O'REILLY®

ВТОРОЕ ИЗДАНИЕ

АЛГОРИТМЫ СПРАВОЧНИК

С ПРИМЕРАМИ НА C, C++, JAVA И PYTHON

Джордж Хайнеман,
Гэри Поллис, Стэнли Селков

Алгоритмы
Справочник

Второе издание

Algorithms in a Nutshell

Second Edition

George T. Heineman,
Gary Pollice
& Stanley Selkow

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Алгоритмы
Справочник

С ПРИМЕРАМИ НА C, C++, JAVA И PYTHON

Второе издание

Джордж Хайнеман
Гэри Поллис
Стэнли Селков

Диалектика

Москва · Санкт-Петербург · Киев
2017

ББК 32.973.26-018.2.75
Х15
УДК 681.3.07

Издательство "Диалектика"
Зав. редакцией *С.Н. Тригуб*
Перевод с английского и редакция канд. техн. наук *И.В. Красикова*

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

Хайнеман, Джордж, **Поллис**, Гэри, **Селков**, Стэнли.
Х15 Алгоритмы. Справочник с примерами на C, C++, Java и Python, 2-е изд. : Пер. с англ. — СпБ. : ООО "Альфа-книга", 2017. — 432 с. : ил. — Парал. тит. англ.
ISBN 978-5-9908910-7-4 (рус.)

ББК 32.973.26-018.2.75

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 O'Reilly & Associates.

Authorized Russian translation of the English edition of *Algorithms in a Nutshell*, 2nd edition (ISBN 978-1-491-94892-7) © 2016 George Heineman, Gary Pollice and Stanley Selkow.

This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the Publisher.

Научно-популярное издание
Джордж Хайнеман, Гэри Поллис, Стэнли Селков

Алгоритмы
Справочник с примерами на C, C++, Java и Python
2-е издание

Литературный редактор *Л.Н. Красножон*
Верстка *О.В. Мишутина*
Художественный редактор *В.Г. Павлютин*
Корректор *Л.А. Гордиенко*

Подписано в печать 12.04.2017. Формат 70x100/16.
Гарнитура Times.
Усл. печ. л. 27,0. Уч.-изд. л. 23,3.
Тираж 300 экз. Заказ № 2767.

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 "Альфа-книга", 195027, Санкт-Петербург, Магнитогорская ул., д. 30

ISBN 978-5-9908910-7-4 (рус.)
ISBN 978-1-491-94892-7 (англ.)

© 2017 Компьютерное издательство "Диалектика", перевод, оформление, макетирование
© 2016 George Heineman, Gary Pollice and Stanley Selkow

Оглавление

Предисловие ко второму изданию 15

Глава 1. Мысли алгоритмически 21

Глава 2. Математика алгоритмов 29

Глава 3. Строительные блоки алгоритмов 57

Глава 4. Алгоритмы сортировки 77

Глава 5. Поиск 119

Глава 6. Алгоритмы на графах 165

Глава 7. Поиск путей в ИИ 205

Глава 8. Алгоритмы транспортных сетей 257

Глава 9. Вычислительная геометрия 289

Глава 10. Пространственные древовидные структуры 331

Глава 11. Дополнительные категории алгоритмов 375

Глава 12. Эпилог: алгоритмические принципы 397

Приложение. Хронометраж 409

Литература 421

Предметный указатель 425

Содержание

Предисловие ко второму изданию 15
Изменения во втором издании 15
Целевая аудитория 16
Соглашения, используемые в данной книге 17
Использование примеров кода 18
Благодарности 18
Об авторах 19
Об изображении на обложке 19
Ждем ваших отзывов! 20

Глава 1. Мысли алгоритмически 21
Понимание проблемы 21
Прямое решение 23
Интеллектуальные подходы 23
Жадный подход 24
Разделяй и властвуй 25
Параллельное вычисление 25
Приближение 25
Обобщение 26
Резюме 28

Глава 2. Математика алгоритмов 29
Размер экземпляра задачи 29
Скорость роста функций 30
Анализ наилучшего, среднего и наихудшего случаев 34
Наихудший случай 37
Средний случай 37
Наилучший случай 38
Нижняя и верхняя границы 39
Семейства производительности 40
Константное поведение 40
Логарифмическое поведение 41

Сублинейное поведение 43
Линейная производительность 44
Линейно-логарифмическая производительность 47
Квадратичная производительность 47
Менее очевидные производительности 49
Экспоненциальная производительность 52
Резюме по асимптотическому росту 53
Эталонные операции 53

Глава 3. Строительные блоки алгоритмов 57

Формат представления алгоритма 58
Название и краткое описание 58
Входные и выходные данные алгоритма 58
Контекст применения алгоритма 58
Реализация алгоритма 58
Анализ алгоритма 58
Вариации алгоритма 59
Формат шаблона псевдокода 59
Формат эмпирических оценок 60
Вычисления с плавающей точкой 60
Производительность 61
Ошибка округления 61
Сравнение значений с плавающей точкой 62
Специальные значения 64
Пример алгоритма 64
Название и краткое описание 65
Входные и выходные данные алгоритма 65
Контекст применения алгоритма 65
Реализация алгоритма 66
Анализ алгоритма 69
Распространенные подходы 69
Жадная стратегия 69
Разделяй и властвуй 70
Динамическое программирование 71

Глава 4. Алгоритмы сортировки 77

Терминология 77
Представление 78
Сравниваемость элементов 79
Устойчивая сортировка 80
Критерии выбора алгоритма сортировки 81
Сортировки перестановкой 81
Сортировка вставками 81

Контекст применения алгоритма 83
Реализация алгоритма 83
Анализ алгоритма 84
Сортировка выбором 86
Пирамидальная сортировка 87
Контекст применения алгоритма 91
Реализация алгоритма 91
Анализ алгоритма 93
Вариации алгоритма 93
Сортировка, основанная на разбиении 93
Контекст применения алгоритма 97
Реализация алгоритма 98
Анализ алгоритма 99
Вариации алгоритма 99
Сортировка без сравнений 101
Блочная сортировка 101
Реализация алгоритма 103
Анализ алгоритма 106
Вариации алгоритма 107
Сортировка с использованием дополнительной памяти 109
Сортировка слиянием 109
Входные и выходные данные алгоритма 110
Реализация алгоритма 110
Анализ алгоритма 111
Вариации алгоритма 112
Результаты хронометража для строк 114
Анализ методов 116

Глава 5. Поиск 119

Последовательный поиск 120
Входные и выходные данные алгоритма 121
Контекст применения алгоритма 121
Реализация алгоритма 122
Анализ алгоритма 123
Бинарный поиск 124
Входные и выходные данные алгоритма 124
Контекст применения алгоритма 124
Реализация алгоритма 125
Анализ алгоритма 127
Вариации алгоритма 128
Поиск на основе хеша 129
Входные и выходные данные алгоритма 131
Контекст применения алгоритма 131
Реализация алгоритма 135

Анализ алгоритма 137
Вариации алгоритма 140
Фильтр Блума 146
Входные и выходные данные алгоритма 147
Контекст применения алгоритма 148
Реализация алгоритма 148
Анализ алгоритма 149
Бинарное дерево поиска 150
Входные и выходные данные алгоритма 151
Контекст применения алгоритма 152
Реализация алгоритма 153
Анализ алгоритма 163
Вариации алгоритма 163

Глава 6. Алгоритмы на графах 165

Графы 166
Проектирование структуры данных 169
Поиск в глубину 169
Входные и выходные данные алгоритма 173
Контекст применения алгоритма 174
Реализация алгоритма 174
Анализ алгоритма 175
Вариации алгоритма 175
Поиск в ширину 175
Входные и выходные данные алгоритма 177
Контекст применения алгоритма 178
Реализация алгоритма 178
Анализ алгоритма 179
Кратчайший путь из одной вершины 180
Входные и выходные данные алгоритма 183
Реализация алгоритма 183
Анализ алгоритма 184
Алгоритм Дейкстры для плотных графов 185
Вариации алгоритма 188
Сравнение вариантов поиска кратчайших путей из одной вершины 191
Данные хронометража 191
Плотные графы 192
Разреженные графы 192
Кратчайшие пути между всеми парами вершин 193
Входные и выходные данные алгоритма 193
Реализация алгоритма 195
Анализ алгоритма 198

Алгоритмы построения минимального остовного дерева 198
Входные и выходные данные алгоритма 200
Реализация алгоритма 200
Анализ алгоритма 202
Вариации алгоритма 202
Заключительные мысли о графах 202
Вопросы хранения 203
Анализ графа 203

Глава 7. Поиск путей в ИИ 205

Дерево игры 206
Функции статических оценок 209
Концепции поиска путей 210
Представление состояния 210
Вычисление доступных ходов 211
Максимальная глубина расширения 211
Minimax 211
Входные и выходные данные алгоритма 213
Контекст применения алгоритма 214
Реализация алгоритма 214
Анализ алгоритма 217
NegMax 217
Реализация алгоритма 219
Анализ алгоритма 221
AlphaBeta 221
Реализация алгоритма 225
Анализ алгоритма 227
Деревья поиска 229
Эвристические функции длины пути 232
Поиск в глубину 232
Входные и выходные данные алгоритма 233
Контекст применения алгоритма 234
Реализация алгоритма 234
Анализ алгоритма 236
Поиск в ширину 238
Входные и выходные данные алгоритма 239
Контекст применения алгоритма 240
Реализация алгоритма 240
Анализ алгоритма 242
A*Search 242
Входные и выходные данные алгоритма 244
Контекст применения алгоритма 244
Реализация алгоритма 247

Анализ алгоритма 251
Вариации алгоритма 252
Сравнение алгоритмов поиска в дереве 254

Глава 8. Алгоритмы транспортных сетей 257

Транспортная сеть 258
Максимальный поток 260
Входные и выходные данные алгоритма 261
Реализация алгоритма 261
Анализ алгоритма 270
Оптимизация 270
Связанные алгоритмы 273
Паросочетания в двудольном графе 274
Входные и выходные данные алгоритма 274
Реализация алгоритма 275
Анализ алгоритма 278
Размышления об увеличивающих путях 278
Поток минимальной стоимости 283
Перегрузка 284
Реализация алгоритма 285
Перевозка 285
Реализация алгоритма 287
Назначение 287
Реализация алгоритма 287
Линейное программирование 287

Глава 9. Вычислительная геометрия 289

Классификация задач 290
Входные данные 290
Вычисления 292
Природа задачи 293
Предположения 293
Выпуклая оболочка 294
Сканирование выпуклой оболочки 295
Входные и выходные данные алгоритма 296
Контекст применения алгоритма 297
Реализация алгоритма 298
Анализ алгоритма 300
Вариации алгоритма 301
Вычисление пересечения отрезков 304
LineSweep 305
Входные и выходные данные алгоритма 306
Контекст применения алгоритма 306

Реализация алгоритма 308
Анализ алгоритма 313
Вариации алгоритма 316
Диаграмма Вороного 316
Входные и выходные данные алгоритма 322
Реализация алгоритма 322
Анализ алгоритма 329

Глава 10. Пространственные древовидные структуры 331

Запросы ближайшего соседа 332
Запросы диапазонов 333
Запросы пересечения 333
Структуры пространственных деревьев 334
k-d-деревья 334
Дерево квадрантов 335
R-дерево 336
Задача о ближайшем соседе 337
Входные и выходные данные алгоритма 339
Контекст применения алгоритма 339
Реализация алгоритма 340
Анализ алгоритма 343
Вариации алгоритма 348
Запрос диапазона 348
Входные и выходные данные алгоритма 349
Контекст применения алгоритма 349
Реализация алгоритма 350
Анализ алгоритма 352
Деревья квадрантов 355
Входные и выходные данные алгоритма 357
Реализация алгоритма 357
Анализ алгоритма 361
Вариации алгоритма 361
R-деревья 362
Входные и выходные данные алгоритма 366
Контекст применения алгоритма 366
Реализация алгоритма 366
Анализ алгоритма 372

Глава 11. Дополнительные категории алгоритмов 375

Вариации на тему алгоритмов 375
Приближенные алгоритмы 376
Контекст применения алгоритма 378
Реализация алгоритма 378
Анализ алгоритма 380

Параллельные алгоритмы 382
Вероятностные алгоритмы 387
Оценка размера множества 388
Оценка размера дерева поиска 390

Глава 12. Эпилог: алгоритмические принципы 397

Используемые данные 397
Разложение задачи на задачи меньшего размера 398
Выбор правильной структуры данных 400
Пространственно-временной компромисс 401
Построение поиска 402
Приведение задачи к другой задаче 403
Тестировать алгоритмы труднее, чем писать 404
Допустимость приближенных решений 406
Повышение производительности с помощью параллелизма 406

Приложение. Хронометраж 409

Статистические основы 409
Пример 411
Решение для Java 411
Решение для Linux 412
Хронометраж с использованием Python 416
Вывод результатов 417
Точность 419

Литература 421

Предметный указатель 425

Предисловие ко второму изданию

Пересматривать содержание книги для нового издания всегда сложно. Мы старались сохранить все достоинства первого издания, опубликованного в 2009 году, но при этом исправить его недостатки и добавить новые материалы. Мы по-прежнему следовали принципам, изложенным в первом издании.

- Использовать для описания алгоритмов только реальный код, а не псевдокод.
- Отделять алгоритм от решаемой им задачи.
- Использовать только необходимое количество математических выкладок, и не более того.
- Сопровождать математический анализ эмпирическими данными.

Во втором издании мы сократили текстовые описания и упростили макет книги, чтобы освободить место для новых алгоритмов и дополнительных материалов. Мы считаем, что, как и ранее, нам удалось достаточно полно рассказать читателям о важной области информатики, которая оказывает значительное влияние на практические программные системы.

Изменения во втором издании

При обновлении предыдущего издания мы руководствовались следующими принципами.

Выбор новых алгоритмов

После публикации первого издания мы часто получали письма с комментариями наподобие "А почему в книге нет сортировки слиянием?" или "Почему вы ничего не рассказали о быстром преобразовании Фурье?" Все запросы удовлетворить попросту невозможно, но мы сумели добавить во второе издание несколько новых алгоритмов.

- **Алгоритм Форчуна** для вычисления диаграммы Вороного для множества точек (см. раздел "Диаграмма Вороного" главы 9, "Вычислительная геометрия").
- **Сортировка слиянием** как для внутренней памяти, так и для внешних файлов (см. раздел "Сортировка слиянием" главы 4, "Алгоритмы сортировки").
- Многопоточная версия **быстрой сортировки** (см. раздел "Параллельные алгоритмы" главы 11, "Дополнительные категории алгоритмов").
- Реализация **сбалансированных бинарных AVL-деревьев** (см. раздел "Бинарное дерево поиска" главы 5, "Поиск").
- Новая глава — глава 10, "Пространственные древовидные структуры", содержащая описания **R-Trees** и **Quadtrees**.

В целом сейчас в книге охвачено около 40 важных алгоритмов.

Упорядоченное представление

Чтобы освободить место для нового материала, мы пересмотрели почти каждый аспект первого издания. Мы упростили шаблон, используемый для описания каждого алгоритма, и сократили сопутствующие описания алгоритмов.

Добавление реализаций на языке Python

Вместо того чтобы переписывать существующие алгоритмы на языке Python, мы преднамеренно использовали Python для реализации большинства добавленных вновь алгоритмов.

Управление кодом

Исходные тексты для первого издания были представлены в виде ZIP-файла. С тех пор мы перешли к хранилищу GitHub (`https://github.com/heineman/algorithms-nutshell-2ed`). За прошедшие годы мы улучшили качество кода и документации. Мы также включили ряд записей из блога, которые были написаны после публикации первого издания. Кроме того, добавлено более 500 модульных контрольных примеров. В целом весь представленный в репозитории код состоит более чем из 110 тысяч строк.

Целевая аудитория

Мы позиционируем эту книгу как основной источник при поиске практической информации о том, как реализовать или использовать тот или иной алгоритм. Мы охватываем широкий диапазон существующих алгоритмов для решения большого количества проблем и при этом придерживаемся следующих принципов.

- При описании каждого алгоритма мы используем шаблон для единообразного описания и пояснения важных мест каждого алгоритма.

- Мы используем различные языки программирования для реализации каждого алгоритма (включая C, C++, Java и Python). При этом мы обсуждаем конкретные реализации алгоритмов на языках, с которыми вы знакомы.
- Мы описываем ожидаемую производительность каждого алгоритма и предоставляем эмпирические доказательства наших утверждений.

Мы писали книгу так, чтобы она была наиболее полезной для практиков программирования — программистов и проектировщиков программного обеспечения. Для достижения своих целей вам необходим доступ к качественному ресурсу, который подсказывает реальные реализации практических алгоритмов, которые нужны для решения конкретных задач. Вы умеете программировать на различных языках программирования; знаете об основных структурах данных, таких как массивы, связанные списки, стеки, очереди, хеш-таблицы, бинарные деревья и ориентированные и неориентированные графы. Вам не нужно реализовывать эти структуры данных, поскольку они обычно предоставляются библиотеками. Так что мы ожидаем, что вы будете использовать эту книгу, чтобы узнать о проверенных эффективных решениях стоящих перед вами задач. Вы узнаете о некоторых новых структурах данных и новых способах их применения для повышения эффективности алгоритмов. Ваши способности к эффективному решению стоящих перед вами задач, несомненно, повысятся, после того как вы познакомитесь с материалами, представленными в нашей книге.

Соглашения, используемые в данной книге

В книге использованы следующие типографские соглашения.

`Код`

```
Все исходные тексты в книге набраны данным шрифтом.

Этот код скопирован непосредственно из репозитория и отражает
реальный код. Все листинги книги отформатированы так, чтобы
подчеркнуть синтаксис соответствующего языка программирования.
```

Курсив

Указывает ключевые термины, используемые для описания алгоритмов и структур данных, а также важных концепций.

`Моноширинный шрифт`

Указывает имена фактических элементов программного обеспечения в реализации, такие как классы Java, имена массивов C или, скажем, такие константы, как `true` или `false`.

Все URL в книге проверены по состоянию на январь 2016 года, и мы постарались использовать только те URL, которые были корректны в течение длительного

времени. Короткие URL, например `http://www.oreilly.com`, приводятся непосредственно в тексте, длинные — в примечаниях или в списке литературы.

Использование примеров кода

Эта книга написана, чтобы помочь вам выполнить свою работу. В общем случае вы можете использовать примеры кода из нее в своих программах и документации. Вам не нужно связываться с издательством для получения разрешения, если только вы не воспроизводите значительную часть кода. Например, для написания программы, в которой используется несколько фрагментов кода из этой книги, разрешение не нужно. Однако для продажи или распространения на CD-ROM примеров из книг издательства O'Reilly необходимо отдельное разрешение. Используя ссылку на книгу и пример кода в ответе на вопрос, получать разрешение не нужно. Но оно необходимо при включении значительного объема кода из этой книги в документацию своего продукта.

Мы не требуем точного указания источника при использовании примеров кода, но были бы признательны за него. Обычно достаточно названия книги, фамилии автора, названия издательства и ISBN.

Если вы полагаете, что использование примеров кода выходит за рамки описанных разрешений, не постесняйтесь связаться с нами по адресу `permissions@oreilly.com`.

Благодарности

Мы хотели бы поблагодарить рецензентов книги за их внимание и предложения, которые позволили улучшить текст и устранить недочеты предыдущих проектов. По первому изданию это Алан Дэвидсон (Alan Davidson), Скот Дрисдейл (Scot Drysdale), Кшиштоф Дулеба (Krzysztof Duleba), Джин Хьюз (Gene Hughes), Мурали Мани (Murali Mani), Джеффри Ясскин (Jeffrey Yasskin) и Дэниэль Ю (Daniel Yoo). По второму изданию — Алан Солис (Alan Solis), Роберт Дэй (Robert P.J. Day) и Скот Дрисдейл (Scot Drysdale).

Джордж Хайнеман хотел бы поблагодарить тех, кто привил ему страсть к алгоритмам, включая профессора Скота Дрисдейла (Scot Drysdale) из Дартмутского колледжа и Цви Галила (Zvi Galil) из Колумбийского университета. Как всегда, Джордж благодарит жену Дженнифер и детей Николя (Nicholas) (который как раз начал учиться программировать) и Александра (Alexander) (который любит делать оригами из черновиков этого издания).

Гэри Поллис (Gary Pollice) хотел бы поблагодарить свою жену Викки за 46 отличных лет. Он также благодарен кафедре информатики Вустерского политехнического института за прекрасную атмосферу.

Стэнли Селков (Stanley Selkow) хотел бы поблагодарить свою жену Деб. Эта книга — еще один шаг на их длинном совместном пути.

Об авторах

Джордж Т. Хайнеман является адъюнкт-профессором информатики в Вустерском политехническом институте. Его научные интересы находятся в области программной инженерии и модульного синтеза программных систем. Он является соредактором книги *Component-Based Software Engineering: Putting the Pieces Together* (Addison-Wesley). Помимо этого, Джордж — страстный любитель головоломок. Он изобрел Суджикен (Sujiken) — вариацию судоку с расположением ячеек в прямоугольном треугольнике, в котором числа не могут повторяться в горизонтали, вертикали или диагонали в любом направлении. Среди опубликованных им книг — *Sudoku on the Half Shell: 150 Addictive SujikenR Puzzles* (Puzzlewright Press, 2011).

Гэри Поллис сам себя называет старым грубияном, который 35 с лишним лет проработал в промышленности, пытаясь выяснить, кем бы он хотел стать, когда вырастет. Несмотря на то что он пока что так и не вырос, в 2003 году он перешел в священные лекционные аудитории, дабы развращать умы следующего поколения разработчиков программного обеспечения такими радикальными идеями, как "разработка программного обеспечения для потребителя", "как работать в команде", "дизайн, качество, элегантность и правильность кода" и "легко быть ботаником, если ты достаточно велик".

Гэри отошел от дел в 2015 году и в настоящее время преподает только один онлайн-курс в год из своего дома престарелых в Куэнка, Эквадор.

Стэнли Селков, профессор информатики из Вустерского политехнического института, в 1965 году получил степень бакалавра в области электротехники в Институте Карнеги, а в 1970 — ученую степень в той же области в Пенсильванском университете. С 1968 по 1970 год он работал в системе здравоохранения в Национальном институте здравоохранения в Бетесде, штат Мэриленд. С 1970 года работал в университетах в Ноксвилле, штат Теннесси, и Вустере, штат Массачусетс, а также в Монреале, Чунцине, Лозанне и Париже. Его основные исследования посвящены теории графов и разработке алгоритмов.

Об изображении на обложке

Животное на обложке этой книги — рак-отшельник (*Pagurus bernhardus*). Имеется более 500 видов таких раков-отшельников. Преимущественно водные, они живут в соленой воде мелководных коралловых рифов. Однако некоторые разновидности раков-отшельников, особенно в тропиках, являются сухопутными (примером может служить пальмовый вор, размер которого достигает 40 см). Но даже наземные отшельники носят в своих раковинах небольшое количество воды, помогающее им дышать и поддерживать необходимый уровень влажности.

В отличие от истинных крабов, отшельники не имеют жесткого панциря и вынуждены искать убежище от хищников в заброшенных раковинах брюхоногих

(улиток). Особенно им нравятся брошенные раковины литорин. По мере роста отшельники находят новые раковины для обитания.

Раки-отшельники являются декаподами, что буквально означает "десятиногие". Из пяти пар ног первые две являются клешнями, которые они используют, чтобы защищать себя и измельчать пищу. Меньшие клешни используются для питания. Вторая и третья пары ног предназначены для передвижения, а последние две помогают удерживаться в раковине.

Как и все ракообразные, отшельники не имеют внутреннего скелета; вместо этого у них довольно твердый экзоскелет из кальция. У них два составных глаза, две пары антенн (которые они используют, чтобы обонять и осязать) и три пары ротовых органов. У основания их антенн есть пара зеленых желез для вывода отходов.

Часто в симбиозе с ними находятся морские анемоны, которые прикрепляются к раковинам отшельников. В обмен на услуги транспортировки и остатки еды морские анемоны помогают ракам защищаться от морских хищников, таких как рыбы и осьминоги.

Известные как "мусорщики моря", отшельники едят что угодно, в том числе мертвые и гниющие останки на берегу моря, и, таким образом, играют важную роль в очистке побережья. Будучи всеядными, они имеют разнообразный рацион, который включает в себя все от червей до органического мусора, такого как трава и листья.

Изображение на обложке взято из второго тома *Library of Natural History* Джонсона.

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 бумажное или 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.com
WWW: http://www.dialektika.com

Наши почтовые адреса:

в России: 195027, Санкт-Петербург, Магнитогорская ул., д. 30, ящик 116
в Украине: 03150, Киев, а/я 152

1 Мысли алгоритмически

Алгоритм имеет значение!

Знание, какой алгоритм следует применить при том или ином наборе обстоятельств, может привести к очень большой разнице в производительности вашего программного обеспечения. Пусть эта книга станет вашим учебником, из которого вы узнаете о ряде важных групп алгоритмов, например о таких, как сортировка и поиск. Мы введем ряд общих подходов, применяемых алгоритмами для решения задач, например подход "разделяй и властвуй" или жадная стратегии. Вы сможете применять полученные знания для повышения эффективности разрабатываемого вами программного обеспечения.

Структуры данных жестко связаны с алгоритмами с самого начала развития информатики. Из этой книги вы узнаете о фундаментальных структурах данных, используемых для надлежащего представления информации для ее эффективной обработки.

Что вам нужно сделать при выборе алгоритма? Мы рассмотрим ответ на этот вопрос в следующих разделах.

Понимание проблемы

Первый шаг в разработке алгоритма состоит в понимании задачи, которую вы хотите решить. Начнем с примера задачи из области вычислительной геометрии. Для данного набора точек двумерной плоскости P, наподобие показанного на рис. 1.1, представим резиновую ленту, которая растягивается так, чтобы охватить все точки, и отпускается. Полученная в результате фигура называется *выпуклой оболочкой* (она представляет собой наименьшую выпуклую фигуру, которая полностью охватывает все точки множества P). Ваша задача — разработать алгоритм для вычисления выпуклой оболочки множества двумерных точек.

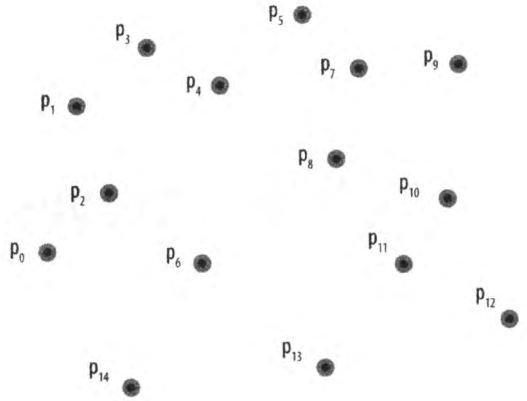


Рис. 1.1. Пример множества из 15 точек на плоскости

Для выпуклой оболочки P любой отрезок между любыми двумя точками P полностью лежит внутри оболочки. Предположим, что мы перенумеровали точки выпуклой оболочки по часовой стрелке. Таким образом, оболочка формируется упорядочением по часовой стрелке h точек L_0, L_1, ..., L_{h-1}, как показано на рис. 1.2. Каждая последовательность из трех точек оболочки L_i, L_{i+1}, L_{i+2} образует правый поворот.

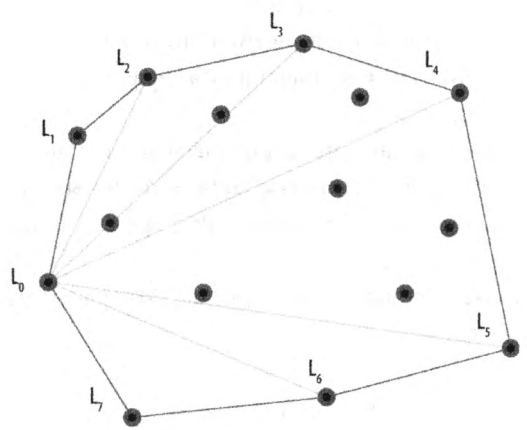


Рис. 1.2. Вычисленная выпуклая оболочка

Имея только эту информацию, вы, вероятно, сможете нарисовать выпуклую оболочку для любого набора точек; но смогли бы вы придумать *алгоритм* (т.е. пошаговую последовательность инструкций, которые будут эффективно вычислять выпуклую оболочку для любого набора точек)?

Задача о выпуклой оболочке представляется нам интересной, в первую очередь, потому, что она не кажется легко классифицируемой в смысле имеющихся областей применения алгоритмов. Не похоже, чтобы помогла любая линейная сортировка точек слева направо, хотя точки оболочки и упорядочиваются по часовой стрелке.

Аналогично не выполняется никакой очевидный поиск, хотя отрезок выпуклой оболочки можно идентифицировать по тому, что оставшиеся n–2 точки плоскости лежат "справа" относительно этого отрезка.

Прямое решение

Очевидно, что выпуклая оболочка существует для любого набора из трех или более точек. Но как ее построить? Рассмотрим следующую идею. Выберем любые три точки из исходного множества и образуем из них треугольник. Если какие-либо оставшиеся n–3 точки содержатся в этом треугольнике, они заведомо не могут быть частью выпуклой оболочки. Мы опишем общий процесс с использованием псевдокода (аналогичные описания вы найдете для каждого из алгоритмов в книге).

Медленный поиск выпуклой оболочки

Наилучший, средний, наихудший случаи: $O(n^4)$

```
slowHull (P)
  foreach p0 in P do                                              ❶
    foreach p1 in {P-p0} do
      foreach p2 in {P-p0-p1} do
        foreach p3 in {P-p0-p1-p2} do
          if p3 содержится в Triangle(p0,p1,p2) then
            Отметить p3 как внутреннюю точку                      ❷

    Создать массив A из всех точек P, не являющихся внутренними
    Определить крайнюю слева точку множества оставшихся точек A
    Отсортировать множество A по углу относительно вертикальной  ❸
          линии, проходящей через крайнюю слева точку
    Вернуть A

❶ Точки p0, p1, p2 образуют треугольник.
❷ Точки, не помеченные как внутренние, образуют выпуклую оболочку.
❸ Эти углы (в градусах) находятся в диапазоне от -90° до 90°.
```

В следующей главе мы рассмотрим математический анализ, который поясняет, почему этот подход является неэффективным. В псевдокоде разъяснены шаги, которые строят выпуклую оболочку для каждого входного набора точек; в частности, такой способ строит выпуклую оболочку на рис. 1.2. Но лучшее ли это, что мы можем сделать?

Интеллектуальные подходы

Многочисленные алгоритмы в этой книге являются результатом стремления получить более эффективные решения для существующего кода. Мы выявляем общие

темы в этой книге, чтобы помочь вам решать ваши собственные задачи. Есть много разных способов вычисления выпуклой оболочки. В набросках этих подходов мы знакомим вас с образцами материала, который будет представлен в последующих главах.

Жадный подход

Вот способ построить выпуклую оболочку по одной точке.

1. Убираем из множества P самую нижнюю точку, *low*, которая должна быть частью оболочки.
2. Сортируем оставшиеся $n-1$ точек в *убывающем* порядке по углу, который образуется по отношению к вертикальной линии, проходящей через точку *low*. Эти углы лежат в диапазоне от 90° для точек слева от линии до -90° для точек справа. p_{n-2} является крайней справа в смысле значения угла точкой, а p_0 — крайней слева. На рис. 1.3 показаны эта вертикальная линия и (тонкими отрезками) соответствующие углы.
3. Начнем с частично выпуклой оболочки, сформированной из трех точек в порядке $\{p_{n-2}, low, p_0\}$. Далее пытаемся расширить оболочку, рассмотрев поочередно каждую из точек от p_1 до p_{n-2}. Если рассматриваемые последними три точки частичной оболочки приводят к повороту отрезка оболочки влево, значит, оболочка содержит неправильную точку, которую необходимо из нее удалить.
4. После того как все точки рассмотрены, частичная оболочка становится полной (рис. 1.3).

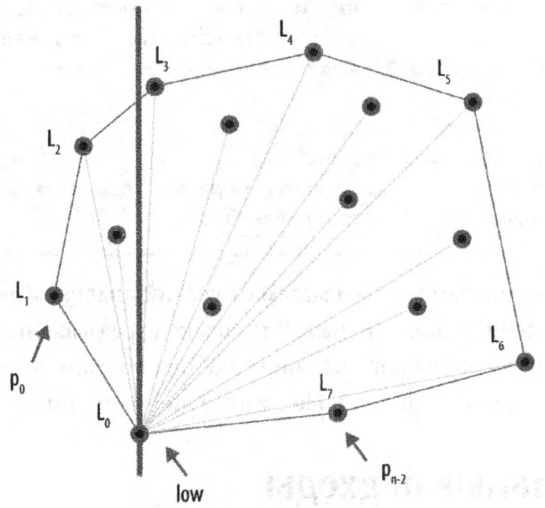


Рис. 1.3. Оболочка, сформированная с использованием жадного подхода

Разделяй и властвуй

Мы можем разделить проблему пополам, если сначала отсортируем все точки P слева направо по координате x (при равных x рассматриваем координату y). Для этой отсортированной коллекции мы сначала вычисляем верхнюю частичную выпуклую оболочку, рассматривая точки в порядке слева направо от p_0 до p_{n-1} в направлении по часовой стрелке. Затем таким же образом строится нижняя частичная выпуклая оболочка, путем обработки тех же точек в порядке справа налево от p_{n-1} до p_0 в том же направлении — по часовой стрелке. При **сканировании выпуклой оболочки** (описанном в главе 9, "Вычислительная геометрия") эти частичные оболочки (показанные на рис. 1.4) вычисляются и объединяются в окончательную выпуклую оболочку всего множества точек.

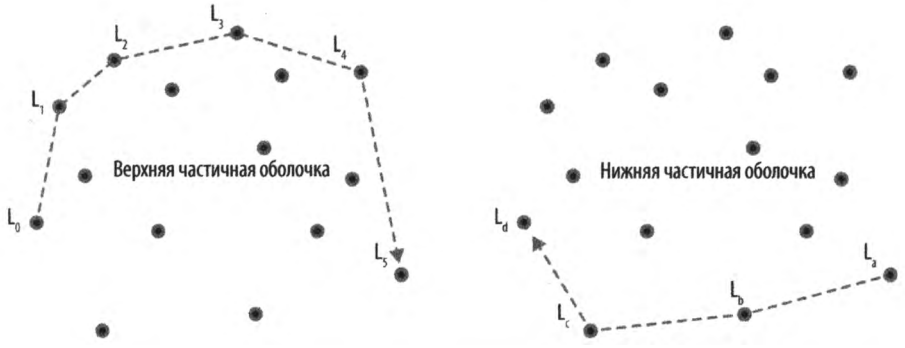


Рис. 1.4. Полная оболочка, образованная верхней и нижней частичными оболочками

Параллельное вычисление

Если у вас есть несколько процессоров, разделим начальное множество точек по координате x, и пусть каждый процессор вычисляет выпуклую оболочку для своего подмножества точек. После того как вычисления будут завершены, окончательная оболочка получается путем *сшивания* в единое целое соседних частичных решений. Параллельный подход делит подзадачи между несколькими процессорами, чтобы ускорить общий ход решения.

На рис. 1.5 показано применение этого подхода на трех процессорах. Две соседние оболочки сшиваются вместе с помощью добавления двух касательных линий — одной вверху, другой внизу — с последующим удалением отрезков, содержащихся в четырехугольнике, образованном этими отрезками.

Приближение

Даже с использованием всех этих улучшений имеется фиксированный *нижний предел* производительности для вычисления выпуклой оболочки, который невозможно превзойти. Однако, возможно, вместо вычисления точного ответа вас мог бы удовлетворить ответ приблизительный, который можно вычислить быстро и погрешность которого *может быть точно определена*.

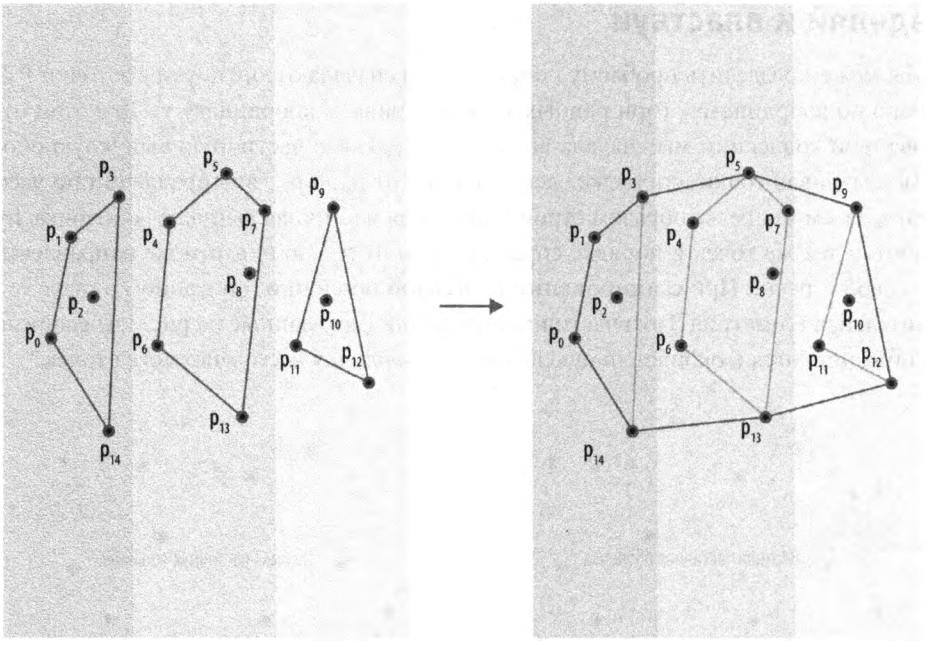


Рис. 1.5. Оболочка, образованная путем параллельного построения и сшивания

Алгоритм **Бентли–Фауста–Препараты** создает приблизительную выпуклую оболочку путем разбиения множества точек на вертикальные полоски [10]. В каждой такой полосе определяются максимальные и минимальные точки (на основе координаты *y*), которые на рис. 1.6 изображены с квадратами вокруг точек. Вместе с крайней слева и крайней справа точками *P* эти крайние точки сшиваются в единую приблизительную выпуклую оболочку. При этом может случиться так, что некоторые точки выходят за рамки полученной оболочки, как, например, точка p_1 на рис. 1.6.

Обобщение

Зачастую оказывается возможным решение более общей задачи, которое затем может быть легко преобразовано в решение нашей конкретной задачи. *Диаграмма Вороного* [47] является геометрической структурой, которая делит набор точек на плоскости на области, каждая из которых оказывается *закрепленной* за одной из исходных точек входного множества *P*. Каждая область R_i представляет собой набор точек (x,y) плоскости, более близких к точке закрепления p_i, чем к любой иной точке множества *P*. После вычисления диаграммы Вороного эти области можно изобразить так, как показано на рис. 1.7. Серые области *полубесконечны* и, как вы можете видеть, соответствуют точкам выпуклой оболочки. Это наблюдение ведет к следующему алгоритму.

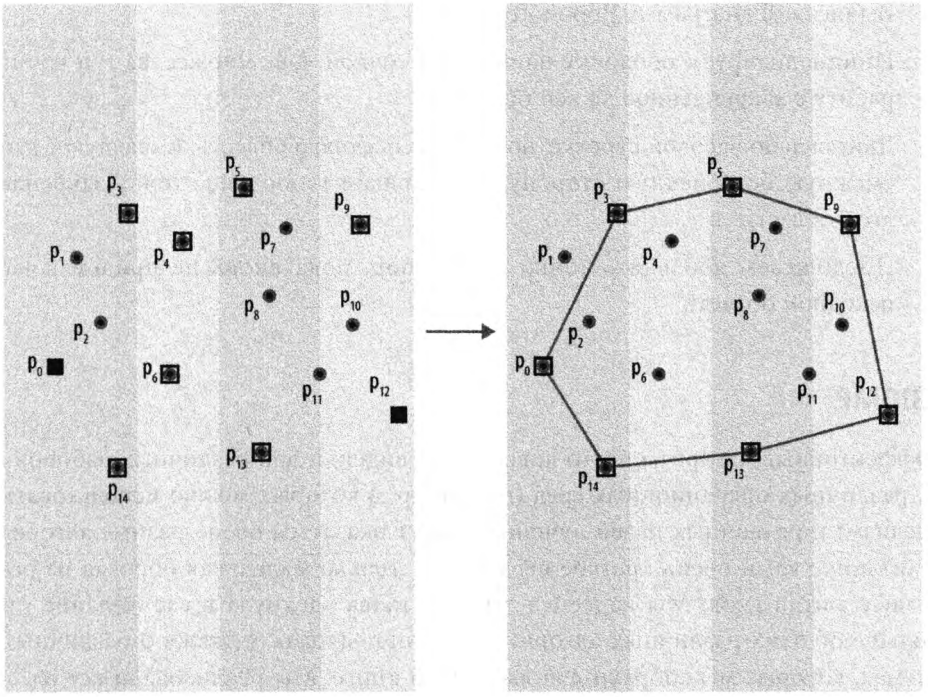


Рис. 1.6. Выпуклая оболочка, полученная путем приблизительных вычислений

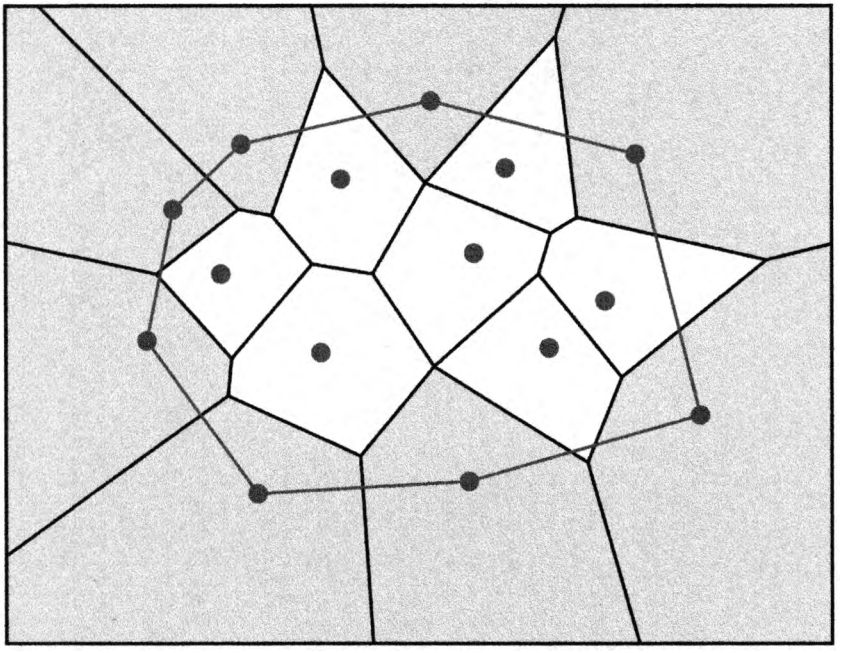


Рис. 1.7. Оболочка, вычисленная из диаграммы Вороного

1. Вычисляем диаграмму Вороного для *P*.
2. Инициализируем оболочку наинизшей точкой, *low*, множества *P* и начинаем работу с закрепленной за ней областью.
3. Двигаясь по часовой стрелке, посещаем соседнюю область, имеющую с данной смежную бесконечную сторону, и добавляем к оболочке точку закрепления этой области.
4. Продолжаем добавлять точки до тех пор, пока вновь не придем к нашей исходной области.

Резюме

Эффективный алгоритм часто вовсе не очевиден, и для различных наборов данных, различных операционных сред (например, в которых можно использовать параллелизм) и различных целей лучшими могут оказаться очень разные алгоритмы. Прочитанное вами очень краткое введение — только маленькая борозда на огромном поле алгоритмов. Мы надеемся, что эта глава вдохнула в вас желание узнать побольше об этих различных алгоритмических подходах, а также о различных алгоритмах, которые мы собрали для вас в этой книге. Мы реализовали все рассматриваемые здесь алгоритмы и задокументировали их, чтобы помочь вам понять, как использовать эти алгоритмы и даже как реализовать их самостоятельно.

2

Математика алгоритмов

Одним из наиболее важных факторов при выборе алгоритма является скорость его работы. Характеристика ожидаемого времени вычисления алгоритма по своей сути является математическим процессом. В этой главе представлены математические инструменты, лежащие в основе этого предсказания времени работы алгоритма. После прочтения этой главы вы должны понимать различные математические термины, используемые как в этой книге, так и в других книгах, посвященных алгоритмам.

Размер экземпляра задачи

Экземпляр задачи представляет собой определенный набор входных данных для программы. В большинстве задач время выполнения программы увеличивается с ростом размера этого набора данных. В то же время чрезмерно компактные представления (возможно, с использованием технологий сжатия) могут необоснованно замедлить выполнение программы. Определить оптимальный способ кодирования экземпляра задачи на удивление трудно, поскольку задачи приходят к нам из реального мира и должны быть переведены в надлежащее представление для решения с помощью программы.

При вычислении алгоритма мы, насколько это возможно, полагаем, что кодировка экземпляра задачи не является определяющим фактором при выяснении вопроса о том, насколько эффективно может быть реализован тот или иной алгоритм. Ваше представление экземпляра задачи должно зависеть только от типа и набора операций, которые должны быть выполнены. Разработка эффективных алгоритмов часто начинается с выбора правильных структур данных для представления задачи.

Поскольку мы не можем формально определить размер экземпляра, мы предполагаем, что экземпляр кодируется некоторым общепринятым лаконичным способом. Например, при сортировке n целых чисел мы принимаем общее соглашение о том, что каждое из n чисел помещается в 32-разрядное слово в используемой

вычислительной платформе и размер экземпляра задачи сортировки равен n. В случае, если некоторые числа требуют для представления более одного слова — но при этом постоянного, фиксированного количества слов, — наша мера размера экземпляра изменяется *только на постоянный множитель*. Таким образом, алгоритм, который выполняет вычисления, используя целые числа, хранящиеся с использованием 64 битов памяти, может оказаться в два раза длиннее аналогичного алгоритма, закодированного с использованием целых чисел, хранящихся в 32 битах.

Исследователи алгоритмов принимают, что они не в состоянии вычислить с достаточной точностью расходы, связанные с использованием в реализации алгоритма конкретной кодировки. Таким образом, они утверждают, что стоимости производительности, которые отличаются на постоянный множитель, *асимптотически эквивалентны*, или, другими словами, их отношение не имеет значения при росте размера задачи. В качестве примера можно ожидать, что применение 64-разрядных целых чисел потребует большего времени работы, чем применение 32-разрядных целых чисел, но мы должны игнорировать это различие и считать, что алгоритм, который хорош для миллиона 32-разрядных целых чисел, будет столь же хорош и для миллиона 64-разрядных целых чисел. Хотя такое определение непрактично для реальных ситуаций (вас бы устроило, если бы расходы на коммунальные услуги оказались в 10 раз больше, чем ожидалось?), оно служит универсальным средством сравнения алгоритмов.

Для всех алгоритмов в этой книге константы практически для всех платформ невелики. Однако при реализации алгоритма в производственном коде необходимо уделять внимание и деталям, которые отражаются на значениях этих констант. Асимптотический подход полезен, поскольку он может предсказать производительность алгоритма для большого экземпляра задачи, основываясь на знаниях о производительности при решении небольших экземпляров задач. Это помогает определить наибольший размер экземпляра задачи, который можно решить с помощью конкретной реализации алгоритма [11].

Для хранения коллекций информации большинство языков программирования поддерживают *массивы*, которые представляют собой смежные области памяти, индексируемые целым числом i для обеспечения быстрого доступа к i-му элементу. Массив является одномерным, когда каждый элемент помещается в слово в используемой платформе (например, массив целых чисел или логических значений). Некоторые массивы могут распространяться на несколько измерений, обеспечивая возможность представления более сложных данных.

Скорость роста функций

Мы описываем поведение алгоритма путем представления *скорости роста времени выполнения* в зависимости от размера входного экземпляра задачи. Такая

характеристика производительности алгоритма является распространенной абстракцией, которая игнорирует многочисленные детали. Чтобы правильно использовать эту меру, требуется осведомленность о деталях, скрытых абстракцией. Каждая программа выполняется на определенной вычислительной платформе; под этим общим термином подразумевается масса конкретных деталей.

- Компьютер, на котором выполняется программа, его процессор (CPU), кеш данных, процессор для вычислений с плавающей точкой (FPU) и другие аппаратные возможности.
- Язык программирования, на котором написана программа, наряду с тем, является ли он компилируемым или интерпретируемым, и настройки оптимизации для генерации кода.
- Операционная система.
- Прочие процессы, выполняющиеся в фоновом режиме.

Мы предполагаем, что с изменением платформы время выполнения программы будет изменяться на постоянный множитель и что мы, таким образом, можем игнорировать различия платформ в соответствии с принципом асимптотической эквивалентности, описанным ранее.

Чтобы поместить эту дискуссию в практический контекст, обсудим вкратце алгоритм **последовательного поиска**, представленный в главе 5, "Поиск". Последовательный поиск анализирует список из $n \geq 1$ отдельных элементов, по одному за раз, пока не будет найдено искомое значение v. Пока что предположим следующее.

- В списке имеется n различных элементов.
- Список содержит искомое значение v.
- Каждый элемент списка с равной вероятностью может быть искомым значением v.

Чтобы разобраться с производительностью **последовательного поиска**, мы должны знать, сколько элементов он просматривает "в среднем". Так как известно, что v имеется в списке и каждый элемент также может быть v, среднее число рассмотренных элементов, $E(n)$, является суммой количества элементов, просмотренных для каждого из n значений, разделенной на n. Математически это записывается так:

$$E(n) = \frac{1}{n}\sum_{i=1}^{n} i = \frac{n(n+1)}{2n} = \frac{1}{2}n + \frac{1}{2}$$

Таким образом, с учетом высказанных выше предположений последовательный поиск анализирует около половины элементов списка из n отдельных элементов. Если число элементов в списке удваивается, то последовательному поиску придется просматривать примерно вдвое больше элементов; ожидаемое количество

просмотров является *линейной* функцией от n. Ожидаемое количество проверок — "около" $c \cdot n$ для некоторой постоянной c; в данном случае $c = 0{,}5$. Фундаментальный факт анализа производительности состоит в том, что постоянная c в долгосрочной перспективе не важна, потому что наиболее важным фактором стоимости является размер экземпляра задачи n. По мере того как n становится все больше и больше, ошибка в утверждении, что

$$\frac{1}{2}n \approx \frac{1}{2}n + \frac{1}{2}$$

становится все менее важной. Фактически отношение между двумя величинами слева и справа от знака приближенного равенства стремится к 1, т.е.

$$\lim_{n \to \infty} \frac{\left(\frac{1}{2}n\right)}{\left(\frac{1}{2}n + \frac{1}{2}\right)} = 1$$

хотя ошибка в оценке имеет важное значение для небольших значений n. В этом контексте мы говорим, что скорость роста ожидаемого количества элементов, просматриваемых последовательным поиском, является линейной. То есть мы игнорируем постоянный множитель и рассматриваем только большие размеры экземпляров задачи.

При использовании абстракции скорости роста при выборе среди нескольких алгоритмов следует помнить о следующем.

Константы имеют значение

Именно поэтому мы используем суперкомпьютеры и постоянно обновляем нашу вычислительную технику.

Размер n не всегда большой

В главе 4, "Алгоритмы сортировки", мы увидим, что скорость роста времени выполнения **Quicksort** меньше, чем скорость роста времени выполнения **сортировки вставками**. Однако сортировка вставками на одной и той же платформе превосходит быструю сортировку для небольших массивов.

Скорость роста для алгоритма определяет, как он будет работать со все бо́льшими и бо́льшими экземплярами задач. Давайте применим этот базовый принцип к более сложному примеру.

Рассмотрим оценку четырех алгоритмов сортировки для конкретной задачи. Приведенные далее данные о производительности были получены путем сортировки блока из n случайных строк. Для блоков размером $n = 1$–512 было выполнено по 50 испытаний. Были отброшены наилучшие и наихудшие результаты, и на рис. 2.1 показано среднее время (в микросекундах) для оставшихся 48 замеренных значений времени работы. Полученные результаты удивительны.

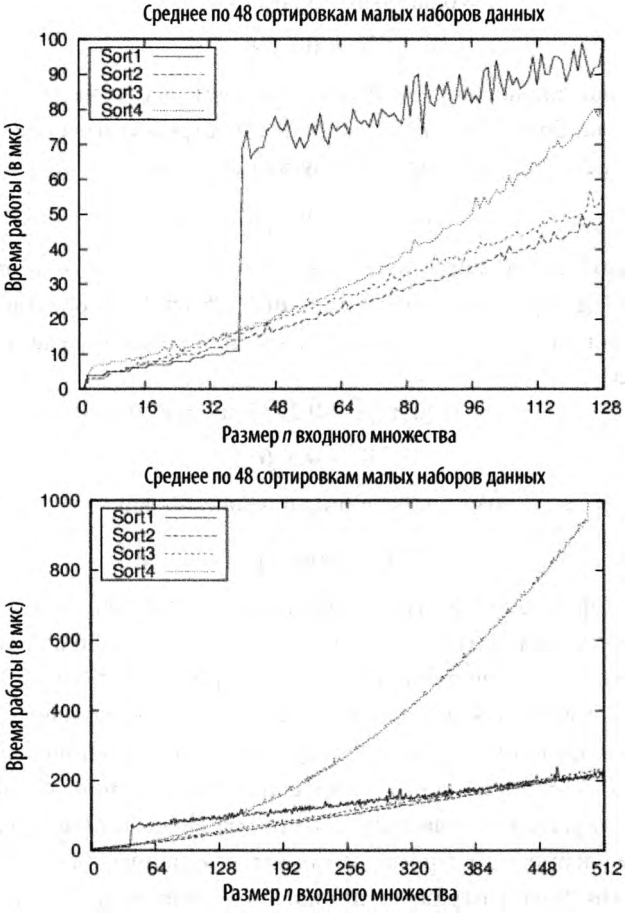


Рис. 2.1. Сравнение четырех алгоритмов сортировки для малых наборов данных

Один из способов интерпретации этих результатов заключается в попытке создать функцию, которая будет предсказывать производительность каждого алгоритма для экземпляра задачи размера *n*. Мы вряд ли сможем угадать такую функцию, поэтому воспользуемся коммерчески доступным программным обеспечением для вычисления линии тренда с помощью статистического процесса, известного как регрессионный анализ. Соответствие линии тренда фактическим данным оценивается как значение в диапазоне от 0 до 1, известное как R^2. Значение около 1 указывает высокую степень соответствия. Например, если $R^2 = 0{,}9948$, то вероятность того, что линия тренда обусловлена случайными отклонениями данных, составляет всего лишь 0,52%.

Очевидно, что среди представленных алгоритмов сортировки алгоритм **Sort-4** имеет наихудшую производительность. Для полученных 512 точек данных линия тренда производительности имеет следующий вид:

$$y = 0{,}0053 \cdot n^2 - 0{,}3601 \cdot n + 39{,}212$$
$$R^2 = 0{,}9948$$

Значение R^2, настолько близкое к 1, объявляет эту линию тренда как точную оценку. Сортировка **Sort-2** является наиболее быстрой сортировкой для заданного диапазона точек. Ее поведение характеризуется следующей формулой линии тренда:

$$y = 0{,}05765 \cdot n \cdot \log(n) + 7{,}9653$$

Изначально **Sort-2** незначительно превосходит **Sort-3**, и ее конечное поведение примерно на 10% быстрее, чем поведение **Sort-3**. **Sort-1** демонстрирует две различные модели поведения. Для блоков размером 39 или менее поведение характеризуется линией тренда

$$y = 0{,}0016 \cdot n^2 + 0{,}2939 \cdot n + 3{,}1838$$
$$R^2 = 0{,}9761$$

Однако при 40 и более строках поведение описывается как

$$y = 0{,}0798 \cdot n \cdot \log(n) + 142{,}7818$$

Числовые коэффициенты в этих уравнениях полностью зависят от платформы, на которой выполняются соответствующие реализации. Как было указано ранее, такие случайные различия не являются важными. При вычислениях, в первую очередь, интересует долгосрочная тенденция, доминирующая при больших значениях *n*. На рис. 2.1 графики поведения алгоритмов представлены с использованием двух различных диапазонов специально, чтобы показать, что, пока *n* не достигнет достаточно больших значений, реальное поведение алгоритма не может быть очевидным.

Разработчики алгоритмов стремятся понять поведенческие различия между разными алгоритмами. **Sort-1** отражает производительность `qsort` в Linux 2.6.9. Просмотрев исходный код (который можно найти в любом из доступных репозиториев кода Linux), мы обнаружим следующий комментарий: "Подпрограмма Qsort из *Engineering a Sort Function* Бентли и Макилроя". Бентли и Макилрой [9] описывают, как оптимизировать быструю сортировку путем изменения стратегии для задач размером меньше 7, между 8 и 39, а также 40 и выше. Приятно видеть, что представленные здесь эмпирические результаты соответствуют описанной реализации.

Анализ наилучшего, среднего и наихудшего случаев

Имеется один вопрос, который нельзя не задать, — будут ли результаты предыдущего раздела верны для всех экземпляров задачи? Как изменится поведение **Sort-2** при изменении входных данных экземпляров задач того же размера?

- Данные могут содержать большие группы элементов, находящиеся уже в отсортированном порядке.

- Входные данные могут содержать повторяющиеся значения.
- Независимо от размера *n* входного множества элементы могут выбираться из гораздо меньшего набора и содержать значительное количество повторяющихся значений.

Хотя **Sort-4** на рис. 2.1 является самым медленным из четырех алгоритмов сортировки *n* случайных строк, оказывается, что она самая быстрая, если данные уже отсортированы. Однако это преимущество быстро исчезает; как показано на рис. 2.2, всего лишь 32 случайных элементов вне правильных позиций достаточно, чтобы **Sort-3** имела более высокую производительность.

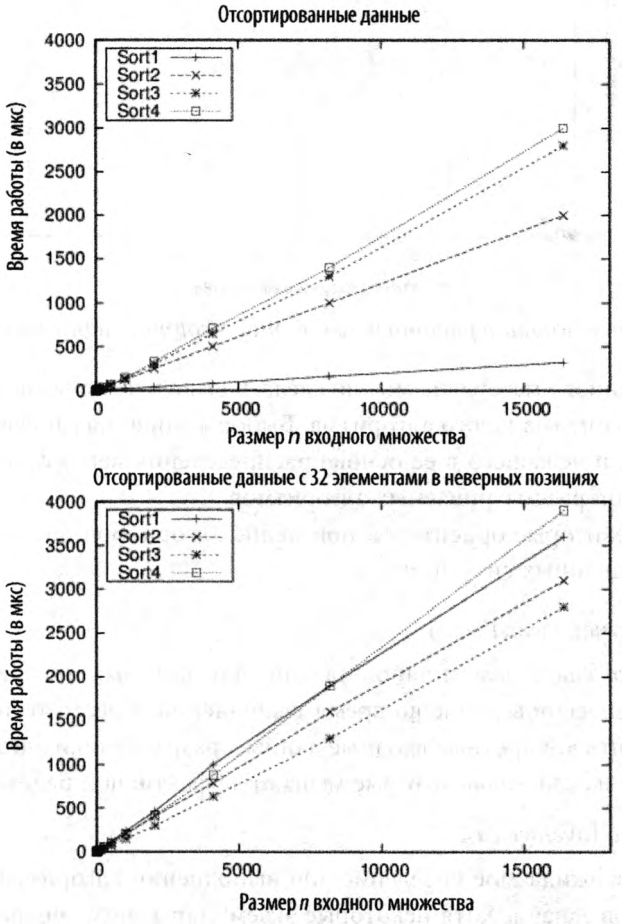


Рис. 2.2. Сравнение алгоритмов сортировки для отсортированных и почти отсортированных данных

Однако предположим, что входной массив с n строками "почти отсортирован", т.е. что $n/4$ строк (25% из них) поменялись местами со строками в другой позиции на расстоянии всего лишь четырех позиций. Это может показаться удивительным, но, как видно из рис. 2.3, теперь сортировка **Sort-4** превосходит прочие алгоритмы сортировки.

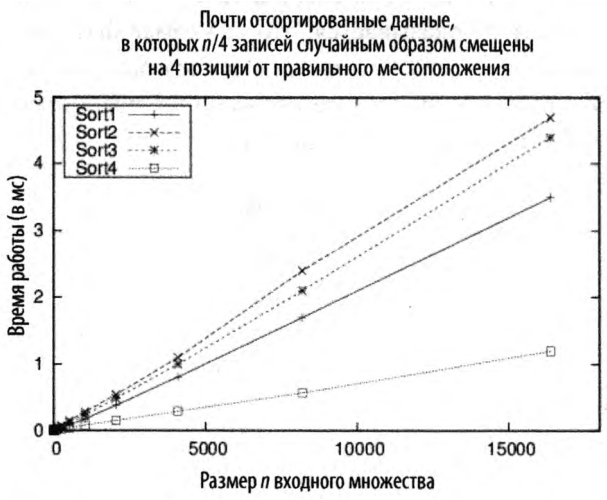


Рис. 2.3. Sort-4 хорошо работает с почти отсортированными данными

Из представленного материала можно сделать вывод, что для многих задач не существует единого оптимального алгоритма. Выбор алгоритма зависит от понимания решаемой задачи и лежащего в ее основе распределения вероятности экземпляров с учетом поведения рассматриваемых алгоритмов.

Чтобы дать некоторые ориентиры, поведение алгоритмов обычно представлено в трех распространенных ситуациях.

Наихудший случай (*worst case*)

Определяет класс экземпляров задачи, для которых алгоритм показывает свое наихудшее поведение во время выполнения. Вместо того чтобы пытаться определить конкретные входные данные, разработчики алгоритмов обычно описывают их *свойства*, которые мешают эффективной работе алгоритма.

Средний случай (*average case*)

Определяет ожидаемое поведение при выполнении алгоритма для случайных экземпляров задачи. Хотя некоторые экземпляры могут потребовать большего времени выполнения, для подавляющего большинства экземпляров задачи этого не произойдет. Эта мера описывает то, чего следует ожидать от применения алгоритма среднему пользователю.

Наилучший случай (best case)

Определяет класс экземпляров задачи, для которых алгоритм демонстрирует наилучшее поведение во время выполнения. Для этих экземпляров алгоритм выполняет наименьшее количество работы. В реальности наилучшие случаи встречаются редко.

Зная производительность алгоритма в каждом из этих случаев, вы можете судить, подходит ли данный алгоритм для использования в вашей конкретной ситуации.

Наихудший случай

Для любого конкретного значения n работа алгоритма или программы может резко изменяться для разных экземпляров одного и того же размера n. Для конкретной программы и конкретного значения n наихудшее время выполнения — это максимальное время выполнения, где максимум взят по всем экземплярам размера n.

Нас интересует наихудшее поведение алгоритма, поскольку зачастую это простейший случай для анализа. Он также показывает, насколько медленной может быть программа в произвольной ситуации.

Говоря более формально, если S_n представляет собой множество экземпляров задачи s_i размера n, а $t()$ — функция, которая измеряет работу алгоритма для каждого экземпляра, то работа алгоритма на множестве S_n в наихудшем случае представляет собой максимум $t(s_i)$ по всем $s_i \in S_n$. Обозначив эту наихудшую на S_n производительность как $T_{wc}(n)$, получаем, что скорость роста $T_{wc}(n)$ определяет сложность алгоритма в наихудшем случае.

Для вычисления каждого отдельного экземпляра s_i, на котором выполняется алгоритм для эмпирического определения экземпляра, приводящего к наихудшей производительности, ресурсов обычно недостаточно. Вместо этого предлагается описание экземпляра задачи, приводящего к наименьшей производительности алгоритма.

Средний случай

Рассмотрим телефонную систему, предназначенную для поддержки большого количества n телефонов. В худшем случае она должна быть в состоянии выполнить все вызовы, когда $n/2$ человек одновременно возьмут свои телефоны и вызовут $n/2$ других человек. Хотя такая система никогда не рухнет из-за перегрузки, создавать ее было бы слишком дорого. В действительности вероятность того, что каждый из $n/2$ человек вызовет своего уникального корреспондента из других $n/2$ человек, чрезвычайно мала. Вместо этого мы могли бы разработать более дешевую систему и использовать математический инструментарий для выяснения вероятности аварии из-за перегрузки.

Для множества экземпляров размера n мы связываем распределение вероятностей $\Pr\{s_i\}$, которое каждому экземпляру s_i назначает вероятность от 0 до 1, так что

сумма вероятностей по всем экземплярам размера n равна 1. Говоря более формально, если S_n — множество экземпляров размера n, то

$$\sum_{s_i \in S_n} \Pr\{s_i\} = 1$$

Если функция $t()$ измеряет работу, выполнимую алгоритмом для каждого экземпляра, то работа алгоритма над множеством S_n в среднем случае равна

$$T_{ac}(n) = \sum_{s_i \in S_n} t(s_i) \Pr\{s_i\}$$

То есть фактическая работа для экземпляра s_i представляет собой $t(s_i)$ с весом, равным вероятности того, что экземпляр s_i будет представлен в качестве входных данных. Если $\Pr\{s_i\} = 0$, то фактическое значение $t(s_i)$ не влияет на ожидаемую работу, выполняемую программой. Обозначая работу с множеством S_n в среднем случае как $T_{ac}(n)$, мы получаем, что скорость роста $T_{ac}(n)$ определяет сложность алгоритма в среднем случае.

Напомним, что при описании скорости роста работы или времени мы последовательно игнорируем константы. Поэтому, когда мы говорим, что последовательный поиск n элементов занимает в среднем

$$\frac{1}{2}n + \frac{1}{2}$$

испытаний (с учетом сделанных нами ранее предположений), то по соглашению мы просто говорим, что при этих предположениях ожидается, что последовательный поиск будет проверять *линейное* количество элементов, или *порядка n* элементов.

Наилучший случай

Знать наилучший случай для алгоритма полезно, несмотря даже на то, что такая ситуация крайне редко встречается на практике. Во многих случаях она обеспечивает понимание оптимальных условий для работы алгоритма. Например, наилучшим случаем для последовательного поиска является тот, когда искомое значение v является первым элементом в списке. Рассмотрим несколько иной подход, который мы будем называть поиском подсчетом и который подсчитывает, сколько раз v встречается в списке. Если вычисленное значение равно нулю, то этот элемент не найден, так что поиск возвращает значение `false`; в противном случае он возвращает значение `true`. Обратите внимание, что поиск подсчетом всегда проходит весь список; поэтому, несмотря на то что его поведением в наихудшем случае является $O(n)$ — так же, как и при последовательном поиске, — его поведение в наилучшем случае так и остается $O(n)$, поэтому невозможно воспользоваться его преимуществами в наилучшем или среднем случае, в котором он мог бы работать быстрее.

Нижняя и верхняя границы

Мы упростили презентацию записи с "большим O" в этой книге. Наша цель — классифицировать поведение алгоритма по тому, как он решает экземпляры задач с увеличением их размера n. Классификация алгоритмов имеет вид $O(f(n))$, где $f(n)$ наиболее часто является такой функцией от n, как n, n^3 или 2^n.

Предположим, например, что существует алгоритм, наихудшая производительность которого *никогда не превышает* прямую пропорциональность размеру входного экземпляра задачи по достижении некоторого "достаточно большого" размера. Точнее говоря, существует некоторая константа $c>0$, такая, что $t(n) \leq c \cdot n$ для всех $n > n_0$, где n_0 является той точкой, в которой экземпляр задачи становится "достаточно большим". В этом случае классификацией будет функция $f(n) = n$ и мы будет использовать запись $O(n)$. Для этого же алгоритма предположим, что производительность в наилучшем случае *никогда не меньше*, чем прямая пропорциональность размеру экземпляра входной задачи. В этом случае существуют другая постоянная c и другой порог размера задачи n_0 и $t(n) \geq c \cdot n$ для всех $n > n_0$. В этом случае классификация вновь имеет вид $f(n) = n$, но в этот раз она записывается как $\Omega(n)$.

Таким образом, фактическая формальная запись выглядит следующим образом:

- *нижняя граница* времени работы алгоритма классифицируется как $\Omega(f(n))$ и соответствует сценарию наилучшего случая;
- *верхняя граница* времени работы алгоритма классифицируется как $O(f(n))$ и соответствует сценарию наихудшего случая.

Необходимо рассматривать оба сценария. Внимательный читатель заметит, что мы могли просто использовать функцию $f(n) = c \cdot 2^n$ для классификации рассмотренного выше алгоритма как $O(2^n)$, хотя это выражение и описывает гораздо более медленное поведение. В действительности это дает нам очень мало информации — все равно что сказать, что нам понадобится не более чем неделя для решения задачи, которая в действительности решается за 5 минут. В этой книге мы всегда представляем классификацию алгоритма, используя наиболее точное соответствие.

В теории сложности есть еще одна запись, $\Theta(f(n))$, которая сочетает в себе обе концепции для определения *точной границы*, т.е. когда нижняя граница $\Omega(f(n))$ и верхняя граница $O(f(n))$ используют для классификации одну и ту же функцию $f(n)$. Мы выбрали широко признанную (и более неформальную) запись $O(f(n))$ для упрощения представления и анализа. В книге мы гарантируем, что при обсуждении поведения алгоритма нет более точной функции $f'(n)$, которая может использоваться для классификации алгоритма, который мы определяем как $O(f(n))$.

Семейства производительности

Мы сравниваем алгоритмы, оценивая их производительность для экземпляров задач размера n. Эта методология является стандартным средством, разработанным в последние полвека для сравнения алгоритмов. Поступая таким образом, мы можем определить, какие алгоритмы масштабируются для решения задач нетривиального размера, оценивая время, необходимое алгоритму для обработки входных данных определенного размера. Вторичной характеристикой является потребность алгоритма в памяти; при необходимости мы будем рассматривать эту проблему в рамках описания отдельных алгоритмов.

Мы используем следующую классификацию, упорядоченную по *убыванию* эффективности.

- Константная: $O(1)$
- Логарифмическая: $O(\log n)$
- Сублинейная: $O(n^d)$ при $d < 1$
- Линейная: $O(n)$
- Линейно-логарифмическая: $O(n \cdot \log n)$
- Квадратичная: $O(n^2)$
- Экспоненциальная: $O(2^n)$

При оценке производительности алгоритма следует иметь в виду, что для определения ее классификации требуется найти самые дорогостоящие вычисления в алгоритме. Например, рассмотрим алгоритм, который подразделяется на две задачи, которые классифицируются как линейная, за которой следует квадратичная. В этом случае общая производительность алгоритма должна быть классифицирована как квадратичная.

Теперь проиллюстрируем данную классификацию производительности на примерах.

Константное поведение

Анализируя производительность алгоритмов в этой книге, мы часто утверждаем, что некоторые примитивные операции обеспечивают константную производительность. Очевидно, что это утверждение не является абсолютным фактором для фактической производительности операции, так как мы никак не затрагиваем аппаратное обеспечение. Например, сравнение двух 32-битных чисел x и y на равенство должно иметь одинаковую производительность независимо от фактических значений x и y. Константная операция определяется как имеющая производительность $O(1)$.

А что можно сказать о производительности сравнения двух 256-битных чисел? Или двух 1024-битных чисел? Оказывается, что для предопределенного фиксированного размера k можно сравнить два k-битных числа за постоянное время. Ключевым является тот факт, что размер задачи (например, сравниваемые значения x и y) не могут выйти за пределы фиксированного размера k. Мы абстрагируемся от дополнительных действий, мультипликативных в терминах k, используя запись $O(1)$.

Логарифмическое поведение

Бармен предлагает следующий спор на 10000 долларов: "Я загадываю число от 1 до 1000000 и даю вам 20 попыток, чтобы его отгадать. После каждой вашей догадки я говорю вам, назвали ли вы слишком малое число, слишком большое или угадали. Если вы угадываете число за 20 или меньше попыток, я плачу вам 10000 долларов. Если не угадали, вы платите мне". Ну как, вы принимаете пари? Вы просто обязаны это сделать, потому что вы всегда можете выиграть. В табл. 2.1 показан пример сценария для диапазона от 1 до 8, в котором задается ряд вопросов, уменьшающий размер задачи примерно наполовину.

Таблица 2.1. Угадывание числа от 1 до 8

Число	Первый раунд	Второй раунд	Третий раунд	Четвертый раунд
1	Это 4? Много	Это 2? Много	Это 1! Угадал!	
2	Это 4? Много	Это 2? Угадал!		
3	Это 4? Много	Это 2? Мало	Это 3! Угадал!	
4	Это 4? Угадал!			
5	Это 4? Мало	Это 6? Много	Это 5! Угадал!	
6	Это 4? Мало	Это 6? Угадал!		
7	Это 4? Мало	Это 6? Мало	Это 7? Угадал!	
8	Это 4? Мало	Это 6? Мало	Это 7? Мало	Это 8! Угадал!

В каждом раунде, в зависимости от конкретных ответов от бармена, размер потенциального диапазона, содержащего загаданное число, сокращается примерно в два раза. В конце концов диапазон окажется ограниченным только одним возможным числом; это происходит после $1+\lfloor \log_2 n \rfloor$ раундов, где $\log_2 x$ вычисляет логарифм x по основанию 2. Функция "пол" $\lfloor x \rfloor$ округляет число x до наибольшего целого числа, не превосходящего x. Например, если бармен выбирает число от 1 до 10,

то вы могли бы отгадать его за $1+\lfloor \log_2 10 \rfloor = 1+\lfloor 3{,}322 \rfloor$, т.е. за четыре попытки. Как еще одно подтверждение верности этой формулы рассмотрим загадывание барменом одного из двух чисел. Вам нужно два раунда, чтобы гарантировать угаданное число: $1+\lfloor \log_2 2 \rfloor = 1+\lfloor 1 \rfloor = 2$. Помните, что согласно правилам бармена вы *должны назвать число вслух*.

Этот подход одинаково хорошо работает и для 1 000 000 чисел. В самом деле, алгоритм отгадывания, показанный в примере 2.1, работает для любого диапазона $[low, high]$ и определяет значение загаданного числа n за $1+\lfloor \log_2 (high-low+1) \rfloor$ раундов. Если всего имеется 1 000 000 чисел, этот алгоритм найдет загаданное число максимум за $1+\lfloor \log_2 1\,000\,000 \rfloor = 1+\lfloor 19{,}932 \rfloor = 20$ раундов (в наихудшем случае).

Пример 2.1. Код на языке программирования Java для угадывания чисел в диапазоне [low, high]

```
// Вычисляет количество раундов, если n гарантированно
// находится в диапазоне [low,high].
public static int turns (int n, int low, int high) {
    int turns = 0;
    // Продолжаем, пока имеется число для угадывания
    while (high >= low) {
        turns++;
        int mid = (low + high)/2;
        if (mid == n) {
            return turns;
        } else if (mid < n) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }
    return turns;
}
```

Логарифмические алгоритмы чрезвычайно эффективны, потому что быстро сходятся к решению. Эти алгоритмы уменьшают размер задачи на определенный множитель при каждой итерации (чаще всего примерно вполовину). Рассмотренный выше алгоритм приводит к решению после максимум $k = 1+\lfloor \log_2 n \rfloor$ итераций и на i-й итерации ($0 < i \le k$) вычисляет пробное значение, о котором известно, что оно находится в пределах $\pm\varepsilon = 2^{k-i} - 1$ от фактического загаданного числа. Значение e рассматривается как ошибка, или неопределенность, угадывания. После каждой итерации цикла значение e уменьшается вдвое.

В оставшейся части книги всякий раз, когда мы ссылаемся на $\log(n)$, предполагается логарифм по основанию 2, так что мы просто убираем индекс из записи $\log_2(n)$.

Еще один пример, демонстрирующий эффективное поведение, — алгоритм **деления пополам**, который вычисляет корень уравнения от одной переменной,

а именно — вычисляет значение x, при котором значение непрерывной функции $f(x)$ равно нулю. Вычисления начинаются с двух значений, a и b, в которых $f(a)$ и $f(b)$ имеют противоположные знаки, т.е. одно из значений положительное, а другое — отрицательное. На каждом шаге метод *делит пополам* диапазон $[a,b]$ путем вычисления его средины c и определяет, в какой из половин должен находиться корень. Таким образом, на каждой итерации значение c является приближенным значением корня с погрешностью, уменьшающейся в два раза.

Чтобы найти корень $f(x)=x\cdot\sin(x)-5\cdot x-\cos(x)$, начнем с $a=-1$ и $b=1$. Как показано в табл. 2.2, алгоритм сходится к решению уравнения $f(x)=0$, корнем которого является значение $x=-0{,}189302759$.

Таблица 2.2. Метод деления пополам

n	*a*	*b*	*c*	*f(c)*
1	-1,0000000	1,0000000	0,0000000	-1,0000000
2	-1,0000000	0,0000000	-0,5000000	1,8621302
3	-0,5000000	0,0000000	-0,2500000	0,3429386
4	-0,2500000	0,0000000	-0,1250000	-0,3516133
5	-0,2500000	-0,1250000	-0,1875000	-0,0100227
6	-0,2500000	-0,1875000	-0,2187500	0,1650514
7	-0,2187500	-0,1875000	-0,2031250	0,0771607
8	-0,2031250	-0,1875000	-0,1953125	0,0334803
9	-0,1953125	-0,1875000	-0,1914062	0,0117066
10	-0,1914062	-0,1875000	-0,1894531	0,0008364
11	-0,1894531	-0,1875000	-0,1884766	-0,0045945
12	-0,1894531	-0,1884766	-0,1889648	-0,0018794
13	-0,1894531	-0,1889648	-0,1892090	-0,0005216
14	-0,1894531	-0,1892090	-0,1893311	0,0001574
15	-0,1893311	-0,1892090	-0,1892700	-0,0001821
16	-0,1893311	-0,1892700	-0,1893005	-0,0000124

Сублинейное поведение

В некоторых случаях поведение алгоритма лучше *линейного*, но не так эффективно, как *логарифмическое*. Как описано в главе 10, "Пространственные древовидные структуры", k-d-дерево (k-мерное дерево) может эффективно разделять множество, состоящее из n d-мерных точек. Если дерево сбалансировано, время поиска для запросов диапазонов, которые соответствуют оси точек, составляет $O(n^{1-1/d})$. Для двумерных запросов производительность равна $O(\sqrt{n})$.

Линейная производительность

Очевидно, что для решения одних задач требуется больше усилий, чем для решения других. Ребенок может вычислить 7+5 и получить 12. Насколько сложнее задача 37+45?

В частности, насколько трудно сложить два n-значных числа $a_{n-1}\ldots a_0 + b_{n-1}\ldots b_0$ и получить в результате $n+1$-значное число $c_n\ldots c_0$? В этом алгоритме используются следующие примитивные операции:

$$c_i \leftarrow \left(a_i + b_i + carry_i\right) \bmod 10$$

$$carry_{i+1} \leftarrow \begin{cases} 1, & \text{если } a_i + b_i + carry_i \geq 10, \\ 0 & \text{в противном случае.} \end{cases}$$

Пример реализации этого алгоритма на языке программирования Java показан в примере 2.2, где n-значное число представлено в виде массива значений типа `int`, в котором старшая (т.е. крайняя слева) цифра находится в позиции с индексом 0. В примерах в этом разделе предполагается, что каждое из этих значений представляет собой десятичную цифру d, такую, что $0 \leq d \leq 9$.

Пример 2.2. Реализация сложения многозначных чисел на языке программирования Java

```
public static void add (int[] n1, int[] n2, int[] sum) {
    int position = n1.length-1;
    int carry = 0;
    while (position >= 0) {
        int total = n1[position] + n2[position] + carry;
        sum[position+1] = total % 10;
        if (total > 9) { carry = 1; } else { carry = 0; }
        position--;
    }
    sum[0] = carry;
}
```

До тех пор, пока входные данные задачи могут храниться в памяти, метод `add` вычисляет сумму двух чисел, представленных входными целочисленными массивами `n1` и `n2` и сохраняет результат в массиве `sum`. Будет ли эта реализация такой же эффективной, как альтернативный метод `plus`, приведенный в примере 2.3, который вычисляет точно такое же значение, используя другие вычисления?

Пример 2.3. Реализация метода `plus` на Java

```
public static void plus(int[] n1, int[] n2, int[] sum) {
    int position = n1.length;
    int carry = 0;
    while (--position >= 0) {
        int total = n1[position] + n2[position] + carry;
        if (total > 9) {
```

```
            sum[position+1] = total-10;
            carry = 1;
        } else {
            sum[position+1] = total;
            carry = 0;
        }
    }
    sum[0] = carry;
}
```

Влияют ли эти небольшие различия реализации на производительность алгоритма? Рассмотрим два других потенциальных фактора, которые могут влиять на производительность алгоритма.

- `add` и `plus` можно тривиально преобразовать в программу на языке программирования C. Как выбор языка влияет на производительность алгоритма?
- Программа может выполняться на разных компьютерах. Как выбор аппаратного обеспечения влияет на производительность алгоритма?

Эти реализации были выполнены 10 000 раз с числами в диапазоне от 256 до 32 768 знаков. Для каждого количества создавалось случайное число такого размера; после этого для каждой из 10 000 проб над обоими числами выполнялся циклический перенос (один — влево, другой — вправо) для создания двух разных чисел. Были использованы два различных языка программирования (C и Java). Мы начинаем с гипотезы, что при удвоении размера задачи время выполнения алгоритма также удваивается. Мы хотели бы убедиться, что это общее поведение, наблюдаемое независимо от компьютера, языка программирования или использованной реализации. Каждый вариант был выполнен на множестве конфигураций.

g

Версия на языке программирования C, скомпилированная с включением отладочной информации.

O1, O2, O3

Версия на языке программирования C, скомпилированная с разными уровнями оптимизации. Увеличение значения приводит к более высокой производительности.

Java

Реализация алгоритма на языке программирования Java.

В табл. 2.3 содержатся результаты и для `add`, и для `plus`. В восьмом, последнем, столбце сравнивается соотношение производительности метода `plus` для задачи размера $2n$ и задачи размера n. Определим $t(n)$ как фактическое время работы алгоритма для входных данных размера n. Этот шаблон роста предоставляет эмпирические данные времени вычисления `plus` для двух n-значных чисел.

Таблица 2.3. Время (в мс) выполнения 10000 вызовов `add/plus` случайных n-значных чисел

n	Add-g	Add-java	Add-O3	Plus-g	Plus-java	Plus-O3	Отношение
256	33	19	10	31	20	11	
512	67	22	20	58	32	23	2,09
1024	136	49	40	126	65	46	2,00
2048	271	98	80	241	131	95	2,07
4096	555	196	160	489	264	195	2,05
8192	1107	392	321	972	527	387	1,98
16384	2240	781	647	1972	1052	805	2,08
32768	4604	1554	1281	4102	2095	1721	2,14
65536	9447	3131	2572	8441	4200	3610	2,10
131072	19016	6277	5148	17059	8401	7322	2,03
262144	38269	12576	10336	34396	16811	14782	2,02
524288	77147	26632	21547	69699	35054	30367	2,05
1048576	156050	51077	53916	141524	61856	66006	2,17

Можно классифицировать алгоритм сложения как *линейный* по отношению к размеру входных данных n. То есть имеется некоторая константа $c>0$, такая, что $c \leq t(n) \cdot n$ для "достаточно большого" n, или, точнее, для всех $n > n_0$. На самом деле нам не нужно вычислять фактическое значение c или n_0; мы просто знаем, что они существуют и могут быть вычислены. Можно выполнить доказательство для установления линейной нижней границы сложности сложения, показав, что должна быть рассмотрена каждая цифра (достаточно рассмотреть последствия пропуска одной из цифр).

Для всех выполнений `plus` (независимо от языка или настроек компиляции) мы можем установить c равным 1/7 и выбрать в качестве n_0 значение 256. Другие реализации сложения будут иметь другие значения констант, но общее поведение попрежнему будет *линейным*. Этот результат может показаться удивительным, учитывая, что большинство программистов предполагают, что целочисленные арифметические операции являются операциями с константным временем выполнения; однако постоянное время сложения достижимо только тогда, когда представление целочисленных значений (например, 16- или 64-битное) использует фиксированный размер n.

При рассмотрении различий в алгоритмах постоянная c не так важна, как знание скорости роста. Кажущиеся несущественными различия приводят к различной производительности. Реализация `plus` пытается повысить эффективность путем устранения вычисления остатка от деления (`%`). Тем не менее при компиляции как метода `plus`, так и метода `add` с оптимизацией `-O3` последний почти на 30% быстрее. Но мы не игнорируем значение c. Если мы выполняем сложение большое количество раз, даже небольшие изменения фактического значения c могут оказать большое влияние на общую производительность программы.

Линейно-логарифмическая производительность

Поведение многих эффективных алгоритмов лучше всего описывается этим семейством производительности. Чтобы объяснить, как это поведение возникает на практике, давайте определим $t(n)$ как время, которое требуется алгоритму для решения экземпляра задачи размера n. Метод "разделяй и властвуй" является эффективным средством решения задач, когда задача размера n делится на (примерно равные) подзадачи размера $n/2$, решаемые рекурсивно. Решения этих подзадач объединяются вместе в решение исходной задачи размера n за *линейное время*. Математически это можно определить следующим образом:

$$t(n) = 2 \cdot t(n/2) + c \cdot n$$

Иначе говоря, $t(n)$ включает стоимость двух подзадач вместе с не более чем линейной временной стоимостью (например, $c \cdot n$) объединения результатов. Теперь в правой части уравнения $t(n/2)$ представляет собой время, необходимое для решения задачи размера $n/2$; используя ту же логику, его можно представить как

$$t(n/2) = 2 \cdot t(n/4) + c \cdot n/2,$$

так что исходное уравнение превращается в

$$t(n) = 2 \cdot \left[2 \cdot t(n/4) + c \cdot n/2 \right] + c \cdot n.$$

Если мы распишем правую часть уравнения еще раз, то получим

$$t(n) = 2 \cdot \left[2 \cdot t(n/4) + c \cdot n/2 \right] + c \cdot n.$$

Это последнее уравнение сводится к

$$t(n) = 2 \cdot \left[2 \cdot \left[2 \cdot t(n/8) + c \cdot n/4 \right] + c \cdot n/2 \right] + c \cdot n.$$

Таким образом, можно просто сказать, что

$$t(n) = 2^k \cdot t(n/2^k) + k \cdot c \cdot n.$$

Это преобразование правой части завершается, когда $2^k = n$ (т.е. когда $k = \log n$). В последнем, базовом случае, когда размер задачи равен 1, производительность $t(1)$ является константой d. Таким образом, мы получаем решение для $t(n)$ в аналитическом виде: $t(n) = n \cdot d + c \cdot n \cdot \log n$. Поскольку $c \cdot n \cdot \log n$ асимптотически больше $d \cdot n$ для любых фиксированных констант c и d, $t(n)$ можно просто записать как $O(n \cdot log\ n)$.

Квадратичная производительность

Рассмотрим теперь задачу, похожую на задачу сложения n-значных чисел, но теперь мы будем их умножать. В примере 2.4 показана реализация такого умножения "в столбик" — элементарный школьный алгоритм, использующий то же представление n-значных чисел, что и использованное ранее при сложении.

Пример 2.4. Реализация умножения n-значных чисел на Java

```
public static void mult (int[] n1, int[] n2, int[] result) {
    int pos = result.length-1;

    // Очистка всех значений
    for (int i = 0; i < result.length; i++) { result[i] = 0; }
    for (int m = n1.length-1; m>=0; m--) {
        int off = n1.length-1 - m;
        for (int n = n2.length-1; n>=0; n--,off++) {
            int prod = n1[m]*n2[n];

            // Вычисление частичной суммы переносом предыдущей позиции
            result[pos-off] += prod % 10;
            result[pos-off-1] += result[pos-off]/10 + prod/10;
            result[pos-off] %= 10;
        }
    }
}
```

И вновь мы написали альтернативную программу `times`, в которой устранена необходимость в дорогостоящем операторе получения остатка от деления и опущено наиболее глубоко вложенное вычисление при нулевом `n1[m]` (описываемый код `times` не показан, но его можно найти в репозитории). Метод `times` содержит 203 строки исходного текста на Java для удаления двух операторов вычисления остатка от деления. Настолько ли велико при этом повышение производительности, что оправдывает дополнительные усилия по разработке и поддержке этого кода?

В табл. 2.4 показано поведение этих реализаций умножения с теми же случайным образом генерируемыми входными данными, которые использовались при демонстрации суммирования. На рис. 2.4 производительность показана графически с помощью параболической кривой, которая является признаком *квадратичного* поведения.

Таблица 2.4. Время (в мс) выполнения 10 000 умножений

n	$mult_n$(ms)	$times_n$(ms)	$mult_{2n}/mult_n$
4	2	41	
8	8	83	4,00
16	33	129	4,13
32	133	388	4,03
64	530	1276	3,98
128	2143	5009	4,04
256	8519	19014	3,98
512	34231	74723	4,02

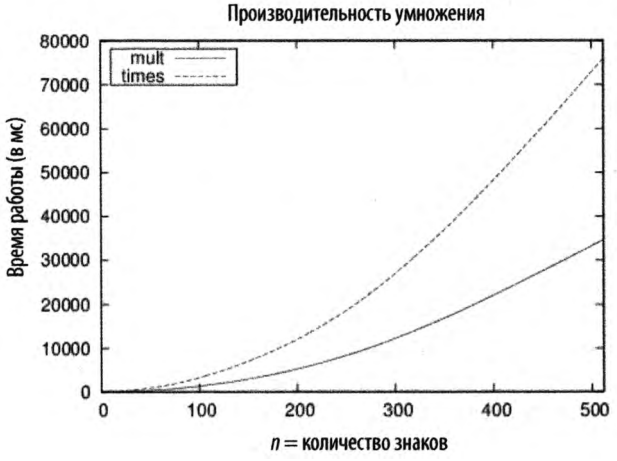


Рис. 2.4. Сравнение умножений `mult` *и* `times`

Несмотря на то что метод `times` примерно в два раза медленнее `mult`, как `times`, так и `mult` демонстрируют одну и ту же асимптотическую производительность. Отношение $\texttt{mult}_{2n}/\texttt{mult}_{n}$ примерно равно 4, что указывает на квадратичную производительность умножения. Давайте определим $t(n)$ как фактическое время работы алгоритма умножения для входных данных размера n. При таком определении должна иметься некоторая константа $c>0$, такая, что $t(n) \leq c \cdot n^2$ для всех $n > n_0$. На самом деле нам не нужно знать все детали значений c и n_0, достаточно знать, что они существуют. Для реализации умножения `mult` на нашей платформе мы можем установить c равным 1/7 и выбрать n_0 равным 16.

И вновь, отдельные изменения реализации не могут "испортить" присущее данному алгоритму квадратичное поведение производительности. Однако существуют и другие алгоритмы [61] для умножения пары n-значных чисел, которые значительно быстрее, чем квадратичный. Эти алгоритмы играют важную роль в таких приложениях, как шифрование данных, когда часто необходимо перемножать очень большие целые числа.

Менее очевидные производительности

В большинстве случаев прочтения описания алгоритма (как, например, описания приведенных выше алгоритмов сложения и умножения) достаточно для классификации алгоритма как *линейного* или *квадратичного*. Например, основным показателем квадратичности является наличие вложенного цикла. Но некоторые алгоритмы не поддаются такому простому анализу. Рассмотрим в примере 2.5 реализацию алгоритма вычисления НОД (наибольшего общего делителя (greatest common divisor — GCD) двух целых чисел), разработанного Евклидом.

Пример 2.5. Алгоритм вычисления НОД Евклида

```
public static void gcd (int a[], int b[], int gcd[]) {
    if (isZero (a)) { assign (gcd, a); return; }
    if (isZero (b)) { assign (gcd, b); return; }

    a = copy (a);   // Делаем копии для гарантии, что
    b = copy (b);   // a и b не изменяются

    while (!isZero (b)) {
        // Последний аргумент subtract – знак результата,
        // который можно игнорировать, так как мы вычитаем
        // только меньшее из большего.
        // compareTo(a, b) положительно при a > b.
        if (compareTo (a, b) > 0) {
            subtract (a, b, gcd, new int[1]);
            assign (a, gcd);
        } else {
            subtract (b, a, gcd, new int[1]);
            assign (b, gcd);
        }
    }

    // В a находится вычисленное значение НОД исходных (a,b)
    assign (gcd, a);
}
```

Этот алгоритм многократно сравнивает два числа (a и b) и вычитает меньшее число из большего до достижения нуля. Реализации вспомогательных методов (`isZero`, `assign`, `compareTo`, `subtract`) можно найти в репозитории кода книги.

Этот алгоритм возвращает наибольший общий делитель двух чисел, но четкого ответа на вопрос о том, сколько (с учетом размера входных данных) итераций для этого потребуется, нет. При каждой итерации цикла либо a, либо b уменьшается, но при этом никогда не становится отрицательным; поэтому мы можем гарантировать, что алгоритм завершит работу, но одни запросы могут потребовать гораздо больше времени, чем другие; например при вычислении НОД(1000,1) алгоритм выполняет 999 шагов! Производительность этого алгоритма явно более чувствительна ко входным данным, чем производительность сложения или умножения; имеются различные экземпляры задачи одинакового размера, которые требуют очень разного количества вычислений. Данная реализация алгоритма демонстрировала наихудшую производительность, когда мы запрашивали вычисление НОД(10^k–1,1); при этом приходилось выполнять $n = 10^k - 1$ итераций. Так как мы уже показали, что сложение и вычитание имеют сложность $O(n)$, где n — размер входных данных, алгоритм вычисления НОД можно классифицировать как $O(n^2)$.

Производительность реализации вычисления НОД в примере 2.5 существенно хуже производительности реализации алгоритма из примера 2.6, в котором используется оператор вычисления остатка от целочисленного деления *a* на *b*.

Пример 2.6. Реализация алгоритма вычисления НОД с использованием операции получения остатка от деления

```
public static void modgcd (int a[], int b[], int gcd[]) {
    if (isZero(a)) { assign (gcd, a); return; }
    if (isZero(b)) { assign (gcd, b); return; }

    // Выравнивание a и b до одинакового количества знаков
    // и работа с копиями
    a = copy(normalize(a, b.length));
    b = copy(normalize(b, a.length));

    // Гарантируем, что a > b. Возврат тривиального НОД
    int rc = compareTo(a,b);
    if (rc == 0) { assign (gcd, a); return; }
    if (rc < 0) {
    int t[] = b;
    b = a;
    a = t;
    }

    int quot[] = new int[a.length];
    int remainder[] = new int[a.length];
    while (!isZero(b)) {
        int t[] = copy (b);
        divide (a, b, quot, remainder);
        assign (b, remainder);
        assign (a, t);
    }

    // В a хранится вычисленное значение НОД(a,b).
    assign (gcd, a);
}
```

Метод `modgcd` получает окончательное значение НОД более быстро, поскольку не тратит время на вычитание малых значений из больших в цикле `while`. Это отличие — не просто деталь реализации; оно отражает фундаментальный сдвиг в подходе алгоритма к решению задачи.

Вычисления, показанные на рис. 2.5 (и подытоженные в табл. 2.5), показывают результат генерации 142 случайных *n*-значных чисел и вычисления НОД для всех 10011 пар этих чисел.

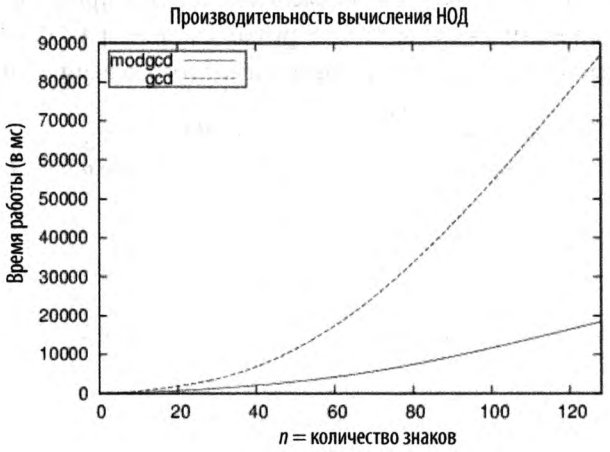


Рис. 2.5. Сравнение производительности методов `gcd` *и* `modgcd`

Таблица 2.5. Время (в мс) вычисления 10011 НОД

n	modgcd	gcd	$\text{modgcd}_{2n}/\text{modgcd}_n$
4	68	45	0,23
8	226	408	3,32
16	603	1315	2,67
32	1836	4050	3,04
64	5330	18392	2,90
128	20485	76180	3,84

Несмотря на то что реализация `modgcd` почти в три раза быстрее, чем соответствующая реализация `gcd` для тех же случайных данных, производительность `modgcd` является *квадратичной*, т.е. $O(n^2)$. Анализ этого алгоритма является сложной задачей, и в результате его выполнения оказывается, что наихудшие показатели для `modgcd` достигаются для двух последовательных чисел Фибоначчи. Тем не менее из табл. 2.5 можно сделать вывод о том, что алгоритм *квадратичен*, поскольку время работы алгоритма возрастает примерно в четыре раза при удвоении размера задачи.

Были разработаны и более сложные алгоритмы для вычисления НОД, хотя применение большинства из них нецелесообразно, за исключением разве что очень больших целых чисел; их анализ свидетельствует о том, что данная задача имеет и более эффективные по сравнению с квадратичными решения.

Экспоненциальная производительность

Рассмотрим замок с тремя числовыми дисками, каждый из которых содержит цифры от 0 до 9. На каждом диске можно независимо один от другого устанавливать одну из 10 цифр. Предположим, что у нас есть такой замок, но не открывающий его

код. Открытие такого замка является вопросом времени и некоторого труда, чтобы испытать каждую из 1000 возможных комбинаций — от 000 до 999. Чтобы обобщить эту задачу, предположим, что замок имеет n дисков, так что общее число возможных вариантов равно 10^n. Решение этой проблемы с помощью грубой силы рассматривается как имеющее экспоненциальную производительность, или $O(10^n)$, в данном случае по основанию 10. Часто основанием экспоненты является 2; но такая производительность справедлива для любого основания $b > 1$.

Экспоненциальные алгоритмы практичны только для очень малых значений n. Некоторые алгоритмы могут иметь экспоненциальное поведение в наихудшем случае и при этом активно использоваться на практике из-за их поведения в среднем случае. Хорошим примером является симплекс-метод для решения задач линейного программирования.

Резюме по асимптотическому росту

Алгоритм с лучшим асимптотическим ростом в конечном итоге будет выполняться быстрее, чем алгоритм с худшим асимптотическим ростом, независимо от реальных констант. Положение точки пересечения может быть различным в зависимости от конкретных констант, но такая точка существует и может быть оценена эмпирически. Кроме того, в ходе асимптотического анализа нас должен интересовать только наиболее быстро растущий член функции $t(n)$. По этой причине, если количество операций алгоритма можно вычислить как $c \cdot n^3 + d \cdot n \cdot \log n$, мы классифицируем этот алгоритм как $O(n^3)$, потому что n^3 является доминирующим членом, который растет гораздо быстрее, чем $n \cdot \log n$.

Эталонные операции

Оператор `**` языка программирования Python выполняет быстрое возведение в степень. Пример вычисления `2**851` показан ниже:

```
150150336576094004599423153910185137226235191870990070733
557987815252631252384634158948203971606627616971080383694
109252383653813326044865235229218132798103200794538451818
051546732566997782908246399595358358052523086606780893692
342385292277744791953321492 48
```

В Python вычисления относительно независимы от базовой платформы (например, вычисление 2^{851} в Java или C на большинстве платформ вызовет целочисленное переполнение). Но быстрое вычисление в Python дает результат, показанный выше. Является ли преимуществом или недостатком абстрагирование Python от базовой архитектуры? Рассмотрим две следующие гипотезы.

Гипотеза H1

Вычисление 2^n носит систематический характер, не зависящий от значения n.

Гипотеза H2

Большие числа (такие, как показанные ранее в развернутом виде) могут рассматриваться так же, как и любое другое число, например 123827 или 997.

Чтобы опровергнуть гипотезу H1, вычислим 10000 значений 2^n. Общее время выполнения для каждого значения n отображено на графике на рис. 2.6.

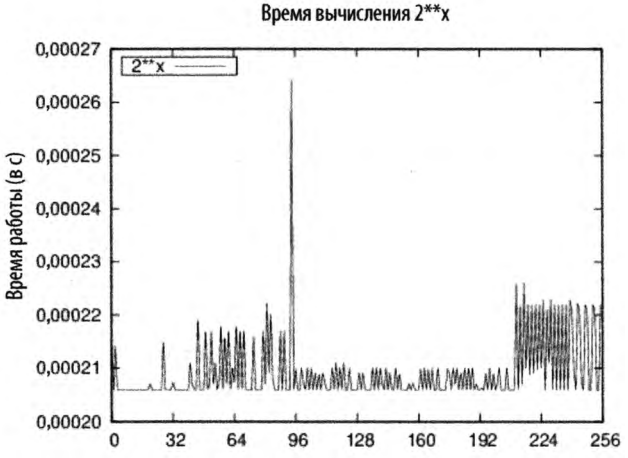


Рис. 2.6. Время вычисления 2^x в языке Python

Как ни странно, производительность, похоже, имеет различное поведение, одно — для x меньше 16, другое — для значений x около 145, а третье — для значений x более 200. Такое поведение показывает, что Python при возведении в степень с использованием оператора `**` применяет алгоритм возведения в степень путем возведения в квадрат. Вычисление 2^x вручную с использованием цикла `for` должно приводить к *квадратичной* производительности.

Чтобы опровергнуть гипотезу H2, мы проводим эксперимент, который выполняет предварительное вычисление значения 2^n, а затем оценивает время вычисления $\pi \cdot 2^n$. Общее время выполнения 10000 таких испытаний показано на рис. 2.7.

Почему точки на рис. 2.7 не находятся на прямой линии? При каком значении x наблюдается разрыв? Как представляется, операция умножения (`*`) в Python перегружена. Она выполняется по-разному в зависимости от того, являются ли перемножаемые значения числами с плавающей точкой или целыми числами, помещающимися в одно машинное слово, или числами, которые настолько велики, что должны храниться в нескольких машинных словах (или наблюдается некоторое сочетание этих крайностей).

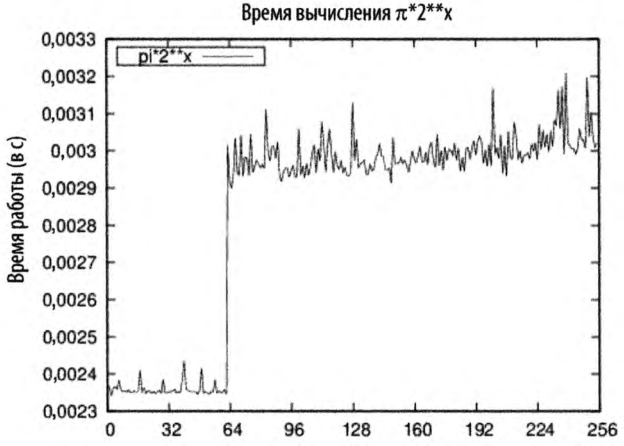


Рис. 2.7. Время вычисления произведения больших чисел

Разрыв в графике наблюдается при $x=\{64,65\}$ и, как представляется, соответствует изменению способа хранения больших чисел с плавающей точкой. И здесь вновь может обнаружиться замедление скорости роста вычислений, которое выявляется только с помощью выполнения подобного хронометража.

3

Строительные блоки алгоритмов

Мы создаем программное обеспечение для решения задач. Но программисты часто слишком сосредоточены на решении задачи, чтобы выяснять, нет ли уже готового ее решения. Даже если программист знает, что задача уже была решена, не очевидно, что существующий код будет соответствовать конкретной задаче, с которой работает программист. В конечном счете не так уж легко найти код на данном языке программирования, который мог бы быть легко изменен для решения конкретной задачи.

Размышлять об алгоритмах можно по-разному. Многие практики просто ищут алгоритм в книге или на веб-сайтах, копируют код, запускают его (а может быть, даже проверяют), а затем переходят к следующей задаче. По нашему мнению, такой процесс ничуть не улучшает понимание алгоритмов. В самом деле, такой подход может вывести вас на кривую дорожку неверного выбора конкретной реализации алгоритма.

Вопрос заключается не только в том, как найти правильный алгоритм для решения вашей задачи, но и в том, чтобы хорошо в нем разобраться и понять, как он работает, чтобы убедиться, что вы сделали правильный выбор. И, когда вы выбрали алгоритм, как эффективно его реализовать? Каждая глава нашей книги объединяет набор алгоритмов для решения стандартной задачи (например, для сортировки или поиска) или связанных с ней задач (например, поиска пути). В этой главе мы представим формат, используемый нами для описания алгоритмов в этой книге. Мы также подытожим общие алгоритмические подходы, используемые для решения программных задач.

Формат представления алгоритма

Реальная мощь применения шаблона для описания каждого алгоритма проявляется в том, что вы можете быстро сравнивать различные алгоритмы и выявлять общие особенности в, казалось бы, совершенно разных алгоритмах. Каждый алгоритм в книге представлен с использованием фиксированного набора разделов, соответствующих упомянутому шаблону. Мы можем также пропустить некоторый раздел, если он не добавляет ничего ценного к описанию алгоритма, или добавить разделы для освещения некоторой конкретной особенности алгоритма.

Название и краткое описание

Описательное название алгоритма. Мы используем это название для лаконичных ссылок одних алгоритмов на другие. Когда мы говорим об использовании **последовательного поиска**, это название точно передает, о каком именно алгоритме поиска мы говорим. Название каждого алгоритма всегда выделено полужирным шрифтом.

Входные и выходные данные алгоритма

Описывает ожидаемый формат входных данных алгоритма и вычисленных им значений.

Контекст применения алгоритма

Описание задачи, которое показывает, когда алгоритм является полезным и когда его производительность будет наилучшей. Описание свойств задачи/решения, которые необходимо рассмотреть для успешной реализации алгоритма. Это те соображения, которые должны привести вас к решению о выборе данного конкретного алгоритма для вашей задачи.

Реализация алгоритма

Описание алгоритма с использованием реального рабочего кода с документацией. Все исходные тексты решений можно найти в репозитории к книге.

Анализ алгоритма

Краткий анализ алгоритма, включая данные о производительности и информацию, которая должна помочь вам понять поведение алгоритма. Хотя раздел, посвященный анализу, не предназначен для формального доказательства указанной производительности алгоритма, вы должны быть в состоянии понять, почему алгоритм ведет себя именно таким образом. Чтобы пояснить описанное поведение алгоритмов,

мы предоставляем ссылки на работы, в которых представлены соответствующие леммы и доказательства.

Вариации алгоритма

Представляет вариации алгоритма или различные альтернативы.

Формат шаблона псевдокода

Каждый алгоритм в этой книге представлен с примерами кода, которые демонстрируют его реализацию на основных языках программирования, таких как Python, C, C++ и Java. Для читателей, которые не знакомы со всеми этими языками, мы описываем каждый алгоритм с помощью псевдокода с небольшим примером, показывающим его выполнение.

Рассмотрим следующий пример описания производительности, которое именует алгоритм и четко классифицирует его производительность для всех трех случаев (наилучший, средний, наихудший), описанных в главе 2, "Математика алгоритмов".

Последовательный поиск

Наилучший: $O(1)$; средний, наихудший: $O(n)$

```
search (A,t)
    for i=0 to n-1 do          ❶
        if A[i] = t then
            return true
    return false
end
```

❶ Последовательное обращение к каждому элементу от 0 до n-1.

Описание псевдокода преднамеренно краткое. Ключевые слова и имена функций приведены с использованием полужирного шрифта. Все переменные имеют имена, записываемые строчными буквами, в то время как массивы записываются прописными буквами, а обращение к их элементам выполняется с помощью индексной записи `[i]`. Отступы в псевдокоде показывают области видимости инструкций `if` и циклов `while` и `for`.

Перед тем как рассматривать исходные тексты реализаций на конкретных языках программирования, следует ознакомиться с кратким резюме алгоритма. После каждого резюме следует небольшой пример (наподобие показанного на рис. 3.1), призванный облегчить понимание работы алгоритма. Эти рисунки показывают выполнение алгоритмов в динамике, часто с изображением основных шагов алгоритма, представленных на рисунке сверху вниз.

Рис. 3.1. Пример выполнения последовательного поиска

Формат эмпирических оценок

Мы подтверждаем производительность каждого алгоритма путем выполнения ряда хронометражей решения задач, соответствующих индивидуальным алгоритмам. Приложение А содержит более подробную информацию о механизмах, используемых для хронометража. Для правильной оценки производительности набор тестов состоит из множества k индивидуальных испытаний (обычно $k \geq 10$). Лучшее и худшее выполнения отбрасываются, остальные же $k-2$ испытания проходят статистическую обработку с вычислением среднего значения и стандартного отклонения. Экземпляры задач, приводимые в таблицах, обычно имеют размеры от $n=2$ до 2^{20}.

Вычисления с плавающей точкой

Поскольку ряд алгоритмов в этой книге включает числовые вычисления, нам нужно описать мощь и ограничения современных компьютеров в этой области. Компьютеры выполняют основные вычисления со значениями, хранящимися в регистрах центрального процессора (CPU). По мере того, как компьютерные архитектуры эволюционировали от 8-разрядных процессоров Intel, популярных в 1970-х годах, до сегодняшних получивших широкое распространение 64-битных архитектур (например, процессоры Intel Itanium и Sun Microsystems Sparc), эти регистры существенно выросли в размерах. Процессор часто поддерживает базовые операции — такие, как сложение, умножение, вычитание и деление — для целочисленных значений, хранящихся в этих регистрах. Устройства для вычислений с плавающей точкой (FPU) могут эффективно выполнять вычисления в соответствии со стандартом IEEE для бинарной арифметики с плавающей точкой (IEEE 754).

Математические вычисления на основе целочисленных значений (таких, как логические значения, 8-битные байты и 16- и 32-разрядные целые числа) традиционно являются наиболее эффективными вычислениями, выполняемыми процессором. Программы часто оптимизированы таким образом, чтобы использовать эту историческую разницу в производительности между вычислениями с целыми числами и с числами с плавающей точкой. Однако в современных процессорах

производительность вычислений с плавающей точкой значительно улучшена. Поэтому важно, чтобы разработчики были ознакомлены со следующими вопросами программирования арифметики с плавающей точкой [28].

Производительность

Общепризнано, что вычисления с целочисленными значениями будут более эффективными, чем аналогичные вычисления с плавающей точкой. В табл. 3.1 перечислены значения времени выполнения 10000000 операций, включая результаты в Linux времен первого издания этой книги и результаты Sparc Ultra-2 1996 года. Как видите, производительность отдельных операций может значительно отличаться от платформы к платформе. Эти результаты показывают также огромные улучшения процессоров в течение последних двух десятилетий. Для некоторых результатов указано нулевое время выполнения, так как они быстрее, чем имеющиеся средства хронометража.

Таблица 3.1. Время (в секундах) выполнения 10000000 операций

Операция	Sparc Ultra-2	Linux i686	В настоящее время
32-битное целое `CMP`	0,811	0,0337	0,0000
32-битное целое `MUL`	2,372	0,0421	0,0000
32-битное `float MUL`	1,236	0,1032	0,02986
64-битное `double MUL`	1,406	0,1028	0,02987
32-битное `float DIV`	1,657	0,1814	0,02982
64-битное `double DIV`	2,172	0,1813	0,02980
128-битное `double MUL`	36,891	0,2765	0,02434
32-битное целое `DIV`	3,104	0,2468	0,0000
32-битное `double SQRT`	3,184	0,2749	0,0526

Ошибка округления

Все вычисления с использованием значений с плавающей точкой могут привести к ошибкам округления, вызванным бинарным представлением чисел с плавающей точкой. В общем случае число с плавающей точкой является конечным представлением, разработанным для приближения действительного числа, бинарное представление которого может быть бесконечным. В табл. 3.2 показана информация о представлениях чисел с плавающей точкой и представлении конкретного значения `3.88f`.

Таблица 3.2. Представления чисел с плавающей точкой

Тип	Знак, бит	Экспонента, бит	Мантисса, бит
`float`	1	8	23
`double`	1	11	52

Пример бинарного представления `3.88f` как `0x407851ec`

`01000000 01111000 01010001 11101100` (всего 32 бита)

`seeeeeee emmmmmmm mmmmmmmm mmmmmmmm`

Тремя следующими за `3.88f` 32-битными представлениями чисел с плавающей точкой (и их значениями) являются:

- `0x407851ed`: `3.8800004`
- `0x407851ee`: `3.8800006`
- `0x407851ef`: `3.8800008`

Вот некоторые случайным образом выбранные 32-битные значения с плавающей точкой:

- `0x1aec9fae`: `9.786529E-23`
- `0x622be970`: `7.9280355E20`
- `0x18a4775b`: `4.2513525E-24`

В 32-разрядном значении с плавающей точкой один бит используется для знака, 8 битов — для экспоненты и 23 бита — для мантиссы. В представлении Java "степень двойки может определяться путем трактовки битов экспоненты как положительного числа с последующим вычитанием из него значения смещения. Для типа float смещение равно 126" [59]. Если сохраненная экспонента равна 128, ее фактическое значение равно 128 – 126, или 2.

Для достижения наибольшей точности мантисса всегда нормализована таким образом, что крайняя слева цифра всегда равна 1; этот бит *в действительности не должен храниться*, но воспринимается FPU как часть числа. В предыдущем примере мантисса представляет собой

`.[1]11110000101000111101100 =`

[1/2] + 1/4 + 1/8 + 1/16 + 1/32 + 1/1 024 + 1/4 096 + 1/65 536 + 1/131 072 + 1/262 144 + 1/524 288 + + 1/20 974 152 + 1/4 194 304,

что равно в точности значению `0.970000028610229492187`5.

При хранении с использованием этого представления значения `3.88f` приближенное значение равно $+1 \cdot 0{,}9700000286102294921875 \cdot 2^2$, т.е. в точности `3.8800011444091796875`. Ошибка этого значения составляет `~0.0000001`. Наиболее распространенный способ описания ошибок чисел с плавающей точкой — использование *относительной ошибки*, которая вычисляется как отношение абсолютной ошибки к точному значению. В данном случае относительная ошибка составляет `0.0000001144091796875/3.88`, или `2.9E-8`. Относительные ошибки достаточно часто оказываются меньше одной миллионной.

Сравнение значений с плавающей точкой

Поскольку значения с плавающей точкой являются всего лишь приблизительными, простейшие операции с плавающей точкой становятся подозрительными. Рассмотрим следующую инструкцию:

```
if (x == y) { ... }
```

Действительно ли эти два числа с плавающей запятой должны быть точно равны? Или достаточно, чтобы они были просто приблизительно равны (что обычно описывается знаком "≈")? Не может ли случиться так, что два значения хотя и различны, но достаточно близки, так что они должны рассматриваться как одно и то же значение? Рассмотрим практический пример: три точки, $p_0=(a,b)$, $p_1=(c,d)$ и $p_2=(e,f)$, в декартовой системе координат определяют упорядоченную пару отрезков (p_0,p_1) и (p_1,p_2). Значение выражения $(c-a)(f-b)-(d-b)(e-a)$ позволяет определить, коллинеарны ли эти два отрезка (т.е. находятся ли они на одной линии). Если это значение

- $=0$, то отрезки коллинеарны;
- <0, то отрезки повернуты влево (против часовой стрелки);
- >0, то отрезки повернуты вправо (по часовой стрелке).

Чтобы показать, как при вычислениях Java могут возникать ошибки с плавающей точкой, определим три точки, используя значения от *a* до *f* из табл. 3.3.

Таблица 3.3. Арифметические ошибки с плавающей точкой

	32-битное значение (`float`)	**64-битное значение** (`double`)
$a = 1/3$	`0.33333334`	`0.3333333333333333`
$b = 5/3$	`1.6666666`	`1.6666666666666667`
$c = 33$	`33.0`	`33.0`
$d = 165$	`165.0`	`165.0`
$e = 19$	`19.0`	`19.0`
$f = 95$	`95.0`	`95.0`
$(c-a)(f-b)-(d-b)(e-a)$	`4.8828125E-4`	`-4.547473508864641E-13`

Как вы можете легко определить, три точки, p_0, p_1 и p_2, коллинеарны и лежат на прямой $y=5{\cdot}x$. При вычислении тестового значения с плавающей точкой для проверки коллинеарности на результат вычисления влияют имеющиеся ошибки, присущие арифметике с плавающей точкой. Использование при вычислениях 32-битных значений с плавающей точкой дает значение 0,00048828125; использование 64-битных значений дает очень небольшое отрицательное число. Этот пример показывает, что как 32-, так и 64-разрядные представления чисел с плавающей точкой не дают истинные значения математических вычислений. И в рассмотренном случае результаты вычислений приводят к разногласиям по поводу того, представляют ли точки угол по часовой стрелке, против часовой стрелки или коллинеарные отрезки. Таков уж мир вычислений с плавающей точкой.

Одним из распространенных решений для этой ситуации является введение небольшого значения δ для определения операции "≈" (приблизительного равенства) между двумя значениями с плавающей точкой. Согласно этому решению, если $|x-y|<\delta$, то мы считаем *x* и *y* равными. Тем не менее даже при такой простой мере

возможна ситуация, когда $x \approx y$ и $y \approx z$, но при этом условие $x \approx z$ не выполняется. Это нарушает математический принцип *транзитивности* и осложняет написание правильного кода. Кроме того, это решение не решает задачу коллинеарности, в которой при принятии решения используется знак полученного значения (нулевое, положительное или отрицательное).

Специальные значения

Хотя все возможные 64-битные значения могут представлять допустимые числа с плавающей точкой, стандарт IEEE определяет несколько значений (табл. 3.4), которые интерпретируются как специальные значения (и часто не могут участвовать в стандартных математических вычислениях, таких как сложение или умножение). Эти значения были разработаны для упрощения восстановления после распространенных ошибок, таких как деление на нуль, квадратный корень из отрицательного числа, переполнение и потеря значимости. Обратите внимание, что несмотря на то, что они могут использоваться в вычислениях, значения положительного и отрицательного нулей также включены в эту таблицу.

Таблица 3.4. Специальные значения IEEE 754

Специальное значение	64-битное представление IEEE 754
Положительная бесконечность	`0x7ff0000000000000L`
Отрицательная бесконечность	`0xfff0000000000000L`
Не число (NaN)	От `0x7ff0000000000001L` до `0x7fffffffffffffffL` и от `0xfff0000000000001L` до `0xffffffffffffffffL`
Отрицательный нуль	`0x8000000000000000`
Положительный нуль	`0x0000000000000000`

Эти специальные значения могут быть результатом вычислений, которые выходят за пределы допустимых границ представлений чисел. Выражение `1/0.0` в Java вычисляется как положительная бесконечность. Если вместо этого инструкция будет записана как `double x = 1/0`, то виртуальная машина Java сгенерирует исключение `ArithmeticException`, поскольку это выражение выполняет целочисленное деление двух чисел.

Пример алгоритма

Чтобы проиллюстрировать наш шаблон алгоритма, опишем алгоритм **сканирования Грэхема** для вычисления выпуклой оболочки множества точек. Это задача, представленная в главе 1, "Мысли алгоритмически", и показанная на рис. 1.3.

Название и краткое описание

Сканирование Грэхема вычисляет выпуклую оболочку множества точек на декартовой плоскости. Алгоритм находит нижнюю точку *low* во входном множестве *P* и сортирует оставшиеся точки {*P*–*low*} по полярному углу относительно нижней точки в *обратном* порядке. Этот порядок позволяет алгоритму обойти *P* по часовой стрелке от самой низкой точки. Каждый левый поворот для последних трех точек в строящейся оболочке показывает, что последняя точка оболочки была выбрана неправильно и может быть удалена.

Входные и выходные данные алгоритма

Экземпляр задачи построения выпуклой оболочки определяется множеством точек *P*.

Вывод представляет собой последовательность точек (x,y), представляющую обход выпуклой оболочки по часовой стрелке. Какая именно точка выбрана первой, значения не имеет.

Контекст применения алгоритма

Этот алгоритм подходит для декартовых точек. Если точки используют иную систему координат, например такую, в которой бо́льшие значения *y* отражают более низкие точки на плоскости, то алгоритм должен соответственно вычислять нижнюю точку *low*. Сортировка точек по полярному углу требует тригонометрических вычислений.

Сканирование Грэхема

Наилучший, средний, наихудший случаи: $O(n \cdot \log n)$

```
graham(P)
    low = точка с минимальной координатой y в P                    ❶
    Удаляем low из P
    Сортируем P по убыванию полярного угла относительно low        ❷

    hull = {P[n-2], low}                                           ❸
    for i = 0 to n-1 do
      while(isLeftTurn(secondLast(hull),last(hull),P[i])) do
        Удаляем последнюю точку из оболочки                        ❹

        Добавляем P[i] к оболочке
    Удаляем дубликат последней точки                               ❺
    Возвращаем оболочку
```

❶ Точки с одинаковыми значениями y сортируются по координате x
❷ P[0] имеет максимальный полярный угол, а P[n-2] — минимальный
❸ Оболочка строится по часовой стрелке начиная с минимального полярного угла и low.
❹ Каждый поворот влево указывает, что последняя точка оболочки должна быть удалена
❺ Поскольку это должна быть P[n-2]

Реализация алгоритма

Если вы решаете эту задачу вручную, вероятно, у вас не возникает проблемы с отслеживанием соответствующих краев, но может оказаться трудно объяснить точную последовательность выполненных вами шагов. Ключевым шагом в этом алгоритме является сортировка точек по убыванию полярного угла относительно самой нижней точки множества. После упорядочения алгоритм переходит к обходу этих точек, расширяя частично построенную оболочку и изменяя ее структуру, если последние три точки оболочки делают левый поворот, что указывает на невыпуклую форму оболочки (пример 3.1).

Пример 3.1. Реализация сканирования Грэхема

```
public class NativeGrahamScan implements IConvexHull {
    public IPoint[] compute(IPoint[] pts) {
        int n = pts.length;
        if (n < 3) {
            return pts;
        }

        // Находим наинизшую точку и меняем ее с последней
        // точкой массива points[], если она не является таковой
        int lowest = 0;
        double lowestY = pts[0].getY();

        for (int i = 1; i < n; i++) {
            if (pts[i].getY() < lowestY) {
                lowestY = pts[i].getY();
                lowest = i;
            }
        }

        if (lowest != n - 1) {
            IPoint temp = pts[n - 1];
            pts[n - 1] = pts[lowest];
            pts[lowest] = temp;
        }
```

```
// Сортируем points[0..n-2] в порядке убывания полярного
// угла по отношению к наинизшей точке points[n-1].
new HeapSort<IPoint>().sort(pts,0,n-2,
                new ReversePolarSorter(pts[n - 1]));

// *Известно*, что три точки (в указанном порядке)
// находятся на оболочке - (points[n-2]), наинизшая точка
// (points[n-1]) и точка с наибольшим полярным углом
// (points[0]). Начинаем с первых двух
DoubleLinkedList<IPoint>list=new DoubleLinkedList<IPoint>();
list.insert(pts[n - 2]);
list.insert(pts[n - 1]);

// Если все точки коллинеарны, обрабатываем их, чтобы
// избежать проблем позже
double firstAngle=Math.atan2(pts[0].getY()-lowest,
                             pts[0].getX()-pts[n-1].getX());
double lastAngle=Math.atan2(pts[n-2].getY()-lowest,
                            pts[n-2].getX()-pts[n-1].getX());
if (firstAngle == lastAngle) {
    return new IPoint[] { pts[n - 1], pts[0] };
}

// Последовательно посещаем каждую точку, удаляя точки,
// приводящие к ошибке. Поскольку у нас всегда есть как
// минимум один "правый поворот", внутренний цикл всегда
// завершается
for (int i = 0; i < n - 1; i++) {
    while (isLeftTurn(list.last().prev().value(),
                      list.last().value(),
                      pts[i])) {
        list.removeLast();
    }

    // Вставляем следующую точку оболочки в ее
    // корректную позицию
    list.insert(pts[i]);
}

// Последняя точка дублируется, так что мы берем
// n-1 точек начиная с наинизшей
IPoint hull[] = new IPoint[list.size() - 1];
DoubleNode<IPoint> ptr = list.first().next();
int idx = 0;

while (idx < hull.length) {
    hull[idx++] = ptr.value();
    ptr = ptr.next();
}
```

```
        return hull;
    }
    /** Проверка коллинеарности для определения левого поворота. */
    public static boolean isLeftTurn(IPoint p1,IPoint p2,IPoint p3){
        return (p2.getX()-p1.getX())*(p3.getY()-p1.getY())-
               (p2.getY()-p1.getY())*(p3.getX()-p1.getX())>0;
    }
}

/** Клас сортировки в обратном порядке по полярному углу
    относительно наинизшей точки */
class ReversePolarSorter implements Comparator<IPoint> {
    /** Сохраненные координаты x,y базовой точки для сравнения. */
    final double baseX;
    final double baseY;
    /** PolarSorter вычисляет все точки, сравниваемые с базовой. */
    public ReversePolarSorter(IPoint base) {
        this.baseX = base.getX();
        this.baseY = base.getY();
    }
    public int compare(IPoint one, IPoint two) {
        if (one == two) {
            return 0;
        }

        // Оба угла вычисляются с помощью функции atan2.
        // Код работает, так как one.y всегда не меньше base.y
        double oneY = one.getY();
        double twoY = two.getY();
        double oneAngle = Math.atan2(oneY-baseY, one.getX()-baseX);
        double twoAngle = Math.atan2(twoY-baseY, two.getX()-baseX);

        if (oneAngle > twoAngle) {
            return -1;
        } else if (oneAngle < twoAngle) {
            return +1;
        }

        // Если углы одинаковы, для корректной работы алгоритма
        // необходимо упорядочение в убывающем порядке по высоте
        if (oneY > twoY) {
            return -1;
        } else if (oneY < twoY) {
            return +1;
        }

        return 0;
    }
}
```

Если все $n>2$ точек коллинеарны, то в данном частном случае оболочка состоит из двух крайних точек множества. Вычисленная выпуклая оболочка может содержать несколько последовательных точек, являющихся коллинеарными, поскольку мы не предпринимаем попыток их удаления.

Анализ алгоритма

Сортировка n точек требует $O(n \cdot log n)$ точек, как известно из главы 4, "Алгоритмы сортировки". Остальная часть алгоритма состоит из цикла `for`, который выполняется n раз; но сколько раз выполняется его внутренний цикл `while`? До тех пор, пока имеется левый поворот, точка удаляется из оболочки; и так до тех пор, пока не останутся только первые три точки. Поскольку в оболочку включаются не более n точек, внутренний цикл `while` может выполняться в общей сложности не более чем n раз. Таким образом, время работы цикла `for` — $O(n)$. В результате общая производительность алгоритма равна $O(n \cdot log n)$, поскольку стоимость сортировки доминирует в стоимости всех вычислений.

Распространенные подходы

В этом разделе представлены основные алгоритмические подходы, используемые в данной книге. Вы должны понимать эти общие стратегии решения задач, чтобы видеть, каким образом они могут быть применены для решения конкретных задач. В главе 10, "Пространственные древовидные структуры", описываются некоторые дополнительные стратегии, такие как поиск приемлемого приблизительного решения или использование рандомизации с большим количеством испытаний для сходимости к правильному результату, не прибегая к исчерпывающему поиску.

Жадная стратегия

Жадная стратегия решает задачу размера n пошагово. На каждом шаге жадный алгоритм находит наилучшее локальное решение, которое можно получить с учетом имеющейся информации; обычно размер задачи при этом уменьшается на единицу. После того, как будут завершены все n шагов, алгоритм возвращает вычисленное решение.

Например, для сортировки массива A из n чисел жадная **сортировка выбором** находит наибольшее значение из $A[0, n-1]$ и обменивает его с элементом в позиции $A[n-1]$, что гарантирует, что $A[n-1]$ находится в правильном месте. Затем процесс повторяется и выполняется поиск наибольшего значения среди оставшихся элементов $A[0, n-2]$, которое затем аналогичным образом меняется местами с элементом в позиции $A[n-2]$. Этот процесс продолжается до тех пор, пока не будет отсортирован весь массив. Более подробно этот процесс описан в главе 4, "Алгоритмы сортировки".

Жадная стратегия обычно характеризуется медленным уменьшением размера решаемых подзадач по мере обработки алгоритмом входных данных. Если подзадача может быть решена за время $O(\log n)$, то жадная стратегия будет иметь производительность $O(n \cdot \log n)$. Если подзадача решается за время $O(n)$, как в **сортировке выбором**, то общей производительностью алгоритма будет $O(n^2)$.

Разделяй и властвуй

Стратегия "Разделяй и властвуй" решает задачу размера n, разделяя ее на две независимые подзадачи, каждая около половины размера исходной задачи[1]. Довольно часто решение является рекурсивным, прерывающимся базовым случаем, который может быть решен тривиально. Должно также иметься некоторое вычисление *разрешения*, которое позволяет получить решение для задачи из решений меньших подзадач.

Например, чтобы найти наибольший элемент массива из n чисел, рекурсивная функция из примера 3.2 конструирует две подзадачи. Естественно, максимальный элемент исходной задачи является просто наибольшим значением из двух максимальных элементов, найденных в подзадачах. Обратите внимание, как рекурсия прекращается, когда размер подзадачи достигает значения 1; в этом случае возвращается единственный имеющийся элемент `vals[left]`.

Пример 3.2. Применение подхода "разделяй и властвуй" для поиска максимального значения массива

```
/** Применение рекурсии. */
public static int maxElement(int[] vals) {
    if (vals.length == 0) {
        throw new NoSuchElementException("Массив пуст.");
    }
    return maxElement(vals, 0, vals.length);
}

/** Вычисляем максимальный элемент в подзадаче vals[left, right).
 *  Правая конечная точка не является частью диапазона! */
static int maxElement (int[] vals, int left, int right) {
    if (right - left == 1) {
        return vals[left];
    }

    // Решение подзадач
    int mid = (left + right)/2;
    int max1 = maxElement(vals, left, mid);
    int max2 = maxElement(vals, mid, right);
```

[1] Эти условия не являются обязательными. Задача в общем случае может быть разбита и на большее количество подзадач, и подзадачи могут иметь разный размер. — *Примеч. ред.*

```
    // Разрешение: получение результата из решений подзадач
    if (max1 > max2) { return max1; }
    return max2;
}
```

Алгоритм "разделяй и властвуй", структурированный так, как показано в примере 3.2, демонстрирует производительность $O(n)$, если шаг разрешения может быть выполнен за постоянное время $O(1)$ (что и происходит). Если бы шагу разрешения самому требовалось время $O(n)$, то общая производительность алгоритма была бы $O(n \cdot log\ n)$. Обратите внимание, что найти наибольший элемент в массиве можно быстрее — путем сканирования каждого элемента и хранения текущего наибольшего значения. Пусть это будет кратким напоминанием, что подход "разделяй и властвуй" не всегда обеспечивает самое быстрое решение.

Динамическое программирование

Динамическое программирование является вариацией подхода "разделяй и властвуй", которая решает задачу путем ее разделения на несколько более простых подзадач, которые решаются в определенном порядке. Она решает каждую из более мелких задач только один раз и сохраняет результаты для последующего использования, чтобы избежать ненужного пересчета. Затем этот метод решает задачи большего размера и *объединяет* в единое решение результаты решения меньших подзадач. Во многих случаях найденное решение является для поставленной задачи доказуемо оптимальным.

Динамическое программирование часто используется для задач оптимизации, в которых цель заключается в минимизации или максимизации некоторого вычисления. Наилучший способ объяснения динамического программирования — на конкретном примере.

Ученые часто сравнивают последовательности ДНК для определения их сходства. Если вы представите такую последовательность ДНК как строку символов A, C, T и G, то задачу можно переформулировать как вычисление *минимального расстояния редактирования* между двумя строками. Иначе говоря, для заданной базовой строки s_1 и целевой строки s_2 требуется определить наименьшее число операций редактирования, которые преобразуют s_1 в s_2, если вы можете:

- заменить символ s_1 другим символом;
- удалить символ из s_1;
- вставить символ в s_1.

Например, для строки s_1, представляющей последовательность ДНК "GCTAC", нужны только три операции редактирования данной базовой строки, чтобы преобразовать ее в целевую строку s_2, значение которой — "CTCA":

- заменить четвертый символ ("A") символом "C";
- удалить первый символ ("G");
- заменить последний символ "C" символом "A".

Это не единственная последовательность операций, но вам нужно как минимум три операции редактирования для преобразования s_1 в s_2. Для начала цель заключается в том, чтобы вычислить значение оптимального ответа, т.е. количество операций редактирования, а не фактическую последовательность операций.

Динамическое программирование работает путем сохранения результатов более простых подзадач; в данном примере можно использовать двумерную матрицу $m[i][j]$, чтобы записывать результат вычисления минимального расстояния редактирования между первыми i символами s_1 и первыми j символами s_2. Начнем с создания следующей исходной матрицы:

```
0 1 2 3 4
1 . . . .
2 . . . .
3 . . . .
4 . . . .
5 . . . .
```

В этой таблице каждая строка индексируется с помощью i, а каждый столбец индексируется с помощью j. По завершении работы запись $m[0][4]$ (верхний правый угол таблицы) будет содержать результат редактирования расстояния между первыми 0 символами s_1 (т.е. пустой строкой "") и первыми четырьмя символами s_2 (т.е. всей строкой "CTCA"). Значение $m[0][4]$ равно 4, потому что вы должны вставить четыре символа в пустую строку, чтобы она стала строкой s_2. Аналогично значение $m[3][0]$ равно 3, потому что, имея первые три символа s_1 (т.е. "GCT"), получить из них первые нуль символов s_2 (т.е. пустую строку "") можно путем удаления всех трех символов.

Хитрость динамического программирования заключается в цикле оптимизации, который показывает, как соединить результаты решения этих подзадач в решение больших задач. Рассмотрим значение $m[1][1]$, которое представляет собой расстояние редактирования между первым символом s_1 ("G") и первым символом s_2 ("C"). Есть три варианта:

- заменить символ "G" символом "C" с ценой 1;
- удалить символ "G" и вставить "C" с ценой 2;
- вставить символ "C" и удалить "G" с ценой 2.

Понятно, что нам нужна минимальная стоимость, так что $m[1][1] = 1$. Как же можно обобщить это решение? Рассмотрим вычисление, показанное на рис. 3.2.

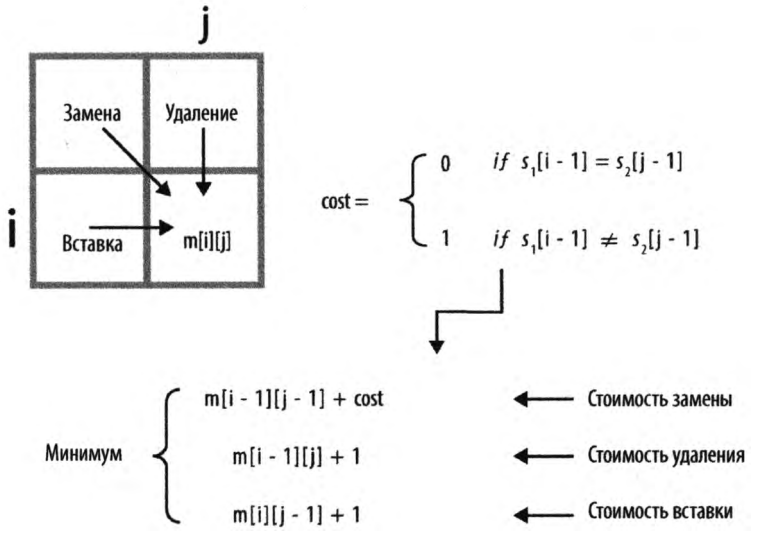


Рис. 3.2. Вычисление m[i][j]

Три варианта вычисления $m[i][j]$ таковы.

Стоимость замены

Вычисляем расстояние редактирования между первыми $i-1$ символами s_1 и первыми $j-1$ символами s_2 и добавляем 1 для замены j-го символа s_2 i-м символом s_1, если они различны.

Стоимость удаления

Вычисляем расстояние редактирования между первыми $i-1$ символами s_1 и первыми j символами s_2 и добавляем 1 для удаления i-го символа s_1.

Стоимость вставки

Вычисляем расстояние редактирования между первыми i символами s_1 и первыми $j-1$ символами s_2 и добавляем 1 для вставки j-го символа s_2.

Визуализируя вычисления, вы увидите, что динамическое программирование должно решать подзадачи в правильном порядке (от верхней строки к нижней строке и слева направо в каждой строке, как показано в примере 3.3). Вычисление проходит по строке от значения индекса $i = 1$ до $len(s_1)$. После того как матрица m будет заполнена начальными значениями, вложенный цикл `for` вычисляет минимальное значение для каждой из подзадач в указанном порядке, пока не будут вычислены все значения в m. Этот процесс не является рекурсивным, вместо этого он использует

результаты предыдущих вычислений для задач меньшего размера. Решение полной задачи находится в $m[len(s_1)][len(s_2)]$.

Пример 3.3. Поиск минимального расстояния редактирования с использованием динамического программирования

```
def minEditDistance(s1, s2):
    """ Вычисление минимального расстояния редактирования s1->s2."""
    len1 = len(s1)
    len2 = len(s2)

    # Создание двумерной структуры, такой, что m[i][j] = 0
    # for i in 0 .. len1 и for j in 0 .. len2
    m = [None] * (len1 + 1)
    for i in range(len1+1):
        m[i] = [0] * (len2+1)

    # Настройка начальных значений по горизонтали и вертикали
    for i in range(1, len1+1):
        m[i][0] = i
    for j in range(1, len2+1):
        m[0][j] = j

    # Вычисление наилучших значений
    for i in range(1,len1+1):
        for j in range(1,len2+1):
            cost = 1
            if s1[i-1] == s2[j-1]: cost = 0
            replaceCost = m[i-1][j-1] + cost
            removeCost = m[i-1][j] + 1
            insertCost = m[i][j-1] + 1
            m[i][j] = min(replaceCost,removeCost,insertCost)
    return m[len1][len2]
```

В табл. 3.5 показаны окончательные значения элементов m.

Таблица 3.5. Результаты решения всех подзадач

0	1	2	3	4
1	1	2	3	4
2	1	2	2	3
3	2	1	2	3
4	3	2	2	2
5	4	3	2	3

Стоимость подзадачи $m[3][2] = 1$ представляет собой расстояние редактирования между строками "GCT" и "CT". Как видите, вам нужно только удалить первый символ, что подтверждает корректность найденной стоимости. Однако этот код

показывает только, как вычислить расстояние минимального редактирования; чтобы записать последовательность операций, которые при этом требуется выполнить, нужно записывать в матрицу *prev*[*i*][*j*], какой именно из трех случаев был выбран при вычислении минимального значения *m*[*i*][*j*]. Чтобы восстановить эти операции, надо просто выполнить обратный проход от $m[len(s_1)][len(s_2)]$ с использованием решений, записанных в *prev*[*i*][*j*], и остановиться по достижении *m*[0][0]. Это модифицированное решение приведено в примере 3.4.

Пример 3.4. Поиск минимального расстояния редактирования и операций с использованием динамического программирования

```
REPLACE = 0
REMOVE = 1
INSERT = 2
def minEditDistance(s1, s2):
    """ Вычисление минимального расстояния
        редактирования и операций s1->s2."""
    len1 = len(s1)
    len2 = len(s2)

    # Создание двумерной структуры, такой, что m[i][j] = 0
    # for i in 0 .. len1 and for j in 0 .. len2
    m = [None] * (len1 + 1)
    op = [None] * (len1 + 1)
    for i in range(len1+1):
        m[i] = [0] * (len2+1)
        op[i] = [-1] * (len2+1)

    # Настройка начальных значений по горизонтали и вертикали
    for j in range(1, len2+1):
        m[0][j] = j
    for i in range(1, len1+1):
        m[i][0] = i

    # Вычисление наилучших значений
    for i in range(1,len1+1):
        for j in range(1,len2+1):
            cost = 1

            if s1[i-1] == s2[j-1]: cost = 0
            replaceCost = m[i-1][j-1] + cost
            removeCost = m[i-1][j] + 1
            insertCost = m[i][j-1] + 1
            costs = [replaceCost,removeCost,insertCost]
            m[i][j] = min(costs)
            op[i][j] = costs.index(m[i][j])
```

```
ops = []
i = len1
j = len2
while i != 0 or j != 0:
    if op[i][j] == REMOVE or j == 0:
        ops.append('remove {} char {} of {}'.format(i,s1[i-1],s1))
        i = i-1
    elif op[i][j] == INSERT or i == 0:
        ops.append('insert {} char {} of {}'.format(j,s2[j-1],s2))
        j = j-1
    else:
        if m[i-1][j-1] < m[i][j]:
            fmt='replace {} char of {} ({}) with {}'
            ops.append(fmt.format(i,s1,s1[i-1],s2[j-1]))
        i,j = i-1,j-1

return m[len1][len2], ops
```

4

Алгоритмы сортировки

Многочисленные расчеты и задачи упрощаются, если заранее выполнить соответствующую сортировку информации. На заре вычислительной техники велся очень активный поиск эффективных алгоритмов сортировки. В то время большинство исследований алгоритмов было сосредоточено на сортировке наборов данных, которые были слишком велики для хранения в памяти имевшихся в то время компьютеров. Так как современные компьютеры гораздо более мощные, чем компьютеры 50-летней давности, размер обрабатываемых наборов данных в настоящее время составляет порядка терабайтов информации. Хотя вам вряд ли понадобится сортировка таких огромных наборов данных, вряд ли вы минуете в своей практике сортировку большого количества элементов. В этой главе мы рассмотрим наиболее важные алгоритмы сортировки и представим результаты наших хронометражей, чтобы помочь вам выбрать лучший алгоритм для использования в любой ситуации, где требуется сортировка.

Терминология

Коллекция A элементов, которые можно сравнивать один с другим, сортируется на месте (без привлечения дополнительной памяти). Мы используем обозначения $A[i]$ и a_i для обозначения i-го элемента коллекции. По соглашению первым элементом коллекции является $A[0]$. Мы используем запись $A[low, low+n)$ для обозначения подколлекции $A[low]...A[low+n-1]$ из n элементов, в то время как подколлекция $A[low, low+n]$ содержит $n+1$ элементов.

Для сортировки коллекции необходимо реорганизовать элементы A так, что если $A[i] < A[j]$, то $i < j$. Если в коллекции имеются повторяющиеся элементы, то они должны быть смежными в результирующей упорядоченной коллекции; т.е. если в отсортированной коллекции $A[i] = A[j]$, то не может быть такого k, что $i < k < j$ и $A[i] \neq A[k]$. Наконец отсортированная коллекция A должна представлять собой перестановку элементов, которые первоначально образовывали коллекцию A.

Представление

Коллекция может храниться в оперативной памяти компьютера, но может находиться и в файле в файловой системе, известной как вторичная память. Коллекция может быть заархивирована в третичной памяти (например, ленточные библиотеки или оптические диски), что может потребовать дополнительного времени только для обнаружения информации; кроме того, прежде чем она сможет быть обработана, возможно, потребуется скопировать такую информацию во вторичную память (например, на жесткий диск).

Информация, хранящаяся в оперативной памяти, обычно принимает одну из двух форм: на основе указателей или на основе значений. Предположим, мы хотим отсортировать строки "eagle", "cat", "ant", "dog" и "ball".[1] При хранении с использованием указателей, показанном на рис. 4.1, массив информации (т.е. смежные поля) хранит указатели на фактическую информацию (здесь — строки в овалах), а не саму информацию в виде строк. Такой подход позволяет хранить и сортировать произвольно сложные записи.

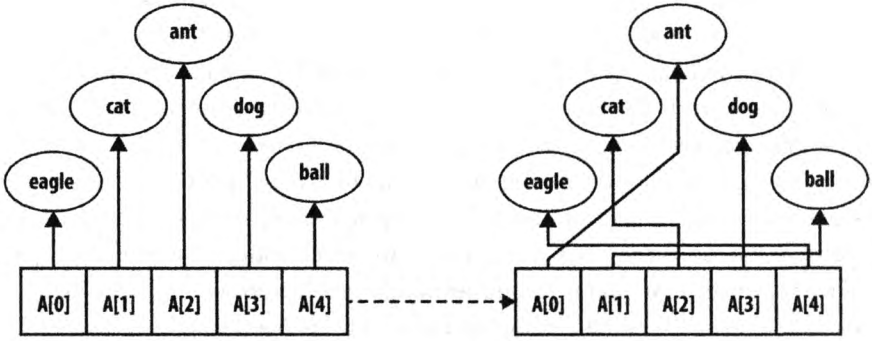


Рис. 4.1. Сортировка с использованием хранения указателей

Напротив, хранение на основе значений упаковывает коллекцию из n элементов в блоки записей фиксированного размера s; этот метод более подходит для вторичной или третичной памяти. На рис. 4.2 показано, как хранятся те же сведения, что и показанные на рис. 4.1, с использованием непрерывного блока памяти, содержащего множество строк длиной по $s=5$ байтов. В этом примере информация представлена в виде строк, но это может быть любая коллекция структурированной информации на основе записей. Символ "¬" представляет собой символ заполнения, который не может быть частью никакой строки; при такой кодировке строки длиной s в символе заполнения не нуждаются. Информация располагается непрерывно и может рассматриваться как одномерный массив $B[0,n{\cdot}s)$. Заметим, что $B[r{\cdot}s{+}c]$ представляет собой c-ю букву r-го слова (где $c\geq 0$ и $r\geq 0$). Кроме того, i-й элемент коллекции ($i\geq 0$) представляет собой подмассив $B[i{\cdot}s,(i{+}1){\cdot}s)$.

[1] Эти строки преднамеренно не переведены на русский язык, чтобы не поднимать не относящийся к собственно сортировке вопрос кодировок различных алфавитов. — *Примеч. пер.*

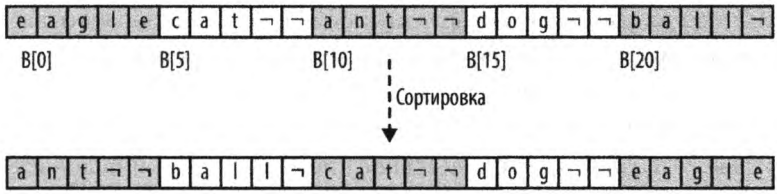


Рис. 4.2. Сортировка с использованием хранения на основе значений

Информация обычно записывается во вторичную память как основанный на значениях непрерывный набор байтов. Алгоритмы в этой главе могут быть переписаны для работы с информацией на диске просто путем реализации функции `swap`, которая меняет местами байты в файлах на диске. Однако итоговая производительность будет резко отличаться из-за увеличения затрат на ввод-вывод при обращении ко вторичной памяти. Для сортировки данных во вторичной памяти особенно хорошо подходит **сортировка слиянием**.

Независимо от способа хранения алгоритм сортировки обновляет информацию так, что коллекция $A[0, n)$ становится упорядоченной. Для удобства мы будем использовать запись $A[i]$ для представления i-го элемента даже при использовании хранения на основе значений.

Сравниваемость элементов

Элементы в наборе данных сравнений должны обеспечивать нестрогое упорядочение. То есть для любых двух элементов p и q из набора данных должен выполняться ровно один из следующих трех предикатов: $p=q$, $p<q$ или $p>q$. Распространенные сортируемые типы-примитивы включают целые числа, числа с плавающей точкой и символы. При сортировке составных элементов (таких, например, как строки символов) на каждую отдельную часть составного элемента накладывается лексикографическое упорядочивание, сводя, таким образом, сложную сортировку к отдельным сортировкам примитивных типов. Например, слово "алфавит" считается меньшим, чем "альтернатива", но бо́льшим, чем "аллигатор", — согласно сравнению каждого отдельного символа слева направо, пока не встретится пара различных символов, находящихся в совпадающих позициях двух слов, или пока у одного из слов не закончатся символы (таким образом, "воспитанник" меньше "воспитателя", а "трактор" меньше "тракториста").

Вопрос упорядочения далеко не так прост при рассмотрении регистров символов ("A" больше или меньше "a"?), диакритических знаков (как соотносятся "á" и "ã"?) и дифтонгов (то же для "a" и "æ"?). Заметим, что мощный стандарт Unicode (`http://www.unicode.org/versions/latest`) использует для представления каждого отдельного знака кодировки, например, UTF-16, где для представления каждого отдельного символа может потребоваться до четырех байтов. Консорциум Unicode (`http://`

www.unicode.org) разработал стандарт сортировки, который обрабатывает широкий спектр правил упорядочения, обнаруженных в разных языках и культурах [21].

В алгоритмах, представленных в этой главе, предполагается, что вы можете предоставить *компаратор* — функцию сравнения `cmp`, которая сравнивает элементы p и q и возвращает 0, если $p=q$, отрицательное число, если $p<q$, и положительное число при $p>q$. Если элементы представляют собой сложные составные записи, функция `cmp` может сравнивать только значения "ключей" элементов. Например, терминал аэропорта может перечислять вылетающие рейсы в возрастающем порядке названий городов назначения или времени вылета, в то время как номера рейсов окажутся неупорядоченными.

Устойчивая сортировка

Когда функция сравнения `cmp` определяет, что два элемента исходной неупорядоченной коллекции a_i и a_j равны, может оказаться важным сохранить их относительный порядок в отсортированном множестве, т.е. если $i<j$, то окончательное расположение a_i должно быть слева от окончательного местоположения a_j. Алгоритмы сортировки, которые гарантируют выполнение этого свойства, называются *устойчивыми*. Например, в четырех левых столбцах табл. 4.1 показан исходный набор полетной информации, уже отсортированный по времени вылета в течение дня (независимо от авиакомпании или города назначения). Если выполнить устойчивую сортировку этих данных с помощью функции сравнения, упорядочивающей записи рейсов по городу назначения, то будет получен единственно возможный результат, показанный в четырех правых столбцах табл. 4.1.

Таблица 4.1. Устойчивая сортировка информации с терминала аэропорта

Город назначения	Компания	Рейс	Время вылета (в порядке возрастания)	→	Город назначения (в порядке возрастания)	Компания	Рейс	Время вылета
Buffalo	Air Trans	549	10:42		Albany	Southwest	482	13:20
Atlanta	Delta	1097	11:00		Atlanta	Delta	1097	11:00
Baltimore	Southwest	836	11:05		Atlanta	Air Trans	872	11:15
Atlanta	Air Trans	872	11:15		Atlanta	Delta	28	12:00
Atlanta	Delta	28	12:00		Atlanta	Al Italia	3429	13:50
Boston	Delta	1056	12:05		Austin	Southwest	1045	13:05
Baltimore	Southwest	216	12:20		Baltimore	Southwest	836	11:05
Austin	Southwest	1045	13:05		Baltimore	Southwest	216	12:20
Albany	Southwest	482	13:20		Baltimore	Southwest	272	13:40
Boston	Air Trans	515	13:21		Boston	Delta	1056	12:05
Baltimore	Southwest	272	13:40		Boston	Air Trans	515	13:21
Atlanta	Al Italia	3429	13:50		Buffalo	Air Trans	549	10:42

Обратите внимание, что все рейсы, которые имеют один и тот же город назначения, отсортированы и по времени вылета. Таким образом, на этом наборе данных алгоритм сортировки продемонстрировал устойчивость. Неустойчивый алгоритм не обращает внимания на взаимоотношения между местоположениями элементов в исходном наборе (он может поддерживать относительное упорядочение, но это не является обязательным, так что относительное упорядочение может и не поддерживаться).

Критерии выбора алгоритма сортировки

Чтобы выбрать алгоритм сортировки для использования или реализации, рассмотрите приведенные в табл. 4.2 качественные критерии.

Таблица 4.2. Критерии выбора алгоритма сортировки

Критерий	Алгоритм сортировки
Только несколько элементов	Сортировка вставками
Элементы уже почти отсортированы	Сортировка вставками
Важна производительность в наихудшем случае	Пирамидальная сортировка
Важна производительность в среднем случае	Быстрая сортировка
Элементы с равномерным распределением	Блочная сортировка
Как можно меньший код	Сортировка вставками
Требуется устойчивость сортировки	Сортировка слиянием

Сортировки перестановкой

Ранние алгоритмы сортировки находили элементы коллекции, которые находились в неверных позициях, и перемещали их в корректное положение путем *перестановки* (или обмена) элементов A. **Сортировка выбором** и (печально известная) **пузырьковая сортировка** принадлежат к этому семейству сортировок. Но все эти алгоритмы превосходит **сортировка вставками**, которую мы сейчас и рассмотрим.

Сортировка вставками

Сортировка вставками (Insertion Sort) многократно использует вспомогательную функцию `insert`, чтобы обеспечить корректную сортировку $A[0, i]$; в конечном итоге i достигает крайнего справа элемента, сортируя тем самым все множество A целиком.

Сортировка вставками

Наилучший случай: O(n) Средний, наихудший случаи: O(n^2)

```
sort (A)
    for pos = 1 to n-1 do
        insert (A, pos, A[pos])
end

insert (A, pos, value)
    i = pos - 1
    while i >= 0 and A[i] > value do    ❶
        A[i+1] = A[i]
        i = i-1
    A[i+1] = value                      ❷
end
```

❶ Сдвигаем элементы, большие value, вправо.
❷ Вставляем value в корректное местоположение.

На рис. 4.3 показано, как работает сортировка вставками на неупорядоченном множестве A размера $n = 16$. 15 нижних строк на рисунке отображают состояние A после каждого вызова `insert`.

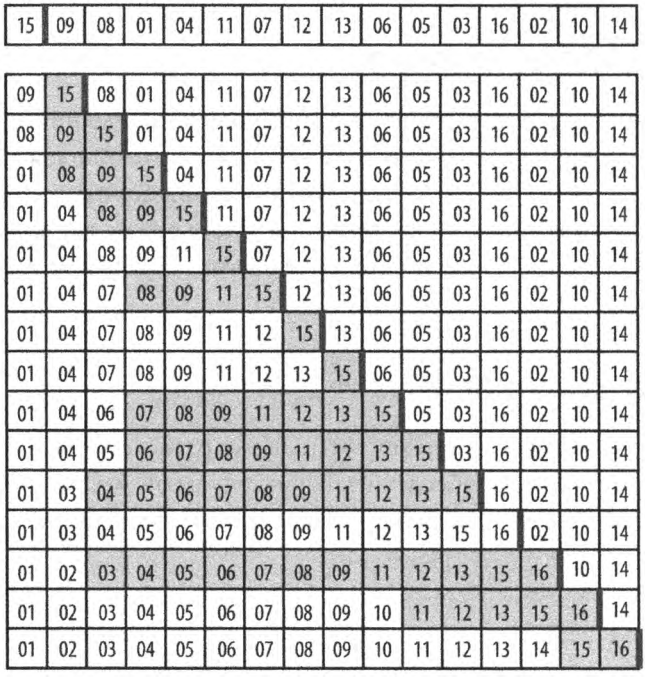


Рис. 4.3. Процесс сортировки вставками небольшого массива

Массив A сортируется без привлечения дополнительной памяти путем увеличения $pos = 1$ до $n - 1$ и вставки элемента $A[pos]$ в его корректную позицию в сортируемом диапазоне $A[0, pos]$, на рисунке ограниченном справа толстой вертикальной линией. Выделенные серым цветом элементы сдвигаются вправо, чтобы освободить место для вставляемого элемента; в целом для данного массива сортировка вставками выполнила 60 обменов соседних элементов (перемещений элементов на одну позицию).

Контекст применения алгоритма

Используйте сортировку вставками, когда у вас есть небольшое количество сортируемых элементов или когда элементы исходного множества уже "почти отсортированы". Критерий "достаточной малости" зависит от конкретной машины и языка программирования. В действительности может играть роль даже тип сравниваемых элементов.

Реализация алгоритма

Программа в примере 4.1 сортирует массив `ar` элементов, которые можно сравнить с помощью функции сравнения `cmp`, для случая хранения информации с использованием указателей.

Пример 4.1. Сортировка вставками при хранении с использованием указателей

```
void sortPointers(void** ar, int n,
                  int (*cmp)(const void*, const void*))
{
    int j;

    for (j = 1; j < n; j++)
    {
        int i = j - 1;
        void* value = ar[j];

        while (i >= 0 && cmp(ar[i], value) > 0)
        {
            ar[i + 1] = ar[i];
            i--;
        }

        ar[i + 1] = value;
    }
}
```

Если A представляет собой хранилище на основе значений, данное множество упаковывается в n строк элементов с фиксированным размером s байтов. Работа со значениями требует как соответствующей функции сравнения, так и средства для копирования значений из одного местоположения в другое. В примере 4.2 приведена программа на языке программирования C, которая использует для эффективного перемещения байтов смежных элементов массива A функцию `memmove`.

Пример 4.2. Сортировка вставками при хранении значений

```
void sortValues(void* base, int n, int s,
                int (*cmp)(const void*, const void*))
{
    int j;
    void* saved = malloc(s);

    for (j = 1; j < n; j++)
    {
        int i = j - 1;
        void* value = base + j * s;

        while (i >= 0 && cmp(base + i * s, value) > 0)
        {
            i--;
        }

        /* Если элемент находится на своем месте, перемещение
         * не требуется. В противном случае сохраняем вставляемое
         * значение и перемещаем промежуточные значения как один
         * БОЛЬШОЙ блок. Затем вставляем сохраненное значение в
         * корректную позицию. */
        if (++i == j) continue;

        memmove(saved, value, s);
        memmove(base + (i + 1)*s, base + i * s, s * (j - i));
        memmove(base + i * s, saved, s);
    }

    free(saved);
}
```

Оптимальная производительность достигается, когда массив уже отсортирован, а массив, отсортированный в обратном порядке, является наихудшим случаем для сортировки вставками. Если массив уже почти отсортирован, сортировка вставками эффективна в силу того, что требуется перестановка только малого количества элементов.

Сортировка вставками требует очень малой дополнительной памяти для работы — нужно зарезервировать место только для одного элемента. Большинство библиотек языков программирования при представлении на основе значений для выполнения более эффективного перемещения элементов используют функции перемещения блоков памяти.

Анализ алгоритма

В наилучшем случае каждый из n элементов находится на своем месте, и, таким образом, сортировка вставками занимает линейное время, или $O(n)$. Это может показаться слишком тривиальным утверждением, чтобы его упоминать (как часто вы

собираетесь сортировать набор уже отсортированных элементов?), но это свойство является важным, поскольку сортировка вставками — единственный алгоритм сортировки на основе сравнений, который имеет такое поведение в наилучшем случае.

Много реальных данных уже частично отсортированы, поэтому оптимизм и реализм могут иногда совпадать и делать сортировку вставками эффективным алгоритмом для решения конкретных задач. Эффективность сортировки вставками увеличивается при наличии одинаковых элементов, поскольку при этом требуется выполнять меньше обменов.

К сожалению, сортировка вставками оказывается не столь эффективной, когда все n элементов различны, а массив данных организован случайным образом (т.е. все перестановки данных равновероятны), поскольку в таком случае каждый элемент находится в массиве на расстоянии в среднем $n/3$ позиций относительно его правильного местоположения, которое он должен занимать в отсортированном массиве. Программа `numTranspositions.c` из репозитория кода эмпирически подтверждает это утверждение для малых n, имеющих значение до 12 (см. также [58]). В среднем и наихудшем случае каждый из n элементов следует перенести на линейное количество позиций, так что сортировка вставками выполняется за квадратичное время $O(n^2)$.

Сортировка вставками менее эффективно работает при хранении информации на основе значений — из-за объема памяти, которую необходимо перенести, чтобы освободить место для нового значения. В табл. 4.3 содержатся сравнения между прямой реализацией сортировки вставками на основе значений и реализацией из примера 4.2. Были проведены десять случайных испытаний сортировки n элементов, и самый лучший и самый худший результаты были отброшены. В таблице приведены средние значения по оставшимся восьми запускам. Обратите внимание, насколько улучшается эффективность при перемещении блоков памяти вместо обмена отдельных элементов. Тем не менее при удвоении размера массива время работы приблизительно учетверяется, подтверждая поведение $O(n^2)$ сортировки вставками. Применение перемещения блоков памяти не меняет квадратичный характер сортировки вставками.

Таблица 4.3. Время работы сортировки вставками: с перемещением блоков памяти и прямая реализация

n	Перемещение блоков памяти, с	Прямая реализация, с
1024	0,0039	0,0130
2048	0,0153	0,0516
4096	0,0612	0,2047
8192	0,2473	0,8160
16384	0,9913	3,2575
32768	3,9549	13,0650
65536	15,8722	52,2913
131072	68,4009	209,2943

Когда сортировка вставками работает с хранилищем на основе указателей, обмен элементами является более эффективным; компилятор может даже создать оптимизированный код, минимизирующий дорогостоящие обращения к памяти.

Сортировка выбором

Одна распространенная стратегия сортировки заключается в том, чтобы выбрать наибольшее значение из диапазона $A[0,n)$ и поменять его местами с крайним справа элементом $A[n-1]$. Впоследствии этот процесс повторяется поочередно для каждого меньшего диапазона $A[0,n-1)$ до тех пор, пока массив A не будет отсортирован. Мы рассматривали **сортировку выбором** (selection sort) в главе 3, "Строительные блоки алгоритмов", в качестве примера жадного подхода. В примере 4.3 содержится реализация этого алгоритма на языке программирования C.

Пример 4.3. Реализация на языке C сортировки выбором

```
static int selectMax(void** ar, int left, int right,
                     int (*cmp)(const void*, const void*))
{
    int maxPos = left;
    int i = left;

    while (++i <= right)
    {
        if (cmp(ar[i], ar[maxPos]) > 0)
        {
            maxPos = i;
        }
    }

    return maxPos;
}

void sortPointers(void** ar, int n,
                  int (*cmp)(const void*, const void*))
{
    /* Многократный выбор максимального значения в диапазоне ar[0,i]
       и его обмен с перестановкой в корректное местоположение */
    int i;

    for (i = n - 1; i >= 1; i--)
    {
        int maxPos = selectMax(ar, 0, i, cmp);

        if (maxPos != i)
        {
```

```
            void* tmp = ar[i];
            ar[i] = ar[maxPos];
            ar[maxPos] = tmp;
        }
    }
}
```

Сортировка выбором является самым медленным из всех алгоритмов сортировки, описанных в этой главе. Она требует квадратичного времени даже в лучшем случае (т.е. когда массив уже отсортирован). Эта сортировка многократно выполняет почти одну и ту же задачу, ничему не обучаясь от одной итерации к следующей. Выбор наибольшего элемента *max* в массиве *A* требует $n-1$ сравнений, а выбор второго по величине элемента *second* выполняется с помощью $n-2$ сравнений — не слишком большой прогресс! Многие из этих сравнений затрачены впустую, например, потому, что если некоторый элемент меньше элемента *second*, то он никак не может быть наибольшим элементом, а потому никак не влияет на вычисление *max*. В силу неэффективности сортировки выбором вместо того, чтобы представить о ней более подробную информацию, мы рассмотрим **пирамидальную сортировку**, которая показывает, как можно применить принцип, лежащий в основе сортировки выбором, гораздо более эффективно.

Пирамидальная сортировка

Чтобы найти наибольший элемент в неупорядоченном массиве *A* из n элементов, требуется как минимум $n-1$ сравнений, но, может, есть способ свести к минимуму количество непосредственно сравниваемых элементов? Например, на спортивных турнирах "наилучшую" среди n команд определяют без того, чтобы каждая команда сыграла со всеми $n-1$ соперниками. Одним из самых популярных баскетбольных событий в США является чемпионат, на котором, по сути, за титул чемпиона соревнуются 64 команды колледжей. Команда-чемпион играет с пятью командами для выхода в финал. Так что, чтобы стать чемпионом, команда должна выиграть шесть игр. И не случайно $6 = log(64)$. **Пирамидальная сортировка** (Heap Sort) показывает, как применить такое поведение для сортировки множества элементов.

Пирамидальная сортировка

Наилучший, средний, наихудший случаи: $O(n \cdot \log n)$

```
sort (A)
    buildHeap (A)
    for i = n-1 downto 1 do
        swap A[0] with A[i]
        heapify (A, 0, i)
end
```

```
buildHeap (A)
    for i = n/2-1 downto 0 do
        heapify (A, i, n)
end

# Рекурсивное обеспечение условия, что A[idx,max)
# является корректной пирамидой
heapify (A, idx, max)
    largest = idx                    ❶
    left = 2*idx + 1
    right = 2*idx + 2

    if left < max and A[left] > A[idx] then
        largest = left               ❷
    if right < max and A[right] > A[largest] then
        largest = right              ❸
    if largest ≠ idx then
        swap A[idx] and A[largest]
        heapify (A, largest, max)
end

❶ Предполагается, что родитель A[idx] не меньше любого из потомков.
❸ Левый потомок больше родителя.
❷ Правый потомок больше родителя или левого "брата".
```

На рис. 4.4 показана работа `buildHeap` для массива из шести значений.

Пирамида представляет собой бинарное дерево, структура которого обеспечивает выполнение двух свойств.

Свойство формы

Левый узел на глубине $k>0$ может существовать, только если существуют все $2k-1$ узлов на глубине $k-1$. Кроме того, узлы на частично заполненных уровнях должны добавляться "слева направо". Корневой узел находится на глубине 0.

Свойство пирамиды

Каждый узел дерева содержит значение, не меньшее значения любого из его дочерних узлов, если таковые имеются.

Пример пирамиды на рис. 4.5, *а* удовлетворяет этим свойствам. Корень бинарного дерева содержит наибольший элемент этого дерева; однако наименьший элемент может оказаться в любом листовом узле. Хотя пирамида гарантирует только то, что узел больше любого из его потомков, пирамидальная сортировка показывает, как использовать преимущества свойства формы для эффективной сортировки массива элементов.

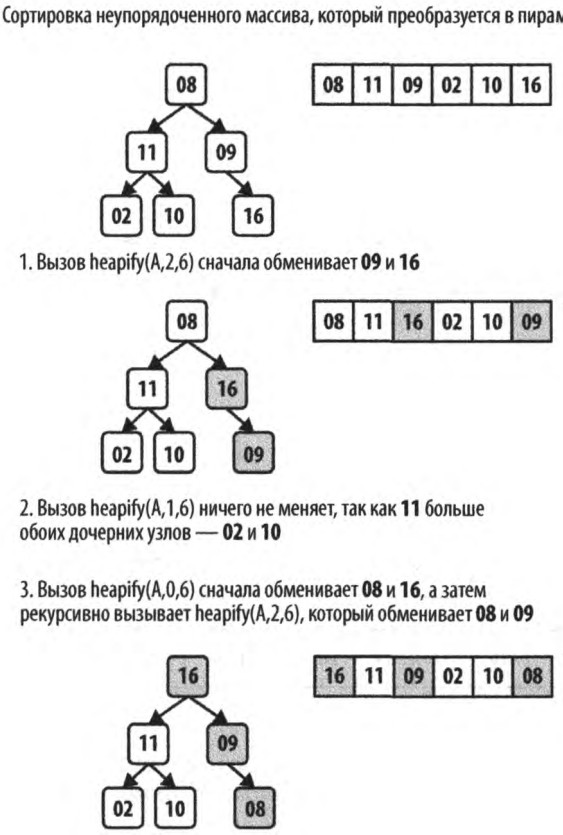


Рис. 4.4. Пример пирамидальной сортировки

С учетом строгой структуры, введенной *свойством формы*, пирамида может храниться в массиве A без потери какой-либо структурной информации. На рис. 4.5, *б* показаны целочисленные метки, назначенные каждому узлу пирамиды. Корень имеет метку 0. Для узла с меткой i его левый потомок (если таковой существует) имеет метку $2 \cdot i + 1$; его правый потомок (если он существует) имеет метку $2 \cdot i + 2$. Аналогично для не корневого узла с меткой i его родительский узел имеет метку $\lfloor (i-1)/2 \rfloor$. С помощью такой схемы можно хранить пирамиды в массиве, сохраняя значение элемента узла в позиции массива, которая определяется меткой узла. Показанный на рис. 4.5, *в* массив представляет собой пирамиду, показанную на рис. 4.5, *а*.

Пирамидальная сортировка сортирует массив A, сначала преобразуя его на месте, без привлечения дополнительной памяти, в пирамиду с помощью процедуры `buildHeap`, которая выполняет неоднократные вызовы процедуры `heapify`. Вызов `heapify(A,i,n)` обновляет массив A, гарантируя при этом, что структура дерева с корнем в $A[i]$ представляет собой корректную пирамиду. На рис. 4.6 показаны детали вызовов `heapify`, которые превращают неупорядоченный массив в пирамиду.

Применение `buildHeap` к уже отсортированному массиву показано на рис. 4.6. Каждая пронумерованная строка на этом рисунке показывает результат выполнения `heapify` над исходным массивом от средней точки $\lfloor n/2 \rfloor - 1$ до крайнего слева индекса 0.

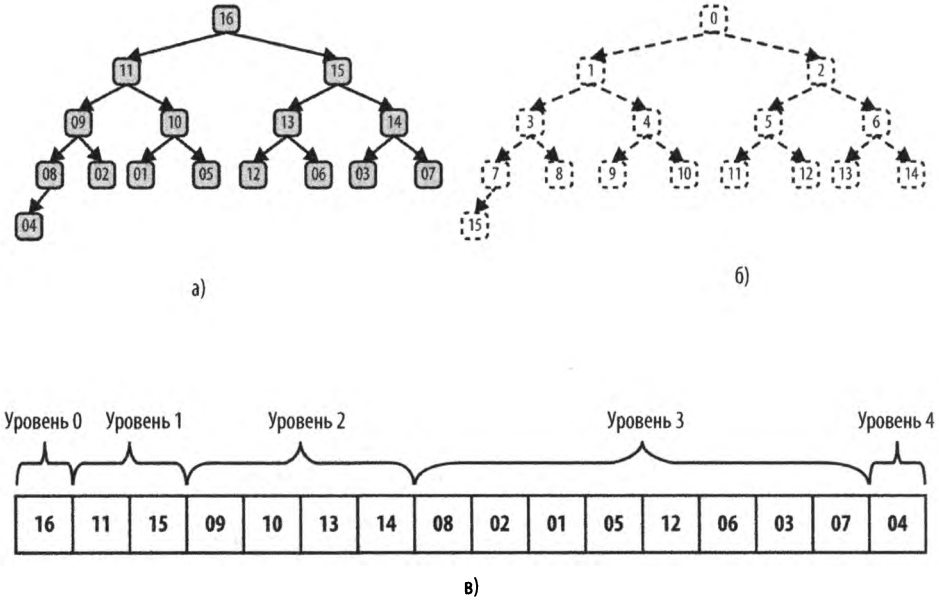


Рис. 4.5. Пример пирамиды из 16 различных элементов (а); метки этих элементов (б); пирамида, хранящаяся в массиве (в)

Как вы можете видеть, большие числа в конце концов "поднимаются" в получающейся в результате пирамиде (это означает, что они меняются местами в A с меньшими элементами слева). Выделенные серым цветом квадраты на рис. 4.6 указывают пары элементов, которые поменялись местами при выполнении `heapify` — в общей сложности их 13, — что гораздо меньше, чем общее число элементов, которые поменялись местами при сортировке вставками, показанной на рис. 4.3.

Пирамидальная сортировка обрабатывает массив A размера n, рассматривая его как два отдельных подмассива, $A[0,m)$ и $A[m,n)$, которые представляют собой пирамиду размера m и отсортированный подмассив из $n-m$ элементов соответственно. При итерациях i от $n-1$ до 1 пирамидальная сортировка увеличивает отсортированный подмассив $A[i,n)$ вниз путем обмена наибольшего элемента в пирамиде (в позиции $A[0]$) с $A[i]$. Затем с помощью вызова `heapify` алгоритм перестраивает $A[0,i)$ таким образом, что он представляет собой корректную пирамиду. В результате непустой подмассив $A[i,n)$ оказывается отсортированным, поскольку наибольший элемент пирамиды, представленной в $A[0,i)$, будет гарантированно меньше или равен любому элементу в отсортированном подмассиве $A[i,n)$.

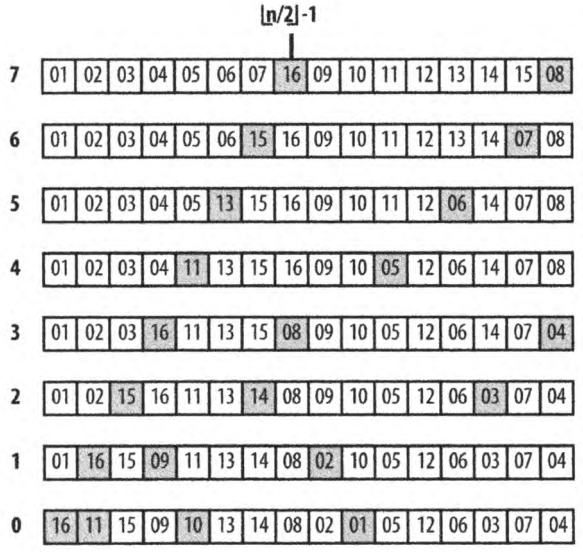


Рис. 4.6. Применение `buildHeap` к изначально отсортированному массиву

Контекст применения алгоритма

Пирамидальная сортировка не является устойчивой. Она позволяет избежать многих неприятных (почти неловких) ситуаций, которые приводят к плохой производительности быстрой сортировки. Тем не менее в среднем случае быстрая сортировка превосходит пирамидальную.

Реализация алгоритма

Пример реализации алгоритма на языке программирования C приведен в примере 4.4.

Пример 4.4. Реализация пирамидальной сортировки на языке программирования C

```
static void heapify(void** ar, int (*cmp)(const void*, const void*),
                    int idx, int max)
{
    int left = 2 * idx + 1;
    int right = 2 * idx + 2;
    int largest;

    /* Поиск максимального элемента
       среди A[idx], A[left] и A[right]. */
    if (left < max && cmp(ar[left], ar[idx]) > 0)
    {
        largest = left;
```

```
    }
    else
    {
        largest = idx;
    }

    if (right < max && cmp(ar[right], ar[largest]) > 0)
    {
        largest = right;
    }

    /* Если наибольший элемент не является родительским,
     * выполняем обмен и переход далее. */
    if (largest != idx)
    {
        void* tmp;
        tmp = ar[idx];
        ar[idx] = ar[largest];
        ar[largest] = tmp;
        heapify(ar, cmp, largest, max);
    }
}

static void buildHeap(void** ar,
                      int (*cmp)(const void*, const void*), int n)
{
    int i;

    for (i = n / 2 - 1; i >= 0; i--)
    {
        heapify(ar, cmp, i, n);
    }
}

void sortPointers(void** ar, int n,
                  int (*cmp)(const void*, const void*))
{
    int i;
    buildHeap(ar, cmp, n);

    for (i = n - 1; i >= 1; i--)
    {
        void* tmp;
        tmp = ar[0];
        ar[0] = ar[i];
        ar[i] = tmp;
        heapify(ar, cmp, 0, i);
    }
}
```

Анализ алгоритма

Центральной операцией пирамидальной сортировки является `heapify`. В `buildHeap` она вызывается $\lfloor n/2 \rfloor - 1$ раз, а за все время сортировки — $n-1$ раз, в общей сложности $\lfloor 3n/2 \rfloor - 2$ раз. Из-за свойства формы глубина пирамиды всегда равна $\lfloor \log n \rfloor$, где n — количество элементов в пирамиде. Как можно видеть, это рекурсивная операция, в которой выполняется не более чем $\log n$ рекурсивных вызовов до момента полного исправления пирамиды или достижения ее конца. Однако `heapify` может остановиться преждевременно, как только пирамида будет исправлена. Как оказалось, в общей сложности необходимо не более чем $2n$ сравнений [20], так что поведение `buildHeap` описывается как $O(n)$.

Вариации алгоритма

В репозитории имеется нерекурсивная реализация пирамиды, а в табл. 4.4 представлено сравнение времени работы 1000 рандомизированных испытаний обеих реализаций (с отбрасыванием лучших и худших запусков каждой из них).

Таблица 4.4. Сравнение пирамидальной сортировки и ее нерекурсивной версии

n	Время нерекурсивной сортировки, с	Время рекурсивной сортировки, с
16384	0,0048	0,0064
32768	0,0113	0,0147
65536	0,0263	0,0336
131072	0,0762	0,0893
262144	0,2586	0,2824
524288	0,7251	0,7736
1048576	1,8603	1,9582
2097152	4,5660	4,7426

Сначала имеется заметное улучшение, связанное с устранением рекурсии в пирамидальной сортировке, однако с ростом n это различие уменьшается.

Сортировка, основанная на разбиении

Стратегия "разделяй и властвуй" решает задачу, разделяя ее на две независимые подзадачи, каждая из которых размером около половины размера исходной задачи. Эту стратегию можно применить к сортировке следующим образом: найти *медианный* элемент в коллекции A и поменять его со средним элементом A. Теперь будем менять местами элементы из левой половины, большие, чем $A[mid]$, с элементами в правой половине, которые меньше или равны $A[mid]$. Это действие разделяет исходный массив на два отдельных подмассива, которые можно отсортировать рекурсивно и получить отсортированную исходную коллекцию A.

Реализация этого подхода является сложной задачей, потому что не очевидно, как вычислить средний элемент коллекции без предварительной сортировки. Но оказывается, что для разделения *A* на два подмассива можно использовать любой элемент *A*. При "мудром" выборе этого элемента оба подмассива будут более или менее одинакового размера, так что можно получить эффективную реализацию данного алгоритма.

Предположим, что имеется функция `partition(A,left,right,pivotIndex)`, которая использует специальное *опорное* значение *pivot* из *A* (которое представляет собой *A*[*pivotIndex*]) для того, чтобы модифицировать *A*, и возвращает местоположение *p* в *A* такое, что:

- $A[p] = pivot$;
- все элементы из $A[left, p)$ не превышают опорного значения;
- все элементы из $A[p+1, right]$ больше опорного значения.

Если вам повезет, то по завершении работы функции *partition* размеры этих двух подмассивов будут более или менее близки к половине размера исходной коллекции. В примере 4.5 показана реализация этой функции на языке программирования C.

Пример 4.5. Реализация функции `partition` для массива `ar[left,right]` и указанного опорного элемента

```
/**
 * Данная функция за линейное время группирует подмассив
 * ar[left,right] вокруг опорного элемента pivot = ar[pivotIndex]
 * путем сохранения pivot в корректной позиции store (которая
 * возвращается данной функцией) и обеспечения выполнения условий:
 * все ar[left,store) < = pivot и все ar[store+1,rigt] > pivot.
*/
int partition(void** ar, int (*cmp)(const void*, const void*),
              int left, int right, int pivotIndex)
{
    int idx, store;
    void* pivot = ar[pivotIndex];

    /* Перемещаем pivot в конец массива */
    void* tmp = ar[right];
    ar[right] = ar[pivotIndex];
    ar[pivotIndex] = tmp;

    /* Все значения, не превышающие pivot, перемещаются в начало
     * массива, и pivot вставляется сразу после них. */
    store = left;
    for (idx = left; idx < right; idx++)
    {
```

```
        if (cmp(ar[idx], pivot) <= 0)
        {
            tmp = ar[idx];
            ar[idx] = ar[store];
            ar[store] = tmp;
            store++;
        }
    }

    tmp = ar[right];
    ar[right] = ar[store];
    ar[store] = tmp;
    return store;
}
```

Алгоритм **быстрой сортировки** (Quicksort), разработанный Ч.Э.Р. Хоаром (C.A.R. Hoare) в 1960 году, выбирает элемент коллекции (иногда случайно, иногда крайний слева, иногда средний) для разбиения массива на два подмассива. Таким образом, быстрая сортировка состоит из двух этапов. Сначала массив разбивается на два, затем рекурсивно сортируется каждый из подмассивов.

Быстрая сортировка

Наилучший и средний случаи: (n·log n), наихудший случай: $O(n^2)$

```
sort (A)
    quicksort (A, 0, n-1)
end

quicksort (A, left, right)
    if left < right then
    pi = partition (A, left, right)
        quicksort (A, left, pi-1)
        quicksort (A, pi+1, right)
end
```

Этот псевдокод намеренно не определяет стратегию выбора индекса опорного элемента. В коде мы предполагаем, что существует некоторая функция `selectPivotIndex`, которая выбирает соответствующий индекс. Здесь мы не рассматриваем сложный математический инструментарий, необходимый для доказательства того факта, что быстрая сортировка имеет в среднем случае поведение $O(n \cdot log n)$; дополнительные сведения по этой теме доступны в [20].

На рис. 4.7 показан пример работы быстрой сортировки. Каждый черный квадрат представляет собой выбранный опорный элемент. Первый выбранный опорный элемент — 2; это оказался плохой выбор, так как он приводит к двум подмассивам

с размерами 1 и 14. Во время следующего рекурсивного вызова сортировки для правого подмассива в качестве опорного элемента выбран элемент 12 (см. четвертую строку), который производит два подмассива размером 9 и 4 соответственно. Уже сейчас вы можете увидеть преимущества разбиения, поскольку последние четыре элемента в массиве являются наибольшими четырьмя элементами исходного массива, хотя они по-прежнему не упорядочены. Из-за случайного характера выбора опорного элемента возможны различные варианты поведения. При другом выполнении того же алгоритма, показанном на рис. 4.8, уже первый выбранный опорный элемент красиво делит задачу на две более или менее одинаковые по размеру подзадачи.

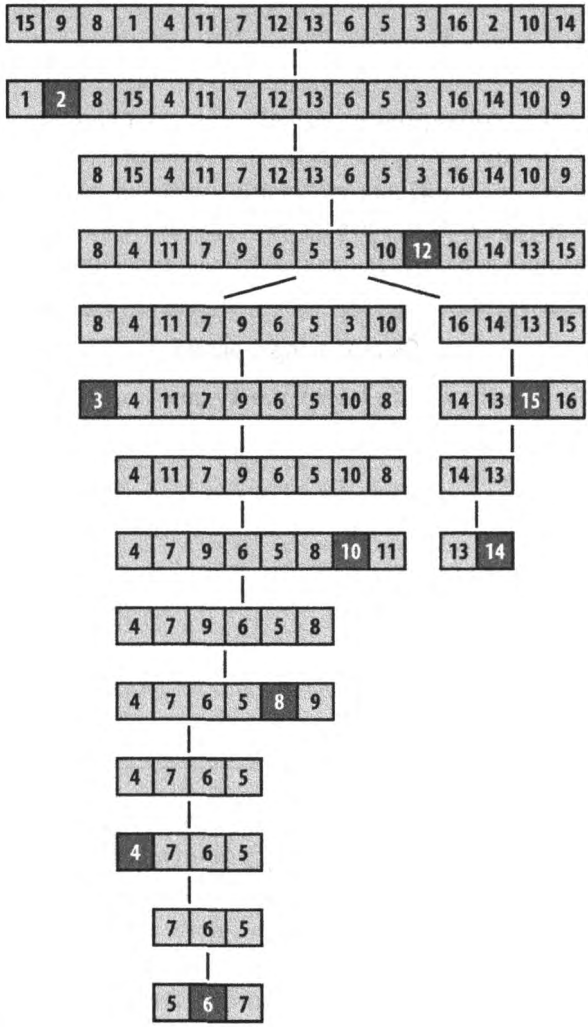


Рис. 4.7. Пример работы быстрой сортировки

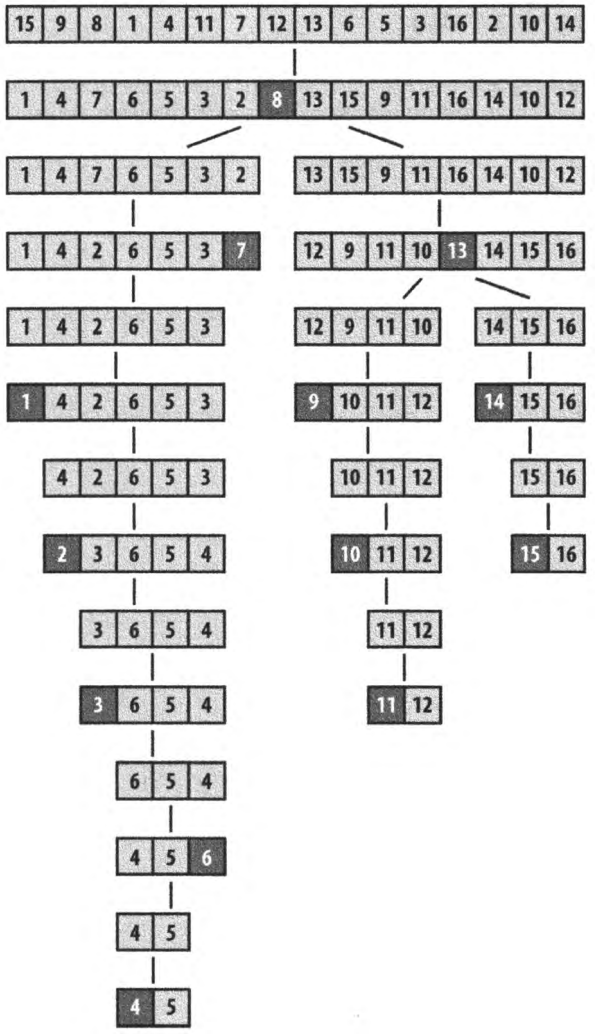


Рис. 4.8. Пример другого поведения быстрой сортировки

Контекст применения алгоритма

Быстрая сортировка демонстрирует наихудшее квадратичное поведение, если разбиение на каждом шаге рекурсии делит коллекцию из n элементов на "пустое" и "большое" множества, где одно из этих множеств не имеет элементов, а другое содержит $n-1$ элементов. (Обратите внимание, что опорный элемент является крайним из n элементов, поэтому ни один элемент не теряется.)

Реализация алгоритма

Показанная в примере 4.6 реализация быстрой сортировки включает стандартную оптимизацию — сортировку вставками, когда размер подмассива сокращается до предопределенного минимального размера.

Пример 4.6. Реализация быстрой сортировки на языке программирования C

```
/**
 * Сортирует массив ar[left,right] с использованием метода
 * быстрой сортировки. Функция сравнения cmp необходима для
 * корректного сравнения элементов.
*/
void do_qsort(void** ar, int (*cmp)(const void*, const void*),
              int left, int right)
{
    int pivotIndex;

    if (right <= left)
    {
        return;
    }

    /* Разбиение */
    pivotIndex = selectPivotIndex(ar, left, right);
    pivotIndex = partition(ar, cmp, left, right, pivotIndex);

    if (pivotIndex - 1 - left <= minSize)
    {
        insertion(ar, cmp, left, pivotIndex - 1);
    }
    else
    {
        do_qsort(ar, cmp, left, pivotIndex - 1);
    }

    if (right - pivotIndex - 1 <= minSize)
    {
        insertion(ar, cmp, pivotIndex + 1, right);
    }
    else
    {
        do_qsort(ar, cmp, pivotIndex + 1, right);
    }
}

/** Вызов быстрой сортировки */
void sortPointers(void** vals, int total_elems,
```

```
                int (*cmp)(const void*, const void*))
{
    do_qsort(vals, cmp, 0, total_elems - 1);
}
```

Внешний метод `selectPivotIndex(ar,left,right)` выбирает значение опорного элемента, относительно которого выполняется разбиение массива.

Анализ алгоритма

Удивительно, но при использовании в качестве опорного случайного элемента быстрая сортировка демонстрирует производительность для среднего случая, которая обычно превосходит производительность всех прочих алгоритмов сортировки. Кроме того, имеется целый ряд исследований по усовершенствованию и оптимизации быстрой сортировки, которые позволяют достичь эффективности, превышающей эффективность любого алгоритма сортировки.

В идеальном случае `partition` делит исходный массив пополам, и быстрая сортировка демонстрирует производительность $O(n \cdot log n)$. На практике быстрая сортировка вполне эффективна при выборе опорного элемента случайным образом.

В наихудшем случае в качестве опорного элемента выбирается наибольший или наименьший элемент массива. Когда это происходит, быстрая сортировка делает проход по всем элементам массива (за линейное время) для того, чтобы отсортировать единственный элемент в массиве. Если этот процесс повторится $n-1$ раз, это приведет к наихудшему поведению данного метода сортировки — $O(n^2)$.

Вариации алгоритма

Быстрая сортировка выбирается в качестве основного метода сортировки в большинстве систем. Так, в Unix-системах имеется встроенная библиотечная функция `qsort`. Зачастую операционная система использует оптимизированную версию алгоритма быстрой сортировки. Основными цитируемыми источниками различных оптимизаций являются работы Седжвика (Sedgewick) [56] и Бентли (Bentley) и Макилроя (McIlroy) [9]. Очень поучительно, что некоторые версии операционной системы Linux реализуют функцию `qsort` с использованием пирамидальной сортировки.

Среди различных оптимизаций быстрой сортировки можно отметить следующие.

- Создание стека для хранения подзадач и устранения тем самым рекурсии.
- Выбор опорного элемента на основе стратегии медианы трех элементов.
- Установка минимального размера разбиения (зависящего от реализации и архитектуры машины), ниже которого вместо быстрой сортировки используется сортировка вставками; в JDK 1.8 таким пороговым значением является 7.

- При обработке двух подзадач общий размер стека рекурсии минимизируется путем решения меньшей задачи в первую очередь.

Однако никакая оптимизация не позволяет устранить наихудшее поведение быстрой сортировки — $O(n^2)$. Единственный способ обеспечить производительность $O(n \cdot log\ n)$ в наихудшем случае — это использовать функцию разбиения, которая может гарантировать, что она обнаруживает "разумное приближение" к фактической медиане набора данных. В работе [15] описывается такой алгоритм (BFPRT — по первым буквам фамилий авторов Blum-Floyd-Pratt-Rivest-Tarjan) с доказуемо линейным временем, но он имеет лишь теоретическое значение. Реализация алгоритма BFPRT имеется в репозитории кода.

Выбор опорного элемента

Выбор опорного элемента из подмассива $A[left, left+n)$ должен быть быстрой операцией; он не должен требовать проверки всех n элементов подмассива. Могут использоваться различные варианты; вот некоторые из них:

- выбор первого или последнего элемента: $A[left]$ или $A[left+n-1]$;
- выбор случайного элемента из $A[left, left+n-1]$;
- выбор медианы из k элементов: среднее значение из k элементов, взятых из $A[left, left+n-1]$.

Часто выбирается медиана трех элементов; Седжвик (Sedgewick) сообщает, что этот подход приводит к улучшению времени работы на 5%; но заметим, что даже для этой альтернативы имеются некоторые размещения данных, которые приводят к плохой производительности [41]. Используется также выбор медианы из пяти элементов. Выполнение большего количества вычислений для определения правильного опорного элемента из-за дополнительных издержек редко дает положительные результаты.

Выполнение разбиения

В методе `partition`, показанном в примере 4.5, элементы, меньшие или равные выбранному опорному элементу, вставляются в начало подмассива. Такой подход может привести к перекосу размеров подмассивов для шага рекурсии, если для выбранного опорного элемента имеется много таких же элементов в массиве. Одним из средств снижения дисбаланса является чередующееся размещение элементов, равных опорному, в первом и втором подмассивах.

Обработка подмассивов

Быстрая сортировка выполняет два рекурсивных вызова для меньших подмассивов. При работе одного из них в стек выполнения вносится запись активации другого. Если сначала обрабатывается больший подмассив, то возможно одновременное размещение в стеке линейного количества записей активации (хотя современные

компиляторы могут устранить эти наблюдаемые накладные расходы). Чтобы свести к минимуму возможную глубину стека, можно сначала обрабатывать подмассивы меньшего размера. Если глубина рекурсии является критичной для вашего приложения, то, возможно, быстрая сортировка вам не подойдет.

Применение сортировки вставками для малых подмассивов

На небольших массивах сортировка вставками быстрее, чем быстрая сортировка, но даже при использовании больших массивов быстрая сортировка в конечном итоге разделяет задачу на большое количество мелких. Один из часто используемых методов улучшения производительности рекурсивной быстрой сортировки заключается в вызове быстрой сортировки только для больших подмассивов, а для маленьких используется сортировка вставками, как показано в примере 4.6.

Седжвик [56] свидетельствует о том, что сочетание медианы трех элементов и сортировки вставками для небольших подмассивов увеличивает скорость на 20–25% по сравнению с чистой быстрой сортировкой.

Интроспективная сортировка

Переключение на сортировку вставками для небольших подмассивов является локальным решением, которое выполняется на основании размера подмассива. Мюссер (Musser) [41] представил вариант быстрой сортировки под названием **интроспективная сортировка** (IntroSort), в котором для обеспечения эффективной работы контролируется глубина рекурсии быстрой сортировки. Если глубина рекурсии превышает *log n* уровней, IntroSort переключается на пирамидальную сортировку. Реализация SGI стандартной библиотеки шаблонов C++ (`http://www.sgi.com/tech/stl/sort.html`) в качестве сортировки по умолчанию использует интроспективную сортировку.

Сортировка без сравнений

В конце этой главы мы покажем, что никакой алгоритм сортировки на основе сравнений не может сортировать *n* элементов быстрее, чем с производительностью $O(n \cdot log\ n)$. Как это ни удивительно, имеются потенциально более быстрые способы сортировки элементов, если об этих элементах что-то известно заранее. Например, если у вас имеется быстрая хеш-функция, которая равномерно распределяет коллекцию элементов по различным упорядоченным блокам данных, можно использовать описанный далее блочный алгоритм сортировки с линейной $O(n)$ производительностью.

Блочная сортировка

Для данного набора из *n* элементов **блочная сортировка** (Bucket Sort) создает множество из *n* упорядоченных блоков, в которые помещаются элементы входного

набора. Блочная сортировка сокращает стоимость обработки за счет этой дополнительной памяти. Если хеш-функция `hash(A[i])` может равномерно разбить входной набор из n элементов на эти n блоков, то блочная сортировка имеет производительность $O(n)$ в худшем случае. Используйте блочную сортировку при выполнении следующих двух условий.

Равномерное распределение

Входные данные должны быть равномерно распределены в заданном диапазоне. На основе этого распределения создаются n блоков для равномерного разбиения входного диапазона.

Упорядочивающая хеш-функция

Блоки упорядочены. Если $i<j$, элементы, вставленные в блок b_i, лексикографически меньше элементов в блоке b_j.

Блочная сортировка

Наилучший, средний и наихудший случаи: $O(n)$

```
sort (A)
    Создаем n блоков B
    for i = 0 to n-1 do                    ❶
        k = hash(A[i])
        Добавляем A[i] в k-й блок B[k]
    extract(B,A)
end

extract (B, A)
    idx = 0
    for i = 0 to n-1 do
        insertionSort(B[i])                ❷
        Для каждого элемента e в B[i]
            A[idx++] = e
end
```

❶ Создание списка блоков и хеширование всех элементов в соответствующие блоки.
Обработка всех блоков для возврата значений в A в отсортированном порядке.

❷ Если в блоке больше одного элемента, блок сначала сортируется.
Копирование элементов назад в A в отсортированном порядке.

Блочная сортировка, например, не подходит для сортировки произвольных строк, потому что обычно невозможно разработать хеш-функцию с требуемыми характеристиками. Однако она может использоваться для сортировки множества равномерно распределенных чисел с плавающей точкой из диапазона [0,1).

После того как все сортируемые элементы будут распределены по блокам, блочная сортировка извлекает значения слева направо с использованием сортировки вставками для содержимого каждого блока. Это упорядочивает элементы каждого блока перед тем, как его значения извлекаются слева направо, чтобы заполнить исходный массив. Пример выполнения блочной сортировки показан на рис. 4.9.

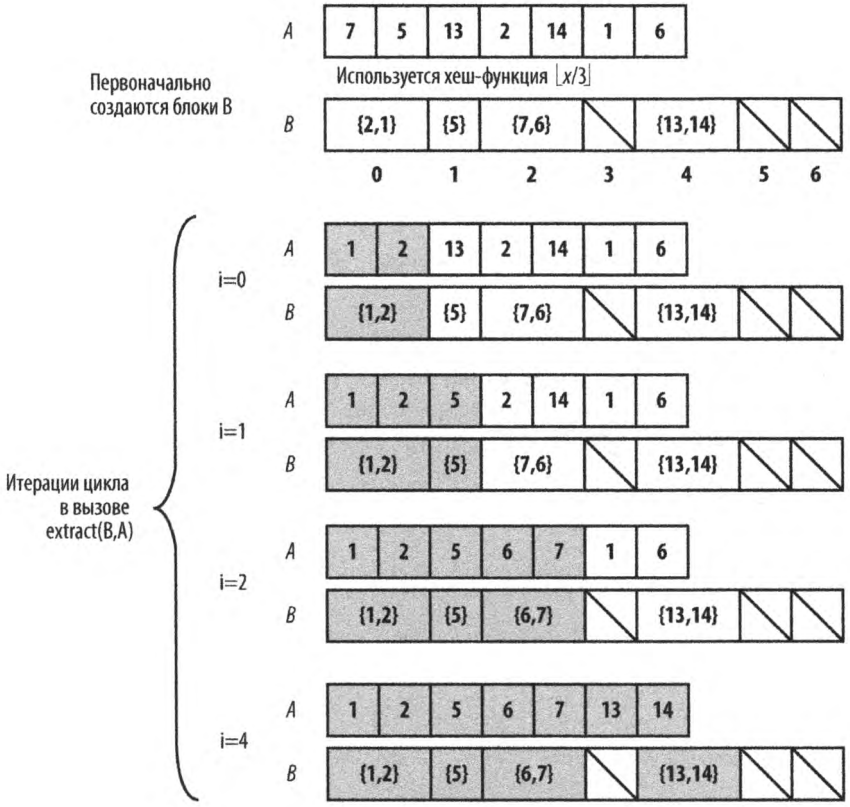


Рис. 4.9. Небольшой демонстрационный пример блочной сортировки

Реализация алгоритма

В реализации блочной сортировки на языке программирования C, показанной в примере 4.7, каждый блок хранит связанный список элементов, хешированных в этот блок. Функции `numBuckets` и `hash` предоставляются извне с учетом особенностей входного множества.

Пример 4.7. Реализация блочной сортировки на языке программирования C

```
extern int hash(void* elt);
extern int numBuckets(int numElements);

/* Связанный список элементов в блоке. */
typedef struct entry
{
    void* element;
    struct entry* next;
} ENTRY;

/* Поддерживаем количество записей в блоке
   и указатель на первую запись */
typedef struct
{
    int size;
    ENTRY* head;
} BUCKET;

/* Память для блоков и количество выделенных блоков */
static BUCKET* buckets = 0;
static int num = 0;

/** Удаление элементов по одному и перезапись в ar */
void extract(BUCKET* buckets, int (*cmp)(const void*, const void*),
             void** ar, int n)
{
    int i, low;
    int idx = 0;

    for (i = 0; i < num; i++)
    {
        ENTRY* ptr, *tmp;

        if (buckets[i].size == 0) continue; /* Пустой блок */

        ptr = buckets[i].head;

        if (buckets[i].size == 1)
        {
            ar[idx++] = ptr->element;
            free(ptr);
            buckets[i].size = 0;
            continue;
        }

        /* Сортировка вставками, в которой элементы выбираются
         * из связанного списка и вставляются в массив. Связанные
         * списки освобождаются. */
```

```
        low = idx;
        ar[idx++] = ptr->element;
        tmp = ptr;
        ptr = ptr->next;
        free(tmp);

        while (ptr != NULL)
        {
            int i = idx - 1;

            while (i >= low && cmp(ar[i], ptr->element) > 0)
            {
                ar[i + 1] = ar[i];
                i--;
            }

            ar[i + 1] = ptr->element;
            tmp = ptr;
            ptr = ptr->next;
            free(tmp);
            idx++;
        }

        buckets[i].size = 0;
    }
}

void sortPointers(void** ar, int n,
                  int (*cmp)(const void*, const void*))
{
    int i;
    num = numBuckets(n);
    buckets = (BUCKET*) calloc(num, sizeof(BUCKET));

    for (i = 0; i < n; i++)
    {
        int k = hash(ar[i]);
        /** Вставка каждого элемента и увеличение счетчика */
        ENTRY* e = (ENTRY*) calloc(1, sizeof(ENTRY));
        e->element = ar[i];

        if (buckets[k].head == NULL)
        {
            buckets[k].head = e;
        }
        else
        {
```

```
            e->next = buckets[k].head;
            buckets[k].head = e;
        }

        buckets[k].size++;
    }

    /* Сортировка, чтение и перезапись ar. */
    extract(buckets, cmp, ar, n);
    free(buckets);
}
```

В примере 4.8 содержится образец реализации хеш-функции и функции numBuckets для равномерно распределенных чисел из диапазона [0,1).

Пример 4.8. Функции hash *и* numBuckets *для диапазона [0, 1)*

```
static int num;
/** Количество блоков равно количеству элементов. */
int numBuckets(int numElements)
{
    num = numElements;
    return numElements;
}

/**
* Хеш-функция, идентифицирующая номер блока по элементу.
* Диапазон чисел – [0,1), так что мы распределяем числа по
* блокам размером 1/num. */
int hash(double* d)
{
    int bucket = num * (*d);
    return bucket;
}
```

Блоки могут также храниться с использованием фиксированных массивов, которые перераспределяются при их заполнении, но реализация на основе связанного списка оказывается примерно на 30–40% быстрее.

Анализ алгоритма

Функция SortPointers из примера 4.7 сортирует каждый элемент входных данных в соответствующий блок на основе предоставленной хеш-функции. Время работы алгоритма — $O(n)$. Благодаря тщательному дизайну хеш-функции мы знаем, что при $i<j$ все элементы в блоке b_i меньше, чем элементы в блоке b_j.

Поскольку значения извлекаются из блоков и записываются обратно во входной массив, когда блок содержит более одного элемента, используется сортировка

вставками. Чтобы блочная сортировка демонстрировала поведение $O(n)$, мы должны гарантировать, что общее время сортировки каждого из этих блоков также представляет собой $O(n)$. Определим число элементов в блоке b_i как n_i. Мы можем рассматривать n_i как случайную величину (используя математическую статистику). Теперь рассмотрим ожидаемое значение $E[n_i]$ для каждого блока b_i. Каждый элемент во входном наборе имеет вероятность попасть в заданный блок, равную $p = 1/n$, потому что каждый из этих элементов равномерно выбирается из диапазона $[0, 1)$. Таким образом,

$$E[n_i] = n \cdot p = n \cdot (1/n) = 1,$$

а дисперсия

$$Var[n_i] = n \cdot p \cdot (1-p) = (1-1/n).$$

Рассмотрение дисперсии существенно, поскольку одни блоки будут пустыми, а другие будут иметь более одного элемента. Мы должны быть уверены, что ни в каком блоке не будет слишком большого количества элементов. Прибегнем еще раз к математической статистике, которая предоставляет следующее уравнение для случайных величин:

$$E[n_i^2] = Var[n_i] + E^2[n_i].$$

Из этого уравнения можно вычислить ожидаемое значение n_i^2. Это очень важное значение, поскольку это фактор, определяющий стоимость сортировки вставками, которая в худшем случае выполняется за время $O(n^2)$. Мы получаем значение $E[n_i^2] = (1-1/n)+1 = (2-1/n)$, которое показывает, что $E[n_i^2]$ можно считать константой. Это означает, что, когда мы просуммируем затраты на выполнение сортировки вставками во всех n блоках, ожидаемая производительности останется равной $O(n)$.

Вариации алгоритма

Вместо n блоков **сортировка хешированием** (Hash Sort) создает достаточно большое количество блоков k, на которые подразделяются элементы; с ростом k производительность сортировки хешированием растет. Ключевым моментом сортировки хешированием является функция хеширования `hash(e)`, которая возвращает целое число для каждого элемента e, такое, что $\texttt{hash}(a_i) \leq \texttt{hash}(a_j)$, если a_i лексикографически меньше, чем a_j.

Функция хеширования `hash(e)`, определенная в примере 4.9, работает с элементами, содержащими только строчные буквы. Она преобразует первые три символа строки (используя систему счисления по основанию 26) в целое число; так, для строки *abcdefgh* ее первые три символа (*abc*) извлекаются и преобразуется в значение $0 \cdot 676 + 1 \cdot 26 + 2 = 28$. Таким образом, эта строка вставляется в блок с меткой 28.

Пример 4.9. Функции `hash` и `numBuckets` для сортировки хешированием

```
/** Количество используемых блоков. */
int numBuckets(int numElements)
{
    return 26 * 26 * 26;
}

/**
* Хеш-функция, определяющая номер блока для элемента.
*/
int hash(void* elt)
{
    return (((char*)elt)[0] - 'a') * 676 +
           (((char*)elt)[1] - 'a') * 26 +
           (((char*)elt)[2] - 'a');
}
```

Производительность сортировки хешированием для различных количеств блоков и размеров входных данных показана в табл. 4.5. Для сравнения мы также представили данные быстрой сортировки с использованием для выбора опорного элемента метода медианы трех элементов.

Таблица 4.5. Производительность сортировки хешированием с различным количеством блоков по сравнению с быстрой сортировкой (время приведено в секундах)

n	26 блоков	676 блоков	17 576 блоков	Быстрая сортировка
16	0,000005	0,000010	0,000120	0,000004
32	0,000006	0,000012	0,000146	0,000005
64	0,000011	0,000016	0,000181	0,000009
128	0,000017	0,000022	0,000228	0,000016
256	0,000033	0,000034	0,000249	0,000033
512	0,000074	0,000061	0,000278	0,000070
1024	0,000183	0,000113	0,000332	0,000156
2048	0,000521	0,000228	0,000424	0,000339
4096	0,001600	0,000478	0,000646	0,000740
8192	0,005800	0,001100	0,001100	0,001600
16384	0,022400	0,002600	0,002000	0,003500
32768	0,094400	0,006900	0,004000	0,007600
65536	0,411300	0,022600	0,010800	0,016800
131072	1,765400	0,087100	0,036000	0,042200

Обратите внимание, что начиная с 17 576 блоков сортировка хешированием превосходит быструю сортировку для количества элементов $n > 8192$ (и эта тенденция продолжается с увеличением n). Однако для 676 блоков по достижении количества

элементов $n > 32768$ (в среднем 48 элементов на блок) хеш-сортировка начинает неизбежно замедляться с увеличением стоимости выполнения сортировки вставками для больших наборов данных. Если рассмотреть сортировку всего лишь с 26 блоками, то после $n > 256$ производительность сортировки хешированием начинает возрастать в четыре раза при удвоении входных данных, демонстрируя тем самым, что слишком малое количество блоков приводит к производительности $O(n^2)$.

Сортировка с использованием дополнительной памяти

Большинство алгоритмов сортировки работают "на месте", не требуя сколь-нибудь заметной дополнительной памяти. Теперь мы представим **сортировку слиянием** (Merge Sort), которая предлагает в худшем случае производительность $O(n \cdot log\ n)$, при использовании дополнительной памяти в количестве $O(n)$. Она может использоваться для эффективной сортировки данных, которые хранятся во внешнем файле.

Сортировка слиянием

Для сортировки множества A разделим его ровно на два меньших множества, каждое из которых затем сортируется. На заключительном этапе выполняется слияние этих двух отсортированных множеств в одно множество размера n. Приведенная далее прямая реализация этого подхода использует слишком много дополнительной памяти:

```
sort (A)
    if A имеет меньше 2 элементов then
        return A
    else if A имеет два элемента then
        Обменять элементы A, если они не в порядке
        return A

    sub1 = sort(left half of A)
    sub2 = sort(right half of A)

    Объединить sub1 и sub2 в новый массив B
    return B
end
```

Каждый рекурсивный вызов `sort` требует количества памяти, равного размеру массива, или $O(n)$, при этом выполняется $O(n \cdot log\ n)$ таких рекурсивных вызовов. Таким образом, прямая реализация требует $O(n \cdot log\ n)$ дополнительной памяти. К счастью, есть способ использовать только $O(n)$ дополнительной памяти, который мы сейчас рассмотрим.

Входные и выходные данные алгоритма

Результат сортировки возвращается в место в исходной коллекции *A*. Дополнительная память, использовавшаяся в процессе сортировки, освобождается.

Сортировка слиянием

Наилучший, средний и наихудший случаи: $O(n \cdot \log n)$

```
sort (A)
    copy = копия A                                    ❶
    mergeSort (copy, A, 0, n)
end

mergeSort (A, result, start, end)                     ❷
    if end - start < 2 then return
    if end - start = 2 then
        Обмен элементов, расположенных не по порядку
        return

    mid = (start + end)/2
    mergeSort (result, A, start, mid)                 ❸
    mergeSort (result, A, mid, end)

    Слияние левой и правой половин A в result         ❹
end
```

❶ Создается полная копия всех элементов.
❷ Элементы A[start,end) помещаются в result[start,end) в отсортированном порядке.
❸ Сортировка results[start,mid) в A[start,mid).
❹ Слияние отсортированных подмассивов A обратно в result.

Реализация алгоритма

Сортировка слиянием объединяет отсортированные подмассивы с использованием двух индексов, *i* и *j*, для перебора элементов левого и правого подмассивов, всегда копируя меньший из элементов $A[i]$ и $A[j]$ в нужное место в $result[idx]$. Следует рассматривать три случая:

- правый подмассив исчерпан ($j \geq end$), и в этом случае все остающиеся элементы берутся из левого подмассива;
- левый подмассив исчерпан ($i \geq mid$), и в этом случае все остающиеся элементы берутся из правого подмассива;

- и в левом, и в правом подмассивах есть элементы; если $A[i] < A[j]$, вставляется $A[i]$; в противном случае вставляется $A[j]$.

По завершении цикла `for` в *result* находятся слитые (и отсортированные) данные исходного массива $A[start, end)$. В примере 4.10 приведена реализация сортировки слиянием на языке программирования Python.

Пример 4.10. Реализация сортировки слиянием на языке программирования Python

```
def sort (A):
    """Сортировка слиянием A на месте."""
    copy = list (A)
    mergesort_array (copy, A, 0, len(A))

def mergesort_array (A, result, start, end):
    """Сортировка слиянием диапазона массива в памяти."""
    if end - start < 2:
        return
    if end - start == 2:
      if result[start] > result[start+1]:
        result[start],result[start+1] = result[start+1],result[start]
      return

    mid = (end + start) // 2
    mergesort_array (result, A, start, mid)
    mergesort_array (result, A, mid, end)

    # слияние левой и правой сторон A
    i = start
    j = mid
    idx = start
    while idx < end:
        if j >= end or (i < mid and A[i] < A[j]):
            result[idx] = A[i]
            i += 1
        else:
            result[idx] = A[j]
            j += 1

    idx += 1
```

Анализ алгоритма

Сортировка слиянием за время $O(n)$ завершает фазу "слияния" после рекурсивной сортировки левой и правой половин диапазона $A[start, end)$, размещая корректно упорядоченные элементы в массиве, который обозначен как *result*.

Так как *copy* является истинной копией всего массива *A*, прерывающие рекурсию базовые случаи будут корректно работать, поскольку в них *используются ссылки на исходные элементы массива непосредственно в местоположениях, соответствующих индексам*. Это замечание носит сложный характер и является ключевым для данного алгоритма. Кроме того, заключительный шаг слияния требует только $O(n)$ операций, что гарантирует общую производительность алгоритма, равную $O(n \cdot log\ n)$. Поскольку единственной дополнительной памятью, используемой алгоритмом, является *copy*, требования алгоритма к дополнительной памяти выражаются как $O(n)$.

Вариации алгоритма

Из всех алгоритмов сортировки сортировка слиянием наиболее просто преобразуется для работы с внешними данными. В примере 4.11 приведена полная реализация алгоритма на языке Java, использующая отображение данных в память для эффективной сортировки файла, содержащего целые числа в бинарном формате. Этот алгоритм сортировки требует, чтобы все элементы имели одинаковый размер, и поэтому не может быть легко адаптирован для сортировки произвольных строк или иных элементов переменной длины.

Пример 4.11. Реализация внешней сортировки слиянием на языке Java

```
public static void mergesort(File A) throws IOException {
    File copy = File.createTempFile("Mergesort", ".bin");
    copyFile(A, copy);

    RandomAccessFile src = new RandomAccessFile(A, "rw");
    RandomAccessFile dest = new RandomAccessFile(copy, "rw");
    FileChannel srcC = src.getChannel();
    FileChannel destC = dest.getChannel();
    MappedByteBuffer srcMap = srcC.map(
                FileChannel.MapMode.READ_WRITE,
                0, src.length());
    MappedByteBuffer destMap = destC.map(
                FileChannel.MapMode.READ_WRITE,
                0, dest.length());

    mergesort(destMap, srcMap, 0, (int) A.length());

    // Два следующих вызова необходимы только в Windows:
    closeDirectBuffer(srcMap);
    closeDirectBuffer(destMap);
    src.close();
    dest.close();
```

```
    copy.deleteOnExit();
}

static void mergesort(MappedByteBuffer A, MappedByteBuffer result,
                      int start, int end) throws IOException {
    if (end - start < 8) {
        return;
    }

    if (end - start == 8) {
        result.position(start);
        int left = result.getInt();
        int right = result.getInt();

        if (left > right) {
            result.position(start);
            result.putInt(right);
            result.putInt(left);
        }
        return;
    }

    int mid = (end + start) / 8 * 4;
    mergesort(result, A, start, mid);
    mergesort(result, A, mid, end);

    result.position(start);
    for (int i = start, j=mid, idx=start; idx<end; idx+=4) {
        int Ai = A.getInt(i);
        int Aj = 0;

        if (j < end) {
            Aj = A.getInt(j);
        }

        if (j >= end || (i < mid && Ai < Aj)) {
            result.putInt(Ai);
            i += 4;
        } else {
            result.putInt(Aj);
            j += 4;
        }
    }
}
```

Эта структура идентична реализации сортировки слиянием, но для эффективности использует отображение в память данных, хранящихся в файловой системе. В операционных системах Windows есть проблемы, связанные с неверным закрытием

`MappedByteBuffer`. В репозитории содержится обходной путь, заключающийся в вызове `closeDirectBuffer(MappedByteBuffer)`, который решает эту проблему.

Результаты хронометража для строк

Чтобы выбрать подходящий алгоритм для различных данных, необходимо знать некоторые характеристики входных данных. Мы создали несколько тестовых наборов данных для того, чтобы сравнить работу с ними алгоритмов, представленных в этой главе. Обратите внимание на то, что фактические значения, приведенные в таблицах, не играют особой роли, так как отражают специфику конкретного оборудования, на котором были выполнены тесты. Вместо этого следует обратить внимание на относительную производительность алгоритмов для соответствующих наборов данных.

Случайные строки

В этой главе мы демонстрировали производительность алгоритмов сортировки для сортировки строк из 26 символов, образованных путем перестановки букв в алфавите. Несмотря на общее количество $n!$, или приблизительно $4{,}03 \cdot 10^{26}$ таких строк, в наших наборах данных встречаются и дубликаты строк. Кроме того, стоимость сравнения элементов не является константной, так как иногда нужно сравнивать несколько символов строк.

Числа с плавающей точкой двойной точности

Используя имеющиеся генераторы псевдослучайных чисел, доступные в большинстве операционных систем, мы сгенерировали множество случайных чисел из диапазона [0,1). По сути, в этом множестве данных нет повторяющихся значений, а стоимость сравнения двух элементов постоянна. Результаты для этих наборов данных здесь отсутствуют, но их можно найти в репозитории.

Входные данные для алгоритмов сортировки можно предварительно обработать, чтобы гарантировать некоторые из следующих свойств (не все они совместимы между собой).

Отсортированность

Входные элементы могут быть отсортированы в порядке возрастания (конечная цель) или убывания.

Противодействие методу медианы трех элементов

Мюссер (Musser) [41] открыл упорядочение, которое гарантирует, что быстрой сортировке потребуется $O(n^2)$ сравнений при использовании для выбора опорного элемента метода медианы трех элементов.

Почти отсортированные данные

Для заданного набора отсортированных данных мы можем выбрать k пар элементов для обмена и расстояние d, на котором происходит обмен (или 0, если поменяться местами могут любые две пары). Используя эту возможность, можно создавать входные наборы, которые могли бы лучше соответствовать вашим входным данным.

Все таблицы упорядочены слева направо на основании того, насколько быстро работают алгоритмы в последней строке таблицы. Чтобы получить результаты, показанные в табл. 4.6–4.8, мы выполняли каждый хронометраж 100 раз и отбрасывали наилучшее и наихудшее значения времени. В таблицах показаны усредненные значения 98 оставшихся испытаний. Столбцы с надписью "Quicksort BFPRT[4] minSize=4" относятся к реализации быстрой сортировки, которая использует алгоритм BFPRT (с группами размером 4) для выбора опорного элемента, и переключается на сортировку вставками, когда подмассив имеет размер четыре или меньше элементов.

Поскольку производительность быстрой сортировки с медианой трех элементов деградирует очень быстро, в табл. 4.8 выполнялись только 10 запусков.

Таблица 4.6. Время сортировки (в секундах) случайных 26-буквенных перестановок алфавита

n	Hash Sort, 17576 блоков	Quicksort, медиана трех элементов	Merge Sort	Heap Sort	Quicksort, BFPRT[4] minSize = 4
4096	0,000631	0,000741	0,000824	0,0013	0,0028
8192	0,001100	0,001600	0,001800	0,0029	0,0062
16384	0,002000	0,003500	0,003900	0,0064	0,0138
32768	0,004000	0,007700	0,008400	0,0147	0,0313
65536	0,010700	0,016800	0,018300	0,0336	0,0703
131072	0,035900	0,042000	0,044400	0,0893	0,1777

Таблица 4.7. Время сортировки (в секундах) отсортированных случайных 26-буквенных перестановок алфавита

n	Insertion Sort	Merge Sort	Quicksort, медиана трех элементов	Hash Sort, 17576 блоков	Heap Sort	Quicksort, BFPRT[4] minSize = 4
4096	0,000029	0,000434	0,000390	0,000552	0,0012	0,0016
8192	0,000058	0,000932	0,000841	0,001000	0,0026	0,0035
16384	0,000116	0,00200	0,001800	0,001900	0,0056	0,0077
32768	0,000237	0,004100	0,003900	0,003800	0,0123	0,0168
65536	0,000707	0,008600	0,008500	0,009200	0,0269	0,0364
131072	0,002500	0,018900	0,019800	0,024700	0,0655	0,0834

Таблица 4.8. Время сортировки (в секундах) данных, полученных методом противодействия медиане трех элементов

n	Merge Sort	Hash Sort, 17576 блоков	Heap Sort	Quicksort, BFPRT[4] minSize = 4	Quicksort, медиана трех элементов
4096	0,000505	0,000569	0,0012	0,0023	0,0212
8192	0,001100	0,001000	0,0026	0,0050	0,0841
16384	0,002300	0,001900	0,0057	0,0108	0,3344
32768	0,004700	0,003800	0,0123	0,0233	1,3455
65536	0,009900	0,009100	0,0269	0,0506	5,4027
131072	0,022400	0,028300	0,0687	0,1151	38,0950

Анализ методов

В ходе анализа алгоритмов сортировки необходимо указать производительность в наилучшем, наихудшем и среднем случаях (как описано в главе 2, "Математика алгоритмов"). Средний случай обычно дается труднее всего, и для определения производительности требуется применение передовых математических методов и оценок. Он также предполагает знания о характеристиках данных, например вероятность того, что входные данные могут быть частично отсортированы. Даже когда показано, что алгоритм имеет желательную производительность в среднем случае, его реализация может быть просто непрактичной. Каждый алгоритм сортировки в этой главе анализируется и с точки зрения его теоретического поведения, и с точки зрения фактического поведения на практике.

Фундаментальным научным результатом является тот факт, что не существует алгоритма сортировки сравнением элементов, который имел бы лучшую производительность, чем $O(n \cdot log\ n)$, в среднем или наихудшем случае. Набросаем здесь эскиз доказательства. Если имеется n элементов, то всего есть $n!$ перестановок этих элементов. Каждый алгоритм, который сортирует элементы с помощью попарных сравнений, соответствует бинарному дереву решений. Листья дерева соответствуют перестановкам, и каждая перестановка должна соответствовать хотя бы одному листу дерева. Узлы на пути от корня к листу соответствуют последовательности сравнений. Высота такого дерева определяется как число узлов сравнения на самом длинном пути от корня к листьям; например, высота дерева на рис. 4.10 равна 5, потому что во всех случаях достаточно только пяти сравнений (хотя в четырех случаях хватает четырех сравнений).

Построим бинарное дерево решений, в котором каждый внутренний узел дерева представляет собой сравнение $a_i \leq a_j$, а листья дерева представляют собой одну из $n!$ перестановок. Чтобы отсортировать набор из n элементов, надо начинать с корня и вычислять операторы сравнения в каждом узле. Если утверждение в узле верно, мы переходим в левый дочерний узел, если нет — в правый. На рис. 4.10 показан пример дерева решений для четырех элементов.

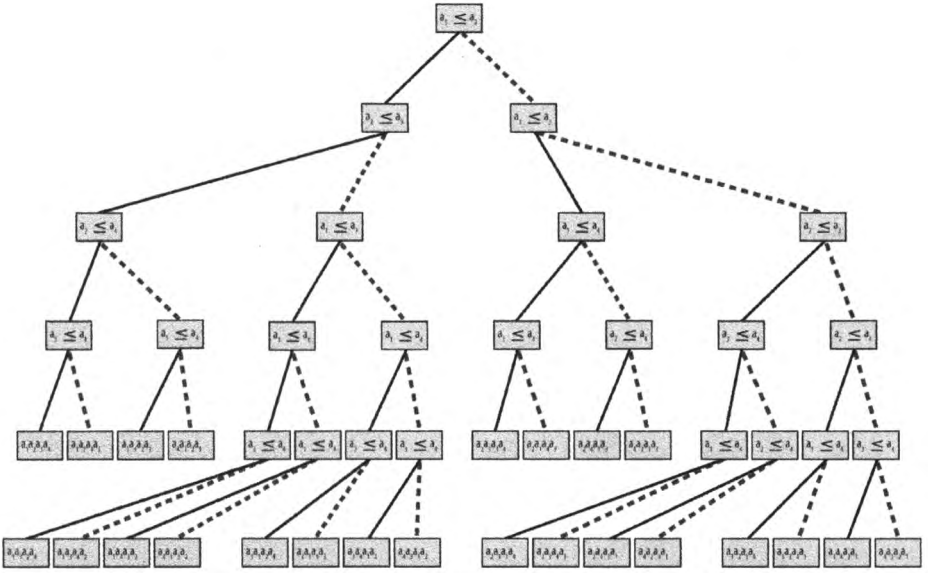


Рис. 4.10. Бинарное дерево решений для упорядочения четырех элементов

Можно построить много различных бинарных деревьев решений. Тем не менее мы утверждаем, что для любого заданного бинарного дерева решений для сравнения n элементов можно указать его минимальную высоту h, т.е. должен быть некоторый лист, который требует h узлов сравнений на пути от корня до этого листа дерева. Рассмотрим полное двоичное дерево высотой h, в котором все узлы, не являющиеся листьями, имеют как левый, так и правый дочерние узлы. Это дерево содержит $n=2^h-1$ узлов, а его высота равна $h=log(n+1)$. Если дерево не является полным, оно может быть несбалансированным любым, пусть даже самым странным, образом, но мы знаем, что $h\geq\lceil log(n+1)\rceil$. Любое бинарное дерево решений с $n!$ конечных узлов уже имеет минимум $n!$ узлов в общей сложности. Нам осталось только вычислить $h=\lceil log(n!)\rceil$ для определения высоты любого такого бинарного дерева решений. Воспользуемся следующими свойствами логарифмов:

$$log\left(a\cdot b\right)=log\left(a\right)+log\left(b\right) \text{ и } log\left(x^y\right)=y\cdot log\left(x\right).$$

Тогда

$$h=log\left(n!\right)=log\left(n\cdot\left(n-1\right)\cdot\left(n-2\right)\cdot\ldots\cdot2\cdot1\right)$$

$$h>log\left(n\cdot\left(n-1\right)\cdot\left(n-2\right)\cdot\ldots\cdot n/2\right)$$

$$h>log\left(\left(n/2\right)^{n/2}\right)$$

$$h>\left(n/2\right)\cdot log\left(n/2\right)$$

$$h>\left(n/2\right)\cdot\left(log\left(n\right)-1\right)$$

Что же означает последнее неравенство? Для заданных n элементов, которые необходимо отсортировать, будет хотя бы один путь от корня к листу длиной h, а значит, алгоритм, который сортирует с помощью сравнений, требует как минимум указанного количества сравнений для сортировки n элементов. Обратите внимание, что h вычисляется как функция $f(n)$; в частности, в данном случае $f(n) = (1/2) \cdot n \cdot log\ n - n/2$. Таким образом, любой алгоритм с использованием сравнений потребует выполнения $O(n \cdot log\ n)$ сравнений для сортировки.

5
Поиск

Для заданной коллекции *C* элементов имеется два фундаментальных вопроса.

Существование

Содержит ли *C* некоторый элемент? Для заданной коллекции *C* мы часто просто хотим знать, содержится ли в ней некоторый элемент *t*. Ответ на такой запрос имеет значение `true`, если в коллекции имеется элемент, соответствующий запрашиваемому элементу *t*, или `false`, если это не так.

Ассоциативный поиск

Возврат информации, связанной в коллекции *C* с искомым значением ключа *k*. Ключ обычно связан со сложной структурой, именуемой значением. Поиск извлекает или заменяет это значение.

Алгоритмы в этой главе описывают пути структуризации данных для более эффективного выполнения поисковых запросов. Например, можно упорядочить коллекцию *C*, используя алгоритмы сортировки, рассматривавшиеся в главе 4, "Алгоритмы сортировки". Как мы увидим, сортировка улучшает производительность поиска, но есть и другие расходы, связанные с поддержкой коллекции в отсортированном состоянии, в особенности когда элементы часто вставляются или удаляются.

В конечном счете производительность определяется тем, сколько элементов проверяется алгоритмом при обработке запроса. Используйте следующие советы при выборе наилучшим образом подходящего для ваших задач алгоритма.

Небольшие коллекции

Последовательный поиск предлагает простейшую реализацию и во многих языках программирования реализуется в виде базовой конструкции. Используйте этот алгоритм, если коллекция доступна только последовательно, как при использовании *итераторов*.

Ограниченная память

Когда коллекция представляет собой неизменный массив и вы хотите сэкономить память, используйте бинарный поиск.

Динамические данные

Если элементы в коллекции часто изменяются, рассмотрите возможность применения поиска на основе хеша, а также бинарного дерева поиска, которые обладают пониженной стоимостью, связанной с поддержанием их структур данных.

Отсортированный доступ

Если вам нужна поддержка часто меняющихся данных и возможность обхода элементов коллекции в отсортированном порядке, используйте бинарное дерево поиска.

Не забывайте о стоимости предварительной обработки структур данных, которая может потребоваться алгоритму до начала выполнения поисковых запросов. Выберите подходящую структуру данных, которая не только ускоряет производительность отдельных запросов, но и сводит к минимуму общую стоимость поддержания структуры коллекции в условиях динамического доступа и многократных запросов.

Мы предполагаем наличие множества U (универсума) всех возможных значений. Коллекция C содержит элементы, взятые из U, а запрашиваемый элемент t является членом U. Если t представляет собой ключевое значение, то мы рассматриваем U как множество всех возможных значений ключа, $k \in U$, а коллекция C может содержать более сложные элементы. Обратите внимание, что в коллекции C могут иметься повторяющиеся значения, и поэтому ее нельзя рассматривать как множество (в котором не допускается наличие одинаковых элементов).

Если коллекция C допускает индексирование произвольных элементов, мы ссылаемся на коллекцию как на массив A с использованием записи $A[i]$, которая представляет i-й элемент A. По соглашению значение `null` представляет элемент, отсутствующий в U; такое значение полезно, когда поиск должен вернуть определенный элемент коллекции, но этот элемент в ней отсутствует. В общем случае мы предполагаем, что найти в коллекции значение `null` невозможно.

Последовательный поиск

Последовательный поиск (Sequential Search), называемый также линейным поиском, является самым простым из всех алгоритмов поиска. Это метод поиска одного значения t в коллекции C "в лоб". Он находит t, начиная с первого элемента коллекции и исследуя каждый последующий элемент до тех пор, пока не просмотрит всю коллекцию или пока соответствующий элемент не будет найден.

Для работы этого алгоритма должен иметься способ получения каждого элемента из коллекции, в которой выполняется поиск; порядок извлечения значения не имеет. Часто элементы коллекции C могут быть доступны только через *итератор*, который предназначен только для чтения и получает каждый элемент из C, как, например, курсор базы данных в ответ на SQL-запрос. В книге рассмотрены оба режима доступа.

Входные и выходные данные алгоритма

Входные данные представляют собой непустую коллекцию C из $n>0$ элементов и целевое значение t, которое мы ищем. Поиск возвращает `true`, если коллекция C содержит t, и `false` в противном случае.

Последовательный поиск

Наилучший случай: $O(1)$; средний и наихудший случаи: $O(n)$

```
search (A,t)
    for i=0 to n-1 do                ❶
        if A[i] = t then
            return true
    return false
end

search (C,t)
    iter = C.begin()
    while iter ≠ C.end() do          ❷
        e = iter.next()              ❸
        if e = t then
            return true
    return false
end
```

❶ Поочередный доступ к каждому элементу от позиции 0 до n-1.
❷ Итерации продолжаются до полного исчерпания элементов.
❸ Все элементы последовательно извлекаются из итератора.

Контекст применения алгоритма

Зачастую требуется найти элемент в коллекции, которая может быть не упорядочена. Без дополнительных знаний об информации, хранящейся в коллекции, последовательный поиск выполняет работу методом "грубой силы". Это единственный алгоритм поиска, который можно использовать, если коллекция доступна только через итератор.

Если коллекция неупорядоченная и хранится в виде связанного списка, то вставка элемента является операцией с константным временем выполнения (он просто добавляется в конец списка). Частые вставки в коллекцию на основе массива требуют управления динамической памятью, которое либо предоставляется базовым языком программирования, либо требует определенного внимания со стороны программиста. В обоих случаях ожидаемое время поиска элемента равно $O(n)$; таким образом, удаление элемента требует по крайней мере времени $O(n)$.

Последовательный поиск накладывает наименьшее количество ограничений на типы искомых элементов. Единственным требованием является наличие функции проверки совпадения элементов, необходимой для выяснения, соответствует ли искомый элемент элементу коллекции; часто эта функция возлагается на сами элементы.

Реализация алгоритма

Обычно реализация последовательного поиска тривиальна. В примере 5.1 показана реализация последовательного поиска на языке программирования Python.

Пример 5.1. Последовательный поиск на языке Python

```
def sequentialSearch(collection, t):
    for e in collection :
        if e == t:
            return True
    return False
```

Код обезоруживающе прост. Функция получает коллекцию и запрашиваемый элемент *t*. Коллекция может быть списком или любым иным *итерируемым* объектом Python. Элементы, участвующие в поиске, должны поддерживать оператор `==`. Этот же код, написанный на Java, показан в примере 5.2. Обобщенный класс `SequentialSearch` имеет параметр типа `T`, который определяет элементы в коллекции; `T` должен предоставлять корректный метод `equals(Object o)` для правильной работы кода.

Пример 5.2. Последовательный поиск на языке Java

```
public class SequentialSearch<T>
{
    /** Применение метода грубой силы для поиска в индексируемой
     *  коллекции (типа T) заданного элемента. */
    public boolean search(T[] collection, T t)
    {
        for (T item : collection)
        {
            if (item.equals(t))
            {
                return true;
```

```
            }
        }

        return false;
    }

    /** Применение метода грубой силы для поиска в итерируемой
     *  коллекции (типа T) заданного элемента. */
    public boolean search(Iterable<T> collection, T t)
    {
        Iterator<T> iter = collection.iterator();

        while (iter.hasNext())
        {
            if (iter.next().equals(t))
            {
                return true;
            }
        }

        return false;
    }
}
```

Анализ алгоритма

Если запрашиваемый элемент принадлежит коллекции и равновероятно может быть найден в любом из индексируемых местоположений (или, в качестве альтернативы, если он с одинаковой вероятностью может быть возвращен итератором в любой позиции), то в среднем последовательный поиск проверяет $n/2 + 1/2$ элементов (как показано в главе 2, "Математика алгоритмов"). Таким образом, для каждого искомого элемента, имеющегося в коллекции, будет проверяться около всех половины элементов, так что производительность поиска оказывается равной $O(n)$. В наилучшем случае, когда запрашиваемый элемент является первым элементом коллекции, производительность оказывается равной $O(1)$. В среднем и наихудшем случаях алгоритм последовательного поиска обладает производительностью $O(n)$. Если размер коллекции вырастает в два раза, то и время, затрачиваемое на поиск в ней, должно примерно удваиваться.

Чтобы показать последовательный поиск в действии, построим упорядоченную коллекцию из n целых чисел из диапазона $[1,n]$. Хотя коллекция и упорядочена, эта информация в коде поиска не используется. Мы проводим набор из 100 испытаний; в каждом испытании мы выполняем 1000 запросов случайного числа t, которое присутствует в коллекции с вероятностью p. Таким образом, из 1000 запросов $p \cdot 1000$ запросов гарантированно находят t в коллекции (для $p = 0{,}0$ искомый элемент

t представляет собой отрицательное число). Как обычно, время выполнения было усреднено, с отбрасыванием наилучшего и наихудшего значений. В табл. 5.1 показано среднее значение для оставшихся 98 испытаний, для четырех конкретных значений p. Обратите внимание, что время выполнения примерно удваивается с удвоением размера коллекции. Вы должны также заметить, что для каждой коллекции наихудшая производительность находится в последнем столбце, в котором показаны результаты неудачного поиска, когда искомого элемента t в коллекции нет.

Таблица 5.1. Производительность последовательного поиска (в секундах)

n	$p = 1,0$	$p = 0,5$	$p = 0,25$	$p = 0,0$
4096	0,0057	0,0087	0,0101	0,0116
8192	0,0114	0,0173	0,0202	0,0232
16384	0,0229	0,0347	0,0405	0,0464
32768	0,0462	0,0697	0,0812	0,0926
65536	0,0927	0,1391	0,1620	0,1853
131072	0,1860	0,2786	0,3245	0,3705

Бинарный поиск

Бинарный (двоичный) поиск обеспечивает лучшую производительность, чем последовательный поиск, поскольку работает с коллекцией, элементы которой уже отсортированы. Бинарный поиск многократно делит отсортированную коллекцию пополам, пока не будет найден искомый элемент или пока не будет установлено, что элемент в коллекции отсутствует.

Входные и выходные данные алгоритма

Входными данными для бинарного поиска является индексируемая коллекция A, элементы которой полностью упорядочены, а это означает, что для двух индексов, i и j, $A[i] < A[j]$ только тогда, когда $i < j$. Мы строим структуру данных, которая хранит элементы (или указатели на элементы) и сохраняет порядок ключей. Выходные данные бинарного поиска — `true` или `false`.

Контекст применения алгоритма

Для поиска по упорядоченной коллекции в худшем случае необходимо логарифмическое количество проб.

Бинарный поиск поддерживают различные типы структур данных. Если коллекция никогда не изменяется, элементы следует поместить в массив (это позволяет легко перемещаться по коллекции). Однако, если необходимо добавить или удалить элементы из коллекции, этот подход становится неудачным и требует дополнительных расходов. Есть несколько структур данных, которые мы можем использовать в этом

случае; одной из наиболее известных является бинарное дерево поиска, описываемое далее в этой главе.

Бинарный поиск

Наилучший случай: $O(1)$; средний и наихудший случаи: $O(\log n)$

```
search (A,t)
    low = 0
    high = n-1
    while low ≤ high do                ❶
        mid = (low + high)/2           ❷
        if t < A[mid] then
            high = mid - 1
        else if t > A[mid] then
            low = mid + 1
        else
            return true
    return false                       ❸
end
```

❶ Повторяется, пока есть диапазон для поиска.
❷ Средняя точка вычисляется с помощью целочисленной арифметики.
❸ Далее, в разделе "Вариации алгоритма", обсуждается поддержка операции "поиск или вставка" на основе конечного значения mid в этой точке.

Реализация алгоритма

Для упорядоченной коллекции элементов, представленной в виде массива, в примере 5.3 приведен код Java параметризованной реализации бинарного поиска для произвольного базового типа T. Java предоставляет интерфейс `java.util.Comparable<T>`, содержащий метод `compareTo`. Любой класс, который корректно реализует этот интерфейс, гарантирует нестрогое упорядочение его экземпляров.

Пример 5.3. Реализация бинарного поиска на языке Java

```
/**
* Бинарный поиск в предварительно отсортированном массиве
* параметризованного типа.
*
* @param T Элементы коллекции имеют данный тип.
* Тип T должен реализовывать интерфейс Comparable.
*/
public class BinarySearch<T extends Comparable<T>>
{
```

```
    /** Поиск ненулевого элемента; возврат true в случае успеха. */
    public boolean search(T[] collection, T target)
    {
        if (target == null)
        {
            return false;
        }

        int low = 0, high = collection.length - 1;

        while (low <= high)
        {
            int mid = (low + high) / 2;
            int rc = target.compareTo(collection[mid]);

            if (rc < 0)        // Искомый элемент < collection[i]
            {
                high = mid - 1;
            }
            else if (rc > 0)   // Искомый элемент > collection[i]
            {
                low = mid + 1;
            }
            else               // Искомый элемент найден
            {
                return true;
            }
        }

        return false;
    }
}
```

Реализация использует три переменные: `low`, `high` и `mid`. `low` представляет собой наименьший индекс текущего подмассива, в котором выполняется поиск, `high` — его наибольший индекс, а `mid` — средняя точка. Производительность этого кода зависит от количества итераций цикла `while`.

Для достижения большей производительности бинарный поиск может быть немного усложнен. Сложность увеличивается, если коллекция хранится не в виде структуры данных в памяти, такой как массив. Большая коллекция может потребовать размещения во вторичной памяти, такого, как в файле на диске. В таком случае доступ к *i*-му элементу осуществляется по смещению его местоположения в файле. При использовании вторичной памяти во времени, необходимом для поиска элемента, преобладают затраты на доступ к памяти; таким образом, может быть целесообразным использование других решений, связанных с бинарным поиском.

Анализ алгоритма

Двоичный поиск на каждой итерации уменьшает размер задачи примерно в два раза. Максимальное количество делений пополам для коллекции размером n равно $\lfloor log\ n \rfloor + 1$. Если одной операции достаточно, чтобы определить, равны ли два элемента, меньше или больше один другого (это делает возможным применение интерфейса `Comparable`), то достаточно выполнить только $\lfloor log\ n \rfloor + 1$ сравнений. Таким образом, алгоритм можно классифицировать как имеющий производительность $O(log\ n)$.

Мы выполнили 100 испытаний по 524 288 поисков элемента, хранящегося в коллекции размера n, размещенной в памяти (размер коллекции колеблется в диапазоне от 4096 до 524 288 элементов) с вероятностью наличия искомого элемента, равной p (и принимающей значения 1,0, 0,5 и 0,0). После удаления лучших и худших значений времени выполнения для каждого испытания результаты хронометража усреднялись. В табл. 5.2 показана усредненная производительность оставшихся 98 испытаний.

Таблица 5.2. Сравнение времени выполнения 524 288 бинарных поисков в памяти с последовательным поиском (в секундах)

n	Время последовательного поиска			Время бинарного поиска		
	$p = 1,0$	$p = 0,5$	$p = 0,0$	$p = 1,0$	$p = 0,5$	$p = 0,0$
4096	3,0237	4,5324	6,0414	0,0379	0,0294	0,0208
8192	6,0405	9,0587	12,0762	0,0410	0,0318	0,0225
16384	12,0742	18,1086	24,1426	0,0441	0,0342	0,0243
32768	24,1466	36,2124	48,2805	0,0473	0,0366	0,0261
65536	48,2762	72,4129	96,5523	0,0508	0,0395	0,0282
131072	*	*	*	0,0553	0,0427	0,0300
262144	*	*	*	0,0617	0,0473	0,0328
524288	*	*	*	0,0679	0,0516	0,0355

Испытания были разработаны так, чтобы обеспечить при $p = 1,0$ равную вероятность поиска любого элемента в коллекции (если это не так, результаты могут быть искажены). И для последовательного, и для бинарного поиска входные данные представляют собой массив отсортированных целых чисел в диапазоне $[0, n)$. Для получения 524 288 искомых значений, которые имеются в коллекции ($p = 1,0$), мы циклически проходили по n числам $524288/n$ раз.

В табл. 5.3 показано время выполнения 524 288 поисков в коллекции, хранящейся на локальном диске. Выполнялись два варианта поиска — имеющегося в коллекции элемента ($p = 1,0$) и элемента, которого там нет (поиск значения –1 в коллекции $[0, n)$). Данные представляли собой просто файл возрастающих целых чисел, в котором каждое целое число упаковано в четыре байта. Доминирование времени обращения к диску очевидно, так как результаты в табл. 5.3 почти в 400 раз медленнее, чем в табл. 5.2. Обратите внимание, как время поиска увеличивается на фиксированную

величину при удвоении размера n, что является четким подтверждением того факта, что производительность бинарного поиска — $O(\log n)$.

Таблица 5.3. Бинарный поиск во вторичной памяти (524 288 поисков, в секундах)

n	$p = 1{,}0$	$p = 0{,}0$
4 096	1,2286	1,2954
8 192	1,3287	1,4015
16 384	1,4417	1,5080
32 768	6,7070	1,6170
65 536	13,2027	12,0399
131 072	19,2609	17,2848
262 144	24,9942	22,7568
524 288	30,3821	28,0204

Вариации алгоритма

Для поддержки операции "поиск или вставка" заметим, что все допустимые индексы не являются отрицательными. Вариация алгоритма на языке Python в примере 5.4 содержит метод `bs_contains`, который возвращает отрицательное число p при поиске элемента, который отсутствует в упорядоченном массиве. Значение $-(p+1)$ представляет собой индекс позиции, в которую следует вставить искомый элемент, как показано в `bs_insert`. Естественно, что при вставке следует перенести все значения с большими индексами, чтобы освободить место для нового элемента.

Пример 5.4. Версия "поиск или вставка" на языке Python

```
def bs_contains (ordered, target):
    """Возврат индекса искомого элемента или -(p+1) для вставки."""
    low = 0
    high = len(ordered)-1
    while low <= high:
        mid = (low + high) // 2
        if target < ordered[mid]:
            high = mid-1
        elif target > ordered[mid]:
            low = mid+1
        else:
            return mid

    return -(low + 1)

def bs_insert (ordered, target):
    """Вставка элемента при его отсутствии в корректное место."""
    idx = bs_contains (ordered, target)
```

```
if idx < 0:
    ordered.insert (-(idx + 1), target)
```

Вставка в упорядоченный массив (или удаление из него) с ростом размера массива становится неэффективной, потому что каждая запись массива должна содержать корректный элемент. Таким образом, вставка включает расширение массива (физическое или логическое) и перемещение в среднем половины элементов на одну позицию вперед. Удаление требует сжатия массива и перемещения половины элементов на одну позицию назад.

Поиск на основе хеша

В предыдущих разделах по поиску описаны алгоритмы, пригодные при небольшом количестве элементов (последовательный поиск) или в упорядоченной коллекции (бинарный поиск). Но нам нужны более мощные методы поиска в больших коллекциях, которые не обязательно упорядочены. Одним из наиболее распространенных подходов к решению этой задачи является использование *хеш-функции* для преобразования одной или нескольких характеристик искомого элемента в индекс в хеш-таблице. **Поиск на основе хеша** (Hash-Based Search) имеет в среднем случае производительность, которая лучше производительности любого другого алгоритма поиска, описанного в этой главе. Во многих книгах об алгоритмах рассматривается поиск на основе хеша в теме, посвященной хеш-таблицам [20]. Вы также можете найти эту тему в книгах о структурах данных, в которых рассматриваются хеш-таблицы.

В поиске на основе хеша n элементов коллекции C сначала загружаются в хеш-таблицу H с b ячейками ("корзинами"), структурированными в виде массива. Этот *шаг предварительной обработки* имеет производительность $O(n)$, но улучшает производительность будущих поисков с использованием концепции *хеш-функции*.

Хеш-функция представляет собой детерминированную функцию, которая отображает каждый элемент C_i на целочисленное значение h_i. На минуту предположим, что $0 \le h_i < b$. При загрузке элементов в хеш-таблицу элемент C_i вставляется в ячейку $H[h_i]$. После того как все элементы будут вставлены, поиск элемента t становится поиском t в $H[hash(t)]$.

Хеш-функция гарантирует только, что если два элемента, C_i и C_j, равны, то $hash(C_i) = hash(C_j)$. Может случиться так, что несколько элементов в C имеют одинаковые значения хеша; такая ситуация называется *коллизией*, и хеш-таблице необходима стратегия для урегулирования подобных ситуаций. Наиболее распространенным решением является хранение в каждой ячейке связанного списка (несмотря на то, что многие из этих связанных списков будут содержать только один элемент); при этом в хеш-таблице могут храниться все элементы, вызывающие коллизии. Поиск в связанных списках должен выполняться линейно, но он будет быстрым, потому

что каждый из них может хранить максимум несколько элементов. Приведенный далее псевдокод описывает решение коллизий с использованием связанного списка.

Общая схема поиска на основе хеша показана на рис. 5.1 (здесь рассматривается небольшой пример ее применения). Ее компонентами являются:

- множество U, которое определяет множество возможных хеш-значений. Каждый элемент $e \in C$ отображается на хеш-значение $h \in U$;
- хеш-таблица H, содержащая b ячеек, в которых хранятся n элементов из исходной коллекции C;
- хеш-функция *hash*, которая вычисляет целочисленное значение h для каждого элемента e, где $0 \le h < b$.

Эта информация хранится в памяти с использованием массивов и связанных списков.

У поиска на основе хеша имеются две основные проблемы: разработка хеш-функции и обработка коллизий. Плохо выбранная хеш-функция может привести к плохому распределению ключей в первичной памяти с двумя последствиями: *а*) многие ячейки в хеш-таблице могут остаться неиспользованными и зря тратить память, и *б*) будет иметься много коллизий, когда много ключей попадают в одну и ту же ячейку, что существенно ухудшает производительность поиска.

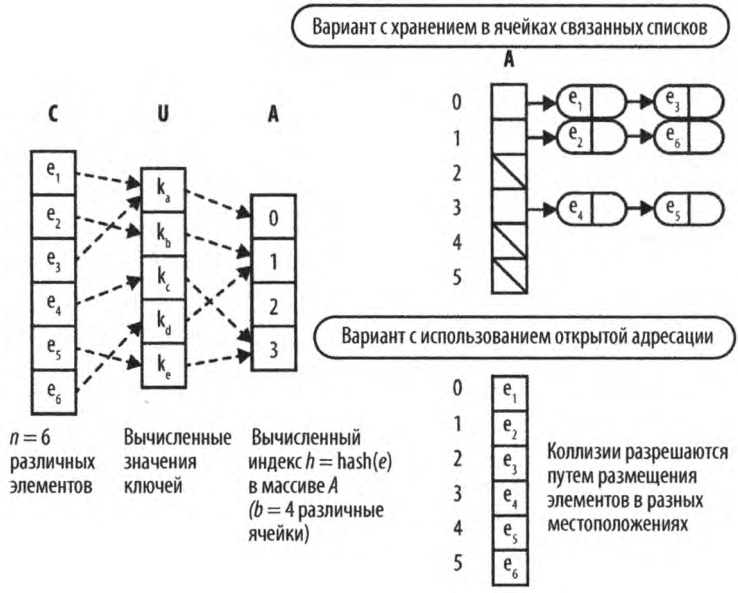


Рис. 5.1. Общая схема хеширования

Входные и выходные данные алгоритма

В отличие от бинарного поиска при поиске на основе хеша исходная коллекция *C* не обязана быть упорядоченной. Действительно, даже если элементы *C* были некоторым образом упорядочены, метод хеширования, вставляющий элементы в хеш-таблицу *H*, не пытается воспроизвести этот порядок в *H*.

Входными данными для поиска на основе хеша являются вычисленная хеш-таблица *H* и искомый элемент *t*. Алгоритм возвращает `true`, если *t* имеется в связанном списке, хранящемся в *H*[*h*], где *h*=*hash*(*t*). Если *t* в связанном списке, хранящемся в *H*[*h*], отсутствует, возвращается значение `false`, указывающее, что *t* нет в *H* (и, таким образом, его нет в *C*).

Контекст применения алгоритма

Предположим, что у нас есть массив из 213 557 отсортированных английских слов (которые можно найти в упоминавшемся в предисловии к книге репозитории). Из нашего обсуждения бинарного поиска известно, что мы можем ожидать в среднем около 18 сравнений строк при поиске слова в этом массиве (поскольку *log*(213 557) = 17,70). При поиске на основе хеша количество сравнений строк связано с длиной связанных списков, а не с размером коллекции.

Сначала необходимо определить *хеш-функцию*. Цель заключается в том, чтобы получить как можно больше различных значений, но не все эти значения обязаны быть уникальными. Хеширование изучалось на протяжении десятилетий, и в многочисленных работах описано множество эффективных хеш-функций, но они могут использоваться для гораздо большего, чем простой поиск. Например, специальные хеш-функции имеют важное значение для криптографии. Для поиска хеш-функция должна обладать хорошим равномерным распределением и быть быстро вычисляемой за малое количество тактов процессора.

Поиск на основе хеша

Наилучший и средний случаи: *O*(*1*); наихудший случай: *O*(*n*)

```
loadTable (size, C)
    H = новый массив заданного размера
    foreach e in C do
        h = hash(e)
        if H[h] пуст then        ❶
            H[h] = new Linked List
        добавить e в H[h]
    return A
end
```

```
search (H, t)
    h = hash(t)
    list = H[h]
    if list пуст then
        return false
    if list содержит t then     ❷
        return true
    return false
end

❶ Создание связанных списков при вставке e в пустую ячейку.
❷ Для малых списков используется последовательный поиск.
```

Популярным методом является получение значения с учетом каждого символа исходной строки:

$$hashCode(s) = s[0] \cdot 31^{(len-1)} + s[1] \cdot 31^{(len-2)} + \cdots + s[len-1],$$

где $s[i]$ является i-м символом (представляющим собой значение ASCII от 0 до 255), а len — длина строки s. Вычисление этой функции достаточно простое, как демонстрируется в примере 5.5 с исходным текстом на языке программирования Java (адаптировано из исходных текстов Open JDK (`http://openjdk.java.net`)), где `chars` — массив символов, который определяет строку. Согласно нашему определению метод `hashCode()` для класса `java.lang.String` не является хеш-функцией, потому что не гарантируется, что вычисленное значение будет находиться в диапазоне $[0, b)$.

Пример 5.5. Пример `hashCode()` на Java

```
public int hashCode()
{
    int h = hash;

    if (h == 0)
    {
        for (int i = 0; i < chars.length; i++)
        {
            h = 31 * h + chars[i];
        }

        hash = h;
    }

    return h;
}
```

Для эффективности этот метод `hashCode` кеширует значение вычисленного хеша, чтобы избежать повторных вычислений (т.е. вычисляет значение, только если `hash` равен 0). Заметим, что эта функция возвращает очень большое целочисленное значение и иногда может становиться отрицательной, поскольку тип `int` в Java может хранить только 32 бита информации. Для вычисления целочисленной ячейки для данного элемента *hash*(*s*) определяется как

$$hash(s) = abs(hashCode(s))\%b,$$

где *abs* представляет собой функцию получения абсолютного значения, а % — оператор деления по модулю, который возвращает остаток от деления числа на b. Это гарантирует, что вычисленное целочисленное значение находится в диапазоне $[0, b)$.

Выбор функции хеширования представляет собой только первое решение, которое нужно сделать при реализации поиска на основе хеша. Еще одним вопросом является использование памяти. Первичная память H должна быть достаточно большой, а связанные списки элементов, хранящиеся в ячейках, должны быть как можно меньшими. Можно решить, что H может содержать $b = n$ ячеек, а хеш-функция — просто однозначно сопоставлять набор строк коллекции целых чисел $[0, n)$, но сделать это нелегко! Вместо этого мы стараемся определить хеш-таблицу, которая будет содержать как можно меньше пустых ячеек. Если наша хеш-функция равномерно распределяет ключи, достичь разумного успеха можно, выбрав размер массива примерно равным размеру коллекции.

В качестве размера H обычно выбирается простое число, чтобы гарантировать, что использование оператора вычисления остатка от деления `%` эффективно распределяет вычисляемые номера ячеек. На практике хорошим выбором является $2^k - 1$, несмотря на то что это значение не всегда простое.

Элементы, хранящиеся в хеш-таблице, непосредственно влияют на память. Так как каждая ячейка хранит элементы в связанном списке, элементами списка являются указатели на объекты в динамической памяти. Каждый список имеет накладные расходы памяти для указателей на первый и последний элементы списка и, если вы используете класс `LinkedList` из Java JDK, для значительного количества дополнительных полей. Мы могли бы написать гораздо более простой класс связанного списка, предоставляющий только необходимые возможности, но это, безусловно, поднимает стоимость реализации поиска на основе хеша.

Заинтересованный читатель должен в этот момент задать вопрос об использовании базовой хеш-функции и хеш-таблицы для данной задачи. Если список слов фиксирован и не будет изменяться, мы можем решить проблему путем создания идеальной хеш-функции. Идеальная хеш-функция гарантирует отсутствие коллизий для определенного набора ключей; этот вариант рассматривается в разделе "Вариации алгоритма" далее в этой главе. Но давайте сначала попробуем решить проблему иначе.

В качестве первой попытки решения задачи выберем базовую хеш-таблицу H, которая будет содержать $b = 2^{18} - 1 = 262\,143$ элемента. Наш список слов содержит 213 557 слов. Если наша хеш-функция идеально распределяет строки, коллизий не будет вовсе, и останется около 40 000 пустых ячеек. Однако на самом деле все не так благополучно. В табл. 5.4 показано распределение значений хеша для класса Java `String` для нашего списка слов с таблицей из 262 143 ячеек. Напомним, что $hash(s) = abs(hashCode(s)) \% b$. Как вы можете видеть, нет ячеек, содержащих более семи строк; для непустых ячеек среднее количество строк в ячейке составляет примерно 1.46. Каждая строка показывает количество использованных ячеек и количество слов, хешированных в эти ячейки. Почти половина ячеек таблицы (116 186) не имеют каких-либо строк, хешированных в них. Таким образом, данная функция хеширования тратит впустую около 500 Кбайт памяти (при условии, что размер указателя составляет четыре байта). Вас может удивить тот факт, что это действительно хорошая функция хеширования и что для поиска функции с лучшим распределением понадобится более сложная схема.

Заметим, кстати, что имеется только пять пар строк с идентичными значениями `hashCode` (например, слова "hypoplankton" и "unheavenly" имеют одно и то же вычисленное значение `hashCode`, равное 427 589 249).

Таблица 5.4. Распределение хеш-значений с использованием метода `String.hashCode()` `cb=262143`

Количество элементов в ячейке	Количество ячеек
0	116 186
1	94 319
2	38 637
3	10 517
4	2 066
5	362
6	53
7	3

Если вы используете класс `LinkedList`, для каждого непустого элемента H потребуется 12 байт памяти, в предположении, что размер указателя равен четырем байтам. Каждый строковый элемент включен в `ListElement`, который требует дополнительных 12 байтов памяти. Для предыдущего примера из 213 557 слов потребуется 5 005 488 байтов памяти, помимо памяти для хранения фактических строк. Вот как распределена эта память:

- размер основной таблицы: 1 048 572 байта;
- размер 116 186 связанных списков: 1 394 232 байта;
- размер 213 557 элементов списка: 2 562 684 байта.

Хранение строк также влечет накладные расходы при использовании класса JDK `String`. Каждая строка имеет 12 байтов накладных расходов. Таким образом, к имеющимся накладным расходам добавляются еще 213 557·12 = 2 562 684 дополнительных байтов. Таким образом, алгоритму, выбранному в этом примере, требуется 7 568 172 байта памяти. Фактическое количество символов в списке слов, которые мы использовали в примере, составляет только 2 099 075, так что алгоритм требует примерно в 3,6 раза больше памяти, чем необходимо для хранения символов строк.

Большая часть этих накладных расходов является ценой использования классов JDK. Чтобы достичь компромисса, следует взвесить простоту и повторное использование классов и сравнить эти преимущества с более сложной реализацией, которая сокращает использование памяти. Когда расходы памяти играют важную роль, можно использовать один из описанных ниже вариантов оптимизации использования памяти. Если же у вас достаточно доступной памяти, имеется хеш-функция, не приводящая к большому количеству коллизий, и готовая реализация связанного списка, то решение с использованием JDK обычно вполне приемлемо.

Основное влияние на реализацию оказывает то, какой является коллекция — статической или динамической. В нашем примере список слов не изменяется и известен заранее. Однако если мы имеем динамическую коллекцию, которая требует много добавлений и удалений элементов, то мы должны выбрать для хеш-таблицы структуру данных, которая оптимизирует эти операции. Наша обработка коллизий в примере работает довольно хорошо, потому что вставка элемента в связанный список может быть выполнена за константное время, а удаление элемента пропорционально длине списка. Если хеш-функция равномерно распределяет элементы, отдельные списки оказываются относительно короткими.

Реализация алгоритма

В дополнение к хеш-функции решение для поиска на основе хеша содержит еще две части. Первая заключается в создании хеш-таблицы. Код в примере 5.6 показывает, как использовать связанные списки для размещения элементов, хешируемых в конкретную ячейку таблицы. Входные элементы из коллекции *C* извлекаются с помощью `Iterator`.

Пример 5.6. Загрузка хеш-таблицы

```
public void load(Iterator<V> it)
{
    table = (LinkedList<V>[]) new LinkedList[tableSize];

    // Выборка каждого элемента из итератора и поиск соответствующей
    // ячейки h. Добавление к существующему списку или создание нового,
    // в который добавляется значение.
```

```
    while (it.hasNext())
    {
        V v = it.next();
        int h = hashMethod.hash(v);

        if (table[h] == null)
        {
            table[h] = new LinkedList<V>();
        }

        table[h].add(v);
        count++;
    }
}
```

Обратите внимание, что `listTable` состоит из `tableSize` ячеек, каждая из которых имеет тип `LinkedList<V>` для хранения элементов.

Теперь поиск элементов в таблице становится тривиальным. Эту работу выполняет код из примера 5.7. После того как хеш-функция возвращает индекс в хеш-таблице, мы смотрим, пуста ли соответствующая ячейка таблицы. Если она пуста, возвращаем значение `false`, указывающее, что искомой строки нет в коллекции. В противном случае просматриваем связанный список для этой ячейки, чтобы определить наличие или отсутствие в нем искомой строки.

Пример 5.7. Поиск элемента

```
public boolean search(V v)
{
    int h = hashMethod.hash(v);
    LinkedList<V> list = (LinkedList<V>) listTable[h];

    if (list == null)
    {
        return false;
    }

    return list.contains(v);
}
int hash(V v)
{
    int h = v.hashCode();

    if (h < 0)
    {
        h = 0 - h;
    }

    return h % tableSize;
}
```

Обратите внимание, что функция `hash` гарантирует, что хеш-индекс находится в диапазоне [0, *tableSize*). При использовании функции `hashCode` для класса `String` хеш-функции должны учитывать возможность того, что целочисленные арифметические операции в `hashCode` приведут к переполнению и возврату отрицательного значения. Необходимо принять надлежащие меры, поскольку оператор остатка от деления (`%`) возвращает отрицательное число, если ему передано отрицательное значение (например, выражение Java `-5%3` равно `-2`). Так, например, метод `hashCode` JDK для объектов `String`, получив строку "*aaaaaa*", возвращает значение -1425372064.

Анализ алгоритма

Пока хеш-функция достаточно равномерно распределяет элементы коллекции, поиск, основанный на хеше, имеет отличную производительность. Среднее время, необходимое для поиска элемента, является константой, или $O(1)$. Поиск состоит из одного просмотра H с последующим линейным поиском в коротком списке коллизий. Компонентами поиска элемента в хеш-таблице являются:

- вычисление хеш-значения;
- доступ к элементу таблицы, индексированному хеш-значением;
- поиск требуемого элемента при наличии коллизий.

Все алгоритмы поиска на основе хеша используют первые два компонента; различное поведение может наблюдаться только в разных вариантах обработки коллизий.

Стоимость вычисления хеш-значения должна быть ограничена фиксированной, константной верхней границей. В нашем примере со списком слов вычисление хеш-значения пропорционально длине строки. Если T_k представляет собой время, необходимое для вычисления хеш-значения для самой длинной строки, то для вычисления любого хеш-значения потребуется время, не превосходящее T_k. Поэтому вычисление хеш-значения считается операцией с константным временем работы.

Вторая часть алгоритма также выполняется за константное время. Если таблица хранится во вторичной памяти, может быть ситуация, когда это время зависит от позиции элемента и времени, необходимого для позиционирования устройства, но и эти значения имеют константную верхнюю границу.

Если мы сможем показать, что третья часть вычислений также имеет константную верхнюю границу, то и общее время выполнения поиска на основе хеша является константным. Определим для хеш-таблицы *коэффициент загрузки* α как среднее количество элементов в связанном списке для некоторой ячейки $H[h]$. Точнее, $\alpha = n/b$, где b — количество ячеек в хеш-таблице, а n — количество элементов, хранящихся в ней. В табл. 5.5 показан фактический коэффициент загрузки в хеш-таблицах, которые мы создаем по мере увеличения b. Обратите внимание, что когда b является достаточно большим, с его ростом максимальная длина связанных списков элементов уменьшается,

в то время как количество контейнеров, содержащих единственный элемент, быстро увеличивается. В последнем ряду 81% элементов хешированы так, что оказываются единственными в своих ячейках, а среднее количество элементов в ячейке составляет только одну цифру. Вне зависимости от первоначального количества элементов можно выбрать достаточно большое значение b, чтобы гарантировать, что в каждой ячейке будет небольшое фиксированное количество элементов (в среднем). Это означает, что время поиска элементов в хеш-таблице не зависит от количества элементов в хеш-таблице (поиск имеет фиксированную стоимость), так что алгоритм поиска на основе хеша имеет амортизированную производительность $O(1)$.

Таблица 5.5. Статистические сведения о хеш-таблицах, созданных для рассматриваемого примера

b	Коэффициент загрузки α	Минимальная длина связанного списка	Максимальная длина связанного списка	Количество ячеек с одним элементом
4095	52,15	27	82	0
8191	26,07	9	46	0
16383	13,04	2	28	0
32767	6,52	0	19	349
65535	3,26	0	13	8190
131071	1,63	0	10	41858
262143	0,815	0	7	94319
524287	0,41	0	7	142530
1048575	0,20	0	5	173912

В табл. 5.6 сравнивается производительность кода из примера 5.7 с классом JDK `java.util.Hashtable` при работе с хеш-таблицами разных размеров. Выполнялись тесты с $p = 1{,}0$, когда каждое из 213557 слов использовалось в качестве искомого (тем самым гарантировалось наличие искомого элемента в хеш-таблице). Для тестов с $p = 0{,}0$ в каждом из этих слов последний символ был заменен символом "*", чтобы гарантировать отсутствие искомого слова в хеш-таблице. Обратите также внимание, что при этом сохраняется длина искомых слов, что обеспечивает идентичную стоимость вычисления хеша для этих двух вариантов поиска.

Каждый тест выполнялся 100 раз, наилучший и наихудший результаты отбрасывались. Среднее значение для оставшихся 98 испытаний приведено в табл. 5.6. Понять эти результаты вам помогут статистические характеристики создаваемых хеш-таблиц, которые приведены в табл. 5.5.

По мере уменьшения коэффициента загрузки средняя длина каждого списка элементов также уменьшается, что приводит к улучшению производительности. Действительно, для $b = 1045875$ ни один связанный список не содержит более пяти элементов. Так как хеш-таблица обычно может расти до размера, достаточно большого, чтобы все связанные списки элементов были небольшими, производительность

поиска считается равной $O(1)$. Однако, как всегда, это зависит от наличия достаточного объема памяти и подходящей хеш-функции, позволяющей равномерно распределить элементы по всем ячейкам хеш-таблицы.

Таблица 5.6. Время поиска (в мс) для разных размеров хеш-таблиц

b	Хеш-таблица из примера 5.7		`java.util.Hashtable` с емкостью по умолчанию	
	p = 1,0	*p* = 0,0	*p* = 1,0	*p* = 0,0
4095	200,53	373,24	82,11	38,37
8191	140,47	234,68	71,45	28,44
16383	109,61	160,48	70,64	28,66
32767	91,56	112,89	71,15	28,33
65535	80,96	84,38	70,36	28,33
131071	75,47	60,17	71,24	28,28
262143	74,09	43,18	70,06	28,43
524287	75,00	33,05	69,33	28,26
1048575	76,68	29,02	72,24	27,37

Производительность готового класса `java.util.Hashtable` превосходит производительность кода из нашего примера, но это превосходство уменьшается по мере увеличения размера хеш-таблицы. Дело в том, что `java.util.Hashtable` содержит оптимизированные классы списков, которые эффективно управляют цепочками элементов. Кроме того, `java.util.Hashtable` автоматически "перехеширует" всю хеш-таблицу, когда коэффициент загрузки становится слишком высоким; эта стратегия рассматривается в разделе "Вариации алгоритма" далее в этой главе. Такая реализация увеличивает стоимость создания хеш-таблицы, но улучшает производительность поиска. Если отключить возможность "перехеширования", производительность поиска в `java.util.Hashtable` становится почти такой же, как и в нашей реализации.

В табл. 5.7 показано, сколько раз выполняется перехеширование при создании хеш-таблицы `java.util.Hashtable`, и общее время построения хеш-таблицы (в миллисекундах). Мы строили хеш-таблицы из упоминавшегося ранее списка слов; после выполнения 100 испытаний наилучший и наихудший результаты отбрасывались; в таблице содержится среднее значение оставшихся 98 испытаний. Класс `java.util.Hashtable` выполняет дополнительные вычисления при создании хеш-таблицы для повышения производительности поиска (распространенный компромисс). В столбцах 3 и 5 табл. 5.7 показано, что перехеширование приводит к заметным расходам. Обратите также внимание на то, что в двух последних строках перестройка хеш-таблиц не выполняется, поэтому результаты в столбцах 3, 5 и 7 оказываются почти идентичными. Выполнение перехеширования при построении хеш-таблиц приводит к повышению общей производительности за счет уменьшения средней длины цепочек элементов.

Таблица 5.7. Сравнение времени построения хеш-таблиц (в мс)

b	Наша хеш-таблица	Хеш-таблица JDK ($\alpha = 0{,}75$)		Хеш-таблица JDK ($\alpha = 4{,}0$)		Хеш-таблица JDK ($\alpha = n/b$) без перехеширования
	Время построения	Время построения	Количество перехешированний	Время построения	Количество перехешированний	Время построения
4095	403,61	42,44	7	35,30	4	104,41
8191	222,39	41,96	6	35,49	3	70,74
16383	135,26	41,99	5	34,90	2	50,90
32767	92,80	41,28	4	33,62	1	36,34
65535	66,74	41,34	3	29,16	0	28,82
131071	47,53	39,61	2	23,60	0	22,91
262143	36,27	36,06	1	21,14	0	21,06
524287	31,60	21,37	0	22,51	0	22,37
1048575	31,67	25,46	0	26,91	0	27,12

Вариации алгоритма

Один популярный вариант поиска на основе хеша модифицирует обработку коллизий путем наложения ограничения, заключающегося в том, что каждая ячейка содержит один элемент. Вместо создания связанного списка для хранения всех элементов, которые хешируются в одну и ту же ячейку в хеш-таблице, используется метод *открытой адресации*, который хранит элементы, вызвавшие коллизию, *в некоторых других пустых ячейках хеш-таблицы H*. Этот подход показан на рис. 5.2. При использовании открытой адресации хеш-таблица снижает накладные расходы на хранение элементов путем устранения всех связанных списков.

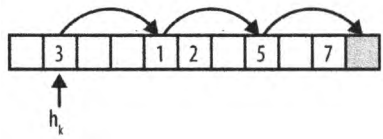


Рис. 5.2. Открытая адресация

Чтобы вставить элемент с использованием открытой адресации, вычисляется ячейка $h_k = hash(e)$, которая должна содержать элемент e. Если ячейка $H[h_k]$ пуста, ей присваивается значение элемента $H[h_k] = e$ так же, как и в стандартном алгоритме. В противном случае выполняется *исследование* (probe) ячеек H с использованием той или иной стратегии исследования и элемент e помещается в первую обнаруженную пустую ячейку.

Линейное исследование

Выполняется неоднократный поиск в ячейках $h_k = (h_k + c \cdot i)\%b$, где c представляет собой целочисленное смещение, а i — количество последовательных исследований H; зачастую $c = 1$. При использовании этой стратегии в H могут появиться кластеры элементов.

Квадратичное исследование

Выполняется неоднократный поиск в ячейках $h_k = (h_k + c_1 \cdot i + c_2 \cdot i^2)\%b$, где c_1 и c_2 — константы. Этот подход обычно позволяет избежать кластеризации. Полезными для практического применения являются значения $c_1 = c_2 = 1/2$.

Двойное хеширование

Сходно с линейным исследованием, но в отличие от него c не является константой, а определяется второй хеш-функцией. Дополнительные вычисления обеспечивают снижение вероятности кластеризации.

Во всех случаях, если пустая ячейка после b проб не обнаружена, вставка является неудачной.

На рис. 5.2 показан пример хеш-таблицы с $b = 11$ контейнерами с использованием линейного исследования с $c = 3$. Коэффициент загрузки для хеш-таблицы равен $\alpha = 0{,}45$, поскольку она содержит пять элементов. На рисунке показано поведение при попытке добавить элемент e, который хешируется в $h_k = 1$. Ячейка $H[1]$ уже занята (значением 3), поэтому выполняется исследование других потенциальных ячеек. После $i = 3$ итераций находится пустая ячейка $H[10]$, в которую и вставляется e.

В предположении, что выполняется просмотр не более чем b потенциальных ячеек, наихудшее время вставки представляет собой $O(b)$, но при достаточно низком коэффициенте загрузки и отсутствии кластеризации обычно требуется только фиксированное количество проверок. На рис. 5.3 показано ожидаемое количество проб при поиске свободной ячейки [34].

$$\frac{1 + \left(\frac{1}{1-\alpha}\right)}{2} \qquad \frac{1 + \left(\frac{1}{1-\alpha}\right)^2}{2}$$

Успешный поиск Неудачный поиск

Рис. 5.3. Ожидаемое количество проверок при поиске свободной ячейки

В хеш-таблице с использованием открытой адресации проблемой являются удаления элементов. Предположим, что на рис. 5.2 хеш-значения 3, 1 и 5 имеют $h_k = 1$ и что они были вставлены в хеш-таблицу в указанном порядке. Поиск значения 5 будет успешным, потому что вы выполните три пробы в хеш-таблице и в конечном итоге

найдете его в ячейке $H[7]$. Если вы удалите значение 1 из хеш-таблицы и очистите ячейку $H[4]$, вы больше не сможете найти значение 5, поскольку поиск прекращается по достижении первой пустой ячейки. Для поддержки удаления при использовании открытой адресации необходимо пометить ячейку как удаленную и соответственно откорректировать функции поиска.

Код с использованием открытой адресации показан в примере 5.8. Класс предполагает, что пользователь предоставляет хеш-функцию, генерирующую допустимый индекс из диапазона $[0, b)$, и функцию поиска для открытой адресации. Альтернативные реализации предоставляются по умолчанию с использованием встроенного метода Python `hash`. Приведенная реализация позволяет удалять элементы и поддерживает список `deleted`, чтобы гарантировать отсутствие разрывов цепочек открытой адресации. Приведенная далее реализация следует семантике *множества* для коллекции и не позволяет иметь в хеш-таблице несколько одинаковых элементов.

Пример 5.8. Реализация хеш-таблицы с открытой адресацией на языке Python

```
class Hashtable:
    def __init__(self, b=1009, hashFunction=None, probeFunction=None):
    """Инициализация хеш-таблицы с b ячейками, заданной хеш-функцией
       и функцией проверки ячеек."""
    self.b = b
    self.bins = [None] * b
    self.deleted = [False] * b

    if hashFunction:
        self.hashFunction = hashFunction
    else:
        self.hashFunction = lambda value, size: hash(value) % size

    if probeFunction:
        self.probeFunction = probeFunction
    else:
        self.probeFunction = lambda hk, size, i : (hk + 37) % size

def add (self, value):
    """
    Добавляет элемент в хеш-таблицу, возвращая -self.b в случае
    неудачи после self.b попыток. При успехе возвращает
    количество проб.

    Выполняет добавление в ячейки, помеченные как удаленные,
    и корректно обрабатывает ранее удаленные записи.
    """
    hk = self.hashFunction (value, self.b)
```

```
    ctr = 1
    while ctr <= self.b:
        if self.bins[hk] is None or self.deleted[hk]:
            self.bins[hk] = value
            self.deleted[hk] = False
            return ctr

        # Уже имеется в коллекции?
        if self.bins[hk] == value and not self.deleted[hk]:
            return ctr

        hk = self.probeFunction (hk, self.b, ctr)
        ctr += 1

    return -self.b
```

Код в примере 5.8 показывает, как открытая адресация добавляет элементы в пустые ячейки или в ячейки, помеченные как удаленные. Она поддерживает счетчик, который обеспечивает производительность $O(b)$ в наихудшем случае. Вызывающая функция может определить, что вызов `add` был успешным, если функция возвращает положительное число. Если функции `probeFunction` не удается найти пустой контейнер за b попыток, то возвращается отрицательное число. Код `delete` в примере 5.9 практически идентичен коду, проверяющему, содержит ли хеш-таблица значение; в частности, в методе `contains` (который здесь не воспроизводится) пропущен код, который выполняет пометку `self.deleted[hk]=True`. Обратите внимание, как этот код использует `probeFunction` для определения следующей испытываемой ячейки.

Пример 5.9. Метод `delete` при открытой адресации

```
def delete (self, value):
    """Удаление значения из хеш-таблицы без
       нарушения имеющихся цепочек."""
    hk = self.hashFunction (value, self.b)

    ctr = 1
    while ctr <= self.b:
        if self.bins[hk] is None:
            return -ctr

        if self.bins[hk] == value and not self.deleted[hk]:
            self.deleted[hk] = True
            return ctr
        hk = self.probeFunction (hk, self.b, ctr)
        ctr += 1

    return -self.b
```

Изучим производительность открытой адресации, рассматривая количество проб для поиска элемента в хеш-таблице. На рис. 5.4 показаны результаты успешных и неудачных поисков с использованием такого же списка из 213 557 слов, что и раньше. С увеличением количества ячеек — от 224 234 (95,2% заполненных) до 639 757 (33,4% заполненных) — вы можете увидеть, как резко снижается количество необходимых проб. В верхней части рисунка показаны средние (и наихудшие) количества проб для успешного поиска, а в нижней — та же информация для неудачного поиска. Короче говоря, открытая адресация уменьшает общее количество используемой памяти, но требует существенно большего количества проб в наихудшем случае. Второе следствие заключается в том, что линейное исследование приводит к большему количеству проб из-за кластеризации.

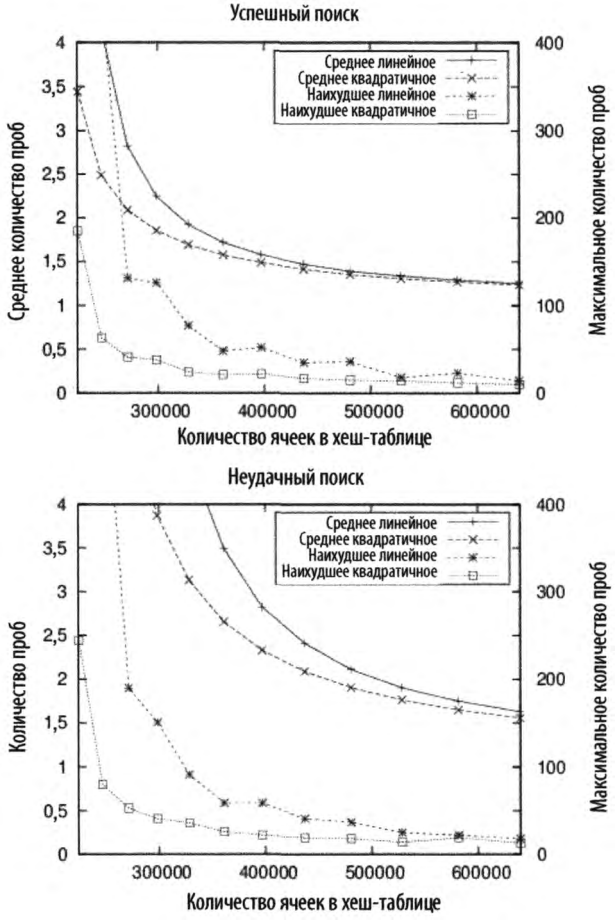


Рис. 5.4. Производительность открытой адресации

Вычисленный коэффициент загрузки для хеш-таблицы описывает ожидаемую производительность поиска и вставки. Если коэффициент загрузки слишком высок, количество проб при поиске элемента становится чрезмерным, будь то в связанном списке ячейки или в цепочке ячеек при использовании открытой адресации. Хеш-таблица может увеличить количество ячеек и перестроиться с помощью процесса, известного как "перехеширование" — нечастой операции, которая уменьшает коэффициент загрузки, хотя и является дорогостоящей операцией, — со временем работы $O(n)$. Типичный способ изменения размера — удвоить количество имеющихся ячеек и добавить еще одну (так как хеш-таблицы обычно содержат нечетное количество ячеек). Когда становится доступным большее количество ячеек, все существующие элементы в хеш-таблице перехешируются и вносятся в новую структуру. Эта дорогостоящая операция снижает общую стоимость будущих поисков, но должна выполняться как можно реже; в противном случае вы не получите амортизированную производительность хеш-таблиц, равную $O(1)$.

Хеш-таблица должна перехешировать свое содержимое, когда обнаруживает неравномерное распределение элементов. Это можно сделать путем установки порогового значения, по достижении коэффициентом загрузки которого выполняется перехеширование. Пороговое значение коэффициента загрузки для класса `java.util.Hashtable` по умолчанию равно 0,75.

В приведенных выше примерах для хеш-таблицы использовался фиксированный набор строк. В этом частном случае можно достичь оптимальной производительности с помощью *идеального хеширования*, которое использует две хеш-функции. Стандартная функция *hash()* выполняет индексацию в главной таблице H. Каждая ячейка $H[i]$ указывает на меньшую по размеру вторичную хеш-таблицу S_i, с которой связанна хеш- функция $hash_i$. Если в ячейку $H[i]$ хешировано k ключей, то таблица S_i будет содержать k^2 ячеек. Это кажется пустой тратой большого количества памяти, но разумный выбор первоначальной хеш-функции может уменьшить эти затраты до величин, аналогичных затратам в предыдущих вариантах хеширования. Выбор соответствующих хеш-функций гарантирует, что во вторичных таблицах коллизии будут отсутствовать. Это означает, что мы получим алгоритм с константной производительностью $O(1)$ для каждого конкретного значения.

Детальный анализ идеального хеширования можно найти в [20]. Дуг Шмидт Doug Schmidt написал прекрасную статью [55] о генерации функций для идеального хеширования, так что теперь для различных языков программирования свободно доступны генераторы функций идеального хеширования. Такой инструмент GPERF для языков программирования C и C++ можно найти по адресу `http://www.gnu.org/software/gperf`, а JPERF для Java можно найти по адресу `http://www.anarres.org/projects/jperf`.

Фильтр Блума

Поиск на основе хеша сохраняет полный набор значений из коллекции *C* в хеш-таблице *H*, в связанных ли списках или с использованием открытой адресации. В обоих случаях чем больше элементов добавляется в хеш-таблицу, тем больше время поиска элемента, если только вы не увеличиваете при этом размер таблицы (в данном случае — количество ячеек). В действительности это поведение ожидается от всех других алгоритмов в этой главе, и мы просто ищем разумный компромисс в отношении количества памяти, необходимого для сокращения числа сравнений при поиске элемента в коллекции.

Фильтр Блума предоставляет альтернативную структуру *битового массива B*, которая обеспечивает *константную производительность* при добавлении элементов из *C* в *B* или проверку того, что элемент не был добавлен в *B*. Как ни удивительно, это поведение не зависит от количества элементов, уже добавленных в *B*. Однако имеется и ловушка: при проверке, находится ли некоторый элемент в *B*, фильтр Блума может возвратить *ложный положительный результат*, даже если на самом деле элемент в *C* отсутствует. Фильтр Блума позволяет точно определить, что элемент не был добавлен в *B*, поэтому никогда не возвращает *ложный отрицательный результат*.

На рис. 5.5 в битовый массив *B* вставлены два значения, *u* и *v*. Таблица в верхней части рисунка показывает позиции битов, вычисленные $k = 3$ хеш-функциями. Как можно видеть, фильтр Блума в состоянии быстро определить, что третье значение *w* не было вставлено в *B*, поскольку одно из его *k* вычисляемых значений битов равно нулю (в данном случае это бит 6). Однако для значения *x* фильтр возвращает ложный положительный результат, поскольку, несмотря на то что это значение не было вставлено в *B*, все *k* вычисляемых значений битов равны единице.

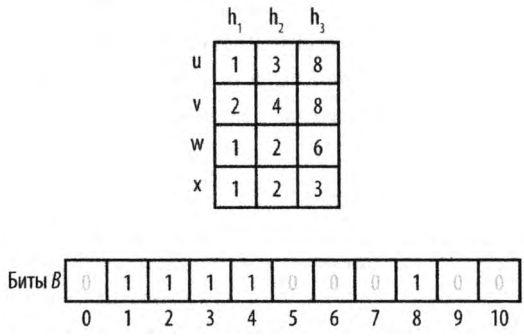


Рис. 5.5. Пример работы фильтра Блума

Фильтр Блума

Наилучший, средний и наихудший случаи: $O(k)$

```
create(m)
    return Битовый массив из m битов          ❶
end

add (bits,value)
    foreach hashFunction hf                   ❷
        setbit = 1 << hf(value)               ❸
        bits |= setbit                        ❹
end

search (bits,value)
    foreach hashFunction hf
        checkbit = 1 << hf(value)
        if checkbit | bits = 0 then           ❺
            return false

    return true                               ❻
end
```

❶ Память для m битов выделяется заранее.
❷ Имеется k хеш-функций, которые вычисляют (потенциально) различные битовые позиции.
❸ Оператор левого сдвига << эффективно вычисляет $2^{hf(value)}$.
❹ Установка k битов при вставке значения.
❺ Если при поиске значения выясняется, что вычисленный бит — нулевой, то значения в наличии быть не может.
❻ Возможен ложный положительный результат, когда все биты установлены, но значение добавлено не было.

Входные и выходные данные алгоритма

Фильтр Блума обрабатывает значения во многом так же, как и поиск на основе хеша. Алгоритм начинает работу с массива из m битов, каждый из которых изначально равен нулю. Имеется k хеш-функций, вычисляющих при вставке значений (потенциально различные) позиции битов в этом массиве.

Фильтр Блума возвращает значение `false`, если может доказать, что целевой элемент t еще не был вставлен в битовый массив и, соответственно, отсутствует в коллекции C. Алгоритм может вернуть значение `true`, которое может быть *ложным положительным результатом*, если искомый элемент t не был вставлен в массив.

Контекст применения алгоритма

Фильтр Блума демонстрирует эффективное использование памяти, но он полезен только тогда, когда допустимы ложные положительные результаты. Используйте фильтр Блума для снижения количества дорогостоящих поисков путем отфильтровывания тех, которые гарантированно обречены на провал, например для подтверждения проведения дорогостоящего поиска на диске.

Реализация алгоритма

Фильтру Блума требуются m битов памяти. Реализация в примере 5.10 использует возможность Python работать со сколь угодно большими значениями.

Пример 5.10. Реализация фильтра Блума на языке программирования Python

```
class bloomFilter:
    def __init__(self, size = 1000, hashFunctions=None):
        """
        Строит фильтр Блума с size битами (по умолчанию — 1000) и
        соответствующими хеш-функциями.
        """
        self.bits = 0
        self.size = size
        if hashFunctions is None:
            self.k = 1
            self.hashFunctions = [lambda e, size : hash(e) % size]
        else:
            self.k = len(hashFunctions)
            self.hashFunctions = hashFunctions

    def add (self, value):
        """Вставляет value в фильтр Блума."""
        for hf in self.hashFunctions:
            self.bits |= 1 << hf (value, self.size)

    def __contains__ (self, value):
        """
        Определяет, имеется ли value в фильтре. Может быть
        возвращен ложный положительный результат, даже если
        элемента нет. Однако ложный отрицательный результат
        никогда не возвращается (если элемент имеется в
        наличии, функция будет возвращать True).
        """
        for hf in self.hashFunctions:
            if self.bits & 1 << hf (value, self.size) == 0:
                return False

        # Элемент может быть в наличии
        return True
```

Данная реализация предполагает наличие k хеш-функций, каждая из которых принимает вставляемое значение и размер битового массива. Всякий раз, когда добавляется значение, в битовом массиве устанавливаются k битов, позиции которых вычисляются с помощью хеш-функций. Этот код использует оператор побитового сдвига << для того, чтобы выполнить сдвиг 1 в соответствующую позицию, и побитовый оператор ИЛИ (|), чтобы установить этот бит равным 1. Чтобы определить, было ли добавлено некоторое значение, выполняется проверка тех же k битов, указываемых теми же хеш-функциями, с помощью побитового оператора И (&). Если хотя бы один из битов имеет значение 0, значит, искомое значение не было добавлено в массив, и возвращается значение `False`. Однако, если все k битов имеют значение 1, можно только утверждать, что искомое значение *могло* быть добавлено в массив.

Анализ алгоритма

Общее количество памяти, необходимое для работы фильтра Блума, фиксировано и составляет m битов, и оно не увеличивается независимо от количества хранящихся значений. Кроме того, алгоритм требует только фиксированного количества испытаний k, поэтому каждая вставка и поиск могут быть выполнены за время $O(k)$, которое рассматривается как константное. Именно по этим причинам фильтр Блума противопоставляется всем прочим алгоритмам в этой главе. Основная сложность фильтра заключается в разработке эффективных хеш-функций, которые обеспечивают действительно равномерное распределение вычисляемых битов для вставляемых значений. Хотя размер битового массива является константой, для снижения количества ложных положительных срабатываний он может быть достаточно большим. И наконец, нет никакой возможности удаления элемента из фильтра, поскольку потенциально это может фатально нарушить обработку других значений.

Единственная причина для использования фильтра Блума заключается в том, что он имеет предсказуемую вероятность ложного положительного срабатывания p^k в предположении, что k хеш-функций дают равномерно распределенные случайные величины [14]. Достаточно точной оценкой p^k является следующая:

$$p^k = \frac{\left(1-\left(1-\frac{1}{m}\right)^{kn}\right)^k}{2},$$

где n — количество добавленных в фильтр значений [16]. Мы эмпирически оценили частоту ложных положительных результатов следующим образом.

1. Случайным образом из полного списка из 213557 слов были удалены 2135 слов (1% от полного списка), и оставшиеся 211422 слова были вставлены в фильтр Блума.

2. Подсчитано количество ложных положительных результатов при поиске отсутствующих в фильтре 2 135 слов.
3. Подсчитано количество ложных положительных результатов при поиске 2 135 случайных строк (длиной от 2 до 10 строчных букв).

Мы провели испытания для различных значений m — от 100 000 до 2 000 000 (с шагом 10 000). Использовались $k = 3$ хеш-функции. Результаты исследований показаны на рис. 5.6.

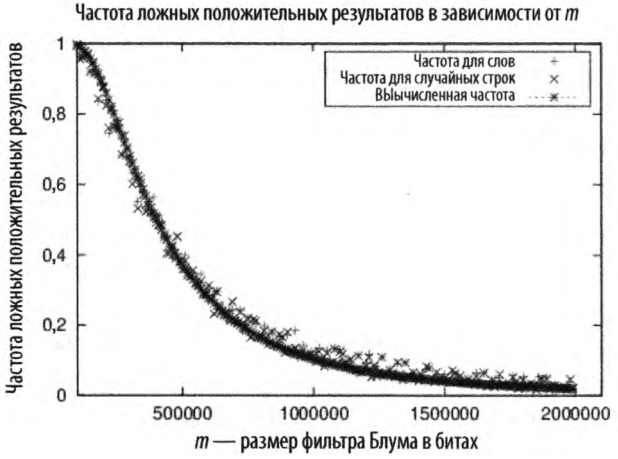


Рис. 5.6. Пример работы фильтра Блума

Если вы заранее хотите, чтобы ложные положительные срабатывания не превышали некоторого небольшого значения, выберите k и m после оценки числа n вставляемых элементов. В литературе предлагается обеспечить значение $1 - (1 - 1/m)^{kn}$, близкое к 1/2. Например, чтобы гарантировать, что ложные положительные срабатывания будут для нашего списка слов менее 10%, установите значение m равным по крайней 1 120 000 битам (что в общей сложности равно 131 250 байтам).

Бинарное дерево поиска

Бинарный поиск в массиве в памяти, как мы уже видели, достаточно эффективен. Однако использование упорядоченных массивов становится гораздо менее эффективным при частых изменениях базовой коллекции. При работе с динамической коллекцией для поддержания приемлемой производительности поиска мы должны использовать иную структуру данных. Поиск на основе хеша может работать с динамическими коллекциями, но с целью эффективного использования ресурсов мы можем выбрать размер хеш-таблицы, который окажется слишком мал; зачастую у нас нет *априорного* знания о количестве элементов, которые будут храниться в таблице,

так что выбрать правильный размер хеш-таблицы оказывается очень трудной задачей. Кроме того, хеш-таблицы не позволяют обходить все элементы в порядке сортировки.

Альтернативной стратегией является использование *дерева поиска* для хранения динамических множеств данных. Деревья поиска хорошо работают как с первичной, так и со вторичной памятью и позволяют возвращать упорядоченные диапазоны элементов, на что не способны хеш-таблицы. Наиболее распространенным типом дерева поиска является *бинарное дерево поиска* (binary search tree — BST), которое состоит из узлов, как показано на рис. 5.7. Каждый узел содержит одно значение из множества и хранит ссылки на два (потенциальных) дочерних узла — левый (*left*) и правый (*right*).

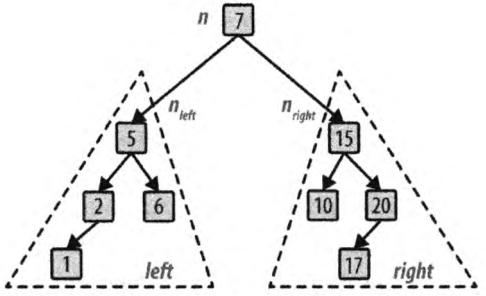


Рис. 5.7. Простое бинарное дерево поиска

Используйте бинарное дерево поиска, когда

- требуется обход данных в возрастающем (или убывающем) порядке;
- размер множества данных неизвестен, а реализация должна быть в состоянии обработать любой возможный размер, помещающийся в памяти;
- множество данных — динамичное, с большим количеством вставок и удалений в процессе работы программы.

Входные и выходные данные алгоритма

Входные и выходные данные для алгоритма с использованием деревьев поиска такие же, как и для бинарного поиска. Каждый элемент e из коллекции C, сохраняемый в бинарном дереве поиска, должен иметь одно или несколько свойств, которые могут использоваться в качестве ключа k; эти ключи определяют универсум U и должны быть вполне упорядочиваемы (т.е. для любых двух заданных ключей со значениями k_i и k_j, либо k_i равен k_j либо $k_i > k_j$, либо $k_i < k_j$.

Когда значения в коллекциях являются примитивными типами (например, строками или целыми числами), ключами могут быть сами значения. В противном случае они являются ссылками на структуры, которые содержат значения.

Контекст применения алгоритма

Бинарное дерево поиска представляет собой непустой набор узлов, содержащих упорядоченные значения, именуемые *ключами*. Верхний *корневой* узел является предком для всех остальных узлов в BST. Каждый узел *n* потенциально может указывать на два узла, n_{left} и n_{right}, каждый из которых является корнем BST — *left* и *right* соответственно. BST гарантирует выполнение *свойства бинарного дерева поиска*, а именно — того, что если *k* является ключом для узла *n*, то все ключи в поддереве *left* не превышают значения *k*, а все ключи в поддереве *right* не меньше *k*. Если n_{left} и n_{right} имеют нулевые значения, то *n* является *листом* дерева. На рис. 5.7 показан небольшой пример BST, в котором каждый узел содержит целочисленное значение. Корень содержит значение 7; в дереве есть четыре листа со значениями 1, 6, 10 и 17. Внутренний узел, например 5, имеет родительский узел и некоторые дочерние узлы. Как видно из рисунка, поиск ключа в этом дереве требует изучения не более четырех узлов, начиная с корня.

BST может не быть *сбалансированным*; при добавлении элементов одни ветви могут оказаться относительно короткими, в то время как другие — более длинными. Это приводит к субоптимальному поиску по более длинным ветвям. В худшем случае структура BST может *выродиться* и превратиться в список. На рис. 5.8 показано вырожденное дерево с теми же значениями, что и дерево на рис. 5.7. Хотя структура такого дерева и соответствует строгому определению BST, фактически она представляет собой связанный список, так как правое поддерево каждого узла является пустым.

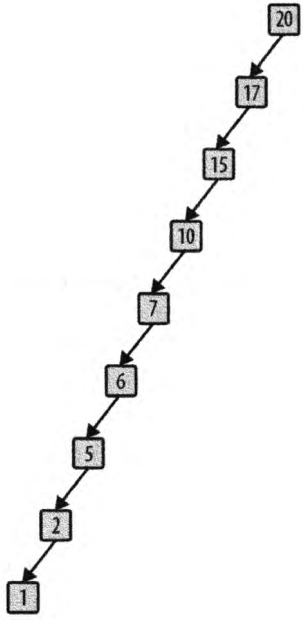


Рис. 5.8. Вырожденное бинарное дерево поиска

Необходимо сбалансировать дерево, чтобы избежать появления вырожденного перекошенного дерева, которое имеет несколько ветвей, гораздо более длинных, чем другие. Мы представим далее решение для сбалансированного AVL-дерева, поддерживающего возможность вставки и удаления значений.

Реализация алгоритма

Изначальная структура Python показана в примере 5.11 вместе с методами add, необходимыми для добавления значений в BST. Методы являются рекурсивными, проходящими вниз по ветвям от корня до пустого места для вставки нового узла в корректном местоположении, обеспечивающем сохранение свойства бинарного дерева поиска.

Пример 5.11. Определение класса бинарного дерева поиска на языке программирования Python

```
class BinaryNode:
    def __init__(self, value = None):
        """Создание узла бинарного дерева поиска."""
        self.value = value
        self.left = None
        self.right = None

    def add (self, val):
        """Добавление нового узла с заданным значением в BST."""
        if val <= self.value:
            if self.left:
                self.left.add (val)
            else:
                self.left = BinaryNode(val)
        else:
            if self.right:
                self.right.add(val)
            else:
                self.right = BinaryNode(val)

class BinaryTree:
    def __init__(self):
        """Создание пустого BST."""
        self.root = None

    def add (self, value):
        """Вставка значения в корректное место в BST."""
        if self.root is None:
            self.root = BinaryNode(value)
        else:
            self.root.add (value)
```

После добавления значения в пустое BST создается корневой узел; затем вставляемые значения помещаются в новые объекты `BinaryNode` в соответствующих местах в BST. В бинарном дереве поиска может быть несколько элементов с одинаковыми *значениями*, но, если вы хотите ограничить дерево семантикой множества (как определено в Java Collections Framework), измените код так, чтобы он предотвращал вставку повторяющихся ключей. Пока что предположим, что в BST могут быть повторяющиеся ключи.

Имея такую структуру, нерекурсивный метод `contains(value)` класса `BinaryTree`, показанный в примере 5.12, ищет `value` в BST. Вместо выполнения рекурсивного вызова функции он просто идет по указателям `left` или `right` до тех пор, пока не находит искомое значение или не определяет, что такого значения в BST нет.

Пример 5.12. Метод `contains` класса `BinaryTree`

```
def __contains__ (self, target):
    """Проверка, содержит ли BST искомое значение."""
    node = self.root
    while node:
        if target < node.value:
            node = node.left
        ielif target > node.value:
            node = node.right
        else:
            node = node.right
    return true
```

Эффективность этой реализации зависит от *сбалансированности* BST. Для сбалансированного дерева размер множества, в котором выполняется поиск, уменьшается в два раза при каждой итерации цикла `while`, что приводит к производительности $O(log\ n)$. Однако для вырожденных бинарных деревьев, таких, как показанное на рис. 5.8, производительность равна $O(n)$. Для сохранения оптимальной производительности необходимо балансировать BST после каждого добавления (и удаления).

AVL-деревья (названные так в честь их изобретателей, Адельсона-Вельского и Ландиса), изобретенные в 1962 году, были первыми *самобалансирующимися* BST. Давайте определим понятие *высоты* AVL-узла. Высота листа равна 0, потому что он не имеет дочерних элементов. Высота узла, не являющегося листом, на 1 больше максимального значения высоты двух его дочерних узлов. Для обеспечения согласованности высота несуществующего дочернего узла считается равной –1.

AVL-дерево гарантирует *AVL-свойство* для каждого узла, заключающееся в том, что *разница высот* для любого узла равна –1, 0 или 1. Разница высот определяется как $height(left) - height(right)$, т.е. высота левого поддерева минус высота правого поддерева. AVL-дерево должно обеспечивать выполнение этого свойства всякий раз при вставке значения в дерево или удалении из него. Для этого необходимы два вспомогательных метода: `computeHeight` для вычисления высоты узла и `heightDifference` для вычисления разницы высот. Каждый узел в дереве AVL хранит свое значение высоты *Height*, что увеличивает общие требования к памяти.

На рис. 5.9 показано, что происходит при вставке значений 50, 30 и 10 в дерево в указанном порядке. Как видите, получаемое дерево не удовлетворяет AVL-свойству, так как высота корня левого поддерева равна 1, а высота его несуществующего поддерева справа равна –1, так что разница высот равна 2. Представим, что мы "захватили" узел 30 исходного дерева и повернули дерево вправо (или по часовой стрелке) вокруг узла 30 так, чтобы сделать узел 30 корнем, создав тем самым сбалансированное дерево (показанное на рис. 5.10). При этом изменяется только высота узла 50 (снижается от 2 до 0) и AVL-свойство дерева восстанавливается.

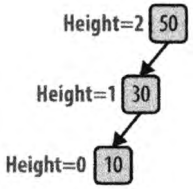


Рис. 5.9. Несбалансированное AVL-дерево

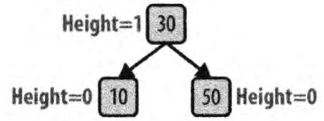


Рис. 5.10. Сбалансированное AVL-дерево

Операция *поворота вправо* изменяет структуру поддерева в несбалансированном BST; как вы можете представить, имеется и аналогичная операция *поворота влево*.

Но что если это дерево имеет и другие узлы, каждый из которых был сбалансирован и удовлетворял AVL-свойству? На рис. 5.11 каждый из заштрихованных треугольников представляет собой потенциальное поддерево исходного дерева. Каждое из них помечено в соответствии с его позицией, так что поддерево **30R** представляет собой правое поддерево узла **30**. Корень является единственным узлом, который не поддерживает *AVL-свойство*. Различные показанные на рисунке высоты в дереве вычислены в предположении, что узел **10** имеет некоторую высоту *k*.

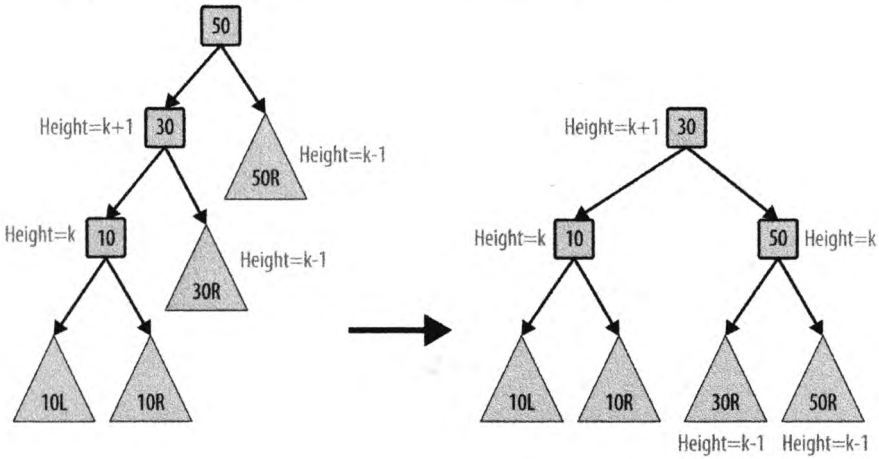


Рис. 5.11. Сбалансированное AVL-дерево с поддеревьями

Модифицированное бинарное дерево поиска по-прежнему гарантирует *свойство бинарного поиска*. Все значения ключей в поддереве **30R** больше или равны 30. Позиции других поддеревьев одно относительно другого не изменились, так что свойство бинарного поиска остается выполненным. Наконец значение 30 меньше значения 50, так что новый корневой узел вполне корректен.

Рассмотрите добавление трех значений в пустое AVL-дерево. На рис. 5.12 показаны четыре различных порядка вставки, которые приводят к несбалансированному дереву. В случае **Left-Left** для балансировки дерева следует выполнить операцию поворота вправо. Аналогично в случае **Right-Right** необходимо выполнить операцию поворота влево. Однако в случае **Left-Right** невозможно просто выполнить поворот вправо, потому что "средний" узел, 10, не может стать корнем дерева; его значение меньше обоих других значений.

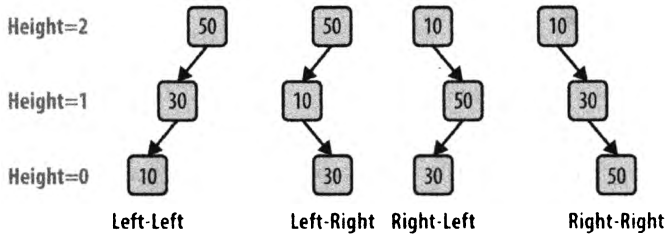


Рис. 5.12. Четыре сценария несбалансированности

К счастью, проблему можно решить, выполнив сначала поворот влево для дочернего узла 10 — дерево при этом будет приведено к виду **Left-Left**. Затем можно будет выполнить поворот вправо, описанный выше. На рис. 5.13 эта ситуация показана для большего дерева. После операции поворота влево дерево становится идентичным более раннему дереву, для которого была описана операция поворота вправо. Аналогичные рассуждения поясняют, как следует поступить в ситуации **Right-Left**.

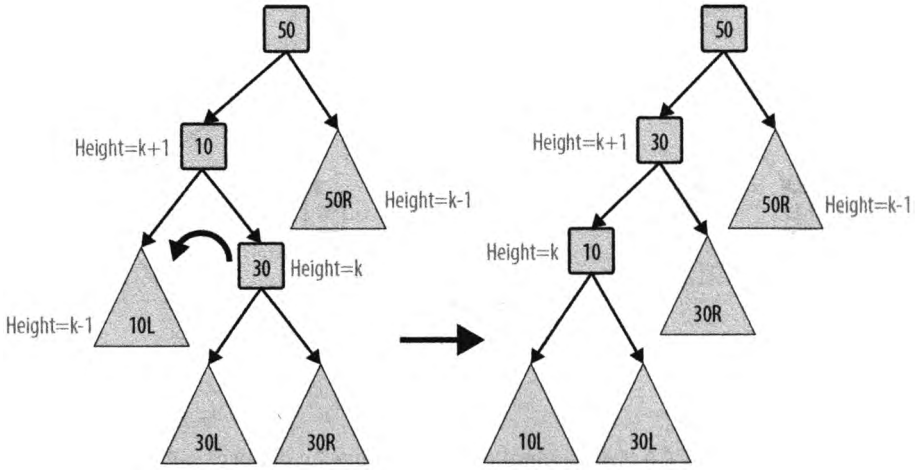


Рис. 5.13. Балансировка сценария Left-Right

Рекурсивная операция add, показанная в примере 5.13, имеет такую же структуру, как и в примере 5.11, и разница только в том, что после вставки нового листа может потребоваться перебалансировка дерева. Операция add класса BinaryNode добавляет новое значение в BST с корнем в указанном узле и возвращает объект BinaryNode, *который должен стать новым корнем этого BST*. Это происходит потому, что операция поворота перемещает новый узел в корень этого BST. Поскольку бинарные деревья поиска являются рекурсивными структурами, вы должны понимать, что поворот может произойти в любой момент. По этой причине рекурсивный вызов внутри add имеет вид `self.left=self.addToSubTree(self.left,val)`. После добавления val в поддерево с корнем в `self.left` это поддерево может потребоваться перебалансировать так, чтобы его корнем стал новый узел, и этот новый узел должен стать левым дочерним узлом для self. В заключение add вычисляет его высоту, чтобы позволить рекурсии распространиться обратно до исходного корня дерева.

Пример 5.13. Методы add классов BinaryTree и BinaryNode

```
class BinaryTree:

    def add (self, value):
    """Вставка value в корректное местоположение дерева."""
    if self.root is None:
        self.root = BinaryNode(value)
    else:
        self.root = self.root.add (value)

class BinaryNode:
    def __init__(self, value = None):
        """Создание узла."""
        self.value = value
        self.left = None
        self.right = None
        self.height = 0

    def computeHeight (self):
        """Вычисление высоты узла в BST из высот потомков."""
        height = -1
        if self.left:
            height = max(height, self.left.height)
        if self.right:
            height = max(height, self.right.height)

        self.height = height + 1

    def heightDifference(self):
        """Вычисление разности высот потомков в BST."""
        leftTarget = 0
        rightTarget = 0
```

```
        if self.left:
            leftTarget = 1 + self.left.height
        if self.right:
            rightTarget = 1 + self.right.height
        return leftTarget - rightTarget

    def add (self, val):
        """Добавление нового узла с value в BST
           и при необходимости - перебалансировка."""
        newRoot = self
        if val <= self.value:
            self.left = self.addToSubTree (self.left, val)
            if self.heightDifference() == 2:
                if val <= self.left.value:
                    newRoot = self.rotateRight()
                else:
                    newRoot = self.rotateLeftRight()
        else:
            self.right = self.addToSubTree (self.right, val)
            if self.heightDifference() == -2:
                if val > self.right.value:
                    newRoot = self.rotateLeft()
                else:
                    newRoot = self.rotateRightLeft()

        newRoot.computeHeight()
        return newRoot

    def addToSubTree (self, parent, val):
    """Добавление val в родительское поддерево (если
       таковое существует) и возврат корня, если он
       был изменен из-за поворота."""
        if parent is None:
            return BinaryNode(val)

        parent = parent.add (val)
        return parent
```

Компактная реализация `add`, показанная в примере 5.13, обладает элегантным поведением: разновидность рекурсии, которая делает выбор между двумя возможными рекурсивными функциями на каждой итерации. Метод рекурсивно обходит дерево, сворачивая влево или вправо, как того требуют обстоятельства, до тех пор, пока `addToSubTree` в конечном итоге не запросит добавление `val` в пустое поддерево (т.е. когда `parent` равно `None`). Рекурсия при этом прекращается, и гарантируется, что добавленное значение всегда является листом BST. После того как это действие завершено, каждый последующий рекурсивный вызов завершается и `add` определяет, необходимо ли выполнить поворот для сохранения AVL-свойства. Эти повороты начинаются глубоко в дереве (рядом с листьями) и выполняются по пути к корню.

Поскольку дерево сбалансировано, количество поворотов ограничено $O(\log n)$. Каждый метод поворота состоит из фиксированного количества выполняемых шагов. Таким образом, дополнительные расходы на поддержание AVL-свойства не превышают $O(\log n)$. Операции `rotateRight` и `rotateRightLeft` показаны в примере 5.14.

Пример 5.14. Методы `rotateRight` и `rotateRightLeft`

```
def rotateRight (self):
    """Правый поворот вокруг данного узла."""
    newRoot = self.left
    grandson = newRoot.right
    self.left = grandson
    newRoot.right = self

    self.computeHeight()
    return newRoot

def rotateRightLeft (self):
    """Правый, а затем левый поворот вокруг данного узла."""
    child = self.right
    newRoot = child.left
    grand1 = newRoot.left
    grand2 = newRoot.right
    child.left = grand2
    self.right = grand1

    newRoot.left = self
    newRoot.right = child

    child.computeHeight()
    self.computeHeight()
    return newRoot
```

Для полноты изложения в примере 5.15 показаны методы `rotateLeft` и `rotateLeftRight`.

Пример 5.15. Методы `rotateLeft` и `rotateLeftRight`

```
def rotateLeft (self):
    """Левый поворот вокруг данного узла."""
    newRoot = self.right
    grandson = newRoot.left
    self.right = grandson
    newRoot.left = self

    self.computeHeight()
    return newRoot

def rotateLeftRight (self):
    """Левый, а затем правый поворот вокруг данного узла."""
```

```
child = self.left
newRoot = child.right
grand1 = newRoot.left
grand2 = newRoot.right
child.right = grand1
self.left = grand2

newRoot.left = child
newRoot.right = self

child.computeHeight()
self.computeHeight()
return newRoot
```

Чтобы закрыть вопрос динамического поведения BST, необходимо не менее эффективно удалять элементы. При удалении значения из BST важно сохранить *свойство бинарного дерева поиска*. Если целевой узел, содержащий удаляемое значение, не имеет левого дочернего узла, то можно просто "поднять" его правый дочерний узел, чтобы он занял место удаленного узла. В противном случае следует найти узел с наибольшим значением в поддереве, корнем которого является левый дочерний узел удаляемого узла. Это наибольшее значение можно обменять с целевым узлом. Обратите внимание, что наибольшее значение в левом поддереве не имеет правого дочернего узла, так что его легко удалить, перемещая вверх его левый дочерний узел, если таковой имеется, как показано на рис. 5.14. Выполняющие эту задачу методы приведены в примере 5.16.

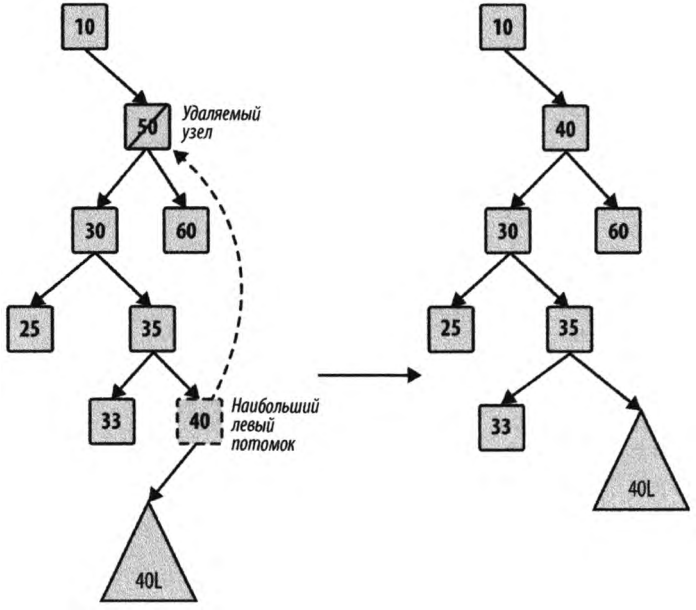


Рис. 5.14. Поиск наибольшего потомка в левом поддереве

Пример 5.16. Методы `remove` *и* `removeFromParent` *класса* `BinaryNode`

```
def removeFromParent (self, parent, val):
    """Вспомогательный метод для метода remove.
       Обеспечивает корректное поведение при удалении
       узла, имеющего дочерние узлы."""
    if parent:
        return parent.remove (val)
    return None

def remove (self, val):
    """ Удаление val из BinaryTree. Работает вместе с
        методом remove из бинарного дерева.
    """
    newRoot = self
    if val == self.value:
        if self.left is None:
            return self.right

        child = self.left
        while child.right:
            child = child.right

        childKey = child.value;
        self.left = self.removeFromParent (self.left, childKey)
        self.value = childKey;

        if self.heightDifference() == -2:
            if self.right.heightDifference() <= 0:
                newRoot = self.rotateLeft()
            else:
                newRoot = self.rotateRightLeft()
    elif val < self.value:
        self.left = self.removeFromParent (self.left, val)
        if self.heightDifference() == -2:
            if self.right.heightDifference() <= 0:
                newRoot = self.rotateLeft()
            else:
                newRoot = self.rotateRightLeft()
    else:
        self.right = self.removeFromParent (self.right, val)
        if self.heightDifference() == 2:
            if self.left.heightDifference() >= 0:
                newRoot = self.rotateRight()
            else:
                newRoot = self.rotateLeftRight()

    newRoot.computeHeight()
    return newRoot
```

Исходный текст метода `remove` имеет структуру, подобную структуре метода `add`. После того как рекурсивный вызов находит целевой узел, содержащий значение, которое следует удалить, он проверяет, не существует ли наибольший потомок в левом поддереве, с которым это значение можно поменять местами. Обратите внимание, как по возвращении из каждого рекурсивного вызова проверяется, не требуется ли выполнить поворот. Так как глубина дерева ограничена величиной $O(log\ n)$, а каждый поворот выполняется за константное время, общее время выполнения удаления представляет собой $O(log\ n)$.

Последняя задача, которую мы хотим видеть решенной в BST, — это возможность выполнения обхода содержимого дерева в отсортированном порядке, то, что попросту невозможно при работе с хеш-таблицами. В примере 5.17 содержатся необходимые изменения `BinaryTree` и `BinaryNode`.

Пример 5.17. Поддержка упорядоченного обхода дерева

```
class BinaryTree:
    def __iter__(self):
        """Упорядоченный обход элементов дерева."""
        if self.root:
            return self.root.inorder()

class BinaryNode:
    def inorder(self):
        """Упорядоченный обход дерева с данным корнем."""
        if self.left:
            for n in self.left.inorder():
                yield n

        yield self.value

        if self.right:
            for n in self.right.inorder():
                yield n
```

С помощью этой реализации вы можете вывести значения `BinaryTree` в отсортированном порядке. Фрагмент кода в примере 5.18 добавляет 10 целых чисел (в обратном порядке) в пустое дерево `BinaryTree` и выводит их в отсортированном порядке.

Пример 5.18. Обход значений в `BinaryTree`

```
bt = BinaryTree()
for i in range(10, 0, -1):
    bt.add(i)
for v in bt:
    print (v)
```

Анализ алгоритма

Производительность поиска в сбалансированном AVL-дереве в среднем случае совпадает с производительностью бинарного поиска (т.е. равна $O(log\ n)$). Обратите внимание, что в процессе поиска никакие повороты не выполняются.

Самобалансирующиеся бинарные деревья требуют более сложного кода вставки и удаления, чем простые бинарные деревья поиска. Пересмотрев методы поворотов, вы увидите, что каждый из них состоит из фиксированного количества операций, так что их можно рассматривать как выполняющиеся за константное время. Высота сбалансированного AVL-дерева благодаря поворотам всегда будет иметь порядок $O(log\ n)$. Таким образом, при добавлении или удалении элемента никогда не будет выполняться больше чем $O(log\ n)$ поворотов. Таким образом, мы можем быть уверены, что вставки и удаления могут выполняться за время $O(log\ n)$. Компромисс, состоящий в том, что AVL-деревья хранят значения высоты в каждом узле (что увеличивает расход памяти), обычно стоит получаемой при этом гарантированной производительности.

Вариации алгоритма

Одним естественным расширением бинарного дерева является дерево с n ветвлениями, в котором каждый узел содержит несколько значений и, соответственно, более чем два потомка. Распространенная версия таких деревьев называется B-деревом, которое с успехом используется при реализации реляционных баз данных. Полный анализ B-деревьев можно найти в [20], а кроме того, в Интернете можно найти полезные руководства по B-деревьям с примерами (см., например, `http://www.bluerwhite.org/btree`).

Еще одним распространенным самобалансирующимся бинарным деревом является *красно-черное дерево*. Красно-черные деревья являются приближенно сбалансированными, гарантирующими, что ни одна ветвь не может иметь высоту, более чем в два раза превосходящую высоту любой другой ветви в дереве. Такая ослабленная гарантия улучшает производительность вставки и удаления путем сокращения количества необходимых поворотов. Реализация этого дерева имеется в классе `algs.model.tree.BalancedTree` в репозитории. Более подробную информацию по красно-черным деревьям можно найти в [20].

6

Алгоритмы на графах

Графы — это фундаментальные структуры, которые представляют сложную структурированную информацию. Изображения на рис. 6.1 являются примерами графов.

В этой главе мы изучим распространенные способы представления графов и связанные с ними алгоритмы. По своей природе граф представляет собой набор элементов, известных как *вершины*, и отношения между парами этих элементов, известные как *ребра*. В этой главе мы используем указанные термины; в других книгах могут использоваться, например, такие термины, как "узел" и "связь", для представления той же информации. Мы рассматриваем только простые графы, в которых нет *а*) петель (ребер из вершины в саму себя) и *б*) нескольких ребер между одной и той же парой вершин.

С учетом структуры, определяемой ребрами графа, многие задачи можно сформулировать в терминах *путей* от исходной вершины графа до целевой вершины, построенных с использованием существующих ребер графа. Иногда ребра имеют соответствующие числовые значения, известные как *вес* ребра. В ряде задач ребра имеют определенную *ориентацию* (подобно улице с односторонним движением). В задаче **кратчайшего пути из одной вершины** указывается одна конкретная вершина s графа, после чего требуется вычислить все кратчайшие пути (с минимальными суммарными весами ребер) для всех прочих вершин графа. Задача **кратчайших путей между всеми парами вершин** состоит в вычислении кратчайших путей для всех пар (u, v) вершин графа.

Некоторые задачи требуют более глубокого понимания базовой структуры графа. Например, минимальное остовное дерево неориентированного взвешенного графа представляет собой подмножество ребер этого графа, такое, что *а*) исходное множество вершин по-прежнему является связным и *б*) сумма весов ребер в этом подмножестве минимальна. Мы покажем, как эффективно решить эту задачу с помощью **алгоритма Прима**.

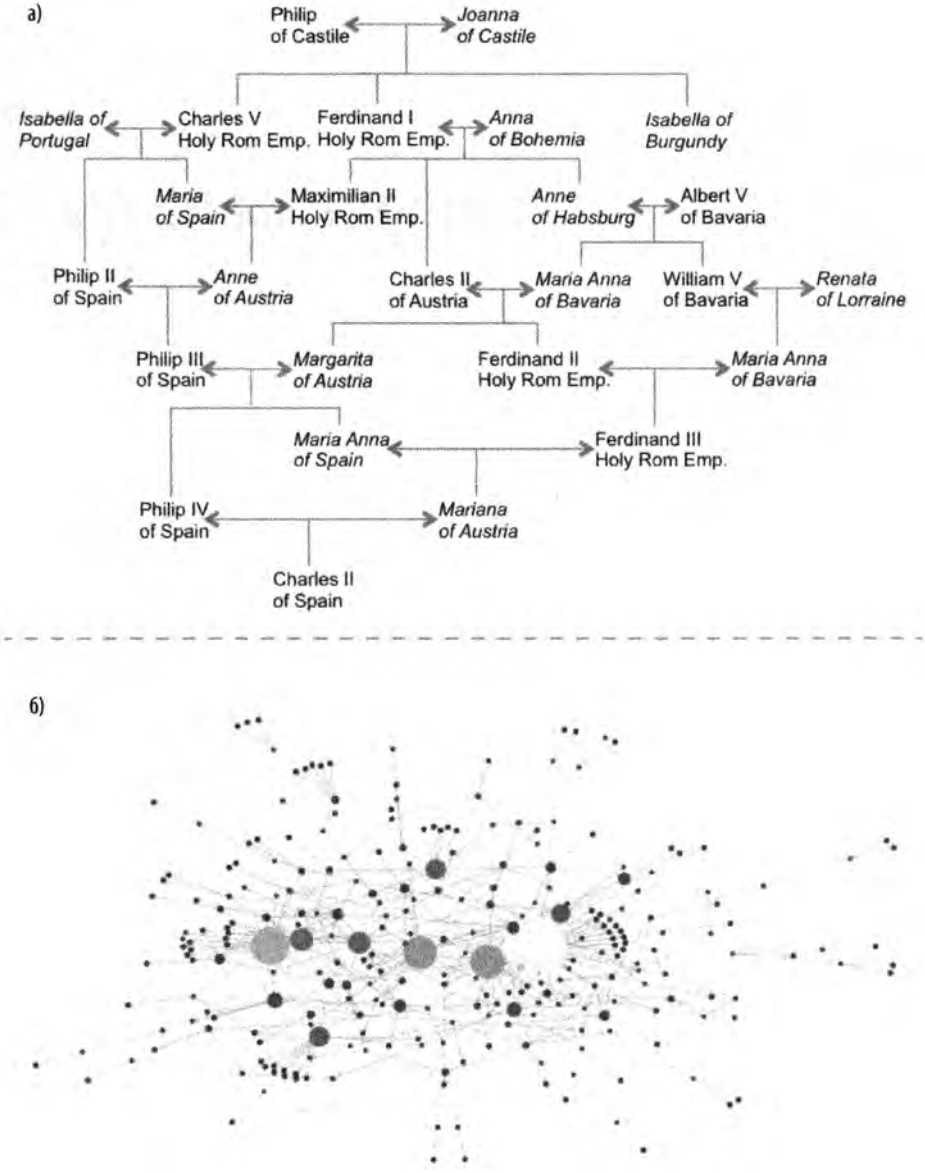


Рис. 6.1. а) Генеалогическое древо короля Испании Карла II (1661–1700); б) молекулярная сеть, связанная с раком печени

Графы

Граф $G=(V,E)$ определяется множеством вершин V и множеством ребер E между парами этих вершин. Имеется три распространенных разновидности графов.

Неориентированные невзвешенные графы

Это модель отношений между вершинами (u, v) без учета направления отношений. Такие диаграммы используются для представления симметричной информации. Например, в графе, моделирующем социальную сеть, если Алиса является другом Боба, то и Боб — друг Алисы.

Ориентированные графы

Это модель отношения между вершинами (u, v), которое отличается от отношения между (v, u), которое может как существовать, так и отсутствовать. Например, программа для построения маршрута должна хранить информацию об улицах с односторонним движением, чтобы не указывать дорогу с нарушением ПДД.

Взвешенные графы

Это модель отношения, в которой с отношениями между вершинами (u, v) связано числовое значение, известное как *вес*. Иногда эти значения могут хранить произвольную, нечисловую информацию. Например, ребро между городами *А* и *Б* может хранить расстояние между ними, предполагаемое время путешествия или название магистрали, соединяющей эти города.

Наиболее высоко структурированные графы — ориентированные взвешенные графы — определяют непустое множество вершин $\{v_0, v_1, v_{n-1}\}$, множество ориентированных (направленных) ребер между парами отдельных вершин (так что каждая пара в каждом направлении имеет не более одного ребра между ними) и положительный вес, связанный с каждым ребром. Во многих приложениях вес представляет собой расстояние или стоимость. Для некоторых приложений ограничение, заключающееся в том, что вес должен быть положительным, ослабляется (например, отрицательный вес может отражать убытки, а не прибыль), но мы будем отдельно предупреждать о таких случаях.

Рассмотрим ориентированный взвешенный граф на рис. 6.2, который состоит из шести вершин и пяти ребер. Мы могли бы хранить этот граф, используя списки смежности, как показано на рис. 6.3, где для каждой вершины v_i поддерживается связанный список узлов, в каждом из которых хранится вес ребра, ведущего к вершине, соседней с v_i. Таким образом, в этом случае базовая структура представляет собой одномерный массив вершин графа.

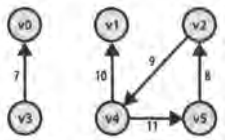


Рис. 6.2. Пример ориентированного взвешенного графа

На рис. 6.4 показано, как хранить ориентированный взвешенный граф в виде матрицы смежности A размером $n \times n$. Запись $A[i][j]$ хранит вес ребра от v_i до v_j; если ребра от v_i до v_j в графе нет, $A[i][j]$ получает некоторое специальное значение, например 0, –1 или даже $-\infty$. Мы можем использовать списки смежности и матрицы и для хранения невзвешенных графов (например, значение 1 для представления существующего ребра). Проверка существования ребра (v_i, v_j) при использовании матриц смежности занимает константное время, но при использовании списков смежности это время зависит от количества ребер в списке v_i. Однако применение матрицы смежности требует большего количества памяти, а кроме того, идентификация всех ребер, исходящих из заданной вершины, выполняется в этом случае за время, пропорциональное количеству вершин, а не количеству этих ребер. Проверка всех возможных ребер становится значительно дороже, когда количество вершин становится большим. Представление матрицы смежности следует использовать при работе с *плотными графами*, в которых имеются почти все возможные ребра.

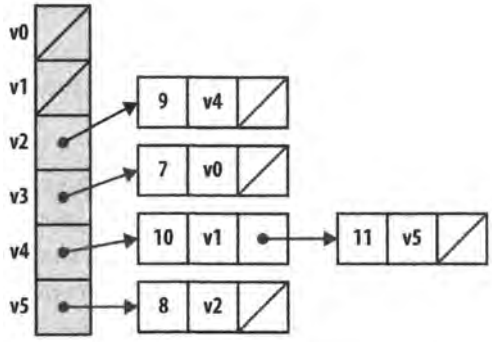


Рис. 6.3. Представление ориентированного взвешенного графа с помощью списков смежности

	v0	v1	v2	v3	v4	v5
v0	0	0	0	0	0	0
v1	0	0	0	0	0	0
v2	0	0	0	0	9	0
v3	7	0	0	0	0	0
v4	0	10	0	0	0	11
v5	0	0	8	0	0	0

Рис. 6.4. Матрица смежности, представляющая ориентированный взвешенный граф

Мы используем запись $<v_0, v_1, \ldots, v_{k-1}>$ для описания пути из k вершин графа, который проходит по $k-1$ ребрам (v_i, v_i+1), где $0 \le i < k$; пути в ориентированном графе могут проходить только по направлениям ребер. Путь $<v_4, v_5, v_2, v_4, v_1>$ на рис. 6.2 является корректным путем. В этом графе имеется *цикл*, который представляет собой путь, который включает одну и ту же вершину несколько раз. Цикл обычно представляется в минимальном виде. Если между любыми двумя вершинами в графе имеется путь, этот граф называется *связным*.

При использовании списка смежности для хранения неориентированного графа одно и то же ребро (u, v) встречается дважды: один раз в связанном списке соседних вершин для u и один раз — для v. Таким образом, для представления неориентированного графа может потребоваться удвоенное количество памяти при использовании списка смежности по сравнению с ориентированным графом с тем же числом вершин и ребер. Это позволяет находить соседей вершины u за время,

пропорциональное фактическому количеству соседей. При использовании матрицы смежности для хранения неориентированного графа $A[i][j] = A[j][i]$.

Проектирование структуры данных

Мы реализуем класс C++ `Graph` для хранения ориентированного (или неориентированного) графа, используя представление списка смежности, реализованное с помощью классов стандартной библиотеки шаблонов (STL). В частности, он хранит информацию в виде массива объектов `list`, по одному списку для каждой вершины. Для каждой вершины u имеется список `list` объектов типа `IntegerPair`, представляющих ребро (u, v) веса w.

Операции на графах подразделяются на несколько категорий.

Создание

Граф может быть изначально построенным из множества n вершин и может быть ориентированным или неориентированным. Когда граф неориентированный, добавление ребра (u, v) добавляет также ребро (v, u).

Изучение

Мы можем определить, ориентированный ли это граф, найти все ребра, инцидентные данной вершине, определить, существует ли конкретное ребро, и найти его вес. Мы также можем построить итератор, который возвращает ребра (и их веса) для любой вершины графа.

Изменение

Мы можем добавить ребра в граф (или удалить их из него). Можно также добавлять или удалять вершины графа, но алгоритмам в этой главе эти операции не требуются.

Мы начнем с рассмотрения путей изучения графа. Двумя распространенными стратегиями являются **поиск в глубину** и **поиск в ширину**.

Поиск в глубину

Рассмотрим лабиринт, показанный в левой части рис. 6.5. После некоторой практики ребенок может быстро найти путь, который проходит от начального поля s к целевому полю t. Один из способов решения этой задачи состоит в том, чтобы проходить как можно дальше вперед, случайным образом выбирая направление всякий раз, когда такой выбор возможен, и помечая, откуда мы пришли. Если вы попадаете в тупик или повторно посещаете место, где уже были, то следует отступить, пока не будет найдено еще не посещенное ответвление. Цифры в правой части рис. 6.5 указывают точки ветвления одного такого решения; фактически при данном решении был посещен каждый квадрат лабиринта.

Мы можем представлять лабиринт на рис. 6.5 с помощью графа, состоящего из вершин и ребер. Вершина создается для каждой точки ветвления в лабиринте (эти точки помечены цифрами в правой части рис. 6.5), а также для тупиков. Ребро между двумя вершинами имеется только тогда, когда между этими вершинами в лабиринте есть прямой путь, на протяжении которого нет никакого ответвления. Представление лабиринта на рис. 6.5 в виде неориентированного графа показано на рис. 6.6. Каждая вершина графа имеет уникальный идентификатор.

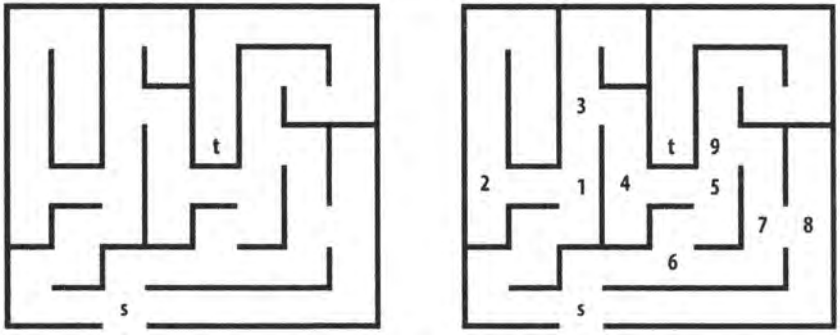


Рис. 6.5. Лабиринт, в котором надо пройти из точки s в точку t

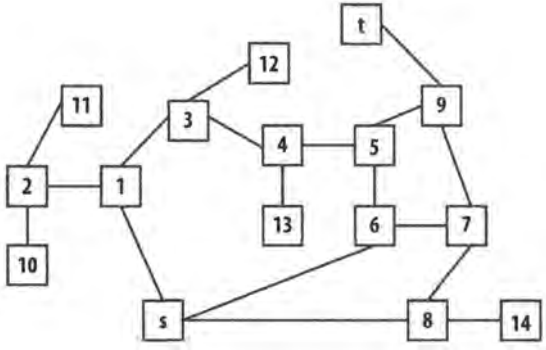


Рис. 6.6. Представление лабиринта на рис. 6.5 в виде графа

Чтобы найти интересующий нас путь в этом лабиринте, нам нужно найти путь в графе $G=(V,E)$ на рис. 6.5 от начальной вершины s к конечной вершине t. В этом примере все ребра неориентированные; однако мы могли бы легко рассмотреть ориентированные ребра, если бы в лабиринте были введены такие ограничения.

Центральной частью **поиска в глубину** является рекурсивная операция `dfsVisit(u)`, которая посещает еще непосещенную вершину `u`. `dfsVisit(u)` записывает свою работу по посещению вершин с помощью окраски их в один из трех цветов.

Белый

Вершина еще не была посещена.

Серый

Вершина была посещена, но может иметь смежную вершину, которая еще не была посещена.

Черный

Вершина была посещена, как и все смежные с ней вершины.

Поиск в глубину

Наилучший, средний и наихудший случаи: $O(V+E)$

```
depthFirstSearch (G,s)
    foreach v in V do
        pred[v] = -1
        color[v] = White              ❶
    dfsVisit(s)
end

dfsVisit(u)
    color[u] = Gray
    foreach neighbor v of u do
        if color[v] = White then      ❷
            pred[v] = u
            dfsVisit(v)
    color[u] = Black                  ❸
end
```

❶ Изначально все вершины помечены как непосещенные.
❷ Поиск непосещенного соседа и перемещение в этом направлении.
❸ Когда все соседи посещены, работа с вершиной завершена.

Изначально каждая вершина имеет белый цвет, указывающий, что она еще не была посещена, и поиск в глубину вызывает `dfsVisit` для исходной вершины *s*. `dfsVisit(u)` окрашивает вершину *u* в серый цвет перед тем, как рекурсивно вызвать `dfsVisit` для всех смежных с *u* и еще непосещенных вершин (т.е. окрашенных в белый цвет). После того как эти рекурсивные вызовы завершены, вершину *u* можно окрашивать в черный цвет, после чего осуществляется возврат из функции. После возврата из рекурсивной функции `dfsVisit` поиск в глубину откатывается к предыдущей вершине в поиске (на самом деле — к вершине, окрашенной в серый цвет), у которой имеются не посещенные соседние вершины, которые необходимо изучить. На рис. 6.7 показан пример работы алгоритма для небольшого графа.

Рис. 6.7. Пример работы поиска в глубину

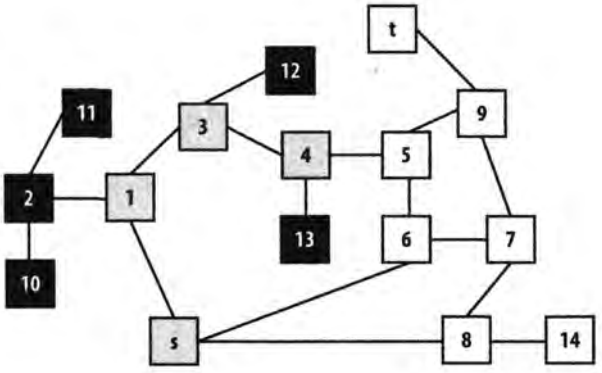
Поиск в глубину исследует как ориентированные, так и неориентированные графы, начиная с вершины *s*, пока не будут посещены все вершины, до которых можно добраться из *s*. Во время выполнения поиск в глубину проходит по ребрам графа, вычисляя информацию, которая раскрывает сложную структуру графа. Для каждой вершины поиск в глубину записывает `pred[v]` — предшественник вершины *v*; эту информацию можно использовать для восстановления пути от вершины-источника *s* до целевой вершины *v*.

Эта вычисленная информация полезна для различных алгоритмов, основанных на поиске в глубину, в том числе для топологической сортировки и выявления сильно связанных компонентов.

Для графа на рис. 6.6 в предположении, что соседние вершины в списках смежности перечислены в порядке возрастания, поиск в глубину дает информацию, показанную на рис. 6.8. Окраска вершин графа приведена для момента после посещения

пятой по счету вершины (в данном случае это вершина 13). Некоторые части графа (вершины, закрашенные черным цветом) посещены полностью и поиск не будет к ним возвращаться. Обратите внимание, что белые вершины еще не были посещены, а серые вершины в настоящее время рекурсивно посещаются процедурой `dfsVisit`.

Поиск в глубину не имеет глобальной информации о графе, поэтому он слепо просматривает вершины <5, 6, 7, 8> несмотря на то, что они находятся в неправильном направлении от целевой вершины *t*. После завершения поиска в глубину значения `pred[]` могут использоваться для воссоздания пути из исходной вершины *s* в каждую вершину графа.



v	pred
s	-1
1	s
2	1
11	2
10	2
3	1
12	3
4	3
13	4
5	*
6	*
7	*
8	*
14	*
9	*
t	*

Рис. 6.8. Вычисленные значения `pred[]` *для образца неориентированного графа; состояние после окрашивания в черный цвет пяти вершин*

Обратите внимание, что найденный путь может не быть кратчайшим возможным путем; когда поиск в глубину завершается, путь от *s* к *t* содержит семь вершин <*s*, 1, 3, 4, 5, 9, *t*>, в то время как имеется более короткий путь из пяти вершин <*s*, 6, 5, 9, *t*>. В данном случае понятие "кратчайшего пути" определяется количеством точек ветвления между *s* и *t*.

Входные и выходные данные алгоритма

Входные данные представляют собой граф $G=(V,E)$ и исходную вершину $s \in V$, представляющую точку начала поиска.

Поиск в глубину генерирует массив `pred[v]`, в котором записаны вершины-предшественники v на основании упорядочения с помощью поиска в глубину.

Контекст применения алгоритма

Поиск в глубину при обходе графа должен хранить только цвет каждой вершины (белый, серый или черный). Таким образом, для поиска в глубину необходимы накладные расходы памяти, равные $O(n)$.

Поиск в глубину может хранить свою информацию в массивах отдельно от графа. Поиску в глубину требуется только возможность обхода всех вершин графа, смежных с данной. Эта возможность позволяет легко выполнять поиск в глубину сложной информации, поскольку функция `dfsVisit` обращается к исходному графу за информацией о структуре с доступом только для чтения.

Реализация алгоритма

В примере 6.1 содержится простая реализация на языке программирования C++. Обратите внимание, что информация о цвете вершин используется только внутри методов `dfsVisit`.

Пример 6.1. Реализация поиска в глубину

```
// Посещение вершины u графа и обновление информации
void dfsVisit(Graph const& graph, int u,                    /* вход  */
              vector<int>&pred,vector<vertexColor>&color) /* выход */
{
    color[u] = Gray;

    // Обработка всех соседей u.
    for (VertexList::const_iterator ci = graph.begin(u);
            ci != graph.end(u); ++ci)
    {
        int v = ci->first;

        // Изучение непосещенных вершин и запись pred[].
        // По завершении вызова откат к смежным вершинам.
        if (color[v] == White)
        {
            pred[v] = u;
            dfsVisit(graph, v, pred, color);
        }
    }

    color[u] = Black; // Отмечаем, что все соседи пройдены.
}
```

```
/**
* Выполнение поиска в глубину начиная с вершины s и вычисление
* pred[u] - вершины, предшествующей u в лесу поиска в глубину.
*/
void dfsSearch(Graph const& graph, int s,     /* вход  */
               vector<int>& pred)             /* выход */
{
    // Инициализация массива pred[]
    // и окраска всех вершин в белый цвет.
    const int n = graph.numVertices();
    vector<vertexColor> color(n, White);
    pred.assign(n, -1);
    // Поиск начиная с исходной вершины.
    dfsVisit(graph, s, pred, color);
}
```

Анализ алгоритма

Рекурсивная функция `dfsVisit` вызывается один раз для каждой вершины графа. В `dfsVisit` необходимо проверить все соседние вершины; для ориентированных графов проход по ребрам осуществляется один раз, в то время как в неориентированных графах проход по ним осуществляется один раз, и еще один раз осуществляется их просмотр. В любом случае общая производительность алгоритма составляет $O(V+E)$.

Вариации алгоритма

Если исходный граф не связанный, то между s и некоторыми вершинами графа может не существовать пути; эти вершины будут оставаться непосещенными. Некоторые вариации гарантируют, что все вершины будут обработаны, достигая этого путем дополнительных выполнений `dfsVisit` для непосещенных вершин в методе `dfsSearch`. Если это будет сделано, то значения `pred[]` описывают лес поиска в глубину. Чтобы найти корни деревьев в этом лесу, следует сканировать массив `pred[]` на наличие в нем вершин `r`, для которых значения `pred[r]` равны –1.

Поиск в ширину

Поиск в ширину представляет собой другой подход, отличный от поиска в глубину. При поиске в ширину систематически посещаются все вершины графа $G=(V,E)$, которые находятся на расстоянии k ребер от исходной вершины s, перед посещением любой вершины на расстоянии $k+1$ ребер. Этот процесс повторяется до тех пор, пока не останется вершин, достижимых из s. Этот поиск не посещает вершины графа G, которые не достижимы из s. Алгоритм работает как для неориентированных, так и для ориентированных графов.

Поиск в ширину гарантированно находит кратчайший путь в графе от вершины s до целевой вершины, хотя в процессе работы может выполнить вычисления

для весьма большого количества узлов. Поиск в глубину пытается найти путь как можно быстрее, но цена этой скорости — отсутствие гарантии того, что этот путь будет кратчайшим.

На рис. 6.9 показан процесс работы поиска в ширину на том же маленьком графе из рис. 6.7, на котором демонстрировалась работа поиска в глубину. Для начала обратите внимание, что серые вершины в графе — те, которые находятся в очереди. При каждой очередной итерации вершина удаляется из очереди, и в нее добавляются ее непосещенные соседи.

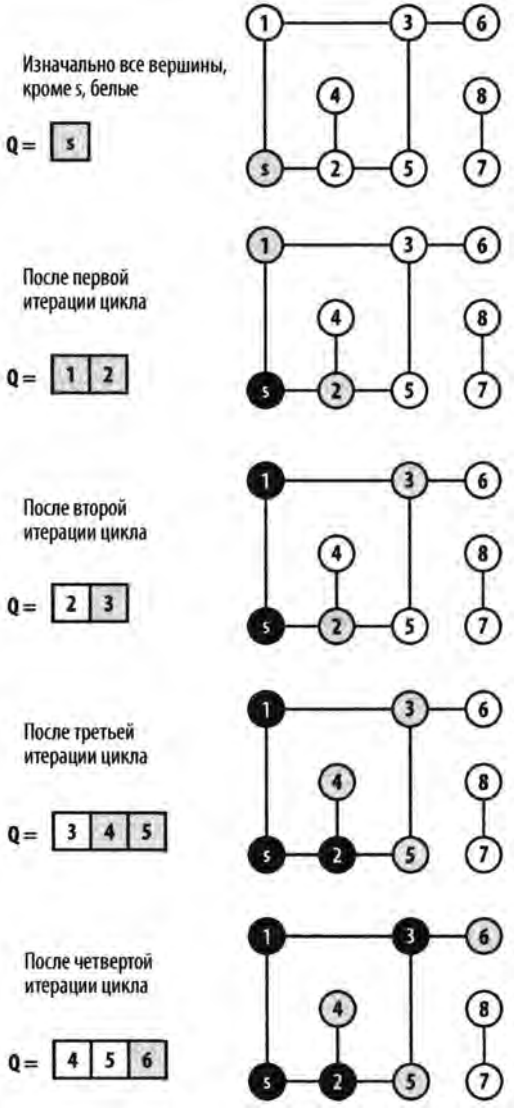
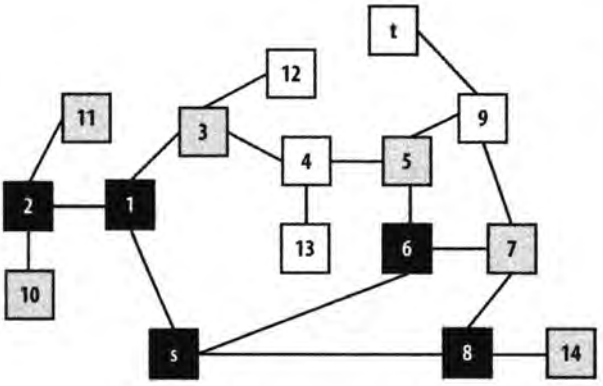


Рис. 6.9. Пример поиска в ширину

Поиск в ширину работает без необходимости какого-либо возврата. Он также окрашивает вершины в белый, серый или черный цвет, как и поиск в глубину. Смысл этих цветов в точности тот же, что и при поиске в глубину. Для сравнения с поиском в глубину рассмотрим поиск в ширину на том же графе (рис. 6.6) в момент, когда, как и ранее, в черный цвет окрашена пятая вершина (вершина 2), как показано на рис. 6.10. В момент, показанный на рисунке, поиск в ширину окрасил в черный цвет исходную вершину *s*, вершины, отстоящие от нее на одно ребро — {1, 6 и 8}, — и вершину 2, отстоящую на два ребра от *s*.



v	pred	dist
s	-1	0
1	s	1
2	1	2
3	1	2
4	*	*
5	6	2
6	s	1
7	6	2
8	s	1
9	*	*
10	2	3
11	2	3
12	*	*
13	*	*
14	8	2
t	*	*

Рис. 6.10. Поиск в ширину на графе с рис. 6.7 после окраски в черный цвет пяти вершин

Все остальные вершины на расстоянии двух ребер от *s* — {3, 5, 7, 14} — находятся в очереди *Q* ожидающих обработки. Там же есть и две вершины, отстоящие от исходной вершины *s* на три ребра — {10, 11}. Обратите внимание, что все вершины, находящиеся в очереди, окрашены в серый цвет, отражающий их активное состояние.

Входные и выходные данные алгоритма

Входными данными алгоритма являются граф $G=(V,E)$ и исходная вершина $s \in V$, представляющая точку начала поиска.

Поиск в ширину создает два вычисляемых массива. `dist[v]` хранит расстояние в ребрах на кратчайшем пути от вершины *s* до *v*. В элементе массива `pred[v]` записывается предшественник вершины *v* при поиске в ширину. Значения `pred[]` хранят результат поиска в виде дерева поиска в ширину; если исходный граф несвязный, то все вершины *w*, недоступные из *s*, имеют значение `pred[w]`, равное -1.

Контекст применения алгоритма

Поиск в ширину хранит вершины, которые должны быть обработаны, в очереди; таким образом, ему требуется память $O(V)$. Поиск гарантированно находит кратчайший путь (хотя на самом деле их может быть несколько) в графах, вершины которых создаются "на лету" (как будет показано в главе 7, "Поиск путей в ИИ"). Фактически все пути в созданном дереве поиска в ширину являются кратчайшими путями из *s* в терминах количества ребер.

Реализация алгоритма

Простая реализация алгоритма на C++ показана в примере 6.2. Поиск в ширину хранит свое состояние в очереди; таким образом, в нем нет рекурсивных вызовов.

Пример 6.2. Реализация поиска в ширину

```
/**
* Выполняет поиск в ширину в графе из вершины s и вычисляет
* BFS-расстояние и предыдущую вершину для всех вершин графа.
*/
void bfsSearch(Graph const &graph, int s,          /* Вход  */
               vector<int>&dist,vector<int>&pred) /* Выход */
{
    // Инициализация dist и pred для пометки непосещенных вершин.
    // Начинаем с s и окрашиваем ее в серый цвет, так как ее
    // соседи еще не посещены.
    const int n = graph.numVertices();
    pred.assign(n, -1);
    dist.assign(n, numeric_limits<int>::max());
    vector<vertexColor> color (n, White);

    dist[s] = 0;
    color[s] = Gray;

    queue<int> q;
    q.push(s);
    while (!q.empty()) {
        int u = q.front();

        // Исследуем соседей u для расширения горизонта поиска
```

```
        for(VertexList::const_iterator ci = graph.begin (u);
            ci != graph.end (u); ++ci) {
            int v = ci->first;
            if (color[v] == White) {
                dist[v] = dist[u]+1;
                pred[v] = u;
                color[v] = Gray;
                q.push(v);
            }
        }

        q.pop();
        color[u] = Black;
    }
}
```

Анализ алгоритма

Во время инициализации поиск в ширину обновляет информацию для всех вершин за время $O(V)$. Когда вершина впервые посещена (и окрашена в серый цвет), она помещается в очередь, так что ни одна вершина не добавляется в очередь дважды. Поскольку очередь может добавлять и удалять элементы за константное время, затраты времени на работу с очередью составляют $O(V)$. И наконец каждая вершина помещается в очередь только один раз, и соседние с ней вершины обходятся тоже ровно один раз. Общее количество итераций по ребрам в графе ограничено общим числом ребер, или $O(E)$. Таким образом, общая производительность алгоритма — $O(V+E)$.

Поиск в ширину

Наилучший, средний и наихудший случаи: $O(V+E)$

```
breadthFirstSearch (G, s)
    foreach v in V do
        pred[v] = -1
        dist[v] = ∞
        color[v] = белый                    ❶
    color[s] = серый
    dist[s] = 0
    Q = пустая очередь                      ❷
    enqueue (Q, s)

    while Q не пустая do
        u = head(Q)
        foreach сосед v вершины u do
```

```
        if color[v] = белый then
            dist[v] = dist[u] + 1
            pred[v] = u
            color[v] = серый
            enqueue (Q, v)
        dequeue (Q)
        color[u] = черный                    ❸
end

❶ Изначально все вершины помечаются как непосещенные.
❷ Очередь содержит коллекцию серых вершин, которые будут посещены.
❸ Когда посещены все соседи, работа с вершиной завершается.
```

Кратчайший путь из одной вершины

Предположим, вы хотите совершить полет на частном самолете по кратчайшему маршруту от Киева до Петербурга. Предположим, что вам известны расстояния между аэропортами для всех пар городов, достижимых один из другого на имеющемся самолете с не слишком большим баком для горючего. Наилучший известный алгоритм для решения этой задачи — **алгоритм Дейкстры** — находит кратчайшие пути из Киева ко всем прочим аэропортам, хотя может прекратить свою работу после того, как обнаружит кратчайший путь до Петербурга.

В этом примере мы минимизировали пройденное расстояние. В других приложениях мы могли бы заменить расстояния временем (например, доставить пакет по сети как можно быстрее) или затратами (например, найти самый дешевый способ долететь из Киева в Петербург). Решения этих задач также соответствуют кратчайшим путям в графе.

Алгоритм Дейкстры основан на применении структуры данных, известной как *очередь с приоритетами*. Очередь с приоритетами поддерживает коллекцию элементов, каждый из которых имеет связанный с ним целочисленный *приоритет*, который представляет важность элемента. Очередь с приоритетами позволяет вставить в нее элемент x со связанным с ним приоритетом p. Более низкие значения p представляют элементы с большей важностью. Основной операцией очереди с приоритетами является `getMin`, которая возвращает элемент очереди с приоритетами, значение приоритета которого является самым низким (или, другими словами, который является наиболее важным элементом). Другая операция, которая может предоставляться очередью с приоритетами — *decreasePriority*, — позволяет найти конкретный элемент в очереди с приоритетами и уменьшить значение связанного с ним приоритета (увеличить его важность), оставляя при этом элемент в очереди с приоритетами.

Алгоритм Дейкстры

Наилучший, средний и наихудший случаи: $O((V+E)*\log V)$

```
singleSourceShortest (G, s)
    PQ = Пустая очередь с приоритетами
    foreach v in V do
        dist[v] = ∞                                    ❶
        pred[v] = -1

    dist[s] = 0
    foreach v in V do                                  ❷
        Вставка (v, dist[v]) в PQ

    while PQ не пустая do
        u = getMin(PQ)                                 ❸
        foreach сосед v вершины u do
            w = вес ребра (u,v)
            newLen = dist[u] + w
            if newLen < dist[v] then                   ❹
                decreasePriority (PQ, v, newLen)
                dist[v] = newLen
                pred[v] = u
end
```

❶ Изначально все вершины рассматриваются как недостижимые.
❷ Заполнение PQ вершинами с расстояниями кратчайшего пути.
❸ Удаление вершины, которая имеет кратчайшее расстояние до исходной.
❹ Если найден кратчайший путь от s к v, запись и обновление PQ.

Исходная вершина s известна заранее, а `dist[v]` устанавливается равным ∞ для всех вершин, за исключением s, для которой `dist[s]=0`. Все эти вершины вставляются в очередь с приоритетами `PQ`, с приоритетом, равным `dist[v]`; таким образом, s будет первой вершиной, удаленной из `PQ`. На каждой итерации алгоритм Дейкстры удаляет из `PQ` вершину, ближайшую к s среди всех оставшихся непосещенными вершин в `PQ`. Вершины в `PQ` потенциально обновляются, отражая более близкие расстояния, получаемые из посещенных к этому моменту вершин, как показано на рис. 6.11. После V итераций `dist[v]` содержит кратчайшие расстояния от s до всех вершин $v \in V$.

Концептуально алгоритм Дейкстры работает жадным образом, путем расширения множества вершин S, для которого известны кратчайшие пути от исходной вершины s до каждой вершины $v \in S$, *но только с использованием путей, которые включают вершины в* S. Первоначально S представляет собой множество из одного элемента — $\{s\}$. Для расширения S, как показано на рис. 6.12, алгоритм Дейкстры находит вершину $v \in V-S$ (т.е. вершины за пределами заштрихованной области), для которой расстояние до s является наименьшим, и следует по ребрам, исходящим

из v, чтобы определить, не существует ли более короткого пути к некоторой другой вершине. Так, например, после обработки v_2 алгоритм определяет, что расстояние от s к v_3 по пути, содержащему только вершины из S, в действительности равно 17 (по пути $<s, v_2, v_3>$). Когда множество S становится равным V, алгоритм завершается с конечным результатом, показанным на рис. 6.12.

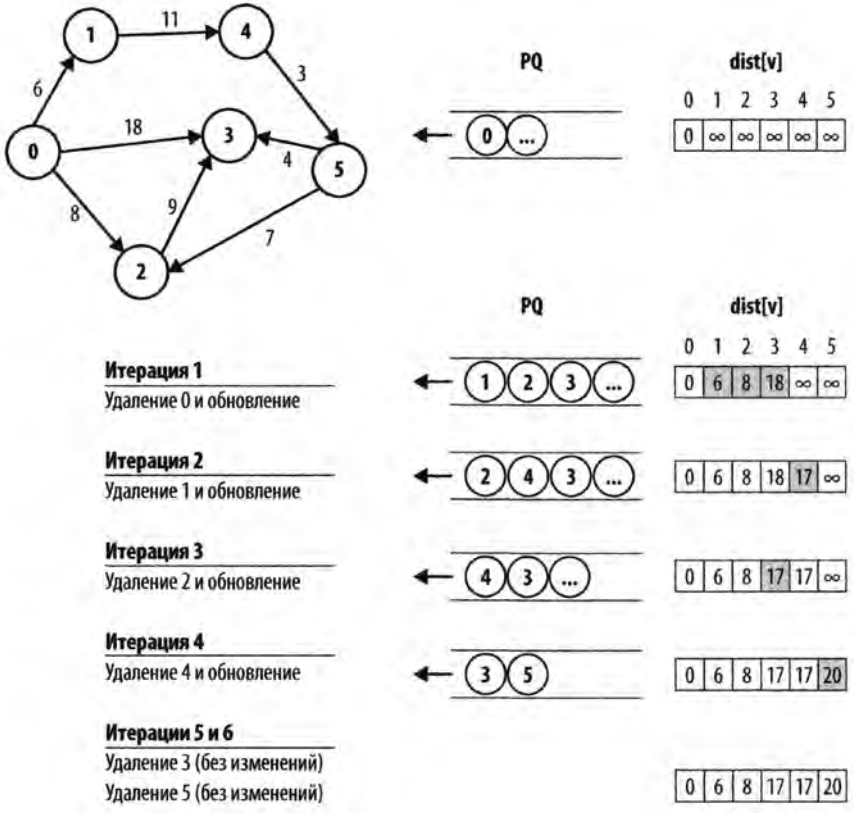
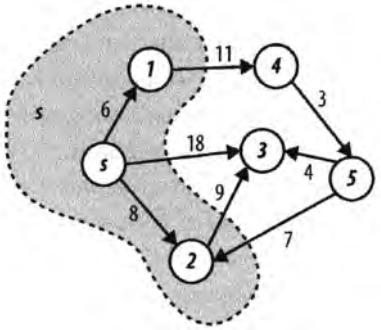


Рис. 6.11. Пример работы алгоритма Дейкстры



v	pred[]	dist[]
s	-1	0
1	s	6
2	s	8
3	2	17
4	1	17
5	4	20

Рис. 6.12. Расширение множества S в алгоритме Дейкстры

Входные и выходные данные алгоритма

Входными данными алгоритма Дейкстры являются ориентированный взвешенный граф $G=(V,E)$ и исходная вершина $s \in V$. Каждое ребро графа $e=(u,v)$ имеет связанный с ним неотрицательный вес.

Алгоритм Дейкстры создает два вычисляемых массива. Основным результатом является массив `dist[]`, значения которого представляют собой расстояние от исходной вершины s до каждой вершины графа. Вторичный результат — массив `pred[]`, который может быть использован для восстановления фактических кратчайших путей от вершины s до каждой вершины графа.

Веса ребер неотрицательные (т.е. они больше или равны нулю); если это предположение не выполняется, то `dist[]` может содержать неверные результаты. Что еще хуже, алгоритм Дейкстры может зациклиться, если в графе существует цикл, сумма весов ребер которого меньше нуля.

Реализация алгоритма

При выполнении алгоритма Дейкстры `dist[v]` представляет максимальную длину кратчайшего пути из исходной вершины s к v с использованием только тех вершин, которые уже посещены (множество S). Кроме того, для каждого $v \in S$ значение `dist[v]` является окончательным и верным. К счастью, алгоритму Дейкстры не нужно создавать и поддерживать множество S. Он изначально создает множество, содержащее вершины в V, а затем удаляет из множества вершины по одной для вычисления правильных значений `dist[v]`; для удобства мы по-прежнему ссылаемся на это постоянно сокращающееся множество как на $V–S$. Алгоритм Дейкстры завершается, когда все вершины либо посещены, либо недостижимы из исходной вершины s.

В приведенной в примере 6.3 реализации на языке программирования C++ вершины множества $V–S$ хранятся в бинарной пирамиде в виде очереди с приоритетами, потому что в ней мы можем найти вершину с *наименьшим приоритетом* за константное время. Кроме того, когда обнаруживается более короткий путь от s к v, `dist[v]` уменьшается и требует модификации пирамиды. К счастью, операция *decreasePriority* (в бинарной пирамиде известная как `decreaseKey`) может быть выполнена в наихудшем случае за время $O(\log q)$, где q — количество вершин в бинарной пирамиде, которое никогда не превышает количество вершин V.

Пример 6.3. Реализация алгоритма Дейкстры

```
/** Для данного ориентированного взвешенного графа
 *  вычисляет кратчайшее расстояние до вершин и
 *  указатели на предшественников для всех вершин. */
void singleSourceShortest (Graph const &g, int s, /* Вход  */
                           vector<int> &dist,     /* Выход */
                           vector<int> &pred)     /* Выход */
```

```
{
    // Инициализация массивов dist[] и pred[]. Начинает работу с
    // вершины s, устанавливая dist[s] равным 0. Очередь с
    // приоритетами PQ содержит все вершины из G.
    const int n = g.numVertices();
    pred.assign (n, -1);
    dist.assign (n, numeric_limits<int>::max());
    dist[s] = 0;
    BinaryHeap pq(n);
    for (int u = 0; u < n; u++) { pq.insert (u, dist[u]); }

    // Поиск вершины в постоянно сокращающемся множестве V-S, у
    // которой dist[] наименьшее. Вычисление заново новых путей
    // для обновления всех кратчайших путей.
    while (!pq.isEmpty()) {
        int u = pq.smallest();
        // Для соседей u проверяем, не является ли newLen (наилучший
        // путь от s->u + вес ребра u->v) лучшим, чем наилучший путь
        // s->v. Если да, обновляется dist[v] и корректируется
        // бинарная пирамида. Вычисления используют тип long, чтобы
        // избежать переполнения.
        for (VertexList::const_iterator ci = g.begin (u);
             ci != g.end (u); ++ci) {
            int v = ci->first;
            long newLen = dist[u];
            newLen += ci->second;
            if (newLen < dist[v]) {
                pq.decreaseKey (v, newLen);
                dist[v] = newLen;
                pred[v] = u;
            }
        }
    }
}
```

Возможно возникновение арифметической ошибки, если сумма весов ребер превышает `numeric_limits<int>::max()` (при том что веса отдельных ребер не превышают этого значения). Чтобы избежать этой ситуации, мы вычисляем `newLen` с использованием типа данных `long`.

Анализ алгоритма

В реализации алгоритма Дейкстры в примере 6.3 цикл `for`, который создает начальную очередь с приоритетами, выполняет операцию вставки V раз, в результате чего его время работы представляет собой $O(V \cdot log\ V)$. В оставшемся цикле `while` каждое ребро посещается один раз, и, таким образом, `decreaseKey` вызывается не более чем E раз, что дает время выполнения $O(E \cdot log\ V)$. Таким образом, общая производительность алгоритма — $O((V+E) \cdot log\ V)$.

Алгоритм Дейкстры для плотных графов

Имеется версия алгоритма Дейкстры, подходящая для плотных графов, представленных с использованием матрицы смежности. Реализация на языке программирования C++ в примере 6.4 больше не нуждается в очереди с приоритетами и оптимизирована для использования двумерного массива для хранения матрицы смежности. Эффективность этой версии определяется тем, насколько быстро можно получить наименьшее значение `dist[]` в V–S. Цикл `while` выполняется V раз, так как S растет по одной вершине за итерацию. Поиск наименьшего `dist[u]` в V–S исследует все вершины V. Обратите внимание, что каждое ребро проверяется только один раз во внутреннем цикле внутри цикла `while`. Так как E не может превышать V^2, общее время работы данной версии алгоритма составляет $O(V^2)$.

Алгоритм Дейкстры для плотных графов

Наилучший, средний и наихудший случаи: $O(V^2 + E)$

```
singleSourceShortest (G, s)
    foreach v in V do
        dist[v] = ∞                                                    ❶
        pred[v] = -1
        visited[v] = false
    dist[s] = 0

    while некоторая непосещенная вершина v имеет dist[v] < ∞ do        ❷
        u = вершина, для которой dist[u] – наименьшее среди            ❸
            всех непосещенных вершин.
        if dist[u] = ∞ then return
        visited[u] = true

        foreach сосед v вершины u do
            w = вес ребра edge (u, v)
            newLen = dist[u] + w
            if newLen < dist[v] then                                   ❹
                    dist[v] = newLen
            pred[v] = u
end

❶ Изначально все вершины рассматриваются как недостижимые.
❷ Останов, если все непосещенные вершины имеют dist[v] = ∞.
❸ Поиск вершины, которая имеет наименьшее расстояние до исходной.
❹ Если найден наикратчайший путь от s к v, записываем новую длину.
```

Из-за структуры матрицы смежности этот вариант больше не нуждается в очереди с приоритетами. Вместо этого на каждой итерации алгоритм выбирает

непосещенную вершину с наименьшим значением dist[]. На рис. 6.13 продемонстрировано выполнение алгоритма над небольшим графом.

Пример 6.4. Оптимизированный алгоритм Дейкстры для плотных графов

```
/** Для заданной матрицы смежности int[][] с весами ребер вычисляет
 *  кратчайшее расстояние ко всем вершинам графа (dist) и
 *  предшественников для всех вершин на кратчайших путях (pred) */
void singleSourceShortestDense(int n, int** const weight,
                               int s,                    /* Вход  */
                               int* dist, int* pred)     /* Выход */
{
    // Инициализация массивов dist[] и pred[]. Работа начинается с
    // вершины s путем установки dist[s] равной 0. Все вершины
    // являются непосещенными.
    bool* visited = new bool[n];

    for (int v = 0; v < n; v++)
    {
        dist[v] = numeric_limits<int>::max();
        pred[v] = -1;
        visited[v] = false;
    }

    dist[s] = 0;

    // Поиск кратчайшего расстояния от s к непосещенным вершинам.
    // Вычисление потенциальных новых путей для обновления всех
    // кратчайших путей.
    while (true)
    {
        int u = -1;
        int sd = numeric_limits<int>::max();

        for (int i = 0; i < n; i++)
        {
            if (!visited[i] && dist[i] < sd)
            {
                sd = dist[i];
                u = i;
            }
        }

        if (u == -1)
        {
            break;    // Выход, если новые пути не найдены
        }

        // Для соседей u проверяем, является ли наилучшая длина пути
```

```
        // s->u + вес ребра u->v лучшей, чем путь s->v. При
        // вычислениях используем тип long.
        visited[u] = true;

        for (int v = 0; v < n; v++)
        {
            int w = weight[u][v];

            if (v == u) continue;

            long newLen = dist[u];
            newLen += w;

            if (newLen < dist[v])
            {
                dist[v] = newLen;
                pred[v] = u;
            }
        }
    }

    delete [] visited;
}
```

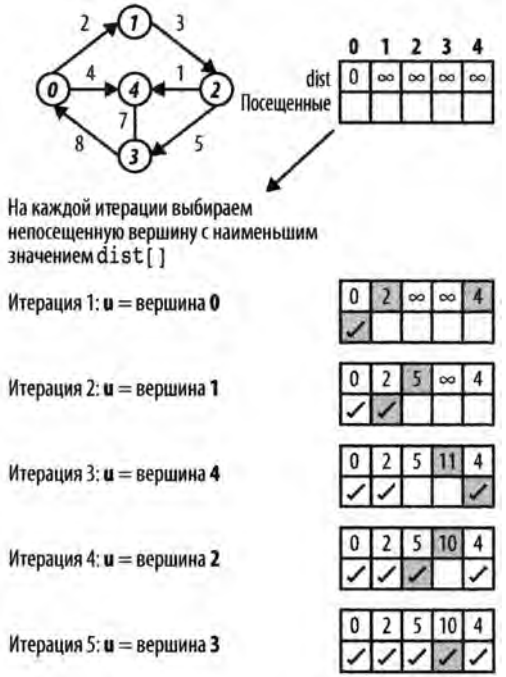


Рис. 6.13. Пример алгоритма Дейкстры для плотного графа

Вариации алгоритма

Мы можем искать самый надежный путь для отправки сообщений из одной точки в другую по сети, в которой нам известны вероятности корректной передачи в каждой ветви. Вероятность корректной доставки сообщения для любого пути (т.е. последовательности ветвей) равна произведению всех вероятностей ветвей вдоль пути. Мы можем заменить вероятность на каждом ребре значением логарифма вероятности с обратным знаком. Кратчайший путь в этом графе соответствует самому надежному пути в исходном графе.

Алгоритм Дейкстры нельзя использовать при наличии отрицательных весов ребер. Однако при этом может использоваться алгоритм **Беллмана–Форда**, если только в графе нет цикла, сумма весов ребер которого меньше нуля. Понятие "кратчайший путь" при наличии такого цикла теряет смысл. Хотя пример графа на рис. 6.14 содержит цикл {1,3,2}, сумма весов составляющих его ребер положительна, поэтому алгоритм Беллмана–Форда будет корректно с ним работать.

Реализация алгоритма Беллмана–Форда на языке программирования C++ приведена в примере 6.5.

Алгоритм Беллмана–Форда

Наилучший, средний и наихудший случаи: $O(V*E)$

```
singleSourceShortest (G, s)
    foreach v in V do
        dist[v] = ∞                                    ❶
        pred[v] = -1
    dist[s] = 0

    for i = 1 to n do
        foreach ребро (u,v) из E do
            newLen = dist[u] + вес ребра (u,v)
            if newLen < dist[v] then                   ❷
                if i = n then Сообщить об "отрицательном цикле"
                dist[v] = newLen
                pred[v] = u
end
```

❶ Изначально все вершины рассматриваются как недостижимые.
❷ Если открыт более короткий путь от s до v, записать новую длину.

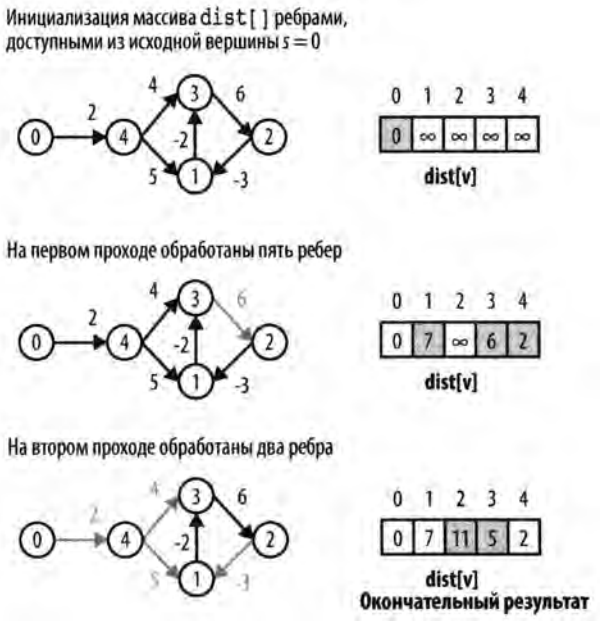


Рис. 6.14. Пример работы алгоритма Беллмана–Форда

Пример 6.5. Алгоритм Беллмана–Форда для кратчайших путей из одной вершины

```
/**
 * Для заданного ориентированного взвешенного графа вычисляет
 * кратчайшие расстояния до вершин графа (dist) и записывает ссылки
 * на предшественников для всех вершин (pred), что позволяет
 * воспроизвести кратчайшие пути. Веса графа могут быть
 * отрицательными, лишь бы в графе не было циклов с отрицательным
 * суммарным весом.
*/
void singleSourceShortest(Graph const& graph, int s, /* Вход  */
                          vector<int>& dist,         /* Выход */
                          vector<int>& pred)         /* Выход */
{
    // Инициализация массивов dist[] и pred[].
    const int n = graph.numVertices();
    pred.assign(n, -1);
    dist.assign(n, numeric_limits<int>::max());
    dist[s] = 0;

    // После n-1 итерации можно гарантировать, что все расстояния
    // от s до всех вершин будут корректно вычисленными кратчайшими
    // расстояниями. Таким образом, изменение значения на n-й
    // итерации гарантирует наличие цикла с отрицательным весом.
    // Если на какой-то итерации изменений нет, цикл можно завершать.
```

```
for (int i = 1; i <= n; i++)
    {
        bool failOnUpdate = (i == n);
        bool leaveEarly = true;

        // Обработка каждой вершины u и ее ребер для выяснения, не
        // дает ли какое-то ребро (u,v) более короткое расстояние
        // s->v путем s->u->v. Для предотвращения переполнений
        // используем тип long.
        for (int u = 0; u < n; u++)
        {
            for (VertexList::const_iterator ci = graph.begin(u);
                    ci != graph.end(u); ++ci)
            {
                int v = ci->first;
                long newLen = dist[u];
                newLen += ci->second;

                if (newLen < dist[v])
                {
                    if (failOnUpdate)
                    {
                        throw "Граф содержит отрицательный цикл";
                    }

                    dist[v] = newLen;
                    pred[v] = u;
                    leaveEarly = false;
                }
            }
        }

        if (leaveEarly)
        {
            break;
        }
    }
}
```

Интуитивно алгоритм Беллмана–Форда работает путем n "выметаний" графа, проверяющих возможность улучшить вычисленные значения `dist[v]` для заданных `dist[u]` и веса ребра (u,v). В экстремальной ситуации, когда кратчайший путь до некоторой вершины проходит через все вершины графа, требуется как минимум $n-1$ итераций. Еще одна причина для использования $n-1$ выметания заключается в том, что ребра могут посещаться в произвольном порядке, так что это количество итераций гарантирует, что найдены все более краткие пути.

Алгоритм Беллмана–Форда не работает только тогда, когда в графе имеется цикл с отрицательным суммарным весом ребер. Чтобы обнаружить такой цикл с отрицательным весом, мы выполняем n итераций (на одну больше, чем это необходимо),

и если на этой итерации выполняется корректировка некоторых значений `dist`, то это означает наличие цикла с отрицательным весом. Производительность алгоритма Беллмана–Форда, как видно из вложенных циклов `for`, равна $O(V \cdot E)$.

Сравнение вариантов поиска кратчайших путей из одной вершины

Ниже подытожены ожидаемые производительности трех алгоритмов, полученные с помощью грубых оценок.

- Алгоритм Беллмана–Форда: $O(V \cdot E)$
- Алгоритм Дейкстры для плотных графов: $O(V^2 + E)$
- Алгоритм Дейкстры с очередью с приоритетами: $O((V + E) \cdot log\ V)$

Мы сравним эти алгоритмы в различных сценариях. Естественно, чтобы выбрать тот алгоритм, который лучше всего подходит для ваших данных, следует выполнить хронометраж разных реализаций. В следующих таблицах мы выполняем алгоритмы по 10 раз и отбрасываем наилучший и наихудший результаты; таблицы показывают среднее для оставшихся восьми выполнений.

Данные хронометража

Достаточно трудно генерировать случайные графы. В табл. 6.1 мы показываем производительность для сгенерированных графов с $|V| = k^2 + 2$ вершинами и $|E| = k^3 - k^2 + 2k$ ребрами (подробности вы сможете найти в коде реализации в репозитории). Обратите внимание, что количество ребер грубо равно $n^{1.5}$, где n — количество вершин в V. Лучшую производительность показывает алгоритм Дейкстры с очередью с приоритетами, но алгоритм Беллмана–Форда отстает не так уж сильно. Обратите внимание на плохую работу вариантов, оптимизированных для плотных графов.

Таблица 6.1. Время (в секундах) вычисления кратчайших путей из одной вершины в тестовых графах

V	*E*	Алгоритм Дейкстры с очередью с приоритетами	Алгоритм Дейкстры, оптимизированный для плотного графа	Алгоритм Беллмана–Форда
6	8	0,000002	0,000002	0,000001
18	56	0,000004	0,000003	0,000001
66	464	0,000012	0,000018	0,000005
258	3872	0,00006	0,000195	0,000041
1026	31808	0,000338	0,0030	0,000287
4098	258176	0,0043	0,0484	0,0076
16386	2081024	0,0300	0,7738	0,0535

Плотные графы

Для плотных графов E имеет порядок $O(V^2)$; например, полный граф с $n=|V|$ вершинами, в котором имеются ребра между любыми двумя вершинами, содержит $n\cdot(n-1)/2$ ребер. Использовать алгоритм Беллмана–Форда для таких плотных графов не рекомендуется, поскольку его производительность при этом вырождается в $O(V^3)$. Множество плотных графов, информация для которых содержится в табл. 6.2, взято из общедоступных наборов данных (`http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/`), используемых исследователями задачи коммивояжера. Мы выполнили по 100 испытаний и отбросили лучшие и худшие из них; в табл. 6.2 приведены средние значения оставшихся 98 попыток. Хотя между алгоритмами Дейкстры с очередью с приоритетами и для плотных графов разница невелика, наблюдается значительное улучшение производительности оптимизированного алгоритма Дейкстры. В последнем столбце показано время работы алгоритма Беллмана–Форда для тех же задач, но эти результаты в связи с быстрой деградацией его производительности являются средними значениями только для пяти выполнений. Урок, который можно извлечь из последнего столбца, заключается в том, что абсолютная производительность Беллмана–Форда для разреженных графов представляется вполне разумной, но при сравнении со своими конкурентами на плотных графах ясно видно, что этот алгоритм использовать для таких данных не стоит (если только в наличии нет ребер с отрицательным весом — в этом случае альтернативы алгоритму Беллмана–Форда нет).

Таблица 6.2. Время (в секундах) вычисления кратчайших путей из одной вершины в плотных графах

V	*E*	Алгоритм Дейкстры с очередью с приоритетами	Алгоритм Дейкстры, оптимизированный для плотного графа	Алгоритм Беллмана–Форда
980	479710	0,0681	0,0050	0,1730
1621	1313010	0,2087	0,0146	0,5090
6117	18705786	3,9399	0,2056	39,6780
7663	29356953	7,6723	0,3295	40,5585
9847	484476781	13,1831	0,5381	78,4154
9882	488224021	13,3724	0,5413	42,1146

Разреженные графы

Большие графы часто бывают разреженными, и результаты из табл. 6.3 подтверждают, что в этом случае лучше использовать алгоритм Дейкстры с очередью с приоритетами, а не реализации, созданные для плотных графов. Обратите внимание, насколько медленнее оказываются реализации для плотных графов. Строки в таблице отсортированы по количеству ребер в разреженных графах, поскольку, как представляется, они являются определяющим фактором стоимости в полученных результатах.

Таблица 6.3. Время (в секундах) вычисления кратчайших путей из одной вершины в больших разреженных графах

V	E	Плотность	Алгоритм Дейкстры, оптимизированный для плотного графа	Алгоритм Беллмана–Форда
3403	137845	2,4%	0,0102	0,0333
3243	294276	5,6%	0,0226	0,0305
19780	674195	0,34%	0,0515	1,1329

Кратчайшие пути между всеми парами вершин

Вместо кратчайших путей из одного источника часто ищут кратчайшие пути между любыми двумя вершинами (v_i, v_j); при этом может иметься несколько путей с одинаковым общим расстоянием. Быстрое решение этой задачи использует метод динамического программирования, представленный в главе 3, "Строительные блоки алгоритмов".

Динамическое программирование имеет два интересных свойства.

- Оно хранит результаты небольших, ограниченных версий задачи.
- Хотя мы ищем оптимальный ответ задачи, проще вычислить *значение* оптимального ответа, а не сам ответ. В нашем случае мы вычисляем для каждой пары вершин (v_i, v_j) длину кратчайшего пути от v_i к v_j и выполняем дополнительные вычисления для восстановления фактического пути. В приведенном ниже псевдокоде k, u и v представляют потенциальные вершины G.

Алгоритм Флойда–Уоршелла вычисляет матрицу `dist` размером $n \times n$ таким образом, что для всех пар вершин (v_i, v_j) `dist[i][j]` содержит длину кратчайшего пути от v_i к v_j.

На рис. 6.15 продемонстрирован пример применения алгоритма Флойда–Уоршелла к графу на рис. 6.13. Как можно убедиться, первая строка вычисляемой матрицы совпадает с вычисленным на рис. 6.13 вектором. Алгоритм Флойда–Уоршелла вычисляет кратчайший путь между всеми парами вершин.

Входные и выходные данные алгоритма

Входными данными для алгоритма Флойда–Уоршелла является ориентированный взвешенный граф $G=(V,E)$. Каждое ребро графа $e=(u,v)$ имеет связанный с ним положительный вес.

Алгоритм Флойда–Уоршелла вычисляет матрицу `dist[][]`, представляющую кратчайшие расстояния от каждой вершины u к каждой вне графа (включая саму эту вершину). Заметим, что если `dist[u][v]` равно ∞, то пути из u в v нет. Фактический кратчайший путь между двумя вершинами может быть вычислен из второй матрицы, `pred[][]`, которая также вычисляется данным алгоритмом.

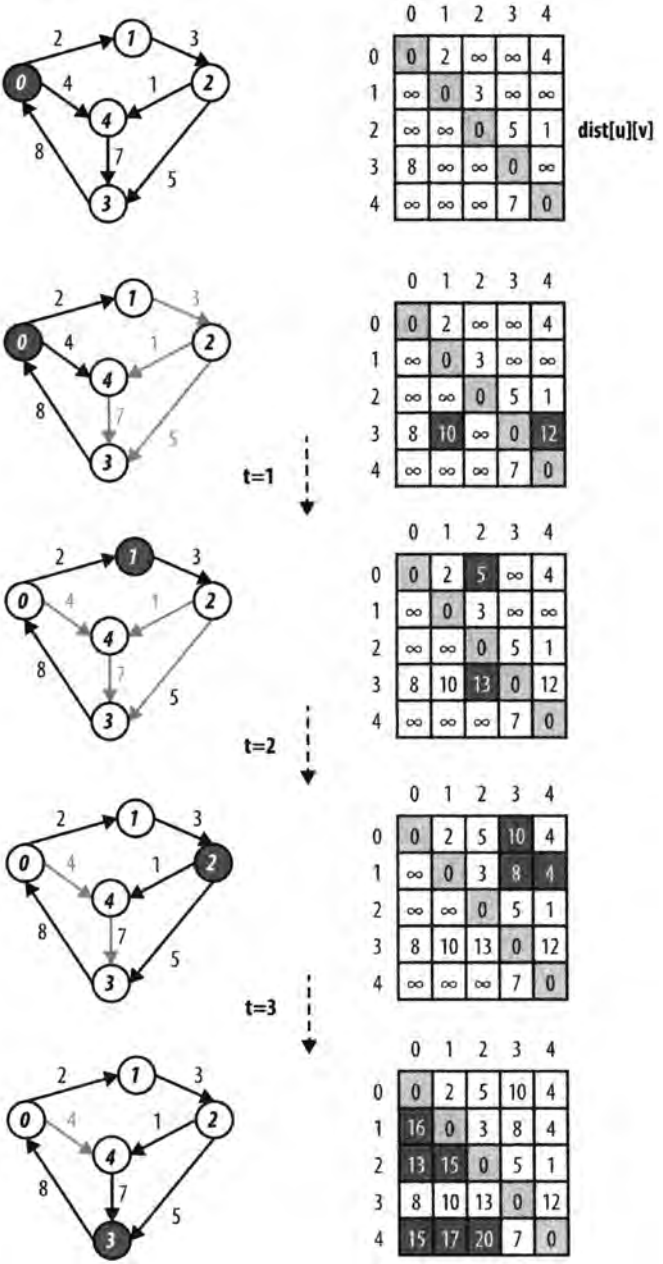


Рис. 6.15. Пример работы алгоритма Флойда–Уоршелла

Алгоритм Флойда–Уоршелла

Наилучший, средний и наихудший случаи: $O(V3)$

```
allPairsShortestPath (G)
    foreach u in V do
        foreach v in V do
            dist[u][v] = ∞                                    ❶
            pred[u][v] = -1
        dist[u][u] = 0
        foreach сосед v вершины u do
            dist[u][v] = вес ребра (u,v)
            pred[u][v] = u

    foreach k in V do
        foreach u in V do
            foreach v in V do
                newLen = dist[u][k] + dist[k][v]
                if newLen < dist[u][v] then                   ❷
                    dist[u][v] = newLen
                    pred[u][v] = pred[k][v]                   ❸
end
```

❶ Изначально все вершины рассматриваются как недостижимые.
❷ Если открыт более короткий путь из s в v, записываем новую длину.
❸ Запись нового предшественника.

Реализация алгоритма

Динамическое программирование поочередно вычисляет более простые подзадачи. Рассмотрим ребра графа G: они представляют длину кратчайшего пути между любыми двумя вершинами u и v, *который не включает никакие другие вершины*. Таким образом, `dist[u][v]` изначально имеет значение ∞, а `dist[u][u]` имеет значение "нуль" (поскольку путь от вершины к самой себе ничего не стоит). Наконец `dist[u][v]` получают значения веса для каждого ребра $(u,v) \in E$. В этот момент матрица `dist` содержит кратчайшие вычисленные пути (пока что только) для каждой пары вершин (u,v).

Теперь рассмотрим следующую по размеру подзадачу, а именно — вычисления длин кратчайших путей между любыми двумя вершинами u и v, которые *могут также включать вершину* v_1. Подход динамического программирования проверяет каждую пару вершин (u,v), чтобы выяснить, не является ли путь $\{u, v_1, v\}$ более коротким, чем имеющийся к этому моменту лучший результат. В некоторых случаях `dist[u][v]` по-прежнему равно ∞, потому что нет никакой информации о наличии каких-либо путей между u и v. Для других вершин сумма пути от u до v_1, а затем от v_1

до *v* оказывается меньшей, чем текущее значение; в таком случае алгоритм записывает в `dist[u][v]` это новое значение. Следующая, еще большая подзадача состоит в попытках вычислить длину кратчайшего пути между любыми двумя вершинами *u* и *v*, *который мог бы включать* v_1 или v_2. В конце концов алгоритм увеличивает эти подзадачи до тех пор, пока не получит в результате матрицу `dist[u][v]`, содержащую кратчайшие пути между любыми двумя вершинами *u* и *v*, которые могут включать также любую вершину графа.

Ключевое вычисление алгоритма Флойда-Уоршелла состоит в проверке справедливости соотношения `dist[u][k]+dist[k][v]<dist[u][v]`. Заметьте, что одновременно алгоритм обновляет элемент матрицы `pred[u][v]`, который "запоминает", что вновь создаваемый кратчайший путь от *u* до *v* должен проходить через вершину *k*. На удивление короткая реализация алгоритма на языке программирования C++ показана в примере 6.6.

Пример 6.6. Алгоритм Флойда-Уоршелла для вычисления кратчайших путей между всеми парами вершин

```
void allPairsShortest(Graph const& graph,          /* Вход  */
                      vector<vector<int>>& dist,   /* Выход */
                      vector<vector<int>>& pred)   /* Выход */
{
    int n = graph.numVertices();

    // Инициализация dist[][] нулями на диагонали, бесконечностью
    // там, где ребер нет, и весами ребер (u,v) для
    // элементов dist[u][v]. Массив pred инициализируется
    // соответствующим образом.
    for (int u = 0; u < n; u++)
    {
        dist[u].assign(n, numeric_limits<int>::max());
        pred[u].assign(n, -1);
        dist[u][u] = 0;

        for (VertexList::const_iterator ci = graph.begin(u);
                ci != graph.end(u); ++ci)
        {
            int v = ci->first;
            dist[u][v] = ci->second;
            pred[u][v] = u;
        }
    }

    for (int k = 0; k < n; k++)
    {
        for (int i = 0; i < n; i++)
        {
```

```
            if (dist[i][k] == numeric_limits<int>::max())
            {
                continue;
            }

            // Если найдено ребро, уменьшающее расстояние, обновляем
            // dist[][]. Вычисления используют тип long во избежание
            // переполнения для бесконечных расстояний.
            for (int j = 0; j < n; j++)
            {
                long newLen = dist[i][k];
                newLen += dist[k][j];

                if (newLen < dist[i][j])
                {
                    dist[i][j] = newLen;
                    pred[i][j] = pred[k][j];
                }
            }
        }
    }
}
```

Функция, показанная в примере 6.7, строит фактический кратчайший путь (их может быть несколько) из данной вершины *s* в вершину *t*. Функция работает путем восстановления информации о предшественнике из матрицы `pred`.

Пример 6.7. Восстановление кратчайшего пути с помощью информации из матрицы `pred[][]`

```
/** Выводит путь как список вершин от s к t для данной матрицы
 *  pred, полученной при выполнении allPairsShortest. Обратите
 *  внимание, что s и t должны быть корректными идентификаторами
 *  вершин. Если между s и t нет никакого пути, то возвращается
 *  пустой путь. */
void constructShortestPath(int s, int t,                    /* Вход  */
                           vector<vector<int>>const&pred, /* Вход  */
                           list<int>& path)                 /* Выход */
{
    path.clear();

    if (t < 0 || t >= (int) pred.size() || s < 0 ||
            s >= (int) pred.size())
    {
        return;
    }

    // Строим путь, пока не доберемся до 's' или до -1,
```

```
    // если пути не существует.
    path.push_front(t);

    while (t != s)
    {
        t = pred[s][t];

        if (t == -1)
        {
            path.clear();
            return;
        }

        path.push_front(t);
    }
}
```

Анализ алгоритма

Время работы алгоритма Флойда–Уоршелла зависит от количества вычислений функции минимизации, которое, как видно из трех вложенных циклов `for`, составляет $O(V^3)$. Функция `constructShortestPath` в примере 6.7 выполняется за время $O(E)$, так как кратчайший путь может включать все ребра графа.

Алгоритмы построения минимального остовного дерева

Для неориентированного связанного графа $G=(V,E)$ требуется найти такое подмножество ST его ребер из E, которое образует "остов" графа, соединяя все его вершины. Если, кроме того, потребовать, чтобы общий вес ребер в ST был наименьшим среди всех возможных остовных деревьев, можно получить задачу поиска минимального остовного дерева.

Алгоритм Прима показывает, как построить минимальное остновное дерево для такого графа с помощью жадного подхода, в котором каждый шаг алгоритма продвигает нас в направлении решения, без откатов к ранее полученным решениям. Алгоритм Прима растит остовное дерево T по одному ребру, пока не будет получено окончательное минимальное остновное дерево (и это остовное дерево доказуемо минимальное). Алгоритм случайным образом выбирает начальную вершину $s \in V$, принадлежащую растущему множеству S, и гарантирует, что T формирует дерево ребер в S. Жадность алгоритма Прима в том, что он постепенно добавляет ребра в T, пока не будет вычислено минимальное остовное дерево. В основе алгоритма лежит интуитивное представление о том, что ребро (u,v) с наименьшим весом между $u \in S$ и $v \in V–S$ должно

принадлежать к минимальному остовному дереву. Когда найдено такое ребро (u, v) с наименьшим весом, оно добавляется к T, а вершина v добавляется к S.

Алгоритм использует очередь с приоритетами для хранения вершин $v \in V–S$ со связанными с ними приоритетами, равными наименьшему весу среди всех весов ребер (u, v), где $u \in S$. Это значение отражает приоритет элемента в очереди с приоритетами; чем меньше значение веса, тем выше приоритет.

Алгоритм Прима

Наилучший, средний и наихудший случаи: $O((V+E)\cdot\log V)$

```
computeMST (G)
    foreach v in V do
        key[v] = ∞                                          ❶
        pred[v] = -1
    key[0] = 0
    PQ = пустая очередь с приоритетами
    foreach v in V do
        Вставка (v, key[v]) в PQ

    while PQ не пуста do
        u = getMin(PQ)                                      ❷
        foreach  ребро (u,v) из E do
            if PQ содержит v then
                w = вес ребра (u,v)
                if w < key[v] then                          ❸
                    pred[v] = u
                    key[v] = w
                    decreasePriority (PQ, v, w)
end
```

❶ Изначально все вершины рассматриваются как недостижимые.
❷ Находим вершину в V с наименьшим вычисленным расстоянием.
❸ Пересматриваем оценки стоимости для v и записываем ребро минимального остовного дерева в pred[v].

На рис. 6.16 продемонстрировано поведение алгоритма Прима для небольшого неориентированного графа. Очередь с приоритетом упорядочена по расстоянию от вершины в очереди до самой близкой вершины, уже содержащейся в минимальном остовном дереве.

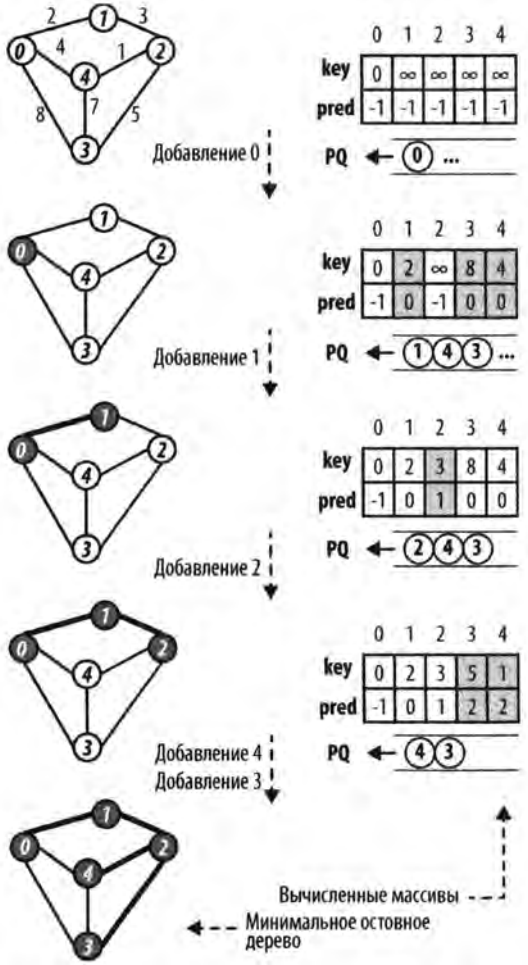


Рис. 6.16. Пример работы алгоритма Прима

Входные и выходные данные алгоритма

Входными данными для алгоритма является неориентированный граф $G=(V,E)$.

На выходе алгоритм выдает минимальное остовное дерево, закодированное с помощью массива `pred[]`. *Корнем* минимального остовного дерева является вершина, для которой `pred[v]=-1`.

Реализация алгоритма

Приведенная в примере 6.8 реализация алгоритма на C++ использует бинарную пирамиду для обеспечения реализации очереди с приоритетами, которая имеет важнейшее значение для алгоритма Прима. Обычно применение бинарной пирамиды неэффективно из-за проверки в главном цикле, является ли конкретная вершина

членом очереди (операция не поддерживается бинарной пирамидой). Однако алгоритм гарантирует, что каждая вершина удаляется из очереди с приоритетами только после ее обработки программой; он поддерживает массив состояний `inQueue[]`, который обновляется всякий раз, когда вершина извлекается из очереди с приоритетами. В другой оптимизации реализации алгоритм поддерживает внешний массив `key[]`, записывающий текущий приоритет для каждой вершины в очереди, что вновь устраняет необходимость поиска заданной вершины в очереди с приоритетами.

Алгоритм Прима случайным образом выбирает одну из вершин в качестве начальной вершины *s*. Когда минимальная вершина *u* удаляется из очереди с приоритетами и "добавляется" во множество посещенных вершин *S*, алгоритм использует имеющиеся ребра между *S* и растущим остовным деревом *T* для переупорядочения элементов в очереди. Напомним, что операция *decreasePriority* перемещает элемент ближе к началу очереди с приоритетами.

Пример 6.8. Реализация алгоритма Прима с использованием очереди с приоритетами

```
/** Для заданного неориентированного графа вычисляет минимальное
 *  остовное дерево, начиная со случайным образом выбранной вершины.
 *  Минимальное остовное дерево кодируется с помощью записей
 *  массива pred. */
void mst_prim(Graph const& graph, vector<int>& pred)
{
    // Инициализация массивов pred[] и key[]. Работа начинается с
    // произвольной вершины s=0. Очередь с приоритетами содержит
    // все вершины v из G.
    const int n = graph.numVertices();
    pred.assign(n, -1);
    vector<int> key(n, numeric_limits<int>::max());
    key[0] = 0;
    BinaryHeap pq(n);
    vector<bool> inQueue(n, true);

    for (int v = 0; v < n; v++)
    {
        pq.insert(v, key[v]);
    }

    while (!pq.isEmpty())
    {
        int u = pq.smallest();
        inQueue[u] = false;

        // Обработка всех соседей u в поисках
        // ребер с меньшим весом
        for (VertexList::const_iterator ci = graph.begin(u);
```

```
                ci != graph.end(u); ++ci)
        {
            int v = ci->first;

            if (inQueue[v])
            {
                int w = ci->second;

                if (w < key[v])
                {
                    pred[v] = u;
                    key[v] = w;
                    pq.decreaseKey(v, w);
                }
            }
        }
    }
}
```

Анализ алгоритма

Этап инициализации алгоритма Прима вставляет каждую вершину в очередь с приоритетами (реализованной с помощью бинарной пирамиды) за общее время $O(V \cdot log\ V)$. Операция `decreaseKey` в алгоритме Прима требует $O(log\ q)$ времени, где q — количество элементов в очереди, которое всегда будет меньше, чем V. Она может быть вызвана максимум $2 \cdot E$ раз, так как каждая вершина удаляется из очереди с приоритетами один раз и каждое неориентированное ребро графа посещается ровно два раза. Таким образом, общая производительность представляет собой $O((V+2 \cdot E) \cdot log\ V)$, или $O((V+E) \cdot log\ V)$.

Вариации алгоритма

Альтернативой для алгоритма Прима является алгоритм Крускала. Он использует структуру данных "непересекающихся множеств" для построения минимального остовного дерева путем обработки всех ребер графа в порядке их веса, начиная с ребра с наименьшим весом и заканчивая ребром с наибольшим весом. Алгоритм Крускала может быть реализован со временем работы $O(E \cdot log\ V)$. Сведения об этом алгоритме можно найти в [20].

Заключительные мысли о графах

В этой главе мы видели, как по-разному ведут себя алгоритмы в зависимости от того, каким является граф: разреженным или плотным. Сейчас мы продолжим исследование этой концепции для анализа точки разделения между разреженными и плотными графами и понимания ее влияния на требования к памяти.

Вопросы хранения

При использовании двумерных матриц смежности для представления потенциальных связей между n элементами множества матрица должна содержать n^2 хранимых элементов, но бывают ситуации, когда количество отношений намного меньше. В этих случаях — известных как *разреженные* графы — из-за ограниченной памяти компьютера может оказаться невозможным хранение больших графов с более чем несколькими тысячами вершин. Кроме того, проход по большим матрицам в поисках нескольких ребер в разреженных графах становится дорогостоящей операцией, так что данное представление не позволяет эффективным алгоритмам раскрыть их потенциал полностью.

Представления со списками смежностей, описанные в этой главе, содержат ту же самую информацию. Предположим, однако, что была написана программа для вычисления наиболее дешевых перелетов между любой парой городов в мире, обслуживаемых коммерческими рейсами. Вес ребра при этом будут соответствовать стоимости наиболее дешевого прямого рейса между этой парой городов. В 2012 году Международный совет аэропортов зарегистрировал в общей сложности 1 598 аэропортов по всему миру в 159 странах, в результате чего получается двумерная матрица с 2 553 604 записями. Вопрос о том, сколько из этих записей имеет некоторое значение стоимости, зависит от количества прямых рейсов. Международный совет аэропортов сообщил о 79 миллионах "перелетов" в 2012 году, что в день дает примерно 215 887 полетов. Даже если каждый перелет представляет собой единственный прямой рейс между двумя аэропортами (на самом деле количество прямых рейсов будет гораздо меньшим), то это означает, что матрица пуста на 92% — хороший пример разреженного графа!

Анализ графа

При применении алгоритмов из этой главы важным фактором, который определяет, следует ли использовать список или матрицу смежности, является разреженность графа. Мы вычисляем производительность каждого алгоритма в терминах количества вершин графа V и количества ребер графа E. Как часто бывает в литературе, посвященной алгоритмам, мы упрощаем представление формул, которые соответствуют наилучшему, среднему и наихудшему случаям, с помощью V и E в записи с использованием "большого O". Таким образом, $O(V)$ означает, что вычисление требует ряда шагов, количество которых прямо пропорционально количеству вершин в графе. Однако плотность ребер в графе также будет иметь значение. Так, $O(E)$ для разреженного графа составляет порядка $O(V)$, в то время как для плотного графа это значение ближе к $O(V^2)$.

Как мы увидим, производительность некоторых алгоритмов зависит от структуры графа; один из вариантов может выполняться за время $O((V+E)\cdot log\ V)$, в то

время как другой — за время $O(V^2+E)$. Какой из них является более эффективным? Как демонстрируется в табл. 6.4, ответ зависит от того, каким является граф G — разреженным или плотным. Для разреженных графов более эффективен вариант $O((V+E)\cdot log\ V)$, в то время как для плотных графов более эффективным вариантом является $O(V^2+E)$. Запись таблицы "Промежуточный граф" определяет тип графов, для которых ожидаемая производительность одинакова для обоих алгоритмов; в этих графах количество ребер составляет порядка $O(V^2/log\ V)$.

Таблица 6.4. Сравнение производительности двух версий алгоритма

Тип графа	$O((V+E)\cdot log\ V)$	Сравнение	$O(V^2+E)$
Разреженный граф: $\|E\|$ равно $O(V)$	$O(V\cdot log\ V)$	Меньше, чем	$O(V^2)$
Промежуточный граф: $\|E\|$ равно $O(V^2/log\ V)$	$O(V^2+V\cdot log\ V)=O(V^2)$	Равно	$O(V^2+V^2/log\ V)=O(V^2)$
Плотный граф: $\|E\|$ равно $O(V^2)$	$O(V^2\cdot log\ V)$	Больше, чем	$O(V^2)$

7

Поиск путей в ИИ

Чтобы решить задачу, когда нет ясного вычисления допустимых решений, мы переходим к поиску пути. В этой главе рассматриваются два связанных подхода к поиску путей: с помощью *деревьев игр* для игр с двумя игроками и *деревьев поиска* для игр для одного игрока. Эти подходы опираются на общую структуру, а именно — на дерево состояния, корневой узел которого представляет начальное состояние, а ребра представляют собой потенциальные ходы, которые преобразуют состояние в новое состояние. Поиск является сложной проблемой, поскольку базовая структура не вычисляется полностью из-за взрывного роста количества состояний. Например, у игры в шашки есть примерно $5 \cdot 10^{20}$ различных позиций на доске [54]. Таким образом, деревья, в которых выполняется поиск, строятся по требованию по мере необходимости. Два подхода к поиску пути характеризуются следующим образом.

Дерево игры

Два игрока по очереди выполняют ходы, которые изменяют состояние игры из ее первоначального состояния. Есть много состояний, в которых любой игрок может выиграть в игре. Могут быть некоторые состояния "ничьей", в которых не выигрывает никто. Алгоритмы поиска пути увеличивают шанс, что игрок выиграет или обеспечит ничью.

Дерево поиска

Один игрок начинает игру с некоторого начального состояния и делает допустимые ходы до достижения желаемого целевого состояния. Алгоритм поиска пути определяет точную последовательность ходов, которые преобразуют исходное состояние в целевое.

Дерево игры

Крестики-нолики — это игра на доске размером 3×3, в которой игроки поочередно ставят знаки X и O на доске. Первый игрок, который размещает три своих знака по горизонтали, вертикали или диагонали, выигрывает; игра заканчивается ничьей, если на поле нет пустых ячеек и ни один игрок не выиграл. В игре "крестики-нолики" есть только 765 уникальных позиций (без учета отражений и поворотов) и вычисленные 26830 возможных игр [53]. Чтобы увидеть некоторые из потенциальных игр, построим *дерево игры*, частично показанное на рис. 7.1, и найдем путь игрока O из текущего состояния игры (представленного в качестве верхнего узла дерева) в некоторые будущие состояния игры, которые обеспечивают для нашего игрока победу или ничью.

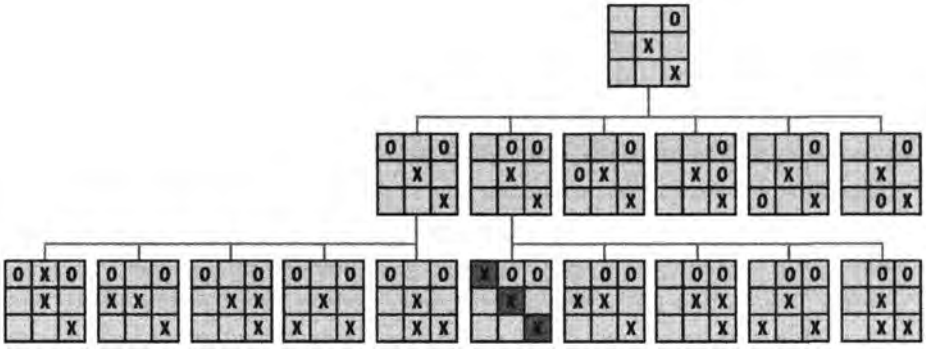


Рис. 7.1. Частичное дерево игры для данного начального состояния

Дерево игры известно также как дерево И/ИЛИ, поскольку оно образуется из двух различных типов узлов. Верхний узел является узлом ИЛИ, так как целью игрока O является выбор только одного из шести доступных ходов на среднем уровне. Узлы среднего уровня являются узлами И, потому что цель (с точки зрения игрока O) состоит в том, чтобы убедиться, что все ходы противника (показанные как дочерние узлы на нижнем уровне) по-прежнему приведут к победе O или ничьей. Дерево игры на рис. 7.1 раскрыто лишь частично, потому что на самом деле на нижнем уровне имеется 30 различных игровых состояний.

В сложной игре дерево игры никогда не может быть вычислено полностью из-за его размера. Целью алгоритма поиска пути является определение на основе состояния игры хода игрока, который максимизирует его шансы на победу в игре (или даже гарантирует ее). Таким образом, мы преобразуем множество решений игрока в *задачу поиска пути* в дереве игры. Этот подход работает для игр с небольшими деревьями, но его можно масштабировать и для решения более сложных задач.

При игре в шашки игроки играют на доске размером 8×8 с начальным набором из 24 шашек (12 белых и 12 черных). На протяжении десятилетий исследователи пытались определить, может ли игрок, делающий первый ход, обеспечить ничью или победу. Хотя точно вычислить размер дерева игры трудно, оно должно быть невероятно большим. После почти 18 лет вычислений (иногда с использованием целых 200 компьютеров) исследователи из Университета Альберты в Канаде показали, что идеальная игра обоих игроков приводит к ничьей [54].

Поиск пути в области искусственного интеллекта (ИИ) предоставляет конкретные алгоритмы для решения невероятно сложных задач, если они могут быть переведены в комбинаторную игру с чередующимися ходами игроков. Ранние исследователи ИИ [57] рассмотрели задачу построения машины для игры в шахматы и разработали два типа подходов для задач поиска, которые по-прежнему определяют сегодняшнее состояние дел.

Тип А

Рассмотрим различные разрешенные ходы для обоих игроков для фиксированного количества будущих ходов и определим наиболее благоприятное положение, получающееся в результате для исходного игрока. Затем выберем начальный ход, который движет игру в этом направлении.

Тип Б

Добавим некоторые адаптивные решения, основанные на знании игры, а не на статических оценках. Говоря более точно, *а*) оценим перспективные позиции на столько ходов вперед, сколько необходимо для выявления устойчивой позиции, в которой вычисления действительно отражают силу полученной позиции, и *б*) выберем подходящие доступные ходы. Этот подход пытается предотвратить возможность бессмысленной траты драгоценного времени.

В этой главе мы опишем семейство алгоритмов типа А, который предоставляет подход общего назначения для поиска в дереве игры наилучшего хода для игрока в игре с двумя игроками. Эти алгоритмы включают **Minimax**, **AlphaBeta** и **NegMax**.

Рассматриваемые в этой главе алгоритмы становятся излишне сложными, если лежащая в их основе информация плохо моделируется. Многие из примеров в учебниках или Интернете естественным образом описывают эти алгоритмы в контексте конкретной игры. Однако отделить представленную игру от основных элементов этих алгоритмов может оказаться трудной задачей. По этой причине мы намеренно разработали набор объектно-ориентированных интерфейсов для поддержания четкого разделения алгоритмов и игр. Далее мы кратко резюмируем основные интерфейсы в нашей реализации деревьев игр, которые показаны на рис. 7.2.

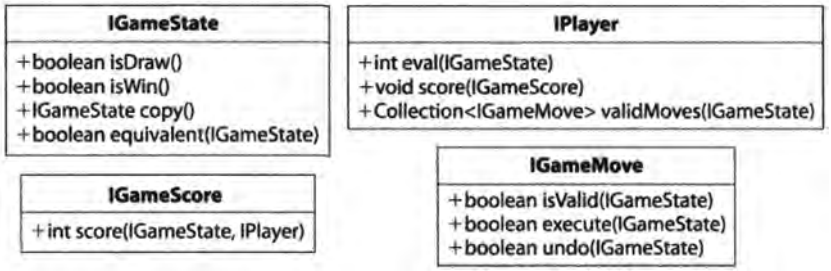


Рис. 7.2. Основные интерфейсы алгоритмов для работы с деревьями игр

Интерфейс `IGameState` абстрагирует основные концепции, необходимые для проведения поисков среди состояний игры. Он определяет, как выполнять следующие действия.

Интерпретировать состояние игры

`isDraw()` определяет, заканчивается ли игра ничьей; `isWin()` определяет, заканчивается ли игра выигрышем.

Управление состоянием игры

`copy()` возвращает идентичную копию состояния игры, так что ходы могут быть сделаны без обновления исходного состояния игры; `equivalent(IGameState)` определяет, одинаковы ли две позиции игры.

Интерфейс `IPlayer` абстрагирует возможности игрока управлять состоянием игры. Он определяет, как выполнять следующие действия.

Оценка позиции

`eval(IGameState)` возвращает целочисленную оценку состояния игры с точки зрения игрока; `score(IGameScore)` определяет вычисление счета, которое игрок использует при оценке состояния игры.

Генерация корректных ходов

`validMoves(IGameState)` возвращает множество доступных ходов для данного состояния игры.

Интерфейс `IGameMove` определяет, как ходы изменяют состояние игры. Классы ходов специфичны для конкретных задач, и алгоритм поиска не должен знать об их конкретной реализации. `IGameScore` определяет интерфейс для оценки счета состояний игры.

С точки зрения программирования ядром алгоритма поиска пути для дерева игры является реализация интерфейса `IEvaluation`, показанная в примере 7.1.

Пример 7.1. Общий интерфейс поиска путей в дереве игры

```
/**
 * Для заданных состояния игры, игрока и противника возвращает
 * наилучший ход игрока. Если доступных ходов нет, возвращает
 * нулевое значение.
 */
public interface IEvaluation
{
    IGameMove bestMove(IGameState state,
                       IPlayer player,
                       IPlayer opponent);
}
```

Для данного узла, представляющего текущее состояние игры, алгоритм вычисляет наилучший ход игрока в предположении, что противник будет идеально играть в ответ.

Функции статических оценок

Существует несколько способов сделать поиск более интеллектуальным [6].

Выбор порядка и количества применяемых разрешенных ходов

При рассмотрении доступных ходов для данного состояния игры сначала следует вычислить ходы, которые вероятнее других приведут к успешным результатам. Кроме того, можно отказаться от определенных ходов, которые заведомо не приводят к успешным результатам.

Выбор состояния игры для "обрезки" дерева поиска

По мере выполнения поиска может быть обнаружена новая информация, которую можно использовать для устранения состояний игры, которые (в некоторый момент времени) были выбраны в качестве части поиска.

Наиболее распространенным подходом является определение *функций статических оценок* для оценки состояния игры в промежуточных точках в процессе вычислений с последующим упорядочением множества доступных ходов так, чтобы сначала испытывались ходы, с более высокой вероятностью ведущие к выигрышу. Однако плохие функции статических оценок могут препятствовать выбору алгоритмом поиска пути наилучших ходов. Как говорится, подавая на вход ерунду, на выходе можно получить тоже только ерунду.

Функция статической оценки должна учитывать различные особенности позиции в дереве игры и возвращать целочисленный балл, который отражает относительную силу позиции с точки зрения игрока. Например, первая успешно играющая в шашки программа, разработанная Артуром Самуэлем (Arthur Samuel) [52], оценивала позицию на доске путем рассмотрения двух десятков параметров, таких как сравнение количества шашек у игрока и его противника или возможностей размена шашек

с противником без потерь. Очевидно, что более точная функция оценки делает программу более хорошим игроком.

В этой главе для игры в крестики-нолики мы используем функцию оценки `BoardEvaluation`, определенную Нилом Нильссоном (Nil Nilsson) [42]. Пусть $nc(gs,p)$ — количество строк, столбцов или диагоналей в состоянии игры gs, в которых игрок p все еще может выстроить в ряд три своих значка. Затем мы определим $score(gs,p)$ следующим образом:

- $+\infty$, если игрок p победил в игре в состоянии gs;
- $-\infty$, если противник игрока p победил в игре в состоянии gs;
- $nc(gs,) - nc(gs,opponent)$, если ни один игрок в состоянии игры gs не победил.

Вместо того чтобы ограничиваться при оценке текущим состоянием игры, функция оценки может временно расширить это состояние на фиксированное количество ходов и выбрать ход, который может в конечном итоге привести к состоянию игры с максимальной выгодой для игрока. На практике это применяется редко из-за *а*) затрат на выполнение операций и из-за *б*) совместного использования логики функцией оценки и функцией поиска, что нарушает четкое разделение их зон ответственности.

Концепции поиска путей

Описанные далее концепции применяются, как для деревьев игр с двумя игроками, в деревьях поиска для игр с одним игроком.

Представление состояния

Каждый узел дерева игры или дерева поиска содержит всю информацию о состоянии, известную как позиция в игре. Например, в шахматах король может рокироваться с ладьей, только если *а*) ни одна из этих фигур еще не делала хода, *б*) промежуточные клетки доски пустые и в настоящее время не находятся под боем фигур противника и *в*) король в настоящее время не находится под шахом. Обратите внимание, что условия *б*) и *в*) могут быть вычислены непосредственно из позиции на доске, и поэтому их не требуется хранить; однако состояние игры должно дополнительно сохранять информацию о том, были ли сделаны ходы королем или ладьей.

Для игр с экспоненциально большими деревьями состояние должно храниться настолько компактно, насколько это возможно. Если в состоянии имеется симметрия, как в играх "Четыре в ряд", "Реверси" или "Пятнашки", дерево может быть значительно уменьшено путем удаления эквивалентных состояний, которые можно получить простым поворотом или отражением. Более сложные представления используются, например, для шахмат или шашек, для того, чтобы справляться с невероятно большим количеством состояний с впечатляющей эффективностью [46].

Вычисление доступных ходов

Чтобы найти наилучший ход, в каждом состоянии должно быть возможно вычисление доступных ходов игрока. Термин *коэффициент ветвления* относится к общему количеству ходов, которые разрешены в любом отдельном состоянии. Например, оригинальный кубик Рубика 3×3 имеет (в среднем) коэффициент ветвления, равный 13,5 [35]. Популярная детская игра "Четыре в ряд" имеет для большинства состояний коэффициент ветвления, равный 7. Шашки оказываются сложнее из-за правила, что игрок обязан выполнить взятие, если оно возможно. На основе анализа большого количества шашечных баз данных было обнаружено, что для позиций со взятием коэффициент ветвления составляет в среднем 1,20, тогда как для позиций без взятия — 7,94; Шеффер (Schaeffer) оценивает средний коэффициент ветвления в шашках как 6,14 [54]. Игра Го имеет начальный коэффициент ветвления, равный 361, поскольку ее игровая доска имеет размер 19×19.

Алгоритмы чувствительны к порядку, в котором испытываются доступные ходы. Если коэффициент ветвления у игры оказывается высоким, а ходы не упорядочены должным образом на основе некоторой вычисляемой меры успеха, слепой поиск в дерева оказывается неэффективным.

Максимальная глубина расширения

Из-за ограниченных ресурсов памяти некоторые алгоритмы поиска ограничивает степень расширения деревьев поиска и игр. Этот подход проявляет свою слабость, в первую очередь, в играх, в которых продуманная стратегия формируется с помощью последовательности ходов. Например, в шахматах нередка жертва фигуры для получения потенциальных преимуществ. Если эта жертва происходит на краю максимального расширения, выгодное состояние игры может не быть обнаружено. Фиксированная глубина расширения формирует "горизонт", за который поиск не может заглянуть, и это часто препятствует успешному поиску. Для игр с одним игроком фиксация максимальной глубины означает, что алгоритм не в состоянии найти решение, которое лежит сразу за горизонтом.

Minimax

Для заданной конкретной позиции в дереве игры с точки зрения начального игрока программа поиска должна найти ход, который привел бы к наибольшим шансам на победу (или хотя бы на ничью). Но вместо только текущего состояния игры и доступных для этого состояния ходов программа должна рассматривать любые ходы противника, которые он делает после хода нашего игрока. Программа предполагает, что существует функция оценки `score(state, player)`, которая возвращает целое число, представляющее оценку состояния игры с точки зрения игрока `player`;

меньшие значения (которые могут быть и отрицательными) означают более слабые позиции.

Дерево игры расширяется путем рассмотрения будущих состояний игры после последовательности из *n* ходов. Каждый уровень дерева по очереди представляет собой уровень *MAX* (на котором цель заключается в обеспечении выгоды для исходного игрока путем максимизации вычисленного состояния игры) и *MIN* (на котором цель заключается в обеспечении выгоды для противника путем минимизации вычисленного состояния игры). На чередующихся уровнях программа выбирает ход, который максимизирует `score(state, initial)`, но на следующем уровне она предполагает, что противник будет выбирать ход, который минимизирует значение `score(state, initial)`.

Конечно, программа может просмотреть наперед только конечное количество ходов, потому что дерево игры потенциально бесконечное. Количество ходов, выбранное для просмотра, называется *предпросмотром* (ply). При выборе подходящего предпросмотра требуется компромисс, который обеспечит завершение поиска за разумное время.

Приведенный далее псевдокод иллюстрирует алгоритм **Minimax**.

Алгоритм Minimax

Наилучший, средний и наихудший случаи: $O(b^{ply})$

```
bestmove (s, player, opponent)
    original = player                                          ❶
    [move, score] = minimax (s, ply, player, opponent)
    return move
end

minimax (s, ply, player, opponent)
    best = [null, null]
    if ply равен 0 или допустимых ходов нет then               ❷
        score = оценка состояния s для исходного игрока
        return [null, score]
    foreach допустимый ход m игрока player в состоянии s do
        Выполнение хода m в состоянии s
        [move, score] = minimax(s, ply-1, opponent, player)    ❸
        Отмена хода m в состоянии s
        if player == original then                             ❹
            if score > best.score then best = [m, score]
        else
            if score < best.score then best = [m, score]
    return best
end
```

```
❶ Запоминаем исходного игрока, так как оценка состояния всегда
   выполняется с его точки зрения.
❷ Если больше ходов не остается, игрок побеждает (или проигрывает),
   что эквивалентно достижению целевой глубины предпросмотра.
❸ При каждом рекурсивном вызове выполняется обмен игрока и
   противника, отражающий чередование ходов.
❹ Последовательное чередование уровней между MAX и MIN.
```

На рис. 7.3 показана оценка хода с использованием Minimax с глубиной 3. Нижняя строка дерева игры содержит пять возможных состояний игры, которые являются результатом того, что игрок делает ход, противник отвечает и игрок делает еще один ход. Каждое из этих состояний игры оценивается с точки зрения исходного игрока и в каждом узле показан целочисленный рейтинг. Вторая строка снизу *MAX* содержит внутренние узлы, оценки которых представляют собой максимумы среди оценок их дочерних узлов. С точки зрения исходного игрока они представляют собой наилучшие результаты, которых он может достичь. Однако третья строка снизу *MIN* представляет собой наихудшие позиции, в которые может привести игрока его противник. Таким образом, здесь оценки узлов представляют собой минимумы оценок его дочерних узлов. Как можно видеть, уровни поочередно используют для оценок максимальные и минимальные оценки дочерних узлов. Окончательная оценка показывает, что исходный игрок может привести противника в состояние игры с оценкой 3.

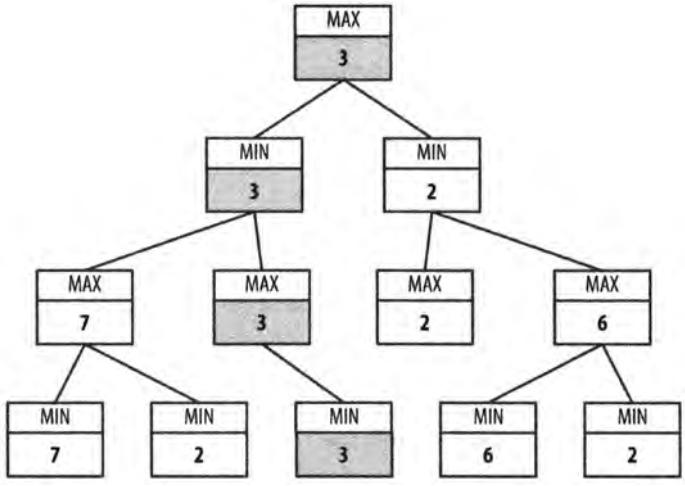


Рис. 7.3. Пример дерева игры для алгоритма Minimax

Входные и выходные данные алгоритма

Minimax выполняет предпросмотр фиксированного количества ходов, именуемого *глубиной*.

Minimax возвращает ход из числа допустимых, который для определенного игрока приводит к наилучшему будущему состоянию игры, определяемому функцией оценки.

Контекст применения алгоритма

Оценка состояния игры является сложным заданием, и для определения лучшего состояния игры мы должны прибегнуть к эвристическим оценкам. Действительно, разработка эффективной функции оценки для таких игр, как шахматы, шашки или "Реверси", является важнейшей задачей при разработке интеллектуальной программы. Мы предполагаем доступность этих функций оценки.

Размер дерева игры определяется количеством доступных ходов b в каждом состоянии игры. Для большинства игр мы можем лишь оценить значение b. В крестиках-ноликах (и других играх, таких как "Мельница") имеется b доступных ходов в начальном состоянии игры и каждый ход отнимает один потенциальный ход противника. Если глубина предпросмотра — d, то количество состояний игры, проверяемых для крестиков-ноликов, равно

$$\sum_{i=1}^{d} \frac{b!}{(b-i)!}$$

где $b!$ — факториал числа b. Для получения представления о масштабе этих чисел подсчитаем, что алгоритм Minimax оценивает 187 300 состояний при $b=10$ и $d=6$.

В процессе рекурсивного вызова в алгоритме Minimax функция `score(state, player)` должна последовательно применяться для *исходного игрока*, ход которого вычисляется. Тем самым согласовываются минимальные и максимальные рекурсивные оценки.

Реализация алгоритма

Вспомогательный класс `MoveEvaluation` объединяет `IMove` и целочисленную оценку, связываемую с этим ходом. Алгоритм Minimax выполняет исследование до фиксированной глубины предпросмотра или до состояния игры с отсутствием корректных ходов игрока. Код на языке программирования Java в примере 7.2 возвращает наилучший ход игрока в данном состоянии игры.

Пример 7.2. Реализация алгоритма Minimax

```
public class MinimaxEvaluation implements IEvaluation
{
    IGameState state; /** Изменяемое в процессе поиска состояние. */
    int ply;          /** Глубина предпросмотра. */
    IPlayer original; /** Все состояния вычисляются
                          с точки зрения этого игрока. */
```

```
public MinimaxEvaluation(int ply)
{
    this.ply = ply;
}

public IGameMove bestMove(IGameState s,
                          IPlayer player, IPlayer opponent)
{
    this.original = player;
    this.state = s.copy();
    MoveEvaluation me = minimax(ply, IComparator.MAX,
                                player, opponent);
    return me.move;
}

MoveEvaluation minimax(int ply, IComparator comp,
                       IPlayer player, IPlayer opponent)
{
    // Если нет разрешенных ходов или достигнут лист
    // дерева, возвращает оценку состояния игры.
    Iterator<IGameMove> it = player.validMoves(state).iterator();

    if (ply == 0 || !it.hasNext())
    {
        return new MoveEvaluation(original.eval(state));
    }

    // Попытка улучшить нижнюю границу (на основе селектора).
    MoveEvaluation best = new MoveEvaluation(comp.initialValue());

    // Генерация состояний игры в результате
    // корректных ходов игрока
    while (it.hasNext())
    {
        IGameMove move = it.next();
        move.execute(state);
        // Рекурсивная оценка позиции. Вычисление Minimax и
        // обмен игрока и противника синхронно с MIN и MAX.
        MoveEvaluation me = minimax(ply - 1, comp.opposite(),
                                    opponent, player);
        move.undo(state);

        // Выбор максимума (минимума) дочерних узлов
        // на уровне MAX (MIN)
        if (comp.compare(best.score, me.score) < 0)
        {
            best = new MoveEvaluation(move, me.score);
        }
```

```
        }

        return best;
    }
}
```

Селекторы *MAX* и *MIN* предназначены для корректной максимальной или минимальной оценки — в зависимости от текущих требований. Данная реализация упрощена путем определения интерфейса `IComparator`, показанного на рис. 7.4, который определяет *MAX* и *MIN* и объединяет выбор лучшего хода с их точек зрения. Переключение между селекторами *MAX* и *MIN* осуществляется с помощью метода `opposite()`. Наихудшая оценка для каждого из этих компараторов возвращается методом `initialValue()`.

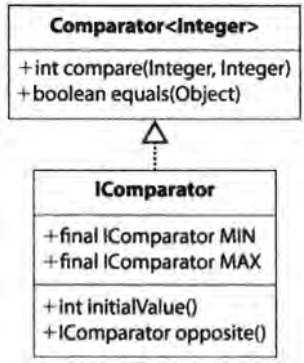


Рис. 7.4. Интерфейс `IComparator` абстрагирует операторы MAX и MIN

Алгоритм Minimax может быстро стать перегруженным огромным количеством состояний игры, создаваемых во время рекурсивного поиска. В шахматах, где среднее число ходов на доске равно 30 [36], предпросмотр только на пять ходов вперед (т.е. $b=30$, $d=5$) требует оценки 25 137 931 позиции на доске в соответствии с выражением

$$\sum_{i=0}^{d} b^i$$

Алгоритм Minimax может воспользоваться симметрий состояний игры, такой как повороты или отражения доски, или кешированием рассмотренных в прошлом состояний (и соответствующих им оценок), но такая экономия вычислений зависит от конкретной игры.

На рис. 7.5 показано исследование алгоритмом Minimax начального состояния игры в крестики-нолики для игрока `O` до глубины предпросмотра, равной 2. Чередование уровней *MAX* и *MIN* демонстрирует, что первый ход слева — размещение `O` в левом верхнем углу — это единственный ход, который предотвращает немедленный проигрыш. Обратите внимание, что расширяются все состояния игры даже

тогда, когда становится ясно, что соперник X может обеспечить себе выигрыш, если O сделает плохой выбор хода.

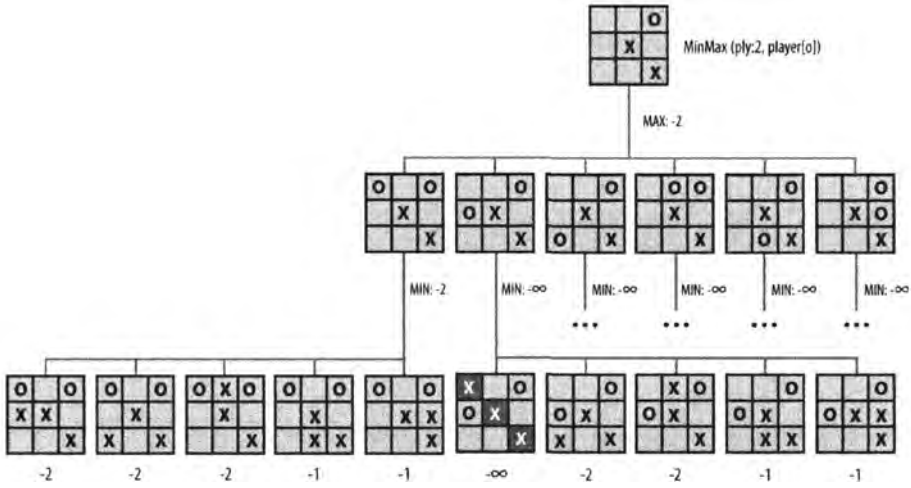


Рис. 7.5. Пример исследования алгоритма Minimax

Анализ алгоритма

Если для каждого состояния игры имеется фиксированное количество ходов b (или даже когда количество доступных ходов уменьшается на единицу с каждым уровнем), общее количество состояний игры, в которых выполняется поиск при предпросмотре глубиной d, представляет собой $O(b^d)$, демонстрируя экспоненциальный рост. Ограничения глубины предпросмотра могут быть устранены, если дерево игры достаточно мало для полной оценки за приемлемый промежуток времени.

Есть ли способ избавиться от исследования бесполезных состояний игры для результатов на рис. 7.5? Поскольку мы предполагаем, что и игрок, и противник играют без ошибок, мы должны найти способ остановить расширение дерева игры после того, как алгоритм определяет, что дальнейшее изучение данного поддерева бессмысленно. Эту возможность корректно реализует алгоритм AlphaBeta; но сначала объясним, как упростить чередование уровней дерева игры *MAX* и *MIN* с помощью алгоритма NegMax.

NegMax

Алгоритм **NegMax** заменяет чередование уровней *MAX* и *MIN* алгоритма Minimax единым подходом, используемым на каждом уровне дерева игры. Он также образует основу представленного далее алгоритма AlphaBeta.

В алгоритме Minimax состояние игры всегда оценивается с точки зрения игрока, делающего начальный ход (что требует от функции оценки хранения этой информации).

Таким образом, дерево игры состоит из чередующихся уровней, которые максимизируют оценку дочерних узлов (для исходного игрока) или минимизируют ее (для противника). Алгоритм NegMax вместо этого последовательно ищет ход, который дает максимум из значений дочерних узлов состояния с обратным знаком.

Алгоритм NegMax

Наилучший, средний и наихудший случаи: $O(b^{ply})$

```
bestmove (s, player, opponent)
    [move, score] = negmax (s, ply, player, opponent)
    return move
end

negmax (s, ply, player, opponent)
    best = [null, null]
    if ply == 0 или больше нет допустимых ходов then
        score = Вычисление s для игрока
        return [null, score]

    foreach Допустимый ход m для игрока в состоянии s do
        Выполнение хода m для s
        [move, score] = negmax (s, ply-1, opponent, player)     ❶
        Отмена хода m для s
        if -score > best.score then best = [m, -score]          ❷
    return best
end
```

❶ Алгоритм NegMax обменивает игроков на каждом следующем уровне.
❷ Выбор наибольшей среди оценок дочерних узлов с обратным знаком.

Интуитивно после того, как игрок сделал свой ход, противник будет пытаться сделать свой наилучший ход. Таким образом, чтобы найти наилучший ход для игрока, следует выбрать тот, который ограничивает противника от слишком высокой оценки. Если вы сравните псевдокоды, то увидите, что алгоритмы Minimax и NegMax создают два дерева игры с одинаковой структурой; единственная разница проявляется в оценке состояний игры.

Структура дерева игры NegMax идентична структуре дерева игры Minimax потому, что они находят одни и те же ходы; единственное отличие состоит в том, что значения на уровнях, ранее помечавшихся как *MIN*, в алгоритме NegMax получаются с обратным знаком. Если вы сравните дерево на рис. 7.6 с деревом на рис. 7.3, то увидите это поведение.

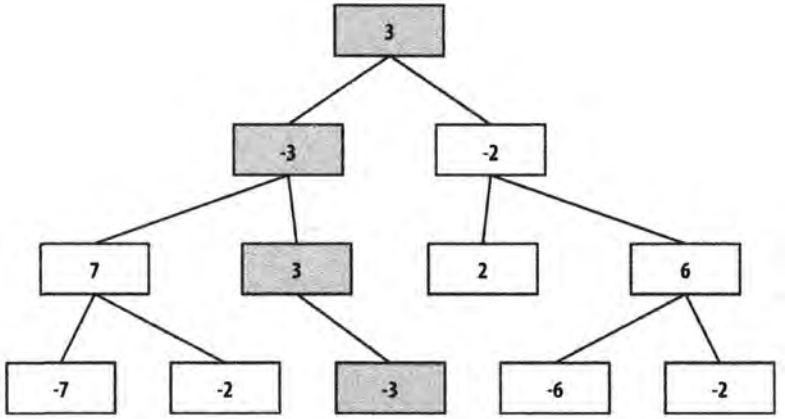


Рис. 7.6. Пример дерева игры для алгоритма NegMax

Реализация алгоритма

В примере 7.3 обратите внимание на то, что оценка для каждого `MoveEvaluation` представляет собой просто оценку состояния игры с точки зрения игрока, делающего данный ход. Переориентация каждой оценки по отношению к игроку, делающему ход, упрощает реализацию алгоритма.

Пример 7.3. Реализация алгоритма NegMax

```
public class NegMaxEvaluation implements IEvaluation
{
    IGameState state; /** Изменяемое в процессе поиска состояние. */
    int ply;          /** Глубина предпросмотра. */

    public NegMaxEvaluation(int ply)
    {
        this.ply = ply;
    }

    public IGameMove bestMove(IGameState s, IPlayer player,
                              IPlayer opponent)
    {
        state = s.copy();
        MoveEvaluation me = negmax(ply, player, opponent);
        return me.move;
    }

    public MoveEvaluation negmax(int ply, IPlayer player,
                                 IPlayer opponent)
    {
        // Если нет разрешенных ходов или достигнут лист
        // дерева, возвращает оценку состояния игры.
```

```
        Iterator<IGameMove> it = player.validMoves(state).iterator();

        if (ply == 0 || !it.hasNext())
        {
            return new MoveEvaluation(player.eval(state));
        }

        // Попытка улучшить нижнюю границу .
        MoveEvaluation best =
                new MoveEvaluation(MoveEvaluation.minimum());

        // Генерация состояний игры в результате корректных ходов
        // игрока. Выбор максимальной из оценок дочерних узлов
        // с обратным знаком.
        while (it.hasNext())
        {
            IGameMove move = it.next();
            move.execute(state);
            // Рекурсивная оценка позиции с помощью negmax.
            MoveEvaluation me = negmax(ply - 1, opponent, player);
            move.undo(state);

            if (-me.score > best.score)
            {
                best = new MoveEvaluation(move, -me.score);
            }
        }

        return best;
    }
}
```

Польза алгоритма NegMax заключается в том, что он готовит простую основу для расширения до алгоритма AlphaBeta. Поскольку оценки позиций в данном алгоритме регулярно меняют знак, мы должны тщательно выбирать значения, представляющие состояния победы и поражения. В частности, минимальное значение должно быть максимальным значением с обратным знаком. Обратите внимание, что значение `Integer.MIN_VALUE` (определенное в Java как `0x80000000`, или -2 147 483 648) не является значением `Integer.MAX_VALUE` (определенным в Java как 2 147 483 647 или `0x7fffffff`) с обратным знаком. По этой причине мы используем в качестве минимального значения, которое получается с помощью статической функции `MoveEvaluation.minimum()`, значение `Integer.MIN_VALUE+1`. Для полноты мы также предоставляем функцию `MoveEvaluation.maximum()`.

На рис. 7.7 показан пример исследования начального состояния игры в крестики-нолики для игрока `O` до глубины предпросмотра, равной 2 с помощью алгоритма

NegMax. NegMax расширяет все возможные состояния игры, даже когда становится ясно, что соперник X может обеспечить выигрыш, если O сделает плохой ход. Оценки, связанные с каждым из листьев, оцениваются с точки зрения игрока (в данном случае — исходного игрока O). Оценка первоначального состояния игры равна –2, потому что это — максимальная из оценок дочерних узлов с обратным знаком.

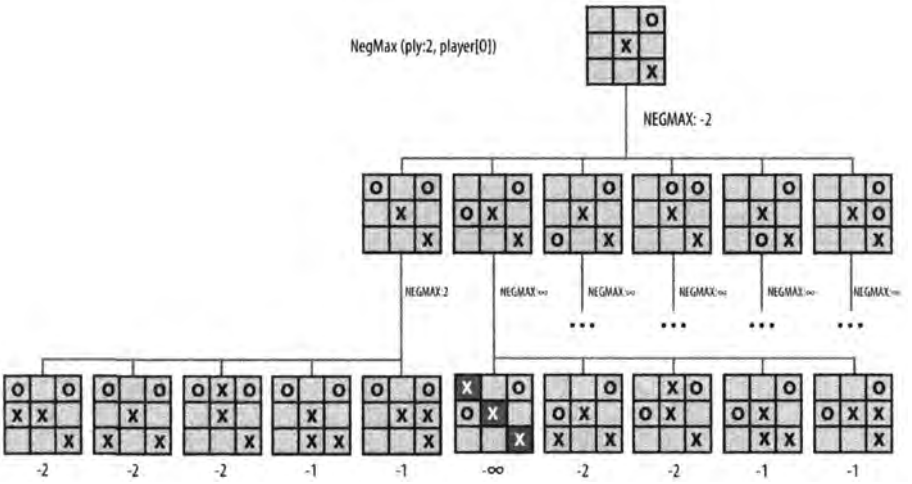


Рис. 7.7. Пример исследования алгоритма NegMax

Анализ алгоритма

Число состояний, изучаемых NegMax, такое же, как и у алгоритма Minimax, и имеет порядок b^d для глубины предпросмотра d при фиксированном количестве ходов b в каждом состоянии игры. Во всех других отношениях он работает так же, как алгоритм Minimax.

AlphaBeta

Алгоритм Minimax оценивает лучший ход игрока при рассмотрении ходов противника, но эта информация не используется при генерации дерева игры! Рассмотрим функцию оценки `BoardEvaluation`, введенную ранее. Вспомните рис. 7.5, который показывает частичное расширение дерева игры из первоначального состояния игры после того, как игрок X сделал два хода, а игрок O — только один.

Обратите внимание, как алгоритм Minimax кропотливо выполняет свою работу, несмотря даже на то, что каждый последующий поиск обнаруживает проигрышную позицию, в которой X имеет возможность завершить диагональ. Оцениваются в общей сложности 36 узлов. Алгоритм Minimax не использует преимущества того факта, что первоначальное решение O сделать ход в левый верхний угол не позволяет X

добиться немедленной победы. Алгоритм **AlphaBeta** определяет стратегию последовательного обрезания всех непродуктивных поисков в дереве.

После оценки поддерева игры с корнем в (1) на рис. 7.8 алгоритм AlphaBeta знает, что если этот ход сделан, то противник не может сделать позицию хуже, чем -3. Это означает, что лучшее, что может сделать игрок, — добиться состояния с оценкой 3. Когда AlphaBeta переходит в состояние игры (2), то его первый дочерний узел — состояние (3) — имеет оценку, равную 2. Это означает, что если выбрать ход, приводящий в (2), то противник может заставить игрока перейти в состояние игры с оценкой, меньшей, чем лучшая найденная к этому моменту (т.е. 3). Таким образом, не имеет смысла проверять поддерево с корнем в (4), и оно полностью отбрасывается.

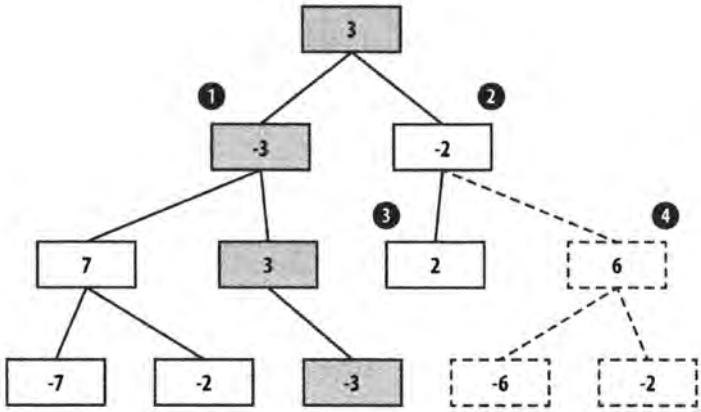


Рис. 7.8. Пример дерева игры алгоритма AlphaBeta

На рис. 7.9 показано, как выглядит расширение уже знакомого нам дерева игры при использовании алгоритма AlphaBeta.

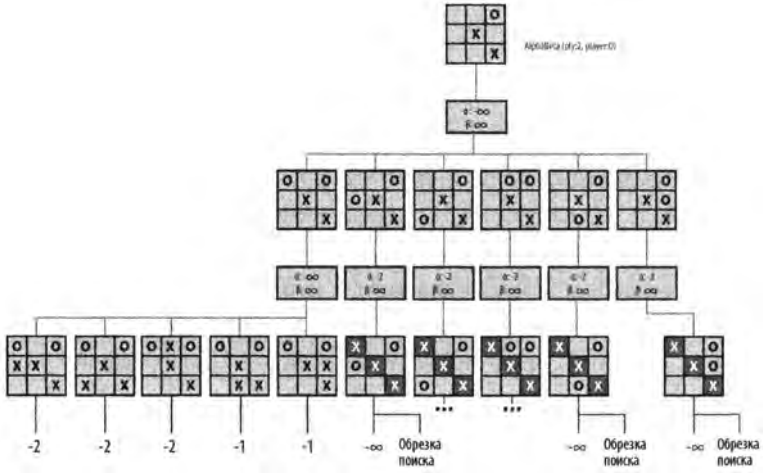


Рис. 7.9. Предпросмотр алгоритма AlphaBeta глубиной 2

Когда алгоритм AlphaBeta ищет лучший ход на рис. 7.9, он помнит, что X может достичь оценки не выше 2, если O делает ход в левый верхний угол. Для каждого другого хода O AlphaBeta определяет, что X имеет хотя бы один ход, который превосходит первый ход O (на самом деле для всех прочих ходов O игрок X может победить). Таким образом, дерево игры расширяется только до 16 узлов, что представляет собой экономию более 50% по сравнению с алгоритмом Minimax. Алгоритм AlphaBeta выбирает тот же ход, что и алгоритм Minimax, но с потенциально более высокой производительностью.

Алгоритм AlphaBeta рекурсивно выполняет поиск в дереве игры и поддерживает два значения, α и β, которые определяют "окно возможностей" для игрока до тех пор, пока $\alpha < \beta$. Значение α представляет нижнюю границу состояний игры, найденную для игрока до настоящего времени (или $-\infty$, если ничего не было найдено) и объявляет, что игрок нашел ход, гарантирующий, что он может достичь как минимум указанного значения. Более высокие значения α означают, что игрок поступает правильно; при $\alpha = +\infty$ игрок выиграл, и поиск можно завершать.

Значение β представляет верхнюю границу состояний игры, найденную для игрока до настоящего времени (или $+\infty$, если ничего не было найдено) и объявляет максимальное значение, которого игрок может достичь. Если значение β уменьшается все сильнее и сильнее, значит, противник поступает верно и делает наилучшие ходы, ограничивающие доступные для игрока варианты. Поскольку алгоритм AlphaBeta имеет максимальную глубину предпросмотра, далее которой поиск выполняться не будет, любые принимаемые им решения ограничены этой областью.

Алгоритм AlphaBeta

Наилучший и средний случаи: $O(b^{ply/2})$; наихудший случай: $O(b^{ply})$

```
bestmove (s, player, opponent)
    [move, score] = alphaBeta (s, ply, player, opponent, -∞, ∞)    ❶
    return move
end

alphaBeta (s, ply, player, opponent, low, high)
    best = [null, null]
    if ply == 0 или нет допустимых ходов then                      ❷
        score = вычисление s для игрока player
        return [null, score]

    foreach Корректный ход m для игрока player в состоянии s do
        execute move m on s
        [move,score] = alphaBeta(s,ply-1,opponent,player,-high,-low)
        Отмена хода m для s
```

```
        if -score > best.score then
            low = -score
            best = [m, -low]
        if low ≥ high then return best                                    ❸
    return best
end

❶ В начале худшее, что может сделать игрок, — проиграть (low = -∞).
   Лучшее, что может сделать игрок, — выиграть (high = +∞).
❷ AlphaBeta оценивает листья, как и алгоритм NegMax.
❸ Прекращение исследования "братьев", когда наихудшая возможная
   оценка противника не ниже нашего максимального порога.
```

В дереве игры на рис. 7.9 показаны значения $[\alpha,\beta]$, вычисляемые по мере выполнения алгоритма AlphaBeta. Первоначально они равны $[-\infty,\infty]$. При предпросмотре глубиной 2 алгоритм AlphaBeta пытается найти наилучший ход O, рассматривая только непосредственный ответный ход X.

Поскольку алгоритм AlphaBeta рекурсивный, мы можем отследить его работу, рассматривая обход дерева игры. Первый ход, который рассматривает AlphaBeta для O, — в левый верхний угол. После оценки всех пяти ответных ходов X очевидно, что X может обеспечить для себя только оценку -2 (с использованием функции статической оценки `BoardEvaluation` для крестиков-ноликов). Когда AlphaBeta рассматривает второй ход O (в середину левого столбца), его значения $[\alpha,\beta]$ теперь представляют собой $[-2,\infty]$, что означает "худшее, чего пока что может добиться O, — состояния с оценкой -2, а лучшее — все еще выиграть игру". Но когда вычисляется первый ответный ход X, алгоритм AlphaBeta обнаруживает, что X выигрывает. Это выходит за рамки нашего "окна возможностей", так что дальнейшие ответные ходы X рассматривать больше не нужно.

Чтобы пояснить, как алгоритм **AlphaBeta** отсекает ветви дерева игры для устранения непродуктивных узлов, взгляните на рис. 7.10, на котором представлен поиск, начатый на рис. 7.5, с глубиной 3, который раскрывает 66 узлов (в то время как соответствующему дереву игры алгоритма Minimax потребовалось бы 156 узлов).

В исходном узле *n* в дереве игры игрок O должен рассмотреть один из шести потенциальных ходов. Обрезка может произойти как на ходу игрока, так и на ходу противника. В поиске, показанном на рис. 7.10, есть два таких примера.

Ход игрока

Предположим, что O ходит в средину левого столбца, а X отвечает ходом в середине верхней строки (крайний слева внук корневого узла дерева поиска). С точки зрения O лучшая оценка, которой может достичь O, равна -1 (обратите внимание, что в диаграмме оценка отображаются как 1, потому что AlphaBeta использует тот же механизм оценки, что и у NegMax). Это значение запоминается,

когда алгоритм пытается определить, чего может достичь O при ответном ходе X в средине нижней строки. Обратите внимание, что [α,β] теперь равно [−∞,−1]. AlphaBeta оценивает результат хода O в средину верхней строки и вычисляет оценку 1. Поскольку это значение больше или равно значению −1, оставшиеся три потенциальных хода O на этом уровне игнорируются.

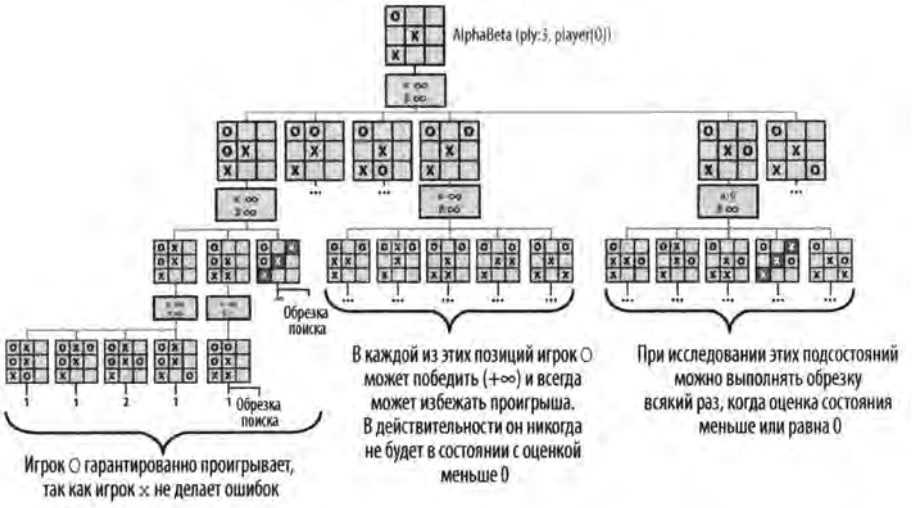


Рис. 7.10. Поиск алгоритма AlphaBeta глубиной 3

Ход противника

Предположим, что игрок O ходит в средину левого столбца, а X отвечает ходом в правый верхний угол, побеждая в игре. Алгоритм AlphaBeta может игнорировать два других возможных хода X, поскольку O отсекает оставшиеся узлы в поддереве поиска с корнем в решении ходить в средину левого столбца.

Обрезка поиска происходит при $\alpha \geq \beta$, или, иными словами, когда "окно возможностей" закрывается. Когда алгоритм AlphaBeta основан на алгоритме Minimax, есть два способа обрезки поиска, известные как обрезка α и обрезка β; в более простых вариациях AlphaBeta на основе NegMax эти два случая объединяются в один, обсуждаемый здесь. Поскольку алгоритм AlphaBeta рекурсивный, диапазон [α,β] представляет окно возможностей для игрока, а окно возможностей для противника представляет собой [−β,−α]. В рекурсивном вызове алгоритма AlphaBeta игрок и соперник меняются местами и аналогично меняются местами и окна.

Реализация алгоритма

Реализация алгоритма AlphaBeta в примере 7.4 дополняет реализацию NegMax прекращением оценки состояний игры, как только становится ясно, что либо игрок

не может гарантировать лучшую позицию (α-обрезка), либо противник не может обеспечить худшее положение (β-обрезка).

Пример 7.4. Реализация алгоритма AlphaBeta

```
public class AlphaBetaEvaluation implements IEvaluation
{
    IGameState state; /** Состояние, модифицируемое
                          в процессе поиска. */
    int ply;          /** Глубина предпросмотра. */

    public AlphaBetaEvaluation(int ply)
    {
        this.ply = ply;
    }

    public IGameMove bestMove(IGameState s,
                              IPlayer player, IPlayer opponent)
    {
        state = s.copy();
        MoveEvaluation me = alphabeta(ply, player, opponent,
                                      MoveEvaluation.minimum(),
                                      MoveEvaluation.maximum());
        return me.move;
    }

    MoveEvaluation alphabeta(int ply, IPlayer player,
                             IPlayer opponent,
                             int alpha, int beta)
    {
        // Если ходов нет, возвращает оценку
        // позиции с точки зрения игрока.
        Iterator<IGameMove> it = player.validMoves(state).iterator();

        if (ply == 0 || !it.hasNext())
        {
            return new MoveEvaluation(player.eval(state));
        }

        // Выбор "максимума значений дочерних узлов с обратным
        // знаком", улучшающего значение "альфа"
        MoveEvaluation best = new MoveEvaluation(alpha);

        while (it.hasNext())
        {
            IGameMove move = it.next();
            move.execute(state);
            MoveEvaluation me = alphabeta(ply - 1, opponent,
```

```
                                        player, -beta, -alpha);
            move.undo(state);

            // При улучшении "альфа" отслеживать этот ход.
            if (-me.score > alpha)
            {
                alpha = -me.score;
                best = new MoveEvaluation(move, alpha);
            }

            if (alpha >= beta)
            {
                return best;    // Дальнейший поиск непродуктивен.
            }
        }
        return best;
    }
}
```

Найденные ходы будут точно такими же, как и найденные алгоритмом Minimax. Но поскольку при расширении дерева многие состояния будут удалены, время выполнения алгоритма AlphaBeta заметно меньше.

Анализ алгоритма

Чтобы измерить преимущество алгоритма AlphaBeta перед алгоритмом NegMax, сравним размер соответствующих деревьев игр. Это сложная задача, потому что алгоритм AlphaBeta демонстрирует свои самые впечатляющие результаты, если при каждом выполнении AlphaBeta сначала вычисляются лучшие ходы противника. При наличии фиксированного количества ходов b в каждом состоянии игры общее число потенциальных состояний игры для поиска глубиной d имеет порядок b^d. Если ходы упорядочиваются в направлении уменьшения их ценности (т.е. наилучшим является первый ход), то мы в любом случае должны оценить все b дочерних узлов для начинающего игру игрока (потому что мы должны выбрать наилучший ход); однако в лучшем случае нам нужно оценить только первый ход противника. Обратите внимание на рис. 7.9, что из-за упорядочения ходов обрезка выполняется после того, как были оценены несколько ходов, так что упорядочение ходов для этого дерева игры является неоптимальным.

Таким образом, в наилучшем случае алгоритм AlphaBeta оценивает b состояний игры для начинающего игрока на каждом уровне, но только одно состояние игры — для его противника. Так что вместо расширения $b{\cdot}b{\cdot}b{\cdot}\ldots{\cdot}b{\cdot}b$ (всего d раз) состояний игры на d-м уровне дерева алгоритму AlphaBeta могут потребоваться только $b{\cdot}1{\cdot}b{\cdot}\ldots{\cdot}b{\cdot}1$ (всего d раз) состояний. Получаемое в результате количество состояний игры равно $b^{d/2}$, что весьма впечатляет.

Вместо того чтобы просто пытаться свести к минимуму количество состояний игры, алгоритм AlphaBeta может также исследовать то же общее количество состояний игры, что и алгоритм Minimax. Это расширит глубину дерева игры до $2 \cdot d$, удваивая, таким образом, выполняемый алгоритмом предпросмотр.

Чтобы эмпирически оценить Minimax и AlphaBeta, мы построили набор первоначальных позиций игры в крестики-нолики, возможные после k ходов. Затем мы запустили алгоритмы Minimax и AlphaBeta с глубиной предпросмотра $9-k$, которая гарантирует, что просмотрены все возможные ходы. Результаты показаны в табл. 7.1. Обратите внимание на значительное уменьшение количества исследованных состояний при использовании AlphaBeta.

Таблица 7.1. Сравнение статистики алгоритмов Minimax и AlphaBeta

Глубина	Состояния Minimax	Состояния AlphaBeta	Совокупное уменьшение
6	549 864	112 086	80%
7	549 936	47 508	91%
8	549 945	27 565	95%

Отдельные сравнения показывают резкое улучшение AlphaBeta по сравнению с Minimax; некоторые из этих случаев объясняют, почему AlphaBeta настолько мощный алгоритм. Для состояния игры, показанного на рис. 7.11, AlphaBeta исследует только 450 состояний (вместо 8 232 для Minimax; уменьшение — на 94,5%), чтобы определить, что игрок X должен выбрать центральный квадрат, после чего ему гарантирован выигрыш.

Однако единственный способ добиться такого глубоких сокращения — упорядочение ходов таким образом, чтобы лучшие из них были первыми. Так как наше решение для игры в крестики-нолики не упорядочивает ходы указанным образом, это приводит к некоторым аномалиям. Например, если ту же позицию, которую мы только что рассматривали, повернуть на 180° (рис. 7.12), алгоритм AlphaBeta исследует 960 состояний игры (уменьшение — на 88,3%), так как расширение дерева игры использует другой порядок допустимых ходов. По этой причине алгоритмы поиска часто переупорядочивают ходы с использованием функции статической оценки для уменьшения размера дерева игры.

Рис. 7.11. Пример позиции после двух ходов

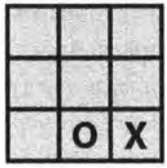


Рис. 7.12. Пример позиции после двух ходов — поворот рис. 7.11

Деревья поиска

Игры с единственным игроком подобны деревьям игр, обладая начальным состоянием (верхний узел в дереве поиска) и последовательностью ходов, которая изменяет позицию на доске до тех пор, пока не будет достигнуто целевое состояние. *Дерево поиска* представляет набор промежуточных состояний доски в процессе выполнения алгоритма поиска пути. Компьютерная структура представляет собой дерево, так как алгоритм гарантирует, что он не посещает никакое состояние доски дважды. Алгоритм выясняет, в каком порядке следует посещать состояния доски в попытках достичь целевого состояния.

Мы будем исследовать деревья поиска на примере головоломки "8", которая играет на доске размером 3×3, содержащей восемь квадратных плиток, пронумерованных от 1 до 8, и пустое пространство, в котором плитки нет. Соседняя с ним плитка (по горизонтали или вертикали) может быть перемещена путем сдвига в пустое пространство. Цель заключается в том, чтобы, начав с полученного путем перетасовки плиток начального состояния и перемещая плитки, достичь целевого состояния. 8-ходовое решение на рис. 7.13 показано как выделенный цветом путь от первоначального узла к целевому.

Деревья поиска могут обладать взрывным ростом и (потенциально) содержать миллиарды и триллионы состояний. Рассматриваемые в этой главе алгоритмы описывают, как выполнить эффективный поиск в этих деревьях быстрее, чем при использовании слепого поиска. Чтобы описать сложность проблемы, мы рассмотрим **поиск в глубину** и **поиск в ширину** как два возможных подхода к алгоритмам поиска пути. Затем мы представим мощный алгоритм **A*Search** для нахождения решения с минимальной стоимостью (при определенных условиях). Сейчас мы кратко подытожим основные классы, показанные на рис. 7.14, которые будут использоваться при обсуждении алгоритмов поиска в дереве.

Интерфейс `INode` абстрагирует основные концепции, необходимые для выполнения поиска среди состояний доски.

Генерация корректных ходов

`validMoves()` возвращает список доступных ходов для данного состояния доски.

Вычисление состояний доски

`score(int)` связывает с состоянием доски целочисленную оценку, представляющую результат функции вычисления; `score()` возвращает результат оценки, ранее связанный с данным состоянием доски.

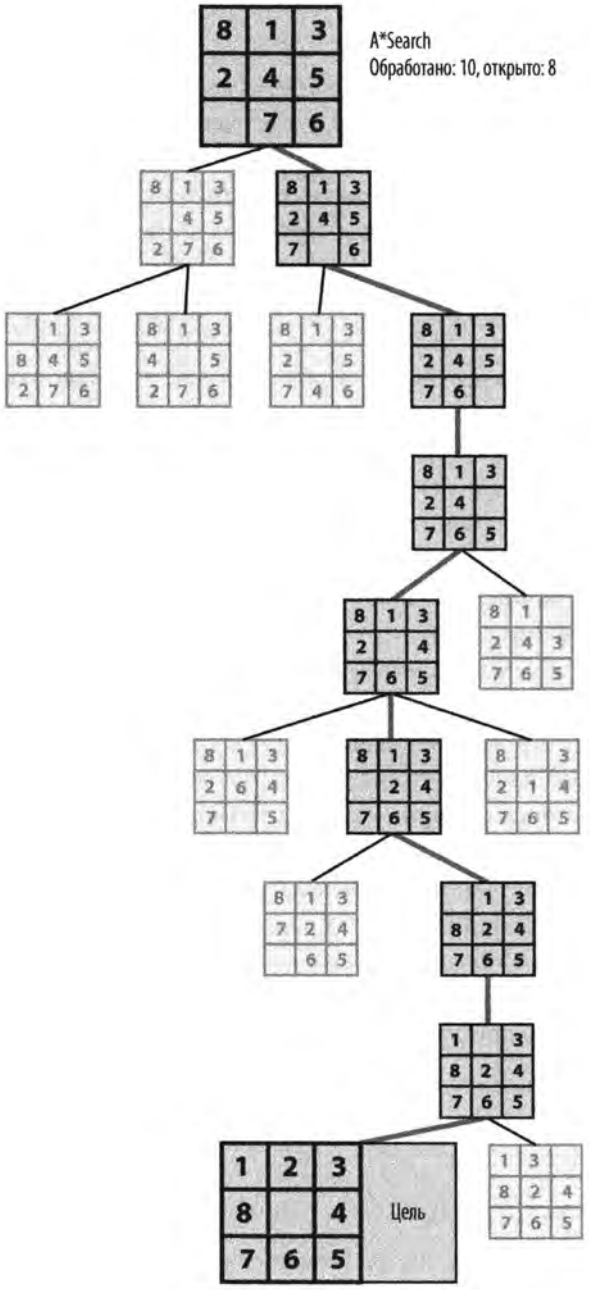


Рис. 7.13. Простая головоломка

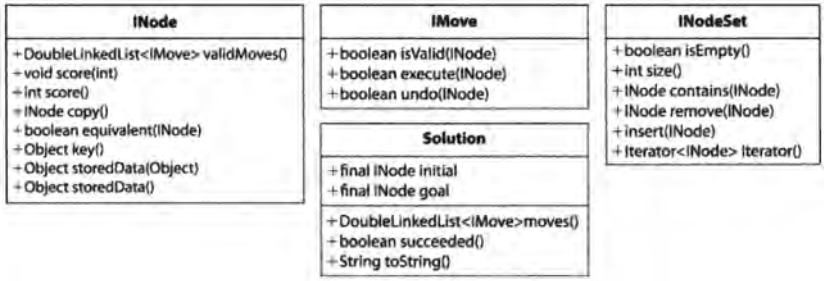


Рис. 7.14. Основные интерфейсы и классы для алгоритмов поиска в дереве

Управление состоянием доски

`copy()` возвращает идентичную копию состояния доски (за исключением необязательных сохраненных данных); `equivalent(INode)` определяет, идентичны ли два состояния доски (интеллектуальная реализация может обнаруживать осевую симметрию состояния или иные пути достижения эквивалентности). `key()` возвращает объект для поддержки проверки эквивалентности: если два состояния доски имеют один и тот же результат функции `key()`, состояния доски эквивалентны.

Управление необязательными данными состояния доски

`storedData(Object o)` связывает данный объект с состоянием доски для использования алгоритмами поиска; `storedData()` возвращает необязательные сохраненные данные, которые могут быть связаны с состоянием доски.

Интерфейс `INodeSet` абстрагирует базовую реализацию множества объектов `INode`. Некоторые алгоритмы требуют очередь объектов `INode`, некоторые — стек, а иные — сбалансированное бинарное дерево. Будучи корректно созданным (с помощью класса `StateStorageFactory`), предоставленные операции позволяют алгоритмам управлять состоянием множества `INode` независимо от используемой базовой структуры данных. Интерфейс `IMove` определяет, как ходы могут манипулировать состоянием доски; конкретные классы ходов зависят от задачи, и алгоритму поиска не нужно знать об их конкретной реализации.

С точки зрения программирования ядром алгоритма поиска пути для дерева поиска является реализация интерфейса `ISearch`, показанная в примере 7.5. Имея такое решение, можно извлекать ходы, которые генерирует это решение.

Пример 7.5. Общий интерфейс поиска путей

```
/**
 *  Для заданного начального состояния возвращает
 *  решение Solution, ведущее в конечное состояние,
 *  или null, если такой путь не может быть найден.
*/
```

```
public interface ISearch
{
    Solution search(INode initial, INode goal);
}
```

Для узла, представляющего начальное состояние доски, и желательного целевого узла реализация `ISearch` вычисляет путь, представляющий решение, или возвращает значение `null`, если решение не найдено. Для отличия от деревьев игр при обсуждении узлов дерева поиска мы используем термин *состояние доски*.

Эвристические функции длины пути

Слепой поисковый алгоритм использует фиксированную стратегию вместо вычисления состояния доски. Слепой поиск *в глубину* просто играет в игру, произвольно выбирая очередной ход из доступных вариантов для данного состояния доски и выполняя откат при достижении максимальной глубины расширения. Слепой поиск *в ширину* методично исследует все возможные решения с k ходами перед тем как испытать любое решение с $k+1$ ходами. Наверняка должен быть способ поиска, основанный на характеристиках исследуемых состояний доски, верно?

Обсуждение алгоритма **A*Search** покажет, как выполняется поиск решения головоломки "8" с использованием различных эвристических функций. Эти эвристические функции не играют в игру, а оценивают количество оставшихся ходов до целевого состояния из данного и могут использоваться для непосредственного поиска пути. Например в головоломке "8" такая функция будет оценивать для каждой плитки в состоянии доски число ходов, необходимых для ее размещения в нужном месте в целевом состоянии. Наибольшая трудность в поиске пути состоит в разработке эффективной эвристической функции.

Поиск в глубину

Поиск в глубину пытается найти путь к целевому состоянию доски, делая столько ходов вперед, сколько возможно. Поскольку некоторые деревья поиска содержат очень большое количество состояний доски, поиск в глубину оказывается практичным, только если максимальная глубина поиска фиксируется заранее. Кроме того, циклы должны избегать запоминания каждого состояния и обеспечивать его посещение только один раз.

Поиск в глубину поддерживает стек *открытых* состояний доски, которые еще не посещены, и множество *закрытых* состояний доски, которые уже были посещены. На каждой итерации поиск в глубину снимает со стека непосещенное состояние доски и расширяет его путем вычисления множества последующих состояний доски с учетом доступных разрешенных ходов. Поиск прекращается, когда достигнуто целевое состояние доски. Любой преемник состояния доски, который уже имеется во множестве закрытых состояний, отбрасывается. Оставшиеся непосещенные состояния доски помещаются в стек открытых состояний, и поиск продолжается.

Поиск в глубину

Наилучший случай: $O(b \cdot d)$; средний и наихудший случаи: $O(b^d)$

```
search (initial, goal, maxDepth)
    if initial = goal then return "Решено"
    initial.depth = 0
    open = new Stack                                          ❶
    closed = new Set
    insert (open, copy(initial))

    while open не пуст do
        n = pop (open)                                        ❷
        insert (closed, n)
        foreach допустимый ход m в n do
            nextState = состояние после хода m в n
            if closed не содержит nextState then
                nextState.depth = n.depth + 1                 ❸
                if nextState = goal then return "Решено"
                if nextState.depth < maxDepth then
                    insert (open, nextState)                  ❹

    return "Решения нет"
end
```

❶ Поиск в глубину использует стек для хранения открытых состояний, которые должны быть посещены.
❷ Последнее состояние снимается со стека.
❸ Поиск в глубину вычисляет глубину, чтобы не превысить максимальную глубину поиска.
❹ Вставка очередного состояния представляет собой операцию внесения в стек.

На рис. 7.15 показано вычисленное дерево поиска для начального состояния головоломки "8" с предельной глубиной поиска 9. Обратите внимание на то, что решение из 8 ходов (с меткой "Цель") найдено после ряда исследований до глубины 9 в других областях дерева. В целом было обработано 50 состояний дерева и 4 все еще остались неисследованными (показаны не заштрихованными). Таким образом, в этом случае мы решили головоломку, но не можем гарантировать, что нашли наилучшее (самое короткое) решение.

Входные и выходные данные алгоритма

Алгоритм начинает работу с начального состояния доски и ищет путь к целевому состоянию. Алгоритм возвращает последовательность ходов, которая представляет собой путь от начального состояния к целевому (или сообщает, что такое решение не найдено).

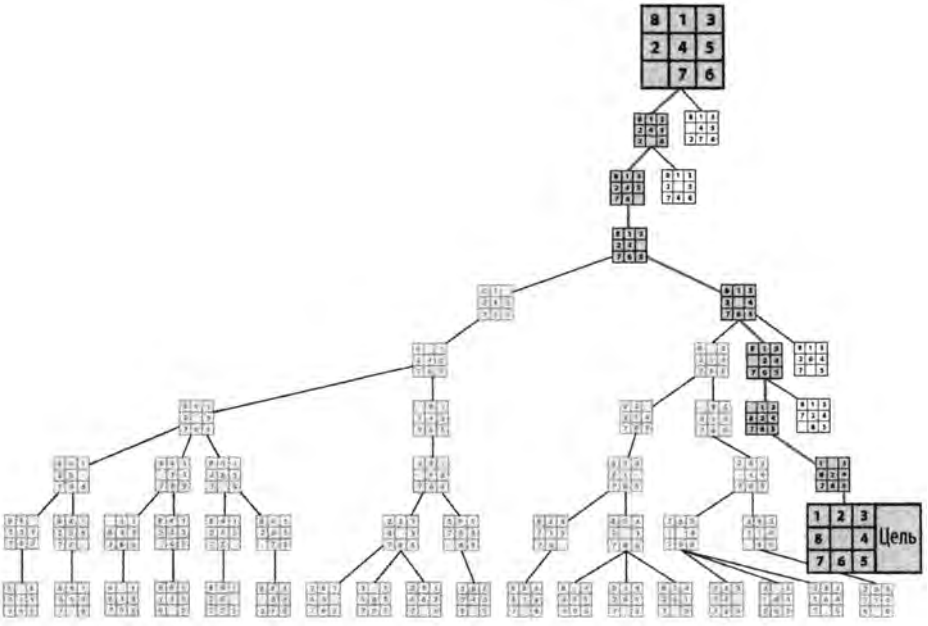


Рис. 7.15. Пример дерева поиска в глубину для головоломки "8"

Контекст применения алгоритма

Поиск в глубину представляет собой слепой поиск, который делается практичным путем ограничения глубины (по достижении фиксированной границы глубины *maxDepth* дальнейшее продвижение в глубину останавливается, что помогает управлять ресурсами памяти).

Реализация алгоритма

Поиск в глубину хранит множество *открытых* (т.е. тех, которые предстоит посетить) состояний доски в стеке и извлекает их оттуда по одному для обработки. В реализации, показанной в примере 7.6, множество *закрытых* состояний хранится в хеш-таблице для того, чтобы эффективно определять, когда не нужно возвращаться к состоянию доски, ранее уже встречавшемуся в дереве поиска; используемая хеш-функция основана на ключе, вычисляемом для каждого объекта `INode`.

Каждое состояние доски хранит ссылку под названием `DepthTransition`, которая записывает *а*) ход, создавший его, *б*) предыдущее состояние и *в*) глубину от начальной позиции. Алгоритм создает копии каждого состояния доски, поскольку ходы применяются непосредственно к состоянию и отменяются. Как только узел идентифицирован как целевой, алгоритм поиска завершается (это верно и для поиска в ширину).

Пример 7.6. Реализация поиска в глубину

```
public Solution search(INode initial, INode goal)
{
    // Если начальное состояние является целевым, возвращаем его.
    if (initial.equals(goal))
    {
        return new Solution(initial, goal);
    }

    INodeSet open =
        StateStorageFactory.create(OpenStateFactory.STACK);
    open.insert(initial.copy());
    // Уже посещенные состояния.
    INodeSet closed =
        StateStorageFactory.create(OpenStateFactory.HASH);

    while (!open.isEmpty())
    {
        INode n = open.remove();
        closed.insert(n);
        DepthTransition trans = (DepthTransition) n.storedData();
        // Все последующие ходы транслируются в открытые состояния.
        DoubleLinkedList<IMove> moves = n.validMoves();

        for (Iterator<IMove> it = moves.iterator(); it.hasNext();)
        {
            IMove move = it.next();
            // Выполнение хода над копией.
            INode successor = n.copy();
            move.execute(successor);

            // Если состояние уже посещалось, переходим к другому.
            if (closed.contains(successor) != null)
            {
                continue;
            }

            int depth = 1;

            if (trans != null)
            {
                depth = trans.depth + 1;
            }

            // Запись предыдущего хода для отслеживания решения.
            // Если решено - выход, если нет - добавление во
            // множество открытых состояний (если позволяет глубина).
```

```
            successor.storedData(new DepthTransition(move, n, depth));

            if (successor.equals(goal))
            {
                return new Solution(initial, successor);
            }

            if (depth < depthBound)
            {
                open.insert(successor);
            }
        }
    }

    return new Solution(initial, goal, false);  // Решения нет
}
```

Состояния доски сохраняются, чтобы избежать посещения одного и того же состояния дважды. Мы предполагаем, что существует эффективная функция генерации уникального ключа для состояния доски; два состояния доски считаются одинаковыми, если эта функция возвращает одно и то же значение ключа для обоих состояний.

Анализ алгоритма

Предположим, что d — это максимальная глубина, ограничивающая поиск в глубину, а b — коэффициент ветвления дерева поиска.

Производительность алгоритма определяется как общими характеристиками, так и специфичными для конкретной задачи. В общем случае замедлить работу алгоритма могут базовые операции над множествами открытых и закрытых состояний, поскольку прямые реализации могут потребовать времени $O(n)$ для обнаружения состояния доски во множестве. Основные операции включают следующее.

`open.remove()`

Удаление очередного состояния доски для оценки.

`closed.insert(INode state)`

Добавление состояния доски во множество закрытых состояний.

`closed.contains(INode state)`

Определение наличия состояния во множестве закрытых состояний.

`open.insert(INode state)`

Добавление состояния доски во множество открытых состояний для позднейшего посещения.

Поскольку поиск в глубину использует для хранения множества открытых состояний стек, операции удаления и вставки выполняются за константное время. Закрытые состояния хранятся в хеш-таблице с использованием значений ключей (предоставляемых классом состояния доски, реализующим интерфейс `INode`), так что амортизированное время поиска — константное.

Специфическими для конкретной задачи характеристиками, которые влияют на производительность, являются *а*) количество состояний-преемников для отдельного состояния доски и *б*) упорядочение допустимых ходов. Некоторые игры имеют большое количество потенциальных ходов в каждом состоянии доски, а это означает, что многие пути в глубину могут оказаться неблагоразумными. Способ упорядочения также будет влиять на поиск в целом. Если имеется какая-то эвристическая информация, убедитесь, что выполняемые ходы, которые вероятнее других ведут к решению, в упорядоченном списке допустимых ходов появляются раньше прочих.

Мы оцениваем поиск в глубину, используя набор из трех примеров (N1, N2 и N3), чтобы показать, насколько капризен такой поиск и насколько он зависит от, казалось бы, незначительных различий состояний. В каждом примере из целевого состояния выполнено по 10 перемещений плиток. Иногда поиск в глубину позволяет быстро найти решение. В общем случае размер дерева поиска растет экспоненциально с основанием, равным коэффициенту ветвления *b*. У головоломки "8" *коэффициент ветвления* находится между 2 и 4, в зависимости от местоположения пустой плитки, и в среднем равен 2,67. Мы делаем следующие два наблюдения.

Неудачный выбор глубины может помешать найти решение

Для начальной позиции N2, показанной на рис. 7.16, и глубины 25 решение не будет найдено после поиска среди 20 441 состояния доски. Как такое вообще возможно? Дело в том, что поиск в глубину не посещает одно и то же состояние доски дважды. В частности, ближе всего к нахождению решения этот поиск подходит на 3451 состоянии доски, проверяемом на 25-м уровне. Начальная позиция находится только на расстоянии трех ходов от решения, но поскольку состояние было посещено при достижении предела глубины, дальнейшее расширение было остановлено, а состояние доски добавлено во множество закрытых. Если поиск в глубину встречает этот узел снова на более высоком уровне, он не исследует его далее просто потому, что этот узел находится во множестве закрытых.

Таким образом, может показаться, что лучше устанавливать максимальную глубину более высокой, но, как показано на рис. 7.17, это обычно приводит к очень большим деревьям поиска и не гарантирует, что решение будет найдено.

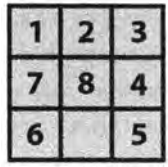


Рис. 7.16. Начальная позиция N2

Рис. 7.17. Размер деревьев поиска для поиска в глубину с увеличением предельной глубины поиска

Увеличение предельной глубины может приводить к тому, что найденное решение будет не оптимальным

Обнаруженные решения увеличиваются с увеличением предела глубины, иногда в два или три раза больше, чем необходимо.

Интересно, что для примера N1 неограниченная глубина поиска находит решение из 30 ходов после обработки только 30 состояний доски, с 23 состояниями во множестве открытых состояний, ожидающих обработки. Однако это везение, такое счастливое стечение обстоятельств не повторяется в примерах N2 и N3.

Поиск в ширину

Поиск в ширину пытается найти путь с помощью методичной оценки состояний доски, находящихся ближе всего к начальному состоянию. Поиск гарантированно находит кратчайший путь к целевому состоянию, если такой путь существует.

Кардинальным отличием от поиска в глубину является то, что поиск в ширину поддерживает очередь открытых и еще непосещенных состояний, в то время как поиск в глубину использует стек. На каждой итерации поиск в ширину удаляет из

очереди очередное непосещенное состояние и расширяет его для вычисления множества состояний-преемников с учетом допустимых ходов. Если достигается целевое состояние, поиск прекращается. Как и поиск в глубину, поиск в ширину гарантирует, что одно и то же состояние не посещается дважды. Любое состояние, имеющееся во множестве закрытых состояний, отбрасывается. Остальные непосещенные состояния добавляются в конец очереди открытых состояний, и поиск продолжается.

Используя в качестве примера головоломку "8" с начальным состоянием, показанным на рис. 7.18, мы получаем дерево поиска, показанное на рис. 7.19. Обратите внимание, что решение, состоящее из пяти ходов, обнаруживается после того, как изучены все пути с четырьмя ходами (и проверены почти все решения с пятью ходами). 20 белых листьев дерева на рисунке представляют собой открытые состояния, дожидающиеся проверки в очереди. В общей сложности были обработаны 25 состояний доски.

Рис. 7.18. Начальная позиция для поиска в ширину

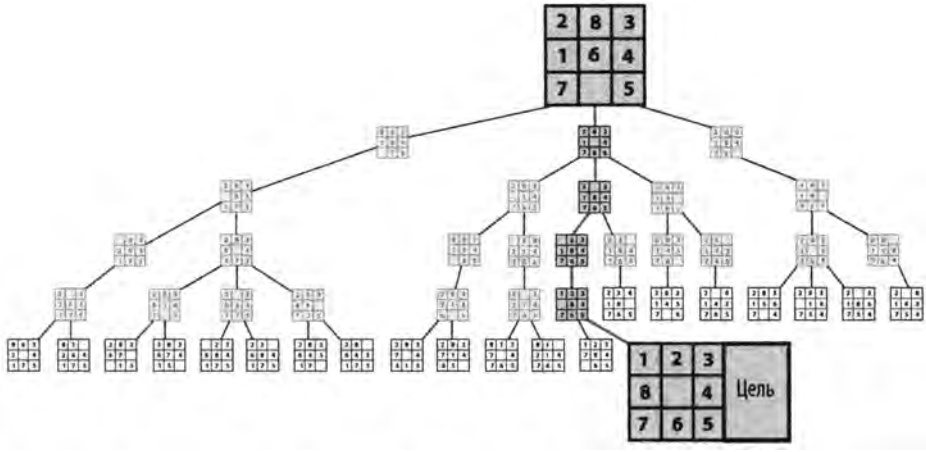


Рис. 7.19. Пример дерева поиска в ширину для головоломки "8"

Входные и выходные данные алгоритма

Алгоритм начинает работу с начального состояния доски и ищет целевое состояние. Алгоритм возвращает последовательность ходов, которая представляет собой минимальное решение, переводящее начальное состояние в целевое (или сообщает, что такое решение (с учетом имеющихся ресурсов) не найдено).

Контекст применения алгоритма

Слепой поиск является практичным, только если прогнозируемому поиску хватает памяти компьютера. Поскольку поиск в ширину сначала методично проверяет все кратчайшие пути, поиск путей с бо́льшим количеством ходов может занять довольно много времени. Этот алгоритм может не быть подходящим, если вам нужно найти только какой-нибудь путь от исходного состояния до целевого (т.е. если нет необходимости в поиске именно кратчайшего пути).

Поиск в ширину

Наилучший, средний и наихудший случаи: $O(b^d)$

```
search (initial, goal)
    if initial = goal then return "Решено"
    open = new Queue                                          ❶
    closed = new Set
    insert (open, copy(initial))

    while очередь open не пуста do
        n = head (open)                                       ❷
        insert (closed, n)
        foreach допустимый ход m в позиции n do
            nextState = Состояние, когда из n сделан ход m
            if nextState нет во множестве закрытых состояний then
                if nextState = goal then return "Решено"
                insert (open, nextState)                      ❸

    return "Решения нет"
end
```

❶ Поиск в ширину использует очередь для хранения открытых состояний, которые должны быть посещены.
❷ Удаление самого старого состояния из очереди.
❸ Вставка очередного состояния будет операцией добавления, так как open представляет собой очередь.

Реализация алгоритма

Поиск в ширину хранит множество открытых (т.е. тех, которые еще предстоит посетить) состояний доски в очереди и извлекает их по одному для обработки. Множество закрытых состояний хранится с помощью хеш-таблицы. Каждое состояние содержит ссылку `Transition`, которая записывает сгенерированный ход и ссылку на предыдущее состояние. Поиск в ширину создает копии каждого состояния доски, поскольку ходы применяются непосредственно к состояниям и не отменяются. Реализация данного алгоритма показана в примере 7.7.

Пример 7.7. Реализация поиска в ширину

```
public Solution search(INode initial, INode goal)
{
    // Выход, если начальное состояние является целевым
    if (initial.equals(goal))
    {
        return new Solution(initial, goal);
    }

    // Начинаем с начального состояния
    INodeSet open =
        StateStorageFactory.create(StateStorageFactory.QUEUE);
    open.insert(initial.copy());
    // Состояния, которые уже были посещены.
    INodeSet closed = StateStorageFactory.create(
                          StateStorageFactory.HASH);

    while (!open.isEmpty())
    {
        INode n = open.remove();
        closed.insert(n);
        // Все последующие ходы транслируются в состояния,
        // добавляемые в очередь открытых.
        DoubleLinkedList<IMove> moves = n.validMoves();

        for(Iterator<IMove> it = moves.iterator(); it.hasNext();)
        {
            IMove move = it.next();
            // Выполняем ход с копией
            INode successor = n.copy();
            move.execute(successor);

            // Если состояние уже посещалось, не обрабатываем
            if (closed.contains(successor) != null)
            {
                continue;
            }

            // Запись предыдущего хода для отслеживания решения.
            // Если решение - выход из метода, в противном случае
            // состояние добавляется в очередь открытых состояний.
            successor.storedData(new Transition(move, n));

            if (successor.equals(goal))
            {
                return new Solution(initial, successor);
            }
```

```
                open.insert(successor);
            }
        }
    }
```

Анализ алгоритма

Как и в случае поиска в глубину, производительность алгоритма определяется как общими характеристиками, так и зависящими от конкретной задачи. Здесь применим тот же анализ глубины поиска, и единственным отличием является размер множества открытых состояний доски. Поиск должен хранить во множестве открытых состояний порядка b^d состояний доски, где b — коэффициент ветвления состояний доски, а d — глубина найденного решения. Это гораздо больше, чем в случае поиска в глубину, которому в любой момент времени необходимо хранить во множестве открытых состояний только около $b \cdot d$ состояний с учетом максимальной глубины поиска d. Поиск в ширину гарантированно находит решение с наименьшим числом ходов, которые преобразуют начальное состояние доски в целевое состояние.

Поиск добавляет состояние доски во множество открытых состояний, только если оно отсутствует во множестве закрытых состояний. За счет некоторой дополнительной обработки можно сэкономить память, проверяя при добавлении состояния доски к открытым состояниям, не содержится ли оно уже в этом множестве.

A*Search

Поиск в ширину находит оптимальное решение (если таковое существует), но он может исследовать огромное количество узлов, так как не делает попытки разумно выбрать порядок ходов для исследования. Поиск в глубину, напротив, пытается быстро найти решение, продвигаясь как можно дальше при исследовании ходов; однако этот поиск может быть ограниченным, потому что в противном случае он может бесплодно исследовать непродуктивные области дерева поиска. Алгоритм **A*Search** добавляет эвристический интеллект для руководства поиском вместо слепого следования одной из этих фиксированных стратегий.

A*Search представляет собой итеративный, упорядоченный поиск, который поддерживает набор открытых состояний доски, предназначенных для исследования в попытке достичь целевого состояния. На каждой итерации A*Search использует функцию оценки $f(n)$ для выбора того состояния доски n из множества открытых состояний, для которого $f(n)$ имеет наименьшее значение. $f(n)$ имеет структуру $f(n) = g(n) + h(n)$, где:

- $g(n)$ записывает длину кратчайшей последовательности ходов из начального состояния в состояние доски n; это значение записывается по мере работы алгоритма;

- $h(n)$ оценивает длину кратчайшей последовательности ходов из состояния n в целевое состояние.

Таким образом, функция $f(n)$ оценивает длину кратчайшей последовательности ходов от исходного состояния до целевого, проходящей через n. A*Search проверяет достижение целевого состояния только тогда, когда состояние доски удаляется из множества открытых состояний (в отличие от поиска в ширину и поиска в глубину, которые выполняют эту проверку тогда, когда генерируются состояния-преемники). Это различие гарантирует, что решение представляет собой наименьшее количество ходов от начального состояния доски, если только $h(n)$ никогда не переоценивает расстояние до целевого состояния.

Низкая оценка $f(n)$ предполагает, что состояние доски n находится близко к конечному целевому состоянию. Наиболее важным компонентом $f(n)$ является эвристическая оценка, которую вычисляет функция $h(n)$, поскольку значение $g(n)$ может быть вычислено "на лету" путем записи в каждом состоянии доски его удаленности от первоначального состояния. Если $h(n)$ не в состоянии точно отделить перспективные состояния доски от бесперспективных, алгоритм A*Search будет выполняться не лучше, чем уже описанный слепой поиск. В частности, функция $h(n)$ должна быть *приемлемой*, т.е. она никогда не должна *преувеличивать* фактическую минимальную стоимость достижения целевого состояния. Если оценка оказывается слишком высокой, алгоритм A*Search может не найти оптимальное решение. Однако определить приемлемую эффективно вычисляемую функцию $h(n)$ — трудная задача. Существуют многочисленные примеры непригодных $h(n)$, которые, тем не менее, приводят к практичным, хотя и не оптимальным решениям.

Алгоритм A*Search

Наилучший случай: $(Ob \cdot d)$; средний и наихудший случаи: $O(b^d)$

```
search (initial, goal)
    initial.depth = 0
    open = new PriorityQueue                                    ❶
    closed = new Set
    insert (open, copy(initial))

    while очередь open не пуста do
        n = minimum (open)
        insert (closed, n)
        if n = goal then return "Решено"
        foreach допустимый ход m в состоянии n do
            nextState = состояние после хода m в состоянии n
            if closed содержит состояние nextState then continue
```

```
            nextState.depth = n.depth + 1
            prior = состояние в open, соответствующее nextState ❷
            if prior отсутствует
               или nextState.score < prior.score then
                if prior имеется в open                          ❸
                    remove (open, prior)                         ❹
                insert (open, nextState)                         ❺

    return "Решения нет"
end

❶ A*Search хранит открытые состояния в очереди с приоритетами
    в соответствии с вычисленной оценкой.
❷ Требуется возможность быстро найти соответствующий узел в open.
❸ Если A*Search повторно посещает в open состояние prior,
    которое теперь имеет меньшую оценку...
❹ ... то заменяем состояние prior в open альтернативой
    с лучшей оценкой.
❺ Поскольку open является очередью с приоритетами, состояние
    nextState вставляется в нее с учетом его оценки.
```

Входные и выходные данные алгоритма

Алгоритм начинает работу с начального состояния доски в дереве поиска и целевого состояния. Он предполагает наличие функции оценки $f(n)$ с приемлемой функцией $h(n)$. Алгоритм возвращает последовательность ходов, являющуюся решением, которое наиболее близко приближается к решению с минимальной стоимостью, переводящему первоначальное состояние в целевое (или объявляет, что такое решение не может быть найдено при имеющихся в наличии ресурсах).

Контекст применения алгоритма

Для примера головоломки "8" с начальным состоянием, показанным на рис. 7.20, на рис. 7.21 и 7.22 приведены вычисленные деревья поиска. Дерево на рис. 7.21 использует функцию $f(n)$ `GoodEvaluator`, предложенную в [42]. На рис. 7.22 использована функция `WeakEvaluator` из той же работы. Эти функции оценки будут описаны чуть позже. Светло-серый цвет состояния показывает множество открытых состояний в момент достижения целевого состояния.

8	1	3
	4	5
2	7	6

*Рис. 7.20. Начальное состояние доски для алгоритма A*Search*

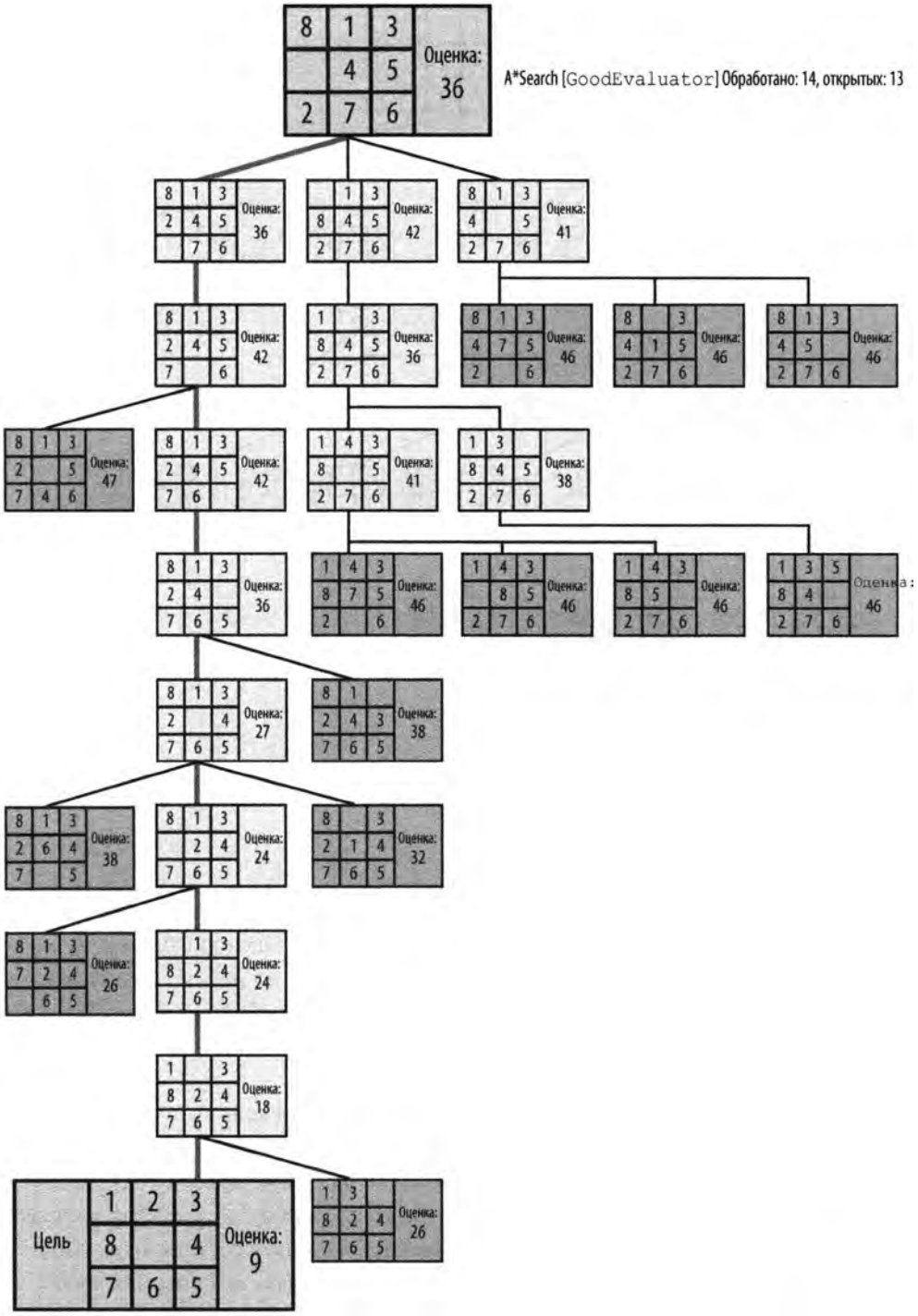


*Рис. 7.21. Пример дерева A*Search при использовании функции* `GoodEvaluator`

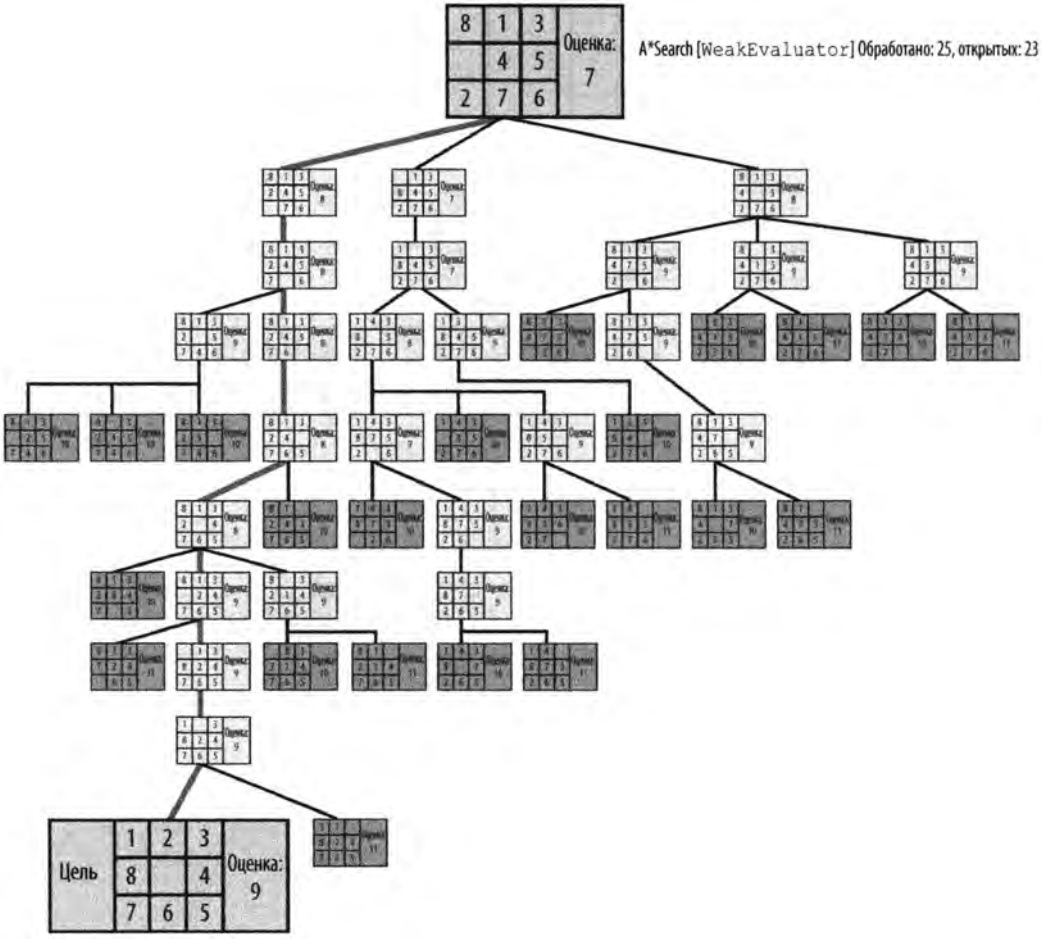


*Рис. 7.22. Пример дерева A*Search при использовании функции* `WeakEvaluator`

Обе функции — и `GoodEvaluator`, и `WeakEvaluator` — находят одно и то же решение из девяти ходов в целевой узел (с меткой "Цель"), но поиск с использованием функции `GoodEvaluator` оказывается более эффективным. Рассмотрим значения $f(n)$, связанные с узлами в обоих деревьях поиска, чтобы понять, почему дерево поиска `WeakEvaluator` исследует большее количество узлов.

Заметим, что через два хода от первоначального состояния в дереве поиска `GoodEvaluator` начинается четкий путь из узлов с постоянно уменьшающимся значением $f(n)$, который приводит к целевому узлу. В дереве же поиска `WeakEvaluator` приходится исследовать четыре хода от исходного состояния до того, как направление поиска будет сужено. `WeakEvaluator` неверно различает состояния доски; в самом деле, обратите внимание, что значение $f(n)$ целевого узла оказывается выше, чем значения $f(n)$ для начального узла и всех трех его дочерних узлов.

Реализация алгоритма

Алгоритм A*Search хранит открытые состояния доски так, что может как эффективно удалять состояния, функция оценки которых дает наименьшее значение, так и эффективно определять, имеется ли некоторое состояние доски среди открытых. В примере 7.8 приведена реализация алгоритма A*Search на языке программирования Java.

*Пример 7.8. Реализация алгоритма A*Search*

```
public Solution search(INode initial, INode goal)
{
    // Работа начинается с начального состояния
    int type = StateStorageFactory.PRIORITY_RETRIEVAL;
    INodeSet open = StateStorageFactory.create(type);
    INode copy = initial.copy();
    scoringFunction.score(copy);
    open.insert(copy);
    // Используем Hashtable для хранения уже посещенных состояний.
    INodeSet closed = StateStorageFactory.create(
                          StateStorageFactory.HASH);

    while (!open.isEmpty())
    {
        // Удаляем узел с наименьшим значением функции оценки
        // и помечаем его как закрытый.
        INode best = open.remove();

        // Выход, если достигнуто целевое состояние.
        if (best.equals(goal))
        {
            return new Solution(initial, best);
        }

        closed.insert(best);
        // Вычисление последующих ходов и обновление списков
        // открытых и закрытых состояний.
        DepthTransition trans = (DepthTransition) best.storedData();
        int depth = 1;

        if (trans != null)
        {
            depth = trans.depth + 1;
        }

        for (IMove move : best.validMoves())
        {
            // Выполняем ход и подсчитываем оценку
```

```
            // нового состояния доски.
            INode successor = best.copy();
            move.execute(successor);

            if (closed.contains(successor) != null)
            {
                continue;
            }

            // Записываем предыдущий ход для отслеживания решения
            // и вычисляем функцию оценки, чтобы увидеть
            // достигнутое улучшение.
            successor.storedData(new DepthTransition(move,best,depth));
            scoringFunction.score(successor);
            // Если еще не посещен или если имеет лучшую оценку.
            INode exist = open.contains(successor);

            if (exist == null || successor.score() < exist.score())
            {
                // Удаление старого узла в случае его наличия
                // и вставка нового
                if (exist != null)
                {
                    open.remove(exist);
                }

                open.insert(successor);
            }
        }
    }

    // Решения нет.
    return new Solution(initial, goal, false);
}
```

Как и в случае поиска в ширину и поиска в глубину, состояния доски после обработки вносятся во множество закрытых состояний. Каждое состояние доски хранит ссылку `DepthTransition`, которая записывает *а*) ход, сгенерировавший его, *б*) предыдущее состояние и *в*) глубину от начальной позиции. Это последнее значение — глубина — используется в качестве компонента $g(n)$ функции оценки. Алгоритм генерирует копии каждого состояния доски, поскольку ходы применяются непосредственно к доске, и их откат не выполняется.

Поскольку A*Search использует эвристическую информацию, которая включает вычисляемый компонент $g(n)$, имеется одна ситуация, когда A*Search может пересмотреть принятое решение на уже посещенных досках. Состояние доски, вставляемое во множество открытых состояний, может иметь оценку, более низкую, чем

идентичное состояние, уже имеющееся во множестве открытых состояний. Если так, то A*Search удаляет состояние с более высокой оценкой, имеющееся во множестве открытых состояний, поскольку оно не может быть частью решения с минимальной стоимостью. Вспомним ситуацию с поиском в глубину, когда состояние доски на предельной глубине находилось (как оказалось) только в трех ходах от целевого состояния (см. раздел "Анализ алгоритма" для поиска в глубину). Эти состояния доски были помещены во множество закрытых состояний, и, тем самым, их обработка стала невозможной. A*Search позволяет избежать этой ошибки, продолжая вычислять состояние доски во множестве открытых состояний с наименьшей оценкой.

Успех алгоритма A*Search непосредственно зависит от его эвристической функции. Компонент $h(n)$ функции оценки $f(n)$ должен быть тщательно разработан, и это в большей степени искусство, чем наука. Если $h(n)$ всегда равно нулю, A*Search вырождается в простой поиск в ширину. Кроме того, если $h(n)$ переоценивает стоимость достижения целевого состояния, A*Search может не суметь найти оптимальное решение, хотя и сможет возвратить какое-то решение в предположении, что $h(n)$ не слишком далека от истины. A*Search будет находить оптимальное решение, если эвристическая функция $h(n)$ является приемлемой.

Большая часть литературы, посвященной A*Search, описывает высокоспециализированные функции $h(n)$ для разных областей, таких как поиск маршрута на цифровой местности [60] или планирование проектов при ограниченных ресурсах [30]. Книга [45] представляет собой обширный справочник по разработке эффективных эвристик. В [35] описывается, как создавать приемлемые функции $h(n)$, а в [40] приведены последние перспективы использования эвристик в решении задач, и не только для A*Search.

Для головоломки "8" можно воспользоваться тремя приемлемыми эвристиками и одной плохо определенной функцией.

`FairEvaluator`

$P(n)$, где $P(n)$ — это сумма манхэттенских расстояний каждой плитки от ее окончательного местоположения.

`GoodEvaluator`

$P(n)+3\cdot S(n)$, где $P(n)$ определена выше, а $S(n)$ представляет собой оценку последовательности, которая по очереди проверяет нецентральные квадраты, давая 0 для каждой плитки, за которой идет ее корректный преемник, и 2 для плитки, не обладающей этим свойством; плитка в центре получает оценку 1.

`WeakEvaluator`

Подсчет количества плиток, находящихся не на своем месте.

`BadEvaluator`

Общая разность противоположных (относительно центра) плиток по сравнению с идеальным значением, равным 16.

Чтобы обосновать приемлемость первых трех эвристических функций, рассмотрим `WeakEvaluator`, которая просто возвращает число от 0 до 8. Очевидно, что это не является переоценкой количества ходов, но является плохой эвристикой для различения состояний доски. `FairEvaluator` вычисляет манхэттенское расстояние $P(n)$, которое представляет собой расстояние между двумя плитками в предположении, что вы можете двигаться только по горизонтали или вертикали; она точно суммирует расстояния от каждой плитки до ее конечного местоположения. Естественно, эта функция недооценивает фактическое количество ходов, потому что в этой головоломке можно перемещать только плитки, соседние с пустой ячейкой. Главное — *не переоценить* число ходов. `GoodEvaluator` добавляет отдельное количество $3 \cdot S(n)$, которое обнаруживает последовательности и тем сильнее увеличивает оценку, чем больше плиток находится вне последовательности. Все три эти функции являются приемлемыми.

Результаты оценки этими функциями состояния доски на рис. 7.23 показаны в табл. 7.2. Как видите, все допустимые функции находят кратчайшее решение из 13 ходов, в то время как неприемлемая эвристическая функция `BadEvaluator` находит решение из 19 ходов с гораздо бо́льшим деревом поиска.

1	4	8
7	3	
6	5	2

Рис. 7.23. Пример состояния доски для тестирования функций оценки

Таблица 7.2. Сравнение трех приемлемых и одной неприемлемой функций $h(n)$

Название	Вычисление $h(n)$	Статистика		
		Решение, ходов	Закрытых состояний	Открытых состояний
`GoodEvaluator`	13 + 3*11 = 46	13	18	15
`FairEvaluator`	13	13	28	21
`WeakEvaluator`	7	13	171	114
`BadEvaluator`	9	19	1496	767

Поиск в ширину и поиск в глубину обращаются ко множеству закрытых состояний, чтобы выяснить, содержится ли в нем то или иное состояние доски, поэтому для повышения эффективности мы использовали хеш-таблицы. Однако алгоритму A*Search, возможно, потребуется пересмотреть состояние доски, которое было посещено ранее, если вычисление функции оценки дает значение, более низкое, чем текущее состояние. Таким образом, хеш-таблица в данном случае не подходит, так как A*Search должен быть в состоянии быстро найти состояние доски в открытой очереди с приоритетами с наиболее низкой оценкой.

Обратите внимание, что поиск в ширину и поиск в глубину получают очередное состояние доски из множества открытых состояний за константное время, потому что они используют очередь и стек соответственно. Если мы храним множество открытых состояний как упорядоченный список, производительность снижается, поскольку вставка состояния доски во множество открытых состояний выполняется за время $O(n)$. Для хранения множества открытых состояний мы не можем использовать и бинарную пирамиду, потому что не знаем заранее, сколько состояний доски будет вычислено. Поэтому мы используем сбалансированное бинарное дерево, которое предлагает для извлечения состояния с наименьшей стоимостью и для вставки узлов производительность $O(log\ n)$, где n — размер множества открытых состояний.

Анализ алгоритма

Вычислительное поведение алгоритма A*Search полностью зависит от эвристической функции. В работе [50] подытожены характеристики эффективных эвристических функций. В [6] для рассмотрения представлены некоторые альтернативы, когда не удается эффективно вычислить приемлемую функцию $h(n)$. Чем более сложными становятся состояния доски, тем большую важность приобретают эвристические функции и тем более сложной становится их разработка. Они должны оставаться эффективно вычисляемыми, иначе процесс поиска резко замедлится. Однако даже грубые эвристические функции способны резко сократить пространство поиска. Например, головоломка "15", естественное развитие головоломки "8", включает 15 плиток на доске размером 4×4. Разработка функции `GoodEvaluator` на основе логики той же функции для головоломки "8" требует буквально нескольких минут. Для целевого состояния (рис. 7.24, слева) и начального состояния (рис. 7.24, справа) A*Search быстро находит решение из 15 ходов после обработки 39 состояний доски. По окончании работы во множестве открытых состояний остаются 43 непросмотренных состояния.

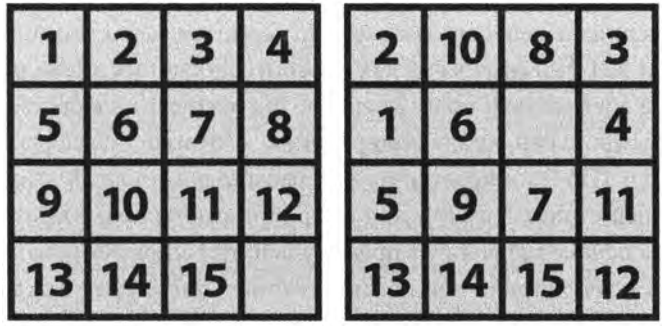


Рис. 7.24. Целевое состояние головоломки "15" (слева) и пример начального состояния (справа)

При ограничении глубины поиска 15 ходами поиск в глубину не в состоянии найти решение после изучения 22 125 состояний доски. Поиск в ширину при решении той же задачи исчерпывает 64 Мбайт памяти после наличия 172 567 состояний (85 213 во множестве закрытых состояний и 87 354 во множестве открытых состояний). Конечно, можно использовать большее количество памяти или увеличить предел глубины поиска, но это возможно не во всех случаях, поскольку каждая проблема имеет свои характеристики.

Но не обманывайтесь тем, как легко алгоритм A*Search справился с приведенным примером; при попытке решения для более сложного начального состояния (рис. 7.25) алгоритм A*Search исчерпывает всю доступную память.

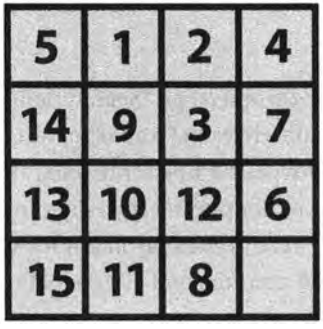


Рис. 7.25. Сложное начальное состояние головоломки "15"

Грубая функция оценки для головоломки "15" при применении ее для головоломки "24" является неэффективной, так как последняя имеет более 10^{25} возможных состояний [35].

Вариации алгоритма

Вместо выполнения поиска только вперед от начального состояния, Кейндл (Kaindl) и Кейнц (Kainz) [32] дополнили этот поиск одновременным поиском в обратном направлении от целевого состояния. Несмотря на то что их метод ранними исследователями ИИ был отвергнут как неработоспособный, Кейндл и Кейнц представили мощные аргументы в пользу пересмотра отношения к своему подходу.

Еще одна распространенная альтернатива A*Search, известная как **Iterative-DeepeningA*** (или **IDA***), опирается на ряд расширений поиска в глубину с фиксированной границей стоимости [49]. Для каждой последовательной итерации граница увеличивается на основе результатов предыдущей итерации. Алгоритм IDA* более эффективен, чем поиск в ширину или поиск в глубину, потому что каждая вычисленная стоимость основывается на фактической последовательности ходов, а не на эвристических оценках. В работе [35] описано, как для решения случайных экземпляров головоломки "15" были применены мощные эвристики в сочетании с IDA*, оценивающие более чем 400 миллионов состояний доски во время поиска.

Хотя A*Search и дает решения с минимальной стоимостью, пространство поиска может быть слишком большим для завершения алгоритма A*Search. Основные идеи, которые позволяют решить эту проблему, включают следующие.

Итеративное углубление

Эта стратегия поиска состояния использует повторяющиеся итерации ограниченного поиска в глубину; при этом на каждой итерации предельная глубина поиска увеличивается. Такой подход может устанавливать приоритеты узлов для поиска в последующих итерациях, снижая количество непродуктивного поиска и увеличивая вероятность быстрой сходимости к победным ходам. Кроме того, поскольку пространство поиска фрагментировано на дискретные интервалы, алгоритмы реального времени могут выполнять поиск в пределах разрешенного времени и возвращать наилучший полученный за это время результат. Эта методика была впервые применена к алгоритму A*Search для создания IDA* [35].

Таблицы переходов

Чтобы избежать повторяющихся вычислений, которые уже оказались безуспешными, можно хешировать состояния доски и хранить в таблице переходов длину пути (от исходного состояния), необходимую для достижения каждого состояния. Если состояние появляется в поиске позже с текущей глубиной, большей, чем обнаруженная ранее, поиск может быть прекращен. Такой подход позволяет избежать поиска во всем поддереве, который в конечном итоге окажется непродуктивным.

Иерархия

Если состояние игры может быть представлено в виде иерархии, а не с помощью "плоской" модели, можно обратиться к методам реструктуризации больших пространств поиска в кластеры, в которых можно применять алгоритм A*Search. Примером такого подхода является Иерархический поиск **Hierarchical Path-Finding A*** (HPA*) [17].

Ограничение памяти

Вместо ограничения пространства поиска путем ограничения времени вычислений можно выполнить поиск "с потерями" и в процессе поиска отбросить различные узлы, сосредоточиваясь на поиске в областях, представляющихся перспективными. Примером этого подхода может служить алгоритм **Simplified Memory Bounded A*** (SMA*) [51].

В работе [48] обобщено множество интересных расширений A*Search. Информация об использовании A*Search в системах ИИ имеется в учебниках и различных источниках в Интернете [6].

Сравнение алгоритмов поиска в дереве

Поиск в ширину гарантированно находит решение с наименьшим числом ходов из первоначального состояния, хотя в процессе работы ему может потребоваться оценивать довольно большое количество возможных последовательностей ходов. Поиск в глубину пытается при каждом поиске пройти как можно дальше и может найти решение довольно быстро, но при этом может также затратить много времени на поиск в тех частях дерева поиска, которые не дают никакой надежды на успех. A*Search, работая в паре с приемлемой эвристической функцией, требует меньшего количества времени и находит оптимальное решение, но определение такой приемлемой функции может оказаться трудной задачей.

Таким образом, имеет смысл сравнить поиск в глубину, поиск в ширину и A*Search непосредственно. Используя головоломку "8" в качестве примера игры, мы создали начальное состояние, случайным образом *n* раз перемещая плитки (от 2 до 14 раз); обратите внимание, что одни и те же плитки не перемещались в строке дважды, поскольку так получалась бы "отмена" хода. Когда *n* достигало 32, для выполнения поисков не хватало памяти. Для каждого состояния доски мы выполняли поиск в ширину, поиск в глубину до максимальной глубины *n*, поиск в глубину до максимальной глубины 2*n* и A*Search. Для каждого размера *n* мы подсчитывали следующее.

- Общее количество состояний доски во множествах открытых и закрытых состояний. Эти значение показывает эффективность алгоритма при поиске решения. Столбцы, помеченные символом #, содержат среднее количество этих значений по всем выполненным поискам. Этот анализ сосредоточен на количестве состояний как на основном факторе, определяющем эффективность поиска.
- Общее количество ходов в найденном решении. Это значение показывает эффективность путей найденного решения. Столбцы, имена которых начинаются с *s*, содержат среднее этих итоговых значений по всем выполненным поискам. Число в скобках указывает количество выполнений, при которых не удалось найти решение в рамках заданной предельной глубины поиска.

В табл. 7.3 содержатся итоговые результаты 1000 испытаний, где *n* — количество случайных ходов (от 2 до 14). В табл. 7.3 содержатся данные *а*) о среднем количестве состояний в сгенерированных деревьях поиска и *б*) среднее количество ходов в найденных решениях. BFS означает поиск в ширину, DFS — поиск в глубину и A — поиск A*Search.

Таблица 7.3. Сравнение алгоритмов поиска

n	#A*	#BFS	#DFS(n)	#DFS(2n)	sA*	sBFS	sDFS(n)	sDFS(2n)
2	4	4,5	3	6,4	2	2	2	2
3	6	13,3	7,1	27,3	3	3	3	3
4	8	25,7	12,4	68,3	4	4	4	5

Окончание табл. 7.3

n	#A*	#BFS	#DFS(n)	#DFS(2n)	sA*	sBFS	sDFS(n)	sDFS(2n)
5	10	46,4	21,1	184,9	5	5	5	5,8
6	11,5	77,6	31,8	321	6	6	6	9,4 (35)
7	13,8	137,9	56,4	767,2	6,8	6,8	6,9	9,7 (307)
8	16,4	216,8	84,7	1096,7	7,7	7,7	7,9 (36)	12,9 (221)
9	21	364,9	144	2520,5	8,7	8,6	8,8 (72)	13,1 (353)
10	24,7	571,6	210,5	3110,9	9,8	9,5	9,8 (249)	16,4 (295)
11	31,2	933,4	296,7	6983,3	10,7	10,4	10,6 (474)	17,4 (364)
12	39,7	1430	452	6196,2	11,7	11,3	11,7 (370)	20,8 (435)
13	52,7	2337,1	544,8	12464,3	13,1	12,2	12,4 (600)	21,6 (334)
14	60,8	3556,4	914,2	14755,7	14,3	13,1	13,4 (621)	25,2 (277)

Обратите внимание, что при линейном увеличении n размер дерева поиска растет в геометрической прогрессии для всех слепых подходов, но для A*Search дерево поиска остается управляемым. Говоря точнее, темпы роста для слепых поисков приближены следующими функциями:

$$\mathrm{BFS}(n) \cong 0{,}24 \cdot (n+1)^{2{,}949}$$

$$\mathrm{DFS}(n) \cong 1{,}43 \cdot (n+1)^{2{,}275}$$

$$\mathrm{DFS}(2n) \cong 3{,}18 \cdot (n+1)^{3{,}164}$$

Поиск в ширину всегда находит кратчайший путь к решению, но обратите внимание, что A*Search отстает не так уж сильно (из-за эвристики `GoodEvaluator`), несмотря на то что исследует значительно меньшее количество состояний. В отдельных испытаниях для A*Search с 30 случайными ходами темп роста дерева поиска был оценен как $O(n^{1{,}5147})$; хотя этот рост и не линейный, размер дерева оказывается значительно меньше, чем для слепых поисков. Фактический показатель степени в каждой из этих функций зависит от коэффициента ветвления решаемой задачи. На рис. 7.26 графически показаны результаты из табл. 7.3.

Наконец заметим, как эффект горизонта не позволяет поиску в глубину найти решение во множестве случаев (напомним, что это происходит, когда узел состояния доски, находящийся только в одном ходе от целевого состояния, добавляется в набор закрытых состояний). В приведенном примере из 1000 испытаний поиск в глубину не мог найти решения более чем в 60% случаев при использовании максимальной глубины поиска 13.

Хотя все три поиска имеют потенциал для исследования экспоненциального количества состояний, A*Search исследует наименьшее их число при условии наличия приемлемой функции оценки $h(n)$.

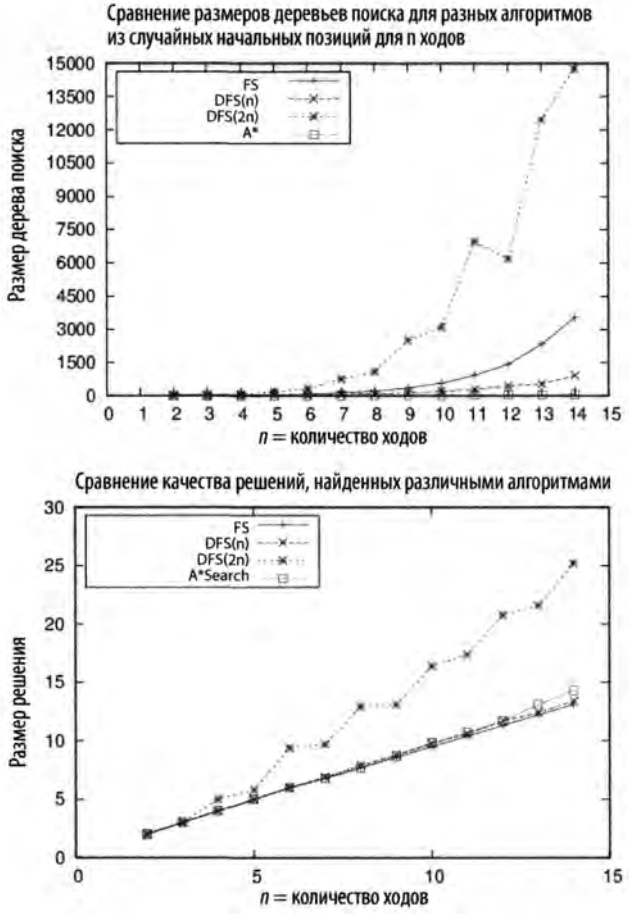


Рис. 7.26. Сравнение размеров деревьев поиска для случайных позиций

Имеются и другие известные способы решения головоломок с n^2-1 перемещаемыми плитками, кроме поиска пути. Один оригинальный подход, предложенный в [44], представляет собой применение стратегии "разделяй и властвуй". Для данной головоломки размером $n \times n$, где $n > 3$, сначала завершите крайний слева столбец и верхнюю строку, а затем рекурсивно решайте получившуюся головоломку $(n-1)^2-1$. Когда вы дойдете до задачи 3×3, просто примените метод "в лоб". Такой подход гарантированно находит решение не более чем из $5 \cdot n^3$ ходов.

8

Алгоритмы транспортных сетей

Многие задачи могут быть представлены в виде сети из вершин и ребер с пропускной способностью (capacity), связанной с каждым ребром. Алгоритмы в этой главе берут начало в необходимости решения определенных классов таких задач. В [2] содержится подробное обсуждение многочисленных приложений алгоритмов транспортных сетей.

Назначения

Для заданного множества заданий, выполняемых множеством сотрудников, причем стоимости выполнения конкретным сотрудником конкретной задачи различны, требуется таким образом распределить задачи между сотрудниками, чтобы свести к минимуму общие расходы.

Максимальное паросочетание

Для заданного множества претендентов, интервьюированных для определенного набора вакансий, следует найти паросочетание, которое максимизирует количество кандидатов, отобранных в соответствии с их квалификацией для разных рабочих мест.

Максимальный поток

Для данной транспортной сети с потенциальными пропускными способностями пересылки товаров между двумя узлами следует вычислить максимальный поток, поддерживаемый сетью.

Перевозка

Определение наиболее экономичного способа отгрузки товаров из набора складов во множество розничных магазинов.

Перегрузка

Определение наиболее экономичного способа отгрузки товаров из набора складов во множество розничных магазинов с потенциальным использованием множества складов для промежуточного хранения.

На рис. 8.1 показано, как каждая из этих задач может быть представлена в виде транспортной сети из одного или нескольких узлов-источников для одного или

нескольких узлов-стоков. Наиболее общая постановка задачи показана в нижней части рисунка, а каждая из других задач является уточнением задачи, находящейся под ней. Например, задача перевозки представляет собой частный случай задачи перегрузки, поскольку в этом случае сеть не содержит промежуточных перегрузочных узлов. Таким образом, программа, которая решает задачу перегрузки, может применяться и для решения задачи перевозки.

В этой главе представлен алгоритм Форда–Фалкерсона, который решает задачу максимального потока. Алгоритм Форда–Фалкерсона может также применяться к задаче максимального паросочетания, как ясно из рис. 8.1. После дальнейших размышлений подход, примененный в алгоритме Форда–Фалкерсона, может быть обобщен для решения более мощной задачи минимальной стоимости потока, что позволяет нам решать с помощью этого алгоритма задачи перегрузки, перевозки и назначения.

В принципе, методы линейного программирования можно применить для решения всех задач, показанных на рис. 8.1, но тогда придется преобразовывать все эти задачи в надлежащий вид задачи линейного программирования, решение которой затем придется вновь преобразовывать в решение исходной задачи (как это сделать, мы покажем в конце главы). Линейное программирование представляет собой метод вычисления оптимального результата (например, максимальной прибыли или низкой стоимости) в математической модели, состоящей из линейных соотношений. Однако на практике специализированные алгоритмы, описанные в этой главе, для указанных на рис. 8.1 задач превосходят линейное программирование на несколько порядков.

Транспортная сеть

Мы моделируем транспортную сеть как ориентированный граф $G=(V,E)$, где V — множество вершин, а E — множество ребер, соединяющих эти вершины. Граф является связным (хотя и не каждое его ребро является необходимым). Выделенная вершина-источник $s \in V$ производит единицы товара, которые затем проходят по ребрам графа до вершины-стока $t \in V$ (известной также как целевая или конечная). Транспортная сеть предполагает бесконечную поставку единиц товара из источника и то, что вершина-сток может потребить все единицы товара, которые она получает.

Каждое ребро (u,v) имеет поток $f(u,v)$, который определяет количество единиц товара, перемещающихся из u в v. Ребро также имеет фиксированную пропускную способность $c(u,v)$, которая ограничивает максимальное количество единиц товара, которые могут передаваться по этому ребру. На рис. 8.2 все вершины между истоком и стоком пронумерованы и каждое ребро имеет метку вида f/c, показывающую поток по этому ребру и максимально допустимую пропускную способность. Ребро между s и v_1, например, помечено как 5/10; это означает, что поток через это ребро составляет 5 единиц, но ребро может выдержать поток до 10 единиц. Если через какое-то ребро нет потока (как в случае с ребром между v_5 и v_2), поток f равен нулю, и указана только пропускная способность ребра.

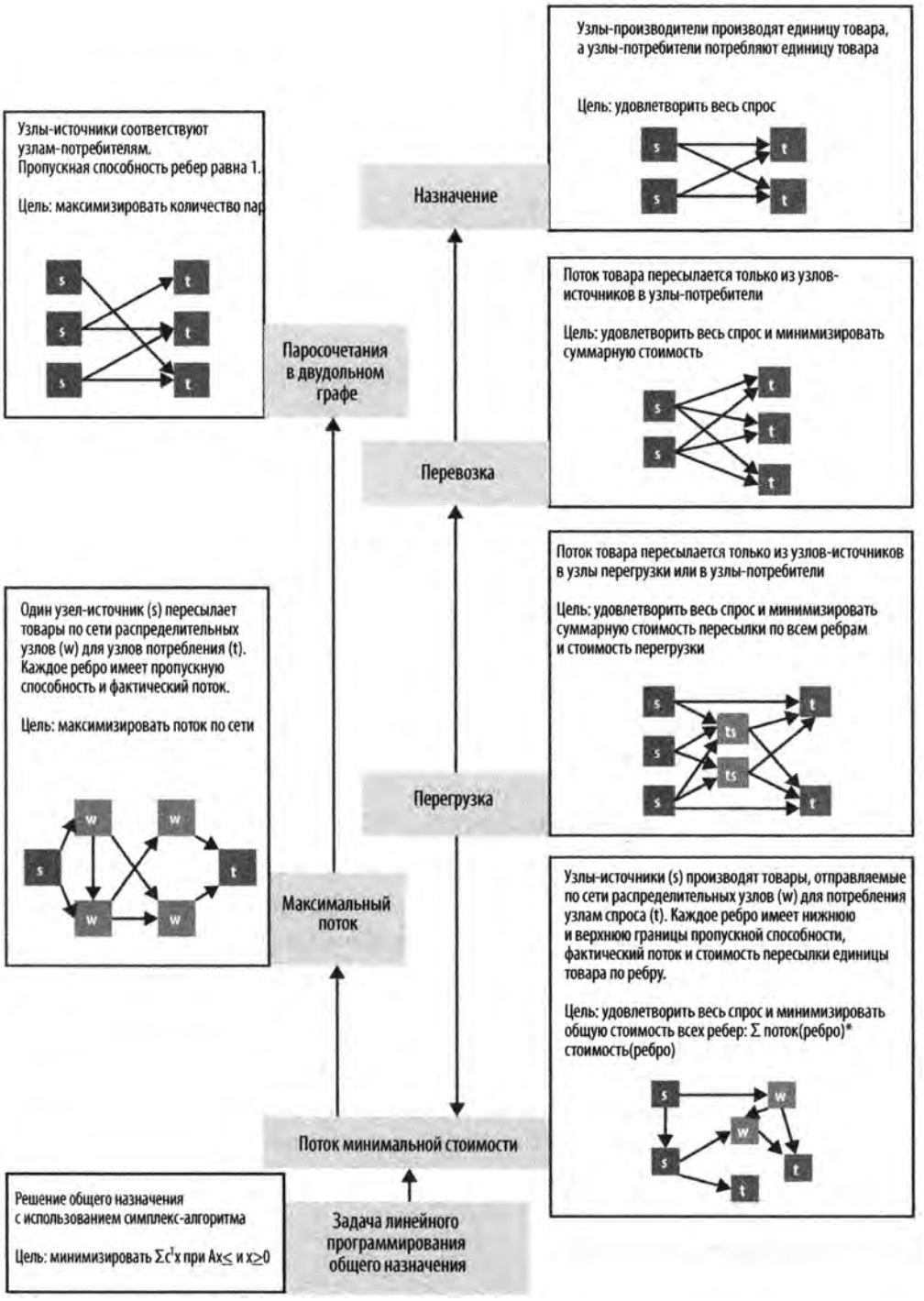


Рис. 8.1. Взаимоотношения между задачами транспортных сетей

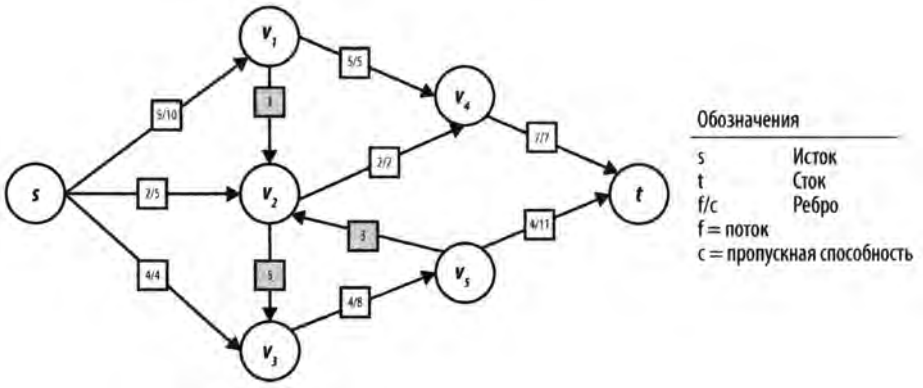


Рис. 8.2. Пример графа транспортной сети

Для любого возможного потока f через сеть должны быть удовлетворены следующие требования.

Ограничения пропускной способности

Поток $f(u,v)$ через ребро не может быть отрицательным и не может превышать пропускную способность ребра $c(u,v)$. Другими словами, $0 \le f(u,v) \le c(u,v)$. Если сеть не содержит ребро от u до v, мы определяем $c(u,v)$ как нулевую.

Сохранение потока

За исключением вершины-истока s и вершины-стока t, каждая вершина $u \in V$ должна удовлетворять свойству сохранения потока — сумма всех $f(v,u)$ для всех ребер $(v,u) \in E$ (потоки, входящие в u) должна равняться сумме $f(u,w)$ для всех ребер $(u,w) \in E$ (потоки, выходящие из u). Это свойство гарантирует, что потоки не создаются и не потребляются нигде в сети, за исключением вершин s и t.

Антисимметрия

Количество $f(v,u)$ представляет собой поток, обратный потоку от вершины u к вершине v. Это означает, что $f(u,v)$ должно быть равно $-f(v,u)$.

В последующих алгоритмах, говоря о пути в сети, мы говорим о нециклическом пути, состоящем из уникальных вершин $<v_1, v_2, \ldots v_n>$ с участием $n-1$ последовательных ребер (v_i, v_j) в E. В транспортной сети, изображенной на рис. 8.2, одним из возможных сетевых путей является путь $<v_3, v_4, v_5, v_2>$. В сетевом пути направления ребер могут быть проигнорированы (как мы увидим в ближайшее время, это необходимо для правильного построения *увеличивающего пути*). На рис. 8.2 возможным сетевым путем является путь $<s, v_1, v_4, v_2, v_5, t>$.

Максимальный поток

Для данной транспортной сети можно вычислить максимальный поток между вершинами s и t с учетом всех возможных ограничений $c(u,v) \ge 0$ для всех

ориентированных ребер $e=(u,v)$ из E. Иначе говоря, можно вычислить наибольшее количество товара, которое может выходить из истока s, проходить по транспортной сети и входить в сток t, с учетом конкретных ограничений в отдельных ребрах. Начиная с наименьшего возможного потока — потока 0 по каждому ребру — алгоритм Форда–Фалкерсона последовательно находит *увеличивающие пути* через сеть от s к t, к которым можно добавить больший поток. Алгоритм завершается, когда оказывается невозможно найти увеличивающий путь. Теорема о максимальном потоке и минимальном разрезе [23] гарантирует, что в случае неотрицательных целочисленных потоков и пропускных способностей алгоритм Форда–Фалкерсона всегда завершается и определяет максимальный поток в сети.

Транспортная сеть определяется графом $G=(V,E)$ с выделенной вершиной-истоком s и вершиной-стоком t. Каждое ориентированное ребро $e=(u,v)$ в E имеет определенную целочисленную пропускную способность $c(u,v)$ и фактический поток $f(u,v)$. *Путь* может быть построен как последовательность из n вершин из V, которые мы назовем $p_0, p_1, \ldots, p_{n-1}$, где p_0 — вершина-*исток* транспортной сети, а p_{n-1} — вершина-*сток*. Путь строится из *прямых ребер*, где ребро из последовательных вершин $(p_i,p_{i+1}) \in E$, и *обратных ребер*, где множеству E принадлежит ребро $(p_{i+1},p_i) \in E$, и путь проходит по ребру в обратном направлении.

Входные и выходные данные алгоритма

Транспортная сеть определяется графом $G=(V,E)$ с выделенной вершиной-истоком s и вершиной-стоком t. Каждое ориентированное ребро $e=(u,v)$ в E имеет определенную целочисленную пропускную способность $c(u,v)$ и фактический поток $f(u,v)$.

Для каждого ребра (u,v) из E алгоритм Форда–Фалкерсона вычисляет целочисленный поток $f(u,v)$, представляющий количество единиц, протекающих по ребру (u,v). В качестве побочного эффекта при завершении алгоритм Форда–Фалкерсона вычисляет *минимальное сечение* (или минимальный разрез) сети — иными словами, множество ребер, образующих узкое место, не позволяющее пропустить по сети большее количество единиц товаров из s в t.

Реализация алгоритма

Реализация алгоритма Форда–Фалкерсона, которая будет описана далее, использует для хранения ребер связанные списки. Каждая вершина u поддерживает два отдельных списка: прямых ребер, исходящих из u, и обратных ребер, входящих в u; таким образом, каждое ребро имеется в двух списках. Репозиторий кода этой книги содержит реализацию с использованием для хранения ребер двумерной матрицы, более подходящую структуру данных для плотного графа транспортной сети.

Алгоритм Форда–Фалкерсона

Наилучший, средний и наихудший случаи: $O(E \cdot max\_flow)$

```
compute (G)
    while существует увеличивающий путь в G do      ❶
        processPath (path)
end

processPath (path)
    v = сток
    delta = ∞
    while v ≠ исток do                              ❷
        u = вершина, предшествующая в пути v
        if edge(u,v) – прямое ребро then
            t = (u,v).capacity - (u,v).flow
        else
            t = (v,u).flow
        delta = min (t, delta)
        v = u

    v = сток
    while v ≠ исток do                              ❸
        u = вершина, предшествующая в пути v
        if edge(u,v) – прямое ребро then            ❹
            (u,v).flow += delta
        else
            (v,u).flow -= delta
        v = u
end
```

❶ Цикл может повторяться вплоть до количества итераций, равного максимальному потоку, что дает производительность O(E·max_flow).
❷ Работа в обратном направлении от истока для поиска ребра с минимальным потенциалом увеличения.
❸ Соответствующее обновление увеличивающего пути.
❹ Прямые ребра увеличивают поток; обратные – уменьшают.

Алгоритм Форда–Фалкерсона использует следующие структуры (рис. 8.3).

`FlowNetwork`

Представляет задачу транспортной сети. Этот абстрактный класс имеет два подкласса: один — для списков смежности, другой — для матрицы смежности. Метод `getEdgeStructure()` возвращает хранилище, используемое для хранения информации о ребрах.

`VertexStructure`

Поддерживает два связанных списка ребер (прямых и обратных), исходящих и входящих в вершину.

`EdgeInfo`

Записывает информацию о ребрах в транспортной сети.

`VertexInfo`

Записывает увеличивающий путь, найденный методом поиска. Записываются предыдущая вершина в увеличивающем пути, а также сведения о том, прямое это ребро или обратное.

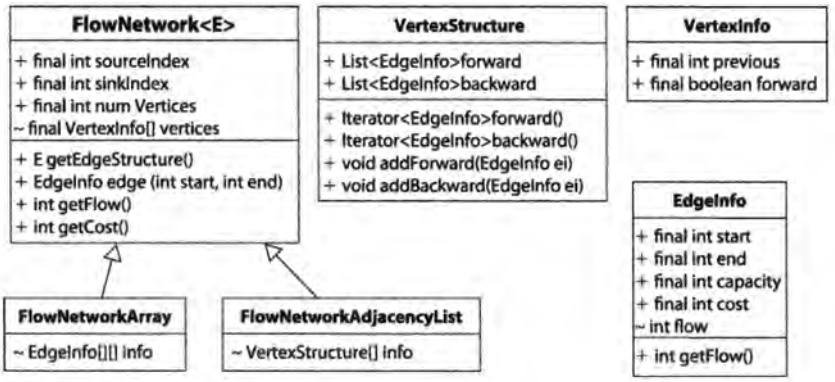


Рис. 8.3. Моделирование информации для алгоритма Форда–Фалкерсона

Алгоритм Форда–Фалкерсона реализован в примере 8.1 и показан на рис. 8.4. Настраиваемый объект `Search` вычисляет увеличивающий путь в сети, по которому можно добавить дополнительный поток без нарушения требований к транспортной сети. Алгоритм Форда–Фалкерсона постоянно продвигается вперед, поскольку субоптимальные решения, сделанные в предыдущих итерациях алгоритма, могут быть исправлены без отмены выполненных действий.

Пример 8.1. Реализация алгоритма Форда–Фалкерсона на языке программирования Java

```
public class FordFulkerson
{
    FlowNetwork network; /** Представляет задачу FlowNetwork. */
    Search searchMethod; /** Используемый метод поиска.       */

    // Построение экземпляра для вычисления максимального потока по
    // данной сети с использованием указанного метода поиска для
    // нахождения увеличивающего пути.
    public FordFulkerson(FlowNetwork network, Search method)
    {
```

```
        this.network = network;
        this.searchMethod = method;
    }

    // Вычисление максимального потока для транспортной сети.
    // Результат вычислений хранится в объекте транспортной сети.
    public boolean compute()
    {
        boolean augmented = false;

        while (searchMethod.findAugmentingPath(network.vertices))
        {
            processPath(network.vertices);
            augmented = true;
        }

        return augmented;
    }

    // Находит ребро увеличивающего пути с наименьшим потенциалом
    // увеличения и дополняет потоки на пути от истока к стоку на
    // эту величину..
    protected void processPath(VertexInfo []vertices)
    {
        int v = network.sinkIndex;
        int delta = Integer.MAX_VALUE; // Цель - наименьшее значение

        while (v != network.sourceIndex)
        {
            int u = vertices[v].previous;
            int flow;

            if (vertices[v].forward)
            {
                // Прямые ребра могут быть обновлены до
                // остающейся пропускной способности ребра.
                flow = network.edge(u, v).capacity -
                       network.edge(u, v).flow;
            }
            else
            {
                // Обратные ребра могут быть уменьшены только в
                // пределах их существующих потоков.
                flow = network.edge(v, u).flow;
            }

            if (flow < delta)
            {
```

```
                delta = flow;     // Наименьший поток-кандидат.
            }

            v = u;  // Следуем по обратному пути к истоку.
        }

        // Обновление пути (прямые ребра увеличиваются,
        // обратные уменьшаются) на величину delta.
        v = network.sinkIndex;

        while (v != network.sourceIndex)
        {
            int u = vertices[v].previous;

            if (vertices[v].forward)
            {
                network.edge(u, v).flow += delta;
            }
            else
            {
                network.edge(v, u).flow -= delta;
            }

            v = u; // Следуем по обратному пути к истоку.
        }

        // Сброс для следующей итерации
        Arrays.fill(network.vertices, null);
    }
}
```

Для поиска увеличивающего пути может использоваться любой метод поиска, который расширяет абстрактный класс `Search` (рис. 8.5). Исходное описание Форда и Фалкерсона использует поиск в глубину, а Эдмондса и Карпа — поиск в ширину (см. главу 6, "Алгоритмы на графах").

Пример транспортной сети на рис. 8.4 демонстрирует применение поиска в глубину для того, чтобы найти увеличивающий путь; его реализация показана в примере 8.2. Структура `path` во время поиска содержит стек вершин. Потенциальный увеличивающий путь расширяется путем снятия вершины u со стека и выявления соседней непосещенной вершины v, которая удовлетворяет одному из двух ограничений: 1) ребро (u,v) является прямым ребром с неполностью загруженной пропускной способностью; 2) ребро (v,u) — обратное ребро с потоком, который может быть уменьшен. Если такая вершина найдена, то v добавляется в конец `path` и внутренний цикл `while` продолжается. В конечном итоге посещается вершина стока t или `path` становится пустым (в этом случае увеличивающего пути не существует).

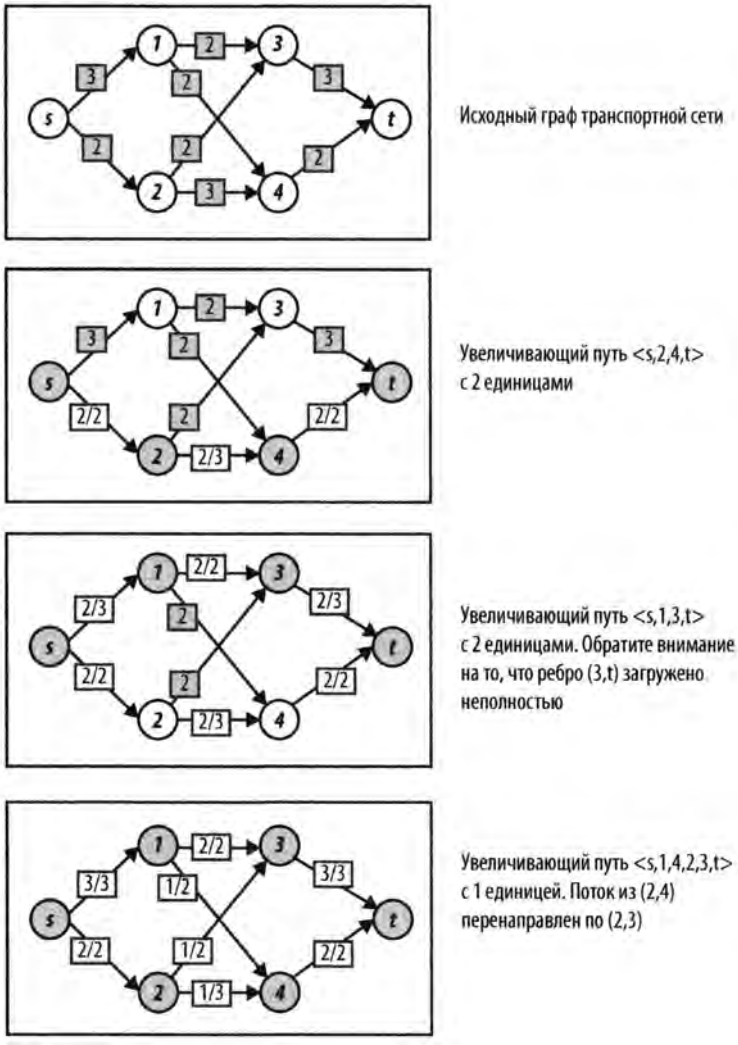


Рис. 8.4. Пример работы алгоритма Форда–Фалкерсона

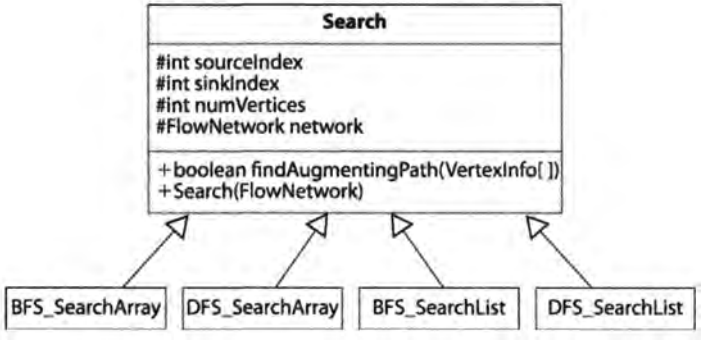


Рис. 8.5. Класс `Search`

Пример 8.2. Использование поиска в глубину для нахождения увеличивающего пути

```
public boolean findAugmentingPath(VertexInfo[] vertices)
{
    // Потенциальный увеличивающий путь начинается в истоке.
    vertices[sourceIndex] = new VertexInfo(-1);
    Stack<Integer> path = new Stack<Integer>();
    path.push(sourceIndex);
    // Обработка прямых ребер из u; затем проверяются обратные ребра.
    VertexStructure struct[] = network.getEdgeStructure();

    while (!path.isEmpty())
    {
        int u = path.pop();

        // Сначала обрабатываются прямые ребра...
        Iterator<EdgeInfo> it = struct[u].forward();

        while (it.hasNext())
        {
            EdgeInfo ei = it.next();
            int v = ei.end;

            // Еще не посещена и имеет неиспользованную пропускную
            // способность? Планируем увеличение.
            if (vertices[v] == null && ei.capacity > ei.flow)
            {
                vertices[v] = new VertexInfo(u, FORWARD);

                if (v == sinkIndex)
                {
                    return true;    // Найден!
                }

                path.push(v);
            }
        }

        // Испытываем обратные ребра
        it = struct[u].backward();

        while (it.hasNext())
        {
            // Пытаемся найти входящее в u ребро, поток
            // которого может быть уменьшен.
            EdgeInfo rei = it.next();
            int v = rei.start;
```

```
            // Теперь рассматриваем обратные еще непосещенные
            // ребра (не может быть стоком!)
            if (vertices[v] == null && rei.flow > 0)
            {
                vertices[v] = new VertexInfo(u, BACKWARD);
                path.push(v);
            }
        }
    }

    return false; // Ничего не найдено
}
```

По мере расширения пути массив `vertices` хранит информацию `VertexInfo` о прямых и обратных ребрах, что позволяет обходить увеличивающий путь в методе `processPath` из примера 8.1.

Реализация с использованием поиска в ширину, известная как алгоритм Эдмондса–Карпа, показана в примере 8.3. Здесь структура `path` в процессе поиска содержит очередь вершин. Потенциальный увеличивающий путь расширяется путем удаления вершины *u* из начала очереди и расширения очереди путем добавления смежных непосещенных вершин, через которые может проходить увеличивающий путь. И вновь либо посещается вершина-сток *t*, либо `path` становится пустым путем (если увеличивающий путь не существует). Если рассмотреть тот же пример транспортной сети на рис. 8.4, то применение поиска в ширину дает четыре увеличивающих пути — <*s*,1,3,*t*>, <*s*,1,4,*t*>, <*s*,2,3,*t*> и <*s*,2,4,*t*>. Получающийся в результате максимальный поток совпадает с полученным ранее.

Пример 8.3. Применение поиска в ширину для поиска увеличивающего пути

```
public boolean findAugmentingPath(VertexInfo []vertices)
{
    // Потенциальный увеличивающий путь начинается в истоке
    // с максимальным потоком.
    vertices[sourceIndex] = new VertexInfo(-1);
    DoubleLinkedList<Integer> path = new DoubleLinkedList<Integer>();
    path.insert(sourceIndex);
    // Обработка прямых ребер, исходящих из u; затем исследуются
    // обратные ребра, входящие в u.
    VertexStructure struct[] = network.getEdgeStructure();

    while (!path.isEmpty())
    {
        int u = path.removeFirst();
        Iterator<EdgeInfo> it = struct[u].forward(); // Ребра из u
```

```
        while (it.hasNext())
        {
            EdgeInfo ei = it.next();
            int v = ei.end;

            // Ребро еще не посещено и имеет неиспользованную
            // пропускную способность? Планируется увеличение.
            if (vertices[v] == null && ei.capacity > ei.flow)
            {
                vertices[v] = new VertexInfo(u, FORWARD);

                if (v == sinkIndex)
                {
                    return true;    // Путь завершен.
                }

                path.insert(v);     // В противном случае добавление
            }                       //  в очередь
        }

        it = struct[u].backward(); // Ребра, входящие в u

        while (it.hasNext())
        {
            // Пытаемся найти входящее в u ребро,
            // поток которого можно уменьшить.
            EdgeInfo rei = it.next();
            int v = rei.start;

            // Не посещено (но не сток!) и поток может быть уменьшен?
            if (vertices[v] == null && rei.flow > 0)
            {
                vertices[v] = new VertexInfo(u, BACKWARD);
                path.insert(v);  // Добавление в очередь
            }
        }
    }

    return false; // Увеличивающий путь не найден.
}
```

Когда алгоритм Форда–Фалкерсона завершается, вершины в V можно разделить на два непересекающихся множества, S и T (где $T = V - S$). Заметим, что $s \in S$, тогда как $t \in T$. Множество S вычисляется как набор вершин из V, которые были посещены при последней неудачной попытке найти увеличивающий путь. Важность этих множеств заключается в том, что прямые ребра между S и T являются *минимальным*

сечением, или узким местом транспортной сети. Таким образом, пропускная способность сети между S и T минимальна и вычисленный поток между S и T уже полностью ее исчерпывает.

Анализ алгоритма

Алгоритм Форда–Фалкерсона в конечном итоге завершается, поскольку единицы потока представляют собой неотрицательные целые числа [23]. Производительность алгоритма Форда–Фалкерсона при использовании поиска в глубину равна $O(E \cdot max\_flow)$ и основана на конечности значения максимального потока *max_flow*. Короче говоря, вполне возможно, что на каждой итерации увеличивающий путь добавляет только одну единицу потока, и, таким образом, транспортной сети с очень большими пропускными способностями может потребоваться большое количество итераций. Удивительно, что время работы основано не на размере задачи (т.е. количестве вершин или ребер), а на пропускной способности самих ребер.

При использовании поиска в ширину (алгоритм Эдмондса–Карпа) производительность алгоритма равна $O(V \cdot E^2)$. Поиск в ширину находит кратчайший увеличивающий путь за время $O(V+E)$, которое, по сути, равно $O(E)$, потому что в связном графе транспортной сети количество вершин существенно меньше, чем количество ребер. В [20] доказывается, что количество увеличений потоков составляет порядка $O(V \cdot E)$, что и приводит к окончательному результату: производительность алгоритма Эдмондса–Карпа равна $O(V \cdot E^2)$. Производительность алгоритма Эдмондса–Карпа часто превосходит производительность алгоритма Форда–Фалкерсона, так как основывается на поиске в ширину и исследует все возможные пути упорядоченными по длине, а не теряет много усилий на поиски пути к стоку в глубину.

Оптимизация

Типичные реализации задач с транспортными сетями используют для хранения информации массивы. Мы решили вместо этого представлять каждый алгоритм с использованием списков, поскольку такой код более удобочитаем, и читателям легче понять, как работает алгоритм. Стоит, однако, учесть, что путем оптимизации кода можно достичь существенного ускорения производительности; так, в главе 2, "Математика алгоритмов", мы продемонстрировали улучшение производительности на 50% за счет оптимизации умножения n-разрядных чисел. Очевидно, что может быть написан более быстрый код, но его будет труднее понимать, или сопровождать в случае изменения задачи. С учетом сказанного представим в примере 8.4 оптимизированную реализацию алгоритма Форда–Фалкерсона на языке программирования Java.

Пример 8.4. Оптимизированная реализация алгоритма Форда-Фалкерсона

```
public class Optimized extends FlowNetwork
{
    int[][] capacity;      // Все пропускные способности.
    int[][] flow;          // Все потоки.
    int[] previous;        // Информация о предшественнике в пути.
    int[] visited;         // Посещенные в процессе поиска
                           // увеличивающего пути ребра.
    final int QUEUE_SIZE; // Размер очереди никогда не превышает n.
    final int queue[];     // Реализация использует циклическую очередь.

    // Загрузка информации
    public Optimized(int n, int s, int t, Iterator<EdgeInfo> edges)
    {
        super(n, s, t);

        queue = new int[n];
        QUEUE_SIZE = n;
        capacity = new int[n][n];
        flow = new int[n][n];
        previous = new int[n];
        visited = new int [n];

        // Изначально поток нулевой. Получение
        // информации из входных данных.
        while (edges.hasNext())
        {
            EdgeInfo ei = edges.next();
            capacity[ei.start][ei.end] = ei.capacity;
        }
    }

    // Вычисление и возврат максимального потока.
    public int compute(int source, int sink)
    {
        int maxFlow = 0;

        while (search(source, sink))
        {
            maxFlow += processPath(source, sink);
        }

        return maxFlow;
    }

    // Увеличение потока в сети вдоль найденного
    // пути от истока до стока.
    protected int processPath(int source, int sink)
```

```
{
    // Определение величины, на которую увеличивается поток.
    // Равна минимуму по вычисленному пути от стока к истоку.
    int increment = Integer.MAX_VALUE;
    int v = sink;

    while (previous[v] != -1)
    {
        int unit = capacity[previous[v]][v] - flow[previous[v]][v];

        if (unit < increment)
        {
            increment = unit;
        }

        v = previous[v];
    }

    // Минимальное увеличение вдоль потока
    v = sink;

    while (previous[v] != -1)
    {
        flow[previous[v]][v] += increment; // Прямые ребра.
        flow[v][previous[v]] -= increment; // Обратные ребра.
        v = previous[v];
    }

    return increment;
}

// Обнаружение увеличивающего пути от истока
// к стоку в транспортной сети.
public boolean search(int source, int sink)
{
    // Сброс состояния посещения.
    // 0=сброс, 1=активен в очереди, 2=посещен
    for (int i = 0 ; i < numVertices; i++)
    {
        visited[i] = 0;
    }

    // Создание циклической очереди для обработки
    // элементов поиска.
    queue[0] = source;
    int head = 0, tail = 1;
    previous[source] = -1; // Завершение в этом месте.
    visited[source] = 1;   // Активен в очереди.
```

```
        while (head != tail)
        {
            int u = queue[head];
            head = (head + 1) % QUEUE_SIZE;
            visited[u] = 2;

            // Добавление в очередь непосещенных соседей u
            // с достаточной пропускной способностью.
            for (int v = 0; v < numVertices; v++)
            {
                if (visited[v] == 0 && capacity[u][v] > flow[u][v])
                {
                    queue[tail] = v;
                    tail = (tail + 1) % QUEUE_SIZE;
                    visited[v] = 1; // Активен в очереди.
                    previous[v] = u;
                }
            }
        }

        return visited[sink] != 0; // Достигнут ли сток?
    }
}
```

Связанные алгоритмы

Алгоритм "поднять в начало", представленный в [27], повышает производительность до $O(V \cdot E \cdot log(V^2/E))$; помимо этого, данный алгоритм может быть распараллелен для повышения производительности. Вариант задачи транспортной сети, известный как задача мультитоварного потока, обобщает рассмотренную здесь задачу поиска максимального потока. Вкратце, вместо одного истока и стока имеется обобщенная транспортная сеть с несколькими истоками s_i и стоками t_i, по которой передаются разные товары. Пропускная способность ребер фиксирована, но требования к каждому истоку и стоку могут варьироваться. Практические применения алгоритмов для решения этой задачи включают маршрутизацию в беспроводных сетях [25]. Работа [37] представляет собой широко цитируемый справочник по задачам такого рода.

Есть еще несколько незначительных вариаций задачи вычисления максимального потока.

Пропускные способности вершин

Что если транспортная сеть указывает максимальную пропускную способность $k(v)$ для потока, протекающего через вершину v в графе? Эту задачу можно решить путем построения модифицированного графа транспортной сети G_m следующим образом. Для каждой вершины v в графе G создайте две вершины — v_a и v_b. Создайте также ребро (v_a, v_b) с пропускной способностью

$k(v)$. Для каждого входящего ребра (u,v) в G с пропускной способностью $c(u,v)$ создайте новое ребро (u,v_a) с пропускной способностью $c(u,v)$. Для каждого исходящего ребра (v,w) в G создайте в G_m ребро (v_b,w) с пропускной способностью $k(v)$ потенциала. Решение задачи транспортной сети G_m определяет решение G.

Неориентированные ребра

Что делать, если транспортная сеть G имеет неориентированные ребра? Постройте модифицированный граф G_m с тем же набором вершин. Для каждого ребра (u,v) в G с пропускной способностью $c(u,v)$ постройте пару ребер (u,v) и (v,u), каждое из которых обладает той же пропускной способностью $c(u,v)$. Решение задачи транспортной сети G_m определяет решение G.

Паросочетания в двудольном графе

Задачи о паросочетаниях имеют множество различных видов. Рассмотрим следующий сценарий. Пять претендентов подали заявления для поступления на пять вакантных рабочих мест. Заявители перечислили вакансии, для занятия которых они имеют достаточную квалификацию. Задача состоит в том, чтобы принять претендентов на работу таким образом, чтобы на каждое место работы был назначен ровно один квалифицированный претендент.

Мы можем использовать для решения задачи паросочетаний в двудольном графе алгоритм Форда–Фалкерсона. Такой метод известен в области компьютерных наук как "приведение задачи". Мы приводим задачу паросочетаний к задаче максимального потока в транспортной сети, показывая, *а*) как отобразить входные данные задачи паросочетаний в двудольном графе на входные данные задачи о максимальном потоке и *б*) как отобразить выходные данные задачи максимального потока в выходные данные задачи о паросочетаниях.

Входные и выходные данные алгоритма

Задача о поиске паросочетаний в двудольном графе состоит из множества из n элементов, где $s_i \in S$; множества из m партнеров, где $t_j \in T$; и множества из p приемлемых пар, где $p_k \in P$. Каждая пара из P связывает элемент $s_i \in S$ с его партнером $t_j \in T$. Множества S и T непересекающиеся, откуда и происходит название задачи.

Выходом алгоритма является множество пар (s_i,t_j), выбранных из исходного множества допустимых пар P. Эти пары представляют максимальное количество возможных паросочетаний. Алгоритм гарантирует, что большего количества пар составить невозможно (хотя могут быть и другие варианты паросочетаний с тем же количеством пар).

Реализация алгоритма

Вместо разработки нового алгоритма для решения этой задачи мы приводим экземпляр задачи паросочетаний в двудольном графе к экземпляру максимального потока в транспортной сети. В задаче паросочетаний в двудольном графе выбор сочетания (s_i, t_j) для элемента $s_i \in S$ с партнером $t_j \in T$ не позволяет элементу s_i или t_j быть выбранным в другой паре. Пусть размер S равен n, а размер T — m. Чтобы получить такое же поведение в графе транспортной сети, построим $G=(V,E)$.

V содержит $n+m+2$ вершины

Каждый элемент s_i отображается на вершину с номером i. Каждый партнер t_j отображается на вершину с номером $n+j$. Создаем новую вершину-исток *src* (с меткой 0) и новую вершину-сток *tgt* (с меткой $n+m+1$).

E содержит $n+m+k$ ребер

Имеется n ребер, соединяющих новую вершину *src* с вершинами, отображенными из S. Имеется m ребер, соединяющих новую вершину *tgt* с вершинами, отображенными из T. Для каждой из k пар $p_k=(s_i,t_j)$ в граф добавляется ребро $(i,n+j)$. Все эти ребра должны иметь пропускную способность, равную 1.

Вычисление максимального потока в графе транспортной сети G дает максимальное решение задачи паросочетания в двудольном графе, как доказано в [20]. В качестве примера рассмотрим рис. 8.6, *а*, на котором предполагается, что две пары, (a,z) и (b,y), образуют максимальное количество пар; соответствующая транспортная сеть, построенная так, как описано выше, показана на рис. 8.6, *б*, на котором вершина 1 соответствует a, вершина 4 соответствует x и т.д. После размышлений мы можем улучшить это решение, выбрав три пары — (a,z), (c,y) и (b,x). Соответствующая корректировка транспортной сети выполняется путем нахождения увеличивающего пути <0,3,5,2,4,7>. Применение этого увеличивающего пути удаляет паросочетание (b,y) и добавляет паросочетания (b,x) и (c,y).

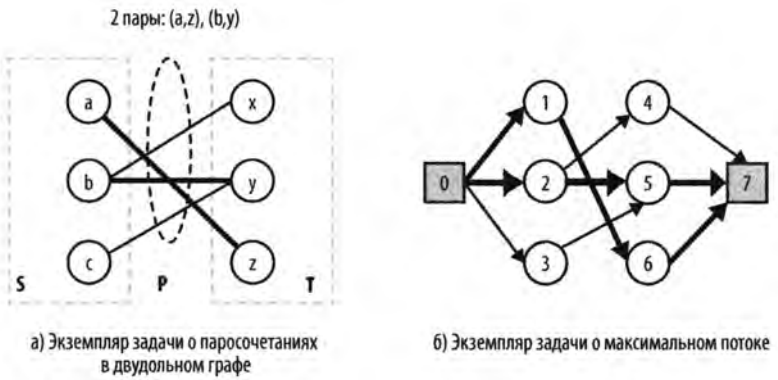


Рис. 8.6. Приведение задачи о паросочетаниях в двудольном графе к задаче о максимальном потоке в транспортной сети

После определения максимального потока в транспортной сети мы преобразуем выход алгоритма поиска максимального потока в выходные данные задачи о паросочетаниях в двудольном графе. Каждое ребро (s_i, t_j), поток которого равен 1, показывает, что выбрано паросочетание $(s_i, t_j) \in P$. В коде, показанном в примере 8.5, для упрощения представления удалена проверка ошибок.

Пример 8.5. Решение задачи о паросочетаниях в двудольном графе с использованием алгоритма Форда-Фалкерсона

```
public class BipartiteMatching
{
    ArrayList<EdgeInfo> edges; /* Ребра для S и T. */
    int ctr = 0;               /* Уникальный счетчик. */
    /* Отображения для приведения экземпляров задач . */
    Hashtable<Object, Integer> map =
            new Hashtable<Object, Integer>();
    Hashtable<Integer, Object> reverse =
            new Hashtable<Integer, Object>();
    int srcIndex;    /* Индекс истока транспортной сети.     */
    int tgtIndex;    /* Индекс стока транспортной сети.      */
    int numVertices; /* Количество вершин транспортной сети. */

    public BipartiteMatching(Object[]S, Object[]T, Object[][]pairs)
    {
        edges = new ArrayList<EdgeInfo>();

        // Преобразование пар во входные данные транспортной
        // сети с пропускными способностями 1.
        for (int i = 0; i < pairs.length; i++)
        {
            Integer src = map.get(pairs[i][0]);
            Integer tgt = map.get(pairs[i][1]);

            if (src == null)
            {
                map.put(pairs[i][0], src = ++ctr);
                reverse.put(src, pairs[i][0]);
            }

            if (tgt == null)
            {
                map.put(pairs[i][1], tgt = ++ctr);
                reverse.put(tgt, pairs[i][1]);
            }

            edges.add(new EdgeInfo(src, tgt, 1));
        }
```

```
        // Добавление вершин истока и стока
        srcIndex = 0;
        tgtIndex = S.length + T.length + 1;
        numVertices = tgtIndex + 1;

        for (Object o : S)
        {
            edges.add(new EdgeInfo(0, map.get(o), 1));
        }

        for (Object o : T)
        {
            edges.add(new EdgeInfo(map.get(o), tgtIndex, 1));
        }
    }

    public Iterator<Pair> compute()
    {
        FlowNetworkArray network = new FlowNetworkArray(numVertices,
                srcIndex, tgtIndex, edges.iterator());
        FordFulkerson solver = new FordFulkerson(network,
                new DFS_SearchArray(network));
        solver.compute();
        // Выборка из исходного множества edgeInfo; игнорируем
        // создание ребер к добавленным вершинам истока и стока.
        // Включаются в решение только при потоке, равном 1.
        ArrayList<Pair> pairs = new ArrayList<Pair>();

        for (EdgeInfo ei : edges)
        {
            if (ei.start != srcIndex && ei.end != tgtIndex)
            {
                if (ei.getFlow() == 1)
                {
                    pairs.add(new Pair(reverse.get(ei.start),
                                       reverse.get(ei.end)));
                }
            }
        }

        return pairs.iterator(); // Итератор, генерирующий решение
    }
}
```

Анализ алгоритма

Для эффективного решения приведения задач нужна возможность эффективно отображать экземпляр задачи и вычисленное решение. Задача паросочетания в двудольном графе $M=(S,T,P)$ преобразуется в граф $G=(V,E)$ за $n+m+k$ шагов. Полученный граф G имеет $n+m+2$ вершин и $n+m+k$ ребер, и, таким образом, размер графа только на константу больше, чем размер исходного двудольного графа. Эта важная особенность построения гарантирует, что у нас есть эффективное решение задачи паросочетаний в двудольном графе. После вычисления максимального потока с помощью алгоритма Форда–Фалкерсона ребра сети с потоком 1 соответствуют искомым парам в задаче паросочетаний. Определение этих ребер выполняется за k шагов, так что дополнительная обработка для получения решения задачи паросочетания в двудольном графе требует времени $O(k)$.

Размышления об увеличивающих путях

Задача о максимальном потоке лежит в основе решения всех остальных задач, приведенных ранее в этой главе на рис. 8.1. Каждой из них требуется выполнить некоторые действия, чтобы представить ее как задачу потока в транспортной сети, после чего можно минимизировать стоимость этого потока. Если мы связываем с каждым ребром (u,v) сети стоимость $d(u,v)$, которая отражает стоимость доставки единицы товара по ребру (u,v), цель будет заключаться в том, чтобы минимизировать сумму

$$\sum f(u,v)\cdot d(u,v)$$

по всем ребрам транспортной сети. Для алгоритма Форда–Фалкерсона мы подчеркивали важность нахождения увеличивающего пути, который позволяет увеличить максимальный поток через сеть. Но что если мы изменим режим поиска, чтобы найти увеличение с наименьшей стоимостью, если таковое существует? Мы уже знакомы с жадными алгоритмами (как, например, алгоритм Прима для построения минимального остовного дерева в главе 6, "Алгоритмы на графах"), которые последовательно выбирают наименее дорогостоящее расширение; возможно, такой подход будет работать и здесь.

Чтобы найти наименее дорогостоящий увеличивающий путь, мы не можем полагаться исключительно на поиск в ширину или глубину. Как мы видели в алгоритме Прима, следует использовать очередь с приоритетами для хранения и вычисления расстояния каждой вершины в транспортной сети потока от вершины-истока. Мы, по сути, вычисляем стоимость доставки дополнительных единиц товара из вершины-истока до каждой вершины сети и поддерживаем очередь с приоритетами, основанную на следующих вычислениях.

1. В процессе поиска очередь с приоритетами хранит упорядоченное множество узлов, которое определяет активный фокус поиска.

2. Для расширения поиска из очереди с приоритетами извлекается вершина u, расстояние которой (в терминах стоимости) от истока является наименьшим. Затем находятся соседние вершины v, которые еще не были посещены и которые соответствуют одному из двух условий: либо *а*) прямое ребро (u, v) все еще имеет пропускную способность, позволяющую увеличить поток, либо *б*) обратное ребро (v, u) имеет поток, который может быть уменьшен.
3. Если во время исследования встречается сток, поиск завершается успешно нахождением увеличивающего пути; в противном случае такого увеличивающего пути не существует.

Реализация метода `ShortestPathArray` на языке программирования Java показана в примере 8.6. Если этот метод возвращает `true`, параметр `vertices` содержит информацию об увеличивающем пути.

Пример 8.6. Поиск кратчайшего (в смысле стоимости) пути для алгоритма Форда–Фалкерсона

```
public boolean findAugmentingPath(VertexInfo[] vertices)
{
    Arrays.fill(vertices, null);  // Сброс для итерации
    // Создание очереди с использованием BinaryHeap. Массив
    // inqueue[] позволяет избежать O(n) поиска для определения
    // наличия элемента в очереди.
    int n = vertices.length;
    BinaryHeap<Integer> pq = new BinaryHeap<Integer> (n);
    boolean inqueue[] = new boolean [n];

    // Инициализация массива dist[]. Используем значение INT_MAX,
    // когда ребро отсутствует.
    for (int u = 0; u < n; u++)
    {
        if (u == sourceIndex)
        {
            dist[u] = 0;
            pq.insert(sourceIndex, 0);
            inqueue[u] = true;
        }
        else
        {
            dist[u] = Integer.MAX_VALUE;
        }
    }

    while (!pq.isEmpty())
    {
        int u = pq.smallestID();
```

```
inqueue[u] = false;

// По достижении sinkIndex работа выполнена.
if (u == sinkIndex)
{
    break;
}

for (int v = 0; v < n; v++)
{
    if (v == sourceIndex || v == u) continue;

    // Прямое ребро с остающейся пропускной
    // способностью и лучшей стоимостью.
    EdgeInfo cei = info[u][v];

    if (cei != null && cei.flow < cei.capacity)
    {
        int newDist = dist[u] + cei.cost;

        if (0 <= newDist && newDist < dist[v])
        {
            vertices[v] = new VertexInfo(u, Search.FORWARD);
            dist[v] = newDist;

            if (inqueue[v])
            {
                pq.decreaseKey(v, newDist);
            }
            else
            {
                pq.insert(v, newDist);
                inqueue[v] = true;
            }
        }
    }

    // Обратное ребро с некоторым потоком
    // и лучшей стоимостью.
    cei = info[v][u];

    if (cei != null && cei.flow > 0)
    {
        int newDist = dist[u] - cei.cost;

        if (0 <= newDist && newDist < dist[v])
        {
            vertices[v] = new VertexInfo(u, Search.BACKWARD);
```

```
                    dist[v] = newDist;

                    if (inqueue[v])
                    {
                        pq.decreaseKey(v, newDist);
                    }
                    else
                    {
                        pq.insert(v, newDist);
                        inqueue[v] = true;
                    }
                }
            }
        }

        return dist[sinkIndex] != Integer.MAX_VALUE;
    }
```

Вооруженные этой стратегией нахождения увеличивающего пути с наименьшей стоимостью, мы теперь можем решить остальные задачи, показанные на рис. 8.1. Чтобы показать эту стратегию поиска низкой стоимости в действии, на рис. 8.7 проиллюстрировано сравнение вычислений максимального потока и потока с минимальной стоимостью для небольшого примера. Каждая итерация цикла `while` в методе `compute()` алгоритма Форда–Фалкерсона перемещает нас на рисунке вниз по вертикали. Результат, показанный в нижней части рисунка, представляет собой максимальный поток, найденный каждым из подходов.

В этом примере вы — менеджер, который отвечает за две фабрики в Чикаго (v_1) и Вашингтоне (v_2), каждая из которых может производить 300 изделий ежедневно. Вы должны гарантировать, что клиенты в Хьюстоне (v_3) и Бостоне (v_4) получают по 300 изделий в день. У вас есть несколько вариантов доставки, показанных на рисунке. Например, вы можете перевозить до 280 изделий между Вашингтоном и Хьюстоном по цене 4 доллара за изделие, но стоимость увеличивается до 6 долларов за изделие, если вы перевозите их из Вашингтона в Бостон (хотя этот маршрут позволяет отправлять до 350 изделий в день).

Может показаться непонятным, как алгоритм Форда–Фалкерсона может использоваться для решения этой задачи, но обратите внимание, что мы можем создать граф G с новой вершиной-истоком s_0, которая соединена с двумя узлами фабрик (v_1 и v_2), а два потребителя (v_3 и v_4) соединены с новой вершиной-стоком t_5. Для экономии места исток s_0 и сток t_5 не показаны. В левой части рис. 8.7 мы выполняем алгоритм Эдмондса–Карпа, чтобы продемонстрировать, что мы можем удовлетворить все запросы наших клиентов со стоимостью ежедневной доставки, равной 3600 долларов. В процессе каждой из четырех итераций алгоритма Форда–Фалкерсона показано влияние увеличивающего пути (обновляемый на данной итерации поток выделен серым цветом).

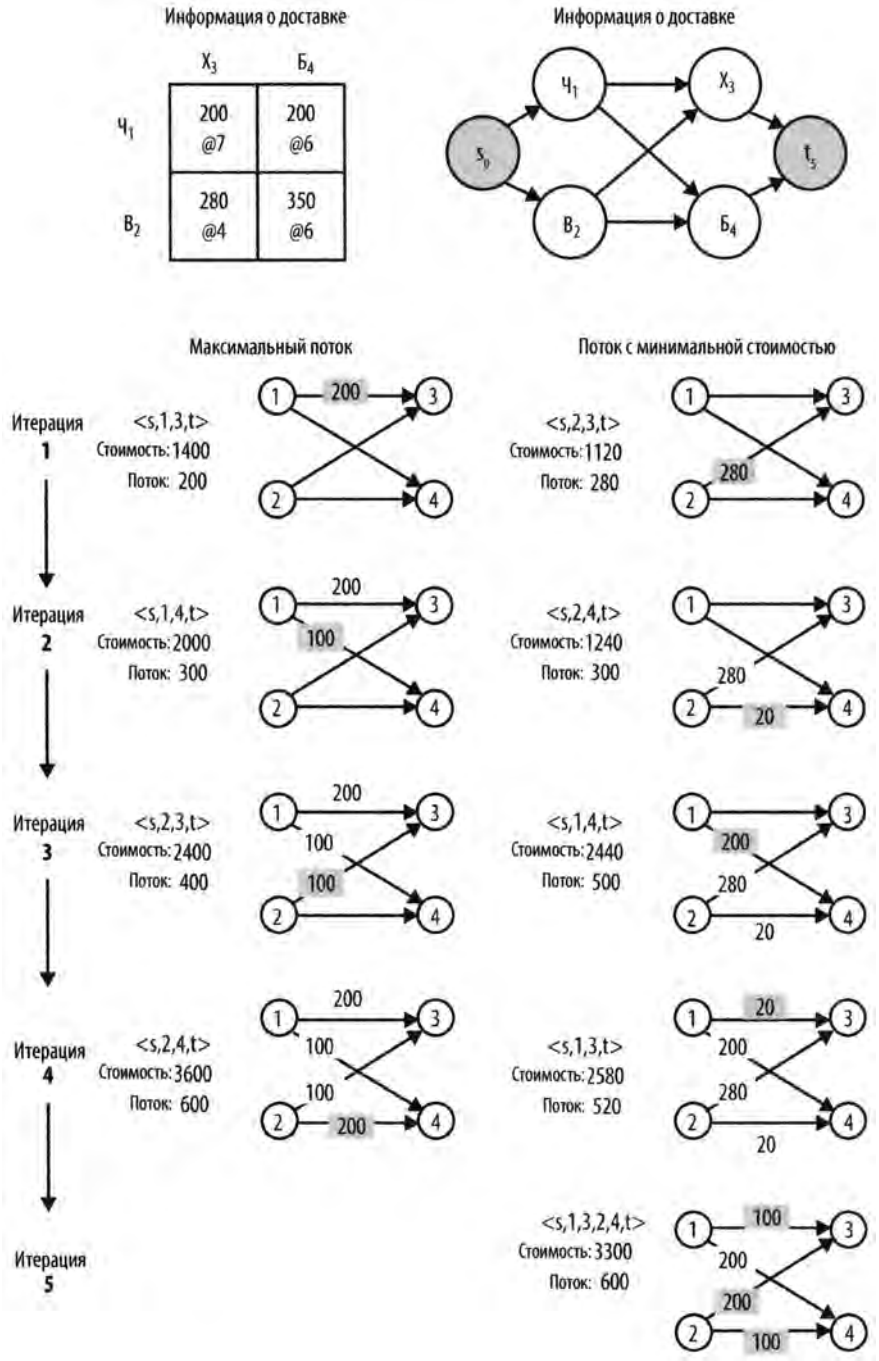


Рис. 8.7. Пошаговые вычисления демонстрируют отличие поиска потока с минимальной стоимостью

Является ли эта стоимость наименьшей, которой мы можем достичь? На рис. 8.7 справа показано выполнение алгоритма Форда–Фалкерсона с использованием в качестве стратегии поиска метода `ShortestPathArray`, описанного в примере 8.6. Обратите внимание, как первый увеличивающий путь использует преимущества самой дешевой доставки. Кроме того, `ShortestPathArray` использует дорогостоящий маршрут доставки из Чикаго (v_1) в Хьюстон (v_3) только тогда, когда нет другого пути для удовлетворения запросов потребителей; когда это происходит, увеличивающий путь уменьшает существующие потоки между Вашингтоном (v_2) и Хьюстоном (v_3), а также между Вашингтоном (v_2) и Бостоном (v_4).

Поток минимальной стоимости

Для решения задачи о потоке минимальной стоимости нам нужно только построить граф транспортной сети и убедиться, что он удовлетворяет рассмотренным ранее требованиям (ограничения пропускной способности, сохранения потока и антисимметрии) и двум дополнительным требованиям (*удовлетворения предложения* и *удовлетворения спроса*). Эти термины пришли из экономического контекста и примерно соответствуют концепциям источника и стока.

Удовлетворение предложения

Для каждой вершины-источника $s_i \in S$ сумма $f(s_i, v)$ для всех ребер $(s_i, v) \in E$ (поток, исходящий из s_i) минус сумма $f(u, s_i)$ для всех ребер $(u, s_i) \in E$ (поток, входящий в s_i) должен быть меньше или равен $sup(s_i)$. Иначе говоря, предложение $sup(s_i)$ в каждой вершине-источнике является точной верхней границей результирующего потока из этой вершины.

Удовлетворение спроса

Для каждой вершины-стока $t_j \in T$ сумма $f(u, t_j)$ для всех ребер $(u, t_j) \in E$ (поток, входящий в t_j) минус сумма $f(t_j, v)$ для всех ребер $(t_j, v) \in E$ (поток, исходящий из t_j) должна быть меньше или равна $dem(t_j)$. Иначе говоря, спрос $dem(t_j)$ в каждой вершине-стоке является точной верхней границей результирующего потока, входящего в эту вершину.

Чтобы упростить алгоритмическое решение, мы ограничим граф транспортной сети одной вершиной-истоком и одной вершиной-стоком. Этого можно легко достичь путем принятия существующего графа транспортной сети с любым количеством истоков и стоков и добавления в него двух новых вершин. Во-первых, добавляются новая вершина s_0, которая является вершиной-истоком для графа транспортной сети, и ребра (s_0, s_i) для всех $s_i \in S$, пропускные способности которых равны $c(s_0, s_i) = sup(s_i)$, а стоимости $d(s_0, s_i) = 0$. Во-вторых, добавляется новая вершина tgt, которая является вершиной-стоком для графа транспортной сети, и ребра (t_j, tgt)

для всех $t_j \in T$, пропускные способности которых равны $c(t_j, tgt) = dem(t_j)$, а стоимости $d(t_0, t_j) = 0$. Как вы можете видеть, добавление этих вершин и ребер не увеличивает стоимость потока сети и не снижает и не увеличивает окончательный вычисляемый поток в сети.

Предложения $sup(s_i)$, спросы $dem(t_j)$ и пропускные способности $c(u,v)$ имеют значения больше 0. Стоимость $d(u,v)$, связанная с каждым ребром, может быть больше или равна нулю. При вычислении результирующего потока все значения $f(u,v)$ будут больше или равны нулю.

Теперь мы представим построения, которые позволят нам решать каждую из оставшихся задач транспортной сети, перечисленных на рис. 8.1. Для каждой задачи мы описываем, как привести ее к задаче потока минимальной стоимости.

Перегрузка

Входными данными для этой задачи являются:

- m пунктов предложения s_i, каждый из которых в состоянии производить $sup(s_i)$ единиц товара;
- n пунктов спроса t_j, каждый из которых потребляет $dem(t_j)$ единиц товара;
- w пунктов перегрузки w_k, каждый из которых может получать и *перегружать* максимум max_k единиц товара с фиксированной стоимостью обработки wp_k за единицу.

Имеется фиксированная стоимость доставки $d(i,j)$ каждой единицы товара от пункта предложения s_i до пункта спроса t_j, фиксированная стоимость перегрузки $ts(i,k)$ для каждой единицы товара, отправленной из пункта s_i на склад w_k, и фиксированная стоимость перегрузки $ts(k,j)$ для каждой единицы товара, отгруженного со склада w_k в пункт спроса t_j. Цель заключается в определении потока $f(i,j)$ единиц товара от пункта предложения s_i к пункту спроса t_j, который сводит к минимуму общую стоимость, которую можно точно определить следующим образом:

$$\text{Общая стоимость}\,(TC) =$$
$$\text{Общая стоимость доставки}\,(TSC) +$$
$$\text{Общая стоимость перегрузки}\,(TTC);$$
$$TSC = \sum_i \sum_j d(i,j) \cdot fi(j, \ );$$
$$TTC = \sum_i \sum_k ts(i,k) \cdot f(i,k) + \sum_j \sum_k ts(j,k) \cdot f(j,k).$$

Наша цель — найти целочисленные значения $f(i,j) \geq 0$, которые обеспечивают минимальное значение TC при удовлетворении всех ограничений предложения и спроса. И наконец, чистый поток единиц товара через каждый склад должен быть

нулевым, чтобы ни одна единица товара не была потеряна (или добавлена). Все предложения $sup(s_i)$ и спросы $dem(t_i)$ товаров больше нуля. Расходы по доставке $d(i,j)$, $ts(i,k)$ и $ts(k,j)$ могут быть больше или равны нулю.

Реализация алгоритма

Мы превращаем экземпляр задачи перегрузки в экземпляр задачи о потоке минимальной стоимости (как показано на рис. 8.8), создавая граф $G=(V,E)$ следующим образом.

V содержит $n+m+2\cdot w+2$ вершин

Каждый пункт предложения s_i отображается на вершину с номером i. Каждый склад w_k сопоставляется с двумя разными вершинами: одна — с номером $m+2\cdot k-1$, вторая — с номером $m+2\cdot k$. Каждый пункт спроса t_j отображается на вершину $1+m+2\cdot w+j$. Создаются также новая вершина-исток *src* (с номером 0) и новая вершина-сток *tgt* (с номером $n+m+2\cdot w+1$).

E содержит $(w+1)\cdot(m+n)+m\cdot n+w$ ребер

Класс `Transshipment` в репозитории кода к книге содержит код построения ребер для экземпляра задачи перегрузки.

Искусственная вершина-исток связана с m вершинами предложения с нулевой стоимостью и пропускной способностью, равной предложению $sup(s_i)$ пункта предложения. Эти m вершин предложения соединены с n вершинами спроса со стоимостями $d(i,j)$ и бесконечными пропускными способностями. n вершин спроса связаны с новой искусственной вершиной-стоком с нулевой стоимостью и пропускной способностью, равной $dem(t_j)$. Имеются также w узлов складов, каждый из которых соединен с m вершинами предложения со стоимостью $ts(i,k)$ и пропускной способностью, равной $sup(s_i)$; эти узлы складов также соединены с n вершинами спроса со стоимостями $ts(k,j)$ и пропускной способностью, равной $dem(t_j)$. И наконец, ребра между складами имеют пропускную способность и стоимость, равную пропускной способности и стоимости складов.

После получения решения задачи о потоке минимальной стоимости расписание перегрузки можно построить путем нахождения тех ребер $(u,v)\in E$, для которых $f(u,v)>0$. Общая стоимость решения представляет собой сумму всех $f(u,v)\cdot d(u,v)$ для этих ребер.

Перевозка

Задача перевозки проще, чем задача перегрузки, потому что в ней нет промежуточных узлов-складов. Входными данными этой задачи являются:

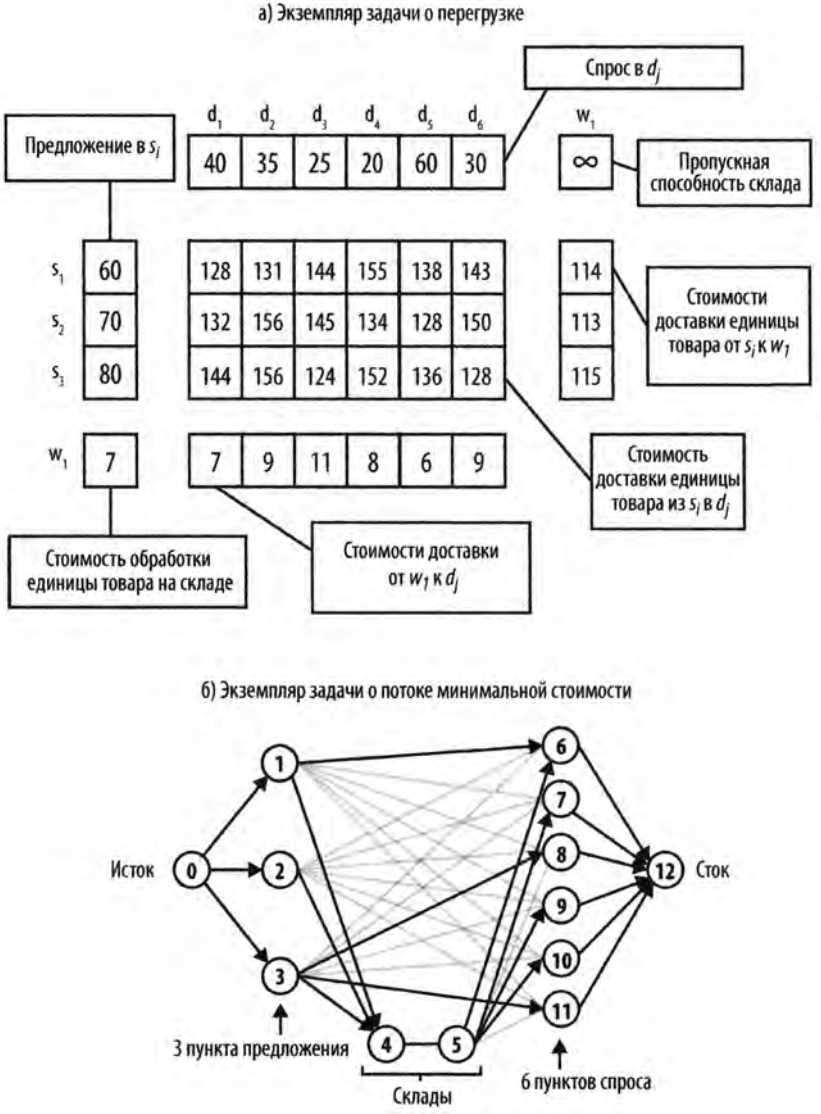


Рис. 8.8. Пример преобразования экземпляра задачи о перегрузке в экземпляр задачи о потоке минимальной стоимости

- m пунктов предложения s_i, каждый из которых в состоянии производить $sup(s_i)$ единиц товара;
- n пунктов спроса t_j, каждый из которых потребляет $dem(t_j)$ единиц товара.

Имеется фиксированная стоимость перевозки единицы товара $d(i,j) \geq 0$, связанная с перевозкой по ребру (i,j). Цель заключается в том, чтобы определить поток $f(i,j)$

товара от пунктов предложения s_i до пунктов спроса t_j, который минимизировал бы общую стоимость перевозки *TSC*, которую можно определить как

$$\text{Общая стоимость перевозки } (TSC) = \sum_i \sum_j d(i,j) \cdot f(i,j).$$

Решение должно удовлетворять всем ограничениям предложения для всех пунктов s_i и спроса для всех t_j.

Реализация алгоритма

Экземпляр задачи перевозки приводится к экземпляру задачи перегрузки с отсутствующими промежуточными складами.

Назначение

Задача назначения представляет собой ограниченную версию задачи транспортировки: каждый пункт предложения должен предоставлять только одну единицу товара, а каждый узел спроса также потреблять только одну единицу.

Реализация алгоритма

Экземпляр задачи назначения преобразуется в экземпляр задачи перевозки с ограничением предложения и спроса каждого узла одной единицей товара.

Линейное программирование

Различные задачи, описанные в этой главе, могут быть решены с использованием *линейного программирования* — мощного метода оптимизации целевой линейной функции при наличии ограничений в виде линейных уравнений и неравенств [8].

Чтобы показать линейное программирование в действии, преобразуем задачу перевозки, изображенную на рис. 8.7, в серию линейных уравнений, которая затем будет решена (мы используем для этого коммерческий пакет математического программного обеспечения общего назначения, известный как Maple (`http://www.maplesoft.com`)). Как вы помните, цель заключается в том, чтобы максимизировать поток по сети при минимизации стоимости. Мы связываем с потоком по каждому ребру в сети переменную; таким образом, переменная `e13` представляет $f(1,3)$. Минимизируемая функция — `Cost`, которая определяется как сумма стоимостей перевозки по каждому из четырех ребер сети. Это уравнение стоимости имеет те же ограничения, которые мы описали ранее для потоков сети.

Сохранение потока

Общая сумма потоков по всем ребрам, исходящим из вершины-истока, должна равняться величине предложения. Общая сумма потоков по всем ребрам, входящим в вершину-сток, должна быть равна величине ее спроса.

Ограничения пропускной способности

Поток по ребру $f(i,j)$ должен быть не меньше нуля, а также удовлетворять неравенству $f(i,j) \leq c(i,j)$.

Применение Maple дает результат {e13 = 100, e24 = 100, e23 = 200, e14 = 200}, который в точности соответствует минимальной стоимости в 3300 долларов, найденной ранее (пример 8.7).

Пример 8.7. Команды Maple для минимизации задачи перевозки

```
Constraints := [
# сохранение единиц товара в каждом узле
e13 + e14   = 300, # Чикаго
e23 + e24   = 300, # Вашингтон
e13 + e23   = 300, # Хьюстон
e14 + e24   = 300, # Бостон

# максимальный поток по отдельным ребрам
0 <= e13, e13 <= 200,
0 <= e14, e14 <= 200,
0 <= e23, e23 <= 280,
0 <= e24, e24 <= 350
];

Cost := 7*e13 + 6*e14 + 4*e23 + 6*e24;

# Решение задачи методами линейного программирования
minimize (Cost, Constraints, NONNEGATIVE);
```

Алгоритм **Simplex**, разработанный Джорджем Данцигом (George Dantzig) в 1947 году, позволяет решать задачи, такие как показанная в примере 8.7, в которых участвуют сотни или тысячи переменных [38]. Simplex-алгоритм многократно показал свою эффективность на практике, хотя данный подход при неудачных обстоятельствах ведет к экспоненциальному количеству вычислений. Не рекомендуется реализовывать этот алгоритм самостоятельно — как из-за его сложности, так и потому, что имеется ряд коммерчески доступных библиотек, которые делают эту работу вместо вас.

9

Вычислительная геометрия

Вычислительная геометрия представляет собой применение математики для точного и эффективного вычисления геометрических структур и их свойств. Мы ограничимся в этой книге решением задач, включающих двумерные структуры, представленные в декартовой системе координат; имеются естественные расширения для n-мерных структур. Математики исследовали такие задачи на протяжении веков, но систематические исследования в этой области начались с 1970-х годов. В настоящей главе представлены вычислительные абстракции, используемые для решения задач вычислительной геометрии. Эти методы отнюдь не ограничиваются проблемами геометрии и имеют много реальных применений.

Алгоритмы этой категории решают множество встречающихся в реальной практике задач.

Выпуклая оболочка

Вычислить наименьшую выпуклую фигуру, которая полностью включает в себя набор n двумерных точек P. Эта задача может быть решена за время $O(n \cdot log\ n)$ вместо времени $O(n^4)$ при применении алгоритма, основанного на "грубой силе".

Пересечение отрезков

Вычислить все пересечения в заданном множестве из n двумерных прямолинейных отрезков S. Эта задача может быть решена за время $O((n+k) \cdot log\ n)$, где k — число пересечений (при применении алгоритма "грубой силы" требуется время $O(n^2)$.

Диаграмма Вороного

Разбиение плоскости на области на основе расстояния до множества из n двумерных точек P. Каждая из n областей состоит из декартовых точек, которые ближе к точке $p_i \in P$, чем к любой другой точке $p_j \in P$. Эта задача может быть решена за время $O(n \cdot log\ n)$.

Попутно мы рассматриваем мощный метод использования сканирующей (выметающей) прямой (Line Sweep), которая может использоваться для решения всех трех перечисленных задач.

Классификация задач

Задачи вычислительной геометрия по своей природе включают геометрические объекты, такие как точки, линии и многоугольники. Задача определяется типом входных данных для обработки, производимыми вычислениями и тем, какой является задача — статической или динамической.

Входные данные

Задача вычислительной геометрии должна определить входные данные. Ниже приведены наиболее распространенные типы обрабатываемых входных данных.

- Точки на двумерной плоскости
- Отрезки на плоскости
- Прямоугольники на плоскости
- Многоугольники на плоскости

Двумерные структуры (линии, прямоугольники и окружности) имеют трехмерные аналоги (плоскости, параллелепипеды и сферы) и даже *n*-мерные аналоги (такие, как гиперплоскости, гиперкубы и гиперсферы). Примеры с более высокими размерностями включают следующее.

Соответствие данных

С помощью своей системы проверки совместимости (патент США № 6735568) служба знакомств eHarmony прогнозирует долгосрочную совместимость между двумя людьми. Все пользователи их системы (по оценкам — 66 миллионов в 2015 году) заполняют анкету из 258 вопросов. Затем eHarmony определяет точность совпадения между двумя людьми на основе 29-мерных данных.

Исправление данных

Входной файл содержит 14 миллионов записей, и каждая запись имеет несколько полей с текстовыми или числовыми значениями. Предполагается, что некоторые из этих значений неверны или отсутствуют. Мы можем вывести или приписать "исправления" для подозрительных (или отсутствующих) значений, находя другие записи, "близкие" к подозрительным.

В этой главе описывается набор базовых интерфейсов для вычислительной геометрии и вводятся классы, которые реализуют эти интерфейсы. Для максимальной совместимости все алгоритмы закодированы с использованием этих интерфейсов.

`IPoint`

Представляет точку в декартовых координатах (x,y) с использованием точности `double`. Реализации предоставляют компаратор по умолчанию, который выполняет сортировку по x слева направо, а неоднозначности разрешаются сортировкой по y снизу вверх.

`IRectangle`

Представляет прямоугольник в декартовых координатах. Реализации определяют, содержит ли он `IPoint` или весь `IRectangle`.

`ILineSegment`

Представляет конечный отрезок в декартовых координатах с фиксированной начальной и конечной точками. В "нормальном положении" начальная точка имеет более высокую координату y, чем конечная точка (за исключением горизонтальных линий — в этом случае в качестве начальной выбирается крайняя слева конечная точка). Реализации могут определить пересечения с другими объектами `ILineSegment` или `IPoint` и то, где находится объект `IPoint` (слева или справа), рассматривая отрезок как направленный от конечной точки к начальной.

Эти концепции естественным образом расширяются на несколько измерений.

`IMultiPoint`

Представляет n-мерную точку с фиксированным числом измерений, каждое значение координаты которой имеет тип `double`. Класс может определить расстояние до другой точки `IMultiPoint` с той же размерностью. Он может возвращать массив значений координат для оптимизации производительности некоторых алгоритмов.

`IHypercube`

Представляет n-мерную фигуру с ограничивающими значениями [*left*, *right*] для фиксированного числа измерений. Класс может определить, пересекается ли гиперкуб с `IMultiPoint` или содержит ли `IHypercube` с той же размерностью.

Каждый из этих типов интерфейса реализуется множеством конкретных классов, используемых для создания фактических объектов (например, класс `TwoDPoint` реализует интерфейсы как `IPoint`, так и `IMultiPoint`).

Значения точек традиционно являются действительными числами, что заставляет реализацию использовать для хранения данных примитивные типы с плавающей точкой. В 1970-х годах вычисления со значениями с плавающей точкой были относительно дорогостоящими по сравнению с вычислениями с целыми числами, но на сегодняшний день они не являются препятствием для высокой производительности. В главе 2, "Математика алгоритмов", рассматриваются важные вопросы, касающиеся вычислений с плавающей точкой, например ошибки округления, которые оказывают влияние на алгоритмы, описанные в этой главе.

Вычисления

Существуют три общие задачи вычислительной геометрии, которые обычно связаны с пространственными вопросами, такими как приведенные в табл. 9.1.

Запрос

Выберите существующие элементы входного множества данных на основе определенного набора ограничений (например, содержащиеся внутри, наиболее близкие или наиболее удаленные). Эти задачи имеют самое непосредственное отношение к алгоритмам поиска, которые обсуждались в главе 5, "Поиск", и будут рассмотрены в главе 10, "Пространственные древовидные структуры".

Вычисления

Выполняет последовательность вычислений над входным множеством данных (например, над множеством отрезков) для создания геометрических структур, которые включают элементы входного множества (например, множества точек пересечения этих отрезков).

Предварительная обработка

Встраивание входного множества в богатую структуру данных, которая будет использоваться для ответа на ряд вопросов. Другими словами, результат предварительной обработки используется в качестве входных данных для целого ряда других вопросов.

Таблица 9.1. Задачи вычислительной геометрии и их приложения

Задачи вычислительной геометрии	Реальные приложения
Найти точку, ближайшую к данной	Для данного местоположения автомобиля найти ближайшую заправку
Найти точку, наиболее удаленную от данной	Для данной станции "Скорой помощи" найти наиболее удаленную больницу из данного списка для определения времени доставки больного в наихудшем случае
Определение, является ли многоугольник простым (т.е. того, что никакие две несмежные стороны не имеют общих точек)	Животное из "Красной книги" помечено с помощью радиопередатчика, который сообщает о местоположении животного. Для поиска часто используемых маршрутов ученые хотели бы знать, когда животное пересекает собственный путь
Вычисление наименьшей окружности, охватывающей множество точек. Вычисление наибольшего внутреннего круга, который *не* содержит ни одной точки	Статистический анализ данных с использованием различных методов. Охватывающие окружности могут определять кластеры, в то время как большие пустоты в данных свидетельствуют об аномальных или отсутствующих данных

Окончание табл. 9.1

Задачи вычислительной геометрии	Реальные приложения
Определение полного множества пересечений во множестве отрезков или во множестве окружностей, прямоугольников или произвольных многоугольников	Проверка правил проектирования СБИС

Природа задачи

Статическая задача требует ответа на поставленный вопрос для определенного набора входных данных. Однако два динамических соображения изменяют возможный способ решения задачи.

- Если требуется решение нескольких задач для одного набора входных данных, то предварительная обработка этих входных данных может повысить эффективность решения каждой из задач.
- При изменении набора входных данных рассмотрите структуры данных, которые эффективно выполняют вставки и удаления.

Динамические задачи требуют применения структур данных, которые могут увеличиваться и уменьшаться в соответствии с требованиями изменения входных данных. Для статических задач вполне пригодны массивы фиксированной длины, но динамические задачи могут потребовать использования связанных списков или стеков.

Предположения

Для большинства задач вычислительной геометрии эффективное решение начинается с анализа предположений и инвариантов, относящихся ко входным данным (или решаемой задаче), например таких.

- Могут ли во входном наборе отрезков быть горизонтальные или вертикальные отрезки?
- Могут ли во входном наборе точек быть коллинеарные наборы из трех точек? Другими словами, могут ли быть в наборе три точки, лежащие на одной прямой? Если нет, то алгоритм не должен обрабатывать никакие частные случаи, связанные с наличием коллинеарных точек.
- Содержит ли входной набор данных равномерно распределенные точки? Или эти точки могут быть расположены асимметрично или быть кластеризованными таким образом, что могут приводить к наихудшему поведению алгоритма?

Большинство алгоритмов, представленных в этой главе, имеют редкие граничные случаи, которые резко усложняют их реализацию; эти ситуации будут указаны в примерах кода.

Выпуклая оболочка

Для заданного набора двухмерных точек P выпуклой оболочкой является наименьшая выпуклая фигура, которая полностью охватывает все точки P (например, отрезок между любыми двумя точками внутри оболочки полностью находится в этой оболочке). Оболочка формируется путем вычисления по часовой стрелке упорядоченного набора из h точек из P, которые мы обозначаем как L_0, L_1, ..., L_{h-1}. Хотя любая точка может быть первой (L_0), алгоритмы обычно используют в качестве начальной крайнюю слева точку из множества P; другими словами, точку с наименьшей координатой x. Если в P имеется несколько таких точек, выбирается точка с наименьшей координатой y.

Если у нас имеется n точек, то они образуют

$$C_n^3 = \frac{n(n-1)(n-2)}{6}$$

различных возможных треугольников. Точка $p_i \in P$ не может быть частью выпуклой оболочки, если она содержится внутри треугольника, образованного тремя другими различными точками P. Например, на рис. 9.1 точка p_6 может быть устранена из выпуклой оболочки путем рассмотрения треугольника, образованного точками p_4, p_7 и p_8. Для каждого из таких треугольников T_i работающий методом "в лоб" алгоритм **медленной оболочки** (Slow Hull) может устранить из оболочки любую из n–3 остальных точек, если она лежит в T_i.

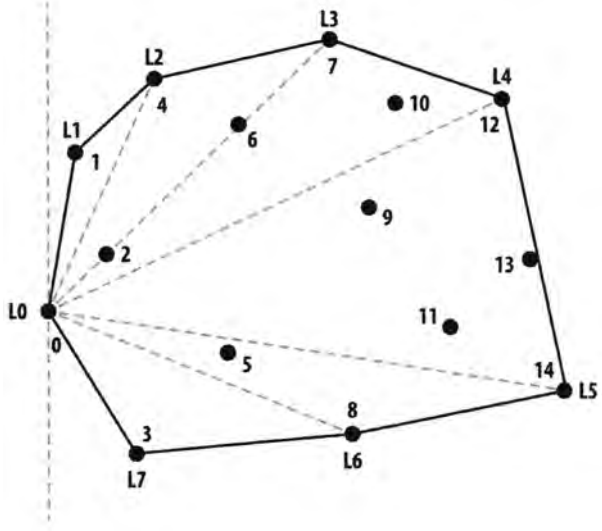


Рис. 9.1. Пример множества точек на плоскости и их выпуклая оболочка

После того как становятся известными точки оболочки, алгоритм помечает крайнюю слева точку как L_0 и сортирует все остальные точки по значению угла, образованного вертикальной линией, проходящей через точку L_0, и линией, проведенной от точки L_0 до рассматриваемой. Каждая последовательность из трех точек оболочки L_i, L_{i+1}, L_{i+2} образует правый поворот (обратите внимание, что это же свойство выполняется и для точек L_{h-2}, L_{h-1}, L_0).

Этот неэффективный подход требует выполнения $O(n^4)$ проверок принадлежности точки треугольнику. Далее мы представим эффективный алгоритм получения выпуклой оболочки путем сканирования, который вычисляет выпуклую оболочку за время $O(n \cdot log\ n)$.

Сканирование выпуклой оболочки

Алгоритм **сканирования выпуклой оболочки** (Convex Hull Scan), открытый Эндрю (Andrew) в 1979 году [3], делит эту задачу на задачу частичного построения верхней и нижней частей оболочки, а затем объединяет их решения. Во-первых, все точки сортируются по их координате x (в случае совпадения координаты x первой идет точка с меньшей координатой y). Верхняя часть оболочки начинается с двух крайних слева точек множества P. Далее алгоритм расширяет верхнюю часть оболочки, находя точку $p \in P$, координата x которой будет следующей в отсортированном порядке после последней точки частично построенной верхней части оболочки L_i. Вычисление нижней части оболочки выполняется аналогично; для получения окончательной выпуклой оболочки следует объединить полученные частичные результаты вместе с их конечными точками.

Если три точки, L_{i-1}, L_i и точка-кандидат p, образуют правый поворот, алгоритм расширяет частично построенную оболочку, включая в нее точку p. Это решение эквивалентно вычислению определителя матрицы 3 × 3, показанной на рис. 9.2, который представляет векторное произведение cp. Если $cp < 0$, то три точки определяют правый поворот, и сканирование выпуклой оболочки продолжается. Если $cp = 0$ (три точки коллинеарны) или если $cp > 0$ (три точки определяют левый поворот), то средняя точка L_i должна быть удалена из частично построенной выпуклой оболочки для сохранения ее свойства выпуклости. Алгоритм вычисляет верхнюю часть выпуклой оболочки, обрабатывая все точки до крайней справа. Аналогично вычисляется и нижняя часть оболочки (на этот раз точки выбираются в порядке уменьшения значения координаты x), и две полученные части оболочки соединяются между собой.

$$cp = \begin{vmatrix} L_{i-1}.x & L_{i-1}.y & 1 \\ L_i.x & L_i.y & 1 \\ p.x & p.y & 1 \end{vmatrix}$$

$$cp = (L_i.x - L_{i-1}.x)(p.y - L_{i-1}.y) - (L_i.y - L_{i-1}.y)(p.x - L_{i-1}.x)$$

Рис. 9.2. Вычисление определителя, указывающего наличие правого поворота

Сканирование выпуклой оболочки

Наилучший, средний и наихудший случаи: $O(n \cdot \log n)$

```
convexHull(P)
    Сортировка P по возрастанию координаты x (при
            одинаковых значениях x - по координате y)          ❶
    if n < 3 then return P

    upper = {p0, p1}                                           ❷
    for i = 2 to n-1 do
        Добавление pi в upper
        while последние три точки в upper образуют левый поворот do ❸
            Удаление средней из трех последних точек из upper  ❹

    lower = {pn-1, pn-2}                                       ❺
    for i = n-3 downto 0 do
        Добавление pi в lower
        while последние три точки в lower образуют левый поворот do
            Удаление средней из трех последних точек из lower

    Соединение upper и lower (с удалением дублей конечных точек) ❻
    return computed hull
```

❶ Сортировка точек вносит наибольшую стоимость в алгоритм.
❷ Эти две точки предлагаются как входящие в верхнюю часть оболочки.
❸ Левый поворот означает, что последние три точки
образуют вогнутый угол.
❹ Средняя точка является ошибочной и должна быть удалена.
❺ Аналогичная процедура вычисления нижней части оболочки.
❻ "Сшивание" частей оболочки в единую оболочку.

На рис. 9.3 показан алгоритм сканирования выпуклой оболочки в действии при вычислении верхней части выпуклой оболочки. Обратите внимание, что данный подход делает многочисленные ошибки при посещении каждой точки P слева направо; оболочка при обнаружении таких ошибочных построений изменяется путем удаления — иногда многократного — средней из последних трех точек; в конечном итоге после исправления всех ошибок алгоритм правильно вычисляет верхнюю часть оболочки.

Входные и выходные данные алгоритма

Входные данные алгоритма представляют собой множество P двумерных точек на плоскости.

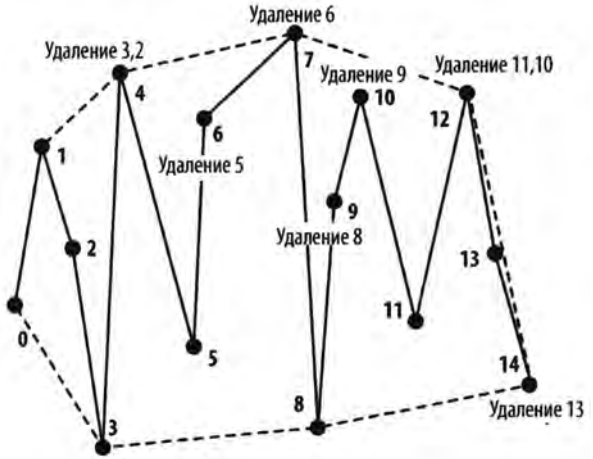


Рис. 9.3. Пример построения верхней части выпуклой оболочки

Алгоритм сканирования выпуклой оболочки вычисляет упорядоченный список L, содержащий h вершин выпуклой оболочки P, упорядоченных по часовой стрелке. Выпуклая оболочка представляет собой многоугольник, определяемый точками L_0, L_1, ..., L_{h-1}, где h — количество точек в L. Обратите внимание, что многоугольник образован h отрезками $<L_0,L_1>$, $<L_1,L_2>$, ..., $<L_{h-1}, L_0>$.

Чтобы избежать тривиальных решений, мы предполагаем $|P| \geq 3$. Никакие две точки не располагаются "слишком близко" одна к другой (определяется реализацией). Если две точки расположены слишком близко одна к другой и одна из них является точкой выпуклой оболочки, то алгоритм может неправильно выбрать точку выпуклой оболочки (или удалить из нее корректную точку); впрочем, разница при этом будет незначительной.

Контекст применения алгоритма

Алгоритм требует выполнения только примитивных операций (таких, как умножение и деление), что обеспечивает более легкую его реализацию по сравнению со **сканированием Грэхема**, в котором используются тригонометрические функции (см. главу 3, "Строительные блоки алгоритмов"). Сканирование выпуклой оболочки может работать с большим количеством точек, поскольку он не является рекурсивным.

Более быстрая реализация возможна, если входной набор точек является равномерно распределенным и, таким образом, их можно отсортировать за время $O(n)$ с помощью блочной сортировки (так что итоговая производительность также будет представлять собой $O(n)$). Если такой информации у нас нет, мы выбираем пирамидальную сортировку для достижения производительности $O(n \cdot log\ n)$ при начальной

сортировке точек. Код в репозитории к данной книге содержит каждую из описанных реализаций (производительности которых мы будем сравнивать позже).

Реализация алгоритма

В примере 9.1 показано, как алгоритм сканирования выпуклой оболочки сначала вычисляет верхнюю часть выпуклой оболочки, а затем меняет направление и вычисляет нижнюю ее часть. Окончательная полная выпуклая оболочка является объединением этих двух частичных оболочек.

Пример 9.1. Реализация алгоритма сканирования выпуклой оболочки

```
public class ConvexHullScan implements IConvexHull
{
    public IPoint [] compute(IPoint[] points)
    {
        // Сортировка по координате x (при равенстве – по y).
        int n = points.length;
        new HeapSort<IPoint>().sort(points,0,n-1,IPoint.xy_sorter);

        if (n < 3)
        {
            return points;
        }

        // Вычисление верхней части выпуклой оболочки начиная
        // с двух крайних слева точек
        PartialHull upper = new PartialHull(points[0], points[1]);

        for (int i = 2; i < n; i++)
        {
            upper.add(points[i]);

            while (upper.hasThree() && upper.areLastThreeNonRight())
            {
                upper.removeMiddleOfLastThree();
            }
        }

        // Вычисление нижней части выпуклой оболочки начиная
        // с двух крайних справа точек
        PartialHull lower = new PartialHull(points[n-1],points[n-2]);

        for (int i = n - 3; i >= 0; i--)
        {
            lower.add(points[i]);
```

```
            while (lower.hasThree() && lower.areLastThreeNonRight())
            {
                lower.removeMiddleOfLastThree();
            }
        }

        // Удаление дубликатов точек при слиянии
        IPoint[] hull = new IPoint[upper.size() + lower.size() - 2];
        System.arraycopy(upper.getPoints(), 0, hull, 0, upper.size());
        System.arraycopy(lower.getPoints(), 1, hull,
                         upper.size(), lower.size() - 2);
        return hull;
    }
}
```

Поскольку первый шаг этого алгоритма состоит в сортировке точек, мы полагаемся на пирамидальную сортировку для достижения наилучшей средней производительности, не рискуя столкнуться с наихудшим поведением быстрой сортировки. Однако в среднем случае быстрая сортировка будет опережать пирамидальную сортировку.

Эвристика Экла–Туссена может заметно повысить производительность алгоритма путем удаления всех точек, находящихся в экстремальном четырехугольнике (из точек с минимальными и максимальными координатами x и y), вычисляемом из входного набора данных (рис. 9.4; здесь показан экстремальный четырехугольник для набора точек на рис. 9.1; удаленные точки показаны серым цветом; ни одна из этих точек не может принадлежать выпуклой оболочке).

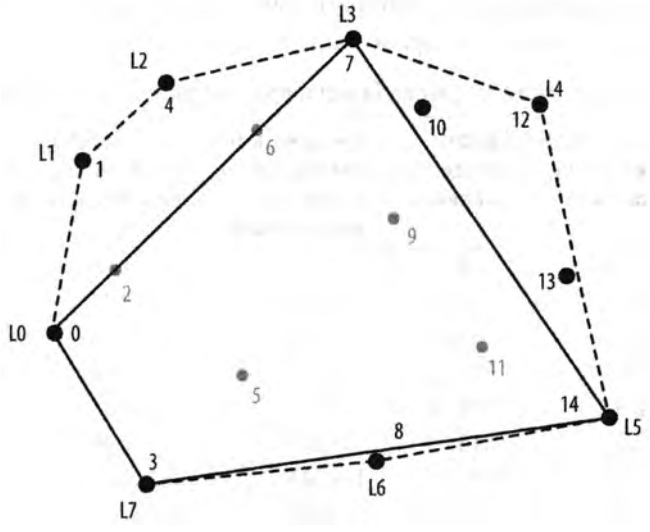


Рис. 9.4. Эвристика Экла–Туссена

Чтобы определить, находится ли точка p в экстремальном четырехугольнике, представим луч s от p до бесконечно удаленной точки ($p.x$, $-\infty$) и подсчитаем, сколько раз s пересекает стороны четырехугольника. Если значение счетчика окажется равным 1, точка p находится внутри четырехугольника и может быть удалена. Реализация обрабатывает особые случаи, например, когда луч s точно пересекается с одной из вершин экстремального четырехугольника. Вычисление требует фиксированного количества шагов $O(1)$, а это означает, что эвристика Экла–Туссена обрабатывает все точки за время $O(n)$. Для больших случайных данных эта эвристика может удалить почти половину точек, а поскольку они удаляются до выполнения сортировки, стоимость дорогостоящего шага сортировки в алгоритме снижается.

Анализ алгоритма

Мы выполнили набор из 100 испытаний над множеством случайно сгенерированных двумерных точек на единичном квадрате и отбросили лучшие и худшие результаты. В табл. 9.2 показаны усредненные результаты для оставшихся 98 испытаний. В таблице также показаны среднее время выполнения эвристики и некоторая информация о решении, которая поясняет, почему сканирование выпуклой оболочки является столь эффективным алгоритмом.

С увеличением размера входных данных около половины точек могут быть удалены с помощью эвристики Экла–Туссена. Возможно, еще более удивительно малое количество точек на выпуклой оболочке. Во втором столбце в табл. 9.2 проверяется утверждение Препараты (Preparata) и Шамоса (Shamos) [47] о том, что количество точек на выпуклой оболочке должно быть порядка $O(\log n)$. Естественно, играет роль и распределение; например, если выбирать точки равномерно из единичной окружности, то выпуклая оболочка содержит порядка кубического корня из n точек.

Таблица 9.2. Время работы алгоритма (в мс) и применение эвристики Экла–Туссена

n	Среднее количество точек в оболочке	Среднее время вычисления	Среднее количество точек, удаленных эвристикой	Среднее время вычисления эвристики	Среднее время вычисления с эвристикой
4096	21,65	8,95	2023	1,59	4,46
8192	24,1	18,98	4145	2,39	8,59
16384	25,82	41,44	8216	6,88	21,71
32768	27,64	93,46	15687	14,47	48,92
65536	28,9	218,24	33112	33,31	109,74
131072	32,02	513,03	65289	76,36	254,92
262144	33,08	1168,77	129724	162,94	558,47
524288	35,09	2617,53	265982	331,78	1159,72
1048576	36,25	5802,36	512244	694	2524,30

Первый шаг в сканировании выпуклой оболочки объясняет стоимость $O(n \cdot log\ n)$, когда точки сортируются с помощью одного из стандартных методов на основе сравнения, описанных в главе 4, "Алгоритмы сортировки". Цикл `for`, который вычисляет верхнюю часть выпуклой оболочки, обрабатывает $n-2$ точки; внутренний цикл `while` не может выполняться более $n-2$ раз. Такая же логика применима и к циклу, который вычисляет нижнюю часть выпуклой оболочки. Таким образом, общее время оставшихся шагов алгоритма сканирования выпуклой оболочки равно $O(n)$.

Проблемы арифметики с плавающей точкой проявляются в алгоритме сканирования выпуклой оболочки при вычислении векторного произведения. Вместо строгого сравнения векторного произведения $cp < 0$ `PartialHull` выполняет проверку $cp < \delta$, где значение δ равно 10^{-9}.

Вариации алгоритма

Сортировку в работе алгоритма сканирования можно устранить, если известно, что точки уже находятся в отсортированном порядке; в этом случае сканирование выпуклой оболочки может быть выполнено за время $O(n)$. В качестве альтернативного решения, если входные точки равномерно распределены, можно использовать блочную сортировку (см. главу 4, "Алгоритмы сортировки"), чтобы достичь производительности $O(n)$. Другой вариант построения выпуклой оболочки, известный как **QuickHull** [22], навеянный алгоритмом быстрой сортировки, использует стратегию "разделяй и властвуй", в состоянии вычислить выпуклую оболочку за время $O(n)$ при условии равномерно распределенных точек.

Есть еще один вариант алгоритма, который следует рассмотреть. Алгоритм сканирования выпуклой оболочки на самом деле не требует отсортированного массива точек при создании верхней части оболочки; он просто должен перебрать все точки P по порядку от наименьшей координаты x до самой большой. Именно такое поведение наблюдается, если создать бинарную пирамиду из точек P и многократно удалять из нее наименьший элемент. Если удаленные точки хранятся в связанном списке, то они могут быть просто "прочитаны" из связанного списка для обработки в обратном порядке, справа налево. Код для этого варианта (на рис. 9.5 помечен как *Пирамида*) доступен в репозитории кода к данной книге.

Результаты исследования производительности различных вариаций алгоритма сканирования выпуклой оболочки для двух вариантов распределения точек показаны на рис. 9.5.

Распределение по окружности

n точек распределены равномерно по краю единичной окружности. Все эти точки принадлежат выпуклой оболочке, так что мы имеем дело с экстремальным случаем.

Равномерное распределение

n точек распределены равномерно в единичном квадрате. По мере увеличения *n* все бо́льшая часть точек не является частью оболочки, так что мы получаем второй экстремальный случай.

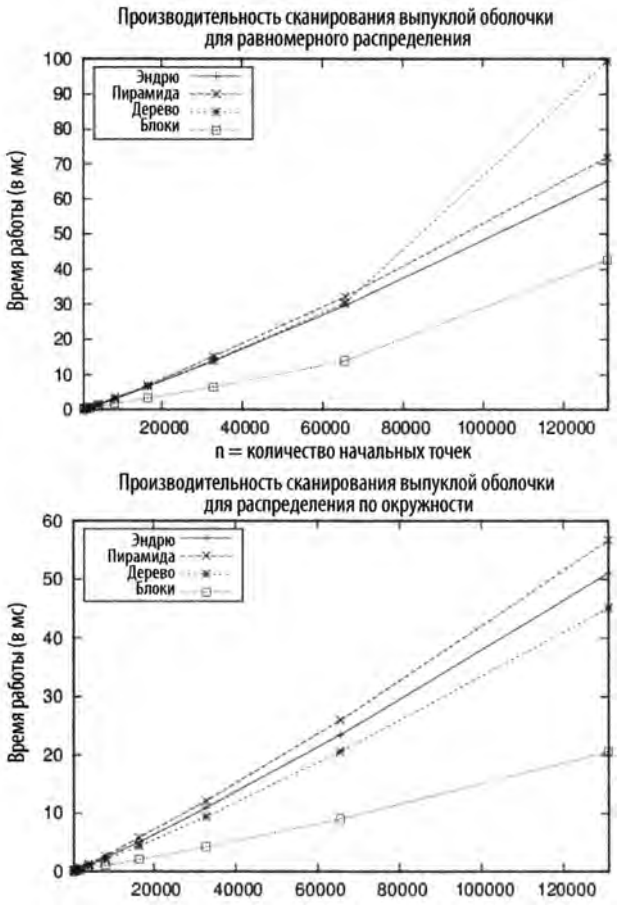


Рис. 9.5. Производительность вариаций алгоритма сканирования выпуклой оболочки

Мы провели серию испытаний с использованием наборов данных с размером от 512 до 131 072 точек, для двух вариантов распределений, с различными реализациями, описанными в примере 9.1 и репозитории кода к данной книге. Мы не использовали эвристику Экла–Туссена. Для каждого размера набора данных мы выполнили 100 испытаний и отбросили наилучший и наихудший результаты. Результирующее среднее время (в миллисекундах) для оставшихся 98 испытаний показано на рис. 9.5. Реализация с помощью сбалансированных бинарных деревьев

показывает наилучшую производительность среди подходов на основе сортировки методом сравнения. Обратите внимание, что реализация с использованием блочной сортировки предполагает наиболее эффективную реализацию, но только потому, что входной набор точек получается с помощью равномерного распределения. В общем случае вычисление выпуклой оболочки может быть выполнено за время $O(n \cdot log\ n)$.

Однако эти реализации также должны страдать низкой производительностью при асимметричных входных данных. Рассмотрим n неравномерно распределенных точек — (0,0), (1,1) и n–2 точки, кластерированные в виде тонких полосок слева от координаты 0,502. Этот набор данных построен таким образом, чтобы испортить поведение блочной сортировки. В табл. 9.3 показано, как блочная сортировка вырождается в алгоритм $O(n^2)$, поскольку использует сортировку вставками для сортировки блоков.

Задача выпуклой оболочки может быть расширена на три измерения и выше, где цель заключается в том, чтобы вычислить многогранник, ограничивающий трехмерное пространство с трехмерными точками. К сожалению, в более высоких измерениях требуются существенно более сложные реализации.

Таблица 9.3. Сравнение времен работы (в мс) для высокоасимметричных данных

n	Эндрю	Пирамида	Дерево	Блоки
512	0,28	0,35	0,33	1,01
1024	0,31	0,38	0,41	3,30
2048	0,73	0,81	0,69	13,54

Мелкманом (Melkman) в 1987 году разработан алгоритм, который генерирует выпуклую оболочку для простой ломаной линии или многоугольника за время $O(n)$. Этот алгоритм позволяет избежать необходимости в начальной сортировке точек, используя упорядоченное расположение точек в самом многоугольнике.

Выпуклую оболочку можно эффективно поддерживать, используя подход, предложенный Овермарсом (Overmars) и ван Льювеном (van Leeuwen) в 1981 году. Точки выпуклой оболочки хранятся в древовидной структуре, которая поддерживает удаление и вставку точек. Стоимость вставки или удаления, как известно, равна $O(log^2 n)$, поэтому общая стоимость построения выпуклой оболочки составляет $O(n \cdot log^2 n)$, и при этом по-прежнему требуется только $O(n)$ памяти. Этот результат является еще одним подтверждением принципа, что каждый выигрыш производительности представляет собой некоторый компромисс.

Сканирование Грэхема представляет собой один из первых алгоритмов построения выпуклой оболочки, разработанный в 1972 году с использованием простых тригонометрических тождеств. Мы уже описывали этот алгоритм в главе 3, "Строительные блоки алгоритмов". Использование вычисления определителя матрицы, показанное ранее, позволяет реализовать алгоритм с помощью только простых структур данных и базовых математических операций. Сканирование Грэхема вычисляет

выпуклую оболочку за время $O(n \cdot log\ n)$, потому что сначала этот алгоритм сортирует точки по углам, которые образуются лучом от точки $s \in P$ с наименьшей координатой y до рассматриваемой, и осью x. Одна из проблем реализации этого алгоритма — точки с одинаковым углом должны быть упорядочены по расстоянию от точки s.

Вычисление пересечения отрезков

Для заданного множества из n прямолинейных отрезков S в двумерной плоскости может потребоваться определить полный набор точек пересечений между всеми отрезками. В примере на рис. 9.6 во множестве из четырех отрезков есть два пересечения (показаны в виде маленьких кругов). Как показано в примере 9.2, подход на основе грубой силы вычисляет все $C_n^2 = n(n-1)/2$ пересечения отрезков в S за время $O(n^2)$. Для каждой пары отрезков реализация выводит точки их пересечения, если таковая существует.

Рис. 9.6. Четыре отрезка с двумя пересечениями

Пример 9.2. Реализация вычисления пересечений методом "в лоб"

```
public class BruteForceAlgorithm extends IntersectionDetection
{
    public Hashtable<IPoint, List<ILineSegment>> intersections
            (ILineSegment[] segments)
    {
        initialize();

        for (int i = 0; i < segments.length - 1; i++)
        {
            for (int j = i + 1; j < segments.length; j++)
            {
                IPoint p = segments[i].intersection(segments[j]);

                if (p != null)
                {
                    record(p, segments[i], segments[j]);
                }
            }
        }
```

```
        return report;
    }
}
```

Это вычисление требует вычисления $O(n^2)$ отдельных пересечений и может потребовать применения сложных тригонометрических функций.

Не сразу очевидно, что возможно улучшение времени работы алгоритма по сравнению с $O(n^2)$. В данной главе представлен инновационный алгоритм использования сканирующей (выметающей) прямой (**LineSweep**), который в среднем позволяет вычислить результаты пересечений за время $O((n+k)\cdot log\ n)$, где k — количество точек пересечения.

LineSweep

Существуют многочисленные ситуации, когда нам нужно обнаружить пересечения между геометрическими фигурами. При проектировании СБИС требуются точные схемы расположения элементов на монтажной плате, и при этом не должно быть незапланированных пересечений. При планировании маршрутов набор дорог может храниться в базе данных как множество отрезков, а перекрестки рассматриваются как пересечения отрезков.

На рис. 9.7 показан пример семи пересечений шестью отрезков. Вероятно, мы не должны сравнивать все возможные $C_n^2 = n(n-1)/2$ пары отрезков. В конце концов, отрезки, явно удаленные один от другого, пересекаться никак не могут (как, например, S1 и S4). Алгоритм LineSweep представляет собой подход, который доказанно повышает эффективность за счет сосредоточения в процессе работы внимания только на подмножестве входных элементов. Представьте себе горизонтальную линию L, "выметающую" весь входной набор линейных отрезков от верхней до нижней горизонтальной границы и сообщающую о встреченных пересечениях. На рис. 9.7 показано состояние выметающей линии L при перемещении сверху вниз (в девяти различных конкретных положениях).

Нововведение LineSweep заключается в признании того факта, что отрезки для каждой конкретной координаты y могут быть упорядочены слева направо. В случае горизонтальных отрезков считаем, что левая конечная точка располагается "выше", чем правая конечная точка. В таком случае пересечение отрезков может иметь место *только между соседними отрезками в состоянии линии выметания*. В частности, чтобы два отрезка, s_i и s_j, имели точку пересечения, в процессе выметания должен быть момент, когда эти отрезки являются соседями. Алгоритм LineSweep может эффективно находить пересечения путем поддержки этого состояния линии.

Рассматривая подробнее девять выбранных положений горизонтальной линии выметания на рис. 9.7 (показаны пунктиром), вы увидите, что все они находятся в 1) начале или конце отрезка или 2) на пересечении отрезков. LineSweep в действительности не выполняет "выметание" в декартовой плоскости; вместо этого она помещает

$2n$ конечных точек отрезков в очередь событий, которая представляет собой модифицированную очередь с приоритетами. Все пересечения, включающие начальные или конечные точки существующих отрезков, могут быть обнаружены при обработке этих точек. Алгоритм LineSweep обрабатывает очередь, чтобы построить те состояния линии выметания L, которые позволят определить пересечения соседних отрезков.

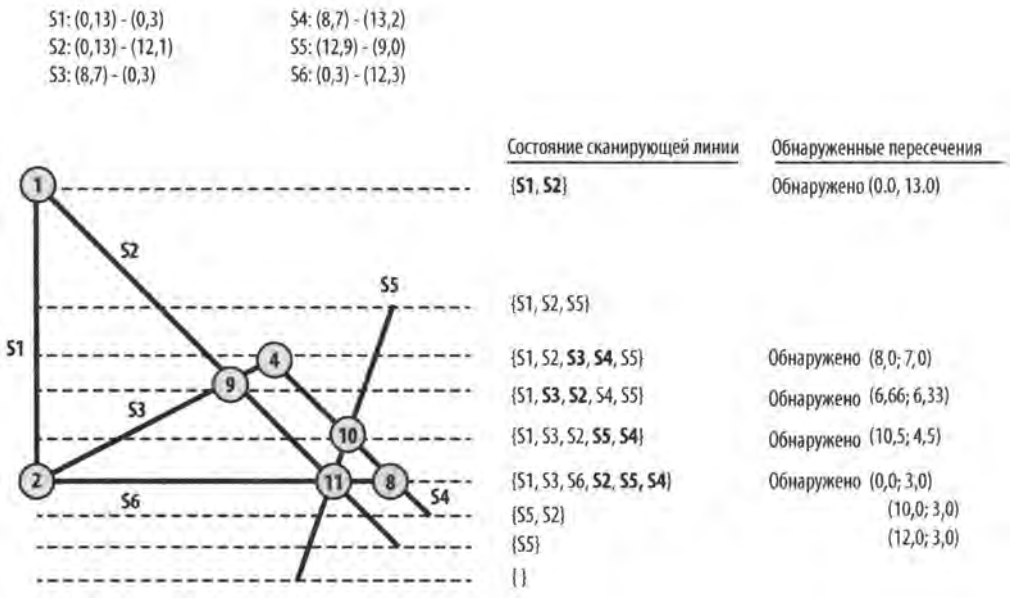


Рис. 9.7. Определение семи пересечений шести отрезков

Входные и выходные данные алгоритма

Алгоритм LineSweep обрабатывает множество из n прямолинейных отрезков S в декартовой системе координат. В S не может быть повторяющихся отрезков; кроме того, в S нет коллинеарных отрезков (т.е. перекрывающихся один с другим и имеющих одинаковый наклон). Алгоритм корректно обрабатывает горизонтальные и вертикальные отрезки путем выполнения точных вычислений и соответствующего упорядочения отрезков. Ни один отрезок не может быть одной точкой (т.е. отрезком, начальная и конечная точки которого совпадают).

Выходные данные алгоритма содержат k точек, представляющих собой точки пересечения (если таковые существуют) между отрезками и, для каждой из этих k точек, отрезки из S, которые пересекаются в этой точке.

Контекст применения алгоритма

Если ожидаемое количество пересечений гораздо меньше, чем количество отрезков, этот алгоритм успешно превосходит подход с использованием грубой силы. При

значительном числе пересечений накладные расходы алгоритма могут перевешивать его преимущества.

Подход, основанный на выметании, полезен, когда вы можете *а*) эффективно строить состояния линии и *б*) управлять очередью событий, которая определяет, когда следует рассматривать выметающую линию. В рамках реализации алгоритма LineSweep необходимо рассматривать множество частных случаев, а результирующий код оказывается намного сложнее, чем код для подхода грубой силы, наихудшая производительность которого — $O(n^2)$. Этот алгоритм может быть выбран только из-за ожидаемой более высокой производительности и улучшения поведения в наихудшем случае.

Алгоритм LineSweep последовательно производит частичные результаты до тех пор, пока не будут обработаны все входные данные и не будут получены все результаты. В данном примере состояние линии представляет собой сбалансированное бинарное дерево отрезков, которое оказывается возможным потому, что мы можем наложить отношение упорядочения на отрезки в выметающей линии. Очередь событий также может быть простым сбалансированным бинарным деревом лексикографически отсортированных точек событий. Лексикографическая сортировка означает, что первыми располагаются точки с более высоким значением *y* (поскольку выметающая линия движется по декартовой плоскости сверху вниз); при наличии двух точек с одним и тем же значением *y* первой идет точка с меньшим значением *x*.

Для упрощения кодирования алгоритма бинарное дерево, используемое для хранения состояния линии, представляет собой расширенное сбалансированное бинарное дерево, в котором фактические сведения содержат только конечные узлы. Внутренние узлы хранят информацию о крайнем слева отрезке в левом поддереве и крайнем справа отрезке в правом поддереве. Упорядочение отрезков в дереве выполняется на основании *точки выметания* (sweep point), текущей точки `EventPoint` из очереди с приоритетами, обрабатываемой в настоящий момент.

Алгоритм LineSweep

Наилучший, средний и наихудший случаи: $O((n+k)\cdot\log n)$

```
intersection (S)
    EQ = new EventQueue
    foreach s in S do                                          ❶
        ep = найти в EQ s.start или создать и вставить в EQ
        Добавить s в ep.upperLineSegments                      ❷

        ep = найти в EQ s.end или создать и вставить в EQ
        Добавить s в ep.lowerLineSegments
```

```
    state = новая lineState
    while очередь EQ не пустая do
        handleEvent (EQ, state, getMin(EQ))
end

handleEvent (EQ, state, ep)
    left = отрезок в state слева от ep
    right = отрезок в state справа от ep
    Вычислить пересечения в state между left и right        ❸

    Удалить отрезки из state между left и right
    Переместить state вниз до ep

    if в ep начинается новый отрезок then                   ❹
        Вставить новый отрезок в state
        update = true
    if пересечения связаны с ep then                        ❺
        Вставить пересечение в state
        update = true
    if update then
        updateQueue (EQ, left, left successor)
        updateQueue (EQ, right, right predecessor)
    else
        updateQueue (EQ, left, right)
end

updateQueue (EQ, A, B)                                      ❻
    if соседние отрезки A и B пересекаются ниже точки выметания then
        Вставить точку их пересечения в EQ
end

❶ Инициализация очереди событий до 2*n точек.
❷ Точки событий ссылаются на отрезки (верхние и нижние конечные точки).
❸ Все пересечения осуществляются между соседними отрезками.
❹ Поддержка состояния линии по мере обнаружения новых отрезков
   ниже выметающей линии.
❺ При пересечении позиции соседних отрезков обмениваются местами.
❻ Добавляем пересечение в очередь событий, только если оно
   ниже выметающей линии.
```

Реализация алгоритма

Решение, представленное в примере 9.3, зависит от классов `EventPoint`, `EventQueue` и `LineState`, которые находятся в репозитории к книге.

Пример 9.3. Реализация алгоритма LineSweep на языке программирования Java

```
public class LineSweep extends IntersectionDetection
{
    // Состояние выметающей линии и очередь событий
    LineState lineState = new LineState();
    EventQueue eq = new EventQueue();
    /** Вычисление пересечений всех отрезков из массива. */
    public Hashtable<IPoint, ILineSegment[]> intersections(
        ILineSegment[] segs)
    {
        // Построение EventQueue из отрезков.
        for (ILineSegment ils : segs)
        {
            EventPoint ep = new EventPoint(ils.getStart());
            EventPoint existing = eq.event(ep);

            if (existing == null)
            {
                eq.insert(ep);
            }
            else
            {
                ep = existing;
            }

            // Добавление верхних концов отрезков в ep
            // (объект в очереди)
            ep.addUpperLineSegment(ils);
            ep = new EventPoint(ils.getEnd());
            existing = eq.event(ep);

            if (existing == null)
            {
                eq.insert(ep);
            }
            else
            {
                ep = existing;
            }

            // Добавление нижних концов отрезков в ep
            // (объект в очереди)
            ep.addLowerLineSegment(ils);
        }

        // Выметание сверху вниз,
        // обработка каждой EventPoint в очереди.
```

```
    while (!eq.isEmpty())
    {
        EventPoint p = eq.min();
        handleEventPoint(p);
    }

    // Возврат отчета обо всех вычисленных пересечениях
    return report;
}

// Обработка событий путем обновления состояния линии
// и сообщение о пересечении.
private void handleEventPoint(EventPoint ep)
{
    // Поиск отрезков, если таковые существуют, слева (и справа)
    // от ep в состоянии линии. Пересечения могут быть только
    // между соседними отрезками.
    AugmentedNode<ILineSegment> left = lineState.leftNeighbor(ep);
    AugmentedNode<ILineSegment> right = lineState.rightNeighbor(ep);

    // Определяем пересечения 'ints' соседних отрезков, получаем
    // верхние концы отрезков 'ups' и нижние концы отрезков 'lows'
    // для данной точки события. Пересечение существует, если с
    // точкой события связано больше одного отрезка.
    lineState.determineIntersecting(ep, left, right);
    List<ILineSegment> ints = ep.intersectingSegments();
    List<ILineSegment> ups  = ep.upperEndpointSegments();
    List<ILineSegment> lows = ep.lowerEndpointSegments();

    if (lows.size() + ups.size() + ints.size() > 1)
    {
        record(ep.point, new List[] { lows, ups, ints });
    }

    // Удаляем все точки после left до тех пор, пока преемником
    // left не станет right. Затем обновляем точку выметания,
    // так что пересечения будут упорядочены. Вставляем только
    // ups и ints, поскольку они остаются активными.
    lineState.deleteRange(left, right);
    lineState.setSweepPoint(ep.point);
    boolean update = false;

    if (!ups.isEmpty())
    {
        lineState.insertSegments(ups);
        update = true;
    }
```

```
    if (!ints.isEmpty())
    {
        lineState.insertSegments(ints);
        update = true;
    }

    // Если состояние демонстрирует отсутствие пересечений в этой
    // точке события, смотрим, не пересекаются ли левый и правый
    // отрезки под выметающей линией, и соответствующим образом
    // обновляем очередь событий. В противном случае при наличии
    // пересечения порядок отрезков между left и right меняется,
    // так что мы проверяем два диапазона, а именно — left и его
    // (новый) преемник и right и его (новый) предшественник.
    if (!update)
    {
        if (left != null && right != null)
        {
            updateQueue(left, right);
        }
    }
    else
    {
        if (left != null)
        {
            updateQueue(left, lineState.successor(left));
        }

        if (right != null)
        {
            updateQueue(lineState.pred(right), right);
        }
    }
}
// Любое пересечение ниже выметающей линии вставляется
// как точка события.
private void updateQueue(AugmentedNode<ILineSegment> left,
                         AugmentedNode<ILineSegment> right)
{
    // Если два соседних отрезка пересекаются, следует убедиться,
    // что новая точка пересечения находится *ниже* выметающей
    // линии и не добавляется дважды.
    IPoint p = left.key().intersection(right.key());

    if (p == null)
    {
        return;
    }
```

```
        if (EventPoint.pointSorter.compare(p, lineState.sweepPt) > 0)
        {
            EventPoint new_ep = new EventPoint(p);

            if (!eq.contains(new_ep))
            {
                eq.insert(new_ep);
            }
        }
    }
}
```

Когда `EventQueue` инициализируется с $2n$ объектами `EventPoint`, каждый из них хранит объекты отрезков `ILineSegment`, которые начинаются и заканчиваются в сохраненных объектах `IPoint`. Когда `LineSweep` обнаруживает пересечение между отрезками линии, в `EventQueue` вставляется `EventPoint`, представляющий это пересечение, *при условии, что это пересечение находится ниже выметающей линии*. Так ни одно пересечение не будет пропущено, и никакие пересечения не будут дублированы. Чтобы реализация алгоритма работала корректно, если эта точка события пересечения уже имеется в `EventQueue`, информация о пересечении обновляется прямо в очереди, а не вставляется в нее дважды. Именно по этой причине алгоритм LineSweep должен быть в состоянии определить, содержит ли очередь событий определенный объект `EventPoint`.

На рис. 9.7, когда точка события, представляющая нижнюю точку отрезка S6 (технически это его *правый* конец, потому что отрезок S6 — горизонтальный), вставляется в очередь с приоритетами, LineSweep хранит S6 только как нижний конец. После его обработки дополнительно сохраняется отрезок S4 — как пересекающийся. В более сложном случае, когда в очередь с приоритетами вставляется точка события, представляющая пересечение отрезков S2 и S5, в этот момент она не хранит никакой дополнительной информации. Но после того, как эта точка события будет обработана, она будет хранить отрезки S6, S2 и S5 как пересекающиеся.

Вычислительной системой алгоритма LineSweep является класс `LineState`, который поддерживает текущую точку в процессе выметания декартовой плоскости сверху вниз. Когда из `EventQueue` извлекается минимальная запись, компаратор `pointSorter` корректно возвращает объекты `EventPoint` сверху вниз и слева направо.

Главная работа алгоритма LineSweep происходит в методе `determineIntersecting` класса `LineState`: здесь путем итерации отрезков между `left` и `right` вычисляются пересечения. Полную информацию об этом вспомогательном классе можно найти в репозитории кода к данной книге.

Алгоритм LineSweep достигает производительности $O((n + k) \cdot log\ n)$ потому, что в процессе выметания он может изменять порядок активных отрезков. Если

выполнение этого шага требует больше времени, чем $O(\log s)$, где s — общее количество отрезков в состоянии, то производительность алгоритма вырождается в $O(n^2)$. Например, если состояние линии хранится просто как двусвязный список (структура, полезная для быстрого поиска предшественника и преемника отрезка), то операции вставки для поиска нужного отрезка в списке будет требоваться время $O(s)$, и с ростом количества отрезков s деградация производительности станет заметной.

Кроме того, очередь событий должна поддерживать эффективную операцию определения наличия точки события в очереди. Применение реализации очереди с приоритетами на основе пирамиды — как, например, предлагает `java.util.PriorityQueue` — также заставляет алгоритм вырождаться в $O(n^2)$. Остерегайтесь кода, который утверждает, что реализует алгоритм с производительностью $O(n \cdot \log n)$, а на самом деле обеспечивает производительность лишь $O(n^2)$!

Анализ алгоритма

Алгоритм LineSweep вставляет до $2n$ конечных точек отрезков в очередь событий, которая является модифицированной очередью с приоритетами, которая поддерживает выполнение перечисленных далее операций за время $O(\log q)$, где q — количество элементов в очереди.

min

Удаление минимального элемента из очереди.

insert(e)

Вставка e в корректное местоположение упорядоченной очереди.

member(e)

Проверка, является ли e членом очереди. Эта операция не является строго необходимой в очереди с приоритетами общего назначения.

В очереди событий находятся только уникальные точки: другими словами, если одна и та же точка события будет вставлена повторно, ее информация будет объединена с информацией точки события, уже находящейся в очереди. Таким образом, когда точки на рис. 9.7 вставляются первоначально, очередь содержит восемь точек событий.

Алгоритм LineSweep выполняет выметание сверху вниз и обновляет состояние линии путем добавления и удаления отрезков в надлежащем порядке. На рис. 9.7 упорядоченное состояние линии отражает отрезки, которые пересекают выметающую линию слева направо, после обработки точки события. Чтобы правильно вычислить пересечения, алгоритм LineSweep определяет отрезок сегмент в состоянии слева (или справа) от данного отрезка s_i. Алгоритм LineSweep использует расширенное сбалансированное бинарное дерево для выполнения перечисленных далее операций за время $O(\log t)$, где t — количество элементов в дереве.

insert(s)

Вставка отрезка *s* в дерево.

delete(s)

Удаление отрезка *s* из дерева.

previous(s)

Возврат отрезка, находящегося в упорядочении непосредственно перед *s*, если таковой существует.

successor(s)

Возврат отрезка, находящегося в упорядочении непосредственно после *s*, если таковой существует.

Чтобы должным образом поддерживать упорядочение отрезков, алгоритм LineSweep меняет порядок отрезков при обнаружении в процессе выметания пересечения между отрезками s_i и s_j; к счастью, это действие также может быть выполнено за время $O(\log t)$ простым путем обновления точки выметающей линии с последующим удалением и повторной вставкой отрезков s_i и s_j. На рис. 9.7, например, такой обмен происходит, когда обнаружено третье пересечение (6,66;6,33).

Этап инициализации алгоритма создает очередь с приоритетами из $2n$ (начальных и конечных) точек из входного набора из n отрезков. Очередь событий дополнительно должна иметь возможность определять, имеется ли уже точка p в очереди; по этой причине мы не можем просто использовать пирамиду для хранения очереди событий, как это обычно делается в случае очередей с приоритетами. Поскольку очередь упорядочена, мы должны определить упорядочение для двумерных точек. Для точек выполняется соотношение $p_1 < p_2$, если $p_1.y > p_2.y$; однако если $p_1.y = p_2.y$, то $p_1 < p_2$, если $p_1.x < p_2.x$. Размер очереди никогда не может быть бо́льшим, чем $2n+k$, где k — количество пересечений, а n — количество входных отрезков.

Все точки пересечения, обнаруженные алгоритмом LineSweep ниже выметающей линии, добавляются в очередь событий, где во время их обработки (когда выметающая линия достигнет точки пересечения) будет изменен порядок пересекающихся отрезков. Обратите внимание, что все пересечения между соседними отрезками будут находиться ниже выметающей линии и ни одна точка пересечения не будет пропущена.

Во время обработки алгоритмом LineSweep каждой точки события происходит добавление отрезков в состояние линии при посещении их верхних конечных точек и удаление при посещении нижних конечных точек. Таким образом, состояние линии никогда не будет хранить более n отрезков. Операции, исследующие состояние линии, могут быть выполнены за время $O(\log n)$, а поскольку над состоянием линии не выполняется больше чем $O(n+k)$ операций, общая стоимость составляет $O((n+k)\cdot\log(n+k))$. Так как k не превышает $C_n^2 = n(n-1)/2$, производительность

алгоритма составляет $O((n+k)\cdot log\ n)$, что в наихудшем случае становится равно $O(n^2\cdot log\ n)$.

Производительность алгоритма LineSweep зависит от сложных свойств входных данных (т.е. от общего количества пересечений и среднего количества отрезков, поддерживаемых выметающей линией в любой конкретный момент). Мы можем оценить производительность для конкретного экземпляра задачи и входных данных. Две такие задачи мы рассмотрим прямо сейчас.

Имеется интересная математическая задача о том, как вычислить приближенное значение π, используя только набор зубочисток и кусок бумаги (известная как задача Бюффона о бросании иглы). Если все зубочистки имеют одну и ту же длину *len*, нарисуйте на бумаге ряд параллельных линий на расстоянии *d* одна от другой ($d \geq len$). Случайным образом бросьте *n* зубочисток на бумагу; пусть *k* — число пересечений зубочисток с параллельными линиями на бумаге. Оказывается, что вероятность того, что зубочистка пересекает линию (которая может быть оценена как k/n), равна $(2\cdot len)/(\pi\cdot d)$.

Когда число пересечений гораздо меньше n^2, метод грубой силы тратит основное время на проверку не пересекающихся отрезков (как показано в табл. 9.4). При наличии большого количества пересечений определяющим фактором будет являться среднее количество отрезков, поддерживаемых `LineState` при работе алгоритма LineSweep. Когда оно невелико (как и следует ожидать при случайных отрезках на плоскости), алгоритм LineSweep будет демонстрировать более высокую скорость работы.

Таблица 9.4. Сравнение времени работы (в мс) разных алгоритмов в задаче Бюффона о бросании иглы

n	LineSweep	Метод грубой силы	Среднее количество пересечений	Оценка π	± Ошибка
16	1,77	0,18	0,84	3,809524	9,072611
32	0,58	0,10	2,11	3,033175	4,536306
64	0,45	0,23	3,93	3,256997	2,268153
128	0,66	0,59	8,37	3,058542	1,134076
256	1,03	1,58	16,20	3,164400	0,567038
512	1,86	5,05	32,61	3,146896	0,283519
1024	3,31	18,11	65,32	3,149316	0,141760
2048	7,00	67,74	131,54	3,149316	0,070880
4096	15,19	262,21	266,16	3,142912	0,035440
8192	34,86	1028,83	544,81	3,128210	0,017720

В качестве второй задачи рассмотрим множество *S*, в котором имеется $O(n^2)$ пересечений отрезков. Здесь алгоритм LineSweep оказывается серьезно "испорченным" из-за накладных расходов на поддержание состояния линии при таком большом количестве пересечений. В табл. 9.5 показано, как метод грубой силы превосходит алгоритм LineSweep. Здесь *n* — количество отрезков, пересечения которых создают $C_n^2 = n(n-1)/2$ точек пересечения.

Таблица 9.5. Сравнение наихудшего случая алгоритма LineSweep с методом грубой силы (в мс)

n	LineSweep (в среднем)	Метод грубой силы (в среднем)
2	0,17	0,03
4	0,66	0,05
8	1,11	0,08
16	0,76	0,15
32	1,49	0,08
64	7,57	0,38
128	45,21	1,43
256	310,86	6,08
512	2252,19	39,36

Вариации алгоритма

Интересный вариант осуществляется тогда, когда от алгоритма требуется указать только одну из точек пересечения, а не все точки. Этот алгоритм полезен, например, при определении, пересекаются ли два многоугольника. Такой алгоритм требует меньшего времени — только $O(n \cdot log\ n)$ — и в среднем случае может находить первое пересечение более быстро. Другой вариант задачи рассматривает входной набор красных и синих отрезков, и нас интересуют только точки пересечения отрезков разного цвета.

Диаграмма Вороного

В 1986 году метод выметающей линии был применен Форчуном (Fortune) для решения другой проблемы вычислительной геометрии, а именно — построения диаграммы Вороного для множества точек P в декартовой системе координат. Эта диаграмма используется во множестве различных дисциплин — от биологии до экономики [5].

Диаграмма Вороного разбивает плоскость на области на основе расстоянии точек плоскости до множества P, состоящего из n двумерных точек. Каждая из n областей состоит из декартовых точек, находящихся ближе к точке $p_i \in P$, чем к любой иной точке $p_j \in P$. На рис. 9.8 показана вычисленная диаграмма Вороного (черные линии) для 13 точек (показанных как квадраты). Диаграмма Вороного состоит из 13 выпуклых областей, образованных ребрами (линии на рисунке) и вершинами (точками, где эти линии пересекаются). Имея диаграмму Вороного для данного набора точек, можно решить следующие задачи.

- Вычисление выпуклой оболочки.
- Поиск наибольшего пустого круга среди точек.
- Поиск ближайшего соседа каждой точки.
- Поиск двух ближайших точек во множестве.

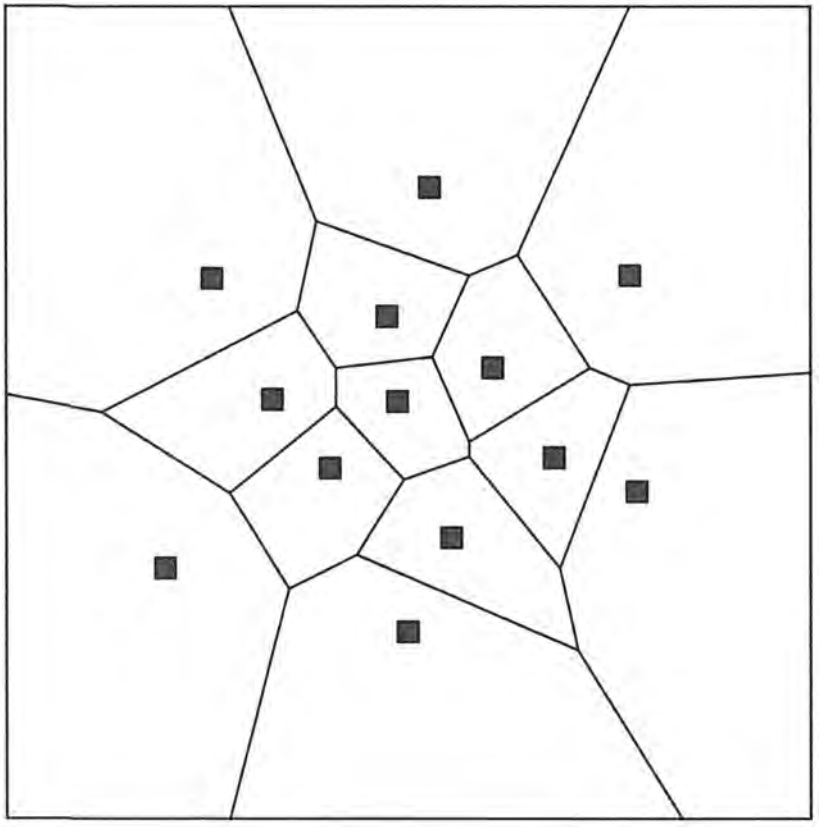


Рис. 9.8. Пример диаграммы Вороного

Алгоритм **выметания Форчуна** реализует выметающую линию, похожую на использующуюся для определения пересечения отрезков. Напомним, что выметающая линия вставляет существующие точки в очередь с приоритетами и обрабатывает эти точки в определенном порядке. Алгоритм поддерживает состояние строки, которое может эффективно обновляться для определения диаграммы Вороного. В алгоритме выметания Форчуна ключевым моментом является наблюдение, что выметающая линия делит плоскость на три отдельные области, как показано на рис. 9.9.

По мере перемещения выметающей линии вниз по плоскости формируется частичная диаграмма Вороного; на рис. 9.9 область, связанная с точкой p_2, полностью вычислена как полубесконечный многоугольник, ограниченный четырьмя участками прямых линий (два из которых представляют собой отрезки, а другие два — лучи). Выметающая линия в показанный момент готова к обработке точек p_6, а точки от p_7 до p_{11} ожидают обработки. Структура состояния строки в указанный момент времени поддерживает точки $\{p_1, p_4, p_5$ и $p_3\}$.

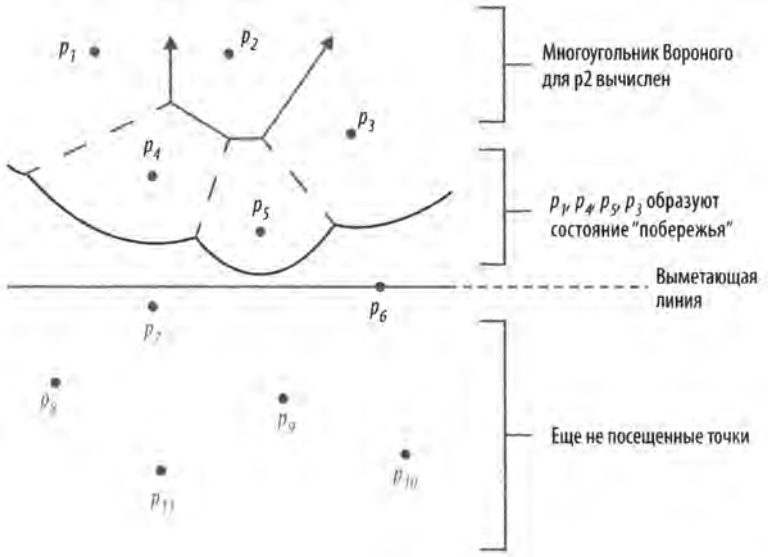


Рис. 9.9. Элементы выметания Форчуна

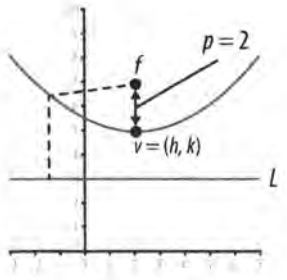
Самым сложным в понимании выметания Форчуна является то, что состояние линии представляет собой сложную структуру, именуемую *побережьем* (beach line). На рис. 9.9 побережьем является набор криволинейных отрезков (дуг) слева направо; точка, в которой встречаются две параболы, называется *точка останова* (breakpoint), а пунктирные линии представляют собой частичные ребра диаграммы Вороного, которые еще должны быть подтверждены. Каждая из точек в состоянии линии побережья определяет параболу относительно выметающей линии. Линия побережья определяется как пересечение парабол, находящихся ближе всего к выметающей линии.

Чтобы объяснить структуру дуг линии побережья, мы должны определить геометрическую форму *параболы*. Для данной точки фокуса f и прямой линии L парабола представляет собой множество точек, которые равноудалены от фокуса f и прямой линии L. Вершина параболы $v=(h,k)$ является самой низкой точкой параболы. Значение p представляет собой расстояние между прямой L и v, а также между v и f. Используя эти значения, можно записать уравнение параболы $4p(y-k)=(x-h)^2$, которое легко визуализировать, если L является горизонтальной линией, а парабола открыта вверх, как показано на рис. 9.10.

Выметание начинается с верхней точки P и выполняется сверху вниз, открывая для обработки точки-*узлы* (site). Параболы в линии побережья по мере продвижения выметающей линии вниз изменяют форму, как показано на рис. 9.11. Это означает, что точки останова также изменяют свое местоположение. К счастью, алгоритм обновляет выметающую линию только $O(n)$ раз.

Вершина в диаграмме Вороного вычисляется путем определения трех точек в P, которые лежат на окружности, не содержащей каких-либо других точек P. Центр этой окружности определяет вершину диаграммы Вороного, потому что эта точка

равноудалена от указанных трех точек. Три луча, исходящих из данного центра, становятся ребрами диаграммы Вороного, потому что эти линии представляют собой множество точек, которые находятся на равном расстоянии от двух точек коллекции. Эти ребра *делят пополам* хорды окружности, соединяющие указанные точки. Например, прямая L_3 на рис. 9.12 перпендикулярна к отрезку, который соединяет точки r_1 и r_3.



$$4p(y-k)=(x-h)^2$$

Рис. 9.10. Определение параболы

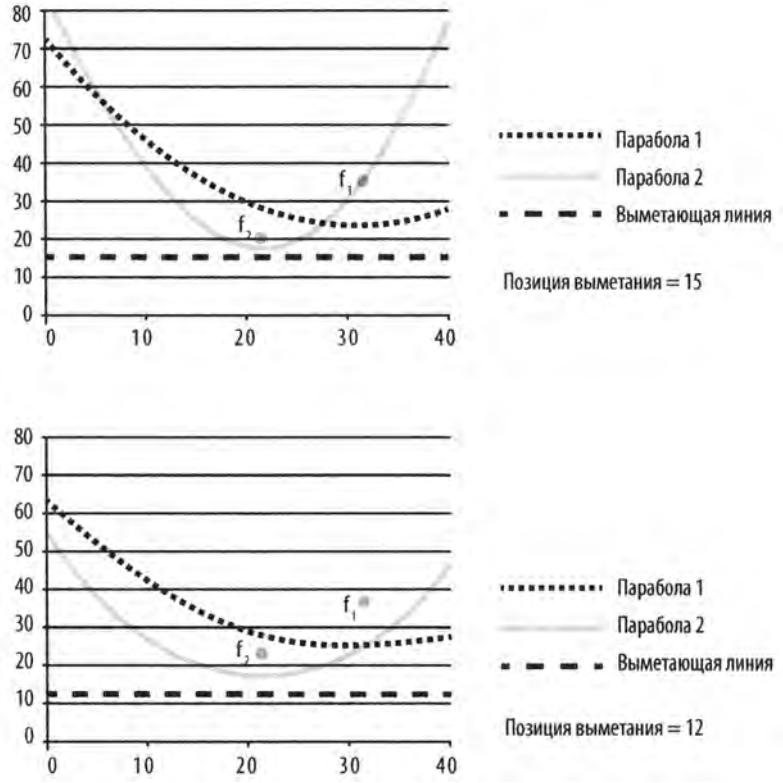


Рис. 9.11. Изменение формы парабол при перемещении вниз выметающей линии

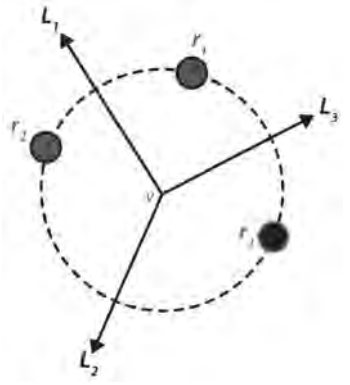


Рис. 9.12. Окружность, образованная тремя точками

Теперь мы покажем, как алгоритм выметания Форчуна хранит информацию о состоянии линии для обнаружения этих окружностей. Характеристики линии побережья минимизируют количество проверок окружностей алгоритмом выметания Форчуна; в частности, всякий раз при обновлении линии побережья алгоритм должен проверять только дуги, соседние (слева и справа) с местом, где произошло обновление. На рис. 9.12 показана механика выметания Форчуна только с тремя точками. Эти три точки обрабатываются в порядке сверху вниз, а именно — в порядке r_1, r_2 и r_3. Окружность становится определенной после того, как обработана третья точка, и известна как окружность, описанная вокруг этих трех точек.

На рис. 9.13 показано состояние побережья после обработки точек r_1 и r_2. Линия побережья формируется частями парабол, которые ближе всего к выметающей линии; в этом случае состояние выметающей линии представлено в виде бинарного дерева, листы которого указывают соответствующие дуги параболы, и внутренние узлы представляют точки останова. Линия побережья слева направо сформирована из трех параболических дуг, s_1, s_2 и вновь s_1, которые взяты из парабол, связанных с точками r_1, r_2 и опять r_1. Точка останова s_1:s_2 представляет координату по оси x, слева от которой ближе к выметающей линии оказывается парабола s_1, а справа от нее к выметающей линии ближе парабола s_2. Такие же характеристики имеет и точка останова s_2:s_1.

На рис. 9.14 показан момент, когда выметающая линия обрабатывает третью точку, r_3. В соответствии со своим расположением вертикальная линия, проходящая через r_3, пересекает линию побережья в крайней справа дуге s_1; обновленное состояние линии побережья показано в правой части рис. 9.14. У нас имеется четыре внутренних узла, не являющихся листьями, которые представляют четыре пересечения между тремя параболами линии побережья. Есть также пять листьев, которые представляют пять дуг парабол, образующих слева направо линию побережья.

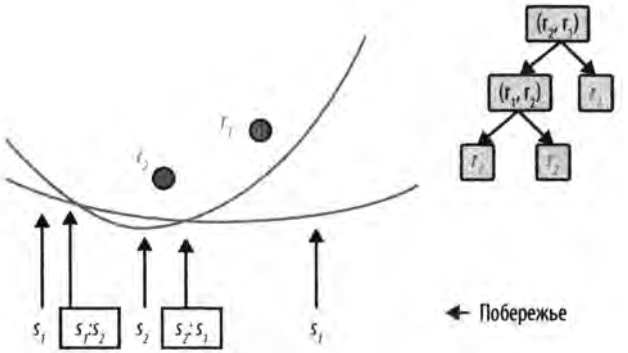


Рис. 9.13. Линия побережья после двух точек

После формирования линии побережья заметим, что эти три точки образуют описывающую окружность. Центр этой окружности потенциально может стать вершиной диаграммы Вороного, но это может быть только в том случае, если внутри данного круга нет другой точки из множества *P*. Алгоритм элегантно обрабатывает эту ситуацию путем создания *события окружности*, координата *y* которого совпадает с самой низкой точкой этой окружности (рис. 9.15), и вставки этого события в очередь с приоритетами. Если до данного события окружности должны быть обработаны некоторые другие события узлов, то данное событие окружности будет устранено. В противном случае оно будет обработано в свою очередь, и центральная точка данного круга станет вершиной диаграммы Вороного.

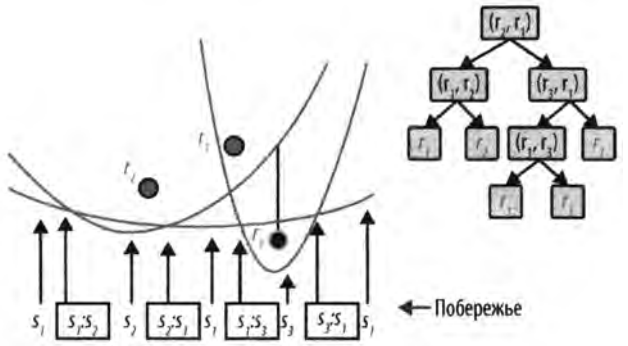


Рис. 9.14. Линия побережья после трех точек

Ключевым шагом в этом алгоритме является удаление узлов из состояния линии побережья, которые не должны оказывать влияния на построение диаграммы Вороного. После обработки события обнаруженной окружности средняя дуга, связанная в данном случае с r_1, никак не влияет на другие точки *P*, поэтому она может быть удалена из линии побережья. Результирующее состояние линии побережья показано в бинарном дереве в правой части рис. 9.15.

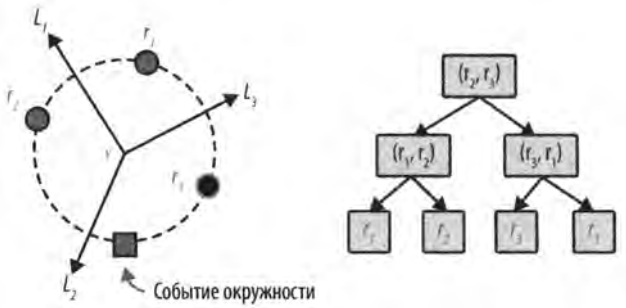


Рис. 9.15. Линия побережья после обработки события окружности

Входные и выходные данные алгоритма

Входными данными для алгоритма выметания Форчуна является множество P двумерных точек на плоскости.

Алгоритм выметания Форчуна вычисляет диаграмму Вороного, состоящую из n многоугольников Вороного, каждый из которых определяет область для одной из точек P. Математически могут иметься частично бесконечные области, но алгоритм устраняет их путем вычисления диаграммы Вороного в ограничивающем прямоугольнике подходящего размера, охватывающем все точки из P. Результатом работы алгоритма является множество отрезков и многоугольники Вороного, которые определяются ребрами вокруг каждой точки P, перечисленными в порядке движения по часовой стрелке.

Некоторые реализации требуют, чтобы множество P не содержало четырех точек, которые находились бы на одной окружности.

В некоторых других реализациях предполагается, что никакие две точки не имеют одну и ту же координату x или y. Такое требование устраняет многие особые случаи. Это требование легко реализовать, если входное множество (x,y) содержит целочисленные координаты: достаточно просто добавить случайное дробное число к каждой координате перед вызовом алгоритма выметания Форчуна.

Реализация алгоритма

Реализация алгоритма выметания Форчуна сложна из-за вычислений, необходимых для поддержки состояния линии побережья. Из представленной здесь реализации исключен ряд частных случаев. Репозиторий кода к данной книге содержит функции, которые выполняют геометрические вычисления для пересекающихся парабол. Классы, поддерживающие эту реализацию, кратко показаны на рис. 9.16. Данная реализация использует модуль `heapq` языка программирования Python, который обеспечивает методы `heappop` и `heappush`, используемые для создания очереди с приоритетами и работы с ней.

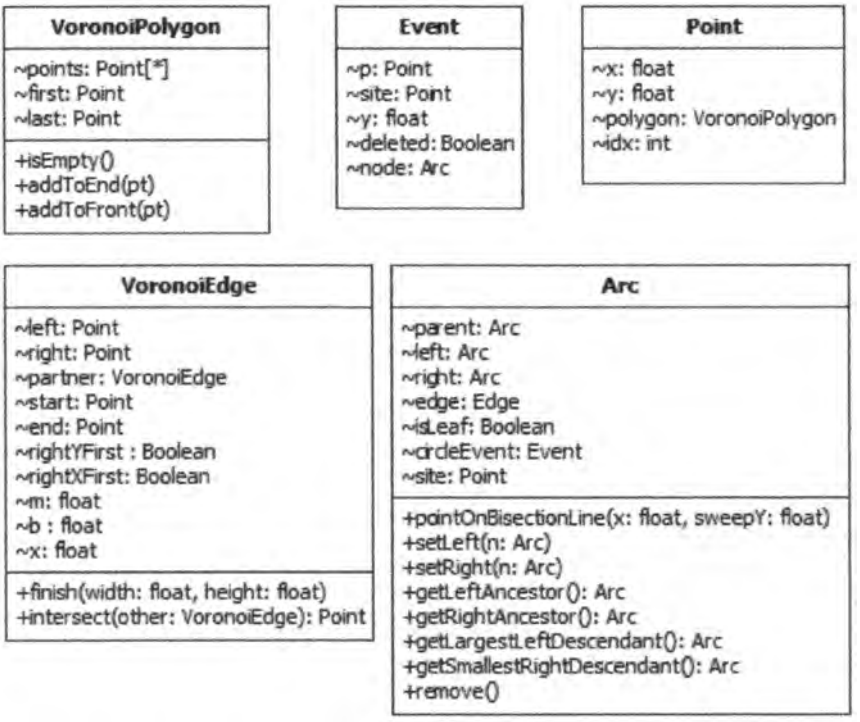


Рис. 9.16. Классы, используемые в коде

Как показано в примере 9.4, метод `process` создает события узлов для каждой из входных точек и обрабатывает события по одному, по убыванию координаты *y* (при одинаковых координатах *y* первой идет точка с меньшей координатой *x*).

Алгоритм выметания Форчуна

Наилучший, средний и наихудший случаи: $O(n \cdot \log n)$

```
fortune (P)
    PQ = new Priority Queue
    LineState = new Binary Tree

    foreach p in P do                                          ❶
        event = new SiteEvent(p)
        Вставка event в PQ

        while очередь PQ не пуста do
            event = getMin(PQ)
            sweepPt = event.p                                  ❷
            if event - событие узла then
                processSite(event)
            else
                processCircle(event)
```

```
        finishEdges()                                                    ❷

    processSite(e)                                                       ❹
        leaf = найти дугу A в побережье, делимую пополам e.p
        Изменить побережье и удалить ненужные события окружности        ❺
        Выявить новые потенциальные события окружности

    processCircle(e)                                                     ❻
        Определить соседние слева и справа дуги на побережье
        Удалить ненужные события окружности
        Записать вершину и ребра Вороного
        Изменить побережье, удалив "среднюю" дугу
        Выявить новые потенциальные события окружности слева и справа

❶ Очередь упорядочивает события по убыванию координаты y.
  Выметающая линия обновляется точкой, связанной с каждым
    удаленным событием.
❸ Оставшиеся точки останова в линии побережья определяют
    окончательные ребра.
❹ Обновление линии побережья с каждой новой точкой ...
❺ ... которая может удалить потенциальные события окружности.
❻ Вычисление точки Вороного и обновление состояния линии побережья.
```

Пример 9.4. Реализация алгоритма выметания Форчуна на языке программирования Python

```
from heapq import heappop, heappush

class Voronoi:
    def process (self, points):
        self.pq = []
        self.edges = []
        self.tree = None
        self.firstPoint = None
        self.stillOnFirstRow = True
        self.points = []

    # Каждая точка имеет уникальный идентификатор
    for idx in range(len(points)):
        pt = Point (points[idx], idx)
        self.points.append (pt)
        event = Event (pt, site=pt)
        heappush (self.pq, event)

    while self.pq:
        event = heappop (self.pq)
```

```
        if event.deleted:
            continue

    self.sweepPt = event.p
    if event.site:
        self.processSite (event)
    else:
        self.processCircle (event)

    # Завершение ребер, которые растягиваются до бесконечности
    if self.tree and not self.tree.isLeaf:
        self.finishEdges (self.tree)

        # Завершение ребер Вороного
        for e in self.edges:
            if e.partner:
                if e.b is None:
                    e.start.y = self.height
                else:
                    e.start = e.partner.end
```

Эта реализация обрабатывает особый случай, когда есть несколько точек, имеющих одно и то же наибольшее значение координаты *y*; она делает это путем сохранения `firstPoint`, обнаруженной в `processSite`.

Детали алгоритма выметания Форчуна содержатся в реализации `processSite`, показанной в примере 9.5, и реализации `processCircle`, показанной в примере 9.6.

Пример 9.5. Обработка событий узлов

```
def processSite (self, event):
    if self.tree == None:
        self.tree = Arc(event.p)
        self.firstPoint = event.p
        return

    # Должна обрабатывать особый случай, когда две точки имеют
    # одну и ту же наивысшую координату y. В таком случае корень
    # представляет собой лист. Обратите внимание, что при сортировке
    # событий эти точки упорядочиваются по координате x, так что
    # следующая точка будет находиться справа.
    if self.tree.isLeaf and event.y == self.tree.site.y:
        left = self.tree
        right = Arc(event.p)

        start = Point(((self.firstPoint.x + event.p.x)/2, self.height))
        edge = VoronoiEdge (start, self.firstPoint, event.p)

        self.tree = Arc(edge = edge)
```

```
        self.tree.setLeft (left)
        self.tree.setRight (right)

        self.edges.append (edge)
        return

    # Если лист представляет собой событие окружности, он больше
    # не является корректным, поскольку разделяется
    leaf = self.findArc (event.p.x)
    if leaf.circleEvent:
        leaf.circleEvent.deleted = True

    # Находим точку параболы, где event.pt.x разбивает ее пополам
    # вертикальной линией
    start = leaf.pointOnBisectionLine (event.p.x, self.sweepPt.y)

    # Между двумя узлами находятся потенциальные ребра Вороного
    negRay = VoronoiEdge (start, leaf.site, event.p)
    posRay = VoronoiEdge (start, event.p, leaf.site)
    negRay.partner = posRay
    self.edges.append (negRay)

    # Изменение побережья новыми внутренними узлами
    leaf.edge = posRay
    leaf.isLeaf = False

    left = Arc()
    left.edge = negRay
    left.setLeft (Arc(leaf.site))
    left.setRight (Arc(event.p))

    leaf.setLeft (left)
    leaf.setRight (Arc(leaf.site))

    # Проверка наличия потенциального события
    # окружности слева или справа.
    self.generateCircleEvent (left.left)
    self.generateCircleEvent (leaf.right)
```

Метод `processSite` изменяет линию побережья с каждым обнаруженным событием узла вставкой двух дополнительных внутренних узлов и двух дополнительных листьев. Метод `findArc` за время $O(log\ n)$ находит дуги, которые должны быть изменены вновь обнаруженным событием узла. При изменении линии побережья алгоритм вычисляет два ребра, которые в конечном итоге будут находиться в окончательной диаграмме Вороного. Они присоединяются к узлам точки останова `Arc` в дереве. Всякий раз, когда выполняются изменения состояния линии побережья, алгоритм выполняет проверку дуг слева и справа, чтобы выяснить, не образуют ли соседние дуги потенциальное событие окружности.

Пример 9.6. Обработка события окружности

```
def processCircle (self, event):
    node = event.node

    # Поиск соседей слева и справа.
    leftA = node.getLeftAncestor()
    left = leftA.getLargestDescendant()
    rightA = node.getRightAncestor()
    right = rightA.getSmallestDescendant()

    # Удаление старых событий окружности при наличии таковых.
    if left.circleEvent:
        left.circleEvent.deleted = True
    if right.circleEvent:
        right.circleEvent.deleted = True

    # Окружность определена точками left - node - right.
    # Завершение лучей Вороного.
    p = node.pointOnBisectionLine (event.p.x, self.sweepPt.y)
    leftA.edge.end = p
    rightA.edge.end = p

    # Обновление узла-предка в линии побережья записью
    # новых потенциальных ребер.
    t = node
    ancestor = None
    while t != self.tree:
        t = t.parent
        if t == leftA:
            ancestor = leftA
        elif t == rightA:
            ancestor = rightA

    ancestor.edge = VoronoiEdge(p, left.site, right.site)
    self.edges.append (ancestor.edge)

    # Удаление средней дуги (лист) из дерева линии побережья.
    node.remove()

    # После удаления могут найтись новые соседи,
    # поэтому необходима проверка на окружности.
    self.generateCircleEvent (left)
    self.generateCircleEvent (right)
```

Метод `processCircle` отвечает за выявление новых вершин в диаграмме Вороного. Каждое событие окружности связано с `node`, крайней верхней точкой окружности, первой сгенерировавшей событие окружности. Этот метод удаляет узел `node` из

состояния линии побережья, так как он не может не влиять на будущие вычисления. При этом возможны новые соседи на линии побережья, поэтому метод проверяет соседей слева и справа, чтобы увидеть, не должны ли создаваться дополнительные события окружностей.

Эти примеры кода зависят от вспомогательных методов, которые выполняют геометрические вычисления, включая методы `pointOnBisectionLine` и `intersect`. Детальную информацию можно найти в репозитории кода к книге. Основные трудности реализации алгоритма выметания Форчуна лежат в корректной реализации всех необходимых геометрических вычислений. Один из способов сведения к минимуму количества особых случаев заключается в принятии требования, что все значения координат во входных данных (и x, и y) уникальны и что никакие четыре точки не лежат на одной окружности. Выполнение этих условий упрощает работу алгоритма, в основном потому, что вы можете игнорировать случаи, когда диаграмма Вороного содержит горизонтальные или вертикальные линии.

В последнем примере кода, `generateCircleEvent`, показанном в примере 9.7, определяется, когда три соседние дуги на линии побережья образуют окружность. Если крайняя снизу точка на этой окружности находится выше выметающей линии (т.е. уже должна была быть обработана), то окружность игнорируется; в противном случае в очередь событий добавляется событие окружности, которое будет обработано в свой срок. Оно также может оказаться удаленным, если другой обрабатываемый узел окажется внутри этого круга.

Пример 9.7. Генерация нового события окружности

```
def generateCircleEvent (self, node):
    """
    Существует возможность события окружности, если этот новый узел
    является средним из трех последовательных узлов. Если это так,
    то новое событие окружности добавляется в очередь с приоритетами
    для дальнейшей обработки.
    """

    # Поиск соседей слева и справа, если таковые существуют.
    leftA = node.getLeftAncestor()
    if leftA is None:
        return
    left = leftA.getLargestLeftDescendant()

    rightA = node.getRightAncestor()
    if rightA is None:
        return
    right = rightA.getSmallestRightDescendant()

    # Проверка, что они различны
```

```
if left.site == right.site:
    return

# Если два ребра не имеют корректного пересечения, выходим.
p = leftA.edge.intersect (rightA.edge)
if p is None:
    return

radius = ((p.x-left.site.x)**2 + (p.y-left.site.y)**2)**0.5

# Выбор точки в нижней части описанной окружности.
circleEvent = Event(Point((p.x, p.y-radius)))
if circleEvent.p.y >= self.sweepPt.y:
    return

node.circleEvent = circleEvent
circleEvent.node = node
heappush (self.pq, circleEvent)
```

Анализ алгоритма

Производительность алгоритма выметания Форчуна определяется количеством событий, вставляемых в очередь с приоритетами. В начале работы алгоритма должны быть вставлены n точек. Во время обработки каждый новый узел может генерировать максимум две дополнительные дуги; таким образом, линия побережья имеет не более $2n-1$ дуг. С помощью бинарного дерева для хранения состояния линии побережья мы можем найти узлы требуемой дуги за время $O(\log n)$.

Модификация листа в `processSite` требует фиксированного количества операций, поэтому ее можно рассматривать как выполняющуюся за константное время. Аналогично удаление узла дуги в методе `processCircle` также является операцией с константным временем выполнения. Обновление узла-предка в линии побережья для записи новых потенциальных ребер остается операцией со временем выполнения $O(\log n)$. Сбалансированность бинарного дерева, содержащего состояние линии, не гарантируется, но добавление этой возможности повышает производительность вставки и удаления до $O(\log n)$. Кроме того, после балансировки бинарного дерева узлы, бывшие ранее листьями, остаются таковыми.

Таким образом, обрабатывает ли алгоритм событие узла или событие окружности, его производительность ограничена значением $2n \cdot \log n$, что дает общую производительность $O(n \cdot \log n)$.

Этот сложный алгоритм нелегко раскрывает свои секреты. Даже ученые в области разработки алгоритмов признают, что это одно из наиболее сложных применений метода выметания. Лучше всего наблюдать за поведением этого алгоритма с помощью его пошагового выполнения в отладчике.

10

Пространственные древовидные структуры

Алгоритмы в этой главе относятся главным образом к моделированию двумерных структур на декартовой плоскости и предназначены для выполнения мощных поисковых запросов, которые выходят за рамки простого членства, как описано в главе 5, "Поиск". Эти алгоритмы включают следующие.

Поиск ближайшего соседа

Для данного множества двумерных точек P определить, какая точка находится ближе всех к целевой точке запроса x. Эта задача может быть решена за время $O(log\ n)$ вместо решения методом грубой силы за время $O(n)$.

Запросы диапазонов

Для данного множества двумерных точек P определить, какие точки содержатся внутри заданной прямоугольной области. Эта задача может быть решена за время $O(\sqrt{n}+r)$ вместо решения методом грубой силы за время $O(n)$.

Запросы о пересечении

Для данного множества двумерных прямоугольников R определить, какие прямоугольники пересекаются с целевой прямоугольной областью. Эта задача может быть решена за время $O(log\ n)$ вместо решения методом грубой силы за время $O(n)$.

Обнаружение столкновений

Для данного множества двумерных точек P определить пересечения квадратов со стороной s и центрами в данных точках. Эта задача может быть решена за время $O(n \cdot log\ n)$ вместо решения методом грубой силы за время $O(n^2)$.

Структуры и алгоритмы из этой главы естественным образом распространяются на несколько измерений, но для удобства в этой главе мы ограничимся двумерными структурами.

Запросы ближайшего соседа

Для данного множества точек P на двумерной плоскости может потребоваться определить, какая точка из P является *ближайшей* к целевой точке запроса x с использованием евклидова расстояния. Заметим, что точка x не обязана находиться в P, что отличает данную задачу от задач, решаемых с помощью алгоритмов из главы 5, "Поиск". Эти запросы расширяются для входных множеств, точки которых находятся в n-мерном пространстве.

Прямая реализация заключается в проверке всех точек P, в результате чего алгоритм обладает линейной эффективностью $O(n)$. Поскольку множество P известно заранее, возможно, есть способ структурировать информацию для ускорения запросов путем отбрасывания больших групп точек во время поиска. Возможно, мы могли бы разделить плоскость на ячейки некоторого фиксированного размера $m \times m$, как показано на рис. 10.1. Здесь 10 входных точек множества P (показаны кружками) помещаются в девять охватывающих ячеек. Большое число в каждой ячейке указывает количество точек в нем. При поиске ближайшего соседа для точки x (показана как маленький черный квадрат) следует найти включающую его ячейку. Если эта ячейка не является пустой, нам нужно выполнить поиск только в ячейках, которые пересекаются окружностью, радиус которой равен $\sqrt{2}m$.

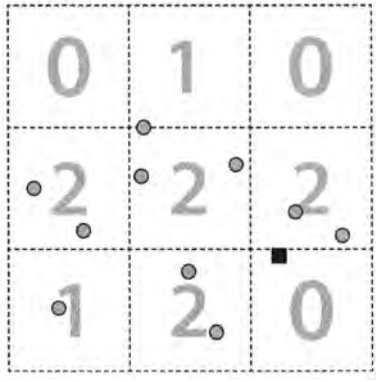


Рис. 10.1. Поиск ближайшего соседа с использованием областей

Однако в этом примере в целевой ячейке нет точек, так что необходимо рассмотреть три соседние ячейки. Этот неэффективный подход является непригодным, поскольку многие ячейки в действительности могут быть пустыми, так что алгоритму по-прежнему придется выполнять поиск в нескольких соседних ячейках. Как мы

видели в главе 5, "Поиск", бинарные деревья поиска могут сократить прилагаемые усилия, убирая из рассмотрения группы точек, которые заведомо не могут быть частью решения. В этой главе мы представим идею *пространственного дерева* (spatial tree) для разделения множества точек в двумерной плоскости, сокращающего время поиска. Дополнительная стоимость предварительной обработки для превращения всех точек множества P в эффективную структуру позже вернется в виде экономии времени на вычисления при выполнении запросов, которые теперь требуют времени $O(\log n)$. Если количество запросов мало, то лучше использовать сравнение, основанное на грубой силе и имеющее время работы $O(n)$.

Запросы диапазонов

Вместо поиска конкретной целевой точки могут быть запрошены все точки P, которые содержатся в заданной прямоугольной области двумерной плоскости. Решение, основанное на грубой силе, проверяет, содержится ли каждая точка в пределах целевой прямоугольной области и имеет производительность $O(n)$.

Та же структура данных, что и разработанная для поиска ближайшего соседа, поддерживает и эти запросы, известные как "ортогональный диапазон" (поскольку прямоугольная область запроса ориентирована по осям x и y). Единственный способ добиться производительности, лучшей, чем $O(n)$, — найти способ отказаться от рассмотрения точек и их включения в результат запроса. При использовании k-d-дерева запрос выполняется с помощью рекурсивного обхода и его производительность может достигать $O(\sqrt{n}+r)$, где r — количество точек, найденных запросом.

Запросы пересечения

Входные данные для запроса могут представлять собой более сложные структуры, чем просто множество n-мерных точек. Рассмотрим вместо него набор прямоугольников R в двумерной плоскости, в котором каждый прямоугольник r_i определяется кортежем $(x_{low}, y_{low}, x_{high}, y_{high})$. Для данного множества R может потребоваться найти все прямоугольники, которые включают заданную точку (x,y) или (в более общем случае) которые пересекаются с целевым прямоугольником (x_1, y_1, x_2, y_2). Структурирование прямоугольников представляется более сложным, поскольку прямоугольники могут перекрываться.

Вместо пересечения с целевым прямоугольником нас могут интересовать пересечения коллекции двумерных элементов. Эта задача известна как задача *обнаружения столкновений*, и далее мы представим ее решение для обнаружения пересечений среди коллекции точек P, окруженных квадратными областями.

Структуры пространственных деревьев

Структуры пространственных деревьев показывают, как представить данные для эффективной поддержки трех описанных распространенных поисков. В данной главе мы представим ряд структур пространственных деревьев, которые используются для разделения *n*-мерных объектов для повышения производительности операций поиска, вставки и удаления. Далее будут представлены три такие структуры.

k-d-деревья

На рис. 10.2, *а* те же 10 точек, что и на рис. 10.1, показаны в *k-d*-дереве, которое названо так потому, что оно может разделить *k*-мерную плоскость вдоль перпендикулярных осей системы координат. Эти точки нумеруются в том порядке, в котором они были вставлены в дерево. Структура *k-d*-дерева на рис. 10.2, *а* изображена в виде бинарного дерева на рис. 10.2, *б*. В оставшейся части нашего обсуждения мы рассматриваем двумерное дерево, однако тот же подход может использоваться для произвольной размерности.

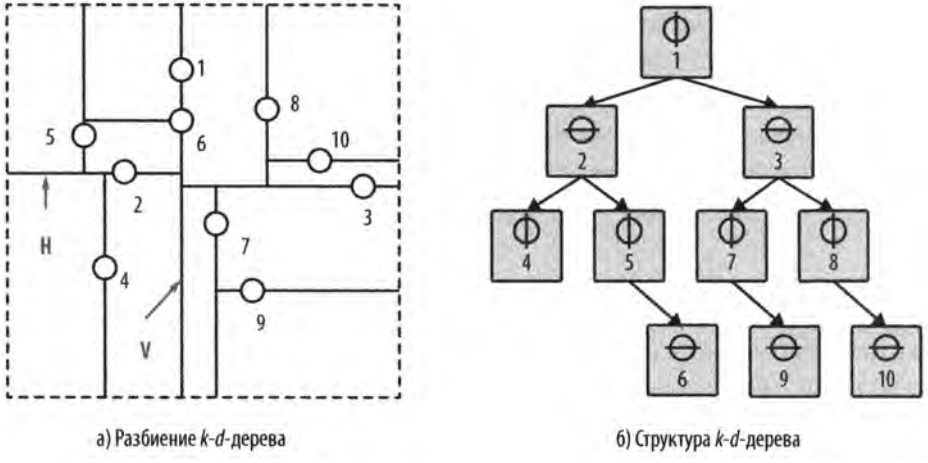


Рис. 10.2. Деление двумерной плоскости с использованием k-d-дерева

k-d-дерево представляет собой рекурсивную структуру бинарного дерева, в которой каждый узел содержит точку и метку координаты (например, *x* или *y*), которая определяет ориентацию разбиения. Корневой узел представляет собой прямоугольную область ($x_{low}=-\infty$, $y_{low}=-\infty$, $x_{high}=+\infty$, $y_{high}=+\infty$) на плоскости, разделенную вдоль вертикальной линии *V* через точку p_1. Левое поддерево разделяет область слева от *V*, в то время как правое поддерево — справа от *V*. Левый дочерний узел корня представляет разделение вдоль горизонтальной линии *H* через точку p_2, которое делит область слева от *V* на область выше линии *H* и область ниже этой линии. Область ($-\infty$, $-\infty$, $p_1.x$, $+\infty$) связана с левым дочерним узлом корня, в то время как область

($p_1.x$, $-\infty$, $+\infty$, $+\infty$) связана с правым дочерним узлом корня. Эти области, по сути, являются вложенными, и можно видеть, что область узла-предка полностью содержит области любого из его узлов-потомков.

Дерево квадрантов

Дерево квадрантов, или *quadtree*, разбивает множество двумерных точек *P*, рекурсивно подразделяя все пространство на четыре квадранта. Это древовидная структура, в которой каждый внутренний узел имеет четыре дочерних узла, помеченных как **NE** (северо-восток), **NW** (северо-запад), **SW** (юго-запад) и **SE** (юго-восток). Существуют две различные разновидности деревьев квадрантов.

Основанные на областях

Для данного изображения размером $2^k \times 2^k$ пикселей, на котором каждый пиксель равен 0 или 1, корень дерева квадрантов представляет все изображение. Каждый из четырех дочерних узлов корня представляет квадрант исходного изображения размером $2^{k-1} \times 2^{k-1}$. Если одна из этих четырех областей не состоит полностью из нулей или единиц, то эта область подразделяется на подобласти, каждая из которых имеет размер, вчетверо меньший, чем размер его родительской области. Листовые узлы представляют квадратные области, состоящие только из нулевых или только из единичных пикселей. На рис. 10.3 показан пример такой древовидной структуры для образца растрового изображения.

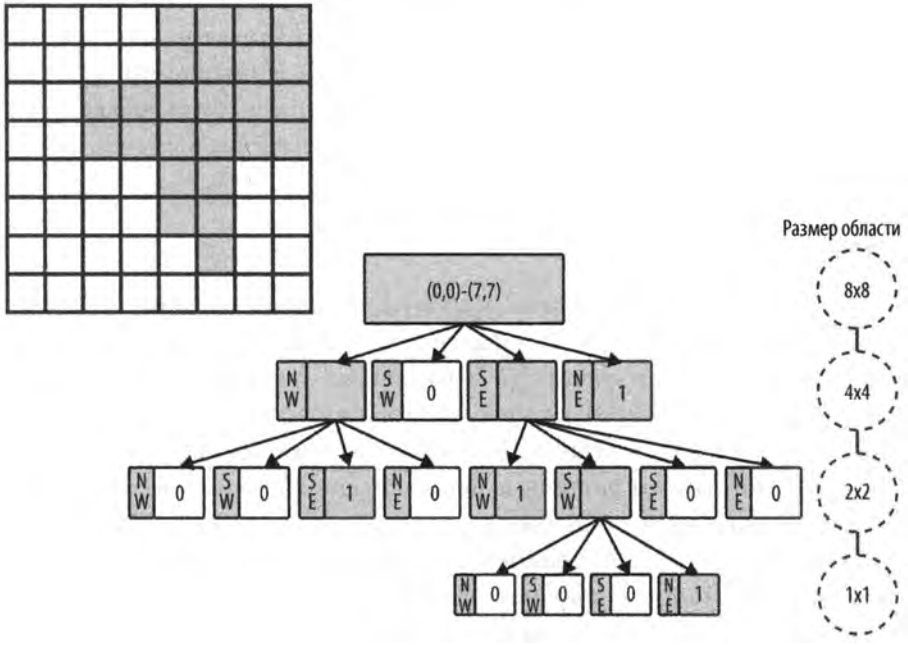


Рис. 10.3. Дерево квадрантов с разбиением на основе областей

Основанные на точках

Для заданного пространства размером $2^k \times 2^k$ в декартовой системе координат дерево квадрантов отображается непосредственно на структуру бинарного дерева, в которой каждый узел может хранить до четырех точек. Если точка добавляется в полную область, то эта область подразделяется на четыре подобласти, каждая из которых вчетверо меньше родительской. Если в квадранте, соответствующем узлу, нет никаких точек, то у этого узла нет дочерних узлов, так что форма дерева зависит от порядка, в котором в него добавляются точки.

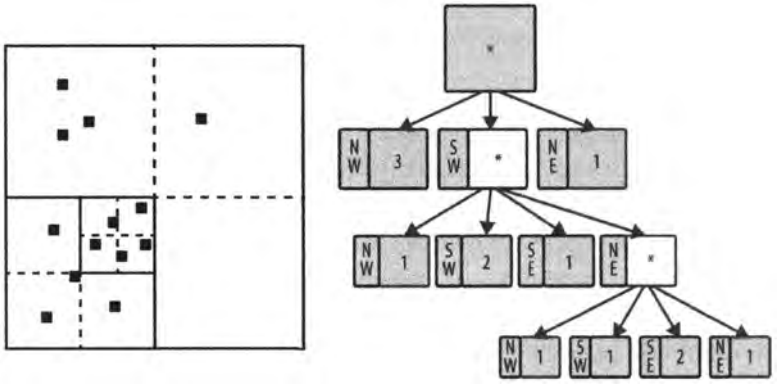


Рис. 10.4. Дерево квадрантов с разбиением на основе точек

В этой главе мы сосредоточимся на деревьях квадрантов на основе точек. На рис. 10.4 показан пример области размером 256×256 с 13 точками. Структура дерева квадрантов показана в правой части рисунка. Обратите внимание, что в юго-восточном квадранте исходной области точек нет, поэтому корень имеет только три дочерних узла. Обратите также внимание на переменное разделение областей, основанное на том, какие точки были добавлены в дерево.

R-дерево

R-дерево (R-tree) — древовидная структура, в которой каждый узел содержит до M ссылок на дочерние узлы. Вся фактическая информация хранится в листах, каждый из которых может хранить до M различных прямоугольников. На рис. 10.5 изображено R-дерево с $M=4$, в которое вставлены шесть прямоугольников (с метками 1, 2, 3, 4, 5 и 6). Результатом является показанное справа дерево, в котором внутренние узлы отражают различные прямоугольные области, внутри которых находятся фактические прямоугольники.

Корневой узел представляет наименьшую прямоугольную область (x_{low}, y_{low}, x_{high}, y_{high}), включающую все прочие прямоугольники в дереве. Каждый внутренний узел имеет связанную с ним прямоугольную область, которая аналогичным образом

включает в себя все прямоугольники в его узлах-потомках. Фактические прямоугольники в R-дереве хранятся только в листьях. Каждая прямоугольная область (независимо от того, с чем она связана — с внутренним узлом или с листом) полностью содержится не только в корневом узле, но и в области родительского узла.

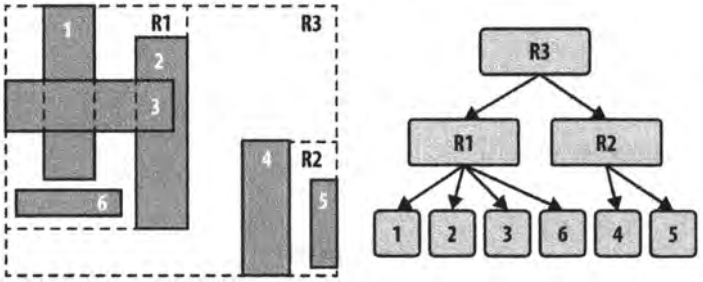


Рис. 10.5. Пример R-дерева

Теперь мы используем эти различные пространственные структуры для решения задач, перечисленных в начале данной главы.

Задача о ближайшем соседе

Для заданного множества двумерных точек P нужно определить, какая точка из P находится ближе всех к целевой точке запроса x. Мы покажем, как можно использовать k-d-деревья для эффективного выполнения таких запросов. В предположении, что дерево эффективно разделяет точки, при каждом рекурсивном обходе дерева будет отбрасываться примерно половина точек в нем. В k-d-дереве для точек с нормальным распределением узлы на уровне i отражают прямоугольники, примерно вдвое большие, чем прямоугольники на уровне $i+1$. Это свойство позволяет решить задачу о ближайшем соседе с производительностью $O(log\ n)$, поскольку оно позволяет отбрасывать целые поддеревья, содержащие точки, которые явно находятся слишком далеко, чтобы быть ближайшими соседями к заданной точке. Однако, как мы покажем, рекурсия в этом случае оказывается немного сложнее, чем для обычных бинарных деревьев поиска.

Как показано на рис. 10.6, если целевая точка (показанная как маленький черный квадрат) будет вставлена, то она станет дочерним узлом для узла, связанного с точкой 9. После вызова `nearest` для этого выбора алгоритм может отбросить все левое поддерево, потому что перпендикулярное расстояние `dp` не меньше, а значит, ни одна из этих точек не ближе. Рекурсия в правом поддереве обнаруживает, что точка 3 находится ближе точки 9. Кроме того, алгоритм определяет, что перпендикулярное расстояние до точки 3 меньше, чем `min`, так что он должен рекурсивно исследовать оба поддерева — для точки 7 и для точки 8. В конечном итоге алгоритм определяет, что точка 3 является ближайшей к целевой точкой.

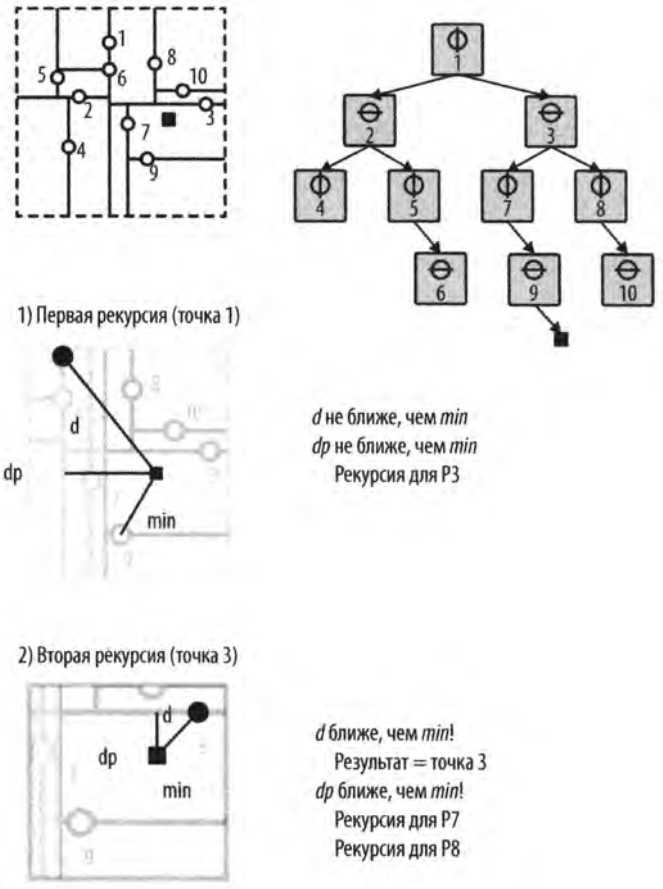


Рис. 10.6. Пример работы алгоритма определения ближайшего соседа

Алгоритм определения ближайшего соседа

Наилучший и средний случаи: $O(\log n)$, наихудший случай: $O(n)$

```
nearest (T, x)
    n = родительский узел в T, куда была бы вставлена точка x
    min = расстояние от x до n.point                    ❶
    better = nearest (T.root, min, x)                   ❷
    if найдено better then return better
    return n.point
end

nearest (node, min, x)
    d = расстояние от x до node.point
    if d < min then
```

```
        result = node.point                                    ❸
        min = d
    dp = перпендикулярное расстояние от x до node
    if dp < min then                                           ❹
        pt = nearest (node.above, min, x)
        if расстояние от pt до x < min then
            result = pt
            min = расстояние от pt до x
        pt = nearest (node.below, min, x)
        if расстояние от pt до x < min then
            result = pt
            min = расстояние от pt до x
    else                                                       ❺
        if node выше x then
            pt = nearest (node.above, min, x)
        else
            pt = nearest (node.below, min, x)
        if существует pt then return pt
    return result
end

❶ Разумное предположение о ближайшей точке.
❷ Обход от корня в попытках найти лучшее решение.
❸ Найдена более близкая точка.
❹ Если достаточно близко, проверяем верхнее и нижнее поддеревья.
❺ В противном случае можно безопасно ограничиться одним поддеревом.
```

Входные и выходные данные алгоритма

Входными данными является *k-d*-дерево, сформированное из множества двумерных точек P на плоскости. Множество (заранее неизвестное) запросов о ближайшем соседе из множества P для точки x обрабатывается по одному запросу.

Для каждой запрашиваемой точки x алгоритм вычисляет точку из P, которая является ближайшим соседом x.

Если две точки оказываются "слишком близкими" одна к другой из-за ошибок вычислений с плавающей точкой, алгоритм может выбрать неправильную точку; однако расстояние до фактической ближайшей точки будет настолько близким, что полученный ответ можно не рассматривать как неверный.

Контекст применения алгоритма

При сравнении этого подхода с подходом грубой силы, который сравнивает расстояния между точкой запроса x и каждой точкой $p \in P$, следует рассмотреть две

важные стоимости: 1) стоимость построения *k-d*-дерева и 2) стоимость размещения точки запроса *x* в структуре дерева. На эту стоимость влияют следующие параметры.

Количество измерений

По мере увеличения количества измерений стоимость построения *k-d*-дерева превышает получаемую от него выгоду. В некоторых авторитетных источниках полагается, что для более чем 20 измерений подход с использованием дерева менее эффективен, чем непосредственное сравнение со всеми точками.

Количество точек во входном множестве

Если количество точек мало, стоимость построения структуры дерева может превысить рост производительности.

Бинарные деревья могут быть эффективными поисковыми структурами, поскольку они могут быть сбалансированы при вставке узлов в дерево и удалении их из него. К сожалению, *k-d*-деревья не могут быть легко сбалансированы, из них не удается выполнять удаление из-за глубокой структурной информации о размерной плоскости, которую они представляют. Идеальным решением является построение начального *k-d*-дерева таким образом, чтобы либо 1) все листья находились на одном и том же уровне в дереве, либо 2) все листья находились в пределах одного уровня от прочих листьев.

Реализация алгоритма

Реализация алгоритма поиска ближайшего соседа для существующего *k-d*-дерева показана в примере 10.1.

Пример 10.1. Реализация алгоритма поиска ближайшего соседа в k-d-дереве

```
// Метод KDTree.
public IMultiPoint nearest(IMultiPoint target)
{
    if (root == null || target == null) return null;

    // Поиск родительского узла, в который была бы
    // вставлена целевая точка.
    // Поиск расстояния до соответствующей точки.
    DimensionalNode parent = parent(target);
    IMultiPoint result = parent.point;
    double smallest = target.distance(result);

    // Начинаем с корня в поисках более близкой точки.
    double best[] = new double[] { smallest };
    double raw[] = target.raw();
    IMultiPoint betterOne = root.nearest(raw, best);
```

```
    if (betterOne != null)
    {
        return betterOne;
    }

    return result;
}

// Метод DimensionalNode. min[0] — наилучшее вычисленное
// кратчайшее расстояние.
IMultiPoint nearest(double[] rawTarget, double min[])
{
    // Обновление минимума, если мы находимся ближе.
    IMultiPoint result = null;
    // Обновление значения минимума
    double d = shorter(rawTarget, min[0]);

    if (d >= 0 && d < min[0])
    {
        min[0] = d;
        result = point;
    }

    // Определяем, должны ли мы погрузиться в поддеревья путем
    // вычисления прямого перпендикулярного расстояния до оси
    // вдоль узла, разделяющего плоскость. Если d меньше, чем
    // текущее наименьшее расстояние, следует проверить обе части.
    double dp = Math.abs(coord - rawTarget[dimension - 1]);
    IMultiPoint newResult = null;

    if (dp < min[0])
    {
        // Должны пройти обе части. Возвращаем кратчайшую.
        if (above != null)
        {
            newResult = above.nearest(rawTarget, min);

            if (newResult != null)
            {
                result = newResult;
            }
        }

        if (below != null)
        {
            newResult = below.nearest(rawTarget, min);

            if (newResult != null)
```

```
            {
                result = newResult;
            }
        }
    }
    else
    {
        // Достаточно пройти одну часть! Определяем, какую.
        if (rawTarget[dimension - 1] < coord)
        {
            if (below != null)
            {
                newResult = below.nearest(rawTarget, min);
            }
        }
        else
        {
            if (above != null)
            {
                newResult = above.nearest(rawTarget, min);
            }
        }

        // Используем меньший результат, если таковой найден.
        if (newResult != null)
        {
            return newResult;
        }
    }

    return result;
}
```

Ключевым моментом для понимания алгоритма поиска ближайшего соседа является то, что мы сначала находим область, в которую была бы вставлена целевая точка, поскольку вполне вероятно, что именно эта область будет содержать ближайшую к целевой точку. Затем мы проверяем правильность этого предположения, рекурсивно проходя от корня до данной области в поисках другой, более близкой точки. Это вполне возможно, потому что прямоугольные области *k-d*-дерева созданы на основе входного множества точек. В несбалансированном *k-d*-дереве этот процесс проверки может привести к общей стоимости $O(n)$, что приводит нас к вопросу о корректной обработке входного множества.

В приведенном выше примере имеется два усовершенствования для ускорения работы. Во-первых, сравнения производятся с использованием "сырого" массива `double`, представляющего каждую точку. Во-вторых, для определения, является ли расстояние между двумя *d*-мерными точками меньшим, чем вычисленное к этому

моменту минимальное расстояние, используется метод `shorter` из `DimensionalNode`. Этот метод завершает работу сразу же после частичного вычисления евклидова расстояния, превышающего минимальное найденное.

В предположении изначальной сбалансированности *k-d*-дерева во время рекурсивных вызовов поиск может отказаться от проверки до половины точек дерева. Иногда требуются два рекурсивных вызова, но только когда вычисленное минимальное расстояние достаточно велико, чтобы пересечь разделяющую линию узла; в этом случае для поиска ближайшей точки необходимо исследовать обе стороны.

Анализ алгоритма

k-d-дерево изначально строится как сбалансированное, в котором разделительная линия на каждом уровне проходит через медиану точек, оставшихся на этом уровне. Обнаружение родительского узла для целевого запроса можно найти за время $O(log\ n)$ путем обхода *k-d*-дерева, как если бы требовалось вставить данную точку в дерево. Однако алгоритму может потребоваться выполнить два рекурсивных вызова: один — для верхнего дочернего узла выше и один — для нижнего.

При частом выполнении двойной рекурсии производительность алгоритма деградирует до $O(n)$, поэтому желательно понять, как часто это может происходить. Несколько вызовов выполняются только тогда, когда перпендикулярное расстояние dp от целевой точки до точки узла меньше, чем лучшее вычисленное минимальное расстояние. По мере увеличения количества измерений становится все больше потенциальных точек, удовлетворяющих этим критериям.

В табл. 10.1 представлены некоторые эмпирические доказательства, которые должны продемонстрировать, как часто это происходит. Сбалансированное *k-d*-дерево создается из n случайных двумерных точек (от 4 до 131 072), распределенных в пределах единичного квадрата. Для случайных точек в пределах этого квадрата выполняется 50 запросов, и в табл. 10.1 записывается среднее количество выполнения двух рекурсивных вызовов (т.е. случаев, когда $dp < min[0]$, и данный узел имеет дочерние узлы как выше, так и ниже) по сравнению с количеством одинарных рекурсивных вызовов.

Таблица 10.1. Отношение количества двойных рекурсий к одинарным

n	*d* = 2 Одинарных рекурсий	*d* = 2 Двойных рекурсий	*d* = 10 Одинарных рекурсий	*d* = 10 Двойных рекурсий
4	1,96	0,52	1,02	0,98
8	3,16	1,16	1,08	2,96
16	4,38	1,78	1,20	6,98
32	5,84	2,34	1,62	14,96
64	7,58	2,38	5,74	29,02
128	9,86	2,98	9,32	57,84

Окончание табл. 10.1

n	*d* = 2 Одинарных рекурсий	*d* = 2 Двойных рекурсий	*d* = 10 Одинарных рекурсий	*d* = 10 Двойных рекурсий
256	10,14	2,66	23,04	114,80
512	12,28	2,36	53,82	221,22
1024	14,76	3,42	123,18	403,86
2048	16,90	4,02	293,04	771,84
4096	15,72	2,28	527,80	1214,10
8192	16,40	2,60	1010,86	2017,28
16384	18,02	2,92	1743,34	3421,32
32768	20,04	3,32	2858,84	4659,74
65536	21,62	3,64	3378,14	5757,46
131072	22,56	2,88	5875,54	8342,68

На основе этих случайных данных число двойных рекурсий представляется равным $0{,}3 \cdot \log n$ для двух измерений, и примерно $342 \cdot \log n$ для 10 измерений (1000-кратное увеличение). Важное наблюдение заключается в том, что обе эти оценочные функции представляют собой $O(\log n)$.

Но что происходит, когда d увеличивается и становится в некотором смысле "достаточно близким" к n? Данные, графически изображенные на рис. 10.7, показывают, что с увеличением d количество двойных рекурсий фактически приближается к $n/2$. Фактически по мере увеличения d число одинарных рекурсий соответствует нормальному распределению, среднее которого находится очень близко к $\log n$; это говорит о том, что в конечном итоге все рекурсивные вызовы становятся двойными. Это открытие влияет на производительность запросов по поиску ближайшего соседа таким образом, что по мере приближения d к $\log n$ преимущества использования k-d-деревьев начинают уменьшаться, пока итоговая производительность не оказывается не лучше $O(n)$ из-за выхода количества двойных рекурсий на "плато" с уровнем $n/2$.

Мы также можем оценить производительность k-d-дерева для поиска ближайшего соседа с алгоритмом грубой силы с производительностью $O(n)$. Насколько большой должна быть размерность d точек входного множества, чтобы для размера входного множества $n = 131\,072$ точек и 128 случайных поисков алгоритм грубой силы превзошел реализацию с использованием k-d-дерева?

Мы выполнили 100 испытаний и отбросили лучший и худший результаты, вычисляя среднее из оставшихся 98 испытаний. Результаты эксперимента показаны на рис. 10.9. Они демонстрируют, что для $d = 11$ и выше поиск ближайшего соседа грубой силой превосходит алгоритм k-d-дерева. Конкретные точки зависят от компьютера, на котором выполняется код, от значений n и d и распределения точек во входном множестве. Мы не включили в этот анализ стоимость построения k-d-дерева, потому что эта стоимость может быть амортизирована по всем поискам.

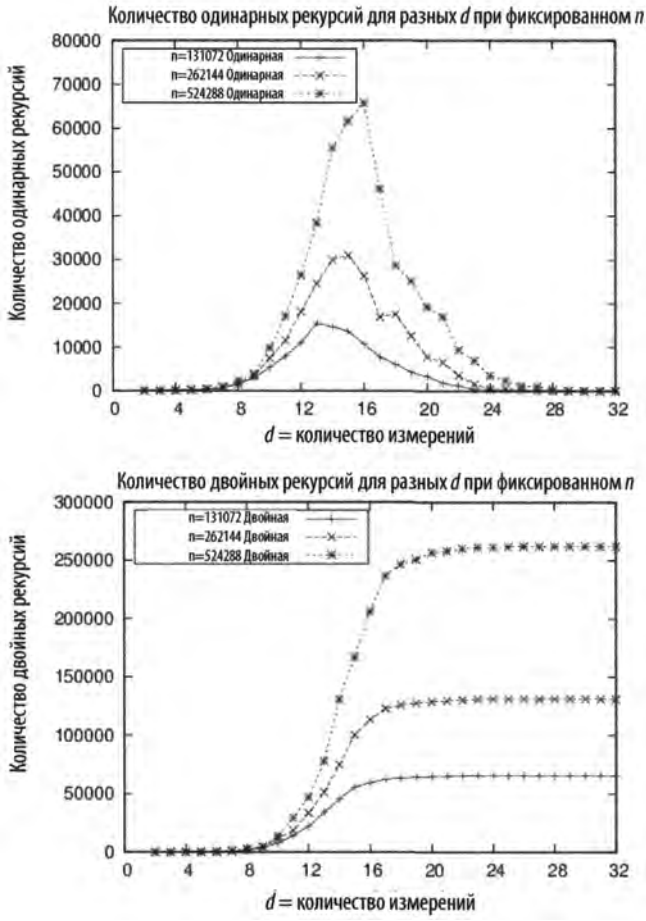


Рис. 10.7. Количество двойных рекурсий по мере роста n и d

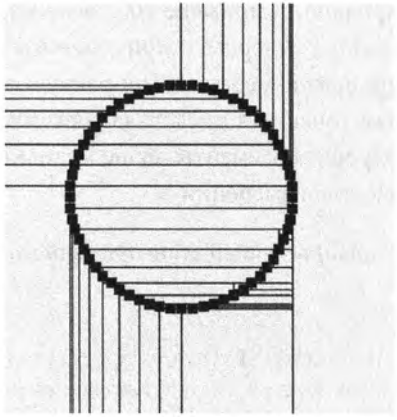


Рис. 10.8. Множество точек на окружности приводит к неэффективному k-d-дереву

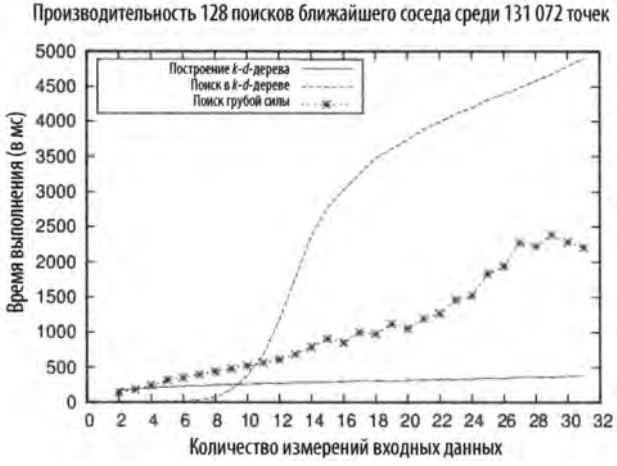


Рис. 10.9. Сравнение k-d-дерева и поиска грубой силой

Приведенные на рис. 10.9 результаты подтверждают, что с ростом количества измерений уменьшается преимущество использования алгоритма поиска ближайшего соседа с помощью *k-d*-деревьев над поиском грубой силой. Стоимость построения *k-d*-дерева не является главным фактором в уравнении, потому что она определяется главным образом количеством вставляемых в дерево точек, а не количеством измерений. Для больших размеров входного множества данных экономия носит выраженный характер. Еще одна причина ухудшения производительности с ростом d — увеличение времени вычисления евклидова расстояния между двумя d-мерными точками, которое является операцией со временем выполнения $O(d)$. Хотя ее можно по-прежнему рассматривать как операцию с константным временем выполнения, она все равно занимает больше времени.

Чтобы максимизировать производительность поиска с использованием *k-d*-дерева, оно должно быть сбалансировано. В примере 10.2 демонстрируется известный метод построения сбалансированного *k-d*-дерева с применением рекурсии для выполнения итерации по каждой из координат измерений. Попросту говоря, метод выбирает медианный элемент из множества точек для представления узла; элементы ниже медианы вставляются в нижнее поддерево, а элементы выше медианы — в верхнее. Код работает для произвольного количества измерений.

Пример 10.2. Рекурсивное построение сбалансированного k-d-дерева

```
public class KDFactory
{
    private static Comparator<IMultiPoint> comparators[];
    // Рекурсивное построение KDTree с использованием медианы.
    public static KDTree generate(IMultiPoint []points)
    {
        if (points.length == 0)
```

```
    {
        return null;
    }

    // Медиана будет корнем.
    int maxD = points[0].dimensionality();
    KDTree tree = new KDTree(maxD);
    // Создаем компаратор для сравнения точек по i-й координате.
    comparators = new Comparator[maxD + 1];

    for (int i = 1; i <= maxD; i++)
    {
        comparators[i] = new DimensionalComparator(i);
    }

    tree.setRoot(generate(1, maxD, points, 0, points.length-1));
    return tree;
}
// Генерация узла для d-го измерения (1 <= d <= maxD)
// для points[left, right]
private static DimensionalNode generate(int d, int maxD,
                                        IMultiPoint points[],
                                        int left, int right)
{
    // Сначала — обработка простых случаев
    if (right < left)
    {
        return null;
    }

    if (right == left)
    {
        return new DimensionalNode(d, points[left]);
    }

    // Упорядочение array[left,right] так, что m-й элемент
    // будет медианой, элементы до него меньше медианы,
    // хотя и не отсортированы. Аналогично элементы после
    // него больше медианы, хотя тоже не отсортированы.
    int m = 1 + (right - left) / 2;
    Selection.select(points, m, left, right, comparators[d]);
    // Медианная точка в этом измерении становится родителем
    DimensionalNode dm = new DimensionalNode(d, points[left+m-1]);

    // Обновление следующего измерения или сброс в 1
    if (++d > maxD)
    {
        d = 1;
    }
```

```
        // Рекурсивное вычисление левого и правого поддеревьев.
        dm.setBelow(maxD, generate(d, maxD, points, left, left+m-2));
        dm.setAbove(maxD, generate(d, maxD, points, left + m, right));
        return dm;
    }
}
```

Операция выбора описана в главе 4, "Алгоритмы сортировки". Она может выбрать k-е по величине значение рекурсивно за время $O(n)$ в среднем случае; однако в наихудшем случае она деградирует до $O(n^2)$.

Вариации алгоритма

В показанной реализации метод `nearest` выполняет проход от корня до вычисленного родителя; альтернативные реализации начинают от родителя и идут восходящим путем к корню.

Запрос диапазона

Для данной прямоугольной области R с координатами (x_{low}, y_{low}, x_{high}, y_{high}) и множества точек P требуется выяснить, какие из точек P содержатся внутри целевого прямоугольника T. Прямолинейный алгоритм грубой силы проверяет все точки P за время $O(n)$; нет ли возможности получить результат быстрее?

В случае алгоритма ближайшего соседа мы организовали точки в k-d-дерево для обработки запросов за время $O(log\ n)$. Сейчас мы покажем, как, используя ту же структуру данных, вычислить запрос диапазона за время $O(\sqrt{n}+r)$, где r — количество точек, возвращаемых запросом. Фактически, когда входное множество содержит d-мерные точки, ответить на запрос можно за время $O(n^{1\ 1/d}+r)$.

Алгоритм запроса диапазона

Наилучший и средний случаи: $O(n^{1-1/d} + r)$; наихудший случай: $O(n)$

```
range (space)
    results = new Set
    range (space, root, results)
    return results
end

range (space, node, results)
    if space содержит node.region then          ❶
        Добавляем node.points и всех его потомков в results
    return
```

```
    if space contains node.point then                    ❷
        Добавляем node.point в results

    if space расширяется ниже node.coord then            ❸
        range (space, node.below, results)
    if space расширяется выше node.coord then
        range (space, node.above, results)
end

❶ Если узел k-d-дерева полностью содержится в пространстве поиска,
   все потомки добавляются к результатам
❷ Обеспечивает добавление содержащейся точки к результатам.
❸ Может потребоваться поиск и ниже, и выше.
```

На рис. 10.10 мы видим целевую область T, которая охватывает левую половину плоскости до точки 7, и можем встретить все три указанные в описании алгоритма случая. Поскольку T не содержит всю бесконечную область, связанную с корневым узлом, алгоритм запроса диапазона проверяет, содержит ли T точку 1 (которую она содержит, так что эта точка добавляется к результату). Теперь T расширяется и на верхний, и на нижний дочерние узлы, так что алгоритм выполняет два рекурсивных вызова. В первой рекурсии *ниже* точки 1 T полностью содержит область, связанную с точкой 2. Таким образом, точка 2 и все ее потомки добавляются к результату. Во второй рекурсии в конечном итоге к результату добавляется обнаруженная точка 7.

Входные и выходные данные алгоритма

Входные данные алгоритма представляют собой множество P из n точек в d-мерном пространстве и d-мерный гиперкуб, определяющий запрашиваемую область. Запрашиваемая область выровнена вдоль осей координат d-мерного набора данных, так что для каждого измерения определяется отдельный диапазон; всего — d диапазонов для всех измерений входного множества. При $d=2$ запрос представляет собой запрос диапазонов по координате x и координате y.

Алгоритм запроса диапазона (Range Query) генерирует полное множество точек, находящихся в запрашиваемой области. Эти точки не обязательно находятся в некотором упорядоченном виде.

Контекст применения алгоритма

k-d-деревья для большого количества измерений становятся громоздкими, поэтому данный алгоритм и весь рассматриваемый подход должны ограничиваться данными с малым количеством измерений. Для двумерных данных k-d-деревья предлагают отличную производительность при поиске ближайшего соседа и при решении задачи о запросе диапазона.

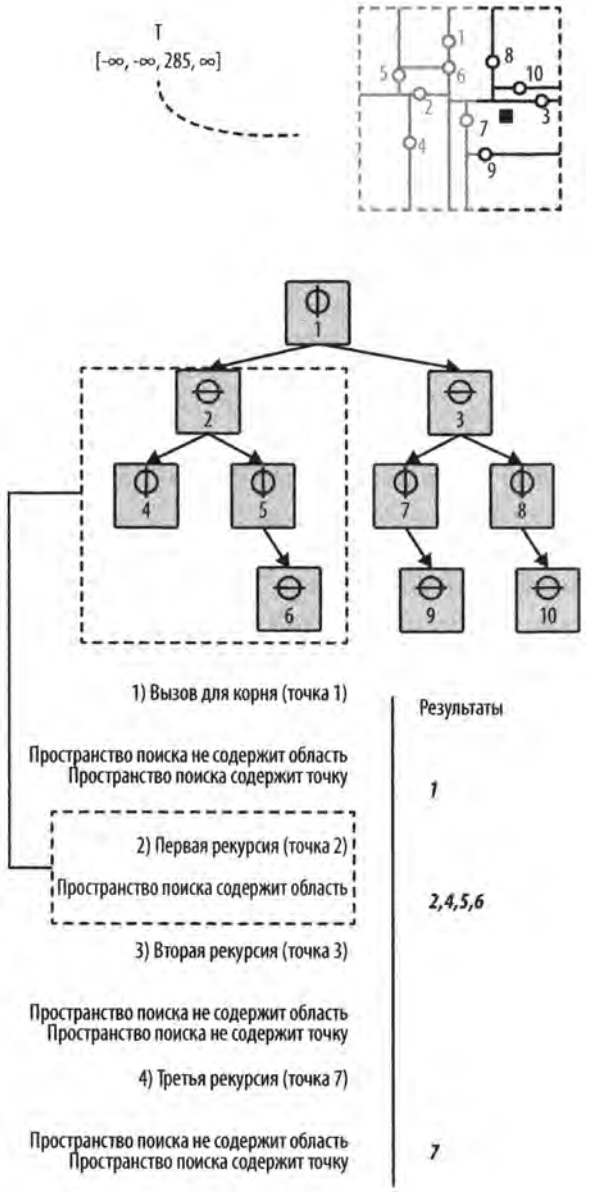


Рис. 10.10. Пример работы алгоритма поиска диапазона

Реализация алгоритма

Реализация алгоритма на языке программирования Java, показанная в примере 10.3, представляет собой метод класса `DimensionalNode`, который просто делегирован методом `range(IHypercube)` из `KDTree`. Высокая эффективность этого алгоритма проявляется, когда область `DimensionalNode` целиком содержится в пределах запрашиваемого

диапазона. В этом случае все узлы-потомки `DimensionalNode` могут быть добавлены к коллекции результатов в силу свойства *k-d*-дерева, гласящего, что дочерние узлы для данного узла целиком содержатся в области любого из его узлов-предков.

Пример 10.3. Реализация запроса диапазона в `DimensionalNode`

```
public void range(IHypercube space, KDSearchResults results)
{
    // Полностью содержится в области? Принимаем все его точки
    if (space.contains(region))
    {
        results.add(this);
        return;
    }

    // Наша точка содержится в области?
    if (space.intersects(cached))
    {
        results.add(point);
    }

    // При необходимости рекурсивно работает вдоль
    // обоих деревьев предков.
    if (space.getLeft(dimension) < coord)
    {
        if (below != null)
        {
            below.range(space, results);
        }
    }

    if (coord < space.getRight(dimension))
    {
        if (above != null)
        {
            above.range(space, results);
        }
    }
}
```

Показанный в примере 10.3 код представляет собой модифицированный обход дерева, который потенциально посещает каждый узел дерева. Поскольку *k-d*-дерево разделяет *d*-мерный набор данных в иерархическом порядке, есть три решения, которые принимает алгоритм запроса диапазона в каждом узле *n*.

Содержится ли связанная с узлом n область полностью в запрашиваемой области?

Когда это происходит, обход `range` можно останавливать, поскольку точки потомков принадлежат результату запроса.

Содержит ли запрашиваемая область точку, связанную с узлом n?

Если содержит, добавьте точку, связанную с *n*, в результирующее множество.

Пересекает ли запрошенная область n вдоль измерения d, представленного узлом n?

Это можно сделать двумя способами, поскольку область запроса может пересекаться как с областью, связанной с поддеревом `below` узла *n*, так и с областью, связанной с поддеревом `above`. Код может выполнить нуль, один или два рекурсивных обхода `range`.

Заметим, что возвращенный результат представляет собой объект `KDSearchResults`, содержащий как отдельные точки, так и целые поддеревья. Таким образом, чтобы получить все точки, вам придется обойти каждое поддерево результата.

Анализ алгоритма

Вполне возможно, что область запроса содержит все точки дерева; в этом случае будут возвращены все точки, что приводит к производительности $O(n)$. Однако когда алгоритм запроса диапазона обнаруживает, что область запроса не пересекается с отдельным узлом внутри *k-d*-дерева, он может сократить обход. Достигаемая при этом экономия стоимости зависит от количества измерений и специфики природы входного множества. Препарата (Preparata) и Шамос (Shamos) [47] показали, что запрос диапазона с использованием *k-d*-деревьев выполняется за время $O(n^{1-1/d}+r)$, где r — количество найденных результатов. По мере увеличения количества измерений преимущества алгоритма уменьшаются.

На рис. 10.11 показана ожидаемая производительность $O(n^{1-1/d}+r)$ алгоритма; отличительной особенностью графика является быстрая производительность для малых значений d, которая постепенно неумолимо приближается к $O(n)$. Из-за добавления r (количества точек, возвращенных запросом) фактическая производительность будет отклоняться от идеальной кривой, показанной на рис. 10.11.

Трудно сгенерировать множества входных данных, чтобы показать производительность алгоритма запроса диапазона. Мы демонстрируем эффективность алгоритма запроса диапазона с использованием *k-d*-дерева, сравнивая его производительность с реализацией алгоритма грубой силы, которая проверяет вхождение каждой точки в запрашиваемую область. *d*-мерные входные данные для каждой из этих ситуаций содержит n точек, координаты которых равномерно распределены в диапазоне $[0,s]$, где $s=4096$. Мы рассматриваем три ситуации.

Запрашиваемая область содержит все точки дерева

Мы строим область запроса, которая содержит все точки в *k-d*-дереве. В этом примере обеспечивается максимальная скорость алгоритма; его производительность не зависит от количества измерений d в *k-d*-дереве. Применение *k-d*-дерева требует примерно в 5–7 раз больше времени для завершения; это накладные расходы, присущие данной структуре. В табл. 10.2 время работы

для алгоритма грубой силы с увеличением d возрастает, поскольку вычисление, находится ли d-мерная точка внутри d-мерного пространства, является неконстантной операцией со временем работы $O(d)$. Реализация грубой силы превосходит реализацию k-d-дерева.

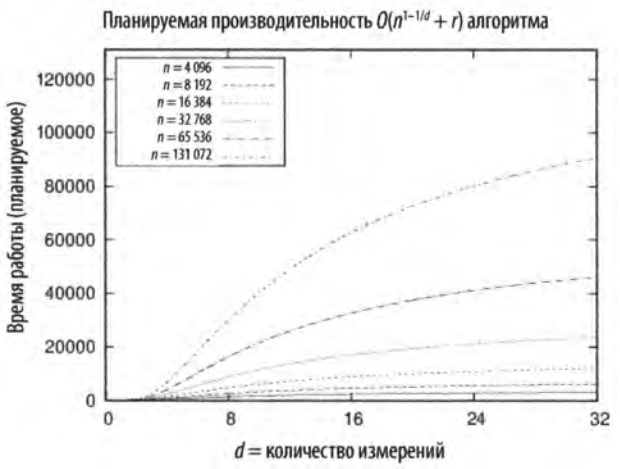


Рис. 10.11. Ожидаемая производительность $O(n^{1-1/d}+r)$ алгоритма

Таблица 10.2. Сравнение времени работы (в мс) алгоритма запроса диапазона (k-d-дерево (KD) и грубая сила (ГС)) для всех точек дерева

n	d = 2 KD	d = 3 KD	d = 4 KD	d = 5 KD	d = 2 ГС	d = 3 ГС	d = 4 ГС	d = 5 ГС
4096	6,22	13,26	19,6	22,15	4,78	4,91	5,85	6
8192	12,07	23,59	37,7	45,3	9,39	9,78	11,58	12
16384	20,42	41,85	73,72	94,03	18,87	19,49	23,26	24,1
32768	42,54	104,94	264,85	402,7	37,73	39,21	46,64	48,66
65536	416,39	585,11	709,38	853,52	75,59	80,12	96,32	101,6
131072	1146,82	1232,24	1431,38	1745,26	162,81	195,87	258,6	312,38

Частичные области

Поскольку количество найденных результатов r играет заметную роль в определении производительности алгоритма, мы строим набор сценариев для выделения этой переменной по мере роста числа измерений.

Равномерность распределения входных данных мешает нам просто построить запрашиваемую область $[s/2, s]$ для каждого измерения. Если бы мы поступили таким образом, то общий объем входного множества составлял бы $(1/2)^d$, а это означало бы, что с ростом d количество ожидаемых точек r не увеличивалось бы, а уменьшалось. Поэтому мы строим запрашиваемые области, размер которых увеличивается по мере увеличения d. Например, запрашиваемая область в двух измерениях с диапазоном $[0{,}5204s, s]$ по каждому измерению

должна возвращать примерно $0{,}23n$ точек, потому что $(1-0{,}5204)^2 = 0{,}23$. Однако для трех измерений необходимо расширить запрашиваемую область до $[0{,}3873s, s]$ по каждому измерению, поскольку $(1-0{,}3873)^3 = 0{,}23$.

Используя этот подход, мы заранее фиксируем желаемое соотношение k таким образом, чтобы наш сконструированный запрос возвращал kn точек (где k — 0,23, 0,115, 0,0575, 0,02875 или 0,014375). Мы сравниваем реализации k-d-дерева с подходом с использованием грубой силы при n, варьирующемся от 4096 до 131 072, и d в пределах от 2 до 15, как показано на рис. 10.12. Графики слева демонстрируют поведение алгоритма k-d-дерева ($O(n^{1-1/d})$), в то время как графики справа показывают линейную производительность алгоритма грубой силы. Коэффициент 0,23 показывает преимущества реализации k-d-дерева перед алгоритмом грубой силы только для $d = 2$ и $n \leq 8192$. Для отношении 0,014375 реализация k-d-дерева выигрывает при $d \leq 6$ и $n \leq 131\,072$.

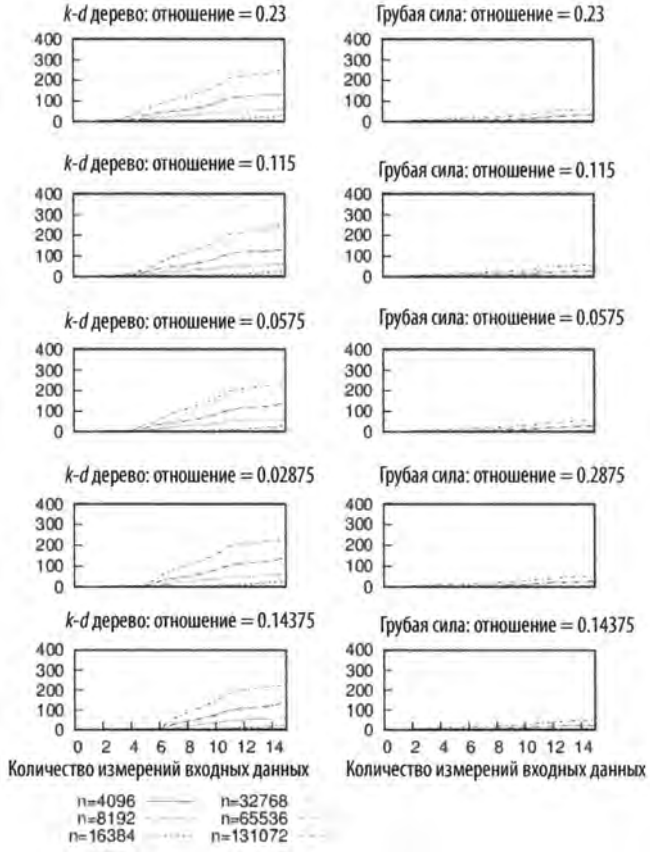


Рис. 10.12. Сравнение применения k-d-деревьев с подходом с использованием грубой силы для частичных областей

Пустая область

Мы строим область запроса из одной точки, случайным образом выбираемой из равномерного входного множества данных. Результаты показаны в табл. 10.3. Алгоритм с использованием *k-d*-дерева выполняется почти мгновенно; все измеренные времена выполнения были меньше доли миллисекунды, так что в таблице приведены данные только для подхода грубой силы.

Таблица 10.3. Время работы (в мс) запроса диапазона с помощью алгоритма грубой силы для пустой области

n	*d* = 2	*d* = 3	*d* = 4	*d* = 5
4096	3,36	3,36	3,66	3,83
8192	6,71	6,97	7,30	7,50
16384	13,41	14,02	14,59	15,16
32768	27,12	28,34	29,27	30,53
65536	54,73	57,43	60,59	65,31
131072	124,48	160,58	219,26	272,65

Деревья квадрантов

Деревья квадрантов могут использоваться для выполнения следующих запросов.

Запросы диапазона

Для заданного набора точек в декартовой системе координат требуется определить точки, которые находятся внутри указанного в запросе прямоугольника. Простое приложение алгоритма показано на рис. 10.13, на котором пользователь указывает нарисованный пунктиром прямоугольник, и на рисунке выделяются точки, содержащиеся в этом прямоугольнике. Когда область дерева квадрантов полностью входит в целевой прямоугольник запроса, приложение изображает эту область с затененным фоном.

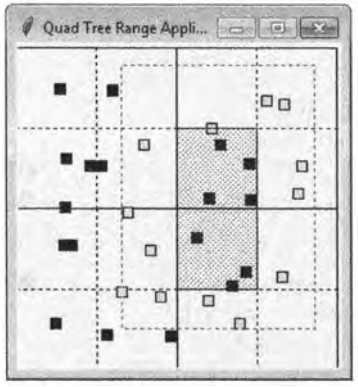


Рис. 10.13. Запрос диапазона с использованием дерева квадрантов

Обнаружение столкновений

Для данной коллекции объектов в декартовой системе координат требуется определить все пересечения между объектами. На рис. 10.14 показан пример такого приложения, которое идентифицирует столкновения между рядом движущихся квадратов, которые движутся назад и вперед в окне, отражаясь от стен. Квадраты, пересекающиеся один с другим, будут выделяться.

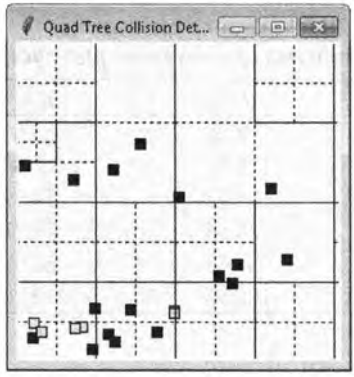


Рис. 10.14. Обнаружение соударений с использованием дерева квадрантов

Использование Quadtree

Наилучший, средний и наихудший случаи: $O(\log n)$

```
add (node, pt)
    if node.region не содержит pt then
        return false
    if node является листом then
        if node уже содержит pt then                ❶
            return false
        if node has < 4 points then                 ❷
            Добавление pt в node
            return true

    q = квадрант для pt в node
    if node является листом then
        node.subdivide()                            ❸
    return add(node.children[q], pt)                ❹

range (node, rect, result)
    if rect содержит node.region then               ❺
        Добавить (node, true) в result
    else if node является листом then
        foreach точка p в node do
            if rect содержит p then
                Добавить (p, false) в result        ❻
```

```
    else
        foreach Дочерний узел из node.children do
            if rect перекрывается с child.region
                range(child, rect, result)          ❼

❶ Наложение семантики множества на дерево квадрантов.
❷ Каждый узел может хранить до четырех точек.
❸ Точки листа распределяются между четырьмя новыми дочерними узлами.
❹ Вставка новой точки в корректный дочерний узел.
❺ Все поддерево содержится в rect и возвращается.
❻ Возвращаются отдельные точки.
❼ Рекурсивно проверяется каждый перекрывающийся потомок.
```

Входные и выходные данные алгоритма

Входными данными является множество *P* двумерных точек на плоскости, на основании которого построено дерево квадрантов.

Для оптимальной производительности запрос диапазона возвращает *узлы* дерева квадрантов, что позволяет возвращать поддеревья точек полностью, когда все поддерево содержится в целевом прямоугольнике. Результатом обнаружения столкновений являются существующие точки, которые пересекаются с целевой.

Реализация алгоритма

Основой реализации алгоритма дерева квадрантов на языке программирования Python являются структуры `QuadTree` и `QuadNode`, показанные в примере 10.4. Вспомогательные методы `smaller2k` и `larger2k` гарантируют, что начальная область имеет размеры сторон, равные степени двойки. Класс `Region` представляет прямоугольную область.

Пример 10.4. Реализация дерева квадрантов QuadNode

```
class Region:
    def __init__(self, xmin,ymin, xmax,ymax):
        """
        Создает область из двух точек - от (xmin,ymin) до
        (xmax,ymax). Выполняет коррекцию, если это не нижний
        левый и верхний правый углы области.
        """
        self.x_min = xmin if xmin < xmax else xmax
        self.y_min = ymin if ymin < ymax else ymax
        self.x_max = xmax if xmax > xmin else xmin
        self.y_max = ymax if ymax > ymin else ymin
```

```
class QuadNode:
    def __init__(self, region, pt = None, data = None):
        """
        Создание пустого QuadNode отцентрированного
        в заданной области.
        """
        self.region = region
        self.origin = (region.x_min+(region.x_max-region.x_min)//2,
                       region.y_min+(region.y_max-region.y_min)//2)
        self.children = [None] * 4

        if pt:
            self.points = [pt]
            self.data = [data]
        else:
            self.points = []
            self.data = []

class QuadTree:
    def __init__(self, region):
        """
        Создание QuadTree поверх квадратной области
        со сторонами, являющимися степенями двойки.
        """
        self.root = None
        self.region = region.copy()

        xmin2k = smaller2k(self.region.x_min)
        ymin2k = smaller2k(self.region.y_min)
        xmax2k = larger2k(self.region.x_max)
        ymax2k = larger2k(self.region.y_max)

        self.region.x_min = self.region.y_min = min(xmin2k, ymin2k)
        self.region.x_max = self.region.y_max = max(xmax2k, ymax2k)
```

Точки добавляются в дерево квадрантов с помощью метода `add`, показанного в примере 10.5. Метод `add` возвращает значение `False`, если точка уже содержится в дереве квадрантов; таким образом обеспечивается семантика математического множества. К узлу добавляются до четырех точек, если точка содержится внутри прямоугольной области этого узла. Когда добавляется пятая точка, область узла подразделяется на квадранты и точки переназначаются индивидуальным квадрантам области этого узла; процесс повторяется, пока все квадранты листьев не будут иметь не более четырех точек.

Пример 10.5. Реализация добавления точек в дерево квадрантов

```
class QuadNode:
    def add (self, pt, data):
```

```
        """Добавление (pt, data) в QuadNode."""
        node = self
        while node:
            # Невозможно поместить в эту область.
            if not containsPoint (node.region, pt):
                return False

            # Если есть точки, это лист. Проверяем.
            if node.points != None:
                if pt in node.points:
                    return False

            # Добавление, если есть куда.
            if len(node.points) < 4:
                node.points.append (pt)
                node.data.append (data)
                return True

        # Находим квадрант для добавления.
        q = node.quadrant (pt)
        if node.children[q] is None:
            # Разделение и переназначение точек квадрантам.
            # Затем добавление точек.
            node.subdivide()
        node = node.children[q]

        return False

class QuadTree:
    def add (self, pt, data = None):
        if self.root is None:
            self.root = QuadNode(self.region, pt, data)
            return True

        return self.root.add (pt, data)
```

В примере 10.6 показано, как при работе с этой структурой метод `range` эффективно находит все точки в дереве квадрантов, содержащиеся в целевой области. Приведенная реализация на языке программирования Python использует оператор `yield` для предоставления интерфейса итератора для результатов. Итератор содержит кортежи, которые являются либо отдельными точками, либо узлами. Когда узел дерева квадрантов целиком содержится в области, то как часть результата возвращается весь узел. Вызывающая функция может извлечь значения всех потомков с помощью прямого обхода узла, предоставляемого `QuadNode`.

Пример 10.6. Реализация запроса диапазона

```
class QuadNode:
    def range(self, region):
        """
        Дает (node,True), когда узел содержится в области,
        в противном случае (region,False) для отдельных точек.
        """
        if region.containsRegion (self.region):
            yield (self, True)
        else:
            # Если мы имеем точки, значит, это лист. Проверяем.
            if self.points != None:
                for i in range(len(self.points)):
                    if containsPoint (region, self.points[i]):
                        yield ((self.points[i], self.data[i]), False)
            else:
                for child in self.children:
                    if child.region.overlap (region):
                        for pair in child.range (region):
                            yield pair

class QuadTree:
    def range(self, region):
        """Дает (node,status) в дереве, содержащемся в области."""
        if self.root is None:
            return None
        return self.root.range(region)
```

Для поддержки обнаружения столкновений в примере 10.7 содержится метод `collide`, который выполняет в дереве квадрантов поиск точек, пересекающихся с квадратом стороной *r* и центром в данной точке *pt*.

Пример 10.7. Реализация обнаружения столкновений

```
class QuadNode:
    def collide (self, pt, r):
        """
        Дает точки в листе, которые пересекаются с квадратом
        со стороной r и центром в точке pt.
        """
        node = self
        while node:
            # Точка должна быть в области
            if containsPoint (node.region, pt):
              # Если есть точки, это лист. Проверяем.
              if node.points != None:
                for p,d in zip(node.points, node.data):
```

```
                if p[X]-r<=pt[X]<=p[X]+r and p[Y]-r<=pt[Y]<=p[Y]+r:
                  yield (p, d)
              # Находим квадрант для дальнейшей проверки
              q = node.quadrant (pt)
              node = node.children[q]

class QuadTree:
    def collide(self, pt, r):
        """Возврат столкновений с точкой в QuadTree."""
        if self.root is None:
            return None

        return self.root.collide (pt, r)
```

Анализ алгоритма

Деревья квадрантов разделяют точки в плоскости, используя ту же базовую структуру, что и бинарные деревья поиска. Показанная здесь реализация на основе областей использует фиксированную схему разделения, которая обеспечивает эффективное поведение при равномерном распределении точек множества. Может случиться так, что все точки группируются в небольшом пространстве, как показано на рис. 10.15. Таким образом, производительность поиска логарифмическая по отношению к размеру дерева. Эффективность запроса диапазона в Python обеспечивается возвратом как отдельных точек, так и целых узлов дерева квадрантов. Однако по-прежнему необходимо учитывать время на извлечение всех значений-потомков в узлах, возвращенных запросом диапазона.

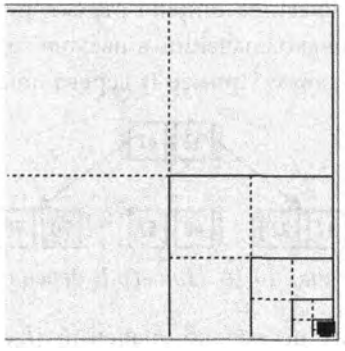


Рис. 10.15. Вырожденное дерево квадрантов

Вариации алгоритма

Структура дерева квадрантов, представленная здесь, является деревом квадрантов для областей. Дерево квадрантов для точек представляет двумерные точки.

Дерево октантов (octree) расширяет дерево квадрантов на три измерения, с восемью дочерними узлами вместо четырех [39].

R-деревья

Сбалансированные двоичные деревья представляют собой невероятно универсальные структуры данных, которые обеспечивают большую производительность операций поиска, вставки и удаления. Однако лучше всего они работают в основной памяти, используя указатели и выделение и освобождение памяти для узлов при необходимости. Эти деревья могут расти только до пределов, определяемых количеством основной памяти, и плохо приспособлены для хранения во вторичной памяти, такой как файловая система. Операционные системы предоставляют *виртуальную память*, так что программы могут работать с памятью, объем которой может превышать фактический объем физической памяти. Операционная система обеспечивает наличие требуемого для работы блока памяти (известного как *страница*) в основной памяти, а старые неиспользуемые страницы при этом сохраняются на диске (если они были изменены) и выбрасываются из основной памяти. Программы работают наиболее эффективно при объединении доступа для чтения и записи с использованием страниц, которые, как правило, имеют размер 4096 байтов. Если считать, что узел в бинарном дереве требует 24 байта для сохранения, то в одной странице могут храниться десятки таких узлов. Однако не очевидно, как же хранить эти узлы на диске, в особенности в условиях постоянного обновления дерева.

В 1972 году была разработана концепция B-дерева [7], хотя она, как представляется, еще до того была открыта разработчиками систем управления базами данных и операционных систем. B-дерево расширяет структуру бинарного дерева, позволяя каждому узлу хранить несколько значений и несколько ссылок на узлы, количество которых может быть больше двух. Пример B-дерева показан на рис. 10.16.

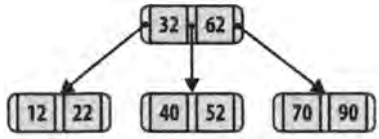


Рис. 10.16. Пример B-дерева

Каждый узел n содержит несколько значений $\{k_1, k_2, ..., k_{m-1}\}$, упорядоченных по возрастанию, и указателей $\{p_1, p_2, ..., p_m\}$, где m определяет максимальное количество дочерних узлов, на которые может указывать n. Значение m называется *порядком* B-дерева. Каждый узел B-дерева может содержать $m-1$ значений.

Для поддержания *свойства бинарного дерева поиска* узлы B-дерева хранят значения ключей таким образом, чтобы все значения в поддереве, на которое указывает p_1, были меньше, чем k_1. Все значения в поддереве, на которое указывает p_i, больше

или равны k_i и меньше, чем k_{i+1}. И наконец все значения в поддереве, на которое указывает указатель p_m, больше k_{m-1}.

Используя определение Кнута, можно считать, что B-дерево порядка m удовлетворяет следующим требованиям.

- Каждый узел имеет не больше m дочерних узлов.
- Каждый внутренний узел (за исключением корня) имеет как минимум $\lceil m/2 \rceil$ дочерних узла.
- Корень имеет как минимум два дочерних узла, если он не является листом.
- Внутренний узел с k дочерними узлами содержит $k-1$ ключевых значений.
- Все листья находятся на одном и том же уровне.

При использовании этого определения традиционное бинарное дерево является вырожденным B-деревом порядка $m=2$. Вставки и удаления в B-дереве должны поддерживать перечисленные свойства. При этом самый длинный путь в B-дереве с n ключами содержит не более $log_m n$ узлов, что приводит к производительности операций дерева $O(log\ n)$. B-деревья могут легко храниться во вторичной памяти путем увеличения числа ключей в каждом узле таким образом, чтобы его общий размер был корректно выровнен по размеру страницы (например, для хранения в одной странице двух узлов B-дерева), что сводит к минимуму число операций чтения диска для загрузки узлов в основную память.

После этого краткого наброска о B-деревьях можно переходить к более подробному описанию структуры R-дерева. R-дерево является деревом со сбалансированной высотой, похожим на B-дерево, которое хранит n-мерные пространственные объекты в динамической структуре, поддерживающей операции вставки, удаления и запроса. Оно также поддерживает *запросы диапазонов* для поиска перекрывающихся объектов. В этой главе мы опишем, как R-деревья обеспечивают эффективное выполнение основных операций.

R-дерево является древовидной структурой, в которой каждый узел содержит до M ссылок на различные дочерние узлы. Вся информация хранится в листьях, каждый из которых может хранить до M различных n-мерных пространственных объектов. Лист R-дерева хранит не сами объекты (которые на самом деле хранятся в некотором репозитории), а только индексы этих объектов в хранилище; R-дерево содержит только уникальные идентификаторы каждого объекта и n-мерные ограничивающие боксы I, которые представляют собой наименьшие n-мерные фигуры, содержащие пространственный объект. Здесь мы предполагаем наличие только двух измерений, а упомянутые фигуры представляют собой прямоугольники; но они естественным образом могут быть расширены на n измерений.

Имеется еще одна константа $m \leq \lfloor M/2 \rfloor$, которая определяет минимальное количество значений, хранящихся в листьях, или ссылок, хранящихся во внутренних узлах R-дерева. Подытожим свойства R-деревьев.

- Каждый лист содержит от m до M (включительно) записей [если только это не корень дерева].
- Каждый не листовой узел содержит от m до M (включительно) ссылок на дочерние узлы [если только это не корень дерева].
- Для каждой записи $(I, child_i)$ во внутреннем узле I представляет собой наименьший прямоугольный пространственный объект, содержащий соответствующие прямоугольники из дочерних узлов.
- Корневой узел имеет как минимум два дочерних узла [если только он не является листом].
- Все листья дерева находятся на одном уровне.

R-дерево представляет собой сбалансированную структуру, которая обеспечивает выполнение перечисленных свойств. Для удобства конечный уровень считается нулевым уровнем, а номер уровня корневого узла является высотой дерева. Структура R-дерева поддерживает вставку, удаление и запрос за время $O(log_m n)$.

В репозитории к данной книге содержатся примеры приложений для изучения поведения R-деревьев. На рис. 10.17 показано динамическое поведение R-дерева с $M = 4$ объектами при добавлении в него нового прямоугольника 6. Как вы можете видеть, места для нового прямоугольника нет, поэтому соответствующий конечный узел R1 делится на два узла на основе метрики, которая стремится минимизировать площадь соответствующего нового внутреннего узла R2. Добавление этого прямоугольника увеличивает общую высоту R-дерева на единицу, так как при этом создается новый его корень R3.

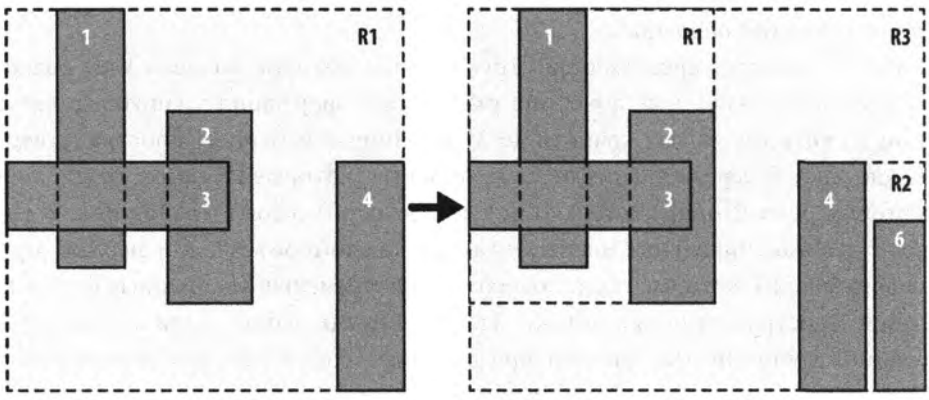


Рис. 10.17. Пример вставки в R-дерево

R-дерево

Наилучший и средний случаи: $O(\log_m n)$; наихудший случай: $O(n)$

```
add (R, rect)
    if R - пустое дерево then
        R = new RTree (rect)
    else
        leaf = chooseLeaf(rect)
        if leaf.count < M then                                    ❶
            Добавить rect к leaf
        else
            newLeaf = leaf.split(rect)                            ❷
            newNode = adjustTree (leaf, newLeaf)                  ❸
            if newNode != null then
                R = новое RTree со старым Root и newNode          ❹
                    в качестве дочерних узлов
end

search (n, t)
    if n - лист then                                              ❺
        return Запись, если n содержит t, иначе вернуть False
    else
        foreach узла c, дочернего для узла n do                   ❻
            if c содержит t then
                return search(c, t)                               ❼
end

range (n, t)
    if целевой объект полностью содержится в боксе n then         ❽
        return Все боксы-потомки
    else if n - лист
        return Все записи, пересекающиеся с t
    else
        result = null
        foreach узла c, дочернего для узла n do                   ❾
            if c перекрывается с t then
                result = объединение result и range(c, t)
        return result
```

❶ Записи добавляются в лист при наличии в нем места.
❷ В противном случае M+1 записей делятся между старым листом и новым узлом.
❸ Может потребоваться обновление записей на пути до корня.
❹ R-дерево при добавлении может вырасти в высоту.
❺ Листья содержат фактические записи.

```
❻ Рекурсивный поиск в каждом дочернем узле, так как области
   могут перекрываться.
❼ Возврат найденной записи.
❽ Эффективная операция по определению членства.
❾ Может потребоваться выполнение нескольких рекурсивных запросов.
```

Входные и выходные данные алгоритма

Двумерное R-дерево хранит коллекцию прямоугольных областей в декартовой системе координат, каждая из которых (необязательно) имеет собственный уникальный идентификатор.

Операции над R-деревом могут изменять состояние R-дерева (например, выполнять вставки или удаления), а также извлекать отдельные прямоугольные области или коллекции областей.

Контекст применения алгоритма

R-деревья были разработаны для индексирования многомерной информации, включая географические структуры или более абстрактные *n*-мерные данные, такие как прямоугольники или многоугольники. Это одна из немногих структур, которые обеспечивает отличную производительность времени выполнения, даже если информация слишком велика для хранения в основной памяти. Традиционные методы индексации по своей природе одномерны, и, таким образом, для указанных предметных областей структура R-дерева является наиболее подходящим решением.

Операции над R-деревом включают возможность вставки и удаления, а также имеется два вида запросов. Вы можете искать конкретную прямоугольную область в R-дереве, а можете определить коллекцию прямоугольных областей, которые пересекаются с запрашиваемым прямоугольником.

Реализация алгоритма

Реализация на языке программирования Python в примере 10.8 связывает необязательный идентификатор с каждым прямоугольником; он будет использоваться для извлечения фактического пространственного объекта из базы данных. Мы начинаем с класса `RNode`, который является фундаментальной единицей R-дерева. Каждый `RNode` поддерживает охватывающую область и необязательный идентификатор. `RNode` является листом, если значение `node.level` равно нулю. У `RNode` имеется `node.count` дочерних узлов, которые хранятся в списке `node.children`. При добавлении дочернего узла `RNode` родительский охватывающий бокс `node.region` должен быть обновлен таким образом, чтобы в него был включен вновь добавленный потомок.

Пример 10.8. Реализация `RNode`

```
class RNode:
    # Монотонно увеличивающийся счетчик для генерации идентификаторов
    counter = 0

    def __init__(self, M, rectangle=None, ident=None, level=0):
        if rectangle:
            self.region = rectangle.copy()
        else:
            self.region = None

        if ident is None:
            RNode.counter += 1
            self.id = 'R' + str(RNode.counter)
        else:
            self.id = ident

        self.children = [None] * M
        self.level = level
        self.count = 0

    def addRNode(self, rNode):
        """
        Добавление ранее вычисленного RNode и
        обновление охватывающей области.
        """
        self.children[self.count] = rNode
        self.count += 1

        if self.region is None:
            self.region = rNode.region.copy()
        else:
            rectangle = rNode.region
            if rectangle.x_min < self.region.x_min:
                self.region.x_min = rectangle.x_min
            if rectangle.x_max > self.region.x_max:
                self.region.x_max = rectangle.x_max
            if rectangle.y_min < self.region.y_min:
                self.region.y_min = rectangle.y_min
            if rectangle.y_max > self.region.y_max:
                self.region.y_max = rectangle.y_max
```

При наличии этой базы пример 10.9 описывает класс `RTree` и метод добавления прямоугольника в R-дерево.

Пример 10.9. Реализация `RTree` и добавления в него

```
class RTree:
    def __init__(self, m=2, M=4):
        """
        Создание пустого R-дерева со значениями
        по умолчанию (m=2, M=4).
        """
        self.root = None
        self.m = m
        self.M = M

    def add(self, rectangle, ident = None):
        """
        Вставка прямоугольника в корректное местоположение
        с (необязательным) идентификатором.
        """
        if self.root is None:
            self.root = RNode(self.M, rectangle, None)
            self.root.addEntry (self.M, rectangle, ident)
        else:
            # I1 [Поиск позиции для новой записи] Вызов ChooseLeaf
            # для выбора листа L, в котором будет размещен E.
            # Возвращает путь к листу.
            path = self.root.chooseLeaf (rectangle, [self.root]);
            n = path[-1]
            del path[-1]

            # I2 [Добавление записи в лист] Если в L имеется место
            # для другой записи, вносим E. В противном случае
            # вызываем SplitNode для получения L и LL, содержащих E
            # и все старые записи L.
            newLeaf = None
            if n.count < self.M:
                n.addEntry (self.M, rectangle, ident)
            else:
                newLeaf = n.split(RNode(self.M,rectangle,ident,
                                        0),self.m,self.M)

            # I3 [Распространение изменений вверх] Вызываем
            # AdjustTree для L, передавая также LL, если было
            # выполнено разделение.
            newNode = self.adjustTree (n, newLeaf, path)

            # I4 [Рост дерева] Если распространение разбиения
            # узлов достигает корня и он должен быть разделен,
            # создаем новый корень, дочерние узлы которого
            # представляют собой два получающихся в результате узла.
```

```
        if newNode:
            newRoot = RNode(self.M, level = newNode.level + 1)
            newRoot.addRNode (newNode)
            newRoot.addRNode (self.root)
            self.root = newRoot
```

Комментарии в примере 10.9 отражают шаги алгоритма, опубликованного в статье [29]. Каждый объект `RTree` записывает значения *m* и *M* и корневой объект `RNode` дерева. После добавления первого прямоугольника в пустое дерево `RTree` просто создается первоначальная структура. После этого метод `add` находит соответствующий лист, в который следует добавить новый прямоугольник. Вычисленный список `path` возвращает упорядоченные узлы от корня до выбранного листа.

Если в выбранном листе достаточно места, новый прямоугольник добавляется в него, а изменения ограничивающих прямоугольников распространяются вверх до корня с помощью метода `adjustTree`. Однако если выбранный лист заполнен, строится лист `newLeaf`, и $M+1$ записей разделяются между `n` и `newLeaf` с использованием стратегии минимизации общей площади ограничивающих боксов этих двух узлов. В этом случае метод `adjustTree` должен также распространять новую структуру вверх до корня, что может вызвать разделение других узлов аналогичным образом. Если разделяется исходный корневой узел `self.root`, то новый корень `RNode` создается как родительский узел исходного корня и вновь создаваемого объекта `RNode`. Таким образом, дерево `RTree` растет только на один уровень.

Запросы диапазона можно выполнять так, как показано в примере 10.10. Эта реализация очень краткая из-за способности Python писать функции-генераторы, которые ведут себя как итераторы. Рекурсивный метод `range` класса `RNode` сначала проверяет, не содержит ли целевой прямоугольник данный `RNode` полностью; если содержит, то прямоугольники всех его листьев-потомков должны быть включены в результат. В качестве заполнителя возвращается кортеж `(self,0,True)`. Вызываемая функция может извлечь все эти области с помощью генератора `leafOrder`, определенного классом `RNode`. В противном случае для внутренних узлов функция рекурсивно дает те прямоугольники, которые найдены в потомках, ограничивающие прямоугольники которых пересекаются с целевым. Прямоугольники в листьях возвращаются в виде `(rectangle,id,False)`, если целевой прямоугольник перекрывается с ограничивающими прямоугольниками этих листьев.

Пример 10.10. Реализация запроса диапазона с использованием `RTree/RNode`

```
class RNode:
  def range (self, target):
    """
    Возвращает генератор (node,0,True) или (rect,id,False)
    всех квалифицированных идентификаторов, перекрывающихся
    с целевой областью.
```

```
    """
    # Полностью содержит все внутренние узлы? Возвращаем весь узел.
    if target.containsRegion (self.region):
        yield (self, 0, True)
    else:
        # Проверка листьев и рекурсия
        if self.level == 0:
            for idx in range(self.count):
                if target.overlaps (self.children[idx].region):
                    yield(self.children[idx].region,
                          self.children[idx].id,False)
        else:
            for idx in range(self.count):
                if self.children[idx].region.overlaps (target):
                    for triple in self.children[idx].range (target):
                        yield triple

class RTree:
  def range (self, target):
    """
    Возвращает генератор для всех квалифицированных
    (node,0,True) или (rect,id,False), перекрывающихся
    с целевой областью
    """
    if self.root:
        return self.root.range (target)
    else:
        return None
```

Поиск отдельного прямоугольника имеет ту же структуру, что и код в примере 10.10; только функция `search` класса `RNode` имеет вид, показанный в примере 10.11. Эта функция возвращает прямоугольник и необязательный идентификатор, использовавшийся при вставке прямоугольника в `RTree`.

Пример 10.11. Реализация запроса поиска с использованием `RNode`

```
class RNode:
  def search (self, target):
    """Возвращает (rectangle,id), если узел содержит
       целевой прямоугольник."""
    if self.level == 0:
        for idx in range(self.count):
            if target == self.children[idx].region:
                return(self.children[idx].region,self.children[idx].id)
    elif self.region.containsRegion (target):
        for idx in range(self.count):
            if self.children[idx].region.containsRegion (target):
                rc = self.children[idx].search(target)
```

```
            if rc:
                return rc
    return None
```

Для завершения реализации R-деревьев нам нужна возможность удаления прямоугольника, который имеется в дереве. В то время как метод `add` выполняет разделение слишком заполненных узлов, метод `remove` должен обрабатывать узлы, имеющие слишком мало дочерних узлов для заданного минимального числа дочерних узлов *m*. Ключевая идея состоит в том, что, как только прямоугольник будет удален из R-дерева, любой из узлов на пути от родительского узла до корня может стать "недозаполненным". Реализация, показанная в примере 10.12, обрабатывает эту ситуацию с помощью вспомогательного метода `condenseTree`, который возвращает список "осиротевших" узлов с количеством дочерних узлов, меньшим, чем *m*. Затем "осиротевшие" узлы заново вставляются в R-дерево.

Пример 10.12. Реализация операции `remove` *для* `RNode`

```
class RTree:
  def remove(self, rectangle):
    """Удаление прямоугольника из RTree."""
    if self.root is None:
        return False

    # D1 [Поиск узла с записью] Вызываем FindLeaf для обнаружения
    # листа n, содержащего R. Останов, если запись не найдена.
    path = self.root.findLeaf (rectangle, [self.root]);
    if path is None:
        return False

    leaf = path[-1]
    del path[-1]
    parent = path[-1]
    del path[-1]

    # D2 [Удаление записи.] Удаляем E из n
    parent.removeRNode (leaf)

    # D3 [Распространение изменений] Вызов condenseTree для parent
    if parent == self.root:
        self.root.adjustRegion()
    else:
        parent,Q = parent.condenseTree (path, self.m, self.M)
        self.root.adjustRegion()

    # CT6 [Повторная вставка "осиротевших" записей] Заново
    # вставляем все записи узлов из множества Q.
    for n in Q:
```

```
        for rect,ident in n.leafOrder():
            self.add (rect, ident)

    # D4 [Уменьшение дерева] Если после выполнения обновления
    # корневой узел имеет только один дочерний узел, делаем этот
    # дочерний узел новым корнем.
    while self.root.count == 1 and self.root.level > 0:
        self.root = self.root.children[0]
    if self.root.count == 0:
        self.root = None

    return True
```

Анализ алгоритма

Эффективность структуры R-дерева обеспечивается способностью поддерживать сбалансированность при вставке прямоугольников. Так как все прямоугольники хранятся *в листьях R-дерева на одной и той же высоте*, внутренние узлы представляют собой вспомогательную структуру, обеспечивающую эффективную работу. Параметры m и M определяют детали этой структуры, но главное — гарантия, что высота дерева будет составлять $O(\log n)$, где n — количество узлов в R-дереве. Метод `split` распределяет прямоугольники между двумя узлами с помощью эвристики, которая минимизирует общую площадь охватывающего бокса для этих двух узлов; в литературе были предложены и другие разновидности эвристик.

Производительность методов поиска в R-дереве зависит от количества прямоугольников в R-дереве и *плотности* этих прямоугольников (среднего числа прямоугольников, содержащих данную точку). Для n прямоугольников, образованных случайным выбором координат в единичном квадрате, случайную точку пересекают около 10% из них. Это означает, что *поиск* должен исследовать несколько дочерних узлов в попытке найти отдельный прямоугольник. В частности, поиск должен исследовать каждый узел-потомок, область которого перекрывается с целевой областью запроса. Для наборов данных с низкой плотностью поиск в R-дереве становится более эффективным.

Вставка прямоугольников в R-дерево может вызвать разделение нескольких узлов, что является дорогостоящей операцией. Аналогично при удалении прямоугольников из R-дерева может быть несколько "осиротевших" узлов, прямоугольники которых должны быть вновь вставлены в дерево. Удаление прямоугольников оказывается более эффективным, чем поиск, так как в процессе обнаружения удаляемых прямоугольников рекурсивные вызовы ограничены потомками, которые полностью содержат целевой удаляемый прямоугольник.

В табл. 10.4 показаны результаты для двух множеств, содержащих 8100 прямоугольников. В табл. 10.4–10.6 мы приводим результаты производительности для различных значений m и M (напомним, что $m \leq \lfloor M/2 \rfloor$). В *разреженном* множестве все прямоугольники имеют одинаковый размер, но не перекрываются. В *плотном*

множестве прямоугольники формируются из двух случайных точек, равномерно выбранных из единичного квадрата. В таблицах указано общее время построения R-дерева из прямоугольников. Время построения несколько выше для плотного множества из-за увеличения количества узлов, которые должны быть разделены при вставке прямоугольников.

В табл. 10.5 показано общее время поиска всех прямоугольников в R-дереве. Поиск в плотном множестве данных выполняется примерно в 50 раз медленнее, чем в разреженном. Кроме того, он демонстрирует минимальное преимущество при принятии условия $m = 2$. В табл. 10.6 содержатся соответствующие показатели для удаления всех прямоугольников из R-дерева. Пиковые значения производительности для плотного множества, вероятно, являются результатом малого размера случайного множества данных.

Таблица 10.4. Производительность построения R-дерева для плотных и разреженных множеств данных

	Плотные данные					Разреженные данные				
M	***m* = 2**	***m* = 3**	***m* = 4**	***m* = 5**	***m* = 6**	***m* = 2**	***m* = 3**	***m* = 4**	***m* = 5**	***m* = 6**
4	1,32					1,36				
5	1,26					1,22				
6	1,23	1,23				1,2	1,24			
7	1,21	1,21				1,21	1,18			
8	1,24	1,21	1,19			1,21	1,20	1,19		
9	1,23	1,25	1,25			1,20	1,19	1,18		
10	1,35	1,25	1,25	1,25		1,18	1,18	1,18	1,22	
11	1,30	1,34	1,27	1,24		1,18	1,21	1,22	1,22	
12	1,30	1,31	1,24	1,28	1,22	1,17	1,21	1,20	1,20	1,25

Таблица 10.5. Производительность поиска в R-дереве для плотных и разреженных множеств данных

	Плотные данные					Разреженные данные				
M	***m* = 2**	***m* = 3**	***m* = 4**	***m* = 5**	***m* = 6**	***m* = 2**	***m* = 3**	***m* = 4**	***m* = 5**	***m* = 6**
4	25,16					0,45				
5	21,73					0,48				
6	20,98	21,66				0,41	0,39			
7	20,45	20,93				0,38	0,46			
8	20,68	20,19	21,18			0,42	0,43	0,39		
9	20,27	21,06	20,32			0,44	0,40	0,39		
10	20,54	20,12	20,49	20,57		0,38	0,41	0,39	0,47	
11	20,62	20,64	19,85	19,75		0,38	0,35	0,42	0,42	
12	19,7	20,55	19,47	20,49	21,21	0,39	0,40	0,42	0,43	0,39

Таблица 10.6. Производительность удаления из R-дерева для плотных и разреженных множеств данных

	Плотные данные					Разреженные данные				
M	*m* = 2	*m* = 3	*m* = 4	*m* = 5	*m* = 6	*m* = 2	*m* = 3	*m* = 4	*m* = 5	*m* = 6
4	19,56					4,08				
5	13,16					2,51				
6	11,25	18,23				1,76	4,81			
7	12,58	11,19				1,56	3,7			
8	8,31	9,87	15,09			1,39	2,81	4,96		
9	8,78	11,31	14,01			1,23	2,05	3,39		
10	12,45	8,45	9,59	18,34		1,08	1,80	3,07	5,43	
11	8,09	7,56	8,68	12,28		1,13	1,66	2,51	4,17	
12	8,91	8,25	11,26	14,8	15,82	1,04	1,52	2,18	3,14	5,91

Теперь мы зафиксируем значения $M=4$ и $m=2$ и вычислим производительность поиска и удаления с ростом n. В общем случае более высокие значения M более выгодны, когда имеется много удалений, так как это уменьшает количество значений, которые заново вставляются в R-дерево из-за не полностью заполненных узлов; но истинное поведение основывается на используемых данных и подходе к балансировке дерева, используемом при разбиении узлов. Результаты исследований приведены в табл. 10.7.

Таблица 10.7. Производительность поиска и удаления (в мс) для разреженного множества данных и удваиваемых значений *n*

n	Поиск	Удаление
128	0,033	0,135
256	0,060	0,162
512	0,108	0,262
1024	0,178	0,320
2048	0,333	0,424
4096	0,725	0,779
8192	1,487	1,306
16384	3,638	2,518
32768	7,965	3,980
65536	16,996	10,051
131072	33,985	15,115

11

Дополнительные категории алгоритмов

В предыдущих главах описаны алгоритмы, которые решают различные распространенные задачи. Очевидно, что в своей карьере программиста вам придется сталкиваться с задачами, которые не вписываются ни в одну распространенную категорию, поэтому в настоящей главе представлены четыре алгоритмических *подхода* к решению задач.

Еще одним отличием этой главы от других является ее ориентация на случайность и вероятность. Они использовались в предыдущих главах в ходе анализа поведения алгоритмов в среднем случае. Здесь же случайность может стать неотъемлемой частью алгоритма. Описываемые здесь вероятностные алгоритмы являются действительно интересными альтернативами детерминистическим алгоритмам. Запуск одного и того же такого алгоритма для одних и тех же входных данных в два разных момента времени могут дать очень разные ответы. Иногда мы будем терпимо относиться к неправильным ответам или просто утверждать, что решение не было найдено.

Вариации на тему алгоритмов

Приведенные ранее в книге алгоритмы решают экземпляры задач, давая точный ответ на последовательном, детерминированном компьютере. Интересно рассмотреть ослабление этих трех позиций.

Приближенные алгоритмы

Вместо поиска точного решения задачи принимаются и решения, близкие к нему, но не обязательно такие же хорошие, как точный ответ.

Параллельные алгоритмы

Не ограничиваясь последовательными вычислениями, создаем несколько вычислительных процессов для одновременной работы над экземплярами подзадач.

Вероятностные алгоритмы

Вместо вычисления одного и того же результата для экземпляра задачи используем рандомизированные вычисления для получения ответа. При неоднократном запуске полученные ответы часто сходятся к верному ответу.

Приближенные алгоритмы

Приближенные алгоритмы представляют собой компромисс между точностью и более эффективной работой. В качестве примера, в котором хватает "достаточно хорошего" ответа, рассмотрим *задачу о рюкзаке*, которая возникает в различных предметных областях. Цель заключается в том, чтобы определить, что следует сложить в рюкзак так, чтобы максимизировать стоимость всего рюкзака, но не превысить максимально допустимый вес W. Задача, известная как *задача о рюкзаке 0/1*, может быть решена с помощью динамического программирования. В ней вы можете упаковать только один экземпляр каждого предмета. Вариант под названием *неограниченная задача о рюкзаке* позволяет упаковать столько экземпляров конкретного предмета, сколько вы желаете. В обоих случаях алгоритм должен возвращать максимальное значение стоимости предметов с учетом наложенного ограничения на общий вес.

Рассмотрим множество из четырех предметов {4, 8, 9, 10}, в котором стоимость каждого предмета в долларах имеет то же значение, что и его вес в фунтах. Таким образом, первый предмет весит 4 фунта и стоит 4 доллара. Предположим, вы можете упаковать максимальный вес $W=33$ фунта.

Как вы помните, динамическое программирование записывает результаты меньших подзадач, чтобы избежать их повторного вычисления, и сочетает в себе решения этих меньших подзадач для решения поставленной. В табл. 11.1 записываются частичные результаты для рюкзака 0/1 и каждая запись $m[i][w]$ записывает максимальное значение, которое будет достигнуто, если первые i предметов (строки) имеют максимальный общий вес w (столбцы). Максимальное значение для данного веса W с использованием до четырех предметов, как показано в правом нижнем углу, составляет 31 доллар. В этом случае в рюкзак добавляется по одному экземпляру каждого предмета.

Таблица 11.1. Решение задачи о рюкзаке 0/1 для малого множества

...	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33
1	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4
2	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12
3	13	13	13	13	17	17	17	17	21	21	21	21	21	21	21	21	21	21	21	21	21
4	13	14	14	14	17	18	19	19	21	22	23	23	23	23	27	27	27	27	31	31	31

Для вариации неограниченной задачи о рюкзаке в табл. 11.2 записываются значения $m[w]$, которые представляют максимальное значение достигаемого веса w, если вы можете упаковывать в рюкзак любое количество экземпляров каждого предмета.

Максимальное значение для данного веса W, как показано в крайней справа записи, составляет 33 доллара. В этом случае в рюкзак уложены шесть предметов по 4 фунта и один 9-фунтовый.

Таблица 11.2. Решение неограниченной задачи о рюкзаке для малого множества

...	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33
...	13	14	14	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33

Задача о рюкзаке 0/1

Наилучший, средний, наихудший случаи: $O(n \cdot W)$

```
Рюкзак 0/1 (weights, values, W)
    n = количество предметов
    m = пустая матрица (n+1) x (W+1)                    ❶
    for i=1 to n do
        for j=0 to W do
            if weights[i-1] <= j then                   ❷
                remaining = j - weights[i-1]
                m[i][j] = max(m[i-1][j],m[i-1][remaining]+values[i-1])
            else
                m[i][j] = m[i-1][j]                     ❸
    return m[n][W]                                      ❹
end

Рюкзак неограниченный (weights, values, W)
    n = количество предметов
    m = пустой вектор длиной (W+1)                      ❺
    for j=1 to W+1 do
        best = m[j-1]
        for i=0 to n-1 do
            remaining = j - weights[i]
            if remaining >= 0 и m[remaining]+values[i] > best then
                best = m[remaining] + values[i]
            m[j] = best
    return m[W]
```

❶ m[i][j] записывает максимальное значение с использованием первых i предметов без превышения веса j.

❷ Можем ли мы увеличить стоимость путем добавления предмета i-1 к предыдущему решению с весом (j - вес этого предмета)?

❸ Предмет i-1 превышает допустимый вес и потому не может улучшить решение.

❹ Возврат наилучшего вычисленного значения.

❺ Для неограниченной вариации задачи m[j] записывает максимальное значение без превышения веса j.

Входные и выходные данные алгоритма

В задаче задаются набор предметов (каждый с целочисленным весом и стоимостью) и максимальный вес W. Задача заключается в том, чтобы определить, какие предметы следует упаковать в рюкзак так, чтобы общий вес не превышал W, а общая стоимость упакованных предметов была как можно большей.

Контекст применения алгоритма

Это тип задачи распределения ограниченных ресурсов, которая распространена в компьютерных науках, математике и экономике. Она активно изучалась более века и имеет множество вариаций. Часто нужно знать фактический набор предметов, а не только максимальную стоимость, поэтому решение задачи должно также возвращать выбранные для размещения в рюкзаке предметы.

Реализация алгоритма

Мы используем динамическое программирование, которое работает путем решения и сохранения полученных результатов для более простых подзадач. Для рюкзака 0/1 двумерная матрица $m[i][j]$ записывает максимальную стоимость, получаемую с помощью первых i предметов без превышения веса j. Структура решения в примере 11.1 соответствует ожидаемому двойному циклу динамического программирования.

Пример 11.1. Реализация задачи о рюкзаке 0/1 на языке программирования Python

```
class Item:
  def __init__(self, value, weight):
    """Создание предмета с данной стоимостью и весом."""
    self.value = value
    self.weight = weight

def knapsack_01 (items, W):
    """
    Вычисление решения задачи о рюкзаке 0/1 (разрешен только
    один предмет каждого вида) для набора предметов с
    соответствующими весами и стоимостями. Возвращает общий вес
    и выбор предметов.
    """
    n = len(items)
    m = [None] * (n+1)
    for i in range(n+1):
        m[i] = [0] * (W+1)

    for i in range(1,n+1):
        for j in range(W+1):
            if items[i-1].weight <= j:
```

```
                valueWithItem = m[i-1][j-items[i-1].weight]+
                                        items[i-1].value
                m[i][j] = max(m[i-1][j], valueWithItem)
            else:
                m[i][j] = m[i-1][j]

    selections = [0] * n
    i = n
    w = W
    while i > 0 and w >= 0:
        if m[i][w] != m[i-1][w]:
            selections[i-1] = 1
            w -= items[i-1].weight
        i -= 1
return (m[n][W], selections)
```

Этот код соответствует структуре метода динамического программирования, вычисляющего по очереди каждую подзадачу. Как только вложенные циклы `for` вычисляют максимальное значение `m[n][W]`, последующий цикл `while` показывает, как восстановить фактические выбранные предметы путем "прохода" по матрице `m`. Он начинается в нижнем правом углу `m[n][W]` и определяет, был ли выбран *i*-й предмет, на основании того, отличается ли `m[i][w]` от `m[i-1][w]`. Если это так, данный выбор записывается, после чего выполняется перемещение влево по `m` путем удаления веса *i*-го предмета, и так продолжается до тех пор, пока мы не попадем в первую строку (предметов больше нет) или левый столбец (больше нет веса); в противном случае рассматривается предыдущий предмет.

Неограниченная задача о рюкзаке использует одномерный вектор `m[j]` для записи максимальной стоимости, не превышающей вес *j*. Ее реализация на языке программирования Python показана в примере 11.2.

Пример 11.2. Реализация неограниченной задачи о рюкзаке на языке программирования Python

```
def knapsack_unbounded (items, W):
    """
    Вычисление решения неограниченной задачи о рюкзаке (разрешено
    любое количество каждого вида предметов) для набора предметов
    с соответствующими весами и стоимостями. Возвращает общий вес
    и выбор предметов.
    """
    n = len(items)
    progress = [0] * (W+1)
    progress[0] = -1
    m = [0] * (W + 1)
    for j in range(1, W+1):
        progress[j] = progress[j-1]
```

```
            best = m[j-1]
            for i in range(n):
                remaining = j - items[i].weight
                if remaining>=0 and m[remaining]+items[i].value>best:
                    best = m[remaining] + items[i].value
                    progress[j] = i
                m[j] = best

    selections = [0] * n
    i = n
    w = W
    while w >= 0:
        choice = progress[w]
        if choice == -1:
            break
        selections[choice] += 1
        w -= items[progress[w]].weight
    return (m[W], selections)
```

Анализ алгоритма

Эти решения не являются решениями с производительностью $O(n)$, как можно было бы ожидать исходя из того, что они требуют общего времени, ограниченного величиной $c \cdot n$, где c — константа для достаточно больших n. Время работы зависит и от W. Таким образом, в обоих случаях решение является решением $O(n \cdot W)$, о чем ясно свидетельствуют вложенные циклы `for`. Восстановление выбора выполняется за время $O(n)$, так что оно не меняет общую производительность.

Почему это важно? В задаче о рюкзаке 0/1, когда W гораздо больше веса отдельных предметов, алгоритму приходится многократно выполнять итерации впустую, так как каждый предмет выбирается только один раз. Аналогичные недостатки есть и у решения неограниченной задачи о рюкзаке.

В 1957 году Джордж Данциг (George Dantzig) предложил приближенное решение для неограниченной задачи о рюкзаке, показанное в примере 11.3. В основе этого решения лежит интуитивное представление о том, что сначала вы должны поместить в рюкзак предметы с максимальным отношением стоимости к весу. Фактически этот подход гарантированно находит приближенное решение, которое не хуже чем половина максимального значения, получаемого путем динамического программирования. На практике результаты оказываются довольно близкими к реальной стоимости, а код выполняется заметно быстрее.

Пример 11.3. Реализация приближенного решения неограниченной задачи о рюкзаке на языке программирования Python

```
class ApproximateItem(Item):
    """
```

```
    Расширяет Item, сохраняя нормализованное значение и
    исходное положение предмета до сортировки.
    """
    def __init__(self, item, idx):
        Item.__init__(self, item.value, item.weight)
        self.normalizedValue = item.value/item.weight
        self.index = idx

def knapsack_approximate (items, W):
    """
    Вычисление приближенного решения задачи о рюкзаке
    с использованием подхода Данцига.
    """
    approxItems = []
    n = len(items)
    for idx in range(n):
        approxItems.append (ApproximateItem(items[idx], idx))
    approxItems.sort (key=lambda x:x.normalizedValue, reverse=True)

    selections = [0] * n
    w = W
    total = 0
    for idx in range(n):
        item = approxItems[idx]
        if w == 0:
            break
        # Количество, помещающееся в рюкзаке
        numAdd = w // item.weight
        if numAdd > 0:
            selections[item.index] += numAdd
            w -= numAdd * item.weight
            total += numAdd * item.value

    return (total, selections)
```

Данная реализация выполняет перебор предметов в порядке, обратном их нормализованному значению (т.е. отношению стоимости к весу). Производительность этого алгоритма — $O(n \cdot log\ n)$, потому что он должен выполнить предварительную сортировку предметов.

Возвращаясь к исходному набору из четырех элементов {4, 8, 9, 10} из предыдущего примера, заметим, что соотношение стоимости к весу равно 1 для каждого предмета, т.е. в соответствии с алгоритмом важность всех их "эквивалентна". Для заданного веса $W = 33$ приближенный алгоритм выбирает упаковку восьми экземпляров предмета весом 4 фунта, что приводит к общей стоимости 32 доллара. Интересно, что все три алгоритма получили разные значения стоимости при одинаковых заданных предметах и ограничении общего веса.

В приведенной далее таблице сравнивается производительность приближенного и точного решений неограниченной задачи о рюкзаке с ростом W. Все предметы в наборе размером $n = 53$ штук имеют разный вес, а стоимость каждого предмета устанавливается равной его весу, в диапазоне от 103 до 407. Как видно из таблицы, при удвоении веса W время выполнения неограниченной задачи о ранце также удваивается (что объясняется его производительностью $O(n \cdot W)$). Однако производительность приближенного алгоритма не меняется с ростом W, поскольку его производительность определяется только величиной $O(n \cdot log\ n)$.

Анализируя последние два столбца, вы можете увидеть, что для $W = 175$ приближенное решение дает 60% от точного ответа. С увеличением W приближенное решение становится все ближе к точному ответу. Кроме того, приближенный алгоритм почти в 1000 раз быстрее.

Таблица 11.3. Производительность разных решений задачи о рюкзаке

W	Время счета неограниченной задачи	Время счета приближенного решения	Точный ответ	Приближенный ответ
175	0,00256	0,00011	175	103
351	0,00628	0,00011	351	309
703	0,01610	0,00012	703	618
1407	0,03491	0,00012	1407	1339
2815	0,07320	0,00011	2815	2781
5631	0,14937	0,00012	5631	5562
11263	0,30195	0,00012	11263	11227
22527	0,60880	0,00013	22527	22454
45055	1,21654	0,00012	45055	45011

Параллельные алгоритмы

Параллельные алгоритмы используют преимущества существующих вычислительных ресурсов путем создания различных потоков выполнения и управления ими.

Алгоритм быстрой сортировки, представленный в главе 4, "Алгоритмы сортировки", может быть реализован на языке программирования Java так, как показано в примере 11.4, в предположении существования функции `partition` для разделения исходного массива на два подмассива на основе опорного значения. Как вы помните из главы 4, "Алгоритмы сортировки", значения слева от `pivotIndex` не превышают опорного значения, а значения справа от `pivotIndex` не меньше опорного.

Пример 11.4. Реализация быстрой сортировки на языке программирования Java

```
public class MultiThreadQuickSort<E extends Comparable<E>>
{
    final E[] ar;   /** Сортируемые элементы. */
```

```
    IPivotIndex pi; /** Функция разбиения.     */
    /** Построение экземпляра для решения. */

    public MultiThreadQuickSort(E ar[])
    {
        this.ar = ar;
    }

    /** Установка метода разделения. */
    public void setPivotMethod(IPivotIndex ipi)
    {
        this.pi = ipi;
    }

    /** Однопоточная сортировка ar[left,right]. */
    public void qsortSingle(int left, int right)
    {
        if (right <= left)
        {
            return;
        }

        int pivotIndex = pi.selectPivotIndex(ar, left, right);
        pivotIndex = partition(left, right, pivotIndex);
        qsortSingle(left, pivotIndex - 1);
        qsortSingle(pivotIndex + 1, right);
    }
}
```

Две подзадачи, `qsortSingle(left,pivotIndex-1)` и `qsortSingle(pivotIndex+1, right)`, являются независимыми и теоретически могут быть решены одновременно. Сразу же встает вопрос о том, как использовать несколько потоков для решения этой задачи. Просто запускать вспомогательный поток для каждого рекурсивного вызова нельзя, поскольку так будут быстро исчерпаны ресурсы операционной системы. Рассмотрим переписанную реализацию `qsort2`, показанную в примере 11.5.

Пример 11.5. Многопоточная реализация быстрой сортировки на языке программирования Java

```
/** Многопоточная сортировка ar[left,right]. */
void qsort2(int left, int right)
{
    if (right <= left)
    {
        return;
    }
```

```
    int pivotIndex = pi.selectPivotIndex(ar, left, right);
    pivotIndex = partition(left, right, pivotIndex);
    qsortThread(left, pivotIndex - 1);
    qsortThread(pivotIndex + 1, right);
}

/**
  * Запуск потока для сортировки ar[left,right] или использование
  * существующего потока, если размер задачи слишком велик или все
  * вспомогательные потоки используются.
*/
private void qsortThread(final int left, final int right)
{
    // Работают все вспомогательные потоки ИЛИ задача слишком
    // велика? Если так - продолжаем с использованием рекурсии.
    int n = right + 1 - left;

    if (helpersWorking == numThreads || n >= threshold)
    {
        qsort2(left, right);
    }
    else
    {
        // В противном случае завершаем в отдельном потоке.
        synchronized(helpRequestedMutex)
        {
            helpersWorking++;
        }
        new Thread()
        {
            public void run()
            {
                // Вызов однопоточной сортировки.
                qsortSingle(left, right);
                synchronized(helpRequestedMutex)
                {
                    helpersWorking--;
                }
            }
        }.start();
    }
}
```

Для каждой из двух подзадач `qsortThread` выполняется простая проверка, должен ли основной поток продолжать рекурсивный вызов функции `qsortThread`. Отдельный вспомогательный поток запускается для работы с подзадачей только в том случае, если поток доступен, а размер подзадачи меньше указанного порогового

значения. Эта логика применяется при сортировке как левого, так и правого подмассивов. Значение `threshold` вычисляется путем вызова `setThresholdRatio(r)`, который устанавливает пороговый размер подзадачи равным n/r, где n — количество сортируемых элементов. По умолчанию это отношение равно 5, что означает, что вспомогательный поток вызывается только для подзадач, которые меньше 20% первоначальной задачи.

Атрибут класса `helpersWorking` хранит количество активных вспомогательных потоков. Всякий раз при создании потока переменная `helpersWorking` увеличивается, а сам поток по завершении уменьшает это значение. Используя мьютекс `helpRequestedMutex` и возможность Java синхронизировать блоки кода для монопольного доступа, данная реализация обновляет переменную `helpersWorking` безопасно с точки зрения потоков. `qsort2` во вспомогательных потоках вызывает однопоточный метод `qsortSingle`. Это гарантирует, что запускать новые потоки может только основной поток.

В этой конструкции вспомогательные потоки не могут порождать дополнительные вспомогательные потоки. Если бы это было позволено, то первичные вспомогательные потоки необходимо было бы синхронизировать со вторичными, чтобы вторичные потоки могли начинаться только после того, как первичный вспомогательный поток правильно разделит массив.

На рис. 11.1 и 11.2 сравниваются однопоточная реализация сортировки случайных целых чисел из диапазона [0, 16777216] на Java и реализация на Java с одной вспомогательной функцией. Мы рассматривали несколько параметров.

Размер n сортируемого массива

Это значение находилось в диапазоне от 65 536 до 1 048 576.

Порог n/r

Этот порог определяет максимальный размер задачи, для которого создается вспомогательный поток. Мы экспериментировали со значениями r в диапазоне от 1 до 20 и со значением `MAXINT`, которое предотвращает использование вспомогательных потоков.

Количество доступных вспомогательных потоков

Мы экспериментировали с количеством вспомогательных потоков от 0 до 9.

Используемый метод разбиения

Мы испытывали как "выбор случайного элемента", так и "выбор крайнего справа элемента".

Таким образом, пространство параметров составляет около 2000 уникальных комбинаций этих параметров. В общем случае мы обнаружили при использовании генератора случайных чисел замедление производительности около 5% во всех

экспериментах, поэтому далее мы сосредоточились только на методе разделения по крайнему справа опорному элементу. Кроме того, наличие более одного вспомогательного потока не улучшало производительность быстрой сортировки, поэтому мы ограничились только одним вспомогательным потоком.

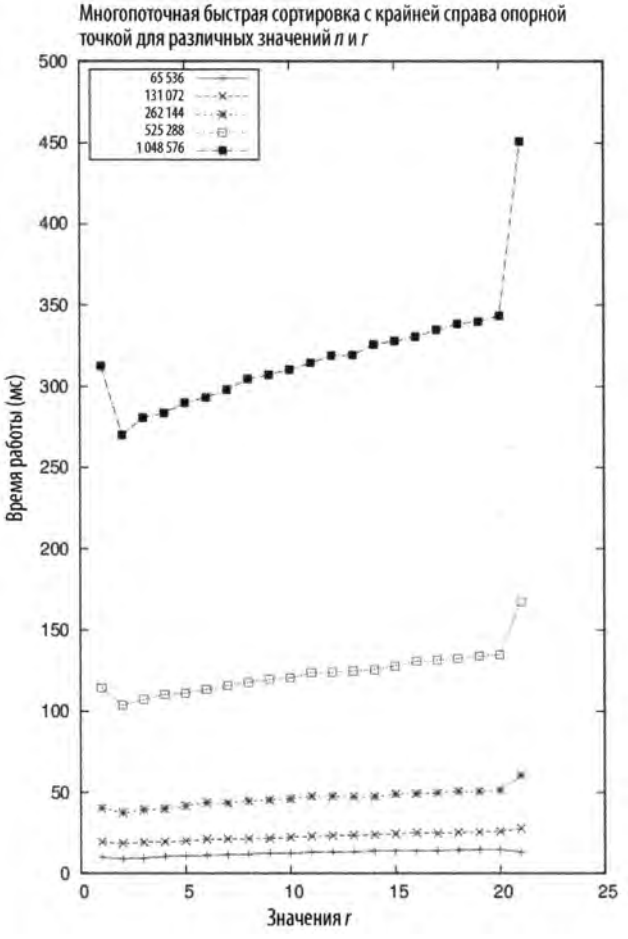


Рис. 11.1. Многопоточная быстрая сортировка при разных n и r

Рассматривая график слева направо, вы можете видеть, что первая точка ($r = 1$) показывает производительность, когда предпринимаются попытки немедленно начать использовать вспомогательный поток, а последняя точка ($r = 21$) показывает производительность, когда вспомогательный поток не используется вовсе. Вычисляя *коэффициент ускорения* для исходного времени T1 и меньшего времени T2, мы используем отношение T1/T2. Использование дополнительного потока дает коэффициент ускорения около 1,3 (табл. 11.4). Это достаточно неплохой результат для такого малого изменения кода.

Таблица 11.4. Ускорение при наличии одного вспомогательного потока (сравнение времени выполнения для `r` =1 и `r`=`MAXINT`

n	Ускорение многопоточной реализации по сравнению с однопоточной
65536	1,24
131072	1,37
262144	1,35
524288	1,37
1048576	1,31

Возвращаясь к рис. 11.1, можно увидеть, что наибольшее улучшение происходит вблизи $r=2$. Применение потоков имеет собственные накладные расходы, так что не следует автоматически создавать и запускать новый поток выполнения без определенных гарантий того, что основному потоку не придется ждать и быть *заблокированным* до тех пор, пока вспомогательный поток не завершит свое выполнение. Результаты экспериментов будут различаться в зависимости от используемой вычислительной платформы.

Во многих случаях ускорение зависит от количества имеющихся процессоров. Так, на рис. 11.2 представлены таблицы ускорения на двух разных вычислительных платформах — двух- и четырехъядерном процессорах. Каждая строка представляет потенциальное количество доступных потоков, в то время как каждый столбец представляет пороговое значение *r*. Общее количество сортируемых элементов установлено равным $n=1\,048\,576$. Результаты для четырехъядерного процессора демонстрируют эффективность использования нескольких потоков, показывая ускорение до полутора раз; но нельзя сказать то же о двухъядерных процессорах, производительность на которых увеличивается не более чем на 5%.

Исследование факторов ускорения параллельных алгоритмов показывает, что существуют определенные ограничения на то, какое количество дополнительных потоков будет реально способствовать повышению производительности реализации алгоритма. В данном случае многопоточная реализация быстрой сортировки достигает хорошего коэффициента ускорения, потому что отдельные подзадачи рекурсивной быстрой сортировки полностью независимы и не будет никаких конфликтов за использование общих ресурсов между несколькими потоками. Если другие задачи обладают такими же характеристиками, то они также должны получать выгоду от возможности воспользоваться многопоточностью.

Вероятностные алгоритмы

Вероятностный алгоритм использует поток случайных битов (т.е. случайные числа) в качестве части процесса вычисления ответа, так что вы получите разные результаты при разных выполнениях алгоритма для одного и того же экземпляра задачи. Часто таким образом получаются алгоритмы, которые выполняются быстрее любых текущих альтернатив.

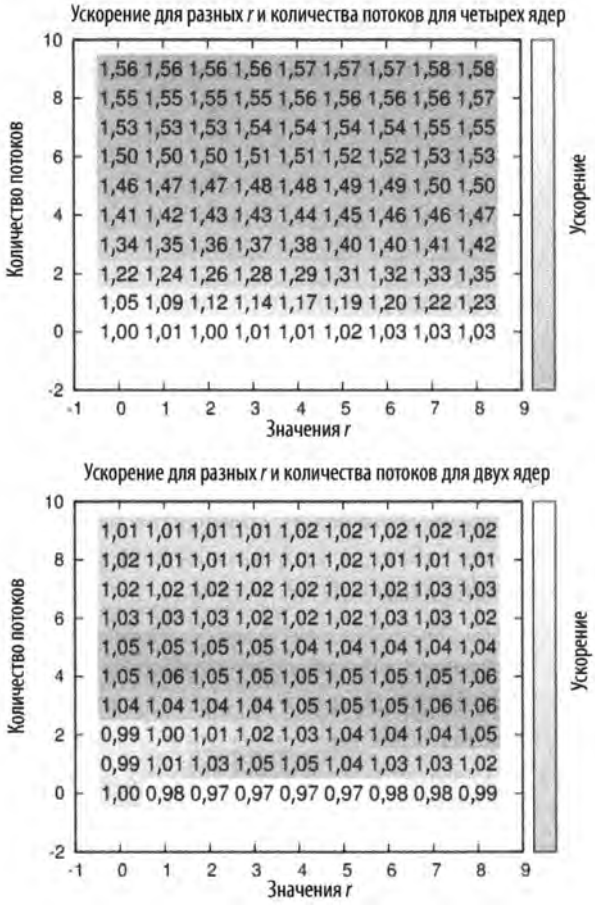


Рис. 11.2. Производительность многопоточной быстрой сортировки при фиксированном n и разных r и количестве потоков

Для практических целей следует знать, что потоки случайных битов весьма трудно генерировать на детерминированных компьютерах. Хотя мы можем генерировать потоки квазислучайных битов, которые практически ничем не отличаются от действительно случайных потоков, стоимость генерации этих потоков игнорировать не следует.

Оценка размера множества

В качестве примера ускорения, которое может быть получено при разрешении использовать вероятностные алгоритмы, предположим, что мы хотим оценить размер множества n отдельных объектов (т.е. оценить значение n, наблюдая отдельные элементы). Было бы просто подсчитать все объекты — за время $O(n)$. Очевидно,

что этот процесс гарантирует точный ответ. Но если неверная оценка значения n терпима, то ее можно вычислить быстрее с помощью алгоритма, описанного в примере 11.6. Этот алгоритм похож на эксперименты биологов, когда они помечают некоторых пойманных особей в популяции и возвращают их в место обитания, а затем через время вновь выполняют отлов. Подход заключается в использовании функции-генератора, которая возвращает случайный объект из "популяции".

Пример 11.6. Реализация вероятностного алгоритма подсчета

```
def computeK(generator):
    """
    Вычисление оценки количества элементов с использованием
    вероятностного алгоритма подсчета. Значение n — размера
    множества — неизвестно.
    """
    seen = set()
    while True:
        item = generator()
        if item in seen:
            k = len(seen)
            return 2.0*k*k/math.pi
        else:
            seen.add(item)
```

Начнем с некоторых интуитивных представлений. Мы должны иметь возможность выбрать случайные элементы из набора и пометить их как уже виденные. Так как мы предполагаем ограниченность множества, в какой-то момент мы должны встретить элемент, который уже видели раньше. Чем дольше мы не встречаем уже выбиравшийся ранее элемент, тем бо́льшим должен быть размер исходного множества. Такое поведение называется "выборка с возвращением" и ожидаемое число выборов k до появления уже выбиравшегося ранее элемента можно оценить как

$$k = \sqrt{\pi n/2}$$

Как только функция `generator` возвращает элемент, который мы уже видели (что она обязательно должна сделать, так как множество конечно), цикл `while` завершается — это происходит через некоторое количество выборов элемента k. После того как вычисляется k, показанная выше формула используется для вычисления приближения значения n. Понятно, что алгоритм никогда не сможет дать точное значение n просто потому, что $2k^2/\pi$ никогда не будет целым числом, но это вычисление представляет собой несмещенную оценку n.

В табл. 11.5 показан пример выполнения алгоритма. Мы многократно записывали результаты выполнения вычислений для ряда испытаний — $t = \{32, 64, 128, 256, 512\}$. Из этих испытаний были удалены самые низкие и высокие оценки, и в каждом столбце отображается среднее значение оставшихся t–2 испытаний.

Таблица 11.5. Результаты вероятностного алгоритма подсчета с ростом количества попыток

n	Среднее из 30 испытаний	Среднее из 62 испытаний	Среднее из 126 испытаний	Среднее из 254 испытаний	Среднее из 510 испытаний
1024	1144	1065	1205	1084	1290
2048	2247	1794	2708	2843	2543
4096	3789	4297	5657	5384	5475
8192	9507	10369	10632	10517	9687
16384	20776	18154	15617	20527	21812
32768	39363	29553	40538	36094	39542
65536	79889	81576	76091	85034	83102
131072	145664	187087	146191	173928	174630
262144	393848	297303	336110	368821	336936
524288	766044	509939	598978	667082	718883
1048576	1366027	1242640	1455569	1364828	1256300

В силу случайного характера испытаний не гарантируется, что окончательный точный результат можно достичь путем простого усреднения все большего количества независимых случайных испытаний. Увеличение количества испытаний мало повышает точность, но в данном случае самое главное то, что вероятностный алгоритм эффективно возвращает оценку на основании выборки небольшого размера.

Оценка размера дерева поиска

Математики уже давно изучили *задачу о восьми ферзях*, в которой первоначально спрашивалось, можно ли разместить восемь ферзей на шахматной доске так, чтобы никакие два из них не угрожали один другому. Эта задача была расширена до более общей задачи подсчета количества уникальных решений размещения таким образом n ферзей на шахматной доске размером $n \times n$. Способ математического вычисления ответа пока не получен; но вы можете написать программу, которая проверяет все возможные конфигурации доски для выяснения ответа. В табл. 11.6 содержатся некоторые вычисленные значения, взятые из энциклопедии целочисленных последовательностей (`https://oeis.org/A000170`). Как видите, количество решений растет очень быстро.

Чтобы подсчитать количество точных решений задачи о четырех ферзях, мы расширяем дерево поиска (показанное на рис. 11.3), основываясь на том факте, что каждое решение будет иметь ферзя на каждой горизонтали. Исчерпывающее построение дерева поиска позволяет нам увидеть, что имеется всего два решения задачи о четырех ферзях. Попытаться вычислить количество решений задачи о 19 ферзях гораздо сложнее, потому что на 19-м уровне это дерево поиска содержит 4968057848 узлов. Генерировать каждое решение просто слишком дорого.

Таблица 11.6. Известные количества решений задачи о ферзях и наши вычисленные оценки

n	Точное количество решений	Оценка по T = 1024 испытаниям	Оценка по T = 8192 испытаниям	Оценка по T = 65536 испытаниям
1	1	1	1	1
2	0	0	0	0
3	0	0	0	0
4	2	2	2	2
5	10	10	10	10
6	4	5	4	4
7	40	41	39	40
8	92	88	87	93
9	352	357	338	351
10	724	729	694	718
11	2680	2473	2499	2600
12	14200	12606	14656	13905
13	73712	68580	62140	71678
14	365596	266618	391392	372699
15	2279184	1786570	2168273	2289607
16	14772512	12600153	13210175	15020881
17	95815104	79531007	75677252	101664299
18	666090624	713470160	582980339	623574560
19	4968057848	4931587745	4642673268	4931598683
20	39029188884	17864106169	38470127712	37861260851

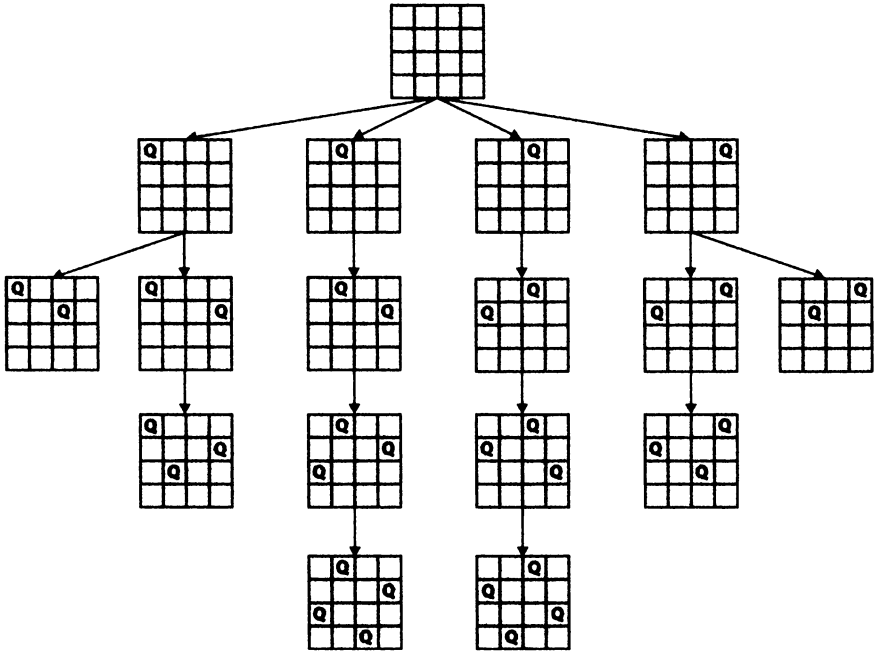


Рис. 11.3. Окончательное решение задачи о четырех ферзях

Однако что если нас интересует только приближенное количество решений, или, другими словами, количество потенциальных состояний доски на уровне *n*? Кнут [33] разработал новый альтернативный подход для оценки размера и формы дерева поиска. Его метод соответствует случайному блужданию вниз по дереву поиска. Для краткости мы применим этот подход для задачи четырех ферзей, но он так же легко может применяться и для оценки числа решений задачи о 19 ферзях. Вместо подсчета всех возможных решений создадим одно состояние с *n* ферзями и оценим общее их число, подсчитывая потенциальные состояния доски, *которые не следуют из него*, полагая, что каждое из этих направлений будет одинаково производительным.

На рис. 11.4 показан подход Кнута для шахматной доски размером 4 × 4. Каждое из состояний доски имеет связанную с ним оценку (показанную в виде числа в маленьком кружке) числа состояний доски во всем дереве *на этом уровне*. Начиная с корня дерева поиска (где ферзей на доске нет), каждое состояние доски расширяется на новый уровень на основе количества дочерних узлов. Проведем большое число случайных блужданий по этому дереву поиска, которое никогда не будет построено полностью во время этого процесса. В ходе каждого блуждания мы будем выбирать ходы случайным образом, пока не будет достигнуто решение или пока не останется доступных ходов. Усредняя число решений, возвращаемых каждым случайным блужданием, мы можем приблизительно оценить фактическое количество состояний в дереве. Выполним два возможных блуждания, начиная с корневого узла на уровне 0.

- Выберем крайнее слева состояние доски на первом уровне. Поскольку у корня имеется четыре дочерних узла, наша наилучшая оценка общего количества состояний на первом уровне — 4. Теперь вновь выберем крайний слева дочерний узел на уровне 2. С его точки зрения, в предположении, что каждый из прочих трех дочерних по отношению к корневому узлов продуктивен в той же степени, общее количество состояний доски на втором уровне можно оценить как равное 4·2 = 8. Однако из этого состояния никаких дальнейших ходов нет, так что остается предположить, что что на третьем уровне количество состояний равно нулю, и поиск прекращается без найденного решения.
- Выберем второе слева состояние доски на первом уровне. Наилучшая оценка для количества состояний на первом уровне — 4. Теперь на каждом из последующих уровней имеется только одно корректное состояние доски, что приводит нас к оценке 4·1 = 4 для числа состояний доски на каждом последующем уровне, в предположении, что все прочие исходные пути столь же продуктивны. По достижении наинизшего уровня мы оцениваем общее количество решений как равное четырем.

Ни одна из этих оценок не является правильной, что характерно для этого подхода — когда различные блуждания ведут к данным, заниженным и завышенным по отношению к точным. Однако если мы проводим большое количество случайных

блужданий, то среднее значение для этих оценок будет сходиться к точному значению. Каждая подобная оценка может быть быстро вычислена, и, таким образом, близкая к точной усредненная оценка может быть получена очень быстро.

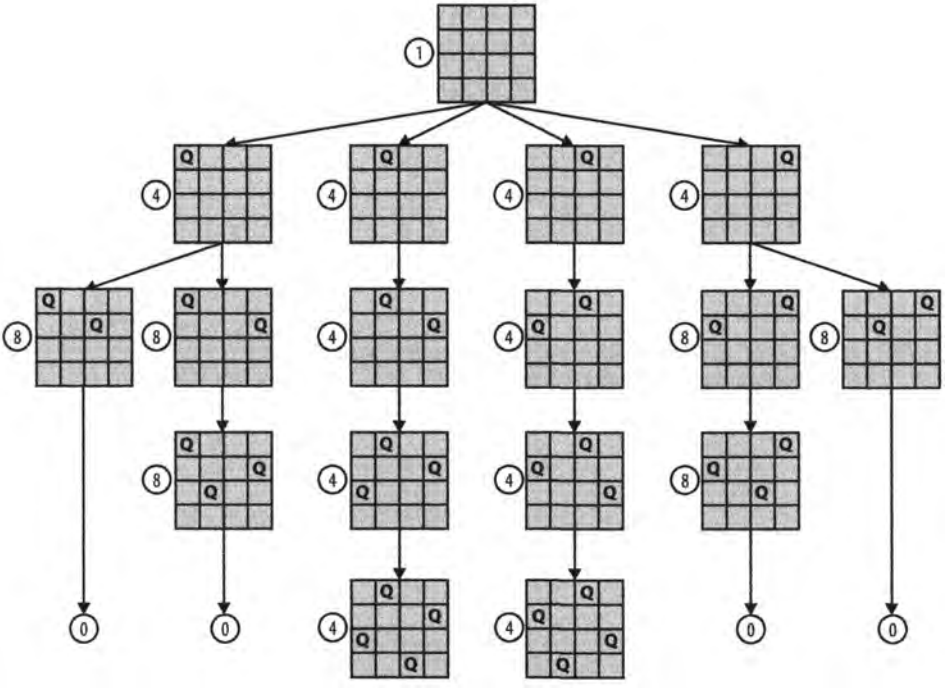


Рис. 11.4. Оценка количества решений задачи о четырех ферзях

Давайте вернемся к табл. 11.6 и рассмотрим результаты, полученные нашей реализацией для 1 024, 8 192 и 65 536 испытаний. Информация о времени работы не показана, потому что все результаты вычислялись менее чем за минуту. Отклонение окончательной оценки для задачи о 19 ферзях при 65 536 испытаниях от точного ответа составляет 3%. Оценки для всех задач при $T = 65\,536$ находятся в пределах 5,8% от точного результата. Этот алгоритм обладает тем желательным свойством, что вычисленная оценка является тем более точной, чем больше выполнено случайных испытаний. В примере 11.7 показана реализация получения одной оценки задачи о *n* ферзях на языке программирования Java.

Пример 11.7. Реализация рандомизированной оценки Кнута для задачи о n ферзях

```
/**
 * Для доски n × n сохраняет до n не бьющих один другого ферзей и
 * выполняет поиск вдоль путей случайных блужданий. Предполагается,
 * что ферзи добавляются построчно начиная с нулевой горизонтали.
*/
public class Board
```

```
{
    boolean [][] board;     /** Доска.        */
    final int n;            /** Размер доски. */

    /** Временное хранилище для последних корректных позиций. */
    ArrayList<Integer> nextValidRowPositions = new ArrayList<Integer>();
    public Board(int n)
    {
        board = new boolean[n][n];
        this.n = n;
    }

    /** Начинаем со строки и работаем вверх, проверяя корректность. */
    private boolean valid(int row, int col)
    {
        // Есть ли другой ферзь на той же вертикали,
        // горизонтали или диагоналях?
        int d = 0;

        while (++d <= row)
        {
            if (board[row - d][col])
            {
                return false;
            }

            if (col >= d && board[row - d][col - d])
            {
                return false;
            }

            if (col + d < n && board[row - d][col + d])
            {
                return false;
            }
        }

        return true; // OK
    }

    /**
     * Выясняем, сколько корректных дочерних состояний
     * обнаруживается при попытках добавить ферзя на данную
     * горизонталь. Возвращает число от 0 до n.
    */
    public int numChildren(int row)
    {
        int count = 0;
```

```
        nextValidRowPositions.clear();

        for (int i = 0; i < n; i++)
        {
            board[row][i] = true;

            if (valid(row, i))
            {
                count++;
                nextValidRowPositions.add(i);
            }

            board[row][i] = false;
        }

        return count;
    }

    /** Если нет доступной позиции в горизонтали, возврат false. */
    public boolean randomNextBoard(int r)
    {
        int sz = nextValidRowPositions.size();

        if (sz == 0)
        {
            return false;
        }

        // Выбор одной позиции случайным образом
        int c = (int)(Math.random() * sz);
        board[r][nextValidRowPositions.get(c)] = true;
        return true;
    }
}

public class SingleQuery
{
    /** Генерация таблицы. */
    public static void main(String []args)
    {
        for (int i = 0; i < 100; i++)
        {
            System.out.println(i + ": " + estimate(19));
        }
    }

    public static long estimate(int n)
    {
```

```
        Board b = new Board(n);
        int r = 0;
        long lastEstimate = 1;

        while (r < n)
        {
            int numChildren = b.numChildren(r);

            // Далее идти некуда — решение не найдено.
            if (!b.randomNextBoard(r))
            {
                lastEstimate = 0;
                break;
            }

            // Вычисление оценки
            lastEstimate = lastEstimate * numChildren;
            r++;
        }

        return lastEstimate;
    }
}
```

12

Эпилог: алгоритмические принципы

Хотя мы добрались до конца этой книги, конца у информации, посвященной алгоритмам, нет. Нет конца и у разновидностей задач, к которым можно применить представленные в этой книге методы.

Теперь мы имеем возможность сделать шаг назад и пересмотреть почти три десятка алгоритмов, которые подробно и с примерами описаны в этой книге. Чтобы показать широту охваченного материала, мы теперь подытожим алгоритмические принципы, представленные в этой книге. При этом мы можем продемонстрировать сходство различных алгоритмов, разработанных для решения разных задач. Вместо того чтобы просто подытожить каждую из предыдущих глав, мы завершим книгу, сосредоточив, в первую очередь, свое внимание на ключевых принципах, которые сыграли важную роль при разработке этих алгоритмов. Мы также воспользуемся возможностью обобщить концепции, используемые каждым алгоритмом.

Используемые данные

В этой книге был рассмотрен целый ряд распространенных действий, которые может понадобиться выполнить для получения определенных данных. Может потребоваться сортировка данных в соответствии с некоторым критерием. Может потребоваться поиск определенной информации в имеющихся данных. Ваши данные могут быть доступны с произвольным доступом (когда можно получить любую часть информации в любой момент времени), а могут быть доступны последовательно посредством итераторов (когда доступ выполняется последовательно, по одному элементу за раз). Без специальных знаний о данных можно рекомендовать алгоритмы лишь в наиболее общем виде.

Часто свойства входных данных имеют огромное влияние на выбор и работу алгоритмов. В главе 9, "Вычислительная геометрия", многие особые случаи могут быть устранены, просто если известно, что вычисляются пересечения отрезков, среди которых нет вертикальных. Аналогично упрощается вычисление диаграммы Вороного, если нет двух точек с одной и той же координатой x или y. Алгоритм Дейкстры из главы 6, "Алгоритмы на графах", будет работать вечно, если существует цикл, сумма весов ребер которого отрицательна. Убедитесь, что вы понимаете все допущения, на которых строится алгоритм, и особые случаи, которые при этом могут иметь место.

Как мы уже говорили, нет единого алгоритма, который последовательно обеспечивал бы наилучшую производительность при всех обстоятельствах. По мере получения информации о входных данных выбирайте тот алгоритм, который в наибольшей степени подходит для них. В табл. 12.1 кратко подытожена информация об алгоритмах сортировки, представленных в главе 4, "Алгоритмы сортировки". Естественным решением будет сосредоточиться на наихудшем случае для каждого алгоритма, но при этом следует уделить внимание и *концепциям*, с которыми приходится сталкиваться при реализации и использовании этих алгоритмов.

Таблица 12.1. Алгоритмы сортировки

Алгоритм	Наилучший случай	Средний случай	Наихудший случай	Концепции	Страница книги
Блочная сортировка	n	n	n	Хеширование	101
Пирамидальная сортировка	$n \cdot \log n$	$n \cdot \log n$	$n \cdot \log n$	Рекурсия, бинарная пирамида	87
Сортировка вставками	n	n^2	n^2	Жадный алгоритм	81
Сортировка слиянием	$n \cdot \log n$	$n \cdot \log n$	$n \cdot \log n$	Рекурсия, "разделяй и властвуй"	109
Быстрая сортировка	$n \cdot \log n$	$n \cdot \log n$	n^2	Рекурсия, "разделяй и властвуй"	95
Сортировка выбором	n^2	n^2	n^2	Жадный алгоритм	86

Разложение задачи на задачи меньшего размера

При разработке эффективного алгоритма для решения задачи бывает полезно разбить задачу на две (или более) меньшие подзадачи. Недаром быстрая сортировка остается одним из самых популярных алгоритмов сортировки. Даже при наличии точно документированных особых случаев, которые вызывают проблемы, быстрая сортировка предлагает наилучший средний случай для сортировки больших массивов информации. В действительности сама концепция алгоритма $O(n \cdot \log n)$ основана

на способности а) разделения задачи размером n на две подзадачи размером около $n/2$ и б) объединения решений двух подзадач в решение задачи. Чтобы разработать алгоритм со временем работы $O(n \cdot log\ n)$, указанные шаги должны выполняться за время $O(n)$.

Алгоритм быстрой сортировки был первым алгоритмом сортировки "на месте", без привлечения дополнительной памяти, который демонстрировал производительность $O(n \cdot log\ n)$. Его эффективность обусловлена (почти парадоксальным) подходом, состоящим в разделении задачи на две половины, каждая из которых может быть рекурсивно отсортирована путем применения того же алгоритма к подзадачам меньшего размера.

Задачи часто можно довольно просто сократить в два раза, что приводит к впечатляющим показателям производительности. Вспомните, как бинарный поиск преобразует задачу размером n в задачу размером $n/2$. Бинарный поиск использует преимущества повторяющегося характера задачи поиска для разработки для нее рекурсивного решения.

Иногда задача может быть решена путем разделения на две подзадачи без привлечения рекурсии. Алгоритм сканирования выпуклой оболочки генерирует окончательную выпуклую оболочку путем создания и объединения двух частичных оболочек (верхней и нижней).

Иногда проблема может быть разбита на ряд других (казалось бы, несвязанных) задач меньшего размера для тех же входных данных. Алгоритм Форда–Фалкерсона вычисляет максимальный поток в транспортной сети, многократно находя увеличивающий путь, к которому можно добавить поток. В конце концов увеличивающие пути больше не смогут быть построены и исходная задача будет решена. Сортировка выбором многократно находит максимальное значение в массиве и обменивает его с крайним справа элементом в массиве; после завершения n итераций массив оказывается в отсортированном состоянии. Аналогично пирамидальная сортировка многократно обменивает наибольший элемент в пирамиде с его правильным местоположением в массиве.

Заметим, что динамическое программирование разлагает задачи на меньшие, но общее поведение алгоритма при этом обычно представляет собой $O(n^2)$ или $O(n^3)$, потому что меньшие задачи, как правило, меньше исходных только на единицу размера, а не вполовину.

В табл. 12.2 сравниваются алгоритмы поиска, обсуждавшиеся в главе 5, "Поиск". Эти алгоритмы используют различные подходы для получения ответа на основополагающий вопрос о членстве во множестве. Анализируя их производительность, мы использовали методику *амортизированной* стоимости серии операций, что позволяет точно характеризовать *среднюю* производительность для данного случайного поискового запроса.

Таблица 12.2. Алгоритмы поиска

Алгоритм	Наилучший случай	Средний случай	Наихудший случай	Концепция	Страница книги
Бинарное дерево поиска AVL	1	*log n*	*log n*	Сбалансированное бинарное дерево	154
Последовательный поиск	1	*n*	*n*	Метод грубой силы	120
Бинарный поиск	1	*log n*	*log n*	"Разделяй и властвуй"	124
Фильтр Блума	*k*	*k*	*k*	Ложный положительный результат	146
Поиск на основе хеша	1	1	*n*	Хеширование	129
Бинарное дерево поиска	1	*log n*	*n*	Бинарное дерево	150

Выбор правильной структуры данных

Говорят, однажды знаменитый разработчик алгоритмов Роберт Таржан (Robert Tarjan) сказал, что любая задача может быть решена за время $O(n \cdot log\ n)$ — нужно только правильно выбрать структуру данных. Многие алгоритмы нуждаются в использовании очереди с приоритетами для хранения состояния или будущих вычислений. Одним из наиболее распространенных способов реализации очереди с приоритетами является бинарная пирамида, обеспечивающая производительность $O(log\ n)$ удаления элемента с наиболее низким приоритетом из очереди с приоритетами. Однако бинарная пирамида не позволяет быстро определить, содержится ли в ней указанный элемент. Мы обсуждали этот момент, рассматривая алгоритм LineSweep в главе 9, "Вычислительная геометрия". Этот алгоритм может обеспечить производительность $O(n \cdot log\ n)$, поскольку использует для реализации очереди с приоритетами усовершенствованное бинарное дерево и по-прежнему обеспечивает производительность $O(log\ n)$ удаления минимального элемента. Еще одна формулировка данного принципа — избегайте выбора неуместных структур данных, которые могут не позволить алгоритму достичь наилучшей производительности.

В главе 6 мы показали, когда следует использовать для представления графа список смежности или матрицу смежности — с учетом того, разреженный или плотный этот граф. Это решение само по себе оказывает очень большое влияние на производительность алгоритмов для работы с графами. В табл. 12.3 перечислены алгоритмы на графах, рассмотренные в главе 6, "Алгоритмы на графах".

При работе со сложными n-мерными данными нужны более сложные рекурсивные структуры для хранения данных. В главе 10, "Пространственные древовидные структуры", описываются структуры сложных *пространственных деревьев*

для эффективной поддержки стандартных поисковых запросов, а также более сложных запросов диапазона. Эти структуры были тщательно разработаны путем расширения *бинарных деревьев* — базовых рекурсивных структур данных в информатике.

Таблица 12.3. Алгоритмы на графах

Алгоритм	Наилучший случай	Средний случай	Наихудший случай	Концепции	Страница книги
Беллмана–Форда	$V \cdot E$	$V \cdot E$	$V \cdot E$	Взвешенный ориентированный граф	188
Поиск в ширину	$V+E$	$V+E$	$V+E$	Граф, очередь	175
Поиск в глубину	$V+E$	$V+E$	$V+E$	Граф, рекурсия, откат	169
Дейкстры с очередью с приоритетами	$(V+E)\log V$	$(V+E)\log V$	$(V+E)\log V$	Взвешенный ориентированный граф, очередь с приоритетами	181
Дейкстры для плотных графов	V^2+E	V^2+E	V^2+E	Взвешенный ориентированный граф	185
Флойда–Уоршелла	V^3	V^3	V^3	Динамическое программирование, взвешенный ориентированный граф	193
Прима	$(V+E)\log V$	$(V+E)\log V$	$(V+E)\log V$	Взвешенный ориентированный граф, бинарная пирамида, очередь с приоритетами, жадный алгоритм	198

Пространственно-временной компромисс

Многие из вычислений, выполняемых алгоритмами, оптимизированы путем хранения информации, которая отражает результаты предыдущих вычислений. Алгоритм Прима для вычисления минимального остовного дерева графа использует очередь с приоритетами для хранения непосещенных вершин в порядке их кратчайшего расстояния до начальной вершины *s*. Во время ключевого шага алгоритма мы должны определить, посещалась ли уже данная вершина. Поскольку реализация очереди с приоритетами на основе бинарной пирамиды не обеспечивает такую операцию, состояние каждой вершины хранится в отдельном булевом массиве. В том же алгоритме другой массив хранит вычисляемые расстояния, чтобы избежать необходимости повторного поиска в очереди с приоритетами. Это дополнительное пространство для хранения данных размером порядка $O(n)$ требуется для обеспечения эффективной реализации алгоритма. В большинстве случаев, если накладные расходы памяти не превышают $O(n)$, алгоритм можно безопасно использовать.

Иногда можно кешировать полностью все вычисления, так что их больше не придется пересчитывать. В главе 6, "Алгоритмы на графах", мы обсуждали, как для повышения производительности хеш-функция класса `java.lang.String` сохраняет вычисленное хеш-значение.

Иногда по своей природе входное множество данных требует большого количества памяти, например таковы плотные графы, описанные в главе 6, "Алгоритмы на графах". С помощью двумерной матрицы для хранения информации о ребрах (вместо простых списков смежности) ряд алгоритмов демонстрирует вполне разумную производительность. Можно также отметить, что в случае неориентированных графов алгоритмы можно упростить за счет удвоения используемой памяти для хранения информации о ребрах как `edgeInfo[i][j]`, так и `edgeInfo[j][i]`. Можно устранить эту дополнительную информацию, если всегда запрашивать только `edgeInfo[i][j]` при $i \leq j$, но это усложнит любой алгоритм, которому требуется знать, существует ли ребро (i,j).

Иногда алгоритм не может работать без более высокого потребления памяти. Блочная сортировка в состоянии выполнить сортировку равномерно распределенного входного множества за линейное время при использовании $O(n)$ дополнительной памяти. Учитывая, что современные компьютеры часто имеют достаточно большие объемы памяти, можно прибегать к блочной сортировке для равномерно распределенных данных, несмотря на высокие требования этого алгоритма к памяти.

В табл. 12.4 приведены алгоритмы с использованием пространственных деревьев, обсуждавшиеся в главе 10, "Пространственные древовидные структуры".

Таблица 12.4. Алгоритмы с пространственными деревьями

Алгоритм	Наилучший случай	Средний случай	Наихудший случай	Концепции	Страница книги
Запрос ближайшего соседа	*log n*	*log n*	*n*	*k-d*-дерево, рекурсия	338
Дерево квадрантов	*log n*	*log n*	*log n*	Дерево квадрантов	335
Запросы диапазона	$n^{1-1/d}+r$	$n^{1-1/d}+r$	*n*	*k-d*-дерево, рекурсия	348
R-дерево	*log n*	*log n*	*log n*	R-дерево	336

Построение поиска

Пионеры в области искусственного интеллекта часто характеризуются как ученые, которые пытаются решить задачи, для которых не существует известного решения. Одним из наиболее распространенных подходов к решению задач в этой области было превращение такой задачи в задачу поиска в очень большом графе. Мы посвятили этому подходу целую главу, поскольку он является важным общим методом решения многочисленных задач. Но будьте осторожны и применяйте его только

тогда, когда иной вычислительной альтернативы нет! Да, подход на основе поиска пути можно использовать для обнаружения последовательности перестановок элементов, которая начинается с неотсортированного массива (начальный узел) и приводит в отсортированный массив (целевой узел), но вы не должны использовать такой алгоритм с экспоненциальным поведением, поскольку для сортировки данных существуют многочисленные алгоритмы с производительностью $O(n \cdot log\ n)$.

В табл. 12.5 показаны алгоритмы поиска путей, которые мы рассматривали в главе 7, "Поиск путей в ИИ". Все они демонстрируют экспоненциальную производительность, но по-прежнему являются предпочтительным подходом к реализации интеллектуальных игровых программ. Во многом эти алгоритмы добиваются успеха в поиске решения из-за сложных эвристик, которые действительно делают процесс поиска более интеллектуальным.

Таблица 12.5. Поиск путей в области ИИ

Алгоритм	Наилучший случай	Средний случай	Наихудший случай	Концепции	Страница книги
Поиск в глубину	$b \cdot d$	b^d	b^d	Стек, множество, откат	232
Поиск в ширину	b^d	b^d	b^d	Очередь, множество	238
A*Search	$b \cdot d$	b^d	b^d	Очередь с приоритетами, множество, эвристика	242
Minimax	b^{ply}	b^{ply}	b^{ply}	Рекурсия, откат, грубая сила	211
NegMax	b^{ply}	b^{ply}	b^{ply}	Рекурсия, откат, грубая сила	217
AlphaBeta	$b^{ply/2}$	$b^{ply/2}$	b^{ply}	Рекурсия, откат, эвристика	221

Приведение задачи к другой задаче

Приведение задач является фундаментальным подходом, применяемым учеными и математиками для решения задач. В качестве простого примера предположим, что вам нужно найти четвертый по величине элемент в списке. Вместо разработки кода специального назначения можно использовать любой алгоритм сортировки для сортировки списка, а затем вернуть четвертый элемент из отсортированного списка. Используя этот подход, вы определяете алгоритм, время работы которого равно $O(n \cdot log\ n)$, хотя это и не самый эффективный способ решения данной задачи.

При использовании алгоритма выметания Форчуна для вычисления диаграммы Вороного выпуклую оболочку можно легко вычислить, находя те точки, у многоугольников диаграммы Вороного которых есть общие бесконечные. В этом отношении алгоритм вычисляет больше информации, чем необходимо, но выходные данные

алгоритма можно использовать для целого ряда интересных задач, таких как плоская триангуляция точек коллекции.

В главе 8, "Алгоритмы транспортных сетей", представлен ряд связанных задач, для которых, как представляется, не существует простого способа связать их вместе. Можно привести все эти задачи к задаче линейного программирования и использовать имеющиеся программные пакеты (такие, как Maple) для вычисления решений. Однако приведения оказываются достаточно сложными, так что конкретные задачи можно решить эффективнее (зачастую — значительно эффективнее) с помощью семейства алгоритмов Форда–Фалкерсона. Мы показали в главе 8, "Алгоритмы транспортных сетей", как решить одну из задач, а именно — задачу вычисления максимального потока с минимальной стоимостью в транспортной сети. Имея этот алгоритм, мы смогли тут же решить пять других задач. В табл. 12.6 показаны алгоритмы для работы с транспортной сетью, описанные в главе 8, "Алгоритмы транспортных сетей".

Таблица 12.6. Алгоритмы транспортной сети

Алгоритм	Наилучший случай	Средний случай	Наихудший случай	Концепции	Страница книги
Форда–алкерсона	$E \cdot mf$	$E \cdot mf$	$E \cdot mf$	Взвешенный ориентированный граф, жадный алгоритм	260
Эдмондса–Карпа	$V \cdot E^2$	$V \cdot E^2$	$V \cdot E^2$	Взвешенный ориентированный граф, жадный алгоритм	260

Тестировать алгоритмы труднее, чем писать

Поскольку алгоритмы, которые мы описываем в книге, большей частью детерминированные (за исключением алгоритмов из главы 11, "Дополнительные категории алгоритмов"), было довольно легко разрабатывать тестовые примеры, чтобы убедиться, что они ведут себя должным образом. В главе 7, "Поиск путей в ИИ", мы начали сталкиваться с трудностями, поскольку использовали алгоритмы поиска пути для обнаружения потенциальных решений, которые мы не знали заранее. Например, хотя очень просто написать тестовые примеры для выяснения, работает ли эвристика `GoodEvaluator` должным образом для головоломки "8", единственным способом тестирования алгоритма A*Search с использованием этой эвристики является вызов процедуры поиска и ручная проверка дерева, чтобы убедиться, что был выбран надлежащий шаг. Таким образом, тестирование алгоритма A*Search осложнено тем, что мы должны проверять его в контексте конкретной задачи и эвристики. У нас есть большое количество тестовых примеров для алгоритмов поиска пути, но во многих случаях они существуют только для того, чтобы убедиться в разумности выбранного хода (в игре или поиске в дереве), а не в выборе конкретного хода.

Тестирование алгоритмов в главе 9, "Вычислительная геометрия", было еще сложнее из-за вычислений с плавающей точкой. Рассмотрим наш подход к тестированию алгоритма сканирования выпуклой оболочки. Первоначальная идея была в том, чтобы выполнить медленный алгоритм на основе грубой силы с производительностью $O(n^4)$ и сравнить его вывод с выводом алгоритма сканирования выпуклой оболочки.

В процессе обширных испытаний мы генерировали двумерные массивы данных, получая случайным образом равномерно распределенные точки из единичного квадрата. Однако, когда множество данных стало достаточно большим, мы стали сталкиваться с ситуациями, когда результаты двух алгоритмов были разными. Было ли это следствием ошибки реализации алгоритма? В конце концов мы обнаружили, что арифметика с плавающей точкой в медленном алгоритме дает немного (очень немного) отличающиеся результаты при сравнении с алгоритмом сканирования выпуклой оболочки. Было ли это просто случайностью? К сожалению, нет. Мы также заметили, что алгоритм LineSweep дает результаты, немного отличающиеся при сравнении от результатов алгоритма поиска пересечении с помощью грубой силы.

Какой же алгоритм дает "правильный" результат? Эти нестыковки заставили нас разработать единообразное понятие сравнения значений с плавающей точкой. В частности, мы определили (несколько произвольно) пороговое значение `FloatingPoint.epsilon`, ниже которого становится невозможным различать два числа, отличающиеся одно от другого на величину, меньшую порогового значения. Когда вычисления приводили к значениям вблизи этого порога (который мы установили равным 10^{-9}), часто встречалось неожиданное поведение алгоритма. Полное удаление порогового значения тоже не решало проблему. В конечном итоге мы прибегли к статистической проверке результатов этих алгоритмов, а не к абсолютным и окончательным ответам для всех случаев.

В табл. 12.7 приведены алгоритмы, представленные в главе 9, "Вычислительная геометрия". Каждый из этих алгоритмов работает с двумерными геометрическими структурами и сталкивается с проблемой точного выполнения геометрических вычислений.

Таблица 12.7. Вычислительная геометрия

Алгоритм	Наилучший случай	Средний случай	Наихудший случай	Концепции	Страница книги
Сканирования выпуклой оболочки	n	$n \cdot \log n$	$n \cdot \log n$	Жадный алгоритм	295
LineSweep	$(n+k) \cdot \log n$	$(n+k) \cdot \log n$	n^2	Очередь с приоритетами, бинарное дерево	305
Диаграмма Вороного	$n \cdot \log n$	$n \cdot \log n$	$n \cdot \log n$	LineSweep, очередь с приоритетами, бинарное дерево	316

Допустимость приближенных решений

Во многих случаях приблизительный результат является приемлемым, если он может быть вычислен гораздо быстрее, чем точный результат, и имеет известное отклонение от правильного результата. Неограниченная задача о рюкзаке представляет собой именно такой случай, поскольку приближение не может быть хуже 50% от точного результата. Эти приближения могут использовать случайные числа для вычисления оценки точного ответа (как мы видели в примере для подсчета количества решений задачи о ферзях). Используйте этот подход, когда известно, что повторные испытания увеличивают точность оценки.

Фильтр Блума тщательно разработан таким образом, чтобы при поиске элемента в коллекции он мог давать ложные положительные результаты, но никогда не мог дать ложный отрицательный результат. На первый взгляд, может показаться, что бесполезно иметь алгоритм, который способен вернуть неверный ответ. Но фильтр Блума может значительно сократить время выполнения алгоритмов, связанных с хранением данных во вторичной памяти или с поиском в базах данных. Когда он возвращает отрицательный ответ, это действительно означает, что элемента в коллекции нет, поэтому нет необходимости проводить более дорогостоящий поиск. Конечно, иногда фильтр Блума приводит к выполнению неудачного поиска, который дает отрицательный результат, но это не влияет на корректность приложения в целом.

Повышение производительности с помощью параллелизма

При рассмотрении алгоритмов, представленных в этой книге, предполагается работа на одном последовательном компьютере. Если удается выделить подзадачи, которые можно вычислять независимо, то можно спроектировать многопоточное решение с использованием имеющихся ресурсов, предоставляемых современными компьютерами. Например, в главе 11, "Дополнительные категории алгоритмов", показано, как ускорить алгоритм быстрой сортировки с помощью распараллеливания. Могут ли какие-то другие алгоритмы из этой книги использовать параллелизм? Напомним, что алгоритм сканирования выпуклой оболочки в качестве начального этапа выполняет сортировку, а затем решает две независимые задачи: построение нижней части оболочки и построение верхней части оболочки. Для повышения производительности эти задачи можно решать параллельно. В табл. 12.8 показано достигаемое при этом ускорение (см. код `algs.model.problems.convexhull.parallel` в репозитории). Несмотря на впечатляющие показатели, алгоритм по-прежнему имеет время работы $O(n \cdot log\ n)$, хотя и с лучшими константами.

Таблица 12.8. Повышение производительности многопоточного алгоритма сканирования выпуклой оболочки

n	Однопоточный	1 вспомогательный поток	2 вспомогательных потока	3 вспомогательных потока	4 вспомогательных потока
2048	0,8571	0,5000	0,6633	0,5204	0,6020
4096	1,5204	0,7041	0,7041	0,7755	0,7857
8192	3,3163	0,9592	1,0306	1,0306	1,0816
16384	7,3776	1,6327	1,6327	1,5612	1,6939
32768	16,3673	3,0612	2,8980	2,9694	3,1122
65536	37,1633	5,8980	6,0102	6,0306	6,0408
131072	94,2653	13,8061	14,3776	14,1020	14,5612
262144	293,2245	37,0102	37,5204	37,5408	38,2143
524288	801,7347	90,7449	92,1939	91,1633	91,9592
1048576	1890,5612	197,4592	198,6939	198,0306	200,5612

Большинство последовательных алгоритмов не могут достичь теоретического максимального ускорения, потому что распараллелить между несколькими потоками можно только часть алгоритма; этот факт известен как *закон Амдала*. Не пытайтесь использовать столько потоков, сколько возможно в данном алгоритме. Добавление множества вспомогательных потоков требует более сложных программ, чем добавление одного вспомогательного потока. Используя один вспомогательный поток, можно обеспечить заметное улучшение производительности с помощью лишь незначительного увеличения сложности.

Однако не каждый алгоритм может быть улучшен с помощью параллелизма. Например, при поиске ближайшего соседа с использованием *k-d*-дерева в процессе работы может возникать двойная рекурсия. Распараллеливание вызовов этого отдельного метода будет снижать общую производительность из-за необходимости синхронизировать вспомогательные потоки так, чтобы оба они завершились одновременно.

Приложение

Хронометраж

Каждый алгоритм в этой книге сопровождается сведениями о его производительности. Поскольку для получения точной производительности важно использовать правильный хронометраж, в этом приложении мы представляем нашу инфраструктуру для оценки производительности алгоритма. Это приложение должно также помочь разрешить любые вопросы или сомнения, которые вы, возможно, испытываете относительно корректности нашего подхода. Мы пытаемся точно описать средства вычисления наших эмпирических данных, чтобы позволить вам убедиться в точности приводимых в книге результатов и понять, когда предположения о работе алгоритма соответствуют контексту, в котором он должен использоваться.

Имеется множество способов анализа алгоритмов. В главе 2, "Математика алгоритмов", представлен теоретический, формальный подход с введением концепций анализа наихудшего и среднего случаев. Эти теоретические результаты в некоторых (но не во всех) случаях могут быть оценены эмпирически. Например, рассмотрим оценку производительности алгоритма для сортировки 20 чисел. Имеется $2{,}43 \cdot 10^{18}$ перестановок этих 20 чисел, и для вычисления среднего случая мы не можем просто исчерпывающим образом вычислить каждую из этих перестановок. Кроме того, мы не можем вычислить среднее путем измерения времени сортировки всех этих перестановок. Мы вынуждены полагаться на законы статистики, чтобы убедиться, что мы правильно вычислили ожидаемую производительность алгоритма.

Статистические основы

В этой главе мы сосредоточимся на основных моментах оценки эффективности алгоритмов. Заинтересованные читатели должны обратиться к любому из большого числа имеющихся учебников по математической статистике для получения

дополнительных знаний о математической статистике, используемой при проведении эмпирических измерений в этой книге.

Для вычисления производительности алгоритма мы создаем множество из T независимых *испытаний*, в каждом из которых выполняется этот алгоритм. Каждое испытание предназначено для выполнения алгоритма для входной задачи размером n. Прикладываются определенные усилия для обеспечения разумной *эквивалентности* этих задач для алгоритма. Когда испытания фактически идентичны, цель заключается в количественном определении *стандартного отклонения* производительности данной реализации алгоритма. Это может быть полезно, например, когда вычисление большого числа независимых эквивалентных испытаний обходится слишком дорого.

Множество испытаний выполняется; при этом показания таймера (с точностью до миллисекунд) снимаются до и после наблюдаемого поведения. Если код написан на Java, системный сборщик мусора вызывается непосредственно перед началом испытания. Хотя это и не может гарантировать, что сборщик мусора не выполняется во время испытания, все же таким образом уменьшается риск траты дополнительного времени, не связанного с алгоритмом. Из полного набора T записанных показаний времени выполнения наилучшее и наихудшее значения времени выполнения отбрасываются как "выбросы". Остальные T–2 записи времени выполнения усредняются, а стандартное отклонение вычисляется с использованием формулы

$$\sigma = \sqrt{\frac{\sum_i (x_i - x)^2}{n-1}}$$

Здесь x_i — время выполнения отдельного испытания, а x — среднее значение для T–2 испытаний. Обратите внимание, что здесь n равно T–2, так что знаменатель под знаком радикала равен T–3. Вычисленные средние значения и стандартные отклонения помогают предсказать будущую производительность алгоритма на основе табл. 1, в которой показана вероятность (от 0 до 1) того, что фактическое значение будет находиться в диапазоне $[x-k\cdot\sigma, x+k\cdot\sigma]$, где σ — стандартное отклонение, вычисленное с помощью приведенной выше формулы. Значения вероятности представляют собой *доверительные интервалы*, описывающие уверенность в наших прогнозах.

Таблица 1. Таблица стандартных отклонений

k	Вероятность
1	0,6827
2	0,9545
3	0,9973
4	0,9999
5	1,0000

Например, при рандомизированном испытании ожидается, что в 68,27% случаев результат измерения времени будет попадать в диапазон $[x-\sigma, x+\sigma]$.

При представлении результатов мы никогда не даем числа более чем с четырьмя десятичными разрядами, чтобы не создавать ошибочное впечатление более высокой точности получаемых нами значений. Так что при получении значения наподобие 16,897986 мы приводим его в книге как 16,8980.

Пример

Предположим, что мы хотим хронометрировать сложение чисел от 1 до n. Эксперимент состоит в измерении времени для значений от $n = 8000000$ до $n = 16000000$ с шагом увеличения 2 млн. Поскольку для каждого значения n задачи идентичны, мы выполняем по 30 испытаний для устранения изменчивости, насколько это возможно.

Наша гипотеза заключается в том, что изменение времени вычисления суммы будет связано непосредственно с n. Мы приводим три программы для решения этой задачи — на C, Java и Python — и инфраструктуру хронометража, показывая, как она используется.

Решение для Java

В испытаниях на Java текущее системное время (в миллисекундах) определяется непосредственно до и после выполнения испытания. Код в примере 1 измеряет время, необходимое для выполнения задачи. В идеальном компьютере все 30 испытаний должны потребовать одного и того же количества времени. Конечно, такое произойдет вряд ли, потому что современные операционные системы выполняют множество задач, которые совместно используют тот же процессор, на котором выполняется испытываемый код.

Пример 1. Пример записи времени выполнения испытания на Java

```
public class Main
{
    public static void main(String[] args)
    {
        TrialSuite ts = new TrialSuite();

        for (long len = 8000000; len <= 16000000; len += 2000000)
        {
            for (int i = 0; i < 30; i++)
            {
                System.gc();
                long now = System.currentTimeMillis();

                /** Хронометрируемая задача. */
                long sum = 0;

                for (int x = 1; x <= len; x++)
```

```
                {
                    sum += x;
                }

                long end = System.currentTimeMillis();
                ts.addTrial(len, now, end);
            }
        }

        System.out.println(ts.computeTable());
    }
}
```

Класс `TrialSuite` хранит времена выполнения испытаний. После того как времена всех испытаний добавлены в набор, вычисляется результирующая таблица. Для этого записанные значения времени суммируются (после удаления минимального и максимального значений) и вычисляются среднее значение и стандартное отклонение.

Решение для Linux

Для тестовых примеров на языке C мы разработали библиотеку хронометража, которая компоновалась с исследуемым кодом. В этом разделе мы кратко опишем основные аспекты данного кода, так что за дополнительной информацией заинтересованный читатель может обратиться к репозиторию кода к данной книге.

Созданная, в первую очередь, для тестирования процедур сортировки, инфраструктура на основе языка C может быть скомпонована с существующим исходным кодом. API измерения времени берет на себя ответственность за анализ аргументов командной строки:

```
Применение: timing [-n NumElements] [-s seed] [-v] [OriginalArguments]
    -n    Объявляет размер задачи [по умолчанию: 100000]
    -v    Подробный вывод [по умолчанию: false]
    -s #  Устанавливает инициализатор ГСЧ [по умолчанию: отсутствует]
    -h    Вывод информации об использовании
```

Библиотека хронометража предполагает, что будет испытана задача с размером, определяемым флагом `[-n]`. Для получения повторяющихся испытаний инициализирующее генератор случайных чисел значение можно установить с помощью параметра `[-s #]`. Для работы с библиотекой хронометража тестируемый код предоставляет следующие функции.

`void problemUsage()`

Выводит на консоль набор параметров командной строки, поддерживаемых данным конкретным кодом. Обратите внимание, что библиотека хронометража анализирует только "свои" параметры, связанные с хронометражом, а остальные аргументы передаются функции `prepareInput`.

`void prepareInput (int size, int argc, char **argv)`

Для некоторых задач эта функция отвечает за создание входного множества данных, которые будут обрабатываться в функции `execute`. Обратите внимание, что эта информация не передается непосредственно функции `execute` через формальный аргумент, а хранится в виде статической переменной в рамках тестового примера.

`void postInputProcessing()`

Если после решения задачи требуется какая-либо проверка, ее можно выполнить здесь.

`void execute()`

Этот метод содержит тело хронометрируемого кода. Поскольку код выполняется как часть измерения времени однократно, он мало влияет на измеряемое время. Если этот метод пуст, считается, что накладные расходы не влияют на общее время выполнения.

В примере 2 показан код для рассмотренного нами сложения *n* чисел.

Пример 2. Задача сложения n чисел

```
extern int numElements; /* Размер задачи */

void problemUsage()        {   /* Пусто */   }
void prepareInput()        {   /* Пусто */   }
void postInputProcessing(){   /* Пусто */   }

void execute() {
    int x;
    long sum = 0;
    for (x = 1; x <= numElements; x++) { sum += x; }
}
```

Каждое выполнение функции соответствует одному испытанию, поэтому у нас имеется набор сценариев для многократного выполнения тестируемого кода и набора статистики. Для каждого множества тестов создается файл конфигурации `config.rc`. В примере 3 показан файл конфигурации для сортировки на основе значения, использовавшийся в главе 4, "Алгоритмы сортировки".

Пример 3. Пример файла конфигурации для сравнения сортировок

```
# Настройка для использования этих сортировок
BINS=./Insertion ./Qsort_2_6_11 ./Qsort_2_6_6 ./Qsort_straight

# Настройка тестов
TRIALS=10
LOW=1
HIGH=16384
INCREMENT=*2
```

Этот файл спецификации объявляет, что множество выполнимых файлов состоит из трех вариантов быстрой сортировки и одной сортировки вставками. Тесты выполняются для размеров задачи в диапазоне от $n=1$ до $n=16384$, где n удваивается после каждого запуска. Для задачи каждого размера выполняются 10 испытаний. Лучшие и худшие значения отбрасываются, и сгенерированная таблица отчета будет содержать средние значения (и стандартные отклонения) для оставшихся восьми испытаний.

В примере 4 содержится сценарий `compare.sh`, который генерирует совокупный набор сведений для конкретного размера задачи, равного n.

Пример 4. Сценарий хронометража `compare.sh`

```
#!/bin/bash
#
# Этот сценарий ожидает два аргумента:
#   $1 -- размер задачи n
#   $2 - количество выполняемых испытаний
# Этот сценарий читает свои параметры из файла конфигурации $CONFIG
#   BINS    множество хронометрируемых выполнимых файлов
#   EXTRAS  дополнительные аргументы командной строки
#           для использования при выполнении этих файлов
#
# CODE устанавливается для каталога со сценариями
CODE='dirname $0'

SIZE=20
NUM_TRIALS=10
if [ $# -ge 1 ]
then
    SIZE=$1
    NUM_TRIALS=$2
fi

if [ "x$CONFIG" = "x" ]
then
    echo "Не определен файл конфигурации (\$CONFIG)"
    exit 1
fi

if [ "x$BINS" = "x" ]
then
    if [ -f $CONFIG ]
    then
        BINS='grep "BINS=" $CONFIG | cut -f2- -d'='
    EXTRAS='grep "EXTRAS=" $CONFIG | cut -f2- -d'='
    fi
    if [ "x$BINS" = "x" ]
    then
        echo "Нет переменной \$BINS и конфигурации $CONFIG "
```

```
        echo "\$BINS должен представлять собой список"
        echo "    выполнимых файлов, разделенных пробелами"
    fi
fi
echo "Report: $BINS on size $SIZE"
echo "Date: 'date'"
echo "Host: 'hostname'"
RESULTS=/tmp/compare.$$
for b in $BINS
do
    TRIALS=$NUM_TRIALS
    # Начинаем с количества попыток с общими значениями
    echo $NUM_TRIALS > $RESULTS
    while [ $TRIALS -ge 1 ] do
        $b -n $SIZE -s $TRIALS $EXTRAS | grep secs | \
            sed 's/secs//' >> $RESULTS
        TRIALS=$((TRIALS-1))
    done
    # Вычисление среднего и стандартного отклонения
    RES='cat $RESULTS | $CODE/eval'
    echo "$b $RES"
    rm -f $RESULTS
done
```

compare.sh использует небольшую программу на языке C — eval, — которая вычисляет среднее значение и стандартное отклонение, используя описанный в начале этой главы метод. Этот сценарий compare.sh многократно выполняется управляющим сценарием suiteRun.sh, который выполняет итерации с требуемыми размерами задач, указанными в файле config.rc, как показано в примере 5.

Пример 5. Сценарий suiteRun.sh

```
#!/bin/bash
CODE='dirname $0'

# При отсутствии аргументов используется
# файл конфигурации по умолчанию
if [ $# -eq 0 ]
then
    CONFIG="config.rc"
else
    CONFIG=$1
    echo "Используется файл конфигурации $CONFIG..."
fi

# Экспорт для передачи информации сценарию compare.sh
export CONFIG
```

```
# Получение информации
if [ -f $CONFIG ]
then
    BINS='grep "BINS=" $CONFIG | cut -f2- -d'='
    TRIALS='grep "TRIALS=" $CONFIG | cut -f2- -d'='
    LOW='grep "LOW=" $CONFIG | cut -f2- -d'='
    HIGH='grep "HIGH=" $CONFIG | cut -f2- -d'='
    INCREMENT='grep "INCREMENT=" $CONFIG | cut -f2- -d'='
else
    echo "Файл конфигурации ($CONFIG) не найден."
    exit -1
fi

# Заголовки
HB='echo $BINS | tr ' ' ','
echo "n,$HB"

# Сравнение испытаний с размерами от LOW до HIGH
SIZE=$LOW
REPORT=/tmp/Report.$$
while [ $SIZE -le $HIGH ]
do
    # По одному разу для каждой записи в $BINS
    $CODE/compare.sh $SIZE $TRIALS | awk 'BEGIN{p=0} \
            {if(p) { print $0; }} \
            /Host:/{p=1}' | cut -d' ' -f2 > $REPORT

    # Соединение информации о средних значениях.
    # Стандартное отклонение игнорируется.
    # ------------------------------
    VALS='awk 'BEGIN{s=""}\
        {s = s "," $0 }\
        END{print s;}' $REPORT'
    rm -f $REPORT

    echo $SIZE $VALS

    # $INCREMENT может быть "+ NUM" или "* NUM".
    SIZE=$(($SIZE$INCREMENT))
done
```

Хронометраж с использованием Python

Код на языке Python в примере 6 измеряет производительность задачи суммирования. Он использует модуль `timeit`, являющийся стандартом для измерения времени выполнения фрагментов кода и целых программ на языке программирования Python.

Пример А.6. Исследование времени выполнения на языке Python

```
import timeit

def performance():
    """демонстрация производительности кода."""
    n = 8000000
    numTrials = 10
    print ("n", "Add time")
    while n <= 16000000:
        setup = 'total=0'
        code = 'for i in range(' + str(n) + '): total += i'
        add_total = min(timeit.Timer(code,
                        setup=setup).repeat(5,numTrials))

        print ("%d %5.4f " % (n, add_total ))
        n += 2000000

if __name__ == '__main__':
    performance()
```

Модуль `timeit` возвращает список значений, отражающих время выполнения фрагмента кода в секундах. Применяя к этому списку функцию `min`, мы извлекаем наилучшее время выполнения среди всех испытаний. Документация по модулю `timeit` объясняет преимущества использования этого подхода для хронометража программ на языке программирования Python.

Вывод результатов

Поучительно проанализировать фактические результаты работы трех различных реализаций (на разных языках программирования) одной и той же программы при вычислениях на одной и той же платформе. Мы представляем три таблицы (табл. 2, 4 и 5), по одной для C, Java и Python. В каждой таблице мы представляем результаты в миллисекундах и краткую таблицу с гистограммой для Java.

Таблица 2. Результаты хронометража при использовании Java

n	Среднее	min	max	Стандартное отклонение
8000000	7,0357	7	12	0,1890
10000000	8,8571	8	42	0,5245
12000000	10,5357	10	11	0,5079
14000000	12,4643	12	14	0,6372
16000000	14,2857	13	17	0,5998

Итоговое поведение, представленное в табл. 2, показано в табл. 3 детализированным в виде гистограммы. Мы опустили строки таблиц, состоящие только из нулевых значений; все ненулевые значения выделены с помощью серого фона.

Таблица 3. Разбивка хронометража на отдельные измерения

Время (мс)	8000000	10000000	12000000	14000000	16000000
7	28	0	0	0	0
8	1	7	0	0	0
9	0	20	0	0	0
10	0	2	14	0	0
11	0	0	16	0	0
12	1	0	0	18	0
13	0	0	0	9	1
14	0	0	0	3	22
15	0	0	0	0	4
16	0	0	0	0	2
17	0	0	0	0	1
42	0	1	0	0	0

Для интерпретации этих результатов обратимся к статистике и *доверительным интервалам*, описанным выше. Мы предполагаем, что времена выполнения для каждого испытания независимы. Если бы нас попросили предсказать длительность предполагаемого выполнения программы для $n=12000000$, то мы заметили бы, что среднее время работы $x=12{,}619$, а стандартное отклонение $\sigma=0{,}282$. Рассмотрим диапазон значений $[x-2\sigma, x+2\sigma]$, который охватывает два стандартных отклонения от среднего значения. Как следует из табл. 1, вероятность того, что измеренное время испытания будет находиться в диапазоне [9,5199; 11,5515], равна 95,45%.

Таблица 4. Результаты хронометража (в мс) для C

n	Среднее	min	max	Стандартное отклонение
8000000	8,376	7,932	8,697	0,213
10000000	10,539	9,850	10,990	0,202
12000000	12,619	11,732	13,305	0,282
14000000	14,681	13,860	15,451	0,381
16000000	16,746	15,746	17,560	0,373

Несколько лет назад мы обнаружили бы заметные различия между значениями времени выполнения этих трех программ. Улучшения реализации языка (особенно оперативная (JIT) компиляция) и аппаратных средств позволяет получить для данного конкретного вычисления практически одинаковую производительность. Результаты гистограммы не являются достаточно информативными, поскольку на самом деле время работы включает дробные части миллисекунд, в то время как стратегия

хронометража Java возвращает только целые значения. Сравнение с использованием более реалистичных программ показало бы большие различия времени выполнения для разных языков программирования.

Таблица 5. Результаты хронометража для Python

n	Время выполнения (мс)
8000000	7,9386
10000000	9,9619
12000000	12,0528
14000000	14,0182
16000000	15,8646

Точность

Вместо таймеров миллисекундного уровня мы могли бы использовать наносекундные таймеры. На платформе Java единственное изменение в приведенном ранее коде состоит в использовании `System.nanoTime()` вместо обращения к `System.currentTimeMillis()`. Чтобы понять, есть ли корреляции между миллисекундным и наносекундным таймерами, мы изменили код так, как показано в примере 7.

Пример 7. Использование наносекундных таймеров в Java

```
TrialSuite tsM = new TrialSuite();
TrialSuite tsN = new TrialSuite();

for (long len = 8000000; len <= 16000000; len += 2000000)
{
    for (int i = 0; i < 30; i++)
    {
        long nowM = System.currentTimeMillis();
        long nowN = System.nanoTime();
        long sum = 0;

        for (int x = 1; x <= len; x++)
        {
            sum += x;
        }

        long endM = System.currentTimeMillis();
        long endN = System.nanoTime();
        tsM.addTrial(len, nowM, endM);
        tsN.addTrial(len, nowN, endN);
    }
}

System.out.println(tsM.computeTable());
System.out.println(tsN.computeTable());
```

В приведенной ранее табл. 2 содержатся результаты хронометража в миллисекундах, табл. 6 — при использовании наносекундного таймера в языке C, а в табл. 7 показаны результаты при использовании наносекундного таймера в Java. Результаты этих вычислений достаточно точные. Поскольку мы не считаем, что использование наносекундных таймеров существенно влияет на точность хронометража, в книге использовались миллисекундные результаты. Мы используем миллисекунды еще и для того, чтобы не создавалось впечатление, что наши таймеры более точные, чем они являются на самом деле. И наконец, наносекундные таймеры в Unix-системах еще не стандартизированы, а так как есть моменты, когда мы хотели сравнивать время выполнения на разных платформах, это послужило еще одной причиной, по которой в книге мы решили использовать именно миллисекундные таймеры.

Таблица 6. Результаты при применении наносекундного таймера в C

n	Среднее	min	max	Стандартное отклонение
8000000	6970676	6937103	14799912	20067,5194
10000000	8698703	8631108	8760575	22965,5895
12000000	10430000	10340060	10517088	33381,1922
14000000	12180000	12096029	12226502	27509,5704
16000000	13940000	13899521	14208708	27205,4481

Таблица 7. Результаты при применении наносекундного таймера в Java

n	Среднее	min	max	Стандартное отклонение
8000000	6961055	6925193	14672632	15256,9936
10000000	8697874	8639608	8752672	26105,1020
12000000	10438429	10375079	10560557	31481,9204
14000000	12219324	12141195	12532792	91837,0132
16000000	13998684	13862725	14285963	124900,6866

Литература

1. Adel'son-Vel'skii, G. M. and E. Landis, "An algorithm for the organization of information," *Soviet Mathematics Doklady*, 3: 1259–1263, 1962.
2. Ahuja, R. K., T. Magnanti, and J. Orlin, *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, 1993.
3. Andrew, A. M., "Another efficient algorithm for convex hulls in two dimensions," *Information Processing Letters*, 9(5): 216–219, 1979, http://dx.doi.org/10.1016/0020-0190(79)90072-3.
4. Armstrong, J., *Programming Erlang: Software for a Concurrent World*. Second Edition. Pragmatic Bookshelf, 2013.
5. Aurenhammer, F., "Voronoi diagrams: A survey of a fundamental geometric data structure," *ACM Computing Surveys*, 23(3): 345–405, 1991, http://dx.doi.org/10.1145/116873.116880.
6. Barr, A. and E. Feigenbaum, *The Handbook of Artificial Intelligence*. William Kaufmann, Inc., 1981.
7. Bayer, R. and E. McCreight, "Organization and maintenance of large ordered indexes," Acta Inf. 1, 3, 173–189, 1972, http://dx.doi.org/10.1007/bf00288683.
8. Bazaraa, M., J. Jarvis, and H. Sherali, *Linear Programming and Network Flows*. Fourth Edition. Wiley, 2009.
9. Bentley, J. and M. McIlroy, "Engineering a sort function," *Software—Practice and Experience*, 23(11): 1249–1265, 1993. http://dx.doi.org/10.1002/spe.4380231105.
10. Bentley, J. L., F. Preparata, and M. Faust, "Approximation algorithms for convex hulls," *Communications of the ACM*, 25(1): 64–68, 1982, http://doi.acm.org/10.1145/358315.358392.
11. Bentley, J., *Programming Pearls*. Second Edition. Addison-Wesley Professional, 1999.
12. Berlekamp, E. and D. Wolfe, *Mathematical Go: Chilling Gets the Last Point*. A K Peters/CRC Press, 1994.

13. Berman, K. and J. Paul, *Algorithms: Sequential, Parallel, and Distributed.* Course Technology, 2004.
14. Bloom, B., "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, 13(7): 422–426, 1970, http://dx.doi.org/10.1145/362686.362692.
15. Blum, M., R. Floyd, V. Pratt, R. Rivest, and R. Tarjan, "Time bounds for selection," *Journal of Computer and System Sciences*, 7(4): 448–461, 1973, http://dx.doi.org/10.1016/S0022-0000(73)80033-9.
16. Bose, P., H. Guo, E. Kranakis, A. Maheshwari, P. Morin, J. Morrison, M. Smid, and Y. Tang, "On the false-positive rate of Bloom filters," *Information Processing Letters* 108: 210–213, 2008, http://dx.doi.org/10.1016/j.ipl.2008.05.018.
17. Botea, A., M. Muller, and J. Schaeffer, "Near optimal hierarchical path-finding," *Journal of Game Development*, 1(1): 2004, http://www.cs.ualberta.ca/~mmueller/ps/hpastar.pdf.
18. Christofides, N., "Worst-case analysis of a new heuristic for the traveling salesman problem," Report 388, Graduate School of Industrial Administration, CMU, 1976.
19. Comer, D., "Ubiquitous B-Tree," *Computing Surveys*, 11(2): 123–137, 1979. http://dx.doi.org/10.1145/356770.356776.
20. Cormen, T. H., C. Leiserson, R. Rivest, and C. Stein, *Introduction to Algorithms.* Third Edition. MIT Press, 2009.
    Имеется русский перевод: Т.Х. Кормен, Ч.И. Лейзерсон, Р.Л. Ривест, К. Штайн, *Алгоритмы: построение и анализ. 3-е изд.* — М.: ООО "И.Д. Вильямс", 2013. — 1328 с.
21. Davis, M. and K. Whistler, "*Unicode Collation Algorithm, Unicode Technical Standard #10*," June 2015, http://unicode.org/reports/tr10/.
22. Eddy, W., "A new convex hull algorithm for planar sets," *ACM Transactions on Mathematical Software*, 3(4): 398–403, 1977, http://dx.doi.org/ 10.1145/355759.355766.
23. Ford, L. R. Jr. and D. Fulkerson, *Flows in Networks.* Princeton University Press, 2010.
24. Fortune, S., "A sweepline algorithm for Voronoi diagrams," Proceedings of the 2nd Annual Symposium on Computational Geometry. ACM, New York, 1986, pp. 313–322, http://doi.acm.org/10.1145/10515.10549.
25. Fragouli, C. and T. Tabet, "On conditions for constant throughput in wireless networks," *ACM Transactions on Sensor Networks*, 2(3): 359–379, 2006, http://dx.doi.org/10.1145/1167935.1167938.
26. Gilreath, W., "*Hash sort: A linear time complexity multiple-dimensional sort algorithm*," Proceedings, First Southern Symposium on Computing, 1998, http://arxiv.org/abs/cs.DS/0408040.
27. Goldberg, A. V. and R. Tarjan, "A new approach to the maximum flow problem," Proceedings of the eighteenth annual ACM symposium on theory of computing, pp. 136–146, 1986, http://dx.doi.org/10.1145/12130.12144.
28. Goldberg, D., "What Every Computer Scientist Should Know About Floating-Point Arithmetic," *ACM Computing Surveys*, March 1991, http://docs.sun.com/source/806-3568/ncg_goldberg.html.

29. Guttman, A., "R-trees: a dynamic index structure for spatial searching," in Proceedings of the SIGMOD international conference on management of data, 1984, pp. 47–57, http://dx.doi.org/10.1145/971697.602266.
30. Hartmann, S., *Project Scheduling Under Limited Resources: Models, Methods, and Applications*. Springer, 1999.
31. Hester, J. H. and D. Hirschberg, "Self-organizing linear search," *ACM Computing Surveys*, 17(3): 295–311, 1985, http://dx.doi.org/10.1145/5505.5507.
32. Kaindl, H. and G. Kainz, "Bidirectional heuristic search reconsidered," *Journal of Artificial Intelligence Research*, 7: 283–317, 1997, http://arxiv.org/pdf/cs/9712102.pdf.
33. Knuth, D. E., "Estimating the efficiency of backtrack programs," *Mathematics of Computation*, 29(129): 121–136, 1975.
34. Knuth, D. E., *The Art of Computer Programming, Volume 3: Sorting and Searching*. Second Edition. Addison-Wesley, 1998.
Имеется русский перевод: Кнут Д.Э., *Искусство программирования, том 3. Сортировка и поиск, 2-е изд.* — М.: ООО "И.Д. Вильямс", 2000. — 832 с.
35. Korf, R. E., "Recent progress in the design and analysis of admissible heuristic functions," Proceedings, Abstraction, Reformulation, and Approximation: 4th International Symposium (SARA), Lecture notes in Computer Science #1864: 45–51, 2000, http://www.aaai.org/Papers/AAAI/2000/AAAI00-212.pdf.
36. Laramee, F. D., "Chess programming Part IV: Basic search," GameDev.net, August 26, 2000, http://www.gamedev.net/reference/articles/article1171.asp.
37. Leighton, T. and S. Rao, "Multicommodity max-flow min-cut theorems and their use in designing approximation algorithms," *Journal of the ACM*, 46 (6): 787–832, 1999. http://dx.doi.org/10.1145/331524.331526.
38. McCall, E. H., "Performance results of the simplex algorithm for a set of real-world linear programming models," *Communications of the ACM*, 25(3): 207–212, March 1982, http://dx.doi.org/10.1145/358453.358461.
39. Meagher, D. J. (1995). US Patent No. EP0152741A2. Washington, DC: U.S. Patent and Trademark Office, http://www.google.com/patents/EP0152741A2?cl=en.
40. Michalewicz, Z. and D. Fogel, *How to Solve It: Modern Heuristics*. Second Edition. Springer, 2004.
41. Musser, D., "Introspective sorting and selection algorithms," *Software—Practice and Experience*, 27(8): 983–993, 1997.
42. Nilsson, N., *Problem-Solving Methods in Artificial Intelligence*. McGraw-Hill, 1971.
43. Orden, A., "The Transhipment Problem," *Management Science*, 2(3): 276–285, 1956, http://dx.doi.org/10.1287/mnsc.2.3.276.
44. Parberry, I., "A real-time algorithm for the (n^2–1)-Puzzle," *Information Processing Letters*, 56(1): 23–28, 1995, http://dx.doi.org/10.1016/0020-0190(95)00134-X.
45. Pearl, J., *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley, 1984.
46. Pepicelli, G., "Bitwise optimization in Java: Bitfields, bitboards, and beyond," O'Reilly on Java.com, February 2, 2005, http://www.onjava.com/pub/a/onjava/ 2005/02/02/bitsets.html.

47. Preparata, F. and M. Shamos, *Computational Geometry: An Introduction*, Springer, 1993.
48. Reinefeld, A. and T. Marsland, "Enhanced iterative-deepening search," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16(7): 701–710, 1994, http://dx.doi.org/10.1109/34.297950.
49. Reinefeld, A., "Complete solution of the 8-puzzle and the benefit of node ordering in IDA," Proceedings of the 13th International Joint Conference on Artificial Intelligence (IJCAI), Volume 1, 1993, http://dl.acm.org/citation.cfm?id=1624060.
50. Russell, S. J. and P. Norvig, *Artificial Intelligence: A Modern Approach*. Third Edition. Prentice Hall, 2009.
51. Russell, S. J., "Efficient memory-bounded search methods," Proceedings, 10th European Conference on Artificial Intelligence (ECAI): 1–5, 1992.
52. Samuel, A., "Some studies in machine learning using the game of checkers," *IBM Journal* 3(3): 210–229, 1967, http://dx.doi.org/10.1147/rd.116.0601.
53. Schaeffer, J., "Game over: Black to play and draw in checkers," *Journal of the International Computer Games Association* (ICGA), https://ilk.uvt.nl/icga/journal/contents/Schaeffer07-01-08.pdf.
54. Schaeffer, J., N. Burch, Y. Bjornsson, A. Kishimoto, M. Muller, R. Lake, P. Lu, and S. Sutphen, "Checkers is solved," *Science Magazine*, September 14, 2007, 317(5844): 1518–1522, http://www.sciencemag.org/cgi/content/abstract/317/5844/1518.
55. Schmidt, D., "GPERF: A Perfect Hash Function Generator," C++ Report, SIGS, 10(10), 1998, http://www.cs.wustl.edu/~schmidt/PDF/gperf.pdf.
56. Sedgewick, R., "Implementing Quicksort programs," *Communications ACM*, 21(10): 847–857, 1978, http://dx.doi.org/10.1145/359619.359631.
57. Shannon, C., "Programming a computer for playing chess," *Philosophical Magazine*, 41(314): 1950, http://tinyurl.com/ChessShannon-pdf.
58. Trivedi, K. S., *Probability and Statistics with Reliability, Queueing, and Computer Science Applications*. Second Edition. Wiley-Blackwell Publishing, 2001.
59. Venners, B., "Floating-point arithmetic: A look at the floating-point support of the Java virtual machine," *JavaWorld*, 1996, http://www.javaworld.com/article/2077257/learn-java/floating-point-arithmetic.html.
60. Wichmann, D. and B. Wuensche, "Automated route finding on digital terrains," Proceedings of IVCNZ, Akaroa, New Zealand, pp. 107–112, November 2004, https://www.researchgate.net/publication/245571114_Automated_Route_Finding_on_Digital_Terrains.
61. Zuras, D., "More on squaring and multiplying large integers," *IEEE Transactions on Computers*, 43(8): 899–908, 1994, http://dx.doi.org/10.1109/12.295852.

Предметный указатель

А

Алгоритм, 21
  AlphaBeta, 222
  A*Search, 242
  Hierarchical Path-Finding A*, 253
  IterativeDeepeningA*, 252
  LineSweep, 305
  Minimax, 212
  NegMax, 217
  QuickHull, 301
  Simplex, 288
  Simplified Memory Bounded A*, 253
  Беллмана–Форда, 188
  Бентли–Фауста–Препараты, 26
  бинарный поиск, 124
  блочная сортировка, 101
  быстрая сортировка, 95
  вероятностный, 387
  выметание Форчуна, 317
  Дейкстры, 180
    для плотных графов, 185
  деления пополам, 42
  жадный. См. Стратегия жадная
  запрос точек в диапазоне, 348
  интроспективная сортировка, 101
  классификация, 39
  Крускала, 202
  линейный поиск, 120
  медленной оболочки, 294
  Мелкмана, 303
  обнаружение столкновений, 356
  параллельный, 382
  пирамидальная сортировка, 87
  поиска ближайшего соседа, 337
  поиск в глубину, 169; 232
  поиск в ширину, 175; 238
  поиск на основе хеша, 129
  последовательный поиск, 59; 120
  приближенный, 376
  Прима, 198
  семейства производительности, 40
  сканирование выпуклой оболочки, 25; 295
  сканирование Грэхема, 65; 303
  сортировка вставками, 81
  сортировка выбором, 86
  сортировка слиянием, 109
  сортировка хешированием, 107
  Флойда–Уоршелла, 193
  Форда–Фалкерсона, 261
  Эдмондса–Карпа, 268
Асимптотическая эквивалентность, 30

Б

Беллмана–Форда алгоритм, 188
Бентли–Фауста–Препараты алгоритм, 26
Бинарное дерево поиска, 150

В

Вороного диаграмма, 26; 316
Выметание Форчуна, 317
Выпуклая оболочка, 21; 294
Вычисления с плавающей точкой, 60
Вычислительная геометрия, 21

Г

Граф, 165
  анализ, 203
  вершина, 165
  взвешенный, 167
  двудольный, 274
  кратчайшие пути между всеми парами вершин, 193
  кратчайший путь из одной вершины, 180
  минимальное остовное дерево, 198
  ориентированный, 167
  петля, 165
  плотный, 168
  поиск в глубину, 169
  поиск в ширину, 175
  представление, 168; 203
  путь, 165
  разреженный, 203
  ребро, 165
    вес, 167
  связный, 168
  цикл, 168

Д

Дейкстры алгоритм, 180
  для плотных графов, 185
Деление пополам, 42
Дерево
  B-дерево, 362
  k-d-дерево, 334
  R-дерево, 336; 363
  игры, 205; 206
  квадрантов, 335; 355
  октантов, 362
  поиска, 151; 205; 229
    сбалансированное, 152
Диаграмма Вороного, 26; 316
Динамическое программирование, 71; 193; 376

З

Закон Амдала, 407

И

Идеальное хеширование, 145
Искусственный интеллект, 207
Итератор, 119; 121

К

Классификация алгоритмов, 39
Компаратор, 80
Коэффициент ветвления, 211
Коэффициент загрузки хеш-таблицы, 137
Крестики-нолики, 206
Крускала
  алгоритм, 202

Л

Линейное программирование, 287

М

Максимальный поток, 260
Массив, 30

Н

Наилучший случай, 37; 38
Наихудший случай, 36; 37

О

Обнаружение столкновений, 333

П

Паросочетание, 274
Пирамида, 88
Поиск, 119
  ассоциативный, 119
  бинарный, 124
  в глубину, 169; 232
  в ширину, 175; 238
  иерархический, 253
  линейный, 120
  на основе хеша, 129
  последовательный, 31; 59; 120
  пути, 205
Поток минимальной стоимости, 283
Предпросмотр, 212
Приведение задач, 403
Прима
  алгоритм, 198

Р

Расстояние редактирования, 71

С

Семейства производительности
  алгоритмов, 40
Сканирование
  выпуклой оболочки, 25; 295
  Грэхема, 65; 303
Скорость роста, 30
Сортировка, 77
  блочная, 101
  быстрая, 95
  вставками, 81
  выбором, 86
  интроспективная, 101
  перестановками, 81
  пирамидальная, 87
  слиянием, 109
  устойчивая, 80
  хешированием, 107
Состояние доски, 232
Средний случай, 36; 37
Статическая оценка, 209
Стратегия
  динамическое программирование, 71;
    193
  жадная, 69; 181; 198
  приведение задач, 274; 403
  разделяй и властвуй, 47; 70
Структура данных, 21
  AVL-дерево, 154
  B-дерево, 163; 362
  k-d-дерево, 334
  quadtree, 335
  R-дерево, 336; 363
  бинарное дерево поиска, 150
  граф, 165; 166
  дерево квадрантов, 335; 355
  дерево октантов, 362
  дерево поиска, 151
  красно-черное дерево, 163
  пирамида, 88
  пространственное дерево, 333
  хеш-таблица, 129

Т

Транзитивность, 64
Транспортная сеть, 257
  задача назначения, 287
  задача перевозки, 286
  задача перегрузки, 284
  максимальный поток, 260
  минимальное сечение, 261; 270
  поток минимальной стоимости, 283
  пропускные способности вершин, 273
  увеличивающий путь, 266; 278

Ф

Фильтр Блума, 146
Флойда–Уоршелла алгоритм, 193
Форда–Фалкерсона алгоритм, 261
Форчуна выметание, 317

Х

Хеш-таблица, 129
  коэффициент загрузки, 137
Хеш-функция, 129; 131

Ш

Шашки, 207

Э

Эвристика Экла–Туссена, 299
Эдмондса–Карпа алгоритм, 268
Экземпляр задачи, 29
  размер, 29

Алгоритмы: построение и анализ *3-е издание*

Томас Кормен,
Чарльз Лейзерсон,
Рональд Ривест,
Клиффорд Штайн

www.williamspublishing.com

Фундаментальный труд известных специалистов в области информатики достоин занять место на полке любого человека, чья деятельность так или иначе связана с вычислительной техникой и алгоритмами. Для профессионала эта книга может служить настольным справочником, для преподавателя — пособием для подготовки к лекциям и источником интересных нетривиальных задач, для студентов и аспирантов — отличным учебником.

Строгий математический анализ и обилие теорем сопровождаются большим количеством иллюстраций, элементарными рассуждениями и простыми приближенными оценками. Широта охвата материала и степень строгости его изложения дают основания считать эту книгу одной из лучших книг, посвященных разработке и анализу алгоритмов.

Третье издание этого классического труда в большой степени доработано. В нем появились новые главы, в том числе посвященные такой важной в последнее время теме, как многопоточные алгоритмы, а старые подверглись переработке, местами весьма существенной, когда уже имевшийся во втором издании материал излагается с иных позиций, чем ранее.

Данная книга будет не лишней как на столе студента и аспиранта, так и на рабочей полке практикующего программиста.

ISBN 978-5-8459-2016-4 **в продаже**

ПРИКЛАДНАЯ КРИПТОГРАФИЯ
ВТОРОЕ ИЗДАНИЕ
ПРОТОКОЛЫ, АЛГОРИТМЫ И ИСХОДНЫЕ КОДЫ НА ЯЗЫКЕ С

Брюс Шнайер

www.williamspublishing.com

Книга представляет собой юбилейный выпуск второго издания, подготовленный к 20-летнему юбилею всемирно признанного бестселлера. В ней приведен энциклопедический обзор криптографических алгоритмов и протоколов по состоянию на 1995 год. Многие годы эта книга остается самым авторитетным источником информации о криптографических протоколах и алгоритмах, которые легли в основу многих современных компьютерных технологий защиты данных. Книга предназначена для всех специалистов, использующих или разрабатывающих криптографические средства.

ISBN 978-5-9908462-4-1 **в продаже**

АЛГОРИТМЫ НА JAVA
4-Е ИЗДАНИЕ

Роберт Седжвик,
Кевин Уэйн

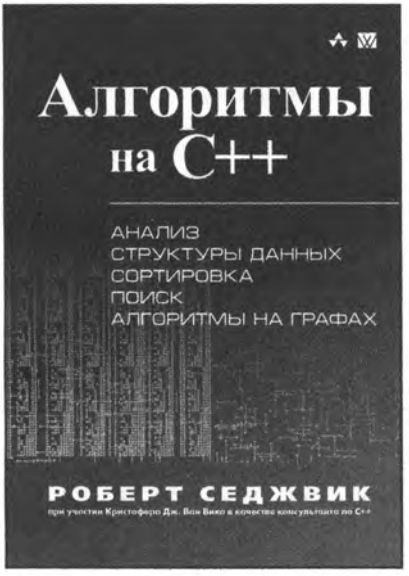
www.williamspublishing.com

Последнее издание из серии бестселлеров Седжвика, содержащее самый важный объем знаний, наработанных за последние несколько десятилетий. Содержит полное описание структур данных и алгоритмов для сортировки, поиска, обработки графов и строк, включая пятьдесят алгоритмов, которые должен знать каждый программист.
В книге представлены новые реализации, написанные на языке Java в доступном стиле модульного программирования — весь код доступен пользователю и готов к применению.
Алгоритмы изучаются в контексте важных научных, технических и коммерческих приложений. Клиентские программы и алгоритмы записаны в реальном коде, а не на псевдокоде, как во многих других книгах.
Дается вывод точных оценок производительности на основе соответствующих математических моделей и эмпирических тестов, подтверждающих эти модели.

ISBN 978-5-8459-2049-2 **в продаже**

АЛГОРИТМЫ НА C++
АНАЛИЗ, СТРУКТУРЫ ДАННЫХ, СОРТИРОВКА, ПОИСК, АЛГОРИТМЫ НА ГРАФАХ

Роберт Седжвик



www.williamspublishing.com

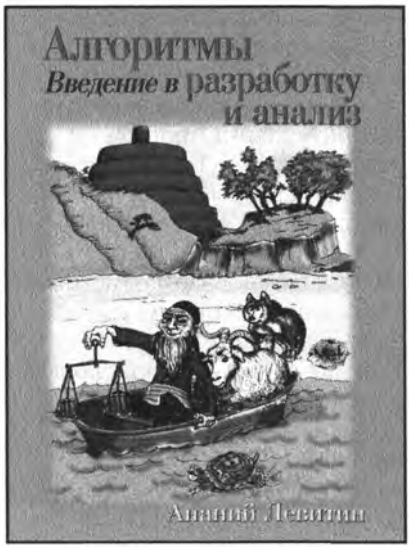
Эта классическая книга удачно сочетает в себе теорию и практику, что делает ее популярной у программистов на протяжении многих лет. Кристофер Ван Вик и Седжвик разработали новые лаконичные реализации на C++, которые естественным и наглядным образом описывают методы и могут применяться в реальных приложениях. Каждая часть содержит новые алгоритмы и реализации, усовершенствованные описания и диаграммы, а также множество новых упражнений для лучшего усвоения материала. Акцент на АТД расширяет диапазон применения программ и лучше соотносится с современными средами объектно-ориентированного программирования. Книга предназначена для широкого круга разработчиков и студентов.

ISBN 978-5-8459-2070-6

в продаже

АЛГОРИТМЫ:
ВВЕДЕНИЕ В РАЗРАБОТКУ И АНАЛИЗ

Ананий Левитин



www.williamspublishing.com

Книга ориентирована в первую очередь на студентов и аспирантов соответствующих специальностей. Несмотря на позиционирование книги в качестве учебного пособия, она может оказаться полезной и профессионалу в разработке алгоритмов — в первую очередь благодаря использованному автором новому подходу к классификации методов проектирования. Описание алгоритмов на естественном языке дополняется псевдокодом, который позволяет каждому, кто имеет хотя бы начальные знания и опыт программирования, реализовать алгоритм на используемом им языке программирования. Широта охвата материала, степень строгости и методология его изложения позволяют считать эту книгу одним из лучших учебников, посвященных разработке и анализу алгоритмов.

ISBN 978-5-8459-0987-9 **в продаже**

O'REILLY®

АЛГОРИТМЫ. СПРАВОЧНИК С ПРИМЕРАМИ НА C, C++, JAVA И PYTHON

Для создания надежного программного обеспечения необходимы эффективные алгоритмы, но программисты редко представляют себе весь спектр алгоритмов для решения своих задач. В данном обновленном издании описываются существующие алгоритмы для решения различных задач. Оно помогает выбрать и реализовать алгоритм, наиболее подходящий для ваших задач, при этом обеспечивая достаточное математическое обоснование для понимания и анализа производительности алгоритма.

Будучи акцентированной на приложениях, а не на теории, эта книга основана на строгих принципах, включая документированные решения реальных задач на разных языках программирования. В это издание добавлены десяток новых алгоритмов, реализованных на языке Python, в том числе реализация диаграмм Вороного, а также новая глава о пространственных древовидных структурах, таких как R-деревья и Quadtrees.

Основные темы книги

- Новые задачи и повышение эффективности имеющихся решений
- Поиск алгоритмов для решения своих задач и выбор наиболее подходящих из них
- Решения на языках программирования C, C++, Java, Python с помощью приведенных рекомендаций
- Оценка производительности алгоритмов и создание условий для достижения максимальной эффективности
- Использование наиболее подходящих структур данных для повышения эффективности алгоритмов

Категория: программирование
Предмет рассмотрения: разработка ПО/алгоритмы
Уровень: для пользователей средней и высокой квалификации

ISBN 978-5-9908910-7-4

Эта книга потрясающая по трем причинам: в ней легко найти нужные алгоритмы и структуры данных; стиль изложения материала скорее разговорный, чем академический; внимание читателя постоянно акцентируется на сравнительном анализе производительности алгоритмов. Если вы живете в реальном мире, эта книга навсегда изменит ваш способ использования структур данных.

Ричард Резник,
директор GQ Life Science

Джордж Хайнеман — адъюнкт-профессор информатики в WPI. В 2005 году был Председателем Международного симпозиума по компонентно-ориентированному программному обеспечению.

Гэри Поллис — профессор Вустерского политехнического института; один из авторов книги *Head First Object-Oriented Analysis and Design*.

Стэнли Селков в течение почти четырех десятилетий преподавал в университетах Ноксвилла, Вустера, Монреаля, Чунцина, Лозанны и Парижа.

ДИАЛЕКТИКА
www.dialektika.com