еще удерживает свои позиции лишь в нескольких играх с неполной информацией, таких как бридж и кригшпиль. В видеоиграх, таких как StarCraft и Dota 2, программы соревнуются с экспертами-людьми, однако часть их успеха можно связать со способностью программ выполнять многие действия очень быстро.

Библиографические и исторические заметки

В 1846 году Чарльз Бэббидж принял участие в дискуссии о возможности компьютерных шахмат и шашек ([1626], 1961). Он, не имея представления об экспоненциальной сложности деревьев поиска, утверждал, что "комбинации, обеспечиваемые его аналитической машиной, с лихвой покрывают любые потребности даже при игре в шахматы". Бэббидж также спроектировал, но не построил машину специального назначения для игры в крестики-нолики. Первая реальная машина для ведения игры была построена в 1890 году испанским инженером Леонардо Торресом и Кеведо. Она была предназначена для разыгрывания шахматного эндшпиля в конфигурации "король и ладья против короля" и гарантировала победу из любой начальной позиции в игре с этими фигурами. Упоминание **минимаксного** алгоритма можно найти уже в 1912 году в статьях Эрнста Цермела, создателя современной теории множеств.

Ведение игр было одной из первых задач, анализировавшихся в исследованиях в области ИИ усилиями таких пионеров, как Конрад Цузе ([2454], 1945), Норберт Винер в его книге "Кибернетика" ([2339], 1948) и Алан Тьюринг ([2236], 1953). Но именно Клод Шеннон в статье "Программирование компьютера для игры в шахматы" ([2040], 1950) изложил основные принципы создания современных программ ведения игр: представление позиций на игровом поле, функция оценки, понятие спокойной позиции и некоторые идеи для селективного поиска по дереву игры. Слейтер ([2085], 1950) выдвинул идею представления функции оценки как линейной комбинации функций, подчеркивая важность изменчивости этой функции в шахматах.

Джон Маккарти сформулировал идею **альфа-бета**-поиска в 1956 году, но опубликовал свое открытие позднее (Харт и Эдвадс [973], 1961). Кнут и Мур ([1245], 1975) доказали правильность альфа-бета-алгоритма и проанализировали его временную сложность, в то время как Перл [1187] показал, что альфа-бета-поиск является асимптотически оптимальным среди всех алгоритмов поиска в дереве игры на постоянную глубину.

В работе Берлинера [185] (1979) представлен алгоритм эвристического поиска B*, в котором поддерживается интервал возможной оценки стоимости узла в дереве игры вместо присваивания ему одноточечной оценки. Дэвид Макаллестер в статье [1522] (1988) предложил метод поиска "заветных чисел" (*conspiracy number*), предполагающего развертывание тех листовых узлов, изменение значения которых может заставить программу предпочесть новый ход в корне дерева. В алгоритме MGSS* ([1946], 1989) теория принятия решений, рассматриваемая в главе 16, используется для оценки ценности развертывания каждого листа с точки зрения ожидаемого улучшения качества решения в корне.

Алгоритм SSS* (Штокман [2132], 1979), который можно рассматривать как версию алгоритма A* для двух игроков, никогда не развертывает больше узлов, чем альфа-бета-алгоритм, но высокие требования к памяти делают его непрактичным. Позднее Корф и Чикеринг ([1288], 1996) разработали на основе алгоритма RBFS

новую версию SSS* с линейно возрастающими потребностями в пространстве. Баум и Смит ([142], 1997) предложили заменить алгоритм минимаксного поиска алгоритмом, основанным на учете вероятностей, и показали, что он приводит к осуществлению лучших вариантов выбора в некоторых играх. Алгоритм поиска на основе **ожидаемого минимаксного значения** был предложен Дональдом Мичи ([1563], 1966). Брюс Боллард ([119], 1983) распространил метод альфа-бета-отсечения на деревья с узлами жеребьевки. В книге Перла "Эвристика" ([1751], 1984) тщательно анализируются многие игровые алгоритмы.

Симуляция по методу Монте-Карло была впервые предложена Метрополисом и Уламом ([1561], 1949) для расчетов, связанных с разработкой атомной бомбы. Алгоритм поиска по дереву методом Монте-Карло (MCTS) был представлен Абрамсоном ([7], 1987). Тесауро и Гальперин ([2194], 1997) показали, как поиск по дереву методом Монте-Карло можно комбинировать с функцией оценки в игре в нарды. Метод раннего завершения прогона изучался Лоренцем ([1444], 2015). В программе AlphaGo успешно реализовано раннее завершение прогонов и применение функции оценки (Сильвер и др. [2063], 2016). Подход с использованием стратегии выбора "Верхние границы доверия, применяемые к деревьям" был усовершенствован Коксисом и Сепесвари ([1251], 2006). В работе [400] (2008) показано, как алгоритм MCTS может быть применен к различным играм, а в [324] (2012) дан соответствующий обзор.

Коллер и Пфеффер в статье [1264] (1997) описали систему для полного решения **частично наблюдаемых** игр. Она могла работать с бо́льшими играми, чем предыдущие системы, но все же не с полной версией сложных игр, таких как покер или бридж. Фрэнк и др. в [771] (1998) описывают несколько вариантов алгоритма поиска по дереву в Монте-Карло для частично наблюдаемых игр, в том числе тот, в котором игрок MIN обладает полной информацией, а игрок MAX — нет. Шофильд и Тильшер в [1997] (2015) сообщают, что они адаптировали общую систему ведения игр к частично наблюдаемым играм.

Фергюсон вручную разработал рандомизированные стратегии победы в эндшпиле кригшпиля для короля со слоном и конем ([728], 1992) или короля с двумя слонами ([729], 1995) против одного короля. Первые программы игры в кригшпиль сосредоточивались на отыскании мата в эндшпиле и выполняли И-ИЛИ поиск в пространстве доверительных состояний ([1964], 2002 и [242], 2003). Алгоритмы инкрементного поиска в пространстве доверительных состояний позволили находить гораздо более сложные матовые решения в середине игры (Рассел и Вольф [1948], 2005; Вольф и Рассел [2371], 2007), но рациональная оценка состояний оставалась главным препятствием для эффективного общего ведения игры (Паркер и др. [1733], 2005). Цианкарини и Фавини сообщили в [439] (2010) об успешном применении в кригшпиле алгоритма MCTS, а в статье [2294] (2018) описывается версия алгоритма MCTS для поиска в пространстве доверительных состояний при игре в "фантом го".

В **шахматах** главные достижения отмечались регулярным определением победителей в конкурсе на получение премии Фредкина: программа Belle (Кондон

и Томпсон [467], 1982) была первой программой, добившейся статуса гроссмейстера. Программа DEEP THOUGHT (Су [1081], 1990) первой получила статус гроссмейстера международного класса, а программа DEEP BLUE (Кэмпбелл и др. [361], 2002; Су [1080], 2004) смогла победить чемпиона мира Гарри Каспарова в показательном матче 1997 года. В программе DEEP BLUE альфа-бета-поиск выполнялся со скоростью более 100 миллионов позиций в секунду и мог генерировать единичные расширения, которые иногда достигали глубины 40 полуходов.

Сегодня лучшие шахматные программы (например, STOCKFISH, KOMODO, HOUDINI) намного превосходят любого игрока-человека. Они уже достигли эффективного фактора ветвления менее чем 3 (по сравнении с действительным фактором ветвления на уровне 35) и осуществляют поиск на примерно 20 полуходов со скоростью около одного миллиона узлов в секунду на стандартном одноядерном компьютере. Они используют такие методы отсечения, как ►**эвристика пустого хода**, позволяющая генерировать хорошую нижнюю границу стоимости состояния за счет использования ограниченного поиска, при котором оппоненту сначала предоставляется возможность сделать два хода. Также важным методом является ►**отсечение бесполезных вариантов**, помогающее заранее определить, какие ходы приведут к бета-отсечению в узлах-преемниках. Программа SUNFISH представляет собой упрощенную шахматную программу для учебных целей, ядро которой включает менее 200 строк кода на языке Python.

Идея использования ретроградного анализа для вычисления эндшпильных таблиц принадлежит Беллману ([166], 1965). Воспользовавшись этой идеей, Кен Томпсон ([2205], 1986; [2206], 1996) и Льюис Стиллер ([2130], 1992; [2131],1996) решили все шахматные эндшпили для пяти фигур. Стиллер обнаружил один случай, когда возможность достижения неизбежного матового состояния существовала, но на это требовалось 262 хода, что вызвало некоторые опасения, поскольку шахматные правила требуют, чтобы взятие или ход пешкой обязательно имел место в интервале в 50 ходов, иначе партия объявляется законченной вничью. В 2012 году Владимир Махнычев и Виктор Захаров вычислили Ломоносовскую табличную базу эндшпиля, в которой были решены все позиции в эндшпиле вплоть до семи фигур — в некоторых случаях требуется более 500 ходов без взятия. Размер такой таблицы для семи фигур достигает 140 терабайтов, а для восьми фигур подобная таблица будет иметь размер в 100 раз больше.

В 2017 году программа ALPHAZERO (Сильвер и др. [2064], 2018) нанесла поражение программе STOCKFISH (чемпион TCEC по компьютерным шахматам 2017 года) в матче из 1000 игр со 155 победами и 6 проигрышами. Дополнительные матчи также окончились решительными победами ALPHAZERO, — даже тогда, когда этой программе предоставлялась лишь 1/10 времени, отведенного программе STOCKFISH.

Гроссмейстер Ларри Кауфман был удивлен успехами этой программы, построенной на использовании алгоритма поиска в дереве игры по методу Монте-Карло, и отметил, что "Возможно, это хорошо, что текущему доминированию минимаксных движков шахматных программ приходит конец, но еще рано утверждать это".

Гарри Каспаров дал такой комментарий: "Это замечательное достижение, даже если его следовало ожидать после появления AlphaGo. Это приближение машинных шахмат к человеческому подходу типа B, о котором мечтали Клод Шеннон и Алан Тьюринг, заменяющее собой методы грубой силы". Он продолжил предсказанием: "Шахматный мир уже потрясен программой AlphaZero до самых корней, но это лишь крошечный пример того, что ждет нас впереди. Такие закоснелые в своих методах дисциплины, как медицина и образование, также будут потрясены" (Сэдлер и Риган [1959], 2019).

Шашки были первой из классических игр, в которые играл компьютер (Стрейчи [2142], 1952). Артур Самуэль ([1969], 1959; [1970], 1967) разработал программу игры в шашки, внутренняя функция оценки которой обучалась в процессе игры с самой собой с помощью определенной формы обучения с подкреплением. Можно считать невероятным достижением то, что Самуэль сумел создать программу, которая играла в шашки лучше него самого на компьютере IBM 704 со всего лишь 10 000 слов памяти и процессором с тактовой частотой 0,000001 ГГц. Menace, *Machine Educable Noughts And Crosses Engine*, машина для обучения игре в крестики-нолики с помощью машинного обучения (Михи, [1566], 1963), также использовала методы обучения с подкреплением, чтобы стать компетентным игроком в крестики-нолики. Ее процессор был еще медленнее: набор из 304 игровых полей с цветными бусинами, предназначенными для представления лучших изученных ходов в каждой позиции.

В 1992 году программа игры в шашки Chinook Джонатана Шеффера бросила вызов легендарному Мариону Тинсли, который был чемпионом мира более 20 лет. Тинсли выиграл этот матч, но все же проиграл две партии — это были четвертый и пятый проигрыши за всю его карьеру. После того как Тинсли вышел на пенсию по состоянию здоровья, корона чемпиона перешла к программе Chinook. Вся эта история была описана Шеффером в [1987] (2008).

В 2007 году Шеффер и его команда полностью "решили" шашки ([1988], 2007): эту игру всегда можно свести к ничьей при идеальной игре. Ричард Беллман ([166], 1965) в свое время предсказал это: "В шашках число возможных ходов в любой заданной ситуации настолько невелико, что для нее можно с уверенностью ожидать нахождение полного цифрового компьютерного решения задачи оптимальной игры". Следует отметить, что Беллман неверно оценил масштабы необходимых усилий: таблица эндшпиля с 10 шашками насчитывает 39 триллионов записей. Принимая во внимание размер этой таблицы, не удивительно, что для нахождения полного решения этой игры потребовалось 18 лет работы процессора, выполняющего альфа-бета-поиск.

И.Дж. Гуд, который учил **игре в го** Алана Тьюринга, написал в [892] (1965): "Я думаю, что запрограммировать компьютер так, чтобы он разумно играл в го, будет еще труднее, чем в шахматы". И он был прав: до 2015 года программы игры в го не поднялись в игре выше уровня любителя. Первые публикации на эту тему были обобщены в статьях Боузи и Казенаве ([271], 2001), а также Мюллера ([1633], 2002).

Зобрист в [251] (1970) в качестве многообещающего метода для игры в го предложил визуальное распознавание, в то время Шраудольф и соавт. ([2001], 1994) проанализировали возможности использования обучения с подкреплением. Любберц и Миккулайнен в [1458] (2001) рекомендовали обратиться к нейронным сетям, а Брюгманн ([327], 1993) применил к игре в го алгоритм поиска в дереве игры по методу Монте-Карло. В программе AlphaGo (Сильвер и др. [2063], 2016) все эти четыре идеи были объединены в одно целое, что позволило победить игроков-профессионалов высшего уровня Ли Седола (со счетом 4:1 в 2015 году) и Ке Джи (со счетом 3:0 в 2016 году).

Ке Джи заметил: "После того как человечество потратило тысячи лет на улучшение нашей тактики, компьютеры показали, что люди полностью заблуждались. Я даже готов пойти так далеко, чтобы открыто заявить, что ни один человек еще даже не прикоснулся к краю истинных знаний о го". Ли Седол удалился от игры в го, сокрушаясь: "Даже если бы я стал номером один, есть сущность, которую победить невозможно".

В 2018 году программа AlphaZero превзошла AlphaGo в игре в го, а также победила лучшие программы игры в шахматы и сёги, обучившись этим играм за счет самостоятельной игры с самой собой без доступа к каким-либо экспертным человеческим знаниям или информации о каких-либо прошлых играх. (Безусловно, она полностью полагалась на людей в определении базовой архитектуры поиска в дереве игры по методу Монте-Карло и глубоких нейронных сетей, а также в обучении с подкреплением и кодировании правил игры.) Успех AlphaZero вызвал повышенный интерес к обучению с подкреплением как ключевому компоненту общего ИИ (см. главу 22). Продвинувшись еще на шаг дальше, система MuZero работает, даже будучи не поставленной в известность о правилах игры, в которую она играет, — она должна самостоятельно их выяснить, просто играя в эту игру. Система MuZero достигла уровня самых современных результатов в играх Pacman, шахматы, го и в 75 играх Атари (Шриттвейзер и др. [2002], 2019). Система научилась обобщать; например, играя в Pacman, она выучила, что действие "вверх" перемещает игрока вверх на один квадрат (если этому не препятствует стена) даже несмотря на то, что возможность наблюдать результат действия "вверх" она могла лишь в небольшом проценте положений на игровой доске.

Игра **реверси**, также называемая "Отелло", имеет меньшее пространство поиска, чем шахматы, но определение для нее функции оценки является сложной задачей, поскольку преимущество в материале в этой игре не столь важно, как мобильность. Соответствующие программы показали уровень игры выше человеческого уже в 1997 году (Буро [349], 2002).

Нарды, игра с бросанием костей, математически была проанализирована еще Джероламо Кардано ([370], 1663), но впервые реализована в качестве компьютерной игры только в программе BKG (Берлинер [187], 1980), в которой автор применил очень сложную, сконструированную им вручную функцию оценки и поиск только на глубину 1. Это была первая программа, которая смогла победить

человека со званием чемпиона мира в одной из основных классических игр (Берлинер [186], 1980), хотя по окончании турнира ее создатель сразу же заявил, BKG очень повезло с бросанием костей. Программа Герри Тезауро ([2193], 1995) TD-GAMMON построила свою функцию оценки за счет обучения с использованием нейронных сетей при игре с самой собой. Она неизменно играла на уровне чемпионов мира и заставила аналитиков-людей изменить свое мнение о лучшем начальном ходе для нескольких результатов бросков костей.

В игре в **покер**, как и в го, в последние годы наблюдаются удивительные успехи. Боулинг и соавт. ([272], 2015) использовали теорию игр (см. раздел 18.2) для определения точной оптимальной стратегии для версии покера только с двумя игроками и фиксированным числом поднятий с фиксированным размером ставки. В 2017 году впервые игроки уровня чемпионов были побеждены в игре один на один (два игрока) в безлимитный Техасский холдем в двух независимых матчах с программой LIBRATUS (Браун и Сэндхолм [320], 2017) и DEEPSTACK (Моравчик и др. [1619], 2017). В 2019 году программа PLURIBUS (Браун и Сэндхолм [321], 2019) победила профессиональных игроков-людей высшего ранга в турнире из серии игр в Техасский холдем с шестью игроками. В многопользовательских играх дополнительно возникают некоторые стратегические проблемы, которые обсуждаются в главе 18. Петоса и Балх ([1782], 2019) реализовали многопользовательскую версию программы ALPHAZERO.

Бридж. Смит и соавт. в [2100] (1998) сообщают о том, как программа BRIDGE BARON выиграла чемпионат по компьютерному бриджу 1998 года, используя иерархические планы (см. главу 11) и действия высокого уровня, такие как импас и сквиз, знакомые игрокам в бридж. Гинзберг ([862], 2001) описывает, как его программа GIB, построенная на симуляциях по методу Монте-Карло (впервые этот подход для бриджа предложил Леви ([1396], 1989)), выиграла очередной компьютерный чемпионат и на удивление хорошо показала себя в игре против игроков-людей уровня экспертов. В XXI веке в компьютерных чемпионатах по бриджу доминировали уже две коммерческие программы — JACK и WBRIDGE5. Ни одна из них не была описана в опубликованных статьях, но, как полагают, в обеих используется метод поиска по методу Монте-Карло. В целом программы игры в бридж играют на уровне людей-чемпионов, когда они, собственно, разыгрывают руки, но заметно отстают в фазе торгов, поскольку не полностью понимают те условные соглашения, которые люди используют при общении со своими партнерами. Программисты, пишущие программы игры в бридж, в большей степени заинтересованы в создании полезных, образовательных программ, побуждающих людей начать игру и не имеющих цели побеждать чемпионов.

Эрудит представляет собой игру, в которой игроки-люди любительского уровня испытывают значительные трудности, придумывая слова с высокой оценкой, но для программы очень легко найти максимально возможное количество очков для игрока в любой позиции (Гордон [906], 1994). Для нее самая сложная часть игрового процесса — это планирование наперед в частично наблюдаемой стохастической

игре. Тем не менее в 2006 году программа QUACKLE победила бывшего чемпиона мира Дэвида Бойса со счетом 3:2. Бойс воспринял это хорошо, заявив: "Лучше быть человеком, чем компьютером". Хорошее описание одной из лучших программ, MAVEN, было дано Шеппардом в [2047] (2002).

Видеоигры, такие как StarCraft II, включают сотни частично наблюдаемых юнитов, перемещающихся в режиме реального времени с весьма объемными, почти непрерывными восприятиями[6] и пространствами действий со сложными правилами. Ориол Виньяльс, бывший в возрасте 15 лет чемпионом Испании по игре в StarCraft, описал, как видеоигра может стать испытательным полигоном и серьезной проверкой для методов обучения с подкреплением (Виньяльс и др. [2274], 2017). В 2019 году Виньяльс и команда DeepMind представили программу ALPHASTAR, построенную на глубоком обучении и обучении с подкреплением, в конечном итоге победившую опытных игроков-людей со счетом 10:1 и уверенно занявшую лидирующие позиции среди игроков-людей, получивших официально присвоенный ранг (Виньяльс и др. [2273], 2019). В программе ALPHASTAR были приняты меры для ограничения количества действий в минуту, которые она может выполнять в критически важных рывках, — в ответ на критику со стороны оппонентов, ощущавших, что у программы здесь было несправедливое преимущество.

Компьютеры уже победили ведущих игроков-людей и в других популярных видеоиграх, таких как Super Smash Bros ([743], 2017), Quake III ([1121], 2019) и Dota 2 ([733], 2018), — во всех случаях за счет использования методов глубокого обучения.

Физические игры, такие как **футбол роботов** (Виссер и др. [2277], 2008; Барретт и Стоун [135], 2015), **бильярд** (Лэм и Гринспен [1346], 2008; Арчибальд и др. [66], 2009), а также **настольный теннис** (Сильва и др. [2061], 2015), привлекали некоторое внимание исследователей в области ИИ. В них все те сложности, которые имеют место в видеоиграх, сочетаются с беспорядочностью реального мира.

Соревнования по компьютерным играм происходят ежегодно, в том числе Компьютерные олимпиады, начиная с 1989 года. В Конкурсе универсальных игр (*General Game Competition*) (Лоув и др. [1445], 2006) тестируются программы, которые должны научиться играть в неизвестную игру, представленную только в виде логического описания ее правил. Международная ассоциация компьютерных игр (*International Computer Games Association — ICGA*) публикует журнал *ICGA Journal* и проводит две чередующиеся двухгодичные конференции, Международную конференцию по компьютерам и играм (*The International Conference on Computers and Games — ICCG* или *CG*) и Международную конференцию по достижениям в компьютерных играх (*International Conference on Advances in Computer Games — ACG*). Ассоциация IEEE публикует журнал *IEEE Transactions on Games* и проводит ежегодную Конференцию по вычислительной разведке и играм (*Conference on Computational Intelligence and Games*).

[6] Для человека-игрока объекты движутся непрерывно, но на самом деле они дискретны на уровне пикселя на экране.

Упражнения

5.1. Предположим, у вас есть оракул, $OM(s)$, который правильно предсказывает ходы противника в любом состоянии. Используя это, сформулируйте определение игры как задачи поиска (с одним агентом). Опишите алгоритм нахождения оптимального хода.

5.2. Рассмотрим задачу одновременного решения двух головоломок ***Восьмерка***.

а) Дайте полную формулировку задачи в стиле главы 3.

б) Насколько велико достижимое пространство состояний? Дайте точное числовое выражение.

в) Предположим, что задача преобразуется в многоагентную следующим образом: два игрока ходят по очереди и, чтобы определить головоломку, в которой игроку в этот раз предстоит сделать ход, подбрасывается монета. Побеждает тот, кто первым решит любую из двух головоломок. Какой алгоритм можно использовать для выбора хода в данных условиях?

г) Будет ли игра в конечном счете завершена при условии строго оптимальных действий игроков? Объясните.

5.3. Представим, что в этом упражнении один из двух друзей хочет избежать встречи с другим, — в этом случае задача превращается в игру с двумя игроками. Далее предположим, что игроки ходят по очереди и игра заканчивается, когда игроки оказываются в одном узле. Конечная стоимость игры для преследователя — общее затраченное время, взятое с обратным знаком. (Преследуемый "выиграет", если встреча никогда не состоится.) Возможный вариант игры приведен на рис. 5.17.

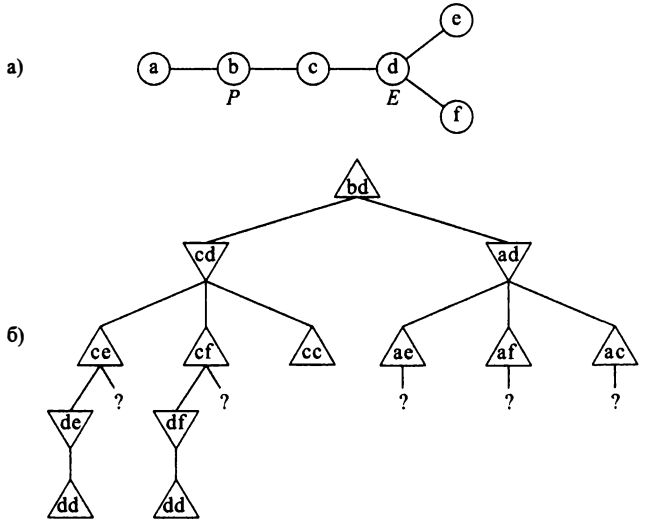


Рис. 5.17. а) Карта, на которой стоимость каждого ребра равна 1. Исходно преследователь P находится в узле b, а преследуемый E — в узле d. **б)** Часть дерева игры для этой карты, на которой каждый узел помечен соответственными позициями P и E (две буквы). Преследователь P начинает движение первым. Ветви, отмеченные знаком "?", еще предстоит исследовать

а) Скопируйте дерево игры и отметьте значения его терминальных узлов.
б) Рядом с каждым внутренним узлом напишите свое наиболее вероятное предположение о его значении (конкретное число, одно или более неравенств, таких как "≥ 14" или "?").
в) Под каждым вопросительным знаком проставьте метку узла, достигаемого по этой ветке.
г) Объясните, как оценка значения узлов, добавленных в п. *в*, может быть получена из рассмотрения длины кратчайшего пути на карте, а затем выведите такие оценки для этих узлов. Держите в памяти стоимость пути к каждому листу, а также стоимость их решения.
д) Теперь предположим, что данное дерево с граничными оценками листьев, полученных в п. *г*, оценивается слева направо. Обведите те узлы с пометкой "?", которые не потребуется разворачивать в дальнейшем с учетом границ, полученных в п. *г*, и вычеркните те узлы, которые вообще не нужно рассматривать.
е) Можете ли вы доказать что-нибудь в общем случае относительно того, кто выигрывает в этой игре на карте, представляющей собой дерево?

5.4. Опишите и реализуйте описания состояний, генераторы ходов, проверки терминальных состояний, функции полезности и функции оценки для одной или нескольких из следующих стохастических игр: монополия, эрудит, бридж (собственно игра после завершения торгов) и покер в версии Техасский холдем.

5.5. Опишите и реализуйте игровую среду для игры в *реальном времени* с *несколькими игроками*, в которой время является частью состояния среды, а игрокам выделяются фиксированные промежутки времени.

5.6. Обсудите, насколько успешно может применяться стандартный подход к ведению игры компьютером в таких играх, как теннис, бильярд и крокет, происходящих в непрерывном физическом пространстве состояний.

5.7. Докажите следующее утверждение: для каждого дерева игры значение полезности, полученное игроком MAX с использованием минимаксных решений в игре против неоптимально действующего игрока MIN, никогда не будет ниже значения полезности, полученной в игре против оптимально действующего игрока MIN. Можете ли вы предложить дерево игры, в котором игрок MAX может действовать еще лучше, используя неоптимальную стратегию против неоптимально действующего игрока MIN?

5.8. Рассмотрим игру с двумя игроками, представленную на рис. 5.18.

а) Нарисуйте полное дерево игры с использованием следующих соглашений:
- запишите каждое состояние как (s_A, s_B), где s_A и s_B обозначают местонахождения фишек;
- обведите каждое терминальное состояние квадратом и напишите его значение в кружке;
- циклические состояния (те, которые уже появлялись на пути к корню) обведите двойными квадратами и, поскольку пока непонятно, как присваивать значения таким циклическим состояниям, обозначьте каждое из них знаком "?" в кружке.

б) Теперь обозначьте каждый узел его зарезервированным минимаксным значением (также в кружке). Объясните, как вы учитывали значения "?" и почему.
в) Объясните, почему стандартный алгоритм минимаксного поиска в этом дереве игры потерпит неудачу, и вкратце опишите, каким образом вы могли бы его

исправить, исходя из вашего ответа в п. *б*. Приведет ли модифицированный вами алгоритм к получению оптимальных решений для всех игр с циклами?

г) Эту игру с четырьмя квадратами можно обобщить до n квадратов для любого $n > 2$. Докажите, что игрок A побеждает, если число n четное, и проигрывает, если число n нечетное.

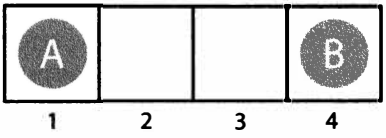


Рис. 5.18. Начальная позиция простой игры. Игрок A ходит первым. Два игрока ходят по очереди, и каждый игрок должен переместить свою фишку в свободный смежный квадрат в любом направлении. Если противник занимает смежный квадрат, то игрок может перенести свою фишку через фишку противника на следующий свободный квадрат, если таковой имеется. (Например, если фишка A стоит в квадрате 3, а фишка B — в квадрате 2, то A снова может перейти в квадрат 1.) Игра заканчивается после того, как один из игроков достигает противоположного конца доски. Если игрок A достигает квадрата 4 первым, то значение этой игры для A равно +1; если игрок B достигает квадрата 1 первым, то значение этой игры для игрока A равно −1

5.9. В этой задаче рассматриваются основные концепции ведения игр с использованием в качестве примера игры в крестики-нолики. Определим X_n как количество строк, столбцов или диагоналей, в которых находится точно n значков X и ни одного значка O. Аналогичным образом O_n — это количество строк, столбцов или диагоналей, в которых находятся только n значков O. Функция полезности присваивает +1 любой позиции с $X_3 = 1$ и −1 — любой позиции с $O_3 = 1$. Все остальные терминальные позиции имеют полезность 0. Для нетерминальных позиций используется линейная функция оценки, определенная как $Eval(s) = 3X_2(s) + X_1(s) - (3O_2(s) + O_1(s))$.

а) Сколько приблизительно существует возможных вариантов игры в крестики-нолики?

б) Представьте полное дерево игры, начиная от пустой доски вплоть до глубины 2 (т.е. когда на доске имеется один значок X и один значок O), учитывая симметрию позиций.

в) Проставьте в этом дереве оценки всех позиций на глубине 2.

г) Используя алгоритм минимаксного поиска, проставьте в дереве зарезервированные значения для позиций на глубине 1 и 0, а затем воспользуйтесь этими значениями для выбора наилучшего начального хода.

д) Обведите кружками узлы на глубине 2, которые не оценивались бы в случае применения альфа-бета-отсечения, исходя из предположения, что эти узлы генерируются в порядке, оптимальном для альфа-бета-отсечения.

5.10. Рассмотрите семейство обобщенных игр в крестики-нолики, определяемых следующим образом. Каждая конкретная игра задается набором квадратов S и коллекцией

выигрышных позиций W. Каждая выигрышная позиция является подмножеством S. Например, в стандартной игре в крестики-нолики S — это множество из 9 квадратов, а W — это набор из 8 подмножеств в W: три строки, три столбца и две диагонали. В остальном игра идентична стандартным крестикам-ноликам. Начиная с пустого поля, игроки поочередно размещают свои метки на свободном квадрате. Выигрывает игру тот игрок, который первым отметил каждый квадрат в некоторой выигрышной позиции. Ничья объявляется в том случае, когда все квадраты помечены, но ни один из игроков не выиграл.

а) Пусть $N = |S|$, т.е. количеству квадратов. Определите верхнюю границу количества узлов в полном дереве игры для обобщенной игры в крестики-нолики как функцию от N.

б) Определите нижнюю границу размера дерева игры для худшего случая, где $W =$.

в) Предложите приемлемую функцию оценки, которую можно будет использовать для любого экземпляра обобщенной игры в крестики-нолики. Функция может зависеть от S и W.

г) Предположим, что можно сгенерировать новую позицию на игровом поле и проверить, является ли она выигрышной при выполнении до $100N$ машинных команд, принимая, что для расчетов используется процессор с тактовой частотой 2 ГГц, а любые ограничения памяти игнорируются. Воспользовавшись своей оценкой из п. *а*, приблизительно оцените, насколько большое игровое дерево может быть полностью решено с использованием альфа-бета-поиска за секунду процессорного времени. А за минуту? За час?

5.11. Разработайте общую игровую программу, способную играть в различные игры.

а) Реализуйте генераторы хода и функции оценки для одной или нескольких из следующих игр: калах, реверси, шашки и шахматы.

б) Сконструируйте агента общего назначения для ведения игры на основе альфа-бета-поиска.

в) Сравните эффект от увеличения глубины поиска, усовершенствования способа упорядочения ходов и улучшения функции оценки. Насколько близко приближается полученный вами эффективный коэффициент ветвления к идеальному случаю самого совершенного упорядочения ходов?

г) Реализуйте алгоритм выборочного поиска, такой как B* ([185], 1979), поиск "заветных чисел" ([1522], 1988) или алгоритм MGSS* ([1946], 1989), и сравните его производительность с производительностью алгоритма A*.

5.12. Опишите, как изменятся алгоритмы минимаксного и альфа-бета-поиска для игр с двумя игроками с ненулевой суммой, в которых у каждого игрока есть собственная функция полезности и обе эти функции известны обоим игрокам. Если нет никаких ограничений на две терминальные полезности, возможно ли, чтобы какой-либо узел был отсечен алгоритмом альфа-бета-поиска? Что если функции полезности игроков в любом состоянии различаются не более чем на константу k, что делает игру почти кооперативной?

5.13. Опишите, как изменятся алгоритмы минимаксного и альфа-бета-поиска для игр с двумя игроками с ненулевой суммой, в которых у каждого игрока есть собственная функция полезности, и обе эти функции известны обоим игрокам. Если нет никаких ограничений на две терминальные полезности, возможно ли, чтобы какой-либо узел был отсечен алгоритмом альфа-бета-поиска? Что если функции полезности игроков

в любом состоянии при суммировании дают число между константами $-k$ и k, что делает игру почти игрой с нулевой суммой?

5.14. Разработайте формальное доказательство правильности алгоритма альфа-бета-отсечения. Для этого рассмотрите ситуацию, показанную на рис. 5.19. Вопрос заключается в том, следует ли выполнять отсечение в узле n_j, который представляет собой узел игрока MAX и является одним из потомков узла n_1. Основная идея такова, что отсечение должно выполняться тогда и только тогда, когда можно показать, что минимаксное значение n_1 не зависит от значения n_j.

а) Значение узла n_1 определяется как минимальное значение из значений его дочерних узлов: $n_1 = \min(n_2, n_{21}, \ldots, n_{2b_2})$. Найдите аналогичное выражение для узла n_2, а отсюда выражение для узла n_1 в терминах значений узла n_j.

б) Допустим, что l_i — минимальное (или максимальное) значение узлов *слева* от узла n_i на глубине i, минимаксное значение которого уже известно. Аналогичным образом допустим, что r_i — минимальное (или максимальное) значение неисследованных узлов справа от n_i на глубине i. Перепишите свое выражение для узла n_1 в терминах значений l_i и r_i.

в) Теперь переформулируйте это выражение так, чтобы показать, что значение узла n_j не должно превышать некоторый предел, полученный на основе значений l_i, для того, чтобы оно могло повлиять на значение узла n_1.

г) Повторите этот процесс для случая, когда n_j является узлом игрока MIN.

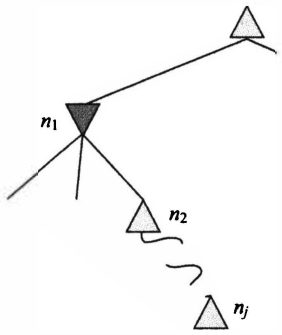


Рис. 5.19. Ситуация, возникающая при определении, следует ли отсекать узел n_j

5.15. Докажите, что алгоритм альфа-бета-отсечения требует времени $O(b^{m/2})$ при оптимальном порядке выполнения ходов, где m — максимальная глубина дерева игры.

5.16. Предположим, что имеется шахматная программа, способная оценивать 5 миллионов узлов в секунду. Выберите компактное представление состояния игры для хранения в таблице транспозиций. Сколько примерно записей удастся вам поместить в таблицу размером 1 Гбайт, находящуюся в памяти? Будет ли этого достаточно для поиска в течение трех минут, отведенных на один ход? Сколько операций поиска в таблице можно выполнить за время, необходимое для одной оценки? Теперь предположим, что таблица транспозиций сохранена на жестком диске. Сколько примерно оценок можно будет сделать за время, необходимое для выполнения одного

поиска на диске с использованием стандартного аппаратного обеспечения жесткого диска?

5.17. Предположим, что имеется шахматная программа, способная оценивать 10 миллионов узлов в секунду. Выберите компактное представление состояния игры для хранения в таблице транспозиций. Сколько примерно записей удастся вам поместить в таблицу размером 2 Гбайт, находящуюся в памяти? Будет ли этого достаточно для поиска в течение трех минут, отведенных на один ход? Сколько операций поиска в таблице можно выполнить за время, необходимое для одной оценки? Теперь предположим, что таблица транспозиций сохранена на жестком диске. Сколько примерно оценок можно будет сделать за время, необходимое для выполнения одного поиска на диске с использованием стандартного аппаратного обеспечения жесткого диска?

5.18. В этом упражнении рассматривается отсечение в играх с узлами жеребьевки. На рис. 5.20 показано полное дерево игры для некоторой тривиальной игры. Предположим, что листовые узлы должны оцениваться в порядке слева направо и что до того, как листовой узел будет оценен, мы ничего не знаем о его значении — диапазон возможных значений составляет от $-\infty$ до ∞.

а) Скопируйте рисунок, проставьте значения всех внутренних узлов и укажите стрелкой лучший ход в корне дерева.

б) Зная значения первых шести листьев, есть ли необходимость оценивать также седьмой и восьмой листовые узлы? Зная значения первых семи листовых узлов, нужно ли оценивать еще и восьмой узел? Поясните свои ответы.

в) Предположим, что нам известно, что значения листовых узлов лежат в диапазоне от –2 до +2 включительно. После оценки первых двух листовых узлов каков будет диапазон значений для левого узла жеребьевки?

г) Обведите все листовые узлы, которые не нужно оценивать, исходя из предположения, сделанного в п. *в*.

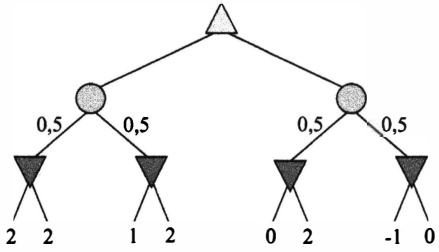


Рис. 5.20. Полное дерево игры для тривиальной игры с узлами жеребьевки

5.19. Реализуйте алгоритм поиска по ожидаемому минимаксному значению и алгоритм *-альфа-бета-поиска, предложенный Баллардом в [119] (1983), для отсечения бесполезных ветвей в деревьях игры с узлами жеребьевки. Опробуйте их работу на такой игре, как нарды, и оцените эффективность отсечения в *-альфа-бета-поиске.

5.20. Докажите, что после положительной линейной трансформации листовых значений (т.е. трансформации значения x в $ax + b$, где $a > 0$) выбор хода в дереве игры остается неизменным, даже если в нем имеются узлы жеребьевки.

5.21. Рассмотрите описанную ниже процедуру выбора ходов в играх с узлами жеребьевки.

а) Сгенерируйте некоторую последовательность серий бросков костей (скажем, 50) вплоть до подходящей для этого глубины (скажем, 8).

б) Как только результаты бросков костей становятся известны, дерево игры преобразуется в детерминированное. Для каждой последовательности бросков костей найдите решение получившегося детерминированного дерева игры с помощью альфа-бета-поиска.

в) Используйте полученные результаты для оценки значения каждого хода и выберите наилучший.

Будет ли эта процедура работать правильно? Почему да (или почему нет)?

5.22. Ниже полагается, что дерево "*max*" состоит только из узлов MAX, тогда как дерево "*expectimax*" состоит из узла MAX в корне с чередующимися ниже слоями узлов жеребьевки и узлов MAX. В узлах жеребьевки вероятности всех исходов отличны от нуля. Цель состоит в том, чтобы *найти значение корня* с помощью поиска с заданным пределом глубины. Для каждого из пп. *а–е* приведите соответствующий пример или объясните, почему это невозможно.

а) Если предположить, что значения листовых узлов конечны, но не ограничены, возможно ли когда-либо в дереве *max* отсечение (как при альфа-бета-отсечении)?

б) При тех же условиях возможно ли когда-либо отсечение в дереве *expectimax*?

в) Если все листовые значения неотрицательны, возможно ли когда-либо отсечение в дереве *max*? Приведите пример или объясните, почему нет.

г) Если все листовые значения неотрицательны, возможно ли когда-либо отсечение в дереве *expectimax*? Приведите пример или объясните, почему нет.

д) Если все листовые значения находятся в диапазоне [0,1], возможно ли когда-либо отсечение в дереве *max*? Приведите пример или объясните, почему нет.

е) Если все листовые значения находятся в диапазоне [0,1], возможно ли когда-либо отсечение в дереве *expectimax*?

ж) Рассмотрим результаты узла жеребьевки в дереве *expectimax*. Какой из следующих порядков оценки, вероятнее всего, предоставит возможность выполнить отсечение?

- Первым — узел с наименьшей вероятностью и дальше в порядке возрастания вероятности.
- Первым — узел с наибольшей вероятностью и дальше в порядке уменьшения вероятности.
- Вероятность не имеет никакого значения

5.23. Ниже полагается, что дерево "*max*" состоит только из узлов MAX, тогда как дерево "*expectimax*" состоит из узла MAX в корне с чередующимися ниже слоями узлов жеребьевки и узлов MAX. В узлах жеребьевки вероятности всех исходов отличны от нуля. Цель состоит в том, чтобы *найти значение корня* с помощью поиска с заданным пределом глубины.

а) Если предположить, что значения листовых узлов конечны, но не ограничены, возможно ли когда-либо в дереве *max* отсечение (как при альфа-бета-отсечении)? Приведите пример или объясните, почему нет.

б) При тех же условиях возможно ли когда-либо отсечение в дереве *expectimax*? Приведите пример или объясните, почему нет.

в) Если все листовые значения находятся в диапазоне [0,1], возможно ли когда-либо отсечение в дереве *max*? Приведите пример или объясните, почему нет.

г) Если все листовые значения находятся в диапазоне [0,1], возможно ли когда-либо отсечение в дереве *expectimax*? Приведите пример (качественно отличающийся от вашего примера в следующем пункте, если таковой есть) или объясните, почему нет.

д) Если все листовые значения неотрицательны, возможно ли когда-либо отсечение в дереве *max*? Приведите пример или объясните, почему нет.

е) Если все листовые значения неотрицательны, возможно ли когда-либо отсечение в дереве *expectimax*? Приведите пример или объясните, почему нет.

ж) Рассмотрим результаты узла жеребьевки в дереве *expectimax*. Какой из следующих порядков оценки, вероятнее всего, предоставит возможность выполнить отсечение?
- Первым — узел с наименьшей вероятностью и дальше в порядке возрастания вероятности.
- Первым — узел с наибольшей вероятностью и дальше в порядке уменьшения вероятности.
- Вероятность не имеет никакого значения

5.24. Какие из следующих утверждений истинны, а какие ложны? Дайте краткое объяснение.

а) В полностью наблюдаемой игре с чередованием ходов и нулевым итогом между двумя совершенно рациональными игроками первому игроку не поможет знание того, какую стратегию использует второй игрок, т.е. какой ход сделает второй игрок после хода первого игрока.

б) В частично наблюдаемой игре с чередованием ходов и нулевым итогом между двумя совершенно рациональными игроками первому игроку не поможет знание того, какой ход сделает второй игрок после определенного хода первого игрока.

в) Совершенно рациональный агент для игры в нарды никогда не проигрывает.

5.25. Тщательно продумайте взаимодействие случайных событий и частичной информации в каждой из игр в упражнении 5.4.

а) Для каких игр подходит стандартная модель ожидаемого минимаксного значения? Реализуйте алгоритм и запустите его в своем игровом агенте с соответствующими изменениями в игровой среде.

б) Для какой игры подойдет схема, описанная в упражнении 5.21?

в) Обсудите, как можно справиться с тем фактом, что в некоторых играх игроки не имеют одинаковых знаний о текущем состоянии.

ГЛАВА 6

Задачи удовлетворения ограничений

В этой главе показано, каким образом, трактуя состояния как нечто большее, чем просто маленькие "черные ящики", можно прийти к новым методам поиска и более глубокому пониманию структуры задачи.

В основу глав 3 и 4 была положена идея, что задачи можно решать, выполняя поиск в пространстве состояний, представленном в виде графа, в котором узлы соответствуют состояниям, а соединяющие их ребра — действиям. Было показано, что для оценки стоимости достижения цели из данного состояния можно использовать эвристические функции, специфические для конкретной проблемной среды. С точки зрения таких алгоритмов поиска каждое состояние является *атомарным*, неделимым — черный ящик без какой-либо внутренней структуры. В результате для каждой задачи требовался код, специфический для ее проблемной среды, предназначенный для описания переходов между состояниями.

В этой главе черные ящики будут раскрыты и их содержимое будет представлено явным образом с использованием **развернутого представления** каждого состояния: в виде набора **переменных**, каждая из которых имеет некоторое **значение**. Задача будет решена, когда каждая переменная будет иметь значение, удовлетворяющее всем наложенным на нее ограничениям. Задачи, описанные подобным образом, называют ▶**задачами удовлетворения ограничений** (УО) или **CSP** — *Constraint Satisfaction Problem*.

Алгоритмы поиска задач удовлетворения ограничений используют преимущество знания структуры состояний, позволяющее применять общие, а не специфические для проблемной среды эвристики, что дает возможность решать сложные задачи. Основная идея состоит в одновременном исключении многих больших частей пространства поиска посредством выявления комбинаций "переменная/значение", нарушающих заданные ограничения. Задачи УО имеют дополнительное преимущество в том, что действия и модель перехода могут быть выведены непосредственно из описания задачи.

6.1. Определение задач удовлетворения ограничений

Определение задачи удовлетворения ограничений включает три компонента: $\mathcal{X}$, $\mathcal{D}$ и $\mathcal{C}$:

$\mathcal{X}$ — множество переменных, $\{X_1, \ldots, X_n\}$;

$\mathcal{D}$ — множество областей определения, $\{D_1, \ldots, D_n\}$, по одной для каждой переменной;

$\mathcal{C}$ — множество ограничений, определяющих допустимые комбинации значений.

Область определения (или *домен*) $\mathcal{D}_i$ состоит из множества допустимых значений $\{v_1, \ldots, v_k\}$ для переменной X_i. Например, булева переменная будет иметь область определения $\{true, false\}$. Различные переменные могут иметь разные области определения разных размеров. Каждое ограничение $\mathcal{C}_j$ состоит из пары $\langle scope, rel\rangle$, где *scope* является некоторым подмножеством (или *кортежем*) переменных, которые принимают участие в ограничении, а *rel* является таблицей (или ▶**отношением**), определяющей, какие значения могут принимать эти переменные. Отношение может быть представлено как явно заданное множество всех кортежей значений, которые удовлетворяют ограничению, или как функция, вычислив которую, можно установить, является ли кортеж членом отношения. Например, если две переменные, X_1 и X_2, имеют одинаковую область определения $\{1, 2, 3\}$, то ограничение, требующее, чтобы значение переменной X_1 всегда было больше значения переменной X_2, можно записать как $\langle(X_1, X_2), \{(3, 1), (3, 2), (2, 1)\}\rangle$ или как $\langle(X_1, X_2), X_1 > X_2)\rangle$.

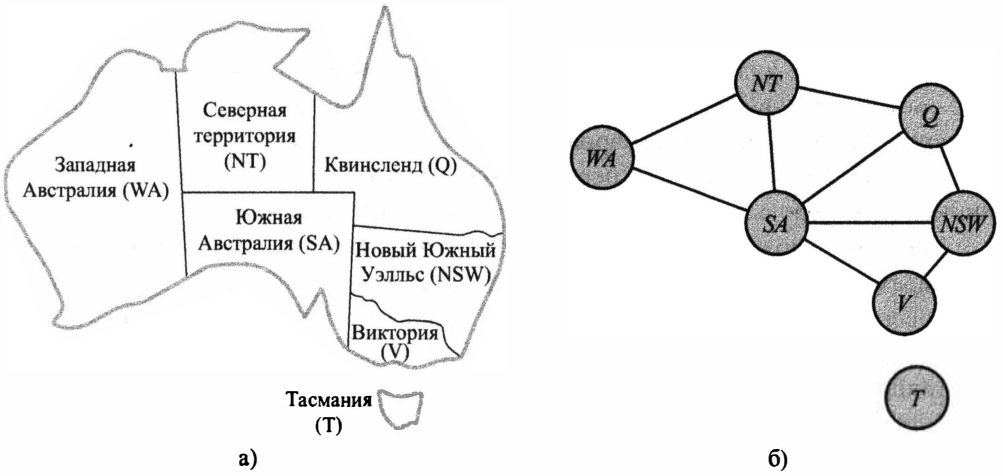
Задачи УО имеют дело с ▶**присваиваниями** значений переменным, $\{X_i = v_i, X_j = v_j, \ldots\}$. Присваивание, которое не нарушает никаких ограничений, называется ▶**совместимым** или допустимым присваиванием. ▶**Полным** называется такое присваивание, в котором значение получает каждая переменная, а **решением** задачи УО является полное присваивание, удовлетворяющее всем ограничениям. В ▶**частичном присваивании** одна или более переменных не имеют значения, а ▶**частичное решение** является частичным присваиванием, которое является совместимым. В общем случае поиск решения CSP является NP-полной задачей, хотя существуют важные подклассы задач удовлетворения ограничений, которые могут быть решены весьма эффективно.

6.1.1. Пример задачи: раскрашивание карты

Предположим, что устав от Румынии, мы обратились к карте Австралии, на которой показаны все ее штаты и территории (рис. 6.1, *а*). Наше задание состоит в том, чтобы раскрасить каждый регион на карте в красный, зеленый или синий цвет

таким образом, чтобы ни одна пара соседних регионов не имела одинакового цвета. Чтобы определить это задание в виде задачи УО, в качестве переменных введем сокращенные названия всех имеющихся на карте регионов:

$\mathcal{X} = \{WA, NT, Q, NSW, V, SA, T\}$.



а) б)

Рис. 6.1. Основные штаты и территории Австралии. **а)** Раскрашивание этой карты может рассматриваться как задача удовлетворения ограничений (УО). Задание состоит в том, чтобы назначить цвет каждому региону таким образом, чтобы ни одна пара соседних регионов не имела одинакового цвета. **б)** Задача раскрашивания карты, представленная в виде графа ограничений

Область определения каждой переменной будет представлять собой множество из трех наименований цвета: $D_i = \{red, green, blue\}$, т.е. *красный*, *зеленый*, *синий*. Ограничения требуют, чтобы все пары соседних регионов имели разные цвета. Поскольку на карте девять различных участков границ, соответствующих разным парам соседствующих территорий, необходимо ввести девять ограничений:

$$C = \{SA \neq WA, SA \neq NT, SA \neq Q, SA \neq NSW, SA \neq V,$$
$$WA \neq NT, NT \neq Q, Q \neq NSW, NSW \neq V\}.$$

В этой записи были использованы определенные сокращения, например $SA \neq WA$ является сокращением для $\langle (SA, WA), SA \neq WA \rangle$, где ограничение $SA \neq WA$, в свою очередь, может быть полностью перечислено как

$$\{(red,green), (red,blue), (green,red), (green,blue), (blue,red), (blue,green)\}.$$

Существует много возможных решений этой задачи, например такое:

$$\{WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = red\}.$$

Задачу УО можно представить и визуально, в виде ▶ **графа ограничений**, как показано на рис. 6.1, *б*. Вершины этого графа соответствуют переменным задачи, а ребра — ограничениям.

Зачем формулировать задачу как CSP? Одна из причин заключается в том, что задачи УО позволяют естественным образом представить широкой спектр задач и часто их очень легко сформулировать как CSP. Другая состоит в том, многолетние исследования в этой области позволили отыскать способы решать задачи УО быстро и эффективно. Третья причина в том, что методы CSP позволяют быстро обрезать большие участки пространства поиска, чего невозможно достичь, работая с состояниями в атомарном представлении. Например, в задаче раскрашивания карты Австралии, выбрав вариант $\{SA = blue\}$, можно сразу сделать вывод, что в этом случае ни одна из пяти переменных, представляющих соседние территории, не может иметь значения *blue*. Процедура поиска, в которой не используются ограничения, должна рассмотреть $3^5 = 243$ варианта присвоения значений этим пяти переменным, тогда как с учетом установленных ограничений достаточно будет проверить только $2^5 = 32$ варианта, — сокращение объема работы на 87%.

При поиске в пространстве состояний с атомарным представлением можно задать только один вопрос: является ли это конкретное состояние целевым? Нет? А вот это? В задачах УО, как только будет установлено, что некоторое частичное присваивание нарушает ограничение, можно немедленно отказаться от дальнейшей его обработки. Более того, можно даже будет понять, *почему* это присваивание не является решением — известно, какие переменные нарушают ограничение, — и поэтому далее сосредоточить внимание только на значимых переменных. В результате многие задачи, неразрешимые в случае поиска в пространстве атомарных состояний, могут быть быстро решены, если сформулировать их как задачи удовлетворения ограничений.

6.1.2. Пример задачи: планирование работы цеха

Заводы сталкиваются с серьезными проблемами в планировании производственных процессов, поскольку это связано с учетом множества ограничений. На практике многие из подобных проблем могут эффективно решаться благодаря использованию методов CSP. Рассмотрим задачу планирования процесса сборки автомобиля. Представим весь процесс как серию заданий, каждое из которых в нашей модели будет представлено отдельной переменной, значением которой будет время начала выполнения этого задания, выраженное в виде целого числа минут, прошедших от начала сборки. Ограничения могут указывать, что выполнение одного задания должно завершиться до начала выполнения другого, например колесо должно быть установлено до того, как на него можно будет надеть колпак, или же определять, что лишь столько-то заданий может выполняться одновременно. Ограничения могут также задавать продолжительность выполнения заданий.

В этом примере рассматривается небольшая часть полного процесса сборки автомобиля, состоящая из 15 заданий: установка осей (передней и задней), закрепление на них всех четырех колес (правых и левых, передних и задних), затягивание гаек на каждом колесе, установка четырех колпаков и финальный осмотр результатов сборки. Отдельные задания будут представлены 15 переменными (здесь *axle* — ось, *wheel* — колесо, *nuts* — гайки, *cap* — колпак, *inspect* — осмотр, *F* — передний, *B* — задний, *R* — правый, *L* — левый):

$$\mathcal{X} = \{Axle_F, Axle_B, Wheel_{RF}, Wheel_{LF}, Wheel_{RB}, Wheel_{LB}, Nuts_{RF}, Nuts_{LF}, Nuts_{RB}, Nuts_{LB}, Cap_{RF}, Cap_{LF}, Cap_{RB}, Cap_{LB}, Inspect\}.$$

Далее, представляем ►**ограничения предшествования** между отдельными заданиями. Всякий раз, когда задание T_1 должно быть выполнено до начала задания T_2 и задание T_1 имеет продолжительность d_1, вводится дополнительное арифметическое ограничение вида

$$T_1 + d_1 \leq T_2.$$

В нашем примере перед установкой колес оси должны быть уже на месте, при этом установка оси занимает 10 минут, поэтому можно записать так:

$$Axle_F + 10 \leq Wheel_{RF}; \quad Axle_F + 10 \leq Wheel_{LF};$$
$$Axle_B + 10 \leq Wheel_{RB}; \quad Axle_B + 10 \leq Wheel_{LB}.$$

Далее принимается такая последовательность действий: необходимо каждое колесо закрепить на оси (что требует 1 минуты), затем затянуть крепежные гайки (2 минуты) и, наконец, укрепить на нем колпак (1 минута, но она пока не представлена в следующих выражениях):

$$Wheel_{RF} + 1 \leq Nuts_{RF}; \quad Nuts_{RF} + 2 \leq Cap_{RF};$$
$$Wheel_{LF} + 1 \leq Nuts_{LF}; \quad Nuts_{LF} + 2 \leq Cap_{LF};$$
$$Wheel_{RB} + 1 \leq Nuts_{RB}; \quad Nuts_{RB} + 2 \leq Cap_{RB};$$
$$Wheel_{LB} + 1 \leq Nuts_{LB}; \quad Nuts_{LB} + 2 \leq Cap_{LB}.$$

Предположим, что для установки колес выделено четыре рабочих, но им приходится использовать один общий инструмент, необходимый для установки оси на место. Следовательно, нужно ввести *разделительное* или ►**дизъюнктивное ограничение**, определяющее, что задания $Axle_F$ и $Axle_B$ не должны перекрываться во времени — первым выполняется либо одно, либо другое:

$$(Axle_F + 10 \leq Axle_B) \text{ или } (Axle_B + 10 \leq Axle_F).$$

Это ограничение выглядит более сложным, сочетающим арифметику и логику. Но оно все еще сводится к набору пар значений, которые могут принимать переменные $Axle_F$ и $Axle_B$.

Также необходимо определить, что задание осмотра результатов сборки выполняется в последнюю очередь и занимает 3 минуты. С этой целью для каждой

переменной, кроме *Inspect*, добавляется ограничение вида $X+d_X \leq Inspect$. И наконец, предположим, что имеется еще одно требование: вся процедура сборки должна быть выполнена в пределах 30 минут. Это требование можно учесть, ограничив область определения для всех переменных:

$$D_i = \{0, 1, 2, 3, \ldots, 30\}.$$

Решение данной конкретной задачи можно считать тривиальным, но методы CSP давно и успешно применяются на практике к подобным проблемам планирования работы цехов с тысячами переменных.

6.1.3. Варианты математического представления задач удовлетворения ограничений

В задачах УО простейшего вида используются переменные, имеющие ▶**дискретные** и ▶**конечные области определения**. Именно к такому виду относятся задачи раскрашивания карты и составления графиков с ограничением по времени. Задача с восемью ферзями (см. рис. 4.3) также может рассматриваться как CSP с конечной областью определения, где переменные $Q_1, \ldots, Q_8$ представляют собой позиции каждого ферзя в столбцах от 1 до 8 и каждая переменная имеет область определения {1,2,3,4,5,6,7,8}. Ограничения в этой задаче состоят в том, что никаких два ферзя не могут находиться в одном и том же ряду или диагонали.

Дискретная область определения может быть и ▶**бесконечной**, такой, например, как множество целых чисел или множество всех строк. (Если в задаче составления графика работ не установить ограничение по времени выполнения, то в ней будет бесконечное количество допустимых моментов начала выполнения задания для каждой переменной.) В случае бесконечной области определения вместо явно заданных кортежей значений переменных придется использовать неявные ограничения, подобные $T_1+d_1 \leq T_2$. Существуют специальные алгоритмы поиска решения (которые здесь не обсуждаются) для случая ▶**линейных ограничений**, накладываемых на целочисленные переменные, т.е. ограничений, подобных только что приведенному, в которых каждая переменная появляется только в линейной форме. Можно показать, что не существует алгоритма решения задач общего вида с ▶**нелинейными ограничениями**, накладываемыми на целочисленные переменные, — эта задача является неразрешимой.

В реальном мире обычны задачи удовлетворения ограничений с ▶**непрерывными областями определения**, — изучение таких задач является основной темой в области исследования операций. Например, для планирования наблюдений на космическом телескопе Хаббл требовалась очень точная их привязка по времени; начало и конец каждого наблюдения и каждого маневра представляли собой переменные с непрерывной областью определения, подчиняющиеся всевозможным ограничениям — астрономическим, в отношении очередности, по энергетическим затратам. Самой известной категорией задач удовлетворения ограничений

с непрерывной областью определения являются задачи ►**линейного программирования**, в которых ограничения должны быть представлены линейными равенствами или неравенствами. Задачи линейного программирования могут быть решены за время, полиномиально зависящее от количества переменных. Также проводились исследования в области задач с другими типами ограничений и целевых функций — задач квадратичного программирования, конического программирования второго порядка и т.д. Эти области исследования представляют собой важное направление в прикладной математике.

Помимо исследования типов переменных, которые могут присутствовать в задачах УО, полезно будет заняться также изучением типов ограничений. Простейшим их типом является ►**унарное ограничение**, ограничивающее значение единственной переменной. Например, в задаче раскрашивания карты Австралии может оказаться, что жители штата Южная Австралия очень не любят зеленый цвет, что можно представить унарным ограничением $\langle(SA), SA \neq green\rangle$. (Первоначальная спецификация области определения переменной также может рассматриваться как унарное ограничение.)

►**Бинарное ограничение** связывает между собой две переменные. Например, ограничение $SA \neq NSW$ является бинарным. ►**Бинарной задачей УО** называется задача, в которой имеют место только унарные и бинарные ограничения; она может быть представлена в виде графа ограничений, подобного приведенному на рис. 6.1, *б*.

Можно также определить ограничения высшего порядка. Так, троичное ограничение между переменными (X, Y, Z) , например, может быть определено как $\langle(X, Y, Z), X < Y < Z$ или $X > Y > Z\rangle$.

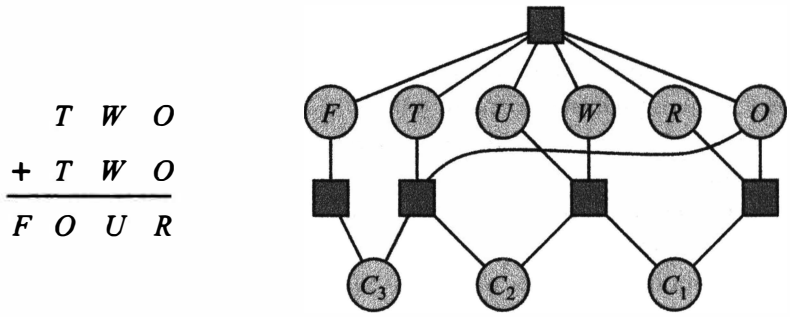
Ограничение, включающее произвольное количество переменных, называется ►**глобальным ограничением**. (Это название является традиционным, но может вводить в заблуждение, поскольку глобальное ограничение не обязательно должно включать *все* переменные в задаче.) Одним из наиболее распространенных глобальных ограничений является *Alldiff*, требующее, чтобы все включенные в него переменные обязательно имели разные значения. В задачах судоку (см. раздел 6.2.6) все переменные в строке, в столбце или блоке 3 × 3 должны удовлетворять ограничению *Alldiff*.

Другим примером таких задач являются ►**криптоарифметические** головоломки, иначе называемые числовыми ребусами (рис. 6.2, *а*). Обычно каждая буква в такой головоломке представляет отдельную цифру. В случае задачи, показанной на рис. 6.2, *а*, такое требование может быть выражено с помощью глобального ограничения $Alldiff(F, T, U, W, R, O)$. Ограничения по правилам сложения для четырех столбцов этой головоломки могут быть записаны как n-арные ограничения в таком виде:

$$O + O = R + 10 \cdot C_1$$
$$C_1 + W + W = U + 10 \cdot C_2$$

$$C_2 + T + T = O + 10 \cdot C_3$$
$$C_3 = F,$$

где C_1, C_2 и C_3 являются вспомогательными переменными, представляющими цифру (0 или 1), которая переносится в следующий столбец десятков, сотен или тысяч. Такие ограничения могут быть представлены в виде ▶**гиперграфа ограничений**, как показано на рис. 6.2, *б*. Гиперграф состоит из обычных вершин (на рисунке они представлены кружками) и гипервершин (квадраты), представляющих *n*-арные ограничения, т.е. ограничения, включающие *n* переменных.



а) б)

Рис. 6.2. а) Пример криптоарифметической задачи: каждая буква обозначает отдельную цифру, и суть задания — найти такую замену букв цифрами, чтобы результирующее выражение суммирования было арифметически правильным, но с дополнительным ограничением (наличие ведущих нулей не допускается). **б)** Гиперграф ограничений для этой криптоарифметической задачи, на котором показано ограничение *Alldiff* (квадрат в верхней части рисунка), а также ограничения сложения по столбцам (четыре квадрата в среднем ряду). Переменные C_1, C_2 и C_3 представляют цифры переноса справа налево для трех столбцов

В качестве альтернативы можно доказать (см. упражнение 6.6), что любое ограничение в конечной области определения может быть сведено к множеству бинарных ограничений, если ввести достаточное количество вспомогательных переменных. Это означает, что любую задачу УО можно трансформировать в задачу, включающую только бинарные ограничения, что, безусловно, делает жизнь разработчиков алгоритмов проще. Другой способ преобразования *n*-арной CSP в бинарную строится на преобразовании ее графа в ▶**двойственный граф**: создается новый граф, в котором будет по одной переменной для каждого ограничения в исходном графе и одно бинарное ограничение для каждой пары ограничений в исходном графе, имеющих общие переменные.

Например, рассмотрим задачу УО с переменными $\mathcal{X}=\{X, Y, Z\}$, каждая из которых имеет область определения {1,2,3,4,5}, и двумя ограничениями C_1: $\langle (X, Y, Z), X+Y=Z\rangle$ и C_2: $\langle (X, Y), X+1=Y\rangle$. Тогда двойственный граф будет иметь переменные $\mathcal{X}=\{C_1, C_2\}$, где область определения переменной C_1 в двойственном графе представляет собой множество кортежей $\{(x_i, y_j, z_k)\}$ в соответствии с ограничением C_1 в исходной задаче, и аналогично область определения переменной C_2 представляет собой множество кортежей $\{(x_i, y_j)\}$ в соответствии с ограничением C_2 в исходной задаче. Двойственный граф имеет бинарное ограничение $\langle (C_1, C_2), R_1\rangle$, где R_1 является новым отношением, определяющим ограничения между переменными C_1 и C_2, — в данном случае оно будет иметь вид $R_1 = \{((1,2,3), (1,2)), ((2,3,5), (2,3))\}$.

Однако имеется две причины, по которым можно предпочесть глобальное ограничение, такое как *Alldiff*, набору бинарных ограничений. Во-первых, будет проще и с меньшей вероятностью совершить ошибку дать описание задачи с помощью ограничения *Alldiff*. Во-вторых, вполне возможно разработать специализированный алгоритм логического вывода для глобальных ограничений, который будет эффективнее манипулирования примитивными ограничениями. Подобные алгоритмы логического вывода обсуждаются в разделе 6.2.5.

Все ограничения, рассматривавшиеся до сих пор, были абсолютными ограничениями, нарушение которых исключает потенциальное решение. Однако многие реальные задачи УО включают ▶**ограничения предпочтения**, указывающие, какие решения являются более предпочтительными. Например, в задаче составления расписания занятий в университете есть абсолютное ограничение, утверждающее, что ни один профессор не может проводить занятия в двух классах одновременно. Но в ней могут присутствовать и ограничения предпочтения, например такое, что профессор R предпочитает проводить занятия по утрам, тогда как профессор N — после полудня. Вариант расписания занятий, в котором профессор R по средам проводит занятия в 14:00, может рассматриваться как допустимое решение (если только он не должен в это же время председательствовать на заседании кафедры), но не будет оптимальным.

Ограничения предпочтения часто можно закодировать как стоимости присваиваний отдельных переменных; например, за назначение профессору R занятий во второй половине дня придется заплатить 2 пункта, которые будут учтены в суммарном значении целевой функции, в то время назначение ему занятий в первой половине дня будет иметь стоимость 1 пункт. При использовании такой формулировки задачи УО с предпочтениями можно решать, используя методы либо поиска с оптимизацией, либо основанные на поиске пути, либо локальные. Подобный тип задач мы будем называть ▶**задачами оптимизации при ограничениях**, одним из важных классов которых являются задачи линейного программирования.

6.2. Распространение ограничений: логический вывод в CSP

Алгоритм поиска в пространстве атомарных состояний продвигается только одним способом: развертывая узел с целью посещения его преемников. У алгоритмов CSP здесь есть выбор. Они могут генерировать преемников, выбирая новое значение переменной с учетом ограничений. Именно этот подход реализуется в особом типе логического вывода, называемом ▶**распространением ограничений**, — здесь ограничения используются для уменьшения количества допустимых значений переменной, что, в свою очередь, может привести к сокращению допустимых значений для другой переменной, и т.д. Идея состоит в том, что такой подход оставляет меньше вариантов, которые потребуется рассмотреть, выполняя следующий выбор при присваивании переменной. Распространение ограничений может быть вплетено непосредственно в процесс поиска или может выполняться как этап предварительной обработки перед началом поиска. Иногда такая предварительная обработка позволяет решить всю задачу, и в результате поиск вообще не потребуется.

Ключевая идея этого метода — ▶**локальная совместимость**. Если каждую переменную рассматривать как вершину в графе (см. рис. 6.1, *б*), а каждое бинарное ограничение — как его дугу, то процесс обеспечения локальной совместимости в каждой части графа приводит к тому, что несовместимые значения удаляются по всему графу. Существуют различные типы локальной совместимости, которые рассматриваются в последующих подразделах.

6.2.1. Вершинная совместимость

Некоторая переменная (соответствующая вершине в графе задачи УО) является ▶**вершинно совместимой**, если все значения в области определения этой переменной удовлетворяют ее унарным ограничениям. Например, в том варианте задачи раскрашивания карты Австралии (см. рис. 6.1), где жители штата Южная Австралия не любят зеленый цвет, переменная *SA*, исходно имеющая область определения {*red*, *green*, *blue*}, станет вершинно совместимой, если удалить из ее области определения значение *green*, оставив переменную *SA* с уменьшенной областью определения {*red*, *blue*}. Говорят, что граф вершинно совместим, если каждая переменная в графе вершинно совместима.

В задаче УО можно легко исключить все унарные ограничения, просто уменьшив в соответствии с ними области определения переменных в самом начале процесса решения задачи. Как было отмечено ранее, также возможно преобразовать все *n*-арные ограничения в бинарные. Именно по этой причине многие программы решения CSP работают только с бинарными ограничениями, возлагая на пользователя задачу предварительного исключения из задачи всех иных типов ограничений. В дальнейшем в этой главе мы также будем придерживаться этого предположения, если не указано обратное.

6.2.2. Дуговая совместимость

В CSP переменная обладает ▶**дуговой совместимостью**,[1] если каждое значение в ее области определения удовлетворяет всем бинарным ограничениям переменной. Более формально переменная X_i обладает дуговой совместимостью (или *дугосовместима*) по отношению к другой переменной X_j, если для каждого значения в текущей области определения D_i имеется некоторое значение в области определения D_j, удовлетворяющее бинарному ограничению по дуге (X_i, X_j). Граф является дугосовместимым, если каждая переменная является дугосовместимой с любой другой переменной. Например, рассмотрим ограничение $Y=X^2$, где область определения обеих переменных X и Y представляет собой множество десятичных цифр. Тогда это ограничение в явном виде можно записать как

$$\langle (X,Y),\{(0, 0),(1, 1),(2, 4),(3, 9)\}\rangle.$$

Чтобы сделать переменную X дугосовместимой с переменной Y, достаточно уменьшить область определения переменной X до $\{0, 1, 2, 3\}$. Если также необходимо, чтобы и переменная Y была дугосовместимой с переменной X, то область определения Y должна быть сокращена до $\{0, 1, 4, 9\}$, и тогда вся эта задача УО станет дугосовместимой. С другой стороны, дуговая совместимость ничем не может нам помочь в решении задачи раскрашивания карты Австралии. Рассмотрим следующие ограничения-неравенства, наложенные на переменные (*SA*, *WA*):

$$\{(red, green),(red, blue),(green, red),(green, blue),(blue, red),(blue, green)\}.$$

Независимо от того, какое значение вы выбираете для *SA* (или для *WA*), для другой переменной есть допустимое значение. В результате применение правила дуговой совместимости не оказывает влияния на область определения любой переменной.

Наиболее популярным алгоритмом обеспечения дугосовместимости является AC-3 (рис. 6.3). Чтобы сделать каждую переменную в задаче дугосовместимой, алгоритм AC-3 использует очередь, содержащую все дуги, которые необходимо проверить. Первоначально создается очередь, включающая все дуги в данной задаче. (Каждое бинарное ограничение представляется двумя дугами, по одной в каждом направлении.) Затем алгоритм AC-3 извлекает из очереди произвольно выбранную дугу (X_i, X_j) и проверяет переменную X_i на дугосовместимость с переменной X_j. Если это требование соблюдается, то алгоритм просто переходит к следующей дуге в очереди. Если это не так, то область определения D_i корректируется (уменьшается посредством отбрасывания нарушающего совместимость значения), а затем в очередь добавляются все дуги (X_k, X_i), где вершины X_k являются соседями X_i. Это необходимо сделать, поскольку сокращение области D_i может вызвать дальнейшие сокращения в областях D_k, даже если дуги от X_k уже были

[1] Выше в этой главе использовался термин "ребро", а не "дуга", поэтому было бы более разумно называть это "реберной совместимостью", но именно название "дуговая совместимость" является исторически сложившимся.

рассмотрены ранее. Если область определения D_i при очередном уменьшении оказывается пустой, то это означает, что вся задача УО не имеет совместимого решения и алгоритм AC-3 немедленно возвращает значение *false*. В противном случае проверка дуг продолжается (при необходимости с удалением значений из областей определения переменных) до тех пор, пока очередь не опустеет. На этот момент у нас есть определение задачи УО, *эквивалентной* исходной задаче, — они обе будут иметь одно и то же решение, — но поиск в дугосовместимом варианте будет выполняться быстрее, поскольку его переменные имеют меньшие области определения. В одних случаях можно даже получить готовое решение (когда область определения каждой переменной содержит лишь одно значение), а в других будет доказано, что решения не существует (размер некоторой области определения уменьшился до нуля).

```
function AC-3(csp) returns false если обнаружена дуговая несовместимость
                                  или определение исходной задачи, возможно,
                                  с меньшими областями определения переменных
  queue ← очередь дуг, первоначально содержит все дуги в csp

  while queue не пуста do
  (X_i, X_j) ← POP(queue)
  if REVISE(csp, X_i, X_j) then
    if размер области определения D_i = 0 then return false
    for each X_k in X_i.NEIGHBORS – {X_j} do
      добавить (X_k, X_i) к очереди queue
  return true

function REVISE(csp, X_i, X_j) returns true если область определения X_i
                                  успешно прошла проверку
  revised ← false
  for each x in D_i do
    if в D_j нет значения y, обеспечивающего для (x, y) удовлетворение
          ограничения между X_i и X_j then
    удалить x из D_i
    revised ← true
  return revised
```

Рис. 6.3. Алгоритм обеспечения дуговой совместимости AC-3. После применения этого алгоритма либо каждая дуга будет дугосовместимой, либо некоторая переменная будет иметь пустую область определения, указывая на то, что эта задача УО не имеет решения. Название "AC-3" было предложено разработчиком данного алгоритма ([1474], 1977), поскольку это третья версия, представленная в его статье

Сложность алгоритма AC-3 можно проанализировать следующим образом. Предположим, что задача УО содержит n переменных, каждая из которых имеет размер области определения не более d с бинарными ограничениями (дугами). Каждая дуга (X_k, X_i) может быть вставлена в очередь только d раз, поскольку переменная X_i имеет не более d значений, которые можно было бы удалить. Проверка дуговой совместимости может быть выполнена за время $O(d^2)$, поэтому для наихудшего случая общее время выполнения можно оценить как $O(cd^3)$.

6.2.3. Путевая совместимость

Предположим, нам нужно раскрасить карту Австралии всего двумя цветами: красным и синим. Дуговая совместимость в этом случае нам ничем не поможет, поскольку каждое ограничение по отдельности легко удовлетворяется простым размещением красного цвета на одном конце и синего — на другом. Однако очевидно, что в действительности решения задачи не существует, поскольку три штата (Западная Австралия, Северные территории и Южная Австралия) граничат друг с другом и необходимо по крайней мере три цвета, чтобы каждый получил свой.

Дуговая совместимость ограничивает области определения переменных (унарные ограничения), используя дуги (бинарные ограничения). Чтобы добиться прогресса в таких задачах, как раскрашивание карты, необходимо использовать более строгое определение понятия совместимости. ▶ **Путевая совместимость** дополнительно ужесточает бинарные ограничения за счет использования неявных ограничений, которые выводятся в результате анализа троек переменных.

Множество двух переменных $\{X_i, X_j\}$ будет *путесовместимо* по отношению к третьей переменной X_m, если для каждого присваивания $\{X_i = a, X_j = b\}$, совместимого с ограничениями (если таковые имеются) на значения $\{X_i, X_j\}$, существует присваивание X_m, удовлетворяющее ограничениям на $\{X_i, X_m\}$ и $\{X_m, X_j\}$. Название этой характеристики определяется требованием общей совместимости дуг на пути от X_i до X_j с X_m в середине.

Давайте посмотрим, можно ли достичь путесовместимости в задаче раскрашивания карты Австралии двумя цветами. Сделаем множество $\{WA, SA\}$ путесовместимым относительно переменной NT. Начнем с перечисления совместимых присваиваний для множества. В этой задаче их имеется только два: $\{WA = red, SA = blue\}$ и $\{WA = blue, SA = red\}$. Очевидно, что для обоих этих присваиваний значение переменной NT не может быть ни *red*, ни *blue*, поскольку они будут конфликтовать либо со значением переменной WA, либо со значением переменной SA. Так как допустимого варианта значения переменной NT не найдено, оба присваивания исключаются из числа допустимых, а значит, допустимых присваиваний для множества $\{WA, SA\}$ у нас нет. Следовательно, можно сделать обоснованный вывод, что эта задача решения не имеет.

6.2.4. *k*-совместимость

Более строгие формы распространения ограничения можно определить с помощью понятия ▶***k*-совместимости**. Задача УО является *k*-совместимой, если для любого множества $k-1$ переменных и для любого совместимого присваивания значений этим переменным любой *k*-й переменной всегда можно присвоить некоторое совместимое значение. Например, 1-совместимость означает, что при заданном пустом множестве можно считать любое множество из одной переменной совместимым с ним, — это понятие называют также вершинной совместимостью. Далее, 2-совместимость — то же, что и дуговая совместимость. Для графов бинарных ограничений 3-совместимость — это то же, что и путевая совместимость.

Задача УО является ▶**строго *k*-совместимой**, если она является *k*-совместимой, а также $(k-1)$-совместимой, $(k-2)$-совместимой и так далее вплоть до 1-совместимой. Теперь предположим, что имеется некоторая задача УО с n узлами, которую мы сделали строго n-совместимой (т.е. строго *k*-совместимой для $k=n$). Тогда эту задачу можно решить следующим образом. Вначале выбираем совместимое значение для X_1. В таком случае существует гарантия, что мы сможем выбрать значение для X_2, поскольку граф является 2-совместимым, для X_3, поскольку он 3-совместимый, и т.д. Для каждой переменной X_i необходимо выполнить поиск только среди d значений в ее области определения, чтобы найти значение, совместимое с $X_1,\cdots,$ X_{i-1}. Это означает, что гарантируется нахождение решения за время $O(n^2d)$.

Безусловно, бесплатных пирожков не бывает и за такую возможность также приходится платить: задача удовлетворения ограничений в общем случае является NP-полной и любой алгоритм обеспечения n-совместимости в наихудшем случае должен требовать времени, экспоненциально зависящего от n. Еще хуже то, что достижение n-совместимости также требует дополнительного объема памяти, экспоненциально зависящего от n. На практике определение подходящего уровня проверки совместимости в основном относится к области эмпирических методов. Вычисление 2-совместимости встречается достаточно часто, а вот вычисление 3-совместимости распространено значительно меньше.

6.2.5. Глобальные ограничения

Вспомним, что **глобальным ограничением** называют ограничение, в котором используется произвольное количество переменных (но не обязательно все). Глобальные ограничения часто встречаются в реальных задачах и могут быть обработаны с помощью специальных алгоритмов, которые более эффективны, чем универсальные методы, обсуждавшиеся до сих пор. Например, ограничение *Alldiff* указывает, что все задействованные в нем переменные должны иметь разные значения (как в криптоарифметической задаче выше или головоломке судоку, речь о которой пойдет ниже). Одна простая форма обнаружения несоответствия для ограничений *Alldiff* работает следующим образом: если ограничение включает m

переменных и если для всех них имеется n возможных разных значений, то при $m > n$ это ограничение не может быть выполнено.

Отсюда можно вывести следующий простой алгоритм. Сначала удаляем из рассмотрения любую переменную в ограничении, имеющую одноэлементную область определения, а затем удаляем значение этой переменной из областей определения оставшихся переменных. Далее повторяем эту процедуру до тех пор, пока имеются переменные с одноэлементными областями определения. Если в какой-то момент времени будет обнаружена пустая область определения или останется больше переменных, чем областей определения, это означает, что имеет место несовместимость.

Этот метод можно применить для обнаружения несовместимости в частичном присваивании $\{WA = red, NSW = red\}$ на рис. 6.1. Обратите внимание на то, что переменные SA, NT и Q фактически связаны ограничением *Alldiff*, поскольку каждая пара соответствующих регионов должна быть раскрашена различными цветами. После применения алгоритма AC-3 в сочетании с этим частичным присваиванием область определения каждой переменной сокращается до $\{green, blue\}$. Это означает, что имеются три переменные и только два цвета, поэтому ограничение *Alldiff* нарушается. Таким образом, иногда простая процедура проверки совместимости для ограничения более высокого порядка оказывается эффективнее в сравнении с процедурой проверки совместимости дуг, применяемой к эквивалентному множеству бинарных ограничений.

Возможно, более важным ограничением высокого порядка является ▶ **ресурсное ограничение**, иногда называемое ограничением *atmost* (*самое большее*). Например, допустим, что в задаче составления графика переменные $P_1, \ldots, P_4$ обозначают количество персонала, назначенного для выполнения каждого из четырех заданий. Ограничение, согласно которому всего к работам может быть привлечено не более 10 человек, записывается как $atmost(10, P_1, P_2, P_3, P_4)$. Несовместимость в этом случае можно обнаружить, проверяя сумму минимальных значений текущих областей определения переменных P_n. Например, если каждая переменная имеет область определения {3,4,5,6}, то ограничение *atmost* не может быть удовлетворено. Кроме того, можно принудительно добиться совместимости, удаляя максимальное значение из любой области определения, если оно не совместимо с минимальными значениями в других областях определения. Следовательно, если каждая переменная в данном примере исходно будет иметь область определения {2,3,4,5,6}, то из каждой области определения можно сразу же удалить значения 5 и 6.

При решении крупных задач проверки ресурсных ограничений с целочисленными значениями (таких, как задачи снабжения, в которых предусматривается перемещение тысяч людей в сотнях транспортных средств) обычно нецелесообразно представить область определения каждой переменной в виде большого множества целых чисел, а затем постепенно сокращать это множество с применением методов проверки совместимости. Вместо этого области определения представляются в виде верхнего и нижнего пределов и согласуются с помощью метода ▶ **распространения**

пределов. Например, в задаче составления графика авиарейсов предположим, что имеются два рейса, F_1 и F_2, которые выполняют самолеты вместимостью 165 и 385 пассажиров соответственно. Тогда исходно области определения количества пассажиров в каждом рейсе можно описать следующим образом:

$$D_1 = [0, 165] \quad \text{и} \quad D_2 = [0, 385].$$

Теперь предположим, что имеется дополнительное ограничение, согласно которому этими двумя рейсами необходимо перевести 420 человек: $F_1 + F_2 = 420$. Воспользовавшись методом распространения пределов, можно сократить области определения для каждой переменной:

$$D_1 = [35, 165] \quad \text{и} \quad D_2 = [255, 385].$$

Задача УО называется ▶**совместимой с пределами** (*bounds-consistent*), если для каждой переменной X и одновременно для нижнего и верхнего предельных значений X существует некоторое значение переменной Y, удовлетворяющее заданному ограничению между X и Y для каждой переменной Y. Такого рода распространение пределов широко используется в практических задачах с ограничениями.

6.2.6. Головоломка судоку

Популярная головоломка ▶**судоку** познакомила миллионы людей с проблемой удовлетворения ограничений, хотя они, возможно, этого и не осознавали. Классическая головоломка судоку представляет собой квадратное игровое поле, разделенное на 81 квадрат, 9 рядов × 9 столбцов, часть из которых первоначально заполнены цифрами от 1 до 9. Задача состоит в том, чтобы поместить во все свободные квадраты цифры от 1 до 9 таким образом, чтобы ни одна цифра не появлялась дважды в любой строке, любом столбце или любом из вложенных малых квадратов 3×3 игрового поля (рис. 6.4). В этой головоломке строку, столбец или меньший квадрат 3×3 принято называть ▶**блоком**.

Головоломки судоку, которые печатают в газетах или сборниках головоломок, обладают тем свойством, что для них существует ровно одно решение. Хотя некоторые из них могут быть достаточно сложными, чтобы на их решение вручную уходили часы, любой решатель CSP способен обрабатывать тысячи головоломок в секунду.

Головоломку судоку можно считать задачей УО с 81 переменной, по одной на каждый квадрат. Здесь мы будем именовать эти переменные как *A1 – A9* для верхнего ряда (слева направо) и вниз до *I1 – I9* для нижнего ряда. Все пустые квадраты имеют область определения {1, 2, 3, 4, 5, 6, 7, 8, 9}, а исходно заполненные квадраты имеют область определения, состоящую из единственного значения, которое в них указано. Кроме того, в задаче есть 27 различных ограничений *Alldiff*, по одному для каждого блока (т.е. для каждой строки, столбца и поля из 9 квадратов).

Alldiff(*A1*, *A2*, *A3*, *A4*, *A5*, *A6*, *A7*, *A8*, *A9*)
Alldiff(*B1*, *B2*, *B3*, *B4*, *B5*, *B6*, *B7*, *B8*, *B9*)
...
Alldiff(*A1*, *B1*,*C1*, *D1*, *E1*, *F1*, *G1*, *H1*, *I1*)
Alldiff(*A2*, *B2*,*C2*, *D2*, *E2*, *F2*, *G2*, *H2*, *I2*)
...
Alldiff(*A1*, *A2*, *A3*, *B1*, *B2*, *B3*,*C1*,*C2*,*C3*)
Alldiff(*A4*, *A5*, *A6*, *B4*, *B5*, *B6*,*C4*,*C5*,*C6*)
...

	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

а)

	1	2	3	4	5	6	7	8	9
A	4	8	3	9	2	1	6	5	7
B	9	6	7	3	4	5	8	2	1
C	2	5	1	8	7	6	4	9	3
D	5	4	8	1	3	2	9	7	6
E	7	2	9	5	6	4	1	3	8
F	1	3	6	7	9	8	2	4	5
G	3	7	2	6	8	9	5	1	4
H	8	1	4	2	5	3	7	6	9
I	6	9	5	4	1	7	3	8	2

б)

Рис. 6.4. а) Пример головоломки судоку. **б)** Решение этого примера

Давайте посмотрим, насколько нам может помочь в решении задачи метод обеспечения дуговой совместимости. Предположим, что ограничения *Alldiff* были расширены до бинарных ограничений (например, таких, как $A1 \neq A2$), и теперь мы можем прямо применить к задаче алгоритм AC-3. Рассмотрим переменную *E6* (см. рис. 6.4, *а*) — это пустой квадрат, расположенный между значениями 2 и 8 в центральном квадратном блоке 3×3. Исходя из ограничений для этого блока, из области определения переменной *E6* можно удалить значения 1, 2, 7 и 8. В соответствии с ограничениями ее столбца из области определения также следует исключить значения 5, 6, 2, 8, 9 и 3 (хотя значения 2 и 8 уже были удалены). В результате в области определения переменной *E6* остается единственное значение {*4*}; иначе говоря, получен ответ для этой переменной (он не противоречит и существующим ограничениям ее строки). Теперь рассмотрим переменную *I6* — квадрат в нижнем блоке 3×3 среднего столбца, — окруженный значени-

ями 1, 3 и 3. Применение метода дуговой совместимости для столбца позволяет удалить из области определения *I6* значения 5, 6, 2, 4 (уже известно, что значением переменной *E6* должно быть 4), 8, 9 и 3. Значение 1 следует исключить по дуговой совместимости с соседней переменной *I5*, и в результате в области определения переменной *I6* остается единственное значение 7. Теперь в столбце 6 есть восемь известных значений, поэтому в соответствии с требованиями дуговой совместимости переменная *A6* должна иметь значение 1. Придерживаясь той же методики вывода, алгоритм AC-3 в конечном счете позволит решить всю головоломку — области определения всех переменных будут сведены к единственному значению, как показано на рис. 6.4, *б*.

Конечно, интерес к головоломке судоку был бы утрачен очень быстро, если бы любой ее экземпляр можно было решить посредством механического применения алгоритма AC-3. В действительности этот алгоритм работает только для самых простых вариантов головоломки судоку. Алгоритм PC-2, работающий по методу путевой совместимости, позволяет решить более сложные варианты задачи, но с бо́льшими вычислительными затратами: в головоломке судоку нужно учитывать 255 960 различных ограничений пути. Однако, чтобы решить самые сложные экземпляры головоломки и добиться прогресса в эффективности, необходимо действовать умнее.

На самом деле привлекательность головоломки судоку для решающего ее человека заключается, прежде всего, в необходимости искать более сложные стратегии логического вывода. Поклонники дают им красочные имена, такие как "голые тройки". Эта стратегия работает следующим образом: в любом блоке (строка, столбец или квадрат 3×3) ищем три квадрата, у каждого из которых область определения состоит из трех одинаковых чисел или подмножества этих чисел. Например, тремя такими областями определения могут быть $\{1, 8\}$, $\{3, 8\}$ и $\{1, 3, 8\}$. Этой информации недостаточно, чтобы установить, какой именно квадрат содержит 1, 3 или 8, но она позволяет утверждать, что эти три числа должны быть распределены между данными тремя квадратами. А значит, можно удалить 1, 3 и 8 из областей определения всех *остальных* квадратов в блоке.

Интересно выяснить, как далеко можно зайти, не используя ничего из того, что является специфическим именно для судоку. Безусловно, следует сказать, что в задаче имеется 81 переменная, области определения которых могут включать любое подмножество цифр от 1 до 9, а также что присутствует 27 ограничений типа *Alldiff*. Однако за рамками этого определения все использованные нами стратегии — дуговая совместимость, путевая совместимость и так далее — в общем случае применимы ко всем задачам удовлетворения ограничений, а не только к судоку. Даже метод "голых троек" в действительности является стратегией для обеспечения совместимости ограничений *Alldiff* и отнюдь не является специфическим для судоку как такового. Это сила формального подхода к CSP: для каждой новой проблемной области достаточно лишь определить задачу в терминах ограничений, а все остальное возьмут на себя общие механизмы решения задач УО.

6.3. Поиск с возвратами в задачах удовлетворения ограничений

Иногда может оказаться, что процесс распространения ограничений уже завершился, а в задаче по-прежнему имеются переменные с несколькими возможными значениями. В этом случае для нахождения окончательного решения необходимо выполнить **поиск**. В данном разделе рассматриваются алгоритмы поиска с возвратами, работающие на частичных присваиваниях, в следующем — алгоритмы локального поиска по полным присваиваниям.

Рассмотрим, как стандартный поиск с ограничением глубины (из главы 3) можно применить для решения задач УО. Здесь состоянием будет частичное присваивание, а действием — расширение присваивания за счет добавления нового значения, например $NSW = red$ или $SA = blue$ для задачи раскрашивания карты Австралии. Для задач УО с n переменными с областью определения размером d в конечном счете может быть построено дерево поиска, в котором все полные присваивания (а следовательно, и все решения) являются листовыми узлами на глубине n. При этом отметим, что фактор ветвления на верхнем уровне должен быть nd, поскольку любое из d значений может быть присвоено любой из n переменных. На следующем уровне коэффициент ветвления будет равен $(n-1)d$ и так далее для всех n уровней. В результате дерево будет иметь $n! \cdot d^n$ листовых узлов, хотя в задаче существует только d^n возможных полных присваиваний!

Однако можно избавиться от этого пугающего коэффициента $n!$, если учесть важное свойство задач УО: ▶ **коммутативность**. Задача является коммутативной, если порядок применения любого заданного множества действий в процессе ее решения не влияет на результат. В задаче УО безразлично, какая переменная получит значение первой: будет это $NSW = red$, а затем $SA = blue$ или наоборот. Следовательно, в каждом узле дерева поиска необходимо рассматривать *только одну* переменную. В корне дерева можно сделать выбор между $SA = red$, $SA = green$ и $SA = blue$, но никогда не следует делать выбор между $NSW = red$ и $SA = blue$. При таком условии количество листьев в дереве сокращается до d^n, к чему мы и стремились. На каждом уровне дерева необходимо выбрать, с какой переменной работать, и никогда не изменять этого выбора.

На рис. 6.5 представлена процедура поиска с возвратами для задач удовлетворения ограничений (CSP). При работе он последовательно выбирает еще не назначенные переменные, а затем по очереди пробует все значения в области определения этой переменной, предпринимая попытки расширить каждое из них до решения посредством рекурсивного вызова. Если вызов завершается успешно, возвращается решение задачи, а если возвращается отказ, присваивание восстанавливается до предыдущего состояния, и проверяется следующее значение. Если все значения приводят к ошибке, алгоритм возвращает отказ. Часть дерева поиска

для задачи раскрашивания карты Австралии показана на рис. 6.6, где значения переменным присваивались в порядке *WA*, *NT*, *Q*...

---

function BACKTRACKING-SEARCH(*csp*) **returns** решение или индикатор отказа *failure*
  return BACKTRACK(*csp*, {})

function BACKTRACK(*csp*, *assignment*) **returns** решение или индикатор отказа *failure*
  if *assignment* является полным **then return** *assignment*
  var ← SELECT-UNASSIGNED-VARIABLE(*csp*, *assignment*)
  for each *value* **in** ORDER-DOMAIN-VALUES(*csp*, *var*, *assignment*) **do**
    if значение *value* является совместимым с присваиванием *assignment* **then**
      добавить {*var* = *value*} к присваиванию *assignment*
      inferences ← INFERENCE(*csp*, *var*, *assignment*)
      if *inferences* ≠ *failure* **then**
        добавить *inferences* к решению *csp*
        result ← BACKTRACK(*csp*, *assignment*)
        if *result* ≠ *failure* **then return** *result*
        удалить *inferences* из решения *csp*
      удалить {*var* = *value*} из присваивания *assignment*
  return *failure*

Рис. 6.5. Простой алгоритм поиска с возвратами для решения задач удовлетворения ограничений. Этот алгоритм составлен по принципу рекурсивного поиска в глубину, описанного в главе 3. Функции SELECT-UNASSIGNED-VARIABLE и ORDER-DOMAIN-VALUES реализуют эвристические функции общего назначения, рассматриваемые в разделе 6.3.1. Функция INFERENCE может на выбор обеспечивать дуговую, путевую или *k*-совместимость, если это необходимо. Если выбор значения приводит к невыполнимости условий (обнаруженной либо функцией INFERENCE, либо функцией BACKTRACK), то присваивания значений (включая те, которые были выполнены функцией INFERENCE) отменяются и проверяется новое значение

Обратите внимание, что функция BACKTRACKING-SEARCH сохраняет только единственное представление состояния (присваивание) и изменяет это представление, а не создает новые (см. раздел 3.4.4).

Хотя алгоритмы неинформированного поиска из главы 3 можно улучшить только за счет использования в них эвристических функций, *соответствующих данной проблемной области*, было установлено, что работу алгоритма поиска с возвратами можно улучшить с помощью *предметно-независимой* эвристики, в которой используются преимущества разложенного представления задачи УО. В следующих четырех разделах будет показано, как это делается.

- (6.3.1) Какой переменной должно быть присвоено значение в следующую очередь (функция SELECT-UNASSIGNED-VARIABLE) и в каком порядке

необходимо проверять ее допустимые значения (функция ORDER-DOMAIN-VALUES)?

- (6.3.2) Какие логические выводы следует выполнять на каждом этапе поиска (функция INFERENCE)?
- (6.3.3) Можно ли вернуться (функция BACKTRACK) больше чем на один этап поиска, если это потребуется?
- (6.3.4) Можно ли сохранить и повторно использовать частичные результаты уже выполненного поиска?

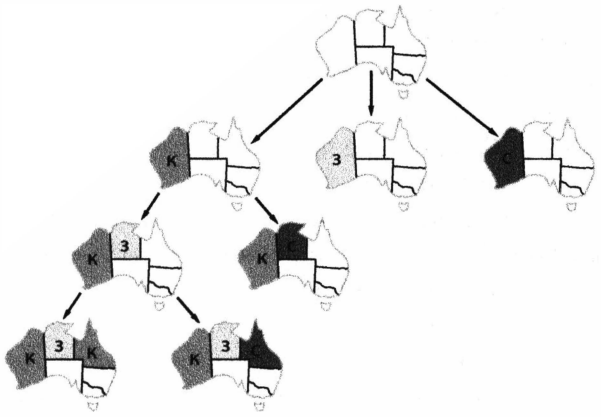


Рис. 6.6. Часть дерева поиска для задачи раскрашивания карты, представленной на рис. 6.1

6.3.1. Упорядочение переменных и значений

Приведенный на рис. 6.5 алгоритм поиска с возвратами содержит следующую строку:

var ← SELECT-UNASSIGNED-VARIABLE(*csp*, *assignment*).

Простейшей стратегией для функции SELECT-UNASSIGNED-VARIABLE является использование заданной статической упорядоченности: переменные выбираются в порядке $\{X_1, X_2, \ldots\}$. Другим простым вариантом является выбор переменных случайным образом. Ни одна из этих стратегий не является оптимальной. Например, на рис. 6.6 после присваивания значений переменным $WA = red$ и $NT = green$ есть только это одно возможное значение для переменной SA, так что имеет смысл на следующем этапе выполнить присваивание $SA = blue$, а не присваивать значение переменной Q. В действительности после присваивания значения переменной SA все варианты выбора значений для переменных Q, NSW и V становятся вынужденными.

Эта довольно очевидная идея выбора в первую очередь переменной с наименьшим количеством "допустимых" значений называется эвристикой ▶**с минимальным количеством оставшихся значений** (*Minimum Remaining Values* — **MRV**). Ее также часто называют эвристикой с "переменной, на которую наложено наибольшее количество ограничений" или эвристикой "первого неудачного завершения", — последнее название применяется потому, что эта эвристическая функция предусматривает выбор переменной, с наибольшей вероятностью приводящей к скорой неудаче, усекая тем самым дерево поиска. Если есть переменная X, уже не имеющая оставшихся допустимых значений, эвристическая функция MRV первой выберет именно ее, и неудача будет обнаружена немедленно, что позволит избежать бессмысленных поисков среди других переменных, которые в любом случае закончатся неудачей после того, как в конечном итоге будет выбрана переменная X. Эвристика MRV обычно работает лучше, чем случайное или статическое упорядочение, иногда на порядок или более, хотя результаты варьируют в зависимости от задачи.

Однако эвристика MRV ничем не поможет в выборе окраски первого региона в задаче раскрашивания карты Австралии, поскольку изначально у каждого региона есть три допустимых цвета. В этом случае более полезной и удобной будет ▶**степенная эвристика**. В ней предпринимается попытка уменьшить коэффициент ветвления в будущих вариантах за счет выбора переменной, которая участвует в наибольшем количестве ограничений на другие переменные с неприсвоенными значениями. На рис. 6.1 переменной с наибольшей степенью, равной 5, является переменная *SA*, — все остальные переменные имеют степень 2 или 3, за исключением *T*, которая имеет степень 0. В действительности после выбора переменной SA применение степенной эвристики позволяет решить задачу без каких-либо неудачных этапов: теперь можно просто выбирать *любой* совместимый цвет в каждой точке выбора и все равно прийти к решению без поиска с возвратами. Эвристика с минимальным количеством оставшихся значений (MRV) обычно обеспечивает более мощное руководство, но степенная эвристика может оказаться полезной в качестве средства разрешения неопределенных ситуаций.

Как только переменная будет выбрана, алгоритм должен принять решение, в каком порядке будут анализироваться ее значения. В этом случае эффективной будет эвристика ▶**с наименее ограничительным значением**. В ней предпочтение отдается значению, при котором из рассмотрения исключается наименьшее количество вариантов выбора значений для соседних переменных в графе ограничений. Например, предположим, что на рис. 6.1 уже сформировано частичное присваивание с $WA = red$ и $NT = green$ и что следующей для проверки выбрана переменная Q. Выбрать для нее синий цвет будет плохим решением, поскольку это исключит последнее допустимое значение в области определения переменной *SA*, непосредственно соседствующей с Q. Именно поэтому эвристика с наименее ограничительным значением предпочтет выбрать значение *red*, а не *blue*. Вообще говоря, в этой эвристике предпринимается попытка сохранить максимальную гибкость для последующих присваиваний значений переменным.

Почему выбор переменной должен выполняться в порядке "худшая — первой", а выбор ее значений в порядке "худшее — последним"? Каждая переменная в конечном итоге должна быть выбрана, поэтому, выбирая те, которые могут привести к неудаче в первую очередь, у нас будет в среднем меньше выполненных успешных назначений, которые затем придется отбрасывать при возврате. При упорядочивании выбора значений хитрость состоит в том, что в этот раз нам нужно только одно успешное решение; поэтому имеет смысл первыми анализировать наиболее вероятные значения. Если бы требовалось найти *все* возможные решения, а не просто первое попавшееся, то порядок выбора анализируемых значений был бы нам безразличен.

6.3.2. Чередование поиска и логического вывода

Выше было показано, как алгоритм AC-3 позволяет сократить области определения переменных *прежде*, чем начнется поиск. Но логический вывод может оказаться даже более мощным, выполняя то же самое непосредственно *в процессе* поиска: при каждом выборе значения для переменной появляется новый шанс прийти к заключению о возможности дополнительного сокращения областей определения для соседних переменных.

Одна из простейших форм логического вывода носит название ▶ **предварительная проверка** (*forward checking*). При каждом присваивании значения переменной X процесс предварительной проверки обеспечивает для нее дуговую совместимость: анализируется каждая переменная Y с неприсвоенным значением, связанная с X некоторым ограничением, и из области определения переменной Y удаляется любое значение, несовместимое с новым значением, выбранным для X.

На рис. 6.7 показан ход поиска с возвратами при решении задачи раскрашивания карты Австралии с использованием предварительной проверки. Из этого примера можно сделать два важных вывода. Во-первых, обратите внимание, что после присваивания $WA = red$ и $Q = green$ области определения переменных NT и SA сокращаются до единственного значения, а значит, все ветвление, связанное с поиском значений для этих переменных, полностью устраняется за счет распространения информации, касающейся переменных WA и Q. Во-вторых, заслуживает внимания тот факт, что после присваивания $V = blue$ область определения переменной SA оказалась пустой. Следовательно, предварительная проверка позволила обнаружить, что частичное присваивание $\{WA = red, Q = green, V = blue\}$ является несовместимым с ограничениями этой задачи, а значит, алгоритм немедленно выполняет возврат.

Для многих задач поиск будет более эффективным, если объединить эвристику MRV с предварительной проверкой. Обратите внимание на ситуацию на рис. 6.7 после выполнения присваивания $\{WA = red\}$. Вполне очевидно, что это присваивание ограничивает возможные назначения для соседних переменных, NT и SA, поэтому есть смысл продолжить обработку этих переменных, и тогда все остальные переменные станут на свои места. Именно это и происходит при использовании

эвристики MRV: каждая из переменных *NT* и *SA* имеет два значения, поэтому одна из них выбирается первой, затем — следующая, затем — *Q*, *NSW* и *V* в указанном порядке. Наконец, переменная *T* все так же имеет три допустимых значения, и ничто не мешает выбрать любое из них. Предварительную проверку можно рассматривать как эффективный способ пошагового вычисления информации, которая нужна эвристике MRV для выполнения своей работы.

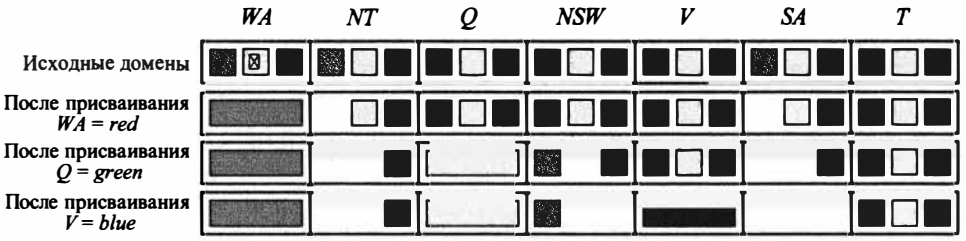


Рис. 6.7. Ход поиска в задаче раскрашивания карты Австралии с использованием предварительной проверки. Присваивание *WA = red* выполняется первым, а затем предварительная проверка приводит к удалению значения *red* из областей определения соседних переменных *NT* и *SA*. После присваивания *Q = green* значение *green* удаляется из областей определения переменных *NT*, *SA* и *NSW*. После присваивания *V = blue* из областей определения *NSW* и *SA* удаляется значение *blue*, в результате чего переменная *SA* остается без допустимых значений

Хотя предварительная проверка обнаруживает много несовместимостей, она не позволяет обнаружить их все. Например, рассмотрим третью строку на рис. 6.7, которая соответствует состоянию после присваивания *Q = green*. Переменные *WA* и *Q* уже приведены к состоянию дуговой совместимости, однако переменные *NT* и *SA* остались с единственным значением *blue* в области определения, а это недопустимо, поскольку их регионы смежные.

Алгоритм **► поддержки дуговой совместимости,** или, иначе говоря, MAC (*Maintaining Arc Consistency*), обнаруживает несоответствия, подобные этому. После того как переменной *X* присваивается значение, процедура INFERENCE вызывает AC-3, но вместо передачи ему очереди из всех дуг в CSP, он запускается только с дугами (X_j, X_i) для всех переменных X_j, которым еще не присвоено значение и которые являются соседями X_i. От этой начальной точки алгоритм AC-3 выполняет распространение ограничения обычным способом, и если у любой переменной окажется пустая область определения, вызов AC-3 завершается неудачей, и это вызывает немедленный возврат. Очевидно, что алгоритм MAC является более мощным, чем метод предварительной проверки, поскольку последний делает то же самое, что и MAC на начальных дугах в очереди MAC; но в отличие от MAC метод предварительной проверки не выполняет рекурсивного распространения ограничений, когда в области определения переменных вносятся изменения.

6.3.3. Интеллектуальный поиск с возвратами: поиск в обратном направлении

В алгоритме BACKTRACKING-SEARCH, приведенном на рис. 6.5, применялось очень простое правило в отношении того, что делать, если какая-то ветвь поиска оканчивается неудачей: просто вернуться к предыдущей переменной и проверить для нее другое значение. Такой метод называется ▶**хронологическим поиском с возвратами**, поскольку повторно посещается тот пункт, в котором было принято *последнее по времени* решение. В данном подразделе рассматриваются лучшие варианты поиска с возвратами.

Рассмотрим, что произойдет при применения простого поиска с возвратами в задаче, показанной на рис. 6.1, при условии фиксированного упорядочивания переменных: Q, NSW, V, T, SA, WA, NT. Предположим, что уже сформировано частичное присваивание $\{Q = red, NSW = green, V = blue, T = red\}$. Но после попытки присвоить значение следующей переменной, SA, обнаруживается, что любое значение нарушает какое-либо ограничение. Осуществляется возврат вверх к переменной T, чтобы назначить новый цвет для Тасмании! Очевидно, что это не имеет смысла — любое изменение цвета для Тасмании не поможет решить проблему с Южной Австралией.

Более разумный подход заключается в возврате к переменной, которая позволит решить возникшую проблему, — к переменной, ответственной за то, что все оставшиеся значения у переменной SA стали недопустимыми. Для этого попробуем отследить множество присваиваний, которые находятся в конфликте с каким-либо значением переменной SA. Такое множество (в данном случае это $\{Q = red, NSW = green, V = blue,\}$) называют ▶**конфликтным набором** для переменной SA. Метод ▶**обратного перехода** выполняет возврат к переменной с последним по времени присвоенным значением из конфликтного множества; в нашем примере обратный переход "перепрыгнет" через узел Тасмании и попробует присвоить новое значение переменной V. Этот метод легко реализуется за счет модификации функции BACKTRACK таким образом, что она будет накапливать множество данных о конфликтах при проверке значений, допустимых для присваивания. Если допустимых значений не будет найдено, алгоритм должен вернуть данные о последнем добавленном элементе из множества конфликтов вместе с индикатором неудачи.

Внимательный читатель уже должен был заметить, что предварительная проверка позволяет определить конфликтное множество без дополнительной работы, поскольку каждый раз, когда процедура предварительной проверки, основанная на присваивании переменной $X = x$, удаляет некоторое значение из области определения переменной Y, она должна добавить $X = x$ к конфликтному множеству переменной Y. Кроме того, каждый раз, когда это последнее значение удаляется из области определения Y, переменные из конфликтного множества Y добавляются к конфликтному множеству переменной X. В этом случае после перехода к Y можно

будет сразу же установить, куда должен быть выполнен обратный переход в случае необходимости. То есть нам теперь известно, что $X=x$ приводит к противоречию (в Y), и следовательно, для X необходимо проверить другое присваивание.

Проницательный читатель уже должен был заметить нечто странное: обратный переход происходит, когда каждое значение в некоторой области определения конфликтует с текущим присваиванием, но предварительная проверка обнаруживает этот случай и вообще исключает возможность достижения такого узла! В действительности можно показать, что каждая ветвь, отсекаемая с помощью обратного перехода, отсекается также с помощью предварительной проверки. Поэтому простой поиск с обратным переходом в сочетании с поиском с предварительной проверкой становится избыточным, и в действительности, когда в поиске используется более строгая проверка совместимости, такая как MAC, необходимо и достаточно выполнить либо одно, либо другое.

Несмотря на замечания, сделанные в предыдущем абзаце, идея, лежащая в основе обратного перехода, остается перспективной, если возврат осуществляется с учетом причин неудачи. При обратном переходе неудача обнаруживается после того, как область определения некоторой переменной становится пустой, но во многих случаях какая-то ветвь поиска становится бесперспективной задолго до того, как это происходит. Еще раз рассмотрим частичное присваивание $\{WA=red, NSW=red\}$ (которое согласно приведенному выше описанию является несовместимым). Предположим, что на следующем этапе предпринимается попытка присваивания $T=red$, а затем осуществляется присваивание значений переменным NT, Q, V, SA. Известно, что ни одно присваивание не применимо для этих последних четырех переменных, поэтому в конечном итоге не остается доступных значений, которые можно было бы попытаться присвоить переменной NT. Теперь возникает вопрос, куда выполнить возврат? Обратный переход не может применяться, поскольку в области определения переменной NT *имеются* значения, совместимые со значениями, ранее присвоенными переменным, — переменная NT не имеет полного конфликтного множества из переменных с ранее присвоенными значениями, которое вызвало бы неудачу при попытке присваивания ей значения. Однако известно, что неудачу вызывают четыре переменные, NT, Q, V и SA, *взятые вместе*, поскольку во множестве переменных с ранее присвоенными значениями должны существовать такие переменные, которые непосредственно конфликтуют с этими четырьмя.

Эти рассуждения приводят к созданию другого, более глубокого определения понятия конфликтного множества для такой переменной, как NT: конфликтным множеством называется множество переменных с ранее присвоенными значениями, *которые наряду со всеми переменными со значениями, присваиваемыми в дальнейшем*, становятся причиной того, что для NT не существует совместимого решения. В таком случае конфликтное множество состоит из переменных WA и NSW, поэтому алгоритм должен выполнить возврат к NSW и пропустить переменную, соответствующую Тасмании. Алгоритм обратного перехода, в котором используются конфликтные множества, определенный таким образом, называется

алгоритмом ▶**обратного перехода, управляемого конфликтами** (*conflict-directed backjumping*).

Теперь необходимо объяснить, как вычисляются эти новые конфликтные множества. Применяемый для этого метод фактически очень прост. "Окончательная" неудача в какой-то ветви поиска всегда возникает из-за того, что область определения некоторой переменной становится пустой; эта переменная имеет типичное конфликтное множество. В нашем примере неудача возникает при присваивании значения переменной *SA*, а ее конфликтным множеством является, скажем, $\{WA, NT, Q\}$. Выполняется обратный переход к переменной Q, и эта переменная *поглощает* конфликтное множество, полученное от *SA* (после исключения из него самой переменной Q), объединяя его со своим прямым конфликтным множеством, представляющим собой $\{NT, NSW\}$. В результате новым конфликтным множеством становится $\{WA, NT, NSW\}$. Это означает, что, продолжая поиск от Q, нельзя найти решение при наличии ранее выполненного присваивания переменным $\{WA, NT, NSW\}$. Поэтому обратный переход выполняется к переменной *NT*, присваивание значения которой выполнено последним по времени по сравнению с другими этими переменными. Переменная *NT* поглощает множество $\{WA, NT, NSW\} - \{NT\}$, объединяя его с собственным прямым конфликтным множеством $\{WA\}$, что приводит к получению множества $\{WA, NSW\}$ (как было указано в предыдущем абзаце). Теперь алгоритм осуществляет обратный переход к *NSW*, как и предполагалось. Подведем итог: допустим, что X_j — текущая переменная, а $conf(X_j)$ — ее конфликтное множество. Если все попытки присваивания каждого возможного значения переменной X_j оканчиваются неудачей, то необходимо выполнить возврат к переменной X_i с последним по времени присвоенным значением во множестве $conf(X_j)$ и установить

$$conf(X_i) \leftarrow conf(X_i) \cup conf(X_j) - \{X_i\}.$$

6.3.4. Определение ограничений с помощью обучения

Если поиск обнаруживает противоречие, обратный переход может указать нам, как далеко нужно вернуться назад, что позволяет не тратить время на проверку значений переменных, бесполезных с точки зрения поиска решения задачи. Но также было бы неплохо избежать повторного попадания в ту же самую ситуацию. Когда поиск приводит к противоречию, нам известно, что возникновение этой проблемы связано с некоторым подмножеством конфликтного множества. ▶**Определение ограничений с помощью обучения** (*constraint learning*) — это идея отыскать минимальный набор переменных из конфликтного множества, вызывающего проблему. Подобный набор переменных вместе с соответствующими их значениями называется ▶**тупиком** (*no-good*). Затем информация о найденном тупике фиксируется либо добавлением в CSP нового ограничения, впредь запрещающего эту комбинацию присваиваний, либо ее сохранением в отдельном кеше тупиков.

Например, рассмотрим состояние $\{WA = red, NT = green, Q = blue\}$, показанное в нижнем ряду на рис. 6.6. Предварительная проверка может показать, что это состояние представляет собой тупик, поскольку в нем отсутствует допустимое значение для присваивания переменной SA. В данном конкретном случае запоминание сведений о найденном тупике ничем нам не поможет, поскольку, как только эта ветвь будет удалена из дерева поиска, тупиковая комбинация уже никогда не встретится. Но если предположить, что дерево поиска на рис. 6.6 на самом деле является частью более крупного дерева поиска, которое начинается с присваивания первых значений переменным V и T, то в этом случае будет целесообразно зафиксировать, что состояние $\{WA = red, NT = green, Q = blue\}$ является тупиком, поскольку поиск будет сталкиваться с этой проблемой вновь и вновь для каждой возможной комбинации присваиваний V и T.

Тупики могут эффективно использоваться в алгоритмах предварительной проверки или обратного перехода. Метод определения ограничений с помощью обучения является одним из наиболее важных методов, используемых в современных решателях CSP для достижения высокой эффективности при решении сложных задач.

6.4. Локальный поиск в задачах удовлетворения ограничений

Было установлено, что алгоритмы локального поиска (см. раздел 4.1) являются очень эффективными средствами решения многих задач УО. В них используется следующая формулировка с полным состоянием (как она была введена в разделе 4.1): в каждом состоянии присваивается значение каждой переменной, а при поиске за один раз изменяется значение одной переменной. В качестве примера будет использоваться задача с восемью ферзями, которая была определена как задача удовлетворения ограничений в разделе 6.1.3. На рис. 6.8 слева представлена начальная позиция с полным присваиванием произвольных значений всем восьми переменным, — как правило, в этом случае нарушается сразу несколько ограничений. Далее случайным образом выбирается конфликтная переменная (в нашем случае это переменная Q_8, представляющая крайний справа столбец). Желательно изменить ее значение таким образом, чтобы это приблизило нас к решению. Наиболее очевидным подходом является выбор такого значения, которое приведет нас к минимальному количеству конфликтов с другими переменными — это эвристика ▶ **минимальных конфликтов**.

На рисунке (позиция слева) имеются две строки, в которых нарушается только одно ограничение. Случайным образом выбираем присваивание $Q_8 = 3$ (т.е. перемещаем ферзя в столбце 8 в ряд 3). На следующей итерации (на рисунке в средине) выбираем для изменения переменную Q_6, обратив внимание на то, что после перемещения ферзя в нижнюю, восьмую, строку с этой переменной больше не будут

связаны какие-либо конфликты. Поскольку в этой позиции (на рисунке справа) больше нет конфликтующих переменных, можно заключить, что найдено решение задачи. Этот алгоритм представлен на рис. 6.9.[2]

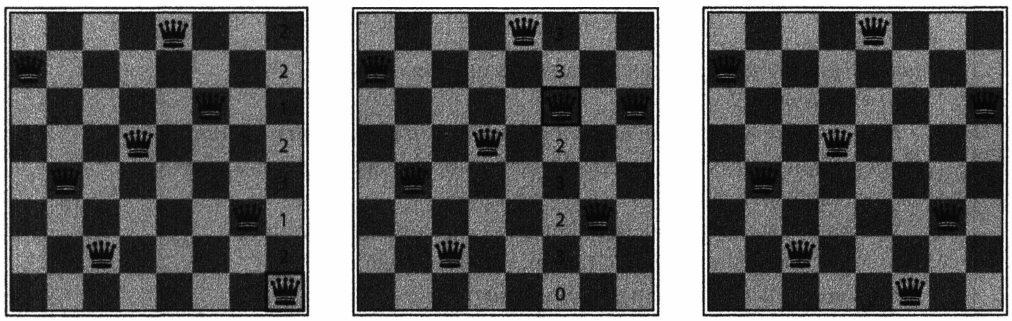


Рис. 6.8. Двухэтапный процесс решения задачи с восемью ферзями на основе эвристики минимальных конфликтов. На каждом этапе выбирается ферзь для повторного назначения ему места в том же столбце. Количество конфликтов (в данном случае — количество атакующих ферзей) показано в каждой клетке. Применяемый алгоритм предусматривает перемещение ферзя в клетку с минимальными конфликтами, разрешая неопределенность посредством случайного выбора

Алгоритм минимальных конфликтов показал себя невероятно эффективным при решении многих задач CSP. Удивляет тот факт, что при решении задачи с n ферзями время выполнения алгоритма минимальных конфликтов (если не учитывать затрат времени на первоначальную расстановку ферзей) оказывается почти *независимым от размера задачи*. Этот алгоритм (после начального присваивания) решает задачу даже с *миллионом* ферзей в среднем за 50 этапов. Это замечательное достижение стало стимулом, побудившим к проведению значительной части исследований по локальному поиску в 1990-х годах, а также позволило уточнить различие между легкими и трудными задачами, которое дополнительно будет рассматриваться в разделе 7.6.3. Грубо говоря, задача с n ферзями является легкой для локального поиска, поскольку решения плотно распределены по всему пространству состояний. Алгоритм минимальных конфликтов также успешно работает и при решении трудных задач. Например, он использовался для планирования наблюдений на космическом телескопе Хаббл, что позволило сократить затраты времени на планирование расписания наблюдений на неделю с трех недель (!) примерно до 10 минут.

[2] Локальный поиск может быть легко расширен до оптимизационных задач с ограничениями (COP). В этом случае для оптимизации целевой функции могут применяться все методы поиска восхождением на вершину и имитации отжига.

```
function MIN-CONFLICTS(csp, max_steps) returns решение current или индикатор
                                                неудачи failure
  inputs: csp, определение задачи удовлетворения ограничений
          max_steps, количество этапов, которые разрешено выполнить,
                прежде чем отказаться от дальнейших попыток

  current ← исходное полное присваивание для csp
  for i = 1 to max_steps do
    if current является решением для задачи csp then return current
    var ← конфликтующая переменная, выбранная случайным образом
          из csp.VARIABLES
    value ← значение v для переменной var, минимизирующее
            значение CONFLICTS(csp, var, v, current)
    присвоить var = value в решении current
  return failure
```

Рис. 6.9. Алгоритм локального поиска MIN-CONFLICTS, предназначенный для решения задач CSP. Начальное состояние может быть выбрано случайным образом или с помощью процесса жадного присваивания, в котором для каждой переменной по очереди выбирается значение с минимальными конфликтами. Функция CONFLICTS подсчитывает количество ограничений, нарушенных данным конкретным значением, с учетом остальной части текущего присваивания

Все методы локального поиска, упоминаемые в разделе 4.1, можно считать кандидатами для применения при решении задач УО, причем некоторые из них оказались особенно эффективными. Ландшафт пространства состояний задач УО при применении эвристики минимальных конфликтов обычно имеет ряд плато. Могут существовать миллионы присваиваний значений переменным, находящихся на расстоянии всего одного конфликта от решения. Поиск на плато — разрешение перемещения в сторону, т.е. перехода к состояниям с тем же значением функции оценки — может помочь локальному поиску быстрее выбраться за его пределы. Подобные блуждания по плато могут контролироваться с помощью метода, получившего название поиска с запретами: в этом случае сохраняется небольшой список недавно посещенных состояний, возвращаться в которые алгоритму запрещено. Алгоритм имитации отжига также может быть успешно использован для выхода за пределы плато.

Другой метод, называемый ►**взвешиванием ограничений**, направлен на то, чтобы сосредоточить поиск на важных ограничениях. Каждому ограничению присваивается числовой весовой показатель, и изначально все веса равны 1. На каждом этапе поиска алгоритм выбирает для изменения ту пару "переменная/значение", которая обеспечивает самый низкий общий вес всех нарушенных ограничений. Далее эти веса регулируются посредством увеличения веса каждого ограничения, нарушаемого в текущем присваивании. Такой подход дает два

преимущества: на плато формируется определенный рельеф, укрепляя уверенность, что есть возможность достичь улучшений из текущего состояния, а также обеспечивается некоторое "обучение": со временем трудные ограничения получают все более высокие веса.

Еще одним преимуществом локального поиска является то, что он может использоваться для оперативной корректировки (см. раздел 4.5) в случае изменения условий задачи. Рассмотрим задачу составления еженедельного расписания полетов. Это расписание может включать тысячи рейсов и десятки тысяч персональных назначений, но из-за плохой погоды в одном аэропорту весь этот график может стать невыполнимым. Поэтому желательно иметь возможность откорректировать график с минимальным количеством изменений. Такую задачу легко выполнить с помощью алгоритма локального поиска, начинающего свою работу с текущего графика. Поиск с возвратами, выполняемый с учетом нового множества ограничений, обычно требует гораздо больше времени и может найти менее удачное решение, требующее внесения в текущий график множества изменений.

6.5. Структура задач

В данном разделе рассматриваются способы, позволяющие использовать для быстрого поиска решений *структуру* самой задачи, представленную в виде графа ограничений. Большинство описанных здесь подходов являются достаточно общими и применимы для решения других задач, помимо CSP, например задач вероятностного формирования рассуждений.

Единственный способ, дающий нам надежду успешно справиться с задачами огромного реального мира, состоит в том, чтобы разложить их на множество подзадач. Еще раз обратимся к графу ограничений задачи раскрашивания карты Австралии (рис. 6.1, *б* или 6.12, *а*). На нем можно отметить один важный факт: Тасмания не связана с материком.[3] Становится понятно, что задачи раскрашивания Тасмании и раскрашивания материка представляют собой ▶**независимые подзадачи** — любое решение для материка в сочетании с любым решением для Тасмании дает допустимое решение для всей карты.

Независимость подзадач можно установить, выделив ▶**связные компоненты** графа ограничений. Каждый компонент соответствует одной подзадаче CSP_i. Если присваивание S_i является решением CSP_i, то $\cup_i S_i$ является решением $\cup_i CSP_i$. Почему это так важно? Предположим, что каждая подзадача CSP_i имеет c переменных из общего количества n переменных, где c — константа. В таком случае должно быть n/c подзадач, и для решения каждой из них потребуется самое большее объем

[3] Аккуратный картограф или патриот Тасмании может возразить, что Тасмания не должна быть окрашена так же, как ближайший к ней штат на материке, чтобы не сложилось впечатление, что она является частью этого штата.

работы d^c, где d — размер области определения. Следовательно, общий объем работы можно оценить как $O(d^c n/c)$, и он *линейно* зависит от n. Без декомпозиции на подзадачи общий объем работы оценивается как $O(d^n)$ и *экспоненциально* зависит от n. Приведем более конкретный пример: разделение булевой задачи CSP с $n = 100$ на четыре подзадачи с $c = 25$ сокращает продолжительность поиска решения в наихудшем случае от величины, сравнимой со временем существования Вселенной, до величины, меньшей одной секунды.

Наличие в задаче полностью независимых подзадач весьма привлекательно, но редко встречается на практике. К счастью, некоторые другие структуры графов также позволяют легко находить решение. Например, граф ограничений представляет собой дерево, когда любые две переменные связаны только одним путем. Ниже будет показано, что ➡ *любая задача УО с древовидной структурой может быть решена за время, линейно зависящее от количества ее переменных.*[4] Ключевым аспектом здесь является новая категория совместимости, называемая ▶**направленной дуговой совместимостью** или DAC (*Directional Arc Consistency*). Задача УО является направленно-дугосовместимой по упорядочению переменных $X_1, X_2, \ldots, X_n$ тогда и только тогда, когда каждая переменная X_i дугосовместима с каждой переменной X_j для $j > i$.

Чтобы найти решение задачи УО с древовидной структурой, сначала надо выбрать переменную, которая будет представлять корень дерева, а затем обеспечить такое упорядочение переменных, когда каждая переменная будет следовать за ее родительской переменной в дереве. Такое упорядочение переменных называется ▶**топологической сортировкой**. На рис. 6.10, *а* приведен пример дерева, а на рис. 6.10, *б* представлен один из возможных вариантов его упорядочения. Любое дерево с n узлами имеет $n - 1$ ребер, поэтому такой граф можно сделать направленно-дугосовместимым за $O(n)$ этапов, на каждом из которых потребуется сравнивать до d возможных значений из области определения для двух переменных за общее время $O(nd^2)$. Когда дуговая совместимость графа будет обеспечена, появится возможность просто просматривать список переменных и выбирать для них любое из оставшихся допустимых значений. Поскольку каждое ребро от родительского узла к дочернему является дугосовместимым, можно утверждать, что для любого значения, выбранного для родительского узла, среди оставшихся значений всегда можно будет выбрать допустимое для его дочернего узла. А это означает, что в процессе поиска возвратов не потребуется и он будет линейно проходить по всему множеству переменных. Полный алгоритм представлен на рис. 6.11.

Теперь, когда у нас есть эффективный алгоритм для деревьев, имеет смысл рассмотреть вопрос о том, можно ли каким-то образом приводить к древовидным

[4] К сожалению, очень немногие регионы мира имеют карты с древовидной структурой, хотя административная структура острова Сулавеси в Индонезии почти соответствует этому определению.

структурам более общие графы ограничений. Существует два основных способа решения этой задачи: один построен на удалении узлов (раздел 6.5.1), а другой — их слиянии друг с другом (раздел 6.5.2).

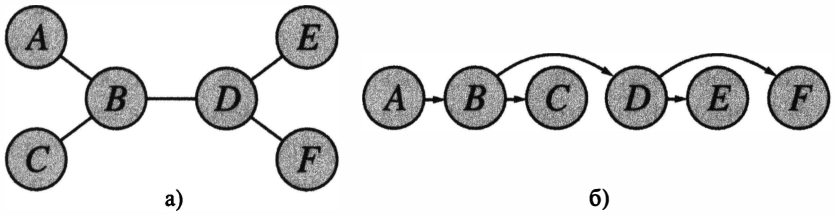


Рис. 6.10. а) Граф ограничений задачи УО с древовидной структурой. **б)** Линейное упорядочение переменных, совместимое с деревом, корнем которого является переменная *A*. Этот тип представления называют топологической сортировкой переменных

function TREE-CSP-SOLVER(*csp*) **returns** решение *assignment* или индикатор отказа *failure*
inputs: *csp*, задача удовлетворения ограничений с компонентами
X (перечень переменных), *D* (область определения),
C (ограничения)

n ← количество переменных в *X*
assignment ← пустое присваивание
root ← любая переменная в *X*
X ← TOPOLOGICALSORT(*X*, *root*)
for *j* = *n* **down to** 2 **do**
  MAKE-ARC-CONSISTENT(PARENT(X_j), X_j)
  if достичь совместимости невозможно **then return** *failure*
for *i* = 1 **to** *n* **do**
  assignment[X_i] ← любое допустимое значение из D_i
  if допустимое значение отсутствует **then return** *failure*
return *assignment*

Рис. 6.11. Алгоритм TREE-CSP-SOLVER для решения задач УО с древовидной структурой. Если задача имеет решение, оно будет найдено за линейное время, а если нет, будет обнаружена несовместимость ограничений

6.5.1. Метод разрыва цикла

Первый способ приведения графа ограничений к древовидной структуре предполагает присваивание значений некоторым переменным таким образом, что в результате все остальные переменные образуют дерево. Рассмотрим граф ограничений для задачи раскрашивания карты Австралии, который еще раз показан на рис. 6.12, *а*. Если из этого графа можно было бы удалить узел *SA*, соответствующий Южной Австралии, то граф превратился бы в дерево, как показано на рис. 6.12, *б*. К счастью, это можно сделать (в графе, разумеется, а не на самом континенте), зафиксировав для переменной *SA* некоторое значение и удалив из областей определения остальных переменных любые значения, несовместимые со значением, выбранным для *SA*.

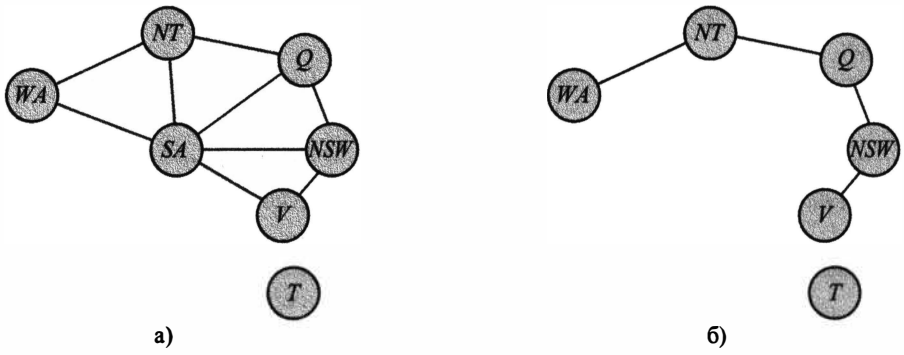


Рис. 6.12. а) Исходный граф ограничений, ранее приведенный на рис. 6.1. **б)** После удаления узла SA этот граф ограничений превращается в лес из двух деревьев

Теперь, после удаления узла *SA* и связанных с ним ограничений, любое решение данной задачи УО будет совместимым со значением, выбранным для *SA*. (Такой способ удобен при решении бинарных задач УО; при наличии ограничений более высокого порядка ситуация усложняется.) Следовательно, для оставшегося дерева можно найти решение с помощью приведенного выше алгоритма, а значит, получить решение и для всей исходной задачи. Безусловно, в общем случае (в отличие от задачи раскрашивания карты) значение, выбранное для узла *SA*, может оказаться неподходящим, поэтому придется проверить каждое из его возможных значений. Общий алгоритм поиска решения указанным способом будет следующим.

1. Выбираем подмножество *S* из множества переменных задачи CSP, такое, что после удаления *S* граф ограничений задачи становится деревом. Подмножество *S* называется ▶**множеством разрыва цикла** (*cycle cutset*).

2. Для каждого возможного присваивания переменным в S, которое удовлетворяет всем ограничениям в S, выполняем следующее:
   а) удаляем из областей определения оставшихся переменных любые значения, несовместимые с данным присваиванием для S;
   б) если оставшаяся задача CSP имеет решение, вернуть это решение вместе с присваиванием для S.

Если множество разрыва цикла имеет размер c, то общее время выполнения алгоритма можно оценить как $O(d^c \cdot (n-c)d^2)$: необходимо проверить каждую из d^c комбинаций значений для переменных в S и для каждой из этих комбинации найти решение задачи с древовидной структурой размером $n-c$. Если исходный граф по своей форме является "почти деревом", то множество c будет небольшим, а экономия времени по сравнению с прямым поиском с возвратами окажется огромной — для задачи со 100 булевыми переменными, если удастся найти множество разрыва цикла размером $c=20$, это позволит сократить время поиска решения от порядка времени существования Вселенной до нескольких минут. Но в наихудшем случае значение c может достигать $(n-2)$. Задача поиска *наименьшего* множества разрыва цикла является NP-трудной, но известно несколько эффективных алгоритмов ее решения. В целом данный алгоритмический подход носит название ►**определение условий выбора множества разрыва цикла** (*cutset conditioning*); мы вернемся к нему в главе 13, где он используется при формировании рассуждений о вероятностях.

6.5.2. Древовидная декомпозиция

Второй способ приведения графа ограничений к древовидной структуре основан на построении ►**древовидной декомпозиции** (*tree decomposition*) графа ограничений. В этом случае исходный граф преобразуется в дерево, каждый узел которого состоит из набора переменных, как показано на рис. 6.13. Древовидная декомпозиция должна удовлетворять следующим трем требованиям.

- Каждая переменная из исходной задачи появляется по меньшей мере в одном из узлов дерева.
- Если в исходной задаче две переменные связаны ограничением, то они должны появиться вместе (и с этим ограничением) по меньшей мере в одном из узлов дерева.
- Если какая-то переменная появляется в двух узлах дерева, то она должна появляться в каждом узле вдоль пути, соединяющего эти узлы.

Первые два условия гарантируют, что в древовидной декомпозиции будут представлены все переменные и ограничения. Третье условие, на первый взгляд, кажется довольно формальным, но на самом деле оно представляет требование, что любая переменная должна иметь одно и то же значение везде, где она появляется:

ограничения в дереве указывают, что переменная в одном узле дерева должна иметь то же значение, что и соответствующая переменная в соседнем узле дерева. Например, переменная *SA* появляется во всех четырех связанных узлах на рис. 6.13, поэтому каждое ребро в древовидной декомпозиции должно включать ограничение, указывающее, что значение переменной *SA* в одном узле должно быть таким же, как значение переменной *SA* в следующем. На основании рис. 6.13 можно убедиться, что эта декомпозиция имеет смысл.

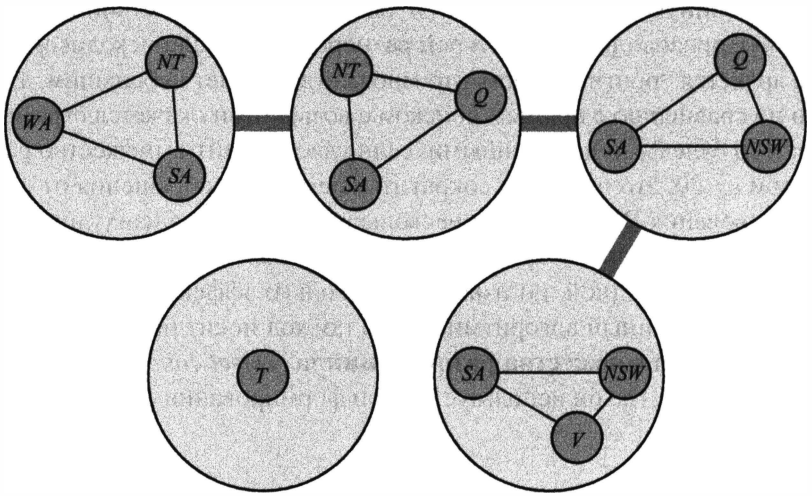


Рис. 6.13. Древовидная декомпозиция графа ограничений, показанного на рис. 6.12, *а*

Сформировав древовидный граф, можно применить к нему алгоритм TREE-CSP-SOLVER и получить решение за время $O(nd^2)$, где n — количество узлов дерева, а d — размер наибольшей области определения. Но обратите внимание на то, что в этом дереве область определения представляет собой множество *кортежей* значений, а не просто отдельных значений.

Например, на рис. 6.13 крайняя слева подзадача на уровне исходной задачи представляет собой подзадачу с переменными $\{WA, NT, SA\}$, областью определения $\{red, green, blue\}$ и ограничениями $WA \neq NT$, $SA \neq NT$, $WA \neq SA$. На уровне дерева данный узел представляет единственная переменная, которой можно дать имя, скажем, *SANTWA* и значением которой должен быть кортеж из трех цветов, например (*red*, *green*, *blue*), но не (*red*, *red*, *blue*), поскольку в этом случае нарушается ограничение $SA \neq NT$ исходной задачи. От этого узла можно перейти к соседнему, переменной которого можно дать имя *SANTQ*, и при этом обнаружить, что есть только один кортеж, (*red*, *green*, *blue*), который является совместимым с тем выбором, который был сделан для переменной *SANTWA*. Точно такой же процесс

повторяется для следующих двух узлов, а в завершение можно произвольным образом выбрать значение для переменной T.

Используя алгоритм TREE-CSP-SOLVER, любую задачу древовидной декомпозиции можно решить за время $O(nd^2)$, что можно полагать эффективным, пока d имеет небольшое значение. Возвращаясь к нашему примеру со 100 булевыми переменными, если каждый узел будет включать 10 переменных, то $d=2^{10}$ и эту задачу можно будет решить за секунды. Однако если в задаче будет присутствовать узел с 30 переменными, на ее решение уйдут столетия.

Любой конкретный граф ограничений допускает много древовидных декомпозиций, и при выборе декомпозиции нужно стремиться к тому, чтобы подзадачи были как можно меньше. ►**Ширина дерева** древовидной декомпозиции графа на единицу меньше размера наибольшей подзадачи, при этом ширина дерева самого графа определяется как минимальная ширина дерева среди всех его древовидных декомпозиций. Если граф имеет ширину дерева w и дана соответствующая древовидная декомпозиция, то данная задача может быть решена за время $O(nd^{w+1})$. А это означает, что ►*задачи УО с графами ограничений, имеющими конечную ширину дерева, могут быть решены за полиномиальное время.*

Что лучше: декомпозиция графа ограничений методом разрыва цикла с временем выполнения $O(d^c \cdot (n-c)\, d^2)$ или древовидная декомпозиция графа с временем выполнения $O(nd^{w+1})$? Всякий раз, когда граф имеет множество разрыва цикла размером c, он также обладает шириной дерева размером $w < c+1$, и в некоторых случаях она может быть намного меньше. Поэтому с точки зрения расхода времени предпочтение следует отдать методу древовидной декомпозиции, однако у метода разрыва цикла есть свое преимущество — его потребности в памяти возрастают по линейному закону, тогда как при древовидной декомпозиции потребности в памяти растут экспоненциально по w.

6.5.3. Симметрия значений

До сих пор здесь рассматривались структурные особенности исключительно графа ограничений. Однако важные структурные особенности могут присутствовать и в *значениях* переменных или в структуре самих отношений ограничений. Рассмотрим задачу раскрашивания карты в d цветов. Для каждого совместимого решения существует набор из $d!$ решений, образуемых перестановкой названий цветов. Например, в отношении карты Австралии нам известно, что переменные WA, NT и SA должны иметь разные цвета, но при этом существует $3! = 6$ способов присвоить три цвета трем областям. Это называется ►**симметрией значений**. Было бы желательно уменьшить пространство поиска в $d!$ раз, избавившись от симметрии в значениях переменных. Добиться этого можно за счет введения ►**ограничений, нарушающих симметрию**. В примере с картой Австралии можно, например, наложить ограничение упорядоченности $NT < SA < WA$, которое требует, чтобы значения этих трех переменных обязательно следовали в указанном

алфавитном порядке. Данное ограничение гарантирует, что допустимым будет только одно из $d!$ возможных решений: $\{NT = blue, SA = green, WA = red\}$.

Как видите, для задачи раскрашивания карты было совсем несложно найти ограничение, устраняющее симметрию значений. Хотя в общем случае задача устранения всей симметрии значений является NP-сложной, было показано, что нарушение симметрии значений является важным и эффективным приемом для широкого спектра задач.

Резюме

- В **задачах удовлетворения ограничений** (УО или CSP) состояния представляются в виде множества пар "переменная/значение", а условия допустимости решения представляются в виде множества ограничений, налагаемых на переменные. Многие важные задачи реального мира могут быть описаны в виде задач УО.
- В ряде методов **логического вывода** ограничения используются для исключения определенных значений переменных. К ним относятся методы обеспечения вершинной, дуговой, путевой и k-совместимости.
- Для решения задач УО обычно применяется **поиск с возвратами** — одна из форм поиска в глубину. С поиском могут тесно переплетаться методы логического вывода.
- Эвристика **определения минимального количества оставшихся значений** и **степенная эвристика** являются независимыми от области определения методами выявления того, какая переменная должна быть выбрана следующей в ходе поиска с возвратами. Эвристика **с наименее ограничительным значением** будет полезна при определении, какие значения выбранной переменной следует проверять в первую очередь. Возврат происходит в том случае, когда для переменной больше не удается найти допустимое присваивание. При использовании метода **обратного перехода, управляемого конфликтами**, возврат происходит непосредственно к источнику проблемы, возникшей в процессе поиска. Метод **определения ограничений с помощью обучения** предполагает запоминание информации о конфликтах по мере их выявления с целью избежать тех же самых конфликтов в дальнейшем поиске.
- Также для решения задач с ограничениями с большим успехом применяется локальный поиск с использованием эвристики **минимальных конфликтов**.
- Сложность решения задач УО в значительной степени зависит от структуры ее графа ограничений. Задачи с древовидной структурой могут быть решены за линейное время. Метод определения **множества разрыва цикла** позволяет преобразовать задачу УО общего вида в задачу с древовидной структурой и является очень эффективным (потребность в памяти возрастает

линейно), если удается найти множество разрыва цикла, небольшое по размеру. Метод **древовидной декомпозиции** предусматривает преобразование задачи УО в дерево подзадач и является эффективным, если **ширина дерева** графа ограничений невелика, однако его потребность в памяти возрастает экспоненциально с увеличением ширины дерева графа ограничений. Комбинирование методов определения множества разрыва цикла и древовидной декомпозиции позволяет добиться необходимого компромисса между затратами времени и потребностью в памяти.

Библиографические и исторические заметки

Греческий математик Диофант (ок. 200–284) предложил задачу, включающую алгебраические ограничения на уравнения, и нашел способ ее решения, хотя и не разработал общей методологии. В его честь уравнения с целочисленными областями определения сейчас называют ▶**диофантовыми уравнениями**. Индийский математик Брахмагупта (ок. 650) первым представил общее решение для уравнений вида $ax + by = c$ с целочисленными областями определения. Систематические методы решения линейных уравнений путем исключения переменных исследовались Гауссом ([822], 1829); первые попытки решения ограничений с линейными неравенствами можно найти в работах Фурье ([764], 1827).

Задачи удовлетворения ограничений с конечными областями определения также имеют долгую историю. Например, одной из старых задач в математике является **раскрашивание графа** (раскрашивание карты — частный случай данной задачи). Гипотезу о четырех цветах (утверждающую, что любой плоский граф можно раскрасить с помощью четырех или менее цветов) впервые выдвинул Френсис Гутри, студент де Моргана, в 1852 году. Эта задача не поддавалась решению — несмотря на то, что в некоторых публикациях утверждалось обратное — до тех пор, пока доказательство не было получено с помощью компьютера Аппелем и Хакеном ([60], 1977) (см. книгу *Four Colors Suffice* (Wilson, 2004)). Пуристы были разочарованы тем, что разработка части доказательства была возложена на компьютер, поэтому Жорж Гонти ([890], 2008), применив средство доказательства теорем COQ, получил формальное доказательство того, что программа Аппеля и Хакена, использованная ими для вывода доказательства, была правильной.

Специальные классы задач удовлетворения ограничений рассматривались на протяжении всей истории развития компьютерных наук. Одним из самых значимых первых примеров решения таких задач была система SKETCHPAD (Сутерленд [2153], 1963), которая решала геометрические ограничения на чертежах и была предшественником современных программ рисования и инструментальных средств САПР. Определение задач УО как *общего* класса задач принадлежит Уго Монтанари ([1608], 1974). Первые попытки приведения задач УО высокого порядка к чисто бинарным задачам с помощью вспомогательных переменных

(см. упр. 6.6) были предприняты в XIX веке логиком Чарльзом Сандерсом Пирсом. Его методы были введены в тематику CSP Риной Дехтер ([579], 1990) и доработаны Бакусом и ван Биком ([100], 1998). Задачи УО с предпочтениями между решениями широко изучались в исследованиях по оптимизации, а обобщение инфраструктуры CSP, позволяющее использовать предпочтения, было приведено в [224] (1997).

Расширению популярности методов распространения ограничений способствовало успешное решение Вальцем ([2291], 1975) задач полиэдральной линейной разметки для систем компьютерного зрения. Вальц показал, что применение методов распространения ограничений при решении многих задач позволяет полностью исключить необходимость в использовании возвратов. Монтанари ([1608], 1974) ввел понятие графов ограничений и предложил метод распространения ограничений на основе понятия путевой совместимости. Алан Макворт ([1474], 1977) предложил алгоритм AC-3 для обеспечения дуговой совместимости, а также выдвинул общую идею о том, что поиск с возвратами необходимо сочетать с обеспечением совместимости некоторой степени. Более эффективный алгоритм обеспечения совместимости дуг, AC-4, разработанный Мором и Хендерсоном ([1605], 1986), в худшем случае выполняется за время $O(cd^2)$, но для средних задач он работает медленнее, чем AC-3. Алгоритм PC-2 (Макворт [1474], 1977) достигает путевой совместимости во многом таким же способом, как в алгоритме AC-3 достигается дуговая совместимость.

Вскоре после появления статьи Макворта исследователи приступили к экспериментам в области поиска компромисса между затратами на обеспечение совместимости и достигаемыми за счет этого преимуществами с точки зрения сокращения объема поиска. Харалик и Эллиот ([964], 1980) предпочли алгоритм минимальной предварительной проверки, описанный Макгрегором ([1548], 1979), тогда как Гашниг ([815], 1979) предложил применять полную проверку дуговой совместимости после присваивания значения каждой переменной, — алгоритм, позднее получивший название "MAC" в работе Сабина и Фрейдера ([1952], 1994). В последней статье представлены некоторые убедительные свидетельства того, что при решении более трудных задач УО полная проверка дуговой совместимости вполне окупается. Фрейдер ([777], 1978; [778], 1982) исследовал понятие k-совместимости и ее связи со сложностью решения задач УО. Апт ([62], 1999) описал универсальную алгоритмическую инфраструктуру, в рамках которой можно проанализировать алгоритмы распространения совместимости, а соответствующие обзоры были представлены Бессьером ([207], 2006) и позднее Бартоком и соавт. ([136], 2010).

Специальные методы обработки ограничений более высокого порядка и глобальных ограничений впервые были разработаны в контексте ►**логического программирования в ограничениях**. Превосходный обзор исследований в этой области приведен в [1500] (1998). Ограничение *Alldiff* исследовалось Регином ([1866], 1994), Стергиу и Уолшем ([2128], 1999), а также ван Хуве ([2256], 2001). Для ограничения *Alldiff* существуют более сложные алгоритмы логического

вывода (см. ван Хуве и Катриель [2257], 2006), в которых распространяется больше ограничений, но ценой бо́льших вычислительных затрат. Ограничения с пределами были включены в тематику логического программирования в ограничениях ван Хентенриком и соавт. ([2255], 1998). Общий обзор по теме глобальных ограничений представлен ван Хуве и Катриелем ([2257], 2006).

Головоломка судоку уже приобрела широкую известность как задача УО и в этом свете была описана, например, в работе Симонис ([2074], 2005). Агербек и Хансен в [17] (2008) описали некоторые стратегии и показали, что судоку на поле $n^2 \times n^2$ относится к классу NP-трудных задач.

В 1850 году Карл Гаусс описал алгоритм рекурсивного поиска с возвратами для решения задачи восьми ферзей, который ранее был опубликован в немецком шахматном журнале *Schachzeitung* в 1848 году. Гаусс назвал этот метод *Tatonniren* — производное от французского слова *tâtonner*, означающего "ощупывать все вокруг, когда находишься в темноте".

По словам Дональда Кнута (в личном общении), Р.Дж. Уокер ввел термин "возврат" (*backtrack*) еще в 1950-х годах, а в [2287] (1960) Уокер описал основной алгоритм поиска с возвратами и его использование для поиска всех решений задачи с 13 ферзями. Голомб и Баумерт в [884] (1965) сформулировали на примерах определение общего класса комбинаторных задач, к которым применимы методы поиска с возвратами, и ввели то, что мы сейчас называем эвристикой MRV. В [225] (1975) Битнер и Рейнгольд представили исчерпывающий обзор методов поиска с возвратами. Брелаз ([300], 1979) использовал степенную эвристику для устранения неопределенности, возникающей после применения эвристики MRV, — полученный в итоге алгоритм, несмотря на его простоту, все еще остается наилучшим методом раскрашивания произвольных графов в k-цветов. Харалик и Эллиот ([964], 1980) предложили эвристику с наименее ограничительным значением.

Основной метод обратного перехода был разработан Джоном Гашнигом ([815], 1977; [816], 1979). Кондрак и ван Бик ([1275], 1997) показали, что этот алгоритм по сути перекрывается алгоритмом предварительной проверки. Обратный переход, управляемый конфликтами, был предложен Проссером ([1824], 1993). Дехтер ([578], 1990) представил метод обратного перехода, основанный на графе, который ограничивает сложность алгоритмов на основе обратного перехода как функцию графа ограничений (Дехтер и Фрост [587], 2002).

Наиболее общая форма интеллектуального поиска с возвратами была разработана уже довольно давно Столлманом и Зюссманом ([2122], 1977). В предложенном ими методе ► **поиска с возвратами, управляемого зависимостями**, комбинируются метод обратного перехода и запоминание тупиков. Этот метод привел к разработке **систем обеспечения истинности** (Дойл [645], 1979), которые будут рассматриваться в разделе 10.6.2. Связь между этими двумя научными областями проанализирована в [558] (1989).

В указанной выше работе Столлмана и Зюссмана была также предложена идея **определения ограничений с помощью обучения**, в соответствии с которой

частичные результаты, полученные в ходе поиска, можно сохранять и повторно использовать на последующих этапах этого поиска. Данная идея была формализована Риной Дехтер ([578], 1990). Особенно простым методом является **проставление обратных отметок** (Гашниг [816], 1979), в котором для предотвращения повторной проверки ограничений сохраняются и используются совместимые и несовместимые попарные присваивания. Проставление обратных отметок можно комбинировать с обратным переходом, управляемым конфликтами. Кондрак и ван Бик ([1275], 1997) представили гибридный алгоритм, который превосходит любой из этих методов, взятых по отдельности.

В методе **динамического поиска с возвратами** (Гинзберг [861], 1993) сохраняются успешные частичные присваивания из полученных позднее подмножеств переменных при поиске с возвратами по предыдущему варианту, которые не влияют на дальнейший успех. Москевич и соавт. ([1627], 2001) показали, как эти и другие методы использовались для создания эффективного решателя SAT. Эмпирические исследования нескольких рандомизированных методов обратного отслеживания были проведены Гомесом и соавт. ([886], 2000; [888], 2001). Ван Бик в [2246] (2006) представил обзор различных вариантов поиска с возвратами.

Применение локального поиска при решении задач удовлетворения ограничений стало популярным после публикации работы Киркпатрика и соавт. ([1230], 1983) с описанием метода эмуляции отжига (см. главу 4), который широко используется для решения задач проектирования СБИС и планирования. Бек и соавт. в [153] (2011) предоставили обзор последних работ по составлению графиков работы цехов и предприятий. Эвристика минимальных конфликтов впервые была предложена Гу в [928] (1989) и независимо разработана Минтоном и соавт. ([1588], 1992). В [2110] (1994) показано, как эта эвристика может использоваться для решения задачи с 3 000 000 ферзями меньше чем за минуту. Этот поразительный успех локального поиска на основе эвристики минимальных конфликтов при решении задачи с n ферзями привел к переоценке природы и распространения "легких" и "трудных" задач. Питер Чизмен и соавт. ([407], 1991) исследовали сложность случайно сгенерированных задач УО и обнаружили, что почти все такие задачи являются либо тривиально легкими, либо не имеют решений. "Трудные" экземпляры задач встречаются только в том случае, если параметры генератора задач устанавливаются в некотором узком диапазоне, в пределах которого лишь примерно половина задач является разрешимой. Этот феномен рассматривается дополнительно в главе 7.

Конолиге ([1280], 1994) показал, что локальный поиск уступает поиску с возвратами при решении задач с локальной структурой определенного типа. Это привело к появлению работ, сочетающих в себе локальный поиск и логический вывод, таких как [1792] (1995). Хуз и Цанг в [1057] (2006) предложили обзор методов локального поиска, а соответствующие учебники были предоставлены Хузом и Стюцле ([1056], 2004), а также Аартсом и Ленстрой ([2], 2003).

Исследования, касающиеся структуры и сложности задач CSP, начались с работ [779] (1985) и [1475] (1985), в которых было показано, что поиск в

дугосовместимых деревьях выполняется без каких-либо возвратов. Аналогичные результаты применительно к ациклическим гиперграфам были получены в области теории баз данных (Бири и др. [155], 1983). Баярдо и Миранкер ([146], 1994) предложили алгоритм для задач УО с древовидной структурой, который выполняется за линейное время без какой-либо предварительной обработки. В [578] (1990) Рина Дехтер описывает подход с использованием метода разрыва цикла.

Со времени публикации этих статей был достигнут значительный прогресс в отношении получения более общих результатов, касающихся связи между сложностью решения задачи УО и структурой ее графа ограничений. Понятие ширины дерева было введено специалистами по теории графов Робертсоном и Сеймуром ([1896], 1986). Дехтер и Перл ([582], 1987; [583], 1989), основываясь на работе Фрейдера, применяли по сути то же самое понятие (названное ими **индуцированной шириной**) к задачам удовлетворения ограничений и разработали подход древовидной декомпозиции, кратко описанный в разделе 6.5.

Опираясь на эту работу и на результаты из области теории баз данных, Готлобб и др. ([911], 1999; [912], 1999) разработали понятие **ширины гипердерева**, которое основано на методе характеризации задачи УО как гиперграфа. Они не только показали, что любую задачу УО с шириной гипердерева w можно решить за время $O(n^{w+1}\log n)$, но и обосновали утверждение, что критерий ширины гипердерева превосходит все ранее предложенные критерии "ширины" в том смысле, что в некоторых случаях ширина гипердерева является конечной, тогда как ширина, определяемая другими критериями, является неограниченной.

Для алгоритма RELSAT Баярда и Шрага ([147], 1997), в котором комбинируются методы определения ограничений с помощью обучения и обратного перехода, было показано, что он превосходит многие другие алгоритмы в отношении времени выполнения. Это привело к разработке алгоритмов И-ИЛИ-поиска, применимых как к CSP, так и к вероятностным рассуждениям ([588], 2007). Браун и соавт. ([318], 1988) предложили идею разрыва симметрии в задачах УО, а Гент и др. ([839], 2006) дали соответствующий обзор.

Тематику задач **удовлетворения распределенных ограничений** можно представить как решение задач УО при наличии совокупности агентов, каждый из которых контролирует некоторое подмножество переменных, на которые наложены ограничения. Ежегодные семинары по этому классу задач проводятся начиная с 2000 года, а хорошее освещение состояния дел можно найти в работах Коллин и др. [462] и Пирс и др. [1748] (2008).

Сравнение алгоритмов решения задач УО является преимущественно эмпирической наукой: лишь несколько теоретических результатов показывают, что один алгоритм доминирует над другим по всем задачам, поэтому необходимо проводить соответствующие эксперименты, чтобы понять, какие алгоритмы работают лучше в типичных классах задач. Как указывает Хукер ([1055], 1995), необходимо проявлять определенную осторожность, чтобы различить конкурентное тестирование (как это происходит в соревнованиях среди алгоритмов, построенных на

оценке времени выполнения) и научное тестирование, целью которого является выявление свойств алгоритма, определяющих его эффективность для некоторого класса задач.

Учебники Апта ([63], 2003), Дехтер ([585], 2003), Цанга ([227], 1993) и Лекотра ([1370], 2009), а также сборник Росси и соавт. ([1920], 2006) являются отличными ресурсами по работе с ограничениями. Имеется несколько хороших обзоров, включая работы Дехтер и Фрост [587] (2002) и Бартак и др. [136] (2010). В Карбоннел и Купер [369], (2016) дан обзор легко разрешимых классов задач УО. В работе [1997] (1997) приведен аналитический обзор алгоритмов поиска с возвратами, а в работе [101] (1995) приведен обзор, характеризующийся большей практической направленностью. Программирование в ограничениях рассматривается в книге Апта ([63], 2003), а также Фрювирта и Абденнадхера ([794], 2003). Статьи на тему удовлетворения ограничений регулярно публикуются в журнале *Artificial Intelligence* и в специализированном журнале *Constraints*. Последние версии решателей SAT описываются в материалах ежегодного конкурса *International SAT Competition*. Основным местом встречи специалистов в этой области является конференция *International Conference on Principles and Practice of Constraint Programming*, часто называемая просто *CP*.

Упражнения

6.1. Сколько решений имеет задача раскрашивания карты, показанная на рис. 6.1? Сколько будет решений, если использовать четыре краски? А если две?

6.2. Рассмотрим задачу размещения k слонов на шахматной доске размером $n \times n$ так, что на доске не будет двух слонов, атакующих друг друга, при условии, что k задано и $k \leq n^2$.

а) Выберите формулировку данной задачи УО. Что в вашей формулировке будет являться переменными?

б) Каковы возможные значения каждой переменной?

в) Какие множества переменных в задаче ограничены и как?

г) Теперь рассмотрим задачу размещения на доске как можно большего количества коней без каких-либо атак. Объясните, как решить эту задачу с помощью локального поиска, определив соответствующие функции ACTIONS и RESULT и разумную целевую функцию.

6.3. Рассмотрите задачу составления (а не решения) кроссвордов: подбора слов, которые укладываются в прямоугольную сетку. Эта сетка, которая задается как часть задачи, определяет, какие квадраты будут пустыми, а какие затененными. Предположим, что предоставлен список слов (т.е. словарь) и задача состоит в том, чтобы заполнить пустые квадраты с использованием любого подмножества из этого списка. Точно сформулируйте эту задачу следующими двумя способами.

а) Как общую задачу поиска. Выберите соответствующий алгоритм поиска и определите эвристическую функцию. Как лучше заполнять пустые квадраты: одновременно по одной букве или по одному слову?

б) Как задачу удовлетворения ограничений. Переменные должны представлять слова или буквы?

Какая формулировка, по вашему мнению, является лучшей? Объясните, почему?

6.4. Дайте точные формулировки каждой из перечисленных ниже задач в виде задач удовлетворения ограничений.

а) Планирование покрытия пола прямоугольниками. Найти в большом прямоугольнике неперекрывающиеся места для размещения меньших прямоугольников.

б) Составление расписания занятий. Исходные данные: фиксированное количество преподавателей и классов, список предлагаемых занятий и список возможных временных интервалов для занятий. С каждым преподавателем связано множество занятий, которые он может проводить.

в) Гамильтонов тур: задана сеть городов, соединенных дорогами. Выберите порядок посещения всех этих городов, не допуская их повторного посещения.

6.5. Вручную решите криптоарифметическую задачу, приведенную на рис. 6.2, используя метод поиска с возвратами с предварительной проверкой и эвристики MRV и наименее ограничительного значения.

6.6. Покажите, как можно преобразовать одно тернарное ограничение типа "$A + B = C$" в три бинарных ограничения с использованием вспомогательной переменной. Можете принять допущение о конечных областях определения. (*Подсказка.* Рассмотрите использование новой переменной, которая принимает значения, являющиеся парами других значений, и примените такие ограничения, как "X является первым элементом пары Y".) Затем покажите, как аналогичным образом можно трактовать ограничения больше чем с тремя переменными. Наконец, покажите, как можно устранить унарные ограничения путем модификации области определения переменных. На этом будет завершена демонстрация того, что любую задачу УО можно преобразовать в задачу УО только с бинарными ограничениями.

6.7. Рассмотрите следующую логическую головоломку. В пяти домах, окрашенных в разные цвета, живут представители пяти национальностей, которые предпочитают разные сорта сладких батончиков, пьют разные напитки и держат разных домашних питомцев. На основе приведенных ниже фактов найдите ответ на вопрос "В каком доме держат зебру и в каком доме пьют воду?"

Англичанин живет в доме красного цвета.
Испанец имеет собаку.
Норвежец живет в первом доме слева.
Зеленый дом находится непосредственно справа от дома цвета слоновой кости.
Человек, который любит батончики *Hershey*, живет в доме, расположенном рядом с домом человека, который держит лису.
Батончики *KitKat* предпочитают в доме желтого цвета.
Норвежец живет в доме, расположенном рядом с домом синего цвета.
Любитель батончиков *Smart* держит улиток.
Тот, кто предпочитает батончики *Snickers*, пьет апельсиновый сок.
Украинец пьет чай.
Японец ест батончики *Milky Way*.
Батончики *KitKat* едят в доме, расположенном рядом с домом, где держат лошадь.
Кофе пьют в доме зеленого цвета.
Молоко пьют в среднем доме.

Обсудите различные представления этой задачи в виде задачи УО. По каким причинам можно предпочесть одно представление другому?

6.8. Рассмотрим граф с 8 узлами: A_1, A_2, A_3, A_4, H, T, F_1, F_2. Узлы A_i соединены с узлами A_{i+1} для всех i, каждый узел A_i соединен с узлом H, узел H соединен с узлом T, а узел T соединен с каждым узлом F_i. Вручную найдите вариант трехцветной раскраски этого графа, используя стратегию поиска с возвратами с обратным переходом, управляемым конфликтами, при следующем упорядочивании: переменные — A_1, H, A_4, F_1, A_2, F_2, A_3, T; значения переменных — R, G, B.

6.9. Объясните, почему при поиске решения задачи УО хорошая эвристика — выбирать переменную, которая является наиболее ограниченной, но при этом выбирать наименее ограничительное значение.

6.10. Создайте случайные примеры задачи раскрашивания карты, формируемые следующим способом: разбросайте n точек на единичном квадрате; случайным образом выберите точку X и соедините ее прямой линией с ближайшей точкой Y, которая еще не соединена с X, причем эта соединительная линия не должна пересекать какую-либо другую линию. Повторяйте эти действия до тех пор, пока есть возможность проводить дальнейшие соединения. Точки представляют регионы на карте, а линии определяют их соседей. Теперь попробуйте найти k-цветные раскраски для каждой карты при $k=3$ и $k=4$, используя методы минимальных конфликтов, поиска с возвратами, поиска с возвратами с предварительной проверкой и поиска с возвратами с эвристикой MAC. Составьте таблицу среднего времени выполнения для каждого алгоритма для различных значений n вплоть до максимального, с которым вам удастся справиться. Прокомментируйте полученные результаты.

6.11. Используйте алгоритм AC-3, чтобы показать, что дуговая совместимость позволяет обнаружить несовместимость частичного присваивания $\{WA = green, V = red\}$ для задачи, приведенной на рис. 6.1.

6.12. Используйте алгоритм AC-3, чтобы показать, что дуговая совместимость позволяет обнаружить несовместимость частичного присваивания $\{WA = red, V = blue\}$ для задачи, приведенной на рис. 6.1.

6.13. Какова в наихудшем случае временная сложность алгоритма AC-3 для задачи УО с древовидной структурой?

6.14. Алгоритм AC-3 помещает обратно в очередь каждую дугу (X_k, X_i) каждый раз, когда *любое* значение удаляется из области определения X_i, даже если каждое значение X_k совместимо с несколькими оставшимися значениями X_i. Предположим, что для каждой дуги (X_k, X_i) отслеживается количество оставшихся значений X_i, которые являются совместимыми с каждым значением X_k. Объясните, как эффективно обновлять эти числа, и тем самым покажите, что дуговую совместимость можно обеспечить за суммарное время $O(n^2 d^2)$.

6.15. Алгоритм TREE-CSP-SOLVER (рис. 6.10) обеспечивает дуговую совместимость, начиная с листовых узлов и перемещаясь обратно к корню дерева. Почему это так? Что будет, если он будет двигаться в противоположном направлении?

6.16. В этой главе головоломка судоку была представлена как задача УО, которую следует решать посредством поиска по частичным присваиваниям, потому что именно так люди обычно приступают к решению этих головоломок. Но, безусловно, решать эти задачи можно и методом локального поиска по полным присваиваниям. Насколько

хорошо доступный вам решатель, использующий эвристику минимальных конфликтов, покажет себя при решении задач судоку?

6.17. Своими словами дайте определение терминам *ограничение*, *поиск с возвратами*, *дуговая совместимость*, *обратный переход*, *минимальные конфликты* и *множество разрыва цикла*.

6.18. Своими словами дайте определение терминам *ограничение*, *коммутативность*, *дуговая совместимость*, *обратный переход*, *минимальные конфликты* и *множество разрыва цикла*.

6.19. Предположим, нам известно, что граф имеет множество разрыва цикла, содержащее не больше k узлов. Опишите простой алгоритм поиска минимального множества разрыва цикла, время прогона которого ненамного превышает $O(n^k)$ для задачи УО с n переменными. Найдите в литературе методы обнаружения приближенно минимальных множеств разрыва цикла за время, которое полиномиально зависит от размера множества разрыва. Обеспечивает ли наличие таких алгоритмов практическую применимость метода разрыва цикла?

6.20. Рассмотрим задачу мозаичного покрытия поверхности (накрывая ее полностью и точно) с использованием n костей домино (прямоугольников размером 2×1). Поверхность представляет собой произвольно соединенную сторонами (т.е. примыкающими друг к другу на длину всей стороны) совокупность из $2n$ квадратов размером 1×1 (например, это может быть шахматная доска, шахматная доска с некоторыми отсутствующими квадратами, ряд из квадратов размером 10×1 и т.д.).

а) Сформулируйте эту задачу в точности как задачу УО, в которой кости домино являются переменными.

б) Сформулируйте эту задачу в точности как задачу УО, в которой квадраты являются переменными, сохраняя ее пространство состояний как можно меньшим. (*Подсказка*: имеет ли значение, какая именно кость домино накрывает данную пару квадратов?)

в) Постройте состоящую из 6 квадратов поверхность таким образом, чтобы для нее ваша формулировка задачи УО из задания *б* имела древовидный граф ограничений.

г) Точно опишите множество разрешимых экземпляров задачи, которые имеют древовидный граф ограничений.

Часть III

Знания, рассуждения и планирование

Логические агенты 369

Логика первого порядка 441

Логический вывод в логике первого порядка 493

Представление знаний 551

Автоматизированное планирование 603

ГЛАВА 7

Логические агенты

В этой главе обсуждается проектирование агентов, способных формировать собственные представления о сложном мире, использовать процесс логического вывода для получения новых представлений о мире, а также применять эти новые представления для определения того, что следует делать.

Совершенно очевидно, что люди обладают знаниями и именно эти знания дают им возможность действовать. Это правило можно распространить и на область искусственного интеллекта: ►**агенты, основанные на знаниях**, используют процесс ►**рассуждения** на основе внутреннего ►**представления** знаний, чтобы принять решение, какие действия предпринять.

Агенты, решающие задачи, обсуждавшиеся в главах 3 и 4, также обладают некоторыми знаниями, но только в очень ограниченном, негибком представлении. Они знают, какие действия им доступны и каким будет результат выполнения конкретного действия в конкретном состоянии, но они не знают общих фактов. Агент, прокладывающий маршрут, не знает, что участок дороги не может иметь длины, выраженной отрицательным числом километров. Агент, решающий головоломку ***Восемь***, не знает, что две плитки не могут одновременно находиться в одном и том же квадрате игрового поля. Знания, которые имеют эти агенты, очень полезны для нахождения пути от начального до целевого состояния, но больше ни для чего другого.

Атомарные представления, используемые агентами, решающими задачи, также очень ограничены. Например, в частично наблюдаемой среде у агента, решающего задачи, единственным доступным вариантом представления того, что он знает о текущем состоянии, является список всех возможных конкретных состояний. Перед человеком можно поставить цель отправиться в любой город США с населением менее 10 тыс. человек, но если потребуется сказать то же самое агенту, решающему задачи, формально описать эту цель можно будет только как явное множество примерно из 16 тыс. городов, удовлетворяющих заданному условию.

В главе 6 мы впервые обратились к развернутому представлению состояний, когда они задаются как присваивания определенных значений переменным, и это шаг в правильном направлении, позволяющий некоторым элементам агента работать независимо от области определения задачи и использовать более эффек-

тивные алгоритмы. В этой главе подобный шаг предпринимается в отношении логического вывода, — мы, так сказать, разработаем **логику** как общий класс представлений для поддержки агентов, основанных на знаниях. В результате эти агенты обретут способность комбинировать и рекомбинировать информацию для достижения множества целей. И эта цель может быть весьма далека от текущих потребностей момента, — как в том случае, когда математик доказывает теорему или астроном вычисляет ожидаемую продолжительность существования нашей планеты. Агенты, основанные на знаниях, могут принимать новые задачи в форме явно описанных целей; они могут быстро достичь компетентности, когда им сообщают новые знания об окружающей среде или обучают их. И они могут успешно адаптироваться к изменениям в окружающей среде посредством обновления соответствующих знаний.

В разделе 7.1 обсуждается общий проект подобного агента. В разделе 7.2 представляется новая простая окружающая среда — вымышленный мир *вампуса* — и иллюстрируется работа агента, основанного на знаниях, без привлечения какой-либо технической детализации. Далее, в разделе 7.3 объясняются общие принципы **логики**, а в разделе 7.4 дается формальное определение **логики высказываний**. Логика высказываний (или иначе — *пропозициональная* логика, от лат. *propositio* — "высказывание") — это раздел логики, изучающий сложные высказывания, образованные из простых. И хотя она менее выразительна по сравнению с **логикой первого порядка** (глава 8), являющейся каноническим структурированным представлением, в логике высказываний иллюстрируются все основные понятия логики. В ней также имеется хорошо разработанная технология формирования рассуждений, обсуждаемая в разделах 7.5 и 7.6. И наконец в разделе 7.7 рассматривается сочетание концепции агентов, основанных на знаниях, с методами логики высказываний, что позволяет создать некоторых простых агентов, способных успешно действовать в мире вампуса.

7.1. Агенты, основанные на знаниях

Центральным компонентом любого агента, основанного на знаниях, является его ▶**база знаний**, или сокращенно **KB** (*Knowledge Base*). База знаний представляет собой множество ▶**высказываний**. (Здесь слово "высказывание" используется как формальный термин, смысл которого близок, но не идентичен понятию высказывания в русском, английском или любом другом естественном языке.) В базе знаний каждое высказывание выражено на языке, называемом ▶**языком представления знаний**, и представляет некоторое утверждение о мире. Если высказывание принято как данное изначально, без вывода из какого-либо другого высказывания, его принято называть ▶**аксиомой**.

Должен существовать некоторый способ добавления к базе знаний новых высказываний, а также способ извлечения из нее того, что уже известно. Стандарт-

ными названиями для этих операций являются соответственно TELL и ASK. Обе такие операции могут быть связаны с проведением ▶**логического вывода**, т.е. могут потребовать получения новых высказываний из старых. Логический вывод должен подчиняться требованию, что ответ на любой запрос ASK к базе знаний должен следовать исключительно из того, что на данный момент уже было введено в нее посредством операции TELL. Ниже будет дано более точное определение важного понятия "логического следствия". А пока будем считать, что процесс логического вывода не должен включать операций, не подчиняющихся строгим правилам.

На рис. 7.1 показана общая структура программы агента, основанной на знаниях. Как и все агенты, описанные в данной книге, этот агент принимает на входе результаты акта восприятия *percept* и возвращает действие *action*. Агент поддерживает базу знаний, *KB*, которая может изначально содержать некоторые ▶**фоновые знания**.

---

function KB-AGENT(*percept*) **returns** действие *action*
  persistent: *KB*, база знаний
    t, счетчик, обозначающий время, изначально равен 0

  TELL(*KB*, MAKE-PERCEPT-SENTENCE(*percept*, *t*))
  action ← ASK(*KB*, MAKE-ACTION-QUERY(*t*))
  TELL(*KB*, MAKE-ACTION-SENTENCE(*action*, *t*))
  t ← *t* + 1
  return *action*

Рис. 7.1. Универсальный агент, основанный на знаниях. Получив восприятие *percept*, агент добавляет его в свою базу знаний, запрашивает базу знаний о наилучшем действии *action* и сообщает базе знаний, что он действительно совершил это действие

При каждом вызове программа агента выполняется в три этапа. Во-первых, с помощью операции TELL программа вводит в базу знаний результаты акта восприятия. Во-вторых, с помощью операции ASK она передает в базу знаний запрос о том, какое действие следует предпринять. В процессе поиска ответа на этот запрос могут быть проведены всесторонние рассуждения о текущем состояния мира, о результатах возможных последовательностей действий и т.д. В-третьих, с помощью операции TELL программа агента регистрирует в базе знаний свой выбор и возвращает действие в таком виде, чтобы его можно было выполнить.

Подробные сведения о языке представления скрыты в трех функциях, которые реализуют интерфейс между датчиками и исполнительными механизмами с одной стороны и между основным представлением и системой формирования рассуждений — с другой. Функция MAKE-PERCEPT-SENTENCE формирует высказывание,

подтверждающее, что агент получил результаты данного акта восприятия в данный момент времени. Функция MAKE-ACTION-QUERY формирует высказывание, запрашивающее о том, какое действие следует выполнить в текущий момент. Наконец, функция MAKE-ACTION-SENTENCE формирует высказывание, подтверждающее, что выбранное действие было выполнено. Подробные сведения о механизме логического вывода скрыты внутри функций TELL и ASK и будут представлены в следующих разделах.

Программа агента, приведенная на рис. 7.1, внешне очень похожа на программы агентов с поддержкой внутреннего состояния, описанные в главе 2. Однако в силу приведенных выше определений операций TELL и ASK программу агента, основанного на знаниях, уже нельзя рассматривать как произвольную программу для вычисления действий. Она теперь больше подходит для описания на таком ►**уровне знаний**, на котором для определения поведения агента требуется указать лишь то, что ему известно и каковы его цели.

Например, агенту, управляющему автоматизированным такси, может быть поставлена цель доставить пассажира из города Сан-Франциско в округ Марин и он может знать, что мост "Золотые ворота" является единственным возможным связующим звеном между пунктом отправления и пунктом назначения. В таком случае можно ожидать, что агент — водитель такси поедет через мост "Золотые ворота", *поскольку ему известно, что именно так можно достичь цели.* Обратите внимание: этот анализ не зависит от того, как агент — водитель такси действует на ►**уровне реализации**. Не имеет значения, как реализованы его знания о географии (в виде связных списков или в виде растровых изображений карт), а также проводит ли он рассуждения, манипулируя строками символов, сохраненными в регистрах, или распространяя зашумленные сигналы в нейронной сети.

Агент, основанный на знаниях, может быть создан просто путем передачи ему с помощью операции TELL всего того, что ему требуется знать. Начав с пустой базы знаний, разработчик агента может сообщать ему высказывания по одному, пока агент не получит полные знания о том, как действовать в окружающей его среде. Этот вариант называется ►**декларативным** подходом к построению системы. В противоположность этому при ►**процедурном** подходе разработчик кодирует желаемое поведение агента непосредственно в его программном коде. В 1970- и 1980-х годах между сторонниками этих двух подходов не прекращались острые дискуссии. Однако сейчас уже не вызывает сомнения, что в конструкции успешного агента должны сочетаться и декларативные, и процедурные элементы и что декларативные знания часто можно скомпилировать в более эффективный процедурный код.

Агента, основанного на знаниях, также можно обеспечить такими механизмами, которые позволят ему обучать самого себя. Подобные механизмы, подробно обсуждаемые в главе 19, создают общие знания об окружающей среде из серии восприятий. Самообучающийся агент может быть полностью автономным.

7.2. Мир вампуса

В этом разделе описывается простейшая среда, в которой агенты, основанные на знаниях, смогут наглядно продемонстрировать свою ценность. ▶**Мир вампуса** — это пещера, состоящая из залов, соединенных проходами. Где-то в этой пещере притаился ужасный вампус — чудовище, пожирающее всех, кто входит в зал, где он находится. Агент может убить вампуса стрелой, но она у него только одна. В некоторых залах есть бездонные ямы, в которые упадет любой, кто в них зайдет (кроме вампуса, который слишком велик для того, чтобы в них провалиться). Единственное, что скрашивает этот мрачный мир, — это возможность найти кучу золота. Хотя мир вампуса является довольно скромным с точки зрения современных стандартов компьютерных игр, он позволяет проиллюстрировать некоторые важные моменты, касающиеся интеллектуальных агентов.

Пример мира вампуса приведен на рис. 7.2. Точное определение среды этой задачи, соответствующее требованиям, которые приведены в разделе 2.3, дано с помощью следующего описания PEAS.

- **Показатели производительности.** За то, что агент находит золото, он получает +1000 очков, за то, что агент попадает в яму или его съедает вампус, ему назначается –1000 очков, он платит –1 очко за каждое выполненное действие и –10 очков за использование стрелы. Игра заканчивается либо когда агент умирает, либо когда он выбирается из пещеры.

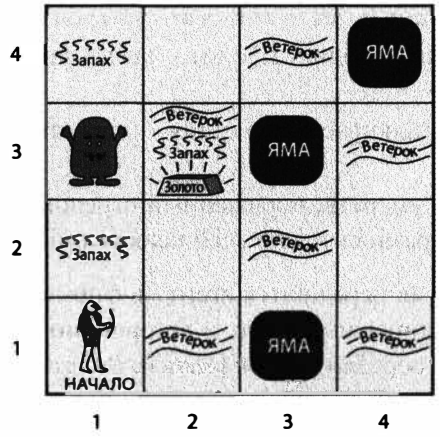


Рис. 7.2. Типичный пример мира вампуса. Агент находится в нижнем левом углу, лицом к востоку (смотрит вправо)

- **Среда.** Залы в пещере образуют квадратную решетку 4 × 4. Агент всегда начинает движение с квадрата, обозначенного как [1,1], и смотрит вправо.

Местонахождения золота и вампуса выбираются случайным образом с равномерным распределением из числа квадратов, отличных от начального квадрата. Кроме того, в каждом квадрате, отличном от начального, с вероятностью 0,2 может находиться яма.

- **Исполнительные механизмы.** Агент может двигаться вперед (действие *Forward*), поворачиваться влево (*TurnLeft*) и вправо (*TurnRight*) на 90°. Агент погибает жалкой смертью, если входит в квадрат, где имеется яма или живой вампус. (Входить в квадрат с мертвым вампусом безопасно, но при этом никуда не деться от неприятного запаха.) Попытка продвижения вперед остается безрезультатной, если перед агентом находится стена. Чтобы поднять золото, лежащее в том квадрате, где находится агент, можно воспользоваться действием *Grab*. С помощью действия *Shoot* можно пустить стрелу по прямой линии в том направлении, куда смотрит агент. Стрела продолжает движение до тех пор, пока она либо не попадет в вампуса (и убьет его), либо не ударится в стену. У агента имеется только одна стрела, поэтому какого-либо эффекта можно достичь лишь при первом выполнении действия *Shoot*. И наконец действие *Climb* позволяет агенту выбраться из пещеры, но только если он находится в квадрате [1,1].
- **Датчики.** У агента есть пять датчиков, каждый из которых предоставляет ему только один вид информации.
  - В квадратах, непосредственно (не по диагонали) смежных с тем, где находится вампус, агент чувствует неприятный запах (восприятие *Stench*).[1]
  - В квадратах, непосредственно (не по диагонали) смежных с тем, где находится яма, агент будет чувствовать ветерок (восприятие *Breeze*).
  - В квадрате, где находится золото, агент замечает блеск (восприятие *Glitter*).
  - Когда агент наталкивается на стену, он чувствует глухой удар (восприятие *Bump*).
  - Перед тем как умереть, пораженный стрелой вампус издает жалобный крик, который разносится по всей пещере (восприятие *Scream*).

Результаты восприятия передаются агенту в форме списка из пяти символов. Например, если есть неприятный запах и ветерок, но нет блеска, удара и крика, программе агента будет передан список [*Stench*, *Breeze*, *None*, *None*, *None*].

Данную среду мира вампуса можно классифицировать в соответствии с характеристиками, определенными в главе 2. Очевидно, что она является детерминированной, дискретной, статической и одноагентной. (К счастью, вампус по пещере не перемещается.) Она является последовательной, потому что вознаграждение

[1] Предположительно в квадрате, содержащем вампуса, также присутствует неприятный запах, но любой агент, входящий в этот квадрат, будет съеден до того, как сможет что-либо почувствовать.

может быть получено только после выполнения многих действий. Она частично наблюдаемая, поскольку нельзя непосредственно воспринять некоторые аспекты ее состояния: местоположение агента, жив или мертв вампус, имеется ли в наличии стрела. Что касается расположения ям и местонахождения вампуса, то можно рассматривать эти аспекты среды как ненаблюдаемые элементы ее состояния, — в этом случае модель перехода для окружающей среды будет полностью известна, а поиск опасных мест будет пополнять знания агента о состоянии. С другой стороны, можно считать, что неизвестна сама модель переходов, поскольку агент не знает, какое именно действие *Forward* окажется фатальными. В этом случае обнаружение мест расположения ям и вампуса будет дополнять знания агента о модели перехода.

Для агента в такой ситуации главная проблема состоит в его первоначальном неведении в отношении конфигурации окружающей среды, и преодоление этого незнания, видимо, требует выполнения логических рассуждений. В большинстве случайных реализаций мира вампуса у агента есть возможность безопасно добыть золото, но иногда агенту приходится выбирать: вернуться ли домой с пустыми руками или рисковать жизнью, чтобы найти золото. Около 21% вариантов среды являются совершенно неблагоприятными, поскольку золото находится в яме или окружено ямами.

Давайте посмотрим, как агент, основанный на знаниях, будет действовать, исследуя мир вампуса, изображенный на рис. 7.2. Воспользуемся неформальным языком представления знаний, заключающимся в записи специальных символов в сетке квадратов 4×4 (как показано на рис. 7.3 и 7.4).

Исходная база знаний агента содержит все действующие в окружающей среде правила, которые были описаны выше. В частности, агент знает, что находится в квадрате [1,1] и что квадрат [1,1] является безопасным. Эти знания представлены на рис. 7.3, *а* как символы **A** и **OK** соответственно, помещенные в квадрат [1,1].

Первым восприятием является [*None*, *None*, *None*, *None*, *None*], на основании которого агент может сделать вывод, что соседние квадраты, [1,2] и [2,1], являются безопасными, т.е. их можно пометить символами **OK**. На рис. 7.3, *а* показано состояние знаний агента на данный момент.

Осторожный агент переходит только в такой квадрат, который, как ему известно, имеет отметку **OK**. Предположим, что агент решил переместиться вперед, в квадрат [2,1]. Здесь его датчики сообщают о наличии ветерка (квадрат получает отметку **B**), и это означает, что в одном из соседних квадратов должна быть яма. В квадрате [1,1] ямы нет, поэтому она должна быть либо в квадрате [2,2], либо в квадрате [3,1], либо и в том, и в другом. На рис. 7.3, *б* на возможность наличия ямы в этих квадратах указывает отметка **P?** в этих квадратах. На данный момент агенту известен только один квадрат с отметкой **OK**, который еще не был посещен. Поэтому агент благоразумно поворачивается кругом и возвращается в квадрат [1,1], а затем переходит в квадрат [1,2].

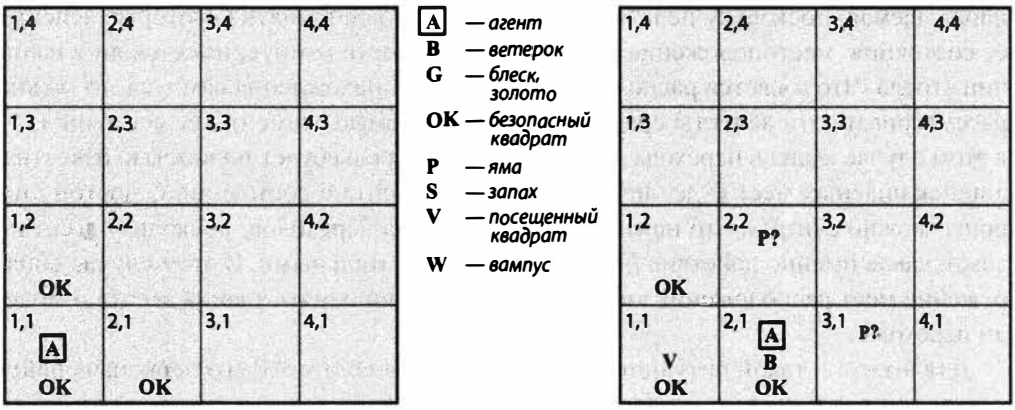


Рис. 7.3. Первое действие, выполненное агентом в мире вампуса. **а)** Исходная ситуация — агент получил восприятие [*None*, *None*, *None*, *None*, *None*]. **б)** Ситуация после перемещения в квадрат [2,1], в котором агент получил восприятие [*None*, *Breeze*, *None*, *None*, *None*]

Здесь агент воспринимает неприятный запах и квадрат [1,2] получает отметку **S**, — новое состояние знаний агента показано на рис. 7.4, *а*. Наличие неприятного запаха в этом квадрате означает, что где-то рядом находится вампус. Но вампус не может находиться в квадрате [1,1] по правилам игры и его нет в квадрате [2,2] (поскольку агент обнаружил бы неприятный запах, когда находился в квадрате [2,1]). Следовательно, агент с уверенностью может сделать вывод, что вампус находится в квадрате [1,3], на что и указывает отметка **W!** в этом квадрате. Кроме того, отсутствие ветерка в квадрате [1,2] означает, что в соседнем квадрате [2,2] ямы нет. Поскольку ранее агент уже сделал вывод, что яма *должна* быть в квадрате [2,2] или [3,1], это означает, что в действительности она присутствует в квадрате [3,1]. Это довольно сложный логический вывод, поскольку он объединяет знания, полученные в разное время в разных местах, и полагается на отсутствие восприятия, чтобы принять ответственное решение.

Теперь агент доказал сам себе, что в квадрате [2,2] нет ни ямы, ни вампуса, поэтому можно удалить для него отметку **P!** и взамен поставить отметку **OK**, а затем безопасно перейти туда. Мы не будем обсуждать состояние знаний агента в квадрате [2,2], а просто предположим, что агент повернулся и перешел в квадрат [2,3], — новое состояние его знаний показано на рис. 7.4, *б*. В квадрате [2,3] агент обнаруживает блеск, поэтому он должен схватить золото и вернуться в исходную точку, тем самым закончив игру.

Обратите внимание, что в каждом конкретном случае, когда агент делает заключение на основании имеющейся у него информации, оно *гарантированно*



а) б)

Рис. 7.4. Два более поздних этапа деятельности агента. **а)** После возврата в [1,1], а затем перехода в [1,2], где было получено восприятие [*Stench*, *None*, *None*, *None*, *None*]. **б)** После перехода в [2,2], а затем в [2,3], где было получено восприятие [*Stench*, *Breeze*, *Glitter*, *None*, *None*]

будет правильным, если эта информация верна. Это фундаментальное свойство логических рассуждений. В оставшейся части данной главы будет показано, как создавать логических агентов, которые могут представлять информацию и делать выводы, подобные приведенным выше в этом разделе.

7.3. Логика

В этом разделе приведен краткий обзор основных понятий логического представления и рассуждения. Эти прекрасные идеи не зависят от какой-либо определенной формы логики, поэтому описание любых технических деталей этих форм откладывается до следующего раздела. Вместо них в данном разделе используются знакомые всем примеры из области обычной арифметики.

В разделе 7.1 было указано, что базы знаний состоят из высказываний. Эти высказывания выражаются в соответствии с ▶**синтаксисом** языка представления, определяющего форму всех правильно сформированных высказываний. Понятие синтаксиса можно вполне очевидно проиллюстрировать примерами из обычной арифметики: "$x+y=4$" — правильно построенное высказывание, а "$x2y+=$" — нет.

Логика должна также определять ▶**семантику** языка или, иначе, "смысл" высказываний. Семантика языка определяет ▶**истинность** каждого высказывания относительно каждого из ▶**возможных миров**. Например, семантика в арифметике определяет, что высказывание "$x+y=4$" истинно в мире, где x равно 2 и y равно 2, но ложно в мире, где x равно 1, а y также равно 1. В стандартной логике каждое вы-

сказывание должно быть либо истинным, либо ложным в каждом из возможных миров — в ней не может быть каких-либо "промежуточных" состояний.[2]

В дальнейшем, когда потребуется уточнить какое-либо определение, вместо выражения "один из возможных миров" будет использоваться термин ▶**модель**. В то время как возможные миры можно рассматривать как (потенциально) реальные варианты среды, в которых агент может находиться или не находиться, модели являются математическими абстракциями, каждая из которых устанавливает значение истинности (истина или ложь) для каждого высказывания, соответствующего данной модели. Неформально модель можно понимать как возможный мир, например рассматривать x и y как количество мужчин и женщин, сидящих за столом для игры в бридж; тогда высказывание $x+y=4$ становится истинным, когда общее количество игроков равно четырем. Формально допустимыми моделями являются просто все возможные присваивания переменным x и y неотрицательных целочисленных значений. Каждое такое присваивание определяет истинность любого высказывания в арифметике, переменными которого являются x и y. Если высказывание α является истинным в модели m, то говорят, что m ▶**удовлетворяет** α или иногда что m **является моделью для** α. Мы будем использовать обозначение $M(\alpha)$ для обозначения множества всех моделей для α.

Теперь, сформулировав определение понятия истинности, можно перейти к обсуждению темы логических рассуждений. Для этого сначала необходимо ввести понятие логического ▶**следствия** между высказываниями — идею о том, что одно высказывание может *логически следовать* из другого. В математических обозначениях это выглядит следующим образом:

$$\alpha \models \beta.$$

Эта запись означает, что высказывание α влечет за собой высказывание β. Формальное определение логического следствия является таковым: $\alpha \models \beta$ тогда и только тогда, когда в любой модели, в которой α является истинным, высказывание β также будет истинным. Воспользовавшись только что введенным обозначением, можно записать

$$\alpha \models \beta \text{ тогда и только тогда, когда } M(\alpha) \subseteq M(\beta).$$

(Обратите внимание на направление здесь операции $\subseteq$: если $\alpha \models \beta$, то α является *более сильным* утверждением, чем β: оно исключает *больше* возможных миров.) Понятие следствия применимо и в арифметике: вполне понятно, что из высказывания $x=0$ следует высказывание $xy=0$. Очевидно, что в любой модели, где x равно нулю, абсолютно неизбежно, что xy также будет равно нулю (независимо от значения y).

Анализ такого же типа можно применить и к примеру рассуждений о мире вампуса, приведенному в предыдущем разделе. Рассмотрим ситуацию, показанную

[2] Нечеткая логика, обсуждаемая в главе 13, допускает разные степени истинности.

на рис. 7.3, *а*: агент ничего не обнаружил в квадрате [1,1], а в квадрате [2,1] почувствовал ветерок. Эти восприятия в сочетании со знаниями агента о правилах, действующих в мире вампуса (описание PEAS в разделе 7.2), составляют его базу знаний. Агенту необходимо узнать, имеются ли ямы в соседних квадратах, [1,2], [2,2] и [3,1]. В каждом из этих трех квадратов может находиться или не находиться яма, поэтому (в данном конкретном примере) существует $2^3 = 8$ возможных моделей. Все они представлены на рис. 7.5.[3]

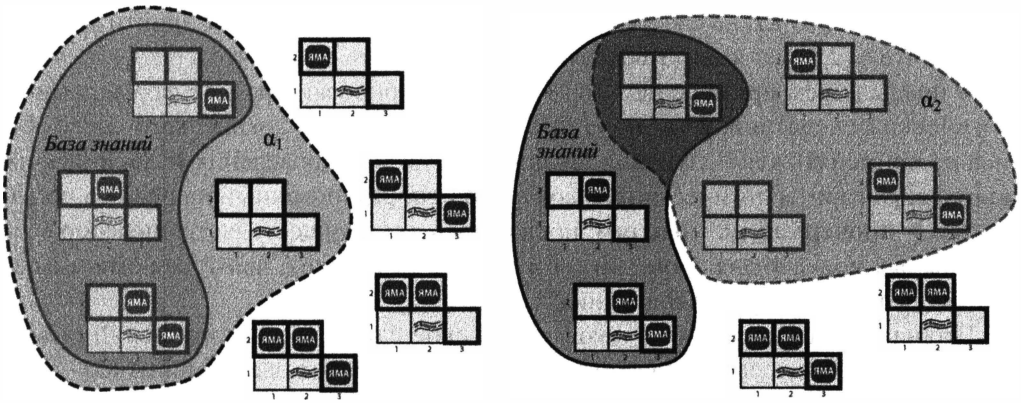


Рис. 7.5. Возможные модели, описывающие наличие ям в квадратах [1,2], [2,2] и [3,1]. База знаний, соответствующая наблюдениям, что в квадрате [1,1] нет ничего, а в квадрате [2,1] чувствуется ветерок, обведена сплошной линией. **а)** Пунктирной линией обведены модели, удовлетворяющие высказыванию α_1 (в квадрате [1,2] нет ямы). **б)** Пунктирной линией обведены модели, удовлетворяющие высказыванию α_2 (в квадрате [2,2] нет ямы)

Базу знаний можно рассматривать как набор высказываний или как одно высказывание, утверждающее все отдельные высказывания. База знаний будет ложной в моделях, противоречащих тому, что знает агент, например база данных будет ложной в любой модели, в которой квадрат [1,2] содержит яму, потому что в квадрате [1,1] не ощущается ветерок. Фактически существуют только три модели, в которых база знаний (или *KB*) является истинной, — на рис. 7.5 они окружены сплошной линией. Теперь рассмотрим два возможных вывода:

α_1 = "В квадрате [1,2] нет ямы." α_2 = "В квадрате [2,2] нет ямы."

[3] Хотя на данном рисунке модели показаны как фрагменты мира вампуса, в действительности они представляют собой не что иное, как варианты присваивания значений *true* и *false* высказываниям "в квадрате [1,2] имеется яма" и т.д. Для определения модели, в математическом смысле этого понятия, не нужно описывать в ней "страшных, покрытых шерстью" вампусов.

Модели, удовлетворяющие высказываниям α_1 и α_2, окружены пунктирными линиями на рис. 7.5, ***а*** и 7.5, ***б*** соответственно. Проанализировав ситуацию, можно установить следующее:

в каждой модели, в которой *KB* истинна, высказывание α_1 также истинно.

Следовательно, $KB \models \alpha_1$: в квадрате [1,2] нет ямы. Также можно увидеть следующее:

в некоторых моделях, в которых *KB* истинна, высказывание α_2 является ложным.

Следовательно, из *KB* не следует высказывание α_2: агент не может прийти к заключению, что в [2,2] нет ямы. (Также он не может сделать вывод, что в [2,2] есть яма.)[4]

В предыдущем примере не только проиллюстрирован процесс формирования логических следствий, но и показано, как определение логического следствия можно применять для получения заключений, т.е. для проведения ▶**логического вывода**. Алгоритм логического вывода, проиллюстрированный на рис. 7.5, называется ▶**проверкой по моделям**, поскольку в нем осуществляется перебор всех возможных моделей для проверки того, что высказывание α является истинным во всех моделях, в которых *KB* истинна, т.е. $M(KB) \subseteq M(\alpha)$.

Чтобы лучше понять, что такое логическое следствие и логический вывод, можно представить себе, что множество всех логических следствий из КВ — это стог сена, а высказывание α — это иголка. Логическое следствие подобно утверждению о том, что в стоге сена есть иголка, а логический вывод можно сравнить с ее поиском. Это различие отражено и в формальных обозначениях: если некоторый алгоритм логического вывода i позволяет логическим путем вывести высказывание α из базы знаний *KB*, то можно записать следующее:

$$KB \vdash_i \alpha,$$

что произносится как "α логически выводится из *KB* с помощью алгоритма *i*" или "алгоритм *i* позволяет логически вывести α из *KB*".

Алгоритм логического вывода, позволяющий получить только такие высказывания, которые действительно следуют из базы знаний, называется ▶**непротиворечивым** или ▶**сохраняющим истинность**. Непротиворечивость — это в высшей степени желательное свойство. Противоречивая процедура логического вывода в ходе своей работы, по сути, создает то, чего нет на самом деле: объявляет об обнаружении несуществующих иголок. Можно легко показать, что алгоритм проверки по моделям, если он применим,[5] является непротиворечивым.

---

[4] Агент может лишь вычислить *вероятность* того, что в квадрате [2,2] есть яма; в главе 12 показано, как это сделать.

[5] Алгоритм проверки по моделям может применяться, если пространство моделей является конечным, например в мирах вампуса с фиксированными размерами. С другой стороны, в арифметике пространство моделей является бесконечным: даже если мы ограничимся целыми числами, то количество пар значений для x и y в выражении $x + y = 4$ является бесконечным.

Желательным является также свойство ▶ **полноты**: алгоритм логического вывода называется полным, если позволяет вывести любое высказывание, которое следует из базы знаний. Когда речь идет о настоящих стогах сена, имеющих конечный объем, то представляется вполне очевидным, что систематическое исследование всегда позволяет определить, есть ли иголка в данном стоге сена. Но со многими базами знаний связан бесконечно большой стог сена, состоящий из следствий, поэтому обеспечение полноты становится важной проблемой.[6] К счастью, существуют полные процедуры логического вывода для многих форм логики, являющиеся достаточно выразительными для того, чтобы справиться со многими базами знаний.

Выше фактически был описан процесс формирования рассуждений, выводы которого гарантированно являются истинными в любом мире, в котором истинны предпосылки; в частности, ▶ *если база знаний является истинной в реальном мире, то любое высказывание, полученное логическим путем из этой базы знаний с помощью непротиворечивой процедуры логического вывода, также является истинным в реальном мире*. Поэтому, несмотря на то, что процесс логического вывода оперирует "синтаксисом" (внутренними физическими конфигурациями, такими как биты в регистрах, или схемами электрических возбуждений в мозговых структурах), этот процесс *соответствует* связям реального мира, согласно которым некоторые аспекты реального мира имеют место благодаря тому, что имеют место другие аспекты реального мира.[7] Это соответствие между миром и его представлением продемонстрировано на рис. 7.6.

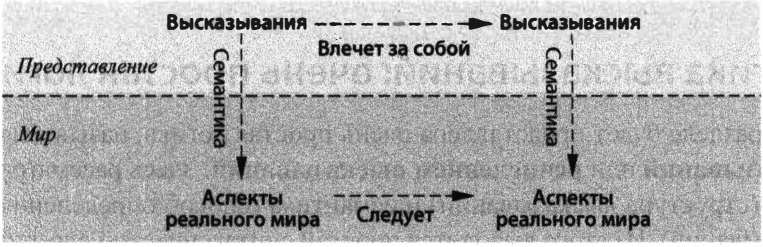


Рис. 7.6. Высказывания представляют собой элементы физической конфигурации агента, а формирование рассуждений — это процесс создания новых элементов физической конфигурации из уже существующих. Процесс формирования логических рассуждений должен гарантировать, что новые элементы конфигурации будут представлять те аспекты мира, которые действительно следуют из аспектов, представленных уже существующими элементами конфигурации

[6] Сравните эту ситуацию со случаем бесконечных пространств поиска, описанным в главе 3, где поиск в глубину является неполным.

[7] Как писал Витгенштейн ([2368], 1922) в своем знаменитом *Трактате*, "Мир — это все, что имеет место".

Последний вопрос, который необходимо рассмотреть, — это проблема ▶**обоснования**: установления связи между процессами логических рассуждений и реальной средой, в которой существует агент. В частности, это вопрос о том, ➡*как узнать, что база знаний является истинной в реальном мире?* (В конце концов, база знаний *KB* — это просто "синтаксические конструкции" в голове агента.) Это философская проблема, которой посвящено очень много книг (см. главу 27). Простой ответ состоит в том, что указанная связь создается с помощью датчиков агента. Например, наш агент для мира вампуса имеет датчик запаха. Обнаружив неприятный запах, программа агента создает соответствующее высказывание в базе данных. Следовательно, всякий раз, когда данное высказывание находится в базе знаний, оно является истинным и в реальном мире. Поэтому смысл и истинность высказываний, в которых выражаются результаты восприятий, определяются с помощью процессов формирования ощущений и составления высказываний, которые ими порождаются. А что можно сказать об остальных знаниях агента, таких как его уверенность, что местонахождение вампусов выдает наличие неприятного запаха в соседних квадратах? Это не прямое представление единственного акта восприятия, а общее правило, которое, возможно, было выведено из опыта восприятий, но не идентично высказываниям об этом опыте. Выработка подобных общих правил в процессе формирования высказываний называется **обучением** и является темой части V данной книги. Обучение чревато ошибками. Может оказаться, что вампусы распространяют неприятный запах во все дни, *кроме 29 февраля високосного года*, поскольку в эти дни они принимают ванну. Поэтому в реальном мире база знаний может оказаться не истинной, но при правильной процедуре обучения у агента есть повод для оптимизма.

7.4. Логика высказываний: очень простая логика

В этом разделе будет представлена очень простая логика, называемая ▶**логикой высказываний** или **исчислением высказываний.** Здесь рассматриваются ее синтаксис (структура высказываний) и семантика (способ определения истинности высказываний). Из этого выводится простой синтаксический алгоритм логического вывода, реализующий семантическое понятие логического следствия. Безусловно, все это рассматривается в применении к миру вампуса.

7.4.1. Синтаксис

Синтаксис логики высказываний определяет допустимые высказывания. ▶**Атомарные высказывания** (неделимые синтаксические элементы) состоят из одного ▶**пропозиционального символа** (пропозициональной переменной). Каждый такой символ (переменная) обозначает высказывание, которое может быть либо истинным, либо ложным. Для их обозначения будем использовать имена, начинающиеся с прописной буквы и, возможно, включающие другие буквы

или индексы: P, Q, R, $W_{1,3}$ или *FacingEast*. Эти имена произвольны, но часто выбирались так, чтобы иметь какое-то мнемоническое значение. Например, символ $W_{1,3}$ может использоваться для обозначения высказывания, согласно которому вампус находится в квадрате [1,3]. (Напомним, что символы, подобные $W_{1,3}$, являются *атомарными*, т.е. элементы W, 1 и 3 не следует воспринимать как осмысленные части этого символа.) Существуют два пропозициональных символа, имеющих постоянный смысл: *True* — высказывание, которое всегда истинно, и *False* — высказывание, которое всегда ложно. ▶**Сложные высказывания** строятся из более простых высказываний с помощью ▶**логических связок**, к которым относятся скобки и логические операторы. Широко применяются пять описанных ниже логических связок.

- $\neg$ (нет). Такое высказывание, как $\neg W_{1,3}$, называется ▶**отрицанием** высказывания $W_{1,3}$. ▶**Литерал** представляет собой либо атомарное высказывание (**положительный литерал**), либо отрицаемое атомарное высказывание (**отрицательный литерал**).

- $\wedge$ (и). Высказывание, основной связкой которого является $\wedge$, такое как $W_{1,3} \wedge P_{3,1}$, называется ▶**конъюнкцией**, а его части называются ▶**конъюнктами**. (Символ $\wedge$ напоминает букву "А" в слове "*And*" — англ. "И".)

- $\vee$ (или). Высказывание, в котором используется связка $\vee$, такое как $(W_{1,3} \wedge P_{3,1}) \vee W_{2,2}$, называется ▶**дизъюнкцией**, а его части **дизъюнктами**, — в данном примере это $(W_{1,3} \wedge P_{3,1})$ и $W_{2,2}$.

- $\Rightarrow$ (влечет за собой). Такое высказывание, как $(W_{1,3} \wedge P_{3,1}) \Rightarrow \neg W_{2,2}$, называется ▶**импликацией** (или **следованием**). Его ▶**посылкой**, или **антецедентом**, является $(W_{1,3} \wedge P_{3,1})$, а его ▶**следствием**, или **консеквентом**, является $\neg W_{2,2}$. Импликации называют также **связкой** или ▶**правилом if–then** (*если–то*). В других книгах символ импликации иногда записывается как $\supset$ или $\rightarrow$.

- $\Leftrightarrow$ (тогда и только тогда). Высказывание вида $W_{1,3} \Leftrightarrow \neg W_{2,2}$ называется ▶**двухсторонней импликацией** (или **эквивалентностью**). Это высказывание будет истинным тогда и только тогда, когда либо обе его части истинны, либо обе его части ложны. В других книгах символ двухсторонней импликации иногда записывается как $\leftrightarrow$ или $\equiv$.

Формальная грамматика логики высказываний приведена на рис. 7.7. (Нотация BNF объясняется в приложении Б.) Эта грамматика дополнена списком, определяющим порядок выполнения операторов и позволяющим устранить неопределенность при наличии в выражении нескольких операций. Оператор "нет" ($\neg$) имеет самый высокий приоритет, и это означает, что в выражении вида $\neg A \wedge B$ связка $\neg$ имеет высший приоритет и все выражение следует понимать как $(\neg A) \wedge B$, а не как

$\neg(A \wedge B)$. (Порядок следования обычных арифметических операций аналогичен: $-2+4$ равно 2, а не -6.) При необходимости для уточнения действительной структуры высказывания и упрощения его понимания можно также использовать круглые скобки, а если их недостаточно, то и квадратные скобки.

Высказывание → *АтомарноеВысказывание* | *СложноеВысказывание*
АтомарноеВысказывание → *Истина* | *Ложь* | P | Q | R | ...
СложноеВысказывание → (*Высказывание*)
| $\neg$ *Высказывание*
| *Высказывание* $\wedge$ *Высказывание*
| *Высказывание* $\vee$ *Высказывание*
| *Высказывание* $\Rightarrow$ *Высказывание*
| *Высказывание* $\Leftrightarrow$ *Высказывание*

Порядок выполнения операций : $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$

Рис. 7.7. Грамматика высказываний в логике высказываний в форме BNF (*Backus-Naur Form* — форма Бэкуса–Наура) наряду с порядком выполнения операторов от высшего приоритета к низшему

7.4.2. Семантика

Определив синтаксис пропозициональной логики, можно приступить к определению ее семантики. Семантика задает правила определения истинности высказывания по отношению к конкретной модели. В логике высказываний любая модель просто фиксирует ▶ **истинностное значение** — *true* или *false* — для каждого пропозиционального символа. Например, если в высказываниях некоторой базы знаний используются пропозициональные символы $P_{1,2}$, $P_{2,2}$ и $P_{3,1}$, то одной из возможных моделей будет

$$m_1 = \{P_{1,2} = false, P_{2,2} = false, P_{3,1} = true\}.$$

При наличии трех пропозициональных символов существует $2^3 = 8$ возможных моделей; именно столько моделей показано на рис. 7.5. Однако следует отметить, что с тех пор, как мы определили синтаксис, модели стали чисто математическими объектами, которые не обязательно должны быть связаны с миром вампуса. Например, $P_{1,2}$ — это просто символ; он может означать, что "в квадрате [1,2] есть яма" или "я буду в Париже сегодня и завтра".

Семантика пропозициональной логики должна определять, как следует вычислять истинностное значение любого высказывания при наличии модели. Эта процедура выполняется рекурсивно. Все высказывания формируются из атомарных

высказываний и пяти связок, поэтому необходимо сначала указать, как следует вычислять истинность атомарных высказываний, а затем — как вычислять истинность высказываний, сформированных с помощью каждой из этих пяти связок. Задача вычисления истинности атомарных высказываний, как показано ниже, очень проста.

- Высказывание *True* истинно в любой модели, а высказывание *False* ложно в любой модели.
- Истинностное значение любого другого пропозиционального символа должно быть указано непосредственно в модели. Например, в приведенной выше модели m_1 высказывание $P_{1,2}$ является ложным.

Для сложных высказываний существует пять правил, которые справедливы для любых входящих в них высказываний P и Q (атомарных или сложных) в любой модели m.

- $\neg P$ истинно тогда и только тогда, когда P ложно в m.
- $P \wedge Q$ истинно тогда и только тогда, когда и P, и Q истинны в m.
- $P \vee Q$ истинно тогда и только тогда, когда либо P, либо Q истинны в m.
- $P \Rightarrow Q$ истинно за исключением случая, когда P истинно, а Q ложно в m.
- $P \Leftrightarrow Q$ истинно тогда и только тогда, когда P и Q оба истинны или оба ложны в m.

Эти правила также могут быть выражены с помощью ▶**таблиц истинности**, определяющих значение истинности сложного высказывания для каждого возможного присваивания значений истинности его компонентам. Истинностные таблицы для рассматриваемых пяти логических связок приведены на рис. 7.8. На основании этих таблиц истинность любого высказывания s может быть вычислена по отношению к любой модели m посредством простой рекурсивной оценки. Например, высказывание $\neg P_{1,2} \wedge (P_{2,2} \vee P_{3,1})$, оцениваемое в модели m_1, вычисляется как $true \wedge (false \vee true) = true \wedge true = true$. В упражнении 7.3 вам будет предложено написать алгоритм PL-TRUE?(s, m), обеспечивающий в логике высказываний вычисление истинностного значения любого высказывания s в модели m.

Истинностные таблицы для связок "И", "Или" и "Нет" почти полностью соответствуют интуитивным представлениям о смысле таких же слов естественного языка. Основным источником возможной путаницы является то, что высказывание $P \vee Q$ истинно, если истинным является P или Q или *оба* эти высказывания. Имеется другая связка, называемая "исключающее ИЛИ" (сокращенно "XOR" — *Exclusive OR*), принимающая ложное значение, если оба дизъюнкта являются истинными.[8] В отношении того, какое обозначение следует применять для связки "исключающее ИЛИ", нет общего согласия: возможными вариантами являются $\dot{\vee}$, $\neq$, $\oplus$.

[8] В латинском языке для обычного, "включающего ИЛИ" используется слово "vel", тогда как для понятия "исключающего ИЛИ" имеется отдельное слово, "aut".

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
false	*false*	*true*	*false*	*false*	*true*	*true*
false	*true*	*true*	*false*	*true*	*true*	*false*
true	*false*	*false*	*false*	*true*	*false*	*false*
true	*true*	*false*	*true*	*true*	*true*	*true*

Рис. 7.8. Истинностные таблицы для пяти логических связок. Чтобы воспользоваться этой таблицей, например для вычисления значения $P \vee Q$, когда P является истинным, а Q — ложным, вначале необходимо найти в левой части таблицы строку, в которой P имеет значение *true*, а Q — *false* (третья строка). Затем нужно искать запись в этой строке, соответствующую столбцу $P \vee Q$, чтобы определить результат — *true*

Истинностная таблица для связки $\Rightarrow$ на первый взгляд может показаться озадачивающей, поскольку не совсем соответствует интуитивному пониманию выражений "P влечет за собой Q" и "если P, то Q". Прежде всего необходимо отметить, что пропозициональная логика не требует, чтобы между высказываниями P и Q устанавливались какие-либо *причинно-следственные* отношения или отношения, определяющие их *релевантность* применительно к друг другу. Так, высказывание "то, что число 5 нечетное, влечет за собой то, что Токио — столица Японии" в логике высказываний является истинным (при обычной интерпретации), даже несмотря на то, что в естественном языке оно, определенно, кажется странным. Еще один источник путаницы состоит в том, что любая импликация является истинной, если ее антецедент ложен. Например, высказывание "то, что число 5 четное, влечет за собой то, что Сэм умен" является истинным независимо от того, умен Сэм или нет. Это может показаться странным, но приобретает смысл, если рассматривать высказывание вида "$P \Rightarrow Q$" как утверждение: "Если P истинно, то я утверждаю, что Q истинно. В противном случае я не высказываю никаких утверждений". Единственный вариант, при котором это высказывание может принять значение *false*, — когда высказывание P является истинным, а Q — ложным.

Высказывание с двухсторонней импликацией, $P \Leftrightarrow Q$, будет истинным, если истинны оба высказывания $P \Rightarrow Q$ и $Q \Rightarrow P$. В словесной форме соответствующее высказывание часто записывают следующим образом: "P (истинно) тогда и только тогда, когда (истинно) Q". Многие правила для мира вампуса лучше всего записывать с помощью связки $\Leftrightarrow$. Например, в некотором квадрате чувствуется ветерок, *если* в соседнем квадрате имеется яма, а ветерок в некотором квадрате может чувствоваться, *только если* в одном из соседних квадратов имеется яма. Поэтому здесь нам необходима двухсторонняя импликация

$$B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1}),$$

где $B_{1,1}$ означает, что в квадрате [1,1] чувствуется ветерок.

7.4.3. Простая база знаний

Теперь, после определения семантики логики высказываний, можно сформировать базу знаний для мира вампуса. В этом разделе для упрощения будут рассматриваться только *неизменяемые* аспекты этого мира, а к тем аспектам, которые могут изменяться, мы вернемся в последующих разделах. На данный момент для каждого квадрата $[x, y]$ нам потребуются следующие пропозициональные символы.

Высказывание $P_{x,y}$ является истинным, если в квадрате $[x, y]$ есть яма.

Высказывание $W_{x,y}$ является истинным, если в квадрате $[x, y]$ есть вампус, мертвый или живой.

Высказывание $B_{x,y}$ является истинным, если в квадрате $[x, y]$ есть ветерок.

Высказывание $S_{x,y}$ является истинным, если в квадрате $[x, y]$ есть неприятный запах.

Высказывание $L_{x,y}$ является истинным, если в квадрате $[x, y]$ находится агент.

Высказываний, которые мы поместим в базу знаний, будет достаточно, чтобы вывести высказывания типа $\neg P_{1,2}$ (в квадрате [1,2] нет ямы), как это было неформально сделано в разделе 7.3. Каждое из этих высказываний мы пометим как R_i, чтобы на них можно было ссылаться.

- В квадрате [1,1] ямы нет:

$$R_1 : \quad \neg P_{1,1}.$$

- В квадрате чувствуется ветерок тогда и только тогда, когда в соседнем квадрате имеется яма. Такое высказывание должно быть сформулировано для каждого квадрата, но на данный момент в рассмотрение включены только непосредственно интересующие нас квадраты:

$$R_2 : \quad B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1});$$
$$R_3 : \quad B_{2,1} \Leftrightarrow (P_{1,1} \vee P_{2,2} \vee P_{3,1}).$$

- Приведенные выше высказывания будут истинными во всех экземплярах мира вампуса. Теперь добавим данные о восприятии ветерка для первых двух квадратов, которые были посещены агентом в том конкретном мире, где он находится, — это приведет нас к ситуации, показанной на рис. 7.3, *б*:

$$R_4 : \quad \neg B_{1,1};$$
$$R_5 : \quad B_{2,1}.$$

Таким образом, на текущий момент база знаний состоит из высказываний R_1–R_5 и все отдельно взятые высказывания в ней являются истинными.

7.4.4. Простая процедура логического вывода

Напомним, что сейчас наша цель — установить, будет ли $KB \models \alpha$ для некоторого высказывания α. Например, следует ли из базы знаний высказывание $\neg P_{1,2}$? Первый алгоритм логического вывода, рассматриваемый в этом разделе, представляет собой непосредственную реализацию определения логического следствия: перебрать (перечислить) все модели и проверить, является ли высказывание α истинным в каждой модели, в которой база знаний KB является истинной. Для логики высказываний модели представляют собой варианты присваивания значений *true* или *false* каждому пропозициональному символу. Для нашего примера с миром вампуса соответствующими пропозициональными символами являются $B_{1,1}$, $B_{2,1}$, $P_{1,1}$, $P_{1,2}$, $P_{2,1}$, $P_{2,2}$ и $P_{3,1}$. Для семи символов может существовать $2^7 = 128$ возможных моделей и только в трех из них база знаний KB является истинной (рис. 7.9). В этих трех моделях также истинным является высказывание $\neg P_{1,2}$, а это значит, что в квадрате [1,2] ямы нет. С другой стороны, высказывание $P_{2,2}$ истинно в двух из этих трех моделей и ложно в одной из них, поэтому мы пока не можем определить, имеется ли яма в квадрате [2,2].

$B_{1,1}$	$B_{2,1}$	$P_{1,1}$	$P_{1,2}$	$P_{2,1}$	$P_{2,2}$	$P_{3,1}$	R_1	R_2	R_3	R_4	R_5	KB
false	*false*	*false*	*false*	*false*	*false*	*false*	*true*	*true*	*true*	*true*	*false*	*false*
false	*false*	*false*	*false*	*false*	*false*	*true*	*true*	*true*	*false*	*true*	*false*	*false*
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
false	*true*	*false*	*false*	*false*	*false*	*false*	*true*	*true*	*false*	*true*	*true*	*false*
false	*true*	*false*	*false*	*false*	*false*	*true*	*true*	*true*	*true*	*true*	*true*	*true*
false	*true*	*false*	*false*	*false*	*true*	*false*	*true*	*true*	*true*	*true*	*true*	*true*
false	*true*	*false*	*false*	*false*	*true*	*true*	*true*	*true*	*true*	*true*	*true*	*true*
false	*true*	*false*	*false*	*true*	*false*	*false*	*true*	*false*	*false*	*true*	*true*	*false*
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
true	*true*	*true*	*true*	*true*	*true*	*true*	*false*	*true*	*true*	*false*	*true*	*false*

Рис. 7.9. Истинностная таблица, сформированная для базы знаний, определенной в тексте раздела. База знаний KB является истинной, если истинны высказывания R_1–R_5, а это имеет место только в 3 из 128 строк (подчеркнуты в крайнем справа столбце). Во всех этих трех строках высказывание $P_{1,2}$ ложно, поэтому в квадрате [1,2] ямы нет. С другой стороны, яма в квадрате [2,2] может быть (или не быть)

На рис. 7.9 в более точной форме воспроизведен процесс формирования рассуждений, проиллюстрированный на рис. 7.5. Общий алгоритм определения логического следствия в логике высказываний приведен на рис. 7.10. Как и в алгоритме BACKTRACKING-SEARCH, приведенном на рис. 6.5, функция TT-ENTAILS? осуществляет рекурсивный перебор конечного пространства вариантов присваивания

значений переменным. Этот алгоритм является **непротиворечивым**, поскольку он непосредственно реализует определение логического следствия, и **полным**, поскольку может применяться для любой базы знаний *KB* и любого высказывания α и всегда заканчивает свою работу, — при условии, что количество моделей, подлежащих проверке, является конечным.

```
function TT-ENTAILS?(KB, α) returns значение true или false
  inputs: KB, база знаний, высказывание в логике высказываний
          α, запрос, высказывание в логике высказываний

  symbols ← список пропозициональных символов в KB и α
  return TT-CHECK-ALL(KB, α, symbols, {})

function TT-CHECK-ALL(KB, α, symbols, model) returns значение true или false
  if EMPTY?(symbols) then
    if PL-TRUE?(KB, model) then return PL-TRUE?(α, model)
    else return true        // Если KB ложна, всегда возвращается значение true
  else
    P ← FIRST(symbols)
    rest ← REST(symbols)
    return (TT-CHECK-ALL(KB, α, rest, model ∪ {P = true})
            and
            TT-CHECK-ALL(KB, α, rest, model ∪ {P = false}))
```

Рис. 7.10. Алгоритм перебора истинностной таблицы для получения пропозициональных логических следствий. (Здесь TT представляет истинностную таблицу.) Функция PL-TRUE? возвращает значение *true*, если некоторое высказывание является истинным в рамках некоторой модели. Переменная *model* представляет частично заданную модель — присваивание только некоторых переменных. В этом алгоритме ключевое слово **and** является обозначением инфиксной функции языка псевдокода, а не оператором логики высказываний. Эта функция принимает два аргумента и возвращает значение *true* или *false*

Безусловно, выражение "является конечным" не всегда означает то же, что и выражение "является небольшим". Если *KB* и α содержат в целом n символов, то количество моделей будет равно 2^n. Таким образом, временная сложность этого алгоритма составляет $O(2^n)$. (Пространственная сложность составляет только $O(n)$, поскольку перебор вариантов присваивания происходит по принципу поиска в глубину.) Ниже будут приведены алгоритмы, гораздо более эффективные на практике. К сожалению, логический вывод высказываний является co-NP-сложной задачей (т.е. вероятно, не проще NP-сложных задач; см. приложение А), ➡ *поэтому любой известный алгоритм логического вывода для логики высказываний в худшем случае имеет сложность, экспоненциально зависящую от размера ввода.*

7.5. Доказательство теорем логики высказываний

На данный момент уже было показано, как определить логическое следствие посредством *проверки моделей*: последовательно перебирая модели и показывая, что высказывание должно присутствовать во всех моделях. В этом разделе будет показано, как логическое следствие может быть сделано посредством ▶**доказательства теоремы** — применяя правила логического вывода непосредственно к высказываниям в базе данных с целью построения доказательства истинности требуемого высказывания без обращения к моделям. Если число моделей велико, а длина доказательства ограничена, то доказательство теоремы может оказаться более эффективным методом в сравнении с проверкой моделей.

Прежде чем углубиться в детали алгоритмов доказательства теорем, необходимо ввести некоторые дополнительные концепции, касающиеся логического следствия. Первой из них является понятие ▶**логической эквивалентности**: два высказывания, α и β, являются логически эквивалентными, если они истинны в одном и том же множестве моделей. Это утверждение записывается как $\alpha \equiv \beta$. (Обратите внимание, что знак $\equiv$ будет использоваться для утверждений в отношении высказываний, в то время как знак $\Leftrightarrow$ будет использоваться только для частей высказываний.) Например, можно легко показать (с помощью истинностных таблиц), что высказывания $P \wedge Q$ и $Q \wedge P$ логически эквивалентны. Другие стандартные эквивалентности приведены на рис. 7.11. В логике они играют практически такую же роль, какую арифметические равенства играют в обычной математике. Альтернативное определение эквивалентности является следующим: любые два высказывания α и β являются эквивалентными тогда и только тогда, когда каждое из них является логическим следствием другого:

$$\alpha \equiv \beta \text{ тогда и только тогда, когда } \alpha \models \beta \text{ и } \beta \models \alpha.$$

Вторым понятием, которое нам потребуется, является ▶**допустимость**. Высказывание допустимо, если оно истинно во *всех* моделях. Например, высказывание $P \vee \neg P$ является допустимым. Допустимые высказывания известны также под названием ▶**тавтологий** — они *обязательно* истинны. Поскольку высказывание *True* является истинным во всех моделях, то любое допустимое высказывание логически эквивалентно *True*. Для чего могут применяться допустимые высказывания? Из определения логического следствия можно вывести ▶**теорему дедукции**, которая была известна еще в Древней Греции:

▶*Для любых высказываний* α *и* β, $\alpha \models \beta$, *если и только если высказывание* $(\alpha \Rightarrow \beta)$ *является допустимым.*

(В упражнении 7.6 предлагается доказать эту теорему.) Следовательно, можно определить, будет ли $\alpha \models \beta$, как посредством проверки, что $(\alpha \Rightarrow \beta)$ истинно в любой модели (именно это выполняется в алгоритме логического вывода,

приведенном на рис. 7.10), так и доказав, что $(\alpha \Rightarrow \beta)$ эквивалентно *True*. И наоборот, теорема дедукции утверждает, что каждое допустимое высказывание в форме импликации описывает приемлемый вариант логического вывода.

$$(\alpha \wedge \beta) \equiv (\beta \wedge \alpha) \quad \text{коммутативность связки } \wedge$$
$$(\alpha \vee \beta) \equiv (\beta \vee \alpha) \quad \text{коммутативность связки } \vee$$
$$((\alpha \wedge \beta) \wedge \gamma) \equiv (\alpha \wedge (\beta \wedge \gamma)) \quad \text{ассоциативность связки } \wedge$$
$$((\alpha \vee \beta) \vee \gamma) \equiv (\alpha \vee (\beta \vee \gamma)) \quad \text{ассоциативность связки } \vee$$
$$\neg(\neg\alpha) \equiv \alpha \quad \text{устранение двойного отрицания}$$
$$(\alpha \Rightarrow \beta) \equiv (\neg\beta \Rightarrow \neg\alpha) \quad \text{контрапозиция}$$
$$(\alpha \Rightarrow \beta) \equiv (\neg\alpha \vee \beta) \quad \text{устранение импликации}$$
$$(\alpha \Leftrightarrow \beta) \equiv ((\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)) \quad \text{устранение двухсторонней импликации}$$
$$\neg(\alpha \wedge \beta) \equiv (\neg\alpha \vee \neg\beta) \quad \text{правило де Моргана}$$
$$\neg(\alpha \vee \beta) \equiv (\neg\alpha \wedge \neg\beta) \quad \text{правило де Моргана}$$
$$(\alpha \wedge (\beta \vee \gamma)) \equiv ((\alpha \wedge \beta) \vee (\alpha \wedge \gamma)) \quad \text{дистрибутивность связки } \wedge \text{ по } \vee$$
$$(\alpha \vee (\beta \wedge \gamma)) \equiv ((\alpha \vee \beta) \vee (\alpha \wedge \gamma)) \quad \text{дистрибутивность связки } \vee \text{ по } \wedge$$

Рис. 7.11. Стандартные логические эквивалентности. Символы α, β и γ обозначают произвольные высказывания логики высказываний

Последним понятием, которое нам потребуется, является ▶**выполнимость**. Некоторое высказывание выполнимо тогда и только тогда, когда оно истинно в *некоторой* модели. Например, приведенная выше база знаний, $(R_1 \wedge R_2 \wedge R_3 \wedge R_4 \wedge R_5)$, выполнима, поскольку существует даже не одна, а три модели, в которых она истинна (см. рис. 7.9). Выполнимость можно проверить, перебирая все возможные модели до тех пор, пока не будет найдена хотя бы одна из них, которая выполняет данное высказывание. Задача определения выполнимости высказываний в пропозициональной логике — ▶**задача SAT** — была первой задачей, для которой было доказано, что она является NP-полной. Многие задачи в компьютерных науках в действительности представляют собой задачи определения выполнимости. Например, во всех задачах удовлетворения ограничений, приведенных в главе 6, по сути, требовалось найти ответ на вопрос о том, выполнимы ли данные ограничения при некотором присваивании.

Безусловно, понятия допустимости и выполнимости связаны друг с другом: высказывание α является допустимым тогда и только тогда, когда высказывание $\neg\alpha$ невыполнимо, и наоборот, высказывание α является выполнимым тогда и только тогда, когда высказывание $\neg\alpha$ недопустимо. На этом основании можно также получить следующий полезный результат:

▶ $\alpha \models \beta$ *тогда и только тогда, когда высказывание* $(\alpha \wedge \neg\beta)$ *невыполнимо.*

Доказательство истинности высказывания β на основании истинности высказывания α путем проверки невыполнимости выражения $(\alpha \wedge \neg\beta)$ точно соответствует

стандартному математическому методу доказательства путем *приведения к абсурду* (буквально "доведение до абсурда" — *reductio ad absurdum*). Его также называют доказательством путем ▶**опровержения** или доказательством с помощью ▶**выявления противоречия**. В этом методе доказательства принимается предположение, что высказывание β ложно, и демонстрируется, что такое предположение приводит к противоречию с известными аксиомами α. Подобное противоречие точно соответствует тому, что подразумевается под утверждением, что выражение $(\alpha \wedge \neg\beta)$ невыполнимо.

7.5.1. Логический вывод и доказательства

В данном разделе рассматриваются стандартные ▶**правила логического вывода**, которые могут применяться для формирования ▶**доказательства** — цепочки заключений, ведущих к желаемой цели. Наиболее широко известное правило называется ▶**правилом отделения** (на латыни *Modus Ponens* — "модус поненс") и записывается следующим образом:

$$\frac{\alpha \Rightarrow \beta,\ \alpha}{\beta}.$$

Эта запись означает, что если даны любые высказывания в форме $\alpha \Rightarrow \beta$ и α, то из них можно вывести высказывание β. Например, если дано (*WumpusAhead* $\wedge$ *WumpusAlive*) $\Rightarrow$ *Shoot* и (*WumpusAhead* $\wedge$ *WumpusAlive*), то можно вывести высказывание *Shoot*.

Еще одним полезным правилом логического вывода является правило ▶**удаления связки "И"**, в котором утверждается, что из конъюнкции можно вывести любой из ее конъюнктов:

$$\frac{\alpha \wedge \beta}{\alpha}.$$

Например, из высказывания (*WumpusAhead* $\wedge$ *WumpusAlive*) можно вывести высказывание *WumpusAlive*.

Рассматривая возможные истинностные значения α и β, можно легко показать, что правило отделения и правило удаления связки "И" являются непротиворечивыми не только применительно к данным примерам, но и ко всем возможным высказываниям. Это значит, что данные правила могут использоваться в любых конкретных случаях, где они применимы, для выработки непротиворечивых логических выводов без необходимости перебора всех моделей.

В качестве правил логического вывода можно также применять все логические эквивалентности, приведенные на рис. 7.11. Например, из эквивалентности устранения двухсторонней импликации можно получить следующие два правила логического вывода:

$$\frac{\alpha \Leftrightarrow \beta}{(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)} \quad \text{и} \quad \frac{(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)}{\alpha \Leftrightarrow \beta}.$$

Однако не все правила логического вывода действуют в обоих направлениях, как это. Например, нельзя применить правило отделения в противоположном направлении, чтобы получить высказывания $\alpha \Rightarrow \beta$ и α из высказывания β.

Давайте посмотрим, как правила логического вывода и эквивалентности могут быть использованы в мире вампуса. Начнем с базы знаний, содержащей высказывания R_1–R_5, и покажем, как доказать высказывание $\neg P_{1,2}$, т.е. утверждение, что в квадрате [1,2] нет ямы.

1. Вначале применим правило устранения двухсторонней импликации к высказыванию R_2, чтобы получить следующее:

$$R_6: \ (B_{1,1} \Rightarrow (P_{1,2} \vee P_{2,1})) \wedge ((P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1}).$$

2. Затем применим к высказыванию R_6 правило удаления связки "И" и получим

$$R_7: \ ((P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1}).$$

3. Из логической эквивалентности отрицаний следует

$$R_8: \ (\neg B_{1,1} \Rightarrow \neg(P_{1,2} \vee P_{2,1})).$$

4. Теперь можно применить правило отделения к высказыванию R_8 и к высказыванию R_4 с данными восприятия (т.е. $\neg B_{1,1}$), чтобы получить следующее:

$$R_9: \ \neg(P_{1,2} \vee P_{2,1}).$$

5. Наконец, применим правило де Моргана, позволяющее получить такое заключение:

$$R_{10}: \ \neg P_{1,2} \wedge \neg P_{2,1}.$$

Это означает, что ни в квадрате [1,2], ни в квадрате [2,1] ямы нет.

Любой из алгоритмов поиска, рассматривавшихся в главе 3, можно использовать для поиска последовательности шагов, составляющих подобное доказательство. Достаточно будет просто определить задачу доказательства следующим образом.

- INITIAL STATE — исходная база знаний.
- ACTIONS — множество действий состоит из всех правил вывода, применяемых ко всем высказываниям, которые соответствуют верхней половине правила логического вывода.
- RESULT — результатом некоторого действия является добавление высказывания в нижнюю половину правила вывода.
- GOAL — целью является состояние, содержащее высказывание, которое предполагается доказать.

Таким образом, поиск доказательств можно рассматривать как альтернативу перебору всех моделей. Во многих практических случаях ➡ *поиск доказательства может оказаться более эффективным методом, поскольку в доказательстве можно игнорировать нерелевантные высказывания независимо от того, как много их имеется.* Например, в приведенном выше доказательстве, ведущем к высказыванию $\neg P_{1,2} \wedge \neg P_{2,1}$, не упоминались высказывания $B_{2,1}$, $P_{1,1}$, $P_{2,2}$ и $P_{3,1}$. Их можно было не рассматривать, поскольку целевое высказывание, $P_{1,2}$ присутствует только в высказывании R_4, а остальные высказывания из состава R_4 присутствуют только в R_4 и R_2, поэтому высказывания R_1, R_3 и R_5 не имеют никакого отношения к доказательству. Те же рассуждения останутся справедливыми и в том случае, если в базу знаний будет введено на миллион больше высказываний, тогда как простой алгоритм перебора строк в истинностной таблице в такой ситуации был бы буквально раздавлен из-за экспоненциального увеличения количества просматриваемых моделей.

Последним фундаментальным свойством логических систем является ➡**монотонность**, согласно которому, множество высказываний, которые могут быть получены посредством логического вывода, будет *увеличиваться* лишь по мере добавления к базе знаний новой информации.[9] Для любых высказываний α и β справедливо следующее:

$$\text{если} \quad KB \models \alpha, \quad \text{то} \quad KB \wedge \beta \models \alpha.$$

Например, предположим, что база знаний содержит дополнительное утверждение β, согласно которому в этом экземпляре мира вампуса имеется точно восемь ям. Эти знания могут помочь агенту прийти к *дополнительным* заключениям, но не способны поставить под сомнение какое-либо уже сделанное заключение α, такое как вывод о том, что в квадрате [1,2] нет ямы. Монотонность означает, что правила логического вывода могут применяться каждый раз, когда в базе знаний обнаруживаются подходящие предпосылки, — полученное заключение будет следствием из данного правила, *независимо от того, что еще находится в базе знаний.*

7.5.2. Доказательство методом резолюций

Выше было показано, что все рассматривавшиеся до сих пор правила логического вывода являются *непротиворечивыми*, но вопрос об их *полноте* применительно к алгоритмам логического вывода, в которых они используются, еще не обсуждался. Алгоритмы поиска, такие как поиск с итеративным углублением (раздел 3.4.4), являются полными в том смысле, что позволяют найти любую достижимую цель, но если доступные правила логического вывода неадекватны, то цель становится недостижимой, — не существует доказательства, в котором

[9] Немонотонные логики, в которых нарушается свойство монотонности, отражают типичную черту человеческого мышления — способность менять свое мнение. Эти варианты логики рассматриваются в разделе 10.6.

могли бы применяться только эти правила логического вывода. Например, если мы откажемся от использования правила удаления двухсторонней импликации, то не сможем довести до конца доказательство, изложенное в предыдущем разделе. В этом разделе будет представлено единственное правило логического вывода, правило **резолюций**, позволяющее получить алгоритм логического вывода, который становится полным в сочетании с любым полным алгоритмом поиска.

Начнем с использования простой версии правила резолюций в мире вампуса. Рассмотрим этапы, ведущие вверх на рис. 7.4, *а*: агент возвращается из квадрата [2,1] в квадрат [1,1], а затем переходит в квадрат [1,2], где его восприятие обнаруживает неприятный запах, но не отмечает ветерка. Введем в базу знаний следующие дополнительные факты:

$$R_{11}: \ \neg B_{1,2};$$
$$R_{12}: \ B_{1,2} \Leftrightarrow (P_{1,1} \vee P_{2,2} \vee P_{1,3}).$$

С помощью того же процесса, который привел к получению высказывания R_{10}, приведенного выше, мы теперь можем сделать заключение об отсутствии ям в квадратах [2,2] и [1,3] (напомним, что в отношении квадрата [1,1] уже известно, что ямы в нем нет):

$$R_{13}: \ \neg P_{2,2};$$
$$R_{14}: \ \neg P_{1,3}.$$

Кроме того, можно применить к высказыванию R_3 правило удаления двухсторонней импликации, после чего применить к полученному результату в сочетании с высказыванием R_5 правило отделения, чтобы установить тот факт, что по меньшей мере в одном из квадратов [1,1], [2,2] и [3,1] яма есть:

$$R_{15}: \ P_{1,1} \vee P_{2,2} \vee P_{3,1}.$$

Теперь у нас появилась первая возможность воспользоваться правилом резолюций (правилом устранения противоречия): литерал $\neg P_{2,2}$ в высказывании R_{13}, противоположный литералу $P_{2,2}$ в высказывании R_{15}, устраняется, что приводит к получению следующей ▶**резольвенты**:

$$R_{16}: \ P_{1,1} \vee P_{3,1}.$$

На обычном языке выразить ход этих рассуждений можно следующим образом: если по меньшей мере в одном из квадратов [1,1], [2,2] и [3,1] имеется яма, но ее нет в квадрате [2,2], то яма должна быть в квадрате [1,1] или [3,1]. Аналогичным образом устраняется литерал $\neg P_{1,1}$ в высказывании R_1, противоположный литералу $P_{1,1}$ в высказывании R_{16}, что приводит к получению высказывания

$$R_{17}: \ P_{3,1}.$$

Словами ход этих рассуждений можно выразить так: если в одном из квадратов [1,1] и [3,1] есть яма, но ее нет в квадрате [1,1], то она находится в квадрате [3,1].

Эти два последних этапа логического вывода представляют собой примеры правила логического вывода, называемого ▶**единичной резолюцией**:

$$\frac{\ell_1 \vee \cdots \vee \ell_k, \qquad m}{\ell_1 \vee \cdots \vee \ell_{i-1} \vee \ell_{i+1} \vee \cdots \vee \ell_k},$$

где каждое из выражений ℓ представляет собой литерал, а выражения ℓ_i и m являются ▶**взаимно обратными литералами** (т.е. один из них является отрицанием другого). Таким образом, в правиле единичной резолюции берется ▶**выражение** (дизъюнкция литералов) и еще один литерал, после чего формируется новое выражение. Обратите внимание, что этот единственный литерал может рассматриваться как дизъюнкция из одного литерала, называемая также ▶**единичным выражением**.

Правило единичной резолюции может быть также обобщено до полного ▶**правила резолюций**:

$$\frac{\ell_1 \vee \cdots \vee \ell_k, \qquad m_1 \vee \cdots \vee m_n}{\ell_1 \vee \cdots \vee \ell_{i-1} \vee \ell_{i+1} \vee \cdots \vee \ell_k \vee m_1 \vee \cdots \vee m_{j-1} \vee m_{j+1} \vee \cdots \vee m_n},$$

где ℓ_i и m_j — взаимно обратные литералы. Это означает, что в правиле резолюций берутся два выражения и вырабатывается новое выражение, содержащее все литералы двух первоначальных выражений, *кроме* двух взаимно обратных литералов. Например, может иметь место такой логический вывод:

$$\frac{P_{1,1} \vee P_{3,1}, \qquad \neg P_{1,1} \vee \neg P_{2,2}}{P_{3,1} \vee \neg P_{2,2}}.$$

За один раз можно устранить только одну пару взаимно обратных литералов. Например, можно устранить P и $\neg P$, чтобы вывести

$$\frac{P \vee \neg Q \vee R, \qquad \neg P \vee Q}{\neg Q \vee Q \vee R},$$

но невозможно одновременно устранить и P, и Q, чтобы вывести R. Применение правила резолюций предполагает выполнение еще одного формального требования: результирующее высказывание должно содержать только по одной копии каждого литерала.[10] Операция удаления дополнительных копий литералов называется ▶**факторизацией**. Например, после удаления взаимно обратных литералов в выражениях $(A \vee B)$ и $(A \vee \neg B)$ будет получено выражение $(A \vee A)$, которое следует сократить до A посредством факторизации.

[10] Если некоторое выражение рассматривается как множество литералов, то такое правило сокращения применяется автоматически. Использование для логических выражений системы обозначений в виде множеств позволяет сделать правило резолюций более понятным, но за счет введения дополнительных обозначений.

Непротиворечивость правила резолюции можно легко доказать, рассматривая литерал ℓ_i, взаимно обратный литералу m_j в другом выражении. Если литерал ℓ_i является истинным, то литерал m_j является ложным, и поэтому выражение $m_1 \vee \cdots \vee m_{j-1} \vee m_{j+1} \vee \cdots \vee m_n$ должно быть истинным, поскольку дано, что выражение $m_1 \vee \cdots \vee m_n$ истинно. Если литерал ℓ_i является ложным, то должно быть истинным выражение $\ell_1 \vee \cdots \vee \ell_{i-1} \vee \ell_{i+1} \vee \cdots \vee \ell_k$, поскольку дано, что истинно выражение $\ell_1 \vee \cdots \vee \ell_k$. Значит, литерал ℓ_i является либо истинным, либо ложным, поэтому справедливо одно или второе из этих заключений, — именно это и утверждается в правиле резолюции.

Еще более удивительное свойство правила резолюции состоит в том, что оно образует основу для семейства *полных* процедур логического вывода. ➡ *Любой алгоритм доказательства теорем, основанный на правиле резолюции, позволяет определить для любых высказываний* α *и* β *пропозициональной логики истинность утверждения* $\alpha \models \beta$. В следующих двух подразделах описано, как правило резолюций может использоваться для этой цели.

Конъюнктивная нормальная форма

Правило резолюций применяется только к выражениям (т.е. к дизъюнкциям литералов), поэтому на первый взгляд оно применимо только к базам знаний и запросам, состоящим из таких дизъюнкций. На каком же основании мы утверждаем, что это правило может служить основой процедуры полного логического вывода для всей логики высказываний? Ответ на этот вопрос состоит в том, что ➡ *каждое высказывание логики высказываний логически эквивалентно конъюнкции выражений*.

Любое высказывание, представленное как конъюнкция выражений (т.е. дизъюнкций литералов), называется высказыванием в ▶**конъюнктивной нормальной форме**, **КНФ** (рис. 7.12) или ▶**CNF** (*Conjunctive Normal Form*). Опишем простую процедуру подобного преобразования, для чего проиллюстрируем ее на преобразовании высказывания $B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$ в форму КНФ. Ниже описаны соответствующие этапы.

1. Исключаем связку $\Leftrightarrow$, заменив высказывание $\alpha \Leftrightarrow \beta$ высказыванием $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$:

$$(B_{1,1} \Rightarrow (P_{1,2} \vee P_{2,1})) \wedge ((P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1}).$$

2. Исключаем связку $\Rightarrow$, заменив высказывание $\alpha \Rightarrow \beta$ высказыванием $\neg\alpha \vee \beta$:

$$(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg(P_{1,2} \vee P_{2,1}) \vee B_{1,1}).$$

3. Конъюнктивная нормальная форма требует, чтобы связка $\neg$ появлялась только перед литералами, поэтому, как принято называть эту операцию, "введем связку $\neg$ внутрь выражения", применив, где необходимо, следующие эквивалентности, приведенные на рис. 7.11:

$$\neg(\neg\alpha) \equiv \alpha \qquad \text{(устранение двойного отрицания)}$$
$$\neg(\alpha \wedge \beta) \equiv (\neg\alpha \vee \neg\beta) \qquad \text{(правило де Моргана)}$$
$$\neg(\alpha \vee \beta) \equiv (\neg\alpha \wedge \neg\beta) \qquad \text{(правило де Моргана)}$$

4. В данном примере требуется применение только одного, последнего, правила:

$$(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge ((\neg P_{1,2} \wedge \neg P_{2,1}) \vee B_{1,1}).$$

5. В результате получено высказывание, содержащее вложенные связки $\wedge$ и $\vee$, которые применяются к литералам. Используем закон дистрибутивности, приведенный на рис. 7.11, распределив связки $\vee$ по связкам $\wedge$ везде, где это возможно:

$$(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg P_{1,2} \vee B_{1,1}) \wedge (\neg P_{2,1} \vee B_{1,1}).$$

Теперь первоначальное высказывание представлено в форме КНФ как конъюнкция трех выражений. Это высказывание стало более сложным для чтения, но зато его можно использовать в качестве входных данных для процедуры резолюции.

$$\begin{aligned}
\textit{CNFSentence} &\rightarrow \textit{Clause}_1 \wedge \cdots \wedge \textit{Clause}_n \\
\textit{Clause} &\rightarrow \textit{Literal}_1 \vee \cdots \vee \textit{Literal}_m \\
\textit{Fact} &\rightarrow \textit{Symbol} \\
\textit{Literal} &\rightarrow \textit{Symbol} \mid \neg\textit{Symbol} \\
\textit{Symbol} &\rightarrow P \mid Q \mid R \mid \ldots \\
\textit{HornClauseForm} &\rightarrow \textit{DefiniteClauseForm} \mid \textit{GoalClauseForm} \\
\textit{DefiniteClauseForm} &\rightarrow \textit{Fact} \mid (\textit{Symbol}_1 \wedge \cdots \wedge \textit{Symbol}_l) \Rightarrow \textit{Symbol} \\
\textit{GoalClauseForm} &\rightarrow (\textit{Symbol}_1 \wedge \cdots \wedge \textit{Symbol}_l) \Rightarrow \textit{False}
\end{aligned}$$

Рис. 7.12. Грамматика конъюнктивной нормальной формы, выражений Хорна и определенных выражений. Выражение в CNF, такое как $\neg A \vee \neg B \vee C$, может быть записано в форме определенного выражения в виде $A \wedge B \Rightarrow C$

Алгоритм резолюции

Процедуры логического вывода, основанные на правиле резолюции, действуют с использованием принципа доказательства путем установления противоречия, который описывался в начале раздела 7.5. Таким образом, чтобы показать, что $KB \models \alpha$, мы покажем, что высказывание (KB $\wedge \neg\alpha$) является невыполнимым. Это можно сделать, доказав, что имеет место противоречие.

Алгоритм резолюции приведен на рис. 7.13. Вначале высказывание (KB $\wedge \neg\alpha$) преобразуется в КНФ, а затем к полученным выражениям применяется правило резолюций. В каждой паре выражений, содержащих взаимно противоположные

литералы, происходит удаление этих противоположных друг другу литералов для получения нового выражения, которое добавляется к множеству существующих выражений, если в этом множестве еще нет такого выражения. Указанный процесс продолжается до тех пор, пока не происходит одно из следующих двух событий:

- больше не вырабатываются какие-либо новые выражения, которые можно было бы добавить к имеющимся, и в этом случае из базы знаний *KB* не следует высказывание α;
- резолюция двух выражений приводит к получению *пустого* выражения, и в этом случае из базы знаний *KB* следует высказывание α.

---

function PL-RESOLUTION(*KB*, α) **returns** значение *true* или *false*
  inputs: *KB*, база знаний, представляет собой высказывание в логике высказываний
    α, запрос, представляет собой высказывание в логике высказываний
  clauses ← множество выражений, полученное после преобразования
    высказывания $KB \wedge \neg\alpha$ в форму КНФ
  new ← {}
  while *true* **do**
    for each пары выражений C_i, C_j **in** *clauses* **do**
      resolvents ← PL-RESOLVE(C_i, C_j)
      if *resolvents* содержит пустое выражение **then return** *true*
      new ← *new* ∪ *resolvents*
    if *new* ⊆ *clauses* **then return** *false*
    clauses ← *clauses* ∪ *new*

Рис. 7.13. Простой алгоритм резолюции для логики высказываний. Функция PL-RESOLVE возвращает множество всех возможных выражений, полученных путем устранения противоположных друг другу литералов из двух высказываний, переданных ей на входе

Пустое выражение (дизъюнкция без дизъюнктов) эквивалентно высказыванию *False*, поскольку дизъюнкция является истинной, только если истинен по меньшей мере один из ее дизъюнктов. Кроме того, пустое выражение возникает только после устранения двух взаимно противоположных единичных выражений, таких как P и $\neg P$.

Эту процедуру резолюции можно применить для формирования очень простого логического вывода в мире вампуса. Когда агент находится в квадрате [1,1], он не чувствует ветерка, поэтому в соседних квадратах не может быть ям. Соответствующая база знаний имеет следующий вид:

$$KB = R_2 \wedge R_4 = (B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})) \wedge \neg B_{1,1},$$

и требуется доказать высказывание α, которое, скажем, имеет вид $\neg P_{1,2}$. После преобразования высказывания $(KB \wedge \neg\alpha)$ в форму КНФ, мы получим все выражения,

показанные в верхней части рис. 7.14. В нижней части этого рисунка показаны выражения, полученные путем устранения противоположных друг другу литералов из всех пар выражений, приведенных в верхнем ряду. В конечном счете после резолюции литерала $P_{1,2}$ и противоположного литерала $\neg P_{1,2}$ будет получено пустое выражение, представленное в виде небольшого пустого квадрата. Анализ рис. 7.14, показывает, что многие этапы резолюции были бессмысленными. Например, выражение $B_{1,1} \vee \neg B_{1,1} \vee P_{1,2}$ эквивалентно выражению *True* $\vee$ $P_{1,2}$, которое эквивалентно *True*. Логический вывод, согласно которому выражение *True* является истинным, не очень полезен. Поэтому любое выражение, в котором присутствуют два взаимно дополнительных литерала, может быть отброшено.

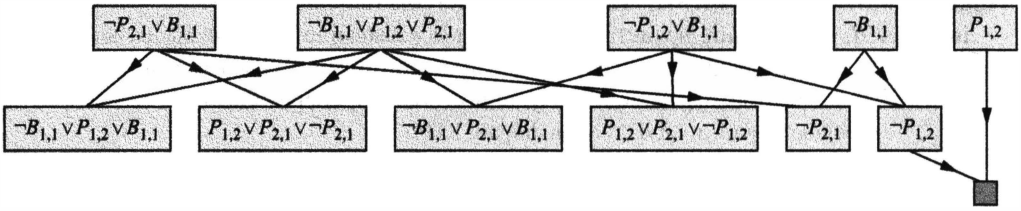


Рис. 7.14. Частичное применение функции PL-RESOLUTION для формирования простого логического вывода в мире вампуса с целью доказательства запроса $\neg P_{1,2}$. Каждое из четырех выражений слева в верхней строке образует пару с каждым из трех оставшихся, и к ним применятся правило резолюций, — результаты представлены в нижней строке. Можно видеть, что пара из третьего и четвертого выражений из верхнего ряда в результате дает выражение $\neg P_{1,2}$, которое затем подвергается резолюции с выражением $P_{1,2}$, давая в результате пустое выражение, а это означает, что истинность запроса доказана

Полнота резолюции

В завершение обсуждения правила резолюции покажем, почему алгоритм PL-RESOLUTION является полным. Для этого целесообразно ввести понятие ►**резолюционного замыкания** *RC*(*S*) множества выражений *S*, представляющего собой множество всех выражений, которые могут быть получены путем повторного применения правила резолюции к выражениям из множества *S* или к их производным. Резолюционным замыканием является множество выражений, которое вычисляется алгоритмом PL-RESOLUTION в качестве окончательного значения переменной *clauses*. Можно легко показать, что множество *RC*(*S*) должно быть конечным, поскольку, благодаря этапу факторизации, на котором уничтожаются дополнительные копии литералов, количество различных выражений, которые могут быть сформированы из символов $P_1, \ldots, P_k$, присутствующих в *S*, является конечным. Поэтому алгоритм PL-RESOLUTION всегда завершает свою работу.

Теорема полноты для правила резолюции в логике высказываний называется ►**основной теоремой резолюции**:

> Если множество выражений является невыполнимым, то резолюционное замыкание этих выражений содержит пустое выражение.

Докажем эту теорему, показав, что справедливо противоположное ей утверждение: если замыкание $RC(S)$ *не содержит* пустое выражение, то множество S выполнимо. В действительности для множества S можно создать модель с подходящими истинностными значениями для $P_1, \ldots, P_k$. Процедура создания такой модели следующая.

> Для i от 1 до k:
>
> – если выражение в множестве $RC(S)$ содержит литерал $\neg P_i$ и все другие его литералы являются ложными при данном присваивании, выбранном для $P_1, \ldots, P_k$, то литералу P_i присваиваем значение *false*;
> – в противном случае присваиваем литералу P_i значение *true*.

Это присваивание $P_1, \ldots, P_k$ является моделью для множества S. Чтобы показать это, предположим обратное — что на каком-то этапе i в процессе выполнения присваивание значения P_i приводит к тому, что некоторое выражение C получает значение *false*. Это может произойти, только если все *остальные* литералы в C уже также получили значение *false* в присваиваниях $P_1, \ldots, P_{i-1}$. Следовательно, выражение C должно в этом случае выглядеть либо как $(false \vee false \vee \cdots false \vee P_i)$, либо как $(false \vee false \vee \cdots false \vee \neg P_i)$. Если хотя бы одно из этих двух выражений присутствует в замыкании $RC(S)$, то алгоритм присвоит соответствующее значение *truth* для P_i, чтобы C стало истинным, поэтому C может получить значение *false*, только если оба эти выражения присутствуют в $RC(S)$.

Теперь, поскольку $RC(S)$ является резолюционным замыканием, оно будет содержать резольвенту этих двух выражений, и у этой резольвенты все литералы уже будут иметь значения *false* после присваиваний $P_1, \ldots, P_{i-1}$. А это противоречит нашему исходному предположению, что первое выражение со значением *false* появляется на этапе i. Таким образом, мы доказали, что предложенное построение не приведет к появлению выражения со значением *false* в замыкании $RC(S)$, т.е. оно создает модель для $RC(S)$. И наконец, поскольку множество S содержится в замыкании $RC(S)$, любая модель для $RC(S)$ представляет собой модель и для собственно S.

7.5.3. Хорновские выражения и определенные выражения

Благодаря своему свойству полноты метод резолюции становится очень важным методом логического вывода. Однако во многих практических ситуациях вся мощь правила резолюции не требуется. Некоторые реальные базы знаний содержат высказывания, форма которых отвечает определенным ограничениям, что

позволяет использовать для них более строгие и эффективные алгоритмы логического вывода.

Одной из таких ограниченных форм является ▶**определенное выражение**, состоящее из нескольких литералов, среди которых точно один является положительным. Например, выражение $(\neg L_{1,1} \vee \neg Breeze \vee B_{1,1})$ является определенным выражением, в то время как $(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1})$ — нет, потому в нем присутствуют два положительных выражения.

Чуть более общим является ▶**хорновское выражение**, которое представляет собой дизъюнкцию литералов, среди которых положительным является *не больше чем один*. Таким образом, все определенные выражения являются хорновскими выражениями, как и выражения без положительных литералов; это так называемые ▶**выражения без заголовка**. Хорновские выражения являются резолюционно замкнутыми: если применить резолюцию к двум хорновским выражениям, будет получено также хорновское выражение. Еще одним классом являются k-CNF высказывания, представляющие собой высказывания в форме КНФ, в которых каждое выражение имеет не более k литералов.

Базы знаний, содержащие только определенные выражения, интересны по трем причинам.

1. Каждое определенное выражение может быть записано как импликация, предпосылкой которой является конъюнкция положительных литералов, а заключением — один положительный литерал (см. упр. 7.16). Например, определенное выражение $(\neg L_{1,1} \vee \neg Breeze \vee B_{1,1})$ может быть записано как импликация $(L_{1,1} \wedge Breeze) \Rightarrow B_{1,1}$. В этой последней форме данное высказывание становится более легким для чтения: в нем утверждается, что если агент находится в квадрате [1,1] и чувствует ветерок, то ветерок чувствуется в квадрате [1,1]. В хорновском выражении предпосылку называют ▶**телом** выражения, а заключение (положительный литерал) — ▶**головой**. Высказывание, состоящее из одного положительного литерала, такого как $L_{1,1}$, называют ▶**фактом**. Его также можно записать в форме импликации как $True \Rightarrow L_{1,1}$, но проще написать просто $L_{1,1}$.
2. Логический вывод с использованием хорновских выражений может осуществляться с помощью алгоритма ▶**прямого логического вывода** и ▶**обратного логического вывода**, которые рассматриваются ниже. Оба эти алгоритма являются очень естественными в том смысле, что этапы логического вывода являются для людей очевидными и за ними можно легко проследить. Этот тип логического вывода является основой для **логического программирования**, которое будет обсуждаться в главе 9.
3. Получение логических следствий с помощью хорновских выражений может осуществляться за время, *линейно* зависящее от размера базы знаний. Этот последний факт является особенно приятным сюрпризом.

7.5.4. Прямой и обратный логический вывод

Алгоритм прямого логического вывода PL-FC-ENTAILS?(KB, q) определяет, следует ли единственный пропозициональный символ q — запрос — из базы знаний, представленной в форме определенных выражений. Он начинает работу с известных фактов (положительных литералов) в базе знаний. Если известны все предпосылки некоторой импликации, то ее заключение добавляется к множеству известных фактов. Например, если известны факты $L_{1,1}$ и *Breeze*, а в базе знаний имеется выражение $(L_{1,1} \wedge \text{Breeze}) \Rightarrow B_{1,1}$, то к ней можно добавить факт $B_{1,1}$. Этот процесс продолжается до тех пор, пока запрос q не будет добавлен к базе знаний или осуществление дальнейшего логического вывода станется невозможным. Этот алгоритм приведен на рис. 7.15, и главное, что следует о нем помнить, — это то, что он выполняется за время, определяемое линейной зависимостью.

Лучший способ понять этот алгоритм — рассмотреть пример и иллюстрацию. На рис. 7.16, *а* показана простая база знаний из хорновских выражений, содержащая выражения A и B как известные факты. На рис. 7.16, *б* приведена та же база знаний, изображенная в виде **графа И–ИЛИ** (см. раздел 4.3.2). В графах И–ИЛИ несколько ребер, соединенных отрезком дуги, обозначают конъюнкцию (в них необходимо доказать истинность каждого ребра), а несколько ребер, не соединенных друг с другом, обозначают дизъюнкцию (достаточно доказать истинность любого из этих ребер). На этом графе очень просто проанализировать, как действует алгоритм прямого логического вывода. Сначала устанавливаются значения известных листовых узлов (в данном случае это A и B), а затем процесс логического вывода распространяется вверх по графу до тех пор, пока это возможно. При появлении любой конъюнкции процесс распространения останавливается и ожидает до тех пор, пока все конъюнкты не становятся известными, и только после этого продолжается. Рекомендуем читателю подробно проработать этот пример.

Легко убедиться, что алгоритм прямого логического вывода является **непротиворечивым**: каждый этап логического вывода по сути представляет собой применение правила отделения. Кроме того, алгоритм прямого логического вывода является **полным**: в нем может быть получено каждое атомарное высказывание, которое следует из базы знаний. Проще всего в этом можно убедиться, рассмотрев заключительное состояние таблицы *inferred* (после того, как этот алгоритм достигнет фиксированной точки, в которой становятся невозможными какие-либо новые этапы логического вывода). Эта таблица содержит значение *true* для каждого символа, выведенного логическим путем в течение этого процесса, и *false* — для всех других символов. Эта таблица может рассматриваться как логическая модель; более того, ➡ *каждое определенное выражение в первоначальной базе знаний KB является истинным в данной модели.*

```
function PL-FC-ENTAILS?(KB, q) returns значение true или false
  inputs: KB, база знаний, множество определенных пропозициональных
                выражений
          q, запрос, пропозициональный символ
  count ← таблица, где count[c] исходно является количеством символов
            в предпосылке выражения c
  inferred ← таблица, где inferred[s] исходно имеет значение false для
               всех символов
  queue ← очередь из символов, исходно содержит все символы в KB,
            о которых известно, что они истинны

  while queue не пуста do
    p ← POP(queue)
    if p = q then return true
    if inferred[p] = false then
      inferred[p] ← true
      for each выражения c в KB, где предпосылка p присутствует
                в c.PREMISE do
        уменьшить на единицу count[c]
        if count[c] = 0 then добавить c.CONCLUSION to queue
  return false
```

Рис. 7.15. Алгоритм прямого логического вывода для логики высказываний. В очереди *queue* хранятся сведения о символах, в отношении которых известно, что они истинны, но которые еще не "обработаны". В таблице *count* отслеживается информация о том, какое количество предпосылок каждой импликации еще не проверено. Каждый раз, когда обрабатывается новый символ p из очереди символов *queue*, количество предпосылок в таблице *count* сокращается на единицу для каждой импликации, в которой появляется предпосылка p. (Такие импликации могут обнаруживаться за постоянное время, если индексация базы знаний KB выполнена должным образом.) Если счетчик обнуляется, все предпосылки для некоторой импликации оказываются известными, поэтому заключение этой импликации может быть добавлено в очередь *queue*. И наконец, необходимо следить за тем, какие символы уже были обработаны: символ, выведенный логическим путем, не следует еще раз добавлять в очередь *queue*, если он уже был обработан. Это позволяет избежать излишней работы, а также предотвращает возникновение бесконечных циклов, которые могут быть вызваны наличием таких импликаций, как $P \Rightarrow Q$ и $Q \Rightarrow P$

Чтобы убедиться в этом, предположим обратное, а именно — что некоторое выражение $a_1 \wedge \cdots \wedge a_k \Rightarrow b$ из базы знаний является ложным в этой модели. Тогда в этой модели выражение $a_1 \vee \cdots \vee a_k$ должно быть истинным и в ней же выражение b должно быть ложным. Но это противоречит нашему предположению о том, что алгоритм достиг фиксированной точки! Поэтому можно сделать вывод, что множество атомарных высказываний, полученное логическим путем к моменту

достижения фиксированной точки, определяет модель первоначальной базы знаний *KB*. Более того, любое атомарное высказывание q, которое следует из базы знаний *KB*, должно быть истинным во всех ее моделях, а также, в частности, в данной модели. Итак, любое высказывание q, которое следует из базы знаний, должно быть выведено логически данным алгоритмом.

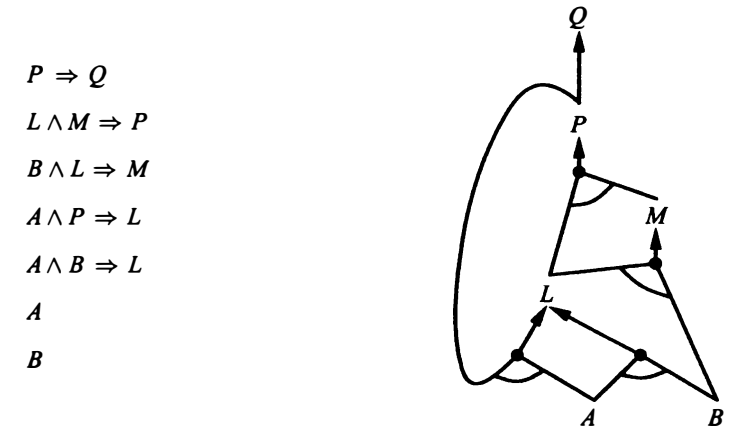


Рис. 7.16. а) Множество хорновских выражений. **б)** Соответствующий И–ИЛИ-граф

Прямой логический вывод представляет собой пример применения общего понятия логических ▶**рассуждений, управляемых данными**, т.е. рассуждений, в которых внимание вначале сосредоточено на известных данных. Прямой логический вывод может использоваться в любом агенте для получения заключений на основе поступающих результатов восприятия, часто даже без учета какого-либо конкретного запроса. Например, агент для мира вампуса может сообщать с помощью операции TELL результаты своих восприятий базе знаний с использованием инкрементного алгоритма прямого логического вывода, в котором новые факты могут добавляться в очередь *queue* для инициализации новых процессов логического вывода. У людей формирование рассуждений, управляемых данными, в определенной степени происходит по мере поступления новой информации. Например, находясь дома и услышав, что пошел дождь, человек, который собирался на пикник, может прийти к заключению, что его придется отменить. Но такое решение вряд ли будет принято, если он узнает лишь то, что семнадцатый лепесток самой крупной розы в саду его соседа оказался мокрым; люди держат процессы прямого логического вывода под тщательным контролем, поскольку в противном случае их просто затопил бы поток несущественных, не относящихся к делу логических заключений.

Алгоритм обратного логического вывода, как указывает само его название, действует в обратном направлении от запроса. Если окажется возможным сразу же узнать, что высказывание в запросе q истинно, то никакой работы выполнять не потребуется. В противном случае алгоритм находит те импликации в базе знаний, из которых следует q. Если можно доказать, что все предпосылки одной из этих импликаций являются истинными (с помощью обратного логического вывода), то высказывание q также будет истинным. Будучи примененным к запросу Q, показанному на рис. 7.16, этот алгоритм будет проходить вниз по графу до тех пор, пока не достигнет множества известных фактов, A и B, образующих основу для доказательства. Этот алгоритм по существу идентичен алгоритму AND-OR-GRAPH-SEARCH (см. рис. 4.11). Как и в случае алгоритма прямого логического вывода, эффективная реализация этого алгоритма выполняет свою работу за линейное время.

Обратный логический вывод представляет собой одну из форм ▶**рассуждений, направляемых целями**. Такая форма будет полезна при получении ответов на конкретные вопросы, подобные следующим: "Что я сейчас должен сделать?" и "Где находятся мои ключи?" Зачастую стоимость обратного логического вывода *намного меньше* по сравнению со стоимостью, линейно зависящей от размера базы знаний, поскольку в этом процессе затрагиваются только факты, непосредственно относящиеся к делу.

7.6. Эффективный пропозициональный логический вывод

В этом разделе рассматриваются два семейства эффективных алгоритмов логического вывода, построенных на проверке пропозициональной модели: один подход основан на поиске с возвратами, а другой — на поиске восхождением к вершине. Эти алгоритмы входят в состав основного "инструментария" логики высказываний. Данный раздел может быть пропущен при первом чтении этой главы.

Рассматриваемые здесь алгоритмы предназначены для проверки выполнимости — это так называемые задачи SAT. (Как отмечалось в разделе 7.5, проверка логического следствия $\alpha \models \beta$ может быть проведена посредством проверки невыполнимости высказывания $\alpha \wedge \neg\beta$.) Выше уже отмечалась связь между поиском модели, обеспечивающей выполнимость логического высказывания, и поиском решения задачи удовлетворения ограничения, поэтому, вероятно, нет ничего удивительного в том, что эти два семейства алгоритмов весьма напоминают алгоритмы поиска с возвратами, описанные в разделе 6.3, и алгоритмы локального поиска, которые представлены в разделе 6.4. Тем не менее приведенные здесь алгоритмы являются чрезвычайно важными сами по себе, поскольку в компьютерных науках существует очень много комбинаторных задач, которые можно свести к проверке выполнимости пропозиционального высказывания. Любое

усовершенствование алгоритмов проверки выполнимости будет иметь очень серьезные последствия в отношении повышения нашей способности справляться со сложностью в целом.

7.6.1. Полный алгоритм поиска с возвратами

Первый рассматриваемый здесь алгоритм часто называют ►**алгоритмом Дэвиса–Патнема** в честь авторов известной статьи, в которой он был опубликован, Мартина Дэвиса и Хилари Патнема ([544], 1960). Затем этот алгоритм фактически стал одной из версий, описанных Дэвисом, Логеманом и Лавлендом ([543], 1962), поэтому мы будем называть его DPLL по первым буквам фамилий всех четырех авторов. Алгоритм DPLL принимает на входе некоторое высказывание в конъюнктивной нормальной форме — высказывание, представленное как множество выражений. Как и алгоритмы BACKTRACKING-SEARCH и TT-ENTAILS?, он фактически выполняет рекурсивный перебор в глубину всех возможных моделей. В этом алгоритме реализованы три описанных ниже усовершенствования, и этим он отличается от простой схемы, применяемой в алгоритме TT-ENTAILS?.

- *Раннее завершение.* Алгоритм обнаруживает, должно ли данное высказывание быть истинным или ложным, даже с помощью частично завершенной модели. Выражение является истинным, если истинен любой его литерал, — даже в том случае, когда для других литералов еще не определены истинностные значения. Поэтому об истинности всего высказывания в целом можно судить еще до того, как модель будет составлена полностью. Например, высказывание $(A \vee B) \wedge (A \vee C)$ является истинным, если истинен литерал A, независимо от значений литералов B и C. Аналогичным образом высказывание является ложным, если ложно любое его выражение, а это происходит, если каждый литерал этого выражения является ложным. Опять-таки, такая ситуация может возникнуть задолго до того, как модель будет полностью составлена. Раннее завершение позволяет обойтись без исследования целых поддеревьев в пространстве поиска.
- *Эвристика чистого символа.* ►**Чистым символом** называется символ, который всегда появляется с одним и тем же "знаком" во всех выражениях. Например, в трех выражениях, $(A \vee \neg B)$, $(\neg B \vee \neg C)$ и $(C \vee A)$, символ A является чистым, поскольку он появляется только в виде положительных литералов. Чистым также можно считать символ B, который появляется только в виде отрицательных литералов, а символ C считается нечистым. Можно легко показать, что если некоторое высказывание имеет модель, то это модель с чистыми символами, значения которым присвоены так, чтобы их литералы приняли значение *true*, поскольку при таком условии ни одно выражение не может стать ложным. Следует отметить, что при определении чистоты символа алгоритм может игнорировать выражения, в отношении которых уже

известно, что они истинны в модели, составленной до сих пор. Например, если модель содержит присваивание $B = \mathit{false}$, то выражение $(\neg B \vee \neg C)$ уже является истинным, поэтому в оставшемся выражении $(C \vee A)$ символ C, присутствующий в нем как положительный, воспринимается как чистый.

- *Эвристика единичного выражения*. **Единичное выражение** было определено ранее как выражение только с одним литералом. В контексте алгоритма DPLL это определение также относится к выражениям, для которых в данной модели всем их литералам, кроме одного, уже было присвоено значение *false*. Например, если модель содержит присваивание $B = \mathit{true}$, то выражение $(\neg B \vee \neg C)$ воспринимается как единичное выражение, поскольку оно эквивалентно выражению $\neg C$. Очевидно, для того, чтобы это выражение приняло истинное значение, литералу C должно быть присвоено значение *false*. Эвристика единичного выражения предусматривает присваивание значений всем таким символам до того, как происходит переход к обработке оставшейся части высказывания. Одним из важных следствий из этой эвристики является то, что любая попытка доказать (путем опровержения) истинность литерала, который уже находится в базе знаний, немедленно приводит к успеху (см. упр. 7.29). Следует также отметить, что присваивание значения одному единичному выражению может привести к созданию еще одного единичного выражения; например, после присваивания символу C значения *false* единичным становится выражение $(C \vee A)$, что влечет за собой присваивание истинного значения символу A. Такое "каскадное" распространение форсированных присваиваний называется ▶ **распространением единичных выражений**. Оно напоминает процесс прямого логического вывода с применением определенных выражений, и в действительности, если рассматриваемое высказывание в конъюнктивной нормальной форме содержит только определенные выражения, то алгоритм DPLL, по сути, сводится к алгоритму прямого логического вывода (см. упражнение 7.30).

Алгоритм DPLL, приведенный на рис. 7.17, представляет лишь основной механизм процесса поиска, без уточнения деталей реализации.

На рис. 7.17 не представлены все хитрости, которые позволяют решателям SAT успешно справляться с большими задачами. Интересно, что большинство из этих хитростей на самом деле носят довольно общий характер и мы уже встречались с ними раньше в других ипостасях.

1. **Выделение компонентов** (как в случае с Тасманией в задаче УО). Когда алгоритм DPLL присваивает истинностные значения переменным, множество выражений может оказаться разделенным на непересекающиеся подмножества, называемые **компонентами**, не имеющими общих неназначенных переменных. При наличии эффективного способа выявления подобных ситуаций решатель может получить значительный выигрыш в скорости, обрабатывая каждый компонент независимо от других.

```
function DPLL-SATISFIABLE?(s) returns значение true или false
  inputs: s, высказывание пропозициональной логики

  clauses ← множество выражений высказывания s, преобразованного
            в форму представления CNF
  symbols ← список пропозициональных символов в высказывании s
  return DPLL(clauses, symbols, {})

function DPLL(clauses, symbols, model) returns значение true или false
  if каждое выражение в множестве clauses имеет значение true в модели model
    then return true
  if какое-то выражение в множестве clauses имеет значение false в модели model
    then return false
  P, value ← FIND-PURE-SYMBOL(symbols, clauses, model)
  if значение P не является пустым
    then return DPLL(clauses, symbols – P, model ∪ {P = value})
  P, value ← FIND-UNIT-CLAUSE(clauses, model)
  if значение P не является пустым
    then return DPLL(clauses, symbols – P, model ∪ {P = value})
  P ← FIRST(symbols); rest ← REST(symbols)
  return DPLL(clauses, rest, model ∪ {P = true}) or
         DPLL(clauses, rest, model ∪ {P = false}))
```

Рис. 7.17. Алгоритм DPLL для проверки выполнимости высказывания в логике высказываний. Принципы работы функций FIND-PURE-SYMBOL и FIND-UNIT-CLAUSE описаны в тексте; каждая из них возвращает символ (или пустое значение), а также истинностное значение, которое должно быть присвоено этому символу. Как и алгоритм TT-ENTAILS?, этот алгоритм работает с частично составленными моделями

2. **Упорядочение переменных и значений** (как это делалось в отношении задач УО). В нашей простой реализации алгоритма DPLL переменные выбираются в произвольном порядке и значение *true* всегда проверяется перед значением *false*. **Степенная эвристика** (см. раздел 6.3.1) предлагает выбирать переменную, которая чаще других появляется во всех оставшихся выражениях.
3. **Интеллектуальный поиск с возвратами** (см. раздел 6.3.3 применительно к задачам УО). Многие задачи, которые часами не удается решить с использованием алгоритмов с хронологическим возвратом, могут быть решены за считанные секунды методом интеллектуального поиска с возвратами, выполняющего откат вплоть до релевантной точки конфликта. Все решатели SAT, применяющие интеллектуальный поиск с возвратами, используют ту или иную форму **определения конфликтных выражений с помощью обучения** для запоминания сведений о конфликтах, что позволяет избежать их

повторения и необходимости отката в последующем поиске. Обычно сохраняется ограниченное количество записей о конфликтах, и те из них, которые используются редко, со временем отбрасываются.

4. **Случайный перезапуск** (см. раздел 4.1.1 применительно к поиску восхождением к вершине). Иногда ход расчетов не дает необходимого прогресса и в этом случае может оказаться полезнее начать поиск заново, вернувшись к вершине поискового дерева, а не продолжать попытки достичь результатов из текущей позиции. После перезапуска выполняются другие случайные выборы (в отношении переменной и значения). Выражения, которые уже были изучены при первом прогоне, сохраняются и после перезапуска, поскольку могут оказаться полезными для отсечения ветвей в пространстве поиска. Перезапуск не гарантирует, что решение будет найдено быстрее, но позволяет уменьшить разброс времени выполнения поиска решения.
5. **Разумное индексирование** (применяется во многих алгоритмах). Методы ускорения, используемые в самом алгоритме DPLL, как и упомянутые выше приемы, используемые в современных решателях, требуют быстрого поиска, а значит, эффективной индексации таких вещей, как "множество выражений, в которых переменная X_i выглядит как положительный литерал". Эта задача усложняется тем фактом, что алгоритмам нужны лишь те выражения, которые еще не были удовлетворены предыдущими присваиваниями переменных, а это значит, что индексируемые структуры должны позволять динамическое обновление по ходу выполнения расчетов.

С помощью этих улучшений современные решатели могут успешно работать с задачами, включающими десятки миллионов переменных. Их появление имело следствием настоящую революцию в таких областях, как верификация аппаратных средств и проверка протоколов безопасности, которые ранее требовали трудоемких ручных доказательств.

7.6.2. Алгоритмы локального поиска

В данной книге уже было представлено несколько алгоритмов локального поиска, включая алгоритмы HILL-CLIMBING (раздел 4.1.1) и SIMULATED-ANNEALING (раздел 4.1.2). Эти алгоритмы могут непосредственно применяться для решения задач проверки выполнимости, при условии, что будет выбрана правильная функция оценки. Поскольку цель состоит в том, чтобы найти присваивание, обеспечивающее выполнение каждого выражения, для этой цели может использоваться любая функция оценки, которая подсчитывает количество невыполненных выражений. Фактически именно этот критерий и применяется в алгоритме MIN-CONFLICTS для задач УО (раздел 6.4). Все эти алгоритмы выполняются в пространстве полных присваиваний, на каждом этапе меняя на противоположное (инвертируя) истинностное значение одного символа. Это пространство обычно содержит

много локальных минимумов, для выхода из которых требуются различные формы рандомизации. В последние годы было проведено множество экспериментов с тем, чтобы можно было найти приемлемый компромисс между стремлением к "жадному" выбору наилучшего преемника и необходимостью случайного выбора очередного преемника.

Одним из простейших и наиболее эффективных алгоритмов, которые были созданы в результате всей этой работы, является алгоритм, получивший название WALKSAT (рис. 7.18). При каждой итерации этот алгоритм выбирает одно невыполненное выражение, а затем в этом выражении выбирается один символ для инвертирования. При этом алгоритм случайным образом переключается между двумя способами выбора символа для инвертирования: первый — это вариант с "минимизацией конфликтов", в котором минимизируется количество невыполненных выражений в новом состоянии, а второй — вариант "случайного перехода", в котором один из символов выбирается случайным образом.

---

function WALKSAT(*clauses*, *p*, *max_flips*) **returns** модель *model*, выполняющую высказывание, или значение *failure*
  inputs: *clauses*, множество выражений пропозициональной логики
    p, вероятность выбора для этапа варианта "случайного перехода", обычно около 0,5
    max_flips, количество инверсий, которые допускается выполнить, прежде чем отказаться от дальнейших попыток

  model ← случайно выбранное присваивание значений *true*/*false* символам в множестве *clauses*
  for each *i* = 1 **to** *max_flips* **do**
    if при модели *model* множество *clauses* выполняется **then return** *model*
    clause ← выбранное случайным образом из множества *clauses* выражение, имеющее значение *false* в модели *model*
    if RANDOM(0, 1) ≤ *p* **then**
      в модели *model* инвертировать значение символа, случайным образом выбранного в множестве *clause*
    else в множестве *clause* инвертировать значение того символа, при выборе которого максимизируется количество выполненных выражений
  return *failure*

Рис. 7.18. Алгоритм WALKSAT для проверки выполнимости путем инвертирования случайным образом значений переменных. Существует много версий этого алгоритма

Когда алгоритм WalkSAT, возвращает некоторую модель, то входное высказывание действительно является выполнимым. Однако, когда он возвращает

индикатор неудачи *failure*, у этого могут быть две причины: либо высказывание невыполнимо, либо алгоритму требуется больше времени, чтобы добиться успеха. Если задать $max_flips = \infty$ и $p > 0$, алгоритм WALKSAT в конечном итоге вернет какую-то модель (если она существует), поскольку на этапах случайного перемещения рано или поздно произойдет попадание в решение. К сожалению, если параметр *max_flips* имеет бесконечно большое значение, а высказывание является невыполнимым, этот алгоритм никогда не закончит своей работы!

По этой причине алгоритм WALKSAT является наиболее полезным, когда можно с уверенностью полагать, что решение существует, например задачи, рассматривавшиеся в главах 3 и 6, как правило, имеют решения. С другой стороны, алгоритм WALKSAT не всегда может обнаружить *невыполнимость*, а именно это и требуется, когда задача заключается в определении, следует ли какое-то высказывание из базы знаний. Например, агент не может *надежно* использовать алгоритм WALKSAT для доказательства, что некоторый квадрат в мире вампуса является безопасным. Вместо этого он может лишь утверждать: "Я размышлял об этом в течение часа, но так и не смог найти хотя бы один из возможных миров, в котором данный квадрат нельзя считать безопасным". Это можно считать хорошим эмпирическим показателем безопасности данного квадрата, но все же это не является доказательством.

7.6.3. Трудные задачи определения выполнимости

Одни задачи SAT сложнее, чем другие, и именно они представляют наибольший интерес, поскольку *легкие* могут быть решены с помощью любого старого алгоритма. Однако нам уже известно, что задачи SAT относятся к классу NP-полных, и по крайней мере некоторые их экземпляры должны требовать экспоненциального времени выполнения. В главе 6 говорилось о некоторых удивительных открытиях в области решения задач определенного типа. Так, задача с n ферзями, которая воспринималась как весьма сложная при ее решении с помощью алгоритмов поиска с возвратами, оказалась тривиально простой для методов локального поиска, таких как метод минимальных конфликтов. Это связано с тем, что решения этой задачи распределены в пространстве присваиваний очень плотно и для любого начального присваивания гарантируется, что решение находится где-то рядом. Таким образом, задача с n ферзями является простой потому, что она ▶**недостаточно ограниченна**.

Если говорить о решении задач проверки выполнимости, представленных в конъюнктивной нормальной форме, то среди них недостаточно ограниченными можно считать такие задачи, которые содержат относительно *немного* выражений, ограничивающих значения переменных. Например, ниже представлено сформированное случайным образом высказывание в форме 3-CNF с пятью символами и пятью выражениями:

$$(\neg D \vee \neg B \vee C) \wedge (B \vee \neg A \vee \neg C) \wedge (\neg C \vee \neg B \vee E) \\ \wedge (E \vee \neg D \vee B) \wedge (B \vee E \vee \neg C).$$

Моделями этого высказывания являются 16 из 32 возможных вариантов присваивания, поэтому в среднем для поиска модели потребуются только две случайно выбранные гипотезы. Это пример легко решаемой задачи проверки выполнимости, какими является большинство подобных недостаточно ограниченных задач. С другой стороны, *чрезмерно ограниченные* задачи содержат много выражений по отношению к количеству переменных и с большой вероятностью не имеют решений. Чрезмерно ограниченные задачи также часто решаются очень легко, поскольку избыток ограничений либо быстро приводит алгоритм к решению, либо заводит его в тупик, из которого нет выхода.

Чтобы выйти за рамки этих основных интуитивных представлений, необходимо точно определить, как генерируются случайные высказывания. Введем обозначение $CNF_k(m, n)$, определяющее высказывание в форме *k-CNF* с m выражениями и n символами, где выражения выбираются единообразно, независимо друг от друга и без замены из числа всех выражений с k различными литералами, которые случайным образом будут определены как положительные или отрицательные. (Любой символ не может дважды появляться в выражении, также как и любое выражение не может дважды появляться в высказывании.)

После определения подобным образом источника случайных высказываний появляется возможность измерить вероятность их выполнимости. На рис. 7.19, *а* представлена вероятность того, что случайно сформированное высказывание в форме $CNF_3(m, 50)$ является выполнимым как функция от отношения "выражение/символ", m/n, при постоянном значении n, равном 50, и выражениях из трех литералов. Как и следовало ожидать, при малых значениях m/n эта вероятность близка к 1, а при больших значениях m/n вероятность близка к 0. На кривой вероятности довольно резкий спад начинается приблизительно после достижения значения $m/n = 4{,}3$. Эмпирически было обнаружено, что по мере увеличения значения n этот "крутой обрыв" остается примерно на том же месте (для $k = 3$) и становится все круче с ростом n.

Теоретически ▶ **теорема критической точки выполнимости** утверждает, что для любых $k \geq 3$ существует пороговое значение отношения r_k, такое, что по мере увеличения значения n в пределе до бесконечности вероятность того, что высказывание $CNF_k(m, n)$ будет выполнимым, стремится к 1 для всех значений r ниже этого порогового значения и стремится к 0 для всех значений r выше его. Эта гипотеза остается недоказанной даже для особых случаев, таких как $k = 3$. Но независимо от того, можно ли считать это утверждение теоремой, данный вид порогового эффекта, безусловно, является распространенным явлением как для задач выполнимости, так и для других типов NP-сложных задач.

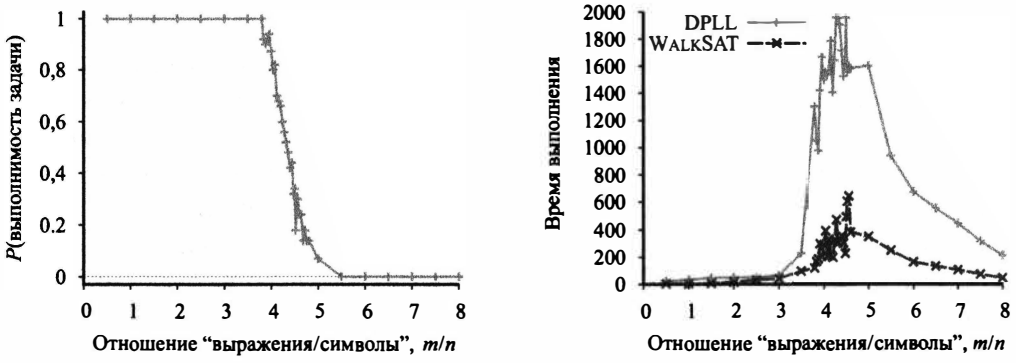


Рис. 7.19. а) График вероятности того, что случайно сформированное высказывание в форме 3-CNF с количеством символов $n = 50$ окажется выполнимым как функция от отношения "выражения/символы", m/n. **б)** График усредненного времени выполнения (измеряемого в количестве итераций) алгоритмов DPLL и WalkSAT для сформированных случайным образом высказываний в форме 3-CNF. Наиболее сложные задачи имеют отношение "выражения/символы" в районе значения 4,3

Теперь, когда у нас есть хорошая идея, где находятся выполнимые и невыполнимые задачи, возникает следующий вопрос: а где искать трудные проблемы? Оказывается, что они также часто находятся вблизи критической точки выполнимости. На рис. 7.19, *б* показано, что задачи с 50 символами и значением r в районе порогового значения 4,3 решить примерно в 20 раз труднее, чем те, у которых соотношение r равно 3,3. Недостаточно ограниченные задачи решить проще всего (потому что очень легко просто угадать решение); чрезмерно ограниченные задачи решаются не так легко, как недостаточно ограниченные, но все же намного легче тех, которые имеют показатель r в непосредственной близости к критической точке.

7.7. Агенты, основанные на логике высказываний

В данном разделе собирается воедино все, что было описано до сих пор с целью разработки агентов для мира вампуса, действующих с использованием логики высказываний. На первом этапе агент получит способность делать выводы о состоянии мира, — насколько это возможно, исходя из истории его восприятий. Это потребует записи полной логической модели результатов действий агента. Далее будет показано, как агент может использовать логический вывод в мире вампуса, а также как он может эффективно хранить информацию о своих действиях в этом мире, не возвращаясь к полной истории восприятий при каждом выводе.

В завершение будет показано, как агент может использовать логический вывод для разработки планов, гарантирующих достижение его целей при условии, что его база знаний верна по отношению к реальному миру.

7.7.1. Определение текущего состояния мира

Как указывалось в начале главы, логический агент действует, принимая решение о том, как ему поступить, посредством логического вывода из базы знаний, состоящей из высказываний о состоянии мира. База знаний включает аксиомы — общие знания о том, как устроен мир — и высказывания о восприятии, сформированные на основании опыта агента в данном конкретном мире. В этом разделе мы сосредоточимся на задаче определения текущего состояния мира вампуса — где я нахожусь, безопасен ли данный квадрат и т.д.

Начнем с набора аксиом, определенных в разделе 7.4.3. Агент знает, что в квадрате [1,1] нет ямы ($\neg P_{1,1}$) или вампуса ($\neg W_{1,1}$). Также он знает, что ветерок в квадрате может ощущаться тогда и только тогда, когда в соседнем квадрате есть яма, а неприятный запах в квадрате может быть тогда и только тогда, когда в соседнем квадрате находится вампус. Следовательно, в базу также нужно включить большое множество высказываний следующего вида:

$$B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1});$$
$$S_{1,1} \Leftrightarrow (W_{1,2} \vee W_{2,1})$$
$$\ldots$$

Агент также знает, что в мире вампуса существует точно один вампус. Соответствующее высказывание состоит из двух частей. Прежде всего необходимо указать, что имеется *самое меньшее* один вампус:

$$W_{1,1} \vee W_{1,2} \vee \cdots \cdot \vee W_{4,3} \vee W_{4,4}.$$

Далее необходимо указать, что существует *самое большее* один вампус. Для этого для каждой пары квадратов в базу добавляется высказывание о том, что хотя бы один из них обязательно должен быть без вампуса:

$$\neg W_{1,1} \vee \neg W_{1,2};$$
$$\neg W_{1,1} \vee \neg W_{1,3}$$
$$\ldots$$
$$\neg W_{4,3} \vee \neg W_{4,4}.$$

Пока все хорошо. Теперь давайте рассмотрим восприятие агента. Для указания на наличие неприятного запаха в квадрате [1,1] выше использовалось высказывание $S_{1,1}$; но можно ли для этой цели воспользоваться единственным высказыванием о чистом восприятии *Stench*, означающем, что агент воспринимает неприятный запах? К сожалению, это невозможно: если на предыдущем по времени этапе восприятие неприятного запаха отсутствовало, то это уже было бы отмечено

утверждением $\neg Stench$ и новое утверждение $Stench$ просто привело бы к противоречию. Эту проблему можно устранить, если прийти к заключению, что восприятие утверждает что-то *только о текущем моменте времени*. Следовательно, если данный момент является четвертым интервалом времени, то в базу данных можно *добавить* утверждение $Stench^4$, а не просто $Stench$ (как и реализовано в функции MAKE-PERCEPT-SENTENCE на рис. 7.1), что позволит ловко избежать какого-либо противоречия с утверждением $\neg Stench^3$. Тот же самый подход можно использовать в отношении восприятия ветерка, удара, блеска и крика.

Идею связать высказывания с моментами времени можно распространить на любой аспект мира, который изменяется во времени. Например, база знаний в исходном состоянии должна включать высказывания $L^0_{1,1}$ — агент находится в квадрате [1,1] в момент времени 0, — а также $FacingEast^0$, $HaveArrow^0$ и $WumpusAlive^0$. Ниже для ссылок на аспекты мира, которые изменяются во времени, мы будем использовать слово ▶**флюента** (от лат. *fluens — текущий, изменчивый*). Здесь "флюента" — это синоним понятия "переменная состояния" в том смысле, как оно было определено при обсуждении особенностей развернутого представления состояний в разделе 2.4.7. Символы, связанные с постоянными аспектами мира, не нуждаются в дополнительном верхнем индексе, определяющем конкретный момент времени, поэтому иногда их называют ▶**вневременными переменными**.

Теперь прямые восприятия *breeze* и *stench* можно связать непосредственно со свойствами тех квадратов, где они будут иметь место.[11] Для любого момента времени t и любого квадрата $[x, y]$ можно утверждать следующее:

$$L^t_{x,y} \Rightarrow (Breeze^t \Leftrightarrow B_{x,y});$$
$$L^t_{x,y} \Rightarrow (Stench^t \Leftrightarrow S_{x,y}).$$

Теперь, конечно же, нам нужны аксиомы, которые позволят агенту отслеживать значения флюент, таких как $L^t_{x,y}$. Все эти флюенты изменяют свои значения в результате действий, предпринятых агентом, поэтому, в терминологии главы 3, нам необходимо записать **функцию определения преемника** в мире вампуса, представленную в виде множества логических высказываний.

Прежде всего нам потребуются пропозиционные символы для представления выполняемых действий. Как и в случае с восприятием, эти символы должны индексироваться по времени, поэтому утверждение $Forward^0$ будет означать, что агент выполнил действие *Forward* в момент времени 0. По соглашению на определенном временном этапе сначала происходит восприятие, затем выполняется действие этого этапа, после чего осуществляется переход к следующему временному этапу.

[11] В разделе 7.4.3 нам было удобнее умолчать об этих требованиях.

Чтобы описать, как меняется мир, можно попробовать записать ▶**аксиомы эффектов**, которые будут определять результат некоторого действия для следующего момента времени. Например, если в момент времени 0 агент находится в квадрате [1,1], смотрит в восточном направлении и выполняет действие *Forward* (идти вперед), то в результате его выполнения агент будет находиться в квадрате [2,1] и больше не будет находиться в квадрате [1,1]. Это можно выразить следующим образом.

$$L^0_{1,1} \wedge FacingEast^0 \wedge Forward^0 \Rightarrow (L^1_{2,1} \wedge \neg L^1_{1,1}) \qquad (7.1)$$

Нам потребуется по одному подобному высказыванию для каждого возможного момента времени, для каждого из 16 квадратов и для каждой из четырех возможных ориентаций агента в квадрате. Аналогичные высказывания нужно будет предоставить и для других действий: *Grab*, *Shoot*, *Climb*, *Turnleft* и *TurnRigh*.

Предположим, что в момент времени 0 агент принял решение выполнить действие *Forward* и должен отметить этот факт в своей базе знаний. Воспользовавшись аксиомой эффекта из уравнения (7.1) в сочетании с исходными утверждениями о своем состоянии в момент времени 0, агент может сделать вывод, что теперь он находится в квадрате [2, 1]. Это значит, что на соответствующий запрос к базе данных будет получен такой ответ: ASK(KB, $L^1_{2,1}$) = *true*. Пока все хорошо. Но, к сожалению, на запрос ASK(KB, $HaveArrow^1$) база данных даст ответ *false*, т.е. агент не сможет подтвердить ни то, что у него все еще есть стрела, ни то, что у него уже *нет* стрелы! Информация о стреле была утрачена потому, что аксиома эффекта не может указать, что ее флюента остается *неизменной* в результате выполнения определенного действия. Необходимость сделать это приводит нас к ▶**проблеме фреймов** (*frame problem*).[12] Одно возможное решение проблемы фреймов заключается в добавлении в базу данных ▶**аксиом фреймов**, явно определяющих все высказывания, которые остаются неизменными при переходе к следующему этапу. Например, для каждого момента времени t можно записать следующее:

$$Forward^t \Rightarrow (HaveArrow^t \Leftrightarrow HaveArrow^{t+1});$$
$$Forward^t \Rightarrow (WumpusAlive^t \Leftrightarrow WumpusAlive^{t+1})$$
$$\ldots$$

Здесь явно упоминается каждое высказывание, которое остается неизменным при переходе от момента времени t к моменту $t+1$ после выполнения действия *Forward*. Хотя агент теперь знает, что после перемещения вперед у него по-прежнему есть стрела и что вампус все еще жив или вновь ожил, стремительное увеличение количества аксиом фреймов кажется крайне неэффективным. В мире с m различными действиями и n флюентами множество аксиом фреймов будет

[12] Название "проблема фреймов" выбрано по аналогии с английским физическим термином *frame of reference*, обозначающим стационарный фон, относительно которого измеряется движение. Можно также провести аналогию с кадрами кинофильма, в которых обычно большая часть фона остается неизменной, а все изменения происходят лишь на переднем плане.

иметь размер $O(mn)$. Эту специфическую особенность проблемы фреймов иногда называют ►**проблемой представления фреймов**. Данная проблема играет значительную роль в истории ИИ; подробнее о ней речь пойдет в разделе "Библиографические и исторические заметки" в конце главы.

Проблема представления фреймов является существенной, потому что в реальном мире имеется, мягко говоря, очень много флюент. К счастью для нас, людей, каждое действие обычно изменяет значение не более какого-то небольшого количества k этих флюент (так мир проявляет свою локальность). Значит, решение проблемы представления фреймов требует определения **функции определения преемника** с набором аксиом размера $O(mk)$, а не размера $O(mn)$. Существует также ►**проблема логического фрейма**: проблема прогнозирования результатов t-этапа плана действий во времени $O(kt)$, а не $O(nt)$.

Решение проблемы предусматривает смещение фокуса с написания аксиом о *действиях* на написание аксиом о *флюентах*. А именно, для каждой флюенты F необходимо предоставить аксиому, определяющую истинностное значение F^{t+1} в терминах флюент (включая саму флюенту F) в момент времени t и действий, которые могут произойти в момент времени t. Теперь истинностное значение флюенты F^{t+1} может быть установлено одним из двух способов: либо действие в момент времени t приводит к тому, что флюента F в момент времени $t+1$ получает значение *true*, либо флюента F в момент времени t уже имела значение *true* и действие в момент времени t не изменило ее значения на *false*. Аксиома такой формы называется ►**аксиомой определения преемника** и имеет следующий вид:

$$F^{t+1} \Leftrightarrow ActionCausesF^t \vee (F^t \wedge \neg ActionCausesNotF^t).$$

Одной из простейших аксиом определения преемника является *HaveArrow*. Поскольку нет никаких действий, позволяющих вернуть ей значение *true*, часть аксиомы $ActionCausesF^t$ просто отсутствует и у нас остается лишь следующее.

$$HaveArrow^{t+1} \Leftrightarrow (HaveArrow^t \wedge \neg Shoot^t) \qquad (7.2)$$

Что касается расположения агента, то аксиомы определения преемника в этом случае будут более сложными. Например, высказывание $L_{1,1}^{t+1}$ будет истинным в двух случаях: во-первых, когда агент выполнил действие *Forward*, находясь в квадрате [1,2] и глядя на юг либо находясь в квадрате [2,1] и глядя на запад, и во-вторых, когда высказывание $L_{1,1}^t$ уже является истинным и выполненное действие не вызывает перемещения агента (либо потому, что выполненное действие отлично от *Forward*, либо потому, что при выполнении действия *Forward* агент стукнулся о стену). Сформулировав все это в логике высказываний, мы получим следующее.

$$\begin{aligned} L_{1,1}^{t+1} \Leftrightarrow\ & (L_{1,1}^t \wedge (\neg Forward^t \vee Bump^{t+1})) \\ & \vee (L_{1,2}^t \wedge (FacingSouth^t \wedge Forward^t)) \\ & \vee (L_{2,1}^t \wedge (FacingWest^t \wedge Forward^t)) \end{aligned} \qquad (7.3)$$

В упражнении 7.34 вам будет предложено записать аксиомы для всех остальных флюент мира вампуса.

Имея полный набор аксиом определения преемника и других аксиом, перечисленных в начале этого раздела, агент получит возможность направлять запросы базе знаний и получать ответы на любой важный для него вопрос о текущем состоянии мира. Например, в разделе 7.2 исходная последовательность восприятий и действий была следующей.

$$\begin{aligned}
&\neg Stench^0 \wedge \neg Breeze^0 \wedge \neg Glitter^0 \wedge \neg Bump^0 \wedge \neg Scream^0 \ ; \ Forward^0 \\
&\neg Stench^1 \wedge Breeze^1 \wedge \neg Glitter^1 \wedge \neg Bump^1 \wedge \neg Scream^1 \ ; \ TurnRight^1 \\
&\neg Stench^2 \wedge Breeze^2 \wedge \neg Glitter^2 \wedge \neg Bump^2 \wedge \neg Scream^2 \ ; \ TurnRight^2 \\
&\neg Stench^3 \wedge Breeze^3 \wedge \neg Glitter^3 \wedge \neg Bump^3 \wedge \neg Scream^3 \ ; \ Forward^3 \\
&\neg Stench^4 \wedge \neg Breeze^4 \wedge \neg Glitter^4 \wedge \neg Bump^4 \wedge \neg Scream^4 \ ; \ TurnRight^4 \\
&\neg Stench^5 \wedge \neg Breeze^5 \wedge \neg Glitter^5 \wedge \neg Bump^5 \wedge \neg Scream^5 \ ; \ Forward^5 \\
&Stench^6 \wedge \neg Breeze^6 \wedge \neg Glitter^6 \wedge \neg Bump^6 \wedge \neg Scream^6
\end{aligned}$$

На этом этапе агент получает ответ на запрос $\text{ASK}(KB, L^6_{1,2}) = true$ и поэтому знает, где находится. Более того, ответы на запросы $\text{ASK}(KB, W_{1,3}) = true$ и $\text{ASK}(KB, P_{3,1}) = true$ говорят ему о том, что он обнаружил местонахождение вампуса и убедился, что в соседнем квадрате находится яма. Но самый важный вопрос для агента иной: можно ли перейти в определенный квадрат, т.е. отсутствует ли в нем яма или живой вампус. Для этого будет удобно добавить в базу знаний еще несколько аксиом, имеющих следующий вид:

$$OK^t_{x,y} \Leftrightarrow \neg P_{x,y} \wedge \neg(W_{x,y} \wedge WumpusAlive^t).$$

Теперь агент сможет получить ответ на запрос $\text{ASK}(KB, OK^6_{2,2}) = true$ и будет уверен, что квадрат [2,2] можно использовать как цель перехода. На самом деле при наличии непротиворечивого и полного алгоритма логического вывода, такого как DPLL, агент сможет получить ответ на любой столь важный для него вопрос "Какие квадраты являются безопасными?" и сделать это за несколько миллисекунд для любых версий мира вампуса от малых до средних размеров.

Решение проблем представления и логического вывода фреймов является большим шагом вперед, но одна пагубная проблема все же остается: необходимость найти подтверждение, что *все* необходимые предпосылки для любого действия учтены, поскольку только так можно достичь ожидаемого эффекта. Мы утверждали, что действие *Forward* вызовет перемещение агента вперед, если только на его пути нет стены, но существует много других необычных, редких исключений, которые могут привести к тому, что это действие не будет выполнено: агент может споткнуться и упасть, у него может случиться сердечный приступ, его может утащить гигантская летучая мышь и т.д. Определение всех подобных исключений называется ►**проблемой квалификации**. В логике для нее нет полного решения, — разработчики систем должны использовать здравый смысл, принимая решения о том, насколько подробно они хотят описать свою

модель и какие детали они намерены опустить. В главе 12 будет показано, что теория вероятностей позволяет обобщить и учесть все возможные исключения без явного их именования.

7.7.2. Гибридный агент

Способность к логическому выводу различных аспектов состояния мира можно довольно просто скомбинировать с правилами "условие–действие" (см. раздел 2.4.2) и алгоритмами решения задач из глав 3 и 4, что позволит создать ▶ **гибридного агента** для мира вампуса. На рис. 7.20 представлен один из возможных способов сделать это. Программа агента поддерживает и обновляет базу знаний, а также текущий план. Исходная база знаний содержит *вневременные* аксиомы — те, которые не зависят от момента времени t, как, например, аксиома, связывающая присутствие в квадрате ветерка с наличием ямы поблизости. На каждом временном этапе в базу знаний добавляется высказывание о текущем восприятии вместе со всеми аксиомами, зависящими от момента времени t, такими как аксиомы определения преемника. (В следующем разделе объясняется, почему агент не нуждается в таких аксиомах для *будущих* моментов времени.) Затем агент использует логический вывод, направляя запросы ASK() в базу знаний, чтобы выяснить, какие квадраты являются безопасными и какие еще требуется посетить.

Основная часть программы агента строит план, исходя из нескольких возможных целей со снижающимся приоритетом. Первая, с наивысшим приоритетом, — если в квадрате замечен блеск, то надо схватить золото, проложить маршрут обратно в исходное положение и вылезть из пещеры. В противном случае, если текущего плана нет, программа создает план перехода в ближайший безопасный квадрат, где агент еще не побывал, прокладывая маршрут только по безопасным квадратам.

Планирование маршрута выполняется с помощью алгоритма поиска A*, а не выдачей запросов ASK в базу данных. Если безопасных квадратов для исследования нет, то следующая цель — если у агента все еще есть стрела — попытаться создать безопасный квадрат, выстрелив в один из тех, в которых может находиться вампус. Последние определяются как те, в которых результатом запроса ASK(KB, $\neg W_{x,y}$) будет *false*, т.е. для которых пока *неизвестно*, что в них *нет* вампуса. Функция PLAN-SHOT (на рисунке не представлена) использует функцию PLAN-ROUTE для планирования последовательности действий, которые приведут к этому выстрелу. Если достичь этой цели не удается, программа переходит к другой цели — ищет квадрат, который еще не исследован и для которого пока не доказано, что он таит опасность, т.е. квадрат, для которого запрос ASK(KB, $\neg OK^t_{x,y}$) = *false*. Если такого квадрата нет, то завершение миссии невозможно и агент отступает к квадрату [1,1] и вылезает из пещеры.

```
function HYBRID-WUMPUS-AGENT(percept) returns действие action
  inputs: percept, список результатов восприятия [stench, breeze, glitter,
                  bump, scream]
  persistent: KB, база знаний, первоначально содержащая лишь определения
                  "законов функционирования" мира вампуса
              t, счетчик, исходно равен 0, определяет время
              plan, намеченная последовательность действий, первоначально
                  пустая

  TELL(KB, MAKE-PERCEPT-SENTENCE(percept, t)) // Запись в базу знаний KB
                          // высказывания о текущей "физике" для времени t
  safe ← {[x, y] : ASK(KB, OK^t_{x,y}) = true}
  if ASK(KB, Glitter^t) = true then
    plan ← [Grab] + PLAN-ROUTE(current, {[1, 1]}, safe) + [Climb]
  if plan пуст then
    unvisited ← {[x, y] : ASK(KB, L^{t'}_{x,y}) = false для всех t' ≤ t}
    plan ← PLAN-ROUTE(current, unvisited ∩ safe, safe)
  if plan пуст и ASK(KB, HaveArrow^t) = true then
    possible_wumpus ← {[x, y] : ASK(KB, ¬W_{x,y}) = false}
    plan ← PLAN-SHOT(current, possible_wumpus, safe)
  if plan пуст then        // Ничего больше не остается, как рискнуть
    not_unsafe ← {[x, y] : ASK(KB, ¬OK^t_{x,y}) = false}
    plan ← PLAN-ROUTE(current, unvisited ∩ not_unsafe, safe)
if plan пуст then
    plan ← PLAN-ROUTE(current, {[1, 1]}, safe) + [Climb]
  action ← POP(plan)
  TELL(KB, MAKE-ACTION-SENTENCE(action, t))
  t ← t + 1
  return действие action

function PLAN-ROUTE(current, goals, allowed) returns последовательность
                                                        действий
  inputs: current, текущее местонахождение агента
          goals, множество квадратов; попробовать спланировать маршрут
                  к одному из них
          allowed, множество квадратов, через которые может проходить
                  маршрут

  problem ← ROUTE-PROBLEM(current, goals, allowed)
  return SEARCH(problem)  // Любой алгоритм поиска из главы 3
```

Рис. 7.20. Программа гибридного агента для мира вампуса. Для логического вывода состояния мира в ней используется пропозициональная база знаний, а для выбора действий — комбинация решения задач поиском и кода, специфического для данной

среды. Каждый раз, когда вызывается функция HYBRID-WUMPUS-AGENT, она добавляет в базу знаний высказывание о текущем восприятии, а затем либо использует ранее созданный план, либо создает новый и выбирает первый этап этого плана в качестве действия, которое будет выполнено следующим

7.7.3. Оценка логического состояния

Программа агента, приведенная на рис. 7.20, работает довольно хорошо, но у нее есть один существенный недостаток: по мере того, как значение счетчика времени увеличивается, вычислительные затраты на выполнение запросов ASK неуклонно возрастают. Это происходит главным образом потому, что с ростом количества этапов при логическом выводе приходится возвращаться все дальше и дальше назад и обрабатывать все больше и больше пропозициональных символов. Очевидно, что это положение неприемлемо — нельзя полагаться на агентов, у которых затраты времени на обработку каждого очередного восприятия возрастают пропорционально продолжительности его жизни! В действительности нам нужно *постоянное* время обработки, не зависящее от t. Очевидным решением является сохранение или ►**кеширование** результатов логического вывода, благодаря чему процедура логического вывода на следующем этапе может основываться на результатах предыдущих этапов вместо того, чтобы вновь начинать все с нуля.

Как было показано в разделе 4.4, историю восприятий и всех их последствий можно успешно заменить **доверительным состоянием**, т.е. некоторым представлением множества всех возможных текущих состояний мира.[13] Процесс обновления доверительного состояния по мере поступления новых восприятий называют ►**оценкой состояния** (см. раздел 4.4.4). В то время как в разделе 4.4 доверительное состояние представляло собой определенный список состояний, здесь его можно представить как логическое высказывание, включающее пропозициональные символы, связанные с текущим моментом времени, а также вневременные символы. Например, логическое высказывание

$$WumpusAlive^1 \wedge L^1_{2,1} \wedge B_{2,1} \wedge (P_{3,1} \vee P_{2,2}) \qquad (7.4)$$

представляет множество всех состояний в момент времени 1, в котором вампус жив, агент находится в квадрате [2,1], в котором ощущается ветерок, и, следовательно, имеется яма в квадрате [3,1] или [2,2] или в обоих одновременно.

Однако выяснилось, что поддержание точного доверительного состояния в виде логической формулы — задача не из простых. Если имеется n символов флюент для момента времени t, то для них имеется 2^n возможных состояний, т.е.

[13] Можно и саму историю восприятия понимать как некоторое представление доверительного состояния, но такое, которое делает логический вывод все более и более дорогостоящим по мере того, как история становится длиннее.

различных присваиваний истинностных значений этим символам. Далее, множество доверительных состояний представляет собой надмножество (множество из всех подмножеств) множества физических состояний. Имеется 2^n физических состояний и, следовательно, 2^{2^n} доверительных состояний. Даже если использовать наиболее компактную из всех возможных форм кодирования логических формул, где каждое доверительное состояние представляется уникальным двоичным числом, для представления текущего доверительного состояния понадобятся числа из $\log_2(2^{2^n}) = 2^n$ бит. Иначе говоря, для точной оценки состояния могут потребоваться логические формулы, размер которых будет экспоненциально возрастать с увеличением количества символов.

Одной очень распространенной и естественной схемой для *приближенной* оценки состояния является представление доверительных состояний в виде конъюнкции литералов, т.е. 1-CNF-формул. Чтобы получить это, программа агента просто пытается доказать X^t и $\neg X^t$ для каждого символа X^t (так же, как и для каждого вневременного символа, истинностное значение которого пока не известно), учитывая доверительное состояние на этапе $t-1$. Конъюнкция доказуемых литералов становится новым доверительным состоянием, а предыдущее доверительное состояние отбрасывается.

Важно понимать, что с течением времени эта схема может терять некоторую информацию. Например, если высказывание в уравнении (7.4) считать истинным доверительным состоянием, то литералы $P_{3,1}$ и $P_{2,2}$ будут недоказуемы по отдельности и ни один из них не появился бы в доверительном состоянии в форме 1-CNF. (В упражнении 7.35 исследуется одно возможное решение этой проблемы.) С другой стороны, поскольку каждый литерал в доверительном состоянии в форме 1-CNF доказывается на основании предыдущего доверительного состояния, а исходное состояние убеждения является истинным утверждением, мы знаем, что все доверительное состояние в форме 1-CNF должно быть истинным. Таким образом, ➨ *множество возможных состояний, представленное доверительным состоянием в форме 1-CNF, включает все состояния, которые действительно возможны с учетом всей истории восприятия*. Как показано на рис. 7.21, доверительное состояние в форме 1-CNF действует как простая внешняя оболочка, или ▶ **первое приближение**, вокруг точного доверительного состояния. Подобную идею первых приближений к сложным множествам периодически можно обнаружить во многих областях ИИ.

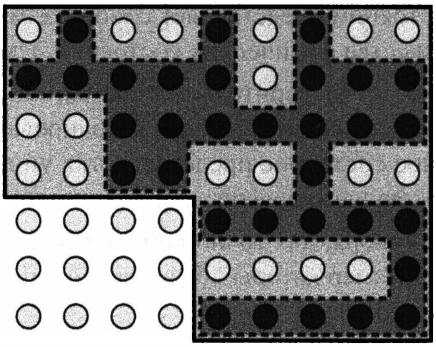


Рис. 7.21. Представление доверительного состояния в форме 1-CNF (сплошной контур) как легко представимого первого приближения к точному (сложной формы) доверительному состоянию (темно-серая область с пунктирным контуром). Каждый возможный мир представлен кружком, при этом кружки с темной заливкой отмечают те миры, которые совместимы со всей историей восприятия

7.7.4. Составление планов с использованием пропозиционального логического вывода

В программе агента на рис. 7.20 логический вывод используется только для определения, какие квадраты являются безопасными, а для составления планов используется поиск по алгоритму A*. В этом разделе показано, как можно составлять планы с помощью логического вывода. Основная идея очень проста.

1. Постройте высказывание, которое будет включать следующее:
   1) $Init^0$, набор утверждений о начальном состоянии;
   2) $Transition^1$, …, $Transition^t$, аксиомы определения преемника для всех возможных действий в каждый момент времени вплоть до максимально допустимого момента времени t;
   3) утверждение, что цель достигнута в момент времени t: $HaveGold^t \wedge ClimbedOut^t$.
2. Представьте полное высказывание в решатель SAT. Если решатель найдет модель, удовлетворяющую высказыванию, то цель является достижимой; если высказывание окажется невыполнимым, то задача неразрешима.
3. Если модель была найдена, извлеките из нее те переменные, которые представляют действия и которым было присвоено значение *true*. В своей совокупности они будут представлять план достижения цели.

На рис. 7.22 представлена функция пропозиционального планирования SATPlan. В ней реализована основная идея, только что предложенная выше, но

с одним изменением. Поскольку агент не знает, сколько этапов ему потребуется выполнить для достижения цели, алгоритм пробует каждое возможное число этапов t, вплоть до некоторой максимально возможной длины плана T_{max}. Таким образом, эта функция позволяет гарантированно найти кратчайший план, если таковой существует. По причине выбранного способа поиска решения функцию SATPLAN нельзя использовать в частично наблюдаемых средах, — функция SATPLAN просто присвоит ненаблюдаемым переменным те значения, которые ей потребуются для создания решения.

---

function SATPLAN(*init*, *transition*, *goal*, $T_{\max}$) **returns** решение или индикатор отказа *failure*
inputs: *init*, *transition*, *goal*, высказывания, в совокупности образующие описание задачи
$T_{\max}$, определяет верхний предел размера плана

for t = 0 **to** $T_{\max}$ **do**
  cnf ← TRANSLATE-TO-SAT(*init*, *transition*, *goal*, *t*)
  model ← SAT-SOLVER(*cnf*)
  if *model* не пуста **then**
    return EXTRACT-SOLUTION(*model*)
return *failure*

Рис. 7.22. Алгоритм SATPLAN. Задача планирования представляется в виде высказывания CNF, в котором утверждается, что цель достигнута на заданном этапе времени t, и в которое включены аксиомы для каждого этапа вплоть до t. Если алгоритм выполнимости найдет модель для этого высказывания, то план будет извлечен из нее за счет отбора тех пропозициональных символов, которые относятся к действиям и которым в модели назначены значения *true*. Если модели не существует, то процесс будет повторен с перемещением момента достижения цели на один этап позже

Ключевым шагом в использовании функции SATPLAN является построение базы знаний. На первый взгляд может показаться, что аксиом описания мира вампуса, предложенных в разделе 7.7.1, будет достаточно для выполнения пп. 1, *а* и 1, *б* из приведенной выше схемы реализации плана. Существуют, однако, значительные различия между требованиями, выдвигаемыми для осуществления логического вывода (как в функции запроса ASK) и выдвигаемыми для обеспечения проверки выполнимости.

Рассмотрим, например, описание местоположения агента, первоначально находящегося в квадрате [1,1], и предположим, что перед агентом поставлена скромная цель — оказаться в квадрате [2,1] в момент времени 1. Исходная база знаний содержит литералы стартового местонахождения $L^0_{1,1}$ и цели $L^1_{2,1}$. Воспользовавшись запросом ASK, можно доказать $L^1_{2,1}$, если утверждается $Forward^0$, и, конечно же,

нельзя доказать $L^1_{2,1}$, если вместо этого утверждается, скажем, $Shoot^0$. Итак, алгоритм SATPLAN найдет план $[Forward^0]$; пока все идет нормально.

К сожалению, алгоритм SATPLAN найдет и план $[Shoot^0]$. Как такое может произойти? Чтобы это выяснить, проверим модель, которую создаст SATPLAN. И вот сюрприз: она включает в себя присваивание $L^0_{2,1}$, т.е. агент может находиться в [2,1] в момент времени 1, будучи там в момент времени 0 и стреляя! Может возникнуть вопрос "Разве не было сказано, что в момент времени 0 агент находится в квадрате [1,1]?" Да, это было сказано, но агенту не было сказано, что он не может находиться в *двух* местах одновременно! Процессу логического вывода присваивание $L^0_{2,1}$ неизвестно и поэтому оно не может быть использовано в доказательстве. С другой стороны, процессу поиска модели присваивание $L^0_{2,1}$ также неизвестно и, следовательно, ему можно присвоить любое значение, которое поможет сделать литерал достижения цели истинным.

Алгоритм SATPLAN представляет собой отличный инструмент для отладки баз знаний, поскольку указывает на те места, где знания отсутствуют. В данном конкретном случае можно исправить базу знаний, добавив утверждение, что на каждом временном этапе агент может находиться точно в одном месте, воспользовавшись для этого полным набором высказываний, аналогичных тем, которые были использованы для утверждения о существовании ровно одного вампуса. В качестве альтернативы можно добавить утверждения $\neg L^0_{x,y}$ для всех квадратов, кроме [1,1], а аксиома выбора преемника для местоположения агента позаботится обо всех последующих моментах времени. Те же самые исправления сработают и для обеспечения гарантии, что на каждом этапе агент может иметь одну и только одну ориентацию.

Однако алгоритм SATPLAN содержит в себе больше сюрпризов. Во-первых, он находит модели с невозможными действиями, такими как стрельба без стрелы. Чтобы понять, почему, нужно внимательнее взглянуть на то, что аксиомы выбора преемника (такие, как в уравнении (7.3)) говорят о действиях, предварительные условия которых будут неудовлетворены. Эти аксиомы правильно предсказывают, что ничего не будет происходить, когда такое действие будет выполнено (см. упрупражнение 7.19), но они ничего не говорят о том, что такое действие не может быть выполнено! Для того чтобы избежать генерации планов с недопустимыми действиями, необходимо добавить ▶ **аксиомы предусловий**, утверждающие, что выполнение некоторого действия требует удовлетворения определенных предварительных условий.[14] Например, для каждого момента времени t следует сказать, что

$$Shoot^t \Rightarrow HaveArrow^t.$$

Это будет гарантировать, что если в любой момент времени в плане выбирается действие *Shoot*, то, чтобы это было возможно, у агента в этот момент должна быть стрела.

[14] Отметим, что добавление аксиом предусловий означает, что уже не нужно включать эти предварительные условия для действий в аксиомы выбора преемника.

Во-вторых, очередным сюрпризом алгоритма SATPLAN является создание планов с несколькими одновременными действиями. Например, он может предложить модель, в которой значения $Forward^0$ и $Shoot^0$ оба имеют значение *true*, что недопустимо. Чтобы устранить эту проблему, вводятся ▶**аксиомы исключения действия**: для каждой пары действий A_i^t и A_j^t добавляется аксиома

$$\neg A_i^t \vee \neg A_j^t .$$

Можно отметить, что идти вперед и одновременно стрелять — это не так уж и сложно, тогда как, скажем, стрелять и одновременно хватать — довольно непрактично. Накладывая аксиомы исключения действия только на те пары действий, которые действительно мешают выполнению друг друга, можно разрешить построение планов, включающих в себя несколько одновременных действий, и тогда, поскольку алгоритм SATPLAN всегда находит самый короткий допустимый план, можно не сомневаться, что он этой возможностью обязательно воспользуется.

Подводя итог, можно сказать, что алгоритм SATPLAN находит модели для высказывания, содержащего исходное состояние, цель, аксиомы выбора преемника, аксиомы предусловий и аксиомы исключения действия. Можно показать, что этот набор аксиом достаточен в том смысле, что алгоритм больше не будет вырабатывать никаких ложных "решений". Любая модель, удовлетворяющая пропозициональному высказыванию, будет представлять собой допустимый план решения исходной задачи. Современная технология решателей SAT делает этот подход довольно практичным. Например, решатель DPLL-типа не испытывает затруднений в выработке решений для экземпляра мира вампуса, приведенного на рис. 7.2.

В этом разделе был описан декларативный подход к конструированию агента: агент работает, комбинируя утверждение высказываний в базе знаний с выполнением логического вывода. У этого подхода есть некоторые недостатки, скрытые в таких фразах, как "для каждого момента времени t" и "для каждого квадрата $[x, y]$". Для любого практического агента эти фразы должны быть реализованы с помощью кода, который автоматически, исходя из общей схемы, генерирует экземпляры высказываний, предназначенных для вставки в базу знаний. Для мира вампуса разумного размера, сопоставимого с размерами наименьшей компьютерной игры, возможно, потребуется поле 100×100 квадратов и до 1000 временных этапов, что приводит к базе знаний с десятками или даже сотнями миллионов высказываний.

Мало того что этот подход оказывается довольно непрактичным, он также вскрывает более глубокую проблему: хотя мы знаем кое-что важное о мире вампуса — а именно, что "физика" в нем работает одинаково во всех квадратах и во все моменты времени, — мы не можем выразить это знание непосредственно на языке логики высказываний. Чтобы решить данную проблему, необходим более выразительный язык — такой, в котором фразы, подобные утверждениям "для каждого времени t" и "для каждого квадрата $[x, y]$", могут быть записаны естественным

образом. Логика первого порядка, обсуждаемая в главе 8, как раз и является таким языком. В логике первого порядка мир вампуса любого размера и с любым допустимым количеством этапов может быть описан примерно десятью логическими выражениями вместо десяти миллионов или десяти триллионов высказываний пропозиционной логики.

Резюме

В этой главе были представлены агенты, основанные на знаниях, и показано, как можно определить логику, с помощью которой такие агенты будут способны формировать рассуждения о мире, в котором они существуют. Основные идеи этой главы перечислены ниже.

- Интеллектуальным агентам требуются знания о мире для того, чтобы они могли находить хорошие решения.
- Знания представлены в агентах в форме **высказываний** на **языке представления знаний**, хранящихся в **базе знаний**.
- Агент, основанный на знаниях, состоит из базы знаний и механизма логического вывода. Он действует путем сохранения высказываний о мире в своей базе знаний, использования механизма логического вывода для получения новых высказываний и применения этих высказываний для принятия решения о том, какое действие следует предпринять.
- Язык представления знаний определяется с помощью **синтаксиса**, устанавливающего структуру высказываний, и **семантики**, определяющей **истинность** каждого высказывания в каждом из **возможных миров** или **модель** этого высказывания.
- Для понимания процесса формирования логических рассуждений крайне важным является отношение **следования** между высказываниями. Из высказывания α следует другое высказывание, β, если β является истинным во всех мирах, где истинно α. Эквивалентные определения включают понятия **допустимости** высказывания $\alpha \Rightarrow \beta$ и **невыполнимости** высказывания $\alpha \wedge \neg\beta$.
- Логический вывод — это процесс получения новых высказываний из старых. **Непротиворечивые** алгоритмы логического вывода обеспечивают получение *только таких* высказываний, которые являются логическими следствиями; **полные** алгоритмы обеспечивают получение *всех* высказываний, являющихся логическими следствиями.
- **Логика высказываний** — это очень простой язык, состоящий из **пропозициональных символов** и **логических связок**. Этот язык позволяет рассматривать высказывания, о которых известно, являются ли они истинными, являются ли они ложными или имеют полностью неизвестное логическое значение.

- При наличии постоянного словаря пропозициональных символов множество возможных моделей является конечным, поэтому логическое следствие можно проверить, просто перебирая модели. Эффективные алгоритмы логического вывода **на основе проверки модели** для логики высказываний включают методы поиска с возвратами и локального поиска и часто позволяют очень быстро решать крупные задачи.
- **Правила логического вывода** представляют собой шаблоны непротиворечивого логического вывода, которые могут использоваться для поиска доказательств. Правило **резолюций** позволяет создать полный алгоритм логического вывода для баз знаний, которые представлены в **конъюнктивной нормальной форме**. **Прямой логический вывод** и **обратный логический вывод** — это алгоритмы логического вывода, которые чрезвычайно естественно подходят для баз знаний, представленных в форме **хорновских выражений**.
- Методы **локального поиска**, такие как алгоритм WALKSAT, могут использоваться для поиска решений. Подобные алгоритмы являются непротиворечивыми, но не являются полными.
- Логическая **оценка состояния** предполагает создание и поддержку логического высказывания, описывающего множество возможных состояний в соответствии с историей восприятий. Каждый новый этап требует выполнения логического вывода с использованием модели мира окружающей среды, построенной на основании **аксиом выбора преемника**, определяющих, как изменяется каждая **флюента**.
- Логический агент может принимать решения посредством решения задач определения выполнимости высказываний (задач SAT): отыскивая возможные модели для определения последовательности будущих действий, приводящих к достижению цели. Такой подход работает только для полностью наблюдаемых или бессенсорных сред.
- Логика высказываний не может применяться к средам неограниченного размера, потому что ей не хватает выразительной силы для достаточно лаконичного представления времени, пространства и универсальных шаблонов отношений между объектами.

Библиографические и исторические заметки

В статье Джона Маккарти "Programs with Common Sense" ([1529], 1958; [1531], 1968) было впервые выдвинуто понятие агента, использующего логические рассуждения для установления связи между восприятиями и действиями. Также Маккарти стал основоположником декларативного подхода, указав, что способ создания программного обеспечения, предполагающий передачу агенту указаний о том, что он должен знать, является весьма изящным. В статье Аллена Ньюэлла

"The Knowledge Level" ([1664], 1982) подчеркнуто, что рациональные агенты могут быть описаны и проанализированы на абстрактном уровне, определяемом знаниями, которыми они обладают, а не программами, которые в них выполняются.

Сама логика имеет свои истоки в древнегреческой философии и математике. Платон рассматривал синтаксическую структуру высказываний, их истинность или ложность, их смысл и допустимость доводов. Первое известное систематическое исследование логики было опубликовано в трактате Аристотеля *Органон*. Его ►**силлогизмы** представляли собой то, что мы теперь называем правилами логического вывода, хотя им недоставало композиционности наших нынешних правил.

В V веке до н.э. мегарская и стоическая школы начали систематические исследования основных логических связок. Применение истинностных таблиц впервые предложил Филон из Мегары. Стоики приняли использование пяти основных правил логического вывода как допустимое без доказательства, в том числе правило, которое теперь принято называть правилом отделения (Modus Ponens). Из этих пяти правил они вывели множество других правил, используя, среди прочих принципов, теорему дедукции (раздел 7.5), и имели гораздо более четкое представление о самом понятии доказательства, чем Аристотель (Мэйтс [1515], 1953).

Идея сведения логического вывода к чисто механическому процессу принадлежит Вильгельму Лейбницу (1646–1716). Джордж Буль ([250], 1847) впервые представил полную и работоспособную систему формальной логики в своей книге *The Mathematical Analysis of Logic*. Логика Буля была почти полностью промоделирована на основе обычной алгебры действительных чисел, и в качестве основного метода логического вывода в ней использовалась подстановка логически эквивалентных выражений. Хотя систему Буля еще нельзя было считать полной логикой высказываний, другие математики сумели быстро заполнить все недостающие элементы. В 1877 году Шрёдер [2003] описал конъюнктивную нормальную форму, тогда как хорновская форма была введена намного позднее Альфредом Хорном ([1059], 1951). Первое полное описание современной логики высказываний (и логики первого порядка) можно найти в книге *Begriffschrift* ("Система обозначения понятий") Готтлоба Фреге ([775], 1879).

Первым механическим устройством для формирования логических выводов была машина *Demonstrator*, сконструированная третьим графом Стенхоупом (1753–1816). Уильям Стэнли Джевонс, один из математиков, расширивших результаты Буля, в 1869 году сконструировал "логическое фортепьяно", предназначенное для формирования логических выводов в булевой логике. Занимательная история этих и других ранних механических устройств логического вывода описана Мартином Гарднером ([813], 1968). Первыми компьютерными программами для формирования логического вывода были программа доказательств в арифметике Пресбургера (Дэвис [542], 1957) и программа *Logic Theorist*, разработанная Ньюэллом, Шоу и Саймоном ([1667], 1957).

Эмиль Пост ([1815], 1921) и Людвиг Витгенштейн ([2368], 1922) первыми независимо друг от друга использовали таблицы истинности в качестве метода

проверки правильности логических высказываний. Алгоритм Дэвиса–Патнема ([544], 1960) был первым эффективным алгоритмом резолюции в логике высказываний, но улучшенный алгоритм поиска с возвратами DPLL (Дэвис и др. [543], 1962), оказался более эффективным. Полное правило резолюций и доказательство его полноты были опубликованы в полной всеобщности для логики первого порядка Дж.Э. Робинсоном ([1900], 1965).

Стивен Кук ([472], 1971) показал, что задача определения выполнимости высказывания в логике высказываний (задача SAT) является NP-полной. Тем не менее известны многие подмножества логики высказываний, для которых задача проверки выполнимости решается за полиномиальное время; одним из таких подмножеств являются хорновские выражения.

Ранние исследования показали, что алгоритм DPLL имеет полиномиальную сложность в среднем случае для некоторых естественных распределений задач. Лучше того Франко и Полл ([769], 1983) показали, что одни и те же проблемы могут быть решены за *постоянное* время простым угадыванием случайных присваиваний. Мотивированные эмпирическими успехами локального поиска, Котсупиас и Пападимитриу ([1293], 1992) показали, что простой алгоритм поиска восхождением к вершине может решить *почти все* экземпляры задачи выполнимости очень быстро, и предположили, что трудные задачи встречаются относительно редко. Шёнинг ([1999], 1999) продемонстрировал рандомизированный алгоритм поиска восхождением к вершине, для которого *наихудшим* ожидаемым временем выполнения для задач 3-SAT является $O(1{,}333^n)$ — зависимость все еще экспоненциальная, но в значительной степени лучше, чем в предыдущих оценках для наихудшего случая. Текущим рекордом является $O(1{,}32216^n)$ (Рольф [1905], 2006).

Эффективность работы пропозициональных решателей быстро возрастала. Имея в своем распоряжении десять минут вычислительного времени, оригинальный алгоритм DPLL на оборудовании 1962 года был способен решать лишь задачи с 10–15 переменными (на ноутбуке выпуска 2019 года он смог бы решать задачи примерно с 30-ю переменными). В 1995 году решатель SATZ (Ли и Энбулаган, 1997) был способен обрабатывать до 1000 переменных — благодаря оптимизированным структурам данных для их индексации. Двумя важными вкладами стали метод ▶ **наблюдаемого литерала** — технология индексирования Чжана и Стикела ([2430], 1996), резко повышающая эффективность распространения единичных выражений, и введение выражения (т.е. ограничение) — технология обучения Байярдо и Шрега ([147], 1997). Используя эти идеи, вдохновленные перспективой решения проблем верификации цепей промышленного масштаба, Москевич и соавт. ([1627], 2001) разработали решатель CHAFF, способный справиться с задачами с миллионами переменных. Начиная с 2002 года проводятся ежегодные SAT-конкурсы и большинство их победителей являлись различными вариантами CHAF. Подробный обзор решателей SAT можно найти в работе Гомеса и соавт. ([887], 2008).

Алгоритмы локального поиска для определения выполнимости были опробованы различными авторами в течение 1980-х годов и строились на идее

минимизации числа неудовлетворенных выражений (Хансен и Жомар [961], 1990). Особенно эффективный алгоритм был разработан Гу ([928], 1989) и независимо от него Селманом и соавт. ([2025], 1992), который назвал его GSAT и показал, что он был способен на решение широкого спектра очень трудных задач за очень малое время. Алгоритм WALKSAT, описанный в этой главе, был предложен Селманом и соавт. ([2023], 1996).

"Фазовый переход" в отношении определения выполнимости сформированных случайным образом задач k-SAT впервые был отмечен Саймоном и Дюбуа ([2073], 1989) и дал начало множеству теоретических и эмпирических исследований, частично в связи с явлениями фазовых переходов в статистической физике. Кроуфорд и Отон ([494], 1993) обнаружили такой переход для задач 3-SAT, когда их значение отношения "высказывания/переменные" находится вблизи 4,26, заметив этот факт как резкий пик во времени выполнения их решателя SAT. Кук и Митчелл ([473], 1997) составили превосходный обзор ранней литературы по этой теме. Алгоритмы, такие как ▶**обзорное распространение** (*survey propagation*) ([1731], 2002; [1482], 2007), используют преимущества особых свойств случайных экземпляров SAT в районе порога определения выполнимости и значительно превосходят обычные решатели SAT для экземпляров задач такого типа. Текущее состояние теоретического понимания данной проблемы обобщил Ахлиоптас в [10] (2009).

Хорошими источниками информации о выполнимости как в теоретическом, так и в практическом плане можно считать справочник *Handbook of Satisfiability* (Бьере и др. [212], 2009), статью Дональда Кнута ([1246], 2015) и материалы *Регулярной международной конференции по теории и приложениям тестирования выполнимости* (*International Conferences on Theory and Applications of Satisfiability Testing*), известной как *SAT*.

Идею о построении агентов, использующих логику высказываний, можно проследить до оригинальной статьи Мак-Каллока и Питтса ([1537], 1943), которая, как известно, послужила началом исследований в области нейронных сетей, но в действительности фокусировалась на реализации проекта агента с мозгом на основе логических схем. Стен Розеншайн ([1916], 1985; [1166], 1990) разработал способы компиляции агентов на основе логических схем из декларативных описаний проблемной среды. Род Брукс ([312], 1986; [313], 1989) продемонстрировал эффективность использования проектов на основе логической схемы для управления роботами. Брукс утверждал ([314], 1991), что для искусственного интеллекта проекты на основе логических схем — это *все*, что необходимо, а методы представления и формирования логических рассуждений громоздки, дорогостоящи и излишни. Но, по мнению авторов, необходимы как рассуждения, так и логические цепи. Вильямс и соавт. ([2349], 2003) описывает гибридного агента — не слишком отличающегося от нашего агента для мира вампуса, — который управляет космическим аппаратом НАСА, выполняет планирование последовательности действий, а также диагностирует и устраняет неисправности.

Общая проблема слежения за частично наблюдаемой средой была введена с ее представлением на основе состояний в главе 4. Ее реализация для пропозициональных представлений была изучена Амиром и Расселом ([42], 2003), определившими несколько классов сред, допускающих эффективное применение алгоритмов оценки состояния и показавшими, что для некоторых других классов сред эта задача неразрешима. Задачу ▶**временной проекции**, включающую определение, какие высказывания остались истинными после выполнения последовательности действий, можно рассматривать как специальный случай оценки состояния с пустым восприятием. Многие авторы уже изучали эту проблему по причине ее важности для процесса планирования, и уже были получены некоторые важные и надежные результаты ([1410], 1997). Идея представления доверительного состояния с помощью высказываний пропозициональной логики может быть прослежена до работы Витгенштейна ([2368], 1922).

Подход к оценке логического состояния с использованием временных индексов для пропозициональных переменных был предложен Каутцем и Сельманом ([1201], 1992). В более поздних поколениях решателей SATPLAN удалось реализовать преимущества от достижений, достигнутых в решателях SAT, и они сохранили свои позиции среди наиболее эффективных способов решения сложных задач планирования (Каутц [1199], 2006).

Проблема фреймов впервые была признана Маккарти и Хейсом ([1529], 1969). Многие исследователи считали ее неразрешимой без использования логики первого порядка, и это стимулировало большое количество исследований в области немонотонной логики. Философы от Дрейфуса ([651], 1972) до Крокетта ([501], 1994) назвали проблему фреймов одним из симптомов неизбежного краха всего предприятия ИИ. Решение проблемы фреймов с помощью аксиом определения преемника принадлежит Рею Рейтеру ([1870], 1991). Тильшер ([2202], 1999) идентифицирует проблему представления фреймов как независимую идею и предлагает ее решение. Оглядываясь назад, можно видеть, что агенты Розеншайна ([1916], 1985) использовали схемы, в которых были реализованы аксиомы определения преемника, но Розеншайн не заметил, что проблема фрейма была тем самым в значительной степени решена.

Современные пропозициональные решатели нашли себе множество применений в различных отраслях промышленности, например в таких, как синтез компьютерного аппаратного обеспечения (Новик и др. [1702], 1993). Контроллер выполнимости SATMC использовался для обнаружения ранее неизвестных уязвимостей в протоколе единого входа в веб-браузерах (Армандо и др. [72], 2008).

Мир вампуса был придуман Грегори Йобом ([2407], 1975). Любопытно то, что Йоб разработал этот мир, поскольку ему надоели игры, которые проходили в мире, представленном в виде квадратной решетки: топология его первоначального мира вампуса представляла собой додекаэдр, а мы снова поместили вампуса в старую скучную квадратную решетку. Майкл Генезерет первым предложил использовать мир вампуса как испытательную площадку для агентов.

Упражнения

7.1. Предположим, что агент достиг пункта, показанного на рис. 7.4, *a*, получив такие результаты восприятия: в квадрате [1,1] — ничего, в квадрате [2,1] — ветерок, а в квадрате [1,2] — неприятный запах. В настоящий момент агента интересует содержимое квадратов [1,3], [2,2] и [3,1]. Каждый из них может содержать яму, и самое большее в одном из них может находиться вампус. На основе примера, приведенного на рис. 7.5, сконструируйте множество возможных миров. (Должно быть найдено 32 таких мира.) Отметьте миры, в которых существующая база знаний является истинной, а также те, в которых истинным является каждое из следующих высказываний:

α_2 = "В квадрате [2,2] нет ямы."

α_3 = "В квадрате [1,3] есть вампус."

На основании этого покажите, что $KB \models \alpha_2$ и $KB \models \alpha_3$.

7.2. На основании приведенных ниже рассуждений сможете ли вы доказать, что единорог — мифическое существо? А как насчет того, что это волшебное существо? Существо, вооруженное рогом?

Если единорог — это мифическое существо, то он бессмертен, а если не мифическое, то он — смертное млекопитающее. Если единорог либо бессмертен, либо является млекопитающим, то он вооружен рогом. Единорог является волшебным существом, если он вооружен рогом.

7.3. Рассмотрите задачу определения, является ли некоторое высказывание пропозициональной логики истинным в данной модели.

Напишите рекурсивный алгоритм PL-TRUE?(s, m), который возвращает значение *true* тогда и только тогда, когда высказывание s является истинным в модели m (где m присваивает истинностное значение каждому символу в s). Этот алгоритм должен заканчивать свою работу за время, линейно зависящее от размера высказывания. (Еще один вариант состоит в том, чтобы воспользоваться версией этой функции из оперативного репозитария кода.)

а) Приведите три примера высказываний, в отношении которых можно определить, являются ли они истинными или ложными, в частичной модели, в которой не заданы истинностные значения для некоторых символов.

б) Покажите, что в общем случае в частичной модели истинностное значение высказывания (если оно имеется) эффективно определить невозможно.

в) Доработайте свой алгоритм PL-TRUE? так, чтобы он иногда позволял судить об истинностном значении по частичным моделям, сохраняя вместе с тем свою рекурсивную структуру и линейную зависимость времени прогона от размера высказывания. Приведите три примера высказываний, истинность которых в частичной модели не обнаруживается вашим алгоритмом.

г) Проведите исследование, позволяет ли этот модифицированный алгоритм повысить эффективность алгоритма TT-ENTAILS?.

7.4. Какие из следующих выражений являются правильными?

а) $False \models True$.

б) $True \models False$.

в) $(A \wedge B) \models (A \Leftrightarrow B)$.

г) $A \Leftrightarrow B \models A \vee B$.

д) $A \Leftrightarrow B \models \neg A \vee B$.

е) $(A \wedge B) \Rightarrow C \models (A \Rightarrow C) \vee (B \Rightarrow C)$.
ж) $(C \vee (\neg A \wedge \neg B)) \equiv ((A \Rightarrow C) \wedge (B \Rightarrow C))$.
з) $(A \vee B) \wedge (\neg C \vee \neg D \vee E) \models (A \vee B) \wedge (\neg D \vee E)$.
и) $(A \vee B) \wedge \neg(A \Rightarrow B)$ выполнимо.
к) $(A \Leftrightarrow B) \wedge (\neg A \vee B)$ выполнимо.
л) $(A \Leftrightarrow B) \Leftrightarrow C$ имеет такое же количество моделей, как и $(A \Leftrightarrow B)$ для любого фиксированного множества пропозиционных символов, включающего A, B, C.

7.5. Какие из следующих выражений являются правильными?

а) $False \models True$.
б) $True \models False$.
в) $(A \wedge B) \models (A \Leftrightarrow B)$.
г) $A \Leftrightarrow B \models A \vee B$.
д) $A \Leftrightarrow B \models \neg A \vee B$.
е) $(A \vee B) \wedge (\neg C \vee \neg D \vee E) \models (A \vee B \vee C) \wedge (B \wedge C \wedge D \Rightarrow E)$.
ж) $(A \vee B) \wedge (\neg C \vee \neg D \vee E) \models (A \vee B) \wedge (\neg D \vee E)$.
з) $(A \vee B) \wedge \neg(A \Rightarrow B)$ выполнимо.
и) $(A \wedge B) \Rightarrow C \models (A \Rightarrow C) \vee (B \Rightarrow C)$.
к) $(C \vee (\neg A \wedge \neg B)) \equiv ((A \Rightarrow C) \wedge (B \Rightarrow C))$.
л) $(A \Leftrightarrow B) \wedge (\neg A \vee B)$ выполнимо.
м) $(A \Leftrightarrow B) \Leftrightarrow C$ имеет такое же количество моделей, как и $(A \Leftrightarrow B)$ для любого фиксированного множества пропозиционных символов, включающего A, B, C.

7.6. Докажите каждое из приведенных ниже утверждений.

а) Высказывание α является допустимым тогда и только тогда, когда $true \models \alpha$.
б) Для любого высказывания α истинно, что $false \models \alpha$.
в) $\alpha \models \beta$ тогда и только тогда, когда высказывание $(\alpha \Rightarrow \beta)$ допустимо.
г) $\alpha \equiv \beta$ тогда и только тогда, когда высказывание $(\alpha \Leftrightarrow \beta)$ допустимо.
д) $\alpha \models \beta$ тогда и только тогда, когда высказывание $(\alpha \wedge \neg\beta)$ невыполнимо.

7.7. Докажите или найдите контрпример для каждого из приведенных ниже утверждений.

а) Если $\alpha \models \gamma$ или $\beta \models \gamma$ (или и то, и другое), то $(\alpha \wedge \beta) \models \gamma$.
б) Если $(\alpha \wedge \beta) \models \gamma$, то $\alpha \models \gamma$ или $\beta \models \gamma$ (или и то, и другое).
в) Если $\alpha \models (\beta \vee \gamma)$, то $\alpha \models \beta$ или $\alpha \models \gamma$ (или и то, и другое).

7.8. Докажите или найдите контрпример для каждого из приведенных ниже утверждений.

а) Если $\alpha \models \gamma$ или $\beta \models \gamma$ (или и то, и другое), то $(\alpha \wedge \beta) \models \gamma$.
б) Если $\alpha \models (\beta \wedge \gamma)$, то $\alpha \models \beta$ и $\alpha \models \gamma$.
в) Если $\alpha \models (\beta \vee \gamma)$, то $\alpha \models \beta$ или $\alpha \models \gamma$ (или и то, и другое).

7.9. Рассмотрите словарь, состоящий только из четырех высказываний, A, B, C и D. Сколько существует моделей для каждого из следующих высказываний?

а) $B \vee C$
б) $\neg A \vee \neg B \vee \neg C \vee \neg D$
в) $(A \Rightarrow B) \wedge A \wedge \neg B \wedge C \wedge D$

7.10. В этой главе были определены четыре бинарные логические связки.

а) Есть ли какие-то иные логические связки, которые могут быть полезны?
б) Сколько бинарных связок может быть определено?
в) Почему некоторые из них не очень полезны?

7.11. Используя метод по своему выбору, проверьте каждую из эквивалентностей, приведенных в таблице на рис. 7.11.

7.12. Определите, является ли каждое из приведенных ниже высказываний допустимым, невыполнимым или ни тем, ни другим. Проверьте полученные результаты с помощью таблиц истинности или правил эквивалентности, приведенных в таблице на рис. 7.11.

а) $Smoke \Rightarrow Smoke$

б) $Smoke \Rightarrow Fire$

в) $(Smoke \Rightarrow Fire) \Rightarrow (\neg Smoke \Rightarrow \neg Fire)$

г) $Smoke \vee Fire \vee \neg Fire$

д) $((Smoke \wedge Heat) \Rightarrow Fire) \Leftrightarrow ((Smoke \Rightarrow Fire) \vee (Heat \Rightarrow Fire))$

е) $(Smoke \Rightarrow Fire) \Rightarrow ((Smoke \wedge Heat) \Rightarrow Fire)$

ж) $Big \vee Dumb \vee (Big \Rightarrow Dumb)$

7.13. Определите, является ли каждое из приведенных ниже высказываний допустимым, невыполнимым или ни тем, ни другим. Проверьте полученные результаты с помощью таблиц истинности или правил эквивалентности, приведенных в таблице на рис. 7.11.

а) $Smoke \Rightarrow Smoke$

б) $Smoke \Rightarrow Fire$

в) $(Smoke \Rightarrow Fire) \Rightarrow (\neg Smoke \Rightarrow \neg Fire)$

г) $Smoke \vee Fire \vee \neg Fire$

д) $((Smoke \wedge Heat) \Rightarrow Fire) \Leftrightarrow ((Smoke \Rightarrow Fire) \vee (Heat \Rightarrow Fire))$

е) $Big \vee Dumb \vee (Big \Rightarrow Dumb)$

ж) $(Big \wedge Dumb) \vee \neg Dumb$

7.14. Любое высказывание логики высказываний логически эквивалентно утверждению, что любой возможный мир, в котором это высказывание было бы ложным, не имеет места. На основании этого наблюдения докажите, что любое высказывание может быть записано в конъюнктивной нормальной форме.

7.15. Методом резолюции докажите высказывание $\neg A \wedge \neg B$ на основании выражений, приведенных в упр. 7.25.

7.16. В этом упражнении рассматривается связь между выражениями и импликативными высказываниями.

а) Покажите, что выражение $(\neg P_1 \vee \ldots \vee \neg P_m \vee Q)$ логически эквивалентно импликативному высказыванию $(P_1 \wedge \ldots \wedge P_m) \Rightarrow Q$.

б) Покажите, что каждое выражение (независимо от количества положительных литералов) может быть записано в форме $(P_1 \wedge \ldots \wedge P_m) \Rightarrow (Q_1 \vee \ldots \vee Q_n)$, где P и Q — пропозициональные символы. База знаний, состоящая из таких высказываний, находится в импликативной нормальной форме или в форме Ковальского (Ковальски [1296], 1979).

в) Составьте полное правило резолюций для высказываний в импликативной нормальной форме.

7.17. Согласно некоторым политическим экспертам, человек, который является радикалом (R), может быть избран (E), если он (или она) консерватив (C); в противном случае он избран быть не может.

а) Что из приведенного ниже является правильным представлением этого утверждения?

1) $(R \wedge E) \Leftrightarrow C$

2) $R \Rightarrow (E \Leftrightarrow C)$

3) $R \Rightarrow ((C \Rightarrow E) \vee \neg E)$

б) Какое из высказываний в п. *а* может быть представлено в форме хорновского выражения?

7.18. В этом упражнении рассматривается представление задач определения выполнимости (SAT) в виде задач УО.

а) Нарисуйте граф ограничений, соответствующий задаче SAT $(\neg X_1 \vee X_2) \wedge (\neg X_2 \vee X_3) \wedge \ldots \wedge (\neg X_{n-1} \vee X_n)$ для частного случая $n = 5$.

б) Сколько решений существует для этой общей задачи SAT в зависимости от значения n?

в) Предположим, что алгоритм поиска с возвратами используется для того, чтобы найти все решения для задачи SAT в виде задачи УО, приведенной в п. *а*. (Для того чтобы найти все решения задачи УО, достаточно модифицировать базовый алгоритм так, чтобы он продолжал поиск и после того, как очередное решение будет найдено.) Предположим, что переменные упорядочены в виде $X_1, \ldots, X_n$ и значения *false* следуют перед значениями *true*. Сколько времени потребуется алгоритму для завершения работы? (Запишите выражение $O(\cdot)$ как функцию от n.)

г) Мы знаем, что задачи SAT в форме хорновских выражений могут быть решены за линейное время с помощью прямого логического вывода (распространение единичных выражений). Мы также знаем, что каждая бинарная задача УО с древовидной структурой и конечной дискретной областью определения может быть решена за время, линейное по отношению к числу переменных. Эти два факта связаны? Поясните свой ответ.

7.19. В этом упражнении рассматривается представление задач определения выполнимости (SAT) в виде задач УО.

а) Нарисуйте граф ограничений, соответствующий задаче SAT $(\neg X_1 \vee X_2) \wedge (\neg X_2 \vee X_3) \wedge \ldots \wedge (\neg X_{n-1} \vee X_n)$ для частного случая $n = 4$.

б) Сколько решений существует для этой общей задачи SAT в зависимости от значения n?

в) Предположим, что алгоритм поиска с возвратами используется для того, чтобы найти все решения для задачи SAT в виде задачи УО, приведенной в п. *а*. (Для того чтобы найти все решения задачи УО, достаточно модифицировать базовый алгоритм так, чтобы он продолжал поиск и после того, как очередное решение будет найдено.) Предположим, что переменные упорядочены в виде $X_1, \ldots, X_n$ и значения *false* следуют перед значениями *true*. Сколько времени потребуется алгоритму для завершения работы? (Запишите выражение $O(\cdot)$ как функцию от n.)

г) Мы знаем, что задачи SAT в форме хорновских выражений могут быть решены за линейное время с помощью прямого логического вывода (распространение единичных выражений). Мы также знаем, что каждая бинарная задача УО с древовидной структурой и конечной дискретной областью определения может быть решена за время, линейное по отношению к числу переменных. Эти два факта связаны? Поясните свой ответ.

7.20. Объясните, почему каждое непустое пропозициональное выражение само по себе выполнимо. Докажите строго, что любое множество из пяти выражений 3-SAT является выполнимым при условии, что в каждом выражении упоминаются ровно три различные переменные. Каково наименьшее множество подобных выражений, которое будет невыполнимым? Постройте такое множество.

7.21. Пропозициональное выражение 2-CNF представляет собой совокупность выражений, каждое из которых содержит ровно 2 литерала, например $(A \vee B) \wedge (\neg A \vee C) \wedge (\neg B \vee D) \wedge (\neg C \vee G) \wedge (\neg D \vee G)$.

а) Используя резолюцию, докажите, что из приведенного выше выражения следует G.

б) Два выражения семантически различны, если они не являются логически эквивалентными. Сколько семантически различных выражений 2-CNF может быть построено из n пропозиционных символов?

в) Используя свой ответ из п. *б*, докажите, что пропозициональная резолюция всегда заканчивается за время, полиномиально зависящее от n, если высказывание в форме 2-CNF содержит не более n различных символов.

г) Поясните, почему ваша аргументация в п. *в* неприменима к высказываниям в форме 3-CNF.

7.22. Докажите каждое из приведенных ниже утверждений.

а) Каждая пара пропозициональных выражений либо не имеет резольвент, либо все их резольвенты являются логически эквивалентными.

б) Не существует выражения, которое при резолюции с самим собой дает (после факторизации) выражение $(\neg P \vee \neg Q)$.

в) Если пропозициональное выражение C может быть подвергнуто резолюции со своей копией, то оно должно быть логически эквивалентно *True*.

7.23. Рассмотрим следующее высказывание: $[(Food \Rightarrow Party) \vee (Drinks \Rightarrow Party)] \Rightarrow [(Food \wedge Drinks) \Rightarrow Party]$.

а) Определите, используя перечисление, является ли это высказывание выполнимым (но не допустимым) или невыполнимым.

б) Преобразуйте левую и правую части основной импликации в форму CNF, письменно фиксируя каждый этап своих действий, и объясните, как полученные результаты подтверждают ваш ответ из п. *а*.

Докажите свой ответ на п. *а*, используя резолюцию.

7.24. Высказывание находится в дизъюнктивной нормальной форме (DNF), если оно представляет собой дизъюнкцию конъюнкций литералов. Например, высказывание $(A \wedge B \wedge \neg C) \vee (\neg A \wedge C) \vee (B \wedge \neg C)$ представлено в DNF.

а) Любое высказывание пропозициональной логики логически эквивалентно утверждению, что существует некоторый возможный мир, в котором оно было бы верно. Исходя из этого наблюдения, докажите, что любое высказывание может быть написано в форме DNF.

б) Разработайте алгоритм, преобразующий любое высказывание логики высказываний в форму DNF. (*Подсказка*: этот алгоритм аналогичен алгоритму преобразования в CNF.)

в) Разработайте простой алгоритм, на вход которого в качестве входных данных передаются высказывания в форме DNF и который возвращает выполняющее присваивание, если оно будет найдено, либо сообщает, что выполняющего присваивания не существует.

г) Примените алгоритмы из пп. *б* и *в* к следующему множеству высказываний.

$A \Rightarrow B$

$B \Rightarrow C$

$C \Rightarrow A$

д) Хотя алгоритм в п. *б* очень похож на алгоритм преобразования в CNF, а алгоритм в п. *в* намного проще, чем любой алгоритм для решения множества высказываний в форме CNF, почему эти методы не используются в автоматизированных рассуждениях?

7.25. Преобразуйте высказывания в следующем множестве в форму выражений.

S_1: $A \Leftrightarrow (B \vee E)$
S_2: $E \Rightarrow D$
S_3: $C \wedge F \Rightarrow \neg B$
S_4: $E \Rightarrow B$
S_5: $B \Rightarrow F$
S_6: $B \Rightarrow C$

Проследите за выполнением алгоритма DPLL на конъюнкции этих выражений.

7.26. Преобразуйте высказывания в следующем множестве в форму выражений.

S_1: $A \Leftrightarrow (C \vee E)$
S_2: $E \Rightarrow D$
S_3: $B \wedge F \Rightarrow \neg C$
S_4: $E \Rightarrow C$
S_5: $C \Rightarrow F$
S_6: $C \Rightarrow$ B

Проследите за выполнением алгоритма DPLL на конъюнкции этих выражений.

7.27. Является ли случайно сгенерированное высказывание в форме 4-CNF с n символами и m выражениями с большей или меньшей вероятностью разрешимым, чем случайно сгенерированное высказывание в форме 3-CNF с n символами и m выражениями? Поясните свой ответ.

7.28. Популярная компьютерная игра *Сапер* (*Minesweeper*) близко соотносится с миром вампуса. Мир сапера — это прямоугольная сетка из N квадратов с M невидимыми минами, разбросанными среди них. Агент может исследовать любой квадрат, и, если он содержит мину, мгновенная гибель неизбежна. Игра указывает на наличие мин выводом подсказки для каждого проверенного квадрата о количестве мин, которые находятся непосредственно рядом с ним или по диагонали. Цель состоит в том, чтобы исследовать каждый свободный от мин квадрат.

а) Пусть высказывание $X_{i,j}$ является истинным тогда, когда в квадрате $[i, j]$ находится мина. Составьте утверждение, согласно которому в квадратах, соседних по отношению к квадрату [1,1], имеются точно две мины, построив его в виде высказывания, включающего определенную логическую комбинацию высказываний $X_{i,j}$.

б) Обобщите свое утверждение, составленное при выполнении п. *а*, и объясните, как следует формировать высказывание в форме CNF, утверждающее, что k из n соседних квадратов содержат мины.

в) Точно объясните, как агент может использовать алгоритм DPLL для доказательства того, что данный квадрат содержит (или не содержит) мину, игнорируя глобальное ограничение, согласно которому всего имеется точно M мин.

г) Предположим, что глобальное ограничение конструируется с помощью метода, созданного вами в п. *б*. Как зависит количество выражений в этом ограничении от M и N? Предложите способ осуществления такой модификации алгоритма DPLL, чтобы в нем не требовалось явно представлять это глобальное ограничение.

д) Становятся ли недействительными какие-либо выводы, полученные при создании метода в п. *в*, если учитывается это глобальное ограничение?

е) Приведите примеры конфигураций проверочных значений, которые порождают такие далеко идущие зависимости, что содержимое любого непроверенного квадрата может предоставить информацию о содержимом какого-то далеко отстоящего от него квадрата. (*Подсказка*: рассмотрите доску размером $N \times 1$.)

7.29. Сколько времени потребуется, чтобы доказать высказывание $KB \models \alpha$ с помощью алгоритма DPLL, если α — литерал, который уже содержится в базе знаний KB? Поясните свой ответ.

7.30. Проследите за поведением алгоритма DPLL при обработке приведенной на рис. 7.16 базы знаний в попытке доказать высказывание Q и сравните это поведение с тем, которое демонстрируется алгоритмом прямого логического вывода.

7.31. Напишите аксиому выбора преемника для предиката *Locked*, который применяется к дверям, при условии, что доступны только действия *Lock* и *Unlock*.

7.32. Обсудите понятие оптимального поведения в мире вампуса. Покажите, что приведенное в этой главе определение алгоритма HYBRID-WUMPUS-AGENT не является оптимальным, и предложите способы его усовершенствования.

7.33. Предположим, что агент обитает в мире с двумя состояниями, S и $\neg S$, и может выполнять только одно из двух действий, a или b. Действие a — это ничего не делать, а действие b переводит агента из одного состояния в другое. Пусть S^t будет утверждением, что агент находится в состоянии S в момент времени t, и пусть a^t будет утверждением, что агент выполняет действие a в момент времени t (аналогично для b^t).

а) Напишите аксиому выбора преемника для S^{t+1}.

б) Преобразуйте высказывание из п. *а* в форму CNF.

в) Приведите доказательство (приведением к противоречию) того, что если агент находится в состоянии $\neg S$ в момент времени t и выполняет действие a, он все равно будет находиться в состоянии $\neg S$ в момент $t+1$.

7.34. В разделе 7.7.1 приводятся некоторые аксиомы выбора преемника, необходимые для мира вампуса. Запишите аксиомы выбора преемника для всех оставшихся символов флюент в этом мире.

7.35. Измените алгоритм HYBRID-WUMPUS-AGENT таким образом, чтобы использовать метод оценки логического состояния 1-CNF, описанный в разделе 7.7.3. Там было отмечено, что такой агент не сможет приобретать, поддерживать и использовать более сложные доверительные состояния, такие как дизъюнкция $P_{3,1} \vee P_{2,2}$. Предложите метод преодоления этой проблемы посредством определения дополнительных пропозициональных символов, и опробуйте его в мире вампуса. Улучшит ли такое решение производительность агента?

ГЛАВА 8

Логика первого порядка

В этой главе мы обращаем внимание на то, что мир наполнен множеством объектов, одни из которых связаны с другими объектами, а также пытаемся рассуждать об этих объектах.

Логика высказываний вполне позволяет проиллюстрировать основные понятия логики и логического вывода, а также принципы функционирования агентов, основанных на знаниях. Но, к сожалению, язык этой логики слишком ограничен и недостаточно выразителен. В настоящей главе речь пойдет о ▶**логике первого порядка**,[1] позволяющей в сжатом виде представить гораздо больше. Изложение этой темы начнем с описания в разделе 8.1 всех языков представления в целом; в разделе 8.2 рассматриваются синтаксис и семантика логики первого порядка, а в разделах 8.3 и 8.4 иллюстрируется использование логики первого порядка для простых представлений.

8.1. Еще раз о представлении

В этом разделе обсуждается общая природа языков представления. Языки программирования (такие, как C++, Java или Python) являются самым большим классом формальных языков из числа получивших широкое распространение. Для представления фактов в программах могут быть использованы структуры данных, например массив 4×4 может использоваться в программе для представления содержимого мира вампуса. Так, оператор языка программирования *World*[2,2] ← *Pit* можно считать вполне естественным средством объявления, что в квадрате [2,2] есть яма. Если строки с подобными операторами собрать вместе, этого будет достаточно для выполнения имитации мира вампуса.

Чего не хватает языкам программирования, так это общего механизма вывода фактов из других фактов: любое обновление структуры данных выполняется с помощью процедур, специфических для проблемной области задачи, детали которых определяются программистом на основании его собственных знаний об этой

[1] Логику первого порядка также иногда называют ▶**исчислением предикатов первого порядка** и сокращенно обозначают как **FOL** — *First-Order Logic* или **FOPC** — *First-Order Predicate Calculus*.

области. Такой процедурный подход может быть противопоставлен **декларативной** природе логики высказываний, в которой знания и методы логического вывода не смешиваются и логический вывод выполняется совершенно независимо от проблемной области задачи. Сочетание декларативных и процедурных знаний используется в базах данных SQL.

Другим недостатком структур данных в программах (и базах данных) является отсутствие какого-либо простого способа сделать, например, такое утверждение: "В квадрате [2,2] или [3,1] имеется яма" или "Если вампус находится в квадрате [1,1], то его нет в квадрате [2,2]". Программы позволяют хранить для каждой переменной лишь единственное значение, и хотя некоторые системы допускают использование "неопределенных" значений, им не хватает выразительности, необходимой для обработки частичной информации.

Логика высказываний — это декларативный язык, поскольку ее семантика основана на истинностных отношениях между высказываниями и возможными мирами. Кроме того, она имеет достаточную выразительную мощь для того, чтобы с ее помощью можно было обрабатывать частично заданную информацию с использованием дизъюнкции и отрицания. Логика высказываний обладает еще одним свойством, которое является желательным для языков представления, а именно ▶ **композициональностью**. В композициональном языке смысл высказывания представляет собой функцию от смысла его частей. Например, смысл высказывания "$S_{1,4} \wedge S_{1,2}$" связан со смыслом высказываний "$S_{1,4}$" и "$S_{1,2}$". Было бы очень странно, если бы "$S_{1,4}$" означало, что в квадрате [1,4] чувствуется неприятный запах, а "$S_{1,2}$" — что неприятный запах чувствуется в квадрате [1,2], и при этом выражение "$S_{1,4} \wedge S_{1,2}$" означало бы, что квалификационный матч по хоккею с шайбой между Францией и Польшей, проходивший на прошлой неделе, закончился со счетом 1:1.

Однако логике высказываний как факторизованному представлению не хватает выразительной силы, которая позволяла бы *кратко* описывать среду со многими объектами. Например, мы были вынуждены записывать отдельное правило, связывающее между собой присутствие ветерка и наличие ям, для каждого квадрата, скажем, таким образом:

$$B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1}).$$

На естественном языке, русском или английском, кажется, совсем несложно раз и навсегда сформулировать утверждение: "В квадратах, непосредственно примыкающих к ямам, чувствуется ветерок". Синтаксис и семантика естественного языка позволяют кратко описать эту среду: естественные языки, как и логика первого порядка, являются структурированным представлением.

8.1.1. Язык мышления

Естественные языки (такие, как русский, английский или испанский) действительно очень выразительны. Нам удалось написать почти всю эту книгу на естественном языке, лишь изредка обращаясь к формальным языкам (в основном к математическим и к диаграммам). В лингвистике и философии языка давно существует традиция, в которой естественный язык рассматривается как декларативный язык представления знаний. Если бы удалось раскрыть правила естественного языка, можно было бы использовать их в системах представления и рассуждения и получить пользу от миллиардов страниц, уже написанных на естественном языке.

Современный взгляд на естественный язык состоит в том, что он служит средством *общения*, а не чистого представления. Когда говорящий указывает куда-то и говорит "Смотри!", слушатель в конечном счете может узнать, что Супермен, наконец-то, появился над крышами домов. Тем не менее нельзя утверждать, что высказывание "Смотри!" представляет именно этот факт. Скорее смысл этого высказывания зависит как от самого высказывания, так и от *контекста*, в котором оно было произнесено. Очевидно, что в базе знаний невозможно сохранить такое высказывание, как "Смотри!", и ожидать, что удастся восстановить его точный смысл, если в базе знаний не будет также представлен соответствующий контекст, из чего следует вопрос "А как можно представить сам этот контекст?"

Также естественные языки страдают от *неоднозначности*, что может стать проблемой для языка представления. Пинкер ([1793], 1995) выразил эту мысль таким образом: "Когда люди думают о *косе*, то их, безусловно, не затрудняет определение того, идет ли речь о женской прическе, о сельскохозяйственном инвентаре или о чем-то другом, — если одно и то же слово может соответствовать нескольким понятиям, сами понятия не могут быть словами".

Знаменитая **гипотеза Сепира–Уорфа** (Уорф [2333], 1956) утверждает, что наше понимание мира сильно зависит от языка, на котором мы говорим. Безусловно верно, что различные речевые сообщества видят мир и выделяют его элементы по-разному. У говорящих на английском языке есть два слова, "autumn" и "fall", для понятия, которое носители русского языка обозначают одним: "осень". Но говорящие на русском языке могут легко распознать категорию *fall* и дать ей собственное имя (дословно — "период листопада"), так язык ли в действительности вызывает различия? Уорф в основном полагался на интуицию и предположения, и его идеи были в основном отвергнуты, но в последующие годы были получены реальные данные в результате проведенных антропологических, психологических и неврологических исследований.

Например, кто из вас сейчас может вспомнить, с какой именно из следующих двух фраз начинался раздел 8.1?

"В этом разделе обсуждается общая природа языков представления..."

"В данном разделе рассматривается тема представления знаний в языках..."

Ваннер ([2295], 1974) провел аналогичный эксперимент и обнаружил, что испытуемые делали правильный выбор на уровне случайности — примерно в 50% случаев, — но помнили содержание прочитанного с точностью, превышающей 90%. Эти данные свидетельствуют о том, что люди воспринимают слова, которые читают, а затем формируют своего рода *несловесное* представление, в котором точные слова уже не имеют значения.

Более интересным является случай, когда концепция полностью отсутствует в языке. Говорящие на языке аборигенов Австралии "гуугу йимитир" не имеют ни одного слова для обозначения относительных (или *эгоцентрических*) направлений, таких как *спереди*, *сзади*, *справа* или *слева*. Вместо этого они используют абсолютные направления, говоря, например, что-то вроде "у меня болит северная рука". Это различие в языке приводит и к определенным различиям в поведении: говорящие на гуугу йимитир лучше ориентируются на открытых просторах, в то время как говорящие на английском лучше ориентируются, когда их просят положить вилку справа от тарелки.

Похоже, что язык также оказывает влияние на мышление через такие, казалось бы, произвольные грамматические функции, как род существительных. Например, слово "мост" в испанском языке мужского рода, а в немецком — женского. Бородицкий ([256], 2003) попросил испытуемых выбрать английские прилагательные для описания фотографии определенного моста. Говорящие на испанском выбрали слова *big* (большой), *dangerous* (опасный), *strong* (прочный) и *towering* (высокий), в то время как говорящие на немецком выбрали эпитеты *beautiful* (прекрасный), *elegant* (элегантный), *fragile* (хрупкий) и *slender* (стройный).

Слова могут служить опорными точками, влияющими на то, как мы воспринимаем мир. Лофтус и Палмер ([1439], 1974) показали участникам эксперимента фильм об автомобильной аварии. Испытуемые, которых спросили "Насколько быстро двигались машины, когда они вошли в контакт друг с другом?", в среднем давали оценку 70 км/ч, в то время как те, которым был задан тот же вопрос, но со словом "врезались" вместо слов "вошли в контакт", оценили скорость в 80 км/ч для тех же автомобилей в том же фильме. В целом действительно имеют место измеримые, но небольшие по величине различия в когнитивных процессах у людей, говорящих на разных языках, но нет убедительных доказательств того, что это приводит к существенным различиям в восприятии мира.

В системе логического рассуждения, использующей конъюнктивную нормальную форму (CNF), легко увидеть, что лингвистические формы "$\neg(A \vee B)$" и "$\neg A \wedge \neg B$" одинаковы, поскольку можно заглянуть внутрь системы и увидеть, что эти два высказывания хранятся в одной и той же канонической форме CNF. В последнее время становится возможным сделать что-то подобное и с человеческим мозгом. Митчелл и соавт. ([1599], 2008) в своем эксперименте помещали испытуемых в аппарат функциональной магнитно-резонансной томографии (фМРТ), показывали им такие слова, как "сельдерей", и фиксировали изображение состояния их мозга. Система машинного обучения, обучаемая на парах "*слово*, *изображение*",

в конечном счете оказалась способной в 77% случаев дать правильный ответ в задачах бинарного выбора (например, между словами "сельдерей" и "самолет"). Система оказалась способной даже успешно прогнозировать на уровне выше обычного вероятности и в случае слов, для которых ей ранее никогда не представляли изображение фМРТ (анализируя изображения связанных слов), и в отношении людей, томограммы которых она никогда не видела раньше (что доказывает, что фМРТ вскрывает некоторый уровень общего представления для всех людей). Этот тип исследований все еще находится в младенческом состоянии, но фМРТ и другие технологии визуализации, такие как внутричерепная электрофизиология (Сахин и др. [1963], (2009)), обещают дать гораздо более конкретные идеи о том, на что похоже представление знаний у человека.

С точки зрения формальной логики представление одного и того же знания двумя различными способами не имеет абсолютно никакого значения, — одни и те же факты можно будет вывести из любого представления. Однако на практике для получения результата одно представление может потребовать выполнения меньшего количества этапов, чем другое, а это означает, что при наличии ограниченных ресурсов рассуждающий сможет получить результат, лишь используя первое представление, но не второе. Для *недедуктивных* задач, таких как обучение на основе накопления опыта, результаты *обязательно* зависят от используемой формы представления. В главе 19 будет показано, что, когда обучаемая программа рассматривает две возможные теории мира, обе из которых полностью согласуются со всеми имеющимися данными, общепринятый подход решения этой дилеммы состоит в выборе наиболее лаконичной теории, — а это, в свою очередь, зависит от языка, использованного для представления теорий. Таким образом, влияние языка на мышление неизбежно для любого агента, который обучается.

8.1.2. Комбинирование лучших черт формальных и естественных языков

Принятый авторами подход состоит в принятии основ логики высказываний — декларативной, композиционной семантики, независимой от контекста и непротиворечивой — и построении на этой основе более выразительной логики, заимствуя идеи представления из естественного языка, но при этом избегая его недостатков. Изучение синтаксиса естественного языка показывает, что наиболее очевидными его элементами являются существительные и именные конструкции, которые обозначают ▶**объекты** (квадраты, ямы, вампусы), а также глаголы и глагольные конструкции, включающие также прилагательные и наречия, обозначающие ▶**отношения** между объектами (чувствовать ветерок, быть соседним, стрелять). Некоторые из этих отношений являются ▶**функциями**, т.е. отношениями, в которых имеется только одно "значение", выдаваемое в ответ на определенный "входной сигнал". Такой подход позволяет сразу же приступить к составлению перечня примеров объектов, отношений и функций, как показано ниже.

- *Объекты*: люди, дома, числа, теории, Рональд Макдональд, цвета, бейсбольные матчи, войны, столетия...
- *Отношения*: могут быть унарными отношениями или ▶**свойствами**, такими как красный, круглый, поддельный, первичный, многоэтажный... или более общими *n*-арными отношениями, такими как "быть братьями", "быть больше", "находиться внутри", "входить в состав", "иметь цвет", "произойти позже", "владеть", "находиться между"...
- *Функции*: "быть отцом", "быть лучшим другом", "быть третьей подачей мяча", "быть на единицу больше", "быть началом"...

Безусловно, почти каждое утверждение может рассматриваться как обозначающее объекты, а также их свойства или отношения. Ниже приведены некоторые примеры.

- "Один плюс два равняется трем".

  Объекты: один, два, три, один плюс два; *отношение*: равняется; *функция*: плюс. ("Один плюс два" представляет собой название объекта, полученного путем применения функции "плюс" к объектам "один" и "два". "Три" — другое название для этого объекта.)
- "В квадратах, соседних с тем, где находится вампус, есть неприятный запах".

  Объекты: вампус, квадраты; *свойство*: неприятный запах; *отношение*: быть соседним.
- "Злой король Джон правил Англией в 1200 году".

  Объекты: Джон, Англия, 1200 год; *отношение*: правил; *свойства*: злой, король.

Язык **логики первого порядка**, синтаксис и семантика которого будут определены в следующем разделе, основан на понятиях объектов и отношений. Он стал чрезвычайно важным для математики, философии и искусственного интеллекта именно потому, что эти области знаний — а фактически основная часть повседневного человеческого существования — могут достаточно продуктивно рассматриваться как касающиеся объектов и отношений между ними. Логика первого порядка позволяет также выражать факты о *некоторых* или обо *всех* объектах во Вселенной. Именно это качество обеспечивает ей возможность представлять общие законы или правила, например такие, как следующее утверждение: "В квадратах, соседних с тем, где находится вампус, есть неприятный запах".

Основное различие между логикой высказываний и логикой первого порядка заключается в различном ▶**онтологическом вкладе** каждого из этих языков, т.е. они по-разному представляют сущность *действительности*. Математически это обязательство выражается через основные свойства формальных моделей, по отношению к которым определяется истинность высказываний. Например, в логике высказываний предполагается, что существуют лишь факты, которые относятся

или не относятся к данному миру. Каждый факт может находиться в одном из двух состояний (быть *истинным* или *ложным*), и в каждой модели значения *true* и *false* присваиваются каждому пропозициональному символу (см. раздел 7.4.2). В логике первого порядка приняты более широкие предположения, а именно — что мир состоит из объектов с определенными отношениями между ними, которые могут существовать или не существовать (рис. 8.1). Эти формальные модели являются, соответственно, более сложными, чем модели логики высказываний.

Язык	Онтологический вклад (что существует в мире)	Эпистемологический вклад (степень доверия, выражаемая агентом в отношении фактов)
Логика высказываний	Факты	Истинно/ложно/неизвестно
Логика первого порядка	Факты, объекты, отношения	Истинно/ложно/неизвестно
Временная логика	Факты, объекты, отношения, интервалы времени	Истинно/ложно/неизвестно
Теория вероятностей	Факты	Степень доверия $\in [0,1]$
Нечеткая логика	Факты со степенью истинности $\in [0,1]$	Известное интервальное значение

Рис. 8.1. Формальные языки и их онтологический и эпистемологический вклад

Этот онтологический вклад есть сильнейшая сторона логики (как пропозициональной, так и первого порядка), поскольку позволяет нам начинать с истинных утверждений и выводить из них другие истинные утверждения. Особенно ярко эта сильная сторона проявляется в областях, где каждое суждение имеет четкие границы, например в математике или в мире вампуса, где любой квадрат либо содержит, либо не содержит ямы и абсолютно невозможно существование квадрата с неопределенным углублением, напоминающим яму. Однако в реальном мире многие суждения имеют нечеткие границы: "Вена большой город?", "В этом ресторане подают вкусные блюда?", "Этот человек высокий?" Все зависит от того, кого вы спрашиваете, и полученный ответ может быть таким: "Как будто".

Один из вариантов реакции на это — уточнить представление: если грубое разделение городов на "большие" и "небольшие" оставляет слишком много неопределенности для рассматриваемого приложения, то можно увеличить количество категорий размеров городов или использовать символ функции *Population* (население). Другое возможное решение предлагает ▶ **нечеткая логика**, в которой вводится онтологическое расширение, допускающее, что любые утверждения могут иметь ▶ **степень истинности** в пределах от 0 до 1. Например, утверждение "Вена является большим городом" в нечеткой логике может иметь степень истинности 0,8, тогда как утверждение "Париж является большим городом" может иметь степень истинности 0,9. Подобный подход больше соответствует нашим интуитивным представлениям о мире, но одновременно делает логический вывод более

сложным: вместо одного правила определения истинности высказывания $A \wedge B$ нечеткая логика предполагает использование разных правил в зависимости от проблемной области задачи. Еще одна возможность, рассматриваемая в разделе 24.1, заключается в связывании каждой концепции с точкой в многомерном пространстве с последующим измерением расстояния между концепцией "большой город" и концепциями "Вена" или "Париж".

Различные логики специального назначения дают еще больше онтологических расширений. Например, во ▶**временной логике** предполагается, что факты имеют место в определенное *время* и что эти моменты времени (которые могут быть представлены точными значениями или интервалами) упорядочены. Таким образом, в разных вариантах логики специального назначения некоторым видам объектов особого рода (и аксиомам об этих объектах) придается статус "первого класса", вместо того чтобы просто ввести их определения в базу знаний. В ▶**логике высшего порядка** как объекты трактуются и сами отношения и функции, как они определяются в логике первого порядка. Это позволяет формировать утверждения обо *всех* отношениях, например если потребуется точно определить, что означает понятие транзитивного отношения. В отличие от большинства логик специального назначения, логика высшего порядка является строго более выразительной, чем логика первого порядка, в том смысле, что некоторые высказывания логики высшего порядка не могут быть выражены с помощью любого конечного количества высказываний логики первого порядка.

Логику можно также охарактеризовать по ее ▶**эпистемологическому вкладу**, — под этим подразумеваются возможные состояния знания, которые она позволяет выразить в отношении каждого факта. И в логике высказываний, и в логике первого порядка любое высказывание представляет собой факт, и агент либо доверяет утверждению, что это высказывание истинно, либо доверяет утверждению, что оно ложно, либо не имеет мнения на этот счет. Поэтому в таких вариантах логики имеются три возможных состояния знания в отношении любого высказывания.

С другой стороны, в системах, где используется **теория вероятностей**, может иметь место любая *степень доверия* (или *субъективная вероятность*), начиная от 0 (полное неверие) и заканчивая 1 (полное доверие). Важно не путать степень доверия в теории вероятностей со степенью истинности в *нечеткой* логике, хотя, конечно, некоторые нечеткие системы допускают неопределенность (степень доверия) в отношении степени истинности. Например, в вероятностном мире вампуса агент может доверять утверждению о том, что вампус находится в квадрате [1,3], с вероятностью 0,75 и с вероятностью 0,25 доверять утверждению, что он находится в квадрате [2,3] (хотя вампус, безусловно, находится в одном конкретном квадрате).

8.2. Синтаксис и семантика логики первого порядка

Этот раздел мы начнем с более точного определения способа, посредством которого возможные миры логики первого порядка отражают ее онтологический вклад в познание объектов и отношений. Затем будут представлены различные элементы языка этой логики, с объяснением по ходу дела их семантики. Основной акцент будет сделан на том, как язык способствует сжатым представлениям и как его семантика приводит к непротиворечивым процедурам рассуждения.

8.2.1. Модели для логики первого порядка

В главе 7 было сказано, что моделями в любом логическом языке служат формальные структуры, представляющие возможные рассматриваемые миры. Каждая модель связывает словарь логических высказываний с элементами возможного мира, что позволяет определить истинность любого высказывания. Следовательно, модели для логики высказываний связывают пропозициональные символы с предопределенными значениями истинности.

Модели для логики первого порядка гораздо интереснее. Прежде всего, в них имеются объекты! ▶**Проблемной областью** модели является множество объектов или ▶**элементов проблемной области**, как их иногда называют, которые эта область содержит. Проблемная область обязательно должна быть *не пустой* — каждый возможный мир должен содержать по меньшей мере один объект. (В упражнении 8.8 обсуждается возможность допущения пустых миров.) С математической точки зрения не имеет значения, *что* эти объекты собой представляют (все, что имеет значение, — это лишь *сколько* их в каждой конкретной модели), но исходя из педагогических соображений в дальнейшем обсуждении мы будем использовать конкретный пример. На рис. 8.2 представлена модель с пятью объектами: Ричард Львиное Сердце, король Англии, который правил с 1189 по 1199 годы; его младший брат, злой король Джон, который правил с 1199 по 1215 год; левые ноги Ричарда и Джона; а также корона.

Отношения между объектами в этой модели могут быть определены различными способами. На данном рисунке показано, что Ричард и Джон — братья. Выражаясь формально, отношение — это просто множество ▶**кортежей** объектов, связанных друг с другом. (Кортеж — это коллекция объектов, упорядоченных в установленном порядке и записанных в угловых скобках ⟨ ⟩, окружающих эти объекты.) Таким образом, отношение братства в этой модели представляет собой следующее множество:

$$\{\langle\text{Ричард Львиное Сердце, король Джон}\rangle, \langle\text{король Джон, Ричард Львиное Сердце}\rangle\}. \quad (8.1)$$

(Здесь эти объекты были указаны на естественном языке, но читатель при желании может мысленно вставить вместо имен этих королей их портреты.) Корона

находится на голове короля Джона, поэтому отношение "быть на голове" содержит только один кортеж, ⟨корона, король Джон⟩. Отношения "быть братом" и "быть на голове" являются бинарными, т.е. они устанавливают связь между парами объектов. Эта модель также содержит унарные отношения, или свойства: свойство "быть человеком" является истинным и для Ричарда, и для Джона; свойство "быть королем" истинно только для Джона (предположительно потому, что к этому моменту Ричард уже был мертв); а свойство "быть короной" истинно только для короны.

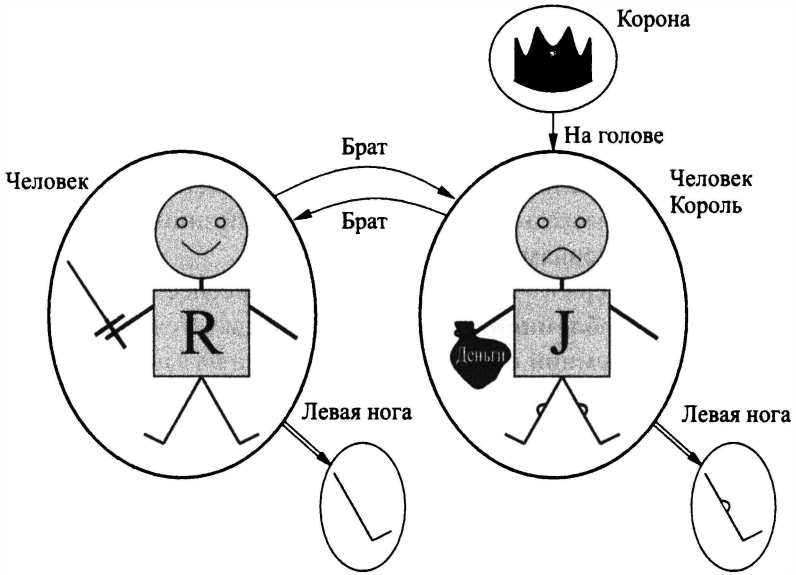


Рис. 8.2. Модель, состоящая из пяти объектов: двух бинарных отношений ("быть братом" и "быть на голове"), трех унарных отношений ("быть человеком", "быть короной", "быть королем") и одной унарной функции ("быть левой ногой")

Некоторые виды отношений удобнее рассматривать как функции, в том смысле, что указанным способом данный объект должен быть связан только с одним объектом. Например, каждый человек имеет только одну левую ногу, поэтому в данной модели имеется унарная функция "быть левой ногой", которая включает следующие отображения:

$$\langle \text{Ричард Львиное Сердце} \rangle \rightarrow \text{левая нога Ричарда},$$
$$\langle \text{король Джон} \rangle \rightarrow \text{левая нога Джона}. \qquad (8.2)$$

Строго говоря, модели в логике первого порядка требуют ►**полностью определенных функций**, т.е. в них должно быть предусмотрено значение для каждого входного кортежа. Таким образом, левую ногу должна иметь и корона, и, безусловно, каждая из левых ног. Имеется некоторое формальное решение этой неприятной

проблемы, касающейся дополнительного "невидимого" объекта, представляющего левую ногу у всего, что не имеет левой ноги, включая и саму левую ногу. К счастью, пока никто не делает утверждений о левых ногах объектов, не имеющих левой ноги, применять эти формальные решения необязательно.

До сих пор мы обсуждали лишь элементы, заполняющие модели для логики первого порядка. Другой существенной частью таких моделей также является связь между этими элементами и лексикой логических высказываний, которая будет обсуждаться в следующем разделе.

8.2.2. Символы и интерпретации

Вернемся к синтаксису логики первого порядка. Нетерпеливый читатель может найти полное описание формальной грамматики логики первого порядка на рис. 8.3.

Основными синтаксическими элементами логики первого порядка являются символы, которые обозначают объекты, отношения и функции. Поэтому сами символы подразделяются на три типа: ▶**символы констант**, которые обозначают объекты; ▶**символы предикатов**, которые обозначают отношения, и ▶**символы функций**, обозначающие функции. Примем соглашение, что имена этих символов будут начинаться с прописных букв. Например, могут использоваться символы констант *Richard* и *John*; символы предикатов *Brother*, *OnHead*, *Person*, *King* и *Crown* и символ функции *LeftLeg*. Как и в случае пропозициональных символов, выбор имен полностью предоставляется пользователю. Каждый символ предиката и функции характеризуется ▶**арностью**, определяющей количество его формальных параметров.

Каждая модель должна обеспечивать всю информацию, необходимую для определения, каким является любое заданное предложение: истинным или ложным. Поэтому в дополнение к объектам, отношениям и функциям каждая модель включает в себя ▶**интерпретацию**, точно определяющую, на какие именно объекты, отношения и функции ссылается каждый из символов констант, предикатов и функций. Для рассматриваемого примера одна из возможных интерпретаций — которую будет логично назвать ▶**предполагаемой интерпретацией** — состоит в следующем.

- Символ *Richard* обозначает Ричарда Львиное Сердце, а символ *John* — злого короля Джона.
- Символ *Brother* обозначает отношение родства между братьями, т.е. множество кортежей объектов, приведенное в уравнении 8.1; символ *OnHead* обозначает отношение "быть на голове", установленное между короной и королем Джоном; символы *Person*, *King* и *Crown* относятся к унарным отношениям, определяющим людей, королей и короны.
- Символ *LeftLeg* относится к функции "быть левой ногой", определенной в уравнении 8.2.

$$
\begin{aligned}
\textit{Высказывание} &\rightarrow \textit{АтомарноеВысказывание} \mid \textit{СложноеВысказывание} \\
\textit{АтомарноеВысказывание} &\rightarrow \textit{Предикат} \mid \textit{Предикат}(\textit{Терм},\ldots) \mid \textit{Терм} = \textit{Терм} \\
\textit{СложноеВысказывание} &\rightarrow (\textit{Высказывание}) \\
&\mid \neg \textit{Высказывание} \\
&\mid \textit{Высказывание} \wedge \textit{Высказывание} \\
&\mid \textit{Высказывание} \vee \textit{Высказывание} \\
&\mid \textit{Высказывание} \Rightarrow \textit{Высказывание} \\
&\mid \textit{Высказывание} \Leftrightarrow \textit{Высказывание} \\
&\mid \textit{Квантор Переменная},\ldots \textit{Высказывание} \\
\\
\textit{Терм} &\rightarrow \textit{Функция}(\textit{Терм},\ldots) \\
&\mid \textit{Константа} \\
&\mid \textit{Переменная} \\
\\
\textit{Квантор} &\rightarrow \forall \mid \exists \\
\textit{Константа} &\rightarrow A \mid X_1 \mid \textit{John} \mid \ldots \\
\textit{Переменная} &\rightarrow a \mid x \mid s \mid \ldots \\
\textit{Предикат} &\rightarrow \textit{True} \mid \textit{False} \mid \textit{After} \mid \textit{Loves} \mid \textit{Raining} \mid \ldots \\
\textit{Функция} &\rightarrow \textit{Mother} \mid \textit{LeftLeg} \mid \ldots \\
\text{Порядок выполнения операций} &: \neg, =, \wedge, \vee, \Rightarrow, \Leftrightarrow
\end{aligned}
$$

Рис. 8.3 Синтаксис логики первого порядка с оператором равенства, заданный в форме Бэкуса–Наура (см. раздел 1 в приложении Б). Приоритеты операторов указаны слева направо от самого высокого до самого низкого. Приоритет кванторов таков, что квантор содержит все, что находится справа от него

Безусловно, также может существовать много других возможных интерпретаций. Например, в одной интерпретации символ *Richard* отображается на корону, а символ *John* — на левую ногу короля Джона. В данной модели имеется пять объектов, поэтому существует 25 возможных интерпретаций только для символов констант *Richard* и *John*. Обратите внимание на то, что не все объекты имеют имя, например в нашей предполагаемой интерпретации не предусмотрены имена для короны или ноги. Также допустимо, чтобы один объект имел несколько имен — возможна интерпретация, в которой и символ *Richard*, и символ *John* определяют корону.[2] Если читатель полагает, что такая возможность приводит к путанице,

[2] Ниже, в разделе 8.2.8, рассматривается семантика, в которой каждый объект должен иметь точно одно имя.

напомним, что в логике высказываний вполне допустимо иметь модель, в которой высказывания *Cloudy* (пасмурно) и *Sunny* (солнечно) одновременно являются истинными; задача исключения из рассмотрения моделей, несовместимых с нашими знаниями, возлагается на базу знаний.

Итак, можно обобщить, что модель в логике первого порядка состоит из множества объектов и некоторой интерпретации, отображающей символы констант на объекты, символы функций на функции над этими объектами и символы предикатов на отношения между объектами. Как и в случае логики высказываний, логическое следствие, допустимость и так далее определяются в терминах *всех возможных моделей*. Получить представление о том, как выглядит множество всех возможных моделей, можно, взглянув на рис. 8.4, наглядно демонстрирующий, что модели могут различаться как количеством объектов, которые они содержат — от одного до бесконечности, — так и способом, которым символы констант отображаются на объекты.

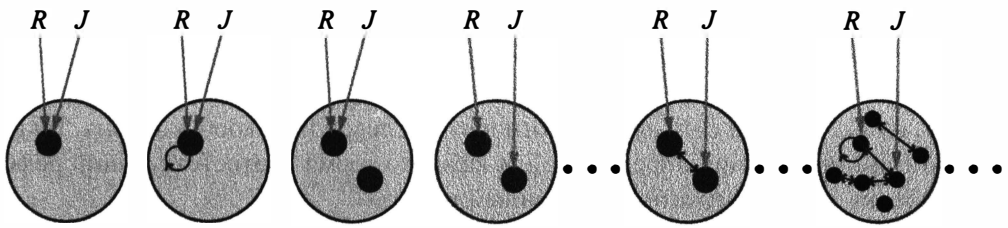


Рис. 8.4. Некоторые члены множества всех моделей для языка с двумя символами констант, *R* и *J*, и одного символа бинарного отношения. Интерпретация каждого символа константы показана серой стрелкой. В пределах каждой модели связанные объекты соединены тонкими темными стрелками

Поскольку число моделей первого порядка не ограничено, мы не можем проверить выполнение простым перебором их всех (как это делалось для логики высказываний). Даже если количество объектов ограничено, количество комбинаций может быть очень большим (см. упр. 8.6). Для примера на рис. 8.4 существует 137 506 194 466 моделей с шестью или менее объектами.

8.2.3. Термы

► **Терм** — это логическое выражение, относящееся к некоторому объекту. Символы констант также являются термами, однако не всегда удобно иметь отдельный символ для именования каждого объекта. Например, в русском языке можно воспользоваться обозначением "левая нога короля Джона", а не присваивать ноге короля собственное имя. Именно для этого и нужны функциональные символы:

вместо использования специального символа константы можно просто записать *LeftLeg*(*John*).[3]

В общем случае сложный терм формируется с помощью символа функции, за которым следует заключенный в круглые скобки список формальных параметров для данного символа функции. Важно помнить, что любой сложный терм — это просто некоторый вид имени, выраженный в сложной форме. Это не "вызов процедуры", которая "возвращает значение". Не существует такой процедуры *LeftLeg*, которая принимает на входе какого-то человека и возвращает его левую ногу. Мы можем рассуждать о левых ногах (например, сформулировать общее правило, что каждый человек имеет таковую, и на основании этого сделать вывод, что Джон также должен иметь ее), даже не представив определение символа *LeftLeg*. Это нечто, что нельзя выполнить с помощью процедур в языках программирования.

Формальное определение семантики термов является несложным. Рассмотрим терм $f(t_1, \ldots, t_n)$. Символ функции f относится к некоторой функции в модели (назовем ее F); термы с обозначением формальных параметров относятся к объектам данной проблемной области (назовем их $d_1, \ldots, d_n$); а сам терм в целом относится к объекту, представляющему собой значение функции F, применяемой к объектам $d_1, \ldots, d_n$. Например, предположим, что функциональный символ *LeftLeg* относится к функции, показанной в уравнении 8.2, а символ *John* относится к королю Джону; в таком случае *LeftLeg*(*John*) относится к левой ноге короля Джона. Таким образом, интерпретация определяет для каждого терма соответствующий референт (объект, к которому относится данный терм).

8.2.4. Атомарные высказывания

Теперь, когда у нас есть определение терма для обозначения объектов и символа предиката для обозначения отношений, можно объединить их для создания ►**атомарных высказываний**, констатирующих факты. Атомарное высказывание (или ►**атом** для краткости) формируется из символа предиката, за которым следует (но необязательно) заключенный в круглые скобки список термов, как показано ниже.

$$Brother(Richard, John)$$

В соответствии с предполагаемой интерпретацией, приведенной выше, это атомарное высказывание констатирует тот факт, что Ричард Львиное Сердце — брат

[3] **λ-выражения** представляют собой удобные обозначения, позволяющие формировать новые функциональные символы "динамически". Например, функция, возводящая в квадрат свой аргумент, может быть записана как $(\lambda x : x \times x)$ и применяться к формальным параметрам точно так же, как и любой другой символ функции. Также λ-выражение, может быть преобразовано путем "вставки" в него соответствующих формальных параметров для получения эквивалентного высказывания.

короля Джона.[4] В качестве фактических параметров атомарные высказывания могут включать и сложные термы. Поэтому в высказывании

$$Married(Father(Richard), Mother(John))$$

утверждается, что отец Ричарда Львиное Сердце был женат на матери короля Джона (опять-таки, при использовании подходящей интерпретации).[5]

➨ *Любое атомарное высказывание является истинным в данной конкретной модели при данной конкретной интерпретации, если отношение, на которое ссылается его символ предиката, соблюдается среди объектов, на которые ссылаются его параметры.*

8.2.5. Сложные высказывания

Для формирования более сложных высказываний могут использоваться **логические связки** — с теми же синтаксисом и семантикой, что и в логике высказываний. Ниже приведены четыре высказывания, которые являются истинными в модели, представленной на рис. 8.2, при использовании принятой предполагаемой интерпретации.

$$\neg Brother(LeftLeg(Richard), John)$$
$$Brother(Richard, John) \wedge Brother(John, Richard)$$
$$King(Richard) \vee King(John)$$
$$\neg King(Richard) \Rightarrow King(John)$$

8.2.6. Кванторы

После определения логики, допускающей использование объектов, вполне естественным будет стремление получить возможность выражать свойства целых коллекций объектов, вместо того чтобы перебирать эти объекты по именам. Сделать это позволяют ▶ **кванторы**. Логика первого порядка включает два стандартных квантора, называемых кванторами *всеобщности* и *существования*.

Квантор всеобщности ($\forall$)

Вспомните, с какими трудностями мы сталкивались в главе 7 при попытках выразить общие правила в логике высказываний. В логике первого порядка выразить такие правила, как "В квадратах, соседних с тем, в котором находится вампус, ощущается неприятный запах" и "Все короли являются людьми" проще простого. Первое из этих правил будет рассматриваться в разделе 8.3, а второе правило,

---

[4] Как правило, мы будем придерживаться такого соглашения об упорядочении параметров, что $P(x, y)$ интерпретируется как "x представляет собой P от y".

[5] Эта онтология распознает только одного отца и одну мать для каждого человека. Более сложная онтология может распознавать биологическую мать, крестную мать, приемную мать и т.д.

"Все короли являются людьми", записывается в логике первого порядка следующим образом:

$$\forall x \ \ King(x) \Rightarrow Person(x).$$

▶**Квантор всеобщности** $\forall$ обычно читается как "для всех..." (Запомните, что перевернутая буква "A" здесь обозначает "all" — *все*.) Таким образом, в этом высказывании утверждается следующее: "Для всех x, если x — король, то x — человек". Символ x называется ▶**переменной**. В соответствии с общепринятым соглашением для обозначения переменных используются строчные буквы. Переменная сама по себе является термом и как таковая может также служить параметром функции, например $LeftLeg(x)$. Терм без переменных называется ▶**базовым термом**.

Интуитивно понятно, что высказывание $\forall x \ P$, где P — любое логическое выражение, утверждает, что P является истинным для каждого объекта x. Точнее, высказывание $\forall x \ P$ истинно в данной модели при данной интерпретации, если выражение P истинно при всех возможных ▶**расширенных интерпретациях**, сформированных из данной интерпретации, где каждая расширенная интерпретация задает элемент проблемной области, на которую ссылается объект x.

Это определение может показаться сложным, но фактически оно представляет собой лишь формальное определение интуитивного смысла применения квантора всеобщности. Рассмотрим модель, показанную на рис. 8.2, и предполагаемую интерпретацию, которая ее сопровождает. Эту интерпретацию можно расширить следующими пятью способами.

$x \rightarrow$ Ричард Львиное Сердце
$x \rightarrow$ король Джон
$x \rightarrow$ левая нога Ричарда
$x \rightarrow$ левая нога Джона
$x \rightarrow$ корона

Высказывание с квантором всеобщности $\forall x \ King(x) \Rightarrow Person(x)$ является истинным в исходной модели, если высказывание $King(x) \Rightarrow Person(x)$ истинно в каждой из пяти расширенных интерпретаций. Это означает, что данное высказывание с квантором всеобщности эквивалентно утверждению об истинности следующих пяти высказываний.

Ричард Львиное Сердце — король $\Rightarrow$ Ричард Львиное Сердце — человек
король Джон — король $\Rightarrow$ король Джон — человек
левая нога Ричарда — король $\Rightarrow$ левая нога Ричарда — человек
левая нога Джона — король $\Rightarrow$ левая нога Джона — человек
корона — король $\Rightarrow$ корона — человек

Давайте внимательно рассмотрим это множество утверждений. Поскольку в нашей модели единственным королем является король Джон, то во втором

высказывании утверждается, что он — человек, как и следовало ожидать. А что можно сказать о других четырех высказываниях, в частности о тех, в которых приведены утверждения о ногах и коронах? Являются ли они частью смысла утверждения "Все короли являются людьми"? Действительно, остальные четыре утверждения истинны в данной модели, но не позволяют судить о том, можно ли считать людьми ноги, короны или даже Ричарда. Причина в том, что ни один из этих объектов не является королем. Рассматривая истинностную таблицу для связки $\Rightarrow$ (см. рис. 7.8), можно убедиться в том, что импликация истинна, даже если ее предпосылка ложна, *независимо* от того, является ли истинным заключение. Таким образом, утверждая истинность высказывания с квантором всеобщности, что эквивалентно утверждению об истинности целого списка отдельных импликаций, мы в конечном итоге утверждаем об истинности правила, выраженного в виде этого высказывания, но только для тех объектов, для которых предпосылка является истинной, и вообще ничего не говорим о тех объектах, для которых предпосылка ложна. Поэтому элементы истинностной таблицы, относящиеся к связке $\Rightarrow$, оказались идеальным средством формулирования общих правил с кванторами всеобщности.

Распространенная ошибка, которую часто допускают даже прилежные читатели, несколько раз прочитавшие предыдущий абзац, состоит в использовании конъюнкции вместо импликации. Тогда высказывание

$$\forall x \ \ King(x) \wedge Person(x)$$

становится эквивалентным таким утверждениям:

Ричард Львиное Сердце — король $\wedge$ Ричард Львиное Сердце — человек,
король Джон — король $\wedge$ король Джон — человек,
левая нога Ричарда — король $\wedge$ левая нога Ричарда — человек

и т.д. Очевидно, что такой ряд утверждений не передает желаемый смысл.

Квантор существования ($\exists$)

Квантор всеобщности позволяет формировать утверждения о каждом объекте. Аналогичным образом можно создать утверждение о *некотором* объекте без его именования, использовав ▶ **квантор существования**. Например, чтобы выразить мысль, что на голову короля Джона возложена корона, можно записать следующее:

$$\exists x \ \ Crown(x) \wedge OnHead(x, John).$$

Выражение $\exists x$ читается как "Существует x, такой, что..." или "Для некоторого x..."

Интуитивно понятно, что в высказывании $\exists x \ \ P$ утверждается, что P истинно по меньшей мере для одного объекта x. Если говорить точнее, то высказывание $\exists x \ \ P$ истинно в данной модели, если выражение P истинно *по меньшей мере*

в одной расширенной интерпретации, в которой переменной x присваивается один из элементов проблемной области. Следовательно, по меньшей мере одно из приведенных ниже утверждений должно быть истинным.

Ричард Львиное Сердце — корона $\wedge$ Ричард Львиное Сердце находится на голове Джона
король Джон — корона $\wedge$ король Джон находится на голове Джона
левая нога Ричарда — корона $\wedge$ левая нога Ричарда находится на голове Джона
левая нога Джона — корона $\wedge$ левая нога Джона находится на голове Джона
корона — корона $\wedge$ корона находится на голове Джона

В нашей модели истинно пятое утверждение, поэтому в ней является истинным и само первоначальное утверждение с квантором существования. Обратите внимание на то, что в соответствии с приведенным выше определением квантора существования это высказывание будет истинным и в такой модели, в которой на голову короля Джона возложены две короны. Такая ситуация является полностью совместимой с первоначальным высказыванием: "На голове короля Джона есть корона".[6]

Аналогично тому, что логическую связку $\Rightarrow$ можно считать наиболее подходящей для использования с квантором $\forall$, логическую связку $\wedge$ следует считать наиболее подходящей для использования с квантором $\exists$. В примере из предыдущего раздела использование $\wedge$ в качестве основной связки в сочетании с квантором $\forall$ приводило к формированию слишком сильного утверждения. В свою очередь, использование связки $\Rightarrow$ в сочетании с квантором $\exists$ обычно приводит к формированию слишком слабых утверждений. Рассмотрим следующее высказывание:

$$\exists x \; Crown(x) \Rightarrow OnHead(x, John).$$

На первый взгляд может показаться, что оно вполне успешно передает мысль о том, что на голове короля Джона есть корона. Так, применив соответствующее определение семантики, можно убедиться в том, что данное высказывание декларирует истинность по меньшей мере одного из следующих утверждений:

Ричард Львиное Сердце — корона $\Rightarrow$ Ричард Львиное Сердце находится на голове Джона,
король Джон — корона $\Rightarrow$ король Джон находится на голове Джона,
левая нога Ричарда — корона $\Rightarrow$ левая нога Ричарда находится на голове Джона

и т.д. Итак, импликация истинна, если и предпосылка, и заключение являются истинными *или если ложна ее предпосылка.* Поэтому, если Ричард Львиное

[6] Применяется также определенный вариант квантора существования, обычно записываемый как $\exists^1$ или $\exists!$, означающий "Существует только один..." Однако тот же смысл можно выразить и с использованием утверждений, содержащих знак равенства.

Сердце — не корона, то первое утверждение истинно и высказывание с квантором существования выполняется. Таким образом, высказывание в форме импликации с квантором существования истинно в любой модели, содержащей объект, для которого предпосылка импликации является ложной, поэтому подобные высказывания фактически не несут почти никакой информации.

Вложенные кванторы

Часто необходимо сформировать более сложные высказывания с использованием нескольких кванторов. Самый простой случай — когда кванторы относятся к одному и тому же типу. Например, утверждение "Братья — это люди, связанные братскими родственными узами", может быть записано следующим образом:

$$\forall x \ \forall y \ \ Brother(x, y) \Rightarrow Sibling(x, y).$$

Последовательно применяемые кванторы могут быть записаны как один квантор с несколькими переменными. Например, чтобы выразить мысль о том, что родственные отношения между людьми, связанными братскими узами, являются симметричными, можно составить следующее высказывание:

$$\forall x, y \ \ Sibling(x, y) \Leftrightarrow Sibling(y, x).$$

В других случаях возникает необходимость в использовании сочетания разных кванторов. Например, строка из песни "Everybody loves somebody" (*Каждый кого-то любит*) означает, что для каждого человека существует кто-то, кого этот человек любит:

$$\forall x \exists y \ \ Loves(x, y).$$

С другой стороны, чтобы сформулировать утверждение "There is someone who is loved by everyone" (*Есть некто, кого любят все*), можно записать следующее:

$$\exists y \ \forall x \ \ Loves(x, y).$$

Следовательно, порядок расположения кванторов очень важен. Он становится очевиднее после вставки круглых скобок. В высказывании $\forall x \ (\exists y \ \ Loves(x, y))$ утверждается, что *каждый* имеет определенное свойство, а именно — то свойство, что он кого-то любит. С другой стороны, в высказывании $\exists y \ (\forall x \ \ Loves(x, y))$ утверждается, что *кто-то* в мире имеет определенное свойство, а именно — свойство быть любимым всеми.

Может возникнуть некоторая путаница, когда два квантора используются с одним и тем же именем переменной. Рассмотрим следующее высказывание:

$$\forall x \ (Crown(x) \vee (\exists x \ \ Brother(Richard, x))).$$

Здесь к переменной x в атомарном высказывании $Brother(Richard, x)$ применяется квантор *существования*. Общее правило состоит в том, что переменная принадлежит к самому внутреннему квантору, в котором она упоминается, и это

означает, что такая переменная не может стать субъектом действия любого другого квантора. Еще один способ анализа приведенного выше высказывания состоит в следующем: $\exists x\ Brother(Richard, x)$ — это высказывание о Ричарде (о том, что у него есть брат), а не о переменной x, поэтому размещение квантора $\forall x$ за пределами данного высказывания не оказывает на него никакого действия и оно может быть равным образом записано как $\exists z\ Brother(Richard, z)$. Поскольку такая ситуация может стать источником путаницы, в выражениях с вложенными кванторами мы всегда будем использовать разные переменные.

Связь между кванторами $\forall$ и $\exists$

Два квантора, $\forall$ и $\exists$, в действительности тесно связаны друг с другом через отрицание. Утверждение о том, что никто не любит пастернак (*parsnip*), равносильно утверждению о том, что не существует никого, кто бы его любил, и наоборот:

$$\forall x\ \neg Likes(x, Parsnip) \text{ эквивалентно } \neg\exists x\ Likes(x, Parsnip).$$

Можем пойти и на один шаг дальше, сформировав более сложную конструкцию: выражение "Все любят мороженое (*IceCream*)" означает, что нет никого, кто не любил бы мороженое:

$$\forall x\ Likes(x, IceCream) \text{ эквивалентно } \neg\exists x\ \neg Likes(x, IceCream).$$

Поскольку в универсуме объектов квантор $\forall$ фактически определяет конъюнкцию, а квантор $\exists$ — дизъюнкцию, нет ничего удивительного в том, что они подчиняются правилам де Моргана. Правила де Моргана для высказываний с кванторами и без кванторов выглядят следующим образом.

$$\begin{array}{lcllcl}
\neg\exists x\ P & \equiv & \forall x\ \neg P & \neg(P \vee Q) & \equiv & \neg P \wedge \neg Q \\
\neg\forall x\ P & \equiv & \exists x\ \neg P & \neg(P \wedge Q) & \equiv & \neg P \vee \neg Q \\
\forall x\ P & \equiv & \neg\exists x\ \neg P & P \wedge Q & \equiv & \neg(\neg P \vee \neg Q) \\
\exists x\ P & \equiv & \neg\forall x\ \neg P & P \vee Q & \equiv & \neg(\neg P \wedge \neg Q)
\end{array}$$

Таким образом, в действительности нет особой необходимости иметь одновременно два квантора, $\forall$ и $\exists$, так же как фактически не нужны обе связки, $\wedge$ и $\vee$. Тем не менее удобство чтения важнее, чем экономия выразительных средств, поэтому мы будем пользоваться обоими этими кванторами.

8.2.7. Равенство

Логика первого порядка включает еще один способ составления атомарных высказываний, отличный от использования символов предикатов и термов, как было описано выше. Для указания на то, что два терма ссылаются на один и тот же объект, может использоваться ▶**символ равенства**. Например, утверждение

$$Father(John) = Henry$$

говорит о том, что объект, на который ссылается высказывание *Father* (*John*), и объект, указанный под именем *Henry*, — это в действительности один и тот же объект. Поскольку в любой интерпретации за каждым термом закрепляется референт, т.е. объект, на который он ссылается, определение истинности любого высказывания с символом равенства сводится к проверке того, представляют ли собой референты двух термов, соединенных символом равенства, один и тот же объект.

Символ равенства может использоваться для констатации фактов, касающихся данной конкретной функции, как было только что сделано применительно к функциональному символу *Father*. Он может также применяться с отрицанием как указание на то, что два терма не представляют собой один и тот же объект. Чтобы выразить мысль о том, что Ричард имеет по меньшей мере двух братьев, можно записать следующее:

$$\exists x, y \;\; Brother(x, Richard) \wedge Brother(y, Richard) \wedge \neg(x = y).$$

Тогда как высказывание

$$\exists x, y \;\; Brother(x, Richard) \wedge Brother(y, Richard)$$

не имеет подобного, нужного нам смысла. В частности, оно истинно в модели, приведенной на рис. 8.2, где у Ричарда имеется только один брат. Чтобы убедиться в этом, рассмотрим расширенную интерпретацию, в которой обеим переменным, x и y, присваивается один и тот же объект, король Джон, — в результате добавления выражения $\neg(x = y)$ такие модели становятся *недействительными*. В качестве сокращения для записи $\neg(x = y)$ иногда используется обозначение $x \neq y$.

8.2.8. Семантика базы данных

Продолжим пользоваться примером из предыдущего раздела и предположим, что на самом деле у Ричарда есть два брата, Джон и Джеффри.[7] Тогда можно было бы записать следующее:

$$Brother(John, Richard) \wedge Brother(Geoffrey, Richard). \qquad (8.3)$$

Однако эта запись не полностью отражает состояние дел. Во-первых, это утверждение является истинным в модели, где Ричард имеет только одного брата — нужно добавить еще и Джона, поскольку $John \neq Geoffrey$. Во-вторых, это высказывание не исключает те модели, в которых Ричард имеет еще множество братьев, помимо Джона и Джеффри.

Следовательно, правильный перевод утверждения "Братья Ричарда — это Джон и Джеффри" на язык логики будет выглядеть следующим образом:

$$Brother(John, Richard) \wedge Brother(Geoffrey, Richard) \wedge John \neq Geoffrey \wedge$$
$$\forall x \;\; Brother(x, Richard) \Rightarrow (x = John \vee x = Geoffrey).$$

[7] На самом деле братьев было четверо, двух остальных звали Уильям и Генри.

Это логическое высказывание кажется гораздо более громоздким, чем соответствующее предложение на русском языке. Однако нужно принять во внимание, что, если перевод с русского языка на язык логики будет неправильным, система логического мышления будет делать ошибки. А нельзя ли разработать семантику, позволяющую записывать те же логические высказывания в более простом виде?

Одно предложение, которое очень популярно в системах баз данных, работает следующим образом. Во-первых, введем требование, что каждый символ константы должен ссылаться на отдельный, отличный от других объект — ▶**допущение об уникальности имен**. Во-вторых, будем предполагать, что атомарные высказывания, об истинности которых нам ничего не известно, в действительности являются ложными — ▶**допущение о замкнутости мира**. И наконец, в-третьих, введем правило ▶**замыкания проблемной области**, означающее, что каждая модель содержит не больше элементов проблемной области, чем имеется элементов, именованных символами констант.

Исходя из предложенной выше семантики, уравнение (8.3) действительно утверждает, что у Ричарда ровно два брата, Джон и Джеффри. Назовем эти допущения ▶**семантикой базы данных**, что позволит отличать ее от стандартной семантики логики первого порядка. Эта же семантика базы данных будет использоваться и при обсуждении логического программирования в разделе 9.4.4.

Будет поучительно, следуя семантике базы данных, рассмотреть множество всех возможных моделей для того случая, который был представлен выше, на рис. 8.4. На рис. 8.5 показаны некоторые из моделей в диапазоне от модели без кортежей, удовлетворяющих отношению, до модели со всеми возможными кортежами, удовлетворяющих отношению. При наличии двух объектов возможны четыре двухэлементных кортежа, поэтому существует $2^4 = 16$ различных подмножеств кортежей, удовлетворяющих отношению. Таким образом, всего имеется 16 возможных моделей — несравненно меньшее количество в сравнении с бесконечным их числом для стандартной семантики логики первого порядка. С другой стороны, использование семантики базы данных требует определенных знаний о том, что содержит мир.

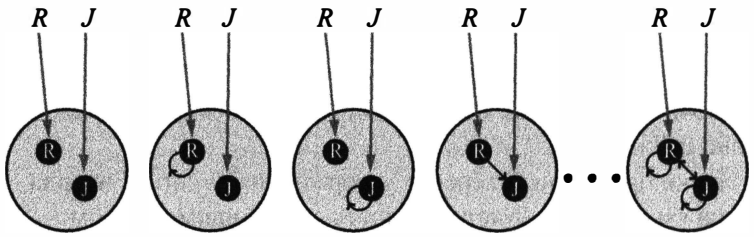


Рис. 8.5. Некоторые члены множества всех моделей для языка с двумя символами констант, *R* и *J*, и одним бинарным символом отношения при использовании семантики базы данных. Интерпретация символов констант фиксирована, и для каждого из них имеется отдельный объект

Этот пример подчеркивает один важный момент: для логики не существует никакой "правильной" семантики. Полезность любой предлагаемой семантики зависит от того, насколько кратко и интуитивно понятно она позволяет выразить тот тип знаний, который требуется описать, и насколько просто и естественно будет разработать для нее соответствующие правила логического вывода. Семантика базы данных наиболее полезна, когда мы уверены в идентичности всех объектов, описываемых в базе знаний, и когда у нас в руках имеются все необходимые факты. В иных ситуациях эта семантика может оказаться неудобной. В остальной части этой главы предполагается использование стандартной семантики, а те случаи, когда это приводит к громоздким выражениям, будут отмечаться особо.

8.3. Использование логики первого порядка

Теперь, после того как выразительный логический язык был определен, можно приступить к изучению способов его использования. В этом разделе будут приведены примеры высказываний в некоторых простых ▶**проблемных областях**. Когда речь идет о представлении знаний, проблемная область — это просто часть мира, в отношении которой необходимо выразить некоторые знания.

Начнем с краткого описания интерфейса TELL/ASK для базы знаний в логике первого порядка. Затем рассмотрим несколько проблемных областей — семейные отношения, числа, множества и списки, а также мир вампуса. В разделе 8.4.2 приведен более развернутый пример (электронные схемы), а в главе 10 рассматривается все, что касается данной области науки.

8.3.1. Утверждения и запросы в логике первого порядка

Высказывания вводятся в базу знаний с помощью операции TELL, точно так же, как и в случае логики высказываний. Такие высказывания называются ▶**утверждениями**. Например, можно ввести в базу знаний утверждения, что Джон — король, Ричард — человек и что все короли — люди:

$$\text{TELL}(KB, King(John)),$$
$$\text{TELL}(KB, Person(Richard)),$$
$$\text{TELL}(KB, \forall x \; King(x) \Rightarrow Person(x)).$$

Используя операцию ASK, можно задавать вопросы о содержимом базы знаний. Например, на вопрос

$$\text{ASK}(KB, King(John))$$

будет возвращено значение *true*. Вопросы, задаваемые с помощью операции ASK, называются **запросами** или **целями**. Вообще говоря, на любой запрос, который логически следует из базы знаний, должен быть получен утвердительный ответ. Например, если в ней содержатся три утверждения, приведенные выше, то запрос

$$\text{ASK}(KB, Person(John))$$

также должен вернуть значение *true*. Кроме того, можно выдавать запросы с кванторами, такие как

$$ASK(KB, \exists x \ \ Person(x)).$$

Ответом будет *true*, но этот ответ, вероятно, будет не столь полезен, как хотелось бы. Это примерно то же самое, что в ответ на вопрос "Вы можете сказать, который час?" получить ответ "Да". Если требуется узнать, какое значение x делает некоторое высказывание истинным, то необходимо использовать другую функцию, которую мы назовем ASKVARS:

$$\text{ASKVARS}(KB, Person(x))$$

и которая возвратит поток ответов. В данном случае будет выдано два ответа: $\{x/John\}$ и $\{x/ Richard\}$. Подобный ответ называют ▶ **подстановкой** или ▶ **списком связывания**. Функция ASKVARS обычно резервируется для баз знаний, состоящих исключительно из хорновских выражений, поскольку в таких базах знаний любой способ приведения запроса к значению *true* предполагает связывание переменных с конкретными значениями. Но это замечание вовсе не относится к логике первого порядка: в базе данных *KB*, в которую было помещено единственное высказывание $King(John) \vee King(Richard)$, нет ни одной связки с x, делающей запрос $\exists x \ \ King(x)$ истинным, хотя этот запрос действительно возвращает значение *true*.

8.3.2. Проблемная область родства

В качестве первого примера рассмотрим проблемную область семейных отношений, или родства. Эта проблемная область включает такие факты, как "Элизабет — мать Чарльза" и "Чарльз — отец Уильяма", и правила наподобие "Чья-то бабушка — это мать одного из родителей этого человека".

Очевидно, что объектами в этой проблемной области являются люди. Прежде всего, в ней будут применяться два унарных предиката, *Male* (Мужчина) и *Female* (Женщина). Отношения родства (связи между родителями и детьми, братьями и сестрами, мужем и женой и т.д.) будут представлены с помощью бинарных предикатов: *Parent* (родитель), *Sibling* (брат или сестра), *Brother* (брат), *Sister* (сестра), *Child* (ребенок), *Daughter* (дочь), *Son* (сын), *Spouse* (супруг или супруга), *Wife* (жена), *Husband* (муж), *Grandparent* (дедушка или бабушка), *Grandchild* (внук или внучка), *Cousin* (двоюродный брат или двоюродная сестра), *Aunt* (тетя) и *Uncle* (дядя). В качестве предикатов *Mother* (мать) и *Father* (отец) мы будем использовать функции, поскольку каждый человек имеет по одному из этих объектов в соответствии с законами природы, хотя при необходимости можно было бы ввести дополнительные функции для приемных матерей, суррогатных матерей и т.д.

Рассмотрим каждую из этих функций и предикатов, записывая все, что мы знаем о них, в терминах других символов. Например, мать — это родитель женского пола:

$$\forall\, m, c \;\; Mother(c) = m \;\Leftrightarrow\; Female(m) \wedge Parent(m, c).$$

Муж — это супруг мужского пола:

$$\forall\, w, h \;\; Husband(h, w) \;\Leftrightarrow\; Male(h) \wedge Spouse(h, w).$$

Отношения между родителями и детьми являются взаимно противоположными:

$$\forall\, p, c \;\; Parent(p, c) \;\Leftrightarrow\; Child(c, p).$$

Дедушка или бабушка — это родитель родителя:

$$\forall\, g, c \;\; Grandparent(g, c) \;\Leftrightarrow\; \exists\, p \;\; Parent(g, p) \wedge Parent(p, c).$$

Брат или сестра — это еще один ребенок тех же родителей:

$$\forall\, x, y \;\; Sibling(x, y) \;\Leftrightarrow\; x \neq y \wedge \exists\, p \;\; Parent(p, x) \wedge Parent(p, y).$$

Формулируя подобные сведения, можно заполнить еще несколько страниц, и в упражнении 8.17 предлагается сделать именно это.

Каждое из этих высказываний может рассматриваться как одна из **аксиом** в проблемной области родства. Аксиомы обычно принято связывать с чисто математическими проблемными областями (и мы вскоре рассмотрим некоторые аксиомы для чисел), но они нужны во всех проблемных областях. Аксиомы предоставляют основную фактическую информацию, на основании которой логическим путем могут быть получены полезные заключения. Кроме того, приведенные выше аксиомы родства также имеют ▶**определения**; последние представлены в форме $\forall\, x, y \;\; P(x, y) \Leftrightarrow \ldots$ Аксиомы определяют функцию *Mother* и предикаты *Husband*, *Male*, *Parent*, *Grandparent* и *Sibling* в терминах других предикатов. Но приведенные выше определения можно "свести" к базовому множеству предикатов (*Child*, *Female* и т.д.), в терминах которых в конечном итоге определяются все остальные предикаты.

Это очень естественный способ, с помощью которого создается представление некоторой проблемной области, и он аналогичен способу, применяемому при создании пакетов программного обеспечения путем последовательного определения процедур из примитивных библиотечных функций. Обратите внимание на то, что множество примитивных базовых предикатов не обязательно должно быть уникальным; с таким же успехом можно использовать множество *Parent* вместо *Child*. Как будет показано ниже, в некоторых проблемных областях нельзя найти четко определимое базовое множество.

Не все логические высказывания о некоторой проблемной области являются аксиомами. Некоторые из них представляют собой ▶**теоремы**, т.е. следуют из аксиом. Например, рассмотрим утверждение, что родственные отношения между братьями и сестрами являются симметричными:

$$\forall\, x, y \;\; Sibling(x, y) \;\Leftrightarrow\; Sibling(y, x).$$

Что это — аксиома или теорема? В действительности это теорема, которая логически следует из аксиомы, определяющей понятие родственных отношений между братьями и сестрами. Если в базу знаний с помощью функции ASK будет введен запрос в виде этого высказывания, то в ответ должно быть получено значение *true*.

С чисто логической точки зрения в базе знаний должны содержаться только аксиомы, но не теоремы, поскольку теоремы не увеличивают множество заключений, которые следуют из базы знаний. Но с практической точки зрения важным свойством теорем является то, что они уменьшают вычислительные издержки на логический вывод новых высказываний. Без них системе формирования рассуждений пришлось бы всякий раз начинать логический вывод с самых фундаментальных принципов, — как если бы математику необходимо было каждый раз заново выводить правила исчисления, приступая к решению новой задачи.

Не все аксиомы имеют определения. Некоторые из них предоставляют более общую информацию об определенных предикатах без формулировки определений. И действительно, некоторые предикаты не имеют полного определения, поскольку мы не знаем достаточно для того, чтобы полностью их охарактеризовать. Например, нельзя найти очевидного способа закончить следующее высказывание с определением понятия "человек":

$$\forall x \; Person(x) \Leftrightarrow \ldots$$

К счастью, логика первого порядка позволяет использовать предикат *Person*, не определяя его полностью. Вместо этого можно записывать частичные спецификации свойств, которыми обладает каждый человек, и свойств, которые идентифицируют некоторый объект как человека:

$$\forall x \; Person(x) \Rightarrow \ldots$$
$$\forall x \; \ldots \Rightarrow Person(x).$$

Аксиомы также могут представлять собой "просто факты", такие как *Male*(*Jim*) или *Spouse*(*Jim*, *Laura*). Подобные факты формируют описания конкретных экземпляров задачи, что позволяет находить ответы на конкретные вопросы. Ответы на эти вопросы представляют собой теоремы, которые следуют из аксиом.

Однако иногда выясняется, что ожидаемые ответы получить не удается, например можно ожидать (в соответствии с законами многих стран), что из аксиомы *Spouse*(*Jim*, *Laura*) удастся логическим путем вывести факт $\neg$*Spouse*(*George*, *Laura*), но этот факт не следует как теорема из приведенных выше аксиом, даже если добавить к ним аксиому $Jim \neq George$, как это было предложено в разделе 8.2.8. Такая ситуация свидетельствует о том, что в базе знаний не хватает какой-то аксиомы. В упражнении 8.9 предлагается предоставить эту аксиому.

8.3.3. Числа, множества и списки

По-видимому, числа представляют собой наиболее яркий пример того, как крупная теория может быть построена из крошечного ядра аксиом. В этом разделе будет описана теория ▶**натуральных чисел**, или неотрицательных целых чисел. Для этого потребуется предикат *NatNum*, который будет принимать истинное значение при использовании параметра, представляющего собой натуральное число. Кроме того, требуется один константный символ, 0, и один функциональный символ, S (*successor* — преемник). ▶**Аксиомы Пеано** определяют натуральные числа и операцию сложения.[8] Натуральные числа определяются рекурсивно:

$$NatNum(0).$$
$$\forall\, n \;\; NatNum(n) \Rightarrow NatNum(S(n)).$$

Это означает, что 0 — натуральное число и для каждого объекта n, если n — натуральное число, $S(n)$ — натуральное число. Поэтому натуральными числами являются 0, $S(0)$, $S(S(0))$ и т.д. Необходимы также аксиомы для ограничения действия функции определения преемника:

$$\forall\, n \;\; 0 \neq S(n);$$
$$\forall\, m, n \;\; m \neq n \Rightarrow S(m) \neq S(n).$$

Теперь мы можем определить операцию сложения в терминах функции определения преемника:

$$\forall\, m \;\; NatNum(m) \Rightarrow +(0, m) = m;$$
$$\forall\, m, n \;\; NatNum(m) \wedge NatNum(n) \Rightarrow +(S(m), n) = S(+(m, n)).$$

В первой из этих аксиом утверждается, что сложение числа 0 с любым натуральным числом m приводит к получению самого числа m. Обратите внимание на использование бинарного функционального символа "+" в терме $+(0, m)$: в обычной математике такой терм принято записывать в виде $0 + m$ с использованием ▶**инфиксной** системы обозначений. (Система обозначений, которая используется в этом разделе для логики первого порядка, называется ▶**префиксной**.) Чтобы было удобнее читать высказывания, касающиеся чисел, в этом разделе будет также разрешено использовать инфиксные обозначения. Кроме того, мы можем записывать $S(n)$ как $n + 1$, в результате вторая аксиома принимает следующий вид:

$$\forall\, m, n \;\; NatNum(m) \wedge NatNum(n) \Rightarrow (m + 1) + n = (m + n) + 1.$$

Эта аксиома сводит сложение к повторному применению функции определения преемника.

---

[8] Аксиомы Пеано включают также принцип индукции, который представляет собой высказывание логики второго порядка, а не логики первого порядка. Важность этого различия объясняется в главе 9.

Такое использование инфиксных обозначений представляет собой пример применения ▶ **синтаксического упрощения**, т.е. расширения или сокращения стандартного синтаксиса, при котором семантика не изменяется. Любое высказывание, в котором используются такие сокращения, может быть "разупрощено" для получения эквивалентного высказывания в обычной логике первого порядка. Другим примером является использование квадратных скобок вместо обычных круглых, чтобы было проще увидеть, какая именно левая скобка соответствует правой. Еще один пример — сворачивание кванторов: замена $\forall x\ \forall y\ P(x, y)$ на $\forall x, y\ P(x, y)$.

После определения понятия сложения становится простой задача определения умножения как повторного сложения, возведения в степень как повторного умножения, целочисленного деления и определения остатков от деления, формулировки понятия простых чисел и т.д. Таким образом, вся теория чисел (включая такие ее приложения, как криптография) может быть сформирована на основе одной константы, одной функции, одного предиката и четырех аксиом.

Проблемная область ▶ **множеств** также является фундаментальной как для математики, так и для рассуждений на уровне здравого смысла. (В действительности теория чисел может быть построена и на основе теории множеств.) Необходимо иметь возможность представлять отдельные множества, включая пустое множество. Также потребуется способ составления множеств из элементов или применения операции к другим множествам. Помимо этого, нужны способ выяснения, входит ли некоторый элемент в состав множества, и возможность отличать множества от объектов, не являющихся множествами.

В качестве синтаксического упрощения будем использовать обычный словарь теории множеств. Пустое множество представляет собой константу, которая записывается как {}. Имеется также один унарный предикат, *Set*, который принимает истинное значение, если его фактическим параметром является множество. Бинарными предикатами являются $x \in s$ (x — элемент множества s) и $s_1 \subseteq s_2$ (s_1 — подмножество, не обязательно строгое, множества s_2). В качестве бинарных функций используются $s_1 \cap s_2$ (пересечение множеств), $s_1 \cup s_2$ (объединение множеств) и $Add(x, s)$ (множество, полученное в результате присоединения элемента x к множеству s). Один из возможных наборов аксиом приведен ниже.

1. Единственными множествами являются пустое множество и множества, полученные путем присоединения некоторого элемента к множеству:

$$\forall s\ Set(s) \Leftrightarrow (s = \{\}) \vee (\exists x, s_2\ Set(s_2) \wedge s = Add(x, s_2)).$$

2. Пустое множество не имеет присоединенных к нему элементов, иными словами, не существует способа разложения множества {} на меньшее множество и еще один элемент:

$$\neg\exists x, s\ Add(x, s) = \{\}.$$

3. Присоединение к множеству элемента, уже имеющегося в этом множестве, не оказывает никакого эффекта:

$$\forall x, s \; x \in s \Leftrightarrow s = Add(x, s).$$

4. Единственными элементами множества являются элементы, которые были к нему присоединены. Мы выразим эту мысль рекурсивно, утверждая, что x — элемент множества s тогда и только тогда, когда s эквивалентно некоторому множеству s_2, к которому присоединен некоторый элемент y, где либо y совпадает с x, либо x является элементом s_2:

$$\forall x, s \; x \in s \Leftrightarrow \exists y, s_2 \; (s = Add(y, s_2) \wedge (x = y \vee x \in s_2)).$$

5. Множество является подмножеством другого множества тогда и только тогда, когда все элементы первого множества являются элементами второго множества:

$$\forall s_1, s_2 \; s_1 \subseteq s_2 \Leftrightarrow (\forall x \; x \in s_1 \Rightarrow x \in s_2).$$

6. Два множества равны тогда и только тогда, когда каждое из них является подмножеством другого:

$$\forall s_1, s_2 \; (s_1 = s_2) \Leftrightarrow (s_1 \subseteq s_2 \wedge s_2 \subseteq s_1).$$

7. Некоторый объект принадлежит к пересечению двух множеств тогда и только тогда, когда он является элементом обоих этих множеств:

$$\forall x, s_1, s_2 \; x \in (s_1 \cap s_2) \Leftrightarrow (x \in s_1 \wedge x \in s_2).$$

8. Некоторый объект принадлежит к объединению двух множеств тогда и только тогда, когда он является элементом любого из этих двух множеств:

$$\forall x, s_1, s_2 \; x \in (s_1 \cup s_2) \Leftrightarrow (x \in s_1 \vee x \in s_2).$$

►**Списки** подобны множествам. Различия между ними состоят в том, что списки упорядочены и один и тот же элемент может появляться в списке несколько раз. Для списков может использоваться словарь ключевых слов языка Lisp: *Nil* — это список-константа без элементов; *Cons*, *Append*, *First* и *Rest* — функции, а *Find* — предикат, выполняющий для списков те же функции, что *Member* выполняет для множеств. *List* — предикат, принимающий истинное значение только при получении параметра, представляющего собой список. Как и в случае множеств, принято использовать некоторые синтаксические упрощения в тех логических высказываниях, в которых применяются списки. Пустой список обозначается как [ ]. Терм *Cons*(x, *Nil*) (т.е. список, содержащий только элемент x) записывается как [x]. Список из нескольких элементов, такой как [A, B, C], соответствует вложенному терму *Cons*(A, *Cons*(B, *Cons*(C, *Nil*))). В упражнении 8.20 предлагается составить необходимый набор аксиом для списков.

8.3.4. Мир вампуса

Некоторые аксиомы логики высказываний для мира вампуса были приведены в главе 7. Аксиомы логики первого порядка, рассматриваемые в этом разделе, являются гораздо более краткими и совершенно естественным способом выражают именно то, что требуется описать в этом мире.

Напомним, что в мире вампуса агент получает вектор восприятий с пятью элементами. Соответствующее высказывание в логике первого порядка, хранящееся в базе знаний, должно включать данные о результатах восприятия и о времени, в которое они были получены; в противном случае у агента возникнет путаница в отношении того, когда и что он воспринимал. Для обозначения моментов времени будут использоваться целые числа. Типичным высказыванием с данными о восприятии является следующее:

$$Percept([Stench, Breeze, Glitter, None, None], 5),$$

где *Percept* — бинарный предикат; *Stench* и т.д. — константы, помещенные в список. Действия в мире вампуса могут быть представлены с помощью логических термов следующим образом:

$$Turn(Right), Turn(Left), Forward, Shoot, Grab, Climb.$$

Чтобы определить, какое действие является наилучшим, программа агента составляет примерно такой запрос:

$$\text{AskVars}(KB, BestAction(a, 5)),$$

который возвращает список связывания, такой как $\{a/Grab\}$. В результате программа агента может вернуть *Grab* как действие, которое должно быть выполнено. Из исходных данных о восприятии следуют некоторые факты о текущем состоянии, например:

$$\begin{aligned}
&\forall t, s, g, w, c \quad Percept([s, Breeze, g, w, c], t) \Rightarrow Breeze(t),\\
&\forall t, s, g, w, c \quad Percept([s, None, g, w, c], t) \Rightarrow \neg Breeze(t),\\
&\forall t, s, b, w, c \quad Percept([s, b, Glitter, w, c], t) \Rightarrow Glitter(t),\\
&\forall t, s, b, w, c \quad Percept([s, b, None, w, c], t) \Rightarrow \neg Glitter(t)
\end{aligned}$$

и т.д. Эти правила являются проявлением простейшей формы процесса формирования рассуждений, называемого восприятием, который будет подробно рассматриваться в главе 25. Обратите внимание на то, что квантификация происходит по переменной t, определяющей момент времени. В логике высказываний приходилось создавать копии каждого высказывания для каждого момента времени.

Также в этой логике могут быть реализованы простые "рефлекторные" варианты поведения с помощью импликационных высказываний с кванторами. Например, можно предусмотреть следующее правило:

$$\forall t \; Glitter(t) \Rightarrow BestAction(Grab, t).$$

При наличии результатов восприятия и правил, приведенных в предыдущих абзацах, применение данного правила привело бы к желаемому заключению *BestAction*(*Grab*, 5), т.е. в данный момент следует выполнить действие *Grab*.

Выше были представлены восприятия и действия агента, а теперь пора представить описание самой среды. Начнем с объектов. Очевидными кандидатами являются квадраты, ямы и вампус. Можно было бы присвоить имя каждому квадрату (*Square*$_{1,2}$ и т.д.), но тогда тот факт, что *Square*$_{1,2}$ и *Square*$_{1,3}$ являются соседними, пришлось бы оформить как "дополнительный" факт и потребовалось бы по одному такому факту для каждой пары квадратов. Поэтому лучше использовать сложный терм, в котором строка и столбец показаны в виде целых чисел, например списковый терм [1, 2]. В таком случае определение понятия соседства любых двух квадратов можно представить следующим образом:

$$\forall x, y, a, b \quad Adjacent([x, y], [a, b]) \Leftrightarrow$$
$$(x = a \wedge (y = b - 1 \vee y = b + 1)) \vee (y = b \wedge (x = a - 1 \vee x = a + 1)).$$

Далее, можно было бы присвоить имя каждой яме, но такое решение является неподходящим по другой причине: нет никакого смысла проводить различия между ямами.[9] Гораздо проще использовать унарный предикат *Pit*, который принимает истинное значение в квадратах, содержащих ямы. Наконец, поскольку имеется точно один вампус, для его представления равным образом подходят и константа *Wumpus*, и унарный предикат (с точки зрения вампуса последний способ обозначения может оказаться даже более почетным).

Местонахождение агента меняется со временем, поэтому будет удобнее применить запись *At*(*Agent*, *s*, *t*) для указания на то, что агент находится в квадрате *s* в момент времени *t*. Связать вампуса с конкретным местоположением, в котором он находится постоянно, можно с помощью высказывания $\forall t \; At(Wumpus, [1, 3], t)$. Теперь следует указать, что в каждый момент времени любые объекты могут находиться только в одном местоположении:

$$\forall x, s_1, s_2, t \; At(x, s_1, t) \wedge At(x, s_2, t) \Rightarrow s_1 = s_2.$$

Зная свое текущее местонахождение, агент посредством логического вывода может выявлять свойства определенного квадрата на основании его текущего восприятия. Например, если агент находится в некотором квадрате и чувствует ветерок, то в этом квадрате присутствует ветерок:

$$\forall s, t \; At(Agent, s, t) \wedge Breeze(t) \Rightarrow Breezy(s).$$

---

[9] Аналогичным образом большинство из нас не присваивают имя каждой птице, которая пролетает у нас над головой или переселяется в теплые края на зиму. С другой стороны, орнитологи, желающие изучить пути миграции, показатели выживания и так далее, *присваивают имя* каждой отловленной птице, закрепив кольцо на ее ноге, поскольку им приходится следить за отдельными птицами.

Агенту важнее знать не то, что он вообще чувствует ветерок, а то, что ветерок чувствуется в определенном *квадрате*, поскольку ему известно, что ямы не меняют своего местоположения. Обратите внимание на то, что предикат *Breezy* (*в квадрате есть ветерок*) не имеет параметра, определяющего момент времени.

Обнаружив, в каких местах чувствуется ветерок (или неприятный запах), а также, что очень важно, в каких местах *не чувствуется* ветерок (или неприятный запах), агент получает возможность логическим путем определять, где расположены ямы (и где находится вампус). В то время как в логике высказываний необходимо было задавать отдельную аксиому для каждого квадрата (см. высказывания R_2 и R_3 в разделе 7.4.3) и приходилось создавать различные наборы таких аксиом для каждого экземпляра мира с иной географией, в логике первого порядка нам потребуется только одна аксиома:

$$\forall s \; Breezy(s) \Leftrightarrow \exists r \; Adjacent(r, s) \wedge Pit(r). \tag{8.4}$$

Аналогичным образом, поскольку в логике первого порядка можно количественно оценивать ход времени, для каждого предиката потребуется только одна аксиома определения преемника вместо новой копии такой аксиомы для каждого временного этапа. Например, аксиома для стрелы (уравнение (7.2) в разделе 7.7.1) приобретает вид

$$\forall t \; HaveArrow(t + 1) \Leftrightarrow (HaveArrow(t) \wedge \neg Action(Shoot, t)).$$

Из этих двух примеров высказываний очевидно, что формулировка описания среды в логике первого порядка не менее лаконична, чем ее исходное описание на русском языке, приведенное в главе 7. Читателю предлагается построить аналогичные аксиомы для определения местоположения и ориентации агента, — в этом случае в аксиомах потребуется количественно оценивать и время, и пространство. Как и в случае оценки состояния в логике высказываний, агент на основании аксиом этого типа сможет использовать логический вывод для отслеживания тех аспектов мира, которые не может наблюдать непосредственно. В главе 11 аксиомы выбора преемника в логике первого порядка рассматриваются глубже, а затем обсуждается возможность их использования для построения планов.

8.4. Инженерия знаний на основе логики первого порядка

В предыдущем разделе иллюстрировалось использование логики первого порядка для представления знаний в трех простых проблемных областях. В этом разделе рассматривается общий процесс конструирования базы знаний, называемый ▶**инженерией знаний**. Инженером по знаниям называется специалист, который исследует конкретную проблемную область, определяет, какие понятия в ней важны, и создает формальное представление объектов и отношений в этой

проблемной области. В этом разделе процесс инженерии знаний будет проиллюстрирован на проблемной области проектирования электронных схем. Применяемый в этом разделе подход будет подходящим для разработки баз знаний *специального назначения*, проблемная область которых тщательно очерчена и круг запросов известен заранее. Базы знаний общего назначения, которые предназначены для поддержки запросов, касающихся широкого диапазона человеческих знаний, обсуждаются в главе 10.

8.4.1. Процесс инженерии знаний

Отдельные проекты в области инженерии знаний значительно отличаются друг от друга по содержанию, охвату темы и сложности, однако все они включают перечисленные ниже этапы.

1. *Определение задания.* Инженер по знаниям должен очертить круг вопросов, которые будет поддерживать база знаний, а также виды фактов, которые будут доступны каждой конкретной реализации задачи. Например, должна ли база знаний о вампусе предоставлять возможность выбирать действия или от нее требуется только давать ответы на вопросы о содержимом окружающей среды? Должны ли факты, полученные от датчиков, включать сведения о текущем местонахождении? Само задание определяет, какие знания должны быть представлены в базе, чтобы можно было связать определенные реализации задачи с ответами. Этот этап аналогичен процессу PEAS проектирования агентов, описанному в главе 2.
2. *Сбор относящихся к делу знаний.* Инженер по знаниям может уже быть экспертом в рассматриваемой проблемной области или же ему может потребоваться общаться с настоящими экспертами для выявления всего, что они знают — этот процесс называется ▶**приобретением знаний**. На этом этапе знания еще не представлены формально. Его назначение состоит в том, чтобы понять, каким должен быть диапазон знаний в базе знаний, определяемый самим заданием, а также разобраться в том, как фактически функционирует рассматриваемая проблемная область.

   Для мира вампуса, определенного с помощью искусственного набора правил, относящиеся к делу (релевантные) знания выявить несложно. (Однако следует отметить, что определение понятия соседства квадратов не формулировалось явно в правилах функционирования мира вампуса.) Для реальных проблемных областей задача выявления релевантных знаний может оказаться весьма сложной, например система для эмуляции работы спроектированных СБИС может требовать или не требовать учета паразитных емкостей и поверхностных эффектов.
3. *Определение словаря предикатов, функций и констант.* Иначе говоря, это перевод всех важных концепций уровня проблемной области в уровень

логических имен. При этом потребуется найти ответы на многие вопросы о *стиле* инженерии знаний. Как и от стиля программирования, от выбранных решений может существенным образом зависеть окончательный успех проекта. Например, должны ли ямы быть представлены с помощью объектов или с помощью унарного предиката, определенного на квадратах? Должна ли ориентация агента быть задана в виде функции или в виде предиката? Должно ли местонахождение вампуса зависеть от времени? Когда все необходимые решения будут приняты, конечным результатом этих усилий будет словарь, который принято называть ►**онтологией** проблемной области. Само слово *онтология* означает определенную теорию бытия или существования. Онтология проблемной области определяет, какого рода объекты в ней существуют, но не определяет их конкретных свойств и взаимосвязей.

4. *Кодирование общих знаний о проблемной области.* Инженер по знаниям записывает аксиомы для всех термов словаря. Тем самым он закрепляет (насколько это возможно) смысл этих термов, позволяя экспертам проверить их содержание. На этом этапе часто обнаруживаются неправильные трактовки или пропуски в словаре, которые необходимо исправить, возвратившись на этап 3 и вновь пройти данную итерацию в текущем процессе проектирования.
5. *Кодирование описания экземпляра задачи.* Если онтология хорошо продумана, этот этап будет несложным. Он сводится к написанию простых атомарных высказываний об отдельных образцах понятий, которые уже являются частью онтологии. Для логического агента сведения об отдельных реализациях задачи предоставляются датчиками, а сама "бестелесная" база знаний снабжается дополнительными высказываниями таким же образом, как традиционные программы снабжаются входными данными.
6. *Передача запросов процедуре логического вывода и получение ответов.* Именно здесь нас ожидает награда: теперь можно применить процедуру логического вывода к аксиомам и фактам о конкретной задаче для получения фактов, которые нам интересно узнать. В результате становится возможным избавиться от необходимости написания алгоритма решения для конкретного приложения.
7. *Отладка и оценка базы знаний.* К сожалению, ответы на запросы редко оказываются правильными уже с первой попытки. Точнее, ответы будут правильными *для базы знаний в том виде, в каком она написана*, при условии, что процедура логического вывода является непротиворечивой, но они могут оказаться не совсем такими, каких ожидает пользователь. Например, если недостает какой-то аксиомы, то найти ответ на некоторые запросы в данной базе знаний будет невозможно. В результате потребуется провести продуманный процесс отладки. Недостающие или слишком слабые аксиомы могут быть легко выявлены путем обнаружения участков, на которых

неожиданно обрывается цепочка этапов логического вывода. Например, если база знаний содержит следующую диагностическую аксиому (см. упражнение 8.16) для обнаружения вампуса:

$$\forall\, s\ Smelly(s) \Rightarrow Adjacent(Home(Wumpus), s)$$

вместо аксиомы с двумя условиями, то агент никогда не сможет доказать *отсутствие* вампусов. Неправильные аксиомы можно выявить при отладке потому, что они являются *ложными* утверждениями о мире. Например, аксиома

$$\forall\, x\ NumOfLegs(x, 4) \Rightarrow Mammal(x)$$

является ложной, поскольку относит к млекопитающим рептилий, амфибий и, что еще важнее, столы с четырьмя ножками. ➨ *Ложность этого высказывания может быть определена независимо от остальной части базы знаний.* В противоположность этому типичная ошибка в программе выглядит примерно так:

```
offset = position + 1.
```

Глядя на эту строку, невозможно сказать, следует ли переменной `offset` присвоить значение `position` или `position + 1`, без понимания окружающего ее контекста.

Достигнув состояния, когда очевидных ошибок в базе знаний уже не осталось, заманчиво немедленно объявить об успехе. Но на самом деле, когда явных ошибок уже нет, самое время формально оценить созданную систему, запустив ее на тестовом наборе запросов и определив, насколько правильным будет полученный результат. Без объективного измерения, слишком легко убедить себя, что работа действительно завершена. Чтобы лучше понять этот процесс, состоящий из семи этапов, ниже мы применим его к расширенному примеру — к проблемной области электронных схем.

8.4.2. Проблемная область электронных схем

В этом примере мы разработаем онтологию и базу знаний, позволяющую рассуждать о цифровых электронных схемах такого типа, как показано на рис. 8.6, и будем руководствоваться описанным выше семиэтапным процессом инженерии знаний.

Определение задания

С цифровыми схемами связано много задач, требующих логических рассуждений. На самом высоком уровне требуется проанализировать функциональное назначение схемы. Например, действительно ли приведенная на рис. 8.6 схема выполняет сложение должным образом? Если на все входы подается высокий потенциал, то каким будет потенциал на выходе логического элемента A_2? Интерес

представляют также вопросы о структуре схемы. Например, что представляют собой все логические элементы, подключенные к первой входной клемме? Содержит ли эта схема петли обратной связи? В этом и заключаются задачи, рассматриваемые в данном разделе. Существуют также более детализированные уровни анализа, включая те, которые относятся к определению продолжительности задержек, площади схемы, потреблению энергии, стоимости производства и т.д. Для каждого из этих уровней могут потребоваться дополнительные знания.

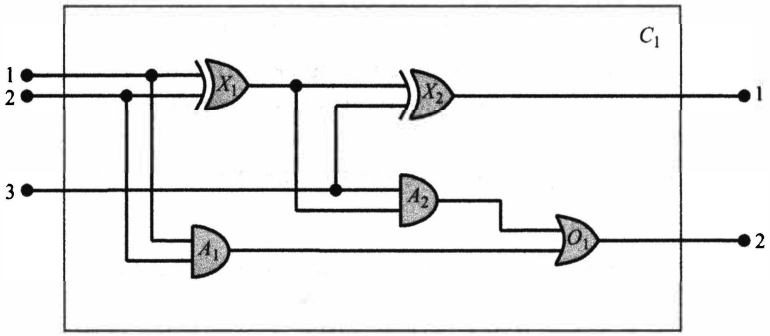


Рис. 8.6. Цифровая схема C_1, предназначенная для использования в качестве однобитового полного сумматора. На первые два входа подаются два бита, подлежащие сложению, а на третий вход подается бит переноса. На первом выходе находится сумма, а на втором — бит переноса для следующего сумматора. Схема включает два логических элемента Исключающее ИЛИ (X_1, X_2), два логических элемента И (A_1, A_2) и один логический элемент ИЛИ (O_1)

Сбор относящихся к делу знаний

Что мы знаем о цифровых схемах? Для наших целей достаточно знать, что они состоят из проводов и логических элементов. Сигналы распространяются по проводам к входным клеммам логических элементов, а каждый логический элемент вырабатывает на выходной клемме сигнал, который распространяется по другому проводу. Чтобы определить, какими должны быть эти сигналы, нам необходимо знать, как логические элементы преобразуют свои входные сигналы. Существует четыре основных типа логических элементов: логические элементы AND (И), OR (ИЛИ) и XOR (Исключающее ИЛИ) имеют по две входные клеммы, а логический элемент NOT (НЕ) имеет одну входную клемму. Все логические элементы имеют одну выходную клемму. Схемы, как и логические элементы, имеют входные и выходные клеммы.

Чтобы рассуждать о функциональных назначениях и связях в электронной схеме, не нужно вести речь о самих проводах, о путях, по которым они проложены, или о соединениях, в которых они встречаются. Имеют значение только

соединения между клеммами — можно утверждать, что одна выходная клемма соединена с другой, входной клеммой, не упоминая о том, как и чем они фактически связаны. В этой проблемной области имеется также множество других факторов, не имеющих отношения к нашему анализу, таких как размеры, форма, цвет или стоимость различных компонентов.

Если бы нашей целью было что-то иное, а не просто проверка схемы на уровне логических элементов, то онтология была бы другой. Например, если бы нас интересовала отладка неисправных электронных схем, то, по-видимому, целесообразно было бы включить в онтологию провода, поскольку неисправный провод может исказить проходящий по нему сигнал. С другой стороны, для устранения ошибок синхронизации потребовалось бы включить в рассмотрение задержки логических элементов. А если бы мы были заинтересованы в проектировании продукта, который должен быть прибыльным, то для нас имели бы значение такие данные, как стоимость производства электронной схемы или ее быстродействие в сравнении с другими продуктами на рынке.

Определение словаря

Теперь нам известно, что речь пойдет о схемах, клеммах, сигналах и логических элементах. На данном этапе необходимо выбрать функции, предикаты и константы для их представления. Прежде всего необходимо иметь возможность отличать один логический элемент от других логических элементов. Каждый логический элемент будет представлен объектом, именованным константой, посредством которой мы будем утверждать, что это элемент, скажем, $Gate(X_1)$. Поведение каждого элемента определяется его типом, определяемым одной из констант *AND*, *OR*, *XOR* и *NOT*. Поскольку каждый логический элемент имеет точно один тип, можно воспользоваться соответствующей функцией: $Type(X_1) = XOR$. Схемы, как и логические элементы, будут идентифицироваться предикатом: $Circuit(C_1)$.

Далее рассмотрим клеммы, которые будут идентифицироваться предикатом *Terminal*(x). Схема может иметь одну или несколько входных клемм и одну или несколько выходных клемм. Будем использовать функцию $In(1, X_1)$ для обозначения первой входной клеммы логического элемента X_1. Аналогичная функция $Out(n, c)$ будет использоваться для выходных клемм. Функция $Arity(c, i, j)$ указывает на то, что схема c имеет i входных и j выходных клемм. Связь между логическими элементами может быть представлена с помощью предиката *Connected*, который принимает в качестве параметров имена двух клемм, как, например, $Connected(Out(1, X_1), In(1, X_2))$.

И наконец, необходимо знать, включен или выключен сигнал на клемме. Одна из возможностей состоит в использовании унарного предиката $On(t)$, который принимает истинное значение, когда сигнал на клемме t включен. Однако при этом затрудняется постановка таких вопросов, как "Каковы возможные значения сигналов на выходных клеммах схемы C_1?" Поэтому введем в качестве объектов два значения сигнала, 1 (включено) и 0 (выключено), и функцию $Signal(t)$, которая определяет значение сигнала для клеммы t.

Кодирование общих знаний о проблемной области

Одним из признаков наличия хорошей онтологии является то, что при ее использовании требуется определить очень немного общих правил, которые могут быть сформулированы четко и кратко. Вот все аксиомы, которые будут нам необходимы.

1. Если две клеммы соединены, то на них присутствует один и тот же сигнал:

$$\forall t_1, t_2 \quad Terminal(t_1) \wedge Terminal(t_2) \wedge Connected(t_1, t_2) \Rightarrow Signal(t_1) = Signal(t_2).$$

2. Сигнал на каждой клемме равен либо 1, либо 0:

$$\forall t \quad Terminal(t) \Rightarrow Signal(t) = 1 \vee Signal(t) = 0.$$

3. Предикат *Connected* является коммутативным:

$$\forall t_1, t_2 \quad Connected(t_1, t_2) \Leftrightarrow Connected(t_2, t_1).$$

4. Существует четыре типа элементов:

$$\forall g \quad Gate(g) \wedge k = Type(g) \Rightarrow k = AND \vee k = OR \vee k = XOR \vee k = NOT.$$

5. На выходе логического элемента *AND* присутствует 0 тогда и только тогда, когда на любом из его входов присутствует 0:

$$\forall g \quad Gate(g) \wedge Type(g) = AND \Rightarrow Signal(Out(1, g)) = 0 \Leftrightarrow \exists n \quad Signal(In(n, g)) = 0.$$

6. На выходе логического элемента *OR* присутствует 1 тогда и только тогда, когда на любом из его входов присутствует 1:

$$\forall g \quad Gate(g) \wedge Type(g) = OR \Rightarrow Signal(Out(1, g)) = 1 \Leftrightarrow \exists n \quad Signal(In(n, g)) = 1.$$

7. На выходе логического элемента *XOR* присутствует 1 тогда и только тогда, когда на его входах присутствуют разные сигналы:

$$\forall g \quad Gate(g) \wedge Type(g) = XOR \Rightarrow Signal(Out(1, g)) = 1 \Leftrightarrow Signal(In(1, g)) \neq Signal(In(2, g)).$$

8. Выход логического элемента *NOT* противоположен его входу:

$$\forall g \quad Gate(g) \wedge Type(g) = NOT \Rightarrow Signal(Out(1, g)) \neq Signal(In(1, g).$$

9. Логические элементы (кроме *NOT*) имеют два входа и один выход:

$$\forall g \quad Gate(g) \wedge Type(g) = NOT \Rightarrow Arity(g, 1, 1),$$
$$\forall g \quad Gate(g) \wedge k = Type(g) \wedge (k = AND \vee k = OR \vee k = XOR) \Rightarrow Arity(g, 2, 1).$$

10. Логический элемент имеет клеммы, количество которых определяется его входной и выходной арностью, и ничем, кроме арности:

$$\forall c, i, j \quad Circuit(c) \wedge Arity(c, i, j) \Rightarrow$$
$$\forall n \quad (n \leq i \Rightarrow Terminal(In(n, c))) \wedge (n > i \Rightarrow In(n, c) = Nothing) \wedge$$
$$\forall n \quad (n \leq j \Rightarrow Terminal(Out(n, c))) \wedge (n > j \Rightarrow Out(n, c) = Nothing).$$

11. Логические элементы, клеммы и сигналы являются различными объектами:

$$\forall g, t, s \quad Gate(g) \wedge Terminal(t) \wedge Signal(s) \Rightarrow$$
$$g \neq t \wedge g \neq s \wedge t \neq s.$$

12. Логические элементы — это цепи:

$$\forall g \quad Gate(g) \Rightarrow Circuit(g).$$

Кодирование описания экземпляра задачи

Схема, показанная на рис. 8.6, представлена как схема C_1 со следующим описанием. Прежде всего определим типы всех логических элементов.

$$Circuit(C_1) \wedge Arity(C_1, 3, 2)$$
$$Gate(X_1) \wedge Type(X_1) = XOR$$
$$Gate(X_2) \wedge Type(X_2) = XOR$$
$$Gate(A_1) \wedge Type(A_1) = AND$$
$$Gate(A_2) \wedge Type(A_2) = AND$$
$$Gate(O_1) \wedge Type(O_1) = OR$$

Теперь определим связи между ними.

$$Connected(Out(1, X_1), In(1, X_2)) \qquad Connected(In(1, C_1), In(1, X_1))$$
$$Connected(Out(1, X_1), In(2, A_2)) \qquad Connected(In(1, C_1), In(1, A_1))$$
$$Connected(Out(1, A_2), In(1, O_1)) \qquad Connected(In(2, C_1), In(2, X_1))$$
$$Connected(Out(1, A_1), In(2, O_1)) \qquad Connected(In(2, C_1), In(2, A_1))$$
$$Connected(Out(1, X_2), Out(1, C_1)) \qquad Connected(In(3, C_1), In(2, X_2))$$
$$Connected(Out(1, O_1), Out(2, C_1)) \qquad Connected(In(3, C_1), In(1, A_2))$$

Передача запросов процедуре логического вывода

Какая комбинация входных сигналов вызовет появление 0 на первом выходе схемы C_1 (бит суммы) и появление 1 на втором выходе схемы C_1 (бит переноса)?

$$\exists i_1, i_2, i_3 \quad Signal(In(1, C_1)) = i_1 \wedge Signal(In(2, C_1)) = i_2 \wedge Signal(In(3, C_1)) = i_3$$
$$\wedge Signal(Out(1, C_1)) = 0 \wedge Signal(Out(2, C_1)) = 1.$$

Ответами являются такие подстановки значений для переменных i_1, i_2 и i_3, что результирующее высказывание следует из базы знаний. Функция AskVars вернет три такие подстановки:

$$\{i_1/1, i_2/1, i_3/0\} \quad \{i_1/1, i_2/0, i_3/1\} \quad \{i_1/0, i_2/1, i_3/1\}.$$

Каковы возможные множества значений сигналов на всех клеммах этой схемы сумматора?

$$\exists\, i_1, i_2, i_3, o_1, o_2 \quad Signal(In(1, C_1)) = i_1 \wedge Signal(In(2, C_1)) = i_2 \\ \wedge\, Signal(In(3, C_1)) = i_3 \wedge Signal(Out(1, C_1)) = o_1 \wedge Signal(Out(2, C_1)) = o_2$$

Этот последний запрос должен вернуть полную таблицу входов и выходов для данного устройства, которая может использоваться для проверки того, действительно ли эта схема правильно суммирует входные данные. Это простой пример ►**проверки схемы**. Можно применять приведенное здесь определение данной схемы и для создания более крупных цифровых систем, для которых также может осуществляться процедура проверки такого же рода (см. упр. 8.33). Для структурированной разработки базы знаний такого же рода подходят многие проблемные области, в которых более сложные понятия определяются на основе более простых понятий.

Отладка базы знаний

В процессе отладки эту базу знаний можно тем или иным способом *нарушить* — для определения того, какие виды ошибочного поведения будут с этим связаны. Например, предположим, что раздел 8.2.8 не был прочитан и в результате оказалось пропущенным утверждение, что $1 \neq 0$. В результате система внезапно потеряла способность доказывать наличие каких-либо выходных сигналов для данной схемы, за исключением тех случаев, когда на вход поданы сигналы 000 и 110. Наличие этой проблемы можно выявить, запрашивая выходные сигналы каждого логического элемента. Например, можно ввести запрос

$$\exists\, i_1, i_2, o \quad Signal(In(1, C_1)) = i_1 \wedge Signal(In(2, C_1)) = i_2 \wedge Signal(Out(1, X_1)) = o,$$

который позволит обнаружить, что выходные значения элемента X_1 становятся неизвестными в случае подачи на его вход сигналов 10 и 01. Теперь рассмотрим аксиому для логических элементов XOR применительно к X_1:

$$Signal(Out(1, X_1)) = 1 \;\Leftrightarrow\; Signal(In(1, X_1)) = Signal(In(2, X_1)).$$

Если известно, что на вход поданы сигналы, скажем, 1 и 0, то эта аксиома сводится к следующей:

$$Signal(Out(1, X_1)) = 1 \;\Leftrightarrow\; 1 \neq 0.$$

Теперь проблема становится очевидной: система неспособна вывести логическое заключение, что $Signal(Out(1, X_1)) = 1$, поэтому ей необходимо сообщить, что $1 \neq 0$.

Резюме

В этой главе приведены вводные сведения о **логике первого порядка** — языке представления, гораздо более мощном по сравнению с логикой высказываний. Ниже перечислены наиболее важные понятия, представленные в данной главе.

- Языки представления знаний должны быть декларативными, композиционными, выразительными, независимыми от контекста и непротиворечивыми.
- Различные варианты логики отличаются друг от друга по своему **онтологическому вкладу** и **эпистемологическому вкладу**. Вклад логики высказываний состоит только в сведениях о существовании фактов, тогда как логика первого порядка дополнительно охватывает сведения о существовании объектов и их отношениях, что повышает ее выразительную мощь до уровня, необходимого для таких проблемных областей, как мир вампуса или электронные схемы.
- Как логика высказываний, так и логика первого порядка испытывают затруднения в представлении неопределенных высказываний. Эти затруднения ограничивают возможность их применения в тех проблемных областях, в которых обязательно присутствуют личные суждения, таких как политика или кухня.
- Синтаксис логики первого порядка построен на синтаксисе логики высказываний. К последнему добавлены лишь термы для представления объектов и два квантора — квантор всеобщности и квантор существования, — необходимые для построения утверждений обо всех или только о некоторых возможных значениях количественных переменных.
- Для логики первого порядка **возможный мир** или **модель** включает множество объектов и **интерпретацию**, отображающую символы констант на объекты, символы предикатов на отношения между объектами и символы функций на те функции, которые могут применяться к объектам.
- Атомарное высказывание будет истинным только тогда, когда обозначенное предикатом отношение действительно имеет место между объектами, обозначенными входящими в это высказывание термами. **Расширенные интерпретации**, отображающие переменные кванторов на объекты в модели, определяют истинность высказываний в кванторах.
- Разработка базы знаний в логике первого порядка требует проведения тщательного анализа проблемной области, выбора словаря и составления аксиом, необходимых для поддержки желаемых процедур логического вывода.

Библиографические и исторические заметки

Хотя первые попытки применять обобщения к объектам предпринимались уже в логике Аристотеля, им было еще очень далеко до выразительной силы логики первого порядка. Основное препятствие в их дальнейшей разработке заключалось в концентрации лишь на одноместных предикатах и исключении из рассмотрения многоместных предикатов отношений. Первый систематический анализ отношений был проведен Аугустусом Де Морганом ([561], 1864), который привел следующий пример с целью продемонстрировать тот тип логического вывода, справиться с которым логика Аристотеля была неспособна: "Все лошади — животные, следовательно, голова лошади — это голова животного". Подобный вывод недоступен логике Аристотеля, потому что в любом допустимом правиле, способном обеспечить этот логический вывод, сначала следует проанализировать высказывание, используя двухместный предикат "x является головой y". Логика отношений глубоко исследовалась Чарльзом Сандерсом Пирсом (Пирс [1764], 1870; Мисак [1590], 2004).

Появление подлинной логики первого порядка датируется введением кванторов Готтлобом Фреге в его книге *Begriffschrift* ("Система обозначения понятий") ([775], 1879). Пирс ([1765], 1883) также разработал логику первого порядка, причем независимо от Фреге, но сделал это немного позже. Предложенная Фреге возможность вкладывать кванторы была большим шагом вперед, но он использовал слишком громоздкую систему обозначений. Современная система обозначений для логики первого порядка появилась в основном благодаря Джузеппе Пеано ([1747], 1889), но ее семантика практически идентична семантике, предложенной в работе Фреге. Хотя сейчас это кажется довольно странным, аксиомы Пеано в значительной степени обязаны своим появлением работе Грассмана ([916], 1861) и публикации Дедекинда ([590], 1888).

Леопольд Лёвенхейм ([1451], 1915) дал систематическую трактовку теории моделей для логики первого порядка. Также в его статье символ равенства рассматривался как неотъемлемая часть логики. Результаты Лёвенхейма были дополнительно расширены Торальфом Сколемом ([2081], 1920). Альфред Тарски ([2174], 1935; [2176], 1956) дал явное определение понятий истинности и модельно-теоретического выполнения в логике первого порядка с использованием теории множеств.

Первым, кто предложил использовать логику первого порядка в качестве инструмента для создания систем искусственного интеллекта, был Джон Маккарти ([1529], 1958). Перспективы искусственного интеллекта на основе логики значительно расширились в результате разработки Робинсоном резолюции ([1900], 1965) — полной процедуры вывода для логики первого порядка. Подход, позднее получивший название *логицистского*, зародился в Станфордском университете. Корделл Грин ([919], 1969; [920], 1969) разработал первую систему формирования рассуждений в логике первого порядка, QA3, что привело к первым попыткам

создания логического робота в институте SRI (Файкс и Нильссон [738], 1971). Логика первого порядка была применена Зохаром Манна и Ричардом Валдингером ([1483], 1971) для формирования рассуждений о программах, а позднее — Майклом Генезеретом ([835], 1984) для формирования рассуждений об электронных схемах. В Европе для использования в лингвистическом анализе ([466], 1973) и для создания общих декларативных систем ([1295], 1974) было разработано логическое программирование (ограниченная форма проведения рассуждений в логике первого порядка). Кроме того, в ходе разработки проекта LCF (Logic for Computable Functions) в Эдинбурге (Гордон и др. [905], 1979) был внесен большой вклад в вычислительную логику. Подробнее история этих разработок будет описана в главах 9 и 10.

Практические приложения, построенные с использованием логики первого порядка, включают систему оценки производственных требований для продуктов электроники (Мэннион [1488], 2002), систему рассуждений о политике доступа к файлам и управления цифровыми правами (Гальперн и Вайсман [953], 2008), а также систему автоматизированного составления веб-сервисов (МакИлрайт и Цзэн [1549], 2001).

Комментарии к гипотезе Уорфа ([2333], 1956) и проблеме языка и мышления в целом встречаются в нескольких книгах (Пуллум [1825], 1991; Пинкер [1794], 2003), включая статьи с, казалось бы, противоположными названиями: "Почему мир выглядит иным на других языках" (Детчер [613], 2010) и "Почему мир выглядит одинаково на любом языке" (МакУотер [1554], 2014), хотя оба автора согласны, что различия есть и эти различия невелики. В теории "Теория" (Гопник и Глимо [902], 2002; Тененбаум и др. [2190], 2007) процесс изучения мира ребенком воспринимается как аналогичный процессу создания научных теорий. Точно так же, как оценки и заключения системы машинного обучения сильно зависят от словаря, на основе которого выполнялось ее обучение, так же и формулировки теорий ребенка будут зависеть от языковой среды, в которой происходит его обучение.

Существует целый ряд хороших вводных учебников по логике первого порядка, в том числе созданных некоторыми из ведущих фигур в истории логики: Альфред Тарский ([2175], 1941), Алонзо Черч ([432], 1956) и У.В. Квайн ([1833], 1982) (один из наиболее популярных). В книге Эндертона [688] (1972) материал в большей степени ориентирован на математику. В высшей степени формальная трактовка логики первого порядка, наряду с описанием более сложных тем логики, предоставлена в учебнике Белла и Макховера ([159], 1977). Позднее Манна и Уолдингер ([1484], 1985) предоставили удобное для восприятия введение в логику с точки зрения компьютерных наук, тогда как Хут и Райан ([1103], 2004) в значительно большей степени сконцентрировались на верификации программ. В книге Барвайза и Этченменди [140] (2002) принят подход, во многом подобный тому, который использован здесь. Смаллиан ([2104], 1995) представляет результаты кратко, используя формат таблицы, а Галлье ([807], 1986) дает чрезвычайно строгое математическое описание логики первого порядка наряду со значительным объемом

материала, который может использоваться в области автоматического формирования рассуждений. В книге Генезерета и Нильссона *Logical Foundations of Artificial Intelligence* (Логические основы искусственного интеллекта) ([836], 1987) приведено не только солидное введение в логику, но и первое систематическое описание логических агентов с восприятиями и действиями. Также имеются два хороших справочника: ван Бенты и тер Мюле ([249], 1997), а также Робинсона и Воронкова ([1899], 2001). Журналом, публикующим материалы в области чистой математической логики, является *Journal of Symbolic Logic*, тогда как *Journal of Applied Logic* в большей степени ориентирован на интересы из области искусственного интеллекта.

Упражнения

8.1. Логическая база знаний представляет мир с использованием множества высказываний без явной структуры. С другой стороны, **аналогическое** представление имеет физическую структуру, которая непосредственно соответствует структуре представляемого объекта. Взгляните на карту дорог своей страны как на образец аналогического представления фактов о стране — она представляет эти факты на языке карт. Двухмерная структура карты соответствует двухмерной поверхности региона.

а) Приведите пять примеров символов на языке карт.

б) Явным высказыванием называется высказывание, формулируемое самим разработчиком представления. Неявным высказыванием называется высказывание, которое следует из явных высказываний в соответствии со свойствами аналогического представления. Приведите по три примера неявных и явных высказываний на языке карт.

в) Приведите три примера сведений о физической структуре территории вашей страны, которые не могут быть представлены на языке карт.

г) Приведите два примера фактов, которые гораздо легче выразить на языке карт, чем на языке логики первого порядка.

д) Приведите еще два примера полезных аналогических представлений. Каковы преимущества и недостатки каждого из этих языков, логического и аналогического?

8.2. Рассмотрите базу знаний, содержащую только два высказывания: $P(a)$ и $P(b)$. Следует ли из этой базы знаний высказывание $\forall x\ P(x)$? Объясните свой ответ в терминах моделей.

8.3. Является ли допустимым высказывание $\exists x, y\ x=y$? Объясните, почему.

8.4. Напишите такое логическое высказывание, что каждый мир, в котором оно является истинным, содержит точно один объект.

8.5. Напишите такое логическое высказывание, что каждый мир, в котором оно является истинным, содержит точно два объекта.

8.6. Рассмотрите словарь символов, содержащий c символов констант, p_k символов предикатов, каждый с арностью k, и f_k символов функций, каждый с арностью k, где $1 \leq k \leq A$. Допустим, что размер проблемной области ограничен значением D. Для каждой конкретной модели каждый символ предиката или функции отображается соответственно на отношение или функцию с той же арностью. Можно принять

предположение, что функции в этой модели допускают, чтобы некоторые входные кортежи не имели значения для этой функции (т.е. чтобы значение было невидимым объектом). Выведите формулу определения количества возможных моделей для проблемной области с D элементами. Не беспокойтесь об устранении избыточных комбинаций.

8.7. Какие из следующих высказываний являются допустимыми (обязательно истинными)?

а) $(\exists x\ x=x) \Rightarrow (\forall y\ \exists z\ y=z)$

б) $\forall x\ P(x) \vee \neg P(x)$

в) $\forall x\ Smart(x) \vee (x=x)$

8.8. Рассмотрите вариант семантики для логики первого порядка, в которой допускаются модели с пустыми проблемными областями. Приведите по меньшей мере два примера высказываний, которые будут допустимыми в соответствии со стандартной семантикой, но не будут таковыми в соответствии с новой семантикой. Обсудите, какой результат имеет больший интуитивный смысл для ваших примеров.

8.9. Является ли факт $\neg Spouse(George, Laura)$ логическим следствием из фактов $Jim \neq George$ и $Spouse(Jim, Laura)$? Если это так, то приведите доказательство, а если нет, то предоставьте необходимые дополнительные аксиомы. Что произойдет, если использовать *Spouse* как символ унарной функции, а не как символ бинарного предиката?

8.10. В этом упражнении используются функция *MapColor* и предикаты $In(x, y)$, $Borders(x, y)$ и $Country(x)$, аргументами которых являются географические регионы, наряду с символами констант для разных регионов. В каждом из следующих пунктов приведены предложение на естественном языке и ряд возможных логических выражений. Для каждого из этих логических выражений укажите, является ли оно 1) точно отражающим смысл исходного предложения; 2) синтаксически неверным и потому бессмысленным; 3) синтаксически правильным, но не отражающим смысл исходного предложения.

а) И Париж, и Марсель находятся во Франции.

1. $In(Paris \wedge Marseilles, France)$
2. $In(Paris, France) \wedge In(Marseilles, France)$
3. $In(Paris, France) \vee In(Marseilles, France)$

б) Существует страна, граничащая и с Ираком, и с Пакистаном.

1. $\exists c\ Country(c) \wedge Border(c, Iraq) \wedge Border(c, Pakistan)$
2. $\exists c\ Country(c) \Rightarrow [Border(c, Iraq) \wedge Border(c, Pakistan)]$
3. $[\exists c\ Country(c)] \Rightarrow [Border(c, Iraq) \wedge Border(c, Pakistan)]$
4. $\exists c\ Border(Country(c), Iraq \wedge Pakistan)$

в) Все страны, которые граничат с Эквадором, находятся в Южной Америке.

1. $\forall c\ Country(c) \wedge Border(c, Ecuador) \Rightarrow In(c, SouthAmerica)$
2. $\forall c\ Country(c) \Rightarrow [Border(c, Ecuador) \Rightarrow In(c, SouthAmerica)]$
3. $\forall c\ [Country(c) \Rightarrow Border(c, Ecuador)] \Rightarrow In(c, SouthAmerica)$
4. $\forall c\ Country(c) \wedge Border(c, Ecuador) \wedge In(c, SouthAmerica)$

г) Ни один регион в Южной Америке не граничит с регионом в Европе.

1. $\neg[\exists c, d\ In(c, SouthAmerica) \wedge In(d, Europe) \wedge Borders(c, d)]$
2. $\forall c, d\ [In(c, SouthAmerica) \wedge In(d, Europe)] \Rightarrow \neg Borders(c, d)]$
3. $\neg \forall c\ In(c, SouthAmerica) \Rightarrow \exists d\ In(d, Europe) \wedge \neg Borders(c, d)$
4. $\forall c\ In(c, SouthAmerica) \Rightarrow \forall d\ In(d, Europe) \Rightarrow \neg Borders(c, d)$

д) Нет двух соседних стран, имеющих одинаковый цвет на карте.

1. $\forall x, y\ \neg Country(x) \vee \neg Country(y) \vee \neg Borders(x, y) \vee \neg(MapColor(x) = MapColor(y))$
2. $\forall x, y\ (Country(x) \wedge Country(y) \wedge Borders(x, y) \wedge \neg(x = y)) \Rightarrow \neg(MapColor(x) = MapColor(y))$
3. $\forall x, y\ Country(x) \wedge Country(y) \wedge Borders(x, y) \wedge \neg(MapColor(x) = MapColor(y))$
4. $\forall x, y\ (Country(x) \wedge Country(y) \wedge Borders(x, y)) \Rightarrow MapColor(x \neq y)$

8.11. Рассмотрите словарь со следующими символами.

Occupation(p, o): предикат. Человек p имеет профессию o.
Customer(p_1, p_2): предикат. Человек p_1 является клиентом человека p_2.
Boss(p_1, p_2): предикат. Человек p_1 является начальником человека p_2.
Doctor, *Surgeon*, *Lawyer*, *Actor*: константы, определяющие профессии.
Emily, *Joe*: константы, определяющие людей.

Используйте эти символы для записи следующих утверждений в логике первого порядка.

а) Эмили — или хирург, или юрист.
б) Джо — актер, но у него есть и другая работа.
в) Все хирурги — врачи.
г) У Джо нет адвоката (т.е. он не является клиентом какого-либо адвоката).
д) У Эмили есть начальник, юрист.
е) Существует адвокат, все клиенты которого являются врачами.
ж) У каждого хирурга есть адвокат.

8.12. В каждом из следующих пунктов приведены предложение на естественном языке и ряд возможных логических выражений. Для каждого из этих логических выражений укажите, является ли оно 1) точно отражающим смысл исходного предложения; 2) синтаксически неверным и потому бессмысленным; 3) синтаксически правильным, но не отражающим смысл исходного предложения.

а) Каждый кот любит свою мать или отца.

1. $\forall x\ Cat(x) \Rightarrow Loves(x, Mother(x) \vee Father(x))$
2. $\forall x\ \neg Cat(x) \vee Loves(x, Mother(x)) \vee Loves(x, Father(x))$
3. $\forall x\ Cat(x) \wedge (Loves(x, Mother(x)) \vee Loves(x, Father(x)))$

б) Каждая собака, которая любит одного из своих братьев, счастлива.

1. $\forall x\ Dog(x) \wedge (\exists y\ Brother(y, x) \wedge Loves(x, y)) \Rightarrow Happy(x)$
2. $\forall x, y\ Dog(x) \wedge Brother(y, x) \wedge Loves(x, y) \Rightarrow Happy(x)$
3. $\forall x\ Dog(x) \wedge [\forall y\ Brother(y, x) \Leftrightarrow Loves(x, y)] \Rightarrow Happy(x)$

в) Ни одна собака не кусает ребенка своего хозяина.

1. $\forall x\ Dog(x) \Rightarrow \neg Bites(x, Child(Owner(x)))$
2. $\neg\exists x, y\ Dog(x) \wedge Child(y, Owner(x)) \wedge Bites(x, y)$
3. $\forall x\ Dog(x) \Rightarrow (\forall y\ Child(y, Owner(x)) \Rightarrow \neg Bites(x, y))$
4. $\neg\exists x\ Dog(x) \Rightarrow (\exists y\ Child(y, Owner(x)) \wedge Bites(x, y))$

г) Каждый почтовый индекс в пределах штата имеет одну и ту же первую цифру.

1. $\forall x, s, z_1\ [State(s) \wedge LivesIn(x, s) \wedge Zip(x) = z_1] \Rightarrow [\forall y, z_2\ LivesIn(y, s) \wedge Zip(y) = z_2 \Rightarrow Digit(1, z_1) = Digit(1, z_2)]$
2. $\forall x, s\ [State(s) \wedge LivesIn(x, s) \wedge \exists z_1\ Zip(x) = z_1] \Rightarrow [\forall y, z_2\ LivesIn(y, s) \wedge Zip(y) = z_2 \wedge Digit(1, z_1) = Digit(1, z_2)]$

3. $\forall x, y, s \ State(s) \wedge LivesIn(x, s) \wedge LivesIn(y, s) \Rightarrow Digit(1, Zip(x) = Zip(y))$
4. $\forall x, y, s \ State(s) \wedge LivesIn(x, s) \wedge LivesIn(y, s) \Rightarrow$
$Digit(1, Zip(x)) = Digit(1, Zip(y))$

8.13. Выполните следующие задания в отношении приведенных логических высказываний.

а) Выполните перевод на нормальный, естественный язык (без "иксов" и "игреков!). (Здесь *SpeaksLanguage* — говорит на языке, а *Understands* — понимает язык.)
$\forall x, y, l \ SpeaksLanguage(x, l) \wedge SpeaksLanguage(y, l) \Rightarrow Understands(x, y) \wedge Understands(y, x)$

б) Объясните, почему данное высказывание имеет следствием указанное высказывание.
$\forall x, y, l \ SpeaksLanguage(x, l) \wedge SpeaksLanguage(y, l) \Rightarrow Understands(x, y)$

в) Переведите в высказывания логики первого порядка следующие предложения.
1. Понимание ведет к дружбе.
2. Дружба является переходящей.
Не забудьте определить все предикаты, функции и константы, которые будете использовать.

8.14. Приведенные ниже утверждения истинны или ложны? Поясните свой ответ.

а) $\exists x \ x = Rumpelstiltskin$ — это допустимое (обязательно истинное) высказывание логики первого порядка.

б) В логике первого порядка каждое высказывание с квантором существования истинно в любой модели, содержащей ровно один объект.

в) Высказывание $\forall x, y \ x = y$ выполнимо.

8.15. Перепишите первые две аксиомы Пеано из раздела 8.3.3 как одну аксиому, определяющую $NatNum(x)$ таким образом, чтобы исключалась возможность существования натуральных чисел, кроме тех, которые генерируются функцией-преемником.

8.16. Уравнение 8.4 в разделе 8.34 определяет условия, при которых в квадрате ощущается ветерок. Здесь мы рассмотрим два других способа описания этого аспекта мира вампуса.

а) Можно написать *диагностическое правило*, приводящее от наблюдаемых эффектов к их скрытым причинам. Очевидные диагностические правила для обнаружения ям гласят, что если в квадрате ощущается ветерок, то в каком-то соседнем квадрате должна быть яма, и наоборот, если ветерка в квадрате нет, то ни один из соседних квадратов не содержит ямы. Запишите эти два правила языком логики первого порядка и покажите, что их конъюнкция логически эквивалентна уравнению 8.4.

б) Можно написать "*причинное*" *правило*, ведущее от причины к следствию. Одно очевидное причинное правило состоит в том, что наличие в квадрате ямы вызывает ощущение ветерка во всех соседних с ним квадратах. Запишите это правило языком логики первого порядка, поясните, почему оно является неполным в сравнении с уравнением 8.4, и предоставьте недостающую аксиому.

8.17. Составьте аксиомы с описанием предикатов *GrandChild* (внук или внучка), *GreatGrandparent* (прадедушка или прабабушка), *Ancestor* (предок), *Brother* (брат), *Sister* (сестра), *Daughter* (дочь), *Son* (сын), *FirstCousin* (двоюродный брат или двоюродная сестра), *BrotherInLaw* (брат мужа или жены), *SisterInLaw* (сестра мужа или жены),

Aunt (тетя) и *Uncle* (дядя). Найдите правильное определение m-го кузена или кузины в n-м колене и запишите это определение в логике первого порядка. Теперь запишите основные факты, представленные в генеалогическом дереве, приведенном на рис. 8.7. Используя подходящую систему формирования логических рассуждений, введите в базу знаний с помощью операции TELL все записанные вами высказывания, а затем с помощью операции ASK определите, кто является внуками или внучками Элизабет (*Elizabeth*), братьями мужа Дианы (*Diana*), прадедушкой и прабабушкой Зары (*Zara*), а также предками Евгении (*Eugenie*).

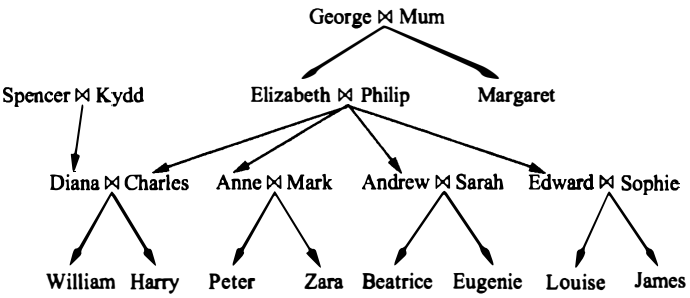


Рис. 8.7. Типичное генеалогическое дерево некоторой семьи. Символ ⋈ соединяет супругов, а стрелки указывают на детей

8.18. Запишите высказывание с утверждением, что функция + является коммутативной. Следует ли это высказывание из аксиом Пеано? В случае положительного ответа объясните, почему, а в случае отрицательного ответа приведите модель, в которой эти аксиомы являются истинными, а ваше высказывание — ложным.

8.19. Объясните, в чем ошибка в следующем предлагаемом определении предиката членства в множестве $\in$:

$$\forall x, s \; x \in \{x|s\} \; \forall x, s \; x \in s \Rightarrow \forall y \; x \in \{y|s\}.$$

8.20. Используя в качестве примеров аксиомы множества, запишите аксиомы для проблемной области списков, включая все константы, функции и предикаты, упомянутые в данной главе.

8.21. Объясните, в чем ошибка в следующем предлагаемом определении соседних квадратов в мире вампуса:

$$\forall x, y \; Adjacent([x, y], [x + 1, y]) \wedge Adjacent([x, y], [x, y + 1]).$$

8.22. Запишите аксиомы, необходимые для формирования рассуждений о местонахождении вампуса, с использованием символа константы *Wumpus* и бинарного предиката *At*(*Wumpus*, *Location*). Не забудьте, что существует только один вампус.

8.23. Предположим, что определены предикаты *Parent*(p, q) (*родитель*) и *Female*(p) (*женщина*), а также константы *Joan* и *Kevin* с очевидными значениями. Представьте каждое из следующих предложений в виде высказывания логики первого порядка. (Можете использовать сокращение $\exists^1$ для обозначения "существует ровно один".)

а) У Джоан есть дочь (возможно, больше одной, а возможно, и сыновья).

б) У Джоан есть одна дочь (но могут быть и сыновья).

в) У Джоан есть один ребенок, дочь.
г) У Джоан и Кевина есть один общий ребенок.
д) У Джоан есть по крайней мере один ребенок от Кевина и нет детей ни от кого другого.

8.24. Арифметические утверждения могут быть записаны в логике первого порядка с использованием символа предиката <, символов двух функций, + и ×, и двух символов констант, 0 и 1. При необходимости также могут быть определены дополнительные предикаты с двумя условиями.
а) Представьте в логике первого порядка свойство "x — четное число".
б) Представьте в логике первого порядка свойство "x — простое число".
в) В гипотезе Гольдбаха (пока недоказанной) утверждается, что каждое четное число равно сумме двух простых чисел. Представьте эту гипотезу как логическое высказывание.

8.25. В главе 6 мы использовали равенство, чтобы указать связь между переменной и ее значением. Например, мы написали $WA = red$, чтобы указать, что на карте Австралии территория штата Западная Австралия окрашена в красный цвет. Чтобы представить это в логике первого порядка, необходима более многословная запись: $ColorOf(WA) = red$. Какой неверный логический вывод может быть сделан, если записать это высказывание просто в виде логического утверждения $WA = red$?

8.26. В логике первого порядка напишите утверждение о том, что каждый ключ и по крайней мере один из каждой пары носков будут в конечном итоге потеряны навсегда, используя только следующий словарь: $Key(x)$, где x — ключ; $Sock(x)$, где x — носок; $Pair(x, y)$, где x и y — пара; Now — текущее время; $Before(t_1, t_2)$, где t_1 — момент времени, предшествующий моменту времени t_2; $Lost(x, t)$, — объект x был потерян в момент времени t.

8.27. Для каждого из приведенных ниже предложений на естественном языке определите, является ли предложенное высказывание логики первого порядка хорошим его переводом. Если нет, объясните, почему, и исправьте ошибку. (Некоторые высказывания могут содержать более одной ошибки!)
а) Нет двух людей с одним и тем же номером социального страхования:
$$\neg\exists\, x, y, n \;\; Person(x) \wedge Person(y) \;\Rightarrow\; [HasSS\#(x, n) \wedge HasSS\#(y, n)].$$
б) Номер социального страхования Джона такой же, как у Мэри:
$$\exists\, n \;\; HasSS\#(John, n) \wedge HasSS\#(Mary, n).$$
в) Каждый номер социального страхования имеет девять цифр:
$$\forall\, x, n \;\; Person(x) \;\Rightarrow\; [HasSS\#(x, n) \wedge Digits(n, 9)].$$
г) Перепишите каждое из приведенных выше (неисправленных) предложений, используя символ функции $\{SS\#\}$ вместо предиката $\{HasSS\#\}$.

8.28. Переведите в высказывание логики первого порядка фразу "ДНК каждого человека уникальна и получена из ДНК его родителей". Необходимо указать точное предполагаемое значение терминов словаря. (*Подсказка*: не используйте предикат $Unique(x)$, поскольку уникальность на самом деле не является свойством самого объекта!)

8.29. Для каждого из приведенных ниже предложений на естественном языке определите, является ли предложенное высказывание логики первого порядка хорошим его переводом. Если нет, объясните, почему, и исправьте ошибку.
а) За любую квартиру в Лондоне просят меньшую арендную плату, чем за некоторые квартиры в Париже:

$$\forall x\ [Apt(x) \wedge In(x, London)] \Rightarrow \exists y\ ([Apt(y) \wedge In(y, Paris)] \Rightarrow (Rent(x) < Rent(y))).$$

б) В Париже ровно одна квартира с арендой платой меньше 1000 долларов:

$$\exists x\ Apt(x) \wedge In(x, Paris) \wedge \forall y\ [Apt(y) \wedge In(y, Paris) \wedge (Rent(y) < Dollars(1000))] \Rightarrow (y = x).$$

в) Если квартира дороже всех квартир в Лондоне, она должна быть в Москве:

$$\forall x\ Apt(x) \wedge [\forall y\ Apt(y) \wedge In(y, London) \wedge (Rent(x) > Rent(y))] \Rightarrow In(x, Moscow).$$

8.30. Представьте следующие предложения в виде высказываний логики первого порядка, используя совместимый словарь (который вы должны определить).

а) Некоторые студенты выбрали изучение французского языка весной 2001 года.

б) Каждый студент, выбравший изучение французского языка, успешно сдает соответствующий экзамен.

в) Только один студент выбрал изучение греческого языка весной 2001 года.

г) Лучшая оценка по греческому всегда выше, чем лучшая оценка по французскому.

д) Каждый человек, который покупает страховой полис, умен.

е) Ни один человек не покупает дорогой страховой полис.

ж) Существует агент, который продает страховые полисы только тем людям, которые не застрахованы.

з) Есть парикмахер, который бреет всех мужчин в городе, которые не бреются сами.

и) Любой человек, рожденный в Великобритании, каждый из родителей которого является гражданином Великобритании или ее резидентом, является гражданином этой страны по рождению.

к) Любой человек, рожденный вне Великобритании, один из родителей которого является гражданином Великобритании по рождению, является гражданином этой страны по происхождению.

л) Политические деятели могут постоянно вводить некоторых людей в заблуждение, они также могут вводить в заблуждение всех людей на некоторое время, но не могут постоянно вводить в заблуждение всех людей.

м) Все греки говорят на одном языке. (Используйте запись $Speaks(x, l)$, чтобы указать, что человек x говорит на языке l.)

8.31. Представьте следующие предложения в виде высказываний логики первого порядка, используя совместимый словарь (который вы должны определить).

а) Некоторые студенты выбрали изучение французского языка весной 2000 года.

б) Каждый студент, выбравший изучение французского языка, успешно сдает соответствующий экзамен.

в) Только один студент выбрал изучение греческого языка весной 2009 года.

г) Лучшая оценка по греческому всегда ниже, чем лучшая оценка по французскому.

д) Каждый человек, который покупает страховой полис, умен.

е) Существует агент, который продает страховые полисы только тем людям, которые не застрахованы.

ж) Есть парикмахер, который бреет всех мужчин в городе, которые не бреются сами.

з) Любой человек, рожденный в Великобритании, каждый из родителей которого является гражданином Великобритании или ее резидентом, является гражданином этой страны по рождению.

и) Любой человек, рожденный вне Великобритании, один из родителей которого является гражданином Великобритании по рождению, является гражданином этой страны по происхождению.

к) Политические деятели могут постоянно вводить некоторых людей в заблуждение, они также могут вводить в заблуждение всех людей на некоторое время, но они не могут постоянно вводить в заблуждение всех людей.

л) Все греки говорят на одном языке. (Используйте запись *Speaks*(x, l), чтобы указать, что человек x говорит на языке l.)

8.32. Напишите общее множество фактов и аксиом, необходимых для представления утверждения "Веллингтон слышал о смерти Наполеона" и выдачи правильного ответа на запрос "Слышал ли Наполеон о смерти Веллингтона?"

8.33. Расширьте словарь, приведенный в разделе 8.4, с целью определения правил сложения для n-битовых двоичных чисел. Затем сформулируйте описание четырехбитового сумматора, представленного на рис. 8.8, и составьте запросы, необходимые для проверки того, что это описание действительно правильно.

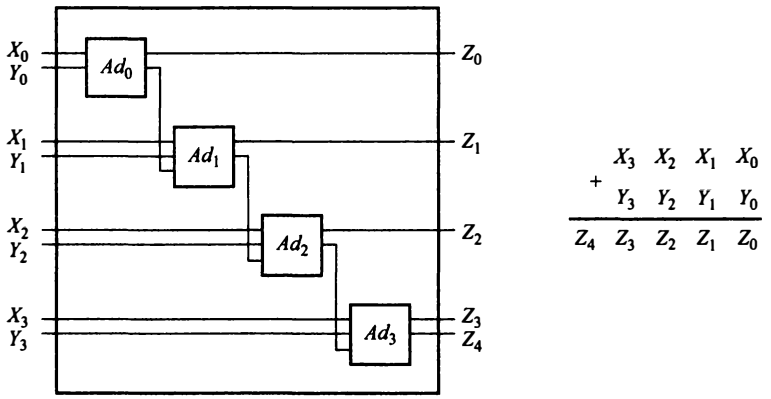


Рис. 8.8. Четырехбитовый сумматор. Каждый элемент Ad_i является однобитовым сумматором, представленным на рис. 4.6.

8.34. Представление электронной схемы, приведенное в этой главе, является более подробным, чем это необходимо, если нас интересуют только функциональные возможности этой схемы. В более простой формулировке рассматриваются любые логические элементы или цифровые схемы с m входами и n выходами с использованием предиката, имеющего $m+n$ параметров, такого, что предикат принимает истинное значение тогда и только тогда, когда входы и выходы согласованы. Например, логические элементы NOT могут быть описаны с помощью бинарного предиката *NOT*(i, o), для которого известны значения *NOT*(0, 1) и *NOT*(1, 0). Композиции логических элементов определяются с помощью конъюнкций предикатов логических элементов, в которых наличие разделяемых переменных указывает на прямые соединения. Например, схема NAND может состоять из элементов *AND* и *NOT*:

$$\forall\, i_1, i_2, o_a, o \;\; AND(i_1, i_2, o_a) \wedge NOT(o_a, o) \;\Rightarrow\; NAND(i_1, i_2, o).$$

Используя это представление, определите однобитовый сумматор, приведенный на рис. 8.6, и четырехбитовый сумматор, приведенный на рис. 8.8, и объясните, какими запросами вы бы воспользовались для проверки этих проектов. Какого рода запросы, поддерживаемые представлением, описанным в разделе 8.4, не поддерживаются данным представлением?

8.35. Получите бланк заявки на оформление паспорта гражданина своей страны, выясните, какие правила определяют права человека на получение паспорта, и переведите эти правила на язык логики первого порядка в соответствии с этапами, описанными в разделе 8.4.

8.36. Рассмотрите базу знаний, представленную в логике первого порядка, описывающую миры, содержащие людей, песни, альбомы (например, "Meet the Beatles") и диски (т.е. конкретные физические экземпляры компакт-дисков). Ее словарь содержит следующие символы.

CopyOf(*d*, *a*): предикат. Диск *d* является копией альбома *a*.

Owns(*p*, *d*): предикат. Человек *p* владеет диском *d*.

Sings (*p*, *s*, *a*): альбом *a* включает запись песни *s* , исполняемой человеком *p*.

Wrote(*p*, *s*): человек *p* написал песню *s*.

McCartney, *Gershwin*, *BHoliday*, *Joe*, *EleanorRigby*, *TheManILove*, *Revolver*: константы с очевидным значением.

Выразите следующие утверждения в логике первого порядка.

а) Гершвин написал песню "The Man I Love".

б) Гершвин не написал песню "Eleanor Rigby".

в) Либо Гершвин, либо Маккартни написали песню "The Man I Love".

г) Джо написал хотя бы одну песню.

д) Джо владеет копией альбома "Revolver".

ж) Каждая песня, которую Маккартни исполняет в альбоме "Revolver", была написана Маккартни.

з) Гершвин не написал ни одной из песен в альбоме "Revolver".

и) Каждая песня, которую написал Гершвин, записана в каком-то альбоме. (Возможно, разные песни записаны в разных альбомах.)

к) Есть один альбом, который содержит каждую песню, написанную Джо.

л) Джо владеет копией альбома, в котором песню "The Man I Love" исполняет Билли Холидей.

м) Джо владеет копией каждого альбома, в котором есть песня в исполнении Маккартни. (Безусловно, каждый отдельный альбом записан на собственном физическом компакт-диске.)

н) Джо владеет копией каждого альбома, на котором все песни поет Билли Холидей.

ГЛАВА 9

Логический вывод в логике первого порядка

В этой главе определяются эффективные процедуры получения ответов на вопросы, сформулированные в логике первого порядка.

В данной главе рассматриваются алгоритмы, позволяющие получить ответ на любой вопрос, сформулированный в логике первого порядка, для которого ответ существует. В разделе 9.1 представлены правила логического вывода для кванторов и показано, как можно свести вывод в логике первого порядка к выводу в пропозициональной логике, хотя и за счет значительных издержек. В разделе 9.2 описывается, как **унификацию** можно использовать для формирования правил логического вывода, применяемых непосредственно к высказываниям логики первого порядка. Далее рассматриваются три основных семейства алгоритмов вывода в логике первого порядка: **прямой логический вывод** (раздел 9.3), **обратный логический вывод** (раздел 9.4) и **доказательство теорем на основе резолюции** (раздел 9.5).

9.1. Логический вывод в логике высказываний и логике первого порядка

Одним из способов логического вывода в логике первого порядка является преобразование высказываний, хранящихся в базе знаний логики первого порядка, в высказывания логики высказываний и применения к ним пропозиционального логического вывода, — о том, как это делается, говорилось в предыдущих главах. Первым шагом на этом пути является устранение универсальных кванторов. Например, предположим, что база знаний содержит следующую стандартную аксиому, отражающую известную из многих сказок мысль о том, что все жадные короли — злые:

$$\forall x \; King(x) \wedge Greedy(x) \Rightarrow Evil(x).$$

Из этой аксиомы можно вывести любое из следующих высказываний:

$$King(John) \wedge Greedy(John) \Rightarrow Evil(John),$$
$$King(Richard) \wedge Greedy(Richard) \Rightarrow Evil(Richard),$$
$$King(Father(John)) \wedge Greedy(Father(John)) \Rightarrow Evil(Father(John)).$$
$$\vdots$$

В общем случае, согласно правилу ▶**конкретизации высказывания квантора всеобщности** (или **UI** — *Universal Instantiation*), можно логическим путем вывести любое высказывание, получаемое в результате подстановки **базового терма** (т.е. терма *без* переменных) вместо переменной в кванторе всеобщности.[1]

Чтобы формально записать это правило логического вывода, воспользуемся понятием **подстановки**, введенным в разделе 8.3. Пусть SUBST(θ, α) обозначает результат применения подстановки терма θ к высказыванию α. В таком случае данное правило для любой переменной v и базового терма g можно записать следующим образом:

$$\frac{\forall v\ \alpha}{\text{SUBST}(\{v/g\},\ \alpha)}.$$

Например, три высказывания, приведенные выше, получены с помощью подстановок $\{x/John\}$, $\{x/Richard\}$ и $\{x/Father(John)\}$.

В соответствии с правилом ▶**конкретизации высказывания квантора существования** (или **EI** — *Existential Instantiation*) переменная в кванторе существования заменяется одним *новым символом константы*. Формально это высказывание определяется следующим образом: для любого высказывания α, переменной v и символа константы k, который не появляется где-либо в базе знаний, имеет место следующее:

$$\frac{\exists v\ \alpha}{\text{SUBST}(\{v/k\},\ \alpha)}.$$

Например, из высказывания

$$\exists x\ Crown(x) \wedge OnHead(x, John)$$

можно вывести высказывание

$$Crown(C_1) \wedge OnHead(C_1, John)$$

при условии, что символ константы C_1 не появляется где-либо в базе знаний. По сути, в первом высказывании с квантором существования утверждается, что существует некоторый объект, удовлетворяющий определенному условию, а в результате конкретизации этому объекту просто присваивается имя. Естественно, что это

[1] Эти подстановки не следует путать с расширенными интерпретациями, которые использовались для определения семантики кванторов в разделе 8.2.6. В подстановке переменная *заменяется* термом (синтаксической конструкцией) для получения нового высказывания, тогда как любая интерпретация *отображает* некоторую переменную на объект в проблемной области.

имя не должно уже принадлежать другому объекту. В математике есть прекрасный пример: предположим, мы открыли, что имеется некоторое число, которое немного больше, чем 2,71828 и которое удовлетворяет уравнению $d(x^y)/dy = x^y$ для x. Этому числу можно присвоить новое имя, такое как e, но было бы ошибкой присваивать ему имя существующего объекта, допустим, π. В логике такое новое имя называется ►**сколемовской константой**.

В то время как конкретизация высказывания с квантором всеобщности может выполняться многократно с целью получения многих разных заключений, конкретизацию высказывания с квантором существования можно выполнить только один раз, а затем это высказывание с квантором существования следует просто отбросить. Например, исходное высказывание $\exists x\ Kill(x, Victim)$ становится уже ненужным после добавления в базу знаний высказывания $Kill(Murderer, Victim)$. (В переводе с английского *kill* — убить, *victim* — жертва, *murderer* — убийца.)

9.1.1. Приведение к пропозициональному логическому выводу

Теперь покажем, как любая база знаний логики первого порядка может быть преобразована в базу знаний логики высказываний. Первая идея состоит в следующем: по аналогии с тем, как высказывание с квантором существования может быть заменено одной конкретизацией, высказывание с квантором всеобщности может быть заменено множеством всех возможных конкретизаций. Например, предположим, что исходная база знаний содержит только следующие четыре высказывания

$$\begin{aligned}&\forall x\ King(x) \wedge Greedy(x) \Rightarrow Evil(x)\\ &King(John)\\ &Greedy(John)\\ &Brother(Richard, John)\end{aligned} \tag{9.1}$$

и что единственными объектами в ней являются *John* и *Richard*. Применим правило конкретизации высказывания с квантором всеобщности к первому высказыванию, используя все возможные подстановки базовых термов из словаря этой базы знаний, в данном случае — $\{x/John\}$ и $\{x/Richard\}$. Будут получены следующие высказывания:

$$\begin{aligned}&King(John) \wedge Greedy(John) \Rightarrow Evil(John),\\ &King(Richard) \wedge Greedy(Richard) \Rightarrow Evil(Richard).\end{aligned}$$

А теперь отбросим высказывание с квантором всеобщности и заменим базовые атомарные высказывания, такие как $King(John)$, пропозициональными символами, такими как *JohnIsKing*. Наконец, применим любой из алгоритмов полного пропозиционального вывода из главы 7 для получения таких заключений, как *JohnIsEvil*, являющееся эквивалентным высказыванию $Evil(John)$.

Как показано в разделе 9.5, такой метод ►**пропозиционализации** (преобразования в высказывания логики высказываний) может стать полностью

обобщенным. Однако здесь имеет место серьезная проблема: если база знаний логики первого порядка включает символ функции, то множество возможных подстановок базовых термов становится бесконечным! Например, если в базе знаний упоминается символ *Father*, то существует возможность сформировать бесконечно большое количество вложенных термов, таких как *Father*(*Father*(*Father*(*John*))).

К счастью, существует знаменитая теорема, предложенная Жаком Эрбраном ([1013], 1930), согласно которой, если некоторое высказывание следует из первоначальной базы знаний в логике первого порядка, то существует доказательство, которое включает лишь *конечное* подмножество этой пропозиционализированной базы знаний. Поскольку любое такое подмножество имеет максимальную глубину вложения среди его базовых термов, это подмножество можно найти, формируя вначале все конкретизации с символами констант (*Richard* и *John*), затем — все термы с глубиной 1 (*Father*(*Richard*) и *Father*(*John*)), после этого — все термы с глубиной 2 и так до тех пор, пока есть возможность составить пропозициональное доказательство высказывания, выведенного из базы знаний.

Выше был кратко описан один из подходов к организации вывода в логике первого порядка с помощью пропозиционализации, который является **полным**, т.е. позволяет доказать любое высказывание, которое следует из базы знаний. Это — важное достижение, если учесть, что пространство возможных моделей является бесконечным. С другой стороны, до тех пор, пока это доказательство не составлено, мы не знаем, *следует* ли данное высказывание из базы знаний! Что произойдет, если это высказывание из нее *не следует*? Можем ли мы это определить? Как оказалось, для логики первого порядка это действительно невозможно. Наша процедура доказательства может продолжаться и продолжаться, вырабатывая все более и более глубоко вложенные термы, а мы не будем знать, вошла ли она в безнадежный цикл или до получения доказательства остался только один шаг. Такая проблема весьма напоминает проблему останова машин Тьюринга. Алан Тьюринг ([2233], 1936) и Алонсо Черч ([431], 1936) доказали неизбежность такого состояния дел, хотя и весьма различными способами. ➔ *Вопрос о следствии для логики первого порядка является* ***полуразрешимым****; это означает, что существуют алгоритмы, позволяющие найти доказательство для любого высказывания, которое следует из базы знаний, но нет алгоритмов, позволяющих также определить, что не существует доказательства для каждого высказывания, которое не следует из базы знаний.*

9.2. Унификация и логический вывод в логике первого порядка

Внимательный читатель должен был заметить, что подход на основе пропозиционализации порождает множество ненужных конкретизаций высказываний с кванторами всеобщности и, следовательно, является довольно неэффективным. Удобнее было бы иметь подход, в котором используется только одно

правило, — следуя рассуждению, что подстановка $\{x/John\}$ позволяет получить ответ на запрос $Evil(x)$ следующим образом: приняв во внимание правило, что жадные короли являются злыми, найти некоторый x, такой, что x — король и x — жадный, а затем вывести, что x — злой. В более общем случае это можно сформулировать так: если существует некоторая подстановка θ, делающая каждый из конъюнктов предпосылки импликации идентичным высказываниям, которые уже находятся в базе знаний, то можно утверждать об истинности заключения этой импликации после применения θ. В данном случае такой цели достигает подстановка $\{x/John\}$. Теперь предположим, что нам известно не то, что жаден Джон — $Greedy(John)$, а что жадными являются все:

$$\forall y \ Greedy(y). \tag{9.2}$$

Но и в этом случае нам все равно хотелось бы иметь возможность получить заключение, что Джон зол, $Evil(John)$, поскольку нам известно, что Джон — король (это дано) и Джон жаден (так как жадными являются все). Для того чтобы подобный метод мог работать, нужно найти подстановку как для переменных в высказывании с импликацией, так и для переменных в высказываниях, которые имеются в базе данных. В данном случае в результате применения подстановки $\{x/John, y/John\}$ к предпосылкам импликации $King(x)$ и $Greedy(x)$, а также к высказываниям в базе знаний $King(John)$ и $Greedy(y)$ эти высказывания становятся идентичными. Таким образом, теперь можно вывести заключение импликации.

Такой процесс логического вывода может быть представлен с помощью единственного правила логического вывода, которое будем называть ▶**обобщенным правилом отделения** (*Generalized Modus Ponens*):[2] для атомарных высказываний p_i, p_i' и q, если существует подстановка θ, такая, что $\text{SUBST}(\theta, p_i') = \text{SUBST}(\theta, p_i)$, то для всех i имеет место следующее:

$$\frac{p_1', \ p_2', \ \ldots, \ p_n', \ (p_1 \wedge p_2 \wedge \ldots \wedge p_n \Rightarrow q)}{\text{SUBST}(\theta, q)}.$$

В этом правиле имеется $n+1$ предпосылка: n атомарных высказываний p_i' и одна импликация. Заключение становится результатом применения подстановки к следствию q. В данном примере имеет место следующее.

p_1' — это $King(John)$
p_2' — это $Greedy(y)$
θ — это $\{x/John, y/John\}$
$\text{SUBST}(\theta, q)$ — это $Evil(John)$

p_1 — это $King(x)$
p_2 — это $Greedy(x)$
q — это $Evil(x)$

[2] Обобщенное правило отделения является более общим, чем правило *Modus Ponens* (см. раздел 7.5.1), в том смысле, что известные факты и предпосылки импликации должны соответствовать только в пределах подстановки, а не точно. С другой стороны, само правило Modus Ponens в качестве предпосылки допускает любое предложение α, а не только конъюнкцию атомарных высказываний.

Можно легко показать, что обобщенное правило отделения — непротиворечивое правило логического вывода. Прежде всего отметим, что для любого высказывания p (в отношении которого предполагается, что на его переменные распространяется квантор всеобщности) и для любой подстановки θ справедливо следующее правило:

$$p \models \text{SUBST}(\theta, p).$$

Оно выполняется по правилу конкретизации высказывания с квантором всеобщности, в частности в любой подстановке θ, которая удовлетворяет условиям обобщенного правила отделения. Поэтому из $p_1', \ldots, p_n'$ можно вывести следующее:

$$\text{SUBST}(\theta, p_1') \wedge \ldots \wedge \text{SUBST}(\theta, p_n'),$$

а из импликации $p_1 \wedge \ldots, \wedge p_n \Rightarrow q$ можно вывести

$$\text{SUBST}(\theta, p_1) \wedge \ldots \wedge \text{SUBST}(\theta, p_n) \Rightarrow \text{SUBST}(\theta, q).$$

Теперь подстановка θ в обобщенном правиле отделения определена так, что $\text{SUBST}(\theta, p_i') = \text{SUBST}(\theta, p_i)$ для всех i, следовательно, первое из этих двух высказываний точно совпадает с предпосылкой второго высказывания. Таким образом, выражение $\text{SUBST}(\theta, p)$ следует из правила отделения.

Обобщенное правило отделения принято называть ►**поднятой** версией правила отделения — оно поднимает правило отделения от базового уровня (не имеющего переменных) логики высказываний до уровня логики первого порядка. В оставшейся части этой главы будет показано, что могут быть разработаны поднятые версии представленных в главе 7 алгоритмов прямого логического вывода, обратного логического вывода и резолюции. Основным преимуществом применения поднятых правил логического вывода по сравнению с пропозиционализацией является то, что в них предусмотрены только те подстановки, которые требуются для обеспечения дальнейшего выполнения конкретных логических выводов.

9.2.1. Унификация

Применение поднятых правил логического вывода связано с необходимостью поиска подстановок, в результате которых различные логические выражения становятся идентичными. Этот процесс называется ►**унификацией** и является ключевым компонентом любых алгоритмов вывода в логике первого порядка. Алгоритм UNIFY принимает на входе два высказывания и возвращает для них ►**унификатор**, если таковой существует:

$$\text{UNIFY}(p, q) = \theta, \quad \text{где} \quad \text{SUBST}(\theta, p) = \text{SUBST}(\theta, q).$$

Давайте рассмотрим несколько примеров того, как должен действовать алгоритм UNIFY. Предположим, имеется запрос *AskVars*(*Knows*(*John*, *x*)) — кого знает Джон? Некоторые ответы на этот запрос можно найти, отыскивая все выска-

зывания в базе знаний, которые унифицируются с высказыванием *Knows*(*John*, *x*). Ниже приведены результаты унификации с четырьмя различными высказываниями, которые могут находиться в базе знаний:

UNIFY(*Knows*(*John*, *x*), *Knows*(*John*, *Jane*)) = {*x*/*Jane*},
UNIFY(*Knows*(*John*, *x*), *Knows*(*y*, *Bill*)) = {*x*/*Bill*, *y*/*John*},
UNIFY(*Knows*(*John*, *x*), *Knows*(*y*, *Mother*(*y*))) = {*y*/*John*, *x*/*Mother*(*John*)},
UNIFY(*Knows*(*John*, *x*), *Knows*(*x*, *Elizabeth*)) = *failure*.

Последняя попытка унификации заканчивается неудачей (*failure*), поскольку переменная *x* не может одновременно принимать значения *John* и *Elizabeth*. Теперь вспомним, что высказывание *Knows*(*x*, *Elizabeth*) означает "Все знают Элизабет", поэтому мы *обязаны* иметь возможность вывести логически, что Джон знает Элизабет. Проблема возникает только потому, что в этих двух высказываниях, как оказалось, используется одно и то же имя переменной, *x*. Возникновения этой проблемы можно избежать, ▶**стандартизируя отличие** (*standardizing apart*) одного из этих двух унифицируемых высказываний. Это означает использование в высказываниях переименования переменных для предотвращения коллизий имен. Например, переменную *x* в высказывании *Knows*(*x*, *Elizabeth*) можно переименовать в x_{17} (новое имя переменной), не меняя смысл этого высказывания. После этого унификация выполняется успешно:

UNIFY(*Knows*(*John*, *x*), *Knows*(x_{17}, *Elizabeth*)) = {*x*/*Elizabeth*, x_{17}/*John*}.

С дополнительными сведениями о том, с чем еще связана необходимость в стандартизации отличия, речь пойдет в упражнении 9.15.

Возникает еще одна сложность: выше было сказано, что алгоритм UNIFY должен возвращать такую подстановку (или унификатор), в результате которой два параметра становятся одинаковыми. Но количество таких унификаторов может быть больше одного. Например, при следующем вызове алгоритма UNIFY(*Knows*(*John*, *x*), *Knows*(*y*, *z*)) может быть возвращено {*y*/*John*, *x*/*z*} или {*y*/*John*, *x*/*John*, *z*/*John*}. Первый унификатор позволяет получить в качестве результата унификации выражение *Knows*(*John*, *z*), а второй дает *Knows*(*John*, *John*). Но второй результат может быть получен из первого с помощью дополнительной подстановки {*z*/*John*}; в таком случае принято считать, что первый унификатор является *более общим* по сравнению со вторым, поскольку налагает меньше ограничений на значения переменных.

Для любой унифицируемой пары выражений существует единственный ▶**наиболее общий унификатор** (*Most General Unifier* — **MGU**), который является уникальным вплоть до переименования и подстановки переменных. Например, эквивалентными полагаются подстановки {*x*/*John*} и {*y*/*John*}, так же как {*x*/*John*, *y*/*John*} и {*x*/*John*, *y*/*x*}.

Алгоритм вычисления наиболее общих унификаторов приведен на рис. 9.1. Процесс его работы очень прост: рекурсивно исследовать два выражения одно-

временно, "бок о бок", наряду с этим формируя унификатор, но фиксировать ситуацию неудачного завершения, если две соответствующие точки в полученных таким образом структурах не совпадают. В этом процессе существует один дорогостоящий этап: если переменная согласуется со сложным термом, необходимо провести проверку того, встречается ли сама эта переменная внутри терма; в случае положительного ответа на данный вопрос согласование заканчивается неудачей, поскольку невозможно сформировать какой-либо совместимый унификатор. Например, $S(x)$ невозможно унифицировать с $S(S(x))$. Из-за этой так называемой ► **проверки вхождения** (*occur check*) сложность всего алгоритма становится квадратично зависимой от размера унифицируемых выражений. В одних системах, включая многие системы логического программирования, такая проверка вхождения просто исключается и обязанность исключения противоречивых логических выводов из результатов возлагается на пользователя. В других системах используются более сложные алгоритмы со сложностью, линейно зависящей от времени.

---

function UNIFY($x, y, \theta = empty$) **returns** подстановка, позволяющая сделать x и y идентичными, или индикатор отказа *failure*
  if $\theta = failure$ **then return** *failure*
  else if $x = y$ **then return** θ
  else if VARIABLE?(x) **then return** UNIFY-VAR(x, y, θ)
  else if VARIABLE?(y) **then return** UNIFY-VAR(y, x, θ)
  else if COMPOUND?(x) **and** COMPOUND?(y) **then**
    return UNIFY(ARGS(x), ARGS(y), UNIFY(OP(x), OP(y), θ))
  else if LIST?(x) **and** LIST?(y) **then**
    return UNIFY(REST(x), REST(y), UNIFY(FIRST(x), FIRST(y), θ))
  else return *failure*

function UNIFY-VAR(var, x, θ) **returns** подстановка
  if $\{var/val\} \in \theta$ для некоторого *val* **then return** UNIFY(*val*, x, θ)
  else if $\{x/val\} \in \theta$ для некоторого *val* **then return** UNIFY(*var*, *val*, θ)
  else if OCCUR-CHECK?(*var*, x) **then return** *failure*
  else return подстановка θ с добавленной к ней парой $\{var/x\}$

Рис. 9.1. Алгоритм унификации. Аргументы x и y могут быть любыми выражениями: константой или переменной, составным выражением, таким как сложное высказывание или терм, либо списком выражений. Аргумент θ представляет собой подстановку, исходно пустую, в которую добавляются пары $\{var/val\}$ в процессе рекурсивного просмотра входных параметров и их поэлементного сравнения. В составном выражении, таком как $F(A, B)$, функция OP(x) выбирает функциональный символ F, а функция ARGS выбирает список параметров (A, B)

9.2.2. Хранение и выборка

В основе функций TELL, ASK и ASKVARS, применяемых для ввода информации и передачи запросов в базу знаний, лежат более примитивные функции STORE и FETCH. Функция STORE(s) сохраняет высказывание s в базе знаний, а функция FETCH(q) возвращает все унификаторы, такие, что запрос q унифицируется с некоторым высказыванием из базы знаний. Описанная выше задача, предназначенная для иллюстрации процесса унификации — поиск всех фактов, которые унифицируются с высказыванием *Knows*(*John*, x)), — представляет собой пример применения функции FETCH.

Самый простой способ реализации функций STORE и FETCH предполагает хранение всех фактов базы знаний в виде одного длинного списка и унифицирование каждого запроса с каждым элементом этого списка. Такой метод является неэффективным, но он работает. В оставшейся части данного раздела описаны способы, позволяющие обеспечить более эффективную выборку.

Функцию FETCH можно сделать более эффективной, обеспечив, чтобы попытки унификации применялись только к тем высказываниям, которые имеют *определенный* шанс на унификацию. Например, нет смысла пытаться унифицировать *Knows*(*John*, x) и *Brother*(*Richard*, *John*). Такой унификации можно избежать, ►**индексируя** факты в базе знаний. Самая простая схема, называемая ►**индексацией по предикатам**, предусматривает размещение всех фактов *Knows* в одном сегменте, а всех фактов *Brother* — в другом. Сами сегменты для повышения эффективности доступа можно хранить в хеш-таблице.

Индексация по предикатам будет удобна, если имеется очень много предикатных символов, но лишь небольшое количество выражений в расчете на каждый символ. Однако иногда на некоторый предикатный символ приходится очень много выражений. Например, предположим, что налоговые органы желают следить за тем, кто кого нанимает, с использованием предиката *Employs*(x, y). Такой сегмент будет очень большим и может включать миллионы нанимателей и десятки миллионов наемных работников. В этом случае при использовании индексации по предикатам для поиска ответа на запрос *Employs*(x, *Richard*), т.е. "Кто является нанимателем Ричарда?", потребовался бы просмотр всего сегмента.

Для данного конкретного запроса поиск будет проще при использовании индексации фактов и по предикату, и по второму параметру, возможно, с использованием комбинированного ключа хеш-таблицы. В таком случае существовала бы возможность просто формировать ключ из запроса и осуществлять выборку именно тех фактов, которые унифицируются с этим запросом. В свою очередь, для поиска ответа на другие запросы, такие как *Employs*(*IBM*, y), было бы полезно проиндексировать факты, комбинируя предикат с первым параметром. Следовательно, факты имеет смысл хранить под разными индексными ключами, что позволит моментально сделать их доступными для разных запросов, с которыми они могли бы унифицироваться.

Имея некоторое высказывание, подлежащее хранению, можно построить индексы для *всех возможных* запросов, которые с ним унифицируются. Так, для факта *Employs*(*IBM*, *Richard*) такими запросами будут следующие.

Employs(*IBM*, *Richard*)	Корпорация IBM является нанимателем Ричарда?
Employs(*x*, *Richard*)	Кто является нанимателем Ричарда?
Employs(*IBM*, *y*)	Для кого корпорация IBM является нанимателем?
Employs(*x*, *y*)	Кто для кого является нанимателем?

Эти запросы образуют ▶**решетку обобщения**, представленную на рис. 9.2, *а*. Подобные решетки обладают некоторыми интересными свойствами. Так, дочерний узел любого узла в этой решетке получен из его родительского узла с помощью единственной подстановки, а "наивысший" общий потомок любых двух узлов является результатом применения наиболее общего унификатора для этих узлов. Высказывание с повторяющимися константами имеет несколько иную решетку, показанную на рис. 9.2, *б*. Хотя на этом рисунке функциональные символы не показаны, они также могут быть включены в структуру решетки.

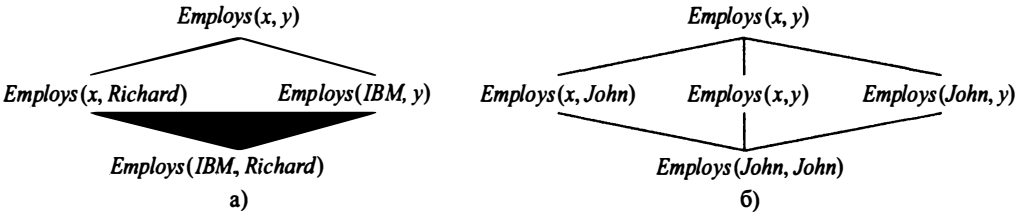


Рис. 9.2. а) Решетка обобщения, самым нижним узлом которой является высказывание *Employs*(*IBM*, *Richard*). **б)** Решетка обобщения для высказывания *Employs*(*John*, *John*)

Для предикатов с небольшим числом аргументов хорошим компромиссом будет создание индекса для каждого узла в решетке обобщения. Такой подход требует немного больших затрат времени при сохранении данных, но существенно ускоряет их поиск. Однако для предиката с n аргументами решетка содержит $O(2^n)$ узлов. Если будут разрешены символы функций, число узлов решетки будет также экспоненциально зависимым от размера термов в высказывании, подлежащем сохранению. Все это может потребовать создания огромного количества индексов.

Необходимо каким-то образом ограничить набор создаваемых индексов лишь теми, которые могут часто использоваться в запросах. В противном случае больше времени будет тратиться на создание и поддержку индексов, чем экономиться за счет их использования. Можно принять какую-то строгую политику, например создавать индексы только для ключей, состоящих из предиката плюс один аргумент. Или можно разработать некую адаптивную политику, согласно которой создание

индексов предусматривается лишь в соответствии с потребностями в поиске ответов на запросы тех типов, которые встречаются чаще всего. Для коммерческих баз данных, количество фактов в которых исчисляется миллиардами, эта проблема является предметом интенсивного изучения, продуктивных технологических разработок и постоянной оптимизации.

9.3. Прямой логический вывод

В разделе 7.5 был приведен алгоритм прямого логического вывода для баз данных из пропозициональных определенных выражений. В данном разделе эта идея расширяется с целью охвата определенных выражений в логике первого порядка.

Безусловно, существуют некоторые логические высказывания, которые не могут быть сформулированы как определенные выражения, а следовательно, их нельзя будет обрабатывать при данном подходе. Однако правил в форме *Посылка* $\Rightarrow$ *Следствие* будет достаточно для успешной реализации широкого диапазона интересных реальных систем.

9.3.1. Определенные выражения в логике первого порядка

Определенные выражения в логике первого порядка представляют собой дизъюнкции литералов, среди которых *положительным является один и только один*. Это означает, что определенное выражение либо является атомарным, либо представляет собой импликацию, посылкой (антецедентом) которой служит конъюнкция положительных литералов, а следствием (консеквентом) — единственный положительный литерал. Кванторы существования не допускаются, а кванторы всеобщности неявно присутствуют слева: если в определенном выражении присутствует x, то это означает неявное наличие в нем квантора $\forall x$. Типичное определенное выражение в логике первого порядка выглядят следующим образом:

$$King(x) \wedge Greedy(x) \Rightarrow Evil(x),$$

но литералы $King(John)$ и $Greedy(y)$ также можно рассматривать как определенные выражения. Литералы логики первого порядка могут включать переменные, поэтому литерал $Greedy(y)$ интерпретируется как "все являются жадными" (неявный квантор всеобщности).

Рассмотрим использование определенных выражений на примере следующей задачи.

> Закон гласит, что продажа оружия недружественным странам, осуществляемая любым американским гражданином, является преступлением. В государстве Ноуноу, враждебном по отношению к Америке, имеются некоторые ракеты, и все ракеты этого государства были проданы ему полковником Вестом, являющимся американским гражданином.

Требуется доказать, что полковник Вест совершил преступление. Прежде всего, представим все имеющиеся факты в виде определенных выражений логики первого порядка.

— "*продажа оружия недружественным странам, осуществляемая любым американским гражданином, является преступлением*":

$$American(x) \wedge Weapon(y) \wedge Sells(x, y, z) \wedge Hostile(z) \Rightarrow Criminal(x). \quad (9.3)$$

— "*В государстве Ноуноу... имеются некоторые ракеты*". Соответствующее высказывание $\exists x \; Owns(Nono, x) \wedge Missile(x)$ преобразуется в два определенных выражения посредством устранения квантора существования с введением новой константы M_1:

$$Owns(Nono, M_1), \quad (9.4)$$

$$Missile(M_1). \quad (9.5)$$

— "*все ракеты этого государства были проданы ему полковником Вестом*":

$$Missile(x) \wedge Owns(Nono, x) \Rightarrow Sells(West, x, Nono). \quad (9.6)$$

Нам необходимо также знать, что ракеты — это оружие:

$$Missile(x) \Rightarrow Weapon(x). \quad (9.7)$$

Кроме того, нам необходимо знать, что государство, враждебное по отношению к Америке, рассматривается как "недружественное":

$$Enemy(x, America) \Rightarrow Hostile(x). \quad (9.8)$$

— "*полковником Вестом, являющимся американским гражданином*":

$$American(West). \quad (9.9)$$

— "*В государстве Ноуноу, враждебном по отношению к Америке*":

$$Enemy(Nono, America). \quad (9.10)$$

Эта база знаний может служить примером баз знаний на языке ▶**Datalog**: она состоит из определенных выражений логики первого порядка, не содержащих функциональных символов. Язык Datalog получил свое название благодаря тому, что позволяет удобно представлять тот тип операторов, которые обычно применяются в реляционных базах данных. При отсутствии функциональных символов логический вывод значительно упрощается.

9.3.2. Простой алгоритм прямого логического вывода

На рис. 9.3 представлен очень простой алгоритм прямого логического вывода. Начиная с известных фактов, он активизирует все правила, предпосылки которых выполняются, и добавляет их заключения к известным фактам. Этот процесс

продолжается до тех пор, пока не будет найден ответ на запрос (при условии, что требуется только один ответ) или добавления новых фактов больше не происходит. Следует отметить, что факт не является "новым", если он представляет собой ▶**переименование** известного факта: одно высказывание является переименованием другого, если оба эти высказывания идентичны во всем, кроме имен переменных. Например, высказывания *Likes*(x, *IceCream*) и *Likes*(y, *IceCream*) по отношению друг к другу представляют собой переименования, поскольку оба имеют один и тот же смысл — "все любят мороженое".

---

function FOL-FC-ASK(*KB*, α) **returns** подстановка или значение *false*
  inputs: *KB*, база знаний — множество определенных выражений логики первого порядка
    α, запрос — атомарное высказывание

  while *true* **do**
    new ← {} // Множество новых высказываний, логически
      // выведенных на каждой итерации
    for each *rule* **in** *KB* **do**
      $(p_1 \wedge \ldots \wedge p_n \Rightarrow q) \leftarrow$ STANDARDIZE-VARIABLES(*rule*)
      for each подстановки θ, такой, что SUBST(θ, $p_1 \wedge \ldots \wedge p_n$) =
          SUBST(θ, $p'_1 \wedge \ldots \wedge p'_n$) для некоторых $p'_1, \ldots, p'_n$
          в базе знаний *KB*
        $q' \leftarrow$ SUBST(θ, q)
        if выражение q' не является переименованием некоторого
          высказывания, которое уже находится в базе знаний *KB*
          или является элементом множества *new* **then**
            добавить q' к множеству *new*
            $\phi \leftarrow$ UNIFY(q', α)
            if значение ϕ не представляет собой *failure* **then return** ϕ
    if *new* = {} **then return** *false*
    добавить множество *new* к базе знаний *KB*

Рис. 9.3. Концептуально простой, но неэффективный алгоритм прямого логического вывода. На каждой итерации к базе знаний KB добавляются все атомарные высказывания, которые могут быть выведены за один этап из тех импликационных и атомарных высказываний, которые уже находятся в базе знаний. Функция STANDARDIZE-VARIABLES заменяет все переменные в своих аргументах новыми, которые еще не использовались ранее

Для иллюстрации работы алгоритма FOL-FC-ASK воспользуемся описанной выше задачей доказательства преступления. Импликационными высказываниями являются высказывания, приведенные в уравнениях 9.3, 9.6–9.8. Требуются следующие две итерации.

- На первой итерации правило 9.3 имеет невыполненные предпосылки. Правило 9.6 выполняется с подстановкой $\{x/M_1\}$ и добавляется высказывание *Sells*(*West*, M_1, *Nono*). Правило 9.7 выполняется с подстановкой $\{x/M_1\}$ и добавляется высказывание *Weapon*(M_1). Правило 9.8 выполняется с подстановкой $\{x/Nono\}$ и добавляется высказывание *Hostile*(*Nono*).
- На второй итерации правило 9.3 выполняется с подстановкой $\{x/West, y/M_1, z/Nono\}$ и добавляется высказывание *Criminal*(*West*).

На рис. 9.4 приведено сформированное дерево доказательства. Обратите внимание, что в этот момент невозможны какие-либо новые логические выводы, поскольку каждое высказывание, заключение которого можно было бы найти с помощью прямого логического вывода, уже явно содержится в базе знаний *KB*. Такое состояние базы знаний называется ▶**фиксированной точкой** (*fixed point*) в процессе логического вывода. Фиксированные точки, достигаемые при прямом логическом выводе с использованием определенных выражений логики первого порядка, аналогичны фиксированным точкам, возникающим при пропозициональном прямом логическом выводе (см. раздел 7.5.4); основное различие состоит в том, что фиксированная точка в логике первого порядка может включать атомарные высказывания с квантором всеобщности.

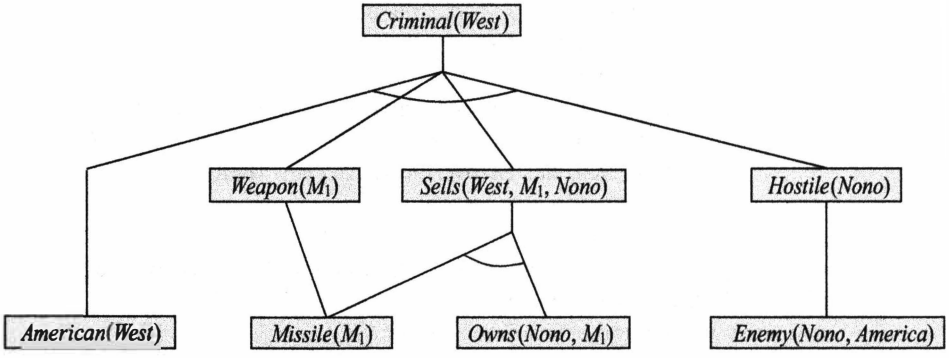


Рис. 9.4. Дерево доказательства, сформированное путем прямого логического вывода для примера с доказательством совершения преступления. Первоначальные факты показаны на нижнем уровне, факты, выведенные логическим путем в первой итерации, — на среднем уровне, а факт, логически выведенный во второй итерации, — на верхнем уровне

Свойства алгоритма FOL-FC-Ask проанализировать несложно. Во-первых, он является **непротиворечивым**, поскольку каждый этап логического вывода представляет собой применение обобщенного правила отделения, которое само является непротиворечивым. Во-вторых, он является **полным** применительно к базам знаний с определенными выражениями, — это означает, что он позволяет ответить

на любой запрос, ответы на который следуют из базы знаний с определенными выражениями.

Для баз знаний на языке Datalog, не содержащих функциональных символов, доказательство полноты является довольно простым. Начнем с подсчета количества возможных фактов, которые могут быть добавлены, определяющего максимальное количество итераций. Допустим, что k — максимальная **арность** (количество параметров) любого предиката, p — количество предикатов и n — количество символов констант. Очевидно, что может быть не больше чем pn^k различных базовых фактов, поэтому алгоритм должен достичь фиксированной точки именно после стольких итераций. В таком случае можно применить обоснование приведенного выше утверждения, аналогичное доказательству полноты пропозиционального прямого логического вывода (см. раздел 7.5.4). Подробные сведения о том, как осуществить переход от пропозициональной полноты к полноте логики первого порядка, приведены применительно к алгоритму резолюции в разделе 9.5.

При применении к более общим определенным выражениям с функциональными символами алгоритм FOL-FC-Ask может вырабатывать бесконечно большое количество новых фактов, поэтому требуется соблюдать осторожность. Для того случая, в котором из базы знаний следует ответ на высказывание запроса q, необходимо прибегать к использованию теоремы Эрбрана (раздел 9.1.1) для обеспечения того, чтобы алгоритм мог найти доказательство (случай, касающийся резолюции, описан в разделе 9.5). А если запрос не имеет ответа, то в некоторых случаях может оказаться, что не удается нормально завершить работу данного алгоритма. Например, если база знаний включает аксиомы Пеано

$$NatNum(0)$$
$$\forall n \; NatNum(n) \Rightarrow NatNum(S(n)),$$

то в результате прямого логического вывода в нее будут добавлены факты $NatNum(S(0))$, $NatNum(S(S(0)))$, $NatNum(S(S(S(0))))$ и т.д. Вообще говоря, избежать этой проблемы невозможно. Как и в общем случае логики первого порядка, задача выполнения логического вывода с использованием определенных выражений является полуразрешимой.

9.3.3. Эффективный прямой логический вывод

Алгоритм прямого логического вывода, приведенный на рис. 9.3, был спроектирован с целью упрощения понимания основной идеи метода, а не обеспечения его эффективной работы. Существуют три возможных источника неэффективности. Во-первых, внутренний цикл этого алгоритма пытается сопоставить все правила с каждым фактом в базе данных. Во-вторых, алгоритм предусматривает повторную проверку каждого правила на каждой итерации, даже если в базу знаний было внесено очень мало дополнений. В-третьих, этот алгоритм может вырабатывать множество фактов, которые не имеют отношения к текущей цели. Устраним каждый из этих источников неэффективности по очереди.

Согласование правил с известными фактами

Проблема согласования предпосылки правила с фактами, хранящимися в базе знаний, может показаться достаточно простой. Например, предположим, что требуется применить следующее правило:

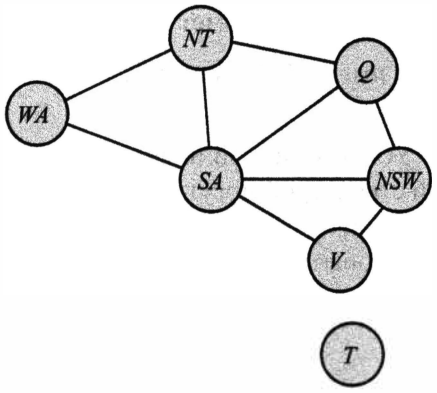
$$Missile(x) \Rightarrow Weapon(x).$$

Для этого необходимо найти все факты, которые согласуются с выражением *Missile*(x). В базе знаний, проиндексированной необходимым образом, это можно выполнить за постоянное время в расчете на каждый факт. А теперь рассмотрим правило, подобное следующему:

$$Missile(x) \wedge Owns(Nono, x) \Rightarrow Sells(West, x, Nono).$$

И вновь, найти все объекты, принадлежащие государству Ноуноу, можно за постоянное время в расчете на каждый объект, а затем можно применить к каждому объекту проверку, является ли он ракетой. Однако если в базе знаний содержится много сведений об объектах, принадлежащих государству Ноуноу, и лишь немного сведений о ракетах, то лучше было бы сначала найти все ракеты, а затем проверить, какие из них принадлежат Ноуноу. Это — проблема ▶**упорядочения конъюнктов**: отыскание упорядочения, позволяющего решать конъюнкты в предпосылке правила таким образом, чтобы общая стоимость решения была минимальной. Как оказалось, задача поиска оптимального упорядочения является NP-трудной, но для нее имеются хорошие эвристики. Например, эвристика **с минимальным количеством оставшихся значений** (MRV) , применявшаяся при решении задач УО в главе 6, подсказывает, что необходимо упорядочить конъюнкты так, чтобы вначале проводился поиск ракет, если количество ракет меньше по сравнению с количеством всех известных объектов, принадлежащих государству Ноуноу.

Между процедурами ▶**согласования с шаблоном** и удовлетворения ограничений действительно существует очень тесная связь. Каждый конъюнкт может рассматриваться как ограничение на содержащиеся в нем переменные; например, *Missile*(x) — это унарное ограничение на x. Развивая эту идею, можно прийти к выводу, что ➔ *существует возможность представить любую задачу УО с конечной областью определения как единственное определенное выражение наряду с некоторыми касающимися ее базовыми фактами.* Рассмотрим приведенную на рис. 6.1 задачу раскраски карты Австралии, которая вновь приведена на рис. 9.5, *а*. Эквивалентная формулировка в виде одного определенного выражения представлена на рис. 9.5, *б*. Очевидно, что заключение *Colorable*() можно вывести из этой базы знаний, только если данная задача УО имеет решение. А поскольку задачи УО в целом включают задачи SAT (выполнимости булевых формул) как частный случай, можно сделать вывод, что ➔ *задача согласования определенного выражения с множеством фактов является NP-трудной.*



а)

$Diff(wa, nt) \wedge Diff(wa, sa) \wedge$
$Diff(nt, q) \wedge Diff(nt, sa) \wedge$
$Diff(q, nsw) \wedge Diff(q, sa) \wedge$
$Diff(nsw, v) \wedge Diff(nsw, sa) \wedge$
$Diff(v, sa) \Rightarrow Colorable()$
$Diff(Red, Blue)$ $Diff(Red, Green)$
$Diff(Green, Red)$ $Diff(Green, Blue)$
$Diff(Blue, Red)$ $Diff(Blue, Green)$

б)

Рис. 9.5. а) Граф ограничений для задачи раскрашивания карты Австралии. **б)** Задача УО раскрашивания карты Австралии, представленная одним определенным выражением. Каждый регион карты представлен переменной, значением которой может быть одна из констант *Red*, *Green* и *Blue* (которые объявлены как *Diff*)

Тот факт, что во внутреннем цикле алгоритма прямого логического вывода необходимо решать NP-трудную задачу согласования, может показаться удручающим. Однако существует три способа, позволяющих улучшить эту ситуацию.

- Напомним, что большинство правил в базах знаний, применяемых на практике, являются небольшими и простыми (подобно правилам, используемым в примере доказательства преступления), а не большими и сложными (как в формулировке задачи УО, приведенной на рис. 9.5). В сообществе пользователей баз данных принято считать, что размеры правил и арности предикатов не превышают некоторого постоянного значения, и принимать во внимание требуется только ►**сложность данных**, т.е. сложность логического вывода как функции от количества базовых фактов в базе данных. Можно легко показать, что в этом смысле сложность данных при прямом логическом выводе определяется полиномиальной зависимостью.
- Можно рассматривать подклассы правил, для которых согласование является наиболее эффективным. По сути, каждое выражение на языке Datalog может рассматриваться как определяющее некоторую задачу УО, поэтому согласование будет осуществимым только тогда, когда соответствующая задача УО будет разрешимой. В главе 6 было описано несколько семейств разрешимых задач УО. Например, если граф ограничений (граф, узлами которого являются переменные, а дугами — ограничения) образует дерево, то эта задача УО может быть решена за линейное время. Точно такой же результат остается в силе в отношении согласования с правилами. Например,

если из графа, приведенного на рис. 9.5, *а*, удалить узел *SA*, относящийся к Южной Австралии, то результирующее выражение примет следующий вид:

$$Diff(wa, nt) \wedge Diff(nt, q) \wedge Diff(q, nsw) \wedge Diff(nsw, v) \Rightarrow Colorable(),$$

что соответствует сокращенной задаче УО, показанной на рис. 6.12. Алгоритмы решения задач УО с древовидной структурой могут непосредственно применяться для решения задачи согласования с правилами.

- Можно попытаться исключить излишние попытки согласования с правилами в алгоритме прямого логического вывода, что и является темой следующего подраздела.

Инкрементный прямой логический вывод

В предыдущем разделе, при демонстрации работы прямого логического вывода на примере доказательства преступления, имело место небольшое упрощение. В частности, были опущены некоторые из согласований с правилами, выполненные алгоритмом, приведенным на рис. 9.3. Например, на второй итерации правило

$$Missile(x) \Rightarrow Weapon(x)$$

согласуется с фактом $Missile(M_1)$ (еще раз), и, безусловно, при этом ничего не происходит, поскольку заключение $Weapon(M_1)$ уже известно. Таких излишних согласований с правилами можно избежать, сделав следующее наблюдение: ➡ *каждый новый факт, выведенный на итерации t, должен быть получен по меньшей мере из одного нового факта, выведенного на итерации* $t-1$. Это наблюдение соответствует истине, поскольку любой логический вывод, который не требовал нового факта из итерации $t-1$, уже мог быть выполнен на итерации $t-1$.

Это наблюдение естественным образом приводит к созданию алгоритма инкрементного прямого логического вывода, в котором на итерации t проверка правила происходит только тогда, когда его предпосылка включает конъюнкт p_i, который унифицируется с фактом p_i', вновь выведенным на итерации $t-1$. Затем на этапе согласования с правилом значение p_i фиксируется для согласования с p_i', но при этом допускается, чтобы остальные конъюнкты в правиле согласовывались с фактами из любой предыдущей итерации. Этот алгоритм в каждой итерации вырабатывает точно такие же факты, как и алгоритм, приведенный на рис. 9.3, но является гораздо более эффективным.

При использовании подходящей индексации можно легко выявить все правила, которые могут быть активизированы любым конкретным фактом. Многие реальные системы действуют в режиме "обновления", при котором прямой логический вывод происходит в ответ на каждый новый факт, сообщенный системе с помощью операции TELL. Операции логического вывода каскадно распространяются через множество правил до тех пор, пока не достигается фиксированная точка, а затем этот процесс выполняется вновь и вновь по мере поступления каждого нового факта.

Как правило, в действительности лишь небольшая доля правил в базе знаний активизируется в результате добавления определенного факта. Это означает, что при многократном построении частичных согласований с правилами, имеющими некоторые невыполненные предпосылки, выполняется значительный объем ненужной работы. Рассматриваемый здесь пример доказательства преступления слишком мал, чтобы на нем можно было наглядно показать такую ситуацию, однако обратите внимание, что на первой итерации конструируется частичное согласование между правилом

$$American(x) \wedge Weapon(y) \wedge Sells(x, y, z) \wedge Hostile(z) \Rightarrow Criminal(x)$$

и фактом *American(West)*. Затем это частичное согласование отбрасывается и снова формируется во второй итерации (в которой данное правило согласуется успешно). Было бы уместно сохранять и постепенно дополнять частичные согласования по мере поступления новых фактов, а не отбрасывать их.

В ▶ **алгоритме rete**[3] впервые была предпринята попытка решить эту проблему. Этот алгоритм предусматривает предварительную обработку множества правил в базе знаний для формирования своего рода сети потока данных, в которой каждый узел представляет собой литерал из предпосылки какого-либо правила. Операции связывания переменных продвигаются по этой сети и пропускаются после того, как выполнить согласование с каким-то литералом не удается. Если в двух литералах некоторого правила совместно используется какая-то переменная (например, $Sells(x, y, z) \wedge Hostile(z)$ в примере доказательства преступления), то варианты связывания из каждого литерала пропускаются через узел проверки равенства. Процессу связывания переменных, достигших узла n-арного литерала, такого как $Sells(x, y, z)$, может потребоваться перейти в состояние ожидания, пока будут определены связывания для других переменных, прежде чем этот процесс сможет продолжиться. В любой конкретный момент времени состояние rete-сети охватывает все частичные согласования с правилами, что позволяет избежать большого объема повторных вычислений.

Не только сами rete-сети, но и различные их усовершенствования стали ключевым компонентом так называемых ▶ **продукционных систем**, принадлежащих к числу самых первых систем прямого логического вывода, получивших широкое распространение.[4] В частности, с использованием архитектуры продукционной системы была создана система XCON (ее первоначальное название — R1; МакДермотт [1546], 1982). Система XCON содержала несколько тысяч правил и предназначалась для проектирования конфигураций компьютерных компонентов для заказчиков корпорации DEC. Ее создание было одним из первых очевидно успешных

[3] Здесь *rete* — латинское слово, которое переводится как "сеть" и читается по-русски как "рете", а по-английски — как "рити".

[4] Слово **продукция** в названии **продукционная система** обозначает правило "условие-действие".

коммерческих проектов в развивающейся области экспертных систем. На основе той же базовой технологии, реализованной на языке общего назначения OPS-5, позднее было создано много других подобных систем.

Продукционные системы также широко применяются в ▶**когнитивных архитектурах** (т.е. в моделях человеческого мышления), таких как ACT (Андерсон [48], 1983) и SOAR (Лейрд и др. [1339], 1987). В подобных системах "рабочая память" системы моделирует кратковременную память человека, а продукции образуют часть долговременной памяти. В каждом цикле функционирования происходит согласование продукций с фактами из рабочей памяти. Продукции, условия которых выполнены, могут добавлять или удалять факты в рабочей памяти. В отличие от типичных ситуаций с большим объемом данных, наблюдаемых в базах данных, продукционные системы часто содержат много правил и относительно немного фактов. При использовании технологии согласования, оптимизированной должным образом, некоторые современные системы могут оперировать в реальном времени больше чем миллионом правил.

Не относящиеся к делу факты

Еще один источник неэффективности прямого логического вывода состоит в выполнении всех допустимых этапов логического вывода на основе всех известных фактов, *даже если они не относятся к рассматриваемой цели*. В примере доказательства преступления не было правил, способных приводить к заключениям, не относящимся к делу. Но если бы в базе знаний присутствовало несколько правил с описанием кулинарных предпочтений американцев или компонентов и цен на ракеты, алгоритм FOL-FC-ASK сгенерировал бы много не относящихся к делу заключений.

Один из способов предотвращения формирования нерелевантных заключений состоит в использовании обратного логического вывода, обсуждаемого в разделе 9.4. Другое решение состоит в ограничении прямого логического вывода избранным подмножеством правил, как в алгоритме PL-FC-ENTAILS? (см. раздел 7.5.4). Третий подход был разработан в области ▶**дедуктивных баз данных**, представляющих собой крупномасштабные базы данных, такие как реляционные БД, но в которых стандартным инструментальным средством являются не SQL-запросы, а прямой логический вывод. Идея состоит в том, чтобы перезаписывать множество правил с использованием информации из цели так, что в процессе прямого логического вывода рассматриваются только релевантные связывания переменных, принадлежащие к так называемому ▶**магическому множеству**. Например, если целью является $Criminal(West)$, то правило, приводящее к заключению $Criminal(x)$, может быть перезаписано для включения дополнительного конъюнкта, ограничивающего значение x:

$$Magic(x) \wedge American(x) \wedge Weapon(y) \wedge Sells(x, y, z) \wedge Hostile(z) \Rightarrow Criminal(x).$$

В базу знаний также добавляется факт *Magic*(*West*). В результате, даже если база знаний содержит факты о миллионах американцев, в процессе прямого логического вывода будет рассматриваться только полковник Вест. Полный процесс определения магических множеств и перезаписи базы знаний слишком сложен для того, чтобы его описание было приведено в этой главе, но основная идея состоит в том, что выполняется своего рода "общий" обратный логический вывод от цели для выяснения того, какие связывания переменных нужно будет ограничивать. Поэтому подход с использованием магических множеств может рассматриваться как гибридный между прямым логическим выводом и обратной предварительной обработкой.

9.4. Обратный логический вывод

Во втором главном семействе алгоритмов логического вывода используется подход с **обратным логическим выводом** на множестве определенных выражений. Эти алгоритмы работают в обратном направлении, от цели, проходя по цепочке от одного правила к другому, чтобы найти известные факты, поддерживающие доказательство.

9.4.1. Алгоритм обратного логического вывода

На рис. 9.6 приведен алгоритм обратного логического вывода для определенных выражений. Запрос FOL-BC-Ask(*KB*, *goal*) будет выполнен, если база знаний содержит правило вида *lhs* $\Rightarrow$ *goal*, где *lhs* (от *left-hand side* — левая сторона) представляет собой список конъюнктов. Атомарный факт, такой как *American*(*West*), рассматривается как выражение, для которого *lhs* является пустым списком. Теперь запрос, содержащий переменные, может быть унифицирован несколькими способами. Например, запрос *Person*(x) может быть выполнен подстановкой {x/*John*}, а также подстановкой {x/*Richard*}. Поэтому алгоритм FOL-BC-Ask реализован как генератор — функция, которая возвращает значения несколько раз, каждый раз предоставляя один возможный результат (см. приложение B).

Обратный логический вывод является своего рода поиском И/ИЛИ — здесь ИЛИ присутствует потому, что целевой запрос может быть выполнен любым правилом в базе знаний, а И — потому, что все конъюнкты в части *lhs* выражения должны быть выполнены. Функция FOL-BC-Or работает, выбирая все выражения, которые могут быть унифицированы с целью, стандартизируя переменные в выражении совершенно новыми переменными, а затем, если часть *rhs* выражения действительно унифицируется с целью, выполняет каждый конъюнкт в части *lhs* с использованием функции FOL-BC-And. Эта функция работает, по очереди подтверждая каждый из конъюнктов и отслеживания накопленные подстановки по ходу выполнения. На рис. 9.7 показано дерево доказательства для получения факта *Criminal*(*West*) из высказываний, приведенных в уравнениях 9.3–9.10.

```
function FOL-BC-ASK(KB, query) returns генератор подстановок
  return FOL-BC-OR(KB, query, {})

function FOL-BC-OR(KB, goal, θ) returns подстановка
  for each rule in FETCH-RULES-FOR-GOAL(KB, goal) do
    (lhs ⇒ rhs) ← STANDARDIZE-VARIABLES(rule)
    for each θ′ in FOL-BC-AND(KB, lhs, UNIFY(rhs, goal, θ)) do
      yield θ′

function FOL-BC-AND(KB, goals, θ) returns подстановка
  if θ = failure then return
  else if LENGTH(goals) = 0 then yield θ
  else
    first, rest ← FIRST(goals), REST(goals)
    for each θ′ in FOL-BC-OR(KB, SUBST(θ, first), θ) do
      for each θ″ in FOL-BC-AND(KB, rest, θ′) do
        yield θ″
```

Рис. 9.6. Простой алгоритм обратного логического вывода для баз знаний логики первого порядка

Алгоритм обратного логического вывода в том виде, в каком он был приведен в этом разделе, безусловно, представляет собой алгоритм поиска в глубину. Это означает, что его потребности в пространстве линейно зависят от размера доказательства. Также это означает, что обратный логический вывод (в отличие от прямого логического вывода) страдает от проблем, обусловленных наличием повторяющихся состояний и неполноты. Тем не менее, несмотря на эти ограничения, обратный логический вывод оказался популярным и доказал свою эффективность в языках логического программирования.

9.4.2. Логическое программирование

Логическое программирование — это технология, позволяющая довольно близко подойти к воплощению декларативного идеала, описанного в главе 7: системы должны конструироваться путем представления знаний на некотором формальном языке, а задачи решаться путем применения процессов логического вывода к этим знаниям. Такой идеал выражен в следующем уравнении Роберта Ковальски:

Алгоритм = Логика + Управление.

Наиболее широко распространенным языком логического программирования является язык ▶**Prolog**. Он применяется в основном в качестве языка быстрой разработки прототипов, а также для решения задач, связанных с манипуляцией символами, таких как написание компиляторов (Ван Рой [2260], 1990) и

синтаксический анализ текстов на естественном языке (Перейра и Уоррен [1777], 1980). На языке Prolog было написано много экспертных систем для юридических, медицинских, финансовых и других проблемных областей.

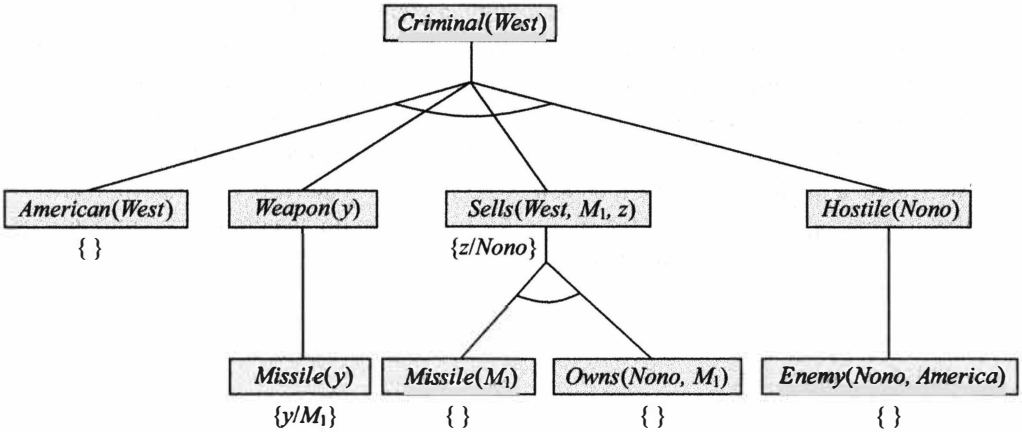


Рис. 9.7. Дерево доказательства, сформированное путем обратного логического вывода для доказательства того, что полковник Вест является преступником. Это дерево следует читать в глубину, слева направо. Чтобы доказать факт *Criminal(West)*, необходимо доказать четыре конъюнкта, находящихся под ним. Одни из них находятся в базе знаний, а другие требуют дальнейшего обратного логического вывода. Связывания для каждой успешной унификации показаны после соответствующей подцели. Обратите внимание, что после успешного достижения одной подцели в конъюнкции ее подстановка применяется для последующих подцелей. Таким образом, к тому времени, когда алгоритм FOL-BC-ASK достигает последнего конъюнкта, первоначально имевшего форму *Hostile(z)*, переменная *z* уже будет связана с *Nono*

Программы на языке Prolog представляют собой множества определенных выражений, записанных в системе обозначений, немного отличающейся от используемой в стандартной логике первого порядка. В языке Prolog прописные буквы применяются для обозначения переменных, а строчные — для обозначения констант, как раз наоборот по отношению к нотации, принятой в этой книге. Запятые разделяют конъюнкты в выражениях, а сами выражения записываются "задом наперед" по отношению к тому, как они записывались выше: вместо $A \wedge B \Rightarrow C$ на языке Prolog записывается `C :- A, B`. Ниже показан типичный пример:

```
criminal(X) :- american(X), weapon(Y), sells(X,Y,Z), hostile(Z).
```

В языке Prolog запись `[E|L]` обозначает список, первый элемент которого — `E`, а остальная часть обозначена как `L`. Вот программа на языке Prolog для предиката `append(X,Y,Z)`, которая выполняется успешно, если список `Z` является результатом добавления списка `X` в конец списка `Y`.

```
append([],Y,Y).
append([A|X],Y,[A|Z]) :- append(X,Y,Z).
```

На естественном языке эти выражения можно прочитать так: во-первых, дополнение списка `Y` пустым списком приводит к получению того же списка `Y`, и, во-вторых, `[A|Z]` — это результат дополнения списка `Y` списком `[A|X]`, при условии, что `Z` — это результат дополнения списка `Y` списком `X`. В большинстве языков высокого уровня можно написать похожую рекурсивную функцию, определяющую, как добавить один список в конец другого. Однако определение в языке Prolog на самом деле является более мощным, поскольку описывает *отношение* между тремя аргументами, а не *функцию*, вычисляемую из двух аргументов. Например, в систему можно ввести запрос `append(X,Y,[1,2,3])`, т.е. какие два списка можно дополнить один другим, чтобы получить список `[1,2,3]`? Система Prolog возвратит следующие решения.

```
X=[]      Y=[1,2,3];
X=[1]     Y=[2,3];
X=[1,2]   Y=[3];
X=[1,2,3] Y=[]
```

Выполнение программ на языке Prolog осуществляется по принципу обратного логического вывода с поиском в глубину, при котором попытка применения выражений выполняется в том порядке, в каком они записаны в базу знаний. Конструктивное решение, заложенное в основу системы Prolog, представляет собой компромисс между декларативностью и эффективностью работы. В результате некоторые аспекты языка Prolog выходят за рамки стандартного логического вывода.

- В языке Prolog используется семантика баз данных (см. раздел 8.2.8), а не семантика логики первого порядка, и это вполне очевидно из его трактовки равенства и отрицания (см. раздел 9.4.4).
- В нем предусмотрено множество встроенных функций для выполнения арифметических операций. Литералы, в которых используются соответствующие функциональные символы, "доказываются" путем выполнения кода, а не осуществления дальнейшего логического вывода. Например, цель "`X is 4+3`" успешно достигается после связывания переменной `X` со значением 7. С другой стороны, попытка достижения цели "`5 is X+Y`" заканчивается неудачей, поскольку эти встроенные функции не обеспечивают решения произвольных уравнений.
- В языке Prolog предусмотрены встроенные предикаты, вызывающие при их выполнении побочные эффекты. К ним относятся предикаты ввода-вывода и предикаты `assert/retract` для модификации базы знаний. Такие предикаты не имеют аналогов в логике и могут порождать некоторые эффекты, вызывающие путаницу, например, если факты подтверждаются (и вводятся в базу

знаний) некоторой ветвью дерева доказательства, которая в конечном итоге заканчивается неудачей.

- Из алгоритма унификации языка Prolog исключена **проверка вхождения**. Это означает, что могут быть сделаны некоторые противоречивые логические выводы, однако на практике эта проблема почти никогда не возникает.
- В системе Prolog используется обратный логический вывод с поиском в глубину без проверок на бесконечную рекурсию. Так сделано для удобства использования языка программирования, который работает очень быстро при правильном использовании. Но это также означает, что некоторые программы, которые выглядят как вполне допустимое логическое построение, никогда не закончат своего выполнения.

9.4.3. Избыточный логический вывод и бесконечные циклы

Теперь обратимся к ахиллесовой пяте языка Prolog: несогласованности между поиском в глубину и деревьями поиска, включающими повторяющиеся состояния и бесконечные пути. Рассмотрим следующую логическую программу, позволяющую определить, существует ли путь между двумя точками в ориентированном графе:

```
path(X,Z) :- link(X,Z).
path(X,Z) :- path(X,Y), link(Y,Z).
```

На рис. 9.8, *а* приведен простой граф, состоящий из трех узлов, который описан с помощью фактов *link*(*a*, *b*) и *link*(*b*, *c*). При использовании этой программы запрос *path*(*a*, *c*) вырабатывает дерево доказательства, показанное на рис. 9.9, *а*. С другой стороны, если эти два выражения расположены в порядке

```
path(X,Z) :- path(X,Y), link(Y,Z).
path(X,Z) :- link(X,Z).
```

то система Prolog следует по бесконечному пути, как показано на рис. 9.9, *б*. Поэтому система Prolog является неполной в качестве программы автоматического доказательства теорем для определенных выражений (как показано в этом примере, даже применительно к программам, соответствующим формату языка Datalog), поскольку для некоторых баз знаний эта система оказывается неспособной доказать высказывания, которые из них следуют. Необходимо отметить, что алгоритм прямого логического вывода не подвержен этой проблеме: сразу после вывода фактов `path(a,b)`, `path(b,c)` и `path(a,c)` процесс прямого логического вывода останавливается.

Обратный логический вывод с поиском в глубину сталкивается также с проблемами, обусловленными излишними вычислениями. Например, при поиске пути от узла A_1 к узлу J_4 на рис. 9.8, *б* система Prolog выполняет 877 этапов логического вывода, большинство из которых связано с поиском всех возможных путей к

узлам, не позволяющим достичь цели. Такая проблема аналогична проблеме повторяющихся состояний, которая описывалась в главе 3. Общее количество этапов логического вывода может определяться экспоненциальной зависимостью от количества базовых фактов, вырабатываемых в процессе вывода. А если бы вместо этого применялся прямой логический вывод, то можно было бы ограничиться выработкой, самое большее, n^2 фактов `path(X,Y)`, связывающих n узлов. Так, для решения задачи, приведенной на рис. 9.8, *б*, потребовалось бы всего 62 этапа логического вывода.

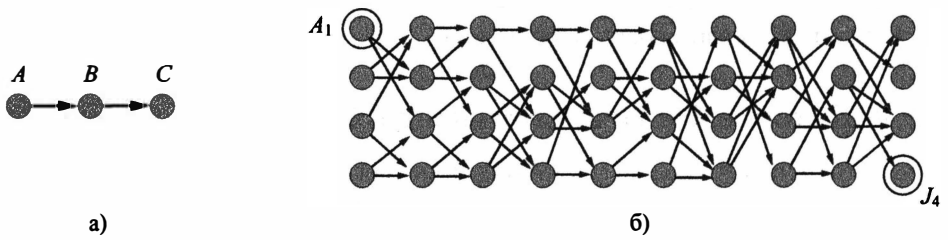


Рис. 9.8. а) Поиск пути от A до C может привести систему Prolog к выполнению бесконечного цикла. **б)** Граф, в котором каждый узел связан с двумя случайно выбираемыми преемниками на следующем уровне. На отыскание пути от A_1 до J_4 требуется 877 этапов логического вывода

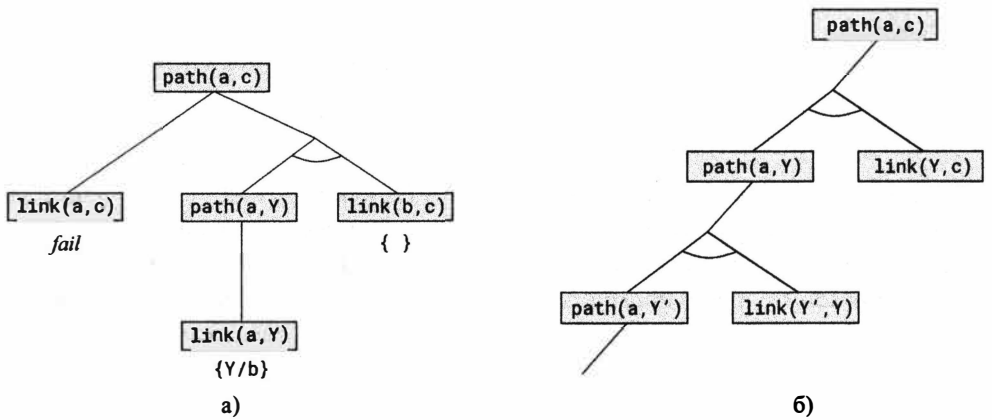


Рис. 9.9. а) Доказательство того, что путь от A до C существует. **б)** Бесконечное дерево доказательства, формируемое из-за того, что выражения находятся в "неправильном" порядке

Применение прямого логического вывода при решении задач поиска в графе представляет собой пример ▶**динамического программирования**, в котором решения подзадач формируются инкрементно из решений меньших подзадач и кешируются для предотвращения повторного вычисления. Аналогичный эффект в системе обратного логического вывода может быть достигнут за счет разбиения больших целей на меньшие по размеру вместо их укрупнения.

В любом случае сохранение промежуточных результатов во избежание дублирования является ключевым фактором. Этот подход применяется в системах ▶**табулированного логического программирования** (*tabled logic programming*), использующих эффективные механизмы сохранения и выборки. В табулированном логическом программировании объединяется целенаправленность обратного логического вывода с эффективностью динамического программирования, присущей прямому логическому выводу. Кроме того, эти системы являются *полными* применительно к базам знаний в формате Datalog, а это означает, что программисту приходится меньше беспокоиться о бесконечных циклах. (Все еще остается возможность попасть в бесконечный цикл в случае предикатов типа `father(X,Y)`, которые ссылаются на потенциально неограниченное количество объектов.)

9.4.4. Семантика базы данных языка Prolog

В языке Prolog семантика базы данных используется в том виде, в каком она описана в разделе 8.2.8. В программе на языке Prolog допущение уникальности имен требует, чтобы каждая константа и каждый базовый терм ссылались на отдельный объект, а допущение замкнутости мира определяет, что истинными могут быть только те высказывания, которые следуют из базы знаний. В языке Prolog нет возможности утверждать, что высказывание является ложным. Это делает его менее выразительным, чем логика первого порядка, но данное решение — часть того, что делает Prolog более эффективным и более лаконичным. Рассмотрим следующие утверждения о нескольких предлагаемых учебных курсах:

$$Course(CS, 101), Course(CS, 102), Course(CS, 106), Course(EE, 101). \quad (9.11)$$

В соответствии с допущением уникальности имен имена *CS* и *EE* различаются (как и имена 101, 102, 106), а это означает, что существует четыре различных курса. Из допущения замкнутости мира следует, что никаких других курсов нет, а значит, существует ровно четыре учебных курса. Но если рассматривать эти утверждения, исходя из семантики логики первого порядка, а не семантики базы данных, то все, что можно было бы сказать, — это что имеется приблизительно от одного до бесконечности курсов. Причина в том, что эти утверждения (в логике первого порядка) не отрицают возможность того, что в действительности предлагаются и другие, не упомянутые здесь, курсы, а также не указывают на то, что эти упомянутые курсы отличаются друг от друга. Если адекватно переписать

уравнение (9.11) с использованием семантики логики первого порядка, то получится следующее высказывание:

$$Course(d, n) \Leftrightarrow (d = CS \wedge n = 101) \vee (d = CS \wedge n = 102) \vee (d = CS \wedge n = 106) \vee (d = EE \wedge n = 101). \quad (9.12)$$

Этот результат называется ▶**завершением** уравнения (9.11). В логике первого порядка он выражает идею, что существует максимум четыре курса. Чтобы выразить в этой логике идею, что есть как минимум четыре курса, необходимо записать завершение предиката равенства:

$$x = y \Leftrightarrow (x = CS \wedge y = CS) \vee (x = EE \wedge y = EE) \vee (x = 101 \wedge y = 101) \vee (x = 102 \wedge y = 102) \vee (x = 106 \wedge y = 106).$$

Завершение полезно для понимания семантики баз данных, но не для практических целей. Если решаемая задача может быть описана с использованием семантики базы данных, эффективнее будет выполнять рассуждения с помощью системы Prolog или какой-либо другой системы, использующей семантику базы данных, а не переводить высказывания на язык логики первого порядка и использовать для выполнения рассуждений соответствующий решатель.

9.4.5. Логическое программирование в ограничениях

Выше, при обсуждении прямого логического вывода (раздел 9.3), было показано, как можно представить задачи удовлетворения ограничений (CSP) в виде определенных выражений. Стандартный язык Prolog позволяет решать подобные задачи точно таким же способом, как и алгоритм поиска с возвратами, приведенный на рис. 6.5.

Поскольку поиск с возвратами предусматривает полный перебор областей определения переменных, он может применяться только для решения задач УО с **конечными областями определения**. В терминах языка Prolog это можно перефразировать таким образом, что для любой цели с несвязанными переменными должно существовать конечное количество решений. (Например, в задаче раскраски карты должно быть указано, что для каждой переменной выбирается один из четырех разных цветов.) Задачи УО с бесконечными областями определения (например, с целочисленными или вещественными переменными) требуют применения совсем других алгоритмов, таких как распространение пределов или линейное программирование.

Рассмотрим следующий пример. Определим предикат `triangle(X,Y,Z)`, выполняющийся успешно, когда три его аргумента являются числами, удовлетворяющими неравенства треугольника:

```
triangle(X,Y,Z) :-
   X>0, Y>0, Z>0, X+Y>Z, Y+Z>X, X+Z>Y.
```

Теперь, если в систему Prolog ввести запрос `triangle(3,4,5)`, она выдаст удовлетворительный ответ. С другой стороны, если ввести запрос `triangle(3,4,Z)`, то решение не будет найдено, поскольку подцель `Z>0` не может быть обработана системой Prolog, — нельзя сравнить несвязанное значение с конкретным термом `0`.

►**Логическое программирование в ограничениях** (*Constraint Logic Programming* — CLP) позволяет *ограничивать*, а не *связывать* переменные. Решением в CLP является наиболее конкретное множество ограничений, налагаемых на переменные запроса, которое может быть определено с помощью базы знаний. Например, для запроса `triangle(3,4,Z)` решением является ограничение `7 > Z > 1`. Программы в стандартной логике представляют собой частный случай программ CLP, в которых ограничения решения должны быть ограничениями равенства, т.е. связываниями.

Системы CLP включают различные алгоритмы решения задач с ограничениями такого типа, которые разрешены к использованию в языке. Например, система, позволяющая использовать линейные неравенства с переменными, имеющими вещественные значения, может включать алгоритм линейного программирования для решения этих ограничений. Кроме того, в системах CLP принят гораздо более гибкий подход к решению запросов стандартного логического программирования. Например, вместо поиска в глубину слева направо с возвратами в них может применяться любой из более эффективных алгоритмов, описанных в главе 6, включая эвристическое упорядочение конъюнктов, обратный переход, определение условий формирования множества отсечения и т.д. В результате в системах CLP сочетаются элементы алгоритмов удовлетворения ограничений, логического программирования и дедуктивных баз данных.

Существует несколько систем, позволяющих программисту получить больший контроль над порядком поиска для логического вывода. Например, язык MRS (Генесерет и Смит [838], 1981; Расселл [1938], 1985) позволяет программисту записывать ►**метаправила** для определения того, какие конъюнкты должны быть опробованы в первую очередь. Пользователь может сформулировать правило с указанием, что в первую очередь следует пытаться достичь цели с наименьшим количеством переменных или оформить характерные для проблемной области правила, касающиеся конкретных предикатов.

9.5. Резолюция

Последнее из трех рассматриваемых в данной главе семейств логических систем и единственное, способное работать с любыми базами знаний, а не только содержащими лишь определенные выражения, основано на использовании **резолюции**. Как было показано в разделе 7.5.1, пропозициональная резолюция — это полная процедура логического вывода для логики высказываний. В этом разделе будет показано, как распространить данную процедуру на логику первого порядка.

9.5.1. Конъюнктивная нормальная форма для логики первого порядка

Первый этап — это преобразование высказываний в **конъюнктивную нормальную форму** (*Conjunctive Normal Form* — **CNF**), т.е. конъюнкцию выражений, в которой каждое выражение представляет собой дизъюнкцию литералов.[5] В CNF литералы могут содержать переменные, на которые, согласно принятому предположению, распространяется квантор всеобщности. Например, высказывание

$$\forall x, y, z \ \ American(x) \wedge Weapon(y) \wedge Sells(x, y, z) \wedge Hostile(z) \Rightarrow Criminal(x)$$

в форме CNF принимает следующий вид:

$$\neg American(x) \vee \neg Weapon(y) \vee \neg Sells(x, y, z) \vee \neg Hostile(z) \vee Criminal(x).$$

Важно то, что ➨ *каждое высказывание в логике первого порядка может быть преобразовано в эквивалентное с точки зрения логического вывода высказывание CNF.*

Процедура преобразования любого высказывания в форму CNF подобна процедуре, применяемой в логике высказываний, представленной в разделе 7.5.2. Принципиальное различие связано с необходимостью устранения кванторов существования. Проиллюстрируем эту процедуру на примере преобразования в форму CNF высказывания "Каждого, кто любит всех животных, кто-то любит", или

$$\forall x \ [\forall y \ Animal(y) \Rightarrow Loves(x, y)] \Rightarrow [\exists y \ Loves(y, x)]$$

(где *Animal* — животное, а *Loves* — любит). Вот этапы этого преобразования.

- **Устранение импликаций**. Замена $P \Rightarrow Q$ на $\neg P \vee Q$. Для нашего примера предложения это нужно сделать дважды:

$$\forall x \ \neg[\forall y \ Animal(y) \Rightarrow Loves(x, y)] \vee [\exists y \ Loves(y, x)];$$
$$\forall x \ \neg[\forall y \ \neg Animal(y) \vee Loves(x, y)] \vee [\exists y \ Loves(y, x)].$$

- **Перемещение связок $\neg$ внутрь выражений.** Кроме обычных правил для отрицаемых связок, нам нужны правила для отрицаемых кванторов. Поэтому получаем следующие правила:

$$\neg \forall x \ p \text{ принимает вид } \exists x \ \neg p;$$
$$\neg \exists x \ p \text{ принимает вид } \forall x \ \neg p.$$

Рассматриваемое здесь высказывание проходит через такие преобразования:

$$\forall x \ [\exists y \ \neg(\neg Animal(y) \vee \text{Loves}(x, y))] \vee [\exists y \ Loves(y, x)];$$
$$\forall x \ [\exists y \ \neg\neg Animal(y) \wedge \neg Loves(x, y)] \vee [\exists y \ Loves(y, x)];$$

[5] Любое выражение также может быть представлено как импликация с конъюнкцией атомов слева и дизъюнкцией атомов справа. Эту форму, называемую импликативной нормальной формой, иногда также называют формой Ковальски, особенно при использовании записи с символом импликации, ориентированным справа налево (Ковальски [1296], 1979). Часто эта форма намного удобнее для чтения по сравнению с другими формами.

$$\forall x \ [\exists y \ Animal(y) \land \neg Loves(x, y)] \lor [\exists y \ Loves(y, x)].$$

Обратите внимание на то, что квантор всеобщности ($\forall y$) в предпосылке импликации стал квантором существования. Теперь это высказывание приобрело такое прочтение: "Либо существует какое-то животное, которого x не любит, либо (если это утверждение не является истинным) кто-то любит x". Очевидно, что смысл первоначального высказывания был сохранен.

- **Стандартизация переменных**. В высказываниях наподобие $(\exists x \ P(x)) \lor (\exists x \ Q(x))$, в которых одно и то же имя переменной используется дважды, изменим имя одной из переменных. В дальнейшем это позволит избежать путаницы после удаления кванторов. В результате получим следующее:

$$\forall x \ [\exists y \ Animal(y) \land \neg Loves(x, y)] \lor [\exists z \ Loves(z, x)].$$

- **Сколемизация**. ▶**Сколемизация** — это процесс устранения кванторов существования путем их удаления. В данном простом случае этот процесс подобен применению правила конкретизации с помощью квантора существования, приведенного в разделе 9.1: преобразовать $\exists x \ P(x)$ в $P(A)$, где A — новая константа. Однако если это правило будет непосредственно применено к высказыванию, рассматриваемому в качестве примера, то будет получено высказывание

$$\forall x \ [Animal(A) \land \neg Loves(x, A)] \lor Loves(B, x),$$

имеющее полностью неправильный смысл: в нем утверждается, что каждый либо не способен любить какое-то конкретное животное A, либо его любит некоторая конкретная сущность B. В действительности первоначальное высказывание позволяет каждому человеку не быть способным любить какое-то другое животное или быть любимым другим человеком. Поэтому желательно, чтобы сущности, определяемые в процессе сколемизации, зависели от x:

$$\forall x \ [Animal(F(x)) \land \neg Loves(x, F(x))] \lor Loves(G(x), x),$$

где F и G — ▶**сколемовские функции**. Общее правило состоит в том, что все параметры сколемовской функции должны быть переменными, на которые распространяются кванторы всеобщности, в область действия которых попадает соответствующий квантор существования. Как и при использовании конкретизации с помощью квантора существования, сколемизированное высказывание является выполнимым тогда и только тогда, когда выполнимо первоначальное высказывание.

- **Удаление кванторов всеобщности**. В данный момент на все оставшиеся переменные должны распространяться кванторы всеобщности. Следовательно, мы не теряем никакой информации, если отбрасываем квантор:

$$[Animal(F(x)) \land \neg Loves(x, F(x))] \lor Loves(G(x), x).$$

- **Распределение связки** $\vee$ **по** $\wedge$:

  $$[Animal(F(x)) \vee Loves(G(x), x)] \wedge [\neg Loves(x, F(x)) \vee Loves(G(x), x)].$$

  На этом этапе может также потребоваться выполнить раскрытие скобок во вложенных конъюнкциях и дизъюнкциях.

Теперь исходное высказывание находится в форме CNF и состоит из двух выражений. Оно гораздо сложнее для восприятия, чем первоначальное высказывание с импликациями. (Помочь его понять может такое пояснение, что сколемовская функция $F(x)$ указывает на животное, которое потенциально может быть нелюбимым лицом x, а функция $G(x)$ указывает на кого-то, кто может любить лицо x.) К счастью, людям редко приходится анализировать высказывания в форме CNF, поскольку приведенный выше процесс преобразования легко автоматизировать.

9.5.2. Правило логического вывода с помощью резолюции

Правило резолюций для выражений в логике первого порядка представляет собой поднятую версию правила резолюций для логики высказываний, приведенного в разделе 7.5.2. Два выражения, которые, как предполагается, должны были быть стандартизированы таким образом, чтобы в них не было общих переменных, могут быть подвергнуты операции резолюции, если они содержат взаимно дополнительные литералы. Литералы логики высказываний являются взаимно дополнительными, если один из них представляет собой отрицание другого, а литералы в логике первого порядка являются взаимно дополнительными, если один из них *унифицируется* с отрицанием другого. Поэтому имеет место следующее правило:

$$\frac{\ell_1 \vee \cdots \vee \ell_k, \qquad m_1 \vee \cdots \vee m_n}{\text{SUBST}(\theta, \ell_1 \vee \cdots \vee \ell_{i-1} \vee \ell_{i+1} \vee \cdots \vee \ell_k \vee m_1 \vee \cdots \vee m_{j-1} \vee m_{j+1} \vee \cdots \vee m_n)},$$

где $\text{UNIFY}(\ell_i, \neg m_j) = \theta$. Например, можно применить операцию резолюции к двум выражениям,

$$[Animal(F(x)) \vee Loves(G(x), x)] \quad \text{и} \quad [\neg Loves(u, v) \vee \neg Kills(u, v)],$$

путем устранения взаимно дополнительных литералов $Loves(G(x), x)]$ и $\neg Loves(u, v)$ с помощью унификатора $\theta = \{u/G(x), v/x\}$ и получения следующего выражения, называемого **резольвентой**:

$$[Animal(F(x)) \vee \neg Kills(G(x), x)].$$

Это правило называется правилом ▶**бинарной резолюции**, поскольку предполагает удаление с помощью резолюции точно двух взаимно дополнительных литералов. Отдельно взятое правило бинарной резолюции не позволяет получить полную процедуру логического вывода. Правило полной резолюции позволяет удалять в каждом выражении подмножества литералов, являющихся

унифицируемыми. Альтернативный подход состоит в распространении операции **факторизации** — удаления избыточных литералов — на логику первого порядка. В пропозициональной факторизации два литерала сводятся к одному, если они *идентичны*, а при факторизации в логике первого порядка два литерала сводятся к одному, если они являются *унифицируемыми*. Унификатор должен быть применен ко всему выражению. Сочетание бинарной резолюции и факторизации позволяет создать *полную* процедуру логического вывода.

9.5.3. Примеры доказательств

При резолюции доказательство того, что $KB \models \alpha$ (из базы знаний KB следует высказывание α) осуществляется путем доказательства невыполнимости выражения $KB \wedge \neg\alpha$, т.е. путем вывода пустого выражения. Алгоритмический подход, применяемый в логике первого порядка, идентичен подходу в логике высказываний, представленному на рис. 7.13, поэтому мы не будем здесь его повторять. Вместо этого приведем два примера доказательства. Первый из них — это пример доказательства преступления, уже обсуждавшийся в разделе 9.3. Соответствующие высказывания, преобразованные в форму CNF, выглядят следующим образом:

$$\neg American(x) \vee \neg Weapon(y) \vee \neg Sells(x, y, z) \vee \neg Hostile(z) \vee Criminal(x)$$
$$\neg Missile(x) \vee \neg Owns(Nono, x) \vee Sells(West, x, Nono)$$
$$\neg Enemy(x, America) \vee Hostile(x)$$
$$\neg Missile(x) \vee Weapon(x)$$
$$Owns(Nono, M_1) \qquad Missile(M_1)$$
$$American(West) \qquad Enemy(Nono, America).$$

В число этих высказываний должна быть включена также отрицаемая цель $\neg Criminal(West)$. Процедура доказательства по методу резолюции приведена на рис. 9.10. Обратите внимание на его структуру: единственная "вертикаль" начинается с целевого выражения, и операция резолюции применяется к выражениям из базы знаний до тех пор, пока не образуется пустое выражение. В этом состоит характерная особенность применения метода резолюции к базам знаний, представленным в виде хорновских выражений. В действительности выражения, расположенные вдоль главной вертикали, *точно* соответствуют последовательным значениям *целевых переменных* в алгоритме обратного логического вывода, представленном на рис. 9.6. Это связано с тем, что для резолюции всегда выбирается выражение, положительный литерал которого унифицируется с крайним слева литералом "текущего" выражения в вертикали; именно это происходит при обратном логическом выводе. Таким образом, обратный логический вывод в действительности представляет собой просто частный случай резолюции, в котором применяется конкретная стратегия управления для определения того, какая операция резолюции должна быть выполнена в следующую очередь.

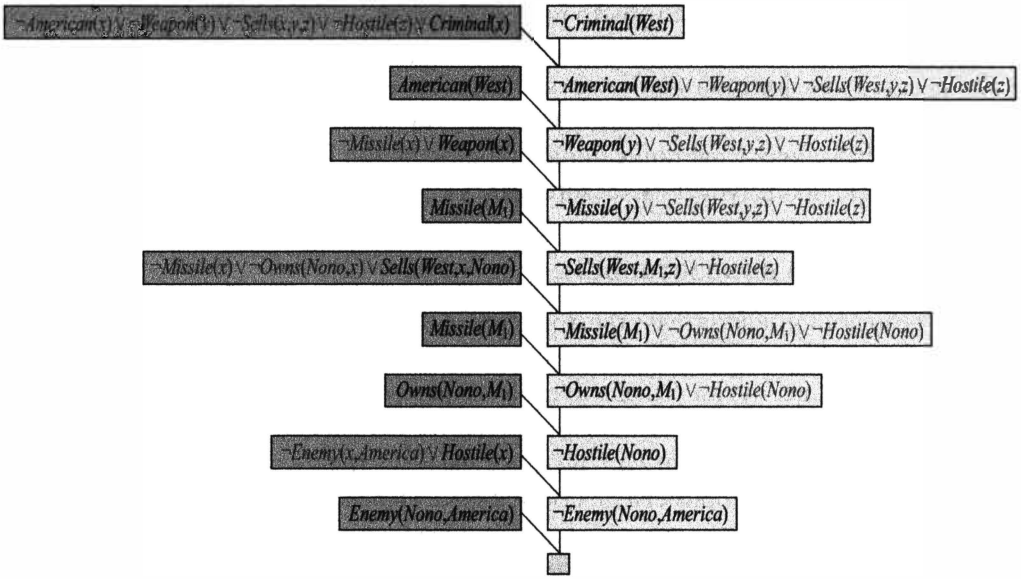


Рис. 9.10. Процедура доказательства с помощью резолюции того, что полковник Вест совершил преступление. На каждом этапе резолюции унифицируемые литералы выделены черным шрифтом, а выражения с положительным литералом имеют более темную заливку

Второй пример демонстрирует использование сколемизации и включает выражения, которые не являются определенными. В результате структура доказательства становится немного более сложной. На естественном языке эта задача формулируется следующим образом.

Каждого, кто любит всех животных, кто-то любит.
Любого, кто убивает животных, никто не любит.
Джек любит всех животных.
Кота по имени Туна убил либо Джек, либо любопытство (англ. — *curiosity*).
Этого кота убило любопытство?

Сначала представим в логике первого порядка первоначальные высказывания, некоторые фоновые знания и отрицаемую цель G.

A. $\forall x\ [\forall y\ Animal(y) \Rightarrow Loves(x, y)] \Rightarrow [\exists y\ Loves(y, x)]$
B. $\forall x\ [\exists z\ Animal(z) \wedge Kills(x, z)] \Rightarrow [\forall y\ \neg Loves(y, x)]$
C. $\forall x\ Animal(x) \Rightarrow Loves(Jack, x)$
D. $Kills(Jack, Tuna) \vee Kills(Curiosity, Tuna)$
E. $Cat(Tuna)$
F. $\forall x\ Cat(x) \Rightarrow Animal(x)$
¬G. $\neg Kills(Curiosity, Tuna)$

Теперь применим процедуру преобразования, чтобы преобразовать каждое высказывание в форму CNF.

A1. $Animal(F(x)) \vee Loves(G(x), x)$
A2. $\neg Loves(x, F(x)) \vee Loves(G(x), x)$
B. $\neg Loves(y, x) \vee \neg Animal(z) \vee \neg Kills(x, z)$
C. $\neg Animal(x) \vee Loves(Jack, x)$
D. $Kills(Jack, Tuna) \vee Kills(Curiosity, Tuna)$
E. $Cat(Tuna)$
F. $\neg Cat(x) \vee Animal(x)$
$\neg$G. $\neg Kills(Curiosity, Tuna)$

Доказательство с помощью метода резолюции того, что кота убило любопытство, приведено на рис. 9.11. На естественном языке это доказательство может быть перефразировано, как показано ниже.

> Предположим, что кота Туну убило не любопытство. Мы знаем, что это сделал либо Джек, либо любопытство; в таком случае это должен был сделать Джек. Итак, Туна — кот, а коты — животные, поэтому Туна — животное. Любого, кто убивает животное, никто не любит, поэтому мы делаем вывод, что никто не любит Джека. С другой стороны, Джек любит всех животных, поэтому есть кто-то, кто его любит; таким образом, возникает противоречие. Следовательно, можно сделать вывод, что кота убило любопытство.

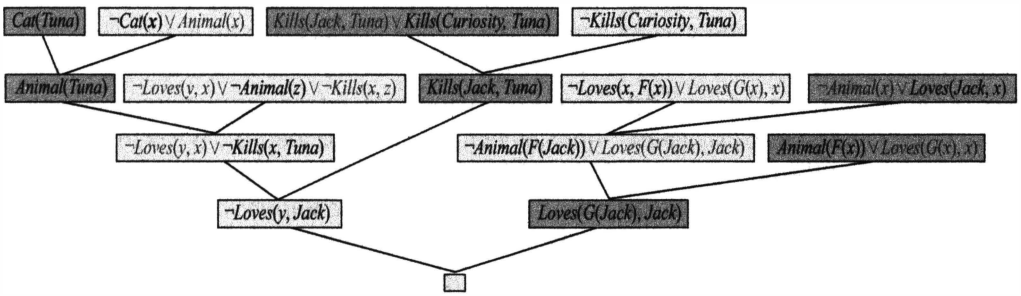


Рис. 9.11. Процедура доказательства с помощью резолюции того, что кота убило любопытство. Обратите внимание, при выводе выражения $Loves(G(Jack), Jack)$ использовалась факторизация. Также обратите внимание, что вверху справа унификация выражений $Loves(x, F(x))$ и $Loves(Jack, x)$ может быть успешной только после того, как переменные были стандартизованы отдельно

Такое доказательство на самом деле отвечает на вопрос "Действительно ли этого кота убило любопытство?", но часто требуется найти ответ на более общие вопросы, такие как "Кто убил кота?" Резолюция позволяет это сделать, но для

получения ответа потребуется немного больше работы. Данная цель может быть представлена в виде выражения $\exists\ w\ \ Kills(w, Tuna)$, которое после его отрицания имеет в форме CNF вид $\neg Kills(w, Tuna)$. Если повторить доказательство, приведенное на рис. 9.11, с новой отрицаемой целью, мы получим аналогичное дерево доказательства, но с подстановкой $\{w/Curiosity\}$ на одном из этапов. Поэтому в данном случае для поиска того, кто убил кота, достаточно проследить за связываниями, которые применяются к переменным запроса в этом доказательстве. К сожалению, метод резолюции иногда может вырабатывать ▶ **неконструктивные доказательства** для существующих целей, когда мы знаем, что запрос верен, но не существует уникального варианта связывания для переменной.

9.5.4. Полнота резолюции

В данном разделе приведено доказательство полноты резолюции. Это доказательство может пропустить без ущерба для дальнейшего понимания текста любой читатель, который готов принять его на веру.

Мы покажем, что метод резолюции обеспечивает ▶ **полноту опровержения** (*refutation completeness*), — это означает, что *если* множество высказываний является невыполнимым, то резолюция всегда позволяет прийти к противоречию. Резолюцию нельзя использовать для выработки всех логических следствий из множества высказываний, но она может применяться для подтверждения того, что данное конкретное высказывание следует из множества высказываний. Поэтому резолюция может служить для поиска всех ответов на любой конкретный вопрос $Q(x)$ путем доказательства того, что выражение $KB \wedge \neg Q(x)$ невыполнимо.

Примем как истинное утверждение, что любое высказывание в логике первого порядка (без использования равенства) может быть перезаписано в виде множества выражений в форме CNF. Это можно доказать методом индукции на основе анализа формы высказывания, применяя в качестве базового случая атомарные высказывания (Дэвис и Путнам [544], 1960). Поэтому наша цель состоит в том, чтобы доказать следующее: ➨ *если S — невыполнимое множество выражений, то применение к S конечного количества этапов резолюции приведет к противоречию.*

Схема данного доказательства повторяет первоначальное доказательство, приведенное Робинсоном, с некоторыми упрощениями, которые были внесены Генезеретом и Нильссоном ([836], 1987). Основная структура доказательства приведена на рис. 9.12; оно осуществляется, следующим образом.

1. Вначале отметим, что если множество выражений S невыполнимо, то существует такое конкретное множество *базовых экземпляров* выражений в S, что это множество также будет невыполнимо (теорема Эрбрана).
2. Далее прибегнем к **основной теореме резолюции**, приведенной в разделе 7.5.2, в которой утверждается, что пропозициональная резолюция является полной для базовых высказываний.

3. Теперь воспользуемся **леммой поднятия**, чтобы показать, что для любого доказательства по методу пропозициональной резолюции, в котором применяется множество базовых высказываний, существует соответствующее доказательство по методу резолюции первого порядка с использованием высказываний в логике первого порядка, из которых были получены базовые высказывания.

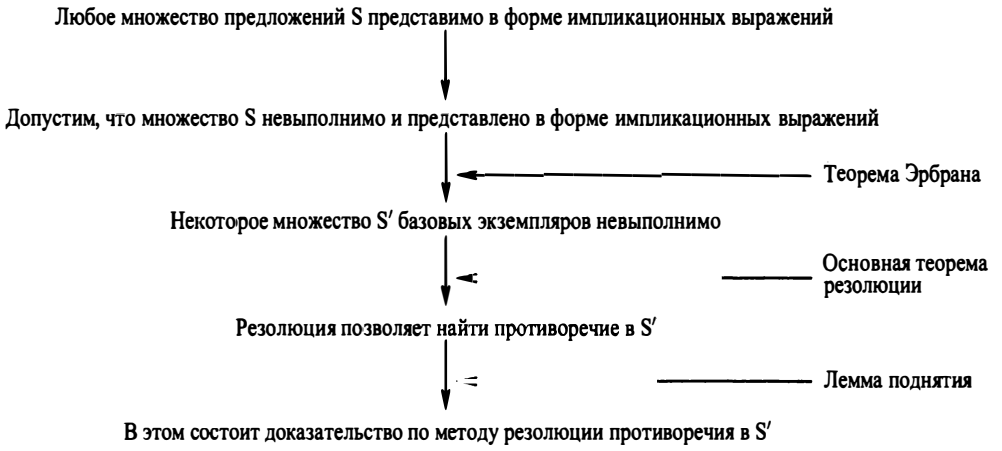


Рис. 9.12. Структурная схема доказательства полноты резолюции

Чтобы выполнить первый этап доказательства, потребуются три новых понятия, описанных ниже.

- ▶**Универсум Эрбрана**. Если S — множество выражений, то H_S, *универсум Эрбрана* для множества S, представляет собой множество всех базовых термов, которые могут быть сформированы из следующего:
  - функциональные символы из множества S, если они имеются;
  - символы констант из множества S, если таковые имеются; если они отсутствуют, то символ константы по умолчанию, S.

  Например, если множество S содержит только выражение $\neg P(x, F(x, A)) \vee \neg Q(x, A) \vee R(x, B)$, то универсум Эрбрана H_S представляет собой следующее бесконечное множество базовых термов:

  $$\{A, B, F(A, A), F(A, B), F(B, A), F(B, B), F(A, F(A, A)), \ldots\}.$$

- **Насыщение**. Если S — множество выражений, а P — множество базовых термов, то $P(S)$, *насыщение S по отношению* к P, представляет собой множество всех базовых выражений, полученное путем применения всех возможных совместимых подстановок базовых термов из P вместо переменных в S.

- ▶**База Эрбрана**. Насыщение множества выражений S по отношению к его универсуму Эрбрана H_S называется *базой Эрбрана* множества S и записывается как $H_S(S)$. Например, если S содержит только приведенное выше выражение, то $H_S(S)$ представляет собой следующее бесконечное множество выражений:

$$\{\neg P(A, F(A, A)) \vee \neg Q(A, A) \vee R(A, B),$$
$$\neg P(B, F(B, A)) \vee \neg Q(B, A) \vee R(B, B),$$
$$\neg P(F(A, A), F(F(A, A), A)) \vee \neg Q(F(A, A), A) \vee R(F(A, A), B),$$
$$\neg P(F(A, B), F(F(A, B), A)) \vee \neg Q(F(A, B), A) \vee R(F(A, B), B), \ldots \}.$$

Эти определения позволяют сформулировать одну из форм ▶**теоремы Эрбрана** (Эрбран [1013], 1930).

> Если множество выражений S является невыполнимым, то существует конечное подмножество $H_S(S)$, которое также является невыполнимым.

Пусть S' — это конечное подмножество базовых высказываний. Теперь можно прибегнуть к основной теореме резолюции (раздел 7.5.2), чтобы показать, что **резолюционное замыкание** $RC(S')$ содержит пустое выражение. А это означает, что доведение процесса пропозициональной резолюции до конца применительно к S' приводит к противоречию.

Теперь, после определения того, что всегда существует доказательство по методу резолюции, в котором применяется некоторое конечное подмножество базы Эрбрана множества S, на следующем этапе необходимо показать, что существует доказательство по методу резолюции, в котором используются выражения из самого множества S, которые не обязательно являются базовыми выражениями. Начнем с рассмотрения одиночного приложения правила резолюции. Из базовой леммы Робинсона следует приведенный ниже факт.

> Допустим, что C_1 и C_2 — два выражения без общих переменных, а C_1' и C_2' — базовые экземпляры C_1 и C_2. Если C' — резольвента C_1' и C_2', то существует выражение C, такое, что, во-первых, C — резольвента C_1 и C_2, и, во-вторых, C' — базовый экземпляр C.

Это утверждение называется ▶**леммой поднятия** (*lifting lemma*), поскольку позволяет поднять любой этап доказательства от базовых выражений к общим выражениям в логике первого порядка. Для того чтобы доказать свою основную лемму поднятия, Робинсону пришлось изобрести унификацию и определить все свойства наиболее общих унификаторов. Вместо повторения доказательства Робинсона просто проиллюстрируем применение этой леммы.

$$\begin{aligned}
C_1 &= \neg P(x, F(x, A)) \vee \neg Q(x, A) \vee R(x, B) \\
C_2 &= \neg N(G(y), z) \vee P(H(y), z) \\
C_1' &= \neg P(H(B), F(H(B), A)) \vee \neg Q(H(B), A) \vee R(H(B), B) \\
C_2' &= \neg N(G(B), F(H(B), A)) \vee P(H(B), F(H(B), A))
\end{aligned}$$

$$C' = \neg N(G(B), F(H(B), A)) \vee \neg Q(H(B), A) \vee R(H(B), B)$$
$$C = \neg N(G(y), F(H(y), A)) \vee \neg Q(H(y), A) \vee R(H(y), B)$$

Очевидно, что C' — действительно базовый экземпляр выражения C. Вообще говоря, для того чтобы выражения C_1' и C_2' имели какие-либо резольвенты, они должны быть получены путем предварительного применения к выражениям C_1 и C_2 наиболее общего унификатора для пары взаимно дополнительных литералов в C_1 и C_2. Из леммы поднятия можно легко получить аналогичное, приведенное ниже утверждение о любой последовательности применений правила резолюции.

> Для любого выражения C' в резолюционном замыкании множества выражений S' существует выражение C в резолюционном замыкании множества выражений S, такое, что C' — базовый экземпляр выражения C и логический вывод C имеет такую же длину, как и логический вывод C .

Из этого факта следует, что если в резолюционном замыкании множества выражений S' появляется пустое выражение, оно должно также появиться в резолюционном замыкании множества выражений S. Это связано с тем, что пустое выражение не может быть базовым экземпляром любого другого выражения. Подведем итог: мы показали, что если множество выражений S невыполнимо, то для него существует конечная процедура логического вывода пустого выражения с помощью правила резолюции.

Поднятие способа доказательства теорем от базовых выражений к выражениям первого порядка обеспечивает огромное увеличение мощи доказательства. Это увеличение связано с тем фактом, что теперь в доказательстве первого порядка конкретизация переменных может выполняться только по мере того, как это потребуется для самого доказательства, тогда как в методах с использованием базовых выражений приходилось исследовать огромное количество произвольных конкретизаций.

Теорема Гёделя о неполноте

Немного дополнив язык логики первого порядка для обеспечения возможности применять **схему математической индукции** в арифметике, Гёдель в своей **теореме о неполноте** сумел показать, что существуют истинные арифметические высказывания, которые не могут быть доказаны.

Доказательство теоремы о неполноте немного выходит за рамки настоящей книги, поскольку в своем непосредственном виде оно занимает не меньше 30 страниц, поэтому здесь мы приведем лишь его набросок. Начнем с логической теории чисел. В этой теории существует единственная константа, 0, и единственная функция, S (функция определения преемника). В предполагаемой модели интерпретации этой теории $S(0)$ обозначает 1, $S(S(0))$ обозначает 2 и так далее, поэтому в рассматриваемом языке имеются имена для всех натуральных чисел. Кроме того, словарь включает функциональные символы $+$, $\times$ и *Expt* (возведение в степень), а также обычное множество логических связок и кванторов.

Прежде всего, отметим, что множество высказываний, которые могут быть записаны на этом языке, может быть пронумеровано. (Для этого достаточно представить себе, что определен алфавитный порядок символов, а затем в алфавитном порядке расположено каждое из множеств высказываний длиной 1, 2 и т.д.) Тогда можно обозначить каждое высказывание α уникальным натуральным числом $\#\alpha$ (которое называется **гёделевским номером**). Это самый важный момент доказательства; в нем утверждается, что теория чисел включает отдельное имя для каждого из ее собственных высказываний. Аналогичным образом с помощью гёделевского номера $G(P)$ можно пронумеровать каждое возможное доказательство P, поскольку любое доказательство — это просто конечная последовательность высказываний.

Теперь предположим, что имеется рекурсивно перечислимое множество A высказываний, которые представляют собой истинные утверждения о натуральных числах. Напомним, что высказывания из множества A можно именовать с помощью заданного множества целых чисел, поэтому можно представить себе, что на нашем языке записывается высказывание $\alpha(j, A)$ такого рода.

> $\forall\ i\ i$ — не гёделевский номер доказательства высказывания, гёделевским номером которого является j, где в доказательстве используются только предпосылки из множества A.

Далее допустим, что σ представляет собой высказывание $\alpha(\#\sigma, A)$, т.е. высказывание, в котором утверждается его собственная недоказуемость из A. (Утверждение о том, что такое высказывание всегда существует, является истинным, но его нельзя назвать полностью очевидным.)

А теперь применим следующий остроумный довод: предположим, что высказывание σ доказуемо из A; в таком случае высказывание σ ложно (поскольку в высказывании σ утверждается, что оно не может быть доказано). Но это означает, что имеется некоторое ложное высказывание, которое доказуемо из A, поэтому A не может состоять только из истинных высказываний, а это противоречит нашей предпосылке. Поэтому высказывание σ *не доказуемо* из A. Но именно это и утверждает о самом себе высказывание σ; следовательно, σ — истинное высказывание.

Итак, мы показали (и сэкономили 29 страниц), что для любого множества истинных высказываний теории чисел и, в частности, для любого множества базовых аксиом существуют другие истинные высказывания, которые не могут быть доказаны из этих аксиом. Из этого, кроме всего прочего, следует, что мы никогда не сможем доказать все теоремы математики *в пределах любой конкретной системы аксиом*. Очевидно, что это открытие имело для математики очень большое значение. Значимость этого открытия для искусственного интеллекта была предметом широких обсуждений, начиная с размышлений самого Гёделя. Мы вернемся к этим обсуждениям в главе 27.

9.5.5. Равенство

Ни в одном из методов логического вывода, описанных до сих пор в этой главе, не было возможности манипулировать выражениями в форме $x = y$ без некоторой дополнительной обработки. Для решения этой проблемы может быть принято три различных подхода. Первый состоит в аксиоматизации равенства — включении в базу знаний высказываний, касающихся отношения равенства. При этом необходимо будет указать, что отношение равенства является рефлексивным, симметричным и транзитивным, а также сформулировать утверждение, что в любом предикате или функции можно заменять равные литералы равными. Следовательно, потребуются три базовые аксиомы и еще по одной аксиоме для каждого предиката и функции, как показано ниже.

$$
\begin{aligned}
&\forall x \quad x = x \\
&\forall x, y \quad x = y \Rightarrow y = x \\
&\forall x, y, z \quad x = y \wedge y = z \Rightarrow x = z \\
&\forall x, y \quad x = y \Rightarrow (P_1(x) \Leftrightarrow P_1(y)) \\
&\forall x, y \quad x = y \Rightarrow (P_2(x) \Leftrightarrow P_2(y)) \\
&\vdots \\
&\forall w, x, y, z \quad w = y \wedge x = z \Rightarrow (F_1(w, x) = F_1(y, z)) \\
&\forall w, x, y, z \quad w = y \wedge x = z \Rightarrow (F_2(w, x) = F_2(y, z)) \\
&\vdots
\end{aligned}
$$

При наличии в базе знаний подобных высказываний любая стандартная процедура логического вывода, такая как резолюция, позволит решать задачи, требующие формирования рассуждений с учетом отношения равенства, например находить решения математических уравнений. Однако эти аксиомы будут генерировать множество выводов, большинство из которых бесполезно для доказательства. Поэтому второй способ учета отношения равенства состоит в использовании дополнительного правила логического вывода. В простейшем правиле, правиле **демодуляции**, берется единичное выражение $x = y$ и некоторое выражение α, содержащее терм x, а затем формируется новое выражение путем подстановки терма y вместо x в выражении α. Это допустимо, даже если терм в α лишь унифицируется с x, точного равенства с x не требуется. Обратите внимание, что демодуляция является направленной: при заданном $x = y$ всегда только x заменяется на y, и никогда наоборот. Это означает, что метод демодуляции может использоваться для упрощения выражений с использованием набора утверждений, таких как $z + 0 = z$ или $z^1 = z$. Можно привести и другой пример. Если дано

$$
\begin{aligned}
&Father(Father(x)) = PaternalGrandfather(x) \\
&Birthdate(Father(Father(Bella)), 1926),
\end{aligned}
$$

применив демодуляцию, можно сделать вывод

$$Birthdate(PaternalGrandfather(Bella), 1926).$$

Можно привести и более формальное определение.

- ▶**Демодуляция**. Для любых термов x, y и z, где z появляется где-либо в литерале m_i и где UNIFY$(x, z) = \theta$, справедливо следующее:

$$\frac{x = y, \qquad m_1 \vee \cdots \vee m_n}{\text{SUB}(\text{SUBST}(\theta, x), \text{SUBST}(\theta, y), m_1 \vee \cdots \vee m_n)},$$

  где функция SUBST является обычной подстановкой связываемого списка, а функция SUB(x, y, m) выполняет замену x на y где-либо в пределах m.

Это правило также может быть расширено для обработки неединичных выражений, в которых присутствует знак равенства:

- ▶**Парамодуляция**. Для любых термов x, y и z, где z появляется где-либо в литерале m_i и где UNIFY$(x, z) = \theta$, справедливо следующее:

$$\frac{\ell_1 \vee \cdots \vee \ell_k \vee x = y, \qquad m_1 \vee \cdots \vee m_n}{\text{SUB}(\text{SUBST}(\theta, x), \text{SUBST}(\theta, y), \text{SUBST}(\theta, \ell_1 \vee \cdots \vee \ell_k \vee m_1 \vee \cdots \vee m_n))}.$$

Например, из

$$P(F(x, B), x) \vee Q(x) \quad \text{и} \quad F(A, y) = y \vee R(y)$$

мы имеем $\theta = \text{UNIFY}(F(A, y), F(x, B)) = \{x/A, y/B\}$ и посредством парамодуляции можем вывести следующее высказывание:

$$P(B, A) \vee Q(A) \vee R(B).$$

Парамодуляция обеспечивает полную процедуру логического вывода для логики первого порядка с отношением равенства.

В третьем подходе формирование логических рассуждений с учетом отношения равенства полностью осуществляется с помощью расширенного алгоритма унификации. Это означает, что термы рассматриваются как унифицируемые, если можно доказать, что они становятся равными при некоторой подстановке, — здесь выражение "можно доказать" допускает включение рассуждений о равенстве. Например, термы $1+2$ и $2+1$ обычно не рассматриваются как унифицируемые, но алгоритм унификации, в котором известно, что $x+y = y+x$, способен унифицировать их с помощью пустой подстановки. Такого рода ▶**унификация с учетом равенства** (*equational unification*) может выполняться с помощью эффективных алгоритмов, разработанных с применением прямого использования определенных аксиом (коммутативность, ассоциативность и т.д.), а не явного логического вывода на основе этих аксиом. Программы автоматического доказательства теорем с использованием этого метода очень близки к системам логического программирования в ограничениях, описанным в разделе 9.4.

9.5.6. Стратегии резолюции

Как известно, повторные применения правила логического вывода на основе резолюции позволяют в конечном итоге найти доказательство, если оно существует. В этом разделе рассматриваются стратегии, позволяющие находить доказательства более *эффективно* в сравнении с простым перебором.

► **Преимущественное использование единичных выражений**. В этой стратегии преимущество отдается таким операциям резолюции, в которых одним из высказываний является единственный литерал (известный также как **единичное выражение** — *unit clause*). В основе этой стратегии лежит идея, что если предпринимаются попытки получения пустого выражения, то может оказаться целесообразным отдавать предпочтение таким операциям логического вывода, в которых вырабатываются более короткие выражения. Применение операции резолюции к единичному высказыванию (такому, как P) в сочетании с любым другим высказыванием (таким, как $\neg P \vee Q \vee R$) всегда приводит к получению выражения более короткого (в данном случае это $\neg Q \vee R$), чем остальные высказывания. Когда стратегия с преимущественным использованием единичных выражений была впервые опробована в пропозициональном логическом выводе в 1964 году, она привела к резкому ускорению работы, обеспечив возможность доказывать теоремы, с которыми не удавалось справиться без использования этого метода предпочтения. **Единичная резолюция** (*unit resolution*) представляет собой ограниченную форму резолюции, в которой на каждом этапе резолюции должно участвовать единичное выражение. В общем случае метод единичной резолюции является неполным, но становится полным при его применении к хорновским выражениям. Процесс доказательства по методу единичной резолюции применительно к хорновским выражениям напоминает прямой логический вывод.

В программе доказательства теорем OTTER (Мак-Кун [1538], 1990) используется особая форма жадного поиска по первому наилучшему совпадению. Ее эвристическая функция измеряет "вес" каждого выражения и более легким из них отдается предпочтение. Точный выбор эвристики возлагается на пользователя, но в целом вес выражения должен коррелировать с его размерами или сложностью. Единичные выражения рассматриваются как легкие, и, следовательно, этот вариант поиска можно рассматривать как обобщение стратегии преимущественного использования единичных выражений.

► **Множество поддержки**. Метод предпочтений, предполагающий в первую очередь попытаться выполнить определенные операции резолюции, вполне оправдан, но в целом более эффективно будет попытаться полностью устранить некоторые возможные этапы резолюции. Например, можно потребовать, чтобы каждый этап резолюции включал хотя бы один элемент из специального подмножества высказываний — *множества поддержки*. Полученная резольвента добавляется к множеству поддержки. Если множество поддержки будет небольшим по

сравнению со всей базой знаний, это позволит резко сократить пространство поиска.

Для обеспечения полноты этой стратегии можно выбирать множество поддержки S так, чтобы оставшиеся высказывания, вместе взятые, оставались выполнимыми. Например, в качестве множества поддержки можно использовать отрицаемый запрос, — в предположении, что база знаний является непротиворечивой. (В конце концов, если она не является непротиворечивой, то сам факт, что запрос является следствием из этой базы, делает его бессмысленным.) Стратегия с использованием множества поддержки имеет дополнительное преимущество в том, что в ней часто вырабатываются деревья доказательства, легко доступные для понимания людей.

► **Резолюция с входными высказываниями** (*input resolution*). В этой стратегии на каждом этапе резолюции одно из входных высказываний (из базы знаний или запроса) комбинируется с некоторым другим высказыванием. В доказательстве, представленном на рис. 9.10, на всех этапах выполнялась резолюция с входными высказываниями, и поэтому дерево доказательства имело характерную форму в виде одной "вертикали" с отдельными высказываниями, комбинирующимися с ней. Очевидно, что пространство деревьев доказательства такой формы меньше по сравнению с пространством всех возможных графов доказательства. В хорновских базах знаний правило отделения может рассматриваться как своего рода стратегия резолюции с входными высказываниями, поскольку при использовании этого правила некоторая импликация из первоначальной базы знаний комбинируется с некоторыми другими высказываниями. Таким образом, нет ничего удивительного в том, что метод резолюции с входными высказываниями является полным применительно к хорновским базам знаний, но в общем случае он неполон. Стратегия ► **линейной резолюции** (*linear resolution*) представляет собой некоторое обобщение, в котором допускается применять в одной операции резолюции высказывания P и Q, если P находится в первоначальной базе знаний или если высказывание P является предком Q в дереве доказательства. Метод линейной резолюции является полным.

► **Обобщение**. В методе обобщения (*subsumption*) устраняются все высказывания, которые обобщаются некоторым существующим высказыванием из базы знаний (т.е. являются более конкретными по сравнению с ним). Например, если в базе знаний есть высказывание $P(x)$, то нет смысла вводить в нее высказывание $P(A)$ и еще меньше смысла вводить в нее высказывание $P(A) \vee Q(B)$. Обобщение позволяет поддерживать небольшие размеры базы знаний и тем самым ограничивать размеры пространства поиска.

► **Обучение**. Можно улучшить программу доказательства теорем, обучая ее на основании уже имеющегося опыта. При наличии подборки ранее доказанных теорем система машинного обучения учится посредством поиска ответа на вопрос "При заданном множестве предпосылок и целей для доказательства какие этапы

доказательства являются схожими с этапами, которые оказались успешными в прошлом?" В системе DEEPHOL Бансал и соавт. ([126], 2019) поступили именно так, использовав глубокие нейронные сети (см. главу 21) для построения моделей (называемых *вложениями*) целей и предпосылок, а затем применив их для осуществления выбора. При обучении в качестве примеров могут использоваться доказательства, выполненные как человеком, так и компьютером, начиная с подборки из 10 000 доказательств.

Практическое использование систем автоматического доказательства теорем

В этой главе было показано, как логика первого порядка позволяет представлять простой сценарий реального мира, включающий такие понятия, как продажа, оружие и гражданство. Но сложные сценарии реального мира обладают слишком большой неопределенностью и включают слишком много неизвестных. Уже доказано, что логика может успешно использоваться в предметных областях, включающих формальные, строго определенные концепции, — таких как ▶**синтез** и ▶**верификация** аппаратного и программного обеспечения. Исследования по доказательству теорем проводятся в областях проектирования аппаратного обеспечения, языков программирования и разработки программного обеспечения, а не только в области ИИ.

В области аппаратного обеспечения аксиомы описывают взаимодействие между сигналами и элементами цепи (см. пример в разделе 8.4.2). Блоки логических рассуждений, предназначенные специально для верификации электронных схем, оказались способными проверить целые процессоры, включая синхронизацию работы их элементов (Шривас и Бикфорд [2118], 1990). Система доказательства теорем AURA применялась для разработки схем, которые оказались более компактными, чем любые предыдущие решения (Войцеховский и Войцик [2370], 1983).

Формальный синтез алгоритмов был одним из первых направлений использования средств автоматического доказательства теорем, как и было намечено Корделлом Грином ([919], 1969), который опирался на идеи, высказанные ранее Гербертом Саймоном ([2071], 1963). Общая идея состояла в том, что следует доказать теорему с утверждением, что "существует программа p, удовлетворяющая определенной спецификации". Хотя полностью автоматизированный ▶**дедуктивный синтез**, как позднее стало называться это направление, еще не осуществим применительно к программированию задач общего назначения, дедуктивный синтез, управляемый вручную, оказался успешным при проектировании нескольких новейших и сложнейших алгоритмов. Кроме того, активной областью исследования является синтез программ специального назначения, например научного вычислительного кода.

Подобные методы сейчас применяются и для проверки программного обеспечения с помощью таких систем, как программа проверки моделей SPIN (Хольцман [1053], 1997). С ее помощью, например, была проверена до и после полета

программа управления космическим аппаратом Remote Agent (Хавелунд и др. [985], 2000). Алгоритм RSA шифрования с открытым ключом и алгоритм сопоставления строк Бойера–Мура также были проверены аналогичным образом (Бойер и Мур [282], 1984).

Резюме

В этой главе представлены анализ логического вывода в логике первого порядка и несколько различных алгоритмов его выполнения.

- В первом подходе используются правила логического вывода (**конкретизации высказывания квантора всеобщности** и **конкретизации высказывания квантора существования**) для преобразования задачи логического вывода в форму **пропозициональной логики**. Как правило, этот подход характеризуется очень низким быстродействием.
- Использование **унификации** для выявления подходящих подстановок для переменных позволяет устранить этап конкретизации в доказательствах логики первого порядка, в результате чего этот процесс становится гораздо более эффективным.
- В поднятой версии **правила отделения** унификация применяется для получения естественного и мощного правила логического вывода — **обобщенного правила отделения**. В алгоритмах **прямого логического вывода** и **обратного логического вывода** это правило применяется к множествам определенных выражений.
- Обобщенное правило отделения является полным применительно к определенным выражениям, но проблема логического следствия остается **полуразрешимой**. Для баз данных в формате языка **Datalog**, состоящих из определенных выражений без функций, проблема логического следствия разрешима.
- Прямой логический вывод используется в **дедуктивных базах данных**, где он может сочетаться с реляционными операциями баз данных. Этот метод также применяется в **продукционных системах**, обеспечивающих эффективное обновление при наличии очень больших наборов правил. Прямой логический вывод для баз данных в формате языка Datalog является полным и выполняется за время, определяемое полиномиальной зависимостью.
- Обратный логический вывод используется в **системах логического программирования**, в которых реализована сложная технология компиляции для обеспечения очень быстрого логического вывода. К недостаткам обратного логического вывода можно отнести избыточные этапы логического вывода и бесконечные циклы, которые можно устранить, используя запоминание.

- В языке **Prolog**, в отличие от логики первого порядка, используются допущения замкнутости мира и уникальности имен, а также отрицания в качестве неудачи. Это делает Prolog более практичным языком программирования, но выводит его за рамки чистой логики.
- Обобщенное правило логического вывода на основе **резолюции** позволяет создать полную систему доказательства для логики первого порядка с использованием баз знаний в конъюнктивной нормальной форме.
- Для систем с резолюцией существует несколько стратегий сокращения пространства поиска без потери его полноты. Одним из наиболее важных и сложных вопросов является работа с равенствами, — было показано, как приемы **демодуляции** и **парамодуляции** могут использоваться для успешного достижения этой цели.
- Эффективные средства автоматического доказательства теорем на основе резолюции использовались для доказательства интересных математических теорем, а также для проверки и синтеза программных и аппаратных средств.

Библиографические и исторические заметки

Готтлоб Фреге, разработавший в 1879 году полную логику первого порядка, построил свою систему логического вывода на большой коллекции логически правильных схем и единственном правиле логического вывода — правиле отделения, Modus Ponens. Уайтхед и Рассел ([2331], 1910) описали так называемые правила прохождения (*rules of passage*) (фактически этот термин принадлежит Эрбрану ([1013], 1930)), которые используются для перемещения кванторов в переднюю часть формул. Торальф Сколем ([2081], 1920) достаточно своевременно предложил понятия сколемовских констант и сколемовских функций. И как это ни странно, именно Сколем представил понятие универсума Эрбрана (Сколем [2082], 1928).

Критически важную роль в разработке методов автоматизированных рассуждений сыграла теорема Эрбрана ([1013], 1930). Эрбран также является изобретателем операции **унификации**. Гёдель ([873], 1930), опираясь на идеи Сколема и Эрбрана, сумел показать, что для логики первого порядка имеется полная процедура доказательства. Алан Тьюринг ([2233], 1936) и Алонсо Черч ([431], 1936) одновременно продемонстрировали, используя очень разные доказательства, что задача определения общезначимости в логике первого порядка не имеет решения. В превосходной книге Эндертона ([688], 1972) все эти результаты описаны в строгой, но труднодоступной для понимания манере.

Абрахам Робинсон сделал предположение о том, что систему автоматизированных рассуждений можно построить с использованием пропозиционализации и теоремы Эрбрана, и Пол Гилмор ([859], 1960) написал первую соответствующую этому утверждению программу. Дэвис и Патнем ([544], 1960) ввели метод пропозиционализации, обсуждавшийся в разделе 9.1. Правиц ([1820], 1960) разработал

ключевую идею использования пропозициональных противоречивостей для управления процессом поиска и генерации термов из универсума Эрбрана, только если это необходимо для определения пропозициональной противоречивости. Эта идея привела Джона Алана Робинсона (не имеет родственных отношений с Абрахамом Робинсоном) к разработке резолюции (Робинсон, 1965).

Метод резолюции был адаптирован Корделлом Грином и Бертрамом Рафаэлем ([921], 1968) для создания систем поиска ответов на вопросы. В ранних реализациях систем искусственного интеллекта большие усилия направлялись на разработку структур данных, которые должны были обеспечить эффективную выборку фактов; эти работы описаны в книгах по программированию для искусственного интеллекта (Чарняк и др. [397], 1987; Норвиг [1700], 1992; Форбус и Де Клер [752], 1993). В начале 1970-х годов в области искусственного интеллекта прочно занял свои позиции метод **прямого логического вывода**, рассматривавшийся как легко понимаемая альтернатива методу резолюции. Приложения ИИ обычно работают с большим количеством правил, поэтому было важно разработать эффективную технологию сопоставления правил, в частности для инкрементных обновлений.

Для поддержки таких приложений была разработана технология **продукционных систем**. Язык продукционных систем OPS-5 (Форги [759], 1981; Браунстон и др. [325], 1985) использовался для экспертной системы R1, предназначенной для мини-компьютеров, и включал процесс согласования с помощью rete-алгоритма (Форги [760], 1982). Краска и соавт. ([1303], 2017) описывают, как нейронные сети могут быть обучены для поддержки эффективной схемы индексации для определенных наборов данных.

Когнитивная архитектура SOAR (Лейрд и др. [1339], 1987; Лейрд [1341], 2008) была разработана для обработки очень больших наборов правил — до миллиона (Доренбос [635], 1994). Примерами применения SOAR являются система управления тренажером, моделирующим самолет-истребитель (Джонс и др. [1145], 1998), система управления полетами (Тейлор и др. [2186], 2007), персонажи ИИ в компьютерных играх (Уинтермьют и др. [2363], 2007), а также системы обучения и тренировки для военнослужащих (Урей и Джонс [2386], 2005).

Исследования в области дедуктивных баз данных начались с симпозиума, организованного в Тулузе в 1977 году для специалистов в области логического вывода и систем баз данных (Галлар и Минкер [806], 1978). Важные работы Чандры и Харела ([388], 1980) и Ульмана ([388], 1980) привели к принятию языка **Datalog** как стандартного языка баз данных. Разработка технологии **магических множеств** для перезаписи правил (Бансильон и др. [121], 1986) позволила прямому логическому выводу воспользоваться преимуществом целенаправленности, свойственному обратному логическому выводу.

Быстрое развитие Интернета привело к повышению доступности крупных интерактивных баз данных. Это способствовало повышению интереса к интеграции нескольких баз данных в согласованное пространство данных (Галеви [949], 2007). Краска и соавт. ([1303], 2017) сообщили об ускорении работ до 70% за счет

использования машинного обучения с целью создания **обученных индексных структур** для эффективного поиска данных.

Обратный логический вывод впервые был применен в языке PLANNER (Хьюит [1018], 1969). В то же время в 1972 году французский исследователь Ален Колмерор разработал и реализовал язык **Prolog**, предназначенный для синтаксического анализа текста на естественном языке, — первоначально выражения Prolog предназначались для использования в качестве правил контекстно-свободной грамматики (Руссе [1921], 1975; Колмерор и др. [466], 1973).

Основная часть теоретических основ логического программирования была разработана Робертом Ковальски из Лондонского имперского колледжа в сотрудничестве с Колмерором (исторический обзор приведен в работах Ковальски ([1297], 1988) и Колмерора и Руссе ([465], 1993)). Эффективные компиляторы Prolog чаще всего основаны на модели абстрактной машины Уоррена (*Warren Abstract Machine* — WAM), разработанной Дэвидом Г.Д. Уорреном ([2297], 1983). Ван Рой ([2260], 1990) показал, что в отношении быстродействия программы на языке Prolog могут конкурировать с программами на языке C.

Методы предотвращения нежелательного зацикливания в рекурсивных логических программах были разработаны независимо Смитом и соавт. ([2090], 1986), а также Тамаки и Сато ([2171], 1986). Последняя статья также включала информацию о методе запоминания для логических программ, интенсивно разрабатываемого Дэвидом С. Уорреном под названием **табулированное логическое программирование**. Свифт и Уоррен ([2165], 1994) показали, как дополнить машину WAM с целью поддержки табулирования, что позволило программам на языке Datalog достичь быстродействия, на порядок превышающего быстродействие дедуктивных систем баз данных с прямым логическим выводом.

Первые теоретические работы по логическому программированию в ограничениях были выполнены Джаффаром и Лассе ([1123], 1987). Джаффар и соавт. ([1124], 1992) разработали систему CLP(R) для обработки ограничений с действительными значениями. В настоящее время существуют коммерческие продукты, предназначенные для решения крупномасштабных задач конфигурации и оптимизации методами программирования в ограничениях. Одним из самых известных является система ILOG (Юнкер [1161], 2003). Методика программирования набора ответов (Гелфонд [830], 2008) расширяет язык Prolog, позволяя использовать дизъюнкции и отрицания.

Объем учебной литературы по логическому программированию и языку Prolog очень велик и включает книги Шохама ([2053], 1994), Братко ([293], 2009), Клоксина ([453], 2003) и Клоксина и Меллиша ([452], 2003). До закрытия в 2000 году официальным журналом для публикаций в этой области был *Journal of Logic Programming*; теперь вместо него выпускается журнал *Theory and Practice of Logic Programming*. К числу основных конференций по логическому программированию относятся *International Conference on Logic Programming* (ICLP) и *International Logic Programming Symposium* (ILPS).

Исследования в области **автоматического доказательства математических теорем** начались еще до того, как были разработаны полные логические системы первого порядка. В разработанной Гербертом Гелернтером программе Geometry Theorem Prover ([828], 1959) использовались методы эвристического поиска в сочетании с диаграммами для отсечения ложных подцелей. С помощью этой программы удалось обосновать некоторые весьма сложные результаты исследований в евклидовой геометрии. Правила **демодуляции** и **парамодуляции** для формирования рассуждений с учетом отношения равенства были предложены соответственно Возом и соавт. ([2383], 1967) и Возом и Робинсоном ([2382], 1968). Эти правила были также независимо разработаны в контексте систем перезаписи термов (Кнут и Бендикс [1247], 1970). Внедрение средств учета отношения равенства в алгоритм унификации было осуществлено Гордоном Плоткином ([1798], 1972). В работе ([1151], 1991) приведен обзор средств унификации с учетом отношения равенства с использованием перезаписи термов. Общий обзор методов унификации дан Баадером и Снайдером ([95], 2001).

Для метода резолюции был предложен целый ряд стратегий управления, начиная со стратегии предпочтения единичного выражения (Воз и др. [2379], 1964). Позднее, в статье [2380] (1965) Воз и соавт. предложили стратегию с использованием множества поддержки, позволяющую обеспечить определенную целенаправленность резолюции. Линейная резолюция впервые была предложена Ловеландом в [1448] (1970). Генесерет и Нильссон в [836] (1987, глава 5) дали краткий, но исчерпывающий анализ всего разнообразия стратегий управления. Алеми и соавт. ([29], 2017) показали, как в системе DEEPMATH глубокие нейронные сети используются для выбора тех аксиом, которые с наибольшей вероятностью приведут к получению доказательства при передаче их традиционной системе доказательства теорем. В некотором смысле нейронная сеть играет здесь роль интуиции математика, а система доказательства теорем лишь выполняет техническую математическую экспертизу. Лоос и соавт. в [1442] (2017) показали, что этот подход может быть расширен с целью оказания помощи в руководстве поиском, что позволит доказать больше теорем.

Основным справочником по программе автоматического доказательства теорем Бойера–Мура является книга *A Computational Logic* (Бойер и Мур [281], 1979). Стикел в [2129] (1992) описывает систему PTTP (*Prolog Technology Theorem Prover*), в которой сочетаются преимущества компиляции языка Prolog с полнотой устранения моделей. Еще одной широко применяемой программой автоматического доказательства теорем, основанной на этом подходе, является SETHEO (Лец и др. [1391], 1992). Эффективной программой автоматического доказательства теорем, реализованной всего лишь в 25 строках на языке Prolog, является LeanTaP (Бекерт и Посегга [154], 1995). В статье Вейденбаха [2304] (2001) описана программа SPASS — одна из сильнейших современных программ автоматического доказательства теорем. Наиболее успешной программой доказательства теорем по результатам ежегодных соревнований последних лет является VAMPIRE (Рязанов и

Воронков [1876], 2002). Система Coq (Берто и др. [199], 2004) и решатель уравнений E (Шульц [2009], 2004) также уже доказали свою ценность в качестве инструментов доказательства правильности.

Программы доказательства теорем также уже использовались для автоматического синтеза и проверки программного обеспечения. Как пример можно привести программное обеспечение управления капсулой Orion (Лоури [1453], 2008) и другими космическими аппаратами NASA (Денни и др. [607], 2006). Правильность разработки 32-битного микропроцессора FM9001 была доказана с помощью системы доказательства теорем NQTHM (Хант и Брок [1098], 1992).

В рамках конференции *Conference on Automated Deduction* (CADE) проводится ежегодный конкурс по автоматизированному доказательству теорем. Сатклифф в статье [2150] (2016) описывает конкурс 2016 года, — наилучшие результаты показали системы Vampire (Рязанов и Воронков [1876], 2002), Prover9 (Сабри [1953], 2015), а также обновленная версия программы E (Шульц [2010], 2013). Видейк в [2336] (2003) сравнивает возможности 15 систем доказательства математических теорем. Полезным инструментом сравнения производительности систем является библиотека задач по доказательству теорем TPTP (*Thousands of Problems for Theorem Provers*) (Сатклифф и Саттнер [2151], 1998; Сатклифф и др. [2152], 2006).

Программы доказательства теорем позволили достичь новых математических результатов, которых людям-математикам не удавалось достичь на протяжении десятилетий, — об этом можно прочитать в книге *Automated Reasoning and the Discovery of Missing Elegant Proofs* (Воз и Пипер [2385], 2003). Программа Sam (*Semi-Automated Mathematics*) была первой, доказавшей лемму в теории решеток (Гвард и др. [929], 1969). Программа Aura также нашла ответы на некоторые вопросы, остававшиеся открытыми в различных областях математики (Воз и Винкер [2384], 1983). Программа доказательства теорем Бойера–Мура (Бойер и Мур [281], 1979) была использована Натараяном Шанкаром для построения формального доказательства теоремы о неполноте Гёделя (Шанкар [2038], 1986). Система Nuprl доказала парадокс Жирара (Хоув [1077], 1987) и лемму Хигмана (Мурти и Расселл [1646], 1990).

В 1933 году Герберт Роббинс предложил простой набор аксиом — ►**алгебру Роббинса**, — который, по-видимому, определяет булеву алгебру, но никаких доказательств этому найти не удавалось (несмотря на серьезные усилия Альфреда Тарского и других математиков), пока программа EQP (версия программы Otter) не вычислила необходимое доказательство (МакКун [1539], 1997). Бенцмюллер и Палео ([177], 2013) использовали систему доказательства теорем в логике высшего порядка для проверки предложенного Гёделем доказательства существования Бога. Теорема Кеплера о сферической упаковке была доказана Томасом Хейлзом ([947], 2005) с помощью некоторых сложных компьютерных вычислений, но это доказательство было не полностью принято, пока формальное доказательство не было получено с помощью программ-помощников HOL Light и Isabelle (Хейлз и др. [948], 2017).

К учебникам, ориентированным на автоматические рассуждения и вывод, относятся классический *Symbolic Logic and Mechanical Theorem Proving* (Чанг и Ли [389], 1973), а также более поздние работы Даффи ([662], 1991), Воза и соавт. ([2381], 1992), Бибела ([210], 1993) и Кауфманна и соавт. ([1198], 2000). Основным журналом для публикаций в области автоматического доказательства теорем является *Journal of Automated Reasoning*, а главными конференциями — ежегодно проводимая конференция *Conference on Automated Deduction* (CADE) и конференция *International Joint Conference on Automated Reasoning* (IJCAR). В книге *Handbook of Automated Reasoning* (Робинсон и Воронков [1899], 2001) собраны статьи по исследованиям в области автоматизированных рассуждений. В книге Маккензи *Mechanizing Proof* ([1473], 2004) описываются история и технология систем доказательства теорем для популярной аудитории.

Упражнения

9.1. Докажите, что процедура конкретизации с помощью квантора всеобщности является непротиворечивой и что процедура конкретизации с помощью квантора существования позволяет получить базу знаний, эквивалентную с точки зрения логического вывода.

9.2. Кажется вполне обоснованным утверждение, что из отдельного факта $Likes(Jerry, IceCream)$ можно вывести высказывание $\exists x\ Likes(x, IceCream)$. Запишите общее правило логического вывода, правило введения квантора существования, позволяющее узаконить такой логический вывод. Тщательно сформулируйте условия, которым должны удовлетворять переменные и термы, участвующие в этом выводе.

9.3. Предположим, что база знаний содержит только одно высказывание, $\exists x\ AsHighAs(x, Everest)$. Какие из следующих фактов являются действительными результатами применения правила конкретизации высказывания квантора существования?

а) $AsHighAs(Everest, Everest)$.

б) $AsHighAs(Kilimanjaro, Everest)$.

в) $AsHighAs(Kilimanjaro, Everest) \wedge AsHighAs(BenNevis, Everest)$ (после двух применений).

9.4. Для каждой приведенной ниже пары атомарных высказываний укажите наиболее общий унификатор, если он существует.

а) $P(A, B, B), P(x, y, z)$.

б) $Q(y, G(A, B)), Q(G(x, x), y)$.

в) $Older(Father(y), y), Older(Father(x), John)$.

г) $Knows(Father(y), y), Knows(x, x)$.

9.5. Для каждой приведенной ниже пары атомарных высказываний укажите наиболее общий унификатор, если он существует.

а) $P(A, B, B), P(x, y, z)$.

б) $Q(y, G(A, B)), Q(G(x, x), y)$.

в) $Older(Father(y), y), Older(Father(x), Jerry)$.

г) $Knows(Father(y), y), Knows(x, x)$.

9.6. Рассмотрите решетки обобщения, приведенные на рис. 9.2.

а) Составьте решетку для высказывания *Employs*(*Mother*(*John*), *Father*(*Richard*)).

б) Составьте решетку для высказывания *Employs*(*IBM*, *y*) ("Компания IBM является нанимателем для всех"). Не забудьте включить запрос любого типа, который унифицируется с этим высказыванием.

в) Предположим, что каждое высказывание индексируется под каждым узлом в его решетке обобщения. Объясните, как следует поступить, если некоторые из этих высказываний содержат переменные. В качестве примера воспользуйтесь высказываниями, приведенными в пп. *а* и *б*, а также запросом *Employs*(*x*, *Father*(*x*)).

9.7. Запишите логическое представление для каждого из приведенных ниже высказываний, применимое для использования с обобщенным правилом отделения.

а) Лошади, коровы и свиньи — млекопитающие.

б) Рожденный лошадью — лошадь.

в) Синяя Борода — лошадь.

г) Синяя Борода — родитель Чарли.

д) Отношения "быть рожденным" и "быть родителем" — обратные.

е) У каждого млекопитающего есть родитель.

9.8. Приведенные ниже вопросы касаются проблем подстановки и сколемизации.

а) При наличии предпосылки $\forall x\ \exists y\ P(x, y)$ будет ли недопустимо утверждать, что $\exists q\ P(q, q)$. Приведите пример предиката P, где первое значение истинно, а второе ложно.

б) Предположим, что процедура логического вывода написана неправильно и проверка наличия ошибок не выполнена, поэтому она может унифицировать литерал типа $P(x, F(x))$ с $P(q, q)$. (Как уже упоминалось выше, большинство стандартных реализаций языка Prolog действительно это позволяют.) Покажите, что такая процедура логического вывода позволит сделать заключение $\exists y\ P(q, q)$ из предпосылки $\forall x\ \exists y\ P(x, y)$.

в) Предположим, что процедура, преобразующая высказывания логики первого порядка в форму выражений, неверно сколемизирует предпосылку $\forall x\ \exists y\ P(x, y)$ в $P(x, Sk_0)$), т.е. заменяет переменную y сколемовской константой Sk_0, а не сколемовской функцией от x. Покажите, что система логического вывода, использующая такую процедуру, также позволит вывести заключение $\exists q\ P(q, q)$ из предпосылки $\forall x\ \exists y\ P(x, y)$.

г) Распространенной ошибкой среди студентов является предположение, что при унификации вместо переменной можно подставить терм сколемовской константы. Так, например, они будут утверждать, что формулы $P(Sk_1)$ и $P(A)$ могут быть унифицированы при подстановке Sk_1/A. Приведите пример, когда такое действие приводит к неверному логическому выводу.

9.9. Этот вопрос относится только к хорновским базам знаний, таким как

$$P(F(x)) \Rightarrow P(x) \qquad Q(x) \Rightarrow P(F(x))$$
$$P(A) \qquad\qquad Q(B)$$

Пусть FC — алгоритм прямо логического вывода в ширину, в котором многократно суммируются все следствия выполняемых в настоящее время правил, а BC — алгоритм обратного логического вывода в глубину, слева направо, проверяющий выражения в порядке, определяемом базой знаний. Какие из следующих предположений верны?

а) Алгоритм FC выведет литерал $Q(A)$.
б) Алгоритм FC выведет литерал $P(B)$.
в) Если алгоритм FC не сможет вывести заданный литерал, то этот литерал не следует из базы знаний.
г) Алгоритм BC вернет *true* при запросе $P(B)$.
д) Если алгоритм BC не возвращает *true* для заданного литерала запроса, то этот литерал не следует из базы знаний.

9.10. Объясните, как записать любую конкретную формулировку задачи 3-SAT произвольного размера с использованием единственного определенного выражения логики первого порядка и не больше 30 базовых фактов.

9.11. Предположим, даны следующие аксиомы.

1. $0 \leq 3$
2. $7 \leq 9$
3. $\forall x \; x \leq x$
4. $\forall x \; x \leq x+0$
5. $\forall x \; x+0 \leq x$
6. $\forall x, y \; x+y \leq y+x$
7. $\forall w, x, y, z \; w \leq y \wedge x \leq z \Rightarrow w+x \leq y+z$
8. $\forall x, y, z \; x \leq y \wedge y \leq z \Rightarrow x \leq z$

а) Методом обратного логического вывода докажите высказывание $7 \leq 3+9$. (Безусловно, используйте только приведенные здесь аксиомы, а не все то, что знаете из арифметики.) Покажите только этапы, которые ведут к успеху, опустив все прочие, бесполезные для доказательства.
б) Методом прямого логического вывода докажите высказывание $7 \leq 3+9$. И вновь, покажите только те этапы, которые ведут к успеху.

9.12. Предположим, даны следующие аксиомы.

1. $0 \leq 4$
2. $5 \leq 9$
3. $\forall x \; x \leq x$
4. $\forall x \; x \leq x+0$
5. $\forall x \; x+0 \leq x$
6. $\forall x, y \; x+y \leq y+x$
7. $\forall w, x, y, z \; w \leq y \wedge x \leq z \Rightarrow w+x \leq y+z$
8. $\forall x, y, z \; x \leq y \wedge y \leq z \Rightarrow x \leq z$

а) Методом обратного логического вывода докажите высказывание $5 \leq 4+9$. (Безусловно, используйте только приведенные здесь аксиомы, а не все то, что знаете из арифметики.) Покажите только этапы, которые ведут к успеху, опустив все прочие, бесполезные для доказательства.
б) Методом прямого логического вывода докажите высказывание $5 \leq 4+9$. И вновь, покажите только те этапы, которые ведут к успеху.

9.13. Одной из известных детских английских загадок является следующая: "Братьев и сестер у меня нет, но отец этого человека — сын моего отца". С использованием правил из области семейных отношений (см. раздел 8.3.2) определите, кто этот человек, о котором говорится в загадке. Можете применять любые методы логического вывода, описанные в этой главе. Почему, по вашему мнению, эту загадку трудно отгадать сразу?

9.14. Предположим, что в логическую базу данных помещена часть данных переписи населения США с указанием возраста, города проживания, даты рождения и имени матери каждого лица с использованием номеров карточки социального страхования в качестве идентифицирующих констант для каждого лица. Таким образом, например, возраст Джорджа задается выражением $\{Age\}(\therefore$mbox443-{65}-{1282}}, {56}). Какая из приведенных ниже схем индексации S1–S5 позволяет эффективно находить ответы на каждый из запросов Q1–Q4 (при условии, что применяется обычный метод обратного логического вывода)?

S1. Индекс для каждого атомарного терма в каждой позиции.
S2. Индекс для каждого первого параметра.
S3. Индекс для каждого атомарного терма предиката.
S4. Индекс для каждой комбинации предиката и первого параметра.
S5. Индекс для каждой комбинации предиката и второго параметра и индекс для каждого первого (нестандартного) параметра.

Q1. $Age(443\text{-}44\text{-}4321, x)$
Q2. $ResidesIn(x, Houston)$
Q3. $Mother(x, y)$
Q4. $Age(x, 34) \wedge ResidesIn(x, TinyTownUSA)$

9.15. Можно предположить, что стандартизация раз и навсегда отличий всех высказываний в базе знаний позволит избежать проблемы конфликта переменных при унификации в процессе обратного логического вывода. Покажите, что для некоторых высказываний такой подход не применим. (*Подсказка.* Рассмотрите высказывание, одна часть которого унифицируется с другой.)

9.16. В этом упражнении для получения ответов на вопросы с помощью алгоритма обратного логического вывода используются высказывания, записанные при выполнении упражнения 9.7.

а) Нарисуйте дерево доказательства, сформированное исчерпывающим алгоритмом обратного логического вывода для запроса $\exists\ h\ \ Horse(h)$ ("Существует некоторая лошадь"), в котором выражения согласуются в указанном порядке.
б) Какие особенности этой проблемной области вы обнаружили?
в) Какое количество решений для h фактически следует из ваших высказываний?
г) Можете ли вы предложить способ поиска всех этих решений? (*Подсказка.* Вам может потребоваться обратиться к работе Смита и соавт. ([2090], 1986).)

9.17. Проследите за выполнением алгоритма обратного логического вывода, приведенного на рис. 9.6, при его применении для решения задачи доказательства преступления (см. раздел 9.3.1). Покажите, какую последовательность значений принимает переменная *goals*, и расположите эти значения в виде дерева.

9.18. Приведенный ниже код на языке Prolog определяет предикат P. (Не забывайте, что в языке Prolog термы, определяемые прописными буквами, — это *переменные*, а не константы.)

```
P(X,[X|Y])
   P(X,[Y|Z]) :- P(X,Z)
```

а) Приведите деревья доказательства и решения для запросов `P(A,[2,1,3])` и `P(2,[1,A,3])`.
б) Какую стандартную операцию со списками представляет предикат P?

9.19. Приведенный ниже код на языке Prolog определяет предикат `P`. (Не забывайте, что в языке Prolog термы, определяемые прописными буквами, — это *переменные*, а не константы.)

```
P(X,[X|Y])
    P(X,[Y|Z]) :- P(X,Z)
```

а) Приведите деревья доказательства и решения для запросов `P(A,[1,2,3])` и `P(2,[1,A,3])`.

б) Какую стандартную операцию со списками представляет предикат `P`?

9.20. В этом упражнении рассматривается применение сортировки в языке Prolog.

а) На языке Prolog напишите выражения, определяющие предикат `sorted(L)`, принимающий истинное значение тогда и только тогда, когда список `L` отсортирован в возрастающем порядке.

б) На языке Prolog напишите определение предиката `perm(L,M)`, принимающего истинное значение тогда и только тогда, когда `L` — перестановка `M`.

в) Определите предикат `sort(L,M)` (`M` — отсортированная версия `L`) с использованием предикатов `perm` и `sorted`.

г) Применяйте предикат `sort` ко все более и более длинным спискам, пока вам это не надоест. Какова временная сложность вашей программы?

д) На языке Prolog реализуйте более быстрый алгоритм сортировки, такой как сортировка вставкой или быстрая сортировка (*quicksort*).

9.21. В этом упражнении рассматривается рекурсивное применение правил перезаписи с использованием логического программирования. Правилом перезаписи (или **демодулятором**) является уравнение с указанным направлением применения. Например, правило перезаписи $x+0 \rightarrow x$ указывает, что любое выражение, которое согласуется с $x+0$, должно заменяться выражением x. Правила перезаписи являются ключевым элементом систем построения рассуждений с учетом отношения равенства. Для представления правил перезаписи используйте предикат `rewrite(X,Y)`. Например, приведенное выше правило перезаписи может быть представлено как `rewrite(X+0,X)`. Некоторые термы являются примитивными и не могут подвергаться дальнейшим упрощениям, поэтому для указания, что `0` — это примитивный терм, используйте запись `primitive(0)`.

а) Запишите определение предиката `simplify(X,Y)`, принимающего истинное значение, если `Y` — упрощенная версия `X`, т.е. к каким-либо подвыражениям `Y` больше не применимы какие-либо правила перезаписи.

б) Запишите коллекцию правил перезаписи для упрощения выражений, включающих арифметические операторы, и примените ваш алгоритм упрощения к некоторым примерам выражений.

в) Запишите коллекцию правил перезаписи для символического дифференцирования и примените их наряду с определенными вами правилами упрощения для дифференцирования и упрощения выражений, в которых есть арифметические выражения, включая возведение в степень.

9.22. В этом упражнении рассматривается реализация алгоритмов поиска на языке Prolog. Предположим, что предикат `successor(X,Y)` принимает истинное значение, если состояние `Y` является преемником состояния `X`, и что предикат `goal(X)` принимает истинное значение, если `X` — целевое состояние. Запишите определение для предиката `solve(X,P)`, который означает, что `P` — путь (список состояний), начинающийся

от X, заканчивающийся в целевом состоянии и состоящий из последовательности допустимых шагов, которые определены предикатом `successor`. Вы обнаружите, что простейшим способом решения этой задачи является поиск в глубину. Насколько просто будет ввести эвристическое управление поиском?

9.23. Предположим, что база знаний содержит только следующие хорновские выражения в логике первого порядка.

$Ancestor(Mother(x), x)$

$Ancestor(x, y) \wedge Ancestor(y, z) \Rightarrow Ancestor(x, z)$

Рассмотрим алгоритм прямого логического вывода, который завершается на j-й итерации, если база знаний KB содержит высказывание, которое унифицируется с запросом, а иначе добавляет в KB каждое атомарное высказывание, которое можно вывести из высказываний, уже находящихся в базе знаний после итерации $j-1$.

а) Для каждого из следующих запросов укажите, как завершится выполнение алгоритма: он даст ответ (если да, то запишите этот ответ) либо завершится, не дав ответа, либо никогда не завершится.

1. $Ancestor(Mother(y), John)$
2. $Ancestor(Mother(Mother(y)), John)$
3. $Ancestor(Mother(Mother(Mother(y))), Mother(y))$
4. $Ancestor(Mother(John), Mother(Mother(John)))$

б) Может ли алгоритм резолюции доказать высказывание $\neg Ancestor(John, John)$ на основании исходной базы знаний? Объясните, как или почему нет.

в) Предположим, что в базу знаний добавлено утверждение $\neg(Mother(x) = x)$ и алгоритм резолюции был дополнен правилами вывода для равенств. Какой теперь будет ответ на вопрос в п. *б*?

9.24. Пусть $\mathcal{L}$ — язык первого порядка с единственным предикатом $S(p, q)$, означающим, что "p бреет q", и примем, что область определения этой задачи — люди.

а) Рассмотрим высказывание "Существует человек P, который бреет каждого человека, который не бреет себя сам, и только тех людей, которые не бреют себя сами". Выразите это высказывание на языке $\mathcal{L}$.

б) Преобразуйте высказывание из п. *а*, представив его в форме выражений.

в) Методом резолюции постройте доказательство того, что выражения из п. *б* по своей сути несовместимы. (*Примечание.* Вам не потребуются никакие дополнительные аксиомы.)

9.25. Как метод резолюции можно использовать для демонстрации того, что некоторое высказывание является общезначимым? Невыполнимым?

9.26. Приведите пример двух выражений, к которым операция резолюции может быть применена двумя разными способами, давая два разных результата.

9.27. Из высказывания "Лошади — животные" следует, что "Голова лошади — голова животного". Продемонстрируйте, что этот логический вывод является допустимым, выполнив приведенные ниже этапы.

а) Преобразуйте предпосылку и вывод этого высказывания в язык логики первого порядка. Воспользуйтесь тремя предикатами: $HeadOf(h, x)$ (который означает, что "h — голова x"), $Horse(x)$ (быть лошадью) и $Animal(x)$ (быть животным).

б) Примените отрицание к заключению и преобразуйте предпосылку и отрицаемое заключение в конъюнктивную нормальную форму.

в) Воспользуйтесь правилом резолюции, чтобы показать, что заключение следует из предпосылки.

9.28. Из высказывания "Овцы — животные" следует, что "Голова овцы — голова животного". Продемонстрируйте, что этот логический вывод является допустимым, выполнив приведенные ниже этапы.

а) Преобразуйте предпосылку и вывод этого высказывания в язык логики первого порядка. Воспользуйтесь тремя предикатами: *HeadOf*(h, x) (который означает, что "h — голова x"), *Sheep*(x) (быть овцой) и *Animal*(x) (быть животным).

б) Примените отрицание к заключению и преобразуйте предпосылку и отрицаемое заключение в конъюнктивную нормальную форму.

в) Воспользуйтесь правилом резолюции, чтобы показать, что заключение следует из предпосылки.

9.29. Ниже приведены два высказывания на языке логики первого порядка.

A $\forall x \; \exists y \; (x \geq y)$

B $\exists y \; \forall x \; (x \geq y)$

а) Примем, что область определения переменных — все натуральные числа 0, 1, 2, ..., ∞ и что предикат "$\geq$" означает "больше или равно". При использовании такой интерпретации переведите высказывания **A** и **B** на естественный язык.

б) Является ли высказывание **A** истинным при этой интерпретации?

в) Является ли высказывание **B** истинным при этой интерпретации?

г) Является ли высказывание **B** логическим следствием **A**?

д) Является ли высказывание **A** логическим следствием **B**?

е) С использованием правила резолюции попытайтесь доказать, что высказывание **A** следует из высказывания **B**. Сделайте эту попытку, даже если считаете, что **A** не следует логически из **B**; продолжайте прилагать усилия до тех пор, пока доказательство не оборвется и вы не сможете его продолжить (поскольку оно оборвалось). Покажите унифицирующую подстановку для каждого этапа резолюции. Если доказательство закончилось неудачей, точно объясните, где, как и почему оно оборвалось.

ж) Теперь попробуйте доказать, что высказывание **B** следует из **A**.

9.30. Метод резолюции способен вырабатывать неконструктивные доказательства для запросов с переменными, поэтому приходится вводить специальные механизмы для извлечения только определенных ответов. Объясните, почему такая проблема не возникает при использовании баз знаний, содержащих только определенные выражения.

9.31. В этой главе говорилось, что метод резолюции не может использоваться для формирования всех логических следствий из некоторого множества высказываний. Позволяет ли какой-то другой алгоритм решить эту задачу?

ГЛАВА 10

Представление знаний

В этой главе показано, как представлять различные факты о реальном мире в форме, которая может быть использована для выполнения рассуждений и решения задач.

В предыдущих главах было показано, как агент, обладающий базой знаний, может делать выводы, позволяющие ему действовать надлежащим образом. В этой главе рассматривается вопрос о том, какой *контент* поместить в базу знаний такого агента, — как представлять факты об окружающем мире. В качестве языка представления будет использоваться логика первого порядка, но в последующих главах будут описаны и другие формализмы представления знаний, такие как иерархические сети задачи для рассуждений о планах (глава 11), байесовские сети для рассуждений при наличии неопределенности (глава 13), марковские модели для рассуждений во времени (глава 17) и глубокие нейронные сети для рассуждений о содержании изображений, звуков и других данных (глава 21). Но независимо от того, какой способ представления фактов об окружающем мире будет использован, их в любом случае потребуется обрабатывать, и из этой главы вы получите необходимое представление о том, как это делается.

В разделе 10.1 вводится идея общей онтологии, позволяющей организовать все, что существует в мире, в виде иерархии категорий. В разделе 10.2 рассматриваются основные категории объектов, веществ и мер. В разделе 10.3 обсуждаются способы представления событий, а в разделе 10.4 рассматриваются знания об убеждениях. Далее речь пойдет о технологиях рассуждений с использованием всего этого контента: в разделе 10.5 обсуждаются системы рассуждений, предназначенные для эффективного вывода с использованием категорий, а в разделе 10.6 — рассуждения с использованием информации по умолчанию.

10.1. Онтологическая инженерия

В экспериментальных, "игрушечных", проблемных областях выбор представления не столь важен; всегда есть множество подходящих вариантов. В сложных проблемных областях, таких как осуществление покупок в Интернете или управление автомобилем на реальных дорогах, требуются более общие и гибкие

способы представления. В этой главе показано, как создать такие представления, сосредоточившись на общих понятиях — таких, как *действия*, *время*, *физические объекты* и *убеждения*, — присутствующих во множестве различных проблемных областей. Способы представления таких абстрактных понятий иногда называют ▶ **онтологической инженерией**.

Безусловно, невозможно формально представить *все*, что существует в мире, какого бы размера ни была эта книга, однако можно оставить достаточно свободных мест, в которые будут укладываться новые знания о любой проблемной области. Например, определяется, что имеется в виду под понятием "физический объект", а внесение подробных сведений о различных типах объектов — роботах, телевизорах, книгах или о чем-то другом — оставляется на будущее. Аналогичным образом поступают разработчики рабочих сред объектно-ориентированного программирования (таких, как графическая среда разработки Java Swing), которые определяют лишь различные общие концепции, например окна, кнопки, события и т.д. Позднее на их основе пользователи-программисты описывают необходимые им более конкретные понятия, например окно сообщения, кнопка отмены, событие ввода символа с клавиатуры и т.д. Такая общая инфраструктура понятий называется ▶ **верхней онтологией**, поскольку принято соглашение составлять схемы онтологии, помещая общие понятия в верхней части, а более конкретные понятия — под ними, как показано на рис. 10.1.

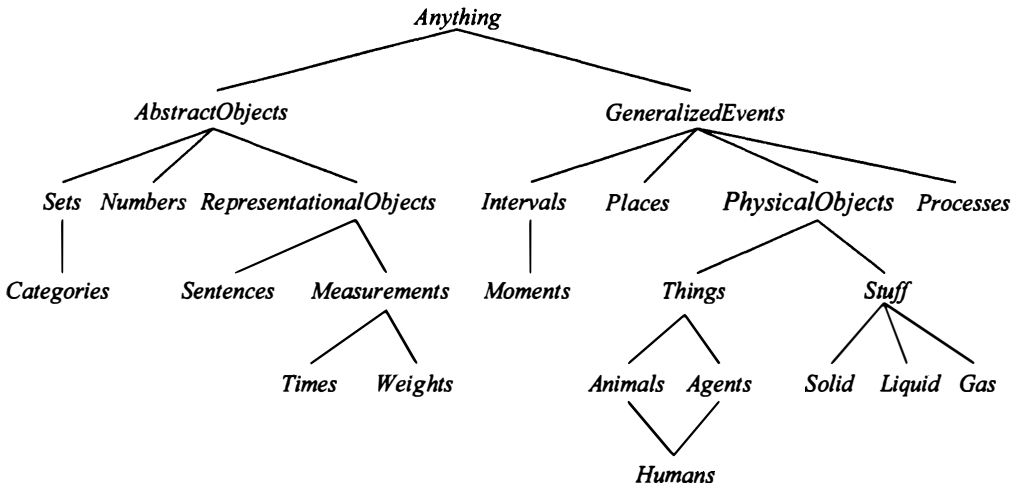


Рис. 10.1. Верхняя онтология мира, в которой показаны темы, рассматриваемые ниже в данной главе. Каждая связь ведет к нижнему понятию, являющемуся уточнением верхнего. Специализации не обязательно являются непересекающимися, например люди (*Humans*) являются и животными (*Animals*), и агентами (*Agents*). В разделе 10.3.2 будет показано, почему физические объекты (*PhysicalObjects*) входят в категорию обобщенных событий (*GeneralizedEvents*)

Прежде чем перейти к более подробному обсуждению онтологии, необходимо высказать одно важное предостережение. Мы решили использовать логику первого порядка для обсуждения содержания и организации знаний, хотя некоторые аспекты реального мира трудно представить в логике первого порядка. Основная сложность состоит в том, что почти все обобщения имеют исключения или соблюдаются только до определенной степени. Например, хотя правило "помидоры красные" является достаточно удобным, некоторые помидоры имеют зеленый, желтый или оранжевый цвет. Аналогичные исключения можно указать применительно почти ко всем общим утверждениям из этой главы. Способность учитывать исключения и неопределенность является чрезвычайно важной, но она ортогональна задаче понимания общей онтологии. По этой причине мы отложим обсуждение исключений до раздела 10.5, а более общей темы рассуждений при наличии неопределенности — до главы 12.

Для чего может применяться верхняя онтология? Еще раз рассмотрим онтологию электронных схем, описанную в разделе 8.4.2. В ней было принято большое количество упрощающих допущений, например полностью исключено понятие времени, сигналы являются постоянными и не распространяются, структура схемы остается неизменной. В более общей онтологии нужно было бы учитывать значения сигналов в определенные моменты времени и включать данные о длине проводников и задержках распространения. Эти расширения позволили бы моделировать временные свойства схемы, и в действительности подобные модели часто создаются разработчиками электронных схем.

Также можно было бы ввести более широкий перечень интересующих классов электронных элементов, например, описывая их технологию (ТТЛ, КМОП и т.д.), а также спецификацию ввода-вывода. Если бы потребовалось учитывать аспекты надежности или контроля качества, то необходимо было бы учесть возможность спонтанного изменения структуры схемы или свойств ее электронных компонентов. Чтобы учесть паразитные емкости, дополнительно пришлось бы так или иначе представить реальное расположение проводников на кристалле.

Обратившись к миру вампуса, можно прийти к аналогичным выводам. Хотя в описание этого мира включено время, оно имеет очень простую структуру: в мире ничего не происходит, кроме действий агента, и все изменения осуществляются мгновенно. Более общая онтология, которая лучше подходила бы для описания реального мира, должна была бы позволить нескольким одновременным изменениям распространяться во времени. Кроме того, для представления информации о том, в каких квадратах имеются ямы, использовался предикат *Pit*. Но можно было бы допустить существование ям разных типов, определив несколько объектов, принадлежащих к классу ям, каждый из которых характеризовался бы разными свойствами. Аналогичным образом в этом мире можно предусмотреть возможность существования других животных, помимо вампусов. Но тогда агент не имел бы возможности точно определить вид встретившегося животного из доступных ему результатов восприятия, поэтому потребовалось бы разработать биологическую

таксономию мира вампуса, позволяющую агенту прогнозировать поведение обитателей пещер на основании столь скупой информации.

При использовании любой онтологии специального назначения существует возможность вносить изменения, подобные приведенным выше, для достижения большей общности. В таком случае возникает очевидный вопрос — сходятся ли все эти онтологии к некоторой онтологии общего назначения? После столетий философских и математических исследований ответ на этот вопрос был найден — "Может быть, возможно". В данном разделе представлена одна из таких версий онтологии общего назначения, синтезирующая идеи прошедших веков. Онтологии общего назначения обладают двумя следующими основными особенностями, отличающими их от коллекций отдельных онтологий специального назначения.

- Любая онтология общего назначения должна быть в большей или меньшей степени применимой к любой специализированной проблемной области (с добавлением аксиом, специфических для этой проблемной области). Это означает, что в ней не следует ни детализировать, ни скрывать какие-либо вопросы конкретного представления.
- В любой достаточно значимой проблемной области различные области знаний должны быть *унифицированы*, поскольку процессы формирования рассуждений и решения задач могут одновременно включать несколько таких областей. Например, в системе робота, предназначенного для ремонта электронных схем, потребуется формировать рассуждения и о самих схемах — с точки зрения их электрических соединений и физической компоновки, — и о времени в отношении как анализа временных свойств схем, так и оценки его затрат на ремонт. Поэтому высказывания с описанием времени должны быть применимыми для комбинирования с высказываниями, описывающими пространственную компоновку, и одинаково хорошо представлять не только наносекунды и минуты, но и ангстремы и метры.

Немного забегая вперед, следует отметить, что все предпринятые усилия по разработке общей онтологической инженерии до сих пор имели лишь ограниченный успех. Ни одно из лучших приложений ИИ (как указано в главе 1) не использует общей онтологии — все они используют лишь инженерию знаний специального назначения и машинное обучение. Социальные или политические соображения могут затруднять конкурирующим сторонам возможность прийти к соглашению в отношении онтологии. Как говорит Том Грубер ([927], 2004), "Каждая онтология есть договор — социальное соглашение — между людьми, имеющими некоторые общие мотивы в ее совместном использовании". Когда конкурирующие интересы перевешивают мотивацию совместного использования, общей онтологии быть не может. Чем меньше заинтересованных сторон, тем легче создать онтологию, а значит, создать онтологию общего назначения намного труднее, чем онтологию ограниченного назначения, например такую, как Open Biomedical Ontology (*открытая*

биомедицинская онтология) (Смит и др. [2089], 2007). Те онтологии, которые уже существуют, были созданы четырьмя различными способами.

1. Командой обученных онтологов или логиков, спроектировавших онтологию и сформулировавших ее аксиомы. Система CYC была построена в основном таким образом (Ленат и Гуха [1384], 1990).
2. Импортом категорий, атрибутов и значений из существующей базы данных или баз данных. Система DBPEDIA была создана посредством импорта структурированных фактов из Википедии (Бизер и др. [226], 2007).
3. Посредством разбора текстовых документов и извлечения из них информации. Система TEXTRUNNER была создана методом прочтения большого количества веб-страниц (Банко и Этциони [125], 2008).
4. Соблазнением неквалифицированных любителей вводить данные, исходя из соображений здравого смысла. Система OPENMIND была создана добровольцами, предлагавшими факты на английском языке (Сингх и др. [2076], 2002; Шкловски и Гил [418], 2005).

Можно привести такой пример: в графе знаний Google используется полуструктурированный контент из Википедии, который комбинируется с другим контентом, собранным в Интернете под руководством человека. Граф содержит более 70 миллиардов фактов и обеспечивает ответы примерно на треть запросов в Google (Донг и др. [634], 2014).

10.2. Категории и объекты

Классификация объектов по ▶**категориям** является жизненно важным элементом любого способа представления знаний. Хотя взаимодействие с миром происходит на уровне отдельных объектов, ➔ *формирование рассуждений в большинстве случаев выполняется на уровне категорий.* Например, покупатель чаще всего ставит перед собой цель купить просто баскетбольный мяч, а не баскетбольный мяч какой-то конкретной марки, допустим, BB9. Категории также могут использоваться для предсказания свойств объектов после того, как они были классифицированы. Сделав заключение о присутствии некоторых объектов на основании входных данных сенсоров, можно определить их принадлежность к категориям, исходя из воспринятых свойств этих объектов, а затем воспользоваться информацией о категории для составления прогнозов, касающихся этих объектов. Например, на основании того, что объект имеет окраску в виде чередующихся зеленых и белых полос, округлую форму и размер чуть больше футбольного мяча, можно сделать вывод, что это арбуз, и на основании этого прийти к заключению, что его можно использовать в качестве десерта.

Для представления категорий в логике первого порядка доступны два варианта: предикаты и объекты. Это означает, например, что для баскетбольных мячей

можно либо применить предикат *Basketball*(*b*), либо ▶**овеществить**[1] (*reify*) всю эту категорию в виде некоторого объекта — *Basketballs*. В последнем случае появляется возможность использовать высказывание *Member* (*b*, *Basketballs*) (которое далее мы будем сокращенно записывать как $b \in Basketballs$) в качестве утверждения, что *b* — элемент категории баскетбольных мячей. Высказывание *Subset* (*Basketballs*, *Balls*), в сокращенном написании — $Basketballs \subset Balls$, может использоваться в качестве утверждения, что *Basketballs* — это ▶**подкатегория** или подмножество всех мячей *Balls*. Далее мы будем использовать термины "подкатегория", "подкласс" и "подмножество" как взаимозаменяемые.

Категории позволяют организовать знания с помощью ▶**наследования**. Если известно, что все экземпляры категории *Food* (*пища*) съедобны, и даны утверждения, что *Fruit* (*фрукты*) — это подкласс класса *Food*, а *Apples* (*яблоки*) — подкласс класса *Fruit*, то можно сделать вывод, что каждое яблоко съедобно. Говорят, что отдельные яблоки **наследуют** свойство съедобности, в данном случае в силу своей принадлежности к категории *Food*.

Отношения между отдельными классами и подклассами позволяют организовать категории в ▶**таксономическую иерархию** или **таксономию**. Таксономии широко использовались в прикладных науках уже многие столетия. Крупнейшая из таких таксономий организует около 10 миллионов известных видов животных (большинство из которых является насекомыми[2]), как еще существующих, так и уже исчезнувших, в единую иерархическую систему. В библиографии разработана таксономия всех областей знаний, закодированная в виде десятичной системы Дьюи, а налоговые органы и другие правительственные организации разработали обширные таксономии профессий и коммерческих продуктов.

Логика первого порядка позволяет легко формулировать факты о категориях, либо связывая категории с объектами, либо применяя к их элементам кванторы, как показано ниже.

- Объект является членом категории.

$$BB_9 \in Basketballs$$

- Категория является подклассом другой категории.

$$Basketballs \subset Balls$$

- Все элементы категории имеют некоторые свойства.

$$(x \in Basketballs) \Rightarrow Spherical(x)$$

---

[1] Это понятие означает превращение некоторого определения в объект. Англоязычный вариант термина "овеществление", *reification*, происходит от латинского слова *res*. Джон Маккарти предложил термин-кальку "thingification", но он так и не прижился.

[2] В ответ на вопрос "Что можно сказать о Создателе, исходя из наших знаний о природе?" биолог Дж. Б. С. Холдейн ответил: "Он чрезмерно увлечен жуками".

- Элементы категории могут быть распознаны по некоторым свойствам.

$$Orange(x) \wedge Round(x) \wedge Diameter(x) = 9{,}5'' \wedge x \in Balls \Rightarrow x \in Basketballs$$

- Вся категория в целом имеет некоторые свойства.

$$Dogs \in DomesticatedSpecies$$

Обратите внимание, что поскольку *Dogs* (*Собаки*) является и категорией, и элементом категории *DomesticatedSpecies* (*Домашние животные*), то последняя должна быть категорией категорий. Безусловно, есть исключения из многих перечисленных выше правил (например, проколотые баскетбольные мячи не являются сферическими), тем не менее обсуждение любых исключений мы пока отложим.

Хотя отношения между подклассами и элементами являются наиболее важными для категорий, необходимо также иметь возможность формулировать отношения и между теми категориями, которые не являются подклассами друг друга. Например, если будет отмечено, что *Ants* (*муравьи*) и *Flies* (*мухи*) — подклассы класса *Insects* (*насекомые*), то это не будет означать, что ни один муравей не может быть одновременно и мухой. Две или более категорий являются ▶**непересекающимися** (*disjoint*), если они не имеют общих элементов. И даже если известно, что категории муравьев и мух не пересекаются, этого недостаточно, чтобы утверждать, что насекомое, не являющееся муравьем, обязательно должно быть мухой, если не сформулировано дополнительное утверждение, что муравьи и мухи образуют ▶**исчерпывающую декомпозицию** (*exhaustive decomposition*) категории насекомых. Исчерпывающую декомпозицию непересекающихся множеств принято называть ▶**сегментацией** (*partition*). Эти три понятия иллюстрируются в приведенных ниже примерах.

$$Disjoint(\{Animals, Vegetables\})$$
$$ExhaustiveDecomposition(\{Americans, Canadians, Mexicans\}, NorthAmericans)$$
$$Partition(\{Animals, Plants, Fungi, Protista, Monera\}, LivingThings).$$

(Обратите внимание на то, что исчерпывающая декомпозиция североамериканцев (*NorthAmericans*) не является сегментацией, поскольку некоторые люди имеют двойное гражданство.) Эти три предиката определены следующим образом:

$$Disjoint(s) \Leftrightarrow (\forall c_1, c_2 \; c_1 \in s \wedge c_2 \in s \wedge c_1 \neq c_2 \Rightarrow Intersection(c_1, c_2) = \{\}),$$
$$ExhaustiveDecomposition(s, c) \Leftrightarrow (\forall i \; i \in c \Leftrightarrow \exists c_2 \; c_2 \in s \wedge i \in c_2),$$
$$Partition(s, c) \Leftrightarrow Disjoint(s) \wedge ExhaustiveDecomposition(s, c).$$

Категории могут быть также *определены* путем указания необходимых и достаточных условий принадлежности к ним. Например, холостяк — это неженатый взрослый мужчина:

$$x \in Bachelors \Leftrightarrow Unmarried(x) \wedge x \in Adults \wedge x \in Males.$$

Как описано во врезке "Естественные разновидности", строгие логические определения для категорий обычно возможны лишь для искусственных формальных терминов, а не для обычных объектов. Но определения необходимы не всегда.

10.2.1. Физическая композиция

Идея о том, что один объект может составлять часть другого, для нас привычна. Нос — это часть лица, Румыния — часть Европы, а данная глава — часть этой книги. Для указания на то, что одна вещь является частью другой, используется общее отношение *PartOf*. Объекты могут группироваться в иерархии *PartOf*, напоминающие иерархию *Subset* (подмножество), например, как показано ниже:

$$PartOf(Bucharest, Romania),$$
$$PartOf(Romania, EasternEurope),$$
$$PartOf(EasternEurope, Europe),$$
$$PartOf(Europe, Earth).$$

Отношение *PartOf* является транзитивным и рефлексивным, т.е. для него справедливы следующие высказывания:

$$PartOf(x, y) \wedge PartOf(y, z) \Rightarrow PartOf(x, z),$$
$$PartOf(x, x).$$

Следовательно, можно сделать заключение $PartOf(Bucharest, Earth)$. Категории ►**составных объектов** часто характеризуются структурными отношениями между частями. Например, любое двуногое (*biped*) имеет две и только две ноги (*leg*), прикрепленные (*attached*) к телу (*body*):

$$\begin{aligned} Biped(a) \Rightarrow \exists\, l_1, l_2, b \;\; & Leg(l_1) \wedge Leg(l_2) \wedge Body(b) \wedge \\ & PartOf(l_1, a) \wedge PartOf(l_2, a) \wedge PartOf(b, a) \wedge \\ & Attached(l_1, b) \wedge Attached(l_2, b) \wedge \\ & l_1 \neq l_2 \wedge [\forall\, l_3 \;\; Leg(l_3) \wedge PartOf(l_3, a) \Rightarrow (l_3 = l_1 \vee l_3 = l_2)]. \end{aligned}$$

Применяемая здесь система обозначения понятия "две и только две" является довольно громоздкой; мы были вынуждены указать, что ног две, что ноги не являются одинаковыми и что если кто-то из двуногих будет утверждать, что у него есть третья нога, она в конечном итоге окажется одной из его двух ног. В разделе 10.5.2 будет показано, что формальная система, называемая описательной логикой, позволяет проще представить ограничения типа "две и только две".

Можно определить отношение *PartPartition*, аналогичное отношению *Partition* для категорий (см. упр. 10.10). Любой объект состоит из частей, принадлежащих к его категориям *PartPartition*, и может рассматриваться как получающий некоторые свойства от собственных частей. Например, масса составного объекта — это сумма масс его частей. Обратите внимание, что это утверждение не распространяется

на категории, не имеющие массы, даже если эти категории состоят из элементов, обладающих массой.

Также целесообразно определить составные объекты, имеющие определенные части, но не имеющие конкретной структуры. Например, может потребоваться сформулировать утверждение "Яблоки в этой сумке весят два килограмма". Может возникнуть соблазн приписать этот вес множеству яблок в сумке, но это было бы ошибкой, поскольку множество — это абстрактное математическое понятие, которое имеет элементы, но не имеет веса. Вместо этого необходимо ввести новое понятие, которое мы будем именовать ▶**совокупностью** (*bunch*). Например, если три яблока обозначить как $Apple_1$, $Apple_2$ и $Apple_3$, то выражение

$$BunchOf(\{Apple_1, Apple_2, Apple_3\})$$

будет определять составной объект, *частями* (а не элементами) которого являются три яблока. Затем эта совокупность может использоваться как обычный, хотя и не структурированный объект. Обратите внимание на то, что $BunchOf(\{x\}) = x$. Также $BunchOf(Apples)$ — это составной объект, состоящий из всех яблок, но его не следует путать с объектом $Apples$ — категорией или множеством всех яблок.

Можно определить понятие *BunchOf* в терминах отношения *PartOf*. Очевидно, что каждый элемент множества s — это часть объекта $BunchOf(s)$:

$$\forall x \quad x \in s \Rightarrow PartOf(x, BunchOf(s)).$$

Более того, $BunchOf(s)$ — *это наименьший объект, удовлетворяющий данному условию.* Иными словами, $BunchOf(s)$ должен быть частью любого объекта, который включает все элементы множества s в качестве части:

$$\forall y \ [\forall x \quad x \in s \Rightarrow PartOf(x, y)] \Rightarrow PartOf(BunchOf(s), y).$$

Эти аксиомы представляют собой пример общего метода, называемого ▶**логической минимизацией**, с помощью которого любой объект может быть определен как наименьший объект, удовлетворяющий определенным условиям.

Естественные разновидности

Некоторые категории имеют строгие определения, например некоторый объект рассматривается как треугольник тогда и только тогда, когда он является многоугольником с тремя сторонами. С другой стороны, большинство категорий в реальном мире не имеют четких определений; они называются категориями ▶**естественных разновидностей**. Например, помидоры, как правило, имеют ярко-красный цвет, округлую форму с углублением в верхней части, где был черенок, диаметр примерно от пяти до десяти сантиметров, тонкую, но прочную кожицу, под которой находятся сочная мякоть и семена. Тем не менее существуют многочисленные вариации: некоторые помидоры — желтые или оранжевые, незрелые помидоры — зеленые, бывают помидоры размером больше или меньше среднего, а все помидоры сорта "черри" одинаково малы. Вместо полного определения помидоров у нас есть лишь набор характеристик,

позволяющих выявлять объекты, которые, безусловно, представляют собой типичные помидоры, но не могут с уверенностью идентифицировать другие объекты этого рода. (Может быть, скоро появятся даже помидоры, покрытые пушком, как персики?) Такая ситуация создает проблему для логического агента. Агент не может быть уверен в том, что некоторый объект, характеристики которого он воспринял, действительно является помидором, и даже если бы он был в этом уверен, он не мог бы с уверенностью указать, какими свойствами типичных помидоров обладает данный объект. Эта проблема является неизбежным следствием необходимости действовать в частично наблюдаемой среде.

Один из полезных подходов состоит в том, чтобы отделить сведения, истинные для всех экземпляров некоторой категории, от сведений, истинных только для типичных экземпляров. Поэтому в дополнение к категории *Tomatoes* необходимо предусмотреть категорию $Typical(Tomatoes)$. Здесь функция *Typical* отображает категорию на подкласс, который содержит только типичные экземпляры:

$$Typical(c) \subseteq c.$$

Большая часть знаний о естественных разновидностях фактически относится к их типичным экземплярам:

$$x \in Typical(Tomatoes) \Rightarrow Red(x) \wedge Round(x).$$

Следовательно, мы получаем возможность формулировать полезные факты о категориях без точных определений. Сложности предоставления точных определений для большинства естественных категорий были глубоко исследованы Витгенштейном ([2369], 1953). Он использовал пример с *играми*, чтобы показать, что элементы любой категории обладают, скорее "семейным сходством", а не строгими характеристиками, необходимыми и достаточными для их классификации, — какие строгие общие характеристики можно найти в игре в шахматы, в догонялки, в вышибалы или в раскладывании пасьянсов?

Полезность понятия строгого определения была также поставлена под сомнение Квайном ([1831], 1953). Он указал, что даже определение "холостяка" как неженатого взрослого мужчины небесспорно, например вряд ли является корректным такое утверждение, как "Папа Римский — холостяк". Хотя это утверждение, строго говоря, не является *ложным*, сам способ употребления в нем несовместимых слов, безусловно, не заслуживает одобрения, поскольку вызывает у некоторых слушателей нежелательные *ассоциации*. Возможно, указанную проблему можно было бы решить, проводя различия между логическими определениями, подходящими для внутреннего представления знаний, и более утонченными критериями, касающимися корректного лингвистического словоупотребления. Последней цели можно было бы достичь, "фильтруя" утверждения, полученные после достижения первой. Кроме того, возможно, что неудачные результаты при лингвистической проверке правильности словоупотребления могут использоваться в качестве обратной связи для модификации внутренних определений, чтобы такая фильтрация стала ненужной.

10.2.2. Система мер

И в научных, и в любых прочих теориях мира объекты имеют высоту, массу, стоимость и т.д. Значения, применяемые для оценки этих свойств, называются ▶**мерами**. Обычные количественные меры можно представить довольно легко. Предположим, что Вселенная включает абстрактные "объекты мер", такие как *длина*, которая представляет собой длину следующего отрезка прямой: |————————————|. Эта длина может быть представлена (именована) как 1,5 дюйма или 3,81 сантиметра. Следовательно, одна и та же длина в нашем языке может иметь разные имена. Представить длину можно с помощью ▶**функции единиц измерения**, имеющей числовой аргумент. (Альтернативная схема рассматривается в упражнении 10.11.)

Если приведенному выше отрезку прямой присвоить имя L_1, то можно записать следующее:

$$Length(L_1) = Inches(1{,}5) = Centimeters(3{,}81).$$

Преобразование из одних единиц измерения в другие осуществляется путем приравнивания кратных значений одной единицы другой:

$$Centimeters(2{,}54 \times d) = Inches(d).$$

Аналогичные аксиомы могут быть записаны для фунтов и килограммов, секунд и суток, долларов и центов. Меры могут использоваться для описания объектов следующим образом:

$$Diameter(Basketball_{12}) = Inches(9{,}5),$$
$$ListPrice(Basketball_{12}) = \$(19),$$
$$Weight(BunchOf(\{Apple_1, Apple_2, Apple_3\})) = Pounds(2),$$
$$d \in Days \Rightarrow Duration(d) = Hours(24).$$

Обратите внимание на то, что запись $\$(1)$ — это *не* долларовая купюра, это цена. Можно иметь две долларовые купюры, но существует только один объект с именем $\$(1)$. Следует также отметить, что объекты $Inches(0)$ и $Centimeters(0)$ обозначают одну и ту же нулевую длину, но не идентичны другим нулевым мерам, таким как $Seconds(0)$.

Простые, количественные меры очень легко представить. Представление других мер является более сложным, поскольку для них не предусмотрена согласованная шкала значений. Упражнения характеризуются сложностью, десерты — тонким вкусом, а стихи — красотой, но эти качества не могут быть обозначены количественными характеристиками. Можно было бы, стремясь к полной описываемости всего, отказаться от этих свойств как бесполезных с точки зрения логических рассуждений или, что еще хуже, попытаться оценить красоту по числовой шкале. Но такое решение было бы грубой ошибкой, поскольку оно отнюдь не является необходимым. Наиболее важное свойство мер — не конкретные числовые значения, а тот факт, что меры могут быть *упорядочены*.

Хотя многие меры не являются числами, все еще остается возможность сравнивать их с помощью некоторого символа упорядочения, такого как ">". Например, можно утверждать, что упражнения, подготовленные Норвигом, более трудные, чем подготовленные Расселом, и что в случае сложных упражнений можно ожидать, что оценка за их выполнение окажется ниже:

$$e_1 \in Exercises \land e_2 \in Exercises \land Wrote(Norvig, e_1) \land Wrote(Russell, e_2) \Rightarrow Difficulty(e1) > Difficulty(e2),$$
$$e_1 \in Exercises \land e_2 \in Exercises \land Difficulty(e_1) > Difficulty(e_2) \Rightarrow ExpectedScore(e_1) < ExpectedScore(e_2).$$

Этого достаточно, чтобы можно было принять решение о том, какие упражнения следует выбирать, даже несмотря на то, что для оценки их сложности не использовались какие-либо числовые значения (однако для этого необходимо иметь возможность установить, кем были подготовлены те или иные упражнения). Такого рода монотонные отношения между мерами образуют основу для ▶**качественной физики** (*qualitative physics*) — одного из направлений искусственного интеллекта, в котором изучаются способы формирования рассуждений о физических системах без погружения в числовые расчеты и подробные уравнения. Качественная физика подробнее обсуждается в разделе с историческими заметками.

10.2.3. Объекты: вещи и вещества

Реальный мир можно рассматривать как состоящий из примитивных объектов (т.е. атомарных частиц) и сложных объектов, составленных из них. Проведение рассуждений на уровне больших объектов, таких как яблоки и автомобили, позволяет преодолеть сложности, связанные с учетом бесконечных количеств примитивных объектов, взятых по отдельности. Тем не менее существует значительная часть реальности, которая, по-видимому, не поддается никаким явным попыткам ▶**индивидуализации** — деления на отдельные объекты. Присвоим этой части мира универсальное имя — ▶**вещество**. Например, предположим, что перед исследователем находится немного масла и муравьед. Аналитик может утверждать, что перед ним один муравьед, но не в состоянии указать какое-то явное количество "масло-объектов", поскольку любая часть масло-объекта — это также масло-объект, по крайней мере до тех пор, пока не удастся дойти до действительно очень мелких его частей. В этом и состоит основное различие между *веществами* и *объектами*. Если разрезать муравьеда надвое, двух муравьедов, к сожалению, не получится.

Во многих языках, в том числе в русском и английском, проводится четкое различие между *веществами* и *объектами*. Например, вполне нормально сказать "вижу двух муравьедов", но выражение "мне два масла" едва ли услышишь где-либо, кроме заводской столовой. Лингвисты проводят различие между ▶**исчисляемыми существительными**, такими как муравьеды, ямы и теоремы, и ▶**неисчисляемыми существительными**, такими как масло, вода и энергия. Несколько

конкурирующих онтологий утверждают, что в них учитывается это различие. В данной главе будет описана только одна из них, а остальные рассматриваются в разделе исторических заметок.

Чтобы правильно представить понятие *вещества*, начнем с очевидного. В своей онтологии как объекты мы должны воспринимать по меньшей мере достаточно большие количества или "куски" *вещества*, позволяющие определить способы взаимодействия с этими объектами. Например, в качестве куска можно рассматривать то масло, которое осталось в масленке от вчерашнего ужина, — его можно взять, взвесить, продать или проделать с ним какие-то другие действия. В этом смысле данный кусок масла представляет собой объект, вполне аналогичный муравьеду. Назовем его $Butter_3$. Кроме того, определим категорию *Butter*. Говоря неформально, ее элементами будут все те объекты, о которых можно сказать "Это масло", включая $Butter_3$. Если не принимать во внимание некоторые предостережения об очень малых частях вещества, которые мы пока опустим, то любая часть масло-объекта также представляет собой масло-объект:

$$b \in Butter \wedge PartOf(p, b) \Rightarrow p \in Butter.$$

Далее можно отметить тот факт, что масло плавится при температуре около 30 градусов по Цельсию:

$$b \in Butter \Rightarrow MeltingPoint(b, Centigrade(30)).$$

Продолжая описание, можно указать, что масло — желтое, имеет плотность меньше воды, размягчается при комнатной температуре, содержит много жиров и т.д. С другой стороны, масло не имеет конкретного размера, формы или веса. Дополнительно могут быть определены более специализированные категории масла, такие как *UnsaltedButter* (несоленое масло), которая также представляет некоторый вид *вещества*. Но обратите внимание: категория *PoundOfButter* (фунт масла), включающая в качестве элементов все масло-объекты, весящие один фунт, уже не является *веществом*! Если фунт масла разрезать пополам, то, к сожалению, два фунта масла мы не получим.

Из этого можно сделать такой вывод: у объектов существуют некоторые свойства, которые являются ▶ **внутренними**, — они принадлежат к каждой части объекта, а не к самому объекту в целом. Разрезав кусок вещества пополам, получим две части, обладающие одним и тем же набором внутренних свойств, таких как плотность, точка кипения, запах, цвет, принадлежность и т.д. С другой стороны, ▶ **внешние** свойства объектов — вес, длина, форма и т.д. — не сохраняются после его разделения на части. Категория объектов, которая включает в свое определение только *внутренние* свойства, характеризует вещество и обозначается качественными, неисчисляемыми существительными, а класс, включающий в свое определение *любые* внешние свойства, характеризует объект, экземпляр, и обозначается количественными, исчисляемыми существительными. Категории *Stuff* и *Thing* — это наиболее общие категории вещества и объекта соответственно.

10.3. События

В разделе 7.7.1 обсуждались действия: что-то, что происходит, случается, — например, $Shoot_t$, и флюенты: аспекты мира, которые могут изменяться, — например, $HaveArrow_t$. И то, и другое представлялось в виде высказываний, а аксиомы определения преемника использовались для того, чтобы утверждать, что флюента будет истинной в момент времени $t+1$, если действие в момент времени t заставит ее принять значение *true*, либо она уже имела это значение в момент времени t и действие не привело к тому, что она изменила свое значение на *false*. Все это подходило для мира, в котором действия являются дискретными, мгновенными, выполняются однократно и не подвержены изменениям в отношении того, как они будут выполнены (т.е. есть только один вариант действия *Shoot* и не существует никаких различий в том, как стрелять — быстро, медленно, нервно и т.д.).

Но если перейти от упрощенных проблемных областей к реальному миру, то придется иметь дело с гораздо более широким спектром действий или событий.[3] Обратимся к непрерывному действию, такому как заполнение ванны водой. Аксиома определения преемника может лишь сообщить нам, что ванна была пустой до выполнения действия и стала наполненной после его выполнения, но ничего не может сказать о том, что происходит *во время* выполнения действия. Также с ее помощью будет непросто описать два действия, происходящих одновременно, например чистку зубов в ожидании наполнения ванны. Чтобы справиться с такими ситуациями, необходимо использовать другой подход, известный как ▶ **исчисление событий** (*event calculus*).

Объектами в исчислении событий являются события, флюенты и точки времени. Так, запись $At(Shankar, Berkeley)$ представляет флюенту: объект, указывающий на факт, что Шанкар находится в Беркли. Событие E_1, указывающее на факт, что Шанкар летит из Сан-Франциско в Вашингтон, округ Колумбия, описывается следующим образом:

$$E_1 \in Flyings \wedge Flyer(E_1, Shankar) \wedge Origin(E_1, SF) \wedge Destination(E_1, DC),$$

где *Flyings* — категория всех событий перелета. Конкретизируя события, мы делаем возможным добавление любого количества произвольной информации о них. Например, можно указать, что в этом полете Шанкар ощущал тряску — $Bumpy(E_1)$. В онтологии, где события представлены n-арными предикатами, нет способа представить некоторую дополнительную информацию, подобную этой, — простой переход к $n+1$-арному предикату нельзя считать масштабируемым решением.

Чтобы сделать утверждение о том, что флюента действительно являлась истинной, начиная с некоторой точки времени t_1 и до точки времени t_2, воспользуемся предикатом T, записав его как $T(At(Shankar, Berkeley), t_1, t_2)$. Аналогичным

[3] Здесь термины "событие" и "действие" мы будем считать взаимозаменяемыми — оба они означают "что-то, что может произойти".

образом используем предикат $Happens(E_1, t_1, t_2)$, чтобы сообщить, что событие E_1 действительно произошло, начавшись в точке времени t_1 и закончившись в точке времени t_2. Полный набор предикатов для одной из версий исчисления событий приведен ниже.[4]

$T(f, t_1, t_2)$	Флюента f является истинной для всех точек времени между t_1 и t_2
$Happens(e, t_1, t_2)$	Событие e началось в точке времени t_1 и закончилось в точке времени t_2
$Initiates(e, f, t)$	Событие e изменило значение флюенты f на истинное в точке времени t
$Terminates(e, f, t)$	Событие e изменило значение флюенты f на не истинное в точке времени t
$Initiated(f, t_1, t_2)$	Значение флюенты f стало истинным в некоторой точке времени между t_1 и t_2
$Terminated(f, t_1, t_2)$	Значение флюенты f перестало быть истинным в некоторой точке времени между t_1 и t_2
$t_1 < t_2$	Точка времени t_1 имеет место раньше точки времени t_2

Теперь можно полностью описать результаты события полета:

$$E = Flyings(a, here, there) \text{ и } Happens(E, t_1, t_2) \Rightarrow Terminates(E, At(a, here), t_1) \wedge Initiates(E, At(a, there), t_2).$$

Также примем еще одно необычное событие, *Start*, описывающее первоначальное состояние, сообщая, какие флюенты истинны (используя предикат *Initiates*) или ложны (используя предикат *Terminated*) в момент старта. Это позволит описать, какие флюенты и в какой точке времени являются истинными, всего лишь парой аксиом для T и $\neg T$, имеющих тот же самый общий формат, что и аксиомы определения преемника. Предположим, что событие происходит между точками времени t_1 и t_3 и в точке t_2 где-то в пределах этого интервала событие изменяет значение флюенты f, либо инициируя ее (сделав истинной), либо завершая ее (сделав ложной). Затем в точке времени t_4 в будущем, если никакое другое промежуточное событие не изменило значение этой флюенты (завершив или инициировав ее соответственно), флюента будет все еще сохранять свое значение. Формально аксиомы определяются так.

$$Happens(e, t_1, t_3) \wedge Initiates(e, f, t_2) \wedge \neg Terminated(f, t_2, t_4) \wedge t_1 \le t_2 \le t_3 \le t_4 \Rightarrow T(f, t_2, t_4)$$

$$Happens(e, t_1, t_3) \wedge Terminates(e, f, t_2) \wedge \neg Initiated(f, t_2, t_4) \wedge t_1 \le t_2 \le t_3 \le t_4 \Rightarrow \neg T(f, t_2, t_4)$$

[4] Наша версия основана на статье Шанахана ([2035], 1999), но с некоторыми изменениями.

Здесь предикаты *Terminated* и *Initiated* определяются следующим образом.

$$Terminated(f, t_1, t_5) \Leftrightarrow$$
$$\exists\, e, t_2, t_3, t_4 \quad Happens(e, t_2, t_4) \wedge Terminates(e, f, t_3) \wedge t_1 \le t_2 \le t_3 \le t_4 \le t_5$$
$$Initiated(f, t_1, t_5) \Leftrightarrow$$
$$\exists\, e, t_2, t_3, t_4 \quad Happens(e, t_2, t_4) \wedge Initiates(e, f, t_3) \wedge t_1 \le t_2 \le t_3 \le t_4 \le t_5$$

Можно расширить исчисление событий для представления одновременных событий (таких, как катание двух человек на качелях), внешних событий (таких, как ветер, перемещающий объект), непрерывных событий (таких, как морские приливы и отливы), недетерминированных событий (таких, как бросание монетки и определение, что выпало — орел или решка) и других всевозможных усложнений.

10.3.1. Время

Исчисление событий открывает возможность говорить о точках и интервалах времени. Мы рассмотрим интервалы двух типов: моменты времени и продолжительные интервалы. Различие между ними состоит только в том, что моменты времени имеют нулевую продолжительность:

$$Partition(\{Moments, ExtendedIntervals\}, Intervals)$$
$$i \in Moments \Leftrightarrow Duration(i) = Seconds(0).$$

Далее необходимо ввести временную шкалу и связать точки на этой шкале с моментами времени, что позволяет сформулировать понятие абсолютных значений времени. Временная шкала выбирается произвольно; в данной книге время измеряется в секундах и используется соглашение, что момент времени в полночь (среднее время по Гринвичу) 1 января 1900 года имел значение времени 0. Функции *Begin* и *End* позволяют определить первый и последний моменты времени в интервале, а функция *Time* сообщает момент времени на временной шкале, соответствующий текущему моменту. Функция *Duration* измеряет разность между временем окончания и временем начала.

$$Interval(i) \Rightarrow Duration(i) = (Time(End(i)) - Time(Begin(i)))$$
$$Time(Begin(AD1900)) = Seconds(0)$$
$$Time(Begin(AD2001)) = Seconds(3187324800)$$
$$Time(End(AD2001)) = Seconds(3218860800)$$
$$Duration(AD2001) = Seconds(31536000)$$

Чтобы было проще определять числа, обозначающие количество секунд от начала отсчета, введем также функцию *Date*, которая принимает шесть параметров (*часы*, *минуты*, *секунды*, *день*, *месяц* и *год*) и возвращает соответствующую точку во времени.

$$Time(Begin(AD2001)) = Date(0, 0, 0, 1, Jan, 2001)$$
$$Date(0, 20, 21, 24, 1, 1995) = Seconds(3000000000)$$

Предикат *Meet* позволяет определить, совпадает ли момент окончания первого интервала времени с моментом начала второго интервала, — эти значения времени задаются в секундах. Полный набор предикатов отношений интервалов показан ниже (Аллен [32], 1983) и на рис. 10.2.

$$
\begin{aligned}
Meet(i,j) &\Leftrightarrow End(i) = Begin(j)\\
Before(i,j) &\Leftrightarrow End(i) < Begin(j)\\
After(j,i) &\Leftrightarrow Before(i,j)\\
During(i,j) &\Leftrightarrow Begin(j) < Begin(i) < End(i) < End(j)\\
Overlap(i,j) &\Leftrightarrow Begin(i) < Begin(j) < End(i) < End(j)\\
Starts(i,j) &\Leftrightarrow Begin(i) = Begin(j)\\
Finishes(i,j) &\Leftrightarrow End(i) = End(j)\\
Equals(i,j) &\Leftrightarrow Begin(i) = Begin(j) \wedge End(i) = End(j)
\end{aligned}
$$

Все эти предикаты полностью соответствуют своему интуитивному смыслу, за исключением предиката *Overlap*: мы склонны понимать перекрытие как симметричное отношение (если i перекрывает j, то и j перекрывает i), но согласно приведенному определению выражение $Overlap(i, j)$ будет истинно только в том случае, если i начинается *раньше* j. Опыт показал, что такое определение более полезно при написании аксиом. Например, чтобы сформулировать утверждение, что царствование Елизаветы II следовало за царствованием Георга VI, а царствование Элвиса Пресли перекрывается с периодом 1950-х годов, можно записать следующее:

$$
\begin{aligned}
&Meets(ReignOf(GeorgeVI), ReignOf(ElizabethII)),\\
&Overlap(Fifties, ReignOf(Elvis)),\\
&Begin(Fifties) = Begin(AD1950),\\
&End(Fifties) = End(AD1959).
\end{aligned}
$$

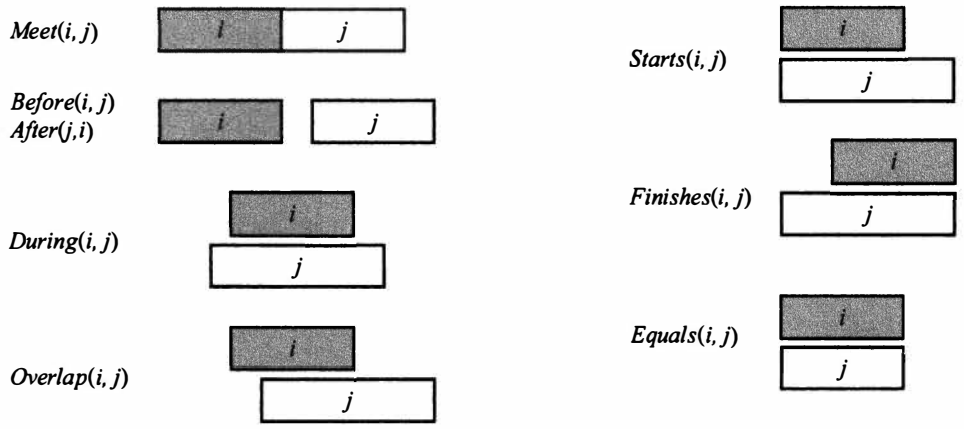


Рис. 10.2. Предикаты, задаваемые на временных интервалах

10.3.2. Флюенты и объекты

Физические объекты можно рассматривать как обобщенные события в том смысле, что любой физический объект представляет собой фрагмент пространства-времени. Например, США можно рассматривать как событие, которое началось, скажем, в 1776 году в виде союза 13 штатов и все еще продолжается в наши дни как союз 50 штатов. Изменяющиеся свойства соответствующего объекта USA можно описать с использованием флюент, например такой, как *Population*(USA) (население). Еще одним свойством объекта USA, которое изменяется через каждые четыре или восемь лет, если не учитывать непредвиденные происшествия, является смена президента этого государства. Можно предложить понимать выражение *President*(USA) как логический терм, который обозначает различные объекты в разное время.

К сожалению, это невозможно, поскольку в каждой конкретной структуре модели любой терм обозначает один и только один объект. (Терм *President*(USA, *t*) мог бы обозначать разные объекты в зависимости от значения *t*, но в принятой нами онтологии временные индексы рассматриваются отдельно от высказываний флюент.) Единственная возможность состоит в том, чтобы считать, что *President*(USA) обозначает единственный объект, который в разное время определяет разных людей. Это объект, которым были Джордж Вашингтон с 1789 по 1796 год, Джон Адамс с 1796 по 1800 год и т.д. (рис. 10.3). Для того чтобы высказать утверждение, что Джордж Вашингтон был президентом в течение всего 1790 года, можно записать следующее:

$$T(Equals(President(USA), GeorgeWashington), Begin(AD1790), End(AD1790)).$$

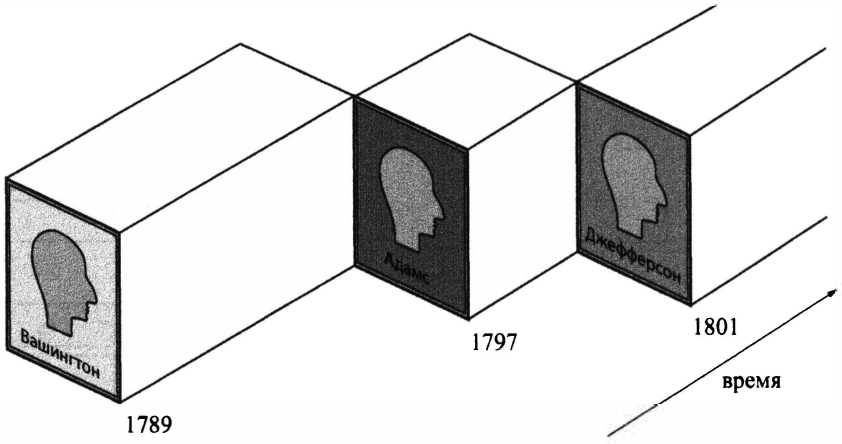


Рис. 10.3. Схематическое представление значения объекта *President*(USA) в первые годы его существования

В данном высказывании знак *равно* представляет собой не стандартный логический предикат =, а символ функции *Equals*, поэтому его интерпретация заключается не в том, что *GeorgeWashington* и *President*(USA) были логически идентичными в 1790 году: логическое равенство есть нечто, неизменное во времени. В данном случае логическое равенство существует между подсобытиями каждого объекта, которые определены на период 1790 года.

10.4. Ментальные объекты и модальная логика

Агенты, которые рассматривались нами до сих пор, имели убеждения и могли выводить новые убеждения логическим путем. Но ни один из этих агентов не обладал какими-либо знаниями *о* самих убеждениях или *о* логическом выводе. Однако знания о собственных знаниях и процессах формирования рассуждений будут полезны для управления логическим выводом. Например, предположим, что Алиса спрашивает: "Чему равен квадратный корень из 1764?", а Боб отвечает: "Я не знаю". Если Алиса будет настаивать: "Ну подумай, же!", Боб начнет понимать, что если немного подумать, то на этот вопрос действительно можно будет ответить. С другой стороны, если Алиса спросит: "Президент сейчас сидит или стоит?", то Боб должен понимать, что, сколько не думай, это едва ли ему поможет. Знания о знаниях других агентов также являются важными: Боб должен понимать, что ответ, безусловно, знает сам президент.

По сути, нам нужна модель тех ментальных (или мыслимых) объектов, которые находятся в чьей-то голове (или чьей-то базе знаний), и ментальных (или мыслительных) процессов, которые манипулируют этими ментальными объектами. Модель не обязательно должна быть детальной. Нет никакой необходимости уметь предсказывать, сколько миллисекунд потребуется какому-то конкретному агенту, чтобы прийти к некоторому логическому заключению. Будет вполне достаточно просто иметь возможность прийти к выводу, что президент знает, сидит он или стоит.

Начнем с ▶**пропозициональных установок**, которые агент может иметь по отношению к ментальным объектам, с таких установок, как *Believes* (убежден), *Knows* (знает), *Wants* (желает) и *Informs* (сообщает). Сложность здесь состоит в том, что эти установки не ведут себя как "нормальные" предикаты. Например, предположим, мы пытаемся сделать утверждение, что Лойс знает, что Супермен умеет летать:

$$Knows(Lois, CanFly(Superman)).$$

Одна небольшая проблема здесь заключается в том, что запись *CanFly*(*Superman*) мы обычно понимаем как логическое высказывание, но здесь оно употребляется как терм, поскольку параметрами предикатов могут быть только термы (а не высказывания). Эту проблему можно устранить посредством овеществления высказывания *CanFly*(*Superman*), в результате чего оно превращается в флюенту.

Более серьезная проблема состоит в том, что если верно, что Супермен — это Кларк Кент, то мы должны заключить, что Лойс знает, что Кларк может летать, но это не так, поскольку (в большинстве версий истории) Лойс *не знает*, что Кларк — это и есть Супермен:

$$(Superman = Clark) \wedge Knows(Lois, CanFly(Superman)) \models Knows(Lois, CanFly(Clark)).$$

Это утверждение является следствием того факта, что рассуждение о равенстве *встроено* в логику. Обычно это хорошо: если агент знает, что $2+2=4$ и $4<5$, то будет полезно, чтобы агент знал, что $2+2<5$. Это свойство называется ▶**ссылочной прозрачностью** — не имеет значения, какой терм использован для ссылки на объект, важно лишь то, какой объект именуется этим термом. Но для пропозициональных установок, таких как *believes* и *knows*, хотелось бы иметь ссылочную непрозрачность, — здесь имеет значение, какой *именно* терм использован, поскольку не все агенты знают, какие из термов ссылаются на один и тот же объект.

Можно исправить эту ситуацию за счет дополнительного овеществления: иметь один объект для представления и Кларка, и Супермена, другой объект — для представления человека, которого Лойс знает как Кларка, и еще один объект — для человека, которого Лойс знает как Супермена. Однако такое вынужденное увеличение количества используемых объектов означает, что высказывания, которые требуется написать, быстро становятся многословными и неуклюжими.

Для решения этой проблемы была разработана ▶**модальная логика**. Обычная логика связана с единственной модальностью, модальностью истины, позволяющей нам выразить утверждения "P истинно" или "P ложно". Модальная логика включает в себя специальные ▶**модальные операторы**, которые в качестве аргументов принимают высказывания (а не термы). Например, высказывание "A знает P" можно представить с использованием нотации $\mathbf{K}_A P$, где $\mathbf{K}$ — это модальный оператор для представления отношения знания. Он имеет два аргумента: имя агента (записывается как индекс) и высказывание. Синтаксис модальной логики такой же, как и логики первого порядка, за исключением того, что высказывания также могут быть сформированы с помощью модальных операторов.

Семантика модальной логики является более сложной. В логике первого порядка **модель** содержит множество объектов и интерпретацию, обеспечивающую отображение каждого имени на соответствующий объект, отношение или функцию. В модальной логике мы хотим иметь возможность рассматривать как возможность того, что секретная личность Супермена является Кларком, так и возможность того, что это не так.

Следовательно, нам необходима более сложная модель, состоящая из набора ▶**возможных миров**, а не из единственного истинного мира. Эти миры связаны в граф ▶**отношениями доступности**, по одному отношению для каждого модального оператора. Говорят, что мир w_1 является доступным из мира w_0 по отношению к модальному оператору $\mathbf{K}_A$, если все в мире w_1 совместимо с тем, что A знает

в мире w_0. Например, в реальном мире Бухарест является столицей Румынии, но для агента, который этого не знает, мир, где столицей Румынии является, скажем, София, вполне доступен. Будем надеяться, что мир, в котором $2+2=5$ окажется недоступен для любого агента.

В общем случае атом знания $\mathbf{K}_A P$ является истинным в мире w тогда и только тогда, когда выражение P является истинным в каждом мире, доступном из w. Истинность более сложных высказываний выводится рекурсивным применением этого правила и обычных правил логики первого порядка. Это означает, что модальная логика может использоваться для рассуждений о вложенных высказываниях о знаниях: что один агент знает о знаниях другого агента. Например, можно сказать, что даже если Лойс не знает, является ли Кларк Кент секретной личностью Супермена, она знает, что Кларк это знает:

$$\mathbf{K}_{Lois}[\mathbf{K}_{Clark} Identity(Superman, Clark) \vee \mathbf{K}_{Clark} \neg Identity(Superman, Clark)].$$

Модальная логика также позволяет решить некоторые сложные проблемы, связанные с взаимодействием квантификаторов и знаний. Например, высказывание на естественном языке "Бонд знает, что кто-то шпион" является двусмысленным. Первое прочтение гласит о том, что существует определенный человек, о котором Бонд знает, что это шпион, что можно записать следующим образом:

$$\exists x \ \mathbf{K}_{Bond} Spy(x).$$

В модальной логике это означает, что существует некто x, такой, что во всех совместимых мирах Бонд знает о нем, что это шпион. Второе возможное прочтение гласит том, что Бонд просто знает, что имеется по крайней мере один шпион:

$$\mathbf{K}_{Bond} \exists x \ Spy(x).$$

В модальной логике это интерпретируется как высказывание, что в каждом совместимом мире существует некто x, являющийся шпионом, но это необязательно должен быть один и тот же x в каждом из этих миров.

Теперь, когда у нас есть модальный оператор для знаний, можно написать для него аксиомы. Во-первых, можно сказать, что агенты способны делать выводы; если агент знает P и знает, что P подразумевает Q, то агент знает Q:

$$(\mathbf{K}_a P \wedge \mathbf{K}_a (P \Rightarrow Q)) \Rightarrow \mathbf{K}_a Q.$$

Из этого (и нескольких других правил о логической идентичности) можно сделать заключение, что $\mathbf{K}_A(P \vee \neg P)$ является тавтологией: каждый агент знает, что каждое высказывание P является либо истинным, либо ложным. С другой стороны, высказывание $(\mathbf{K}_A P) \vee (\mathbf{K}_A \neg P)$ тавтологией не является: в общем случае может существовать множество высказываний, о которых агент не знает, что они истинны, и в то же время не знает, что они ложны.

Общепризнанно еще со времен Платона, что знание — это убеждение, истинность которого подтвердилась. Иначе говоря, если что-то является истиной, если

вы в этом убеждены и если у вас для этого есть неопровержимая уважительная причина, то вы знаете это. Для нас это означает, что если вы знаете что-то, то это должно быть истиной, следовательно, можно сформулировать аксиому:

$$\mathbf{K}_a P \Rightarrow P.$$

Более того, логические агенты (но не все люди) способны проанализировать их собственные знания. Если они что-то знают, то они знают, что знают это:

$$\mathbf{K}_a P \Rightarrow \mathbf{K}_a(\mathbf{K}_a P).$$

Теперь мы можем определить подобные аксиомы для убеждений (их часто обозначают как **B**) и других модальных качеств. Однако одной из проблем, связанных с подходом, принятым в модальной логике, является то, что в ней предполагается ►**логическое всеведение** со стороны агентов. То есть, если агент знает некоторое множество аксиом, то он знает и все следствия из этих аксиом. Это ненадежная основа даже для несколько абстрактного понятия знаний, и она кажется еще худшей для убеждений, поскольку убеждения в большей степени обращаются к вещам, которые физически представлены в агенте, а не просто потенциально выводимы.

Было предпринято немало попыток определить формы ограниченной рациональности агентов, — так, чтобы убеждения агента определялись только теми утверждениями, которые можно вывести с использованием не более k этапов рассуждений или за s секунд вычислений. Как правило, все эти попытки оказывались безуспешными.

10.4.1. Другие модальные логики

Для различных видов модальности, помимо знаний, было предложено много модальных логик. Одно из предложений состоит в добавлении модальных операторов для *возможности* и *необходимости*: возможно, является истинным утверждение, что один из авторов этой книги в данный момент сидит на стуле, и необходимо истинным является утверждение, что $2+2=4$.

Как было отмечено в разделе 8.1.2, некоторые логики предпочитают модальности, связанные со временем. В ►**линейных временных логиках** добавляются следующие модальные операторы.

- **X** P — "Высказывание P будет истинно на следующем временном этапе".
- **F** P — "Высказывание P в конечном итоге (*finally*) станет истинным на некотором будущем временном этапе".
- **G** P — "Высказывание P всегда (*globally*) верно".
- P **U** Q — "Высказывание P остается истинным, пока (*until*) не произойдет Q".

В некоторых случаях используются дополнительные операторы, которые можно вывести из этих. Добавление таких модальных операторов делает логику саму по себе более сложной (а значит, для алгоритма логического вывода задача

нахождения доказательства усложняется). Но эти операторы также позволяют описать некоторые факты в более сжатой форме (что ускоряет логический вывод). Выбор того, какую логику лучше использовать, похож на выбор, какой язык программирования предпочесть: тот, который больше подходит для вашей задачи, или тот, который хорошо знаком вам и вашим коллегам и при этом достаточно эффективен, чтобы успешно достичь поставленных целей.

10.5. Системы рассуждений о категориях

Категории являются первичными строительными блоками каждой крупномасштабной схемы представления знаний. В этом разделе описаны системы, специально разработанные для организации и выполнения рассуждений в отношении категорий. Существуют два тесно связанных семейства подобных систем. ▶**Семантические сети** предоставляют графические средства, необходимые для визуализации базы знаний, и эффективные алгоритмы логического вывода сведений о любом объекте на основании его принадлежности к некоторой категории. ▶**Описательные логики** предоставляют формальный язык для конструирования и комбинирования определений категорий, а также эффективные алгоритмы для установления связей между категориями на уровне подмножеств и надмножеств.

10.5.1. Семантические сети

В 1909 году Чарльз Пирс предложил графическую систему обозначений в виде узлов и дуг, получившую название ▶**экзистенциальные графы**, которую он назвал "логикой будущего". С того времени начались непрекращающиеся дебаты между приверженцами "чистой логики" и приверженцами "семантических сетей". К сожалению, эти дебаты затмили тот факт, что на деле семантические сети представляют собой всего лишь одну из форм логики. Система обозначений, предложенная в семантических сетях для высказываний некоторых типов, часто является более удобной, но если отбросить эти особенности, касающиеся "интерфейса, удобного для человека ", то базовые понятия (объекты, отношения, кванторы и т.д.) окажутся теми же самыми.

Существует много вариантов семантических сетей, но все они способны представлять отдельные объекты, категории объектов и отношения между объектами. В типичной графической системе обозначений имена объектов или категорий изображаются в овалах или прямоугольниках, а связи между ними обозначаются с помощью дуг с метками. Например, на рис. 10.4 показана связь *MemberOf* между *Mary* и *FemalePersons*, что соответствует логическому утверждению $Mary \in FemalePersons$; аналогичным образом связь *SisterOf* между *Mary* и *John* соответствует утверждению *SisterOf*(*Mary*, *John*). Категории можно соединять с помощью связи *SubsetOf* и т.д. Рисовать эти овалы и стрелки так интересно, что часто можно забыть обо всем. Например, мы знаем, что матерями людей являются особы

женского пола, но можно ли нарисовать связь *HasMother* от категории *Persons* к категории *FemalePersons*? Ответом является "нет", поскольку *HasMother* — это связь между человеком и его матерью, а категории не имеют матерей.[5]

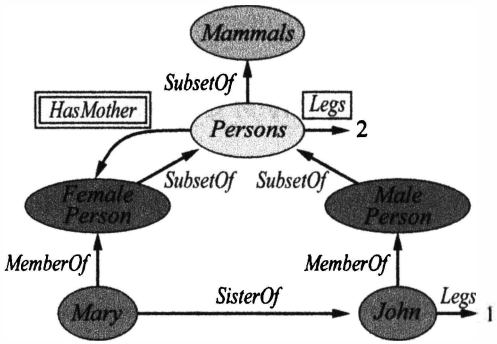


Рис. 10.4. Семантическая сеть с четырьмя объектами (*John*, *Mary*, 1 и 2) и четырьмя категориями. Отношения обозначаются связями с метками

По этой причине на рис. 10.4 используется специальное обозначение — связь с меткой в прямоугольнике, вычерченном двойной линией. Эта связь соответствует следующему утверждению:

$$\forall x \; x \in Persons \Rightarrow [\forall y \; HasMother(x, y) \; \Rightarrow \; y \in FemalePersons].$$

Также можно утверждать, что люди имеют две ноги:

$$\forall x \; x \in Persons \Rightarrow Legs(x, 2).$$

Как и прежде, необходимо тщательно следить, чтобы в этом высказывании не утверждалось, что ногами обладает категория, — на рис. 10.4 для обозначения утверждений о свойствах каждого элемента категории используется метка в прямоугольнике, вычерченном одинарной линией.

Система обозначений семантических сетей предоставляет значительные удобства при формировании рассуждений о **наследовании** такого типа, которые были представлены выше, в разделе 10.2. Например, благодаря тому, что Мэри является

[5] В некоторых ранних системах отсутствовало различие между свойствами элементов категории и свойствами всей категории в целом. Это могло непосредственно приводить ко многим несогласованностям, как указал Дрю Макдермотт в своей статье "Artificial Intelligence Meets Natural Stupidity" (Искусственный интеллект сталкивается с естественной глупостью) ([1540], 1976). Еще одной распространенной проблемой было использование связей *IsA* и для отношений между подмножествами и множествами, и для отношений между элементами и подмножествами, в полном соответствии с грамматикой английского языка: "a cat is a mammal" (Кошка — млекопитающее) и "Fifi is a cat" (Фифи — кошка). Дополнительная информация по этой теме приведена в упражнении 10.27.

человеком, она наследует свойство иметь две ноги. Поэтому, чтобы узнать, сколько ног имеет Мэри, алгоритм наследования проходит по связи *MemberOf* от объекта *Mary* к категории, к которой она принадлежит, а затем проходит по связям *SubsetOf* вверх по иерархии до тех пор, пока не находит категорию, для которой имеется связь с обведенной прямоугольником меткой *Legs*, в данном случае — категорию *Persons*. Простота и эффективность этого механизма логического вывода по сравнению с полуразрешимым логическим доказательством теорем всегда была одной из основных привлекательных особенностей семантических сетей.

Наследование становится сложнее, если некоторый объект может принадлежать более чем к одной категории или если категория может быть подмножеством более чем одной отличной от нее категории, — такая ситуация называется ▶**множественным наследованием**. В подобных случаях алгоритм наследования может находить два или несколько конфликтующих значений, представляющих собой ответ на запрос. Именно по этой причине множественное наследование может быть запрещено в некоторых языках **объектно-ориентированного программирования** (*Object-Oriented Programming* — OOP), таких как язык Java, в котором наследование используется в иерархии классов. В семантических сетях множественное наследование обычно разрешено, но мы отложим обсуждение этой темы до раздела 10.6.

Читатель должен был обнаружить очевидный недостаток системы обозначений семантических сетей в сравнении с логикой первого порядка: тот факт, что между овалами проводятся связи, соединяющие два овала, говорит о том, что в семантических сетях могут быть представлены только *бинарные* отношения. Например, высказывание *Fly*(*Shankar*, *NewYork*, *NewDelhi*, *Yesterday*) не может быть представлено непосредственно в семантической сети. Тем не менее *можно* достичь эффекта реализации *n*-арных высказываний путем овеществления самого рассматриваемого высказывания как события, принадлежащего к соответствующей категории событий. На рис. 10.5 показана структура семантической сети для указанного выше конкретного события. Обратите внимание, что ограничение, связанное с необходимостью применять только бинарные отношения, вынуждает создавать богатую онтологию овеществленных понятий.

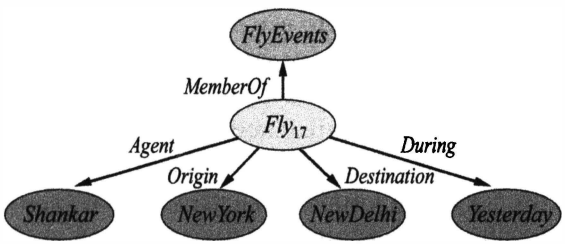


Рис. 10.5. Фрагмент семантической сети, демонстрирующий способ представления логического утверждения *Fly*(*Shankar*, *NewYork*, *NewDelhi*, *Yesterday*)

Овеществление высказываний позволяет представить в системе обозначений семантической сети любое базовое, атомарное высказывание логики первого порядка, не содержащее функций. Реализация некоторых видов высказываний с кванторами всеобщности может осуществляться с помощью инверсных связей и применяемых к категориям связей с метками в прямоугольниках с одинарным и двойным контуром, но даже такие ухищрения отнюдь не позволяют приблизиться к возможностям полной логики первого порядка. В этой системе обозначений недостает таких выразительных возможностей, как отрицание, дизъюнкция, вложенные функциональные символы и кванторы существования. В последнее время появилась *возможность* расширить эту систему обозначений, чтобы сделать ее эквивалентной логике первого порядка — как при использовании экзистенциальных графов Пирса, — но это приводит к исчезновению одного из основных преимуществ семантических сетей — простоты и прозрачности процессов логического вывода. Простая система обозначений позволяет проектировщикам создавать большие сети и все еще иметь полное представление о том, какие запросы должны быть эффективными, поскольку, во-первых, можно легко визуально представить, через какие этапы должна пройти процедура логического вывода, и, во-вторых, в некоторых случаях язык запросов настолько прост, что формулировать в нем сложные, запутанные запросы не представляется возможным.

Для тех случаев, когда выясняется, что выразительная мощь слишком ограничена, во многих системах семантических сетей предусмотрены ►**процедурные вложения**, позволяющие компенсировать эти недостатки. Процедурное вложение — это метод, с помощью которого при выполнении запроса, касающегося некоторого отношения (а иногда и при вводе в базу знаний некоторого утверждения), осуществляется вызов специальной процедуры, предназначенной лишь для обработки этого отношения, а не обращение к общему алгоритму логического вывода.

Одной из самых важных характерных особенностей семантических сетей является их способность представлять ►**заданные по умолчанию значения** для категорий. Внимательно изучив рис. 10.4, можно обнаружить, что Джон имеет одну ногу, несмотря на тот факт, что он — человек, а все люди имеют две ноги. В строго логической базе знаний такое сочетание фактов было бы противоречивым, но в семантической сети утверждение, что все люди имеют две ноги, обладает лишь статусом значения, применяемого по умолчанию; т.е. применительно к любому человеку предполагается, что он имеет две ноги, если этому не противоречит более конкретная информация. Принудительное применение семантики заданных по умолчанию значений осуществляется естественным образом с помощью алгоритма наследования, поскольку этот алгоритм следует по связям, направленным вверх от самого объекта (в данном случае от Джона), и останавливается, как только находит искомое значение. Принято говорить, что заданное по умолчанию значение ►**переопределяется** более конкретным значением. Обратите внимание на то, что можно было бы также переопределить информацию о заданном по умолчанию

количестве ног, создав категорию одноногих людей *OneLeggedPersons*, подмножество категории *Persons*, элементом которого является *John*.

Для этой сети также можно было бы сохранить строго логическую семантику, сформировав высказывание, что утверждение *Legs* для *Persons* содержит исключение для *John*:

$$\forall x \; x \in Persons \wedge x \neq John \Rightarrow Legs(x, 2).$$

Для *фиксированной* сети такое дополнение было бы семантически адекватным, но стало бы гораздо менее кратким по сравнению с самой сетевой системой обозначений, если бы в сети было много исключений. Однако в сети, обновляемой посредством добавления дополнительных утверждений, такой подход оказался бы полностью неприемлемым, — фактически нужно было бы указывать, что к исключениям относятся любые люди, в отношении которых еще не известно, имеют ли они только одну ногу. В разделе 10.6 приведены дополнительные сведения об этой проблеме и о формировании рассуждений в логике умолчаний в целом.

10.5.2. Описательные логики

Синтаксис логики первого порядка предназначен для упрощения процедуры формирования высказываний об объектах, тогда как ►**описательные логики** представляют собой системы обозначений, которые предназначены для упрощения процедуры описания определений и свойств категорий. Системы описательной логики развились из семантических сетей как ответ на необходимость формализовать тот смысл, который несут в себе сети, сохранив вместе с тем акцент на использование таксономической структуры в качестве принципа организации.

Основные задачи логического вывода для описательных логик сводятся к ►**обобщению** (проверке того, является ли одна категория подмножеством другой, путем сравнения их определений) и ►**классификации** (определению принадлежности некоторого объекта к какой-то категории). Некоторые системы включают также проверку ►**непротиворечивости** определения категории, т.е. того, являются ли выполнимыми критерии принадлежности к категории с точки зрения логики.

Типичным языком описательной логики является CLASSIC (Боргида и др. [255], 1989). Синтаксис описаний языка CLASSIC приведен на рис. 10.6.[6] Например, чтобы сформулировать утверждение, что холостяками называют неженатых взрослых мужчин, можно записать следующее:

$$Bachelor = And(Unmarried, Adult, Male).$$

[6] Обратите внимание, что этот язык не позволяет просто заявить, что одно понятие (или категория) является подмножеством другого. Это требование введено сознательно: отношение обобщения между категориями должно быть выведено логическим путем из некоторых аспектов описаний категорий. Если этого сделать не удается, то в описаниях имеются какие-то упущения.

Эквивалент этого утверждения в логике первого порядка выглядит так:

$$Bachelor(x) \Leftrightarrow Unmarried(x) \wedge Adult(x) \wedge Male(x).$$

$$\begin{aligned}
Concept &\rightarrow \textbf{Thing} \mid ConceptName \\
&\mid \textbf{And}(Concept, \ldots) \\
&\mid \textbf{All}(RoleName, Concept) \\
&\mid \textbf{AtLeast}(Integer, RoleName) \\
&\mid \textbf{AtMost}(Integer, RoleName) \\
&\mid \textbf{Fills}(RoleName, IndividualName, \ldots) \\
&\mid \textbf{SameAs}(Path, Path) \\
&\mid \textbf{OneOf}(IndividualName, \ldots) \\
Path &\rightarrow [RoleName, \ldots] \\
ConceptName &\rightarrow Adult \mid Female \mid Male \mid \ldots \\
RoleName &\rightarrow Spouse \mid Daughter \mid Son \mid \ldots
\end{aligned}$$

Рис. 10.6. Синтаксис описаний в подмножестве языка CLASSIC

Обратите внимание, что в этом языке описательной логики имеется алгебра операции с предикатами, что, безусловно, невозможно в логике первого порядка. Любое описание на языке CLASSIC может быть переведено в соответствующее эквивалентное высказывание логики первого порядка, но некоторые описания формулируются проще на языке CLASSIC. Например, чтобы описать множество мужчин, имеющих по меньшей мере трех сыновей, из которых все безработные и женаты на врачах, и, самое большее, двух дочерей, из которых все являются преподавателями на кафедрах физики или математики, можно записать следующее:

$$\begin{aligned}
&And(Man, AtLeast(3, Son), AtMost(2, Daughter), \\
&\quad All(Son, And(Unemployed, Married, All(Spouse, Doctor))), \\
&\quad All(Daughter, And(Professor, Fills(Department, Physics, Math)))).
\end{aligned}$$

Перевод этого высказывания на язык логики первого порядка предлагается выполнить читателю в качестве упражнения.

Вероятно, одной из самых важных характерных особенностей описательных логик является присутствующий в них акцент на осуществимости логического вывода. Любой экземпляр проблемы решается посредством его описания с последующим выполнением запроса, является ли этот экземпляр обобщением одной из нескольких возможных категорий решений. В стандартных системах логики первого порядка предсказание затрат времени на поиск решения часто оказывается невозможным. Чаще всего, пользователю самому приходится разрабатывать

представление, позволяющее исключать множества высказываний, которые, вероятно, вынудят систему выполнять вычисления в течение нескольких недель, пока задача будет решена. С другой стороны, в описательных логиках все направлено на обеспечение того, чтобы проблема проверки обобщения могла быть решена за время, полиномиально зависящее от размера описаний.[7]

На первый взгляд, такое свойство описательных логик может показаться очень привлекательным, пока не становится понятно, что оно ведет к одному из двух следствий: либо задача оказывается настолько сложной, что ее описание вообще невозможно сформулировать, либо она потребует экспоненциально большого описания! Тем не менее анализ осуществимости логического вывода позволяет пролить свет на то, какого рода конструкции вызывают проблемы, и поэтому помочь пользователю понять, какие следствия вытекают из использования различных представлений. Например, в описательных логиках обычно не используются такие отношения, как *отрицание* и *дизъюнкция*. Дело в том, что каждое из этих отношений вынуждает логические системы первого порядка для обеспечения полноты проходить через этап анализа вариантов, который может потенциально характеризоваться экспоненциальной сложностью. В языке CLASSIC допускается использовать только ограниченную форму дизъюнкции в конструкциях *Fills* и *OneOf*, которые допускают выполнение дизъюнкции по явно заданным объектам, а не по их описаниям. Если бы было разрешено использовать дизъюнктивные описания, то вложенные определения могли бы легко привести к появлению экспоненциального количества альтернативных путей, по которым одна категория могла бы обобщать другую.

10.6. Рассуждения при наличии информации по умолчанию

В предыдущем разделе был приведен простой пример утверждения с заданной по умолчанию информацией о состоянии: люди имеют две ноги. Это заданное по умолчанию значение можно переопределить с помощью более конкретной информации, например такой, что Джон Сильвер имеет одну ногу. Кроме того, было показано, что механизм наследования в семантических сетях предоставляет простой и естественный способ переопределения значений, заданных по умолчанию. В этом разделе проводится более общее обсуждение заданной по умолчанию информации, с тем чтобы можно было понять *семантику* умолчаний, а не просто создать какой-то процедурный механизм.

[7] В целом язык CLASSIC обеспечивает эффективную проверку обобщения, но в худшем случае время прогона может стать экспоненциальным.

10.6.1. Логика косвенного описания и логика умолчания

Выше приводились два примера процессов формирования рассуждений, нарушающих свойство ▶**монотонности** логики, доказанное в главе 7.[8] В этой главе речь шла о свойстве, наследуемом всеми элементами категории в семантической сети, которое можно переопределить с использованием более конкретной информации, касающейся подкатегории. Во втором примере, приведенном в разделе 9.4.4, было показано, что на основании предположения о замкнутости мира, если высказывание α не упоминается в KB, то $KB \models \neg\alpha$, но $KB \wedge \alpha \models \alpha$.

Простой самоанализ показывает, что подобные нарушения монотонности логики широко распространены и в обыденных рассуждениях здравого смысла. Создается впечатление, что люди часто просто "перепрыгивают" к заключениям. Например, увидев припаркованный на улице автомобиль, мы обычно предполагаем, что у этого автомобиля четыре колеса, даже если нам видны только три. Действительно, теория вероятностей позволяет уверенно прийти к заключению, что вероятность существования четвертого колеса весьма велика, но большинству людей мысль о том, что у автомобиля нет четвертого колеса, *даже не приходит в голову, если не появятся некоторые новые факты*. Следовательно, создается впечатление, что вывод о наличии у наблюдаемого автомобиля четырех колес достигается *по умолчанию*, если нет каких-либо причин поставить этот вывод под сомнение. Если же поступают новые факты — например, становится очевидно, что владелец катит колесо, а сам автомобиль поднят домкратом, — то первоначальный вывод может быть отброшен. Принято говорить, что в рассуждениях такого рода проявляется ▶**немонотонность**, поскольку множество убеждений не возрастает монотонно со временем, по мере поступления новых фактов. Для того чтобы можно было описывать подобное поведение, были разработаны ▶**немонотонные логики**, в которых используются модифицированные определения понятий истинности и логического следствия. В данном разделе рассматриваются две такие логики, изучению которых были посвящены обширные исследования: логика косвенного описания и логика умолчания.

▶**Косвенное описание** (*circumscription*) может рассматриваться как более мощная и точная версия допущения замкнутости мира. Идея состоит в том, что должны быть заданы конкретные предикаты, в отношении которых предполагается, что они являются "настолько ложными, насколько это возможно", т.е. ложными для всех объектов, за исключением тех, для которых эти предикаты заведомо истинны. Например, предположим, что необходимо ввести применяемое по умолчанию правило, что птицы летают. Для этого введем предикат, скажем $Abnormal_1(x)$, и запишем следующее:

$$Bird(x) \wedge \neg Abnormal_1(x) \Rightarrow Flies(x).$$

[8] Напомним, что условие монотонности требует, чтобы все высказывания, которые следуют из базы знаний KB, по-прежнему следовали бы из нее после добавления новых высказываний. Это означает, что если $KB \models \alpha$, то $KB \wedge \beta \models \alpha$.

Если будет указано, что для предиката $Abnormal_1$ должно применяться **косвенное описание**, то программа формирования рассуждений будет предполагать, что истинным является утверждение $\neg Abnormal_1(x)$, а не $Abnormal_1(x)$. Это позволит вывести заключение $Flies(Tweety)$ (летающие чирикают) из предпосылки $Bird(Tweety)$, но подобное заключение станется недействительным, если в базу знаний вводится утверждение $Abnormal_1(Tweety)$.

Косвенное описание может рассматриваться как один из примеров логики ►**предпочтения моделей** (*model preference*). В подобных логиках высказывание следует из базы знаний (со статусом, заданным по умолчанию), если оно истинно во всех *предпочтительных* моделях базы знаний, в противоположность требованию истинности во *всех* моделях в классической логике. С точки зрения косвенного описания одна модель является предпочтительной по отношению к другой, если в ней имеется меньшее количество аномальных объектов.[9] Рассмотрим, как эта идея может применяться в контексте множественного наследования в семантических сетях. Стандартный пример, демонстрирующий проблемы множественного наследования, называется "парадоксом Никсона". Этот пример основан на том наблюдении, что президент Ричард Никсон был одновременно и квакером (поэтому по умолчанию пацифистом), и республиканцем (поэтому по умолчанию не пацифистом). Эту ситуацию можно описать следующим образом:

$$Republican(Nixon) \wedge Quaker(Nixon),$$
$$Republican(x) \wedge \neg Abnormal_2(x) \Rightarrow \neg Pacifist(x),$$
$$Quaker(x) \wedge \neg Abnormal_3(x) \Rightarrow Pacifist(x).$$

Если применяется косвенное описание для предикатов $Abnormal_2$ и $Abnormal_3$, то возникают две предпочтительные модели: в одной из них истинны выражения $Abnormal_2(Nixon)$ и $Pacifist(Nixon)$, а в другой — $Abnormal_3(Nixon)$ и $\neg Pacifist(Nixon)$. Таким образом, программа формирования рассуждений на основе косвенных описаний остается в полном неведении в отношении того, является ли Никсон пацифистом. При желании можно дополнительно ввести утверждение, что религиозные убеждения имеют приоритет над политическими убеждениями; для этого можно воспользоваться формальной системой, называемой ►**косвенным описанием с приоритетами** (*prioritized circumscription*), чтобы отдать предпочтение моделям, в которых минимизируется предикат $Abnormal_3$.

►**Логика умолчания** (*default logic*) — это формальная система, в которой для вывода непротиворечивых немонотонных заключений могут быть записаны

[9] С точки зрения допущения о замкнутости мира одна модель является предпочтительной по отношению к другой, если в ней имеется меньше истинных атомов, т.е. предпочтительными моделями являются минимальные модели. Существует естественная связь между допущением о замкнутости мира и базами знаний с определенными выражениями, поскольку фиксированная точка, достигаемая в процессе прямого логического вывода в такой базе знаний, представляет собой уникальную минимальную модель (см. раздел 7.5.4).

►**правила по умолчанию**. Заданное по умолчанию правило выглядит следующим образом:

$$Bird(x) : Flies(x)/Flies(x).$$

Это правило означает, что если выражение $Bird(x)$ является истинным, а выражение $Flies(x)$ не противоречит базе знаний, то вывод $Flies(x)$ может быть сделан по умолчанию. В общем случае заданное по умолчанию правило выглядит следующим образом:

$$P : J_1, \ldots, J_n/C,$$

где P называется предпосылкой, C — заключением, а J_i представляют собой обоснования. Если можно доказать, что любое из обоснований ложно, то заключение вывести нельзя. Любая переменная, появляющаяся в J_i или C, должна присутствовать и в P. Пример с парадоксом Никсона может быть представлен в логике умолчаний с помощью одного факта и двух заданных по умолчанию правил следующим образом:

$$Republican(Nixon) \wedge Quaker(Nixon),$$
$$Republican(x) : \neg Pacifist(x)/\neg Pacifist(x),$$
$$Quaker(x) : Pacifist(x)/Pacifist(x).$$

Для интерпретации того, что означают заданные по умолчанию правила, определим понятие ►**расширения** теории умолчаний как максимального множества следствий из этой теории. Таким образом, расширение S состоит из первоначально известных фактов и множества заключений, полученных на основе заданных по умолчанию правил, — таких, что из S нельзя больше вывести дополнительные заключения, а обоснования всех сделанных по умолчанию заключений в S не противоречат S. Как и в случае предпочтительных моделей, в логике косвенного описания существуют два возможных расширения для парадокса Никсона: согласно одному из них он является пацифистом, а согласно другому — нет. Имеются также схемы с приоритетами, в которых определенные заданные по умолчанию правила могут получать преимущества над другими, что позволяет разрешать некоторые неопределенности.

С 1980 года, когда впервые были предложены немонотонные логики, был достигнут большой прогресс в понимании их математических свойств. Однако некоторые вопросы остаются нерешенными. Например, если высказывание "Автомобили имеют четыре колеса" является ложным, то что влечет за собой наличие такого высказывания в некоторой базе знаний? Каковым является наиболее приемлемое множество заданных по умолчанию правил? Если нет возможности отдельно для каждого правила принять решение о том, должно ли оно находиться в нашей базе знаний, то налицо серьезная проблема отсутствия модульности. Наконец, как могут использоваться для принятия решений убеждения, имеющие заданный по умолчанию статус? По-видимому, эта проблема является наиболее сложной для систем формирования рассуждений по умолчанию.

Принятие решений часто связано с поиском компромиссов, и поэтому необходимо сравнивать *силу* убеждений в отношении предполагаемых результатов различных действий со *стоимостью* принятия неправильного решения. В тех случаях, когда решения одного и того же типа должны быть приняты повторно, появляется возможность интерпретировать заданные по умолчанию правила как высказывания с "пороговой вероятностью". Например, заданное по умолчанию правило "Тормоза в моем автомобиле всегда в порядке" фактически означает "Если нет другой информации, то вероятность, что тормоза в моем автомобиле в порядке, достаточно велика, чтобы оптимальным решением для меня был выезд без их проверки". Если контекст принятия решения изменяется — например, если приходится спускать тяжело груженый самосвал вниз по крутой горной дороге, — такое заданное по умолчанию правило внезапно становится неприемлемым, даже если нет новых фактов, говорящих о том, что тормоза неисправны. Подобные соображения привели некоторых исследователей к выводу, что необходимо продумать, как внедрить средства формирования рассуждений по умолчанию в теорию вероятностей.

10.6.2. Системы поддержки истинности

Выше было показано, что многие логические выводы, полученные с помощью той или иной системы представления знаний, могут иметь лишь некоторый заданный по умолчанию статус, а не быть абсолютно достоверными. Поэтому некоторые из таких полученных логическим путем фактов неизбежно оказываются ложными и должны быть отброшены на основании новой информации. Этот процесс называется ▶**пересмотром убеждений** (*belief revision*).[10] Предположим, что база знаний *KB* содержит высказывание *P* (возможно, заключение, сформированное по умолчанию с помощью алгоритма прямого логического вывода, или, возможно, просто неверное утверждение) и требуется выполнить операцию TELL(*KB*, $\neg P$). Но чтобы избежать возникновения противоречия, вначале необходимо выполнить операцию RETRACT(*KB*, *P*). На первый взгляд, в этом нет ничего сложного, но если на основании *P* были выведены и внесены в базу знаний какие-то *дополнительные* высказывания, то возникают проблемы. Например, импликация $P \Rightarrow Q$ могла использоваться для внесения в базу знаний высказывания *Q*. Очевидное "решение" (извлечение всех высказываний, которые следуют из *P*) неприемлемо, поскольку подобные высказывания могут иметь и другие обоснования, помимо *P*. Например, если в базе знаний имеются также высказывания *R* и $R \Rightarrow Q$, то в таком

[10] Пересмотр убеждений часто противопоставляется **обновлению убеждений** (*belief update*), которое происходит, когда осуществляется пересмотр базы знаний для того, чтобы она отражала какое-то изменение в мире, а не добавление новой информации о неизменном мире. В обновлении убеждений объединяется пересмотр убеждений с рассуждениями о времени и изменениях; процесс обновлений связан также с процессом **фильтрации**, описанным в главе 14.

случае Q вообще не следует удалять. ▶**Системы поддержки истинности** (*Truth Maintenance System* — TMS) предназначены именно для того, чтобы справляться с подобными осложнениями.

Один из простых подходов к созданию системы поддержки истинности заключается в отслеживании порядка, в котором высказывания вводятся в базу знаний, путем присваивания им номеров от P_1 до P_n. Тогда при поступлении вызова RETRACT(KB, P_i) система возвращается к состоянию, непосредственно предшествующему добавлению высказывания P_i, удаляя тем самым и само P_i, и любые результаты логического вывода, полученные на основании P_i. После этого могут быть вновь добавлены высказывания от P_{i+1} до P_n. Такая организация работы является простой и гарантирует, что база знаний всегда будет оставаться непротиворечивой, но для извлечения P_i необходимы извлечение и повторная вставка $n - i$ высказываний, а также отмена и повторное выполнение всех логических выводов, вытекающих из этих высказываний. Для систем, в которые происходит добавление большого количества фактов (таких, как крупные коммерческие базы данных), указанный подход является практически не применимым.

Более эффективный подход состоит в создании системы поддержки истинности на основе обоснований, или системы ▶**JTMS** (*Justification-Based Truth Maintenance System*). В системе JTMS к каждому высказыванию в базе знаний прилагается аннотация в виде ▶**обоснования**, состоящего из множества высказываний, на основании которых было выведено это высказывание. Например, если база знаний уже содержит высказывание $P \Rightarrow Q$, то операция TELL(P) вызовет добавление Q с обоснованием $\{P, P \Rightarrow Q\}$. В общем случае высказывание может иметь любое количество обоснований. Обоснования используются для обеспечения эффективного извлечения. При поступлении вызова RETRACT(P) система JTMS удалит такие и только такие высказывания, для которых P является элементом каждого обоснования. Поэтому, если высказывание Q имеет единственное обоснование $\{P, P \Rightarrow Q\}$, оно будет удалено; если оно имеет также дополнительное обоснование $\{P, P \vee R \Rightarrow Q\}$, оно также будет удалено; но если, кроме этого, оно имеет обоснование $\{R, P \vee R \Rightarrow Q\}$, оно будет сохранено. Таким образом, время, требуемое для извлечения высказывания P, зависит только от количества высказываний, полученных на основании P, а не от количества других высказываний, добавленных после того, как P было введено в базу знаний.

В системе JTMS предполагается, что высказывания, которые уже когда-то рассматривались, по-видимому, будут рассматриваться снова, поэтому вместо полного удаления из базы знаний некоторого высказывания после того, как оно теряет все обоснования, это высказывание просто отмечается как находящееся *вне* базы знаний. Если же какое-то последующее утверждение восстанавливает одно из обоснований, то это высказывание снова отмечается как находящееся *внутри* базы. Благодаря этому система JTMS позволяет сохранить все цепочки логического вывода, которые в ней используются, и не нуждается в повторном выводе высказываний после того, как некоторое обоснование вновь становится действительным.

Помимо исключения из базы знаний неправильной информации, системы поддержки истинности могут использоваться для ускорения анализа многочисленных гипотетических ситуаций. Предположим, например, что Олимпийский комитет Румынии выбирает площадки для проведения соревнований по плаванию, легкой атлетике и конному спорту для олимпийских игр 2048 года, которые должны проводиться в Румынии. Например, допустим, что первая гипотеза такова: *Site*(*Swimming*, *Pitesti*), *Site*(*Athletics*, *Bucharest*) и *Site*(*Equestrian*, *Arad*).

Потребуется провести большой объем рассуждений, чтобы определить логические следствия, а значит, и целесообразность этого выбора. Если же вместо этого потребуется рассмотреть вариант *Site*(*Athletics*, *Sibiu*), то система TMS позволит избавиться от необходимости начинать всю эту работу с нуля. Достаточно будет просто извлечь гипотезу *Site*(*Athletics*, *Bucharest*) и ввести *Site*(*Athletics*, *Sibiu*), после чего система TMS возьмет на себя весь необходимый пересмотр. А цепочки логического вывода, выработанные на основании выбора площадки в Бухаресте, могут повторно использоваться для Сибиу, при условии, что логические заключения остаются теми же самыми.

Особенно эффективное переключение контекста между гипотетическими мирами обеспечивает система поддержки истинности на основе предположения, или ►**ATMS** (*Assumption-based Truth Maintenance System*), предназначенная именно для подобных целей. В системе JTMS средства сопровождения обоснований позволяют быстро переходить от одного состояния к другому, выполнив лишь небольшой объем извлечений и вставок, но в них в любой момент времени представлено только одно состояние. В системе ATMS представлены *все* состояния, которые когда-либо рассматривались одновременно. Это означает, что в системе JTMS каждое высказывание обозначается как находящееся *вне* или *внутри* базы знаний, тогда как в системе ATMS для каждого высказывания отслеживается, какие предположения могли бы вынудить это высказывание стать истинным. Иными словами, каждое высказывание имеет метку, состоящую из множества множеств предположений. Это высказывание становится истинным только в том случае, если истинными являются все предположения в одном из множеств предположений.

Системы поддержки истинности предоставляют также механизм выработки ►**объяснений**. Формально объяснением высказывания P является такое множество высказываний E, что из E следует P. Если уже известно, что высказывания E истинны, то E просто предоставляет достаточную базу для доказательства того, что высказывание P также является истинным. Но объяснения могут также включать ►**предположения** — высказывания, в отношении которых неизвестно, являются ли они истинными, но которые были бы достаточными, чтобы доказать истинность P, если бы они были истинными. Например, если ваш автомобиль не заводится, у вас, вероятно, нет достаточной информации для убедительного доказательства причины этой проблемы. Но обоснованное объяснение может включать предположение, что аккумулятор разряжен. Оно, в сочетании со знаниями о

том, как работает двигатель автомобиля, позволяет объяснить наблюдаемое неправильное поведение автомобиля. В большинстве случаев предпочтительным является минимальное объяснение E; под этим подразумевается, что не существует строгого подмножества E, которое также было бы объяснением. Система ATMS может сформировать объяснения для проблемы "автомобиль не заводится", делая предположения (такие, как "в машине кончился бензин" или "аккумулятор разряжен") в любом желательном для пользователя порядке, даже если некоторые предположения противоречат друг другу. После этого достаточно посмотреть на метку, предусмотренную для высказывания "автомобиль не заводится", чтобы ознакомиться с множествами предположений, которые могли бы подтвердить это высказывание.

Точные алгоритмы, используемые для реализации систем поддержки истинности, являются довольно сложными, и здесь они не рассматриваются. Вычислительная сложность задачи поддержки истинности является по меньшей мере такой же, какая характерна для пропозиционального логического вывода, т.е. NP-трудной. Поэтому не следует рассчитывать на то, что подход на основе поддержки истинности окажется панацеей. Однако при продуманном использовании системы TMS могут обеспечить существенное повышение способности логической системы в отношении обработки сложных сред и гипотез.

Резюме

Описывая детали того, какие способы используются для представления различных знаний, авторы надеялись дать читателю возможность понять, как создаются реальные базы знаний и почувствовать интересные философские вопросы, которые при этом возникают. Основные идеи, высказанные в этой главе, перечислены ниже.

- Для крупномасштабного представления знаний требуется онтология общего назначения, позволяющая организовать и связать воедино различные специализированные области знаний.
- Онтология общего назначения должна охватывать широкий круг знаний и быть способной в принципе представить любую проблемную область.
- Построение крупномасштабной онтологии общего назначения является серьезной задачей, которая решена еще не полностью, хотя уже существующие системы кажутся достаточно надежными.
- В этой главе описана **верхняя онтология**, основанная на категориях и исчислении событий. Были рассмотрены категории, подкатегории, части, структурированные объекты, меры, вещества, события, время и пространство, изменения и убеждения.
- Естественные разновидности не могут быть полностью определены в логике, но свойства естественных разновидностей могут быть представлены.

- Действия, события и время могут быть представлены с помощью исчисления событий. Подобные представления позволяют агенту составлять последовательности действий и осуществлять логический вывод для определения, что в мире будет истинным после их выполнения.
- Были рассмотрены системы представления общего назначения, такие как **семантические сети** и **описательные логики**, позволяющие строить иерархии категорий. **Наследование** является важной формой логического вывода, которая позволяет логическим путем определять свойства объектов на основании данных об их принадлежности к категориям.
- **Допущение замкнутости мира**, будучи реализованным в логических программах, предоставляет простой способ избежать необходимости задавать большие объемы отрицательной информации. Такую информацию проще всего интерпретировать как **заданную по умолчанию**, которую можно переопределять с помощью дополнительной информации.
- **Немонотонные логики**, такие как **логика косвенного описания** и **логика умолчания**, предназначены для обеспечения возможности проведения рассуждений по умолчанию.
- **Системы поддержки истинности** позволяют эффективно осуществлять обновления и пересмотры баз знаний.
- Трудно построить большие онтологии вручную; извлечение знаний из текста облегчает эту задачу.

Библиографические и исторические заметки

Бриггс ([305], 1985) утверждает, что исследования в области представления знаний начались с классических работ индийских теоретиков, посвященных грамматике шастрических текстов на санскрите, относящихся к первому тысячелетию до нашей эры. Начало работы западных философов по этой теме можно проследить до 300 годов до н.э. в труде *Метафизика* Аристотеля. Безусловно, разработка технической терминологии в любой области независимо от формы также может рассматриваться как способ представления знаний.

Первые дискуссии о представлении в области искусственного интеллекта в основном сосредоточивались на "представлении *задач*", а не на "представлении *знаний*" (см., например, приведенное Амарелем ([40], 1968) обсуждение задачи с миссионерами и каннибалами). В 1970-х годах основной акцент в искусственном интеллекте был сделан на разработке "экспертных систем" (называемых также "системами, основанными на знаниях"), которые были способны при наличии соответствующих знаний в проблемной области достичь или превзойти производительность людей-экспертов при решении узко определенных задач. Например, первая экспертная система, DENDRAL (Фейгенбаум и др. [716], 1971; Линдси и др. [1417], 1980), интерпретировала выходные данные масс-спектрометра (прибора,

применяемого для анализа структуры органических химических веществ) так же точно, как и опытные химики. Хотя успех системы DENDRAL позволил сообществу исследователей по искусственному интеллекту убедиться в важности представления знаний, формальные средства представления, используемые в DENDRAL, были в высшей степени специализированными и относящимися только к данной проблемной области химии.

Со временем у исследователей появился интерес к стандартизации формальных средств представления знаний для толкований и онтологий, что позволило бы упростить процесс создания новых экспертных систем. В ходе решения этой проблемы им пришлось проникнуть на территорию, которая до сих пор исследовалась только философами, изучающими проблемы науки и языка. Принятое в искусственном интеллекте требование, согласно которому предлагаемые теории должны "работать", привело к более быстрому и глубокому прогрессу в решении проблем по сравнению с той ситуацией, когда эти проблемы рассматривались как принадлежащие исключительно к области философии (хотя такой подход время от времени приводил к "повторному изобретению колеса").

Но в какой степени можно доверять экспертным знаниям? Еще в 1955 году Пол Миль ([1555], 1955; см. также Гроув и Миль [926], 1996) изучал процессы принятия решений подготовленными экспертами при выполнении субъективных задач, таких как прогнозирование успеха учащегося в программе обучения или рецидив преступлений у преступника. В 19 из 20 исследований, которые он изучил, Миль обнаружил, что простые статистические алгоритмы обучения (такие, как линейная регрессия или наивный байесовский прогноз) предсказывают лучше, чем эксперты. Тетлок ([2195], 2017) также изучал экспертные знания и обнаружил их недостаточность в сложных случаях. Служба образовательного тестирования использует автоматизированную программу для анализа миллионов оценок, выставленных на экзаменах GMAT с 1999 года. Эта программа одобряет решение выносящих оценку людей в 97% случаев, — примерно на том же уровне, на котором приходят к согласию два оценивающих человека (Бурштейн и др. [350], 2001). (Это не означает, что программа понимает ответы, просто она может отличить хорошие от плохих так же хорошо, как и экзаменаторы.)

Первые попытки создания всеобъемлющих таксономий или классификаций, относятся к временам древности. Аристотель (384–322 до н.э.) настоятельно требовал применения правильных схем классификации и категоризации. В его *Органон*, сборник трудов Аристотеля по логике, собранный учениками Аристотеля после его смерти, включен трактат под названием *Категории*, в котором Аристотель попытался создать то, что мы теперь называем верхней онтологией. Он также ввел понятия **родов** и **видов** для классификации нижнего уровня. Современная система биологической классификации, включая использование "биноминальной номенклатуры" (в формальном смысле слова — классификации по родам и видам), была предложена шведским биологом Карлом Линнеем (1707–1778). Проблемы, связанные с естественными разновидностями и неточными границами категорий,

кроме многих других работ, рассматривались Витгенштейном ([2369], 1953), Квайном ([1831], 1953), Лакоффом ([1343], 1987) и Шварцем ([2013], 1977).

Обсуждение представления слов и понятий в глубоких нейронных сетях можно найти в главе 24, — в нем не только отходят от некоторых проблем строгой онтологии, но и жертвуют некоторой степенью точности. Мы до сих пор не знаем, как лучше объединить преимущества нейронных сетей и логической семантики для целей представления.

Продолжает расти интерес к крупномасштабным онтологиям, что задокументировано в Справочнике по онтологиям (Стааб [2121], 2004). В рамках проекта OpenCYC (Ленат и Гуха [1384], 1990; Матушек и др. [1516], 2006) была разработана онтология из 150 000 понятий, верхняя онтология которой сходна с представленной на рис. 10.1, а также со специфическими концепциями, подобными "OLED дисплей" и "iPhone", являющимися типами категории "сотовые телефоны", которая, в свою очередь, является типом категорий "потребительская электроника", "телефоны", "устройства беспроводной связи" и нескольких других. Проект NextKB расширил онтологию CYC и другие ресурсы, включая FrameNet и WordNet, до базы знаний почти с 3 миллионами фактов и предоставляет систему рассуждений FIRE для работы с ней (Форбус и др. [753], 2010).

В проекте DBpedia структурированные данные извлекаются из Википедии, в частности из Infoboxes — пар "атрибут/значение", которые прилагаются ко многим статьям в Википедии (Ву и Вельд [2390], 2008; Бизер и др. [226], 2007). По состоянию на 2015 год база знаний проекта DBpedia содержала 400 миллионов фактов о 4 миллионах объектов только в английской версии; принимая в расчет все 110 языков, получаем 1,5 миллиарда фактов (Леманн и др. [1379], 2015).

Рабочая группа IEEE P1600.1 создала SUMO — *Suggested Upper Merged Ontology* (Нильс и Пиэс [1684], 2001; Пиэс и Нильс [1761], 2002) — примерно с 1000 терминов в верхней онтологии и связями более чем с 20 000 терминов, специфических для области определения. Стоффель и соавт. ([2133], 1997) описывают алгоритмы для эффективного управления очень большой онтологией. Обзор методов извлечения знаний из веб-страниц дается Этциони и соавт. ([703], 2008).

В Сети появляются новые языки представления. Язык RDF (Брикли и Гуха [304], 2004) позволяет делать утверждения в форме реляционных триплетов и предоставляет некоторые средства для развития значения имен во времени. Язык OWL (Смит и др. [2097], 2004) — это язык описательной логики, поддерживающий логический вывод по этим триплетам. На настоящий момент их использование кажется обратно пропорциональным репрезентативной сложности: традиционные HTML- и CSS-форматы используются для представления более 99% веб-контента, за ними следуют простейшие схемы представления, такие как RDFa (Адида и Бирбек [15], 2008), и микроформаты (Харе [1221], 2006; Пател-Шнейдер [1743], 2014), в которых разметка HTML и XHTML используется для добавления атрибутов к тексту веб-страниц. Использование сложных онтологий RDF и OWL еще не получило распространения, и полное видение семантического веба

(Бернерс-Ли и др. [190], 2001) пока не реализовано. Конференции *Formal Ontology in Information Systems* (FOIS) по формальной онтологии в информационных системах охватывают как общие, так и предметные онтологии.

Таксономия, используемая в этой главе, была разработана авторами данной книги и частично основана на опыте их участия в проекте CYC и на работах Хванга и Шуберта ([1110], 1993), а также Дэвиса ([534], 1990; [535], 2005). Воодушевляющее обсуждение общего проекта представления обыденных знаний приведено в книге Хейса "The Naive Physics Manifesto" ([989], 1978; [992], 1985).

Успешные глубокие онтологии в пределах определенной области включают в себя проект Gene Ontology (Консорциум Жене-онтология [834], 2008) и Chemical Markup Language (Мюррей-Раст и др. [1645], 2003). Сомнения в возможности создания единой онтологии для всех знаний выражены Докторовым ([628], 2001), Грубером ([927], 2004), Халеви и соавт. ([950], 2009), а также Смитом ([2088], 2004).

Исчисление событий, разработанное Ковальски и Серготом ([1298], 1986), было введено как инструмент представления непрерывного времени, а позднее появилось еще несколько его вариантов (Садри и Ковальски [1960], 1995; Шанахан [2034], 1997) и обзоров (Шанахан [2035], 1999; Мюллер [1631], 2006). Джеймс Аллен предложил использовать для этой же цели временные интервалы ([33], 1984), указывая, что интервалы являются гораздо более естественным средством формирования рассуждений о продолжительных и одновременных событиях, чем ситуации. В работе ван Ламбалгена и Хамма ([2258], 2005) показано, как логика событий влияет на язык, который мы используем, говоря о событиях. Альтернативой исчислениям событий и ситуаций является исчисление флюент (Тильшер [2202], 1999), в котором овеществляются факты, из которых образованы состояния.

Питер Ладкин ([1335], 1986; [1336], 1986) ввел "вогнутые" временные интервалы (интервалы с перерывами; по сути, объединения обычных "выпуклых" временных интервалов) и применил методы математической абстрактной алгебры для представления времени. Аллен ([34], 1991) систематически исследовал широкий спектр методов, которые могут применяться для представления времени; ван Бик и Манчак ([2248], 1996) проанализировали алгоритмы для временных рассуждений. Между онтологией на основе событий, приведенной в данной главе, и анализом событий, выполненным философом Дональдом Давидсоном ([531], 1980), имеются весьма важные аналогии. К тому же типу относятся ►**хронологии** (*history*), предложенные в работе Патрика Хейса ([991], 1985) по онтологии вневременных событий (*liquid event*) и ►**хроники** (*chronicles*) в теории МакДермотта ([1543], 1985) о планах, оказавшей также значительное влияние в той области, которой посвящена данная глава.

Проблема онтологического статуса веществ имеет долгую историю. Платон считал, что вещества — это абстрактные сущности, полностью отличающиеся от физических объектов; с его точки зрения следовало бы сказать, что кусок масла *сделан* из масла — $MadeOf(Butter_3, Butter)$, — а не что кусок масла является *элементом* множества масла — $Butter_3 \in Butter$. Такая идея ведет к созданию

иерархии веществ, в которой, например, несоленое масло *UnsaltedButter* является более конкретно определенным веществом, чем само масло *Butter*. Научная позиция, принятая в этой главе, согласно которой вещества представляют собой категорию объектов, была обоснована Ричардом Монтегю ([1607], 1973). Также эта позиция была принята и в проекте CYC. На эту позицию Копеланд в работе [476] (1993) предпринял серьезную, но не настолько уж неотразимую атаку.

Альтернативный подход, упомянутый в данной главе, согласно которому масло представляет собой единственный объект, состоящий из всех маслоподобных объектов во вселенной, был первоначально предложен польским логиком Лесневским ([1389], 1916). В разработанной им **мереологии** (это название происходит от греческого слова, обозначающего "часть") используется отношение "часть–целое" в качестве замены математической теории множеств в целях устранения таких абстрактных сущностей, как множества. Более удобное для чтения изложение этих идей приведено Леонардом и Гудманом ([1385], 1940), а в книге Гудмана "The Structure of Appearance" ([901], 1977) эти идеи применяются для решения различных проблем в области представления знаний.

Хотя в некоторых аспектах мереологический подход является весьма громоздким — например, в нем требуется отдельный механизм наследования, основанный на отношениях "часть–целое", — он получил поддержку Квайна ([1832], 1960). Гарри Бант ([343], 1985) провел широкий анализ перспектив использования этого подхода в области представления знаний. Касати и Варзи ([377], 1999) охватывают части, целые и общую теорию пространственных положений.

Существует три основных подхода к изучению ментальных объектов. Тот, который был выбран в этой главе, основан на модальной логике и возможных мирах и является классическим подходом со стороны философии (Хинтикка [1024], 1962; Крипке [1309], 1963; Хьюджес и Крессвелл [1093], 1996). В книге "*Reasoning about Knowledge*" (Фагин и др. [708], 1995) предоставлено подробное введение, а Гордон и Хоббс ([904], 2017) предлагают формальную теорию психологии здравого смысла.

Второй подход — это теория первого порядка, в которой ментальные объекты являются флюентами. В работах Дэвиса ([535], 2005) и Дэвиса и Моргенштерна ([541], 2005) описан этот подход. Он основан на формализме возможных миров и строится на работах Роберта Мура ([1616], 1980; [1617], 1985).

Третий подход — это **синтаксическая теория**, в которой ментальные объекты представлены символьными строками. Строка является просто комплексным термом, представляющим собой список символов, поэтому утверждение *CanFly(Clark)* может быть представлено с помощью списка символов [*C*, *a*, *n*, *F*, *l*, *y*, (, *C*, *l*, *a*, *r*, *k*,)]. Синтаксическая теория ментальных объектов впервые глубоко изучалась Капланом и Монтегю ([1185], 1960), которые показали, что эта теория может приводить к парадоксам, если при использовании ее средств не соблюдается чрезвычайная осторожность. Эрни Дэвис ([534], 1990) дал превосходное сравнение синтаксической и модальной теорий знания. Пнуэли ([1800], 1977)

описывает использование временной логики для рассуждений о программах, — эта работа принесла ему премию Тьюринга и позднее была расширена Варди ([2263], 1996). Литтман и соавт. (2017) показали, что временная логика может быть хорошим языком определения целей для робота, обучаемого по методу обучения с подкреплением, позволяя человеку легко определять эти цели и обобщать их на различные окружающие среды.

Греческий философ Порфирий (ок. 234–305 н.э.) в своих комментариях к трактату Аристотеля *Категории* продемонстрировал то, что может рассматриваться как первая семантическая сеть. Чарльз С. Пирс ([1766], 1909) разработал экзистенциальные графы, которые могут рассматриваться как первое формальное определение семантической сети с использованием современной логики. Инициатором исследований по семантическим сетям, проводимых в рамках искусственного интеллекта, был Росс Квиллиан ([1830], 1961), основным стимулом для которого был интерес к человеческой памяти и языковой обработке. В важной статье Марвина Мински ([1582], 1975) представлена одна из версий семантических сетей, называемых **фреймами**; фреймы служили для представления объектов или категорий и характеризовались наличием атрибутов и отношений с другими объектами или категориями.

Вопросы семантики приобрели особую остроту применительно к семантическим сетям, разработанным Квиллианом (и теми, кто стал последователем предложенного им подхода), в связи с тем, что в них использовались вездесущие и весьма неопределенные "связи *IS-A*". Знаменитая статья Вудса "What's In a Link?" ([2376], 1975) привлекла внимание исследователей в области искусственного интеллекта к тому, что в формальных системах представления знаний семантика должна быть точно определена. Рон Брачман ([283], 1979) провел исследования по этой проблеме и предложил некоторые решения. Патрик Хейс в статье *The Logic of Frames* ([990], 1979) провел еще более глубокие исследования и сформулировал утверждение, что "так называемые «фреймы» по большей части представляют собой просто новые синтаксические обозначения для фрагментов логики первого порядка". Дрю Макдермотт в работе *Tarskian Semantics, or, No Notation without Denotation!* ([1542], 1978) доказывал, что модельно-теоретический подход к семантике, используемый в логике первого порядка, должен быть распространен на все формальные системы представления знаний. Однако эта идея является внутренне противоречивой, — примечательно, что сам Макдермотт пересмотрел свою позицию в работе *A Critique of Pure Reason* ([1543], 1987). Селман и Левеск ([2024], 1993) обсудили сложность наследования с исключениями и показали, что в большинстве формулировок эта задача является NP-полной.

Описательные логики представляют собой полезное подмножество логики первого порядка, для которой задача логического вывода является осуществимой с помощью вычислительных средств. Гектор Левеск и Рон Брачман ([1392], 1987) показали, что определенные виды использования дизъюнкции и отрицания, в первую очередь, ответственны за неразрешимость логического вывода. Это позволило

гораздо лучше понять зависимости между сложностью и выразительностью в системах формирования рассуждений. Общий итог современного состояния дел в этой области подведен в работе Кальванезе и соавт. ([358], 1999), а Баадер и соавт. ([94], 2007) предоставили всеобъемлющее руководство по описательной логике.

Три основные формальные системы, предназначенные для использования в немонотонном логическом выводе — косвенное описание (МасКарти [1532], 1980), логика умолчания (Рейтер [1869], 1980) и модальная немонотонная логика (МакДермотт и Дойл [1545], 1980), — были предложены в одном специальном выпуске *AI Journal*. Дельгранде и Шауб в работе [597] (2003) обсуждают достоинства этих вариантов, предоставляя 25-летний ретроспективный взгляд на их историю. Программирование множества ответов может рассматриваться как расширение понятия отрицания как недостижения цели или как уточнение понятия косвенного описания. Основополагающая теория семантики стабильных моделей была предложена Гельфондом и Лившицем в работе [829] (1988), а ведущими системами программирования множества ответов являются DLV (Эйтер и др. [679], 1998) и SMODELS (Нимелё и др. [1682], 2000). Лифшиц в работе [1411] (2001) обсуждает возможность использования программирования множества ответов для планирования. Бревка и соавт. в [303] (1997) приводят хороший обзор различных подходов к немонотонной логике. Кларк в [445] (1978) предлагает свой подход к логическому программированию, основанный на отрицании как недостижении цели, и анализирует дополнение Кларка. Лифшиц в [1411] (2001) обсуждает применение программирования множества ответов в планировании. Целый ряд систем формирования немонотонных рассуждений, основанных на логическом программировании, описан в трудах конференции *Logic Programming and Nonmonotonic Reasoning* (LPNMR).

Исследования в области систем поддержки истинности начались с создания систем TMS (Дойл [645], 1979) и RUP (МакАлистер [1521], 1980), которые по существу представляли собой системы JTMS. Форбус и де Клер в [752] (1993) подробно объясняют, как системы TMS могут применяться в приложениях искусственного интеллекта. Нейяк и Вильямс ([1660], 1997) показали, как использование эффективной инкрементальной системы TMS под названием "ITMS" обеспечило возможность планирования операций космического аппарата агентства NASA в режиме реального времени.

По понятным причинам в этой главе подробно не рассматривается *каждая* область представления знаний. Ниже описаны три основные темы, которые в ней не представлены.

► **Качественная физика**. Это подобласть представления знаний, относящаяся в основном к формированию логической, нечисловой теории физических объектов и процессов. Этот термин был предложен Йоханом де Клером ([557], 1975), хотя вполне можно считать, что это направление исследований началось с создания Фалманом ([709], 1974) программы BUILD — развитого планировщика для

построения сложных башен из блоков. В процессе разработки Фалман установил, что основная часть усилий (по его оценке, 80%) уходит на моделирование физических свойств мира блоков в целях вычисления устойчивости различных субфрагментов структур блоков, а не на само планирование как таковое. Он сформулировал набросок гипотетического процесса, подобного проведению рассуждений в рамках наивной физики, для объяснения причин того, почему дети младшего возраста способны решать задачи, подобные рассматриваемым в программе Build, без обращения к быстродействующей арифметике с плавающей точкой, которая используется в физическом моделировании программой Build. Хейс ([991], 1985) применил "истории" (четырехмерные фрагменты пространства-времени, аналогичные событиям Дэвидсона) для построения довольно сложной наивной физической теории жидкостей. Дэвис в [538] (2008) предлагает обновление онтологии жидкостей, описывающее заливку жидкостей в контейнеры.

В работах де Клера и Брауна ([559], 1985), Форбуса ([751], 1985) и Куперса ([1321], 1985) представлены системы, разработанные независимо друг от друга и почти одновременно, способные рассуждать о физических системах на основе качественных абстракций соответствующих физических уравнений. Вскоре качественная физика была разработана до такой степени, что с ее помощью стало возможным анализировать сложные физические системы, разнообразие которых весьма впечатляет (Йип [2405], 1991). Качественные методы использовались для разработки новых конструкций часов, дворников для ветрового стекла и шестиногих шагающих машин (Субраманиан и Ванг [2146], 1994). Хорошим введением в эту область являются сборник статей *Readings in Qualitative Reasoning about Physical Systems* (Вельд и де Клер [2312], 1990), энциклопедическая статья Куперса ([1323], 2001) и справочная статья Дэвиса ([537], 2007).

► **Пространственные рассуждения**. Рассуждения, необходимые для навигации в мире вампуса, являются тривиальными в сравнении с тем, что требуется в богатой пространственной структуре реального мира. Результаты первых серьезных попыток описать обыденные рассуждения о пространстве приведены в работах Эрнеста Дэвиса ([533], 1986; [534], 1990). В исчислении региональных связей Кона и соавт. ([461], 1997) поддерживается определенная форма качественных пространственных рассуждений, — эти исследования привели к созданию географических информационных систем нового типа (Девис [536], 2006). Как и с помощью качественной физики, агент, так сказать, многое может сделать без обращения к полному метрическому представлению.

► **Психологические рассуждения**. Психологические рассуждения охватывают проблему разработки для искусственных агентов работоспособных *психологических моделей*, предназначенных для использования в их рассуждениях о себе и других агентах. Такие рассуждения часто основаны на так называемой народной психологии (*folk psychology*) — теории, которую люди, как полагают, используют в рассуждениях о себе и других людях. Когда исследователи в области

искусственного интеллекта снабжают своих искусственных агентов психологическими теориями для формирования рассуждений о других агентах, эти теории часто основаны на описании исследователями проектов самих этих логических агентов. В настоящее время психологические рассуждения оказываются наиболее полезными в контексте понимания естественного языка, где понимание намерений говорящего имеет первостепенное значение.

В сборнике Минкера [1581] (2001) собраны статьи ведущих исследователей в области представления знаний, суммируя итоги 40 лет работы в этой области. В материалах международных конференций *Principles of Knowledge Representation and Reasoning* собраны все новейшие источники по результатам работы в этой области. Сборники *Readings in Knowledge Representation* (Брачман и Левеск [284], 1985) и *Formal Theories of the Commonsense World* (Хоббс и Мур [1037], 1985) являются превосходными антологиями по представлению знаний, — первый в большей степени фокусируется на исторически важных документах, а второй — на накоплении самих знаний. В книгах Девиса ([534], 1990), Стефика ([2125], 1995) и Совы ([2111], 1999) предлагаются учебные материалы в области представления знаний, а издание ван Хармелена и соавт. ([2253], 2007) — это отличный справочник по теме. Дэвис и Моргенштерн ([540], 2004) предоставляют отредактированный специальный выпуск журнала "Искусственный интеллект" по данной теме. В работе [539] (2017) Дэвис предлагает обзор по логике обыденных рассуждений. Проводимая раз в два года конференция *Theoretical Aspects of Reasoning About Knowledge* (TARK) посвящена применению теории знаний в области ИИ, экономике и распределенных системах.

Упражнения

10.1. Создайте онтологию в логике первого порядка для игры в крестики-нолики. Онтология должна содержать ситуации, действия, квадраты, игроков, отметки (**X**, **O** и пробел), а также понятие выигрыша, проигрыша или ничьей в игре. Также определите понятие гарантированного выигрыша (или ничьей): позиция, с которой игрок может гарантированно выиграть (или сыграть вничью) при правильной последовательности действий. Напишите аксиомы для этой области определения. (*Примечание.* Аксиомы, перечисляющие различные квадраты и характеризующие выигрышные позиции, довольно длинные. Не требуется выписывать их полностью, — достаточно четко указать, как они выглядят.)

10.2. Требуется создать систему консультирования студентов по компьютерным наукам о том, какие курсы необходимо прослушать в течение длительного периода времени, чтобы удовлетворить все требования программы. (Используйте те требования, которые подходят для вашего учебного заведения.) Сначала выберите словарь для представления всей информации, а затем представьте ее. Далее сформулируйте запрос к системе, в ответ на который будет возвращена допустимая программа обучения в качестве решения. Необходимо предусмотреть некоторую настройку для отдельных студентов, в которой система будет

спрашивать, какие курсы или их эквиваленты студент уже прослушал, и исключить возможность создания программ, в которых эти курсы будут повторяться. Предложите способы улучшения своей системы, например принимать во внимание знания о предпочтениях студентов, нагрузке, хороших и плохих преподавателях и т.д. Для каждого вида знаний объясните, как это можно выразить логически. Может ли ваша система легко включить эту информацию, чтобы найти все возможные программы обучения для студента? Сможет ли она найти лучшую программу?

10.3. На рис. 10.1 приведены верхние уровни иерархии онтологии для всех понятий. Дополните ее, чтобы включить как можно больше реальных категорий. Удобным способом выполнения этого задания является описание всего, с чем вам приходится сталкиваться в повседневной жизни. К этому относятся и объекты, и события. Начните с утреннего пробуждения и последовательно учитывайте все, что вы видите, чего касаетесь, что делаете и о чем думаете. Например, случайно выбранный день может привести к такой цепочке: музыка, новости, молоко, ходьба, поездка, заправка топливом, вестибюль университета, ковер, разговор, профессор Фейтмен, цыпленок под соусом карри, обожженный язык, 7 долларов, солнце, дневная газета и т.д. Вы должны подготовить единую схему иерархии (на большом листе бумаги), а также список объектов и категорий с отношениями, которым соответствуют элементы каждой категории. Каждый объект должен находиться в некоторой категории, а каждая категория должна относиться к этой иерархии.

10.4. Разработайте систему представления для рассуждений об окнах в компьютерном оконном интерфейсе. В частности, ваше представление должно позволять описать следующее.

- Состояние окна: свернуто, отображается или отсутствует.
- Какое окно (если есть) является активным.
- Положение каждого окна в данный момент.
- Порядок (от переднего плана к заднему) перекрывающихся окон.
- Действия по открытию, закрытию, изменению размера и перемещению окон, изменению состояния окна и его выведению на передний план (активизации). Относитесь к этим действиям как к атомарным, т.е. не занимайтесь вопросом их соотнесения с действиями мыши. Запишите аксиомы, описывающие влияние действий на флюенты. Можете использовать исчисление событий или исчисление ситуаций.

Предположим, что ваша онтология будет включать ситуации, действия, целые числа (для представления координат x и y) и окна. Определите язык для этой онтологии, т.е. список констант, функциональных символов и предикатов с описанием каждого из них на русском языке. Если потребуется добавить в онтологию больше категорий (например, пиксели), можете это сделать, но обязательно отметьте данный факт в сопроводительной записке. Вы также можете (и должны) использовать символы, определенные в тексте главы, но обязательно перечислите их явно.

10.5. Представьте следующее на языке, который был разработан в упражнении 10.4.

а) В ситуации S_0 окно W_1 находится под окном W_2, но выступает из-под него слева и справа. Не задавая для них точные координаты, опишите эту общую ситуацию.

б) Если окно отображается на экране, то его верхний край всегда выше нижнего.

в) Окно w после открытия будет отображено на экране.

г) Окно может быть свернуто, только если оно отображается.

10.6. Представьте следующее на языке, который был разработан в упражнении 10.4.

а) В ситуации S_0 окно W_1 находится под окном W_2, но выступает из-под него сверху и снизу. Не задавая для них точные координаты, опишите эту общую ситуацию.

б) Если окно отображается на экране, то его верхний край всегда выше нижнего.

в) Окно w после открытия будет отображено на экране.

г) Окно может быть свернуто, только если оно отображается.

10.7. Ваша задача состоит в том, чтобы в логической форме представить достаточный объем знаний, позволяющий ответить на ряд вопросов о следующем простом высказывании: *Вчера Джон пошел в супермаркет Safeway и купил два фунта помидоров и фунт говяжьего фарша.* Начните с попытки представить содержимое этого высказывания в виде ряда утверждений. Запишите высказывания, имеющие простую логическую структуру (например, утверждения о том, что объекты имеют определенные свойства, связаны определенными отношениями или что объекты, удовлетворяющие некоторому свойству, удовлетворяют также какому-то другому свойству). Чтобы начать, попробуйте ответить на следующие вопросы.

– Какие классы, объекты и отношения вам потребуются? Есть ли у них родительские объекты, сестринские объекты и т.д.? (Кроме всего прочего, вам могут потребоваться события и их упорядочение во времени.)

– В каком месте эти объекты должны войти в более общую иерархию?

– Каковы ограничения и взаимосвязи между ними?

– Насколько подробно следует определять каждое из различных понятий?

Чтобы дать ответы на приведенные ниже вопросы, созданная вами база знаний должна включать некоторые фоновые знания. Потребуется разобраться, какого рода товары бывают в супермаркете, что связано с покупкой выбранных товаров, для чего могут использоваться покупки и т.д. Попытайтесь сделать применяемое вами представление настолько общим, насколько это возможно. Приведем тривиальный пример: не следует вводить в базу знаний высказывание "Люди покупают продукты в супермаркете Safeway", поскольку это не позволит учесть существование тех, кто покупает продукты в другом супермаркете. Также не превращайте вопросы в ответы. Например, в п. *в* спрашивается "Купил ли Джон какие-либо мясные продукты?", а не "Купил ли Джон фунт говяжьего фарша?" Наметьте цепочки рассуждений, которые позволят ответить на приведенные вопросы. Если это возможно, используйте какую-то систему формирования логических рассуждений, чтобы продемонстрировать достаточность вашей базы знаний. Многие из составленных вами высказываний в реальности могут оказаться лишь приблизительно правильными, но не следует об этом слишком беспокоиться; идея состоит в том, чтобы извлечь обыденные знания (на уровне здравого смысла), которые в принципе позволят вам ответить на приведенные вопросы. Задача составления действительно полного ответа на многие из этих вопросов является чрезвычайно трудной и, по-видимому, выходит за рамки современных средств представления знаний. Но вы должны быть способны составить непротиворечивое множество аксиом для получения ответов на приведенный здесь ограниченный круг вопросов.

а) Джон — ребенок или взрослый? [*Взрослый*]

б) У Джона теперь имеется не меньше двух помидоров? [*Да*]

в) Купил ли Джон какие-либо мясные продукты? [*Да*]

г) Если бы Мэри покупала помидоры в то же время, что и Джон, он увидел бы ее? [*Да*]
д) Помидоры были сделаны в супермаркете? [*Нет*]
е) Что Джон собирается сделать с помидорами? [*Съесть их*]
ж) В супермаркете Safeway продаются дезодоранты? [*Да*]
з) Джон принес немного денег или кредитную карточку в супермаркет? [*Да*]
и) Стало ли у Джона меньше денег после посещения супермаркета? [*Да*]

10.8. Внесите необходимые дополнения или изменения в свою базу знаний, разработанную в упражнении 10.8, так, чтобы можно было получить ответы на приведенные ниже вопросы. Включите в свой отчет описание внесенных изменений, объяснив, почему они были необходимы, были они незначительными или крупными и какие вопросы потребуют внесения дальнейших изменений.

а) Находились ли другие люди в супермаркете Safeway в то время, когда там был Джон? [*Да — работники супермаркета!*]
б) Джон вегетарианец? [*Нет*]
в) Кому принадлежит дезодорант в супермаркете Safeway? [*Корпорации Safeway*]
г) Имеется ли у Джона унция говяжьего фарша? [*Да*]
д) Есть ли бензин на расположенной рядом бензозаправочной станции Shell? [*Да*]
е) Имеются ли помидоры в багажнике автомобиля Джона? [*Да*]

10.9. Представьте приведенные ниже семь высказываний, используя и дополняя средства представления, описанные в этой главе.

а) Вода находится в жидком состоянии при температуре от 0 до 100 градусов Цельсия.
б) Вода кипит при температуре 100 градусов Цельсия.
в) Вода в бутылке Джона для воды замерзла.
г) Минеральная вода "Перрье" — это тоже вода.
д) В бутылке Джона для воды находится минеральная вода "Перрье".
е) Все жидкости имеют определенную температуру замерзания.
ж) Литр воды весит больше, чем литр алкоголя.

10.10. Запишите определения для следующих предикатов.

а) *ExhaustivePartDecomposition*
б) *PartPartition*
в) *PartwiseDisjoint*

Эти определения должны быть аналогичны определениям для предикатов *ExhaustiveDecomposition*, *Partition* и *Disjoint*. Является ли, по вашему мнению, общезначимым выражение *PartPartition*(*s*, *BunchOf*(*s*))? Если да, докажите его общезначимость; в противном случае приведите контрпример и определите достаточные условия, при которых это выражение становится истинным.

10.11. Альтернативная схема представления мер предусматривает применение функции единиц к абстрактному объекту длины. В такой схеме следует писать $Inches(Length(L_1)) = 1{,}5$. Как выглядит эта схема по сравнению с той, которая описана в данной главе? Заслуживают внимания такие вопросы, как аксиомы преобразования, имена для абстрактных количеств (такие, как "50 долларов") и сравнения абстрактных мер в различных единицах (например, "50 дюймов больше 50 сантиметров").

10.12. Запишите множество высказываний, позволяющих рассчитать цену отдельного помидора (или другого объекта), если указана цена за килограмм. Дополните эту теорию, чтобы иметь возможность рассчитать цену пакета помидоров.

10.13. Введите правила, позволяющие расширить определение предиката *Name*(*s*, *c*) таким образом, чтобы строка, такая как "портативный компьютер", отвечала названиям соответствующих категорий из множества магазинов. Попробуйте сделать свое определение общим. Проверьте это, посетив сайты десяти интернет-магазинов и уточнив названия категорий, которые они дают для трех разных категорий. Например, для категории ноутбуков мы нашли названия "Ноутбуки", "Лаптопы", "Переносные компьютеры", "Ноутбук", "Лаптопы и ноутбуки" и "Переносные ПК". Одни из них могут быть представлены явными фактами *Name*, тогда как другие могут быть представлены высказываниями для обработки множественного числа, оформления перечислений и т.д.

10.14. Напишите аксиомы исчисления событий, необходимые для описания действий в мире вампуса.

10.15. Определите отношение алгебры интервалов, которое выполняется между каждой парой следующих реальных событий.

- *LK*: жизнь президента Кеннеди.
- *IK*: детство президента Кеннеди.
- *PK*: президентство президента Кеннеди.
- *LJ*: жизнь президента Джонсона.
- *PJ*: президентство президента Джонсона.
- *LO*: жизнь президента Обамы.

10.16. В данном упражнении рассматривается проблема планирования маршрута для робота, который должен добраться из одного города в другой. Основным действием, предпринимаемым роботом, является действие *Go*(*x*, *y*), которое позволяет ему добраться из города *x* в город *y*, если между ними есть прямой путь. Предикат *Road*(*x*,*y*) принимает истинное значение тогда и только тогда, когда есть прямой путь из города *x* в город *y*. Если он существует, то предикат *Distance*(*x*,*y*) определяет длину этого пути (см. карту на рис. 3.1). Робот начинает движение из Арада и должен достичь Бухареста.

а) Запишите подходящее логическое описание начальной ситуации для робота.

б) Запишите подходящий логический запрос, решения которого могут служить описаниями возможных путей к цели.

в) Запишите высказывание, описывающее действие *Go*.

г) Теперь предположим, что в пути робот расходует топливо в количестве 0,02 галлона на милю. Робот начинает движение с 20-ю галлонами топлива. Дополните разработанное вами представление с учетом этих соображений.

д) Теперь предположим, что в некоторых городах есть автозаправочные станции, на которых робот может заполнить топливный бак. Расширьте свое представление и запишите все правила, необходимые для описания АЗС, включая действие *Fillup* (заправка).

10.17. Исследуйте способы дополнения исчисления событий, позволяющие учитывать в нем одновременные события. Можно ли избежать комбинаторного взрыва количества аксиом?

10.18. Разработайте представление для курсов обмена валют, допускающее ежесуточные колебания курсов.

10.19. Определите предикат *Fixed*, где *Fixed*(*Location*(*x*)) означает, что местонахождение объекта *x* постоянно во времени.

10.20. Опишите событие обмена некоторого объекта на какой-то другой. Опишите покупку как разновидность обмена, в которой одним из объектов, участвующих в обмене, является некоторая денежная сумма.

10.21. В двух предыдущих упражнениях подразумевается использование довольно примитивного понятия владения. Например, покупатель начинает с владения денежными купюрами. Такое описание процесса покупки становится неадекватным, если, например, деньги покупателя хранятся в банке, поскольку у него на руках больше нет какой-то конкретной совокупности денежных купюр, которыми он владеет. Ситуация становится еще более сложной, если учесть возможность займа, сдачи в аренду, взятия в аренду и передачи в залог. Исследуйте различные обыденные и юридические понятия владения и предложите схему, с помощью которой они могут быть представлены формально.

10.22. Рассмотрим игру, в которую играют с колодой всего из 8 карт: 4 тузов и 4 королей. Трое игроков, Алиса, Боб и Карл, получают по две карты. Не глядя на них, они помещают эти карты себе на лоб, чтобы другие игроки могли их видеть. Затем игроки по очереди либо объявляют, что знают, какие карты у них на лбу, и тем самым выигрывают игру, либо говорят: "Я не знаю". Все знают, что игроки правдивы и правильно рассуждают на основании своих убеждений.

а) *Игра 1.* Алиса и Боб сказали: "Я не знаю". Карл видит, что у Алисы два туза (А-А), а у Боба два короля (К-К). Что должен сказать Карл? (*Подсказка.* Рассмотрите все три возможных для Карла случая: А-А, К-К, А-К.)

б) Опишите каждый этап *Игры 1*, используя обозначения модальной логики.

в) *Игра 2.* Карл, Алиса и Боб на первом ходу сказали: "Я не знаю". Алиса держит К-К, а Боб держит А-К. Что должен сказать Карл на своем втором ходу?

г) *Игра 3.* Алиса, Карл и Боб на первом ходу говорят: "Я не знаю", как и Алиса на своем втором ходу. Алиса и Боб оба держат А-К. Что должен сказать Карл?

д) Докажите, что в этой игре всегда будет победитель.

10.23. Предположение о логическом всеведении, обсуждавшееся в разделе 10.4, конечно же, неверно ни для каких реальных рассуждений. Скорее, это *идеализация* процесса рассуждения, которая может быть более или менее приемлемой в зависимости от приложений. Обсудите обоснованность такого предположения для каждого из следующих случаев использования рассуждений о знаниях.

а) Частичное знание в играх с противником, например в карточных играх. Здесь один игрок должен рассуждать о том, что его оппонент знает о состоянии игры.

б) Шахматы с часами. Здесь игрок может рассуждать об ограничениях в отношении способности своего оппонента или его собственной способности найти лучший ход за оставшееся время. Например, если у игрока A осталось намного больше времени, чем у игрока B, то игрок A иногда делает ход, значительно усложняющий ситуацию в надежде получить преимущество, поскольку у него имеется больше времени для выработки правильной стратегии.

в) Торговый агент, действующий в такой среде, где есть затраты на сбор информации.

г) Рассуждения о криптографии с открытым ключом, основанной на неразрешимости определенных вычислительных задач.

10.24. Предположение о логическом всеведении, обсуждавшееся в разделе 10.4, конечно же, неверно ни для каких реальных рассуждений. Скорее, это *идеализация* процесса

рассуждения, которая может быть более или менее приемлемой в зависимости от приложений. Обсудите обоснованность такого предположения для каждого из следующих случаев использования рассуждений о знаниях.

а) Шахматы с часами. Здесь игрок может рассуждать об ограничениях в отношении способности своего оппонента или его собственной способности найти лучший ход за оставшееся время. Например, если у игрока *A* осталось намного больше времени, чем у игрока *B*, то игрок *A* иногда делает ход, значительно усложняющий ситуацию в надежде получить преимущество, поскольку у него имеется больше времени для выработки правильной стратегии.

б) Торговый агент, действующий в такой среде, где есть затраты на сбор информации.

в) Автоматизированная программа обучения математике, рассуждающая о том, что студенты понимают.

г) Рассуждения о криптографии с открытым ключом, основанной на неразрешимости определенных вычислительных задач.

10.25. Переведите приведенное ниже выражение описательной логики (см. раздел 10.5.2) на язык логики первого порядка и прокомментируйте полученный результат.

And(*Man*, *AtLeast*(3, *Son*), *AtMost*(2, *Daughter*),
All(*Son*, *And*(*Unemployed*, *Married*, *All*(*Spouse*, *Doctor*))),
All(*Daughter*, *And*(*Professor*, *Fills*(*Department*, *Physics*, *Math*))))

10.26. Напомним, что информация о наследовании в семантических сетях может быть логически представлена подходящими импликативными высказываниями. В этом упражнении исследуется эффективность использования таких предложений для наследования.

а) Рассмотрим информацию в каталоге подержанных автомобилей, таком как "Синяя книга" Келли, — например, о том, что фургоны Dodge 1973 года (или, возможно, еще какого-то) стоят 575 долларов. Предположим, что вся эта информация (для 11 000 моделей) закодирована в виде логических высказываний, как предлагается в этой главе. Запишите три таких высказывания, в том числе для фургонов Dodge 1973 года. Как бы вы использовали эти высказывания для определения стоимости конкретного автомобиля при наличии системы доказательства теорем с обратным логическим выводом, подобной системе Prolog?

б) Сравните временную эффективность метода обратного логического вывода для решения данной задачи с методом наследования, используемым в семантических сетях.

в) Объясните, как прямой логический вывод позволяет логической системе эффективно решить ту же проблему в предположении, что база знаний содержит только 11 000 высказываний о ценах.

г) Опишите ситуацию, в которой ни прямой, ни обратный логический вывод по высказываниям не позволяет эффективно обработать запрос о цене отдельного автомобиля.

д) Можете ли вы предложить решение, позволяющее во всех случаях эффективно решать этот тип запросов в логических системах? (*Подсказка.* Помните, что две машины одного года и модели имеют одинаковую цену.)

10.27. Можно предположить, что в семантических сетях синтаксическое различие между ссылками без пометок и ссылками с пометками в прямоугольниках с одиночным

контуром не является необходимым, поскольку ссылки в прямоугольниках с одиночным контуром всегда привязаны к категориям. Алгоритм наследования может просто предполагать, что ссылка без пометки, прикрепленная к категории, предназначена для применения ко всем членам этой категории. Покажите, что этот аргумент ошибочен, и приведите примеры ошибок, которые в этом случае могут возникнуть.

10.28. Одна из составляющих процесса совершения покупок, которая не рассматривалась в этой главе, касается проверки совместимости товаров. Например, если клиент заказывает цифровую фотокамеру, какие аксессуары — аккумуляторы, карты памяти или чехлы — будут совместимы с этой камерой? Разработайте базу знаний, позволяющую принимать решение о совместимости среди множества товаров и предлагаемых замен или дополнительных товаров, если покупатель сделал выбор, который является несовместимым. База знаний должна работать по меньшей мере с одной линией товаров, и ее должно быть достаточно просто расширить для поддержки других линий.

10.29. Полное решение проблемы неточных соответствий описаниям, сформулированным покупателем при совершении покупок, является очень сложным и требует применения полного спектра средств обработки естественного языка и методов выборки информации (см. главу 23). Одним из небольших шагов в этом направлении является предоставление пользователю возможности задавать минимальные и максимальные значения для различных атрибутов. Допустим, что покупатель должен использовать следующую грамматику при описании товара.

$$\begin{aligned} \textit{Description} &\rightarrow \textit{Category}\ [\textit{Connector Modifier}]* \\ \textit{Connector} &\rightarrow \text{"with"} \mid \text{"and"} \mid \text{","} \\ \textit{Modifier} &\rightarrow \textit{Attribute} \mid \textit{Attribute Op Value} \\ \textit{Op} &\rightarrow \text{"="} \mid \text{">"} \mid \text{"<"} \end{aligned}$$

Здесь *Category* обозначает категорию товара, *Attribute* — это некоторая характеристика, такая как "CPU" или "цена", а *Value* — желаемое значение этой характеристики. Поэтому запрос "компьютер с частотой процессора по меньшей мере 2,5 ГГц по цене меньше $500" должен быть выражен так: "компьютер с CPU > 2,5 ГГц и ценой < 500". Реализуйте торгового агента, который принимает описания на этом языке.

10.30. В описании процесса осуществления покупок пока был упущен наиболее важный этап — фактическая покупка товара. Предоставьте формальное логическое описание этапа покупки с использованием исчисления событий. Иначе говоря, определите последовательность событий, которая происходит, когда покупатель приобретает товар по кредитной карточке, а затем в конечном итоге получает выставленный ему счет и ему вручается товар.

ГЛАВА 11

Автоматизированное планирование

В этой главе показано, как агент может воспользоваться информацией о структуре задачи для эффективного создания сложных планов действий.

Планирование последовательности действий — это ключевое требование к интеллектуальному агенту. В разделе 11.1 рассматривается общий **формальный** язык описания данных в развернутом представлении для задач планирования, позволяющий естественно и лаконично представить широкое разнообразие проблемных областей, эффективно масштабировать решения до самых крупных проблем и не требующих разработки специальных эвристик для новых областей применения. В разделе 11.4 этот язык представления расширяется для поддержки иерархических действий, что обеспечивает возможность успешно решать более сложные задачи. Эффективные алгоритмы планирования обсуждаются в разделе 11.2, а эвристики для них рассматриваются в разделе 11.3. В разделе 11.5 полученные результаты распространяются на частично наблюдаемые и недетерминированные проблемные области, а в разделе 11.6 язык представления вновь расширяется с целью охвата задач составления планов и графиков их выполнения при наличии ограничений на ресурсы. Это приближает нас к программам-планировщикам, используемым в реальном мире для разработки планов и расписаний действий космических аппаратов, работы промышленных предприятий и проведения военных операций. В разделе 11.7 обсуждается эффективность всех этих методов.

11.1. Определение классической задачи планирования

►**Классическое планирование** определяется как задача поиска последовательности действий для достижения поставленной цели в дискретной, детерминированной, статичной (изменения в ней происходят только в результате действий агента), полностью наблюдаемой среде. Мы уже рассматривали два подхода к решению этой задачи: агент, решающий задачи в главе 3, и гибридный

пропозициональный логический агент в главе 7. В обоих случаях имеют место два ограничения. Во-первых, оба эти агента требуют использования специальной эвристики для каждого нового домена: эвристической функции оценки для поиска и написания специального программного кода для гибридного агента в мире вампуса. Во-вторых, в обоих случаях требуется явное представление пространства состояний, размеры которого возрастают экспоненциально. Например, в пропозициональной логической модели мира вампуса аксиома перемещения на шаг вперед должна быть повторена для всех четырех возможных ориентаций агента, T временных этапов и n^2 его текущих местоположений.

С учетом этих ограничений исследователи в области планирования разработали синтаксис для ▶**развернутого представления данных** в виде семейства языков, получившего название ▶**PDDL** (*Planning Domain Definition Language* — язык определения проблемной области планирования (Халлаб и др. [846], 1998)), позволяющий представить все $4Tn^2$ действий в виде единственной схемы действий и не требующих использования знаний, специфических для каждой проблемной области. Базовый вариант PDDL позволяет работать с проблемными областями классического планирования, а расширенные варианты могут применяться для непрерывных, частично наблюдаемых, конкурентных и мультиагентных проблемных областей. Синтаксис языков PDDL строится на основе синтаксиса языка Lisp, но здесь мы будем переводить его в такую форму, которая соответствует системе обозначений, принятой в данной книге.

В языке PDDL ▶**состояние** представляется как конъюнкция базовых атомарных флюент. Напомним, что характеристика "базовый" означает отсутствие переменных, термин "флюента" определяет аспект мира, меняющийся со временем, а сочетание "базовый атомарный" означает, что имеется единственный предикат и если есть какие-либо аргументы, то они должны быть константами. Например, выражение $Poor \wedge Unknown$ может представлять состояние агента-неудачника (бедного и безвестного), а выражение $At(Truck_1, Melbourne) \wedge At(Truck_2, Sydney)$ может представлять состояние в задаче доставки пакета. В языке PDDL используется **семантика базы данных**: допущение о замкнутости мира означает, что любые флюенты, которые не упоминаются в базе знаний, являются ложными, а допущение об уникальности имен означает, что объекты $Truck_1$ и $Truck_2$ являются различными.

Следующие флюенты являются *недопустимыми* в некотором состоянии: $At(x, y)$ (поскольку здесь присутствуют переменные), $\neg Poor$ (поскольку это отрицание) и $At(Spouse(Ali), Sydney)$ (поскольку здесь присутствует символ функции *Spouse*). Когда это удобно, конъюнкцию флюент можно рассматривать как *множество* флюент.

▶**Схема действий** представляет семейство базовых действий. Например, вот схема действий для представления перелета самолета из одного места в другое.

$Action(Fly(p, from, to),$
PRECOND: $At(p, from) \wedge Plane(p) \wedge Airport(from) \wedge Airport(to)$
EFFECT: $\neg At(p, from) \wedge At(p, to))$

Эта схема состоит из имени действия, списка всех переменных, используемых в схеме, ▶**предусловия** и ▶**эффекта**. И предусловие, и эффект представлены конъюнкцией литералов (положительных или отрицательных атомарных высказываний). Далее достаточно выбрать нужные константы для конкретизации переменных, и в результате будет получено базовое (не включающее переменных) действие:

$$\begin{aligned}&Action(Fly(P_1, SFO, JFK),\\ &\quad \text{PRECOND}: At(P_1, SFO) \wedge Plane(P_1) \wedge Airport(SFO) \wedge Airport(JFK)\\ &\quad \text{EFFECT}: \neg At(P_1, SFO) \wedge At(P_1, JFK))\end{aligned}$$

(здесь *SFO* — Международный аэропорт Сан-Франциско, а *JFK* — Международный аэропорт имени Джона Кеннеди в Нью-Йорке; используются обозначения Международной ассоциации авиатранспорта IATA для кодировки аэропортов).

Базовое действие будет **применимо** в состоянии *s*, если *s* влечет за собой предусловие, т.е. если каждый положительный литерал предусловия присутствует в *s*, а каждый отрицательный литерал отсутствует в нем.

Результат выполнения применимого действия *a* в состоянии *s* определяется как состояние *s'*, представленное множеством флюент, полученным из начального их множества для состояния *s* после удаления всех флюент, присутствующих в виде отрицательных литералов в выражении, определяющем эффект действия (то, что мы называем ▶**списком удаления** или DEL(*a*)), и добавления флюент, присутствующих в виде положительных литералов в выражении эффекта действия (то, что мы называем ▶**списком добавления** или ADD(*a*)):

$$\text{RESULT}(s, a) = (s - \text{DEL}(a)) \cup \text{ADD}(a). \qquad (11.1)$$

Например, для действия $Fly(P_1, SFO, JFK)$ следует удалить флюенту $At(P_1, SFO)$ и добавить флюенту $At(P_1, JFK)$.

Множество схем действий используется для описания *проблемных областей* планирования. Конкретная *задача* в этой проблемной области определяется добавлением исходного состояния и цели. **Начальное состояние** представляется конъюнкцией базовых флюент (вводится с помощью ключевого слова *Init*, как показано на рис. 11.1). Как и для всех прочих состояний, для него используется допущение о замкнутости мира, а это означает, что любые атомарные высказывания, которые в нем не упомянуты, являются ложными. **Цель** (вводится с помощью ключевого слова *Goal*) представляется аналогично предусловию: в виде конъюнкции литералов (положительных или отрицательных), которые могут содержать переменные. Например, цель $At(C_1, SFO) \wedge \neg At(C_2, SFO) \wedge At(p, SFO)$ ссылается на любое состояние, в котором груз C_1 находится в аэропорту *SFO*, тогда как груз C_2 — нет, и помимо этого, в аэропорту *SFO* есть какой-либо самолет.

11.1.1. Пример проблемной области: грузовые авиаперевозки

На рис. 11.1 приведено описание задачи выполнения грузовых авиаперевозок, включающей погрузку и разгрузку грузов, а также их перевозку самолетами из одного места в другое. Эту задачу можно определить с помощью трех действий: *Load*, *Unload* и *Fly*. Эти действия влияют на значения двух предикатов: предикат $In(c, p)$ означает, что груз c находится в самолете p, а предикат $At(x, a)$ означает, что объект x (самолет или груз) находится в аэропорту a. Обратите внимание: чтобы предикат *At* воспринимался должным образом, необходимо сделать некоторые дополнительные уточнения. Когда самолет летит из одного аэропорта в другой, весь груз внутри самолета перемещается вместе с ним. В логике первого порядка было бы достаточно просто применить соответствующий квантор ко всем объектам, находящимся внутри самолета. Но в языке PDDL нет квантора всеобщности, поэтому нужно найти другое решение. Используемый подход заключается в утверждении, что предикат *At* больше не указывает на местонахождение некоторого груза, если предикат *In* определяет, что этот груз находится в самолете. Этот груз вновь попадет в область воздействия предиката *At* только после его разгрузки в аэропорту прибытия. Поэтому предикат *At* фактически означает "доступен для использования в указанном местонахождении". Ниже приведен план, являющийся решением данной задачи.

$$[Load(C_1, P_1, SFO), Fly(P_1, SFO, JFK), Unload(C_1, P_1, JFK),$$
$$Load(C_2, P_2, JFK), Fly(P_2, JFK, SFO), Unload(C_2, P_2, SFO)]$$

$Init(At(C_1, SFO) \land At(C_2, JFK) \land At(P_1, SFO) \land At(P_2, JFK)$
$\quad \land Cargo(C_1) \land Cargo(C_2) \land Plane(P_1) \land Plane(P_2)$
$\quad \land Airport(JFK) \land Airport(SFO))$
$Goal(At(C_1, JFK) \land At(C_2, SFO))$
$Action(Load(c, p, a),$
$\quad$ PRECOND: $At(c, a) \land At(p, a) \land Cargo(c) \land Plane(p) \land Airport(a)$
$\quad$ EFFECT: $\neg At(c, a) \land In(c, p))$
$Action((Unload(c, p, a),$
$\quad$ PRECOND: $In(c, p) \land At(p, a) \land Cargo(c) \land Plane(p) \land Airport(a)$
$\quad$ EFFECT: $At(c, a) \land \neg In(c, p))$
$Action((Fly(p, from, to),$
$\quad$ PRECOND: $At(p, from) \land Plane(p) \land Airport(from) \land Airport(to)$
$\quad$ EFFECT: $\neg At(p, from) \land At(p, to))$

Рис. 11.1. Описание на языке PDDL задачи планирования грузовых авиаперевозок

11.1.2. Пример проблемной области: задача замены колеса

Рассмотрим задачу замены колеса с проколотой шиной (рис. 11.2). Цель состоит в том, чтобы исправное запасное колесо (*Spare*) было правильно смонтировано на оси автомобиля (*Axle*), исходя из того, что в начальном состоянии на этой оси находится колесо с проколотой шиной (*Flat*), а исправное запасное колесо лежит в багажнике (*Trunk*). Чтобы упростить эту задачу, рассмотрим ее абстрактную версию, в которой не учитываются намертво прикипевшие крепежные гайки или другие подобные усложнения. В задаче можно выделить только четыре действия: извлечение запасного колеса из багажника, снятие колеса с проколотой шиной с оси, установка на ось запасного колеса и оставление автомобиля без присмотра на ночь. Предполагается, что автомобиль находится в районе с неблагоприятной криминогенной обстановкой, поэтому результатом его пребывания ночью на улице будет исчезновение обоих колес. Решение этой задачи можно записать, как

$$[Remove(Flat, Axle), Remove(Spare, Trunk), PutOn(Spare, Axle)].$$

$Init(Tire(Flat) \land Tire(Spare) \land At(Flat, Axle) \land At(Spare, Trunk))$
$Goal(At(Spare, Axle))$
$Action(Remove(obj, loc),$
  PRECOND: $At(obj, loc)$
  EFFECT: $\neg At(obj, loc) \land At(obj, Ground))$
$Action(PutOn(t, Axle),$
  PRECOND: $Tire((t) \land At(t, Ground) \land \neg At(Flat, Axle) \land \neg At(Spare, Axle)$
  EFFECT: $\neg At(t, Ground) \land At(t, Axle))$
$Action(LeaveOvernight,$
  PRECOND:
  EFFECT: $\neg At(Spare, Ground) \land \neg At(Spare, Axle) \land \neg At(Spare, Trunk)$
    $\land \neg At(Flat, Ground) \land \neg At(Flat, Axle) \land \neg At(Flat, Trunk))$

Рис. 11.2. Простая задача замены проколотого колеса запасным

11.1.3. Пример проблемной области: мир блоков

Одной из наиболее широко известных проблемных областей задач планирования является так называемый ▶**мир блоков**. Эта проблемная область состоит из множества блоков кубической формы, находящихся на столе.[1] Блоки можно укладывать в столбик, но непосредственно на верхней поверхности одного блока может быть размещен только один блок. Робот может брать манипулятором блок и

[1] Мир блоков, используемый в исследованиях задач планирования, гораздо проще по сравнению с версией SHRDLU, представленной в разделе 1.3.2.

перемещать его в другую позицию — либо на стол, либо на верхнюю поверхность другого блока. Манипулятор может поднять только один блок, поэтому невозможно взять блок, на котором стоит еще один блок. Типичная цель может состоять в том, чтобы поставить блок *A* на *B*, а блок *B* — на *C* (рис. 11.3).



Исходное состояние Целевое состояние

Рис. 11.3. Пример задачи в проблемной области мира блоков. (Определение этой задачи представлено на рис. 11.4.)

Будем использовать предикат $On(b, x)$ для указания на то, что блок b находится на блоке x, где x — либо другой блок, либо стол. Перемещение блока b с верхней поверхности блока x на верхнюю поверхность блока y будет представлено функцией $Move(b, x, y)$. Напомним, что одним из предусловий перемещения блока b является то, что на нем не стоит какой-то другой блок. В логике первого порядка это требование может быть представлено с помощью выражения $\neg\exists\, x\ \ On(x, b)$ или — альтернативный вариант — выражения $\forall\, x\ \ \neg On(x, b)$. Однако базовый вариант языка PDDL не допускает использования кванторов, поэтому потребуется ввести особый предикат $Clear(x)$, имеющий значение *true*, если на блоке x ничего нет (его верхняя поверхность свободна). Полное определение этой задачи приведено на рис. 11.4.

Действие *Move* позволяет переместить блок b с блока x на блок y, если свободны верхние поверхности и блока b, и блока y. После выполнения этого действия верхняя поверхность блока x будет свободна, а блока y — нет. Простейшим вариантом схемы действия *Move* будет следующая:

Action($Move(b, x, y)$,
PRECOND: $On(b, x) \wedge Clear(b) \wedge Clear(y)$,
EFFECT: $On(b, y) \wedge Clear(x) \wedge \neg On(b, x) \wedge \neg Clear(y)$).

К сожалению, эта схема не определяет предикат *Clear* должным образом, если блок x или блок y находится на столе. Если x имеет значение *Table*, то для этого действия эффектом будет $Clear(Table)$, но поверхность стола вовсе не должна стать полностью свободной, а если значение *Table* имеет переменная y, то действие

будет иметь предусловие *Clear*(*Table*), но поверхность всего стола не обязана быть свободной для того, чтобы на нее можно было поместить блок. Для исправления этой ситуации необходимо внести два изменения. Во-первых, введем еще одно действие перемещения блока b с блока x на стол:

$$\begin{aligned}&Action(MoveToTable(b, x),\\ &\quad \text{PRECOND: } On(b, x) \wedge Clear(b),\\ &\quad \text{EFFECT: } On(b, Table) \wedge Clear(x) \wedge \neg On(b, x)).\end{aligned}$$

$$\begin{aligned}&Init(On(A, Table) \wedge On(B, Table) \wedge On(C, A)\\ &\quad \wedge Block(A) \wedge Block(B) \wedge Block(C) \wedge Clear(B) \wedge Clear(C) \wedge Clear(Table))\\ &Goal(On(A, B) \wedge On(B, C))\\ &Action(Move(b, x, y),\\ &\quad \text{PRECOND: } On(b, x) \wedge Clear(b) \wedge Clear(y) \wedge Block(b) \wedge Block(y) \wedge\\ &\qquad (b \neq x) \wedge (b \neq y) \wedge (x \neq y),\\ &\quad \text{EFFECT: } On(b, y) \wedge Clear(x) \wedge \neg On(b, x) \wedge \neg Clear(y))\\ &Action(MoveToTable(b, x),\\ &\quad \text{PRECOND: } On(b, x) \wedge Clear(b) \wedge Block(b) \wedge Block(x),\\ &\quad \text{EFFECT: } On(b, Table) \wedge Clear(x) \wedge \neg On(b, x))\end{aligned}$$

Рис. 11.4. Определение задачи планирования в мире блоков: построение столбика из трех блоков. Возможным решением является последовательность действий [*MoveToTable*(*C*, *A*), *Move*(*B*, *Table*, *C*), *Move*(*A*, *Table*, *B*)]

Во-вторых, примем интерпретацию предиката *Clear*(x) как означающую, что "на x имеется достаточно свободного места, чтобы поместился один блок". При этой интерпретации предикат *Clear*(*Table*) всегда будет истинным. Единственная проблема состоит в том, что ничто не будет препятствовать планировщику использовать действие *Move*(b, x, *Table*) вместо действия *MoveToTable*(b, x). Можно смириться с этой проблемой (она приводит к созданию пространства поиска с размерами больше необходимого, но не становится причиной получения неправильных ответов) или ввести предикат *Block* и добавить в предусловие действия *Move* выражение *Block*(b) $\wedge$ *Block*(y), как показано на рис. 11.4.

11.2. Алгоритмы классического планирования

Рассмотренное выше описание задачи планирования предлагает очевидный способ поиска в пространстве состояний в направлении от начального состояния к искомой цели. Декларативное представление схем действия дает также важное преимущество — возможность ведения поиска в обратном направлении, от цели к начальному состоянию (варианты ведения поиска в прямом и обратном направлениях сравниваются на рис. 11.5). Третья возможность заключается в переводе

описания задачи в множество логических высказываний, к которым затем можно будет применить алгоритм логического вывода, чтобы найти решение.

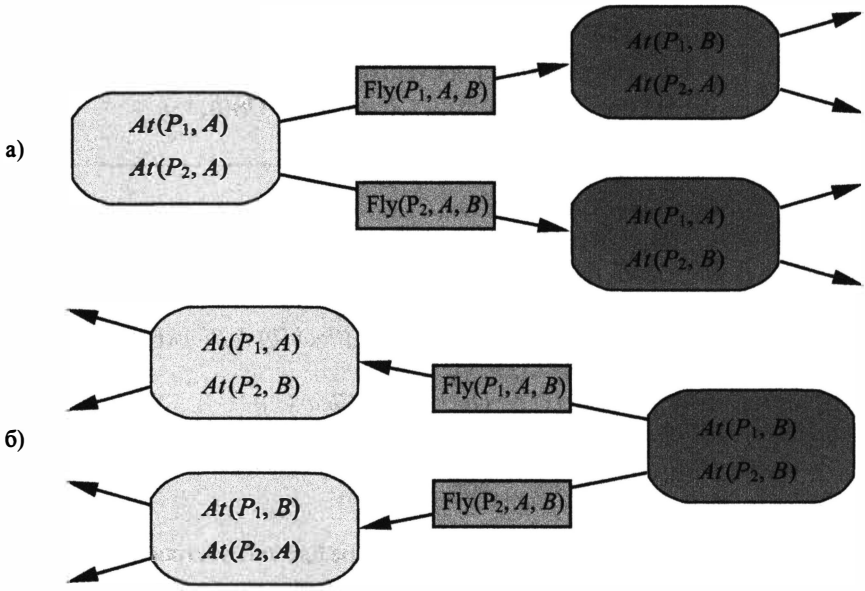


Рис. 11.5. Два подхода к организации поиска плана. **а)** Прямой (прогрессивный) поиск в пространстве базовых состояний, начинающийся с начального состояния, в котором действия задачи используются для прямого поиска члена группы целевых состояний. **б)** Обратный (регрессивный) поиск по описаниям состояний, начинающийся с цели и использующий инверсии действий для обратного поиска начального состояния

11.2.1. Прямой поиск плана в пространстве состояний

Можно решать задачи планирования, воспользовавшись любым из эвристических алгоритмов поиска, обсуждавшихся в главах 3 и 4. Состояниями в этом поисковом пространстве состояний являются базовые состояния, в которых каждая флюента является либо истинной, либо ложной. Цель — это состояние, в котором присутствуют все положительные флюенты целевого состояния задачи, а все его отрицательные флюенты отсутствуют. Применимые действия в некотором состоянии *Actions*(*s*) — это базовые конкретизации схем действий, т.е. действия, в которых все переменные заменены константами.

Чтобы выявить применимые действия, текущее состояние унифицируется с предусловиями каждой схемы действий. Для каждой унификации, которая дает удовлетворительные результаты при подстановке, эта подстановка применяется к

схеме действия для получения базового действия без переменных. (Схемы действий требуют, чтобы любая переменная, упоминаемая в эффекте, присутствовала и в предпосылке, а это гарантирует нам, что при подстановке никаких переменных не останется.)

Каждая схема может унифицироваться несколькими способами. В примере с проколотой шиной (раздел 11.1.2), действие *Remove* имеет предусловие *At*(*obj*, *loc*), которое можно сопоставить с начальным состоянием двумя способами, что приводит к двум подстановкам: {*obj*/*Flat*, *loc*/*Axle*} и {*obj*/*Spare*, *loc*/*Trunk*}, — при применении этих подстановок получаем два базовых действия. Если в предусловии действия есть несколько литералов, то каждый из них потенциально может быть сопоставлен с текущим состоянием несколькими способами.

На первый взгляд может показаться, что для многих задач пространство состояний может оказаться слишком большим. Рассмотрим задачу воздушных грузовых перевозок с 10 аэропортами, где в каждом аэропорту исходно имеется 5 самолетов и 20 единиц груза. Цель состоит в том, чтобы переместить все грузы из аэропорта *A* в аэропорт *B*. Существует решение этой задачи, включающее 41 этап: погрузить 20 единиц груза в один из самолетов в аэропорту *A*, выполнить на нем перелет в аэропорт *B* и выгрузить эти 20 единиц груза.

Однако формальный поиск этого очевидного решения может оказаться затруднительным, поскольку средний коэффициент ветвления является огромным: каждый из этих 50 самолетов может полететь в 9 других аэропортов, а каждая из 200 единиц груза может быть либо выгружена (если она загружена), либо загружена в любой самолет в том аэропорту, где она находится (если она выгружена). Поэтому в любом состоянии возможно как минимум 450 действий (когда все единицы груза находятся в аэропорту, где нет ни одного самолета) и как максимум 10 450 действий (когда все грузы и все самолеты находятся в одном аэропорту). В среднем можно принять, что для каждого состояния существует около 2000 возможных действий, поэтому граф поиска до глубины в 41 этап будет иметь примерно 2000^{41} узлов.

Очевидно, что без точной эвристики прямой поиск решения даже для этого относительно небольшого экземпляра задачи является делом, абсолютно безнадежным. Хотя многие реальные приложения по планированию полагались исключительно на эвристики, специфические для своей проблемной области, выяснилось (как будет показано в разделе 11.3), что мощные эвристики, независимые от проблемной области задачи, могут быть выведены автоматически; а это делает использование прямого поиска вполне допустимым.

11.2.2. Обратный поиск плана в пространстве состояний

При обратном поиске (его также называют ▶**регрессивным поиском**) отправной точкой является целевое состояние и действия применяются в обратном направлении, пока не будет найдена последовательность этапов, приводящая к начальному состоянию. На каждом этапе рассматриваются ▶**релевантные**

действия (в отличие от прямого поиска, где рассматриваются **применимые** действия), что приводит к значительному уменьшению коэффициента ветвления, особенно в проблемных областях с множеством возможных действий.

Релевантными являются такие действия, эффект которых унифицирует один из литералов цели и при этом не приводит к отрицанию любого из ее элементов. Например, при цели $\neg Poor \wedge Famous$ действие с единственным эффектом $Famous$ будет релевантным, тогда как действие с эффектом $Poor \wedge Famous$ не будет рассматриваться как релевантное: даже хотя это действие можно использовать в некоторой иной точке плана (чтобы получить литерал $Famous$), его нельзя применить в данной точке плана, поскольку в этом случае в конечном состоянии появится литерал $Poor$.

Что понимается под применением действия в обратном направлении? При заданных цели g и действии a ▶ **регрессия** от g через a дает нам состояние g', описание положительных и отрицательных литералов которого будет иметь такой вид:

$$\text{POS}(g') = (\text{POS}(g) - \text{ADD}(a)) \cup \text{POS}(Precond(a)),$$
$$\text{NEG}(g') = (\text{NEG}(g) - \text{DEL}(a)) \cup \text{NEG}(Precond(a)).$$

То есть предусловия должны были быть выполнены ранее, иначе действие не могло бы быть выполнено, но положительные/отрицательные литералы, которые были добавлены/удалены действием, не должны были быть истинными ранее.

Эти уравнения, безусловно, справедливы для базовых литералов, но при наличии в g и a переменных потребуется некоторая осторожность. Например, предположим, что целью является доставка определенной единицы груза в SFO: $At(C_2, SFO)$. Схема действия *Unload* имеет эффект $At(c, a)$. Если унифицировать его с целью, то получим подстановку $\{c/C_2, a/SFO\}$. Применение этой подстановки к схеме дает новую схему, отражающую идею использования любого самолета, находящегося в SFO:

$Action(Unload(C_2, p', SFO),$
PRECOND: $In(C_2, p') \wedge At(p', SFO) \wedge Cargo(C_2) \wedge Plane(p') \wedge Airport(SFO),$
EFFECT: $At(C_2, SFO) \wedge \neg In(C_2, p'))$.

Здесь переменная p заменена новой переменной с именем p' — это результат **стандартизации отличия** имен переменных, позволяющей исключить возможность конфликта имен между различными переменными, которым случайно было присвоено одинаковое имя (см. раздел 9.2.1). Определение достигнутого применением регрессии состояния дает новую цель:

$$g' = In(C_2, p') \wedge At(p', SFO) \wedge Cargo(C_2) \wedge Plane(p') \wedge Airport(SFO).$$

В качестве другого примера рассмотрим цель приобрести книгу с конкретным номером ISBN: $Own(9780134610993)$. При наличии триллиона 13-значных номеров ISBN и единственной схемы действия

$$A = Action(Buy(i), \text{PRECOND: } ISBN(i), \text{EFFECT: } Own(i))$$

прямой поиск без использования эвристики начал бы работу с перечисления 10 млрд. базовых действий *Buy*. Но в случае обратного поиска можно унифицировать цель *Own*(9780134610993) с эффектом *Own*(i') и получить подстановку $\theta = \{i'/9780134610993\}$. Теперь можно выполнить регрессию цели действием *Subst*(θ, A) и получить определение состояния-преемника в виде *ISBN*(9780134610993). Это элемент начального состояния, так что найдено полное решение, достигнутое после рассмотрения единственного действия, а не триллиона.

Более формально предположим, что имеется определение цели g, содержащее литерал цели g_i и схему действия A. Если схема A включает литерал эффекта e'_j, где $Unify(g_i, e'_j) = \theta$ и где определено $A' = \text{SUBST}(\theta, A)$, и если в эффекте A' нет отрицания литерала, присутствующего в g, то A' является релевантным действием в отношении цели g.

Для большинства проблемных областей при обратном поиске коэффициент ветвления будет ниже, чем при прямом поиске. Однако тот факт, что при обратном поиске используются состояния с переменными, а не с базовыми состояниями, усложняет создание хорошей эвристики. Это основная причина, по которой в большинстве современных систем предпочтение отдается прямому поиску.

11.2.3. Планирование как булева выполнимость

В разделе 7.7.4 было показано, как некоторые хитроумные переписывания аксиом могут превратить задачу мира вампуса в задачу определения выполнимости высказывания в логике высказываний, которую затем можно будет решить с помощью специальной программы: эффективного решателя выполнимости SAT. Решатели SAT, такие как SATPLAN, работают посредством перевода описания задачи на языке PDDL в пропозициональную форму. Подобный перевод включает в себя несколько этапов.

- *Пропозиционализация действий.* Для каждой схемы действия формируются базовые высказывания посредством подстановки констант для каждой из переменных. В результате вместо единственной схемы действия *Unload*(c, p, a) будут получены отдельные высказывания для этого действия, выполняемого с каждой комбинацией груза, самолета и аэропорта (здесь они записываются с помощью индексов), и для каждого момента времени (здесь они записываются в виде верхних индексов).
- *Добавление аксиом исключения действия.* Аксиомы исключения действия утверждают, что два некоторых действия не могут происходить в одно и то же время, например

$$\neg(FlyP_1SFOJFK^1 \wedge FlyP_1SFOBUH^1).$$

- *Добавление аксиом предусловий.* Для каждого базового действия A^t добавляется аксиома $A^t \Rightarrow \text{PRE}(A)^t$, т.е. если действие выполняется в момент времени t, то должны быть выполнены заданные предусловия, например

$$FlyP_1SFOJFK^1 \Rightarrow At(P_1, SFO) \wedge Plane(P_1) \wedge Airport(SFO) \wedge Airport(JFK).$$

- *Определение начального состояния.* Вводится утверждение F^0 для каждой флюенты F, присутствующей в исходном состоянии, и утверждение $\neg F^0$ для каждой флюенты, не упоминаемой в начальном состоянии.
- *Пропозиционализация цели.* Цель формулируется как дизъюнкция всех ее базовых экземпляров, в которых переменные заменяются константами. Так, цель "Поместить блок A на другой блок", $On(A, x) \wedge Block(x)$ в мире блоков с объектами A, B и C, будет заменена высказыванием

  $$(On(A, A) \wedge Block(A)) \vee (On(A, B) \wedge Block(B)) \vee (On(A, C) \wedge Block(C)).$$

- *Добавление аксиом определения преемника.* Для каждой флюенты F добавляется аксиома вида

  $$F^{t+1} \Leftrightarrow ActionCausesF^t \vee (F^t \wedge \neg ActionCausesNotF^t),$$

  где $ActionCausesF$ означает дизъюнкцию всех базовых действий, которые добавляют флюенту F, а $ActionCausesNotF$ означает дизъюнкцию всех базовых действий, которые удаляют F.

Полученный в результате перевод, как правило, намного больше по размеру, чем оригинальное определение задачи на языке PDDL, однако высокой эффективности современных решателей SAT часто более чем достаточно, чтобы компенсировать это усложнение.

11.2.4. Другие классические подходы к планированию

Три подхода, обсуждавшиеся выше, — не единственные из тех, которые были опробованы в 50-летней истории автоматизированного планирования. Некоторые другие кратко описываются в этом разделе.

В подходе, получившем название Graphplan, используется специализированная структура данных, ►**граф планирования**, позволяющая наглядно представить ограничения в отношении того, как действия связаны с их предпосылками и эффектами и какие вещи являются взаимно исключающими.

►**Исчисление ситуаций** представляет собой метод описания задач планирования в логике первого порядка. В нем используются аксиомы определения преемника, как и в случае использования решателя SATPLAN, но логика первого порядка обеспечивает большую гибкость и использует более сжатые аксиомы. В целом этот подход внес значительный вклад в наше теоретическое понимание планирования, но не оказал большого влияния на приложения, используемые на практике, — возможно, потому, что программы поиска доказательств для логики первого порядка оказались не столь совершенными, как программы определения выполнимости для логики высказываний.

Ограниченную задачу планирования (т.е. задачу о нахождении плана длиной k) можно закодировать как **задачу удовлетворения ограничений** (УО). Кодирование такой задачи схоже с кодированием задач SAT (раздел 11.2.3), но с одним важным упрощением: в каждый момент времени нам потребуется только одна переменная, $Action^t$, областью определения которой является множество возможных действий. В этом случае уже не требуется иметь по одной переменной на каждое действие и нет необходимости в определении аксиом исключения действия.

Все упомянутые выше подходы приводят к созданию *полностью упорядоченных* планов, состоящих из строго линейной последовательности действий. Но если в задаче авиаперевозок потребуется погрузить 30 единиц груза в один самолет и 50 единиц груза — в другой, то будет практически бессмысленно устанавливать некоторую линейную упорядоченность для действий погрузки 80 единиц груза.

Альтернативный подход называется ▶**планированием с частичным упорядочиванием**. В этом случае план представляет собой граф, а не линейную последовательность: каждое действие представлено в этом графе узлом, а каждое предусловие этого действия представляет собой ребро, идущее от другого действия (или из начального состояния), указывая, что предшествующее действие устанавливает эту предпосылку. В результате мы получаем план с частичным упорядочиванием, указывающий, что действия *Remove*(*Spare*, *Trunk*) и *Remove*(*Flat*, *Axle*) должны быть выполнены до действия *PutOn*(*Spare*, *Axle*), но без прямого указания на то, какое из двух действий *Remove* следует выполнять первым. В результате поиск ведется в пространстве планов, а не состояний мира, посредством вставки действий для удовлетворения условий.

В 1980- и 1990-х годах планирование с частичным упорядочиванием рассматривалось как наилучший способ решения задач планирования с помощью независимых подзадач. К 2000 году для планировщиков, работающих по методу прямого поиска, были разработаны отличные эвристики, позволявшие им эффективно обнаруживать независимые подзадачи, для чего, собственно, ранее и был разработан метод планирования с частичным упорядочиванием. Более того, новый решатель SATPLAN позволил получить преимущество и от закона Мура: пропозиционализация, которая была безнадежно велика в 1980 году, теперь выглядела крошечной, — просто потому, что у компьютеров теперь было в 10 000 раз больше памяти. В результате планировщики с частичным упорядочиванием уже не могли конкурировать с полностью автоматизированными решателями задач классического планирования.

Тем не менее метод планирования с частичным упорядочиванием остается важной частью в этой области. Для некоторых конкретных задач, таких как планирование операций, лучшим выбором по-прежнему остается планирование с частичным упорядочиванием с использованием эвристик, специфических для конкретных проблемных областей. Многие из подобных систем используют библиотеки планов высокого уровня, как это описывается в разделе 11.4.

Планирование с частичным упорядочиванием также часто используется в проблемных областях, в которых людям важно понимать разработанные планы. Например, оперативные планы для космических кораблей и марсоходов создаются специализированными планировщиками с частичным упорядочиванием, а затем проверяются операторами-людьми, прежде чем будут загружены в действующие космические аппараты для выполнения. Такой подход с уточнением планов упрощает людям понимание того, как действуют алгоритмы планирования, и позволяет убедиться в правильности разработанных ими планов до того, как они будут выполнены.

11.3. Эвристики для задач планирования

Ни прямой, ни обратный поиск не будет эффективен без хорошей эвристической функции. Вспомним из главы 3, что эвристическая функция $h(s)$ оценивает расстояние от состояния s до целевого состояния и что, если мы можем вывести **допустимую** эвристику для этого расстояния — такую, которая не переоценивает, — появляется возможность использовать поиск A* для нахождения оптимального решения.

По определению нет никакого способа проанализировать атомарное состояние, и поэтому аналитику (обычно человеку) потребуется некоторая изобретательность, чтобы определить хорошую, специфическую для проблемной области эвристику для задач поиска с атомарными состояниями. Однако в планировании используется *развернутое* представление состояний и действий, что позволяет определить хорошую эвристику, независимую от проблемной области.

Напомним, что допустимую эвристику можно получить за счет определения **ослабленной задачи**, которую будет проще решить, — точную стоимость решения этой ослабленной задачи можно использовать как эвристику для исходной задачи. Задачу поиска можно сформулировать в виде графа, узлы которого представляют состояния, а ребра — действия. Задача состоит в том, чтобы найти путь, соединяющий начальное состояние с целевым состоянием. Существует два основных способа ослабления этой задачи, чтобы сделать ее решение проще: посредством добавлении в граф новых ребер, что существенно упрощает поиск пути, либо посредством группирования нескольких узлов в одно целое для получения абстрактного представления пространства состояний с меньшим числом состояний, что, опять же, упрощает поиск.

Сначала рассмотрим эвристику, в которой к графу добавляются ребра. Возможно, самым простым случаем является ▶**эвристика игнорирования предусловий**, — в этом случае для действий просто отбрасываются все предусловия. Каждое действие становится применимым в любом состоянии, и любая отдельная флюента цели может быть достигнута за один этап (если имеются какие-либо применимые действия; если их нет, задача неразрешима). Из этого почти безусловно следует, что количество этапов, необходимых для решения ослабленной задачи, равно

количеству невыполненных целей — почти, но не совсем, поскольку, во-первых, некоторое действие может достигать сразу нескольких целей, и во-вторых, некоторые действия могут отменять эффекты, достигнутые другими действиями.

Для многих задач можно получить более точную эвристику, принимая во внимание действия, достигающие нескольких целей, и игнорируя возможность отмены уже достигнутых результатов. Сначала дополнительно ослабим действия посредством удаления всех предусловий и всех эффектов, кроме тех, которые представлены литералами в цели. Далее подсчитываем минимальное количество требуемых действий таким образом, чтобы объединение положительных результатов этих действий позволяло выполнить поставленную цель. Это пример ▶**задачи о покрытии множества**. Но здесь есть один довольно неприятный момент: задача о покрытии множества является NP-трудной. К счастью, простой алгоритм жадного поиска гарантированно возвращает покрытие множества, размер которого находится в пределах фактора $\log n$ от истинного минимального покрытия, где n является количеством литералов в цели. К сожалению, применение алгоритма жадного поиска исключает гарантию допустимости.

Также можно игнорировать только некоторые выбранные предусловия действий. Обратимся к головоломке ***Восемь*** (или ***Пятнадцать***) из раздела 3.2. Можно было бы закодировать ее как задачу планирования, включающую плитки с одной схемой *Slide*.

$$\begin{aligned}&Action(Slide(t, s_1, s_2),\\ &\quad \text{PRECOND: } On(t, s_1) \wedge Tile(t) \wedge Blank(s_2) \wedge Adjacent(s_1, s_2)\\ &\quad \text{EFFECT: } On(t, s_2) \wedge Blank(s_1) \wedge \neg On(t, s_1) \wedge \neg Blank(s_2))\end{aligned}$$

Как было показано в разделе 3.6, если для этой головоломки удалить предусловия $Blank(s_2) \wedge Adjacent(s_1, s_2)$, то любая плитка за одно действие может быть перемещена в любое место и мы получаем эвристику "Плитки не на своих местах". Если же удалить только предусловие $Blank(s_2)$, то получим эвристику "Манхэттенское расстояние". Легко увидеть, как эти эвристики можно автоматически вывести из описания схемы действия. Легкость манипулирования схемами действия является большим преимуществом развернутого представления задач планирования в сравнении с атомарным представлением поисковых задач.

Другой возможностью является ▶**эвристика с пустым списком удаления**. На минуту предположим, что все цели и предпосылки содержат только положительные литералы.[2] Наша цель — создать ослабленную версию исходной задачи, которую будет легче решить и в которой длина решения будет служить хорошей эвристикой. Этого можно достичь, очистив списки удаления во всех действиях (т.е. удалив все отрицательные литералы из эффектов). В результате появляется возможность неуклонно продвигаться в направлении к цели, — никакие действия

[2] Описание многих задач соответствует этому соглашению. В задачах, в которых это невозможно, замените все отрицательные литералы $\neg P$ в цели или предусловии новым положительным литералом P'.

уже не будут отменять прогресс, достигнутый с помощью другого действия. Как оказалось, поиск оптимального решения этой ослабленной задачи сам по себе является NP-трудной задачей, но хорошее приближенное решение может быть найдено за полиномиальное время методом поиска восхождением к вершине.

На рис. 11.6 представлены графы пространства состояний для двух задач планирования при применении эвристики с пустым списком удаления. Точки представляют состояния, а ребра — действия, при этом высота каждой точки относительно нижней плоскости представляет значение ее эвристической оценки. Состояния на нижней плоскости являются решениями. Для обеих этих задач существует множество путей достижения цели. В их графах нет тупиков, поэтому поиск с возвратами не требуется и решение легко может быть найдено простым поиском восхождением к вершине (хотя это найденное решение не обязательно будет оптимальным).

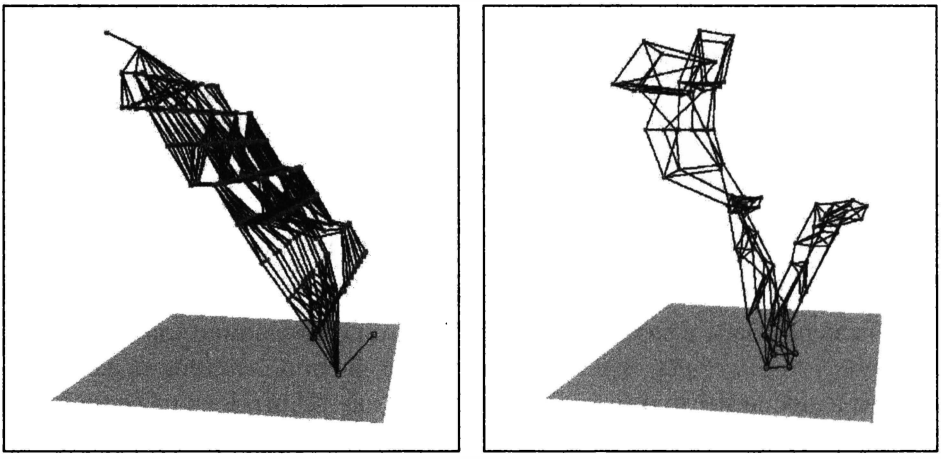


Рис. 11.6. Пространства состояний двух задач планирования при применении эвристики с пустым списком удаления. Высота узла относительно нижней плоскости представляет значение эвристической оценки этого состояния, а состояния непосредственно на нижней плоскости представляют цели. Локальных минимумов здесь нет, поэтому поиск цели не вызовет затруднений (Гофман [1044], 2005)

11.3.1. Отсечение, независимое от проблемной области

Развернутое представление состояний делает очевидным то, что в действительности многие состояния являются лишь вариантами других состояний. Например, предположим, что на столе имеется дюжина блоков и цель состоит в том, чтобы сделать блок A верхним в столбике из трех блоков. Первый этап решения заключается в установке некоторого блока x поверх блока y (где x, y и A являются разными

блоками). Теперь достаточно поместить блок A поверх блока x — и цель будет достигнута. В этой задаче существует 11 вариантов выбора блока x, а после выбора x остается еще 10 вариантов выбора блока y, — таким образом, имеется 110 состояний, подлежащих рассмотрению. Но все эти состояния являются симметричными: выбор того или иного блока не имеет никакого значения для достижения цели, и следовательно, планировщику достаточно будет рассмотреть только один из этих вариантов. Это процесс ►**сокращения симметрии**: в поисковом дереве отсекаются все симметричные ветви за исключением одной, которая и анализируется в дальнейшем поиске. Для многих проблемных областей это меняет ситуацию от неразрешимой до эффективно решаемой.

Другая возможность заключается в предварительном отсечении ветвей без рассмотрения, принимая риск отсечения оптимального решения ради того, чтобы сфокусировать поиск только на перспективных ветвях дерева поиска. Понятие ►**предпочтительного действия** можно определить следующим образом. Сначала определяем ослабленную версию задачи и решаем ее, чтобы получить ►**ослабленный план**. Теперь предпочтительным действием может быть либо этап ослабленного плана, либо действие достижения некоторого предусловия ослабленного плана.

Иногда можно эффективно решить задачу, определив, что отрицательные взаимодействия могут быть исключены. Говорят, что у задачи есть ►**упорядочиваемые подцели**, если существует такой порядок подцелей, что планировщик может достигать их в этом порядке без необходимости отмены ранее достигнутых подцелей. Например, в мире блоков, если цель состоит в том, чтобы построить столбик из трех блоков (например, блок A находится на блоке B, который, в свою очередь, находится на блоке C, который, в свою очередь, находится на столе *Table*, как показано на рис. 11.3), то подцели можно упорядочить в направлении снизу вверх. Если первым действием поместить C на *Table*, то это действие уже не потребуется отменять в процессе достижения других подцелей. В мире блоков планировщик, использующий подход "снизу вверх", сможет решить любую задачу, не отменяя уже выполненных действий (хотя он не всегда сможет найти кратчайший план). Возьмем другой пример. Предположим, есть помещение с n выключателями освещения, каждый из которых управляет отдельным источником света, а цель состоит в том, чтобы включить их все. В этом случае не имеет смысла рассматривать разные перестановки в последовательности включения источников света, — можно с уверенностью ограничиться планами, в которых выключатели включаются, скажем, в восходящем порядке.

Для планировщика из ПО "Remote Agent" космического зонда НАСА "Deep Space One" было принято, что все сообщения, связанные с командованием космическим кораблем, являются упорядочиваемыми. Возможно, это не покажется таким уж удивительным, если принять во внимание, что его создатели специально сконструировали этот космический зонд так, чтобы управлять им было как можно проще (с учетом других ограничений). Используя преимущества упорядочивания команд целеуказания, планировщик в ПО "Remote Agent" мог отсекать бо́льшую

часть дерева поиска. А это означало, что ПО на борту действовало достаточно быстро, чтобы космическим зондом можно было управлять в реальном времени, что прежде считалось невозможным.

11.3.2. Абстрактное представление состояний в планировании

Ослабленная задача предоставляет возможность упростить задачу планирования только в отношении вычисления значений эвристической функции. Многие задачи планирования включают 10^{100} и более состояний, и ослабление *действий* не позволяет уменьшить количество состояний, а это означает, что вычисление эвристики все еще будет оставаться дорогостоящим. По этой причине теперь мы обратимся к ослаблениям, которые уменьшают число *состояний* за счет формирования ►**абстракции состояний** — отображения в абстрактное представление типа "многие к одному" состояний в базовом представлении задачи.

Простейшая форма абстракции состояний заключается в игнорировании некоторых флюент. Например, рассмотрим задачу грузовых авиаперевозок с 10 аэропортами, 50 самолетами и 200 единицами груза. Каждый самолет может находиться в одном из 10 аэропортов, и каждая единица груза может либо находиться в одном из самолетов, либо быть выгруженной в одном из аэропортов. Таким образом, существует $10^{50} \times (50+10)^{200} \approx 10^{405}$ состояний. Теперь рассмотрим конкретную задачу в этой проблемной области, в которой ситуация складывается так, что все единицы груза находятся только в 5 из общего числа аэропортов и все единицы груза в одном аэропорту необходимо доставить в один и тот же пункт назначения. В такой ситуации полезную абстракцию для данной задачи можно будет получить, отбросив все флюенты *At*, кроме тех, которые включают один самолет и одну единицу груза в каждом из указанных 5 аэропортов. В этом случае в задаче останется лишь $10^5 \times (5+10)^5 \approx 10^{11}$ состояний. Решение в подобном абстрактном пространстве состояний будет короче, чем решение в исходном пространстве (и, следовательно, будет представлять собой допустимую эвристику), и это абстрактное решение легко будет расширить до решения исходной задачи (посредством добавления дополнительных действий *Load* и *Unload*).

Ключевой идеей в определении эвристик является ►**декомпозиция**: деление задачи на части, каждая из которых решается независимо от других, с последующим объединением частей. Предположение о ►**независимости подцелей** — это допущение, что затраты на решение конъюнкции подцелей аппроксимируются суммой затрат на решение каждой подцели *независимо* от других. Предположение о независимости подцелей может быть оптимистичным или пессимистичным. Оно оптимистично, когда существуют отрицательные взаимодействия между подпланами для каждой подцели — например, когда действие в одном подплане удаляет цель, достигнутую другим подпланом. Оно пессимистично и, следовательно, недопустимо, когда подпланы содержат избыточные действия, например два действия, которые можно было бы заменить одним действием в объединенном плане.

Предположим, что цель представлена множеством флюент G, которое мы разбиваем на непересекающиеся подмножества $G_1, \ldots, G_n$. Далее выполняется поиск оптимальных планов $P_1, \ldots, P_n$, достигающих соответствующих подцелей. Как оценить стоимость плана достижения общей цели G? Можно рассматривать каждую стоимость $\text{COST}(P_i)$ как эвристическую оценку, и нам известно, что если объединить эти оценки, принимая их максимальное значение, то всегда будет получена допустимая эвристика. Поэтому оценка $\max_i \text{COST}(P_i)$ является допустимой, а иногда и точно правильной: может случиться, что план P_1 по счастливой случайности достигает всех подцелей G_i. Но, как правило, оценка оказывается слишком низкой. Можно ли вместо этого просто просуммировать стоимости? Для многих задач это даст разумную оценку, но недопустимую. Наилучший вариант, когда G_i и G_j независимы, — в том смысле, что планы для одного не могут снизить стоимость планов для других. В этом случае оценка $\text{COST}(P_i) + \text{COST}(P_j)$ является допустимой и более точной, чем максимальная оценка.

Очевидно, что метод сокращения пространства поиска путем формирования абстракций имеет большой потенциал. Вся сложность заключается в правильном выборе абстракций и использовании их таким способом, который делает общую стоимость — суммарную стоимость определения абстракции, выполнения абстрактного поиска и обратного отображения абстракции на исходную задачу — меньше стоимости решения исходной задачи. Здесь может оказаться полезным применение метода баз данных с шаблонами, обсуждавшегося в разделе 3.6.3, поскольку издержки на создание самой базы данных с шаблонами могут быть амортизированы за счет решения многих экземпляров задачи.

Примером системы, в которой используется эффективная эвристика, является система FF или FASTFORWARD (Хоффманн [1044], 2005). В ней применен прямой поиск в пространстве состояний с использованием эвристики с пустым списком удаления и определением эвристических оценок с помощью графа планирования. Затем для поиска решения с этой эвристикой в системе FF используется поиск восхождением к вершине (модифицированный для отслеживания плана). В системе FF используется нестандартный алгоритм поиска восхождением к вершине: он избегает локальных максимумов посредством поиска в ширину от текущего состояния, пока не будет найдено лучшее. Если это не удается, система FF переключается на жадный поиск по первому наилучшему совпадению.

11.4. Иерархическое планирование

Все методы решения задач и планирования, обсуждавшиеся в предыдущих главах, работают с фиксированным множеством атомарных действий. Действия могут быть связаны друг с другом, и современные алгоритмы могут генерировать решения, содержащие тысячи действий. Все это хорошо, когда мы планируем отпуск и действия находятся на уровне "перелет из Сан-Франциско в Гонолулу", но в планах на уровне управления движениями типа "согни левый коленный сустав

на угол 5 градусов" потребуется объединять уже не тысячи, а миллионы или миллиарды действий.

Преодоление этой пропасти требует планирования на более высоких уровнях абстракции. План высокого уровня для отдыха на Гавайях может быть следующим: "Проезд в аэропорт Сан-Франциско; перелет рейсом НА 11 в Гонолулу; отдых две недели; перелет рейсом НА 12 обратно в Сан-Франциско; проезд домой". Имея такой план, действие "Проезд в аэропорт Сан-Франциско" можно само по себе рассматривать как задачу планирования со следующим решением: "Выбрать службу автоперевозок пассажиров; заказать машину; проехать в аэропорт". Каждое из этих действий, в свою очередь, может быть дальше разложено на составляющие, пока не будет достигнут максимально низкий уровень действий, например нажатие отдельной кнопки при наборе телефонного номера службы перевозок.

В этом примере планирование и действие чередуются, например пассажир такси может отложить проблему планирования прохода от обочины дороги до входных дверей, пока его не высадят в нужном месте. Таким образом, это конкретное действие остается на абстрактном уровне планирования непосредственно до фазы его выполнения. Мы отложим обсуждение этой темы до раздела 11.5, а в этом разделе сконцентрируемся на обсуждении идеи ►**иерархической декомпозиции** — идеи, пронизывающей почти все попытки управлять сложностью. Например, сложное программное обеспечение создается из иерархии подпрограмм и классов; армии, правительства и корпорации имеют организационную иерархию. Ключевое преимущество иерархической структуры состоит в том, что на каждом уровне иерархии ее компоненты — вычислительные задачи, военные миссии или административные функции — сводятся к небольшому количеству мероприятий на следующем, более низком уровне, поэтому расчетная стоимость нахождения правильного способа упорядочивания этих мероприятий для текущей проблемы будет невелика.

11.4.1. Действия высокого уровня

Основной способ формального представления иерархической декомпозиции, который мы примем, взят из области ►**иерархических сетей задач** или HTN (*Hierarchical Task Network*) планирования. На данный момент мы принимаем полную наблюдаемость и детерминизм среды, а также наличие множества действий, которые далее будем называть ►**примитивными действиями**, со стандартными схемами "предусловия–эффект". Ключевой дополнительной концепцией является ►**действие высокого уровня**, или HLA (High-Level Action), например действие "Проезд в аэропорт Сан-Франциско". Каждое действие высокого уровня (HLA) имеет одно или несколько возможных ►**уточнений** (*refinement*) в виде последовательности действий, каждое из которых может быть некоторым HLA или примитивным действием. Например, действие "Проезд в аэропорт Сан-Франциско", которое формально можно представить как *Go*(*Home*, *SFO*), может иметь два

возможных уточнения, как показано на рис. 11.7. На том же рисунке показано **рекурсивное** уточнение для навигации в мире пылесоса: чтобы добраться до пункта назначения, сделать шаг, а затем идти к пункту назначения.

Refinement(*Go*(*Home*, *SFO*),
  STEPS: [*Drive*(*Home*, *SFOLongTermParking*),
    Shuttle(*SFOLongTermParking*, *SFO*)])
Refinement(*Go*(*Home*, *SFO*),
  STEPS: [*Taxi*(*Home*, *SFO*)])

Refinement(*Navigate*([*a*, *b*], [*x*, *y*]),
  PRECOND: $a = x \wedge b = y$
  STEPS: [])
Refinement(*Navigate*([*a*, *b*], [*x*, *y*]),
  PRECOND: *Connected*([*a*, *b*], [$a-1$, *b*])
  STEPS: [*Left*, *Navigate*([$a-1$, *b*], [*x*, *y*])])
Refinement(*Navigate*([*a*, *b*], [*x*, *y*]),
  PRECOND: *Connected*([*a*, *b*], [$a+1$, *b*])
  STEPS: [*Right*, *Navigate*([$a+1$, *b*], [*x*, *y*])])

Рис. 11.7. Определения возможных уточнений для двух действий высокого уровня (HLA): "Проезд в аэропорт Сан-Франциско" и "Навигация в мире пылесоса". В последнем случае обратите внимание на рекурсивный характер уточнений и использование предварительных условий

Эти примеры показывают, что действия высокого уровня и их уточнения воплощают знания о том, *как делать что-то*. Например, уточнения для действия *Go*(*Home*, *SFO*) указывают, что для того, чтобы добраться до аэропорта, можно поехать туда на собственной автомашине или взять такси, а такие действия, как купить молоко, сесть на диван или пойти конем на e4, здесь не рассматриваются.

Уточнение, содержащее только примитивные действия, называется ▶**реализацией** HLA. В клеточном мире обе последовательности [*Right*, *Right*, *Down*] и [*Down*, *Right*, *Right*] реализуют HLA *Navigate*([1,3], [3,2]). Реализация плана высокого уровня (последовательности HLA) представляет собой объединение реализаций каждого HLA в единую последовательность.

Имея определения "предусловия–эффект" для каждого примитивного действия, можно легко определить, достигает ли данная реализация плана высокого уровня поставленной цели.

Таким образом, можно сказать, что ➨*план высокого уровня достигает цели из заданного состояния, если хотя бы одна из его реализаций достигает цели из этого состояния*. В этом определении фраза "хотя бы одна" является ключевой, — не *все*

реализации должны достигать цели, поэтому агенту необходимо решить, какую именно реализацию он будет выполнять. Следовательно, множество возможных реализаций в HTN-планировании, каждая из которых может иметь различный результат, — это не то же самое, что множество возможных исходов в недетерминированном планировании. Там мы требуем, чтобы план работал для *всех* результатов; поскольку агент не может выбрать результат, это делает природа.

Самый простой случай — это действие высокого уровня (HLA), у которого есть только одна реализация. В этом случае мы можем вычислить предусловия и эффекты HLA непосредственно из предусловий и эффектов реализации (см. упражнение 11.21), а затем обработать HLA точно так, как если бы это было одно примитивное действие. Можно показать, что построение правильного набора HLA может иметь следствием падение временной сложности слепого поиска от экспоненциального до линейного относительно глубины решения, хотя разработка такого набора HLA сама по себе может быть нетривиальной задачей. Когда у действий высокого уровня есть несколько возможных реализаций, то возможны два разных подхода: первый — это поиск среди реализаций той, которая работает, как предлагается в разделе 11.4.2, а второй — рассуждения непосредственно о самих HLA, несмотря на множественность их реализаций, как объясняется в разделе 11.4.3. Последний подход позволяет получить доказуемо правильные абстрактные планы без необходимости рассмотрения их реализации.

11.4.2. Поиск примитивных решений

HTN-планирование часто формулируется с помощью единственного действия "верхнего уровня" под названием *Act* с намерением отыскать такую реализацию *Act*, которая достигает цели. Этот подход является абсолютно общим. Например, задачи классического планирования могут быть определены следующим образом: для каждого примитивного действия a_i дать одно уточнение действия *Act* с этапами [a_i, *Act*]. В результате будет создано рекурсивное определение действия *Act*, позволяющее добавлять новые действия. Но здесь необходим какой-то способ остановить рекурсию: это можно сделать, предоставив еще одно уточнение действия *Act* — с пустым списком этапов и предусловием, эквивалентным цели задачи. По сути это указание о том, что если цель уже достигнута, то правильная реализация — ничего не делать.

Такой подход приводит к простому алгоритму: повторять выбор некоторого действия высокого уровня (HLA) в текущем плане и заменять его одним из его уточнений, пока план не достигнет цели. Одна из возможных реализаций этого алгоритма на базе поиска в ширину по дереву представлена на рис. 11.8. Планы рассматриваются в порядке глубины вложенности уточнений, а не количества примитивных этапов. Легко спроектировать версию этого алгоритма с поиском по графу, а также версии с поиском в глубину и поиском с итеративным углублением.

```
function HIERARCHICAL-SEARCH(problem, hierarchy) returns решение или индикатор
                                                          отказа failure

  frontier ← очередь типа FIFO, содержащая единственный элемент [Act]
  while true do
    if IS-EMPTY(frontier) then return failure
    plan ← POP(frontier)          // выбор первого сверху плана в очереди frontier
    hla ← первое HLA в plan, или null, если его нет
    prefix, suffix ← подпоследовательности действий до и после hla в plan
    outcome ← RESULT(problem.INITIAL, prefix)
    if hla есть null then          // план состоит из примитивных действий,
                                   // и в outcome находится результат его выполнения
      if problem.IS-GOAL(outcome) then return plan
    else for each sequence in REFINEMENTS(hla, outcome, hierarchy) do
      добавить APPEND(prefix, sequence, suffix) в очередь frontier
```

Рис. 11.8. Реализация прямого иерархического планирования с поиском в ширину. Исходный план предоставляется алгоритму как [*Act*]. Функция REFINEMENTS возвращает множество последовательностей действий, по одной для каждого уточнения HLA, предусловия которого удовлетворяются указанным состоянием, *outcome*

В сущности, эта форма иерархического поиска исследует пространство последовательностей, соответствующих содержащемуся в библиотеке HLA знанию о том, как все это может быть сделано. Большое количество знаний может быть закодировано не только в последовательности действий, указанных в каждом уточнении, но и в предпосылках уточнений. Для некоторых проблемных областей HTN-планировщики уже оказались в состоянии генерировать огромные планы при очень небольшом количестве выполняемого поиска. Например, планировщик O-PLAN (Белл и Тейт [158], 1985), сочетающий HTN-планирование с составлением расписаний, успешно использовался для разработки планов производства в корпорации Hitachi. Типичная задача планирования охватывала производственную линию из 350 различных продуктов, 35 сборочных станций и более 2000 различных операций. Планировщик генерировал 30-дневный график работ с тремя 8-часовыми сменами в сутки, включающий десятки миллионов этапов. Другим важным аспектом HTN-планов является то, что они по определению иерархически структурированы, а это обычно существенно облегчает их понимание людьми.

Вычислительные преимущества иерархического поиска легко можно увидеть, рассмотрев идеализированный случай. Предположим, что у задачи планирования имеется решение с d примитивными действиями. В случае планировщика с неиерархическим прямым поиском в пространстве состояний с b допустимыми действиями в каждом состоянии затраты времени можно оценить как $O(b^d)$, как объясняется в главе 3. Для HTN-планировщика предположим строго регулярную

структуру уточнений: любое не примитивное действие имеет r возможных уточнений, каждое размером в k действий на очередном более низком уровне. Требуется узнать, сколько существует различных деревьев уточнения при данной структуре. Итак, если имеется d действий на примитивном уровне, то количество уровней ниже корня дерева составит $\log_k d$, поэтому количество внутренних узлов уточнений будет равно $1 + k + k^2 + \ldots + k^{\log_k d - 1} = (d-1)/(k-1)$. Каждый внутренний узел имеет r возможных уточнений, поэтому может быть построено $r^{(d-1)/(k-1)}$ различных деревьев декомпозиции.

Внимательно посмотрев на эту формулу, можно заметить, что, сохраняя значение r небольшим, а значение k — достаточно большим, можно получить громадную экономию: мы говорим о корне k-й степени из затрат для неиерархического поиска, если b и r сравнимы по величине. Малое значение r и большое значение k означает, что библиотека HLA будет включать небольшое количество уточнений, каждое из которых будет генерировать длинную последовательность действий. Однако это не всегда возможно: длинные последовательности действий, которые можно использовать для решения широкого круга задач, встречаются крайне редко.

Ключом к успешному применению HTN-планирования является наличие библиотеки планов, содержащей уже известные способы реализации сложных высокоуровневых действий. Один из способов создания такой библиотеки — это *изучение* методов из опыта решения проблем. После мучительного опыта построения плана "с нуля" агент может сохранить этот план в библиотеке как метод реализации высокоуровневого действия, определенного в задаче. Действуя таким образом, агент со временем может становиться все более и более компетентным по мере построения новых методов на основе уже известных. Одним из важных аспектов этого процесса обучения является возможность *обобщения* методов, которые уже были построены, за счет устранения деталей, специфических для определенных экземпляров задач (например, название компании-исполнителя или адрес местонахождения объекта), и сохранения только ключевых элементов плана. Нам кажется невероятным, что люди смогли бы стать настолько компетентными, насколько они есть, без какого-либо аналогичного внутреннего механизма.

11.4.3. Поиск для абстрактных решений

Алгоритм иерархического поиска из предыдущего раздела выполняет уточнение действий высокого уровня (HLA) на всем пути к получению последовательности из примитивных действий, чтобы определить, является ли план работоспособным. Это противоречит здравому смыслу: нужно иметь возможность определить, что следующий план высокого уровня с двумя HLA

[*Drive*(*Home*, *SFOLongTermParking*), *Shuttle*(*SFOLongTermParking*, *SFO*)]

позволяет добраться в аэропорт без необходимости определения точного маршрута, выбора места для парковки и т.д. Решение состоит в том, чтобы составить

описание действий высокого уровня (HLA) на уровне "предусловия–эффект", как это делается для примитивных действий. На основании этих описаний должно быть не сложно доказать, что план высокого уровня достигает цели. Это положение, можно сказать, является "святым Граалем" иерархического планирования, поскольку если можно вывести план высокого уровня, доказуемо достигающий поставленной цели, работая в небольшом пространстве поиска мероприятий высокого уровня, то далее можно просто выполнять этот план, работая над задачей уточнения каждого этапа плана в отдельности. Это дает нам экспоненциальное сокращение времени выполнения, к которому мы и стремились.

Чтобы этот подход работал, ситуация должна быть такой, когда каждый план высокого уровня, "претендующий" на достижение цели (в силу описаний его этапов), действительно достигает цели в том смысле, как он был определен ранее: должна существовать по крайней мере одна реализация, которая действительно достигает цели. Это свойство называется ▶**свойством нисходящего уточнения** для описаний действий высокого уровня (HLA).

Создать описание HLA, удовлетворяющее свойству нисходящего уточнения, в принципе, легко: до тех пор, пока описания являются *истинными*, любой план высокого уровня, претендующий на достижение цели, должен фактически обеспечивать это, — в противном случае описания содержат некоторые ложные утверждения о том, что выполняют HLA. Мы уже обсуждали, как писать истинные описания для задач высоко уровня, имеющих точно одну реализацию (упражнение 11.21); проблемы возникают лишь тогда, когда HLA имеет *несколько* реализаций. Как можно описать эффекты действия, которое можно реализовать различными способами?

Один надежный ответ (по крайней мере, для задач, в которых все предпосылки и цели являются положительными) состоит во включении в эффект только тех положительных результатов, которые достигаются в каждой реализации этой HLA, и негативных результатов, достигаемых в *любой* его реализации. В этом случае свойство нисходящего уточнения удовлетворяется. К сожалению, такая семантика для HLA слишком осторожна.

Еще раз вернемся к действию высокого уровня *Go*(*Home*, *SFO*), которое имеет два уточнения, и предположим для конкретности простой мир, в котором любой может добраться в аэропорт на автомобиле и оставить его на парковке, а поездка на такси как предварительное условие требует иметь при себе наличные деньги. В таком случае действие *Go*(*Home*, *SFO*) не всегда обеспечит доставку агента в аэропорт. В частности, цель не будет достигнута, если *Cash* имеет значение *false*, поэтому мы не можем принять утверждение *At*(*Agent*, *SFO*) как эффект данного HLA. Однако отсутствие у агента наличных денег не имеет никакого значения, если он отправится в аэропорт на своем автомобиле. Требование о том, что любой эффект сохраняется для *каждой* реализации, является эквивалентным предположению, что кто-то другой — некий противник — будет выбирать реализацию для агента. Это трактует множественность исходов HLA так, как если бы HLA

было **недетерминированным** действием, как в разделе 4.3. Однако в нашем случае агент сам выбирает реализацию.

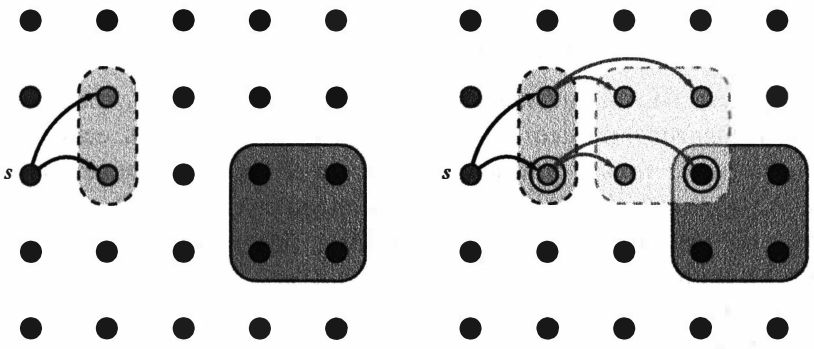
Сообщество языков программирования ввело термин ▶**демонический недетерминизм** для случая, когда выбор делает противник, противопоставляя его ▶**ангельскому недетерминизму**, когда выбор делает сам агент. Мы заимствуем этот термин, чтобы определить ▶**ангельскую семантику** для описания действий высокого уровня (HLA). Основной концепцией, необходимой для понимания ангельской семантики, является ▶**достижимое множество** из некоторого HLA: при заданном состоянии s достижимым множеством для HLA h, что записывается как REACH(s, h), является множество состояний, достижимых в любой из реализаций этого HLA.

Ключевая идея состоит в том, что агент может выбрать, на *каком* элементе из достижимого множества он остановится, при выполнении HLA. Следовательно, действие высокого уровня с несколькими уточнениями является более "мощным", чем такое же HLA, но с меньшим количеством уточнений. Можно также определить достижимое множество и для некоторой последовательности из HLA. Например, достижимым множеством для последовательности $[h_1, h_2]$ является объединение всех достижимых множеств, полученных при применении h_2 в каждом состоянии из достижимого множества h_1:

$$\text{REACH}(s, [h_1, h_2]) = \bigcup_{s' \in \text{REACH}(s, h_1)} \text{REACH}(s', h_2).$$

С учетом этих определений план высокого уровня — последовательность HLA — достигает цели, если его достижимое множество *пересекается* с множеством целевых состояний. (Сравните это условие с гораздо более сильным условием для демонической семантики, в которой *каждый* член достижимого множества должен быть состоянием цели.) И наоборот, если достижимое множество не пересекается с множеством целевых состояний, то план, определенно, не сработает. На рис. 11.9 эти идеи иллюстрируются наглядно.

Введение понятия достижимого множества приводит к простому алгоритму: поиск среди планов высокого уровня для отыскания того, достижимое множество которого пересекается с целевым множеством. Как только это произойдет, алгоритм может *выполнить* этот абстрактный план, зная, что он сработает, и сосредоточиться на дальнейшем уточнении этого плана. Мы вернемся к алгоритмическим вопросам позже; а пока рассмотрим, как эффекты HLA — т.е. достижимое множество для каждого возможного начального состояния — могут быть представлены. Примитивное действие может присвоить флюенте значение *истина* или *ложь* либо оставить ее *без изменений*. (При наличии условных эффектов (см. раздел 11.5.1) появляется четвертая возможность: изменить значение переменной на противоположное.)



а) б)

Рис. 11.9. Схематические примеры достижимых множеств. Множество целевых состояний находится на темно-сером фоне. Черные и серые стрелки указывают на возможные реализации HLA h_1 и h_2 соответственно. **а)** Достижимое множество для h_1 в состоянии s. **б)** Достижимое множество для последовательности $[h_1, h_2]$ в состоянии s, — поскольку оно пересекается с целевым множеством состояний, эта последовательность HLA достигает цели

При использовании ангельской семантики действие высокого уровня может сделать больше: оно может управлять значением флюенты, *устанавливая* его равным *true* или *false* в зависимости от того, какая реализация была выбрана. А это означает, что HLA может оказывать на флюенту девять различных эффектов: если в начале переменная имеет значение *true*, она может в любом случае сохранить это значение, всегда получать значение *false* или иметь возможность выбора. Если флюента в начале имеет значение *false*, она может в любом случае сохранить его, всегда получать значение *true* или иметь возможность выбора. Кроме того, эти три варианта для обоих случаев могут произвольно комбинироваться, что в итоге дает девять возможных эффектов.

В общем, все это довольно сложно. Здесь мы будем использовать язык списков добавления и списков удаления (а не флюенты со значениями *true*/*false*) вместе с символом ~, означающим "возможно, если агент сделает такой выбор". Таким образом, эффект $\tilde{+}A$ означает "возможно, добавить A", т.е. либо оставить A без изменений, либо сделать его истинным. Аналогично $\tilde{-}A$ означает "возможно, удалить A", а $\tilde{\pm}A$ означает "возможно, добавить или удалить A". Например, HLA *Go*(*Home*, *SFO*) с двумя уточнениями, представленными на рис. 11.7, возможно, удаляет флюенту *Cash* (если агент решает взять такси), поэтому оно должно иметь эффект $\tilde{-}Cash$. Таким образом, мы видим, что такие описания HLA являются *выводимыми* из описаний их уточнений. Теперь предположим, что даны следующие схемы для двух действий высокого уровня h_1 и h_2:

$$Action(h_1, \text{PRECOND: } \neg A, \text{EFFECT: } A \wedge \tilde{-} B),$$
$$Action(h_2, \text{PRECOND: } \neg B, \text{EFFECT: } \tilde{+} A \wedge \tilde{\pm} C).$$

Это означает, что HLA h_1 добавляет A и, возможно, удаляет B, тогда как HLA h_2, возможно, добавляет A и имеет полный контроль над C. Теперь, если в начальном состоянии истинно только B и целью является $A \wedge C$, последовательность действий $[h_1, h_2]$ достигает цели: мы выбираем такую реализацию h_1, в которой B переводится в состояние *false*, а затем выбираем ту реализацию действия h_2, которая оставляет A в состоянии *true* и делает C истинным.

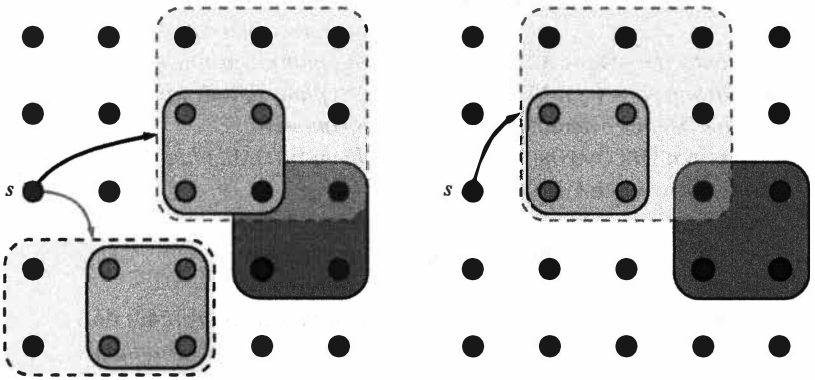
В приведенном выше обсуждении предполагается, что указанные эффекты HLA — достижимое множество для любого заданного состояния — могут быть точно описаны путем описания эффекта для каждой флюенты. Было бы очень хорошо, если бы это всегда было так, но во многих случаях мы можем только *аппроксимировать* эффекты, поскольку HLA может иметь бесконечно много реализаций и при этом вырабатывать произвольно меняющиеся достижимые множества — наподобие доверительного состояния сложной формы, представленного на рис. 7.21. Например, выше было сказано, что при выполнении HLA *Go*(*Home*, *SFO*) возможно удаление *Cash*, но также возможно и добавление *At*(*Car*, *SFOLongTermParking*), однако и то, и другое не может произойти одновременно — в действительности должно быть выполнено лишь что-то одно. Как и в случае доверительных состояний, нам, возможно, потребуется давать *приблизительные* описания. Мы будем использовать два вида приближенного описания: ▶**оптимистическое описание** $\text{REACH}^+(s, h)$ для HLA h может переоценивать его достижимое множество, тогда как ▶**пессимистическое описание** $\text{REACH}^-(s, h)$ может недооценивать его достижимое множество. Таким образом, мы имеем

$$\text{REACH}^-(s, h) \subseteq \text{REACH}(s, h) \subseteq \text{REACH}^+(s, h).$$

Например, оптимистичное описание HLA *Go*(*Home*, *SFO*) указывает, что возможно удаление *Cash и* возможно добавление *At*(*Car*, *SFOLongTermParking*). Другим хорошим примером является головоломка ***Восемь***, половина состояний которой недоступна из любого заданного состояния: оптимистическое описание HLA *Act* в этом случае вполне может включать в себя все пространство состояний, поскольку точное достижимое множество имеет довольно сложную форму.

При использовании приблизительных описаний проверку того, достигает ли план цели, следует слегка изменить. Если оптимистическое достижимое множество плана не пересекается с целевым, то этот план не работает; если пессимистическое достижимое множество пересекается целевым множеством, то план действительно работает (рис. 11.10, *а*). При точных описаниях план либо работает, либо нет, но в случае приблизительных описаний возникает некоторое промежуточное состояние: если оптимистическое достижимое множество пересекается с целевым, а пессимистическое достижимое множество — нет, то мы не можем

сказать, работает ли план (рис. 11.10, *б*). Когда это обстоятельство имеет место, неопределенность может быть устранена за счет уточнения плана, — это очень распространенная ситуация для мышления человека. Например, при планировании вышеупомянутых двухнедельных каникул на Гавайях можно было бы принять решение проводить по два дня на каждом из семи островов архипелага. Однако благоразумие подсказывало бы, что этот амбициозный план следует доработать, добавив к нему определенные детали переезда с острова на остров.



а) б)

Рис. 11.10. Достижение цели для планов высокого уровня (HLA) с приблизительными описаниями. Множество целевых состояний находится на темно-сером фоне. Для каждого плана показаны пессимистические (сплошной контур, серый фон) и оптимистические (пунктирный контур, светло-серый фон) достижимые множества. **а)** План, указанный черной стрелкой, определенно, достигает цели, тогда как план, указанный серой стрелкой, определенно, нет. **б)** План, который, *возможно,* достигает цели (оптимистическое достижимое множество пересекается с целевым), но *необязательно* ее достигает (пессимистическое достижимое множество не пересекается с целевым). Этот план необходимо доработать, чтобы выяснить, действительно ли он достигает цели

Алгоритм иерархического планирования с приблизительными ангельскими описаниями показан на рис. 11.11. Для простоты он был реализован в той же общей схеме, которая использовалась ранее, на рис. 11.8, т.е. поиск в ширину в пространстве уточнений. Как только что объяснялось, алгоритм может обнаруживать планы, которые будут и которые не будут работать, проверяя наличие пересечения оптимистического и пессимистического достижимых множеств с целевым. (Детали того, как вычисляются достижимые множества плана, учитывая приблизительные описания каждого его этапа, рассматриваются в упражнении 11.19.)

```
function ANGELIC-SEARCH(problem, hierarchy, initialPlan) returns решение
                                                  или индикатор отказа fail

  frontier ← очередь типа FIFO, содержащая единственный элемент initialPlan
  while true do
    if EMPTY?(frontier) then return fail
    plan ← POP(frontier)          // выбор спервого сверху элемента в очереди frontier
    if REACH+(problem.INITIAL, plan) пересекается с problem.GOAL then
      if plan примитивный then return plan   // достижимое множество REACH+
                                             // является точным для примитивных планов
      guaranteed ← REACH−(problem.INITIAL, plan) ∩ problem.GOAL
      if guaranteed ≠ { } и MAKING-PROGRESS(plan, initialPlan) then
        finalState ← любой элемент из guaranteed
        return DECOMPOSE(hierarchy, problem.INITIAL, plan, finalState)
      hla ← некоторое HLA в plan
      prefix, suffix ← пара действий, предшествующее
                       и последующее для hla в plan
      outcome ← RESULT(problem.INITIAL, prefix)
      for each sequence in REFINEMENTS(hla, outcome, hierarchy) do
        frontier ← Insert(APPEND(prefix, sequence, suffix), frontier)

function DECOMPOSE(hierarchy, s0, plan, sf) returns решение

  solution ← пустой план
  while plan не пуст do
    action ← REMOVE-LAST(plan)
    si ← состояние in REACH−(s0, plan), такое, что sf ∈ REACH−(si, action)
    problem ← задача с INITIAL = si и GOAL = sf
    solution ← APPEND(ANGELIC-SEARCH(problem, hierarchy, action), solution)
    sf ← si
  return solution
```

Рис. 11.11. Алгоритм иерархического планирования, использующий ангельскую семантику для определения и выполнения планов высокого уровня, позволяющих достичь цели, игнорируя при этом те планы высокого уровня, которые этого не позволяют. Предикат MAKING-PROGRESS нужен для проверки, что алгоритм не застрял в бесконечной регрессии уточнений. На более высоком уровне алгоритм ANGELIC-SEARCH вызывается с HLA [*Act*] в качестве аргумента *initialPlan*

Когда работоспособный абстрактный план найден, алгоритм выполняет декомпозицию исходной задачи на подзадачи, по одной для каждого этапа плана. Начальное и целевое состояния для каждой подзадачи получаются за счет регрессивного поиска гарантированно достигаемого целевого состояния с использованием схем действий для каждого этапа плана. (Обсуждение того, как

выполняется регрессивный поиск, см. в разделе 11.2.2.) На рис. 11.9, *б* проиллюстрирована основная идея: целевое состояние с гарантированной достижимостью обведено двойным контуром и находится справа, а левое обведенное двойным контуром состояние представляет собой промежуточную цель, найденную посредством регрессивного поиска цели в пространстве конечного действия.

Способность выполнить или отказаться от выполнения планов высокого уровня может дать алгоритму ANGELIC-SEARCH значительное вычислительное преимущество в сравнении с алгоритмом HIERARCHICAL-SEARCH, который, в свою очередь, может иметь большое преимущество над простым старым алгоритмом BREADTH-FIRST-SEARCH. Например, рассмотрим очистку большого экземпляра мира пылесоса, состоящего из некоторого количества комнат, соединенных узкими коридорами, каждая комната которого представляет собой прямоугольник размером $w \times h$ квадратов. Имеет смысл построить некоторое HLA для действия *Navigate* (как показано на рис. 11.7) и еще одно — для действия *CleanWholeRoom* (очистить всю комнату). (Очистка всей комнаты может быть реализована как циклическое применение других HLA, например HLA очистки отдельной горизонтальной серии квадратов в комнате.) Поскольку имеется пять примитивных действий, стоимость выполнения алгоритма BREADTH-FIRST-SEARCH можно оценить как 5^d, где d является длиной кратчайшего решения (она примерно вдвое больше общего числа квадратов), — фактически этот алгоритм не способен справиться даже с двумя комнатами размером 3×3. Алгоритм HIERARCHICAL-SEARCH более эффективен, но по-прежнему страдает от экспоненциального роста вычислительных затрат, поскольку проверяет все пути очистки, совместимые с иерархией. Вычислительные затраты алгоритма ANGELIC-SEARCH возрастают приблизительно в линейной зависимости от количества квадратов — он выполняет лишь хорошие высокоуровневые последовательности этапов очистки комнат и навигации, отсекая все остальные варианты.

Очистку множества комнат посредством поочередной уборки каждой из них вряд ли можно считать высоким научным достижением: подобная задача не вызывают затруднений у людей благодаря иерархической структуре самой задачи. Если проанализировать, насколько трудно людям решать даже небольшие головоломки, такие как ***Восемь***, кажется весьма вероятным, что способность людей решать сложные проблемы проистекает не за счет эффективного использования комбинаторики, а скорее из умения абстрагировать задачи, а затем выполнять их декомпозицию с целью устранения комбинаторики.

Подход с использованием ангельской семантики можно расширить с целью поиска решения с наименьшими затратами, просто обобщив понятие достижимого множества. Вместо того чтобы состояние рассматривалось как достижимое либо нет, каждое состояние связывается со стоимостью самого эффективного способа его достижения. (Для недостижимых состояний стоимость бесконечна.) Оптимистические и пессимистические описания ограничивают диапазон этих стоимостей. Таким образом, ангельский поиск может найти доказуемо оптимальные абстрактные планы без необходимости рассматривать их реализацию. Тот же

подход можно использовать для получения эффективных алгоритмов ▶**иерархического прогнозирования** для выполнения интерактивного поиска в стиле LRTA* (см. раздел 4.5.3).

В некотором смысле такие алгоритмы отражают определенные аспекты размышлений человека при решении таких задач, как планирование отпуска на Гавайях, — рассмотрение альтернатив первоначально осуществляется на абстрактном уровне, возможно, в течение длительного периода времени, причем одни части плана остаются довольно абстрактными вплоть до момента их исполнения — например, как именно провести два ленивых дня на острове Молокаи, — тогда как другие части плана сразу разрабатываются в деталях, например выбираются наиболее подходящие авиарейсы и бронируются номера в самых привлекательных местах проживания. И если этого не сделать, не будет достаточных гарантий, что этот план вообще будет осуществимым.

11.5. Планирование и действие в недетерминированных проблемных областях

В этом разделе планирование распространяется на решение задач в частично наблюдаемых, недетерминированных и неизвестных проблемных областях. Основные концепции в целом отражают те, которые приведены в главе 4, но есть и различия, возникающие из-за использования развернутых представлений вместо атомарных. Это влияет как на способ представления способности агента к действию и наблюдению, так и на способ представления **доверительных состояний** — множества возможных физических состояний, в которых может находиться агент — для частично наблюдаемых сред. Также появляется возможность воспользоваться преимуществами многих методов, независимых от проблемной области, предложенных в разделе 11.3 для вычисления эвристик поиска.

Будут рассмотрены темы **бессенсорного планирования** (также называемого **конформным планированием**) для ненаблюдаемых проблемных областей; **условное планирование** (или *планирование с учетом непредвиденных ситуаций*) — для частично наблюдаемых и недетерминированных проблемных областей, а также **непрерывное планирование** (или *интерактивное планирование*) и **перепланировка** — для неизвестных проблемных областей. Все это позволит успешно справляться с решением значительных по размерам задач реального мира.

Рассмотрим следующую задачу: даны стул (*Chair*) и стол (*Table*), а цель состоит в том, чтобы сделать их соответствующими друг другу, т.е. имеющими одинаковый цвет (*Color*). В исходном состоянии есть две банки (*Can*) краски, но цвета краски и мебели неизвестны. Исходно в поле зрения (*InView*) агента находится только стол.

$$Init(Object(Table) \land Object(Chair) \land Can(C_1) \land Can(C_2) \land InView(Table))$$
$$Goal(Color(Chair, c) \land Color(Table, c))$$

Агенту доступны два действия: снятие крышки (*RemoveLid*) с банки с краской и покраска (*Paint*) объекта краской из открытой (*Open*) банки.

$$\begin{aligned}&\mathit{Action}(\mathit{RemoveLid}(\mathit{can}),\\&\quad \text{PRECOND: } \mathit{Can}(\mathit{can})\\&\quad \text{EFFECT: } \mathit{Open}(\mathit{can}))\\&\mathit{Action}(\mathit{Paint}(x, \mathit{can}),\\&\quad \text{PRECOND: } \mathit{Object}(x) \wedge \mathit{Can}(\mathit{can}) \wedge \mathit{Color}(\mathit{can}, c) \wedge \mathit{Open}(\mathit{can})\\&\quad \text{EFFECT: } \mathit{Color}(x, c))\end{aligned}$$

Схемы действий просты и понятны за одним исключением: предварительные условия и эффекты теперь могут содержать переменные, которые не являются частью списка переменных действия. Так, в действии *Paint*(x, *can*) не упоминается переменная c, представляющая цвет краски в банке. В случае полностью наблюдаемой среды это не допускается, — потребуется явно упомянуть цвет краски в действии: *Paint*(x, *can*, c). Но в случае частично наблюдаемой среды мы можем знать, а можем и не знать цвет краски в конкретной банке.

Чтобы решить поставленную задачу в частично наблюдаемой среде, агенту потребуется рассуждать о результатах своего восприятия, получаемых им в процессе выполнения плана. Реальные данные восприятия будут предоставляться датчиками агента, когда он действительно будет выполнять действия, поэтому при планировании ему потребуется некоторая модель его датчиков. В главе 4 эта модель была представлена функцией PERCEPT(s). Для целей планирования потребуется дополнить описание на языке PDDL схемой нового типа — ▶**схемой восприятия**.

$$\begin{aligned}&\mathit{Percept}(\mathit{Color}(x, c),\\&\quad \text{PRECOND: } \mathit{Object}(x) \wedge \mathit{InView}(x))\\&\mathit{Percept}(\mathit{Color}(\mathit{can}, c),\\&\quad \text{PRECOND: } \mathit{Can}(\mathit{can}) \wedge \mathit{InView}(\mathit{can}) \wedge \mathit{Open}(\mathit{can}))\end{aligned}$$

Первая схема сообщает, что всякий раз, когда агент будет видеть некоторый объект, он будет воспринимать цвет этого объекта (т.е. для объекта x агент узнает значение истинности выражения *Color*(x, c) для всех c). Вторая схема говорит, что если агент увидит открытую банку, то воспримет цвет краски в ней. Поскольку в этом мире нет внешних событий, цвет объекта останется прежним, даже если он непосредственно не воспринимается агентом, пока он не выполнит действие, изменяющее цвет этого объекта. И конечно, агенту также потребуется действие, позволяющее объектам (по одному за раз) попасть в поле зрения агента.

$$\begin{aligned}&\mathit{Action}(\mathit{LookAt}(x),\\&\quad \text{PRECOND: } \mathit{InView}(y) \wedge (x \neq y)\\&\quad \text{EFFECT: } \mathit{InView}(x) \wedge \neg \mathit{InView}(y))\end{aligned}$$

В случае полностью наблюдаемой проблемной среды в схеме *Percept* отсутствовали бы какие-либо предусловия для каждой флюенты. С другой стороны,

бессенсорный агент вообще не имеет схем *Percept*. Обратите внимание, что даже агент вообще без каких-либо датчиков также может решить эту задачу с покраской мебели. Одно из возможных решений состоит в том, чтобы открыть любую банку с краской и покрасить ею как стул, так и стол, **принудительно** приведя их к одному и тому же цвету (даже если сам агент не знает, какой это цвет).

Агент, использующий условное планирование, способен создать лучший план. Прежде всего он посмотрит на стол и стул, чтобы определить их цвет, и если он одинаков, то план выполнен. Затем, если это не так, агент посмотрит на банки с краской и, если есть банка с краской такого же цвета, как и у одного из предметов мебели, покрасит этой краской и другой предмет. В противном случае агент покрасит и стол, и стул одной произвольно выбранной краской.

Наконец непрерывно планирующий (*online planning*) агент может сначала создать условный план с меньшим количеством ветвей — возможно, проигнорировав возможность того, что краска ни в одной из банок не совпадет с цветом мебели — и решать проблемы, когда они возникают, посредством перепланировки. Он также способен справляться с неточностями в схемах его действий. В то время как при условном планировании просто предполагается, что результат выполняемых действий всегда будет успешным — достаточно покрасить стул, и на этом работа заканчивается, — перепланирующий агент способен проверить полученные результаты и составить дополнительный план для исправления любых непредвиденных отказов: наличия неокрашенных участков или просвечивания прежней краски через новое покрытие.

В реальном мире агенты обычно используют комбинацию разных подходов. Производители автомобилей продают запасные колеса и надувные подушки безопасности, что представляет собой физическое воплощение ответвлений условного плана, позволяющих справляться с проколами шин или авариями. С другой стороны, большинство водителей автомобилей никогда не учитывают такой возможности, поэтому отвечают на возникновение подобных проблем как перепланирующие агенты. Как правило, агенты создают условные планы только для таких непредвиденных ситуаций, которые могут иметь важные последствия и есть достаточно большая вероятность того, что они возникнут. Поэтому водитель автомобиля, решившийся на поездку через пустыню Сахара, должен иметь четко разработанные планы на случай различных поломок и отказов, тогда как рядовая поездка в соседний супермаркет требует гораздо меньше предварительного планирования. Далее каждый из трех упомянутых выше подходов к планированию будет рассмотрен более подробно.

11.5.1. Бессенсорное планирование

В разделе 4.4.1 была представлена основная идея поиска в пространстве доверительных состояний с целью нахождения решения для бессенсорных задач. Преобразование бессенсорной задачи планирования в задачу планирования доверительных

состояний выполняется примерно так же, как и в разделе 4.4.1. Основные различия заключаются в том, что базовая модель физического перехода представлена набором схем действий, а доверительное состояние может быть представлено логической формулой, а не явно перечисленным множеством состояний. Будем предполагать, что лежащая в основе задача планирования является детерминированной.

В начальном доверительном состоянии для бессенсорной задачи окраски мебели флюенты *InView* можно игнорировать, поскольку у агента нет никаких датчиков. Кроме того, как известные неизменяемые факты можно принять $Object(Table) \wedge Object(Chair) \wedge Can(C_1) \wedge Can(C_2)$, поскольку они присутствуют в каждом доверительном состоянии. Агент не знает, какого цвета объекты или краска в банках, а также открыты банки или закрыты, но он знает, что объекты и краска в банках имеют цвет: $\forall x \exists c \ Color(x, c)$. После сколемизации (см. раздел 9.5.1) мы получаем начальное доверительное состояние:

$$b_0 = Color(x, C(x)).$$

В классическом планировании, где принято **допущение о замкнутости мира**, можно было бы предположить, что любые флюенты, не упоминаемые явно в определении состояния, имеют значение *false*, но в бессенсорном (и частично наблюдаемом) планировании необходимо перейти к *допущению об открытости мира*, при котором состояния включают как положительные, так и отрицательные флюенты, и если некоторая флюента не упоминается, то ее значение нам просто неизвестно. Следовательно, доверительное состояние точно соответствует множеству возможных миров, удовлетворяющих этой формуле. Учитывая это начальное доверительное состояние, решением будет следующая последовательность действий:

$$[RemoveLid(Can_1), Paint(Chair, Can_1), Paint(Table, Can_1)].$$

Теперь покажем, как доверительное состояние изменяется в процессе прохождения последовательности выполняемых действий, чтобы показать, что конечное доверительное состояние удовлетворяет цели.

Во-первых, отметим, что в некотором доверительном состоянии b агент может рассматривать любое действие, предпосылки которого удовлетворяются в b. (Другие действия не могут использоваться, так как функция определения преемника не определяет эффектов действий, предпосылки которых не могут быть удовлетворены.) В соответствии с уравнением (4.4) в разделе 4.4.1 общая формула для обновления доверительного состояния b при заданном применимом действии a в детерминированном мире выглядит следующим образом:

$$b' = \text{RESULT}(b, a) = \{s' : s' = \text{RESULT}_P(s, a) \text{ и } s \in b\},$$

где функция RESULT_P определяет физическую функцию определения преемника. На текущий момент мы полагаем, что начальное доверительное состояние всегда является конъюнкцией литералов, т.е. формулой 1-CNF. Чтобы построить новое доверительное состояние b', следует рассмотреть, что происходит с каждым

литералом ℓ в каждом физическом состоянии s в доверительном состоянии b, когда применяется действие a. Для литералов, значение истинности которых уже известно в b, значение истинности в b' вычисляется из текущего значения и списка добавления и удаления действия a. (Например, если ℓ находится в списке удаления действия, то в b' добавляется $\neg\ell$.) А как поступить с литералом, значение истинности которого в b нам неизвестно? Здесь возможны три варианта.

1. Если действие добавляет литерал ℓ, то ℓ будет истинным в b' независимо от его исходного значения.
2. Если действие удаляет литерал ℓ, то ℓ будет ложным в b' независимо от его исходного значения.
3. Если действие никак не влияет на литерал ℓ, то ℓ будет сохранять свое первоначальное значение (которое является неизвестным) и не будет представлено в b'.

Следовательно, можно видеть, что процедура вычисления b' практически идентична варианту с полностью наблюдаемой средой, который описывается уравнением (11.1) в разделе 11.1:

$$b' = \text{RESULT}(b, a) = (b - \text{DEL}(a)) \cup \text{ADD}(a).$$

Однако здесь мы не можем полностью использовать семантику множеств, потому что, во-первых, сначала нужно убедиться в том, что b' не содержит как ℓ, так и $\neg\ell$, и во-вторых, потому что атомарные высказывания могут содержать несвязанные переменные. Но это все еще тот случай, когда RESULT(b, a) вычисляется, начиная с b и устанавливая значение *false* для любого атомарного выражения, которое присутствует в списке DEL(a), и значение *true* — для любого атомарного выражения, которое присутствует в списке ADD(a). Например, если к начальному доверительному состоянию b_0 применить действие $RemoveLid(Can_1)$, то можно получить

$$b_1 = Color(x, C(x)) \wedge Open(Can_1).$$

Затем применяем действие $Paint(Chair, Can_1)$ — предусловие $Color(Can_1, c)$ удовлетворяется литералом $Color(x, C(x))$ со связкой $\{x/Can_1, c/C(Can_1)\}$ — и получаем новое доверительное состояние

$$b_2 = Color(x, C(x)) \wedge Open(Can_1) \wedge Color(Chair, C(Can_1)).$$

Наконец применяем действие $Paint(Table, Can_1)$ и получаем

$$b_3 = Color(x, C(x)) \wedge Open(Can_1) \wedge Color(Chair, C(Can_1)) \wedge Color(Table, C(Can_1)).$$

Это конечное доверительное состояние удовлетворяет цели $Color(Table, c) \wedge Color(Chair, c)$ с переменной c, связанной с $C(Can_1)$.

Приведенный выше анализ правила обновления продемонстрировал очень важный факт: ➔ *семейство доверительных состояний, определенных как конъюнкции*

литералов, является замкнутым для обновлений, определяемых схемами действий языка PDDL. То есть, если изначально доверительное состояние представлено как конъюнкция литералов, любое его обновление в результате даст конъюнкцию литералов. Это означает, что в мире с n флюентами любое доверительное состояние может быть представлено в виде конъюнкции размера $O(n)$. Это очень обнадеживающий результат, учитывая, что в таком мире возможно 2^n различных состояний. Он также указывает, что можно компактно представить все подмножества из этих 2^n состояний, которые когда-нибудь потребуются. Более того, процесс проверки доверительных состояний, являющихся подмножествами или надмножествами ранее посещенных доверительных состояний, также будет простым, по крайней мере в пропозициональном случае.

Ложкой дегтя в этой приятной картине является тот факт, что она работает только для схем действий, имеющих *один и тот же эффект* для всех состояний, в которых выполняются их предусловия. Именно это свойство позволяет сохранить представление доверительного состояния в форме 1-CNF. Если же хотя бы один эффект может зависеть от состояния, между флюентами вводятся зависимости и возможность представления доверительного состояния в форме 1-CNF утрачивается.

Рассмотрим, например, простой мир пылесоса, представленный в разделе 3.2.1. Определим в нем флюенты *AtL* и *AtR* для местонахождения робота и флюенты *CleanL* и *CleanR* — для состояния квадратов. Согласно определению задачи действие *Suck* не имеет предусловий — оно всегда может быть выполнено. Сложность здесь в том, что его эффект зависит от расположения робота: когда истинна флюента *AtL*, результатом будет *CleanL*, а когда истинна *AtR*, результатом будет *CleanR*. Для представления таких действий в схемы действий потребуется ввести нечто новое: ►**условный эффект**. Его синтаксис имеет вид "**when** *условие*: *эффект*", где *условие* — это логическая формула для сравнения с текущим состоянием, а *эффект* — это формула, описывающая результирующее состояние. Для мира пылесоса схема действия *Suck* будет такой:

Action(*Suck*,
EFFECT: **when** *AtL*: *CleanL* ∧ **when** *AtR*: *CleanR*).

Применительно к исходному доверительному состоянию *True* результирующее доверительное состояние будет (*AtL* ∧ *CleanL*) ∨ (*AtR* ∧ *CleanR*), которое уже не представлено в форме 1-CNF. (Этот переход можно увидеть на рис. 4.14 в разделе 4.4.1.) В общем случае условные эффекты могут приводить к произвольным зависимостям между флюентами в доверительном состоянии, что в худшем случае имеет следствием экспоненциальный рост размера доверительных состояний.

Важно понимать разницу между предусловиями и условными эффектами. Для *всех* условных эффектов, условия которых удовлетворяются, их эффект будет применен при генерировании результирующего доверительного состояния. Если же ни для какого условного эффекта условие не удовлетворяется, состояние остается

неизменным. С другой стороны, если не удовлетворено *предусловие*, то данное действие вообще не применимо и результирующее состояние является неопределенным. С точки зрения бессенсорного планирования лучше иметь условные эффекты, чем неприменимое действие. Например, очень просто разделить действие *Suck* с условным эффектом на два действия с безусловными эффектами:

Action(*SuckL*,
PRECOND: *AtL*; EFFECT: *CleanL*)
Action(*SuckR*,
PRECOND: *AtR*; EFFECT:*CleanR*).

Теперь у нас есть только безусловные схемы, так что доверительные состояния всегда будут сохранять свою форму 1-CNF, но, к сожалению, у нас отсутствует возможность определить применимость действий *SuckL* и *SuckR* в исходном доверительном состоянии.

Кажется невозможным избежать того, что нетривиальные задачи будут включать в себя доверительные состояния сложной формы, — точно такие, как те, с которыми мы встретились при обсуждении задачи оценки состояния в мире вампуса (см. рис. 7.21 в разделе 7.7.3). Как решение тогда было предложено использовать **первое приближение** к точному доверительному состоянию. Например, доверительное состояние может оставаться в форме 1-CNF, если оно будет включать все литералы, значения истинности которых могут быть определены, а все остальные литералы будут обрабатываться как неизвестные. Хотя такой подход является *непротиворечивым* — в том смысле, что никогда не будет сгенерирован неправильный план, — он является *неполным*, так как может оказаться не в состоянии найти решение для задач, которые по необходимости включают взаимодействие между литералами. Можно привести простейший пример: если цель состоит в том, чтобы робот находился на чистом квадрате, то план [*Suck*] является решением, но бессенсорный агент с обязательным сохранением формы 1-CNF для доверительных состояний его не найдет.

Вероятно, лучшее решение заключается в поиске таких последовательностей действий, которые сохраняют доверительное состояние настолько простым, насколько это возможно. В бессенсорном мире пылесоса последовательность действий [*Right*, *Suck*, *Left*, *Suck*] генерирует следующую последовательность доверительных состояний.

$$b_0 = True$$
$$b_1 = AtR$$
$$b_2 = AtR \wedge CleanR$$
$$b_3 = AtL \wedge CleanR$$
$$b_4 = AtL \wedge CleanR \wedge CleanL$$

Таким образом, агент все же *может* решить задачу, сохраняя при этом свое доверительное состояние в форме 1-CNF, даже если некоторые последовательности действий (например, начинающиеся с *Suck*) выводят его из формы 1-CNF. Этот общий

подход справедлив и для людей: мы всегда выполняем небольшие действия (уточняем время, похлопываем по карманам, чтобы убедиться, что ключ от машины с собой, читаем дорожные указатели, проезжая по незнакомому городу), чтобы устранить неопределенность и сохранить свое доверительное состояние управляемым.

Существует и другой, совершенно отличный подход к проблеме неуправляемого, сложного доверительного состояния: вообще не беспокоиться о его вычислении. Предположим, что b_0 — начальное доверительное состояние, и нам необходимо установить доверительное состояние, которое будет получено в результате выполнения последовательности действий $[a_1, \ldots, a_m]$. Вместо того чтобы выполнять все требуемые вычисления явно, просто представим его как "b_0 затем $[a_1, \ldots, a_m]$". Это ленивое, но однозначное представление доверительного состояния, и оно вполне краткое — $O(n+m)$, где n — это размер начального доверительного состояния (предполагается, что оно представлено в форме 1-CNF), а m — это максимальная длина последовательности действий. Как представление доверительного состояния этот вариант, однако, имеет один недостаток: определение, достигнута ли цель или является ли действие применимым, может потребовать большого объема вычислений.

Расчет может быть реализован как проверка логического следствия: если A_m представляет набор аксиом определения преемника, необходимых для определения возможности реализации действий $a_1, \ldots, a_m$ — как это объяснялось в отношении решателя SATPLAN в разделе 11.2.3, — и G_m утверждает, что целевое выражение будет истинным после m этапов, то план достигает цели, если $b_0 \wedge A_m \models G_m$, т.е. $b_0 \wedge A_m \wedge \neg G_m$ невыполнимо. Учитывая возможности современных решателей SAT, вполне возможно, что проверить это будет намного быстрее, чем вычислять полные доверительные состояния. Например, если ни одно из действий в последовательности не имеет некоторой флюенты цели в своем списке добавления, решатель SAT немедленно это обнаружит. Он также поможет, если частичные результаты о доверительном состоянии — например, знание о том, какие флюенты являются истинными или ложными — кешируются для упрощения последующих вычислений.

Последняя часть головоломки бессенсорного планирования — это эвристическая функция для управления поиском. Смысл эвристической функции здесь такой же, как и в классическом планировании: оценка (возможно, допустимая) стоимости достижения цели из заданного доверительного состояния. При этом в отношении доверительных состояний у нас есть один дополнительный факт: найти решение для любого подмножества доверительного состояния будет обязательно проще, чем найти решение для доверительного состояния в целом.

$$\text{Если } b_1 \subseteq b_2, \text{ то } h^*(b_1) \leq h^*(b_2).$$

Следовательно, любая допустимая эвристика, вычисленная для подмножества, будет допустимой и для доверительного состояния в целом. Наиболее очевидными кандидатами являются одноэлементные подмножества, т.е. отдельные физические

состояния. Можно взять любой случайный набор состояний $s_1, \ldots, s_N$, присутствующих в доверительном состоянии b, применить любую допустимую эвристику h и вернуть

$$H(b) = \max\{h(s_1), \ldots, h(s_N)\}$$

в качестве эвристической оценки для решения b. Можно также использовать недопустимую эвристику, например такую, как эвристика с пустым списком удаления (раздел 11.3), которая, как показывает практика, также работает довольно хорошо.

11.5.2. Условное планирование

В главе 4 было показано, что условное планирование — разработка планов с ветвлением, выполняемым на основании результатов восприятия — подходит для сред с частичной наблюдаемостью, недетерминированностью или и тем, и другим. Для частично наблюдаемой задачи окраски мебели со схемой восприятия, приведенной выше, одно из возможных условных решений будет следующим.

[*LookAt*(*Table*), *LookAt*(*Chair*),
  if *Color*(*Table*, *c*) ∧ *Color*(*Chair*, *c*) **then** *NoOp*
    else [*RemoveLid*(Can_1), *LookAt*(Can_1), *RemoveLid*(Can_2), *LookAt*(Can_2),
      if *Color*(*Table*, *c*) ∧ *Color*(*can*, *c*) **then** *Paint*(*Chair*, *can*)
      else if *Color*(*Chair*, *c*) ∧ *Color*(*can*, *c*) **then** *Paint*(*Table*, *can*)
      else [*Paint*(*Chair*, Can_1), *Paint*(*Table*, Can_1)]]]

Переменные в этом плане следует считать связанными с кванторами существования; во второй строке говорится, что если существует какой-то цвет c, в который уже окрашены и стол, и стул, то для достижения цели агенту ничего не нужно делать. При выполнении этого плана агент с условным планированием может поддерживать свое доверительное состояние как логическую формулу и оценивать условие в каждой ветви плана, определяя, следует ли из доверительного состояния формула условия или же ее отрицание. (Обеспечение того, чтобы агент никогда не оказался в доверительном состоянии, в котором значение истинности формулы условия является неизвестным, возлагается на алгоритм условного планирования.) Обратите внимание, что при условиях, сформулированных в логике первого порядка, формула может быть удовлетворена более чем одним способом, например условие *Color*(*Table*, *c*) ∧ *Color*(*can*, *c*) может быть удовлетворено подстановками {*can*/Can_1} и {*can*/Can_2}, если обе банки содержат краску того же цвета, в который выкрашен стол. В этом случае агент может выбрать любую удовлетворяющую условие подстановку и применить ее к остальной части плана.

Как было показано в разделе 4.4.2, вычисление нового доверительного состояния $\hat{b}$ после выполнения действия a и последующего восприятия выполняется в два этапа. На первом этапе вычисляется доверительное состояние после выполнения действия точно так же, как и в случае бессенсорного агента:

$$\hat{b} = (b - \text{DEL}(a)) \cup \text{ADD}(a),$$

где, как и ранее, предполагается, что доверительное состояние представлено в виде конъюнкции литералов. Второй этап немного сложнее. Предположим, что результаты восприятия были получены как литералы $p_1, \ldots, p_k$. Можно подумать, что необходимо просто добавить их в доверительное состояние; фактически можно также сделать вывод, что предусловия для восприятия выполнены. Теперь, если восприятие p имеет ровно одну схему восприятия, *Percept*(p, PRECOND: c), где c представляет собой конъюнкцию литералов, то эти литералы могут быть введены в доверительное состояние вместе с p. С другой стороны, если восприятие p имеет более одной схемы восприятия, для которых предусловия могут иметь место в соответствии с предсказанным доверительным состоянием $\hat{b}$, то нам потребуется добавить *дизъюнкцию* этих предусловий. Очевидно, что в результате доверительное состояние утратит форму 1-CNF и это приведет к тем же сложностям, что и в случае условных эффектов, с почти такими же классами решений.

Имея механизм для вычисления точных или приблизительных доверительных состояний, можно генерировать условные планы с использованием расширенного прямого И–ИЛИ-поиска в пространстве доверительных состояний, как это предлагалось в разделе 4.4. Действия с недетерминированными эффектами — которые просто определяются с помощью дизъюнкции в предложении EFFECT схемы действия — могут быть включены в расчеты обновленного доверительного состояния с минимальными изменениями и без необходимости внесения каких-либо изменений в алгоритм поиска.[3] В отношении эвристической функции многие из методов, предложенных для бессенсорного планирования, применимы и в частично наблюдаемых, недетерминированных задачах.

11.5.3. Непрерывное планирование

Представьте, что вы наблюдаете за роботом, выполняющим точечную сварку на сборочной линии автомобильного завода. Быстрые, точные движения робота повторяются вновь и вновь каждый раз, когда очередной автомобиль поступает к нему на обработку. Хотя технически это впечатляет, робот, вероятно, вовсе не покажется вам разумным, поскольку все его движения представляют собой фиксированную, заранее запрограммированную последовательность, — робот явно "не знает, что он делает" в любом смысле этих слов. Теперь предположим, что плохо прикрепленная дверь падает с автомобиля на пол, как только робот собирается применить к ней точечную сварку. Робот быстро заменяет свой сварочный привод захватом, поднимает дверцу, проверяет наличие царапин и снова прикрепляет ее к машине, затем отправляет электронное письмо начальнику линии, вновь подключает сварочный

[3] Если для недетерминированной задачи потребуются циклические решения, алгоритм И–ИЛИ-поиска следует обобщить до циклической версии, такой как LAO* (Хансен и Зильберштейн [960], 2001).

привод и возобновляет свою работу. Теперь поведение робота кажется *целенаправленным*, а не рутинным, и можно предположить, что это уже не часть обширного, заранее разработанного условного плана, а результат непосредственной, вызванной сложившейся ситуацией перепланировки процесса, что, в свою очередь, означает, что робот действительно *должен* знать, что он пытается сделать.

Перепланировка предполагает некоторую форму ▶**контроля выполнения** с целью обнаружения необходимости нового плана. Подобная потребность может возникнуть, когда агенту с условным планированием просто надоедает планировать свою реакцию на каждую небольшую непредвиденную ситуацию, например на то, что с неба ему на голову может что-нибудь упасть.[4] И это означает, что условный план остается в незавершенной форме. Так, некоторые ветви частично составленного условного плана могут просто заканчиваться указанием *Replan* (перепланировать), — если подобная ветвь плана будет достигнута в процессе его выполнения, агент вернется в режим планирования. Как уже отмечалось, решение о том, какую часть задачи следует решить заранее, а какую оставить для возможного перепланирования, включает поиск множества компромиссов между возможными событиями с различными затратами и вероятностью их возникновения. Никто не хочет сначала дождаться ситуации, когда его автомобиль откажет посреди пустыни Сахара, и только тогда задуматься над тем, хватит ли ему имеющегося запаса воды.

Перепланирование может потребоваться, если модель мира агента неверна. Модель для действия может иметь ▶**отсутствующее предусловие**, например агент может не знать, что для снятия крышки банки с краской часто может требоваться отвертка. Модель может иметь ▶**отсутствующий эффект** — при покраске объекта краска также может попасть на пол. Или же в модели может обнаружиться ▶**отсутствующая флюента**, которой просто нет в используемом представлении, например обсуждавшаяся выше модель задачи окраски мебели не содержит упоминаний о количестве краски в банке, о том, как действия в модели влияют на это количество, или о необходимости, чтобы это количество не было нулевым. В модели также может не хватать мер предосторожности в отношении ▶**внешних событий**, таких как ситуация, когда кто-то опрокинул банку с краской. Внешние события могут также включать изменения в цели, такие как добавление требования, чтобы стол и стул не были окрашены в черный цвет. Без способности контролировать и перепланировать поведение агента, вероятно, будет очень уязвимым, легко подвергающимся нарушениям, если он полностью полагается на абсолютную правильность своей модели.

---

[4] В 1954 году миссис Ходжес из Алабамы ударил метеорит, пробивший крышу ее дома. В 1992 году осколок метеорита Мбале попал в голову маленького мальчика, — к счастью, его полет затормозили банановые листья. (Дженнискенс и др. [1134], 1994). А в 2009 году немецкий мальчик утверждал, что ему в руку попал метеорит размером с горошину. В результате каждого из этих инцидентов никто не получил серьезных травм, так что можно предположить, что потребность в предварительном планировании защиты от подобных случайностей несколько преувеличена.

У непрерывно планирующего агента есть выбор из (как минимум) трех различных подходов к контролю за состоянием среды во время выполнения плана.

- ►**Контроль действий**. Перед выполнением действия агент проверяет, все ли его предусловия по-прежнему выполняются.
- ►**Контроль плана.** Перед выполнением действия агент проверяет выполнимость всей оставшейся части плана.
- ►**Контроль цели**. Перед выполнением действия агент проверяет наличие лучших вариантов целевых состояний, которых можно попытаться достичь.

На рис. 11.12 схематически представлено, как осуществляется контроль действий. Агент отслеживает как свой первоначальный план (*весь_план*), так и ту часть плана, которая еще не была выполнена (*план*). После выполнения нескольких первых этапов плана агент полагает, что должен быть в состоянии *E*, но обнаруживает, что на самом деле находится в состоянии *O*. В первоначальный план необходимо внести изменения — найти в нем некоторую точку *P*, в которую агент сможет вернуться из состояния *O*, и продолжить выполнение первоначального плана. (Вполне может оказаться, что искомая точка *P* — это и есть целевое состояние *G*.) При перепланировании агент стремится минимизировать общую стоимость нового плана: часть *исправление* (от *O* до *P*) плюс часть *продолжение* (от *P* до *G*).

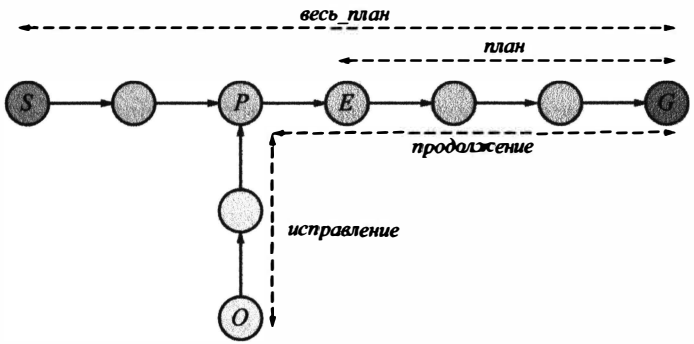


Рис. 11.12. В начале предполагается, что выполнение последовательности действий *весь_план* позволит агенту перейти из состояния *S* в состояние *G*. Агент последовательно выполняет этапы плана, пока не выясняется, что вместо состояния E, как он ожидал, в действительности он оказался в состоянии *O*. Далее агент перепланирует свои действия и вырабатывает поправку *исправление* для возвращения в состояние *P* первоначального плана, что позволит ему продолжить выполнение исходного плана (*продолжение*) и добраться до состояния G

А теперь вернемся к предыдущему примеру: к задаче покраски мебели, в которой нужно добиться, чтобы цвета стула и стола совпадали. Предположим, агент разработал следующий план:

```
[LookAt(Table), LookAt(Chair),
  if Color(Table, c) ∧ Color(Chair, c) then NoOp
    else [RemoveLid(Can1), LookAt(Can1),
        if Color(Table, c) ∧ Color(Can1, c) then Paint(Chair, Can1)
        else REPLAN]].
```

Теперь агент готов выполнить план. Агент обнаруживает, что стол и краска в банке белые, а стул — черный, поэтому далее он выполняет действие *Paint*(*Chair*, *Can1*). На данный момент классический планировщик объявил бы о победе — план был полностью выполнен. Но непрерывно планирующий агент, контролирующий действия, должен проверить результат и убедиться, что действие было выполнено *успешно*.

Предположим, агент видит, что стул получился не белый, а серый, с неровной пятнистой окраской, поскольку прежняя, черная, краска просвечивает сквозь новую, белую. Следовательно, агенту необходимо выяснить, в какую позицию в прежнем плане ему нужно перейти для исправления ситуации, и выбрать требуемую последовательность корректирующих действий, позволяющих туда попасть. Агент замечает, что по предусловиям текущее состояние идентично действию *Paint*(*Chair*, Can_1), и поэтому выбирает пустую последовательность *исправление* (см. рис. 11.12) и принимает как последовательность *продолжение* ту же последовательность [*Paint*], которую он только что выполнил. Приняв этот новый план, агент возобновляет контроль выполнения и предпринимает попытку выполнить действие *Paint*. Так он будет поступать до тех пор, пока стул не будет полностью окрашен в действительно белый цвет. Но обратите внимание: в данном случае цикл создается процессом "запланировать–выполнить–перепланировать", а не явным циклом в исходном плане. Также обратите внимание, что первоначальный план не должен покрывать все непредвиденные обстоятельства. Если агент достигает этапа, отмеченного как REPLAN, он может создать новый план (возможно, с использованием краски в банке Can_2).

Контроль действий — очень простой метод контроля выполнения, но иногда он может приводить к менее разумному поведению, чем хотелось бы. Например, предположим, что в двух банках нет ни черной, ни белой краски, но есть банка с красной краской, поэтому агент формирует план окраски стула и стола в красный цвет. Однако на самом деле этой краски хватает только для окраски стула. В случае *контроля действий* неудача будет обнаружена только после того, как стул уже будет окрашен, и выяснится, что для покраски стола краски недостаточно. Агенту потребуется выполнить перепланирование и подготовить новый план, например окраски стола и стула в зеленый цвет, воспользовавшись краской из второй банки. Решить проблему позволяет использование *контроля плана* — нарушение

будет обнаружено сразу же, как только выяснится, что текущее состояние уже не позволяет успешно завершить план. Таким образом, агент, открыв первую банку, сразу смог бы понять, что не стоит тратить время на окраску стула в красный цвет.

В методе контроля плана эта цель достигается путем проверки предусловий успеха всего оставшегося плана, т.е. предусловий каждого этапа в плане, за исключением тех из них, которые достигаются с помощью другого этапа в оставшемся плане. При использовании контроля плана выполнение потенциально невыполнимого плана прекращается настолько быстро, насколько это возможно, вместо того чтобы следовать ему, пока отказ фактически не будет обнаружен.[5] Метод контроля плана также позволяет учитывать **случайные открытия** — случайный успех. Если кто-то придет на помощь агенту и покрасит стол в красный цвет, пока он красит в красный цвет стул, то предусловия завершения плана будут выполнены (т.е. цель достигнута) и агент сможет раньше завершить работу.

Относительно несложно модифицировать алгоритм планирования таким образом, чтобы каждое действие в плане помечалось его предусловиями, что позволит реализовать контроль действий. Несколько сложнее будет реализовать контроль плана. Планировщики с частичным упорядочиванием обладают такими преимуществами, поскольку в них уже встроены структуры, поддерживающие связи, необходимые для контроля плана. Расширение пространства состояний планировщиков необходимыми пометками может быть реализовано посредством тщательной регистрации того, как флюенты цели проходят в плане при просмотре в обратном направлении.

Теперь, после описания метода контроля и перепланирования, необходимо ответить на вопрос "Будет ли он работать?" Ответ на этот вопрос на удивление сложен. Если иметь в виду "Можем ли мы гарантировать, что агент всегда будет достигать цели?", то ответом будет "Нет", поскольку агент может непреднамеренно попасть в тупиковую ситуацию, из которой у него не будет возможности выбраться. Например, агент-пылесос может использовать ошибочную модель самого себя, не учитывающую, что его аккумуляторы способны разрядиться. Если это произойдет, он не сможет подготовить какой-либо план исправления такой ситуации. Если исключить из рассмотрения тупиковые ситуации, предположив, что агент может сформировать план достижения цели из *любого* состояния в своей среде, и предположив, что эта среда действительно является недетерминированной — в том смысле, что подобный план всегда имеет *какой-то* шанс на успех в любой конкретной попытке его выполнения, — то агент в конечном итоге достигнет цели.

Неприятности возникают лишь тогда, когда, казалось бы, недетерминированное действие на самом деле не является случайным, а скорее зависит от некоторого

[5] Используя контроль плана, наш агент, наконец-то, становится умнее навозного жука (см. раздел 2.2.3). Теперь наш агент заметил бы, что шарик навоза в его лапках отсутствует, и поэтому перепланировал бы свои действия, чтобы получить еще один шарик навоза и заткнуть им свою норку.

предварительного условия, о котором агент не знает. Например, иногда банка с краской может оказаться пустой, поэтому окраска с ее использованием не даст никакого эффекта. И любое количество повторных попыток ничего в этом не изменит.[6] Одно из решений состоит в случайном выборе одного из множества возможных планов исправления ситуации вместо попыток каждый раз выполнить один и тот же план. В данном случае спасти ситуацию может план исправления, состоящий в открытии другой банки с краской. Лучшим подходом является **обучение** лучшей модели. Каждая ошибка в предсказании — это возможность для обучения, и агент должен быть способен модифицировать свою модель мира в соответствии со своим восприятием. С этого момента перепланировщик получит возможность строить планы исправления, исходя из сути проблемы, а не полагаясь на удачу при выборе подходящего варианта.

11.6. Время, расписания и ресурсы

Классическое планирование говорит о том, *что делать* и *в каком порядке*, но ничего не говорит о времени: *как долго* действие будет выполняться и *когда* оно будет происходить. Например, в проблемной области функционирования аэропорта оно позволяет разработать план с указанием, какой самолет куда летит и что перевозит, но никак не определяет время вылета и время прилета. Это предмет анализа в теории ▶**составления расписаний**.

Реальный мир также налагает ▶**ограничения на ресурсы**: авиакомпания имеет конкретное число сотрудников, и те, которые выполняют один рейс, не могут одновременно принимать участие в выполнении другого. В этом разделе представлены методы планирования и составления расписаний для задач с ограничениями на ресурсы.

Используемый здесь подход состоит в том, чтобы "сначала планировать, а затем составлять расписание". То есть общий процесс решения задачи делится на этап *планирования*, на котором определяются действия, необходимые для достижения поставленных целей с учетом некоторых ограничений на порядок их выполнения, и этап *составления расписания*, на котором к плану добавляется временная информация, необходимая для получения гарантий его соответствия ресурсам и установленным срокам. Подобный подход обычно используется в реальных производственных и логистических организациях, где этап планирования иногда автоматизирован, а иногда выполняется специалистами-людьми.

[6] Бесполезное повторение одного и того же способа исправления плана полностью повторяет поведение, которое демонстрирует оса-сфекс (см. раздел 2.2.3).

11.6.1. Представление временных и ресурсных ограничений

Типичная задача ▶**планирования производства** (см. раздел 6.1.2) включает множество ▶**операций**, каждая из которых имеет набор **действий** с некоторыми требованиями к порядку их выполнения. Каждое действие имеет ▶**продолжительность** и множество ограничений на ресурсы, необходимых для его выполнения. Ограничение определяет тип ресурса (например, болты, гаечные ключи или пилоты), требуемое количество этого ресурса и является ли данный ресурс ▶**расходным материалом** (например, болты будут уже недоступны для дальнейшего использования) или он может ▶**использоваться повторно** (например, пилот занят во время полета, но снова будет доступен по его окончании). Действия также могут иметь своим результатом ресурсы (например, производственные действия и действия по пополнению запасов).

Решение задачи планирования производства определяет время начала для каждого действия и должно удовлетворять все требования к порядку их выполнения и все ограничения на ресурсы. Как и в случае задач поиска и планирования, решения могут быть оценены в соответствии с функцией стоимости, но в данном случае это может оказаться довольно сложно: возможны нелинейные затраты на ресурсы либо зависящие от времени потери из-за задержек опоздания и т.д. Для простоты предположим, что функция затрат — это только общая продолжительность плана, которую принято называть ▶**временем выполнения** (*makespan*).

На рис. 11.13 приведен простой пример: предельно упрощенная задача сборки двух автомобилей. В ней имеется две операции и каждая состоит из трех действий (*job*): [*AddEngine*, *AddWheels*, *Inspect*]. В высказывании *Resources* объявляется, что в задаче используется четыре типа ресурсов и для каждого из них указывается их количество, доступное в начале работ: один подъемник для двигателя, одна станция установки колес, два инспектора и 500 гаек. В схемах действий определяется продолжительность выполнения и потребность в ресурсах каждого действия. Гайки являются *расходным материалом*, используемым при установке колес, а все остальные ресурсы "привлекаются" в начале выполнения действия и освобождаются после его завершения.

Представление ресурсов в виде числовых величин, например *Inspectors*(2), а не в качестве именованных сущностей, например *Inspector*(I_1) и *Inspector*(I_2), является примером метода, называемого ▶**агрегированием**: группировка отдельных объектов в категории, в которой все отдельные объекты неразличимы. В нашей задаче сборки не имеет никакого значения, *какой* из инспекторов проверит машину, поэтому здесь нет необходимости делать между ними различие. Агрегирование очень важно для уменьшения сложности задачи. Рассмотрим, что произойдет, если рассматриваемый график работ включает 10 одновременно выполняемых действий *Inspect* при наличии только 9 инспекторов. Если инспекторы будут представлены в задаче как категория, отказ обнаружится немедленно, и алгоритм совершит возврат для поиска другого варианта графика. Если каждый инспектор будет представлен индиви-

дуально, алгоритм попробует использовать все 9! способов назначения инспекторов на отдельные действия, прежде чем обнаружит, что ни один из них не работает.

Jobs({*AddEngine1* ≺ *AddWheels1* ≺ *Inspect1*},
{*AddEngine2* ≺ *AddWheels2* ≺ *Inspect2*})

Resources(*EngineHoists*(1), *WheelStations*(1), *Inspectors*(2), *LugNuts*(500))

Action(*AddEngine1*, DURATION: 30,
USE: *EngineHoists*(1))
Action(*AddEngine2*, DURATION: 60,
USE: *EngineHoists*(1))
Action(*AddWheels1*, DURATION: 30,
CONSUME: *LugNuts*(20), USE: *WheelStations*(1))
Action(*AddWheels2*, DURATION: 15,
CONSUME: *LugNuts*(20), USE: *WheelStations*(1))
Action(*Inspect*$_i$, DURATION: 10,
USE: *Inspectors*(1))

Рис. 11.13. Задача планирования работ по сборке двух автомобилей при ограниченных ресурсах. Обозначение $A \prec B$ означает, что действие A должно предшествовать действию B

11.6.2. Решение задач составления расписаний

Начнем с рассмотрения задачи только временного планирования, отложив на время вопрос о возможных ресурсных ограничениях. Чтобы свести к минимуму время выполнения плана (его продолжительность), необходимо найти самое раннее время начала выполнения для всех его действий в соответствии с ограничениями в отношении порядка их выполнения, имеющимися в задаче. Удобно представить эти ограничения в виде направленного графа, отображающего связи между действиями, как показано на рис. 11.14. Чтобы определить возможное время начала и завершения каждого действия, можно воспользоваться ▶**методом критического пути** (*Critical Path Method* — CPM), применив его к графу. **Путем** через этот граф, представляющий план с частичным упорядочиванием, называется линейно упорядоченная последовательность действий, начинающаяся в момент времени *Start* и заканчивающаяся в момент времени *Finish*. (Например, в плане с частичным упорядочиванием на рис. 11.14 имеется два пути.)

Путь, имеющий максимальную суммарную продолжительность, называется ▶**критическим путем**, поскольку именно он определяет продолжительность выполнения всего плана. Сокращение продолжительности других путей не приведет к сокращению времени выполнения всего плана в целом, но задержка

начала любого действия в критическом пути замедлит осуществление всего плана. У действий вне критического пути есть определенный запас времени — временное окно, в течение которого они могут быть выполнены. Это окно задается в терминах самого раннего возможного времени начала выполнения, *ES*, и самого позднего возможного времени начала выполнения, *LS*. Разность *LS* – *ES* называется ▶**резервом времени** для действия. На рис. 11.14 видно, что для выполнения всего плана потребуется 85 минут, что каждое действие в верхнем задании имеет пятнадцатиминутный резерв времени, в пределах которого оно может быть начато, и что каждое действие на критическом пути не имеет резерва времени (по определению). Значения времени *ES* и *LS* для всех действий, взятые вместе, составляют ▶**расписание**, представляющее собой решение данной задачи.

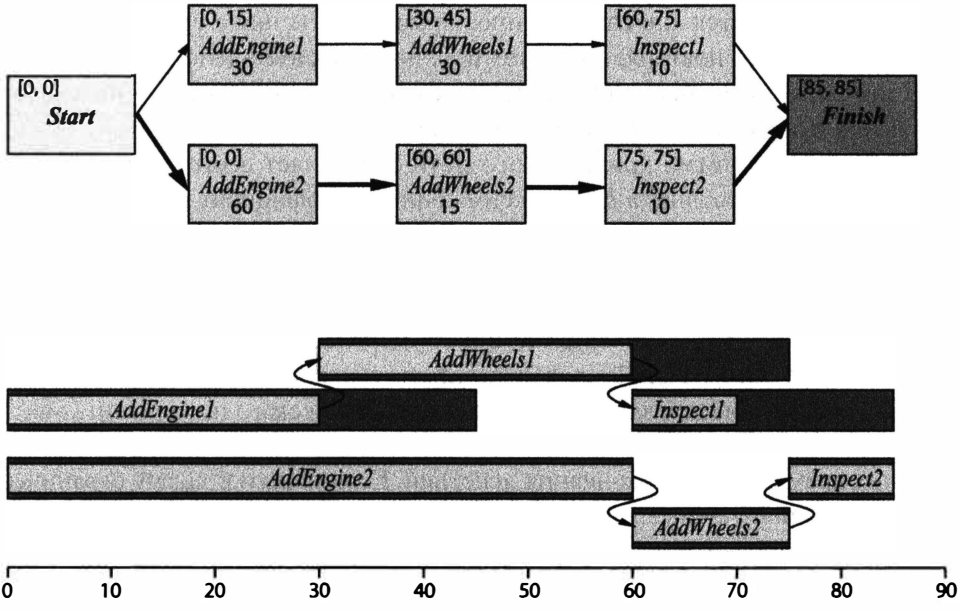


Рис. 11.14. *Вверху*: представление временных ограничений для задачи планирования работ по сборке двух автомобилей, приведенной на рис. 11.13. Продолжительность каждого действия указана в нижней части прямоугольника. При решении этой задачи вычисляются самое раннее и самое позднее значения возможного времени начала выполнения действия, представленные в виде пары [*ES*, *LS*] в верхнем левом углу прямоугольника. Разность между этими двумя числами определяет *резерв* времени выполнения для данного действия. Действия с нулевым резервом находятся на критическом пути, показанном на рисунке жирными стрелками. *Внизу*: то же самое решение показано относительно временной шкалы. Темно-серые прямоугольники представляют собой временные интервалы, в течение которых действие может быть выполнено при соблюдении ограничений их упорядочивания. Свободная часть каждого серого прямоугольника определяет резерв времени данного действия

Приведенные ниже формулы определяют значения *ES* и *LS*, они составляют основу алгоритма динамического программирования, применяемого для их вычисления.

$$ES(Start) = 0$$
$$ES(B) = \max_{A \prec B} ES(A) + Duration(A)$$
$$LS(Finish) = ES(Finish)$$
$$LS(A) = \min_{B \succ A} LS(B) - Duration(A)$$

Идея состоит в том, что вначале терму *ES*(*Start*) присваивается значение 0. Затем, как только будет выявлено действие *B*, такое, что все действия, непосредственно предшествующие B, уже имеют присвоенные значения *ES*, терму *ES*(*B*) присваивается максимальное из самых ранних значений времени завершения этих непосредственно предшествующих действий, где самое раннее время завершения действия определяется как самое раннее время начала плюс его продолжительность. Этот процесс повторяется до тех пор, пока каждому действию не присваивается значение *ES*. Значения *LS* вычисляются аналогичным образом, но в ходе передвижения в обратном направлении от действия *Finish*.

Сложность алгоритма критического пути составляет всего лишь $O(Nb)$, где N — количество действий, а b — максимальный коэффициент ветвления на входе или выходе любого действия. (Чтобы убедиться в этом, достаточно отметить, что вычисления значений *LS* и *ES* осуществляются по одному разу для каждого действия, а в каждом вычислении происходит итерация самое большее по b других действий.) Поэтому, если дано частичное упорядочение действий, задача поиска расписания с минимальной продолжительностью решается достаточно просто.

Говоря математически, задачи определения критического пути решить так легко потому, что они представляются в виде *конъюнкции линейных* неравенств, определяющих время начала и окончания действий. Если ввести в задачу ограничения на ресурсы, то полученные в результате определения времени начала и окончания окажутся более сложными. Например, действия *AddEngine*, которые на рис. 11.14 начинаются одновременно, требуют использования одного и того же ресурса *EngineHoist* и поэтому не могут выполняться одновременно, или, как говорят, *перекрываться*. Здесь ограничение "не перекрываются" — это *дизъюнкция* двух линейных неравенств, по одному для каждого возможного упорядочения действий. Введение дизъюнкций делает задачу составления расписаний с учетом ограничений на ресурсы NP-сложной.

На рис. 11.15 показано решение с самым коротким временем завершения — 115 минут. Это на 30 минут больше, чем 85 минут, необходимых для расписания без ограничений на ресурсы. Обратите внимание, что в этом расписании нет ни одного момента времени, когда требуется работа одновременно двух инспекторов, поэтому можно немедленно перевести одного из двух предлагаемых инспекторов на другую площадку, где его работа будет более продуктивной.

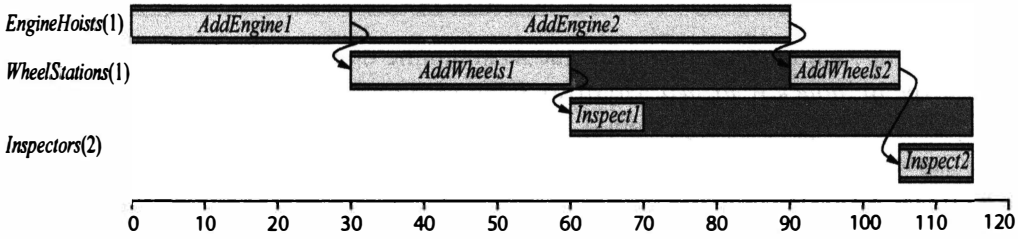


Рис. 11.15. Решение задачи планирования работ по сборке двух автомобилей, приведенной на рис. 11.13, с учетом ограничений на ресурсы. Слева перечислены три ресурса, а действия выровнены по горизонтали с теми ресурсами, которые в них используются. Существуют два возможных расписания, зависящих от того, при сборке какого из автомобилей подъемник для двигателя используется первым. Здесь показано оптимальное решение продолжительностью 115 минут

Существует долгая история работ в области оптимального составления расписаний. Основная задача была поставлена в 1963 году — найти оптимальное расписание для задачи, включающей всего 10 машин и 10 операций по 100 действий — и оставалась нерешенной в течение 23 лет (Лоулер и др. [1365], 1993). Было испытано много подходов, включая методы ветвей и границ, имитации отжига, поиска с запретами и удовлетворения ограничений. Одним из популярных подходов является эвристика ▶ **минимального резерва** времени: на каждой итерации планируется как можно более ранний запуск любого из незапланированных действий со всеми уже запланированными предшественниками, имеющего наименьшую задержку. Затем обновляются значения времени ES и LS для каждого затронутого действия — и цикл повторяется. Эта жадная эвристика напоминает эвристику минимальных оставшихся значений (MRV) в методах удовлетворения ограничений. На практике она часто работает хорошо, но для нашей задачи планирования работ по сборке автомобилей она дает 130-минутное решение, а не 115-минутное, показанное на рис. 11.15.

До этого момента предполагалось, что множество действий и ограничения упорядоченности фиксированы. При таких допущениях каждая задача составления расписания может быть решена с помощью неперекрывающейся последовательности действий, в которой избегаются все конфликты ресурсов при условии, что каждое действие осуществимо само по себе. Однако, если задача составления расписания доказуемо очень сложна, то может оказаться не слишком хорошей идеей решать ее именно таким способом, — может быть лучше пересмотреть действия и ограничения в плане, что иногда приводит к получению гораздо более простой задачи составления расписания. Следовательно, имеет смысл *объединить* планирование и составление расписания, приняв во внимание продолжительность и возможности перекрытия действий при построении плана. Некоторые алгоритмы планирования из раздела 11.2 могут быть расширены с целью обработки подобной информации.

11.7. Анализ различных подходов к планированию

В планировании объединяются две важнейшие области ИИ, которые мы уже рассмотрели: *поиск* и *логика*. Поэтому планировщик может рассматриваться как программа, которая ищет решение, или как программа, которая (конструктивно) доказывает существование решения. Перекрестное обогащение идеями из этих двух областей позволило программам-планировщикам масштабироваться от игрушечных проблем, в которых количество действий и состояний было ограничено примерно дюжиной, до реальных промышленных приложений с миллионами состояний и тысячами действий.

Планирование — это, прежде всего, приложение усилий в борьбе с комбинаторным взрывом. Если в проблемной области имеется n примитивных высказываний, то количество состояний становится равным 2^n. В борьбе с этой печальной ситуацией мощным оружием может оказаться выявление в задаче независимых подзадач. В лучшем случае — при полной декомпозиции задачи — достигается экспоненциальное ускорение. Однако возможность декомпозиции разрушается при наличии негативных взаимодействий между действиями. Решатель SATPLAN позволяет закодировать логические отношения между подзадачами. Применение прямого поиска предполагает решение задачи с использованием эвристик в попытках отыскать шаблоны (подмножества высказываний), которые покрывают независимые подзадачи. Поскольку этот подход эвристический, он может работать, даже если подзадачи не являются полностью независимыми.

К сожалению, у нас пока нет четкого понимания, какие методы лучше всего подходят для решения тех или иных типов задач. Вполне вероятно, что появятся новые методы, возможно, обеспечивающие синтез высоковыразительных представлений первого порядка и иерархических представлений с высокоэффективными развернутыми и пропозициональными представлениями, которые доминируют сегодня. Уже имеются примеры ▶ **портфолио** алгоритмов систем планирования, набор алгоритмов которых может быть применен к любой возникшей задаче. Это может быть сделано выборочно (система классифицирует каждую новую задачу, выбирая для нее наилучший алгоритм), параллельно (все алгоритмы будут работать одновременно, каждый на собственном ЦПУ) или с чередованием алгоритмов в соответствии с некоторым графиком.

Резюме

В этой главе описано представление на языке PDDL как классических, так и расширенных задач планирования и предложено несколько алгоритмических подходов поиска их решений. Основные идеи, изложенные в этой главе, перечислены ниже.

- Системы планирования строятся на применении алгоритмов решения задач, работающих с развернутым представлением состояний и действий. Такие представления обеспечивают возможность вывода эффективных эвристик, не зависящих от проблемной области, а также разработки мощных и гибких алгоритмов для решения задач.
- В языке PDDL (*Planning Domain Definition Language* — язык определения проблемной области планирования) исходные и целевые состояния описываются как конъюнкции литералов, а действия представляются в терминах их предусловий и результирующих эффектов. Расширения языка обеспечивают представление в планах времени, ресурсов, восприятия, условных и иерархических планов.
- Поиск в пространстве состояний может выполняться в **прямом** направлении (прогрессия) или в **обратном** направлении (регрессия). Эффективные эвристики могут быть выведены за счет принятия предположения о независимости подцелей, а также с использованием различных ослаблений задачи планирования.
- Другие подходы включают кодирование задачи планирования как задачи булевой выполнимости (SAT) или как задачи удовлетворения ограничений и выполнение явного поиска в пространстве планов с частичным упорядочиванием.
- Планирование с помощью **иерархической сети задач** (*Hierarchical Task Network* — HTN) позволяет агенту получить совет от проектировщика проблемной области в форме **действий высокого уровня** (*High-Level Actions*, HLA), которые могут быть реализованы различными способами в виде последовательности примитивных действий. Описание эффектов HLA может быть выполнено с использованием **ангельской семантики**, позволяющей вывести доказуемо корректные планы высокого уровня без рассмотрения их низкоуровневой реализации. Метод HTN позволяет создавать очень большие планы, необходимые для многих реальных приложений.
- **Условные планы** позволяют агенту во время их выполнения получать информацию о мире с помощью датчиков и использовать ее для определения той ветви плана, по которой он будет двигаться далее. В некоторых случаях также может использоваться **бессенсорное планирование** или **конформное планирование** для формирования плана, который в процессе выполнения не требует получения результатов восприятия. Как конформные планы, так

и условные планы могут быть выработаны посредством поиска в пространстве **доверительных состояний**. Эффективное представление или вычисление доверительных состояний является в этой области ключевой проблемой.

- **Непрерывно планирующий агент** использует контроль выполнения и, если требуется, осуществляет корректирующие действия с целью возврата к первоначальному плану, выполнение которого было нарушено непредвиденной ситуацией, возникшей по причине недетерминированности действий, внешних событий или неточностей в его модели окружающей среды.
- Во многих действиях потребляются **ресурсы**, такие как деньги, топливо или сырье. Эти ресурсы удобнее рассматривать в целом, как числовые величины, и не пытаться рассуждать, скажем, о каждой отдельной монете или купюре во всем мире. Время является одним из самых важных ресурсов. За его расходованием можно следить, применяя специализированные алгоритмы составления расписаний или объединяя составление расписаний с планированием.
- В этой главе классическое планирование расширяется для охвата недетерминированных сред (где результаты действий являются неопределенными), но это не последнее слово в планировании. В главе 17 описываются методы для стохастических сред (в которых результаты действий имеют определенную связанную с ними вероятность): марковские процессы принятия решений, частично наблюдаемые марковские процессы принятия решений и теория игр. В главе 22 показано, что обучение с подкреплением дает агенту возможность учиться правильному поведению на основании своих прошлых успехов и неудач.

Библиографические и исторические заметки

Исследования в области планирования в теории ИИ явились результатом исследований в областях выполнения поиска в пространстве состояний, доказательства теорем и теории управления. Первой крупной системой планирования стала система STRIPS (Фикес и Нильссон [738], 1971; [739], 1993), разработанная как планировщик для робота Shakey в Станфордском исследовательском институте. Первая версия программы работала на компьютере с объемом памяти всего 192 Кбайт. Ее общая структура управления была создана на основе программы GPS (*General Problem Solver* — общий решатель задач, Ньюэлл и Саймон [1668], 1961) — системы поиска в пространстве состояний, использующей анализ целей и средств (*means–ends analysis*).

Со временем язык представления системы STRIPS эволюционировал в язык ADL (*Action Description Language* — язык описания действий (Педнаулт [1762], 1986)), а затем — в язык PDDL (*Problem Domain Description Language* — язык описания проблемной области задачи (Халлаб и др. [846], 1998)), который используется в

Международном конкурсе планирования AIPS с 1998 года. Последняя версия этого языка — PDDL 3.1 (Ковач [1294], 2011).

Планировщики в ранних 1970-х выполняли декомпозицию задач посредством вычисления подпланов для каждой подцели с последующей увязкой субпланов в единую цепочку в некотором порядке. Для этого подхода, названного Сакердоти ([1955], 1975) ▶**линейным планированием**, вскоре было обнаружено, что он является неполным. Он не позволяет решать некоторые очень простые задачи, такие как аномалия Зюссмана (см. упражнение 11.8), обнаруженная Алленом Брауном во время экспериментов с системой HACKER (Зюссман [2149], 1975). Полный планировщик должен обеспечивать **чередование** действий из различных субпланов в пределах единой последовательности. Система WARPLAN Уоррена ([2296], 1974) достигла этой цели и продемонстрировала, что язык логического программирования Prolog позволяет создавать исключительно краткие программы: WARPLAN — это всего 100 строк кода.

В течение следующих 20 лет исследований доминировало планирование с частичным упорядочиванием, — после публикации теоретических работ, описывающих обнаружение конфликтов (Тейт [2179], 1975) и защиту достигнутых условий от вмешательства (Зюссман [2149], 1975), а также появления реализации, включая NOAH (Сакердоти [1956], 1977) и NONLIN (Тейт [2181], 1977). Это привело к появлению формальных моделей (Чепмен [392], 1987; Макаллистер и Розенблитт [1524], 1991), позволяющих проводить теоретический анализ различных алгоритмов и задач планирования, а также к появлению широко распространенной системы UCPOP (Пенберти и Вельд [1770], 1992).

Позже Дрю МакДермотт заявил о своем подозрении, что акцент на планировании с частичным упорядочиванием вытеснил другие методы, которые, возможно, следует пересмотреть в настоящее время, когда компьютеры получили в 100 раз больше памяти, чем во времена Shakey. Его программа UNPOP (МакДермотт [1544], 1996) являлась планировщиком в пространстве состояний, использующем эвристику с пустым списком удаления. Планировщик HSP (*Heuristic Search Planner* — планировщик с эвристическим поиском, Бонет и Геффнер [247], 1999; Хаслум [979], 2006) сделал поиск в пространстве состояний практичным и для больших задач планирования. Планировщик FF или Fast Forward (Хоффманн [1041], 2001; Хоффманн и Небель [1045], 2001; Хоффманн [1044], 2005) и его вариант FASTDOWNWARD (Хельмерт [1007], 2006) выигрывали международные соревнования по планированию в 2000-х годах.

Двунаправленный поиск (см. раздел 3.4.5), как известно, также страдает от недостатка эвристик, но некоторые успехи уже были достигнуты при использовании обратного поиска для создания **периметра** вокруг цели, а затем — уточнения эвристики для прямого поиска в направлении к этому периметру (Торральба и др. [2223], 2016). Планировщик двунаправленного поиска SYMBA* победил в конкурсе 2016 года (Торральба и др. [2223], 2016).

В первом десятилетии XXI века исследователи обратились к языку PDDL и парадигме планирования в поисках эвристик, которые можно было бы использовать независимо от проблемной области. Хоффманн ([1044], 2005) анализирует пространство поиска эвристики с пустым списком удаления. Эделькамп ([675], 2009) и Хаслум и соавт. ([977], 2007) описывают, как строить базы данных шаблонов для планирования эвристик. Фельнер и соавт. ([722], 2004) приводят обнадеживающие результаты использования базы данных шаблонов для решения головоломок со скользящими плитками, которые также можно рассматривать и как область планирования, но Хоффманн и соавт. ([1046], 2006) указывают на некоторые ограничения в этой абстракции относительно задач классического планирования. Ринтанен в работе [1887] (2012) обсуждает специфическую для задач планирования эвристику выбора переменных для решателей SAT.

Хелмерт и соавт. ([1009], 2011) описывают систему Fast Downward Stone Soup (FDSS), планировщик с портфолио алгоритмов, который, как в сказке о супе из камня, приглашает нас бросить в него столько алгоритмов планирования, сколько возможно. Система поддерживает множество учебных задач и для каждой задачи и каждого алгоритма записывает время выполнения и итоговую смету затрат на ее решение. Затем, столкнувшись с новой задачей, система использует свой прошлый опыт, чтобы принять решение, какой (или какие) из алгоритмов к ней применить, чтобы при заданных ограничениях на время выполнения попытаться найти решение с минимальными издержками. Система FDSS победила в международных соревнованиях по планированию в 2018 году (Зип и Рёгер [2021], 2018). В работе Зипа и соавт. [2022] (2015) описывается подход к машинному обучению с целью автоматического обучения хорошему портфолио алгоритмов для конкретной новой задачи. Валлати и соавт. ([2245], 2015) дают обзор портфолио алгоритмов планирования. Идея портфолио алгоритмов для задач комбинаторного поиска восходит к работе Гомеса и Сельмана [888] (2001).

Систла и Годенфроид в работе [2079] (2004) рассматривают снижение симметрии, а Годенфроид в [872] (1990) — эвристику для планирования с частичным упорядочиванием. Рихтер и Хельмерт в работе [1879] (2009) демонстрируют прирост эффективности прямого отсечения при использовании предпочтительных действий.

Блум и Фюрст ([232], 1997) возродили область планирования с помощью своей системы Graphplan, которая работала на несколько порядков быстрее, чем планировщики с частичным упорядочиванием того времени. Брайс и Камбхампати ([328], 2007) дают обзор графов планирования. Использование для планирования исчисления ситуаций было введено Джоном Маккарти ([1530], 1963) и уточнено Рэем Рейтером ([1871], 2001).

Кац и соавт. ([1200], 1996) исследовали различные способы пропозиционализации схем действий и установили, что большинство компактных форм необязательно приводит к сокращению поиска решения. Систематический анализ был проведен Эрнстом и соавт. ([693], 1997), который также разработал автоматический

"компилятор", генерирующий пропозициональные представления для задач на языке PDDL. Планировщик BLACKBOX, сочетающий в себе идеи из решателей Graphplan и SATPLAN, был разработан Каутцем и Селманом ([1202], 1998). Планировщики, основанные на методах удовлетворения ограничений, включают CPLAN ван Бика и Чена ([2247], 1999) и GP-CSP (До и Камбхампати [626], 2003).

Был также отмечен интерес к представлению плана в виде ▶**бинарной диаграммы решений (BDD)**, компактной структуры данных для булевых выражений, широко изучаемой в сообществе верификации оборудования (Кларк и Грумберг [450], 1987; Макмиллан [1553], 1993). Существуют методы доказательства свойства из бинарных диаграмм решений, в том числе для свойства быть решением задачи планирования. Циматти и соавт. в работе [441] (1998) представляют планировщик, основанный на этом подходе. Также использовались и другие представления, такие как целочисленное программирование (Фоссен и др. [2284], 2001).

Имеется несколько интересных сравнений различных подходов к планированию. Элмерт ([1006], 2001) анализирует несколько классов задач планирования и показывает, что подходы на основе ограничений, такие как Graphplan и SATPLAN, являются лучшими для NP-трудных проблемных областей, в то время как подходы на основе поиска лучше проявляют себя в проблемных областях, где возможные решения могут быть найдены без использования поиска с возвратами. Решатели Graphplan и SATPLAN сталкиваются с затруднениями в проблемных областях с большим количеством объектов, поскольку это означает, что они должны создавать множество действий. В некоторых случаях возникновение этой проблемы можно отложить или обойти за счет генерации пропозиционализированных действий динамически, по мере необходимости, вместо создания сразу всех их экземпляров перед началом поиска.

Первым механизмом иерархического планирования была функция программы STRIPS для изучения ▶**макрооператоров** (*macrop*), состоящих из последовательности примитивных этапов (Файкс и др. [737], 1972). В системе ABSTRIPS (Сакердоти [1954], 1974) было введено представление об ▶**иерархии абстракции**, благодаря чему планированию на более высоких уровнях было разрешено игнорировать низкоуровневые предпосылки действий при выводе общей структуры рабочего плана. В докторской диссертации Остина Тейта ([2180], 1975) и работе Эрла Сакердоти [1956] (1977) были развиты основные идеи планирования HTN. В работах Эрола, Хендлера и Нау [694] (1994) и [695] (1996) представлены полный иерархический декомпозирующий планировщик, а также диапазон сложности результатов для чистых HTN-планировщиков. Метод представления HLA и ангельской семантики, использованный в этой книге, принадлежит Марти и соавт. ([1506], 2007; [1507], 2008).

Одна из целей иерархического планирования состоит в повторном использовании опыта предыдущего планирования, представленного в форме обобщенных планов. Метод ▶**обучения, основанного на объяснениях** (*explanation-based learning*), использовался в качестве средства обобщения ранее вычисленных планов в таких

системах, как SOAR (Лейрд и др. [1340], 1986) и PRODIGY (Карбонелл и др. [368], 1989). Альтернативный подход заключается в сохранении ранее вычисленных планов в их первоначальном виде и последующем повторном их использовании для решения новых, сходных проблем по аналогии с исходной задачей. Этот подход принят в области под названием ▶ **планирование по прецедентам** (*case-based planning*, Карбонелл [368], 1983; Алтерман [38], 1988), и его сторонники полагают, что планирование по прецедентам должно рассматриваться как форма уточнения планирования, и предоставляют формальное обоснование для планирования с частичным упорядочиванием по прецедентам (Камбхампати [1176], 1994).

В ранних планировщиках отсутствовали условные выражения и циклы, но в некоторых могло использоваться принудительное формирование конформных планов. В программе NOAH, разработанной Сакердоти, такой принудительный подход использовался для решения задачи о "ключах и ящиках" (задача, в которой планировщик имеет мало информации о начальном состоянии). Мэйсон ([1511], 1993) показал, что в планировании поведения роботов часто можно и нужно отказаться от сбора информации с помощью датчиков, и описал бессенсорный план, позволяющий переместить инструмент в заданную позицию на столе посредством последовательности раскачивающих действий, независимой от начальной позиции.

Голдмен и Бодди ([882], 1996) ввели термин **конформное планирование** (*conformant planning*), отметив, что бессенсорные планы часто бывают эффективнее, даже если агент имеет датчики. Первым достаточно эффективным конформным планировщиком была программа Conformant Graphplan (CGP) Смита и Уэлда ([2092], 1998). Феррарис и Гуинкилья ([734], 2000), а также Ринтанен ([1885], 1999) независимо разработали конформные планировщики на основе алгоритма SATPLAN. Боне и Геффнер в работе [248] (2000) описали конформный планировщик, основанный на эвристическом поиске в пространстве доверительных состояний, реализованный на идеях, впервые разработанных в 1960-х годах для частично наблюдаемых марковских процессов принятия решения, или POMDP (*Partially Observable Markov Decision Processes*) (см. главу 17).

В настоящее время существует три основных подхода к конформному планированию. В первых двух используется эвристический поиск в пространстве доверительных состояний: в системе HSCP (Бертоли и др. [197], 2001) для представления доверительных состояний используются двоичные диаграммы решений (Binary Decision Diagram — BDD), тогда как Хоффманн и Брафмен ([1042], 2006) приняли ленивый подход вычисления предусловий и проверки целей по требованию с использованием решателя SAT.

В третьем подходе, который отстаивает главным образом Юсси Ринтанен ([1886], 2007), вся задача бессенсорного планирования формулируется в виде булевой формулы с кванторами (QBF) и решается с помощью QBF-решателя общего назначения. Современные конформные планировщики работают на пять порядков быстрее, чем программа CGP. Победителем в области конформного планирования на Международном конкурсе по планированию 2006 года стала система T_0

(Палакиос и Геффнер [1724], 2007), использующая эвристический поиск в пространстве доверительных состояний и сохраняющая представление доверительных состояний простым за счет определения производных литералов, покрывающих эффекты условий. Брайс и Камбхампати в работе [328] (2007) обсуждают, как граф планирования может быть обобщен для создания хороших эвристик для конформного и условного планирования.

Подход к условному планированию, используемый в этой главе, основан на идеях из статьи Хоффманна и Брафмана [1043] (2005) с определенным влиянием эффективных алгоритмов поиска для циклических И–ИЛИ-графов, разработанных Хименесом и Торрасом ([1138], 2000), а также Хансеном и Зильберштейном ([960], 2001). Проблема условного планирования получила больше внимания после публикации влиятельной статьи Дрю Мак-Дермотта *Планирование и действие* ([1541], 1978). Бертоли и соавт. ([198], 2001) описывают планировщик MBP (*Model-Based Planner*), использующий двоичные диаграммы решений для конформного и условного планирования. Одни авторы используют термины "обусловленное планирование" (*conditional planning*) и "условное планирование" (*contingent planning*) как синонимы, тогда как другие считают их различными, полагая, что "обусловленное" (conditional) относится к действиям с недетерминированными эффектами, тогда как "условное" (contingent) означает использование датчиков для преодоления частичной наблюдаемости.

Глядя в ретроспективе, можно видеть, как появление основных классических алгоритмов планирования привело к созданию расширенных версий этих алгоритмов для проблемных областей, связанных с учетом неопределенности. Разработка методов быстрого прямого эвристического поиска привела к созданию методов поиска в пространстве доверительных состояний (Боне и Геффнер [248], 2000; Хоффманн и Брафмен [1043], 2005). Накопленный опыт использования алгоритма SATPLAN привел к созданию стохастического алгоритма SATPLAN (Майерчик и Литтман [1478], 2003) и к разработке методов планирования с применением булевых логических выражений с кванторами (Ринтанен [1886], 2007). Разработка методов планирования с частичным упорядочиванием привела к созданию программ UWL (Этциони и др. [704], 1992) и CNLP (Пеот и Смит [1775], 1992). Опыт эксплуатации алгоритма Graphplan привел к появлению программы Sensory Graphplan, или SGP (Вельд и др. [2311], 1998).

Первым планировщиком с непрерывным планированием и контролем выполнения был тандем из программы PLANEX (Фикес и др. [737], 1972) и планировщика STRIPS, управлявшими роботом Shakey. Система SIPE (*System for Interactive Planning and Execution monitoring*) Уилкинса ([2344], 1988) была первым планировщиком, систематически занимавшимся перепланировкой. Она использовалась в демонстрационных проектах в нескольких проблемных областях, включая планирование операций на полетной палубе авианосца, планирование производственных процессов для австралийского пивзавода и планирование строительства многоэтажных зданий (Картам и Левитт [1194], 1990).

В середине 1980-х годов пессимизм в отношении медленной работы и длительности расчетов систем планирования привел к предложению рефлексных агентов, получивших название "системы ▶**реактивного планирования**" (Брукс [312], 1986; Агре и Чапмен [24], 1987). Был разработан метод "Universal plans" (Шопперс [2000], 1989), поддерживающий справочные таблицы для реактивного планирования, но вскоре выяснилось, что это повторное открытие идеи **стратегий**, которая уже давно использовались в марковских процессах принятия решений (см. главу 17). Кёниг в статье *Agent-Centered Search* ([1259], 2001) дал обзор методов непрерывного планирования.

Планирование с учетом ограничений по времени впервые было реализовано в программе DEVISER (Вере [2269], 1983). Проблема представления времени в планах была решена Алленом ([33], 1984) и Дином и соавт. ([566], 1990) в системе FORBIN. Программы NONLIN+ (Тейт и Уайтер [2182], 1984) и SIPE (Уилкинс [2345], 1990) обладали способностью рассуждать о распределении ограниченных ресурсов по различным этапам плана. Программа O-PLAN (Белл и Тейт [158], 1985) применялась для планирования поставок программного обеспечения в компании Price Waterhouse и для планирования сборки заднего моста автомобиля в компании Jaguar Cars.

В двух планировщиках, SAPA (До и Камбхампати [627], 2001) и T4 (Хаслум и Геффнер [978], 2001), используется прямой поиск в пространстве состояний в сочетании со сложными эвристиками для обработки действий, имеющих заданную продолжительность и требующих ресурсов. Альтернативой этому подходу является применение очень выразительных языков действий, но под управлением составленных людьми эвристик, характерных для конкретной проблемной области, как это сделано в системах ASPEN (Фукунага и др. [797], 1997), HSTS (Джонссон и др. [1148], 2000) и IxTeT (Халлаб и Лорель [847], 1994).

Был разработан и развернут ряд гибридных систем планирования и составления расписаний: система ISIS (Фокс и др. [768], 1982; Фокс [767], 1990) использовалась при составлении производственных графиков в компании Westinghouse, система GARI (Дескотт и Латомб [611], 1985) осуществляла планирование машинной обработки и конструирования механических деталей, система FORBIN применялась для управления фабрикой, а система NONLIN+ использовалась как инструмент планирования поставок для военно-морского флота. В этой книге планирование и составление расписаний рассматриваются как две независимые задачи, однако Кушинг и соавт. ([508], 2007) показали, что для определенных задач это может приводить к неполноте.

Долгую историю имеет задача составления расписаний и графиков для аэрокосмической отрасли. Программа T-SCHED (Драббл [647], 1990) использовалась для составления графика выполнения последовательности команд миссии для спутника Uosat-II. Программы OPTIMUM-AIV (Ааруп и др. [3], 1994) и PLAN-ERS1 (Фучс и др. [795], 1990), основанные на программе O-PLAN, применялись в Европейском космическом агентстве для сборки космических аппаратов и планирования

наблюдений соответственно. Программа SPIKE (Джонстон и Адорф [1142], 1992) использовалась в NASA для планирования наблюдений с помощью космического телескопа Хаббл, а система Space Shuttle Ground Processing Scheduling System (Дил и др. [564], 1994) осуществляла составление производственного расписания с охватом до 16 000 рабочих смен. Программа Remote Agent (Мускеттола и др. [1648], 1998) стала первой *автономной* программой планирования и составления расписаний, используемой для управления космическим аппаратом, после ее развертывания на борту космического зонда Deep Space One в 1999 году. Использование приложений в космических исследованиях стимулировало разработку алгоритмов для решения задачи распределения ресурсов (см. статьи Лабори ([1334], 2003) и Мускеттола ([1647], 2002)). Литература по теме составления расписаний представлена классической статьей-обзором Лаулера и соавт. ([1365], 1993), книгой Пинедо ([1791], 2008) и справочником под редакцией Блазевича (Блазевич и др. [227], 2007).

Вычислительная сложность задач планирования была проанализирована несколькими авторами (Байлендер [354], 1994; Халлаб и др. [848], 2004; Ринтанен [1888], 2016). Здесь есть две основные задачи: **PlanSAT** — это вопрос о том, существует ли какой-либо план, который решает некоторую задачу планирования, и **Bounded PlanSAT** — вопрос, существует ли решение длиной k или меньше, что может быть использовано для отыскания оптимального плана. Обе задачи разрешимы в классическом планировании (поскольку число состояний является конечным). Но если добавить в язык функциональные символы, то число состояний становится бесконечным и задача PlanSAT превращается лишь в полуразрешимую. Для пропозициональных задач обе основные задачи имеют сложность класса PSPACE — класса, который больше (и, следовательно, сложнее), чем NP. Он относится к задачам, которые могут быть решены с помощью детерминированной машины Тьюринга с полиномиальным размером пространства. Эти теоретические результаты обескураживают, но на практике задачи, которые требуется решать, обычно не настолько плохи. Истинным преимуществом формализма классического планирования является то, что он облегчает разработку очень точных эвристик, независимых от проблемной области, — другие подходы в этом смысле являются не столь плодотворными.

Книга *Readings in Planning* (Аллен и др. [35], 1990) — это всеобъемлющая антология ранних работ в этой области. Вельд ([2309], 1994; [2310], 1999) предлагает два отличных обзора алгоритмов планирования на период 1990-х. Интересно наблюдать за изменениями, произошедшими в течение пяти лет между выходом этих двух обзоров: первый концентрируется на планировании с частичным упорядочиванием, а во втором вводятся алгоритмы Graphplan и SATPLAN. Книга *Automated Planning and Acting* (Халлаб и др. [849], 2016) является отличным учебником по всем аспектам этой области. Учебник Лаваля *Planning Algorithms* ([1363], 2006) охватывает как классическое, так и стохастическое планирование с развернутым охватом задач планирования движения робота.

Исследования по планированию играли центральную роль в искусственном интеллекте со времени появления этого научного направления, а статьи по планированию занимают основной объем ведущих журналов и материалов конференций по искусственному интеллекту. Также проводятся специализированные конференции по планированию, такие как *International Conference on Automated Planning and Scheduling* и *International Workshop on Planning and Scheduling for Space.*

Упражнения

11.1. Рассмотрим робота, возможные действия которого описываются следующими операторами языка PDDL.

$Op(Go(x, y), At(Robot, x), \neg At(Robot, x) \wedge At(Robot, y))$
$Op(Pick(o), At(Robot, x) \wedge At(o, x), \neg At(o, x) \wedge Holding(o))$
$Op(Drop(o), At(Robot, x) \wedge Holding(o), At(o, x) \wedge \neg Holding(o)$

а) Эти операторы позволяют роботу держать более одного объекта. Покажите, как с помощью предиката *EmptyHand* (пустая рука) изменить их для робота, который может держать только один объект.

а) Предполагая, что это единственно возможные действия в данном мире, напишите аксиому определения преемника для предиката *EmptyHand.*

11.2. Опишите различие и сходство между решением задач и планированием.

11.3. Принимая во внимание схемы действий и начальное состояние задачи планирования грузовых авиаперевозок, представленные на рис. 11.16, каковы будут все применимые конкретные реализации действия $Fly(p, from, to)$ в состоянии, описываемом высказыванием

$$At(P_1, JFK) \wedge At(P_2, SFO) \wedge Plane(P_1) \wedge Plane(P_2) \wedge Airport(JFK) \wedge Airport(SFO)?$$

$Init(At(C_1, SFO) \wedge At(C_2, JFK) \wedge At(P_1, SFO) \wedge At(P_2, JFK)$
$\wedge Cargo(C_1) \wedge Cargo(C_2) \wedge Plane(P_1) \wedge Plane(P_2)$
$\wedge Airport(JFK) \wedge Airport(SFO))$
$Goal(At(C_1, JFK) \wedge At(C_2, SFO))$
$Action(Load(c, p, a),$
PRECOND: $At(c, a) \wedge At(p, a) \wedge Cargo(c) \wedge Plane(p) \wedge Airport(a)$
EFFECT: $\neg At(c, a) \wedge In(c, p))$
$Action((Unload(c, p, a),$
PRECOND: $In(c, p) \wedge At(p, a) \wedge Cargo(c) \wedge Plane(p) \wedge Airport(a)$
EFFECT: $At(c, a) \wedge \neg In(c, p))$
$Action((Fly(p, from, to),$
PRECOND: $At(p, from) \wedge Plane(p) \wedge Airport(from) \wedge Airport(to)$
EFFECT: $\neg At(p, from) \wedge At(p, to))$

Рис. 11.16. Описание на языке PDDL задачи планирования грузовых авиаперевозок

11.4. Перед обезьяной в лаборатории поставлена задача "обезьяна и бананы": на потолке подвешены бананы, которые нельзя достать непосредственно с пола, однако в помещении имеется ящик, позволяющий обезьяне достать бананы, если она на него заберется. Первоначально обезьяна находится в позиции A, бананы — в позиции B, а ящик — в позиции C. Обезьяна и ящик имеют высоту *Low*, но если обезьяна заберется на ящик, то она будет иметь высоту *High* — такую же, как и у бананов. Действия, доступные для обезьяны, включают *Go* (переход из одного места в другое), *Push* (перемещение объекта из одного места в другое), *ClimbUp* (подъем на объект) и *ClimbDown* (спуск с объекта), а также *Grasp* (схватить объект) или *Ungrasp* (отпустить объект). Схватывание приводит к овладению объектом, если обезьяна и объект находятся в одном и том же месте и на одной и той же высоте.

а) Составьте описание начального состояния.

б) Запишите шесть схем действий.

в) Предположим, что обезьяна хочет поставить в тупик ученых (которые отправились пить чай), схватив бананы, а затем вернув ящик на первоначальное место. Запишите эту задачу как общую цель (т.е. не предполагая, что ящик в конечном счете обязательно будет находиться в позиции C) на языке исчисления ситуаций. Может ли задача достижения этой цели быть решена с помощью классической системы планирования?

г) Вероятно, ваша схема перемещения является неверной, поскольку, если объект слишком тяжелый, его положение при применении схемы *Push* не изменится. Исправьте схему этого действия с целью учета возможности существования тяжелых объектов.

11.5. Оригинальный планировщик STRIPS был разработан для управления роботом Shakey. На рис. 11.17 представлена одна из версий мира Shakey, состоящая из четырех комнат (*Room*), расположенных вдоль коридора (*Corridor*), где в каждой комнате имеются дверь (*Door*) и выключатель света (*Switch*). Действия в мире робота Shakey включают перемещение из одного места в другое, передвижение подвижных объектов (таких, как ящики), подъем на прочные объекты (такие, как ящики) и спуск с них, а также включение и выключение света с помощью выключателей. Сам робот никогда не забирался на ящики и не щелкал выключателями света, но планировщик STRIPS был способен находить и выводить на печать планы, расширяющие базовые способности робота. Ниже перечислены шесть действий робота Shakey.

- Действие $Go(x, y)$, которое требует, чтобы Shakey был в позиции x и перешел в позицию y, а также определяет, что позиции x и y находятся в одной и той же комнате r. В соответствии с общепринятым соглашением дверь между двумя комнатами считается находящейся одновременно в обеих этих комнатах.
- Действие $Push(b, x, y, r)$ предполагает перемещение ящика b из позиции x в позицию y в пределах одной и той же комнаты r. Дополнительно потребуются предикат *Box* и константы для описания ящиков.
- Действие $ClimbUp(b)$ — подъем на ящик b, и действие $ClimbDown(b)$ — спуск с ящика b. Дополнительно потребуются предикат *On* и константа *Floor* (пол).
- Действие $TurnOn(s, b)$ — включение выключателя света s, и действие $TurnOff(s, b)$ — его выключение. Для того чтобы включить или выключить свет, робот Shakey должен стоять на ящике b, расположенном в том месте, где находится выключатель света.

На языке PDDL напишите высказывания, определяющие шесть действий робота Shakey и начальное состояние, представленное на рис. 11.17. Составьте план, с помощью которого робот Shakey мог бы переместить ящик *Box* 2 в комнату *Room* 2.

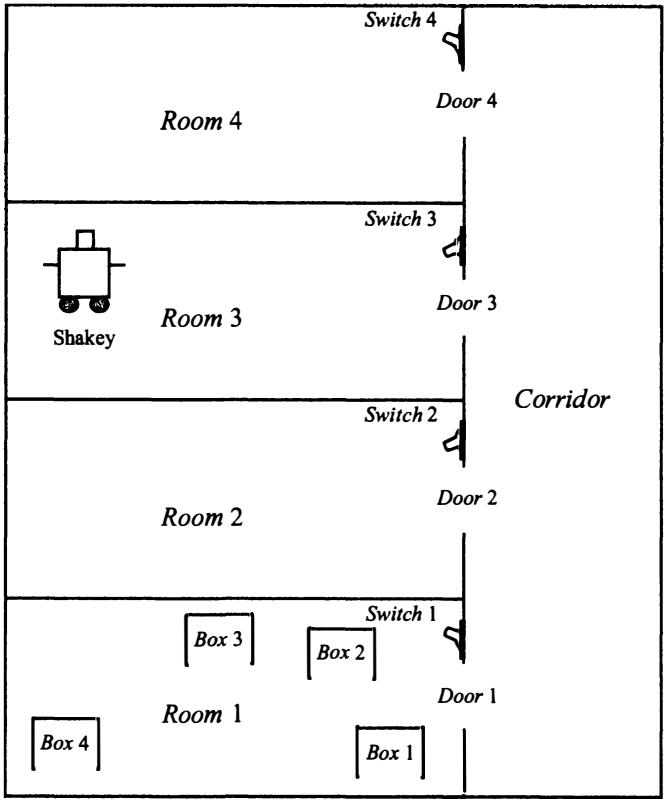


Рис. 11.17. Мир робота Shakey. Робот Shakey может передвигаться между объектами в пределах комнаты, проходить через двери между комнатами и коридором, взбираться на те объекты, на которые ему разрешено взбираться, и двигать те объекты, которые ему разрешено двигать, а также щелкать выключателями света

11.6. Конечная машина Тьюринга представляет собой конечную одномерную ленту с ячейками, каждая из которых содержит один из конечного числа символов. Над одной из ячеек находится головка записи/чтения. Существует конечное множество состояний, в которых может находиться такая машина, одним из которых является состояние чтения. Для каждого этапа времени существует множество действий, из которых можно делать выбор, зависящее от символа в ячейке под головкой чтения/записи и текущего состояния машины. Каждое действие включает в себя запись символа в ячейку под головкой, перевод машины в некоторое состояние и, возможно, перемещение головки на одну ячейку влево или вправо. Отображение, определяющее, какие действия разрешены, является программой

для машины Тьюринга. Ваша цель — перевести машину в состояние чтения. Представьте задачу перевода машины Тьюринга в состояние чтения как задачу планирования. Если вы сможете сделать это, то продемонстрируете, что определение, есть ли у задачи планирования решение, по крайней мере так же сложно, как задача перевода машины Тьюринга в состояние чтения, которая имеет сложность PSPACE.

11.7. Объясните, почему удаление отрицательных эффектов из каждой схемы действия приводит к получению ослабленной задачи, — в том случае, когда предварительные условия и цели содержат только положительные литералы.

11.8. На рис. 11.18 показана задача в мире блоков, известная как аномалия Зюссмана. Эта задача рассматривается как аномальная, поскольку планировщики без чередования, применявшиеся в начале 1970-х годов, не могли ее решить. Запишите определение этой задачи и решите ее либо вручную, либо с использованием программы планирования. Планировщиком без чередования является такой планировщик, который после получения двух подцелей, G_1 и G_2, вырабатывает либо план для G_1, который соединяется с планом для G_2, либо наоборот. Объясните, мог ли планировщик без чередования решить эту задачу. Если да, то как именно, а если нет, то почему?



Исходное состояние Целевое состояние

Рис. 11.18. Задача планирования в мире блоков, получившая название "аномалия Зюссмана"

11.9. Докажите, что обратный поиск является полным для задач, представленных на языке PDDL.

11.10. Вернитесь к определению двунаправленного поиска в главе 3.

а) Будет ли идея двунаправленного поиска в пространстве состояний перспективной для использования в планировании?

б) А что можно сказать в отношении двунаправленного поиска в пространстве планов с частичным упорядочиванием?

в) Разработайте версию планирования с частичным упорядочиванием, в которой действие разрешается добавить в план, если его предусловия могут быть достигнуты с помощью эффектов действий, уже находящихся в плане. Объясните, как в этом случае можно справиться с конфликтами и ограничениями упорядоченности. Будет ли полученный алгоритм по существу идентичным прямому поиску в пространстве состояний?

11.11. Планировщики с прямым и обратным поиском в пространстве состояний были противопоставлены планировщикам с частичным упорядочиванием указанием на то, что последние представляют собой средства поиска в пространстве планов. Объясните, на основании чего средства прямого и обратного поиска в пространстве состояний можно также рассматривать как средства поиска в пространстве планов, и укажите, чем в этом случае являются операторы уточнения плана.

11.12. До сих пор мы предполагали, что создаваемые планы всегда обеспечивают выполнение предусловий действия. А теперь рассмотрим, что должны утверждать пропозициональные аксиомы определения преемника, такие как $HaveArrow^{t+1} \Leftrightarrow (HaveArrow^t \Leftrightarrow \neg Shoot^t)$, в отношении действий, для которых предусловия не выполняются.

а) Покажите, что эти аксиомы предсказывают, что ничего не произойдет, когда действие выполняется в таком состоянии, в котором его предусловия не выполнены.

б) Рассмотрите план p, включающий действия, необходимые для достижения цели, но также включающий недопустимые действия. Можно ли описать данную категорию как

$$initialstate \wedge successor - stateaxioms \wedge p \models goal?$$

в) Возможно ли в исчислении ситуаций с помощью аксиом определения преемника первого порядка доказать, что план, содержащий недопустимые действия, позволяет достичь цели?

11.13. Рассмотрите, как перевести множество схем действий в аксиомы определения преемника в исчислении ситуаций.

а) Рассмотрите схему для действия $Fly(p, from, to)$. Напишите логическое определение для предиката $Poss(Fly(p, from, to), s)$, который будет истинным, если предусловия для действия $Fly(p, from, to)$ выполняются в ситуации s.

б) Далее, предполагая, что действие $Fly(p, from, to)$ является единственной схемой действия, доступной для агента, запишите аксиому определения преемника для флюенты $At(p, x, s)$, которая фиксирует ту же информацию, что и схема действия.

в) Теперь предположим, что существует дополнительный метод путешествия: действие $Teleport(p, from, to)$, имеющее дополнительное предусловие $\neg Warped(p)$ (*warped* — покоробленный) и дополнительный эффект $Warped(p)$. Объясните, как в этом случае следует изменить базу знаний исчисления ситуаций.

г) И наконец разработайте общую и точно определенную процедуру для выполнения перевода множества схем действий в множество аксиом определения преемника.

11.14. В алгоритме SATPLAN, приведенном на рис. 7.22, предусмотрено, что каждый вызов алгоритма проверки выполнимости подтверждает цель g^T, где T находится в пределах от 0 до T_{max}. Предположим вместо этого, что алгоритм проверки выполнимости вызывается только один раз с целью $g^0 \vee g^1 \vee \ldots \vee g^{T_{max}}$.

а) Будет ли такой вызов всегда возвращать план, если таковой существует, с длиной, меньшей или равной T_{max}?

б) Приведет ли такой подход к появлению каких-либо новых фиктивных "решений"?

в) Обсудите, как можно было бы модифицировать алгоритм проверки выполнимости, такой как WALKSAT, чтобы он находил короткие решения (если они существуют) после получения дизъюнктивной цели в указанной выше форме.

11.15. Все цели, которые рассматривались до сих пор, требовали от планировщика привести мир к целевому состоянию всего один раз, план выполнялся однократно. Однако не все цели могут быть выражены таким образом: вы не достигнете цели подвешивания люстры над полом, подбросив ее в воздух. А если серьезно, то очень не хотелось бы, чтобы система жизнеобеспечения космического корабля подавала кислород в первый день, но больше не занималась этим в последующие. Цель технического обслуживания достигается в том случае, когда план агента предусматривает условие непрерывного поддержания заданного состояния (не только один раз, но и в дальнейшем). Опишите, как расширить использовавшийся в этой главе синтаксис для поддержки целей обслуживания.

11.16. У вас есть несколько грузовиков, с помощью которых можно доставить множество грузов. Каждый груз отправляется из какого-то места на карте сеточного мира и должен быть доставлен в какое-то другое место. Управление каждым грузовиком выполняется посредством двух действий — движение вперед и поворот. Постройте иерархию действий высокого уровня для этой задачи. Какие знания о решении кодирует построенная вами иерархия?

11.17. Предположим, что действие высокого уровня (HLA) имеет только одну реализацию в виде последовательности примитивных действий. Приведите алгоритм вычисления его предусловий и эффектов, исходя из полной иерархии уточнений и схем для примитивных действий.

11.18. Предположим, что оптимистическое достижимое множество плана высокого уровня является надмножеством целевого множества. Можно ли сделать какой-либо вывод о том, достигает ли план поставленной цели? Что если пессимистическое достижимое множество не пересекается с целевым множеством? Объясните.

11.19. Напишите алгоритм, который на входе принимает начальное состояние (заданное множеством пропозициональных литералов) и последовательность HLA (каждое из которых определяется предусловиями и ангельскими спецификациями оптимистических и пессимистических достижимых множеств) и вычисляет оптимистическое и пессимистическое описания достижимого множества заданной последовательности.

11.20. На рис. 11.14 было показано, как описать действия в задаче планирования, используя отдельные поля для DURATION, USE и CONSAME. Теперь предположим, что была поставлена задача объединить планирование с недетерминированным планированием, которое требует недетерминированных и условных эффектов. Рассмотрите каждое из трех полей и объясните, должны ли они оставаться отдельными полями или они должны стать эффектами действия. Приведите пример для каждого из этих трех полей.

11.21. Некоторые операции в стандартных языках программирования можно моделировать как действия, меняющие состояние мира. Например, в операции присваивания изменяется содержимое участка памяти, а в операции печати изменяется состояние выходного потока. Программу, состоящую из этих операций, можно также рассматривать как план, цель которого задана в спецификации программы. Следовательно, алгоритмы планирования могут использоваться для создания программ, реализующих заданную спецификацию.

а) Напишите схему действий для оператора присваивания (в котором значение одной переменной присваивается другой). Помните, что первоначальное значение должно быть перезаписано!

б) Покажите, как процедура создания объекта может использоваться в планировщике для разработки плана обмена значениями между двумя переменными с применением временной переменной.

11.22. Ознакомьтесь со следующими доводами: "В инфраструктуре, которая допускает наличие неопределенных начальных состояний, **недетерминированные эффекты** (дизъюнктивные результаты) являются просто удобным способом обозначения, а не источником дополнительной выразительной мощи. Для любой схемы действий a с недетерминированным эффектом $P \vee Q$ его всегда можно заменить условным результатом R: $P \wedge \neg R$: Q, который, в свою очередь, можно свести к двум обычным действиям. Высказывание R обозначает случайное высказывание, которое не известно в начальном состоянии и для которого не предусмотрено каких-либо действий по получению информации с помощью датчиков. Являются ли эти доводы правильными? Рассмотрите отдельно два случая, в одном из которых в плане имеется только один экземпляр схемы действия a, тогда как во втором имеется больше одного экземпляра.

11.23. Предположим, что действие *Flip* всегда изменяет значение истинности переменной L. Покажите, как определить его эффекты, используя схему действий с условными эффектами. Покажите, что, несмотря на использование условных эффектов, представление доверительного состояния в форме 1-CNF сохраняет форму 1-CNF после выполнения действия *Flip*.

11.24. В мире блоков мы были вынуждены ввести два действия, *Move* и *MoveToTable*, чтобы обеспечить поддержку предиката *Clear* должным образом. Покажите, как условные результаты можно использовать для представления обоих этих случаев с помощью одного действия.

11.25. Условные результаты были проиллюстрированы на примере действия *Suck* в мире пылесоса — от того, в каком квадрате находится робот, зависит, какой из квадратов станет чистым. Можете ли вы предложить новое множество пропозициональных переменных для определения состояний мира пылесоса, таких, что действие *Suck* имеет безусловное описание? С использованием этих пропозициональных переменных составьте описания действий *Suck*, *Left* и *Right* и покажите, что их достаточно для представления всех возможных состояний мира.

11.26. Найдите достаточно грязный ковер, на поверхности которого нет препятствий, и почистите его пылесосом. Нарисуйте путь, проделанный чистящей насадкой пылесоса, настолько точно, насколько сможете. Объясните характер этого пути, ссылаясь на формы планирования, описанные в этой главе.

11.27. Приведенные ниже цитаты взяты из инструкций на флаконах с шампунем. Определите каждую из этих инструкций как безусловный план, условный план или план с контролем выполнения. **а)** "Нанесите и разотрите до образования пены. Ополосните волосы. Повторите процедуру". **б)** "Нанесите шампунь на волосы и оставьте на несколько минут. Прополосните волосы и повторите процедуру в случае необходимости". **в)** "Если не удается устранить проблему, обратитесь к врачу".

11.28. Рассмотрите следующую задачу: пациент поступил в приемную врача с симптомами, которые могут быть вызваны либо обезвоживанием, либо болезнью D (но не обеими этими причинами вместе). Существуют два возможных действия: *Drink* (питье), которое, безусловно, позволяет справиться с обезвоживанием, и *Medicate* (медикаментозное лечение), излечивающее заболевание D, но имеющее

нежелательный побочный эффект, который возникает, если организм пациента обезвожен. Составьте описание этой проблемы и разработайте план без использования датчиков, позволяющий решить эту задачу, перечислив все относящиеся к ней возможные миры.

11.29. В задаче из области медицины, приведенной в упражнении 11.28, добавьте действие *Test* (лабораторный анализ), имеющее условный результат *CultureGrowth* (рост бактериальной культуры), если предикат *Disease* (болезнь) имеет истинное значение и в любом случае имеет результат из восприятия *Known*(*CultureGrowth*) (*known* — известный). Составьте условный план, позволяющий решить эту задачу и свести к минимуму использование действия *Medicate*.

ПРИЛОЖЕНИЕ A

Математические основы

A.1. Анализ сложности и нотация O()

Специалистам в области компьютерных наук часто приходится решать задачу сравнения алгоритмов для определения того, насколько быстро они действуют или сколько памяти для них требуется. Для решения этой задачи предусмотрены два подхода. Первым из них является применение эталонных тестов (►**benchmarking**) — прогон реализующих алгоритмы программ на компьютере и измерение их быстродействия в секундах и потребления памяти в байтах. Безусловно, в конечном итоге нас действительно интересуют именно такие практические характеристики, но эталонное тестирование может оказаться неудовлетворительным просто потому, что оно слишком специфично: в нем измеряется производительность конкретной программы, написанной на конкретном языке, выполняемой на конкретном компьютере с конкретным компилятором и с конкретными входными данными. К тому же на основании единственного результата, полученного с помощью эталонного тестирования, очень трудно предсказать, насколько успешно этот алгоритм будет действовать в случае использования другого компилятора, компьютера или набора данных. Второй подход предполагает математический ►**анализ алгоритмов**, не зависящий от их конкретной реализации и входных данных. Именно этот подход будет обсуждаться ниже.

A.1.1. Асимптотический анализ

Подход, основанный на математическом анализе алгоритмов и не зависящий от их конкретной реализации и входных данных, мы рассмотрим на примере приведенной ниже программы, вычисляющей сумму последовательности чисел.

```
function SUMMATION(последовательность) returns число
  сумма ← 0
  for i = 1 to LENGTH(последовательность) do
     сумма ← сумма + последовательность[i]
  return сумма
```

Первый этап анализа состоит в том, что создается определенное абстрактное представление входных данных, позволяющее найти какой-то параметр или

параметры, характеризующие объем входных данных. В рассматриваемом примере объем входных данных можно охарактеризовать с помощью такого параметра, как длина последовательности, которую мы обозначим как n. На втором этапе необходимо определить абстрактное представление реализации и найти какой-то критерий, отражающий продолжительность выполнения алгоритма, но не привязанный к конкретному компилятору или компьютеру. Применительно к программе SUMMATION этим критерием может служить количество строк выполняемого кода; кроме того, данный критерий может быть более детализированным и измеряющим количество сложений, присваиваний, обращений к элементам массивов, а также ветвлений, выполняемых в этом алгоритме. В любом случае будет получена характеристика общего количества шагов, выполняемых алгоритмом, как функция от объема входных данных. Обозначим эту характеристику как $T(n)$. Если за основу берется количество строк кода, то в данном примере $T(n)=2n+2$.

Если бы все программы были такими же простыми, как SUMMATION, то область анализа алгоритмов не заслуживала бы названия научной. Но исследования в этой области существенно усложняются из-за наличия двух проблем. Первая проблема заключается в том, что редко удается найти параметр, подобный $T(n)$, который бы полностью характеризовал количество шагов, выполняемых при прогоне алгоритма. Вместо этого чаще всего можно лишь вычислить данный показатель для наихудшего случая $T_{worst}(n)$ или для среднего случая $T_{avg}(n)$. Причем, для вычисления среднего показателя необходимо, чтобы аналитик принял какие-то обоснованные предположения в отношении распределения, характеризующего набор входных данных.

Вторая проблема состоит в том, что алгоритмы обычно не поддаются точному анализу. В этом случае приходится прибегать к аппроксимации. В нашем случае, например, можно было бы сказать, что быстродействие алгоритма SUMMATION характеризуется величиной $O(n)$, имея в виду, что быстродействие этого алгоритма измеряется величиной, пропорциональной n, — возможно, за исключением нескольких небольших значений n. Более формально это определение можно представить с помощью следующей формулы.

$$T(n) \text{ есть } O(f(n)), \text{ если } T(n) \le kf(n) \text{ для некоторого } k, \text{ для всех } n > n_0$$

Перейдя к использованию нотации $O(\ )$, мы получаем возможность воспользоваться так называемым ▶**асимптотическим анализом**. В рамках этого подхода можно, например, безоговорочно утверждать, что если n асимптотически приближается к бесконечности, то алгоритм, характеризующийся показателем $O(n)$, проявляет себя лучше по сравнению с алгоритмом $O(n^2)$, тогда как единственное число, полученное с помощью эталонного тестирования, не может служить обоснованием подобного утверждения.

Нотация $O(\ )$ позволяет создать абстрактное представление, в котором не учитываются постоянные коэффициенты, благодаря чему она становится более простой в использовании, но менее точной, чем нотация $T(\ )$. Например, в конечном

итоге алгоритм $O(n^2)$ всегда будет считаться худшим по сравнению с алгоритмом $O(n)$, но если бы эти два алгоритма характеризовались значениями $T(n^2+1)$ и $T(100n+1000)$, то фактически алгоритм $O(n^2)$ был бы лучше при $n<110$.

Несмотря на этот недостаток, асимптотический анализ представляет собой наиболее широко используемый инструмент анализа алгоритмов. Этот метод можно считать достаточно точным, поскольку в процессе анализа создается абстрактное представление и для точного количества операций (поскольку игнорируется постоянный коэффициент k), и для точного содержимого входных данных (поскольку рассматривается исключительно их объем n); благодаря чему анализ становится вполне осуществимым с использованием математических методов. Система обозначений $O()$ представляет собой хороший компромисс между точностью и простотой анализа.

A.1.2. Изначально сложные и недетерминированные полиномиальные задачи

Анализ алгоритмов и нотация $O()$ позволяют рассуждать об эффективности конкретного алгоритма. Однако эти методы не позволяют определить, может ли существовать лучший алгоритм для рассматриваемой задачи. В области ▶**анализа сложности** исследуются задачи, а не алгоритмы. Первая широкая градация в этой области проводится между задачами, которые могут быть решены за время, измеряемое полиномиальным соотношением, и задачами, которые не могут быть решены за время, измеряемое полиномиальным соотношением, независимо от того, какой алгоритм для этого используется. Класс полиномиальных задач — т.е. задач, которые могут быть решены за время $O(n^k)$ для некоторого k), — обозначается как ▶**P**. Эти задачи иногда называют "простыми", поскольку данный класс содержит задачи, имеющие такую продолжительность выполнения, как $O(\log n)$ и $O(n)$. Но он содержит и задачи с затратами времени $O(n^{1000})$, поэтому определение "простая" не следует понимать слишком буквально.

Другим важным классом является ▶**NP** (Nondeterministic Polynomial) — класс недетерминированных полиномиальных задач. Задача относится к этому классу, если существует алгоритм, позволяющий выдвинуть гипотезу о возможном решении, а затем проверить правильность этой гипотезы с помощью полиномиальных затрат времени. Идея такого подхода состоит в том, что если бы можно было воспользоваться сколь угодно большим количеством процессоров, чтобы проверить одновременно все гипотезы, или оказаться крайне удачливым и всегда с первого раза находить правильную гипотезу, то NP-трудные задачи стали бы P-трудными задачами. Одним из самых важных нерешенных вопросов в компьютерных науках является то, будет ли класс NP эквивалентным классу P, если нельзя воспользоваться бесконечным количеством процессоров или способностью находить правильную гипотезу с первого раза. Большинство специалистов в области компьютерных наук согласны с тем, что $P \neq NP$, иными словами, что NP-задачи

являются изначально трудными и для них не существует алгоритмов с полиномиальными затратами времени. Но это утверждение так и не было доказано.

Ученые, пытающиеся найти ответ на вопрос о том, эквивалентны ли классы P и NP, выделили подкласс класса NP, называемый ▶**NP-полными** задачами. В этой формулировке слово "полный" означает "являющийся наиболее ярким представителем" и поэтому указывает на самые трудные задачи из класса NP. Было доказано, что либо все NP-полные задачи принадлежат к классу P, либо ни одна из них к нему не относится. Таким образом, данный класс представляет определенный теоретический интерес, но он важен также с точки зрения практики, поскольку известно, что многие серьезные задачи являются NP-полными. В качестве примера можно указать задачу установления выполнимости: если дано высказывание логики высказываний, то есть ли такой вариант присваивания истинностных значений символам высказывания, при котором оно становится истинным? Если не произойдет чудо и не совпадут друг с другом классы P и NP, то нельзя будет найти алгоритм, который позволяет решить *все* задачи установления выполнимости за полиномиальное время. Но исследователей в области искусственного интеллекта в большей степени интересует то, существуют ли алгоритмы, действующие достаточно эффективно при решении *типичных задач*, выбранных с помощью заранее заданного распределения; как было показано в главе 7, существуют алгоритмы наподобие WalkSAT, которые действуют вполне успешно при решении многих задач.

Класс ▶**NP-сложных** задач состоит из тех задач, которые сводятся (за полиномиальное время) ко всем проблемам в NP, поэтому, если вы решили любую NP-сложную задачу, вы сможете решить все проблемы в NP. Все NP-полные задачи являются NP-сложными, но есть некоторые NP-сложные проблемы, которые даже сложнее, чем NP-полные.

Класс ▶**co-NP** является комплементарным (дополнительным) по отношению к классу NP в том смысле, что для каждой задачи принятия решения из класса NP существует соответствующая задача в классе co-NP, на которую может быть дан положительный или отрицательный ответ, противоположный ответу на задачу класса NP. Известно, что класс P является подмножеством и NP, и co-NP, кроме того, считается, что к классу co-NP относятся некоторые задачи, не входящие в класс P. Такие задачи называются ▶**co-NP-полными** и являются самыми трудными задачами в классе co-NP.

Класс #P (произносится как "шарп P" или "диез P") представляет собой множество задач подсчета количества вариантов, соответствующих задачам принятия решения из класса NP. Задачи принятия решения имеют однозначный (положительный или отрицательный) ответ. Примером задачи такого типа является следующая: "Существует ли решение для данной формулы 3-SAT?" Задачи подсчета количества вариантов имеют целочисленный ответ, например: "Сколько решений существует для данной формулы 3-SAT?" В некоторых случаях задача подсчета количества вариантов намного труднее по сравнению с соответствующей задачей

принятия решения. Например, принятие решения о том, имеет ли двухдольный граф идеальное сочетание пар, может быть выполнено за время $O(VE)$ (где V — количество вершин; E — количество ребер графа), тогда как задача подсчета того, какое количество идеальных сочетаний пар имеется в этом двухдольном графе, является #P-полной. Это означает, что она не менее трудна, чем любая задача из класса #P, и поэтому по меньшей мере столь же трудна, как и любая задача NP.

Другим классом является класс задач PSPACE. К нему относятся задачи, для которых требуется объем пространства, определяемый полиномиальной зависимостью, даже при их прогоне на недетерминированной машине. Считается, что PSPACE-трудные задачи решаются хуже NP-полных задач, но не исключено, что в ходе дальнейших исследований может быть установлено, что класс NP эквивалентен классу PSPACE и класс P эквивалентен классу NP.

А.2. Векторы, матрицы и линейная алгебра

В математике ▶ **вектор** определяется как элемент векторного пространства, но мы будем использовать более конкретное определение: *вектор* — это упорядоченная последовательность значений. Например, в двухмерном пространстве могут быть определены такие векторы, как $\mathbf{x} = \langle 3, 4 \rangle$ и $\mathbf{y} = \langle 0, 2 \rangle$. В этом приложении соблюдаются обычные соглашения об обозначении векторов с помощью полужирных символов, хотя некоторые авторы отмечают имена векторов с помощью стрелок ($\vec{x}$) или знаков надчеркивания ($\overline{y}$). Элементы вектора обозначаются с помощью подстрочных индексов: $\mathbf{z} = \langle z_1, z_2, \ldots, z_n \rangle$. К сожалению, в этой книге синтезированы работы из многих областей исследований, где упорядоченные последовательности элементов могут называться по-разному ("векторы", "списки" или "кортежи"), в соответствии с чем для них используются и разные нотации: $\langle 1,2 \rangle$, $[1,2]$ или $(1,2)$.

Двумя фундаментальными операциями над векторами являются векторное сложение и скалярное умножение. Векторное сложение $\mathbf{x} + \mathbf{y}$ — это поэлементная сумма: $\mathbf{x} + \mathbf{y} = \langle 3 + 0, 4 + 2 \rangle = \langle 3, 6 \rangle$, а скалярное умножение — это операция умножения каждого элемента вектора на некоторую константу: $5\mathbf{x} = \langle 5 \times 3, 5 \times 4 \rangle = \langle 15, 20 \rangle$.

Длина вектора обозначается как $|\mathbf{x}|$ и вычисляется путем извлечения квадратного корня из суммы квадратов его элементов: $|\mathbf{x}| = \sqrt{(3^2 + 4^2)} = 5$. Точечное произведение (называемое также скалярным произведением) двух векторов, $\mathbf{x} \cdot \mathbf{y}$, представляет собой сумму произведений их соответствующих элементов; иными словами, $\mathbf{x} \cdot \mathbf{y} = \sum_i x_i y_i$, или в данном конкретном случае: $\mathbf{x} \cdot \mathbf{y} = 3 \times 0 + 4 \times 2 = 8$.

Векторы часто интерпретируются как направленные отрезки прямых (подобные стрелкам) в n-мерном евклидовом пространстве. В таком случае операция сложения векторов эквивалентна совмещению конца одного вектора с началом другого, а точечное произведение $\mathbf{x} \cdot \mathbf{y}$ эквивалентно выражению $|\mathbf{x}| \cdot |\mathbf{y}| \cdot \cos\theta$, где θ — угол между векторами $\mathbf{x}$ и $\mathbf{y}$.

►**Матрица** представляет собой прямоугольный массив значений, упорядоченных по строкам и столбцам. Ниже показана матрица **A** размером 3×4.

$$\begin{pmatrix} \mathbf{A}_{1,1} & \mathbf{A}_{1,2} & \mathbf{A}_{1,3} & \mathbf{A}_{1,4} \\ \mathbf{A}_{2,1} & \mathbf{A}_{2,2} & \mathbf{A}_{2,3} & \mathbf{A}_{2,4} \\ \mathbf{A}_{3,1} & \mathbf{A}_{3,2} & \mathbf{A}_{3,3} & \mathbf{A}_{3,4} \end{pmatrix}$$

Первый подстрочный индекс в обозначении $\mathbf{A}_{i,j}$ определяет строку, а второй — столбец. В языках программирования $\mathbf{A}_{i,j}$ часто записывается как `A[i,j]` или `A[i][j]`.

Сумма двух матриц определяется путем сложения соответствующих элементов, поэтому $(\mathbf{A} + \mathbf{B})_{i,j} = \mathbf{A}_{i,j} + \mathbf{B}_{i,j}$. (Если матрицы **A** и **B** имеют разные размеры, то их сумма не определена.) Можно также определить операцию умножения матрицы на скаляр: $(c\mathbf{A})_{i,j} = c\mathbf{A}_{i,j}$. Операция умножения матриц (получения произведения двух матриц) является более сложной. Произведение **AB** определено, только если матрица **A** имеет размеры $a \times b$, а матрица **B** имеет размер $b \times c$ (т.е. вторая матрица имеет количество строк, совпадающее с количеством столбцов первой матрицы). Результатом этой операции является матрица с размерами $a \times c$. Если матрицы имеют допустимые размеры, то результат их умножения является следующим:

$$(\mathbf{AB})_{i,k} = \sum_j \mathbf{A}_{i,j}\mathbf{B}_{j,k}.$$

Умножение матриц не является коммутативным даже для квадратных матриц: $\mathbf{AB} \neq \mathbf{BA}$ в любом случае. Однако эта операция является ассоциативной: $(\mathbf{AB})\mathbf{C} = \mathbf{A}(\mathbf{BC})$. Обратите внимание, что точечное произведение векторов может быть выражено в терминах транспонирования и умножения матриц: $\mathbf{x} \cdot \mathbf{y} = \mathbf{x}^{\mathsf{T}}\mathbf{y}$.

►**Единичная матрица I** имеет элементы $\mathbf{I}_{i,j}$, равные 1, если $i = j$, и равные 0 в противном случае. Она обладает таким свойством, что $\mathbf{AI} = \mathbf{A}$ для всех матриц **A**. ►**Транспонирование** — это специфическая для матриц операция (записывается как $\mathbf{A}^{\mathsf{T}}$), заключающаяся в превращении строк матрицы в столбцы и обратно, или, более формально, $\mathbf{A}^{\mathsf{T}}{}_{i,j} = \mathbf{A}_{j,i}$. Если матрица **A** квадратная, то ее ►**обратная матрица** $\mathbf{A}^{-1}$ определяется как такая матрица, для которой $\mathbf{A}^{-1}\mathbf{A} = \mathbf{I}$. Для ►**вырожденной матрицы** обратной матрицы не существует, а для невырожденной матрицы обратная матрица может быть вычислена за время $O(n^3)$.

Матрицы используются для решения систем линейных уравнений за время $O(n^3)$, которое необходимо для инвертирования матрицы, составленной из коэффициентов уравнений. Рассмотрим следующую систему уравнений, для которой требуется найти решение в терминах переменных x, y и z.

$$\begin{aligned} +2x + y - z &= 8 \\ -3x - y + 2z &= -11 \\ -2x + y + 2z &= -3 \end{aligned}$$

Эту систему уравнений можно представить в виде матричного уравнения $\mathbf{Ax} = \mathbf{b}$, где:

$$\mathbf{A} = \begin{pmatrix} 2 & 1 & -1 \\ -3 & -1 & 2 \\ -2 & 1 & 2 \end{pmatrix}, \quad \mathbf{x} = \begin{pmatrix} x \\ y \\ z \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} 8 \\ -11 \\ -3 \end{pmatrix}.$$

Чтобы решить уравнение $\mathbf{Ax} = \mathbf{b}$, достаточно умножить обе его стороны на обратную матрицу $\mathbf{A}^{-1}$, в результате чего получим $\mathbf{A}^{-1}\mathbf{Ax} = \mathbf{A}^{-1}\mathbf{b}$, что можно упростить до $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$. Проинвертировав матрицу $\mathbf{A}$ и умножив результат на вектор $\mathbf{b}$, получим искомый ответ:

$$\mathbf{x} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 2 \\ 3 \\ -1 \end{pmatrix}.$$

Еще несколько замечаний по поводу используемых в книге обозначений.

1. Мы используем запись $\log(x)$ для представления натуральных логарифмов $\log_e(x)$.
2. Мы используем запись $\text{argmax}_x f(x)$ для ссылок на значение x, при котором значение функции $f(x)$ является максимальным.

А.3. Распределения вероятностей

Вероятность — это мера, заданная на множестве событий, которая удовлетворяет трем приведенным ниже аксиомам.

1. Мера каждого события находится в пределах от 0 до 1. Это утверждение записывается как $0 \leq P(X = x_i) \leq 1$, где X — случайная переменная, представляющая событие, а x_i — возможные значения X. Обычно принято обозначать случайные переменные прописными буквами, а их значения — соответствующими строчными.
2. Мера всего множества равна 1, а это означает, что $\sum_{i=1}^{n} P(X = x_i) = 1$.
3. Вероятность объединения взаимоисключающих событий равна сумме вероятностей отдельных событий, т.е. $P(X = x_1 \vee X = x_2) = P(X = x_1) + P(X = x_2)$, гдесобытия x_1 и x_2 являются взаимоисключающими.

Вероятностная модель состоит из пространства выборок взаимоисключающих возможных результатов вместе с вероятностной мерой для каждого результата. Например, в модели погоды на завтра результатами могут быть *sunny* (солнечно), *cloudy* (облачно), *rainy* (дождь) и *snowy* (снег). Подмножество этих результатов представляет собой событие. Например, событие выпадения осадков — это подмножество $\{rainy, snowy\}$.

Мы будем использовать запись $\mathbf{P}(X)$ для обозначения вектора значений $\langle P(X = x_1), \ldots, P(X = x_n)\rangle$. Мы также используем запись $P(x_i)$ в качестве сокращения для $P(X = x_i)$ и $\sum_x P(x)$ в качестве сокращения для $\sum_{i=1}^{n} P(X = x_i)$.

Условная вероятность $P(B|A)$ определяется как $P(B \cap A)/P(A)$. Переменные A и B являются условно независимыми, если $P(B|A) = P(B)$ (или, равным образом, если $P(A|B) = P(A)$).

Непрерывные переменные имеют бесконечное количество значений, и если распределение вероятностей этих значений не характеризуется наличием пиковых значений в отдельных точках, то вероятность любого отдельно взятого значения равна 0. Поэтому имеет смысл говорить о значении вероятности в пределах некоторого диапазона. Это реализуется с помощью ▶**функции плотности вероятности**, которая имеет немного иной смысл по сравнению с дискретной функцией вероятности. Поскольку вероятность $P(X = x)$ — т.е. вероятность, которую X имеет при точном значении x — равна нулю, мы будем использовать другую меру: отношение вероятности того, что X попадает в интервал вокруг x, к ширине этого интервала, измеряемой в пределе приближения ширины интервала к нулю. Итак, функция плотности вероятности $P(X = x)$ определяется как

$$P(x) = \lim_{dx \to 0} P(x \le X \le x + dx) / dx.$$

Функция плотности вероятности должна быть неотрицательной при всех x и соответствовать следующему требованию:

$$\int_{-\infty}^{\infty} P(x)dx = 1.$$

Мы также можем определить ▶**кумулятивное распределение** $F_X(x)$, которое является вероятностью того, что случайная величина будет меньше x. Кумулятивная функция распределения определяется следующим образом:

$$F_X(x) = P(X \le x) = \int_{-\infty}^{x} P(u)du.$$

Следует отметить, что функция плотности вероятности измеряется в определенных единицах, а дискретная функция вероятности является безразмерной. Например, если переменная X измеряется в секундах, то плотность вероятности измеряется в герцах (Гц, т.е. 1/с). Если X — точка в трехмерном пространстве, измеряемом в метрах, то плотность вероятности будет измеряться в $1/м^3$.

Одним из наиболее важных распределений вероятностей является ▶**распределение Гаусса**, известное также под названием ▶**нормальное распределение**. Для этого распределения мы используем обозначение $\mathcal{N}(x; \mu, \sigma^2)$ — как функции от x со средним значением μ и среднеквадратичным отклонением σ (и, следовательно, с дисперсией σ^2). Нормальное распределение определяется следующей формулой:

$$\mathcal{N}(x;\mu;\sigma^2) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/(2\sigma^2)},$$

где x — непрерывная переменная, изменяющаяся в пределах от $-\infty$ до $+\infty$. Если среднее $\mu = 0$ и дисперсия $\sigma^2 = 1$, то имеет место частный случай нормального распределения, называемый ▶**стандартным нормальным распределением**. Если распределение задано на векторе **x** в пространстве с d измерениями, то оно представляет собой ▶**многомерное гауссово распределение**:

$$\mathcal{N}(\mathbf{x};\mu,\Sigma) = \frac{1}{\sqrt{(2\pi)^n \mid \Sigma \mid}} e^{-\frac{1}{2}\left((\mathbf{x}-\mu)^T \Sigma^{-1}(\mathbf{x}-\mu)\right)},$$

где μ — вектор средних, а S — ▶**матрица ковариации** этого распределения (см. ниже). Кумулятивное распределение для одномерного нормального распределения определяется как

$$F(x) = \int_{-\infty}^{\infty} \mathcal{N}(z;\propto,\sigma^2)dz = \frac{1}{2}(1 + \mathrm{erf}(\tfrac{x-\mu}{\sigma\sqrt{2}})),$$

где erf(x) — так называемая ▶**функция ошибок**, не имеющая представления в замкнутой форме.

В ▶**центральной предельной теореме** утверждается, что среднее n случайных переменных приближается к нормальному распределению по мере того, как n стремится к бесконечности. Такому свойству подчиняется почти любая коллекция случайных переменных, при том условии, что значение дисперсии любого конечного подмножества переменных не доминирует над значениями дисперсии других конечных подмножеств.

▶**Математическое ожидание** случайной величины, $M(X)$, представляет собой среднее значение, взвешенное по вероятности каждого значения. Для дискретной переменной это

$$M(X) = \sum_x x_i P(X = x_i).$$

Для непрерывной переменной суммирование следует заменить интегралом и использовать функцию плотности вероятности, $P(x)$:

$$M(X) = \int_{-\infty}^{\infty} xP(x)dx.$$

Для любой функции f мы также имеем

$$M(f(X)) = \int_{-\infty}^{\infty} f(x)P(x)dx.$$

И наконец, при необходимости математическое ожидание можно определить через распределение случайной величины:

$$M_{X\sim Q(x)}(g(X)) = \int_{-\infty}^{\infty} g(x)Q(x)dx.$$

Помимо математического ожидания, другие важные статистические характеристики распределения включают ▶**дисперсию** $D[X]$, которая является математическим ожиданием квадрата отклонения случайной величины от ее математического ожидания:

$$D[X] = M((X - M(X))^2)$$

и среднеквадратическое или ▶**стандартное отклонение** σ, которое является квадратным корнем из дисперсии (по этой причине математическое ожидание в статистике часто обозначают как σ^2).

▶**Среднее квадратическое** s для набора значений (чаще всего это выборка значений случайной величины) представляет собой квадратный корень из среднего арифметического квадратов значений в наборе:

$$s(x_1,\ldots,x_n) = \sqrt{\frac{x_1^2 + \ldots + x_n^2}{n}}.$$

▶**Ковариация** двух случайных величин определяется как математическое ожидание произведения разностей их значений и соответствующих средних значений:

$$\text{cov}(X, Y) = M((X - \mu X)(Y - \mu Y)).$$

▶**Ковариационная матрица**, часто обозначаемая как S, является матрицей, составленной из попарных ковариаций между элементами вектора случайных величин. Принимая, что вектор представлен как $\mathbf{X} = \langle X_1,\ldots X_n\rangle^{\mathrm{T}}$, элементами ковариационной матрицы будут:

$$\mathrm{S}_{i,j} = \text{cov}(X_i, X_j) = M((X_i - \mu_i)(X_j - \mu_j)).$$

▶**Выборка** — это часть общей совокупности возможных элементов случайной величины, полученная в результате некоторого эксперимента. Мы не знаем, каково будет распределение вероятности для любой конкретной выборки, но в пределе достаточно большой набор данных выборок будет приближаться к той же функции распределения плотности вероятности, которая свойственна общей совокупности элементов этой случайной величины. ▶**Равномерное распределение** характеризуется тем, что каждый элемент в общей совокупности в равной степени (равномерно) вероятен. Поэтому, если мы говорим, что "случайная величина имеет равномерное распределение в диапазоне целых чисел от 0 до 99", то это означает, что при выборке с одинаковой вероятностью может быть получено любое целое число из этого диапазона.

Библиографические и исторические заметки

Система обозначений $O(\ )$, которая широко используется в компьютерных науках в наши дни, была впервые определена в контексте теории чисел немецким математиком П.Г.Н. Бахманом (1894). Понятие NP-полноты было предложено Куком (1971), а современный метод определения способа сведения одной проблемы к другой предложен Карпом (1972). И Кук, и Карп получили за свою работу премию Тьюринга — высшую награду в компьютерных науках.

Из лучших учебников по анализу и разработке алгоритмов следует упомянуть книги Седвига и Уэйна (2011), а также Кормена, Лейзерсона, Ривеста и Штейна (2009). В этих изданиях особое внимание уделяется разработке и анализу алгоритмов для решения поддающихся решению задач. Теория NP-полноты и других форм неразрешимости представлена в книгах Гэри и Джонсона (1979) и Пападимитроу (1994). Хорошими учебниками по теории вероятности являются книги Чунга (1979), Росса (2015), а также Бертсекаса и Цициклиса (2008).

ПРИЛОЖЕНИЕ Б

Сведения о языках и алгоритмах, используемых в книге

Б.1. Определение языков с помощью формы Бэкуса–Наура

В этой книге дается определение нескольким языкам, в том числе языку логики высказываний (раздел 7.4), языку логики первого порядка (раздел 8.2) и подмножеству английского языка (раздел 23.4). Формальный язык определяется как множество строк, в котором каждая строка представляет собой последовательность символов. Все языки, которые были необходимы нам в этой книге, состоят из бесконечного множества строк, поэтому нужен простой способ, позволяющий охарактеризовать это множество. Для достижения данной цели удобнее всего воспользоваться **грамматикой**. Конкретный тип грамматики, который мы выбрали, носит название ►**контекстно-свободная грамматика**, поскольку каждое выражение в этом случае будет иметь одну и ту же форму в любом контексте. В качестве способа оформления грамматик мы приняли формальную систему под названием ►**форма Бэкуса–Наура** или **БНФ** (*Backus-Naur form*, *BNF*). В любой БНФ-грамматике присутствуют четыре компонента.

- Множество ►**терминальных символов**. Это символы или слова, из которых состоят строки языка. В качестве таких символов могут использоваться буквы (**A**, **B**, **C**, ...), слова (**a**, **aardvark**, **abacus**, ...) или любые символы, являющиеся допустимыми для области определения.
- Множество ►**нетерминальных символов**, которые обозначают компоненты предложений языка. Например, нетерминальный символ *NounPhrase* (именное словосочетание) в английском языке обозначает бесконечное множество строк, включая такие, как *you* или *the big slobbery dog*.
- ►**Начальный символ**, который представляет собой нетерминальный символ, обозначающий законченный набор из строк языка. В грамматике английского языка таковым является символ *Sentence*; в грамматике

арифметических выражений для этой цели может быть определен символ *Expr*, а в грамматике языка программирования это может быть *Program*.

- Множество ▶**правил подстановки** в форме $LHS \rightarrow RHS$, где *LHS* — нетерминальный символ, а *RHS* — последовательность, в которую входят нуль и больше символов. Это могут быть как терминальные, так и нетерминальные символы, а также символ ϵ, который мы будем использовать для обозначения пустой строки.

Правило подстановки вида

Sentence → *NounPhrase VerbPhrase*

означает, что при наличии двух строк, обозначенных как *NounPhrase* (именное словосочетание) и *VerbPhrase* (глагольное словосочетание), их можно соединить друг с другом и классифицировать результат как *Sentence*. Определение этого правила также можно записать в сокращенном виде: если есть два правила ($S \rightarrow A$) и ($S \rightarrow B$), то можно записать ($S \rightarrow A \mid B$). Чтобы проиллюстрировать эту концепцию, ниже приведена БНФ-грамматика для простых арифметических выражений.

Expr	→	*Expr Operator Expr* | (*Expr*) | *Number*
Number	→	*Digit* | *Number Digit*
Digit	→	**0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**
Operator	→	+ | - | ÷ | ×

Более подробные сведения о языках и грамматиках приведены в главе 23. Следует отметить, что в других книгах в системе обозначений БНФ могут использоваться немного другие правила, например нетерминальные символы могут обозначаться ⟨*Digit*⟩, а не *Digit*, терминальные символы — 'word', а не **word**, а в правилах может использоваться символ ::= вместо →.

Б.2. Описание алгоритмов с помощью псевдокода

В этой книге все упоминаемые алгоритмы представлены на ▶**псевдокоде**. Основные конструкции этого псевдокода должны быть понятны пользователям таких языков, как Java, C++ и в особенности Python. В некоторых местах для описания вычислений в псевдокоде используются математические формулы или текст на естественном языке, поскольку в противном случае пришлось бы применять более громоздкие конструкции. Также следует отметить, что в алгоритмах используются приведенные ниже соглашения.

- **Сохраняемые переменные**. Ключевое слово **persistent** (сохраняемая) используется как указание на то, что переменной присваивается начальное значение при первом вызове содержащей ее функции, после чего это

значение (или значение, присвоенное переменной с помощью выполненных в дальнейшем операторов присваивания) сохраняется при всех последующих вызовах функции. Таким образом, сохраняемые переменные подобны глобальным переменным в том, что они сохраняют свое значение после каждого вызова содержащей их функции, но при этом доступны только в данной функции. В программах агентов, приведенных в данной книге, сохраняемые переменные используются в качестве "памяти". В объектно-ориентированных языках, таких как C++, Java, Python и Smalltalk, программы с сохраняемыми переменными могут быть реализованы в виде *объектов*. В функциональных языках они могут быть реализованы с помощью *функциональных замыканий* в той среде, в которой определены требуемые переменные.

- **Функции как значения.** В нашем псевдокоде имена функций и процедур начинаются с прописных букв, а имена переменных состоят из строчных букв и выделяются курсивом. Поэтому в большинстве случаев вызов функции выглядит наподобие Fn(x). Однако также допускается, чтобы значением переменной была функция, например если значением переменной f является функция вычисления квадратного корня, то выражение $f(9)$ возвращает 3.
- **Значимость отступов**. Наличие отступов при записи псевдокода имеет большое значение — по аналогии с языками Python и CoffeeScript, но в отличие от языков Java, C++ и Go (в которых используются скобки) или языков Lua и Ruby (в которых используется оператор **end**), в нашем псевдокоде они определяют область действия цикла или условного выражения.
- **Деструктуризация наборов данных**. Нотация "$x, y \leftarrow$ *пара*" означает, что правая часть выражения после вычисления должна представлять собой двухэлементную коллекцию, первый элемент которой присваивается переменной x, а второй — y. Та же самая идея может быть представлена записью "**for** x, y **in** *пара* **do**". Эта нотация также может использоваться для обмена значениями между двумя переменными: "$x, y \leftarrow y, x$".
- **Значения по умолчанию для параметров**. Нотация "**function** F(x, y = 0) **returns** число" означает, что в этой функции y является дополнительным аргументом со значением по умолчанию, равным 0. Иначе говоря, вызовы функции F(3, 0) и $F(3)$ являются эквивалентными.
- **Ключевое слово yield**. Функция, содержащая ключевое слово **yield**, представляет собой ▶**генератор**, генерирующий *последовательность* значений, — по одному каждый раз, когда встречается выражение, содержащее это ключевое слово. После генерации очередного значения функция продолжает свое выполнение со следующего оператора. В языках Python, Ruby, C# и JavaScript (ECMAScript) присутствует подобная функциональная возможность.

- **Циклы**. В псевдокоде допускается четыре типа циклов.
  - "**for** *x* **in** *c* **do**" — выполняется цикл с управляющей переменной *x*, возможные значения которой указаны как последовательность элементов в коллекции *c*.
  - "**for** *i* = 1 **to** *n* **do**" — выполняется цикл с управляющей переменной *i*, значения которой представляют собой последовательность целых чисел от 1 до *n* включительно.
  - "**while** *условие* **do**" — в этом цикле выполнение *условия* проверяется перед началом очередной его итерации, и если условие ложно, цикл завершается.
  - "**repeat** ... **until** *условие*" — этот цикл безусловно выполняется первый раз, а затем проверяется *условие*. Если оно истинно, цикл завершается, иначе он выполняется вновь (и в конце опять проверяется *условие*).
- **Списки.** Нотация $[x, y, z]$ определяет список из трех элементов. Оператор "+" может употребляться для выполнения конкатенации списков: [1, 2] + [3, 4] = [1, 2, 3, 4]. Список может использоваться и как стек: функция Pop удаляет и возвращает последний элемент списка, а функция Top возвращает его последний элемент.
- **Множества.** Нотация $\{x, y, z\}$ определяет множество (набор) из трех элементов. Запись $\{x: p(x)\}$ определяет множество из всех элементов x, для которых $p(x)$ истинно.
- **Индексация в массивах начинается с 1.** Первый элемент массива всегда имеет индекс 1, как это принято в обычной математической записи (и в языках R и Julia), а не 0 (как это принято в языках Python, Java и C).

Б.3. Дополнительный материал в Интернете

Для этой книги имеется собственный веб-сайт, содержащий дополнительные материалы и инструкции по отправке предложений, а также предоставляющий посетителям возможность присоединиться к дискуссионным группам.

- `aima.cs.berkeley.edu`

Все приведенные в этой книге алгоритмы и несколько дополнительных упражнений по программированию были реализованы на языках Python и Java (а некоторые и на других языках). Код этих реализаций находится в интернет-хранилище и доступен на веб-сайте, который в настоящее время размещается по адресу

- `github.com/aimacode`

Предметный указатель

A

AGI 72
ASI 72
ATMS 585

B

BDD 659
Benchmarking 673
Bounded PlanSAT 663

C

CNF 397, 522
co-NP-задачи 676
co-NP-полные задачи 676
CPM 650
CSP 319
Cybernetics 46

D

DAC 350
Datalog 504, 540
DPLL 407

E

EI 494

H

HLA 622
  ангельская семантика 628
  достижимое множество 628
  приблизительные описания 630
  реализация 623
  уточнение 622
HLAI 72
HTN 622
  примитивное действие 622
  High-Level Action 622

J

JTMS 584

K

k-совместимость 332

L

Lisp 52
LRTA* 247

M

MA* 172
MAC 342
MGU 499
Modus Ponens 392
  обобщенное 497
MRV 340

N

NP-задачи 675
NP-полнота 36
NP-полные задачи 676
NP-сложные задачи 676

O

OOP 575

P

P-задачи 675
PDDL 604
  предусловие 605
  состояние 604
  схема
    восприятия 635
    действий 604
  эффект 605
PlanSAT 663
Prolog 514

S

SMA* 172

U

UCB1 285
UCT 285
UI 494

А

Абстрактный план 632
Абстракция 129, 299
  состояний 620
Автоматическая сборка 136
Автоматические рассуждения 25
Автономность 89
Автономные вычисления 118
Агент 27
  автономность 89
  архитектура 98
  безмодельный 109
  внутреннее состояние 103
  для частично наблюдаемых сред 237
  обучающийся 110
    вознаграждение 112
    компоненты 110
    штраф 112
  обучение 89
  основанный
    на знаниях 369
    на логике высказываний 414
    на модели 104
  показатель производительности 85
  практичный 108
  программа 97
  программный 92
  простой рефлекторный 100
  рациональный 28, 85
  решающий задачи 125
  целенаправленный 106
Агрегирование 649
Аксиома 370, 465
  вневременная 420
  исключения действия 427
  определения преемника 418, 668
  предусловий 426
  фреймов 417
  эффектов 417
Алгебра Роббинса 543
Алгоритм 35
  АС-3 329
  восхождения к вершине 206
  генетический 213
  Дэвиса–Патнема 407
  имитации отжига 211
  итеративного развертывания 193
  логического вывода
    непротиворечивый 380
    проверка по моделям 380
  локального поиска 410
  Метрополиса 252
  обзорного распространения 432
  обратного логического вывода 513
  обратного перехода, управляемый конфликтами 345
  поддержки дуговой совместимости 342
  поиска 137
    минимаксный 266
    с возвратами полный 407
  показатели производительности 143
  полнота 381
  прямого логического вывода 403, 506
    инкрементный 510
  резолюции 398
  систематический 144
  эволюционный 213
  DPLL 408
  rete 511
  SATPLAN 426
  WALKSAT 411
Альфа-бета-отсечение 270
Альфа-бета-поиск 300
Альянс 269
Анализ
  алгоритмов 673
  асимптотический 674
  сложности 675
Ангельский недетерминизм 628
Аргумент противника 242
Арность 451, 507
Архитектура агента 98
Асимптотический анализ 674
Атомарное высказывание 454
Атрибут, значение 114

Б

База
  данных
    допущения 462
    с непересекающимися шаблонами 182
    с шаблонами 181
    семантика 461
  знаний 370, 387
    индексация 501

монотонность 394
сложность данных 509
упорядочение конъюнктов 508
фиксированная точка 506
Эрбрана 530
Байесовская сеть 61
Безмодельный агент 109
Безопасно исследуемое пространство состояний 242
Бесконечная область определения 324
Бессенсорная задача 228, 232
Бессенсорное планирование 634
Бинарная
диаграмма решений 659
задача УО 325
резолюция 524
Бинарное ограничение 325
Бихевиоризм 42
Большие данные 61
Быстрый поиск 167

В

Вектор 677
Верификация 537
Вероятностное отсечение 281
Вероятность 679
Верхняя онтология 552, 553
Вершинная совместимость 328
Вещество 562
свойства 563
Взвешенная линейная функция 277
Визуализация, работы мозга 26
Вневременная переменная 416
Возможные миры 570
Восприятие 82
возможное 234
Восхождение к вершине
с выбором первого варианта 210
со случайным перезапуском 210
стохастическое 210
Временная
логика 448
проекция 433
сложность алгоритма 143
Время, интервалы 566
Всезнание 88
Выборка 682
Выборочные продления 280
Выбор цели 126
Выполнение 126
Выполнимость 391
Выпуклая оптимизация 222
Выпуклое множество 222
Выражение
без заголовка 402
единичное 396, 408, 535
завершение 520
определенное 402, 503
факт 402
хорновское 402
Вырожденная матрица 678
Высказывание 370
аксиома 370
атомарное 382
возможный мир 377
выполнимость 391
допустимость 390
истинность 377
конъюнктивная нормальная форма 397
модель 378
отрицание 383
преобразование
в CNF 522
сложное 383
удовлетворение 378
унификатор 498
утверждения 463
факторизация 396
Вычислимость 36
Вычислительная лингвистика 48

Г

Генератор 686
Генетический алгоритм 213, 218
кроссинговер 214
процесс выбора 214
схема 218
функция приспособленности 214
частота мутации 214
элитарность и отбраковка 214
эффект Болдуина 217
Генетическое программирование 214, 253
Гиперграф ограничений 326
Гипотеза, Сепира–Уорфа 443
Глобальное ограничение 325, 332

Глобальный
максимум 206
минимум 206
Глубина оптимального решения 144
Глубокое обучение 62
для естественного языка 17
Головоломка
криптоарифметическая 325
Сокобан 132
с перемещением плиток 132
судоку 334
Гомеостатика 47
Градиент ландшафта 220
Градиентный спуск 206
Грамматика 684
контекстно-свободная 684
Граф 129
И–ИЛИ 403
ограничений 322, 350
связные компоненты 349
симметрия значений 355
планирования 614
пространства состояний 263

Д

Датчик 81
Двойственный граф 326
Двунаправленный поиск 154
Дедуктивные базы данных 512
Дедуктивный синтез 537
Действие 128, 229
базовое 605
высокого уровня (HLA) 622, 628
уточнение 622
недетерминированное 628
применимость 128
примитивное 622
пропозиционализация 613
резерв времени 651
результат 605
релевантное 612
чередование из субпланов 657
Декларативный подход 372
Декомпозиция 620
иерархическая 622
Демодуляция 534, 542
Демонический недетерминизм 628
Деонтологическая этика 34
Дерево
игры 263
поиска 263
Диаметр графа 151
Дизъюнктивное ограничение 323
Дизъюнкция 383, 579
Динамическое программирование 519
Диофантовы уравнения 357
Дискретизация 219
Дисперсия 682
Доверительное состояние 222, 229, 294, 298, 422
обновление 234
Доказательство опровержением 392
Доказуемая полезность 30
Доминирование 178
Допустимость 390, 453
эвристики 161
Допущение
об уникальности имен 462
о замкнутости мира 462
Достигнутое состояние 138
Дочерний узел 137
Древовидная декомпозиция 353
ширина дерева 355
Дуализм 32
Дуговая совместимость 329
направленная 350

Е

Единичная
резолюция 396
матрица 678
Естественные разновидности 559
Естественный язык, обработка 25

Ж

Жадный поиск
локальный 208
по первому наилучшему совпадению 157

З

Завершение 520
Задача
бессенсорная 228
декомпозиция 620
коммивояжера 135

компоновки СБИС 136
линейного программирования 325
навигации робота 136
независимость подцелей 620
о восьми ферзях 207
о покрытии множества 617
оптимизации 206
при ограничениях 327
ослабленная 179
отображения 241
планирования
обхода 135
производства 649
реальная 130
решение 129
совместимая с пределами 334
соответствия 228
состояния 128
стандартизированная 130
удовлетворения ограничений 319
поиск 337
упорядочиваемые подцели 619
формальное определение 128
формулировка 126
Bounded PlanSAT 663
PlanSAT 663
SAT 391
определение выполнимости 412
Закон Мура 44
Запрос 463
подстановка 464
Знания 551, 571
представление 25, 369
уровень 372
язык представления 370
Значение
заданное по умолчанию 576
переопределение 576

И

Игра 37
альянсы 269
позиция 262
полуход 265
с неполной информацией 262
с нулевой суммой 262
с полной информацией 262
стоимости материала 277
с элементами случайности 289
упорядочивание ходов 273
ход 262
частично наблюдаемая 293
Игры-помощники 74
Иерархическая сеть задач 622
Иерархическое
планирование 621
прогнозирование 634
Иерархия
абстракции 659
таксономическая 556
Избыточный путь 142
И–ИЛИ-дерево 224
ИЛИ-узел 224
Имитация отжига 211
Импликация 383
антецедент 383
двухсторонняя 383
консеквент 383
Инвертированное обучение с подкреплением 74
Индекс
искусственного интеллекта 63
объезда 165
Индексация 501
по ключам 502
по предикатам 501
решетка обобщения 502
Индивидуализация 562
Индукция 32
Инженерия знаний 472
процесс 473
Инкрементный поиск 249
в доверительных состояниях 233
Интеллект искусственный 23
на уровне человека 72
особый момент 41
Интеллектуальный агент 14
Интерпретация
в модели 451
предполагаемая 451
Интерфейс мозг–машина 41
Инфиксная система обозначений 467
Информированный поиск 155
Искусственная жизнь 253
Искусственный

интеллект общий 72
суперинтеллект 72
Исполнительный механизм 81
Использование состояний 284
Исследование
состояний 284
операций 38
Истинность 377
Исчерпывающая декомпозиция 557
Исчисление
ситуаций 614
событий 564
Итеративное углубление 274
И-узел 224

К

Карточные игры 298
Категория 555
исчерпывающая декомпозиция 557
наследование 556
непересекающиеся 557
подкатегории 556
сегментация 557
типичный экземпляр 560
Качественная физика 562, 593
Квантовые вычисления 45
Квантор 455
базовый терм 456
вложение 459
всеобщности 456
конкретизация высказывания 494
переменная 456
расширенная интерпретация 456
сворачивание 468
существования 457
конкретизация высказывания 494
эквивалентность 460
Кибернетика 46
Класс вариантов среды 97
Классификация 577
Классическое планирование 603
алгоритмы 609
Клеточный мир 131
Ковариационная матрица 682
Ковариация 682
Когнитивная
архитектура 512
наука 26, 43
психология 42
Коммутативность 337
Композициональность 442
Компонент
генератор проблем 111
действующий 110
критик 110
обучающий 110
Компьютерное зрение 25
Конечная область определения 324
Коннекционистская модель 59
Консеквенциализм 34, 85
Константа 451
сколемовская 495
Конструирование белка 137
Конструкция
if–then–else 224
while 227
Контекстно-свободная грамматика 684
Контроллер 117
Контроль
действий 645
плана 646
цели 645
Контур поиска 162
Конфликтный набор 343
Конформное планирование 634
Конъюнктивная нормальная форма 397, 522
Конъюнкция 383
Кортеж 449
Косвенное описание 580
с приоритетами 581
Коэффициент
ветвления 144
эффективный 177
конкурентоспособности 242
Кратчайший путь 184
Кригшпиль 294
вероятностный мат 295
гарантированный мат 295
случайный мат 297
стратегия 295
Критический путь 650
Кроссинговер 214
Кумулятивное распределение 680

Л

Ландшафт пространства состояний 206, 209, 348
  градиент 220
Лемма поднятия 529, 530
Линейное
  ограничение 324
  планирование 657
  программирование 221
Линейный поиск 220
Листовой узел 138
Литерал 383
  взаимно обратные 396
Логика 377
  высказываний 370, 382
    выражение 396
    доказательство теорем 390
    литерал 383
    пропозициональный символ 382
    таблица истинности 385
  высшего порядка 448
  модальная 570
  немонотонность 580
  нечеткая 447
  описательная 577
  первого порядка 441, 463
    атомарное высказывание 454
    запросы 463
    инженерия знаний 472
    кванторы 455
    логические связки 455
    модели 449
    определенные выражения 503
    пропозиционализация 495
    равенства 533
    символ равенства 460
    синтаксис 451
    терм 453
    унификация 498
    факторизация 525
    язык 446
  полуразрешимость 496
  предпочтения моделей 581
  следствие 378
  умолчания 581
  формальная 34, 445
Логицизм 27
Логическая
  минимизация 559
  связка 383, 455
  эквивалентность 390
Логический
  вывод 341, 371, 380
    доказательство 392
    логика первого порядка 493
    обоснование 382
    обратный 513
    правила 392
    правило отделения 392
    пропозициональный планирование 424
    процедура 388
    прямой 503
    прямой и обратный 402, 406
    резольвента 395
    сохранение истинности 380
  позитивизм 32
Логическое
  всеведение 572
  программирование 402, 514
    в ограничениях 358, 521
    табулированное 519
  следствие 453
Локализация 238
Локальная совместимость 328
Локальный
  максимум 208
  поиск 206, 219, 346
    в оперативном режиме 245
    по лучу 213

М

Магическое множество 512
Макрооператор 659
Манхэттенское расстояние 177
Математическое ожидание 681
Материал
  повторно используемый 649
  расходный 649
Материализм 32
Матрица 678
  вырожденная 678
  единичная 678
  ковариации 681
  обратная 678

Машинная эволюция 55
Машинное обучение 25
Мера 561, 562
Мереология 591
Метаправила 521
Метарассуждения 302
Метауровневое
  обучение 186
  пространство состояний 185
Метод
  ветвей и границ 192
  взвешивания ограничений 348
  критического пути 650
  Монте-Карло 283
  наблюдаемого литерала 431
  Ньютона-Рафсона 221
  обратного перехода 343
  проставление обратных отметок 360
  разрыва цикла 352
  резолюций 394
Микромир 53
Минимаксное
  значение 266
  решение 266
Минимаксный поиск 265
Мир
  блоков 53
  вампуса 373, 470
    база знаний 387
    гибридный агент 420
  клеточный 131
    пылесоса 131
Многомерное гауссово распределение 681
Множество 468
  поддержки 535
  разрыва цикла 352
Модальная логика 570
  квантификаторы 571
  линейная временная 572
  логическое всеведение 572
Модальный оператор 570
Модель 378
  вероятностная 679
  возможные миры 570
  интерпретация 451
  коннекционистская 59
  логика первого порядка 449
  мира 104
  отношения 449
  проблемная область 449
  сенсоров 104
  скрытая марковская 60
  стандартная 28
  функции 450
Момент времени 566
Мониторинг 237
Монотонность 163, 394
  логики 580
  эвристики 161
Мультиагентная система 118

Н

Наиболее крутой подъем 207
Направленная дуговая совместимость 350
Наследование 556, 574
  множественное 575
Насыщение 529
Натурализм 32
Натуральные числа 467
  аксиомы Пеано 467
Начальное состояние 128, 229
Начальный символ 684
Недетерминированные полиномиальные задачи 675
Независимая подзадача 349
Независимость подцелей 620
Нейрон 39
Нейронауки 38
Немонотонная логика 580
  логика умолчания 581
Немонотонность 580
Ненадежный мир пылесоса 226
Непредсказуемый мир пылесоса 223
Непрерывное планирование 643
  контроль выполнения 644
Непротиворечивость 380, 389, 577
Неравенство треугольника 161
Нетерминальный символ 684
Нечеткая логика 447
Нормальное распределение 680
Нотация О большое 673
Нулевая сумма 262

О

Область определения
дискретная 324
непрерывная 324
Обобщение 577
Обобщенное правило отделения 497
Обоснование в JTMS 584
Обработка естественного языка 25
Обратная матрица 678
Обратный
логический вывод 513, 541
минимаксный поиск 283
поиск 611
Обучение 89, 382
глубокое 62
основанное на объяснениях 659
при поиске в оперативном режиме 249
Объект 445, 568
кортеж 449
ментальный 569
пропозициональная установка 569
свойства 563
совокупность 559
составной 558
Обязательно развертываемые узлы 164
Овеществление 556
высказываний 576
Ограничение
глобальное 332
дизъюнктивное 323
на ресурсы 648
нарушающее симметрию 355
нелинейное 324
предпочтения 327
предшествования 323
распространение ограничений 328
ресурсное 333
типы 325
Alldiff 332
atmost 333
Ограниченная рациональность 29
Ожидаемая полезность 108
Ожидаемое
значение 276, 291
минимаксное значение 291
Окружающая среда 81
Онтологическая инженерия 552
Онтологический вклад 446
Онтология
верхняя 552
категории 555
общего назначения 554
проблемной области 474
специального назначения 554
хроники 590
хронологии 590
Оператор модальный 570
Операция 649
с предикатами 578
Описательная логика 573, 577
язык 577
Опорная точка 183
Определение 465
ограничений с помощью обучения 345
условий выбора множества разрыва цикла 353
Оптимальное решение 129
Оптимальность
затрат 143
по эффективности 164
Оптимизация при ограничениях 221
Оптимизм в отношении неопределенности 248
Оптимистическое описание 630
Оптогенетика 40
Ослабленная задача 179
Ослабленный план 619
Особый момент 41
Отбраковка 214
Отладка базы знаний 474, 480
Отношение 320, 445
Отношения доступности 570
Отрицание 579
высказывания 383
Отсечение 164, 262, 270
бесполезных вариантов 306
Оценка состояния 237, 294, 422
Очередь 141
с приоритетом 141
FIFO 141
LIFO 141

П

Парамодуляция 534, 542
Переименование 505
Переменная 114
  локальный поиск 219
  стандартизация 499
Перемещение
  в сторону 210
  связок ¬ 522
Переопределение 576
Перепланирование 644
  внешние события 644
  контроль 645
  отсутствующая флюента 644
  отсутствующее предусловие 644
Пересмотр убеждений 583
Перестановки 274
Периферия 138
  разделение графа 139
Перцептрон 54
Пессимистическое описание 630
План
  абстрактный 632
  время выполнения 649
  контроль выполнения 644
  ослабленный 619
Планирование 106
  абстракция состояний 620
  бессенсорное 634, 636
  булева выполнимость 613
  доверительное состояние 634, 637
    первое приближение 640
  задача
    начальное состояние 605
    цель 605
  иерархическая декомпозиция 622
  иерархическое 621
    абстрактное решение 626
    свойство нисходящего уточнения 627
  классическое 603
  конформное 660
  линейное 657
  непрерывное 634, 636, 643
  обратный поиск 612
  по прецедентам 660
  примеры 606
  прямой поиск 611
  реактивное 662
  сокращение симметрии 619
  составление расписаний 648
  с частичным упорядочиванием 615
  условное 634, 642
  формальное представление 604
  эвристика 616
    игнорирования предусловий 616
    с пустым списком удаления 617
  HTN 622
    библиотека планов 626
    действие Act 624
Плато 209, 348
Плотность вероятности 680
Подкатегория 556
Поднятая версия 498
Поднятие 498
Подстановка 464, 494
Поиск 106, 125, 126
  в автономном режиме 240
  в глубину 148
    с итеративным углублением 151
  в графе 142
  в И–ИЛИ дереве 224
  в оперативном режиме 240, 241
    агенты 243
    обучение 249
  восхождением к вершине 207, 210, 245
  в условиях противодействия 261
  в частично наблюдаемых средах 233
  в ширину 145
  двунаправленный 154
    спереди назад 174
    спереди на передний план 174
    эвристический 173
  жадный по первому наилучшему совпадению 157
  иерархический 192
  инкрементный в доверительных состояниях 233
  информированный 155
  контуры 162
  линейный 220
  локальный 206, 218
    в задачах УО 346
    по лучу 213
  минимаксный 265
  обратный минимаксный 283

отсечение 262
по дереву 143
методом Монте-Карло 283
по критерию стоимости 147
по лучу 168, 281
по методу Монте-Карло 301
по первому наилучшему совпадению 140
приближенный 167
рекурсивный
по первому наилучшему совпадению 169
с возвратами 150, 337
динамический 360
управляемый зависимостями 359
хронологический 343
с неограниченной стоимостью 167
с ограничением глубины 151
с ограниченной стоимостью 167
спокойных позиций 279
стохастический по лучу 213
упрощенный MA* 172
цикл 142
чистый по методу Монте-Карло 284
A* 158
с весовым коэффициентом 165
с итеративным углублением 168
с ограничением памяти 172
Показатель производительности 85
Полезность 34, 107
доказуемая 30
ожидаемая 108
Полиномиальные задачи 675
Полное присваивание 320
Полнота
алгоритма 143, 381
опровержения 528
резолюции 400
Полуразрешимость 496
Полуход 265
Портфолио алгоритмов 654
Последовательность восприятия 82
Постановка цели 126
Правило
единичной резолюции 395
замыкания проблемной области 462
отделения 392
обобщенное 497
по умолчанию 582
подстановки 685
резолюций 395, 396, 524
удаления связки 392
условие–действие 101
if–then 383
Предварительная проверка 341
Предвычисление 183
Предикат, арность 451, 507
Предпочтение моделей 581
Предпочтительное действие 619
Представление
атомарное 113
знаний 25, 48, 369
развернутое 114
атрибуты 114
Предусловие 605
Преемник 137
Префиксная система обозначений 467
Приближенный поиск
с органичениями 167
Применимость 128
Принудительный перевод 229
Принцип индукции 32
Принятые обозначения 15
Приобретение знаний 473
Присваивание 320
конфликтный набор 343
Проблема
выравнивания ценностей 30
гориллы 73
квалификации 419
логического фрейма 418
представления фреймов 418
фреймов 417, 433
царя Мидаса 74
Проблемная
область 449, 463
аксиомы 465
онтология 474
определения 465
правило замыкания 462
примеры 464, 467–470, 475
теоремы 465
среда 90
детерминированная 94
динамическая 95
дискретная 95
известная 95

класс вариантов 97
мультиагентная 93
конкурентная 93
кооперативная 93
недетерминированная 94
неизвестная 95
непрерывная 95
одноагентная 93
полностью наблюдаемая 93
полудинамическая 95
последовательная 94
статическая 95
стохастическая 94
частично наблюдаемая 93
эпизодическая 94
PEAS 90
Пробуксовка 173
Проверка
вхождения 500
по моделям 380
схемы 480
терминального состояния 263
цели 230
Прогнозирование 234
Прогон 284
Программа агента 83, 97
Программирование
динамическое 519
логическое 514
Программный бот 92
Продолжительность действия 649
Продукционная система 511, 540
Пропозиционализация 495
Пропозициональная установка 569
Пропозициональный символ 382
Просмотр таблицы 282
Простой рефлекторный агент 100
Пространственная сложность алгоритма 143
Пространственные рассуждения 594
Пространство
доверительных состояний 229
состояний 128
безопасно исследуемое 242
ландшафт 206
метауровневое 185
объектного уровня 185
Протокольное предложение 32
Процедурное вложение 576
Процедурный подход 372
Прямое отсечение 281
Прямой логический вывод 503
магическое множество 512
Прямой поиск 610
Псевдокод 685
списки 687
циклы 687
Психологические рассуждения 594
Психологический эксперимент 26
Путевая совместимость 331
Путь 129
избыточный 142

Р

Равенство
аксиоматизация 533
демодуляция 533
Равномерное распределение 682
Развертывание узла 137
Размер шага 220
Разрешимость 36
Разумная достаточность 38
Рандомизация 103
Раннее завершение прогона 288
Ранняя проверка достижения цели 145
Расписание 651
критический путь 650
Распределение
Гаусса 680
кумулятивное 680
многомерное гауссово 681
нормальное 680
стандартное 681
равномерное 682
с тяжелым хвостом 252
Распространение
единичных выражений 408
ограничений 328
пределов 334
Расстояние
в городских кварталах 176
манхэттенское 177
по прямой 157
Рассуждения 369

автоматические 25
направляемые целями 406
о категориях 573
пространственные 594
психологические 594
ссылочная прозрачность 570
управляемые данными 405
Расходный материал 649
Расширение 582
Рациональность 24
ограниченная 29
Рациональный агент 28, 85
определение 86
Реализация HLA 623
действие Act 624
Реальная задача 130
Регрессивный поиск 611
Регрессия 612
Резерв времени 651
Резервирование 267
Резольвента 395, 524
Резолюционное замыкание 400, 530
Резолюция 521
бинарная 524
единичная 535
использование единичных выражений 535
линейная 536
обобщение 536
обучение 536
основная теорема 401, 528
полнота 400, 528
с входными высказываниями 536
стратегии выполнения 535
Результат, список
добавления 605
удаления 605
Рекомбинация 213
Рекурсивный поиск
по первому наилучшему совпадению 169
Ресурсное ограничение 333
Решение 126, 129
оптимальное 129
субоптимальное 165
частичное 320
Решетка обобщения 502
Робототехника 25
Родительский узел 137

С

Самоанализ 25
Сбор информации 88
Свойства внешние и внутренние 563
Связные компоненты 349
Сегментация 557
Семантика 377
ангельская 628
базы данных 462
истинностное значение 384
Семантическая сеть 573
наследование 574
фреймы 592
Силлогизм 27, 430
Символ 684
константы 451
предиката 451
функции 451
Симметрия значений 355
Симуляция 284
Синтаксис 377
логики высказываний 382
Синтаксическое упрощение 468
Синтез 537
Система
без обратной связи 127
обеспечения истинности 359
обозначений
инфиксная 467
префиксная 467
синтаксическое упрощение 468
поддержки истинности 584
объяснения 585
предположения 585
ATMS 585
JTMS 584
с обратной связью 127
Сколемизация 523
Сколемовская
константа 495
функция 523
Скрытая марковская модель 60
Слабые методы 56
Следствие 378
Сложность данных 509
Случайное блуждание 246
Событие 564, 565

Совместимое присваивание 320
Совместимость 320
  вершин 328
  дуговая 329
  кратности k 332
  локальная 328
  путевая 331
Совокупность 559
Согласование с шаблоном 508
Сокращение
  последних ходов 281
  симметрии 619
Составление расписания 648
Составные объекты 558
Состояние 128
  атомарное представление 113
  достигнутое 138
  локализованное представление 115
  начальное 128
  повторяющееся 142
  представление
    выразительность 115
  развернутое представление 114
  распределенное представление 115
  характеристики 186
  целевое 128
Список 469
  добавления 629
  связывания 464
  удаления 629
Спокойная позиция 279
Способность поступать правильно 28
Среда
  недетерминированная 223
  ненаблюдаемая 93
  частично наблюдаемая 234
Среднее квадратическое 682
Ссылочная прозрачность 570
Стандартизация 499
  переменных 523
Стандартизированная задача 130
Стандартная модель 28
Стандартное
  нормальное распределение 681
  отклонение 682
Статистика 35
Стек 141
Степенная эвристика 340
Степень
  доверия 448
  истинности 447
Стоимость действия 230
Стохастические игры 289
Стохастический поиск по лучу 213
Стохастическое восхождение
    к вершине 210
Стратегия 283
  выбора 284
  прогонов 284
  типы 275
Строгая k-совместимость 332
Структура задач УО 349
Судоку 334
Существительные, исчисляемость 562
Схема
  восприятия 635
  действий 604
  примеры 218
Счетчик ссылок 168

T

Таблица
  истинности 385
  перестановок 274
Тавтология 390
Таксономия 556
Теорема 465
  дедукции 390
  критической точки выполнимости 413
  о неполноте 36, 531
  основная резолюции 401, 528
  центральная предельная 681
  Эрбрана 530
Теория
  вероятности 27, 35
  игр 38, 261
  подтверждения 32
  принятия решений 37
  управления 46
  NP-полноты 36
Терм 453
  базовый 456
Терминальное состояние 263
Терминальный символ 684

Тест
  останова 275
  Тьюринга 24
    общий 25
  эталонный 673
Типичный экземпляр 560
Топологическая сортировка 350
Транспонирование 678
Тупик 242, 345

У

Удаление кванторов всеобщности 523
Узел 137
  дочерний 137
  жеребьевки 290
  листовой 138
  развертывание 137
  родительский 137
Унарное ограничение 325
Универсум Эрбрана 529
Унификатор 498
  наиболее общий 499
Унификация 498, 539
  с учетом равенства 534
Упорядочение конъюнктов 508
Упорядочиваемые подцели 619
Уровень
  абстракции 130
  знаний 372
  реализации 372
Усиление интеллекта 43
Условное планирование 634, 642
Условный
  план 222, 224
  эффект 639
Усреднение по ясновидению 298
Устранение импликаций 522
Утверждение 463
  степень истинности 447
Утилитаризм 34
Уточнение HLA 622
  реализация 623

Ф

Факт 402
  переименование 505
Фактор уверенности 57
Факторизация 396, 525
Физическая символическая система 51
Фиксированная точка 506
Фильтрация 237
Флюента 564, 416, 568
  отсутствующая 644
Форма 1-CNF 423
  первое приближение 423
  Бэкуса–Наура 684
Формальная логика 34
Формирование узла 137
Формулировка
  задачи 126
  полного состояния 207
Фрейм 58
Функция 445
  агента 82
  арность 451
  единиц измерения 561
  определения преемника 128, 224, 230, 263, 416
  оценки 262, 275, 291
  ошибок 681
  плотности вероятности 680
  полезности 107, 263
  полностью определенная 450
  приспособленности 214
  стоимости действия 128
  целевая 206
  эвристическая 155, 176

Х

Характеристики состояния 186, 276
Хеббовское обучение 49
Ходы-убийцы 274
Хорновское выражение 402
  тело и голова 402
Хребет 209

Ц

Целевая функция 47, 206
Целевое состояние 128
Цель 605
  периметр 657
  пропозиционализация 614
Центральная предельная теорема 681
Цикл 142
Циклическое решение 227

Ч

Частичная наблюдаемость 293
Частичное присваивание 320
Частота мутации 214

Ш

Ширина
  гипердерева 361
  дерева 355

Э

Эволюционные алгоритмы 213
Эвристика
  доминирование 178
  допустимая 161
  единичного выражения 408
  игнорирования предусловий 616
  минимального резерва 653
  минимальных конфликтов 346
  преемственность 161
  пустого хода 306
  разностная 184
  с наименее ограничительным значением 340
  с ориентирами 183
  с пустым списком удаления 617
  убийцы 274
  чистого символа 407
  MRV 340, 508
Эвристическая функция 155
  оценки 276
Эквивалентность 383
  логическая 390
Экзистенциальный граф 573
Экономика 261
Экспертная система 57
Элемент проблемной области 449
Элитарность 214
Эмпиризм 32
Эмпирический градиент 220
Эпистемологический вклад 448
Эталонный тест 673
Эффект 605
  Болдуина 217
  горизонта 279
  отсутствующий 644
  условный 639
Эффективная глубина 177
Эффективный коэффициент ветвления 177

Я

Язык
  декларативный 442
  логики первого порядка 446
  мышления 443
  онтологический вклад 446
  представления 441
  представления знаний 370
  синтаксис 377
  элементы
    объект 445
    отношения 445
  эпистемологический вклад 448
  CLASSIC 577
  Datalog 504, 540
  Lisp 52
  MRS 521
  PDDL 604
    состояние 604
  Prolog 58, 514, 541
    особенности 516
    семантика базы данных 519

Самое полное и актуальное введение в теорию и практику искусственного интеллекта!

В четвертом, обновленном, пересмотренном и дополненном издании этой книги область искусственного интеллекта (ИИ) исследуется и анализируется во всей ее обширности и глубине. Здесь представлены все современные достижения и изложены идеи, которые были сформулированы в исследованиях, проводившихся в течение последних пятидесяти лет, а также собраны на протяжении двух тысячелетий в областях знаний, ставших стимулом к развитию ИИ как науки. Предыдущие издания этой книги стали классическими образцами литературы по ИИ и приняты в качестве учебного пособия более чем в 1400 университетах 128 стран мира, где были высоко оценены как убедительный итог обобщения результатов, достигнутых в этой области науки.
Книга дополнена обширным набором интернет-ресурсов, включая упражнения, программные и исследовательские проекты, реализации алгоритмов, а также дополнительные материалы и ссылки для студентов и преподавателей.

Что нового в четвертом издании

В четвертом издании читатель познакомится с новейшими технологиями и концепциями, представленными в более унифицированном виде с новым или расширенным охватом таких тем, как машинное обучение, глубокое обучение, трансферное обучение, многоагентные системы, робототехника, обработка естественного языка, проблема причинности, вероятностное программирование, а также конфиденциальность, беспристрастность и безопасность ИИ.

Что представлено в этом томе

В связи с исключительно большим объемом этого энциклопедического издания его перевод на русский язык представлен в виде трехтомника. В текущем томе, получившем подзаголовок *Решение проблем: знания и рассуждения*, излагаются основы основ ИИ — общая характеристика этой области как науки об интеллектуальных агентах, действующих в различных средах и решающих различные задачи на основании знаний, результатов восприятия, рассуждений и планирования.

Что в других томах

В двух последующих томах, II и III, охвачены аспекты действия агентов в окружении, более приближенном к реальному: знания, рассуждения и принятие решений в условиях неопределенности, в том числе в многоагентной среде (том II), а также методы и возможности их обучения, совершенствования восприятия и реализации в виде автономных устройств — роботов и роботизированных систем, включая такие важные социальные аспекты, как справедливость, доверие, общественное благо и безопасность (том III).

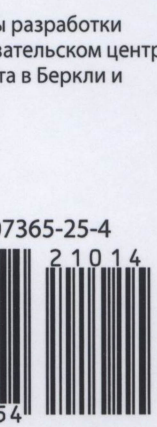
ОБ АВТОРАХ

Стюарт Рассел, профессор компьютерных наук Калифорнийского университета в г. Беркли, адъюнкт-профессор неврологической хирургии Калифорнийского университета в г. Сан-Франциско, основатель и директор центра Center for Human-Compatible AI и заведующий кафедрой инженерного искусства Смита–Задэ в Калифорнийском университете в г. Беркли.

Питер Норвиг, директор по исследованиям в компании Google, Inc., прежде — глава ее группы разработки основных алгоритмов веб-поиска. До этого руководил отделом компьютерных наук в Исследовательском центре Эймса, НАСА, был профессором в Университете Южной Калифорнии и членом совета факультета в Беркли и Станфорде.
Авторы этой книги удостоены первой награды AAAI/EAAI "Выдающийся педагог" в 2016 году.

Загрузите дополнительные материалы к книге с сайта издательства по адресу:
http://www.williamspublishing.com/Books/978-5-907365-25-4.html

ISBN 978-5-907365-25-4







www.dialektika.com



www.pearson.com