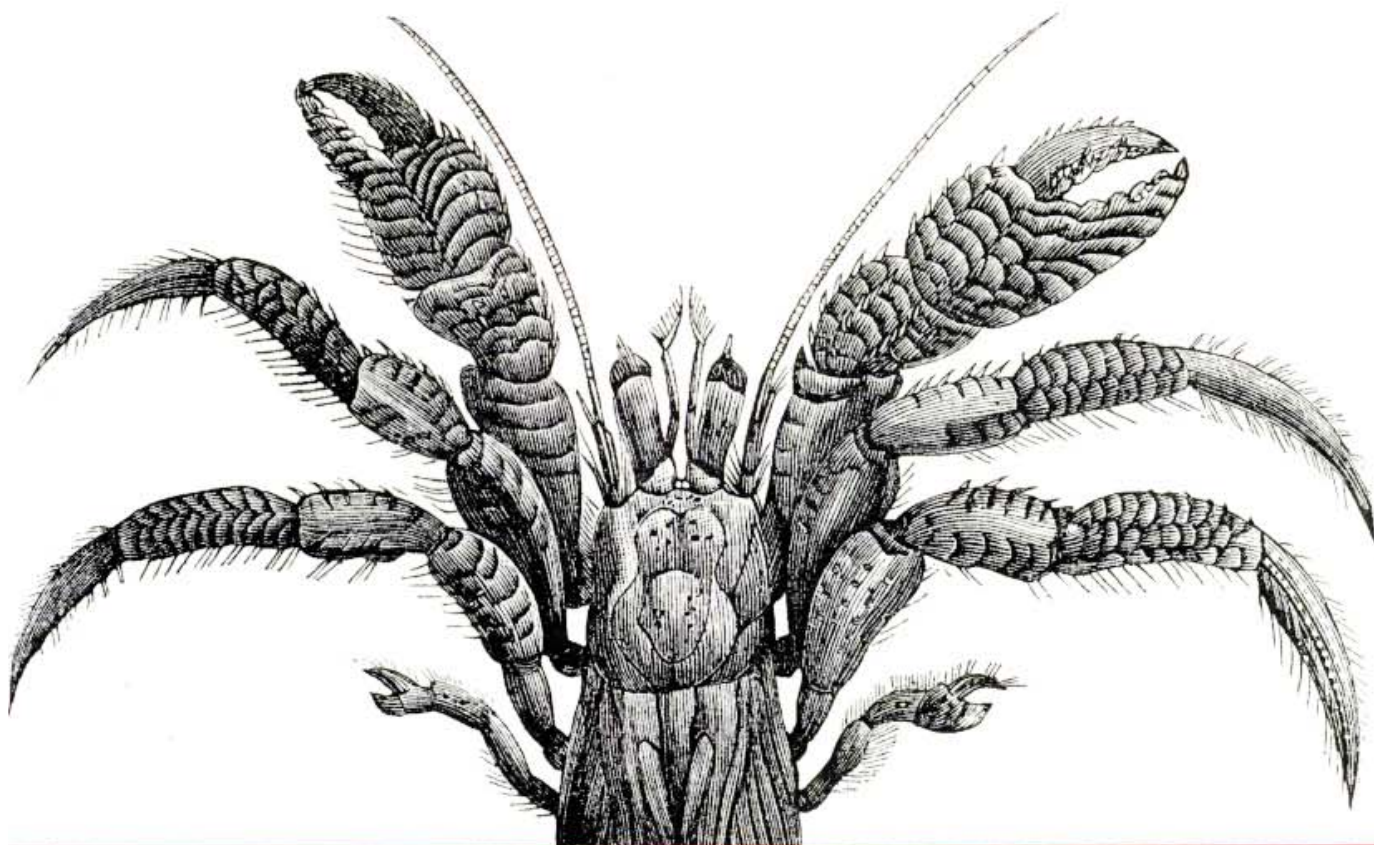


ALGORITHMS In a Nutshell



算法

技术手册

O'REILLY®

机械工业出版社
China Machine Press



George T. Heineman

Gary Pollice & Stanley Selkow 著

杨晨 李明 译

Java 程序员工资高吗？

年薪 10 万是一个很正常的薪资水平，你的目标应该是年薪 30 万+，这样就可以在北京、深圳、上海等城市拥有自己的房子、车子。可以让垂垂老矣的父母更好的享受人生；经常带着爱人去看爱情海...

我为什么学不好，找不到高薪工作？

也许你会在学无所用之中度过一生，因为你不知道大型企业现在和未来需要什么样的人才，方向错了一切努力等于白费。

如何才能找到高人指点迷津？

Java 学习群：72030155，一年 365 天，每天晚上 8: 30-11:30 有腾讯、百度、阿里内部首席架构师来免费讲课

我们为什么每天都要免费讲课？

- 1、口碑宣传
- 2、塑造品牌
- 3、免费课程都很棒，那么有更好的收费课程，你也会选择的

算法技术手册



开发健壮的软件需要高效的算法，然而程序员们往往直至问题发生之时，才会去求助于算法。本书讲解了许多现有的算法，可用于解决各种问题。通过阅读本书，你可以学会如何选择和实现正确的算法，来达成自己的目标。另外，书中的数学深浅适中，足够使你了解并分析算法的性能。

从理论而言，本书更专注于应用。本书提供了高效的代码解决方案，使用多种语言进行编写，让你可以轻松地将其应用于特定的工程当中。通过本书，你可以：

解决特定代码的问题，或者提升现有解决方案的性能。

快速找到与你所解决的问题相关的算法，并决定哪种算法才是最适合的。

探索使用C、C++、Java以及Ruby实现的算法解决方案以及开发小贴士。

了解算法预期的性能，以及它达到最高性能时所需要的条件。

发现不同算法之间相似的设计哲学。

学习高级数据结构，来提升算法的性能。

通过本书，你能学到如何提升算法的性能，这将是你的软件应用程序走向成功的关键。

“作者完成了一项了不起的工作，将晦涩的学术加以精炼，完美地在理论和实践之间取得了平衡，从而使本书成为了一本不可或缺的指南。从此，彻底领悟算法就变得十分简单了。”

——Matthew Russell,
Digital Reasoning Systems
高级技术总监，
《Dojo权威指南》
(O'Reilly) 的作者

George T. Heineman、Gary Pollice和Stanley Selkow均为伍斯特理工学院计算机科学系的教授。George是《Component-Based Software Engineering: Putting the Pieces Together》(Addison-Wesley)的联合主编。Gary是《Head First Object-Oriented Analysis and Design》(O'Reilly)的合著者。

服务热线：(010) 88378991, 88361066

书热线：(010) 68326294, 88379649, 68995259

稿热线：(010) 88379604

者信箱：hzjsj@hzbook.com

章网站：<http://www.hzbook.com>

上购书：www.china-pub.com



O'Reilly Media, Inc. 授权机械工业出版社出版

O'REILLY®
www.oreilly.com

无防伪贴标为盗版

ISBN 978-7-111-28674-5



9 787111 286745

定价：55.00元

简体中文版仅限于在中华人民共和国境内（但不允许在中国香港、澳门特别行政区和中国台湾地区）销售发行
This Authorized Edition for sale only in the territory of People's Republic of China (excluding Hong Kong, Macao and Taiwan)

算法技术手册

*George T. Heineman, Gary Pollice &
Stanley Selkow* 著

杨晨 李明 译

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Taipei • Tokyo

O'Reilly Media, Inc. 授权机械工业出版社出版

机械工业出版社

图书在版编目 (CIP) 数据

算法技术手册/ (美) 海涅曼 (Heineman, G. T.), 波利切 (Pollice, G.), 塞克欧 (Selkow, S.) 著; 杨晨等译. —北京: 机械工业出版社, 2009.10

书名原文: Algorithms in a Nutshell

ISBN 978-7-111-28674-5

I. 算… II. ①海… ②波… ③塞… ④杨… III. 电子计算机—算法理论—技术手册
IV. TP301.6-62

中国版本图书馆CIP数据核字 (2009) 第189407号

北京市版权局著作权合同登记

图字: 01-2009-2368号

©2008 by O'Reilly Media, Inc.

Simplified Chinese Edition, jointly published by O'Reilly Media, Inc. and China Machine Press, 2010. Authorized translation of the English edition, 2008 O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

All rights reserved including the rights of reproduction in whole or in part in any form.

英文原版由O'Reilly Media, Inc. 出版2008。

简体中文版由机械工业出版社出版 2010。英文原版的翻译得到O'Reilly Media, Inc.的授权。此简体中文版的出版和销售得到出版权和销售权的所有者——O'Reilly Media, Inc.的许可。

版权所有, 未得书面许可, 本书的任何部分和全部不得以任何形式重制。

本书法律顾问

北京市展达律师事务所

书 名/ 算法技术手册

书 号/ ISBN 978-7-111-28674-5

责任编辑/ 陈佳媛

封面设计/ Karen Montgomery, 张健

出版发行/ 机械工业出版社

地 址/ 北京市西城区百万庄大街22号 (邮政编码100037)

印 刷/ 北京京师印务有限公司

开 本/ 178毫米×233毫米 16开本 21.75印张

版 次/ 2010年3月第1版 2010年3月第1次印刷

定 价/ 55.00元 (册)

凡购本书, 如有倒页、脱页、缺页, 由本社发行部调换

客服热线: (010) 88378991; 88361066

购书热线: (010) 68326294; 88379649; 68995259

投稿热线: (010) 88379604

读者信箱: hzsj@hzbook.com

算法技术手册



O'Reilly Media, Inc.介绍

为了满足读者对网络 and 软件技术知识的迫切需求，世界著名计算机图书出版机构 O'Reilly Media, Inc. 授权机械工业出版社，翻译出版一批该公司久负盛名的英文经典技术专著。

O'Reilly Media, Inc. 是世界上在 UNIX、X、Internet 和其他开放系统图书领域具有领导地位的出版公司，同时也是联机出版的先锋。

从最畅销的 *The Whole Internet User' Guide & Catalog*（被纽约公共图书馆评为20世纪最重要的50本书之一）到 GNN（最早的Internet门户和商业网站），再到 WebSite（第一个桌面PC的Web服务器软件），O'Reilly Media, Inc. 一直处于Internet发展的最前沿。

许多书店的反馈表明，O'Reilly Media, Inc. 是最稳定的计算机图书出版商——每一本书都一版再版。与大多数计算机图书出版商相比，O'Reilly Media, Inc. 具有深厚的计算机专业背景，这使得 O'Reilly Media, Inc. 形成了一个非常不同于其他出版商的出版方针。O'Reilly Media, Inc. 所有的编辑人员以前都是程序员，或者是顶尖级的技术专家。O'Reilly Media, Inc. 还有许多固定的作者群体——他们本身是相关领域的技术专家、咨询专家，而现在编写著作，O'Reilly Media, Inc. 依靠他们及时地推出图书。因为 O'Reilly Media, Inc. 紧密地与计算机业界联系着，所以 O'Reilly Media, Inc. 知道市场上真正需要什么图书。



译者序

算法，神秘而晦涩的词汇。算法，是计算机科学中最重要同时也是最基础的一环。从开始学习计算机，我们就深知，算法是整个计算机科学的核心。然而直至我们工作数年后，能够真正学好算法的人，却依旧是凤毛麟角。这并不是计算机教育的错，也不是计算机从业人员的错，更不是算法的错。长久以来，算法就像古老的咒语，算法背后高深的数学知识更让人望而生畏。其实，我们始终没有找到一条从理论走向实践的路。

在这里，我们很高兴能向大家介绍本书。它正是能够带领你学好算法的一本不可多得的好书。

本书的三位作者是伍斯特理工学院的教授，其中George T. Heineman毕业于达特茅斯学院和哥伦比亚大学，曾经获得过GE、IBM和AT&T的研究奖金，在软件工程方面有独到的研究。而Gary Pollice曾经供职于Rational Software、Sun等多家巨头，有着丰富的工业界经验，知道如何将学术和工业结合起来。Stanley M. Selkow毕业于卡内基梅隆大学和宾夕法尼亚大学，擅长图论和算法设计。本书由这三位伍斯特理工学院计算机理论专家合著，向我们展示了工业界和学术界对算法的不同看法以及如何高效地将理论和实践相结合。本书搭建了一条真正属于开发者的路。

本书的读者主要面向本科生以及程序设计人员，同样也适用于产品和项目管理人员。由于译者的知识和经验有限，翻译中难免有疏漏或错误，敬请广大读者谅解并批评指正。

杨晨 李明

2009年7月



作者简介

George T. Heineman是伍斯特理工学院计算机科学系的一名副教授,专注于软件工程方面的研究。他还参与编写了一本论述基于组件的软件工程方面的书:《Putting the Pieces Together》,由Addison-Wesley于2001年出版。George是2005年国际组件软件工程研讨会的议程主席。

Gary Pollice自称是一个乖戾的人(其实就是一个顽固的、坏脾气的老人),在工业界花费了35年来探索他究竟希望成为什么。但是在2003年,他毅然决定以不成熟之身从工业界转移到学术的殿堂。在这里他可以用激进的话语影响下一代软件工程师,“为你的客户开发软件”,“学会如何成为团队的一员”,“要思考软件设计、代码质量、优雅程度和正确度”和“只要你成为牛人,那么即使成为书呆子也无所谓啦”。

Gary是伍斯特理工学院的一位实践教授(这是一个头衔,也就是说他在成为一名教授之前有一份真实的工作)。由于他对一起工作数年的WPI的毕业生们印象非常好,所以决定来WPI做一名教授。他和妻子Vikki,以及两条狗Aloysius和Ignatius一起住在麻省中部。他一直在做和极客相关的事情。你可以通过他的WPI网页<http://web.cs.wpi.edu/~gpollice/>来了解他,还可以随意地给他留言,抱怨或者赞美本书。

Stanley Selkow是伍斯特理工学院计算机系的一名教授,于1965年在卡内基理工学院(现卡内基梅隆大学)获电子电气工程学士学位,1970年在宾夕法尼亚大学获电子电气工程博士学位。在1968~1970年间,他在马里兰州贝塞斯达的国家卫生研究所从事公众健康服务相关的工作。自1970年,他先后在田纳西的诺克斯维尔和麻省的伍斯特从事教员工作,他也在蒙特利尔、重庆、洛桑和巴黎做过访问学者。他的主要研究领域是图论和算法设计。

封面介绍

本书封面的动物是一只寄居蟹(Pagurus bernhardus)。在世界各地有超过500种寄居蟹。它们大多为水生,生活在珊瑚礁和潮池的盐水中。一些寄居蟹,尤其是在热带地区的,是陆生的。例如一种名为盗蟹的寄居蟹,它能够长到椰子那么大。陆生寄居蟹在它们的壳里面存放着少量的水,用来帮助呼吸以及保持腹部湿润。

寄居蟹和其他螃蟹不一样,它们不需要一个坚硬的属于自己的外壳,它们寻找食肉动物不能食用的腹足动物(例如蜗牛)的外壳作为庇护所。它们特别偏好玉黍螺和海螺的

壳。随着身体的长大，它们需要寻找更大的壳寄居。如果他们身体的任何部分暴露出来，那么会非常容易受到食肉动物的攻击；此外，如果没有一个合适的外壳，就会阻碍它们的成长，因为昆虫腹足动物的壳是有限的，所以竞争也是一个问题。

寄居蟹是十足（也就是说十只脚）甲壳动物。在它们的五对足中，第一对是钳子，或者是螯，较大的那个是它们用来防御和撕碎食物用的，而较小的是用来辅助进食的。第二对和第三对足帮助它们走路，最后两对足用于将它们固定在壳中。

寄居蟹具有明显的甲壳动物的特征：它们没有内骨架，而是有一个钙质的外骨骼。它们也有两只复眼，两对触角（它们用来感知味道和振动）并拥有三对口器。在触角的底部是一对触角腺，用来排出废物。

可以经常看见海葵附在寄居蟹的壳上。海葵以这种方式移动并食用寄居蟹的食物残渣，不过作为交换，海葵能够伪装寄居蟹以逃过海洋食肉动物的捕食，例如鱼和章鱼。其他的食肉动物也包括：鸟和其他的螃蟹，以及一些哺乳动物（例如人）。

寄居蟹号称“海洋的垃圾收集器”，它能够吃掉几乎所有的东西，例如海岸上腐烂的物质，因此它们在海岸清洁中扮演着一个非常重要的角色。作为杂食动物，它们的食物是极其多样化的，从虫子到有机废品（例如草和叶子）无所不有。

封面的图片来自于Johnson的《Library of Natural History》卷2。



目录

前言	1
----------	---

第一部分

第1章 算法真的很重要	11
-------------------	----

理解问题	12
如果需要，尽可能用实践检验	13
解决问题的算法	15
花絮	16
故事的寓意	17
参考文献	19

第2章 算法的数学原理	20
-------------------	----

问题样本的规模	20
函数的增长率	22
最好最坏和平均情况下的性能分析	26
性能指标	30
混合操作	43
基准测试	43
最后一点	45

参考文献	46
------------	----

第3章 模式和领域..... 47

模式：一种交流语言	47
算法模式的格式	49
伪代码模式的格式	49
设计格式	50
基于经验的评价格式	52
领域和算法	52
浮点计算	54
手动内存分配	58
选择一门编程语言	60
参考文献	61

第二部分

第4章 排序算法..... 65

概述	65
插入排序	71
中值排序	75
快速排序	85
选择排序	92
堆排序	93
计数排序	98
选择排序算法的标准	105
参考文献	110

第5章 查找..... 112

概述	112
顺序查找	113
二分查找	118
基于散列的查找	122
二叉查找树	135

参考文献	141
第6章 图算法	143
概述	143
深度优先搜索	149
广度优先搜索	155
单源最短路径	159
所有点对最短路径	170
最小生成树算法	173
参考文献	176
第7章 人工智能中的寻路	177
概述	177
深度优先搜索	185
广度优先搜索	194
A*搜索	198
比较	208
Minimax	211
NegMax	216
AlphaBeta	219
参考文献	226
第8章 网络流算法	229
概述	229
最大流	232
二部图匹配	240
在增广路上的深入思考	244
最小开销流	246
转运问题	248
运输问题	248
任务分配问题	250
线性编程	250
参考文献	251

第9章 计算几何	252
概述	252
凸包扫描	261
线段扫描	269
最近点查询	280
范围查询	289
参考文献	296

第三部分

第10章 最后的招数	299
另类算法	299
近似算法	300
离线算法	300
并行算法	300
随机算法	301
结果可能出错但可以衰减错误率的算法	308
参考文献	311
第11章 尾声	312
概述	312
原则：了解数据	312
原则：将问题分解至更小的问题	313
原则：选择正确的数据结构	314
原则：空间换时间	315
原则：如果没有显而易见的解法，使用搜索	315
原则：如果没有显而易见的解法，将问题归约为另一个有解的问题	316
原则：编写算法难，测试算法更难	317

第四部分

附录 基准测试	321
----------------------	------------



前言

就像《黑客帝国》里面的Trinity所说的：

“Neo，是这个问题驱使着我们，是这个问题带你来到这儿。”
你知道这个问题，我也是。

作为本书的作者，我们将回答引领你到此的问题：

我能够使用某个算法解决我的问题吗？如果可以，那么怎么实现呢？

你也许并不需要理解一个算法为什么是正确的。如果你需要，那么请看看其他的资料，例如1180页的算法圣经——《算法导论》，作者是Thomas H. Cormen等（2001）。在那本书中你会了解到推论、定理以及证明；你也会从一些练习题和逐步递进的样例中看到算法是如何执行的。也许你会惊奇地发现，在算法导论中你找不到任何的代码，仅仅是一些伪代码的片段，伪代码是无数的算法教科书用来阐述算法的高级描述手段。在课堂上，这些教科书是非常重要的，但是在实际软件开发中，它们却起不到应有的作用，因为这些书假定伪代码都能够直接变成实际代码。

我们希望经验丰富的程序员在寻找问题的解决方案时，能够频繁参考本书。作为一名程序员，你每天要解决的问题都能在这里找到解决方案。在软件中，算法是决定成败的关键因素，在这里你能够了解到哪些决定能够改善关键算法的性能，也能够找到适合你的需求和解决方案的实际代码。

所有的算法都有实现，并且都使用测试工具经过仔细测试，以确保其正确性。而且，它们有足够的代码文档，能在这本书的代码库附录中找到它们。我们严格地遵照一系列的原则来设计算法、实现算法，以及编写这本书。如果这些原则对你很有意义，那么这本书也会同样有用。

原则：使用实际代码，而不是伪代码

为了计算最大网络流，一个实践者应该做些什么才能将图P-1的Ford-Fulkerson算法描述转换成实际代码呢？

Ford-Fulkerson Algorithm:

Input Graph G with flow capacity c , a source node s , and a sink node t

Output A flow f from s to t which is a maximum

1. $f(u,v) \leftarrow 0$ for all edges (u,v)
2. **while** (there is a path p from s to t in G_f such that $c_f(u,v) > 0$ for all edges $(u,v) \in p$) **do**
3. Find $c_f(p) = \min \{ c_f(u,v) \mid (u,v) \in p \}$
4. **foreach** edge $c_f(u,v) \in p$ **do**
5. $f(u,v) \leftarrow f(u,v) + c_f(p)$ // Send flow along the path
6. $f(v,u) \leftarrow f(v,u) - c_f(p)$ // The flow might be "returned" later
- end**

图P-1：教科书中常见的伪代码

图中的算法描述来自于维基百科 (http://en.wikipedia.org/wiki/Ford_Fulkerson)，这个描述与《算法导论》上的伪代码极其相似。最好还是不要期望一个软件的开发者能够根据这个Ford-Fulkerson算法的描述开发出实际的代码。翻到第8章，对比一下我们的代码。我们只使用有注释的，并且是精心设计过的代码。在你自己写的代码或者软件系统中使用我们提供的现成代码，或者这些代码的逻辑吧。

一些算法教科书确实有完整的C或者Java代码。但是这些教科书的目的是教初学者编程语言，或者是解释如何实现抽象数据类型。而且代码都只是在页面的狭窄边栏，作者通常都会忽略注释和错误处理，或者使用在实际应用中不会用到的快捷方法。我们相信程序员能够从有注释的，并且是精心设计过的代码中学到更多的东西，这就是我们为什么做如此多的工作来开发算法的实际解决方案。

原则：将算法和将要解决的问题分开

如果不和具体的问题联系起来，我们很难“泛化地”实现一个算法。我们反对那些给出了算法完全实现的书，这些书上的实现将泛化问题和特定问题纠缠在一起，从而很难看出算法的原始结构。更糟糕的是，很多可用的实现依赖于一些特定的数据存储方式，虽然这种方式能够容易被机器理解，但是很难被人理解。

我们将会为泛化的问题和特殊的问题各设计算法的实现。例如，在第7章，当我们描述A*搜索算法的时候，我们用了一个例子，叫做八数码问题（在一个 3×3 格子的棋盘上移

动编号为1~8的小方块)。A*算法的实现依赖于一系列良好定义的接口。因为有良好定义的接口,实现这些接口的类能够清晰地封装八数码这类特定问题的细节。

在本书中,我们使用了大量的编程语言,并且遵循一种严格的设计规范,使得代码可读性高并且生成高效的解决方案。由于我们具备软件工程背景,所以根据泛化的算法和特定领域的解决方案设计一个清晰的接口是一件很容易的事情。按照这样的流程编码,产生的软件是易于测试、维护并且能够根据面临的问题即时扩展。另外一个好处是读者能够更加容易地阅读和理解算法的描述和结果。当选择了一个算法,我们将会告诉读者,如何将我们编写的可读并且高效的代码转换成高度优化(虽然稍稍降低了可读性)并且性能优秀的代码。毕竟,优化是在问题已经解决之后才进行的,而且客户需要的是运行更快的代码。即便如此,我也认为我们需要听从C. A. R. Hoare的建议:“过早的优化是一切问题之源”。

原则: 仅仅讲述足够的数学

很多算法注重专门关注于证明算法的正确性,并且抽象地解释其细节。我们关注的却是如何将算法应用到实践中去。最后我们才会介绍一些数学知识,这些数学知识都是为了读者能够更好地理解数据结构和解的控制流程。

例如,一个人需要理解在很多算法中使用的集合和二叉树的性质。但是,没有必要证明二叉树的高度如何得到,并且解释红黑二叉树是如何平衡的。如果你需要了解这些细节,请阅读《算法导论》的第13章。我们仅仅是需要才会解释结果,如果读者需要理解如何证明结论,我们将会告诉读者在哪儿能够找到数学证明。

在这本书你将会学习到使用一些关键术语和分析技术,并且基于数据结构的功能来区分算法行为。

原则: 用经验来支持数学分析

在这本书中,我们从数学角度来分析算法的性能,以帮助程序员了解在哪种情况下算法能够得到最好的性能。我们将会提供现成的代码样例,在相关代码库中,有大量的JUnit (<http://sourceforge.net/projects/junit>) 测试样例为每个算法的实现提供了文档。我们也会生成基准性能数据,供分析算法性能时参考。

我们将每个算法归到一个特定的性能族中,并且提供基准测试数据来得到算法的性能,支持我们的分析结论。有一些算法是那些具有数学背景的算法设计人员证明能够非常高效但是却不可能实现的,我们需要避免使用这样的算法。我们将在各种平台上执行我们的算法,以证明算法的高效并不依赖于特定平台,而是由于其优秀的设计。

附录包含了我们采用的基准测试方法的全部细节，并且这个基准测试能够独立地验证书中描述的所有性能结论。我们能够给你的建议在开源社区非常常见：“你得到的利益也许会不一样”。虽然你不可能准确地复制我们的结论，但是你能看出我们描述的趋势。我们鼓励你在决定使用哪个算法的时候使用我们的这种基于经验的方法。

目标读者

如果你被困在一个沙漠孤岛上，并且只能选择一本算法书，我们推荐Donald Knuth的《计算机程序设计艺术（卷1~3）》（1998）。Knuth在这本书中描述了大量的数据结构和算法，并且进行了精巧的处理和分析。这本书包含了大量的脚注和练习，并且能够帮助一名程序员在接下来的时间中保持活力和竞争力。这些练习非常有挑战性，并且你能够直接接触到Knuth的思想。

但是你没有被困在孤岛上，对吗？你接手了一些劣质的代码，而且这些代码必须在周五前进行改进，你需要知道如何去处理才行！

在你面对一个算法问题，需要解决一个特定的问题或者对现有解决方案进行改进的时候，我们希望本书能够成为你的第一选择。为解决各种问题，我们广泛地讨论了现存的算法，并且遵循以下原则：

- 我们使用模式化的统一格式来描述每一个算法，进行每一次讨论并解释算法的重点。正因如此，我们的书才如此易读。我们才能够看出相似的设计会对不同的算法产生什么样的影响。
- 在这本书中，我们使用了不同的编程语言（包括C、C++、Java还有Ruby）来描述算法。得益于此，我们能够使用你熟悉的编程语言对算法进行详细的讨论。
- 我们描述了算法的期望性能，并且根据经验提供了支持这些结论的证据。无论你相信数学还是相信实际的运行时间，你都会被我们说服。

对软件工程师、程序员以及设计师来说，我们希望这本书能够派上用场。为了达到目标，你需要参考大量的关于实际解决方案和算法的资料，才能够解决手头的实际问题。你已经知道了如何用多种语言编写一个程序。你也知道了计算机科学的关键数据结构，例如数组、链表、栈、队列、散列表、二叉树还有有向图或者无向图。你不需要实现这些数据结构，因为它们大都在库函数中可以找到。

我们希望你从这本书中学到如何选择并且测试解决方案以快速高效地解决问题，也能够学到一些高级数据结构和一些使用标准数据结构的新方法，来改善程序的性能。在选择算法时，如果你可以知道什么样的决定可以改善解决方案的效率，那么就能够提高你解决问题的能力。

本书组织方式

本书分为三个部分。第一部分（第1章～第3章）介绍了算法的必要数学基础，以便于读者理解本书的描述。在每个算法的论述中，我们使用了一种模式化的格式。我们仔细地设计这个格式，确保前后一致。第二部分（第4章～第9章）介绍了一系列的算法。这些章节的每个独立部分都是一个完备的算法描述。

第三部分（第10章和第11章）为那些感兴趣的读者提供一些较为高深的资源。当没有一个高效的解决方案来解决问题，而且“最后的线索”为解决问题提供了有意义的线索的时候，这个部分就能够告诉我们如何利用这些线索来解决问题。最后我们以一个重要领域的讨论结束本书。这个讨论在第2章之所以被忽略，是因为它的内容太高深，太过前沿，甚至还没有被证明。第四部分包括了一个附录，这个附录描述了本书每章中对算法进行评测的方法以及数学分析。在业界，这是一种标准的基准测试方法，但是几乎没有算法教科书介绍这个方法。

本书体例

在印刷上的一些例行惯例：

代码 (Code)

这些代码都是直接取自代码库，是现实中使用的代码。

斜体 (*Italic*)

表示这个术语用于描述算法和数据结构。同样在伪代码描述中表示变量。

等宽字体 (Constant Width)

表示实现中的软件元素，例如Java类、C语言的数组名称以及常量 (`true`或者 `false`)。

小型大写字母 (SMALL CAPS)

表示算法名称。

在本书中，我们引用了大量的书籍、文章和网站。这些引用都用括号标示出来，例如 (Cormen等, 2001)，每一章最后部分列出本章所使用的参考文献。正文中的引用列出作者的名字和文献的年份。

本书中的所有URL都在2008年8月时验证有效，而且我们抛弃了那些看起来不久就会失效的URL。短URL，例如<http://www.oreilly.com>，直接在文本中使用；否则，这些URL会放入参考文献中。

代码使用说明

本书的目的是帮助你更好地完成工作，一般来说，你可以在程序和文档中直接使用本书代码。除非你需要重构这些代码的重要部分，否则你不需要联系我们获得许可。例如，你可以编写一个使用了本书中一些代码的程序而不需要许可。销售或者分发来自O'Reilly出版书籍中的样例不需要许可。引用本书来回答问题或者引用样例代码也不需要许可。在你的产品文章中过度地使用本书中的代码也不需要许可。我们希望，但是不要求标明归属。一个归属说明通常包括标题、作者、出版商和ISBN。例如：“George T. Heineman, Gary Pollice和Stanley Selkow编写的《Algorithms in a Nutshell》。版权属于George T. Heineman, Gary Pollice和Stanley Selkow, 978-0-596-51624-6”。

如果你觉得你需要使用代码的情况超出了我们的允许范围，那么请联系：permissions@oreilly.com。

联系我们

对于本书，如果有任何意见或疑问，请按照以下地址联系本书出版商：

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街2号成铭大厦C座807室（100035）
奥莱利技术咨询（北京）有限公司

本书也有相关的网页，我们在上面列出了勘误表、范例以及其他一些信息。你可以访问：

<http://www.oreilly.com/catalog/9780596516246>（英文版）
<http://www.oreilly.com.cn/book.php?bn=978-7-111-28674-5>（中文版）

对本书做出评论或者询问技术问题，请发送E-mail至：

bookquestions@oreilly.com

希望获得关于本书、会议、资源中心和O'Reilly网络的更多信息，请访问：

<http://www.oreilly.com>
<http://www.oreilly.com.cn>

致谢

我们感谢书籍的审阅者，感谢他们对于本书细节的关注以及所给出的建议，这些建议提高了本书的质量。他们是：Alan Davidson、Scot Drysdale、Krzysztof Duleba、Gene Hughes、Murali Mani、Jeffrey Yasskin和Daniel Yoo。

George Heineman感谢那些带领他走入算法世界的人，包括Scot Drysdale教授（达特茅斯学院），Zvi Galil（哥伦比亚大学）。同样，George也感谢他的妻子Jennifer，他的孩子Nicholas（他一直想要知道爸爸在写什么便条）和Alexander（出生于我们书籍定稿时）。

Gary Pollice感谢他的妻子陪伴他走过了40年的美好时光。他同样感谢WPI计算机科学系提供给他伟大工作和幽雅的环境。

Stanley Selkow感谢他的妻子Deb。这本书是在他们相伴走过的时光中的又一个结晶。

参考文献

Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to Algorithms*, Second Edition. McGraw-Hill, 2001.

Knuth, Donald E., *The Art of Computer Programming*, Volumes 1–3, Boxed Set Second Edition. Addison-Wesley Professional, 1998.



第一部分

第1章 算法真的很重要

第2章 算法的数学原理

第3章 模式和领域





算法真的很重要

算法真的很重要！在哪些情况下使用哪种算法会给你编写的软件的性能带来巨大的差异。如果你不信的话，那么请看看Gary是怎么通过一些小小的分析，选择了一个正确的算法，扭转了整个败局（注1）。

Gary曾经在一个有着很多天才软件开发者的公司工作。和大多数有着很多天才的组织一样，这里有着层出不穷的伟大想法，这些天才们将这些伟大想法实现并且集成到了软件产品中。例如Graham，他从公司创立之初就加入了这个公司。Graham曾经思考过一个问题：如何知道一个程序是否存在内存泄漏，这是当时C和C++程序都存在的一个问题。如果一个存在内存泄漏的程序运行足够长时间，那么这个程序会由于内存耗尽而崩溃。任何一个存在此类问题的程序都不支持自动内存管理和垃圾回收。

Graham决定编写一个库，以包装操作系统的内存分配以及回收函数，例如`malloc()`、`free()`，还有他自己的函数。Graham的函数将每次内存的分配以及回收都记录在一个特定的数据结构中，这样在程序结束之后，程序员能够查询到这些记录。这些包装函数记录这些信息，并且调用操作系统函数来进行内存管理。Graham只花了几个小时就实现了这个库而且能够正常被调用执行。但是现在有一个问题：调用了Graham的库的程序运行得如此缓慢，以至于没有人会去使用它。这是真的名副其实地缓慢。你可以先启动程序，然后离开去喝一杯咖啡，或许喝一壶咖啡也可以，然后回来，你会看到程序还在缓慢地运行着。这种速度当然不能够接受。

为了解决这个问题，Graham很快就了解了操作系统以及内部工作原理。他是一个极其优秀的程序员，能在一个小时内写出其他程序员需要一天才能写出的代码。他已经在大学学习了算法、数据结构以及其他标准课程，所以这点事情肯定难不倒他。那么为什么使

注1： 本文涉及的人员以及组织名均被改变以保护隐私，避免不必要的法律纠纷。

用了这些包装的函数，程序会执行得如此缓慢呢？看来，这就是一个典型的知道如何使用程序运行，却不知道如何使用程序运行得快的案例。但是像很多具有创新精神的人一样，Graham已经在思考他的下一个程序，而不是回到他的那个内存泄漏程序中继续寻找问题。于是，他希望Gary检查一下这个库，看看能不能修复运行缓慢的问题。Gary是一个编译以及软件工程方面的能手，他擅长精炼代码，使得它们可供发布。

在开始深入代码之前，Gary希望先和Graham谈谈这个程序。这样的话，他能够更好地理解Graham如何构造他的解决方案以及为什么他选择了这样的实现方式。

注意：在继续读下去之前，想想如果你，你会问Graham一些什么。然后在接下来的部分，看看你是否能够得到和Gary相同的信息。

理解问题

解决问题最好从大局观开始：理解问题，找出潜在原因，然后深入挖掘细节。如果你觉得你知道原因，然后决定解决这个问题，那么你可能会出错，或者你也许不会发现其他更好的答案。Gary首先希望Graham能够描述这个问题以及他的解决方案。

Graham说他的目的是希望能够知道一个程序是否存在内存泄漏。他想知道这个问题答案的最好方法是记录所有被程序分配的内存，不管这些内存是否在程序终止之前被释放掉了，同时也记录用户的程序在哪儿请求内存分配。他的解决方案是构建一个包含三个函数的小型库。

`malloc()`

一个包装了系统内存分配函数的函数。

`free()`

一个包装了系统内存回收函数的函数。

`exit()`

一个包装了系统退出函数的函数。

那就写一个使用了这个库的程序来进行测试吧，在程序中调用库中自定义函数而不是系统函数。自定义的`malloc()`和`free()`将会记录每一次内存分配和释放。当程序测试结束的时候，如果记录中，每一个内存分配接下来都有相对应的内存释放，那么就证明没有内存泄漏。如果存在内存泄漏，那么Graham的函数将会记录相关信息以供程序员参考。当调用自定义的`exit()`函数的时候，在程序实际退出之前，将会显示内存的分配释放记录。Graham将他的解决方案用图表示出来，如图1-1。

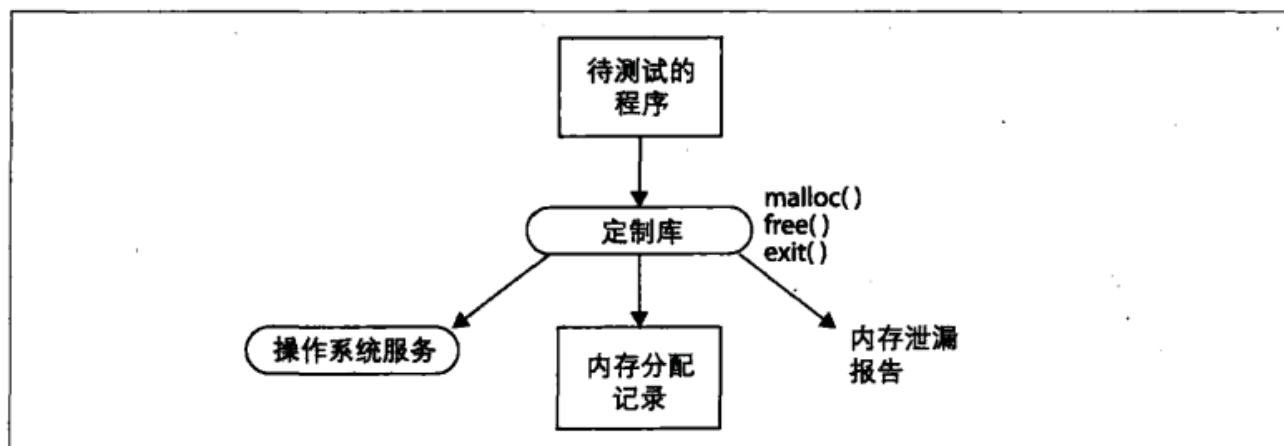


图1-1: Graham的解决方案

描述貌似已经很清楚了。除非Graham在他包装了系统函数的代码中犯了一些相当严重的错误，否则还真的很难想象在包装的代码中存在性能问题。如果存在的话，那么所有程序的缓慢程度都应该与调用的次数成比例。于是Gary询问Graham测试过的程序是否存在性能差异。Graham解释说，通过性能剖析发现，那些小的程序通常能够在一个勉强可以接受的运行时间内，不管这些程序是否有内存泄漏。但是，那些需要进行大量处理工作和有内存泄漏的程序运行起来慢得不像话。

如果需要，尽可能用实践检验

在继续深入之前，Gary希望能够较好地了解程序的运行状况。他和Graham坐下来一起写了一些小程序，通过调用这个库来检查这些程序是如何运行的。也许他们能够更好地明白是什么导致了这个诡异的问题。

注意：你会进行哪些实验呢？你的程序将会是什么样？

Gary和Graham的第一个测试程序（程序A）如例1-1所示。

例1-1：程序A代码

```
int main(int argc, char **argv) {
    int i = 0;
    for (i = 0; i < 1000000; i++) {
        malloc(32);
    }
    exit (0);
}
```

他们运行了这个程序然后等待结果。这个程序运行了数分钟。即便在当时的计算机速度比较缓慢的情况下，这个运行时间也很明显不可接受。当程序运行结束的时候，记录

显示有32MB的内存泄漏。那么如果程序中所有分配的内存都被释放了会出现什么情况呢？他们做了一点小修改，这就是程序B，如例1-2所示。

例1-2：程序B代码

```
int main(int argc, char **argv) {
    int i = 0;
    for (i = 0; i < 1000000; i++) {
        void *x = malloc(32);
        free(x);
    }
    exit (0);
}
```

编译并且运行程序B，程序B在几秒内就运行结束了。Graham确信这个问题和程序结束的时候没有释放的内存大小相关。但是他不知道问题出在哪。他搜寻了整个代码，花费了数个小时却也找不出问题在哪儿。Gary并不像Graham那样确信问题是与内存泄漏的数目相关。他建议做更多的实验，并且对程序做了另一次的修改，将所有的内存释放操作集中到程序的最后。如例1-3所示。

例1-3：程序C代码

```
int main(int argc, char **argv) {
    int i = 0;
    void *addrs[1000000];
    for (i = 0; i < 1000000; i++) {
        addrs[i] = malloc(32);
    }
    for (i = 0; i < 1000000; i++) {
        free(addrs[i]);
    }
    exit (0);
}
```

这个程序就像第一个程序那样缓慢地运行着！这个例子证明了性能问题并不是由内存泄漏的数量导致的。不仅如此，这个程序让Gary明白了问题原因的真正所在。并不是程序结尾的内存分配次数导致性能问题，而是任意时刻没有被释放的内存的最大数量严重影响了整个程序的性能。如果程序本身的内存泄漏并不是影响问题的唯一因素，那么Graham维护内存分配以及释放的记录的方法就存在一些问题。在程序B的执行过程中，任何时刻都没有超过32字节的未释放内存。但是第一个和第三个程序运行的时候却有100万个内存分配操作没有相应的释放操作。分配和释放内存并不是关键，所以问题肯定和Graham记录内存分配和释放请求的代码有关。

Gary询问了Graham他是怎么记录分配的内存的。Graham回答说他使用了一个二叉树来记录，每一个节点都记录了指向子节点的指针（若有的话）、指向分配内存的地址、分配的大小以及程序在哪儿进行的分配请求。Graham还补充道他是使用内存地址作为节点的键值，因此不能重复，这个方法使得插入和删除内存分配的记录更加容易。

使用二叉树通常比简单地使用有序链表更加高效。假设有一个包含了 n 个元素的有序链表，这个表中的每个元素被请求的概率是相等的。那么一个寻找表中一个元素平均将要花费 $n/2$ 次比较。从表中插入或者删除元素平均也需要大约 $n/2$ 次操作。计算机科学专业的教科书描述这些操作（查找、插入和删除）的性能为 $O(n)$ ，实际上这些操作的数量两倍就是和这个链表长度有关，进行这些操作花费的时间期望也是这么多（注2）。

使用一棵二叉树能够将上述操作降到 $O(\log n)$ 的花费，虽然代码可能有一些难以编写和维护。跟链表的时间花费不一样，使用了二叉树，这些操作的花费仅仅是常数量的增长。当处理1 000 000个元素的时候，我们只需要平均检查20个元素，相比有序链表需要500 000次操作，效率之高可想而知。如果键值是平均分布在树中的，那么使用二叉树是一个很好的选择。如果键值不是平均分布的，那么树会变得扭曲，不平衡，在某种程度上这个数据结构就会认为不适合用来查找。

知道了一些关于二叉树以及如何工作的知识后，Gary询问Graham一个价值64 000美元（毕竟这个数字是对数）的问题：“你对这个二叉树进行了平衡处理吗？”Graham非常惊讶，因为他是一名优秀的软件工程师，不可能不知道这样简单的东西。“没有，为什么我需要做这个？这个操作使得代码更加复杂了。”不过事实的确是由于Graham没有做平衡处理，使得这棵树右倾到操作时间近似线性的程度，这种情况下更像是一个有序链表而不是一棵二叉树。想知道为什么吗，看看图1-2中的两棵二叉树。在a树中顺序插入数字1~15。它的根节点值为1，从根节点出发到达值为15的节点需要经过14个节点。b树插入了相同的数字，不过是按照这样的顺序： $\langle 8, 4, 12, 2, 6, 10, 14, 1, 3, 5, 7, 9, 11, 13, 15 \rangle$ 。在这种情况下，根节点的值是8，不过从根节点到其他节点的路径在3个节点或者更少。我们在第5章将会看到，查找时间直接和最长路径的长度相关。

解决问题的算法

如果一棵二叉查找树的根节点到任意一个叶子节点的路径长度都基本相等，那么这棵树就是二叉平衡查找树。定义 $depth(L_i)$ 为从根节点到叶子节点 L_i 的路径长度。如果一棵二叉树具有良好的平衡度，那么对任意两个叶子节点 L_1 和 L_2 来说， $depth(L_2)$ 和 $depth(L_1)$ 的差的绝对值，即 $|depth(L_2) - depth(L_1)| \leq 1$ ，同样对于任何一个叶子节点 L_i 来说， $depth(L_i) \leq \log(n)$ 。Gary翻阅了他的算法书籍，决定对Graham的代码进行修改，用更加平衡的红黑树记录内存的分配。红黑树（Cormen, 2001）是一个平衡二叉树的高效实现。在红黑树中，给定两个叶子节点 L_1 和 L_2 ， $depth(L_2)/depth(L_1) \leq 2$ ；同样对任意叶子节点 L_i 来说， $depth(L_i) \leq 2 * \log_2(n+1)$ 。也就是说，一棵红黑树是大致地平衡的，这确保了在红黑树中，不存在一条路径，其长度是另外一条路径的两倍。

注2： 本书的第二部分将具体介绍各种算法。

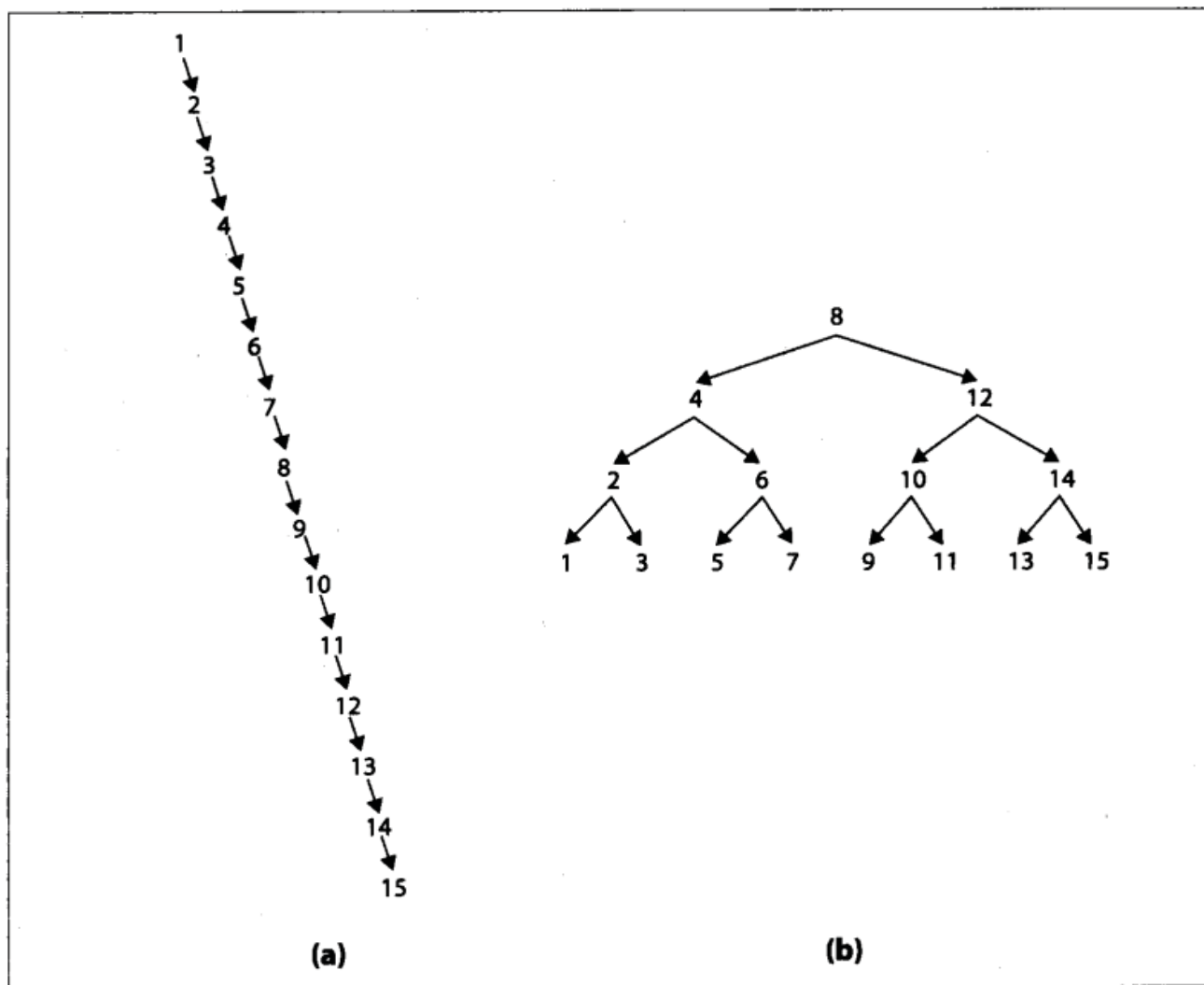


图1-2：构建两棵二叉树样例

Gary花了几个小时进行了修改和测试。当他完成之后，他将结果演示给Graham看。他们使用新的库，重新运行了上面的三个程序。程序A和程序C仅仅只是比程序B多花了几毫秒而已。性能很明显地大幅改善了，反映在速度提高了近5000倍。这样的性能改善是可想而知的，你可以看到平均访问节点的数目从500 000个降到了20个。事实上，这是数量级的降低：你也许期望能够加速25 000倍，但是对树进行平衡的操作开销降低了部分性能。尽管如此，性能的改善仍然是惊人的，Graham的内存泄漏检测程序（包括Gary的修改）将再下一个产品中发布。

花絮

已知使用红黑树的效率，原生`malloc()`的实现可能会使用这个结构吗？毕竟，内存分配在某种程序上必须维护分配的区域集合，以使得这些区域将来能够被安全地释放。同样，注意到上述程序每次请求32字节的内存分配，那么每次请求的内存大小是否会影响

malloc()和free()请求的性能? 为了研究一下malloc()的行为, 我们又进行了一系列的试验。首先, 我们记下分配4096个内存块的时间, 内存块的大小 n 从1字节取到2048字节。然后我们看看释放相同大小的内存需要多久, 不过将才需三种策略来释放。

freeUp

按照分配的顺序释放, 和程序C类似。

freeDown

按照分配的顺序逆序释放。

freeScattered

一次性释放所有内存。

对于每个 n 的值, 我们进行100次试验, 并且去掉最好的和最坏的成绩。图1-3显示了剩下98次试验的平均结果。正如我们所预料的, 分配操作的开销与 n 的增长呈线性关系。令人惊讶的是, 释放内存的方式在性能上有比较大的差异。freeUp的性能最好, freeDown比freeUp慢4倍左右。

经验表明, 现在还不能回答是否原生malloc()以及free()使用了二叉树(无论是否平衡!)来记录信息。为什么内存释放的顺序会引起性能的差异, 如果不查看free()的源代码, 这个问题没有一个简单的解释。

我们给出这个例子是有两个目的。第一, 内存分配以及释放背后的算法是惊人的复杂, 这些算法也高度依赖于操作系统(也就是复杂的计算机)的特性。我们在整本书中将会看到, 不同的算法都有他们自己最佳使用的情况。开发者能够根据问题的特殊信息使用相应的算法来改善性能。第二, 我们在整本书中描述了不同的算法, 并且解释了为什么一个算法会比另外一个算法表现更好。我们还是坚持使用数学来解释这一切。

故事的寓意

上面那个故事是真实的。你要知道, 算法真的很重要。你也许会问树平衡算法真的是这个问题的优化方案吗? 这是一个很好的问题, 对于问这个问题的人, 我们会问另外一个问题: 它确实很关键重要吗? 对于一个问题来说, 寻找正确的算法就像是寻找正确的解决方案一样。现有的算法已经可以很好地工作了, 我们无需寻找最完美的解决方案。当选择一个解决方案的时候, 你必须在它的开销和带来的好处之间寻找一个平衡点。Gary的实现方案是能够被改进的, 或者优化代码, 或者采用不同的算法。但是, 内存泄漏检测软件的性能不仅是可接受的, 而且还必须是能够满足预期使用的, 任何额外的改进将会产生没有收益的开销。

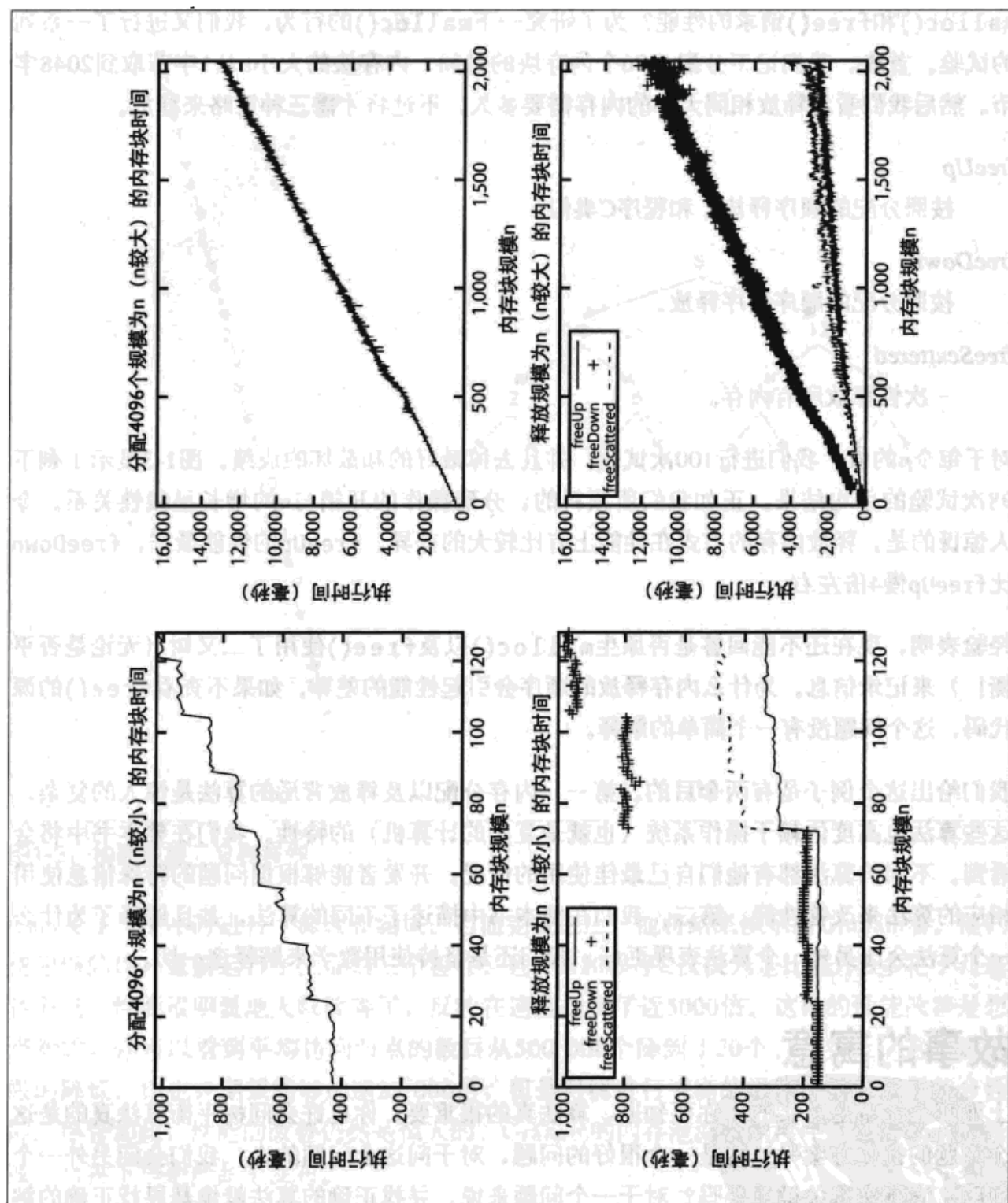


图1-3: malloc/free请求的性能分析

按需选择可以接受的算法，这种能力是非常重要的，每一个优秀的软件开发者都应该具备。并不需要你能够对算法进行详细的数学分析，但是你必须能够理解这些分析。并不需要你发明新的算法，但是你要知道哪个算法可以解决手头的问题。本书就是希望能够帮助你提高这种能力。当你具备这种能力之后，你的软件开发工具箱中就又多了一件软件开发的利器。

参考文献

Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein,
Introduction to Algorithms, Second Edition. McGraw-Hill, 2001.





在解决问题之前，我们可以试着猜测下，在这个问题所依赖的平台（或者平台家族）和数据上，哪个算法将会获得最好的性能。计算一个算法的期望运行时间本质上是一个数学运算过程。在本章，我们将透过现象看本质，阐述隐藏在这些复杂计算内的数学工具以及使用方法。通过本章的阅读，读者应该能够理解本书将要使用的数学术语。

贯穿本章（实际上是贯穿整本书）的一个共同主题就是所有的假设和近似操作都会花费常数时间，但是最终我们得到抽象结论的时候，都会忽略这些常数时间。在所有的平台上，相比影响本书所述的算法性能的其他因素来说，这些常数都是小到可以忽略不计的。

问题样本的规模

问题样本是解决问题的程序所使用的数据集，在大部分问题中，随着已编码样本的规模的增长，程序的执行时间也不断增加。同时，紧凑地对样本数据进行编码（也许使用压缩技术）可能会不必要地降低程序的执行效率。寻找一条最优的样本编码方式是极其困难的，因为问题发生在现实世界，而且机器需要理解这种编码方式。我们需要合理地编码样本数据，以供计算机进行求解。让我们看看接下来边栏“编码的样本”中对数字 x 的两种表示方法。

我们评价算法时，尽量假设算法是否能够被有效实现不取决于问题样本的编码。在上的一个问题中，我们看到，虽然编码方式在规模上差不多，但是在查询奇偶性的这个关键操作上，它们的性能差异非常明显。

选择合适的问题样本编码取决于将会进行的操作。设计一个高效的算法通常是从选择一个合适的数据结构开始的，数据结构通常是在计算机中表示将要解决的问题。如图2-1所示。

样本编码

给你一个很大的数字 x ，要求计算 x 的二进制形式中1的个数的奇偶性（也就是说，1的个数是奇数还是偶数）。例如，如果 $x=15\ 137\ 300\ 128$ ，其二进制表示是：

$x_2=1110000110010000001101111010100000$

可以看到1的个数是偶数，我们考虑下面两种可行的编码策略：

x 的编码1：1110000110010000001101111010100000

x 的这个二进制表示方法使用了34个字节。注意到 $\log_2(x)$ 的值是33.82，因此这个编码方式是最理想的。但是为了计算1的个数的奇偶性，需要检查每一个位。这种编码方式下，计算奇偶性的最佳时间的增长与 x 的长度 n （ x 的对数）增长呈线性关系。

x 也能够用下面的方式来表示，使用 n 个位，再加上一个额外的校检位来表示1的数目的奇偶性。

x 的编码2：1110000110010000001101111010100000[0]

我们可以看到 x 的这种编码方式的最后一个位是0，这表示1的个数为偶数个（偶数用0表示）。对于这种表示法，需要35个字节。在两种编码方式中，样本编码的规模都随 x 的值对数增长。但是，在第一种编码方式中，计算奇偶性最佳算法的时间增长情况与 x 的编码规模的增长呈对数关系，但是第二种编码方式中，最佳算法只需要花费常数的时间，而且与 x 编码规模无关。

我们来看看下面这段17世纪由松尾芭蕉所写的俳句：

古池や蛙飛込む水の音

这首诗可以表示为：

编码1：30字节Unicode序列：

E547A91CCB1A071A0908E89B4CBA5469BE85F2F1C68280732C168E381573A53EB3

编码2：40字节罗马音字符串：

“furu ike ya kawazu tobikomu mizu no oto”

编码3：一个 3×18 的字符数组，用英文表示：

o	l	d		P	o	n	d										
a		f	r	o	g		j	u	m	p	s		i	n	t	o	
t	h	e		S	o	u	n	d		o	f		w	a	t	e	r

图2-1：问题样本的复杂编码

由于不能够限定问题样本的规模，所以我们假设样本的编码使用一种简洁的方式，可以大致被接受的。当对 n 个数字进行排序时，我们根据惯例在平台上用一个字来表示数字，这样样本的规模就是 n 。可能有些数字需要用不止一个字来表示，所以样本的规模也会受到相应的影响。例如，相比使用32位存储整数的类似算法，使用64位来存储整数的算法需要花费两倍时间。

为了存储数据集，大多数编程语言都支持数组，数组是连续的内存区域，这些区域都可以通过一个整数索引 i 直接快速存取。数组是一维的，每个元素大小用一个字来存储，例如整数数组，布尔数组，或者字符数组，字的大小是平台相关的。有些数组是多维的，这样能够更丰富地表示数据，如图2-1所示，“编码对性能的影响”指出了编码能够影响一个算法的性能。

由于编程语言和平台的巨大差异，算法研究人员认为，即使给定编码方式，计算出精确的性能开销也是不可能的。因此，他们假定，如果一些算法的性能开销仅仅是常数倍差异，那么这些性能开销可以被认为是渐进等价的。但是这个假定在现实世界中是不可行的（谁会愿意将自己的应付账单乘以1000呢？），因此只是在比较算法的时候作为通用的手段。当实现一个算法的时候，对细节的重视反映在关注常数倍差异。

函数的增长率

一个广为接受的描述算法的方法是使用执行时间的增长率作为问题样本规模的函数。但是这种方法是抽象地描述一个算法的性能，忽略了细节问题。所以，在使用的时候需要对抽象背后的细节有一个清醒的认识。

程序都必须运行在某个平台上，在这里，平台的广义包括：

- 程序运行的计算机，包含了CPU、数据缓存、浮点运算单元以及其他芯片。
- 使用的语言、编译器/解释器以及生成代码的优化设置。
- 操作系统。
- 后台运行的其他进程。

一个基本的假设是：上述组成平台的任何条件的改变，都会改变程序的执行时间，但只是在原有的时间上乘以一个常数因子。在这里，我们简要地讨论一下顺序搜索算法，这个算法会在第5章讲到。顺序搜索算法会顺序搜索列表中的所有 n 个元素，直到找到想要的值 v 。现在，我们假设：

- 列表中有 n 个不同的元素。
- 将要寻找的元素 v 在这个列表里面。

编码对性能的影响

假设一个程序需要处理元素周期表，有三个频繁的操作，它们是：a) “第 N 个元素的原子量是多少？”；b) “ x 元素的原子数是多少？”；c) “第 N 个元素的名字是什么？”。一个有趣的挑战是：2008年1月的时候，第117个元素还仍未发现，但是第118个元素已经被发现，叫做Ununoctium。

周期表的第一种编码：两个数组，`elementName[]`，存储元素的名字，还有`elementWeight[]`，存储元素原子量。

周期表的第二种编码：用一个长度为2626个字符的字符串来表示整个周期表。最开头的62个字符是：

1 H Hydrogen 1.00794

2 He Helium 4.002602

3 Li Lithium 6.941

为了理解编码对于性能的影响，我们做了32次实验，每次实验包含100 000个随机请求（包括无效的）。我们舍弃了最好的和最坏的，留下了30次实验的结果，下面这张表表示了余下30次实验执行时间的平均值（和标准差），用毫秒表示：

	原子量查询	元素序号查询	元素名称查询
编码1	2.1 ± 5.45	131.73 ± 8.83	2.63 ± 5.99
编码2	635.07 ± 41.19	1050.43 ± 75.60	664.13 ± 45.90

就像预期的那样，编码2的性能比编码1的性能差很多，因为采用编码2的时候，每一次请求都要执行一次字符串操作。编码1能够有效地处理原子量查询和元素名称查询，但是元素序号查询需要对整个表进行查找，因为表是元素序号无序的。

这个例子告诉我们，不同的编码会在执行时间上可能产生巨大的差异。同样也告诉设计人员必须根据将要进行的操作选择合理的编码来优化性能。

- 列表中每个元素可能为 v 的概率都是相等的。

通过计算顺序搜索算法平均搜索多少个元素，我们能大致了解这个算法的性能情况。因为我们知道 v 肯定在这个列表里面，而且每个元素可能为 v 的概率都是相等的，用数学语言描述就是，对于一个包含有 n 个元素的列表，平均搜索的元素个数 $E(n)$ 就是所有次数总共搜索元素个数的和除以搜索次数。

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

因此，上述假设作为前提的条件下，顺序搜索算法搜索了这个表中的大约一半元素。如果列表的元素个数加倍，那么顺序搜索会大约搜索原来两倍的元素；期望的搜索数目是 n 的线性函数。也就是说，期望搜索数目可以用 $c*n$ 来表示， c 为常数。这里的 c 为 $1/2$ 。关于性能分析的一个基本的观点是，在长时间的运行中，常数 c 是不重要的，因为最重要的影响性能的因子是问题样本的规模 n ，随着 n 越来越大，那么错误率就如：

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

变得不再重要，实际上，它们的比值接近1说明了：

$$\lim_{n \rightarrow \infty} \frac{(\frac{1}{2}n)}{(\frac{1}{2}n + \frac{1}{2})} = 1$$

虽然对于比较小的 n 来说，预计错误会是致命的。但是在上下文中，我们所谓的顺序搜索算法期望搜索数目的增长率是线性的。所以，在大样本规模情况下，我们可以忽略作为乘数的常数，而主要关注样本的规模。

依据增长率这个抽象的概念选择算法时，我们需要注意以下问题。

常数的问题

这就是为什么我们需要使用超级计算机和定期升级我们的计算机。

n 并不总是非常大

在第4章我们将会看到，快速排序执行时间的增长率比插入排序执行时间的增长率要低。但是在小数据集上，插入排序表现得比快速排序要好。

算法的增长率决定了算法在逐渐增长的大数据集上的性能表现。我们在一个更复杂的例子上使用这个基本的原理。

给定一个排序任务，对四个排序算法进行评价。这个任务是对 n 个随机字符串进行排序。 n 的取值从1~512，对于每个 n 的值，我们都将进行50次实验，并且舍弃掉最好和最坏的实验结果，图2-2的图描述了剩下48次实验的平均运行时间（毫秒）。这些结果的差异实在令人惊异。

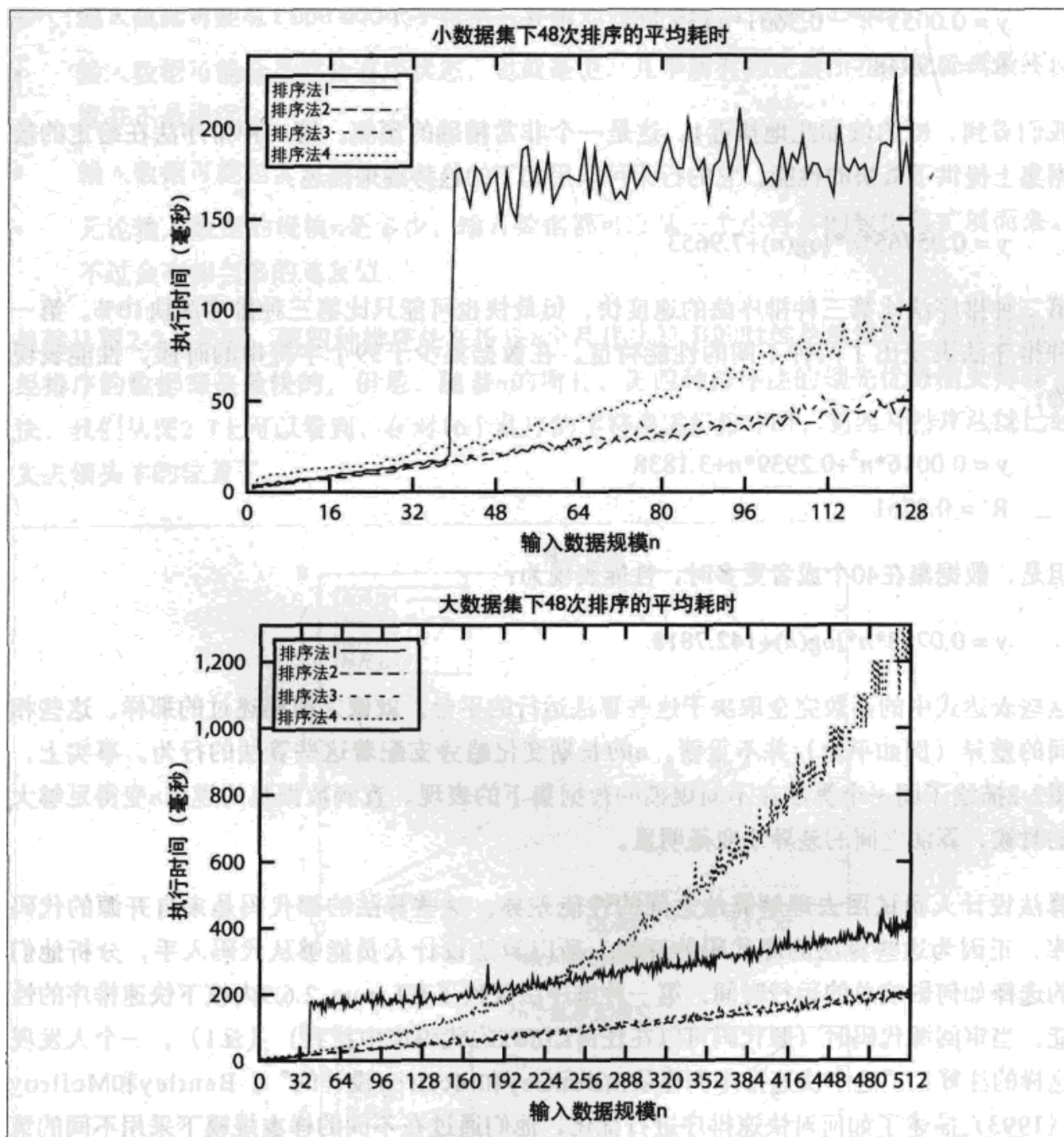


图2-2：小数据集上四个算法的比较结果

我们设计一个函数来解释这个结果，这个函数能够预测每个算法在规模为 n 的数据样本上的性能。我们不太可能精确地知道这个函数的具体形式，我们只能使用商业软件对结果使用统计方法进行回归分析，描绘出一条趋势线。趋势线与实际数据的拟合度设为 R^2 ，取值为0~1。 R^2 越趋近于1，表示拟合度越高。例如， $R^2=0.9948$ 表示，由于数据的随机变化，只有0.52%的几率造成趋势线和数据不拟合。

第四种排序法很明显地是四种算法中最差的，在电子数据表上描绘512个数据点，与数据相符合的趋势线是：

$$y = 0.0053 * n^2 - 0.3601 * n + 39.212$$

$$R^2 = 0.9948$$

我们看到， R^2 的值如此地接近1，这是一个非常精确的预测。第二种排序法在给定的数据集上提供了最快的性能。它的行为可以用如下的趋势线来描述：

$$y = 0.05765 * n * \log(n) + 7.9653$$

第二种排序法比第三种排序法的速度快，但最快也可能只比第三种排序法快10%。第一种排序法表现出了两种不同的性能特征。在数据集少于39个字符串的时候，性能表现为：

$$y = 0.0016 * n^2 + 0.2939 * n + 3.1838$$

$$R^2 = 0.9761$$

但是，数据集在40个或者更多时，性能表现为：

$$y = 0.0798 * n * \log(n) + 142.7818$$

这些表达式中的系数完全取决于这些算法运行的平台。就像之前描述过的那样，这些相同的差异（例如平台）并不重要。 n 的长期变化趋势支配着这些算法的行为。事实上，图2-2描绘了同一个算法在不同规模的数据集下的表现。直到数据集的规模 n 变得足够大的时候，算法之间的差异才变得明显。

算法设计人员试图去理解算法之间的性能差异。这些算法的源代码是来自开源的代码库，正因为这些算法的源代码的开源，所以算法设计人员能够从代码入手，分析他们的选择如何影响总的运行时间。第一种排序法反映了在Linux 2.6.9内核下快速排序的性能。当审阅源代码时（源代码可以在任何Linux的代码库中找到）（注1），一个人发现这样的注释：“这个快速排序例程是由Bentley和McIlroy设计的”。Bentley和McIlroy（1993）描述了如何对快速排序进行优化，他们通过在不同的样本规模下采用不同的策略，例如规模小于7的，规模在8~39的以及40或者更大的规模下采取的策略。实验结果表明，这个隐藏的优化手段是非常有效的。

最好最坏和平均情况下的性能分析

现在有一个问题，对于所有的输入来说，前面得到的结果是否都成立呢？第二种排序法在少量字符串的时候性能也许是最好的。但是，输入数据有很多地方可能变化：

注1：<http://lxr.linux.no/linux+v2.6.11/fs/xfs/support/qsrt.c>.

- 输入数据可能有1 000 000个字符串。算法如何处理如此大规模的数据？
- 输入数据可能会是部分有序状态，也就是说，几乎所有的元素所在的位置离最终位置并不是很远。
- 输入数据可能包含重复的值。
- 无论输入数据的规模 n 是多少，输入数据都可以从一个小得多的数据集扩展而来，不过会有相当多的重复值。

虽然从图2-2上来看，第四种排序法在排序 n 个乱序字符串的时候是最慢的，但是处理已经排序的数据却是最快的。但是，随着 n 的增长，第四种排序法的领先优势消失得非常快，我们从图2-3上可以看到，在对16个乱序的字符串进行排序时，第四种排序法就已经失去领头羊的位置了。

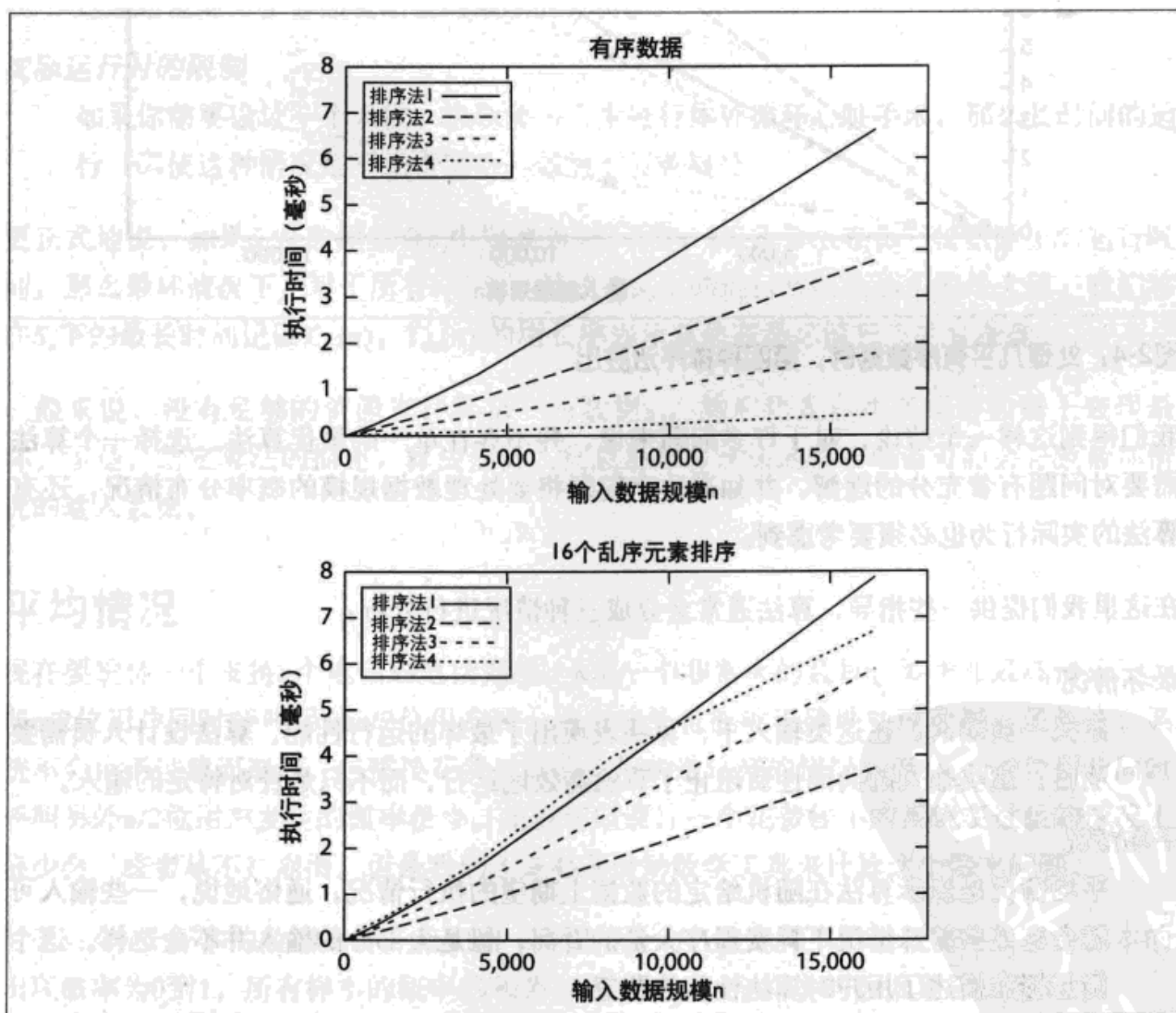


图2-3：处理完全或者几乎有序的数据时，排序算法的性能表现

尽管如此，假设一个包含 n 个字符串的输入，并且这个输入已经是几乎有序，其中 $n/4$ 个

字符串（占总字符串的25%）被交换到仅仅是离最终位置4个元素。最终结果如图2-4所示，我们看到第四种排序法性能表现远远好于其他的排序算法。

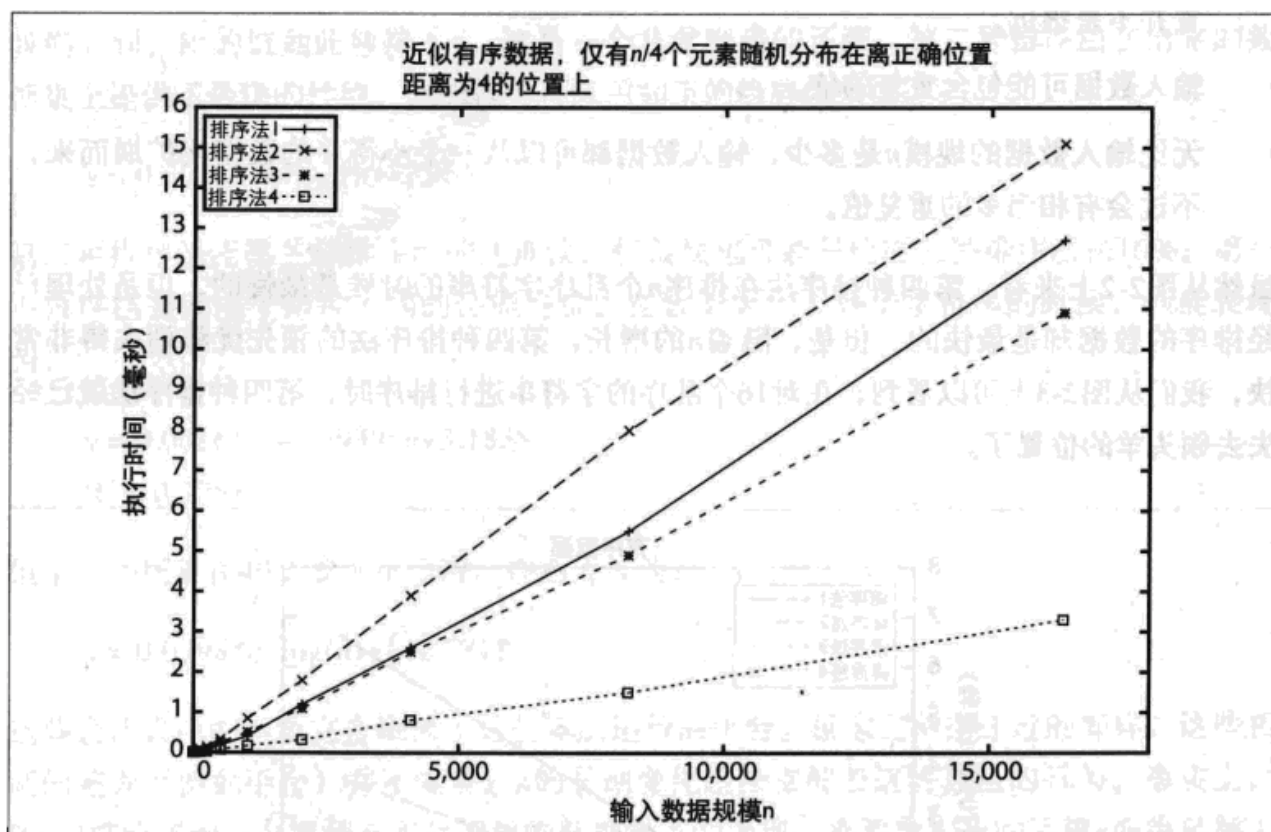


图2-4：处理几乎有序数据时，第四种排序法胜出

我们得到这样一个结论，对于许多问题来说，并不存在单一的最优算法。选择一个算法需要对问题有着充分的理解，并知道这个问题将要处理数据规模的概率分布情况，还有算法的实际行为也必须要考虑到。

在这里我们提供一些指导，算法通常会分成三种情况进行分析：

最坏情况

定义一类输入，在这类输入下，算法表现出了最坏的运行性能。算法设计人员需要明白，这类输入的共同性质阻止了算法高效地运行，而不只是针对特定的输入。

平均情况

平均情况是表示算法在随机给定的数据上期望的执行情况。通俗地说，一些输入可能会在某些特殊情况下耗费程序大量的时间，但是大部分的输入并不会这样。这个衡量标准描述了用户对算法性能的期望。

最好情况

定义一类输入，算法在这类输入下表现出了最好的运行性能。对于这类输入来说，算法只进行很少的计算。不过在实际情况下，最好情况很少出现。

通过分析三种情况，大致了解了算法的性能，那么接下来你需要为将要面临的问题选择一个算法。

最坏情况

大多数问题都可能会处理一些比 n 大的数据。任意给定一个 n ，算法或者程序在处理所有规模为 n 的数据，其性能可能会动态地发生变化。给定一个程序和 n ，这个程序的最坏运行时间就是处理所有规模为 n 的数据所需要的最长运行时间。

我们也看到，最坏情况是对整个世界的一个悲观的分析。但是我们还是非常关注最坏情况，因为：

追根刨底的欲望

这通常是对一个算法复杂度的最早的分析。

实际运行时的限制

如果你需要设计一个系统，协助外科医生进行体外循环心脏手术，那么长时间的运行（即使这种情况是不经常发生）当然不可能接受。

更正式地说，如果 S_n 是数据集 s_i 中规模为 n 的子集， t 表示算法在每一份数据上的运行时间，那么最坏情况下，对于所有 $s_i \in S_n$ ，算法在 S_n 上的运行时间是 $t(s_i)$ 的最大值。我们将在 S_n 下的最长时间记做 $T_{wc}(n)$ ， $T_{wc}(n)$ 的增长率表示算法在最坏情况下的复杂度。

一般来说，没有足够的资源来计算每一份数据 s_i ，然后检查算法在哪份数据上表现最坏。于是，给定算法的描述，算法分析人员总是想方设法地精心准备可能会导致最坏情况的输入数据。

平均情况

现在要设计一个支持 n 个电话的电话系统， n 是一个非常大的数目，要求在最坏情况下，即 $n/2$ 位用户同时呼叫另外 $n/2$ 位用户时，这个系统也能够正确地处理数据。虽然这个系统不会由于过载而崩溃，但需要花费过高的代价构造这样的情况。而且， $n/2$ 位用户同时呼叫另外 $n/2$ 位用户发生的概率极小。也许可以设计一个花费较小的系统在过载的情况下很少会（或者从不）崩溃。但是我们还是必须借助数学工具来计算这个概率问题。

对于一个 n 个样本的数据集合，我们用一个概率分布 Pr 表示样本的出现概率，单个样本的出现概率为0到1，所有样本的概率的和为1。如果 S_n 是 n 个样本的数据集合，那么：

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

如果 t 表示算法在单个样本的执行时间，那么在 S_n 上的平均运行时间是：

$$T_{ac}(n) = \frac{1}{|S_n|} \sum_{s_i \in S_n} t(s_i) \Pr\{s_i\}$$

也就是说，在样本 s_i 的实际执行时间， $t(s_i)$ 将会与概率加权。如果 $\Pr\{s_i\}=0$ ，那么 $t(s_i)$ 的实际值将不会影响程序的期望运行时间。我们用 $T_{ac}(n)$ 表示算法在 S_n 上的平均运行时间，那么 $T_{ac}(n)$ 的增长率表示算法在平均情况下的复杂度。

回忆一下，当我们描述时间或者工作量的增长率时，我们一直忽视了常数，所以我们认为顺序搜索 n 个元素的平均情况为：

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

探测（符合我们之前的假设），按照惯例，在符合这些假设的前提下，期望顺序搜索能够处理线性或者是 n 阶的元素。

最好情况

知道算法的最好情况是非常有用的，即便这种情况很少发生。在很多情况下，最好情况能让我们看到算法的最优状况。例如，线性搜索的最好情况是当它在 n 个元素中搜索 v 的时候，第一个元素恰好就是要找的那个。一个稍微有些不同的算法，我们叫做计数搜索（Counting Search），在 n 个元素中搜索 v ，并且记录 v 在表中出现的次数。如果 v 的计数是0，那么这个值是不存在的，所以会返回false，否则返回true。注意，计数搜索总是会搜索整个表，因此，它的最坏情况是 $O(n)$ （和顺序搜索一样），最好情况还是 $O(n)$ ，所以我们不能够使用这个算法，因为它的最好或者平均情况没有改善性能。

性能指标

比较算法的时候，我们使用输入数据的规模 n 来评价算法的性能。这个方法被认为是比较算法的标准方法，并且在过去的半个世纪中不断地被改进。这样一来，我们就能够通过输入数据的规模，计算算法的运行时间，从而得知，哪个算法能够更好地适应一些异常规模的问题。性能评价的第二种方法是考虑算法将会耗费多少内存或者存储空间。随后我们将在各自的算法章节中讨论这个问题。

我们将要使用的本书特有的算法分类（按照效率降序排列），如下所示。

- 常数级

- 对数级
- 次线性级
- 线性级
- $n \log(n)$ 级
- 平方级
- 指数级

我们现在进行一些讨论，陈述一下衡量这些性能的标准。

讨论0：常数级算法的性能

当本书分析算法性能时，我们将不止一次地强调原生的操作都是常数级的性能。很明显，这个声明并不是完全准确地描述了操作的性能，因为我们没有考虑到硬件问题。例如，比较两个32位的数 x 和 y 大小是否一样，这个操作耗费的时间与 x 、 y 的大小无关。常数的操作被定义为 $O(1)$ 的性能。

那么比较256位的数字时，性能表现又如何呢？1024位呢？我们认为在给定 k 的前提下，比较两个 k 位的数字的操作将在常数时间内完成。关键在于问题的规模（例如 x 、 y 的值）不可能超过 k 。我们把额外的工作，例如 k 的倍增，也抽象为 $O(1)$ 的性能。

讨论1：对数级算法的性能

一个酒保和顾客打了一个10 000美元的赌。“我将选择一个从1~1 000 000的数字，然后你能够每次猜1个数字，共猜20次；每一次猜过之后，我将会告诉你高了，低了或者是你赢了。如果你能在20个问题之内赢，我给你10 000美元，否则你给我10 000美元。”你会打这个赌吗？你也许会，因为你可以一直赢下去。表2-1给了一个场景，不过候选数字是从1~10。问一系列的问题，并且每个问题都把候选数据缩减了一半。

表2-1：猜1~10的例子

数字	第一次猜的数字	第二次猜的数字	第三次猜的数字
1	是5吗？高了	是2吗？高了	是1吗？你赢了
2	是5吗？高了	是2吗？你赢了	
3	是5吗？高了	是2吗？低了	是3吗？你赢了
4	是5吗？高了	是2吗？低了	是3吗？低了，所以是4
5	是5吗？你赢了		
6	是5吗？低了	是8吗？高了	是6吗？你赢了

表2-1: 猜1~10的例子 (续)

数字	第一次猜的数字	第二次猜的数字	第三次猜的数字
7	是5吗? 低了	是8吗? 高了	是6吗? 低了, 所以是7
8	是5吗? 低了	是8吗? 你赢了	
9	是5吗? 低了	是8吗? 低了	是9吗? 你赢了
10	是5吗? 低了	是8吗? 低了	是9吗? 低了, 所以是10

每次根据酒保的回答, 将可能的数字范围缩减一半。最后, 可能的数字只剩下了一个, 这种情况将会在 $\lceil \log(n) \rceil$ 次询问之后出现。将 x 归入到等于或者大于 x 的最小整数。例如, 如果酒保选择了数字范围是1~10, 那么你最多只需要猜测 $\lceil \log(10) \rceil = \lceil 3.32 \rceil$, 即4次。

这种方法在1 000 000个数字的时候也同样可行。事实上, 例2-1所示的猜数算法能够对于任何的范围 $[low, high]$ 有效, 并且在 $\lceil \log(high - low + 1) \rceil$ 次猜测之后得到数字 n 。如果有1 000 000个数字, 那么这个算法将在至多 $\lceil \log(1\,000\,000) \rceil$ 即 $\lceil 19.93 \rceil$, 也就是说20次 (最坏情况) 知道是哪个数字。

例2-1: 在范围 $[low, high]$ 之间猜数字的Java代码

```
// 计算需要多少次猜测, 当n在范围[low, high]的时候
public static int turns(int n, int low, int high) {
    int turns = 0;
    // 当还剩下多于两个数字时, 继续
    while (high - low > 2) {
        // 将[low, high]范围的中间数作为猜的数字
        turns++;
        int mid = (low + high) / 2;
        if (mid == n) {
            return turns;
        } else if (mid < n) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }
    // 在这里, 只剩下两个数字, 我们猜其中一个, 如果是错的话那么剩下那个就是对了。
    // 因此还能剩下1次或者更多的猜测机会。
    return 1 + turns;
}
```

对数级算法是非常高效的, 因为它们能够快速收敛到解。一般来说, 这些算法的成功之处在于它们每次将问题的规模缩减一半。这个猜测算法至多经过 $k = \lceil \log(n) \rceil$ 次迭代便可以得到解, 在第 i 次 ($i > 0$) 迭代的时候, 算法在 $\pm \epsilon = 2^{k-i}$ 个数中进行猜测。 ϵ 可以看做是不确定的个数。在每一次迭代之后, ϵ 缩减一半。

另外一个例子中则展现了牛顿法如何高效地解决一元方程（也就是说， x 取什么值，使得 $f(x)=0$ ）。解 $x*\sin(x)-5*x=\cos(x)$ ，设 $f(x)=x*\sin(x)-5*x-\cos(x)$ ，其导数为 $f'(x)=x*\cos(x)+\sin(x)-5-\sin(x)=x*\cos(x)-5$ 。牛顿迭代计算出 $x_{n+1}=x_n-f(x_n)/f'(x_n)$ 。开始猜测 x 等于0，这个算法能够快速地给出一个正确解， $x=-0.189302759$ ，见表2-2。被用括号[]围起来的二进制和十进制数表示精确位。

表2-2：牛顿法

n	十进制 x_n	二进制 x_n
0	0.0	
1	-0.2	[1011111111001]0011001100110011001100110...
2	-[0.18]8516717588...	[1011111111001000001]0000101010000110110...
3	-[0.1893]59749489...	[101111111100100000111]10011110000101101...
4	-[0.189]298621848...	[10111111110010000011101]011101111111011...
5	-[0.18930]3058226...	[1011111111001000001110110001]0101001001...
6	-[0.1893027]36274...	[1011111111001000001110110001001]0011100...
7	-[0.189302759]639...	[101111111100100000111011000100101]01001...

讨论2：次线性的算法的性能，时间复杂度为 $O(n^d)$ ， $d < 1$

在某些情况下，这种算法的性能好于线性算法，但还是不如对数算法高效。在第9章，我们将会讨论多维kd树，它能够高效地划分 n 个多维点。如果这种树是平衡树，那么区间查询的查询时间将会是 $O(n^{1-1/d})$ 。

讨论3：线性算法的性能

有一些问题的解决看起来明显地需要更多的工作。任何一个8岁大的孩子都知道 $7+5$ 等于12。那么难一点的问题， $37+45$ 呢？一般来说，相加两个 n 位的数 $(a_n \cdots a_1 + b_n \cdots b_1)$ 得到一个 $n+1$ 位的 $c_{n+1} \cdots c_1$ 数字有多难？相加算法使用了如下的原生操作：

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

$$carry_{i+1} \leftarrow \begin{cases} 1 & \text{if } a_i + b_i + carry_i \geq 10 \\ 0 & \text{otherwise} \end{cases}$$

例2-2是相加算法的一个Java实现， n 位数字被用一个int数组表示。在本节的例子中，我们假设数组中的元素表示一个十进制数 $d(0 \leq d \leq 9)$ 。

例2-2: Java的add实现

```
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;
}
```

只要这个输入数据能够被存储在内存中，add就能够使用两个int数组 n_1 和 n_2 来计算两个数的加法。例2-3有一些小改动，这个实现能够更加高效吗？

例2-3: Java的last实现

```
public static void last(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;
}
```

这个小改动确实影响了算法的性能吗？让我们考虑其他两个潜在因素，这两个因素将会影响算法的性能。

- 编程语言是一个因素，如果这两个实现都能够转换成C程序，那么编程语言是如何影响算法的性能？
- 如果这个程序能够在另外一台不同的计算机上执行，那么计算机硬件是如何影响算法性能？

这些实现将会执行10 000次，数的位数从256位到32 768位。数的每一位都是随机产生的。其后，在每次执行中，这两个数都会循环移位，产生两个不同的数进行相加。我们使用了两台计算机：桌面PC和一台高端计算机。并且使用两种不同的编程语言（C和Java）。我们一开始就假设随着数据规模的加倍，算法执行时间也会加倍。我们希望重新确认算法总体行为和计算机、编程语言以及实现无关。

图2-5表示了数据的规模（用x轴表示）与执行10 000次（用y轴表示）的时间（单位是毫秒）的关系。每一次变化都更改了这些配置选项：

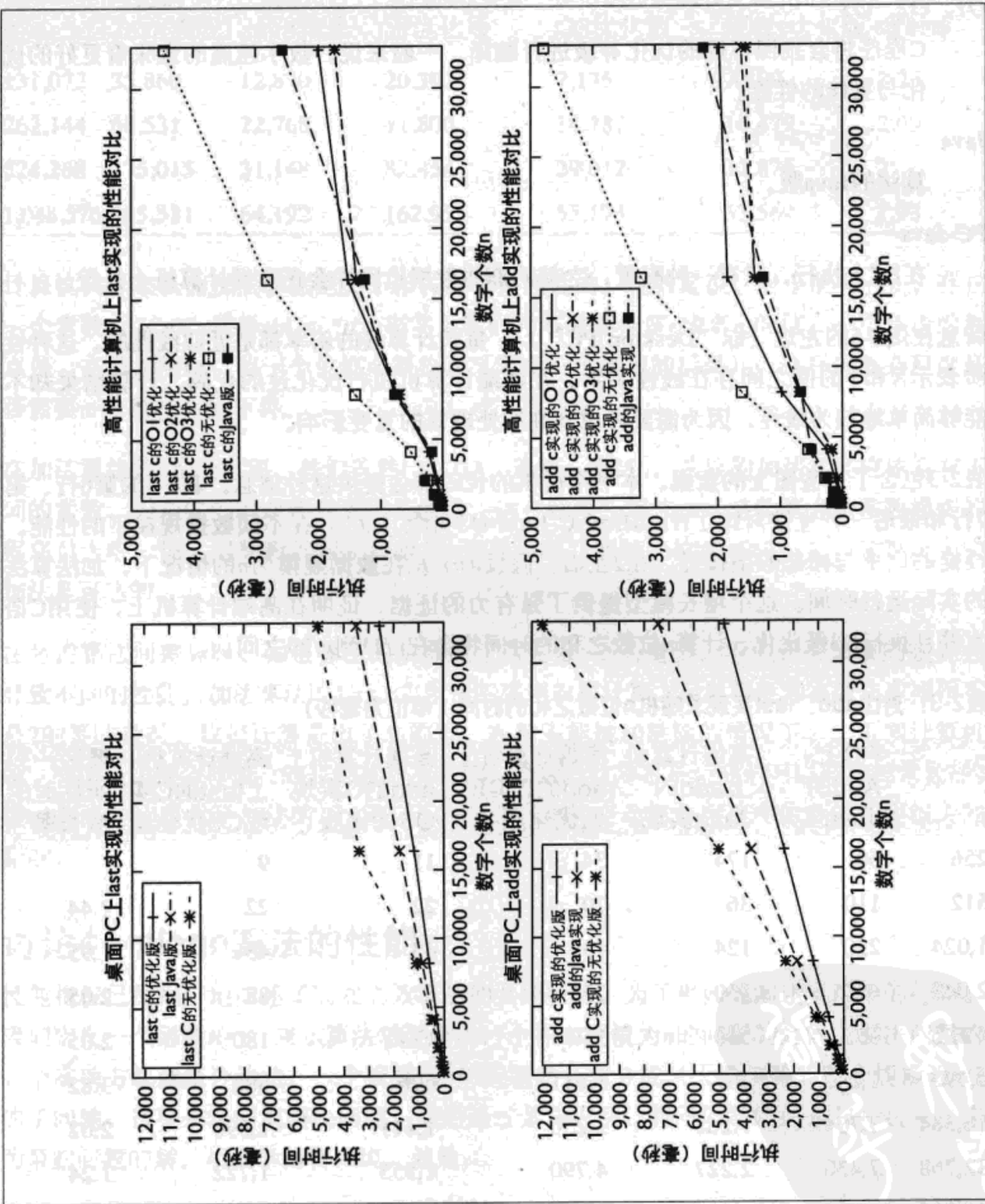


图2-5：在不同的情境下比较add和last的表现

g

C程序将会把调试信息编译进去。

none

C程序不会进行任何优化。

O1, O2, O3

C程序将会按照不同的优化等级进行编译。一般来说，数字越高的意味着更好的优化与更佳的性能。

Java

算法的Java版。

PC-Java

在PC上执行，只有一种配置，之前的那些数据也同样会在高端计算机上执行。

留意在图2-5的左边（标“Desktop PC”），每条计算线的斜率都是近似线性的，这种性质表示X和Y的值之间存在线性关系。在高端计算机执行优化过的代码，计算结果却不能够简单地归为线性，因为需要考虑到高端处理器的重要影响。

表2-3包含了一些图上的数据。本书中提供的代码都会提供这种信息。表2-3的第6行、第7行和最后一行直接对比了HighEnd-C-Last-O3实现（注2）在不同数据规模下的性能。性能的比率与预想的很接近，在2左右。假设 $t(n)$ 是在数据规模为 n 的情况下，加法算法的实际运行时间。这个增长模型提供了强有力的证据，证明在高端计算机上，使用C语言并且执行O3级优化，计算 n 位数之和的时间将会在 $n/11$ 到 $n/29$ 之间。

表2-3：对比add、last实现求随机 n 位数之和的时间（单位为毫秒）

n	PC上 Add的 Java实现	高端计算机 上add的 Java实现	高端计算机上 add的C实现 无优化版	高端计算机上 add的C实现 O3优化版	高端计算机 上last的C实 现O3优化版	最后一列 所耗时的 增长率
256	60	174	34	11	9	
512	110	36	70	22	22	2.44
1,024	220	124	139	43	43	1.95
2,048	450	250	275	87	88	2.05
4,096	921	500	550	174	180	2.05
8,192	1,861	667	1,611	696	688	3.82
16,384	3,704	1,268	3,230	1,411	1,390	2.02
32,768	7,430	2,227	4,790	1,555	1,722	1.24
65,536	17,453	2,902	9,798	3,101	3,508	2.04

注2： 也就是说，last的C实现O3优化版在高端计算机上的性能如附录基准测试中描述的那样一致。

表2-3：对比add、last实现求随机n位数之和的时间（单位为毫秒）（续）

n	PC上 Add的 Java实现	高端计算机 上add的 Java实现	高端计算机上 add的C实现 无优化版	高端计算机上 add的C实现 O3优化版	高端计算机 上last的C实 现O3优化版	最后一列 所耗时的 增长率
131,072	35,860	12,870	20,302	7,173	7,899	2.25
262,144	68,531	22,768	41,800	14,787	16,479	2.09
524,288	175,015	31,148	82,454	29,012	32,876	2
1,048,576	505,531	64,192	162,955	55,173	63,569	1.93

计算机科学家认为加法算法是关于输入规模呈线性的。也就是说，对于所有 $n > n_0$ ，存在一个常数 c ， $c > 0$ ，使得 $t(n) \leq c * n$ 。事实上我们并不需要知道 c 或者 n_0 的值。一个争论的焦点是，在做加法时，每一个位都必须检查（考虑一下漏掉的后果），这样的复杂程度是否需要一个线性时间下界。

在加法算法的last实现，我们设置 c 为1/11，选择 n_0 为256。其他的加法实现应该会有不同的常数，但是它们的总体表现仍呈线性。这个结果让那些认为整数算术是常数操作的程序员大吃一惊。当整数的固定位数用 n 来表示时（例如16位或者64位），常数时间的加法是可达的。

在考虑算法间差异时，知道算法的阶数比了解常数 c 更为重要。看似无关紧要的差异会导致不同的性能。加法算法的last实现在不采用取模计算之后显得更加高效。在操作不是2的幂的数时，取模计算是出了名的慢。在数不能被10整除的情况下，二进制计算机做“%10”计算会有较大的开销且并不高效。当然，这并不是说我们可以忽略掉常数 c 的值。如果我们做大量加法的话，那么 c 的一点点小的变动都会对程序的性能产生很大的影响。

讨论4：nlogn算法的性能

性能指标已经很好地描述了同类高效算法的共同行为。为了更好地阐述实践中的行为，我们定义一个函数 $t(n)$ ，表示算法需要解决一个样本规模为 n 的问题的时间。解决问题的一个高效方法就是分治法，一个规模为 n 的问题将会被分成（大致相等）两个规模为 $n/2$ 的子问题，这样来递归地解决问题，最后通过某些方法，将子问题的结果归并起来，作为最初问题的解。用数学语言来说，就是：

$$t(n) = 2 * t(n/2) + O(n)$$

也就是说， $t(n)$ 包括了解决两个子问题的开销以及归并结果的开销，归并结果的开销不

会高于线性时间。在等式的右边， $t(n/2)$ 是解决规模为 $n/2$ 的问题的时间，按此逻辑推理， $t(n/2)$ 能表示为：

$$t(n/2) = 2 * t(n/4) + O(n/2)$$

所以最初的等式可以写为：

$$t(n) = 2 * [2 * t(n/4) + O(n/2)] + O(n)$$

我们再扩展一层，得到：

$$t(n) = 2 * [2 * [2 * t(n/8) + O(n/4)] + O(n/2)] + O(n)$$

最后一个等式归约到 $t(n) = 8 * t(n/8) + O(3 * n)$ 。将其推广到一般情况，我们可以说 $t(n) = 2^k * t(n/2^k) + O(k * n)$ 。这个扩展在 $n = 2^k$ ，也就是 $k = \log(n)$ 时停止。扩展到最后，问题规模仅仅只是1了， $t(1)$ 是一个常数。因此我们能够得到闭合解为 $t(n) = n * c + O(n * \log(n))$ 。因为对于任何常数 c 来说， $n * \log(n)$ 都比 $c * n$ 要高阶，所以 $t(n)$ 能够简写为 $O(n \log n)$ 。

讨论5a：二次方的算法性能

现在考虑一个类似的问题：两个 n 位的数相乘。例2-4是这个乘法的一个实现，采用小学使用的很简单的算法。

例2-4：Java的乘法实现mult

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

    // 清除所有的值
    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];
            // 计算部分和，并且加上进位

            result[pos-off] += prod % 10;
            result[pos-off-1] += result[pos-off]/10 + prod/10;
            result[pos-off] %= 10;
        }
    }
}
```

同样地，我们也会给出一个变形算法：alt，这个算法消除了取模运算的高开销，并且避开了当 $n1[m]$ 等于0时不必要的内层计算（注意，alt算法没有在这里描述，但是能够在提供的代码库中找到）。alt算法包含了203行Java代码来避免使用两个取模操作。那么这个变形算法在有额外的开销时，能够节省总体的开销吗？

表2-4给出了这些乘法算法的性能，所处理的数据是在描述加法算法时使用的随机生成数据。图2-6描绘了性能的图形，抛物曲线是二次方算法的典型标志。

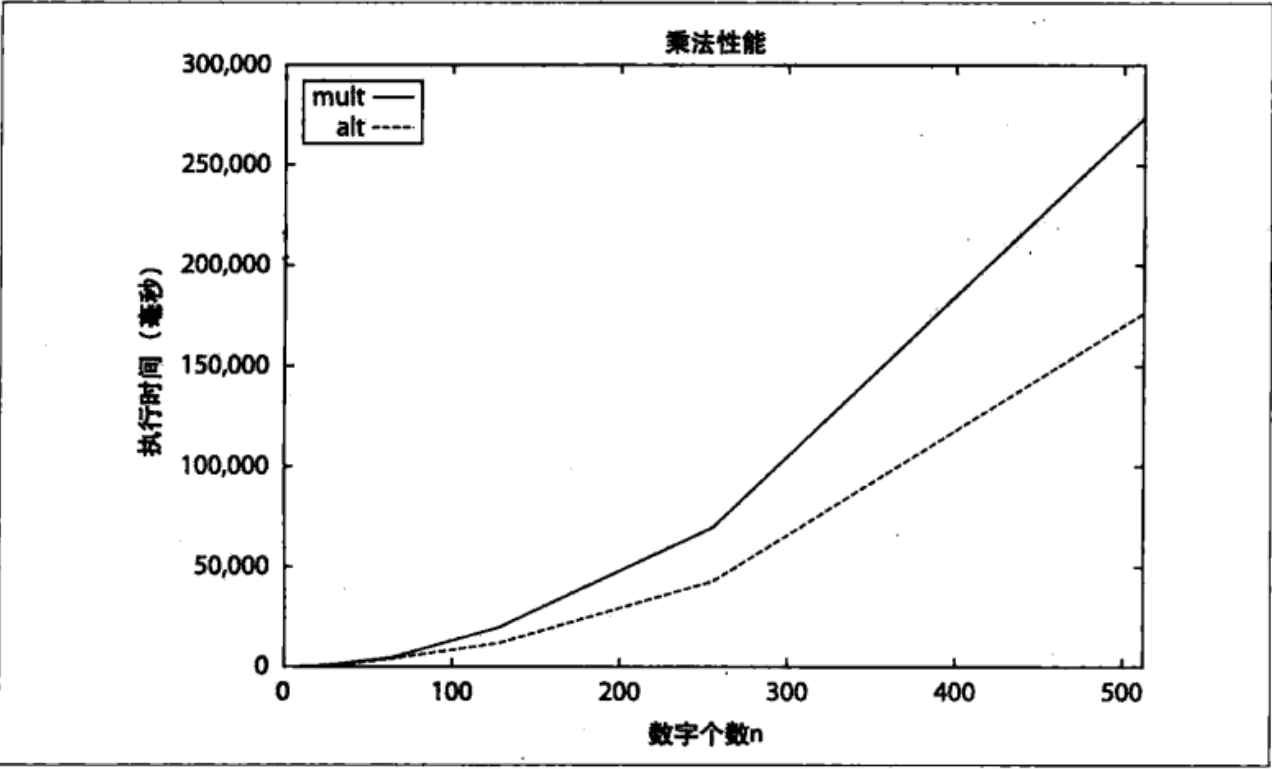


图2-6: 比较mult和alt

表2-4: 执行10 000次乘法的时间（毫秒）

n	Mult _n (ms)	alt _n (ms)	mult _{2n} /mult _n
2	15	0	
4	15	15	1
8	62	15	4.13
16	297	218	4.80
32	1,187	734	4.00
64	4,516	3,953	3.80
128	19,530	11,765	4.32
256	69,828	42,844	3.58
512	273,874	176,203	3.92

虽然alt变种的速度能够快大概40%，但是alt和mult展现了相同的渐进性能。mult_{2n}/mult_n的比率大约是4，证明了乘法的性能是二次方的。让我们定义一个 $t(n)$ ，表示乘法算法在输入规模为n的时候的实际运行时间。根据这个定义，肯定存在一些常数 $c(c>0)$ ，使

得 $t(n) \leq c \cdot n^2$ ，对于所有的 $n > n_0$ 来说。我们不需要 c 和 n_0 的实际值是多少，只需要知道它们肯定存在即可。在我们平台上运行的这个mult算法实现，我们设 c 为1.2， n_0 为64。

再次说明，改变算法的实现来提高效率是不可能超越算法本身固有的二次方性能的。但是，存在其他的算法（Zuras, 1994）使得 n 位数相乘的速度能够快于二次方。在数据加密这样的需要大量频繁的大数乘法应用中，这种算法是非常重要的。

讨论5b：性能不明显的计算

在很多情况下，通过算法的描述（如加法和乘法）就已经足够分辨一个算法是线性的还是二次方的。例如二次方的主要特征，是嵌入的循环结构。但是，有一些算法不能够使用如此直接的分析。让我们来看看例2-5的GCD算法，这个算法是由欧几里德设计，用来计算两个整数的最大公约数。

例2-5：欧几里德GCD算法

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

    // 确保a和b没有被修改
    a = copy (a);
    b = copy (b);

    while (!isZero(b)) {
        //subtract的最后一个参数表示我们能够忽略结果的符号，因为我们总是用大数减去小数
        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);
        }
    }

    // a的值就是原始的a、b的最大公约数
    assign (gcd, a);
}
```

这个算法不断地比较 a 和 b ，然后用大数减去小数直到得到0为止。这个实现的辅助函数（isZero、assign、compareTo和subtract）没有在这里给出，不过能够在代码库中找到。

这个算法计算出了两个数的最大公约数，但是对于给定输入的规模，要经过多少次迭代，这个答案并不明显。在每一次循环之后， a 或者 b 都不会为负，所以我们能够保证这个算法会结束，但是一些GCD的计算可能会需要比较长的时间，例如，这个算法计算

gcd(1000,1)会执行999步！现在这个算法的性能对于输入数据比乘法和加法算法要敏感得多，对于相同规模的输入来说，gcd算法的时间是随着输入数据的变化而变化的。gcd算法在计算 $(10^n - 1, 1)$ 的最大公约数时性能最差，它需要处理 $10^n - 1$ 次循环！我们已经知道加法和减法在数据规模为 n 时时间复杂度为 $O(n)$ ，gcd总共需要 $n \cdot (10^n - 1)$ 次操作。把这个等式转换到二进制上来，我们得到 $n \cdot (2^{3.3219 \cdot n}) - n$ ，这个等式是指数级的，所以我们认为这个算法是 $O(n \cdot 2^n)$ 。

例2-6中MODGCD算法的性能要比例2-5的gcd实现好得多，这个算法使用取模计算一个数模 b 的余数。

例2-6: MODGCD算法计算最大公约数

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

    // 将a、b调整到相同位数，然后在其副本上计算
    a = copy(normalize(a, b.length));
    b = copy(normalize(b, a.length));

    // 确保a比b大，同时返回最大公约数
    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);
    }

    // a的值就是原始的a、b的最大公约数
    assign (gcd, a);
}
```

MODGCD将能够更快地得到解，因为它不会不断地在while循环中从大数中减去小数。这个差异并不仅仅是实现细节的差异，它告诉我们，在解决问题时，要融会贯通。

图2-7（也在表2-5中进行了枚举）描绘了对10 011对随机产生的142位数求最大公约数的计算结果。

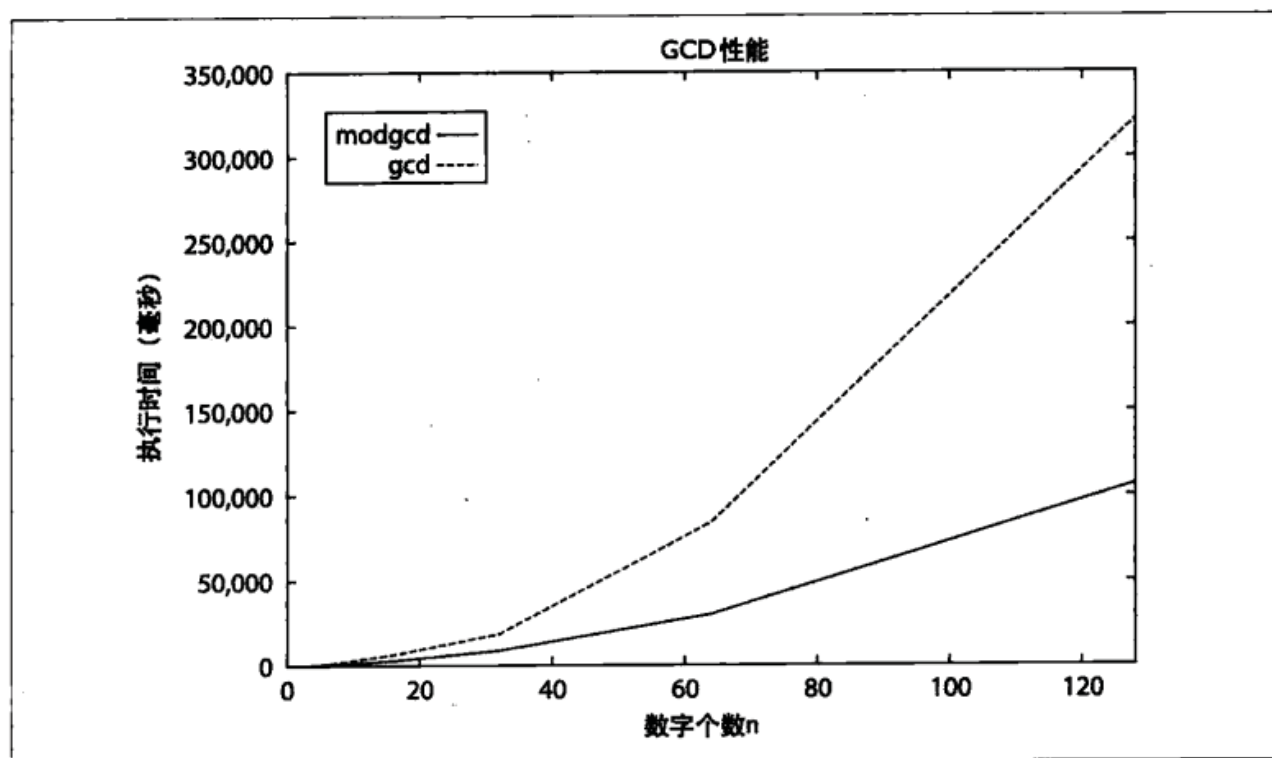


图2-7：比较gcd和MODGCD算法

表2-5：计算10 011次最大公约数的时间（ms）

n	modgcd	gcd	n^2/modgcd	n^2/gcd	$\text{modgcd}_{2n}/\text{modgcd}_n$	$\text{gcd}_{2n}/\text{gcd}_n$
2	234	62	0.017	0.065		
4	391	250	0.041	0.064	1.67	4.03
8	1,046	1,984	0.061	0.032	2.68	7.94
16	2,953	6,406	0.087	0.040	2.82	3.23
32	8,812	18,609	0.116	0.055	2.98	2.90
64	29,891	83,921	0.137	0.049	3.39	4.51
128	106,516	321,891	0.154	0.051	3.56	3.84

即使MODGCD实现比其他的gcd实现快60%，MODGCD的性能仍然是二次方，也就是 $O(n^2)$ ，但是gcd算法却是指数级的。也就是说，gcd的最坏性能（不会在小数据集下展现）会比MODGCD的最坏性能要慢几个数量级。

更好的计算最大公约数的算法已经设计出来了，但是除非是特别大的整数，很多都由于实现复杂度太高而不可能投入使用。通过分析这个问题，也表明需要更加高效的算法。

混合操作

在之前的边栏“编码对性能的影响”中，设计者不得不考虑同时发生的多个乘法操作。并不是每个操作都能够被优化。事实上，优化一个操作将会降低另外一个操作的执行性能。例如，考虑这样一个数据结构，包含两个操作 $op1$ 和 $op2$ 。假设这个数据结构能够有 A 和 B 两种方法实现。基于这个讨论的目的，知道这个数据结构或者各个方法是不重要的。我们构建下列两种情景：

小数据集

数据集大小 n 为1000，然后混合执行2000次 $op1$ 操作和3000次 $op2$ 操作。

大数据集

数据集大小 n 为100 000，然后混合执行200 000次 $op1$ 操作和300 000次 $op2$ 操作。

表2-6包含了两个实现 A 和 B 在两个数据集上的预期性能。表中的第一列是 A 实现在 $n=1000$ 的数据集上执行 $op1$ 的平均花费时间，大概为0.008毫秒，第二列和第三列的数据也是类似的。最后一列是执行的总时间。因此，对于在 $n=1000$ 的数据集上，我们期望 A 实现需要花费 $2000 \times 0.008 + 3000 \times 0.001 = 16 + 3 = 19$ 毫秒，虽然 B 实现在小数据集上初始表现要好于 A 实现，但是在问题规模扩大了两个数量级之后，我们发现结果出现了戏剧性的变化， A 实现适应了变化，而 B 实现表现得不尽如人意。

表2-6：比较不同的实现

输入规模	op1 (ms)	op2 (ms)	# op1	# op2	总时间(ms)
A on 1,000	0.008	0.001	2,000	3,000	19
A on 100,000	0.0016	0.003	200,000	300,000	1,220
B on 1,000	0.001	0.001	2,000	3,000	5
B on 100,000	0.1653	0.5619	200,000	300,000	201,630

基准测试

例2-7是一个计算 2^n 的程序抽象。例子中我们要计算 2^{851} 。

例2-7：耗时的计算

```
;; TwoToTheN: number -> number
(define (TwoToTheN n)
  (let loop ([i n]
             [result 1])
    (if (= i 0)
        result
        (loop (sub1 i) (* 2 result))))))
```

```
;; the result of a sample computation
(TwoToTheN 851)
15015033657609400459942315391018513722623519187099007073355798781525263125238463
41589482039716066276169710803836941092523836538133260448652352292181327981032007
94538451818051546732566997782908246399595358358052523086606780893692342385292277
74479195332149248
```

在这个抽象中，计算是与依赖的平台相互独立的。也就是说，在大多数平台下，使用Java或者C语言计算 2^{851} 都会导致数据溢出。不过抽象的程序中却能够快速得到结果。依赖的结构非常抽象，对于我们不可见，这是一个优点还是缺点呢？考虑以下两个假设。

假设H1

计算 2^n 的行为与 n 的值无关。

假设H2

大数（例如在之前处理的数）能够和其他的数用同样的方式进行处理，例如123 827或者997。

为了验证假设H1，我们将做50次实验，计算10 000次 2^n 。我们抛弃最好和最坏的结果，留下48次。48次实验的平均时间如图2-8所示。

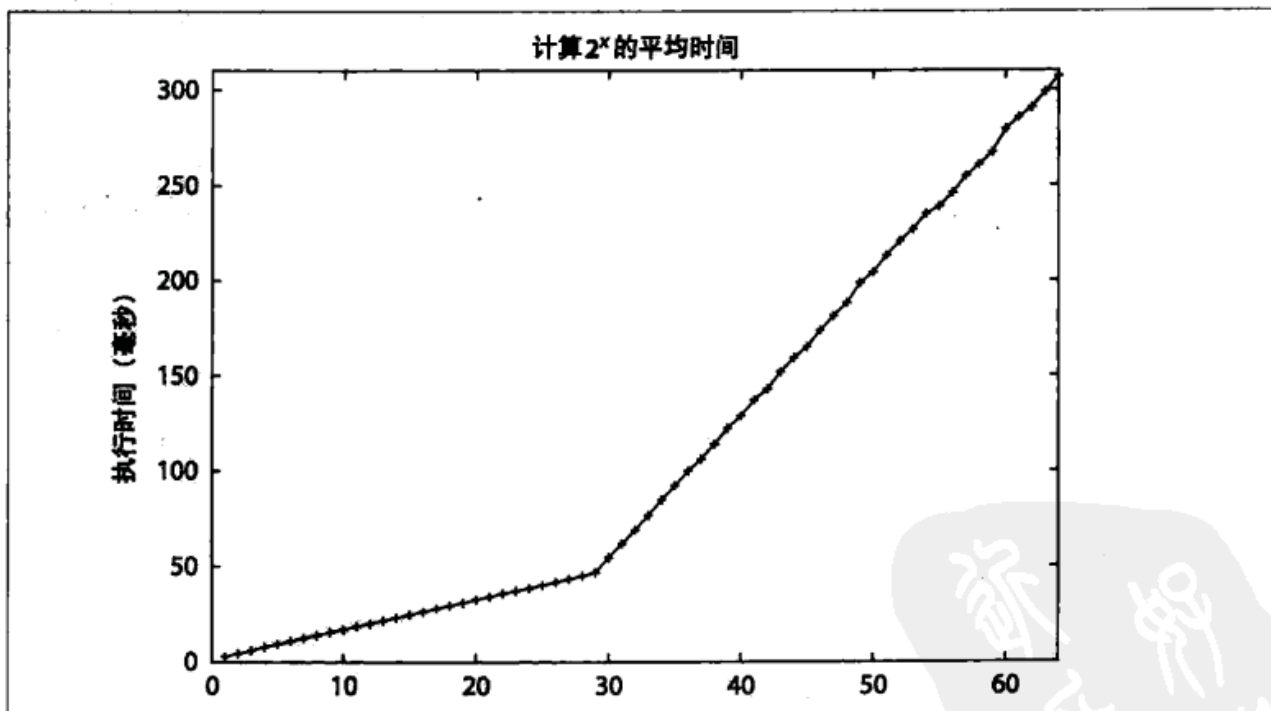


图2-8：计算 2^x 的执行时间

求2的幂计算的性能初始很明显是线性。但是，一旦 x 达到30左右，出现了另外一种线性关系。由于某些原因，数是2的幂且比30大时，性能会发生改变。

为了验证假设H2，我们进行另外一个实验，首先计算出 2^n ，然后对计算 $3.14159 \cdot 2^n$ 的时

间进行对比。我们将做50次试验，每次实验执行10 000次计算，并且抛弃最好和最坏的结果，留下48个。图2-9所示的是48次试验的平均时间（这些结果非常相似，即使我们用1.0000001代替3.14159）。

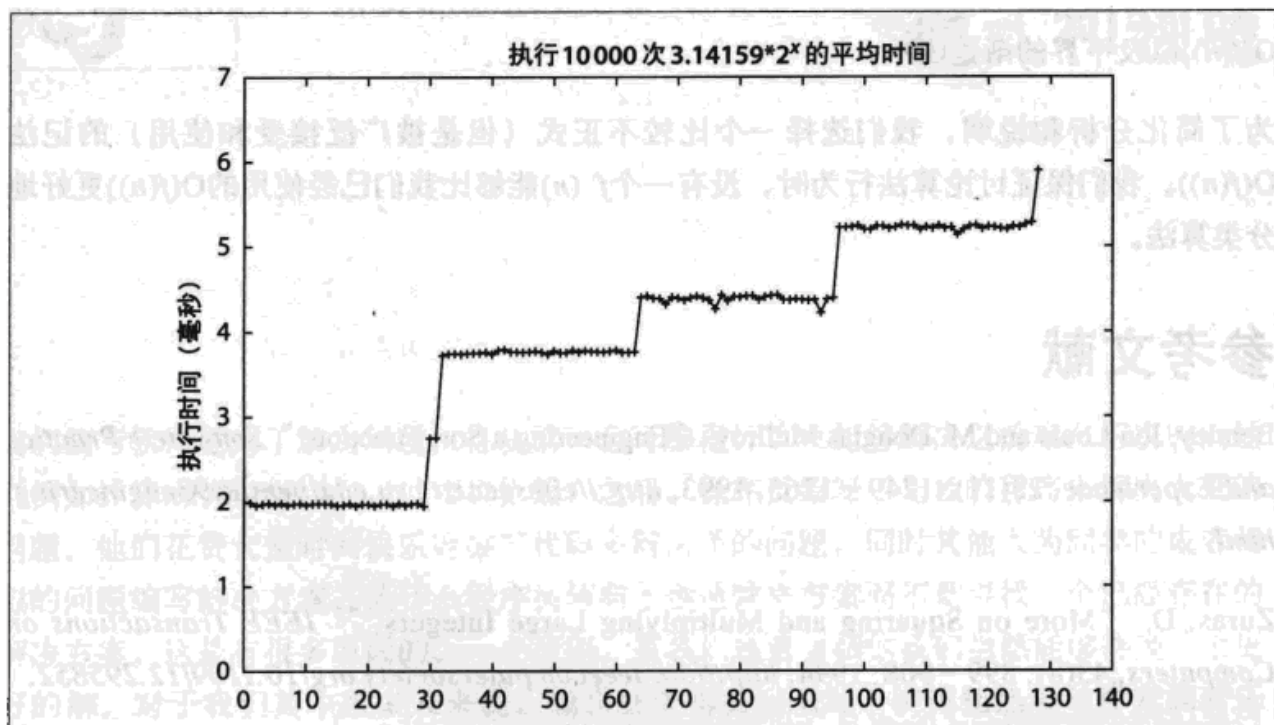


图2-9：大数计算的执行时间

为什么图2-9的点不在一条直线上？当 x 等于多少时这条线开始不为直线？乘法操作显得超负荷。这个操作根据乘数的不同会做不同的事情，乘数可以是浮点数、单字的整型、多字的大整数，或是这些的聚合形式。

第一个断点是在 $x=\{30,31\}$ 时，这个断点的产生很难解释。剩下的平稳状态则有传统的解释，这些改变发生在 $(32, 64, 96, 128)$ 时，这些表示的是做实验的计算机的字长（即单字长、双字长、三字长以及四字长）。随着数据需要的存储空间的增长，乘法计算的时间也不断增长。

基准操作对算法来说是非常重要的，通过计算基准操作的执行次数能够对程序的执行时间有一个很好的预测。TwoToTheN的基准操作就是乘法。

最后一点

我们已经简化了“大O记法”的表示。例如，加法算法与输入规模呈线性，当讨论这个算法的行为时，我们认为存在一些常数 $c>0$ ，对于所有的 $n>n_0$ ，使得 $t(n)\leq c*n$ ，回忆一下， $t(n)$ 是表示加法的实际运行时间。因此我们认为，加法的性能是 $O(n)$ 。细心的读者会发现我们用过一个函数 $f(n)=c*2^n$ ，这个函数比 $c*n$ 增长要迅速得多。事实上，虽然对加

法算法进行精确分析得到其性能应该为 $O(2^n)$ ，但是这样的结论提供了非常少的信息（有可能你需要用一周多的时间来计算一个5分钟的计算任务）。解释一下原因，考虑这个函数 $\Omega(g(n))$ ，其中， $g(n) \leq t(n)$ ，这个函数表示实际运行时间的下界。一个能够计算出执行时间的上界（ O ）和下界（ Ω ）的算法，通常用 $\Theta(f(n))$ 来表示，其中 $f(n)$ 是 $t(n)$ 的上界 $O(f(n))$ 以及下界的渐进表示。

为了简化分析和说明，我们选择一个比较不正式（但是被广泛接受和使用）的记法 $O(f(n))$ 。我们保证讨论算法行为时，没有一个 $f(n)$ 能够比我们已经使用的 $O(f(n))$ 更好地分类算法。

参考文献

Bentley, Jon Louis and M. Douglas McIlroy, "Engineering a Sort Function," *Software—Practice and Experience*, 23(11): 1249–1265, 1993, <http://citeseer.ist.psu.edu/bentley93engineering.html>.

Zuras, D. "More on Squaring and Multiplying Large Integers," *IEEE Transactions on Computers*, 43(8): 899–908, 1994, <http://doi.ieeecomputersociety.org/10.1109/12.295852>.



我们编写软件是为了解决问题。自从第一台可编程计算机能够解决之前难以解决的问题（例如计算 π 的第400 052 412 247位数）之后，程序员编写了无数的程序来解决大量的问题。他们花费大量时间快乐地编写代码来解决手的问题，同时其他人为同类的或者相似的问题编写解决方案。为什么程序员倾向于创造解决方案而不是寻找一个已经存在的解决方案，这是有很多原因的。一个原因，是我们通常会相信我们当然能够得到一个更好的解。对于我们其中很多人来说，编程就像棋类运动对于棋类狂热者一样充满着乐趣。当考虑为什么程序员不断地编写同样的解决方案时，还有一些更重要的原因：

- 程序员不会意识到问题已经被解决。当我们讨论问题领域时，我们将会看得更远。
- 虽然程序员知道问题已经用相似的方法解决了，但是现存的代码是否适合程序员面对的问题事实上并不清楚。
- 并不容易找到完全或者能够经过小小修改适合手头问题的代码。

模式：一种交流语言

在20世纪80年代末，一小部分充满想象力的软件开发者开始寻找新的办法，使得彼此能够顺利交流软件设计经验。其中一些工作恰好是由Christopher Alexander负责，他是伯克利加利福尼亚大学建筑系的一位教授。他在1977年编写了《A Pattern Language: Towns, Buildings, Construction》一书。在这个有发展潜力的工作中，Alexander开发了一套用于描述建筑结构设计的理论。1987年，Kent Beck和Ward Cunningham，这两位面向对象范型的著名领导者，在当年的面向对象编程系统、语言和应用大会（OOPSLA）上介绍了如何将设计模式应用到编程。这个想法迅速流行开来，人们开始思考有关软件设计模式方面的东西。在1995年，Erich Gamma、Richard Helm、Ralph Johnson和John Vlissides四个人（GoF）编写了一本具有开创意义的书——《Design Patterns: Elements

of Reusable Object-Oriented Software》(Gamma等人, 1995), 关于设计模式的研究和应用活动迅速开展起来。

和其他的优秀思想一样, 大量的模式被迅速应用到软件工业, 使用模式来设计软件已经成为很普遍的事情, 甚至到了什么东西都可以用模式来描述的地步。编码规范是一种模式, 和同伴坐在一起调试程序也是一种模式。模式是精确和简洁交流良构概念的最好方法。我们将模式应用到计算机科学其他领域, 例如使用模式来描述本书的算法。

在描述我们如何为本书的算法构造模式语言之前, 先看看什么是模式, 为什么它是如此优秀。关于什么是设计模式, 我们推荐如下定义:

设计模式是一个验证过的解决方案, 这个解决方案能够解决一般问题。

这个定义非常简短, 但是表达出了设计模式的本质。首先, 设计模式是对实际问题的解决方案。事实上, 它是对一个普遍问题集合的通解。但是, 设计模式不是模板, 只做简单地填空是行不通的。它是一种方法, 或者是一个计划, 用来解决一个特定类型的问题。在你的工具箱里装上一系列的设计模式, 你就踏上了软件设计大师之路。

我们能够从不同的角度考虑算法。很多开发者喜欢在书中或者在一些网站上找到算法, 复制代码, 然后运行, 或者可能还需要一些测试, 然后迅速地转移到下一个任务中。我们认为, 这种行为对于提高自身对于算法的理解一点用处也没有。事实上, 这种行为将会使得你在选择算法实现的时候处于一个错误的位置。记住第1章Graham是如何盲目地选择了二叉树却没有不厌其烦地去平衡它。当你使用了看似能解决问题的最初的想法, 那么接下来会发生什么就可想而知了。

所以, 问题是你如何快速定位到正确的算法并且对它有足够的了解, 确保你已经做出了正确的选择。模式能够帮助我们这样做。事实上, 算法就是对于已知问题的验证过的解决方案, 这也符合我们对于设计模式的定义。

算法模式的格式

设计模式通常表示为一种模式化的方法, 使得开发者能够易于交流。并不是所有的模式作者或者书籍都赞成使用特定的格式, 但是它们有很多东西是相同的。我们采用了类似于模式的方法来阐述算法, 因为我们相信这样做, 读者能够更高效地阅读。如果你能够对本书内容有更好的理解, 可以将这种模式化的方法纳为己用。

按照我们的模式语言, 每个算法用一些固定的章节来表述。有时如果一个章节被认为是对描述算法没有帮助的话, 那么这个章节可以省略掉。有时我们可能需要增加一些额外的章节来描述一些特定的知识点。

算法模式的格式

每个算法都会按照统一的模式来描述，它包含如下章节：

名称

算法的描述性的名字。我们用这个名字简洁地区分算法。例如，如果我们谈论顺序查找，这个名字传达的信息是我们谈论的是哪种查找算法。每一个算法的名字都是用小型大写形式表示，本书中按照这样排版的所有的单词都是表示算法的名称。

梗概

算法的高级描述和它的目的。

使用环境

这是一个问题的描述，这个问题说明了算法的最佳使用位置。

驱动因素

描述了问题或者解决方案的性质，一个成功的实现能够很好地处理这些问题或者维护这些解决方案。这些就是导致你为什么选择这个算法的理由。

解决方案

使用实际工作的带注释的代码描述算法。如果必要的话，UML类图也会包含在内。

结论

区分和阐述算法的优缺点，以及反例。

分析

对算法的大致分析，包括性能数据以及其他帮助读者理解算法行为的数据。虽然分析章节并不是想证明从描述中得到的算法性能，但是读者需要理解为什么算法会这样做。我们同样提供了具体阐述了相关的引论和证明的参考文献，以帮助读者理解为什么算法的行为是和描述的相一致。

相关算法

阐述了算法的细微变化或者完全不同的变换。

这样的算法模板使得你能快速地对不同的算法，同时，你也能够得知看似不同的算法的共性。

伪代码模式的格式

书中的每一个算法都同时用代码样例表示出来，这些代码是用主流的编程语言编写的，例如C、C++、Java、Scheme和Ruby。对于那些不熟悉所有这些语言的读者，我们首先使用伪代码来介绍代码，并且用一个小样例来解释其执行。

看看图3-1。每一个算法都有名字，并且其性能被三个性能指标（最好、平均和最坏情况）很清晰地标示出来。表格的右上角列出来了算法使用的一系列概念。使用这个区域表示这些概念能够非常快速地看出不同算法的共性（例如，“这两个算法使用了优先队列”）。

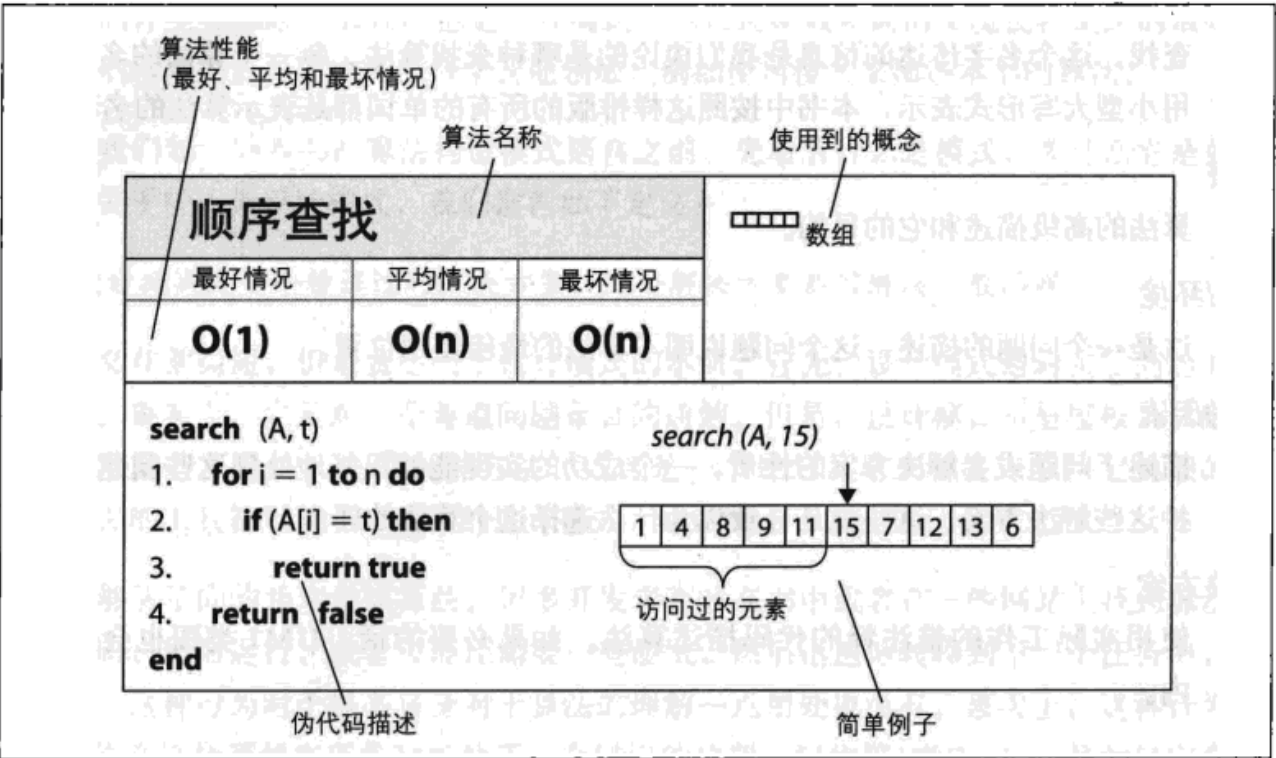


图3-1：顺序查找详解

详解中出现的符号在图3-2有更详细的描述。有些概念是一些算法使用的数据结构（例如“队列”），而其他的则是解决问题的方法（例如“分治”）。

伪代码描述应该保持简洁，并且不应该超过一页的3/4。关键字和函数名字用粗体字。所有的变量使用小写字母，数组名称大写，元素采用A[i]这样的表示形式。伪代码的意图是描述条件语句和循环语句。函数中的所有表达式在单独的章节中都会具体描述（如果必要的话）。

你应该阅读上面的详解表，并且在阅读源代码实现的时候将其作为参考。在详解中，用一个简单示例来解释算法的执行。一般来说，为了表示算法的动态性能，算法的每一步都被顺序地纵向描绘出来，表示基本情况介绍表上时间是向下流动的。

设计格式

我们为解决方案提供了一系列的UML类图，这些解决方案是用C++或者Java编写的。这些图非常有用，能够帮助我们更好地理解那些利用了类继承和多态的代码。图3-3包含了

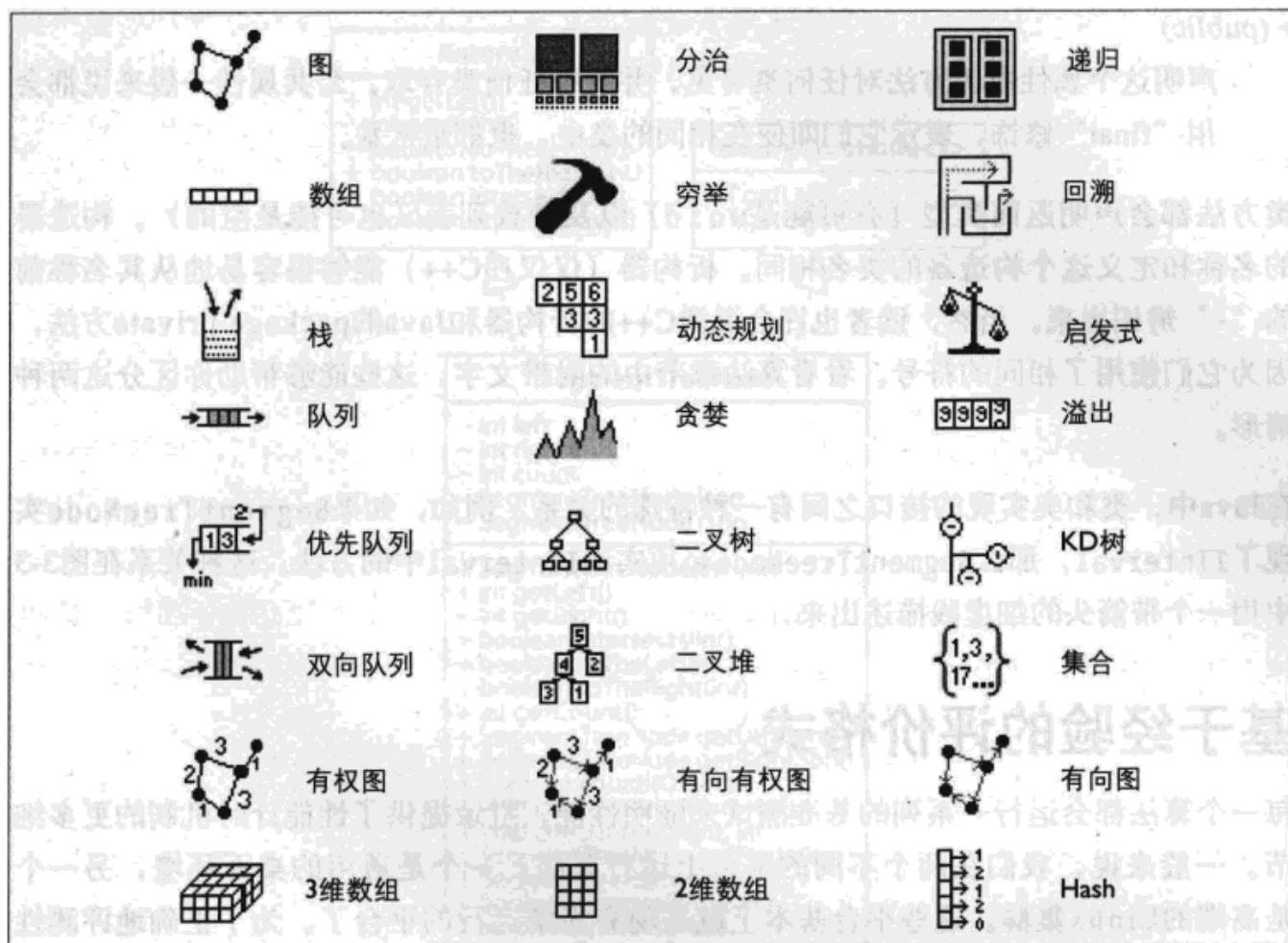


图3-2：区分算法中使用的各种概念的符号

一个小小的类图，图中展现的是一个超类SegmentTreeNode和两个子类，DefaultSegmentTreeNode和StoredIntervalsNode的关系，这两个子类使用继承的方法扩展SegmentTreeNode（箭头指向的类）。每一个类匣有两个部分：上部列出了实例变量，下部列出了实例方法。每个属性或者方法名称前的符号是非常重要的。

(protected)

声明这个方法或者属性只对这个类或者其子类可见，如果是使用Java实现的话，那么这个方法或者属性只对在同一个包中的类可见。注意在C++实现中，我们不会使用多重继承或者友元，所以在本书的这两种语言实现中，这个语法是相同的。

~ (package-private)

声明这个属性或者方法只对同一个包内的类可见，只在Java中使用。

- (private)

声明这个属性只对定义了这个属性的类本身可见。我们没有在类图中列出任何可能存在的私有方法。

+ (public)

声明这个属性或者方法对任何类可见，并能被任何类存取。公共属性一般来说都会用“final”修饰，表示它们即使在相同的类中，也都是常量。

类方法都会声明返回类型（有可能是void）以及参数列表（也可能是空的）。构造器的名称和定义这个构造器的类名相同。析构器（仅仅在C++）能够很容易地从其名称前的“~”辨识出来。当然，读者也许会混淆C++的析构器和Java的package-private方法，因为它们使用了相同的符号。看看算法章节中的附带文字，这些能够帮助你区分这两种情形。

在Java中，类和类实现的接口之间有一种特殊的关系。例如，如果SegmentTreeNode实现了IInterval，那么SegmentTreeNode必须实现IInterval中的方法。这种关系在图3-3中用一个带箭头的细虚线描述出来。

基于经验的评价格式

每一个算法都会运行一系列的基准测试来证明性能。附录提供了性能计时机制的更多细节。一般来说，我们在两个不同的平台上运行算法：一个是通用的桌面环境，另一个是高端的Linux集群。这些平台基本上就是现有系统运行的平台了。为了正确地评测性能，一个测试套件由 k 个独立的实验组成（通常 $k \geq 10$ ）。最好和最坏的结果将会作为离群点抛弃掉，剩余的 $k-2$ 次实验结果会综合在一起，计算出平均值和标准方差。有一张表会表示问题样本的规模 n ， n 从 $2 \sim 2^{20}$ 。

领域和算法

在20世纪80年代后期，当研究人员开始研究如何通过面向对象原理推动软件复用的时候，围绕着领域的讨论开始变得流行起来。在复用的使用环境中，领域是应用程序具有共性的区域。每一个领域都有自己的词表，这个词表提供了描述领域的语言。这个语言帮助程序员设计适合特殊领域的系统和复用组件。领域专用语言（DSL）用来建立领域的模型，产生专有领域的软件，而不是手动地构建它。

算法，就像应用程序一样，也有领域一说。这个领域和专有应用领域是正交的。算法领域告诉我们应用领域是和特定算法有关的。例如，如果一名开发人员希望开发一个能够从移动电话上访问Web服务的应用程序，那么应该使用那些专为空间使用和额外存储优化的算法，而不是使用总体运行时间最快的算法。

算法领域并不像应用领域那样良好地被定义，算法领域更加的泛化，并且扩展了一些应用领域。算法领域和计算机科学领域更加的紧密。例如，我们可以看到，查找和图遍

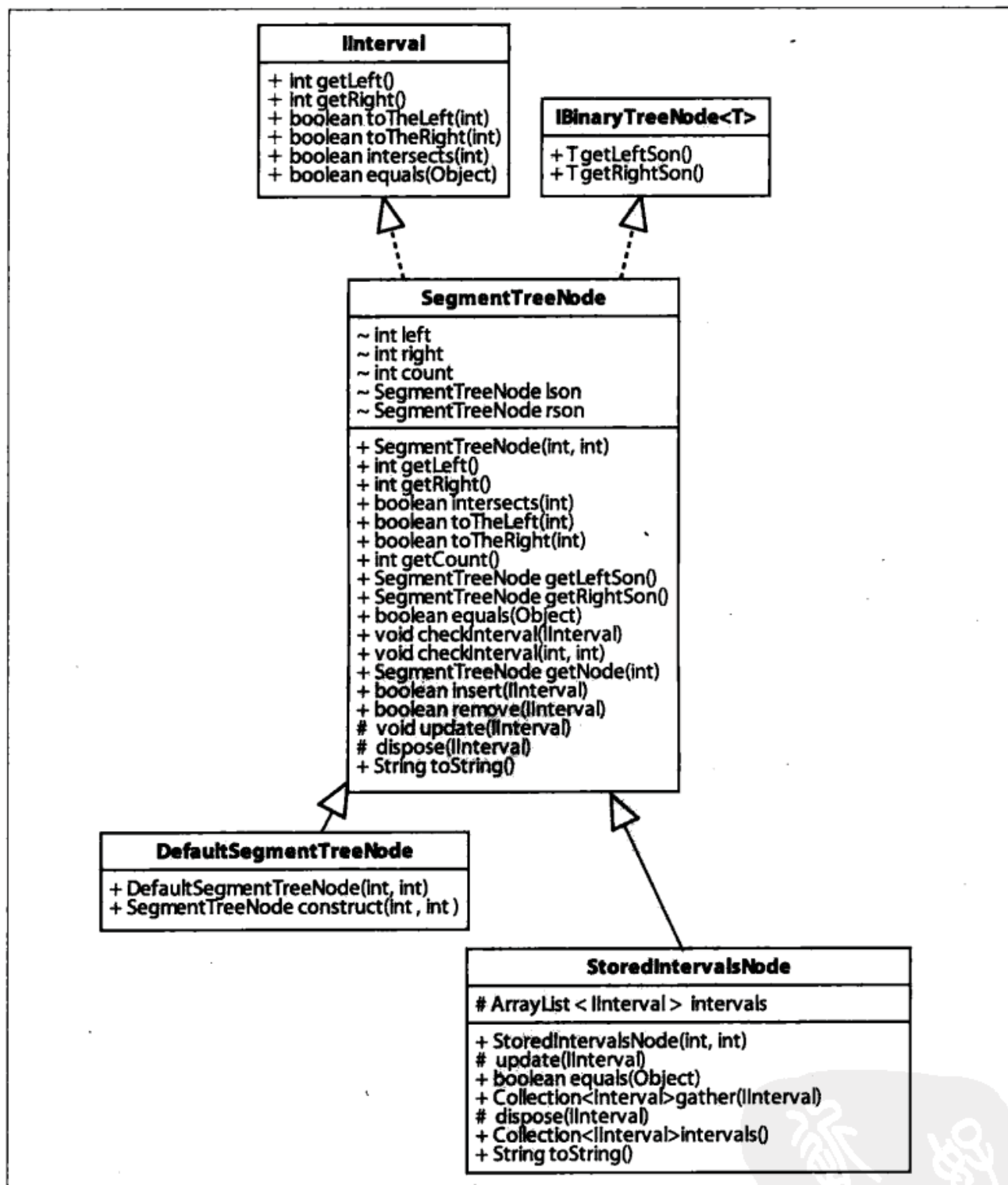


图3-3: UML图样例

历算法而不是数值算法在人工智能领域中被频繁地使用。数据库管理系统有着它们自己的算法，这些算法比在其他的应用领域中使用得都要频繁。

在我们的算法模式中，我们并不指明一个算法适用的领域，因为没有有一个标准的对照表存在。但是，每一个算法出现的使用环境给读者指明了这个算法适用的特定领域。当一

个人对算法越来越熟悉，以及开始考虑算法模式方面的事情时，一个能够良好映射应用领域和算法领域的分类就显得尤为重要。

开发一个算法领域和应用领域的映射关系是一个非常有趣和重要的研究领域。一个适当的分类能够帮助我们开发和生产更好的软件组件和应用。研究人员将高级数学分析应用到了分类研究中（Algorithm Formalization, 2007）。

从一个从业者的观点来看，算法的分类通常情况下是决定于自己的经验或者经历过的一些印象特别深刻的情境（Skiena, 1998）。在使用特定类型的算法时，这些情境给予了软件开发人员更敏锐的直觉和自信。文献，包括学术和工业的文献，充满了关于算法的这些令人印象深刻的故事，我们鼓励你得到自己的这些故事。这样一种锻炼能够帮助你更好地研究算法，并且成为一名更好的软件开发人员。算法之美是在于算法领域和适用性是在不断扩展的。新的算法不断地开发出来，但更重要的是，更多使用现存算法的新应用不断被发现。思考算法领域将能够帮助你更好地掌控这种一直非常复杂的关系。

浮点计算

计算机是有限机，设计初衷是对存储在CPU寄存器的值进行基本计算。随着计算机架构从20世纪70年代流行的8位Intel处理器增长到如今广泛接受的64位架构（如Intel的Itanium以及Sun的Sparc处理器），寄存器的大小也发生了变化。对存储在寄存器的整数来说，例如ADD、MULT、DIVIDE和SUB这些基本操作通常都由CPU支持。浮点运算单元（FPU）能够高效地处理浮点数，当然这些浮点数必须遵守IEEE关于二进制浮点算数的标准（IEEE754）。

整数值（如布尔型、8位短整型，以及16位或者32位整数）的计算通常能被处理器高效地执行。高效程序一般都会利用浮点数和整数算术的性能差异来改善程序的执行效率。当编写含有浮点计算的程序的时候，有一些问题是软件开发人员必须注意的（Goldberg, 1991）。我们在这本书的算法和代码中也会考虑到这些问题，下面我们就看看这些重要的问题究竟是什么。

舍入的错误

由于浮点数本身的表示方法，任何使用浮点数的计算都必须关注一下舍入的错误。一般来说，最初设计浮点数的时候，是用一个有限数来近似地表示一个实数，也许这个实数是无限小数。表3-1给出了3.88的浮点表示和特有的表示。

表3-1: 浮点表示

原生类型	符号	指数	尾数
Float	1位	8位	23位
Double	1位	11位	52位

3.88的表示 (0x407851ec)

01000000 01111000 01010001 11101100 (总共32个字节)

S mmmmmmm mmmmmmmmm mmmmmmmmm

eeeeeee e

接下来三个连续的浮点表示是:

0x407851ec	3.88
0x407851ed	3.8800004
0x407851ee	3.8800006
0x407851ef	3.8800008

随机选择三个浮点数，表示为：

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

用1位来表示符号，8位表示指数，23位表示尾数（也就是有效数）。在Java的浮点表示中，“将指数部分作为一个正数，然后从这个正数中减去一个基准值，得到2的幂。对于一个浮点数来说，这个基准值是126”（Venners, 1996）。指数部分是128，所以实际的指数值是 $128 - 126$ ，或2。

为了得到最高的精确度，尾数必须是规格化的，所以最左边的数字永远是1。在之前的例子中，尾数是 $[1]11110000101000111101100 = [1/2] + 1/4 + 1/8 + 1/16 + 1/32 + 1/1024 + 1/4096 + 1/65536 + 1/131072 + 1/262144 + 1/524288 + 1/2097152 + 1/4194304$ ，在进位之后，这个值是0.9700000286102294921875

因此，当使用这种表示方法存储3.88f时，近似值是 $+1*0.9700000286102294921875*2^2$ ，也就是3.8800001144091796875。固有误差是 ~ 0.0000001 。描述浮点复查的最常用方法是使用相对误差，相对误差计算的是绝对误差和期望值的一个比率。在这里，相对误差是 $0.0000001144091796875/3.88$ ，或者写作 $2.9E-8$ 。低于百万分之一的相对误差是相当普遍的。

值之间的比较

因为浮点值只是近似的，所以即使是最简单的浮点操作都变得不可信任。看看如下表达式：

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

这个表达式是真的表示两个浮点数完全相等吗？或者是表示这两个数近似相等吗（我们使用 \approx 这个符号）？如果两个值是不同的，但是相差非常小，那么它们是否应该被认为是相同的呢？在笛卡儿坐标系上给定三个点， $p_0=(a, b)$ 、 $p_1=(c, d)$ 和 $p_2=(e, f)$ ，表示两个有向线段 (p_0, p_1) 和 (p_1, p_2) 。我们能够使用 $(c-a)(f-b)-(d-b)(e-a)$ 这个式子来计算这两条线段是否共线（也就是在同一条线上）。如果这个式子的结果是0，那么这两条线段共线。让我们看看表3-2，就能够知道Java浮点计算中，误差是如何出现的。

表3-2：浮点算术误差

	浮点	双浮点
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)	4.8828125 E-4	-4.547473508864641 E-13

你能够很快地看出，这三个点是共线的，直线方程是 $y=5*x$ 。当计算浮点数的时候，浮点计算的固有误差影响了这个计算过程。使用浮点数，计算结果是0.00048828125。使用双浮点数，计算结果事实上是一个很小的负数！现在我们能够引入一个小值 d ，来决定两个浮点数是否存在 \approx 关系。如果 $|a-b|<d$ ，那么我们认为 a 和 b 是相等的。否则，就有可能是 $x \approx y$ 和 $y \approx z$ ，但是可能不是 $x \approx z$ 。这个违背了传递性，增加了我们写出正确代码的难度。

特殊的量

当浮点数能够用64位来表示的时候，IEEE标准定义了一些值来表示特殊的数字（并且不会被标准的数学计算操作所使用，例如ADD或者MULT），见表3-3。设计这些值是为了易于从共同误差中恢复，例如除以0，平方根是负数，计算时的上溢和下溢。注意正零和负零也在这张表中，虽然它们能够在计算中使用。

表3-3：特殊IEEE754量

特殊量	64位IEEE754表示
正无穷	0x7ff0000000000000L
负无穷	0xfff0000000000000L
非数 (NaN)	从0x7ff0000000000001L到 0x7fffffffffffffffffL和从 0xfff0000000000001L到 0xfffffffffffffffffL
正零	0x8000000000000000
负零	0x0000000000000000

这些特殊量是在计算出现异常时的结果。例如，正无穷在Java中作为double `x=1/0.0`的结果。有趣的是，如果这个表达式换成double `x=1/0`，Java虚拟机将会抛出`java.lang.ArithmeticException`异常，因为这个表达式是计算的两个整数的除法。

性能

一个广为接受的观点是整数计算会比浮点计算高效得多。表3-4列出了在高端计算机上，进行10 000 000次计算的时间。第三列是一台1996年的Sparc Ultra-2的结果。你可以看到，单个操作的性能根据平台的不同，会有很大的变化。

表3-4：10 000 000操作的性能

操作	Linux i686 (以秒为单位)	Sparc Ultra-2 (以秒为单位)	高端计算机 (以秒为单位)
32位整数CMP	0.0337	0.811	0.0553
32位整数MUL	0.0421	2.372	0.0723
32位浮点MUL	0.1032	1.236	0.0666
64位双浮点MUL	0.1028	1.406	0.0864
32位浮点DIV	0.1814	1.657	0.0768
64位双浮点DIV	0.1813	2.172	0.1005
128位双浮点MUL	0.2765	36.891	0.2447
32位整数DIV	0.2468	3.104	0.3061
32位双浮点SQRT	0.2749	3.184	0.2414

手动内存分配

大多数现代编程语言都允许程序员在程序执行的时候从堆（与栈相对）中分配动态内存。看看例3-1中的C程序：

例3-1：分配内存的样例程序

```
#include <stdlib.h>
#include <string.h>

void f(char *inner) {
    char temp[11];
    strcpy (temp, "algorithms");
    int i;

    for (i=0; i<11; i++) {
        inner[i] = temp[i];
    }
}

int main (int argc, char **argv) {
    char *ar1 = malloc(132);
    char *ar2 = malloc(132);
    int i = 17, j;

    f (ar2);
    return 0;
}
```

当程序执行的时候，函数的局部变量（如main的argc和argv）都是在执行栈中，执行栈存储的是程序执行的过程。动态分配的内存（如main中的ar1）是存储在堆中的，堆是内存的一块单独区域，用来进行存储空间的分配。变量的地址表示哪儿能够寻找到存储的空间。表3-5表示了一个可能的变量赋值过程（在Linux i686平台上）。

表3-5：变量和符号的地址

f的变量	地址	main变量	地址	全局符号	地址
f.inner	3221222728	main.argc	3221222796	&f	4195576
f.temp	3221222704	main.argv	3221222784	&main	4195648
f.i	3221222700	main.ar1	3221222776		
		main.ar2	3221222768		
		main.i	3221222764		
		main.j	3221222760		
		*main.ar1	5246992		
		*main.ar2	5247136		

注意，这些变量的地址相差不远，意味着它们在同一个程序中。`*main.ar1`的地址是从属于堆的地址范围的。在传统的计算机图表中，栈是在内存中“向下增长”，而堆是“向上增长”，如图3-4所示。

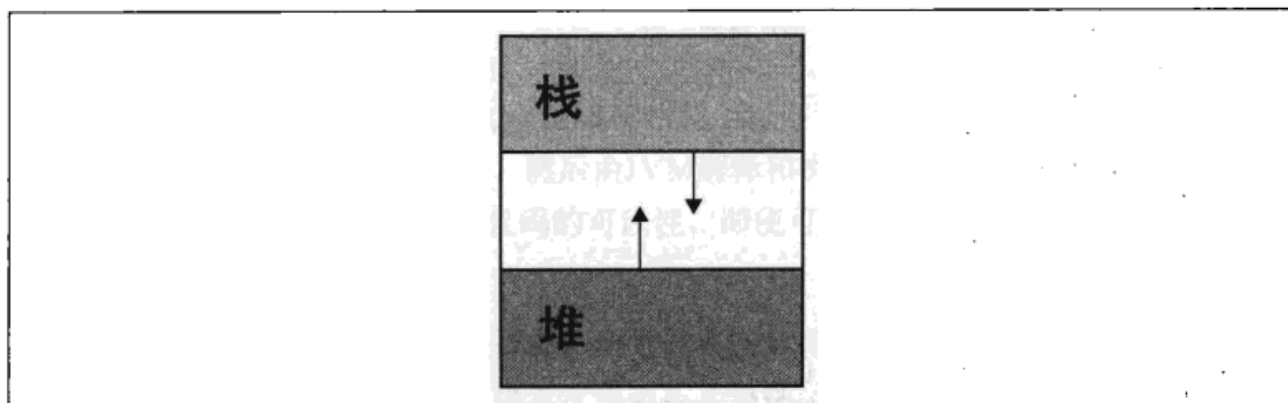


图3-4：堆和栈的动态行为

地址的变化揭示了计算机从处理`main`函数转移到处理`f`函数的过程，变量的地址是不断地向下增长的。如果栈太大的话，程序就会崩溃，因为独立栈帧的内存将会覆盖一些内存，而堆却在安全地保护着这些内存。这种交叉的发生与硬件平台以及为操作系统进程初始分配的内存有关。

在例3-2中，这个无限循环在第393 060次循环调用的时候，产生了一个“段错误”，这个时候执行栈的大小已经超过了12 577 888字节。

例3-2：无限循环的代码

```
#include <stdio.h>
```

```
int f(int n) {  
    printf (" n %d[%u] \n", n, &n);  
    return f(n+1);  
}
```

```
int main (int argc, char **argv) {  
    return f(0);  
}
```

程序执行时不断地调用函数和从函数中返回，执行栈在不断地收缩。所有属于栈的内存都会被自动地回收。但是，属于堆的内存是在程序员的控制之下，必须显式地释放。在多数情况下，如果一个程序释放内存失败，它也许可以继续正常运行（当程序结束时，它使用的内存会被操作系统回收）。

一个程序可能超出堆的空间（虽然这是一个典型的严重缺陷的标志），如果堆变得过大，那么就存在和执行栈相互干涉的风险。考虑例3-3的程序，这个程序反复地请求分配内存来存储字符串。

例3-3: 内存泄漏的代码

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

/** 将字符串“abcdef”以“fabcde”形式返回。*/
char *cycle (char *s) {
    int n = strlen(s);
    char *u = malloc (n+1);
    strncpy (u, s+1, n-1);
    u[n-1] = s[0];
    u[n] = '\0';
    return u;
}

int main (int argc, char **argv) {
    char *s = strdup ( "ThisStringHas25Characters" );
    int num=0;

    for (;;) {
        printf ( "%d\n" , ++num);
        s = cycle(s);
    }
}
```

malloc调用向内存分配系统请求一个固定大小的内存（单位是字节）。如果有足够的堆空间，分配的内存地址就会返回给程序员（否则返回空地址0x0）。在这个例子中，没有任何释放内存的操作，在经过了356 331 411次迭代之后，这个程序被手动地结束了（注1）。每一次迭代分配26个字节内存，这个程序结束的时候大概分配了这台机器上16G内存中的15.9G。一旦内存不再需要（由程序员决定），那么就必须释放，内存空间会返回给堆以备将来之用。

选择一门编程语言

在本书中，我们使用了一系列的语言来阐述算法。没有一种算法能够适用所有情况。有的时候，一种特定的语言会简单被经常使用，因为它会使用在一系列的简单的工程中。如果你对算法感兴趣，那么你应该会希望你的实现运行得尽可能快。虽然我们描述了一些实例，这些实例被仔细地进行代码优化，性能上有了可观的改进，但是这种优化或者调整超出了本书的范围。选择一种语言通常依赖于以下因素：

垃圾收集器与手动内存分配

在前面一节我们描述了C程序执行时，数据底层存储方式的细节。使用标准的内存

注1：如果你是在一台与别人共享的机器上运行这段代码就要小心了，因为它会占用所有的CPU及操作系统资源直到程序结束。

分配函数，大多数C程序员手动地按需分配和回收内存。一个替代的方法是使用Java或者Scheme这些内置了管理分配内存的垃圾收集器的语言。垃圾收集技术发展非常快速，已经存有一些包可供使用，这些包使得即使是C程序也能将垃圾回收器和默认的内存分配模式集成在一起。

字节码解释与编译代码

通常的看法是编译代码每次都能够在性能上比字节码要表现得好。例如，在Java中，Java编译器产生字节码，然后由JVM解释和执行。你应该认真地考虑是否使用像Java那样的语言，来增强代码的可读性，即使可能会导致性能上的损失。

动态与静态类型

静态类型语言增强了类型的规则，使得能够在编译的时候检测到错误，这样能够提高生产率，因为错误能够被迅速发现而不是等到运行的时候才找到。对于强类型函数式语言，例如ML，在函数语言社区，有一个普遍的看法是类型系统可以避免大多数缺陷。动态类型语言经常是解释性的，而且变量的值只有在运行时才能知道，因此不可能被静态地查找出来。很多脚本语言都提供了动态类型。

参考文献

Alexander, Christopher, *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, USA, 1977.

“Algorithm Formalization,” Software Engineering Institute, last modified January 11, 2007, <http://www.sei.cmu.edu/str/descriptions/algorithm.html>, accessed June 9, 2008.

Gamma, Erich, Richard Helm, Ralph Johnson, and John M. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

Goldberg, David, “What Every Computer Scientist Should Know About Floating-Point Arithmetic.” *ACM Computing Surveys*, March 1991, http://docs.sun.com/source/806-3568/ngc_goldberg.html.

Skiena, Steve S., *The Algorithm Design Manual*. Springer, 1998.

Venners, Bill, “Floating Point Arithmetic: Floating-Point Support in the Java Virtual Machine.” *JavaWorld*, 1996, <http://www.artima.com/underthehood/floating.html>.

第二部分

第4章 排序算法

第5章 查找

第6章 图算法

第7章 人工智能中的寻路

第8章 网络流算法

第9章 计算几何





概述

图4-1给出了一个状态列表，从这个列表中判断状态“Wyoming”是否存在，你需要多久可以完成？也许你用不了1秒就能回答这个问题。那么如果这个列表增至1000个单词，你需要花多少时间呢？乐观地估计需要100秒吧，因为这个问题的数量级增加了100倍。那么，给定同样的状态列表，你能告诉我出名字以s结尾的状态的数量吗？这个问题非常简单，你能够很快地找出答案，因为这些状态是以名字的末尾字母升序排序的。类似地，如果这个列表包含了1000个单词呢？也许只要再多花几秒就能答出来，因为你可以利用状态的有序特性。

Alabama
Florida
Alaska
Rhode Island
Delaware
Maine
Wyoming
Texas
Kansas
Vermont

图4-1：10个状态的列表

大量计算任务和作业因为进行了合理的排序预处理而变得简单。所以在计算机发展的早期，研究人员将大多数精力放在寻找高效的排序算法上。实际上，很多早期的算法研究都关注于大数据量的排序，早期计算机的内存不能完整地存储那些数据。相比50年前的计算机，现代计算机在计算能力和速度上都有了惊人的发展，这些需要处理的数据的数据量现在通常已经达到了太字节（Terabyte）的级别。虽然不会有人让你手工排序如此巨大的数据集，但是你还是有可能需要给数万或者数十万的数据集进行排序。在这章

里，我们将介绍那些最重要的排序算法，并且根据我们的标准给出评估结果，或者告诉你哪一个算法适合解决你的问题。

术语

现在有一个可比较的数据集合 A 需要进行排序，我们用 $A[i]$ 和 a_i 来表示这个集合中的第 i 个元素。从习惯上来说，这个集合中的第一个元素被记做 $A[0]$ 。为了标记方便，我们用 $A[low, low+n]$ 表示含有 n 个元素的从 $A[low]$ 到 $A[low+n-1]$ 的子集合，但是 $A[low, low+n]$ 却表示含有 $n+1$ 个元素的从 $A[low]$ 到 $A[low+n]$ 的子集合。

对这个集合进行排序的要求是：如果 $A[i] < A[j]$ ，那么 $i < j$ 。如果有重复的元素，那么在结果集合中，这些重复的元素必须被相邻地排列在一起。也就是说，在一个有序的集合中，如果 $A[i] = A[j]$ ，那么不存在 k ，使得 $i < k < j$ 并且 $A[i] \neq A[k]$ 。

有序序列 A 是一个从原始的 A 序列的元素的排列。虽然有人认为根据原始的无序序列来生成新的有序序列是简单可行的，但出于对效率的考虑，下面描述的算法将会用排好的序列覆盖原始的序列。

表述

这个集合可能已经存储在计算机的随机存储器（RAM）中，当然也有可能只是存储在文件系统的某个文件上，而文件系统通常叫做二级存储器。这个数据集合也有可能被存档在一个第三级的存储器上，此时在存储器上寻找数据就需要额外的处理时间。而且，这些数据在处理之前需要复制到第二级存储器上。第三级存储器一般包括磁带和光学光碟柜。

数据在随机存储器上一般以两种形式保存：基于指针的存储和基于值的存储。假设有人想排序“eagle”、“cat”、“ant”、“dog”和“ball”这些字符串。使用基于指针的存储，如图4-2所示，一个数组（图中连续的方块）包含了指向实际信息（椭圆中的字符串）的指针，而不是直接将信息存储在数组元素的存储空间里面。使用这种方式，我们能够灵活地存储和排序任意复杂结构的数据。

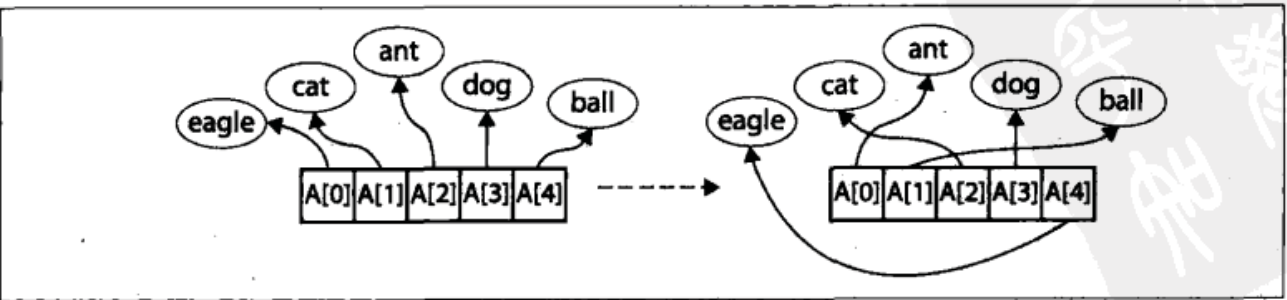


图4-2：使用指针排序

相反地，基于值的存储将 n 个元素的数据集合打包存储在固定大小的记录块中，这个固定大小定为 s （这种方法比较适合于第二级或者第三级存储器）。图4-3表示了如何使用每个大小为5字节的连续存储块来存储图4-2中的数据。在这个例子中，我们处理的数据是字符串，不过同样也可以是其他的结构化的基于记录的信息。“~”字符表示字符串的结束。在编码中，长度为 s 的字符串并不包含终止字符。这个字符串集合中的字符串是紧挨着的，并且可以看做是一个一维的数组 $B[0, n*s)$ 。注意， $B[r*s+c]$ 表示第 r 个单词的第 c 个字符（ $c \geq 0, r \geq 0$ ），同样的，这个数据集合的第 i 个元素（ $i \geq 0$ ）是子序列 $B[i*s, (i+1)*s)$ 。

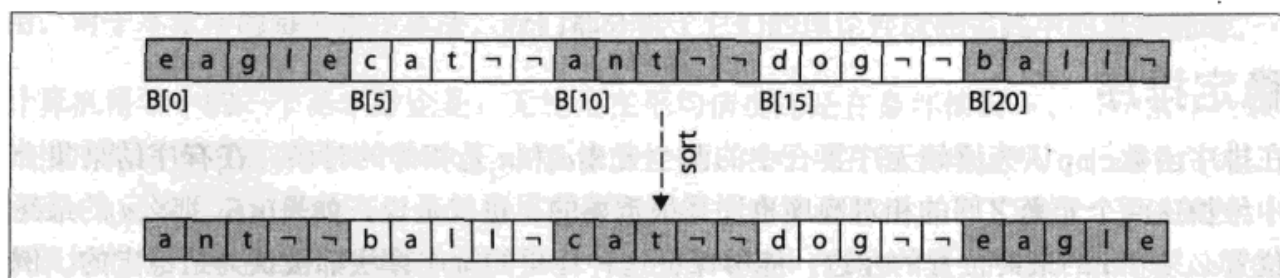


图4-3：基于值的排序

在第二级存储器上存储的信息一般都是基于值的连续字节集合。在第二级存储器上保存一个整数值来表示指针，指向一个有序集合存在磁盘文件的偏移量是可行的。本章将要介绍的算法也能够对磁盘上的数据进行排序，只需要简单地实现在磁盘文件之间交换字节的函数即可。但是，因为对第二级存储器的存取操作会增加很多输入输出请求，性能将会大幅下降。

无论是基于指针的存储还是基于值的存储，一个排序算法都使得数据（在两种情况下，方框均表示数据本身）变得有序。为了简单起见，即使是使用基于值的存储时，我们都将使用 $A[i]$ 来表示第 i 个元素。

可比较的元素

在待排序的数据集合中的元素必须是全序的。也就是说，对于数据集合里的任意两个元素，它们之间的关系只有以下三种表述的一种： $p=q$ 、 $p<q$ 或者 $p>q$ 。通常排序的原始类型包括整数、浮点值和字符。当复合元素（如字符串）需要被排序时，复合结构中的每个对应元素的词典序就用来决定这些复合结构的顺序位置，这样就将复杂的排序降为原始类型的排序了。例如，单词“alphabet”被看做比“alternate”小但是比“alligator”大，这是按照这样的规则来排序的，从左至右对比每一个字母直到一个单词末尾或者这个单词中的字母比其他单词中的同样位置的字母大或者小（如ant比anthem小）。考虑了如下因素之后，排序问题就变得复杂了，例如大小写（“A”是否大于“a”），变音符号（“ ”是否小于“ ”）和双元音（“ ”是否小于“a”）。注意，强大的Unicode

标准（参考<http://www.unicode.org/versions/latest>）对字符进行了编码，例如UTF-16利用4个字节来表示每个字符。Unicode协会（www.unicode.org）已经开发了一个排序标准（叫做“校对算法”）来处理不同语言和文化中的各种各样的排序法则（Davis和Whistler, 2008）。

对于本章中将要讨论的算法，我们假设存在一个比较函数，叫做`cmp`，用来比较 p 、 q 两个元素，如果 $p=q$ ，返回0； $p<q$ 返回负数； $p>q$ 返回正数。如果元素是复杂结构的数据，`cmp`函数也许只是比较这些元素的“键值”。例如，一个有着视频显示的机场候机楼只是按照目的地或者离港时间升序显示离港航班，而航班号却是无序的。

稳定排序

在排序函数`cmp`认为原始无序集合中的两个元素 a_i 和 a_j 是相等的时候，在有序结果集合中维护这两个元素之间的相对顺序也许是很重要的，也就是说，如果 $i<j$ ，那么 a_i 的最终位置必须在 a_j 的最终位置的左边。能够保证这种性质的排序算法都被认为是稳定的。例如，在图4-4的顶部表示了一个已经按照当天航班时间（忽略航线或者目的地这两个参数）排好序的航班信息集合A。如果A使用一个排序函数（`cmpDestination`）按照目的地来对航班进行排序，得到的一个可能结果如图4-4的底部所示。

目的地	航空公司	航班	起飞时间
Buffalo	Air Tran	549	10:42 AM
Atlanta	Delta	1097	11:00 AM
Baltimore	Southwest	836	11:05 AM
Atlanta	Air Tran	872	11:15 AM
Atlanta	Delta	28	12:00 PM
Boston	Delta	1056	12:05 PM
Baltimore	Southwest	216	12:20 PM
Austin	Southwest	1045	1:05 PM
Albany	Southwest	482	1:20 PM
Boston	Air Tran	515	1:21 PM
Baltimore	Southwest	272	1:40 PM
Atlanta	Allitalia	3429	1:50 PM
目的地	航空公司	航班	起飞时间
Albany	Southwest	482	1:20 PM
Atlanta	Delta	1097	11:00 AM
Atlanta	Air Tran	872	11:15 AM
Atlanta	Delta	28	12:00 PM
Atlanta	Allitalia	3429	1:50 PM
Austin	Southwest	1045	1:05 PM
Baltimore	Southwest	836	11:05 AM
Baltimore	Southwest	216	12:20 PM
Baltimore	Southwest	272	1:40 PM
Boston	Delta	1056	12:05 PM
Boston	Air Tran	515	1:21 PM
Buffalo	Air Tran	549	10:42 AM

图4-4：机场候机楼信息的一个稳定排序

你将注意到所有具有相同到达地点的航班按照既定离港时间排好序了，因此，这个排序

算法是稳定的。一个不稳定的算法不会关注原始集合中的元素位置之间的关系（也许它会维护相对顺序，也许不会）。

分析技术

讨论排序时，肯定得分析一个算法在最好情况、最坏情况和平均情况下的性能（在第2章中讨论过）。平均情况的性能分析一般是最难进行准确量化分析的，而且需要高级的数学技巧。一个合理的可能性理解是，我们假设输入也许是部分有序的。即便是当一个算法在平均情况下已经被认为有一个令人满意的表现时，它却可能不那么适合于实际应用。对于本章中的每个排序算法，我们都分析了它们的理论性能和实践中的真实性能。

计算机科学中的一个基本结论是：无论是在平均情况还是在最坏情况下，一个基于比较的排序算法不可能得到比 $O(n \log n)$ 更快的性能。我们现在简要地证明一下这个结论。给定 n 个元素，有 $n!$ 种排列。每个使用两两比较的排序算法都相当于下面的二叉决策树。决策树的叶子就相当于下面的一个排列，每个排列至少在树中与一个叶子相对应。每条路径从根到叶子上的节点就相当于一个比较序列。这棵树的高度就是从根到叶子的最长路径上的比较节点的数目，例如，图4-5中的树的高度就是5，因为在所有的情况下，五次比较是必需的（虽然在四种情况下只需要四次比较）。

构建一个二叉决策树，使得每个内部节点表示一个 $a_i \leq a_j$ 的比较行为，叶子表示 $n!$ 的一个排列。对一个含有 n 个元素的集合进行排序，从根节点开始向下遍历，对在每一个节点中的表达式进行求值。如果表达式为真则向左遍历，否则向右遍历。图4-5表示了一个有四个元素的决策树。

一个数据集合可以构造大量的二叉决策树。即便这样，我们给定任意一个对 n 个元素进行比较的二叉决策树，我们都能够计算它的最小高度，也就是说，肯定存在含有一些叶子节点，从根节点到达这些叶子节点只需要经过 h 个比较节点。考虑一个高度为 h 的完全二叉树，这棵树所有的非叶节点都有左右子节点。这棵树总共包括 $n=2^h-1$ 个节点，高度 h 为 $\log(n+1)$ 。如果这棵树不是完全二叉树的话，也许在某些特殊情况下并不是不平衡的。但是我们知道 $h \geq \lceil \log(n+1) \rceil$ （注1）。已经证明了任何具有 $n!$ 个叶子节点的二叉决策树最少含有 $n!$ 个节点。我们只需要按照 $h = \lceil \log(n!) \rceil$ 来计算任何一个二叉决策树的高度即可。在这里，我们利用了对数的如下性质： $\log(a*b) = \log(a) + \log(b)$ 和 $\log(x^y) = y*\log(x)$ 。

$$h = \log(n!) = \log(n*(n-1)*(n-2)*\dots*2*1)$$

$$h > \log(n*(n-1)*(n-2)*\dots*n/2)$$

注1：如果 x 不是一个整数，那么ceil函数 $\lceil x \rceil$ 返回的是比 x 大的最小整数（它将 x 的小数部分直接入到下一个整数）。

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

$$h > (n/2) * \log(n/2)$$

$$h > (n/2) * (\log(n) - 1)$$

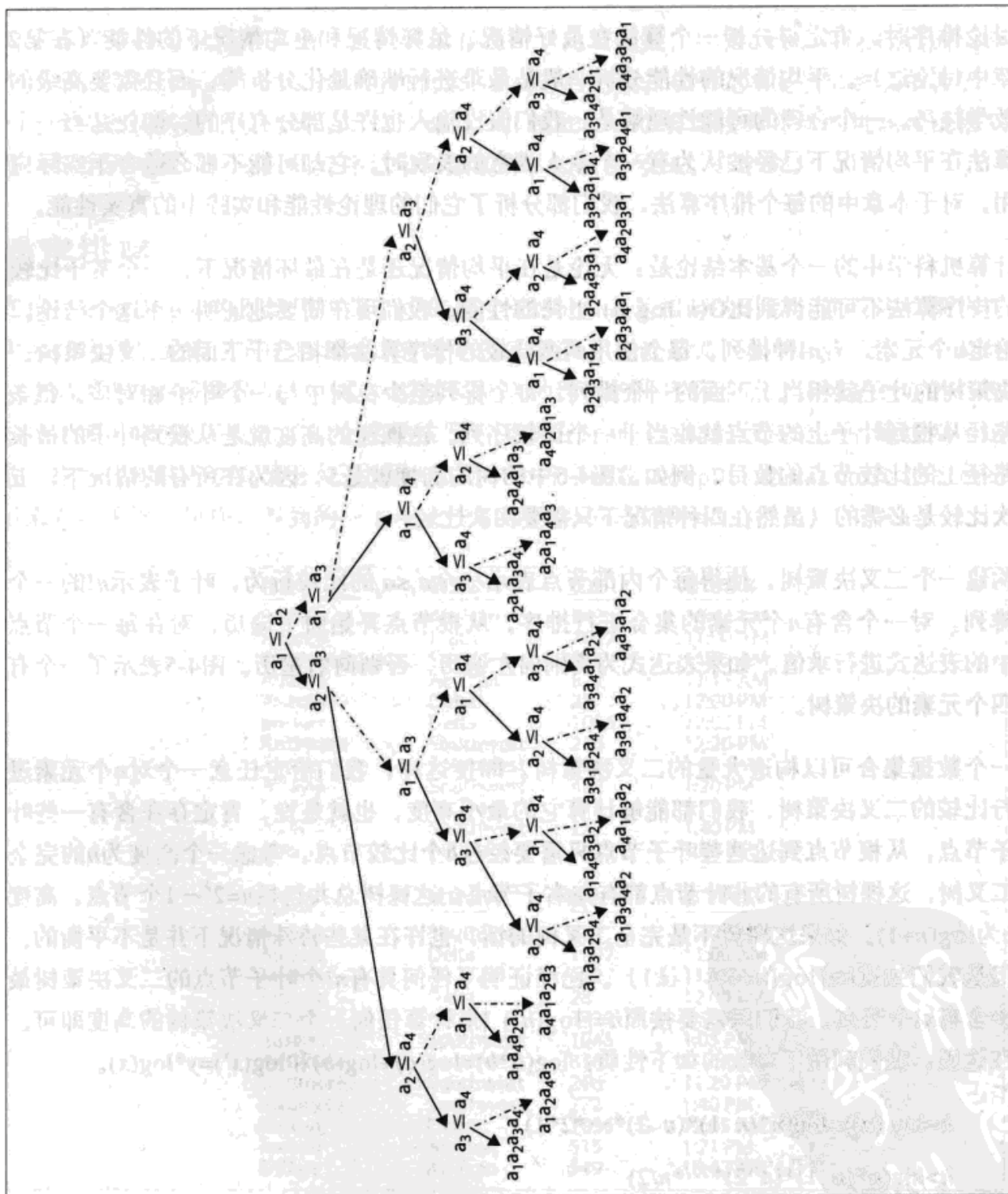


图4-5：对四个元素进行排序的二叉决策树

因此 $h > (n/2) * (\log(n) - 1)$ 。这个公式的含义是什么？给定 n 个元素对其进行排序，那么至

少存在一条路径从根到叶子，路径长度为 h ，也就是说，这个算法在排序的过程中进行至少需要 h 次比较。注意 h 是由一个函数 $f(n)$ 计算出来的，尤其在这里， $f(n)=(1/2)*n*\log(n)-n/2$ ，表示任何基于比较的排序算法至少需要大约 $O(n \log n)$ 次比较进行排序。在下面的章节“桶排序”中，我们将会看到一种不一样的算法，这种算法并不完全是基于元素之间的比较，因此在某些特殊条件下能得到比较好的性能表现。

通用输入

对于每个排序算法，我们假设存储在内存中的输入数据要么是以基于值的方式连续存储在一个内存块中，要么是用一个指针数组来指向。为了获得最大的一般性，我们假设存在一个比较函数 $\text{cmp}(p,q)$ ，就像之前描述的那样。

插入排序

在桥牌游戏中，为每个人派发13张牌，牌的正面朝下。一种排列手中牌的方法是，一次挑出一张牌来，然后插入到适当的位置。需要保持的一个特征是手中的牌必须按照花色和大小排好序。首先拿出一张牌抓在手中，这时候手中的牌是（平凡）有序的。然后对于每一张拿到的牌，找到合适的位置并且插入，这样就保证了手上的牌是有序的状态。当所有的牌都拿到之后，这个特征仍然不变，这就是算法如何工作的。当牌在你手上时，在合适的位置插入一张牌是一件非常容易的事情，因为其他的牌只需要移出一个位置即可。当数据集合存储在内存中时，排序算法就必须做更多的工作来移动数据，以使得能够留出一个位置来给插入的元素。

图4-6的伪代码描述了插入排序反复地调用`insert helper`函数来确保 $A[0,i]$ 是严格有序的，最后， i 到达了最右的元素，会对整个 A 排序。图4-7描绘了插入排序是如何在运行在一个无序的小数据集 A 上的， A 的大小为16。接下来的15行阐述了每一次插入之后 A 的状态。 pos 从1增长到 $n-1$ 的过程中，算法不断将 $A[pos]$ 插入到不断增长的有序区域 $A[0,pos]$ （这个区域被用粗体竖线隔开）的正确位置，最后 A 成为一个有序序列。灰色的元素被移到右边确保为即将插入的元素腾出位置，总体来说，插入排序执行了60次移位（一次将一个元素移动一个位置）。

使用环境

如果你有小规模数据需要排序，并且初始数据几乎是有序的，这个时候可以使用插入排序。什么样的数据集是小规模的，这个根据编程语言和机器的不同而存在不同的答案。事实上，元素的类型也是很重要的。

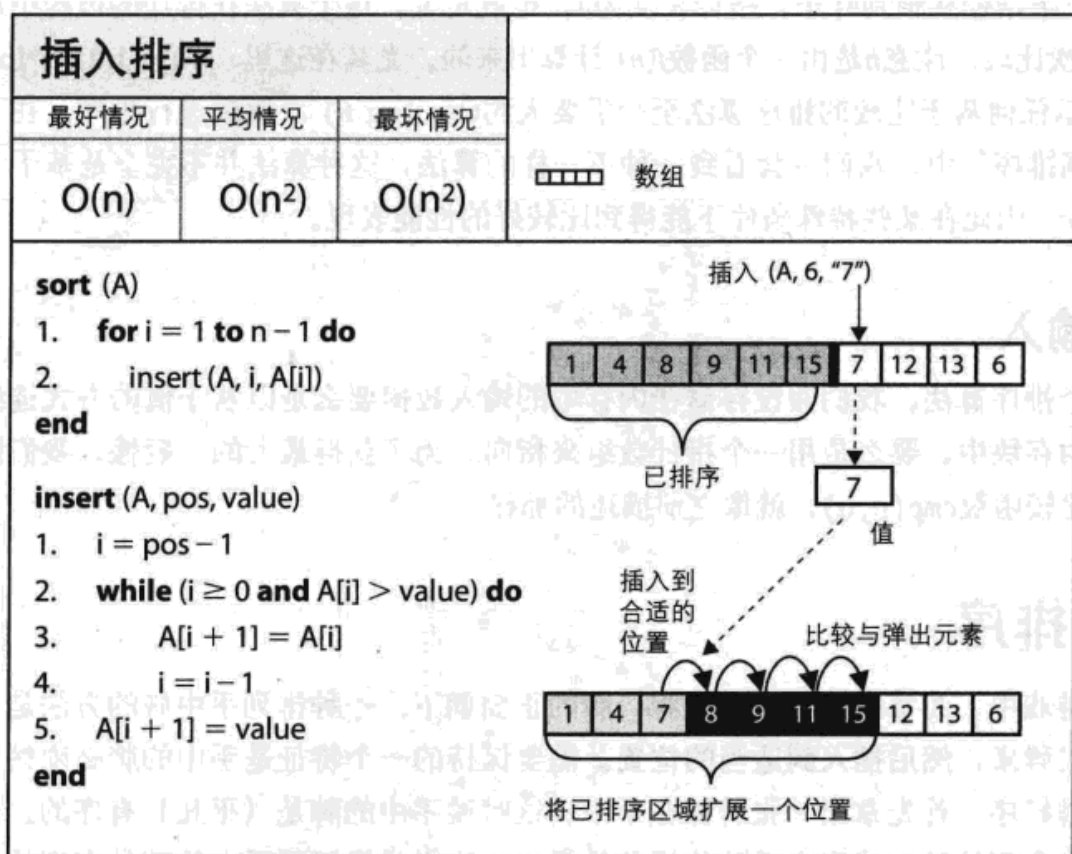


图4-6: 插入排序详解

15	09	08	01	04	11	07	12	13	06	05	03	16	02	10	14
09	15	08	01	04	11	07	12	13	06	05	03	16	02	10	14
08	09	15	01	04	11	07	12	13	06	05	03	16	02	10	14
01	08	09	15	04	11	07	12	13	06	05	03	16	02	10	14
01	04	08	09	15	11	07	12	13	06	05	03	16	02	10	14
01	04	07	08	09	11	15	12	13	06	05	03	16	02	10	14
01	04	07	08	09	11	12	15	13	06	05	03	16	02	10	14
01	04	07	08	09	11	12	13	15	06	05	03	16	02	10	14
01	04	06	07	08	09	11	12	13	15	05	03	16	02	10	14
01	04	05	06	07	08	09	11	12	13	15	03	16	02	10	14
01	03	04	05	06	07	08	09	11	12	13	15	16	02	10	14
01	03	04	05	06	07	08	09	11	12	13	15	16	02	10	14
01	02	03	04	05	06	07	08	09	11	12	13	15	16	10	14
01	02	03	04	05	06	07	08	09	10	11	12	13	15	16	14
01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16

图4-7: 插入排序在小数据集上的执行过程

驱动因素

插入排序需要一个额外的存储一个元素的空间来更好地执行。对于基于指针的排序，这是没有必要的，但是对于基于值的排序来说，程序就需要分配足够的内存来存储值（请求一个固定的值 s ，在“表示”一节中描述过）来为排序服务，排序完毕之后这块内存可以被释放。这个算法中没有复杂的嵌入循环，简单地调用一个`cmp`比较函数，一个通用的函数就能够排序很多不同类型的元素。对于基于值的排序，大多数语言的库都提供了内存块的移动函数，以使算法更加高效。

解决方案

当数据使用指针进行存储时，例4-1的C程序对一个数组`ar`进行排序，并且其中的元素可以用提供的比较函数`cmp`进行比较。

例4-1：基于指针的插入排序

```
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;
    }
}
```

当 A 是基于值的形式存储时，这个数组被装入 n 行，并且分配给它一个固定大小的内存 s 。使用比较函数对值进行比较，也就是将值从一个位置复制到另外一个位置（比较函数的栈中）。例4-2给出了一个C程序，这个程序使用了`memmove`来高效地移动数据。

例4-2：基于值的插入排序

```
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++) {
        /* 从末尾开始逆向遍历，直到i<0或者找到一个相比较小的元素。*/
        int i = j-1;
        void *value = base + j*s;
        while (i >= 0 && cmp(base + i*s, value) > 0) { i--; }
        /*如果已经在其正确的位置了，那么就没有必要移动，
        *否则保存将要插入的值并且移动这一大块内存，将其插入适当的位置。*/
        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);
}

```

结论

如图4-7所示，插入排序需要移动60个已经有序的元素。由于只有15次迭代，平均每次迭代要移动4个元素。最优性能出现在数组是顺序有序的时候，如果数组是逆序有序的时候，那么很显然是最坏的情况。如果这个数组是已经几乎有序，插入排序将会做得很好，因为只需要移动很少的元素。

分析

在最好情况下， n 个元素中每一个元素都在其正确的位置，因此插入排序将花费线性时间 $O(n)$ 。不过提这个也许看起来是没有价值的（你会有多大几率排序一个有序序列呢？），不过却是很重要的，因为插入排序是本章中讨论的唯一基于比较的排序算法。

很多现实世界的的数据都已经是部分有序了，所以若要使插入排序在现实数据上高效运行，优化和现实就会发生冲突。但是很不幸的，如果每一种输入数列排列出现概率是相等的，那么最优数据（所有元素已经有序）的概率仅仅是 $1/n!$ 。当 $n=10$ 时，360 000个输入数据才会出现一个。注意重复元素越多，插入排序的效率越高，因为插入排序需要执行更少的交换操作。

不幸的是，插入排序在随机数据面前太过于保守。如果所有 n 个元素都是不同的，而且数组是随机（数据的所有排列出现概率相等），那么每一个元素离其最终的位置平均距离是 $n/3$ （注2）。所以在平均情况和最坏情况下， n 个元素中的每个元素会被移动一个线性的距离；插入排序是二次方的运行时间， $O(n^2)$ 。

插入排序对于基于值的数据效率很低，因为很多内存需要移位，以给新的值腾出位置。表4-1包含了在直接比较。我们将会执行10次随机实验，排序 n 个元素，最好和最坏结果将会被抛弃。这个表给出了剩余8次结果的平均值。注意这个实现是如何通过内存块移动而不是通过单独的内存交换来改善性能的。当数组大小加倍的时候，执行时间大约也会乘以4，正好验证了插入算法的 $O(n^2)$ 性能。即使使用了内存块移动来改善性能，插入排序仍然是二次的，因为使用内存块移动的插入排序的性能是原生插入排序性能的常数

注2：代码库中的num Transpositions程序对于较小的 n ($n \leq 12$) 从经验上验证了这种说法，还可参见Trivedi (2001)。

倍（大约1/7）。插入排序的问题是它是一种保守的算法（叫做本地移位排序），这类算法每个元素一次移动一个位置。

表4-1：块移动的插入排序和原生插入排序的性能对比（单位：秒）

n	块移动的插入排序 (B_n)	原生插入排序 (S_n)	B_{2n}/B_n	S_{2n}/S_n
1,024	0.0055	0.0258		
2,048	0.0249	0.0965	4.5273	3.7403
4,096	0.0932	0.3845	3.7430	3.9845
8,192	0.3864	1.305	4.1459	3.3940
16,384	1.3582	3.4932	3.5150	2.6768
32,768	3.4676	12.062	2.5531	3.4530
65,536	15.5357	48.3826	4.4802	4.0112
131,072	106.2702	200.5416	6.8404	4.1449

当插入排序排序基于值的数据时，交换元素的操作将会更加的高效，编译器能够产生更高效代码，以图最少地执行高开销的内存存取操作。

中值排序

在计算机科学中，分治是一种非常常见的方法，它将一个问题分成两个独立的子问题，每个子问题的规模是原始问题规模的一半。下面让我们来看看中值排序（图4-8），在一个包含有 $n \geq 1$ 个元素的数组A排序时，它交换中值元素 $A[me]$ 和A的中位元素（第2~4行），将数组分成左右各一半。中值排序然后将左边大于 $A[mid]$ 的元素和右边小于等于 $A[mid]$ 的元素互换（第5~8行）。这样将原始数组细分成两个不同的子数组，每一个子数组的大小是原始大小的一半，这两个子数组也同样需要排序。中值排序就这样递归地应用在每一个子数组上（第9~10行）。

在图4-9中，我们一步一步地解释了中值排序的行为，图中的每一行都表示对此算法的一个递归调用。每一步问题的数量都会加倍，但是每一个问题的规模都会减少一半。因为子问题之间都是相互独立的，一旦递归结束，我们也就得到了最终排序的结果。

行1a表示最初的无序数组，中值元素 $A[me]$ 被用灰色方块标示出来。在行1b中， $A[me]$ 和中位元素（黑色方块表示）进行交换，然后大于中值元素的元素（行1b中点左边的灰色方格）和小于等于中值元素的元素（行1b中点右边的灰色方格）互换，得到行1c中的分割后的数组。在递归的每一步，每一个子数组都会被排序，行2a~2c，3a~3c和4a~4c表示了这个过程。

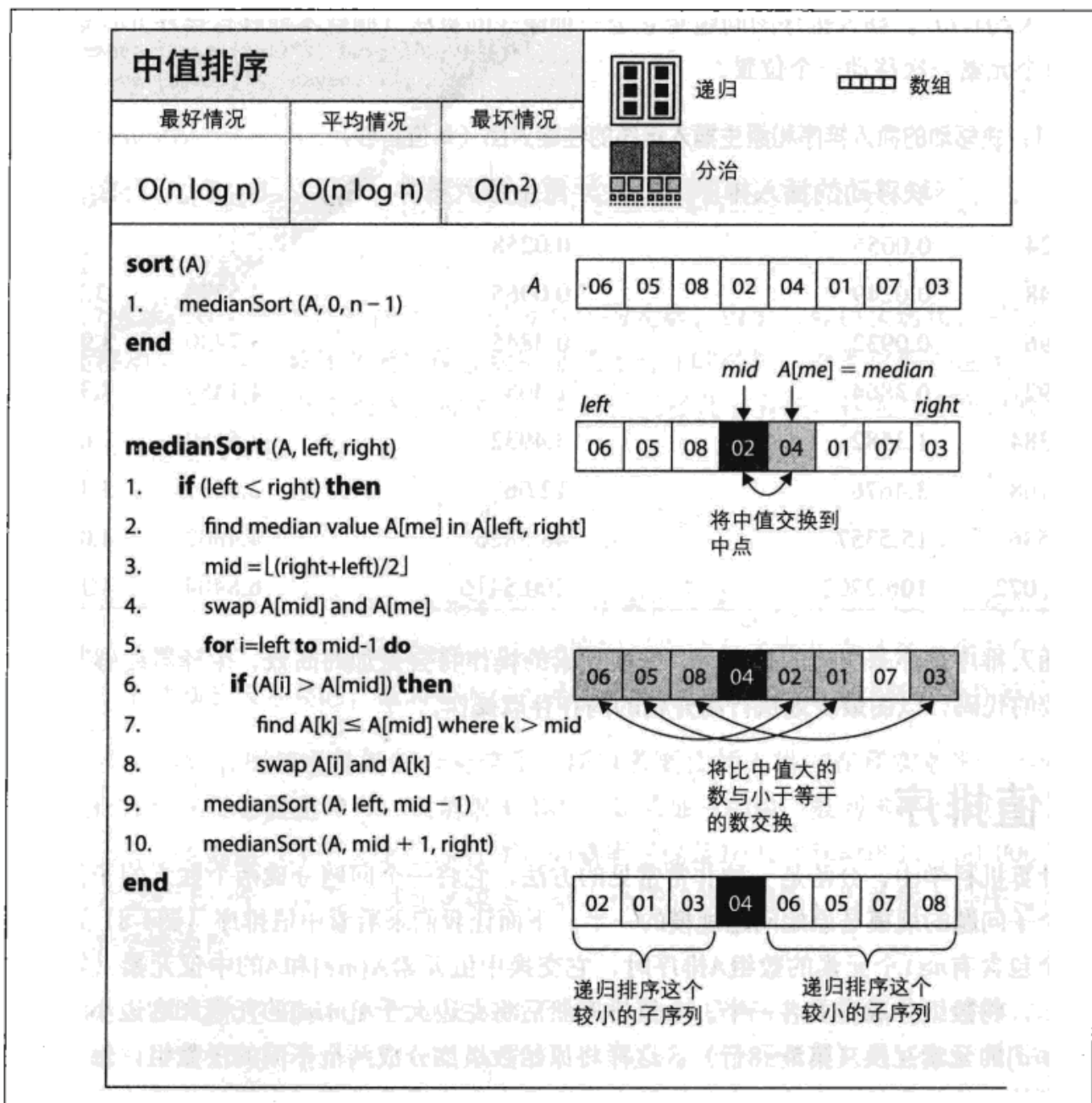


图4-8：中值排序详解

使用环境

使用中值排序的关键在于高效地从无序数组中选择出中值元素。我们先不回答这个问题，我们来看看另一个在计算机科学中非常典型的问题，这个问题最终能够给我们的初始问题一个很好的解决方案。想象一下一个人给了你一个函数 $p = \text{partition}(\text{left}, \text{right}, \text{pivotIndex})$ ，这个函数将元素 $A[\text{pivotIndex}]$ 作为一个特殊的中枢值，其将数组 $A[\text{left}, \text{right}]$ 分成两半，其中一半中所有元素都小于等于中值元素，另一半的所有元素都大于等于中值元素。注意 $\text{left} \leq \text{pivotIndex} \leq \text{right}$ ，返回值 p 是 p 在子序列 $A[\text{left}, \text{right}]$ 的最终位置。例4-3是这个 partition 函数的C实现。

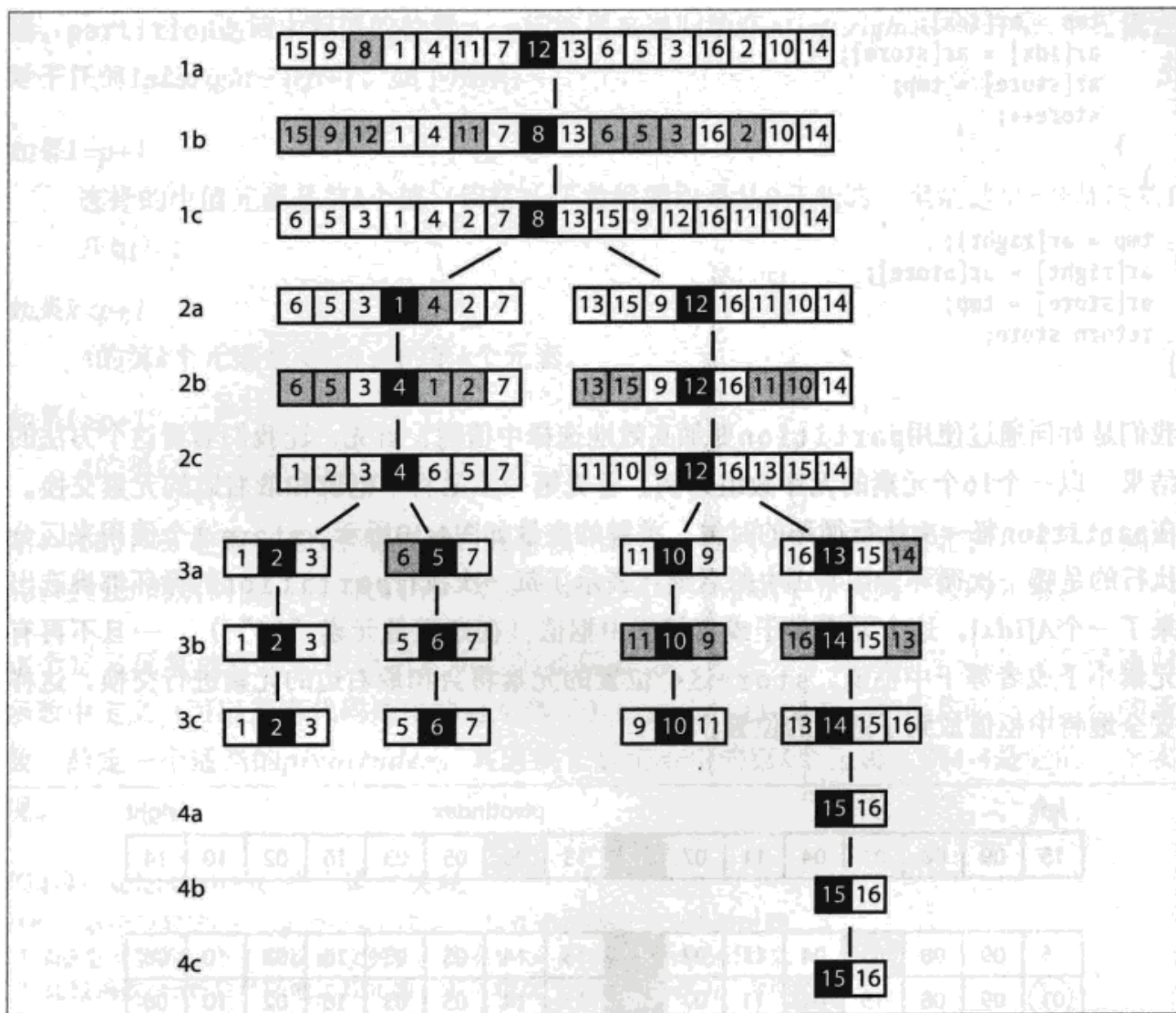


图4-9：在一个小数组上进行中值排序

例4-3：根据给定的中枢值切分数组`ar[left, right]`的C语言实现

```
/**
 * 在线性时间，根据给定的中枢值重新组合子序列ar[left, right]，
 * 将中枢值存储在其正确的位置store，确保所有在子序列ar[left, store)中的
 * 元素 $\leq$  pivot并且所有在ar[store+1, right]中的元素 $>$  pivot。
 */
int partition (void **ar, int(*cmp)(const void *, const void *),
               int left, int right, int pivotIndex) {
    int idx, store;
    void *pivot = ar[pivotIndex];

    /*将中枢值移动到数组结尾。*/
    void *tmp = ar[right];
    ar[right] = ar[pivotIndex];
    ar[pivotIndex] = tmp;

    /*所有小于等于中枢值的元素都会移动到数组的前面，然后将中枢值插在它们后面。*/
    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;
}

```

我们是如何通过使用partition更加高效地选择中值呢？首先，让我们看看这个方法的结果，以一个16个元素的无序数组为例。首先第一步是将中枢值和最右边的元素交换。在partition每一次执行循环的时候，关键的变量如图4-10所示。store这个值用来区分执行的是哪一次循环。图4-10中的后继行表示了每一次执行partition的循环都挑选出来了一个A[idx]，这个元素小于或者等于中枢值（在这里是元素“06”）。一旦不再有元素小于或者等于中枢值，store这个位置的元素将会和最右边的元素进行交换，这样安全地将中枢值放到了适当的位置。

left								pivotIndex				right			
15	09	08	01	04	11	07	12	13	06	05	03	16	02	10	14

15	09	08	01	04	11	07	12	13	14	05	03	16	02	10	06
01	09	08	15	04	11	07	12	13	14	05	03	16	02	10	06
01	04	08	15	09	11	07	12	13	14	05	03	16	02	10	06
01	04	05	15	09	11	07	12	13	14	08	03	16	02	10	06
01	04	05	03	09	11	07	12	13	14	08	15	16	02	10	06
01	04	05	03	02	11	07	12	13	14	08	15	16	09	10	06
01	04	05	03	02	06	07	12	13	14	08	15	16	09	10	11

中值位置

图4-10: partition(0, 15, 9) 返回5，并且相应地更新了A

partition(0,15,9)执行并且返回 $p=5$ ，你能看到 $A[left,p)$ 中的所有元素都小于等于中枢值，反之， $A[p+1,right]$ 中的元素都大于等于中枢值。选择出中值元素的过程是怎样的呢？注意到 p 是在中值在有序数组的最终位置的左侧（标记为“中值位置”的黑色元素）。因此， p 的左边没有元素会是中值！我们只需要反复调用partition（这次是一个不同的在 $A[p+1,right]$ 中的 $A[pivotIndex]$ ）知道其返回 p 等于中值位置。

注意到partition高效地将数组组织成两个不同的子数组，但是并没有排序每一个元

素。partition返回中枢值的位置 p ， p 能够用来递归地在 $A[\text{left}, \text{right}]$ 中检查第 k 个元素，对于任何 $1 \leq k \leq \text{right} - \text{left} + 1$ ，如下所示：

如果 $k=p+1$

选择的中值元素是第 k 个值（回忆一下数组索引是从0开始的，但是这里的 k 却是从1开始）。

如果 $k < p+1$

A 的第 k 个元素是 $A[\text{left}, p]$ 的第 k 个元素。

如果 $k > p+1$

A 的第 k 个元素是 $A[p+1, \text{right}]$ 的第 $k-p$ 个元素。

图4-10的目标是确定 A 的中值位置，或者换句话说，第八大（ $k=8$ ）的元素。得到了调用partition的返回值 $p=5$ ，我们下一步递归地在 $A[p+1, \text{right}]$ 中寻找第二大的元素。

这个定义反复地在递归中使用，不过它也能够迭代地在一个循环而不是在一个尾递归函数中定义（可以参考代码库中的这个例子）。selectKth是一个平均时间为线性的函数，给定一个适当的pivotIndex，其返回了数组 ar 中的第 k 个元素，例4-4是它的一个实现。

例4-4：selectKth的C语言递归实现

```
/** 在ar中寻找第k大元素的位置的算法，其在平均情况下为线性时间。
 * A随着计算的进行不断被修改更新。注意 $1 \leq k \leq \text{right} - \text{left} + 1$ 。
 * 比较函数cmp将会严格地比较元素。最坏情况下性能为二次方，即 $O(n^2)$ 。
 */
int selectKth (void **ar, int(*cmp)(const void *, const void *),
               int k, int left, int right) {
    int idx = selectPivotIndex (ar, left, right);
    int pivotIndex = partition (ar, cmp, left, right, idx);
    if (left+k-1 == pivotIndex) { return pivotIndex; }

    /*继续循环，并且缩小范围。如果我们在中值的左边，那么k将不变。*/
    if (left+k-1 < pivotIndex) {
        return selectKth (ar, cmp, k, left, pivotIndex-1);
    } else {
        return selectKth (ar, cmp, k - (pivotIndex-left+1), pivotIndex+1, right);
    }
}
```

selectKth函数必须为数组 $A[\text{left}, \text{right}]$ 选出pivotIndex来在递归中使用。如何选择这里存在多种策略，包括：

- 选择第一个位置（左边）或者最后一个位置（右边）。
- 选择一个随机位置（ $\text{left} \leq \text{random} \leq \text{right}$ ）。

如果多次出现选择了不好的`pivotIndex`，那么这个函数的性能将会退化成为 $O(n^2)$ ；但是，其最好和平均性能都是线性的，即 $O(n)$ 。

驱动因素

由于`selectKth`的特殊尾递归结构，一个非递归的实现会更直接一些。

解决方案

现在将讨论回到中值排序上来，你也许会非常惊讶，因为注意到无论`pivotIndex`的值是多少，`selectKth`都能很好地运行。而且，当`selectKth`返回的时候，不需要执行中值排序算法的第5~8行代码（图4-8），因为切分已经完成了。也就是说，左半部的元素都是小于或者等于中值的，反之，右半部的元素都是大于等于中值的。

例4-5的中值排序函数将要对 $A[0, n-1]$ 进行排序。

例4-5：中值排序的C语言实现

```
/**
 * 使用medianSort方法排序ar[left,right]。比较函数cmp将会严格地比较元素。
 */
void mediansort (void **ar, int(*cmp)(const void *, const void *),
                int left, int right) {
    /* 如果待排序的子数组只有一个（或者更少）元素，返回。*/
    if (right <= left) { return; }

    /*得到中点和中值元素位置k, (1<=k<=right-left-1)*/
    int mid = (right - left + 1)/2;
    int me = selectKth (ar, cmp, mid+1, left, right);

    mediansort (ar, cmp, left, left+mid-1);
    mediansort (ar, cmp, left+mid+1, right);
}
```

结论

中值排序做了很多不需要做的事。虽然生成的子问题是最优的（因为它们都是原始问题规模的一半左右），但产生子问题的额外开销应该考虑。在“快速排序”一节中，我们将会看到，随机地选择`pivotIndex`就足够了，这样就能够避免退化的情况（如果原始数组是有序的，这种情况很可能会发生）。

分析

中值排序保证递归子问题的规模都大致一样。这就意味着中值排序的平均性能是 $O(n \log n)$ 。但是，在最坏情况下，`partition`函数执行时间为 $O(n^2)$ ，这样就使得

中值排序的性能退化到 $O(n^2)$ 。因此，当 n 个元素几乎有序的时候，即使待递归排序的子问题是非常理想化的，总体性能也会受到影响。我们在中值排序中使用一个随机化的selectPivotIndex函数，来对最坏情况的数据进行测试，在这些数据上，selectPivotIndex总是返回最左边的位置。我们进行了10次试验，最好和最坏的结果都被排除，剩余8次实验的平均结果见表4-2。注意观察在最坏情况下，随着问题规模的加倍，中值排序的时间增加了4倍多，这就是经典的二次方算法的标志。

表4-2：在最坏情况下中值排序的性能（秒）

n	随机选择中枢元素	选择最左的元素 作为中枢元素	使用BFPRT算法 选择中值元素
256	0.000088	0.000444	0.00017
512	0.000213	0.0024	0.000436
1,024	0.000543	0.0105	0.0011
2,048	0.0012	0.0414	0.0029
4,096	0.0032	0.19	0.0072
8,192	0.0065	0.716	0.0156
16,384	0.0069	1.882	0.0354
32,768	0.0187	9.0479	0.0388
65,536	0.0743	47.3768	0.1065
131,072	0.0981	236.629	0.361

因此，看起来，任何依赖于切分的排序算法的最坏情况都会退化到 $O(n^2)$ 。事实上，因为这个原因，我们在图4-8已经指定了最坏情况。

幸运的是，有一个线性时间的选择算法，能够确保最坏情况仍然是 $O(n \log n)$ 。这个选择算法被称为BFPRT（Blum-Floyd-Pratt-Rivest-Tarjan）算法（Blum等，1973），其性能见表4-2的最后一列。在统一生成的随机化数据上执行10次试验，最好和最坏的结果都被舍弃。表4-3给出了使用不同的切分子数组方法的中值排序的性能。平均情况下，使用随机中枢选择算法的计算趋势线（见表4-3）是：

$$1.82 \cdot 10^{-7} \cdot n \cdot \log(n)$$

而BFPRT的趋势线是：

$$2.35 \cdot 10^{-6} \cdot n \cdot \log(n)$$

表4-3：平均情况下中值排序的性能（秒）

n	随机选择中枢元素	选择最左的元素 作为中枢元素	使用BFPRT算法 选择中值元素
256	0.00009	0.000116	0.000245
512	0.000197	0.000299	0.000557
1,024	0.000445	0.0012	0.0019
2,048	0.0013	0.0035	0.0041
4,096	0.0031	0.0103	0.0128
8,192	0.0082	0.0294	0.0256
16,384	0.018	0.0744	0.0547
32,768	0.0439	0.2213	0.4084
65,536	0.071	0.459	0.5186
131,072	0.149	1.8131	3.9691

因为复杂的BFPRT算法的常数更高，所以10次运行都比较缓慢，即便如此其平均情况下的执行时间为 $O(n \log n)$ 。

BFPRT选择算法能够保证性能稳定，因为这个算法在一个无序数组中选择的是一个可接受的近似中值。简单来说，BFPRT将含有 n 个元素的数组聚集成 $n/4$ 个集合，每个集合有4个元素（忽略不能组成一个大小为4的集合的元素，注3）。BFPRT然后寻找到每一个4元集合的中值，这一步的开销是多少？从早前的图4-5的二叉决策树中，你能够回忆起来只需要5次比较就能给排序4个元素，所以这一步的开销最多是 $(n/4)*5=1.25*n$ ，仍然是 $O(n)$ 。得到这些4元集合，每一个中值是其第三个元素。我们将这些集合的中值作为集合 M ， M 的中值（ me ）和原始集合 A 的中值非常近似。BFPRT算法用到的一个小技巧是递归地在集合 M 上使用BFPRT算法。编码这个算法是非常有意思的（例4-6是我们的实现，在这里我们通过递归地寻找固定距离、间隔的元素，最少化了元素交换操作）。注意到 A 中元素的 $3*n/8$ 个是显而易见的小于或者等于 me ， $2*n/8$ 个也是显而易见的大于或者等于 me 。因此我们能够保证切分的递归调用在左子数组不会坏于37.5%，在右子数组不会坏于75%。这样就保证了BFPRT的总体最坏性能是 $O(n)$ 。

例4-6：BFPRT算法的C实现

```
#define SWAP(a, p1, p2, type) { \
    type _tmp__ = a[p1]; \
    a[p1] = a[p2]; \
    a[p2] = _tmp__; \
}
```

注3： 文中描述的BFPRT算法将集合分成大小为5的组，但是在基准测试中，我们的实现在分组大小为4的时候更快。

```
}
```

```
/* 寻找在数组ar[left]、ar[left+gap]、ar[left+gap*2]、ar[left+gap*3]的  
* 4个元素的中值，在结束的时候确保ar[left+gap*2]包含有中值。  
*/
```

```
static void medianOfFour(void **ar, int left, int gap,  
                          int(*cmp)(const void *,const void *)) {
```

```
    int pos1=left, pos2, pos3, pos4;
```

```
    void *a1 = ar[pos1];
```

```
    void *a2 = ar[pos2=pos1+gap];
```

```
    void *a3 = ar[pos3=pos2+gap];
```

```
    void *a4 = ar[pos4=pos3+gap];
```

```
    if (cmp(a1, a2) <= 0) {
```

```
        if (cmp(a2, a3) <= 0) {
```

```
            if (cmp(a2, a4) <= 0) {
```

```
                if (cmp(a3, a4) > 0) {
```

```
                    SWAP(ar,pos3,pos4,void *);
```

```
                }
```

```
            } else {
```

```
                SWAP(ar,pos2,pos3,void *);
```

```
            }
```

```
    } else {
```

```
        if (cmp(a1, a3) <= 0) {
```

```
            if (cmp(a3, a4) <= 0) {
```

```
                if (cmp(a2, a4) <= 0) {
```

```
                    SWAP(ar,pos2,pos3,void *);
```

```
                } else {
```

```
                    SWAP(ar,pos3,pos4,void *);
```

```
                }
```

```
            }
```

```
        } else {
```

```
            if (cmp(a1, a4) <= 0) {
```

```
                if (cmp(a2, a4) <= 0) {
```

```
                    SWAP(ar,pos2,pos3,void *);
```

```
                } else {
```

```
                    SWAP(ar,pos3,pos4,void *);
```

```
                }
```

```
            } else {
```

```
                SWAP(ar,pos1,pos3,void *);
```

```
            }
```

```
        }
```

```
    } else {
```

```
        if (cmp(a1, a3) <= 0) {
```

```
            if (cmp(a1, a4) <= 0) {
```

```
                if (cmp(a3, a4) > 0) {
```

```
                    SWAP(ar,pos3,pos4,void *);
```

```
                }
```

```
            } else {
```

```
            }
```

```
        } else {
```

```
            if (cmp(a2, a3) <= 0) {
```

```
                if (cmp(a3, a4) <= 0) {
```

```
                    if (cmp(a1, a4) <= 0) {
```



```

        SWAP(ar, pos1, pos3, void *);
    } else {
        SWAP(ar, pos3, pos4, void *);
    }
}
} else {
    if (cmp(a2, a4) <= 0) {
        if (cmp(a1, a4) <= 0) {
            SWAP(ar, pos1, pos3, void *);
        } else {
            SWAP(ar, pos3, pos4, void *);
        }
    } else {
        SWAP(ar, pos2, pos3, void *);
    }
}
}
}
}
}

```

/* 使用间隔距离，对元素进行的特定的插入排序。 */

```

static void _insertion (void **ar, int(*cmp)(const void *, const void *),
    int low, int right, int gap) {

```

```

    int loc;
    for (loc = low+gap; loc <= right; loc += gap) {
        int i = loc-gap;
        void *value = ar[loc];
        while (i >= low && cmp(ar[i], value) > 0) {
            ar[i+gap] = ar[i];
            i -= gap;
        }
        ar[i+gap] = value;
    }
}

```

/**

* 为ar[left, right]寻找合适的pivotIndex。考虑大小为b的集合，在这个代码中，b=4。

* 原始的BFPRT算法中b=5。

* 1. 将元素分成 $\text{floor}(n/b)$ 个集合，每个集合包含b个元素，在每一个集合中寻找中值元素。

* 然后将这些中值元素组成集合M。

* 2. 如果 $|M| > b$ ，那么递归地执行上一步，直到分成少于或者等于b个集合。

* 3. 在递归的基础情况中，简单地使用插入排序来排序小于等于b个中值，

* 然后选择出这些中值中的中值。

*/

```

static int medianOfMedians (void **ar, int(*cmp)(const void *, const void *),
    int left, int right, int gap) {

```

```

    int s, num;
    int span = 4*gap;

```

/*元素不够组成集合？对这些元素插入排序，然后返回中值*/

```

    num = (right - left + 1) / span;

```

```

    if (num == 0) {

```

```

        _insertion (ar, cmp, left, right, gap);

```

/* 基础情况 */

```

        num = (right - left + 1)/gap;

```

```

        return left + gap*(num-1)/2;
    }
}

```

```

}

/* 得到所有集合的中值 */
for (s = left; s+span < right; s += span) {
    medianOfFour(ar, s, gap, cmp);
}

/* 如果仍然有足够的集合，那么增加gap的值，递归地应用在子数组[left, s-1]上
 * 否则的话执行插入排序，返回中值 */
if (num < 4) {
    _insertion (ar, cmp, left+span/2, right, span); /* 基础情况 */
    return left + num*span/2;
} else {
    return medianOfMedians (ar, cmp, left+span/2, s-1, span);
}
}

/**
 * 在ar[left, right]中寻找中值的最坏情况为线性的算法，比较函数cmp用来比较元素。
 */
int selectMedian (void **ar, int(*cmp)(const void *, const void *),
                  int left, int right) {
    int k = (right-left+1)/2;
    while (k > 0) {
        /* 选择切分时候应该根据的元素索引*/
        int idx = medianOfMedians (ar, cmp, left, right, 1);

        /**
         * 根据中值集合的中值x切分输入数组。如果能够找到第k大的元素，返回其索引，
         * 否则在A[left,pivotIndex-1]寻找第k大元素或者在A[pivotIndex+1,right]中
         * 寻找(k-p)大元素
         * 在A[Pivot Index+1, right]中。
         */
        int pivotIndex = partition (ar, cmp, left, right, idx);

        /* 注意0 <= k <= right-left, 返回值pivotIndex是left <= pivotIndex <= right */
        int p = left+k;
        if (p == pivotIndex) {
            return pivotIndex;
        } else if (p < pivotIndex) {
            right = pivotIndex-1;
        } else {
            k = k - (pivotIndex-left+1);
            left = pivotIndex+1;
        }
    }
    /*如果执行到这里，那么left=right，所以仅仅只需要返回两个中的一个作为中值。*/
    return left;
}

```

快速排序

如果我们仔细考虑中值排序的性能，我们会发现随机选择pivotIndex将仍使selectKth

的性能为线性时间。我们可以简化算法但是不会引起额外的性能开销吗？在某些情况下，这种简化实现能够快些吗？C.A.R.Hoare发明的快速排序虽然使用了大致相同的思想（这就是为什么我们首先介绍中值排序），但事实上确实比中值排序简单。

在快速排序中，我们不再寻找中值，而是通过某些策略（有时是随机的，有时是最左的，有时是中间的）来选择一个元素，这个元素将数组切分成了两个子数组。快排包含两步，如图4-11所示。首先数组根据中枢值分成两个数组，左子数组的元素都小于或者等于中枢值，右子数组的元素都大于或者等于中枢值。然后每个子数组被递归地排序。

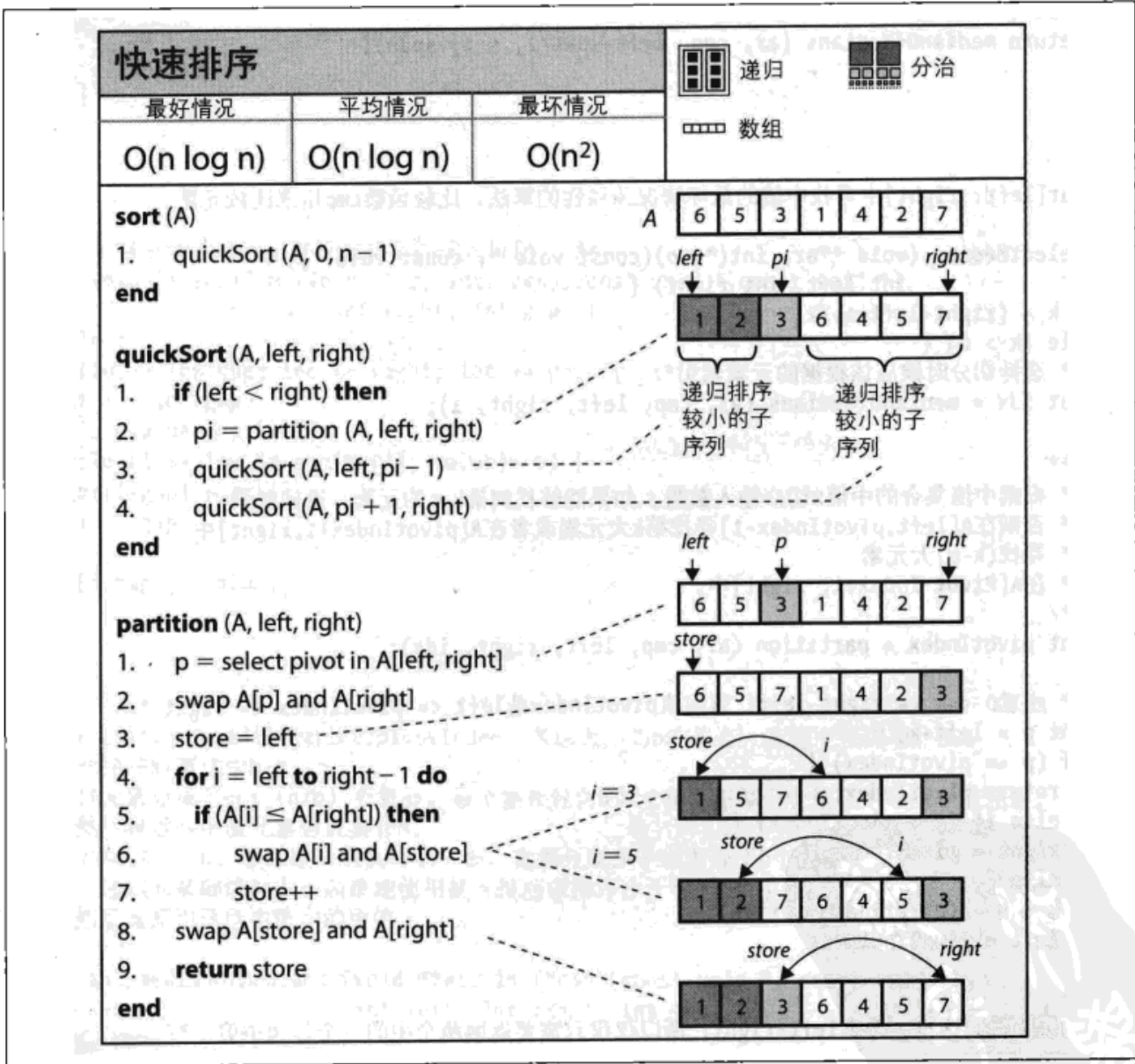


图4-11：快速排序详解

就像描述的那样，此算法的随机性使得我们很难证明平均情况下的性能仍然为 $O(n \log n)$ 。在这里我们不会介绍得到这个结果所使用的高级数学分析工具，更多的细节在Cormen等（2001）中可以找到。

图4-12中的是快速排序的行为。图中的黑色方块表示切分函数第一行中的中枢值。选择的第一个中枢值是“2”，这是一个质量比较差的选择，因为这个值将数组切分成两个数组，一个大小为1，另一个为14。在下一次快速排序的递归调用，右边数组中的“12”被选中作为中枢值，这个中枢值切分成两个大小为9和4的子数组。在这里，你已经可以看到使用切分的好处，因为数组中的最后四个元素是最大的四个。因为中枢值选择的随机性，所以会有很多不同的行为。在不同的执行过程中，如图4-13所示，第一次选择的中枢值精确地将这个问题细分成两个可比较的问题。

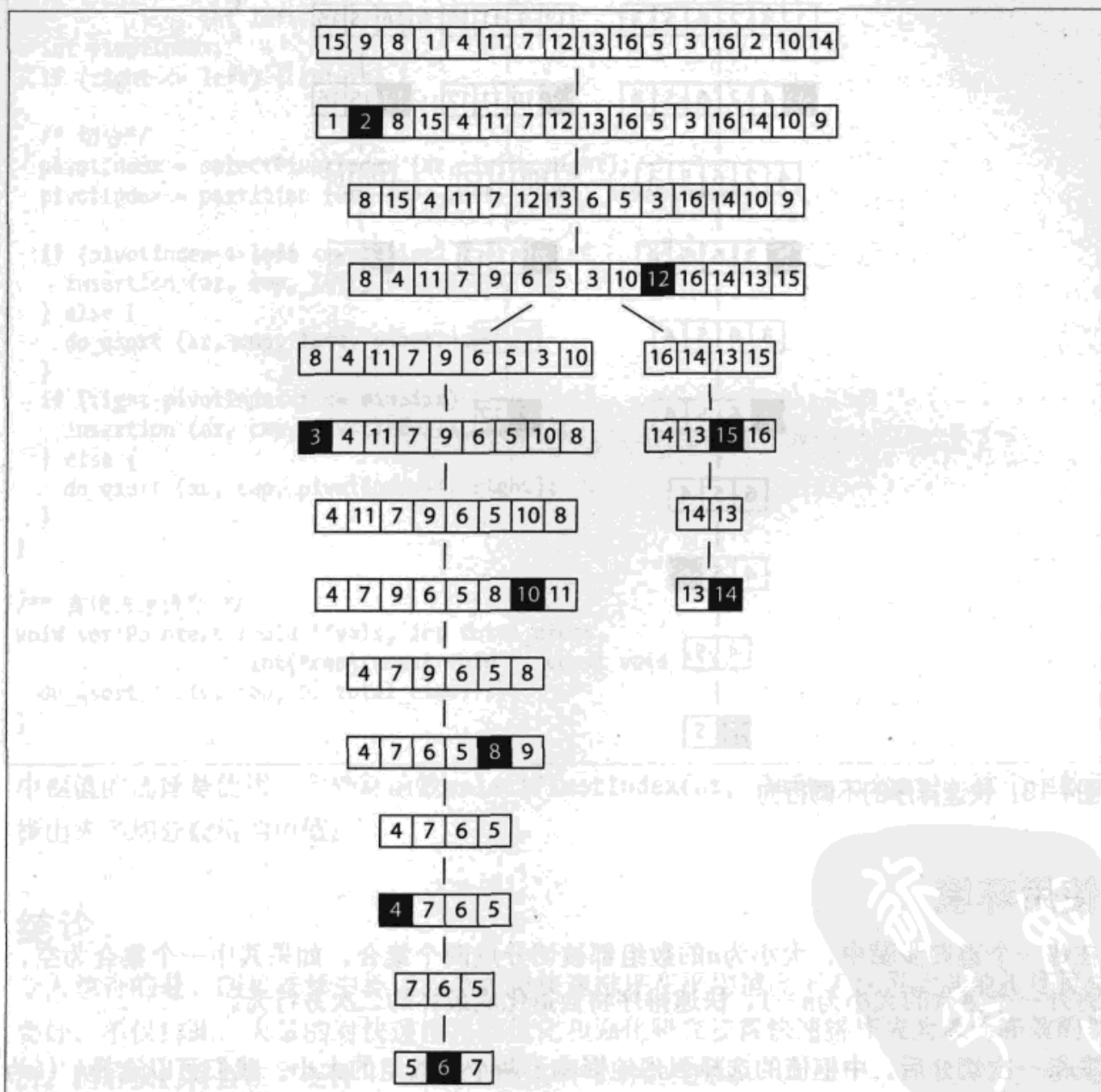


图4-12：快速排序的样例

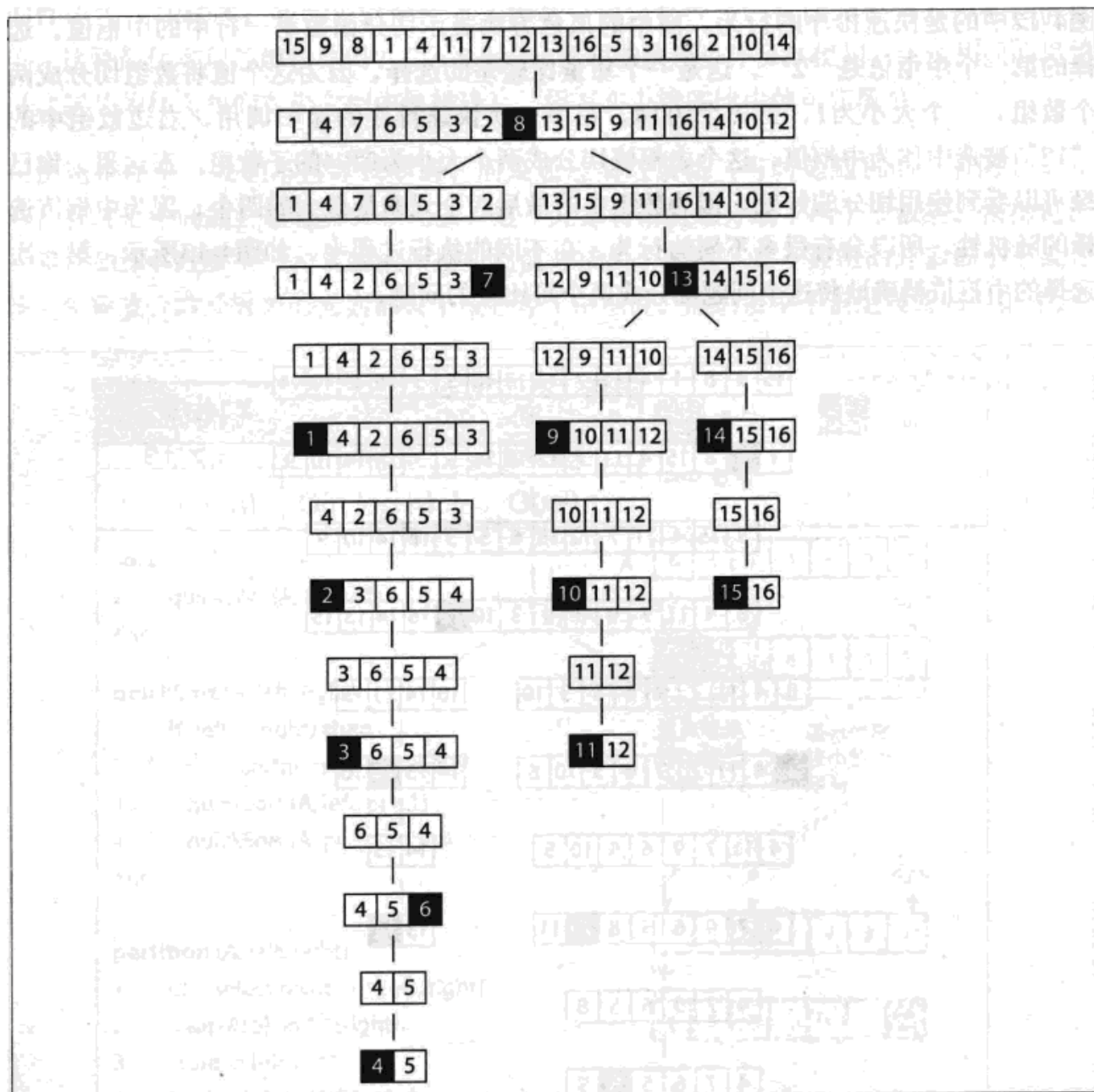


图4-13: 快速排序的不同行为

使用环境

在每一个递归步骤中，大小为 n 的数组都被切分成两个集合，如果其中一个集合为空，另外一个集合的大小为 $n-1$ ，快速排序将会退化到最坏的二次方行为。

在每一次切分后，中枢值的选择强烈地影响了两个子数组的大小。我们可以选择 k （ k 为奇数）个随机元素的中值，而不是选择一个随机元素。在随后的“变种”一节中，我们将讨论不同的选择中枢值的策略。

解决方案

例4-7中实现的快速排序使用了一些在“中值排序”中已经介绍过的函数。我们使用了一个标准的优化技术，即当待排序的数组大小低于某个预先设定的值时，我们使用插入排序。

例4-7：快速排序的C语言实现

```
/**
 * 使用快速排序来对ar[left, right]排序。比较函数cmp用来严格地排序元素
 */
void do_qsort (void **ar, int(*cmp)(const void *, const void *),
               int left, int right) {
    int pivotIndex;
    if (right <= left) { return; }

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

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

/** 直接快速排序 */
void sortPointers (void **vals, int total_elems,
                  int(*cmp)(const void *, const void *)) {
    do_qsort (vals, cmp, 0, total_elems-1);
}
```

中枢值的选择是使用一个外部函数selectPivotIndex(ar, left, right)，这个函数选择出来了切分数组的中值。

结论

令人惊奇的是，随机选择中枢值的算使得快速排序在平均情况下表现得比其他排序算法要好。不仅如此，大量的对快速排序的优化和强化研究使得快速排序成为最为高效的算法。稍后我们将会“变种”一节中讨论各种变种的细节。

分析

在理想情况下，切分操作平分原始数组，如果在每一次递归的时候都能够保持平分，那

么性能将会和中值排序一样，而且不会有额外的性能开销。我们定义 $t(n)$ ，表示快速排序 n 个元素的数组所花费的时间。因为快速排序是递归的，所以我们能够认为：

$$t(n) = 2 * t(n/2) + O(n)$$

$O(n)$ 表示切分数组需要花费线性时间。第2章已经介绍过，如果一个算法能够用这样的 $t(n)$ 来定义行为，那么其性能为 $O(n \log n)$ 。从大量实践的经验上来看，因为被抽象成为 $O(n)$ 的常数比较小，快速排序要比中值排序快。快速排序的总开销也比较少，因为它不需要在构建子问题时寻找中值元素。实际上，在实践中，即使是随机选择中枢值也已足够，这样的话，在快速排序每一次递归的时候只需要进行一次切分操作（而中值排序需要递归地调用切分操作来寻找中值元素）即可。

最坏情况下，最大或者最小的元素被挑选成为中枢元素。当这种情况发生的时候，快速排序将会遍历一遍数组中所有的元素（线性时间），仅仅排序数组中的一个元素。在最坏情况下，这个过程将会重复 $n-1$ 次，导致性能退化到 $O(n^2)$ 。

变种

快速排序是大多数系统选择的排序算法。在基于UNIX和Linux的系统中，有内建的库函数`qsort`（注4）。操作系统经常使用的是优化过的快速排序算法。在论述如何优化的资源中，两个经常被引用的是Sedgewick（1978）和Bentley与McIlroy（1993）。

各种各样的优化：

- 利用存储子任务的栈来消除递归。
- 选择基于三值分区的中枢值。
- 设定一个使用切分时数组长度的最小值，如果小于这个值，就使用插入排序（这个值根据实现和机器架构的不同而不同；例如，在Sun4架构上，这个最小值根据经验，一般设定为4）。
- 当处理子数组的时候，首先将大的子数组压入栈中，这样来最小化栈的总大小，确保小的问题首先被解决。

认识到这样一个问题是非常重要的，那就是这些优化都不能够使最坏情况下快速排序的性能高于 $O(n^2)$ 。唯一能够确保最坏情况下 $O(n \log n)$ 性能的是使用例如BFPRT的选择算法，虽然使用这个算法会使平均情况下的性能降低，例4-6描述了BFPRT算法。如果要求最坏情况下 $O(n \log n)$ 的性能，那么考虑使用堆排序，这个算法将在接下来讲述。

注4： 值得注意的是，有些版本的Linux操作系统是使用堆排序来实现`qsort`的，在接下来的“堆排序”一节中将会讲到。

选择中枢值

从一个子数组 $A[\text{left}, \text{left}+n)$ 中选择一个中枢元素的操作必须是高效的，不应该需要检查子数组中的所有 n 个元素。有一些改进如下：

- 选择第一个或者最后一个： $A[\text{left}]$ 或者 $A[\text{left}+n-1]$ 。
- 在 $A[\text{left}, \text{left}+n-1]$ 中选择一个随机元素。
- 选择 k 中值：在 $A[\text{left}, \text{left}+n-1]$ 随机选择 k 个元素，然后选择这 k 个元素的中值。

通常我们都选择 $k=3$ ，这个变种叫做三中值。Sedgewick描述了这个方法，并且认为这个改进能够有5%的性能提升，不过他注意到数据的某些排列将会使得算法，包括这个改进的性能在一般水准以下。一个五中值的中枢值选择也投入使用。这个改进进行更深入的计算来选择合适的中枢值，由于带来了过多开销，所以很少能够得到性能提升。这种改进使得性能和中值排序的性能很接近（事实上，中值排序也会接受这种选择 n 中值的极端变化）。

处理切分

在例4-3所示的切分函数中，小于或者等于中枢元素的值的元素都被插入到子数组的前端。如果数组中有很多和选择的中枢元素重复的元素，这个方法也许会使得子数组的大小不平衡。一种减少失衡问题的解决方法是将等于中枢元素值的元素轮流地插入到第一个和第二个子数组中。

处理子数组

快速排序通常在切分出来的两个子数组上分别进行递归调用快速排序。当处理一个子数组时，另一个的活动记录将会被压入执行栈中。如果大的子数组首先被处理，那么此时栈中可能存储了线性数目的活动记录（虽然现代编译器也许会观察到并且消除这个开销）。为了最小化栈的可能深度，那么首先处理小的子数组。

如果系统栈的深度是可预见的，那么快速排序也许不适合你的应用程序。一种打破这个僵局的方法是使用一个栈的数据结构来存储将要解决的子问题，而不是依赖于递归。

为小子数组使用简单的插入排序

在小子数组上，插入排序比快速排序要快得多，当处理大数组时，快速排序最终还是将大数组切分成无数的待排序的小数组。为了改善快速排序的递归性能，一个广泛使用的技术是在排序大子数组时调用快速排序，在排序小子数组时使用插入排序，如例4-7所示。

Sedgewick (1978) 建议在快速排序中结合三中值和排序小子数组时使用插入排序这两种手段，这样能够相比纯快速排序提高20%~25%的性能。

内观排序

是否切换到对小数组的插入排序取决于数组的大小。Musser (1997) 介绍了一个快速排序的变种, 叫做内观排序 (IntroSort), 这个算法监视快速排序的递归深度, 以确保高效地处理。如果快速排序的递归深度超过 $\log(n)$ 级, 那么内观排序切换到堆排序。C++STL的SGI实现使用了内观排序作为其默认的排序算法 (<http://www.sgi.com/tech/stl/sort.html>)。

选择排序

给你一堆标上数字的牌, 一个通常的排序方法是选择并且拿走最大的牌, 然后重复这个过程直到所有的牌都被拿走。这个方法描述了选择排序的过程, 如例4-8所示。

例4-8: 选择排序的C语言实现

```
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 *)) {
    int i;

    /*重复选择A[0, i]的最大值, 并且交换到合适位置*/
    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;
        }
    }
}
```

选择排序在本章描述的算法中是最慢的, 即使在最好情况下 (如数组已经有序) 它都需要二次方时间。它重复地进行着几乎相同的工作, 而不会从每一次迭代中学到什么东西。在数组A选择最大的元素 max 需要 $n-1$ 次比较, 选择第二大的需要 $n-2$ 次比较。很多比较都是浪费的, 因为如果一个元素比第二大的元素小, 那么它不可能是最大的元素,

所以不会对 max 的选择有任何影响。我们将不会阐述关于这个低效算法的更多细节，现在我们看看堆排序，这个算法展示了如何高效地应用选择排序背后隐藏的原理。

堆排序

在数组 $A[0,n)$ 中寻找最大的元素最少需要 $n-1$ 次比较，但是我們希望能够最少化直接比较的元素。在体育领域，通过锦标赛在 n 个队伍中寻找“最好的”队伍，而不是强制最终的优胜者和所有其他的 $n-1$ 支队伍比赛。NCAA冠军锦标赛是美国最流行的篮球赛事之一，这项赛事有64支学校的代表队参加，角逐冠军头衔。最终的冠军队伍在进入决赛之前会和5支队伍进行比赛，所以它需要赢得6场比赛。非常巧合的是 $\log(64)=6$ 。堆排序则表明了如何应用这种行为来排序元素，其伪代码描述如图4-14所示。

一个堆就是一棵二叉树，它具有以下两个性质：

外形的性质

如果深度 $k-1$ 存在 2^{k-1} 个节点，那么深度 k ($k>0$) 上存在叶子节点。另外，在一个部分填充的深度级上，节点是“从左到右”添加的。

堆的性质

树中每一个节点的值都大于或者等于任意一个子节点的值（如果有的话）。

图4-15标记为a的堆就满足这些性质。二叉树的根节点的值必须是树中最大的，但是，注意到最小的元素可以是任何一个叶子节点。虽然二叉树的有序信息受限于这样一个事实：一个节点大于任意一个子节点。堆排序表明了如何利用这个外形的性质来高效地排序元素。

给定严格满足外形性质的一个结构，堆能够存储在一个数组 A 中，而不损失任何结构信息。图4-15b描绘了给堆中每一个节点都赋予一个整数值标签。根节点被标记为0。对于每一个标记为 i 的节点，其左子节点（存在的话）被标记为 $2*i+1$ ；其右子节点（如果存在的话）被标记为 $2*i+2$ 。类似地，对于一个标记为 i 的非根节点，其父节点被标记为 $\lfloor (i-1)/2 \rfloor$ 。使用这个标记，我们能够将堆存储在一个数组中，节点存储在数组中的位置就是其标签。图4-15 c表示了存储在数组中的堆。A中元素的顺序从左至右能够被简单地看做节点被发现的深度。

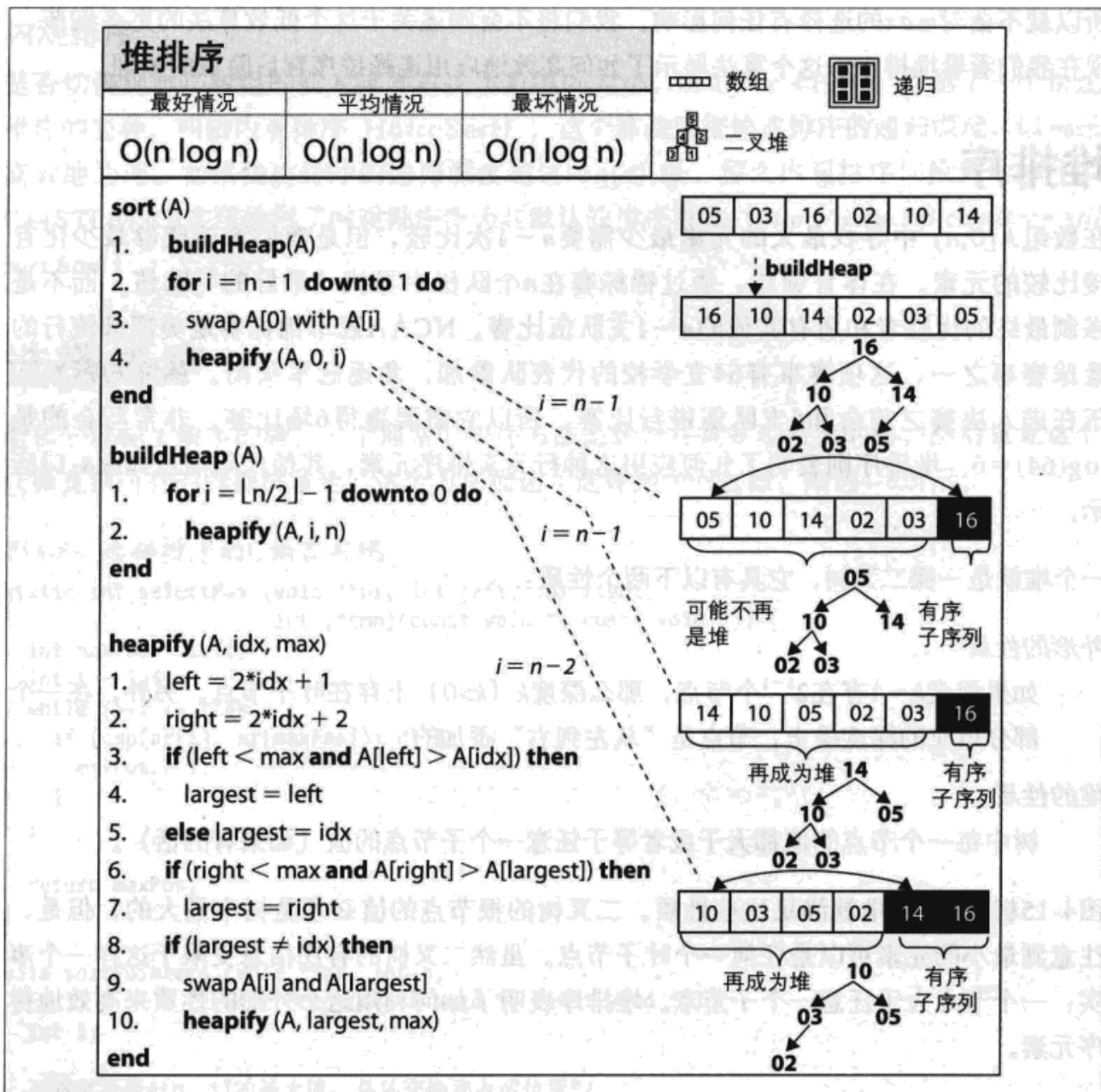


图4-14：堆排序详解

堆排序在排序数组的时候，首先会将数组中的元素放到堆中“合适的位置”。事实上，图4-15所示的堆是在一个已经有序的数组上执行buildHeap（图4-14有这个函数的伪代码）得到的结果。buildHeap执行的过程如图4-16所示。如果 $A[i]$ 的两个子节点 $A[2*i+1]$ 和 $A[2*i+2]$ 至少有一个比 $A[i]$ 大，heapify(A, i, n)通过交换 $A[i]$ 和两个子节点中较大的来更新A。如果交换过程发生了，heapify递归地检查其孙子节点来正确地维护A的堆性质。你能够看到，在结果堆中，最终大数会被“抬升”（意思是它们会和左边的较小元素交换）。图4-16灰色的方格描述了heapify在第9行被交换的元素。

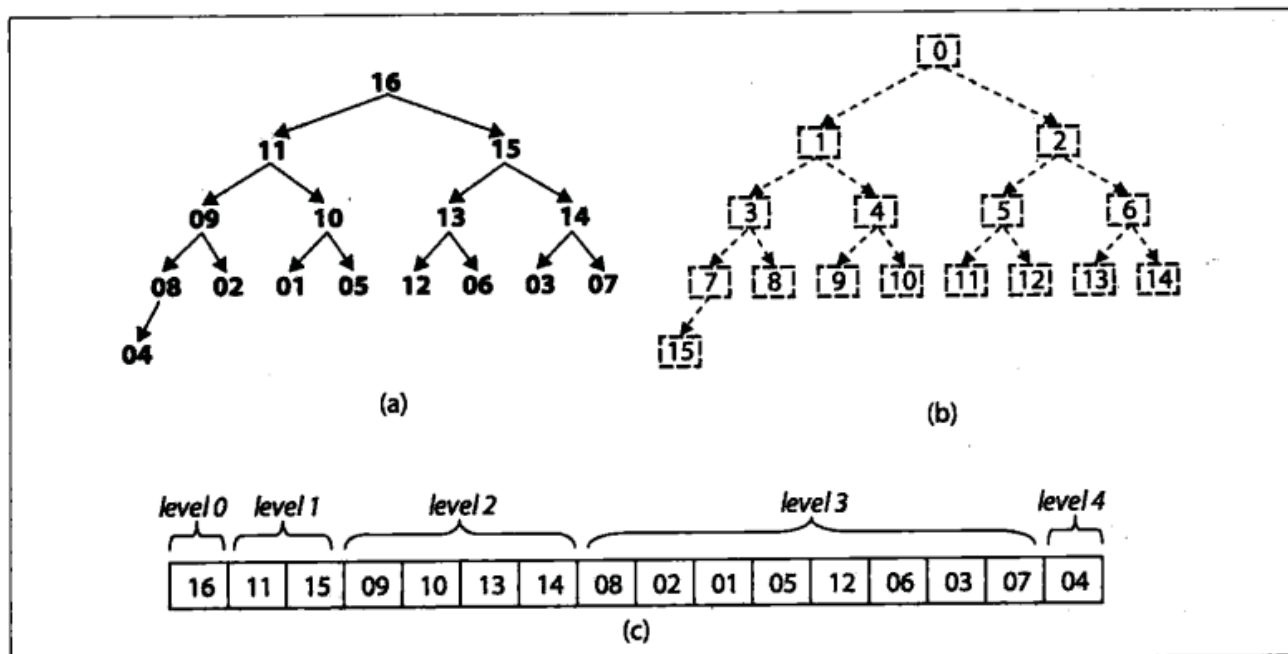


图4-15: (a) 有16个不同元素的堆; (b) 这些元素的标签; (c) 存储在数组中的堆

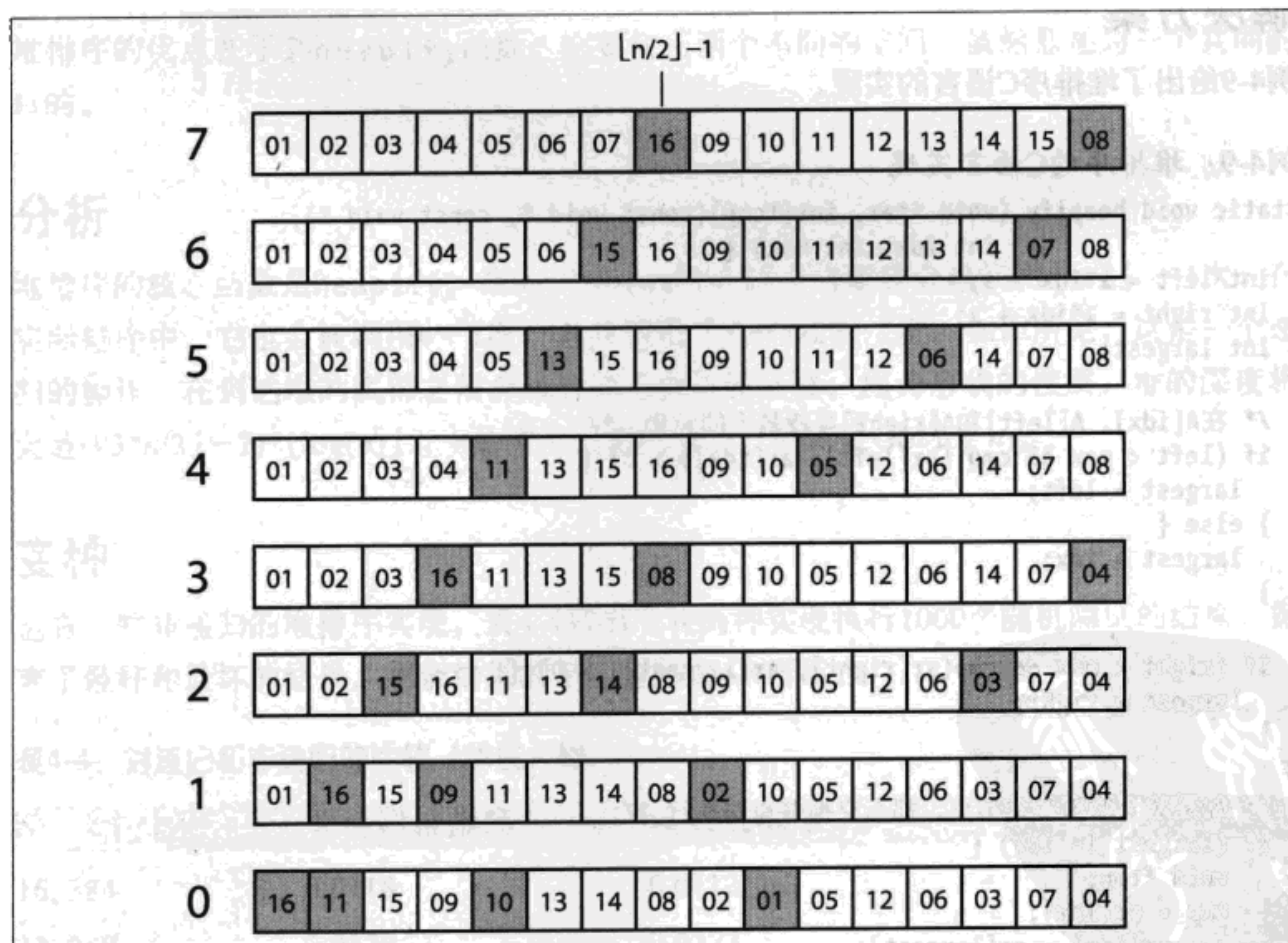


图4-16: 在一个初始有序的数组上执行buildHeap操作

在排序一个大小为 n 的数组 A , 堆排序将其切分成两个相异的子数组, $A[0, m)$ 和 $A[m, n)$, 分别表示一个大小为 m 的堆和一个有 $n - m$ 个元素的有序子数组。随着 i 从 $n - 1$ 迭代到1,

堆排序交换堆中的最大元素（位置是 $A[0]$ 和 $A[i]$ ，使子数组 $A[i,n)$ 的大小不断增长；然后堆排序执行 `heapify` 重构 $A[0,i)$ 使其成为一个有效堆（图4-14描述了伪代码）。结果非空子数组 $A[i,n)$ 将会有序，因为堆中的最大的元素保证比这个有序子数组中的任何一个元素都要小。

使用环境

堆排序不是一个稳定的排序。因为它非常频繁地移动元素，所以它不能使用基于值的数据。

驱动因素

堆排序避免了很多导致快排性能退化的令人讨厌的（大都是令人尴尬的！）情况。尽管如此，平均情况下，快速排序比堆排序表现要好。

解决方案

例4-9给出了堆排序C语言的实现。

例4-9：堆排序的C语言实现

```
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;

    /* 在A[idx], A[left]和A[right]寻找最大的元素。*/
    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;
    }

    /* 如果最大的不是父节点，那么交换并递归执行 */
    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);
    }
}

```

堆排序的优点在于其heapify函数。它要使用两个不同的空间，虽然都是为一个共同的目的。

分析

堆排序的核心函数是heapify。在buildHeap中，这个函数将会被调用 $\lfloor n/2 \rfloor - 1$ 次，在实际排序中，它也会被调用 $n-1$ 次，总共调用了 $\lfloor 3*n/2 \rfloor - 2$ 次。如你所见，这是一个递归的操作，在到达堆的底部之前会执行固定数目的计算。因为形状的性质，堆的深度将会是 $(\lfloor 3*n/2 \rfloor - 2) * \lceil \log(n) \rceil$ ， n 是堆中元素的数目。性能是 $O(n \log n)$ 。

变种

也有一些非递归的堆排序实现，表4-4列出了对两种实现执行1000个随机测试的结果，抛弃了最好和最坏的结果。剩余结果的平均值如表4-4所示。

表4-4：对递归和非递归的比较（单位：秒）

n	非递归堆排序	递归堆排序
16,384	0.0118	0.0112
32,768	0.0328	0.0323
65,536	0.0922	0.0945
131,072	0.2419	0.2508
262,144	0.5652	0.6117

表4-4：对递归和非递归的比较（单位：秒）（续）

n	非递归堆排序	递归堆排序
524, 288	1.0611	1.1413
1, 048, 576	2.0867	2.2669
2, 097, 152	4.9065	5.3249

计数排序

一个会计师负责对一个小饭店的账本进行审核。每天晚上饭店打烊时，饭店主人记录白天的总销售额，然后打印出有总额和日期的收据。这些收据存放在一个大盒子里面。每年年终，会计师审核盒子中的这些收据，检查是否有的已经丢失。你能想象，盒子中的收据都是无序状态的。

会计师可以按照日期降序地排列所有的收据，然后审核。另一种方法是，他找到一个当年的空白日历，从盒子中一条接一条取出收据来，然后在日历上相应日期用X标记。一旦盒子为空之后，会计师仅仅需要检查一下日历上哪些日子没有标记。注意第二种方法中，从来没有两个收据会相互进行比较。如果饭店已经营业了60年，并且会计师有60年的日历，如果盒子中只有5张收据，那么这个方法将会非常低效；但是有20 000张收据的话，这将是一个有效的方法。收据的数量与总日子的比例决定了方法的效率。

在本章开头，我们证明了基于比较的排序算法不可能好于 $O(n \log n)$ 的时间排序元素。令人惊异的是，如果能知道这些元素的更多信息，那么就会有其他的排序方法。例如，假设你被指定排序 n 个元素，并且告知你每一个元素的值范围都在 $[0, k)$ 之间， k 比 n 小得多。你就能够利用这个信息，使用一个线性的排序算法，叫做计数排序。

使用环境

计数排序，如图4-17所示，不需要一个比较函数，是对范围固定在 $[0, k)$ 的整数排序的最佳选择。即使 k 个元素的全序能够被决定以及每一个元素的值都是唯一的，这个算法也可用。例如，如果排序一系列形如 $1/p$ （ p 为整数）的小数， p 的最大值为 k ，那么每个小数 $1/p$ 能够被分配一个唯一的值 $k-p$ 。

驱动因素

因为元素的 k 值形式的全序关系，所以计数排序能成功进行排序。

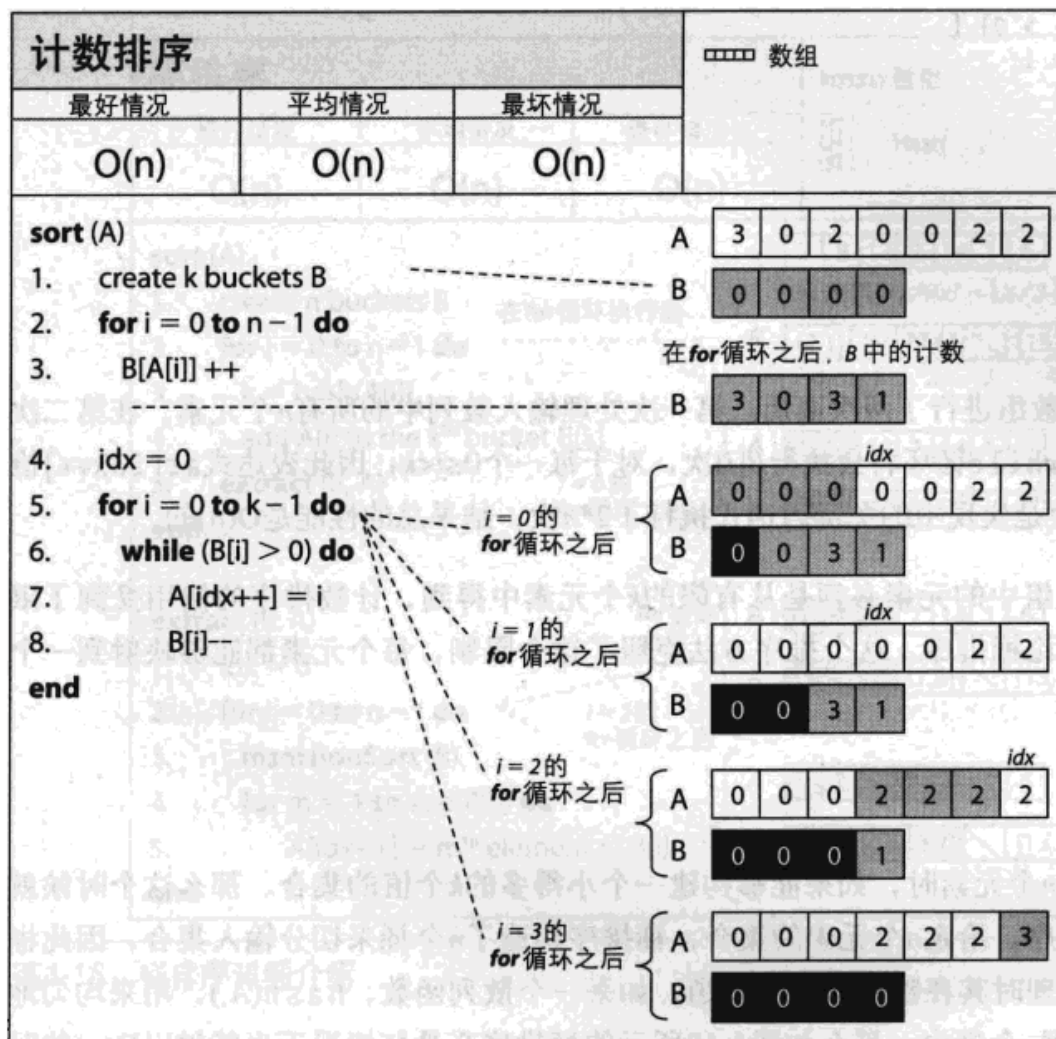


图4-17: 计数排序详解

解决方案

计数排序创建了 k 个桶用来存储输入数列中第 k 个元素值出现的次数。计数排序将对输入数列进行两次遍历。在第一次遍历中, 计数排序增加桶的计数。在第二次遍历时, 通过处理桶中得到的全序的计数值, 计数排序重写原始的数列。例4-10是计数排序的一个实现, k 的值是函数的一个参数。

例4-10: 计数排序的实现

```

/** 排序ar中的n个元素, 范围是[0, k)。*/
int countingSort (int *ar, int n, int k) {
    int i, idx = 0;
    int *B = calloc (k, sizeof (int));

    for (i = 0; i < n; i++) {
        B[ar[i]]++;
    }

    for (i = 0; i < k; i++) {

```

```

    while (B[i]-- > 0) {
        ar[idx++] = i;
    }
}

free(B);
}

```

分析

计数排序对整个数组进行了两次遍历。第一次处理输入数列中的所有 n 个元素。在第二次遍历时，内部的while循环将会执行 $B[i]$ 次，对于每一个 $0 \leq i < k$ ，因此表达式 $ar[idx++]$ 恰好执行 n 次。这个是实现中的关键表达式执行了 $2 \cdot n$ 次，结果总的性能是 $O(n)$ 的。

因为待排序的数组中的元素必须是从有限的 k 个元素中得到，计数排序的使用受到了限制。我们现在讨论桶排序，这个排序方法克服了这个限制，每个元素都能够映射到一个桶中。

桶排序

当计算集合中的 n 个元素时，如果能够构建一个小得多的 k 个值的集合，那么这个时候就可以使用计数排序。给定 n 个元素的集合，桶排序构造了 n 个桶来切分输入集合，因此桶排序能够降低处理时其在额外空间的开销。如果一个散列函数， $hash(A_i)$ ，用来均匀地将 n 个元素分配到 n 个桶中，那么如图4-18所示的桶排序在最坏情况下也能够以 $O(n)$ 的时间进行排序。如果希望使用桶排序，那么需要满足以下两点。

均匀分布

输入数据需要均匀分布在一个给定的范围内。基于这种分配，算法创建了 n 个桶来平分输入数据。

有序散列函数

桶必须是有序的。也就是说，如果 $i < j$ ，桶 b_i 中的元素要字典序小于桶 b_j 中的元素。

桶排序不适合排序随机字符串，但是，它能够被用来排序在区间 $[0,1)$ 间均匀分配的浮点数。

一旦所有待排序的元素都被放入桶中，桶排序从左至右对每个桶中的值使用插入排序。每一个桶中的元素按照顺序从左至右抽取出来重新构造原始的数组。

使用环境

当待排序元素能够使用一个快速散列函数均匀分配时，桶排序是最快的排序法。

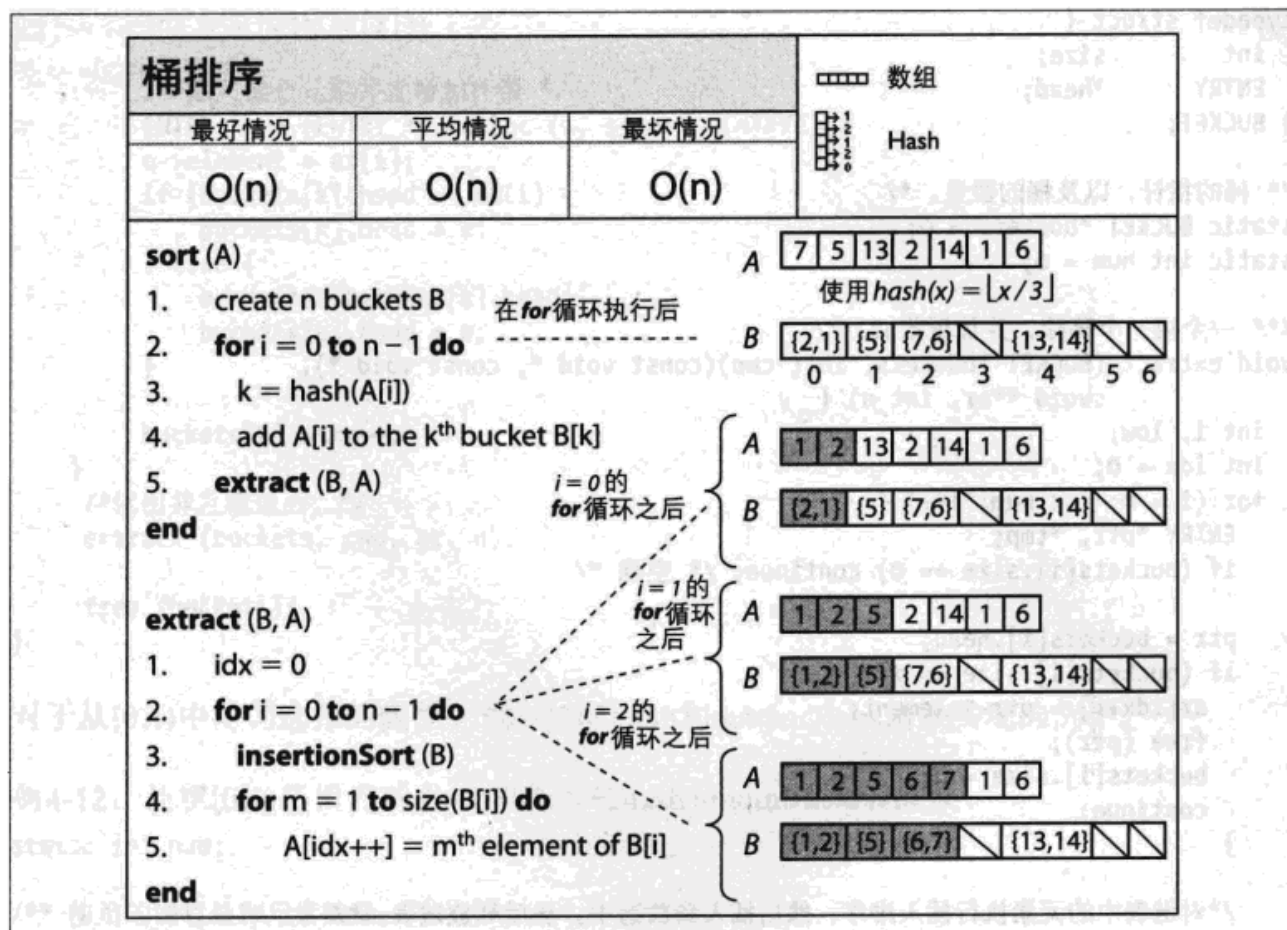


图4-18: 桶排序详细介绍

驱动因素

如果存储空间不重要，而且元素服从一个全序关系，桶排序就能够利用这些信息，节省大量开销。

解决方案

在桶排序的C语言实现中，如例4-11所示，每一个桶存储一个元素的链表，每个元素都是使用散列映射到这个桶中。适用于输入数据的函数numBuckets和hash是外部提供的。

例4-11: 桶排序的C语言实现

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

```
/* 桶中的元素链表。*/
typedef struct entry {
    void *element;
    struct entry *next;
} ENTRY;
```

```
/* 维护每个桶中元素的数目，并且指向第一个元素。*/
```



```

typedef struct {
    int      size;
    ENTRY    *head;
} BUCKET;

/* 桶的指针, 以及桶的数量。*/
static BUCKET *buckets = 0;
static int num = 0;

/** 一个接一个移除, 并且覆盖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; /* 空桶 */

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

        /*对链表中的元素执行插入排序, 然后插入到数组中。然后释放链表。*/
        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]);

    /**插入每个元素并且增加计数 */
    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++;
}
/*读出并且覆盖ar。*/
extract (buckets, cmp, ar, n);

free (buckets);
}

```

对于从[0,1)中均匀选择的数字，例4-12即hash和numBuckets函数实现的例子。

例4-12：处理[0,1)范围内的元素时使用的hash和numBuckets函数

```

static int num;

/** 使用的桶数量和元素数量一样。*/
int numBuckets(int numElements) {
    num = numElements;
    return numElements;
}

/**
 * 散列函数将元素映射到桶中。因为散列函数值的范围是[0, 1) ,
 * 所以我们将每个桶的大小分为1/num;
 */
int hash(double *d) {
    int bucket = num*(double)*d;
    return bucket;
}

```

桶也能够用固定大小的数组来存储，当桶都满之后，数组可以重新分配空间，但是链表的实现要快30%~40%。

分析

在例4-11的sortPointers函数中，输入中的每一个元素根据提供的散列函数，被插入到相应的桶中，花费是线性时间 $O(n)$ 。桶中的元素并不是有序的，但是由于仔细设计的散列函数，我们知道，如果 $i < j$ ，在桶 b_i 中的所有的元素都小于 b_j 中的所有元素。

随着值从桶中抽取出来并且写回到输入数组，当一个桶包含多个元素的时候就要使

用插入排序。对于表现出 $O(n)$ 性能的桶排序，我们需要保证排序每个桶所花费的总时间为 $O(n)$ 。我们定义 n_i 是分到桶 b_i 的元素数目。我们能够将 n_i 看做随机变（使用统计理论）。现在考虑 n_i 的期望值 $E[n_i]$ 。输入中的每一个元素插入到一个给定的桶中的概率为 $1/p$ ，因为每个元素的值是从 $[0,1)$ 之间均匀抽取的。因此 $E[n_i]=n*p=n*(1/n)=1$ ，方差 $Var[n_i]=n*p*(1-p)=(1-1/n)$ 。考虑方差是非常重要的，因此有些桶可能是空的，而其他的可能有多于一个元素；我们需要保证没有桶包含过多的元素。我们再一次通过统计理论得到下面的等式：

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

从这个等式我们能够计算出 n_i^2 的期望值。这个计算是非常关键的，因为这是决定插入排序开销的关键因素，插入排序的最坏情况性能为 $O(n^2)$ 。我们计算出 $E[n_i^2]=(1-1/n)+1=(2-1/n)$ ，可以看到 $E[n_i^2]$ 是一个常数。这个意味着当我们将 n 个桶上执行插入排序的总开销加起来之后，总的期望性能为 $O(n)$ 。

变种

在散列排序中，每一个桶代表的是散列函数返回的唯一值。散列排序并没有创建 n 个桶，而是创建了 k 个桶，随着 k 的增长，散列排序的速度不断加快。散列排序的关键在于散列函数 $hash(e)$ 返回的整数值，对于每个元素 e ，如果 $a_i \leq a_j$ ，那么 $hash(a_i) \leq hash(a_j)$ 。

例4-13定义了散列函数 $hash(e)$ ，这个函数计算仅仅包含了小写字母的元素。它将字符串的前三个字符转换成一个值（基准值为26），对于字符串“abcdefgh”，它的前三个字符（“abc”）被抽取出来然后转换得 $0*676+1*26+2=28$ 。这个字符串将插入到标为28的桶中。

例4-13：散列排序的 $hash$ 和 $numBuckets$ 函数

```
/** 使用的桶数量。*/
int numBuckets(int numElements) {
    return 26*26*26;
}

/**
 * 散列函数计算出每个元素应映射到的桶。
 */
int hash(void *elt) {

    return (((char*)elt)[0] - 'a')*676 +
           (((char*)elt)[1] - 'a')*26 +
           (((char*)elt)[2] - 'a');
}
```

散列排序的性能和桶的大小以及输入的规模有关，具体可参考表4-5。我们将一个使用了

三均值方法来选择pivotIndex的快速排序和散列排序进行比较，并且给出了一个可以比较的排序时间。

表4-5：基于不同数目的桶的散列排序以及快速排序的性能（单位：秒）

n	26个桶	676个桶	17576个桶	快速排序
16	0.000007	0.000026	0.000353	0.000006
32	0.00001	0.000037	0.000401	0.000007
64	0.000015	0.000031	0.000466	0.000016
128	0.000025	0.000042	0.000613	0.000031
256	0.000051	0.000062	0.00062	0.000045
512	0.000108	0.000093	0.000683	0.000098
1,024	0.000337	0.000176	0.0011	0.000282
2,048	0.0011	0.000456	0.0013	0.000637
4,096	0.0038	0.0012	0.0018	0.0017
8,192	0.0116	0.0027	0.0033	0.0037
16,384	0.048	0.0077	0.0069	0.009
32,768	0.2004	0.0224	0.0162	0.0207
65,536	0.8783	0.0682	0.0351	0.0525
131,072	2.5426	0.1136	0.0515	0.1151

注意在17 576个桶， $n>8192$ 个元素的情况下，散列排序的性能要优于快速排序（并且这个趋势随着 n 的增长继续保持）。但是，仅仅只有676个桶， $n>32768$ （平均每个桶48个元素）的话，散列排序随着数据集的增长，执行插入排序的开销也不断增长，速度越来越慢。事实上，在26个桶， $n>256$ 的情况下，随着 n 的加倍，散列排序所需的时间增加3倍，在桶很少时，散列排序的性能是 $O(n^2)$ 。

选择排序算法的标准

选择一个排序算法时，考虑表4-6的定性标准。这些标准可能能够帮你做出最初的决定，但是你需要更多量化标准作为指引。

表4-6：如何选择排序算法

标准	排序算法
很少的元素	插入排序
几乎有序的元素	插入排序

表4-6：如何选择排序算法（续）

标准	排序算法
关注最坏情况	堆排序
希望能够得到一个好的平均情况下性能	快速排序
元素是从一个密集集合中抽取	桶排序
希望尽可能少地写代码	插入排序

在为不同的数据选择合适的算法的时候，你需要知道输入数据的一些性质。我们创建了一些基准测试数据来比较本章所讲述的算法。注意表中的实际值是并不重要的，因为它们和基准测试运行的硬件相关。然而，你需要注意算法在这些数据集合的相对性能。

随机字符串

通过本章的论述，给定 $n!$ 个字符串，或者大约 4.03×10^{26} 个字符串，并且这些字符串很少重复，不仅如此，比较元素的开销不是常数。因为有时候会需要比较多个字符。通过计算这样的数据，我们可以知道明白了当排序26个字母的字符串时各个排序算法的性能。

双精度的浮点值

使用在大多数操作系统上可用的伪随机数生成器，我们得到了一系列范围在 $[0,1)$ 的随机数。在这个数据集合中基本上没有重复的元素，比较两个元素的开销是一个固定的常数。

给排序算法的输入数据可以被预处理，为了确保以下性质（并不是都相容的）。

有序

输入数据可是预先有序的，降序或者升序（最终目的）。

三值方法的反例

Musser (1997) 发现了一个排列的顺序，这个顺序能够确保快速排序在使用三值选择中枢值的时候需要 $O(n^2)$ 次比较。

几乎有序

给定一个有序的数据，我们选择相距 d 的 k 对元素进行交换（如果 d 为0则表示任意两个元素能够交换）。这样就能够构造一个和输入很相似的数据。

接下来的表中的列是按照算法在表最后一行的性能表现从左到右排好序。每一节有四个表，分别表明在本章早些描述的四种情况下的性能。

综合分析基准测试结果

因为插入排序和选择排序是本章中在随机均匀数据（提升一些数量级）最慢的两个算法，所以我们在表4-7~表4-11中忽略这两个算法。但是，这两个算法值得处理有序数据（表4-8）和几乎有序数据（表4-10和表4-11）。插入排序比其他算法表现要好，通常要快一个数量级。为了得到表4-7~表4-11的结果，我们在高端计算机上执行了100次实验，抛弃最好和最坏的结果，把剩下的98次结果的平均值填在表中。标记为Quicksort BFPRT⁴ minSize=4的列表示一个快速排序的实现，它使用了BFPRT（集合大小为4）来选择切分值，并且在子数组规模少于4个元素的时候切换到插入排序。

表4-7：在基于26个字母随机生成转置排序上的结果（单位：秒）

n	17576个桶的 Hash排序	三总值的 快速排序	堆排序	中值排序	使用minSize=4的 BFPRT ⁴ 的快速排序
4,096	0.0012	0.0011	0.0013	0.0023	0.0041
8,192	0.002	0.0024	0.0031	0.005	0.0096
16,384	0.0044	0.0056	0.0073	0.0112	0.022
32,768	0.0103	0.014	0.0218	0.0281	0.0556
65,536	0.0241	0.0342	0.0649	0.0708	0.1429
131,072	0.0534	0.0814	0.1748	0.1748	0.359

表4-8：在基于26个字母随机生成的有序转置上排序的结果（单位：秒）

n	17576个桶的 Hash排序	三总值的 快速排序	堆排序	中值排序	使用minSize=4的 BFPRT ⁴ 的快速排序
4,096	0.0011	0.0007	0.0012	0.002	0.0031
8,192	0.0019	0.0015	0.0027	0.0042	0.007
16,384	0.0037	0.0036	0.0062	0.0094	0.0161
32,768	0.0074	0.0082	0.0157	0.0216	0.0381
65,536	0.0161	0.0184	0.0369	0.049	0.0873
131,072	0.0348	0.0406	0.0809	0.1105	0.2001

表4-9：在对中值选择最不利的数据上进行排序的结果（单位：秒）

n	17576个桶 的Hash排序	堆排序	中值排序	使用minSize=4的 BFPRT ⁴ 的快速排序	三总值的 快速排序 ^注
4,096	0.0011	0.0012	0.0021	0.0039	0.0473
8,192	0.0019	0.0028	0.0045	0.0087	0.1993

表4-9：在对中值选择最不利的数据上进行排序的结果（单位：秒）（续）

n	17576个桶 的Hash排序	堆排序	中值排序	使用minSize=4的 BFPRT ⁴ 的快速排序	三中值的 快速排序 ^注
16,384	0.0038	0.0066	0.0101	0.0194	0.8542
32,768	0.0077	0.0179	0.024	0.0472	4.083
65,536	0.0171	0.0439	0.056	0.1127	17.1604
131,072	0.038	0.1004	0.1292	0.2646	77.4519

注：因为三中值快速排序速度退化得厉害，因此只进行了10次实验；表中所示的是将最好和最坏的情况去除以后，8次运行的平均值。

表4-10：在随机交换16对相距8个位置的元素的数据上排序的结果（单位：秒）

n	17576个桶的 Hash排序	三中值的 快速排序	堆排序	中值排序	使用minSize=4的 BFPRT ⁴ 的快速排序
4,096	0.0011	0.0007	0.0012	0.002	0.0031
8,192	0.0019	0.0015	0.0027	0.0042	0.007
16,384	0.0038	0.0035	0.0063	0.0094	0.0161
32,768	0.0072	0.0081	0.0155	0.0216	0.038
65,536	0.0151	0.0182	0.0364	0.0491	0.0871
131,072	0.0332	0.0402	0.08	0.1108	0.2015

表4-11：在随机交换n/4对相距4个位置的元素的数据上排序的结果（单位：秒）

n	17576个桶的 Hash排序	三中值的 快速排序	堆排序	中值排序	使用minSize=4的 BFPRT ⁴ 的快速排序
4,096	0.0011	0.0008	0.0012	0.002	0.0035
8,192	0.0019	0.0019	0.0028	0.0044	0.0078
16,384	0.0039	0.0044	0.0064	0.0096	0.0175
32,768	0.0073	0.01	0.0162	0.0221	0.0417
65,536	0.0151	0.024	0.0374	0.0505	0.0979
131,072	0.0333	0.0618	0.0816	0.1126	0.2257

双浮点数的基准测试结果

使用双浮点数的基准测试（表4-12～表4-16）消除了很多字符串比较时候产生的总开销。我们再一次在这些表中忽略了插入排序和选择排序。

表4-12：排序随机浮点数的算法性能（单位：秒）

n	桶排序	三 中 值 的 快 速 排 序	中 值 排 序	堆排序	使用minSize=4的 BFPRT ⁴ 的快速排序
4,096	0.0009	0.0009	0.0017	0.0012	0.0003
8,192	0.0017	0.002	0.0039	0.0029	0.0069
16,384	0.0041	0.0043	0.0084	0.0065	0.0157
32,768	0.0101	0.0106	0.0196	0.0173	0.039
65,536	0.0247	0.0268	0.0512	0.0527	0.1019
131,072	0.0543	0.0678	0.1354	0.1477	0.26623

表4-13：排序有序浮点数的算法性能（单位：秒）

n	桶排序	堆排序	中 值 排 序	三 中 值 的 快 速 排 序	使用minSize=4的 BFPRT ⁴ 的快速排序
4,096	0.0007	0.0011	0.0015	0.0012	0.0018
8,192	0.0015	0.0024	0.0032	0.0025	0.004
16,384	0.0035	0.0052	0.0067	0.0055	0.0089
32,768	0.0073	0.0127	0.015	0.0133	0.0208
65,536	0.0145	0.0299	0.0336	0.0306	0.0483
131,072	0.0291	0.065	0.0737	0.0823	0.1113

表4-14：在对中值选择最不利的数据上进行排序的结果（单位：秒）

n	桶排序	堆排序	中 值 排 序	三 中 值 的 快 速 排 序	使用minSize=4的 BFPRT ⁴ 的快速排序
4,096	0.0008	0.0011	0.0015	0.0015	0.0025
8,192	0.0016	0.0024	0.0034	0.0033	0.0056
16,384	0.0035	0.0053	0.0071	0.0076	0.0122
32,768	0.0079	0.0134	0.0164	0.0192	0.0286
65,536	0.0157	0.0356	0.0376	0.0527	0.0686
131,072	0.0315	0.0816	0.0854	0.1281	0.1599

表4-15：在随机交换16对相距8个位置的元素的数据上排序的结果（单位：秒）

n	桶排序	堆排序	中值排序	三 中 值 的 快速排序	使用minSize=4的 BFPRT ⁴ 的快速排序
4,096	0.0007	0.0011	0.0015	0.0012	0.0018
8,192	0.0015	0.0024	0.0032	0.0025	0.004
16,384	0.0035	0.0051	0.0067	0.0054	0.0089
32,768	0.0071	0.0127	0.0151	0.0133	0.0209
65,536	0.0142	0.0299	0.0336	0.0306	0.0482
131,072	0.0284	0.065	0.0744	0.0825	0.111

表4-16：在随机交换n/4对相距4个位置的元素的数据上排序的结果（单位：秒）

n	桶排序	堆排序	三 中 值 的 快速排序	中 值 排 序	使用minSize=4的 BFPRT ⁴ 的快速排序
4,096	0.0001	0.0014	0.0015	0.0019	0.005
8,192	0.0022	0.0035	0.0032	0.0052	0.012
16,384	0.0056	0.0083	0.0079	0.0099	0.0264
32,768	0.0118	0.0189	0.0189	0.0248	0.0593
65,536	0.0238	0.0476	0.045	0.0534	0.129
131,072	0.0464	0.1038	0.1065	0.1152	0.2754

参考文献

Bentley, Jon Louis and M. Douglas McIlroy, "Engineering a Sort Function," *Software-Practice and Experience*, 23(11): 1249–1265, 1993, <http://citeseer.ist.psu.edu/bentley93engineering.html>.

Blum, Manuel, Robert Floyd, Vaughan Pratt, Ronald Rivest, and Robert Tarjan, "Time bounds for selection." *Journal of Computer and System Sciences*, 7(4): 448–461, 1973.

Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to Algorithms*, Second Edition. McGraw-Hill, 2001.

Davis, Mark and Ken Whistler, "Unicode Collation Algorithm, Unicode Technical Standard #10," March 2008, <http://unicode.org/reports/tr10/>.

Gilreath, William, "Hash sort: A linear time complexity multiple-dimensional sort algorithm." Proceedings of First Southern Symposium on Computing, December 1998, <http://www.citebase.org/abstract?id=oai:arXiv.org:cs/0408040>.

Musser, David, "Introspective sorting and selection algorithms." *Software—Practice and Experience*, 27(8): 983 – 993, 1997.

Sedgewick, Robert, "Implementing Quicksort Programs." *Communications ACM*, 21: 847 – 857, 1978.

Trivedi, Kishor Shridharbhai, *Probability and Statistics with Reliability, Queueing, and Computer Science Applications*, Second Edition. Wiley-Interscience Publishing, 2001.





概述

我们可能会在一个给定的元素集合 C 上执行下述三个基本查询：

存在性查询： C 包含一个目标元素 t 吗？

很多时候我们只想知道一个给定集合 C 中是否包含一个特定的值 t 。如果这样的元素存在并且其值和 t 相等，那么这个查询的结果将是真，否则的话就是假。

检索查询：返回 C 中值和 t 匹配的元素。

当复杂的元素存储在该集合时，元素匹配的定义是基于键值或者元素的属性集合的子集匹配。例如，当在一个从机动车辆部门获得的信息集合中检索时，查询者需要提供驾驶员的驾照号码来获得许可驾驶员的全部信息。

关联查找：返回集合中和目标键值元素 k 关联的信息。

在复杂的结构中存储额外的信息是非常常见的。我们可以将额外的信息和集合中的元素 k 相关联，然后根据需要检索和操作。

对于本章讨论的算法，我们假设存在一个集合 U ，并且 U 包含了将要检索的值 e 。集合 C 是 U 的一个子集，目标元素 t 是 U 的一个元素。如果 t 是一个键值，那么 U 是一个可能的键值集合。

正如我们所见，知道 C 中的元素能否被随机存储或者能否被遍历是非常重要的。当集合能够为任意元素建立索引时，我们使用记号 A 来表示这种集合，用 $A[i]$ 表示集合中的第 i 个元素。为了方便起见，我们使用值 nil 来表示不在 U 中的元素，当查询要求返回集合中的特定元素，但是元素又不是当前指向的元素时，这个值是非常有用的。一般来说，我们假设不可能在集合中寻找 nil 。

本章的算法描述了特定的方法来结构化数据，以便于更加高效地处理这些请求。一种方法是考虑使用第4章介绍过的排序算法对集合 C 进行排序。我们将会看到，处理查询的效

率的确会得到改善，但是在维护有序集合时，会有一些其他开销，尤其是在可能会有对几何中的元素进行大量的删除和插入操作的时候。你首先必须选择合适的结构，并不仅仅是为了加速某一类查询的性能，同时也要在面对随机存储和大量重复的查询，维护集合结构时，降低整体的性能开销。

顺序查找

顺序查找也叫做线性查找，是最简单的查询算法。它通过穷举来寻找集合 C 中的单独目标元素 t 。它从集合中的第一个元素开始查找，然后检查每一个随后的元素，知道匹配的元素查找到或者集合中每一个元素都被查找过。

考虑这样一个案例，一个中等规模的饭店有10~20张桌子可以被顾客预定。这个饭店靠近一个大型的医疗中心，这个饭店的很多顾客都是中心的医疗人员或者病人家属。饭店宣传说他们能够在紧急时刻定位任何一位顾客，然后顺序地将信息传递给这个人。当顾客在指定的桌子旁坐下来，饭店老板将记录下这张桌子的顾客的名字，在他们离开时会擦除名字。为了能够在任何时候寻找到饭店中的顾客，饭店老板将简单地浏览记录中每张桌子的顾客姓名。

输入/输出

肯定有多种方法来从查找的集合中获取元素，顺序并不重要。如果一个集合 A 支持索引，那么能用索引值 i 来获得元素 $A[i]$ 。一个集合 C 通常只能通过一个只读的迭代器来获得其中的每一个元素（例如，一个和SQL查询语句相关数据库的游标）。使用游标存取这些元素的方法在图5-1的下半部描述。

输入

输入包括一个集合 C ，包含 $n \geq 0$ 个元素，以及需要寻找的元素 t 。

输出

如果 t 属于 C ，那么返回真，否则返回假。

使用环境

有时你需要在集合中定位元素。你也许不得不在一个未知是否有序的集合中频繁地查找元素。当无法知道集合中更多的信息时，顺序查找能够以穷举的方式完成这个工作。如果集合只能通过迭代器来存取，并且一次只能返回集合中的一个元素，那么顺序查找就是你能使用的唯一查找算法。

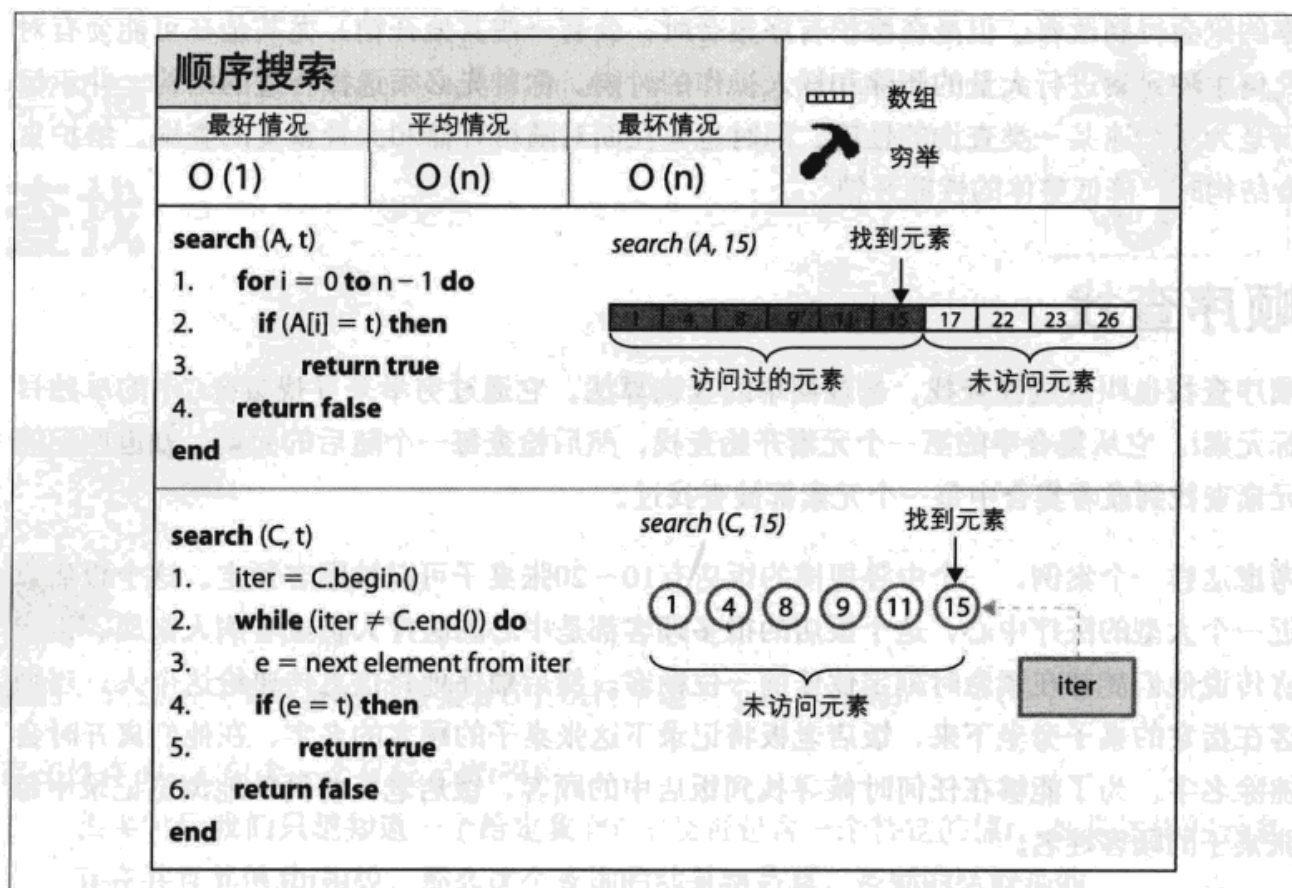


图5-1：顺序查找详解

驱动因素

从集合 C 中的元素个数 n 以及你将要执行的查找次数来看，顺序查找也许是最有效的查找算法。如果 n 相当低效，或者你不经常执行查找操作，那么对元素排序或者使用一个复杂的数据结构将得不偿失。

如果集合是无序的，并且是以链表的形式存储的，那么插入元素是一个常数时间的操作（仅仅简单地插入到表头或者表尾）。频繁地插入到一个数组形式存储的集合需要动态的数组管理，这个特性要么由编程语言提供，要么需要程序员自己处理。在这两种情况下，寻找一个元素的期望时间是 $O(n)$ ，因此，移除一个元素需要至少 $O(n)$ 时间。

顺序查找在元素的类型上限制最少。唯一的要求就是必须有一个匹配函数来决定是否目标元素和集合中的当前元素匹配，这个功能适合元素本身相关的。对于是否有序没有额外的要求。

解决方案

顺序查找的实现是最简单的。如果集合存储形式是数组，你仅仅需要从第一个元素开始

进行比较，如果匹配，返回真，否则的话移动到下一个元素并且继续比较，直到你找到想要的元素。如果你到了数组的末尾还没有寻找到，那么返回假。

例5-1是顺序查找数组的Ruby代码。

例5-1: Ruby的顺序查找

```
def sequentialSearch(collection, t)
  collection.each {
    |i| if i == t then return true; end
  }
  return false
end
```

代码是如此的简单。这个函数的输入是一个集合和需要寻找的目标元素。集合可以是数组或者是Ruby中任何一种能够支持each方法的集合。查找中的元素必须支持==操作符，否则的话，你需要使用Ruby支持的其他类型的相等操作符。例5-2是相同例子的Java实现。SequentialSearch类有一个类型参数，T表示集合中的元素类型，T必须提供一个有效的equals(Object o) 方法。

例5-2: 顺序查找的Java实现

```
package algs.model.search;
import java.util.Iterator;
public class SequentialSearch<T> {

    /** 使用穷举的顺序查找在已索引的集合（类型为T）中寻找给定的元素 */
    public boolean sequentialSearch (T[] collection, T t) {
        for (T item : collection) {
            if (item.equals(t)) {
                return true;
            }
        }
        return false;
    }

    /** 使用穷举的顺序查找在已索引的集合（类型为T）中寻找给定的元素。*/
    public boolean sequentialSearch (Iterable<T> collection, T t) {
        Iterator<T> iter = collection.iterator();
        while (iter.hasNext()) {
            if (iter.next().equals(t)) {
                return true;
            }
        }
        return false;
    }
}
```

例5-3是顺序查找的C语言实现，集合是存储在数组ar中，并且两个元素如果匹配，比较函数返回0。

例5-3: 顺序查找的C语言实现

```
int search (void *t, int(*cmp)(const void *,const void *)) {
    int i;
    for (i = 0; i < n; i++) {
        if (!cmp(ar[i], t)) {
            return 1;
        }
    }
    return 0;
}
```

结论

对于无序的小集合来说，顺序查找是最容易实现的并且总体上很有效率。最坏情况是你寻找的元素不在集合中，因为你需要检查集合中的每一个元素。如果查找的结果普遍都是假，那么你需要考虑一下使用一个不同的查找算法，即使集合相对来说比较小。

有些时候，一个已索引的集合也许会有一些“空洞”。也就是说，集合中有一些索引指向的是空元素（注1）。在这样的情况下，你需要在实现中添加代码来检查索引值，确保其指向的元素不为空。我们将例5-1的代码加上了额外的校验，如例5-4所示。

例5-4: 增加了校验空元素的顺序查找

```
def sequentialSearch(collection, t)
  collection.each {
    |i| if (i != nil) && (i == t) then return true; end
  }
  return false
end
```

如果时间要求很高并且集合很大，那么元素之间的额外比较就需要引起注意，所以这里存在另外一个解法。不在空槽中放入nil，而是放入一个特定的标记元素。当检查这个标记是否和其他元素相等时候总是返回假。例如，如果已知寻找的*t*是一个正整数，那么你可以使用-1作为标记来避免额外的比较（注2）。

分析

如果寻找的元素属于集合并且在任意索引位置的概率是相等的（换句话说，它被迭代器在任意位置找到的几率是相等的），那么这就是平均情况下顺序查找（如我们在第2章所述）。

注1: 更确切地说，数组的第*i*个元素A[i]可能等于一个特定值nil。对于数组来说通过迭代器访问来给出nil值的用法很少见。

注2: 在Ruby中，nil可以和任何对象进行值比较。这意味着在示例5-4中额外的比较实际上是不需要的。

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

也就是说，对于每个目标元素来说，你将需要寻找集合中大约一半的元素。最好情况即待查找的元素是集合中的第一个。算法在最坏和平均情况下都是线性增长的。如果你加倍集合的大小，那么查找的时间也大约会是两倍。

为了得到顺序查找的实际性能，我们构造了一个包含 n 个整数（范围为 $[1, n]$ ）的有序集合。虽然这个集合是有序的，但是查找时并没有用到这个信息。我们将会进行100次试验，每一次试验我们执行1000次查找随机值 t 。我们抛弃最好和最坏的结果，然后对剩余的结果取平均值。表5-1是四个特定的 p 值，剩余98次的平均时间。注意当集合的大小加倍时，执行时间也大约加倍。你也需要看到，对每一种输入集合的大小来说，最后一列表示最坏情况的时间。

表5-1：顺序查找的性能（单位：秒）

n	$p=1.0$	$p=0.5$	$p=0.25$	$p=0.0$
4,096	0.0081	0.0106	0.0111	0.0138
8,192	0.016	0.0223	0.0236	0.0297
16,384	0.0324	0.0486	0.0524	0.0629
32,768	0.0624	0.0958	0.1141	0.1263
65,536	0.1304	0.2065	0.226	0.226
131,072	0.2627	0.3625	0.3779	0.4387

在最好情况下，集合的第一个元素就是需要的元素，这样仅仅需要花费 $O(1)$ 常数时间。最坏的情况是寻找的元素不在集合中，在这种情况下，你需要检查集合中的所有元素，导致了 $O(n)$ 的性能，平均情况下（如第2章所述，并且如表5-1所示），性能是 $O(n)$ 。

变种

当目标元素并不是均匀分布时，有时就有一些变种可以来改善顺序查找的性能，而不是给程序员添加额外的负担。我们本能地希望待查找的元素能够尽可能地靠近数组的最前面。如果你知道一些目标元素的信息（尤其是你查找的顺序），这里有一些策略能够改善顺序查找的性能。这些策略是非常有用的，但是仅仅对于能够按照如下的查找过程来修改的索引数组：

成功查找到则移动到最前面

这个策略适用于一个查找的元素 t 有更大的可能被再次查找到。当 t 在位置 i 找到时，

将 $A[0, i-1]$ 的所有元素移动到 $A[1, i]$, 然后把 t 移动到 $A[0]$ 。这个就是最多最近使用 (MRU) 分页算法的基础。

成功查找到则移动前一位

这个策略适用于如果一个查找的元素 t 有更大的可能能够被再次查找到; 并且, 这里可以避免移动大量元素所带来的开销。当 t 在位置 i 找到时, 交换 $A[i-1]$ 和 $A[i]$ (除非当 i 在集合的最前面)。直观地看, 如果元素非常频繁地被找到, 它们就会最终冒泡到集合的最前面, 并且总开销非常小 (仅仅是一个元素的交换)。

成功查找到则移动到最后

如果一个元素不会被多次查找, 那么当它被查找到一次之后移动到集合的末尾将会改善未来查找的性能, 将 $A[i+1, n)$ 移动到 $A[i, n-1)$, 然后将找到的元素移动到 $A[n-1]$ 。

这些变种的性能都各不相同, 但是都是基于某个查找元素的概率。根据你对数据集合的深刻理解和目标元素, 选择这些策略中的一个。分析这样的算法是众所周知的难, 需要的高级数学技巧已经超出了本书范围。

二分查找

二分查找 (图5-2) 在预先排好序的集合上表现出了比顺序查找更好的性能。二分查找每次将有序集合折半直到找到待查找元素, 或者发现待查找元素不在当前的集合中。

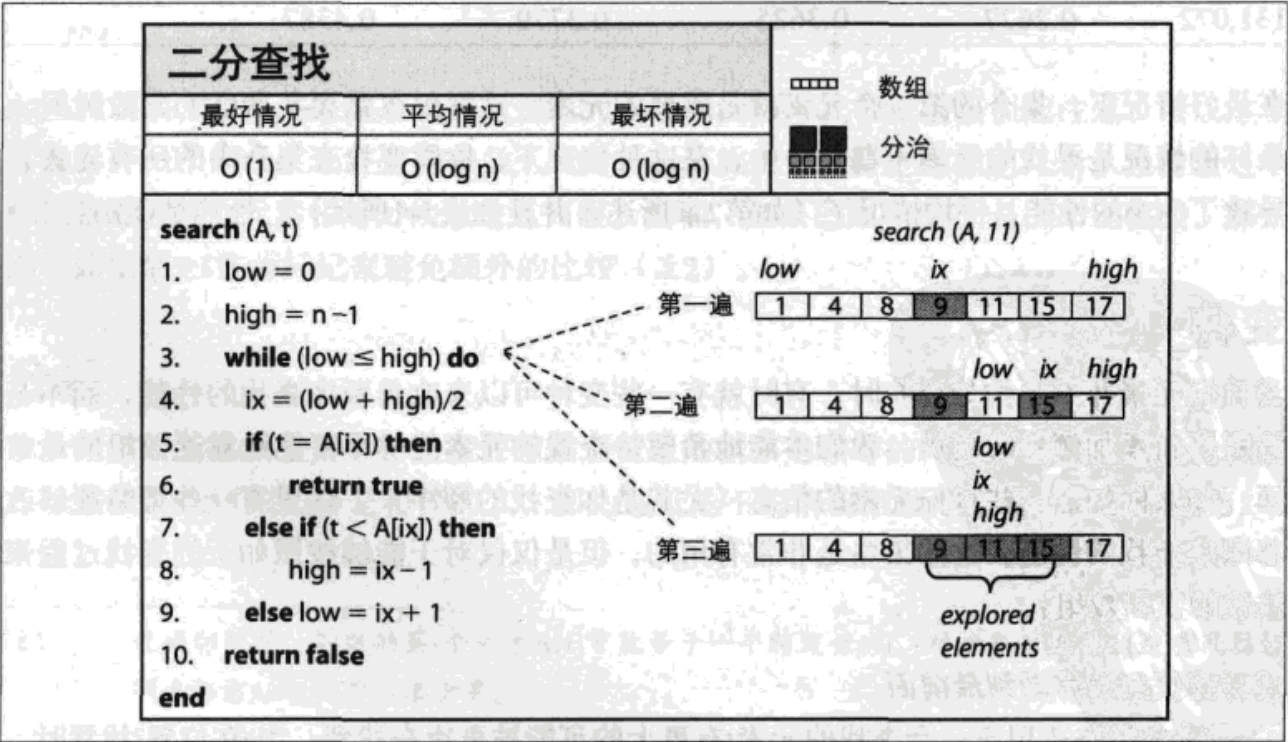


图5-2: 二分查找详解

假设你有一本纽约电话簿，并且需要找到“David Mamet”的电话。如果你使用顺序查找的话，你也许需要花费下午的大部分时间来寻找他的电话号码，或者更坏的是，在知道他的号码是否已经列出之前，你不得不检查电话簿中的每一个条目。很明显的，这个方法没有利用电话簿上名字有序这个特性。所以，你可以翻到电话簿的中页看看“David Mamet”是否在那页上。如果“David Mamet”这个条目在那一页，那么就到此为止。如果不在，并且“Mamet”在字典序上比这一页上任何一个姓都要靠前，那么你需要在电话簿的前半部去寻找。否则的话，你就需要在电话簿的后半部寻找。这个过程是一种“分治”的策略，能够快速定位想要找的目标。注意，如果你翻到某一页，这一页上“Mamet”应该出现但是没有出现，那么你知道不用再去查找任何一页，因为Mamet的电话不在这本电话簿上。

输入/输出

二分查找的输入是一个索引集合A。每一个元素 $A[i]$ 有一个键值 k_i 能够用来区分元素。这些键值是有序的，意思是，给定两个键值 k_i 和 k_j ，要么 $k_i < k_j$ ，要么 $k_i = k_j$ ，或者 $k_i > k_j$ 。我们构造了一个数据结构来保存这些元素（或者这些元素的指针）和维护键值的有序。我们也必须能够将这个数据结构分成数个子集进行查找，这样我们就使用了“分治法”来解决这个问题。二分查找的输出是真或者假。

使用环境

无论查找一个数字集合或者是一个字典序的名字列表，这个方法都能够很好地处理。我们能够看到在最坏时间下需要对数次数的检查。

驱动因素

元素的键值必须是全序的，使得你能够知道一个元素是“大于或者等于”另外一个元素。二分查找能够支持不同种类的数据结构。如果集合是静态的，那么元素能够被放入一个数组中。这样能够更方便快捷地遍历集合。但是，如果你需要从集合中添加或者移除元素，这个方法就变得非常笨拙。你可以使用很多数据结构，最著名的一个是二叉树，将在“变种”中讨论。

解决方案

给定数组形式的一个有序集合，例5-5的Java代码是操作任何基类T（使用Java的泛型）的二分查找的实现。Java提供了`java.util.Comparable`接口，这个接口包含一个方法，`compareTo`。任何正确实现了这个接口的类能够保证其实例的全序。

例5-5：二分查找的Java实现

```
package algs.model.search;

/**
 * 二叉查找处理一个参数类型的有序数组
 *
 * @param 参数T必须实现了Comparable接口
 */
public class BinarySearch<T extends Comparable<T>> {

    /* 在集合中寻找目标，成功返回真 */
    public boolean search(T[] collection, T target) {
        // null不存在集合中。
        if (target == null) { return false; }

        int low = 0, high = collection.length - 1;
        while (low <= high) {
            int ix = (low + high)/2;
            int rc = target.compareTo(collection[ix]);

            if (rc < 0) {
                //目标小于collection[i]
                high = ix - 1;
            } else if (rc > 0) {
                // 目标大于collection[i]
                low = ix + 1;
            } else {
                //找到了
                return true;
            }
        }

        return false;
    }
}
```

在这个实现中使用了三个变量：low、high和ix。low是当前查找子数组的最低索引，high是最高索引，ix是子数组的中点。这段代码的性能取决于循环执行的次数。

结论

二分查找实现稍稍有点复杂但是却获得了巨大的性能提升。当集合并不是存储在简单的内存中数据结构（例如数组）中时会增加实现的复杂性。基于元素的自然序，必须有能够集合中直接在存取元素 $A[i]$ ($0 \leq i < n$) 的方式，并且在Comparable接口中实现。较大的集合也许需要存储在二级存储器中，例如在磁盘上以文件的形式。在这种情况下，第 i 个元素能够根据其在文件中的偏移量来存取。如果使用二级存储器，查找一个元素所需要的时间就主要是存取二级存储器所需要的开销，其他和二分查找相关的解决方案也适用。参考算法的“变种”一节，其中对这些问题进行了处理。

分析

二分查找每次执行循环时都会大约将问题折半进行处理。将大小为 n 的集合折半的最大次数为 $\log(n)$ ，如果 n 是2的幂，否则的话，就是 $\lfloor \log(n) \rfloor$ 。如果我们仅仅使用一个操作来决定两个元素是相等，小于或者大于（这个决定也许是有Comparable接口来做出的），那么仅仅需要 $\lfloor \log(n) \rfloor$ 次比较操作，这个算法的性能是 $O(\log n)$ 的。

我们执行了100次实验，每次实验在集合中执行524 288次查找，这个集合是存储在内存中的，大小为 n （ n 的范围是从4096~524 288），目标存在于这个集合中的概率是 p （在1.0、0.5和0.0处采样）。表5-2列出的是在抛弃最好和最坏的实验结果后，剩余98次实验的平均结果。

表5-2：在内存中执行524 288次查找，二分查找和顺序查找的比较（单位：秒）

n	顺序查找			二分查找		
	p = 1.0	p = 0.5	p = 0.0	p = 1.0	p = 0.5	p = 0.0
4,096	3.8643	5.5672	7.2143	0.0809	0.0773	0.0704
8,192	7.2842	10.7343	14.1308	0.0861	0.0842	0.0755
16,384	14.0036	20.9443	27.7101	0.0928	0.0902	0.0814
32,768	27.8042	40.7164	54.3972	0.0977	0.1065	0.1067
65,536	54.8484	81.1192	107.8211	0.1206	0.1155	0.1015
131,072	107.6957	161.6651	215.1825	0.1246	0.1251	0.1127
262,144	*	*	*	0.1373	0.1346	0.1232
524,288	*	*	*	0.1479	0.1475	0.133

设计这些实验的目的是确保在 $p=1.0$ 时，集合中的所有元素都能够等概率地被查找到，如果不是这样的话，那么这个结果将是不可靠的。对于顺序查找和二分查找来说，输入时一个有序的整数数组，整数的范围在 $[0, n)$ 。为了产生524 288个目标元素，并且这些元素都在这个集合中（ $p=1$ ），我们循环地产生 n 个元素524 288/ n 次。

表5-3列出了在本地磁盘上执行524 288次查找的时间。目标元素要么总是存在于集合中（例如 $p=1.0$ ）或者从不存在（例如，我们在集合 $[0, n)$ 中查找-1）。数据是一个简单的文件，文件存储的是按升序排列的整数，每个整数是4个字节。磁盘存储的优劣势是非常明显的。因为表5-3的结果相比表5-2的结果要慢近200倍。当 n 加倍时，你将会注意到查找只是增加了固定的时间，这种特性非常明显地表明了二分查找是 $O(\log n)$ 算法。

表5-3：二级存储器上执行524 288次查找的二分查找的性能（单位：秒）

n	p = 1.0	p = 0.0
4,096	2.8403	2.8025
8,192	3.4829	3.4942
16,384	4.1842	4.0046
32,768	4.9028	4.7532
65,536	5.5598	5.449
131,072	6.4186	6.1808
262,144	7.1226	6.9484
524,288	7.8316	7.4513

变种

如果你希望支持“查找或者插入”操作，那么图5-2中第10行的ix最终值表示了缺失元素需要插入的索引值（所有较高的索引值都需要增加）。

二分查找有两个主要的变种。第一个和处理动态数据有关，在允许高效地插入或者删除时也需要维护一个可以接受的查找性能。如果集合是以数组的形式存储的话，因为每一个数组的条目都是一个有效的元素，所以插入和删除将会比较低效。因此，插入操作将会扩展这个数组（从逻辑上或者物理上）并且平均需要移动一半的元素。删除需要缩短数据并且也需要移动一半的元素。任何一个都是不可接受的。

如果集合能够存储在内存中，那么一个好的方法是使用基于散列，并且使用冲突链的查找。见下一节“基于散列的查找”，这个查找描述了查找动态数据的方法。一个替代的方法是在内存中构造一个二叉查找树。如果插入和删除操作是随机的，那么这个替代的方法实现起来非常简单，并且树不会偏置。但是，经验告诉我们这种情况很少发生，所以需要使用查找树的一个更复杂类型——平衡二叉树（Cormen等，2001）。

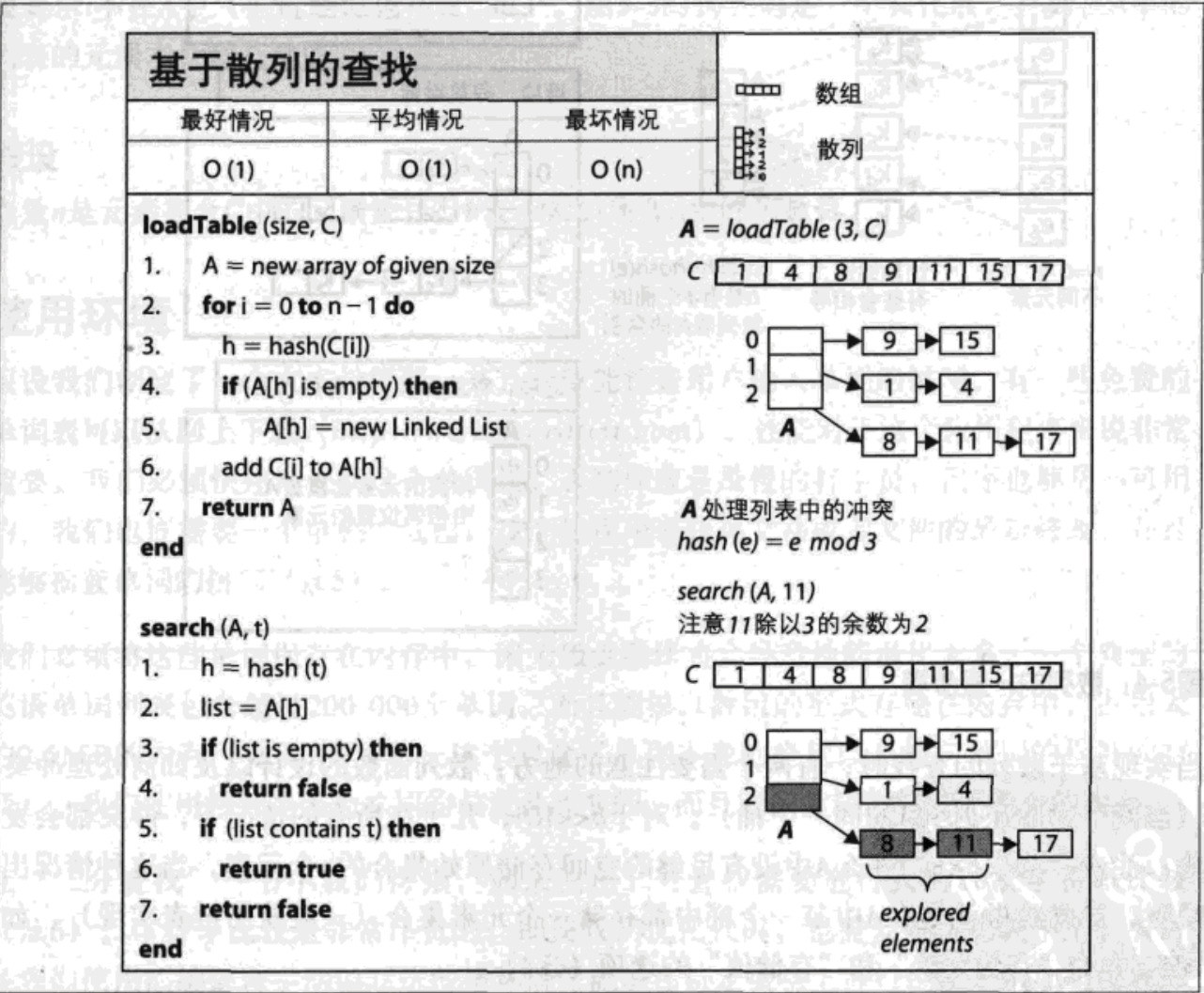
第二种变种处理的数据是动态的，并且由于过大不能存放在内存中。当这种情况发生时，查找时间就取决于二级存储器的输入输出操作所花费的时间。一个更有效的解决方法是使用叫做B树的n元树。这是一个多级树，并且在二级存储器上有较好的性能表现。在<http://www.bluerwhite.org/btree>上你能找到B树的教程，还包含一些例子。

基于散列的查找

前面讨论的查找算法在处理小数据量（顺序查找）或者有序的数据集合（二分查找）时才使用。我们需要更加强大的算法能够查找较大的集合，而且并不需要有序。最常使用

的一个方法是使用散列函数来将目标元素的一个或者多个特征转换成一个值，这个值用来索引一个已经索引的散列表。基于散列的查找有着比本章描述的其他算法在平均情况下更好的性能。很多算法的书籍都是在讨论散列表时才介绍基于散列的查找，你也能够在数据结构的书籍中讨论散列表的章节中找到这个算法。

在一个基于散列的查找（图5-3），集合 C 的 n 个元素首先会加载到一个有着 b 个桶的散列表 A 中。键值的概念使得这个操作可行。对于每个元素 $e \in C$ 来说，它能够通过 $k=key(e)$ 映射到一个值 k ，如果 $e_i=e_j$ ，那么 $key(e_i)=key(e_j)$ （注3）。一个散列函数 $h=hash(e)$ 使用了键值 $key(e)$ 来将 e 插入到桶 $A[h]$ 中。一旦散列表 A 被构造好了，那么查找一个元素 t 也被转换成在桶 $A[h]$ 中寻找一个元素 t ， $h=hash(t)$ 。



- 可能键值的全集 U 。每一个元素 $e \in C$ 将映射到一个键值 $k \in U$ 。
- 散列表 A 存储着原始集合 C 的 n 个元素。 A 也许包含着元素本身或者也包含了元素的键值。在 A 中有 b 个位置。
- 散列函数 hash ，使用 $\text{key}(e)$ 计算一个整数索引 h 。

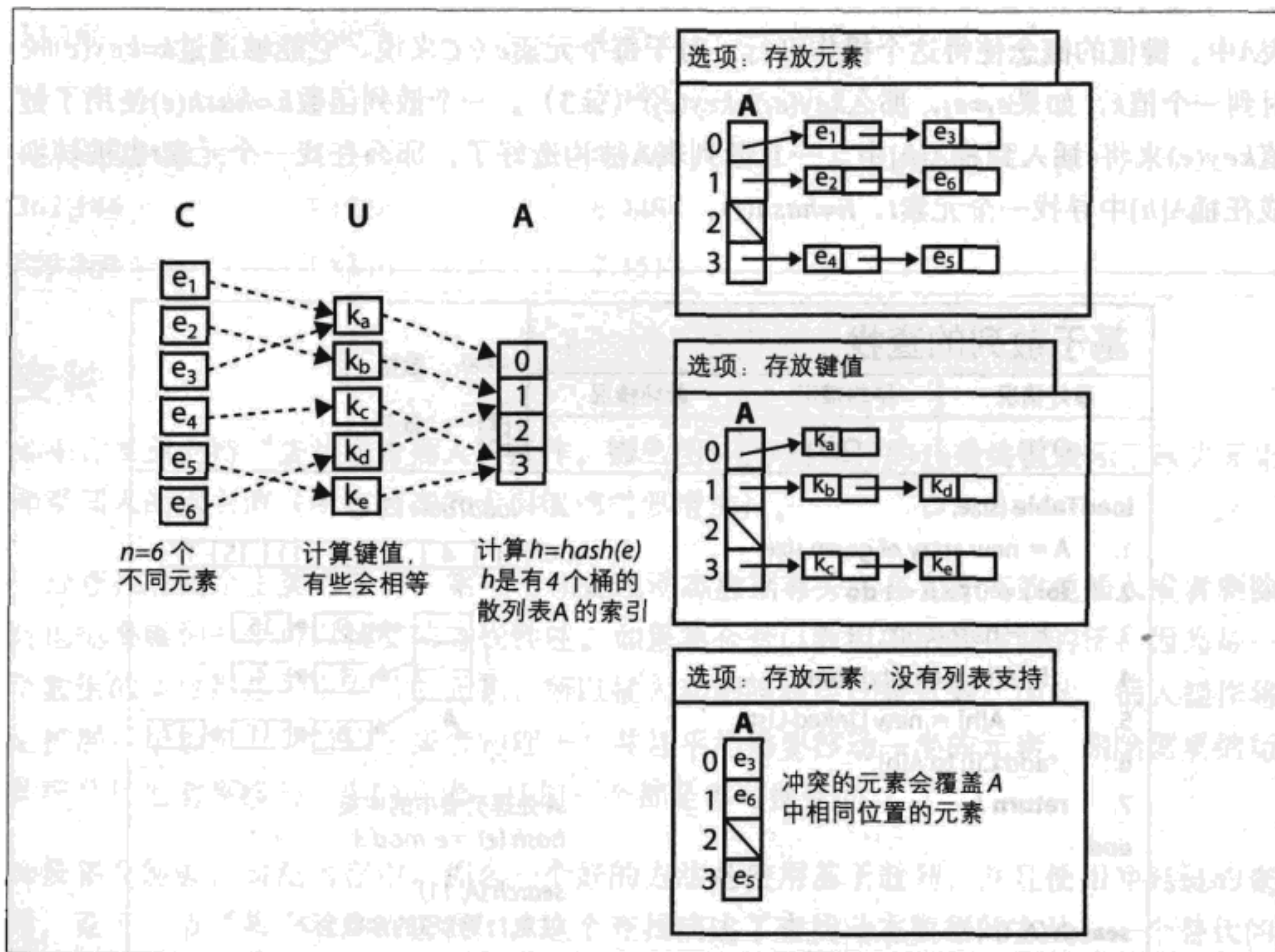


图5-4：散列的一般步骤

当实现基于散列的查找时，有两个需要注意的地方：散列函数的设计以及如何处理冲突（当两个键值映射到 A 的同一个桶）。对于 $b \ll |U|$ ，几乎在所有的情况下，冲突都会发生。注意，如果 $b < n$ ，那么 A 中没有足够的空间存储原始集合的 n 个元素。当这种情况出现时，通常的做法是在 A 中每一个桶中都存储一个元素集合（一般使用链表实现），如图5-4中的“存储元素”和“存储值”的选项（注4）。

不适当的散列函数可能会导致一个不均匀的键值分布。这样会导致两个结果：散列表中的很多桶会是没有使用过的，浪费了空间，而使用的桶又会因为多个键值同时映射到了一个桶中，导致了大量的冲突，这样使得性能更加糟糕。

注4：此外，如果元素直接存于 A 中（如图5-4的“存放元素，没有列表支持”），那么你需要处理冲突，否则之前存放在散列表中的元素可能丢失。

对于大多数输入来说，冲突都是可能存在的。期望的冲突数量增长时，处理冲突的策略对于算法的性能有着非常重要的作用。

输入/输出

寻找的目标元素 t 必须有一个或者多个特征能够用来作为键值，这些键值组成了全集 U 。不像二分查找，原始集合 C 不需要是有序的。事实上，即使 C 中的元素在某种程度上是有序的，散列方法在插入元素到散列表 A 时也不会尝试去保持其有序性。

基于散列的查找的输入是散列表 A ，以及寻找的目标元素 t 。如果 t 在 $A[h]$ 的指向的链表中，算法返回真，这里 $h=\text{hash}(t)$ 。如果 $A[h]$ 是空或者不在 $A[h]$ 指向的链表中，那么返回假表示 t 不在 A 中（也就是说也不在 C 中）。图5-3的伪代码是一个简化版，存储在 A 中的列表的元素本身就是键值。

假设

变量 n 是元素集合 C 中的元素数目， b 表示索引集合 A 中的桶数目。

使用环境

假设我们创建了一个文本编辑器，并且希望能检查用户输入单词的拼写。有一些免费的单词表可以从网上下载（<http://www.wordlist.com>）。性能对于这个应用程序来说非常重要。我们必须快速地查找这个单词表，否则即使是最慢的打字员，程序也都是不可用的。我们也许需要一个单独的线程，这个线程用来获得文档或者文件的最新修改，并且能够检查单词的拼写（注5）。

我们必须将这些单词保存在内存中，因为磁盘操作将会导致性能退化太多。一个典型的英语单词列表包含超过200 000个单词，而且能够以数组的形式存储在内存中，占用大约2.5MB的内存（如图5-4所示，这个包含了单词本身的空间以及指向单词的指针的空间）。我们使用指针是因为单词的长度是改变的，而且我们希望能够使用最少的内存。

在“二分查找”一节中我们得知，如果使用二分查找需要进行大约18次字符串比较（注6）。字符串比较是非常昂贵的，即使我们优化代码，也需要循环比较字节。有时候我们使用汇编语言手动编写这些循环确保能够在特定架构上进行优化，例如确保我们

注5： 在搜索之前基于散列的查找需要完整的单词，而使用二分查找，我们并不需要一次性输入所有的字母，但是这样程序变得更加复杂了。

注6： $\log(200\,000)=17.61$

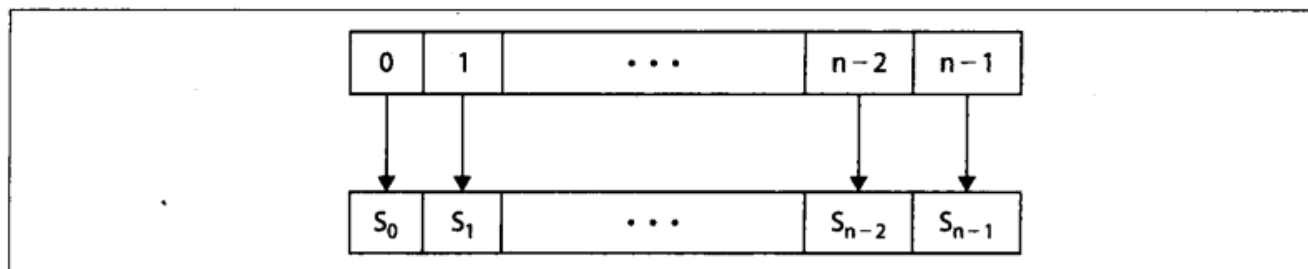


图5-5：存放基于散列的查找所使用的字符串

不会在大部分的情况下停顿在流水线上以及在可能时展开循环来填充流水线。目标是最小化字符串比较的次数。

我们首先需要定义一个函数来计算字符串 s 的键值。这个键值函数的一个目标是产生足够多的不同值，不过并不要求这些值都是唯一的。根据初始字符串的每一块信息来产生值的一个流行的方法是：

$$\text{key}(s) = s[0] * 31^{(\text{len}-1)} + s[1] * 31^{(\text{len}-2)} + \dots + s[\text{len}-1]$$

$s[i]$ 是第 i 个字符（值在0~255之间）， len 就是字符串 s 的长度。计算这个函数的值非常简单，例5-6就是这个函数的实现（根据Open JDK源代码改写）， chars 是字符数组，即一个字符串（注7）。根据我们的定义， java.lang.String 的 $\text{hashCode}()$ 方法是 $\text{key}()$ 函数。

例5-6：Java hashCode 样例

```
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;
}
```

因为 hashCode 方法尽量尝试优化，它缓存了已经计算过的散列值，避免重复计算（例如，当且仅当散列是0时的计算值）。

下一步，我们构造散列表。我们有 n 个字符串，那么散列表 A 的大小应该是多少呢？在最理想的情况下， A 有 $b=n$ 个桶，散列函数是一个从字符串集合到整数 $[0, n)$ 的一一映射函数。这是在正常情况下不可能发生的，所以我们尝试着构造一个空桶尽可能少的散列表。如果我们的散列函数平均分配键值，我们就能够获得一个比较理想的情况，数组的大小和集合的大小差不多。我们定义 $\text{hash}(s) = \text{key}(s) \% b$ ， $\%$ 是取模运算符，返回 $\text{key}(s)$ 除以 b 的余数。

注7： 代码可以从Open JDK的网站上下到，网址是<http://openjdk.java.net>。

高水平读者此时就会问基本的散列函数和散列表在这里会起什么作用。因为单词表是静态的，所以我们能够通过构建一个完美的散列函数来做得更好。一个完美的散列函数能够保证在一个特定的键值集合中没有冲突产生。在这种情况下，一个完美的散列函数就能投入使用，在接下来的“变种”一节中将讨论这个问题。让我们先在没有这个完美散列函数的情况下尝试解决这个问题。

在我们第一次尝试解决这个问题时，我们选择了一个数组A，有着217个桶，以及262 143个元素。我们的单词表包含213 557个单词。如果我们的散列函数能够使字符串完美地分布，那么就会没有冲突产生并且需要大约40 000个槽。但是情况并非如此。表5-4给出了我们单词表中字符串的散列值（注8）分布情况，散列表有着262 143个槽。正如你所见，没有一个槽会存储超过7个字符串，对于非空槽来说，每个槽的平均字符串数量大约是1.46。表5-4的每一行表明了使用的槽的数量以及有多少字符串被映射到这些槽中。散列表中几乎一半的槽（116 186个）没有字符串映射到。所以，这个散列函数浪费了大概500K的内存空间——假设每个指针的大小是4个字节，我们不会使用在空条目上使用冲突处理策略。你也许会非常惊讶这是一个非常优秀的散列函数，如果需要寻找到一个比这个更好的，那么需要一个更加复杂的结构。对于这个数据集来说，只有五对字符串有着相同的键值（例如，hypoplankton和unheavenly有着相同的键值427 589 249）！

表5-4：使用Java的String.hashCode()方法作为键值，b=262 143时，散列分布情况

命中次数	槽个数
0	116,186
1	94,319
2	38,637
3	10,517
4	2,066
5	362
6	53
7	3

最后，我们需要决定采取什么策略来处理冲突。一个可行的办法是在主散列表存储指向一系列条目的指针，而不是存储一个对象。这个方法，如图5-6所示，叫做链式。

注8：在这章中，每个Java类的hashCode方法使用的是之前描述过的key函数，回忆一下 $hash(s)=key(s)\%b$ 。

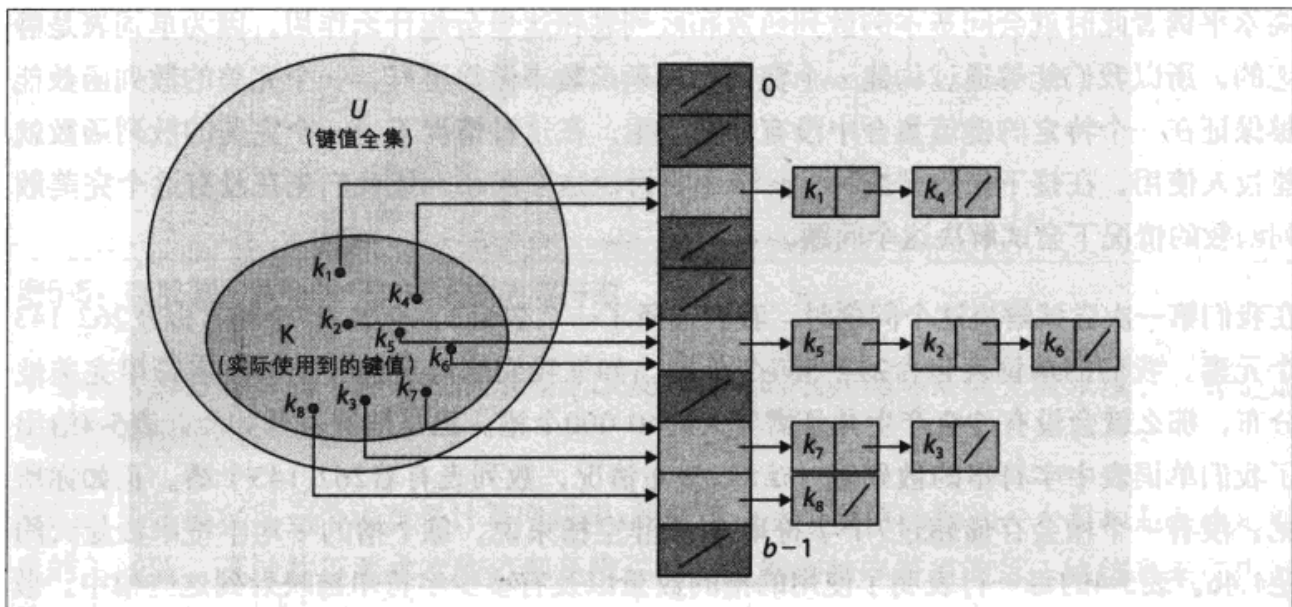


图5-6：使用表结构来处理冲突

这个方法的总开销取决于你在每个槽中是元素列表还是nil值（表示没有列表）。当一个槽只有一个元素时，它可以使用列表来存取。作为我们的第一个近似的解决方案，我们将会使用这种方法并且在性能成为瓶颈时改进它。

驱动因素

选择散列函数是在实现基于散列的查找之前必须做的第一个决定。散列已经被研究多年，并且有大量的描述高效散列函数的论文，不过用在查找上使用这些函数简直是杀鸡用牛刀。例如，某些特定的散列函数对于加密非常重要。对于查找来说，一个散列函数必须要有一个好的分布，并且计算速度非常快。

存储器空间是设计基于散列查找时需要考虑的另外一个因素。主存储器A必须能够足够大，以存下所有的查找键值，并且要给冲突键值留下足够的空间。A的大小通常是一个素数。但是，如果我们不使用开放定址的话（见接下来的“变种”一节），我们能够使用任何数字作为A的大小。实践中一个好的选择是 $2^k - 1$ ，即使这个值并不是素数。存储在散列表的元素对内存有直接的影响。考虑图5-3的是如何在链表中存储每个字符串的，所以存储在A中的元素看做链表元素。堆中包含着指向元素的指针。每一个链表都有存储的开销，包含着指向链表中的第一个和最后一个元素。如果你使用Java的LinkedList类，一些很重要的附加字段和类使得实现非常灵活。程序员可以写一个简单得多的链表类，只提供必需的功能，但是确实给基于散列查找算法带来了额外的开销。

如果你使用LinkedList类，假设指针是四个字节，那么A中的每一个非空元素都需要12字节内存。每个字符串元素不可以直接转换成ListElement，需要12个额外字节。对于之前的那个有213 557个单词的例子，我们需要5 005 488字节的额外存储。这种情况是：

- 主表的规模：1 048 572字节。
- 包含116 186个元素的规模：1 394 232字节。
- 包含213 557元素的规模：2 562 684字节。

如果你使用Java的String类，那么存储字符串还会需要额外的开销。每个字符串有12个字节的额外开销。因此我们就将增加 $213\,557 \times 12 = 2\,562\,684$ 个额外字节，所以，例子中选择的算法需要7 568 172字节的内存来正常运行。单词表中字符串的实际字符数量只有2 099 075。我们的算法还需要大约是字符的存储空间4.6倍的额外存储空间。

我们之后将会讨论一些优化内存使用的变种，当内存价格高高在上时，你可以使用这些变种来节省开支。但是，如果你有足够的内存，还有一个合理的并不会产生太多的冲突的散列函数以及一个可以立即使用的链表实现，那么你为什么选择不选择JDK提供的解决方案呢？

只要散列函数能够将元素均匀分布，那么基于散列的查找将会得到非常好的性能。查找一个元素的平均时间将是常数，也就是 $O(1)$ 。

有其他的影响实现的因素。主要都是关于如何处理集合的静态或者动态特性。在我们的例子中，我们知道我们的单词表有多大，并且我们不会在单词表中添加或者删除单词，至少不会在程序执行时操作。但是，如果我们有一个动态的集合，会做很多插入或者删除元素的操作，我们必须为散列表选择一个合适的数据结构来优化性能。在我们的例子中，我们的冲突处理策略工作得非常好，因为在链表中插入元素只需要常数时间，删除元素的时间和链表长度成比例。如果散列函数能够较好地分布元素，那么每一个链表相对都比较短。

解决方案

基于散列搜索算法有两个部分。第一个是创建散列表。例5-7的代码表明了如何使用链表来存储元素。我们使用迭代器来从集合C中获取元素然后输入。

例5-7：加载散列表

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

    // 从迭代器中取得元素然后寻找到合适的桶h。
    // 添加到存在的链表或者创建一个然后插入。
    while (it.hasNext()) {
        V v = it.next();

        int h = hashMethod.hash(v);
        if (listTable[h] == null) {
```

```

        listTable[h] = new LinkedList<V>();
    }

    // 将元素添加到链表的桶h。
    LinkedList<V> list = (LinkedList<V>) listTable[h];
    list.add(v);
}
}

```

注意表A是如何由桶组成的，每一个链表的类型都是LinkedList<V>（注9）。同样我们也要注意表是如何在链表中存储集合C中的元素的，而不是仅仅存储指针。

在表中查找元素是极为平常的。例5-8的代码做的就是这个工作。一旦散列函数返回散列表的索引，我们需要看看相应的桶是否为空。如果为空，返回空，表示查找的字符串不在集合中，否则我们在相应的桶中遍历链表来查找字符串。

例5-8：查找元素

```

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);
}

// 上面散列方法的实现。
int hash(V v){
    int h = v.hashCode();
    if (h < 0) { h = 0 - h; }
    return h % tableSize;
}

```

注意散列函数确保散列索引是在 $[0, \text{tableSize})$ 范围内的。如果在String类上使用hashCode函数，散列函数必须考虑到hashCode内整数计算可能会溢出并且返回负数。这个是要处理的，因为如果值是负数的话，那么取模运算也会返回负数（注10），例如，在String类上使用JDK的hashCode函数，字符串“aaaaaa”返回值为-1 425 372 064。

结果

也许相比其他的查找方法，基于散列的查找更能够反映出我们设计的优劣，尤其是我们选择了什么样的存储元素的数据集合。理解输入数据的动态性质是非常必要的，这有助于选择合适的数据结构。

注9： <v>是HashTable中的类型参数，此处表示可以存放任何类型元素。

注10： 在Java中，表达式 $-5\%3=-2$ 。

分析

基于散列的查找有着非常优秀的性能特征。我们将简略地分析一下。在散列表中查找一个元素可以分为如下几个部分：

- 计算散列值。
- 通过散列值索引到相应元素。
- 如果有冲突，那么寻找到需要找的元素。

所有基于散列的查找算法都包含头两个部分，但是不同的变种处理冲突的策略是不同的。

计算散列值的开销必须限制在常数时间内。如果你思考一下本节的例子，计算散列值将会和字符串的长度成比例。对于任何有穷单词集合来说，都存在一个最长的字符串，长度为 k 。如果 t_k 是计算最长字符串散列值的时间，那么它计算任何散列值需要 $\leq t_k$ 的时间。计算散列值因此被认为是一个常数时间的操作。

算法的第二部分也是时间的操作。如果表存储在二级存储器，根据元素的位置以及在设备上定位元素所需要的时间可能会有相应的改进算法，不过这个也是常数时间的操作。

如果我们能够证明计算的第三部分也是有着常数时间的上界，那么我们就很容易地证明基于散列的查找是常数时间的性能。当我们使用链时，我们将会定义一个负载因子， $\alpha=n/b$ ， b 是散列表中桶的数量， n 是存储在散列表中元素的数量。负载因子表示的是在单个链中的平均元素数量。

在链中寻找元素的最坏时间是 $O(n)$ ，只有在所有的元素都被映射到表中同样的桶时。要查找的桶的平均数目是 α 。如果希望能够知道详细的分析，请参见（Cormen等，2001）的第11章。

表5-5比较了例5-8的代码，它们都使用JDK类`java.util.Hashtable`，只是大小有不同而已。对于标有 $p=1.0$ 的测试来说，213 557个单词中的每一个都会被用来作为目标元素。对于标记为 $p=0.0$ 的测试来说，每一个单词都会用*来替换最后一个字符，以确保目标元素不会在表中。注意到我们在两种情况下都使用同样多的查找单词，确保散列的开销是相同的。我们进行了100次实验，并且抛弃掉最好和最坏的结果。表5-5表明了剩余98次实验的平均值。表5-6是我们在对散列表相关数据做的统计。随着负载因子的降低，链表的平均元素数量也下降了，这直接改善了性能。

散列表的规模加倍时，寻找一个元素的时间却下降了，因为链表的长度变得更短，从实践中得知，在 $b=1\ 045\ 875$ 时，没有链表包含超过5个元素。因为一个散列表能够变得

足够大，使得每个链表都很短，所以其查找性能可以看做是 $O(1)$ 。但是，即使是在（一直）有充足的内存以及优秀的散列函数来分布元素到散列表中时，这也是很少发生的情况。在 $b=1\,045\,875$ ，并且采用例5-6中的散列函数时，213 557个单词中大约81%被映射到唯一的桶中。

表5-5：不同的散列表规模下，查找需要花费的时间（单位：毫秒）

b	例5-8中的散列表		默认情况下的java.util.Hashtable	
	p = 1.0	p = 0.0	p = 1.0	p = 0.0
4,095	1143.33	2039.10	104.42	47.85
8,191	673.91	1095.67	99.48	45.26
16,383	433.87	615.63	111.74	48.60
32,767	308.03	364.88	107.11	47.33
65,535	237.07	245.86	98.53	45.14
131,071	194.37	172.81	98.53	44.47
262,143	164.85	118.93	96.62	44.32
524,287	144.97	79.87	94.24	44.97
1,048,575	136.97	62.42	96.77	43.22

表5-6中给出了随着 b 增长，散列表的实际负载因子。注意当 b 足够大时，只有一个元素的桶数目变得越来越多，而且链表的最大长度也在下降。你可以看到当散列表的规模变得足够大时，查找的性能就是 $O(1)$ 。

表5-6：样例代码得到的散列表统计数据

b	负载因子 α	链表 最小长度	链表 最大长度	只有一个元素的 桶数目
4,095	54.04	27	82	0
8,191	27.5	9	46	0
16,383	15	2	28	0
32,767	9.5	0	19	349
65,535	6.5	0	13	8,190
131,071	5	0	10	41,858
262,143	3.5	0	7	94,319
524,287	3.5	0	7	142,530
1,048,575	2.5	0	5	173,912

已有的java.util.Hashtable类性能比我们的代码要优秀，但是随着散列表规模的增长，节省的时间越来越少。因为java.util.Hashtable对List类进行了优化，能够高效地管理元素链表。此外，当负载因子太高时，java.util.Hashtable自动地“重新散列”整个散列表，重散列策略将在接下来的“变种”一节中讨论。它将增加构造散列表的开销，但是改善了查找的性能。如果我们阻止使用“重散列”策略，那么java.util.Hashtable的查找性能将和我们的实现的性能差不多。表5-7给出了重散列的次数以及构造散列表的总时间的关系。我们从之前描述过的单词表中构造散列表，在进行了100次实验，抛弃最好和最坏的结果后，这个表是剩余98次实验结果的平均值。在构造散列表来改善性能时，这个类的设计者进行了一些额外的计算（做了一个很常见的权衡）。在表5-7的第3、5列，我们注意到当重散列发生时，有着很明显的性能开销。并且也发现最后两行没有进行重散列，所以在列3、5、7中的结果是差不多的。通过重新构建散列表，减少元素链表的平均长度，重散列的确改善了整体性能。

表5-7：构建散列表的时间比较（单位：毫秒）

b	我们的	JDK散列表 ($\alpha=0.75f$)		JDK散列表 ($\alpha=4.0f$)		JDK散列表 ($\alpha=n/b$) 没有重散列
	散列表	构造时间	#重散列	构造时间	#重散列	构造时间
4,095	1116.08	165.77	7	166.81	4	644.31
8,191	647.32	162.96	6	165.48	3	364.56
16,383	421.57	164.20	5	162.85	2	230.42
32,767	332.82	164.36	4	149.29	1	164.08
65,535	273.77	155.62	3	131.19	0	131.05
13,1071	256.39	147.17	2	118.47	0	116.07
262,143	280.06	127.57	1	90.20	0	91.09
524,287	264.87	89.93	0	89.77	0	89.61
1,048,575	257.83	92.09	0	93.55	0	92.65

变种

基于散列查找的一个主要变种是修改了处理冲突的策略。与其在槽中放入一个元素的链表，我们能够使用一种叫做开放定址的技术，这种技术直接将冲突元素存储在散列表A中。如图5-7所示。使用了开放定址技术，散列表减少了存储开销，例如在冲突的链表中使用指针指向元素。使用开放定址技术，我们将散列函数调整成为带两个参数的函数， $h(u, j)=i, u \in U$ ， i 和 j 都是在范围 $[0, b)$ 中的整数， b 是A的规模。一般来说，我们使

$h(u,0)=i=h(u)$, h 是之前描述过的一个散列函数。如果 $A[i]$ 的槽被占用了, 并且这个元素和目标元素不匹配, 我们计算 $h(u,1)$ 的值。如果这个值指向的槽还是被占用了, 并且这个元素和目标元素不匹配, 那么我们计算 $h(u,2)$ 的值, 我们将重复这个过程, 直到目标元素被找到, 或者槽是空, 或者散列函数产生了一个我们之前已经访问过得槽 (此时就产生了一个错误)。

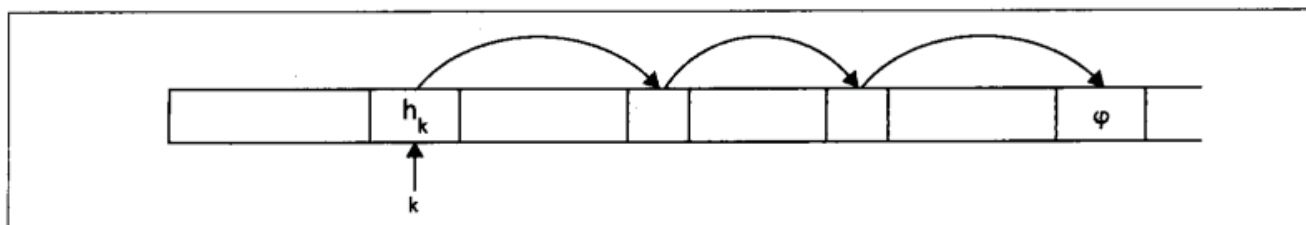


图5-7: 开放定址

假设我们能够确保我们不会重新访问一个槽, 那么这种方法的最坏性能是 $O(b)$ 。开放定址技术在查找时表现非常好。即使在一个不成功的查找中, 期望的探测数目也只是 $1/(1-\alpha)$, 并且最坏情况下, 成功查找的性能是 $(1/\alpha)\ln(1/1-\alpha)$ (注11), 详细信息请参见算法导论 (Cormen等, 2001)。开放定址技术中有两种广泛使用的探测方法。第一种是线性探测, 这种方法下, 我们的散列函数是 $h(u,j)=(h(u)+j)\bmod n$ 。当我们发现有冲突时, 我们简单地查看散列表中的下一个槽, 使用这个散列函数构成一个环形的查找。这个方法受元素的分布范围影响较大, 当元素分布比较密集时, 将会导致需要探测很多槽, 尤其是 α 接近于 1.0 时。为了避免这种情况, 我们可以使用二次方探测, 散列函数变为 $h(u,j)=(h(u)+f(j))\bmod m$, f 是 j 的一个二次方函数。

在表5-5, 我们看到在强制使用散列表不进行重散列之后, 节省了大量的时间。如果在一个元素插入之前, 原散列表的负载因子已经非常高了, 那么一个重散列操作是有必要的。当散列表包含的元素比其预想的要多时, 那么可以调整自身的大小。最典型的方式就是将其桶的数目加倍然后加一 (因为散列表通常包含奇数个桶)。当只有很少的桶可用时, 所有表中存在的元素必须重散列以放到新的位置。这是一个昂贵的操作, 但是能够减少未来查找的总开销, 不过这个操作不能非常频繁, 要不然散列表的性能将会退化。当你对散列表的性能不满时, 你必须允许散列表能够重散列使得元素平均分布到散列表中。java.util.Hashtable默认的负载因子是 0.75; 如果你将这个值设为 n/b , 那么它将永远不会重散列。

之前的“使用环境”一节中的例子使用了一个静态的字符串集合。当面对这种特殊情况时, 我们能够通过使用完美的散列函数, 得到最优化的性能。完美的散列使用两个散列函数。我们用一个标准的散列函数来索引主表 A 。每一个槽, $A[i]$, 指向一个小得多的

注11: \ln 为以 e 为底的自然对数。

二级散列表, S_i , 这个散列表和散列函数 h_i 绑定。如果有 k 个键值映射到槽 $A[i]$ 。那么 S_i 将包含 k^2 个槽。这看起来浪费了很多内存, 但是明智地选择初始散列函数能够减少这种浪费。选择合适的散列函数能够保证在二级表中不会存在冲突。也就是说, 我们能够得到一个性能为 $O(1)$ 的算法。完美散列分析的细节在算法导论 (Cormen等, 2001) 中能够找到。Doug Schmidt (1990) 发表过一篇非常优秀的论文, 论述了完美散列操作。网络上有一些免费的多种语言的完美散列函数生成器可以下载。

一般来说, 虽然有着很多潜在的元素 e_i 会得到特定的键值 k , 但是散列表也许被设计成只能存储其中的一个元素。也就是说, 如果 $e_i \in C$ 且 $e_j \in C$, 那么 $i=j$ 当且仅当 $key(e_i)=key(e_j)$ 。有这个限制的原因是使得给定键值 $key(e)$ 下, 能够快速寻找到元素 e 。如果原始集合 C 包含两个相同的元素, 那么仅仅只有一个会正确存储到散列表 A 中 (注12)。

二叉查找树

在内存中进行二分查找的效率我们已经看到了, 非常高效。但是, 当查找集合频繁地变动时, 二分查找的效率退化得非常严重。如果需要处理动态的数据集合, 我们需要采用一种不同的数据结构来得到可以接搜的查找性能。

要存储动态的查找数据集合的话, 查找树可能是最常用的数据结构了。无论是在内存中还是在二级存储器上, 查找树的性能都非常优秀。最经常使用的查找树是二叉查找树, 其每一个内部节点都有两个子节点。另外一种查找树—— B 树, 是一棵 n 元树, 是属于磁盘上的数据结构。

输入/输出

使用了查找树之后的输入和输出和二分查找的一样。集合 C 中每一个元素 e 都将被存储到查找树中, 并且这些元素都需要有一个或者更多的属性能够作为键值, 这些键值决定了全集 U 。元素也必须有能够区分它们的属性。查找树将会存储 C 中的元素。

使用环境

对于程序员来说, 内存泄漏是一个非常严重的问题。当程序需要长时间运行时, 例如现在的大多数服务器程序, 内存泄漏将会最终导致程序耗尽物理内存然后崩溃掉, 这就会产生非常严重的后果。

注12: GPERF for C/C++ 可以在 <http://www.gnu.org/software/gperf/> 下载到, JPERF for Java 可以在 <http://www.anarres.org/projects/jperf/> 下载到。

一个程序员可能会写一个程序来监视内存分配和回收，并且报告程序的使用情况，以便于监视内存泄漏。这个内存分析程序可以仅仅是简单地重写一个新的`malloc()`和`free()`函数来记录每一次分配和释放内存的信息。我们希望记录每一次内存分配操作，并且当内存释放时，我们必须及时更新信息。

在前面的描述中，我们不可能知道我们将要存储多少个元素。一个基于散列的查找也许可以正常工作。但是我们也许会选择一个错误的散列表规模，以至于影响高效的资源使用。一个替代的查找策略是采用查找树。当我们准备在内存中维护我们的查找数据时，我们使用二叉查找树能够很好地做到这个。二叉查找树在动态数据，尤其是插入和删除操作非常频繁时，性能表现非常好。

一棵二叉查找树， T 是一个节点的有限集合，并且节点之间能够通过有序的特征，如键值，来进行区分。节点的集合可以是空，或者只是包含了一个根节点 n_r 。每一个节点 n 都指向了两棵子二叉查找树， T_l 和 T_r ，并且满足这样的性质：如果 k 是节点 n 的键值，那么所有在 T_l 的元素的键值都小于等于 k ，所有在 T_r 的元素的键值都大于 k 。这个性质叫做二叉查找树性质（Cormen等，2001）。图5-8给出了一棵二叉树的例子。每一个节点都有一个整数键值，唯一的区分节点。你可以看到在图5-8的树中寻找一个键值只需要从根节点开始，最多探测3个节点。这棵树是完全平衡的。也就是说，每一个节点恰好有两个子节点。一棵完全平衡二叉树有 $2^n - 1$ 个节点。

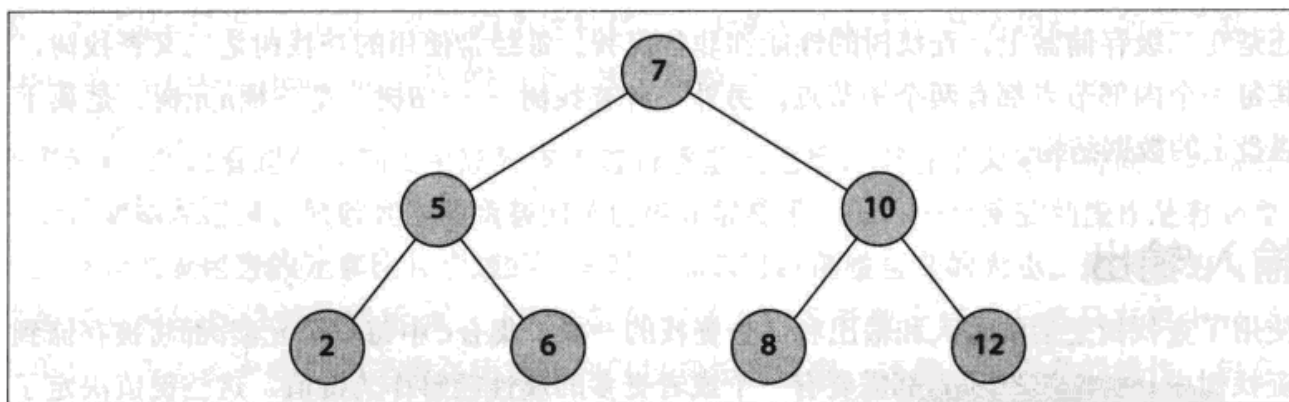


图5-8：一棵简单的二叉查找树

树也可能不会总是平衡的。在最坏情况下，一棵树可能退化到链表的性能。考虑使用和图5-8相同的节点，但是如果按照图5-9的方式来排列的话。节点被按照升序添加，虽然这个结构的确满足了二叉树的严格定义，每一个节点的左子树都是空的。这棵树简直就是一个链表。

驱动因素

如果我们简单地需要定位一个查找元素，第一选择是基于散列的查找。一些因素可能会促使我们选择二叉查找树。

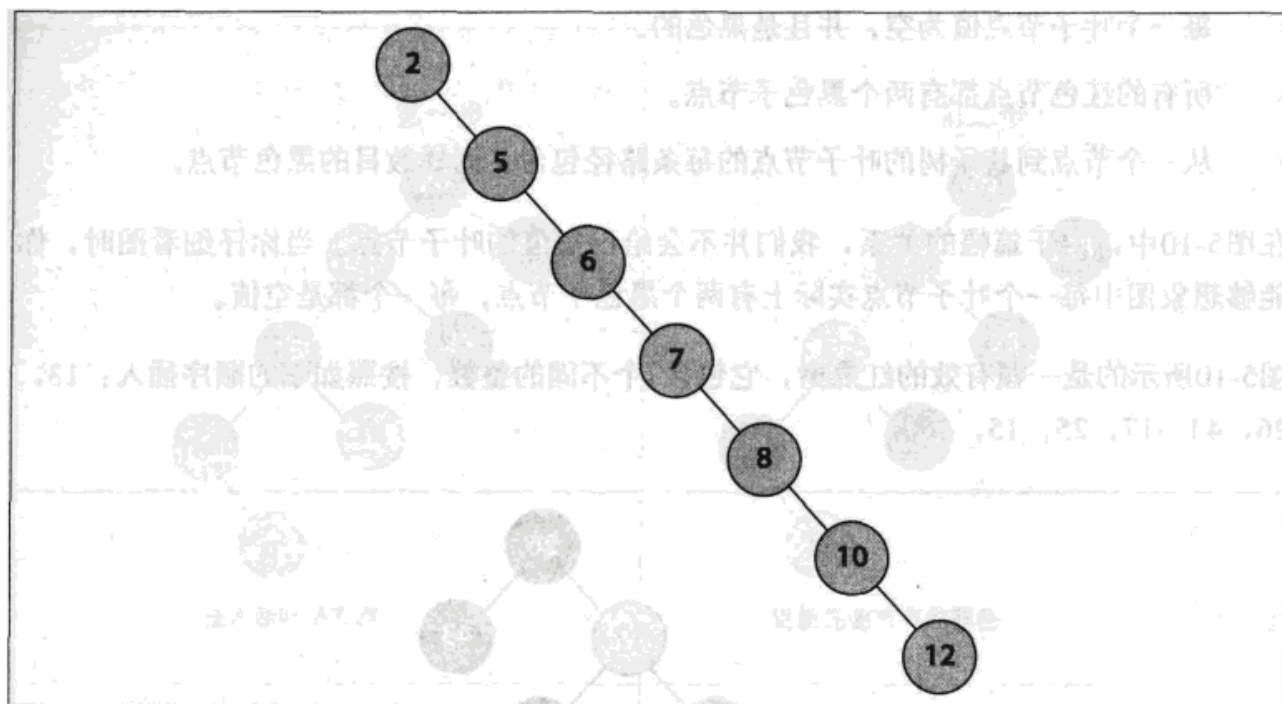


图5-9：一棵退化的二叉查找树

- 数据规模位置，并且程序必须能够处理任何规模的数据。
- 数据集是高度动态的，在程序运行时候会有大量的插入和删除操作。
- 程序需要按照升序或者降序来遍历元素。

一旦我们决定使用二叉查找树，我们必须做出如下的决定。

- 如果我们需要从任何指定节点起有序遍历数据集，那么在节点的结构中必须包括指向父节点的指针。
- 如果数据是动态的，我们必须平衡树。

在大多数应用程序中，我们需要平衡树结构来避免出现偏置，偏置树的某些分支可能会比其他分支长或者短很多，在最坏情况下，会导致图5-9那样的退化树。两个流行的平衡树可以使用。一个是AVL树，由Adel son-Vel skii和Landis于1962年提出。一棵AVL树遵循如下的平衡性质：任何一个节点的子树的高度都不可能比其他节点的子树的高度大1。

最近更加频繁使用的平衡树叫做红黑树。红黑树是一个近似平衡的。使用红黑树能够保证任何一个分支的长度都不会超过其他的分支长度的两倍。一个红黑树满足如下条件（Cormen等，2001）：

- 每一个节点被标记为红色或者黑色。
- 根节点是黑色的。

- 每一个叶子节点值为空，并且是黑色的。
- 所有的红色节点都有两个黑色子节点。
- 从一个节点到其子树的叶子节点的每条路径包含了同样数目的黑色节点。

在图5-10中，由于篇幅的关系，我们并不会给出空值的叶子节点。当你仔细看图时，你能够想象图中每一个叶子节点实际上有两个黑色子节点，每一个都是空值。

图5-10所示的是一颗有效的红黑树，它包含7个不同的整数，按照如下的顺序插入：13，26，43，17，25，15，16。

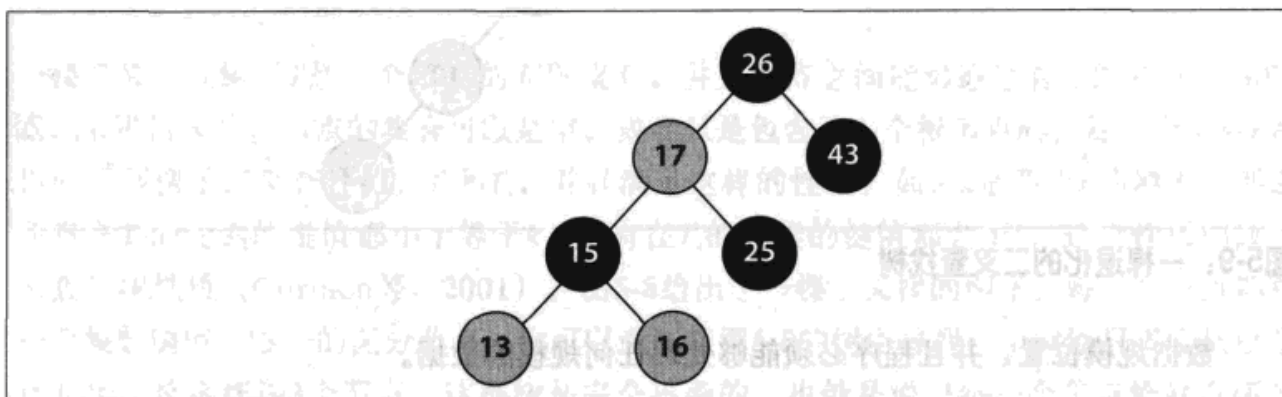


图5-10：红黑树样例

现在我们希望加入一个键值是14的节点。按照如上所述的性质，14将会被作为13的右子节点。红黑树插入算法将会修改这棵树，如图5-11所示，我们将在下一节“解决方案”中描述这个过程。

解决方案

在一棵红黑树中查找和在其他任何二叉查找树中查找没什么不同。例5-9是在红黑树中进行查找的代码。从根节点开始，我们将会检查每一个节点，如果键值小于左子节点，那么我们准备开始遍历左子树，否则遍历右子树。

例5-9：查找的Java代码

```

public V search(K k) {
    BalancedBinaryNode<K,V> p = root;
    while (p != null) {
        int cmp = compare(k, p.key);
        if (cmp == 0) {
            return p.value;
        } else if (cmp < 0) {
            p = p.left;
        } else {
            p = p.right;
        }
    }
}
  
```

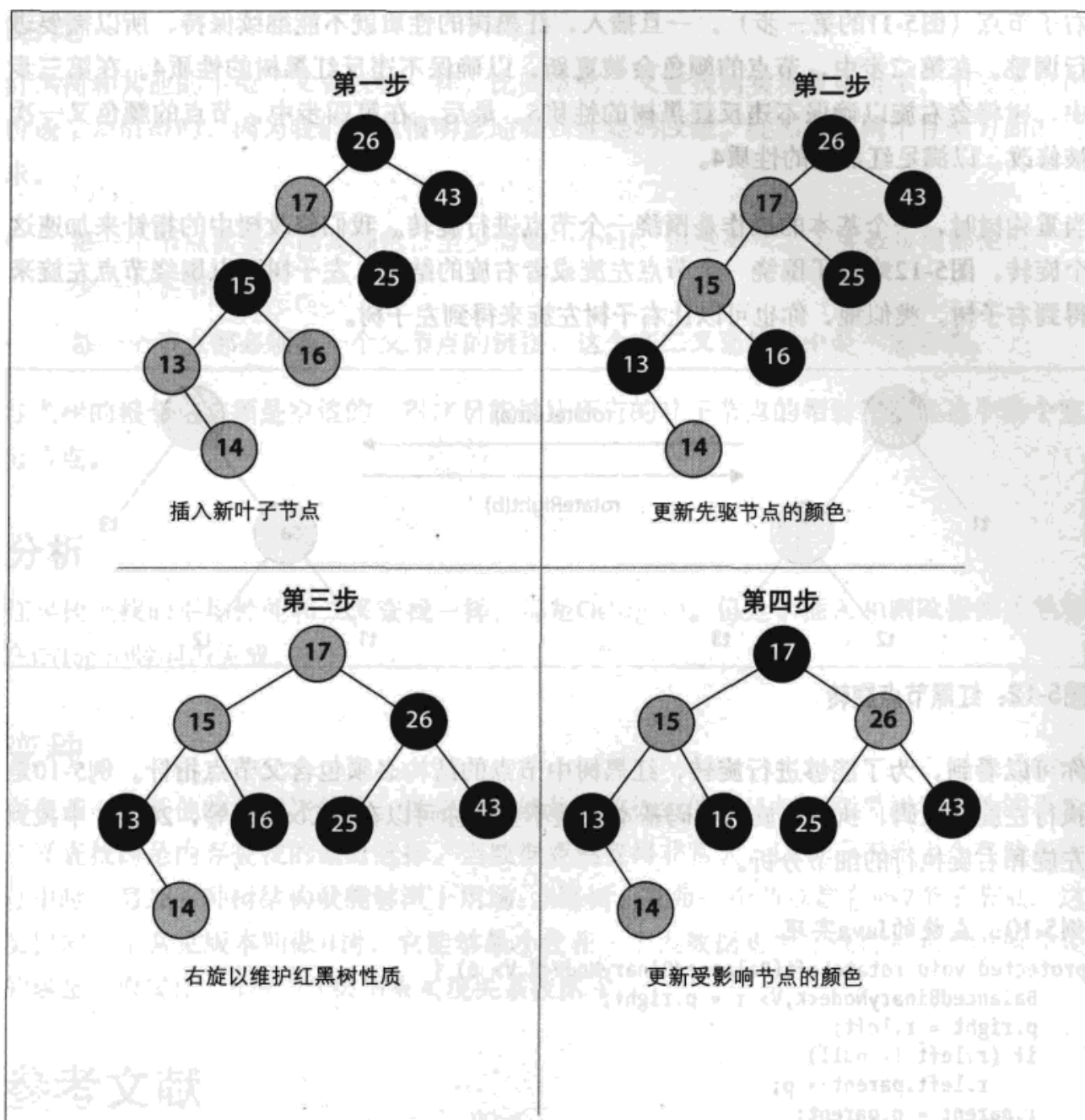


图5-11：添加键值14的元素到红黑树所需要的4步

```

    }
}

// 未找到
return null;
}

```

红黑树算法的关键是在于不断地插入和删除操作时保持红黑树的性质。我们在这里描述插入的过程,删除的过程是非常复杂的,我们将在代码中实现这个过程,你可以在 `algs.model.tree.BalancedTree` 类中阅读相关代码。

当我们在图5-10的树中插入14时,一个值为14的新节点将会成为值为13的节点的

右子节点（图5-11的第一步）。一旦插入，红黑树的性质就不能继续保持，所以需要进行调整。在第二步中，节点的颜色会被更新，以确保不违反红黑树的性质4。在第三步中，树将会右旋以确保不违反红黑树的性质5。最后，在第四步中，节点的颜色又一次被修改，以满足红黑树的性质4。

当重构树时，一个基本的操作是围绕一个节点进行旋转。我们修改树中的指针来加速这个旋转。图5-12给出了围绕一个节点左旋或者右旋的结果。左子树可以围绕节点左旋来得到右子树。类似地，你也可以让右子树左旋来得到左子树。

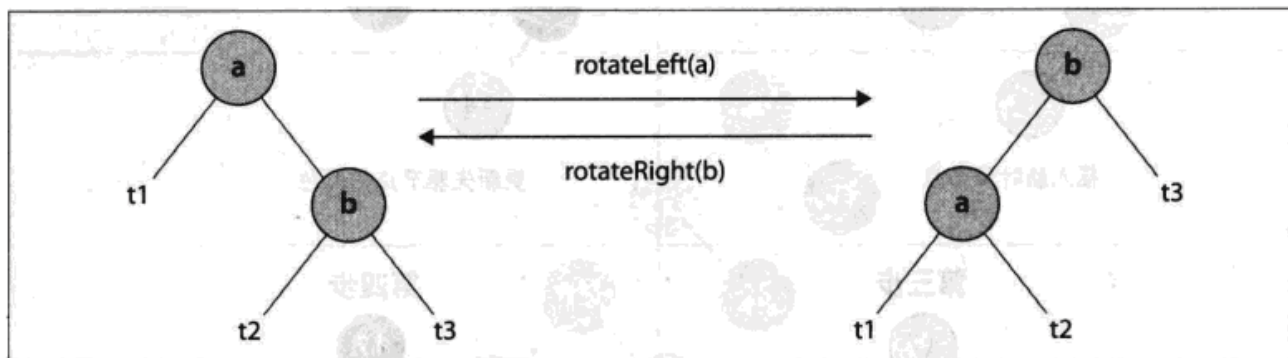


图5-12：红黑节点旋转

你可以看到，为了能够进行旋转，红黑树中节点的结构必须包含父节点指针。例5-10是执行左旋的代码，执行右旋的代码基本上差不多。你可以在（Cormen等，2001）中找到左旋和右旋执行的细节分析。

例5-10：左旋的Java实现

```
protected void rotateLeft(BalancedBinaryNode<K,V> p) {
    BalancedBinaryNode<K,V> r = p.right;
    p.right = r.left;
    if (r.left != null)
        r.left.parent = p;
    r.parent = p.parent;
    if (p.parent == null)
        root = r;
    else if (p.parent.left == p)
        p.parent.left = r;
    else
        p.parent.right = r;
    r.left = p;
    p.parent = r;
}
```

注意旋转操作维护的是二叉查找树的性质，因为节点的序并没有改变。一旦新的节点插入到红黑树中，树将会自动调整以保持红黑树的性质4和性质5。

结论

红黑树和其他的平衡二叉查找树一样，比简单的二叉查找树实现复杂得多。不过这个代价通常是值得的，因为我们可以很明显地看到性能的改进。红黑树有两个存储方面的需求。

- 每一个节点需要存储其颜色，至少需要一个bit，但是事实上大多数实现都使用了至少一个字节。
- 每一个节点都必须有一个父节点的链接，这个在二叉查找树中是不需要的。

红黑树的根节点必须是空值的。程序员能够让所有的叶子节点的指针都指向这个单个空值节点。

分析

红黑树查找的平均性能和二叉查找一样，都是 $O(\log n)$ 。但是，插入和删除操作也能够，在 $O(\log n)$ 时间内完成。

变种

存在着一些其他的平衡树结构。最常见的就是之前提到的AVL树。红黑树和其他的平衡二叉查找树是内存查找的最好选择。当数据规模变得非常大，以至于不能完全存储在内存中时，另外一种树结构就能够派上用场： n 路树，其每一个节点都有 $n > 2$ 个子节点。这类树的一个常见版本叫做B树，它能够最小化在一个大数据集中寻找特定元素时所需要的磁盘存取操作。B树同样被用来实现关系数据库。

参考文献

Adelson-Velskii, G. M., and E. M. Landis, "An algorithm for the organization of information," *Soviet Mathematics Doklady*, 3:125 – 1263, 1962.

Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to Algorithms*, Second Edition. McGraw-Hill, 2001.

Hester, J.H. and D.S. Hirschberg, "Self-Organizing Linear Search," *ACM Computing Surveys*, 17 (3): 295 – 312, 1985.

Knuth, Donald, *The Art of Computer Programming, Volume 3: Sorting and Searching*, Third Edition. Addison-Wesley, 1997.

Schmidt, Douglas C., “GPERF: A Perfect Hash Function Generator,” Proceedings of the Second C++ Conference : 87–102, 1990. Available at <http://citeseerx.ist.psu/viewdoc/summary?doi=10.1.1.44.8511>, accessed May 10, 2008.





概述

图是计算机科学中用来表示复杂结构信息的一种基本结构。图6-1所示的就是图。

本章我们会讨论一些通用的图表示法，以及一些频繁使用的图算法。本质上来说，一个图包含一个元素集合（也就是顶点），以及元素两两之间的关系（也就是边）。由于应用范围所限，本章我们仅仅讨论简单图，简单图并不会如（a）那样有一个顶点的一条边是自己指向自己，以及不会如（b）那样一对顶点之间存在多条边。

图

一个图 $G=(V,E)$ 由一个顶点集 V 以及一个边集 E 组成。算法中通常会出现如下几种图：

无向图

顶点 (u,v) 之间的关系模型不需要考虑关系的方向如何。在处理对称信息时，这种图非常有用。例如，汽车可以在A镇和B镇之间的路上双向行驶。

有向图

顶点 (u,v) 之间的关系和顶点 (v,u) 之间的关系不同，后者或许不存在。例如，一个提供驾驶信息的程序必须存储单线道路的信息，以避免给出错误的方向。

有权图

顶点 (u,v) 之间的关系是有权重的。可以是数值的也可以是非数值的。例如，在A镇和B镇之间的道路有公里数，当然也包含了驾驶时间这个信息。

超图

在顶点 (u,v) 之间可能存在多种关系，本章我们将讨论限定在简单图上。

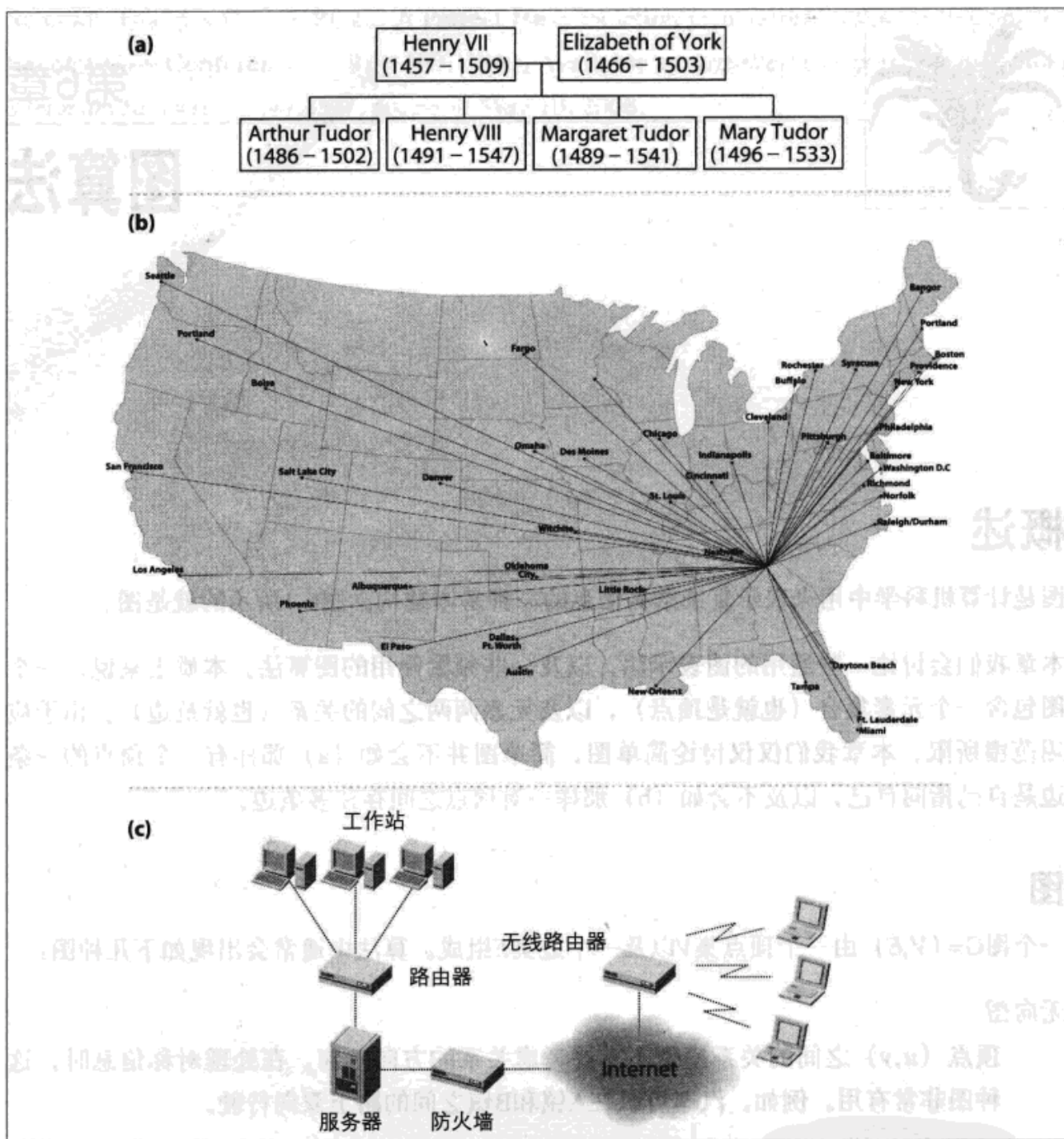


图6-1: (a) 都铎王朝, (b) 计算机网络, (c) 航线时间表

如果图中任意两点之间都存在一条边, 那么这个图是连通图。一个有向有权图的定义为: 一个非空顶点集合 $\{v_0, \dots, v_{n-1}\}$, 一个有向的边集合, 每对顶点之间只有一条边, 以及每条边上都有一个正权值。在很多应用中, 这个权值被认为是距离或者开销。对某些应用来说, 我们希望能够放宽权值必须为正的的限制 (例如, 负值可以反映利润情况), 但是我们必须高度关注这样做的后果。

考虑图6-2的有向有权图, 它由6个顶点4条边组成。存储这个图有两个标准的数据结构: 它们都显式地存储了权值, 隐式表示了边的方向。一种数据结构叫邻接表, 如图6-3所

示, 在这个数据结构中每一个顶点 v_i 维护一个顶点的链表, 链表中的每一个元素存储有 v_i 到邻接节点的边权重。这种基础数据结构是顶点的一维数组。添加一条边时, 需要额外的处理来保证不会出现重复的边。

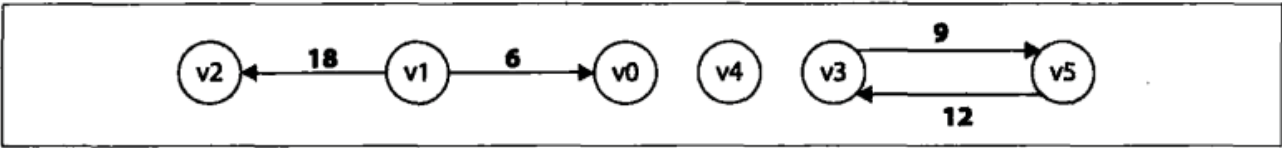


图6-2: 有向有权图的例子

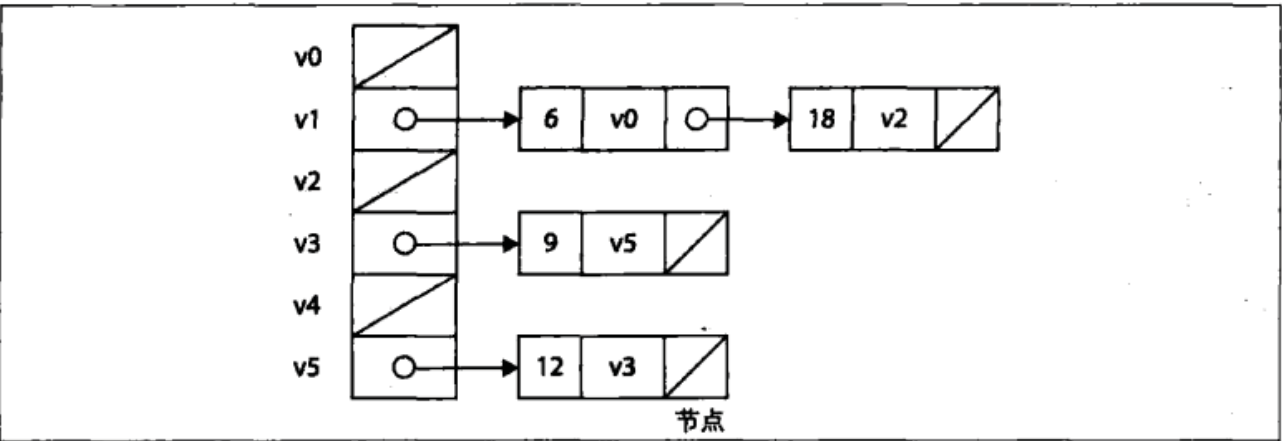


图6-3: 有向有权图的邻接表表示

图6-4则是表明了如何用一个 n 阶的邻接矩阵来存储有向有权图, 每个维度可以被索引。条目 $A[i][j]$ 存储的是从 v_i 到 v_j 的边的权值; 当 $A[i][j]=0$ 时, 顶点 v_i 和顶点 v_j 之间不存在边。使用邻接矩阵表示法, 添加一条边只需要常数时间。

	v0	v1	v2	v3	v4	v5
v0	0	0	0	0	0	0
v1	6	0	18	0	0	0
v2	0	0	0	0	0	0
v3	0	0	0	0	0	9
v4	0	0	0	0	0	0
v5	0	0	0	12	0	0

图6-4: 有向有权图的邻接矩阵表示

我们也可以使用邻接表和邻接矩阵来存储无向图。看看图6-5的无向图, 我们使用 $\langle v_0, v_1, \dots, v_{k-1} \rangle$ 来描述一条有 k 个顶点的路。这条路遍历了 $k-1$ 条边, 一个有向图的路径是由同一方向的边组成。在图6-5中, 路径 $\langle v_3, v_1, v_5, v_4 \rangle$ 是有效的。环是一个多次包含了同一个顶点的路径。一个环通常用其最小形式表示。在图6-5中, 在路 $\langle v_3, v_1, v_5, v_4, v_2, v_1, v_5, v_4, v_2 \rangle$ 中存在一个环, 这个环可以用 $\langle v_1, v_5, v_4, v_2, v_1 \rangle$ 来表示。注意图6-2的有向有权图, 存在一个环 $\langle v_3, v_5, v_3 \rangle$ 。

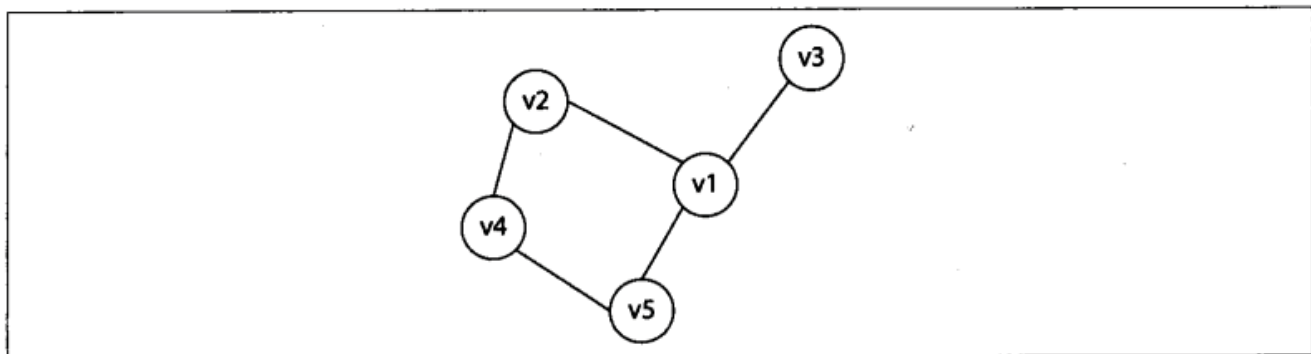


图6-5：无向图样例

当使用邻接表来存储无向图时，同一条边 (u,v) 在邻接表中出现了两次，一次是在 u 的邻接节点链表，一次是在 v 的邻接节点链表。因此，相比同样数目的顶点和边的有向图，无向图在邻接表中存储所需要的存储空间是前者的两倍。当使用邻接矩阵来存储无向图时，你必须保证条目 $A[i][j]=A[j][i]$ ；不需要任何的额外存储空间。

存储问题

当使用二维矩阵来表示元素集合中 n 个元素的潜在关系时，需要注意一些事项。首先，矩阵需要 n^2 个元素的存储空间，即使有时候元素之间的关系集合非常小。在这些情况下，也就是所谓的稀疏图，由于计算机内存的限制，不可能存储超过数千个顶点的图。例如，Java虚拟机堆使用默认值256MB，创建一个二维的 `int[4096][4096]` 矩阵就已经超出可用内存。即使程序员能够在一台有着更多内存的计算机上执行程序，事实上创建的矩阵规模存在一个固定的上界。另外，在大规模矩阵中存储稀疏图，遍历矩阵来寻找边的操作的开销会很大，并且这种存储方法也限制了更多的改进。第二，当顶点之间有多种关系时，不适合使用矩阵。为了能够在矩阵中存储这种关系，矩阵中每一个元素将会当作一个表。

每一种邻接表示法都存储了相同的信息。假设，你写了一个程序，需要计算出世界上任意两个城市之间的最便宜航班。每条边的权重和两个城市之间最便宜直航的价格有关（假设航线不会中转）。根据2005年国际机场协会（ACI）发布的一份报告，世界上有总共1659个机场，这样也就需要一个有2 752 281个条目的二维矩阵。那么就有一个问题：“多少条目是有值的”，它的答案取决于直航航线的数目，ACI的报告也显示，2005年有71 600 000次“航空活动”，简单地说每天有大概196 164次航班。即使所有的航班都是在两个不同机场之间的直航（显然直航的数目要比这个小很多），这个矩阵也有93%是空值——一个非常好的稀疏矩阵的例子！

当使用一个邻接表表示无向图时，我们可以使用一些方法来减少需要的存储空间。假设一个顶点 u 的邻接顶点为：2、8、1、5、3、10、11和4。首先，这些邻接顶点会以升序来存储，这样做能够快速定位是否存在一条边 (u,v) 。在这种结构下，虽然有8个邻

接顶点，但是确定是否边 $(u,6)$ 是否存在只需要6次比较。当然，添加一条边就不再是常数时间，这里我们需要权衡一下利弊。其次，为了能够更加高效地检查是否边 (u,v) 存在，邻接表可以存储邻接顶点的范围，例子中的邻接表中八个顶点可以减少到三个：1~5、8、10~11。这种方法也能够减少是否存在一条边的计算次数，不过这会降低添加或者删除边的效率。

图分析

当使用本章的算法时，决定使用邻接表还是邻接矩阵的最重要因素是图是否为稀疏图。算法的性能是和图的顶点数 $|V|$ 还有边数 $|E|$ 相关的。和其他算法书一样，我们简化了性能的公式，无论在最好、最坏还是平均情况下，我们都使用 V 、 E 的大 O 记法来表示性能。 $O(V)$ 表示需要和图中顶点数成直接比例的计算步数。但是图中边的密度也必须考虑。在稀疏图中， $O(E)$ 近似等于 $O(V)$ ，然而在稠密图中，它近似等于 $O(V^2)$ 。

我们将会看到，根据图的结构，一些算法会有两个变种，并且这两个变种的性能不尽相同，一个变种也许执行时间为 $O((V+E)*\log V)$ ，而另外一个为 $O(V^2+E)$ 。哪个更加高效呢？表6-1告诉我们这个答案取决于图 G 是稀疏图还是稠密图。对于稀疏图来说， $O((V+E)*\log V)$ 显然更加高效一些，而对于稠密图， $O(V^2+E)$ 就更加快速了。标记为“均衡图”的条目，这种图的期望性能在稀疏图和稠密图上都是相同的，为 $O(V^2)$ ，在这些图上，边的数目大致等于 $O(V^2/\log V)$ 。

表6-1：两种算法变种的比较

图类型	$O((V+E)*\log V)$	比较关系	$O(V^2+E)$
稀疏图： E 是 $O(V)$	$O(V \log V)$	小于	$O(V^2)$
均衡图： E 为 $O(V^2/\log V)$	$O(V^2+V*\log V)=O(V^2)$	等于	$O(V^2+V^2/\log V)=O(V^2)$
稠密图： E 为 $O(V^2)$	$O(V^2 \log V)$	大于	$O(V^2)$

数据结构设计

图6-6的UML图中是本章我们将会在图上执行的关键操作，参考第3章的UML图。C++ Graph类使用的是邻接表来存储的（有向或者无向）图，邻接表是用C++STL的核心类来实现的。并且，它将众多的表存储在数组中，一个顶点一个表。顶点 u 的表存储的是IntegerPair类，表示权重为 w 的边 (u,v) 。

```
using namespace std;
enum vertexColor { White, Gray, Black };
enum edgeType { Tree, Backward, Forward, Cross };
// 对于每个顶点 u, 将权值为 w 的边 (u, v)
// 的信息 (v, w) 存放于此
typedef pair <int,int>      IntegerPair;

// 顶点的邻接表
typedef list<IntegerPair>   VertexList;
```

Graph
#VertexList *vertices_ #int n_ #bool directed_
+Graph() +Graph(int n, bool directed) +Graph(int n) ~Graph() +void load(char *file) +bool directed() +int numVertices() +bool isEdge(int u, int v) +bool isEdge(int u, int v, int &weight) +int edgeWeight(int u, int v) +void addEdge(int u, int v) +void addEdge(int u, int v, int weight) +bool removeEdge(int u, int v) +VertexList::const_iterator begin (int u) +VertexList::const_iterator end(int u)

图6-6：关键图操作

图6-6的操作可以分为如下几类：

创建

图是创建自 n 个顶点的集合，也许有向，也许无向。`load(char *)`读取文件中的点和边的信息来更新图。如果图是无向的，那么添加边 (u, v) 的同时添加了边 (v, u) 。

查询

我们可能会做如下的查询：图是否为有向图，寻找给定顶点的出边，查询是否某条边存在，查询边的权值。程序员可以创建一个迭代器，返回图中任一顶点的邻边（以及其权值）。

更新

程序员可以在图中删除或者添加一条边。

问题

使用图结构可以解决很多问题。本章我们只是讨论其中一些，不过你仍然有机会寻找一些我们没有讨论的问题，然后去研究。给定一个用边来定义的图，很多问题会和图中两顶点之间的最短路径相关，路的长度就是路上边的长度之和。在“单源最短路径”这个问题（例6-6）上，给定一个顶点 s ，我们需要计算出到图中其他所有顶点的最短路径。在“对顶点间最短路径”问题（例6-7）中，我们需要计算图中所有的顶点对 (u,v) 之间的最短路径。有些问题则是在更加深入地使用了图的结构。我们可以从一个图中构造出一个最小生成树（MST），最小生成树是一个无向有权图，是原图边的一个子集，但是，原图的顶点仍然是连通的，而边的权值总和最小。在本章稍后部分，“最小生成树算法”一节，我们将会描述如何高效地解决这个问题。

我们首先从如何探索图开始讨论。两个最常使用的搜索方法是深度优先搜索（DFS）和广度优先搜索（BFS）。

深度优先搜索

考虑图6-7左边的迷宫。在经过一些尝试后，一个孩子能够快速找到从起点 s 到终点 t 的路。但是计算机解决这个问题看起来就比较复杂，一种方法是假设离目标并不太远，然后做尽可能多的深度移动。也就是说，只要可以，随机选择一个方向，然后向这个方向前进，标记一下起点。如果你走上一条死路，或者不能在不重新访问顶点的情况下做任何深度移动，那么就会退到另一条未访问的分支上，并且向这个方向前进。图6-7右边的数字表示的是一个解的分支点；事实上，在这个解中，我们访问了迷宫中的所有点。

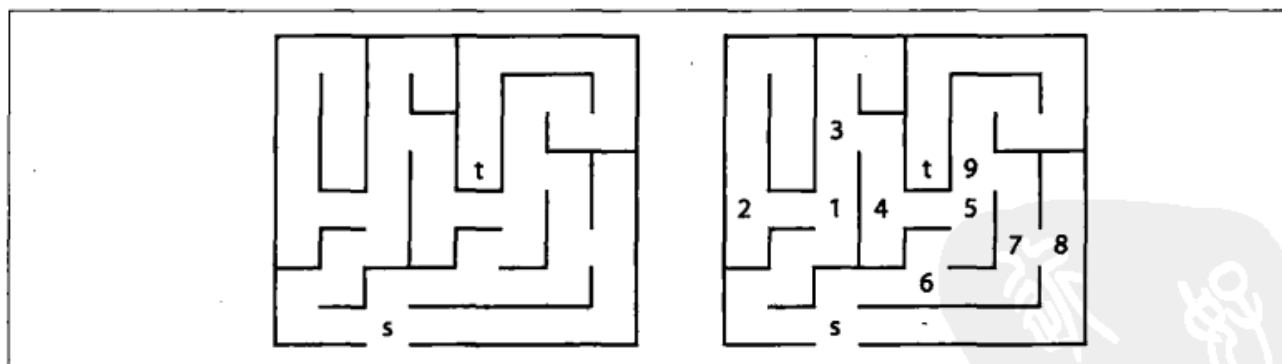


图6-7：从 s 到 t 的一个小迷宫

我们能够用一个包含了点和边的图来表示图6-7的迷宫。一个顶点表示迷宫中的每一个分支点（在图6-7的右边用数字标记），当然也包括“末路点”。如果迷宫中在两个顶点之间存在一条有向路，并且在这个方向上没有其他的选择，那么我们就说存在一条边。从图6-7得到的迷宫的无向图表示如图6-8所示，每一个顶点都有一个唯一标识符。

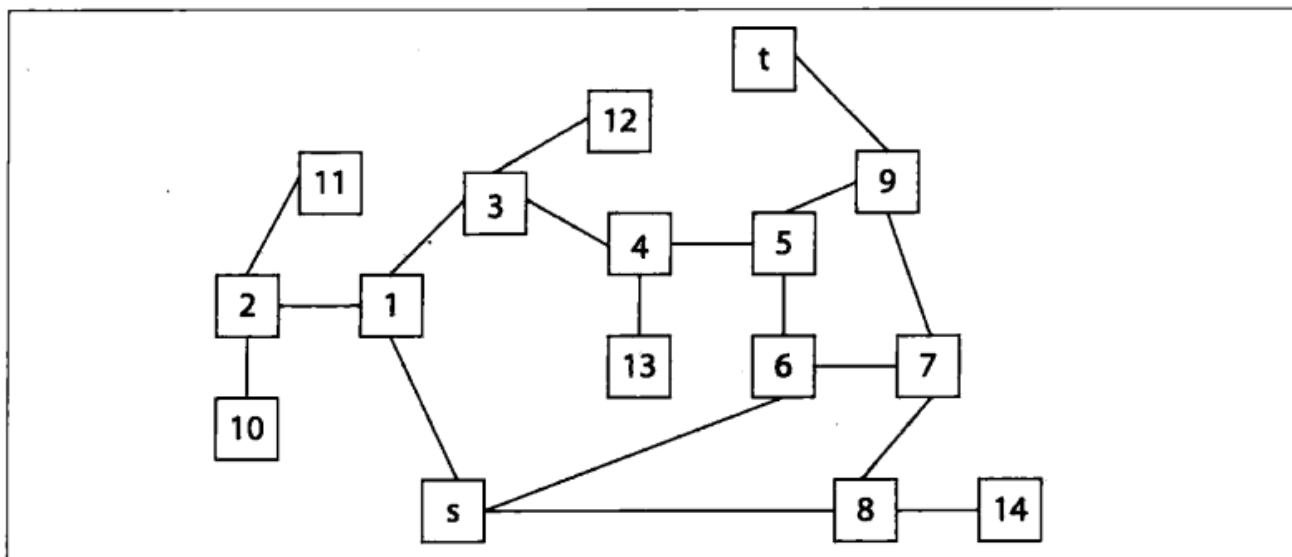


图6-8: 图6-7的迷宫的图表示

为了解决这个问题，我们需要知道在图 $G=(V,E)$ 的顶点 s 到顶点 t 间是否存在一条路。本例中，所有的边都是无向的，但是即使在迷宫上加一些限制条件，我们也能够非常容易地将其看成有向图。

图6-9的详解包括了用伪代码描述的深度优先搜索。深度优先搜索的核心是递归的`dfs_visit(u)`操作，这个操作访问之前没有访问过的顶点 u 。并且这个操作通过对顶点染色记录下了搜索进程。顶点可能会染成如下三种颜色：

白色

顶点还未访问。

灰色

顶点已经被访问过了，但是其可能还有没有被访问过的顶点。

黑色

顶点以及其所有的邻接顶点都已经被访问过了。

首先，所有的顶点初始为白色，然后深度优先搜索在源顶点 s 上调用`dfs_visit`。在对 u 所有的未访问邻接顶点（它们为白色）递归调用`dfs_visit`之前，`dfs_visit(u)`将 u 染成灰色。一旦这些递归调用完成，那么我们将 u 染成黑色，然后函数返回。当递归函数`dfs_visit`返回，深度优先搜索开始回溯，直到回溯到一个有邻接顶点未被访问过的顶点（事实上，回溯到标记为灰色的顶点）。

对有向和无向图来说，深度优先搜索从 s 开始，访问了图中所有 s 可达的顶点。如果 G 中仍然有未访问，但是从 s 不可达的顶点存在，深度优先搜索将会随机从其中选择一个作为源点，然后重复操作。这个过程将会一直重复，直到 G 中所有的顶点都被访问。

深度优先搜索

最好情况

平均情况

最坏情况

$O(V+E)$

$O(V+E)$

$O(V+E)$



图



递归



数组



回溯

depthFirstSearch (G, s)

1. **foreach** $v \in V$ **do**

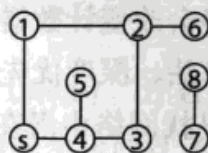
2. $d[v] = f[v] = \text{pred}[v] = -1$

3. $\text{color}[v] = \text{White}$

4. $\text{counter} = 0$

5. **dfs_visit** (s)

dfs_visit 递归地访问顶点1~5 将其标记为灰色，直到它找到了一个点，其有非白色的邻点（例如5）



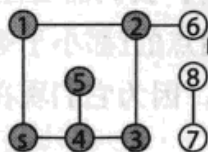
6. **foreach** $v \in V$ **do**

7. **if** ($\text{color}[v] = \text{White}$) **then**

8. **dfs_visit** (v)

end

每一次 **dfs_visit** 调用完毕，最开始忽略的未访问顶点就可以找到了（例如6是2的白色邻点），完成的顶点被标为黑色



dfs_visit (u)

1. $\text{color}[u] = \text{Gray}$

2. $d[u] = ++\text{counter}$

3. **for each neighbor** v **of** u **do**

4. **if** ($\text{color}[v] = \text{White}$) **then**

5. $\text{pred}[v] = u$

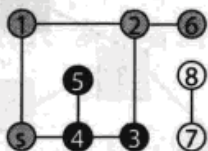
6. **dfs_visit** (v)

7. $\text{color}[u] = \text{Black}$

8. $f[u] = ++\text{counter}$

end

如果图为不连通的，那么一些顶点将会标记为白色，继续探测这些顶点



$\text{pred}[]$ 记录的深度优先森林，如箭头所示

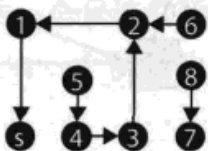


图6-9：深度优先搜索详解

在这个执行过程中，深度优先搜索遍历图中的边，计算和复杂的图结构有关的信息。深度优先搜索维护一个计数器，在一个顶点第一次被访问（标记为灰色）和完成在这个顶点上的深度优先搜索（标记为黑色）时，这个计数器增加计数。对于每个顶点，深度优先搜索记录如下信息。

$\text{pred}[v]$

前驱顶点，用来恢复从源点 s 到顶点 v 的路。

discovered[v]

其值为当深度优先搜索第一次访问 v 时，计数器增加后的值，简写为 $d[v]$ 。

finished[v]

其值为完成在这个顶点 v 上的深度优先搜索，计数器增加后的值，简写为 $f[v]$ 。

顶点访问的顺序将会改变计数器的值，所以需要注意邻接节点的顺序。对很多构建在深度优先搜索上的算法，深度优先搜索计算出来的信息都是非常有用的，包括拓扑排序，寻找强连通部，寻找网络中潜在的弱点。让我们来看看图6-8，假设顶点的邻接顶点是升序排列。那么计算出来的信息如图6-10所示。当计数器为18，顶点8正在被访问时，让我们看看图中顶点的颜色。图中有些部分（例如标记为黑色的点）已经被完全搜索并且不会被重复访问。我们需要注意白色顶点的 d 都将大于18（因为它们现在还没有被访问），以及黑色顶点的 f 都小于等于18，因为它们已经被完全搜索过，灰色顶点的 d 小于等于18， f 大于18，因为它们现在正处于某条递归访问的路上。

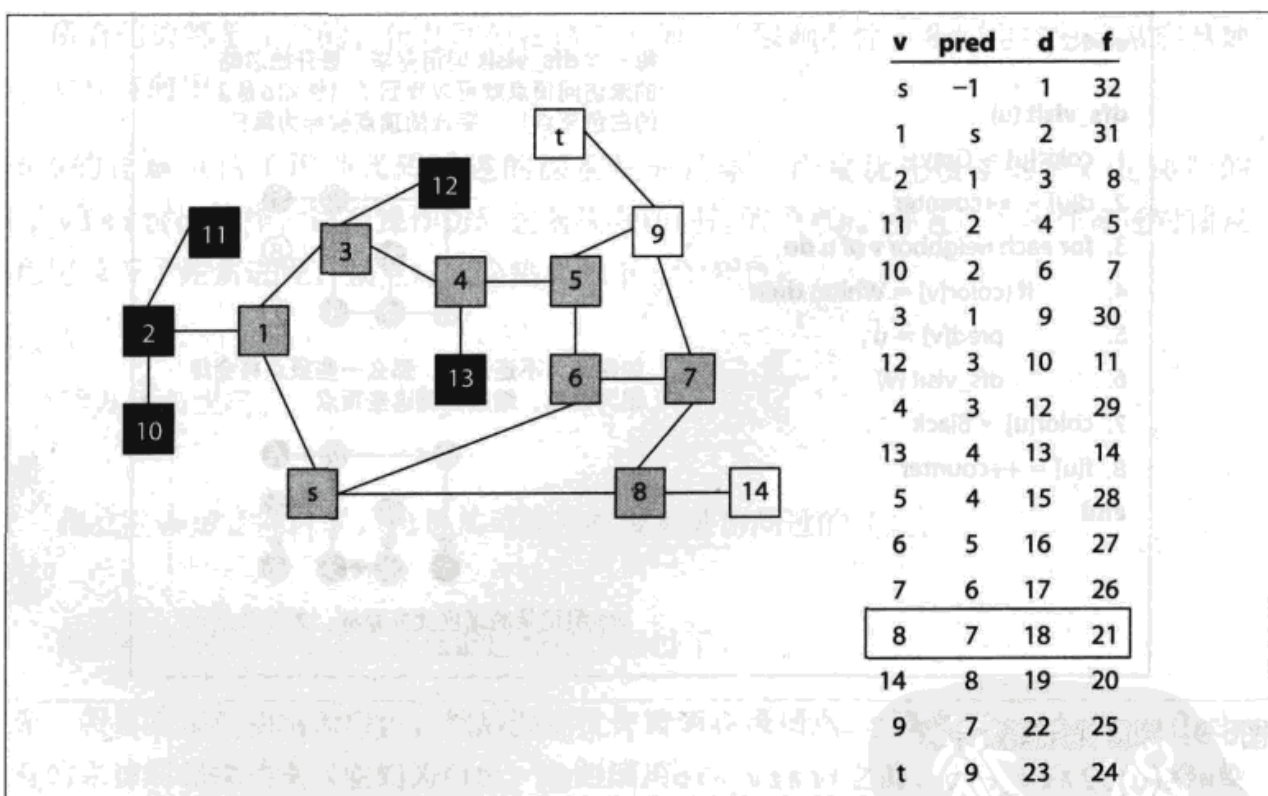


图6-10：在一个例图上计算 d 、 f 、 $pred$ ，计数器为18时顶点的着色情况

深度优先搜索没有对图的一个整体认识，所以它是盲目地搜索顶点 $\langle 5, 6, 7, 8 \rangle$ ，即使是南辕北辙。一旦深度优先搜索结束， $pred[]$ 的值能够用来生成一条从任意顶点到源点 s 的路。当然，这条路也许不会是最短路径，例如 $\langle s, 1, 3, 4, 5, 9, t \rangle$ ，但是最短的路径是 $\langle s, 6, 5, 9, t \rangle$ （注13）。

注13： 注意，这里的“最短路径”指的是 s 到 t 经过的点的个数最少。

输入/输出

输入

一个图 $G=(V, E)$ ，源点是 $s \in V$ 。 n 表示的是 G 中的顶点数。

输出

深度优先搜索计算三个数组。 $d[v]$ 表示的是 v 第一次被访问时，计数器的值。 $pred[v]$ 是深度优先搜索时，顶点 v 的先驱顶点。 $f[v]$ 则是当 v 被完全访问过之后，计数器的值，即 $dfs_visit(v)$ 之后计数器的值。如果原图是不连通的，那么根据 $pred[]$ 的值实际上可以得到一个由深度优先树组成的深度优先森林。对于森林中的树来说，它们根节点的 $pred[r]$ 的值都为 -1 。

假设

此算法对于有向图和无向图同样适用。

使用环境

深度优先搜索只需要在遍历图时存储每个顶点的颜色（白、灰或者黑）。因此深度优先搜索在遍历图时，只需要很小的存储开销。

深度优先搜索能够在数组中存储其遍历图时的信息。事实上，深度优先搜索对图的唯一要求是能够遍历给定顶点的所有邻接顶点，于是这样，我们能够在复杂的数据结构上方便地进行深度优先搜索。因为 dfs_visit 函数将原图看做一个只读结构。但是，深度优先搜索仅仅依靠当前的信息，是一种盲目的搜索，它并没有一个明智的计划来快速达到目标顶点 t 。

解决方案

例6-1是深度优先搜索的C++的实现。注意顶点的颜色信息只是在 dfs_search 和 dfs_visit 中使用。

例6-1：深度优先搜索实现

```
#include "dfs.h"
```

```
// 访问顶点u，更新信息
```

```
void dfs_visit (Graph const &graph, int u,           /* 输入*/  
               vector<int> &d, vector<int> &f,       /* 输出 */  
               vector<int> &pred, vector<vertexColor> &color, /* 输出 */  
               int &ctr, list<EdgeLabel> &labels) {   /* 输出 */  
    color[u] = Gray;  
    d[u] = ++ctr;
```

```

//处理u的所有邻接顶点
for (VertexList::const_iterator ci = graph.begin(u);
     ci != graph.end(u); ++ci) {
    int v = ci->first;

    // 计算edgeType并且将其加到标注中。
    edgeType type = Cross;
    if (color[v] == White) { type = Tree; }
    else if (color[v] == Gray) { type = Backward; }
    else { if (d[u] < d[v]) type = Forward; }
    labels.push_back(EdgeLabel (u, v, type));

    // 遍历未访问顶点，并且记录pred[]。
    // 一旦递归结束，回溯到邻近顶点。
    if (color[v] == White) {
        pred[v] = u;
        dfs_visit (graph, v, d, f, pred, color, ctr, labels);
    }
}

color[u] = Black; // 在完成对u的邻接顶点搜索之后，将u置为完成态。
f[u] = ++ctr;
}

/**
 * 从s开始进行深度优先搜索，然后计算d[u]（当u第一次访问时），
 * f[u]（当u的所有邻接顶点都被搜索过之后）
 * pred[u]（在深度优先搜索树中u的前驱顶点），以及标注的边。
 */
void dfs_search (Graph const &graph, int s,          /* 输入 */
                vector<int> &d, vector<int> &f,      /* 输出 */
                vector<int> &pred, list<EdgeLabel> &labels) /* 输出 */
{
    // 初始化d[], f[]和pred[]数组。将所有的顶点标记为白色，表示未访问。清除所有边上的标注。
    int ctr = 0;
    const int n = graph.numVertices();
    vector<vertexColor> color (n, White);
    d.assign(n, -1);
    f.assign(n, -1);
    pred.assign(n, -1);
    labels.clear();

    // 从源点开始搜索，结束时，搜索任意未被访问过的顶点。
    dfs_visit (graph, s, d, f, pred, color, ctr, labels);
    for (int u = 0; u < n; u++) {
        if (color[u] == White) {
            dfs_visit (graph, u, d, f, pred, color, ctr, labels);
        }
    }
}

```

如果d[]和f[]不需要，那么这些值的计算过程（以及将其作为函数参数）可以从例6-1中的代码删除掉。深度优先搜索可以得到关于图中边的额外信息。尤其，在深度优先森林中，有四种边。

树边

对于所有 $\text{pred}[v]=u$ 的顶点 v ， $\text{dfs_visit}(u)$ 访问边 (u,v) 来遍历图。这些边记录了深度优先搜索的进程。图6-10中的边 $(s,1)$ 就是一个很好的例子。

后边

当 $\text{dfs_visit}(u)$ 处理到顶点 v 时，如果 v 是 u 的邻接顶点并且是灰色，那么深度优先搜索就会知道它已经访问过这个顶点。图6-10中的边 $(8,s)$ 就是一个很好的例子。

前边

当 $\text{dfs_visit}(u)$ 处理到顶点 v 时，如果 v 是 u 的邻接顶点， v 的颜色是黑色，而且 u 在 v 之前被访问，那么边 (u,v) 是一个前边。那么深度优先搜索就会知道它已经访问过这个顶点了。图6-10中的边 $(5,9)$ 就是一个很好的例子。

交叉边

当 $\text{dfs_visit}(u)$ 处理到顶点 v 时，如果 v 是 u 的邻接顶点， v 的颜色是黑色，而且 u 在 v 之后被访问，那么边 (u,v) 是一个交叉边。交叉边只是在有向图中存在。

标记这些边的代码在例6-1中。对于无向图，边 (u,v) 可能会被标记很多次，但是一般来说，只有在这条边第一次被标记时其标记有效。

分析

图中的每一个顶点都会调用递归的 dfs_visit 函数一次。 dfs_search 中的循环不会执行超过 n 次。在 dfs_visit 函数中，每一个邻接顶点都要被检查，对于有向图来说，每一条边都只会遍历一次，然而在无向图中，它们会被遍历一次然后会被检查一次。在任何情况下，性能开销都是 $O(V+E)$ 。

广度优先搜索

广度优先搜索（图6-11）使用了和深度优先搜索一种不同的方法来搜索图。广度优先搜索系统地搜索顶点，在搜索离源点 s 的距离为 $k+1$ 条边的顶点之前，所有的离源点 s 距离为 k 条边的顶点都已经被访问。广度优先搜索不会访问图中那些源点不可达的顶点。

广度优先搜索不会进行任何回溯。我们可以通过查看顶点的颜色来查看进度，就像深度优先搜索那样。事实上，在广度优先搜索中，我们使用同样的颜色和定义。为了和深度优先搜索进行直接对比，我们使用一个类似的计数器，用来记录顶点第一次访问和最后一次访问的时刻。如图6-8所示，在相同的时间（例如，当计数器达到18时），广度优先搜索能够前进到图6-12的状态，即顶点12被标记为灰色。注意深度优先搜索已经处理顶点 $\{1, 6, 8\}$ ，这些顶点都是离源点 s 一条边的距离，以及顶点 $\{2, 3\}$ ，它们离源点 s 两条

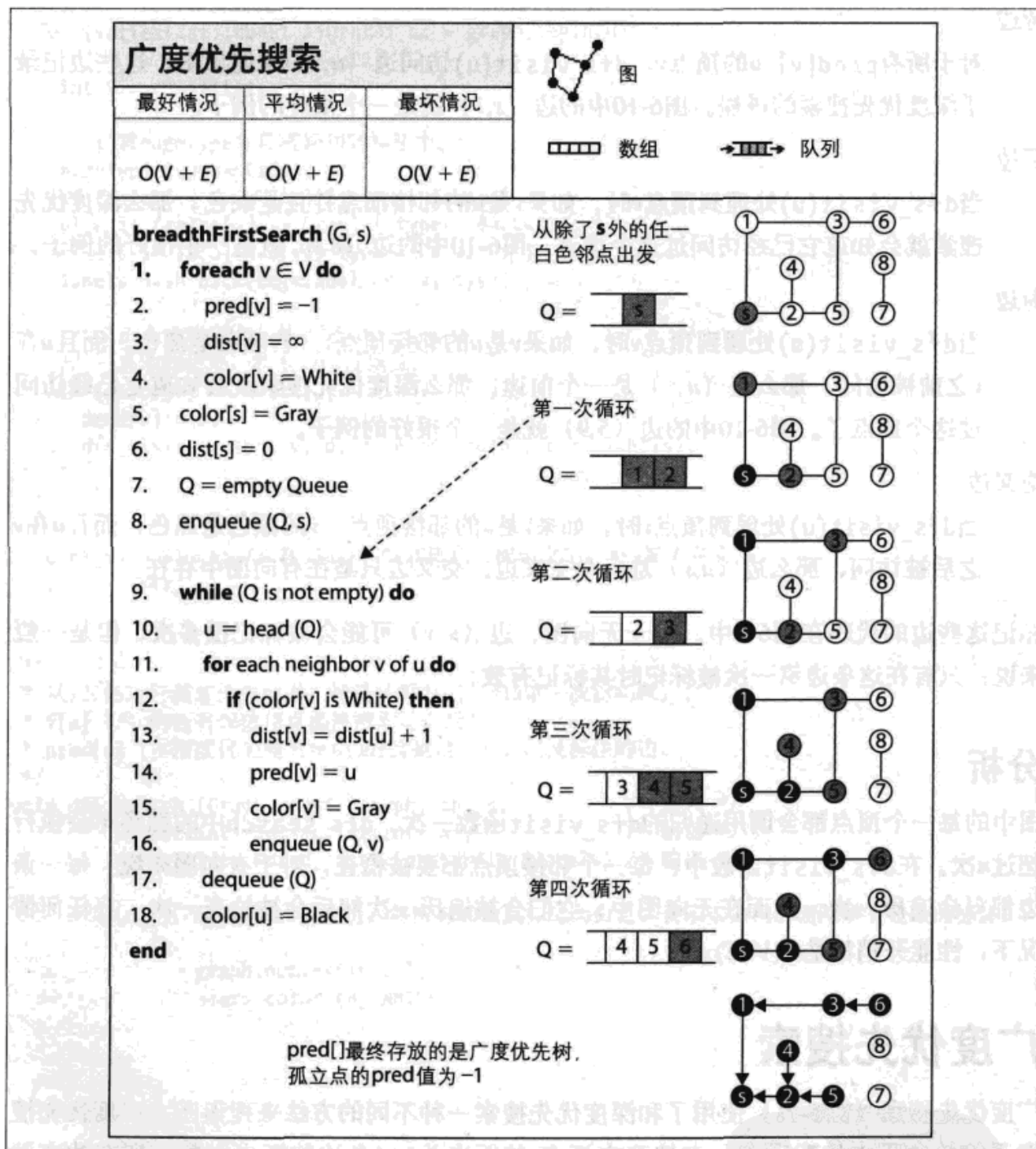


图6-11: 广度优先搜索详解

边的距离，还有顶点{7, 14, 5}，它们在队列中等待被处理。有些顶点离源点 s 三条边距离，例如{11, 10, 12, 4}，已经被访问过了，虽然广度优先搜索还没有处理这些顶点。注意队列中的所有顶点都被染成灰色，表示它们现在的状态。

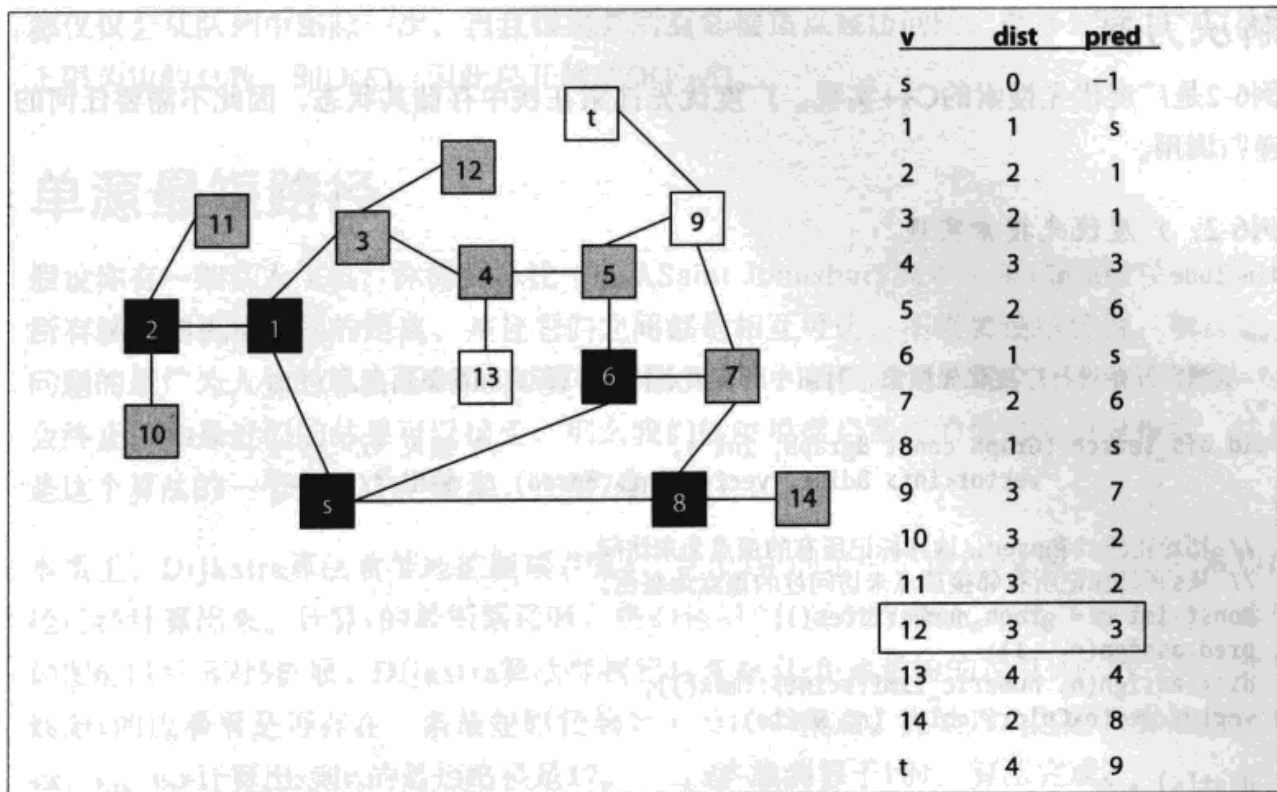


图6-12：当计数器达到18时，广度优先搜索在图中的进程

输入/输出

输入

图 $G=(V,E)$ 以及一个源点 $s \in V$ 。 n 表示 G 中的顶点数。

输出

广度优先搜索得到了两个数组。 $\text{dist}[v]$ 是从 s 到 v 的最短路径上的边数。 $\text{pred}[v]$ 是在广度优先搜索中 v 的前驱顶点。根据 $\text{pred}[]$ 的值，我们可以得到广度优先树，如果原始图是非连通的，那么对于所有源点不可达的顶点 w ，其 $\text{pred}[w]$ 值是 -1 。

假设

此算法适用于有向图和无向图。

使用环境

广度优先搜索存储队列中那些处于“活跃态”的顶点，因此对于一个大图来说，可能需要一个相当大的存储器空间。广度优先搜索保证能够为图中的顶点寻找到最短路径。事实上，广度优先树中的所有路都是从源点 s 的最短路径（以边数来说）。

解决方案

例6-2是广度优先搜索的C++实现。广度优先搜索在栈中存储其状态，因此不需要任何的递归调用。

例6-2：广度优先搜索实现

```
#include "bfs.h"

/**
 * 从源点开始进行广度优先搜索，对图中所有顶点计算BFS距离和前驱顶点。
 */
void bfs_search (Graph const &graph, int s,           /* 输入 */
                 vector<int> &dist, vector<int> &pred) /* 输出 */
{
    // 初始化dist和pred，这样标记所有的顶点为未访问。
    // 从s开始标记所有邻接顶点未访问过的顶点为灰色。
    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();

        // 访问u的邻接顶点。
        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;
    }
}
```

分析

在初始化时，广度优先搜索会对所有顶点的信息进行更新，因此初始化的开销是 $O(V)$ 。当一个顶点第一次被访问（染为灰色）时，将它放入队列，顶点不会重复添加。因为队列能够在常数时间添加或者删除元素，所以管理队列的开销是 $O(V)$ 。最后，每一个顶点

都仅仅会从队列中删除一次，当且仅当其所有邻接顶点被访问时。循环的上限为边的总数，即 $O(E)$ 。因此总开销是 $O(V+E)$ 。

单源最短路径

假设你有一架私人飞机，你需要寻找一条从Saint Johnsbury到Waco的最短路径。你知道所有城市的机场之间的距离，并且它们之间都是相互可达，不需要途中停留。解决这个问题的最广为人知的算法是Dijkstra算法（图6-13）。找到最短路径之后，这个搜索可能会终止。如果近似的结果可以接受，那么我们能够根据启发式的信息来引导搜索，这就是这个算法的一个变种（A*搜索，在第7章讨论）。

本质上，Dijkstra算法贪婪地扩展顶点集 S ，对于 S 中的每一个顶点 v 来说， s 到 v 的最短路径已经计算出来。计算 v 的最短路径时，我们使用 S 中顶点的路径。 S 最开始为 $\{s\}$ 。算法如图6-14所示对 S 扩展，Dijkstra算法寻找到距离源点 s 距离最短的顶点 $v(v \in V-S)$ ，然后顺着 v 的边看看是否存在一条最短路径到达另外一个顶点。在处理 v_2 之后，算法通过路 $\langle s, v_2, v_3 \rangle$ 计算出 s 到 v_3 的最短路径是17。一旦 S 扩展到等于 V 时，算法完成。

输入/输出

输入

一个有向有权图 $G=(V,E)$ 以及一个源点 $s \in V$ 。图中每一条边 $e=(u,v)$ 都有一个相关的正权值。 n 是图 G 中的顶点数。

输出

Dijkstra算法得到两个数组。主要的计算结果是数组 $\text{dist}[]$ ，其值表示 s 到图中每个顶点的距离。注意 $\text{dist}[s]$ 等于0。次要的计算结果是数组 $\text{pred}[]$ ，这个数组能够用来构造从源点 s 到图中每个顶点的最短路径。你需要重新审视一下例6-3，看看 pred 是如何更新的。

假设

边的权值是正数，如果这个假设不是真的，那么 $\text{dist}[u]$ 的值有可能会是无效的。更坏的是，如果图中存在一个负值环，Dijkstra算法可能会陷入死循环。在图6-13的第12行中，我们假设不会有算术溢出，你必须注意，简单地添加一个对 $\text{newLen} \geq 0$ 的检查。

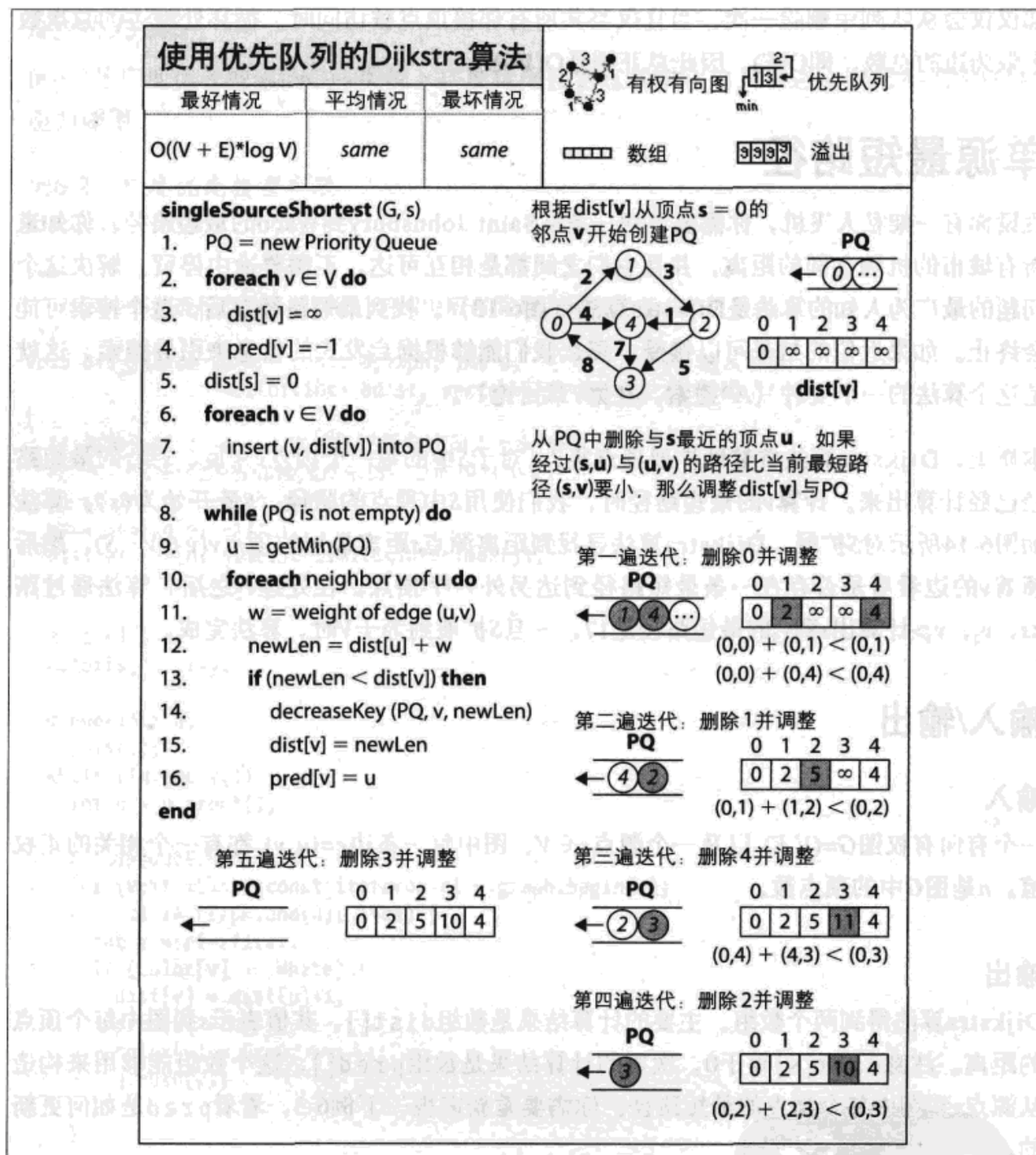


图6-13: 使用优先队列的Dijkstra算法详解

解决方案

Dijkstra算法执行时， $\text{dist}[v]$ 表示的是根据 S 集中的顶点组成的子图，我们能够找到的从源点 s 到 v 的最短路径的最大长度。并且，对于每一个顶点 $v \in S$ ， $\text{dist}[v]$ 为整数。幸运的是，Dijkstra算法不需要真正地计算和存储集合 S 。它只初始创建一个包含所有 V 中的顶点的集合，然后每次从集合中拿出一个顶点，计算出正确的 $\text{dist}[v]$ 值，为了方便

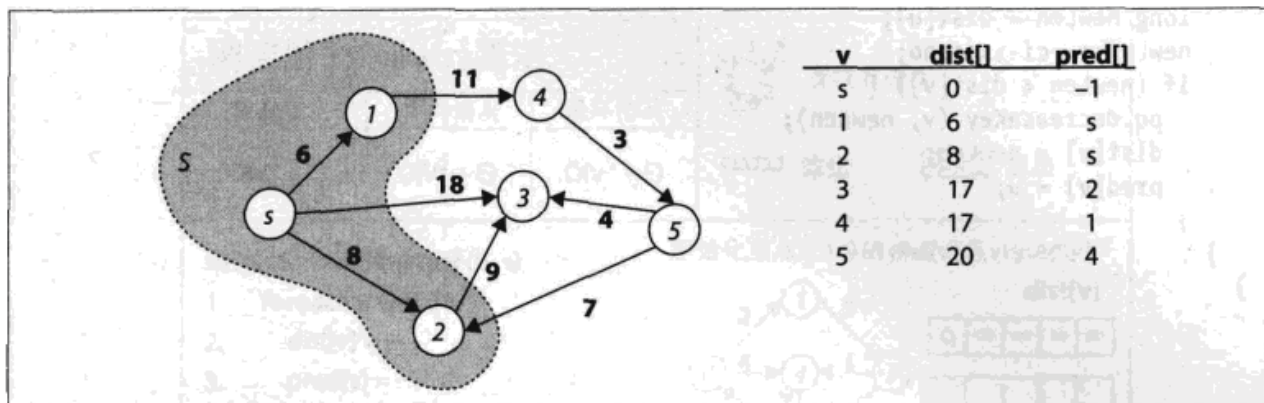


图6-14: Dijkstra算法扩展S集

起见，我们使用 $V-S$ 来记录这个集合。当所有的顶点都被访问或者为被访问的顶点都是源点不可达时，Dijkstra算法结束。

在例6-3的C++实现中，我们使用了优先队列来存储 $V-S$ 中的顶点。在这里优先队列是二叉堆，因为我们可以常数时间寻找到优先级最小的顶点（优先级是顶点到 s 的距离）。此外，当 s 到 v 的最短路径找到之后， $\text{dist}[v]$ 的值减少，那么堆也必须修改。幸运的是，二叉堆形式的优先队列 decreaseKey 操作在平均情况下花费时间为 $O(\log q)$ ， q 是二叉堆中顶点的数目， q 永远小于或者等于顶点的总数 n 。

例6-3: 使用优先队列的Dijkstra算法实现

```
#include "BinaryHeap.h"
#include "Graph.h"

/** 给定一个有向有权图，计算所有顶点的最短路径（dist数组），
 * 记录前驱节点的指针（pred数组）。 */
void singleSourceShortest(Graph const &g, int s,                /* 输入 */
                          vector<int> &dist, vector<int> &pred) { /* 输出 */
    // 初始化dist[]和pred[]数组。从顶点s开始，将dist[]值设为0。
    // 优先队列PQ包含G中所有的顶点v。
    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]); }

    // 在V-S中寻找dist最小的顶点。
    // 寻找新的路径，然后更新所有最短路径。
    while (!pq.isEmpty()) {
        int u = pq.smallest();

        // 遍历u的所有邻接节点，如果新的路径newLen（从s->u的最佳路径+边（u, v）的权重）
        // 比已知s->v的最佳路径要好。
        // 那么更新dist[v]，并且重新调整二叉堆。使用长整型避免溢出错误。
        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;
        }
    }
}
}

```

结论

如果边权值之和超过了`numeric_limits<int>::max()`，那么会产生溢出。为了避免这种情况，`newLen`使用了长整型。

分析

在例6-3的Dijkstra算法实现中，构造初始优先队列的循环将会执行 V 次插入操作，将会花费 $O(V \log V)$ 的时间。在剩下的while循环中，每条边都会被访问一次，因此`decreaseKey`操作执行不会超过 E 次，花费 $O(E \log V)$ 的时间。因此，总开销是 $O((V + E) \log V)$ 。

图6-15详解描述的是Dijkstra算法的一个版本，这个版本适用于稠密图，使用的是一个邻接矩阵。例6-4中的C++实现更加简单，因为它避免了使用二叉堆。这个版本的效率是由如何快速在 $V-S$ 中得到最小的`dist[]`。while循环将会执行 n 次，因为 S 每次只会增加一个顶点。在 $V-S$ 中得到最小的`dist[u]`将要探测 n 个顶点。注意在while循环的内循环中，每条边都要被检测一次。因此，总运行时间为 $O(V^2 + E)$ 。

例6-4：稠密图的Dijkstra算法实现

```

#include "Graph.h"
void singleSourceShortest(Graph const &graph, int s,           /* 输入*/
                          vector<int> &dist, vector<int> &pred) { /* 输出*/

    // 初始化dist[]和pred[]数组。从顶点s开始，将dist[]设置为0。
    const int n = graph.numVertices();
    pred.assign(n, -1);
    dist.assign(n, numeric_limits<int>::max());
    vector<bool> visited(n);
    dist[s] = 0;

    // 在V-S中寻找顶点。
    // 计算所有可能的新路径，更新最短路径。
    while (true) {
        // 在未访问顶点中寻找最短距离。
        int u = -1;
        int sd = numeric_limits<int>::max(); // 假设不可达。
    }
}

```

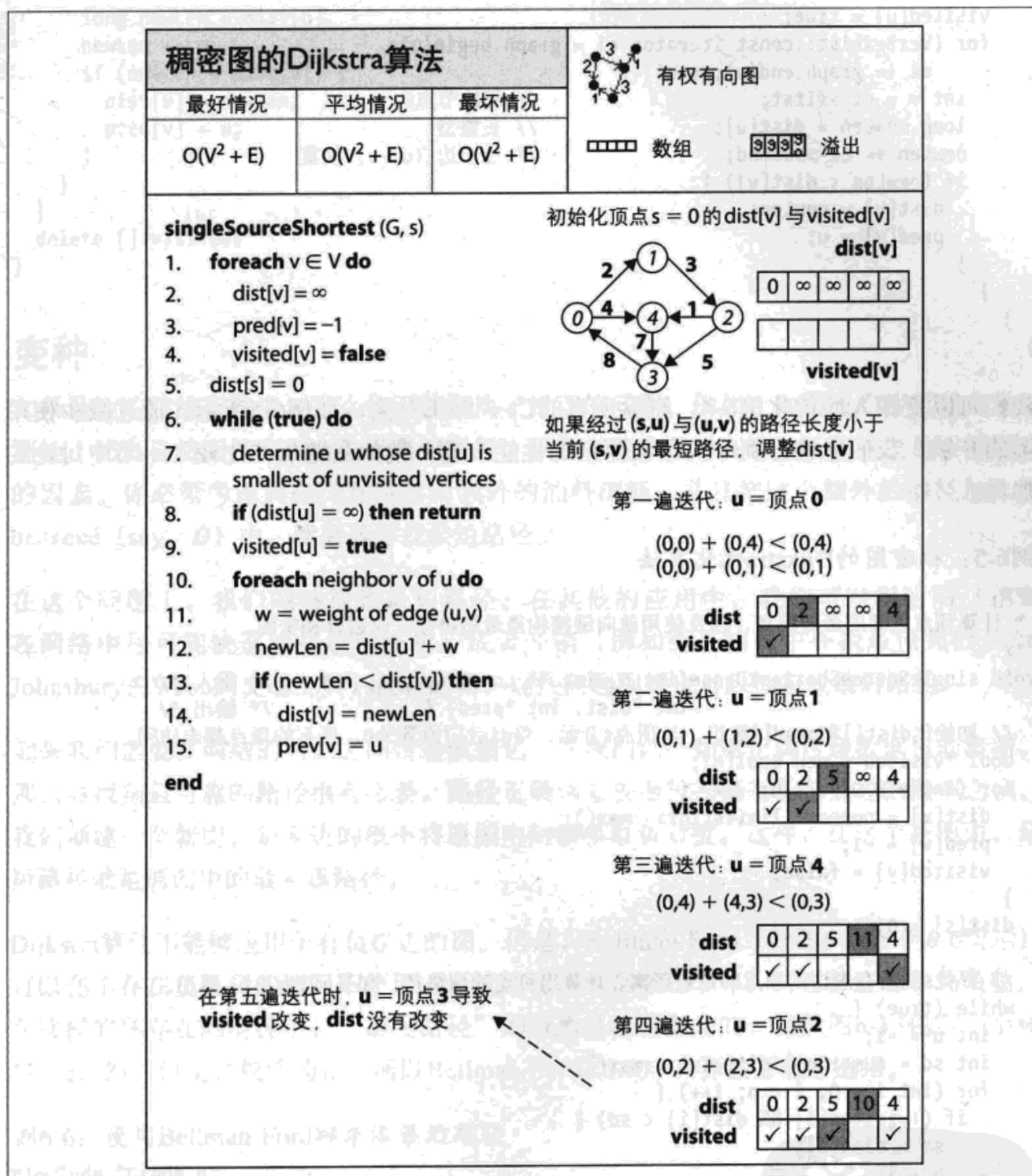


图6-15: 稠密图的Dijkstra算法详解

```

for (int i = 0; i < n; i++) {
    if (!visited[i] && dist[i] < sd) {
        sd = dist[i];
        u = i;
    }
}
if (u == -1) { break; }    // 不需要进行下一步处理。

// 遍历u的所有邻接节点，检查新的路径newLen（从s->u的最佳路径+边（u，v）的权重）
// 是否比已知s->v的最佳路径要好。

```

```

visited[u] = true;
for (VertexList::const_iterator ci = graph.begin(u);
     ci != graph.end(u); ++ci) {
    int v = ci->first;           // 邻接节点v
    long newLen = dist[u];      // 长整型
    newLen += ci->second;        // 加上边 (u, v) 权重
    if (newLen < dist[v]) {
        dist[v] = newLen;
        pred[v] = u;
    }
}
}
}
}

```

我们可以更深入地优化例6-4，删除所有的C++ STL对象，如例6-5所示。通过减少使用类的开销，我们能够得到令人印象深刻的性能改进，就如在随后“比较”一节中讨论的那样。

例6-5：稠密图的Dijkstra优化算法

```

/**
 * 计算顶点之间的最短距离，以及使用前向链接构造最短路径。仅适合稠密图。
 */
void singleSourceShortestDense(int n, int ** const weight, int s, /* 输入 */
                               int *dist, int *pred) { /* 输出 */
    // 初始化dist[]和pred[]数组。从顶点s开始，将dist[]设置为0。所有的顶点都未访问。
    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;

    // 寻找s到所有未访问顶点的最短距离。计算出可能的新路径，如果u仍为-1，退出。
    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; }

        // 遍历u的所有邻接节点，检查新的路径newLen（从s->u的最佳路径+边 (u, v) 的权重）
        // 是否比已知s->v的最佳路径要好。使用长整型计算。
        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;
}

```

变种

如果最短的航线有许多，那么你可能解决了错误的问题。例如，如果我们希望减少油料消耗，那么自然而然我们会选择最短路径。但是在着陆和起飞时油料消耗也是必须考虑的因素。你必须考虑到起飞和降落时额外的油料消耗，并且把这个额外的消耗加入到 distance (say, D) 中。然后再寻找最短路径。

在这个问题上，我们能够找到最短路径。在其他的应用中，我们可以用时间（例如在网络中尽可能快速地投递网络包）或者开销（例如商务旅行中寻找最便宜的从 St. Johnsbury 去 Waco 的交通工具）代替距离。这些问题同样也可以抽象成最短路径。

如果我们需要在网络的两点之间传递数据包，在我们已经知道正确传递数据包的概率，那么寻找到最可靠的路径很有必要。路径正确传递数据包的概率是每条边的概率之积。我们创建一个新图，每条边的概率将原图中的概率取负对数。这样，在这个新图中，最短路径就是原图中的最可靠路径。

Dijkstra 算法不能够应用于有负权边的图。但是，Bellman-Ford（如图 6-16 和例 6-6 所示）可以在不存在负值环的情况下使用。也就是说，不存在一个环，其边权值总和为负数。在这样的环存在的情况下，“最短路径”的概念是没有意义的。虽然图 6-13 包含一个环 {1, 3, 2}，但是边权值为正，所以 Bellman-Ford 和 Dijkstra 算法都能够适用。

例 6-6：使用 Bellman-Ford 解单源最短路径

```

#include "Graph.h"
/**
 * 给定有向有权图，计算到所有顶点的最短距离，并且记录前向节点构造最短路径。
 */
void singleSourceShortest(Graph const &graph, int s,          /* 输入 */
                          vector<int> &dist, vector<int> &pred){ /* 输出 */
    //初始化dist[]和pred[]数组。
    const int n = graph.numVertices();
    pred.assign(n, -1);
    dist.assign(n, numeric_limits<int>::max());
    dist[s] = 0;
}

```

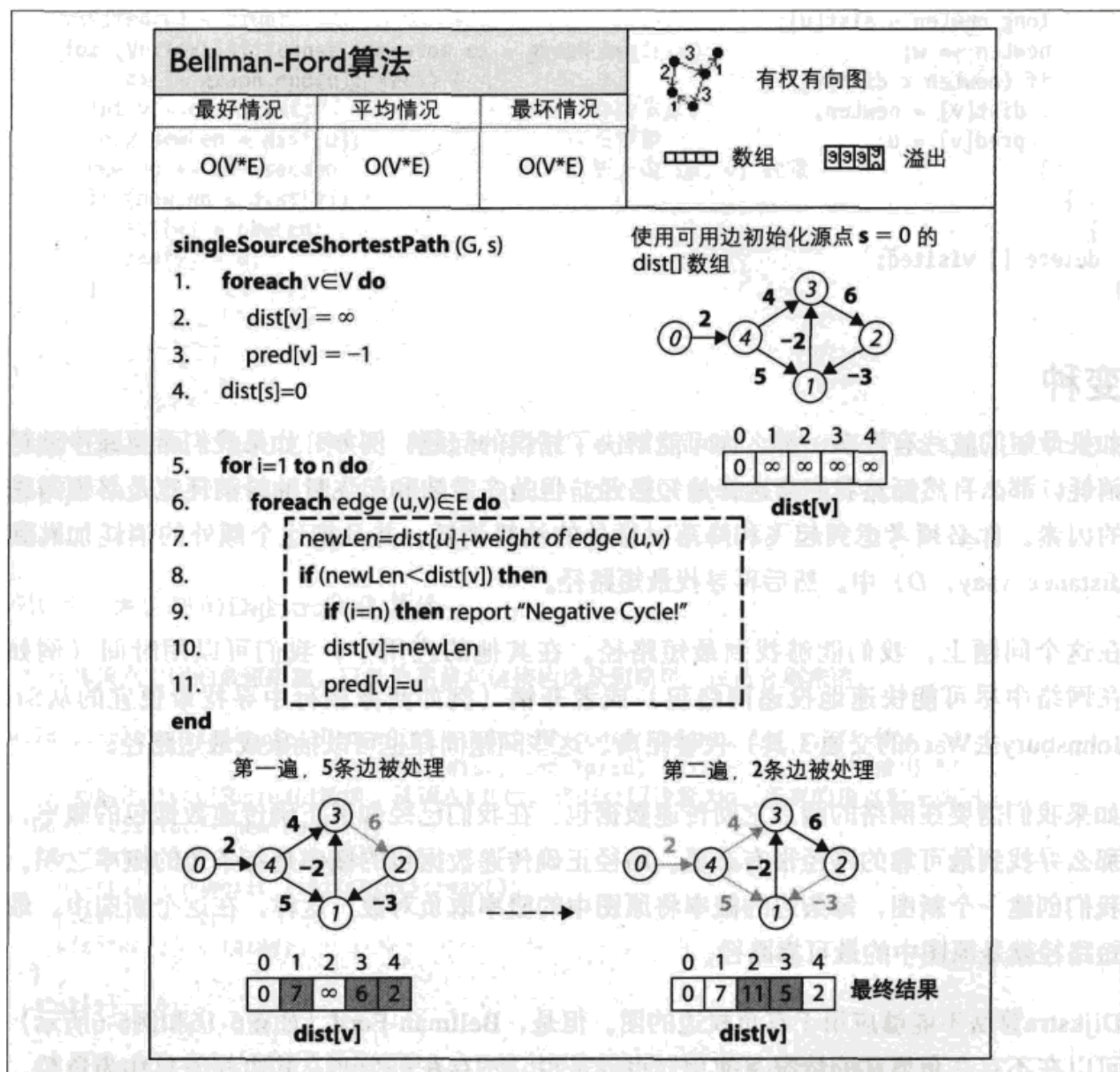


图6-16: Bellman-Ford详解

// n-1次之后，我们能够保证s到其他所有顶点的最短距离已经计算出来。

// 在第n次计算时，如果有值改变了，那么就说明有负环。

// 如果值没有任何改变，将会很早退出。

```
for (int i = 1; i <= n; i++) {
    bool failOnUpdate = (i == n);
    bool leaveEarly = true;
```

// 处理每个顶点u，检查某条边 (u, v) 是否能够构建出一条 $s \rightarrow u \rightarrow v$ 的路径。

// 这条路径的距离小于已知的 $s \rightarrow v$ 最短路径。使用长整型避免溢出。

```
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 "Graph has negative cycle"; }
        dist[v] = newLen;
        pred[v] = u;
        leaveEarly = false;
    }
}
}
if (leaveEarly) { break; }
}
}

```

Bellman-Ford算法很直观地在图中推进 n 次，寻找是否存在一条边 (u,v) ，能够在给定 $dist[u]$ 和边 (u,v) 的权值下，更新 $dist[v]$ 。我们需要至少 $n-1$ 次推进，例如，在极端情况下，从源点 s 到某个顶点 v 的最短路径需要通过图中所有的顶点。另外，因为边可以以任意顺序访问，使用 $n-1$ 次推进就能够确保所有的缩减路能够被发现。事实上，以不同的顺序访问边将会导致一个不同的计算 $dist[]$ 值过程。图6-17所示的是在两个不同的图上执行Bellman-Ford算法，这两个图的区别在于标记为 v_4 和 v_1 的顶点。但是如果你考虑重标记的话，两个执行过程都能够得到同样的结果。

如果存在负环，Bellman-Ford是不能正常工作的。为了检测负环，我们执行一个循环 n 次（不止一次），如果有调整 $dist$ 的值，那么就存在负环。从嵌入的循环中可以很明显地看出来，Bellman-Ford的性能是 $O(V * E)$ 。

比较

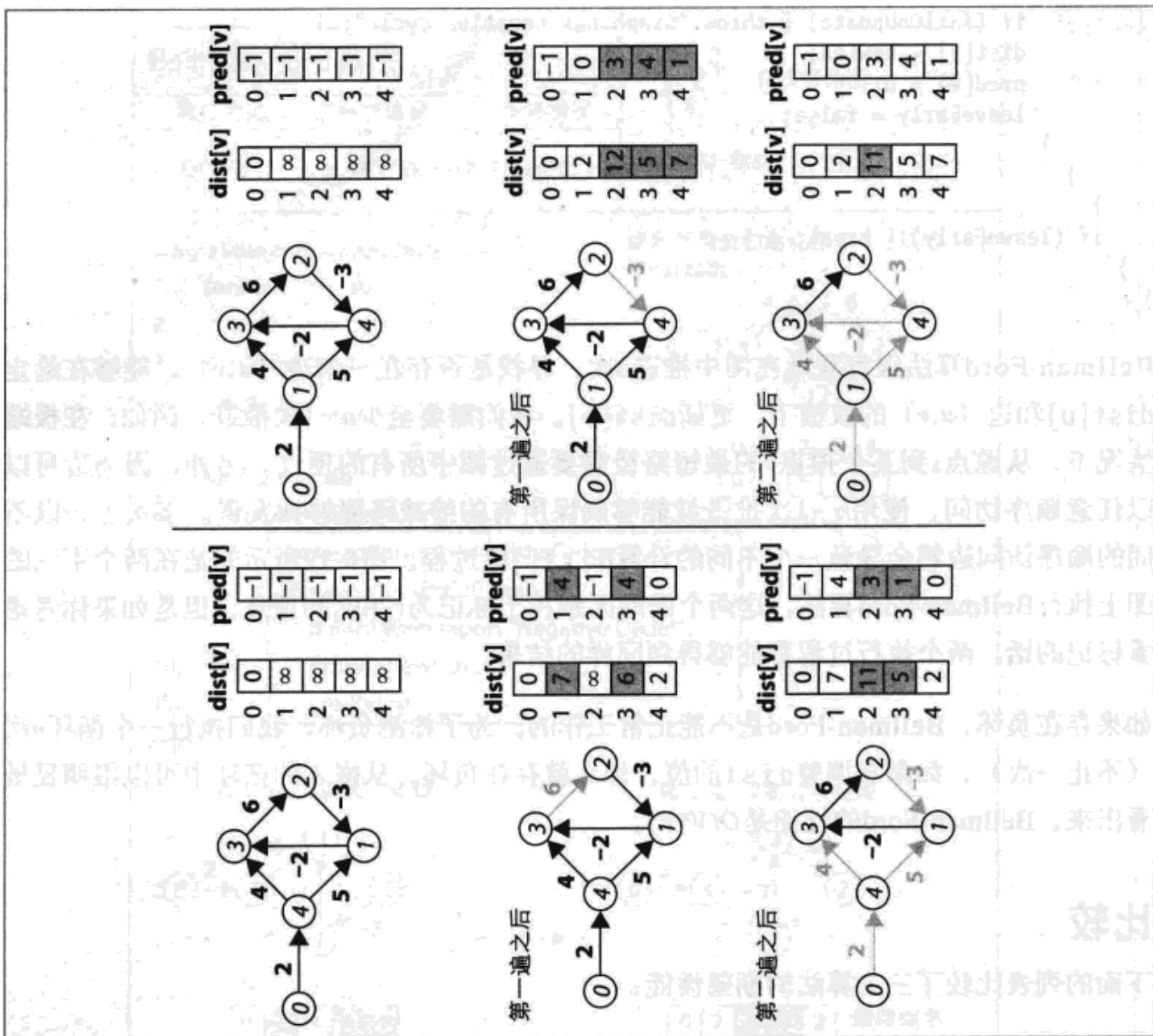
下面的列表比较了三个算法的期望性能：

- Bellman-Ford: $O(V * E)$
- 在稠密图上的Dijkstra算法: $O(V^2 + E)$
- 使用优先队列的Dijkstra算法: $O((V + E) * \log V)$

我们在不同的情况下比较这些算法。当然，为了选择最适合你手上数据的算法，你也可以像我们这样进行基准评测。我们将会执行算法10次，然后抛弃最好和最坏的结果，表格中的结果是剩余8次的平均值。

基准测试数据

产生一个“随机图”是非常困难的。在表6-2中，我们使用的是有 $V = k^2 + 2$ 个顶点，以及 $E = k^3 - k^2 + 2k$ 条边的图（图的构造细节参见代码库中的实现）。注意边的数目大概是 $n^{1.5}$ ， n 是顶点的数目。使用了优先队列的Dijkstra算法的性能最好，Bellman-Ford的性能紧随其后。我们可以看到变种在稠密图上并没有多大性能改进。



稠密图

在稠密图中, E 大约是 $O(V^2)$; 例如, 在一个有着 n 个顶点的完全图中, 存在 $n(n-1)/2$ 条边。我们不推荐在这样的稠密图上使用 Bellman-Ford, 因为这样的性能将会退化至 $O(V^3)$ 。表6-3的稠密图是来自于研究人员研究旅行商问题 (TSP) (注1) 时使用的数据。我们进行了100次实验, 抛弃最好和最坏的结果, 表中的结果是剩余98次的平均值。虽然在优先队列和稠密图版本的Dijkstra算法之间存在一点点差异, 但是优化过后的Dijkstra算法的性能得到了非常大的改进, 如表中的第5列所示。在最后一列中, 我们给出了在同样的问题上, Bellman-Ford算法的性能, 但是结果只是五次执行结果的平均值, 因为性能退化得太厉害了。从最后一列我们知道, Bellman-Ford算法在小图上的绝对性能看起来还是合理的, 但是相比其他算法, 在稠密图上使用这个算法就是一个错误。

表6-3: 在稠密图上计算单源最短路径的时间 (单位: 秒)

V	E	使用优先队列 的Dijkstra 算法	稠密图上 的Dijkstra 算法	优化过的 稠密图 Dijkstra算法	Bellman-Ford 算法
980	479,710	0.0918	0.1147	0.0128	0.2444
1,621	1,313,010	0.2194	0.2601	0.0329	0.7978
6,117	18,705,786	3.6256	4.0361	0.2301	66.0659
7,663	29,356,953	8.3147	8.8592	0.3644	222.3107
9,847	48,476,781	15.2602	16.2169	0.6116	431.2807
9,882	48,822,021	14.7536	16.5594	0.6224	277.8776

稀疏图

大图通常是稀疏的, 图6-4的结果确认了, 使用优先队列的Dijkstra算法更加适合稀疏图, 为稠密图设计的实现慢10倍左右。表格中的行依稀疏图中的边数排序, 因为这样可以明显地看出结果中的开销因子。

表6-4: 在大稀疏图上计算单源最短路径的时间 (单位: 秒)

V	E	密度	使用优先队列 的Dijkstra算法	稠密图上的 Dijkstra算法	优化过的稠密图 Dijkstra算法
3,403	137,845	2.8 %	0.0453	0.2038	0.098
3,243	294,276	1.2 %	0.017	0.1922	0.1074
19,780	674,195	0.17 %	0.1002	2.9697	1.805

注1: <http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/>.

所有点对最短路径

我们通常不是寻找单源最短路径（注2），而是在任意两个顶点 (v_i, v_j) 之间寻找最短路径。最快解决这个问题的是使用一个强有力的方法——动态规划。

动态规划有两个令人激动的特征：

- 它专注解决较小的，有条件限制的问题。当限制条件很严格时，函数将会非常简单，系统地松弛限制条件，直到最后产生希望得到的结果。
- 虽然我们希望为问题寻找到最优解，计算最优解的值比寻找什么是最优解要简单许多。在我们的例子中，计算每个点对 (v_i, v_j) 之间的最短距离，然后进行一些额外的计算得到实际的路径。

给定一个 $n \times n$ 的 dist 矩阵，我们计算得到，对于所有的顶点对 (v_i, v_j) 来说， $\text{dist}[i][j]$ 是 v_i 到 v_j 的最短路径。图6-18是Floyd-Warshall的伪代码，以及它在一个小例子上的执行过程。

输入/输出

输入

一个有向有权图 $G=(V,E)$ 。每条边 $e=(u,v)$ 都有一个正权值。图 G 的顶点数为 n 。

输出

Floyd-Warshall算法得到一个矩阵 $\text{dist}[][]$ ， $\text{dist}[u][v]$ 的值表示从 u 到 v 的最短路径。注意如果 $\text{dist}[u][v]$ 是 ∞ ，那么从 u 到 v 之间不存在路径。在两个顶点之间的实际路径可以从矩阵 $\text{pred}[][]$ 推导出来，当然， $\text{pred}[][]$ 也是这个算法计算的结果。

假设

边的权值必须为正值（例如必须大于0）。

解决方案

动态规划方法将会依序计算 $\text{dist}_k[i][j]$ ， $0 \leq k \leq n$ ，得到经过顶点 v_1, v_2, \dots, v_k 的从 v_i 到 v_j 的最短路径， $\text{dist}_k[i][j]$ （注3）。例如，当 $k=0$ 时， $\text{dist}_0[i][j]$ 将会是边

注2：可能有多条有相同长度的路径。

注3：这些顶点并不需要区分开；也就是说， i 可能等于 j 或者 $1 \leq i \leq k$ 或者 $1 \leq j \leq k$ 。

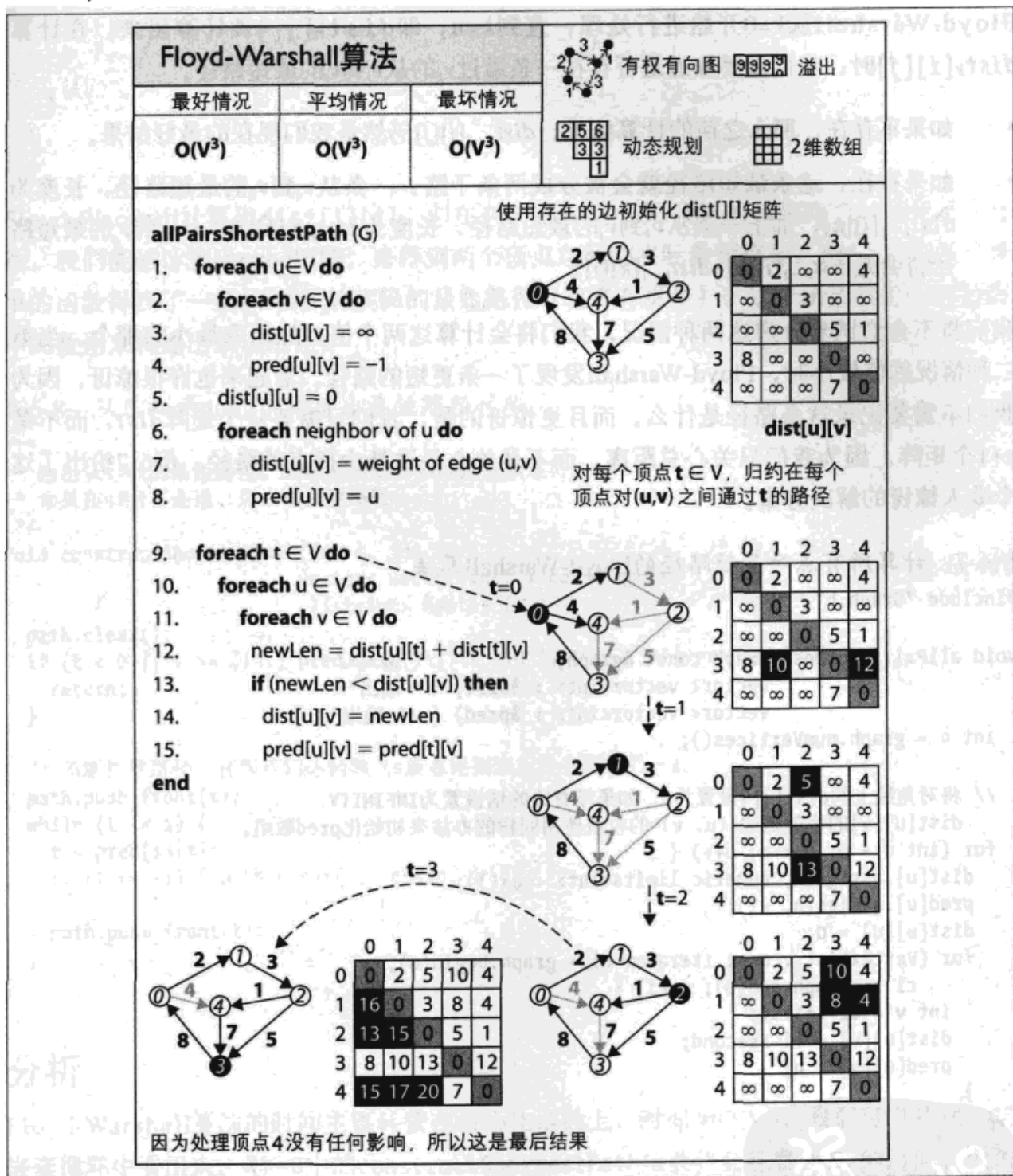


图6-18: Floyd-Warshall详解

(i,j) 的权值, 如果不存在这样一条边的话, 那么值为 ∞ 。当 $k=1$ 时, 我们将会检查所有的 i 和 j , 是否存在两条边 (v_i, v_1) 和 (v_1, v_j) , 其和小于边 (v_i, v_j) 。如果我们继续这个逻辑, 直到 $k=n$, 这时我们得到 v_i 到 v_j 的最终最短路径。

对于 $k>0$, 我们假设计算 dist_k 时, dist_{k-1} 已经被计算出来 (在动态规划中, 我们需要以一种正确的方法解决这些子问题)。

Floyd-Warshall从 $k=0$ 开始进行处理,直到 $k=n$,即 $dist_n[][]$ 被计算出来。在计算 $dist_k[i][j]$ 时,我们需要知道是否存在一条通过 v_k 的从 v_i 到 v_j 的最短路径。

- 如果不存在,那么之前的计算结果, $dist_{k-1}[i][j]$ 仍然是我们现在的最好结果。
- 如果存在,这条最短路径就会被分成两条子路:一条从 v_i 到 v_k 的最短路径,长度为 $dist_{k-1}[i][k]$,加上一条从 v_k 到 v_j 的最短路径,长度为 $dist_{k-1}[k][j]$ 。从 v_i 到 v_j 的最短路径将会是 $dist_{k-1}[i][k]+dist_{k-1}[k][j]$ 。

我们将不会尝试着分辨这两种情况,我们将会计算这两个值,然后取最小的那个。当第二种情况的值较小时,Floyd-Warshall发现了一条更短的路径。看起来也许很惊讶,因为我们不需要记录这条路径是什么。而且更惊讶的是,我们只需要一个矩阵 $dist$,而不是 $n+1$ 个矩阵。因为我们只关心总距离,而不是关心经过最少顶点的路径。例6-7给出了这个令人惊讶的解决方案。

例6-7: 计算所有点对最短路径的Floyd-Warshall算法

```
#include "Graph.h"

void allPairsShortest(Graph const &graph,          /* 输入*/
                      vector< vector<int> > &dist, /* 输出*/
                      vector< vector<int> > &pred) { /* 输出*/
    int n = graph.numVertices();

    // 将对角线上的dist[][]设置为0,如果没有边的话设置为INFINITY,
    // dist[u][v]的值就是边(u, v)的权值使用同样的办法来初始化pred数组。
    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; }

            // 如果能够找到一条减少距离的路径,更新dist[i][j]。
            // 使用长整型避免溢出。
            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];
    }
}
}
}
}

```

Floyd-Warshall计算出 $\text{dist}[i][j]$ ，即在有向有权图中从 v_i 到 v_j 的最短路径，按照这个方法，我们能够计算出 pred 矩阵，来得到两个顶点之间的实际最短路径。例6-8中的那个简单的函数得到了一条从 v_i 到 v_j 的实际的最短路径（也许有更多）。这个函数根据 pred 矩阵中的数据来构建出最短路径。

例6-8：从已知的 $\text{pred}[][]$ 构建出最短路径代码

```

/**
 * 输出从s到t的最短路径。s和t必须是有效地顶点描述符。
 * 如果在s和t不连通，那么返回空路径。
 */
void constructShortestPath(int s, int t, /* 输入 */
                           vector< vector<int> > const &pred, /* 输入 */
                           list<int> &path) { /* 输出 */

    path.clear();
    if (t < 0 || t >= (int) pred.size() || s < 0 || s >= (int) pred.size()) {
        return;
    }

    // 不断扩展路径，直到我们达到源点s或者得到表示没有路径的-1。
    path.push_front(t);
    while (t != s) {
        t = pred[s][t];
        if (t == -1) { path.clear(); return; }

        path.push_front(t);
    }
}

```

分析

Floyd-Warshall算法的时间主要耗费在最小化函数上，时间为 $O(V^3)$ ，我们可以从这三层嵌套循环中看出来。例6-8中的 $\text{constructShortestPath}$ 函数将会花费 $O(E)$ 的时间，因为最短路径可能会经过图中的每一条边。

最小生成树算法

给定一个无向连通图 $G=(V,E)$ ，我们可能会需要寻找一个边的子集 ST ，这个子集组成的图仍然是连通的。如果我们要求 ST 中的边的权值总和最小，那么我们就遇到这个问题：最小生成树（MST）。

图6-19所示的Prim算法告诉我们如何从一个图中构造一个MST，这个方法使用一种贪心的策略，每一步，这个算法都朝着解前进，但是不会推翻之前的决定。在扩展一棵树 T 时，Prim算法每次添加一条边，直到生成MST（可以证明这是最小的）。它随机选择一个开始顶点 $s \in V$ ，将 s 添加到一个集合 S 中，那么生成树 T 的根节点将会是 s 。Prim算法是贪心算法，每一次将一条边添加到 T 中，直到计算出MST。寻找一条权值最小的边 (u,v) ，并且 $u \in S$ ， $v \in V-S$ ，然后将这条边添加到MST中，将顶点 v 添加到集合 S 中。

算法使用优先队列来存储 $V-S$ ， $V-S$ 中的每个顶点 v 的优先级等于边 (u,v) 的最小值，其中 $u \in S$ 。这个精心的设计得到了一个高效的实现。

解决方案

例6-9的C++实现是基于二叉堆的优先队列。一般来说，使用二叉堆会降低实现的效率，因为主循环中检查是否一个顶点存在于优先队列（二叉堆不支持这个操作）将会很耗时间。但是，算法能够确保，顶点只会从优先队列中删除，所以我们只需要维护一个状态数组`inQueue[]`，这个数组在顶点从优先队列中取出时被更新。在另外一个优化实现版本中，我们维护一个外部数组`key[]`，记录队列中每一个顶点的优先级键值，这个值使得我们不需要从优先队列中寻找给定的顶点。

例6-9：基于二叉堆的Prim算法实现

```
/**
 * 给定无向图，从随机选择的一个顶点开始构造最小生成树。我们使用pred来记录MST。
 */
void mst_prim (Graph const &graph, vector<int> &pred) {
    // 初始化pred[]和key[]数组。优先队列PQ包含图G中所有的顶点。
    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;

        // 处理u的所有邻接节点，检查是否存在一条边能够更新最短路径。
        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);
    }
}
}
}

```

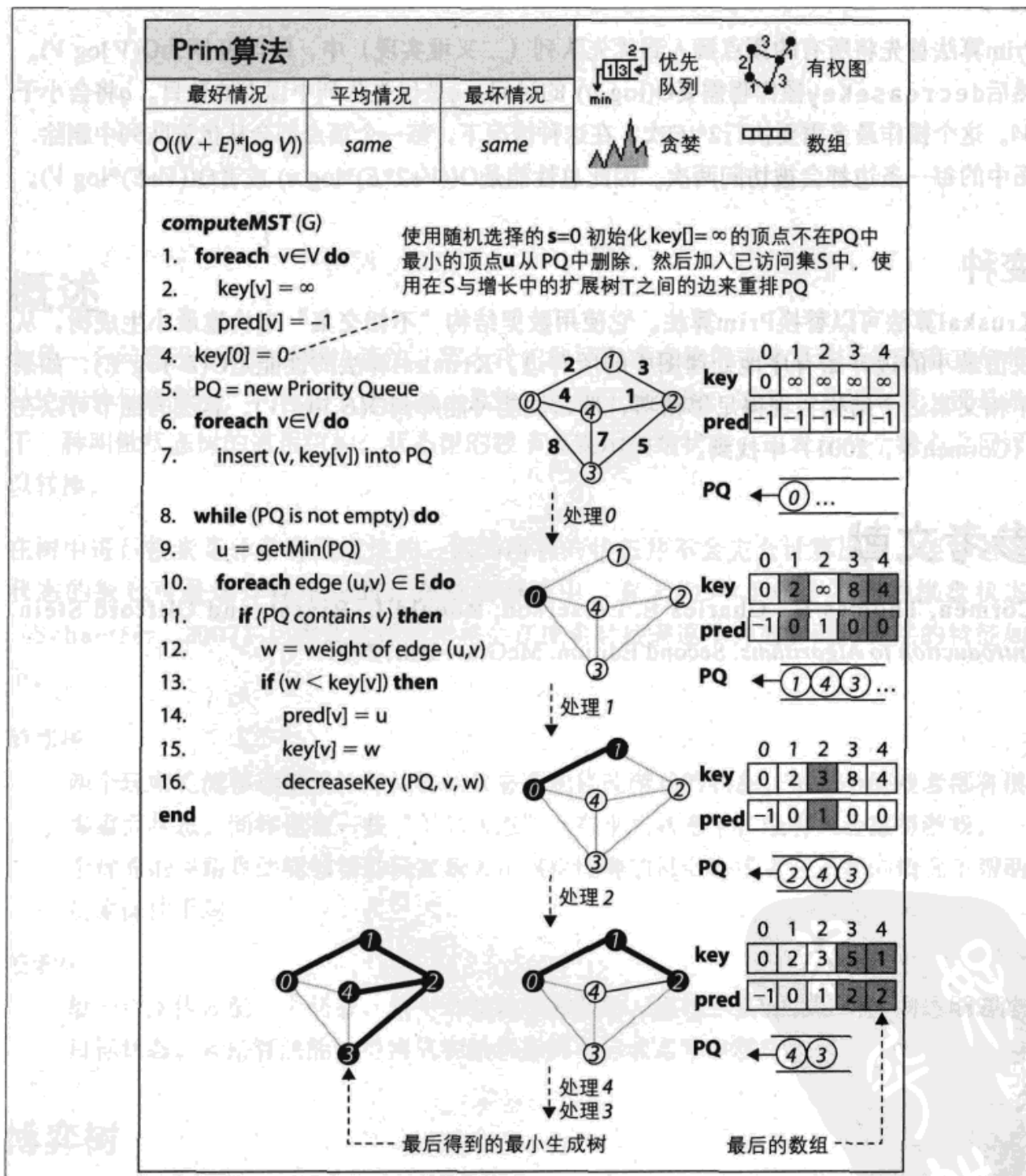


图6-19: Prim算法详解

结论

当图是稠密图时，我们可以用斐波那契堆来实现优先队列。这样性能将会提升到 $O(E+V*\log V)$ ，相比二叉堆实现而言，性能获得了相当大的提升。

分析

Prim算法首先将所有的顶点插入到优先队列（二叉堆实现）中，时间开销为 $O(V \log V)$ 。然后decreaseKey操作将需要 $O(\log q)$ 的时间， q 是优先队列中顶点的数目， q 将会小于 $|V|$ 。这个操作最多需要执行 $2*|E|$ 次，在这种情况下，每一个顶点都会从优先队列中删除，图中的每一条边都会被访问两次。因此总性能是 $O((V+2*E)*\log n)$ 或者 $O((V+E)*\log V)$ 。

变种

Kruskal算法可以替换Prim算法。它使用数据结构“不相交集”来构建最小生成树，从权值最小的边开始有序地处理图中的所有边。Kruskal算法的性能是 $O(E \log E)$ ，如果不相交这个结构实现得足够精巧，那么性能可能降到 $O(E \log V)$ 。算法的细节可以在（Cormen等，2001）中找到。

参考文献

Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, Second Edition. McGraw-Hill, 2001.





概述

如果一个问题没有明确的解决算法，那么我们就可以考虑使用寻路算法。在这章我们将讨论两种寻路算法，一种基于博弈树，另外一种基于搜索树。这两个方法本质上都是基于一种叫做状态树的通用结构。状态树的根节点表示初始状态，边表示两个状态之间可以转换。

在树中进行搜索是非常有挑战性的，因为所有的状态并不会完全计算出来，要知道，状态的数目可是爆炸性增长的。在跳棋游戏中，有大约 5×10^{20} 个不同的棋盘状态（Schaeffer, 2007）。因此这样的树将会在搜索时按需逐步构造。两种方法的特征如下。

博弈树

两个玩家轮流移动棋子，即从初始状态开始修改游戏的状态。每一个游戏者都有很多赢面状态。同样也有一些“平局状态”，在平局状态下，没有人会赢得游戏。一个优秀的寻路算法能够帮助玩家最大化赢得比赛的机会，或者在不利的情况下帮助玩家保住平局。

搜索树

给一个主体分配一个任务，从一个初始状态开始；通过一系列的移动，到达期望的目标状态。寻路算法能够得到从初始状态到目标状态的转移序列。

博弈树

有一种叫做一字棋的游戏，它需要一个 3×3 格的棋盘，玩家在棋盘上轮流画X或者O。第一个将三个标记画在同一行的玩家将会赢得这次游戏，如果没有足够的空间画标记并且没有玩家赢得游戏，那么这场游戏就是平局。10岁的孩子都知道先手玩家不可能失

败，即使对手不会犯任何错误。一个一字棋的计算机程序同样也能够完成这样的工作，这个程序使用为组合对局（例如跳棋或者象棋，但是没有扑克，因为扑克含有更多的运气成分）设计的人工智能算法。在一个组合对局中，有一个初始局面状态，每个玩家轮流地进行游戏，更新局面状态，直到达到某些胜利条件（或者是平局，没有玩家将赢得比赛）。

一字棋游戏中，仅仅只有765个局面状态（忽略掉那些对称的局面）以及26 830个不同的比赛（Schaeffer, 2002）。先手玩家总能够获胜或者保平，通常计算机专业的本科生都会被要求编写使用AI算法写一个一字棋的程序。我们仅仅需要构建博弈树（图7-1所示的是树的一部分），然后从当前局面状态（树中的顶部节点）开始，寻找到一条到达目标局面状态的路径，即胜利或者平局。

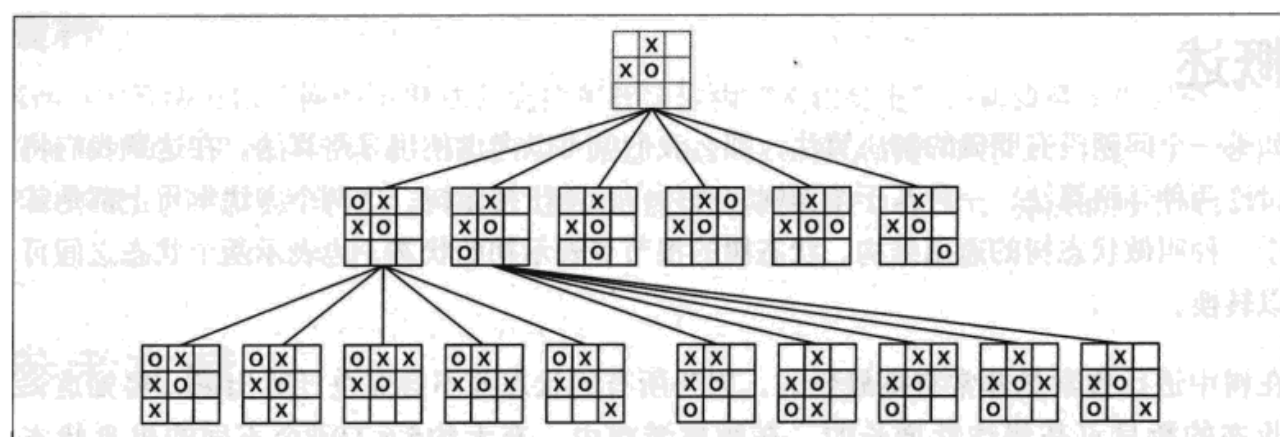


图7-1：一字棋局面状态的部分博弈树

博弈树的另外一个广为人知的名称是与或树，因为节点只有两种类型，与或者或。玩家O的博弈树如图7-1所示。顶部节点是一个或节点，因为我们仅仅在六个可能的下一步局面中选择一个而已。中间层节点是与节点，因为O的目标是无论X怎么下，O都能够至少收获平局。图7-1的博弈树仅仅是有一部分被展开，因为事实上在底层只有30个不同的局面状态。由于玩家O或者X都没有到达底层的局面状态，我们将会看到顶部的局面状态特别有意思。直观地说，一字棋程序至少要能够预先计算三步之后的局面，然后来指导玩家O如何应付当前的局面。

博弈树表示游戏中从初始局面开始，经过有效的规则能够达到的所有的局面状态集合，当然，由于这棵树的规模过于庞大，它不可能完全计算出来。寻路算法的目标就是从从一个局面状态开始，寻找一个走法来最大化玩家赢得游戏的几率。我们因此将玩家的智能决策集合转换成一个在博弈树上的寻路问题。这个方法对于较小的博弈树来说，能够很好地完成任务，它也可以被修改来解决更加复杂的问题。

跳棋的棋盘是 8×8 ，一开始有24粒棋子（12红12黑）。长久以来，研究人员都一直尝试着寻找先手玩家是否能够至少保平。虽然精确地计算出这个结果会很困难，但是我们还

是能够估算出大概有 5×10^{20} 个状态。这种情况下，博弈树的规模将会不可思议地庞大。经过近18年的计算（动用了多达200台计算机），Alberta大学的研究人员声称他们已经证明了如果双方都能完美地下每一步棋，那么结果将会是平局（Schaeffer, 2007）。

在人工智能（AI）中，有些复杂的问题可以被转换成玩家轮流进行的组合对局游戏，在这种情况下，我们可以使用寻路算法来解决。早期的人工智能（Shannon, 1950）研究人员考虑过制造一台下棋机器，并且研究出两种解决搜索问题的方法，并且沿用至今。

A类

考虑双方将来可能的一些走法集合，这个走法集合是固定的，这个策略为先手玩家找到有利位置。因此，第一步的走法决定了局势发展的方向。

B类

我们可以根据一些游戏的知识来添加一些自适应的决策，而不是简简单单地使用静态评估函数。显然地，（a）尽量多地评估能够达到目的的位置，来为下一步走法寻找到一个有利的位置，这时当前局面的评价真实反映了当前位置的重要性。以及（b）选择合适的走法，所以那些漫无目的的走法不会消耗宝贵的时间。

在本章中，我们将会介绍一些最广泛使用并且最有效的方法来减少组合对局问题的搜索空间。我们首先介绍A类算法，在双人对局中，这类算法搜索博弈树，然后提供给玩家最优走法。是属于一种通用的方法。这些算法包括Minimax、AlphaBeta和NegMax。更高级的B类算法也会介绍（例如迭代加深）。

本章讨论的算法复杂程度与数据建立的模型好坏程度息息相关。教科书或者Internet上的很多例子都是使用一个特定的对局来描述这类算法。但是，算法和对局结合得非常紧密。因此，我们希望设计一系列面向对象的接口，来将算法和对局分开。我们现在简单地概括一下博弈树算法的核心概念，如图7-2所示。

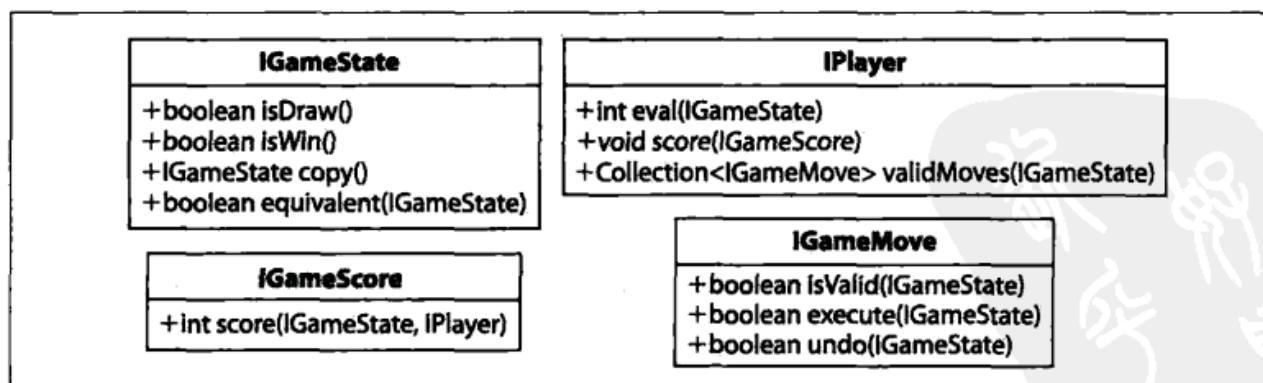


图7-2：博弈树的核心思想

IGameState接口抽象出来重要的思想，这些思想能够指导局面状态的搜索。这个接口提供了如下的功能。

解释当前局面状态

`isDraw()` 计算是否当前局面会导致平局, `isWin()` 计算是否当前局面下某个玩家已经获胜。

管理当前局面状态

`copy()` 返回了当前局面的一份副本。所以玩家的移动可以不破坏原始局面状态, `equivalent(IGameState)` 决定两个局面状态是否相等。

`IPlayer` 接口抽象出玩家能够对局面做如下操作。

评估一个局面

`eval(IGameState)` 返回一个整数, 表示玩家对当前局面的一个看法, `score(IGameScore)` 为玩家计算局面当前的得分。

得到有效的走法

`validMoves(IGameState)` 返回当前局面下的有效走法。

`IGameMove` 接口定义了当前局面下的有效走法。走法类根据问题的不同而不同, 因此搜索算法不需要了解这些类的特定实现。`IGameScore` 定义了得分计算的接口。在本章中, 我们使用 `BoardEvaluation` 计分函数来计算一字棋的得分, 这个函数由 Nil Nilsson (1971) 定义。`nc(gs, p)` 返回一字棋局面下的行数、列数或者对角数, `gs` 表示玩家 `p` 可能在一行连成三个的局面。我们定义 `score(IGameState gs, IPlayer player)` 如下:

- $+\infty$ 如果玩家能够在当前局面 `gs` 下赢得游戏。
- $-\infty$ 如果玩家在当前局面 `gs` 下输掉游戏。
- $nc(gs, player) - nc(gs, opponent)$, 如果是当前局面会导致平局。

从编程的角度来看, 博弈树寻路算法的核心是实现例7-1所示的 `IEvaluation` 接口。

例7-1: 博弈树寻路的通用接口

```
public interface IEvaluation {  
    IGameMove bestMove(IGameState state, IPlayer player, IPlayer opponent);  
}
```

从一个表示当前局面状态的节点开始, 算法假设对手走法相当完美, 基于这样的假设, 计算出玩家的最优走法。稍后我们将讨论如何在博弈树的搜索中使用 `MiniMax`、`NegMax` 以及 `AlphaBeta` 策略。

搜索树

八数码游戏是在一个 3×3 的网格上, 放着8个小方块, 分别标上1~8, 有一个空的格子没有方块。这个空格周围(水平的或者垂直的)方块可以移动到这里, 方块原有位置形

成一个新的空格。这个游戏的目标就是从一个初始状态开始，移动空格周围的方块，达到目标状态。如图7-3所示。这种游戏不存在对手玩家，但是玩家的行为和博弈树描述的行为相似。我们将问题抽象成一个初始状态（搜索树的顶部节点），以及一个从当前状态达到目标状态（标记为“GOAL”）的走法序列。从图7-3的例子中，我们可以看到从初始节点到目标节点的路径用粗体标示出来，这条路径就是解法，一共用了8步。

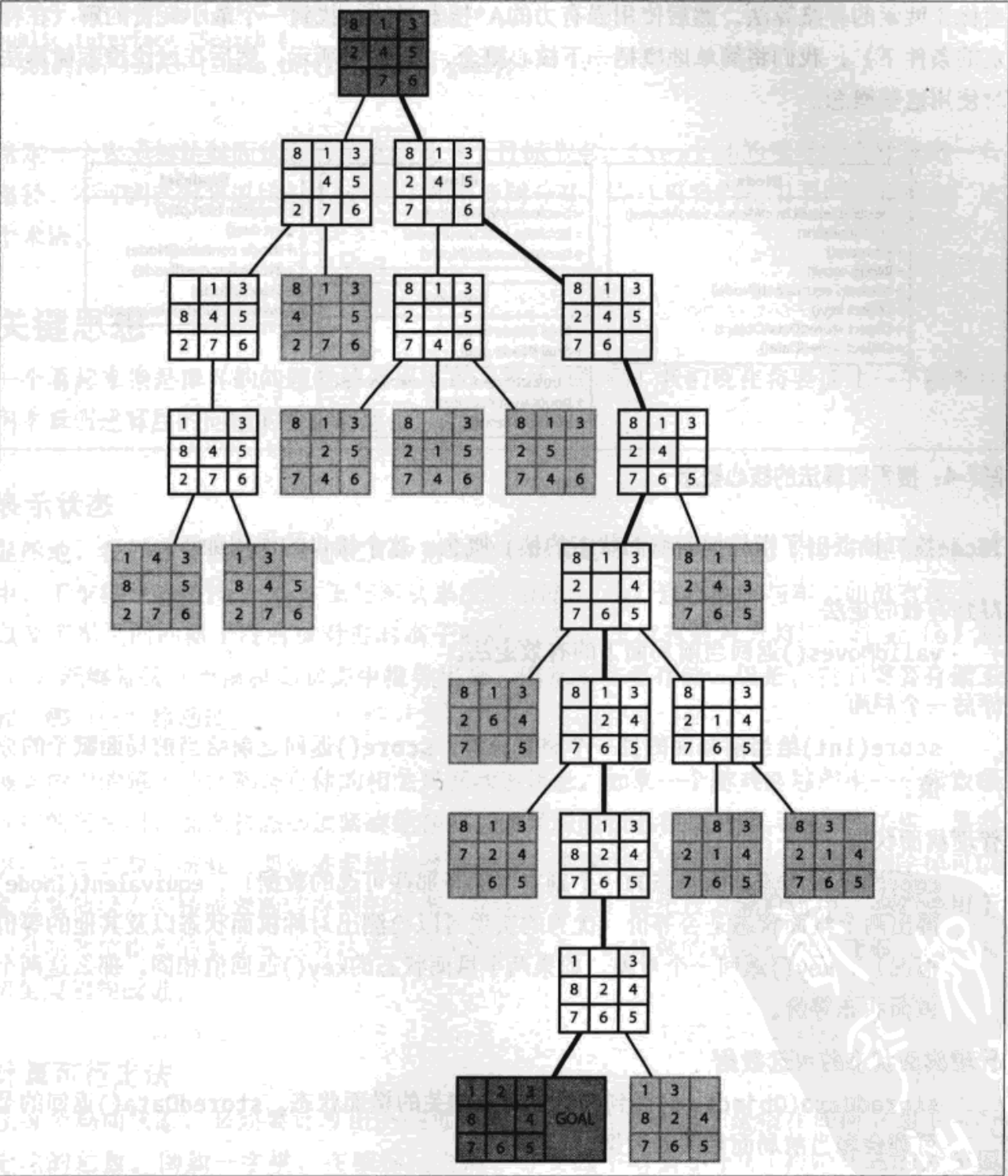


图7-3：八数码搜索样例

在这里我们使用术语搜索树，搜索树是表示一系列的中间棋面状态的树，寻路算法处理这些状态然后逐步推进。我们使用树这个结构是因为算法保证它不会处理两次同样的棋面状态。算法选择一种访问节点的顺序，然后根据这个顺序尝试达到目标状态。

搜索树通常会快速增长到数百万个状态，本章的算法将会告诉我们如何在搜索树中高效快速地搜索。想知道问题潜在的复杂度有多高吗？我们首先使用采用深度优先搜索和广度优先搜索的寻路算法。然后使用强有力的A*搜索算法，找到一个最小耗费的解（在特定的条件下）。我们将简单地概括一下核心概念，如图7-4所示，然后在讨论搜索树算法时使用这些概念。

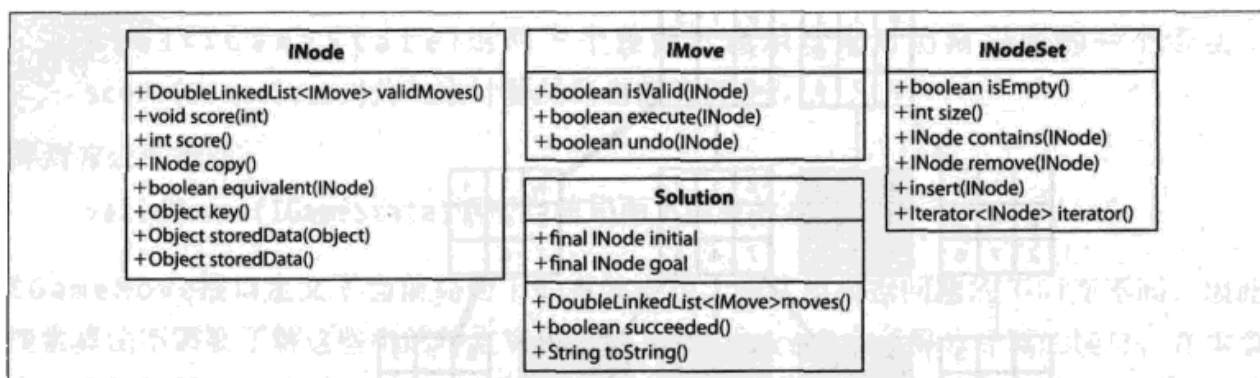


图7-4：搜索树算法的核心概念

INode接口抽象出了指导搜索棋面状态的核心概念。这个接口的作用如下：

得到有效的走法

validMoves()返回当前局面下的有效走法。

评估一个局面

score(int)给当前局面的打一个整数分值，**score()**返回之前给当前局面赋予的分值。

管理棋面状态

copy()返回当前棋面状态的一个副本（除开那些可选的数据），**equivalent(INode)**得出两个棋面状态是否等价（优秀的实现可以检测出对称棋面状态以及其他的等价情况）。**key()**返回一个对象，如果两个棋面状态的**key()**返回值相同，那么这两个棋面状态等价。

管理棋面状态的可选数据

storedData(Object o)得到和给定对象相关的棋面状态。**storedData()**返回的是可能会和当前局面相关的数据。

INodeSet接口抽象出了INode集合组织形式的实现。有些算法需要队列，有些需要栈，有些需要平衡二叉搜索树。一旦选择了适当的结构（使用StateStorageFactory类），

算法将会利用此结构能够支持的操作来维护INode集合。IMove接口定义了走法如何影响局面，不同的问题走法不同，搜索算法不需要关注于它们的实现。

从编程角度来看，搜索搜索树时，寻路算法的核心是例7-2所示的ISearch接口的实现。我们知道解法后，能够反推出走法。

例7-2：搜索树的寻路算法通用接口

```
public interface ISearch {  
    Solution search (INode initial, INode goal);  
}
```

给定一个表示初始棋面状态的节点以及一个目标节点，ISearch的实现将会计算出一条路径，不可到达时将返回null。为了和博弈树分开，在这里我们使用了“棋面状态”这个术语。

关键思想

一个看起来像是博弈的问题能够使用本章的算法来解吗？我们现在将要描述一下能够使用本章所述算法的问题的关键思想。

表示状态

显然地，我们需要能够从当前位置中获取这个位置的所有状态信息。例如，在国际象棋中，王车移位条件是（a）王与车从来没有动过，（b）自己的王与车之间没有棋子，以及王车之间的格子没有被对方的棋子威胁，（c）王没有被对方将军。注意（b）和（c）能够直接从当前棋面状态中推导出来，因此不需要存储，但是，我们需要存储王和车哪个已经移动过。

搜索树中的每个节点都是存储的相关局面状态信息。如果一个游戏能够产生一个指数级规模的博弈树，那么状态必须紧凑地存储。如果局面状态存在对称，例如四子棋、黑白棋或者十五数码游戏，那么搜索树的规模将会大幅减少，因为可以检测并且删除掉可以通过其他状态旋转或者翻转得到的状态。在国际象棋，跳棋或者黑白棋中，我们使用了一种叫做位棋盘的复杂表示方法来管理大量的状态，这样做的好处是性能能够得到令人印象深刻的改进。

计算可行走法

在每个局面状态，必须要计算出一些可行走法。术语分支因子是指在任何局面下可行走法的总数。例如一字棋，在画标记之后，分支因子将会小于9（初始状态的分支因子）。3×3的魔方初始状态分支因子（平均）是13.5（Korf, 1985），而四子棋大多数情况下分支因子为7。国际跳棋（不同于我们经常见到的那种）则复杂得多，因为玩家

下棋需要遵守的规则比较复杂。分析过大量跳棋数据库之后，吃子位置的分支因子大概是1.20，而非吃子位置的分支因子是7.94，Schaeffer计算出整个比赛中，平均的分支因子为6.14（Schaeffer，2008）。而围棋的初始分支因子超过360（Berlekamp和Wolfe，1997）。

许多算法对于走法的顺序是非常敏感的。事实上，如果某个游戏的分支因子非常高，而且走法由于某些原因没有被正确地排序，那么盲目地搜索博弈树是得不到解的。

使用启发式信息

如果盲目搜索的算法不可能利用局面状态的信息，它仅仅是根据固定的策略。深度优先盲目搜索简单地迭代所有可行的走法，直到找到一个解法。而广度优先盲目搜索将会尝试完 k 步是否能够得到可行解之后再去尝试 $k+1$ 步是否能够得到。

我们可以通过如下几种方法来得到智能搜索（Barr和Feigenbaum，1981）。

根据棋面状态选择如何扩展而不是使用固定的策略

并不一直使用深度优先或者广度优先结构，算法将根据当前需要在这两种策略中灵活转换。

选择适当数目的可行走法，以及走的顺序

当考虑当前棋面状态下可行走法时，玩家必须评估一下这些走法，哪些走法看起来能够赢得比赛，而哪些看起来不会。另外，玩家可能会抛弃那些看起来不会赢得比赛的走法。

选择棋面状态，剪枝搜索树

在搜索过程中，可能会获得新的知识，利用这些新知识，我们可以从选择列表中删除不必要的棋面状态。

最通用的方法是定义一个静态的评价函数，在计算的过程中评价局面状态，排序可行走法。评价函数对寻路算法的性能影响非常大。事实上，使用了坏的评估函数的寻路算法将不可能在每一步选择到最优走法。有句谚语说得好：“错误的输入必然导致错误的输出。”

我们不会限制评价当前局面状态，一个评价函数能够短暂地扩展博弈树，插入一些走法，然后从中选择看起来能够为玩家带来更多好处的走法。但是现实往往不尽如人意，因为（a）这些操作的开销，以及（b）通用的选择逻辑。在A*搜索的讨论中，我们将会给出一个搜索八数码的样例，这个样例使用了不同的评价函数，我想你将会非常高兴地看到设计高效函数是一门含蓄的艺术。

静态函数必须考虑到当前所出位置的各种特征，返回一个整数得分，这个得分反映了

玩家对当前位置的一个看法。例如，第一个成功的国际跳棋程序是由Arthur Samuel (1959) 开发的，它评价每个棋盘位置，考虑游戏的多达24个特征，例如“棋子优势特性”（比较玩家和其对手的棋子数目）以及“获胜交换特性”（在胜利时交换棋子而不是在将要失败时）。很明显，如果评价函数能够做得足够好，那么一个下棋引擎可以看做是一名优秀的玩家。

最大扩展深度

由于内存资源的限制，一些搜索算法在扩展搜索树和博弈树时会选择扩展到一定程度为止。这个方法有自己的局限性，例如一个玩家需要走很多步来布一个局。例如，在国际象棋中，经常会出现丢卒保车这样的现象，为了获得更多的利益而牺牲掉某些棋子，如果恰好在最大扩展深度的边界处牺牲掉某些棋子，那么我们做出的牺牲将不会得到任何回报。使用固定的扩展深度，程序不会搜索某条水平线以下的状态，这样就对成功搜索造成了不利的影响。

假设

我们假设问题的局面状态都可以高效地表示，并且在博弈树或者搜索树的节点上，只有有限多个可行走法。搜索空间事实上可能是一个图，而不是一棵树，因为到达同一个状态可能有多种走法。即便这样，我们也在一个类似于树而不是图的结构上使用寻路算法，并且评价线性的走法序列。

我们假设存在这样一个问题，我们叫它Tiny-Puzzle，作为深度优先搜索，广度优先搜索以及A*搜索的伪代码所描述的例子。在Tiny-Puzzle中，一个棋面状态包含两个非负数， s_0 和 s_1 。每一个棋面状态存在两种可行走法——（1）增加 s_0 ，以及（2）增加 s_1 ，所以这个游戏的分支因子是2。例如，给定初始状态 $\langle s_0=0, s_1=0 \rangle$ ，目标状态 $\langle s_0=1, s_1=2 \rangle$ 能够在3步内达到：增加 s_1 ，增加 s_0 ，然后增加 s_1 。

深度优先搜索

深度优先搜索（图7-5）不断地向前寻找可行状态，试图一次找到通向目标状态的道路，它并不会两次访问同一个状态。因为某些搜索树包含大量的棋面状态，因此深度优先搜索只是在最大搜索深度固定的情况下才具有可行性。深度优先搜索维护一个栈，保存未访问过的棋面状态。在每次迭代时，深度优先搜索从栈中弹出一个未访问的棋面状态，然后从这个棋面状态开始扩展，根据走法计算其后继棋面状态。如果达到了目标棋面状态，搜索终止。如果没有达到的话，任何在闭合集集中的后继棋面状态都会被抛弃。剩余的未访问棋面状态被压入栈中，然后继续搜索。

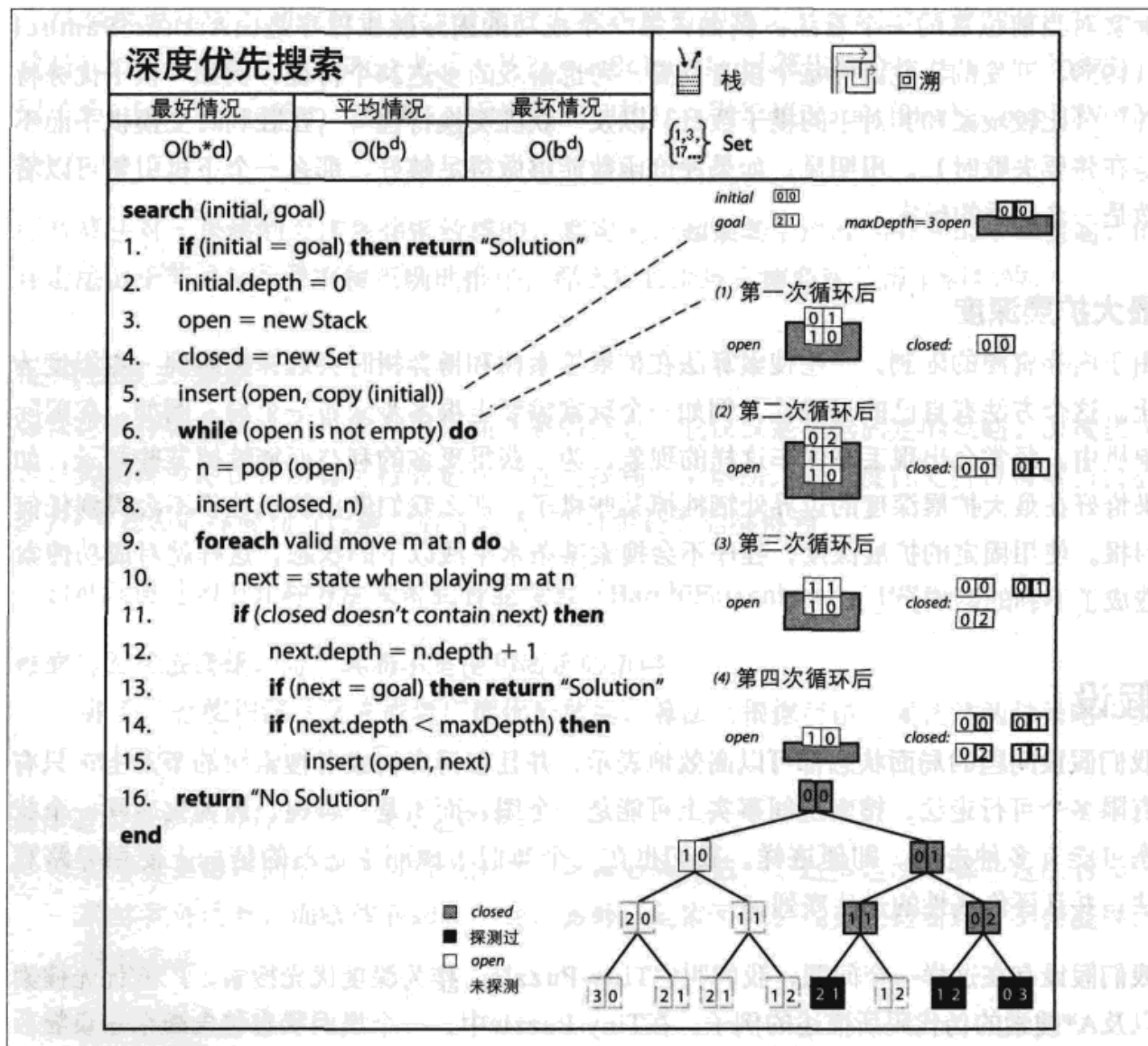


图7-5: 深度优先搜索详解

在一个八数码问题中, 从初始棋面状态开始, 限定最大深度为9, 我们可以得到一棵搜索树, 如图7-6所示。我们从图中可以看到, 虽然有些地方已经扩展到深度9, 但是我们找到的解只需经过8步。在这棵树中, 我们已经访问了50个棋面状态, 还剩下4个未访问 (浅灰色表示)。

输入/输出

输入

算法从一个初始棋面状态开始, 寻找一个目标状态。算法假设在给定棋面状态下可以尝试所有的可行走法。

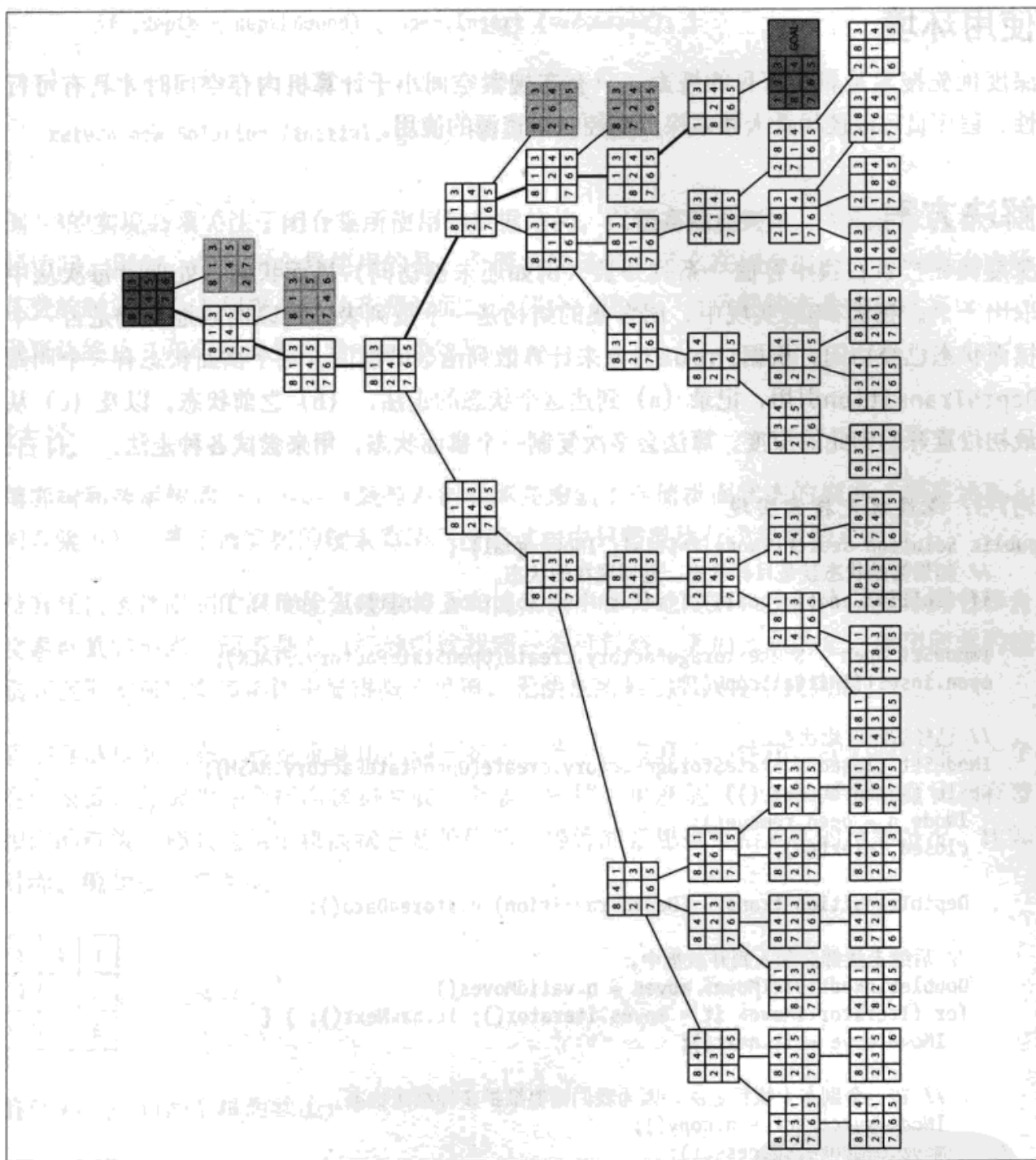


图7-6：八数码问题的深度优先搜索树

输出

返回一个走法的序列，表示从初始状态到目标状态的路径（或者只是输出是否存在一个解）。

假设

为了分析方便，我们假设 d 为深度优先搜索的最大深度， d 为搜索树的分支因子。

使用环境

深度优先搜索是属于盲目的搜索，只有在搜索空间小于计算机内存空间时才具有可行性。程序员可以使用最大搜索深度来控制对资源的使用。

解决方案

深度优先搜索在栈中存储一系列开放（例如还未被访问）棋面状态，处理时每次从中取出一个。在例7-3的实现中，闭合集的结构是一个散列表，可以高效地查询是否一个棋面状态已经访问，根据INode对象来计算散列函数的键值、每个棋面状态有一个叫做DepthTransition引用，记录（a）到达这个状态的走法，（b）之前状态，以及（c）从最初位置开始到此的深度。算法会多次复制一个棋面状态，用来尝试各种走法。

例7-3：深度优先搜索实现

```
public Solution search(INode initial, INode goal) {
    // 如果初始状态就是目标状态，返回现在的状态。
    if (initial.equals(goal)) { return new Solution (initial, goal); }

    INodeSet open = StateStorageFactory.create(OpenStateFactory.STACK);
    open.insert(initial.copy());

    // 已经访问过此状态。
    INodeSet closed = StateStorageFactory.create(OpenStateFactory.HASH);
    while (!open.isEmpty()) {
        INode n = open.remove();
        closed.insert(n);

        DepthTransition trans = (DepthTransition) n.storedData();

        // 后继走法都会加入到开放集中。
        DoubleLinkedList<IMove> moves = n.validMoves();
        for (Iterator<IMove> it = moves.iterator(); it.hasNext(); ) {
            IMove move = it.next();

            // 在一个副本上执行走法，因为我们需要维护多种棋面状态。
            INode successor = n.copy();
            move.execute(successor);

            // 如果已经访问，尝试另外一种状态。
            if (closed.contains(successor) != null) { continue; }

            int depth = 1;
            if (trans != null) { depth = trans.depth+1; }

            // 记录求解轨迹中前一步走法。如果找到解，那么立刻退出，
            // 否则的话在限制的深度内将其加入开放集。
            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); // 无解。
}

```

例7-3的实现必须关注于闭合集所使用的数据结构，以便高效地得知一个棋面状态是否已经访问。例如，如果闭合集使用的是一个简单的链表，那么在闭合集中搜索一个元素所花费的时间将会占用整个算法花费时间的大部分。注意，当后继状态是目标状态时，搜索算法终止（在广度优先搜索中同样如此）。

结论

博弈树和搜索树的一个不同就是搜索树必须在搜索中存储棋面状态的副本（在开放集和闭合集中）。基于博弈树的搜索算法在搜索过程中只需要执行或者撤销走法。

没有任何条件限制的深度优先搜索将会在搜索树中盲目地搜索，我们可以预见到它将会搜索巨量的节点，而不是有目标地尝试找到一条可行路。更加讽刺的是，使用固定的搜索深度限制在巨型搜索树中显得微不足道，无法在所限制的深度中找到解。

我们存储棋面状态，避免重复访问同一状态。为了改进算法的性能，我们假设存在一个高效函数，能够为每个棋面状态生成一个唯一键值，也就是说，如果两个棋面状态有着相同的键值，那么这两个棋面状态是等价的。等价的意思也包括其他的相等情况，比如对称。例如棋面状态：

8	1	3
2	4	5
	7	6

旋转90° 得到如下棋面状态：

3	5	6
1	4	7
8	2	

这两个棋面状态可以认为是等价的。

相比广度优先搜索，深度优先搜索在开放集中存储更少的信息，因此需要的空间更少。

分析

算法的性能取解决问题的共性和特性。一般来说，开放集和闭合集的核心操作会出乎意料地降低算法的性能，因为在集合中寻找元素的朴素实现将会需要 $O(n)$ 的时间。核心操作包括：

`open.remove()`

从开放集中删除状态。

`closed.insert(INode state)`

向闭集中添加状态。

`closed.contains(INode state)`

查询是否给定的状态在闭集中。

`open.insert(INode state)`

向开放集中添加状态，以供稍后访问。

因为深度优先搜索使用了栈来存储开放集，因此添加和删除操作只需要常数时间。但是，如果闭合集是一个简单链表，那么`closed.contains(INode)`的时间花费将会是 $O(n)$ ， n 是闭集中状态的数目。这个高昂的开销可以通过使用树或者散列结构来避免，在两种结构中，都需要用到键值（实现INode接口的棋面状态类提供）。

问题的特性在如下方面能够影响性能：（a）一个棋面状态的后继状态数目，以及（b）可行走法的顺序。有些问题中，一个棋面状态下有着大量的可行走法，也就是说，深度优先地选择很多的路径将会是不明智的。同样的，走法的排序也会影响整个搜索过程。如果可以使用某些启发式信息，那么我们将看起来可能能够得到解的走法排在表的前面。我们也可以利用对称性来改进搜索。无论棋盘如何旋转，计算出来的棋面状态键值都是相等的，现在，我们不可能改进开放集的结构，因为深度优先搜索的确需要一个栈结构，但是，我们可以从闭合集入手，减少空间消耗。

我们使用三个例子来讨论一下深度优先搜索的性能，我们可以看到仅仅是状态的些许不同，搜索却差异甚大。在每个例子中，至多需要走10步才能找到解，表7-1列出了使用可变深度限制的深度优先搜索性能。偶尔深度优先搜索能够很快地找到解，见表7-2，使用固定深度8，算法从棋面状态N1开始，在搜索25个棋面状态之后，找到了一个需要走8步的解（比我们预想的要好两步）。图7-7表示的是随着搜索深度的增长，深度优先搜索的搜索树规模变化情况。一般来说，搜索树的规模是分支因子 b 的指数级。对于八数码问题来说，根据空格的位置，分支因子在1.6~3.81之间变化（Reinefeld, 1993）。三个近似函数表示的是给定搜索深度 d ，从三个初始位置开始，搜索树的规模。

$$N1: \text{size}(n)=0.3429*d^{2.6978}$$

$$N2: \text{size}(n)=0.2403*d^{3.2554}$$

$$N3: \text{size}(n)=0.2814*d^{3.044}$$

表7-1：三个初始位置下，搜索树的规模

深度限制d	<table><tr><td>8</td><td>1</td><td>3</td></tr><tr><td>2</td><td>4</td><td>5</td></tr><tr><td></td><td>7</td><td>6</td></tr></table>	8	1	3	2	4	5		7	6	<table><tr><td>8</td><td>1</td><td>3</td></tr><tr><td>7</td><td>2</td><td>5</td></tr><tr><td></td><td>4</td><td>6</td></tr></table>	8	1	3	7	2	5		4	6	<table><tr><td>1</td><td>4</td><td></td></tr><tr><td>7</td><td>3</td><td>2</td></tr><tr><td>6</td><td>8</td><td>5</td></tr></table>	1	4		7	3	2	6	8	5
	8	1	3																											
	2	4	5																											
	7	6																												
8	1	3																												
7	2	5																												
	4	6																												
1	4																													
7	3	2																												
6	8	5																												
1	1	1	1																											
2	3	3	3																											
3	7	7	7																											
4	15	15	15																											
5	31	31	31																											
6	51	51	51																											
7	90	90	90																											
8	25	149	150																											
9	37	254	259																											
10	54	345	113																											
11	81	421	195																											
12	105	518	293																											
13	168	728	401																											
14	216	678	400																											
15	328	1,133	652																											
16	1,374	1,991	886																											
17	2,121	2,786	1,443																											
18	729	1,954	1,897																											
19	1,218	3,240	3,575																											
20	1,601	4,350	2,211																											
21	1,198	5,496	7,437																											
22	2,634	6,315	1,740																											
23	4,993	11,053	6,267																											
24	3,881	8,031	4,214																											
25	6,723	20,441	11,494																											

表7-1：三个初始位置下，搜索树的规模（续）

深度限制d	<table><tr><td>8</td><td>1</td><td>3</td></tr><tr><td>2</td><td>4</td><td>5</td></tr><tr><td></td><td>7</td><td>6</td></tr></table>	8	1	3	2	4	5		7	6	<table><tr><td>8</td><td>1</td><td>3</td></tr><tr><td>7</td><td>2</td><td>5</td></tr><tr><td></td><td>4</td><td>6</td></tr></table>	8	1	3	7	2	5		4	6	<table><tr><td>1</td><td>4</td><td></td></tr><tr><td>7</td><td>3</td><td>2</td></tr><tr><td>6</td><td>8</td><td>5</td></tr></table>	1	4		7	3	2	6	8	5
8	1	3																												
2	4	5																												
	7	6																												
8	1	3																												
7	2	5																												
	4	6																												
1	4																													
7	3	2																												
6	8	5																												
26	2,071	15,074	15,681																											
27	5,401	10,796	26,151																											
28	1,206	16,233	2,081																											
29	1,958	33,526	22,471																											
无限制	53	1,962	67,808																											

表7-2：三个初始位置下，深度优先搜索树得到的结果位置

d	N1走法	N2走法	N3走法
1	0	0	0
2	0	0	0
3	0	0	0
4	0	0	0
5	0	0	0
6	0	0	0
7	0	0	0
8	8	8	0
9	8	8	0
10	8	8	10
11	8	8	10
12	12	12	10
13	12	0	10
14	8	14	14
15	8	14	14
16	10	0	14
17	10	0	10
18	12	18	10
19	12	18	10
20	20	18	10

表7-2：三个初始位置下，深度优先搜索树得到的结果（续）

位置

d	N1走法	N2走法	N3走法
21	20	0	14
22	20	18	22
23	20	18	10
24	24	14	22
25	8	0	18
26	26	14	26
27	20	24	0
28	28	14	28
29	28	26	18
无限制	30	1,029	37,980

程序员想知道图7-7的搜索树这样的规模下，解的质量如何。我们可以得到如下两个结论：

一个不恰当的搜索深度得不到解

考虑初始状态N2，搜索深度为25，在搜索20 441个棋面状态之后，我们仍然没有得到解。这怎么可能？因为深度优先搜索不会重复访问棋面状态。搜索可是能够在第3451个棋面状态时就能够找到解。

1	2	3
7	8	4
6		5

上图是在深度为25时得到的棋面状态，这个棋面状态只有3步就能够得到解。由于这个棋面状态已经访问过，搜索树不会再继续扩展，所以这个状态会被加到闭合集中。并且搜索深度已经达到了深度限制，所以不会从这个状态开始继续搜索，即使之后深度优先搜索在较早的等级访问到这个状态，它也不会继续搜索，因为这个状态已经在闭合集中。

随着深度的增加，解的质量可能会越来越不尽如人意

随着深度的增加，我们可以看到有时得到的解的规模是如何数倍于期望规模的。

很有意思的是，给定初始状态N1，一个没有限制的深度优先搜索将会在访问30个棋面状态之后找到一个30步的解，这时开放集中仍然有23个棋面状态。但是，这种好事可不会一直发生，我们可以从初始状态N2和N3得到的解中可以看出。

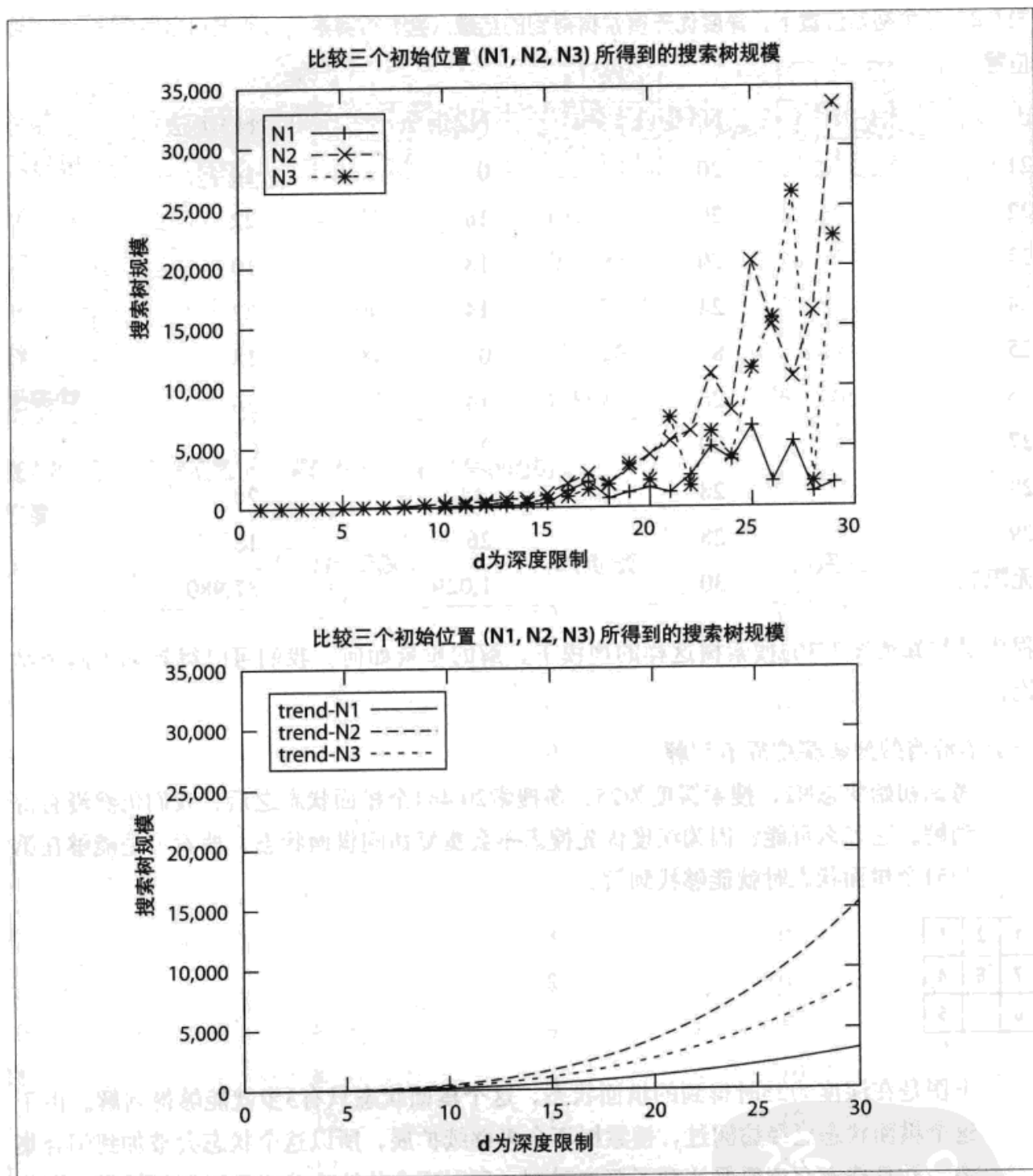


图7-7：随着深度增加，深度优先搜索搜索树的规模

广度优先搜索

广度优先搜索（图7-8）尝试在不重复访问状态的情况下，寻找到一条最短路径。广度优先搜索保证如果存在一条到目标状态的路径，那么找到的肯定是最短路径。

事实上，深度优先搜索和广度优先搜索的唯一不同就是广度优先搜索使用队列来保存开放集，而深度优先搜索使用栈。每次迭代时，广度优先搜索从队列头拿出一个未访问的

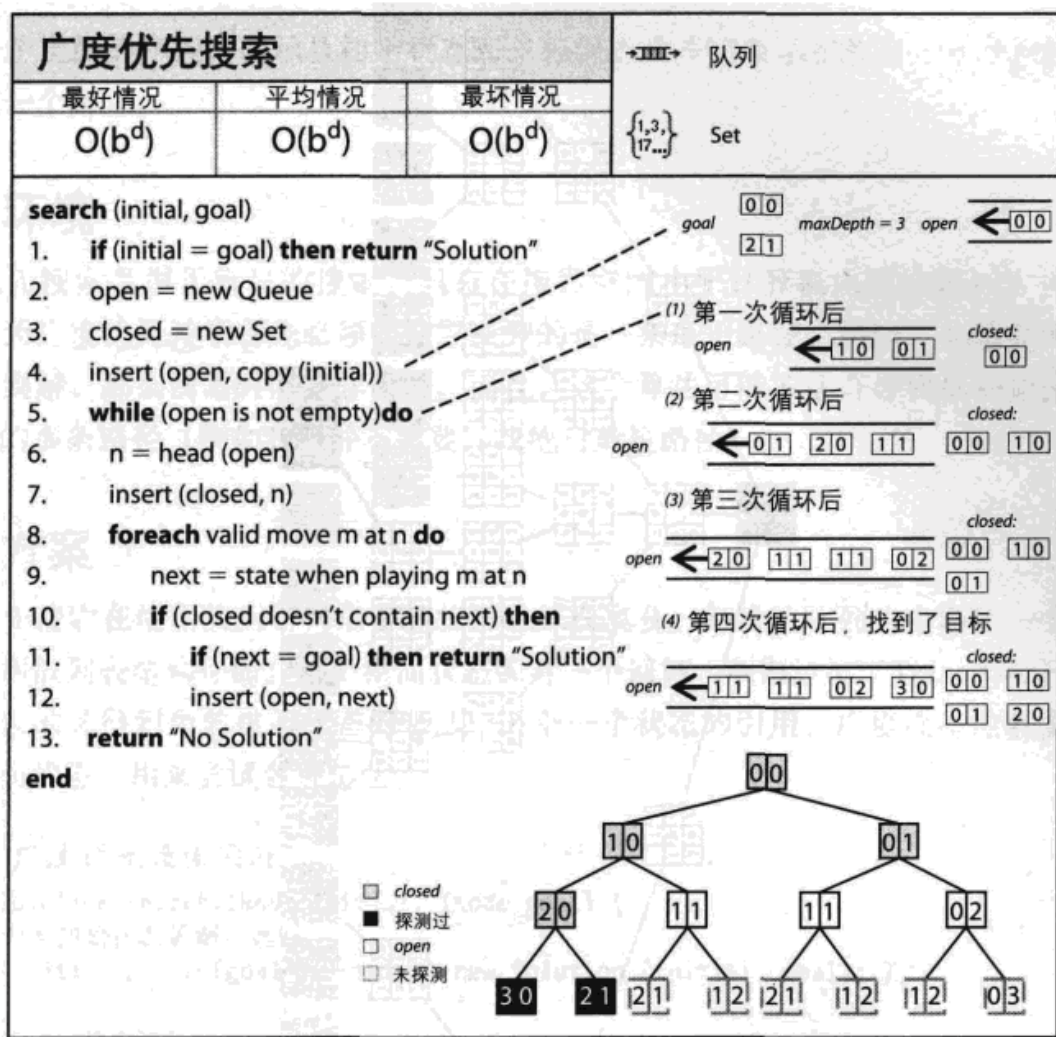


图7-8：广度优先搜索详解

状态，然后从这个状态开始，计算后继状态。如果达到了目标状态，那么搜索结束。任何已经在闭合集集中的后继状态将会被抛弃。剩余的未访问棋面状态将会放入开放集队列尾部，然后继续搜索。

使用如下状态作为八数码游戏的初始状态：

2	8	3
1	6	4
7		5

图7-9是计算出的搜索树，我们可以看到算法是如何在搜索所有4步长的路径之后，找到了一个5步长路径的解（几乎所有5步长的路径都被探测过）。图中20个灰黑色的棋面状态是在开放集中等待被搜索的。总共处理了25个棋面状态。

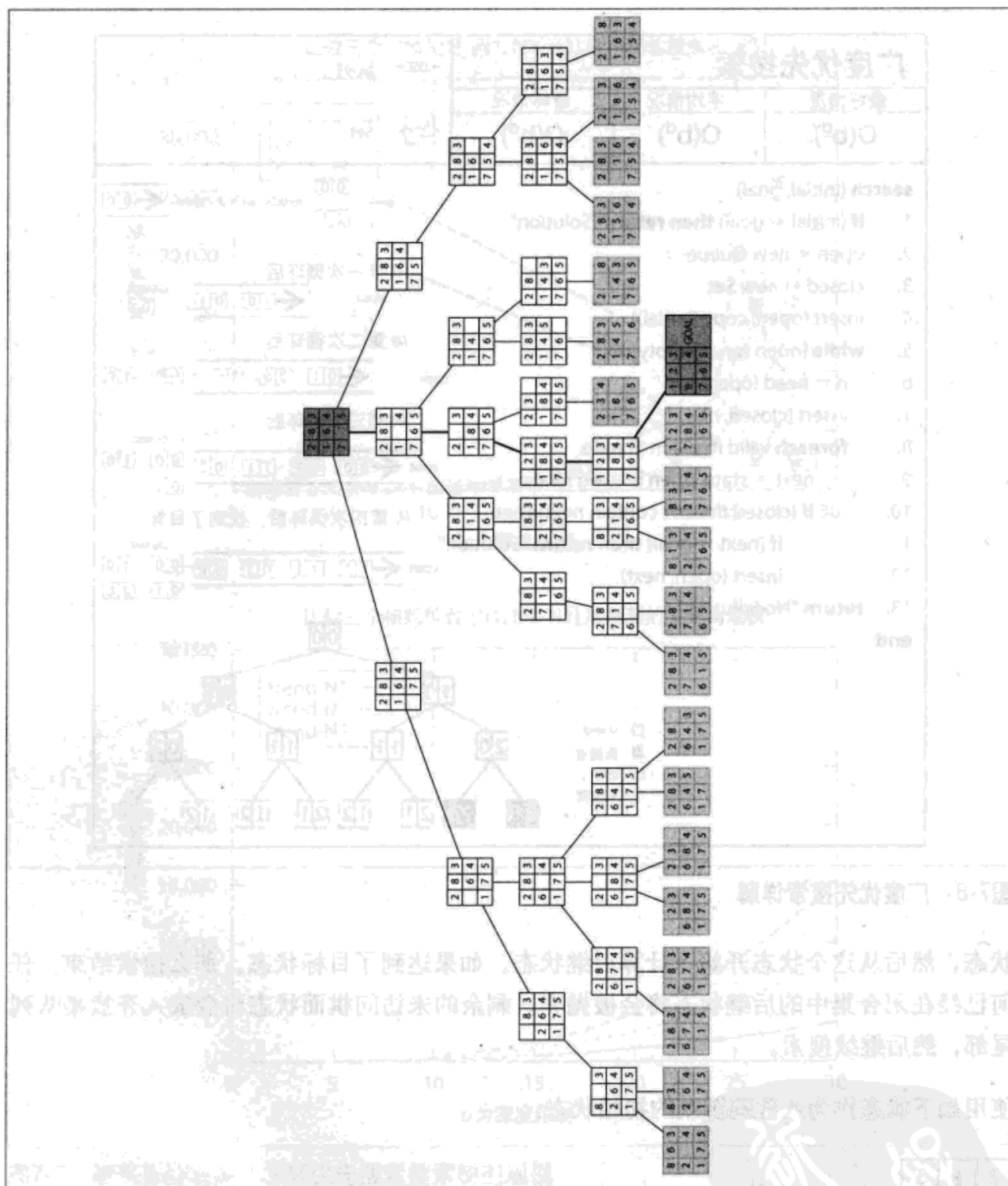


图7-9：八数码问题的广度优先搜索树

输入/输出

输入

算法从一个初始棋面状态开始，寻找一个目标状态。算法假设在给定棋面状态下可以尝试所有的可行走法。

输出

返回一个走法的序列，表示从初始状态到目标状态的开销最小的路径（或者只是输出是否存在一个解）。

使用环境

广度优先搜索是属于盲目的搜索，只有在搜索空间小于计算机内存空间时才具有可行性。因为广度优先搜索首先必须保证寻找到的是一条最短路径，所以，如果需要很多步才能得到解，那么所需时间会非常长。而且，这个算法可能不适合寻找从初始状态到目标状态的多条路径（例如我们并不需要寻找绝对最短路径）。

解决方案

广度优先搜索在使用队列结构来存储状态的开放集合，每次从队列头中取出一个状态。闭合集用散列表结构存储。每个棋面状态都有一个回链，叫做过渡（Transition），这个链接记录的是得到当前棋面状态的走法以及前一个状态的引用。广度优先搜索多次复制一个棋面状态，用来尝试各种走法。

例7-4：广度优先搜索实现

```
public Solution search(INode initial, INode goal) {
    // 如果初始状态是解，返回。
    if (initial.equals(goal)) { return new Solution (initial, goal); }

    // 从初始状态开始。
    INodeSet open = StateStorageFactory.create(StateStorageFactory.QUEUE);
    open.insert(initial.copy());

    // 我们已经访问过的状态。
    INodeSet closed = StateStorageFactory.create(StateStorageFactory.HASH);
    while (!open.isEmpty()) {
        INode n = open.remove();
        closed.insert(n);

        // 所有的后继走法都会追加到开放集中。
        DoubleLinkedList<IMove> moves = n.validMoves();
        for (Iterator<IMove> it = moves.iterator(); it.hasNext(); ) {
            IMove move = it.next();

            // 创建一个副本。
            INode successor = n.copy();
            move.execute(successor);

            // 如果已经访问过此状态，不再访问。
            if (closed.contains(successor) != null) {
                continue;
            }
        }
    }
}
```



```

        // 记录求解轨迹中前一步走法。如果找到解，那么立刻退出，否则将其加入开放集。
        successor.storedData(new Transition(move, n));
        if (successor.equals(goal)) {
            return new Solution (initial, successor);
        }
        open.insert(successor);
    }
}

return new Solution (initial, goal, false); //无解。
}

```

结论

广度优先搜索也是在搜索树中盲目地搜索，在搜索可行的路径时会访问大量的节点。它虽然保证能够找到最短路径，但是也需要维护一个规模很大的开放集。不过，让我们感到宽心的是，由于使用的队列来存储开放集合，所以插入和删除操作能够在常数时间完成。

分析

和深度优先搜索一样，广度优先搜索的性能和问题的共性和特性有关。深度优先搜索对于共性的分析同样适用于广度优先搜索，但是只是在开放集合的规模上有不同。广度优先搜索需要在开放集合中顺序存储 b^d 个棋面状态， b 是棋面状态的分支因子， d 是找到的解的深度。开放集合的规模比深度优先搜索大得多，深度优先搜索的开放集合只需要存储 $b \times d$ 个棋面状态，即深度 d 时候的候选棋面状态数量。不过广度优先搜索能够保证找到一个解，从初始状态开始到目标状态，这个解的步数是最少的。

A*搜索

广度优先搜索能够找到一个最优解（如果存在），但是可能需要访问大量的节点，因为我们可以看到，它并没有尝试对候选走法进行排序。相反，深度优先搜索却是尽可能多地向前探测路径，不过，深度优先搜索的搜索深度必须得到限制，要不然它很有可能会在没有任何结果的路径上花费大量的时间。A*搜索在搜索时能够利用启发式信息，智能地调整搜索策略。

如图7-10所示，A*搜索是一种迭代的有序搜索，它维护一个棋面状态的开放集合。在每次迭代时，A*搜索使用一个评价函数 $f^*(n)$ 评价开放集合中的所有棋面状态，选择最小的棋面状态。我们定义 $f^*(n)=g^*(n)+h^*(n)$ ：

$g^*(n)$ 估算从初始状态到状态 n 的最短走法序列。

$h^*(n)$ 估算从状态 n 到目标状态的最短走法序列。

$f^*(n)$ 估算从初始状态开始，经过状态 n ，到达目标状态的最短走法序列。

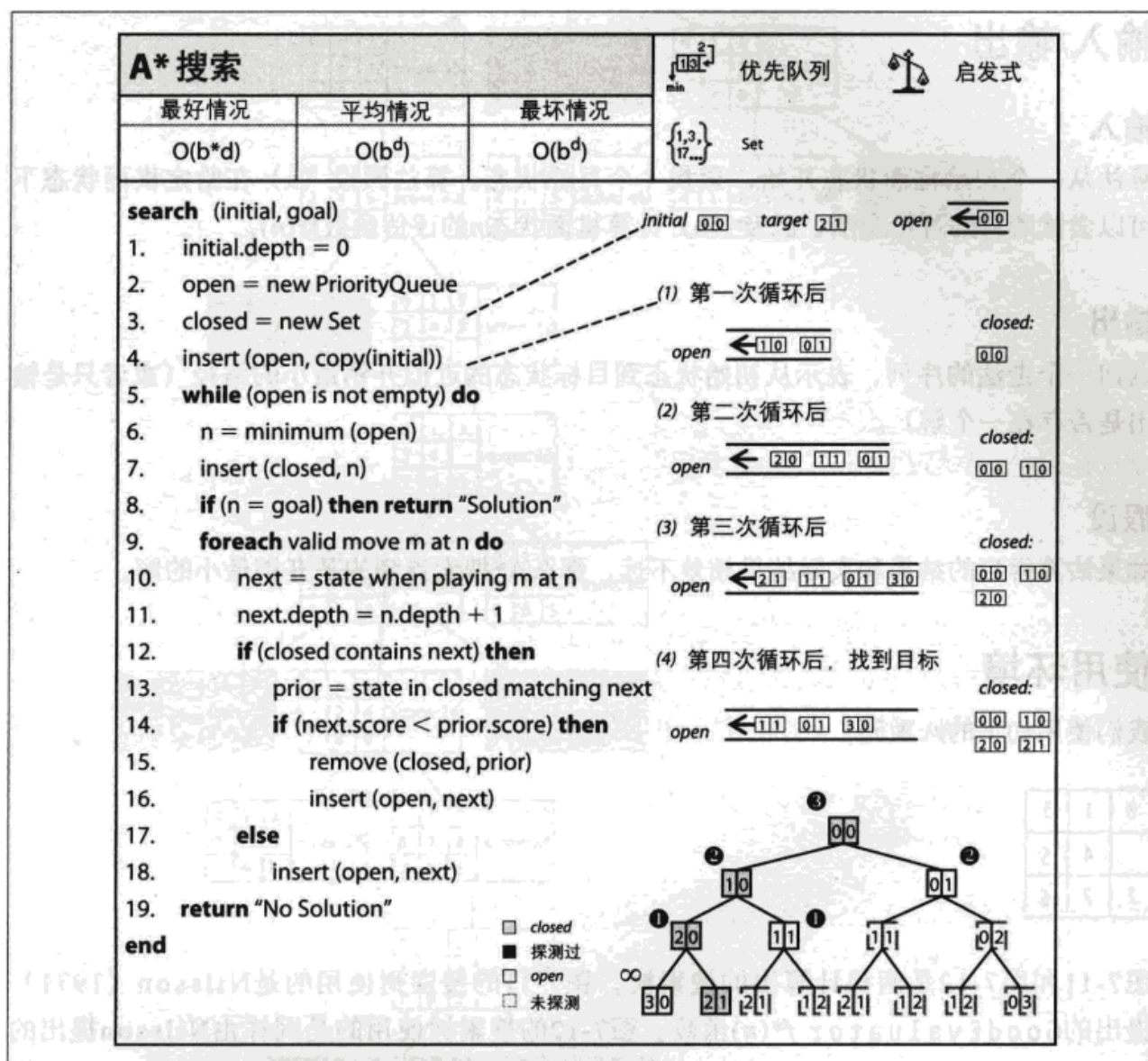


图7-10: A*搜索详解

星号*表示使用了启发式信息（自从1968年开发出此算法后，这个记法被广泛接受），因此 $f^*(n)$ ， $g^*(n)$ 以及 $h^*(n)$ 是对实际开销 $f(n)$ ， $g(n)$ 和 $h(n)$ 的估算，这些实际开销只能在得到解之后才能够知道。简而言之， $f^*(n)$ 越低，表示状态 n 越接近目标状态。

$f^*(n)$ 最关键的部分是启发式的计算 $h^*(n)$ ，因为 $g^*(n)$ 能够在搜索的过程中，通过记录状态 n 的深度计算出来（注1）。如果 $h^*(n)$ 不能够准确地区分开有继续搜索价值的状态和没

注1： 注意 $g^*(n)$ 可能 $\geq g(n)$ ，因为事实上可能有更少的走法达到目标。

有价值的状态，那么A*搜索不会表现得比上述任何盲目搜索要好。如果能够准确地估算 $h^*(n)$ ，那么使用 $f^*(n)$ 就能够得到一个开销最小的解。

输入/输出

输入

算法从一个初始棋面状态开始，寻找一个目标状态。算法假设（a）在给定棋面状态下可以尝试所有的可行走法，以及（b）计算棋面状态 n 的评估函数 $f^*(n)$ 。

输出

返回一个走法的序列，表示从初始状态到目标状态的近似开销最小的路径（或者只是输出是否存在一个解）。

假设

如果估算得到的结果和实际结果相差不远，那么A*搜索得到的是开销最小的解。

使用环境

我们使用如下的八数码：

8	1	3
	4	5
2	7	6

图7-11和图7-12是两棵计算出的搜索树。图7-11的搜索树使用的是Nilsson (1971) 提出的GoodEvaluator $f^*(n)$ 函数。图7-12的搜索树使用的是同样由Nilsson提出的WeakEvaluator $f^*(n)$ 函数。浅灰色的棋面状态表示达到目标状态时开放集合的元素。GoodEvaluator和WeakEvaluator都能够得到同样的解，但是使用GoodEvaluator函数在搜索中更加高效。根据两棵树中 $f^*(n)$ 的值，我们希望知道为什么WeakEvaluator需要访问更多的节点。仔细观察在GoodEvaluator产生的搜索树的最初两步，我们可以清楚地看到 $f^*(n)$ 呈递减趋势。而WeakEvaluator的搜索树在走了4步之后才能够减少搜索的候选方向。WeakEvaluator不能很好地差异化棋面状态，事实上，目标状态节点的 $f^*(n)$ 值比初始状态节点以及其子节点的 $f^*(n)$ 都大。

$f^*(n)$ 函数的 $h^*(n)$ 部分需要非常优秀的设计，这是一门工程而不是一门科学。 $h^*(n)$ 必须非常高效，否则，搜索时间将会非常长。很多A*搜索相关的文献都指出， $h^*(n)$ 函数是高度特化的，根据问题的不同而不同。例如在数字地形的寻路（Wichmann和

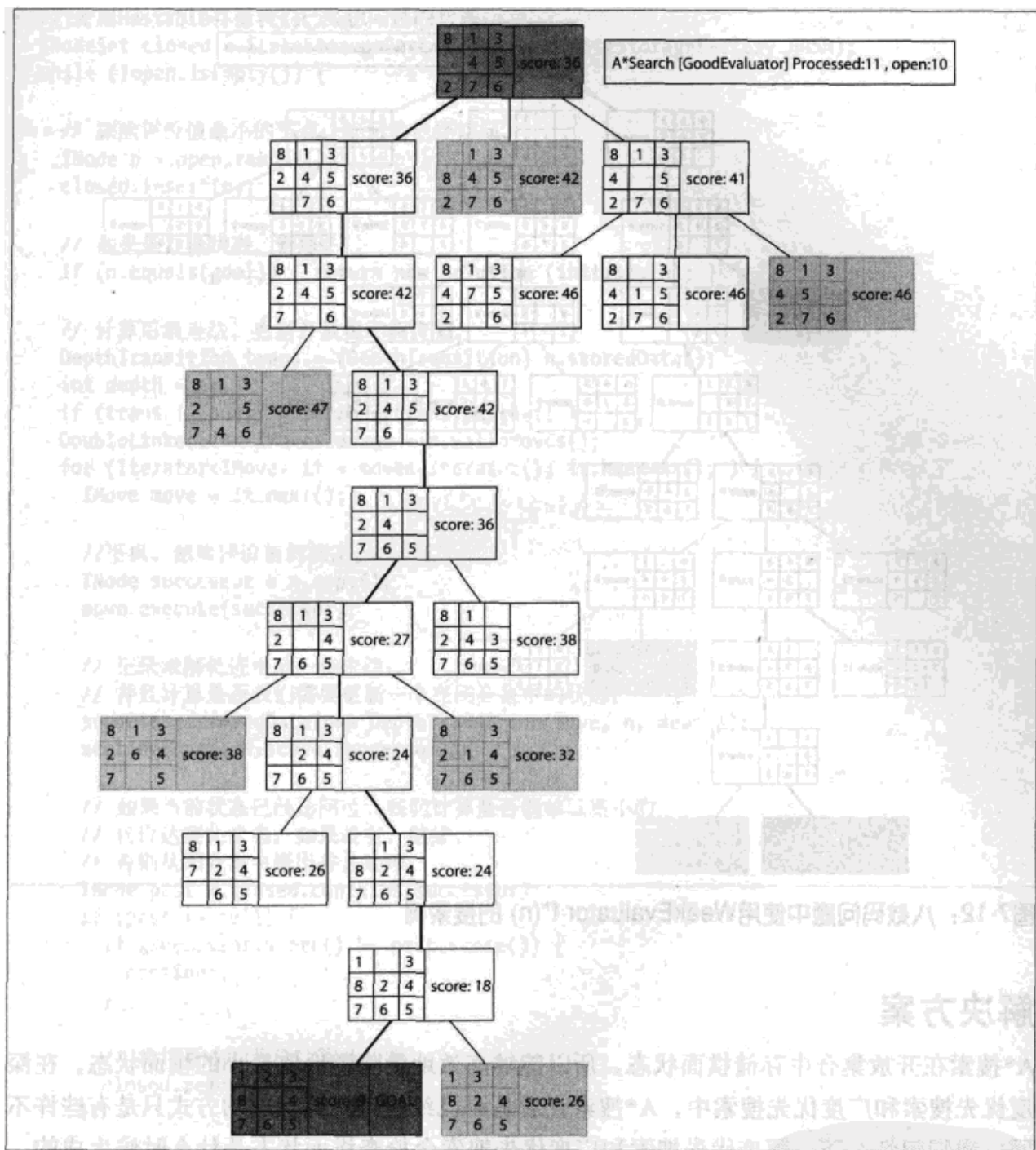


图7-11：八数码问题中使用GoodEvaluator $f^*(n)$ 的搜索树

Wuensche, 2004) 或者是有限资源下的工程排期 (Hartmann, 1999)。Pearl (1984) 写过一篇设计高效启发式函数的文章 (很遗憾已经绝版了)。Korf (2000) 讨论了设计可行 $h^*(n)$ 函数的高级策略。Michalewicz和Fogel (2004) 提出了一种新的解决问题的启发式思路, 不仅仅是只能用于A*搜索。

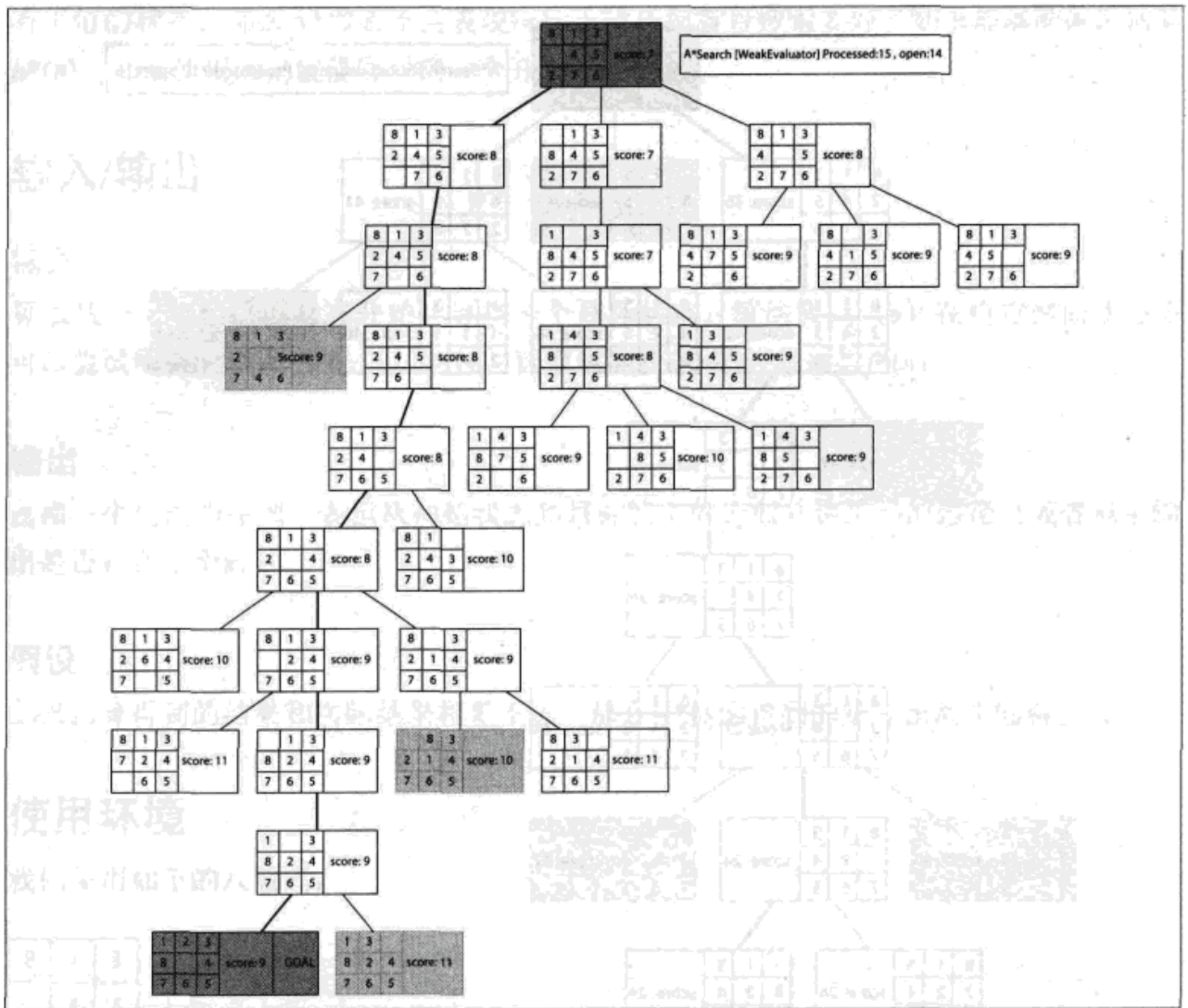


图7-12：八数码问题中使用WeakEvaluator $f^*(n)$ 的搜索树

解决方案

A*搜索在开放集合中存储棋面状态，所以能够高效地删除评价价值最小的棋面状态。在深度优先搜索和广度优先搜索中，A*搜索查看是否已经达到目标状态的方式只是有些许不同。我们回忆一下，深度优先搜索和广度优先搜索会检查棋面状态是什么时候生成的。尤其，A*搜索在从开放集合中删除元素时，会检查是否已经达到目标状态，这样做的目的是确保得到的解是最短路径。例7-5是A*搜索的Java实现。

例7-5：A*搜索实现

```
public Solution search(INode initial, INode goal) {
    // 从初始状态开始。
    INodeSet open = StateStorageFactory.create(StateStorageFactory.TREE);
    INode copy = initial.copy();
    scoringFunction.score(copy);
    open.insert(copy);
```

```

// 使用Hashtable存储我们已经访问过的状态。
INodeSet closed = StateStorageFactory.create(StateStorageFactory.HASH);
while (!open.isEmpty()) {

    // 删除评价值最小的节点，标记为已经访问。
    INode n = open.remove();
    closed.insert(n);

    // 如果是目标状态，返回。
    if (n.equals(goal)) { return new Solution (initial, n); }

    // 计算后继走法，更新开放集和闭合集。
    DepthTransition trans = (DepthTransition) n.storedData();
    int depth = 1;
    if (trans != null) { depth = trans.depth+1; }
    DoubleLinkedList<IMove> moves = n.validMoves();
    for (Iterator<IMove> it = moves.iterator(); it.hasNext(); ) {
        IMove move = it.next();

        //下棋，然后评价新的状态。
        INode successor = n.copy();
        move.execute(successor);

        // 记录求解轨迹中前一步走法，
        // 并且计算是否我们需要更新一个在闭合集中的状态。
        successor.storedData(new DepthTransition(move, n, depth));
        scoringFunction.score(successor);

        // 如果当前状态已经访问过，我们计算是否能够以更小的
        // 代价达到此状态，如果没有，继续，
        // 否则从闭合集中提出并且处理。
        INode past = closed.contains(successor);
        if (past != null) {
            if (successor.score() >= past.score()) {
                continue;
            }

            // 将用更小的代价达到此状态。
            closed.remove(past);
        }

        // 放入开放集中。
        open.insert (successor);
    }
}

// 无解。
return new Solution (initial, goal, false);
}

```

在深度优先搜索和广度优先搜索中，棋面状态都会在处理之后放入闭合集中。由于A*搜索的启发式信息需要计算 $g^*(n)$ ，所以可能存在一种情况，这种情况下，A*搜索需要重新评价已经做出的决定。比如，如果存在一个将要被放入开放集合的棋面状态，其评估

分数要比已经访问过的相同棋面状态评估函数低。这样，A*搜索会将这个状态从闭集中删除，因为我们可能能够得到一个最小耗费的解。

每个棋面状态都存储了一个后链，叫做DepthTransition，记录的是（a）生成此棋面状态的走法，（b）之前的状态以及（c）初始位置开始的深度。在A*搜索中， $g^*(n)$ 通常当作深度。算法多次复制一个棋面状态，用来尝试各种走法。

结论

A*搜索的效果直接依赖于其启发式函数。如果 $h^*(n)$ 一直为0，那么A*搜索不会比广度优先搜索要好。但是如果 $h^*(n) > h(n)$ ，那么A*搜索虽然能够给出一些解，但是也许不能够得到最优解。

如果启发函数 $h^*(n)$ 性能可以接受，那么这时候我们就可以使用A*搜索。如果，那么表示启发式函数是可以接受的。这种限制需要满足两个方面的条件，如果 $h^*(n)$ 返回一个负数（表示有时候棋面状态 n 越过了目标状态），那么 $g^*(n)$ 的效果就会被抵消掉。如果 $h^*(n) > h(n)$ ，那么A*搜索可能找不到最优解。寻找到一个合适的高效 $h^*(n)$ 函数是非常困难的。不过有大量不可接受的 $h^*(n)$ ，它们可以得到可行解，但是并不最优。

如果 $h^*(n)$ 可行，那么A*搜索将会寻找到最优解。对于八数码问题来说，表7-3列出了下述棋面状态下，三个启发式函数的情况。

1	4	8
7	3	
6	5	2

表7-3：比较三个 $h^*(n)$ 评价函数

评价函数名字	$h^*(n)$ 函数的描述	$h^*(n)$ 的评价分数	统计
GoodEvaluator	$P(n)+3*S(n)$ ， $P(n)$ 是每个方块到其目标位置的曼哈顿距离之和。 $S(n)$ 是一个分数序列，依次检查非中部的每个方块，如果一个方块的后继方块不是目标状态中的后继方块，那么得分为2，否则为0，中部方块的得分为1。	$13+3*11=46$	13步的解 闭合集规模：14 开放集规模：12
WeakEvaluator	计算位置错误的方块个数	7	13步的解 闭合集规模：145 开放集规模：110

表7-3: 比较三个 $h^*(n)$ 评价函数 (续)

评价函数名字	$h^*(n)$ 函数的描述	$h^*(n)$ 的评价分数	统计
BadEvaluator	计算以中间方块对称的两个方块差值, 加起来然后求和理想值16的差的绝对值。我们忽略空格子	$(7-0) + (6-8) + (5-4) + (2-1) = 7$ 得分是 $ 16-7 =9$	421步的解 闭合集规模: 2499 开放集规模: 1583

驱动因素

广度优先搜索和深度优先搜索会检查闭合集是否包含某个棋面状态, 所以我们可以用散列表来提高效率。但是, 对于A*搜索来说, 我们可能需要重新评估已经访问过的棋面状态。那么, 会发生什么呢? 回忆一下在深度优先搜索中, 一个达到深度限制的棋面状态, 尽管只离目标状态只有3步, 但是也会被放入闭合集, 并且不会再被处理。在A*搜索中, 如果棋面状态的得分比之前更低, 那么它们就需要重新处理。

A*搜索需要在开放集合中快速找到评价值最小的棋面状态。注意, 深度优先搜索和广度优先搜索从开放集合中得到下一个棋面状态只需要常数时间, 因为它们使用队列或者栈。如果开放集合是一个有序链表, 那么插入棋面状态的性能将会是 $O(n)$, 我们不能使用二叉堆, 因为我们事先不知道有多少棋面状态需要评估。因此我们使用平衡二叉树, 插入棋面状态和得到最小开销的棋面状态的性能都是 $O(\log n)$ 。

分析

A*搜索的行为完全取决于启发函数。最近的研究结果 (Russel和Norvig, 2003) 表明, 如果 $|h(x) - h^*(x)| \leq \log h^*(x)$, 那么性能将会是 $O(d)$, d 表示解的距离, 而不是 $O(b^d)$, b 表示搜索树的分支因子。但是, 这个条件难以满足, 在八数码问题表现很好的GoodEvaluator函数也不能满足这个条件。

随着棋面状态变得越来越复杂, 启发式函数越来越重要, 同时也越来越难以设计。首先启发式函数必须足够高效, 或者能够影响整个搜索过程。但是, 即使是最差的启发式函数都能够较好地剪枝搜索空间。例如, 十五数码问题, 八数码问题的一个扩展, 使用八数码问题的GoodEvaluator函数扩展时, 目标状态如下:

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

初始状态如下:

2	10	8	3
1	6		4
5	9	7	11
13	14	15	12

A*搜索在处理39个棋面状态之后，快速找到了一个15步的解，这个时候开放集合中还有43个棋面状态，如图7-13所示。

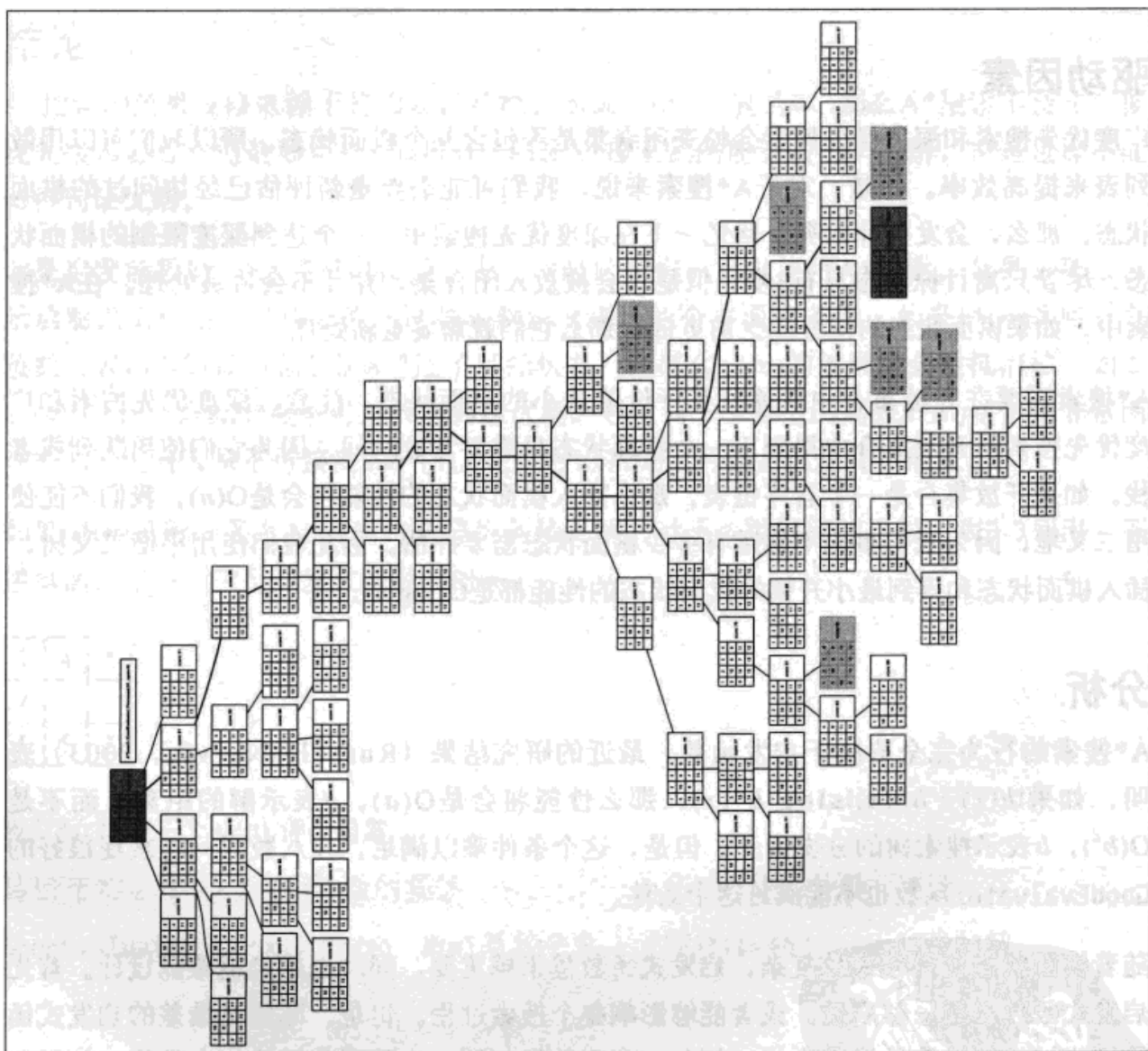


图7-13：十五数码问题的A*搜索树

由于有着15步的限制，深度优先搜索在处理22 136个棋面状态之后，仍然没有找到解。而广度优先搜索在处理172 567个状态（85 213在闭合集中，剩余87 354在开放集中）时，已经耗尽了所有的64MB内存。当然你可以加内存或者增大搜索深度限制，但是这些并不是你无法解出这些问题的理由。

不要被A*搜索这么容易解出十五数码所迷惑，当初始棋面状态更加复杂时，例如：

5	1	2	4
14	9	3	7
13	10	12	6
15	11	8	

A*搜索也耗尽了内存。很明显，这个评估函数并不高效，这个例子有超过 10^{25} 个可能的状态（Korf, 2000）。

变种

有一些算法使用了双向搜索（Kaindl和Kainz, 1997），而不仅仅是从初始状态向前搜索。一开始这种方法由于不能工作，被早期的AI研究人员放弃，Kaindl和Kainz提出了强有力的理由，认为这种方法应该被重新考虑。

一种广为人知的A*搜索变种叫做迭代加深A*(IDA*)，由Korf（1985）提出。它依赖于一系列逐渐扩展的有限制的深度优先搜索。对于每次后继迭代，搜索深度限制都会在前次的基础上增加。IDA*比单独的深度优先搜索或者广度优先搜索要高效得多，因为每次计算出的开销值都是基于实际的走法序列而不是启发式函数的估计。Korf（2000）描述了高效的启发式信息是如何和IDA*结合起来，用于解决十五数码问题，在整个搜索过程中，会对超过4亿个棋面状态进行评估。

Barr和Feigenbaum（1981）提出了一些改进方法，在不能高效地得到一个可接受的 $h^*(n)$ 函数时使用。

相关算法

虽然A*搜索产生了最小耗费的解，但是搜索空间可能会过大以至于无法继续计算。增强A*搜索和处理这些巨大规模的问题的改进思想主要包括：

迭代加深

这个策略重复迭代有深度限制的深度优先搜索，每次迭代都会增加深度限制。这种方法能够选择出在下次迭代中优先被处理的节点，因此减少了不必要的搜索，增加了快速收敛到获胜走法的可能性。同时，由于搜索空间被切分成离散的部分，实时算法能够在尽可能多的空间限制下，在时限内寻找到一个较优解。最先应用到A*搜索算法的是Korf（1985），他发明了IDA*。

转移表

为了避免重复计算，程序员可以对局面状态进行散列，在一个转移表中存储路径长度。如果状态在后面的搜索中会出现，而且当前深度大于先前的深度，那么搜索将会终止。这种方法能够避免搜索一棵低效的子树。

层次化

如果局面状态能够层次化地表示，那么我们可以重新构建一下搜索空间。层次化寻路A*（HPA*）就应用了这种方法（Botea等，2004）。

内存限制

与其在计算时限制搜索空间，程序员可以执行一种“有损”搜索，在搜索的过程中舍弃一些节点，专注于搜索那些被认为和结果相关的区域。简化内存限制A*（SMA*）就是一个例子（Russel，1992）。

Reinefeld和Marstrand（1994）总结了一系列的A*的有趣的扩展。更多的在AI系统中使用A*搜索的信息在教科书和大量的在线资源可以找到相关信息（Barr和Feigenbaum，1981）。

比较

广度优先搜索能够保证找到步数最少的解，但是它可能需要评价规模相当大的移动序列。深度优先搜索试着在每次搜索时能够前进更多步，它可能能够快速得到一个解，但是它也可能在搜索树的某个部分浪费大量的时间，尽管看起来这个部分不可能得到解。

我们认为比较深度优先搜索，广度优先搜索和A*搜索是值得的。使用八数码问题作为样例，我们通过随机移动 n 个方块（ n 从2、4、8和16开始），注意同样的方块不会在一行中移动两次，因为这就等于什么也没做。当 $n \geq 32$ 时，内存耗尽。对于每个棋面状态，我们执行BREADTH-FIRST SEARCH, DEPTH-FIRST SEARCH(n), DEPTH-FIRST SEARCH($2*n$) 和A*搜索。对于每个 n ：

- 我们将计算开放集合和闭合集中的状态数目，从而可以看出算法得到解的效率如何。标记为#的列表示的是多次运行的平均值。
- 一旦找到解，我们将会计算解的步数，这样能够看出得到的路径的质量。标有s的列表示的是多次运行的结果。括号中的数字技术的是在给定深度限制下，没有能够得到解的次数。

表7-4综合了1000次运行的结果，列出了两个统计值：（a）生成的搜索树的平均状态数目，（b）同样的结果的平均步数。

表7-4：比较搜索算法

n	#A*	#BFS	#DFS(n)	#DFS(2n)	sA*	sBFS	sDFS(n)	sDFS2(n)
2	5.0	4.5	3.0	6.4	2	2	2	2
3	7.0	13.4	7.0	26.8	3	3	3	3
4	9.0	25.6	12.3	66.1	4	4	4	5.0

表7-4: 比较搜索算法 (续)

n	#A*	#BFS	#DFS(n)	#DFS(2n)	sA*	sBFS	sDFS(n)	sDFS2(n)
5	11.1	46.3	21.2	182.5	5	5	5	5.9
6	12.5	77.2	31.7	317.8	6	6	6	9.5 (45)
7	14.9	136.5	57.4	751.4	6.8	6.8	6.92	9.6 (279)
8	17.1	220.5	85.6	1095.2	7.7	7.7	7.9 (40)	13 (209)
9	22.0	367.9	147.2	2621.7	8.8	8.7	8.8 (75)	13.1 (355)
10	25.5	578.8	211.7	3152.9	9.8	9.6	9.8 (236)	16.5 (316)
11	33.1	926.4	296.6	6723.3	10.6	10.4	10.6 (431)	17.1 (369)
12	42.3	1445.7	440.8	5860.5	11.9	11.3	11.6 (350)	20.7 (402)
13	56.6	2351.3	558.9	12483.1	13.2	12.2	12.3 (615)	21.7 (313)
14	60.7	3579.7	900.3	14328.1	14.5	13.0	13.3 (593)	25.1 (259)

注意看随着 n 的增加, 盲目搜索的搜索树的规模指数增长, 但是A*搜索的搜索树规模还是可控的。更精确地说, 这些盲目搜索的增长率如下:

$$\text{DFS2}(n) \approx 0.2867 * n^{4.0722}$$

$$\text{DFS}(n) \approx 0.2405 * n^{2.9517}$$

$$\text{BFS}(n) \approx 0.2585 * n^{3.4041}$$

广度优先搜索一直能够找到解的最短路径, 但是注意即使A*搜索检查更少的棋面状态, 但是得到的解质量也不错 (由于GoodEvaluator启发函数)。在达到30步之后, 搜索树的增长率达到 $O(n^{1.5147})$, 虽然不是线性, 但是相比起盲目搜索, 这个规模已经相当小了。这些增长率函数的指数部分依赖于问题的分支因子。表7-4的图形化表示如图7-14所示。

广度优先搜索能够找到最短路径, 而A*搜索找到的路径也差不多是最短的。最后, 我们看看水平效应是如何阻止深度优先搜索解决问题的 (回忆一下那些离目标状态只有两三步远的状态却被加入闭合集)。事实上, 在1000次实验中, 深度优先搜索使用最大深度限制为13时, 60%的实验都失败了。

分析主要关注于搜索状态的数目, 这是决定搜索效率的主要因素。当三种搜索都有可能搜索指数级的状态时, 由于 $h^*(n)$ 函数计算出的启发式信息, A*搜索将搜索最少的状态。

解决滑动 n^2-1 个方块这样的问题不仅仅只是有寻路这种方法。Parberry (1995) 提出了一种别出心裁的方法, 使用分治思想。也就是说, 给定一个 $n \times n$ 数码, $n > 3$, 首先完成

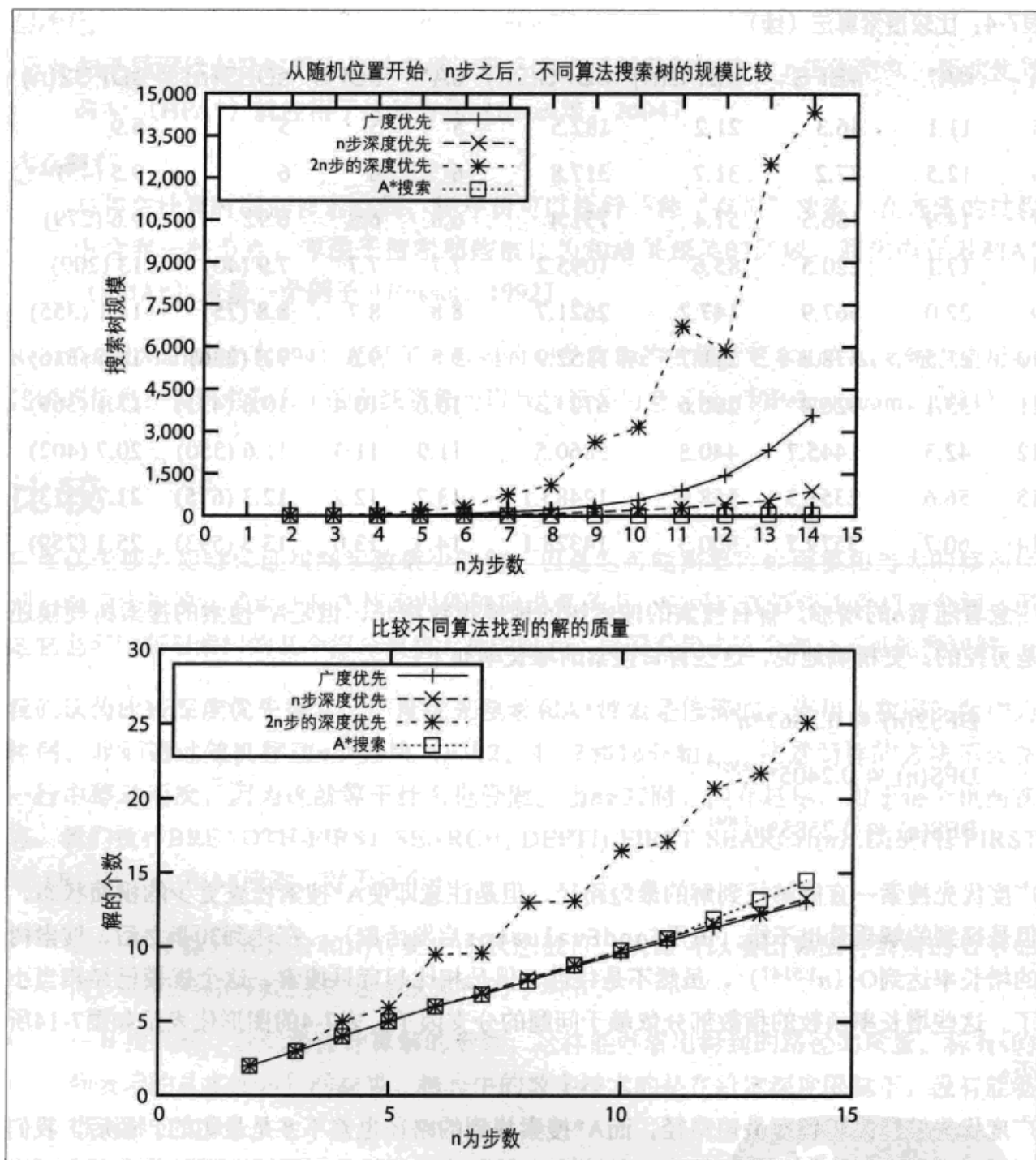


图7-14：随机位置的搜索树对比

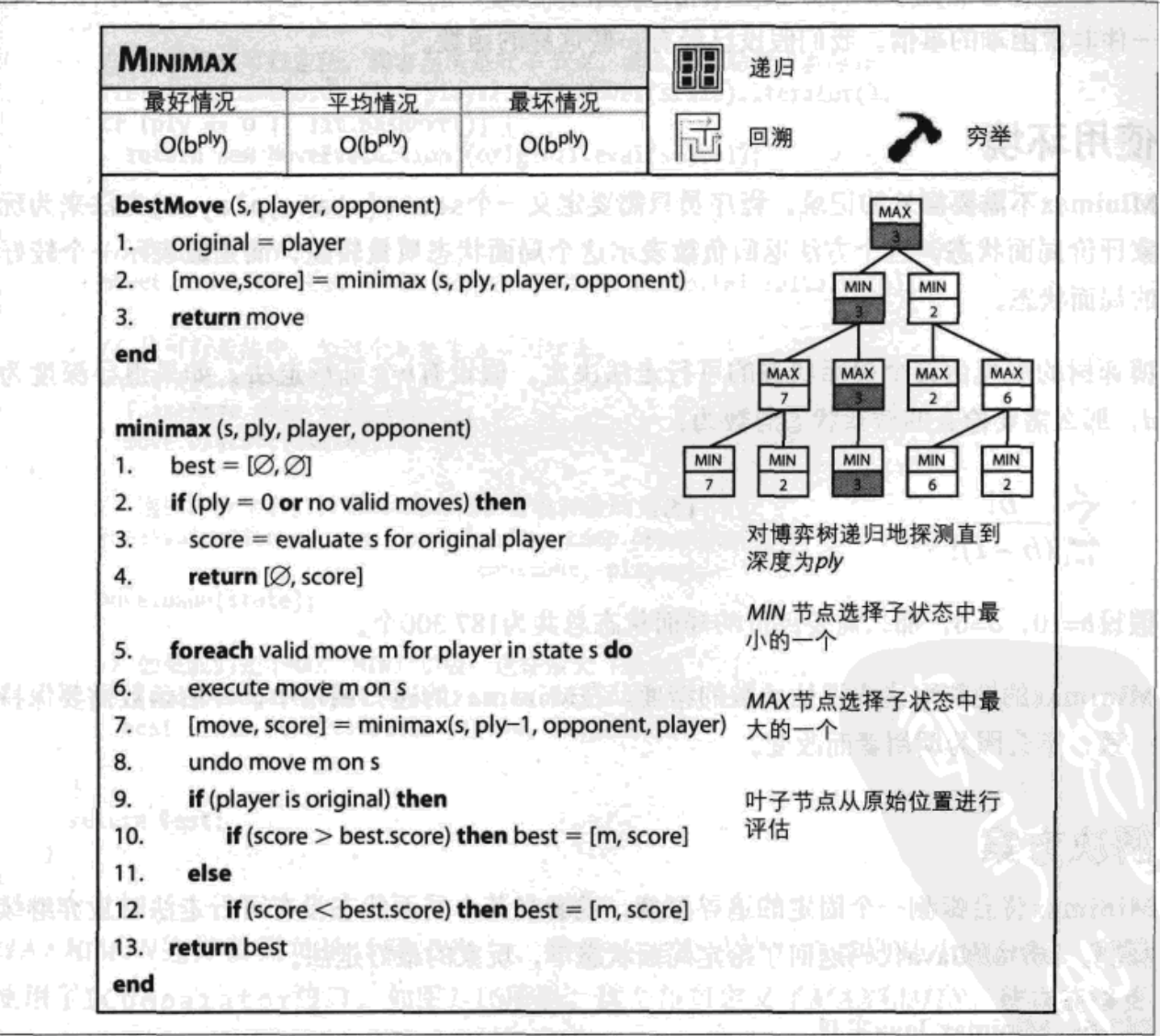
最左边的列和最上部的行，然后递归解决 $(n-1)^2-1$ 数码问题。我们得到一个 3×3 的子问题，这是可以简单地使用穷举法来解决。这个方法保证能在 $5 \cdot n^3$ 步内找到解。

我们现在结束对搜索树的讨论。本章剩余的算法都是基于博弈树的。

Minimax

玩家希望搜索程序能够最大化从指定位置开始走，获得胜利的概率（或者至少要保平）。程序不会仅仅考虑当时的局面状态以及当时的可行走法，还需要考虑对手会采取的任何对策。程序需要假设对手会采取最完美的走法，并且不会犯错。程序假设存在一个评估函数 $score(state, player)$ ，返回一个整数，表示局面状态的得分，分数越小（可以为负）表示状态越差。

扩展博弈树时也需要考虑 n 步之后的局面状态。树的每一级轮流标注为MAX层级（目标是最大化局面状态的得分，使得对玩家有利）和MIN层级（目标是最小化局面状态的得分，使得对对手有利）。在每一级，如果是玩家走，那么程序选择最大化 $score(state, initial)$ 的走法，如果是对手走的话，那么程序将会假设对手足够聪明，所以选择最小化 $score(state, initial)$ 的走法。Minimax算法如图7-15所示。



输入/输出

输入

算法从博弈树中的一个初始位置开始，假设它能够（a）选择某个局面状态的所有可行走法，以及（b）评价某个局面状态，表示从玩家来看这个状态的质量。得分越少则质量越差。算法会预测固定步数，这个数目叫做追寻深度。

输出

返回所有可行走法中最能使得玩家赢得比赛的走法，由评估函数决定。

假设

评估局面状态是非常复杂的，程序员需要依靠评估函数来选择最好的局面状态。事实上，在设计智能程序中，开发一个精确的评估函数，例如象棋、跳棋，或者黑白棋，是一件非常困难的事情。我们假设已经有一些这样的函数。

使用环境

Minimax不需要额外的记录。程序员只需要定义一个`score(state,player)`方法来为玩家评价局面状态，这个方法返回负数表示这个局面状态质量较差，而正数表示一个较好的局面状态。

博弈树的规模由每个局面状态的可行走法决定。假设有 b 个可行走法。如果追寻深度为 d ，那么需要检查的游戏状态总数为：

$$\sum_{i=1}^d \frac{b!}{(b-i)!}$$

假设 $b=10$ ， $d=6$ ，那么需要评价的局面状态总共为187 300个。

Minimax的性能取决于评估函数的精度。在Minimax的递归调用中，评估函数需要保持一致，不会因为调用者而改变。

解决方案

Minimax将会探测一个固定的追寻深度，或者当某个局面状态没有可行走法时放弃继续探测。例7-6的Java代码返回了给定局面状态下，玩家的最好走法。

例7-6: Minimax Java实现

```
public class MinimaxEvaluation implements IEvaluation {
```



```

IGameState state; /** 在搜索中会修改的状态 */
int ply;           /** 追寻的深度 */
IPlayer original; /** 从这个角度出发评价状态 */

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;
}

private MoveEvaluation minimax (int ply, IComparator comp,
                                IPlayer player, IPlayer opponent) {

    // 如果没有可行走法，或者当前是叶子节点，那么返回局面状态得分。
    Iterator<IGameMove> it = player.validMoves(state).iterator();
    if (ply == 0 || !it.hasNext()) {
        return new MoveEvaluation (original.eval(state));
    }

    // 尝试更改这个下界。
    MoveEvaluation best = new MoveEvaluation (comp.initialValue());

    // 从可行走法中，为这个玩家生成局面状态。
    while (it.hasNext()) {
        IGameMove move = it.next();
        move.execute(state);

        // 递归评价状态。计算Minimax值，交换评价的角色。
        MoveEvaluation me = minimax (ply-1, comp.opposite(),
                                     opponent, player);
        move.undo(state);

        // 如果我们处于MAX (MIN) 层级，选择最大 (最小) 的子节点。
        if (comp.compare(best.score, me.score) < 0) {
            best = new MoveEvaluation (move, me.score);
        }
    }
    return best;
}
}

```

*MAX*和*MIN*选择器简单地计算出得分，然后按需选出最大的或则最小的分数。实现使用了*IComparator*接口，如图7-16所示。这个接口定义了*MAX*和*MIN*，使得能够更加正确地为玩家挑选出最优走法。*opposite()*方法是在*MAX*和*MIN*选择器中切换。

`initialValue()`返回这些比较器中最差的分数，根据所处层级属于MAX或是MIN，我们得到不同的实际值。

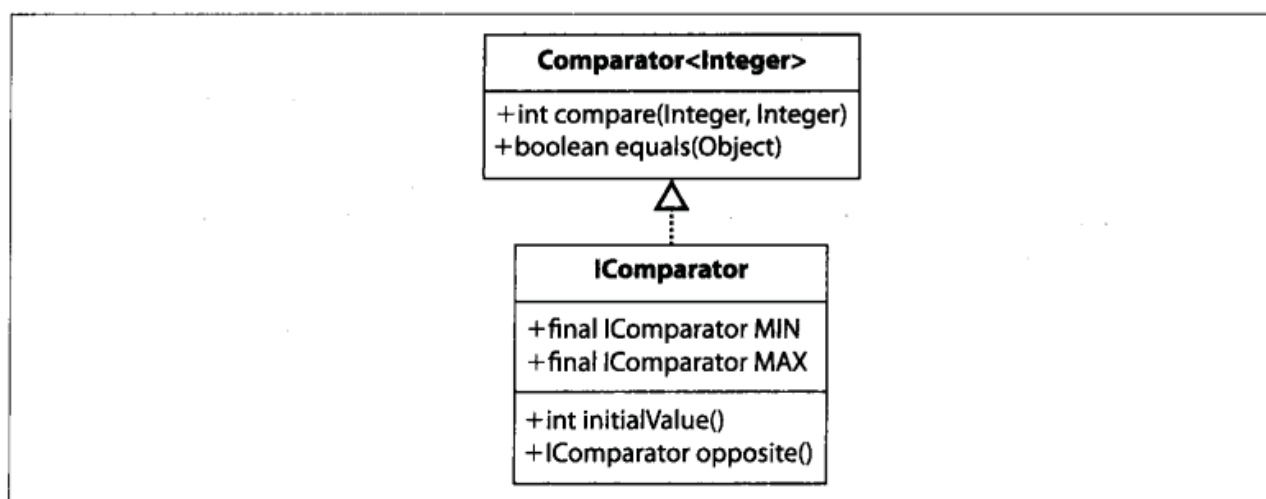


图7-16: IComparator接口重载了MAX和MIN操作符

结论

算法不能处理在递归搜索中产生的巨大数量的局面状态。在国际象棋中，在棋面上，平均走法的数目是30个（Laram e, 2000），也就是说，仅仅向前预计算5步（即 $b=30$ ， $d=5$ ），那么就需要评估25 137 931个棋面状态。这个值是这样计算出来的：

$$\sum_{i=0}^d b^i$$

Minimax可以利用局面状态的对称性（例如棋盘的旋转或者反转），缓存已经检查过的状态（以及它们的分数），但是节省的开销因问题而异。

分析

图7-17是一字棋游戏中，玩家O采取Minimax搜索时的追寻深度为2的轨迹。交替的MAX和MIN层级告诉玩家，要避免被秒杀，那么在左上角画一个O是唯一走法。注意这里会扩展出所有可能的棋面状态，即使确定玩家X可以稳赢。

博弈树的深度是固定的，我们可以根据追寻长度的走法序列，得到一系列可能的局面状态。当每个局面状态都有固定的 b 个走法时，追寻深度 d 的Minimax算法需要搜索的局面状态的总数大约是 $O(b^d)$ ，很明显是指数级增长。

我们看看图7-17的结果，肯定有办法能够消除掉无用的局面状态。玩家MAX试着最大化局面状态的得分，因为每一步都会被评价，玩家MAX计算出他能够保证的最大值

变种

如果博弈树能够完全存储在内存中，那么可以不用设置追寻深度。

NegMax

NegMax算法不再使用Minimax的MAX和MINI层级，而是每一级都使用同样的方法。这种思想也形成了另外一种算法，AlphaBeta，我们在稍后会讨论这个算法。在Minimax中，局面状态是一直从玩家的视角来观察，然后选择走法（需要存储棋子的信息以供评价函数使用）。

此算法不再将博弈树划分为多个层级，NegMax对走法采取一致的评价策略，从最坏的角度来评价子节点。从本质上来说，玩家走完之后，对手肯定会试着做出最好的决定，因此，算法指导玩家选择最好的走法，限制对手的可行走法。如果你比较图7-15和图7-18的伪代码，你将会看到两个几乎相同的博弈树，唯一的区别就是局面状态是如何被评分的。注意，对玩家来说，两个可行走法中的第一个是最好的，因为这个走法对对手的限制最多。

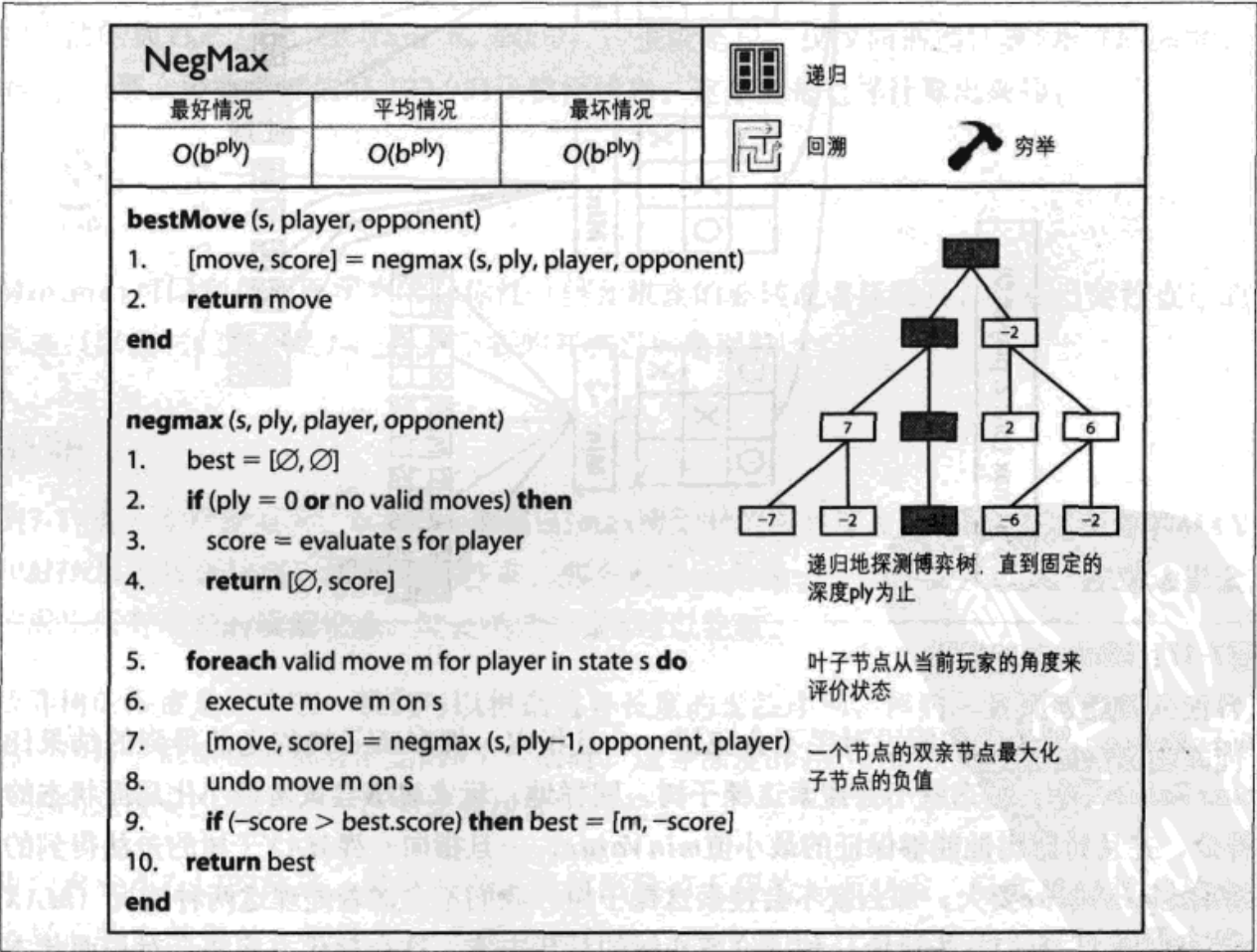


图7-18：NegMax详解

输入/输出

输入输出同Minimax。

使用环境

使用环境也同Minimax。

解决方案

在例7-7中，每一个MoveEvaluation的得分是简单地以走棋的玩家角度来评价棋面状态。这样能够简化算法的实现。

例7-7: NegMax实现

```
public class NegMaxEvaluation implements IEvaluation {

    IGameState state;          /** 在搜索中将会修改的状态。*/
    int ply;                   /** 追寻深度。*/

    public NegMaxEvaluation (int ply) {
        this.ply = ply;
    }

    public IGameMove bestMove (IGameState s,
                               IPlayer player, IPlayer opponent) {
        this.state = s.copy();
        MoveEvaluation me = negmax(ply, player, opponent);
        return me.move;
    }

    public MoveEvaluation negmax (int ply, IPlayer player, IPlayer opponent) {

        // 如果没有可行走法，或者当前是叶子节点，那么返回棋面状态得分。
        Iterator<IGameMove> it = player.validMoves(state).iterator();
        if (ply == 0 || !it.hasNext()) {
            return new MoveEvaluation(player.eval(state));
        }

        // 尝试更改这个下界。
        MoveEvaluation best = new MoveEvaluation (MoveEvaluation.minimum());

        // 得到这个玩家的走法，并且生成这些走法能够达到的状态。
        // 选择负分最大的子节点。
        while (it.hasNext()) {
            IGameMove move = it.next();
            move.execute(state);

            // 使用一直的negmax函数评价位置。
            MoveEvaluation me = negmax (ply-1, opponent, player);
```

```

        move.undo(state);

        if (-me.score > best.score) {
            best = new MoveEvaluation (move, -me.score);
        }
    }

    return best;
}
}

```

结论

NegMax是非常实用的算法，如果需要，可以扩展成AlphaBeta算法。因为每个棋面的得分都是负数，我们只需要仔细地区分胜利和失败状态。尤其是最小值必须是最大值的负数。注意Integer.MIN_VALUE（在Java中，这个定义为0x80000000或者-2 147 483 648）并不是Integer.MAX_VALUE（在Java中，这个定义为0x7fffffff或者2 147 483 647）的负值。因此，我们将最小值定义为Integer.MIN_VALUE+1，这个值可以调用静态函数MoveEvaluation.minimum()获得。从完整性来看，我们也提供了MoveEvaluation.maximum()函数。

分析

图7-19是一字棋游戏中，玩家O采取NegMax搜索时的追寻深度为2的轨迹。注意这里会扩展出所有可能的棋面状态，即使确定玩家X可以稳赢时。每个叶子局面状态的得分都是从玩家的角度出发。我们看到初始局面状态的得分是-2，因为这是所有其子节点的最大负分。

NegMax探测的状态数目和Minimax一样，如果追寻深度为 d ，每个局面状态的可行走法为 b 个，那么数目近似于 b^d 。

最后我们可以看到NegMax是如何处理博弈树的叶子节点的。你同样可以在例7-7的代码中看到，这些叶子状态节点都是从玩家的角度出发，也就是说双亲节点从叶子节点选择的MoveEvaluation只是简单的最大化这些叶子节点的得分。

NegMax的操作好似流水线一般，它不需要来回切换MAX或者MIN层级。我们回忆一下Minimax是如何要求评估函数要从将要走棋的玩家的观点来进行评估的。在NegMax中，局面状态是基于走棋之后的玩家的视角。因为算法是选择“负分最大的子节点”。

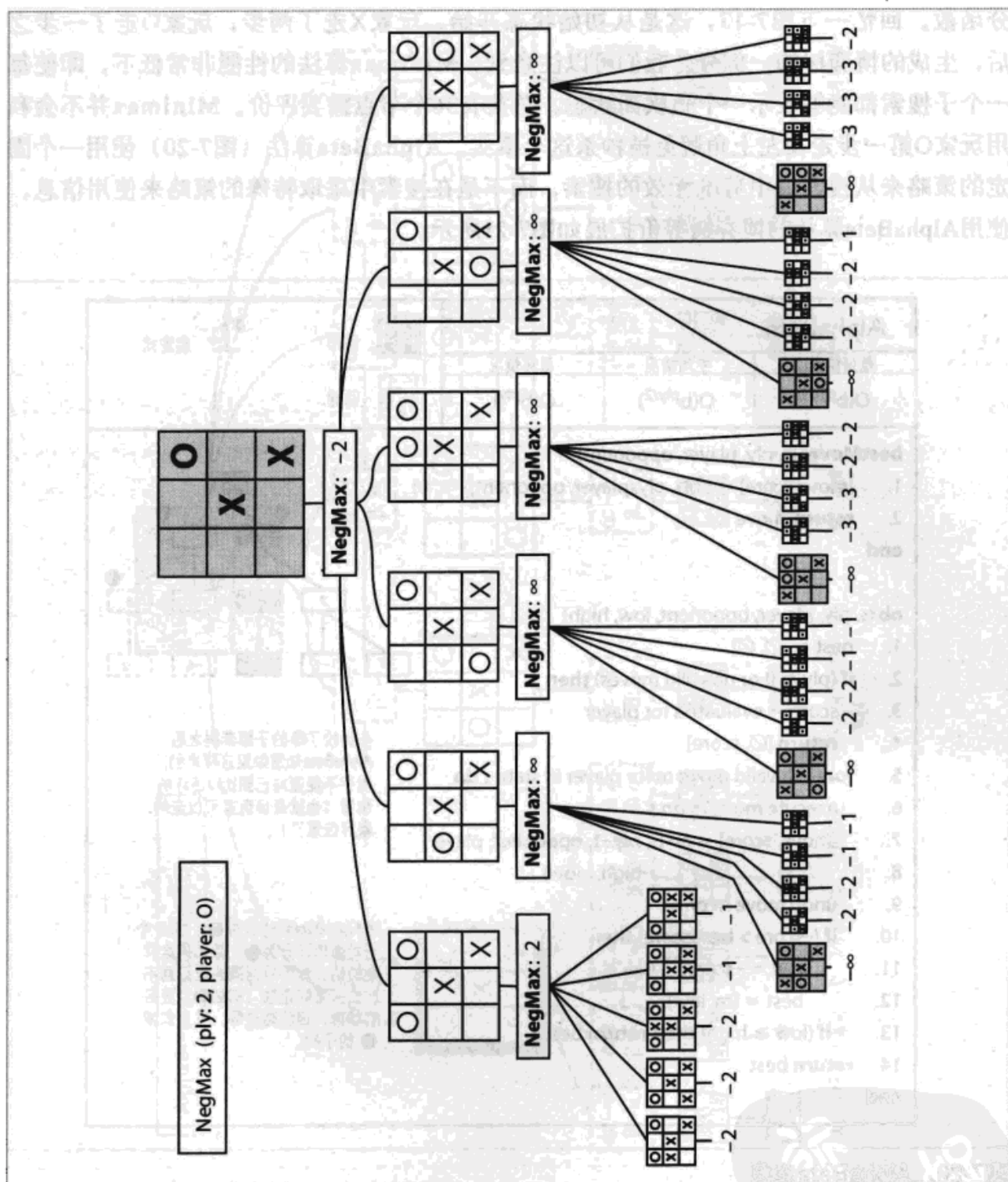


图7-19: NegMax探测

AlphaBeta

在考虑到对手的可能走法之后，Minimax算法能够较为恰当地找出玩家的最优走法。但是，在生成博弈树时，这个信息却没有使用！我们看看早先介绍的BoardEvaluation评

分函数。回忆一下图7-17，这是从初始状态开始，玩家X走了两步，玩家O走了一步之后，生成的博弈树的一部分。我们可以注意到，Minimax算法的性能非常低下，即使每一个子搜索都能够表示一个负棋面状态。总共有36个节点需要评价。Minimax并不会利用玩家O第一步走在左上角避免被秒杀这个事实。AlphaBeta算法（图7-20）使用一个固定的策略来从搜索树中剪除无效的搜索，而不是在搜索中采取特殊的策略来使用信息。使用AlphaBeta算法的博弈树等价扩展如图7-21所示。

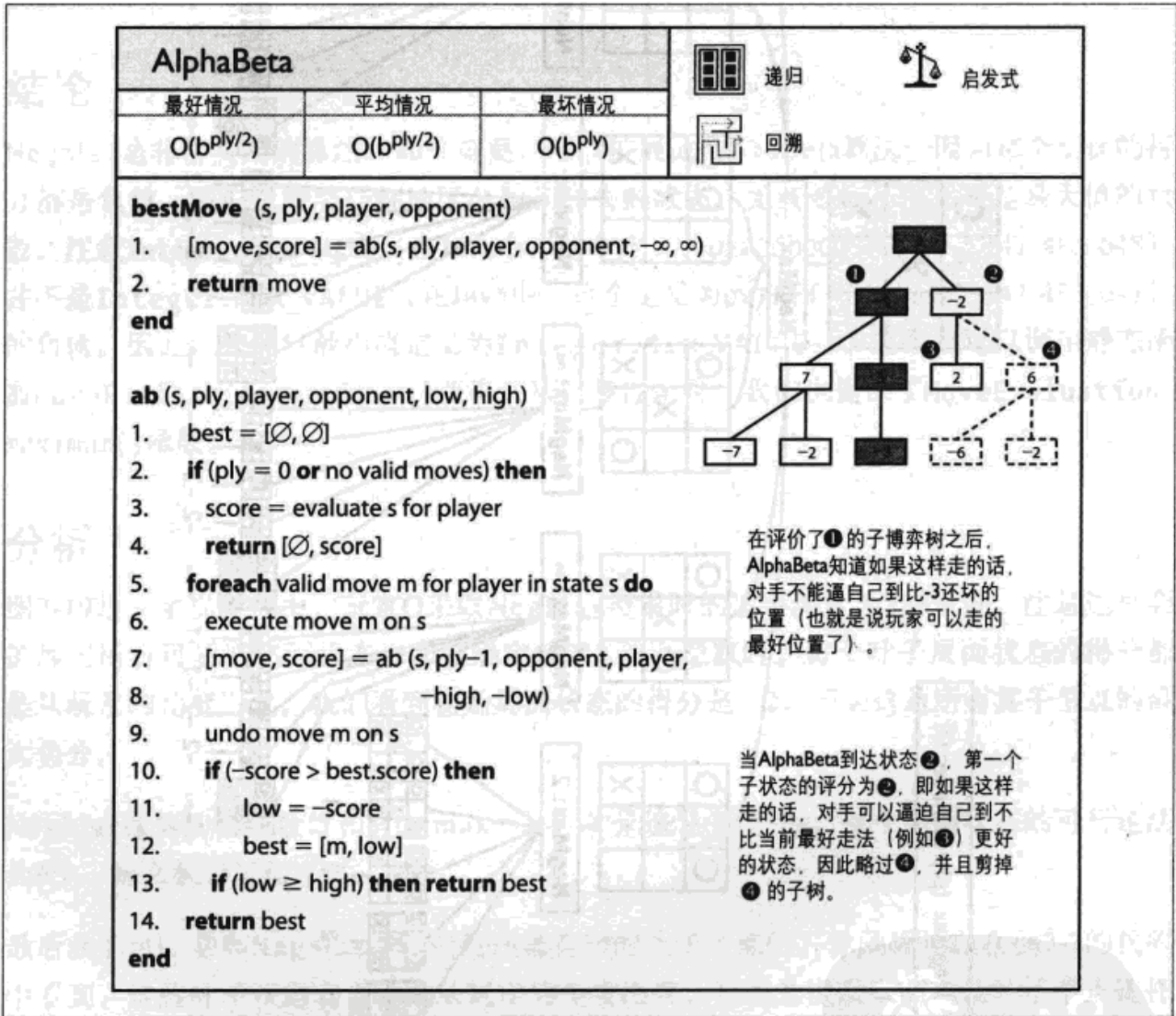


图7-20：AlphaBeta详解

AlphaBeta算法也是搜索最好的走法，它能够记住如果玩家O走在左上角，那么玩家X的得分不会超过2这样的信息。对于玩家O的后续每一步，AlphaBeta会决定是否玩家X有至少一步这样的走法，这步走法能够比玩家O第一步要好。因此，博弈树只扩展了16个节点（相比Minimax节省了50%的开销）。更重要的是，如果这个部分的结果无关紧要，那么算法就不需要预计算可能的走法并且修改状态。

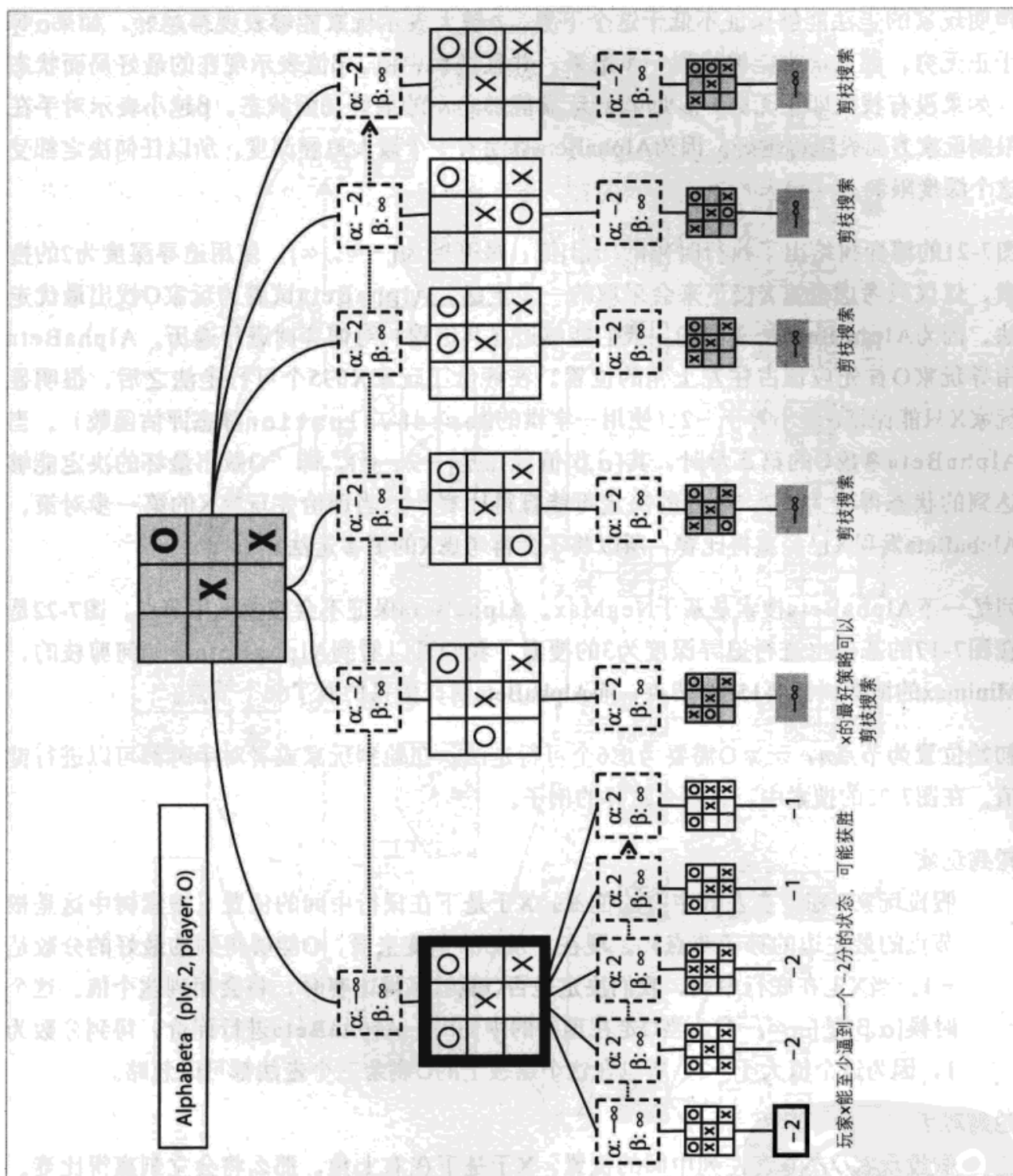


图7-21: AlphaBeta追寻深度为2时的搜索

AlphaBeta选择Minimax也同样会选择的走法，但是会节省大量的开销。和本章之前描述的其他寻路算法一样，AlphaBeta算法也假设对手不会犯错，这样避免了搜索那些对局面没有影响的部分。

AlphaBeta递归地搜索博弈树，并且维护两个值， α 和 β ，如果 $\alpha < \beta$ ，那么表示玩家 α 有胜利的机会。值 α 表示已经为玩家找到的最坏局面状态（如果没有找到即负无穷），并且

声明玩家的走法能够保证不低于这个下界。 α 越大表示玩家能够表现得越好，如果 α 等于正无穷，那么玩家能够找到一步绝杀，并且搜索中值。 β 值表示现在的最好局面状态（如果没有找到即正无穷），即我们玩家能够达到的最好局面状态。 β 越小表示对手在限制玩家方面表现得越好。因为AlphaBeta算法有一个最大追寻深度，所以任何决定都受这个深度限制。

图7-21的博弈树给出了执行时候的 $[\alpha, \beta]$ 值，最开始为 $[-\infty, \infty]$ 。使用追寻深度为2的搜索，仅仅只考虑玩家X接下来会采取的一步走法，AlphaBeta试着为玩家O找出最优走法。因为AlphaBeta是递归的，我们能够追寻其进程，对博弈树进行遍历。AlphaBeta指导玩家O首先应该占住左上角的位置。在评价了玩家X的5个可行走法之后，很明显玩家X只能保证 α 最小等于-2（使用一字棋的BoardEvaluation静态评估函数）。当AlphaBeta考虑O的第二步时，其 $[\alpha, \beta]$ 值现在是 $[-2, \infty]$ ，即“O做出最坏的决定能够达到的状态得分为-2，最好的决定可能赢得比赛”。当评价完玩家X的第一步对策，AlphaBeta发现X已经赢得比赛，所以将不会再考虑X的更多走法。

回忆一下AlphaBeta搜索是基于NegMax。AlphaBeta保证不会搜索无用节点。图7-22是在图7-17的基础上进行追寻深度为3的搜索，我们可以看到AlphaBeta是如何剪枝的，Minimax的博弈树需要156个节点，而AlphaBeta博弈树只扩展了66个节点。

初始位置为节点n，玩家O需要考虑6个可行走法。在轮到玩家或者对手时都可以进行键值。在图7-22的搜索中，有两个这样的例子。

轮到玩家

假设玩家O在下在左列中间的位置，X于是下在顶行中间的位置（搜索树中这是根节点的最左边的孙子节点）。现在，从O的角度来看，O能够得到的最好的分数是-1。当X走在底行中间，我们决定是否O能够赢得比赛时，将会用到这个值。这个时候 $[\alpha, \beta]$ 是 $[-\infty, -1]$ 。当O走在顶行的中间时，AlphaBeta进行评价，得到分数为1，因为这个值大于-1，所以在这个层级上的O剩余三个走法都可以忽略。

轮到对手

假设玩家O在下在左列中间的位置，X于是下在右上角，那么将会立刻赢得比赛。AlphaBeta将不会考虑X的其他两个可行走法，因为在以玩家O上一步走法为根节点的子树中，其剩余的搜索节点都会剪除。

搜索将会剪掉那些确定会失败的子树。当基于Minimax扩展出AlphaBeta时，存在两种剪枝的方法， α 剪枝和 β 剪枝，当基于NegMax扩展出简单的AlphaBeta时，那么只有一种剪枝的方法。因为AlphaBeta是递归的，那么 $[\alpha, \beta]$ 表示玩家的获胜机会大小， $[-\beta, -\alpha]$ 表示对手获胜机会的大小。在递归调用AlphaBeta时，会轮流从玩家和对手的角度来考虑问题。AlphaBeta会返回Minimax（如果是基于NegMax扩展，返回和NegMax相同的结

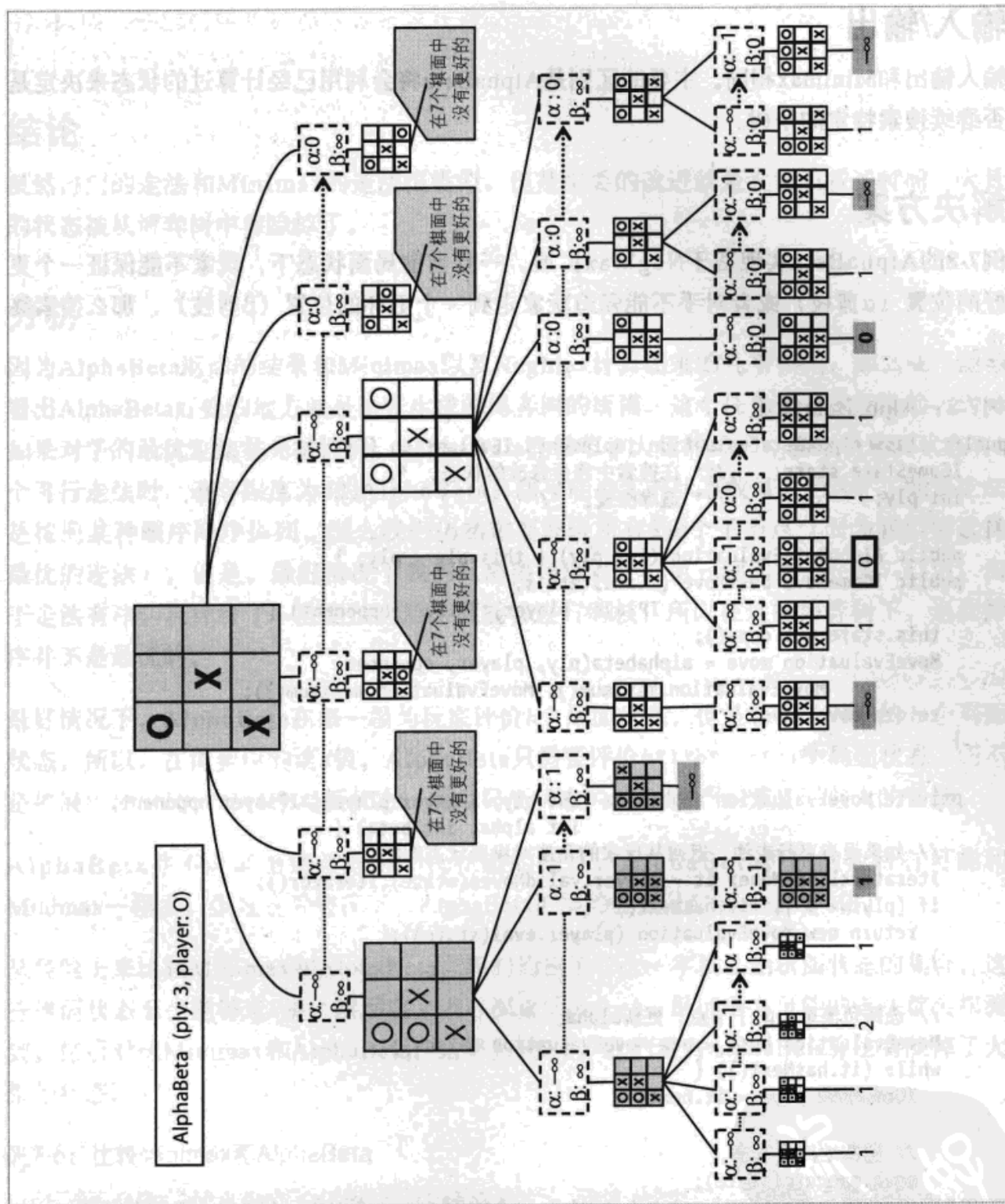


图7-22: AlphaBeta的追寻深度为3的搜索

果) 相同的结果, 但是它只会很少地扩展博弈树。AlphaBeta仍然使用深度的限制, 这样来看, 它的行为非常类似于深度优先搜索。

输入/输出

输入输出和Minimax相同。主要的区别是AlphaBeta将会利用已经计算过的状态来决定是否继续搜索特定的子树。

解决方案

例7-8的AlphaBeta实现基于NegMax扩展。一旦当前局面状态下，玩家不能保证一个更好的位置（ α 剪枝）或者对手不能强迫玩家走到一个更坏的位置（ β 剪枝），那么搜索终止。

例7-8: AlphaBeta实现

```
public class AlphaBetaEvaluation implements IEvaluation {
    IGameState state;    /** 在搜索中将会修改的状态。 */
    int ply;             /** 追寻深度。 */

    public AlphaBetaEvaluation (int ply) { this.ply = ply; }
    public IGameMove bestMove (IGameState s,
                                IPlayer player, IPlayer opponent) {
        this.state = s.copy();
        MoveEvaluation move = alphabeta(ply, player, opponent,
                                         MoveEvaluation.minimum(), MoveEvaluation.maximum());
        return move.move;
    }

    private MoveEvaluation alphabeta (int ply, IPlayer player, IPlayer opponent,
                                       int alpha, int beta) {
        // 如果没有可行走法，返回从玩家的角度对棋面状态的评分。
        Iterator<IGameMove> it = player.validMoves(state).iterator();
        if (ply == 0 || !it.hasNext()) {
            return new MoveEvaluation (player.eval(state));
        }

        // 选择负值最大的子节点，更新alpha。
        MoveEvaluation best = new MoveEvaluation (alpha);
        while (it.hasNext()) {
            IGameMove move = it.next();

            // 递归地评价状态。
            move.execute(state);
            MoveEvaluation me = alphabeta (ply-1, opponent, player, -beta, -alpha);
            move.undo(state);

            // 如果某个走法能够更新alpha，那么就从这个走法继续走下去。
            if (-me.score > alpha) {
                alpha = -me.score;
                best = new MoveEvaluation (move, alpha);
            }
            if (alpha >= beta) { return best; } // 不会有更好的结果了。
        }
        return best;
    }
}
```


}
}

结论

虽然得到的走法和Minimax的走法很类似，但是主要的改进就是大量节省了时间，大量的状态被从博弈树中剪除掉了。

分析

因为AlphaBeta返回的结果和Minimax以及NegMax计算出来的几乎相同，那么唯一能够看出AlphaBeta好处的地方就是检查生成的博弈树的规模。这个任务是非常复杂的，因为如果对手的最优走法首先被评价，AlphaBeta能够节省大量的时间。当每个局面状态有 b 个可行走法时，追寻深度为 d 的AlphaBeta算法可能搜索的状态数目大约为 b^d 。如果走法是按照某种顺序降序排列，那么我们仍然需要评价所有的 b 个子节点（因为我们要选择最优的走法），但是，最好情况下我们只需要评价对手的第一种走法。在图7-21中，由于走法有序，在评价了一些走法之后，就能够进行剪枝，所以在这棵博弈树下，走法排序并不是最优的。

最好情况下，AlphaBeta在每一级为玩家评价 b 个局面状态，但是只评价对手的一个局面状态。所以，在博弈树的第 d 级，AlphaBeta只需要评价 $b*1*b*\dots*b*1$ 个局面状态，而不是扩展出 $b*b*b*\dots*b*b$ 个局面状态。所以局面状态的总数是 $b^{d/2}$ ，节省了巨大的开销。

AlphaBeta并不会试图简单地最小化状态的数目，AlphaBeta扩展的数目有可能和Minimax一样多，但是这种情况下，AlphaBeta扩展的深度是Minimax的两倍。

从经验上来比较Minimax和AlphaBeta，我们构建了一个一字棋初始棋面状态的集合，这些棋面状态至少能够走 k 步。然后定义追寻深度为 $d=9-k$ ，保证所有可能的走法都会探测到，然后对比Minimax和AlphaBeta。结果见表7-5。我们看到AlphaBeta算法剪除掉了大量的状态。

表7-5：比较Minimax和AlphaBeta

k	Minimax状态数	AlphaBeta状态数	减少程度	波动范围
1	549,945	27,565	95%	$\pm 1.3\%$
2	549,936	47,508	91%	$\pm 6.8\%$
3	549,864	112,086	80%	$\pm 10.2\%$

每个比较都说明了AlphaBeta的巨大改进。而且有些情况能够解释为什么AlphaBeta如此优秀，例如这个局面状态：

X	O	

AlphaBeta只需扩展47的局面状态（Minimax需要扩展8232个，节省了99.4%）来告知玩家X应该选择中间的方块。但是，能够得到这样大的性能改进的前提是可行走法有序，并且最优走法排在最前面。因为我们的一字棋的解不会排序走法，可能会得到一些异常结果。例如，将上面的棋面旋转180°：

	O	X

那么AlphaBeta只需要探测960个局面状态（节省了88.3%）。

参考文献

Barr, Avron and Edward A. Feigenbaum, *The Handbook of Artificial Intelligence*. Willian Kaufmann, Inc., 1981.

Berlekamp, Elwyn and David Wolfe, *Mathematical Go: Chilling Gets the Last Point*. A. K Peters Ltd., May 1997.

Botea, A., M. M ller, and J. Schaeffer, "Near Optimal Hierarchical Path-finding," *Journal of Game Development*, 1(1), 2004, <http://www.jogd.com/Vol1issue1.html>.

Hartmann, Sonke, *Project Scheduling Under Limited Resources: Models, Methods, and Applications*. Springer, 1999.

Kaindl, Hermann and Gerhard Kainz, "Bidirectional Heuristic Search Reconsidered," *Journal of Artificial Intelligence Research*, Volume 7: 283 – 317, 1997.

Korf, Richard E., "Depth-First Iterative-Deepening: An Optimal Admissible Tree Search," *Artificial Intelligence*, Volume 27: 97 – 109, 1985, <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.91.288>.

Korf, Richard 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.

- Laram e, Fran ois Dominic, "Chess Programming Part IV: Basic Search," GameDev.net, August 26, 2000, <http://www.gamedev.net/reference/articles/article1171.asp>.
- Michalewicz, Zbigniew and David B. Fogel, *How to Solve It: Modern Heuristics*, Second Edition. Springer, 2004.
- Nilsson, Nils, *Problem-Solving Methods in Artificial Intelligence*. McGraw-Hill, 1971.
- Parberry, Ian, "A Real-Time Algorithm for the (n^2-1) -Puzzle," *Information Processing Letters*, Volume 56: 23 – 28, 1995, <http://www.eng.unt.edu/ian/pubs/saml.pdf>.
- Pearl, Judea, *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley, 1984.
- Pepicelli, Glen, "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>.
- Reinefeld, Alexander, "Complete Solution of the Eight-Puzzle and the Benefit of Node Ordering in IDA*," Proceedings of the 13th International Joint Conference on Artificial Intelligence (IJCAI), Volume 1, 1993, <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.40.9889>.
- Reinefeld, Alexander and T. Anthony Marsland, "Enhanced Iterative-Deepening Search," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16(7): 701 – 710, 1994. <http://citeseer.ist.psu.edu/article/reinefeld94enhanced.html>.
- Russel, Stuart, "Efficient memory-bounded search methods," Proceedings, 10th European Conference on Artificial Intelligence (ECAI): 1 – 5, 1992.
- Russell, S. J. and P. Norvig, *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2003.
- Samuel, Arthur, "Some Studies in Machine Learning Using the Game of Checkers," *IBM Journal* 3(3): 210 – 229, <http://www.research.ibm.com/journal/rd/033/ibmrd0303B.pdf>.
- Schaeffer, Jonathan, "Game Over: Black to Play and Draw in Checkers," *Journal of the International Computer Games Association (ICGA)*, <http://www.cs.unimaas.nl/icga/journal/contents/Schaeffer07-01-08.pdf>.
- Schaeffer, Jonathan, Neil Burch, Yngvi Björnsson, Akihiro Kishimoto, Martin Müller, Robert Lake, Paul Lu, and Steve Sutphen, "Checkers is Solved," *Science Magazine*,

September 14, 2007, 317(5844): 1518 – 1522, <http://www.sciencemag.org/cgi/content/abstract/317/5844/1518>.

Shannon, Claude, “Programming a Computer for Playing Chess,” *Philosophical Magazine*, 41(314), 1950, <http://tinyurl.com/ChessShannon-pdf>.

Wichmann, Daniel R. and Burkhard C. Wuensche, “Automated Route Finding on Digital Terrains,” Proceedings of IVCNZ, Akaroa, New Zealand, pp. 107 – 112, November 2004, http://www.cs.auckland.ac.nz/~burkhard/Publications/IVCNZ04_WichmannWuensche.pdf.





概述

有很多问题可以抽象成顶点和有容量限制的边的网络。如何解决这类问题将是本章所要关注的。Ahuja (1993) 对现有大量的网络流算法应用进行了深入的讨论。

任务分配

现在雇员们要完成一些任务，由于不同雇员在相同个任务上的花费不同，所以我们需要找到一种分配方式，使得总花费最小。

二部图匹配

有一些求职者需要面试一些工作，寻找到一个合理的分配方式，使得尽可能多的人能够找到能够胜任的工作。

运输问题

找到从工厂运输商品到零售商店性价比最高的方法。

转运问题

找到从工厂运输商品到零售商店性价比最高的方法，但是这类问题中，我们可以使用一些中转站作为临时仓库。

最大流

给定一个网络，网络中的每条边都有一个最大容量，计算出这个网络能够支持的最大流。

首先我们会阐述网络流问题之间的关系，这样能够帮助我们理解这些问题是如何解决的。图8-1描绘的是这些问题的关系。一条有向边将问题的一般化实例和问题的特例联系起来。例如运输问题就是转运问题的一个特例，因为运输图并不包含任何的中转节点。因此一个能够解决转运问题的程序肯定能够用于解决运输问题。

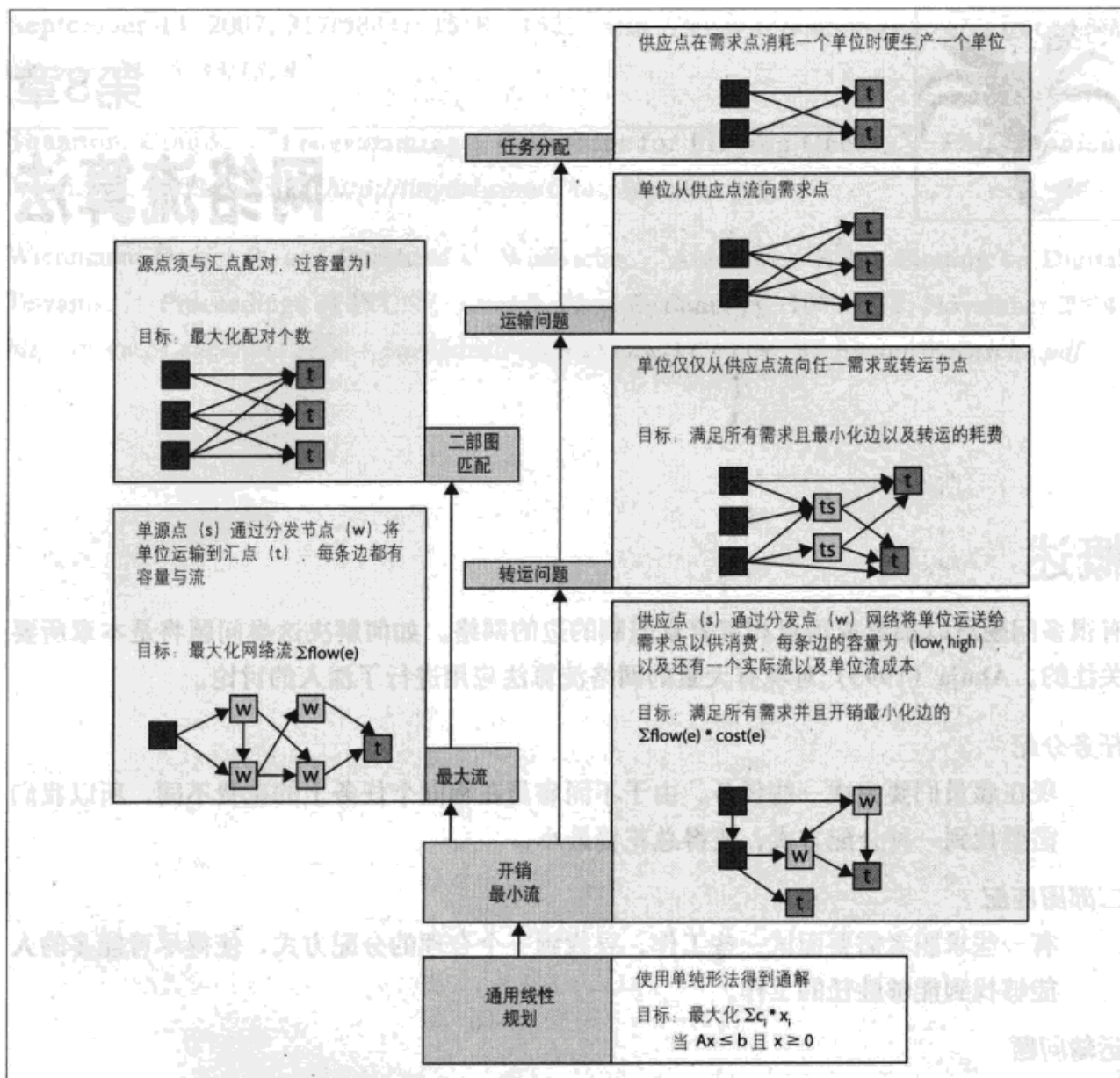


图8-1：网络流问题之间的关系

在本章中，我们将阐述Ford-Fulkerson算法。此算法用来解最大流问题。Ford-Fulkerson算法也可以直接用于二部图匹配问题，它们之间的关系如图8-1所示。而且，一旦理解了Ford-Fulkerson算法的基本原理，我们不难发现，Ford-Fulkerson算法的框架可以用来解决最小耗费流问题，例如转运问题、运输问题以及任务分配问题。

从理论上来说，你可以利用线性编程（LP）来解决图8-1的所有问题，但是你必须得先将这些问题转换成正确的线性表示形式，而且得到的解能够转换成原有的形式。在本章末尾，我们将会举例使用线性编程来解网络流问题。实际上，本章描述的特定算法解决图8-1的问题效率要比LP高数个数量级。

网络流

如图8-2所示，通常我们抽象出来的网络流模型是一个有向图 $G=(V,E)$ ， V 是顶点集， E 是边集。图是连通的（虽然并不是每条边都需要展示出来）。有一个源点 $s \in V$ ， s 生产商品，然后通过图的边运输到汇点 $t \in V$ 消费（也称为终点或者目标点）。网络流假设能够生产无限多的商品，并且汇点能够消费完所有接收到的商品。

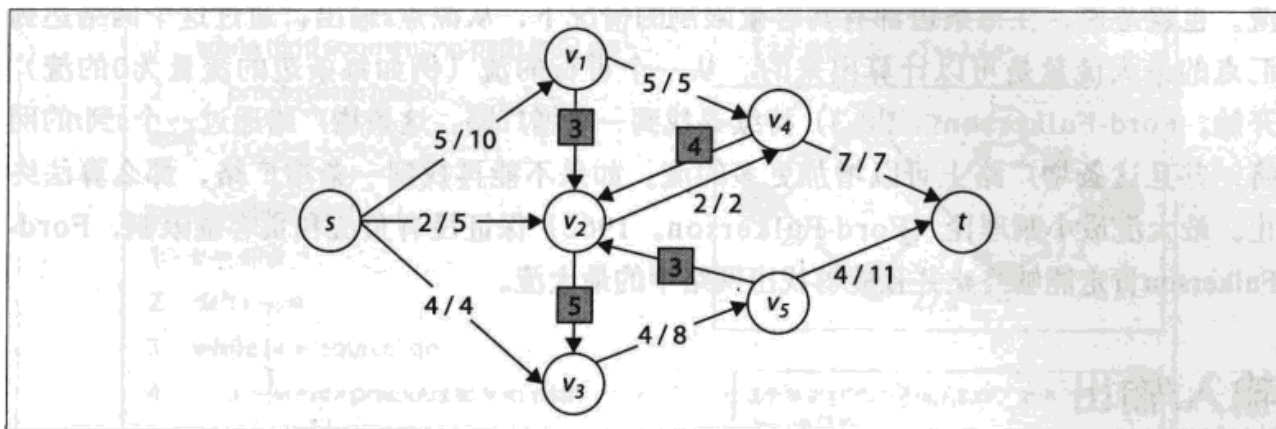


图8-2：网络流图

每条边 (u,v) 都有一个流 $f(u,v)$ ，表示从 u 运输到 v 的商品单位数目。同样每条边有一个容量 $c(u,v)$ ，表示从 u 运输到 v 的最大商品单位数。在图8-2中，每个顶点和每条边都做了标记，我们用数字标记顶点（源点和汇点用 s 和 t 标记），用 f/c 标记边， f/c 表示一条边上的流容量以及最大的流容量。例如边 (s,v_1) 标记为5/10，表示这条边上有5个单位的商品在通过这条边，这条边能够承受的最大容量为10个单位。当没有单位通过边（例如 v_5 和 v_2 的边），那么只会在一个灰色方格中标示容量。

在一个流通过网络时，必须满足如下的限制条件。

容量限制

通过一条边的流 $f(u,v)$ 不能为负，而且不能超过边的容量 $c(u,v)$ ，即 $0 \leq f(u,v) \leq c(u,v)$ 。如果在网络中不存在边 (u,v) ，那么 $c(u,v)=0$ 。

流守恒

除开源点 s 和汇点 t ，每个顶点 $u \in V$ 必须满足所有的入流等于所有的出流。这个性质确保了在网络中，除了源点和汇点，不会产生和消耗流。

反对称

从一致性角度来看， $f(v,u)$ 表示的是从顶点 u 到 v 的网络流。 $f(u,v) = -f(v,u)$ ，即使在一个有向图中存在两条边 (u,v) 和 (v,u) ，这个性质也同样成立。

在接下来的算法中，我们使用一条网络路径 $\langle v_1, v_2, \dots, v_n \rangle$ ，表示 $n-1$ 条连续的边 (v_i, v_{i+1}) ，这条网络路径中不存在环。在图8-2的有向图中，一条可能的网络路径是

$\langle v_3, v_5, v_2, v_4 \rangle$ 。在一条网络路径中，我们忽略边的方向。图8-3的一条可能的网络路径是 $\langle s, v_1, v_4, v_2, v_3, t \rangle$ 。

最大流

在一个流网络中，如果给定了所有边的容量，那么我们可以计算出这个流网络的最大流。也就是说，在每条边都有其容量限制的情况下，从源点 s 输出，通过这个网络达到汇点的最大流量是可以计算出来的。从一个可行的流（例如每条边的流量为0的流）开始，Ford-Fulkerson（图8-3）持续寻找到一条增广路，这条增广路通过一个 s 到 t 的网络，并且这条增广路上可以增加更多的流。如果不能再找到一条增广路，那么算法终止。最大流最小割理论（Ford-Fulkerson, 1962）保证没有负流和负容量限制，Ford-Fulkerson肯定能够终止并且能够找出网络中的最大流。

输入/输出

我们使用链表来存储边，每个顶点 u 维护着两个不同的列表：从 u 发出的前向边和指向 u 的后向边，因此每条边都会在这两个列表中出现，需要多一倍的存储空间。本书的代码库中的算法实现使用的是一个二维的矩阵来存储边，这个数据结构更适用于稠密的流网络图。

输入

流网络 $G=(V,E)$ 有一个源点 s 和一个汇点 t 。每条有向边 $e=(u,v)$ 都有一个整数容量 $c(u,v)$ 以及一个实际的流 $f(u,v)$ 。

输出

Ford-Fulkerson会处理每条边 (u,v) 。计算出给定的容量限制下，从 s 到 t 的最大流。这个时候，Ford-Fulkerson也计算出来了这个网络的最小割，也就是说形成瓶颈的边集。

解决方案

Ford-Fulkerson基于如下结构：

FlowNetwork

表示这个网络流问题。这个抽象类可以有两种实现，一种基于邻接表，另外一种使用数组。getEdgeStructure()方法返回的是存储边集的结构。

VertexStructure

为顶点维护两个链表，分别存储的是前向边和后向边。

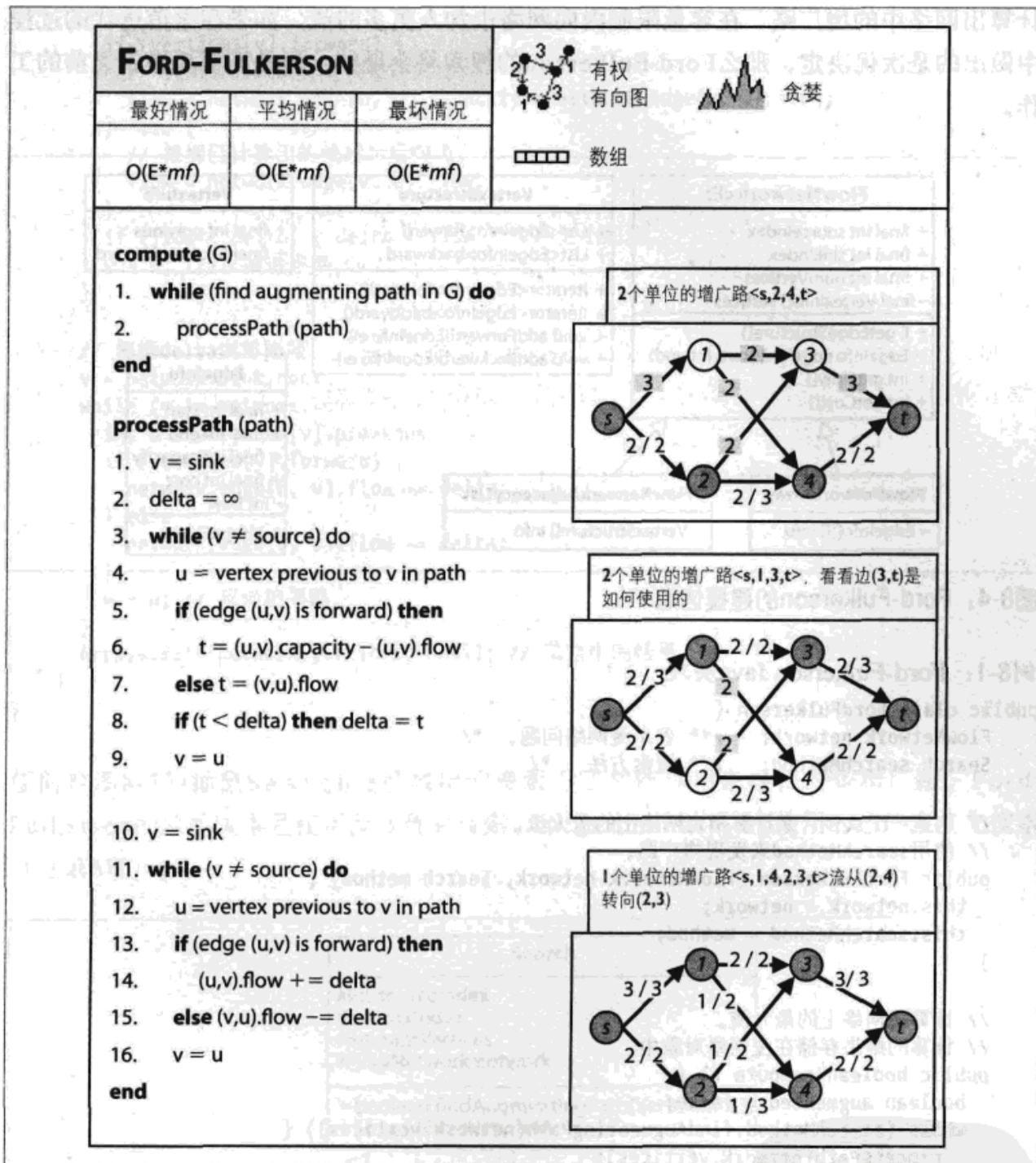


图8-3: Ford-Fulkerson详解

EdgeInfo

在网络流中记录边的信息。

VertexInfo

在数组中记录搜索到的增广路，增广路的前部节点和这个节点是否有一个前向边或者后向边。

例8-1是Ford-Fulkerson的一个实现，其流程如图8-4所示。一个可配置的Search对象能够

计算出网络中的增广路，在容量限制内向网络中加入更多的流。如果在之前迭代的过程中做出的是次优决定，那么Ford-Fulkerson的搜索将会继续，但是不需要撤销之前的工作。

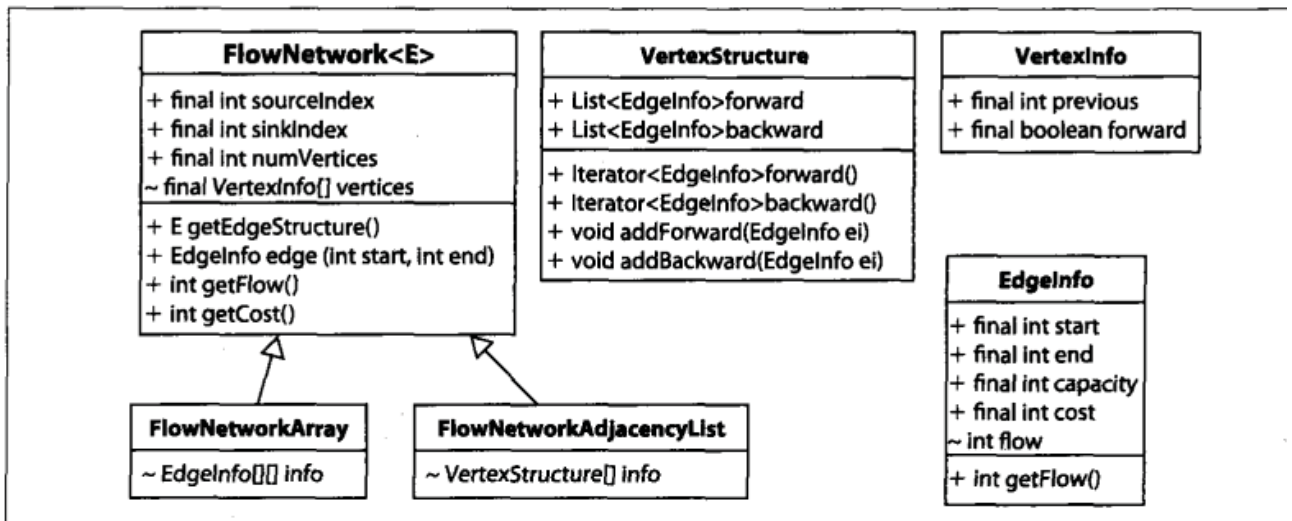


图8-4: Ford-Fulkerson的建模信息

例8-1: Ford-Fulkerson Java实现

```

public class FordFulkerson {
    FlowNetwork network;    /** 表示流网络问题。 */
    Search searchMethod;    /** 搜索方法。 */

    // 构造一个实例，来计算指定网络上的最大流，
    // 使用searchMethod来发现增广路。
    public FordFulkerson (FlowNetwork network, Search method) {
        this.network = network;
        this.searchMethod = method;
    }

    // 计算流网络上的最大流。
    // 计算的结果存储在流网络对象中。
    public boolean compute () {
        boolean augmented = false;
        while (searchMethod.findAugmentingPath(network.vertices)) {
            processPath(network.vertices);
            augmented = true;
        }
        return augmented;
    }

    // 寻找一条可扩展流最小的边，
    // 然后更新源点到汇点的流。
    protected void processPath(VertexInfo []vertices) {
        int v = network.sinkIndex;
        int delta = Integer.MAX_VALUE; // 目标是找到最小的
        while (v != network.sourceIndex) {
            int u = vertices[v].previous;

```

```

    int flow;
    if (vertices[v].forward) {
        // 利用边的剩余容量调整前向边。
        flow = network.edge(u, v).capacity - network.edge(u, v).flow;
    } else {
        // 根据已计算出的流减少后向边。
        flow = network.edge(v, u).flow;
    }
    if (flow < delta) { delta = flow; } // 更小的流。
    v = u; // 反向追寻路。
}

// 根据delta调整路径。
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; // 反向追寻路。
}
Arrays.fill(network.vertices, null); // 清空中间结果，进行下一次迭代。
}
}

```

任何自图8-5的抽象Search类扩展出的搜索方法都能够用来寻找一条增广路。Ford-Fulkerson的原始版本是使用深度优先搜索，不过Edmonds-Karp使用的是广度优先搜索（见第6章）。

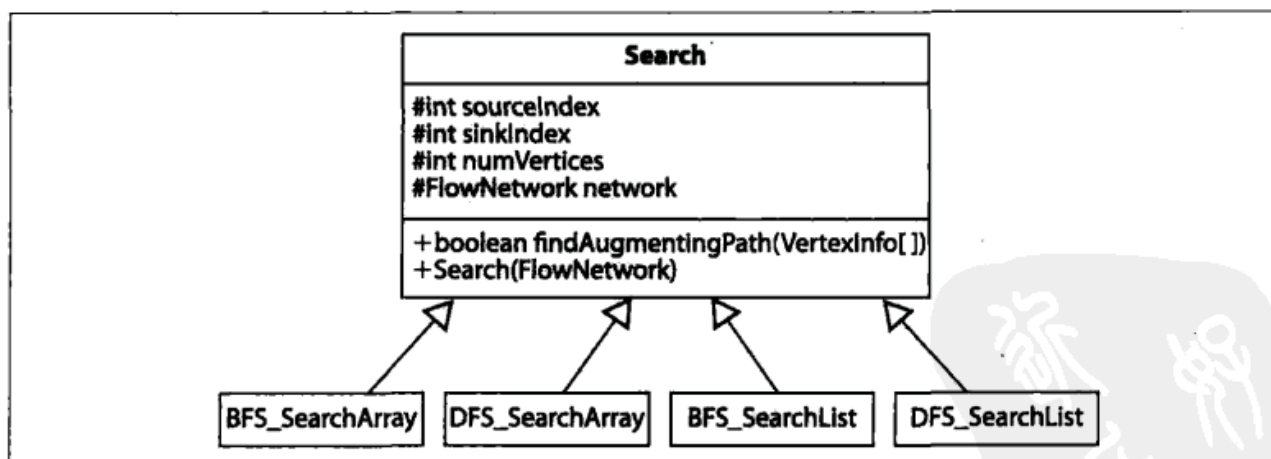


图8-5: Search的功能

例8-3的流网络是使用深度优先搜索寻找增广路得到的结果，例8-2是其实现。路径使用栈来存储顶点。我们从栈中弹出一个顶点 u ，然后找到 u 的一个未访问顶点 v ，且满足这样两个条件：(i) 边 (u, v) 是一条前向边并且还有多余的容量；(ii) 边 (v, u) 是一

条前向边，并且通过这条边的流可以减少。那么我们就能够找到一条潜在的增广路。最后，我们达到了汇点 t ，这样就找到了一条增广路径，或者路径为空，表示没有增广路。

例8-2：使用深度优先搜索来寻找增广路

```
public boolean findAugmentingPath (VertexInfo[] vertices) {
    // 从源点开始寻找增广路。
    vertices[sourceIndex] = new VertexInfo (-1);
    Stack<Integer> path = new Stack<Integer>();
    path.push (sourceIndex);

    // 处理u的前向边，然后再处理后向边。
    VertexStructure struct[] = network.getEdgeStructure();
    while (!path.isEmpty()) {
        int u = path.pop();

        // 首先处理前向边。
        Iterator<EdgeInfo> it = struct[u].forward();
        while (it.hasNext()) {
            EdgeInfo ei = it.next();
            int v = ei.end;

            // 如果还没有访问，并且有剩余容量，那么就准备增加流。
            if (vertices[v] == null && ei.capacity > ei.flow) {
                vertices[v] = new VertexInfo (u, FORWARD);

                if (v == sinkIndex) { return true; } // 我们已经找到了增广路!
                path.push (v);
            }
        }

        // 尝试后向边。
        it = struct[u].backward();
        while (it.hasNext()) {
            // 尝试找到一条到达u的能够减少流的边。
            EdgeInfo rei = it.next();
            int v = rei.start;

            // 尝试处理非汇点的后向边。
            if (vertices[v] == null && rei.flow > 0) {
                vertices[v] = new VertexInfo (u, BACKWARD);
                path.push(v);
            }
        }
    }

    // 没有找到。
    return false;
}
```

顶点之间的后向指针由VertexInfo[]结构来维护，这样FordFulkerson.processPath方法能够遍历这条增广路。

广度优先搜索的实现，又名Edmonds-Karp算法，其实现见例8-3。在搜索过程中，路径的结构是顶点组成的队列。我们从队列头移除顶点 u ，然后将可能组成增广路径的邻接顶点加入到队列尾部，这就是我们扩展潜在增广路径的方式。同样，我们达到了汇点 t ，这样就找到了一条增广路径，或者路径为空，表示没有增广路，此时算法结束。我们使用图8-3的流网络，使用广度优先搜索能够找到4条增广路，分别是 $\langle s, 1, 3, t \rangle$ ， $\langle s, 1, 4, t \rangle$ ， $\langle s, 2, 3, t \rangle$ 和 $\langle s, 2, 4, t \rangle$ 。也得到了相同的最大流。

例8-3：使用广度优先搜索来寻找增广路径

```
public boolean findAugmentingPath (VertexInfo []vertices) {
    // 寻找增广路。
    vertices[sourceIndex] = new VertexInfo (-1);
    DoubleLinkedList<Integer> path = new DoubleLinkedList<Integer>();
    path.insert (sourceIndex);

    // 首先处理前向边，然后再处理后向边。
    VertexStructure struct[] = network.getEdgeStructure();
    while (!path.isEmpty()) {
        int u = path.removeFirst();

        Iterator<EdgeInfo> it = struct[u].forward(); // 前向边。
        while (it.hasNext()) {
            EdgeInfo ei = it.next();
            int v = ei.end;

            // 如果还没有访问，并且有剩余容量，那么就准备增加流。
            if (vertices[v] == null && ei.capacity > ei.flow) {
                vertices[v] = new VertexInfo (u, FORWARD);
                if (v == sinkIndex) { return true; } // 找到增广路。
                path.insert (v); // 否则追加到队列尾。
            }
        }

        it = struct[u].backward(); // 后向边。
        while (it.hasNext()) {
            // 尝试找到一条到达u的能够减少流的边。
            EdgeInfo rei = it.next();
            int v = rei.start;

            // 是否还没有访问并且能够减少流？
            if (vertices[v] == null && rei.flow > 0) {
                vertices[v] = new VertexInfo (u, BACKWARD);
                path.insert (v); // 追加到队列尾。
            }
        }
    }

    return false; // 没有找到增广路。
}
```

结论

当Ford-Fulkerson算法终止时，顶点集 V 被分成了两个不相交的子集， S 和 T ($T=V-S$)。注意 $s \in S$, $t \in T$ 。 S 是 V 中不能组成增广路径的顶点集合。我们得到的这两个集合是非常重要的，因为在 S 和 T 之间的前向边组成了流网络的“最小割”。也就是说，这些边成了这个流网络的“瓶颈”，因为（a）流网络被分成了两个顶点集合， S 和 T ，这两个顶点集合的流是最小的；（b）在 S 和 T 之间的流已经达到容量限制了。

分析

Ford-Fulkerson肯定能够终止，因为流的单位为整数（Ford-Fulkerson, 1962）。使用深度优先搜索的Ford-Fulkerson性能为 $O(E*mf)$ ，取决于得到的最大流的值 mf 。简单地说，我们可以在每次迭代时向增广路中加入一个单位的流，但是容量巨大的网络可能会需要大量的迭代。在这种情况下，我们会非常惊讶地看到运行时间与问题的规模无关（例如顶点数或者边数），而是和边的容量相关。当使用广度优先搜索时（又名Edmonds-Karp算法），性能为 $O(V*E^2)$ 。广度优先搜索能够在 $O(V+E)$ 的时间内找到最短的增广路，如果顶点比边数少很多，那么实际性能是 $O(E)$ 。Cormen等（2001）曾证明基于广度优先搜索的Ford-Fulkerson算法是按照长度来依次求出增广路的，而不是像深度优先搜索那样花费大量的时间，尝试尽可能接近汇点。

优化

流网络的典型实现方式是使用数组存储信息。我们在这里使用的是可读性比较高的代码来解释算法，能够使读者理解算法是如何工作的。但是我们很有必要考虑一下，在代码上做优化能够获得多大的性能提升。第2章中，我们在 n 位数乘法中获得了40%的性能提升。很明显，我们可以编写出更快的代码，但是代码可能可读性较差，而且不易维护。例8-4是Ford-Fulkerson算法Java实现的优化版本。

例8-4：优化的Ford-Fulkerson实现

```
public class Optimized extends FlowNetwork {
    int[][] capacity;    // 容量。
    int[][] flow;        // 流。
    int[] previous;      // 路节点的前驱信息。
    int[] visited;       // 记录是否某个节点访问过。

    final int QUEUE_SIZE; // 队列的最大规模。
    final int queue[];    // 使用循环队列。

    // 加载信息。
    public Optimized(int n, int s, int t, Iterator<EdgeInfo> edges) {
        // 初始化超类。
        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];

// 最开始, 流为0。
while (edges.hasNext()) {
    EdgeInfo ei = edges.next();
    capacity[ei.start][ei.end] = ei.capacity;
}

// 计算并且返回最大流。
public int compute (int source, int sink) {
    int maxFlow = 0;
    while (search(source, sink)) { maxFlow += processPath(source, sink); }
    return maxFlow;
}

// 增大寻找到的路上的流。
protected int processPath(int source, int sink) {
    // 计算将要增加多少流。
    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];
    }

    // 增加流。
    v = sink;
    while (previous[v] != -1) {
        flow[previous[v]][v] += increment; // 前向边。
        flow[v][previous[v]] -= increment; // 不要忘了后向边。
        v = previous[v];
    }

    return increment;
}

// 寻找流网络中源点到汇点的增广路。
public boolean search (int source, int sink) {
    // 清除访问状态, 0=未访问, 1=队列中, 2=已访问
    for (int i = 0 ; i < numVertices; i++) { visited[i] = 0; }

    // 创建一个循环数组
    queue[0] = source;
    int head = 0, tail = 1;
    previous[source] = -1; // 保证我们会在源点终止。
    visited[source] = 1; // 放入队列。
    while (head != tail) {

```



```

int u = queue[head]; head = (head + 1) % QUEUE_SIZE;
visited[u] = 2;

// 将u的邻接顶点中, 仍然有剩余容量的顶点加入到队列。
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; // 放入队列。
        previous[v] = u;
    }
}
}

return visited[sink] != 0; //我们访问过了汇点吗?
}
}

```

相关算法

Goldberg和Tarjan (1986) 提出的压入与重标记算法将性能提升到 $O(V \cdot E \cdot \log(V^2/E))$, 并且能够并行化获得更大的性能提升。最大流问题的一个变种, 又名多重物资流问题, 泛化之后就是最大流问题。简单地说, 这种问题就是一个有着多重源点和汇点的网络, 并且在不同源点和汇点之间运输不同的物资, 而不是像最大流问题中, 仅仅只有一个源点和一个汇点。尽管边的容量是固定的, 但是每对汇点和源点的使用需求并不一样。现实中, 这种算法的应用于无线网中的路由问题 (Fragouli和Tabet, 2006)。Leighton和Rao撰写过一篇文章 (1999), 这篇文章被大量的多重物资流问题所引用。

最大流的一些简单变种如下:

顶点容量

如果流网络中的每个顶点 v 都有一个最大容量 $k(v)$ 将会怎么样呢? 构建一个修改过的图 G' 。对于网络中的每个顶点 v , 构建两个顶点 v' 和 v'' 。并且创建一条边 (v'', v') , 这条边的容量是 $k(v)$ 。在图 G 中创建一条容量为 $c(x, v)$ 的边 (x, v) , 对应原始图 G 中的容量为 $c(x, v)$ 的每条边 (x, v'') 。对应着原始图 G 中的每条边 (v, w) , 在 G' 中创建一条新边 (v', w) , 其容量为 $k(v)$ 。解出 G' 也就得到了 G 的解。

无向边

如果流网络 G 是无向边, 那么会怎么样呢? 构建一个流网络 G' 。在新图中, 所有的顶点都一样, 对应着原始图中容量为 $c(u, v)$ 的边 (u, v) , 在 G' 中构建一对新边 (u, v) 和 (v, u) , 每条边的容量都为 $c(u, v)$ 。 G' 的解也就是 G 的解。

二部图匹配

匹配问题有多种形式。我们考虑如下这种情况, 五个求职者会面试五个职位。每个求

职者都已经列出了满足能力要求的职位。现在的任务就是将尽可能多地分配工作给求职者，但是每份工作只能提供给一位求职者。我们会非常惊讶地发现可以使用Ford-Fulkerson算法来解二部图匹配问题。在计算机科学中，这种方法叫做“问题归约”。我们将二部图匹配问题归约为在流网络上的最大流问题，同时会介绍（a）如何将二部图匹配问题的输入映射成最大流问题的输入，以及（b）怎么将最大流问题的输出映射成二部图匹配问题的输出。

输入/输出

输入

二部图匹配包含一个包含 n 个元素 s_i 的集合， $s_i \in S$ ；一个包含 m 个参与者 t_j 的集合，一个包含 p 个可行的配对 p_k 的集合（ $1 \leq k \leq p, p_k \in P$ ），配对将元素 $s_i \in S$ 和参与者 $t_j \in T$ 连接起来。 S 集和 T 集是不连通的，这就是二部图匹配名字的来历。

输出

原始可行配对集合中得到的一个配对集合 (s_i, t_j) ， P 。这个集合的规模是最大能够匹配上的配对数目。算法保证不存在更多的匹配数（虽然会有其他的匹配方式能够得到同样多的匹配个数）。

解决方案

我们将二部图匹配问题归约成为一个最大流问题，而不是重新设计一个算法来解决问题。在二部图匹配中，从元素集 S 和参与者 T 中选择了—个匹配 (s_i, t_j) 之后， s_i 或者 t_j 就不能被其他的匹配选择了。在一个流网络图 $G=(V,E)$ 中得到这样的性质的话，我们应该这样构建图 G 。

V 包含 $n+m+2$ 个顶点

每个元素 s_i 映射为顶点 i ，每个参与者映射为顶点 $n+j$ 。创建一个新的源点 src （标记为0）和—个新的汇点 tgt （标记为 $n+m+1$ ）。

E 包含 $n+m+k$ 条边

首先会构建 n 条边，连接源点 src 和从 S 映射过来的顶点。然后再构建 m 条边，连接汇点 tgt 和从 T 映射过来的顶点 T 。每个匹配 $p_k=(s_i, t_j)$ 会映射成边 $(i, n+j)$ 。每条边的流容量为1。

我们在流网络图 G 中计算出的最大流就是原始二部图匹配问题中的最大匹配集合，证明见（Cormen等，2001）。例如，考虑图8-6a中两个匹配 (a,z) 和 (b,y) 的组成最大匹

配集合，依照上述过程构造的流网络如图8-6b。我们可以将其扩展到三对： (a,z) 、 (c,y) 和 (b,x) 。在找到一条增广路 $\langle 0,3,5,2,4,7 \rangle$ 之后，会对流网络进行相应的调整。

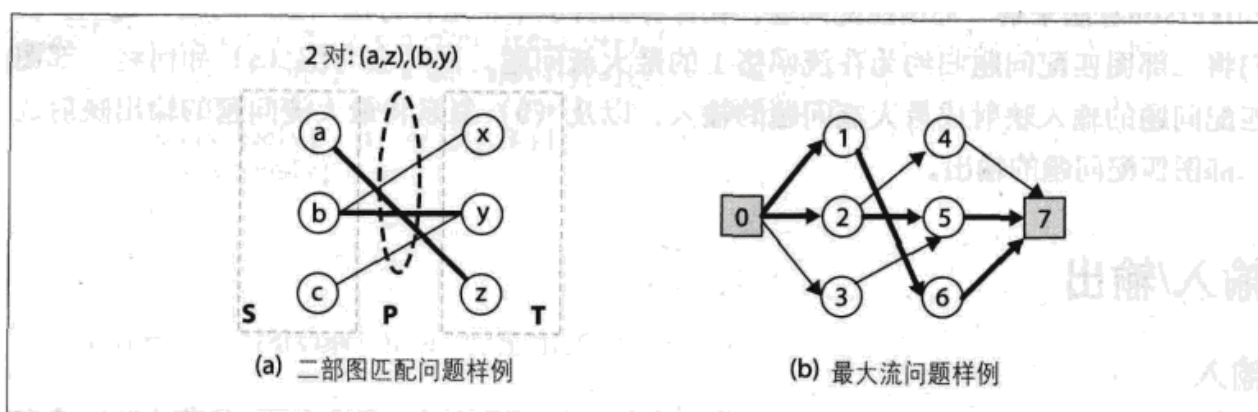


图8-6：小规模的二部图匹配归约成最大流问题

一旦找到最大流，我们将要将最大流问题的输出转换成二部图匹配问题的输出。也就是说，对于每一条流为1的边 (s_i, t_j) ，输出选中了匹配 $(s_i, t_j) \in P$ 。为了简化表述，例8-5的代码中删除了错误检测（注1）。

例8-5：使用Ford-Fulkerson的二部图匹配

```
public class BipartiteMatching {

    ArrayList<EdgeInfo> edges; /* S和T的边。 */
    int ctr = 0; /* id计数器。 */

    /* 问题实例之间的相互映射。 */
    Hashtable<Object,Integer> map = new Hashtable<Object,Integer>();
    Hashtable<Integer,Object> reverse = new Hashtable<Integer,Object>();

    int srcIndex; /* 源点索引。 */
    int tgtIndex; /* 汇点索引。 */
    int numVertices; /* 顶点数。 */

    public BipartiteMatching (Object[] setS, Object[] setT, Object[][] pairs)
        throws RuntimeException {

        edges = new ArrayList<EdgeInfo>();

        // 将匹配转换成流网络中的输入。
        // 每条边的容量为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);
            }
        }
    }
}
```

注1： 要了解更多的细节，请参见代码库中的algs.model.network.matching.Bipartite-Matching。


```

        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));
}

// 添加额外的源点和汇点。
srcIndex = 0;
tgtIndex = setS.length + setT.length+1;
numVertices = tgtIndex+1;
for (Object o : setS) {
    edges.add(new EdgeInfo(0, map.get(o), 1));
}
for (Object o : setT) {
    edges.add(new EdgeInfo(map.get(o), ctr+1, 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();

    // 从原始的边集中取出边进行处理，忽略连接添加的“源点”和“汇点”的边。
    // 当且仅当流等于1时，这条边属于解的一部分。
    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();
}
}

```

分析

如果要能够高效地归约，那么必须高效地从原始问题映射到目标问题。二部图问题 $M=(S,T,P)$ 可以在 $n+m+k$ 步内转换成为一个图 $G=(V,E)$ 。 G 有 $n+m+2$ 个顶点以及 $n+m+k$ 条边，因此图 G 的规模仅仅比原始二部图问题的规模大一个常数规模而已。这个重要特性决定了我们能够高效地求解二部图问题。一旦 Ford-Fulkerson 算法计算出最大流，那么网络中流为 1 的边对应着二部图中选择的匹配。要找到这些边只需要 k 步，所以需要 $O(k)$ 的时间来将解映射成二部图问题的解。

在增广路上的深入思考

解决最大流问题并不能帮助我们立刻解决本章之前讨论的其他问题。但是，从最大流问题的解决出发，我们去考虑一类相似的问题，这类问题都是在流网络中寻找最大流以及最小化开销。如果我们将网络中的每条边 (u, v) 赋一个开销 $d(u, v)$ 表示在此边上运输一个单位的开销，那么目标就是最小化：

$$\sum f(u, v) * d(u, v)$$

现在，在Ford-Fulkerson算法中，我们强调寻找能够增大网络中流量的增广路的重要性。如果我们修改搜索方法，寻找最小开销的增广路会发生什么呢？我们已经讨论过一些贪心算法（例如第6章中构造最小生成树的Prim算法）就是不断地选择最小开销的方向进行扩展，也许这样的方法能够在这里可行。

为了寻找最小开销的增广路，我们就不能拘泥于单纯使用广度优先搜索或者是深度优先搜索。正如我们在Prim算法中所看到的那样，我们必须使用一个优先队列来存储和计算流网络中，每个顶点到源点的距离。我们会计算从源点到每一个顶点运输一个单位的开销，然后我们需要在不断的计算过程中维护一个优先队列。随着搜索的进行，优先队列存储着节点的有序集合，这个集合中的元素将会指导着搜索进行的方向。从优先队列中取出到源点距离（这里是开销）最小的顶点 u ，然后选择一个还未被访问的邻接顶点 v ，并且满足（a）前向边 (u, v) 还有多余的容量可供扩展，或者（b）后向边 (v, u) 的流可以减少，继续进行搜索。如果我们在搜索的过程中遇到了汇点，那么搜索就可以终止，而且成功地找到一条增广路，否则，不存在这样的增广路。例8-6是ShortestPathArray的Java实现。

例8-6: Ford-Fulkerson寻找最短路径（以开销计算）

```
public boolean findAugmentingPath (VertexInfo[] vertices) {
    Arrays.fill(vertices, null); // 重置

    // 构建基于二叉堆的队列。inqueue[]数组的目的是避免花费O(n)的
    // 时间来查找是否一个元素在队列中。
    int n = vertices.length;
    BinaryHeap<Integer> pq = new BinaryHeap<Integer> (n);
    boolean inqueue[] = new boolean [n];

    // 初始化dist[]数组。如果边不存在，dist等于INT_MAX。
    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;

    /** 我们得到了想要的结果。 */
    if (u == sinkIndex) { break; }

    for (int v = 0; v < n; v++) {
        if (v == sourceIndex || v == u) continue;

        // 处理前向边。
        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;
                }
            }
        }
    }

    // 处理后向边。
    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;
}

```

使用这样的策略来寻找最小开销的增广路，我们可以解决图8-1的剩余问题。在图8-7中我们使用一个小例子来比较一个最大流计算和最小开销流计算，从中可以看到这个最小开销搜索策略的高效了。图的底部是每种方法找到的最大流。

在这个例子中，你是负责两家工厂的船务经理，这两家工厂一家在芝加哥（ v_1 ），另一家在华盛顿特区（ v_2 ），每家工厂每天能够生产300件工具。你必须保证在休斯敦（ v_3 ）和波士顿（ v_4 ）的消费者每天能够有300件工具供应。你有一些选择策略，如图8-7所示。例如，在华盛顿特区和休斯敦之间每天运输280件工具，每件工具花费4美元，但是如果你从华盛顿特区运到波士顿，开销会增至6美元（虽然在这条航路每天可以运送350件工具）。

我们不清楚Ford-Fulkerson算法是否可以用来解决这个问题，但是我们能够构造一个图 G ，有一个源点 s_0 ，连接两个工厂顶点（ v_1 和 v_2 ）以及有一个新的汇点 t_5 ，连接两个消费顶点（ v_3 和 v_4 ）。在图8-7的左边我们使用Edmonds-Karp来证明我们能够满足消费者的所有需求，但是每天的运输费用是3600美元。为了节省空间，源点和汇点都被忽略了。在Ford-Fulkerson的每次迭代中，增广路的影响（当一次迭代更新了边的流，流的值标记为灰色）显而易见。

我们能够得到最小开销吗？在图8-7的右边，我们给出了使用ShortestPathArray作为搜索策略的Ford-Fulkerson算法执行过程。我们可以看到第一条增广路的寻找过程是如何利用最低开销的航运率。同样ShortestPathArray使用了开销最高昂的路径，即从芝加哥（ v_1 ）到休斯敦（ v_3 ）的路，因为没有其他的路可以满足用户的需求，事实上，如果这种情况发生，我们将会看到增广路减少了从华盛顿特区（ v_2 ）和到休斯敦（ v_3 ）、波士顿（ v_4 ）之间的流。

最小开销流

解决最小开销流问题，我们仅仅需要构造一个流网络图，并且保证满足之前讨论过的那些限制条件：容量限制、流量守恒以及反对称。而且还要保证如下两个条件。

供应满足

对每个源点 $s_i \in S$ ，所有边 $(s_i, v) \in E$ 的 $f(s_i, v)$ 之和（流出 s_i 的流）减去所有边 $(u, s_i) \in E$ 的 $f(u, s_i)$ 之和（流入 s_i 的流）必须小于等于 $sup(s_i)$ ，也就是说，每个源点供应的商品的上限是 $sup(s_i)$ 。

需求满足

对于每个汇点 $t_j \in T$ ，所有边 $(u, t_j) \in E$ 的 $f(u, t_j)$ 之和（流入 t_j 的流）减去所有边 $(t_j, v) \in E$ 的 $f(t_j, v)$ 之和（流出 t_j 的流）必须小于等于 $dem(t_j)$ 。也就是说，每个汇点需要的商品上限是 $dem(t_j)$ 。

为了简化算法的实现，我们将会限制流网络只有一个源点和一个汇点。通过向现存的多源多汇点的流网络图添加两个新顶点，这种流网络可以轻易得到。首先，添加一个新的顶点（ s_0 ）成为流网络的源点，然后对每个源点 $s_i \in S$ ，添加一条容量 $c(s_0, s_i) = sup(s_i)$ 以及

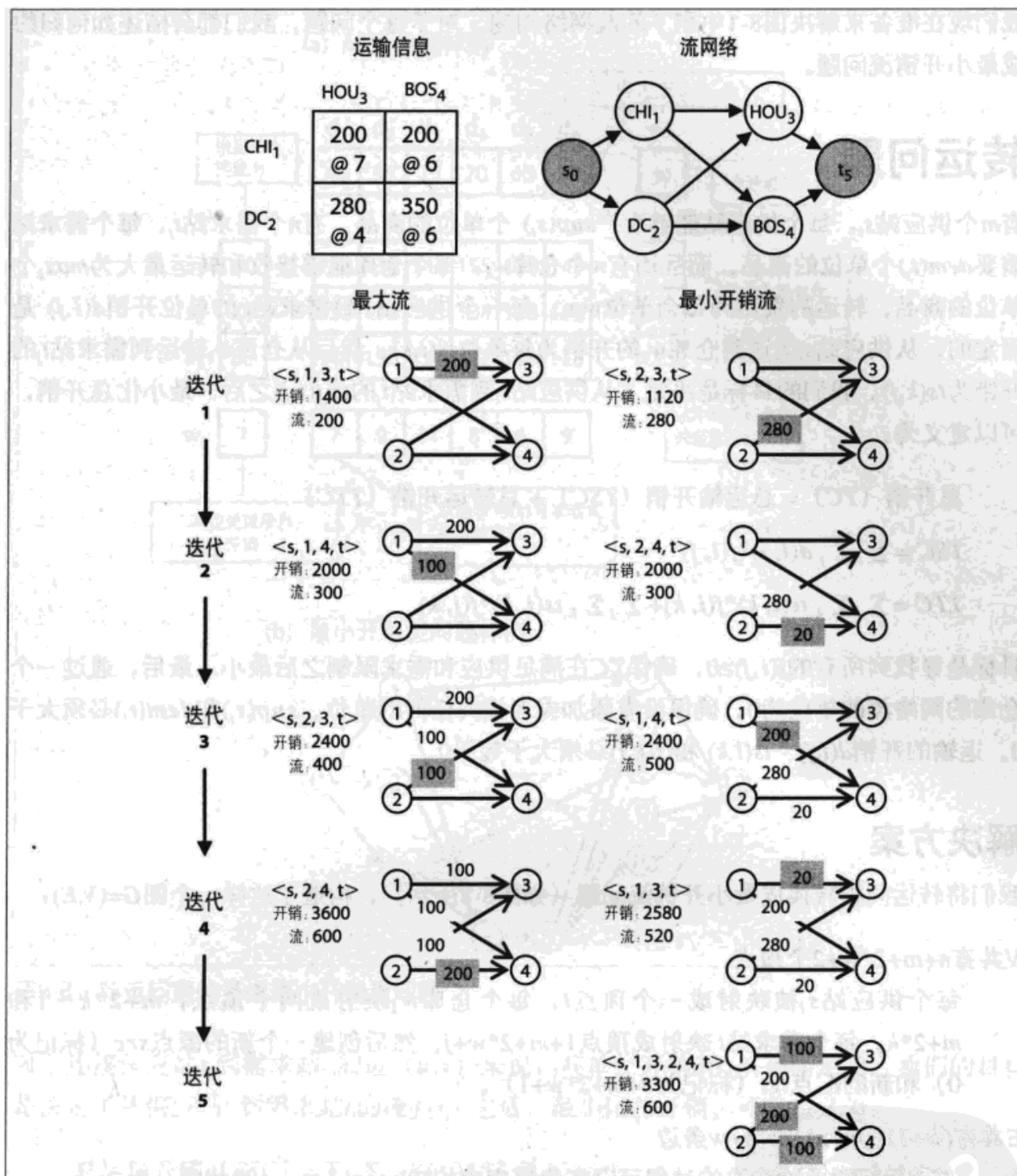


图8-7：揭示最小开销搜索策略的高效

开销 $d(s_0, s_i)=0$ 的边 (s_0, s_i) 。然后，添加一个新的顶点 (t_j, tgt) 成为流网络的汇点，然后对每个源点 $t_j \in T$ ，添加一条容量 $c(t_j, tgt)=dem(t_j)$ 以及开销 $d(t_0, t_j)=0$ 的边 (t_j, tgt) 。你可以看到，添加这些顶点和边不会增加原始流网络的开销，但是也不会减少最后计算出的流。

供应 $sup(s_i)$ ，需求 $dem(t_j)$ 以及容量 $c(u, v)$ 都大于0。每条边的航运开销 $d(u, v)$ 可能大于等于0。计算出结果之后，所有的 $f(u, v)$ 都会大于等于0。

我们现在准备来解决图8-1中剩下的流网络问题。对于每个问题，我们都会描述如何归约成最小开销流问题。

转运问题

有 m 个供应站 s_i ，每个供应站能够生产 $sup(s_i)$ 个单位的商品。有 n 个需求站 t_j ，每个需求站需要 $dem(t_j)$ 个单位的商品。而且还有 w 个仓库 w_k ，每个仓库能够接收和转运最大为 max_k 个单位的商品，转运的费用为每个单位 $w p_k$ 。每一个供应站 s_i 到需求站 t_j 的单位开销 $d(i, j)$ 是固定的，从供应站 s_i 运送到仓库 w_k 的开销为每单位 $ts(i, k)$ ，然后从仓库 w_k 转运到需求站 t_j 的开销为 $ts(k, j)$ 。我们的目标是决定了从供应站 s_i 到需求站 t_j 的流 $f(i, j)$ 之后，最小化总开销，可以定义为：

$$\text{总开销 (TC)} = \text{总运输开销 (TSC)} + \text{总转运开销 (TTC)}$$

$$TSC = \sum_i \sum_j d(i, j) * f(i, j)$$

$$TTC = \sum_i \sum_k ts(i, k) * f(i, k) + \sum_j \sum_k ts(k, j) * f(j, k)$$

目标是寻找到所有的 $f(i, j) \geq 0$ ，确保 TC 在满足供应和需求限制之后最小。最后，通过一个仓库的网络流的单位为0，确保没有添加或者损失任何的单位。 $sup(s_i)$ 和 $dem(t_j)$ 必须大于0。运输的开销 $d(i, j)$ 、 $ts(i, k)$ 和 $ts(k, j)$ 必须大于等于0。

解决方案

我们将转运问题转换成最小开销流问题（如图8-8所示），构造了这样一个图 $G=(V, E)$ ：

V 共有 $n+m+2*w+2$ 个顶点

每个供应站 s_i 被映射成一个顶点 i ，每个仓库 w_k 映射成两个顶点， $m+2*k-1$ 和 $m+2*k$ 。每个需求站 t_j 映射成顶点 $1+m+2*w+j$ 。然后创建一个新的源点 src （标记为0）和新的汇点 tgt （标记为 $n+m+2*w+1$ ）。

E 共有 $(w+1)*(m+n)+m*n+w$ 条边

在转运问题中构建边的过程可以在代码库中的Transshipment类找到。

如果对应的最小开销流问题能够得到解，那么转运问题也能够得到解，所有的 $f(u, v) > 0$ 的边 $(u, v) \in E$ 构成了结果集合。这些边的 $f(u, v) * d(u, v)$ 之和是结果集合的总开销。

运输问题

运输问题比转运问题要简单得多，因为没有中继仓库节点。有 m 个供应站 s_i ，每个供应站能够生产 $sup(s_i)$ 个单位的商品。有 n 个需求站 t_j ，每个需求站需要 $dem(t_j)$ 个单位的商品。

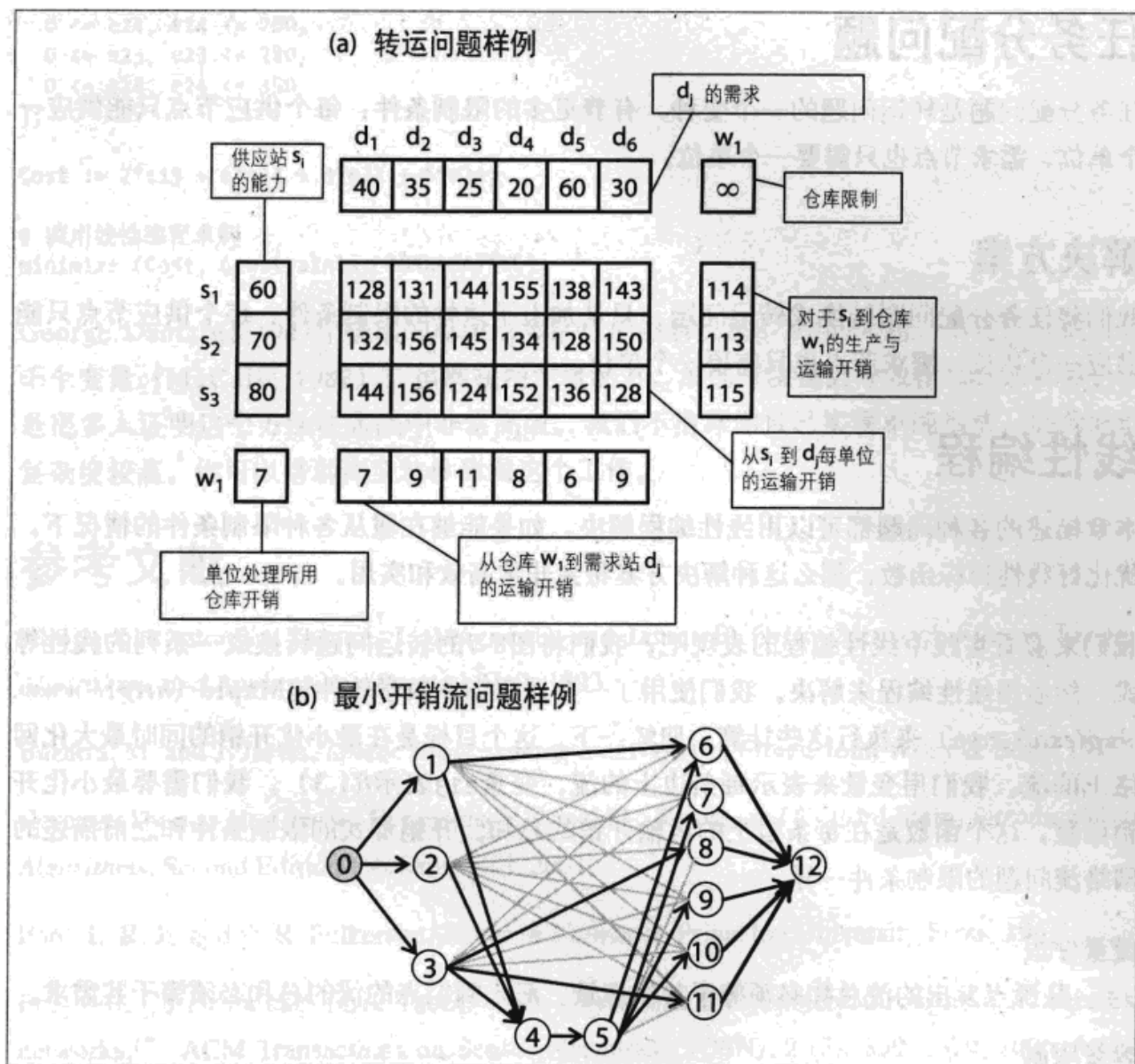


图8-8：转运问题转换成最小开销流问题

对于连接供应站 s_i 到需求站 t_j 的边 (u, v) 来说，其单位开销 $d(i, j)$ 是固定的。我们的目标是决定了从供应站 s_i 到需求站 t_j 的流 $f(i, j)$ 之后，最小化总开销。可以定义为：

$$\text{总运输开销 (TSC)} = \sum_i \sum_j d(i, j) * f(i, j)$$

解必须满足不大于每个需求站 t_j 的需求和每个供应站 s_i 的供应能力。

解决方案

我们将运输问题转换成没有中继仓库节点的转运问题。

任务分配问题

任务分配问题是转运问题的一个变种，有着更多的限制条件：每个供应节点只能供应一个单位，需求节点也只需要一个单位。

解决方案

我们将任务分配问题转换成转运问题，只是加上了这样的限制条件：每个供应节点只能供应一个单位，需求节点也只需要一个单位。

线性编程

本章描述的各种问题都可以用线性编程解决，如果能够在服从各种限制条件的情况下，优化好线性目标函数，那么这种解决方案将会非常高效和实用。

我们来看看实践中线性编程的表现吧，我们将图8-7的转运问题转换成一系列的线性等式，然后用线性编程来解决。我们使用了一个通用商业数学软件包Maple (<http://www.maplesoft.com>) 来执行这些计算。回忆一下，这个目标是在最小化开销的同时最大化网络上的流。我们用变量来表示每条边上的流，变量 e_{13} 表示 $f(1,3)$ 。我们需要最小化开销函数，这个函数是在每条边上的运输开销的总和。开销等式的限制条件和之前描述的网络流问题的限制条件一样。

流量守恒

从源点发出的流总和必须等于其供应量。汇入到汇点的流的总和必须等于其需求。

容量限制

一条边的流 $f(i,j)$ 必须大于等于0。同样 $f(i,j) \leq c(i,j)$ 。

Maple计算的结果是 $\{e_{13}=100, e_{24}=100, e_{23}=200, e_{14}=200\}$ ，恰好等于之前计算出来的最小总开销（见例8-7）。

例8-7：求转运问题的最小化开销的Maple命令

```
with(simplex);
```

```
Constraints := [  
  # 每个顶点的单位限制  
   $e_{13}+e_{14} = 300$ , # 芝加哥  
   $e_{23}+e_{24} = 300$ , # 华盛顿  
  
   $e_{13}+e_{23} = 300$ , # 休斯敦  
   $e_{14}+e_{24} = 300$ , # 波士顿  
  
  # 每条边上的最大流  
   $0 \leq e_{13}, e_{13} \leq 200$ ,
```

```

0 <= e14, e14 <= 200,
0 <= e23, e23 <= 280,
0 <= e24, e24 <= 350
];

Cost := 7*e13 + 6*e14 + 4*e23 + 6*e24;

# 调用线性编程求解
minimize (Cost, Constraints, NONNEGATIVE);

```

George Dantzig在1947年提出的单纯形算法也能够解决例8-7中的问题，不过需要成百上千个变量（McCall, 1982）。虽然在特定情况下，单纯形法也会导致指数级的计算，但是很多人证明这个方法在实践中非常高效。我们不推荐你自己实现单纯形法，因为它的复杂度较高，你可以借助商业软件库做这个工作。

参考文献

- Ahuja, Ravindra K., Thomas L. Magnanti, and James B. Orlin, *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, 1993.
- Bazarrar, M. and J. Jarvis, *Linear Programming and Network Flows*. John Wiley & Sons, 1977.
- Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to Algorithms*, Second Edition. McGraw Hill, 2001.
- Ford, L. R. Jr. and D. R. Fulkerson, *Flows in Networks*. Princeton University Press, 1962.
- Fragouli, Christina and Tarik Tabet, "On conditions for constant throughput in wireless networks," *ACM Transactions on Sensor Networks (TOSN)*, 2 (3): 359–379, 2006, <http://portal.acm.org/citation.cfm?id=1167938>.
- Goldberg, A. V. and R. E. 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://portal.acm.org/citation.cfm?doid=12130.12144>.
- Leighton, Tom and Satish Rao, "Multicommodity max-flow min-cut theorems and their use in designing approximation algorithms," *Journal of the ACM*, 46 (6): 787–832, 1999, <http://portal.acm.org/citation.cfm?doid=331524.331526>.
- McCall, Edward 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://portal.acm.org/citation.cfm?id=358461>.
- Orden, Alex, "The Transshipment Problem," *Management Science*, 2(3), 1956.



概述

我们将会介绍一系列计算几何的相关问题。在过去几个世纪里，数学家们已经研究过很多这样的问题，自20世纪70年代起，计算几何开始被看做几何算法的一个子学科系统，相关数据结构保证了算法能够高效地执行。在这样的条件下，算法帮助技术人员解决了现实中大量的问题，本章将会介绍其中一些算法。本章阐述的一些数据结构和算法对于本科生来说比较高深。但是，软件专业人员能够很容易地理解这些数据结构和算法的原理，并且将其应用到实践中。

经典问题

计算几何和点、线和面这样的几何物体关系非常紧密。我们通过如下定义来判断是否一个问题为计算几何问题：(a) 输入的数据类型。(b) 将要进行的计算。(c) 任务是静态的还是动态的。我们利用这些分类方法，针对不同的跨领域问题采取不同的技术来改善性能。

输入数据

计算机和问题必须定义输入数据。最常用的几种输入数据是：

- 二维平面点集。
- 二维平面线段集。
- 二维平面矩形集合。
- 二维平面任意多边形集合。

二维结构（线、矩形和圆）有着对应的三维结构（面、立方体和球面）和 n 维结构（例如超平面、超立方和超球体）。高级计算几何问题均可以扩展到更高的维度。

我们需要三个以上的维度吗？

一个计算几何问题：在一个29维面上有1400万个点，寻找每个点的最近邻接点。

现实世界的服务：eHarmony婚介服务 (<http://www.eharmony.com>) 宣称他们是“第一家使用科学方法来匹配单身男女的网络婚介服务商”。这家公司使用的匹配相容系统注册了专利（美国专利号6735568），eHarmony将会预测两个人是否能够长期包容对方。这个系统的所有用户（据估计在2007年2月时达到了1400万）需要填写一个436个问题的关系调查问卷。eHarmony然后从29个方面（即29个维度）来计算两个人的匹配度。在2003年11月的一份报告中，eHarmony指出91%的用户有10个或者更多的匹配人选。

数据置信度问题：一个包含1400万份记录的输入文件，每份记录有29个文本或者数字值的字段。有些值可能错误或者缺失。我们会用其他的近似记录来代替这份有问题的记录。

我们首先将阐述一系列计算几何的核心接口，然后再介绍一些实现这些接口的类。在实现这些接口的基础上，所有的算法都会保证最大的便携性。

本章的算法基于图9-1所示的核心思想：

IPoint

表示最基本的笛卡儿点 (x, y) ，坐标是双浮点精度。我们编写了一个比较器，首先对 x 坐标从左至右进行排序，然后再对 y 从下到上进行排序。

IRectangle

表示笛卡儿平面上的矩形，能够判断是否和一个点相交或者是否包含另外一个 IRectangle。

ILineSegment

表示笛卡儿平面上的一个有限长度线段，并且有固定的起始点。在“一般位置”上，除了水平线（这种情况下，最左边的点被当作起始点），起始点的 y 坐标要高于终止点的 y 坐标。它能够返回是否和其他的 ILineSegment 或者 IPoint 对象相交，在考虑线段的方向时，还能返回一个 IPoint 对象在其左边还是右边。

这些处理二维的核心思想可以很自然地扩展到如何处理多维度，如图9-2所示。

IMultiPoint

表示一个 n 维的笛卡儿平面点，每个坐标值是双浮点精度，能够返回和另外一个相同维度的 IMultiPoint 的距离，也可成一个坐标值的数组，用来优化某些算法的性能。

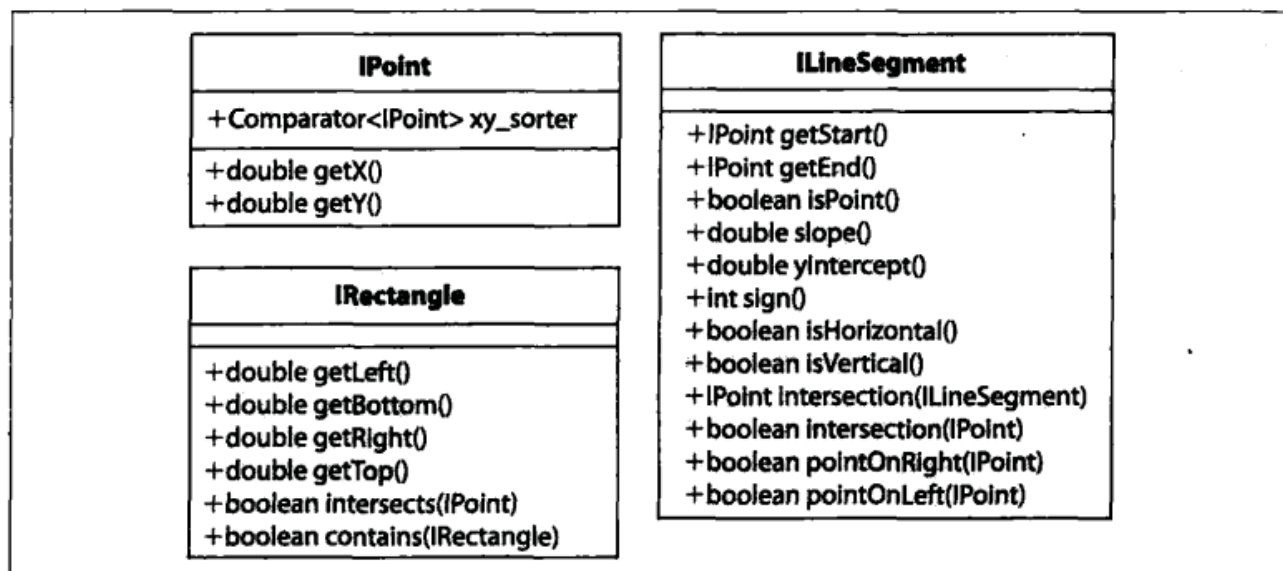


图9-1: 计算几何的核心接口

IHypercube

表示一个 n 维的固体, 能够判断是否和一个IMultiPoint相交或者是否包含另外一个同一维度上的IHypercube。

IMultiLineSegment

表示在 n 维笛卡儿平面上一个有限长度线段。

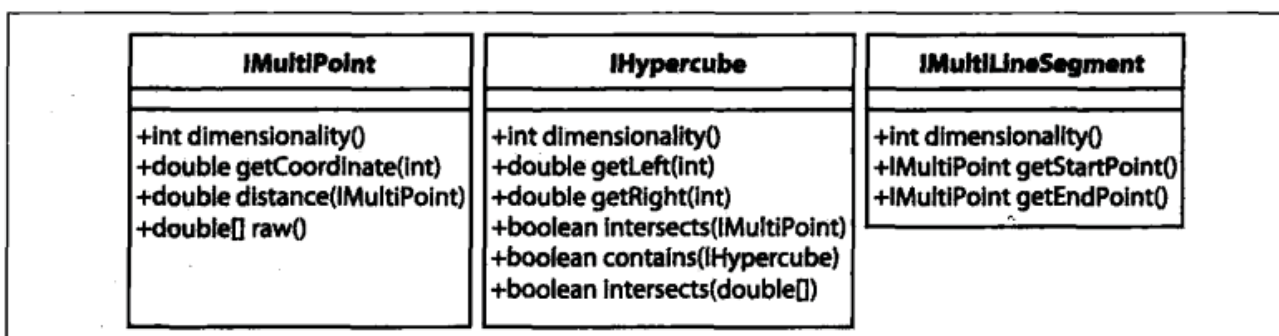


图9-2: 多维数据的接口

一般来说, 点的坐标值都是实数, 所以我们的实现必须用浮点值来存储数据。在20世纪70年代, 由于计算机硬件的限制, 浮点计算的开销要大于整数计算的开销。但是现在的计算机已经不存在这个问题。第2章讨论了更重要的几个问题, 例如舍入错误, 这些问题和浮点计算息息相关, 对本章算法产生了不利影响。

最后, 有些计算几何的算法需要整数值的间距, 如图9-3所示。

IInterval

表示一个半开区间 $[left, right)$, $left$ 和 $right$ 为整数值, 并且最小单位为1, 它包含了

*left*值，但是不包含*right*值。它能够返回和一个整数值的关 系（左边、右边或者相交）。

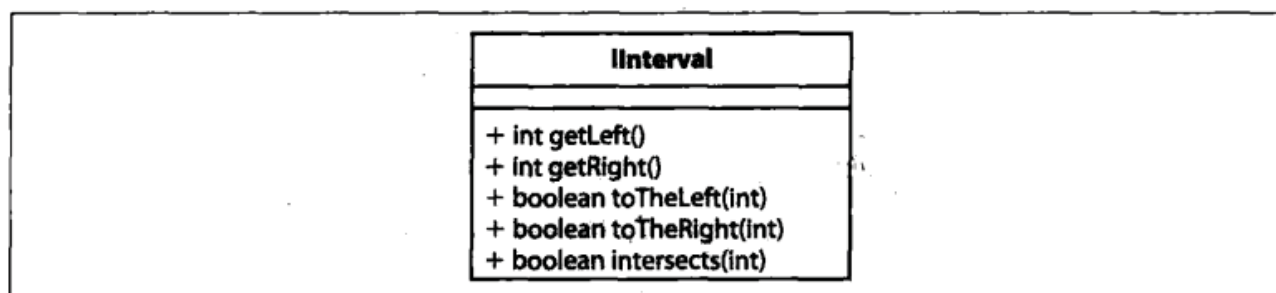


图9-3：表示间距[*left*, *right*)的接口

我们会编写一系列类实现这些接口类型，用来初始化对象（例如，类*TwoDPoint*实现了*IPoint*和*IMultiPoint*接口）。

计算

计算几何明显和空间问题相关，见表9-1。有三种主要的任务。

查询

寻找数据中满足条件限制（例如最近或者最远）的元素，这类任务和第5章的查找算法直接相关。

计算

在输入数据（例如线段）上进行一系列的计 算，得到一些包含了输入数据元素的几何结构（例如线段的相交点）。计算的结果就是问题的结果。

预处理

将巨大规模的输入数据组织成一种数据结构，来计算一系列问题的结果。也就是说，预处理任务的结果就是这一系列问题的输入。

表9-1：计算几何问题及其应用

计算几何问题	现实世界的应用
寻找给定点的最近点和最远点	寻找给定位置的最近加油站 寻找最近位置的医院，并且计算出最坏情况下所需的到达时间
计算一个多边形是否为简单多边形（例如，两条不连续的边不能有公共顶点）	科学家给濒危物种带上能够报告位置的无线发射器，希望能够知道动物的最频繁运动轨迹

表9-1：计算几何问题及其应用（续）

计算几何问题	现实世界的应用
计算包含点集的最小圆。计算一个点集的内部不包含特定点的最大圆	统计学家使用不同的技术来分析数据。前面的圆可以用来寻找簇，而后面的圆则是用来寻找异常或者缺失的数据
寻找线段、圆、矩形以及任意多边形集合的交集	在超大规模集成电路（VLSI）中检查设计规则

在本章的剩下部分，我们将会抽象各种计算几何问题然后进行计算。这些技术的应用范围并不仅仅限于几何问题，在现实中也有很多应用。例如，扫描技术能够帮助我们对无序对象（例如点、线和多边形）排序然后进行后续处理。

任务的本质

静态任务是处理一个固定的输入集合。但是，有两个非常重要的动态思想改变了解决问题的方法。

- 是否任务需要多次在同一个输入数据集上进行计算？如果是的话，那么我们需要对输入数据进行预处理来改善远期性能。
- 输入数据是否在处理一次之后会改变？如果会的话，那么我们需要选择能够应对这种变化的数据结构。

动态任务要求数据结构能够较好地应对变化的输入。固定长度的数组也许适用于静态的任务，但是对于动态任务就显得力不从心，此时就需要链表或者栈来实现通用化。回忆一下第6章存储图的两种方法，邻接表和邻接矩阵，仔细考虑一下它们的优点和局限性。

假设

得到或者理解一个高效的解的最有效方法是从输入（或者任务）相关的假设和不变性入手。例如：

- 给定一个线段集，存在垂直或者水平线段吗？
- 给定一个点集，是否有三点共线？如果不存在，那么认为这些点处于一般位置，能够简化很多算法。
- 输入包含的点是否均匀分布？或者这些点在某种方式上聚成簇，可能会导致某些算法处于最坏情况？

本章描述的大多数算法都有值得我们去寻求正确方法的异常边缘情形，我们将在代码中描述这些情形。

计算几何经典问题

我们通过阐述一些经典问题来解释计算几何关注的领域。程序员要高效解决这些问题，首先必须理解一些在其他领域也非常有用的关键数据结构和算法技术。我们将会在阐述问题的过程中简单地描述和分析解决问题的原始算法，得知算法的期望运行时间。在本章剩余部分，我们将会提出一些更加优雅和高效的算法，这些算法也同样能够解决这些问题。我们再次重申，我们可以通过如下方法来改进算法：（a）利用问题的特殊信息。（b）在算法支持的范围内，寻找最合适的数据结构。

凸包

在二维平面上有一个点集 P ，凸包是能够包含 P 中所有点的最小凸多边形，在任意两点间画一条线段，凸包的边都属于这个线段集。有 h 个点的凸包是顺时针计算出来的，从 L_0 到 L_{h-1} ，第一个点 L_0 是 P 中最左边的点（虽然任意的点都可以作为起始点）（注1）。顺着凸包上三个相邻的点 L_i, L_{i+1}, L_{i+2} ，路线将会向右转，这种性质对于 L_{h-2}, L_{h-1}, L_0 也成立。

给定 n 个点，存在 $C(n,3)$ ，或者：

$$\binom{n}{3} = \frac{n(n-1)(n-2)}{6}$$

个不同的三角形。如果一个点 $p_i \in P$ 被 P 中其他三个点组成的三角形所包含，那么它不可能是凸包的一部分（例如，图9-4的点 p_6 被包含在 p_4, p_7 和 p_8 组成的三角形中）。一个穷举算法可以从这些三角形中一个一个地剔除点，然后得到凸包。一旦我们知道那些点可以组成凸包，那么接下来以最左边的点为起始点（ p_0 ）画一条线 L_0 ，然后将每个点 p_i 和 p_0 连成一条线 L_i ，根据 L_i 和 L_0 的夹角从小到大进行排序，然后依次将这些点连起来即可。我们需要考虑到共线的情况。

很明显这种方法非常低效，在三角形检测时就需要 $O(n^4)$ 的时间。接下来我们将会给出一个高效的凸包扫描算法，能够在 $O(n \log n)$ 的时间内计算出凸包。

计算线段集的相交

在一个二维平面上有一个线段集 S ，找出所有线段之间的交点。我们也许只想知道是否存在至少一个交点（可能会在找到第一个交点之后终止算法）。在图9-5的例子中，我们找到了两个交点。图9-6的穷举算法在 $O(n^2)$ 的时间内能够找到所有的交点。

注1：如果 P 中多个点的 x 坐标相同，那么 L_0 选择 y 坐标最小的点作为起始点。

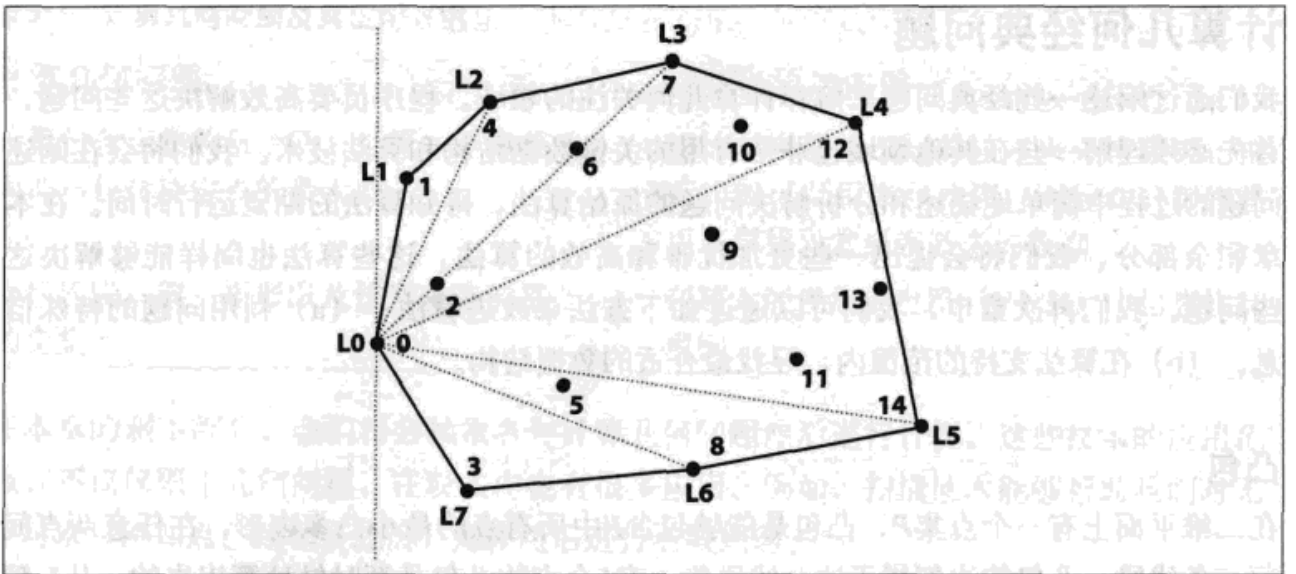


图9-4：平面上的点集及其凸包

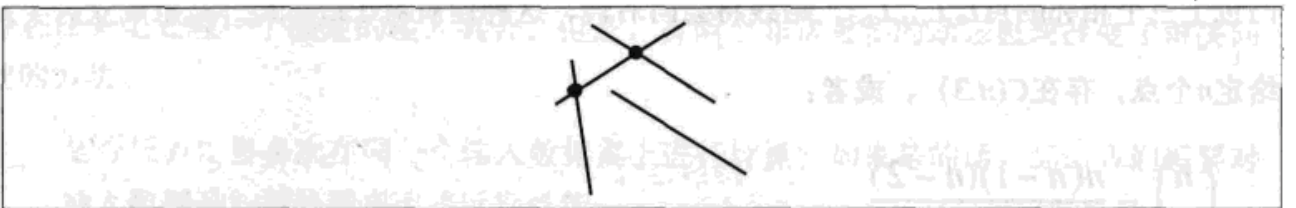


图9-5：有两个交点的线段集


穷举探测			 穷举
最好情况	平均情况	最坏情况	
$O(n^2)$	$O(n^2)$	$O(n^2)$	
intersections (S) <ol style="list-style-type: none"> 1. foreach $s_1 \in S$ do 2. foreach $s_2 \in S - \{s_1\}$ do 3. $p = \text{intersection point of } s_1 \text{ and } s_2$ 4. if (p exists) then record (p, s_1, s_2) end			

图9-6：穷举探测详解

例9-1是穷举探测的一个实现。有 $C(n,2)$ 个线段对，或者：

$$\binom{n}{2} = \frac{n(n-1)}{2}$$

n 个可能的线段对。对于每个线段对，下面的实现都会输出交点。

例9-1: 穷举探测实现

```
public class BruteForceAlgorithm extends IntersectionDetection {

    public Hashtable<IPoint, ILineSegment[]> intersections
        (ILineSegment[] segments) {

        startTime();
        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]);
                }
            }
        }
        computeTime();
        return report;
    }
}
```

其主要步骤需要 $O(n^2)$ 的时间。寻找两条线段的交点可能需要三角函数或者除法，这些都是开销非常大的操作，此外，如第3章所述，这些操作经常会引发一些舍入的错误。我们将会使用一种新的方法来检测交点，这种方法只使用加法、减法、乘法以及比较操作（Cormen 等，2001）。

在之前我们不清楚是否可以通过一些改进得到优于 $O(n^2)$ 的性能，所以本章将会介绍一种创新型的线段扫描算法，其在平均情况下能够在 $O((n+k)\log n)$ 时间内得到结果（ k 是得到交点的个数）。

寻找最近邻点

我们可能要回答这类问题：“在二维平面上有一个点集 P ，以欧几里德距离为点之间的距离， P 集合中离 x 最近的点是哪个？”当然 x 可能不在 P 中。

我们计算 x 和 P 中所有其他点的距离。这需要 $O(n)$ 步。如第5章所述，二叉树能够帮助查找不去考虑那些不可能成为解的点。我们用树结构切分平面上的点，减少查找时间。查询所节省的时间能够补偿预处理的额外开销，我们能够在 $O(\log n)$ 的时间内得到结果。如果查询的次数比较少，那么 $O(n)$ 的穷举方法可能更加好一些。

回答基于范围的查询

这种查询并不是在二维平面内寻找特定点，而是希望知道给定的矩形区域是否包含所有的点。最简单的实现需要花费 $O(n)$ 的时间来得到解。

处理最邻接点的数据结构在这里也同样适用。我们如果希望能够在 $O(n)$ 的性能上继续改进，必须做到（a）从候选点集中舍弃一部分。（b）候选点集需要包含结果可能有的点。我们在这里使用kd树这种数据结构，进行递归的遍历查询，这种实现的性能是：

$$O(\sqrt{n} + r)$$

r 是得到的点的数目。

总结

下述代码遵从表9-2的API定义。同时这个表也总结了本章描述的算法的性能。

表9-2：本章讨论问题的API定义

问题	API描述
凸包扫描	<pre>public interface IConvexHull { /** 计算凸包点的有序序列。 */ IPoint[] compute (IPoint[] points); }</pre> <p>最简单的解：$O(n^4)$ Average-case CONVEX HULL SCAN: $O(n \log n)$</p>
线段交点	<pre>public abstract class IntersectionDetection { /** 计算交点。 */ public abstract Hashtable<IPoint, ILineSegment[]> intersections (ILineSegment[] segments); }</pre> <p>最简单的解：$O(n^2)$ Average-case LINE SWEEP: $O((n+k) \log n)$</p>
最近点	<pre>public class KDTree { /** 返回最近的点。 */ public IMultiPoint nearest (IMultiPoint target); }</pre> <p>最简单的解：$O(n)$ Average-case kd-tree NEAREST NEIGHBOR: $O(\log n)$</p>
范围查询	<pre>public class KDTree { /** 返回区域中的点。 */ public ArrayList<IMultiPoint> search (IHypercube space); }</pre> <p>最简单的解：$O(n)$ 平均情况下，基于kd树的范围查询：$O(\sqrt{n} + r)$，r是结果集中的点数。</p>

凸包扫描

我们选择了迭代的方法，希望能够得到高效计算凸包的算法，如图9-8所示。首先，根据最后找到的两个凸包点连成的线段，在非凸包点中，计算出角度偏差最小的点，这就是下一个得到的凸包点。当计算出的部分凸包包含 h 个点时，我们需要计算 $n-h$ 个点的夹角，然后才能找出哪个是下一个凸包点，很显然每一步都是必不可少的。

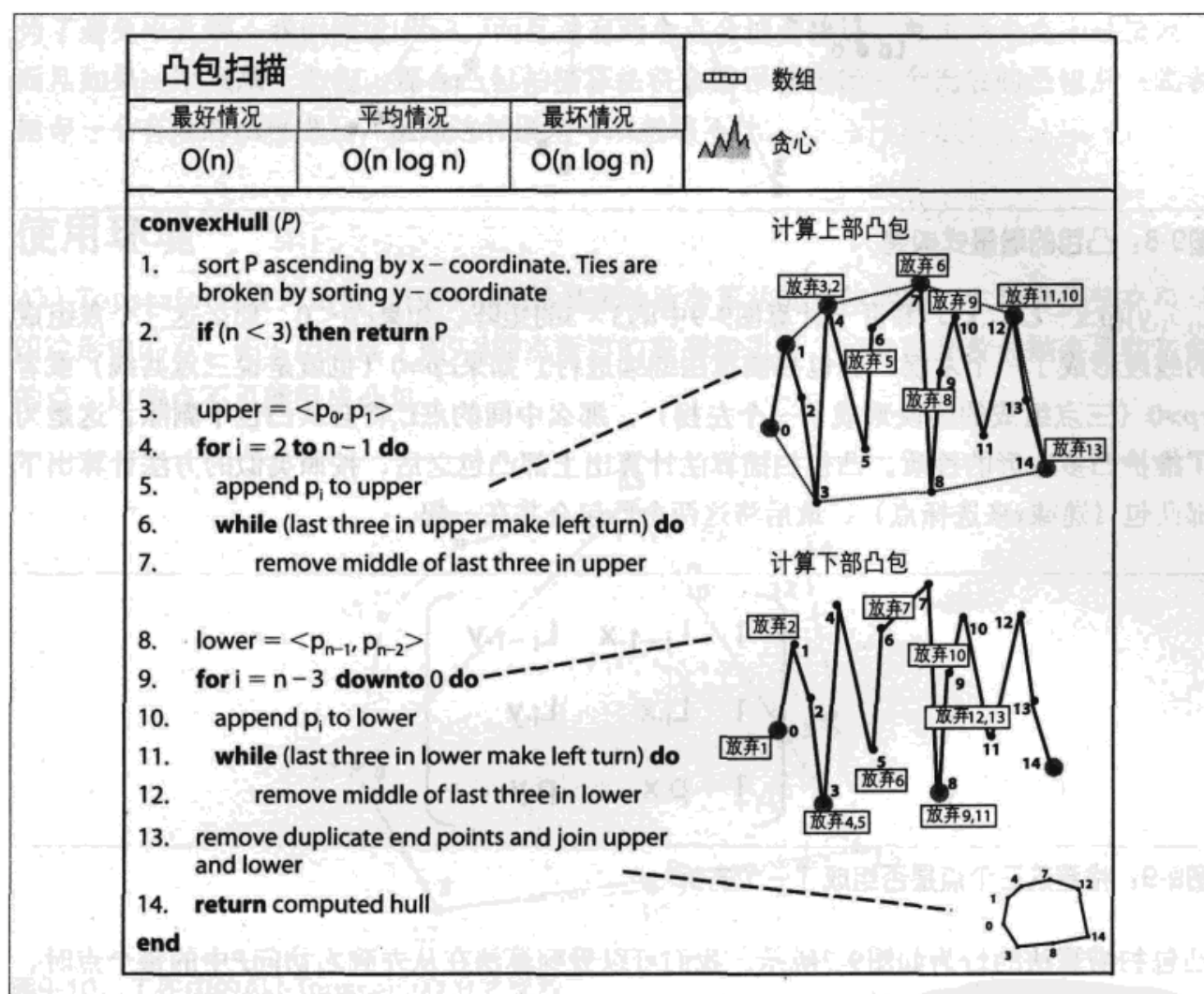


图9-7：凸包扫描详解

Andrew凸包扫描算法将整个问题分成两个部分，分别构建一个上部凸包和一个下部凸包。首先，点集 P 中所有的点按照 x 坐标排序（ x 坐标相同时，对 y 坐标进行排序）。图9-8中的点已经按照 x 坐标从左至右标上了数字。上部凸包从 P 中的最左边两个点开始构建。凸包扫描算法首先对上部凸包进行扩展，在 P 中找到这样一个点 p ，排序后，其前一个位置即上部凸包的最后一个点 L_i 。

如果三个点 L_{i-1} 、 L_i 和 p 组成的两条线段形成了一个右折，那么凸包扫描算法将会把这个点 p 加入到这个凸包中。这里计算了一个叉积 $cp = (L_i.x - L_{i-1}.x)(p.y - L_{i-1}.y) - (L_i.y -$

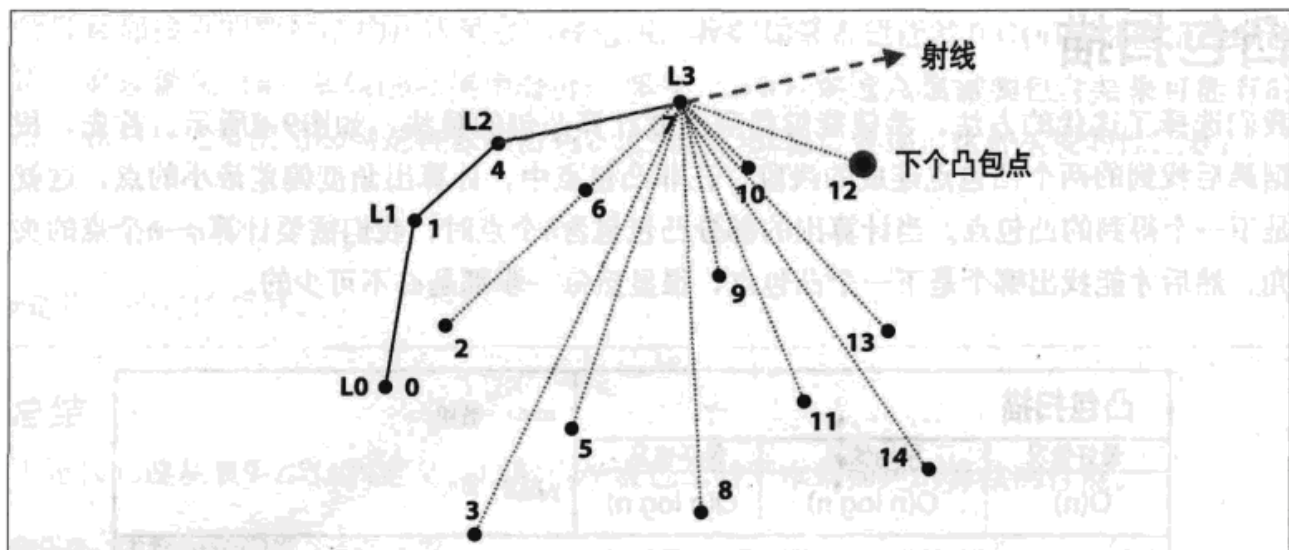


图9-8：凸包的增量式构建

$L_{i-1}.y(p.x - L_{i-1}.x)$ ，等价于计算图9-9中的 3×3 的矩阵。如果 $cp < 0$ ，那么这三个点组成的线段形成了一个右拐，凸包扫描算法继续进行。如果 $cp = 0$ （也就是说三点共线）或者 $cp > 0$ （三点组成的线段形成了一个左拐），那么中间的点 L_i 将会从凸包中删除，这是为了维护凸多边形的性质。凸包扫描算法计算出上部凸包之后，按照类似的方法计算出下部凸包（递减 x 来选择点），最后将这两个凸包合并在一起。

$$\begin{bmatrix} 1 & L_{i-1}.x & L_{i-1}.y \\ 1 & L_i.x & L_i.y \\ 1 & p.x & p.y \end{bmatrix}$$

图9-9：检查这三个点是否组成了一个右折

凸包扫描算法的行为如图9-7所示。我们可以看到算法在从左到右访问 P 中的每个点时，还是做出了一些错误的决定。但是在最后一步，算法不断地删除掉最后三个点中不满足条件的中点，从而纠正了之前犯下的错误。

输入/输出

输入

平面上的二维点集 P 。

输出

一个有序的表 L ，其包含组成凸包的 h 个顶点，这些顶点的顺序是顺时针顺序。点 L_0, L_1, \dots, L_{h-1} 组成的多边形便是凸包， h 是点的数目，这个凸包有 h 条边： $\langle L_0, L_1 \rangle, \langle L_1, L_2 \rangle, \dots, \langle L_{h-1}, L_0 \rangle$ 。

假设

为了避免平凡解，我们假设 $|P| \geq 3$ 。而且没有两个点会过于接近。如果两个点太过接近，而且如果一个点属于凸包，那么凸包扫描算法将会错误地选择一个无效的凸包点（或者抛弃一个有效的凸包点），但是这种误差可以忽略不计。

使用环境

Akl-Toussaint启发式函数（1978）能够显著地改善算法的总体性能。它抛弃那些在极端四边形中的点。图9-10描绘了图9-4的点所得的极端四边形，启发式函数会抛弃那些灰色的点，这些点不可能组成凸包。

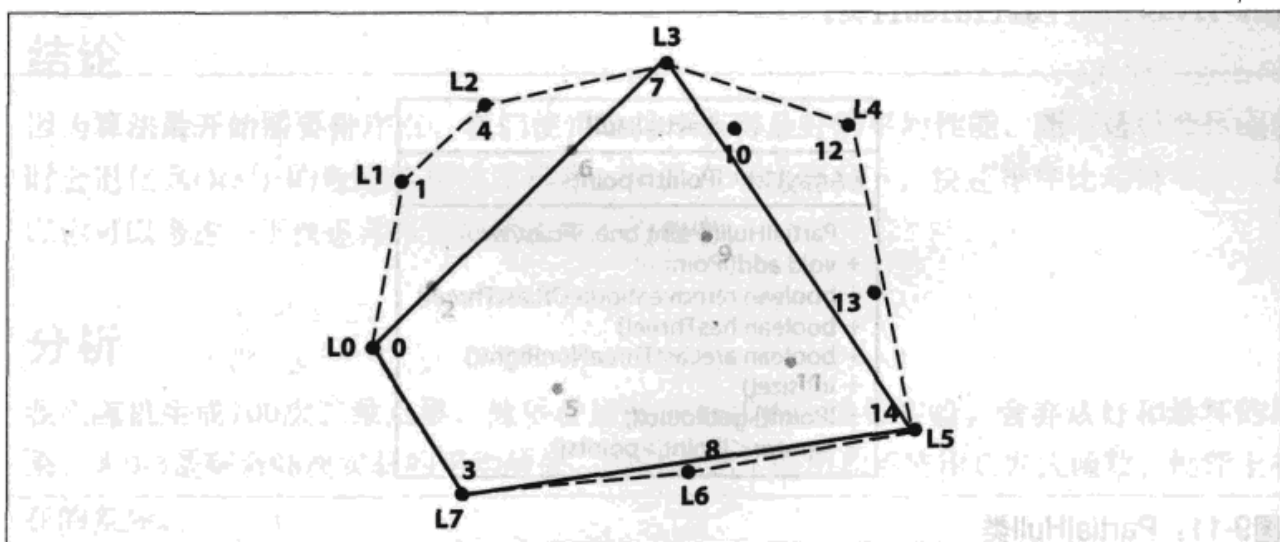


图9-10：工作中的Akl-Toussaint启发式函数

检查是否一个点 p 在极端四边形中，我们可以通过如下方法：想象一条从 p 开始到一个极端点 $(p.x, -\infty)$ 的线段 s ，计算 s 和四边形四条边相交的次数（注2），如果只有一次，那么 p 是属于四边形内，可以被抛弃。这个计算需要的步数是固定的，所以性能是 $O(1)$ ，也就是说Akl-Toussaint启发式函数的性能是 $O(n)$ 。对于大量的随机点来说，这个启发式函数能够在排序前删除掉几乎一半的点，排序的性能将会明显提高。

注2：具体实现处理了一些特殊情况，例如当线段 s 恰好与极端四边形的某个终点相交。

驱动因素

凸包扫描算法仅仅需要一些原始的操作（如乘法和除法），因此比另外一种凸包算法“Graham扫描”（Graham, 1972）要简单得多。Graham算法需要使用三角函数。如果凸包扫描算法使用快速排序来排序点，那么其性能将会受快速排序影响，比如几乎有序的数据情况下，快速排序的性能严重退化（详见第4章“快速排序”）。凸包扫描算法能够支持巨大规模的点集，因为它并不是采用的递归策略。例9-2的实现使用了数组，如果使用链表，那么我们不得不采用插入排序，这将会使性能退化到 $O(n^2)$ 。使用平衡二叉树而不是数组存储输入可以省去排序，虽然实现起来可能更加复杂一些，但是还是合算的。如果输入的点均匀分布的，那么我们可以使用桶排序，在 $O(n)$ 的时间内将点排好序，算法最后的总体性能也是 $O(n)$ ，这种实现是最快的。在稍后的“分析”一节中，我们将会分析各种实现的性能，这些实现的代码都可以在代码库中找到。

解决方案

例9-2的代码告诉我们凸包扫描算法如何计算上部凸包。最后的凸包合并上下两个凸包。图9-11归纳出了PartialHull类。

PartialHull
ArrayList<IPoint> points
+ PartialHull(IPoint one, IPoint two)
+ void add(IPoint p)
+ boolean removeMiddleOfLastThree()
+ boolean hasThree()
+ boolean areLastThreeNonRight()
+ int size()
+ IPoint[] getPoints()
+ Iterator<IPoint> points()

图9-11: PartialHull类

例9-2: 凸包扫描算法

```
public class ConvexHullScan implements IConvexHull {

    public IPoint [] compute (IPoint[] points) {
        // 排序x坐标，如果x坐标相等，排序y坐标。
        int n = points.length;
        new HeapSort<IPoint>().sort(points, 0, n-1, IPoint.xy_sorter);
        if (n < 3) { return points; }

        // 从最左边两个点开始计算上凸包。
        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();
        }
    }

    // 从最右边两个点开始计算下凸包。
    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();
        }
    }

    // 合并时删除掉重复的点。
    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;
}
}

```

结论

因为算法最开始需要排序点，我们使用堆排序获得最好的平均性能，而不是冒着性能有时会退化到 $O(n^2)$ 的危险使用快速排序。但是在平均情况下，快速排序比堆排序好，所以你可以考虑一下快速排序最坏情况发生的概率。

分析

我们随机生成100次二维点集，然后在这些数据基础上进行实验，舍弃最好和最坏的结果，表9-3是剩余98次实验的平均结果。同时对比了使用和不使用启发式函数，性能上存在的差异。

在 $[0,1]$ 这个单位正方形上，均匀分布着 n 个点，表9-3的统计结果也许能够给出为什么凸包扫描算法能够如此高效的深层次原因。

表9-3：实验的结果（单位：毫秒）

n	凸包中的 平均点个数	计算的 平均时间	启发式函数 删除掉的 平均点个数	启发式 函数计算的 平均时间	使用启发式 函数算法的 平均执行时间
4,096	21.65	8.95	2,023	1.59	4.46
8,192	24.1	18.98	4,145	2.39	8.59
16,384	25.82	41.44	8,216	6.88	21.71

表9-3: 实验的结果 (单位: 毫秒) (续)

n	凸包中的 平均点个数	计算的 平均时间	启发式函数 删除掉的 平均点个数	启发式 函数计算的 平均时间	使用启发式 函数算法的 平均执行时间
32,768	27.64	93.46	15,687	14.47	48.92
65,536	28.9	218.24	33,112	33.31	109.74
131,072	32.02	513.03	65,289	76.36	254.92
262,144	33.08	1168.77	129,724	162.94	558.47
524,288	35.09	2617.53	265,982	331.78	1159.72
1,048,576	36.25	5802.36	512,244	694	2524.30

随着输入数据规模的增长, 越来越多 (几乎有一半) 的点被启发式函数删除掉。更令人惊讶的是, 规模如此之大的点集, 组成的凸包的点个数却如此之少。表9-3的第2列证明了Preparata和Shamos的论断 (1985): 他们认为构成凸包的点个数应该为 $O(\log n)$, 这是非常令人吃惊的结论。如此之大的点集, 组成凸包的点个数如此之少, 那么每个点组成凸包的概率将会非常小。

凸包扫描算法的第一步是使用标准的基于比较的排序方法, 其性能开销为 $O(n \log n)$ 。之前我们提到过, 如果输入的点本来是有序的, 那么这一步可以跳过。计算上部凸包 (图9-7中第4~7行) 的循环需要处理 $n-2$ 个点。内部的while循环 (第6~7行) 最多也只会执行 $n-2$ 次; 计算下部凸包 (第9~12行) 也同样如此。因此凸包扫描算法其余步骤的总时间是 $O(n)$ 。

当凸包扫描算法计算叉积时, 可能会出现浮点运算的误差。因此我们不会严格地判断 $cp < 0$, PartialHull采用的是计算 $cp < \delta$ 代替 $cp < 0$, 希望能够努力消除误差, 此时 $\delta = 10^{-9}$ 。

变种

如果输入的点已知有序, 那么凸包扫描算法的排序步骤可以跳过, 在这种情况下, 凸包扫描算法的性能是 $O(n)$ 。如果输入的点均匀分布的, 那么我们可以使用桶排序 (详见第4章“桶排序”), 也能够得到 $O(n)$ 的性能。另一种计算凸包的变种是QuickHull (Preparata和Shamos, 1985), 受到快速排序的启发, 使用分治思想计算凸包。

最后我们还要讨论另一类变种。我们已经讨论过, 凸包扫描算法在构建上部凸包时, 并不是真正地需要一个有序的数组, 它仅仅是从 x 坐标最小到最大, 遍历 P 中所有的点。如果我们使用二叉堆来存储 P , 那么我们只需要重复地从堆中删除最小的元素即可。如果

使用链表来存储删除的点，那么这些点可以轻易地从链表中以逆序的方式回滚。这类变种的代码（在图9-12中标记为堆）在本书的代码库中可以找到。

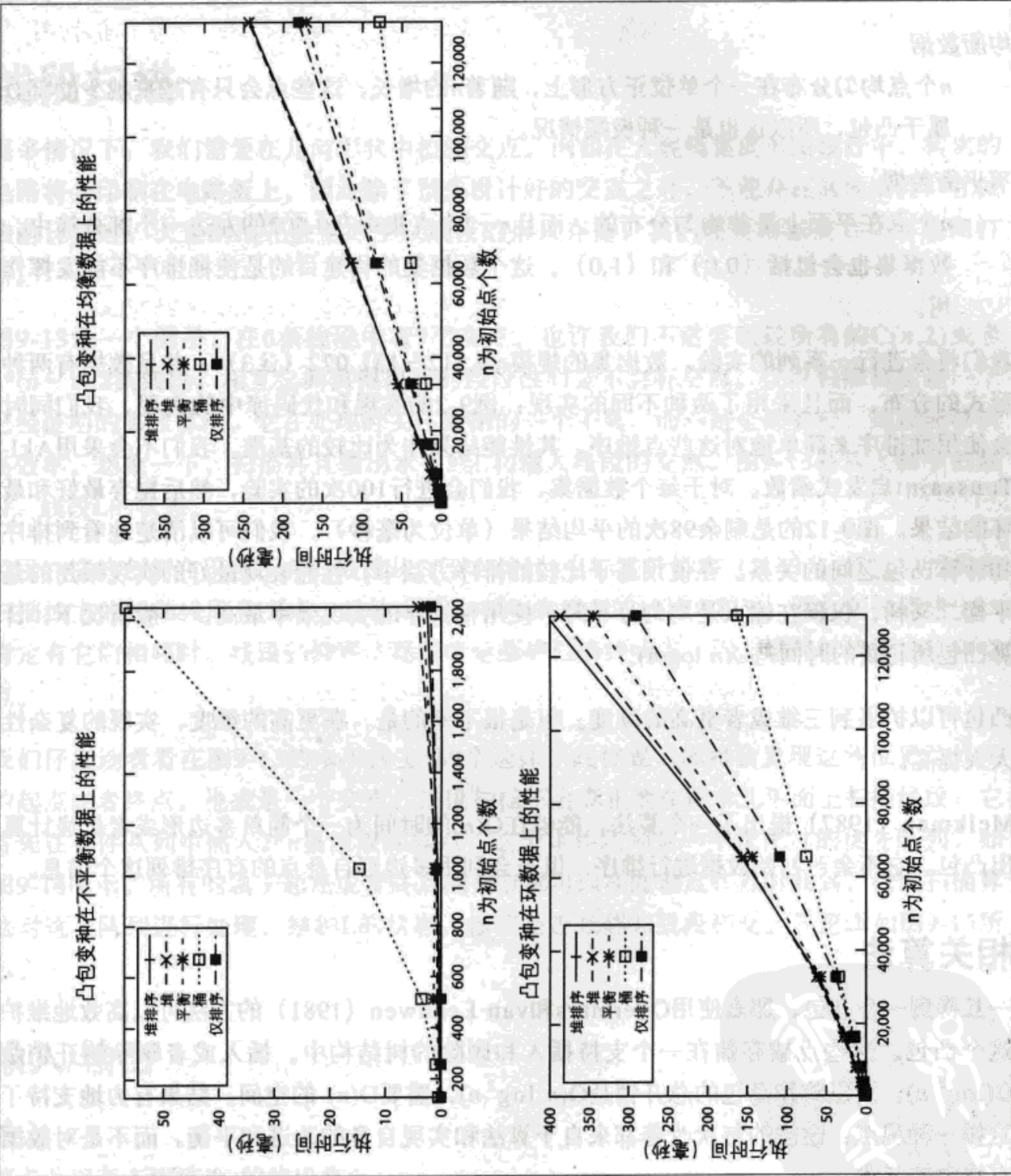


图9-12: 凸包变种的性能

图9-12的性能结果基于以下三种数据。

环数据

在一个单位圆上均匀分布着 n 个点。所有这些点都属于凸包，所以这是一种极端情况。

均衡数据

n 个点均匀分布在一个单位正方形上，随着 n 的增长，这些点会只有越来越少的部分属于凸包，所以这也是一种极端情况。

不平衡数据

n 个点在平面上是非均匀分布的，而且 $n-2$ 个点集中在0.502的左边一小部分簇中。数据集也会包括(0,0)和(1,0)。这个数据集的构建目的是使桶排序不能发挥作用。

我们将会进行一系列的实验，数据集的规模 n 从512~131 072（注3），并且数据有两种形式的分布，而且采用了两种不同的实现：例9-2的实现和代码库中的实现。我们同时会使用堆排序来简单地对这些点排序，其性能结果作为比较的基准。我们不会采用Akl-Toussaint启发式函数。对于每个数据集，我们会进行100次的实验，然后抛弃最好和最坏的结果。图9-12的是剩余98次的平均结果（单位为毫秒）。我们可以清楚地看到排序和计算凸包之间的关系。在使用基于比较的排序方法中，性能表现最好的实现采用的是平衡二叉树。仅仅在输入是均匀分布时，使用桶排序的实现效率最高。一般情况下，计算凸包所花费的时间是 $O(n \log n)$ 。

凸包可以扩展到三维或者更高的维度。但是很不幸的是，在更高的维度，实现的复杂性大大提高。

Melkman (1987) 提出了一个算法，能够在 $O(n)$ 的时间为一个简单多边形或者折线计算出凸包。它不会对初始数据进行排序，但是会利用多边形自身点的有序排列这个信息。

相关算法

一旦得到一个凸包，那么使用Overmars和van Leeuwen (1981) 的方法可以高效地维护这个凸包。这些点被存储在一个支持插入和删除的树结构中。插入或者删除的开销是 $O(\log^2 n)$ ，所以构建凸包的总开销是 $O(n \log^2 n)$ ，需要 $O(n)$ 的空间。结果有力地支持了这样一种观点：性能的每次改善都来自于算法和实现自身的改进和平衡，而不是对数据有特定的要求。

最早计算凸包的一个算法叫做Graham扫描，这个算法在1972年提出，需要使用到简单的三角函数性质。使用图9-9检查右折的方法，一个恰当的实现只需要简单的数据结构和基

注3： 我们将不平衡数据集的规模限制在2048，因为桶排序很快会退化到 $O(n^2)$ 的性能。

本的数学计算。Graham扫描首先将所有的点按照和最左下方的点的夹角进行排序，它能在 $O(n \log n)$ 的时间内计算出凸包。需要注意的是夹角相同的点，其顺序是按照距离来决定的。

线段扫描

很多情况下，我们需要在几何形状中检测交点。例如在大规模集成电路设计中，精密的电路将会印制在电路板上，因此除了预先设计好的交点之外，不能存在其他交点。在做旅游计划时，大量的路在数据库中以线段的形式存储，我们需要根据交点来将路网打断。

图9-13是一个例子，在6条线段中有7个交点。也许我们不需要比较所有的 $C(n, 2)$ 或者 $n*(n-1)/2$ 条线段。相互之间离得很远的线段很明显不会有交点。线段扫描算法是一个已经证明的高效算法，它在处理时关注数据的一个子集，而不是全部数据，这样能够提高效率。想象一下，扫描并且输出水平线 L 和输入线段的交点。图9-13是从上到下扫描时，线段 L 的状态。

线段扫描的创新在于将所有的线段从左至右按照 y 坐标进行排序（注4）。线段只可能和扫描线上的相邻线段有交点。具体来说，有两条相交的线段 S_i 和 S_j ，那么在扫描线上，肯定有它们相邻时。线段扫描算法能高效地维护扫描线状态，因此它能够快速地检测交点。

我们仔细地看看在图9-13的水平线上那9个选择的位置，你将会发现这些位置是线段的起点或者终点，也或是一个交点。线段扫描不会真正地在笛卡儿平面上扫描线段，它首先在事件队列中插入 $2*n$ 条线段的端点，这个事件队列是一个优化过的优先队列，如图9-14所示。所有的属于起点或者终点的交点都可以在处理这些点时找到。线段扫描算法对这个队列进行处理，维护 L 的状态，检测是否相邻的线段相交。其逻辑如图9-15所示。

输入/输出

输入

笛卡儿平面上规模为 n 的线段集 S 。

注4： 水平线段被认为左端点（高于）右端点。

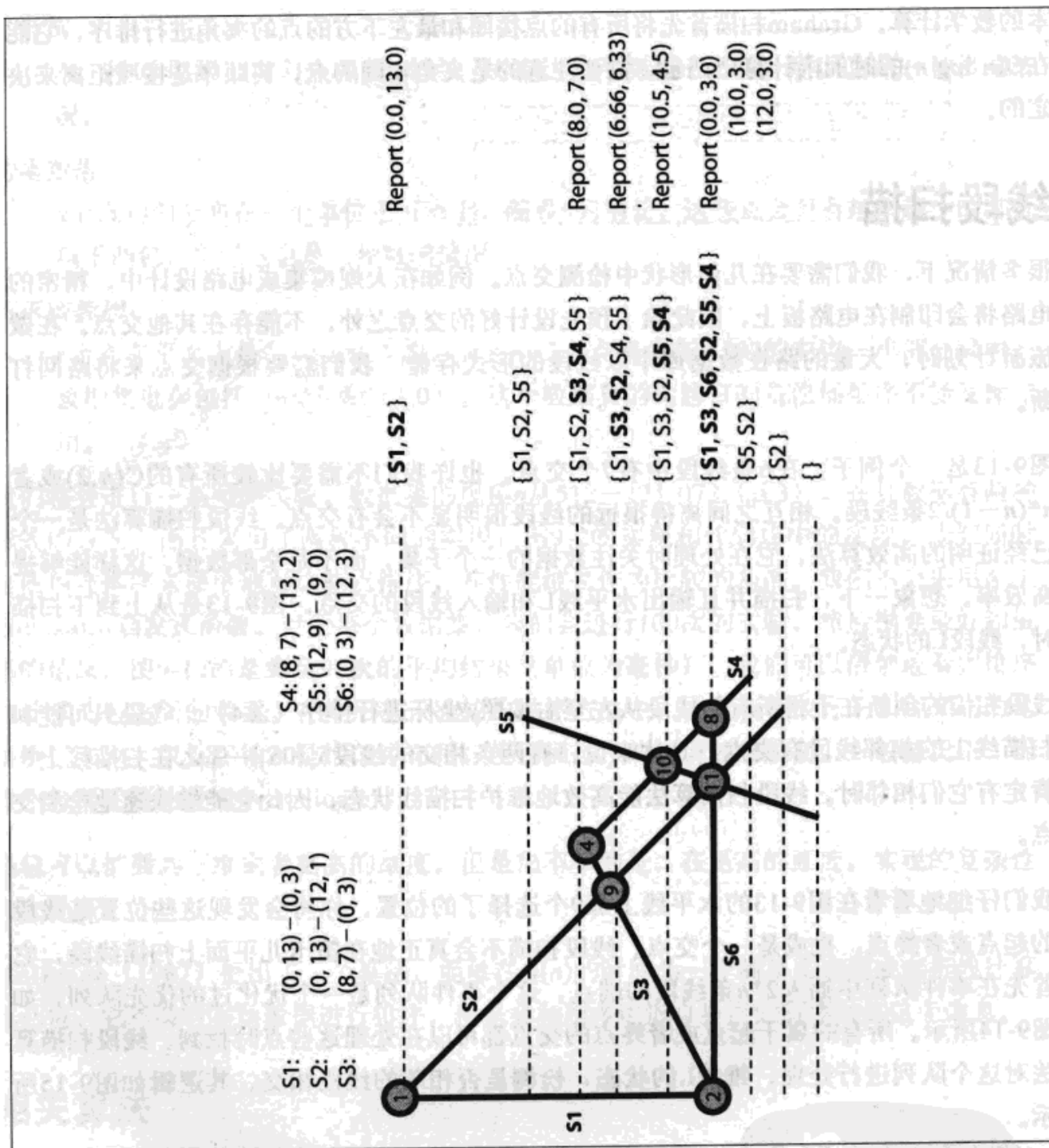


图9-13：在6条线段中检测到7个交点

输出

这些线段的 k 个交点（如果存在）。

假设

在 S 中不会出现重复的线段。没有两条线段会共线（也就是说，重叠或者有相同的斜率）。算法需要仔细地计算和精准地排序线段，这样才能支持垂直线段或者水平线段。没有一条线段只有一个点（例如，一条起点和终点相同的线段）。

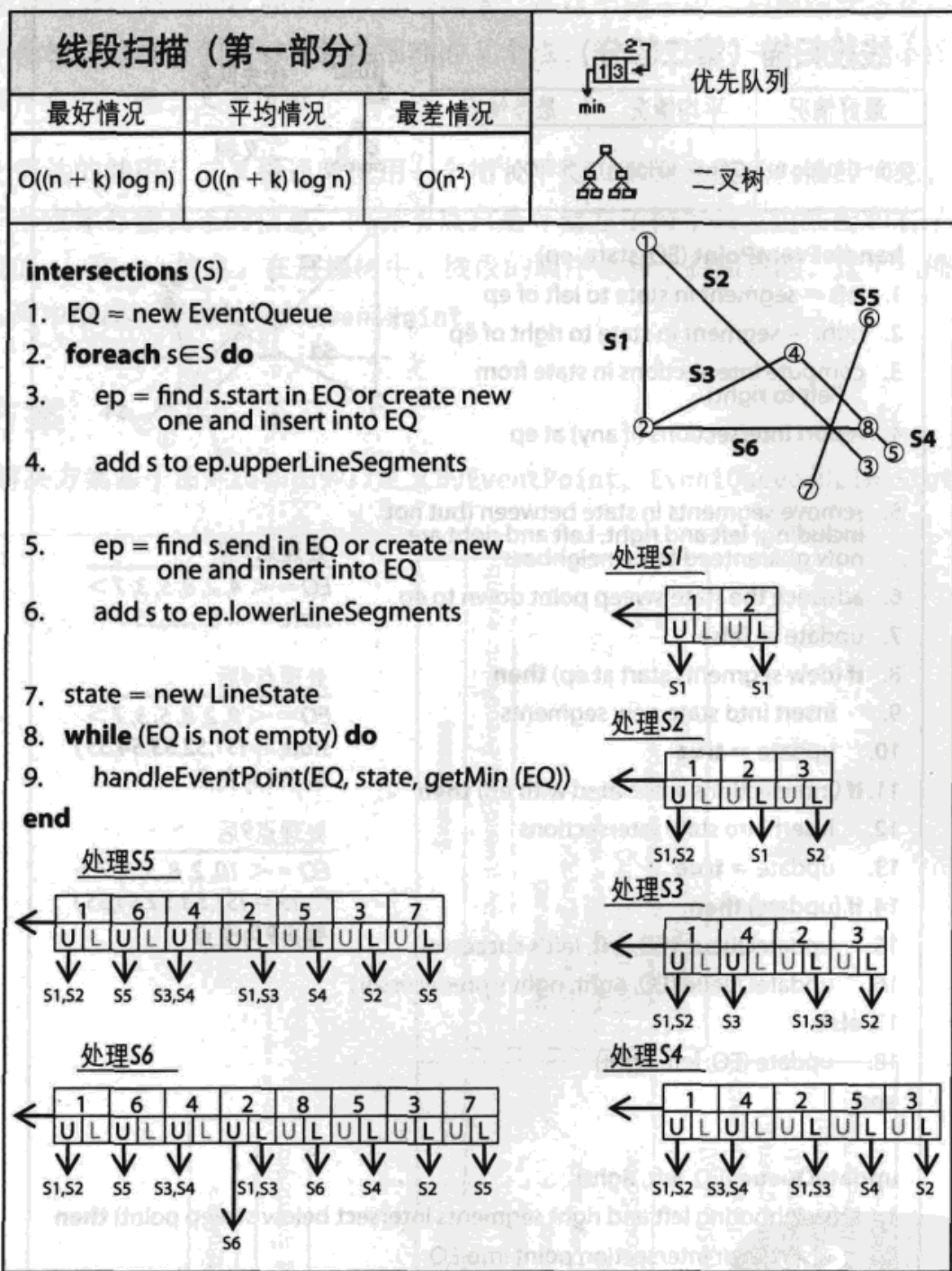


图9-14：线段扫描算法详解（第一部分）。

使用环境

当交点的期望数目远远少于线段的个数时，此算法能够轻而易举地得到比穷举算法好很多的性能。如果有大量的交点，那么算法的记录管理功能会更加重要。

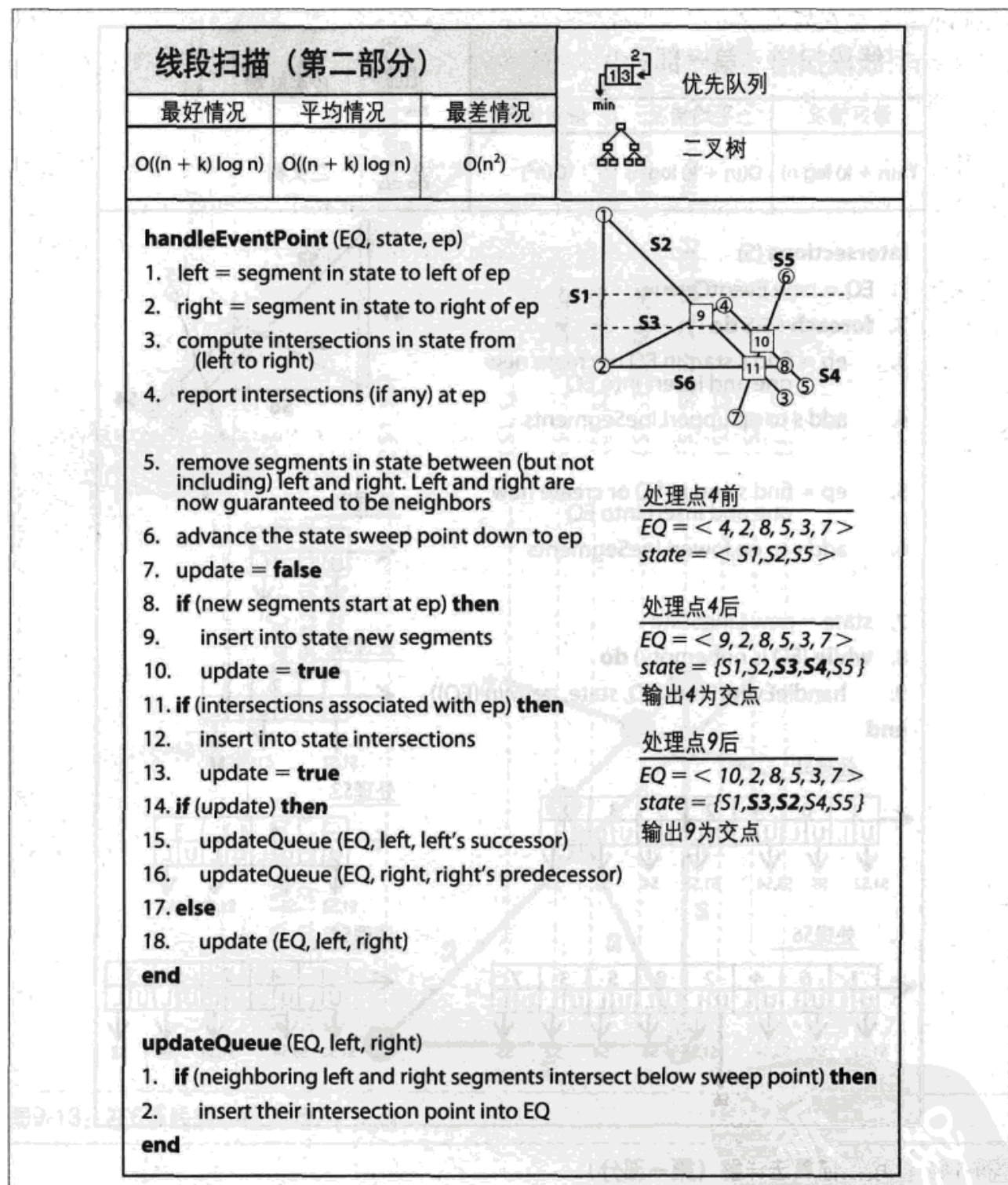


图9-15：线段扫描算法详解（第二部分）

驱动因素

一个基于扫描的方法是非常有用的，因为你可以高效地构建扫描线状态，并维护事件队列。在线段扫描算法中，要考虑大量的特殊情况，因此代码会比穷举算法复杂很多。如果不是特别希望性能高效，我们不推荐采用此算法。

线段扫描算法会在处理的过程中生成部分结果。在这个例子中，扫描线状态是一个线段的二叉平衡树，这样我们能够根据扫描线的某个点，得到线段的一个顺序。事件队列也可以是事件点的一棵二叉查找树。

为了简化算法的编码，二叉树通常使用一个增长平衡二叉树来表示扫描线状态，这棵树只有叶子节点才存储真正的信息。内部节点只是存储左子树中最左边线段和右子树中最右边线段的min和max信息。在这棵树中，线段的顺序是基于扫描点的，这个扫描点就是从优先队列中取出，正在处理的EventPoint。

解决方案

例9-3的解决方案基于图9-16和图9-17定义的EventPoint、EventQueue和LineState类。

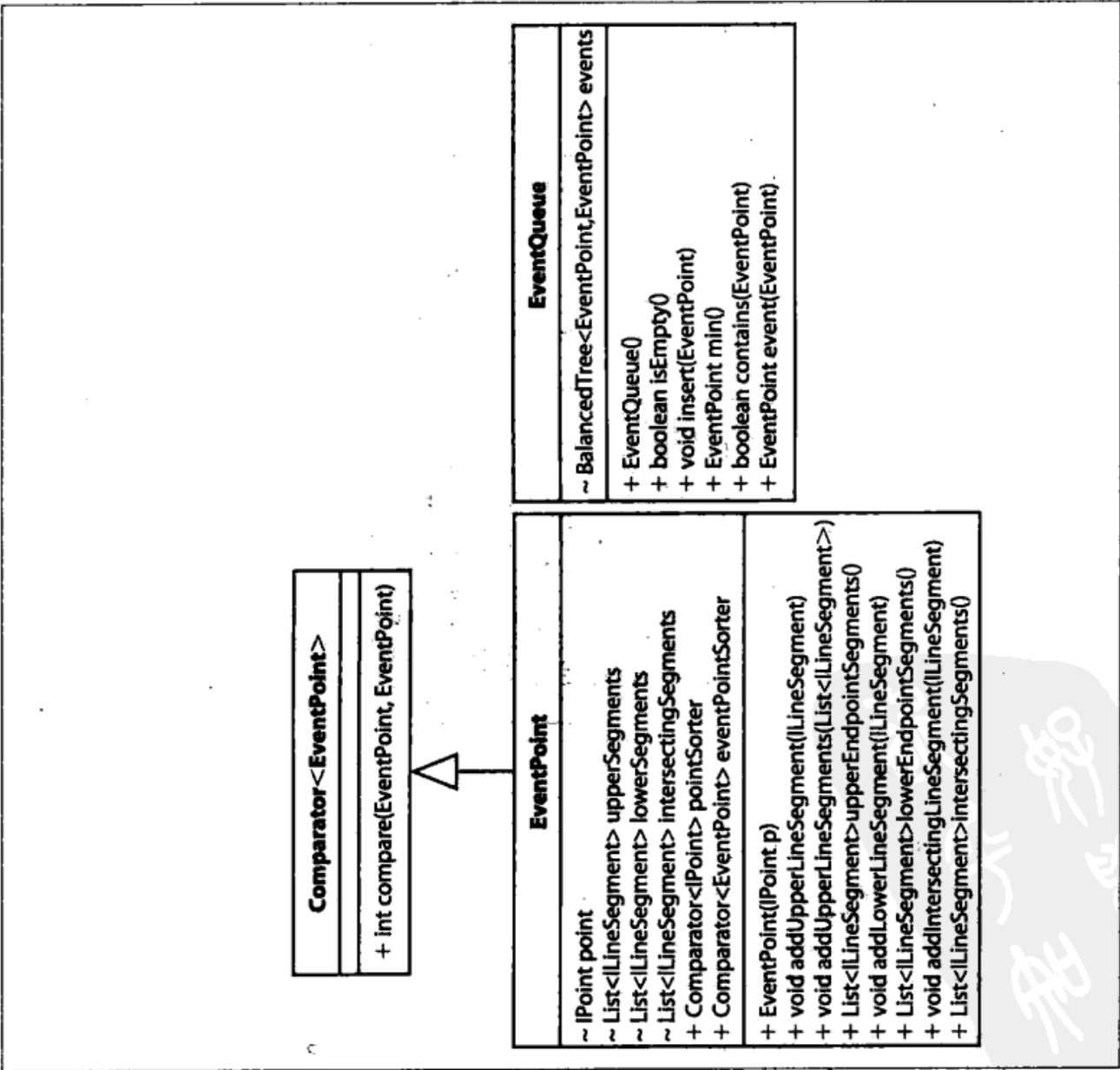


图9-16: EventPoint类

LineState
~ IPoint sweepPt + Comparator<ILineSegment> seg_order ~ AugmentedBalancedTree<ILineSegment> state + IPoint getSweepPoint() + AugmentedNode<ILineSegment> leftNeighbor(EventPoint) + AugmentedNode<ILineSegment> rightNeighbor(EventPoint) + void determineIntersecting(EventPoint, AugmentedNode<ILineSegment> left, AugmentedNode<ILineSegment> right) + AugmentedNode<ILineSegment> successor(AugmentedNode<ILineSegment>) + AugmentedNode<ILineSegment> pred(AugmentedNode<ILineSegment>) + void insertSegments(List<ILineSegment>) + void deleteRange(AugmentedNode<ILineSegment> left, AugmentedNode<ILineSegment> right)

图9-17: LineState类

例9-3: 线段扫描算法Java实现

```

public class LineSweep extends IntersectionDetection {
    // 存储线段扫描状态和事件队列。
    LineState lineState = new LineState();
    EventQueue eq = new EventQueue();

    // 计算数组中所有线段之间的交点。
    public Hashtable<IPoint, ILineSegment[]>
    intersections (ILineSegment[] segs){

```

```

// 从线段中构建事件队列。需要确保点的唯一性。
for (ILineSegment ils : segs) {
    EventPoint ep = new EventPoint(ils.getStart());
    EventPoint old = eq.event(ep);
    if (old == null) { eq.insert(ep); } else { ep = old; }

    // 向ep中添加上部线段。
    ep.addUpperLineSegment(ils);

    ep = new EventPoint(ils.getEnd());
    old = eq.event(ep);
    if (old == null) { eq.insert(ep); } else { ep = old; }

    // 向ep中添加下部线段。
    ep.addLowerLineSegment(ils);
}

// 从上至下扫描，处理队列中的每个事件点。
while (!eq.isEmpty()) {
    EventPoint p = eq.min();
    handleEventPoint(p);
}

// 返回计算出的交点。
return report;
}

// 通过更新扫描线状态和寻找交点来处理事件。
private void handleEventPoint (EventPoint ep) {
    // 寻找线段。
    AugmentedNode<ILineSegment> left = lineState.leftNeighbor(ep);
    AugmentedNode<ILineSegment> right = lineState.rightNeighbor(ep);

    // 寻找相邻线段的交点，放入ints中。当前事件点的上部线段放入ups，下部线段放入lows。
    // 如果多于一个线段和当前事件点相关，那么就找到了一个交点。
    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.p, new List[]{lows, ups, ints});
    }

    // 删除left之后的所有东西，直到后继是right。然后更新扫描点，
    // 交点有序。仅有ups和ints会被插入元素。
    lineState.deleteRange(left, right);
    lineState.setSweepPoint(ep.p);
    boolean update = false;
    if (!lows.isEmpty()) {
        lineState.insertSegments (lows);
        update = true;
    }
    if (!ints.isEmpty()) {
        lineState.insertSegments (ints);
    }
}

```



```

        update = true;
    }

    // 如果状态告诉我们当前事件点上没有交点, 那么检查在扫描线下相交的
    // left和right线段, 然后更新事件队列。
    // 否则的话, 如果存在一个交点, 那么我们检查left及其后继和right及其前驱。
    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); }
    }
}

// 任何在扫描线下的交点都被作为事件点加入队列。
private void updateQueue (AugmentedNode<ILineSegment> left,
                          AugmentedNode<ILineSegment> right) {
    // 是否两个邻近线段相交。确保在扫描线下的交点不会添加两次。
    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); }
    }
}
}
}

```

EventQueue最开始存储 $2*n$ 个EventPoint对象, 每个对象存储的是以这个点为起点的ILineSegment (上部线段) 和为终点的ILineSegment (下部线段)。当线段扫描算法发现线段之间存在交点时, 表示交点的EventPoint对象就会插入到EventQueue中, 直到它在扫描线下方时。这样, 所有的交点都能够被找到, 而且不会存在重复的点。如果交点已经存在于EventQueue中, 那么在队列中, 相交的信息将会更新, 而不是重复插入 (注5)。

在图9-14中, 当表示线段 S_6 低点的事件点被插入到优先队列中 (注6), 那么线段扫描算法会将 S_6 作为下部线段存储, 处理这条线段时, 线段扫描算法会将 S_4 作为相交线段存储。在一个更加复杂的例子中, 当表示线段 S_2 和 S_5 交点的事件点插入到优先队列中时, 优先队列不会存储额外的信息。一旦这个事件点被处理时, 优先队列会把 S_6 、 S_2 和 S_5 作为相交线段存储。

线段扫描算法的计算引擎包含在图9-17的LineState类中。LineState在从上到下扫描时, 维护当前的扫描点。pointSorter这个比较器在笛卡儿平面从上至下, 从优先队列中, 返回EventPoint对象。线段扫描算法的工作实际上是在LineState的

注5: 这就是LINESWEEP必须能够决定是否优先队列中包含一个特定的EventPoint对象的原因。

注6: 事实上是最右边的端点, 因为 S_6 是水平的。

determineIntersecting方法中：遍历左边和右边之间的这些线段来寻找交点。这些类的详细信息可以在本书所附的代码库中找到。

结论

线段扫描算法的性能是 $O((n+k)\log n)$ ，因为它会在扫描点处理时，对活动线段进行重排序。如果这一步需要超过 $O(\log s)$ 的时间， s 是状态中的线段数目，那么整体性能将会退化到 $O(n^2)$ 。例如，如果线段状态只是简单地使用双向链表（一个能够快速找到前驱和后继线段的数据结构），那么由于需要在表中找到正确的插入位置，插入操作将会增加到 $O(s)$ 。并且随着线段集 S 的增长，性能的退化需要引起更多的注意。

类似地，事件队列需要能够高效地查询某个事件点是否已经在队列中。使用基于堆的优先队列实现，在java.util.PriorityQueue中提供同样使得算法退化到 $O(n^2)$ 。我们不希望口口声声说是 $O(n \log n)$ 的算法，最后的实现却是 $O(n^2)$ 的性能！

分析

线段扫描算法会插入 $2*n$ 个线段的端点到一个事件队列，这个队列是一个修改过的优先队列，它能够支持在 $O(\log q)$ 的时间内（ q 是队列中元素个数）完成如下操作：

min

从队列中删除最小的元素。

insert (e)

将元素插入到有序队列中的适当位置。

member (e)

是否给定的元素已经在队列中。这个操作并不一定需要一个泛型的优先队列类型。

在事件队列中，所有的点都是唯一的。如果要插入的事件点已经存在，那么它的信息会被加入到已经存在的事件点的信息中。因此将图9-13的点都插入到优先队列中之后，优先队列的规模是8个事件点。

线段扫描算法从上到下进行扫描，在线段状态中添加或者删除线段。在图9-13中，有序的线段状态从左至右表示处理了事件点之后，和扫描线相交的线段。为了正确计算交点，线段扫描算法需要计算出线段状态中给定线段 S_i 左边或者右边的线段。线段扫描算法使用了一种增量平衡二叉树，这样能在 $O(\log t)$ 的时间内（ t 是树中元素的个数）完成如下操作：

insert (s)

将线段插入树中。

delete (s)

从树中删除线段。

previous (s)

返回有序线段集合，给定线段的前驱线段。

successor (s)

返回有序线段集合，给定线段的后继线段。

为了正确地维护线段的有序性，线段扫描算法在检测到线段 S_i 和 S_j 存在交点时，交换这两条线段的顺序，很幸运的是，这个操作在 $O(\log t)$ 的时间内就能完成，只需要简单地更新一下扫描线和删除并再插入线段 S_i 和 S_j 。例如，在图9-13中，当找到交点(6.66,6.33)时进行交换。

算法的最初阶段将会构建一个包含了 $2*n$ 个点的优先队列。这个事件队列需要支持查询某个点是否已经存在的操作，因此，我们不能简单地使用堆这种通常用来组成队列的结构。由于队列要求有序，所以我们必须定义一个点的顺序。如果 $p1.y > p2.y$ 或 $p1.y = p2.y$ 时， $p1.x < p2.x$ ，那么 $p1 < p2$ 。这个队列的规模不会超过 $2n+k$ ， k 是 n 条线段的交点个数。

线性扫描算法会将所有在扫描线下方的交点加入到事件队列，当扫描线到达交点时，相交线段的顺序会被交换。一开始所有的交点都会在扫描线下方，所以我们会扫描所有的交点。

线段扫描算法会处理所有的事件点，一条线段在上端点被扫描时加入到状态结构中，并且在下端点被扫描时从状态结构中删除。因此线段状态存储的线段不会超过 n 条。在线段状态中查找线段的操作能在 $O(\log n)$ 的时间内完成，而且不会执行超过 $O(n+k)$ 次操作，所以算法的开销是 $O((n+k)\log(n+k))$ 。由于 k 不会大于 $C(n,2)$ 或者 $n*(n-1)/2$ ，所以这个等式可以化简：

$$O((n+k) \log(n+k)) = O((n+k) \log(n*(n+1)/2))$$

使用对数的性质，可以得到：

$$\log(n*(n+1)/2) = \log n + \log(n+1) - \log 2 \leq \log n + \log(2n) - 1 \leq 2 \log n$$

结果是：

$$O((n+k) \log(n*(n+1)/2)) = O(2*(n+k) \log n)$$

因此总体性能还是 $O((n+k)\log n)$ 。

因为线段扫描算法的性能和输入的复杂程度有关（例如：交点的总数，任何时候扫描线

维护的线段平均个数)，我们使用特定的问题和输入数据来做算法的基准测试。我们现在讨论两个特定问题。

一个很有意思的数学问题是如何使用一些牙签和一张纸，计算 π 的一个近似值（叫做 Buffon's needle problem）。假设牙签长度为10，在纸上画一些垂直线段，每条线段相隔 d ，线段长 len 。随机将 n 根牙签投向纸上，牙签和线段会有 k 个交点。一根牙签有可能和一条线段相交的概率是 $(2*len)/(\pi*d)$ （注7）。

当交点的个数远远少于 n^2 时，穷举检测算法将会浪费大量的时间在检测那些不相交的线段之间是否存在交点（见表9-4）。当存在大量交点时，采用哪种算法，其决定因素便是 LineState 在线段扫描时维护的线段个数。如果这个值比较小，那么我们需要使用线段扫描算法来获得更好的性能。

表9-4：比较两个算法在解决 Buffon's needle problem 上所花费的时间（单位：毫秒）

n	线段扫描	穷举	平均相交个数	π 的估计值
16	479.5918	0	1.02	3.147541
32	153.0612	0	2.14	3.047619
64	469.3878	0	3.99	3.324675
128	316.3265	795.9184	8.53	3.213389
256	1255.102	3346.939	17.83	3.237092
512	4448.98	10357.14	40.48	3.191688
1,024	8448.98	44408.16	97.15	3.223505
2,048	22122.45	172234.7	263.07	3.086274
4,096	61632.65	685438.8	789.46	3.152588

最坏情况下，线段之间的交点是 $O(n^2)$ 个，此时线段扫描算法的性能将会特别差，因为它在维护线段状态结构上开销过大。我们从表9-5的结果中可以看到穷举检测算法轻易地胜过了线段扫描算法，这里 n 条线段最多会有 $n*(n-1)/2$ 个交点。

表9-5：最坏情况下，两种算法的性能比较（单位：毫秒）

n	线段扫描（平均）	穷举（平均）
2	0	0
4	0.1531	0
8	0.6429	0

注7：<http://mathworld.wolfram.com/BufonsNeedleProblem.html/>.

表9-5：最坏情况下，两种算法的性能比较（单位：毫秒）（续）

n	线段扫描（平均）	穷举（平均）
16	0.6327	0
32	6.4082	0.6327
64	36.7959	0.6429
128	218.2551	3.2245
256	1566.4898	34.5918
512	9791.4592	209.0102

变种

一个有趣的变种算法是只需要返回是否存在交点，而不是找出所有的焦点，这个算法在检测两个多边形是否相交时很有用，需要花费 $O(n \log n)$ 的时间。在平均情况下，这个算法能够快速地寻找到第一个交点。另一个变种则要处理这样的情况：输入的线段被标记为蓝色或者红色，我们只需要寻找到不同颜色线段之间的交点（Palazzi和Snoeyink，1994）。

最近点查询

在二维笛卡儿平面上存在一个点集 P ，我们可能需要进行这样的查询“在 P 中离 x 点最近的点是哪个？”。 x 可能并不属于 P 。这些查询也可以扩展到 n 维的空间。初级的实现方式是探测所有 P 中的点，是一种线性的算法。由于我们一开始就能知道 P ，所以应该能够对 P 的信息构建合理的数据结构，在查询时抛弃一些不必要的点，加快查询的速度。

也许我们可以将平面分成 k^2 个格子，每个格子的大小是固定的， $m \times m$ ，如图9-18a。在 P 中有10个点被放入了9个相邻的格子中。当寻找 x 的最邻近点时，在它周围的格子中寻找。如果格子是非空的，那么我们只需要在这些格子中寻找和某些圆相交的点，这些圆的半径是：

$$\sqrt{2}m$$

在这个例子中，目标的格子中没有任何点，所以我们需要检查邻近的三个格子。这个方法可能会带来潜在的性能下降，因为a大多数格子可能为空。b算法仍然需要查找大多数邻近格子。简而言之，将 P 划分到多个格子中并不是一个很高明的方法。

另外一种解决方案便是构建Voronoi图（Preparata和Shamos，1985），将平面划分成 n 个区域 R_i （ $0 \leq i < n$ ）， R_i 的定义为“离 p_i 最近的点集”。因此这个区域是能够根据情况扩

展到所需的大小（注8）。在一个二维平面中，这样的区域是一个多边形（在更高的维度，每个区域是一个 n 维的多面体）。图9-18b是使用图9-18a的点得到的Voronoi图。一旦这个结构构建起来，那么只需找到邻近的区域 R_i 就能找到最邻近点。平均情况下，构建Voronoi图需要 $O(n \log n)$ 的时间，但是实现起来却非常复杂。使用Voronoi图，寻找最邻近点只需要 $O(\log n)$ 。

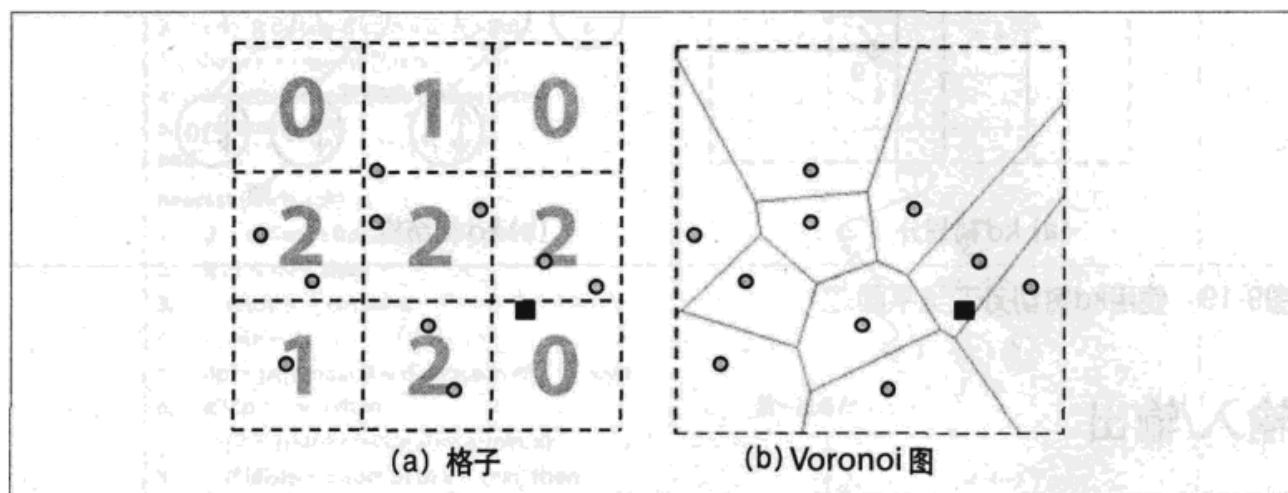


图9-18：划分格子和Voronoi图两种方法

在图9-19 (a) 中，我们使用kd树来表示图9-18的19个点。kd树顾名思义，是沿着坐标系统的垂直坐标轴，切分一个 k 维的平面。在接下来的讨论中，我们假设这棵树是基于二维平面的，但是我们也要注意，kd树可以扩展到任意高维。一棵kd树是一个递归的二叉树结构，其节点包含多个点以及一个坐标标记（ x 或者 y ），这个标记表示切分线。根节点表示矩形范围是 $[x_{\text{low}}=-\infty, y_{\text{low}}=-\infty, x_{\text{high}}=+\infty, y_{\text{high}}=+\infty]$ 。通过点 p_1 画一条垂直切分线 V 。左子树表示是 V 左边的区域，右子树表示右边的区域。通过点 p_2 画一条水平切分线 H ，根节点的左子节点会根据这条分割线将 V 的左部切分成上下两个部分。根节点的左子节点表示的区域是 $[-\infty, -\infty, p_1.x, +\infty]$ ，右子节点表示 $[p_1.x, -\infty, +\infty, +\infty]$ 。这些区域都高效地嵌套在一起。我们看到一个祖先节点的区域完全包含其所有后代的区域。

当kd树被正确地构建起来之后，在第 i 层的节点表示的矩形区域是第 $i+1$ 层节点表示的矩形区域的两倍。这个性质使得最邻近点算法（图9-20）能够以 $O(\log n)$ 的性能高效地查找目标点，因为它能够抛弃整棵不可能包含最近点的子树。在接下来的“范围查询”一节中，我们将会看到这个实现是如何高效地进行范围查询的。

注8：在Voronoi图中，凸包上的区域是“开口”的，直到遇到图的边界，而内部节点的区域面积有限。

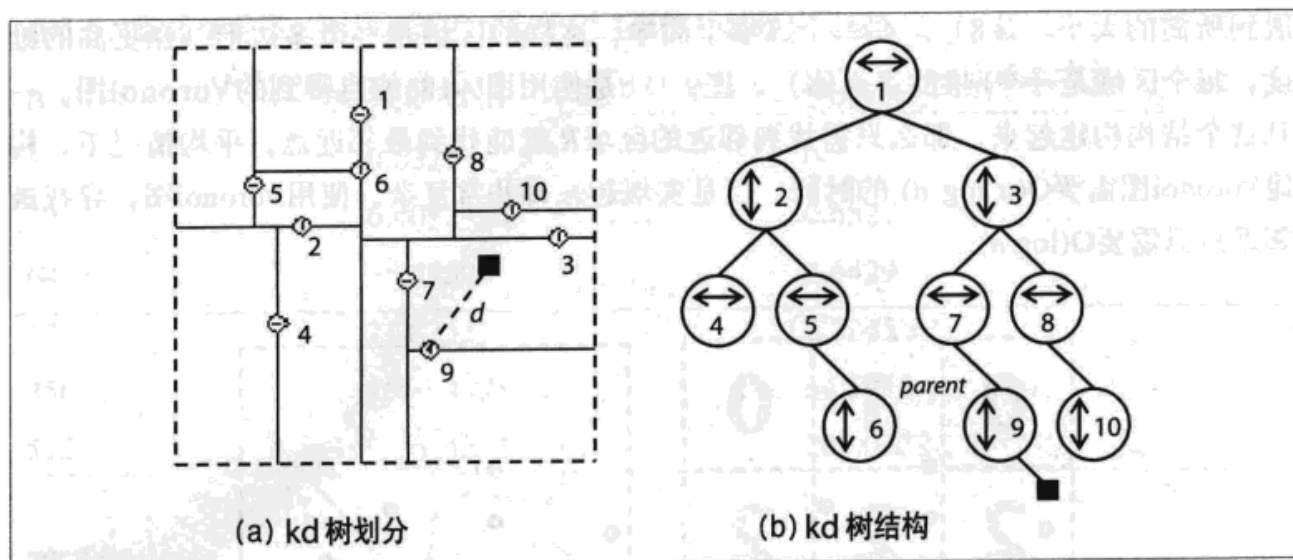


图9-19：使用kd树切分二维平面。

输入/输出

输入

一个二维的点集 P 。一些最近点的查询（实现并不知道）。

输出

计算出一棵kd树。对于每个查询，计算在 P 中离 x 点最近的点。

假设

可能会有两个点非常近，在计算时会因为浮点的误差导致算法选择了错误的点，但是，由于这两个点如此接近，所以不会对结果产生致命的影响。

使用环境

当比较此实现和穷举实现时，有两个重要的因素需要考虑：（1）构建kd树的开销。（2）在kd树结构中寻找点 x 的开销。权衡这两个因素，我们可以做：

维数

随着维度的增加，构建kd树的开销也越来越大。一些权威人士认为，当维度超过20时，这个方法不如穷举高效。

输入的点数

当点很少时，构建kd树的开销大于其在性能方面的改善。

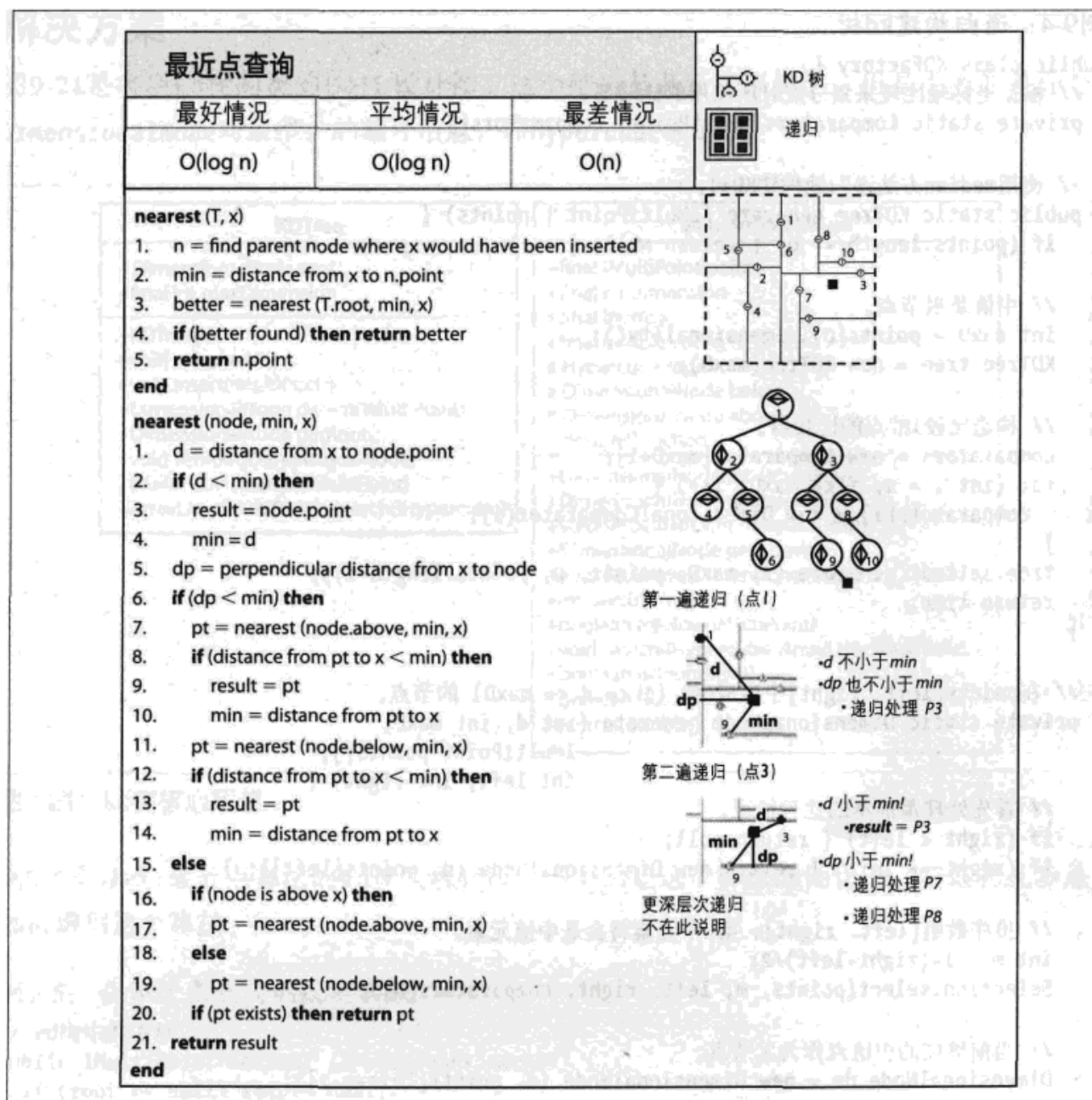


图9-20：最近点查询详解

驱动因素

二叉查找树仍然是一个高效的查找结构，它也能够节点插入和删除时做出平衡调整。但不幸的是，kd树的平衡调整就不这么容易了，因为有些深层次的结构信息和所表示的平面区域相关。最理想的解决方案是构建一棵kd树，使得a叶子节点都是同一层，或者b所有的叶子节点在其他叶子节点一级。例9-4使用递归来迭代维度的每个坐标。它从点集中选择中位元素，然后将在这个元素“下方”的元素插入左子树，“上方”的元素插入右子树。这段代码适用于任何维度。

例9-4：递归构建kd树

```
public class KDFactory {
    // 根据坐标轴比较来划分点时所使用的比较器。
    private static Comparator<IMultiPoint> comparators[];

    // 使用median方法递归地构建KD树。
    public static KDTree generate (IMultiPoint []points) {
        if (points.length == 0) { return null; }

        // 中值是根节点。
        int maxD = points[0].dimensionality();
        KDTree tree = new KDTree(maxD);

        // 构造比较i维点的比较器。
        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;
    }

    // 在points[left, right]中生成d维 (1 <= d <= maxD) 的节点。
    private static DimensionalNode generate (int d, int maxD,
                                             IMultiPoint points[],
                                             int left, int right) {

        // 首先处理最简单的这种情况。
        if (right < left) { return null; }
        if (right == left) { return new DimensionalNode (d, points[left]); }

        // 排序数组[left, right], 第m个元素将会是中值元素。
        int m = 1+(right-left)/2;
        Selection.select(points, m, left, right, comparators[d]);

        // 当前维度的中值点作为父节点。
        DimensionalNode dm = new DimensionalNode (d, points[left+m-1]);

        // 进行下一维度计算, 或者返回1维。
        if (++d > maxD) { d = 1; }

        // 递归地计算左右子树, 也就是所谓的在切分线“上方”或者“下方”。
        dm.setBelow(maxD, generate (d, maxD, points, left, left+m-2));
        dm.setAbove(maxD, generate (d, maxD, points, left+m, right));
        return dm;
    }
}
```

选择操作在第4章的“快速排序”一节中已经描述过了。能够在平均情况下在 $O(n)$ 时间内选择第 k 小的元素, 但是, 在最坏情况下会退化到 $O(n^2)$ 的性能。为了避免这种情况, 我们使用BFPRT选择算法, 在第4章中我们讨论过这个算法, 这个算法能够保证, 虽然在平均情况下性能不如标准选择算法, 在最坏情况下的性能仍然为 $O(n)$ 。

解决方案

图9-21是实现kd树的类的UML设计图。这个结构是基于二叉树的，但是主要的差异是DimensionalNode对象维护的额外信息，即Hypercube区域。

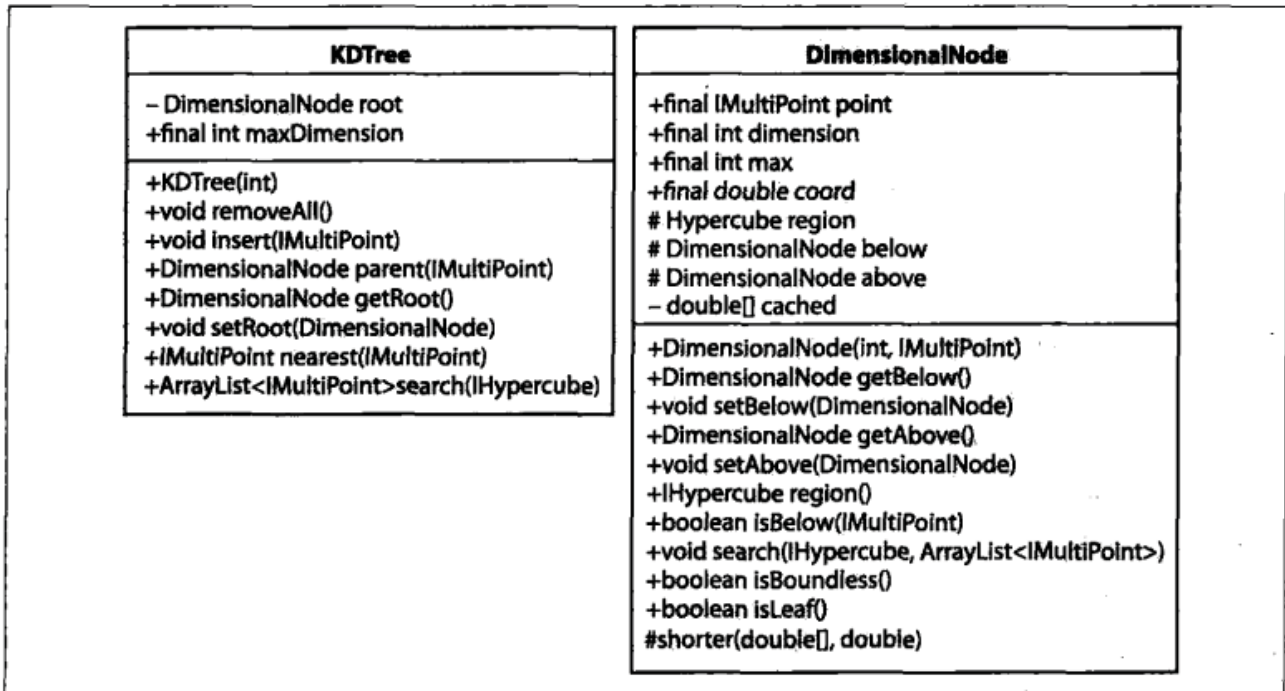


图9-21: kd树核心思想

例9-5是寻找x最近点算法的kd树实现代码。图9-20是这个算法的伪代码，其最初几步是如何调用这个算法。

例9-5: 最邻近点查询的kd树实现

// kd树中的方法。

```
public IMultiPoint nearest (IMultiPoint target) {  
    if (root == null) return null;
```

// 寻找目标节点的父节点。我们得到现阶段的最短距离。

```
DimensionalNode parent = parent(target);  
IMultiPoint result = parent.point;  
double smallest = target.distance(result);
```

// 从根节点出发，检查所有可能得到的最短距离。如果找到一个，那么返回。

```
double best[] = new double[] { smallest };  
double raw[] = target.raw();  
IMultiPoint betterOne = root.nearest (raw, best);  
if (betterOne != null) { return betterOne; }  
return result;  
}
```

// DimensionalNode中的方法。min[0]是计算出来的最短距离。

```
IMultiPoint nearest (double[] rawTarget, double min[]) {
```

```

// 如果得到更短的距离, 更新min。
IMultiPoint result = null;

// 如果更短, 那么更新min。
double d = shorter(rawTarget, min[0]);
if (d >= 0 && d < min[0]) {
    min[0] = d;
    result = point;
}

// 决定是否我们需要递归进入子树。如果d小于现在的最短距离,
// 那么我们就需要处理两棵子树。
double dp = Math.abs(coord - rawTarget[dimension-1]);
IMultiPoint newResult = null;

if (dp < min[0]) {
    // 进入两棵子树处理, 返回最短距离。
    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 {
    // 仅仅需要进入其中一棵子树。
    if (rawTarget[dimension-1] < coord) {
        if (below != null) {
            newResult = below.nearest (rawTarget, min);
        }
    } else {
        if (above != null) {
            newResult = above.nearest (rawTarget, min);
        }
    }
}

// 如果找到一个较小的结果, 返回该结果。
if (newResult != null) { return newResult; }
}
return result;
}

```

理解最邻近点算法的关键在于我们要能够快速定位点 x 所在的区域, 因为这个区域内很可能存在最近的点。之后我们从根节点到这个区域, 递归地检查是否存在其他的更近的点, 因为kd树的矩形区域划分是根据输入集的变化而变化。在一棵不平衡的kd树中, 这个检查过程的开销可能会达到 $O(n)$, 因此我们需要更加合理地处理输入。

我们有两个地方可以改进样例解的性能。首先, 我们使用原始的double数组来表示点, 然后在这之上进行比较。其次, DimensionalNode有一个方法, 这个方法返回两个 d 维点

的距离是否小于当前得到的最小距离，这个方法（在本书的代码库中）将会在距离超过已知最小距离时立即退出。

结论

假设初始的kd树是平衡的，那么查找能够在递归调用时抛弃几乎一半的点。在某些时候可能需要调用两个查找，但是这种情况只会发生在我们需要在切分线两边的区域来寻找最近点时。

分析

如果kd树初始是平衡的，那么每一层的切分线正好穿过此层的中点。我们能够在 $O(\log n)$ 的时间内找到目标点区域。但是，这个算法有可能会调用两个查找：一个是查找左子树，一个是查找右子树。如果这种情况频繁地出现，算法的性能会退化到 $O(n)$ ，所以值得我们分析这种情况发生的频率。当从目标点到节点的垂直距离 dp 小于已知最小距离时，我们可能需要执行多个调用。随着维度的增长，可能会有更多的点满足这个条件。表9-6是这种情况发生几率的经验性总结。我们在单位正方形中生成 n 个随机二维点， n 从4~131 072，在这些点上构建一棵平衡的kd树。我们会有50个查询，表9-6记录了两个递归调用和单个递归调用发生的平均次数。

表9-6：两个递归调用和单个递归调用的比

n	d=2#递归	d=2#两个递归	d=10#递归	d=10#两个递归
4	1.54	0.54	1.02	1
8	2.8	1.08	1.04	3
16	4.3	1.36	1.48	6.84
32	5.66	2.14	1.86	14.58
64	8.08	2.58	3.54	30.42
128	9.24	2.58	8.64	60.06
256	10.36	2.42	25.42	109.9
512	11.76	2.8	52.44	222.44
1,024	13.2	3.06	122.32	421.68
2,048	15.48	3.22	244.54	730.84
4,096	15.96	2.72	466.1	1183.1
8,192	17.18	3.3	925.22	1876.66
16,384	19.9	3.38	1552.98	2939.08
32,768	18.78	3.14	2769.72	5118.76

表9-6：两个递归调用和单个递归调用的比（续）

n	d=2#递归	d=2#两个递归	d=10#递归	d=10#两个递归
65,536	20.88	3.16	3272.24	4788.3
131,072	23.32	3.98	5376.06	7703.72

我们可以看到，在随机生成的二维数据上，产生两个调用的次数是 $0.3 \cdot \log(n)$ ，但是在十维上，这个数目会升到 $342 \cdot \log(n)$ （1000倍左右的提升）。我们可以观察到，这个估计值大约是 $O(\log n)$ 。但是当 d 增长到“足够接近” n 时，会发生什么呢？图9-22的结果告诉我们，随着 d 的增长，两个调用发生的次数逐渐逼近 $n/2$ 。事实上， d 增长时，单次调用的次数遵从正态分布，其中值趋近于 $\log(n)$ ，两种递归调用最终恰好是各占一半。这个结论对与性能的影响就是使得查询在 d 增长的情况下，性能逼近 $\log(n)$ 。在这种极端情况下，使用kd树，性能不会好于 $O(n)$ 。

算法在某些特定的输入数据上性能退化非常严重。例如，我们将表9-6的输入点改成一个半径大于1的圆内，但是查询的点还是在单位正方形中。当 $n=131\,072$ 时，单个递归调用的数量跳到235.8个，而两次递归调用的数量会增加到932.78个。因此最邻近点算法的性能会退化到最坏的 $O(n)$ 。

我们也可以对比一下基于kd树的最邻近点算法和穷举算法的性能。我们有这样一个例子，4096个点和128个查询，那么维度 d 从多大开始，kd树的性能会差于最原始的穷举算法呢？我们执行了100次实验，抛弃最好和最坏的结果，取剩余98次的平均值。结果如图9-23所示，维度从10开始，原始的穷举算法的性能就开始占优。当 n 增加到131 072个，原始穷举算法的性能在 $d=12$ 时开始占优，所以拐点出现的位置和 n 、 d 以及输入点的分布情况相关。我们不会分析在这个拐点上，构建kd树的开销是多少，因为这个开销都被分摊到了每次查询上。

图9-23的结果证实了随着维度的增加，使用kd树获得的性能改进越来越少。等式中主要的驱动因素不是构建kd树的开销，而是将要插入kd树中的点的个数。在大规模数据集上，节省的开销是显而易见的。另外一个影响性能的因素是维度 d ，计算两个 d 维点的欧几里德距离是 $O(d)$ 的操作，随着 d 增加，每次计算需要更多的时间。

变种

我们描述的实现是从根节点向下遍历，有一种实现可以从底向上遍历，最后访问根节点（注9）。

注9： 参见<http://www.codeproject.com/KB/architecture/KDTree.aspx>。

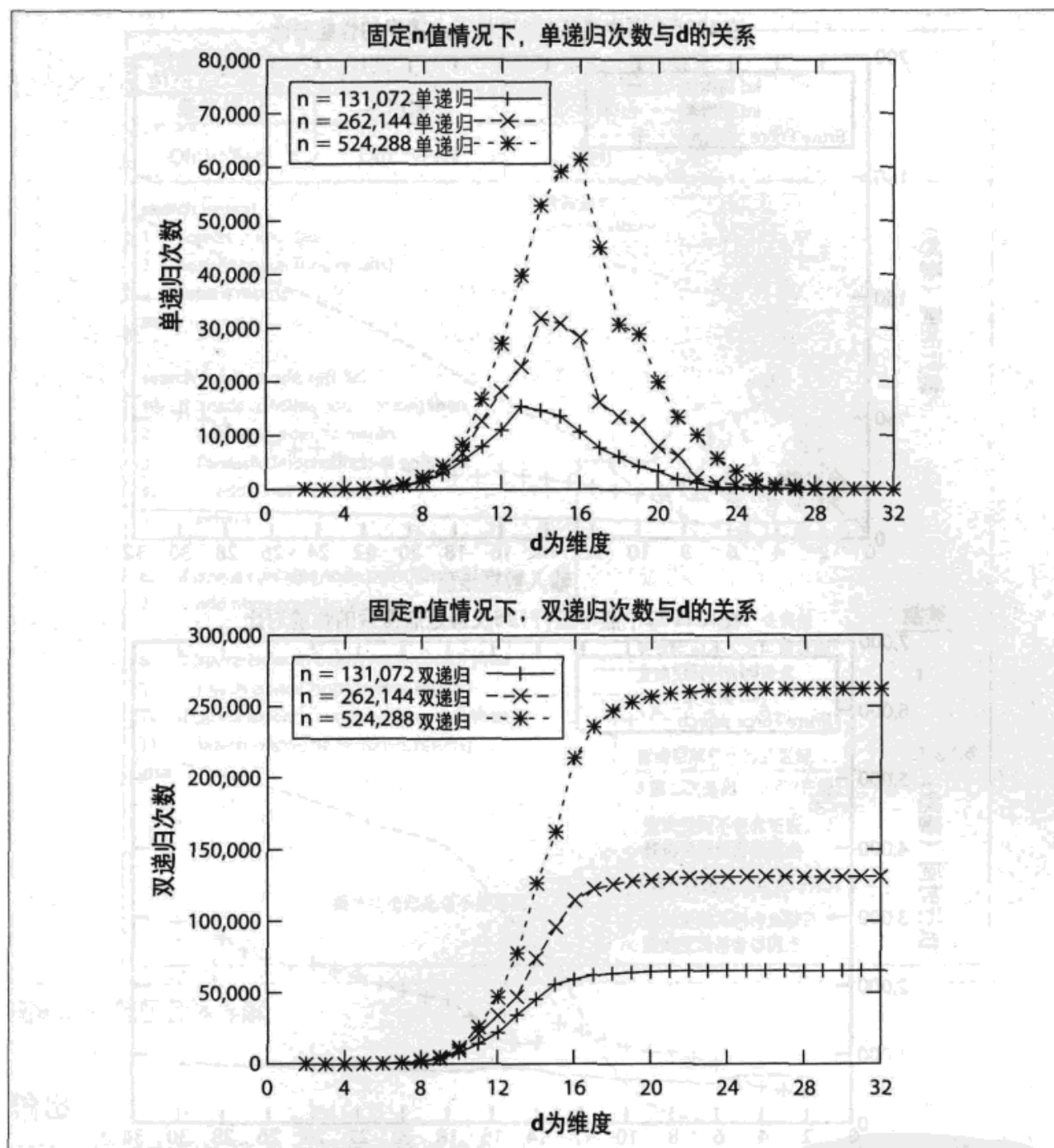


图9-22：随着n和d增长，两个递归调用的次数

范围查询

有一个矩形区域 $R[x_{\text{low}}, y_{\text{low}}, x_{\text{high}}, y_{\text{high}}]$ ，以及一个点集 P ，请问 P 中哪些点是在矩形区域 R 中的呢？穷举算法会检查所有的点，花费的时间为 $O(n)$ ，我们能够做得更好么？使用kd树，我们将会阐述如何高效地解决在笛卡儿平面上的范围查询。

$$O(\sqrt{n} + r)$$

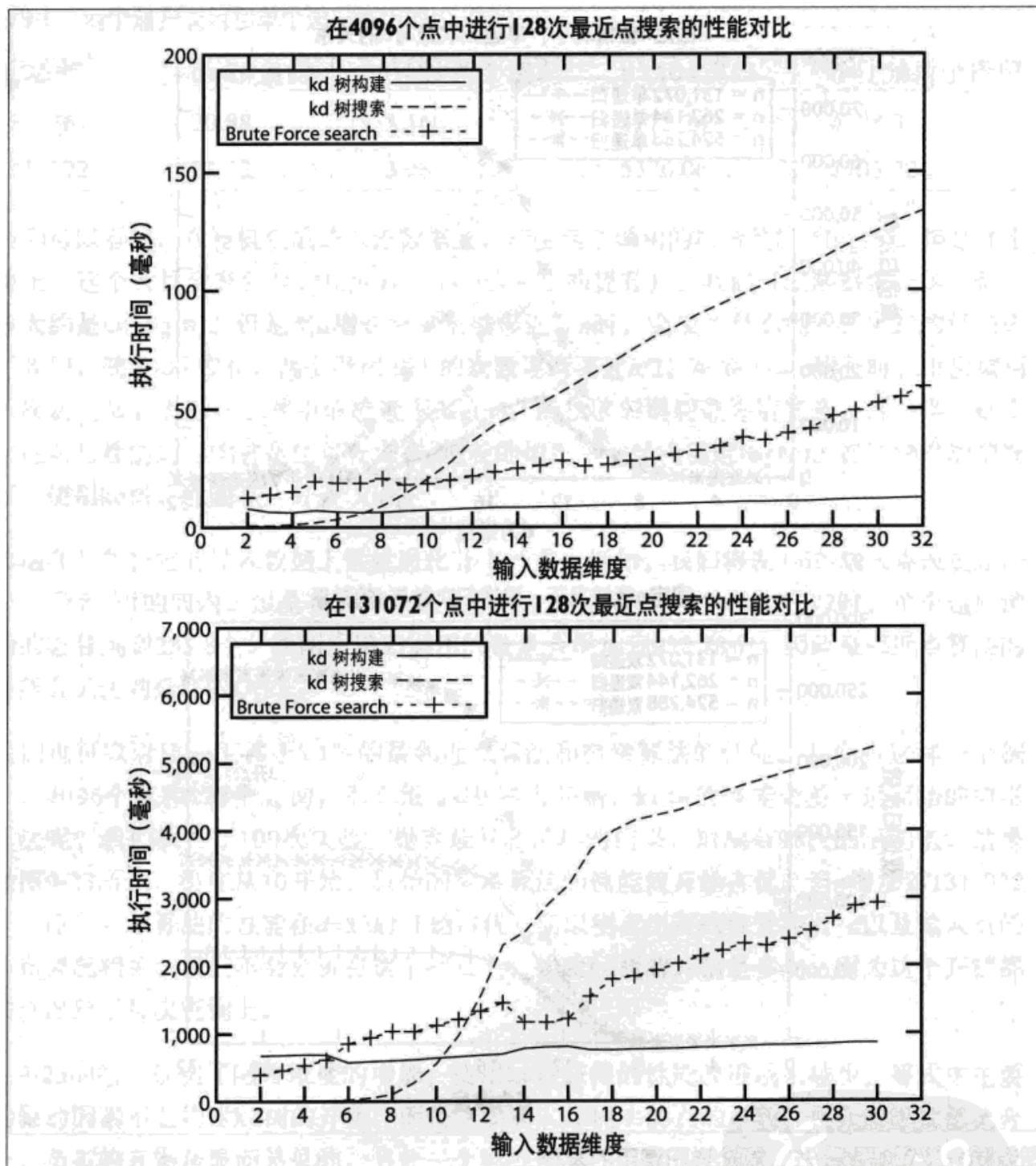


图9-23: 比较kd树和穷举实现

r 是在 R 中的点的个数。当然，当点是 d 维时，这个问题就成了 d 维范围查询，能够在 $O(n^{1-1/d}+r)$ 的时间内得到解，如图9-24所示。

输入/输出

输入

一个 d 维的点集 P 和一个 d 维的正方体。

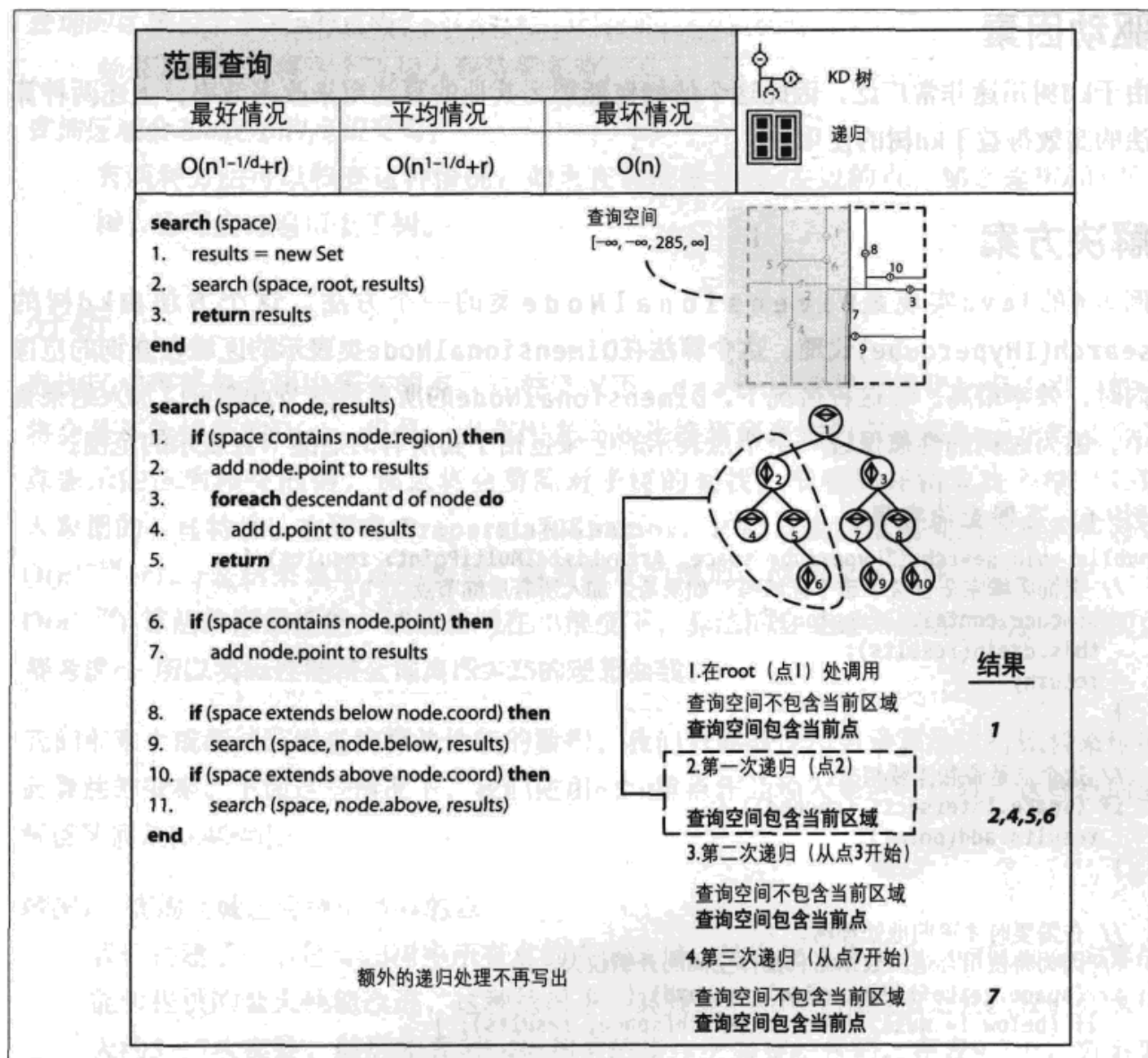


图9-24：范围查询详解

输出

在这个区域内的所有点，不要求按顺序输出。

假设

查询的范围维度是数据的维度保持一致。

使用环境

kd树的性能在高维度下非常差，算法在高维度下，性能会退化。

驱动因素

由于kd树用途非常广泛，因此这个结构也能够为其他的算法用来改进效率。上述两种算法的高效得益于kd树的使用。

解决方案

例9-6的Java实现是DimensionalNode类的一个方法，这个方法由kd树的search(IHypercube)代理。这个算法在DimensionalNode类表示的区域在查询的范围内时，效率最高。在这种情况下，DimensionalNode的所有后继节点都可以加入结果集中，因为kd树的性质保证一个节点表示的区域包括了其所有的后继节点表示的范围。

例9-6：范围查询实现

```
public void search (IHypercube space, ArrayList<IMultiPoint> results) {
    // 查询区域完全包含了这个区域吗？如果是，加入所有后继节点。
    if (space.contains (region)) {
        this.drain(results);
        return;
    }

    // 这个点是否和区域相交。
    if (space.intersects (cached)) {
        results.add(point);
    }

    // 在需要时才递归地处理树。
    // 因为将使用合适的数据结构操作空间的开销较大。
    if (space.getLeft(dimension) < coord) {
        if (below != null) { below.search(space, results); }
    }
    if (coord < space.getRight(dimension)) {
        if (above != null) { above.search(space, results); }
    }
}

/** 访问给定节点的所有后继节点 */
private void drain(ArrayList<IMultiPoint> results) {
    if (below != null) { below.drain (results); }
    results.add(this.point);
    if (above != null) { above.drain (results); }
}
```

例9-6的代码是一个修改过的遍历树实现。kd树以一种层次的方式来划分 d 维点，范围查询在每个节点 n 上将会做三个决定：

这个节点 n 相关的区域是否已经完全包含了查询区域？

当这种情况发生时，查找将会停止，因为所有的后继节点都属于结果集。drain方法会遍历子树的所有节点，并且将这些点加入结果集中。

查询的区域包含节点 n 的点吗？

如果是的话，将这个点加入到结果集中。

查询区域会和 n 表示的点相交吗？

有两种方法可以检查这种情况：如果查询范围寻找 n 左边的点，那么遍历 n 的左子树，否则应该遍历右子树。

分析

查询区域可能包含树中所有的点，这种情况下，`drain`方法将会遍历所有的点，性能将会是不能接受的 $O(n)$ 。但是，当范围查询算法检测到查询的范围不和kd树的某个节点表示的范围相交的话，那么将会剪除对子树的查找。节省的开销取决于维度和输入数据的一些特性。有研究(Preparata和Shamos, 1985)表明范围查询算法的性能是 $O(n^{1-1/d}+r)$ ， r 是结果集中点的数目。随着维度的增加，性能逐渐下降。图9-25是一个 $O(n^{1-1/d})$ 算法的期望性能，该图表明在小维度下，算法的性能逐渐逼近 $O(n)$ 。因为还需要考虑 r ，所以实际性能将会偏离图9-25的理想曲线。

我们很难生成测试范围查询算法性能的数据。我们只能通过和穷举算法进行比较来证明此算法的效率。下面这些情况下，我们使用 n 个 d 维点作为输入来进行比较。这些点的坐标值区间是 $[0,4096]$ 。

情况1：查询区域包含树中所有的点

我们构建了一个包含kd树中所有点的查询区域。这个例子告诉我们的区间查询算法能够提供的最大性能改进，这种情况下，其性能和kd树的维度 d 无关。我们会执行大约5~7次实验，然后能看到kd树相关的操作所需要的开销。在表9-7中，穷举算法的开销随着维度 d 的增加而增大，因为计算 d 维点之间距离的操作是 $O(d)$ 而不是常数。穷举算法在高维度上性能轻易地超过了基于kd树的最邻近点算法。

表9-7：第一种情况下，范围查询算法和穷举算法的比较（单位：毫秒）

n	$d=2$ RQ	$d=3$ RQ	$d=4$ RQ	$d=5$ RQ	$d=2$ BF	$d=3$ BF	$d=4$ BF	$d=5$ BF
4,096	51.4	73.9	94.3	124.5	10.5	13.0	12.7	13.6
8,192	199.6	204.3	215.6	228.8	17.8	20.8	25.4	26.0
16,384	354.3	375.1	401.7	422.9	33.7	44.4	55.7	66.1
32,768	678.5	765.8	780.7	827.0	90.8	116.3	129.9	145.3
65,536	1397.3	1482.2	1612.6	1817.8	189.7	226.6	266.4	315.0
131,072	2924.5	3146.4	3305.6	3738.9	378.3	458.9	534.5	638.9

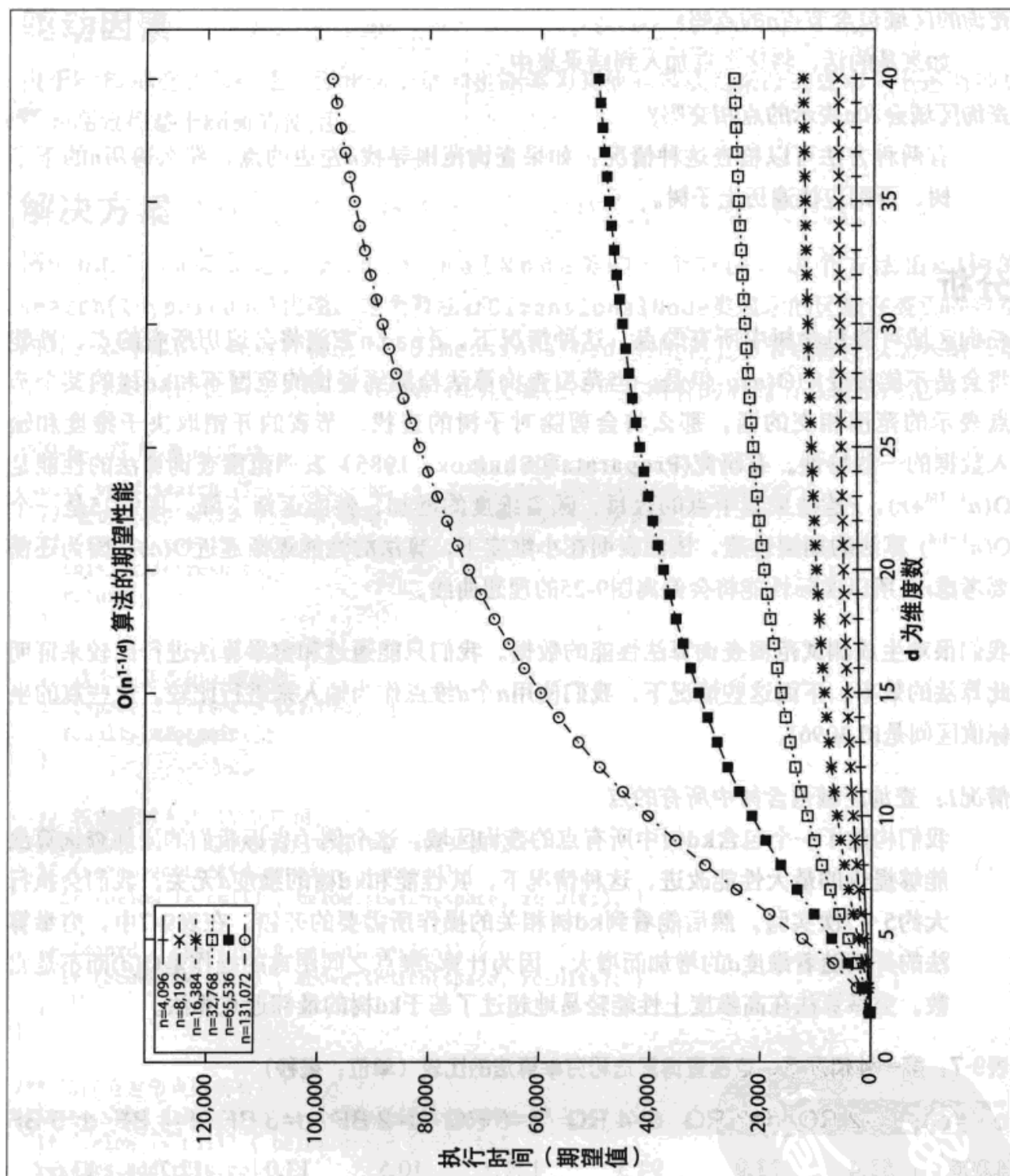


图9-25: $O(n^{1-1/d})$ 算法的期望性能

情况2: 微小区域

因为结果集中的点数 r 能较大程度地影响算法的性能,我们构建一系列的场景,使得在维度增加的情况下, r 并不改变。由于输入集合的一致性,我们不可能简单地构造一个查询区域。如果我们这样做,那么选择的输入集合总数将会是 $(1/2)^d$,也就是说,维度 d 的增加会导致 r 的增加。实际上,我们会构建一些查询区域,这些区

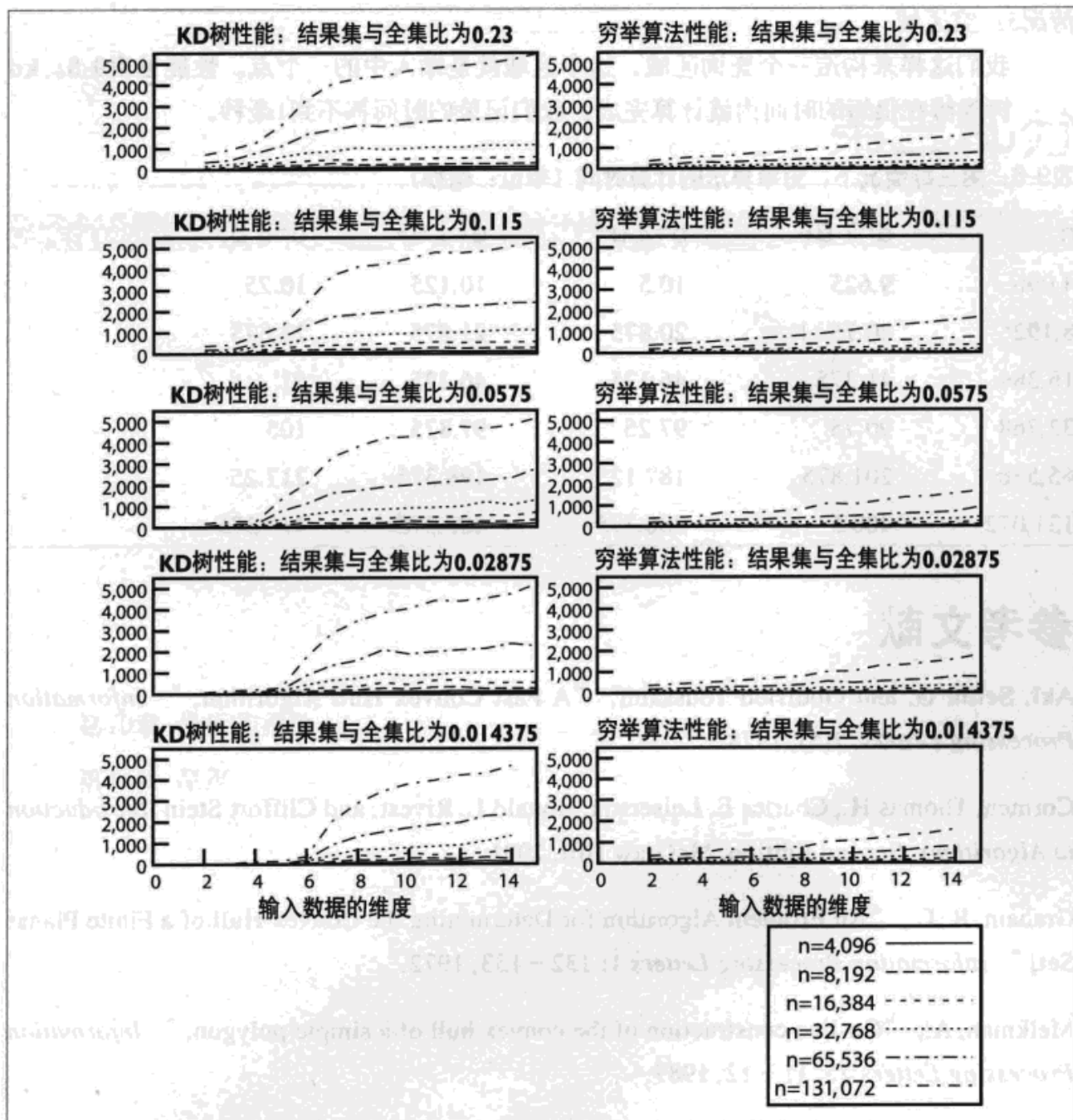


图9-26：第二种情况下，范围查询算法和穷举算法的比较

域的规模随着维度的增加而增加。例如，二维的查询区域会返回 $0.23 \cdot n$ 个点。但是，在三维上，查询区域应该扩展更大的范围。这样的话，我们就能预先限定因子 k ，结果集包含 $k \cdot n$ 个点。我们也会比较kd树的实现和穷举方法的实现， n 的范围从4096~131 072， d 从2~15，结果如图9-26所示。左边的表是kd树实现的典型行为，右边是穷举算法的线性行为。 $k=0.23$ 时，kd树的实现只是在 $d=2$ 和 $n \leq 8192$ 时表现较好。

情况3: 空区域

我们这样来构造一个查询区域, 这个区域就是输入中的一个点。性能见表9-8。kd树能够在很短的时间内就计算完成。我们记录的时间都不到1毫秒。

表9-8: 第三种情况下, 穷举算法的计算时间(单位: 毫秒)

n	d=2 BF	d=3 BF	d=4 BF	d=5 BF
4,096	9.625	10.5	10.125	10.25
8,192	20.75	20.875	21.875	23.875
16,384	41.375	46.125	46.375	51
32,768	90.75	97.25	97.875	105
65,536	201.875	187.125	198.375	217.25
131,072	400.5	386.375	400.375	411.375

参考文献

Akl, Selim G. and Godfried Toussaint, "A Fast Convex Hull Algorithm," *Information Processing Letters*, 7(5), 1978.

Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to Algorithms*, Second Edition. McGraw Hill, 2001.

Graham, R. L., "An Efficient Algorithm for Determining the Convex Hull of a Finite Planar Set," *Information Processing Letters* 1: 132 – 133, 1972.

Melkman, A., "On-line construction of the convex hull of a simple polygon," *Information Processing Letters* 25: 11 – 12, 1987.

Overmars, Mark and Jan van Leeuwen, "Maintenance of Configurations in the Plane," *Journal of Computer and System Sciences*, 23(2): 166 – 204, 1981.

Palazzi, Larry and Jack Snoeyink, "Counting and Reporting Red/Blue Segment Intersections," *CVGIP: Graphical Models and Image Processing*, 56(4): 304 – 310, 1994.

Preparata, Franco and Michael Shamos, *Computational Geometry: An Introduction*. Springer-Verlag, 1985.

第三部分

第10章 最后的招数

第11章 尾声





本章不同于前面的章节，前面所讲述的都是用于解决常见问题的算法，而这一章的问题有所不同，需要一些特别的算法。这些算法的知识能够帮助设计者去选择如何使用它们去解决那些很不寻常的问题。

另一个不同在于，在前面的内容中，随机和概率是用于分析算法的一般行为，而这里随机则是算法本质的部分。实际上，这里我们将要介绍的概率算法是确定性算法一个很有趣的替代方案。同样的输入同样的算法，运行两次的结果可能是不一样的，有时候我们会容忍错误的发生，也会接受算法最终告知问题无法解决。

一个重要假设是，算法可以利用随机位序列。其实随机性很难定义，虽然我们已经有几个用于检测随机性的测试，不过实际上很难生成满足这些测试的比特串。

另类算法

本书前面所提到的所有算法都是在一个顺序的确定性的计算平台上，为问题的单个实例计算出正确的结果。而现在已经出现的不少有趣研究却是摒弃以下四大设定的算法。

- 结果必须是正确的。
- 每次只解决一个实例。
- 顺序的。
- 确定性的。

抛开这些设定使我们能够创造出各种各样不同类型的算法。

近似算法

近似算法寻找的是接近，但并不一定要是完全正确的答案。总的来说就是一个权衡：牺牲结果的准确性，换取更短的运行时间。

当完全正确的结果并不必要，而较好的答案就可以接受时，就可以考虑以准确性为代价来提高解决问题的速度，旅行商问题（TSP）就是这样一个典例。旅行商问题是指，给定一个需要走访的城市集合以及每对城市之间的距离，如何找到一条最短的旅游路线：从某个城市开始，走遍每一个城市，且每个只经过一次，最后回到这个城市。这是计算机科学所有问题中研究最深入的问题之一，不太可能存在一个多项式复杂度的算法能够解决TSP问题。也就是说，没有算法能够以 $O(n^K)$ 的复杂度解决，其中 K 是固定的某个整数。它属于 $NP-hard$ 问题，人们坚信这类问题天生就很难找到完全正确的答案。

但是，假设各城市间的距离满足三角形不等式（对于 a 、 b 、 c 三个点来说， a 到 b 之间的距离不会大于 a 到 c 的距离加上 c 到 b 的距离）。Christofides（1976）设计了一种有效的解法能够找到一条较短路径，而这条路径不会长过最短路径50%。

离线算法

与在线算法的通常假设不同，我们可能会一次提交众多问题实例批量解决，而不要求每个实例提交后就马上返回结果。

假设我们要实现一个字典，先在这个空字典中插入一个含有 n 个数字 $y_1 \cdots y_n$ 的集合，然后进行 $n/2$ 次的查询 $\text{contains}(x_i): x_1 \cdots x_{n/2}$ 。最优的数据结构是将每个 y_i 插入到一个无序的数组 Y 中，复杂度为 $O(n)$ ，然后通过数组 Y 中顺序查找实现 $\text{contains}(x_i)$ ，这样的最差情况的复杂度为 $O(n)$ 。所以这 $n+1$ 次操作的总体最大复杂度为 $O(n)$ 。

执行 $n/2$ 次顺序查找的复杂度为 $O(n^2)$ 。由于无法预测下一个查询是什么，所以在线算法无法主动的为下一个特定的查询进行优化。不过，如果我们在线下批量进行这 $n/2$ 个查询，就可以分别将 $y_1 \cdots y_n$ 以及 $x_1 \cdots x_{n/2}$ 进行排序，存在有序数组 Y 和 X 中，这两个操作最坏情况的复杂度为 $O(n \log n)$ ，然后扫描这两个有序数组寻找重复元素，最坏情况复杂度为 $O(n)$ 。批量进行 $n/2$ 个查询的这种离线算法的最坏情况复杂度为 $O(n \log n)$ ，而如果执意在线算法，要求每个查询在下一个查询之前返回结果的话，最坏情况的复杂度则为 $O(n^2)$ 。

并行算法

一个计算过程可能会产生多个其他的计算过程用于同时处理一个问题的多个子实例。还拿上一小节离线算法的例子来说，如果在 n 个处理器上并行查找就有可能提高这 $n/2$ 个顺

序查找的性能，这样最坏情况的复杂度为 $O(n)$ 。想要更深入了解并行算法，建议去读一下Berman和Paul（2004）年关于这个方面的著作。另外了解一些利用多核处理器并行机制的实际系统对此也很有裨益，具体参见Armstrong的《Erlang编程：实现并发的利器》（2007）。

随机算法

某些算法可以通过随机位序列（或者随机数）来解决问题。当假定随机访问时，我们通常能够找到解决问题的快速算法。实际上，大家应该意识到，在确定性计算当中，真正随机的序列是很难生成的。尽管我们可以生成伪随机序列，来替代真正的随机字节序列，然而生成这些随机序列的成本却是不可忽视的。

估算集合的大小

下面举个例子来说明如何利用概率算法获得计算速度的提升，假设我们要估算一个包含 n 个元素的集合： $\{x_1, \dots, x_n\}$ 的大小，也就是 n 的值。最直接的方法就是对所有元素进行计数。但是如果不太准确的结果也可以接受，而同时希望计算速度更快一些的话，那么下面例10-1中描述的算法会是一个更好的选择。

例10-1：概率计数算法的实现

```
public static double computeK (int n) {
    // Make sure we use data structure with efficient lookup.
    Hashtable<Integer, Boolean> setS = new Hashtable<Integer, Boolean>();

    // Repeatedly probe to see if already located
    int y = 1+((int)(Math.random()*n));
    while (!setS.containsKey(y)) {
        setS.put(y, Boolean.TRUE);
        y = 1+((int)(Math.random()*n));
    }

    // return estimate of original size
    int k = setS.size();
    return 2.0*k*k/Math.PI;
}
```

算法预期的执行时间为：

$$O(\sqrt{n})$$

也就是说，while循环预期的执行次数为：

$$\sqrt{\frac{\pi n}{2}}$$

这个算法与生物学家用来估算有限空间内生物群体规模的标识再捕法很相似。显然，它不可能得到 n 的确切数值，因为 $2*k^2/\pi$ 不可能是个整数。但是 $2*k^2/\pi$ 是 n 的无偏估计，它的期望值等于 n 。

表10-1给出了该算法在大量独立实验组中得到的结果。概率算法产生了 n 的估算值的 t 次实验结果（ t 分别为32、64、128、256）。在这些实验中，最大和最小的估算结果都已去掉，每列所示的是剩下 $t-2$ 次实验结果的平均值。最后三行通过计算估算值/实际值的最小比率 a 、最大比率 b 以及两者之间的差值给出了“估算平均值”的准确率。例如，32次实验中，估算值353 998相对其目标值524 288的比率最低（0.68），而1 527 380相对1 048 576的比率则最高（1.46）。

表10-1：概率统计算法的实验运行结果

n	32次（实验的平均值）	64次	128次	256次
256	314	210	296	362
512	511	684	643	664
1,024	941	905	1,150	1,314
2,048	2,611	3,038	2,405	2,532
4,096	3,779	6,068	4,812	5,378
8,192	7,858	10,656	8,435	10,860
16,384	22,786	21,617	19,169	19,809
32,768	33,509	40,549	36,395	38,863
65,536	85,421	77,335	80,119	93,807
131,072	131,728	172,175	148,549	160,750
262,144	270,187	421,345	375,442	299,551
524,288	353,998	463,923	736,396	642,986
1,048,576	1,527,380	1,417,047	1,299,312	1,334,487
2,097,152	2,291,903	2,106,072	2,615,379	2,445,086
4,194,304	5,348,730	4,565,833	5,653,524	5,132,245
8,388,608	8,017,734	9,791,002	12,220,879	10,064,671
16,777,216	23,006,070	28,363,383	20,316,904	19,470,289
Accuracy	Low: 0.68	Low: 0.82	Low: 1.03	Low: 1.14
	High: 1.46	High: 1.69	High: 1.46	High: 1.43
	Range: 0.78	Range: 0.87	Range: 0.43	Range: 0.29

由于这次实验的随机性，所以不能完全保证可以简单的通过增加独立随机实验的次数来

达到最终正确的结果。事实上，你可能需要进行非常大数量的实验才能得到想要的估算值。与其试图通过利用随机算法来获得一个确切的值，还不如去投入精力去寻找一些能够得到正确结果的算法。

估算搜索树的大小

如果在同一个棋盘上的两个皇后在同一行、同一列或者同一对角线上，则它们可以互相攻击。接下来我们要将一个集合的皇后放置在棋盘上，并且让其中任意两个都不会互相攻击。很明显，我们不能把 $n+1$ 个皇后放在 $n \times n$ 的棋盘上，因为两个皇后不能在同一行。我们总能放置 n 个不会互相攻击的皇后吗？这个问题就是有名的 n 皇后问题（ n -Queens Problem）。我们来一般化这个问题，计算在一个 $n \times n$ 的棋盘上有多少种方式可以放置不会互相攻击的 n 个皇后。我们将要介绍的随机方法的应用不仅于此，除了我们这里谈到的游戏之外，还有很多应用，它能够用来估计搜索树的形状。

目前没有已知的技术可以高效地求解 n 皇后问题。表10-2所包含的数据是早些时候Sloane的整数序列在线百科全书所计算的（注1）。

表10-2：n皇后问题已知求解与估算求解

n	实际解	T=1,024时的估算解	T=8,192时的估算解	T=65,536时的估算解
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	2,680	2,473	2,499	2,600
12	14,200	12,606	14,656	13,905
13	73,712	68,580	62,140	71,678
14	365,596	266,618	391,392	372,699

注1：<http://www.research.att.com/~njas/sequences/A000170>.

表10-2: n皇后问题已知求解与估算求解 (续)

n	实际解	T=1,024时的估算解	T=8,192时的估算解	T=65,536时的估算解
15	2,279,184	1,786,570	2,168,273	2,289,607
16	14,772,512	12,600,153	13,210,175	15,020,881
17	95,815,104	79,531,007	75,677,252	101,664,299
18	666,090,624	713,470,160	582,980,339	623,574,560
19	4,968,057,848	4,931,587,745	4,642,673,268	4,931,598,683
20	39,029,188,884	17,864,106,169	38,470,127,712	37,861,260,851

为了求出4皇后问题的精确解，我们基于每行只能有一个皇后的事实，生成了一棵搜索树。起始状态是没有放置任何皇后，图10-1展示了如何通过在第一行上依次摆下4个皇后来对搜索树进行直接扩展。

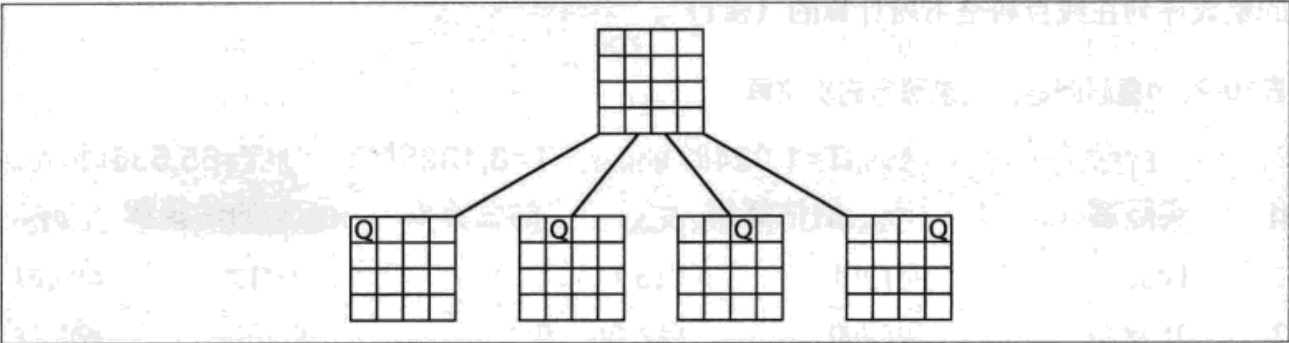


图10-1: 4皇后问题的初始搜索树

对每种情况再进行扩展，在第二行摆上不会互相攻击的另一个皇后，如图10-2所示。

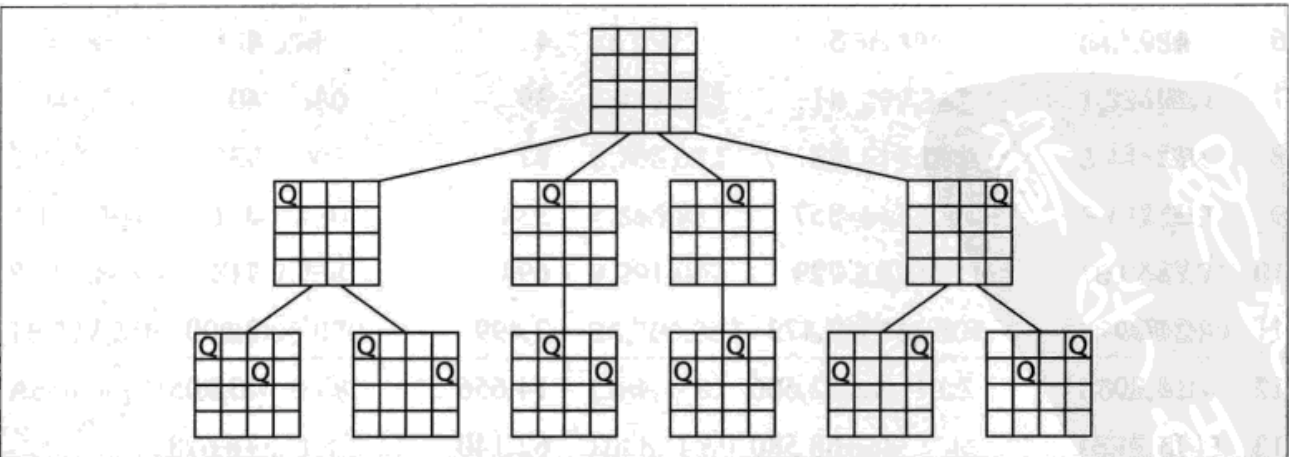


图10-2: 放置了第二个皇后的4皇后搜索树

第一个和最后一个中间解不能再放下第三个皇后了。中间的两个可以放置第三个，而最中间的两个则可以放置第四个皇后（如图10-3所示）。

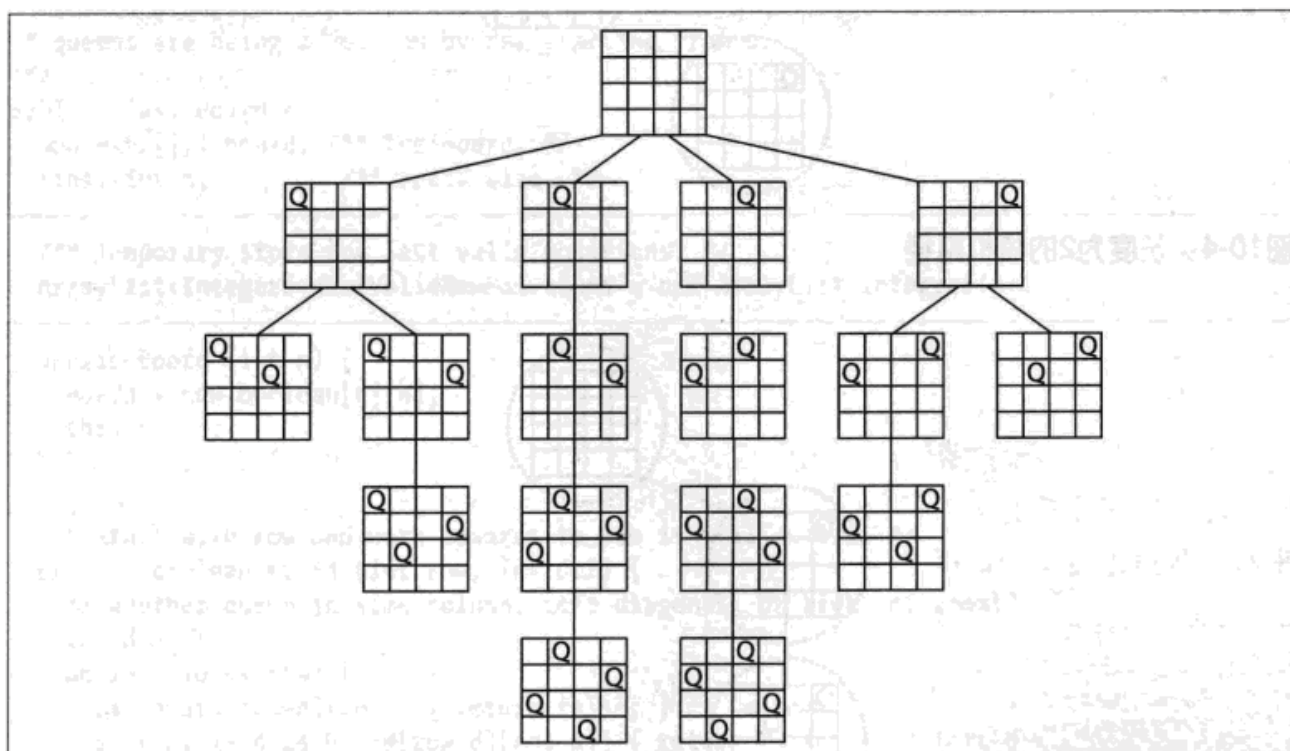


图10-3：4皇后问题的最终解

如此详尽地阐述搜索树让我们看到，4皇后有2个解。通过计算来解19皇后就难得多了。在第19层搜索树上，有4 968 057 848个节点，而整个树的节点更多，需要很长很长时间才能算出结果。

Donald Knuth (1975)提出了另一种全新的方法来估算搜索树的形状和大小。它的方法会随机地向下遍历搜索树。为了简洁起见，我们将会用4皇后来说明这个方法，明显地，它可以轻易地应用于估算19皇后问题的解的数量。从搜索树的根开始（没放置任何皇后），我们估计出在第0层有1个节点。接下来，我们会在所有节点上去确定其儿子节点的数量（由该节点所对应的中间解的直接扩展的数量），并且随机地选择它们。我们看到根节点有四个孩子，因此我们（正确地）估计在第1层有4个节点。接下来，我们随机地选择四个中的一个节点，比如说第一个，如图10-4所示。

对于当前随机路径的更低层节点，我们要查看它有多少个子节点（在第二行有多少种方式放置一个皇后），再随机选择其中一个。注意到它有两个孩子，我们估计在下一层的节点数是第1层节点数的两倍。也就是说，我们估计在第2层有8个节点，接下来我们在两个节点中随机选择其中一个，如图10-5所示。

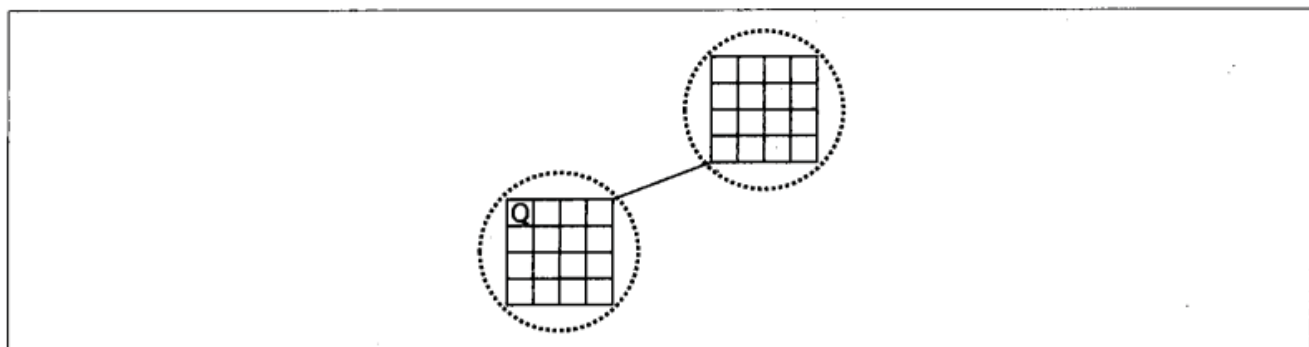


图10-4：长度为2的随机路径

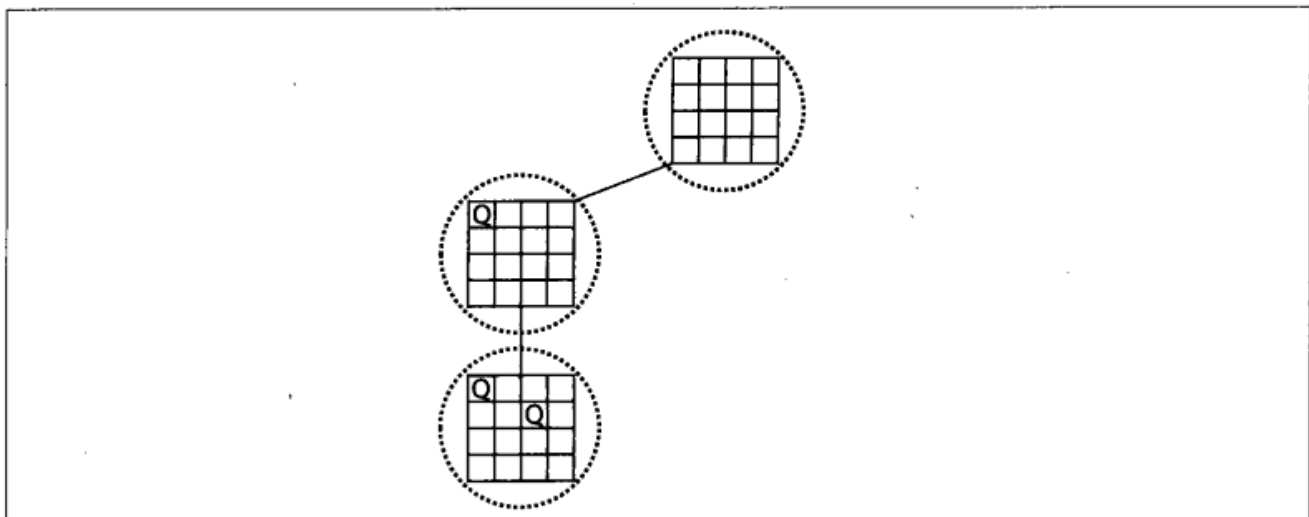


图10-5：长度为3的随机路径

现在，当前随机路径的最低层节点，我们来确定其子节点的数量：能够有多少种方法放置第3个皇后？答案是0个，我们估计第3层节点的数量是第2层的0倍。也就是说，我们估计第3层有 0×8 个节点，那么在第4层就有0个节点。这说明我们估计出4皇后问题无解。事实上有些随机遍历并不准确，但是如果我们多次随机遍历，并取这些估计的平均，就会得到越来越接近真实值的结果。因为每一次估算是可以快速完成的，因此最终的（平均）估计值也可以快速求解的。每次估计的目标值都是正确结果，然而估计的平均值却可能随着增加尝试的次数更逼近正确答案。表10-2列出了我们所实现的算法的计算结果，分别为1024次、8192次和65 536次尝试。未包含时间信息，因为所有的结果都是一分钟内完成。19皇后问题 $T=65\ 536$ 次尝试的最终估计值和真实结果相差3%。事实上，所有 $T=65\ 536$ 的估计与真实值的差异都在5.8%以内。这个算法的特性就是，尝试的次数越多，最终计算的结果与真实值就越准确。例10-2展示了单次计算 n 皇后估计的Java实现。用于生成表10-2的完整代码存于代码库中。

例10-2: Knuth的随机估计n皇后算法的实现

```
/**
 * For an n-by-n board, store up to n non-threatening queens and support
 * search along the lines of Knuth's random walk. It is assumed the
 * queens are being added row by row starting from 0.
 */
public class Board {
    boolean [][] board; /** The board. */
    final int n;         /** board size. */

    /** Temporary store for last valid positions. */
    ArrayList<Integer> nextValidRowPositions = new ArrayList<Integer>();

    public Board (int n) {
        board = new boolean[n][n];
        this.n = n;
    }

    /** Start with row and work upwards to see if still valid. */
    private boolean valid (int row, int col) {
        // another queen in same column, left diagonal, or right diagonal?
        int d = 0;
        while (++d <= row) {
            if (board[row-d][col]) { return false; } // column
            if (col >= d && board[row-d][col-d]) { return false; } // left-d
            if (col+d < n && board[row-d][col+d]) { return false; } // right-d
        }
        return true; // OK
    }

    /**
     * Find out how many valid children states are found by trying to add
     * a queen to the given row. Returns a number from 0 to 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;
    }

    /** If no board is available at this row then return false. */
    public boolean randomNextBoard(int r) {
        int sz = nextValidRowPositions.size();
        if (sz == 0) { return false; }
    }
}
```

```

        // select one randomly
        int c = ((int)(Math.random()*sz));
        board[r][nextValidRowPositions.get(c)] = true;
        return true;
    }
}

public class SingleQuery {

    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);

            // no more to go, so no solution found.
            if (!b.randomNextBoard(r)) {
                lastEstimate = 0;
                break;
            }

            // compute estimate based on ongoing tally and advance
            lastEstimate = lastEstimate*numChildren;
            r++;
        }

        return lastEstimate;
    }
}

```

结果可能出错却可以衰减错误率的算法

在这个小节中，我们要学习的是这样一种算法：它的结果可能是错的，但是会衰减错误率。通过适度的计算，它的错误率可以非常低。

检测数据库间的不一致

假设某个公司为了快速响应来自许多不同点的查询需求而为一个大型数据库创建了多个复本，数据库的查询请求远多于更新，更新操作会作用到每个复本上。另外，还会有其他的更新会修改数据库。确保所有复本一致的一个方法是将任意某个点的数据库复本发

送给所有其他点以检测一致性。不过这个数据库规模之大会让传送复本的代价无比之高。

另外一个方法就是将其中任意一份复本的特征值发送到其他点上，检测这些点上复本的特征值是否与此吻合。更明确的说法是，假设数据库是一个超长的比特串 $(b_0 \cdots b_{n-1})$ ，这个长串的特征值为 $(b_0 + b_1 \times 2^1 + b_2 \times 2^2 + \cdots + b_{n-1} \times 2^{n-1}) \bmod p$ ， p 是随机选择的一个素数，我们所要发送的只是一个长度为 $\log(p)$ 的比特串。如果发送的特征值与当前数据库的特征值不同，则可以肯定两个数据库之间不一致，不过特征值相同也不能确定两个数据库完全一致。而不同数据库具有相同特征值的概率为 $1/p$ 。要降低错误率重复测试多次就可以了。例10-3给出了该一致性检测算法的伪码。

例10-3：一致性检测算法的伪码

Sub Fingerprint Generation

```
Generate a sequence of k primes  $p_1, \dots, p_k$ 
for each prime  $p_k$ 
    transmit  $p_k$ 
    transmit  $(b_0 + b_1 \times 2 + b_2 \times 2^2 + b_3 \times 2^3 + \dots + b_{n-1} \times 2^{n-1}) \bmod p_k$ 
```

Sub Coherence Test

```
for each prime  $p_k$  and fingerprint  $f_k$ 
    if  $(f_k \neq (a_0 + a_1 \times 2 + a_2 \times 2^2 + \dots + a_{n-1} \times 2^{n-1}) \bmod p_k)$  then
        return "database incoherent"
    return "database coherent"
end sub
```

零知识证明

假设证明者Patti想让验证者Victor确认她的身份，而他们的通信信道是不安全的。假设分析者Albert在监听通信，并且试图将自己伪装成Patti。如果Patti和Victor知道某个密码：“Rosebud”，如果Patti通过发送这个密码来确认自己的身份，那么之后Albert也能够以同样的密码向Victor声明自己就是Patti。所以Patti想要建立一个更加安全的协议。

下面这个协议假设：对于大图来说同构图和Hamiltonian回路问题的复杂程度非常大，几乎无法解决。

Hamiltonian回路

给定一个图，是否存在这样一个回路：经过所有的点，且经过每个点仅一次，最后返回到起点。

同构图

给定两个图： $G_1=(V_1, E_1)$ 和 $G_2=(V_2, E_2)$ ，是否存在节点集合 V_1 和 V_2 间的某种映射，使得两张图完全一致。

图10-6给出了两张同构图及其映射关系的示例。

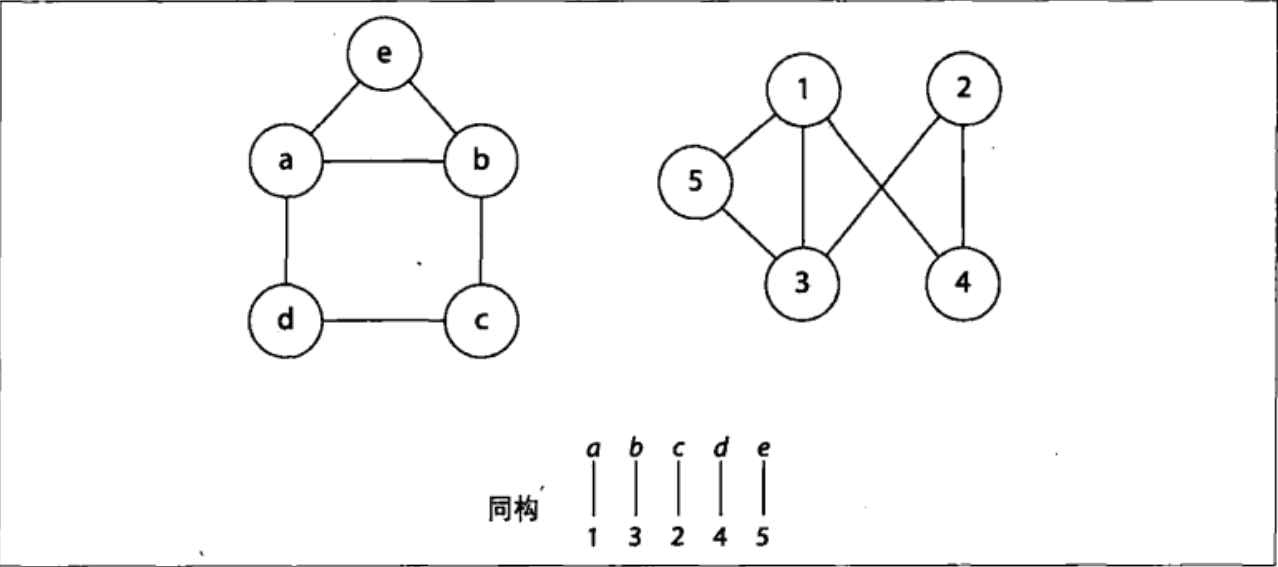


图10-6：同构图示例

对于大型实例来说这两个问题都不太可能找到有效的解决方法。我们利用了这些问题的难度开发了一种适合在不安全信道上的确认协议，同时保证Albert无法伪装成Patti。在开始确认前，Patti先建一张存在Hamiltonian回路的大图，具体可以这样做：先从一条回路开始，而这条回路会通过每一个节点，然后再不断地加边，直到对于其他人来说很难找出这个Hamiltonian回路的程度。然后Patti就可以在她名下的一个公开目录公布这个图 G_{patti} 。Victor和Albert都可以读取这张图，但只有Patti能够找出图 G_{patti} 中的Hamiltonian回路。

她可以通过像Victor发送这个Hamiltonian回路的节点顺序来确认自己的身份，但是，之后Albert或者Victor就都可以伪装成她（她的证明不是零知识证明）。她想要让Victor确认她知道这个秘密回路，而同时Victor或是Albert又不会知道她的知识。例10-4中描述了Patti的协议。

例10-4：不会泄露任何信息的协议

Patti通过随机映射图 G_{patti} 中的节点构建图H
Patti发送H到Victor
Victor掷硬币（两面分别表示ShowIsomorphism和ShowHamiltonianCycle）
Victor将掷硬币的结果发送给Patti
如果Patti收到的是ShowIsomorphism（给出映射关系）
 她就将用于构建H的映射表发送过去
否则
 她就将H中的Hamiltonian回路发送过去

无论Victor问Patti的是什么（无论掷硬币结果如何），Patti都能轻松应答，而Victor也能轻松验证。如果Albert想要伪装成Patti，就有两种可能性：他能够构造并传送自己的图H

和图 G_{patti} 或多或少有些类似（比如可能含有相同数量的边和节点），并且他知道其中的Hamiltonian回路。但是如果作为验证者的Victor说“ShowIsomorphism”的话，他便答不出来。又或者Albert重新标记并传送了图 G_{patti} 的节点。但是如果作为验证者的Victor说“ShowHamiltonianCycle”，他也答不出来。因此Albert只能伪造协议的一半。为了增加更多的可信度，Victor可以将协议进行100次（毕竟效率很高），Patti可以轻松成功，但是Albert可以成功伪造100次协议的概率则是 0.788×10^{-30} 。而且，即使Albert观察100次也无济于事。

参考文献

Armstrong, Joe, *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

Berman, Kenneth and Jerome Paul, *Algorithms: Sequential, Parallel, and Distributed*. Course Technology, 2004.

Christofides, Nicos, “Worst-case analysis of a new heuristic for the travelling salesman problem,” Report 388, Graduate School of Industrial Administration, CMU, 1976.

Knuth, Donald, “Estimating the efficiency of backtrack programs,” *Mathematics of Computation* 29: 121–136, 1975.





概述

我们已经到了本书的结尾，但是这并不意味着关于算法的方面知识已经到了尽头。更确切地说，本书所讲到的这些技术所面向的问题是无穷无尽的。

我们终于有机会回顾本书详细讲解并提供示例的三十多个算法。我们希望你对本书的讲解感到满意。为了显示出我们所涉及内容的广度，现在我们将汇总本书所提到的算法背后的原则。通过这样做，我们能展示用于解决不同问题的不同算法之间的相似性。我们不打算通过简单的汇总各个章节来作为结束，而是将目光放在关键性原则上，这些原则是最初设计这些算法的动机。我们还将利用这个机会来汇总每个算法所涉及的概念，这些概念在讲解算法时罗列在说明中，位于图表的右上角。这样我们便实现了一个快速汇总，通过交叉索引在不同的算法之间共享概念。

原则：了解数据

我们谈到了很多通用行为，这些行为需要作用在某些数据之上。也许你需要对数据排序来实现一个特定的顺序，也许你需要在数据之中搜索以定位某些特殊的信息。你的数据可能是可以随机访问的（也就是说，可以在任何时候读取任何一块数据），也可能只能通过迭代器来顺序访问（每次只能处理一个元素）。如果你不了解数据的特性，就只能使用一些较为通用的算法。

如果你需要对数据排序，没有“一刀切”的方法可以总能获得最好的性能。表11-1汇总了第4章谈到的排序算法的结论。你是要对一个整数集合进行排序，而这些整数的范围是有限的吗？那么最快的算法莫过于计数排序了，尽管它比其他算法需要更多的存储空间。你需要对已经几乎有序的复杂数据进行排序吗？那么插入排序将更适合。你需要关心那些相等元素的相对位置？那么你需要稳定排序算法。你确定数据提取自均匀分布

吗？那么你一定要试试桶排序，因为它能够利用这个特性来获得额外的性能提升。如果你对这些都十分了解，那么就能够通过数据来选择最合适的算法。

表11-1：第4章：排序算法

算法	最佳	平均	最差	涉及概念
INSERTION SORT	n	n^2	n^2	数组
MEDIAN SORT	$n \log n$	$n \log n$	n^2	数组、递归、分治
SELECT KTH	n	n	n^2	分治
BLUM-FLOYD- PRATT- RIVESTTARJAN (BFPR) Select K^{th}	n	n	n	递归、分治
QUICKSORT	$n \log n$	$n \log n$	n^2	数组、递归、分治
SELECTION SORT	n^2	n^2	n^2	数组、贪心
HEAP SORT	$n \log n$	$n \log n$	$n \log n$	数组、递归、二叉堆
COUNTING SORT	n	n	n	数组
BUCKET SORT	n	n	n	数组、散列

原则：将问题分解至更小的问题

在设计一个高效算法来解决问题时，能将问题分解成两个（甚至更多）更小的子问题是非常有帮助的。快速排序毫无疑问是最流行的排序算法之一。尽管通过精心构造的特殊情况会引发性能问题，快速排序还是做到在大数据集上平均性能最高。事实上，在 $O(n \log n)$ 的算法之中，都包含分解问题的思想，首先将一个大小为 n 的问题分解为两个 $n/2$ 的子问题，然后再将两个子问题的结果汇总，作为原问题的结果。要达到 $O(n \log n)$ 的复杂度，这两个步骤都需要达到 $O(n)$ 的复杂度才行。

快速排序是第一个能够达到 $O(n \log n)$ 性能的原地置换排序算法。它的成功之处就在于将问题分成两个部分，并递归地通过快速排序来解决更小的子问题（几乎与直觉恰恰相反）。

这种新方法仅仅简单地将问题一分为二，就能够带来令人惊异的性能提升。看看二分查找如何将一个大小为 n 的问题转换为两个大小为 $n/2$ 的问题。二分查找借助于查找问题的重复特性来递归地解决问题。

有时候通过分解为两个子问题来解决问题时可以不使用递归。比如凸包扫描就将两个上凸包和下凸包合并来构建最终的凸包。

有时，一个问题在相同的输入数据下可以被分解为不同的（看似无关的）子问题。Ford-Fulkerson算法通过反复搜索可以增加流的增广路来计算网络流中的最大流，一旦找不到增广路，问题的求解就完成了。插入排序反复查找数组中的最大值，并将其置换到数组的最右端，经过 n 次计算，数组便有序了。相似地，堆排序也是反复将堆中最大的元素置换到适当的位置上。

表11-2比较了第5章讨论过的查找算法。

表11-2：第5章：查找算法

算法	最佳	平均	最差	涉及概念
SEQUENTIAL SEARCH	1	n	n	数组、Brute Force
BINARY SEARCH	1	$\log n$	$\log n$	数组、分治
HASH-BASED SEARCH	1	1	n	数组、散列
BINARY TREE SEARCH	1	$\log n$	n	二叉树

原则：选择正确的数据结构

著名算法设计者Robert Tarjan曾经引用过这样一句话：“如果使用正确的数据结构，任何问题都能够在 $O(n \log n)$ 的时间内解决掉”。许多算法需要使用有限队列来存储中间过程或者下一步运算。实现优先队列最常见的要数二叉堆了，它能够在 $O(\log n)$ 的时间内从优先队列中将最低优先级的元素移除。然而，优先堆却不能判定是否包含某个元素。我们在谈到线段扫描（第9章）时提到了这一点，这个算法可以达到 $O(n \log n)$ 的性能，因为其使用了增广二叉树来实现优先队列，并且在移除最小元素时，还能够达到 $O(\log n)$ 的性能。这个原则也可以从反面来理解，如果选择了不恰当的数据结构，那么算法就很难发挥出它最大的威力。

表11-3：第6章讨论过的图算法

算法	最佳	平均	最差	涉及概念
DEPTH-FIRST SEARCH	$V+E$	$V+E$	$V+E$	图、数组、递归回溯
BREADTH-FIRST SEARCH	$V+E$	$V+E$	$V+E$	图、数组、队列
DIJKSTRA S ALGORITHM	$(V+E)$	$(V+E)$	$(V+E)$	加权有向图
PQ	$\log V$	$\log V$	$\log V$	数组、优先队列 Overflow
DIJKSTRA S ALGORITHM	V^2+E	V^2+E	V^2+E	加权有向图
DG				数组、Overflow

表11-3：第6章讨论过的图算法（续）

算法	最佳	平均	最差	涉及概念
BELLMAN-FORD ALGORITHM	$V \cdot E$	$V \cdot E$	$V \cdot E$	加权有向图 数组、Overflow
FLOYD-WARSHALL ALGORITHM	V^3	V^3	V^3	动态编程、2D 数组、加权有向图 Overflow
PRIM S ALGORITHM	$(V+E)$ $\log V$	$(V+E)$ $\log V$	$(V+E)$ $\log V$	加权图、二叉堆 优先队列 贪心、数组

原则：空间换时间

很多算法计算方面的优化是通过将过去的计算结果保存来实现的。计算图的最小生成树的Prim算法使用了一个优先队列来存储没有访问过的节点，来计算其与初始节点s之间的最短距离。作为算法中的关键一环，必须要判断出一个给定节点是否已经访问过。因为二叉堆实现的优先队列无法解决这个问题，通过一个单独的布尔数组inQueue来保存每个节点的访问状态。同样在这个算法中，一个重复的key数组保存了计算过的距离，以避免在优先队列中的重复查找。这些额外的存储需要耗费 $O(n)$ 的空间，却能够确保算法的高效实现。在大多数情况下，用上个 $O(n)$ 的空间无伤大雅。

有时，输入数据本就需要大量的存储空间，比如说第6章提到的稠密图。通过一个二维数组来存储节点信息，而不是简单地采用邻接表存储，可以显著地提高算法性能。另外，你可能注意到了，对于无向图来说，如果我们使用双倍的存储，即是说通过二维数组存储信息，让edgeInfo[i][j]和edgeInfo[j][i]相等，就可以让算法更加简化。现在就不必考虑当 $i \leq j$ 时访问edgeInfo[i][j]了，然而如果算法里只是需要知道边 (i, j) 是否存在，这样做可能会导致算法的复杂化。

在有些情况下，一个算法在没有大容量存储时是无法完成运算的。以桶排序为例，如果输入呈正态分布，它能够利用 $O(n)$ 的存储在线性时间内完成排序。即便目前的现代计算机能够提供非常大的内存，你还是应该注意桶排序的内存占用是非常高的。

原则：如果没有显而易见的解法，使用搜索

人工智能（AI）领域的先驱们经常需要解决一些没有已知解法的问题。解决这些问题的常用方法就是将这些问题的转化为一个（巨大）图的搜索问题。我们之所以花了一整章讲

述此问题，就是因为它非常重要，可以解决大量的问题。然而，要确保真地没有其他可替代的计算方法存在！你可以使用寻径方法来将一个无序数组作为起点（初始节点），找到一个元素变换序列，来生成目标数组（目标节点），但是你不应该用这个指数级时间的算法，因为已经存在大量的 $O(n \log n)$ 排序算法。表11-4展示了第7章谈到的寻径算法。

表11-4：第7章：人工智能中的寻径算法

算法	最佳	平均	最差	涉及概念
DEPTH-FIRST SEARCH	$b*d$	b^d	b^d	栈、集合、回溯
BREADTH-FIRST SEARCH	b^d	b^d	b^d	队列、集合
A*SEARCH	$b*d$	b^d	b^d	优先队列、集合、Heuristics
MINIMAX	b^{ply}	b^{ply}	b^{ply}	递归、回溯、Brute Force
NEGMAX	b^{ply}	b^{ply}	b^{ply}	递归、回溯、Brute Force
ALPHABETA	$b^{\text{ply}/2}$	$b^{\text{ply}/2}$	b^{ply}	递归、回溯、Heuristics

原则：如果没有显而易见的解法，将问题归约为另一个有解的问题

问题归约（Problem reduction）是计算机科学和数学中用于解决问题的一个基本方法。一个简单的例子，假如你需要一个算法来找到列表中的第四大的元素。不需要编写什么特殊的代码，你只需要用任何一种排序算法排个序，就能够在在一个有序的列表中取到第四大的元素了。使用这个方法，你可以获得 $O(n \log n)$ 的性能。当然这并不是最高效的解决方法，看看第4章中我们讲到的selectKth，它才是最快的。

第8章所描述的问题看似有关联，却不太容易将它们联系在一起。其实可以将这些问题归约为线性编程（Linear Programming），并使用现成的商用软件包（比如Maple）来计算，然而归约过程太过复杂。其实通用算法完全可以来求解线性编程问题，尤其是Ford-Fulkerson算法族。

我们在第8章里已经展示了如何解决一类问题，这类问题称为网络流的最大流最小割。一旦这个算法在手，其他五个问题也就迎刃而解了。

表11-5展示了第8章谈到了网络流算法。

表11-5：第8章：网络流算法

算法	最佳	平均	最差	涉及概念
FORD-FULKERSON	$E*mf$	$E*mf$	$E*mf$	加权有向图、数组、贪心
EDMONDS-KARP	$V*E^2$	$V*E^2$	$V*E^2$	加权有向图、数组、贪心

原则：编写算法难，测试算法更难

因为我们所谈到的算法主要都是确定的（除了第11章之外），因此可以直接开发测试用例来保证其行为正确。但是在第7章，我们遇到了麻烦，因为我们使用的是寻径算法来找到可能存在的解，我们对此却一无所知。例如，尽管可以编写出测试用例，来确定启发式的GoodEvaluator对于八数码是否可以正常工作。但是测试A*算法的唯一方法，却需要运行搜索并手工地查看搜索树，来验证是否选择了正确的移动。因此，测试A*算法是很复杂的，因为需要在一个特定问题和启发式的使用环境中去测试算法。我们有大量的测试用例用于寻径算法，但是在许多用例中，他们只是用来确保选择了一个“合理”的移动（无论是游戏还是搜索树），而不是确保选择了一个特定的解。

测试第9章中的算法就更是难上加难了，因为涉及了浮点计算。比如我们要测试凸包扫描，一开始我们打算使用一个穷举凸包扫描算法，算法复杂度为 $O(n^4)$ ，来生成结果并和Andrew的凸包扫描算法进行比较。在我们的测试中，在 $[0, 1]$ 之间正态地随机生成了二维点集。然而，当数据集变大时，我们总是遇到两个算法的结果不相等的问题。难道是数据发现了一个重大的缺陷？最终我们发现穷举扫描所使用的浮点数计算得到的结果和凸包扫描的结果有细微的不同（尽管非常细微）。这只是侥幸？不幸的是，并不是这样的。我们也注意到线段扫描算法所产生的结果和穷举相交算法的结果也有些许不同。那么，哪个算法的结果才是“正确”的结果呢？其实并不是那么简单，因为用到了浮点数，因此我们需要找到一个一致性的概念来比较浮点数的值。特别地，我们定义了FloatingPoint.epsilon作为阈值，来解决无法辨识两个数是否完全相等的问题。当两个结果之间的差异接近阈值时（我们设定为 10^{-9} ），还是会有无法预期的事情出现。将阈值完全删除也不能解决这个问题。最终，我们重新排序，并通过统计的方法来检查算法的结果，而不是仅仅去校验所有案例的返回结果和预期结果的比较。

表11-6综合了第9章谈到了计算几何问题。

表11-6: 第9章: 计算几何

算法	最佳	平均	最差	涉及概念
CONVEX HULL SCAN	n	$n \log n$	$n \log n$	数组、贪心
LINE SWEEP	$(n+k) \log n$	$(n+k) \log n$	n^2	优先队列、 二叉树
NEAREST NEIGHBOR QUERY	$\log n$	$\log n$	n	kd树、递归
RANGE QUERIES	$n^{1-1/d} + r$	$n^{1-1/d} + r$	n	kd树、递归



第四部分

附录 基准测试





本书阐述了多种算法，在其各自章节中，你能够找到这些算法的性能描述。本章中，我们将会描述如何评估算法性能。我们认为，根据经验使用特定的数据来精确解释算法行为是非常重要的。这样能够帮助读者验证结果的正确性，以及理解在使用的过程中，这些假设是否适用。

分析算法的方法有很多种。第2章是一种正式的理论分析方法，介绍了如何分析最坏情况和平均情况下的性能。在某些情况下，我们可以根据经验得到这些理论结果。例如，考虑排序20个数的算法性能。这20个数有 2.43×10^{18} 种转置，但是我们不可能详尽地计算每一种转置下算法的平均性能。而且，我们不可能对所有转置进行排序，然后计算平均性能。我们必须使用基于统计的方法来正确计算算法的期望性能。

统计基础

在本章中，我们简要地介绍一下评估算法性能的关注点。有兴趣的读者可以参考任意一本统计学教材来获取有关统计学的知识。

为了计算算法的性能，我们构架了一套独立的实验集 T 。每次实验的输入规模是 n 。我们也会做一些额外的工作来保证这些实验的等价。这时，我们就可以将算法实现中的变量量化，而不用全部进行所有的实验。这样也许比较合适，例如，计算大量独立的等价实验的开销非常巨大。这个集合执行的结果将会精确到毫秒。当代码使用Java编写时，我们会在执行之前先调用系统的垃圾回收器，虽然不能保证垃圾回收器不会在实验进行中执行，但是这样做能够减少和算法执行无关的时间花费。而且我们会将最好的和最坏的结果当作异常值抛弃，然后使用下面的公式计算出剩余的 $T-2$ 次结果的平均值和标准方差。

$$\sigma = \frac{\sum_i (x_i - x)^2}{n - 1}$$

x_i 是单独的实验执行时间，而 x 的 $T-2$ 次实验的平均值。这里 n 等于 $T-2$ ，所以平方根中的坟墓是 $T-3$ 。计算平均值和标准方差能够帮助我们预测算法的远期性能，表A-1是实际值在 $[x-k*\sigma, x+k*\sigma]$ 区间内的概率， σ 是计算出的标准方法。这些概率值组成了算法性能预测的置信区间。

表A-1：标准方差表

k	概率
1	0.6827
2	0.9545
3	0.9973
4	0.9999
5	1

例如，在一个随机实验中，68.27%的时间期望结果都落在 $[x-\sigma, x+\sigma]$ 这个范围内。

得出结果精度不会超过4位数，我们相信精度已经足够，不可能出现重大的误差。当精度超过5位数时，我们会将其截断，或者合理地舍入。例如16.897 986会舍入为16.8980。

硬件

在本书中，我们通过大量的表格说明了算法在样例数据上的性能。我们使用了两种不同的机器：

桌面PC

我们使用了一台“家庭办公”的PC。这台计算机的处理器是Pentium(R)4CPU 2.8GHz，内存512MB。

高端计算机

我们可以使用Linux集群中的一部分计算机。这些计算机的处理器是双核的AMD Opteron 2.6GHz，16G内存。

我们使用的高端计算机是由美国国家科学基金会提供，授权号是No.0551584。本书使用的任何选项、发现、结论和推荐都来自于作者，并不代表美国国家科学基金会的观点。

我们在这些表中指明了这些计算机。

例子

假设我们需要对 n 个数的加法做一个基准测试。那么我们可以设计一个实验， n 从1 000 000~5 000 000，步长是1000000。我们执行30次实验，试图消除尽可能多的奇异值。

我们假设计算的时间和 n 直接相关。我们使用Java、C和Scheme分别编写了三个程序来进行基准测试。

Java基准测试解决方案

在Java的测试用例中，我们会在执行前后记录当前系统时间（毫秒级）。例A-1的代码测量了完成任务所需的时间。在一台理想的计算机上执行30次实验，每次需要的时间应该精确相等。当然这不可能发生，因为现代操作系统需要处理大量的后台程序，这些程序会在代码执行时共享同一块CPU。

例A-1: Java测量任务执行时间程序

```
public class Main {
    public static void main (String[] args) {
        TrialSuite ts = new TrialSuite();
        for (long len = 1000000; len <= 5000000; len += 1000000) {
            for (int i = 0; i < 30; i++) {
                System.gc();
                long now = System.currentTimeMillis();

                /** 任务会被计时 */
                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());
    }
}
```

TrialSuite类存储实验集合。在所有的实验被加入到这个集合后，我们开始计算。我们会统计出总时间、最长时间和最短时间。如前所述，最大值和最小值将会在计算平均值和标准方差时删除。

Linux基准测试解决方案

对于C测试用例，我们开发了一个基准测试库。本节中，我们简单地描述一下计时代码的核心思想，并告诉感兴趣的读者源代码所在的代码库。

这个库主要是用于测试排序程序。计时的API也要负责对命令行参数进行解析：

```
用法: timing [-n NumElements] [-s seed] [-v] [OriginalArguments]
  -n 问题规模                [默认: 100, 000]
  -v 完全输出                [默认: 否]
  -s # 随机值的种子          [默认: 没有种子]
  -h 打印帮助信息
```

计时库假设问题的规模是由-n这个参数设定的。为了产生可复用的实验，随机值的种子可以通过-s来设定。一个测试用例需要包含以下函数：

void problemUsage()

计时库会解析计时参数，然后剩余的参数会传递给prepareInput函数。

void prepareInput (int size, int argc, char **argv)

这个函数和解决的问题相关，它负责构建输入集合。这个函数不能直接地通过形式参数传递执行，而是需要使用测试用例中的静态变量。

void postInputProcessing()

如果在问题解决之后需要做一些验证和后续处理，那么就在这里写一些代码。

void execute()

这个方法包含了功能代码。这里永远只会有一个方法调用。当这个方法为空时，总开销（在高端计算机上）平均是0.002毫秒，这个值被认为对最后的结果没有影响。

测试用例如例A-2所示。

例A-2: n个数的加法

```
extern int numElements; /* n的规模 */
void problemUsage() { /* 无实现体*/ }
void prepareInput() { /* 无实现体*/ }
void postInputProcessing() { /* 无实现体*/ }

void execute() {
    int x;
    long sum = 0;
    for (x = 1; x <= numElements; x++) { sum += x; }
}
```

每次实验都将会执行一次这些函数，所以我们需要写一个shell脚本，在不同数据上多次执行程序。每次测试时，我们都需要写一个配置文件。例A-3就是第4章中基于值排序算法的配置文件`config.rc`。

例A-3: 比较排序算法的配置文件样例

```
# 使用BINS进行配置
BINS=./Insertion ./Qsort_2_6_11 ./Qsort_2_6_6 ./Qsort_straight

#配置套件
TRIALS=10
LOW=1
HIGH=16384
INCREMENT=*2
```

这个配置文件声明了将会执行三个快速排序的变种和一个插入排序。这次测试的数据规模 n 从1~16 384, n 每次增加一倍。对于特定的数据规模, 将会执行10次实验, 最好和最坏的结果将被抛弃。然后输出剩下8次实验结果的平均值(和标准方差)。

例A-4: compare.sh基准测试脚本

```
#!/bin/bash
#
# This script expects TWO arguments:
#   $1 -- size of problem n
#   $2 -- number of trials to execute
# This script reads its parameters from the $CONFIG configuration file
#   BINS set of executables to execute
#   EXTRAS extra command line arguments to use when executing them
#
# CODE is set to directory where these scripts are to be found
CODE=`dirname $0`

SIZE=20
NUM_TRIALS=10
if [ $# -ge 1 ]
then
    SIZE=$1
    NUM_TRIALS=$2
fi

if [ "x$CONFIG" = "x" ]
then
    echo "No Configuration file (\$CONFIG) defined"
    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 "no \$BINS variable and no $CONFIG configuration "
        echo "Set \$BINS to a space-separated set of executables"
```

```

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

    # start with number of trials followed by totals (one per line)
    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

    # compute average/stdev
    RES=`cat $RESULTS | $CODE/eval`
    echo "$b $RES"

    rm -f $RESULTS
done

```

*compare.sh*脚本使用一个C程序eval计算平均值和标准方差，这个程序使用了本章开头的一些方法。例A-5是一个管理脚本*suiteRun.sh*，这个脚本加载配置文件，然后根据配置文件重复执行*compare.sh*脚本。

例A-5: *suiteRun.sh*基准测试脚本

```

#!/bin/bash
CODE=`dirname $0`

# if no args then use default config file, otherwise expect it
if [ $# -eq 0 ]
then
    CONFIG="config.rc"
else
    CONFIG=$1
    echo "Using configuration file $CONFIG..."
fi

# export so it will be picked up by compare.sh
export CONFIG

# pull out information
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=" $CONFIG | cut -f2- -d='`
else
    echo "Configuration file ($CONFIG) unable to be found."
    exit -1
fi

# headers
HB=`echo $BINS | tr ' ' ', '`
echo "n, $HB"
# compare trials on sizes from LOW through HIGH
SIZE=$LOW
REPORT=/tmp/Report.$$
while [ $SIZE -le $HIGH ]
do
    # one per $BINS entry
    $CODE/compare.sh $SIZE $TRIALS | awk 'BEGIN{p=0} \
        {if(p) { print $0; }} \
        /Host:/{p=1}' | cut -d' ' -f2 > $REPORT

    # concatenate with , all entries ONLY the average. The stdev is
    # going to be ignored
    # -----
    VALS=`awk 'BEGIN{s=""}\
        {s = s ", " $0 }\
        END{print s;}' $REPORT`
    rm -f $REPORT

    echo $SIZE $VALS

    # $INCREMENT can be "+ NUM" or "* NUM", it works in both cases.
    SIZE=$(( $SIZE $INCREMENT ))
done

```

Scheme基准测试解决方案

本节的Scheme代码测量的是在给定数据上，代码的性能。本例中，我们只需将数据规模作为参数传递给函数。首先我们会列出一些helper函数来计算平均值和标准方差。请看例A-6。

例A-6: Scheme计时程序的helper函数

```

;; foldl: (X Y -> Y) Y (listof X) -> Y
;; Folds an accumulating function f across the elements of lst.
(define (foldl f acc lst)
  (if (null? lst)
      acc
      (foldl f (f (car lst) acc) (cdr lst))))

;; remove-number: (listof number) number -> (listof number)
;; remove element from list, if it exists
(define (remove-number nums x)

```

```

(if (null? nums) '()
    (if (= (car nums) x) (cdr nums)
        (cons (car nums) (remove-number (cdr nums) x)))))

;; find-max: (nonempty-listof number) -> number
;; Finds max of the nonempty list of numbers.
(define (find-max nums)
  (foldl max (car nums) (cdr nums)))
;; find-min: (nonempty-listof number) -> number
;; Finds min of the nonempty list of numbers.
(define (find-min nums)
  (foldl min (car nums) (cdr nums)))

;; sum: (listof number) -> number
;; Sums elements in nums.
(define (sum nums)
  (foldl + 0 nums))

;; average: (listof number) -> number
;; Finds average of the nonempty list of numbers.
(define (average nums)
  (exact->inexact (/ (sum nums) (length nums))))

;; square: number -> number
;; Computes the square of x.
(define (square x) (* x x))

;; sum-square-diff: number (listof number) -> number
;; helper method for standard-deviation
(define (sum-square-diff avg nums)
  (foldl (lambda (a-number total)
            (+ total (square (- a-number avg))))
    0
    nums))

;; standard-deviation: (nonempty-listof number) -> number
;; Calculates standard deviation.
(define (standard-deviation nums)
  (exact->inexact
    (sqrt (/ (sum-square-diff (average nums) nums)
              (length nums)))))

```

例A-7的计时代码采用了例A-6的helper函数，执行一系列的测试用例。

例A-7: Scheme计时代码

```

;; Finally execute the function under test on a problem size
;; result: (number -> any) -> number
;; Computes how long it takes to evaluate f on the given probSize.
(define (result f probSize)
  (let* ((start-time (current-inexact-milliseconds))
        (result (f probSize))
        (end-time (current-inexact-milliseconds)))
    (- end-time start-time)))

```

```

;; trials: (number -> any) number number -> (listof number)
;; Construct a list of trial results
(define (trials f numTrials probSize)
  (if (= numTrials 1)
      (list (result f probSize))
      (cons (result f probSize)
            (trials f (- numTrials 1) probSize))))

;; Generate an individual line of the report table for problem size
(define (smallReport f numTrials probSize)
  (let* ((results (trials f numTrials probSize))
        (reduced (remove-number
                   (remove-number results (find-min results))
                   (find-max results))))
    (display (list 'probSize: probSize
                   'numTrials: numTrials
                   (average reduced)))
    (newline)))

;; Generate a full report for specific function f by incrementing
;; one to the problem size
(define (briefReport f inc numTrials minProbSize maxProbSize)
  (if (>= minProbSize maxProbSize)
      (smallReport f numTrials minProbSize)
      (begin
        (smallReport f numTrials minProbSize)
        (briefReport f inc numTrials (inc minProbSize) maxProbSize))))

;; standard doubler and plus1 functions for advancing through report
(define (double n) (* 2 n))
(define (plus1 n) (+ 1 n))

```

例A-8的largeAdd函数将 n 个数相加。(briefReport largeAdd millionplus 30 1000000 5000000) 得到的输出见表A-2。

例A-8: largeAdd函数

```

;; helper method
(define (millionplus n) (+ 1000000 n))

;; Sum numbers from 1..probSize
(define (largeAdd probSize)
  (let loop ([i probSize]
            [total 0])
    (if (= i 0)
        total
        (loop (sub1 i) (+ i total)))))

```


表A-2: largeAdd函数30次实验的执行时间

n	执行时间 (单位: 毫秒)
1,000,000	382.09
2,000,000	767.26
3,000,000	1155.78
4,000,000	1533.41
5,000,000	1914.78

报告

审视在相同的平台上（这里是Linux 2.6.9-67.0.1.ELsmp i686，机器是高端计算机和桌面PC）的实际结果是非常有帮助的。我们列出3个表（表A-3、表A-5、表A-6），分别是Java、C和Scheme的结果。在每张表中，时间的单位都是毫秒，并且给Java的结果描绘一张直方图。

表A-3: Java进行30次计算的计时

n	平均	最小	最大	标准差	#
1,000,000	8.5	8	18	0.5092	28
2,000,000	16.9643	16	17	0.1890	28
3,000,000	25.3929	25	26	0.4973	28
4,000,000	33.7857	33	35	0.4179	28
5,000,000	42.2857	42	44	0.4600	28

表A-3的行为概况请见表A-4的直方图。我们忽略那些0值的表行。所有非零值都用阴影表示。

表A-4: 计时结果

时间 (毫秒)	1,000,000	2,000,000	3,000,000	4,000,000	5,000,000
8	15	0	0	0	0
9	14	0	0	0	0
16	0	2	0	0	0
17	0	28	0	0	0
18	1	0	0	0	0
25	0	0	18	0	0
26	0	0	12	0	0

表A-4: 计时结果 (续)

时间 (毫秒)	1,000,000	2,000,000	3,000,000	4,000,000	5,000,000
33	0	0	0	7	0
34	0	0	0	22	0
35	0	0	0	1	0
42	0	0	0	0	21
43	0	0	0	0	8
44	0	0	0	0	1

为了解释这些结果, 我们使用统计的方法。如果假设每次实验的计时都是独立的, 那么我们将会使用之前的置信区间。如果有人希望我们预测一下 $n=4\,000\,000$ 时算法的性能, 我们可以说期望时间在 $[32.9499, 34.6215]$ 之间的几率是95.45%。

表A-5: C进行30次计算的结果

n	平均	最小	最大	标准差	#
1,000,000	2.6358	2.589	3.609	0.1244	28
2,000,000	5.1359	5.099	6.24	0.0672	28
3,000,000	7.6542	7.613	8.009	0.0433	28
4,000,000	10.1943	10.126	11.299	0.0696	28
5,000,000	12.7272	12.638	13.75	0.1560	28

C实现的速度比Java实现快3倍左右。但是这个直方图不能提供足够的信息, 因为C的计时精确到了毫秒级, 而Java的计时却只是秒级。

最后一张表是Scheme的结果。其结果的变化程度比Java和C更高。一个可能的原因是递归解需要更多的簿记计算。

表A-6: Scheme进行30次计算的结果

n	平均	最小	最大	标准差	#
1,000,000	1173	865	1,274	7.9552	28
2,000,000	1921.821	1,824	2,337	13.1069	28
3,000,000	3059.214	2,906	3,272	116.2323	28
4,000,000	4040.607	3,914	4,188	81.8336	28
5,000,000	6352.393	6,283	6,452	31.5949	28

精度

我们发现毫秒级的精度不能满足我们的需求，因此我们引入了纳秒级的计时器。在Java平台上，计时代码唯一需要做出的改变就是调用`System.nanoTime()`这个计时函数。为了检查纳秒级和毫秒级计时器之间是否存在某些关联，我们做了一些修改，代码如例A-9所示。

例A-9：在Java中使用纳秒级计时器

```
TrialSuite tsM = new TrialSuite();
TrialSuite tsN = new TrialSuite();
for (long len = 1000000; len <= 5000000; len += 1000000) {
    for (int i = 0; i < 30; i++) {
        long nowM = System.currentTimeMillis();
        long nowN = System.nanoTime();
        long sum = 0;
        for (int x = 0; x < len; x++) { sum += x; }
        long endM = System.currentTimeMillis();
        long endN = System.nanoTime();
        tsM.addTrial(len, nowM, endM);
        tsN.addTrial(len, nowN, endN);
    }
}
```

之前的表A-3是毫秒级的计时结果，表A-7所示的计时结果是纳秒级。我们可以看到最明显的差别就是标准方差收缩了一个数量级，因此能够更加准确地预测代码的期望执行时间。但是这个结果仍然存在精度上的问题——注意 $n=5\,000\,000$ 时的标准方差。这个值看起来是一个奇异值。

表A-7：使用纳秒级计时器的结果

n	平均	最小	最大	标准差	#
1,000,000	8.4833	8.436	18.477	0.0888	28
2,000,000	16.9096	16.865	17.269	0.0449	28
3,000,000	25.3578	25.301	25.688	0.0605	28
4,000,000	33.8127	33.729	34.559	0.0812	28
5,000,000	42.3508	42.19	43.207	0.2196	28

因为我们相信使用纳秒级的计时器提供的帮助非常有限，所以我们仍然在算法相关章中使用毫秒级计时器来进行基准测试。我们也会对计时器进行修改，提供更精确的计时，避免出现重大的误差。而且，当我们希望跨平台比较执行时间时，UNIX系统上的纳秒级计时器还没有标准化，这也是我们采用毫秒级计时器的另外一个原因。

为什么同一份数据执行的时间不一致呢？让我们看看表A-3的数据，不同结果之间有1:1毫秒或16毫秒的波动。这些波动反映了Windows平台上Java计时器的精度问题，而不是我们的代码的问题。`System.currentTimeMillis()`执行时会导致这些偏差值，但是只有在执行时间非常短时，这些值才会非常重要（例如，大约16毫秒）。

Sun Java工程师已经意识到了Windows平台上Java计时器的问题，但是现在没有解决这个问题的计划（这种状况已经存在6年）。相关信息可参见http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4423429。

