计 算 机 科 学 丛 书

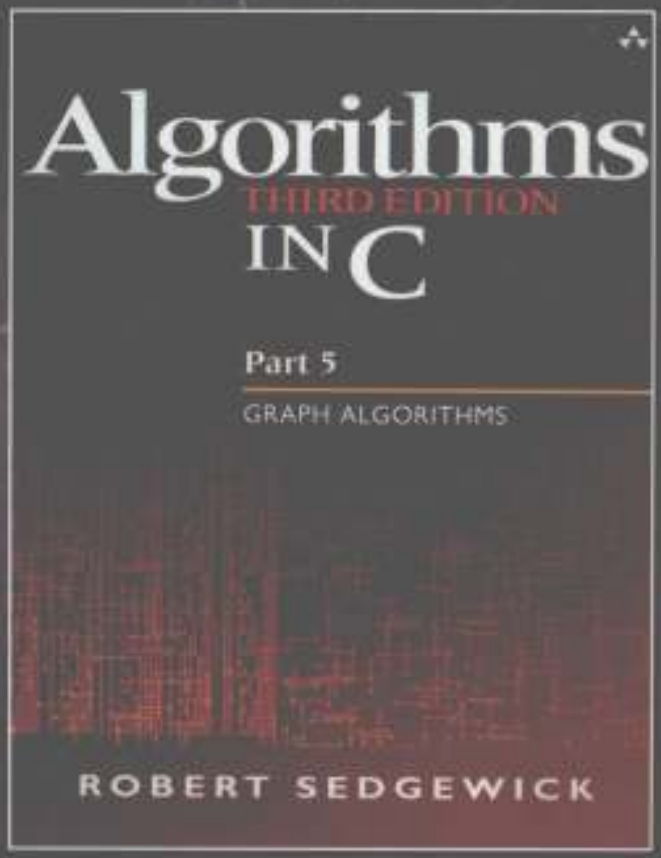
原书第3版

算法：C语言实现

（第5部分）

图算法

(美) Robert Sedgewick 普林斯顿大学 著 霍红卫 西安电子科技大学 译



Algorithms in C

Parts 5, Graph Algorithms

Third Edition

机械工业出版社
China Machine Press

算法：C语言实现（第5部分）图算法（原书第3版）

对于在数学分析方面不算熟练且需要留意理论算法的普通程序员来说，本书是一本可读性很强的优秀读本，他们应该会从中获益良多。

——Steve Summit，《C Programming FAQs》的作者

Sedgewick有一种真正的天赋，可以用易于理解的方式来解释概念。书中采用了一些易懂的实践程序，其篇幅仅有一页左右，这更是锦上添花。而书中大量采用的图、程序、表格也会极大帮助读者的学习和理解，这使本书更显得与众不同。

——William A. Ward，南亚拉巴马大学

本书是Sedgewick彻底修订和重写的C算法系列的第二本，集中讲解图算法。全书共有6章（第17~22章）。第17章详细讨论图性质和类型，第18~22章分别讲解图搜索、有向图和DAG、最小生成树、最短路径以及网络流。

书中提供了用C语言描述的完整算法源程序，并且配有丰富的插图和练习。作者用简洁的实现将理论和实践成功地结合了起来，这些实现均可在真实应用上测试，使得本书自问世以来备受程序员的欢迎。

本书可作为高等院校计算机相关专业算法与数据结构课程的教材和补充读物，也可供自学之用。

本书作者的网站http://www.cs.princeton.edu/~rs/为程序员提供了本书的源代码和勘误表。

作者简介

Robert Sedgewick 拥有斯坦福大学博士学位（导师为Donald E. Knuth），普林斯顿大学计算机科学系教授，Adobe Systems公司董事，曾是Xerox PARC的研究人员，还曾就职于美国国防部防御分析研究所以及INRIA。除本书外，他还与Philippe Flajolet合著了《算法分析导论》一书。





影印版
ISBN 7-111-19769-0
定价：49.00元



影印版
ISBN 7-111-19764-X
定价：69.00元



中文版
ISBN 7-111-27571-8
定价：79.00元

客服热线：(010) 88378991, 88361066
购书热线：(010) 68326294, 88379649, 68995259
投稿热线：(010) 88379604
读者信箱：hzjsj@hzbook.com

华章网站 http://www.hzbook.com

网上购书：www.china-pub.com

www.PearsonEd.com

上架指导：计算机、算法

ISBN 978-7-111-28505-2

9 787111 285052

定价：59.00元

计 算 机 科 学 丛 书

原书第3版

算法：C语言实现
（第5部分）
图算法

（美） Robert Sedgewick 普林斯顿大学 著 霍红卫 西安电子科技大学 译

Algorithms in C
Part 5, Graph Algorithms
Third Edition

机械工业出版社
China Machine Press

本书是深入论述算法的三卷本教程《算法：C语言实现》（第3版）中的第二卷——图算法。作者在这次修订中重写了许多内容，增加了数千个新练习、数百个新图表、数十个新程序，并对图表和程序做了详尽的注释说明。新版中不仅涵盖了新的主题，而且还提供了对许多经典算法的更充分的解释，包括图的性质、图搜索、有向图、最小生成树、最短路径和网。本书涵盖了足够的基本内容及较详细的图算法高级主题，既可单独用作数据结构与算法课程的教材，也可与第一卷（第1～4部分）结合使用。

本书适合高等院校计算机专业师生参考，也可供软件开发人员参考。

Simplified Chinese edition copyright © 2010 by Pearson Education Asia Limited and China Machine Press.

Original English language title: *Algorithms in C: Part 5, Graph Algorithms; Third Edition* (ISBN 0-201-31663-3) by Robert Sedgewick. Copyright © 2002.

All rights reserved.

Published by arrangement with the original publisher, Pearson Education, Inc., publishing as Addison-Wesley.

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

本书法律顾问　北京市展达律师事务所

本书版权登记号：图字：01-2006-3990

图书在版编目（CIP）数据

算法：C语言实现——第5部分，图算法（原书第3版）/（美）塞奇威克（Sedgewick, R.）著；霍红卫译. —北京：机械工业出版社，2009.11

（计算机科学丛书）

书名原文：Algorithms in C: Part 5, Graph Algorithms; Third Edition

ISBN 978-7-111-28505-2

Ⅰ.算…　Ⅱ.①塞…　②霍…　Ⅲ.①电子计算机－算法理论　②C语言－程序设计　Ⅳ.TP301.6　TP312

中国版本图书馆CIP数据核字（2009）第183990号

机械工业出版社（北京市西城区百万庄大街22号　邮政编码100037）

责任编辑：刘立卿

北京市荣盛彩色印刷有限公司印刷

2010年1月第1版第1次印刷

184mm×260mm · 19.75印张

标准书号：ISBN 978-7-111-28505-2

定价：59.00元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：（010）88378991；88361066

购书热线：（010）68326294；88379649；68995259

投稿热线：（010）88379604

读者信箱：hzjsj@hzbook.com

出版者的话

文艺复兴以降，源远流长的科学精神和逐步形成的学术规范，使西方国家在自然科学的各个领域取得了垄断性的优势；也正是这样的传统，使美国在信息技术发展的六十多年间名家辈出、独领风骚。在商业化的进程中，美国的产业界与教育界越来越紧密地结合，计算机学科中的许多泰山北斗同时身处科研和教学的最前线，由此而产生的经典科学著作，不仅擘划了研究的范畴，还揭示了学术的源变，既遵循学术规范，又自有学者个性，其价值并不会因年月的流逝而减退。

近年，在全球信息化大潮的推动下，我国的计算机产业发展迅猛，对专业人才的需求日益迫切。这对计算机教育界和出版界都既是机遇，也是挑战；而专业教材的建设在教育战略上显得举足轻重。在我国信息技术发展时间较短的现状下，美国等发达国家在其计算机科学发展的几十年间积淀和发展的经典教材仍有许多值得借鉴之处。因此，引进一批国外优秀计算机教材将对我国计算机教育事业的发展起到积极的推动作用，也是与世界接轨、建设真正的世界一流大学的必由之路。

机械工业出版社华章分社较早意识到“出版要为教育服务”。自 1998 年开始，华章分社就将工作重点放在了遴选、移译国外优秀教材上。经过多年的不懈努力，我们与 Pearson，McGraw-Hill，Elsevier，MIT，John Wiley & Sons，Cengage 等世界著名出版公司建立了良好的合作关系，从他们现有的数百种教材中甄选出 Andrew S. Tanenbaum，Bjarne Stroustrup，Brain W. Kernighan，Dennis Ritchie，Jim Gray，Afred V. Aho，John E. Hopcroft，Jeffrey D. Ullman，Abraham Silberschatz，William Stallings，Donald E. Knuth，John L. Hennessy，Larry L. Peterson 等大师名家的一批经典作品，以“计算机科学丛书”为总称出版，供读者学习、研究及珍藏。大理石纹理的封面，也正体现了这套丛书的品位和格调。

“计算机科学丛书”的出版工作得到了国内外学者的鼎力襄助，国内的专家不仅提供了中肯的选题指导，还不辞劳苦地担任了翻译和审校的工作；而原书的作者也相当关注其作品在中国的传播，有的还专程为其书的中译本作序。迄今，“计算机科学丛书”已经出版了近两百个品种，这些书籍在读者中树立了良好的口碑，并被许多高校采用为正式教材和参考书籍。其影印版“经典原版书库”作为姊妹篇也被越来越多实施双语教学的学校所采用。

权威的作者、经典的教材、一流的译者、严格的审校、精细的编辑，这些因素使我们的图书有了质量的保证。随着计算机科学与技术专业学科建设的不断完善和教材改革的逐渐深化，教育界对国外计算机教材的需求和应用都将步入一个新的阶段，我们的目标是尽善尽美，而反馈的意见正是我们达到这一终极目标的重要帮助。华章分社欢迎老师和读者对我们的工作提出建议或给予指正，我们的联系方法如下：

华章网站：www. hzbook. com
电子邮件：hzjsj@ hzbook. com
联系电话：（010）88379604
联系地址：北京市西城区百万庄南街 1 号
邮政编码：100037



华章教育
华章科技图书出版中心

译者序

本书是算法方面的优秀著作之一。它系统地阐述了算法学的特征以及算法的应用领域，讨论了算法分析在理论计算机科学中的作用。并通过实验数据和分析结果表明选择何种算法来解决实际应用问题。这卷书对图的性质和类型、图搜索、有向图、最小生成树、最短路径及网络流等内容进行了透彻的论述。通过归约的概念，建立了调度问题与最短路径问题之间的关系。

这本书不仅适合于软件设计人员和计算机科学专业的学生，而且也适合于那些想利用计算机并想使它更有效或是想要解决更大问题的人们。这本书中的算法代表了图算法领域所研究的知识主体。对于大量的应用问题，这些知识主体已经成为有效使用计算机的不可缺少的部分，尤其体现在网络算法、电路设计、调度、事务处理、资源分配等领域，在此所描述的基本方法在这些领域的科学研究及应用中已日显重要。作为最有影响的搜索引擎之一Google，其中最有名的PageRank算法就是图模型成功应用的一个典型代表。

本书主要内容及特点如下：

- 对于图的性质及类型做了完整的综述。
- 提供了有向图、最小生成树、最短路径及网络流方面的诸多算法，这些算法是计算机科学的核心基础。
- 对算法提供了很多可视化的信息，还有大量实验研究和基本分析研究，从而为选择算法解决实际应用问题提供了依据。
- 提供了700多个练习题，从而有助于深入理解算法的特征以及设计有效的算法。
- 本书以大量图例说明算法的工作过程，使得算法更加易于理解和掌握。
- 本书提供了读者易于实现和调试的C语言描述的算法的详尽信息，并通过示例程序展示了算法工作的详尽过程。
- 适合作为高等院校算法设计课程的教材，同时可作为从事软件开发和工程设计的专业人员的参考书。

由于时间较紧及译者水平有限，译文难免有错误及不妥之处，恳请读者批评指正。

最后感谢本书的编辑们所做的细致工作，使得本书的文字更加优美和流畅。

霍红卫
西安电子科技大学计算机学院
2009年9月

中文版序

In the years since this book was first published in English, there has been a profound shift in the way that students can learn algorithms. Specifically, the widespread deployment of personal computers and workstations has meant that each student can have computational experience with each algorithm learned. Teachers, students, and readers in China who take advantage of this shift have the opportunity to gain deeper understanding of a broader range of material about algorithms than ever before.

Students can know that all of the code in this book has been implemented and tested. The code that appears in the book is available on the web, and each student should, from the beginning, get into the habit of running each algorithm, testing it on various inputs, developing basic experimental results, and comparing the performance of different algorithms for the same task. With this preparation, a student often can answer a question that comes to mind about an algorithm, by running it. If a modification that might improve the algorithm comes to mind, a student can study whether or not it does so. I am certain that new variations of these algorithms and new algorithms to solve the problems that we address will be discovered as hundreds of thousands (or even millions) of a new generation of students learn to work with algorithms in such a concrete manner.

Teachers can encourage students to learn properties of each algorithm in detail, with knowledge that the book builds layers of abstraction. Data structures such as priority queues and symbol tables might seem rather abstract when students first encounter them, but they play an essential role in developing efficient solutions to complicated real-world problems. With a solid understanding of the basic abstractions, students are prepared to appreciate their importance in solving the more difficult problems. By encouraging students to treat the understanding of each algorithm and data structure as an intellectual exercise, teachers often can later reward that effort by showing how to solve a real-world problem that could not be approached without proper use of a fundamental algorithm.

As mentioned in the Notes on Exercises in the original book, there are far too many exercises for anyone to attempt them all. The most important advice that I can give each student, teacher, and reader in China is to read all the exercises and use them (and the basic material in the text) as a starting point to expand our knowledge about algorithms.

自本书首次在英国出版以来，学生们学习算法的方法一直在发生着深刻的转变。具体来说，个人电脑和工作站的广泛使用，意味着每个学生都可以实现所学的每个算法。中国的教师、学生以及读者谁利用了这种转变，就有机会获得关于算法的更广泛素材的深层次的理解，这是前所未有的。

学生们可能知道，书中的所有代码均已实现并测试过。出现在书中的代码可在网上得到，每个学生应该从一开始，就习惯性地运行每个算法，测试它的各种输入，开发基本的实验结果，并比较同一任务的不同算法的性能。有了这个准备，学生通过运行算法，往往能回答所想到的算法问题。如果学生想到改进算法的办法，可以进行深入的研究以确定是否会这样。我确信，随着成千上万（甚至百万）新一代的学生学会以这种具体方式所表示的算法，就会发现这些算法新的变型以及发现求解所涉及问题的新算法。

教师可以鼓励学生利用本书所构建的抽象层的知识来详细了解每个算法的性质。当学生首次遇到优先队列和符号表这样的数据结构时，它们看起来相当抽象，但这些数据结构在开发复杂实际现实问题的有效解决方案方面起着重要的作用。有了对于基本抽象的扎实理解，学生就可准备欣赏它们在解决更加困难问题中的重要性。教师要鼓励学生理解每个算法和数据结构，这样他们将能解决现实世界的问题，而这些是要通过合理地使用基本算法才能做到的。

正如原书练习中的注释中提到的，有太多的练习需要人们尝试。我能给中国学生、教师以及读者的最重要的建议，就是阅读所有的练习，并使用它们（和文中的基本素材）作为我们扩充关于算法知识的起点。

Robert Sedgewick

2009 年 10 月

前　言

图和图算法在现代计算机应用中颇为常见。对于在实际中出现的一些图处理问题，本书描述了目前最重要的解决方法。由于需要相关知识的人日渐增多，本书的主要目标是使读者了解这些方法及其所蕴含的基本原理。本书从基本原理展开，并从基本信息开始，从经典方法到现代仍在研发中的技术逐一展开讨论。精心选择的示例、详尽的图表以及完善实现的补充材料无一不体现在算法和应用的描述中。

算法

本书是对当前使用的最重要的计算机算法进行深入研究的三卷中的第二卷：图算法。第一卷（第 1 ~4 部分）覆盖了基本概念（第 1 部分）、数据结构（第 2 部分）、排序算法（第 3 部分）和搜索算法（第 4 部分）；本卷（第 5 部分）覆盖了图与图算法；（尚未出版的）第三卷（第 6 ~8 部分）覆盖了字符串（第 6 部分）、计算几何（第 7 部分）和高级算法及应用（第 8 部分）。

这些书可作为计算机科学低年级本科生的教材。学习本课程之前要求学生掌握基本程序设计技巧并熟知计算机系统，不过尚未选修计算机科学或计算机应用的高级领域的专业课程。这些书还可用作自学或作为从事计算机系统或应用程序开发的参考读本，因为它们包含了实用算法的实现以及关于这些算法性能特征的详尽信息。这些书包含内容广泛，适合作为这一领域的入门读物。

多年以来，这三卷书共同构成的《算法：C 语言实现》（第 3 版）已经得到世界各地的学生和程序员的广泛使用。我完全重写了这一版的内容，并且增加了数千个新练习、数百个新图表、数十个新程序以及对图表和程序详尽的注释说明。这个新版本不仅涵盖了新的主题，而且还提供了对许多经典算法的更充分的解释。全书对抽象数据类型的强调使这些程序使用更为广泛，而且在现代面向对象的编程环境中也更为适用。对于已经阅读过以前版本的人来说，会从这一版找到更为丰富的信息；并且所有读者都会从中找到富有教益的内容，有效地学习本书提供的基本概念。

这些书适合于程序员和计算机科学专业的学生阅读。每一个使用计算机的人都希望它能运行得更快，或者可解决更大规模的问题。我们所考虑的算法代表了近 50 年发展起来的知识体系，该体系是在各种应用中有效地使用计算机解决问题不可缺少的部分。从物理学中的 *N*-体模拟问题到分子生物学中的基因序列问题，在此所描述的基本方法在科学研究中已日显重要；另外，对于从数据库系统到 Internet 搜索引擎，这些方法已经成为现代软件系统的重要组成部分。随着计算机应用的覆盖面越来越广，基本算法的影响也日益显著，特别是本书所涵盖的基本图算法，作用更为突出。本书的目标是要提供一种资源，使广大学生以及专业人士可以了解并明智地利用图算法来解决计算机应用中出现的问题。

本书范围

本书共 6 章，包含了图的性质及类型、图搜索、有向图、最小生成树、最短路径和网。希望本书描述的方法使读者能够尽可能广地理解图算法的基本属性。

如果读者已经学过算法设计与分析的基本原理，并且有利用C、Java或C++等高级编程语言的经验，你将会更有效地学习本书的内容。当然，可以利用本书第一卷（第1～4部分）做充分的准备。本书假设读者有关于数组、链表和ADT设计的基本知识，并使用过优先队列、符号表以及合并－查找ADT。它们都在前四部分中描述过（同时也在其他关于算法与数据结构的书中介绍过）。

图和图算法的基本性质根据基本原理即可确立，但要充分理解算法的性质却需要扎实的数学基础。尽管这里所讨论的高级数学概念是简明的、概括性的和描述性的，但相对于前四部分的主题，还是需要读者有较好的数学基础才能更深入地理解图算法。不过，有不同数学基础的读者都可从中受益。这是因为每个人应该理解并使用的基本图算法只是与并非所有人都理解的高级算法略有差异。在此的主要意图是把一些重要的算法与贯穿本书的其他一些方法放在一起讨论，而不是对所有数学知识做全面的介绍。但是，严谨性是好的数学基础所要求的，由此可以得到好的程序。因此我尽量在理论家所喜爱的形式化处理和实践家所需要的内容丰富性之间进行权衡，同时不失严谨性。

教学用法

在教学中如何使用本书内容有很大的灵活性，这取决于教师的偏好以及学生所做的准备。这里所描述的算法多年以来已经得到广泛应用，而且无论对于从事实际工作的程序员还是计算机科学专业的学生，这些算法都代表了基本的知识体系。书中涵盖了足够的基本内容可用作数据结构和算法课程的学习，也有足够详细的高级主题用于图算法课程。有些教师可能希望强调与实现和实践有关的内容，而另外一些教师则可能把重点放在分析和理论概念上。

也可把本书与前四部分的某些内容结合起来作为一个更为全面的课程教材。这样，教师就可以用一种一致的风格来讲授基础知识、数据结构、排序、搜索和图算法的内容。教学中使用的幻灯片、程序设计示例作业、为学生提供的交互式练习以及与课程有关的其他资料都可在本书的主页上找到。

书中的练习（几乎全都是在这一版中新增加的）可分为几类。一类练习的目的是为了测试对正文中内容的理解，要求读者能够完成某个例子或应用正文中描述的概念。另一类练习则涉及实现算法和把算法整理到一起，或者进行实验研究从而对各种算法进行比较以及了解其性质。还有一类练习则是一些重要信息的知识库，其详细程度本身不适合放在正文中。阅读并思考这些练习，会使每个读者受益匪浅。

实用算法

任何人若希望更有效地使用计算机，都可以把这本书用作参考书，或用于自学。有程序设计经验的人可以从本书中找到有关某个特殊主题的信息。从很大程序上说，尽管某些情况下某一章中的算法使用了前一章中的方法，但读者仍可以独立于本书的其他章节阅读本书的某个章节。

本书的定位是研究实用的算法。本书提供了算法的详尽信息，读者可以放心地实现和调试算法，并使算法能够用于求解某个问题，或者为某个应用提供相关功能。书中包括了所讨论的方法的完整实现，并在一系列一致的示例程序中给出了这些操作的描述。由于我们使用了实际代码，而不是伪代码，因而在实际中可以很快地使用这些程序。通过访问本书的主页可以得到程序的代码清单。

实际上，书中算法的一个实际应用会产生数百幅图表。正是这些图表提供的立体视觉直观地发现了许多算法。

本书详细讨论了算法的特征以及它们可能应用的场合。尽管并不强调，但是书中论述了算法分析与理论计算机科学的联系。在适当的时候，书中都给出了经验性的数据和分析结果用以说明为什么选择使用某些算法。如果有趣，书中还会描述所讨论的实际算法与纯理论结果之间的关系。关于算法性能特征和实现的特定信息的综合、概括和讨论都会贯穿本书的始终。

编程语言

书中所有实现所用的程序设计语言均为 C 语言（本书的 C++ 版本和 Java 正在开发）。任何特定语言都有优缺点；我们使用 C 语言是因为它是一种广泛使用的语言，并且能够为本书的实现提供所需的特征。由于没有多少结构是 C 所特有的，因而用 C 编写的程序可以很容易地变成用其他现代编程语言编写的程序。在适当的时候，我们会使用标准 C 中的术语，但本书并不打算成为 C 程序设计的参考手册。

我们力争编写优雅、简明和可移植的代码实现，但同时关注实现的效率，因而我们在开发的各个阶段都试图了解代码的性能特征。在这一版中有很多新的程序，老版的很多程序也已更新，主要的目的是使它们在用作抽象数据类型实现时更为实用。对程序所做的广泛的实验性比较研究贯穿在本书中。

本书的目标是以尽可能简单、直接的方式来展示算法。本书使用尽可能一致的风格，因而很多程序看起来是相似的。对于许多算法而言，无论使用哪种语言，算法都具有相似性：例如（选取一个著名的例子）Dijkstra 算法就是 Dijkstra 算法，无论采用 Algol-60、Basic、Fortran、Smalltalk、Ada、Pascal、C、C++ 、Modula-3、PostScript、Java 表示，还是任何一种其他编程语言表示，也不管其所在的环境，它都被证明是一种有效的图处理算法。

有关练习的说明

给练习分类是一件充满风险的事情，因为本书读者的知识背景和经验参差不齐。虽然如此，还应适当加上指导，所以许多练习都加了一个记号（共有 4 种不同记号），以帮助你判断如何加以解决。

测试你对内容理解程度的练习用空心三角符号标识，如下所示：

▷ 17.2　给出下图的所有连通子图。

0-1　0-2　0-3　1-3　2-3

通常，这样的练习是与正文中的例子直接相关的。它们的难度不大，但是完成这些练习会使你了解某个事实或掌握某个概念，它们可能是你在阅读正文时感到困惑不解的问题。

为正文内容添加新的且需要思考的信息的练习用空心圆符号标识，如下所示：

○ 18.2　在对图 18.2 和 18.3 中所示的迷宫进行 Trémaux 探索时，以下序列中，哪一个不能作为各交叉点开灯的顺序？

0-7-4-5-3-1-6-2
0-2-6-4-3-7-1-5
0-5-3-4-7-1-6-2
0-7-4-6-2-1-3-5

这种练习将鼓励你考虑与正文中内容相关的重要概念，或者回答出现在你阅读正文时遇

到的一个问题。即使你没有时间做这些练习，也会发现阅读这些练习是非常有价值的。

具有挑战性的练习用黑色圆点标识，如下所示：

● 19.2 在线找出一个大型 DAG 图，可以是一个大型软件系统中函数定义依赖关系所定义的 DAG 图，也可以一个大型文件系统的目录连接所定义的 DAG 图。

这种练习可能需要花费大量时间才能完成，这取决于你的经验。一般而言，最有效的方法是分几个阶段来解决。

少数难度极大的练习（相对于其他大多数练习）用两个（甚至三个）黑色圆点标识，如下所示：

●● 20.35 开发一个用于生成 V 个顶点、E 条边的随机图的合理程序，使得 Prim 算法的 PFS 实现（程序 20.4）的运行时间是非线性的。

这种练习类似于研究文献上所讨论的问题，但本书中的内容可能会让你做好准备，乐于尝试解决这些问题（而且很可能成功）。

对于考察你的程序设计能力和数学能力的练习，则没有明确记号。这些要求程序设计能力或数学分析能力的练习是一种自我检查。我们鼓励所有的读者都通过实现算法来检测对算法的理解程度。

致 谢

对于本书早期的版本，很多人提供了有用的反馈信息。特别要提出的是普林斯顿大学和布朗大学成百上千的学生们在过去的多年里一直承受着初稿的粗糙。特别要感谢 Trina Avery 和 Tom Freeman 对于第一版的出版所给与的帮助；还要感谢 Janet Incerpi，是她的创造力和聪明才智使我们使用早期原始的数字排版硬件和软件来制作本书的第一版；要感谢 Marc Brown，他参与了算法可视化研究，而本书中的很多图表正来源于此；感谢 Dava Hanson，对于我提出的关于 C 语言方面的所有问题，他总是乐于回答。感谢 Kevin Wayne，它耐心回答了我关于网的基本问题。我还要感谢众多读者，他们对各个版本提供了详尽的评论，这其中包括：Guy Almes、Jon Bentley、Marc Brown、Jay Gischer、Allan Heydon、Kennedy Lemke、Udi Manber、Dana Richards、John Reif、M. Rosenfeld、Stephen Seidman、Michael Quinn 和 William Ward。

为了完成这一新版本，我有幸与 Addison-Wesley 的 Peter Gordon 和 Helen Goldstein 一起工作。他们耐心地指导这个项目的完成，使其经历了从一般性修改到大幅度重写的过程。同样我还有幸与 Addison-Wesley 的许多专业人员一起工作。这个项目的性质使得本书带给他们非同寻常的挑战，对于他们的忍耐力我致以诚挚的感谢。

在写这本书的过程中，我得到了两位良师益友。在此我要特别向他们表示感谢。首先，Steve Summit 从技术的角度对各版本的初稿都做了仔细的检查，并向我提出了数千条详尽的意见，尤其是关于程序方面的建议。Steve 很清楚我的目标是要提供优雅、有效且实用的实现代码，他的意见不仅帮助我给出实现一致性的度量标准，而且还帮助我对其中许多实现做了重大的改进。其次是 Lyn Dupre，他也对我的手稿提出了数千条的意见，这些建议对我来说是无价之宝，不仅帮助我改正和避免了许多语法错误，而且更重要的是，使我找到一种一致性的写作风格，由此才使我把如此繁多的技术资料整理在一起。能够有机会向 Steve 和 Lyn 学习，我万分感激。他们的投入对于本书的完成至关重要。

我在这里所写的内容大多受益于 Don Knuth 的授课和著作，他是我在斯坦福大学的导

师。尽管 Don 对本书没有直接的影响，但在本书中仍然能够感受到他的存在，因为正是他为算法研究奠定了科学基础，才使像本书这样的工作得以完成。我的朋友兼同事 Philippe Flajolet 是使算法分析发展成为一个成熟领域的主力，他对这本书具有同样的影响力。

非常感谢普林斯顿大学、布朗大学以及法国国立计算机与自动化研究所（Institute National de Recherce en Informatique et Automatique，INRIA）给予的支持，在这些地方，我完成了本书大部分的工作；还要感谢美国国防部防御分析研究所（Institute for Defense Analysis）以及施乐的帕洛阿尔托研究中心（Xeror Palo Alto Research Center），我在他们那里访问期间完成了本书的一些工作。本书的某些部分离不开国家自然科学基金（National Science Foundation）和海军研究中心（Office of Naval Research）的慷慨支持。最后，我要感谢 Bill Bowen、Aaron Lemonick 和 Neil Rudenstine，感谢他们对建立普林斯顿大学学术环境的支持，使我能够在这样良好的环境中，在承担众多其他事务的同时完成本书。

Robert Sedgewick

Marly-le-Roi，法国，1983 年 2 月

普林斯顿，新泽西州，1990 年 1 月

詹姆斯镇，罗得岛，2001 年 5 月

目　录

出版者的话
译者序
中文版序
前　言

第五部分　图　算　法

第 17 章　图的性质及类型 …… 1
17.1　术语 …… 3
17.2　图的 ADT …… 9
17.3　邻接矩阵表示 …… 12
17.4　邻接表表示 …… 15
17.5　变量、扩展和开销 …… 18
17.6　图生成器 …… 23
17.7　简单路径、欧拉路径和哈密顿路径 …… 30
17.8　图处理问题 …… 40
第 18 章　图搜索 …… 47
18.1　探索迷宫 …… 47
18.2　深度优先搜索 …… 50
18.3　图搜索 ADT 函数 …… 54
18.4　DFS 森林的性质 …… 57
18.5　DFS 算法 …… 62
18.6　可分离性和双连通性 …… 66
18.7　广度优先搜索 …… 72
18.8　广义图搜索 …… 79
18.9　图算法分析 …… 85
第 19 章　有向图和有向无环图 …… 90
19.1　术语和游戏规则 …… 91
19.2　有向图中的 DFS 剖析 …… 98
19.3　可达性和传递闭包 …… 103
19.4　等价关系和偏序 …… 112
19.5　有向无环图 …… 114
19.6　拓扑排序 …… 117
19.7　有向无环图中的可达性 …… 124
19.8　有向图中的强连通分量 …… 126
19.9　再论传递闭包 …… 133
19.10　展望 …… 136
第 20 章　最小生成树 …… 139
20.1　表示 …… 141
20.2　MST 算法的基本原理 …… 144
20.3　Prim 算法和优先级优先搜索 …… 149
20.4　Kruskal 算法 …… 156
20.5　Boruvka 算法 …… 160
20.6　比较与改进 …… 163
20.7　欧几里得 MST …… 167
第 21 章　最短路径 …… 169
21.1　基本原理 …… 174
21.2　Dijkstra 算法 …… 177
21.3　所有对最短路径 …… 185
21.4　无环网中的最短路径 …… 191
21.5　欧几里得网 …… 197
21.6　归约 …… 201
21.7　负权值 …… 211
21.8　展望 …… 223
第 22 章　网络流 …… 225
22.1　流网络 …… 228
22.2　增大路径最大流算法 …… 236
22.3　预流 - 推进最大流算法 …… 254
22.4　最大流归约 …… 264
22.5　最小成本流 …… 275
22.6　网络单纯形算法 …… 282
22.7　最小成本流归约 …… 293
22.8　展望 …… 299
第五部分参考文献 …… 302

第五部分 图 算 法

第 17 章 图的性质及类型

实际应用中的许多计算问题不仅仅涉及元素（item）的集合，还会涉及元素之间连接（connection）的集合。这些连接所蕴含的关系很自然地引出大量的问题：沿着连接是否存在从一个元素到另一个元素的路径？从某一给定元素可以到达多少个其他元素？从这个元素到另一个元素的最佳路径是什么？

为了建立这些情况的模型，我们使用一种称为图（graph）的抽象对象。在这一章里，我们会详细考察图的基本性质，为学习多种有用算法打下基础，这些算法可用于回答前面所提出的那些问题。这些算法充分利用了我们在本套书第 1 ~ 4 部分所讨论的计算工具。另外它们还可以作为解决重要应用问题的基础，因为没有好的算法技术，我们甚至无法考虑这些问题的解决方案。

图论是组合数学的一个主要分支。它在数百年来已得到深入研究。很多重要且有用的图的性质已经得到证明，但仍有大量难题未被解决。尽管仍有很多内容需要学习，但是在本书中我们将只从有关图的众多知识中抽取我们需要理解的知识，并使用大量有用且基础的算法。

与我们研究的其他很多问题领域类似，图的算法学的研究相对较新。尽管一些基本算法非常古老，但多数算法都是在最近几十年发现的。即使是最简单的图算法，也会得出很有用的计算机程序，而我们考察的非平凡算法是已知的最为优雅和有趣的算法。

为了阐明涉及图处理的应用的多样性，我们从几个例子来探索这个内容丰富的领域中的算法。

地图 某人计划一次旅行，他需要回答诸如这样的几个问题："如何用最小花费（least expensive）从普林斯顿到达圣何塞？"。如果一个人更加关注时间而不是金钱，他可能会问："如何用最短时间（fastest）从普林斯顿到达圣何塞？"。为了回答这些问题，我们需要处理元素（城镇）之间的连接（旅行路线）信息。

超文本 当我们浏览 Web 时，会遇到一些文档，其中包含着引用其他文档的链接（link）。通过单击这些链接，我们就从一个文档转到另一个文档，整个 Web 就是一个图，其中元素就是文档，连接是链接。图处理算法是搜索引擎的基本组成部分，可以帮助我们在 Web 上定位信息。

电路 电路中包括诸如晶体管、电阻器和电容等元件，它们复杂地连接在一起。我们使用计算机来控制组成电路的机器，检查电路是否执行所需的功能。我们要回答一些简单的问题，"这是一条较短的路径吗？"，复杂的问题有："我们是否可以把这个电路布在一个芯片上以使这些连线互不交叉？"。在这种情况下，对第一个问题的回答只取决于连接（导线）的性质，而对第二个问题的回答则需要关于导线、这些导线连接的元素以及芯片的物理约束

条件等详细信息。

调度　制造过程中需要执行大量满足一些约束条件的任务，这些条件规定某个任务必须在其他任务完成之后才能开始执行。我们把这些约束条件看作任务（元素）之间的连接，这样就得到一个经典的调度（scheduling）问题：如何调度任务，才能既满足约束条件，又能在最短时间内完成整个过程。

事务　某电话公司维护一个电话呼叫通信量的数据库。这里，连接表示为电话呼叫。我们对互连结构的特点很感兴趣，因为我们希望布线及设置开关使这个结构能够有效地处理通信量。另一个例子是金融机构跟踪市场中的买/卖订单状况。这种情况下连接表示两个客户之间的现金转移。在这个例子中有关连接结构特点的知识可以增强我们对于市场情况的了解。

匹配　学生们要在诸如社会团体、大学或医院等选择性的机构申请职位。这里的元素对应着学生和机构；连接对应着申请。我们希望找到一些方法使感兴趣的学生与某些可用职位相匹配。

网络　计算机网络由互联的站点组成，这些站点可以发送、转发和接收各种类型的消息。我们不仅关注可以得到从一个站点到其他站点的消息，同时关注在网络变化时维持站点之间的连接关系。例如，我们可能希望检查某个给定的网络，以确信其中没有一小部分站点或连接对网络是如此关键，以至于一旦它们出现问题将会造成其余站点之间无法连接。

程序结构　编译程序用构建图的方式来表示一个大型软件系统的调用结构。这里的元素是各种函数或组成系统的模块；连接要么关联一个函数调用另一个函数的可能性（静态分析），要么关联正在运行的系统的实际调用（动态分析）。我们需要分析这个图以确定如何最好地为程序分配资源以达到高效。

这些例子表明，在很多应用中，图是最合适的抽象模型，也表明了我们使用图时所遇到的计算问题。这样的问题将是我们本书的重点。在实际中遇到上述很多应用时，所涉及的数据量非常巨大，高效的算法则会影响到一个解决方案是否可行。

在这套书的第 1 部分我们已经遇到了图。实际上，我们在第 1 章详细介绍的第一个算法是合并 - 查找算法，它是图算法的一个基本例子。我们在第 3 章还使用图来说明二维数组和链表的应用。在第 5 章中用图说明了递归程序和基本数据结构之间的关系。任何链式数据结构都是一种图的表示，处理树和其他一些链表结构的常用算法都是图算法的特例。本章的目的是为理解所有图算法提供一种环境，包括从第 1 部分的简单算法到第 18 章至第 22 章的复杂算法。

像往常一样，我们很想知道解决某个特定问题使用哪种算法最为有效。研究图算法的性能特征具有很大的挑战性，这是因为：

- 算法的开销不仅依赖于元素集性质，而且依赖于连接集的很多性质（以及连接关系中所蕴含的图的全局性质）。
- 我们可能遇到的各类图的精确模型是难以建立的。

我们通常假设图算法的最坏情况下的性能界限，即使某些情况下的实际性能评价可能更高。幸运的是，正如我们将要看到的，大量的算法是*最优的*（optimal），并涉及极少不必要的工作。对于所有给定规模的图，其他一些算法消耗同量的资源。我们可以精确地预测这些算法在特定情况下是如何执行的。不能做出这样精确的预测时，我们需要特别关注各种图的性质，以便对实际情况做出估计，而且必须对这些性质对于算法性能的影响做出评估。

我们首先讨论图的定义和性质，考察用于描述它们的标准术语。接着将定义基本 ADT

（抽象数据类型）接口，使用这些接口我们可以研究图算法，还将定义两个表示图的最重要的数据结构，它们是邻接矩阵（adjacency matrix）表示和邻接表（adjacency list）表示。还要研究实现基本 ADT 函数的各种方法。然后，我们考察可以生成随机图的客户程序，可以使用这些程序来测试我们的算法和学习图的性质。所有这些材料为我们引入图处理算法提供了基础，这些图处理算法可用于求解与在图中找路径相关的三个经典问题。它说明了图问题的难度有很大的差别，即使这些问题看上去类似。我们在本章最后对本书中将要考察的最重要的图处理问题进行了综述，并根据它们的解决难度进行了排列。

17.1　术语

有大量与图有关的术语。大多数术语有简明的定义。为了便于参考，这里集中介绍这些术语。我们在第 I 部分讨论基本算法时已经使用过一些概念；而另一些概念在第 18 章至第 22 章讨论高级算法时才会涉及。

定义 17.1　图（graph）由一个顶点（vertex）集和一个边（edge）集组成，其中边将一对不同顶点连接在一起（每对顶点之间至多有一条边连接）。

在 V 个顶点的图中，我们用 0 到 $V-1$ 作为各顶点的名字。选择这种命名方法的主要原因是我们可以使用数组的下标快速地访问每个顶点对应的信息。在 17.6 节中，我们考虑使用符号表的一个程序，它建立了 V 个顶点名与 $0 \sim N-1$ 之间的 V 个整数的一一（1-1）对应。有了这个程序，我们可以不失一般性地使用下标作为顶点名（为表示方便）。有时假设隐含地定义了顶点集，通过用边集来定义图，且只考虑那些至少在一条边上包含的顶点。为了避免诸如“带有如下边集的 10 顶点图”这样的累赘用法，当顶点个数可以从上下文中明显看出时，我们并不会明确地指出顶点个数。根据约定，我们总是用 V 表示一个给定的图中的顶点数，用 E 表示其中的边数。

我们将图的这一定义（在第 5 章中遇到过）作为标准定义，但在技术上有两点简化。首先，该定义不允许有重复边（数学上把这样的边称为平行边（parallel edge），包含这样边的图称为多重图（multigraph）。其次，该定义不允许出现顶点连接到自身的边；这样的边称为自环（self-loop）。有时把没有平行边或自环的图称为简单图（simple graph）。

在形式定义中我们使用简单图，因为这样更容易表达它们的基本性质，而且在很多应用中不需要平行边或自环。例如，在一个给定顶点数的简单图中，可以限定它的边数。

性质 17.1　有 V 个顶点的图至多有 $V(V-1)/2$ 条边。

证明　在 V^2 个可能的顶点对中包含 V 个自环，而对于不同顶点对的每条边都计算了两次，因此，边数至多为 $(V^2-V)/2 = V(V-1)/2$。 ■

如果允许平行边，这个限制就不能成立。因为如果一个图不是简单图，那么它也可能由两个顶点及连接它们的数亿条边组成（即使图中只有一个顶点，也可能有数亿个自环）。

对于某些应用，我们可能把消除平行边和自环看作实现中必须强调的数据处理问题。不论在本书的什么地方，只要为方便起见使用包含平行边或自环的扩展定义来考虑一个应用或是研制一个算法时就要这样做。例如，在 17.4 节讨论的一个经典算法中，自环起着关键的作用；而平行边在第 22 章所讨论的应用中也相当常见。一般来讲，从上下文就能知道术语“图”是指“简单图”、“多重图”还是“带有自环的多重图”。

数学家们往往不加区分地交换使用顶点（vertex）和结点（node）这两个词，不过我们一般在讨论图时使用顶点，在讨论表示时使用结点。例如，在 C 数据结构中就使用结点。我们一般假设顶点都有一个名字，可以携带其他有关信息。类似地，数学家们也用弧（arc）、

边（edge）和链接（link）等词来描述两个顶点之间连接的抽象，但我们统一在讨论图时使用边（edge），讨论 C 数据结构时使用链接（link）。

如果一条边连接两个顶点，就称这两个顶点是相互邻接的（adjacent），这条边依附于（incident）这两个顶点。顶点的度（degree）就是依附于它的边数。我们使用 *v-w* 记法表示连接 *v* 和 *w* 的一条边；用 *w-v* 表示同一条边的另一种记法。

子图（subgraph）是组成图的边（及相关顶点）的一个子集。很多计算任务都涉及识别不同类型的子图。如果要识别一个图中顶点的子集，则称该子集以及连接该子集中某两个成员的所有边为与这些顶点关联的导出子图（induced subgraph）。

通过将顶点标记为点，并用线连接这些点可以画出一个图。绘图能够带给我们关于图的结构的直观认识；但是这种直观性可能产生误导，因为图的定义与表示无关。例如，图 17-1 中的两种画法以及边的列表表示同一个图，因为图只包含其顶点集和边（顶点对）集，而且顶点集和边集均为无序的，除此以外再无其他。虽然将图看作边集已足够，但是我们还将讨论其他的一些表示，它们特别适合于作为 17.4 节的图数据结构的基础。

将一个给定图的顶点放在平面上，并画出这些点和连接它们的边称为绘图（graph drawing）。顶点的可能放置方式、边的绘制风格以及绘制中的美学约束不计其数。遵从各种自然约束的绘图算法已经得到深入研究，而且已有很多成功的应用（见第五部分参考文献）。例如，最简单的约束是要求边不相交。平面图（planar graph）就是能在平面上画出的边不相交的图。确定一个图是否是平面的是一个很吸引人的算法学问题，我们在 17.8 节中将会简略讨论。能够产生一种有助于可视化的表示也是一个很好的目标，另外绘图是一个很吸引人的研究领域，但是成功的绘图往往很难实现。顶点数和边数都有很多的图是抽象的对象，要合适地绘制它们不太可行。

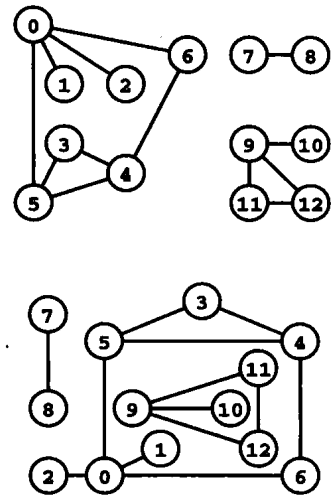


图 17-1　一个图的三种不同表示

图是由它的顶点和边来定义的，而不是由所选择的绘图方法来定义。这两个绘图方式表示了同一个图，下面的边列表也同样表示此图。给定图中有 13 个顶点，编号从 0 到 12。

对于某些应用，例如表示地图或电路的图，所绘出的图可以带有很丰富的信息，因为顶点即为平面上的点，而且它们之间的距离是相关的。我们将这种图称为欧几里得图（Euclidean graph）。对于其他一些应用，例如表示关系或调度的图，图中只含有连通性信息，顶点的几何位置的特定信息并未提供。我们在第 20 章和第 21 章的欧几里得图中所讨论的算法将会探讨几何信息，但我们的主要工作放在不使用任何几何信息的算法上，并且强调图一般来说独立于绘制或计算机中的任何特定的表示。

如果从只关注连通性自身考虑，我们可能希望顶点编号只是方便使用的标号，而且对于两个图，如果能够改变一个图的编号，使得其边集等于另一个，则认为它们是同构的（isomorphic）。确定两个图是否同构的问题是一个困难的计算问题（见图 17-2 和练习 17.5），它很有挑战性，因为对顶点编号有 *V*! 种可能的方式，要穷尽所有可能性不太可能。因此，尽管通过将同构图处理为相同的结构来减少所考虑的图结构种类的做法很诱人，但我们很少这样做。

正如在第 5 章所看到的，我们常常关注基本的结构性质，这些性质可通过图中边的特定序列推导出。

定义 17.2　图中的一条路径（path）是顶点的一个序列，其中每个后继顶点（除了第一个顶点之外）与路径中其前驱顶点相邻接。在简单路径（simple path）中，顶点和边都不相同。环（cycle）是第一个顶点和最后一个顶点相同的一条简单路径。

我们有时使用术语回路（cyclic path）来表示第一个顶点和最后一个顶点相同的一条路径（不必是简单路径）；使用术语周游路径（tour）表示包含图中每个顶点的回路。定义一条路径的等价方式是连接连续顶点的边的序列。我们在表示中也强调了这一点，连接路径中的顶点名的方式与我们在边中连接它们的方式相同。例如，图 17-1 中的简单路径有 3-4-6-0-2 和 9-12-11，图中的环有 0-6-4-3-5-0 和 5-4-3-5。我们将路径或环的长度（length）定义为其边的数目。

我们采用以下约定：单个顶点是一条长为 0 的路径（从顶点到自身且不含边的路径，不同于自环）。除了这个约定之外，还约定图中不含平行边且没有自环。一对顶点唯一地确定了一条边。路径上必须至少有两个不同的顶点，环上必须至少有三条不同的边和三个不同的顶点。

如果两条简单路径除了它们的端点之外没有公共的顶点，则称它们是不相交的（disjoint）。相对于要求路径中根本没有公共顶点，这个条件要弱一些。这样定义很有用，因为如果 s 和 u 是不同的，我们可以将从 s 到 t 和从 t 到 u 的简单不相交路径组合起来，来得到一条从 s 到 u 的简单不相交路径（如果 s 和 u 相同，则得到一个环）。有时还会用到另一个术语顶点不相交（vertex disjoint），以区别更强的条件边不相交（edge disjoint），此时要求路径中没有公共边。

定义 17.3　如果图中每个顶点到另一个顶点都存在路径，则称该图是一个连通图（connected graph）。不连通的图由连通分量（connected component）组成，这些连通分量是最大连通子图。

术语最大连通子图（maximal connected subgraph）意味着不存在从该子图中的顶点到图中其他顶点的路径。直觉上看，如果顶点是物理对象，如结或珠子，边是物理连接，如绳子或线，如果拿起任何一个顶点，连通图都是一个整体。不连通的图由两个或多个这样的部分组成。

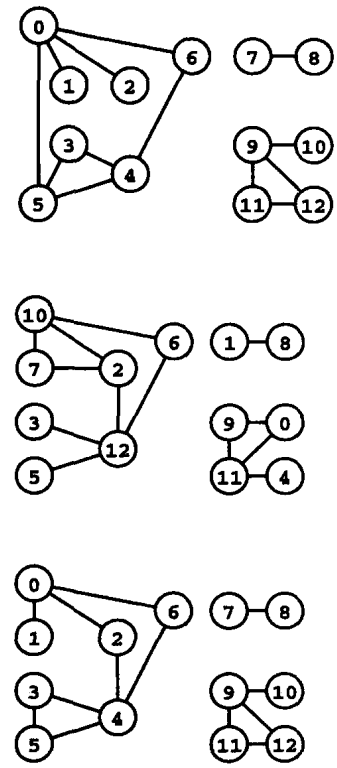


图 17-2　图同构示例

上面的两个图是同构的，因为我们可以对顶点重新编号，使两个边集相等（为了使中间的图与上图一样，将 10 变成 4，7 变成 3，2 变成 5，3 变成 1，12 变成 0，5 变成 2，9 变成 11，0 变成 12，11 变成 9，1 变成 7，4 变成 10）。下图与上图两个图不同构，因为无法对顶点重新编号以使其边集等于另两个边集。

定义 17.4　一个无环连通图成为树（tree，见第 4 章）。树的集合称为森林（forest）。连通图的生成树（spanning tree）是一个子图，它包含了此图中的所有顶点，且是一棵树。图的生成森林（spanning forest）是一个子图，它包含了此图中的所有顶点，且是一个森林。

例如，图 17-1 中描述的图有三个连通分量，可以得到生成森林 7-8　9-10　9-11　9-12　0-1　0-2　0-5　5-3　5-4　4-6（还有很多其他生成森林）。图 17-3 中的大图中突出显示了这些森林以及其他一些特征。

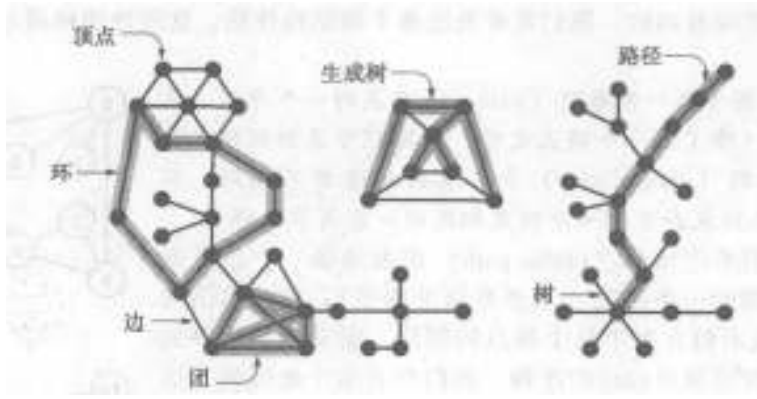


图 17-3　图的术语

此图中有 55 个顶点，70 条边，3 个连通分量。其中一个连通分量是一棵树（右图）。图中有多个环，在大连通图（左图）中突出显示了其中的一个环。在小的连通分量（中图）中也显示了一个生成树。此图自身并不包含一个生成树，因为它不是连通的。

我们在第 4 章研究了关于树的一些细节，并介绍了各种等价定义。例如，有 V 个顶点的图 G 是一棵树，当且仅当它满足以下 4 个条件之一：

- G 有 $V-1$ 条边，且没有环。
- G 有 $V-1$ 条边，且是连通的。
- 只有一条简单路径连接 G 中的每对顶点。
- G 是连通的，但删除任何一条边会使它不连通。

上述的任何一个条件都是证明另一个的充分必要条件。我们可以由此得出其他关于树的一些结论（见练习 17.1）。形式上，我们应该选择一个条件作为定义；非形式上，可将它们组合起来作为定义，并在定义 17.4 中选择使用“无环连通图”。

所有边都存在的图称为完全图（complete graph，见图 17-4）。从一个与原图 G 有相同顶点集的完全图删除 G 的边得到的图定义为图 G 的补（complement）图。两个图的并（union）是由其边集的并所导出的。有 V 个顶点的所有图是有 V 个顶点的完全图的子图。有 V 个顶点的不同图的总数为 $2^{V(V-1)/2}$（也即从 $V(V-1)/2$ 条可能边选择一个子集的不同方式）。一个完全子图被称为一个团（clique）。

我们在实际中遇到的大多数图只有相对少的边出现。为了量化这个概念，我们定义图的稠密度（density）为平均顶点度，或 $2E/V$。稠密图（dense graph）是一个其平均顶点度与 V 成正比的图；稀疏图（sparse graph）是一个其补图为稠密图的图。换句话说，如果 E 与 V^2 成正比，则将一个图看作稠密的，否则看作是稀疏的。对于某个特定的图，这种近似定义不一定有意义，但区别是显然的：有数百万个顶点和数千条边的图肯定是一个稀疏图，有数千个顶点和数百万条边的图肯定是稠密图。我们可能考虑去处理一个有数十亿个顶点的稀疏图，但是有数十亿个顶点的稠密图中的边数却多得不得了。

了解是否一个图是稀疏还是稠密图是选择处理图的有效算法的关键因素。例如，对于给定的一个问题，我们可能开发一个需要约 V^2 步的算法，也可能需要一个 $E\lg E$ 步的算法。这些公式告诉我们，第一个算法适合于稀疏图，第二个算法更适合于稠密图。例如，对于有数百万条边的一个稠密图可能只有数千个顶点：在这种情况下，V^2 和 E 的大小具有可比性。V^2 算法要比 $E\lg E$ 算法快 20 倍。另一方面，有数百万条边的稀疏图也有数百万个顶点，因而 $E\lg E$ 算法要比 V^2 算法快数百万倍。我们可以仔细分析这些算法的公式来做出特定的权衡。

但在实际中使用术语稀疏和稠密一般就足以帮助我们理解算法的基本性能特征。

在分析算法时，我们假设 V/E 有一个小的常数界，因而我们可以将诸如 $V(V+E)$ 这样的表达式简化为 VE。这一假设在边数与顶点数相比微小时才成立，这是一种少有的情形。典型地，边数远超过顶点数（V/E 远小于 1）。

二分图（bipartite graph）是一个其顶点可划分成两个集合的图，所有边连接的顶点均处于这两个不同的集合中。图 17-5 给出了二分图的一个例子。二分图很自然地出现在很多情形中，如本章开始描述的匹配问题。二分图的任何子图是一个二分图。

这样定义的图称为无向图（undirected graph）。在有向图（ditected graph）中，也这样称 dirgraph。边是单向的：我们将定义每条边的顶点对看作是有序对，它指定了一个单向的邻接关系。因此，我们可以从第一个顶点到达第二个顶点，但不能从第二个顶点到达第一个顶点。很多应用（例如，表示 Web、调度约束或电话呼叫业务）很自然地用有向图表示。

我们将有向图中的边称为有向边（directed edge），尽管根据上下文很容易区分（有些作者保留术语弧（arc）来称有向边）。有向边中的第一个顶点称为源点（source）；第二个顶点称为目的点（destination）。（有些作者分别称为头（head）和尾（tail）以区别有向边中的顶点。我们不使用这种用法，因为这会与数据结构实现中使用的相同术语重复。）我们画一条从源点指向目的点的箭头表示有向边。当在有向图中使用表示 v-w 时，它的意思是代表一条从 v 指向 w 的边；它不同于 w-v，后者表示从 w 指向 v 的边；我们还谈到一个顶点的入度（indegree）和出度（outdegree）（分别表示以该顶点为目的点的边数，以及以该顶点为源点的边数）。

有时候，将一个无向图看作有向图也是合理的，其中每条边有两个方向（每个方向一条边）；其他时候，根据连接来考虑无向图是很有用的。正如在 17.4 节中将详细讨论的，通常我们对于有向图和无向图使用同一表示法（见图 17-6）。也就是说，我们一般会为无向图的每条边维持两种表示，分别指向不同方向，因而我们可以直接回答哪些顶点与顶点 v 连接的问题。

第 19 章主要探索有向图的结构性质；它们一般要比无向图中对应的性质更为复杂。有向图中的一个有向环（directed cycle）是一个其所有邻接顶点对按照（有向的）图边所指明的顺序出现的环。有向无环图（directed acyclic graph，DAG）是一个不含有向环的图。DAG（有向无环图）和一棵树（无环无向图）不同。有时，我们会谈到一个有向图的潜在无向图（underlying undirected graph），含义是同一组边集所定义的无向图，但是这些边不解释为有向的。

第 20 ~ 22 章大致论述了求解各种与图有关的计算问题的算法，在这些图中，顶点和边还关联其他信息。在加权图（weighted graph）中，我们为每条边关联数（权值，weight），一般表示距离或开销。还可以为每个顶点（或顶点和边）关联（多个）权值。第 20 章研究加权无向图；第 21 ~ 22 章我们研究加权图，也称它们为

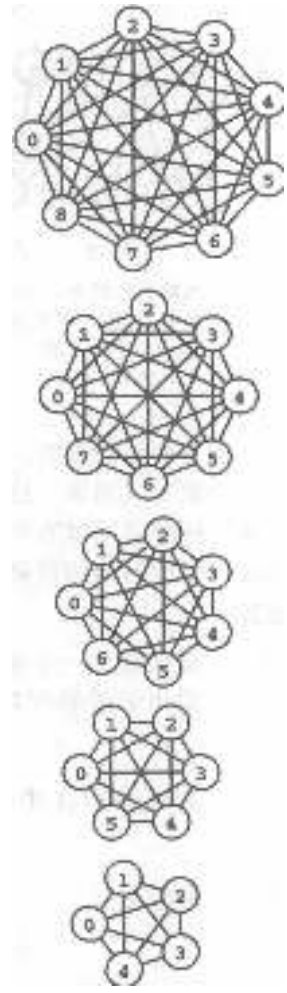


图 17-4　完全图

这些完全图，其中每个顶点都与其他顶点连接。分别有 10、15、21、28 和 36 条边（从下图到上图）。每个顶点数介于 5 个到 9 个之间的图都是这些图的一个子图（这样的图不少于 680 亿个）。

网（network）。第 22 章中的算法可以解决经典问题，这些问题都是从网的一种特定解释（称之为流网络，flow network）产生的。

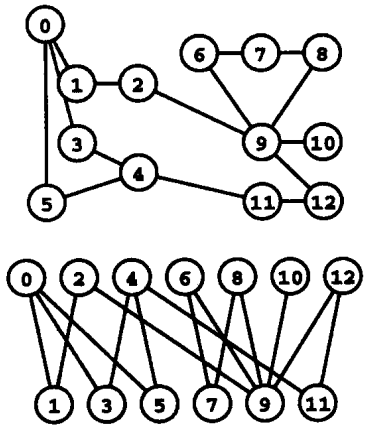


图 17-5　二分图

此图中的所有边将奇数编号的顶点与偶数编号的顶点连接起来。因而它是一个二分图。下图的这个性质更加显著。

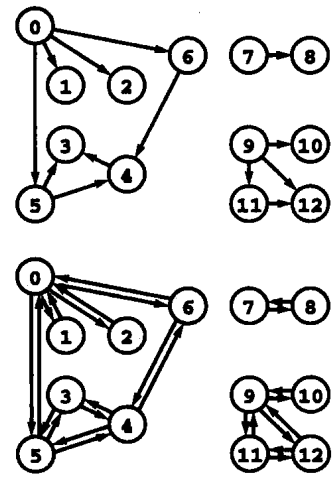


图 17-6　两个有向图

上图是图 17-1 中的示例图的一种表示，解释为有向图，其中的边是有序对，并画从第一个顶点到第二个顶点的一个箭头来表示。它也是一个 DAG。下图是图 17-1 中示例图的一种无向图表示，指明了我们通常表示无向图的方法：作为有向图，对应每个连接有两条边。（每个方向各一条）。

第 1 章已经表明，图的组合结构是扩展的。这种结构的扩展性更为显著，因为它由一个简单抽象发展而来。这种潜在的简洁性在我们开发的很多基本图处理代码中都能反映出来。然而，这种简洁性有时会掩盖复杂的动态性质，这些性质需要深入理解图自身的组合性质。比起代码所给出的简洁性质，图算法能够按照需要工作，常常使我们自己很难相信。

练习

17.1　证明任何一个有 V 个顶点的无环连通图都有 $V-1$ 条边。

▷ **17.2**　给出下图的所有连通子图。

```
0-1  0-2  0-3  1-3  2-3
```

▷ **17.3**　列出图 17-1 中图的所有非同构环。例如，如果列表中包含环 3-4-5-3，它应该不含

3-5-4-3，4-5-3-4，4-3-5-4，5-3-4-5，或 5-4-3-5。

17.4　考虑图

```
3-7  1-4  7-8  0-5  5-2  3-8  2-9  0-6  4-9  2-6  6-4
```

确定连通分量的数目，给出一个生成森林，列出至少含有三个顶点的所有简单路径。并列出所有非同构环（见练习 17.3）。

○ **17.5**　考虑以下四个边集所定义的图：

```
0-1  0-2  0-3  1-3  1-4  2-5  2-9  3-6  4-7  4-8  5-8  5-9  6-7  6-9  7-8
0-1  0-2  0-3  0-3  1-4  2-5  2-9  3-6  4-7  4-8  5-8  5-9  6-7  6-9  7-8
0-1  1-2  1-3  0-3  0-4  2-5  2-9  3-6  4-7  4-8  5-8  5-9  6-7  6-9  7-8
4-1  7-9  6-2  7-3  5-0  0-2  0-8  1-6  3-9  6-3  2-8  1-5  9-8  4-5  4-7
```

上述哪些图是相互同构的？哪些是平面图？

17.6　考虑图 17-4 中标题所提到的 680 亿多个图。其中顶点数少于 9 个的图占多少？

▷ 17.7　在给定的 V 个顶点、E 条边的图中，有多少个不同子图？

● 17.8　给出有 V 个顶点、E 条边的图的连通分量数目的上界和下界。

○ 17.9　对于有 V 个顶点和 E 条边的图，有多少个不同的无向图？

●●● 17.10　如果认为不同构的两个图是不同的，那么有多少个 V 个顶点和 E 条边的不同的图？

17.11　V 个顶点的二分图有多少？

17.2　图的 ADT

我们使用定义了所关注任务的 ADT 来开发图处理算法，并使用第 4 章中考虑的标准机制。程序 17.1 是我们用于此目的的 ADT 核心接口。这个 ADT 的基本图表示和实现是 17.3 ~ 17.5 节的主题。在本书后面，只要考虑一个新的图处理问题，就会考虑求解这个问题的算法及其在这个接口中的新的 ADT 函数的实现问题。这种模式使得我们解决图处理任务的范围涵养从基本的维护函数到困难问题的复杂解决方案。

该接口是根据隐含在客户端程序（见 4.8 节）中的表示和实现的标准机制。它还包含了一个简单结构类型定义，使得我们的程序可以以一种统一的方式维持对边的管理。这个接口提供了允许客户构建图的基本机制（首先初始化，然后增加边），来维持对于图的管理（删除某些边，并添加其他边），以及对图进行检索（采用边数组形式）。

程序 17.1 中的 ADT 主要是一个我们可以用于开发并测试算法的工具；它不是一个通用的接口。如常，我们使用这个最简单的接口工作，该接口支持我们所考虑的基本图处理操作。定义实际应用中所用的这样一个接口要涉及简洁性、效率和通用性之间的很多权衡。接下来我们考虑几个权衡问题；另外在本书中的实现和应用中还会解决其他的权衡问题。

程序 17.1　图 ADT 接口

这个接口是一个实现和测试图算法的起点。通过本章和接下来的几章，我们会向这个接口添加求解各种图处理问题的函数。在正文中还讨论了围绕设计通用图处理接口来简化代码和其他问题的各种假设。

此接口定义了两种数据类型：简单边数据类型 Edge，它包含了一个构造器函数 EDGE，用来构造两个顶点的一条边 Edge；还有一个图数据类型 Graph，它使用第 4 章中标准的独立于表示的构造来定义。我们用于处理图的基本操作是 ADT 函数，包括创建、复制、销毁；添加和删除边；以及抽取一条边表。

```
typedef struct { int v; int w; } Edge;
Edge EDGE(int, int);

typedef struct graph *Graph;
Graph GRAPHinit(int);
 void GRAPHinsertE(Graph, Edge);
 void GRAPHremoveE(Graph, Edge);
  int GRAPHedges(Edge [], Graph G);
Graph GRAPHcopy(Graph);
 void GRAPHdestroy(Graph);
```

为简洁起见，假设图的表示由整数 V 和 E 组成，分别表示顶点数目和边的数目。因此我们可以直接在 ADT 的实现中调用这些值。在方便的时候，我们做出图表示中变量的其他类似假设，目的是使实现简洁。为方便起见，我们还提供图中的最大可能顶点数，作为 GRA-

PHinit ADT 函数的参数，从而实现可以相应地分配内存空间。我们采用这些约定，只是为了使代码简洁和易读。

更为一般的接口提供了增加和删除顶点及边的能力（还可能包含返回顶点数和边数的函数），对于实现未做假设。这种设计使得 ADT 的实现可以随着图的增大和变小而扩展和收缩其数据结构。我们也可能选择在中间层次的抽象上工作。考虑支持实现中使用的对图的更高级抽象操作的接口设计。我们在考虑了几种具体表示和实现之后，会在 17.5 节重温这一思想。

通用的图 ADT 需要考虑到平行边和自环，因为无法避免一个客户程序调用 GRAPHinsert 来插入一条已在图中存在的边（平行边），也不能避免插入一条两个顶点编号相同的边（自环）。在某些应用中，这些边是不允许的。而在一些应用中这样的边是允许的。在另外的一些应用中可能忽略这样的边。自环处理起来是简单的，但平行边的处理则比较困难，这取决于图的表示。在某种情况下，添加一个*删除平行边* ADT 函数可能是合适的，这样，实现可使平行边集中在一起，客户程序在得到允许时可以删除或者处理平行边。

程序 17.1 包含了一个函数实现，它向客户程序返回图中的边集，放在数组中。图最重要的是其边集，我们常常需要一种用这种形式检索图的方法，不论其内部表示如何。我们甚至可能将*边数组*表示作为 ADT 实现的基础（见练习 17.15）。然而，那种表示并没有为我们提供有效执行基本图处理操作需要的灵活性。

在这本书里，我们一般处理*静态*（static）图，其中图中有固定的 V 个顶点和 E 条边。一般而言，我们通过执行 E 次调用 GRAPHinsertE 来构建图，然后调用某个以图作为参数的 ADT 函数来处理它们，并返回关于那个图的一些信息。*动态*（dynamic）问题（其中将图处理和边与顶点的插入和删除混在一起）带我们走进*在线算法*（online algorithm）（也称之为*动态算法*，dynamic algorithm）的领域，代表着一组不同的挑战。例如，我们在第 1 章中讨论的合并 - 查找问题是一个在线算法的例子。因为我们在插入边时，可以得到图的连通性信息。程序 17.1 中的 ADT 支持*插入边*（insert edge）和*删除边*（delete edge）的操作，从而客户程序可任意使用它们来对图作出改变，但对于某些操作序列，性能会受到影响。例如，合并 - 查找算法仅对那些不使用*删除边*操作（remove edge）的客户程序有效。

ADT 可能还包括一个以边数组作为参数的函数，在初始化图时使用。对于每条边，我们可以通过调用 GRAPHinsert 来实现这个函数（见练习 17.13），或者取决于图的表示，我们可能设计更有效的实现。

我们还可能调用客户端提供的针对图中每条边或顶点的函数，来提供遍历图的函数。对于某些简单问题，使用 GRAPHedge 返回的数组就足够了。然而，大多数的实现确实执行复杂的遍历，揭示出图结构的一些信息，同时实现函数为客户程序提供更高一级的抽象。

在 17.3 节和 17.5 节中，我们主要考察程序 17.1 中的 ADT 函数的经典图表示和实现。这些实现为我们提供了扩展接口使其包含图处理任务的基础，这是接下来的几章中要讨论的内容。

当我们考虑一个新的图处理问题时，要扩展 ADT 使其包含求解问题的实现算法。一般而言，这些任务可分为两类：

- 计算图中某个度量的值。
- 计算图中某个边的子集。

前者的例子有连通分量数目和图中两个给定顶点之间的最短路径长度；后者的例子有生成树和包含给定顶点的最长环。实际上，我们在 17.1 节中的定义的术语直接地引出了大量

的计算问题。

程序 17.2　图处理客户程序示例

该程序以 V 和 E 作为标准输入，产生一个 V 个顶点和 E 条边的随机图，如果图较小则打印，并计算（和打印）连通分量数目。它使用 ADT 函数 GRAPHrand（见程序 17.8）、GRAPHshow（见练习 17.16 和程序 17.5），以及 GRAPHcc（见程序 18.4）。

```
#include <stdio.h>
#include "GRAPH.h"
main(int argc, char *argv[])
  { int V = atoi(argv[1]), E = atoi(argv[2]);
    Graph G = GRAPHrand(V, E);
    if (V < 20)
        GRAPHshow(G);
    else printf("%d vertices, %d edges, ", V, E);
    printf("%d component(s)\n", GRAPHcc(G));
  }
```

程序 17.2 是一个图处理客户程序的例子。它使用了程序 17.1 中的基本 ADT，通过产生随机图（generate random graph）ADT 函数（返回包含给定顶点数和边数的一个随机图（见 17.6 节））以及连通分量（connected component）ADT 函数（返回给定图中连通分量数目，见 18.4 节）来增大。我们使用类似但更复杂的客户程序生成其他类型的图来测试算法，从而了解图的性质。这个基本接口可用于任何图处理应用中。

在开发 ADT 实现中我们面临的第一个决策是使用哪种图表示。我们有三个基本要求。第一，必须能够包含在应用中可能遇到的各种图类型（而且我们也不愿浪费空间）。第二，应该能够有效地构造所需的数据结构。第三，我们希望开发有效的算法来解决图处理问题，而不过度受到表示所强加的限制。这样的要求对于任何我们考虑的领域都是标准的，这里再次强调它们，因为我们将会看到，即使是对最简单的问题，不同表示也会产生巨大的性能差异。

对于大多数的图处理应用，用两种简单的经典表示（即邻接矩阵（adjacency-matrix）或邻接表（adjacency-list）表示）之一就能很好地处理，而这两种经典表示只比边数组表示略显复杂。我们在 17.3 和 17.4 节将会详细讨论这些基于基本数据结构的表示（实际上，我们在第 3 章和第 5 章作为顺序分配和链式分配的应用例子曾讨论过它们）。选择哪一种表示主要取决于图是稠密的还是稀疏的，虽然所完成操作的本质在决定使用哪种表示时也起着重要的作用。

练习

▷ **17.12**　编写一个程序，从标准输入读入边（0 ~ $V-1$ 之间的整数对）来构建一个图。

17.13　给定边数组，编写一个独立于表示的图初始化 ADT 函数，返回一个图。

17.14　编写一个独立于表示的图 ADT 函数，它使用 GRAPHedges 来输出图中所有的边，格式同正文（顶点号用连字符分隔）。

17.15　提供程序 17.1 中的 ADT 函数的一种实现，使用边数组来表示此图。修改 GRAPHinit 使其将最大允许的边数作为它的第二个参数，用于对边数组空间的分配。使用蛮力实现 GRAPHremoveE，它通过扫描数组找出 v-w 或 w-v，然后将所找到的边与数组中的最后一条边交换来删除边 v-w。通过在 GRAPHinsertE 中进行类似的扫描来删除平行边。

17.3　邻接矩阵表示

图的邻接矩阵（adjacency-matrix）表示是一个 $V \times V$ 的布尔数组，如果图中顶点 v 和 w 有一条边相连，则该数组的第 v 行和第 w 列的元素定义为 1，否则定义为 0。程序 17.3 是图 ADT 的一种实现，它使用该矩阵的一个直接表示。该实现维护一个两维整数数组，如果图中 v 和 w 之间有边相连，则元素 a[v][w] 的值设为 1，否则设为 0。在无向图中，每条边实际上是用两个元素表示的：边 v-w 用 a[v][w] 和 a[w][v] 中的 1 表示出，边 w-v 的表示也同样。

如在第 17.2 节中所提到的，在图初始化时，我们一般假设顶点数对客户程序是已知的。对于很多应用，我们可能将顶点数设置为编译时间约束，并使用静态分配数组。但程序 17.3 采用了稍微更一般的方法来动态地为邻接矩阵分配空间。程序 17.4 是 C 中动态分配二维数组的标准方法，即作为数组指针，如图 17-8 中所述。程序 17.4 还包含初始化图的代码，其中将数组中的所有元素初始化为给定值。这个操作所需时间与 V^2 成正比。为简化代码，其中没有包含对于内存不足的错误检查。在使用这个代码之前，加上错误处理是一个严谨的程序设计实践（见练习 17.22）。

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	1	1	0	0	1	1	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	1	1	0	0	0	0	0	0	0
4	0	0	0	1	0	1	1	0	0	0	0	0	0
5	1	0	0	1	1	0	0	0	0	0	0	0	0
6	1	0	0	0	1	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	1	0	0	0	0
8	0	0	0	0	0	0	0	1	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	1	1	1
10	0	0	0	0	0	0	0	0	0	1	0	0	0
11	0	0	0	0	0	0	0	0	0	1	0	0	1
12	0	0	0	0	0	0	0	0	0	1	0	1	0

图 17-7　邻接矩阵图表示

这个矩阵是图 17-1 中描述的图的另一种表示。如果顶点 v 和 w 有一条边相连，那么在矩阵的第 v 行和第 w 列处为 1。此数组关于对角线对称。例如，第 6 行（和第 6 列）表示顶点 6 与顶点 0 和顶点 4 相连。对于某些应用，我们将采用以下约定：每个顶点与自身连接，在主对角线上赋值 1。右上角和左下角的大块都为 0，这是我们为此例指定顶点编号所致，不是图的特征（而它们的确指出图是稀疏的）。

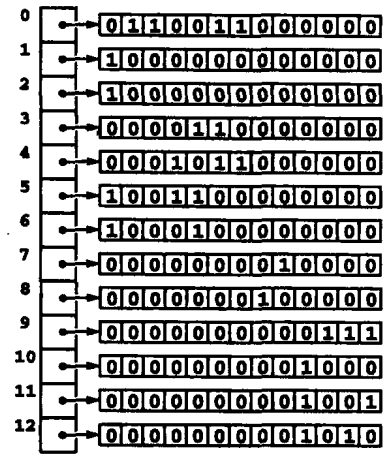


图 17-8　邻接矩阵数据结构

此图描述了图 17-1 中的图的 C 语言表示，表示为数组的数组。

为了添加一条边，我们设置两个指定的数组元素值为 1。并在此表示中不允许平行边存在。如果一条边要被插入到数组元素已是 1 的地方，此代码无效。在某些 ADT 设计中，它可能希望通知客户程序试图插入一条平行边，可能会使用 GRAPHinsertE 的返回码。在这种表示中我们的确允许自环：边 v-v 用 a[v][v] 中的非零元素表示。

为了删除一条边，我们设置两个指定的数组元素的值为 0。如果要删除的边不存在，那么此代码不起作用（对应的数组元素的值已为 0）。而且，在某些 ADT 设计中，我们可能希望能安排通知客户程序这样的条件。

如果正在处理规模很大的图，或是大量规模较小的图，而空间又有限，那么有几种方法可以节省空间。例如，表示无向图的邻接矩阵是对称的：a[v][w] 总是等于 a[w][v]。在 C 语言中，只存储这个对称矩阵的一半就可以简单地节省空间（见练习 17.20）。极端情况

下，我们可能考虑使用位数组（比如在这种情况下，可以用大约 6 400 万 64 位的字表示约 64 000 个顶点的图）（见练习 17.21）。这些实现稍微复杂一些，我们需要添加一个 ADT 操作来测试一条边存在性（见练习 17.19）。（在实现中我们并不使用这样的操作，因为通过简单测试 a[v][w] 来测试一条边存在性的代码要稍微容易理解一些。）这种节省空间的技术是很有效的，但是对于时间很重要的应用，随之而来会带来内循环的额外开销。

程序 17.3　图 ADT 实现（邻接矩阵）

程序 17.1 中的接口实现使用了一个二维数组。在程序 17.4 中给出了函数 MATRIXint 的实现，对数组分配空间并初始化。代码的其余部分非常简单：边 i-j 在图中出现，当且仅当 a[i][j] 和 a[j][i] 都为 1。插入边和删除边的时间为常量。忽略重复边。每次初始化并提取所有边所需时间与 V^2 成正比。

```
#include <stdlib.h>
#include "GRAPH.h"
struct graph { int V; int E; int **adj; }
Graph GRAPHinit(int V)
  { Graph G = malloc(sizeof *G);
    G->V = V; G->E = 0;
    G->adj = MATRIXint(V, V, 0);
    return G;
  }
void GRAPHinsertE(Graph G, Edge e)
  { int v = e.v, w = e.w;
    if (G->adj[v][w] == 0) G->E++;
    G->adj[v][w] = 1;
    G->adj[w][v] = 1;
  }
void GRAPHremoveE(Graph G, Edge e)
  { int v = e.v, w = e.w;
    if (G->adj[v][w] == 1) G->E--;
    G->adj[v][w] = 0;
    G->adj[w][v] = 0;
  }
int GRAPHedges(Edge a[], Graph G)
  { int v, w, E = 0;
    for (v = 0; v < G->V; v++)
      for (w = v+1; w < G->V; w++)
        if (G->adj[v][w] == 1)
          a[E++] = EDGE(v, w);
    return E;
  }
```

很多应用涉及关联每条边的其他信息，在很多情况下，可以将邻接矩阵推广为能存放任何信息的数组。我们至少要保存用于数组元素的数据类型的一个值，来表明指示的边是否存在。在第 20 章和第 21 章，我们探索这些图的表示。

邻接矩阵的使用取决于所关联的顶点名，它是介于 0 和 $V-1$ 之间的整数。这种指派可有多种形式完成，例如，可以考虑 17.6 节中所作的一个程序。因此，在 C 中用二维数组所表示的特定 0-1 矩阵只是给定任何图的一种邻接矩阵表示，因为另一程序对于我们用于确定行和列的下标可能给出顶点名的不同指派。两个看上去非常不同的数组却可能表示同一个图（见练习 17.17）。这一结果是对图同构问题的重新阐述：虽然我们可以确定

两个不同的矩阵是否表示同一个图，但没有人设计出总能这么高效的算法。这个难度是根本性的。例如，我们寻找各种重要的图处理问题有效解的能力完全取决于顶点编号的方法（例如，见练习 17.25）。

程序 17.4　邻接矩阵分配和初始化

此程序使用标准 C 的嵌套数组来表示二维邻接矩阵（见 3.7 节）。分配了 r 行，每行 c 个整数。然后初始化数组中的所有元素的值为 val。程序 17.3 中调用 MATRIXint（V，V，0）创建一个表示 V 个顶点没有边的图，所需时间与 V^2 成正比。例如，对于较小 V，V 次调用 malloc 的开销可以忽略。

```
int **MATRIXint(int r, int c, int val)
  { int i, j;
    int **t = malloc(r * sizeof(int *));
    for (i = 0; i < r; i++)
      t[i] = malloc(c * sizeof(int));
    for (i = 0; i < r; i++)
      for (j = 0; j < c; j++)
        t[i][j] = val;
    return t;
  }
```

开发一个 ADT 函数来打印图的邻接矩阵表示是一个简单练习（见练习 17.16）。程序 17.5 描述了一种适合于稀疏图的不同实现：它打印出与每个顶点相邻的顶点，如图 17-9 所示。这些程序（特别是其输出）清晰地描述了基本性能权衡。为了打印出数组，我们需要 V^2 个元素的空间；打印出表，只需要 $V+E$ 个数的空间。对于稀疏图，与 $V+E$ 相比，V^2 很大，我们选择表表示；对于稠密图，E 和 V^2 可比时，我们选择数组表示。正如我们将要看到的，在比较邻接矩阵表示与显式表表示时，会做出基本的权衡。

```
 0:  1 2 5 6
 1:  0
 2:  0
 3:  4 5
 4:  3 5 6
 5:  0 3 4
 6:  0 4
 7:  8
 8:  7
 9:  10 11 12
10:  9
11:  9 12
12:  9 11
```

图 17-9　邻接表格式

此表还描述了表示图 17-1 中的图的另一种方式：我们为每个顶点关联一个其相邻顶点集（那些与它有一条边相连的顶点）。每条边影响两个集合：对于图中的每条边 u-v，u 出现在 v 的集合中，v 出现在 u 的集合中。

程序 17.5　图 ADT 输出（邻接表格式）

打印稀疏图的整个邻接矩阵不实用，因此我们选择简单输出。对于每个顶点，输出那些与该顶点通过边相连的顶点。

```
void GRAPHshow(Graph G)
  { int i, j;
    printf("%d vertices, %d edges\n", G->V, G->E);
    for (i = 0; i < G->V; i++)
      {
        printf("%2d:", i);
```

```
        for (j = 0; j < G->V; j++)
          if (G->adj[i][j] == 1) printf(" %2d", j);
        printf("\n");
      }
  }
```

对于大型稀疏图，邻接矩阵表示并不是令人满意的方式：数组要求 V^2 位的存储空间而且只是初始化就需 V^2 步。在稠密图中，当边的数目（矩阵中 1 的个数）与 V^2 成正比时，这个开销是可接受的，因为要求与 V^2 成正比的时间来处理边，而不论我们使用哪种表示。然而，在稀疏图中，初始化数组就会成为算法运行时间的主要因素。而且，我们甚至没有足够空间存储矩阵。例如，我们可能面临有数百万个顶点和数千条边的图，但我们不想/也不能付出存储数邻接表的数万亿个 0 的代价。

另一方面，当我们的确需要处理大型稠密图时，代表不存在边的 0 只对空间需求增加一个常量因子，这就使我们只需访问一次数组，就能确定某条边是否存在。例如，不允许平行边在邻接矩阵中是自动的行为，但在其他表示中则开销很大。如果我们确实有空间存放邻接矩阵，要么 V^2 很小使得表示的时间可被忽略，要么我们运行一个复杂算法，需要多于 V^2 步来完成，那么无论图是否是稠密的，均可以选择邻接矩阵表示。

练习

▷ **17.16**　给出 GRAPHshow 的实现，打印出类似图 17-7 中所示的二维 0-1 矩阵。其中包含邻接表形式的图 ADT 实现（程序 17.3）。

▷ **17.17**　给出图 17-2 中所示三个图的邻接矩阵表示。

17.18　给定一个图，考虑与前一个图相等的另一个图，但使其中两个顶点的名字（对应的整数）交换。试问这两个图的邻接矩阵有何不同？

▷ **17.19**　在图 ADT 中增加一个函数 GRAPHedge，使客户程序可以检查连接两个顶点的边是否存在，并给出邻接矩阵的一种实现。

▷ **17.20**　修改程序 17.3，并按练习 17.19 中描述的方法改进，对于 $w > v$，不存储数组元素 a[v][w]，使空间需求约减半。

17.21　修改程序 17.3，并按练习 17.19 中描述的方法改进，使用位数组，而不是整数数组。也就是说，如果你的计算机每个字有 B 位，则你的实现应该能使用 V^2/B 个字（不是 V^2）来表示 V 个顶点的一个图。进行实验研究来评价使用位数组实现 ADT 操作所需要的时间性能。

17.22　修改程序 17.4 来检查 malloc 返回码，使其在没有足够内存可用于表示数组时返回 0。

17.23　编写程序 17.4 的一个版本，要求只调用一次 malloc。

17.24　在程序 17.3 中，增加 GRAPHcopy 和 GRAPHdestroy 的实现。

○ **17.25**　假设一个小组中的所有 k 个顶点有连续的索引。由邻接矩阵如何确定组中的顶点是否构成一个团？在图 ADT 的邻接矩阵实现（程序 17.3）中增加一个函数，找出构成一个团且连续索引最多的一组顶点，所用时间与 V^2 成正比。

17.4　邻接表表示

对于非稠密的图，合适的标准表示称为*邻接表*（adjacency-list）表示，其中将与每个顶点连接的所有顶点保存在一个链表中，并将链表关联到那个顶点。我们维持一个链表数组，

给定一个顶点，可以直接访问到其链表；使用链表时，增加新边只需常量的时间。

程序 17.6 是基于这种方法对程序 17.1 中的 ADT 接口的一种实现，而且图 17-10 给出了一个示例。要向此图的表示中添加一条连接 w 和 v 的边，我们将 w 添加到 v 的邻接表中，将 v 添加到 w 的邻接表中。用这种方法，我们依然可以在常量的时间添加新的边。而所用的空间只与顶点数及边数之和成正比（而不是邻接矩阵表示中的顶点数的平方）。我们在两个不同地方表示每条边：连接 v 与 w 的一条边作为结点出现在两个邻接表中。包含这两个表示很重要的；否则，将不能有效地回答诸如“哪些顶点与顶点 v 直接相连?”这样简单的问题。

与程序 17.3 不同，程序 17.6 构建多重图，这是因为它没有删除平行边。在邻接表结构中检查重复边需要搜索链接表，这样所需时间与 V 成正比。类似地，程序 17.6 并不含删除边（remove edge）操作实现。而且，添加这样的实现并不复杂（见练习 17.28），但是每次删除所需时间会与 V 成正比。这些开销使得基本邻接表的表示不适合于那些涉及大规模图的应用问题，因为其中不允许平行边，或者是涉及频繁删除边操作的应用。在 17.5 节中，我们讨论基本数据结构技术的应用来扩充邻接表，使其支持常量时间删除边和常量时间平行边检测（parallel edge detection）操作。

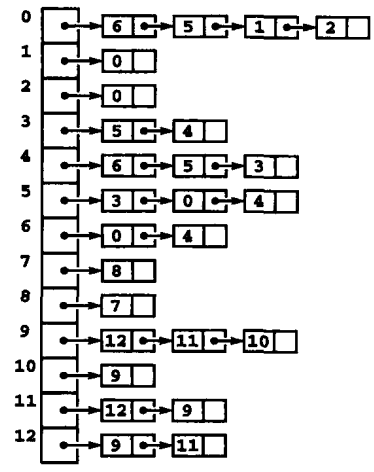


图 17-10　邻接表数据结构

此图描述图 17-1 中图的一种链表数组表示。所用空间与结点数与边数的和成正比。为了找出与某一给定顶点 v 连接的顶点索引，需要检查数组中的第 v 个位置，在此包含一个指向链表的指针。在该链表中，对应每个与 v 连接的顶点有一个结点。结点在链表中出现的次序取决于我们构造链表时所使用的方法。

如果图的顶点名不是整数，那么（像邻接矩阵）两个不同的程序可能会以两种不同方式将顶点名与 $0 \sim V-1$ 之间的整数相关联，导致两种不同的邻接表结构（例如，程序 17.10）。我们不能期望区分两个不同的结构是否表示同一个图，这是由图同构问题的难度所决定的。

而且，有了邻接表，即使是给定了顶点编号，对于给定的图也有很多种表示。无论边在邻接表中以何种顺序出现，邻接表结构都表示同一个图（见练习 17.31）。了解邻接表的这一特征很重要，因为边在邻接表中出现的顺序反过来会影响到算法对边处理的顺序。也就是说，邻接表结构决定了各种算法是如何处理图的。虽然无论边是以何种顺序在邻接表中，算法都要产生正确的结果，但是对于不同的顺序，算法会通过不同的计算顺序来产生该答案。如果算法并不需要检查图的所有边，此效果会对它所需要的时间产生影响。如果有多个正确答案，不同输入顺序可能导致不同输出结果。

程序 17.6　图 ADT 实现（邻接表）

程序 17.1 中的接口实现使用链表数组，每个顶点对应一个链表。边 v-w 由链表 v 中结点 w 和链表 w 中结点 v 所表示。与程序 17.3 中一样，GRAPHedge 只将每条边的两个表示中的一个放进输出数组。GRAPHcopy、GRAPHdestroy 以及 GRAPHremove 的实现都被省略。GRAPHinsertE 代码不检查重复边，从而保持插入时间为常量。

```
#include <stdlib.h>
#include "GRAPH.h"
typedef struct node *link;
struct node { int v; link next; };
```

```
struct graph { int V; int E; link *adj; };
link NEW(int v, link next)
  { link x = malloc(sizeof *x);
    x->v = v; x->next = next;
    return x;
  }
Graph GRAPHinit(int V)
  { int v;
    Graph G = malloc(sizeof *G);
    G->V = V; G->E = 0;
    G->adj = malloc(V*sizeof(link));
    for (v = 0; v < V; v++) G->adj[v] = NULL;
    return G;
  }
void GRAPHinsertE(Graph G, Edge e)
  { int v = e.v, w = e.w;
    G->adj[v] = NEW(w, G->adj[v]);
    G->adj[w] = NEW(v, G->adj[w]);
    G->E++;
  }
int GRAPHedges(Edge a[], Graph G)
  { int v, E = 0; link t;
    for (v = 0; v < G->V; v++)
      for (t = G->adj[v]; t != NULL; t = t->next)
        if (v < t->v) a[E++] = EDGE(v, t->v);
    return E;
  }
```

邻接表表示优于邻接矩阵表示的主要优点在于，它所使用的空间总是与 $E + V$ 成正比，而不是邻接矩阵中的 V^2。其主要缺点是，检查特定边是否存在需要花费的时间与 V 成正比，而在邻接矩阵中则是常数。本质上，这些差别源于利用链表和数组来表示依附于每个顶点的顶点集的不同。

因此，我们再次看到，如果要开发有效的图 ADT 实现，理解链式数据结构和数组的基本性质是至关重要的。之所以对这些性能差异感兴趣，是因为我们希望当各种操作都包括在 ADT 中时，避免在不可预料的情况下使用低效的实现。在 17.5 节中，我们讨论基本符号表算法学技术应用，从而认识到这两种结构的理论上的好处。不过，因为程序 17.6 是提供基本特性的一种简单实现，我们需要学习处理稀疏图的有效算法，并将它作为本书中很多实现的基础。

练习

▷ **17.26**　按照图 17-10 的格式，使用程序 17.6，显示按照顺序将以下边插入到初始为空的图中后的所得结果。

```
3-7  1-4  7-8  0-5  5-2  3-8  2-9  0-6  4-9  2-6  6-4
```

17.27　给出 GRAPHshow 的实现，使其与练习 17.16 和程序 17.5 有同样的作用，包含在邻接表图 ADT 实现中（程序 17.6）。

17.28　给出邻接表图 ADT 的实现（程序 17.6）中删除边函数 GRAPHremoveE 的一种实现。注意：记住可能有重复边。

17.29　在邻接表的图 ADT 实现（程序 17.6）中添加 GRAPHcopy 和 GRAPHdestroy 实现。

○ **17.30** 给出邻接表图表示的一个简单例子，它不能由程序 17.6 通过反复地添加边来构建。

17.31 要表示图 17-10 中所示的同一个图，有多少种不同的邻接表表示？

17.32 编写程序 17.6 的一个版本，按照顶点编号的顺序来保存邻接表。描述这种方法有用的一种情况。

○ **17.33** 在图 ADT（程序 17.1）中添加一个函数声明，删除自环和平行边。给出在基于邻接矩阵的 ADT 实现（程序 17.3）中该函数的一种平凡实现，并给出基于邻接表的 ADT 实现（程序 17.6）中该函数的一种平凡实现，要求所用时间与 E 成正比，所用额外空间与 V 成正比。

17.34 扩展练习 17.33 的解决方案，使其能够删除度为 0 的顶点（孤立点）。注意：要删除顶点，需要对其他顶点重新命名，重新构建数据结构，而且只能做一次。

● **17.35** 编写一个邻接表表示（程序 17.6）的 ADT 函数，折叠只由度为 2 的顶点构成的路径。明确地说，图中每个度为 2 且没有平行边的顶点出现在某条路径 u-...-w 上，其中 u 和 w 要么相等，要么度不为 2。用 u-w 替换掉这样的路径，然后像在练习 17.34 那样删除所有未用的度为 2 的顶点。注意：这个操作可能会引入自环和平行边，但它保持了未被删除的顶点的度不变。

▷ **17.36** 给出将练习 17.35 中描述的变换应用到图 17-1 中的示例图上所得的一个（多部）图。

17.5 变量、扩展和开销

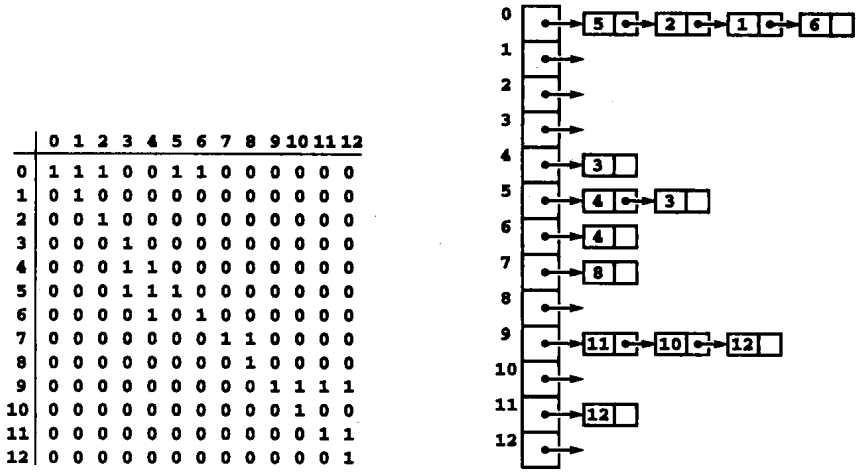
在这一节里，我们描述大量在 17.3 节和 17.5 节中讨论的图表示的改进方法。这些改进可分为两类。第一，基本邻接矩阵和邻接表机制可容易地扩展为表示其他类型的图。在相关的章节中，我们将详细讨论这些扩展并给出例子；这里，我们只简略讨论。第二，我们常常需要修改或者增大数据结构使得某些操作更为高效。在接下来的章节中将会这样做；在本节中，我们会讨论将数据结构设计技术应用到有效实现几种基本函数。

对于有向图，每条边只表示一次，如图 17-11 所示。有向图中的边 v-w 在邻接矩阵的第 v 行和第 w 列的元素表示为 1，或者在邻接表表示的 v 的邻接表中出现 w。这些表示要比无向图中对应的表示更为简单，但是与无向图相比，非对称性使有向图成为更为复杂的组合对象，我们在第 19 章中将会看到。例如，标准邻接表表示没有给出直接的方法来找出一个有向图中进入一个顶点的所有边，因此，如果需要支持这个操作，我们必须做出相应的修改。

对于*加权图*（weighted graph）和*网*（network），我们使用权值代替布尔值来填充邻接矩阵（使用某些不存在的权值来表示不存在的边）；在邻接表表示中，我们在邻接结构中加上一个顶点的权值，或者在邻接表元素中加上一个边权值。

常常需要在一个图中的顶点或边中关联更多的信息，使得图可以对更复杂的对象建模。在合适的时候，可以对程序 17.1 中的 Edge 类型进行扩展，使每条边上关联额外的信息，然后使用邻接矩阵中类型或者邻接表中表结点的实例。或者，由于顶点名为介于 0 和 $V-1$ 之间的整数，我们可以使用顶点索引数组来为顶点关联额外信息，也许使用一个合适的 ADT 来实现。另一种做法，我们可以简单地使用一个单独的符号表 ADT，为每个顶点和边关联额外信息（见练习 17.46 和程序 17.10）。

为了处理各种特定的图处理问题，我们通常要定义类，其中包含与该图相关的特定辅助数据结构。最常见的此类数据结构为顶点索引数组，比如在第 1 章我们使用顶点索引数组回答了连通性查询。在本书的大量实现中我们都使用了顶点索引数组。



	0	1	2	3	4	5	6	7	8	9	10	11	12
0	1	1	1	0	0	1	1	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0	0	0	0	0	0
2	0	0	1	0	0	0	0	0	0	0	0	0	0
3	0	0	0	1	0	0	0	0	0	0	0	0	0
4	0	0	0	1	1	0	0	0	0	0	0	0	0
5	0	0	0	1	1	1	0	0	0	0	0	0	0
6	0	0	0	0	1	0	1	0	0	0	0	0	0
7	0	0	0	0	0	0	0	1	1	0	0	0	0
8	0	0	0	0	0	0	0	0	1	0	0	0	0
9	0	0	0	0	0	0	0	0	0	1	1	1	1
10	0	0	0	0	0	0	0	0	0	0	1	0	0
11	0	0	0	0	0	0	0	0	0	0	0	1	1
12	0	0	0	0	0	0	0	0	0	0	0	0	1

图 17-11　有向图的表示

在有向图的邻接矩阵和邻接表表示中，每条边只有一种表示，将图 17-1 中的边集解释为有向图（见图 17-6 所示，左图），它的邻接矩阵和邻接表表示分别如左图和右图所示。

作为一个例子，假设我们希望了解图中的一个顶点 v 是否是孤立的，即 v 的度为 0 吗？对于邻接表表示，我们可以很快得到这个信息，只需要检查 adj[v] 是否为空。但对于邻接表表示，我们需要检查每行或每列中与 v 对应的所有 V 个元素，才能知道每个顶点与其他顶点连接的情况；对于边数组表示法，除了检查所有 E 条边来确定是否与 v 相关，我们没有更好的方法。为了避免这些可能耗时的计算，我们可以实现一个简单的在线算法，维护一个顶点索引的数组，使其可以在常量时间内查找任何一个顶点的度（见练习 17.40）。这样一个简单问题的潜在性能差异在图处理中非常典型。

表 17-1 显示了各种简单图处理操作对我们使用的表示方法在开销上的相关性。在讨论更加复杂操作的实现之前，有必要先研究一下该表；它有助于你对各种基本操作的难度有一个直观的认识。直接观察代码可得以下列出的大多数的开销，仅最后一行除外，这会在本节的最后详细讨论。

表 17-1　图处理操作最坏情况开销

基本图处理 ADT 操作的性能特征对于不同的图表示法有很大的不同，即使是对于简单任务，如此表中描述的最坏情况开销所示（对于较大的 V 和 E，都相差一个常量因子）。这些开销是前面章节中所描述的简单实现。对于影响开销的各种修改也在本节中讨论。

	边数组	邻接矩阵	邻接表
占用空间	E	V^2	$V+E$
初始化为空	1	V^2	V
复制	E	V^2	E
销毁	1	V	E
插入边	1	1	1
查找/删除边	E	1	V
v 是否是独立点	E	V	1
从 u 到 v 是否存在路径	$E\lg^* V$	V^2	$V+E$

在一些情况下，可以修改表示法使得操作更有效，但我们还要注意这样做是否会增加其他简单操作的开销。例如，邻接矩阵的销毁（destroy）操作是 C 对二维数组（见 3.7 节）的嵌套数组分配模式的产物。不难将此开销降至为常量（见练习 17.23）。另一方面，如果图的边是非常复杂的结构，使得矩阵中的元素是指针，那么销毁一个邻接矩阵所花费的时间与 V^2 成正比。

由于查找边和删除边的操作在典型应用中使用频繁，我们详细讨论查找边和删除边的操作。特别是，我们需要查找边的操作能够删除或禁止平行边。如 17.3 节中看到的，如果使用邻接矩阵表示法，这些操作非常简单。只需要检查或者设置一个可以直接索引的数组元素即可。但是在邻接表表示中如何有效地实现这些操作呢？一种方法接下来描述，另一种方法在练习 17.48 中描述。这两种方法都是基于符号表的实现。如果我们使用动态散列表（见 14.5 节），这两种方法所需空间与 E 成正比，使我们可以在常量时间执行其中的任何一种操作（平均意义，平摊分析得此结果）。

明确地说，在使用邻接表时为了实现查找边的操作，我们可以使用一个辅助的符号表用于边。可以为边 v-w 指定整数关键字 $v * V + w$，并使用第 4 部分中的符号表的任一实现。（对于无向图，我们可能为 v-w 和 w-v 指定相同的关键字。）可以在首次检查每条边是否已经插入后，将它插入到符号表中。如果愿意，我们可以禁止平行边（见练习 17.47），在符号表中维护平行边的一个副本记录，或者构建一个完全基于这些操作的图 ADT 实现（见练习 17.46）。在目前的情况下，我们对这项技术的主要兴趣在于，它能为我们提供在邻接表上常量时间的查找边实现。

为了删除边，我们可以在符号表的记录中为每条边设置一个指针，指向其邻接表结构的表示。然而即使有这个信息，也不足以在常量时间内删除边，除非表是双向链表（见 3.4 节）。而且，在无向图中，也不能从邻接表中删除结点，因为每条边会出现在两个不同的邻接表中。解决这个难题的一种方法是将两个指针都放在在符号表中；另一种方法是将对应特殊边的这两个表结点链接在一起（见练习 17.44）。无论哪一种方法，我们都可以在常量时间内删除一条边。

删除顶点的开销更大。在邻接矩阵表示中，我们需要从矩阵中删除一行和一列，比起重新从一个更小的矩阵开始，这样做的开销并不会少多少。如果我们使用邻接表表示，那么从该顶点的邻接表中删除结点还不够，因为邻接表中的每个结点指定了另一个顶点，我们必须查找该顶点的邻接表以删除表示同一条边的另一个顶点。如果要使删除一个顶点的时间与 V 成正比，则需要另外的链接来支持上段描述的常量时间的边删除操作。

我们在这里省略这些操作的实现，是因为使用第 1 部分中的基本技术，这些工作可以作为简单的编程练习。在与静态图有关的典型应用中，维护有多个指针的复杂结构并不合适。而且在实现其他地方并不使用的图处理算法时，我们希望避免陷入维护其他指针的细节中。在第 22 章中，我们还要讨论一种类似结构的实现，它在我们本章所讨论的强大的通用算法中起着重要的作用。

在描述和开发感兴趣的算法实现时，为了简明起见，我们使用最简单的合适表示。一般而言，我们会在与当前任务相关的代码中直接构造一个链接或者辅助数组。很多编程人员认为这种机制是理所当然的，他们知道，实际上维护有多个不同组件的数据结构的完整性确实是一件挑战性的任务。

在性能调整的过程中，我们还可能修改基本数据结构，以节省空间或时间，特别是在处理大规模的图时（或大量小规模的图）。例如，对于大规模静态图，我们可以改变表示方

法，使用变长数组而不是邻接表来表示依附于每个顶点的顶点集，从而大大改善算法的性能。采用这种技术，我们最终可以只用 $2E$（小于 V）个整数和 V 个整数（小于 V^2）来表示一个图（见练习 17.51 和 17.53）。对于处理大规模的静态图，这些表示很有吸引力。

我们讨论的很多算法可以很容易改造为本节所讨论的所有变型算法，因为它们是以几种高级抽象操作为基础的，如“对于与顶点 v 邻接的每条边执行如下操作。”实际上，我们考虑的某些程序只在这种抽象操作的实现方法上有所不同而已。

为什么不能在更高级的抽象开发这些算法，然后讨论表示数据和实现相关操作的不同选项呢？就像我们在本书中的很多实例中所做的那样。这个问题的答案并不是一句话就能说清楚的。对于稀疏图常选择用邻接表，对于稠密图常选择用邻接矩阵，这些选择我们很清楚。在主要实现中，我们会直接选择这两种特殊表示中的其中一种，因为使用低级表示的代码中，算法的性能特征非常清晰，而且比起那些利用高级抽象编写的代码，该代码一般而言不难读懂和理解。

在某些实例中，算法设计决策依赖于表示的某些性质。在一个更高的抽象级上处理，可能使我们不了解这种依赖性的相关知识。如果我们知道某种表示将导致很差的性能，而另一种表示不会，在不适当的抽象级考虑算法时，就会存在不必要的风险。如常，我们的目标是精细实现，从而可以对性能做出准确的说明。

使用严格的抽象方法可以解决这些问题，我们建立算法中需要的抽象操作的抽象层次。添加一个 ADT 操作来测试一条边的存在性是一个例子（见练习 17.19），我们还能建立与表示无关的代码，来处理与给定顶点邻接的每个顶点（见练习 17.60）。在很多情况下，这样的方法很有意义。然而，在本书中，我们关注的是在稠密图上使用直接访问邻接矩阵的代码和在稀疏图上直接访问邻接表的代码的性能，并增大数据结构以适合所处理的任务。

到目前为止我们所讨论的所有操作是简单的而又必要的数据处理函数；本节要讨论的是第 1 ~ 3 部分的基本算法和数据结构对于处理这些函数是有效的。当开发更复杂的图处理算法时，我们在找出特定实际问题的最佳实现时面临更困难的挑战。为了说明这一点，考虑表 17-1 中的最后一行，其中给出了确定两个给定顶点是否存在一条路径的开销。

在最坏情况下，17.7 节中的简单算法检查图中的所有 E 条边（如我们在第 18 章中讨论的一些其他方法的做法一样）。表 17-1 中的最下一行的中间和右边列的元素分别表示，该算法可能检查 V^2 个元素（对于邻接矩阵表示法）和所有 V 个链表头以及表中的所有 E 个结点（对于链表表示法）。这些事实表明该算法的运行时间是图表示大小的线性函数，但是它们也展示了两种异常情况：如果我们使用邻接矩阵表示来表示稀疏图，或者使用任何一种表示来表示极端稀疏的图（有大量孤立点），那么最坏情况下的运行时间不是图中边数的线性时间。为了避免重复讨论这些异常情况，我们假设所使用的表示的大小与图中的边数成正比。这一点对于大多数的应用而言是成立的。因为这些应用都涉及大量的稀疏图，因此要求用邻接表表示。

表 17-1 中最后一行左列是使用第 1 章中的合并 - 查找算法得出的（见 17.3 节）。由于这种方法只需要与 V 成正比的空间，因而很有吸引力，但也有缺点，即不能展示这条路径。这一项强调了完整而又准确地指定图处理问题的重要性。

即使考虑了所有这些因素，在开发实用图处理算法我们所面临的最重要的一个挑战性是，评价最坏情况性能分析的程度，如表 17-1 中的那些结果，会过度估计我们在实际中遇到的图的性能。关于图算法的大多数文献是根据这些最坏情况下的界限来描述性能的。这个信息在辨别算法具有不可接受的坏性能时是有帮助的，但不能辨别出几个简单、直接的程序

是否适合一个给定的应用。这种情况在为图算法开发平均情况性能的有用模型时难度更为加大，只能提供（可能不可靠）基准测试和（可能过于保守）最坏情况性能保证。例如，我们在第 18 章中讨论的图搜索算法对于查找给定顶点的之间的一条路径都是有效的线性时间算法，但其性能特征却有很大差异，这与被处理的图及其表示有关。实际使用图处理算法时，我们总是希望消除证明的最坏情况性能特征与期望的实际性能特征之间的不一致性。这种思想将在本书中反复出现。

练习

17.37 开发稠密度多重图的一种邻接矩阵表示，并给出使用该表示的程序 17.1 的一个 ADT 实现。

○ **17.38** 为什么不使用图的一种直接表示（对图精确建模的一种数据结构，其中将顶点表示为已分配的记录，边表包含指向顶点的链接而不是指向顶点名）？

17.39 编写一个独立于表示的 ADT 函数，它返回指向顶点索引数组的一个指针（给定图中每个顶点的度）。提示：使用 GRAPHedges。

17.40 修改邻接矩阵 ADT 实现（程序 17.3），使其在图表示中包含顶点索引的数组，用以存放每个顶点的度。增加一个返回给定顶点度的 ADT 函数 GRAPHdeg。

17.41 对于邻接表表示完成练习 17.40。

▷ **17.42** 在表 17-1 中增加一行，确定图中孤立点的数目所需的最坏情况开销。利用采用这三种表示法的各个函数实现来支持你的答案。

○ **17.43** 在表 17-1 中添加一行，确定一个给定图中是否存在入度为 V、出度为 0 的顶点所需的最坏情况开销。利用采用这三种表示法的各个函数实现来支持你的答案。注意：邻接矩阵表示的行（列）数应为 V。

17.44 对于邻接表图 ADT 实现（程序 17.6），使用正文中描述的带有十字链接的双向邻接表来实现一个常量时间的删除边函数 GRAPHremoveE。

17.45 试为练习 17.44 描述的双向邻接表图 ADT 实现（程序 17.6）增加一个删除顶点（remove vertex）的函数 GRAPHremoveV。

○ **17.46** 修改练习 17.15 的解，使用正文中描述的动态散列表，使得插入边和删除边的平摊时间为常量。

17.47 修改邻接表图 ADT 实现（程序 17.6），使用符号表来忽略重复边，使其在功能上等价于邻接矩阵图 ADT 实现（程序 7.3）。在符号表的实现中使用动态表，以保证你的实现所占用的空间与 E 成正比，且能在常量时间内插入和删除边（平均情况，平摊分析）。

17.48 试开发一个基于符号表数组表示的图 ADT 实现（每个顶点对应一个符号表，其中包含其邻接边链表）。符号表实现采用动态散列，以保证你的实现使用的空间与 E 成正比，且能在常量时间内插入和删除边（平均情况，平摊分析）。

17.49 基于一个以边数组作为参数并构建图的函数 GRAPHconstruct，开发一个面向静态图的图 ADT。开发使用程序 17.1 中的 GRAPHinit 和 GRAPHinsert 的 GRAPHconstruct 的一种实现（这样的实现可能对于练习 17.51 ~ 17.54 中描述实现进行性能比较有用）。

17.50 开发程序 17.1 的 GRAPHinit 和 GRAPHconstruct 的一个实现，其中使用练习 17.49 中描述的 ADT（这样的实现可能对于带有诸如程序 17.2 的驱动程序后向兼容比较有用）。

17.51 开发练习 17.49 中描述的 GRAPHconstruct 函数的一种实现，使用基于以下数据结构的压缩表示：

```
struct node { int cnt; int* edges; };
struct graph { int V; int E; node *adj; };
```

图中包含顶点数，边数和顶点的数组。顶点包含一个边数和一个数组，其中每个顶点下标对应每个邻接边。实现这种表示的 GRAPHshow。

● **17.52**　在练习 17.51 中增加一个函数，消除自环和平行边，类似于练习 17.33 中的工作。

○ **17.53**　开发练习 7.49 中描述的静态图 ADT 的一种实现，要求只使用两个数组来表示该图：一个是 E 个顶点的数组，另一个是指向第一个数组的 V 个索引或指针的数组。实现这种表示的 GRAPHshow。

● **17.54**　在练习 17.53 中增加一个函数，消除自环和平行边，类似于练习 17.33 中的工作。

17.55　开发一个图 ADT 接口，为每个顶点关联一对坐标 (x, y)，使你可以进行绘图。适当地修改 GRAPHinit，在初始化中增加函数 GRAPHdrawV 和 GRAPHdrawE，分别用于绘制顶点和边。

17.56　编写一个客户程序，使用你的接口来绘制加入到一个小规模图中的边。

17.57　开发练习 17.55 得到的接口的一种实现，生成以绘图作为输出的 PostScript 程序。

17.58　找出一种合适的图形接口，从而为练习 17.55 得到的接口开发一种实现，能够在显示屏的一个窗口内直接画图。

● **17.59**　对练习 17.55 和练习 17.58，使其包含擦除顶点和边的函数，并可用不同风格进行绘制，这样就可以编写出相应的客户程序，为正在执行的图算法提供动态的图形动画。

○ **17.60**　定义一个图 ADT 函数，使客户可以访问（运行客户提供的函数，以边作为参数）与给定顶点邻接的所有边。对于邻接矩阵表示和邻接表表示分别给出该函数的实现。为了测试你的函数，使用它来实现独立于表示的 GRAPHshow 版本。

17.6　图生成器

为了进一步开发出作为组合结构的图的各种各样的性质，我们现在讨论一些图类型的详细例子，稍后我们会用这些图来测试所研究的算法。其中某些例子是从应用中得到的，另一些例子是从数学模型得到的，这些数学模型不仅具有我们在实际的图中所发现的性质，而且可以扩展测试算法可用实验输入的范围。

为了使这些例子更具体，我们将它们作为程序 17.1 接口的扩展，因而可以在测试所讨论的图算法实现时直接使用它们。而且，我们考虑一个从标准输入读入任意名字对序列的程序，并构建顶点对应名字，边对应名字对的图。

我们在本节考虑的实现基于程序 17.1 中的接口，因而对于图的任何表示，它们在理论上都能正常工作。然而，在实际中，接口和表示的某些组合可能会产生难以接受的差性能，稍后我们会看到。

程序 17.7　随机图生成器（随机边）

此 ADT 函数通过生成 0 ~ $V-1$ 之间的 E 对随机整数对，并将其中的顶点解释为顶点标号，最后将顶点标号对解释为边，来构建一个图。它将对于是否含有平行边和自环的判断放在 GPAPHinsertE 的实现中，并假设此 ADT 实现将对图中边数和顶点数的记数分别放在 $G->E$ 和 $G->V$ 中。由于这种方法所生成的平行边的数量很大，因此这种方法一般并不适合生成大型的稠密图。

```
int randV(Graph G)
  { return G->V * (rand() / (RAND_MAX + 1.0)); }
Graph GRAPHrand(int V, int E)
  { Graph G = GRAPHinit(V);
    while (G->E < E)
      GRAPHinsertE(G, EDGE(randV(G), randV(G)));
    return G;
  }
```

程序 17.8　随机图生成器（随机图）

像程序 17.7 一样，此图客户程序生成 0 ~ $V-1$ 之间的随机整数对来创建一个图。但它使用一种不同的概率模型，其中每条可能边以某个概率 p 独立出现。p 值是按照期望的边数（$pV(V-1)/2$）等于 E 来计算的。此代码产生的任何一个图的边数近似于 E，但不可能准确为 E。这种方法主要适合于稠密图，因为它的运行时间与 V^2 成正比。

```
Graph GRAPHrand(int V, int E)
  { int i, j;
    double p = 2.0*E/V/(V-1);
    Graph G = GRAPHinit(V);
    for (i = 0; i < V; i++)
      for (j = 0; j < i; j++)
        if (rand() < p*RAND_MAX)
          GRAPHinsertE(G, EDGE(i, j));
    return G;
  }
```

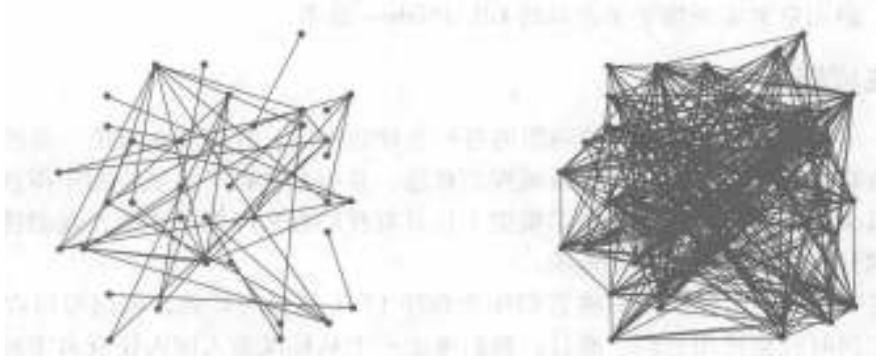


图 17-12　两种随机图

这两种随机图都有 50 个顶点。上面的稀疏图有 50 条边，而下面的稠密图有 500 条边。稀疏图不是连通的。其中每个顶点只与几个其他顶点相连；稠密图肯定是连通的，其中每个顶点平均而言都与其他 20 个顶点相连。这些图还表明开发能够画出任意图的难度（这里的顶点被放置在随机位置上）。

如常，我们关注“随机问题实例”不仅可以训练具有任意输入的程序，而且可以得到程序在实际应用中的性能。对于图，与我们考虑过的其他领域相比，后一个目标更含糊。虽然它仍然是一个很有价值的目标。我们将会遇到各种随机化的不同模型，先从如下的两个模型开始：

随机边　此模型实现起来很简单，如程序 17.7 中给出的生成器所示。对于给定的顶点数 V，我们通过产生 0 ~ $V-1$ 之间的数对来产生随机边。结果很可能是一个具有自环的随机多重图，而不是定义 17.1 中所定义的一个图。给定的数对可能有两个相同的数（因此，自环可能出现）；而且任何数对都可能重复出现多次（因此会出现平行边）。程序 17.7 不断地

生成边，直到有 E 条边生成，并在实现中需要判定是否存在平行边。如果要消除平行边，所生成的边数要比稠密图中所用的边数（E）大很多（见练习 17.62）；因此这种方法通常用于稀疏图。

随机图　随机图的这种经典数学模型是要考虑所有可能的边，并将每条边以一个固定的概率 p 加入到图中。如果我们希望图中的边数为 E，可以选择 $p = 2E/V(V-1)$。程序 17.8 是使用这种方法来生成随机图的一个函数。这个模型排除了重复边，但图中的边数只能平均等于 E。这种实现适合于稠密图，但不适合于稀疏图。因为它的运行时间为 $V(V-1)/2$，只生成 $E = pV(V-1)/2$ 条边。也就是说，对于稀疏图，程序 17.8 的运行时间是图大小的二次函数（见练习 17.67）。

这些模型得到了深入的研究。且不难实现，但它们并不需要生成我们在实际中所见到的具有某些性质的图。特别地，对地图、电路、调度、事务、网络以及其他情况建模的图，通常不仅是稀疏的，而且展示*局部*（locality）性质，这些图中，边更可能将某个给定顶点连接到某个特殊的顶点集中，而不大可能将该顶点连接到不在该集合中的顶点上。我们可能考虑对局部性建模的很多不同的方法。如下图中所示几个例子。

***k*-近邻图**　图 17-13 的左图描述由对随机边图生成器的一个简单修改而得到，其中我们随机选取第一个顶点 v，然后从那些其下标与顶点 v 相距在常数距离 k 的顶点中随机选取第二个顶点（当顶点排列成一个圆圈时，从 $V-1$ 回到 0）。这样的图很容易生成，肯定会展示随机图中未发现的局部性。

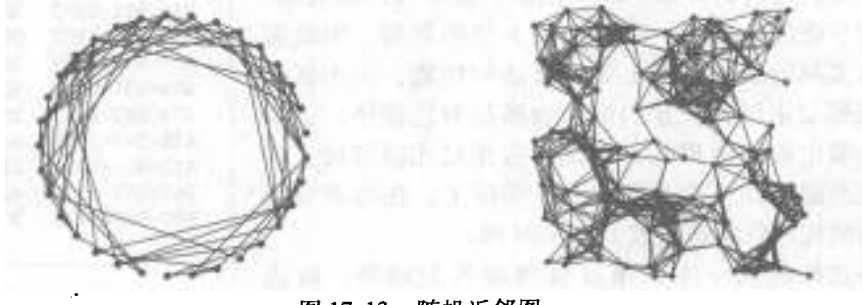


图 17-13　随机近邻图

这些图描述了稀疏图的两个模型。左面的近邻图有 33 个顶点、99 条边。每条边将顶点限制连接到与其标号小于 10（模 V）的范围。右面的欧几里得近邻图对我们在应用中可能发现的图建模，其中顶点固定到几何位置。顶点是平面上的随机点；边在给定的距离 d 内连接任何顶点。此图是稀疏的（177 个顶点，1 001 条边）；通过调整 d，我们可以生成想要的任何密度的图。

欧几里得近邻图　图 17-13 的右图是由生成器生成平面上 0 ~ 1 之间的随机坐标的 V 个顶点，然后生成连接距离 d 内的任何两个点的边而得到。如果距离 d 较小，则图是稀疏的；如果 d 较大，则图是稠密的（见练习 17.73）。对于处理地图、电路或顶点与几何位置相关联的其他应用时可能遇到的这种类型的图，可用此图建模。它们很容易可视化显示，以一种直观的方式来展示算法的性质，并展示在这些应用中所发现的很多结构性质。

这种模型的一种可能缺陷是这些图在稀疏时可能不是连通的；此外其他问题还包括：这些图中不太可能有度数大的顶点，而且不会有很长的边。如果必要，对这些模型进行修改使其能够处理这些情况，或者可以考虑一些为其他情况建立模型的类似示例（例如，见练习 17.71 和练习 17.72）。

还有一种做法，就是可以在实际的图上测试我们的算法。在很多应用中，并不缺少可以用来测试算法的有实际数据的问题实例。例如，由实际地理数据得到的大型图很容易

找到；在接下来的两段中给出了另外两个例子。使用真实数据工作而非随机图模型的优点是，我们可以随着算法的发展得到实际问题的解决方案。缺点是我们可能失去通过算法分析预则算法性能的好处。我们在第 18 章的末尾准备比较同一问题的几种算法时，还将讨论这个主题。

事务图 图 17-14 描述了我们在一个电话公司的计算机上找到的图的一小部分。顶点定义为电话号码，边定义为 i 和 j 组成的对，具有以下性质：在某段固定的时间内，i 给 j 拨打过电话。这个边集表示了一个多重图。它肯定是稀疏的，因为每个人只与可用电话的一个很小部分打电话。它可以作为很多其他应用的代表。例如，金融机构的信用卡和商家账户记录可能都有类似信息。

函数调用图 我们可以将图与任何计算机程序关联。函数看作顶点，函数 X 调用函数 Y 时，则在 X 和 Y 之间有一条边。我们可以指导程序建立这样的一个图（或让编译器来完成）。我们关注两种完全不同的图：对于静态图，在编译时创建边，对应着出现在各个函数的程序文本中的函数调用；对于动态图，在运行时创建边，对应调用实际发生。我们使用静态函数调用图来研究程序结构，动态函数调用来研究程序行为。这些图一般而言是很大且稀疏的。

在类似于此的应用中，我们面临大量的数据，因此可能倾向于在实际的示例数据上研究算法的性能，而不是基于随机数据模型来研究。我们可能会随机对边排序，或者在算法的决策中引入随机性，以试图避免退化的情况，不过这与生成图随机图完全是两回事。实际上，在很多应用中，了解图结构的性质本身就是一个目标。

900-435-5100	201-332-4562
415-345-3030	757-995-5030
757-310-4313	201-332-4562
747-511-4562	609-445-3260
900-332-3162	212-435-3562
617-945-2152	408-310-4150
757-995-5030	757-310-4313
212-435-3562	803-568-8358
913-410-3262	212-435-3562
401-212-4152	907-618-9999
201-232-2422	415-345-3120
913-495-1030	802-935-5112
609-445-3260	415-345-3120
201-310-3100	415-345-3120
408-310-4150	802-935-5113
708-332-4353	803-777-5834
413-332-3562	905-828-8089
815-895-8155	208-971-0020
802-935-5115	408-310-4150
708-410-5032	212-435-3562
201-332-4562	408-310-4150
815-511-3032	201-332-4562
301-292-3162	505-709-8080
617-833-2425	208-907-9098
800-934-5030	408-310-4150
408-982-3100	201-332-4562
415-345-3120	905-569-1313
413-435-4313	415-345-3120
747-232-8323	408-310-4150
802-995-1115	908-922-2239

图 17-14 事务图

像这样的数对序列可能表示本地交换机上的电话号码表，或者账户之间的资金转移，或者是涉及带有唯一标识符的实体间事务的类似情况。这种图很少是随机的——某些电话的使用频繁大大高于其他电话，而某些账户则比其他账户要活跃得多。

在一些这样的例子中，顶点自然命名为对象，而边命名为对象对。例如，事务图可能由电话号码对的序列构建，欧几里得图可能由城市对或城镇对的序列构建。程序 17.9 是我们使用构建这种公共情况的图的一个客户程序。为了客户便利，需要将边集定义为图，并根据边中顶点的使用来推导出顶点名集合。具体地说，程序从标准输入读入符号对的序列，使用符号表将 0 ~ $V-1$ 之间的顶点编号关联到符号（其中 V 是输入中不同符号的数目），并通过插入边构建一个图，如程序 17.1 和 17.8 所示。我们可以改变任何符号表的实现来支持程序 17.9 的需求；程序 17.10 是使用三叉搜索树（TST）的一个例子（见第 14 章）。这些程序使我们很容易在实际图上测试我们的算法，这些图可能是任何概率模型所不能准确刻画的。

程序 17.9 也很重要，因为它证实了我们在所有算法中所做的假设：顶点名是 0 和 $V-1$ 之间的整数。如果有一个图，它有其他的顶点名集，那么表示一个图的第一步是使用一个诸如程序 17.9 的程序来将顶点名映射到 0 ~ $V-1$ 之间的整数上。

有些图是根据元素之间的隐式连接得到的。我们并不关注这样的图，但会在后面的几个例子中指出它们的存在，并提供了几个有关的练习。在处理这种图时，肯定可以编写程序通

过枚举所有的边来构造显式图；但是对于特定问题还可能有其他的解决方案，这种方案无需我们枚举出所有边，因此可以在亚线性的时间内运行。

程序 17.9　由符号对构建一个图

此函数使用一个符号表从标准输入读入符号对来构建一个图。符号表 ADT 函数 STindex 将一个整数与一个符号关联：在大小为 N 的表中进行不成功查找中，将该符号添加到表中，并关联整数 $N+1$；在成功的查找中，它简单地返回以前关联到此符号的整数。第 4 部分的任何符号表方法都可以为此适当修改；例如，如程序 17.10 所示。检查边数不超过 Emax 的代码已省略。

```
#include <stdio.h>
#include "GRAPH.h"
#include "ST.h"
Graph GRAPHscan(int Vmax, int Emax)
  { char v[100], w[100];
    Graph G = GRAPHinit(Vmax);
    STinit();
    while (scanf("%99s %99s", v, w) == 2)
      GRAPHinsertE(G, EDGE(STindex(v), STindex(w)));
    return G;
  }
```

分离度图　考虑由 V 个元素所得的子集集合。我们定义一个图，其中顶点对应子集并集中每个元素，如果两个顶点出现在同一子集中，则这两个顶点之间有一条边相连（见图 17-15）。如果需要，只要边的标号指定了适当的子集，该图可能是一个多重图。依附于某一元素 v 的所有元素被称为 v 的 1 度分离（1 degree of separation）。否则，依附于任意与 v 成不大于 i 度分离的元素的所有元素被称为 v 的 $i+1$ 度分离（i + 1degree of separation）。这个构造过程从数学家（Erdös 数）到影迷（Kevin Bacon 分离度）的很多人都很感兴趣。

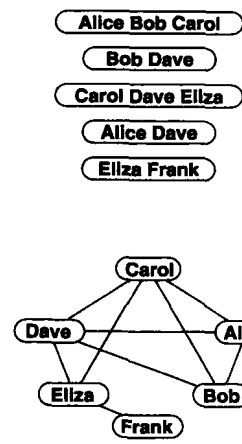


图 17-15　分离度图

下图是由上图中的组定义，每个顶点表示一个人，当他们在同一组时，则有一条边相连。图中的最短路径长度对应分离度。例如，Frank 与 Alice 和 Bob 是 3 度分离。

区间图　考虑实线上 V 个区间的一个集合（实数对）。定义一个图，其中每个顶点对应一个区间，如果相应区间相交（有共同点），则在这些顶点间有边相连。

de Bruijn 图　假设 V 是 2 幂。我们将顶点对应每个小于 V 的非负整数、从 i 到 $2i$ 的每个顶点和 $(2i+1) \bmod \lg V$ 存在边。这些图对于研究定长移位寄存器中一系列操作可能出现的序列值很有用。在这些操作中，我们不断地将所有位左移一个位置，去掉最左的一位，并用 0 或 1 填充最右位。图 17-16 描述了有 8、16、32 和 64 个顶点的 de Bruijn 图。

对于本节讨论的各种类型的图，其特性可谓千差万别。然而，对程序来说都看起来一样：它们只是边的集合。如我们在第 1 章中看到的，了解即使是关于它们的最简单的事实在计算上也存在挑战性。在这本书中，我们讨论大量精巧的算法，它们被开发出来用于解决与多种类型的图相关的实际问题。

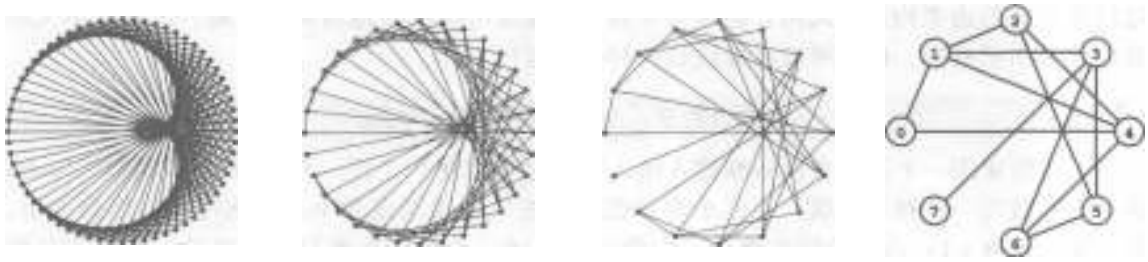


图 17-16　de Bruijn 图

阶为 n 的 de Bruijn 图有 2^n 个顶点，且从 i 到 $2i \bmod 2^n$ 及 $(2i+1) \bmod 2^n$ 存在边。这里的图是阶为 6、5、4 和 3（从上到下）的隐含无向 de Bruijn 图。

程序 17.10　顶点名的符号索引

这是字符串关键字（程序 17.9 的补充说明中描述过）的符号表索引函数的实现，完成了向已有 TST（见程序 15.8）中的一个结点添加 index 域的工作。每个关键字所关联的索引存储在该结点的索引域中，对应其字符串结束字符。当到达一个搜索关键字的末尾时，需要时就设置其索引，并且设置一个全局变量，在所有对函数的递归调用返回之后，将它返回给调用者。

```
#include <stdlib.h>
typedef struct STnode* link;
struct STnode { int index, d; link l, m, r; };
static link head;
static int val, N;
void STinit()
  { head = NULL; N = 0; }
link stNEW(int d)
  { link x = malloc(sizeof *x);
    x->index = -1; x->d = d;
    x->l = NULL; x->m = NULL; x->r = NULL;
    return x;
  }
link indexR(link h, char* v, int w)
  { int i = v[w];
    if (h == NULL) h = stNEW(i);
    if (i == 0)
      {
        if (h->index == -1) h->index = N++;
        val = h->index;
        return h;
      }
    if (i < h->d) h->l = indexR(h->l, v, w);
    if (i == h->d) h->m = indexR(h->m, v, w+1);
    if (i > h->d) h->r = indexR(h->r, v, w);
    return h;
  }
int STindex(char* key)
  { head = indexR(head, key, 0); return val; }
```

基于本节所提出的一些例子，可以看出比起我们在第 1 ~ 4 部分所研究的其他基本算法来说，图是复杂的组合对象。在许多情况下，我们在应用中需要考虑的图很难甚至是不可能

刻画的。在随机图上执行很多的算法常常有应用上的局限性，因为很难相信随机图与应用中所出现的图具有相同的结构特征。克服这个缺点的常用方法是设计在最坏情况下执行很好的算法。尽管这种方法在某些情况下是成功的，但在其他情况下则会因为过于保守而凸显其最坏性能。

假设对由所讨论的随机图模型所产生的图进行性能研究，可以给出足够准确的信息来预测实际图的性能，对此往往不能证实。但是本节所讨论的图生成器是有用的，可以帮助我们测试算法实现以及理解我们算法的性能。在试图预测应用的性能之前，我们必须至少要验证对于应用数据和何种模型之间关系或使用的样本数据所做出的可能假设。再将模型应用任何领域之前，这样的验证是明智的，在处理图时这一点尤为重要，因为我们会遇到大量各种类型的图。

练习

▷ 17.61　在我们使用程序 17.7 来生成密度为 αV 的随机图时，多大比例的生成边是自环？

● 17.62　使用程序 17.7 来生成密度为 α 且顶点数为 V 的随机图时，试计算所生成的平行边的期望数。并利用此计算结果画出图表（$V=10$，100 和 1 000），以 α 的函数形式显示生成的平行边比例。

● 17.63　试找出在线的一个大型无向图，可以是基于网络连通性的信息，也可以是一个由文献合作者或电影中的演员所定义的分离度图。

▷ 17.64　对于选择好的 V 和 E 值集，编写一个生成随机稀疏图的程序，并打印它用于图表示所使用的空间以及构建此图所需的时间。用一个稀疏图 ADT 实现（程序 17.6）以及随机图生成器（程序 17.7）来测试你的程序。从而对此模型下所导出的图进行有意义的实验研究。

▷ 17.65　对于选择好的 V 和 E 值集，编写一个生成随机稠密图的程序，并打印它用于图表示所使用的空间以及构建此图所需的时间。用一个稠密图 ADT 实现（程序 17.3）以及随机图生成器（程序 17.8）来测试你的程序。从而对此模型下所导出的图进行有意义的实验研究。

● 17.66　试给出程序 17.8 所生成的边数的标准方差。

● 17.67　试编写一个程序，使用与程序 17.8 中完全相同的概率生成各种可能的图，要求使用的时间和空间只与 $V+E$ 成正比，而非与 V^2 成正比。像练习 17.64 中所述测试你的程序。

○ 17.68　试编写一个程序，以与程序 17.7 中完全相同用的概率生成各种可能的图，要求使用的时间与 E 成正比，即使当密度近似为 1 时也这样。像练习 17.65 中所述测试你的程序。

● 17.69　试编写一个程序，以等概率生成 V 个顶点和 E 条边的图（见练习 17.9）。像练习 17.64（低密度）和练习 17.65（高密度）中所述测试你的程序。

○ 17.70　试编写一个程序，对于按照 $\sqrt{V}\times\sqrt{V}$ 网格排列的顶点，生成将此网格中的顶点与其近邻连接的随机图（见图 I.2），要求每个顶点有 k 条额外的边连接到另一个随机选择的目的顶点（每个目的顶点等概率出现）。确定如何选择 k，使得期望边数为 E。像练习 17.64 中所述测试你的程序。

○ 17.71　试编写一个程序，对于按照 $\sqrt{V}\times\sqrt{V}$ 网格排列的顶点，生成将此网格中的顶点与其近邻随机连接的随机有向图，要求每条可能的边以概率 p 出现（见图 I.2）。确定如何设置 p，使得期望边数为 E。像练习 17.64 中所述测试你的程序。

○ 17.72　对练习 17.71 中的程序进行扩展，增加 R 条额外随机边，计算如程序 17.7 所示。对于较大的 R，收缩网格使边的总数仍约为 V。

● 17.73　试编写一个程序，要求在平面上生成 V 个随机点，然后构建一个图，其中边是由连

接一给定距离 d 内的所有点对而成（见图 17-13 和程序 3.20）。确定如何选择 d，使得期望边数为 E。像练习 17.64（低密度）和练习 17.65（高密度）中所述测试你的程序。

• **17.74** 试编写一个程序，要求用单位区间生成 V 个随机区间，所有区间长度都为 d，然后构建一个相应的区间图。确定如何选择 d，使得期望边数为 E。像练习 17.64（低密度）和练习 17.65（高密度）中所述测试你的程序。

• **17.75** 试编写一个程序，从练习 17.63 中找到的实际图中选择 V 个顶点和 E 条边。像练习 17.64（低密度）和练习 17.65（高密度）中所述测试你的程序。

○ **17.76** 定义运输系统的一种方法是利用一组顶点序列，每个序列定义一条连接顶点的路径。例如，序列 0-9-3-2 定义了边 0-9、9-3 和 3-2。试编写一个程序，要求由一个输入文件建立一个图，文件中每行有一个使用符号名的序列。建立适当的输入从而可以利用程序来构建一个对应于巴黎地铁系统的图。

17.77 对练习 17.76 描述的解决方案进行扩展，使其包含顶点坐标，采取练习 17.59 中思路，从而可以进行图形化表示。

○ **17.78** 将练习 17.33 ~ 17.35 中描述的变换应用到各种图（见练习 17.63 ~ 17.76）中，并制表表示出每次变换所删除的顶点数和边数。

○ **17.79** 设计一个合适的扩展，从而可以使用程序 17.1 来构建分离度图，而不必对每条隐式边都调用一个函数。也就是说，构建图所需要的函数调用次数应该与各组规模之和成正比。（图处理函数的实现留下了一个有效处理隐式边的问题。）

17.80 对于有 N 个不同组且组内有 k 个人的分离度图，试给出其边数的紧致上界。

▷ **17.81** 按照图 17-16 的风格绘图，其中 V 个顶点分别编号为 $0 \sim V-1$，而且每个顶点 i 与 $\lfloor i/2 \rfloor$ 均有一条边，$V = 8$，16 和 32。

17.82 修改程序 17.1 中的 ADT 接口，从而允许客户程序使用符号顶点名和边作为通用 Vertex 类型的实例对。对用户完全隐藏顶点索引表示和符号表 ADT 使用。

17.83 在练习 17.82 的 ADT 接口中增加一个函数，从而支持图的 join 操作，并给出邻接矩阵表示和邻接表表示的实现。注意：无论哪个图中的顶点或边都应该在 join 中，但在两图中的均出现的顶点只在 join 中出现一次，而且应该删除平行边。

17.7 简单路径、欧拉路径和哈密顿路径

我们的第一个图处理算法解决了图中涉及路径的基础性的问题。它们引入了我们在本书中使用的通用递归范型，并且它们说明了看上去类似的图处理算法在难度上会有很大的不同。

这些问题涉及如边的存在性、顶点的度等局部性质，还涉及告知图结构之类的全局性质。最基本的性质是判断两个顶点是否是连通的。

简单图 给定两个顶点，图中是否存在一条将它们连接起来的简单路径呢？在某些应用中，我们可能只需知道路径的存在性就满足了；而在另一些应用中，我们可能需要能找出一条特定路径的算法。

程序 17.11 是找出一条路径的直接解法。它基于*深度优先搜索*（depth-first search），这是一个我们在第 3 章和第 5 章简略讨论的基本图处理范型，在第 18 章我们将进行详细研究。该算法是基于递归算法，通过检查对于依附于 v 的每条边 v-t，是否存在从 t 到 w 的一条不经过 v 的简单路径，来确定从 v 到 w 是否存在简单路径。它使用顶点索引的数组对 v 进行标记，从而在递归调用中不再检查通过 v 的路径。

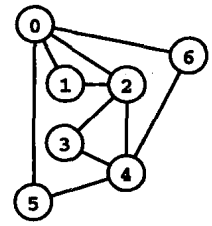
程序 17.11 中的代码简单测试一条路径的存在性。我们如何对它进行修改，从而打印出

路径上的边呢？可以通过递归思靠得到一个简单的解：

- 在 pathR 中的递归调用找到一条从 t 到 w 的路径之后，添加一条打印 t-v 的语句。
- 在 GRAPHpath 中，交换调用 pathR 中的 w 和 v。

如果只做第一步，则所打印出来的从 v 到 w 的路径是逆序的：如果调用 pathR（G, t, w）找到一条从 t 到 w 的路径（并且按照逆序打印出路径上的边），那么打印 t-v 就完成了从 v 到 w 的路径的工作。第二个修改是将顺序颠倒过来：为了打印出从 v 到 w 的路径上的边，我们以逆序打印出从 w 到 v 的路径上的边。（这一技巧不适合于有向图）。我们可以使用同一种策略来实现 ADT 函数，对于每条路径上的边调用客户提供的函数（见练习 17.87）。

图 17-17 给出了递归动态性的一个例子。有了递归程序（实际上，任何存在函数调用的程序都适用），这样的轨迹很容易实现：对程序 17.11 进行修改来产生一个轨迹，我们可以增加一个变量 depth，在进入时增 1，退出时减 1，以记录递归的深度，然后在递归函数的开始增加一个代码，打印出 depth 个空格，其后再加上适当信息（见练习 17.85 和 17.86）。




```
2-0 pathR(G, 0, 6)
  0-1 pathR(G, 1, 6)
    1-0
    1-2
  0-2
  0-5 pathR(G, 5, 6)
    5-0
    5-4 pathR(G, 4, 6)
      4-2
      4-3 pathR(G, 3, 6)
        3-2
        3-4
      4-6 pathR(G, 6, 6)
```

图 17-17　简单路径搜索的轨迹

此轨迹显示了程序 17.11 中的递归操作调用 pathR（G, 2, 6）来找出上图中从 2 到 6 之间的一条简单路径的过程。对于考虑的每条边轨迹中都有一条线，每次递归调用缩进一层。要考察 2-0，我们调用 pathR（G, 0, 6），致使我们考察 0-1，0-2 和 0-5。考察 0-1，我们调用 path（G, 1, 6），致使我们考察 1-0 和 1-2，不再能够递归调用，因为 0 和 2 已被标记。对于此例，该函数发现路径 2-0-5-4-6。

程序 17.11 主要关注检查与一个给定顶点相邻的所有边。如我们已看到的，如果使用邻接表表示稀疏图，或者使用我们讨论的任何一种变型，这个操作也很容易实现（见练习 17.89）。

性质 17.2　*可以在线性时间找出图中连接两个给定顶点的一条路径。*

证明　程序 17.11 中的递归深度优先搜索直接蕴含着一个使用归纳法的证明，该 ADT 函数确定一条路径是否存在。这样的证明很容易扩展到确定在最坏情况下，程序 17.11 对于邻接矩阵中的所有元素只检查一次。类似地，我们可以表明在最坏情况下，邻接表的类似程序对于图中的所有边只检查两次（在每个方向检查一次）。■

在图算法中，我们使用术语线性（linear）来表明其值是在图的规模 $V+E$ 的一个常量因子内。如在 17.5 节末尾所讨论的，这个值也是在图的表示规模的常量因子范围之内。性质 17.2 表明，该结论允许使用稀疏图的邻接表表示和稠密图的邻接矩阵表示，这是一般的做法。对于一个使用邻接矩阵且运行时间与 V^2 成正比算法，使用线性来描述该算法是不合适的（即使它是图表示大小的线性函数），除非图是稠密图。实际上，如果我们使用邻接矩阵来表示稀疏图，那么对于要求检查所有边的图处理问题，无法得到它的一个线性时间的算法。

在下一章中我们将从更一般的角度详细讨论深度优先搜索算法，而且还会考虑几个其他的连通性算法。例如，程序 17.11 的一个更为通用的版本，它给出了一种遍历图中所有边的方法，构建了一个顶点索引数组，允许客户在常量时间内测试是否存在连接任何两个顶点的一条路径。

性质 17.2 可能过分估计了程序 17.11 的实际运行时间，因为它可能在只检查几条边之后就找到一条路径。此时，我们只对在线性时间内保证找出连接图中的任何两个顶点的一条路径的方法感兴趣。对比之下，出现的类似其他问题的求解要困难得多。例如，考虑如下问题，我们要找出连接顶点对的路径，但是还要加上限制条件：访问图中的所有其他顶点。

程序 17.11　简单路径搜索（邻接矩阵）

函数 GRAPHpath 检查连接两个给定顶点的一条简单路径的存在性。它使用了递归深度优先搜索函数 pathR，该函数对于给定的两个顶点 v 和 w，检查与 v 相邻的每条边 v-t，以确定它是否可能是到 w 的路径上的第一条边。顶点索引的数组 visited 保持对任何顶点修改的功能，因而能保证路径是简单的。

为使函数打印出路径上的边（按照逆序），恰好在 pathR 结束附近返回 1 之前，添加语句 `printf("%d-%d", t, v);`（见正文）。

```
static int visited[maxV];
int pathR(Graph G, int v, int w)
  { int t;
    if (v == w) return 1;
    visited[v] = 1;
    for (t = 0; t < G->V; t++)
      if (G->adj[v][t] == 1)
        if (visited[t] == 0)
          if (pathR(G, t, w)) return 1;
    return 0;
  }
int GRAPHpath(Graph G, int v, int w)
  { int t;
    for (t = 0; t < G->V; t++) visited[t] = 0;
    return pathR(G, v, w);
  }
```

哈密顿路径　给定两个顶点，是否存在连接它们的一条简单路径，使得访问图中的每个顶点一次且只有一次？如果这条路径是从一个顶点又返回到自身，那么称此问题为哈密顿回路问题（Hamilton tour problem）。如图 17-18 所示。是否存在一条路径访问图中每个顶点一次且只一次？

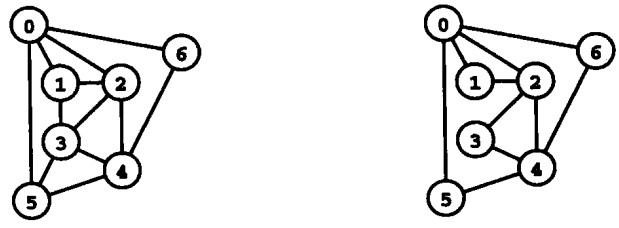


图 17-18　哈密顿回路

左图中含有哈密顿回路 0-6-4-2-1-3-5-0，访问每个顶点有且只有一次，并返回起始顶点，但是右图中不存在哈密顿回路。

初看起来，这个问题似乎是有一种简单解：我们可以编写一个找哈密顿路径问题的简单递归程序（见程序 17.12）。但这个程序对很多图不适用，因为它的最坏情况下的运行时间与图中顶点数呈指数（exponential）关系。

程序 17.12　哈密顿路径

此函数 GRAPHpathH 查找从 v 到 w 的一条哈密顿路径。它使用不同于程序 17.11 中的递归函数，不同之处有两方面：第一，仅当它找到一条长为 V 的路径时才成功返回；第二，在不成功返回之前，重新设置 visited 标记。除非对于规模较小的图，不要期望此函数会结束。

```
static int visited[maxV];
int pathR(Graph G, int v, int w, int d)
  { int t;
    if (v == w)
      { if (d == 0) return 1; else return 0; }
    visited[v] = 1;
    for (t = 0; t < G->V; t++)
      if (G->adj[v][t] == 1)
        if (visited[t] == 0)
          if (pathR(G, t, w, d-1)) return 1;
    visited[v] = 0;
    return 0;
  }
int GRAPHpathH(Graph G, int v, int w)
  { int t;
    for (t = 0; t < G->V; t++) visited[t] = 0;
    return pathR(G, v, w, G->V-1);
  }
```

性质 17.3　*哈密顿回路的递归搜索需要指数时间。*

证明　考虑一个图，其中顶点 $V-1$ 孤立的，而边与其他 $V-1$ 个顶点构成一个完全图。程序 17.12 将找不到一条哈密顿路径，但是由归纳法易得它会检查完全图中的所有 $(V-1)!$ 条路径，每次检查都涉及 $V-1$ 个递归调用。因此，全部递归的次数大约为 $V!$，或 $(V/e)^V$，这要大于任何常数的 V 次方。 ■

程序 17.11 找出简单路径和程序 17.12 找出哈密顿路径的实现是极其相似的例子，这两个程序都在 visited 数组中的所有元素被设置为 1 时终止。为什么运行时间有如此的不同？程序 17.11 保证可以很快地完成，因为每当 pathT 被调用时，至少会设置 visited 数组中的一个元素为 1。而程序 17.12 可能将 visited 数组中的元素重新设置为 0，因而我们不能保证它很快会结束。

在程序 17.11 中搜索简单路径时，我们知道，如果存在从 v 到 w 的一条路径，那么可以通过取从 v 出发的一条边 v-t，找到这条路径，同样对于哈密顿路径也成立。但只是存在相似性而已。如果无法找到从 t 到 w 的一条简单路径，那么我们可以得出结论，从 v 到 w 不存在经过 t 的简单路径；但是对于哈密顿的类似情况并不成立。可能是这样的一种情况，不存在从 v-t 开始的到 w 的哈密顿路径，但是对于某个其他顶点 x，存在从 v-x-t 开始的哈密顿路径。我们必须从 t 进行一个递归调用，这对应着从 v 通向 t 的每条路径。简言之，我们可能必须检查图中的每一条路径。

对于一个阶乘时间的算法，值得考察一下它有多慢。如果我们可以在 1 秒之内处理一个有 15 个顶点的图，该算法会花费 1 天来处理 19 个顶点的图，花费 1 年来处理一个 21 个顶点的图，用 6 个世纪以上来处理一个 23 个顶点的图。快速计算机也无济于事。一个比原计算机快 200 000 倍的计算机需要 1 天多的时间来解 23 个顶点的问题。处理有 100 或 1 000 个

顶点的图的开销则大得无法想象。更不用说我们在实际中可能遇到的图的规模。对于一个包含数百万个顶点的图，仅仅是在本书中写出处理这样一个图需要的世纪数就会用上百万页。

图 17-19 显示了程序 17.12 检查哈密顿回路的搜索轨迹。

在第 5 章中，我们考察大量简单递归程序，它们在特征上与程序 17.12 具有相似之处，但是使用自顶点向下的动态规划算法，在性能上会有显著改进。该递归程序的特征完全不同：必须保存的中间结果的数目是指数级的。虽然很多人对哈密顿问题进行了大量研究，对于较大规模（甚至是中等规模的）图，还没有人能够找到此问题的具有合理性能的一个算法。

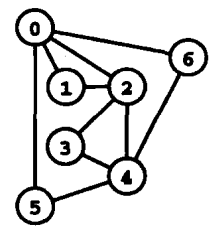
现在，假设改变限制，既不要求访问所有顶点，而是要求访问所有边。这个问题是不是像查找一条简单路径那样简单呢，或者会不会像查找哈密顿回路那样极为困难呢？

欧拉路径　是否存在连接两个给定顶点的一条路径，使得使用图中的边有且只有一次？路径不必是简单的，但顶点可能被访问多次。如果这条路径是从一个顶点又回到该顶点，我们就得到欧拉回路（Euler tour）问题。是否存在一条路径使用图中的每条边有且只有一次？我们在性质 17.4 的推论中证明：此路径问题等价于添加一条连接两个顶点的边的图的回路问题。图 17-20 给出了两个较小的例子。

这个经典问题由欧拉（L. Euler）于 1936 年首次开始研究。实际上，对于图和图论的研究可以追溯到欧拉对此问题的研究，从一个称为哥尼斯堡桥（bridge of Königsberg）问题（见图 17-21）的特例开始。瑞士的哥尼斯堡桥有七座连接河岸和小岛的桥，城镇中的人们发现，他们无法走过所有桥同时又不重复走过每座桥。他们的问题就引出了欧拉回路问题。

猜谜爱好者非常熟悉这些问题。这样的谜题很常见，比如要求一笔画出一个给定的图形，同时满足必须从特定的点开始和结束的约束条件。在开发一个图处理算法时，很自然地会考虑欧拉路径，因为欧拉路径是图的一种有效表示（将边置以特定的顺序），我们可以将它作为开发有效算法的基础。

欧拉路径说明很容易确定是否存在一条路径，因为我们需要做的只是检查每个顶点的度。这个性质也很容易阐述和应用，但是其证明在图论中却很困难。




```
0-1
  1-2
    2-3
      3-4
        4-5
        4-6
    2-4
      4-3
      4-5
      4-6
0-2
  2-1
  2-3
    3-4
      4-5
      4-6
  2-4
    4-3
    4-5
    4-6
0-5
  5-4
    4-2
      2-1
      2-3
    4-3
      3-2
        2-1
    4-6
0-6
```

图 17-19　哈密顿回路搜索轨迹

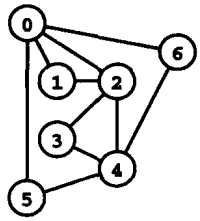
此轨迹显示了当发现上图所示的图中不存在哈密顿回路时，程序 17.12 所检查的边。为简明起见，这里忽略了已标记顶点的边。

性质 17.4　一个图含有欧拉回路，当且仅当它是连通的。而且所有顶点都有偶数度数。

证明　为简化证明，我们允许自环和平行边，虽然不难修改证明以使这个性质对于简单图也成立（见练习 17.94）。

如果一个图中有欧拉回路，那么该图必定是连通的，因为该回路定义了一条连接每对顶点的路径。同样，任何给定的顶点 v 必定有偶数度数，因为当我们在回路上遍历时（从任何

其他顶点开始），我们从一条边进入 v，离开 v 则在另一条边（这两条边都不会再在这条回路中出现）；因此在遍历这条回路时，依附于 v 的边数必定是我们访问 v 的次数的两倍。这是一个偶数。



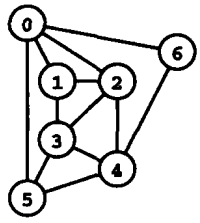
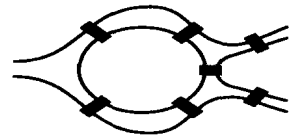


图 17-20　欧拉回路和路径示例

上图存在欧拉回路0-1-2-0-6-4-3-2-4-5-0，所有边只使用了一次。下图中不含欧拉回路，但它存在欧拉路径1-2-0-1-3-4-2-3-5-4-6-0-5。



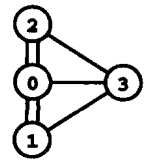


图 17-21　哥尼斯堡桥问题

欧拉所研究的一个著名问题是在哥尼斯堡小镇，那里有一个小岛，位于 Pregel 河的交叉处。那里有七座桥将此岛与河的两岸以及交叉口连接起来，如上图所示。是否存在一种方法连续地走过这七座桥，同时又不重复走过每座桥呢？如果将小岛标号为0，河岸标号为1和2，交叉口之间的标号为3，并将每座桥定义为一条边，就可以得到下图所示的多重图。这个问题就是要找出一种穿越此图的路径，满足每一条边仅使用一次且只一次。

我们对边数使用归纳法证明这个结果。结论对于不含边的图肯定成立。考虑任一有多于一条边的连通图，其中所有顶点的度为偶数。假设从任一顶点 v 开始，我们沿着任意一条边行进，并删除这条边。继续这样做，直到到达没有更多边的一个顶点。这个过程必定终止，因为我们每一步都会删除一条边，但是可能的结果会是什么呢？图 17-22 描述了一些例子。立即可以得出，我们必将回到 v，因为如果我们结束在一个不同于 v 的顶点，当且仅当该顶点的度数为奇数。

一种可能是我们已经遍历了整个回路；如果是这样，那么就得证。否则，图中余下的所有顶点仍有偶数度数，但可能不是连通的。但由归纳假设每个连通分量有一个欧拉回路。此外，才删除的回路路径将这些（连通分量中的）欧拉回路连接在一起形成原始图的欧拉回路：遍历回路路径，并进行迂回，求通过删除回路路径上的边所定义的连通分量的欧拉回路（每次迂回是一条真正的欧拉路径，可使我们返回到开始时的回路路径上的顶点）。注意到迂回可能多次触及回路路径（见练习 17.99）。在这种情况下，我们只迂回一次（比如说，当首次遇到时）。 ■

程序 17.13　欧拉路径的存在性

基于性质 17.4 中的推论，此函数使用练习 17.40 中的 GRAPHdeg ADT 函数，测试在一个连通图中从 v 到 w 是否存在一条欧拉路径。它需要的时间与 V 成正比，不包含检查连通性和使用 GRAPHdeg 构建顶点度表的预处理时间。

```
int GRAPHpathE(Graph G, int v, int w)
  { int t;
    t = GRAPHdeg(G, v) + GRAPHdeg(G, w);
    if ((t % 2) != 0) return 0;
```

```
  for (t = 0; t < G->V; t++)
    if ((t != v) && (t != w))
      if ((GRAPHdeg(G, t) % 2) != 0) return 0;
  return 1;
}
```

推论　一个图含有欧拉路径，当且仅当它是连通的，并且其中只有两个顶点的度为奇数。

证明　对于通过增加一条边来连接两个奇数度数的顶点（路径的端点）而构成的图，这个推论与性质 17.4 等价。■

因此，例如，没有人能够连续走过哥尼斯堡的所有桥而不倒退，因为在相应图中的所有 4 个顶点的度数都为奇数（见图 17-21）。

如在 17.5 节中所讨论的，对于邻接表或边集表示，我们可以在与 E 成正比的时间内找出所有顶点的度，对于邻接矩阵表示所需时间则与 V^2 成正比，或者我们也可以维护一个顶点索引数组，其中顶点度作为图表示的一部分（见练习 17.40）。给定数组，我们可以在与 V 成正比的时间内检查性质 17.4 是否得到满足。程序 17.13 实现了这个策略，并且证实了确定是否一个给定的图存在欧拉路径是一个简单的计算问题。这一点很重要，因为我们的直觉往往是：此问题应该不比确定一个给定的图是否存在哈密顿路径更简单。

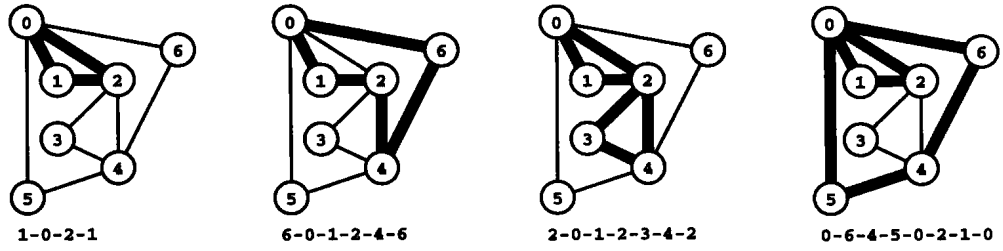


图 17-22　部分路径

沿着图中存在欧拉回路的任何一个顶点的边，总是可以回到那个顶点，如图中的例子所示。回路不必使用图中的所有边。

现在假设我们确实希望找出一条欧拉路径。对此我们如履薄冰，因为直接递归实现（尝试每条边，然后进行递归调用来找出余图中的一条路径，以此找到整个图的路径）将与程序 17.12 有同样的阶乘时间性能。我们不希望忍受这种性能，因为检查一条路径是否存在很容易，因而我们寻找更好的算法。可以使用一个固定开销测试来确定是否使用一条边（而不是递归调用的未知开销）来避免阶乘时间的膨胀。我们将这种方法留作练习（见练习 17.96 和练习 17.97）。

程序 17.14　线性时间欧拉路径（邻接表）

此函数 pathEshow 打印 v 和 w 之间的一条欧拉路径。使用常量时间实现 GRAPHremoveE（见练习 17.44），此函数运行时间为线性时间。辅助函数 path 跟踪并删除回路上的边，并将顶点压入栈中，以用于检查旁环（见正文）。只要存在带有旁环的顶点需要遍历，主循环就调用 path。

```
#include "STACK.h"
int path(Graph G, int v)
  { int w;
    for (; G->adj[v] != NULL; v = w)
```

```
      {
        STACKpush(v);
        w = G->adj[v]->v;
        GRAPHremoveE(G, EDGE(v, w));
      }
    return v;
  }
void pathEshow(Graph G, int v, int w)
  {
    STACKinit(G->E);
    printf("%d", w);
    while ((path(G, v) == v) && !STACKempty())
      { v = STACKpop(); printf("-%d", v); }
    printf("\n");
  }
```

另一种方法可由性质 17.4 的证明得出。遍历一个回路，删除遇到的边，并将它遇到的顶点压入栈中，因而（i）我们可以回溯此路径，并打印出它的边；（ii）对于另外的旁路（它可被接入到主路径上），可以检查每个顶点。这个过程如图 17-23 所示。

对于邻接表图 ADT，程序 17.14 是根据这些思路的一种实现。它假设存在欧拉路径，并且撤销图的表示；因此，客户在合适时有必要使用 GRAPHpathE、GRAPHcopy 和 GRAPHdestroy。代码有些技巧，初学者可能希望在接下来的几章的图处理算法有更多的理解，再来理解此代码。我们将它放在这里的目的是表明好的算法和巧妙的实现对于解决图处理问题更有效。

性质 17.5　*如果一个图中存在欧拉回路，则可以在线性时间内找到这条回路。*

证明　我们将完整的归纳证明留作一个练习（见练习 17.101）。非形式地，在第一次调用 path 之后，栈中包含从 v 到 w 的一条路径，并且余图（去除孤立顶点后）由较小的连通分量组成（它至少与目前找到的路径共享一个顶点），这些连通分量也存在欧拉路径。我们将孤立顶点从栈中弹出，并采用同样的方法使用 path 来找出含有非孤立顶点的欧拉回路。图中的每条边被压入（弹出）栈中只有一次，因而，总的运行时间与 E 成正比。 ■

虽然欧拉回路是作为一种遍历所有边和顶点的系统方法出现的，我们在实际中很少使用它，这是因为很少有图包含此路径。我们一般使用深度优先搜索来探索一个图，有关内容将在第 18 章中详细描述。实际上，如我们将看到的，在一个无向图中进行深度优先搜索相当于计算一条双向欧拉路径：遍历每一条边只有两次的路径，其中在每个方向各一次。

总之，本节我们已经看到了在图中找到简单路径是很容易的，较之于要了解是否可以遍历一个较大图中的所有边而不重新访问其中任何一条边（通过之检查所有顶点的度是否为偶数）来说，前者更简单。而且还有一种巧妙的算法来找到这样的一条回路；但在实际中，我们不可能知道是否能够遍历图中的所有顶点，而不重新访问其中任何一个顶点。这些问题都有简单的递归解法。但是运行时间上的潜在指数增长使得这些解法在实际中毫无意义。有人经过深思熟虑，给出了快速实用算法。

这些例子所描述的显然很类似的问题之间的难度存在很大变化，这在图处理中具有代表性。正如 17.8 节中简要介绍和第 8 部分详细介绍的那样，我们必须承认在一些需要指数时间（如哈密顿回路问题以及其他一些常见问题）的问题和一些我们能够保证在多项式时间内找到解法的问题（如欧拉回路问题以及其他一些常见问题）之间，存在不可逾越的障碍。在本书中，我们的主要目标是为后一类问题开发高效的算法。

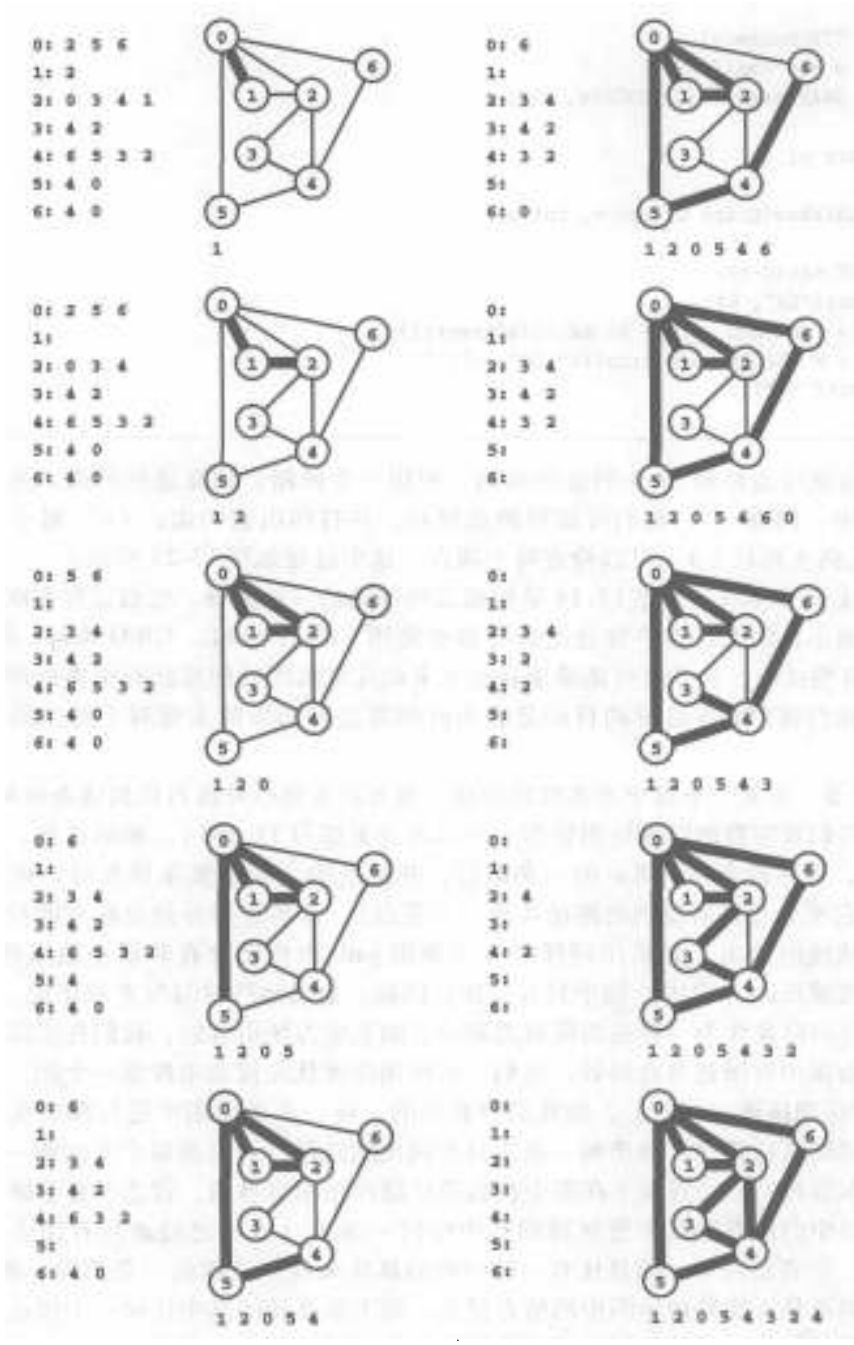


图 17-23 去除回路寻找欧拉路径

此图显示了程序 17.14 如何在一个示例图中发现一条从 0 返回 0 的欧拉回路。粗的黑边为回路中的边，栈中的内容显示在每个图的下面，对于未在回路中的边，其邻接表在左边显示。

第一，此程序向回路中添加边 0-1，并将它从邻接表中删除（要在两处删除）（左上图，邻接表在其左边）。第二，以同样的方式在回路中加入边 1-2（左边自上第二个图）。接下来，它会回到 0，但是继续对另一个回路 0-5-4-6-0 进行处理，再次回到 0，且不再有更多依附于 0 的边（右边自上第二个图）。然后，将孤立顶点 0 和 6 从栈中弹出，直到 4 位于栈顶，并从 4 开始一条回路（右边自上第三个图）。经过 3，2，并回到 4。在此它将所有目前孤立的顶点 4、2、3 弹出，依此类推。从栈中弹出的顶点序列定义了整个图的欧拉回路 0-6-4-2-3-4-5-0-2-1-0。

练习

▷ **17.84**　按照图 17-17 的格式，当程序 17.11 在下图中找一条从 0 到 5 的路径时，显示其递归调用的轨迹（以及所跳过的顶点）。图为

```
3-7  1-4  7-8  0-5  5-2  3-8  2-9  0-6  4-9  2-6  6-4
```

17.85　修改程序 17.11 中的递归函数，使用正文中描述的一个全局变量，打印出类似图 17-17 的一条轨迹。

17.86　通过向递归函数中添加一个参数来记录深度，重新完成练习 17.85。

17.87　使用正文中描述的方法，给出 GRAPHpath 的一种实现，对于从 v 到 w 的路径（如果路径存在的话）上的每条边，调用客户提供的函数。

○ **17.88**　修改程序 17.11，使其有第 3 个参数 d，并测试是否存在一条连接 u 和 v 且长度大于 d 的路径。特别是，GRAPHpath（v，v，2）应该非零，当且仅当 v 在环上。

▷ **17.89**　修改 GRAPHpath，使其使用邻接表图表示（程序 17.6）。

● **17.90**　对于各种图，进行实验来确定程序 17.11 找出两个随机选择的顶点之间的一条路径的概率（见练习 17.63 ~ 17.76），并计算对于每种图所找到路径的平均长度。

○ **17.91**　试考虑由以下 4 组边定义的图：

```
0-1  0-2  0-3  1-3  1-4  2-5  2-9  3-6  4-7  4-8  5-8  5-9  6-7  6-9  7-8
0-1  0-2  0-3  1-3  0-3  2-5  5-6  3-6  4-7  4-8  5-8  5-9  6-7  6-9  8-8
0-1  1-2  1-3  0-3  0-4  2-5  2-9  3-6  4-7  4-8  5-8  5-9  6-7  6-9  7-8
4-1  7-9  6-2  7-3  5-0  0-2  0-8  1-6  3-9  6-3  2-8  1-5  9-8  4-5  4-7
```

其中哪些图存在欧拉回路？哪些图存在哈密顿回路？

○ **17.92**　请给出一个有向图存在一个（有向）欧拉回路的必要充分条件。

17.93　试证明每个连通的无向图都有一条双向的欧拉回路。

17.94　修改性质 17.4 中的证明，使其对于带有平行边和自环的图成立。

▷ **17.95**　说明如果再增加一座桥，就可以给出哥尼斯堡问题的一种解决方案。

● **17.96**　证明一个连通图中存在从 v 到 w 的路径，仅当其中存在一条依附 v 的边，当将其删除时会使图不连通（除了可能孤立 v 的情况）。

● **17.97**　使用练习 17.96 来开发一个一种有效的递归方法，找出含欧拉回路的图中的一条欧拉回路。除了基本的图 ADT 函数，你还可以使用 ADT 函数 GRAPHdeg（见练习 17.40）和 GRAPHpath（见程序 17.11）。针对邻接矩阵图表示，实现并测试你的程序。

17.98　开发程序 17.14 的一个独立于表示的版本，使用通过接口来访问与一个给定邻接的所有边（见练习 17.60）。注意：要当心你的代码和 GRAPHremove 之间的交互。确保你的实现在出现平行边和自环时也能正确工作。

▷ **17.99**　给出一个例子，在程序 17.14 中首次调用 path 之后，余图是不连通的（对于存在欧拉回路的图）。

▷ **17.100**　描述如何修改程序 17.14，使其能用于在线性时间内检测一个给定的图中是否存在欧拉路径。

17.101　对于正文中描述并在程序 17.14 中实现的线性时间的欧拉回路算法，试给出使用归纳法对该算法能正确地找出一条欧拉回路的一个完整证明。

○ **17.102**　对于 V 个顶点的图（对于在计算中可行的尽可能大的 V 值），找出其中存在欧拉回路的个数。

● 17.103　对于各种图进行实验研究，确定在程序 17.14 中首次调用 path 所找到路径的平均长度（见练习 17.63～76）。计算该路径为环的概率。

○ 17.104　编写一个程序，要求计算一个 2^n+n-1 位的序列，其中不存在有 n 个连续位匹配的两对。（例如，对于 $n=3$，序列 0001110100 具有此性质。）提示：找出 de Bruijn 有向图中的一条欧拉路径。

▷ 17.105　按照图 17-19 中的风格，当程序 17.11 在如下的图中找出一条哈密顿回路时，指出递归调用的轨迹（且跳过顶点）：

```
3-7  1-4  7-8  0-5  5-2  3-8  2-9  0-6  4-9  2-6  6-4
```

○ 17.106　修改程序 17.12，使其在找到哈密顿回路时打印出这条回路。

● 17.107　找出下图中的一条哈密顿回路或者指出不存在哈密顿回路。

```
1-2  5-2  4-2  2-6  0-8  3-0  1-3  3-6  1-0  1-4  4-0  4-6  6-5  2-6
6-9  9-0  3-1  4-3  9-2  4-9  6-9  7-9  5-0  9-7  7-3  4-5  0-5  7-8
```

●● 17.108　对于 V 个顶点的图（对于在计算中可行的尽可能大的 V 值），找出其中存在哈密顿回路的个数。

17.8　图处理问题

有了在本章中所开发的基本工具，在第 18～22 章中我们将考虑用各种算法来解决图处理问题。这些算法都是很基础的算法，在许多应用中都很有用。但是它们只是图算法这个研究领域的一个入门。很多已经开发的有趣且有用的算法都不在本书的讨论范围内，而且还有很多有趣的问题已经得到解决，但尚未为之发明好的算法。正如在任何领域一样，我们面临的第一个挑战是确定求解一个给定的问题是多么困难。对于图处理问题，确定这个难度远比我们想象的难得多。即使是对于看上去简单求解的问题。而且，我们的直觉并不总是能够帮助我们将简单问题从复杂问题或至今尚未解决的问题中识别出来。在这一节里，我们简略描述重要的经典问题，并阐述对这些问题的认识程度。

给定一个新的图处理问题，在开发求解该问题的一个实现中，我们要面临何种挑战呢？令人遗憾的是，对于我们可能遇到的任何情况，没有一种好的方法能够对这个问题做出解答，但对于各种经典的图处理问题确实可以提供其解决难度的一般性的描述。为此，我们按照求解它们的难度，将其粗略地分为如下几类：

- 简单
- 易解
- 难解
- 未知

这些术语旨在反映出相对于彼此的信息，以及目前对相应图算法的认知程度。

正如这些术语所指出的，之所以用这种方式来对问题分类，其主要原因在于：存在许多诸如哈密顿回路的图问题，没有人知道如何对它们有效地求解。我们最终会看到（在第八部分）如何以一种准确的形式使这个结论更有意义。但在目前，至少需要注意，要编写一个解决这类问题的高效程序会遇到很大的障碍。

我们将把许多图处理问题的充分描述留待本书后面介绍。这里将对容易理解的内容作简要说明，以此引入对图处理问题难度进行分类的一般性的问题。

一个容易的（easy）图处理问题是指，可以用某种精致而有效的小程序来解决的问题。

我们在第 1 ~ 4 部分已经对这些小程序有所熟悉。对于简单问题，我们常常寻求最坏情况下运行时间为线性或者为顶点数或边数的低次多项式的算法。一般而言，就像我们在其他领域的做法，可以确定虽然开发简单问题的一个蛮力解可能对于大规模的图来说可能较慢，但对于小规模甚至是中等规模的图则很有用。然后，一旦知道了该问题是简单的，就可以寻求在实际中使用的高效解法，并试图找出其中最好的解法。我们在 17.7 节考虑的欧拉回路问题就是这种问题的一个主要例子。在第 18 ~ 22 章中我们还会看到其他一些例子，其中包括如下的一些最为著名的问题。

简单连通性　一个给定图是连通的吗？也就是说，是否每对顶点之间都存在一条彼此连接的路径？图中是否存在环，或者它是否为森林？给定两个顶点，它们在一个环上吗？我们在第 1 章的最初考虑过这些基本图处理问题。在第 18 章讨论了这些问题的很多解法。一些解法很容易就在线性时间内实现；另一些有相对复杂的线性时间解法则需要进行仔细地研究。

有向图中的强连通性　对于有向图，是否存在连接图中每对顶点的有向路径？给定两个顶点，它们是否通过两个方向的有向路径连接（它们在有向环上吗）？实现这些问题的高效解法要比相应的无向图中的简单连通性问题更具有挑战性。第 19 章的大部分内容致力于这些问题的研究。虽然在求解这些问题中有些复杂，但我们还是将这些问题归为简单问题，因为我们可以为之编写一个简洁、高效且有用的实现。

传递闭包　有向图中，从每个顶点沿着有向边可达那些顶点？这个问题与强连通性以及其他基本计算问题紧密相关。我们在第 19 章研究其经典的解法，其中仅包含数行代码。

最小生成树　在一个加权图中，找出连接所有顶点的具有最小权值的边集。这是最古老且得到深入研究的图处理问题之一；第 20 章的内容致力于求解这个问题的各种经典算法。研究人员仍在寻求此问题的快速算法。

单源点最短路径　在加权有向图（网）中，将一个给定顶点 v 与图中的各个顶点相连的最短路径是什么？第 21 章致力于这个问题的研究。此问题在众多的应用中极其重要。如果边的权值可以为负值，那么这个问题绝对不简单。

易解的（tractable）图处理问题是这样的问题，求解它的算法的时间和空间需求可以保证由图的规模（$V+E$）的一个多项式函数所限定。所有简单问题都是易解问题，但是我们还要做出区分。因为很多易解问题有一种特征，开发求解此问题的一种高效实用程序仍然是一项极具挑战性的任务，这是很可能的事情。解决方案可能过于复杂，不能在本书中呈现。因为实现可能要求数百甚或数千行的代码。以下是这类最重要问题中的两个例子。

平面性　是否可以画出一个给定图，并且用来表示边的任何线段都不相交？我们可以自由地将定点放在任何位置，因此对于许多图这个问题都可以解决，但对于许多其他的图则不可能解决。有一个著名的经典结论，称为库拉托夫斯基定理（Kuratowski's theorem）。它提供了一种简单检测方法来确定一个给定的图是否是平面图：定理称，无法在没有边交叉的条件下画出的图是那些一定包含一些子图的图，删除度为 2 的顶点之后，这些子图与图 17-24 中所示的某个图同构。即使不考虑度为 2 的顶点，对于较大规模的图（见练习 17.111），直接实现这个检测也会很慢。然而，R. Tarjan 在 1974 年研制了一种精巧（但复杂）的算法，使用深度优先搜索机制，在线性时间内解决了这个问题，这种机制对我们在第 18 章中讨论的机制作了扩展。Tarjan 的算法并没有给出一种实际的布局；只是证实这样的布局存在。如在第 17.1 节中所讨论的，对于顶点并不与物理世界直接相关的应用，开发一个视觉合理的布

局成为一个挑战性的研究问题。

匹配　给定一个图，对于其边集的某些子集，其中任意两条边都不会连接到同一个顶点上，那么满足此性质的最大的边子集是什么？已经知道，这个经典问题可在与 V 和 E 的一个多项式函数成正比的时间内解决，但适合于大型图的快速算法仍然是一个难解的研究目标。如果对此问题在某些方面做出限制，该问题会容易一些。例如，将学生与选择机构中的可供职位相匹配，这就是一个*二分匹配*（bipartite matching）的例子：我们有两种不同类型的顶点（学生和职位），而且只关心将某种类型的顶点与另一种类型的顶点相连接的边。第 22 章将介绍此问题的一种解决方法。

对于一些易解问题，其解决方案从来没有被写为程序，而且由于其运行时间过高以至于我们无法在实际中使用其解法。以下例子即属于这一类。它还表明了图处理的难度在数学实现性上的多变性。

有向图中的偶环　一个给定的有向图中，是否存在一个长度为偶数的环？这个问题看上去容易解决，因为对于无向图的相应问题就很容易解决（见 18.4 节），而且另外一个相应的问题（即一个图中是否有长度为奇数的环）也同样容易解决。然而，多年以来，这个问题并没有得到充分的理解，我们甚至不知道是否存在一个解决这个问题的高效算法。确认存在高效算法的定理在 1999 年得到证明，但是这个方法太复杂，任何数学家或程序员都没打算实现它。

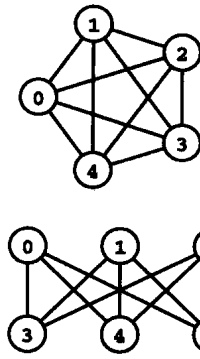


图 17-24　平面图中的禁止子图

这两个图不仅都不能在没有交叉边的情况下在平面上画出，而且对于将其中任意一个图作为子图的任何图（删除其中度为 2 的顶点后），同样也不能画出；但是所有其他图都可以在平面上画出。

第 22 章的一种重要主题是，很多易解的图问题最适合用那些求解整类问题的一般算法来处理。第 21 章中的*最短路径*（shortest path）算法、第 22 章中的*网络流*（network flow）算法，以及 22 章的强大的*网络单纯形*（network-simple）算法都是能够求解很多图处理问题的算法，如果不采用这些算法，那些问题将很难求解。这样问题的一些例子列举如下：

指派　这个问题也称为*二分加权匹配*（bipartite weighted matching）问题，它是一个在二分图中找出具有最小权值的最佳匹配问题。使用网络流算法很容易求解这个问题。目前已知有一些特定的方法可直接处理这个问题，但已证实这些方法基本上等价于网络流算法。

一般连通性　如果将图中的某条边删除，将把这个图分为不相交的两部分，那么这种边最少是多少（边连通性）？如果将图中的某些顶点删除，将把这个图分为不相交的两部分，那么这种顶点最少是多少（顶点连通性）？如我们在 22 章中看到的，这些问题，虽然难以直接求解，但都能使用网络流算法来解决。

邮差问题　给定一个图，找出一条边数最小的回路，要求图中的每条边至少使用一次（但允许多次使用边）。这个问题要比欧拉回路问题困难得多，但比哈密顿回路问题的难度要小一些。

从确认某个问题是易解问题，到提供可在实际中使用的软件，这确实是很困难的一步。一方面，在证明一个问题是易解时，研究人员期望去除在实现中需要处理问题的一些细节；另一方面，他们必须对多种可能的情况进行说明，即使这些情况可能并不在实际中出现。在考虑图算法时，理论和实践之间的冲突尤其尖锐。因为数学研究中充满了我们处理图时可能

需要考虑的各种结构性质的深奥结果，还因为这些结果与实际中出现的图的性质之间的关系很少被理解。开发像网络单纯形算法那样的一般模式一直是处理这些问题的极其有效的方法。

难解的（intractable）图处理问题是这样的问题，尚没有一个已知算法可以保证在合理时间求解此问题。这些问题都有一个特点，就是使用蛮力法通过穷尽所有可能性来求解问题。我们将这类问题看作难解问题，因为有太多种情况要考虑。这类问题很广泛，包括我们希望知道如何求解的很多重要问题。术语 NP-难（NP-hard）描述了此类问题。大多数专家认为对于这些问题不存在有效的算法，我们将在第八部分详细讨论此术语及看法的基础。在 17.7 节讨论的哈密顿回路问题就是 NP-难的图处理问题的一个例子，以下列出的问题也归为此类。

最长路径　在一个图中，连接两个给定顶点的最长简单路径是什么？尽管它看上去与最短路径问题相似，这个问题是哈密顿回路问题的一种版本，且是 NP-难的。

可着色性　是否存在一种方法可以使用 k 种颜色对图中的每个顶点着色，使得不存在连接两个着相同颜色顶点的边？这类问题对于 $k=2$ 时是简单的（见练习 18.4），但对于 $k=3$ 时则是 NP-难问题。

独立集　存在一些图顶点的子集，其中任何两个顶点都没有边相连，那么满足这种性质的最大顶点子集的规模如何？正如我们在与欧拉回路问题和哈密顿回路问题对比时所看到的，这个问题是 NP-难的，虽然它看上去与匹配问题很相似，但后者是在多项式内求解的问题。

团　对于一个给定的图，其中最大团（完全子图）的规模是多少？这个问题推广了部分平面性问题，因为如果最大团中有 4 个以上的顶点，那么这个图就不是平面的。

这些问题都形式化为*存在性*（existence）问题。问题是确定是否存在特定子图。有些问题是问某个特定类型的最大子图的规模，我们可以通过存在性问题的框架来解此问题，测试是否存在大小为 k 且满足此性质的子图，然后使用二叉查找找出最大子图。实际中，我们通常想要找出一个完整的解决方案。这可能要难得多。例如，著名的 4 色定理（four-color theorem）告诉我们：可能只使用 4 种颜色就能对一个平面图中的所有顶点着色，并满足不存在连接两个同色顶点的边。但是对于一个特定的平面图，定理并没有告诉我们如何对图中的顶点进行着色：知道一种着色存在并不能帮助我们找到此问题的一个完整的解决方案。另一个著名的例子是*旅行商*（salesperson）问题，要求找出经过一个加权图中所有顶点的最短路径长度。这个问题与哈密顿回路问题有关：但肯定不会比它简单：如果我们不能找出哈密顿回路问题的一个高效解，我们就不能期望找出旅行商问题的一个高效解。通常，在面对这些困难的问题时，我们先处理这些不能解决问题的最简单版本。对于存在性问题可以沿用通常的做法，但是它们在理论上也起着重要的作用，我们将在第 8 部分看到。

这里所列出的问题只不过是已证实的数以千计 NP-难问题中的几个。它们在各种类型的计算应用中都会出现；在图处理领域中它们尤其常见，因此在本书中我们必须知道存在这些问题。

注意我们一直坚持我们的算法在最坏情况下也能保证有效。也许应该考虑对特定输入高效的算法（未必是最坏情况）。类似地，很多问题都涉及*优化*（optimization）。但可能满足一条长路径（未必是最长）或一个大的团（未必是最大团）。对于图处理问题，很容易为实际出现的图找到一个好的答案，对于从未见过的假想图，我们甚至对于寻找一个找出此问题最

优解的算法不感兴趣。实际上，难解问题常常可以使用直接法或类似程序 17.12 的通用算法求解，虽然它们在最坏情况下有指数的运行时间，但对于实际中出现的某些问题实例，仍然可以快速求解（或找到一个好的近似解）。我们不愿使用一个将会崩溃或对于某些输入将产生不好结果的程序，但有时确实使用了一些对于特定输入在指数时间内运行的程序。我们将在第 8 部分讨论这种情况。

还有许多研究结果表明，即使放宽一些限制，各种难处理的问题仍然是难处理的。而且，还有许多我们无法解决的特定实际问题，因为没有人知道一个足够快的算法。在本书的这一部分中，我们将在遇到这些问题时将其标为 NP-难问题，并将此标记的含义解释为（至少如此解释）：我们不期望能够找到可以解决这些问题的高效算法，而且如果不使用高级技术，我们也不打算对其进行处理（除非可能使用蛮力法来解决小问题）。

还有一些图处理问题，其难度是*未知的*（unknown）。对于这些问题，既不知道其存在高效的算法，也未认定它们是 NP-难问题。随着我们对图处理算法和图性质了解的增多，其中一些问题会成为易处理问题，甚至成为简单问题。以下这个重要的自然问题，我们已经遇到过（见图 17-2），是此类中最著名的问题。

图同构　通过对顶点重命名，能否使两个给定的图相同？对于一些特殊类型的图，已经知道对此问题有一些高效的算法，但是一般问题的难度仍是未确定的。

与我们考虑过的其他类型的问题相比，内在难度未知的重要问题少之又少，因为在过去的数十年中已经对此领域做了深入的研究。在这一类问题中，有一些特定问题，比如同构问题有着极大的实际意义；此类中其他问题的意义则主要在于不能做其他分类。

对于简单算法类，我们习惯于将不同的最坏情况性能特性与算法进行比较，并试图通过分析和实验测试预测出性能。对于图处理，这些任务尤其困难，这是由于对于可能在实际中出现的各类图，要描述其特性存在着难度。幸运的是，许多重要的经典算法都有最优的或接近于最优的最坏情况性能，或者其运行时间仅取决于顶点数和边数，而不是依赖于图结构；因此我们可以关注实现的改进，并仍能自信地预测性能。

总之，对于图处理，已有大量的问题和算法。表 17-2 对我们已经讨论过的一些内容作了总结。对于不同类型的图（有向图、加权图、二分图、平面图、稀疏图和稠密图），每个问题都有不同的版本，而且有数以千计的问题和算法需要考虑。我们当然不能期望解决可能遇到的每个问题，而且某些看上去简单的问题可能连专家都无法解决。我们往往会自然地做出预计，认为把简单问题从难处理问题中区别出来应该没问题，但所讨论的许多例子都表明，对于某个问题，即使要将它大致分为某一类，也可能会变成一个重要的研究挑战。

随着我们对图和图算法了解的增加，给定的问题可能会从某一类变为另一类。尽管 20 世纪 70 年代进行了大量的研究活动，而且自此以后很多研究人员都做了深入的工作，但我们讨论的所有问题将来仍有可能归为“简单问题”（可以用某个简洁、高效且可能睿智的算法解决）。

上面已经对有关背景作了介绍，下面把重点放在各种实用的图处理算法上。我们可以解决的问题确实会经常出现，所研究的图算法在相当多的应用中都能很好地发挥作用，而且对于许多需要处理的其他问题，这些算法尽管不能保证有高效的解决方案，但仍可作为解决这些问题的基础。

表 17-2　经典图处理问题的难度

此表概述了正文中所讨论的各种图处理问题的相对难度，并对这些问题做了主观性比较。这些例子不仅表明问题难度的范围，而且表明对一个给定的问题进行分类是一个挑战性的任务。

	E	T	I	?
无向图				
连通性	*			
一般连通性		*		
欧拉回路	*			
哈密顿回路			*	
二分匹配	*			
最大匹配		*		
平面性		*		
最大团			*	
2-可着色性	*			
3-可着色性			*	
最短路径	*			
最长路径			*	
顶点覆盖			*	
同构				*
有向图				
传递闭包	*			
强连通性	*			
奇数长度环	*			
偶数长度环		*		
加权图				
最小生成树	*			
旅行商			*	
网				
最短路径（非负权值）	*			
最短路径（负权值）			*	
最大流	*			
指派		*		
最小成本流		*		

说明：E　容易（Easy）——已知的高效经典算法（见本部分参考文献）
T　易解的（Tractable）——解存在（实现困难）
I　难解的（Intractable）——没有已知的高效解（NP-难）
?　未知问题（Unknown）

练习

● **17.109**　试证明图 17-24 中描述的两个图都不是平面图。

17.110　编写一个邻接表表示（程序 17.6）的函数，确定是否一个图包含图 17-24 中描述的图。使用蛮力算法，测试团的 5 个顶点的所有可能子集以及完全二分图的 6 个顶点的所有可能子集。注意：这种测试并不足以说明一个图是否是平面图，因为它有可能忽略了一个条件，即在一些子图中删除度数为 2 的顶点可能会得出两个禁止子图中的一个。

17.111　画出下图，要求没有交叉边，或证明无法画出这样的图。

```
3-7  1-4  7-8  0-5  5-2  3-0  2-9  0-6  4-9  2-6
6-4  1-5  8-2  9-0  8-3  4-5  2-3  1-6  3-5  7-6
```

17.112　使用 3 种颜色，找出一种对下图进行着色的方式，满足不存在连接两个相同颜色的顶点的边，或证明无法实现这种着色。

```
3-7  1-4  7-8  0-5  5-2  3-0  2-9  0-6  4-9  2-6
6-4  1-5  8-2  9-0  8-3  4-5  2-3  1-6  3-5  7-6
```

17.113　对于下图求解独立集问题。

```
3-7  1-4  7-8  0-5  5-2  3-0  2-9  0-6  4-9  2-6
6-4  1-5  8-2  9-0  8-3  4-5  2-3  1-6  3-5  7-6
```

17.114　对于一个阶为 n 的 de Bruijn 图，其最大团的规模是多大？

第18章 图 搜 索

我们通常通过系统地检查图中的每个顶点和每条边来学习其性质。如果只是检查每条边（以任何一种顺序），那么确定一些简单的性质（例如，计算所有顶点的度数）是很容易的。图的许多其他性质都与路径有关，因而，学习它们的一种自然方式是沿着图的边从一个顶点移动到另一个顶点。我们考虑的几乎所有图处理算法都使用这个基本抽象模型。在这一章里，我们考虑基本的图*搜索*算法，并使用这种方法在图中移动，在此过程中学习图的结构性质。

这种方式的图搜索等价于探索迷宫。具体地说，迷宫中的通道对应图中的边，迷宫中通道交叉的点对应图中的顶点。当一个程序由于边 v-w 改变了从顶点 v 到顶点 w 的一个变量的值时，我们就将它看作迷宫中的一个人从点 v 移动到点 w。通过考察系统性的迷宫探索开始本章的讨论。与此过程对应，我们将看到基本的图搜索算法是如何通过图中的每条边和每个顶点的。

特别地，*递归深度优先搜索*（depth-first search，DFS）算法恰好对应于18.1节中的特定的迷宫探索策略。DFS是一种经典且使用广泛的算法，已经用于解决了连通性和很多其他的图处理问题。此基本算法有两种简单实现：一种是递归实现，另一种使用显式栈。用一个FIFO队列代替栈则得到另一个经典算法，*广度优先搜索*（breadth-first search，BFS），该算法用于解决与最短路径有关的另一类图处理问题。

本章的主要专题是DFS、BFS、相关算法以及它们在图处理中的应用。我们在第5章简要介绍了DFS和BFS；在此我们将从最基本的原则开始讨论。在对图搜索ADT函数的内容进行扩展后，将用于求解各种图处理问题，并使用它们来表明各种图算法之间的关系。特别是，我们将讨论一种通用的图搜索算法，其中包括很多经典的图处理算法，有DFS和BFS。

作为展示这些基本图处理方法用于解决更复杂问题的应用，我们考虑求解各种其他图处理问题的连通分量、双连通分量、生成树以及最短路径的算法。这些实现代表了我们在第19～22章将要用于求解更困难问题的典型方法。

本章最后讨论在图算法分析中所涉及的基本问题，这里的用例是比较图中求解连通分量数的几种不同算法。

18.1 探索迷宫

根据一个具有悠久历史且著名的等价问题（见本部分参考文献）来考虑搜索一个图的过程是有启发性的：找出通过由交叉点连接的通道所组成的迷宫的方式。本节提出了一种详细的基本方法研究，来探索任何给定迷宫的每一个通道。某些迷宫可用简单规则进行处理，但是大多数的迷宫都需要一种更为复杂的策略（见图18-1）。使用术语*迷宫*（maze）而不是*图*（graph），*通道*（passage）而不是*边*（edge），*交叉点*（intersection）而不是*顶点*（vertex），这只是语义上的区别，但是就目前而言，这样做会帮助我们更直观地认识问题。

自古以来，我们就知道探索迷宫而不迷路的技巧（至少可以追溯到特修斯和米诺托的传说），就是在我们身后抛开一个线球。这根线可以保证我们总是可以找到一个出口，但我们还关心确实能够探索迷宫的每一部分，而且除非万不得已，我们并不想走回头路。为了达到这些目标，就需要一些方法来标记到过的地方。也可以使用线来达到这些目的。但我们采用

另一种方法，可以更接近地为计算机实现建模。

假设每个交叉点都有灯，而且开始时这些灯都是关着的，另外在每个通道的两端都有门，开始时也是关着的。进一步假设这些门有窗户，而且灯光足够强，通道足够直，以至于在通道的一端开门就能确定通道另一端的交叉点是否是亮着的（即使另一端的门是关着的）。我们的目标是打开所有的灯和所有的门。为了达到这个目标，我们需要遵循一组规则。以下的迷宫探索策略，称为 Trémaux 探索，至少从 19 世纪以来就为人们所知了（见本部分参考文献）：

（ⅰ）如果在当前的交叉点没有关着的门，则转向第（ⅲ）步；否则，打开通向当前交叉点的任何通道上关闭的门（并使这扇门一直开着）。

（ⅱ）如果你能看到通道另一端的交叉点已经是亮着的，则尝试一下当前交叉点的另一扇门（第（ⅰ）步）。否则（如果看到通道另一端的交叉点上灯没开），则沿着通道走到该交叉点，边走边放线绳，开灯，并转到第（ⅰ）步。

（ⅲ）如果当前交叉点的所有门是打开的，则检查是否回到了起始点。如果是，就停止。否则，利用线绳返回到首次带你到该交叉点的通道上，边走边放线绳，并寻找另一扇关着的门（也就是说，返回到第（ⅰ）步）。

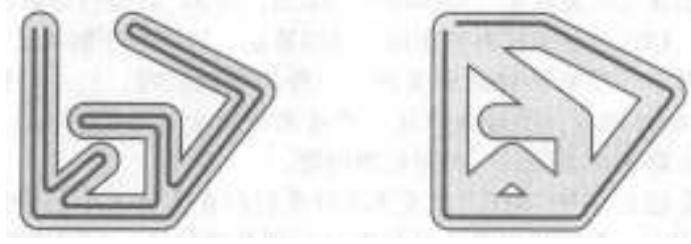


图 18-1　探索迷宫

我们可以遵循一个诸如“保持右手靠墙”的简单规则，探索一个简单迷宫中的每条通道。对于图中左面的迷宫，沿用这个规则可以探索整个迷宫，经历每个方向的每条通道一次。但是如果在一个带有环路的迷宫中使用这个规则，则可能回到起始点而无法探索整个迷宫。如图中右面的迷宫所示。

图 18-2 和图 18-3 描述了遍历一个简单图的过程，实际上表明，对于此例，每个灯都将打开，每个门也会打开。这些图只是显示了很多可能的探索中的一种，因为我们可以随意地按照任何顺序打开各个交叉点上的门。可以使用数学归纳法来表明这种方法总是高效的，这是一个很有意思的练习。

性质 18.1　*采用 Trémaux 迷宫探索，可以打开迷宫中的所有的灯和所有的门，并返回到起始位置。*

证明　为了使用递归法证明这个断言，首先注意到，显然对于只有一个交叉点且没有通道的迷宫断言成立，因为只需打开那个灯即可。对于包含不止一个交叉点的迷宫，假设此性质对于更少交叉点的所有迷宫都成立。证明能够访问所有交叉点就足够了，因为我们打开了所访问的交叉点的所有门。现在，考虑由第一个交叉点出发的第一条通道，并将交叉点划分为两个子集：（ⅰ）经过该通道可达且不需返回到起始位置的交叉点；以及（ⅱ）如果不返回起始点就不能由该通道到达的交叉点。由归纳假设可得，我们知道可以访问（ⅰ）中的所有交叉点（忽略返回起始交叉点的任何通道，该通道是亮的），并返回到起始点结束。然后，再次应用归纳假设可得，我们访问了（ⅱ）中的所有交叉点（忽略了（ⅰ）中从起始点到交叉点的通道，它们都是亮的）。■

为了使用归纳法证明 Trémaux 迷宫探索，图 18-4 中对迷宫进行了分解。

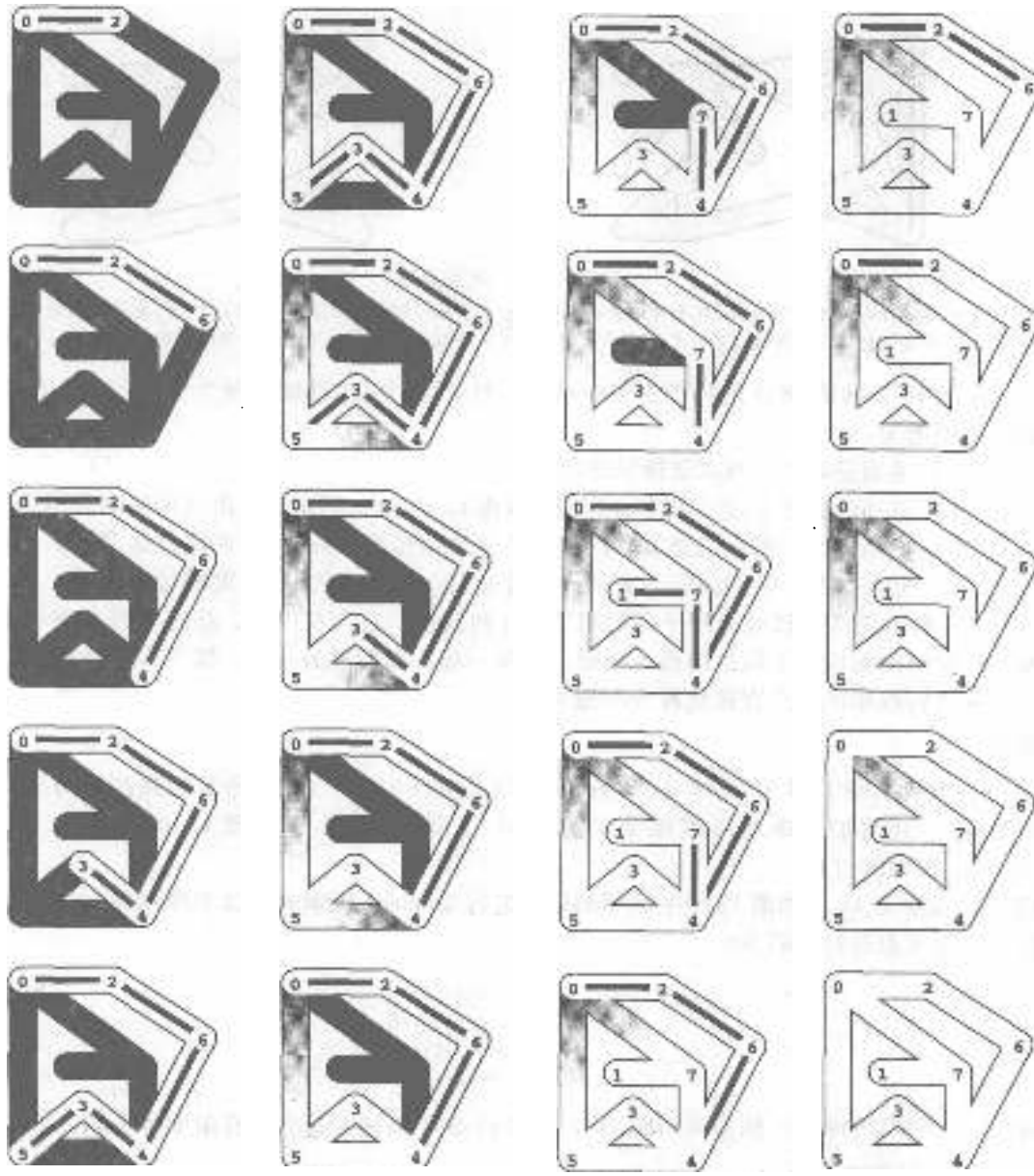


图 18-2　Trémaux 迷宫探索示例

在此图中，尚未访问的地方以灰色标出（暗），已访问的地方以白色标出（亮）。我们假设交叉点处有灯，当我们打开通道两端灯亮的交叉点的门时，通道就亮了。为了探索这个迷宫，我们从 0 开始，取通道走到 2（左上图），再前进到 6，4，3 和 5，打开该通道的门，并在走过时打开各交叉点的灯，然后在身后留下线绳轨迹（左图）。在打开从 5 到 0 的门时，可以看到 0 已经是亮的，因此跳过此通道（右上图）。类似地，还要跳过从 5 到 4 的通道（右边自上第二个图），这样除了从 5 退回到 3，再退回到 4 外，没有地方可去。在回退过程中还要卷起线绳。当打开从 4 到 5 的通道上的门时，通过另一端已经打开的门，可以看到 5 已经开灯，因此，要跳过这条通道（右下图）。我们从未走过连接 4 和 5 的通道，但是因为打开了两端的门而使这条通道是亮的。

图 18-3　Trémaux 迷宫探索示例（续）

接下来，再前进到 7（左上图），打开门，可以看到 0 是亮的（左边自上第二个图），然后再前进到 1（左边自上第三个图）。到此，迷宫的大部分都已经遍历到，我们沿着线绳再回到起始点，即从 1 到 7，再到 4，再到 6，再到 2，最后回到 0。回到 0 时，通过检查通向 5（右边自下第二个图）和 7（右下图）的通道，完成探索，并且各条通道和交叉点都已开灯。同样，连接 0 到 5 和 0 到 7 的通道也是亮的，这是因为我们打开了两端的门，但是并没有走过这两个通道。



图 18-4　迷宫分解

为了使用归纳法证明 Trémaux 探索可以将我们带到迷宫中的任何一个位置（左图），我们去掉将第一个交叉点与由第一个通道所达且无须返回第一个交叉点的任何交叉点连接的所有边，将它分解为两个较小的部分（右图）。

由图 18-2 和图 18-3 的详细示例中可得，对于我们考虑选取的每个通道，会出现 4 种可能的不同情况：

（ⅰ）通道是暗的，所以选择该通道。

（ⅱ）该通道曾进去过（其中有我们的线绳），因此，利用它退出（同时卷起线绳）。

（ⅲ）通道的另一端的门是关着的（但是交叉点是亮着的），因而跳过该通道。

（ⅳ）通道的另一端的门是开着的（而且交叉点是亮着的），因此跳过此通道。

第 1 种和第 2 种情况可用于我们遍历的任何通道，首先在一端，然后再另一端。第 3 种和第 4 种情况则应用于跳过的所有通道，先是一端，然后再另一端。接下来，我们来看如何将这种迷宫探索的观点直接解释为图搜索。

练习

▹ **18.1**　假设从图 18-2 和图 18-3 中的迷宫删除交叉点 6 和 7（以及所有连接到这两个交叉点的通道），并增加一条通道连接交叉点 1 和 2。按照图 18-2 和图 18-3 的样子，显示对 Trémaux 探索的过程。

○ **18.2**　在对图 18-2 和图 18-3 中所示的迷宫进行 Trémaux 探索时，以下序列中，哪一个不能作为各交叉点开灯的顺序？

```
0-7-4-5-3-1-6-2
0-2-6-4-3-7-1-5
0-5-3-4-7-1-6-2
0-7-4-6-2-1-3-5
```

● **18.3**　要使用 Trémaux 探索遍历图 18-2 和图 18-3 中描述的迷宫，有多少种不同的方法？

18.2　深度优先搜索

我们对 Trémaux 探索感兴趣的原因是，通过这项技术可以直接得到遍历图的经典递归函数：要访问一个顶点，我们将它标记为已被访问过的顶点，然后（递归）访问与它相邻且未被访问的所有顶点。我们在第 3 章和第 5 章简洁地讨论了这种方法，并用于解决第 17.7 节中的路径问题。这种方法称为*深度优先搜索*（depth-first search，DFS）。这是我们遇到的最重要的算法之一。DFS 看起来简单，因为它基于一个很熟悉的概念，且很容易实现。事实上，它是一个非常巧妙且强大的算法，在很多困难的图处理问题中都要用到它。

程序 18.1 是 DFS 的一种实现，它访问一个连通图中的所有顶点和所有边，使用邻接矩阵表示。如常，邻接表表示的对应函数只是访问边的机制有所不同（见程序 18.2）。就像我们在第 17.7 节讨论的简单路径搜索函数，其实现基于一个递归函数，在该函数中使用了一

个全局数组和一个增量计数器用以记录顶点被访问的顺序，从而对顶点做出标记。图 18-5 显示了程序 18.1 对于图 18-2 和图 18-3 中描述的示例，访问边和顶点的顺序的轨迹（也见图 18-7）。图 18-6 描述了使用标准绘图程序的同一过程。

这些图说明了递归 DFS 的动态行为，并显示了与迷宫的 Trémaux 探索的对应过程。首先，顶点索引的数组对应着交叉点中的灯：当我们遇到连接至一个已访问顶点的一条边时（看见通道那端的一个灯），我们并不进行递归调用来走过这条边（不沿着这条通道走下去）。第二，程序中的函数调用 - 返回机制对应着迷宫中的线绳：当我们处理了与一个顶点相邻的所有边时（探索了所有离开交叉点的通道），则“返回”（一语双关）。

程序 18.1 深度优先搜索（邻接矩阵）

此代码的作用是使用一个通用的图搜索 ADT 函数，初始化计数器 cnt 为 0，顶点索引数组 pre 中的所有元素为 -1，然后对于每个连通分量调用 search 函数一次（见程序 18.3），假设调用 search(G, EDGE(v, v)) 将同一连通分量中的所有顶点标记为 v（通过将 pre 数组中的相应元素设为非负）。

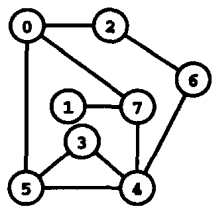
这里，通过扫描邻接矩阵中的每一行，并对通向一个未标记顶点的每条边调用自身，使用访问连接到 e.w 的所有顶点的递归函数 dfsR 来实现 search。

```
#define dfsR search
void dfsR(Graph G, Edge e)
  { int t, w = e.w;
    pre[w] = cnt++;
    for (t = 0; t < G->V; t++)
      if (G->adj[w][t] != 0)
        if (pre[t] == -1)
          dfsR(G, EDGE(w, t));
  }
```

对于迷宫中的每个通道，我们都遇到了两次（通道两端分别遇到一次），同样，对于图中的每条边也遇到了两次（边的两个顶点分别遇到一次）。在 Trémaux 探索中，我们打开了每个通道两端的门；在一个无向图的 DFS 中，我们会分别检查每条边的两个表示。如果遇到一条边 v-w，要么做一个递归调用（如果 w 未被标记），要么跳过这条边（如果 w 已被标记过）。下次在其相反的方向 w-v 遇到这条边时，我们总是会将其忽略，因为可以肯定目标顶点 v 已经访问过（在第一次遇到这条边时就进行了访问）。

程序 18.1 实现的 DFS 和图 18-2 和 18-3 中描述的 Trémaux 探索之间的一个差别有必要花时间来理解，虽然在很多时候不尽合理。当我们从顶点 v 移动到顶点 w 时，还没有检查邻接表中的任何元素，这些元素对应图中从 w 到其他顶点的边。特别是，我们知道从 w 到 v 有一条边，且在到达它时会忽略此边（因为 v 已经标记为已访问过）。这一决定所发生的时间与 Trémaux 探索中有所不同，对于后者，从 v 第一次指向 w 时会打开对应于此边的门。如果我们在进入这些门时将它们关闭，并在走出这些门时将其打开（用线绳来识别通道），就可以在 DFS 和 Trémaux 探索之间做到完全一致的对应。

我们向递归过程传递一条边，而不是向它传递目的顶点，因为这条边会告诉我们如何到达这个顶点。知道了边就相当于知道了哪条通道通向迷宫中的一个特定的交叉点。这个信息在很多 DFS 函数中很有用。当只是记录哪些顶点被访问过时，这个信息影响有限；但更有趣的问题需要我们知道源于何处。



```
                01234567
0-0             0*******
  0-2           0*1*****
    2-0
    2-6         0*1***2*
      6-2
      6-4       0*1*3*2*
        4-3     0*143*2*
          3-4
          3-5   0*14352*
            5-0
            5-3
            5-4
        4-5
        4-6
        4-7     0*143526
          7-0
          7-1   07143526
            1-7
          7-4
  0-5
  0-7
```

图 18-5　DFS 轨迹

对于对应图 18-2 和图 18-3 中示例的邻接矩阵表示法（上图），此图显示了 DFS 检查边和顶点顺序的轨迹，随着搜索的行进，跟踪 pre 数组（右图）中的内容（*表示 -1，表示未见过的顶点）。对于图中的每条边，在轨迹中有两行（每行代表一个方向）。缩排表示递归的层次。

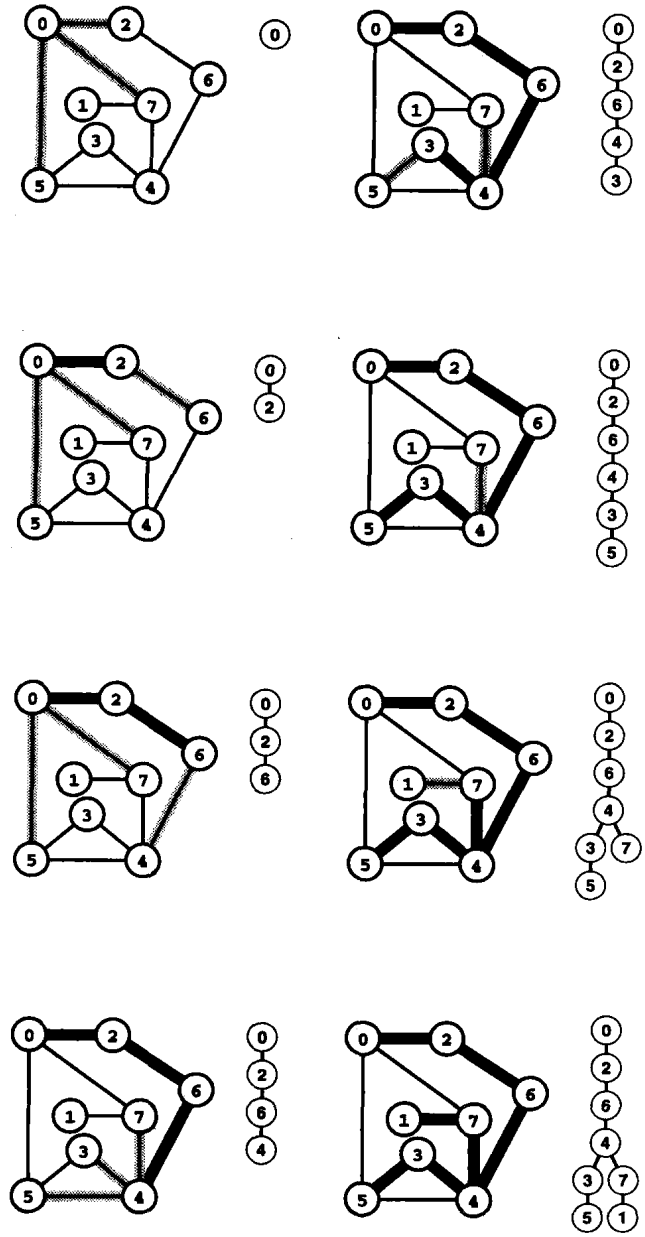


图 18-6　深度优先搜索

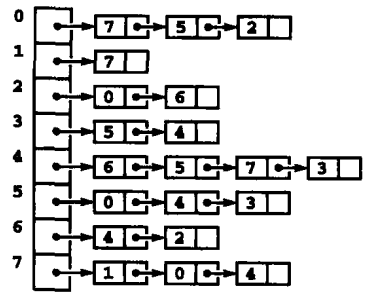
这些图是对图 18-5 中描述的过程的一种图形显示，它们显示了 DFS 运行中的递归调用树。图中的黑色粗边对应于 DFS 树中的边，DFS 树显示在每个图的右侧。阴影边则是下次向树中添加的候选边。在开始阶段（左图），树是沿着直线向下扩展，此时分别对 0、2、6 和 4 进行递归调用（左图）。然后对 3 进行递归调用，然后是对 5 进行递归调用（右边上面两个图）；并从这些递归调用中返回，从而由 4 对 7 做递归调用（右边倒数第二个图），再由 7 到 1 做递归调用（右下图）。

图 18-6 还描述了对应于递归调用所形成的树，与图 18-5 对应。这个递归调用树称为 DFS 树（DFS tree），它是对搜索过程的一种结构性的描述。如我们将在 18.4 节中所看到的，只要适当地改进 DFS 树，就可以完全地描述搜索过程的动态行为，而不只是描述调用结构。

同样的基本模式也对图的邻接表表示有效，如程序 18.2 中所述。如常，不是通过扫描邻接矩阵的一行而是通过扫描该顶点的邻接表来找出与给定顶点邻接的顶点。和以前一样，我们遍历（经由递归调用）连接到那些尚未访问的顶点的所有边。如果图中出现自环和重复边，程序 18.2 中的 DFS 会将其忽略，因而不存在从邻接表中去除它们的麻烦。

对于邻接矩阵表示法，我们按照编号次序检查依附于每个顶点的边；对于邻接表表示法，则按照它们出现在邻接表中的次序进行检查。这种差别就导致了不同递归搜索算法的动态性，如图 18-7 所示。DFS 发现图中边和顶点的顺序完全取决于图表示中邻接表中边所出现的次序。我们也可能按照不同的次序检查邻接矩阵每一行中的元素，这样就得到另一种搜索的动态行为（见练习 18.5）。

尽管存在这些可能性，关键的事实仍然成立，就是 DFS 访问连接到起始顶点的所有边和所有顶点，而不论其以何种次序来检查依附于每个顶点的边。这个事实是性质 18.1 的直接结果，因为此性质的证明并不取决于在任何给定的交叉点上打开门的次序。我们考察的所有基于 DFS 的算法都具有这个相同的基本性质。尽管这些算法操作的动态行为可能变化很大，这取决于图的表示和搜索的实现细节。递归结构给出了一种使我们对图自身进行相关推理的方法，无论其如何表示，也不管我们选择何种次序检查依附于每个顶点的边。




```
                    01234567
0-0                 0*******
  0-7               0******1
    7-1             02*****1
      1-7
    7-0
    7-4             02**3**1
      4-6           02**3*41
        6-4
        6-2         025*3*41
          2-0
            2-6
        4-5         025*3641
          5-0
          5-4
          5-3       02573641
            3-5
            3-4
      4-7
      4-3
  0-5
  0-2
```

图 18-7 DFS 轨迹（邻接表）

对于图 18-5 所示的同一个图的邻接表表示，此轨迹显示了 DFS 检查边和顶点的次序。

程序 18.2 深度优先搜索（邻接表）

此 dfsR 的实现是用邻接表表示图的 DFS。该算法与邻接矩阵表示的相应算法（程序 18.1）一样：要访问一个顶点，先对它进行标记，然后扫描依附它的边，在遇到通向一个未标记的顶点的边时，进行递归调用。

```
void dfsR(Graph G, Edge e)
  { link t; int w = e.w;
    pre[w] = cnt++;
    for (t = G->adj[w]; t != NULL; t = t->next)
      if (pre[t->v] == -1)
        dfsR(G, EDGE(w, t->v));
  }
```

练习

▷ **18.4**　按照图 18-5 的样式，显示下图的标准邻接矩阵 DFS 所做的递归函数调用轨迹。

0-2　0-5　1-2　3-4　4-5　3-5

画出对应的 DFS 递归调用树。

▷ **18.5**　按照图 18-6 的样式，如果修改搜索函数，使得按照逆序扫描顶点（从 v-1 降至 0），显示搜索的过程。

○ **18.6**　使用独立于表示的 ADT 函数处理练习 17.60 中的边表，实现 DFS。

18.3　图搜索 ADT 函数

DFS 以及本章稍后考虑的其他图搜索方法都会涉及沿着图中从顶点到顶点的边的行进，目标是系统地访问图中的每个顶点和每条边。但是，沿着图中从顶点到顶点的边可能会使我们只能到达该起始顶点所在的连通分量的所有顶点。当然，一般来说，图可能不是连通的，因而，对于每个连通分量需要调用一次搜索函数。我们通常会使用一个通用的搜索函数执行以下步骤，直至图中所有顶点都已标记为访问过；

- 查找一个未标记的顶点（一个起始顶点）。
- 访问包含该起始顶点的连通分量中的所有顶点（且标记为已访问）。

在此描述中并没有指定标记顶点的方法，但是我们最常使用第 18.2 节 DFS 实现中所使用的方法：将一个全局顶点索引数组中的所有元素初始化为一个负整数，并通过设置顶点所对应的元素为一个正值来标记该顶点。使用这一过程对应着使用一个单独的位（符号位）来做标记；大多数实现还会关注将其他与已标记顶点相关的信息保存在数组中（比如，在 18.2 节的 DFS 实现中，保存标记顶点的次序）。还没有指定在下一个连通分量中查找顶点的方法，但我们最常按照索引递增的顺序扫描数组。

程序 18.3　图搜索

我们一般使用此代码来处理那些可能不连通的图。函数 GRAPHsearch 假设在调用含有指向 v 的自环作为其第二个参量时，search 将 pre 中的元素设为非负值，它对应于连接到 v 的每个顶点。在这一假设之下，对于图中的每个连通分量，这种实现调用 search 一次，我们也可将它用于任何图的表示和任何 search 函数。

结合程序 18.1 或程序 18.2，此代码计算 pre 中顶点被访问的次序。其他基于 DFS 的实现要进行其他计算，但使用相同的模式，将顶点索引数组中的非负元素解释为顶点标记。

```
static int cnt, pre[maxV];
void GRAPHsearch(Graph G)
  { int v;
    cnt = 0;
    for (v = 0; v < G->V; v++) pre[v] = -1;
    for (v = 0; v < G->V; v++)
      if (pre[v] == -1)
        search(G, EDGE(v, v));
  }
```

程序 18.3 中的 GRAPHsearch ADT 函数是描述这些选择的一种实现。结合程序 18.1 和程序 18.2，图 18-8 展示了此函数中 pre 数组的作用（或者任何将同一连通分量中的所有顶点标记为其参数的图搜索函数）。我们考虑的图搜索函数还检查依附于每个所访问过顶点的所

有边，就像在 Trémaux 遍历中那样，知道访问的所有顶点也就说明访问了所有的边。

在一个连通图中，ADT 函数只不过是封装器，对于 0-0 调用 search 一次，然后发现所有其他顶点被标记。在一个有多个连通分量的图中，ADT 函数直接检查所有连通分量。DFS 是搜索一个连通分量所考虑的几种方法中的首选方法。不管采用何种方法（也不管采用哪种图的表示），程序 18.3 是一种访问图中顶点的有效方法。

性质 18.2 图搜索函数检查图中的每条边和标记每个顶点，当且仅当它使用的搜索函数对含有起始顶点的连通分量中每个顶点进行标记且检查每条边。

证明 对连通分量个数运用归纳法证明。 ■

图搜索函数提供了处理图中每个顶点和每条边的一种系统方法。一般来说，通过对每条边做定量的处理，可使实现的运行时间为线性或近似线性时间。我们证明这个事实对 DFS 成立。要注意同样的证明技术对于其他几个搜索策略同样适用。

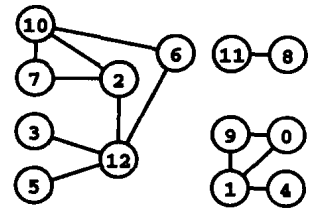
性质 18.3 使用邻接矩阵表示的图的 DFS 需要与 V^2 成正比的时间。

证明 类似于性质 18.1 的证明，dfsR 不仅对连通到起始顶点的所有顶点进行标记，而且对于每个顶点只会调用自身一次（来标记那个顶点）。类似性质 18.2 的证程，调用 GRAPHsearch 将会导致对于图中的每个顶点只调用一次 dfsR。在 dfsR 中，我们检查邻接矩阵中顶点所在行的每个元素。换句话说，检查邻接矩阵中的每个元素只有一次。 ■

性质 18.4 使用邻接表表示的图的 DFS 需要与 $V+E$ 成正比的时间。

证明 根据刚才的论述，由此可得我们只调用递归函数 V 次（因此有 V 这一项），并且要检查每个邻接表的每个元素（因此有 E 这一项）。 ■

性质 18.3 和 18.4 中蕴含的主要结论是它们确立了 DFS 的运行时间与用于表示图的数据结构的规模呈线性。在大多数情况下，我们还认为 DFS 的运行时间与图的规模呈线性。如果有一个稠密图（图中边数与 V^2 成正比），那么无论哪种表示都可得到这个结果；如果有一个稀疏图，则假设使用邻接表表示。实际上，我们一般认为 DFS 的运行时间与 E 呈线性。如果对于稀疏图或极其稀疏且大多数顶点为孤立顶点的图（$E << V$）使用邻接矩阵，那么上述结论不再成立。但我们通常可以避免前一种情况，而且在后一种情况下去掉孤立顶点（见练习 17.33）。



	0	1	2	3	4	5	6	7	8	9	10	11	12
	*	*	*	*	*	*	*	*	*	*	*	*	*
0-0	0	0	*	*	0	*	*	*	*	0	*	*	*
2-2	0	0	0	0	0	0	0	0	*	0	0	*	0
8-8	0	0	0	0	0	0	0	0	0	0	0	0	0

图 18-8 图搜索

下面的表显示了对上图的一个典型搜索过程中的顶点标记（pre 数组中的内容）。初始时，程序 18.3 中的函数 GRAPHsearch 通过将所有顶点的标记设为 -1（用 * 表示）指定所有顶点未标记。然后，对于虚拟边 0-0 调用 search，将同一连通分量中所有顶点的标记设为非负（用 0 表示），使该连通分量中的所有顶点标记为 0（第二行）。在此例中，按照顺序，分别将顶点 0、1、4 和 9 标记为 0、1、2、3。接下来，它从左到右扫描来找出未标记的顶点 2，对虚拟边 2-2 调用 search（第三行），并将同一连通分量中的 7 个顶点标记为 2。继续从左到右扫描，对于 8-8 调用 search 对 8 和 11 进行标记（最后一行）。最后，GRAPHsearch 发现顶点 9 ~ 12 都已被标记，完成这个搜索过程。

我们将会看到，这些结论可以应用到有相同 DFS 基本特性的算法中。如果算法对每个顶点标记，检查后者的所有依附顶点（并完成其他工作，所耗时间的界限为常量），那么这些性质就适用。更一般地，如果在每个顶点上的时间由某个函数 f(V,E) 所限定，那么可以保证搜索的时间与 $E+V$f(V, E) 成正比。在 18.8 节中，我们会看到 DFS 只是有这些特征的算法家族中的一个；在第 19 ~ 22 章，会看到来自此家族的算法可以作为本书中所考虑的大量代码的基础。

我们所考察的大量图处理代码是针对一些特定任务的 ADT 实现代码，在此我们扩展了一个基本搜索，用以计算在其他顶点索引的数组中的结构信息。从本质上讲，每当我们构建一个围绕搜索的算法实现时，就重新实现搜索。采用这种方法的原因是，很多算法作为一种参数化的图搜索函数已得到透彻理解。一般地，我们通过搜索图来进行探测。我们通常会扩展搜索函数，补充在每个顶点被标记的时候才执行的代码，而不是研究一种更一般性的搜索（例如，每当访问一个顶点就调用一个特定函数的搜索），这样才能使代码简洁且自包含。为客户提供一种更一般的 ADT 机制来处理带有客户提供函数的所有顶点是很值得考虑的事情（见练习 18.12 和 18.13）。

在 18.5 节和 18.6 节中，我们按照这种方法考察了很多基于 DFS 的图处理函数。在 18.7 节和 18.8 节中，我们考虑 search 的其他实现以及基于它们的其他图处理函数。虽然在代码中并没有构建这一层的抽象，我们还是留意每个算法中潜在的基本图搜索策略。例如，我们使用术语 DFS 函数（DFS function）来指示基于递归 DFS 模式的所有实现。简单路径搜索函数程序 17.11 就是 DFS 函数的一个例子。

很多图处理函数是基于使用顶点索引的数组。在实现中我们一般地将这些数组分为三种方式使用：

- 作为全局变量
- 在图的表示中
- 作为函数参数，由客户提供

在收集关于搜索学习图结构事实的信息时，我们使用第一种方式，可以有助于解决各种有趣的问题。这种数组的一个例子是程序 18.1 ~ 18.3 中使用的 pre 数组。在实现计算关于图信息的预处理函数时，使用第二种方式，这些预处理函数能够高效地实现所关联的 ADT 函数。例如，我们可能维护这样一个数组来支持返回任一顶点的度的 ADT 函数。在实现一个计算顶点索引的数组的 ADT 函数时，我们可能使用第二种或第三种方式，这取决于具体上下文环境。

在图搜索函数中，一个约定是将顶点索引的数组中的元素初始化为 -1，并在搜索函数中将每个已访问过的顶点所对应的元素设置为非负值。任何这样的数组都可以起到程序 18.1 ~ 18.3 中的 pre 数组的作用（将顶点标记为已访问）。当图搜索函数是基于使用或者计算一个顶点索引的数组时，我们使用那个数组来标记顶点，而不是维护 pre 数组。

图搜索的特定结果不仅取决于搜索函数的特性，而且取决于图的表示，甚至取决于 GRAPHsearch 中间检查顶点的顺序。对于本书中的例子和练习，我们使用术语*标准邻接表 DFS*（standard adjacency-lists DFS）来表示将一系列的边插入到由邻接表表示所实现的图 ADT 中（程序 17.6），然后用程序 18.3 和 18.2 执行 DFS 的过程。对于邻接矩阵表示，边插入的顺序不会影响搜索的动态性，但我们使用相对应的术语*标准邻接矩阵 DFS*（standard adjacency-matrix DFS）来表示将一系列的边插入到由邻接矩阵表示所实现的图 ADT 中（程序 17.3），然后用程序 18.3 执行 DFS 的过程。

练习

18.7 按照图 18-5 的样式，显示下图的标准邻接矩阵 DFS 所做的递归函数调用轨迹。

```
3-7  1-4  7-8  0-5  5-2  3-8  2-9  0-6  409  2-6  6-4
```

18.8 按照图 18-7 的样式，显示下图的标准邻接表 DFS 所做的递归函数调用轨迹。

```
3-7  1-4  7-8  0-5  5-2  3-8  2-9  0-6  409  2-6  6-4
```

18.9 修改程序 17.3 中的标准邻接矩阵图 ADT 实现，使用一个连接到其他所有顶点的虚拟顶点，然后提供一个可以利用此改进的简化的 DFS 实现。

18.10 对程序 17.6 中的邻接表 ADT 实现完成练习 18.9。

• 18.11 在图 18-8 中描述的图中有 13！种不同的顶点排列顺序。其中有多少种排列可能指定程序 18.3 访问顶点时的顺序？

18.12 定义一个 ADT 函数，对于图中的每个顶点调用客户提供的函数。提供邻接矩阵和邻接表表示的实现。

18.13 定义一个 ADT 函数，对于图中的每条边调用客户提供的函数。提供邻接矩阵和邻接表表示的实现。（在 ADT 设计中，这样的一个函数代替 GRAPHedges 是合理的。）

18.4 DFS 森林的性质

18.2 节中已经提到，描述 DFS 函数调用的递归结构是我们理解 DFS 如何操作的关键。在这一节里，我们通过检查 DFS 树的性质来分析算法的性质。

DFS 实现中的 pre 数组是对 DFS 树的内部结点的*前序编号*（preorder numbering）。计算一个显式的*父链接表示*（parent-link representation）的 DFS 树也很容易：可以将顶点索引数组 st 中的所有元素初始化为 -1，并在程序 18.1 和 18.2 的递归 DFS 函数中包括语句 `st[e.w] = e.v`。

如果向 DFS 树中添加外部结点，来记录已经访问过的顶点跳过递归调用的时刻，我们就可以得到图 18-9 中描述的 DFS 动态性的简洁表示。这种表示值得仔细研究。此树是图的一种表示，树中顶点对应图中的每个顶点，树中的每条边对应图中的每条边。我们可以选择显示出所处理边的两种表示（每个方向一种表示），如左图所示，或每条边只有一种表示，如中图和右图所示。前者的表示对于理解算法处理每一条边很有用；后者对于理解 DFS 树只是另一种图的表示有用。前序遍历这棵树的内部结点给出 DFS 访问结点的顺序。而且，当我们按照前序遍历此树时，访问树中边的次序与 DFS 检查图中边的次序一样。

实际上，图 18-9 中的 DFS 树与图 18-5 中的轨迹包含相同的信息与图 18-2 和图 18-3 进行 Trémaux 遍历所作的一步一步的描述包含的信息也相同。指向内部结点的边表示通向未被访问顶点（交叉点）的边（通道），指向外部结点的边表示如下情况：DFS 检查的边通向以前访问过的顶点（交叉点），阴影结点表示的边，对于其指向的顶点正在执行递归 DFS（当我们打开通向通道的一扇门时，此通道另一端的门已经打开）。有了这些解释，可得树的前序遍历的过程与复杂的遍历迷宫的情景完全相同。

为了研究更为复杂的图的性质，我们将图中的边按照其在搜索中所起的作用进行分类。可得两类不同的边：

- 表示一个递归调用的边（*树边*，tree edge）
- 将一个顶点与其 DFS 树中的一个祖先相连的边，且该祖先不是其父结点（*回边*，back edge）。

在第 19 章研究有向图的 DFS 树时，我们会检查其他类型的边，不只是考虑方向，还会考虑存在穿越树的边，它们将既不是此树中的祖先也不是树中的子孙结点连接起来。

因为图中的每条边都有两种表示，每种表示对应 DFS 树中的一个链接，因此可以将树的链接分为 4 类。对于 DFS 树中从 v 到 w 的表示树边的一个链接，我们称为

- *树链接*（tree link），如果 w 未被标记
- *父链接*（parent link），如果 st[w] 为 v

且对于从 v 到 w 的表示回边的一个链接，我们称为

- 回链接（back link），如果 pre［w］ < pre［v］
- 下链接（down link），如果 pre［w］ > pre［v］

图中的每条树边对应 DFS 树中的一个树链接和一个父链接，图中的每条回边对应着 DFS 树中的一个回链接和一个下链接。

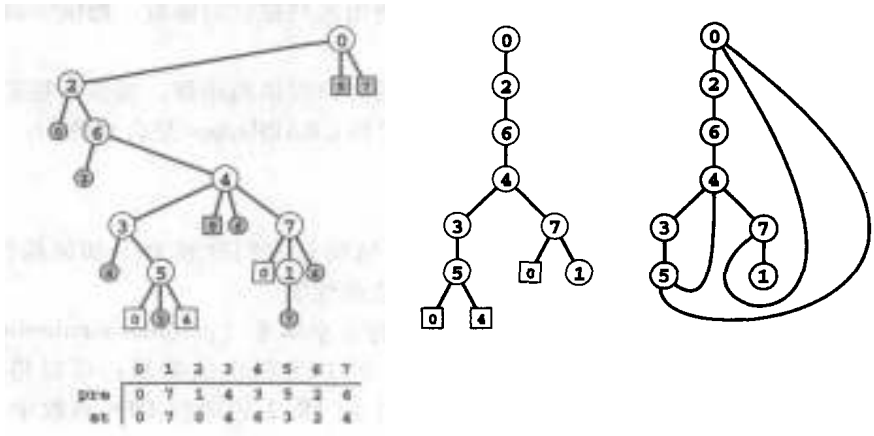


图 18-9　DFS 树表示

如果增大 DFS 递归调用树来表示被检查但未走过的边，我们就得到 DFS 过程的一个完整描述（左图）。树中每个结点有一个子结点，表示与其邻接的每个结点，DFS 按照此顺序检查它们。而且前序遍历给出如图 18-5 的相同信息：首先，我们沿着 0-0，接着 0-2，然后跳过 2-0，再沿着 2-6，然后跳过 6-2，下面沿着 6-4，其后是 4-3，如此继续。数组 pre 指定了我们在前序遍历过程中访问树中顶点的顺序，这个顺序与用 DFS 访问图中的顶点的顺序相同。数组 st 是 DFS 递归调用树的一个父链接表示（见图 18-6）。

对于图中的每条边，在树中都有两个链接。每个链接表示遇到该边的两次中的一次。第一次是到一个无阴影的结点，或者对应执行一次递归调用（如果它是一个内部结点），或者由于它到达一个祖先，而对这个祖先的递归调用正在进行，因而跳过一次递归调用（如果要到一个外部结点）。第二次是到一个有阴影的外部结点，由于它要么返回到父结点（圆形），要么走到一个父结点的子孙且对子孙的递归调用正在进行（方形），因而总是对应跳过一次递归调用。如果我们消除阴影的结点（中图），那么用边替换掉外部结点，就得到图的另一种绘制（右图）。

在图 18-9 中所示的图形化 DFS 表示中，树链接指向未加阴影的圆形顶点，父链接指向加阴影的圆形顶点，回链接指向未加阴影的方形顶点，下链接指向加阴影的圆形顶点。图中每条边或者用一个树链接和一个父链接表示，或者用一个下链接和一个回链接表示。这些分类很有技巧，值得研究。例如，注意到即使父链接和回链接都指向树中的祖先，它们也完全不同：父链接只是树链接的另一种表示，而回链接为我们提供了有关图结构的新信息。

以上给出的定义为区分 DFS 实现中的树链接、父链接、回链接和下链接提供了足够信息。注意到父链接和回链接都有条件 pre［w］ < pre［v］，因而，我们还必须知道 st［w］不是 v，从而知道 v-w 是一条回链接。对于图中的每条边，在示例的 DFS 中遇到该边时，图 18-10 描述了其对 DFS 树链接分类的打印结果。这是基本搜索过程的另一种完整表示，它是介于图 18-5 和图 18-9 之间的一个中间步骤。

4 种类型的树链接对应在一个 DFS 中处理边的 4 种完全不同的方法，正如 18.1 节最后所描述的（迷宫探索的方式）。树链接对应的 DFS 遇到了树边两种表示的第一种表示，这将导致一个递归调用（指向尚未见过的顶点）；父链接对应的 DFS 遇到了树边两种表示的另一种表示（它是在第一次递归调用中走过邻接表示遇到的），并忽略此链接。回链接对应的 DFS 遇到了回边两种表示的第一种表示，指向一个顶点，对于该顶点递归搜索函数尚未完成；下链接对应的 DFS 遇到了一个顶点，对于该顶点，递归搜索函数遇到该边时已经完成。

在图 18-9 中，树链接和回链接将未加阴影的结点连接起来，表示第一次遇到相应边，并构造图的一种表示；父链接和下链接走到加阴影的结点，表示第二次遇到相应边。

我们已经详细地讨论了递归 DFS 动态特性的树表示，这不仅是由于这种表示为图和算法的操作都提供了一个完整而简洁的描述，而且还由于它为理解无数重要图处理算法的提供了基础。在本章其余部分，以及接下来的几章里，我们将考虑大量图处理问题的例子，并由 DFS 树得到图的结构。

图的搜索是树遍历的一种推广。调用一棵树时，DFS 就完全等价于递归树遍历；对于图，使用 DFS 对应于遍历生成此图的一棵树。随着搜索进行将发现这棵树。如我们将看到的，所遍历的特定树取决于图的表示。DFS 对应前序树遍历。在第 18.6 节，我们将考察类似于树的层次遍历的图搜索方法，并探索它与 DFS 的关系；在第 18.7 节，我们将考察包含多种遍历方法的一般模式。

在对图进行遍历时，我们按照开始处理顶点的顺序对它们指定前序编号（preorder number）（仅在进入递归搜索函数之后）。我们还可以按照完成（finish）处理顶点的顺序给它们指定后序编号（postorder number）。在处理一个图时，我们所做的远不只是遍历顶点，如将看到的，前序编号和后序编号提供了图的全局性质的知识，这有助于我们完成手边的任务。前序编号对于本章所讨论的算法已经够用，但我们在后续的章中会使用后序编号。

```
0-0 tree
  0-2 tree
    2-0 parent
    2-6 tree
      6-2 parent
      6-4 tree
        4-3 tree
          3-4 parent
          3-5 tree
            5-0 back
            5-3 parent
            5-4 back
        4-5 down
        4-6 parent
        4-7 tree
          7-0 back
          7-1 tree
            1-7 parent
          7-4 parent
  0-5 down
  0-7 down
```

图 18-10 DFS 轨迹（树链接分类）

图 18-5 的这个版本显示了对 DFS 树链接的分类，对应于遇到的图中每条边表示。树边（对应递归调用）第一次遇到时被表示为树链接，第二次遇到时被表示为父链接。回边第一次遇到时为回链接，第二次遇到时则为下链接。

我们用 DFS 森林（forest）来描述一般无向图的 DFS 的动态性，其中每个连通分量都有一个 DFS 树。图 18-11 显示了一个 DFS 森林的例子。

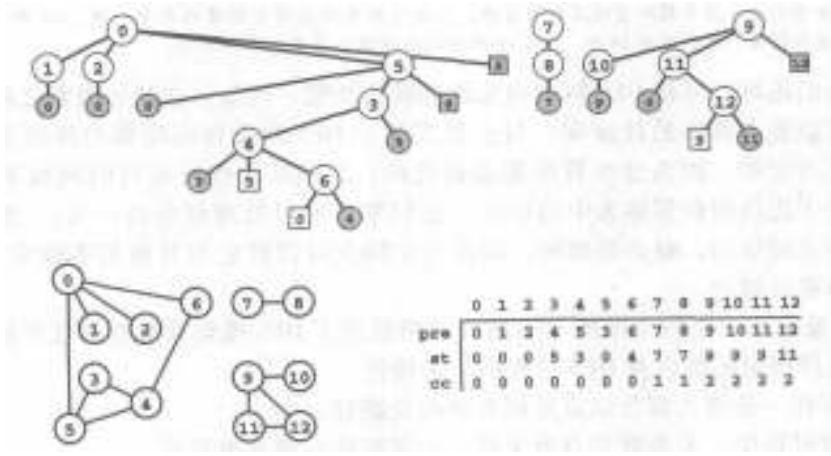


图 18-11 DFS 森林

上图中的 DFS 森林表示了右下图邻接矩阵表示的 DFS。此图有三个连通分量，因而该森林有三棵树。pre 数组是树中结点的前序编号（DFS 检查它们的顺序），st 数组是该森林的父链接表示。cc 数组将每个顶点与一个连通分量索引关联（见程序 18.4）。如在图 18-9 中那样，指向圆形结点的边是树边；指向方形结点的边是回边；有阴影的结点表明所依附的边在搜索中较早被遇到过（但在另一个方向）。

使用邻接表表示，我们访问连接到每个顶点的边，访问顺序不同于邻接矩阵表示，因而会得到一个不同的 DFS 森林，如图 18-12 所示。DFS 树和森林是图的表示，不仅描述了 DFS 的动态性，而且描述了图的内部表示。例如，从左到右读取图 18-12 中的任何结点的子结点，我们将看到它们出现在对应那个结点的顶点的邻接表中的顺序。对于同一个图，我们可有多个不同的 DFS 森林，邻接表中结点的每种顺序都导致一个不同的森林。

特定森林结构的细节有益于我们理解特定图的 DFS 操作是如何进行的，但我们考虑的大多数重要的 DFS 性质取决于图的性质，这些性质独立于森林的结构。例如，图 18-11 和图 18-12 中的森林都有三棵树（同一个图的其他所有 DFS 森林也是如此），因为它们只是同一个图的不同表示，而此图有三个连通分量。实际上，DFS 可访问一个图中的所有结点和所有边（见性质 18.2 ~ 18.4），这个基本证据的一个直接结果就是图中的连通分量个数等于 DFS 森林中的树的个数。这个例子展示了贯穿本书所使用的图搜索的基础：大量图 ADT 函数实现都以通过处理一个特定图表示（对应于搜索的森林）来学习图的性质为基础。

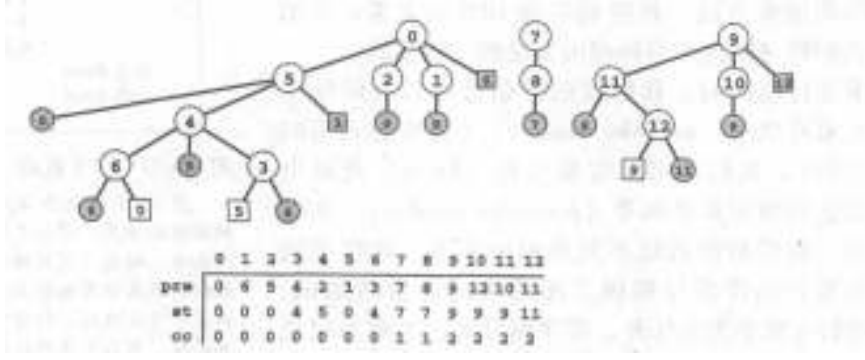


图 18-12　另一个 DFS 森林

此森林描述了对于图 18-9 所示的同一个图的深度优先搜索，但使用程序 18.2，因而搜索顺序不同，这是由于此顺序是由结点出现在邻接表中的顺序来确定的。实际上，该森林自身就告诉了我们这个顺序：这个顺序就是树中每个结点的子结点排列的顺序。例如，0 的邻接表中的结点被查找的顺序为 5 2 1 6；4 的邻接表中结点的顺序为 6，5，3，等等。如前，在搜索中检查图中的所有顶点和所有边，方法与前序遍历树中描述的完全一样。pre 和 st 数组取决于图的表示和搜索的动态行为，有别于图 18-9。但是 cc 数组取决于图的性质，因而相同。

也许，我们还可以分析 DFS 树结构来改进算法性能。例如，在开始搜索之前，通过重排邻接表是否可以提高算法的速度呢？对于很多基于 DFS 的重要而经典的算法来说，对这个问题的答案是否定的，因为这些算法都是最优的，其最坏情况下运行时间既不取决于图结构，也不取决于边出现在邻接表中的顺序（它们基本上只处理每条边一次）。然而，DFS 森林有一个有特点的结构，有必要理解，因为这个特点可以将它与其他基本搜索模式区分开，我们将在本章稍后讨论。

图 18-13 显示了一个较大图的 DFS 树，该树描述了 DFS 搜索动态性的基本特性。树高且细，它展示了所搜索的图以及 DFS 过程的许多特性。

- 至少存在一条将大部分结点连接起来的长路径。
- 在搜索过程中，大多数顶点至少有一个邻接顶点尚未被看到。
- 对于任何顶点，我们很少进行一次以上的递归调用。
- 递归深度与图中的顶点数成正比。

尽管不能保证所有图都有这些特性，但这一行为对于 DFS 而言是很典型的。验证图模型的这些事实以及实际中出现的各种类型的图需要详尽的研究。然而，对于实际中得到的基于

DFS的算法，这个例子还是给出了一种直观认识。图18-13和其他类似图搜索算法的图（见图18-24和图18-29）能够帮助我们理解其行为上的差异。

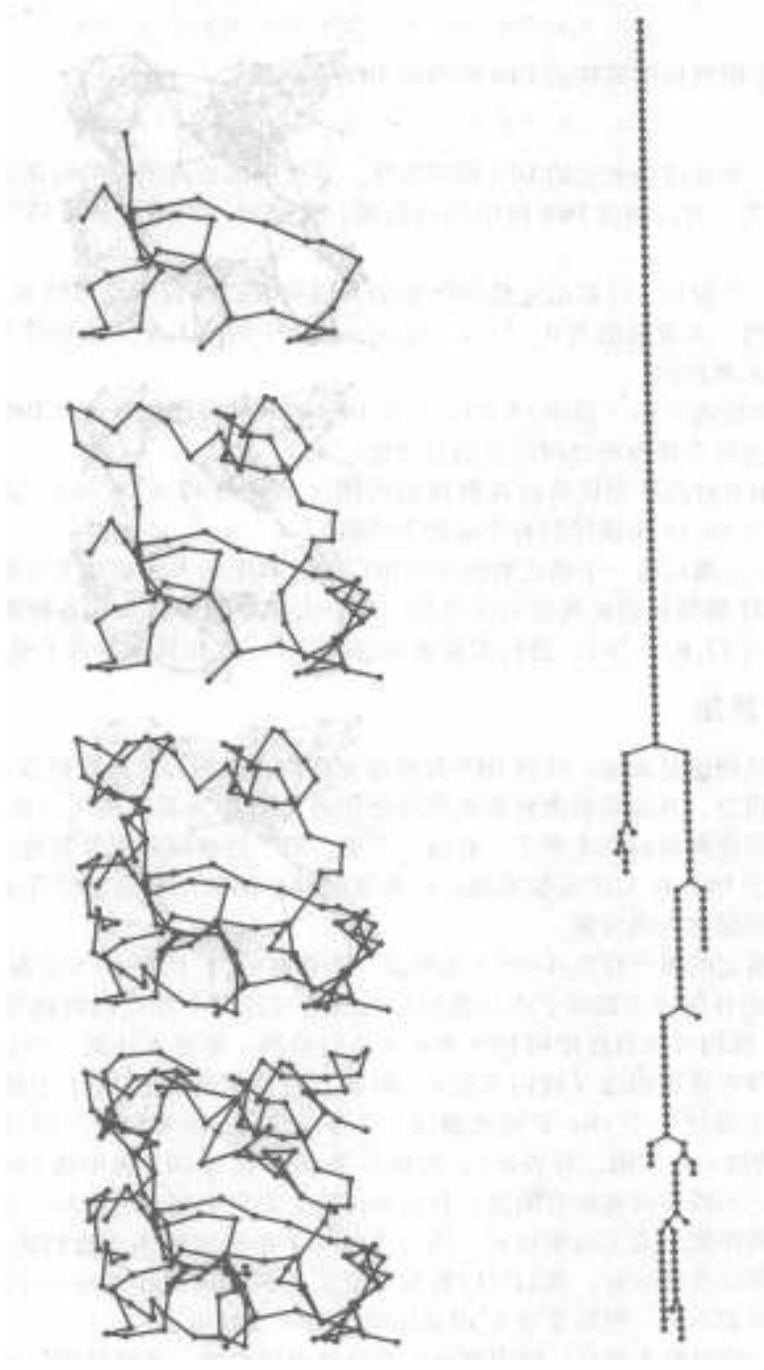


图18-13 深度优先搜索

此图描述了在一个随机欧几里得近邻图中DFS的过程（左图）。这些图（从上到下）显示了搜索到图的1/4、1/2、3/4和全部顶点时DFS树的顶点和边。DFS（仅有树边）显示在右边。在此例中很明显，对于这种类型的图，DFS的搜索树往往是很高且细（对于在实际中常常遇到的许多其他类型的图也同样如此）。我们通常会在附近找到一个以前未见过的顶点。

练习

18.14 画出下图的标准邻接矩阵 DFS 所得的 DFS 森林。

```
3-7  1-4  7-8  0-5  3-8  2-9  0-6  4-9  2  6  6-4
```

18.15 画出下图的标准邻接表 DFS 所得的 DFS 森林。

```
3-7  1-4  7-8  0-5  3-8  2-9  0-6  4-9  2  6  6-4
```

18.16 编写一个邻接表表示的 DFS 跟踪程序，产生的输出将图中的每条边的两种表示中的每一种进行分类，对应到该 DFS 树中的树链接、父链接、回链接或下链接，显示风格同图 18-10。

○ **18.17** 编写一个程序，计算出完整 DFS 树的父链接表示（包括外部结点），使用含有 E 个整数的一个数组，元素取值为 $0 \sim V-1$。提示：数组中的前 V 个元素应该与正文中描述的 st 数组中的对应元素相同。

○ **18.18** 对于邻接表表示（程序 18.2），给出 DFS 的一种实现，打印出 DFS 树的高度和回边个数，以及检查每个顶点所处理的边的百分比。

● **18.19** 对于由各种图模型所得的各种规模的图（见练习 17.63 ~ 76），进行实验研究，通过实验确定练习 18.18 中描述的各个量的平均值。

● **18.20** 编写一个通过将一个给定数组中的边以随机顺序插入到初始为空的图来构建图的函数。基于图 ADT 的邻接表实现使用该函数，对于大样本的图以及从各种图模型所得的邻接表表示（见练习 17.63 ~ 76），进行实验来确定练习 18.18 中描述的各个量的分布特性。

18.5 DFS 算法

无论是图结构还是表示，任何 DFS 森林都允许将边识别为树边或回边，为我们提供图结构的一些可靠信息，并在求解大量图处理问题中将 DFS 作为基础使用。在第 17.7 节中我们已经看到有关查找路径的基本例子。在这一节里，对于这些问题以及其他一些典型问题，我们考虑一些基于 DFS 的 ADT 函数实现；在本章的其余部分以及后面的几章中，还将看到很多更为困难的问题的解决方案。

环检测 给定的图中存在环吗？（此图是一个森林吗？）使用 DFS 很容易解决这个问题。因为 DFS 树中的任何回边都属于由这条回边以及连接这两个结点的树路径组成的环（见图 18-9）。因此，我们可以直接使用 DFS 来对环进行检测：图是无环的，当且仅当在一次 DFS 中，如果我们没有遇到回边（或向下边）。例如，为了测试程序 18.1 中的这个条件，我们只需在 if 语句中增加一个 else 子句来测试 t 是否等于 v。如果相等，则只是遇到了父链接 w-v（通向 w 的边 v-w 的第二种表示）。如果不等，w-t 就与 DFS 树中从 t 向下到 w 的边形成一个环。而且，不需要检查所有的边：我们知道要么必定会找到一个环，要么在检查 V 条边之前且没有找到环就完成了搜索过程。因为任何有 V 条边或更多条边的图必定存在一个环。因此，当使用邻接表表示时，我们可以在与 V 成正比的时间内检查出一个图是否是无环的，但使用邻接矩阵表示时，则需要与 V^2 成正比的时间（来找出边）。

简单路径 给定两个顶点，图中存在包含这两个顶点的一条路径吗？由第 17.7 节可知，设计一个在线性时间求解此问题的 DFS 函数是简单的。

简单连通性 如在第 18.3 中所讨论过的，无论何时使用 DFS，我们都可以在线性时间内确定一个图是否是连通的。实际上，我们基本图搜索问题策略是基于对于每个连通分量都调用一个搜索函数。在一个 DFS 中，图是连通的，当且仅当图搜索函数调用递归 DFS 函数

只一次（见程序18.3）。图中的连通分量个数就是GRAPHsearch调用递归函数的次数，因此我们只需要记录这些调用的个数，就能找到连通分量的个数。

更一般地，程序18.4描述了邻接表表示的基于DFS的ADT实现，在经过线性时间预处理之后，可支持常量时间的连通性查询。DFS森林中的每棵树都标示出一个连通分量，因而，我们可以在图的表示中包含顶点索引的数组，通过DFS填充并由连通性访问，很快就可以断定两个顶点是否在同一个分量中。在递归DFS函数中，我们将分量计数器的当前值设置为每个已访问过的顶点所对应的元素。然后，我们知道两个顶点在同一分量中，当且仅当它们在数组中的相应元素是相同的。同样，注意到这个数组反映了图的结构性质，而不是图表示或搜索动态性的产物。

程序18.4 图的连通性（邻接表）

DFS函数GRAPHcc在线性时间内计算一个图中的连通分量的个数，并将关联到每个顶点的连通分量索引存储到图表示的顶点索引的数组G->cc中。（因为它并不需要结构信息，在程序18.2中，递归函数使用顶点而不是边作为它的第二个参数）。调用GRAPHcc之后，客户程序可以在常量的时间内测试任何一对顶点是否是连通的（GRAPHconnect）。

```
void dfsRcc(Graph G, int v, int id)
  { link t;
    G->cc[v] = id;
    for (t = G->adj[v]; t != NULL; t = t->next)
      if (G->cc[t->v] == -1) dfsRcc(G, t->v, id);
  }
int GRAPHcc(Graph G)
  { int v, id = 0;
    G->cc = malloc(G->V * sizeof(int));
    for (v = 0; v < G->V; v++)
      G->cc[v] = -1;
    for (v = 0; v < G->V; v++)
      if (G->cc[v] == -1) dfsRcc(G, v, id++);
    return id;
  }
int GRAPHconnect(Graph G, int s, int t)
  { return G->cc[s] == G->cc[t]; }
```

程序18.4代表了我们在求解各种图处理问题中将要使用的基本方法。我们花费一些预处理时间并扩展图ADT来计算图的结构性质，以帮助我们为重要的ADT函数提供高效实现。在这种情况下，我们使用（线性时间）DFS进行预处理，并保存一个数组cc，这可使我们在常量时间内回答连通性问题。对于其他图处理问题，我们可能使用更多空间、预处理时间或查询时间。通常，我们的目标是最小化这个开销，虽然这样做常常极富挑战性。例如，第19章的大部分主要是解决有向图的连通性问题，其中要达到像程序18.4那样的性能特性是一个困难的目标。

给定一个边表，与我们在第1章所考虑的用于确定一个图是否是连通的合并-查找方法比较，程序18.4中基于DFS的图的连通性的解决方案如何呢？理论上，DFS要比合并-查找快，因为它提供了常量时间的保证，而合并-查找不能；实际上，这种差别是可以忽略的，合并-查找更快是因为它不必构建图的一个完整表示。更重要的，合并-查找是一个在线算法（我们可以随时在近似常量的时间内检查两个顶点是否是连通的），而DFS的解决方案由于对图进行了预处理，可在常量时间回答连通性查询。因此，比如说，确定连通性是唯

一的任务时，或者在有大量查询混合着边插入时，更倾向使用合并－查找算法，但是在图的ADT中使用时，我们会发现DFS更合适，因为它有效地利用了现有的基础结构。如果混合大量的边插入、边删除和连通性查询，这两种方法都不能有效地处理；二者都要求单独的DFS来计算路径。这些考虑说明了我们在分析图算法时所面临的复杂性。我们在第18.9节详细地研究这些问题。

双向欧拉回路　程序18.5使用DFS来解决查找路径问题，其中将图中的所有边均使用了两次，每个方向一次（第17.7节）。这条路径对应着一条Trémaux探索，其中走到哪里都带着线绳，并且检查线绳而不是使用灯（这样，对于已经访问过的交叉点，我们必须沿着指向该交叉点的通道向下走）。首先考虑在每条回链接上来回走过（第一次遇到每条回边时），然后忽略下链接（第二次遇到每条回边时）。我们还可以忽略回链接（第一次遇到），在下链接来回走过（第二次遇到）（见练习18.23）。

程序18.5　双向欧拉回路

此邻接矩阵表示法的DFS函数按照双向欧拉路径的顺序将每条边打印两次，每个方向一次。我们在在回边上来回走过，并忽略掉向下边（见正文）。

```
void dfsReuler(Graph G, Edge e)
  { link t;
    printf("-%d", e.w);
    pre[e.w] = cnt++;
    for (t = G->adj[e.w]; t != NULL; t = t->next)
      if (pre[t->v] == -1)
        dfsReuler(G, EDGE(e.w, t->v));
      else if (pre[t->v] < pre[e.v])
        printf("-%d-%d", t->v, e.w);
    if (e.v != e.w)
        printf("-%d", e.v);
    else printf("\n");
  }
```

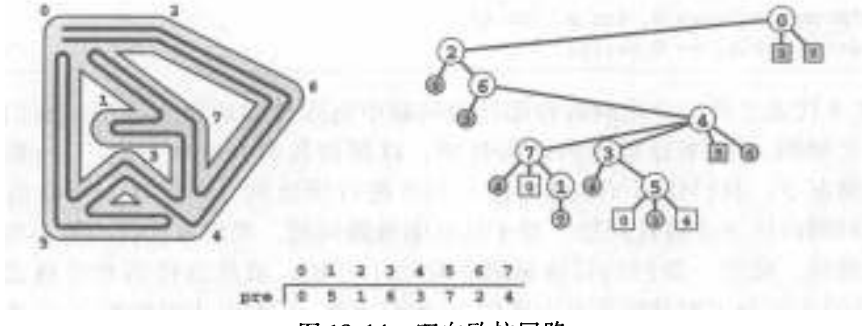


图18-14　双向欧拉回路

深度优先搜索给出了一种探索迷宫的方法，遍历每个方向的两个通道，我们修改Trémaux探索，不论走到哪里都带着线绳，而且对于通向已经访问过的交叉点的通道，在该通道上来回走过无需留下任何线绳。此图显示的顺序与图18-2和图18-3不同，这样做的主要原因是我们画出周游路径而不会自我交叉。这个顺序可能导致一些结果，例如，在构建图的邻接表表示时，按照某个不同的顺序来处理边，或者我们可能显式地修改DFS，将结点的几何位置考虑在内（见练习18.24）。沿着直通0的较低轨迹，我们从0到2、再到6、再到4，然后到7，之后从7到0，并且由于pre[0]小于pre[7]，将返回。然后走到1、回到7、回到4、到3、再到5，从5到0并返回，从5到4并返回，返回到3，返回到4，返回到6，返回到2，再返回到0。这条路径可以通过一个递归前序或后序遍历DFS树来得到（忽略那些有阴影的顶点，表示我们第二次遇到这些边），在打印顶点名时，递归访问这些子树，然后再次打印顶点名。

生成树 给定一个 V 个顶点的连通图，找出将 V 个顶点连接起来的 $V-1$ 条边的一个集合。DFS 解决了这个问题，因为任何一个 DFS 树都是一棵生成树。对于一个连通图，我们的 DFS 实现恰好执行 $V-1$ 次递归调用，生成树上的每条边执行一次，而且可以简单地使它产生该树的一个父链接表示，如在 18.4 节中开始所讨论的那样。如果此图有 C 个连通分量，我们可以找出有 V-C 条边的生成森林。在一个 ADT 函数的实现中，可能选择向图表示中添加父链接数组，形式如同程序 18.4，或者用客户提供的数组将它计算出来。

顶点搜索 在给定顶点所在的连通图中有多少个顶点呢？我们可以从该顶点开始 DFS，并对标记的顶点计数，就可以解决这个问题。在一个稠密图中，我们在标记出 V 个顶点之后使 DFS 终止，可以大大提高处理的速度。此后，我们知道不再会有边将把我们带到一个尚未见过的顶点，因此将忽略掉其余所有边。这一改进很可能使我们在与 $V \log V$ 而非 E 成正比的时间内访问所有顶点（见第 18.8 节）。

2-可着色性，二分图性，奇环 是否存在一种方式对图中的每个顶点指定两种颜色之一，满足不存在连接两个着相同颜色的顶点的边。给定的图是二分图吗（见 17.1 节）？给定的图含有长度为奇数的环吗？这三个问题都是等价的：前两个问题是同一问题的不同说法；任何具有奇环的图显然不是 2-可着色的，程序 18.6 表明不含奇环的任何图都是 2-可着色的。该程序是一个基于 DFS 的 ADT 函数实现，测试一个图是否是二分的、2-可着色的以及是否无奇环。该递归函数是用归纳法进行证明的一个框架，程序对任何不含奇环的图进行 2-着色（或者找出一个奇环来说明一个含有奇环的图不是 2 可着色的）。为了对一个图进行 2-可着色，先对一个顶点 v 指定一种颜色，然后对余图进行 2 着色，将与 v 邻接的顶点指定另一种颜色。这个过程随着 DFS 树的向下执行，等价于按照层次交替指定颜色，并在着色过程中检查回边是否存在不一致性，如图 18-15 所示。任何连接同一颜色的两个顶点的回边表明存在奇环。

这些基本的例子展示了 DFS 可以通过多种方式使我们深刻理解图的结构。它们还表明我们可以在单个线性时间内扫描图来了解图的各种重要性质，其中检查每条边两次，在每个方向上检查一次。接下来，我们考虑展示 DFS 用途的一个例子，它能发现关于图结构的更加复杂的细节，但仍在线性时间内完成。

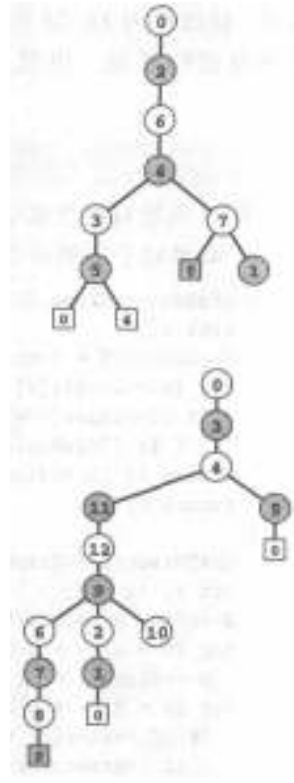


图 18-15 对 DFS 进行 2 着色

为了对一个图进行 2-着色，我们沿着 DFS 树交替着色，然后检查回边的不一致性。在上图的树中（图 18-9 中描述的示例图的一棵 DFS 树），回边 5-4 和 7-0 证明了此图不是 2-可着色的。因为分别存在奇数长度的环 4-3-5-4 和 0-2-6-4-7-0。在下图的树中（图 17-5 中二分图的一棵 DFS 树），由于没有这样的不一致性，其中所示的阴影就是图的一种 2-着色方案。

练习

○ **18.21** 给出图的邻接表表示的一个 ADT 函数实现，要求时间与 V 成正比，找出一个环并打印出来，或者报告环不存在。

18.22 描述 V 个顶点的一系列图，对于这些图，标准邻接矩阵 DFS 进行环检测所需的时间与 V^2 成正比。

▷ **18.23** 修改程序 18.5，产生一个双向欧拉路径，使其在下边而不是回边上做来回遍历。

● **18.24**　修改程序 18.5，使其总是产生一个类似图 18-14 中的双向欧拉回路，且该回路可画出而且在任何顶点上都不交叉。例如，如果图 18-14 中的搜索在取边 4-7 之前先取边 4-3，那么，该回路将会产生交叉；你的任务是保证算法避免这些情况。

18.25　对于图的邻接表表示（程序 17.6），开发程序 18.5 的一个版本，将图中的边按照双向欧拉回路的顺序排列。你的程序应该返回指向结点的循环链表的一个链接，该链表对应一个双向欧拉回路。

18.26　修改练习 18.25 的解决方案，使其可用于大型图，你可能没有空间来对每条边对应的表结点进行复制。也就是说，使用所分配的表结点来构建图，并销毁最初图的邻接表表示。

程序 18.6　2-可着色性

该 DFS 函数对顶点索引的数组 G->color 指定值 0 或 1。同时在返回值中表明是否能够进行指派，满足对于图的每条边 v-w，G->color［v］和 G->color［w］是不同的。

```
int dfsRcolor(Graph G, int v, int c)
  { link t;
    G->color[v] = 1-c;
    for (t = G->adj[v]; t != NULL; t = t->next)
      if (G->color[t->v] == -1)
        { if (!dfsRcolor(G, t->v, 1-c)) return 0; }
      else if (G->color[t->v] != c) return 0;
    return 1;
  }
int GRAPHtwocolor(Graph G)
  { int v, id = 0;
    G->color = malloc(G->V * sizeof(int));
    for (v = 0; v < G->V; v++)
      G->color[v] = -1;
    for (v = 0; v < G->V; v++)
      if (G->color[v] == -1)
        if (!dfsRcolor(G, v, 0)) return 0;
    return 1;
  }
```

18.27　证明一个图是 2-可着色的，当且仅当它不含奇环。提示：用归纳法证明程序 18.6 确定一个给定的图是否是 2-可着色的。

○ **18.28**　解释为什么不能将程序 18.6 中使用的方法进行推广，用来给出确定一个图是否是 3-可着色的高效方法？

18.29　大多数的图都不是 2-可着色的，DFS 很快就能发现这一点。对于由各种图模型所得的各种规模的图，进行实验测试来研究程序 18.6 所检查的边数。

○ **18.30**　证明每个连通图中存在顶点，将其删除之后使图不连通。并编写一个 DFS 函数找出这样一个顶点。提示：考虑 DFS 树的叶结点。

18.31　证明包含多于一个顶点的图中至少含有这样两个顶点，将它们删除之后不会增加连通分量的个数。

18.6　可分离性和双连通性

为了说明 DFS 作为图处理算法基础的强大能力，下面转向图中连通性的一般概念。我们

研究的问题如下：给定两个顶点，是否存在连接它们的两条不同路径？

如果图在某种情况下是连通的很重要，那么当一条边或一个顶点被删除后保持连通性也很重要。也就是说，在图中的每对顶点之间，我们可能希望它们之间有不止一条路径，这样就可以处理可能的失败情况。例如，即使芝加哥下雪，我们也可以取道丹佛从纽约飞往旧金山。或者想象战时的情景，我们希望安排铁路网，使敌方至少破坏两个车站才能切断我们铁路线。类似地，我们可能期望一个集成电路活通信网中的主要通信线路是连通的，即使某条线断了或某个链接中断，其余电路也能起作用。

这些例子说明了两个不同的概念：在电路和通信网中，我们感兴趣的是删除一条边仍能保持连通；在航空路线或铁路线中，感兴趣的是删除一个顶点仍能保持连通。以下开始对前者进行详细讨论。

定义 18.1 *图中的桥（bridge）是一条边，如果删除这条边，会将一个连通图分成两个不相交的子图。不含桥的图被称为是边连通的（edge-connected）。*

当我们谈到删除一条边时，意思是从定义图的边集中删除一条边，即使这个行为可能会使边的一个顶点或两个顶点变成孤立的。当我们删除任何单独边时，边连通图仍然是连通的。在某些情况下，强调使图不连通的能力，而非保持图的连通性的能力更为自然，因此我们交替使用有所强调的术语：我们将不是边连通的图称为*边可分离的图*（edge-separable graph），称桥为*可分离的边*（separable edge）。如果我们删除一个边可分离图中的所有桥，那么就将它分成*边连通分量*（edge-connected component）或*桥连通分量*（bridge-connected component）：即不含桥的最大子图。图 18-16 是说明这些概念的一个小例子。

乍一看，查找图中的桥似乎不是图处理中的一个简单问题，但这实际上是 DFS 的一个应用，在这里我们可以利用已经解释过的 DFS 树基本性质。具体地说，回边不能是桥，因为我们知道它们连接的两个结点还通过 DFS 树中的一条路径连通。而且，我们可以在递归 DFS 函数中添加一个简单测试，来测试树边是否是桥。以下对它的基本思想做了形式化阐述，如图 18-17 所示。

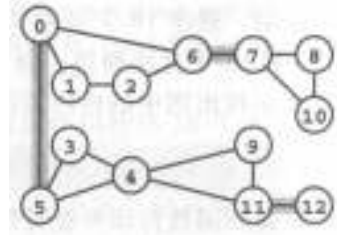


图 18-16 边可分离图

此图不是边可分离的。边 0-5、6-7 和 11-12（有阴影的）是分离边（桥）。此图有 4 个连通分量：一个连通分量包含顶点 0、1、2 和 6；另一个包含顶点 3、4、9 和 11；还有一个包含顶点 7、8 和 10；最后一个只含一个顶点 12。

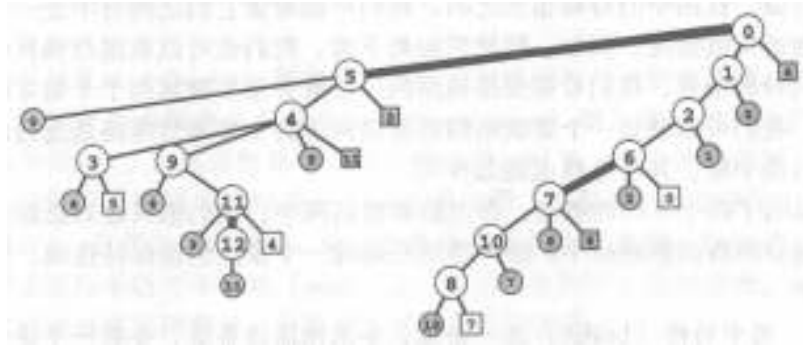
性质 18.5 *在任何 DFS 树中，一个树边 v-w 是一个桥，当且仅当不存在回边将 w 的子孙结点与 w 的祖先连接起来。*

证明 如果存在这样的一条边，v-w 就不可能是桥。反之，如果 v-w 不是桥，那么图中除了 w-v 自身外，必定存在从 w 到 v 的某条路径。每条这样的路径必定存在这样的回边。■

确定这个性质等价于说以 w 为根的子树中，指向不在该子树中的一个结点的唯一链接就是从 w 返回到 v 的父链接。这个条件成立，当且仅当连接 w 的子树中的任一结点与不在 w 的子树中的任一结点的路径上包含 v-w。换句话说，删除 v-w 将会使其余图与对应 w 的子树的子图不再连通。

程序 18.7 显示了使用性质 18.5，可以增强 DFS 来识别图中的桥。对于每个顶点 v，我们使用递归函数来计算一系列 0 或多条树边后跟一个回边所达到的最小前序编号，该回边来自以 v 为根的子树中的任何结点。如果所计算的编号等于 v 的前序编号，那么不存在将一个子孙结点与一个祖先连接起来的边，我们就识别出一个桥。对每个顶点的计算是直接的：我们使用邻接表进行处理，并记录沿着每条边所达到的最小顶点数。对于树边，进行递归计

算；对于回边，我们使用邻接顶点的前序编号。如果对某条边 w-v 调用递归函数没有发现指向某个结点且其前序编号小于 v 的前序编号的路径，那么 w-v 是一个桥。



	0	1	2	3	4	5	6	7	8	9	10	11	12
pre	0	7	8	3	2	1	9	10	12	4	11	5	6
low	0	0	0	1	1	1	0	10	10	2	10	2	6

图 18-17　查找桥的 DFS 树

对于图 18-16 中的图，其 DFS 树中的结点 5、7 和 12 都具有性质：不存在将子孙结点与祖先连接起来的回边，而且不存在具有这个性质的其他结点。因此，如图所示，断开这些结点中的某个结点与其父结点之间的边，将会使以该结点为根的子树与其余的图不再连通。也就是说，边 0-5、11-12 和 6-7 是桥。我们使用顶点索引数组 low 来保存以该顶点为根的子树中任何回边所引用的最小前序编号。例如，low［9］的值为 2，因为以 9 为根的子树中，其中一条回边指向 4（该顶点的前序编号为 2），而且不存在其他回边指向此树中的更高编号。结点 5、7 和 12 是那些其 low 值等于其 pre 值的结点。

性质 18.6　可以在线性时间内找出图的一个桥。

证明　程序 18.7 是对 DFS 稍做了修改，添加了几个常量时间的测试，因此可由性质 18.3 和 18.4 直接而得，对于邻接表表示找出图中的桥所需的时间与 V^2 成正比，对于邻接矩阵表示找出图中的桥所需的时间与 V + E 成正比。■

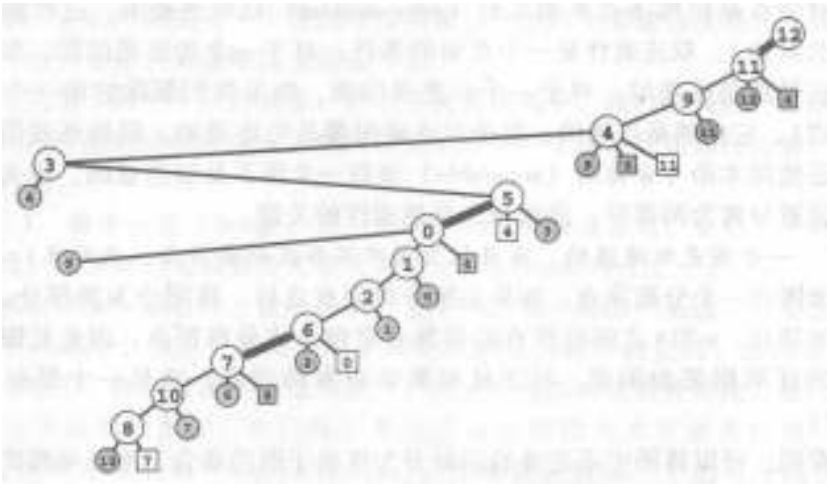
程序 18.7　边连通性（邻接表）

此递归函数打印并统计图中桥的个数。假设增强了程序 18.3 使其包含一个计数器 bnct 和一个顶点索引的数组 low，它们的初始化分别与 cnt 和 pre 的初始化过程相同。low 数组记录从每个顶点通过一系列树边后跟一条回边可达到的最小前序编号。

```
void bridgeR(Graph G, Edge e)
  { link t; int v, w = e.w;
    pre[w] = cnt++; low[w] = pre[w];
    for (t = G->adj[w]; t != NULL; t = t->next)
      if (pre[v = t->v] == -1)
        {
          bridgeR(G, EDGE(w, v));
          if (low[w] > low[v]) low[w] = low[v];
          if (low[v] == pre[v])
            bcnt++; printf("%d-%d\n", w, v);
        }
      else if (v != e.v)
        if (low[w] > pre[v]) low[w] = pre[v];
  }
```

在程序 18.7 中，我们使用 DFS 来找出图的性质。图表示肯定影响到搜索的顺序，但它并不影响搜索的结果。因为桥是图的一个特征，而不是我们选择表示或搜索图的特征。如

常，任何 DFS 树只是图的另一种表示，因此所有 DFS 树具有相同的连通性质。该算法的正确性取决于这个基本的事实。例如，图 18-18 描述了图的不同搜索（从不同顶点开始）当然会找出相同的桥。尽管有性质 18.6，当检查同一图的不同 DFS 树时，我们看到某些搜索开销可能不仅依赖于图的性质，而且依赖于 DFS 树的性质。例如，对于图 18-18 中的示例，支持递归调用的栈所需的空间量大于图 18-17 中的示例。



	0	1	2	3	4	5	6	7	8	9	10	11	12
pre	6	7	8	4	3	5	9	10	12	2	11	1	0
low	6	6	6	3	1	3	6	10	10	1	10	1	0

图 18-18　查找桥的另一种 DFS 树

对于图 18-16 中的图，此图显示了一个与图 18-17 中的图不同的 DFS 树，其中从一个不同结点开始搜索。尽管对结点和边的访问按照完全不同的顺序，我们仍然能够找出相同的桥。在这棵树中，0、7 和 11 就是那些 low 值等于 pre 值的结点。因此将它们之中任何一个与其父结点连接起来的边是桥（分别为 12-11、5-0 和 6-7）。

如同在程序 18.4 中对于正则连通性所做的那样，我们可能希望使用程序 18.7 来构建一个 ADT 函数，用以测试一个图是否是边连通的，或用它来计算边连通分量的个数。如果需要，我们可以像在程序 18.4 中那样进行处理，来创建 ADT 函数，使得客户端有能力在线性时间内调用预处理函数，然后在常量时间内响应询问两个顶点是否在同一个边连通分量中的查询（见练习 18.35）。

本节的最后将讨论连通性的其他一般情况，其中包括确定哪些顶点是保持一个图的连通性的关键。这里包含这些内容，目的是对于在第 22 章所考虑的更复杂的算法给出一个基本背景。如果你是初次接触连通性问题，可能希望跳过 18.7 节，并在学习第 22 章时再回到这里。

在谈到删除一个顶点时，还说明要删除它的所有依附边。如图 18-19 所示。删除桥上的任何一个顶点将会使图不连通（除非此桥是依附该顶点或这两个顶点的唯一边），但还有一些其他顶点与桥没有关系，也具有这个性质。

定义 18.2　图中的一个关结点（articulation point）是一个顶点，如果删除此顶点将把一个连通图至少分成两个不相交的子图。

我们也称关结点为分离顶点（separation vertex）或切

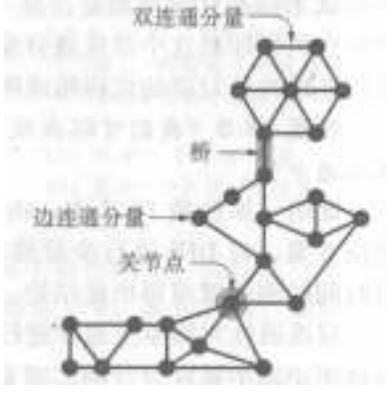


图 18-19　图可分性术语

此图中有两个边连通的分量和一个桥。桥上的边连通分量也是连通的；桥下的边连通分量由两个双连通分量组成，它们由关结点连接。

割点（cut vertex）。我们可以使用术语“顶点连通的”来描述一个不含分离顶点的图，但使用基于相关特性的另一个术语，它们是等价的。

定义 18.3 如果图中每一对顶点都由两条不相交的路径相连，则称该图是双连通的（biconnected）。

对路径做不相交的（disjoint）要求是用来区分双连通性与边的连通性。边连通度的另一定义是，每一对顶点是由两条边不相交的（edge-disjoint）路径连接的。这些路径可有公共顶点（但没有公共边）。双连通性是一个更强的条件：对于一个边连通的图，如果我们删除它的一条边，它仍然是连通的。对于一个双连通的图，如果我们删除它的一个顶点（以及该顶点的依附边），它仍然是连通的。每个双连通图都是边连通的，但边连通图不一定是双连通的。我们还使用术语可分离的（separable）来指一个图不是双连通的，因为只需去掉一个顶点，它们就被分离为两部分。分离顶点是连通性的关键。

性质 18.7 一个图是双连通的，当且仅当图中不存在分离顶点（关结点）。

证明 假设图有一个分离顶点。如果去掉分离顶点之后，将图分为两部分。令 s 和 t 是在这两部分中的顶点。s 和 t 之间的所有路径都必定包含该分离顶点，因此此图不是双连通的。另一方向的证明则更为困难，对于从事数学研究的读者，这是一个很有意义的练习（见练习 18.39）。■

我们已经看到，可以将图中不连通的边划分为连通子图的集合，而且可将图中不是边连通的边划分为桥和边连通子图（通过桥连接）的集合。类似地，我们可以将不是双连通的图划分为桥和双连通分量的集合，每个分量都是一个双连通子图。双连通分量和桥不是图的一个合适划分，因为关结点可能出现在多个连通分量中（例如，见图 18-20）。双连通分量在关节点处可能通过桥连接。

图的连通分量具有以下性质，图中的任何两个顶点之间存在一条路径。类似地，双连通分量具有这样的性质：任何一对顶点之间存在两条不相交的路径。

我们可以使用程序 18.7 中所用同样的基于 DFS 的方法来确定是否一个图是双连通的，并识别关结点。我们省略了此代码，因为它与程序 18.7 非常类似，只是加上了一个额外测试来检查 DFS 树根是否是一个关结点（见练习 18.42）。开发代码打印出这个双连通分量也是很有意义的练习，这只比边连通性所对应的代码稍微困难一些（见练习 18.43）。

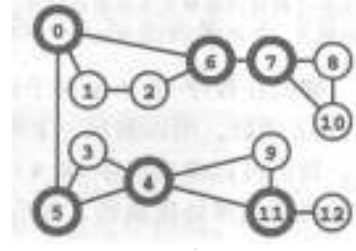


图 18-20 关结点（分离顶点）

此图不是双连通的。顶点 0、4、5、6、7 和 11（加阴影）是关结点。此图有 5 个双连通分量：一个包含边 4-9、9-11 和 4-11；另一个包含边 7-8、8-10 和 7-10；下一个包含边 0-1、1-2、2-6 和 6-0；还有一个包含边 3-5、4-5 和 3-4；最后一个包含单独顶点 12。增加一条边将 12 与 7、8 或 10 相连就可以使此图成为双连通图。

性质 18.8 我们可以在线性时间内找出图的关结点和双连通分量。

证明 如性质 18.7 中，由观察练习 18.42 和 18.43 的解决方案，对 DFS 进行少量修改，在每条边上增加一个常量时间的测试就可得出此结论。■

双连通性对简单连通性进行了推广。进一步推广一直是经典图论和图算法设计中广泛研究的主题。这些推广表明我们可能面临的图处理问题的范围，其中很多是容易提出但难于求解的问题。

定义 18.4 如果至少有 k 条不相交的路径连接图中每对顶点，则称此图是 k 连通的（k-connected）。图的顶点连通度（vertex connectivity）是将图分为两部分所需删除的最小顶点数。

在此术语中，“1 连通”等价于“连通”，而“2 连通”等价于“双连通”。对于有关结点的图，其顶点的连通性为 1（或 0），因此性质 18.7 指出，一个图是 2 连通的，当且仅当其顶点连通度不小于 2。这是图论中经典结果（称之为 Whitney 定理）的一个特例。该定理指出一个图是 k 连通的，当且仅当其顶点连通度不小于 k。由 Menger 定理（见 22.7 节）直接可得 Whitney 定理，Menger 定理是指，将图中的两个顶点变为不连通所需删除的最小顶点数等于这两个顶点之间顶点不相交的路径的最大个数（要证明 Whitney 定理，可以将 Menger 定理应用到每对顶点上）。

定义 18.5 *如果至少有 k 条边不相交的路径连接图中每对顶点，则称此图是 k 边连通的（k-edge-connected）。图的边连通度（edge connectivity）是将图分为两部分所需删除的最小边数。*

在此术语中，“2 边连通”等价于“边连通”（也就是说，一个边连通图其边的连通性大于 1）。至少含一个桥的图其边连通性为 1。Menger 定理的另一个版本指出，将图中的两个顶点变为不连通所需删除的最小顶点数等于这两个顶点之间顶点不相交的路径的最大个数，这蕴含着：一个图是 k 边连通的，当且仅当其边连通度为 k。

有了这些定义，我们可以对本节开始时所考虑的连通性问题进行推广。

***st* 连通性** 在一个给定图中，使两个给定顶点 s 和 t 分离所需删除的最小边数是多少？如果使给定图中的两个给定顶点 s 和 t 分离所需删除的最小顶点数是多少？

一般连通性 给定的一个图是 k 连通图吗？给定的一个图是 k 边连通图吗？给定图的边连通度和顶点连通度是多大？

比起本节讨论的简单连通性问题，这些问题的求解难度更大，但它们仍然是一大类图处理问题的成员，我们可以使用第 22 章（其中 DFS 起着重要的作用）讨论的通用算法学工具来求解；我们在 22.7 节讨论特殊解决方案。

练习

▷ **18.32** 如果图是一个森林，它的所有边是分离边；但它的哪些顶点是分离顶点？

▷ **18.33** 考虑下图

```
3-7  1-4  7-8  0-5  5-2  3-8  2-9  0-6  4-9  2-6  6-4
```

画出标准邻接表 DFS 树。使用它找出桥和边连通分量。

18.34 证明任何图中的每个顶点只属于一个边连通分量。

○ **18.35** 风格同程序 18.4，扩展程序 18.7 来支持 ADT 函数，用来测试是否两个顶点在同一个边连通分量中。

▷ **18.36** 考虑下图

```
3-7  1-4  7-8  0-5  5-2  3-8  2-9  0-6  4-9  2-6  6-4
```

画出标准邻接表 DFS 树。使用它找出关结点和双连通分量。

▷ **18.37** 使用标准邻接矩阵 DFS 树完成前一练习。

18.38 证明图中的每一条边，要么是一个桥，要么只属于一个双连通分量。

● **18.39** 证明没有关结点的任何图都是双连通的。提示：给定一对顶点 s 和 t，以及连接它们的一条路径，使用事实：此路径上不存在顶点是构造连接 s 和 t 的两条不相交路径的关节点。

18.40 修改程序 18.3 得到一个程序，用来确定是否一个图是双连通的，使用运行时间与 $V(V+E)$ 成正比的蛮力算法。提示：如果你在开始搜索之前，将一个顶点标记为已经看见过，则可从图中有效地删除它。

○ 18.41 扩展练习 18.40 中的解决方案得到一个程序，用来确定是否一个图是 3 连通的。给出一个公式，用于描述你的程序检查图中边的近似次数，该公式是 V 和 E 的一个函数。

18.42 证明：DFS 树根是关结点，当且仅当它有两个或多个（内部）子结点。

● 18.43 编写一个邻接表表示的 ADT 函数，打印图的双连通分量。

18.44 在一个有 V 个顶点的双连通图中，必定出现的最小边的个数是多少？

18.45 修改程序 18.3 和 18.7 来实现一个 ADT 函数，用来确定是否图是边连通的（如果图不是连通的，当它识别出桥时立即返回）。对于由各种图模型所得的各种规模的图（见练习 17.63 ~ 76），进行实验研究测试你的函数所检查的边数。

○ 18.46 改进程序 18.3 和 18.7，打印出关结点数、桥数和双连通分量数。

● 18.47 对于由各种图模型所得的各种规模的图（见练习 17.63 ~ 76），进行实验研究来确定练习 18.46 中所描述的各个量的平均值。

18.48 给出下图的边连通度和顶点连通度

```
0-1  0-2  0-8  2-1  2-8  8-1  3-8  3-7  3-6  3-5  3-4  4-6  4-5  5-6  6-7  7-8
```

18.7 广度优先搜索

假设我们希望找出图中两个特定顶点之间的*最短路径*（shortest path），这是一条连接这两个顶点且具有以下性质：不存在包含这两个顶点的且有更少边的其他路径。完成这项任务的经典方法称为*广度优先搜索*（BFS，breadth-first search），它也是很多图处理算法的基础，因此我们在本节详细讨论这个算法。DFS 对于解决这个问题提供的帮助有限，因为 DFS 遍历图的顺序与查找最短路径的目标没有关系。与此相反，BFS 则是基于这个目标。为了找出从 v 到 w 的一条路径，我们从 v 开始，并对于通过一条边所达到的所有顶点，检查其中的 w，然后对沿着两条边所达到的所有顶点进行检查，如此等等。

在进行图搜索时，如果碰到一个点有多于一条以上边可遍历，就选择其中一条，并将其他边保存起来以待以后探索。在 DFS 中，我们使用下推栈（可由支持递归搜索函数的系统来管理）可达此目的。使用作为下推栈特征的后进先出（LIFO）规则对应于在一个迷宫中探索就近的通道，在已探索的通道中，我们选择最近遇到的通道。在 BFS 中，我们希望按照与起始点的远近顺序来探索顶点。对于一个迷宫，用这个顺序进行探索可能需要一个搜索小组；但在计算机程序中易于处理：只需使用（先进先出 FIFO）队列代替栈就可以了。

对于邻接矩阵表示的图，程序 18.8 是 BFS 的一种简单实现。它是基于维护所有边的一个队列，其中的边将访问过的顶点与一个未访问的顶点连接。我们在队列的起始顶点上设置一个虚拟自环，然后执行如下步骤直到队列为空：

- 从队列中取边，直到找到一条边，它指向一个未被访问的顶点。
- 访问该顶点；将从此顶点到未被访问顶点所有边放入队列中。

图 18-21 显示了 BFS 在示例上一步一步进行的过程。

如我们在 18.4 节中所见的，DFS 类似一个人探索迷宫的过程。BFS 则类似一组人在所有方向散开探索的过程。尽管 DFS 和 BFS 在很多方面都不同，但在这两种方法之间有种基本的潜在关系。这个关系在第 5 章简要介绍这些方法时已经提到。在第 18.8 节中将考虑一种推广的图搜索方法，我们会专门讨论这两个算法以及大量其他算法。每个算法都有用于求解相关图处理问题的特殊的动态特征。对于 BFS，从每个顶点到起始顶点的距离（连接这两个顶点的最短路径的长度）是我们感兴趣的关键性质。

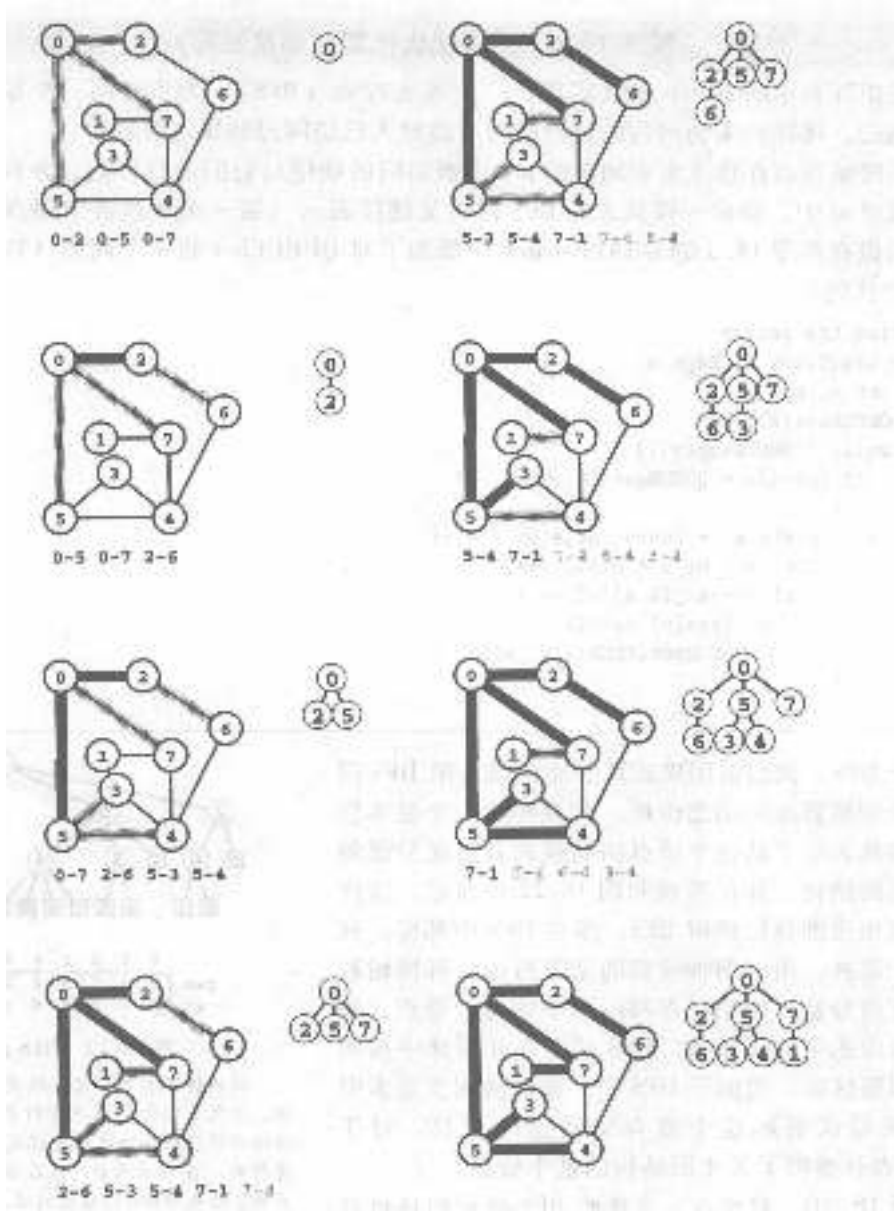


图 18-21 广度优先搜索

此图跟踪了 BFS 在示例图上的操作过程。我们从队列中与起始顶点邻接的所有边开始（左上图）。接下来，将边 0-2 从队列移到树中，并处理它的依附边 2-0 和 2-6（左边上数第二个图），我们并不将 2-0 放入队列，因为 0 已经在树中。然后，将边 0-5 从队列移到树中；同样的 5 的依附边（5 到 0）未依附任何新结点，但将 5-3 和 5-4 添加到队列中（左图从上图数第三个图）。接下来，我们将 0-7 添加到树中，并将 7-1 放入队列中（左下图）。

边 7-4 打印成灰色，因为我们同样未将它放入队列中，由于还存在另一条边将我们带到已在队列中的结点 4。为了完成搜索，我们将余下的边从队列中取出，对于那些出现在队列前面的灰色边则完全忽略（右图）。边是按照它们与 0 的距离的大小进入和离开队列的。

性质 18.9 在 BFS 中，顶点按照它们与起始顶点的距离大小进入或离开 FIFO 队列。

证明 有一个更强的性质也成立：队列总是包含 0 个或与起始顶点距离为 k 的更多顶点，其后跟着与起始顶点距离为 $k+1$ 的更多顶点，k 是某个整数。这个更强的性质很容易由归纳法证明。 ■

程序 18.8　广度优先搜索（邻接矩阵）

邻接矩阵表示的 search 函数实现是广度优先搜索（BFS）。为了访问一个顶点，我们扫描其依附边，将指向未访问过的顶点的所有边放入已访问过的顶点队列中。

我们按照顶点在顶点索引的数组 pre 中被访问的顺序对它们进行标记，并在另一个顶点索引的数组 st 中，构建一棵显式的 BFS 树的父链接表示（第一次到达各个顶点的那些边）。此代码假设在程序 18.3 的 GRAPHsearch 中添加了对 QUEUEinit 的一个调用（以及声明和初始化 st 的代码）。

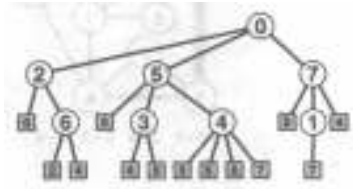
```
#define bfs search
void bfs(Graph G, Edge e)
  { int v, w;
    QUEUEput(e);
    while (!QUEUEempty())
      if (pre[(e = QUEUEget()).w] == -1)
        {
          pre[e.w] = cnt++; st[e.w] = e.v;
          for (v = 0; v < G->V; v++)
            if (G->adj[e.w][v] == 1)
              if (pre[v] == -1)
                QUEUEput(EDGE(e.w, v));
        }
  }
```

对于 DFS，我们借用描述算法递归结构的 DFS 搜索森林来理解算法的动态性质。此森林的一个基本性质是：森林表示了从各个顶点回到搜索其连通分量的开始位置的路径。如在实现和图 18-22 中所示，这样的生成树也帮助我们理解 BFS。像在 DFS 中那样，我们有一个森林，用以刻画搜索的动态行为，每棵树对应一个连通分量，树中结点对应图中的每个顶点，每条树边对应图中的每条边。BFS 对应在此森林中按照层序遍历每棵树。类似于 DFS 中，我们使用顶点索引的数组来显式表示这个带有父链接的森林。对于 BFS，此森林携带了关于图结构的基本信息。

性质 18.10　对于以 v 为根的 BFS 树中的任何结点 w，从 v 到 w 的树路径对应着在相应图中从 v 到 w 的一条最短路径。

证明　从队列中取出的结点到根的树路径长度是非递减的，而且比 w 更靠近根的所有结点都在这条路径上；因此在将 w 从队列中取出之前，不会发现到 w 的更短路径，也不会在将 w 从队列取出之后，所发现的到 w 的路径要比 w 的树路径长度更短。 ■

如在图 18-21 所指示和第 5 章所提到的，如果有相同目的顶点的一条边已在队列中，则不需要将有该目的顶点的任何边放入队列中，因为 FIFO 策略保证



	0	1	2	3	4	5	6	7
pre	0	7	1	5	6	2	4	3
st	0	7	0	5	5	0	2	0

图 18-22　BFS 树

此树提供了 BFS 动态性质的一个简洁描述，方式类似于图 18-9 中所示的树。按照层序遍历这棵树可知搜索过程是如何一步一步进行的：首先访问 0；然后访问 2、5 和 7；再由 2 检查得知 0 已被访问过，则访问 6；如此继续。树中每个结点都有一个表示与它相邻的每个结点的子结点，其顺序即为 BFS 处理时的顺序。像在图 18-9 中一样，BFS 树中的链接对应着图中的边：如果我们用指向指示结点的线替换掉指向外部结点的边，就可以绘制出此图。指向外部结点的链接表示不放入队列中的边，因为它们会通向已标记过的结点：它们可能是父链接，也可能是指向同层或与根更近一层的某个结点的交叉链接。

st 数组是树的一种父链接表示，我们可以使用它来找出从任何结点到根的一条最短路径。例如，3-5-0 是此图中从 3 到 0 的一条最短路径，因为 st[3] 为 5，st[5] 为 0。不存在从 3 到 0 的其他更短的路径。

了我们在得到新边之前先处理队列中的旧边（并访问该顶点）。实现此策略的一种方法是使用一个队列 ADT 实现，通过“忽略新项”策略（见 4.7 节）来禁止这样的重复。另一种选择是使用全局顶点标记的数组完成此目的：不是在从队列中取出顶点，将它标记为已经被访问过，而是在将它放入队列时就做标记。测试是否一个顶点已被标记过（是否它的元素值已不同于其初始标记值有所改变），然后对于任何指向队列中的同一顶点的其他边则不放入队列中。程序 18.9 中所显示的这种变化给出了 BFS 的一种实现，其中队列中的边数不会超过 V 条（每条边至多指向一个顶点）。

程序 18.9 所对应的关于图的 BFS 的这段代码用邻接表表示，简洁直接，并可由我们在 18.8 节中所考虑的一般图搜索方法得出，因此我们这里不再重复。与 DFS 一样，我们将 BFS 考虑为一个线性时间的算法。

程序 18.9 改进 BFS

为了保证在 BFS 过程中我们使用的队列至多有 V 个元素，当将顶点放入队列中时，对顶点进行标记。

```
void bfs(Graph G, Edge e)
  { int v, w;
    QUEUEput(e); pre[e.w] = cnt++;
    while (!QUEUEempty())
      {
        e = QUEUEget();
        w = e.w; st[w] = e.v;
        for (v = 0; v < G->V; v++)
          if ((G->adj[w][v] == 1) && (pre[v] == -1))
           { QUEUEput(EDGE(w, v)); pre[v] = cnt++; }
      }
  }
```

性质 18.11 对于邻接矩阵表示，BFS 访问图中的所有顶点和边，所需时间与 V^2 成正比，对于邻接表表示，则与 $V+E$ 成正比。

证明 如同在证明关于 DFS 性质一样，通过检查代码注意到，对于访问的每个顶点，都只检查其邻接矩阵行或邻接表中的每个元素一次，因此这足以表明我们访问了每个顶点。现在，对于每个连通分量，算法保持以下不变式：由起始顶点可达的所有顶点（ⅰ）都在 BFS 树上，（ⅱ）都在队列中，或（ⅲ）可由队列中的某个顶点达到。每个顶点可从（ⅲ）移到（ⅱ）再到（ⅰ），而且（ⅰ）中的顶点个数在循环的每次迭代中增加，因此 BFS 树最终包含由起始顶点可达的所有顶点。 ■

有了 BFS，我们可以解决生成树、连通分量、顶点搜索以及在 18.4 节中描述的其他基本连通性问题，这是因为对于我们所考虑的解决方案，只依赖于检查连接到起始顶点的每个顶点和边的搜索能力。更重要的是，就像在本节开始提到的，如果我们希望了解两个特定顶点之间的最短路径，BFS 是自然的图搜索算法。接下来，我们考虑此问题的一种特定解决方案，以及用于解决另外两个相关问题的扩展算法。

最短路径 找出图中从 v 到 w 的一条最短路径。我们可以开始 BFS 完成这项任务，维护在 v 的搜索树的父链接表示 st，然后在到达 w 时停止。从 w 到 v 的树中向上的路径是一条最短路径。例如，以下代码将 w 与 v 连接的路径打印出来：

```
for (t = w; t !=v; t = st[t]) printf("%d-", t);
printf("%d\n", t);
```

如果我们希望得到从 v 到 w 的路径，可以用栈压入操作代替这段代码中的 printf 操作，然后进入循环，一旦从栈中弹出顶点索引，则打印出其索引。或者在 w 开始搜索，并到开始的 v 停止。

单源点最短路径 找出图中连接给定顶点 v 与每个其他顶点的最短路径。以 v 为根的完全 BFS 树提供了完成这项任务的一种方法：从每个顶点到根的路径是到根的最短路径。因此，为了解决这个问题，可以从 v 开始运行 BFS 来完成。由此计算得到的数组 st 是此 BFS 的一个父链接表示。上面的代码可以提供到任何其他顶点 w 的最短路径。

所有对之间的最短路径 找出连接图中每对顶点之间的最短路径。完成此项任务的方法是运行 BFS 来求解图中每个顶点的单源点最短路径问题，并且支持 ADT 函数，它们能够处理大量最短路径有效查询，并将每个顶点的路径长度和父链接树表示保存起来（见图 18-23）。这个预处理过程所需时间与 VE 成正比，所需空间与 V^2 成正比。对于大型稀疏图开销巨大。然而，它使我们可以构建一个具有最优性能的 ADT：做了预处理（并预留保存结果的空间）之后，我们可以在常量时间内返回最短路径长度，而且可在与其长度成正比的时间内返回路径自身（见练习 18.53）。

这些基于 BFS 树的解决方案是有效的，但我们这里并未考虑进一步的实现细节，因为它们是第 21 章详细考虑的算法的特例。在图中术语*最短路径*一般会被认为可用来描述有向图和网中的相应问题。第 21 章主要研究这个专题。在那里我们考察的解决方法是对这里描述的基于 BFS 的方法的严格推广。

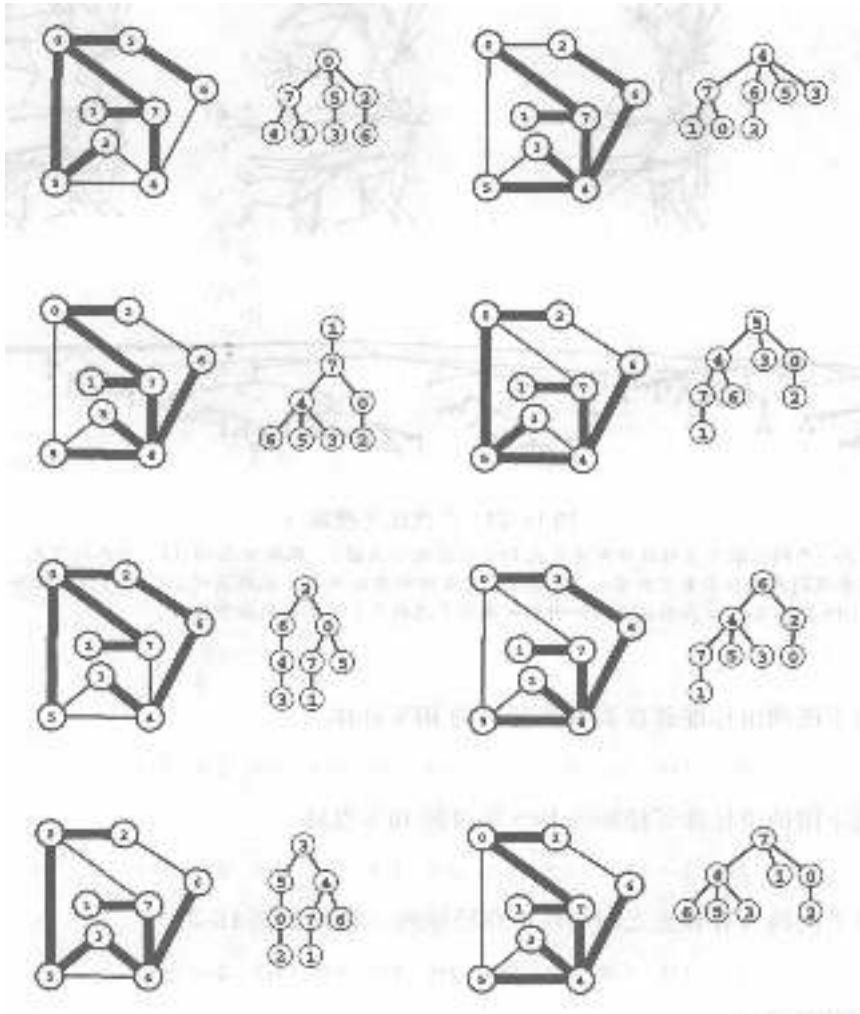
BFS 搜索动态性的基本特征与 DFS 搜索动态性的那些特征形成了鲜明的对比。如在图 18-24 中对于大型图所显示的那样，你应该将它与图 18-13 进行比较。这棵树矮且宽，这表明了两种搜索过程有着的不同。例如，

- 存在一条相对短的路径将图中的每对顶点连接起来。
- 在搜索过程中，大多数顶点与很多未被访问的顶点相邻。

  同样，这个例子也代表了我们期望 BFS 的一种行为，但验证这类图模型的结果是有趣的，且对于实际中出现的图还需做详细分析。

DFS 纵向遍历图，将有其他分支路径的顶点保存在栈中；BFS 横向遍历图，使用队列来记住已访问过的区域。DFS 通过查找远离起始点的新的顶点来探索图，只有当遇到走不通的路时才取较近的顶点；BFS 则完全覆盖与起始点相近的区域，只有检查了每一个相近的顶点后才向更远的位置移动。访问顶点的顺序取决于图的结构和表示方法，但搜索树的这些全局性质更多的是由算法体现出来，而非图或其表示体现。

理解图处理算法的关键是要认识到，不仅各种不同的搜索策略是了解各种图的不同性质的有效途径，而且还可以一致地实现这些策略。例如，图 18-13 中所描述的 DFS 告诉我们图中有一条较长的路径，而图 18-24 中描述的 BFS 指出有多条较短的路径。尽管存在这些显著的动态差异，DFS 和 BFS 还是很相似的，基本上只在用于保存尚未探索的边的数据结构上有所不同（另外实现递归 DFS 时，系统维护一个隐式栈所用的环境不同）。实际上，我们下面会转到涵盖 DFS、BFS 以及很多其他有用策略的广义的图搜索算法。这将会成为大量经典图处理问题的解决方案的基础。



	0	1	2	3	4	5	6	7
0	0	2	1	2	2	1	2	1
1	2	0	3	3	2	3	3	1
2	1	3	0	3	2	2	1	2
3	2	3	3	0	1	1	2	2
4	2	2	2	1	0	1	1	1
5	1	3	2	1	1	0	2	2
6	2	3	1	2	1	2	0	2
7	1	1	2	2	1	2	2	0

	0	1	2	3	4	5	6	7
0	0	7	0	5	7	0	2	0
1	7	1	0	4	7	4	4	1
2	2	7	2	4	6	0	2	0
3	5	7	0	3	3	3	4	4
4	7	7	6	4	4	4	4	4
5	5	7	0	5	5	5	4	4
6	2	7	6	4	6	4	6	4
7	7	7	0	4	7	4	4	7

图 18-23 所有对顶点之间的最短路径示例

这些图显示了从每个顶点执行 BFS 的结果，因此计算出连接每对顶点之间的最短路径。每次搜索给出一棵 BFS 树，它定义了连接图中所有顶点与根顶点之间的最短路径。所有搜索结果都保存在下图的两个矩阵中。在左边的矩阵中，第 v 行与第 w 列中的元素给出了从 v 到 w 的最短路径长度（v 在 w 所在树中的深度）。右边矩阵的每一行包含表示对应搜索的 st 数组。例如，从 3 到 2 的最短路径有 3 条边，如左边矩阵行 3 和列 2 中的元素所表明的。由左边从上数第 3 个 BFS 得知，路径为 3-4-6-2，这个信息编码在右边矩阵的行 2 中。在有一条以上的最短路径时，该矩阵未必是对称矩阵，因为所找到的路径依赖于 BFS 搜索顺序。例如，由左边下图的 BFS 树和右边矩阵的行 3 可知，从 2 到 3 的最短路径是 2-0-5-3。

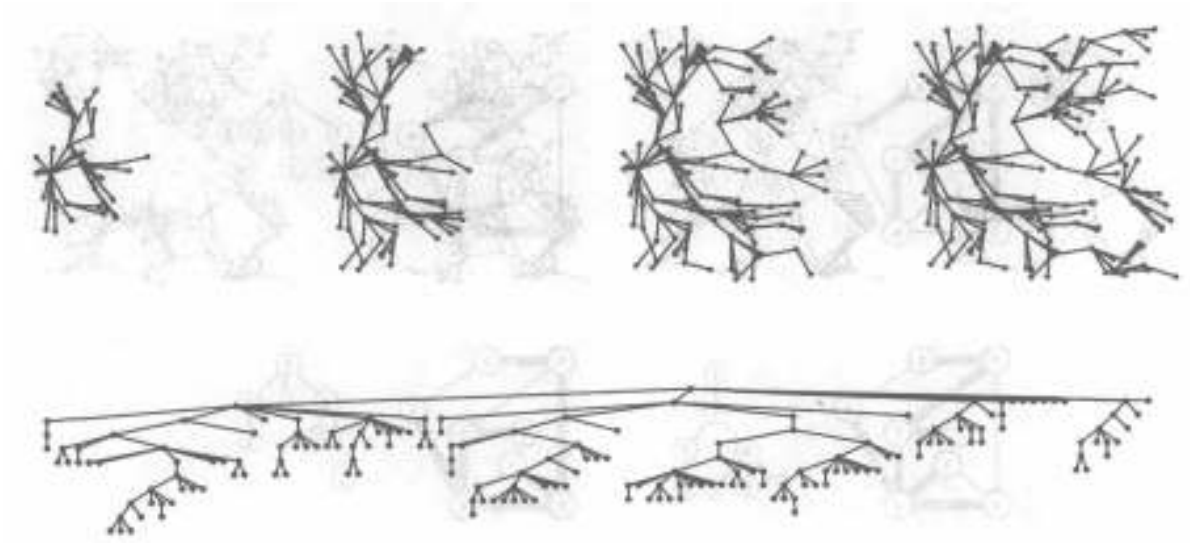


图 18-24 广度优先搜索

此图描述了在一个随机欧几里得近邻图中完成 BFS 的过程（左图）。风格如图 18-13。由此例可见，对于这种类型的图（实际中还会遇到其他很多类型的图），其 BFS 的搜索树相当矮而宽。也就是说，顶点通常由相当短的路径连接到另一个顶点。DFS 树和 BFS 树在形状上的鲜明差异表明了这两个算法不同的动态性质。

练习

18.49 画出下图的由标准邻接表 BFS 所得的 BFS 森林。

3-7 1-4 7-8 0-5 5-2 3-8 2-9 0-6 4-9 2-6 6-4

18.50 画出下图的由标准邻接矩阵 BFS 所得的 BFS 森林。

3-7 1-4 7-8 0-5 5-2 3-8 2-9 0-6 4-9 2-6 6-4

18.51 给出下图的所有顶点之间的最短路径矩阵（风格如图 18-23）

3-7 1-4 7-8 0-5 5-2 3-8 2-9 0-6 4-9 2-6 6-4

假设使用邻接矩阵表示。

○ **18.52** 给出 BFS 的一种实现（程序 18.9 的一个版本），使用顶点的标准 FIFO 队列。在 BFS 的搜索代码中包含一个对队列中不含重复元素的测试。

18.53 开发一个图 ADT（程序 17.6）的邻接表实现的 ADT 函数，用来支持在计算出每对顶点之间最短路径的预处理之后，对最短路径的查询。具体地说，添加两个指向图表示的数组指针，编写一个预处理函数 GRAPHshortpaths，并如图 18-23 中描述的那样，为其所有元素指定值。然后，增加两个查询函数 GRAPHshort（v，w）（返回 v 和 w 之间的最短路径长度）和 GRAPHpath（v，w）（返回 v 和 w 的一条最短路径上与 v 邻接的顶点）。

▷ **18.54** 当 v 和 w 都不是 BFS 树中的根结点时，它们之间的距离表示什么？

18.55 开发一个图 ADT 函数，用来返回足以将每对顶点连接起来的路径长度。（这个量称为图的直径）。注意：对于不连通的图，你需要为返回值定义一个约定。

18.56 给出一种简单最优递归算法，找出树的一个直径（见练习 18.55）。

○ **18.57** 对于邻接表表示的 BFS 进行实现，打印出 BFS 树的高度和要看到的每个顶点所必需处理的边的百分比。

● **18.58** 对于不同规模和来自不同图模型（见练习 17.63～76）的图，进行实验来确定练习

18.57 中所描述的各种量的平均值。

18.8 广义图搜索

DFS 和 BFS 是基础和基本的图遍历方法，它们是许多图处理算法的核心。了解它们的基本性质后，我们现在转到更高一级的抽象，我们将会看到，这两种方法都是遍历图的一个广义策略的特例。这在 BFS 实现中提到过（见练习 17.63～76）。

基本思想很简单：回顾在 18.6 节中对于 BFS 的描述，但这里我们使用通用的术语边缘集（fringe），而不是队列（queue）来描述下一次向树中添加的可能候选边的集合。我们可以直接得到搜索图的连通分量的一个通用策略。从指向边缘集中一个起始顶点的自环和一个空树开始，执行以下操作直到边缘集为空：

> 将一条边从边缘集中移到树中。如果它所到的顶点未被访问过，就访问该顶点，并将由此顶点所能到达未访问顶点的所有边放入边缘集中。

此策略描述了访问图中所有顶点和边的一类搜索算法，它与使用哪种广义队列来保存边缘集中的边无关。

当我们使用队列来实现边缘集时，就得到 BFS。这是 18.6 节中所讨论的专题。当我们使用栈来实现边缘集时，就得到 DFS。与图 18-6 和图 18-21 比较，图 18-25 详细说明了这一现象。证明递归 DFS 和基于栈的 DFS 的等价性是消除递归的一个有趣的练习。实质上是将递归程序底层的栈转换为实现边缘集的栈（见练习 18.61）。图 18-25 中所描述的 DFS 的搜索顺序与图 18-6 中所描述的 DFS 的搜索顺序不同的原因在于，栈规则表明，我们检查依附每个顶点的边的顺序与在邻接矩阵（或者邻接表）中遇到它们的顺序正好相反。如果我们用栈代替队列（这样做很简单，因为这两个数据结构的 ADT 接口仅函数名有所不同），改变程序 18.8 中使用的数据结构，就将 BFS 的程序变成了 DFS 程序，基本事实仍然成立。

如在 18.7 节中所讨论的，现在这个通用的方法不如我们希望的那么高效，因为边缘集中散乱着放着指向顶点的边，当边在边缘集中时，其顶点已被移到树中。对于 FIFO 队列，我们可以在将边放入队列时标记目的顶点来避免这种情况。我们忽略掉指向边缘顶点的那些边，因为我们知道这些边从不会被使用：旧边（及所访问的顶点）将在新边之前出队列（见程序 18.9）。对于栈实现，我们希望正相反：当一条边被添加到边缘集中时，如果其中已有相同目的顶点的边存在，那么旧边永远不会用到，因为新边（及所访问的顶点）将在旧边之前出栈。为了将这两种极端情况包括在内，并允许边缘集实现可以使用一些其他策略，从而禁止边缘集中的边指向同一顶点，我们将通用方案修改如下：

> 将边缘集中的一条边移到树中。访问它指向的顶点，并将由该顶点指向的未访问顶点的所有边放入边缘集中，在边缘集中使用一种替换策略以保证其中任意两条边都不指向同一顶点。

这种边缘集的“无重复目的顶点”策略可以保证，对于离开边缘集的边，无需检查其目的顶点是否已经访问过。对于 BFS，我们使用了一个带有“忽略新元素”策略的队列实现；对于 DFS，则需要一个带有“忘记旧元素”策略的栈；然而，由任何广义队列和任何替换策略都可以得到一种有效的方法，使得可在线性时间内访问图的所有顶点和边，而且额外的空间需求与 V 成正比。图 18-27 是对这些差别的图示说明。我们有一系列的图搜索策略，其中包括 DFS 和 BFS，还有其他一些策略，这些策略只是所用的广义队列实现有所不同。我们将看到，这个图搜索策略系列还涵盖了许多其他的经典图处理算法。

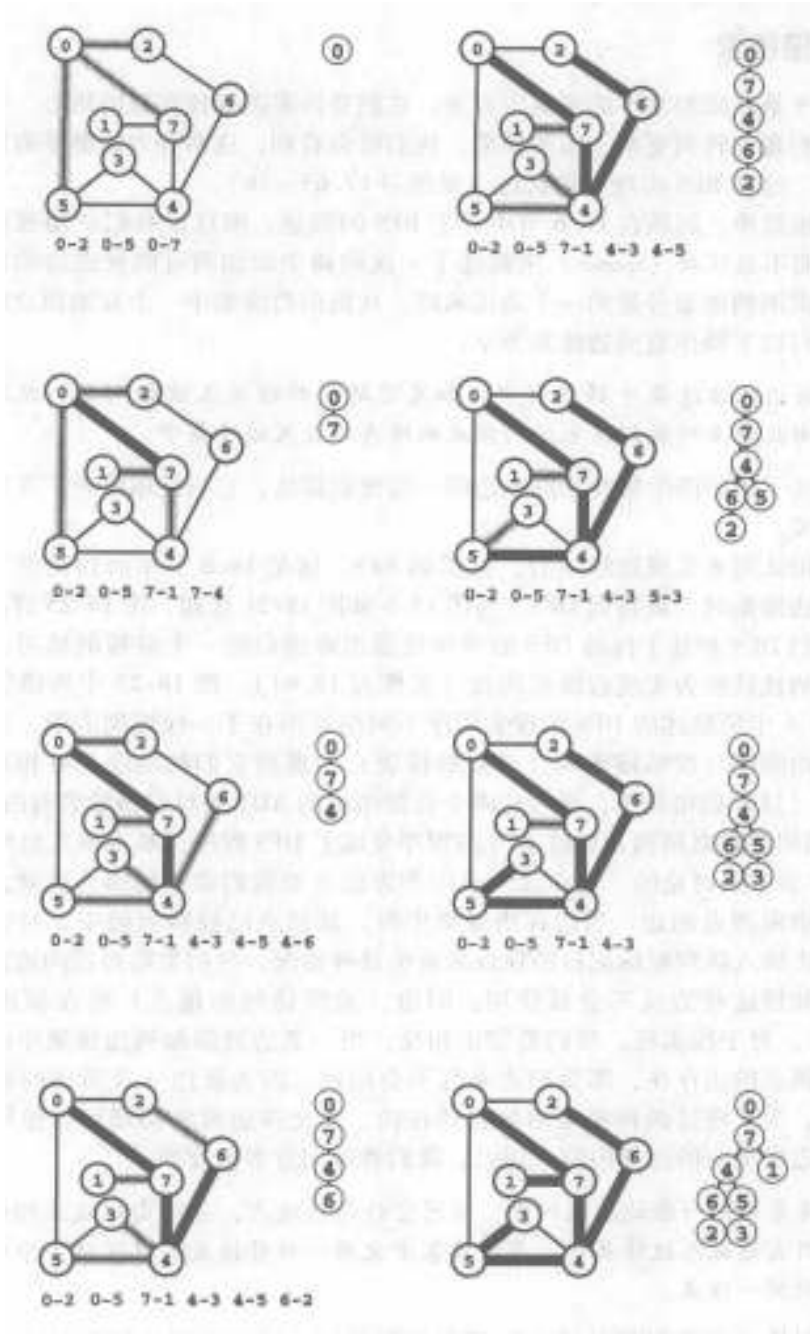


图 18-25　基于栈的 DFS

结合图 18-21，此图说明了 BFS 和 DFS 只在基本数据结构上不同。对于 BFS，我们使用队列；对于 DFS，则使用栈。我们从与栈中的起始顶点邻接的所有边开始（左上图）。第二，从栈中将边 0-7 移到树中，并将其指向的尚未在树中的顶点的依附边 7-1 和 7-4 压入栈中（左边从上数第二个图）。LIFO 栈规则表明，当我们向栈中放入一条边时，指向同一顶点的任何边都将过时，且当它们到达栈顶时将被忽略。这种边用灰色打印出。第三，我们将栈中的边 7-4 移到树中，并将其依附边压入栈中（左边从上数第三个图）。接下来，弹出 4-6，并将其依附边压入栈中，其中有两条边指向新的顶点（左下图）。为了完成搜索，我们从栈中取出其余边，完全忽略掉到达栈顶的那些灰色的边（右图）。

对于采用邻接表表示的图，程序 18.10 是基于以上这些思路的一种实现。它将边缘边放入一个广义队列中，并使用通常的顶点索引数组来识别边缘顶点，从而在遇到指向一个边缘顶点的另一条边时，可以使用一种显式的更新 ADT 操作。此 ADT 实现可以选择忽略新边或替换旧边。

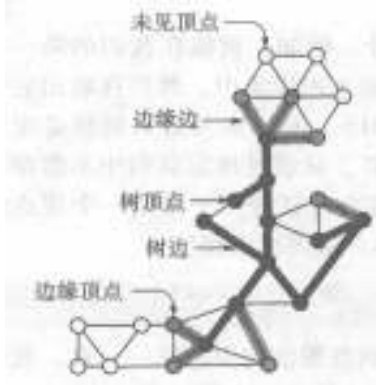


图 18-26 图搜索术语

在图搜索过程中，我们维持一棵搜索树（黑色）和一个边缘集（灰色），它们是下一次将要添加到树中的候选边。每个顶点要么在树中（黑色），要么在边缘集中（灰色），要么尚未被看见（白色）。树顶点由树边连接起来，每个边缘顶点由边缘边与树顶点连接。

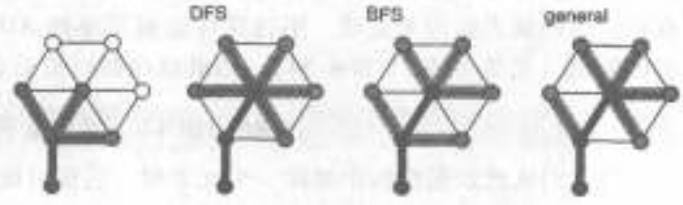


图 18-27 图搜索策略

对于图 18-26 中描述的搜索过程，此图描述了当我们进行下一步时的几种可能性。我们将边缘集中的一个顶点移到树中（右上图所示的转轮中心点），并检查它的所有边，将指向未见过顶点的那些边放入边缘集中，并使用特定算法的替换规则来决定是否要跳过指向边缘顶点的那些边，或者应该替换掉指向同一顶点的边缘边。在 DFS 中，我们总是替换掉新边，而且一般而言，我们可能替换一些边并且跳过另一些边。

程序 18.10 广义图搜索

程序 18.3 的 search 实现推广了 BFS 和 DFS，并支持许多其他图处理算法（见 21.2 节对于这些算法的讨论以及其他实现）。它维持了称为边缘集的一个边的广义队列。我们使用一个指向起始顶点的自环来初始化该边缘集；然后在边缘集不为空时，将边缘集中的边 e 移到树中（依附于 P[e.v]），并扫描 e.w 的邻接表，并将未见过的顶点移到边缘集中，同时对于指向边缘顶点的新边调用 GQupdate。

这段代码巧妙地使用了 pre 和 st，以保证边缘集中不存在两条边指向同一顶点。顶点 v 是边缘边的目的顶点，当且仅当它已做标记（pre[v]非负）但尚未出现在树中（st[v]为 -1）。

```
#define pfs search
void pfs(Graph G, Edge e)
  { link t; int v, w;
    GQput(e); pre[e.w] = cnt++;
    while (!GQempty())
      {
        e = GQget(); w = e.w; st[w] = e.v;
        for (t = G->adj[w]; t != NULL; t = t->next)
          if (pre[v = t->v] == -1)
            { GQput(EDGE(w, v)); pre[v] = cnt++; }
          else if (st[v] == -1)
            GQupdate(EDGE(w, v));
      }
  }
```

性质 18.12 在一个规模为 V 的广义队列中，对于邻接矩阵表示，广义图搜索访问图中的所有顶点和边所需时间与 V^2 成正比，对于邻接表表示，所需时间与 $V+E$ 成正比，加上在最坏情况下，V 次插入、V 次删除和 E 次更新操作所需的时间。

证明 性质 18.11 的证明并不依赖于队列实现，因此可以适用于此。这里所阐述的广义队列操作所需的额外时间可由实现直接而得。 ■

还有其他很多可以考虑的用于边缘集的有效的 ADT 设计。例如，就像在我们的第一个 BFS 实现中，可以坚持第一种通用模式，并简单地将所有边放入边缘集中，然后在取出它们时忽略掉那些指向树顶点的边。这种方法的缺点是，像在 BFS 一样，队列最大规模必定为 E，而不是 V。或者，我们可以在 ADT 实现中隐式地处理更新，只通过指定队列中不能存在有相同目的顶点的边来完成。不过这样做最简单的 ADT 实现方法就等价于使用一个顶点索引的数组（见练习 4.51 和 4.54），因此这项测试更适合于客户端图搜索程序。

程序 18.11 随机队列实现

当我们从此数据结构中删除一个元素时，它很可能是当前数据结构中的任一元素。我们可以使用这段代码实现图搜索的广义队列 ADT 以“随机”方式搜索一个图（见正文）。

```
#include <stdlib.h>
#include "GQ.h"
static Item *s;
static int N;
void RQinit(int maxN)
  { s = malloc(maxN*sizeof(Item)); N = 0; }
int RQempty()
  { return N == 0; }
void RQput(Item x)
  { s[N++] = x; }
void RQupdate(Item x)
  { }
Item RQget()
  { Item t;
    int i = N*(rand()/(RAND_MAX + 1.0));
    t = s[i]; s[i] = s[N-1]; s[N-1] = t;
    return s[--N];
  }
```

结合程序 18.10 和广义队列抽象，可以得到一种通用且灵活的图搜索机制。为了说明这一点，下面简要考虑两种可以替换 BFS 和 DFS 的很有意思且很有用的策略。

第一种候选策略所基于的是随机队列（randomized queue）（见 4.6 节）。在随机队列中，我们随机地删除元素：数据结构中的每个元素被删除的几率都相同。程序 18.11 时提供此功能的一个实现。如果我们使用此代码为程序 18.10 实现广义队列 ADT，则得到一个随机化的图搜索算法，其中边缘集中的每个顶点都有相同的几率作为下一个顶点被添加到树中。将要添加到树中（指向该顶点）的边则取决于更新（update）操作的实现。程序 18.11 中的实现并没有更新，因此每个边缘顶点被添加到树中时，同时会造成此移动的边移至边缘集中。另一种做法，可以选择总是完成更新（这将导致最近遇到的指向每个边缘顶点的边会被添加到树中），或者做随机选择。

还有一种策略对于研究图处理算法至关重要，因为它是我们将在第 20 ~ 22 章中讨论的许多经典算法的基础，这种策略就是为边缘集使用一个优先队列（priority-queue）ADT（见

第9章)：我们为边缘集中的每条边指定一个优先级，并在合适的情况下对它们进行更新，而且选择优先级最高的边作为下一条要被添加到树中的边。在第20章，我们将详细地讨论这种方法。与栈和队列相比，优先队列的队列维护操作开销更大，因为它们涉及队列中元素间的隐式比较，但是它们可以支持更广泛的图搜索算法。我们将会看到，对于许多关键的图处理问题，只需要在基于优先队列的广义图中通过巧妙地选择优先级就可得到解决。

所有广义图搜索算法都只检查每条边一次，而且在最坏情况下所需的额外空间与 V 成正比；然而它们在某些性能度量上的确存在差异。例如，对于DFS、BFS和随机搜索，图18-28显示了随着搜索的进行相应边缘集的规模；对于图18-13和图18-24所示的同一个示例，图18-29显示了由随机搜索计算得到的树。随机搜索既没有DFS的长路径，也没有BFS的度很大的结点。这些树的形状和边缘集图依赖于所搜索的特定图的结构，但它们也刻画了不同算法的特征。

通过在搜索中处理一个森林（不必是一棵树），我们还可以对图搜索进一步推广。然而，这里不再做这一层次推广，第20章会讨论几个此类算法。

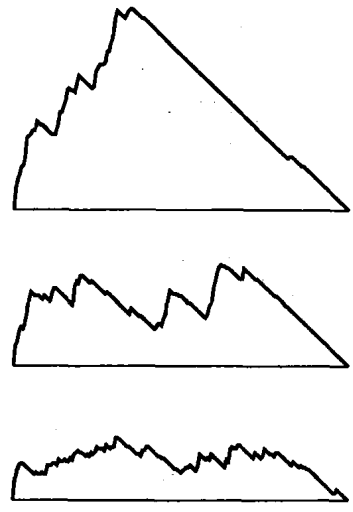


图18-28 DFS、随机化的图搜索和BFS的边缘集大小

对于图18-13、图18-24和图18-29中所描述的搜索过程中边缘集规模的变化，这些图表明了为边缘集选择数据结构对于图搜索具有很大的影响。对上图的DFS中使用栈时，随着每一步发现新结点，在搜索的早期就填满边缘集，然后通过删除所有内容来结束搜索过程。当使用随机队列时（中图），最大队列规模要低得多。当在BFS中使用FIFO队列时（下图），最大队列规模仍然很小，而且我们发现新结点会出现在整个搜索过程中。

练习

○ 18.59 讨论基于以下策略的广义图搜索实现的优点和缺点："将边从边缘集中移到树中。如果它指向的顶点未访问过，则访问该顶点，并将它的所有依附边放入边缘集中。"

18.60 修改邻接表ADT，将边（不仅是目标顶点）保存在链表中，然后基于练习18.59中描述的策略实现图搜索，访问每条边，但要销毁图，为此要利用通过一个链接调整，就可将一个顶点的所有边移到边缘集中的事实。

● 18.61 证明递归DFS（程序18.3和程序18.2）等价于使用栈（程序18.10）的广义图搜索，因为对于任何图这两个程序访问顶点的顺序完全相同，当且仅当程序扫描邻接表的顺序正相反。

18.62 对于下图的随机搜索，给出三种可能不同的遍历顺序

3-7 1-4 7-8 0-5 5-2 3-8 2-9 0-6 4-9 2-6 6-4

18.63 随机搜索访问下图中的顶点，能够按照顶点索引指示的数字顺序访问吗？证明你的结论。

3-7 1-4 7-8 0-5 5-2 3-8 2-9 0-6 4-9 2-6 6-4

18.64 对于图中的边，提供广义队列的一个实现，禁止队列中含有重复顶点的边，使用"忽略新元素"策略。

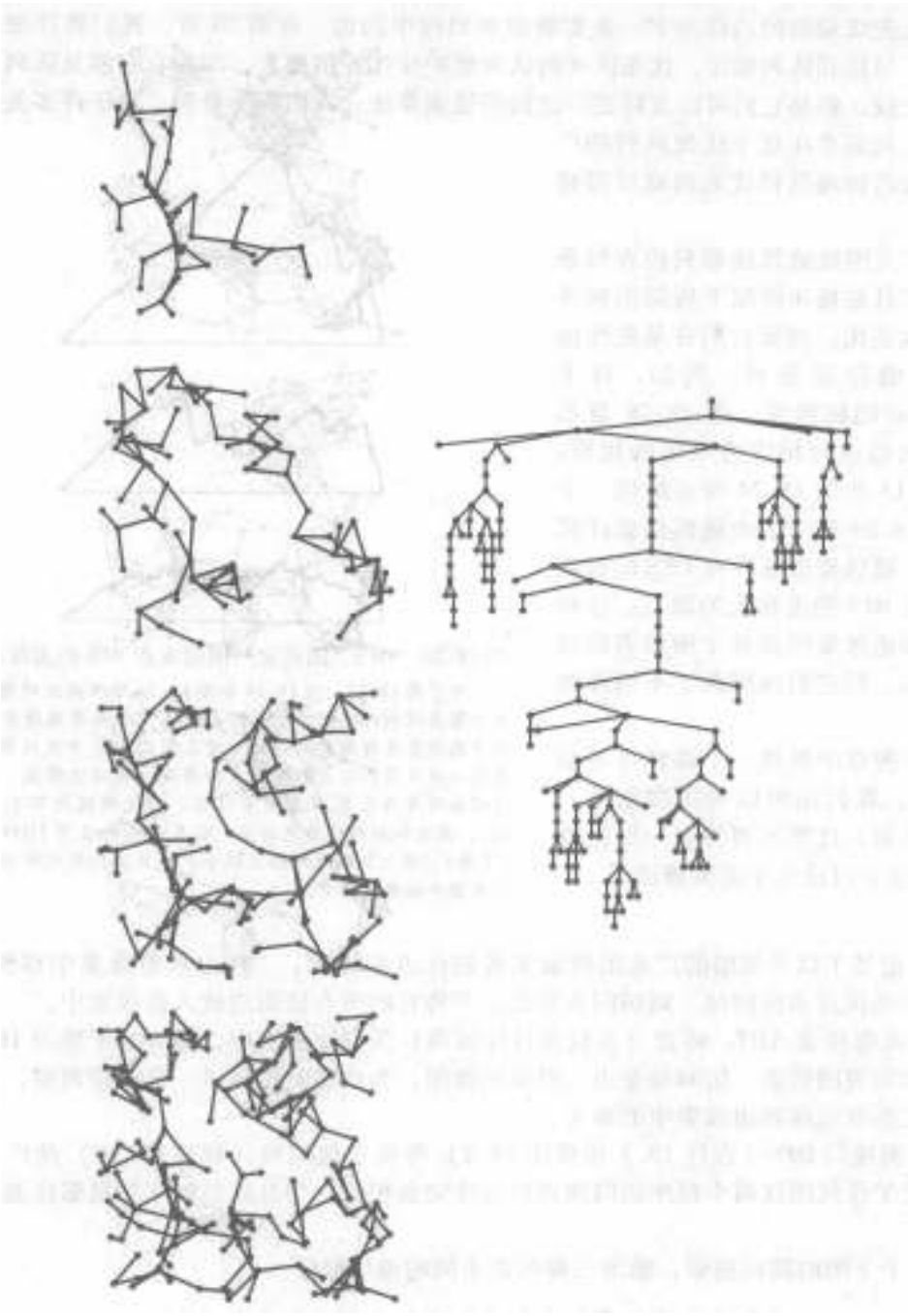


图 18-29　随机化的图搜索

此图描述了随机化图搜索的过程（左图），风格如图 18-3 和图 18-24。搜索树的形状介于 BFS 树和 DFS 树之间。这三个算法的动态行为对于要完成的工作只在数据结构上有所不同，除此之外再无不同。

○ **18.65** 开发一种随机化的图搜索，以等概率选择边缘集中的每条边。提示：见程序18.8。

○ **18.66** 描述一种遍历迷宫的策略，该策略对应使用标准下推栈的广义图搜索（见18.1节）。

○ **18.67** 对广义图搜索进行实验（见程序18.10），打印出树的高度和要看到的每个顶点所处理边的百分比。

● **18.68** 对于不同规模和来自不同图模型（见练习17.63～76）的图，进行实验来确定练习18.67中对于带有随机队列的广义图搜索所描述的各种量的平均值。

● **18.69** 如果图中每个顶点都有一个与之关联的（x，y）坐标，试编写一个客户程序，以动态图像动画方式展示广义图搜索的过程（见练习17.55～17.59）。在随机欧几里得近邻图上测试你的程序，在合理的时间内使用能处理的尽可能多的点。你的程序应该产生类似图18-13、18-24和18-29中快照那样的图形，但是你可以使用任意颜色而不是灰色阴影来表述树、边缘集及未见过的顶点和边。

18.9 图算法分析

我们已经讨论了大量的图处理问题以及解决它们的方法。因此并不像在其他领域中那样总是比较求解同一问题的大量不同算法。然而用实际数据或者模拟数据测试这些算法所得的经验总是有价值的。我们理解这些数据，且它们有着实际应用中我们可能期望的相关特征。

如在第2章简略介绍的那样，我们寻求理想、自然的输入模型，具有以下三个关键性质：

- 它们充分地反映实际，使得我们可以使用它们来预测性能。
- 它们足够简单，经得起数学分析的检验。
- 我们可以编写生成器，提供问题实例用于对算法进行测试。

有了这三部分，就可以进入设计－分析－实现－测试的各个环节，从而得到为解决实际问题的有效算法。

对于诸如排序和查找等领域，我们已经在第3部分和第4部分看到了沿着这种思路所取得的巨大成功。我们可以分析算法，产生随机问题的实例，再改进实现为在大量实际情况中使用提供非常高效的程序。对于我们研究的其他领域，可能出现各种困难。例如，对于很多几何问题的数学分析已经超出了我们的能力所及范围，而且开发输入的一个精确模型对于很多串处理算法极具挑战性（实际上，这样做是计算的一个基本部分）。类似地，图算法将我们带到了另一种境地：对于很多应用问题，关于上段列出的三个性质我们都如履薄冰。

- 数学分析有难度，很多基本的分析问题尚未定论。
- 有大量不同类型的图，我们不能在所有这些图上合理地测试算法。
- 要刻画在实际问题中出现的各种类型的图的特征，从很大程度上将，还是一个知之甚少的问题。

图相当复杂，我们常常不能完全理解在实际中所见的图或是可能生成和分析的人工图的基本性质。

情况也许并不像刚才所描述的那样糟糕，一个主要原因是，我们考虑的很多图算法是最坏情况下的最优算法，因此预测性能也是一个简单的练习。例如，程序18.7在检查每条边和每个顶点一次后找出桥。这个开销与构建图的数据结构的开销一样，这样我们就能很自信地做出预测，无论处理何种类型的图，如果边数加倍，那么运行时间也加倍。

当一个算法的运行时间依赖于输入图的数据结构时，得出预测就要困难得多。然而，如果需要处理大量的大型图，那么如同任何其他问题领域，出于同样的原因，我们也希望得到高效的算法，而且对于理解算法和应用的基本性质这一目标，我们将会矢志不渝，并且对于

实际中可能出现的图，力争找出最适合于这些图的方法。

为了说明一些这样的问题，我们回顾图的连通性问题，这是我们在第 1 章所讨论的一个问题（!）。随机图的连通性数年来一直令数学家着迷不已，而且已经成为大量文献所讨论的主题。这些文献超出了本书的范围，但它为我们验证问题作为某些实验研究的基础提供了背景，有助于我们理解所使用的基本算法以及所考虑的图的类型。

例如，通过向一个初始孤立的顶点集中添加随机边来扩展图（实际上，这是程序 17.7 所完成的过程）是一个得到深入研究的过程，已经作为经典图论的基础。众所周知，随着边数的增长，图将结合成为一个巨大的分量。有关随机图的文献给出了关于此过程特征的大量信息。例如：

性质 18.13　如果 $E>\frac{1}{2}V\ln V+\mu V$（$\mu$ 为正数），V 个顶点和 E 条边的随机图包含单个连通分量，而且孤立顶点的平均数小于 $e^{-2\mu}$，当 V 趋于无穷大时，概率趋近于 1。

证明　这一事实由 Erdös 和 Renyi 在 1960 年的开创性工作确立。其证明超出了本书的范围（见参考文献）。■

这个性质说明，大型的非稀疏随机图很可能是连通的。例如，如果 $V>1\ 000$ 且 $E>10V$，则 $\mu>10-\frac{1}{2}\ln 1\ 000>6.5$，而且不在此大分量中的平均顶点数（几乎肯定）小于 $e^{-13}<.000\ 003$。如果产生 1 百万个有 1 000 个顶点的随机图，其密度大于 10，就有可能会得到一些仅有单个孤立顶点的图，其余图都是连通的。

图 18-30 对随机图与随机近邻图进行了比较，其中只允许索引相差一个小的常数范围以内的顶点之间有边相连。近邻图模型产生的图显然与随机图的特征存在很大的差异。然而，当两个大的分量归并时，最后也会得到一个巨大的分量。

表 18-1　两种随机图模型的连通性

对于由两个不同分布所得的 100 000 个顶点的图。此表显示了它们的连通分量个数和最大连通分量的规模。对于随机图模型，这些实验证实了一个众所周知的事实，即如果平均的顶点度大于一个较小的常量，那么图极有可能主要包含一个巨大的分量。当我们将边的选择限制为只能是将每个顶点连接到其 10 个特定近邻之一的边时，则得到右边两列所示的实验结果。

	随机边		随机 10 个近邻	
E	C	L	C	L
1 000	99 000	5	99 003	3
2 000	98 000	4	98 010	4
5 000	95 000	6	95 075	5
10 000	90 000	8	90 300	7
20 000	80 002	16	81 381	9
50 000	50 003	1 701	57 986	27
100 000	16 236	79 633	2 8721	151
200 000	1 887	98 049	3 818	6 797
500 000	4	99 997	19	99 979
1 000 000	1	100 000	1	100 000

说明：C　连通分量个数
L　最大连通分量规模

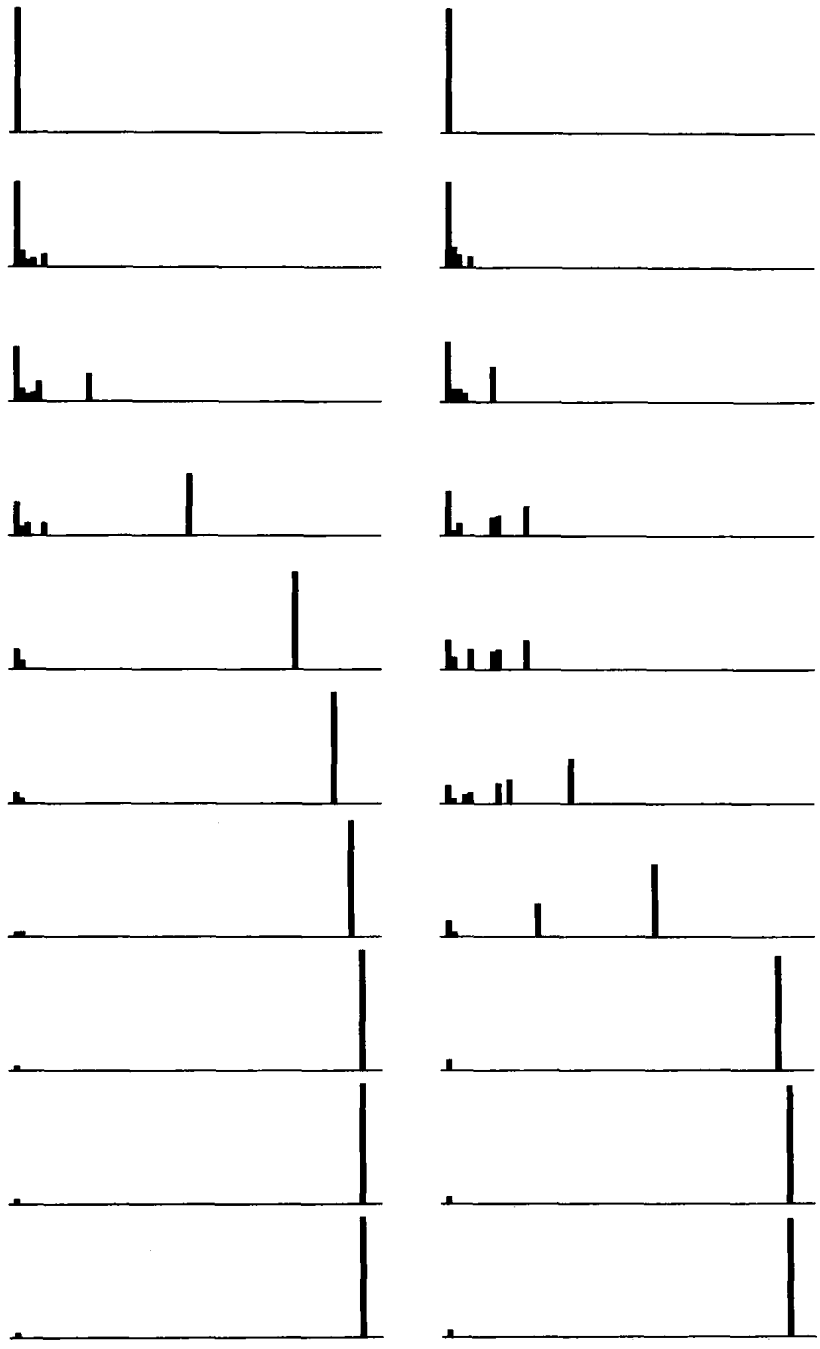


图 18-30 随机图的连通性

此图描述了向初始为空的两类随机图中添加总数为 $2E$ 条边的过程，每次增加均等的 10 条边。每幅图都是规模为从 1 到 $V-1$ 的分量中顶点数的直方图（从左到右）。开始是所有顶点都在规模为 1 的分量中，结束时几乎所有顶点都在一个巨大的分量中。左图对应于一个标准随机图：巨大分量很快形成，其他所有分量很小。右图对应随机近邻图：在一段较长时间内一直保持着各种规模的分量。

表 18-2 显示了随机图和近邻图（包括 V 个顶点和 E 条边）之间的结构差异，其中 V 和 E 均为在实际情况下有意义的范围内。这种结构差异自然可能会在算法的性能上有所反映。

表 18-2 给出了在一个随机图中使用各种算法查找连通分量数所需开销的实验结果。尽管算法不能直接用于特定任务的直接比较，因为它们是设计用来处理不同的任务的，但这些实验验证了我们已得出的一部分一般结论。

表 18-2 图搜索算法的实验研究

对于具有不同顶点数和边数的图，此表显示了确定连通分量数（以及最大连通分量规模）的各种算法的相对时间性能。正如所期望的，对于稀疏图使用邻接－矩阵表示的算法较慢。但对于稠密图这种表示则有竞争力。对于这项特殊的任务，我们在第 1 章讨论过的合并－查找算法是最快的，因为它们构建的数据结构非常适合于求解这个问题，而且不需要其他数据结构来表示图。然而，一旦表示图的数据结构构造出来，DFS 和 BFS 就会更快、更灵活。对于稠密图，增加一个测试，使得当图中包含单个连通分量时算法就停止，这样做可以显著加速 DFS 和合并－查找算法。

			邻接矩阵			邻接表				
E	U	U*	I	D	D*	I	D	D*	B	B*
5 000 个顶点										
500	1	0	255	312	356	1	0	0	0	1
1 000	0	1	255	311	354	1	0	0	0	1
5 000	1	2	258	312	353	2	2	1	2	1
10 000	3	3	258	314	358	5	2	1	2	1
50 000	12	6	270	315	202	25	6	4	5	6
100 000	23	7	286	314	181	52	9	2	10	11
500 000	117	5	478	248	111	267	54	16	56	47
100 000 个顶点										
5 000	5	3				3	8	7	24	24
10 000	4	5				6	7	7	24	24
50 000	18	18				26	12	12	28	28
100 000	34	35				51	28	24	34	34
500 000	133	137				259			88	89

说明：U 带有折半的加权快速合并（程序 1.4）
I 图表示的初始构造
D 递归 DFS（程序 18.1 和程序 18.2）
B BFS（程序 18.9 和程序 18.10）
* 当发现图是完全连通时退出

首先，从此表中可以明显看出，对于大型稀疏图不应使用邻接矩阵表示（对于超大型的图也不能用这种表示），这不仅仅是因为初始化数组的开销太过昂贵，还因为算法将检查数组中的每一个元素，因此其运行时间是与数组的规模（V^2）成正比，而不是与数组中 1 的个数（E）成正比。例如，这个表显示出如果使用一个邻接矩阵，那么处理一个包含 1 000 条边的图所需时间与处理一个包含 100 000 条边的图大致相同。

其次，由表 18-2 还可清楚地看出，为大型稀疏图构建邻接表时，为链表结点分配内存的开销相当大，构建邻接表的开销是对其遍历的开销的 5 倍还多。在典型情况下，我们会在构建图之后完成各种类型的搜索，这个开销是可以接受的。否则，就应像在第 2 章中那样，考虑预先分配数组来降低内存分配的开销。

第三，对于大型稀疏图，DFS 对应的几列中没有给出数字，这一点很重要。这些图将使得递归深度过大，而这（最终）会导致程序崩溃。如果希望对这些图采用 DFS，则需要使用 18.7 节中所讨论的非递归版本。

第四，此表显示出基于合并 - 查找的方法要快于 DFS 或 BFS，主要原因是它不必表示整个图。然而，如果没有这种表示，我们就无法回答诸如“是否存在连接 v 到 w 的一条边?”等简单问题，因此，如果我们希望做更多的工作（想要回答“v 和 w 之间是否存在一条路径”的查询，再混杂着填加边的任务），那么基于合并 - 查找的方法就不适用了。一旦构建了图的内部表示，只为确定图是否连通而实现一个合并 - 查找算法就不值得了，因为 DFS 或 BFS 可以很快地做出回答。

在进行实验测试来得到诸如此类的表时，可能需要对各种特殊情况作进一步的解释。例如，在很多计算机上，缓存体系和内存系统的其他性能可能对大型图的处理性能产生极大的影响。改善关键应用中的性能可能需要详细了解机器的体系结构以及我们考虑的所有因素。

对这些表进行仔细的研究，还将揭示出这些算法的更多性质。我们的目标不是一个详尽的比较，而是要说明尽管在比较图算法时面临一些挑战，但是我们能够而且应该进行实验研究并利用任何可用的分析结果，以认识算法的重要特性，并预测其性能。

练习

○ **18.70** 进行像表 18-2 那样的实验研究，来确定一个图是否是二分的（2-可着色的）。

18.71 进行像表 18-2 那样的实验研究，来确定一个图是否是双连通的。

○ **18.72** 对于由各种图模型所得的各种规模的稀疏图进行实验研究，找出图中第二大连通分量的期望大小。

18.73 编写一个程序产生像图 18-30 中的图表，并用由各种图模型所得的不同规模的图（见练习 17.63 ~ 76）对它进行测试。

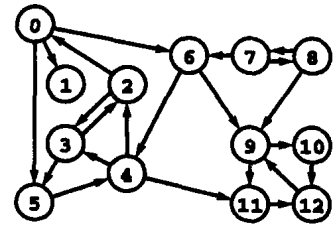
○ **18.74** 修改练习 18.73 中的程序，产生类似的边连通分量规模的直方图。

●● **18.75** 本节的表中的数字均来自于一个样本。我们可能希望准备一个类似的表，对每个元素进行 1000 次实验，给出样本的均值和标准方差。但可能不会包含太多的元素。这种方法是否更好地利用了计算机时间？论证你的答案。

第 19 章　有向图和有向无环图

对于图中的每条边，如果我们强调一个图中每条边的两个顶点所指定顺序的重要性，就得到了一个完全不同的组合对象，称之为有向图（digraph 或 directed graph）。图 19-1 显示了有向图的一个例子。在一个有向图中，记号 *s*-*t* 描述了从 *s* 到 *t* 的一条边，但是没有给出是否从 *t* 到 *s* 存在边的任何信息。在一个有向图中，两个顶点可以通过 4 种不同的方式相关联：两个顶点之间不存在边；存在从 *s* 到 *t* 的一条边 *s*-*t*；存在从 *t* 到 *s* 的一条边 *t*-*s*；或存在两条边 *s*-*t* 和 *t*-*s*，这表明在两个方向都有连接。在很多应用问题中，单向限制是很自然的，在实现中容易实行，而且看上去似乎无关紧要；但它蕴含着额外的组合结构，这对于算法具有深远意义，并使得对有向图的处理与对无向图的处理截然不同。处理有向图类似于在一个城市中到处游览，其中城市中所有街道都是单向的，而且方向不一定是按照某种统一的模式来指定的。在这种情况下，我们可以想象从一点到达另一点可能会是一个挑战。

可以有很多种方式来解释有向图中边的方向。例如，在一个电话 - 呼叫图中，我们可能考虑一条边是有向的，表示从呼叫方到接收方。在一个事务图中，也有类似关系，可以将边解释为代表从一个实体到另一个实体的现金流、货物流或信息流。我们发现目前的因特网也非常适合于这个经典模型，其中顶点表示 Web 页面，边表示页面之间的链接。在 19.4 节中，我们考察其他一些例子，其中很多模型将更为抽象。



4-2	11-12	4-11	5-4
2-3	12-9	4-3	0-5
3-2	9-10	3-5	6-4
0-6	9-11	7-8	6-9
0-1	8-9	8-7	7-6
2-0	10-12		

图 19-1　有向图

有向图定义为结点和边的一个列表（下图）。在指定一条边时，列出结点的顺序就蕴含着边是从第一个结点指向第二个结点的。在画出有向图时，我们使用箭头描绘出有向边（上图）。

一种常见的情况就是使用边方向来反映一种优先关系。例如，可以用向图对生产线建模：顶点对应所要做的工作，如果对应顶点 *s* 的工作必须在对应顶点 *t* 的工作之前完成，则从顶点 *s* 到顶点 *t* 存在一条边。对同种情况建模的另一种方法是使用一个 PERT 图（PERT chart）：边表示工作，顶点蕴含着指定的优先关系（对于每个顶点，所有进入该顶点的工作必须在任何出顶点的工作开始之前完成）。我们如何确定何时执行每个工作，从而保证不违反优先关系？这个问题称为调度问题（scheduling problem）。如果有向图中存在环，则此问题毫无意义。因而在这种情况下，我们只处理有向无环图（directed acyclic graph，DAG）。我们在第 19.5 节和第 19.7 节考虑 DAG 图的基本性质以及这个简单问题的算法，即*拓扑排序*。在实际应用中，调度问题通常涉及对顶点或边上加权，是对各个作业的时间或代价建立模型。我们将在第 21 章和第 22 章讨论此类问题。

可能的有向图的数目极其庞大。在 V^2 条可能的有向边中，每条边（包括自环）可能出现也可能不出现，因此不同的有向图的总数则为 2^{V^2}。如图 19-2 中所描述的，这个数增长很快，即使与不同有向图的数目比较而且当 V 很小也是如此。与无向图一样，彼此同构（这类图可以对顶点重新标号就与另一个图相同）的有向图这一类要少得多，但我们不能利用这种归约，因为我们尚不知道有向图同构的一种有效算法。

当然，任何程序都必定只能处理可能有向图中的一小部分；实际上，这些数是如此大以至于我们可以确信，任何一个给定的程序都不可能对全部有向图加以处理。通常情况下，对

于在实际中可能遇到的有向图，往往很难刻画其特征，因此我们设计算法，使其可以处理作为输入的任何有向图。一方面，这种情况对于我们也不新鲜（例如，对于 1 000 个元素的 1 000! 排列，任何一个排序程序都不会对其中任何一种进行处理）。另一方面，有些情况可能会使我们不安，例如，即使宇宙中所有电子都可以运行超级计算机，而这些超级计算机每秒能够处理 10^{10} 个图，那么在估计的宇宙存在期间之内，这些超级计算机所能处理的图也不到 10 个顶点的有向图的 10^{-100}（见练习 19.9）。

以上有关图枚举的讨论有些离题，但是每当我们在考虑算法分析时就会发现一些问题，而以上讨论则强调了这些问题。以上的这些讨论还表明图枚举与有向图的研究有特定的相关性。设计一个在最坏情况下也能执行很好的算法，这一点非常重要吗？尤其是我们不太可能看到任何特定最坏情况下的有向图。基于平均情况分析来选择一个算法是否有用，这是否是一个数学猜想呢？如果我们的目的是要得到在实际中看到的有向图上的高效执行的实现，我们直接就会面临刻画这些有向图特征的问题。对于在应用中可能遇到的有向图，能够很好地描述这些有向图的数学模型，比描述无向图的模型开发起来更为困难。

V	无向图	有向图
2	8	16
3	64	512
4	1024	65536
5	32768	33554432
6	2097152	68719476736
7	268435456	562949953421312

图 19-2　图枚举

V 个顶点的不同无向图的个数是巨大的，即使在 V 较小时。V 个顶点的不同有向图的个数要大得多。对于无向图，个数由公式 $2^{V(V+1)/2}$ 确定；对于有向图，为 2^{V^2}。

在这一章里，我们重新回顾一些在第 17 章中已经讨论过的有向图意义上的基本图处理问题，而且考察几个有向图上的特定问题。特别是要讨论 DFS 及其几个应用，包括环检测（cycle detection）（确定一个图是否是有向无环图）；拓扑排序（topological sort）（例如，确定刚才描述的 DAG 上调度问题）；以及传递闭包和强连通分量的计算（transitive closure and strong component）（此问题的基础是确定两个给定的顶点之间是否存在一条有向路径）。对于其他的图处理领域，这些算法从简单到复杂都有涵盖；它们都可由有向图的复杂组合结构所获悉，而且使我们对此结构有深入的理解。

练习

- ● 19.1　在线找出一个大型有向图，可以是某个在线系统的事务图，也可以是由 Web 页面上的链接所定义的有向图。
- ● 19.2　在线找出一个大型 DAG 图，可以是一个大型软件系统中函数定义依赖关系所定义的 DAG 图，也可以一个大型文件系统的目录连接所定义的 DAG 图。

19.3　制作一个像图 19-2 那样的表，但不考虑带有子环的图和有向图的数目。

19.4　试问包含 V 个顶点和 E 条边的有向图有多少个？

- ○ 19.5　试问对于包含 V 个顶点和 E 条边的每个无向图，有多少个与之对应的有向图？
- ▷ 19.6　试问需要多少位十进制数来表示出 V 个顶点的有向图的个数。
- ○ 19.7　画出有 3 个顶点的非同构的有向图。
- ●●● 19.8　如果仅当两个图不同构才认为是不同的，那么有 V 个顶点、E 条边的不同有向图有多少个？
- ○ 19.9　试计算一台计算机能够处理的 10 个顶点有向图占全部 10 个顶点有向图的百分比的上界，在此仍然沿用正文中所描述的假设，另外假设宇宙中的电子数不超过 10^{80} 个，宇宙的寿命不超过 10^{20} 年。

19.1　术语和游戏规则

我们对于有向图的定义与第 17 章中有关无向图的定义几乎完全相同（我们所用的一些

算法和程序也同样如此），然而在此这些定义仍然值得重新阐述。这里解释边方向的用词上蕴含的结构化性质将是本章关注的重点。

定义 19.1 有向图（digraph，directed graph）由顶点集和连接有序顶点对的有向边集（不含重复边）组成，我们说一条边从其第一个顶点指向第二个顶点。

与无向图的定义类似，在此定义中不允许重复边，但对于各种应用和实现，保留了在方便时也可允许有重复边这一选择。我们显式地允许在有向图中可以有自环（而且通常采用一个约定：每个顶点都有一个自环），这是因为它们在基本算法中起着重要的作用。

定义 19.2 在一个有向图中的有向路径（directed path）是顶点的一个序列，其中有一条有向图边将序列中的每个顶点与序列中该顶点的后继连接起来。如果存在从 s 到 t 的一条路径，则称顶点 t 从顶点 s 是可达的（reachable）。

我们采用了约定，每个顶点可由其自身可达。通常只要保证自环出现在有向图的表示中，就能实现这个假设。

理解本章的很多算法要求理解有向图的连通性质，以及这些性质对于沿着有向图中的边从一个顶点移到另一个顶点的基本过程的影响。较之于无向图，对于有向图建立这种理解更为复杂。例如，我们看一眼就能说出一个较小的无向图是否是连通的，或是否包含环；这些性质在有向图中并不容易看出，如同在图 19-3 中所描述的典型例子中所表明的那样。

而像这样的例子强调了差异，需要说明的重要一点是，人们认为困难或不困难的事情，程序是否也这样认为呢？例如，编写一个 DFS 函数来找出有向图中的环并不比在在无向图中的处理更困难。更重要的是，有向图中和无向图有着本质上的结构差异。例如，在一个有向图中，t 是由 s 可达的事实并未对由 t 是否可达 s 提供任何信息。这个差异是明显的，但相当关键，我们将会看到。

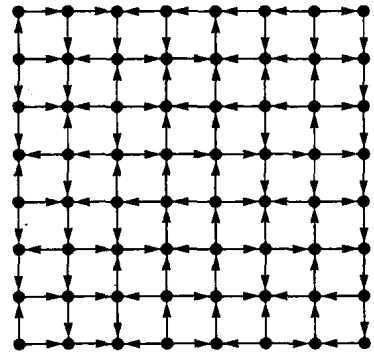


图 19-3 网格有向图

这个小的有向图类似于我们在第 17 章中首次所考虑的大型网格，除了此图在每个网格线上都有一条有向边，方向是随机选择的。即使图中的结点数相对较少，它的连通性质也并不明显。从左上角到右下角是否存在一条有向路径呢？

如在 17.3 节中所提到的，我们用于有向图的表示基本上与用于无向图的表示一样。实际上，它们更直接，因为每条边只表示一次，如图 19-4 中所示。在邻接表表示中，边 s-t 被表示为对应 s 的单链表中的一个包含 t 的表结点。在邻接矩阵表示中，我们需要维护一个完全的 $V \times V$ 矩阵，并将边 s-t 表示为行 s 和列 t 上的一个 1。只有在存在边 t-s 示，我们才在行 t、列 s 放置一个 1。一般而言，有向图的邻接矩阵并不关于对角线对称。在这两种表示中，我们一般包括自环（对于每个顶点 s，表示为 s-s）。

如果在无向图中连接每个顶点的边都有两条边（每个方向一条边），那么这些表示在无向图和带有自环的有向图中的每个顶点上的表示并无差异。因此，如果能对结果进行合理地解释，那么我们就能使用本章为有向图开发的算法来处理无向图。此外，我们使用第 17 章中所考虑的程序作为有向图程序的基础，去掉对于非自环的每条边的第二个表示的引用和隐含假设。例如，为了使程序能够适合有向图来产生、构建和显示图（程序 17.1 ~ 17.9），我们从邻接表版本的函数 GRAPHinsertE（程序 17.6）中删除以下语句

```
G->adj[w] = NEW(v, G->adj[w]);
```

从邻接矩阵版本的函数 GRAPHinsertE 和 GRAPHremoveE（程序 17.3）中删除引用 G -> adj [w][v]，并对于 GRAPHedges 的两个版本作相应的调整。

有向图中顶点的入度（indegree）为通向该顶点的有向边的数目。有向图中顶点的出度（outdegree）为由该顶点发出的有向边的数目。出度为 0 的顶点不能到达任何顶点，该顶点称为汇点（sink）；入度为 0 的顶点称为源点（source），它无法从其他任何顶点达到。允许自环且每个顶点出度为 1 的有向图称为一个映射（map，这是一个将 $0 \sim V-1$ 上的整数集映射到自身的函数）。使用顶点索引的数组（见练习 19.19），我们可以在线性时间内计算出每个顶点的入度和出度，并找出源点和汇点，所用空间与 V 成正比。

有向图的*逆图*（reverse）也是一个有向图，可以通过将所有边上的方向逆转得到。图 19-5 显示了图 19-1 的有向图的逆图及其表示。由于标准表示只说出了边的走向，如果需要了解边来自哪里，可以在有向图的算法中使用逆图。例如，在我们对有向图取逆时，入度和出度将互换角色。

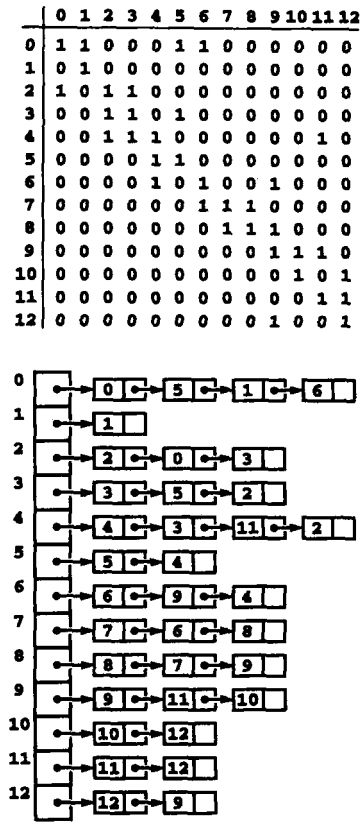


图 19-4　有向图表示

有向图的邻接数组和邻接表表示对于每条边只有一种表示，图 19-1 中所描述的邻接数组表示如上图所示，邻接表表示如下图所示。这两种表示在每个顶点上包含自环，这在图处理算法中是很典型的。

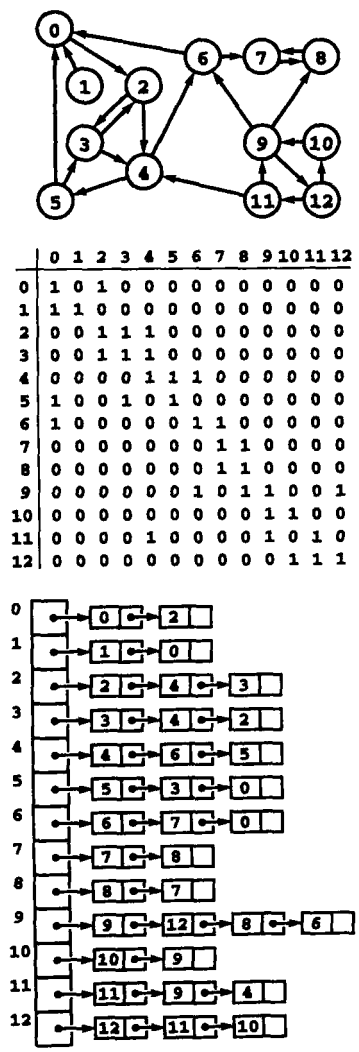


图 19-5　有向图的逆

逆转有向图中的边就对应着对邻接矩阵进行转置，但需要重构邻接表（见图 19-1 和图 19-4）。

对于邻接矩阵表示，可以建立矩阵数组的一个副本，并对它进行转置（交换其行和列）来计算逆图。如果已知不再准备对图进行修改，那么无需做额外计算就可以使用逆图，在希望引用逆图时，只要简单地交换对行和列的引用。也就是说，有向图 G 中的边 s-t 在 G->adj[s][t] 中为 1，因此，如果要计算 G 的逆图 R，那么在 R->adj[t][s] 处为 1；然而我们并不需要这样做，因为如果只是检查 G 的逆图中从 s 到 t 是否存在一条边，只要检查 G->adj[t][s] 即可。这一点看上去是显然的，但经常被忽视。对于邻接表表示，逆图就是一种完全不同的数据结构，我们需要花费与边数成正比的时间来构建这个逆图，如程序 19.1 所示。

程序 19.1　求有向图的逆图（邻接表）

给定邻接表表示的一个有向图，此函数建立有向图的一种新的邻接表表示，其中顶点与原有向图中的顶点相同，边的方向则相反。

```
Graph GRAPHreverse(Graph G)
  { int v; link t;
    Graph R = GRAPHinit(G->V);
    for (v = 0; v < G->V; v++)
      for (t = G->adj[v]; t != NULL; t = t->next)
        GRAPHinsertE(R, EDGE(t->v, v));
    return R;
  }
```

还有一种做法（将在第 22 章中讨论），其中维护每条边的两种表示，就像对于无向图所做的那样（见第 17.3 节），但使用额外一位来表示方向。例如，如果要在邻接表表示中使用这种方法，就需要将边 s-t 表示为 s 的邻接表上对应 t 的结点（并设置方向位，以表示从 s 到 t 是该边的正向遍历），还需要将边 s-t 表示为 t 的邻接表上对应 s 的结点（并设置方向位，以表示从 t 到 s 是该边的后向遍历）。这种表示将支持算法在有向图中边的两个方向进行遍历。一般而言，在这种情况下，包含指针将每条边的两种表示连接起来也较方便。我们将在第 22 章详细讨论这种表示，它在第 22 章中起着重要的作用。

在有向图中，通过类比无向图，我们谈到有向环，这是由某个顶点回到其自身的有向路径，而且是带有环的简单有向路径，其中顶点和边不互相同。注意在有向图中，s-t-s 是一个长度为 2 的环，但在无向图中，环必定包含 3 个不同的顶点。

在有向图的很多应用中，我们不希望看到任何环，我们处理的是另一种类型的组合对象。

定义 19.3　*有向无环图*（directed acyclic graph，DAG）*是没有有向环的有向图。*

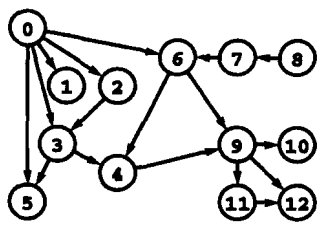
例如，在应用中，我们希望用 DAG 对优先关系建模。DAG 不仅自然地出现在这些应用以及其他重要的应用中，而且如我们将看到的那样，还会出现在一般有向图结构的研究中。图 19-6 给出了一个 DAG 示例。

因此，对于不是 DAG 的有向图，有向环是理解其连通性的关键。如果无向图中每个顶点到其他各个顶点间都存在一条路径，那么这个无向图就是连通的；对于有向图，定义修改如下：

定义 19.4　*如果有向图中的每个顶点均由每个顶点可达，则称有向图是强连通的*（strong connected）。

图 19-1 中的图并不是强连通的，例如，对于从顶点 9～12 到图中的任何其他顶点，不存在有向路径。

如强（strong）这个概念所指示的，此定义蕴含着每对顶点之间存在着一种比可达性更强的关系。在任何有向图中，如果从 s 到 t 存在一条有向路径且从 t 到 s 也存在一条有向路径，则称顶点 s 和 t 是*强连通的*（strongly connected）或是*相互可达的*（mutually reachable）。（我们的约定：每个顶点由其自身可达蕴含着每个顶点到其自身是强连通的。）有向图是强连通的当且仅当每对顶点都是强连通的。强连通有向图所定义的性质就是我们在连通无向图中的性质：如果从 s 到 t 存在一条路径，那么从 t 到 s 也存在一条路径。对于无向图的情况，我们知道这一点必然成立，是因为在某个方向上遍历一条路径，在另一个方向也可以遍历它；而在有向图中，它必定是不同的路径。




```
2-3  11-12  3-5  6-4
0-6  9-12   8-7  6-9
0-1  9-10   5-4  7-6
2-0  9-11   0-5
```

图 19-6　有向无环图（DAG）

此有向图中没有环，这一性质不能由边表明显地表现出来，即使检查其绘图也不能很快得出。

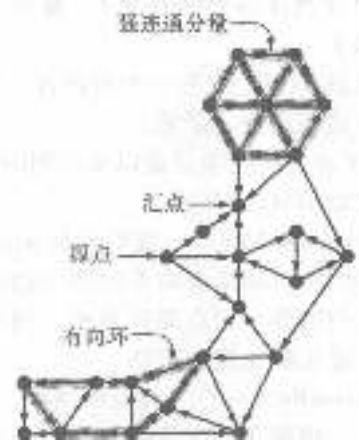


图 19-7　有向图术语

在类似于此的有向图中，源点（没有边进入的顶点）和汇点（没有边出去的顶点）是容易识别出的，但是有向环和强连通分量较难以识别出。此图中的最长有向环是什么？此图中有多少个顶点数大于 1 的强连通分量？

阐述顶点是强连通的另一种方式是：它们能够位于某个有向环路上。我们使用过环路径（cycle path）而非环（cycle）来表明路径不必是简单的。例如，图 19-1 中，5 和 6 是强连通的，因为 6 通过有向路径 5-4-2-0-6 由 5 可达。5 通过有向路径 6-4-3-5 由 6 可达。而且这些路径蕴含着 5 和 6 位于有向环路 5-4-2-0-6-4-3-5 上，但它们并不在任何（简单）有向环上。需要说明，包括不只一个顶点的 DAG 都不是强连通的。

与无向图中的简单连通性一样，这种关系也是可传递的。如果 s 与 t 强连通，而 t 与 u 强连通，那么 s 与 u 也是强连通的。强连通性是一种等价关系，它将顶点划分为等价类，各类中均包含有彼此强连通的顶点。（有关等价关系更详细的讨论见 19.4 节）。同样，强连通性给出了有向图的一个性质，这个性质对于无向图的连通性当然成立。

性质 19.1　*不是强连通的有向图由一组强连通分量*（strongly connected component，简称*强分量*，strongly component）*以及由一个连通分量到另一个连通分量的一组有向边组成，其中强分量是最大强连通子图。*

证明　类似于无向图中的分量，有向图中的强分量为顶点子集的导出子图：每个顶点只在一个强分量中。为了证明这一点，首先注意到每个顶点至少属于一个至少包含顶点自身的强连通分量。其次注意到每个顶点至多属于一个强连通分量：如果某个顶点属于两个不同的连通分量，那么就会存在通过那个顶点的路径，将这两个连通分量中的顶点相互连接起来，而且在两个方向。这与两个连通分量的最大性相矛盾。 ■

例如，由单个有向环组成的有向图只有一个强分量。另一种极端情况，DAG 中的每个顶点是一个强分量，因此 DAG 中的每条边都是由一个分量指向另一个分量。一般而言，有向图中的并非所有边都在强分量中。这种情况与无向图的连通分量的情形有所不同，无向图中的每个顶点和每条边都属于某个连通分量，但它与无向图的边连通分量类似。有向图中的强连通分量是由边连接起来的，从分量中的一个顶点连接到另一个分量中的一个顶点，但不会返回来。

性质 19.2　*给定一个有向图 D，定义另一个有向图 $K(D)$，其中顶点对应于 D 的每个强连通分量，$K(D)$ 中的边对应 D 中将不同强分量中顶点连接起来的边（连接 K 中的顶点，对应着它在 D 中所连接的强分量）。因此，$K(D)$ 是一个 DAG（我们称之为 D 的核心 DAG (Kernal DAG)）。*

证明　如果 $K(D)$ 中有一个有向环，那么在 D 的两个不同强分量中的顶点就会落在一个有向环中，这就产生了矛盾。■

图 19-8 显示了这些强分量以及示例图中的核心 DAG。我们在 19.6 节中将会看到找出强分量和构建核心 DAG 的算法。

连通性（connectivity）　我们保留无向图的连通的这个术语。在有向图中，如果忽略有向图中边的方向，如果这样定义的无向图中的两个顶点是连通的，则称这两个顶点是连通的。但我们通常避免这种用法。

可达性（reachability）　在有向图中，如果从 s 到 t 存在一条有向路径，则称顶点 t 是由顶点 s 可达。对于无向图的情况，要避免术语可达的（reachable），虽然可以认为它等价于“连通”，因为在某些无向图中的一个顶点由另一个顶点可达的概念是很直观的（例如表示迷宫的无向图就是如此）。

强连通性（strong connectivity）　如果有向图中的两个顶点是相互可达的，则称它们是强连通的；在无向图中，两个连通顶点蕴含着它们相互之间存在着路径。有向图中的强连通性在某些方面类似于无向图中的边连通性。

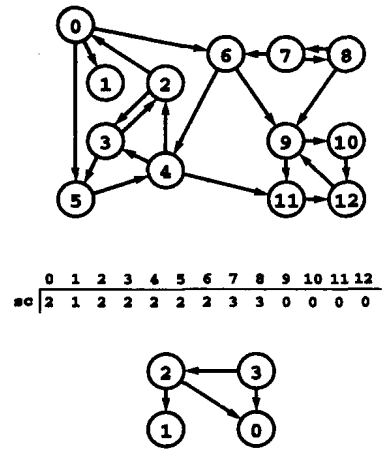


图 19-8　强分量和核心 DAG

此有向图（上图）包含 4 个强分量，可由顶点索引数组 sc 来标识（整数标记是任意指定的）（中图）。分量 0 包含顶点 9、10、11 和 12；分量 1 包含单个顶点 1；分量 2 包含顶点 0、2、3、4、5 和 6；分量 3 包含顶点 7 和 8。如果画出由不同分量之间边所定义的图，就得到了一个 DAG（下图）。

我们希望能够支持有向图 ADT 操作，取两个顶点 s 和 t 作为参数，可以检测：

- t 是否由 s 可达
- s 和 t 是否为强连通的（相互可达）

如果希望扩展这些操作，有那些资源需求呢？如在 17.5 节中那样，DFS 给出了无向图的连通性的一种简单解决方案，所需时间与 V 成正比，但如果愿意花费与 $V+E$ 成正比的预处理时间和与 V 成正比的预处理空间，那么在常量时间内就能回答连通性查询问题。在本章后面部分，我们将考察强连通性算法，它们也有同样的性能特征。

然而，我们的目的主要是解决有向图中的可达性查询问题，这是比处理连通性或强连通性查询更困难的问题。在这一章里，我们考察其预处理时间与 VE 成正比和空间与 V^2 成正比的经典算法，并为某些有向图开发在常量时间完成可达性查询的实现，它们只需要线性空间和线性预处理时间。此外，还会研究对于所有有向图要到这种最优性能的难度。

练习

▷ **19.10**　对于如下有向图，给出程序 17.6（像正文中所述对它进行修改，使其能够处理有向图）所构建的邻接表结构。

3-7　1-4　7-8　0-5　5-2　3-8　2-9　0-6　4-9　2-6　6-4

▷ **19.11**　基于程序 17.6（像正文中所述对它进行修改），实现稀疏有向图的 ADT。包含一个基于程序 17.7 的随机有向图生成器。对于精心选择的一组 V 和 E 值，编写一个产生随机有向图的客户端程序，使得你能对于由此模型所得的图，使用该程序进行有意义的实验测试。

▷ **19.12**　基于程序 17.3（像正文中所述对它进行修改），实现稠密有向图的 ADT。包含一个基于程序 17.8 的随机有向图生成器。对于精心选择的一组 V 和 E 值，编写一个产生随机图的客户端程序，使得你能对于由此模型所得的图，使用该程序进行有意义的实验测试。

○ **19.13**　编写一个产生随机有向图的程序，将排列为$\sqrt{V} \times \sqrt{V}$网格的顶点与其近邻连接起来，其中边的方向随机选择（见图 19-3）。

○ **19.14**　扩展你在练习 19.13 中的程序，增加 R 条额外随机边（所有可能边等概率出现）。对于较大的 R，收缩网格使得边的总数大约为 V。像练习 19.11 中所述测试你的程序。

○ **19.15**　修改练习 19.14 中的程序，使得从顶点 s 到顶点 t 的额外边以与它们之间的欧式距离成反比的概率出现。

○ **19.16**　编写一个产生单位区间的 V 个随机区间的程序，区间长度为 d，然后构建一个有向图，从区间 s 到区间 t 有一条边，当且仅当 s 的至少一个端点落入 t 内（见练习 17.73）。确定如何设置 t，使得边的期望数为 E。像练习 19.11 中所述测试你的程序（对于低密度的图）和像练习 19.12 中所述测试你的程序（对于高密度的图）。

● **19.17**　编写一个程序，从练习 19.1 中所找到的实际图中选择 V 个顶点和 E 条边。像练习 19.11 中所述测试你的程序（对于低密度的图）和像练习 19.12 中所述测试你的程序（对于高密度的图）。

● **19.18**　编写一个程序，以相同的概率产生 V 个顶点和 E 条边的每个可能的有向图（见练习 17.69）。像练习 19.11 中所述测试你的程序（对于低密度的图）和像练习 19.12 中所述测试你的程序（对于高密度的图）。

▷ **19.19**　增加一个有向图 ADT 函数，返回有向图中的源点数和汇点数。修改邻接表 ADT 实现（见练习 19.11），使得可以在常量时间内实现该函数。

○ **19.20**　使用练习 19.19 中的程序来找出不同类型有向图中源点和汇点的平均数（见练习 19.11 ~ 18）。

▷ **19.21**　试给出使用程序 19.10 来找出以下有向图的逆图时所产生的邻接表结构：

3-7　1-4　7-8　0-5　5-2　3-8　2-9　0-6　4-9　2-6　6-4

○ **19.22**　描述映射的逆图的特征。

19.23　设计一个有向图 ADT，可以显式地为客户端提供有关功能，使之能够同时引用一个有向图及其逆图，并提供一种使用邻接矩阵表示的实现。

19.24　给出练习 19.23 中 ADT 的一种实现，使其可以维护有向图及其逆图的邻接表表示。

▷ **19.25**　描述一组有 V 个顶点的强连通有向图，其中不含长度大于 2（简单）有向环。

○ **19.26**　给出下面有向图的强分量和核心 DAG。

3-7 1-4 7-8 0-5 5-2 3-8 2-9 0-6 4-9 2-6 6-4

- **19.27** 给出图 19-3 中所示的网格有向图的核心 DAG。

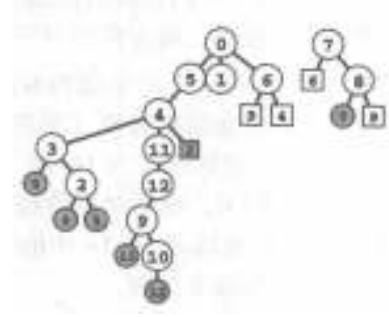
19.28 含有 V 个顶点且所有顶点出度均为 k 的有向图有多少？

- **19.29** 对于一个随机有向图，其不同邻接表表示的期望数是多少？提示：将可能表示的总数除以有向图的总数。

19.2 有向图中的 DFS 剖析

我们可以使用第 18 章中的无向图的 DFS 来访问有向图中的每个顶点和每条边。递归算法之后的基本原理依然成立：访问从某个给定顶点可达的每个顶点，对已经访问过的顶点作出标记，然后（递归地）访问由其邻接表上每个顶点可达的所有顶点。

在一个无向图中，每条边有两种表示，但是在 DFS 中遇到的第二种表示总是通向一个标记过的顶点，且被忽略（见 18.2 节）。在一个有向图中，每条边只有一种表示，因此我们可能会以为 DFS 算法更加直接。然而，有向图自身是比无向图更加复杂的组合对象，因而这个希望并不成立。例如，我们所使用的理解算法执行过程的搜索树对于有向图而言比无向图有更复杂的结构。这种复杂性使得有向图的处理算法更难以设计。例如，我们将会看到，在有向图中进行有向路径的推理要比在无向图中进行路径的推理更困难。



	0	1	2	3	4	5	6	7	8	9	10	11	12
pre	0	9	4	3	2	1	10	11	12	7	8	5	6
post	10	8	0	1	6	7	9	12	11	3	2	5	4

图 19-9 有向图的 DFS 森林

此森林描述了图 19-1 中示例图的一个标准邻接表 DFS。外部结点以相同标记表示了以前访问过的内部结点；另外，此森林是有向图的一个表示，所有边的方向均向下。其中存在 4 种类型的边：树边，指向内部结点；回边，指向表示祖先结点（有阴影的圆结点）的外部结点；下边，指向表示子孙结点（有阴影的方结点）的外部结点；交叉边，指向外部结点，这些外部结点既不表示祖先结点，也不表示子孙结点（白色方结点）。我们可以通过比较结点的源点和目的点的前序编号和后序编号来确定指向已访问结点的边的类型：

pre	post	示例	类型
<	>	4-2	下边
>	<	2-0	回边
>	>	7-6	交叉边

例如，7-6 是一条交叉边，因为 7 的前序编号和后序编号都大于 6 的前序编号和后序编号。

如第 18 章一样，我们使用术语*标准邻接表* DFS（standard adjacency-list DFS）来表示向使用邻接表表示实现的有向图 ADT 中插入一组边的过程（程序 17.6，加以修改使其只插入一条有向边），然后使用程序 18.3 和 18.2 来执行 DFS，同时使用术语*标准邻接矩阵* DFS（standard adjacency-list DFS）表示向使用邻接矩阵表示实现的有向图 ADT 中插入一组边的过程（程序 17.3，加以修改使其只插入一条有向边），然后，使用程序 18.3 和 18.1 来执行 DFS。

例如，图 19-9 显示的递归调用树，它描述了标准邻接矩阵 DFS 在图 19-1 中示例有向图上操作的过程。与无向图一样，这样的树的内部结点对应于对每个顶点的递归 DFS 函数调用。指向外部结点的链接则对应于指向已访问过的顶点的边。对结点和链接的分类为我们提供了搜索（以及有向图）的信息，但是对有向图的分类与对无向图的分类完全不同。

在无向图中，对于 DFS 中的每个链接，按照它是否对应图中一条指向递归调用的边，以及它对应 DFS 所遇到的边的第一种表示还是第二种表示，来指定它所属的 4 种类型之一。在有向图中，树链接和图中的边是一一对应的，而且它们均可分成 4 个不同的类：

- 表示递归调用（树边）
- 从一个顶点到它的 DFS 树中的一个祖先结点（回边）

- 从一个顶点到它的 DFS 树中的一个子孙结点（下边）
- 从一个顶点到另一个顶点，此顶点既非它在 DFS 树中的祖先也非子孙结点（交叉边）

树边是指向未访问顶点的边，对应于 DFS 中的递归调用。回边、交叉边和下边指向访问过的顶点。为了识别出一条给定边的类型，我们使用前序和后序编号。

程序 19.2　有向图的 DFS

此邻接表表示的有向图的 DFS 函数显示了图中的每条边在 DFS 中所起的作用。在此假设对程序 18.3 进行了扩展，采用与 pre 和 cnt 同样的方法声明和初始化数组 post 和计数器 entP。对于示例输出和实现 show 的讨论见图 19-10。

```
void dfsR(Graph G, Edge e)
  { link t; int i, v, w = e.w; Edge x;
    show("tree", e);
    pre[w] = cnt++;
    for (t = G->adj[w]; t != NULL; t = t->next)
      if (pre[t->v] == -1)
        dfsR(G, EDGE(w, t->v));
      else
        { v = t->v; x = EDGE(w, v);
          if (post[v] == -1) show("back", x);
          else if (pre[v] > pre[w]) show("down", x);
          else show("cross", x);
        }
    post[w] = cntP++;
  }
```

性质 19.3　在对应有向图的一棵 DFS 森林中，对于一条指向已访问结点的边，如果它指向一个具有较大后序编号的结点，那么这条边是一个回边；否则，如果它指向一个具有较小前序编号的结点，那么这条边是一个交叉边，如果它指向一个具有较大前序编号的结点，那么这条边是一个下边。

证明　由定义可得这些事实。DFS 树中的一个结点的祖先有着较小的前序编号和较大的后序编号；它的子孙具有较大的前序编号和较小的后序编号。相对于其他 DFS 树中以前所访问过的结点，这两个编号均较小，而相对于其他 DFS 树中待访问的结点，这两个编号均较大，但我们不需要编写代码对这些情况进行测试。 ■

程序 19.2 是一个 DFS 有向图搜索函数原型，它能识别出有向图中每条边的类型。图 19-10 展示了它在图 19-1 的示例图上的操作过程。在搜索过程中，测试一条边是否指向一个具有较大后序编号的结点，就等价于测试一个后序编号是否已指定。任何已经指定了前序编号而尚未指定后序编号的结点都是 DFS 树中的一个祖先，它的后序编号将会大于当前结点的后序编号。

与第 17 章中的无向图类似，边类型为搜索的动态性质，而不是图的动态性质。实际上，同一图的不同 DFS 森林在特征上差异很大，如图 19-11 所示。例如，甚至 DFS 森林中树的个数要依赖于起始顶点。

尽管存在这些差异，对于一个经典的有向图处理算法，在 DFS 中遇到不同类型的边时，通过采取适当的做法，它们就能够确定有向图的性质。例如，考虑以下基本问题：

有向环检测　一个给定的有向图中是否存在有向环？（一个有向图是否为一个 DAG？）在无向图中，指向已访问顶点的边就表明了图中存在一个环；在有向图中，我们必须将注意

力放到回边上。

性质 19.4 一个有向图是一个 DAG，当且仅当使用 DFS 检测每条边时，没有遇到回边。

证明 任何回边都属于某个有向环，此有向环包含该边以及连接相应两个结点的树路径，因此对一个 DAG 使用 DFS 不会发现任何回边。证明另一方面，我们表明如果此有向图中存在环，那么 DFS 遇到一个回边。假设 v 是 DFS 所访问的此环上的第一个顶点。在此环的所有顶点中，该顶点的前序编号最小，指向该顶点的边将会是一条回边：会在对 v 的递归调用中遇到这条边（必定会遇到这条边的证明见性质 19.5）；而且它从环上的某个顶点指向 v，这是一个具有较小前序编号的结点（见性质 19.3）。 ■

通过完成 DFS，并去掉图中对应于 DFS 中回边的所有边，就可以将有向图转换为 DAG。例如，由图 19-9 可知，去掉边 2-0、3-5、2-3、9-11、10-12、4-2 和 8-7 使图 19-1 成为一个 DAG。我们用这种方法得到的特定 DAG 依赖于图的表示以及 DFS 的动态性的相关含义（见练习 19.38）。这种方法是一种随机产生任意 DAG 的有用方法（见练习 19.79），可用于 DAG 处理算法的测试中。

有向环检测是一个简单问题，但相对我们在第 18 章对于无向图所讨论的解决方法，上述解决方法有所不同，这就深刻地表明将这两种类型的图考虑为不同组合对象的必要性，即使它们的表示如此类似，甚至对于某些应用，同样的程序对这两种类型的图都能加以处理，仍应将它们加以区别。由我们的定义，似乎要使用无向图中环检测所用的同一方法来解决这个问题（寻找回边），但在无向图中所用的实现对有向图并不适用。例如，在 18.5 节中我们仔细地区别了父链接和回链接，因为父链接的存在性并不能表明有环（无向图中的环必定至少包含 3 个顶点）。但在有向图中忽略回到某个结点的父结点的链接会不正确；对于有向图我们的确认为双向连通的顶点对就是一个环。理论上说，我们在无向图中所定义的回边与这里有向图中所定义的回边一样，但需要明确区分出两个顶点的情况。更重要的是，可以在与 V 成正比的时间内检测出无向图中的环（见 18.5 节），但我们可能需要与 E 成正比的时间来找出一个有向图中的环（见练习 19.33）。

DFS 的根本目的是要提供一种系统的方法来访问图中的所有顶点和所有边。因而它给出了求解有向图可达性问题的一种基本方法，虽然这种情况要比无向图的情况要复杂得多。

```
0-0 tree
  0-5 tree
    5-4 tree
      4-3 tree
        3-5 back
        3-2 tree
          2-0 back
          2-3 back
      4-11 tree
        11-12 tree
          12-9 tree
            9-11 back
            9-10 tree
              10-12 back
    4-2 down
  0-1 tree
  0-6 tree
    6-9 cross
    6-4 cross
7-7 tree
  7-6 cross
  7-8 tree
    8-7 back
    8-9 cross
```

图 19-10 有向图的 DFS 轨迹

图 19-1 中的示例图的 DFS 轨迹完全对应图 19-9 中所描述的 DFS 树一个前序遍历。对于图 17-7 和其他类似轨迹，我们可以修改程序 19.2，添加一个全局变量 depth 来保存递归的深度和 show 的实现来打印出 depth 个空格，后面跟着一个有相应参数的合适 printf，来产生精确的输出。

单源点可达性（single-source reachability） 在一个给定的有向图中，从给定的起始顶点 s 可达哪些顶点呢？存在多少个这样的顶点？

性质 19.5 使用一个从 s 开始的递归 DFS，可以解决从顶点 s 开始的单源点可达性问题，所用时间与可达顶点所导出的子图中边的个数成正比。

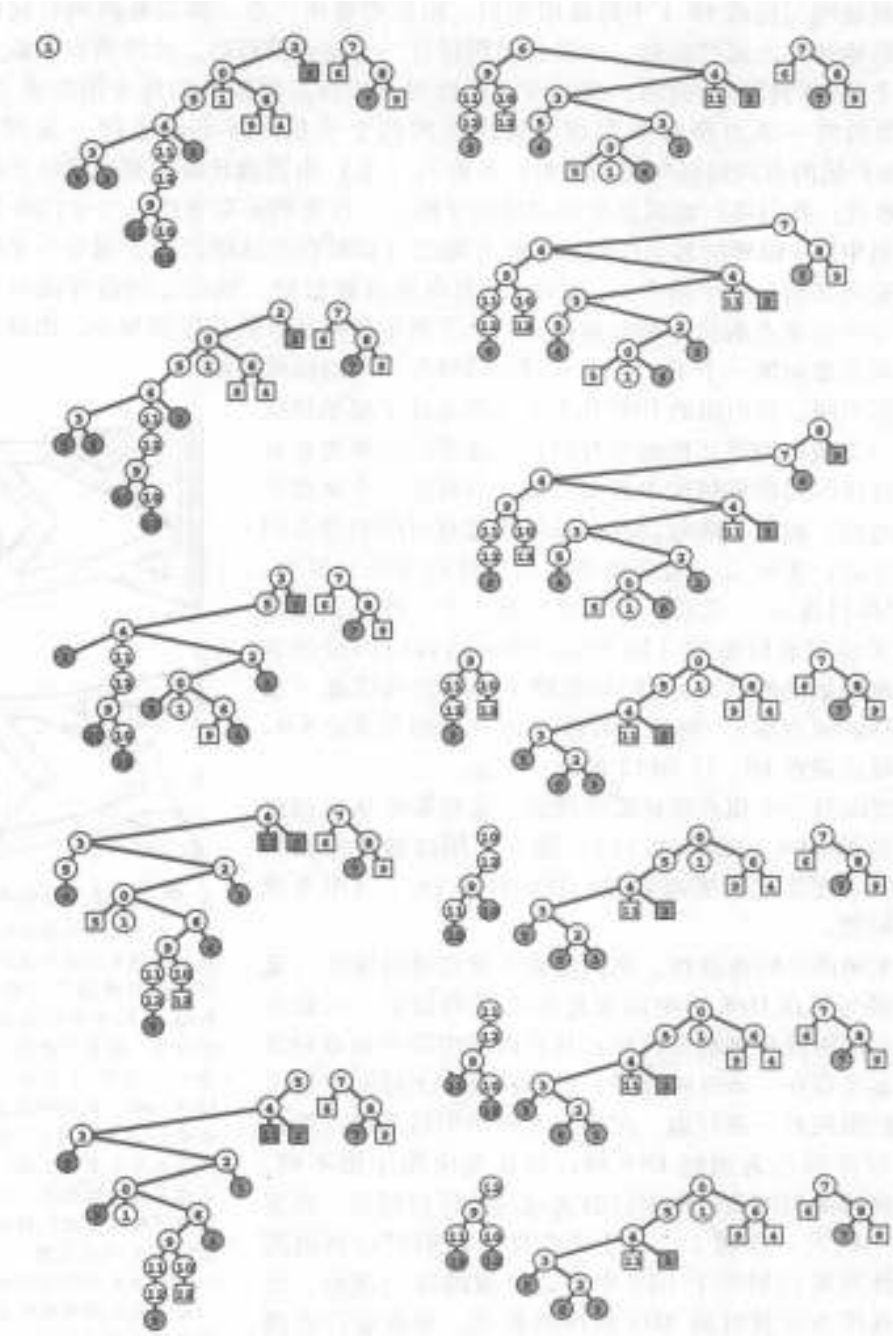


图 19-11　有向图的 DFS 森林

这些森林描述了图 19-9 中同一个图的深度优先搜索，对于每个 s，图搜索函数将按照 $s, s+1, \ldots, V-1, 0, 1, \ldots, s-1$ 的顺序检查顶点（并对未访问顶点调用递归函数）。此森林结构由搜索的动态性和图的结构所确定。每个结点在各个森林中都有相同的子结点（在其邻接表中按照顺序排列的各个结点）。每个森林中的最左树包括了由其根可达的所有结点，但是对于其他结点的可达性推导则很复杂，因为存在着回边、交叉边和下边。甚至森林中树的个数也要取决于开始结点，因此没有必要在森林中的树和强分量之间建立一个直接的对应，就像我们在无向图中对于分量所采用的做法。例如，可以看到，只有从 8 开始 DFS 时才能得到由 8 可达的所有顶点。

证明　此证明与性质 18.1 中的证明类似。但值得重申一点，即需要强调有向图的可达性与无向图的连通性之间的区别。如果有向图仅有一个顶点没有边，此性质自然成立。对于包含不止一个顶点的任何有向图，我们假设此性质对于顶点数较少的所有图均成立。现在，从 s 出发所取的第一条边将有向图划分为顶点的两个子集所导出的子图（见图 19-12）：（ⅰ）由该边开始的有向路径所达的顶点；不含 s；（ⅱ）由该边开始的有向路径不能达到且不回到 s 的顶点；我们将归纳假设应用到这些子图上，注意到不存在第一个子图中某顶点指向第二个子图中除 s 以外的其他任何顶点的有向边（如果存在这样的边，则发生矛盾，因为其目的顶点应当在第一个子图中），指向 s 的有向边将被忽略，因为它的前序编号最小，且第一个子图中所有顶点的前序编号都比第二个子图中任何顶点的前序编号小，因此从第二个子图中某个顶点指向第一个子图中某个顶点的所有有向边将被忽略。■

与无向图不同，有向图的 DFS 并不能给出从除了起始顶点之外的任何一个顶点的可达性的所有信息，这是因为树边是有方向的，而且还因为搜索结构中有交叉边。当离开一个顶点下行到一条树边时，我们不能假设存在某种通过有向图的边返回到该顶点的方法；实际上，通常也不存在这样的方法。例如，在图 19-9 中取树边 4-11 之后，就无法回到 4 了。此外，在我们忽略掉交叉边和前向边时（因为它们指向已经访问过的顶点，并且不再是活动的结点），同时忽略了所蕴含的信息（由目的顶点可达的顶点集）。例如，沿着图 19-9 中的交叉边 6-9，是找到由 6 可达顶点 10、11 和 12 的唯一方法。

为了确定由另一个顶点可达哪些顶点，显然需要从该顶点再开始一个新的 DFS（见图 19-11）。能否利用以前搜索得到的信息使以后的搜索过程更高效呢？我们将在 19.7 节中考虑这种可达性问题。

要确定无向图中的连通性，我们依赖各顶点通过树中（至少）一条路径与其在 DFS 树中的祖先相连接的信息。与此不同，有向图中的树路径则是反向的：从有向图中某个顶点到其某个祖先，如果存在一条有向路径，仅当从某个子孙到该顶点或一个更远的祖先有一条回边。此外，无向图中每个顶点的连通性限制为以该顶点为根的 DFS 树；而在有向图中则不同，交叉边可将我们带到搜索结构中以前访问过的任何部分，甚至在 DFS 森林中的另一棵树上。对于无向图，我们可以利用连通性的这些性质来识别单个 DFS 中连通分量的各个顶点，然后利用此信息作为常量时间 ADT 操作的基础，来确定任意两个顶点是否是连通的。而对于有向图，在本章我们可以看到，这个目标是难以实现的。

在本章和前一章中，我们已经多次强调了选择未访问顶点的不同方式将导致 DFS 不同的搜索动态性。对于有向图，DFS 树的结构复杂性带来了搜索动态性的差异，较之于无向图中所见的差异，这一点更为明显。例如，图 19-11 说明，对于有向图，即使只是在顶层的搜索函数中简单改变顶点检查的顺序，

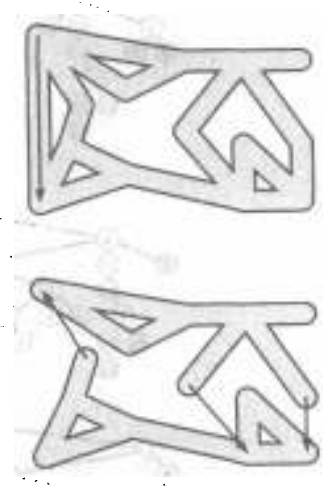


图 19-12　分解有向图

为了用归纳法证明 DFS 可使我们访问有向图中某个给定顶点可达的所有顶点，我们使用与 Trémaux 探索中的证明基本相同的方法。这里关键的一步是描绘为一个迷宫（上图），为与图 18-4 比较，我们将图分为两个较小的部分（下图），它们是由两个顶点集导出的子图：一个集合是由沿着起始顶点（且不会再次访问该起始顶点）的第一条边可达的那些顶点组成（下图），另一个集合是由经过起始顶点的第一条边且返回该起始顶点才能达到的那些顶点组成（上图）。从第一个集合中的某个顶点指向起始顶点的任何边在搜索第一个集合的过程中都被忽略，这是由于起始顶点上的标记。从第二个集合中的某个顶点指向第一个集合中的某个顶点的任何边也被忽略，这是由于第一个集合中的所有顶点在搜索第二个集合之前都已做了标记。

也会带来显著的差异。图中只显示了这些可能性中很小的部分，原则上，存在 $V!$ 种检查顶点的不同顺序，这都可能得到不同的结果。在 19.7 节中，我们将考察一个重要的算法，其中特别利用了这种灵活性，在顶层（DFS 的树根）按照某种特殊顺序处理未访问的顶点，直接得出强分量。

练习

▷ **19.30**　向程序 19.2 中添加代码，打印出像图 19-10 中描述的那种缩进形式的 DFS 轨迹。

19.31　画出由以下有向图的标准邻接表 DFS 所得的 DFS 森林

```
3-7  1-4  7-8  0-5  5-2  3-8  2-9  0-6  4-9  2-6  6-4
```

19.32　画出由以下有向图的标准邻接矩阵 DFS 所得的 DFS 森林

```
3-7  1-4  7-8  0-5  5-2  3-8  2-9  0-6  4-9  2-6  6-4
```

○ **19.33**　描述一组 V 个顶点、E 条边的有向图，其标准邻接表 DFS 进行环检测所需时间与 E 成正比。

▷ **19.34**　在有向图上执行 DFS 的过程中，说明不存在将一个结点与另一个前序编号和后序编号都更小的结点连接起来的边。

○ **19.35**　对于如下的有向图，显示它的所有 DFS 森林，并列表给出每个森林的树边、回边、交叉边和下边的个数。

```
0-1  0-2  0-3  1-3  2-3
```

19.36　如果分别用 t、b、c 和 d 定义树边、回边、交叉边和下边的个数，那么对于 V 个顶点、E 条边的有向图中的任一 DFS，都有 $t+b+c+d=E$ 且 $t<V$。你还能找出这些变量之间的其他关系吗？其中哪些值只依赖于图的性质？哪些值依赖于 DFS 的动态性质？

▷ **19.37**　证明有向图中的每个源点必定是其森林（对应于那个有向图的任一 DFS）中某棵树的树根。

○ **19.38**　构造一个连通的 DAG，它是由删除图 19-1 中的 5 条边而得的一个子图。

19.39　定义一个有向图 ADT 函数，为客户端提供检查一个有向图的确是 DAG 图的能力，并提供邻接矩阵表示基于 DFS 的实现。

19.40　使用练习 19.39 的解决方案，对于各种类型的有向图（见练习 19.11 ~ 18），（实验性地）估计一个 V 个顶点、E 条边的随机有向图是一个 DAG 的概率。

19.41　进行实验性的研究，对于各种类型的有向图（见练习 19.11 ~ 18），确定运行 DFS 时树边、回边、交叉边和下边的相对百分比。

19.42　描述如何构造 V 个顶点的一系列有向边，其中不含交叉边或下边，对于标准邻接表 DFS，且回边个数与 V^2 成正比。

○ **19.43**　描述如何构造 V 个顶点的一系列有向边，其中不含回边或下边，对于标准邻接表 DFS，且回边个数与 V^2 成正比。

19.44　描述如何构造 V 个顶点的一系列有向边，其中不含交叉边或向边，对于标准邻接表 DFS，且回边个数与 V^2 成正比。

○ **19.45**　对于一个其所有通道都是单向的迷宫，给出对应于 Trémaux 遍历的规则。

● **19.46**　扩展练习 17.55 ~ 60 中的解决方案，使得在边上包含箭头（见本章示例中的图）。

19.3　可达性和传递闭包

要开发有向图中可达性问题的一个有效解决方案，我们从以下基本定义开始。

定义 19.5　有向图的传递闭包（transitive closure）也是一个有向图，其中包括同样的顶点，但在此传递闭包中有一条从 s 到 t 的边，当且仅当在给定的有向图中存在从 s 到 t 的一条有向路径。

换句话说，对于有向图中从某个顶点可达的所有顶点，在传递闭包中从该顶点到每个可达顶点都有一条边。显然，传递闭包包含了求解可达性问题的所有必要的信息。图 19-13 显示了一个简单的例子。

基于有向图的邻接矩阵表示，并根据以下的计算问题来理解传递闭包是一种很好的方法。

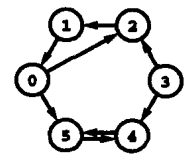
布尔矩阵相乘（Boolean matrix multiplication）　布尔矩阵（Boolean matrix）是一个元素为二进制值 0 或 1 的矩阵，给定两个布尔矩阵 A 和 B，分别使用逻辑与（and）和或（or）操作，而不是算术操作 * 和 +，来计算一个布尔乘积矩阵 C。

计算两个 $V \times V$ 矩阵的算法，教科书中的算法是：对于每个 s 和 t，第一个矩阵中 s 行与第二个矩阵中的 t 列的点积，如下：

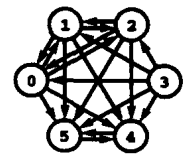
```
for (s = 0; s < V; s++)
  for (t = 0; t < V; t++)
    for (i = 0, C[s][t] = 0; i < V; i++)
      C[s][t] += A[s][i]*B[i][t];
```

采用矩阵表示法，可以将这个操作简记为 $C = A * B$。对于含有任何由 0、+ 和 * 所定义的元素类型均可使用这个操作。特别是，如果我们将 a + b 解释为逻辑或（or），a * b 解释为逻辑与（and）操作，那么就得到布尔矩阵相乘。在 C 中，我们可以使用以下版本：

```
for (s = 0; s < V; s++)
  for (t = 0; t < V; t++)
    for (i = 0, C[s][t] = 0; i < V; i++)
      if (A[s][i] && B[i][t]) C[s][t] = 1;
```



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



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

图 19-13　传递闭包

此图（上图）只有 8 条有向边，但它的传递闭包（下图）显示了存在一些有向路径将 30 对顶点中的 19 对顶点连接起来。有向图的结构性质反映在传递闭包中。例如，该传递闭包的邻接矩阵的第 0、1 和 2 行是相同的（第 0、1 和 2 列也是如此），因为这些顶点位于有向图的有向环上。

为了计算乘积中的 C[s][t]，我们先将它初始化为 0，然后如果找到某个值 i，使得 A[s][i] 和 B[i][t] 均为 1，则将其设置为 1。执行这个计算等价于以下过程：设置 C[s][t] 为 1，当且仅当 A 中 s 行与 B 中 t 列的位逻辑与的结果为非零值。

现在假设 A 是有向图 A 的邻接矩阵，我们使用前面的代码来计算 $C = A * A \equiv A^2$（只要在代码中将指向 B 的引用改为指向 A）。根据对邻接矩阵中元素的解释来阅读代码，立即可知它所计算的是什么：对于每对顶点 s 和 t，在 C 中放置一条边，当且仅当存在某个顶点 i，使得 A 中存在从 s 到 i 的路径和从 i 到 t 的路径。换句话说，A^2 中的有向边恰好对应 A 中长度为 2 的有向路径。如果我们在 A 中每个顶点上包含自环，那么 A^2 也有 A 中的这些边；否则，则没有。图 19-14 中描述了布尔矩阵相乘和有向图中的路径。由此直接得到一种精巧的方法来计算任何有向图中的传递闭包。

性质 19.6　可以通过构造有向图的邻接矩阵 A，再在每个顶点上加上自环，并计算 A^V，来计算有向图的传递闭包。

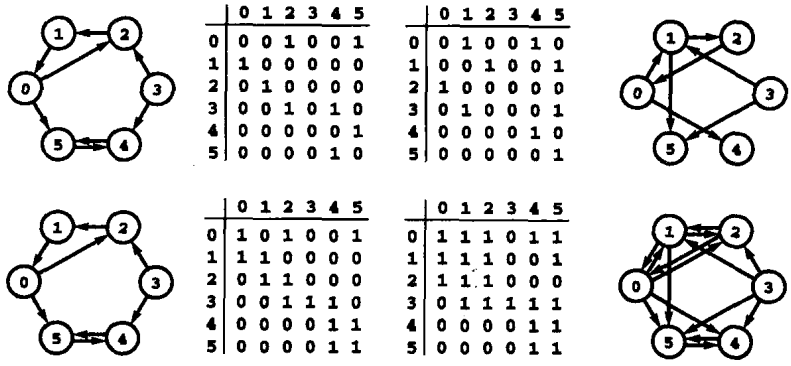


图 19-14　邻接矩阵平方

如果将一个有向图邻接矩阵的对角线上的元素均设置为 0，则此矩阵平方所表示的图中的每条边对应于原有向图中长度为 2 的路径（上图），如果将对角线上元素均设置为 1，此矩阵平方所表示的图中的每条边对应于原有向图长度为 1 或 2 的路径（下图）。

证明　继续上一段的证明过程，对应有向图中长度小于或等于 3 的每条路径 A^3 中均有一条边，对应有向图中长度小于或等于 4 的每条路径 A^4 中均有一条边，如此等等。由鸽巢原理，我们无需考虑长度大于 V 的路径：任何长度大于 V 的路径都必然要再次访问某些顶点（因为仅有 V 个顶点），而且由于同样的两个顶点已经由长度小于 V 的一条有向路径所连接（通过去除指向再次访问顶点的环，可得到这样的路径），因此它对于传递闭包不会增加任何信息。 ■

图 19-15 显示了一个示例有向图收敛到传递闭包时的邻接矩阵的幂方。此方法需要进行 V 次矩阵相乘，每一次所需时间均与 V^3 成正比，总共所需时间为 V^4。实际上，我们可以只用 $\lceil \lg V \rceil$ 次布尔矩阵相乘操作来计算任何有向图的传递闭包：可以计算 A^2、A^4、A^8，…，直到达到一个大于或等于 V 的指数时为止。如在性质 19.6 中的证明那样，对于任意 $t > V$，$A^t = A^V$；因此，这个计算的结果 A^V 就是传递闭包，所需时间与 $V^3 \lg V$ 成正比。

虽然上述描述的方法非常简洁，极具吸引力。但还有一个更简单的方法。可以用一次操作计算这个传递闭包，由邻接矩阵原位构建传递闭包，如下所示：

```
for (i = 0; i < V; i++)
  for (s = 0; s < V; s++)
    for (t = 0; t < V; t++)
      if (A[s][i] && A[i][t]) A[s][t] = 1;
```

这个经典的方法由 S. Warshall 在 1962 年发明，是计算稠密有向图的传递闭包时所选择的方法。此代码类似

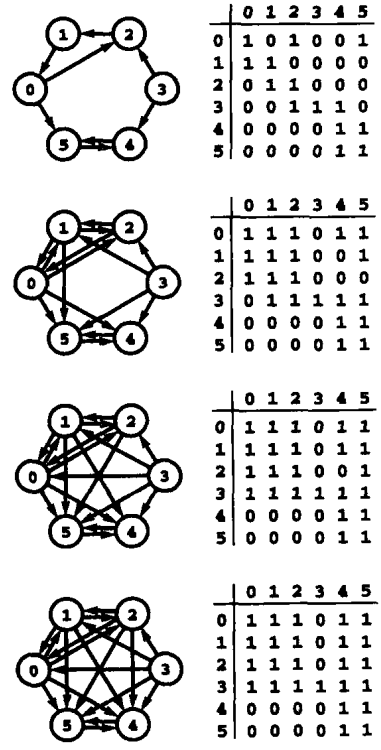


图 19-15　邻接矩阵的幂方与有向路径

此序列显示了右上图的邻接矩阵的 1 次幂，2 次幂，3 次幂和 4 次幂，并给出了 4 个图（左列，从上到下），其中每条边对应矩阵表示的图中的长度分别小于 1、2、3 和 4 的路径。下图是此例的传递闭包，不存在长度大于 4 的路径所连接的顶点，因为它们会由更短路径连接。

于在原位计算布尔矩阵平方所用的代码：差别（显著差别）！在于 for 循环的顺序。

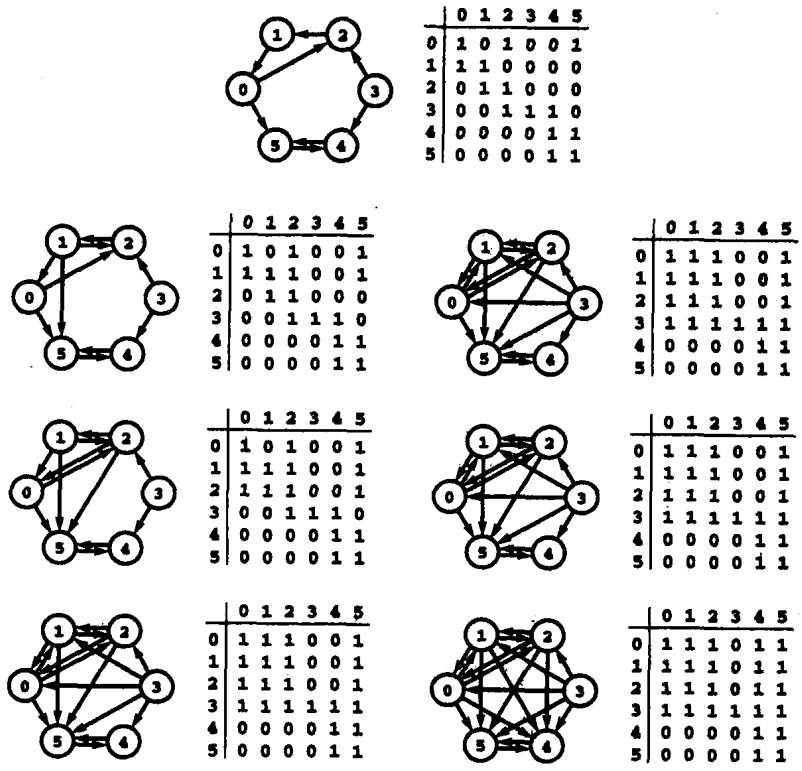


图 19-16　Warshall 算法

此序列显示了利用 Warshall 算法来计算一个示例有向图（上图）的传递闭包（下图）的过程。循环的第 1 次迭代（左列最上图）增加了边 1-2 和 1-5，这是因为存在路径 1-0-2 和 1-0-5，它们都包含顶点 0（但不包含带有更大编号的顶点）；循环的第 2 次迭代（左列自上第 2 个图）则增加了边 2-0 和 2-5，这是因为存在路径 2-1-0 和 2-1-0-5，其中包含顶点 1（但不包含带有更大编号的顶点）；而循环的第 3 次迭代（左列最下面的图）将增加边 0-1、3-0、3-1 和 3-5，这是因为存在路径 0-2-1、3-2-1-0、3-2-1 和 3-2-1-0-5，它们都包含顶点 2（但不含带有更大编号的顶点）。右边一列则显示了考虑到路径通过 3、4 和 5 时所增加的边。循环的最后一次迭代（右列最下图）将增加分别从 0、1 和 2 指向 4 的边，因为从这些结点指向 4 的唯一的有向路径都包含 5，即最大编号的顶点。

性质 19.7　利用 Warshall 算法，可以在与 V^3 成正比的时间计算出一个有向图的传递闭包。

证明　根据代码的结构，可直接得出其运行时间。我们通过对 i 进行归纳来证明它可以计算出传递闭包。在循环第 1 次迭代后，矩阵的 s 行、t 列上有一个 1，当且仅当有一条路径 s-t 和 s-0-t。第 2 次迭代检查 s 和 t 之间包含 1、可能还有 0 的所有路径，如 s-1-t，s-1-0-t 和 s-0-1-t。我们做出以下归纳假设：循环的第 i 次迭代将矩阵中 s 行 t 列出设置为 1，当且仅当有向图中从 s 到 t 存在一条有向路径，且这条路径上不包含索引大于 i 的任何顶点（除了可能的端点 s 和 t 以外）。如上所述，i 为 0 时，即循环第 1 次迭代后此条件成立。假设此条件对于循环的第 i 次迭代成立，从 s 到 t 存在一条其中不含索引大于 $i+1$ 的任何顶点的路径，当且仅当（ⅰ）存在一条不含索引大于 i 的任何顶点的从 s 到 t 的路径，这种情况下，在循环的前一次迭代时 A[s][t] 即被设置（由归纳假设）；或者（ⅱ）存在从 s 到 $i+1$ 以及从 $i+1$ 到 t 的路径，在这两条路径中都不含索引大于 i 的任何顶点（端点除外），这种情况下，此前 A[s][i+1] 和 A[i+1][t] 均设置为 1（由假设）。因此内循环将把 A[s][t] 设置

为 1。 ■

我们可以通过对代码做一个简单的变换来改进 Warshall 算法的性能：将对 A[s][i] 的测试从内循环中移出，因为在 t 变化时其值并未做改变。当 A[s][i] 为 0 时，这种移动使我们能够完全避免执行 t 次循环。由这种改进节省的时间取决于有向图，而且对于许多有向图来说都改进相当可观（见练习 19.54 和 19.55）。程序 19.3 实现了这个改进，并将 Warshall 方法封装为有向图的一对 ADT 函数，从而使客户程序能够对有向图进行预处理（计算传递闭包），然后用常量时间计算出任何可达性查询的结果。

程序 19.3　Warshall 算法

这个 Warshall 算法的 ADT 实现，在调用 GRAPHtc 计算传递闭包之后，为客户端提供了一种测试有向图中的任何顶点是否由其他顶点可达的能力。图表示中的数组指针 tc 用于存储传递闭包矩阵。

```
void GRAPHtc(Graph G)
  { int i, s, t;
    G->tc = MATRIXint(G->V, G->V, 0);
    for (s = 0; s < G->V; s++)
      for (t = 0; t < G->V; t++)
        G->tc[s][t] = G->adj[s][t];
    for (s = 0; s < G->V; s++) G->tc[s][s] = 1;
    for (i = 0; i < G->V; i++)
      for (s = 0; s < G->V; s++)
        if (G->tc[s][i] == 1)
          for (t = 0; t < G->V; t++)
            if (G->tc[i][t] == 1) G->tc[s][t] = 1;
  }
int GRAPHreach(Graph G, int s, int t)
  { return G->tc[s][t]; }
```

我们对找到更高效的解决方法感兴趣，特别是对于稀疏图。我们希望降低预处理时间和空间，因为对于稀疏图，这两方面都使得 Warshall 算法的代价太高。

在现代应用中，抽象数据类型为我们提供了将操作从任何特定实现中分离出来的能力，因此，我们可以集中在高效实现上。对于传递闭包问题，这一观点使我们认识到不需要计算出整个矩阵，就能为客户程序提供传递闭包抽象。一种可能性就是，传递闭包是一个大型稀疏矩阵，因而使用邻接表表示。因为我们无法存储数组表示。甚至在传递闭包为稠密时，客户程序可能只检查其中很少部分的边，因而计算整个矩阵是相当浪费的。

我们使用术语*抽象传递闭包*（abstract transitive closure）来指一个 ADT，它为客户程序提供在预处理之后测试可达性的能力，如程序 19.3 所示。在本文中，不仅需要度量一个算法计算传递闭包（预处理的开销）的开销，而且还要度量它所需要的空间和所用的查询时间。也就是说，我们得性质 19.7 重述如下：

性质 19.8　*对于一个有向图，可在常量时间内支持可达性测试（抽象传递闭包），所用空间与 V^2 成正比，预处理的时间与 V^3 成正比。*

证明　根据 Warshall 算法的基本性能特征可以立即得出此性质。 ■

对于大多数的应用，我们的目标是不仅仅是快速地计算出有向图的传递闭包，而且与性质 19.8 相比，要用少得多的空间和预处理时间使抽象传递闭包支持常量的查询时间。能否找到一种实现，使我们能够构建客户程序有能力处理这种有向图呢？我们将在 19.8 节再来

讨论这个问题。

在计算有向图的传递闭包问题和大量其他基础性的计算问题之间存在一种密切的关系。这种关系可以帮助我们理解这个问题的难度。我们讨论这些问题的两个例子来结束本节。

首先，我们来考虑传递闭包与所有点对之间的最短路径（all-pairs shortest path）问题的关系（见18.7节）。对于有向图的每对顶点，该问题是找出具有最少边数的一条有向路径。在18.7节所考虑的无向图上的基于BFS的方法对于有向图也适用（做相应修改），但是目前我们只对修改Warshall算法来解决这个问题有兴趣。最短路径是第21章中的主题，因此，我们到第21章再详细考虑这两者的性能比较。

给定一个有向图，我们初始化 $V \times V$ 的一个整数矩阵，如果从 s 到 t 存在一条边，则设A[s][t]为1，否则设为观察哨值 V。目标是使A[s][t]为从 s 到 t 的最短有向路径长度（在这条路径上的边数），使用观察哨值 V 来表示从 s 到 t 不存在路径。以下代码完成这个目标：

```
for (i = 0; i < V; i++)
  for (s = 0; s < V; s++)
    for (t = 0; t < V; t++)
      if (A[s][i] + A[i][t] < A[s][t])
        A[s][t] = A[s][i] + A[i][t];
```

这个代码与性质19.7中所看到的Warshall算法的不同只在于内循环中的if语句不同。实际上，采用适当的抽象设置，计算是完全一样的（见练习19.56和19.57）。将性质19.7中的证明转换为这个方法完成既定目标的直接证明非常简单。这个方法是Floyd算法查找加权图的最短路径的一个特例（见第21章）。

其次，我们已经看到，传递闭包问题与布尔矩阵相乘问题也是非常相似的。我们所见的这两个问题的基本算法所需时间与 V^3 成正比（所使用的计算机制类似）。已经知道布尔矩阵相乘是一个困难的计算问题：比直接方法更快的算法也已知道，但是节省的时间是否足够大，足以抵消实现这些算法的努力，这一点还存在争议。这个事实在目前看来非常重要，因为我们可以使用图19-15中描述的反复平方法，采用求解布尔矩阵相乘的一个快速算法来开发一个更快的传递闭包算法（仅慢一个 $\lg V$ 因子）。反之，我们得到计算传递闭包问题难度的一个下界：

性质19.9 可以使用任何传递闭包算法来计算两个布尔矩阵相乘的乘积，而运行时间最多存在常量因子的差别。

证明 给定两个 $V \times V$ 的布尔矩阵 A 和 B，构造如下的 $3V \times 3V$ 的矩阵：

$$\begin{pmatrix} I & A & 0 \\ 0 & I & B \\ 0 & 0 & I \end{pmatrix}$$

这里，0表示其所有元素均为0的 $V \times V$ 矩阵，I 表示除对角线上为1其余元素均为0的 $V \times V$ 矩阵。现在，我们将这个矩阵考虑为有向图的一个邻接矩阵，并通过反复平方法来计算它的传递闭包。

$$\begin{pmatrix} I & A & 0 \\ 0 & I & B \\ 0 & 0 & I \end{pmatrix}^2 = \begin{pmatrix} I & A & A*B \\ 0 & I & B \\ 0 & 0 & I \end{pmatrix}$$

等式右边的矩阵为传递闭包，因为进一步相乘得到前面的相同矩阵。但这个矩阵在其右上角处有一个 $V \times V$ 的乘积 $A * B$。无论使用哪种算法求解传递闭包问题，都可以以相同的开销使

用它来求解矩阵相乘问题。 ■

此性质的重要性依赖于专家认定布尔矩阵相乘问题是困难问题：数学家数十年来一直试图准确地了解它的难度，但问题依然没有解决。已知最好结果的运行时间大约与 $V^{2.5}$ 成正比（见第五部分参考文献）。现在，如果我们能够找出求解传递闭包问题的一个线性时间（与 V^2 成正比）解，那么我们也能得到求解布尔矩阵相乘问题的线性时间解。问题之间的这种关系称为*归约*（reduction）：称布尔矩阵相乘问题可以归约到传递闭包问题（见 21.6 节和第八部分）。实际上，证明确实表明，布尔矩阵相乘可以归约到找出一个有向图中长度为 2 的路径。

尽管很多人做了大量的研究，还没有人能够找出一个线性时间的布尔矩阵相乘的算法，因此，也就不能给出一个简单的线性时间的传递闭包算法。另一方面，也没有人证明这样的算法不存在，因此我们对于存在这种算法的可能性持开放的态度。简而言之，通过性质 19.9 说明，除非在研究上取得突破，我们不能期望所构造的任何传递闭包算法在最坏情况下的运行时间与 V^2 成正比。虽则如此，我们仍可以开发某类有向图的快速算法。例如，我们已经触及到计算传递闭包的一种简单方法，对于稀疏图而言，它要比 Warshall 算法快得多。

性质 19.10　*对于有向图的抽象传递闭包，使用 DFS 可以支持常量的查询时间，而且所用空间与 V^2 成正比，所用预处理时间与 $V(V+E)$ 成正比（计算传递闭包）。*

证明　在上一节中看到，DFS 给出了由起始顶点可达的所有顶点，如果使用邻接表表示（见性质 19.5 和图 19-11），所用时间与 E 成正比。因此，如果我们将 DFS 运行 V 次，每个顶点作为起始顶点一次，就能计算出由每个顶点可达的顶点集合，也即传递闭包，所用时间与 $V(E+V)$ 成正比。对于线性时间的广义搜索，这一结论同样成立（见 18.8 节和练习 19.69）。 ■

程序 19.4　基于 DFS 的传递闭包

此程序在功能上等价于程序 19.3。它从每个顶点开始分别完成 DFS 计算出其可达的结点集，来计算出传递闭包。每次对递归过程的调用都增加由起始顶点开始的一条边，并进行递归调用填充传递闭包矩阵中对应的行。此矩阵也可用于在 DFS 中标记访问过的顶点。

```
void TCdfsR(Graph G, Edge e)
  { link t;
    G->tc[e.v][e.w] = 1;
    for (t = G->adj[e.w]; t != NULL; t = t->next)
      if (G->tc[e.v][t->v] == 0)
        TCdfsR(G, EDGE(e.v, t->v));
  }
void GRAPHtc(Graph G, Edge e)
  { int v, w;
    G->tc = MATRIXint(G->V, G->V, 0);
    for (v = 0; v < G->V; v++)
      TCdfsR(G, EDGE(v, v));
  }
int GRAPHreach(Graph G, int s, int t)
  { return G->tc[s][t]; }
```

程序 19.4 是基于这种搜索的传递闭包算法的一种实现。对于图 19-1 中的示例有向图运行此程序的结果显示在图 19-11 中的每个森林的第一棵树中。封装此实现的方法如同程序 19.3 中封装 Warshall 算法一样：一个计算传递闭包的预处理函数，以及通过检查传递闭包数组中所指示的元素从而在常量时间确定是否任何顶点可由另一顶点可达的函数。

对于稀疏有向图，这种基于搜索的方法是一种可以选择的方法。例如，如果 E 与 V 成正比，那么程序 19.4 计算传递闭包所需时间与 V^2 成正比。给定我们所考虑的到布尔矩阵相乘的归约，如何做到这一点呢？答案就是此传递闭包算法实际上是为某些类型（certain type）的布尔矩阵（那些有 $O|V|$ 个非 0 元素的矩阵）相乘提供了一种最优的方法。由下界得知，我们不应该期望找到对于所有有向图其运行时间与 V^2 成正比的一个传递闭包算法，但并没有排除找到这种算法的可能性，就像这个算法，对于某类的有向图就会更快。如果这样的图就是我们需要处理的图，那么传递闭包与布尔矩阵的关系可能与我们无关。

很容易对本节描述的方法进行扩展，像 17.8 节所描述的那样，通过保存搜索树，为客户程序提供找出连接两个顶点的一条特定路径的能力。在第 21 章中，我们在更一般的最短路径问题的环境下，考虑这类特定 ADT 的实现。

对于本节所描述的基本传递闭包算法，表 19-1 比较了它们的实验结果。基于搜索的算法的邻接表实现对于稀疏图是最快的方法。这些实现都计算出一个大小为 V^2 的邻接矩阵，因此，它们都不适合于大型稀疏图。

对于稀疏有向图，其传递闭包也是稀疏的，可以使用一个邻接表实现此闭包，使得输出的规模与传递闭包中的边数成正比。这个数也是计算传递闭包的开销的下界，对于某种类型的有向图，使用各种算法学技术（见练习 19.67 和 19.68）也能达到这个下界。虽然存在这种可能性，我们一般将传递闭包计算的目标看作是邻接矩阵表示，因而可以很容易地回答可达性查询，而且将计算该矩阵的传递闭包算法认为是最优的，它与 V^2 成正比，因为这些算法所需的时间与输出的规模成正比。

如果邻接矩阵是对称的，则等价于一个无向图，并且找出传递闭包就如同找出连通分量，即传递闭包是连通分量中顶点的完全图的并集（见练习 19.49）。18.5 节中的连通性算法可看作是对称有向图（无向图）的抽象传递闭包计算，所使用的空间与 V 成正比，但仍然支持常量时间的可达性查询。对于一般的图的也能这样做吗？可以进一步地降低预处理时间吗？对于哪种类型的图可以在线性时间内计算出其传递闭包呢？要回答这些问题，我们需要更详细地研究有向图的结构，特别是 DAG 的结构。

表 19-1 传递闭包算法的实验研究

此表显示的运行时间展示了各种计算随机有向图的传递闭包算法，无论是对于稠密图还是稀疏图都有着巨大的性能差异。当顶点数 V 加倍时，除了邻接表 DFS，其他算法的运行时间都增长 8 倍，这与 V^3 成正比的结论一致。邻接表 DFS 所需时间与 VE 成正比，这表明当 V 和 E 均加倍时（稀疏图），算法的运行时间大约增长 4 倍；当 V 和 E 均加倍时（稠密图），算法的运行时间大约增长 2 倍；但对于高度稠密图，表遍历的开销使得性能有所下降。

稀疏图（$10V$ 条边）					稠密图（250 个顶点）				
V	W	W*	A	L	E	W	W*	A	L
25	0	0	1	0	5 000	289	203	177	23
50	3	1	2	1	10 000	300	214	184	38
125	35	24	23	4	25 000	309	226	200	97
250	275	181	178	13	50 000	315	232	218	337
500	2 222	1 438	1 481	54	100 000	326	246	235	784

说明：W Warshall 算法（19.3 节）
W* 改进的 Warshall 算法（程序 19.3）
A DFS，邻接矩阵表示（练习 19.64）
L DFS，邻接标表示（程序 19.4）

练习

▷ 19.47　对于只包含 V 个顶点的一个有向环的有向图，其传递闭包是什么？

19.48　对于只包含 V 个顶点的一条简单有向路径的有向图，其传递闭包中有多少条边？

▷ 19.49　给出以下无向图的传递闭包

3-7 1-4 7-8 0-5 5-2 3-8 2-9 0-6 4-9 2-6 6-4

● 19.50　显示如何构造一个 V 个顶点和 E 条边的有向图，具有性质：对于介于 E 和 V^2 之间的任何 t 值，其传递闭包中的边数与 t 成正比。与往常一样，假设 $E > V$。

19.51　如果有向图是一个有向森林，给出其传递闭包边数的一个公式，它是该森林结构性质的一个函数。

19.52　按照图 19-15 的风格，使用反复平方法，显示计算以下有向图的传递闭包的过程：

3-7 1-4 7-8 0-5 5-2 3-8 2-9 0-6 4-9 2-6 6-4

19.53　按照图 19-16 的风格，使用 Warshall 算法，显示计算以下有向图的传递闭包的过程：

3-7 1-4 7-8 0-5 5-2 3-8 2-9 0-6 4-9 2-6 6-4

○ 19.54　给出一组稀疏图，使得 Warshall 算法计算传递闭包的改进版本（程序 19.3）的运行时间与 VE 成正比。

○ 19.55　给出一个稀疏图，使得 Warshall 算法计算传递闭包的改进版本（程序 19.3）的运行时间与 V^3 成正比。

○ 19.56　开发带有相应实现的整型矩阵的一个 ADT，使得单个客户程序能够包含 Warshall 算法和 Floyd 算法。（该练习是练习 19.57 的一个版本，针对那些对抽象数据类型比抽象代数更熟悉的人们。）

● 19.57　使用抽象代数开发一个既包含 Warshall 算法又包含 Floyd 算法的通用算法。（该练习是练习 19.56 的一个版本，针对那些对抽象代数比抽象数据类型更熟悉的人们。）

○ 19.58　按照图 19-16 的风格，对于示例图，显示用 Floyd 算法计算所有点对之间最短路径矩阵的开发过程。

19.59　两个对称矩阵的布尔乘积矩阵是对称的吗？解释你的答案。

19.60　修改程序 19.3 和程序 19.4，为有向图 ADT 函数提供一种实现，返回该有向图的传递闭包中的边数。

19.61　使用计数器作为有向图的 ADT 的一部分，实现练习 19.60 中描述的函数，并在添加边或删除边时对它进行修改。给出利用此实现添加边和删除边的开销。

▷ 19.62　增加一个用于程序 19.3 和程序 19.4 的有向图 ADT 函数，返回顶点索引的数组，它指出由给定顶点可达的那些顶点。可以要求客户程序已经调用 GRAPHtc 进行了预处理。

○ 19.63　进行实验研究，对于各种类型的有向图（见练习 19.11 ~ 19.18），确定传递闭包中的边数。

▷ 19.64　对于邻接矩阵表示，实现基于 DFS 的传递闭包算法。

● 19.65　考虑练习 17.21 中描述的位数组图表示法。哪种方法能够带来 B 倍的加速（B 是你的计算机中每个字的位数）：Warshall 算法还是基于 BFS 的算法？通过开发一个达到此要求的实现来证实你的结果。

○ 19.66　开发一个独立于表示的抽象有向图 ADT（见练习 17.60），包括一个传递闭包含函数。注意：传递闭包本身应该是一个抽象有向图，不引用任何特定的表示。

○ **19.67** 如果有向图是一个有向森林，给出计算其邻接表表示的传递闭包的程序，所用时间与传递闭包中的边数成正比。

○ **19.68** 实现稀疏图的一个抽象传递闭包算法，使用空间与 T 成正比，在所用时间与 $VE+T$ 成正比的预处理后，可在常量时间内回答可达性查询，其中 T 是传递闭包中的边数。提示：使用动态散列。

▷ **19.69** 提供基于广义图搜索（见 18.8 节）的程序 19.4 的一个版本。

19.4 等价关系和偏序

本节所讨论的是集合论中的基本概念，以及这些概念与抽象传递闭包算法之间的关系。其目的是将我们所研究的思路置于一个更大的环境中，并且说明所讨论的算法有着广泛的应用性。对于那些熟悉集合论的数学方向的读者，可以直接跳到 19.5 节，因为这里所介绍的内容是基础性的（但对这些术语做一个简要的回顾可能会有所帮助）；对于不熟悉集合论的读者，则可能希望参考离散数学方面的相关文献，因为这里的介绍相当简洁。有向图和这些基本数学概念之间的联系是如此重要，因而我们不能忽视。

给定一个集合，其对象之间的*关系*（relation）定义为对象的有序对集合。除了可能涉及平行边和自环的细节上有所不同，此定义与我们对有向图的定义完全一样：关系和有向图是同一抽象的不同表示。这里数学概念更强一些，因为集合可以是无限的，而我们的计算机程序只能处理有限集合，不过目前先忽略此差别。

通常，我们选用一个符号 R，并使用 sRt 记法作为语句“有序对（s，t）之间存在关系 R”的简写。例如，我们用符号“$<$”来表示数字间的“小于”关系。利用这种术语，则可表征关系的各种性质。例如，如果对于所有 s 和 t，sRt 蕴含着 tRs，则称关系 R 是*对称的*（symmetric）；如果对于所有 s，都有 sRs，则称关系 R 是*自反的*（reflexive）。对称关系与无向图相同。自反关系则对应于所有顶点均有自环的图；对于任何顶点都没有自环的图，与之对应的关系则称为*反自反的*（irreflexive）。

对于所有 s、t 和 u，如果有 sRt 和 tRu 蕴含着 sRu，则称关系 R 是*传递的*（transitive）。关系的*传递闭包*（transitive closure）是一个良定义的概念；但在此并不打算用集合论的术语再对它进行定义，我们希望采用 19.3 节中给出的对于有向图定义。任何关系都等价于一个有向图，而关系的传递闭包则等价于有向图的传递闭包。任何关系的传递闭包也是传递的。

在图算法领域中，我们特别感兴趣的是通过进一步约束来定义的两个特定传递关系。这两种得到广泛应用的关系，分别称为*等价关系*（equivalence relation）和*偏序*（partial order）。

等价关系“$\equiv$”是一种自反且对称的传递关系。注意，对于某个有序对，一个包含其中每个对象的对称、传递关系必定也是一个等价关系：如果 $s\equiv t$，那么 $t\equiv s$（由对称性）和 $s\equiv s$（由传递性）。等价关系将一个集合中的对象划分为*等价类*（equavalence class）的子集。两个对象 s 和 t 是在同一个等价类中，当且仅当 $s\equiv t$。以下例子是典型的等价关系：

模运算 任何正整数 k 定义了整数集合上的一个等价关系，且 $s\equiv t\ (\bmod k)$ 当且仅当 s 除以 k 的余数等于 t 除以 k 的余数相等。此关系显然是对称的；简单证明确定此关系也是传递的（见练习 19.70），因而它是一个等价关系。

图中的连通性 顶点之间的关系“位于同一连通分量中”是一个等价类，因为它是对称且传递的。该等价类对应于图中的连通分量。

在构建一个图 ADT 以使客户有能力检查两个顶点是否在同一个连通分量中，我们实现了一个等价关系 ADT，它能为客户提供检查两个对象是否等价的能力。在实际中，这个对应

非常重要，因为该图是等价关系的一种简洁表示（见练习 19.74）。实际上，如同我们在第 1 章和第 18 章中所看到的，为了构建这样一个 ADT，我们只需要维持单个顶点索引的数组。

偏序 $\prec$ 也是一个反自反的传递关系。作为传递性和非自反性的直接结果，容易证明偏序也是反对称的（asymmetric）：如果 $s \prec t$ 且 $t \prec s$，那么 $s \prec s$（由传递性），这与反自反性矛盾，因而 $s \prec t$ 且 $t \prec s$ 不能同时成立。此外，扩展某些论述过程表明偏序不能存在环，比如 $s \prec t$，$t \prec u$ 和 $u \prec s$。以下例子是典型的偏序：

子集包含　一个给定集合的子集之间的关系“包含但不等于”（$\subset$）是一个偏序，也即它肯定是反自反的，而且如果 $s \subset t$ 且 $t \subset u$，则必定有 $s \subset u$。

DAG 中的路径　对于不含自环的 DAG 中的顶点，关系“通过非空有向路径可达的”是一个偏序，因为它是传递的和反自反的。类似于等价关系和无向图，这个特定的偏序对于很多应用非常重要，因为一个 DAG 给出了偏序的一种简明的隐式表示。例如，图 19-17 说明了子集包含偏序的 DAG，其边数只是偏序基数的一部分（见练习 19.76）。

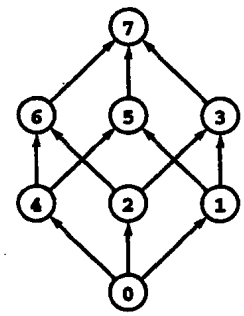
实际上，我们很少通过枚举它们的所有有序对来定义偏序，因为这样的有序对太多。而是，我们通常指定一个反自反的关系（DAG），并考虑其传递闭包。对于 DAG，这种用法是考虑抽象 - 传递 - 闭包 ADT 实现的一个主要原因。使用 DAG，就可以考虑 19.5 节中的偏序示例。

对于所有 $s \neq t$，全序（total order）T 就是 sTt 或 tTs 只能有一个成立。我们所熟悉的全序例子是整数或实数之间以及字符串之间的字典顺序的“小于”关系。我们在第 3 部分和第 4 部分关于排序和搜索算法的研究是基于对集合的一个全序 ADT 实现。在一个全序中，又且只有一个方式来排列集合中的元素，使得在排列中，只要 s 在 t 之前，就有 sTt；在一个不含全序的偏序中，可能有多种方式这样做。在 19.5 节中，我们将考察完成此任务的算法。

总而言之，以下关于集合和图模型之间的对应关系有助于我们理解基本图算法的重要性和广泛应用性。

- 关系和有向图
- 对称关系和无向图
- 传递关系和图中的路径
- 等价关系和无向图中的路径
- 偏序和 DAG 中的路径

以上内容使我们对于所研究的图类型和算法有了更深入的认识，而且促使我们要对 DAG 的基本性质以及处理这些 DAG 的算法作进一步的研究。



0	000	ϕ
1	001	$\{c\}$
2	010	$\{b\}$
3	011	$\{b,c\}$
4	100	$\{a\}$
5	101	$\{a,c\}$
6	110	$\{a,b\}$
7	111	$\{a,b,c\}$

图 19-17　子集包含 DAG

在上图的 DAG 中，我们将顶点索引解释为一个 3 元素集合的子集，如下面的表中所示。此 DAG 的传递闭包表示了子集包含偏序：两个结点之间存在一条有向路径，当且仅当第一个结点表示的子集被包含在第二个结点索表示的子集中。

练习

19.70　说明整数集合的“除以 k 之后余数相同”是一个传递关系（因此也是一个等价关系）。

19.71　说明任何图中顶点之间的“位于同一个连通分量中”是一个等价关系。

19.72　说明对于所有图中的顶点之间的“位于同一双连通分量中”不是等价关系。

19.73　证明等价关系的传递闭包是一个等价关系，因而偏序的传递闭包是一个偏序。

▷ 19.74　关系的基数是其有序对的个数。证明等价关系的基数等于此关系等价类的基数的平方和。

○ **19.75** 使用在线字典，构建字之间的一个表示等价关系“有 k 个公共字符”的图。对于 $k=1\sim5$，确定等价类的数目。

19.76 偏序的基数是其有序对的数目。对于一个 n 个元素的集合，其子集包含偏序的基数是多大？

▷ **19.77** 说明整数之间的“一个整数是另一个整数的倍数”是一个偏序。

19.5 有向无环图

在这一节中，我们考虑有向无环图（DAG）的各种应用。讨论有向无环图有两个原因。首先是因为有向无环图可作为偏序的隐式模型，在很多应用中直接利用 DAG，且需要有效的算法对其进行处理。其次是各种应用使我们可以对 DAG 有深入的认识，而且理解 DAG 对于理解一般有向图非常重要。

由于 DAG 是一种特殊类型的有向图，所有 DAG-处理的问题都可以简单地归约为有向图处理问题。比起处理一般的有向图，虽然希望处理 DAG 更容易，但我们知道遇到用 DAG 难解的问题时，不能指望在一般的有向图上求解同一个问题会更好。我们将要看到，计算传递闭包的问题就属于这类问题。相反，理解处理 DAG 的难度很重要，因为每个有向图都有一个核心 DAG（见性质 19.2），因而即使所处理的有向图不是 DAG 时，也会遇到 DAG。

直接出现 DAG 的原型应用称为*调度*（scheduling）。一般而言，求解调度问题必须在一组约束之下，通过指定何时及如何执行任务，来安排完成一组任务。约束可能是所需时间的函数或者任务消耗的其他资源的函数。最重要的约束类型是*优先约束*（precedence constraint），指定了某些任务必定在其他一些任务之前执行。不同类型的额外约束则导致了很多不同类型的调度问题，难度也差别很大。数以千计的不同问题已经得到了研究。研究人员仍在寻求其中很多问题的更好的算法。也许最简单的非平凡调度问题阐述如下。

调度（scheduling） 给定要完成的一组任务，其上的偏序指定了某些任务必须在其他一些任务开始之前完成。如何调度任务从而在保持偏序的同时又能完成任务呢？

这种基本形式的调度问题称为*拓扑排序*（topological sorting）；它并不难解决。下一节会讨论解决这个问题的两个算法。在更复杂的实际应用中，我们可能需要增加如何调度任务的其他的约束条件，因而问题变得困难得多。例如，任务可能对应着学生课表的课程，指定了先修课程的偏序关系。拓扑排序给出了一种满足先修课程要求的可行的课程调度，但要在此模型中添加其他约束可能就不满足了，如课程冲突、选课限制等。另一个例子，任务可能是制造过程的一部分，指定了特定过程顺序要求的偏序关系。拓扑排序给出了一种调度这些任务的方法，但也可能存在需要更少时间、金钱或模型中未包含的其他资源的另一种调度方式。我们将在第 21 和 22 章讨论能够涵盖更一般情况的调度问题。

尽管与有向图有很多的相似性，但在实现 DAG 特定的算法时，采用针对 DAG 的不同 ADT 更为合适。在这些情况下，我们使用类似程序 17.1 的 ADT 接口，用 DAG 替换掉所有的 GRAPH。19.6 和 19.7 节致力于实现拓扑排序的 ADT 函数（DAGts）和 DAG 中的可达性的 ADT 函数（DAGtc 和 DAGreach）；程序 19.13 是此 ADT 客户程序的一个例子。

我们的第一个任务常常是检查一个给定的 DAG 是否不含有向环。我们在 19.2 节已看到，通过运行一个标准的 DFS 并检查 DFS 森林不存在回边，可以在线性时间内检查一个一般有向图是否是一个 DAG。DAG 图的 ADT 应该包含一个允许客户程序执行这种检查的 ADT 函数（见练习 19.78）。

在某种意义上，DAG 半树半图。处理它们时肯定可以利用其特殊结构。例如，如果需

要，几乎可以将 DAG 看作一棵树。以下简单程序就像是递归树遍历：

```
void traverseR(Dag D, int w)
 { link t;
   visit(w);
   for (t = D->adj[w]; t != NULL; t = t->next)
     traverseR(D, t->v);
 }
```

此程序的结果是遍历 DAG 中的顶点，就好像它是一棵根为 w 的树。例如，用这个程序遍历图 19-18 中的两个 DAG，将会得到一样的结果。然而，我们很少使用这样的完全遍历，因为我们通常想要利用 DAG 中节省空间的同样措施来节省其遍历时间（例如，在一个常规的 DFS 中，对所访问过的顶点进行标记）。同样的思想应用到搜索上（search），只对依附每个顶点的一条链接进行递归调用。在这样的算法中，搜索开销对于 DAG 和树是一样的。但 DAG 使用的空间要少得多。

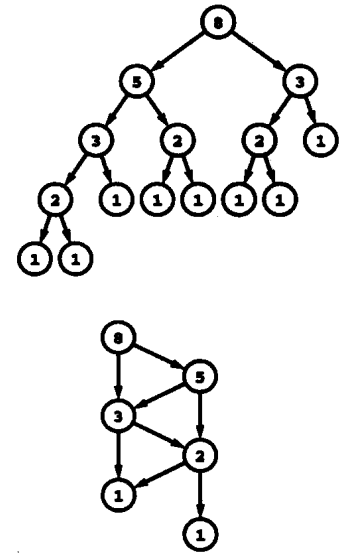


图 19-18　斐波纳契计算的 DAG 模型

上图的树显示了每个斐波纳契数的计算依赖于它的两个前驱。下图的 DAG 用一部分结点显示了同样的依赖关系。

因为 DAG 提供了一种简洁方法表示含有相同子树的树，因而在表示计算抽象时，我们常常使用 DAG 而不是树。在算法设计的环境中，执行中程序的 DAG 表示和树表示之间的差别正是动态规划的基本差别（例如，见图 19-18 和练习 19.81）。DAG 有广泛的应用，如在编译器中作为算术表达式和程序的中间表示（例如，见图 19-19），以及在电路设计系统中作为组合电路的中间表示。

除了这些应用之外，考虑二叉树时也有一个重要的例子有很多应用。我们可以将对 DAG 的同样的限制应用到树上来定义二叉树。

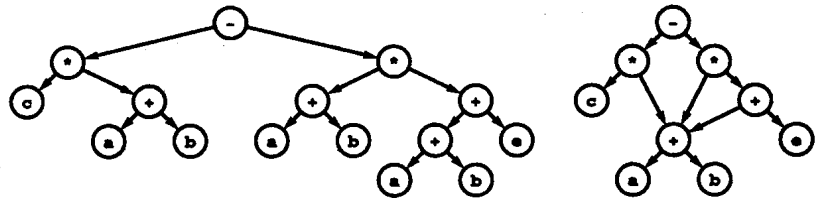


图 19-19　算术表达式的 DAG 表示

这两个 DAG 都表示算术表达式 (c*(a+b))-((a+b)*((a+b)+e))。在左边的二叉分析树中，叶结点表示操作数，每个内部结点表示应用到其两个子树（见图 5.31）所表示表达式的运算符。右边的 DAG 是同一棵树的更简洁表示。更重要的是，我们可以计算表达式的值，所用时间与 DAG 的规模成正比，这要比树的规模小得多（见练习 19.114 和 19.115）。

定义 19.6　*一棵二叉树是一个有向无环图，每个结点均发出两条边，分别标识为左边和右边，每条边都可能为空，或者两者都为空。*

二叉 DAG 与二叉树的区别在于，在二叉 DAG 中，指向一个结点的链接可以不止一条。如同对于二叉树的定义，这个定义也建立了一个自然的表示模型，其中每个结点是一个结构，带有指向其他结点（或空）的左链接和右链接，条件是只有一个全局限制：不允许有向环存在。二叉 DAG 非常重要，因为它们为某些应用提供了二叉树的一种简洁表示。例如，我们可以将一个存在线索压缩为一个二叉 DAG，而不必改变搜索实现，如图 19-20 和程序 19.5 所示。

程序 19.5 用二叉 DAG 来表示一个二叉树

这个递归程序是一个后序遍历，通过识别出公共子树，用来构造与二叉树的结构（见第 12 章）所对应的二叉 DAG 的一个简洁表示。我们使用一个类似程序 17.10 中 STindex 的索引函数，对其修改使其接受整数而非字符串。对于每个不同的树结构赋以一个唯一的整数，在将 DAG 表示为一个 2-整数结构的数组时使用（见图 19-20）。空树（空链接）赋以索引 0。单个结点的树（有两个空链接的结点）赋以索引 1，以此类推。

我们递归地计算与每个子树对应的索引。然后创建一个关键字，使得具有相同子树的任何结点都将有相同的索引，并在填充 DAG 的边（子树）链接之后返回那个索引。

```
int compressR(link h)
  { int l, r, t;
    if (h == NULL) return 0;
    l = compressR(h->l);
    r = compressR(h->r);
    t = STindex(l*Vmax + r);
    adj[t].l = l; adj[t].r = r;
    return t;
  }
```

一个等价的应用是将查看线索关键字看作对应布尔函数真值表中函数为真的某些行（见练习 19.87 ~ 19.90）。二叉 DAG 是计算此函数的经济型电路的一个模型。在这个应用中，二叉 DAG 被称为二叉决策图（binary decision diagram）。

由于这些应用的驱动，我们在以下两节转向 DAG 处理算法的研究。不仅这些算法可以得到高效且有用的 DAG 的 ADT 函数实现，而且它们还提供了对于处理有向图的深刻认识。我们将会看到，即使是 DAG 看上去比一般有向图具有更简单的结构，但是对于某些基本问题的解决并不会容易多少。

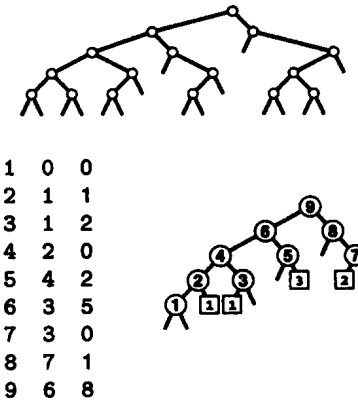


图 19-20 二叉树压缩

左下图的 9 对整数表是一个二叉 DAG（右下图）的简洁表示，而该二叉 DAG 是上图的二叉树结构的一种压缩版本。结点标记并没有显式存储在数据结构中：这个表表示了 18 条边，分别为 1-0、1-0、2-1、2-1、3-1、3-2 等，且从每个结点出发都指派了一条左边和一条右边（像二叉树那样），将每条边的源点隐式地保存在表的索引中。

只依靠树形状的算法对于 DAG 也能高效地工作。例如，假设该树是一个存在线索，其二叉关键字对应叶结点，因而它表示关键字 0000、0001、0010、0110、1100 和 1101。在线索中对于关键字 1101 的成功查找是向右移、右移、左移和右移，到一个叶结点结束。在 DAG 中，同样的查找是从 9 走到 8、到 7，再到 2，最后到 1。

练习

▷ **19.78** 定义一个适合于处理 DAG 的 ADT 接口，并构建邻接表及邻接矩阵实现。其中包含一个 ADT 函数，用于验证 DAG 中不存在环。使用 DFS 实现。

○ **19.79** 通过产生随机图，编写一个产生随机 DAG 的程序，从一个随机起点开始执行 DFS，并去掉回边（见练习 19.41）。给定 V，进行实验来确定在程序中如何设置参数使得 DAG 有 E 条边。

▷ **19.80** 对于 F_N，第 N 个斐波纳契数，对应于图 19-18 的树和 DAG 中有多少结点？

19.81 对于第 5 章中的背包模型（见图 5-17），给出对应于动态规划示例的 DAG。

○ **19.82** 为二叉 DAG 开发一个 ADT。

● 19.83　每个 DAG 都能表示为一个二叉 DAG（见性质 5.4）吗？

○ 19.84　编写一个函数，对单源点二叉 DAG 执行中序遍历。也就是说，该函数应该访问经由左边可达的所有顶点，然后访问源点，再访问经由右边可达的所有顶点。

▷ 19.85　按照图 19-20 中的风格，对于关键字 01001010 10010101 00100001 11101100 0101000100100001 00000111 01010011，给出存在线索和对应的二叉 DAG。

19.86　实现基于由一组 32 位关键字构建存在线索的一个 ADT，将它压缩为一个二叉 DAG。然后使用该数据结构来支持存在性查询。

○ 19.87　画出 4 个变量的奇偶效验真值表的 BDD，其值为 1，当且仅当值为 1 的变量的个数为奇数。

19.88　编写一个取 2^n 位真值表作为参数的函数，并返回对应的 BDD。例如，给定输入 1110001000001100，你的程序应该返回图 19-20 中的二叉 DAG 的一个表示。

19.89　编写一个取 2^n 位真值表作为参数的函数，计算出其参数变量的每种排列，并使用练习的 19.88 中的解决方案，找出导致最小 BDD 的那个排列。

● 19.90　对于不论是标准还是随机产生的各种布尔函数，进行实验研究来确定练习 19.90 中策略的有效性。

19.91　编写一个类似程序 19.5 的程序，支持公共子表达式的删除：给定一棵表示算术表达式的二叉树，计算出一个表示同一表达式的二叉 ADT，其中公共子表达式已删除。

○ 19.92　画出 2 个、3 个、4 个、5 个顶点的所有非同构的 DAG。

●● 19.93　有多少个不同的包含 V 个顶点和 E 条边的 DAG？

●●● 19.94　如果仅当两个 DAG 是非同构时才认为它们不相同，那么有多少个包含 V 个顶点和 E 条边的不同的 DAG？

19.6　拓扑排序

拓扑排序的目标是能够处理一个 DAG 中的顶点，从而使每个顶点在其指向的所有顶点之前得到处理。有两种自然的方式来定义这个基本的操作；它们本质上是等价的。两个任务都调用 $0 \sim V-1$ 的一个排列，通常放在顶点索引的数组中。

拓扑排序（重新编号）　给定一个 DAG，对其顶点重新编号，使得每条有向边从编号小的顶点指向编号大的顶点（见图 19-21）。

拓扑排序（重新排列）　给定一个 DAG，在水平线上对其顶点重新排列，使得所有有向边都从左指向右（见图 19-22）。

如图 19-22 所指示的，很容易确立重新编号与重新排列是互逆的：给定一个重新排列，可以通过对表中的第一个顶点指定标号 0，表中的第二个顶点指定标号 1，等等，来得到一个重新编号。例如，如果数组 ts 中的顶点都是拓扑有序的，那么循环在顶点索引的数组 tsI 中定义了一种重新编号。

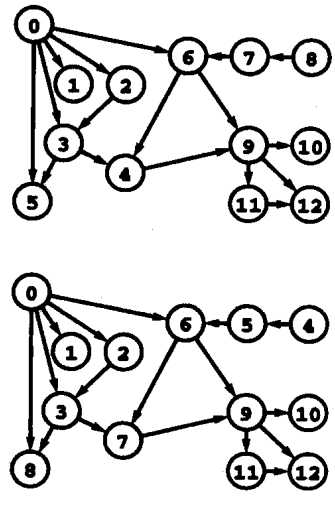
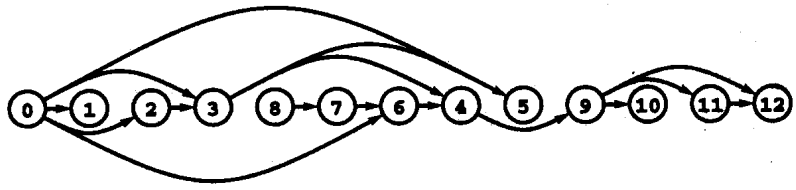


图 19-21　拓扑排序（重新编号）

给定任意 DAG（上图），拓扑排序使我们可对顶点重新编号，使得每条边从编号小的顶点指向编号大的顶点（下图）。在此例中，对顶点 4、5、7 和 8 分别重新编号为 7、8、5 和 4，如数组 tsI 中所示。有多种可能的标号都可以达到想要的结果。



	0	1	2	3	4	5	6	7	8	9	10	11	12
ts	0	1	2	3	8	7	6	4	5	9	10	11	12
tsI	0	1	2	3	7	8	6	5	4	9	10	11	12

图 19-22 拓扑排序（重新排列）

此图显示了对于图 19-21 进行拓扑排序的另一种方法。在此指定了一种重新对顶点排列的方法。而不是对其重新编号。将顶点按照数组 ts 中指定的顺序从左到右放置，就得到所有有向边都从左指向右。排列 ts 的逆为 tsI，指定了图 19-21 中所描述的重新编号。

```
for (i = 0; i < V; i++) tsI[ts[i]] = i;
```

相反，如果数组 tsI 是重新编号的，那么可由循环得到重新排列。

```
for (i = 0; i < V; i++) ts[tsI[i]] = i;
```

它将标号为 0 的顶点先放在表中，再将标号为 1 的顶点放入表中，等等。我们最常使用的术语*拓扑排序*（topological sort）是指问题的重新排列。

一般而言，由拓扑排序所产生的顶点顺序不是唯一的。例如，

```
8  7  0  1  2  3  6  4   9  10  11  12   5
0  1  2  3  8  6  4  9  10  11  12   5   7
0  2  3  8  6  4  7  5   9  10   1  11  12
8  0  7  6  2  3  4  9   5   1  11  12  10
```

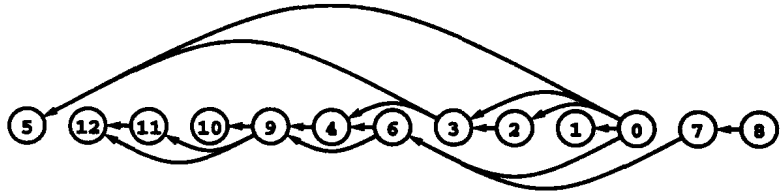
都是图 19-6 中示例 DAG 的拓扑排序（还有很多其他方式）。在调度应用问题中，只要一个任务对于另一个任务没有直接或间接的依赖关系，就会出现这种情况。因此它们执行的顺序可前可后（甚或并行执行）。可能的调度数目与这种成对的任务数成指数增长。

我们已经提到，有时将有向图中的边解释为另一种方式是很有用的：我们称一条边由 s 指向 t，含义是顶点 s “依赖于” 顶点 t。例如，顶点可以表示一本书中所定义的术语，如果 s 的定义使用了 t，则从 s 到 t 有一条边。在这种情况下，找出一种顺序，使得每个术语在另一个定义中使用之前得到定义。使用这个顺序，对应于将顶点排列在一条线上，使得边都从右指向左，即*逆拓扑排序*（reverse topological sorting）。图 19-23 说明了示例 DAG 的一种逆拓扑排序。

现在，可以看出我们已经见过逆拓扑排序的算法：就是标准的递归 DFS。在输入图是一个 DAG 时，*后序编号*（postorder numbering）将顶点按照逆拓扑排序排列。也就是说，我们对每个顶点编号作为递归 DFS 函数的最终行为。就像在程序 19.2 中 DFS 实现中的 post 数组一样。如图 19-24 所示，使用这个编号等同于对 DFS 森林中的结点按照后序遍历的顺序进行编号。按照其后序编号的顺序，取出此例中的顶点，就得到图 19-23 中的顶点编号，即 DAG 的一个逆拓扑排序。

性质 19.11 DFS 中的后序编号能得出任何 DAG 的一个逆拓扑排序。

证明 假设 s 和 t 是满足以下条件的两个顶点：即使图中存在一条有向边 s-t，s 在后序编号中也先于 t 出现。由于在赋以 s 编号时就完成了对 s 的递归 DFS，特别是，检查了边 s-t。但如果 s-t 是一个树边、下边或交叉边，则对 t 的递归 DFS 也会完成，而 t 会有一个更小的编号；然而，s-t 不能是一条回边，因为这会蕴含着一个环。此矛盾蕴含着边 s-t 不存在。 ■



	0	1	2	3	4	5	6	7	8	9	10	11	12
ts	5	12	11	10	9	4	6	3	2	1	0	7	8
tsI	10	9	8	7	5	0	6	11	12	4	3	2	1

图 19-23　逆拓扑排序

在这个示例有向图的逆拓扑排序中，边都从右指向左。通过逆排列 tsI 指定的顶点编号，就得到一个图，图中的每条边都从一个编号大的顶点指向一个编号小的顶点。

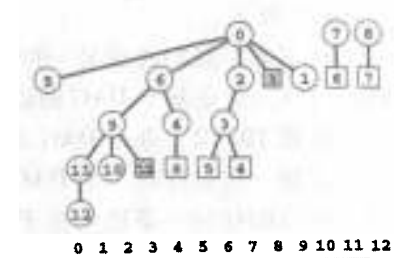
因此，我们可以很容易地对标准 DFS 进行调整以进行拓扑排序，如程序 19.6 所示。取决于应用问题，我们希望将拓扑排序的一个 ADT 函数封装起来，用后序编号填充客户提供的顶点索引数组，或者希望返回那个排列的逆，其顶点索引是按照拓扑排序的顺序。无论哪种情况，我们都希望进行前向拓扑排序或逆拓扑排序，总共需要处理四种不同情况。

总的说来，拓扑排序和逆拓扑排序之间的差别并不是关键的。如果想使它产生一种正常的拓扑排序，至少有三种方式来修改有后序编号的 DFS 就可做到。

- 在给定 DAG 的逆图上进行逆拓扑排序。
- 不是用它作为后序编号的一个索引，将顶点编号压入栈中作为递归过程的最后行为。而是在搜索完成之后，从栈中将栈顶元素弹出。它们按照拓扑排序的顺序离开堆栈。
- 按照逆序对顶点编号（从 $V-1$ 开始，并递减至 0）。

如果需要，可计算顶点编号的逆来得到拓扑排序。

这些改变可以得到一个正确的拓扑排序的证明留作练习由你来完成（见练习 19.99）。



	0	1	2	3	4	5	6	7	8	9	10	11	12
post	5	12	11	10	9	4	6	3	2	1	0	7	8

图 19-24　DAG 的 DFS 森林

有向图的 DFS 森林没有回边（指向有较大后序编号的结点的边），当且仅当该有向图是一个 DAG 图。对于图 19-21 中的 DAG，此 DFS 森林中的非树边要么是下边（有阴影的方结点），要么是交叉边（无阴影的方结点）。在对森林的后序遍历中，顶点被遇到的顺序是一个逆拓扑排序（见图 19-23），如下图所示。

程序 19.6　逆拓扑排序（邻接表）

此版本的 DFS 返回一个数组，包含 DAG 的一个顶点索引，使得每条边的源点出现在目的顶点的右边（逆拓扑排序）。例如，将 TSdfsR 的最后一行与程序 19.2 的 dfsR 的最后一行进行比较，对于理解后序编号排列的逆图计算非常有益。

```
static int cnt0;
static int pre[maxV];
void DAGts(Dag D, int ts[])
  { int v;
    cnt0 = 0;
    for (v = 0; v < D->V; v++)
      { ts[v] = -1; pre[v] = -1; }
    for (v = 0; v < D->V; v++)
      if (pre[v] == -1) TSdfsR(D, v, ts);
```

```
}
void TSdfsR(Dag D, int v, int ts[])
  { link t;
    pre[v] = 0;
    for (t = D->adj[v]; t != NULL; t = t->next)
      if (pre[t->v] == -1) TSdfsR(D, t->v, ts);
    ts[cnt0++] = v;
  }
```

对于稀疏图（用邻接表表示），要实现上一段中所列出的第一种方式，我们需要使用程序 19.1 来计算逆图。这样做将使空间使用加倍，对于大型图将变成难以承受。对于稠密图（邻接矩阵表示），如在 19.1 节中所指出的，可以在逆图上执行 DFS，在引用矩阵元素时，不使用任何额外空间或者做额外的工作，只需进行简单的行列交换即可，如程序 19.7 所示。

接下来，我们考虑另一种经典的拓扑排序方法，它更像是广度优先搜索（BFS）（见 18.7 节）。它是基于 DAG 的以下性质。

性质 19.12 每个 DAG 至少有一个源点和一个汇点。

证明 假设存在一个 DAG 没有汇点。因此，从任何顶点开始，只要沿着从那个顶点到其他顶点的任何一条边（由于没有汇点，至少会有一条边），就可以构建一条任意长的有向路径，然后，沿着由该顶点发出的另一条边，如此继续。但一旦我们有了 $V+1$ 个顶点，由鸽巢原理，必定得到一个有向环（见性质 19.6），这与假设我们有一个 DAG 相矛盾。因此，每个 DAG 至少有一个汇点。由此可得每个 DAG 也至少有一个源点：其逆图的汇点。 ■

程序 19.7 拓扑排序（邻接矩阵）

此邻接数组的 DFS 计算一个拓扑排序（不是逆拓扑排序），因为用 a[w][v] 替换掉在 DFS 中的引用 a[v][w]，就变成处理逆图（见正文）。

```
void TSdfsR(Dag D, int v, int ts[])
  { int w;
    pre[v] = 0;
    for (w = 0; w < D->V; w++)
      if (D->adj[w][v] != 0)
        if (pre[w] == -1) TSdfsR(D, w, ts);
    ts[cnt0++] = v;
  }
```

由此性质，我们可以导出一个（重新编号）的拓扑排序算法：使用最小未用标号对源点编号，然后删除该源点，使用同样算法对 DAG 其余顶点进行编号。图 19-25 是该算法在示例 DAG 上操作的轨迹。

有效地实现这个算法是数据结构设计中一个经典的练习（见第 5 部分参考文献）。首先，可能有很多源点，因而我们只需用一个队列记录这些源点（可以是任意广义队列）。第二，在删除一个源点之后，还要识别出 DAG 中其余的源点。我们用一个顶点索引的数组记录每个顶点的入度，来完成这项任务。入度为 0 的顶点是源点，因而我们可以对 DAG 进行一遍扫描来初始化这个队列（使用 DFS 或其他检查所有边的方法）。然后，我们进行如下操作，直到源点队列为空：

- 从该队列中删除一个源点，并对它进行标号。

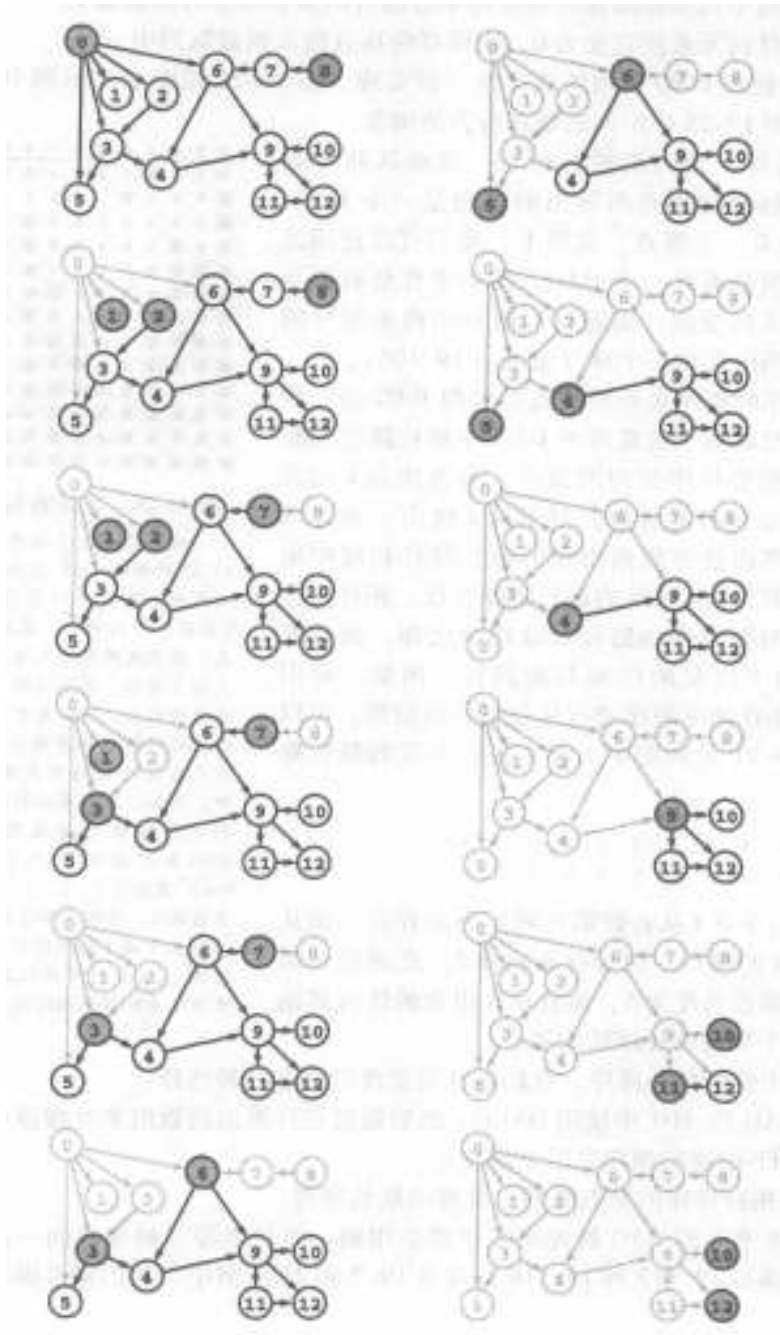


图 19-25　通过删除源点对一个 DAG 进行拓扑排序

因为 0 在这个示例图（左上图）中是一个源点（没有边指向它），因而它可首先出现在拓扑排序中。如果我们删除 0（及由它指向其他顶点的所有边），那么 1 和 2 成为结果 DAG 中的源点（左列，从上数第二个图），然后使用同一方法进行拓扑排序。此图描述了程序 19.8 的操作过程，使用 FIFO 机制从这些源点（每个图中有阴影的结点）中选择一个源点，虽然每一步中任何源点都可能被选。由图 19-26 可知，数据结构中的内容控制着算法所做的特定选择。这里所示的拓扑排序结果的结点顺序为 0 8 2 1 7 3 6 5 4 9 11 10 12。

- 将入度数组中与被删除顶点相应每条边的目的顶点对应的元素减 1。
- 如果减小任何元素使它变为 0，则将对应顶点插入到源队列中。

程序 19.8 是使用 FIFO 队列的该方法一种实现，图 19-26 说明了在示例 DAG 上操作的过程，它提供了图 19-25 中示例的动态行为的细节。

当 DAG 中的每个顶点都被标号后，源点队列才为空。因为由尚未标号的顶点所导出的子图是一个 DAG，且每个 DAG 至少有一个源点。实际上，我们可以使用此算法来检查一个图是否是一个 DAG，即如果在所有顶点被标号之前源点队列变空，必定可以推出由尚未标号的顶点所导出的子图中存在一个环（见练习 19.106）。

按照拓扑排序的顺序来处理顶点是处理 DAG 的一种基本技术。一个经典例子就是找出 DAG 中最长路径的长度。考虑按照逆拓扑排序排列的顶点，由各顶点 v 出发的最长路径的长度很容易计算：对于从 v 仅由一条边可达的各个顶点，在由这些顶点出发的最长路径长度中取最大值，再加 1 即为从 v 出发的最长路径长度。拓扑排序可以确保处理 v 时所有这些路径长度均为已知，而且此后不会再找到由 v 出发的任何其他路径。例如，对图 19-23 所示的逆拓扑排序顺序进行从左到右的扫描，可以很快计算出图 19-21 示例图中由每个顶点出发的最长路径长度的表。

```
5  12  11  10  9  4  6  3  2  1  0  7  8
0   0   1   0  2  3  4  4  5  0  6  5  6
```

例如，6 对应于 0（从右数第三列）表示存在一条从 0 开始的长度为 6 的路径，因为存在边 0-2，此前找到的从 2 出发的最长路径长度为 5，而且从 0 出发的任何其他边都不能指向一个导致更长路径的顶点。

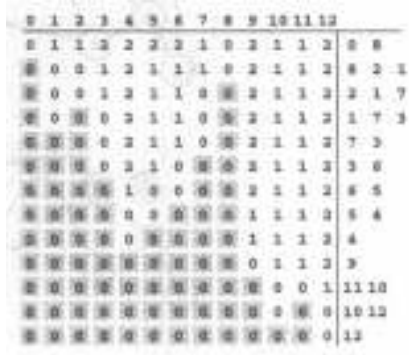


图 19-26　入度表和队列中的内容

该序列描述了程序 19.8 在对应图 19-25 的示例 DAG 上执行程序 19.8 的过程中，入度表（左边）和源点队列（右边）中的内容。在给定的任何时间点，源点队列包含入度为 0 的顶点。从上向下读取，我们从源点队列中删除最左边的结点，并使离开该结点的每条边所对应的结点的度减小 1，同时将所对应元素变为 0 的顶点添加到源点队列中。例如，表的第二行反映了从源点队列中删除 0 后的结果，然后（由于 DAG 含有边 0-1、0-2、0-3、0-5 和 0-6）使对应 1、2、3、5 和 6 的入度元素值减 1，并将 2 和 1 添加到源点队列中（由于减 1 后使得它们的入度元素值为 0）。从上到下读取源点队列中的最左边元素就得到此图的拓扑排序。

只要在应用中使用拓扑排序，我们在开发实现时都有几种选择：

- 在一个 DAG 的 ADT 中使用 DAGts，然后通过它计算出的数组来处理顶点。
- 在一个 DFS 的递归调用之后处理顶点。
- 按照基于拓扑排序的源点队列，处理出队的顶点。

这些方法在本书中的 DAG 处理实现中都会用到，而且需要了解重要的一点：这些方法都是等价的。在练习 19.113 和 19.116 以及第 19.7 和 21.4 节中，我们将考虑拓扑排序的其他应用。

程序 19.8　基于源点队列的拓扑排序

此实现维持一个源点队列，并使用一个表来记录尚未从队列中删除的顶点所导出的 DAG 中每个顶点的入度。在从队列中删除一个源点时，就使其邻接表中每个顶点所对应的入度元素值减 1（并将元素值变为 0 的任何顶点插入到队列中）。顶点按照拓扑排序的顺序出队。

```
#include "QUEUE.h"
static int in[maxV];
void DAGts(Dag D, int ts[])
  { int i, v; link t;
    for (v = 0; v < D->V; v++)
      { in[v] = 0; ts[v] = -1; }
    for (v = 0; v < D->V; v++)
      for (t = D->adj[v]; t != NULL; t = t->next)
        in[t->v]++;
    QUEUEinit(D->V);
    for (v = 0; v < D->V; v++)
      if (in[v] == 0) QUEUEput(v);
    for (i = 0; !QUEUEempty(); i++)
      {
        ts[i] = (v = QUEUEget());
        for (t = D->adj[v]; t != NULL; t = t->next)
          if (--in[t->v] == 0) QUEUEput(t->v);
      }
  }
```

练习

▷ **19.95**　在练习 19.78 的 DAG 的 ADT 中添加一个拓扑排序函数，再添加一个函数，用于检查给定的 DAG 顶点的排列是否是此 DAG 的一个拓扑排序。

19.96　对于图 19-6 描述的 DAG，有多少种不同的拓扑排序？

▷ **19.97**　给出由在以下 DAG 上执行标准邻接表 DFS（且后序编号）所得的 DFS 森林和逆拓扑排序。

```
3-7  1-4  7-8  0-5  5-2  3-8  2-9  0-6  4-9  2-6  6-4  4-3  2-3
```

○ **19.98**　给出由在以下 DAG 上执行标准邻接表表示所得的 DFS 森林和拓扑排序。

```
3-7  1-4  7-8  0-5  5-2  3-8  2-9  0-6  4-9  2-6  6-4  4-3  2-3
```

● **19.99**　为了能够计算出拓扑排序而不是逆拓扑排序，对于用后序编号的 DFS，正文中给出了三条修改建议，证明其中每个建议的正确性。

▷ **19.100**　给出在以下 DAG 上执行带有隐式转置的标准邻接矩阵 DFS（且后序编号）所得的 DFS 森林和拓扑排序（见程序 19.7）。

```
3-7  1-4  7-8  0-5  5-2  3-8  2-9  0-6  4-9  2-6  6-4  4-3  2-3
```

● **19.101**　给定一个 DAG，无论如何选择与每个顶点所邻接顶点的顺序，都存在不能由应用基于 DFS 的算法而得的拓扑排序吗？证明你的结论。

▷ **19.102**　按照图 19-26 的样式，显示对 DAG 进行拓扑排序的过程，使用源点队列算法（程序 19.8）。

```
3-7  1-4  7-8  0-5  5-2  3-8  2-9  0-6  4-9  2-6  6-4  4-3  2-3
```

▷ **19.103**　如果在图 19-25 中描述的示例所使用的数据结构是栈而不是队列，给出拓扑排序的结果。

● **19.104**　给定一个 DAG，无论使用何种队列，都存在不能由应用源点队列的算法而得的拓扑排序吗？证明你的结论。

19.105　使用广义队列修改源点队列的拓扑排序算法。分别针对 LIFO 队列、栈和随机队列

使用修改过的算法。

▷ 19.106 使用程序 19.8 为 ADT 函数提供一个实现，用于验证 DAG 中不存在环（见练习 19.78）。

○ 19.107 将源点队列的拓扑排序算法变成用于逆拓扑排序的汇点队列算法。

19.108 对于给定的 DAG，编写一个产生其所有可能拓扑排序顺序的程序，或者如果其个数超过某个作为参量的界限，则打印出这个数。

19.109 编写一个将 V 个顶点、E 条边的有向图转换成 DAG 的程序，可以通过执行基于 DFS 拓扑排序，并改变所遇到的回边的方向来完成。证明这种策略总是能产生一个 DAG。

●● 19.110 编写一个程序，使得产生有 V 个顶点、E 条边的每个 DAG 是等概率的（见练习 17.69）。

19.111 给出一个 DAG 的顶点只可能有一种拓扑排序的充分必要条件。

19.112 进行实验研究，对于各种类型的 DAG（见练习 19.2、19.79、19.109 和 19.110），对本节中给出的拓扑排序算法进行比较。并像练习 19.11（针对低密度图）和 19.12（针对高密度图）中所述测试你的程序。

▷ 19.113 修改程序 19.8，使得它能计算从 DAG 中的任何源点到其他顶点的不同简单路径的数目。

○ 19.114 编写一个程序得出表示算术表达式（见图 19-19）的 DAG。使用邻接表图 ADT，对它进行扩展使其包含对应顶点的 double 类型（来存放其值）。假设与叶结点对应的值已经确定。

○ 19.115 描述一组算术表达式，具有性质：表达式树的规模比对应的 DAG 的规模大指数级（从而练习 19.114 中的程序对应 DAG 的运行时间与对应此树运行时间的对数成正比）。

○ 19.116 编写一个找出 DAG 中的最长简单有向路径的程序，所需时间与 V 成正比。使用你的程序实现一个 ADT 函数，用于找出一个给定 DAG 中的哈密顿回路（如果存在的话）。

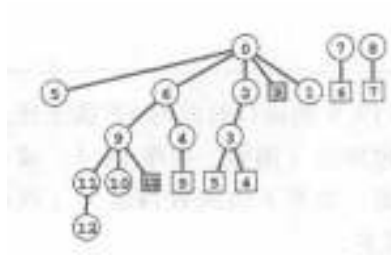
19.7 有向无环图中的可达性

最后我们研究计算 DAG 的传递闭包的问题。是否能够为 DAG 开发一个算法，使之比 19.3 中为一般有向图开发的算法更高效？

拓扑排序中的任何方法都可用作 DAG 传递闭包算法的基础，如下：继续以逆拓扑排序的顺序处理顶点，对于每个顶点，由对应于其邻接顶点的行计算出它的可达性向量。逆拓扑排序保证所有这些行都已经被计算过。总而言之，我们要检查该向量的 V 个元素中的每个元素，对应着 E 条边的每个目的顶点。总运行时间与 VE 成正比。虽然它易于实现，但较之于一般有向图，这种针对 DAG 的方法不会更高效。

在使用一个标准的 DFS 进行拓扑排序时（见程序 19.7），对于某些 DAG 而言，可以改进其性能，如程序 19.9 所示。因为在 DAG 中不存在环，因而在任何 DFS 中也没有回边。更重要的是，交叉边和下边都指向 DFS 已经完成的结点。利用这一点，开发一个计算由给定起始顶点可达的所有顶点的递归函数，但是（DFS 中就是这样）我们对可达集合中已经计算的那些顶点并不做递归调用。在这种情况下，可达顶点被表示为传递闭包中的一行，而且递归函数要对与其邻接边相关的所有行完成逻辑或操作。对于树边，进行递归调用来计算那一行；对于交叉边，则可以跳过递归调用，因为我们知道那一行已经由前一个递归调用计算得出；对于下边，则可以跳过整个计算，因为增加的任何可达顶点都已经在目的顶点的可达集合中计算得出（在 DFS 树的更低位置且更早出现）。

使用这种版本的 DFS 可以表述为使用动态规划来计算传递闭包，因为我们利用了已经计算得到的结果来避免进行不必要的递归调用。图 19-27 描述了计算图 19-6 中示例 DAG 的传递闭包的过程。



```
 5:  0 0 0 0 0 1 0 0 0 0 0 0
12:  0 0 0 0 0 0 0 0 0 0 0 1
11:  0 0 0 0 0 0 0 0 0 0 1 1
10:  0 0 0 0 0 0 0 0 0 1 0 0
 9:  0 0 0 0 0 0 0 0 1 1 1 1
 4:  0 0 0 0 1 0 0 0 0 1 1 1
 6:  0 0 0 0 1 0 1 0 0 1 1 1
 3:  0 0 0 1 1 1 0 0 0 1 1 1
 2:  0 0 1 1 1 1 0 0 0 1 1 1
 1:  0 1 0 0 0 0 0 0 0 0 0 0
 0:  1 1 1 1 1 1 1 0 0 1 1 1
 7:  0 0 0 0 1 0 1 1 0 1 1 1
 8:  0 0 0 0 1 0 1 1 1 1 1 1
```

图 19-27　DAG 的传递闭包

此行向量序列是图 19-21 中 DAG 的传递闭包，在此用逆拓扑排序的顺序创建行，这个计算作为递归 DFS 中的最后操作（见程序 19.9）。每一行都是对邻接顶点所在的行做逻辑或操作。例如，要计算对应 0 的行，需要对 5、2、1 和 6 行做逻辑或操作（并对 0 本身设置 1），因为边 0-5、0-2、0-1 和 0-6 可将我们带到由 0 指向的顶点可达的任何顶点。可以忽略下边，因为它们并不增加新的信息。例如，我们忽略从 0 到 3 的边，因为由 3 可达的顶点已经在 2 对应的行中计算得出。

性质 19.13　利用动态规划和 DFS，对于 DAG 的抽象传递闭包，可以支持常量的查询时间，所用空间与 V^2 成正比，预处理时间与 $V^2 + VX$ 成正比（计算传递闭包），其中 X 为 DFS 森林中交叉边的数目。

证明　证明可由对程序 19.9 中的递归函数使用归纳法得出。我们按照逆拓扑排序的顺序访问顶点。每条边指向一个已经计算出其所有可达顶点的顶点，而且可以通过将与每条边的目的顶点相关的可达顶点集合并在一起，计算出任何顶点的可达顶点集。对于邻接矩阵中的特定行取逻辑或操作就可完成这一合并。对于每条树边和每条交叉边，我们要访问规模为 V 的一行。这里没有回边，而且可以忽略下边，因为对于它们到达的任何顶点，在先前的搜索中处理这两类结点的任何祖先时就已计算得出。 ■

程序 19.9　DAG 的传递闭包

此代码使用单个 DFS 计算出 DAG 的传递闭包，在 DFS 树中，从每个孩子结点的可达顶点开始，递归地计算出由其每个顶点可达的顶点。对于树边做递归调用，使用前面用于交叉边所计算出的值，并忽略那些下边。

```
void DAGtc(Dag D)
  { int v;
    D->tc = MATRIXint(D->V, D->V, 0);
    for (v = 0; v < D->V; v++) pre[v] = -1;
    for (v = 0; v < D->V; v++)
      if (pre[v] == -1) TCdfsR(D, EDGE(v, v));
  }
void TCdfsR(Dag D, Edge e)
  { int u, i, v = e.w;
    pre[v] = cnt++;
    for (u = 0; u < D->V; u++)
      if (D->adj[v][u] != 0)
        {
```

```
            D->tc[v][u] = 1;
            if (pre[u] > pre[v]) continue;
            if (pre[u] == -1) TCdfsR(D, EDGE(v, u));
            for (i = 0; i < D->V; i++)
              if (D->tc[u][i] == 1) D->tc[v][i] = 1;
          }
      }
  int DAGreach(Dag D, int s, int t)
    { return D->tc[s][t]; }
```

如果 DAG 没有下边（见练习 19.43），程序 19.9 的运行时间与 VE 成正比，这表明对我们在 19.3 节中所考察的一般有向图中的传递闭包算法（例如，程序 19.4）或本节一开始所描述的基于拓扑排序的方法没有改进。另一方面，如果下边的数目很大（或者等价地，交叉边数目很小），程序 19.9 将比这些方法都快得多。

寻找一个计算稠密 DAG 的传递闭包的最优算法（保证在与 V^2 成正比的时间内完成）的问题仍然是未解问题。已知的最坏情况性能界限为 VE。然而，使用对于大量 DAG 都运行更快的算法（如程序 19.9）要比使用一个运行时间总是与 VE 成正比的算法（如程序 19.4）更好。如我们在 19.9 节所看到的，对 DAG 的这个性能改进对于计算出一般有向图的传递闭包有着直接影响。

练习

○ **19.117** 按照 19.27 的风格，显示使用程序 19.9 计算以下 DAG 的传递闭包所得的可达性向量

```
3-7  1-4  7-8  0-5  5-2  3-8  2-9  0-6  4-9  2-6  6-4  4-3  2-3
```

19.118 开发程序 19.9 的邻接矩阵 DAG 表示的版本。

○ **19.119** 开发程序 19.9 的一个版本，使用传递闭包的邻接表表示，运行时间与 $V^2 + \sum_e v(e)$ 成正比，其中和式为对 DAG 中所有边求和，$v(e)$ 为由边 e 的目的顶点可达的顶点数目。对于某些稀疏图 DAG（见练习 19.68）此开销要比 VE 小很多。

○ **19.120** 开发 DAG 的抽象传递闭包的一种 ADT 实现，使用的额外空间至多与 V 成正比（适合于巨型 DAG）。在顶点不连通时，使用拓扑排序做出快速响应，并使用源点队列实现来返回顶点连通时的路径长度。

○ **19.121** 开发一个基于汇点队列和逆拓扑排序为基础的传递闭包实现（见练习 19.107）。

○ **19.122** 在练习 19.121 的解中需要检查 DAG 中的所有边吗？还是有些边会被忽略，如 DFS 中的下边？给出一个需要检查所有边的例子，或描述所能跳过的边的特征。

19.8 有向图中的强连通分量

比起一般有向图，无向图和 DAG 都有着简单的结构，这是因为结构的对称性刻画了顶点之间的可达性关系特征：在一个无向图中，如果从 s 到 t 存在一条路径，那么我们知道从 t 到 s 也存在一条路径；在一个 DAG 中，如果从 s 到 t 存在一条有向路径，那么我们知道从 t 到 s 不存在有向路径。对于一般有向图，知道 t 由 s 可达并没有给出 s 是否由 t 可达的信息。

为了理解有向图的结构，我们考虑强连通性（strong connectivity），它具有我们寻求的对称性。如果 s 和 t 是强连通的（其中顶点相互可达），那么，按照定义，t 和 s 也是强连通的。

如 19.1 节所讨论的，这种对称性蕴含着有向图的顶点可划分为强分量（string component），这些强分量是由相互可达的顶点组成。在这一节里，我们讨论寻找一个有向图中强分量的三个算法。

我们使用与无向图的通用图搜索算法中有关连通性相同的接口（见程序 18.3）。算法的目标是对在顶点索引数组中的每个顶点指定分量编号，使用 0、1、...对强分量编号。最大编号为强分量的个数，而且我们可以使用分量数，在常量时间内检查两个顶点是否是在同一个连通分量中。

解决这个问题的蛮力算法很容易开发。使用一个抽象传递闭包 ADT，检查每对顶点 s 和 t，看看 t 是否由 s 可达，且 s 是否由 t 可达。将这样的顶点对定义为无向图中的一条边：此图的连通分量就是有向图的强分量。这个算法易于描述和实现，且它的运行时间由抽象传递闭包实现的开销所决定，正如性质 19.10 中所述。

本节所考虑的算法是现代算法设计的胜利，能在线性时间内找出任何图的强分量，它要比蛮力算法快 V 倍。对于 100 个顶点，这些算法将比蛮力算法快 100 倍；对于 1 000 个顶点，这些算法将比蛮力算法快 1 000 倍；对于涉及数十亿顶点的问题也可以解决。这个问题充分体现了良好算法设计的作用。正是由此促使许多人们对图算法做更深入的研究。对于一个重要的实际问题，有一个能够降低数十亿资源使用的方法，还能用在其他什么地方吗？

了解这个问题的历史是很有益的（见第 5 部分参考文献）。在 20 世纪 50 年代和 60 年代，数学家和计算机科学家开始热衷于研究图算法，当时算法自身的分析也是一个发展的研究领域。各类图算法都需要考虑，还有正在发展的计算机系统和语言，另外对有效地执行计算的理解也在发展，这些留下很多未解的困难问题。随着计算机科学家开始理解了算法分析的基本原理，他们也对哪些图问题可以高效求解，哪些问题没有高效解决方案开始理解，然后为那类存在高效解决方案的问题开发更高效的算法。实际上，在 1972 年，R. Tarjan 提出了强连通问题以及其他问题的线性时间的算法，同年，R. Karp 对旅行商问题和很多其他图问题的难解性进行了论述。Tarjan 的算法多年来一直被认为是高级算法分析课程中的主要内容，因为它使用简单数据结构解决了一个重要的实际问题。在 20 世纪 80 年代，R. Kosaraju 提出了这个问题的一种新观点，并开发了一种新的解决方案；人们后来认识到有一篇描述同样方法的文章早在 1972 年就出现在俄罗斯科学文献中。随后，在 1999 年，对于 20 世纪 60 年代所最先尝试的方法，H. Gabow 找到了一种简单实现，给出了此问题的第三种线性时间的算法。

这个故事的关键并不只是困难的图处理问题可以有简单解决方案，而是我们使用的抽象（DFS 和邻接表）比我们原先认识的更为强大。随着我们更为熟悉地使用这些抽象和类似工具，我们应该不会惊讶于发现其他重要的图问题的简单解决方案。对于很多其他重要的图算法，研究人员仍在寻求像这些算法一样的简单实现。很多这样的算法仍然有待于人们去发现。

Kosaraju 的方法解释和实现都很简单。要找出一个图的强分量，首先在其逆图上运行 DFS，计算出按照后序编号所定义的顶点排序顺序。（如果有向图是 DAG，这个过程就是一个拓扑排序）。然后，在图上再次运行 DFS，但是找出搜索的下一个顶点（可在最外层，也可在递归搜索函数返回给顶层搜索函数时调用递归搜索函数），要使用具有最大后序编号的未被访问的顶点。

此算法的神奇之处在于，以这种方法按照拓扑排序的顺序来检查未被访问的顶点时，

DFS 森林中的树定义了无向图中的连通分量，即两个顶点在同一强分量中，当且仅当它们属于此森林中的同一棵树。图 19-28 显示了我们示例中这一点就成立，稍后我们对此进行证明。因此，如同无向图的做法，我们也可以指定分量编号，在每次递归函数返回到顶层搜索函数时对编号加 1。程序 19.10 是该方法的一个完整实现。

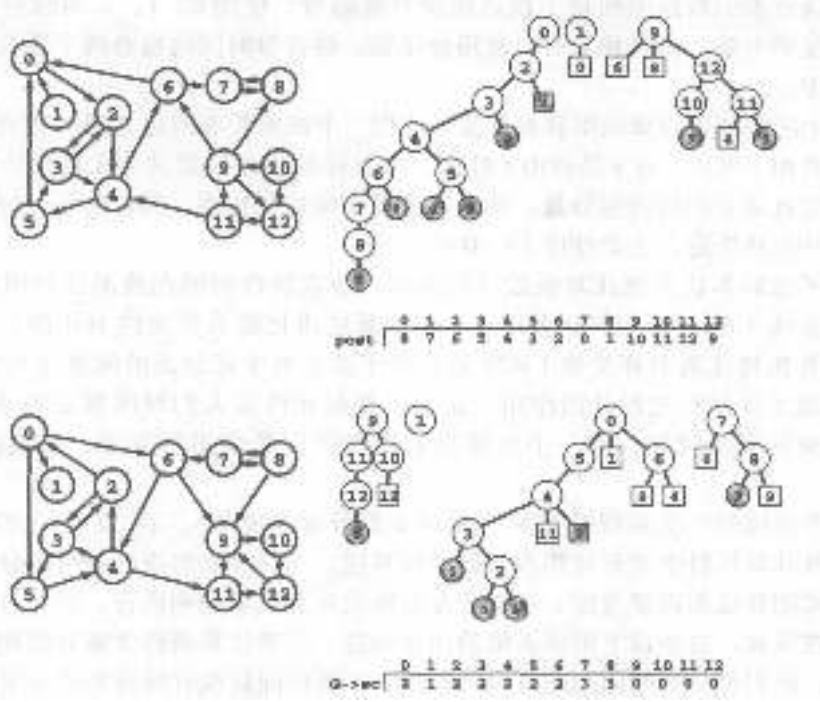


图 19-28　计算强分量（Kosaraju 算法）

要计算出左下图有向图的强分量，我们首先对它的逆图执行 DFS（左上图），计算出一个后序数组，给出的顶点编号是按照递归 DFS 完成的顺序编号（上图）。此顺序等价于对 DFS 森林进行后序遍历（右上图）。然后使用这个顺序的逆序对原有向图执行 DFS（下图）。首先我们检查由 9 可达的所有顶点，然后从右向左扫描数组，发现 1 是最右边未被访问的顶点，因而对 1 进行递归调用，如此继续。由此过程所得的 DFS 森林中的树定义了强分量：每棵树中的所有顶点在顶点索引数组 id 中都有相同的值（下图）。

程序 19.10　强分量（Kosaraju 算法）

此实现找出邻接表表示的有向图的强分量。如同 18.5 节求解无向图的连通性问题的方法，此方法对顶点索引的数组 sc 进行设置，使得对应任何顶点对的元素值如果相等，当且仅当它们在同一个强分量中。

首先，我们构建逆图，并执行 DFS 来计算一个后序排序。接下来，在原有向图上执行 DFS，并在调用递归函数的搜索循环中使用由第一次 DFS 得到的后序的逆序。第二个 DFS 中的每次递归调用将访问一个强分量中的所有顶点。

```
static int post[maxV], postR[maxV];
static int cnt0, cnt1;
void SCdfsR(Graph G, int w)
  { link t;
    G->sc[w] = cnt1;
    for (t = G->adj[w]; t != NULL; t = t->next)
      if (G->sc[t->v] == -1) SCdfsR(G, t->v);
    post[cnt0++] = w;
```

```
  }
int GRAPHsc(Graph G)
  { int v; Graph R;
    R = GRAPHreverse(G);
    cnt0 = 0; cnt1 = 0;
    for (v = 0; v < G->V; v++) R->sc[v] = -1;
    for (v = 0; v < G->V; v++)
      if (R->sc[v] == -1) SCdfsR(R, v);
    cnt0 = 0; cnt1 = 0;
    for (v = 0; v < G->V; v++) G->sc[v] = -1;
    for (v = 0; v < G->V; v++) postR[v] = post[v];
    for (v = G->V-1; v >=0; v--)
      if (G->sc[postR[v]] == -1)
        { SCdfsR(G, postR[v]); cnt1++; }
    GRAPHdestroy(R);
    return cnt1;
  }
int GRAPHstrongreach(Graph G, int s, int t)
  { return G->sc[s] == G->sc[t]; }
```

性质 19.14　Kosaraju 的方法可在线性时间和线性空间内找出一个图的强分量。

证明　方法由对两个 DFS 过程稍作修改组成，因此与以往一样，对于稠密图，运行时间与 V^2 成正比，对于稀疏图运行时间与 $V+E$ 成正比（使用邻接表表示）。要证明它正确地计算出强分量，必须证明：两个顶点 s 和 t 在第二次搜索的 DFS 森林中的同一个树中，当且仅当它们是相互可达的。

如果 s 和 t 相互可达，它们必定会在同一个 DFS 树中，因为当访问这两个顶点的前一个时，第二个未被访问且由第一个顶点可达，因而会在对根结点的递归调用结束之前被访问。

证明另一方面，假设 s 和 t 在同一个树中，令 r 是树的根结点。由 r 可达 s（通过树边的一条有向路径）的事实蕴含着在逆有向图中存在从 s 到 r 的一条有向路径。现在，证明的关键是在逆有向图中必定也存在从 r 到 s 的一条路径，因为 r 的后序编号比 s 大（由于 r 是这两个顶点均未被访问时第二次 DFS 中第一个被选的结点）；而且从 s 到 r 存在一条路径：如果从 r 到 s 不存在路径，那么在逆图中从 s 到 r 的路径将使 s 有更大的后序编号。因此，在有向图和它的逆图中，从 s 到 r 和从 r 到 s 分别存在有向路径：s 和 r 是强连通的。同理可证 t 和 r 也是强连通的，因此 s 和 t 是强连通的。 ■

对于邻接矩阵有向图表示的 Kosaraju 算法的实现甚至要比程序 19.10 还要简单，因为我们并不需要显式地计算出逆图；这个问题留作练习（见练习 19.128）。

程序 19.10 被封装为一个 ADT，表示强可达性问题的一个最优解，与第 18 章的连通性解决方案类似。在 19.9 节中，我们考察将这个解扩展以计算传递闭包以及求解有向图的可达性问题（抽象传递闭包）。

然而，我们首先考虑 Tarjan 算法和 Gabow 算法，这些都是天才的方法，只要求对基本 DFS 过程做一点简单修改。较之于 Kosaraju 算法，它们更为可取，因为其中仅对图用了一遍 DFS，而且它们并不要求计算稀疏图的逆图。

Tarjan 算法类似于我们在第 17 章中为寻找无向图中的桥所讨论的程序（见程序 18.7）。该方法基于我们在其他环境下所做的两点观察。首先，以逆拓扑排序的顺序考虑顶点，从而在到达某个顶点的递归函数的末尾时，知道将不会遇到同一个强分量中任何其他顶点（因

为由该顶点可达的所有顶点都已被处理完)。其次，树中的回链接为从一个顶点到另一个顶点提供了第二条路径，并将强分量连在一起。

递归 DFS 函数使用了和程序 18.7 一样的计算来找出由每个顶点的任何子孙可达的最大顶点（通过一条回边)。它还使用了一个顶点索引的数组来记录强分量，以及一个栈来记录当前搜索路径。它在进入递归函数时将该顶点名压入栈中，而在访问了每个强分量的最后一个成员之后，将它们从栈中弹出，并赋以分量编号。算法就是基于在递归过程结束时，利用一个简单测试来识别出这个时刻（通过从每个结点的所有子孙的一个上链接来记录最大的可达祖先)，由此可知，进入此递归调用以来遇到的所有顶点（除了已经分配给某个分量的顶点以外）都属于同一个强分量。

程序 19.11 的实现是该算法的一个简洁而完整的描述，对刚才给出的简明轮廓补充了相关细节。图 19-29 对于图 19-1 中的示例有向图显示了算法的操作过程。

性质 19.15 *Tarjan 算法在线性时间内找出一个有向图的强分量。*

证明（大纲） 如果顶点 s 在 DFS 树中没有子孙或上链接，或者它在 DFS 树中有一个子孙，带有一条指向 s 的链接，且没有子孙带有指向树中更高位置的上链接，那么它和它的所有子孙（除了那些满足同一性质的顶点及其子孙之外）构成一个强分量。为证明这一点，需要注意的是 s 的每个不满足所述性质的子孙 t 也有某个子孙，在树中有一条上链接指向比 t 还高的位置。树中存在从 s 到 t 的向下路径，而且我们可以找出一条从 t 到 s 的路径：从 t 向下走到某个顶点，该顶点有一条上链接，到达已经经过的 t，然后继续由该顶点做同样的处理，直至到达 s。

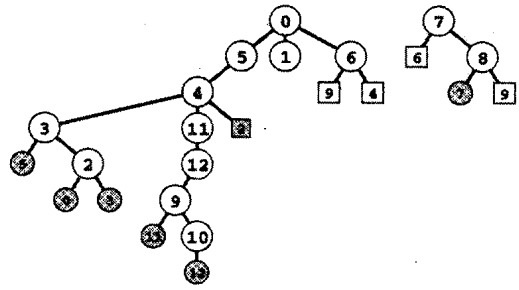
与以往一样，该方法是线性时间，因为它是由将几个常量时间的操作添加到标准 DFS 所组成的。 ■

程序 19.11 强分量（Tarjan 算法）

使用递归 DFS 函数的这一实现，标准邻接表有向图 DFS 将得到强分量，按照我们的约定，这在顶点索引的数组 sc 中确定。

我们使用栈 s（带有栈指针 N）来存放每个顶点，直到确定降至到栈顶某个位置的所有顶点都属于同一个强分量为止。顶点索引的数组 low 记录由每个结点通过一系列下链接后跟一个上链接可达的最小前序编号（见正文)。

```
void SCdfsR(Graph G, int w)
  { link t; int v, min;
    pre[w] = cnt0++; low[w] = pre[w]; min = low[w];
    s[N++] = w;
    for (t = G->adj[w]; t != NULL; t = t->next)
      {
        if (pre[t->v] == -1) SCdfsR(G, t->v);
        if (low[t->v] < min) min = low[t->v];
      }
    if (min < low[w]) { low[w] = min; return; }
    do
      { G->sc[(v = s[--N])] = cnt1; low[v] = G->V; }
    while (s[N] != w);
    cnt1++;
  }
```



	0	1	2	3	4	5	6	7	8	9	10	11	12
pre	0	9	4	3	2	1	10	11	12	7	8	5	6
low	0	9	0	0	0	0	0	11	11	5	6	5	5
G->sc	2	1	2	2	2	2	2	3	3	0	0	0	0

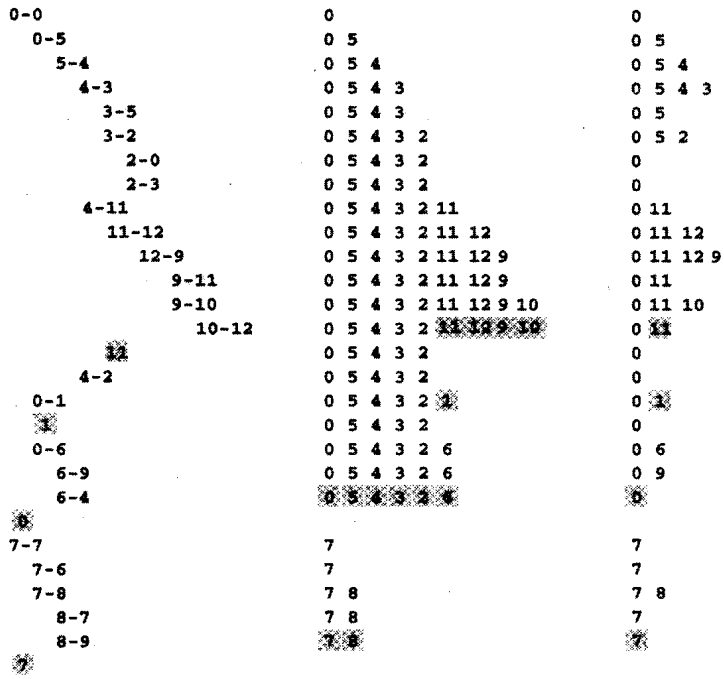


图 19-29　计算强分量（Tarjan 算法和 Gabow 算法）

Tarjan 算法基于递归 DFS，扩展从而将顶点压入栈中。它使用辅助数组 pre 和 low（中图），在一个顶点索引数组 G->sc 中计算出每个顶点的一个分量索引。示例图的 DFS 树显示在上图，边的轨迹显示在左下图。下图的中图中是主栈：由树边所达的顶点将压入此栈。使用一个 DFS 按照逆拓扑排序的顺序考虑这些顶点，对于每个 v，计算出由祖先（low [v]）通过一条回链接可达的最大点。如果顶点 v 有 pre [v] = low [v]（顶点 11、1、0 和 7 满足此条件），那么将它及其上方的所有顶点（加阴影）弹出，并为它们均赋予下一个分量编号。

在 Gabow 算法中，我们将顶点压入主栈中，就像在 Tarjan 算法中那样，但还需对搜索路径上已经知道在不同强分量中的顶点维护第二个栈（右下图），每条回边的目的顶点之后的所有顶点弹出。在完成对 v 的处理时，即 v 位于第二个栈的栈顶时（加阴影），可知主栈中 v 以上的所有顶点都在同一个强分量中。

程序 19.12 强分量（Gabow 算法）

该程序执行与程序 19.11 相同的计算，但使用第二个栈 path，而非顶点索引的数组 low 来确定何时从主栈中弹出每个强分量中的顶点（见正文）。

```
void SCdfsR(Graph G, int w)
  { link t; int v;
    pre[w] = cnt0++;
    s[N++] = w; path[p++] = w;
    for (t = G->adj[w]; t != NULL; t = t->next)
      if (pre[t->v] == -1) SCdfsR(G, t->v);
      else if (G->sc[t->v] == -1)
            while (pre[path[p-1]] > pre[t->v]) p--;
    if (path[p-1] != w) return; else p--;
    do G->sc[s[--N]] = cnt1; while (s[N] != w);
    cnt1++;
  }
```

1999 年 Gabow 发现了程序 19.12 的 Tarjan 算法版本。此算法也像 Tarjan 算法那样维护了一个顶点栈，但它使用第二个栈（而不是顶点索引的前序编号的数组）来确定何时从主栈中弹出每个强分量中的顶点。第二个栈包含搜索路径中的顶点。当一条回边显示出这样的一个顶点序列都属于同一强分量时，则弹栈只留下回边的目的顶点，该顶点比其他任何顶点距树的根结点更近一些。处理完每个顶点的所有边之后（对树边进行递归调用，弹出回边的路径栈，并忽略下边），我们查看当前顶点是否是在路径栈顶。如果是，该顶点及主栈中所有在其上的顶点构成一个强分量，将这些顶点弹出，并将下一个强分量编号赋给这些顶点，这一点与 Tarjan 算法做法一样。

图 19-29 中示例也显示了第二个栈的内容。因此，这个图也展示了 Gabow 算法的操作过程。

性质 19.16 *Gabow 算法可在线性时间内找出一个有向图的强分量。*

证明 对以上的讨论形式化，并证明其所依赖的栈中内容之间的关系对于擅长数学的读者是一个有益的练习（见练习 19.136）。如常，该方法是线性的，因为它由对标准 DFS 添加几个常量时间的操作所组成。■

本节所讨论的强分量算法都很奇妙，而且具有不易觉察的简单性。我们已经考虑了所有三种算法，因为这些算法证实了基础数据结构和经过精心设计的递归程序的强大作用。从实用的观点来看，所有算法的运行时间与有向图中的边数成正比，性能上的差异可能与实现细节有关。例如，下推栈 ADT 操作构成了 Tarjan 算法和 Gabow 算法的内循环。实现中使用了显式编码的栈；使用栈 ADT 的实现可能会慢一些。Kosaraju 算法是这三种中最简单的算法，但它有点不足（对于稀疏图），需要对边进行三遍处理（一遍是构造逆图，另外执行两遍 DFS）。

接下来，我们考虑计算强分量的关键应用：构建有向图的一种高效的可达性 ADT（抽象传递闭包）。

练习

▷ **19.123** 描述在使用 Kosaraju 算法来找出 DAG 的强分量时会出现的情况。

▷ **19.124** 描述在使用 Kosaraju 算法来找出只由单个环组成的有向图的强分量时会出现的情况。

●● **19.125** 通过使用 19.4 节中提到的三种技术之一，在进行拓扑排序时避免计算逆图，在 Kosaraju 方法的邻接表版本（程序 19.10）中，使用其中的某种技术可以避免计算有向图的

逆图吗？对于每种技术，要么给出证明它可行，要么给出一个反例证明它不行。

○ **19.126**　按照图 19-28 的风格，显示使用 Kosaraju 算法来计算图 19-5 中有向图的逆图的强分量时，所得 DFS 森林和辅助顶点索引数组中的内容。（应该得到相同的强分量。）

19.127　按照图 19-28 的风格，显示使用 Kosaraju 算法来计算如下有向图的强分量时，所得 DFS 森林和辅助顶点索引数组中的内容。

```
3-7  1-4  7-8  0-5  5-2  3-8  2-9  0-6  4-9  2-6  6-4
```

○ **19.128**　实现使用邻接矩阵表示的 ADT 来找出有向图的强分量的 Kosaraju 算法。不需要显式地计算出逆图。提示：考虑使用两种不同的递归 DFS 函数。

▷ **19.129**　描述在使用 Tarjan 算法来找出 DAG 的强分量时会出现的情况。

▷ **19.130**　描述在使用 Tarjan 算法来找出只由单个环组成的有向图的强分量时会出现的情况。

○ **19.131**　按照图 19-29 的风格，显示使用 Tarjan 算法来计算图 19-5 中的有向图的逆图的强分量时，所得 DFS 森林、算法执行中栈中内容以及辅助顶点索引数组中的最终内容。（应该得到相同的强分量。）

19.132　按照图 19-29 的风格，显示使用 Tarjan 算法来计算如下有向图的强分量时，所得 DFS 森林、算法执行中栈中内容以及辅助顶点索引数组中的最终内容。

```
3-7  1-4  7-8  0-5  5-2  3-8  2-9  0-6  4-9  2-6  6-4
```

○ **19.133**　修改程序 19.11 中的 Tarjan 算法实现和程序 19.12 中的 Gabow 算法实现，使其使用观察哨来避免显式地检查交叉链。

19.134　修改程序 19.11 中的 Tarjan 算法实现和程序 19.12 中的 Gabow 算法实现，使其使用栈 ADT。

19.135　按照图 19-29 的风格，显示使用 Gabow 算法来计算如下有向图的强分量时，所得 DFS 森林、算法执行中两个栈中的内容以及辅助顶点索引数组中的最终内容。

```
3-7  1-4  7-8  0-5  5-2  3-8  2-9  0-6  4-9  2-6  6-4
```

● **19.136**　给出性质 19.16 的完整证明。

○ **19.137**　开发 Gabow 算法的一个版本，找出无向图中桥及边连通分量。

● **19.138**　开发 Gabow 算法的一个版本，找出无向图中关结点及双连通分量。

19.139　根据表 18-1 的精神，建立一个表来研究随机有向图的强连通性（见表 19-2）。设 S 是最大强分量的顶点数。记录 S 的规模，并研究以下四类边的百分比：连接 S 中两个顶点的边；指向 S 之外的那些边；指向 S 内的那些边；连接不在 S 中的两个顶点的边。

19.140　进行实验研究，对于各种类型有向图（见练习 19.11 ~ 18），对使用本节开始所描述的蛮力法、Tarjan 算法、Gabow 算法计算强分量进行比较。

●●● **19.141**　开发一个强 2 连通性的线性时间算法：确定是否一个强连通的有向图具有以下性质：在删除其任何一个顶点（及所有它的依附边）之后仍然是强连通的。

19.9　再论传递闭包

将前两节的结果放在一起，我们可以开发出求解有向图的抽象传递闭包问题的一个算法。虽然在最坏情况下，该算法对基于 DFS 的解决方案没有改进，但在很多情况下确实会提供一个最优解。

该算法基于对有向图进行预处理来构建后者的核心 DAG（见性质 19.2）。如果核心

DAG 相对于原始有向图的规模来说较小，则算法是高效的。如果有向图是一个 DAG（因此等同其核心 DAG），或者有向图中只有几个小环，那么就不会节省很多的开销；然而，如果有向图中含有大环或大的强分量（因此有小的核心 DAG），我们可以开发最优或渐近最优的算法。为简洁起见，我们假设核心 DAG 足够小，使得可以利用邻接矩阵表示。虽然基本的概念对于大型核心 DAG 仍然是有效的。

为了实现抽象传递闭包，我们对有向图预处理如下：

- 找出其强分量
- 构建其核心 DAG
- 计算核心 DAG 的传递闭包

可以使用 Kosaraju 算法、Tarjan 算法或 Gabow 算法来找出强分量；对边做一遍处理来构建核心 DAG（下一段中描述）；且使用 DFS（程序 19.9）来计算出其传递闭包。在此预处理之后，可以立即得到确定可达性所需信息。

一旦有了有向图的强分量的顶点索引数组，那么构建其核心 DAG 的邻接数组表示就是一件简单的事情。DAG 的顶点时有向图中的分量编号。对于原有向图中的每条边 s-t，我们简单地设置 D -> adj[sc[s]][sc[t]] 为 1。如果使用一个邻接表表示，还需处理核心 DAG 中的重复边的情况，但在一个邻接矩阵中，重复边仅对应为将一个已经设置为 1 的矩阵元素再设置为 1。这点微小的差别很重要，因为在此应用中重复边的数目可能相当大（相对于核心 DAG 的规模）。

性质 19.17 *给定有向图 D 中的两个顶点 s 和 t，设 $sc(s)$ 和 $sc(t)$ 分别是 D 的核心 DAG K 中所对应的顶点。那么 t 在 D 中由 s 可达，当且仅当 $sc(t)$ 在 K 中由 $sc(s)$ 可达。*

证明 由定义可得这个结果。特别是，此性质假设一个约定，顶点由自身可达（所有顶点都有自环）。如果这些顶点都在同一强分量中（$sc(s) = sc(t)$），那么它们相互可达。 ■

确定一个给定顶点 t 是否由一个给定顶点 s 可达，采用的方法与构建核心 DAG 时的一样：使用强分量算法计算出的顶点索引数组，得到分量编号 $sc(s)$ 和 $sc(t)$（常量时间），然后使用这些编号来索引核心 DAG 中的传递闭包，即可得结果。程序 19.13 是体现这些思路的抽象传递闭包 ADT 的一种实现。

对于核心 DAG，也使用抽象传递闭包接口。为了分析，我们假设使用邻接矩阵表示来表示核心 DAG，因为我们期望核心 DAG 不大，最好也不稠密。

程序 19.13 基于强分量的传递闭包

此程序通过计算出有向图的强分量、核心 DAG 以及核心 DAG 的传递闭包（见程序 19.9）来计算其抽象传递闭包。顶点索引数组 sc 给出每个顶点的强分量索引，或其在核心 DAG 中对应的顶点索引。在该有向图中顶点 t 由顶点 s 可达，当且仅当在核心 DAG 中 sc[t] 由 sc[s] 可达。

```
Dag K;
void GRAPHtc(Graph G)
  { int v, w; link t; int *sc = G->sc;
    K = DAGinit(GRAPHsc(G));
    for (v = 0; v < G->V; v++)
      for (t = G->adj[v]; t != NULL; t = t->next)
        DAGinsertE(K, dagEDGE(sc[v], sc[t->v]));
    DAGtc(K);
  }
```

```
int GRAPHreach(Graph G, int s, int t)
  { return DAGreach(K, G->sc[s], G->sc[t]); }
```

性质 19.18　对于一个有向图的抽象传递闭包，可以支持常量的查询时间，所用空间与 $V+v^2$ 成正比，预处理（计算传递闭包）时间与 $E+v^2+vx$ 成正比，其中 v 是核心 DAG 中的顶点数，x 是其 DFS 森林中的交叉边数。

证明　由性质 19.13 直接可得。 ■

如果该有向图是一个 DAG，那么强分量计算就不提供新的信息，而且该算法与程序 19.9 一样；然而，在有环的一般有向图中，这个算法很可能比 Warshall 算法和基于 DFS 的解决方案快得多。例如，性质 19.18 直接蕴含着以下结果。

性质 19.19　对于任何有向图的抽象传递闭包，可以支持常量的查询时间，其核心 DAG 中的顶点数少于 $\sqrt[3]{V}$，所用空间与 V 成正比，预处理时间与 $E+V$ 成正比。

证明　在性质 19.18 中，取 $v<\sqrt[3]{V}$，由于 $x<v^2$，可得 $xv<V$。 ■

我们可能考虑这些界限的其他一些变型。例如，如果我们希望使用与 E 成正比的空间，当核心 DAG 中有 $\sqrt[3]{E}$ 顶点时，也可以达到同样的时间界限。此外，这些时间界限是保守的，因为它们假设核心 DAG 是交叉边稠密的，当然，实际并不需要这样。

将此方法应用时的主要局限因素为核心 DAG 的规模。我们的有向图与 DAG 越类似（核心 DAG 越大），在计算其传递闭包时面临的困难越多。注意到我们仍然没有违背性质 19.9 中蕴含的下界，因为对于稠密 DAG，算法的运行时间与 V^3 成正比。然而，我们已经大大加宽了能够避免最坏性能的一类图的范围。实际上，构造产生有向图的随机有向图模型，使得算法运行较慢也是一个难题（见练习 19.146）。

表 19-2 展示了实验性的研究结果。它说明了即使是对于中等密度的随机有向图，甚至在边的放置上有严格的限制，随机图的核心 DAG 也较小。尽管不能保证最坏情况下的核心 DAG 较小，但可以希望看到在实际中，巨型有向图有较小核心 DAG。当确有这样的有向图时，我们可以提供抽象传递闭包 ADT 的一个高效实现。

表 19-2　随机有向图的性质

此表显示了随机有向图的核心 DAG 中的边数及顶点数，随机有向图由两种不同模型产生（表 18-1 中的模型的有向版本）。对于这两种模型，核心 DAG 随着图的密度增加变小（也变稀疏）。

		随机边图		随机 10-近邻	
	E	v	e	v	e
1 000 顶点					
	1 000	983	981	916	755
	2 000	424	621	713	1039
	5 000	13	13	156	313
	10 000	1	1	8	17
	20 000	1	1	1	1
10 000 顶点					
	50 000	144	150	1 324	150
	100 000	1	1	61	123
	200 000	1	1	1	1

说明：v　核心 DAG 中的顶点数
　　e　核心 DAG 中的边数

练习

• **19.142** 开发有向图的一个基于邻接表表示核心 DAG 的抽象传递闭包实现。你所面临的困难是不使用额外的时间或空间来消除表中的重复项（见练习 19.68）。

▷ **19.143** 对于如下有向图，显示程序 19.13 计算出的核心 DAG 以及传递闭包

```
3-7  1-4  7-8  0-5  5-2  3-8  2-9  0-6  4-9  2-6  6-4
```

○ **19.144** 将基于强分量的抽象传递闭包实现（程序 19.13）转换为一个高效的程序，用以计算邻接矩阵表示的有向图的传递闭包的邻接矩阵，要求使用 Gabow 算法计算强分量和改进的 Warshall 算法计算 DAG 的传递闭包。

19.145 进行实验研究，对于各种类型的图（见练习 19.11～18），估计核心 DAG 的期望大小。

•• **19.146** 开发一个随机图模型，对于含有较大核心 DAG 的有向图进行推广。你的生成器必须一次产生一条边，但不能使用结果图的任何结构性质。

19.147 通过找出强分量和构建核心 DAG，开发有向图的抽象传递闭包的一个实现，然后，回答以下问题：如果两个顶点是在同一个强分量中，对可达性查询做出肯定性回答，否则在 DAG 中执行 DFS 来确定可达性。

19.10 展望

在这一章里，我们考虑了求解有向图和 DAG 的拓扑排序、传递闭包、最短路径问题的一些算法，包括用于找出有向图中的环和强分量的基本算法。这些算法本身有很多重要应用，同时也作为其他更困难问题的基础，包括在接下来的两章中所考虑的加权图。这些算法的最坏情况下的运行时间在表 19-3 中做了总结。

表 19-3 有向图处理操作的最坏情况开销

对于随机图和边随机将各个顶点连向其 10 个近邻之一的图，此表总结了本章考虑的各种有向图处理问题的算法开销（最坏情况下的运行时间）。所有开销假设使用邻接表表示；对于邻接矩阵表示，E 个元素就变成 V^2 个元素，例如，计算所有点对最短路径的开销是 V^3。此线性时间算法是最优的。因而这些开销将能可靠地预测针对任何输入的运行时间；还有一些算法过度保守估计了开销，因而对于某种类型的图，运行时间可能会小一些。计算有向图的传递闭包的最快算法依赖于有向图的结构，特别是其核心 DAG 的规模。

	问 题	开 销	算 法
有向图			
	环检测	E	DFS
	传递闭包	V(E+V)	从每个顶点开始的 DFS
	单源点最短路径	E	DFS
	所有点对最短路径	V(E+V)	从每个顶点开始的 DFS
	强分量	E	Kosaraju、Tarjan 或 Gabow
	传递闭包	E+v(v+x)	核心 DAG
DAG			
	环验证	E	DFS 或源点队列
	拓扑排序	E	DFS 或源点队列
	传递闭包	V(V+E)	DFS
	传递闭包	V(V+X)	DFS/动态规划

特别是，贯穿本章的主题一直是抽象传递闭包问题的求解方法，我们希望在预处理之后，有一个能够快速确定从一个给定顶点到另一个顶点是否存在有向路径的 ADT。尽管下界蕴含着最坏情况下的预处理开销大大高于 V^2，但在 19.7 节所讨论的方法将本章中的几种基本方法结合起来，变成一种简单方法，为很多类型的有向图提供了最优性能。一个例外是稠密 DAG。下界表明对于所有图要保证有更好的性能很难达到，但对于实际中的图，可以使用这些方法来得到良好的性能。

要开发一个算法，使其性能特征与第 1 章中稠密有向图的合并 - 查找算法类似，这个目标仍很困难。理想情况下，我们希望定义一个 ADT，可以添加有向边或者检查一个顶点是否由另一个顶点可达，并能开发一个实现，其中支持所有操作在常量时间内完成（见练习 19.157 ~ 19.159）。如在第 1 章中所讨论的，对于无向图可以接近这个目标，但对于有向图或 DAG 其结果仍然未知。（即使对于无向图，删除边也是一个难题。）这个*动态可达性*（dynamic reachability）问题不仅是基本理论和实际应用的基础，而且在高级抽象的算法开发中起着关键的作用。例如，对于最小成本流问题，可达性是实现网络单纯形算法问题的核心，具有更广泛应用的解问题的模型将在第 22 章考虑。

处理有向图和 DAG 的很多算法都有重要的实际应用，而且得到了深入研究。很多图处理问题仍然需要开发高效的算法。以下列出一些有代表性问题。

支配者　给定一个 DAG，其中所有顶点都由一个源点 r 可达，如果从 r 到 t 的每条路径都包含 s，则称顶点 s 支配顶点 t。（特别是，每个顶点都支配自己）。除了源点之外的每个顶点 v 都有一个*直接支配者*（immediate dominator），它支配 v，但不会支配除 v 和自身之外的 v 的任何支配者。直接支配者的集合是一棵树，它覆盖了由源点可达的所有顶点。可用基于 DFS 的方法在线性时间内计算出支配者树，只需要使用少量的辅助数据结构，不过在实际中会使用稍慢一点的版本。

传递归约　给定一个有向图，存在一些与之有相同传递闭包的有向图，在所有这些有向图中找出边数最少的一个有向图。这个问题是易解的（见练习 19.154）；但如果将结果限制为原图的子图，则是 NP-难的。

有向欧拉路径　给定一个有向图，是否存在连接两个给定顶点的一条有向路径，要求这条路径只使用有向图中的每条边一次？问题很简单，所用论述过程与我们在无向图中对相应问题所做的论述相同，在 17.7 节中讨论过（见练习 17.92）。

有向邮差问题　给定一个有向图，要找出一条边数最少的有向回路，要求使用图中的每条边至少一次（允许多次使用边）。我们将在 22.7 节看到，这个问题可归约到最小成本流问题，因此它是易解的。

有向哈密顿路径　找出一个有向图中的最长简单有向路径。这个问题是 NP-难的，但如果有向图是 DAG，则是简单的（见练习 19.116）。

单连通子图　如果任意顶点对之间至多存在一条有向路径，则称一个有向图是*单连通的*（uniconnected）。给定一个有向图和一个整数 k，确定是否存在至少有 k 条边的一个单连通子图。对于一般 k，这个问题已知是 NP-难的。

反馈顶点集　确定一个给定有向图是否存在至多 k 个顶点的子集，它至少包含来自 G 的每个有向环中的一个顶点，这个问题已知是 NP-难的。

偶环　确定一个给定有向图是否存在长度为偶数的环。在 17.8 节已经提到，这个问题尽管不是难解的，也很难求解，以至于还没有人能够设计一个在实际中可用的算法。

正如对无向图一样，大量的图处理问题已经得到研究，而且要确定一个问题是否是简单

还是难解的，常常是一种挑战（见 17.8 节）。正如贯穿本章的结论，我们所发现的关于有向图的一些结论表达了更一般的数学现象，很多算法也在不同于工作的层面应用于各个抽象层次中。一方面，难解性的概念表明，在寻求能够保证高效求解某些问题的高效算法时，可能遇见基础性的障碍。另一方面，本章描述的经典算法都极其重要，而且应用广泛。因为它们已经给出了常见实际问题的高效解决方案，否则这些问题将会很难求解。

练习

19.148 修改程序 17.13 和 17.14，实现打印有向图中一条欧拉路径的 ADT 函数（如果这条路径存在的话）。解释需要对代码所做的补充或修改的作用。

▷ **19.149** 画出以下有向图的支配者树。

```
3-7  1-4  7-8  0-5  5-2  3-0  2-9  0-6  4-9  2-6
6-4  1-5  8-2  9-0  8-3  4-5  2-3  1-6  3-5  7-6
```

●● **19.150** 编写一个 ADT 函数，它使用 DFS 来创建给定有向图的支配者树的父链接表示（见第 5 部分参考文献）。

○ **19.151** 找出如下有向图的传递归约

```
3-7  1-4  7-8  0-5  5-2  3-0  2-9  0-6  4-9  2-6
6-4  1-5  8-2  9-0  8-3  4-5  2-3  1-6  3-5  7-6
```

● **19.152** 找出与如下有向图有相同传递闭包的一个子图，且使该子图中的边数最少。

```
3-7  1-4  7-8  0-5  5-2  3-0  2-9  0-6  4-9  2-6
6-4  1-5  8-2  9-0  8-3  4-5  2-3  1-6  3-5  7-6
```

○ **19.153** 证明每个 DAG 有唯一的传递归约，并给出计算 DAG 的传递归约的一个高效的 ADT 函数实现。

● **19.154** 编写一个计算传递归约的有向图的 ADT 函数。

19.155 给出一个确定给定有向图是否是单连通的算法。你的算法的最坏情况运行时间应该与 VE 成正比。

19.156 找出如下有向图的最大单连通子图

```
3-7  1-4  7-8  0-5  5-2  3-0  2-9  0-6  4-9  2-6
6-4  1-5  8-2  9-0  8-3  4-5  2-3  1-6  3-5  7-6
```

▷ **19.157** 开发一个有向图 ADT 的函数包，从边数组构造一个图，支持插入一条边、删除一条边、检查两个顶点是否在同一个强分量中，使得在最坏情况下构造、插入、删除都需线性时间，强连通性查询需要常量时间。

○ **19.158** 求解练习 19.157，要求插入、删除和强连通性查询所需时间在最坏情况下都与 $\log V$ 成正比。

●●● **19.159** 求解练习 19.157，要求插入、删除和强连通性查询所需时间为近似常量（类似于无向图中连通性的合并 - 查找算法）。

第 20 章　最小生成树

很多应用需要一个图模型，其中每条边上关联权值（weight）或成本（cost）。在一个航线图中，其中边表示航线，权值可能表示距离或费用。在一个电路中，边表示电线，权值可能表示电线的长度、也可能表示信号通过它传播的成本或时间。在作业调度问题中，权值可能表示执行任务或等待任务完成的时间或成本。

在这种情况下，自然地出现了成本最小化的问题。我们考察这两种问题的算法：（ⅰ）找出将所有点连接起来的最小成本；（ⅱ）找出两个给定点之间的最小成本的路径。第一类算法在无向图中（表示诸如电路等对象）是有用的，即寻找最小生成树（minimum spanning tree），这类算法是本章的主题。第二类算法对有向图（表示诸如航线图等对象）很有用，即寻找最短路径（shortest path）问题；这类算法是第 21 章的主题。除了电路和地图应用之外，这些算法还有广泛的应用，可以扩展到加权图上出现的大量问题。

在我们研究处理加权图的算法时，直觉上常常将权值作为距离：我们谈到“与 x 最近的顶点”，诸如此类的说法。实际上，术语“最短路径”包含了这种狭义的认识。尽管在大量应用中我们确实要用到距离，而且尽管在理解基本算法时几何直观性具有好处，我们还是要记住权值并不需要与距离成正比；它们可能表示时间、成本或完全不同的变量。实际上，我们将在第 21 章看到，最短路径问题中的权值甚至可以是负值。

在描述算法和示例中要求这样的直觉，同时还要保持一般的应用性，我们使用模糊术语，交替地使用边长度和权值。当我们谈到一条“短”边时，意思是“权值较小”的边，如此等等。对于本章中的大多数例子，使用与两个顶点之间距离成正比的权值，如图 20-1 所示。这样的图对于示例很方便，因为不需要对边编号，仍然可以一眼看出较长的边相对于较短的边有较大的权值。在权值确实表示距离时，可以考虑通过结合几何性质获得高效的算法（见 20.7 节和 21.5 节）。除有特别说明，我们所考虑的算法只是简单的处理边，并没有利用任何隐含的几何信息（见图 20-2）。

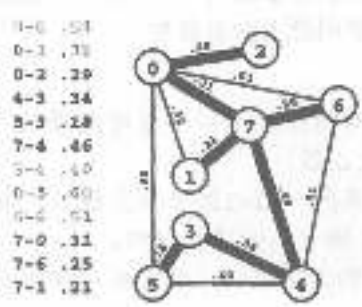


图 20-1　加权无向图及其 MST

加权无向图是一组加权边。MST 为连接所有顶点且总权值最小的一组边（边表中以黑体显示，图示中以粗边表示）。在此特定的图中，权值与顶点之间的距离成正比，但我们考虑的基本算法适合于一般图，对权值不做假设（见图 20-2）。

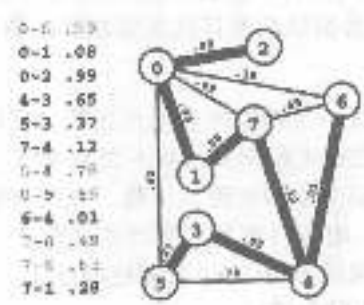


图 20-2　任意权值

在此例中，权值是任意的，而且与所画图表示的几何性质完全没有关系。这个例子也说明了如果边上的权值可以相等，那么 MST 不一定是唯一的：我们将 3-4 包含进去就得到一个 MST，将 0-5 包含进去则得到另一个 MST（尽管 7-6 也有与这两条边相同的权值，但它不在 MST 中）。

找出任意加权无向图中的最小生成树问题有很多重要的应用，而且求解此问题的算法至少从20世纪20年代就已经出现；但实现的效率却大相径庭，而且研究人员仍在寻求更好的方法。在这一节里，我们考察在概念层次上易于理解的三个经典算法；在20.3 ~20.5节将考察每种算法的详细实现；在20.6节，我们对这些基本方法进行比较和改进。

定义20.1 *加权图的一棵最小生成树（minimum spanning tree，MST）是一棵生成树，其权值（边上的权值之和）不大于任何其他生成树的权值。*

如果边权值都是正值，将MST定义为连接所有顶点的总权值最小的一组边就足够了。因为这样的一组边必定形成一棵生成树。定义中包含了生成树的条件，因而它可用于可能含有负边权值的图（见练习20.2和练习20.3）。

如果边上的权值可以相等，最小生成树可能不是唯一的。例如，图20-2显示的图中有两个不同的生成树。权值相等的可能性也使我们对某些算法的描述和正确性证明变得复杂化。我们必须仔细考虑权值相等的情况，因为在应用中这是很常见的，而且在出现权值相等的情况时，我们希望知道算法也能够正确操作。

不仅可能存在不止一个MST，而且这一术语也并没有完全涵盖以下概念：我们最小化的是权值，而不是使树本身最小化。描述一棵特定树的合适的形容词是最小的（minimal）（有最小权值的一棵树）。由于这些原因，很多作者使用准确的术语，像“最小生成树”或“最小权值生成树”。缩写MST是最为常用的，普遍认为它涵盖了其中的基本概念。

然而，对于边上权值可能相等的网络，在描述其上的算法时，为了避免混淆，特别注意准确地使用以下术语，即“最小的”表示“有最小权值的边”（在某个特定集合的所有边中），“最大的”表示“有最大权值的边”。也就是说，如果边上权值互不相同，最小边即为最短边（且是唯一最小边）；但是如果有多于一条的最小权值的边，其中任何一条都可以是最小边。

本章只处理无向图。在有向图中找出最小权值有向生成树的问题是另一个问题，而且更困难。

对于MST问题已经开发了几种经典的算法。这些方法是本书中最古老且最知名的算法。以前曾看到，经典方法提供了一种一般方法，但是现代算法和数据结构可以给出简洁且高效的实现。实际上，这些实现提供了令人信服的示例，说明了精心进行ADT设计和正确选择基本ADT数据结构及算法实现在解决越来越困难的算法学问题上的有效性。

练习

20.1 假设一个图中的权值是正数。证明可以通过对所有权值加上一个常量或乘以一个常量对权值进行重新调整，而不影响MST，假如调整权值是正数。

20.2 如果边上的权值是正数，表明将所有顶点连接起来的一组边如果其上权值之和为不大于另外一组边（将所有顶点连接起来）上的权值之和，那么这组边为MST。

20.3 说明练习20.2中所阐述的性质，对于包含负权值的图也成立。假定不存在所有边都为非正权值的负环。

○ 20.4 如何找出加权图的一个最大生成树？

▷ 20.5 说明如果图中所有边上的权值互不相同，那么MST是唯一的。

▷ 20.6 考虑断言：一个图中有唯一的MST，仅当其边上的权值互不相同，给出它的一个证明或给出一个反例。

● 20.7 假设图中有$t < V$条边有相同的权值，其他边上的权值都不相同。给出该图可能有的不同MST数目的上界和下界。

20.1　表示

在这一章里，我们主要关注加权无向图，这是 MST 问题最自然的应用环境。对第 17 章的基本图表示进行扩展来表示加权图很直接：在邻接矩阵表示中，矩阵可以包含边权值，而不是布尔值；在邻接表表示中，可以在表元素中增加一个表示边上权值的域。

在我们例子中，我们一般假设边的权值为 0 和 1 之间的实数。这个假设并不与应用中需要的各种其他选择发生矛盾。因为可以显式或隐式重新调整权值以适合这个模型（见练习 20.1 和练习 20.8）。例如，如果权值是小于一个已知最大值的正整数，可以将它们除以该最大值，将它们转换为 0 和 1 之间的实数。

我们使用在第 17 章所用的同一基本图 ADT 接口（见程序 17.1），只是要在边数据类型中添加一个权值域，如下：

```
typedef struct { int v; int w; double wt; } Edge;
Edge EDGE(int, int, double);
```

为了避免简单类型过多，本章和第 21 章使用 double 类型表示边权值的类型。如果希望这样做，我们就能构建一个更一般的 ADT 接口，并使用任何支持加、减和比较的数据类型，因为我们常常只是对权值求和并基于其值作出决策，对权值并不做其他工作。在第 22 章中，算法主要关注的是比较边权值的线性组合，某些算法的运行时间依赖于权值的算术性质，因此，我们会切换到整数权值以便更简单地分析算法。

我们使用观察哨权值来指示不存在的边。另一种直接方法是使用标准邻接矩阵来指示边是否存在，且使用一个平行矩阵来保存权值。有了观察哨，很多算法都不需要显式检查边是否存在。图 10.3 显示了我们的示例图的邻接矩阵表示；程序 20.1 给出了邻接矩阵表示的加权图 ADT 的实现细节。它使用了一个分配矩阵权值的辅助函数，并用观察哨权值填充矩阵。插入一条边要将权值存放在矩阵的两个地方，因为边有两个方向。指示边不存在的观察哨权值大于其他所有权值，而不是 0，这表示长为 0 的边（另一种选择是不允许长度为 0 的边）。类似于面向不加权图使用邻接矩阵表示的算法，使用这种表示的算法的运行时间与 V^2 成正比（初始化矩阵）或更高。

类似地，程序 20.2 给出了邻接表表示的加权图 ADT 的实现细节。顶点索引的数组将每个顶点与该顶点依附边的一个链表关联。每个表结点表示一条边，且包含一个权值。图 20-3 显示了示例图的一种邻接表表示。

程序 20.1　加权图 ADT（邻接矩阵）

对于稠密加权无向图，我们使用权值矩阵，其 v 行、w 列和 w 行、v 列处的元素含有边 v-w 上的权值。观察哨值 maxWT 表示边不存在。此段代码假设边权值的类型为 double，并使用辅助例程 MATRIXdouble 来分配一个 $V \times V$ 的权值数组，其所有元素初始化为 maxWT（见程序 17.4）。要修改这段代码用于加权有向图 ADT 实现（见第 21 章），删除 GRAPHinsertE 的最后一行。

```
#include <stdlib.h>
#include "GRAPH.h"
struct graph { int V; int E; double **adj; };
Graph GRAPHinit(int V)
  { int v;
    Graph G = malloc(sizeof *G);
```

```
    G->adj = MATRIXdouble(V, V, maxWT);
    G->V = V; G->E = 0;
    return G;
  }
void GRAPHinsertE(Graph G, Edge e)
  {
    if (G->adj[e.v][e.w] == maxWT) G->E++;
    G->adj[e.v][e.w] = e.wt;
    G->adj[e.w][e.v] = e.wt;
  }
```

	0	1	2	3	4	5	6	7
0	*	.32	.29	*	*	.60	.51	.31
1	.32	*	*	*	*	*	*	.21
2	.29	*	*	*	*	*	*	*
3	*	*	*	*	.34	.18	*	*
4	*	*	*	.34	*	.40	.51	.46
5	.60	*	*	.18	.40	*	*	*
6	.51	*	*	*	.51	*	*	.25
7	.31	.21	*	*	.46	*	.25	*

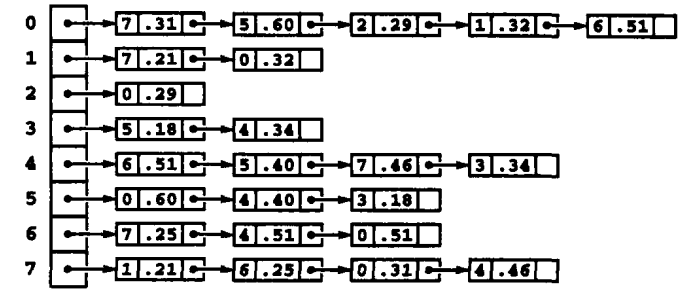


图 20-3 加权图表示（无向）

在这两种加权无向图的标准表示中，每边表示中都包含权值。使用图 20-1 中所示的图说明了其邻接矩阵表示（左图）和邻接表表示（右图）。邻接矩阵是对称的，邻接表中包含每条边的两个结点，这与无权有向图一样。不存在的边在矩阵中用观察哨值（图中用星号指示）表示，而在表中则根本不会出现。在此所示的两个表示中都没有自环，因为如果没有自环 MST 会简单一些；处理加权图的其他算法会用到自环（见第 21 章）。

与无向图表示一样，在上述的任何一种表示中，我们并不显式地检查平行边。取决于应用，我们可能会改变邻接矩阵表示，以保存最小权值或最大权值的平行边，或者有效地将平行边合并为带有其权值之和的一条边。在邻接表表示中，可以允许平行边在数据结构中存在，或者使用上面描述的关于邻接矩阵的规则之一，来构建更强大的数据结构以消除平行边（见练习 17.47）。

程序 20.2 加权图 ADT（邻接表）

此邻接表表示适合于稀疏加权无向图。与在加权无向图中一样，每条边用两个表结点表示，分别在这条边各个顶点的相应邻接表中。为了表示这些权值，在表结点域添加一个权值域。

这个实现并没检查重复边。除了要考虑无权图的这些因素之外，还要做出一个设计决策，是否允许连接同一对顶点的多条边有不同权值，这种情况适合于某些应用。

要修改此段代码用于加权有向图的 ADT 实现，（见第 21 章），删除添加从 v 到 w 的边的相应代码行（删除 GRAPHinsertE 的倒数第二行）

```
#include "GRAPH.h"
typedef struct node *link;
struct node { int v; double wt; link next; };
struct graph { int V; int E; link *adj; };
link NEW(int v, double wt, link next)
  { link x = malloc(sizeof *x);
    x->v = v; x->wt = wt; x->next = next;
    return x;
```

```
  }
Graph GRAPHinit(int V)
  { int i;
    Graph G = malloc(sizeof *G);
    G->adj = malloc(V*sizeof(link));
    G->V = V; G->E = 0;
    for (i = 0; i < V; i++) G->adj[i] = NULL;
    return G;
  }
void GRAPHinsertE(Graph G, Edge e)
  { link t;
    int v = e.v, w = e.w;
    if (v == w) return;
    G->adj[v] = NEW(w, e.wt, G->adj[v]);
    G->adj[w] = NEW(v, e.wt, G->adj[w]);
    G->E++;
  }
```

如何表示 MST 自身呢？图 G 的 MST 是 G 的一个子图，也是一棵树，因此有很多选择。主要包括：

- 图
- 边链表
- 边数组
- 有父链接的顶点索引的数组

对于图 20-1 中的示例 MST，图 20-4 展示了这些选择。另一种选择是定义并使用树的 ADT。

对于任何一种模式，同一棵树都可能有多种不同的表示。在边表表示中边是按照何种顺序出现的？在父链接表示中应该选择哪个结点作为根结点（见练习 10.15）？一般而言，当运行一个 MST 算法时，所得特定 MST 表示与所用算法有关，而不能反映 MST 的任何种要特征。

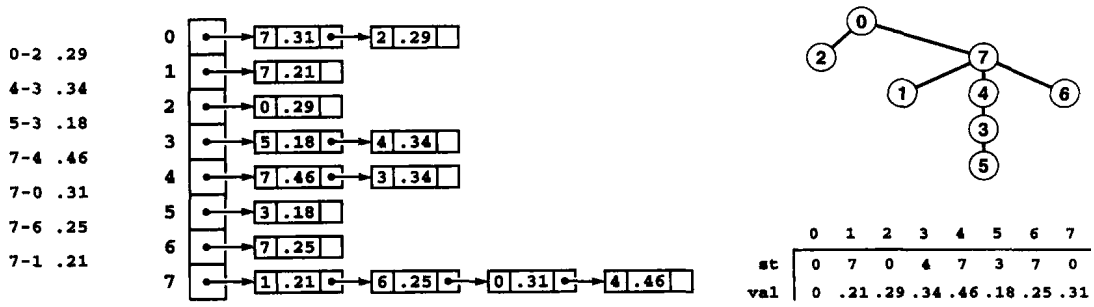


图 20-4　MST 表示

此图描述了图 20-1 中的 MST 的各种表示。最直接的表示是边表，没有特定的顺序（左图）。MST 也是一个稀疏图，可以用邻接表表示（中图）。最简洁的表示是父链接表示：选择其中一个顶点作为根，并保存两个顶点索引的数组，一个用于存放树中每个顶点的父结点，另一个用于存放从一个顶点到其父结点的边上的权值（右图）。树的方向（根顶点的选择）是任意的，不是 MST 的性质。我们可以在线性时间内从任何一种表示转换为另一种表示。

从算法的观点来看，选择 MST 的表示没有太多意义。因为我们可以很容易地将每种表示转换为另一种表示。要从图表示转换为边数组表示，可以使用图 ADT 中的 GRAPHedges 函数。要从数组 st 的父链接表示转换为边数组放在数组 mst 中，可以使用如下简单循环

```
for (k = 1; k < G->V; k++) mst[k] = EDGE(k, st[k]);
```

这个代码对应根结点为0的MST典型情况，它不会将虚拟边0-0放入MST的边表中。

这两种转换很容易，但如何将边数组表示转换为父链接表示呢？还有一些基本工具可以完成容易地完成这项任务：使用类似于上面给出的循环，可以转换到图表示（变成对于每条边调用GRAPHinsert），然后从任何顶点开始运行DFS，在线性时间内计算出DFS树的父链接表示。

简而言之，选择MST的表示对于算法而言是一件便利的事情。依赖于应用的需要，我们可以增加包装器函数，使客户程序具有所需的灵活性。对于我们考虑的一些算法，父链接表示更自然一些；而对于其他算法，其他表示更自然一些。本章的目标是开发能够支持图ADT函数GRAPHmst的高效实现；并不指定接口细节，从而为满足客户需要的简单封装函数留出灵活性，同时又允许每个算法以自然的方式产生MST的实现（例如，见练习20.18～20.20）。

练习

▷ 20.8 构建一个使用整数权值的图ADT，但记录图中最小和最大权值并包含一个总是返回0～1之间的数的ADT函数。

▷ 20.9 修改程序17.7中的稀疏随机图生成器，给每条边赋以一个随机权值（0～1之间）。

▷ 20.10 修改程序17.8中的稠密随机图生成器，给每条边赋以一个随机权值（0～1之间）。

20.11 编写一个产生随机加权图的程序，将排列在$\sqrt{V}\times\sqrt{V}$网格上的顶点与近邻连接起来（如图19-3，但无向），每条边赋以0～1之间的随机权值。

20.12 编写一个产生随机完全图的程序，由高斯分布来选择图中的权值。

● 20.13 编写一个产生平面上V个随机点的程序，然后构建一个加权图，将平面上给定距离d内的每对顶点连接起来，其边上的权值即为该距离。（见练习17.60）确定如何设置d，使得边的期望值为E。

● 20.14 找出一个大型的在线加权图，可以是带有距离的地图、带有成本的电话连接，或者是航线费率表。

20.15 编写一个8×8矩阵，要求包含图20-1中图的MST的所有方向的父链接表示，要求将以i为根的树的父链接表示放在矩阵的第i行中。

▷ 20.16 对于程序20.1和程序20.2中的邻接矩阵和邻接表的实现，向其中添加GRAPHscan、GRAPHshow和GRAPHedges函数。

▷ 20.17 提供程序20.1中所用的MATRIXdouble函数的一种实现。

▷ 20.18 假设函数GRAPHmstE可在数组mst中产生MST的一个边数组表示。在图ADT中添加ADT函数GRAPHmst，要求调用GRAPHmstE，但将该MST的父链接表示放在一个数组中，该数组由客户程序作为参数传递。

○ 20.19 假设函数GRAPHmstV可在数组st中产生MST的一个父链接表示，与另一数组wt中的权值相对应。在图ADT中添加ADT函数GRAPHmst，要求调用GRAPHmstV，但将该MST的边数组表示放在一个数组中，该数组由客户程序作为参数传递。

▷ 20.20 定义树的tree ADT。然后在练习20.19的假设之下，向图ADT中添加一个调用GRAPHmst但在Tree中返回该MST的ADT函数。

20.2 MST算法的基本原理

MST是本书中遇到的研究最为深入的问题之一。在现代数据结构和研究算法性能的现代

技术之前，研究此问题的基本方法已有很长历史。当时找出含有数千条边的图的 MST 是一项棘手的任务。我们将会看到，几种新的 MST 算法与以往的算法在针对基本任务的使用和现代算法和数据结构的实现上有所不同，而这（与现代计算能力的结合）使我们有可能计算出有数百万甚至数十亿条边的 MST。

树所定义的性质之一（见 5.4 节）就是向树中添加一条边会创建一个环。这个性质是我们将要考虑的证明 MST 的两个基本性质的基础。我们遇到的所有算法都是基于这两个性质或其中一个性质。

我们称第一个性质为*割性质*（cut property），此性质必定能识别出在给定图的 MST 中的边。接下来定义几个图论中的基本术语，然后简洁阐述这个性质，如下：

定义 20.2　图中的*割*（cut）是将顶点分为两个不相交集的一个划分。*交叉边*（crossing edge）是将其中一个集合中的顶点与另一个集合中的顶点连接起来的边。

我们有时通过指定顶点集来指定一个割，这蕴含着假设：割包含了那个顶点集和它的补集。一般地，我们使用的割，其中两个集合都非空。否则，就不存在交叉边。

性质 20.1（割性质）　给定图的任意一个割，每条最小的交叉边属于图的某个 MST 中，且每个 MST 包含一条最小交叉边。

证明　用反证法。假设 e 是一条不在任何 MST 中的最小交叉边，且设 T 是任意一个 MST；或假设 T 是一个不包含最小交叉边的 MST，且 e 是任意一条最小交叉边。无论哪一种情况，T 都是一个不含最小交叉边 e 的 MST。现在考虑向 T 添加 e 所形成的图。此图含有包含 e 的环，而且该环必定至少包含另一条交叉边，比如说 f，其权值等于或大于 e（因为 e 是最小的）。删除 f 并添加 e，可以得到一棵权值相等或更小的生成树。这要么与 T 的最小性相矛盾，要么与假设 e 不在 T 中相矛盾。 ■

如果图中边的权值互不相同，则有唯一 MST；由割性质可得每个割的最短交叉边必定在 MST 中。当出现权值相等的情况时，可能会有多个最小交叉边。至少其中之一会在任何给定的 MST 中，其他最小交叉边可能出现在 MST 中，也可能不出现在 MST 中。

图 20-5 展示了割的这一性质的几个例子。注意，最小边会是连接这两个集合的唯一 MST 的边，这一点并不作要求。实际上，对于典型的割，存在几条 MST 边将一个集合中顶点与另一个集合中的顶点连接起来。如果能够确信存在唯一这样的边，我们就能开发一个基于审慎选择的分治算法，但是情况并非如此。

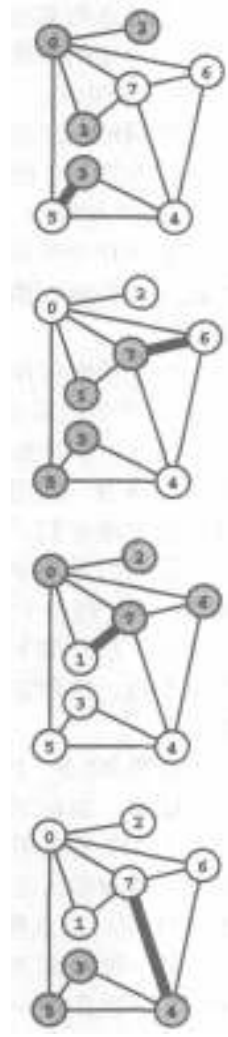


图 20-5　割性质

这四个例子展示了性质 20.1。如果我们对一个集合中顶点着灰色，另一个集合中的顶点着白色，那么连接一个灰色顶点与一个白色顶点的最短边属于 MST。

我们使用割性质作为寻找 MST 的算法的基础。它也可用作表征 MST 的*优化条件*（optimality condition）。具体地说，对于由边连接起来的两个子树中的顶点所定义的割，割的性质蕴含着 MST 中的每条边是最小交叉边。

第二个性质称为*环性质*（cycle property），此性质必定能识别出不在图的 MST 中的边。也就是说，如果忽略了这些边，仍然可以找到一个 MST。

性质 20.2（环性质）　给定一个图 G，考虑向 G 中添加一个边所定义的图 G'。将 e 添加到 G 的一个 MST 中，并删除所得环中的一条最大边，可得 G' 的一个 MST。

证明　如果 e 是此环中所有其他的边中较长的，由性质 20.1 可得，它不可能在 G' 的 MST 中：从任何这样的 MST 中删除 e，都将把该 MST 分为两部分，而 e 也不是连接这两部分中某对顶点（每一部分取一个顶点）的最短边，因为环中必定存在其他的边满足此条件。否则，对于向 G 的 MST 增加 e 而创建的环，设 t 为此环中的一条最大边。删除 t 将把原 MST 分解为两部分，而且 G 中连接这两部分的 G 的边不会比 t 短；因此 e 是 G' 中连接这两部分顶点的一条最小边。对于 G 和 G' 而言，由两个顶点子集所导出的子图是相同的，因此 G' 的一棵 MST 将由 e 和这两个子集的 MST 所组成。

特别地，注意如果 e 是环中最大的边，则表明 G' 中存在一棵不包含 e 的 MST（G 的 MST）■

图 20-6 展示了这一环性质。注意取一棵生成树，添加边创建环，然后删除环中的最大边的过程，可得权值小于或等于原生成树的一棵生成树。新树权值小于原树权值，当且仅当所添加的边比环中的某条边更短。

环性质也可作为表征 MST 的优化条件的基础：它蕴含着图中不在给定 MST 中的每条边是其与 MST 所形成的环中的一条最大边。

割性质和环性质是我们所考虑的 MST 问题经典算法的基础。每次考虑一条边，使用割性质将它们接受为 MST 中的边，并在需要时使用环性质拒绝它们。有效地识别割和环的不同方法导致了不同算法。

我们所考虑的找 MST 的第一种方法是通过一次一条边来构建 MST：从任何一个顶点作为单个顶点的 MST 开始，然后向它添加 $V-1$ 个顶点，总是取下一条连接 MST 中的一个顶点与不在 MST 的一个顶点的最小边。这种方法称为 Prim 算法（Prim's algorithm）；它是 20.3 节中的主题。

性质 20.3　Prim 算法计算任何连通图的一个 MST。

证明　如在 20.2 中详细描述的，该方法是一种广义图搜索方法。性质 18.12 的证明中蕴含着：所选择的边是一棵生成树。为了显示它们是一棵 MST，应用割性质，将在 MST 中的顶点作为第一个集合，不在 MST 中的顶点作为第二个集合。■

另一种计算 MST 的方法是反复地应用环性质：我们向一个假想 MST 中一次添加一条边，如果形成环，则删除此环中的一条最大边（见练习 20.28 和 20.66）。较之于我们考虑的其他方法，这种方法受到的关注较少。因为维持一个支持操作“删除环中的最长边”的有效实现的数据结构难度相当大。

我们所考虑的第二种找 MST 的方法是按照边的长度顺序来处理边（最短边优先），向 MST 中添加一条边，使得这条边不与以前加入的边形成环，在添加 $V-1$ 条边之后停止。这种方法称为 Kruskal 算法（Kruskal's algorithm）；它是 20.4 中描述的主题。

性质 20.4　Kruskal 算法可以计算任何一个连通图的 MST。

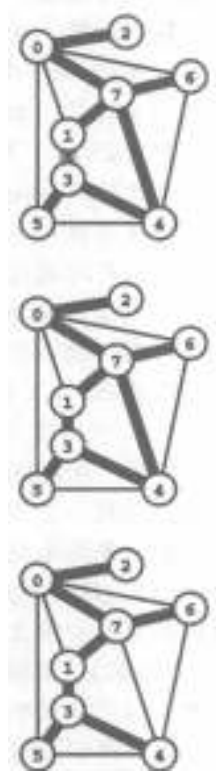


图 20-6　环性质

向图 20-1 中的图添加边 1-3 使 MST 无效（上图）。要找出新图的 MST，向原图的 MST 中添加新边，创建一个环（中图）。删除此环中的最长边（4-7）得到新图的 MST（下图）。验证这棵生成树是最小的一种方法是，检查不在 MST 中的每条边在其与树边形成的环中有最大权值。例如，在下图中，4-6 在环 4-6-7-1-3-4 中有最大权值。

证明　用归纳法证明该方法维持 MST 子树的森林。如果要被考虑的下一条边会创建环，那么，它是环上的一条最大边（因为按照排序的顺序，其他的边都会先于这条边出现），因而，由环性质，忽略它仍然是一个 MST。如果所要考虑的下一条边不构成环，则应用割性质，使用由 MST 的边连接到该边某个顶点的集合所定义的割。因为该边并不创建环，它是唯一交叉边，而且由于是按照有序顺序考虑边，因而这是一条最小边，因此它在 MST 中。归纳基础是 V 个单独的顶点，一旦选择了 $V-1$ 条边，则有了一棵树（MST）。不存在未经检查的边比 MST 的一条边更短，而且都将产生一个环，因而，由环性质，忽略掉其余所有边仍然得到一个 MST。■

我们所考虑的构建一个 MST 的第三种方法称为 Boruvka 算法；它是 20.4 节的主题。第一步是向 MST 中添加将每个顶点与它的最近邻连接起来的边。如果边上权值互不相同，这一步就创建了 MST 子树的一个森林（我们证明了这一点，稍后将考虑即使出现权值相等的边也能做到这一点的改进方法）。然后，向 MST 中添加将每个树与其最近邻连接起来的边（连接一个树中的顶点与另一个树中的顶点的最小边），反复进行这一过程，直到只剩下一棵树。

性质 20.5　*Boruvka 算法可以计算任何连通图的 MST。*

证明　首先，假设边的权值互不相同。在这种情况下，每个顶点都有唯一的最近邻，MST 是唯一的，应用割性质可知，每条所添加的边都是 MST 的一条边（它是从一个顶点到其他所有顶点的穿越割的最短边），因为每条所选择的边来自于唯一 MST，不可能存在环。每条添加的边将来自森林的两棵树合并在一起，形成一棵更大的树。继续这一过程，直到只剩一棵树，也即 MST。

如果边权值并不是互不相同，可能会有多个最近邻，在向最近邻添加边的时候就可能形成环（见图 20-7）。换句话说，对于某个顶点，可能包含来自最小交叉边集合中的两条边。为了避免这个问题，我们需要一个合适的解结规则。一种选择是在最小近邻中选择具有最小顶点编号的近邻。然后，任何环都会引出矛盾：如果 v 是环中编号最大的顶点，那么不存在 v 的近邻会使其选择是最近的，而且 v 也会使编号较小的近邻之一被选上，而不会将两个顶点都选上。■

这些算法都是研究人员寻求新的 MST 算法仍在使用的一般范型的特例。具体地说，我们可以以*任意的次序*（arbitrary order）应用割性质来接受一条边作为 MST 的一条边，或者环性质来拒绝一条边。继续这个过程，直到进一步应用性质不再增加边或拒绝边为止。此时，将图顶点分为两个集合的任何划分都会有一条 MST 的边，将这两个顶点集连接起来（因此应用割性质不再增加 MST 的边数），而且图的所有环都至少有一条非 MST 的边（因而应用环性质不会增加非 MST 的边数）。结合这两点，这些性质蕴含着可以计算出一个完整的 MST。

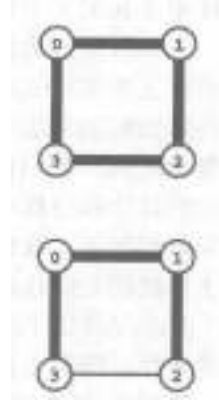


图 20-7　Boruvka 算法中的环

在这里显示的 4 个顶点和 4 条边的图中，边的长度都相等。将每个顶点与其最近邻连接时，必须在最小边之间作出选择。在此例上图中，从顶点 0 选择顶点 1，从顶点 1 选择顶点 2，从顶点 2 选择顶点 3，从顶点 3 选择顶点 0。这导致在假定的 MST 中有一个环。每条边都可出现在某个 MST 中，但并不是所有的边都要出现在每个 MST 中。为了避免出现这个问题，我们采用一个破结规则，如下图所示：选择指向有最小索引顶点的最小边。因此，我们从顶点 0 选择顶点 1，从顶点 1 选择顶点 0，从顶点 2 选择顶点 1，从顶点 3 选择顶点 0，产生一个 MST。环被破坏掉，因为编号最大的顶点 3 没有被它的近邻 2 或 1 选上，而且它只能选择其中的一个（0）。

更具体地说，我们详细考虑的三个算法可以统一为一个广义算法，其中从单个顶点的MST子树的森林开始（每棵子树中都不含边），向MST中添加连接森林中任何两个子树的最小边，继续这个过程，直到添加了 $V-1$ 条边且只剩单个MST为止。由割性质，引起环的边都不会在该MST中考虑，因为对于包含其每个顶点的MST子树之间的割，此前已经增加了某条穿越这个割的另一条最小边。利用Prim算法，每次将增加一条边来建立一棵树；利用Kruskal算法和Boruvka算法，则是将森林中的树进行合并。

如本节以及经典文献所述，这些算法都涉及一些高层次的抽象操作，如下：

- 找出连接两棵子树的最小边
- 确定增加一条边是否会产生一个环
- 删除环中的最长边

一个挑战是开发能够高效实现这些操作的算法和数据结构。幸运的是，这个挑战使我们有机会使用本书早些时间开发的算法和数据结构。

MST算法有一个长且多姿多彩的历史，而且仍在发展之中：我们将在详细讨论算法时再介绍有关历史。对于实现基本抽象操作的不同方法的理解仍在深入，多年来造成了围绕算法起源的一些误解。实际上，此方法最早20世纪20年代描述过，早于我们所知的计算机的开发，同样早于我们关于排序和其他算法的基本知识。现在我们知道，即使在实现最为基本的模式时，基本算法和数据结构的选择对于其性能也会有极大的影响，最近的数年里，对MST问题的研究主要关注其实现问题，仍然使用经典模式。为简明和一致起见，尽管较早时候考虑过这些算法的抽象版本，而且这些方法的现代实现所使用的算法和数据结构远在这些方法首次被提出之后，我们仍将使用在此列出的名字来表示这些基本方法。

算法的设计和分析中至今为止尚未解决的问题是线性时间的MST算法。我们将会看到，在许多实际情况下，我们的实现可以做到线性时间，但是在最坏情况下却是非线性的。开发一种对于稀疏图是线性的算法仍然是一个研究目标。

除了要求寻找这个基本问题的最优算法，关于MST的研究还强调了理解基本算法性能特征的重要性。编程人员会继续在更高层次的抽象上使用算法和数据结构，这种情况将越来越普遍。我们的ADT实现有着不同的性能特征，在解决更高层次的问题时，我们将会把高层次的ADT作为组件，因此可能存在多种情况。实际上，解决更高层次抽象的其他问题时，通常会使用基于MST和类似抽象（由本章所讨论的高效实现所支持）的算法来提供帮助。

练习

▷ **20.21** 使用0～5对以下平面上的点分别进行编号

```
(1, 3) (2, 1) (6,5 ) (3, 4) (3, 7) (5, 3)
```

取边长度为权值，给出由以下边所定义的图的MST

```
1-0  3-5  5-2  3-4  5-1  0-3  0-4  4-2  2-3
```

20.22 假设图中边权值不同。它的最短边必定属于MST吗？请给出证明或给出一个反例。

20.23 对于图的最长边，回答练习20.22中的问题。

20.24 给出一个反例，表明为何以下策略未必找出该MST：“从作为单个MST的任何顶点开始，然后在该MST中添加 $V-1$ 条边，总是取依附于最近添加到该MST中的顶点的最小边。”

20.25 假设图中的权值互不相同。每个环中的最小边必定属于该MST吗？请给出证明或给出一个反例。

20.26　给定图 G 的一个 MST，假设删除 G 中的一条边。描述如何找出新图中的一个 MST，所用时间与 G 中的边数成正比。

20.27　显示不断地将环性质应用于图 20-1 中的图上所得的结果，要求按照给定的顺序取得边。

20.28　证明不断地将环性质应用于图 20-1 中的图可以得到一个 MST。

20.29　描述如何对本节描述的每个算法进行改编（需要时），使得其能用于找出加权图的一个最小生成森林（minimal spanning forest）的问题（其连通分量的 MST 的并集）。

20.3　Prim 算法和优先级优先搜索

Prim 算法可能是实现最简单的 MST 算法，它适合于稠密图。我们维护图的一个割，它由树（tree）顶点（被选入 MST 中的顶点）和非树（nontree）顶点（还未被选入 MST 中的顶点）组成。从将任何放入 MST 中的顶点开始，然后将一条最小交叉边放入 MST 中（这会将它的非树顶点变为树顶点），并重复相同的操作 $V-1$ 次，将所有顶点放到树中。

Prim 算法的蛮力法实现直接由此描述而得。要找出下一个添加到 MST 中的边，我们会检查从一个树顶点到一个非树顶点的所有边，然后选择找到的最短边，并放入 MST 中。这里并不考虑实现，因为其开销过大（见练习 20.30 ~ 20.32）。增加一个简单数据结构可消除过度重复计算，会使算法变得简单且快速。

向 MST 中添加顶点是一个递增的改变：要实现 Prim 算法，我们主要关注增量的变化。我们感兴趣的关键是从每个非树顶点到树顶点的最短距离。当向树添加一个顶点 v 时，对于每个非树顶点 w 的唯一可能的变化是，添加了 v 使得 w 比以前更靠近树。简言之，我们并不需要检查从 w 到所有树顶点的距离，只需要记录最小距离，并检查向树添加 v 后，是否需要更新这个最小距离。

要实现这个思想，需要能够提供如下信息的数据结构：

- 对于每个树顶点，其在 MST 中的父结点
- 对于每个非树顶点，最近的树顶点
- 对于每个树顶点，其父链接长度.
- 对于每个非树顶点，它与树的距离

对于每种数据结构，尽管通过组合数组或使用结构，有很多方法都可以节省空间，但最简单的实现是使用顶点索引数组。

对于邻接矩阵图 ADT 实现，程序 20.3 给出了 Prim 算法的一种实现。实现中使用了数组 st、fr 和 wt 来表示这 4 种数据结构，其中 st 和 fr 分别用于前两种数据结构，wt 用于后两种数据结构。对于树顶点 v，元素 wt[w] 表示与树的距离（与 fr[w] 对应）。此实现被封装在函数 GRAPHmstV 中，将 st 和 wt 作为客户程序提供的数组。如果需要，我们可以添加一个包装器函数 GRAPHmst 来构建一个边表（或其他）MST 的表示，如 20.1 节中所述。

在树中添加一条新边（及顶点）之后，还有两项任务要完成：

- 检查所添新边是否能使任何非树顶点更接近树
- 找出要添加到树中的下一条边

程序 20.3 中的实现只需处理一次非树中的顶点就完成了这些任务，如果 v-w 使 w 更接近此树，则更新 wt[w] 和 fr[w]，而且如果 wt[w]（w 与 fr[w] 的距离）表明 w 比其他顶点更接近此树，那么还要更新当前最小值。

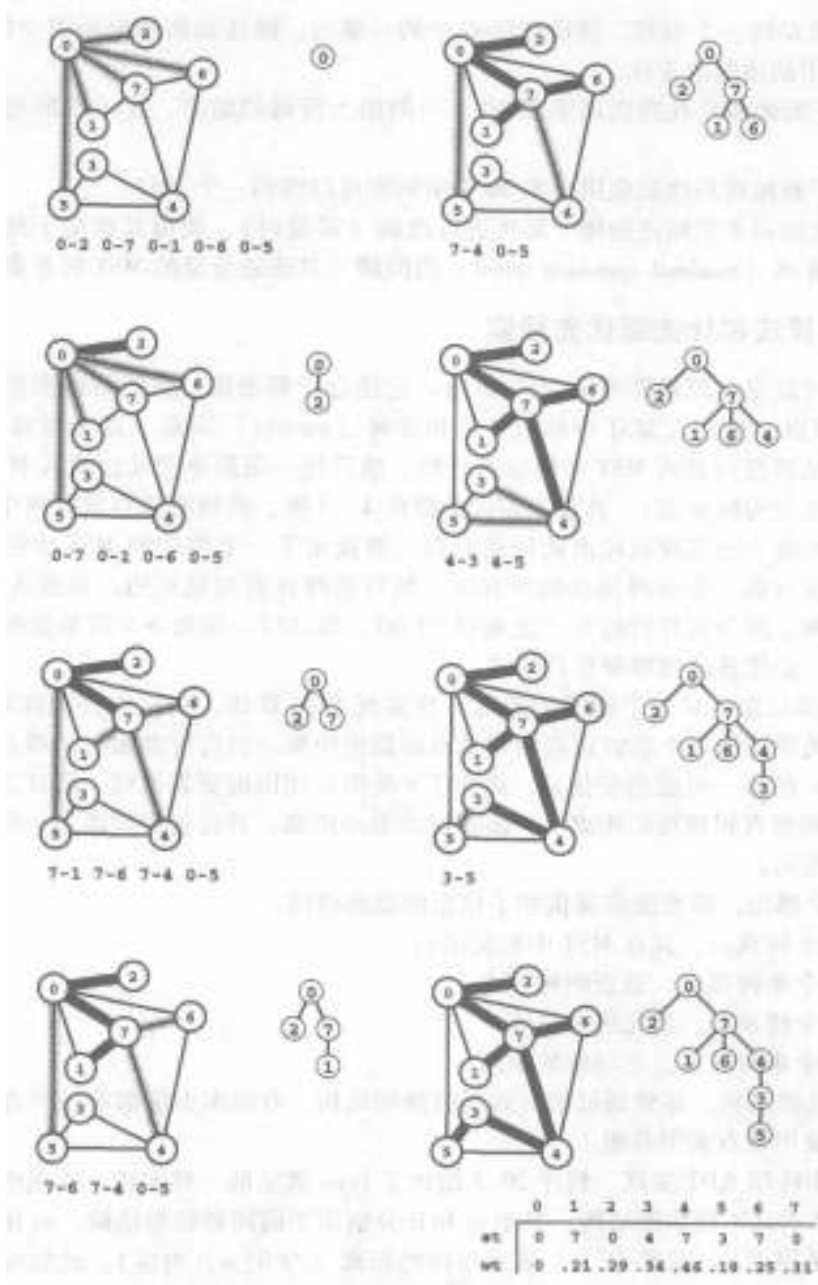


图 20-8 MST 的 Prim 算法

使用 Prim 算法计算 MST 的第一步是将顶点 0 添加到树中。然后，找出连接 0 与其他顶点（尚未在树中）的所有边，并保存最短的边（左上图）。将连接树顶点与非树顶点的边用灰色阴影示出，并列在画出的每个图下方。为简化此图，我们按照其长度顺序列出边缘集中的边，长度最短的是表中第一个元素。Prim 算法的不同实现使用了不同的数据结构来维护这个表并找出这个最小边。第二步是将最短边 0-2（以及所指向的顶点）从边缘集移到树中（左列自上第 2 个图）。第三步是将最短边 0-7 从边缘集移到树中，并用 7-1 代替边缘集中的 0-1，7-6 代替边缘集中的 0-6（因为将 7 增加到树中使得 1 和 6 离树更近），并且将 7-4 添加到边缘集中（因为将 7 添加到树中使 7-4 成为连接一个树顶点与一个非树顶点的边）（左列自上第 3 个图）。接下来，将边 7-1 移到树中（左下图）。要完成计算，我们将 7-6、7-4、4-3 和 3-5 取出队列，并在每次插入之后更新边缘集以反映出任何更短或更新的路径（右列自上向下的全部图）。

有方向地画出逐步增长的 MST 在右图中给出。这种方向性是算法自身的结果：我们将 MST 自身看作既无顺序也无方向的边的集合。

性质 20.6　使用 Prim 算法，可以在线性时间找出一个稠密图的 MST。

证明　观察程序直接可得，运行时间与 V^2 成正比，因此对于稠密图是线性的。 ■

程序 20.3　Prim 算法

Prim 算法的实现是稠密图可选的方法。外层循环通过选择一条穿越割（MST 中的顶点与不在 MST 中的顶点之间）的最小边来增长 MST，w 循环则找出最小边，同时（如果 w 不在 MST 中）又维持循环不变式：从 w 到 fr[w] 的边是从 w 到 MST 的短边（权值 wt[w]）。

```
static int fr[maxV];
#define P G->adj[v][w]
void GRAPHmstV(Graph G, int st[], double wt[])
  { int v, w, min;
    for (v = 0; v < G->V; v++)
      { st[v] = -1; fr[v] = v; wt[v] = maxWT; }
    st[0] = 0; wt[G->V] = maxWT;
    for (min = 0; min != G->V; )
      {
        v = min; st[min] = fr[min];
        for (w = 0, min = G->V; w < G->V; w++)
          if (st[w] == -1)
            {
              if (P < wt[w])
                { wt[w] = P; fr[w] = v; }
              if (wt[w] < wt[min]) min = w;
            }
      }
  }
```

图 20-8 显示了使用 Prim 算法构造一个示例 MST 的过程。图 20-9 对于一个较大的示例显示了 MST 的构造过程。

程序 20.3 是基于以下观察，可以将查找最小和更新操作放在一个循环中，并在其中检查所有非树边。在一个稠密图中，要更新从非树顶点到树的距离，可能需要检查的边的数目与 V 成正比，因此要检查所有非树边，找出距离树最近的边并不会带来过多的额外开销。但在稀疏图中，我们希望使用比 V 少得多的步数来执行每个操作。策略的关键是关注下一次要添加到 MST 中的潜在边的集合，称这个集合为*边缘集*（fringe）。边缘边的数目一般会小于非树边的数目，我们对算法重新描述如下。从一个指向边缘集中一个起始顶点的自环和空树开始，执行如下操作，直到边缘集为空为止：

> 从边缘集中将一条最小边移至树中。访问其指向的顶点。再将由该顶点到一个非树顶点的任何边置于边缘集中，如果边缘集中两条边都指向该同一个顶点，则用较短边取代较长的一条边。

根据这种形式化描述，Prim 算法只不过是一个广义的图搜索算法（见 18.8 节），其中边缘集是一个基于删除最小值（delete the minimum）操作的优先队列（见第 9 章）。我们称带有优先队列的广义图搜索为*优先级优先搜索*（priority-first search，PFS）。用边权值作为优先级，PFS 实现了 Prim 算法。

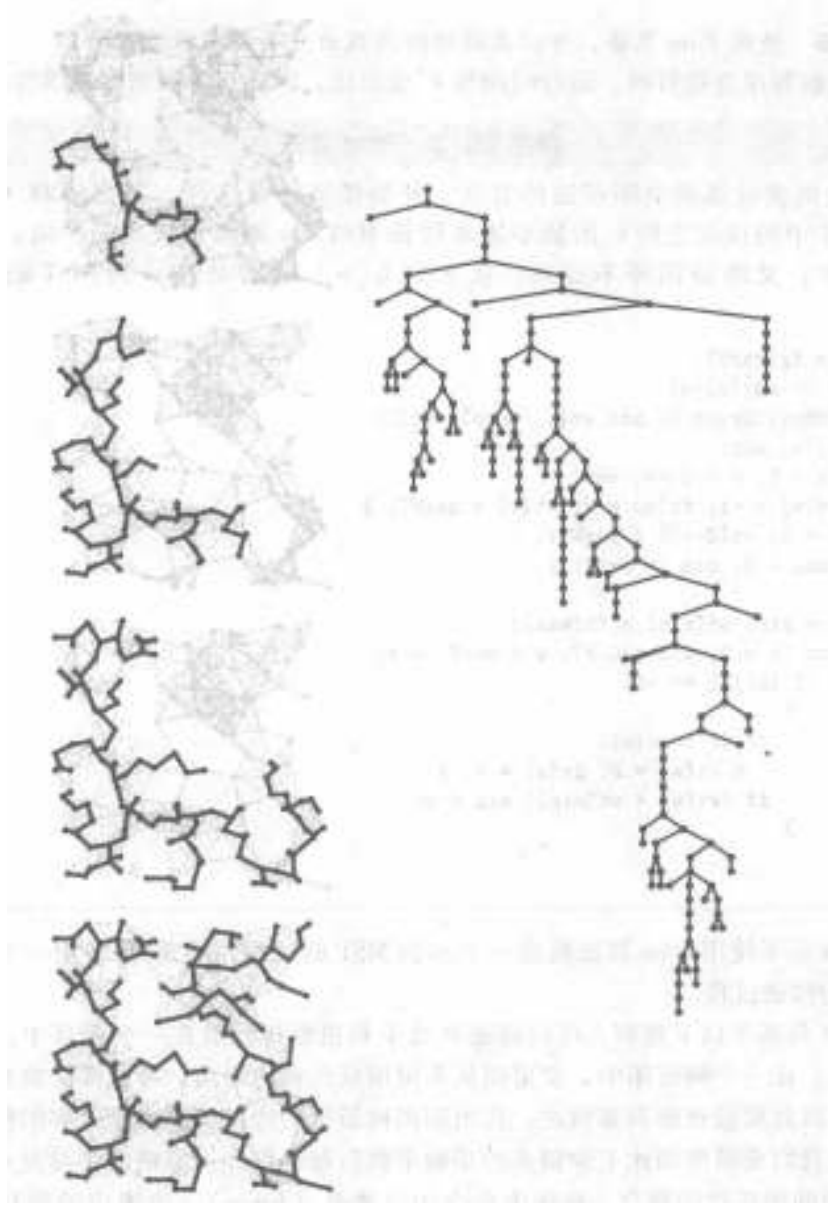


图 20-9　Prim 的 MST 算法

这个序列显示了随着 Prim 算法发现 MST 的 1/4、1/2、3/4 和所有边时，MST 是如何增长的（自上而下）。完全 MST 的有向表示在右图中显示。

这种描述包含了一个关键观察，我们准备建立与 18.7 节中实现 BFS 的联系。甚至更简单的一种方法是将依附于树顶点的所有边保存在边缘集中，并由优先队列机制找出最短的一条边，并忽略较长的边（见练习 20.37）。在 BFS 中已经看到，这种方法并不吸引人，因为边缘集数据结构会由于充斥着永远不会放入 MST 中的边变得混乱。边缘集的大小可能增长，与 E 成正比（还有维护此边缘集所涉及的辅助开销），而以上概述的 PFS 方法则保证了边缘集不会超过 V 个元素。

与一个通用算法的任何实现一样，有大量可用方法都可与优先队列 AOT 建立接口。一种方法是使用边的优先队列，与在程序 19.10 中的广义图搜索实现一样。程序 20.4 是本质

上与程序 18.10 等价的一种实现，但使用了基于顶点的方法，因而它可以使用 9.6 节的索引优先队列接口。我们将边缘集顶点（fringe vertex）区分出来，它是非树顶点的一个子集，通过边缘集连接到树顶点，与在程序 20.3 中一样，也保存同样的顶点索引数组 st、fr 和 wt。优先队列包含着每个边缘顶点的索引，通过第 2 和第 3 个数组，由其中的各个元素可以得到连接到边缘顶点的最近树顶点以及与该树的距离。

因为我们要寻找一个 MST，因而程序 20.4 假设图是连通的。它实际上找出的是包含顶点 0 的连通分量中的 MST。要扩展它来找出图中的最小生成森林，可以在程序 18.1 中增加一个循环（见练习 20.29）。

性质 20.7　使用 Prim 算法的 PFS 实现，其中使用堆来实现优先队列，可以在与 $E \lg V$ 成正比的时间内计算出一个 MST。

证明　算法直接实现了 Prim 算法的一般思想（将下一条连接 MST 中的顶点与不在 MST 中的顶点的最小边添加到 MST 中）。每个优先队列操作需要少于 $\lg V$ 步即可完成。使用删除最小值（delete the minimum）的操作来选择每个顶点；在最坏情况下，每条边可能需要一个改变优先级（change priority）的操作。 ■

优先级优先搜索是对宽度优先和深度优先的一个适当推广，因为这些方法也可以通过合适的优先级设置得出。例如，我们可以（人工地）使用一个变量 cnt，从而在将各个顶点置于优先队列中时，分别为之赋予一个唯一的优先级 cnt++。如果定义 P 为 cnt，则得到了前序编号和 DFS，因为新遇到的结点有最大的优先级。如果定义 P 为 V-cnt，则得到 BFS，因为老的结点有最大的优先级。这些优先级指派使优先队列的操作分别就像是一个栈或队列。等价性证明是学术上感兴趣的事情，优先队列的操作对于 DFS 和 BFS 来说并不是必要的。而且，如 18.8 节所讨论的，等价性的形式证明还需要准确地关注替换规则，从而得到与经典算法同样的顶点序列。

程序 20.4　优先级优先搜索（邻接表）

此程序是一种推广的图搜索方法，使用优先队列来管理边缘集（见第 18.8 节）。定义优先级 P，使得 ADT 函数 GRAPHpfs 实现了稀疏（连通）图的 MST ⌴ Prim 算法。其他优先级定义实现不同的图处理算法。

程序将最高优先级（最小权值）的边从边缘集移到该树中，然后检查对与新树顶点邻接的每条边，从而查看是否蕴含着要对边缘集改变。指向不在边缘集或该树中的顶点的那些边被添加到边缘集中；指向边缘顶点的较短边替换了对应的边缘边。

我们使用 9.6 节的优先队列 ADT 接口，并用子例程 PQdelmin 代替 PQdelmax，PQdec 代替 PQchange（强调通过减小优先级来改变优先级）。静态变量 priority 和函数 less 允许优先队列函数使用顶点名作为句柄，并比较此段代码中由 wt 数组维护的优先级。

```
#define GRAPHpfs GRAPHmst
static int fr[maxV];
static double *priority;
int less(int i, int j)
  { return priority[i] < priority[j]; }
#define P t->wt
void GRAPHpfs(Graph G, int st[], double wt[])
  { link t; int v, w;
    PQinit(); priority = wt;
    for (v = 0; v < G->V; v++)
```

```
    { st[v] = -1; fr[v] = -1; }
  fr[0] = 0; PQinsert(0);
  while (!PQempty())
    {
      v = PQdelmin(); st[v] = fr[v];
      for (t = G->adj[v]; t != NULL; t = t->next)
        if (fr[w = t->v] == -1)
          { wt[w] = P; PQinsert(w); fr[w] = v; }
        else if ((st[w] == -1) && (P < wt[w]))
          { wt[w] = P; PQdec(w); fr[w] = v; }
    }
}
```

我们将会看到，PFS 不只包含了 DFS、BFS 和 MST 的 Prim 算法，还包含了其他几种经典算法。各种算法只是其优先函数有所不同。因此，所有这些算法的运行时间依赖于优先队列 ADT 的性能。实际上，我们所得到的是一个通用的结果，不仅包含本节中所考察的 Prim 算法的两种实现，而且涵盖一大类的基本图处理算法。

性质 20.8 *对于所有图和所有优先级函数，在一个规模至多为 V 的优先队列中，可以利用 PFS 计算出一棵生成树，所需时间为线性时间加上与进行 V 次插入（insert），V 次删除最小值（delete the minimum），以及 E 次减小关键字（decrease key）操作所需时间成正比的时间。*

证明 由性质 20.7 的证明，可得此一般结果。我们必须检查此图中的所有边；因而也是“线性”的原因。此算法永远不会增加优先级（只会将优先级变得更小）；通过更准确地指定从优先队列 ADT 中所需的操作（即减小关键字（decrease key），不必是改变优先级（change priority）），可以增强关于性能的陈述。 ■

特别地，使用无序数组优先队列实现给出了稠密图的一种最优方法，其最坏情况下的性能与 Prim 算法（程序 20.3）的经典实现相同。也就是说，性质 20.6 和 20.7 是性质 20.8 的特例。贯穿全书，我们会看到大量其他只在优先函数和其优先队列实现的选择上有所不同的算法。

性质 20.7 是一个重要的一般结果：所述的时间界限是最坏情况下的上界，对于大量图处理问题，保证了其性能是最优（线性时间）的 lg V 倍之内。对于实际中遇到的很多图，这个结论有些悲观。有两点原因，第一，优先队列操作的 lg V 界限只对边缘集中的顶点数为与 V 成正比时成立，甚至只是一个上界。对于实际应用中真实的图而言，边缘集可能很小（见图 20-10 和 20-11），某些优先队列的操作需要的步数可能比 lg V 步要少。尽管这一点显而

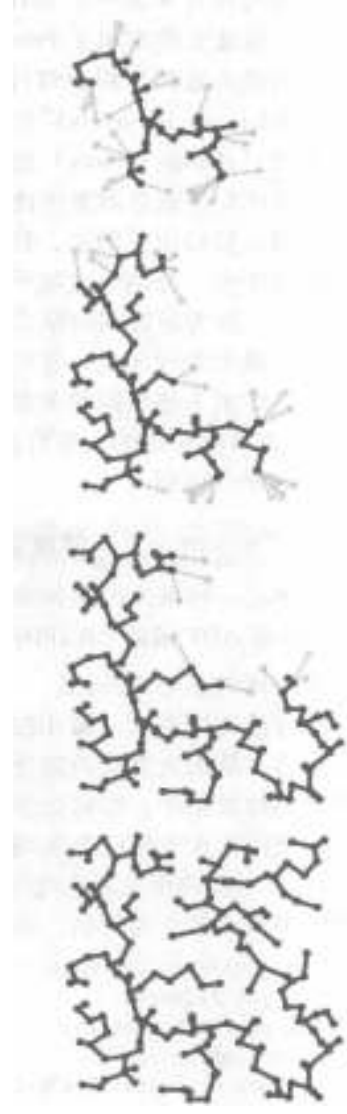


图 20-10 Prim MST 算法的 PFS 实现

使用 PFS，Prim 算法只处理最接近 MST 的顶点和边（灰色表示）。

易见，但是这一结果只占运行时间的一个小的常量因子。例如，边缘集不超过$\sqrt{V}$个顶点的证明对于这个界限的改进只是一个 2 倍因子。更重要的是，一般执行减小关键字（decrease key）的操作远少于 E 次，因为仅当找到一条指向边缘结点的边且该边权值小于当前指向边缘结点的已知边时才执行该操作。这种情况相对稀少：大多数的边都对优先队列没有影响（见练习 20.35）。将 PFS 看作实际上的一个线性算法是合理的，除非 $V \lg V$ 比 E 大得多。

优先队列 ADT 和广义图搜索抽象使我们很容易理解不同算法之间的关系。由于这些抽象（以及支持其使用的软件机制）是在出现以上基本方法多年之后才得以开发的，因此将算法与其经典描述联系起来就成为历史学家们的功课。然而，我们在研究文献或在其他资料中遇到的 MST 算法的描述时，了解有关历史的基本事实仍是很有用的，而且这样几个简单的抽象就可以将众多研究人员数十年以来所做的工作联系在一起，理解这是如何做到的对于证实其价值和能力也很有说服力。因此，我们将简要介绍这些算法的起源。

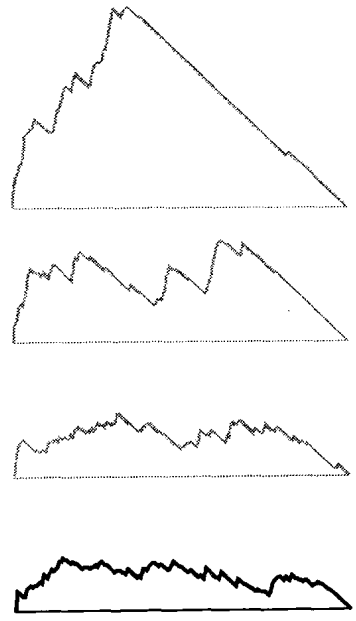


图 20-11　Prim MST 算法 PFS 实现的边缘集大小

下图显示了 PFS 处理图 20-10 中的示例时其边缘集大小。为了进行比较，上方画出了图 18-28 所对应的 DFS、随机搜索以及 BFS，以灰色显示。

对于稠密图的一种 MST 实现（基本上等价于程序 20.3）最早由 Prim 于 1961 年提出，而 Dijkstra 随后也独立地提出了一个 MST 实现。通常称之为 Prim 算法。然而，Dijkstra 的表示更具有一般性，因此有些学者将 MST 算法称为是 Dijkstra 算法的一个特例。然而，Jarnik 早在 1939 年就提出了这一基本思路，因此有些学者也称此方法为 Jarnik 算法，这样 Prim（Dijkstra）所做的贡献就成为找出对于稠密图的此算法的一个高效实现。随着优先队列 ADT 于 20 世纪 70 年代的使用，它直接应用于寻找稀疏图 MST：在与 $E \lg V$ 成正比的时间内可计算出稀疏图的 MST，这一事实之众所周知的，而不能归功于哪一个研究人员。在此以后，正如我们将在 20.6 节中讨论的那样，许多研究人员都在致力于找出高效的优先队列实现，这是找出稀疏图的 MST 算法的关键所在。

练习

▷ 20.30　对于本节开始提到的针对 V 个顶点的完全加权图的 Prim 算法的蛮力实现，分析其性能。提示：以下组合求和可能有用：$\sum_{1 \le k < V} k(V-k) = (V+1)V(V-1)/6$。

○ 20.31　对于所有顶点都有相同固定度 t 的图，回答练习 20.30。

○ 20.32　对于有 V 个顶点、E 条边的一般稀疏图，回答练习 20.30。因为运行时间依赖于边的权值和顶点的度，进行最坏情况下的分析。展示一组图，证实你的最坏情况下的界限。

20.33　按照图 20-8 的风格，显示使用 Prim 算法计算练习 20.21 中所定义的网的 MST 的结果。

● 20.34　描述 V 个顶点、E 条边的述一组图，使得 Prim 算法的 PFS 实现有最坏情况下的运行时间。

●● 20.35　开发一个用于生成 V 个顶点、E 条边的随机图的合理程序，使得 Prim 算法的 PFS 实

现（程序 20.4）的运行时间是非线性的。

20.36 转换程序 20.4，使其用于邻接矩阵图 ADT 实现。

○ 20.37 修改程序 20.4，使其像程序 18.8 那样工作，其中将依附树顶点的所有边保存在边缘集中。进行实验研究，对于各种类型的加权图（见练习 20.9～14），与程序 20.4 的实现进行比较。

● 20.38 提供 Prim 算法的一种实现，要求使用练习 17.51 中独立于表示的图 ADT。

20.39 假设使用优先队列实现来维持一个有序表。对于 V 个顶点、E 条边的图，分析其最坏情况下的运行时间（限于一个常量因子）。此方法在何种情况下适用？证实你的结论。

○ 20.40 如果有一条 MST 边，将它从图中删除将导致 MST 的权值增加，那么称此边为*关键边*（critical edge）。证明如何在与 $E \lg V$ 成正比的时间内找出图的关键边。

20.41 如果优先队列使用无序数组实现，进行实验研究，对于各种类型的加权图（见练习 20.9～14），比较程序 20.3 与程序 20.4 的性能。

● 20.42 对于各种类型的加权图（见练习 20.9～14），进行实验研究，从而确定使用索引－堆－竞赛优先队列实现而不是程序 9.12，对程序 20.4 的影响。

20.43 对于各种类型的加权图（见练习 20.9～14），进行实验研究来分析树权值（作为 V 的函数）。

20.44 对于各种类型的加权图（见练习 20.9～14），进行实验研究来分析最大边缘集规模（作为 V 的函数）。

20.45 对于各种类型的加权图（见练习 20.9～14），进行实验研究来分析树的高度（作为 V 的函数）。

20.46 对于各种类型的加权图（见练习 20.9～14），进行实验来研究练习 20.44 和 20.45 的结果对于起始顶点的依赖性。值得使用一个随机起始点吗？

● 20.47 编写一个客户端程序，显示 Prim 算法的动态图形动画过程。你的程序应该产生像图 20-10（见练习 17.55～17.59）的图形。使用随机欧几里得近邻图和网格图（见练习 20.11 和 20.13）来测试你的程序，在合理时间内使用能处理的尽可能多的点。

20.4 Kruskal 算法

Prim 算法是通过一次找出一条边来构建 MST，其中每一步都是要找到一条新边连向一棵不断增长的树。Kruskal 算法也是一次找出一条边来构建 MST；但是与前面有所不同，它是要找出连接两棵树的一条边，这两棵树处于一个 MST 子树的分布森林中，其中的 MST 子树将不断增长。我们从一个包括 V 个单个顶点树的退化森林开始，然后执行合并两棵树的操作（使用可能的最短边），直至只剩下一棵树：即 MST。

图 20-12 逐步显示了 Kruskal 算法的操作过程；图 20-13 则基于一个更大的示例展示了此算法的动态特征。不连通的 MST 子树森林将逐步演化为一棵树。边将按其长度顺序增加到 MST 中，所以组成此森林的顶点相互之间都通过相对短的边连接。在算法执行的任何时刻，每个顶点与其子树中的顶点（较之于并非处于其子树中的任何顶点）更为接近。

基于本书中考虑过的基本算法学工具，Kruskal 算法很容易实现。实际上，可以使用第 3 部分的排序算法将边按权值排序，以及第 1 章的任何连通性算法消除导致环的边！程序 20.5 是对图 ADT 根据这些思路关于 MST 函数的一个实现，其功能等价于本章考虑的 MST 的其他实现。此实现并不依赖于图的表示：它调用 ADT 函数，返回包含图中边的一个数组，然后由此数组来计算该 MST。

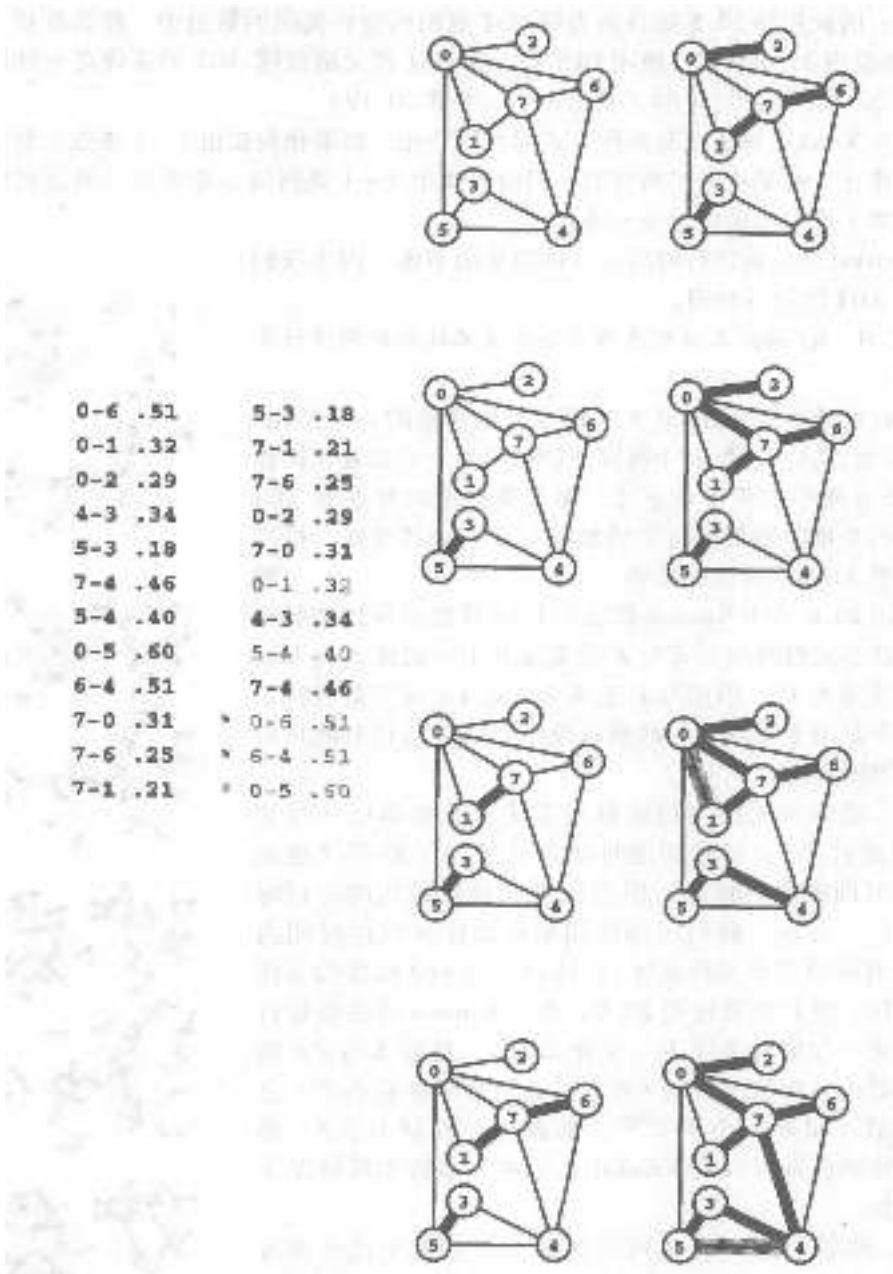


图 20-12　Kruskal MST 算法

给定一个任意顺序的图的边表（左列的边表）。Kruskal 算法的第一步是按照权值对其进行排序（右列的边表）。然后按照权值顺序扫描边表，并将不会创建环的边加入到 MST 中。首先加入边 5-3（最短的边），然后加入 7-1，再加入 7-6（左图），然后加入 0-2（右上图），再加入 0-7（右列顺数第 2 个图）。下一条最大权值的边 0-1 会创建一个环，因而不包含在 MST 中。没有加入到 MST 中的边在排序表中以灰色显示。然后，我们加入 4-3（右列顺数第 3 个图）。接下来，我们拒绝 5-4，因为它创建一个环，然后，我们加入 7-4（右下图）。一旦完成 MST，任何有较大权值的边都会导致环，因此将被拒绝（当在 MST 中加入了 $V-1$ 条边时就停止）。这些边在排序表中用星号表示。

有了 Kruskal 算法，边数组表示就是用于 MST 的一种自然选择，而不是 Prim 算法所用的父链接表示；因此程序 20.5 被封装为将 MST 返回到用户提供的数组中。修改此接口使其与程序 20.3 和程序 20.4 兼容，使用 DFS 来计算 MST 的父链接树 ADT 的实现是一件简单的事情，如在第 20.1 节中所讨论的（见练习 20.18 和 20.19）。

注意到在 Kruskal 算法中有两种方式都可以终止。如果我们找出 $V-1$ 条边，那么得到一棵生成树且停止。如果检查了所有边，但没有找出 $V-1$ 条树边，那么可以确定此图是不连通的，与在第 1 章中的做法完全一样。

分析 Kruskal 算法的运行时间是一件简单的事情，因为我们知道其组成 ADT 的运行时间。

性质 20.9 *Kruskal 算法可在与 $E \lg E$ 成正比的时间内计算出图的 MST。*

证明 此性质是对程序 20.5 进行更一般观察的一个结论。程序 20.5 的运行时间与对 E 个数排序的开销加上 E 次查找操作和 $V-1$ 次合并操作的开销成正比。如果我们使用标准的 ADT 实现，诸如归并排序和带有折半的加权合并 - 查找方法，排序的开销控制着 Kruskal 算法的开销。 ■

我们考虑 20.6 节中 Kruskal 算法和 Prim 算法的可比较的性能。此时，注意运行时间与 $E \lg E$ 成正比并不一定比 $E \lg V$ 要坏，因为 E 至多为 V^2，因而 $\lg E$ 至多为 $2 \lg V$。对于某些特定图的性能差异是由于实现上的性质以及是否实际运行时间接近这些最坏情况的界限。

实际上，我们可能使用快速排序或快速系统排序（可能也是基于快速排序）。尽管快速排序方法给出了排序（理论上）的二次时间的最坏情况，但它也可能导致最快的运行时间。实际上，一方面，我们可以使用基数排序在线性时间内完成排序（对权值要求某种条件），从而 E 次 find 操作的开销占据主导作用，然后调整性质 20.9，可得 Kruskal 算法的运行时间在某些关于权值的条件下（见第 2 章），是在 $E \lg^* E$ 的一个常量倍数内。回顾函数 $\lg^* E$ 是二元对数函数在小于 1 之前的迭代次数，如果 E 小于 2^{65536}，那么这个数就小于 5。换句话说，这些调整都可以使 Kruskal 算法在大多数实际情况下是线性有效的。

一般地，使用 Kruskal 算法找出 MST 的开销甚至比处理所有边的开销还要低。因为在图的（长）边的相当部分尚未被考虑时，MST 就已经完成了。考虑到这个因素，可以大大降低很多实际情况下的运行时间，将比 MST 的最长边还长的边排除在排序之外。完成这个目标的一种简单方法是使用优先队列，采用一个在线性时间内执行构造（construct）操作的实现，以及在对数时间内执行删除最小（delete the minimum）的操作。

例如，可以使用一个堆实现来达到这些性能特征，使用自底向上的构造（见 9.4 节）。具体地说，我们对程序 20.5 作如

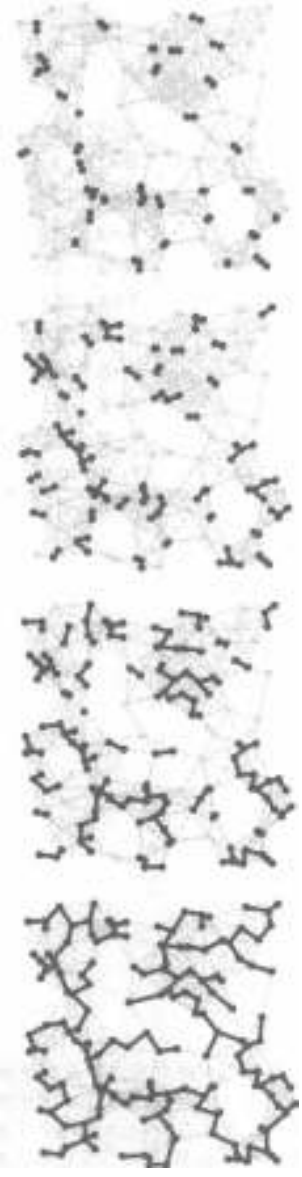


图 20-13 Kruskal MST 算法

此序列显示了 1/4、1/2、3/4 和整个 MST 的演化过程。

下的改变：首先，将对 sort 的调用改为 PQconstruct 调用，在与 E 成正比的时间内来构建堆。第二，改变内循环，使用 e = PQdelete 取出优先队列中最短的边，并将对 a[i] 的所有引用改为指向 e。

性质 20.10　*Kruskal 算法基于优先队列的版本可在与 $E + X \lg V$ 成正比的时间内计算出一个图的 MST，其中 X 是图中不大于 MST 中最长边的边数。*

证明　参见前述讨论，表明此开销为构建一个规模为 E 的优先队列的开销加上执行 X 个删除最小（delete the minimum）操作、X 个查找（find）操作，以及 $V-1$ 个合并（union）操作的开销。注意优先队列构造开销占据主导作用（且该算法为线性时间），除非 X 大于 $E/\lg V$。 ■

程序 20.5　Kruskal MST 算法

此实现使用第 6 章的排序 ADT 以及第 4 章的合并 - 查找 ADT，考虑按照边权值大小来找出一个 MST，并抛弃那些创建环的边，直到找出包含一个生成树的 $V-1$ 条边。

此实现被封装为一个函数 GRAPHmstE，并在用户提供的的边数组中返回一个 MST。在需要时，可添加一个包装器函数 GRAPHmst 来构建该 MST 的一个父链接（或某种其他表示），如在 20.1 节中所讨论的那样。

```
void GRAPHmstE(Graph G, Edge mst[])
  { int i, k; Edge a[maxE];
    int E = GRAPHedges(a, G);
    sort(a, 0, E-1);
    UFinit(G->V);
    for (i= 0, k = 0; i < E && k < G->V-1; i++)
      if (!UFfind(a[i].v, a[i].w))
        {
          UFunion(a[i].v, a[i].w);
          mst[k++] = a[i];
        }
  }
```

在基于快速排序的实现中，也可以应用同样的思想得到类似的好处。考虑在直接使用递归排序会发生何种情况，在 i 处进行划分，然后，递归地对 i 左边的子文件和 i 右边的子文件进行排序。注意到由算法的构造过程可知，在前一个递归调用完成之后，前 i 个元素是有序的（见程序 9.2）。这个显然的结果可以直接得到 Kruskal 算法的一种快速实现：如果将对边 a[i] 是否会导致环的检查放在这两个递归调用之间，那么就得到一个算法，由构造，该算法在完成第一个递归调用之后，就已经按有序顺序检查了前面的 i 条边！如果加上一个在找到 $V-1$ 条 MST 的边时就返回的检查，就得到一个只需要对计算 MST 所需的边进行排序的算法，加上几个涉及较大元素的划分步骤（见练习 20.52）。类似直接排序实现，此算法最坏情况下的运行时间为二次方，但从概率上可以保证，最坏情况下的运行时间将不会接近这个界限。而且，就像直接排序实现，这个程序很可能比基于堆的实现更快，因为它的内循环较短。

如果此图不是连通的，Kruskal 算法的部分排序版本将不具优势，因为要考虑所有的边。即使对于一个连通图，图中最长的边也可能在 MST 中，因而，Kruskal 方法的任何实现都会检查所有的边。例如，此图可能是由顶点的紧致类组成的，这一类都由短边连接在一起，只有一个相距较远的顶点由一条长边连向其中的一个顶点。虽然有这种异常的情况，部分排序

方法仍然是很有意义的，因为在应用时它能提供很大收益，如果有开销也是很小的。

历史方面的情况很重要，也有启发性。Kruskal 在 1956 年提出了他的算法，但多年来相关 ADT 的实现却没有深入研究，因此，实现诸如优先队列的性能特征直到 20 世纪 70 年代才得以充分理解。还有一些有趣的历史事件是，在 Kruskal 论文中提到了 Prim 算法的一个版本（见练习 20.54）以及 Boruvka 提到了这两种方法。对于稀疏图，Kruskal 算法的有效实现在 Prim 算法的实现之前，这是由于合并 - 查找算法（及排序）ADT 的使用在优先队列 ADT 之前。一般地，正如 Prim 算法的实现，Kruskal 算法所取得的进步主要归功于 ADT 性能的进展。另一方面，将合并 - 查找抽象应用到 Kruskal 算法以及将优先队列抽象应用到 Prim 算法一直是很多研究人员寻求这些 ADT 更好实现的主要动机。

练习

▷ **20.48** 按照图 20-12 的风格，使用 Kruskal 算法，显示练习 20.21 中所定义的网的 MST 的计算结果。

○ **20.49** 对于各种类型的加权图（见练习 20.9 ~ 14），进行实验研究来分析 MST 中最长边的长度和图中不大于这条边的边数的关系。

● **20.50** 开发合并 - 查找 ADT 的一种实现，使其在常量时间实现查找操作和在与 $\lg V$ 成正比的时间实现合并操作。

20.51 对于各种类型的加权图（见练习 20.9 ~ 14），进行实验研究，在 Kruskal 算法作为客户端时，将练习 20.50 中你的 ADT 实现与带有折半加权合并 - 查找（程序 1.4）进行比较。排除对边进行排序的时间开销，因此来研究此改变对总开销与使用合并 - 查找 ADT 有关的那部分开销的影响。

20.52 开发基于正文中所描述的思路的一个实现，其中将 Kruskal 算法与快速排序结合，从而对于每条边，一旦知道其所有的更小边都已经得到检查，则能够检查 MST 中边的合法性。

○ **20.53** 改编 Kruskal 算法来实现两个 ADT 函数，填写一个客户程序提供的顶点索引的数组，它将顶点分为 k 类，具有性质：在不同的类中，不存在长度大于 d 的边连接两个顶点。对于第一个函数，取 k 作为参数，并返回 d。对于第二类，取 d 作为参数，并返回 k。对于各种不同的 k 值和 d 值，使用随机欧几里得近邻图和网格图（见练习 20.11 和 20.13）测试你的程序。

20.54 开发一种基于对边进行预排序的 Prim 算法的实现。

● **20.55** 编写一个程序，完成对 Kruskal 算法（见 20.47）动态图形动画演示的过程。使用随机欧几里得近邻图和网格图（见练习 20.11 和 20.13）测试你的程序，使用在合理时间内所能处理的尽可能多的点。

20.5 Boruvka 算法

我们所考虑的下一个 MST 算法也是最古老的一个。类似 Kruskal 算法，要向 MST 子树的一个分布森林中增加边来构建 MST；但是这里是分步完成的，即每一步都增加数条 MST 的边。在每一步中，会找出连接每一棵 MST 子树与不同子树的最短边，再将所有这样的边都增加到 MST 中。

同样，由第 1 章的合并 - 查找 ADT 可得一个有效实现。对于此问题，很容易对接口进行扩展，构造一个客户程序可用的查找操作。我们使用此函数为每棵子树关联一个索引，因而可以很快地知道一个给定的顶点属于哪棵子树。有了这种能力，我们可以高效地实现

Boruvka 算法所需的各种操作。

首先，要维护一个顶点索引的数组，对于每个 MST 子树，它能识别出它的最近邻。然后，对图中的每条边执行如下操作：

- 如果此边连接同一棵树中的两个顶点，则抛弃它。
- 否则，检查由该边所连接的这两棵树之间的最近邻距离，如果存在最近邻则更新距离。

在对图中所有边做此扫描后，最近邻数组中就有了连接子树的所需信息。对于每个顶点索引，执行一个 union 操作将该顶点与它的最近邻连接起来。在下一步中，抛弃目前连通的 MST 子树中连接其他顶点对的所有更长的边。图 20-14 和图 20-15 展示了基于示例算法的这个过程。

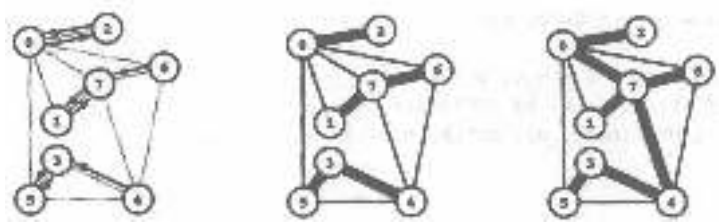


图 20-14　Boruvka MST 算法

左图显示了从每个顶点到其最近邻的所有有向边。这些边显示了 0-2、1-7 和 3-5 每条都是依附于其顶点的最短边。6-7 是 6 的最短边，且 4-3 是 4 的最短边。这些边都属于该 MST，并且构成 MST 子树的森林（中图），类似 Boruvka 算法的第一步。在第二步中，该算法添加 0-7 和 4-7 后就完成 MST 的计算（右图），其中 0-7 边是依附于它所连接的子树中的顶点的最短边，4-7 边是依附于下图的子树中的顶点的最短边。

程序 20.6 是 Boruvka 算法的直接实现。该实现被封装为使用边数组的一个函数 GRAPHmstE，可由构建其他表示的 GRAPHmst 函数调用，类似 20.1 节中的讨论。有三个主要因素使此实现非常高效：

- 每个查找操作的开销基本上是常量。
- 每一步都会减少森林中 MST 子树的数目。
- 在每一步中，有大量的边被抛弃掉。

很难对上述的三个因素进行量化，但容易建立如下的界限。

	0	1	2	3	4	5	6	7
initial	0	1	2	3	4	5	6	7
stage 1	0	1	0	3	3	3	1	1
stage 2	1	1	1	1	1	1	1	1

图 20-15　在 Boruvka 算法中的合并 - 查找数组

此图描述了与图 20-14 中的示例所对应的合并 - 查找数组中的内容。初始时，每个元素包含自己的索引，表示单个顶点组成的森林。第一步之后，有 3 个分量，用索引 0、1 和 3 表示（对于这个规模极小的示例，其合并 - 查找树都是平的）。第二步之后，只有一个分量，用 1 表示。

程序 20.6　Boruvka MST 算法

Boruvka MST 算法的实现使用了第 4 章中的合并 - 查找 ADT，并为所构建的 MST 子树关联索引。它假设查找操作（对于属于每个子树的所有顶点返回一个索引）是在接口中。

每一步都对应着检查所有剩余的边；连接不同分量中的顶点的边被保留到下一步中。数组 a 用于存放尚未被抛弃且尚未在 MST 中的边。索引 N 用于存放为下一步所保留的边（代码在每一步后都将 E 重置为从 N 开始），且索引 h 用于访问要被检查的下一条边。每个分量的最近邻被保存在数组 nn 中，以 find 分量编号作为索引。在每一步最后，我们将每个分量与它的最近邻合并，并将最近邻边添加到 MST 中。

```
Edge nn[maxV], a[maxE];
void GRAPHmstE(Graph G, Edge mst[])
```

```
{ int h, i, j, k, v, w, N; Edge e;
  int E = GRAPHedges(a, G);
  for (UFinit(G->V); E != 0; E = N)
    {
      for (k = 0; k < G->V; k++)
        nn[k] = EDGE(G->V, G->V, maxWT);
      for (h = 0, N = 0; h < E; h++)
        {
          i = find(a[h].v); j = find(a[h].w);
          if (i == j) continue;
          if (a[h].wt < nn[i].wt) nn[i] = a[h];
          if (a[h].wt < nn[j].wt) nn[j] = a[h];
          a[N++] = a[h];
        }
      for (k = 0; k < G->V; k++)
        {
          e = nn[k]; v = e.v; w = e.w;
          if ((v != G->V) && !UFfind(v, w))
            { UFunion(v, w); mst[k] = e; }
        }
    }
}
```

性质 20.11 Boruvka 算法计算图的 MST 的运行时间为 $O(E \lg V \lg^* E)$。

证明 由于每一步的森林中的数目均减半，所以步数不会大于 $\lg V$，每一步的时间则最多与 E 个查找操作的开销成正比，它小于 $E \lg^* E$，或在某些实际情况下是线性的。 ■

性质 20.11 中给出的运行时间是一个保守的上界。因为它并没有考虑每一步中大量减少的边的数目。查找操作在最初的几遍中花费常量的时间，而在后面几遍的搜索中，仅有少量的边。实际上，对于很多图，边数与顶点的数目呈指数减小，而且总运行时间与 E 成正比。例如，如图 20-16 所示，该算法仅在 4 步内就能够找出这个较大示例图的 MST。

可能去掉 $\lg^* E$ 这个因子来使 Boruvka 算法理论上的运行时间的界限降低到与 $E \lg V$ 成正比，方法是使用一个双向链表而不是合并和查找操作来表示 MST 的子树。然而，这种改进在实现上更为复杂，而且潜在性能改进也是微乎其微，在实际中使用很可能不值得考虑（见练习 20.61 和 20.62）。

我们已经提到，Boruvka 算法是我们所考虑的最古老的算法：它最初在 1924 年是为电力分布应用而设计的。此方法在 1961 年经 Sollin 重新发现；后来由于其有效的渐近性能作为 MST 算法的基础得到关注，另外作为并行 MST 算法的基础也得到关注。

练习

▷ 20.56 按照图 20-14 的风格，使用 Boruvka 算法，显示练习 20.21 中所定义的网的 MST 的计算结果。

○ 20.57 程序 20.6 为什么能够在执行合并操作之前进行查找检查？提示：考虑等长的边。

○ 20.58 程序 20.6 为什么能够在执行合并操作之前检查 v != G->V?

● 20.59 描述一组 V 个顶点、E 条边的图，使得在 Boruvka 算法执行的每一步中所剩下的边数足够多，导致最坏情况下的运行时间。

20.60 开发一种基于对边进行预排序的 Boruvka 算法的实现。

○ 20.61 开发 Boruvka 算法的一个实现，其中使用双向循环链表表示子树，从而在每一步中，可

以在与 E 成正比的时间内完成子树合并和重命名操作(因而不需要等价关系 ADT)。

○ **20.62**　对于各种类型的加权图(见练习 20.9 ~ 14),进行实验研究来比较练习 20.61 中的 Boruvka 算法实现与正文(程序 20.6)中的 Boruvka 算法实现。

● **20.63**　对于各种类型的加权图(见练习 20.9 ~ 14),进行实验研究,用表格列出 Boruvka 算法中的步数与每步中所处理的边数。

20.64　开发 Boruvka 算法的一个实现,它在每步中构造一个新图(每个顶点表示森林中的一棵树)。

● **20.65**　编写一个程序,完成对 Boruvka 算法(见 20.47 和 20.55)动态图形动画演示的过程。使用随机欧几里得近邻图和网格图(见练习 20.11 和 20.13)测试你的程序,使用在合理时间内所能处理的尽可能多的点。

20.6　比较与改进

表 20-1 总结了我们所考虑过的基本 MST 算法的运行时间;表 20-2 给出了对这些算法进行比较的实验研究结果。由这些表格可得,Prim 算法的邻接矩阵实现是对于稠密图的所选方法,对于中等密度的图,其他方法的性能都在最佳可能结果(取出边所需时间)的一个小的常量因子范围内。对于稀疏图,Kruskal 方法本质上将问题归约为排序问题。

简而言之,我们可以将 MST 问题看作针对实际问题的"已解决的"问题。对于大多数图,找出 MST 的开销只是比取出图的边的开销稍高一些。除了那些极端稀疏的巨型图,这个规则均适用,但是对已知最好算法在性能上的可用改进最好能达到大约 10 倍。表 20-2 中的这些结果依赖于产生图所使用的模型,但它们也适用于很多其他图模型(例如,练习 20.74)。而且,理论上的结果并不能否认存在对于所有图的保证线性时间算法的可能性;这里,我们将简要介绍对这些方法的改进实现所做的广泛研究。

首先,很多研究是开发更好的优先队列实现。Fibonacci 堆(Fibonacci heap)数据结构是一种扩展的二项队列,其减小关键字(decrease key)操作为常量时间,删除最小(delete the minimum)操作为对数时间,达到了理论上的最优性能。由性质 20.8 可知,其行为对于 Prim 算法而言,运行时间与 $E + V \lg V$ 成正比。Fibonacci 堆比二项队列更为复杂,在实际中不太实用,但某些更简单的优先队列实现可有类似的性能特征(见第 5 部分参考文献)。

一种有效的方法是在优先队列的实现中使用基数方法。这种方法的性能往往等价于 Kruskal 方法中基数排序的性能,甚至等价于对 20.4 节中所讨论的在部分 - 排序方法中使用一个基数快速排序方法。

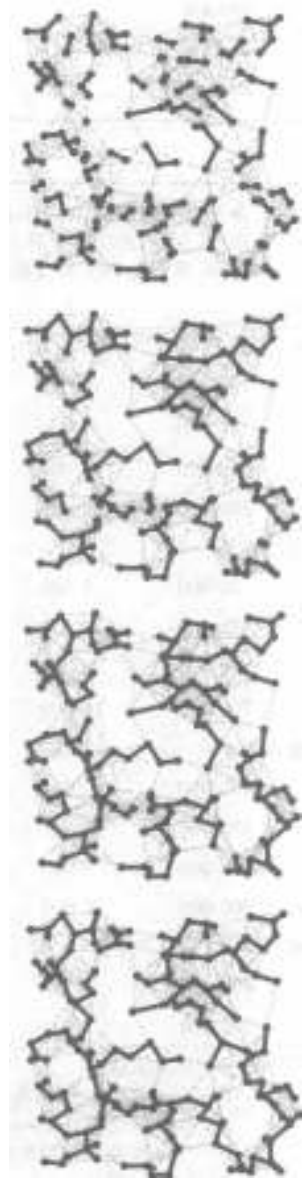


图 20-16　Boruvka MST 算法

对于这个例子,只用 4 步就演化出 MST(上图到下图)。

表 20-1 MST 算法的开销

此表总结了本章中所考虑的各种 MST 算法的开销(最坏情况下的运行时间)。这里的公式基于以下假设:存在 MST(蕴含着 $E \geqslant V-1$)且存在 X 条边不大于 MST 中的最长边(见性质 20.10)。这些最坏情况下的界限可能太保守,而不能预测真实图的性能。在很多实际情况下,算法的运行时间都为线性的。

算 法	最坏情况开销	注 释
Prim(标准)	V^2	对于稠密图是最优的
Prim(PFS,堆)	$E \lg V$	保守上界
Prim(PFS,d-heap)	$E \log_d V$	除了极端稀疏图外,是线性算法
Kruskal	$E \lg E$	排序开销占据主导作用
Kruskal(部分排序)	$E + X \lg V$	开销依赖于最长边
Boruvka	$E \lg V$	保守上界

表 20-2 MST 算法实验研究

对于各种密度的随机加权图,此表显示了各种算法找出 MST 的相对时间。对于低密度的图,Kruskal 算法是最好的,因为它相当于一个快速排序。对于高密度的图,Prim 算法的经典实现是最好的,因为它不会带来表处理的开销。对于中密度的图,Prim 算法的 PFS 实现是检查图中每条边所需时间的一个小的倍数。

E	V	C	H	J	P	K	K*	e/E	B	e/E
密度为 2										
20 000	10 000	2	22	27		9	11	1.00	14	3.3
50 000	25 000	8	69	84		24	31	1.00	38	3.3
100 000	50 000	15	169	203		49	66	1.00	89	3.8
200 000	100 000	30	389	478		108	142	1.00	189	3.6
密度为 20										
20 000	1 000	2	5	4	20	6	5	0.20	9	4.2
50 000	2 500	12	12	13	130	16	15	0.28	25	4.6
100 000	5 000	14	27	28		34	31	0.30	55	4.6
200 000	10 000	29	61	61		73	68	0.35	123	5.0
密度为 100										
100 000	1 000	14	17	17	24	30	19	0.06	51	4.6
250 000	2 500	36	44	44	130	81	53	0.05	143	5.2
500 000	5 000	73	93	93		181	113	0.06	312	5.5
1 000 000	10 000	151	204	198		377	218	0.06	658	5.6
密度 v/2.5										
400 000	1 000	61	60	59	20	137	78	0.02	188	4.5
2 500 000	2 500	597	409	400	128	1 056	687	0.01	1 472	5.5

说明:C 只取出边
H Prim 算法(邻接表/索引堆)
J Prim 算法的 Johnson 版本(d-叉堆优先队列)
P Prim 算法(邻接矩阵表示)
K Kruskal 算法
K* Kruskal 算法的部分排序版本
B Boruvka 算法
e 被检查的边(合并操作)

还有一种简单方法早期由 D. Johnson 于 1977 年提出，也是最有效的一种方法：用 d-叉堆实现 Prim 中的优先队列，而不是用标准二叉堆（见图 20-17）。对于这种优先队列的实现，减小关键字（decrease key）操作少于 $\log_d V$ 步，删除最小（delete the minimum）操作所需时间与 $d \log_d V$ 成正比。由性质 20.8，这种行为导致 Prim 算法的运行时间与 $Vd \log_d V + E \log_d V$ 成正比，对于非稀疏图而言则是线性的。

性质 20.12　给定一个有 V 个顶点、E 条边的图，令 d 表示密度 E/V。如果 $d<2$，那么 Prim 算法的运行时间与 $V \lg V$ 成正比。否则，可以使用 $\lceil E/V \rceil$ - 叉堆表示优先队列，从而使最坏情况下运行时间改进 $\lg(E/V)$ 倍。

证明　接着前一段的讨论，步数为 $Vd \log_d V + E \log_d V$，因此运行时间至多与 $E \log_d V$ 成正比，即 $(E \lg V) / \lg d$。 ■

在 E 与 $V^{1+\varepsilon}$ 成正比时，性质 20.12 导致的最坏情况下的运行时间为 E/ε，对于任何常量 ε，该值是线性的。例如，如果边数与 $V^{3/2}$ 成正比，那么开销小于 $2E$；如果边数与 $V^{3/4}$ 成正比，那么开销小于 $3E$；而且如果 $V^{5/4}$ 成正比，那么开销小于 $4E$。对于一个有百万个顶点的图，开销小于 $6E$，除非密度小于 10。

用这种方法试图最小化最坏情况下运行时间界限需要认识到，开销的 $Vd \log_d V$ 部分是不可避免的（对于删除最小操作，当向下移动时必需检查堆中的 d 个后继），但 $E \lg d$ 部分不太可能达到（因为更多的边将不需要更新优先队列，类似在性质 20.8 后面的讨论所示）。

对于表 20-2 实验中的典型的图，减小 d 对运行时间没有任何影响，而使用一个大的 d 值则会使实现少有所减慢。然而，它对于最坏情况性能提供了少许保证，使得该方法很有价值，因为它易于实现。原则上，我们可能微调实现，挑出某种类型图的最好 d 值（选择不会是算法减慢的最大 d 值），但是一个小的固定值（比如说 4、5 或 6）会很好，除了具有某些特征的特定类型的巨型图之外。

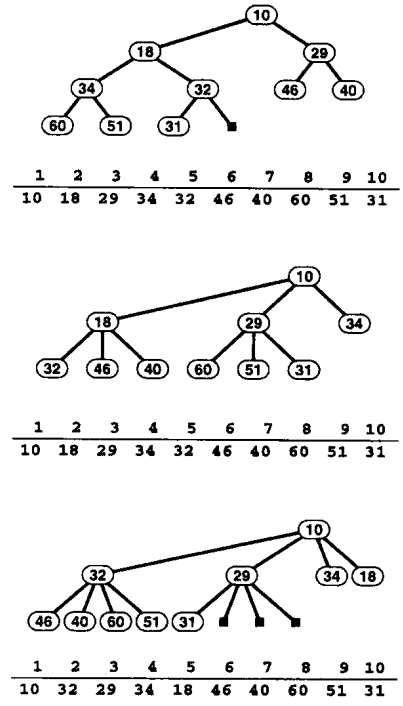


图 20-17　2 叉堆、3 叉堆和 4 叉堆

在将标准二叉堆排序的完全树存放在数组中时（上图），我们使用隐式链接来表示结点 i 连至树中下方的子结点 $2i$ 和 $2i+1$，以及树中上方的父结点 $i/2$。在 3 叉堆中（中图），i 的隐式链接连向其子结点 $3i-1$，$3i$ 和 $3i+1$ 及父结点 $\lfloor (i+1)/3 \rfloor$；在 4 叉堆中（下图），i 的隐式链接连向其子结点 $4i-2$，$4i-1$，$4i$ 和 $4i+1$ 及父结点 $\lfloor (i+2)/4 \rfloor$。增加隐式堆实现中的分支因子在应用中很有价值，类似 Prim 算法，需要大量减小关键字的操作。

程序 20.7　多路堆 PQ 实现

堆 PQ 实现（见程序 9.5）的这些 fixUp 和 fixDown 函数维持一个 d-叉堆；因此，删除最小操作所需时间与 d 成正比，但 decrease key 操作需要的步数少于 $\log_d V$。对于 $d=2$，这些函数分别等价于程序 9.3 和 9.4。

```
fixUp(Item a[], int k)
  {
    while (k > 1 && less(a[(k+d-2)/d], a[k]))
      { exch(a[k], a[(k+d-2)/d]); k = (k+d-2)/d; }
```

```
  }
fixDown(Item a[], int k, int N)
  { int i, j;
    while ((d*(k-1)+2) <= N)
      { j = d*(k-1)+2;
        for (i = j+1; (i < j+d) && (i <= N); i++)
          if (less(a[j], a[i])) j = i;
        if (!less(a[k], a[j])) break;
        exch(a[k], a[j]); k = j;
      }
  }
```

对于稀疏图使用 d-叉堆并不是很有效。因为 d 必须为一个大于等于 2 的整数，这个条件隐含着不能产生小于 $V \lg V$ 的渐近运行时间。如果密度是较小的常数，那么线性时间 MST 算法的运行时间必须与 V 成正比。

开发在线性时间内计算稀疏图的 MST 的实用算法的目标仍然是难以解决的。已对 Boruvka 算法的各种变形做了大量的研究，作为稀疏图的渐近线性时间 MST 算法的基础（见第 5 部分参考文献）。这些研究使得最终有可能得到一种实用的线性时间 MST 算法，甚至表明存在随机线性时间的算法。尽管这些算法一般会相当复杂，但在实际中，其中有些简化的版本也是很有用的。同时，在大多数实际情况下，可以使用这里所考虑的基本算法在线性时间内计算 MST，对某些稀疏图可能要增加一个额外的因子 $\lg V$。

练习

○ **20.66** ［V. Vyssotsky］开发 20.2 节中所讨论算法的一种实现，通过每次增加边并删除所构成环中的最长边来构建 MST（见练习 20.28）。使用 MST 子树森林的父链接表示。提示：在树中遍历路径时将指针反向。

20.67 对于各种类型的加权图（见练习 20.9～14），进行实验研究，将练习 20.66 中你的实现与 Kruskal 算法进行比较。检查考虑边的随机顺序是否对结果有影响。

● **20.68** 对于每次只能有 V 个顶点装入主存的大型图，描述如何找出该图的 MST。

○ **20.69** 开发优先队列的一个实现，其中删除最小操作和查找最小（find the minimum）操作都是常量时间的操作，减少关键字所需时间与优先队列大小的对数成正比。对于各种类型的加权图（见练习 20.9～14），在使用 Prim 算法来找出稀疏图的 MST 时，将你的实现与 4-叉堆进行比较。

20.70 开发一个实现，将 Boruvka 算法进行推广，维护一个包含 MST 子树森林的广义队列。（使用程序 20.6 对应着使用 FIFO 队列）。对于各种类型的加权图（见练习 20.9～14）进行实验，并与其他广义队列实现进行比较。

● **20.71** 开发随机连通立方图（每个顶点的度为 3）的一个生成器，其中边上的权值为随机权值。针对这种情况，对于所讨论的 MST 算法进行微调，然后确定哪一种算法是最快的。

○ **20.72** 对于 $V = 10^6$，画出使用 d-叉堆的 Prim 算法开销的上界与 E 的比率（作为密度 d 的函数）。d 的取值为 1～100。

○ **20.73** 由表 20-2 可得，对于低密度图，Ksuskal 算法的标准实现比部分排序实现要快得多。解释这种现象。

● **20.74** 按照图 20-2 的风格，对于含有高斯权值的随机完全图（见练习 20.12），进行实验研究。

20.7　欧几里得 MST

假设给定平面上的 N 个点，我们想要找出连接所有点的最短线段集合。这个集合问题称为欧几里得 MST 问题（见图 20-18）。求解此问题的一种方法是构建一个 N 个顶点、$N(N-1)/2$ 条边的完全图，每对顶点都有一条边连接，边上权值为对应点之间的距离。那么，可以使用 Prim 算法在与 N^2 成正比的时间内找出该 MST。

一般说来，这种解决方案太慢。欧几里得问题有点不同于我们一直考虑的图问题，因为所有的边是隐含定义的。输入规模只与 N 成正比，因而上面所描述的算法是此问题的一个二次（quadratic）时间的算法。研究已经证明还可能有更好的方法。几何结构使完全图中大多数的边都与问题不相关，而且在构造最小生成树之前，并不需要将大多数的边增加到这个图中。

性质 20.13　*我们可以在与 $N \log N$ 成正比的时间内找到 N 个点的欧几里得 MST。*

证明　这一点是关于平面上点的两个基本事实的直接结果（在第 7 部分将详细讨论）。首先，根据定义，称为 Delauney 三角（Delauney triangulation）的图含有 MST。其次，Delauney 三角是一个边数与 N 成正比的平面图。■

这样，原则上讲，可以在与 $N \log N$ 成正比的时间内计算出 Delauney 三角，然后运行 Kruskal 算法或优先级优先搜索方法来找出欧几里得 MST。所用时间与 $N \log N$ 成正比。但是写一个计算 Delauny 三角的程序即使是对于有经验的程序员也是一种挑战，因为对于这个问题，这种方法在实际中过于复杂。

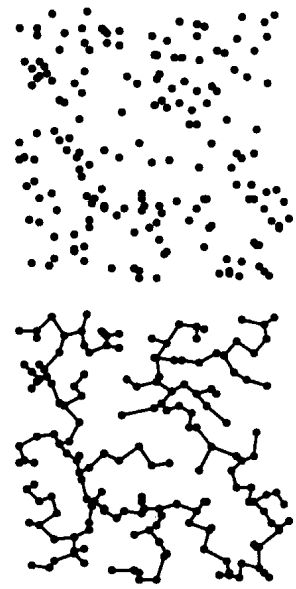


图 20-18　欧几里得 MST

给定平面上 N 个点的集合（上图），欧几里得 MST 是将这些点连接在一起的最小线段集合（下图）。此问题不只是一个图处理问题，因为我们需要利用点的全局几何信息来避免处理连接这些点的所有 N^2 个隐含边。

其他的方法则源于将在第 7 部分考虑的几何算法。对于随机分布的点，我们可以将平面划分成若干方形，每个方形中都包含大约 $\lg N/2$ 个点，类似于程序 3.20 中的最近点的计算。然后，即使仅将各点与其相邻正方形中的点连接的边包含在图中，也很有可能（但并不能保证）得到最小生成树中的所有边；在这种情况下，我们可以运行 Kruskal 算法或 Prim 算法的 PFS 实现来有效地完成这项工作。图 20-10、图 20-13、图 20-16 以及类似图中的所用示例都是用这样建方法创建的（见图 20-19）。或者，还可以基于使用近邻算法开发 Prim 算法的另一个版本，来避免更新远距的顶点。

基于求解此问题的所有可能选择，以及求解一般 MST 可能存在的线性算法，重要的说明是存在一个我们可能做到的简单最佳下界。

性质 20.14　*找出 N 个点的欧几里得 MST 并不比对 N 个数进行排序简单。*

证明　给定一组待排序的数字列表，将这个表转化为一个点列表，其中 x 坐标取自数字表中对应的数，y 坐标取 0。找出这个点列表的 MST。然后（与 Kruskal 算法所作的一样），将这些点放到图的 ADT 中，并且运行 DFS 来产生一棵生成树，从最小 x 坐标的那个点开始。这棵生成树相当于对这些数的进行链表排序；因此，就解决了排序问题。■

准确地表述这个下界很复杂，因为这两个问题所用的基本操作（对于排序问题是比较坐标，对于 MST 问题则是比较距离）是不同的，而且还有可能使用诸如基数排序和网格等方

法。然而，可以对此边界做以下解释，它表示在进行排序时，应当将使用 $N \lg N$ 次比较的欧几里得 MST 算法认为是最优的，除非利用了坐标的数值性质，在这种情况下可以期望达到线性时间。

研究欧几里得 MST 所引出的图算法和几何算法之间的关系是很有趣的。我们可能遇到的很多实际问题要么可以形式化为几何问题，要么可以形式化为图问题。如果对象的物理位置是一个主要特征，那么就可以调用第 7 部分的几何算法；但如果对象之间的互连关系非常重要，那么本节的图算法可能更胜一筹。

欧几里得 MST 似乎落入这两种方法的交界处（输入涉及几何特征，输出涉及互连特征），开发欧几里得 MST 的简单、直接方法仍然是一个困难的目标。在第 21 章里，我们看到一个同样落入这个交界处的类似问题，但对于该问题，欧几里得方法本质上允许比解决相应图问题更快的算法。

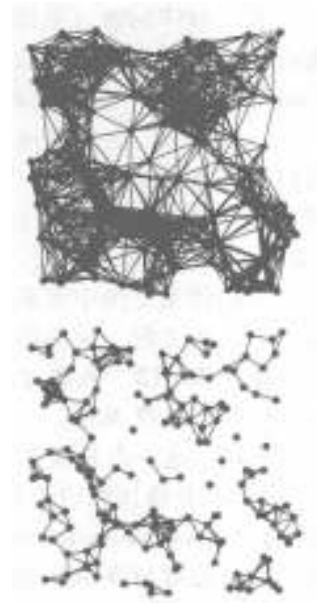


图 20-19 欧几里得近邻图

计算欧几里得 MST 的一种方法是生成一个图，其中由边连接距离 d 内的每一对点，如图 20-8 等的图所示。然而，如果 d 太大（上图），这种方法也产生太多条边，而且，如果 d 小于 MST 中的最长边，这种方法也不能保证能有连接所有点的边（下图）。

练习

▹ **20.75** 给出一个反例来说明以下方法对于找出欧几里得 MST 不可行："对这些点按照其 x 坐标排序，然后找出前一半点和后一半点的最小生成树，再找出连接它们的最短边。"

○ **20.76** 开发一种 Prim 算法的快速版本，对于平面上的一个均匀分布的点集，忽略远距点直到接近它们的一棵树为止，计算出欧几里得 MST。

●● **20.77** 开发一个算法，给定平面上的 N 个点集，找出一个基数与 N 成正比的边集，其中必然包含 MST 而且易于计算，使得能够为该算法开发一个简洁而高效的实现。

○ **20.78** 在一个单位正方形中，给定一个 N 个点的随机集合（均匀分布），进行实验确定一个值 d（保留两位小数），使得距离 d 内的各对顶点所定义的边集中 99% 包含 MST。

○ **20.79** 如果每个点的坐标由均值为 0.5、标准方差为 0.1 的高斯分布得到，对这些点做练习 20.78。

● **20.80** 描述对于稀疏欧几里得图如何改进 Kruskal 算法和 Boruvka 算法的性能。

第21章 最短路径

加权有向图中的每条路径都有相关的路径权值（path weight），其值为那条路径上的边权值之和。这个基本的度量允许我们对诸如“找出两个给定顶点之间权值最小的路径”这样的问题形式化。这些最短路径问题（shortest-paths problem）就是本章的主题。最短路径问题不仅在很多应用中非常直观，而且还带我们进入一个强大和通用的领域，使我们可以寻求解决包含大量特定应用的一般问题的高效算法。

本章所考虑的几个算法与第17～20章所考察的各种算法直接相关。基本的图搜索范型可以直接应用，还有一些在第17章和第19章论述图的连通性时所使用的特定机制，可以提供作为解决最短路径问题的基础。

为简洁起见，我们称加权有向图为网（network）。图21-1使用标准表示显示了一个示例网。从我们在20.1节所考虑的无向图表示所对应的函数，派生处理网所需的基本ADT函数是一件简单的事情。只要保留每条边的一种表示，与从17章的无向图表示派生第19章中有向图表示完全一样（见程序20.1和20.2）。从第17章和第18章中的非加权无向图，第19章中的非加权有向图，或第20章中的加权无向图的ADT可以派生ADT（见练习21.9），在需要各种类型图的应用或系统中，通过这些派生的ADT来定义网的ADT，是软件工程书中的一个练习。

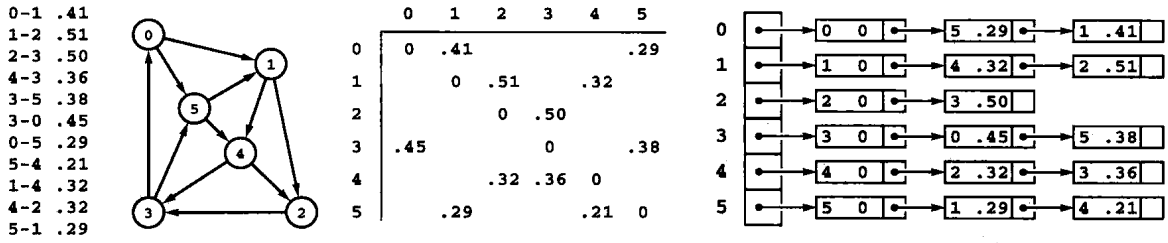


图21-1 示例网和表示

此网（加权图）用四种表示显示：边表，绘制图形，邻接矩阵和邻接表（从左到右）。类似于MST算法中的做法，常常使用与其绘制长度成正比的边权值，但并不要求遵守这一规则。

因为大多数的最短路径算法都可处理任意的非负权值（负权值确实会带来特殊的挑战）。邻接矩阵表示和邻接表表示包含了带有每边表示的权值，与在加权无向图中一样。该邻接矩阵不是对称的，且邻接表中包含每条边的一个结点（类似于非加权有向图）。不存在的边在矩阵中用一个观察哨值（图中为空）表示出，在表中则根本不会出现。出现了长为0的自环，是因为它们能够简化最短路径算法的实现。为简便起见，左边的边表中省略了自环，而且当创建一个邻接矩阵或邻接表表示时，按照约定增加这些自环时会指示出这种典型情况。

处理网时，在所有的表示中保留自环很方便。这个约定使算法可以灵活地使用最大权值观察哨来指示一个顶点由自身不可达。在我们的示例中，使用权值为0的自环，尽管正权值自环在应用中的确也是有意义的。很多应用还需要平行边，也许平行边还带有不同权值。就像在20.1节中提到的那样，各种忽略或组合这些边的选项适合于不同的应用。在这一章里，为了简明起见，示例中都不使用平行边，在邻接矩阵表示中也不允许平行边；也不检查平行边或删除邻接表中的平行边。

我们在第19章中所考虑的有向图的所有连通性质在网中也很重要。在那一章中，希望知道是否可以从一个顶点到达另一个顶点；在这一章里，我们考虑权值，希望找出从一个顶点到另一个顶点的最佳方式。

定义 21.1　网中两个顶点 s 和 t 之间的最短路径（shortest path）是一条从 s 到 t 的简单有向路径，具有性质：不存在其他有更小权值的路径。

此定义很简洁，但这种简洁性掩盖了值得深入讨论的内容。首先，如果 t 不能由 s 可达，那么根本不存在路径。因此也就不存在最短路径。为方便起见，我们所考虑的算法往往将这种情况认为是等价于从 s 到 t 存在一条有无限权值的路径。其次，类似在 MST 算法中所做的那样，示例中使用的网中边权值与边长度成正比，但在定义中没有这个要求，而且算法（除了 21.5 中的算法）中也并不做这个假设。实际上，如果要发现直观上无法体现出来的捷径，最短路径算法就是最好的算法。比如说，通过几个其他顶点的两个顶点之间的路径，其总权值小于连接这两个顶点的直接的边上的权值。第三，从一个顶点到另一个顶点可能存在多条路径有相同的权值；一般只要找出其中的一条就满足了。图 21-2 显示了带有一般权值的一个例子，此图就描述了这些情况。

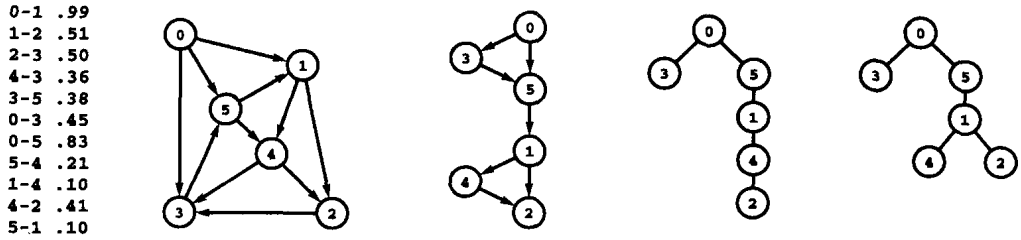


图 21-2　最短路径树

最短路径树（SPT）定义了从根到其他顶点的最短路径（见定义 21.2）。一般地，不同路径可能有相同的长度，因此，可能有多个 SPT 定义了从某个给定的顶点开始的最短路径。在左边所示的示例网中，从 0 开始的所有最短路径都是此网右边所示的 DAG 的子图。以 0 为根的树生成了此 DAG，当且仅当它是 0 开始的一个 SPT。右边的两棵树就是这样的树。

定义中限制路径为对简单路径，对于包含边上具有非负权值的网是不必要的，因为在这样的网中，路径中的任何环都会被删除，从而给出一条不会更长的路径（而且一般会更短，除非此环由 0 权值的边组成）。但是在考虑可能有负权值的边的网时，需要限制为简单路径的要求就变得明显了；否则，如果一个网中存在有负权值的环，那么最短路径的概念就没有意义了。例如，假设图 21-1 中的网中，边 3-5 的权值为 -0.38 和 5-1 的权值为 -0.31。那么环 1-4-3-5-1 上的权值为 0.32 +0.36 -0.38 -0.31 = -0.01，我们可以沿着环旋转来产生任意短的路径。如果仔细可以注意到，负权值环中的所有边不必都是负权值；关注的是边上权值之和。为简明起见，使用术语负环（negative cycle）表示其总权值为负数的有向环。

在此定义中，假设从 s 到 t 的路径上的某个顶点也在一个负环上。在这种情况下，从 s 到 t 存在一条（非简单）最短路径会是一个矛盾，因为我们可以使用此环来构造一个权值比任何给定值更小的路径。为了避免出现此矛盾，在定义中限制到简单路径，因而最短路径的概念在任何网中都是良定义的。然而，到 21.7 节才会考虑负环，因为，我们会看到，它们设置了求解最短路径问题的真正的基本障碍。

为了找出一个加权无向图中的最短路径，我们构建一个有相同顶点、且原图中每条边都有两条边的网。网中的简单路径与图中的简单路径存在一一对应关系，而且路径的成本是相同的；因此，最短路径问题是等价的。实际上，当构建一个加权无向图的标准邻接表或邻接矩阵表示时，就构建了这个网（例如，见图 20-3 所示）。如果权值可能是负值，这种构造没有么用处，因为它会导致网中有负环，而且我们并不知道如何求解含有负环的网中的最短路径问题（见 21.7 节）。否则，在本章所考虑的网算法同样适用于加权无向图。

在某些应用中，除了在边上带有权值，顶点上带有权值也很方便；我们也可能考虑更复杂的问题，其中路径上的边数及路径上的总权值起着一个作用。可以根据边权值网重构来解决这些问题，也可以稍微扩展基本算法来解决（例如，见练习21.52）。

通常从上下文中就能区分，我们并不引入特殊术语对加权图中的最短路径与不带权值的图的最短路径进行区分（其中路径权值就是其上的边数）（见17.7节）。通常的情况指的是（边加权的）网，类似本章中所使用的。因为无向图或非加权图所引出的特例都可由处理网的算法来解决（例如，见练习21.9）。

```
   0        .41  0-1      .82  0-5-4-2  .86  0-5-4-3  .50  0-5-4    .29  0-5
1.13  1-4-3-0       1      .51  1-2      .68  1-4-3    .32  1-4     1.06  1-4-3-5
 .95  2-3-0    1.17  2-3-0-1       2     .50  2-3     1.09  2-3-5-4  .88  2-3-5
 .45  3-0       .67  3-5-1    .91  3-5-4-2   0  3       .59  3-5-4    .38  3-5
 .81  4-3-0    1.03  4-3-5-1  .32  4-2      .36  4-3      0  4       .74  4-3-5
1.02  5-4-3-0   .29  5-1      .53  5-4-2    .57  5-4-3   .21  5-4       0  5
```

图21-3 所有对最短路径

此表给出了图21-1中网的所有最短路径及其长度。此网是强连通的，因此存在将每对顶点连接起来的路径。

源点－汇点最短路径算法的目标是计算此表中的一个元素；单源点最短路径算法的目标是计算出此表中的一行；所有对最短路径算法的目标是计算出整个表。一般地，我们采用更为简洁的表示，它们包含着同样的基本信息，并且允许客户程序在与边数成正比的时间内跟踪任何路径（见图21-8）。

我们对在18.7节中为无向图和非加权图所定义的同一些基本问题感兴趣。这里重新阐述这些问题，注意到定义21.1隐含着推广了这些问题，从而网中考虑了权值。

源点－汇点最短路径 给定一个起始顶点 s 以及一个完成顶点 t，找出图中从 s 到 t 的一条最短路径。我们称起始顶点为源点（source），完成顶点为汇点（sink），但在某些上下文中，这种用法会与有向图中源点（没有进入边的顶点）和汇点（没有离开边的顶点）的定义发生矛盾。

单源点最短路径 给定一个起始顶点 s，找出此图中从 s 到图中其他顶点的最短路径。

所有点对最短路径 找出连接图中每对顶点之间的最短路径。为简化起见，我们有时使用术语所有最短路径（all shortest path）来指这 V^2 条路径的集合。

如果连接任何顶点的给定对之间存在多条路径，只要找出一条就满足了。因为路径上的边的数目可以不同，我们的实现提供ADT函数，允许客户程序在与路径长度成正比的时间内跟踪这些路径。任何最短路径也隐含着给出了这条最短路径的长度，但是实现上显式地提供了长度。总而言之，为了准确，在上述给出的问题中当我们称“找一条最短路径”时，含义是“计算最短路径长度并在与此路径长度成正比的时间内跟踪某条路径。”

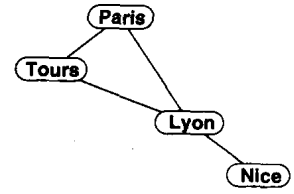
图21-3展示了图21-1中的示例网中的最短路径。在 V 个顶点的网中，需要确定 V 条路径来求解这个单源点问题，并且需要确定 V^2 条路径来求解所有对问题。在我们的实现中，使用一种比路径表更简洁的表示；首先在18.7中提到过这种表示，然后在21.1节详细考虑。

在现代实现中，我们将这些问题的算法学解构建为ADT实现，允许我们构建高效的客户程序，解决大量实际图处理问题。例如，类似在21.3节中看到的，封装所有对最短路径问题的解的一种吸引人的方法是将它作为ADT接口中的预处理函数，提供一个常量时间的最短路径查询实现。我们也可以提供一个求解单源点问题的预处理函数，因此计算从一个特定顶点（或顶点的一个小集合）的最短路径的客户程序可以避免计算其他顶点的最短路径的成本。仔细地考虑这个问题并使用我们所考虑的合适算法就可以体会到实际问题的高效解和客户程序无法使用的昂贵解之间的差异。

最短路径问题以各种形式出现在各种应用中。很多应用可以直接反映几何直观性，但很多其他应用会涉及任意的成本结构。类似在第 20 章对最小生成树（MST）的做法，我们有时利用了几何直观性来帮助开发对求解问题的算法的理解，但仍然保持着算法在更为一般的情况下的正确操作。在第 21.5 节中，我们会考虑针对欧几里得网的专门算法。更重要的是，在第 21.6 和 21.7 节，我们会看到基本算法对于大量使用网来表示计算模型的应用问题是有效的。

公路图（Road map）　给出所有主要城市对之间距离的表是很多公路图的一个显著特征。我们假设制图者已经下功夫确定了最短距离，但这个假设并不一定成立（例如，见练习 21.10）。一般地，这样的表是用于无向图的，我们在网中应该将边处理为两个方向，对应着每条道路。我们可能不太重视处理城市地图中的单向街道以及一些类似的情况。在 21.3 中将看到，要提供其他有用信息并不困难，比如说一个表会告诉我们如何执行最短路径（见图 21-4）。在现代应用中，嵌入式系统在汽车及运输系统中提供了这种能力。地图是欧几里得图；在第 21.4 节中，在查找最短路径时，我们会考察考虑顶点位置的最短路径算法。

航班路线（airline route）　航班或者其他运输系统的路线图和调度表可表示为网，其中各种最短路径问题都有着直接的重要性。例如，在两个城市间飞行时，可能希望乘飞机时间最小或者旅途成本最小。在这样的网中的成本会是时间、金钱和其他复杂资源的函数。例如，两个城市之间的航班由于风的原因，一般在一个方向所花费的时间比在另一个方向要多。乘客也知道费用未必是城市之间距离的一个简单函数，在这种情况下，使用迂回航线（或忍受短暂停留）要比直航更便宜是很普遍的。使用本章所考虑的基本最短路径算法就可处理这样的复杂情况；所设计的算法可用于处理任何正值成本。



	Lyon	Nice	Paris	Tours
Lyon		200	400	420
Nice	200		600	620
Paris	400	600		120
Tours	420	620	120	

	Lyon	Nice	Paris	Tours
Lyon		Nice	Paris	Tours
Nice	Lyon		Lyon	Lyon
Paris	Lyon	Lyon		Tours
Tours	Lyon	Lyon	Paris	

图 21-4　距离和路径

对于这样一个由高速公路所连接的法国城市的小子集（上图所示），其公路图一般地包含类似中图的距离表。尽管很少在地图中找到，像下图这样的表也是有用的，因为它告诉我们应该根据哪些标志来选择最短路径。例如，要确定如何从 Paris 前往 Nice，可以检查此表，它告诉我们由指向 Lyon 的标志开始。

这些应用所提到的基本最短路径计算只是触及到最短路径算法应用的表面。在 21.6 中，我们考虑哪些似乎与这些自然问题不相关的应用领域中的问题，在上下文中讨论*归约*（reduction）的概念，为问题之间的关系提供一种形式化机制。我们将这些应用问题转化为抽象最短路径问题来对它们求解，这些问题没有上述描述的几何上的直观感受。实际上，某些应用问题使我们考虑带有负权值的网中的最短路径问题。这些问题要比没有负权值出现的那些问题要难解的多。对于这样应用的最短路径问题，不仅建立了基本算法与未解算法学难题之间的桥梁，而且还导致强大和一般求解问题的机制。

类似我们在第 20 章中的 MST 算法所做的，我们常常混合使用权值、成本和距离这些概念。即当处理的是有任意边权值的一般情况时，仍会利用几何直观性的自然优势；因此当谈到“权值”，是指路径和边“长度”，当谈到“权值更小”是指一条路径比另一条路径更短。当我们谈到从 s 到 v 的“最小权值有向路径”比从 s 到 w 的“最小权值有向路径”更小时，是指 v 比 w“更接近”s，如此等等。对于标准使用术语“最短路径”，其用法是固有的，而且即使权值与路径无关，也是自然的（见图 21-2）；然而，在 21.6 节对算法进行扩展以处理负权值时，还必须放弃这种用法。

本章组织如下。在 21.1 节介绍基本原理后，21.2 和 21.3 介绍单源点最短路径问题和所

有对最短路径问题。然后，在21.4节考虑无环网（或用缩写形式，加权DAG），并在21.5节中研究针对欧几里得图源点－汇点问题利用几何性质的几种方法。然后，21.6和21.7节从另一角度观察更一般的问题，将最短路径问题作为高级求解问题的工具，其中可能会涉及带有负权值的网。

练习

▷ 21.1 在平面上对以下点分别进行标号（0～5）：

```
(1, 3)  (2, 1)  (6, 5)  (3, 4)  (3,7 )  (5, 3)
```

取边长度为权值，考虑由以下边所定义的网：

```
1-0  3-5  5-2  3-4  5-1  0-3  0-4  4-2  2-3
```

画出该网，并给出由程序17.8所构建的邻接表结构，对其进行修改使之适合于处理网。

21.2 按照图21-3的风格，显示练习21.1中所定义的网的所有最短路径。

○ 21.3 对于顶点及边上均带有非负权值的网，表明其最短路径的计算（其中路径的权值定义为此路径上顶点和边的权值之和）可以通过构建只在边上带权的网ADT来解决。

21.4 找出一个大型在线网，可以是一个地理数据库，其元素为连接城市的公路，或是包含距离或成本的航班或铁路时刻表。

21.5 基于程序17.6，编写一个稀疏网的网ADT，网中的权值介于0～1之间。其中包含基于程序17.7的一个随机网生成器。使用分离的ADT来产生边的权值，并编写两个实现：一个用于产生均匀分布的权值，另一个用于产生服从高斯分布的权值。对于仔细选择的V值和E值，编写针对这两种权值分布产生随机网的客户程序，从而可以使用它们对于有各种边权值分布的图进行实验测试。

○ 21.6 基于程序17.3，编写一个稠密网的网ADT，网中的权值介于0～1之间。其中包含基于程序17.8的一个随机网生成器，且边权值的生成如练习21.5中的描述。对于仔细选择的V值和E值，编写针对两种权值分布产生随机网的客户程序，从而可以使用它们对于有各种边权值分布的图进行实验测试。

21.7 实现一种与表示无关的网ADT，要求用带有权值的边（0～$V-1$之间的整数对，且权值介于0～1之间）来构建一个网，这些边的权值由标准输入而得。

● 21.8 编写一个产生平面上V个随机点的程序，将此平面上距离d内的每对顶点连接起来，然后用这些边（在两个方向）构建一个网（见练习17.72），将每条边上的权值设置为该边所连接的两个点之间的距离。确定如何设置d，使得边的期望数为E。

○ 21.9 编写使用网ADT来实现无向图和非加权有向图的ADT函数。

▷ 21.10 下表得自于一个已出版的公路图，声称给出了连接城市最短路径的长度。它包含有一个错误。改正此表。另外，按照图21-4的风格，再增加一个显示最短路线的表。

	Providence	Westerly	New London	Norwich
Providence	—	53	54	48
Westerly	53	—	18	101
New London	54	18	—	12
Norwich	48	101	12	—

21.1 基本原理

我们的最短路径算法基于一个称为松弛（relaxation）的简单操作。开始一个最短路径算法时，只知道网的边及权值。随着算法的进行，收集连接每对顶点之间最短路径的信息。算法递增地更新所有最短路径的信息，并根据目前所得知识，做出最短路径的推理。在每一步中，都会检查是否可以找到一条比某条已知路径更短的路径。术语“松弛”常用于描述这一步，对最短路径放松约束。可以将连接两个顶点的一条路径看作是绷紧的橡皮筋：成功的松弛操作允许我们在一条更短的路径上对橡皮筋的绷紧程度进行放松。

我们的算法基于反复地应用以下两种类型的松弛操作：

- 边松弛：检查沿着给定边行进是否会产生到目的顶点的一个新的最短路径。
- 路径松弛：检查沿着给定边行进是否会产生连接两个其他给定顶点的一条新的最短路径。

边松弛是路径松弛的一个特例；然而，由于要分别使用这两个操作（前者在单源点算法中使用，后者在所有对最短路径算法中使用），因而我们分别考虑。在这两种情况下，在所用数据结构上施加的主要要求是容易更新以反映松弛操作中蕴含的变化，这个数据结构用于表示网的最短路径的当前状态。

首先，我们考虑边松弛操作，如图 21-5 所示。我们考虑的所有单源点最短路径算法都是基于这一步：对于给定的一条边，它能带来一条从源点到其目的顶点的更短路径吗？

支持此操作所需的数据结构很简单。首先，基本要求是要计算出从源点到每个其他顶点的最短路径长度。常规是将从源点到每个顶点的已知路径的最短路径长度存储在一个顶点索引的数组 wt 中。其次，当从顶点到顶点移动时，要记录路径自身。其中所用数据结构与在第 18 章～第 20 章中的其他图搜索算法所使用的一样：使用一个顶点索引的数组 st 来记录从源点到每个顶点的最短路径上的前一个顶点。这个数组就是树的父链接表示。

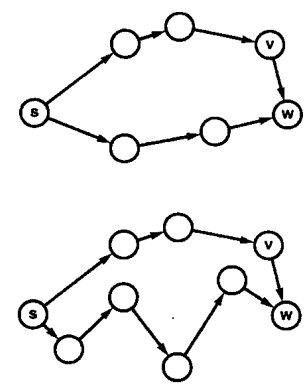


图 21-5 边松弛

这些图展示了单源点最短路径算法的基本松弛操作。将从 s 到每个顶点的已知最短路径保存起来，并检查边 v-w 是否给出到 w 的更短路径。在上图的例子中，它没有给出；因而忽略它。在下图的例子中，可以给出到 w 的更短路径，因此我们会更新数据结构以便指示出从 s 到 w 的已知最好方式是先到 v，再取 v-w。

有了这些数据结构，实现边松弛就是一件很简单的任务。在单源点最短路径代码中，使用以下松弛代码来松弛从 v 到 w 的一条边：

```
if (wt[w] > wt[v] + e.wt)
 { wt[w] = wt[v] + e.wt; st[w] = v; }
```

此段代码既简单又具描述性。我们就以这种形式包含在实现中，而不是将松弛定义为高级抽象操作。

定义 21.2 给定一个网及一个指定的顶点 s，s 的最短路径树（SPT）是包含 s 和由 s 可达的所有顶点的一个子网，该子网形成以 s 为根的一个有向树，满足每条树路径是网中的一条最短路径。

连接给定结点对且有相同长度的路径可能有多，因此，SPT 不必是唯一的。一般地，如图 21-2 所示，如果我们取从顶点 s 到网中由 s 可达的每个顶点的最短路径，以及这些路径上的边所导出的子网，则可得一个 DAG。连接每对结点的不同最短路径都可能作为子路径出现在连接这些顶点的某个更长的路径中。由于这些影响，对于给定的有向图及起始顶点，只要计算出任何一个 SPT 就满足了。

我们的算法一般会将数组 wt 中的元素初始化为观察哨值 maxWT。这个值需要足够小，使得在松弛检查中增加此值不会引起上溢，而且还要足够大，使得不存在有更大权值的简单路径。例如，如果边权值在 0 和 1 之间，我们可以使用值 V（作为 maxWT）。注意到对于可能带有负权值的网，在用观察哨时，必须特别仔细来检查对观察哨所做的假设。例如，如果两个顶点都有观察哨值，倘若 e.wt 非负，则上述给出的松弛代码不作任何改变（这是大多数的实现中我们所期望的），但如果权值为负，松弛将改变 wt[w] 和 st[w]。

st 数组是最短路径树的一种父链接表示，其中的链接指向与在网中的链接指向相反，如图 21-6 所示。我们可以从 t 到 s 向树的上方行进，计算出从 s 到 t 的最短路径，以相反的顺序访问此路径上的顶点（t、st[t]、st[st[t]] 如此等等）。在某些情况下，相反的顺序正是我们想要的顺序。例如，如果返回的是此路径的链表表示，可以（按照以往对于链表的约定，其中 NEW 是为结点分配内存的函数，用其参数填充结点的域，并返回指向该结点的一个链接）使用类似下面的代码：

```
p = NEW(t, null);
while (t != s)
  { t = st[t]; p = NEW(t, p); }
return p;
```

另一种选择是使用类似于将路径上的顶点压入栈中的代码，然后客户程序可以按照从栈中弹出的顺序访问路径上的顶点。

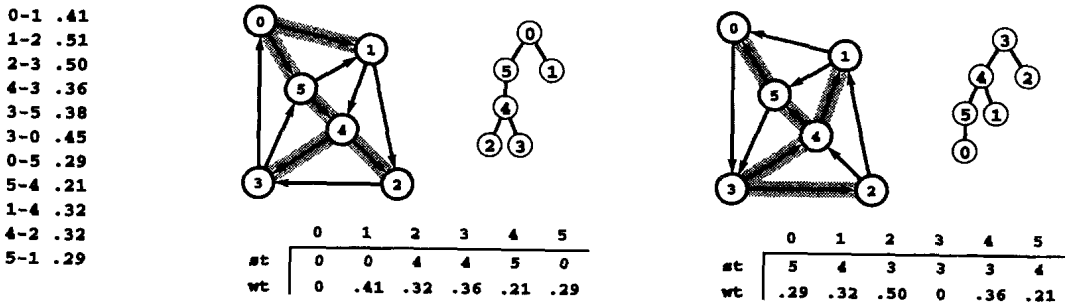


图 21-6 最短路径树

从 0 到此网中其他结点的最短路径分别是 0-1、0-5-4-2、0-5-4-3、0-5-4 和 0-5。这些路径定义一个生成树，用中间的三种表示显示（网中绘出的灰边，有向树，以及带有权值的父链接）。父链接表示（通常计算的表示）中的链接与有向图中的链接恰好相反，因此有时会处理其逆有向图。从 3 到逆图中其他结点的最短路径所定义的生成树在右图中描绘出。此树的父链接表示给出了原图中从每个其他顶点到 3 的最短路径。例如，通过沿着链接 st[0]=5，st[5]=4，st[4]=3，就可以找出从 0 到 3 的最短路径为 0-5-4-3。

另一方面，如果我们想要打印或者处理路径中的顶点，逆序则不方便，因为我们必须按照逆序在这条路径上行进才能达到第一个顶点，然后，沿着这条路径返回以处理顶点。一种绕过这种困难的方法是处理逆图，如图 21-6 所示。

接下来，我们考虑路径松弛，它是某些所有对最短路径算法的基础。通过一个给定的顶点可得连接其他两个顶点的更短的路径吗？例如，假定有三个顶点 s，x 和 t，我们希望知道是否从 s 到 x，然后再从 x 到 t 会比从 s 到 t 而不经过 x 更好呢？对于欧几里得空间中的直线连线，由三角不等式可知，通过 x 的路径不会比从 s 到 t 的直接路径更短，但对于网中的路径，确是可能的（见图 21-7）。为了确定是哪一条更短，需要知道从 s 到 x，x 到 t 的长度，以及从 s 到 t 的路径（此路径中不含 x）长度。然后，简单检查前两个路径长度的和是否小于第三条路径的长度；如果小于，则更新相应的记录。

路径松弛适合于所有对最短路径的问题，其中维护着遇到的所有对最短路径长度。具体地说，在所有对最短路径的这类代码中，维护一个数组 d，满足 d[s][t] 是从 s 到 t 的最短路径长度，同时还维护一个数组 p，满足 p[s][t] 是从 s 到 t 的最短路径上的下一个顶点。我们称前者为距离（distance）矩阵，后者为路径（path）矩阵。图 21-8 显示了示例网的这两个矩阵。距离矩阵是此计算的一个主要目标，我们使用路径矩阵是因为它比图 21-3 中所描述的完全路径表更简洁，但携带相同的信息。

根据这些数据结构，路径松弛的代码如下：

```
if (d[s][t] > d[s][x] + d[x][t])
  { d[s][t] = d[s][x] + d[x][t]; p[s][t] = p[s][x]; }
```

类似于边松弛，此段代码可以认为是对已经给出的非形式化描述的重新表述，因而可以在实现中直接使用。更正式地，路径松弛反映了如下内容：

性质 21.1 *如果顶点 x 在从 s 到 t 的最短路径上，那么那条最短路径包含从 s 到 x 的最短路径，后面跟着一条从 x 到 t 的最短路径。*

证明 用反证法。可以使用从 s 到 x 或从 x 到 t 的更短路径来构建从 s 到 t 的最短路径。 ■

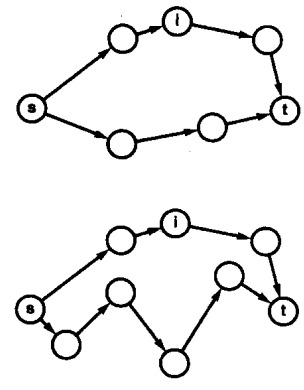


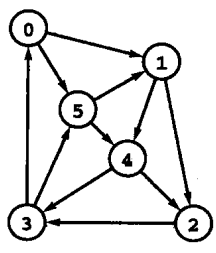
图 21-7 路径松弛

这些图展示了所有对最短路径算法的基本松弛操作。将所有对已知最短路径保存起来，并检查顶点 i 是否能成为改进从 s 到 t 的已知最短路径的证据。在上图的例子中，它不是；在下图的例子中，它是。每当遇到一个顶点 i，使得从 s 到 i 的已知最短路径长度加上 i 到 t 的已知最短路径长度要比 s 到 t 的已知最短路径长度更小时，则更新数据结构以便指示出从 s 到 t 的更短路径（先走到 i）。

在 19.3 节讨论传递闭包时，就遇到过路径松弛操作。如果此边和路径权值要么为 1，要么为无穷大（也就是说，路径的权值为 1，仅当那条路径中的所有边权值为 1），那么路径松弛就是在 Warshall 算法中所使用的操作（如果从 s 到 x 有一条路径，从 x 到 t 有一条路径，那么从 s 到 t 有一条路径）。如果我们将路径的权值定义为那条路径中的边数，那么 Warshall 算法就推广为找出非加权有向图的所有对最短路径的 Floyd 算法；进一步推广可应用到网中，如在 21.3 节中所看到的。

从数学家的角度来看，务必请注意这些算法都能转换到统一的代数环境中，并帮助我们理解它们。从程序员的角度来看，务必注意使用一个抽象 + 操作符（由边权值来计算路径权值）和一个抽象 < 操作符（计算路径权值集中的最小值）就可以实现这些算法，这两种方式是在松弛操作的上下文中使用的唯一操作（见练习 19.53 和练习 19.54）。

```
0-1 .41
1-2 .51
2-3 .50
4-3 .36
3-5 .38
3-0 .45
0-5 .29
5-4 .21
1-4 .32
4-2 .32
5-1 .29
```



	0	1	2	3	4	5
0	0	.41	.82	.86	.50	.29
1	1.13	0	.51	.68	.32	1.06
2	.95	1.17	0	.50	1.09	.88
3	.45	.67	.91	0	.59	.38
4	.81	1.03	.32	.36	0	.74
5	1.02	.29	.53	.57	.21	0

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

图 21-8 所有对最短路径

右边的两个矩阵是左边示例网的所有顶点最短路径的简洁表示，包含着与图 21-3 中完全表中的相同信息。左边的距离矩阵包含着最短路径长度：s 行和 t 列中的元素为从 s 到 t 的最短路径长度。右端的路径矩阵包含着执行路径所需的信息：s 行和 t 列中的元素为从 s 到 t 的路径上的下一个顶点。

性质21.1 蕴含着从 s 到 t 的最短路径包含着从 s 到通向 t 的路径上的其他顶点的最短路径。大多数最短路径算法还计算从 s 到距离 s 比 t 更近的每个顶点的最短路径（而不论该顶点是否在从 s 到 t 的最短路径上），尽管没有要求计算出该顶点的最短路径（见练习21.16）。当 t 是距离 s 最远的顶点时，用此算法求解源点－汇点最短路径问题就等价于求解从 s 开始的单源点的最短路径问题。相反，可以使用求解从 s 开始的单源点最短路径问题的方法来找出距离 s 最远的顶点。

在所有对问题的实现中所使用的路径数组表示每个顶点的最短路径树。将 p[s][t] 定义为从 s 到 t 的最短路径中在 s 之后的顶点。也是逆网中从 t 到 s 的最短路径中在 s 之前的同一顶点。换句话说，在一个网的路径矩阵中，列 t 表示其逆网中对应顶点 t 的 SPT 的一个顶点索引的数组。反之，对于一个网，可以使用其逆网中相应顶点的 SPT 的顶点索引数组表示填充每列，来构建一个路径矩阵。这种对应关系如图21-9所示。

我们在21.4节详细考虑 ADT 设计，那时就会看到求解所有对问题的具体解决方案。在21.2节，考虑单源点最短路径问题，并使用边松弛来计算任何给定源点的 SPT 的父链接表示。

练习

▷ **21.11** 绘出练习21.1中所定义的网及其逆网的从0开始的 SPT。给出这两棵树的父链接表示。

21.12 将练习21.1中所定义的网中的边考虑为无向边，使得每条边在此网中的两个方向对应着权值相等的边。对于这个网回答练习21.11中的问题。

▷ **21.13** 改变图21-2中边0-2的方向。绘出此修改的网中以2为根的两个不同的 SPT。

▷ **21.14** 使用 SPT 的父链接表示，编写一段打印出到根的每条路径的代码。

▷ **21.15** 使用网中所有对路径的路径矩阵表示，编写一段打印出所有路径的代码，风格同图21-3。

21.16 给出一个例子，对于某个 x，在不知道从 s 到 x 的一条更短的路径长度时，说明如何知道从 s 到 t 的一条路径是最短的。

21.2 Dijkstra 算法

在20.3节中，我们讨论了找出一个加权无向图的最小生成树（MST）的 Prim 算法：每次选择一条边构建 MST 时，总是选择连接 MST 的一个顶点与不在 MST 中的一个顶点的最短的边。我们可以使用一种近乎相同的方法来计算 SPT。从将源点放在 SPT 中开始，然后每次选择一条边构建 SPT，总是选择从源点到不在 SPT 中的一个顶点的最短路径的边。换句话说，向 SPT 中增加顶点是按照它与起始顶点的距离（通过 SPT）的顺序来进行的。这种方法被称为是 Dijkstra 算法（Dijkstra's algorithm）。

如常，需要在抽象层次的算法（采用这种非形式化描述）和各种具体实现（主要是图的表示和优先队列实现的差异，如程序21.1）之间做出区分，即使在文献中并不总是加以区分。在确定了 Dijkstra 算法能够正确地执行单源点最短路径计算后，还将考虑其他实现并讨论这些实现与程序21.1之间的关系。

性质21.2 *Dijkstra 算法能够解决带有非负权值的网的单源点最短路径问题。*

证明 给定一个源顶点 s，必须确定 Dijkstra 算法所计算出的从根 s 到树中每个顶点 x 的树路径对应图中从 s 到 x 的一条最短路径。这一点由归纳法可得。假设目前所计算出的子树具有这个性质，只需要证明增加一个顶点 x 就增加了到此顶点的一条最短路径。但是到 x 的所有其他路径必定从树路径开始，后面跟着指向不在树中顶点的一条边。由构造，所有这样的路径都比当前考虑的从 s 到 x 的路径更长。

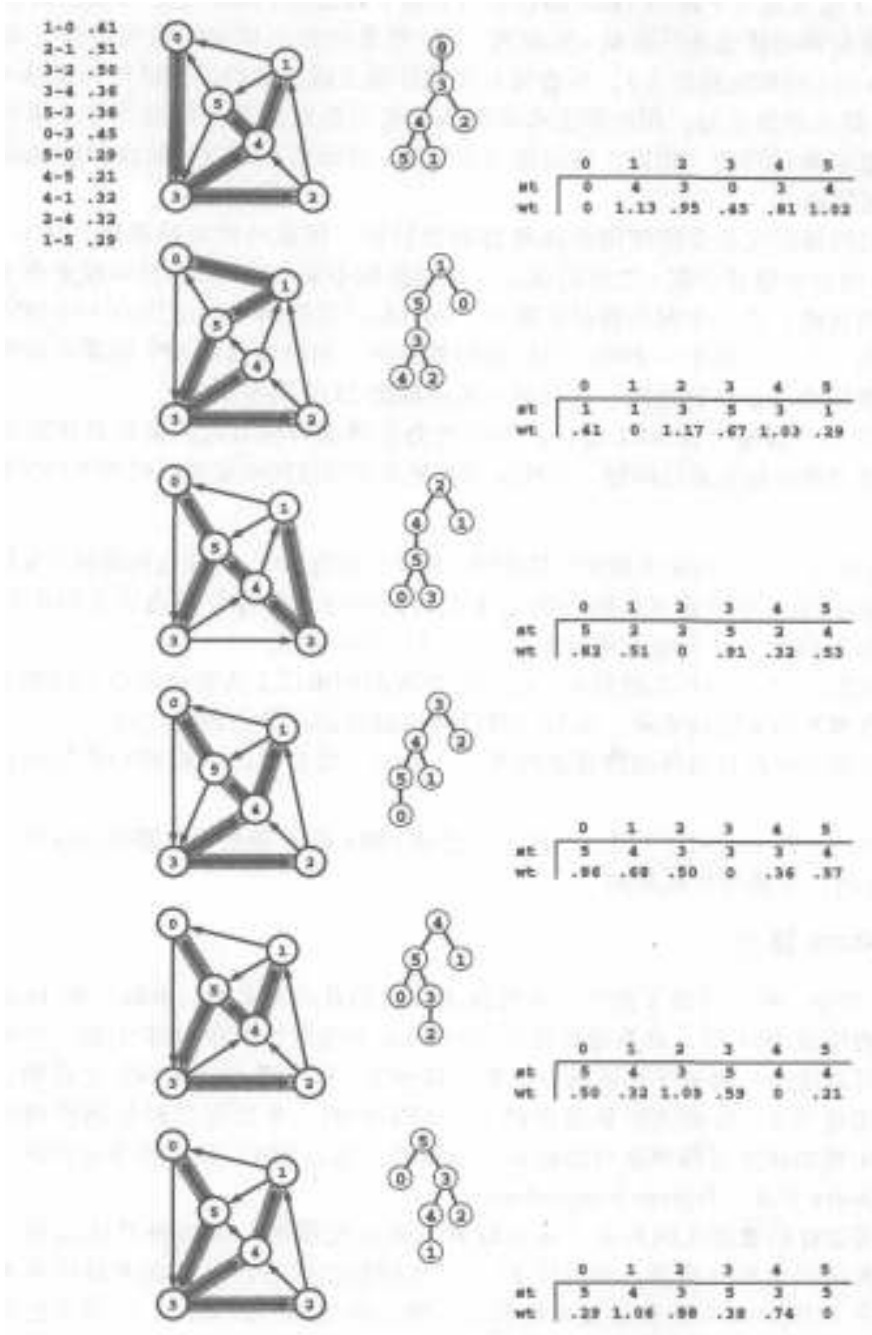


图 21-9 网中所有对最短路径

这些图描述了图 21-8 的逆网中每个顶点的 SPT（从上到下 0～5），表示为网的子树（左图），有向树（中图），以及包含一个对应路径长度的顶点索引数组的父链接表示（右图）。将顶点索引数组放在一起构成路径矩阵和距离矩阵（每个索引数组成为其中的一列），就得到图 21-8 中所描述的所有对最短路径。

同理可证如果 Dijkstra 算法从源点开始，在汇点从优先队列中出队后能终止，则该算法解决了源点－汇点最短路径问题。■

如果边权值可能为负，则结论不成立。因为在此假设当向路径增加更多的边时，路径长度不会减小。在含有负的边权值的网中，此假设无效，因为我们遇到的任何边都可能导致某个树顶点，而且可能有足够大的负权值，从而给出一条到此顶点的比树路径更短的路径。在21.7节考虑这个问题（见图21-28）。

图21-10显示了一个示例图使用 Dijkstra 算法计算时的 SPT 的演化过程；图21-11显示了一个大型 SPT 树的带有方向的绘制图。尽管 Dijkstra 算法与 Prim 算法仅在优先级选择上有所不同，但 SPT 树在特征上还是不同于 MST。它们的根都是起始顶点，所有边都是在离开根的方向，而 MST 是无根和无向的。在使用 Prim 算法时，将 MST 表示为有向、有根树，但这些结构仍然在特征上不同于 SPT（将图20-9的有方向的绘制与图21-1中的绘制进行比较）。实际上，SPT 的本质还有点依赖于起始顶点的选择，如图21-12所示。

Dijkstra 的原始实现适合于稠密图，非常类似于 Prim MST 算法。具体地说，我们可以将程序20.3中的 P 的定义由

```
#define P G->adj[v][w](边权值)
```

变为

```
#define P wt[v] + G->adj[v][w](从源点到边的目的顶点的距离)
```

这个改变给出了 Dijkstra 算法的经典实现：一次增加 SPT 的一条边，每次都检查所有非树顶点，从而找出一条边移到此树中，使该边的目的顶点是距源点有最小距离的一个非树顶点。

性质21.3 使用 Dijkstra 算法，可以在线性时间内找出一个稠密网的任何 SPT。

证明 类似于 Prim MST 算法，显然由程序20.3中的代码直接可得。运行时间与 V^2 成正比，对于稠密图是线性的。■

对于稀疏图，可以做的更好。将 Dijkstra 算法看作一种推广的图搜索方法，与深度优先搜索（DFS）、广度优先搜索（BFS），以及 Prim MST 算法的不同之处仅在于所使用的向树中增加边的规则不同。如同第20章中的做法，可以将连接树顶点与非树顶点的边保存在一个称为*边缘集*（fringe）的广义队列中，使用优先队列来实现广义队列，并提供更新优先级操作，从而在单个实现中包含 DFS、BFS 和 Prim 算法（见20.3节）。这种优先级优先搜索（PFS）的模式也包含了 Dijkstra 算法。也就是说，将程序20.4中 P 的定义（从源点到边的目的顶点的距离）改变为

```
#define P wt[v] + t->wt
```

就给出了适合于稀疏图的 Dijkstra 算法的一种实现。

程序21.1是用于稀疏图的另一种 PFS 实现，比程序20.4稍微简单一些，而且与本节开始时给出的 Dijkstra 算法的非形式描述非常一致。它与程序20.4的不同之处在于，它用网中的所有顶点来初始化优先队列，并借助于观察哨值来维护此队列中的顶点，这些顶点（带有观察哨值的不可见顶点）既不在树中也不在边缘集中；对比之下，程序20.4只将由此树中的一条边可达的那些顶点保存在优先队列中。将所有顶点保存在优先队列中简单了代码，但对于某些图可能带来一些性能上的代价（见练习21.31）。

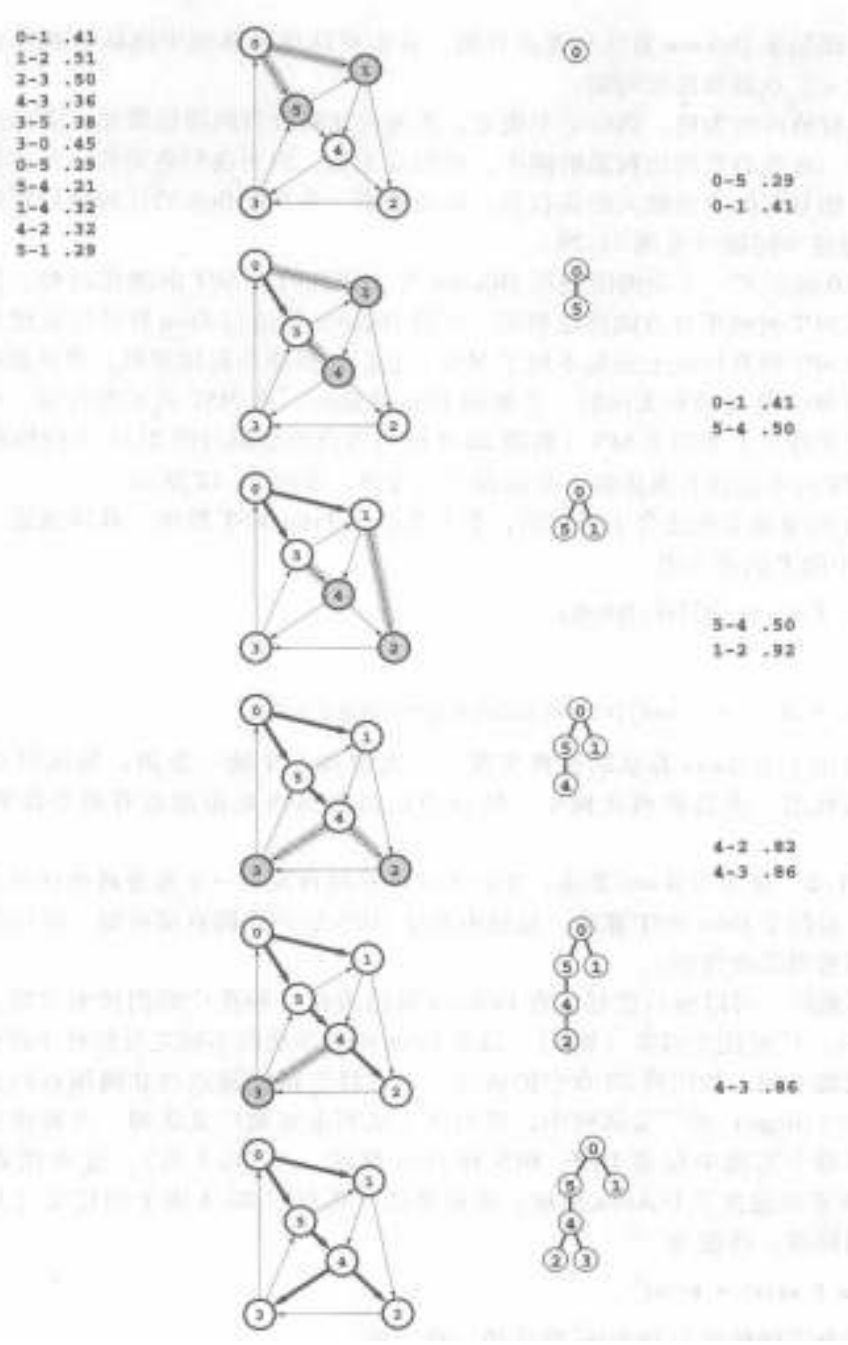


图 21-10 Dijkstra 算法

对于一个示例网，此序列描述了使用 Dijkstra 算法构造根为 0 的最短路径生成树的过程。网图中加粗的黑边是树边，加粗的灰边是边缘边。中图显示出了随着这棵树增大相应的有向绘制图。右边则给出了边缘集列表。

第一，将 0 增加到树中，并将离开它的边（0-1 和 0-5）增加到其边缘集中（上图）。第二，将这些边中的最短边 0-5 从边缘集中移到此树中，并检查离开它的边：将边 5-4 增加到边缘集中，且边 5-1 被丢弃，因为对于已知的路径 0-1，它不是从 0 到 1 的一条更短路径的一部分（自上第二个图）。边缘集中的 5-4 的优先级是从 0 起始的路径长度，表示为 0-5-4。第三，将 0-1 从边缘集中移到树中，将边 1-2 增加到边缘集中，并丢弃 1-4（自上第三个图）。第四，将 5-4 从边缘集中移到树中，将边 4-3 增加到边缘集中，并用 4-2 替代 1-2，因为 0-5-4-2 是比 0-1-2 更短的路径（自上第四个图）。至多保留一条指向边缘集中任何顶点的边，且选择那条距 0 有最短路径的边。通过将 4-2，然后是 4-3 从边缘集移到此树中（下图）完成此计算。

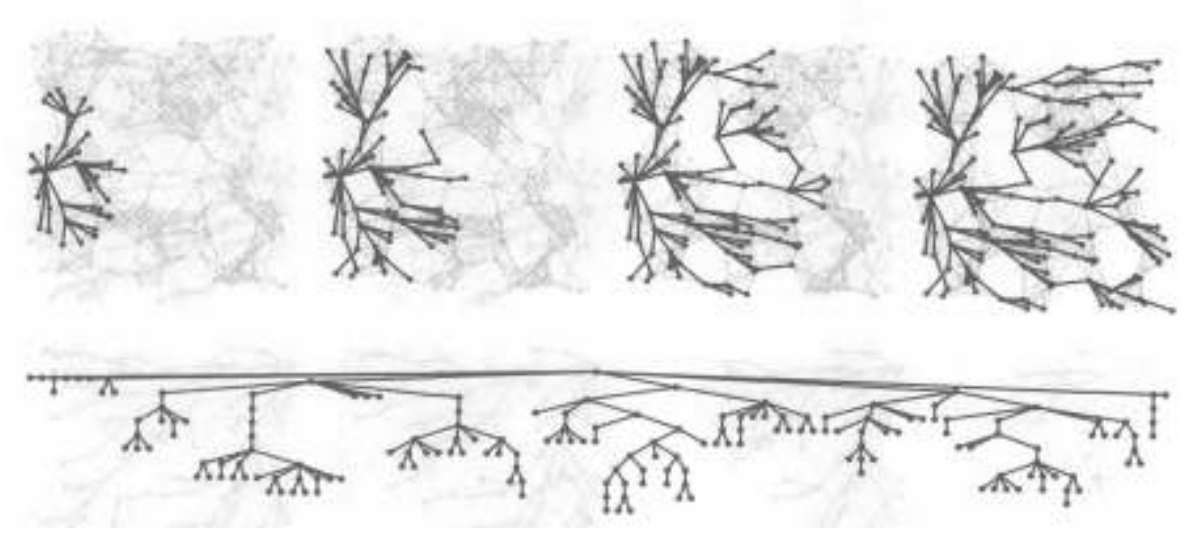


图21-11 最短路径生成树

按照图18-13、图18-24和图20-9的风格，对于一个随机欧几里得近邻图（在两个方向都有有向边，对应画出的每一条线），此序列描述了使用Dijkstra算法求解单源点最短路径的问题的过程。搜索树在特征上类似于BFS，因为顶点会按照短路径连向另一个顶点，但此树会稍微深一些，而且宽度较小。这是因为距离会导致比路径长度稍微长一些的路径。

在第20章中所考虑的关于优先级优先搜索（PFS）的性能的一般结论给出了对于稀疏图关于Dijkstra算法的这些实现的性能的特定信息（程序21.1和程序20.4，适当修改）。为了说明，我们在当前上下文中重新阐述这些结果。因为证明并不依赖于优先级函数，因而这些结论不经修改就可应用。这是应用到这两个程序上的最坏情况下的结果，但对于某些类型的图，由于维护一个较小的边缘集，程序20.4可能更高效。

性质21.4 对于所有网和所有优先级函数，可以使用大小至多为V的优先队列，在与V次插入（insert）、V次删除最小（delete the minimum）和E次减小关键字（decrease key）操作所需时间成正比的时间内，计算出带有PFS的一个生成树。

证明 这一点由程序20.4或程序21.1中基于优先队列的实现直接可得。此结果代表了一个保守界限，因为优先队列的大小往往比V小得多，尤其是对于程序20.4。 ■

性质21.5 使用Dijkstra算法的PFS实现（其中使用堆来实现优先队列），那么可以在与$E \lg V$成正比的时间内计算出任何SPT。

证明 由性质21.4可直接得此结论。 ■

性质21.6 给定V个顶点、E条边的一个图，令d表示图的密度E/V。如果$d<2$，那么Dijkstra算法的运行时间与$V \lg V$成正比。否则，可以将$\lceil E/V \rceil$-叉堆用于优先队列，将最坏情况下的运行时间改进$\lg(E/V)$的一个因子，达到$O(E \lg_d V)$（如果E至少为$V^{1+\varepsilon}$，那么为线性的）。

证明 结果直接反映了性质20.12和其后讨论的多路堆优先队列实现。 ■

程序21.1 Dijkstra算法（邻接表）

Dijkstra算法的这一实现使用了顶点的优先队列（按照距源点的距离的顺序）来计算一个SPT。使用源点的优先级0和其他顶点的优先级为maxWT来初始化此队列，然后进入一个循环，将优先级最小的顶点从队列中移到SPT中，并松弛该顶点的依附边。

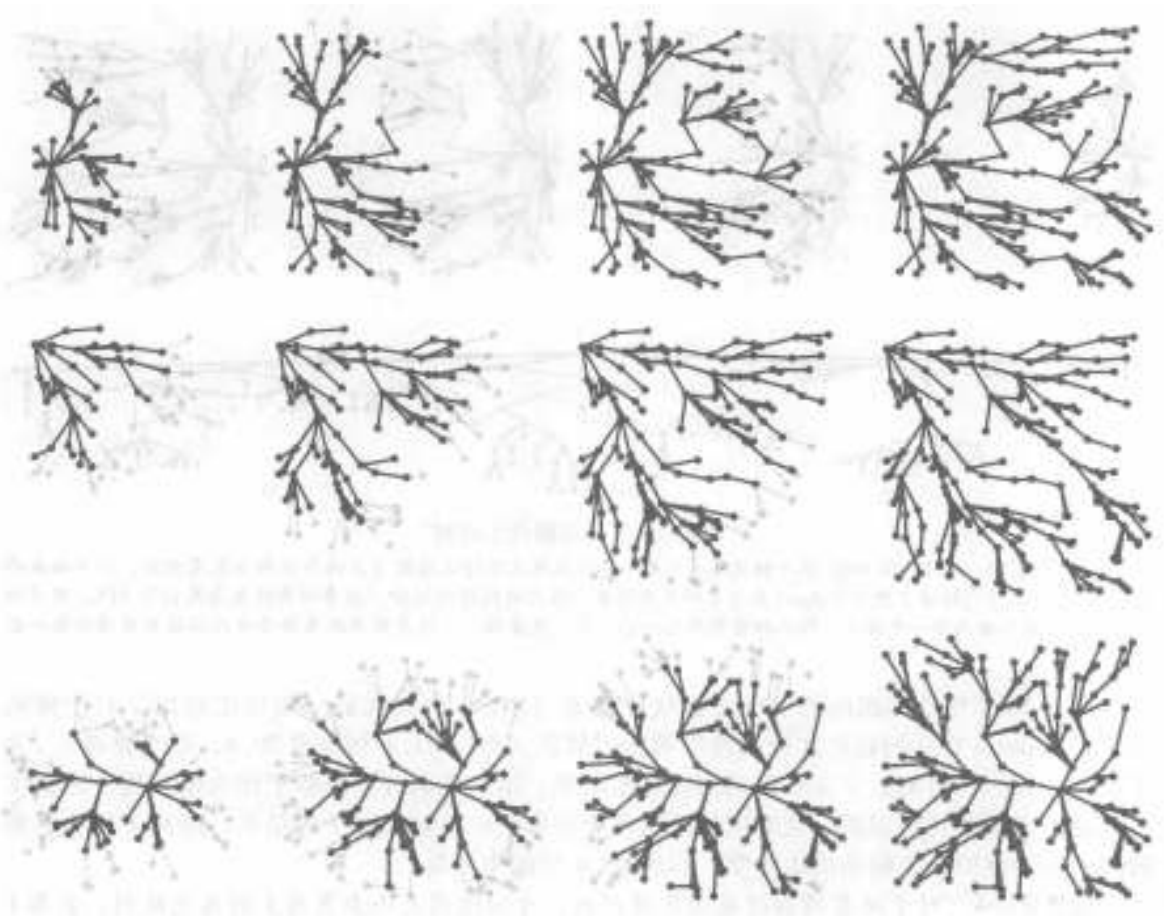


图 21-12 SPT 示例

这 3 个示例显示了对于三个不同的源点位置的 SPT 的增长过程：即源点在左边（上图），左上角（中图），以及中部（下图）位置。

间接的优先队列接口代码与程序 20.4 相同，这里略去。程序中定义了一个静态变量 priority 和一个函数 less，允许优先队列函数操纵顶点名（索引）并使用 less 来比较由此代码中的 wt 数组所维持的优先级。

此段代码是一个广义的图搜索，一种 PFS 实现。P 的定义实现了 Dijkstra 算法；其他定义实现了 PFS 的其他算法（见正文）。

```
#define GRAPHpfs GRAPHspt
#define P (wt[v] + t->wt)
void GRAPHpfs(Graph G, int s, int st[], double wt[])
  { int v, w;  link t;
    PQinit(); priority = wt;
    for (v = 0; v < G->V; v++)
      { st[v] = -1; wt[v] = maxWT; PQinsert(v); }
    wt[s] = 0.0; PQdec(s);
    while (!PQempty())
      if (wt[v = PQdelmin()] != maxWT)
        for (t = G->adj[v]; t != NULL; t = t->next)
          if (P < wt[w = t->v])
```

```
            { wt[w] = P; PQdec(w); st[w] = v; }
    }
```

表21-1总结了我们考虑的4种主要PFS算法的有关信息。它们的差别仅在所使用的优先级函数上有所不同，但这种差别导致了在特征上（所需的）完全不同的生成树。对于表中所引用的这些图（以及很多其他图）中的示例，DFS树又高又瘦，BFS树有矮又胖。SPT树很像BFS树，但既不十分矮也不十分胖，MST既不矮胖也不高瘦。

我们还考虑了PFS的4种不同实现。第一种是经典稠密图实现，包含了Dijkstra算法和Prim MST算法（程序20.3）；其他3种是稀疏图的实现，差异仅在优先队列中的内容有所不同：

- 边缘边集（程序18.10）
- 边缘顶点集（程序20.4）
- 所有顶点（程序21.1）

在这些实现中，第一种实现主要具有教学价值；第二种实现这3种中最优化的；第三种实现可能是最简单的。这个框架已描述了经典图搜索算法的16种不同实现，即当使用不同的优先队列实现时，可能性还会出现更多的情况。网、算法和实现的这种多样性强调了性质21.4～性质21.6中有关性能的一般性结论的实用性，也在表21-2中给出。

正如MST算法，最短路径算法的实际运行时间很可能比建议的那些最坏情况下的界限还要低。主要是因为大多数边不必执行减小关键字（decrease key）的操作。实际上，除了最稀疏的图之外，我们都认为算法的运行时间是线性的。

表21-1 优先级优先搜索算法

这4种经典的图处理算法都可以使用PFS来实现，它是一种推广的基于优先队列的图搜索方法，每次增加一条边来构建图的生成树。搜索的动态细节依赖于图的表示、优先队列实现和PFS实现；但搜索树一般地表征了各种算法的特征。如第4列中所引用的图所示。

算法	优先级	结果	图
DFS	逆前序	递归树	18.13
BFS	前序	SPT（边）	18.24
Prim	边权值	MST	20.8
Dijkstra	路径权值	SPT	21.9

表21-2 Dijkstra算法的实现开销

此表总结了Dijkstra算法的各种实现开销（最坏情况下的运行时间）。使用合适的优先队列实现，除了极端稀疏的网之外，算法的运行时间为线性（对于稠密网与V^2成正比，对于稀疏网与E成正比）。

算 法	最坏情况下的开销	注 释
经典	V^2	对稠密图最优
PFS、完全堆	$E \lg V$	最简单的ADT代码
PFS、边缘集堆	$E \lg V$	保守界限
PFS、d-叉堆	$E \lg_d V$	除了极端稀疏图之外，是线性的

Dijkstra 算法通常既用于表示按照顶点与源点的距离的顺序增加顶点来构建 SPT 的抽象方法，又指这种抽象方法的实现（作为邻接矩阵表示的 V^2 算法），因为 Dijkstra 在其 1959 年的论文中提出了这两种方法（同时还显示了这种方法还可以计算 MST）。对于稀疏图的性能改进依赖于后来 ADT 技术以及优先队列实现上的改进，这些改进并不是特定为最短路径问题所提出的。Dijkstra 算法的改进性能是这项技术最为重要的应用之一（见第 5 部分参考文献）。正如对于 MST 中的做法，我们使用诸如“使用 d-叉堆的 Dijkstra 算法的 PFS 实现”这一术语来明确特定的组合。

在 18.8 节我们已经看到，非加权有向图中对优先级采用前序编号可使对优先队列的操作像一个 FIFO 队列，并得到一个 BFS。Dijkstra 算法给出了 BFS 的另一种实现：在所有权值都为 1 时，按照距起始顶点最短路径的边数的顺序来访问顶点。在这种情况下，优先队列上的操作并不完全像 FIFO 队列上的一样，因为有相同优先级的元素不必按照它们进入队列时的顺序出队列。

各种实现使用顶点索引的参数数组 st 来构建从顶点 0 起始的 SPT 的父链接表示，同时使用顶点索引的参数数组 wt 来存放 SPT 中每个顶点的最短路径长度。如常，我们可以围绕这种模式来构建各种便利的 ADT 函数（见练习 21.19 ~ 练习 21.28）。

练习

▷ **21.17** 按照图 21-10 的风格，显示利用 Dijkstra 算法计算练习 21.1 中所定义的网的 SPT 的结果，起始顶点为 0。

○ **21.18** 如何找出一个网中从 s 到 t 的次短（second）路径。

▷ **21.19** 在标准网 ADT 中增加一个函数，使用 GRAPHspt 来计算连接两个给定顶点 s 和 t 的一条最短路径的长度。

21.20 在标准网 ADT 中增加一个函数，使用 GRAPHspt 来找出距一个给定顶点 s 最远的顶点（距 s 的最短路径最长的那个顶点）。

21.21 在标准网 ADT 中增加一个函数，使用 GRAPHspt 来计算从一个给定顶点到由它可达的每个顶点的最短路径的平均长度。

○ **21.22** 在标准邻接表网 ADT 中增加一个函数，使用 GRAPHspt 来计算连接两个给定顶点 s 和 t 的一条最短路径的链表表示，来求解源点 - 汇点最短路径问题。

○ **21.23** 在标准网 ADT 中增加一个函数，通过 GRAPHspt 使用连接两个给定顶点 s 和 t 的一条最短路径的连续顶点索引，填充一个给定的参数数组的初始元素，来求解源点 - 汇点最短路径问题。

21.24 开发一个基于程序 21.1 的接口和实现，将连接两个给定顶点 s 和 t 的一条最短路径压入用户提供的栈中。

21.25 在标准邻接表网 ADT 中增加一个函数，找出一个网中距给定顶点的距离在 d 内的所有顶点。你的函数的运行时间应该与这些顶点以及依附于它们的顶点所导出的子图的规模成正比。

21.26 开发找出一个给定网中一条边的算法，使得删除此边导致从一个给定顶点到另一个给定顶点的最短路径长度有最大幅度的增加。

● **21.27** 在标准邻接矩阵网 ADT 中增加一个函数，对于给定的一对顶点 s 和 t，对网中的边进行*敏感性分析*（*sensitivity analysis*）：计算一个 $V \times V$ 数组，使得对于每个 u 和 v，如果 u-v 是网中的一条边，且其权值增加时，从 s 到 t 的最短路径不会增加，则 u 行、v 列的元素值为 1；否则，此处的值为 0。

○ **21.28** 在标准邻接表网 ADT 中增加一个函数，找出给定网中连接一个给定顶点集与另一给定的顶点集的一条最短路径。

21.29 使用练习 21.28 的解决方案来实现找出随机网格网（见练习 20.7）的一条从左边到右边的最短路径。

21.30 显示一个无向图的 MST 等价于此图的瓶颈 SPT（bottleneck SPT）：对于每对顶点 v 和 w，给出连接这对顶点的路径，使其最长边尽可能短。

21.31 进行实验研究，对于各种类型的网（见练习 21.4～21.8），给出本节所描述的针对稀疏图的两种版本的 Dijkstra 算法（程序 21.1 和程序 20.4，使用合适的优先级定义）性能比较。使用标准堆优先队列实现。

21.32 进行实验研究，对于各种类型的网（见练习 21.4～21.8），对于所讨论过的 3 种 PFS 的每种实现（程序 18.10、程序 20.4 和程序 21.1），给出使用 d-叉堆优先队列实现（见程序 20.7）的最好 d 值。

● **21.33** 进行实验研究，对于各种类型的网（见练习 21.4～21.8），确定在程序 21.1 中使用索引－堆－锦标赛优先队列实现（见练习 9.53）的效果。

○ **21.34** 进行实验研究，对于各种类型的网（见练习 21.4～21.8），分析 SPT 中的高度及平均路径长度。

21.35 开发一个源点－汇点最短路径问题的实现，使用源点和汇点初始化队列。这样做可使自每个顶点开始的 SPT 增长；你的主要任务是正确地确定当两个 SPT 冲突时应该怎样做。

● **21.36** 描述 V 个顶点、E 条边的一组图，使得 Dijkstra 算法对于每个图都达到最坏情况运行时间。

●● **21.37** 开发一个 V 个顶点、E 条边的随机图的合理生成器，使得 Dijkstra 算法的基于堆的 PFS 实现的运行时间为超线性的。

● **21.38** 编写一个客户程序，显示 Dijkstra 算法的动态图形动画过程。你的程序应该产生类似图 21-11 的图像（见练习 17.55～17.59）。使用随机欧几里得网（见练习 21.8）测试你的程序。

21.3 所有对最短路径

在这一节里，我们要考虑一个 ADT 以及用于求解所有对最短路径问题的两个基本实现。所实现的算法直接推广了在 19.3 节中求解传递闭包问题时所考虑的两个基本算法。第一种方法是对每个顶点运行 Dijkstra 算法，得到从该顶点到其余顶点的最短路径。如果使用一个堆实现此优先队列，那么这种方法的最坏情况下的运行时间与 $V E \lg V$ 成正比，而且对于很多类型的网，使用 d-叉堆可以将这个界限改进为 VE。第二种方法，使我们可以直接在与 V^3 成正比的时间内求解这个问题。这是 Warshall 算法的一个扩展，称为 Floyd 算法（Floyd algorithm）。

程序 21.2 所有对最短路径 ADT

所有对最短路径问题的解为客户程序提供了一个预处理函数 GRAPHspALL 及两个查询函数：一个用于返回从第一个参数到第二个参数的最短路径长度（GRAPHspDIST），另一个用于返回从第一个参数到第二个参数的最短路径上的下一个顶点（GRAPHspPATH）。由约定，如果不存在这样的路径，GRAPHspPATH 返回 G->V，且 GRAPHspDIST 返回一个大于此最长路径长度的观察哨值。

```
  void GRAPHspALL(Graph G);
double GRAPHspDIST(Graph G, int s, int t);
   int GRAPHspPATH(Graph G, int s, int t);
```

可以使用其中任一算法来实现一个预处理函数，以支持网 ADT 中的*抽象最短路径*（abstract shortest-path）函数，该函数可在常量时间内返回任何两个顶点的最短路径，如同在第 19 章中基于计算传递闭包来构建 ADT 从而处理连通性查询。程序 21.2 是一个接口，它描述了可增加到标准网 ADT 中的三个函数，从而支持以这种方式找出最短距离和路径。第一个函数是一个预处理函数，第二个函数是一个返回从一个给定顶点到另一个顶点的最短路径长度的查询函数，第三个函数是一个返回从一个给定顶点到另一个顶点的最短路径的下一个顶点的查询函数。支持此 ADT 是在实际中使用所有对最短路径算法的主要原因。

程序 21.3 计算网的直径

此客户程序说明了程序 21.2 中的接口的使用。它在给定的网中找出最短路径中的最长路径，并打印其权值（网的直径）及该路径。

```
void GRAPHdiameter(Graph G)
  { int v, w, vMAX = 0, wMAX = 0;
    double MAX = 0.0;
    GRAPHspALL(G);
    for (v = 0; v < G->V; v++)
      for (w = 0; w < G->V; w++)
        if (GRAPHspPATH(G, v, w) != G->V)
          if (MAX < GRAPHspDIST(G, v, w))
            { vMAX = v; wMAX = w;
              MAX = GRAPHspDIST(G, v, w); }
    printf("Diameter is %f\n", MAX);
    for (v = vMAX; v != wMAX; v = w)
      { printf("%d-", v);
        w = GRAPHspPATH(G, v, wMAX); }
    printf("%d\n", w);
  }
```

程序 21.3 是一个示例客户程序，使用最短路径 ADT 函数来找出一个网的*加权直径*（weighted diameter）。它调用预处理函数，然后检查所有顶点对来找出最短路径长度是最长的一对顶点；然后，一个顶点一个顶点地遍历此路径。图 21-13 显示了此程序对于欧几里得网示例所计算的路径。

本节的算法目标是支持查询函数的常量时间实现。通常，我们可以预期到有大量这样的请求，因而愿意投入足够的内存资源，并进行预处理从而快速满足这些请求。

因此，使用预处理函数实现了求解所有对最短路径问题问题，然后在查询函数中简单访问这个解。假设扩展了标准网 ADT，使其包含了指向两个矩阵的指针：一个是用于距离矩阵的 $V \times V$ 数组 G->dist，另一个是用于路径表的 $V \times V$ 数组 G->path。对于每对顶点 s 和 t，我们所考虑的两个算法的预处理函数都将 G->dist[s][t] 设置为从 s 到 t 的最短路径长度，将 G->path[s][t] 设置为从 s 到 t

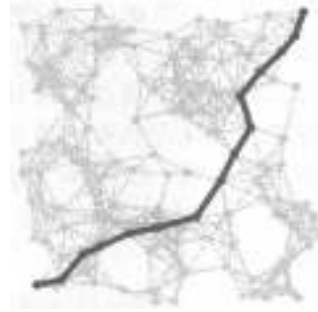


图 21-13 网的直径

网的所有最短路径矩阵中最大的是网的直径：即最短路径中的最长路径的长度。这里显示了示例欧几里得网的直径。

的最短路径上下一个顶点的索引；其他两个 ADT 函数在常量时间内只返回这些值。

程序 21.4 所有对最短路径的 Dijkstra 算法

程序 21.2 中的接口实现在逆网上使用 Dijkstra 算法来找出到每个顶点的所有最短路径（见正文）。

假设网的数据类型有一个指针 dist 用于距离数组，一个指针 path 用于路径数组。Dijkstra 算法所计算的 wt 和 st 数组分别是距离矩阵和路径矩阵中的列（见图 21-9）。

```
static int st[maxV];
static double wt[maxV];
void GRAPHspALL(Graph G)
  { int v, w; Graph R = GRAPHreverse(G);
    G->dist = MATRIXdouble(G->V, G->V, maxWT);
    G->path = MATRIXint(G->V, G->V, G->V);
    for (v = 0; v < G->V; v++)
      {
        GRAPHpfs(R, v, st, wt);
        for (w = 0; w < G->V; w++)
          G->dist[w][v] = wt[w];
        for (w = 0; w < G->V; w++)
          if (st[w] != -1) G->path[w][v] = st[w];
      }
  }
double GRAPHspDIST(Graph G, int s, int t)
  { return G->dist[s][t]; }
int GRAPHspPATH(Graph G, int s, int t)
  { return G->path[s][t]; }
```

这种通用方法的主要缺点为，对于大型网，我们可能不能承受预处理的时间或没有可用空间存放这些表。原则上，此接口为我们提供了权衡预处理时间和空间与查询时间的标尺。如果只是几个查询，可以不进行预处理，只对每次查询简单运行单源点算法，但是中间情形则需要更为高级的算法（见练习 21.48 ~ 21.50）。对于第 19 章的大部分内容有个难题：使用有限空间支持常量时间可达性查询，而这个问题则是此难题的推广。

我们考虑的第一个所有对最短路径 ADT 函数实现是对于每个顶点，用求解单源点问题的 Dijkstra 算法解决了这个问题。对于每个顶点 s，使用对于 s 的单源点最短路径问题的解 wt 数组来填充距离矩阵中的第 s 行。这种方法将在 17.7 节中所考虑的非加权有向图的基于 BFS 的方法进行了推广。它也类似于程序 19.4 中使用在每个顶点开始的 DFS 来计算非加权有向图的传递闭包。

如在图 21-8 和 21-9 中说明的，计算路径矩阵稍微复杂一些。因为在数组 st 中的父链接 SPT 表示给出的边是错误的方向。这个困难在无向图中并不会出现，因为所有边都有两个方向。为了解决这个问题，我们处理逆网，并在逆网中用 t 的 SPT 父链接表示（数组 st）填充路径表中的 t 列。由于最短路径长度在两个方向相同，也可以用从 t 开始的单源点最短路径问题的解 wt 数组填充距离矩阵的 t 列。

程序 21.5 所有对最短路径的 Floyd 算法

GRAPHspALL 的此段代码（以及程序 21.4 的 GRAPHspDIST 和 GRAPHspPATH 的在线实现）使用 Floyd 算法实现了程序 21.2 中的接口，它是 Warshall 算法的一种推广（见程序

19.4），它找出了最短路径而不只是检查路径的存在性。

使用图的边初始化距离矩阵和路径矩阵后，我们要做一系列松弛操作，来计算最短路径。此算法简单，易于实现，但验证它计算出了最短路径较为复杂（见正文）。

```
void GRAPHspALL(Graph G)
  { int i, s, t;
    double **d = MATRIXdouble(G->V, G->V, maxWT);
    int **p = MATRIXint(G->V, G->V, G->V);
    for (s = 0; s < G->V; s++)
      for (t = 0; t < G->V; t++)
        if ((d[s][t] = G->adj[s][t]) < maxWT)
          p[s][t] = t;
    for (i = 0; i < G->V; i++)
      for (s = 0; s < G->V; s++)
        if (d[s][i] < maxWT)
          for (t = 0; t < G->V; t++)
            if (d[s][t] > d[s][i]+d[i][t])
              { p[s][t] = p[s][i];
                d[s][t] = d[s][i]+d[i][t]; }
    G->dist = d; G->path = p;
  }
```

程序21.4是一种基于这些想法的ADT实现。它可用于邻接矩阵表示，也可用于邻接表表示，但主要用于稀疏图，因为它利用了Dijkstra算法对于此类图的高效性。

性质21.7 *对于一个有非负权值的网，使用Dijkstra算法，可以在与$V E \log_d V$成正比的时间内找出其中的所有最短路径，如果$E<2V$，那么$d=2$，否则$d=E/V$。*

证明 由性质21.6直接可得。 ■

正如对于单源点最短路径和MST问题的界限，这个界限也是保守的，而且VE的运行时间对于一般图很可能。

对于稠密图，可以使用一个邻接矩阵表示，并通过隐式地转置矩阵（交换行和列索引）来避免计算逆图，类似于程序19.8中的做法。根据这些原则来开发实现是一个很有趣的程序设计练习，并且可以得到一个简洁的实现（见练习21.43）；然而，我们下面将考虑一种不同的方法，会得到一种更为简洁的实现。

选择用于求解稠密图中的所有对最短路径算法问题的方法是由R. Floyd开发的，除了在使用逻辑或操作来记录路径的存在性之外，它与Warshall方法完全相同。该方法将检查每条边的距离来确定该边是否是一条更短路径的一部分。实际上，我们已经提到过，Floyd和Warshall算法在合理抽象级上是相同的（见19.3节和21.1节）。程序21.5是实现Floyd算法的所有对最短路径的ADT实现。

性质21.8 *利用Floyd算法，可以在与V^3成正比的时间内计算出一个网中的所有最短路径。*

证明 由代码直接可得运行时间。使用归纳法来证明算法的正确性，与在Warshall算法中的证明方法相同。循环的第i次迭代计算网中从s到t的一条最短路径，此路径中不含索引大于i的任何顶点（除了端点s和t之外）。假设对于第i次的循环结论成立，我们证明对于第$(i+1)$次的循环结论也成立。从s到t的一条不含索引大于$i+1$的顶点的最短路径，要么是（ⅰ）从s到t的不含索引大于的i的任何顶点且长度为d[s][t]的路径，由归纳假设，它是在此循环的上一次迭代中找到的；要么是（ⅱ）包含从s到i的一条路径和从i到t的一条路径，

这两条路径都没不含索引大于 i 的任何顶点，在这种情况下，内循环将设置 d[s]t[]。 ■

图 21-14 是 Floyd 算法应用于示例网的详细跟踪过程。如果将每个空元素转换为 0（指示该边不存在），并将每个非空元素转换为 1（指示该边存在），那么这些矩阵描述了 Warshall 算法的操作过程，就和在图 19-15 中的方式一样。对于 Floyd 算法，非空元素不仅表明存在路径；还给出了关于最短已知路径的信息。距离矩阵中的元素为连接对应给定行和列的顶点的已知最短路径的长度。路径矩阵中对应的给出了那条路径的下一个顶点。当这些矩阵被非空元素填满时，运行 Warshall 算法就相当于重复检查新的路径，它们连接了已经知道有一条路径连接的每对顶点。反之，Floyd 算法必须比较（如果需要还要更新）每个新的路径，来查看是否新路径会导致更短的路径。

要比较 Dijkstra 算法和 Floyd 算法的最坏情况下的运行时间界限，可以采用类似于 19.3 中的相应传递闭包算法的做法，得出这些所有对最短路径算法的同样的结论。对于稀疏网，显然选择对每个顶点运行 Dijkstra 算法，因为此运行时间近似为 VE。随着密度增加，Floyd 算法（其所需时间总是为 V^3）开始具有竞争性（见练习 21.67）；由于其简单易于实现，因此应用广泛。

这些算法之间还有一个更为基本的差别，即 Floyd 算法甚至在有负权值的网中也是有效的（假设不存在负环），21.7 节将检查这些差别。21.2 节中已经提到，在这样的图中，Dijkstra 算法不一定能找出最短路径。

对于所有对最短路径问题，我们所描述的经典解决方案均假设有可用的空间来保存距离矩阵和路径矩阵。对于大型的稀疏图，则无法承受任何 $V \times V$ 矩阵，这样就带来另外一些困难和有趣的问题。在第 19 章已经看到，将此空间开销减少为与 V 成正比，同时仍能支持常量时间的最短路径查询，这是一个开放问题。甚至我们发现，对于更简单的可达性问题（即满足于能够在常量时间内知道是否存在连接一个给定的顶点对的任何一条路径）的算法也很困难，因此不能期望对于所有对最短路径问题有一个简单的解决方案。实际上，即使对于稀疏图，不同最短路径长度的数目通常也与 V^2 成正比。从某种意义上说，这个值度量了需要处理的信息量，而且可能表明我们在空间上确实有限制，所以可能必须在各个查询上花费更多的时间（见练习 21.48 ~ 21.50）。

练习

▷ **21.39** 如果要求使用 Floyd 算法在 10 秒内计算出一个图的所有最短路径，估计你的计算机和程序设计系统所能处理的密度为 10 的最大图（使用顶点数度量），数量级为 10。

▷ **21.40** 如果要求使用 Dijkstra 算法在 10 秒内计算出一个图的所有最短路径，估计你的计算机和程序设计系统所能处理的密度为 10 的最大图（使用边数度量），数量级为 10。

21.41 按照图 21-9 的风格，显示利用 Dijkstra 算法计算练习 21.1 中所定义的网的所有最短路径的结果。

21.42 按照图 21-14 的风格，显示利用 Floyd 算法计算练习 21.1 中所定义的网的所有最短路径的结果。

○ **21.43** 组合程序 20.3 和程序 21.4 形成一个用于稠密网的所有对最短路径 ADT 接口实现（基于 Dijkstra 算法），使得不需要显式计算逆网。请不要为 GRAPHpfs 定义另一个函数，只要将程序 20.3 中的代码直接放入内循环中，消除参数数组 wt 和 st，并将结果直接 G -> dist 和 G -> path（或像程序 21.5 那样使用局部数组 d 和 p）。

21.44 进行实验研究，对于各种类型的网（见练习 21.4 ~ 21.8），对 Dijkstra 算法（程序 21.4 和练习 21.43）和 Floyd 算法（程序 21.5）进行比较。

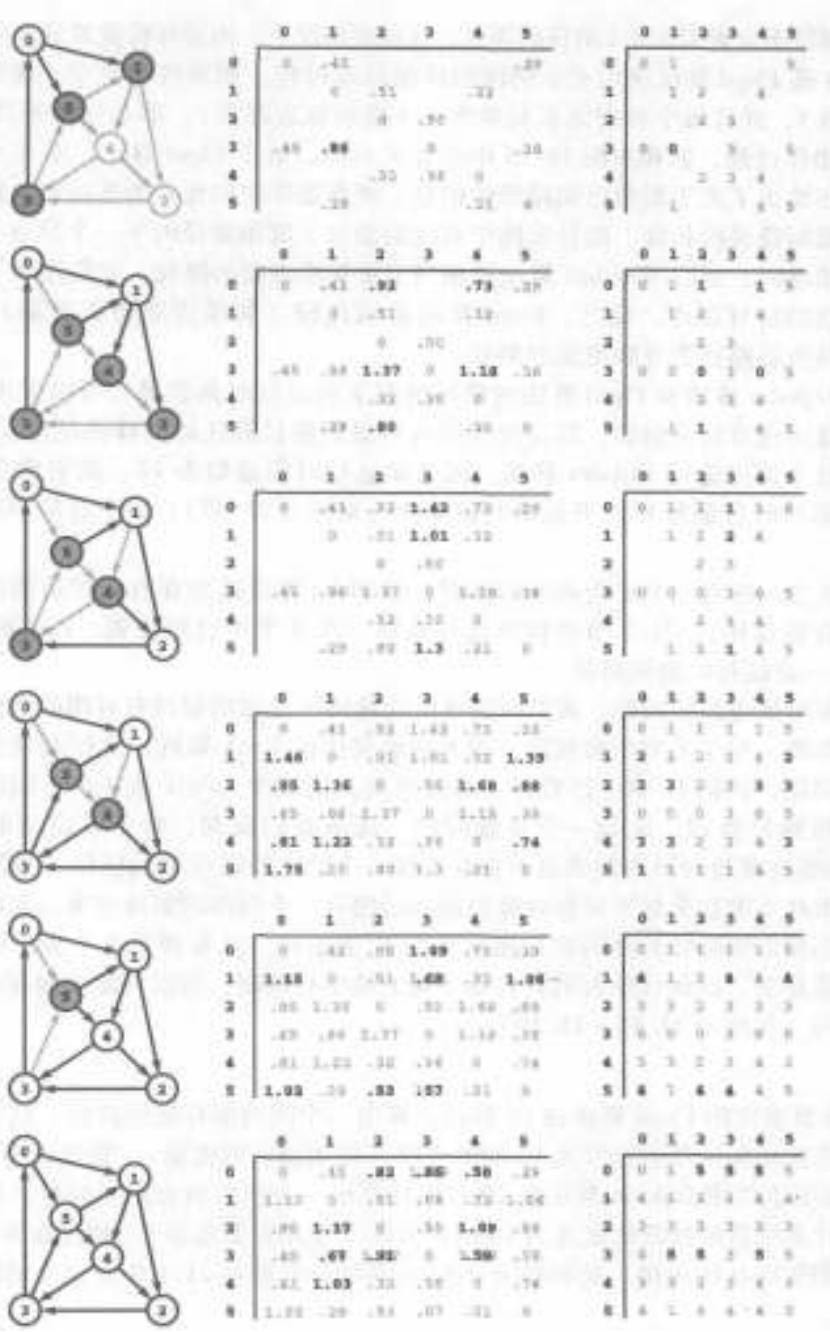


图 21-14　Floyd 算法

此序列显示了使用 Floyd 算法构造所有对最短路径矩阵的过程。对于 i 从 0 到 5（从上图到下图），对于所有 s 和 t，考虑从 s 到 t 的所有路径，其中没有大于 i 的中间顶点（加阴影的顶点）。初始时，唯一的路径就是网中的边，因而距离矩阵（中图）就是图的邻接矩阵，而用 p[s][t] = t 设置每条边就得路径矩阵（右图）。对于顶点 0（最上图），由于不存在边 3-1，算法找出 3-0-1 比现有观察哨值更短，因此要更新矩阵。对于诸如 3-0-5 等路径则不会这样做，因此它不比已知的 3-5 更短，接下来，算法将考虑通过 0 和 1 的路径（自上第二个图），并找出新的更短路径 0-1-2、0-1-4、3-0-1-2、3-0-1-4 和 5-1-2。上数第 3 行显示了对应通过 0、1 和 2 的更短路径的更新，如此继续。

将矩阵中的灰色改成黑色表示出，算法找出了一条较之以前找到的更短路径。例如，下图中的 3 行、2 列处 1.37 改成了 0.91 是因为算法发现 3-5-4-2 是比 3-0-1-2 更短的路径。

21.45 进行实验研究，对于各种类型的网（见练习21.4～21.8），确定Floyd算法和Dijkstra算法对距离矩阵中的元素更新的次数。

21.46 给出一个矩阵，其中s行和t列的元素等于图21-1中连接s和t的简单不同有向路径的数目。

21.47 实现一个网ADT函数，计算练习21.46中所描述的路径数矩阵。

21.48 开发稀疏图的一个抽象最短路径ADT的实现，通过将查询时间增加到与V成正比，从而将空间开销降低到与V成正比。

● 21.49 开发稀疏图的一个抽象最短路径ADT的实现，使用空间远小于$O(V^2)$，但支持查询时间远小于$O(V)$。提示：计算顶点某个子集的所有对最短路径。

● 21.50 开发稀疏图的一个抽象最短路径ADT的实现，使用空间远小于$O(V^2)$，并（使用随机化）支持常量期望查询时间。

○ 21.51 开发一个抽象最短路径ADT的实现，采用一种“懒”方法，即当客户程序首次提出以s起始的最短路径查询时，使用Dijkstra算法来构建SPT（以及相关距离向量），然而，在以后的查询中引用这些向量。

21.52 修改最短路径ADT和Dijkstra算法，来处理网中顶点和边均有权值的最短路径计算。请不要重建该图的表示（练习21.3中描述了此方法）；修改代码即可。

● 21.53 构建航班路线和连接时间的一个小型模型，可以基于你已经采用的某些航班。使用练习21.52的解决方案来计算从某个航班终点到另一航班终点的最快路线。然后，用真实数据测试你的程序（见练习21.4）。

21.4 无环网中的最短路径

在第19章中，我们发现，尽管直观地可以看出DAG要比一般有向图要易于处理，但与一般有向图相比，对于DAG要开发性能更好的算法是一个极为困难的目标。对于最短路径问题，确实有一些DAG的算法，要比我们考虑过的用于一般有向图的基于优先队列的方法更简单、快速。具体地说，本节我们将考虑无环网的算法，如下：

- 在线性时间内求解单源点问题。
- 在与VE成正比的时间内求解所有对问题。
- 解决其他问题，如找最长路径。

在前两种情况下，可以将运行时间中的对数因子去掉，该因子出现在对于稀疏图的最好的算法中；在第三种情况下，对于一般网，有些难解的问题有简单算法。这些简单的算法都是第19章中所考虑的DAG中对于可达性和传递闭包的算法的直接扩展。

由于根本不存在环，也就不存在负环；因此负权值不会对DAG上的最短路径问题带来困难。因此，本节对边权值不做任何限制。

接下来，对于术语的解释：称边上有权值且不存在环的有向图为*加权DAG*（weighted DAG）或*无环网*（acyclic network）。我们交替使用这两个术语，以强调它们的等价性。在引用文献时也可避免混淆，它们在文献中得到广泛使用。有时使用前者来强调与有隐含权值的非加权DAG的差别，有时则使用后者来强调无环性所隐含的一般网的差异，这会很方便。

我们应用的4种基本思路可以导出第19章中的未加权DAG的高效算法，这些思路甚至对于加权DAG更有效。

- 使用DFS求解单源点问题。
- 使用源队列求解单源点问题。

- 调用上述其中任意一种方法，对于每个顶点调用一次，求解所有对问题。
- 使用单个 DFS（及动态规划）求解所有对问题。

这些方法可在与 E 成正比的时间内解决单源点问题，而在与 VE 成正比的时间内解决所有对问题。它们都是有效的。这是因为存在拓扑排序，使我们可以为每个顶点计算最短路径而不需作重新访问。本节，将对各个问题考虑一个实现：其他实现则留作练习（见练习 21.62～21.65）。

我们稍微换个角度。每个 DAG 至少有一个源点，但可能有多个源点，因此，自然考虑以下最短路径问题：

多源点最短路径问题 给定一组起始顶点，对于每个其他顶点 w，找出从每个起始顶点到 w 的最短路径中的一条最短路径。

此问题基本等价于单源点最短路径问题。可以增加一个虚拟顶点，将多源点问题转化为单源点问题，其中该虚拟顶点到网中其他顶点的边的长度为 0。相反，可以通过处理由所有顶点及源点可达的边所定义的导出子网，将单源点问题转换为多源点问题。我们很少会显式构造这样的子网，因为如果将起始顶点处理为好像它是网中唯一的源点（即使并非如此），算法就会自动对这些子网进行处理。

拓扑排序直接给出了多源点最短路径问题以及很多其他问题的一种解决方案。我们维护一个顶点索引数组 wt，此数组给出了从任何源点到每个顶点已知的最短路径的权值。为了解决多源点最短路径问题，将数组 wt 初始化，对于源点初始化为 0，其他顶点初始化为 MAXwt。然后，按照拓扑排序的顺序处理顶点。为了处理顶点 v，对每条离开边 v-w 执行一个松弛操作，如果 v-w 给出从一个源点到 w（通过 v）的一条更短路径，则更新到 w 的最短路径。这个过程检查从任一源点到图中每个顶点的所有路径；松弛操作保存最小长度的此类路径，而拓扑排序保证了按照一种合适的顺序处理顶点。

可以采用以下两种方法之一来直接实现此方法。第一种方法是在程序 19.8 中的拓扑排序中增加几行代码：在从源点队列中删除一个顶点 v 之后，对于该顶点的每条边执行指示的松弛操作（见练习 21.56）。第二种方法是将顶点按照拓扑排序放置，然后扫描它们，并完全按照上一段述的描述执行松弛操作。这些相同的过程（以及其他松弛操作）可以解决很多图处理问题。例如，程序 21.6 是第二种方法（排序，然后扫描）求解多源点最长路径（multisource longest path）问题的一种实现：对于网中的每个顶点，从某个源点到那个顶点的最长路径是什么？

可以将关联每个顶点的 wt 中的元素解释为从任一源点到那个顶点的已知最长（longest）路径的长度，将所有权值均初始化为 0，用松弛操作改变比较的意义。图 21-15 跟踪了程序 21.6 在示例无环网上的操作过程。

性质 21.9 *可以在线性时间内解决无环网中的多源点最短路径问题和多源点最长路径问题。*

证明 对于最长路径、最短路径以及一些其他路径性质均可使用相同的证明。为与程序 21.6 一致，我们给出最长路径的证明过程。对循环变量 i 使用归纳法证明。对于已经处理的所有顶点 v = ts[j]，且 $j < i$，wt[v] 是从一个源点到 v 的最长路径长度。在 v = ts[i] 时，令 t 是从源点到 v 的任何路径上在 v 之前的顶点。因为 ts 数组中的顶点是按照逆拓扑有序的顺序，t 必定已经被处理过。由归纳假设，wt[t] 是到 t 的最长路径的长度，且代码中的松弛步检查该路径是否给出通过 t 到 v 的一条更长的路径。归纳假设还蕴含着当处理 v 时，按照这种方式检查到 v 的所有路径。 ■

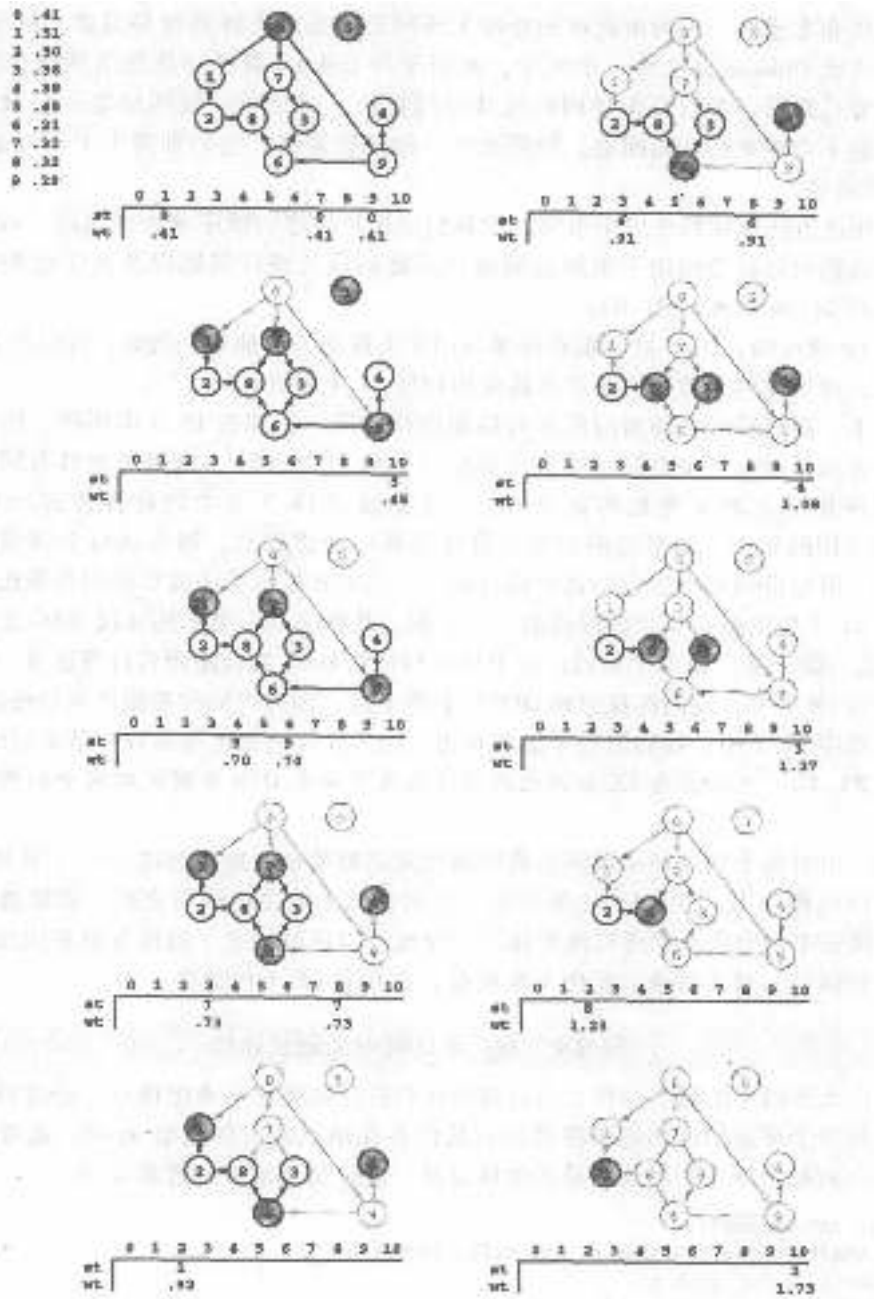


图 21-15 计算无环网中的最长路径

在这个网中，每条边都关联一个顶点的权值，该边由此顶点发出。在左上图列出。汇点有一条边指向虚拟顶点10，未在图中显示。wt 数组含有从某个源点到每个顶点的已知最长路径的长度，st 数组含有这条最长路径上的前一个顶点。此图展示了程序 21.6 的操作过程。使用 FIFO 原则从源点中进行选择（每个图中有阴影的顶点），然而在每一步中每个源点都可能被选。通过删除 0 开始，并检查它的每条依附边，找出到达 1、7 和 9 且仅有一条边的路径长度为 0.41。接下来，删除 5，并记录从 5 到 10 的含一条边的路径（左边自上第 2 个图）。然后，删除 9，并记录长度为 0.70 的路径 0-9-4 和 0-9-6（左边自上第 3 个图）。按照这种方式继续，并在找到更长的路径时修改数组。例如，删除 7（左边自下第 2 个图），记录到达 8 和 3 的路径长度 0.73；然后，删除 6，记录到达 8 和 3 的更长的路径（长度为 0.91）（右上图）。计算的关键是找出到虚拟结点 10 的最长路径。在这种情况下，路径是 0-9-6-8-2，其长度为 1.73。

此性质非常重要，因为由此可知处理无环网要比处理有环网容易得多。对于最短路径，源－队列法比 Dijkstra 算法快一个因子，此因子与 Dijkstra 算法中优先队列操作的开销成正比。对于最长路径，我们有无环网的线性时间算法，而对于一般网却是一个难解问题。此外，负权值不会带来特殊的困难，但是对于一般网的算法，它们则带来了不可逾越的障碍，如 21.7 节所述。

以上所述方法仅依赖于以下事实，即我们以拓扑排序的顺序来处理顶点。因此，任何拓扑排序算法都可以被改编用于求解最短路径问题和最长路径问题以及其他此类的问题（例如，见练习 21.56 和练习 21.62）。

由第 19 章可知，DAG 抽象是在许多应用中出现的一般抽象。例如，可以在 21.6 节看到一个应用，看似与网无关，但是可直接使用程序 21.7 解决。

接下来，我们转到无环网的所有对最短路径问题。正如在 19.3 中那样，用于求解此问题的一种方法是对每个顶点运行单源点算法（见练习 21.65）。这里考虑具有同样效果的一种方法是使用带有动态规划的单个 DFS，这类似于 19.5 节中的计算传递闭包（见程序 19.9）所采用的做法。如果按照逆拓扑排序的顺序考虑顶点，那么从每个邻接顶点的最短路径向量，可以得到每个顶点的最短路径向量，只需在松弛步中简单使用每条边。

程序 21.7 是按照这些原则得到的一个实现。此程序在一个示例加权 DAG 上的操作如图 21-16 所示。除了推广加入了松弛，这个计算与针对 DAG 的传递闭包计算还有一个重要的区别：在程序 19.9 中，我们选择忽略 DFS 树中的下边，因为它们没有提供可达性的新的信息；然而，在程序 21.7 中，我们需要考虑所有边，因为任何边都可能导致一条更短的路径。

性质 21.10 *可以在与 VE 成正比的时间内使用单个 DFS 求解无环网中的所有对最短路径问题。*

证明 由对每个顶点使用单源点最短路径问题的策略直接可得这一点（见练习 21.65）。我们也可以由程序 21.7 用归纳法来证明。在对顶点 v 做递归调用之后，就知道已经计算出了 v 的邻接表中每个顶点的所有最短路径，因此可以通过检查 v 的每条边来找出从 v 到每个顶点的最短路径。对于每条边要做 V 次松弛，总共有 VE 个松弛步。 ■

程序 21.6 无环网中的最长路径

要找出无环网中的最长路径，可以按照拓扑排序的顺序来考虑顶点，通过对每条边进行松弛，将到每个顶点的已知最长路径的权值保存在顶点索引的数组 wt 中。此程序还计算出从一个源点到每个顶点的最长路径的生成森林，放在顶点索引的数组 st 中。

```
static int ts[maxV];
void GRAPHlpt(Graph G, int s, int st[], double wt[])
  { int i, v, w; link t;
    GRAPHts(G, ts);
    for (v = ts[i = 0]; i < G->V; v = ts[i++])
      for (t = G->adj[v]; t != NULL; t = t->next)
        if (wt[w = t->v] < wt[v] + t->wt)
          { st[w] = v; wt[w] = wt[v] + t->wt; }
  }
```

因此，对于无环网，逆拓扑排序使我们可以避免 Dijkstra 算法中优先队列的开销。类似 Floyd 算法，程序 21.7 所求解的问题比 Dijkstra 算法求解的问题更具一般性，因为不像 Dijkstra 算法（见 21.7 节），该算法即使对于出现负边权值的情况也能处理，如果在无环网中使

所有权值为负，运行此算法，那么它找出所有最长路径。如图21-17所示。或者，在松弛算法中，将不等式检查反向，可以找出最长路径，如程序21.6。

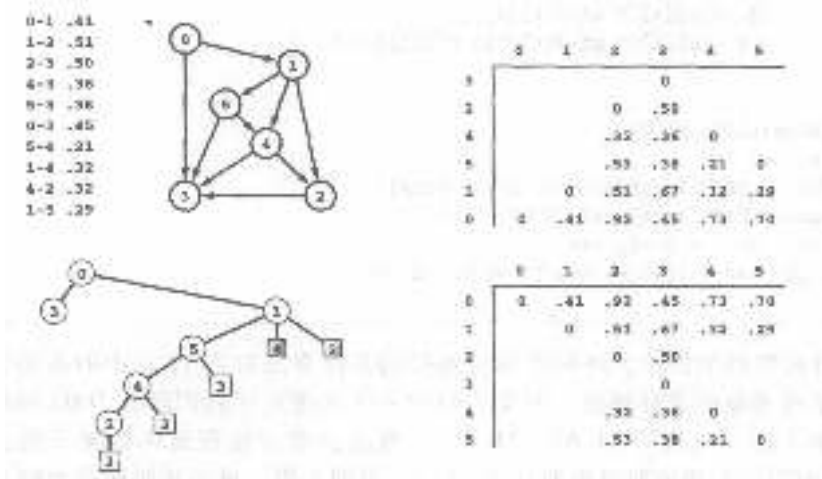


图21-16 无环网中的最短路径

此图描述了一个示例加权DAG（左上图）的所有对最短距离的计算（右下图），计算每一行作为递归DFS函数中的最后一个操作。由邻接顶点的相应行来计算出这一行，这些行出现在表的前面，因为是按照逆拓扑顺序来计算每一行的（左下图显示了后序遍历此DFS树）。右上图中的数组显示了按计算的顺序所得出的矩阵的行。例如，要计算对应于0的行中的元素，在对应于1的相应行的元素增加0.41（采用0-1之后，得到从0到该顶点的距离），然后在对应于3的相应行的元素增加0.45（采用0-3之后，得到从0到该顶点的距离），并采用这两者中的较小者。这里的计算与DAG传递闭包的计算基本相同（例如，见图19-23）。这两者最大的区别是传递闭包算法可能忽略下边（此例中为1-2），因为它们走到已知可达的顶点，而最短路径算法必须检查关联下边的路径要比已知路径更短。如果要忽略此例中的1-2，就会找不到最短路径0-1-2和1-2。

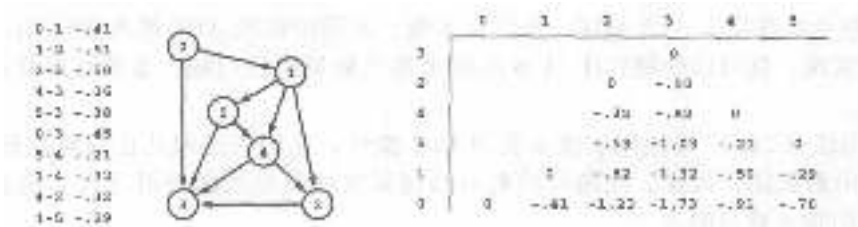


图21-17 无环网中的所有最长路径

计算无环网中所有最长路径的方法即使对于负权值也能处理。因此，可以使用此方法来计算最长路径，只需先简单对所有权值取负，类似图21-16中对于网的描述。最长简单路径是0-1-5-4-2-3，权值为1.73。

程序21.7 无环网中的所有最短路径

GRAPHspALL的针对加权DAG的实现是通过在程序19.10的基于动态规划的传递闭包函数中增加合适的松弛操作得到的。

```
void SPdfsR(Graph G, int s)
  { link u; int i, t; double wt;
    int **p = G->path; double **d = G->dist;
    for (u = G->adj[s]; u != NULL; u = u->next)
      {
        t = u->v; wt = u->wt;
        if (d[s][t] > wt)
          { d[s][t] = wt; p[s][t] = t; }
```

```
        if (d[t][t] == maxWT) SPdfsR(G, t);
        for (i = 0; i < G->V; i++)
          if (d[t][i] < maxWT)
            if (d[s][i] > wt+d[t][i])
              { d[s][i] = wt+d[t][i]; p[s][i] = t; }
      }
  }
void GRAPHspALL(Graph G)
  { int v;
    G->dist = MATRIXdouble(G->V, G->V, maxWT);
    G->path = MATRIXint(G->V, G->V, G->V);
    for (v = 0; v < G->V; v++)
      if (G->dist[v][v] == maxWT) SPdfsR(G, v);
  }
```

在本节开始提到的找出无环网中最短路径的其他算法对第 19 章中的方法做了推广，方法类似于本章所考察的其他算法。开发它们的一个实现对于巩固理解 DAG 和最短路径是一件很有意义的工作（见练习 21.62 ~ 21.65）。所有这些方法在最坏情况下的运行时间都与 *VE* 成正比，而实际的开销取决于 DAG 的结构。原则上说，对于某些稀疏加权 DAG，还可以做得更好（见练习 19.119）。

练习

▷ **21.54** 给出练习 21.1 中定义的网的多源点最短路径问题和多源点最长路径问题的解决方案，其中将边 2-3 和 1-0 反向。

▷ **21.55** 修改程序 21.6，使其可以求解无环网的多源点最短路径问题。

▷ **21.56** 给出 GRAPHlpt 的一种实现，由程序 19.8 的基于源点队列的拓扑排序代码导出，对于每个顶点要求将此顶点从此源点队列中删除后执行松弛操作。

○ **21.57** 为松弛操作定义一个 ADT，提供其实现。并使用你的 ADT 修改程序 21.6，使得只要改变松弛实现，就可以使用程序 21.6 求解多源点最短路径问题、多源点最长路径问题以及其他问题。

21.58 使用练习 21.57 的通用实现来实现 ADT 操作，它们可返回从任何源点到 DAG 中任何其他顶点的最长路径长度、此路径的最短路径长度以及经由路径可达（长度落入一个给定范围内）的顶点数目的。

● **21.59** 定义松弛的性质，从而可以修改性质 21.9 的证明，以应用程序 21.6 的一个抽象版本。（类似练习 21.57 中所描述的版本）。

▷ **21.60** 按照图 21-16 的风格，显示使用程序 21.7 计算练习 21.54 中定义的网的所有对最短路径矩阵的结果。

○ **21.61** 给出程序 21.7 所访问的边权值数目的一个上界。并将其作为网的基本结构性质的一个函数。编写一个程序来计算此函数，对于各种类型的无环网（在第 19 章中的模型中适当增加权值），使用此函数估计 *VE* 界的精确度。

○ **21.62** 编写一个基于 DFS 的求解无环网的多源点最短路径问题的解决方案。你的解决方案能正确处理负边权值的情况吗？解释你的回答。

▷ **21.63** 扩展练习 21.62 的解决方案，为无环网提供所有对最短路径 ADT 接口的实现，并在与 *VE* 成正比的时间内构建所有路径数组和所有距离数组。

21.64 按照图 21-9 的风格，显示使用练习 21.63 的基于 DFS 的方法来计算练习 21.54 中所定义的网的所有对最短路径的结果。

▷ **21.65** 修改程序21.6，使其能够求解无环网中单源点最短路径问题，然后使用它来开发无环网的所有对最短路径ADT接口的一个实现，此接口在与 VE 成正比的时间内构建所有路径数组和所有距离数组。

○ **21.66** 对于所有对最短路径ADT的基于DFS（练习21.63）的实现和基于拓扑排序（练习21.65）的实现，处理练习21.61。对于这三种方法的开销比较，可得出什么结论？

21.67 进行实验研究，按照图20-2的风格，对于各种类型的无环网（在第19章中的模型中适当增加权值），对本节描述的所有对最短路径问题的3个程序（见程序21.7、练习21.63和练习21.65）进行比较。

21.5 欧几里得网

在用网来对地图建模的应用中，主要兴趣是找出从一个位置到另一个位置的最佳路线。在这一节里，我们考察此问题的一种策略：*欧几里得网*（Euclidean network）中源点－汇点最短路径问题的一种快速算法，这种网的顶点是平面上的点，边权值定义为这些点之间的几何距离。

这些网满足一般边权值不必满足的两个重要性质。第一，距离满足三角不等式：从 s 到 d 的距离不大于从 s 到 x 的距离加上从 x 到 d 的距离。第二，顶点位置给出了路径长度的一个下界。不存在从 s 到 d 的路径会比从 s 到 d 的距离更短。本节所考察的源点－汇点最短路径问题的算法就利用这两个性质来提高性能。

欧几里得网常常还是*对称的*（summetric）：在两个方向上都有边。在本章已开始提到过，如果将无向加权欧几里得图（见20.7节）的邻接矩阵或邻接表解释为一个加权有向图（网），那么就会出现这种网。在画出一个无向欧几里得网时，我们假设有此解释以避免绘图时出现过多箭头的混乱情况。

基本思路很简单：优先级优先搜索为我们提供了搜索图中路径的一种一般机制。使用Dijkstra算法，按照与起始顶点距离的远近顺序来检查路径。这一顺序可以保证在达到汇点时，就已经检查了图中所有的更短路径，其中无一能够达到汇点。但在欧几里得图中，还有另外的信息：如果正在查找一条从源点 s 到汇点 d 的路径，并且遇到第三个顶点 v，那么就知道不仅要取已经找到的从 s 到 v 的路径，而且要从 v 访问到 d，最好的做法是首先取边 v-w，然后找出一条长度为从 w 到 d 的直线距离的路径（见图21-18）。使用优先级优先搜索，可以很容易将这一额外的信息考虑进来，以提高算法的性能。我们使用标准算法，但使用以下三个量的和作为每条边 v-w 的优先级：从 s 到 v 的已知路径长度，边 v-w 的权值，以及从 w 到 t 的距离。如果总是选择此数最小的边，那么，当到达 t 时，仍然可以确保图中不存在从 s 到 t 的更短的路径。此外，在典型网中，要得到这个结论，相对于使用Dijkstra算法，我们所做的工作要少得多。

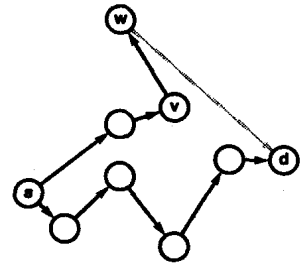


图21-18 边松弛（欧几里得）

在一个欧几里得图中，当计算最短距离时，可将到目的点的距离考虑在松弛操作中。这此例中，我们可以得出结论：从 s 到 v 的所示路径加上 v-w 不会导致比已经找到的从 s 到 d 的路径更短的路径，因为任何这种路径的长度至少为从 s 到 v 的路径长度，加上 v-w 的长度，加上从 w 到 d 的直线距离，这要比已知的从 s 到 d 的路径长度更长。这样检查可以大大减少必须考虑的路径数目。

为了实现这种方法，我们使用了Dijkstra算法的一个标准PFS实现（程序21.1，这是因为欧几里得图常常是稀疏的，也可参见练习21.73），其中有两个改变：第一，在搜索的开始不是将wt[s]初始化为0.0，而是将它设置为距离量dist(s, d)，其中dist是一个返回两个顶点间距离的函数。

第二，将优先级 P 定义为如下函数

```
(wt[v] + t->wt + dist(t->v, d) - dist(v, d))
```

而非程序 21.1 中所使用的函数（wt[v] + t->wt）。这些变化我们称之为欧几里得启发式搜索（Euclidean heuristic）维持了以下不变式，且对对每个顶点 v，量 wt[v] - dist(v, d) 是网中从 s 到 v 的最短路径长度（因而，wt[v] 也是从 s 到 d 且通过 v 的最短可能路径长度的下界）。我们将边权值（到 t->v 的距离）再加上从 t->v 到汇点 d 的距离增加到此量中来计算 wt[t->v]。

性质 21.11 *带有欧几里得启发式搜索的优先级优先搜索可以解决欧几里得图中源点 - 汇点最短路径问题。*

证明 应用性质 21.2 的证明：在将一个顶点 x 增加到树中时，从 x 到 d 的距离会增加到优先级中，这并不影响到以下结论，即从 s 到 x 的树路径是图中从 s 到 x 的一条最短路径，因为同一个量被增加到所有到 x 的路径上。当 d 增加到树中时，我们知道不存在其他从 s 到 d 的路径会比树路径更短，因为任何这样的路径必定包含一条树路径后跟着到达不在此树中的某个顶点 w 的一条边，再跟着一条从 w 到 d 的路径（其长度不会比从 w 到 d 的距离更短）；而且，由构造过程，我们知道从 s 到 w 的路径长度加上从 w 到 d 的距离不会小于从 s 到 d 的树路径的长度。 ■

在 21.6 节中，我们讨论实现欧几里得启发式搜索的另一种简单方法。第一，对图进行一边扫描以改变每条边的权值：对于每条边 v-w，增加量 dist(w, d) - dist(v, d)。然后，运行标准最短路径算法，在 s 开始（同时将 wt[s] 初始化为 dist(s, d)）并在到达 d 时终止。这种方法在计算上等价于我们已经描述的方法（基本上动态地计算同样的权值），而且是一个基本操作的特例，此操作称为对网进行*重新加权*（reweighting）。重新加权在解决有负权值的最短路径问题中起着关键的作用；我们将在 21.6 节详细讨论这种方法。

欧几里得启发式搜索会影响到 Dijkstra 算法对于源点 - 汇点最短路径计算的性能，但对其正确性没有影响。如同在性质 21.2 中所讨论的，使用标准算法来解决源点 - 汇点问题相当于构建一个 SPT，其中包含了较之汇点 d 距起始点更近的所有顶点。利用欧几里得启发式搜索 SPT 只包含某些顶点，从 s 到这些顶点的路径加上到 d 的距离小于从 s 到 d 的最短路径的长度。对于很多应用，我们期望这棵树非常小，因为启发式搜索会剪掉大量的长路径。具体的节省依赖于图的结构和顶点的几何位置。图 21-19 显示了欧几里得启发式搜索在示例图上的操作过程，其中的节省是巨大的。将这种方法称为

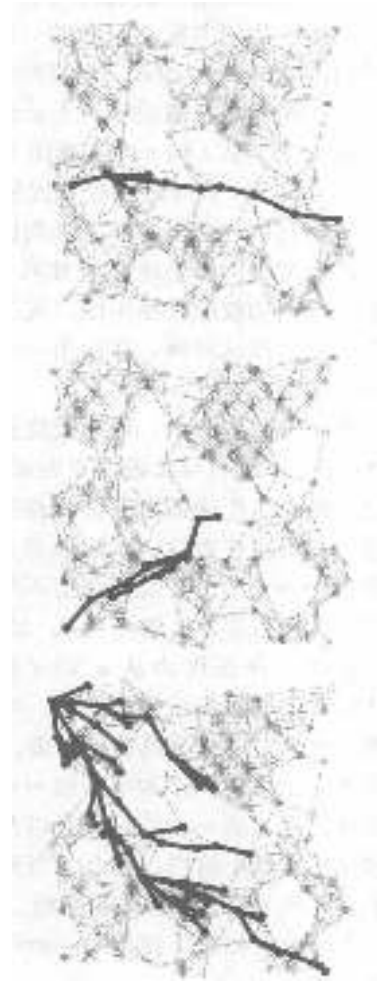


图 21-19 欧几里得图中的最短路径

当我们朝着目的顶点搜索最短路径时，可将搜索限制到路径周围的一个相对小的椭圆范围内的顶点，如以上 3 个例子所示，在此显示了图 21-12 中示例的 SPT 子树。

启发式搜索，是因为此方法不能保证一定会带来节省：也可能会有这种情况，从源点到汇点的唯一路径是一条长的路径，在回到汇点之前，在距源点很远的地方漫游（见练习21.80）。

图21-20说明了欧几里得启发式搜索所描述的直观性的基本几何性质：如果从 s 到 d 的最短路径长度为 z，那么算法所检查的顶点大致落入点 x 的位置所定义的椭圆内，其中从 s 到 x 的距离再加上从 x 到 d 的距离等于 z。对于典型的欧几里得图，可以预期此椭圆内的顶点数远小于以源点为中心的半径为 z 的圆内的顶点数（而这些顶点是Dijkstra算法所要检查的）。

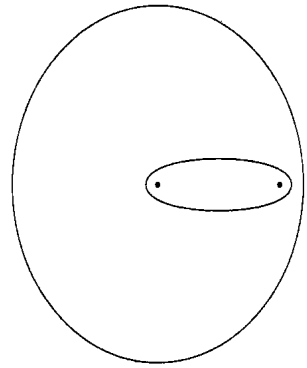


图21-20 欧几里得启发式搜索的开销界限

当我们朝着目的顶点搜索最短路径时，可将搜索限制到路径周围的一个相对小的椭圆范围内的顶点。并与Dijkstra算法所要求的以s为中心的圆进行比较。此圆的半径以及椭圆的形状由最短路径的长度决定。

对于节省量的精确分析是一件困难的分析问题，而且既取决于随机顶点集的模型，还取决于随机图的模型。对于典型的情况，如果标准算法在计算一条源点－汇点最短路径时检查了 X 个顶点，那么可以期望欧几里得启发式搜索会将开销降至与 $\sqrt{X}$ 成正比，这样对于稠密图，就会得到与 V 成正比的期望运行时间，而对于稀疏图，期望运行时间则与 $\sqrt{V}$ 成正比。这个示例说明了，开发一个合适的模型或者分析相关算法的难度不应阻碍我们利用在很多应用中可用的大量节省，特别是在实现很简单的情况下（只需要为优先级增加一项）更可充分利用。

如果函数可给出从每个顶点到 d 的距离的一个下界，性质21.11的证明就适用于任何（any）这样的函数。是否可能存在其他函数也能使算法所检查的顶点数比欧几里得启发式搜索更少呢？这个问题已经在一般环境中作了研究，可适用于大量的组合搜索算法。实际上，欧几里得启发式搜索是一种称为A*（读作“ay-star”）算法的特例。此理论说明，使用最佳可用的下界函数是最优的（optimal）；换一种说法，界限函数越好，搜索就越高效。在这种情况下，由A*的最优性可知，较之于Dijkstra算法（为下界为0的A*算法）所检查的顶点，欧几里得启发式搜索所检查的点不会更多。上述分析结果对于特定的随机网模型给出了更为准确的信息。

我们还可以利用欧几里得网的性质帮助构建抽象最短路径ADT的高效实现，相对于一般网可以在更有效地对时间和空间加以权衡（见练习21.48～21.50）。这些算法在诸如地图处理等应用中非常重要，在这些应用中，网常常都是巨型而稀疏的。例如，对于一个有数百万条道路的地图，假设希望开发一个基于最短路径的导航系统。我们也许可以将地图本身存储在一个小的机载计算机上，然而，距离矩阵和路径矩阵会很大，以至于无法保存（见练习21.39和练习21.40）；因此，21.3节的所有对最短路径算法就是无效的。Dijkstra算法对于大型地图也可能无法提供足够短的响应时间。练习21.77～21.78探索了有关策略，可以投入一些合理的预处理时间和空间，从而为源点－汇点最短路径查询提供快速响应。

练习

- **21.68** 找出一个在线大型欧几里得图，可以是一个包含位置表以及它们之间距离的地图，也可以是有成本的电话连接，或航班线路和费用。

21.69 使用练习17.13～17.15中所描述的策略，编写一个将按照 $\sqrt{V} \times \sqrt{V}$ 排列的顶点连接

起来，产生随机欧几里得图的程序。

▷ **21.70** 显示欧几里得启发式搜索所计算的部分 SPT 与初始化 wt[s] 的值无关。解释如何由初始值计算最短路径长度。

▷ **21.71** 按照图 21-10 的风格，显示使用欧几里得启发式搜索来计算练习 21.1 中所定义的网中从 0 到 6 的一条最短路径。

○ **21.72** 对于每对顶点，如果在欧几里得启发式搜索中所使用的函数 dist（s，t）返回从 s 到 t 的实际最短路径长度，描述会发生什么情况？

21.73 基于邻接矩阵网 ADT 和 Dijkstra 算法实现（程序 20.3，及一个合适的优先级函数），为稠密欧几里得图中的最短路径开发一个最短路径 ADT 实现。

21.74 进行实验研究，对于各种类型的欧几里得网（见练习 21.8、21.68、21.69 和 21.80），测试欧几里得启发式搜索的有效性。对于每个图，生成 $V/10$ 对随机顶点，并打印一个表，显示顶点之间的平均距离、顶点之间最短路径的平均长度、欧几里得启发式算法所检查的顶点数与 Dijkstra 算法所检查的顶点数的平均比率，以及与欧几里得启发式搜索相关的椭圆区域与 Dijkstra 算法相关的圆形区域的平均比率。

21.75 基于练习 21.35 中所描述的双向搜索，开发欧几里得图的源点 - 汇点最短路径问题的一种实现。

○ **21.76** 针对源点 - 汇点问题，对于 Dijkstra 算法所产生的 SPT 中的顶点数与练习 21.75 中所描述的两路版本所产生 SPT 中的顶点数，使用几何解释估计它们之间的比率。

21.77 开发欧几里得图的最短路径的一个 ADT 实现，执行如下预处理步：将图的区域分成 $W \times W$ 的网格，然后使用 Floyd 所有对最短路径算法来计算 $W^2 \times W^2$ 数组，其中 i 行、j 列包含连接方格 i 中任意顶点和方格 j 中任意顶点的最短路径的长度。然后，使用这些最短路径长度作为下界来改进欧几里得启发式搜索。用不同的值 W 来进行实验，使得每个方格中的期望顶点数为一个小的常量。

21.78 组合练习 21.57 和练习 21.77 中的思想，开发欧几里得图的所有对最短路径的一个实现。

21.79 进行实验研究，对于各种欧几里得网（见练习 21.8、21.68、21.69 和 21.80），比较练习 21.75 ~ 21.78 中所描述的启发式搜索的有效性。

21.80 扩展你的实验，使其包含如下欧几里得图：该图由删除位于中心、半径为 r 的圆中的所有顶点和边而得到，$r = 0.1$、0.2 和 0.3。（这些图为欧几里得启发式搜索提供了严格测试。）

21.81 对于平面上的 N 个点以及一些边（将相互距离为 d 之内的相连接）所定义的隐式欧几里得图，对其网 ADT 实现给出一个直接的 Floyd 算法实现。不要显式地表示图；对于给定的两个顶点，计算它们的距离来确定一条边是否存在，如果边存在，其长度是多少。

21.82 对于练习 21.81 中所描述的情况开发一个实现，构建近邻图，然后对于每个顶点使用 Dijkstra 算法（见程序 21.1）。

21.83 进行实验研究，比较练习 21.81 和 21.82 中的算法所需的时间和空间，$d = 0.1$、0.2、0.3 和 0.4。

● **21.84** 编写一个客户程序，动态地演示欧几里得启发式搜索的过程。你的程序应该产生类似图 21-19 那样的图像（见练习 21.38）。对于各种欧几里得网（见练习 21.8、21.68、21.69 和 21.80）测试你的程序。

21.6 归约

由此可以得出最短路径问题（尤其是允许负权值的一般情况（21.7节的主题））代表了用于求解各种其他问题的通用数学模型，这些问题似乎与图处理不相关。这种模型是我们遇到的几个通用模型的第一个模型。当转到更为困难的问题和更为通用的模型时，面临的挑战之一就是准确地刻画各种问题之间关系的特征。给定一个新的问题，我们会问：是否能够将它转换为一个已知如何求解的问题来容易地求解此问题？如果对问题加以限制，是否能够更容易地求解此问题呢？为了帮助回答这些问题，本节先讨论用于描述这些问题之间类型关系得一些技术语言。

定义21.3 *对于两个问题A和B，如果使用求解B的一个算法来开发一个求解A的算法，且最坏情况下该算法的总时间不会超过最坏情况下求解B的算法的运行时间的常量倍，则称问题A可归约（reduce）为问题B。如果两个问题可相互归约，则称这两个问题是等价的（equivalent）。*

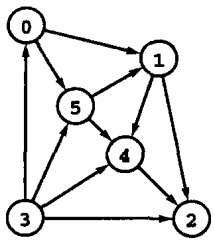
对于“使用”一个算法来“开发”另一个算法的含义，我们将在第8部分给出它的严格定义。对于大多数的应用，我们满足于以下简单方法。只要能采用以下3步来求解A的任何实例，就表明A可归约为B。

- 将它转换为B的一个实例。
- 求解B的那个实例。
- 将B的解决方案转换为A的解决方案。

只要能够有效地执行转换（并求解B），就能有效地求解A。为了说明这种证明技术，考虑两个例子。

性质21.12 *传递闭包问题可归约为有非负权值的所有对最短路径问题。*

证明 我们已经指出了Warshall算法和Floyd算法之间的直接关系。在目前的上下文中，考虑这种关系的另一种方式是，想象需要使用计算网中所有最短路径的库函数来计算有向图的传递闭包。为了做到这一点，如果有向图中没有自环，则增加自环；然后由有向图的邻接矩阵直接构建一个网，对应1的指定任意一个权值（比如说0.1），对应0的指定为观察哨权值。然后，调用所有对最短路径函数。接下来，由此函数计算出的所有对最短路径矩阵很容易计算出传递闭包：给定两个顶点u和v，有向图中从u到v存在一条路径，当且仅当网中从u到v的路径长度非零（见图21-21）。 ■



	0	1	2	3	4	5
0	.10	.10				.10
1		.10	.10		.10	
2			.10			
3	.10		.10	.10	.10	.10
4			.10		.10	
5		.10			.10	.10

	0	1	2	3	4	5
0	.10	.10	.20		.20	.10
1		.10	.10		.10	
2			.10			
3	.10	.20	.10	.10	.10	.10
4			.10		.10	
5		.10	.20		.10	.10

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

图21-21 传递闭包归约

给定一个有向图（左图），可以通过对每条边指定一个任意权值（左边矩阵），将其邻接矩阵（带有自环）转换为表示网的邻接矩阵。通常，矩阵中的空元素表示一个观察哨值，指示一条边不存在。给定此网的所有对最短路径长度矩阵（中间矩阵），有向图的传递闭包（右边矩阵）就是由将观察哨值代之以0并将其他元素代之以1所构成的矩阵。

此性质是对传递闭包问题不会比所有对最短路径问题更困难的一个形式化的论述。对于我们熟知的所有对最短路径问题的算法，因为已经知道传递闭包算法速度更快，因此这个信息并不令人惊讶。当使用归约建立尚不知道如何求解的问题之间的关系，或者建立这些问题与其他可以求解的问题之间的关系时，归约是很有意义的。

性质 21.13 *在边权值没有限制的网中，（单源点或所有对）最长路径和最短路径问题是等价的。*

证明 给定一个最短路径问题，将所有权值取负。修改所得的网中的一条最长路径（具有最大权值）是原网中的一条最短路径。同理可证最短路径问题可归约到最长路径问题。 ■

此证明很简单，但此性质还说明了表述和证明归约时需要仔细，因为很容易理所当然地归约，从而被误导。例如，如果认为最长路径问题和最短路径问题在非负权值的网中是等价的，肯定不正确。

本章一开始，我们给出了一个论点，即找出无向加权图中最短路径可归约为找出网中的最短路径的问题，因此可以使用对于网的算法来解决无向加权图中的最短路径问题。在目前的环境下，关于此归约，还有两点要考虑。第一，其逆命题不成立：了解如何求解无向加权图中的最短路径问题并不能帮助我们求解网中的最短路径问题。第二，在论点中看到存在一个问题：如果边权值可以为负，那么归约会得到带有负环的网，而且我们并不知道如何求解这种网中的最短路径。即使是不能归约，也可以说明使用一种异常复杂的算法，仍然可能找出不带负环的无向加权图中的最短路径（见第 5 部分参考文献）。因为这个问题并没有归约为有向版本，此算法不能帮助我们求解一般网中的最短路径问题。

归约的概念基本上描述了使用一个 ADT 实现另一个 ADT 的过程，就像现代系统程序员例行的工作。如果两个问题是等价的，我们知道，如果能够高效地解决其中一个问题，就可以高效地解决另一个问题。我们常常找出简单的一一对应关系，比如性质 21.13，它显示了两个问题是等价的。在这种情况下，我们还没有讨论如何求解其中任何一个问题，但如果找出其中之一的问题的高效解决方案，就能使用这个解决方案来求解另一个问题。我们在第 17 章看过另一个问题：当面对确定一个图是否存在奇环的问题时，我们指出道该问题等价于确定该图是否是 2-可着色的。

归约在算法的设计和分析中有两个主要应用。首先，它可以帮助我们按照问题的难度在一个适当的抽象级上进行分类，而不比开发和分析完整的实现。其次，我们常常进行归约从而确定求解各种问题难度的下界，帮助指明何时停止搜索更好的算法。在 19.3 和 20.7 中我们已经看到这些应用的例子；本节后面会看到其他例子。

除了这些直接的实际应用，归约的概念还对计算理论有着广泛而深远的影响；在处理越发困难的问题时，其含义对于我们理解问题相当重要。本节最后将简要讨论这一内容，并且将在第 8 部分全面而详细的讨论。

转换的开销不应该占主导地位，这个限制很自然，也常常适用。然而，在很多情况下，即使转换的开销起着主导作用，也可能选择使用归约。归约的最重要的应用之一是为似乎用其他方法难处理的问题提供高效的解决方案，将问题转换为一个我们已知如何求解的易于理解的问题。将 A 转换为 B，即使假如计算此转换要比求解 B 代价要高得多，较之于可能设计的其他方法，也可能给出一个求解 A 的高效得多的算法。还有很多其他的可能性。也许我们关注期望开销，而不是最坏情况下的开销。也许我们需要解决两个问题 B 和 C，才能解决 A。也许我们需要解决问题 B 的多个实例。我们将这些问题留到第 8 部分再讨论，因为此前所考虑的所有例子都属于以上讨论的简单类型。

在特定环境下，通过将问题 A 简化为另一个问题 B 来解决问题 A，则称 A 归约为 B，但反之不然。例如，选择可归约为排序，是因为先对文件进行排序，然后索引（或扫描）第 k 个位置，就可找出一个文件中的第 k 个最小元素，但这一点并不蕴含着排序可归约为选择。在当前的上下文中，加权 DAG 的最短路径问题和带有正权值网的最短路径问题均可归约为一般最短路径问题。归约的这一用法对应为一种直观的说法，即一个问题比另一个问题更一般。任何排序算法都可以解决任何选择问题，而且如果可以解决一般网中的最短路径问题，则肯定可以将这种解决方案用于各种限制的网中；但是反之则不一定成立。

归约的这种用法很有用，但是用归约来得到有关不同领域问题之间关系的信息则更显有用。例如，考虑以下问题，这些问题乍眼一看与图处理问题不相关。但是通过归约，可以建立这些问题与最短路径问题之间的特定关系。

作业调度 有大量持续时间不同的作业需要完成。在某一给定时间可以处理任意多个作业，但是为一组作业对指定一组优先关系，要求第一个作业必须在第二个作业开始之前完成。在满足所有优先约束的条件下，完成所有作业所需的最短时间是多少？具体地说，给定一组作业（带有持续时间），以及一组优先约束，调度作业（为每个作业找出一个起始时间）使得完成时间最短。

图 21-22 描述了作业调度问题的一个示例。它使用一种自然网络表示，稍后我们将这个表示用作为归约的基础。此问题的这一版本也许是文献中已得到研究的数百个版本中的最简单的一个，其他版本会涉及作业的其他特征和其他约束，比如对作业的人员或资源的指定，与特定作业关联的其他开销，截止期等等。在这种情况下，我们所描述的这一版本通常称为*带有无限并行性的优先约束调度*（precedence-constrained scheduling with unlimited parallelism）；我们使用术语*作业调度*（job scheduling）作为简称。

为了帮助开发一个解决作业调度问题的算法，我们考虑如下问题，其本身也有着广泛的应用。

差分约束 为一组变量 $x_0 \ldots x_n$ 赋予非负值，从而使 x_n 的值达到最小，同时满足对一组变量的*差分约束*（difference constraint）。每个约束指定了两个变量的差必定大于或等于一个给定的常量。

图 21-23 描绘了此问题的一个示例。这是一个纯粹的抽象数学公式，可以用作求解很多实际数值问题的基础（见第 5 部分参考文献）。

差分约束问题是更一般问题的一个特例，其中允许在等式中出现变量的一般线性组合。

线性规划 为一组变量 $x_0 \ldots x_n$ 赋予非负值，从而使变量的特定线性组合的值达到最小，满足对变量的一组约束，每个约束指定了变量的某个线性组合必定大于或等于一个给定的常量。

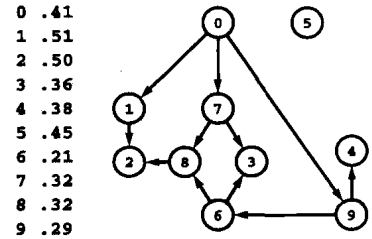


图 21-22 作业调度

在这个网中，顶点表示要完成的作业（权值指示所需时间），边表示作业之间的优先关系。例如，从 7 到 8 和 3 的边的含义是作业 7 必须在作业 8 和作业 3 开始之前完成。完成所有作业所需的最小时间是多少？

线性规划是一种广泛使用的一般方法，可用于求解一大类的组合问题。我们将在第 8 部分详细讨论这些问题。显然，差分 - 约束问题可归约为线性规划，像其他很多问题的做法。就目前而言，我们关注的是差分约束、作业调度和最短路径问题之间的关系。

性质 21.14 作业调度问题可归约为差分约束问题。

证明 增加一个虚拟作业，并为每个作业增加一个优先约束，表示此作业必须在虚拟作

业开始之前结束。给定一个作业调度问题，定义一组差分方程，其中每个作业 i 对应一个变量 x_i，要求 j 在 i 完成之前不能开始的约束对应为方程 $x_j \geq x_i + c_i$，其中 c_i 为作业 i 的长度。差分约束问题的解决方案恰好给出了作业调度问题的一个解决方案，每个变量的值指定了所对应作业的起始时间。 ■

图 21-23 说明了由对图 21-22 中的作业调度问题完成此归约所建立的差分方程组。此归约的实际意义是，可以使用解决差分约束问题的任何算法来解决作业调度问题。

$$
\begin{aligned}
x_1 - x_0 &\geq .41\\
x_7 - x_0 &\geq .41\\
x_9 - x_0 &\geq .41\\
x_2 - x_1 &\geq .51\\
x_8 - x_6 &\geq .21\\
x_3 - x_6 &\geq .21\\
x_8 - x_7 &\geq .32\\
x_3 - x_7 &\geq .32\\
x_2 - x_8 &\geq .32\\
x_4 - x_9 &\geq .29\\
x_6 - x_9 &\geq .29\\
x_{10} - x_2 &\geq .50\\
x_{10} - x_3 &\geq .36\\
x_{10} - x_4 &\geq .38\\
x_{10} - x_5 &\geq .45
\end{aligned}
$$

图 21-23 差分约束

找出对变量的一组非负值，从而使 x_{10} 的值达到最小，并满足这组不等式等价于图 21-22 中所描述的作业调度问题的实例。例如，方程 $x_8 \geq x_7 + 0.32$ 的含义是作业 8 在作业 7 完成之前不能开始。

考虑是否可在相反的方向使用这个构造很有益处：给定一个作业调度问题，我们可以使用它解决差分约束问题吗？对此问题的回答是：性质 21.14 的证明中的对应性并不能表明差分约束问题可归约为作业调度问题，因为由作业调度问题所得的差分方程组中有一个性质并没有必要在每一个差分约束问题中也成立。具体地说，如果两个方程的第二个变量相同，那么它们有相同的常数。因此，作业调度问题的算法不能直接给出包含两个方程 $x_i - x_j \geq a$ 和 $x_k - x_j \geq b$ 的差分方程组的求解方法，其中 $a \neq b$。在证明归约性时，需要注意到如下情况：证明 A 可归约为 B，必须证明可以使用求解 B 的算法来求解 A 的任何实例。

由构造过程，性质 21.14 的证明的构造所产生的差分约束问题的常数总是非负的。这一点显然很重要。

性质 21.15 *有正常数的差分约束问题等价于无环网中的单源点最长路径问题。*

证明 给定一组差分方程，构建一个网，其中每个变量 x_i 对应一个顶点 i，每个方程对应一条权值为 c 的边 i-j。例如，对图 21-22 的有向图的每条边赋以其源顶点的权值，就得到对应于图 21-23 中的差分方程组的网。在网中增加一个虚拟顶点，且该虚拟顶点到其他每个顶点的边上的权值为 0。如果此网中存在环，那么差分方程组无解（因为正权值蕴含着，当沿着路径行进时，与每个顶点对应的变量值严格递减，因此环意味着某个变量比自身更小），这是报告这种情况。否则，如果网中不存在环，因此问题就是从此虚拟顶点求解单源点最长路径问题。因为网是无环的（见 21.4 节），网中存在从每个顶点开始的一条最长路径。对每个变量赋以网中从虚拟顶点开始到对应顶点的最长路径的长度。对于每个变量，这条路径表明其值满足约束且不存在更小的值能做到这一点。

不同于性质 21.14 中的证明，这个证明可以扩展为证明这两个问题是等价的，因为构造在两个方向都成立。我们并不限制有相同第二个变量的两个方程必定有相同常数，而且也不限制离开网中任何给定顶点的边必定有相同权值。给定带有正权值的任一无环网络，同样的对应性可以给出一组有正常数的差分方程，其解决方案直接得到网中单源点最长路径问题的解决方案。对此的详细证明留作练习（见练习 21.90）。 ■

图 21-22 的网描述了示例问题的对应关系，而图 21-15 显示了使用程序 21.6（虚拟顶点蕴含在实现中）对此网中最长路径的计算。用这种方法所计算的调度如图 21-24 所示。

程序 21.8 是显示此理论在实际环境中应用的一个实现。它将作业调度问题的任意实例转换为无环网中最长路径问题的一个实例，然后是用程序 21.6 进行求解。

程序21.8 作业调度

此实现从标准输入读入带有长度的作业列表，后跟一组优先约束，然后在标准输出上打印满足此优先约束的一组作业起始时间。通过使用性质21.14和性质21.15以及程序21.6，将其归约为无环网的最长路径问题来求解作业调度问题。稍做调整以适应此接口（例如，这里并没有使用st数组），这是使用一个已存在实现来实现一个归约的典型手段。

```
#include <stdio.h>
#include "GRAPH.h"
#define Nmax 1000
main(int argc, char *argv[])
  { int i, s, t, N = atoi(argv[1]);
    double length[Nmax], start[Nmax];
    int st[Nmax];
    Graph G = GRAPHinit(N);
    for (i = 0; i < N; i++)
      scanf("%lf", &length[i]);
    while (scanf("%d %d", &s, &t) != EOF)
      GRAPHinsertE(G, EDGE(s, t, length[s]));
    GRAPHlpt(G, 0, st, start);
    for (i = 0; i < N; i++)
      printf("%3d %6.2f\n", i, start[i]);
  }
```

我们隐含地假设了对于作业调度问题的任何实例都存在一个解决方案；然而，如果在一组优先约束中存在一个环，那么就不存在满足这些约束的作业调度。在查找最长路径之前，应当通过检查相应网中是否存在环来检查这个条件（见练习21.100），这种情况很典型，而且通常用一个特定的技术术语来描述。

定义21.4 *称一个没有解决方案的问题实例是不可行的*（infeasible）。

换句话说，对于作业调度问题，要确定一个作业调度问题实例是否是可行的，这个问题可以归约为确定一个有向图是否是无环的问题。在转向更为复杂的问题时，可行性问题成为在计算负荷上越来越重要（而且也越来越困难!）的一部分。

现在我们已经考虑了三个相关的问题。可能已经直接说明了作业调度问题可以归约为无环网中的单源点最长路径问题，我们还说明了可以用类似的方法（见练习21.94）解决任何差分约束问题（带有正常数），这种方法同样可用于解决可以归约为差分约束问题或作业调度问题的任何其他问题。我们还可以开发一个求解差分约束问题的算法，并用此算法来解决其他问题，但是，我们并未证实作业调度问题的解决方案可以给出其他问题的解决方法。

```
 0   0
 1   .41
 2  1.23
 3   .91
 4   .70
 5   0
 6   .70
 7   .41
 8   .91
 9   .41
10  1.73
```

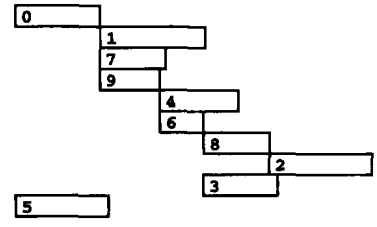


图21-24 作业调度

此图描述了图21-22中的作业调度问题的解决方案，由加权DAG中的最长路径和作业调度之间的对应性得到。wt数组中的最长路径长度是用程序21.6的最长路径算法计算而得（见图21-15），恰好为作业所需的起始时间（上图，右列）。在0时刻开始作业0和作业5，在时刻0.41开始作业1、7和9，在时刻0.70开始作业4和6，依此继续。

这些例子说明了归约的使用可以拓宽已证明实现的应用性。实际上，现代系统编程强调通过开发新的接口并使用现有软件资源来构建实现，从而满足软件重用的需求。这个重要的过程有时称为*库编程*（library programming），是归约思想的一种实际实现。

库编程在实际中极其重要，但它只体现了归约含义中的一部分。为了说明这一点，我们考虑作业调度问题的以下版本。

具有截止期的作业调度 在作业调度问题中允许增加一类约束，来指定一个作业必定相对于另一个作业在扫过一个特定的时间量之前开始。（常规截止期是相对于起始作业。）在时间很关键的生产过程以及很多其他应用中这样的约束常常是必须的，而且它们使得作业调度问题解决起来要困难得多。

假设需要为图 21-22 ~ 21-24 中的例子增加一个约束，作业 2 必须在作业 4 开始之后的 c 个时间单位之前开始。如果 c 大于 0.53，那么所计算的调度合乎要求，因为它说明作业 2 在时刻 1.23 开始，为作业 4（在时刻 0.70 开始）结束时间后的 0.53。如果 c 小于 0.53，我们可以将作业 4 的开始时间往后移以满足约束。如果作业 4 是一个长作业，这个调整可能会增加整个调度的完成时间。更糟的是，如果对作业 4 还有其他存在约束，那么可能就不能调整其开始时间。实际上，我们可能找不出满足约束的调度：例如，作业 2 必须在作业 6 开始后的 d 个时间单位之内开始，这里 d 小于 0.53，因为作业 2 必须在作业 8 之后，作业 8 必须在作业 6 之后，蕴含着作业 2 必须在作业 6 开始 0.53 时间后才能开始。

如果在示例中增加上一段所描述的两个约束，那么这两个约束影响作业 4 调度的时间，整个调度的完成时间，还会影响到是否存在一个可行调度，都依赖于 c 和 d 的值。增加更多这种类似的约束带来更多的可能性，并将一个简单的问题变成一个困难的问题。因此，我们寻求将此问题归约为一个已知问题的方法是合理的。

性质 21.16 *带有截止期的作业调度问题可归约为（允许带有负权值的）最短路径问题。*

证明 采用与性质 21.14 中所描述的同样的归约将优先约束转换为不等式。对于任何截止期约束，增加一个不等式 $x_i - x_j \leqslant d_j$，或等价不等式 $x_j - x_i \geqslant -d_j$，其中 d_j 是一个正常数。使用性质 21.15 中所描述的同样的归约，将不等式组转换为一个网。将所有权值取负。由性质 21.15 的证明中给出的相同构造，可得网中以 0 为根的任何最短路径树都对应一个调度。 ■

此归约带我们进入了带有负权值的最短路径领域。它说明如果我们可以找出带有负权值的最短路径问题的一个高效解决方案，那么就能找出有截止期的作业调度问题的一个高效解决方案。（同样，性质 21.16 证明中的对应性并不能确定相反方向的转换（见练习 21.91）。）

为作业调度问题增加截止期对应于在差分约束问题中允许有负常数，以及在最短路径问题中允许有负权值。（这个变化还要求我们合适地修改差分约束问题，以处理最短路径问题中类似负环的情况。）较之开始所考虑的那些版本，这些问题的更一般的版本更难求解，但是它们很可能作为更一般的模型更为有用。求解所有这些问题的一种不确定的方法似乎是寻求一种求解带有负权值的最短路径问题的高效方法。

遗憾的是，采用这种方法存在着本质上的难度，而且还显示出使用归约的另一个方面，评估问题的相对难度。我们一直在正面意义上使用归约，扩展解决方案的可应用性以适用一般问题。但它也可应用在负面的情况来表明这些扩展的局限性。

一般最短路径问题的求解的困难相当大，以至于无法解决。接下来将会看到归约的概念是如何帮助我们确信这一点。在 17.8 节中，我们讨论了一组称为 NP-难的问题，并认为这些问题是难解的，因为对于它们的所有已知算法最坏情况下都需要指数时间。这里我们表明

一般的最短路径问题是 NP-难的。

正如在 17.8 节中提到的及在第 8 部分详细讨论的，对于某个问题为 NP-难问题这一事实，我们通常会认为不仅没有可确保解决该问题的已知高效算法，而且对于找出这样算法的希望也很小。在这种情况下，我们使用术语高效（efficient）来表明一个算法在最坏情况下的运行时间由输入规模的某个多项式函数所限定。假设任何 NP-难问题的高效算法的发现是一个令人震惊的突破性研究。NP-难度的概念在识别难以求解的问题时非常重要，因为使用以下技术，往往易于证明一个问题是 NP-难问题。

性质 21.17 *对于某个问题，如果存在一种高效归约能将任何 NP-难问题归约为此问题，则称此问题是 NP-难的（NP-hard）。*

证明 此性质取决于从一个问题 A 高效归约为另一个问题 B 的准确含义。我们将这些定义放到第 8 部分再做说明（通常使用两种不同的定义）。目前而言，我们只是使用此术语来说明以下情况，即将 A 的实例转换为 B 的实例，以及将 B 的一个解决方案转换为 A 的一个解决方案的高效算法。

现在，假设存在一个高效归约将一个 NP-难问题 A 转换为一个给定的问题 B。以下用反证法证明。如果有一个求解 B 的高效算法，那么可以使用此算法来在多项式的时间内求解 A 的任何实例，由归约可得（将 A 的一个给定实例转换为 B 的一个实例，并求解这个问题，然后再对此解决方案进行转换）。但是不存在已知算法可以对 A 做出此保证（因为 A 是 NP-难的），因此假设存在 B 的一个多项式算法是不正确的：B 也是 NP-难的。 ■

这项技术极为重要，因为人们一直用此技术来说明大量的问题都是 NP-难问题时，从而提供了大量的问题，当我们试图开发一个证明以表明新问题是 NP-难问题时，可在这些问题中进行选择。例如，我们在 17.7 节遇到了一个经典的 NP-难问题。即哈密顿回路问题，它询问是否存在包含给定图中所有顶点的一条简单路径，这是最早被证明是 NP-难的一个问题（见第 5 部分参考文献）。很容易将它形式化为一个最短路径问题，因此性质 21.17 蕴含着最短路径问题自身是 NP-难的。

性质 21.18 *在带有边权值（可能为负）的网中，最短路径问题是 NP-难问题。*

证明 我们的证明包含了将哈密顿回路问题归约为最短路径问题。也就是说，我们表明，对于带有负权值边的网，可以使用找出其最短路径的任何算法来求解哈密顿问题。给定一个无向图，构建一个网，其中在无向图中的每条边在此网中都有两条边，且所有边的权值为 -1。开始于此网中的任何顶点的最短（简单）路径长度为 $-(V-1)$，当且仅当此图存在一个哈密顿回路。注意到此网中充满了负环。不仅图中每个环对应网中的一个负环，而且图中的每条边（edge）对应网中权值为 -2 的一个环。 ■

此构造蕴含着最短路径问题是 NP 难的，因为如果开发网的最短路径问题的一种有效算法，那么，就可以开发图的哈密顿问题的一种有效算法。

发现一个给定的问题是 NP-难时，一种反应就是去寻找能够求解（can solve）此问题的其他一些版本。对于最短路径问题，我们注意到对于无环网或边权值非负的网有大量高效的算法，而对于存在环和负权值的网则没有好的解决方案。是否还有其他类型的网可以得到解决呢？这是 21.7 节的主题。例如，我们将会看到具有截止期的作业调度问题可归约为可以有效求解的最短路径问题的一个版本。这种情况是典型的：随着更多困难的计算问题的解决，就会发现我们自己可以识别出能够解决的那些问题的某些版本。

正如这些例子所示，归约是一种有助于算法设计的简单技术，我们会频繁地使用这种技术。对于一个新问题，要么通过证明它可以归约为一个已知如何求解的问题，从而加以解决，

要么通过证明一个已知为困难的问题可以归约为当前问题，以此证明这个新问题是困难的。

表 21-3 归约的含义

此表使用本节已经讨论过的例子，概述了将问题 A 归约为另一个问题 B 的含义。情况 9 和 10 所蕴含的含义是如此的深奥，以至于我们一般假设不可能证明这样的归约（见第 8 部分）。归约在情况 1、6、11 和 16 是最有用的，说明可以学习 A 的一种新算法，或证明 B 的一个下界；情况 13-15 是学习 A 的新算法；情况 12 是学习 B 的难度。

	A	B	$A \Rightarrow B$ 蕴含	示例
1	简单	简单	B 的新下界	排序⇒EMST
2	简单	易解	无	TC⇒APSP（+）
3	简单	难解	无	SSSP（DAG）⇒SSSP（±）
4	简单	未知	无	
5	易解	简单	A 简单	
6	易解	易解	A 的新解决方案	DC（+）⇒SSSP（DAG）
7	易解	难解	无	
8	易解	未知	无	
9	难解	简单	深奥	
10	难解	易解	深奥	
11	难解	难解	同 1 或 6	SSLP（±）⇒SSSP（±）
12	难解	未知	B 难解	HP⇒SSSP（±）
13	未知	简单	A 简单	JS⇒SSSP（DAG）
14	未知	易解	A 易解	
15	未知	难解	A 可解	
16	未知	未知	同 1 或 6	JSWD⇒SSSP（±）

说明：EMST 欧几里得最小生成树
TC 传递闭包
APSP 所有对最短路径
SSSP 单源点最短路径
SSLP 单源点最长路径
（+）（带有非负权值的网）
（±）（权值可能为负的网）
（DAG）（无环网）
DC 差分约束
HP 哈密顿回路
JS（WD） 作业调度（带有截止期）

对于第 17 章中所讨论的 4 种一般问题，表 21-3 提供了归约在其中有不同含义的更为详细的内容。注意，在一些情况下，归约并不提供任何新信息；例如，尽管选择问题可归约为排序问题，并且找出无环网中最长路径问题的问题可归约为找出一般网中最短路径的问题，但这些事实并未给出问题的相对难度的任何明示。在其他情况下，归约可能会或也可能不会提供新的信息；另外还有一些情况，归约的含义则实在是深奥。为了建立这样的概念，我们需要对归约做出准确而正式的描述，对此将在第 8 部分详细讨论；这里，我们对已经见过的例子，在实际中归约的最重要用法做出非形式的概述。

上界（upper bound） 如果我们有一个求解问题 B 的高效算法，而且可以证明 A 可归约为 B，那么就有一个求解 A 的高效算法。可能存在求解 A 的其他更好的算法，但是 B 的性能

是求解 A 的最佳性能的一个上界。例如，证明作业调度问题可归约为无环网中最长路径问题，使对后者的算法成为对于前者的一个高效算法。

下界（lower bound） 如果知道问题 A 的任何算法都有某些资源需求，而且可以证明 A 可归约为 B，那么我们知道 B 至少也有这些资源需求，因为 B 的一个更好的算法将会蕴含着存在 A 的一个更好的算法（只要归约的开销低于 B 的开销）。也就是说，A 的性能是求解 B 的最佳性能的一个下界。例如，在 19.3 节中，我们使用这项技术来显示计算传递闭包与 Boolean 矩阵相乘一样困难，而且在 20.7 节使用这项技术来显示计算欧几里得 MST 与排序问题一样困难。

难解性（intractability） 特别是，要证明一个问题是难解的，可以将一个难解的问题归约为该问题来表明。例如，性质 21.18 显示了最短路径问题是难解的，因为哈密顿回路问题可归约为最短路径问题。

除了以上的这些一般含义，对于求解特定问题的特定算法，有关其性能的更详细信息显然可以直接关联到归约为前一类问题的其他问题。在找到一个上界时，可以分析相关算法、进行实验研究等等来确定它是否表示问题的一个更好的解决方案。当研究出一个通用的算法时，可以开发并测试它的一个良好的实现，然后开发扩展其应用性的相关的 ADTs。

在本章和下一章中，我们将归约作为基本的工具。对于所考虑的问题，通过将其他问题归约为这些问题，强调了问题的一般相关性，还强调了求解这些问题的算法的一般应用性。还要注意到很重要的一点，即对于一般性逐渐增长的问题形式化模型中的层次结构。例如，线性规划是一种通用的形式化模型，它之所以重要，不仅是因为很多问题可以归约为此问题，而且还因为并未确定它是 NP-难问题。换句话说，还没有已知的方法将一般最短路径问题（或任何其他 NP-难的问题）归约为线性规划问题。我们将在第 8 部分讨论这些问题。

并非所有问题都是可以解决的，但是对于已设计出的良好的通用模型，适合于大量我们确实知道如何求解的问题。网中的最短路径问题是此模型的第一个例子。当我们转到更一般的问题领域时，就进入了运筹学（operation research，OR）研究范畴，研究决策问题的数学方法，核心是开发和研究这样的模型。OR 中的一个关键挑战是找出最适合于求解一个问题的模型，并使此问题与模型相一致。这种行为有时称为数学规划（mathematical programming，这是在计算机出现之前给出的名字，现在“programming”这个词有新用法）。归约是一个现代的概念，与数学规划有着相同的本质，也是理解大量应用中的计算开销的基础。

练习

▷ **21.85** 使用性质 21.12 的归约来开发一个传递闭包实现（19.3 节中的 ADT 函数 GRAPH-tc），其中使用 21.3 节中所有对最短路径 ADT。

21.86 说明计算有向图中强分量的个数的问题可归约为带有负权值所有对最短路径问题。

21.87 按照性质 21.14 和性质 21.15 的构造方法，给出对应作业调度问题的差分约束和最短路径问题，其中作业 0～7 的长度如下。

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

约束为：

```
5-1  4-6  6-0  3-2  6-1  6-2
```

▷ **21.88** 给出练习 21.87 中的作业调度问题的一个解决方案。

○ **21.89** 假设练习 21.87 中的作业还有以下约束，作业 1 必定在作业 6 结束之前开始，作业 2 必定在作业 4 结束之前开始。使用性质 21.16 的证明中所描述的构造，给出此问题可归约

的最短路径问题。

21.90　显示带有正权值的无环网中所有对最长路径问题可归约为带有正常数的差分约束问题。

▷21.91　解释性质21.16的证明中的对应性为什么不能进行扩展，来说明最短路径问题可归约为具有截止期的作业调度问题。

21.92　扩展程序21.8来使用符号名而不是整数来表示作业（见程序17.10）。

21.93　设计一个ADT接口，为客户程序提供提出并解决差分约束问题的能力。

21.94　基于将差分约束问题的解归约为无环网中最短路径问题，对于练习21.93的接口，提供ADT函数实现。

21.95　基于归约为差分约束问题并使用练习21.93的接口，提供GRAPHsp的一种实现（用于无环网中）。

○21.96　在练习21.95中，无环网中最短路径问题的解决方案假设了存在解决差分约束问题的一种实现。如果你使用练习21.94中的实现，假设无环网中存在最短路径问题的一种实现，又会如何？

○21.97　证明任何两个NP-难的问题的等价性（也就是说，选择两个问题，并证明它们可相互归约）。

●●21.98　给出一种显式构造，将带有整数权值的网的最短路径问题归约为哈密顿问题。

●21.99　使用归约来实现一个ADT函数，此函数使用求解单源点最短路径问题的网ADT来解决如下问题：给定一个有向图，一个顶点索引的正权值数组，以及一个起始顶点v，找出从v到每个其他顶点的路径，使得此路径上顶点的权值之和达到最小。

○21.100　程序21.8并没有检查作为其输入的作业调度问题是否可行（有环）。描述其为不可行问题打印出的调度的特征。

21.101　设计一个ADT接口，提供客户程序提出并解决作业调度问题的能力。类似于程序21.8，基于将作业调度问题的解归约为无环网中最短路径问题，提供你的接口的ADT函数实现。

○21.102　在ADT接口中增加一个函数（并提供一个实现），打印出此调度中的最长路径。（这样的路径被称为是*关键路径*（critical path）。）

21.103　对于练习21.101的接口，提供一个ADT函数实现，按照图21-24的风格（见4.3节），输出绘制调度的PostScript的程序。

○21.104　开发一个产生作业调度问题的模型。对于一组合理大小的问题，使用此模型来测试练习21.101和21.103的实现。

21.105　基于将作业调度问题的解归约为差分约束问题，对于练习21.101的接口提供ADT函数实现。

○21.106　PERT（性能－评估－分析－技术）图是一个表示作业调度问题的网，其中边表示作业，如图21-25所示。开

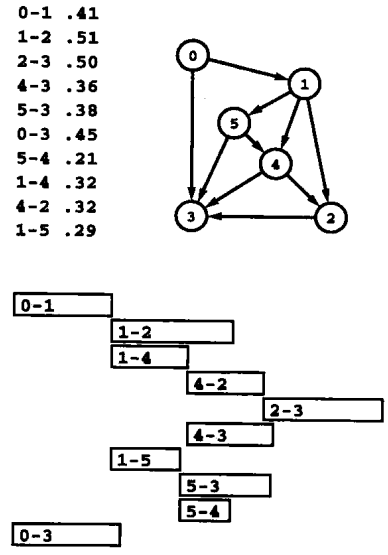


图21-25　PERT图

PERT图是一种表示作业调度问题的网，其中边表示作业。上图的网表示图21-22中描述的作业调度问题，其中边0-1、1-2、2-3、4-3、5-3、0-3、5-4、1-4、4-2和1-5分别表示图21-22中的作业0~9。调度中的关键路径是网中的最长路径。

发基于 PERT 图的练习 21.101 的作业调度问题的一种实现。

21.107 对于 V 个作业、E 个约束的作业调度问题，其 PERT 图中有多少个顶点？

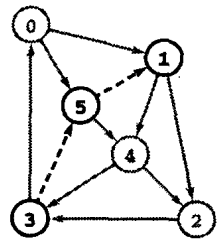
21.108 对于练习 21.106 中所讨论的基于边的作业调度表示（PERT 图）以及正文中所用的基于顶点的表示（见图 21-22），编写它们之间进行转换的程序。

21.7 负权值

我们现在转向处理最短路径中有负权值的难题。在本章的大部分内容中，大多数都是直观的例子，其中权表示距离或代价，使得负的边权值看上去不太可能；然而，我们在 21.6 节也看到当将其他问题归约为最短路径问题时，自然地出现了负的边权值。负权值不只是数学难题；相反，它们将最短路径问题的应用性显著地扩展成为求解其他问题的一个模型。这一潜在实用性是我们寻找涉及负权值的网问题的高效算法的源动力。

图 21-26 是一个较小的例子，描述了在网的最短路径中引入负权值所带来的影响。在出现负权值时，也许最重要的影响是小权值的最短路径往往比较大权值的最短路径有更多的边（low-weight shortest paths tend to have more edges than higher-weight path）。对于正权值，我们的重点是放在寻找捷径（shortcut）；但在负权值出现时，我们寻求绕道（detour），使用所能找到的尽可能多的带有负权值的边。这种影响将我们寻求“短”路径的直觉，变成对算法的理解，因此我们需要克制直觉，在一个基本的抽象层次上考虑问题。

```
0-1   .41
1-2   .51
2-3   .50
4-3   .36
3-5  -.38
3-0   .45
0-5   .29
5-4   .21
1-4   .32
4-2   .32
5-1  -.29
```



	0	1	2	3	4	5
0	0	0	.51	.68	.32	.29
1	1.13	0	.51	.68	.32	.30
2	.95	-.17	0	.50	.15	.12
3	.45	-.67	-.16	0	-.35	-.38
4	.81	-.31	.20	.36	0	-.02
5	.84	-.29	.22	.39	.03	0

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

图 21-26 带有负权值边的示例网

除了边 3-5 和 5-1 的权值为负，此示例网与图 21-1 所描述的网相同。自然地，这个变化对最短路径的结构产生了极大的影响，因为可以将右边的距离矩阵和路径矩阵与图 21-9 所对应的部分进行比较，很容易地看出这种影响。例如，此网中从 0 到 1 的最短路径是 0-5-1，其长度为 0；且从 2 到 1 的最短路径为 2-3-5-1，其长度为 -0.17。

性质 21.18 的证明中显示的网中最短路径和图中哈密顿路径之间的关系，与我们的观察结论有着密切的关系，即找出小权值的路径（一直称为“短”）等价于找出含有大量边数的路径（可以认为是“长”）。有了负权值，我们寻找长路径而不是短路径。

对这种情况进行修正的想法是找出最小边权值（最小的负值），然后，在所有边权值中加上此数的绝对值，将此网转换为一个没有负权值的网。这种朴素的方法根本不行，因为新网中的最短路径与原网中的最短路径没有多少关系。例如，在图 21-26 所描述的网中，从 4 到 2 的最短路径为 4-3-5-1-2。如果我们将 0.38 加到此图中的所有边权值上，使所有边权值为正，那么此路径的权值就从 0.20 增长到 1.74。但 4-2 的权值从 0.32 增长到 0.70，因此这条边将成为从 4 到 2 的最短路径。一条路径上的边越多，转换所带来的危害就越大；由上段的观察可得，此结果与我们的需要完全相反。即使这种简单的想法不起作用，将此网转换为一个等价的没有负权值的网但有相同的最短路径的目标是值得的；在本节最后，我们考虑达到此目标的一个算法。

到目前为止，我们的最短路径算法对最短路径问题做了两个限制，从而才能提供一个高

效的解决方案：它们要么不允许环，要么不允许负权值。是否存在对网强加的更松限制，使得含有环和负权值的网仍然会导致一个易解的最短路径问题？在本章一开始，我们接触过此问题的解决方案，但必须要加上限制条件：如果存在负环，路径必须是简单的，才能使得此问题有意义。是不是应当只关注不含这种环的网呢？

无负环网中的最短路径 给定一个可能有负边权值但并没任何负权值环的网，求解以下某个问题：找出连接两个给定顶点的一条最短路径（最短路径问题），找出从某个给定顶点到所有其他顶点的最短路径（单源点最短路径问题），或者找出连接所有对顶点之间的最短路径（所有对问题）。

性质 21.18 的证明将解决此问题的高效算法的可能性留做开放问题，因为如果不允许负环，则性质不成立。为了解决哈密顿回路问题，我们需要能够求解有大量负环的网中的最短路径问题。

此外，很多实际问题都完全可以归约为找出不含负环的网中的最短路径问题。我们已经看到这样的例子。

性质 21.19 *具有截止期的作业调度问题可归约为不含负环的网中的最短路径问题。*

证明 性质 21.15 的证明中，我们所提出的理由说明由性质 21.16 证明中的构造会得到不含负环的网。由作业调度问题，可以构造一个差分约束问题，其中变量对应着作业的开始时间；从差分约束问题，可以构造一个网。将所有权值取负，从而将一个最长路径问题转换为一个最短路径问题，也即转换对应着将所有不等式反向。网中从 i 到 j 的任何简单路径对应着包含这些变量的不等式组。由这些不等式可知，路径的存在性蕴含着 $x_i - x_j \leqslant w_{ij}$，其中 w_{ij} 为从 i 到 j 的这条路径上的权值之和。负环对应这此不等式的左边为 0，而右边为负值，因此，存在这样的环是矛盾的。 ■

21.6 节做出讨论的作业调度问题时提到过，这个结论隐含了假设我们的作业调度问题是可行的（存在解决方案）。实际上，我们并不做这样的假设，部分计算负担将会确定带有截止期的作业调度问题是否是可行的。在性质 21.19 证明的构造中，网中的负环蕴含着一个不可行的问题，因此这项任务对应于以下问题：

负环检测（negative cycle detection）给定网中是否存在一个负环？如果存在，找出此环。

一方面，这个问题未必是简单问题（有向图的简单环检测算法不能适用）；另一方面，它也未必是困难的问题（性质 21.16 中由哈密顿回路问题进行的归约也不适用）。我们的第一个挑战就是开发此任务的一个算法。

在具有截止期的作业调度应用中，负环对应着非常罕见的错误条件，但是要对此进行检查。我们甚至可能开发删除边以打破负环的算法，并进行迭代直到不存在负环为止。在其他应用中，检测负环是主要的目标，如下例所示。

套汇（arbitrage） 很多报纸都印有一些显示世界货币兑换率的表（例如，见图 21-27）我们可以将此表看作完全网的邻接矩阵表示。边 s-t 上的权值表示可以将货币 s 的 1 个单位兑换为货币 t 的 x 个单位。此网中的路径指定了多步兑换。例如，如果还存在一条权值为 y 的边 t-w，那么路径 s-t-w 表示将货币 s 的 1 个单位兑换为货币 w 的 xy 个单位。我们可能期望在任何情况下 xy 等于 s-w 的权值，但是此表表示一种复杂的动态系统，其中不能保证这样的一致性。如果我们找到一种 xy 小于 s-w 权值的情况，那么就能超越系统。假设 w-s 的权值为 z，且 $xyz > 1$，那么环 s-t-w-s 给出了一种将货币 s 的 1 个单位兑换为货币 s 的多于 1 个（xyz）单位的方法。也就是说，我们可以挣得百分比为 $100(xyz-1)$ 的收益。这种情况正是一个套汇机会的例子，如果在此模型之外没有强制要求（如对交易规模有所限制），那么这

就使我们可获得无限的收益。为了将这个问题转换为一个最短路径问题，我们将所有数字取对数，使得路径权值对应对边权值相加，而不是相乘，然后取负来改变比较方向。然后，从 s 到 t 的一条最短路径给出了将货币 s 兑换为货币 t 的最佳方式。最小权值环就是最佳套汇机会，但任何负环都可获利。

是否可以检测出一个网中的负环？或者是否可以找出不含负环的网中的最短路径呢？求解这些问题的高效算法的存在性与我们在性质 21.18 中所证明的一般问题的 NP-难度并不矛盾，因为由哈密顿问题得到的这两个问题的归约尚未得知。具体地说，由性质 21.18 中的归约可知，我们无法做到精心设计一个算法，能够保证有效地找出任何给定网中的最小权值路径，在这种网中允许负边权值。此问题的结论太一般。但我们可以解决刚才提到的此问题的受限版本。尽管不如本章早些时候所研究的此问题的另一种受限版本简单（正权值和无环网）。

	$	P	Y	C	S
$	1.0	1.631	0.669	0.008	0.686
P	0.613	1.0	0.411	0.005	0.421
Y	1.495	2.436	1.0	0.012	1.027
C	120.5	197.4	80.82	1.0	82.91
S	1.459	2.376	0.973	0.012	1.0

	$	P	Y	C	S
$	0.0	0.489	-0.402	-4.791	-0.378
P	-0.489	0.0	-0.891	-5.278	-0.865
Y	0.402	0.89	0.0	-4.391	0.027
C	4.791	5.285	4.392	0.0	4.418
S	0.378	0.865	-0.027	-4.415	0.0

图 21-27 套汇

上面的此表指定了从一种货币兑换为另一种货币的兑换率。例如，顶部行中的第二个元素表明 $1 可以买入 1.631 个货币 P。将 $1 000 兑换为货币 P，再换回来将得到 $1 000 * (1.631) * (0.613) = $999，就损失了 $1。但是将 $1000 兑换为货币 P，再换为货币 Y，然后再换回来，则得到 $1 000 * (1.631) * (0.411) * (1.495) = $1 002，将获得 0.2% 的套汇机会。如果将表中的所有数字的对数取负（下表），可以将它看作边权值可正可负的完全网的邻接矩阵。在这种网中，结点对应着货币，边对应着兑换，路径对应着兑换序列。以上所描述的兑换对应着图中的环 $-P-Y-$，其权值为 $-0.489 + 0.890 - 0.402 = -0.002$。最佳套汇机会为图中的最短环。

一般而言，正如在 21.2 节所提到的，Dijkstra 算法并不适合于有负权值的情况，即使是限制为不含负环的网。图 21-28 说明了这一点。基本困难在于算法依赖于按照路径长度递增的顺序来检查路径。证明此算法是正确的（见性质 21.2）假设了在路径中增加一条边会使路径更长。

Floyd 算法没有做出这种假设，即使在边权值可能为负时也很有效。如果不存在负环，该算法计算出最短路径；引人注目的是，如果存在负环，该算法至少能够检测出一个环。

性质 21.20 *Floyd 算法可以解决负环检测问题，并且可以解决不含负环的网中的所有对最短路径问题，所用时间与 V^2 成正比。*

证明 性质 21.8 并不依赖于边权值是否为负，然而，当出现负的边权值时，我们需要将此结果另做解释。矩阵中的每个元素都证实算法找出一条该长度的路径；特别是，在此距离矩阵的对角上的任何负元素可证实至少出现一个负环。在出现负环时，不能直接得出任何信息，因为算法隐式检查的路径不一定是简单路径：某些路径可能包含一个或多个涉及一个或多个负环的路线。然而，如果不存在负环，那么算法所计算的路径就是简单的，因为任何带有环的路径都意味着存在更少边的路径，且连接同样两个点的权值不会更大（删除环后的同一路径）。 ■

性质 21.20 的证明没有给出关于如何由 Floyd 算法所计算出的距离矩阵和路径矩阵找出一条特定负环的特定信息。我们将这个任务留作练习（见练习 21.122）。

Floyd 算法解决了不含负环的图的所有对最短路径问题。尽管 Dijkstra 算法在可能含有负权值的网中失效，我们也可使用 Floyd 算法来求解不含负环的稀疏网中的所有对最短路径问题，所用时间与 V^3 成正比。如果在这样的网中有一个单源点问题，那么可以使用所有对问题的 V^3 解决方案，尽管过于浪费，但也是我们所见到的单源点问题的最佳解决方案。是否能开发这些问题的快速算法呢？即其运行时间可以达到在边权值为正时的 Dijkstra 算法的运

行时间（对于单源点问题为 $E \lg V$，对于所有对问题为 $VE \lg V$）呢？对于所有对问题，答案是肯定的，而对于单源点问题则可能将最坏情况下的界限降至 VE。然而，对于一般的单源点最短路径问题要打破 VE 的屏障仍然是一个长期开放的问题。

以下方法为解决不含负环的网中的单源点最短路径问题提供了一个简单而有效的基础。为了计算出从顶点 s 出发的最短路径，我们（通常）维护一个顶点索引的数组 wt，使得 wt[t] 包含从 s 到 t 的最短路径。初始化 wt[s] 为 0，wt 的其他元素为一个大的观察哨值，然后如下计算最短路径：

以任何顺序考虑网中的边，并沿着每条边松弛。执行 V 遍。

我们使用术语 Bellman-Ford 算法来指对边执行 V 遍的一般方法，并以任意顺序考虑边。某些作者使用此术语来描述一种更一般的方法（见练习 21.130）。

性质 21.21 *使用 Bellman-Ford 算法，可以解决不含负环的网中的单源最短路径问题，所用时间与 VE 成正比。*

证明 对所有 E 条边执行 V 遍，因此总时间与 VE 成正比。为了显示此计算可以得到所需结果，我们对 i 进行归纳来证明，即在执行第 i 遍之后，对于所有 v，wt[v] 不大于从 s 到 v 且包含 i 条或更少边的最短路径的长度（如果此路径不存在则为 maxWT）。如果 i 为 0，则此断言自然成立。假设此断言对于 i 成立，那么对于每个给定的顶点 v 有两种情况：在从 s 到 v 且有 $i+1$ 条或更少条边的路径中，可能存在（也可能不存在）有 $i+1$ 条边的最短路径。如果在从 s 到 v 且有 $i+1$ 条或更少条边的路径中，其中最短的一条路径长度为 i 或更少，那么 wt[v] 不会改变，仍然有效。否则，存在一条从 s 到 v 且包含 $i+1$ 条边的路径，它比任何从 s 到 v 且有 i 条边或更少的路径都要短。该路径必然由一条从 s 到某个顶点 w 的路径再加上边 w-v 组成。由归纳假设，wt[w] 是从 s 到 w 的最短距离的上界，而且第 $(i+1)$ 遍检查是否每条边构成到达其目的顶点的一条新的最短路径上的最后一条边。特别是，它将检查边 w-v。

经过 $V-1$ 次迭代后，对于所有顶点 v，wt[v] 就是从 s 到 v 的任何有 $V-1$ 条边或更少边的最短路径长度的下界。我们可以在 $V-1$ 次迭代后终止，因为有 V 条边或更多条边的任何路径必定有一个（正或零成本）环，可以找到一条包含 $V-1$ 条边或更少边的路径，该路径与删除环之前的路径有相同的长度，或者更短。由于 wt[v] 是从 s 到 v 的某条路径的长度，它也是此最短路径长度的一个上界，因此必定等于最短路径长度。

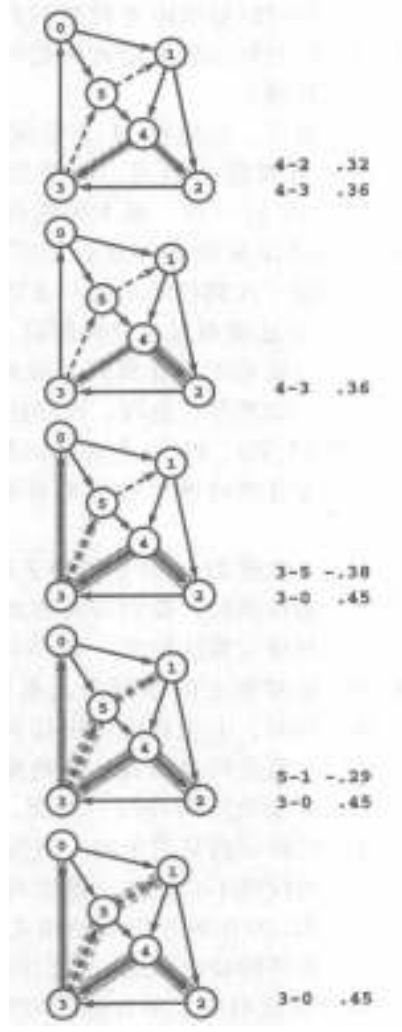


图 21-28 Dijkstra 算法失效（负权值）

在此例中，Dijkstra 算法确定 4-2 是从 4 到 2 的最短路径（长度为 .32），漏掉了一条更短路径 4-3-5-1-2（长度为 .20）。

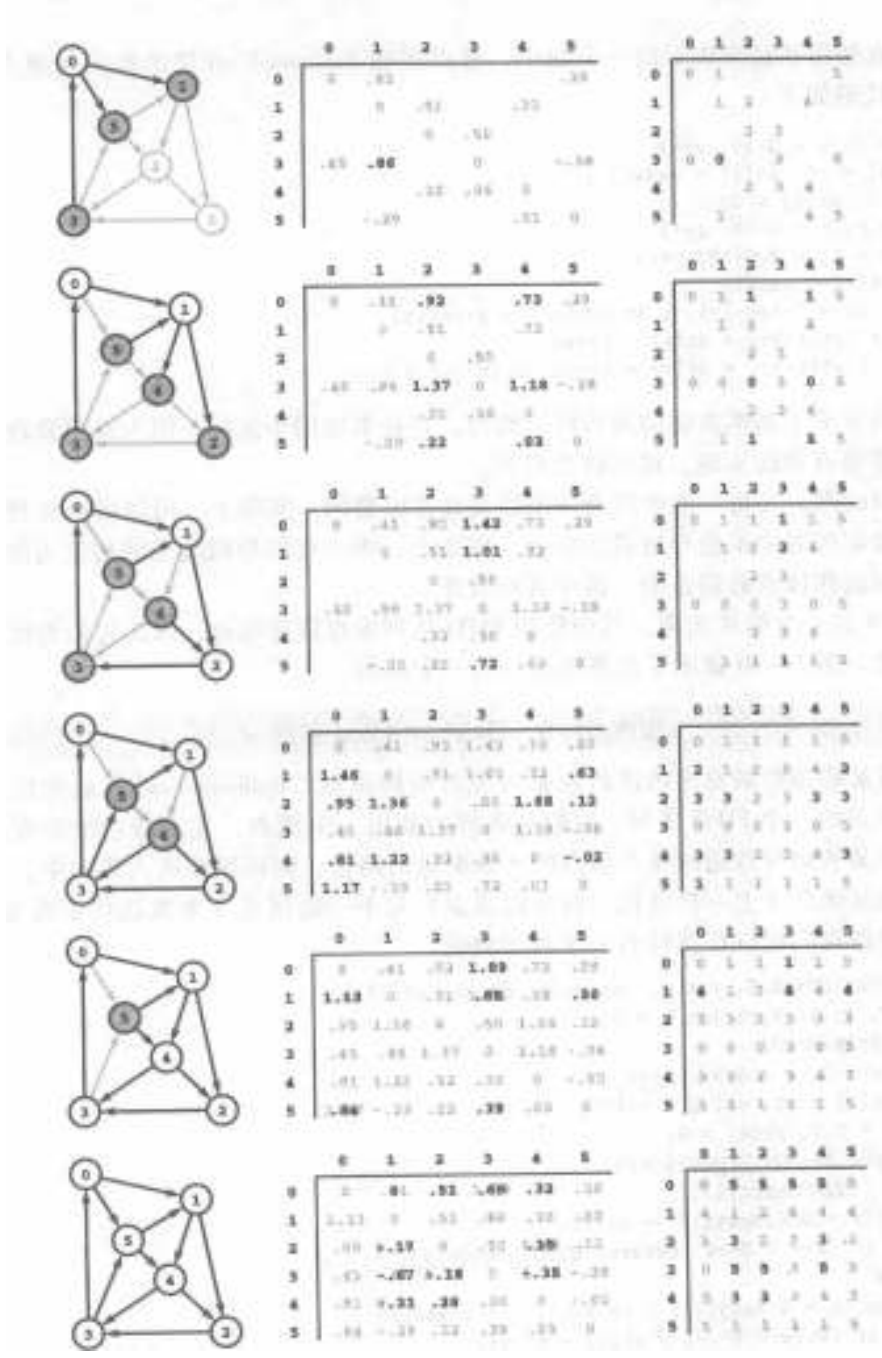


图21-29 Floyd算法（带有负权值）

此序列显示了使用Floyd算法，构造带有负权值的有向图的所有对最短路径矩阵的过程。第一步与图21-14中所示相同。然后负边5-1在第二步中起作用，并发现路径5-1-2和5-1-4。此算法包含对于边权值的完全相同的松弛序列，但结果却不同。

尽管我们并没有显式地考虑 st 数组，同理可证 st 数组是以 s 为根的最短路径树的父链接表示。■

例如，在使用邻接表表示的一个图中，可以实现 Bellman-Ford 算法找出从顶点 s 出发的最短路径，代码如下：

```
for (v = 0; v < G->V; v++)
  { st[v] = -1; wt[v] = maxWT; }
wt[s] = 0; st[s] = 0;
for (i = 0; i < G->V; i++)
  for (v = 0; v < G->V; v++)
   if (wt[v] < maxWT)
     for (t = G->adj[v]; t != NULL; t = t->next)
       if (wt[t->v] > wt[v] + t->wt)
         { wt[t->v] = wt[v] + t->wt; st[t->v] = v; }
```

此段代码展示了基本方法的简洁性。然而，它并未应用于实际。因为简单修改就能得到对于大多数图更有效的实现，稍后将会看到。

对于典型的图，在每一遍中检查每条边是非常浪费的。实际上，可以很容易预计到大量的边在任何给定的遍中不会导致成功松弛。事实上，唯一可以导致改变的边仅为由某个顶点发出的边，而这些顶点的值在前一遍中有所改变。

程序 21.9 是一个简单实现，其中使用 FIFO 队列来存放这些边，因此它们是每遍中唯一需要检查的边。图 21-30 显示了此算法运行的一个示例。

程序 21.9 Bellman-Ford 算法

对于沿着某些顶点的发出边进行松弛可能有效的顶点，Bellman-Ford 算法的这一实现维护所有这些顶点的一个 FIFO 队列。我们从队列中取出一个顶点，并沿着它的所有边进行松弛。如果其中某些边导致通向某个顶点的一条更短的路径，则该顶点放入队列中。观察哨值 G->V 将当前批顶点（上一次迭代中改变的顶点）与下一批顶点（本次迭代中改变的顶点）分离开，并使得我们可以在执行 G->V 遍后终止。

```
void GRAPHbf(Graph G, int s, int st[], double wt[])
  { int v, w; link t; int N = 0;
    QUEUEinit(G->E);
    for (v = 0; v < G->V; v++)
      { st[v] = -1; wt[v] = maxWT; }
    wt[s] = 0.0; st[s] = 0;
    QUEUEput(s); QUEUEput(G->V);
    while (!QUEUEempty())
      if ((v = QUEUEget()) == G->V)
        { if (N++ > G->V) return; QUEUEput(G->V); }
      else
        for (t = G->adj[v]; t != NULL; t = t->next)
          if (wt[w = t->v] > wt[v] + t->wt)
            { wt[w] = wt[v] + t->wt;
              QUEUEput(w); st[w] = v; }
  }
```

对于实际应用中出现的网，程序 21.9 用于求解单源点最短路径问题问题很有效，但它的最坏情况下的性能仍然与 VE 成正比。对于稠密图，此运行时间并不比 Floyd 算法更好，而 Floyd 算法是找出所有对的最短路径，而非仅找出从单个顶点出发的那些最短路径。对于

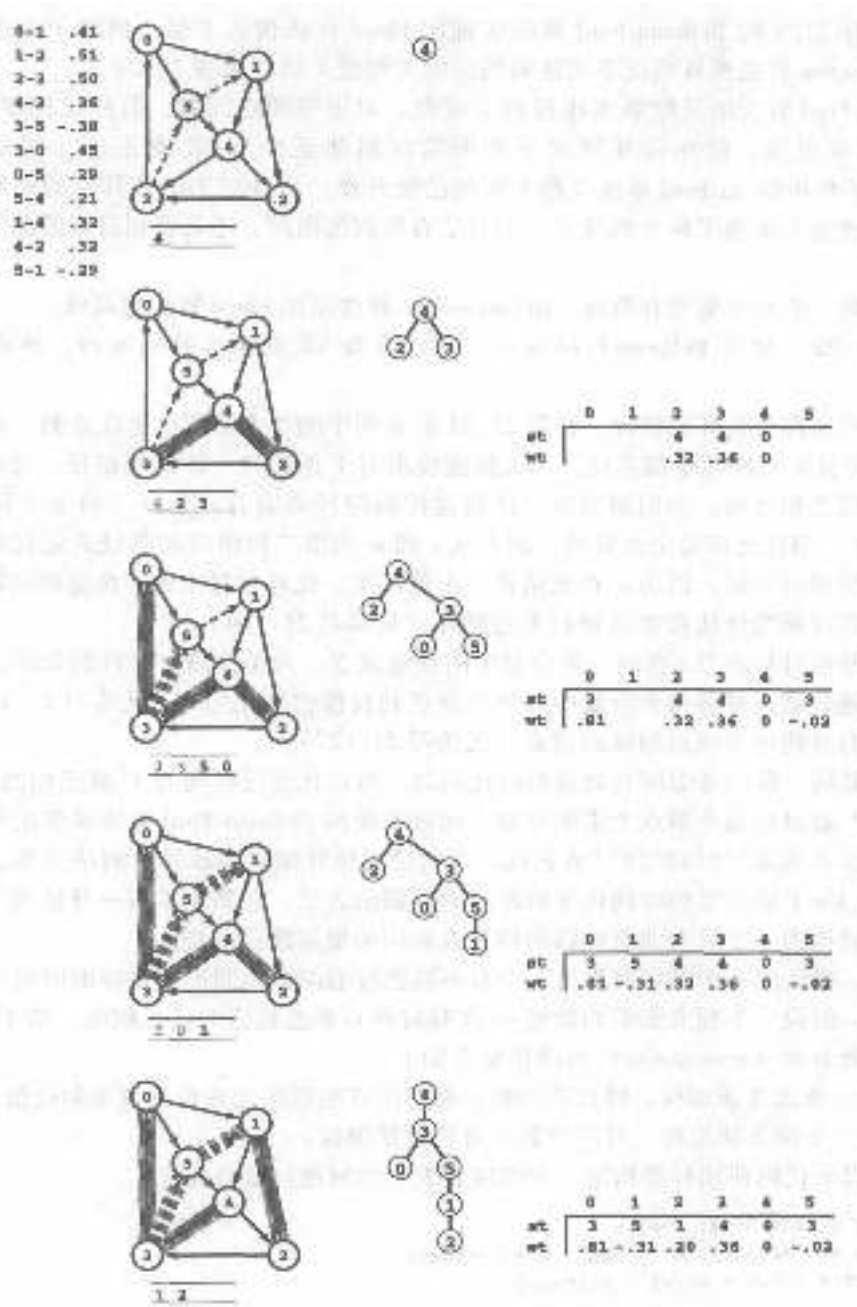


图 21-30 Bellman-Ford 算法（带有负权值）

对于图 21-26 中所描述的网，此图显示了使用 Bellman-Ford 算法来找出从顶点 4 出发的最短路径的结果。算法逐遍运行，每一遍检查由 FIFO 队列中的所有顶点发出的边。队列中的内容显示在每个绘制图的下方，有阴影的元素表示上一遍中队列中的内容。每当找到一条能够减小从 4 到其目的顶点的路径长度的边时，就执行松弛操作，将此目的顶点放入队列中，其边放入 SPT 中。所绘制的灰边包含了每步后的 SPT，同时也在中间以有方向的形式显示（所有边都向下）。我们从一个空 SPT 开始，队列中仅有 4（上图）。在第 2 遍中，沿着 4-2 和 4-3 松弛，将 2 和 3 放在队列中。在第 3 遍中，检查了 2-3，但并没有松弛。然后沿着 3-0 和 3-5 松弛，将 0 和 5 放入队列中。在第 4 遍中，沿着 5-1 松弛，然后检查 1-0 和 1-5，未做松弛，将 1 放入队列中。在最后一遍中（下图），沿着 1-2 松弛。算法开始时的操作类似 BFS，但与其他图的搜索方法都不同，如在最后一步中，它可能改变树边。

稀疏图，程序 21.9 的 Bellman-Ford 算法实现比 Floyd 算法快达 V 倍，但对于无负权值边的网，要比 Dijkstra 算法最坏情况下所达到的性能大约慢 V 倍（见表 19-2）。

Bellman-Ford 算法的其他版本也得到了研究，对于单源点问题，有些比程序 21.9 中的 FIFO 队列版本更快，但在最坏情况下的所需时间都至少与 VE 成正比（例如，见练习 21.132）。基本 Bellman-Ford 算法在数十年前已经开发；而且对于很多其他的图问题，尽管已经看到在性能上取得了巨大的进步，但对于有负权值的网，还未看到最坏情况下有更好性能的算法。

对于检测一个网中是否有负环，Bellman-Ford 算法也比 Floyd 算法更高效。

性质 21.22 *使用 Bellman-Ford 算法，可以在与 VE 成正比的时间内，解决负环检测问题。*

证明 即使存在负环的情况，性质 21.21 的证明中的基本归纳也是成立的。如果执行第 V 次迭代，而且所有松弛步都成功，那么就能找出有 V 条边的一条最短路径，此路径将 s 与网中的某个顶点相连接。由归纳假设，任何这样的路径必定有一个环（将某个顶点 w 与其自身相连接），而且此环必定是负的，因为从 s 到 w 的第二次出现的路径必定比从 s 到 w 的第一次出现的路径要短，因为 w 在此路径上出现两次。此环也将出现在负链接的数组中；因此，也可以通过周期性地检查 st 链接来检测环（见练习 21.134）。

这个证明仅对与源点 s 在同一强分量中的顶点成立。为了检测一般性的负环，我们可以计算出强连通分量，并将每个分量中的每个顶点的权值初始化为 0（见练习 21.126），或者增加一个连向其他每个顶点的虚拟顶点（见练习 21.127）。 ■

在本章最后，我们考虑所有对最短路径问题。可以比运行时间与 V^3 成正比的 Floyd 算法做得更好吗？通过在每个顶点上求解单源点问题来使用 Bellman-Ford 算法来解决所有对最短路径问题揭示出其运行时间与 V^2E 成正比。我们没有更详细地考虑这个解决方案，因为存在可以在与 VE log V 成正比的时间内求解所有对问题的方法。它基于本节一开始所考虑的一种想法：将网转换为一个只有非负权值的网且有相同的最短路径结构。

事实上，将任何一个网转换为另一个有不同边权值但有相同最短路径的网时，存在着很大的灵活性。假设一个顶点索引的数组 wt 含有对网 G 顶点权值的任意赋值。基于这些权值，对图进行*重新加权*（reweighting）的操作定义如下：

- 要对一条边重新加权，将此边的源点和目的点的权值之差加到该边的权值上。
- 要对一个网重新加权，对网中的所有边重新加权。

例如，以下代码使用标准约定，对邻接表表示的网进行重新加权：

```
for (v = 0; v < G->V; v++)
  for (t = G->adj[v]; t != NULL; t = t->next)
    t->wt = t->wt + wt[v] - wt[t->v]
```

此操作是一个简单的线性时间处理，对于所有网都是良定义的，而不论其权值如何。显然，变换后网中的最短路径与原网中的最短路径相同。

性质 21.23 *对网重新加权并不影响其最短路径。*

证明 给定两个顶点 s 和 t，重新加权改变了从 s 到 t 的任何路径的权值，其中只是增加了 s 和 t 的权值之差。通过对路径长度进行归纳很容易证明这一点。在对网进行重新加权时，从 s 到 t 的每条路径的权值改变量相同，长路径和短路径都如此。特别是，这个事实直接表明转换后的网中的任何两个顶点之间的最短路径长度与原网中它们之间的最短路径长度相同。 ■

由于不同顶点对之间的路径要做不同的重新加权，对于涉及最短路径长度的比较问题（例如，计算网的直径），重新加权可能会对其产生影响。在这些情况下，我们需要在完成最短路径计算之后但在使用该结果之前反向进行重新加权。

重新加权对有负环的网没有作用：此操作不会改变任何环的权值，因此无法通过重新加权来去除环。但是对于不含负环的网，则可以试图找出一组顶点权值，使得重新加权可导致边权值非负，而无论原来的边权值是什么。有了非负边权值，就可以利用 Dijkstra 算法的所有对版本来解决所有对最短路径问题。例如，对于示例网，图 21-31 给出了一个这样的例子，而图 21-32 则显示了在不含负边的转换后的网中利用 Dijkstra 算法来计算最短路径的过程。以下性质说明了总是可以找出这样一组权值。

性质 21.24 *在不含负环的任何网中，选择任何一个顶点，并为各顶点赋予一个权值，此权值等于从 s 到 v 的一条最短路径的长度。可以利用这些顶点权值对网进行重新加权，使得将 s 与其可达顶点的相连接的边均有非负权值。*

证明 给定任意边 v-w，v 的权值为到达 v 的一条最短路径的长度，w 的权值为到达 w 的一条最短路径的长度，如果 v-w 是到达 w 的最短路径上的最后一条边，那么 w 的权值与 v 的权值之差恰好为 v-w 的权值。换句话说，对此边重新加权将得到的权值为 0。如果通过 w 的最短路径的权值没有通过 v，那么 v 的权值加上 v-w 的权值必定大于或等于 w 的权值。也就是说，重新对此边进行加权将得到一个正权值。 ■

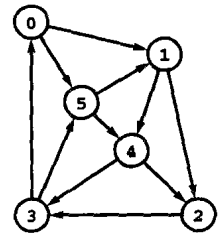
正如使用 Bellman-Ford 算法检测负环一样，在一个不含负环的网中，有两种方法进行处理可使每个边权值为非负。要么从每个强连通分量的一个源点开始，要么增加一个虚拟顶点，它与网中的每个顶点之间的边的权值为 0。无论哪一种情况，其结果都是最短路径生成森林，我们可以使用它来为每个顶点赋以权值（从此根到它在 SPT 中顶点的路径的权值）。

例如，图 21-31 中所选择的权值即为从 4 出发的最短路径的长度，因此对于以顶点 4 为根的最短路径上的边，在重新加权后的网中其权值为 0。

总之，对于含有负边权值且不含负环的网，可以如下处理来求解所有对最短路径问题：

- 应用 Bellman-Ford 算法找出原网中的一个最短路径森林。
- 如果算法检测出一个负环，则报告这一事实并终止。
- 由森林对网进行重新加权。
- 将所有对版本的 Dijkstra 算法应用到重新加权的网中。

这个计算完成之后，路径矩阵给出这两个网中的最短路径，距离矩阵给出重新加权的网中的路径长度。这一系列步骤有时称为 Johnson 算法（Johnson's algorithm）（见第 5 部分参考文献）。



	0	1	2	3	4	5
wt	.81	-.31	.20	.36	0	-.02

```
0-1  .41 +  .81 -(-.31) = 1.53
1-2  .51 +(-.31)-  .20  =   0
2-3  .50 +  .20 -  .36  =  .34
4-3  .36 +   0  -  .36  =   0
3-5 -.38 +  .36 -(-.02) =   0
3-0  .45 +  .36 -  .81  =   0
0-5  .29 +  .81 -(-.02) = 1.12
5-4  .21 +(-.02)-   0   =  .19
1-4  .32 +(-.31)-   0   =  .01
4-2  .32 +   0  -  .20  =  .12
5-1 -.29 +(-.02)-(-.31) =   0
```

图 21-31 对网重新加权

给定顶点权值的赋值（上图），通过对网中每条边的权值增加其源点和目的顶点的权值之差，可以对网的所有边进行重新加权。重新加权并不影响最短路径，因为它对连接每对顶点路径的改变量是一样的。例如，考虑路径 0-5-4-2-3：其权值为 .29 + .21 + .32 + .50 = 1.32；它在重新加权网中的权值为 1.12 + .19 + .12 + .34 = 1.77；这些权值差为 .45 = .81 − .36，即 0 和 3 的权值差；0 和 3 之间的所有路径的权值都改变这个量。

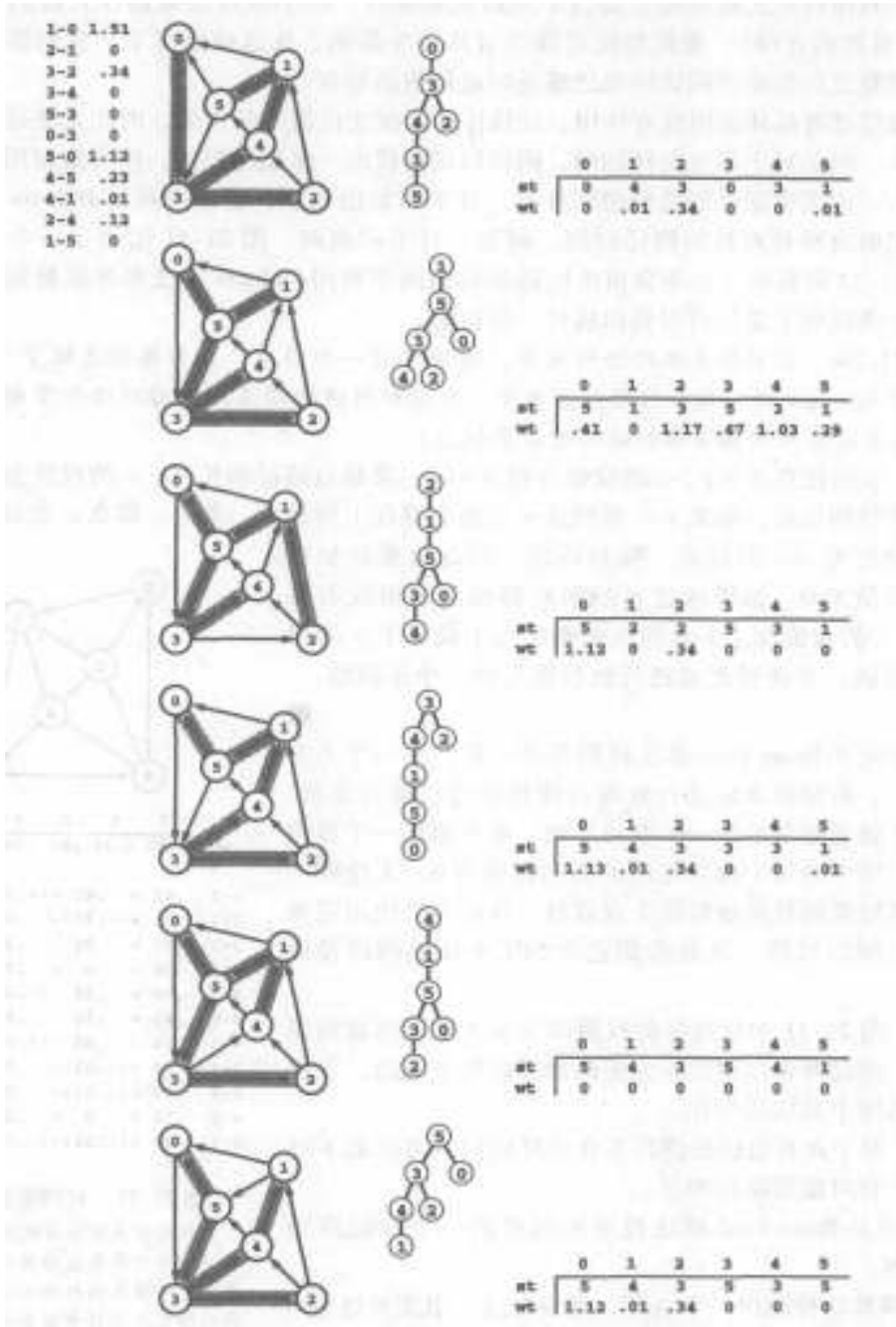


图 21-32 重新加权网中的所有对最短路径

对于图 21-31 中重新加权的网，这些图描述了其逆网中每个顶点的 SPT，它们可由 Dijkstra 算法计算得到图 21-26 中的原网中的最短路径。这些路径与重新加权之前的网中的路径相同。因此，如同在图 21-9 中所示，这些图中的 st 向量即为图 21-26 中的路径矩阵的那些列。此图中的 wt 向量对应距离矩阵中的列，但我们必须撤销对每个元素的重新加权，可以通过减去路径中源顶点的权值并增加目的顶点的权值来做到（见图 21-31）。例如，下图的第 3 行，可见这两个网中的从 0 到 3 的最短路径均为 0-5-1-4-3，在这里所示的加权网中其长度为 1.13。再看图 21-31，可以计算出它在原网中的长度，即减去 0 的权值并加上 3 的权值得到此结果 1.13 − .81 + .36 = .68，即为图 21-26 中的距离矩阵的第 0 行和第 3 列的元素。此网中到 4 的所有最短路径长度均为 0，因为我们正是用这些路径来进行重新加权的。

性质21.25 使用Johnson算法，可以在与 $VE\log_d V$ 成正比的时间内，解决不含负环的网中的所有对最短路径问题，其中如果 $E<2V$，则 $d=2$；否则，$d=E/V$。

证明 见性质21.22～21.24。以及上一段中的总结。由性质21.7和21.22可直接得到运行时间的最坏情况下界。■

为了实现Johnson算法，可以组合程序21.9的实现、性质21.23之前描述的重新加权代码以及程序21.4中Dijkstra算法的所有对最短路径实现（或者对于稠密图，程序20.3）。在性质21.22的证明中提到过，对于非强连通的网，必须对Bellman-Ford算法进行适当修改（见练习21.135～21.137）。为了完成所有对最短路径接口的实现，在将两个向量复制进距离矩阵和路径矩阵时，要么通过减去起始顶点的权值再加上目的顶点的权值来计算出真正的路径长度（撤销对这些路径上的重新加权）；要么将此计算放入此ADT实现的GRAPHdist中。

对于不含负环的网，检测环的问题比计算由一个源点到所有其他顶点的最短路径问题更容易解决；而后者较之于计算连接所有顶点对的最短路径问题解决起来又更为容易。这些事实与我们的直观认识完全一致。与此相反，对于包含负权值的网，有一些类似的事实则不是直观的：本节所讨论的算法表明，对于有负权值的网，对于这3个问题，已知的最好算法都有着类似的最坏情况性能特征。例如，在最坏情况下，要确定一个网是否有一个负环，显然这与在有同样规模且无负环的网中找出所有最短路径一样困难。

练习

▷**21.109** 修改练习21.5和21.6的随机网生成器，使其通过重新调整来产生 a 和 b 之间（其中 a 和 b 均位于 -1 和1之间）的权值。

▷**21.110** 修改练习21.5和21.6的随机网生成器，使其通过对边权值中的固定百分比（该值由客户程序提供）取负来产生负权值。

○**21.111** 对于尽可能大的范围的 V 和 E 值，开发一个程序，使其利用练习21.109和21.110中的生成器来产生一个负权值占较大百分比、但至多只有几个负环的网。

21.112 找出一个在线或报纸上的货币兑换表。使用它构建一个套汇表。注意：要避免仅由几个值计算出该表，因为这样不能给出足够精确的引起关注的兑换信息。额外收获：可以借此在货币市场上大赚一笔！

●**21.113** 使用为练习21.112所找出的兑换信息源，构建一列套汇表（任何信息源都周期性发布不同的表）。找出你能在表中发现的所有套汇机会并试图找出它们之间的模式。例如，机会是在每日出现，还是出现之后就很快修正呢？

21.114 开发一种产生随机套汇问题的模型。你的目标是产生与练习21.113中所用表尽可能类似的表。

21.115 开发一种产生带有截止期的随机作业调度问题的模型。你的目标是产生可能可行的非平凡的问题。

21.116 修改练习21.10的接口与实现，使用对最短路径问题的一个归约，使客户程序有能力提出并解决带有截止期的作业调度问题。

○**21.117** 解释为什么以下观点是不成立的：通过性质21.15证明中所用的构造，最短路径问题可归约为差分约束问题，而差分约束问题可平凡地归约为线性规划，因此，由性质21.17，线性规划是NP-难的。

21.118 不含负环的网的最短路径问题可归约为带有截止期的作业调度问题吗？（这两个问题是等价的吗？）证明你的答案。

○ **21.119** 找出图21-27中所示例子的最小权值环（最佳套汇机会）。

▷ **21.120** 对于可能含有负边权值的网，证明找出其最小权值环是NP-难的。

▷ **21.121** 对于某种网，如果仅仅是离开源点的边带有负权值，说明Dijkstra算法对于这样的网也可正确工作。

▷ **21.122** 基于Floyd算法开发一个ADT实现，为客户程序提供检查网中是否存在负环的能力。

21.123 按照图21-29风格，对于练习21.1中所定义的网，对边5-1和4-2的权值取负，使用Floyd算法，显示计算其所有对最短路径的过程。

● **21.124** Floyd算法对于完全网（有V^2条边的网）是最优的吗？证明你的答案。

21.125 按照图21-3～21-32风格，对于练习21.1中所定义的网，对边5-1和4-2的权值取负，使用Bellman-Ford算法，显示计算其所有对最短路径的过程。

▷ **21.126** 基于Bellman-Ford算法开发一个ADT实现，使用从每个强连通分量中的某个源点开始的方法，为客户程序提供检查网中是否存在负环的能力。

▷ **21.127** 基于Bellman-Ford算法开发一个ADT实现，使用一个边连向网中所有顶点的虚拟顶点，为客户程序提供检查网中是否存在负环的能力。

○ **21.128** 给出一组图，使得程序21.9对于这组图找出负环所用时间与VE成正比。

▷ **21.129** 对于练习21.89中的带有截止期的作业调度问题，显示程序21.9所计算出的调度。

○ **21.130** 证明以下一般算法可以解决单源点最短路径问题："松驰任何边；继续直到没有边可被松驰。"

21.131 修改程序21.9的Bellman-Ford算法实现，使之使用随机队列，而不是FIFO队列（练习21.130的结果证明了这种方法是正确的）。

○ **21.132** 修改程序21.9的Bellman-Ford算法实现，使之使用双端队列，而不是FIFO队列，满足边按照以下规则放入双端队列中：如果边以前已放入过双端队列，则将其放在队列开始处（类似于堆栈）；如果边是首次遇到，则将其放在队列最后（类似于队列）。

21.133 进行实验研究，对于各种类型的网（见练习21.109～21.111），比较练习21.131和练习21.132的实现与程序21.9性能。

○ **21.134** 修改程序21.9的Bellman-Ford算法的实现，从而实现一个函数GRAPHnegcycle，使其返回任何负环中的任何顶点的索引，如果网中不存在负环，则返回-1。当出现一个负环时，此函数还应该保存st数组，使得数组中的以下链接能以正常的方式（从返回值开始）通过此环进行跟踪。

○ **21.135** 修改程序21.9的Bellman-Ford算法的实现，从而为Johnson算法设置所需要的顶点权值，使用以下方法。每当队列为空时，扫描st数组找出一个其权值尚未设置的顶点，并返回以此顶点做为源点的算法（将同一连通分量中的所有顶点的权值设置为该源点），继续这一过程，直到所有强连通分量都已被处理。

▷ **21.136** 对于一般网的邻接表表示（基于Johnson算法），通过对程序21.9和程序21.4进行适当的修改，开发一个所有对最短路径ADT接口的实现。

21.137 开发稠密网（基于Johnson算法）的所有对最短路径的ADT接口的实现（见练习21.136和练习21.43）。进行实验研究，对于各种类型的网（见练习21.109～21.111），将你的实现与Floyd算法（程序21.5）进行比较。

● **21.138** 在练习21.137的解中增加一个ADT函数，允许客户程序减小一条边的成本。返回

一个指示此行为是否创建负环的标志。如果没有，则更新路径矩阵和距离矩阵，以反映任何新的最短路径。你的函数所需时间应该与 V^2 成正比。

- **21.139** 实现网 ADT 函数，它类似练习 21.138 中所描述的函数，并允许客户程序插入和删除边。
- **21.140** 对于权值限定在一个常量的绝对值范围内的特殊情况，开发一个算法，突破一般网中单源点最短路径算法问题的屏障 VE。

21.8 展望

表 21-4 总结了本章所讨论过的算法，并给出了其最坏情况下的性能特征。如在 21.6 节中所讨论的，这些算法有着广泛应用，因为从某种技术层面，最短路径问题与大量的其他问题有关，它们直接导致了求解这类问题的高效算法，或者至少指明了存在这样的算法。

对于边权值可能为负的网，寻找最短路径的一般问题是难解的。最短路径问题是划分难解问题和容易问题界限的一个良好尺度，因为当将边权值限制到正的边权值或是无环时，甚至在对子问题进行限制，存在负边权值但不存在负环时，有大量的算法可以求解这个问题的不同版本。尽管对于不含负环的网的单源点问题和含有非负权值的网的所有对最短路径，其最佳已知下界和最佳已知算法之间存在巨大的鸿沟，但有些算法确实是最优的或渐近最优的。

表 21-4 最短路径算法的开销

此表总结了本章所考虑的各种最短路径算法的开销（最坏情况下的运行时间）。保守给出的最坏情况下的界限可能对于预测实际网的性能并无用处，特别是 Bellman-Ford 算法，其一般情况下运行时间为线性。

权值约束	算 法	开 销	注 释
单源点			
非负	Dijkstra	V^2	最优（稠密图）
非负	Dijkstra（PFS）	$E \lg V$	保守界
无环	源点队列	E	最优
不含负环	Bellman-Ford	VE	有改进空间?
无	开放问题	?	NP-难
所有对			
非负	Floyd	V^3	对所有网相同
非负	Dijkstra（PFS）	$V E \lg V$	保守界
无环	DFS	$V E$	对所有网相同
不含负环	Floyd	V^3	对所有网相同
不含负环	Johnson	$V E \lg V$	保守界
无	开放问题	?	NP-难

这些算法都是基于少数抽象操作并可以调整到一般环境中。具体地说，我们对边权值所做的唯一操作是加法操作和比较操作：任何允许这些操作的环境都可作为最短路径算法的平台。如前面提到的，这一点将计算有向图的传递闭包的算法与找出网中最短路径的算法统一起来。负边权值所带来的困难对应于这些抽象操作的一个单调性：如果我们可以确保两个权值之和不会小于其中任何一个权值，那么就可以使用 21.2 ~ 21.4 节中的算法；如果不能确

保这一点，就必须使用21.7中的算法。将这些考虑封装在ADT中是很容易做到的，而且扩展了算法的用途。

最短路径问题将我们放在一个十字路口上，一边是基本的图处理算法，另一边是无法解决的问题。最短路径问题是我们所考虑的具有类似特征的几个其他类问题中的第一类问题，包括网络流问题（network flow problem）和线性规划（linear programming）。如在最短路径中存在界限一样，在这些领域的容易问题和难解问题之间也存在明显界限。在各种限制适当时，不仅有大量有效算法可用，而且有大量机会发明更好的算法，还有很多情况会遇到NP-难问题。

在计算机和计算机算法发明之前，许多此类的问题作为运筹学问题得到了深入的研究。在历史上，运筹学主要关注的是一般性的数学和算法模型，而计算机科学所强调的则是即可允许有效实现又可允许构建一般解的特定算法学解决方案和基本抽象。作为源于运筹学的模型和源于计算机科学的基本算法学抽象均已用于开发可以解决大型实际问题的计算机实现。运筹学和计算机科学之间的界限在这些领域已经模糊：例如，在这两个研究领域的科研人员寻求诸如最短路径等问题的有效解决方法。在处理更为困难的问题时，会利用这两个研究领域中的经典方法。

第22章 网 络 流

图、有向图和网不仅是数学上的抽象，在实际中也相当有用，因为它们可以帮助我们解决很多重要的问题。在这一章里，我们扩展求解网络问题的模型，以便包含以下动态的情形，想象有物流通过网络，而且不同路径上成本不同。这些扩展可以处理一大类问题，并且具有广泛应用。

我们看到利用少数几个自然模型就能处理这些问题和应用，而且这些模型可以通过归约从一个模型变成另一个模型。形式化这些基本问题的方法有几种，它们在理论上是等价的。为了实现解决这些问题的方法，我们要解决两个特定的问题，首先研制解决这些问题的高效算法，然后研制一些通过找出归约到已知问题的其他问题的求解算法。

在实际中，我们并不总是能够像这种理想情况一样，能自由地选择。因为并没有证明每一对问题之间的归约关系，而且解决这些问题的最优算法极少。可能某个问题的高效解法尚未发现，也可能对于一对给定的问题，尚未发现其高效的归约。我们在本章中所讨论的网络流问题的形式化一直很成功，不仅由于可以容易地定义很多问题与网络流问题的归约，而且由于解决基本网络流的大量高效算法已被发明出来。

以下例子可说明使用网络流模型、算法和实现所能处理的问题。这些问题可分成几类：配送（distribution）问题、匹配（matching）问题和分割（cut）问题。我们依次考察。我们不只是研究这些例子中的细节，还将指出几个不同的相关问题。本章稍后在着手研制和实现算法时，将会给出这里提到过的这些问题的严格描述。

在配送问题中，我们所关心的是在网络中将对象从一处移到另一处。无论是通过遍及全国的高速公路将汉堡和鸡块送抵快餐店或是将玩具和衣物送抵折扣店，还是通过遍布世界的通信网络将软件送至计算机或将位流送至显示屏，其核心问题都是一样的。在管理一个大型而复杂的业务时，配送问题是我们所面临的一个典型难题。解决这类问题的算法已得广泛应用，并且在很多应用中非常关键。

商品配送（Merchandise distribution） 一个公司有多家工厂生产商品；有配送中心临时存储商品，还有零售店销售商品。公司必须通过配送中心定期地将商品从工厂配送至零售店，其中使用不同容量和单位配送成本的配送渠道。是否能将商品由仓库送至零售店，并使得各处满足供需关系呢？达到这一目的的最小成本是多少？图22-1描述了一个配送问题的例子。

图22-2展示了一个运输（transportation）问题，它是商品配送问题的一个特例。在这个问题中，没有配送中心和渠道容量的限制。这个版本的问题自身很重要，并且意义重大（我们将在22.7中讨论）。这不仅是由于其存在重要的直接应用，而且还因为它根本不是一个“特例”。实际上，它在难度上等价于该问题的一般形式。

通信（Communication） 通信网络有一组在服务器间传输消息的需求，这些服务器由信道（抽象线路）相连接，而信道能以不同的速率传送信息。在网络中的两个特定服务器之间，信息所能传输的最大速率是多少？如果信道关联有成本，那么以某个小于最大速率的给定速率发送信息的最便宜的方法是什么？

交通流量（Traffic flow） 一个市政府需要制定一个计划疏散在紧急情况下城市中的人群。如果假设可以控制交通流量来达到最小时间，那么城市人群疏散所需的最少时间是多

少？交通计划的制定者也可以提出确定哪些新的道路、桥梁或隧道可以缓解上班高峰或假期－周末交通之类的问题。

在匹配（matching）问题中，网络表示连接各对顶点的可能方式，我们的目标是从中（按照某种特定的准则）选择某些连接，且任何顶点不能选择两次。换句话说，所选出的边集定义了一种将顶点彼此配对的方法。我们可以把学生与大学相匹配、求职者与职位相匹配、课程与学校学时相匹配，或者是议员与议会坐席相匹配。在以上各种情况中，对于所寻求的匹配，可以想象出许多定义其特征的准则。

供应点		渠道	单位配送成本	最大容量
0:	4	0-3:	2	2
1:	4	0-7:	1	3
2:	6	1-4:	5	5
配送点		2-4:	3	4
3		2-9:	1	4
4		3-5:	3	2
5		3-6:	4	1
6		4-5:	2	5
需求点		4-6:	1	4
7:	7	5-7:	6	6
8:	3	5-8:	3	4
9:	4	6-9:	4	3

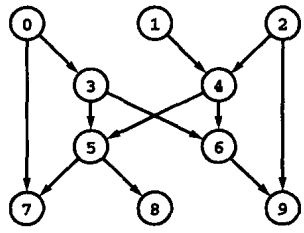


图 22-1 配送问题的例子

在这个配送问题的实例中，有3个供应顶点（0至2），4个配送顶点（3至6），3个需求顶点（7至9），以及12个配送渠道。每个供应顶点对应一个生产率；每个需求顶点对应一个消费率；而每条渠道有一个最大容量以及单位配送成本。这个问题是通过渠道（不超过容量限制）配送物料使成本最小，并使离开每个供应顶点的物料的速率等于该顶点的生产率；物料到达每个需求顶点的速率等于该顶点的消费率；并且物料到达每个配送顶点的总速率等于物料离开该顶点的总速率。

供应点		渠道	单位配送成本
0:	3	0-6:	2
1:	4	0-7:	1
2:	6	0-8:	5
3:	3	1-6:	3
4:	2	1-5:	1
需求点		2-8:	3
5:	6	2-9:	4
6:	6	3-6:	2
7:	7	3-7:	1
8:	3	4-9:	6
9:	4	4-5:	3

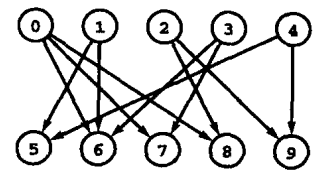


图 22-2 运输问题

这个运输问题与配送问题类似，但无渠道容量限制，也没有配送顶点。在这个实例中，我们有5个供应顶点（0至4），5个需求顶点（5至9），以及12个渠道。问题是求配送物料的最小成本，使得各处的供应量和需求量正好相等。明确地说，我们要求对边上赋以权值（配送率），使得出边上的权值之和等于每个供应顶点上的供应量；入边上的权值之和等于每个需求顶点上的需求量；并使所有这些分配的总成本达到最小（所有边上权值乘以成本的和）。

工作安排（Job placement） 工作安排服务为一组学生和一组公司安排相应的面试；这些面试结果会提供一组工作机会。假设一个面试后有一个工作机会表示学生与公司工作之间相互感兴趣，大家最感兴趣的是使公司工作安排的数目最大化。图22-3是一个例子，说明这个问题可以很复杂。

最小距离点匹配（Minimum-distance point matching） 已知两个N个点的集合，找N条线段的一个集合，每条线段的两个端点分别来自这两个集合，并使线段的长度之和达到最小。这个纯几何问题可用于雷达跟踪系统。雷达的每次扫描给出表示飞机的一个点集。我们假设飞机之间有足够的空间，求解这个问题是将各个飞机在扫描中的位置与其在下一次扫描中的位置关联起来，这样就得到了所有飞机的路径。其他的数据采样应用也可以基于这个框架考虑。

在分割问题中，如图22-4所示，我们去掉一些边把网络分割成两个或多个部分。分割问题与我们在第18章中最早讨论的图的连通性等基本问题直接相关。本章讨论一个中心定理，它显示出分割问题与流问题有着令人惊奇的关联，大大扩展了网络流算法的应用范围。

网络可靠性（Network reliability） 可以把一个电话网络看成通过交换机连接电话的一组线路组成。对于任何两个给定的电话，可能存在通过骨干线路连接的一条开关路径。问题是要使任何一对交换机保持连通，能够分割的最少骨干线路数是多少？

分割补给线（Cutting supply line） 战争中的国家会将补给物资沿着互连的高速公路系统从仓库转移至军队。敌方可能通过炸毁公路来切断军队的补给，破坏公路所需的炸弹数与路宽成正比。问题是为了导致没有军队得到补给，敌方搞破坏所需的最小炸弹数是多少？

以上所述的每个应用都直接引出很多的相关问题，另外，还有其他相关模型，比如说我们在第21章中所考虑的作业调度问题。本章中会考虑很多的例子，但也只是对直接有关的一小部分重要问题进行处理。

本章所讨论的网络流模型非常重要，这不仅在于它为我们提供了两个可简单表述的问题，很多实际问题都可以归约到这两个问题，而且我们还有求解这两个问题的高效算法。应用的广度已导致开发了很多算法及实现。我们考虑的很多解决方法说明了对一般应用实现的需求与要求特定问题的高效算法之间的对立。网络流算法的研究非常吸引人，因为它使我们相当接近达到这两个目标的简洁而精巧的实现。

我们在网络流模型内考虑两个特殊的问题：最大流（maxflow）问题和最小成本流（mincost-flow）问题。我们讨论求解问题的这些模型、第21章中的最短路径模型、第8部分的线性规划（LP）模型，以及大量特定问题的模型（包括刚才所讨论的那些模型）之间的特定关系。

Alice	Adobe
Adobe	Alice
Apple	Bob
HP	Dave
Bob	Apple
Adobe	Alice
Apple	Bob
Yahoo	Dave
Carol	HP
HP	Alice
IBM	Carol
Sun	Frank
Dave	IBM
Adobe	Carol
Apple	Eliza
Eliza	Sun
IBM	Carol
Sun	Eliza
Yahoo	Frank
Frank	Yahoo
HP	Bob
Sun	Eliza
Yahoo	Frank

图22-3 工作安排

假设有6个学生，每个学生需要工作，有6个公司，每个公司需要雇佣一个学生。用两个列表（一列按照学生名排序，另一列按照公司名排序）表示学生与工作之间的相互兴趣。问题是能否找到这样一种方式，使学生和工作匹配，满足每个工作都有人应聘，每个学生都找到一个工作。如果不能，可以确定的工作的最大数是多少？

乍一看，很多问题与网络流模型完全不同。确定一个给定问题与已知问题的关系，常常对于开发该问题的一个解决方案是最重要的一步。此外，这一步通常也很重要，因为如同图算法一样，在试图开发实现之前，我们必须理解容易问题和难解问题之间的细微的界限。本章中我们将要考虑的基础结构和问题之间的关系，为我们讨论这些问题提供了很有用的环境。

在第17章开始讨论的大致分类中，我们在本章考察的算法表明了网络流算法要归为“容易问题”，因为我们直接实现的算法可以保证它的运行时间与网络规模的多项式成正比。其他实现尽管不能保证在最坏情况下的多项式时间，但相当简洁、精巧。它们也已被证明可用于求解一大类的其他实际问题，诸如这里讨论的那些问题。我们详细考虑它们是因为它们的实用性。研究人员仍在寻求这些问题的更快速的算法，以便用于大型问题和在关键应用中节省成本。对于网络流问题，确保尽可能快的理想的最优算法还有待发现。

一方面，我们已经知道，已经归约到网络流问题的一些问题使用专门算法更容易求解。原则上，可以讨论实现和改进这些专门的算法。尽管这种方法在某些情况下很有效，但是可

解决很多问题（除了那些可以归约到网络流的问题）的高效算法仍然未知。即使专门算法是已知的，开发胜过优秀网络流代码的实现具有挑战性。此外，研究人员仍然在改进网络流算法，对于一个给定的实际问题，也有这种可能性，一个好的网络流算法胜过已知的专门算法。

另一方面，网络流问题是我们在第 8 部分所讨论的更一般线性规划（LP）问题的特例。尽管可以（人们常这样做）使用一个求解 LP 问题的算法来解决网络流问题，但是我们所考虑的网络流算法比那些求解 LP 问题的算法更简单，也更为高效。然而研究人员仍然在改进 LP 问题的算法，因此在用于实际的网络流问题时，一个对于 LP 问题的好算法仍有可能会优于本章中所考虑的所有算法。

网络流问题的经典解决方案与我们考察过的其他图算法密切相关。而且还能使用已开发的算法工具写出求解这些问题的非常简洁的程序。正如在很多其他情况所见到的，好算法和数据结构可以在本质上减少运行时间。开发经典通用算法的一个更好的实现仍在研究，新的方法仍有待发现。

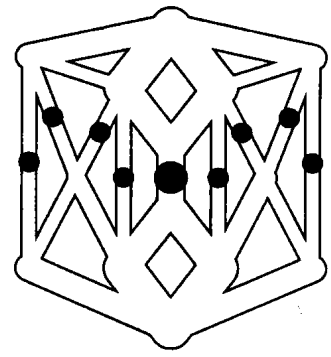


图 22-4 分割补给线

这个图表示了连接位于顶部的军队补给仓库和位于底部的军队的道路。黑点表示敌方切断军队补给线的轰炸计划。敌方的目标是使轰炸的成本达到最小（可以假设分割一条边的成本与其宽度成正比），我方军队的目标是设计一个公路网络使敌方的最小成本达到最大。这个模型对于改进通信网的可靠性和其他很多应用也很有用。

在 22.1 节中我们考察流网络（flow network）的基本性质，其中我们把网络的边上权值解释为容量（capacity），并考察流（flow）的性质，它是满足某些自然约束的第二组边权值。接下来，我们考察最大流（maxflow）问题，即计算出最适合于某种条件的一个流。在 22.2 节和 22.3 节中，我们考察求解最大流问题的两种方法，并考虑它的多种实现。我们考察的很多算法和数据结构与开发最大流问题的高效解直接相关。我们尚无解决最大流问题的可能的最佳算法，但是会讨论一些有用的特定实现。在 22.4 中，为了说明最大流问题的范围，我们考察不同的形式化描述，以及涉及其他问题的归约。

最大流算法和实现为我们讨论一个更为重要且通用的模型做好了准备，该模型称为“最小成本流问题（mincost flow problem）”。在这个模型中，我们首先指派成本（cost）（另一组边的权值），并定义流，然后找出有最小成本的最大流问题的一种解决方案。我们考察最小成本流问题的一个称为消环算法的经典通用解；然后在 22.6 节，给出称为网络单纯形算法（network simplex algorithm）的消环算法的一种特定实现。在 22.7 节，我们讨论归约到最小成本流问题的方法，其中包括以上提到的所有应用（以及其他应用）。

网络流算法作为本书的最后一章是由很多原因的。它们表示了对我们学习诸如链表、优先队列和通用图搜索算法的基本算法付出的一种回报。我们学习的 ADT 实现直接导致了网络流问题的简洁和高效的 ADT 实现。这些实现使我们拥有更高层次的求解问题的能力，而且在许多实际应用中直接可用。进而，研究其应用以及理解其局限性也可以为我们在第八部分考察更好的算法和更难的问题奠定基础。

22.1 流网络

为了描述网络流算法，我们先从一个理想的物理模型开始，在这个模型中的很多基本概念都是直观的。具体地说，想象有一组相互连接的大小可变的输油管道，有开关控制着接合

处流的方向，如图22-5中描述的示例。我们进一步假设网络只有一个源点（比如说一个油田），且只有一个汇点（比如说一个大的炼油厂），所有管道最终都连到这个汇点。在每个顶点上，当流进量与流出量相等时，所流的油达到平衡。我们使用同一单位度量流和管道容量（比如说每秒加仑数）。

如果每个开关具有如下性质：流进管道的总容量与流出管道的总容量相等，那么没有问题需要解决。我们只需把所有管道充满到其容量即可。否则，并不是所有管道都会充满，而是油流通过网络，并由接合处的开关设置控制，从而使流入每个接合处的油量等于流出的油量。但是接合处的这种局部平衡蕴含着网络作为整体的平衡，我们在性质22.1中将证明流入汇点的油量等于流出源点的量。此外，如在图22-6中所描述的，从源点到汇点的流量接合处的开关设置对于通过网络的流有显著影响。给定这些事实，我们对以下问题感兴趣：如何设置开关才能使从源点到汇点的油的流量达到最大？

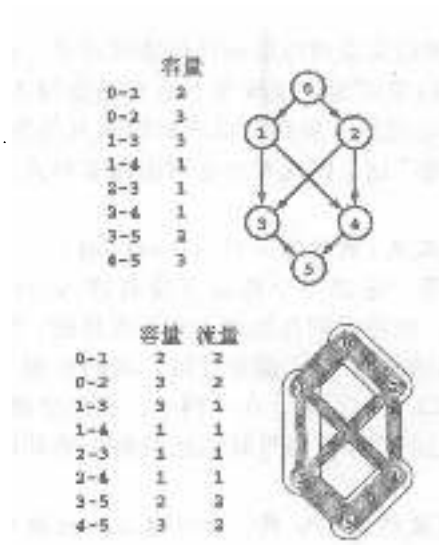


图22-5 网络流

流网络是一个加权网络，其中边上的权值为容量（上图）。我们的目标是计算受容量限制的另一组边权值（称为流）。下图说明了画流网络的一些规定。每条边的宽度与它的容量成正比；每条边上的流量用灰色阴影表示；流总是有方向的，从页面顶部的单源点到达底部的单个汇点方向向下；交叉处（如此例中的1-4和2-3交叉处）并不表示顶点，除非有标号。除了源点和汇点，在每个顶点上，流入量与流出量是相等的。例如，在顶点2上，流入量为2个单位（从顶点0开始），流出量为2个单位（其中一个单位流向顶点3，另一个单位流向顶点4）。

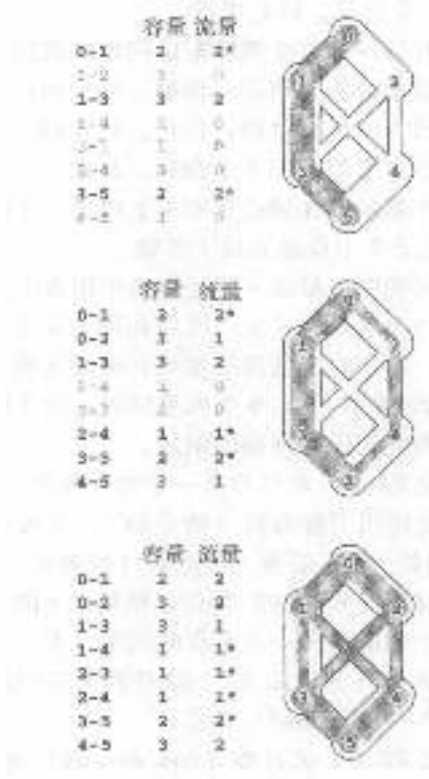


图22-6 控制网络中的流

我们可以沿着路径0-1-3-5打开开关来初始化网络中的流，这样可以处理2个单位的流（上图），沿着路径0-2-4-5打开开关得到网络中另外1个单位的流（中图），星号表示流已充满边。

因为0-1、2-4和3-5已充满，因而，从0到5不能直接得到更多的流。但如果我们改变顶点1处的开关，使有足够的流重定向以充满1-4，那么就在3-5中打开了足够的容量，从而增加0-2-3-5上的流，最终得到这个网络的一个最大流（下图）。

我们可以利用一个网络（加权有向图，在第 12 章定义）对这种情况建模。在这个网络中有一个源点和一个汇点。网络中的边对应着输油管道，顶点对应着带有开关的接合处，它们控制着流向每条出边的油量。边上的权值对应输油管道的容量。我们假设边是有向的，规定油只能在管道的一个方向流动。每个管道有一定量的流，这个流小于等于它的容量，每个顶点满足平衡条件，即流入量等于流出量。

这种流网络的抽象是一种求解问题的有用模型。它的直接应用很广泛，也有很多间接应用。我们有时借助于油流过管道的思路来直观地设想基本思路，但我们的讨论同样可以很好地应用到通过配送渠道移动商品的问题，以及很多其他情况。

该模型直接应用到配送情况：我们将流值解释为流的速率，因此，流网络可以用完全类似于油流的方式来描述商品流。例如，我们可以把图 22-5 中的流做如下解释：它指定了我们应该在每个时间单位从 0 到 1 和从 0 到 2 发送 2 个商品。每个单位时间从 1 到 3 和从 1 到 4 发送 1 个商品，以此类推。

还有另一种方法解释配送问题的流模型，把流值解释为商品量，因而，一个流网络描述了商品以前的传输情况。例如，我们可以把图 22-5 中的流用一个 3 步的过程来解释 4 个商品从 0 到 5 的传输过程：首先，从 0 到 1 发送出了 2 个商品，从 0 到 2 发送出了 2 个商品，在一些顶点上均留下 2 个商品。其次，从 1 到 3、从 1 到 4、从 2 到 3 以及从 2 到 4 分别发送出了 1 个商品，在顶点 3 和 4 上留下 2 个商品。第三，从 3 到 5 发送出了 2 个商品，从 4 到 5 发送出了 2 个商品完成了传输。

如同我们在最短 - 路径算法中距离的使用，在方便时完全可以放弃任何物理意义，因为我们所讨论的所有定义、性质和算法完全是基于一种抽象模型。该模型不必遵循物理法则。实际上，我们对网络流模型感兴趣的主要原因是，通过归约，我们可以求解很多其他问题。正如在 22.4 节和 22.6 节所见到的。由于模型应用范围广泛，因此有必要对前面非形式引入的术语和概念进行准确的阐述。

定义 22.1 *我们称有一个指定源点 s 和一个指定汇点 t 的网为 st 网*（st-network）。

在此使用了修饰符“指定的”，来表示 s 并没必要一定是一个源点（没有进入边的顶点），另外 t 也不必为一个汇点（没有发出边的顶点）。但将它们作为源点和汇点处理，因为我们的讨论（和算法）将会忽略指向 s 的边和从 t 指出的边。为了避免混淆，我们在例子中使用一个单源点和一个汇点的网络。更一般的情况在 22.4 中讨论。在 *st* 网中，我们分别称 s 为“源点”，t 为“汇点”，这是因为它们在网络中所起的作用。另外我们把网络中的其他顶点称为内部（internal）顶点。

定义 22.2 *流网络*（flow network）*是一个有正的边权值的 st 网，称此边上的权值为容量*（capacity）。*流网络中的一个流*（flow）*是一个非负边权值集合，称之为边流*（edge flow）。*且满足：边流不大于该边上的容量，进入每个内部顶点的总流量等于从此顶点流出的总流量。*

我们将进入一个顶点的总流量（该顶点的进入边上的流之和）称为该顶点的流入量（inflow），将流出一个顶点的总流量（该顶点的发出边上的流之和）称为该顶点的流出量（outflow）。按照约定，可以把进入源点的边上的流和从汇点发出边上的流设置为 0。并根据性质 22.1，可以证明从源点的流出量等于汇点的流入量。称这个量为该网的值。有了这些定义，我们对基本问题的形式阐述就很简单了。

最大流 给定一个 *st* 网，找出一个流，使从 s 到 t 的任何其他流不会比此流值大。为简洁起见，我们称这样的流为最大流（maxflow），称找一个网络中的最大流问题为最大流问题

(maxflow problem)。在某些应用中，可能只需知道最大流值，但一般而言我们想要知道达到该流值的一个流（边流值）。

我们很快就会想到这个问题的一些变型问题。能否允许多源点和多汇点吗？是否能够处理无源点或无汇点的情况？允许在边的两个方向有流吗？除了/取代对边上有对流的容量限制，顶点是否有流的容量限制呢？如同图算法中的典型情况，很难将很容易处理的限制和具有深刻意义的限制加以区分，它是一个挑战性任务。我们研究这种挑战，在考虑解决基本问题的算法之后，将在22.2节和22.3节给出看起来本质不同的大量问题归约为最大流的例子。

流的显著特征是局部平衡条件，在每个内部顶点上的流入量等于流出量。对于容量没有限制；实际上，流入边上的总容量和流出边上的总容量之间的不平衡性就表征了最大流问题。平衡限制必须在每个内部顶点上成立。因而，这个局部性质也确定了网络中全局的移动。尽管这个思路很直观，但仍需要证明：

性质22.1 任何一个 *st* 网具有以下性质：从 s 的流出量等于到 t 的流入量。

证明 （我们使用术语 *st* 流来表示“一个 *st* 网中的流”）。增加一条从虚拟顶点进入 s 的边，将从 s 的流出量作为该边的流和容量，增加从 t 到另一虚拟顶点的边，将到 t 的流入量作为该边的流和容量，从而扩展网络。然后，我们可以用归纳法证明更一般的性质：对任意顶点集（不包括虚拟顶点）流入量等于流出量。

由局部平衡性，这个性质对于任意单个顶点成立。现在，假设结论对于给定的顶点集 S 成立，我们向 S 添加一个顶点 v，得到集合 $S' = S \cup \{v\}$。现计算 S' 的流入量和流出量。注意到从 v 到 S 中某个顶点的每条边减少的流出量（从 v）与（到 S）的流入量的减少量相同；从 S 中某个顶点到 v 的每条边减少的流入量（到 v）与（从 S）的流出量的减少量相同；而且，所有其他边为 S' 提供了流入量或流出量当且仅当它们对 S 或 v 也有此作用。因此，S' 的流入量和流出量相等。流的值等于 v 与 S 的流值之和减去连接 v 与 S 中顶点的边上（包括两个方向）的流的和。

把这一性质应用到网络中的所有顶点集，我们发现源点从它相关虚拟顶点的流入量（等于源点的流出量）等于汇点到它相关虚拟顶点的流出量（等于汇点的流入量）。 ■

推论 两个顶点集的并集，其流值等于两个顶点集的流值之和减去将一个集合中的顶点与另一个集合中的顶点连接的边上的权值之和。

证明 在上述对于集合 S 与顶点 v 的证明中，如果用集合 T（与集合 S 不相交）代替 v 结论仍然成立。图22-7给出了说明这个性质的一个例子。 ■

在性质22.1的证明中可以无需虚拟顶点，我们可以用一条从 s 到 t 的边对任何流网络进行扩展，其中流和容量等于网的值，且对任何扩展过的网络中的结点集，其流入量等于流出量。称这样的流为循环流（circulation）。

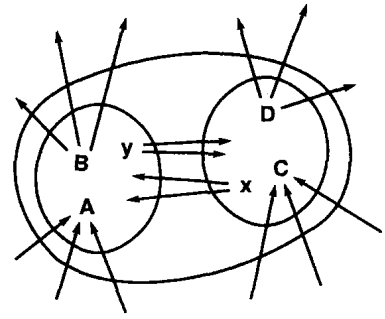


图22-7 流平衡

该图说明了当合并两个顶点集时仍保持流平衡。两个较小的图表示两个不相交的顶点集，字母表示所示边集中的流：A 是从右边集合外面进入左边集合中的流量，x 为从右边的集合进入左边的集合中的流量，等等。现在，如果这两个集合中的流平衡，那么，对于左边的集合必有：

$$A + x = B + y$$

对于右边的集合，必有：

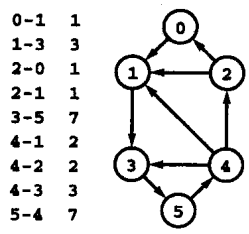
$$C + y = D + x$$

将上面两式相加，并消去 $x + y$ 项，可得：

$$A + C = B + D$$

即对于两个集合的并集，其流入量等于流出量。

而且这个构造过程表明，最大流问题可归约到找出一个循环流，使沿着给定边上的流达到最大。这种形式化简化了我们对于某些情况的讨论。例如，可以得到把流表示为一个环集合的另一种有趣表示。如图 22-8 所示。



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

图 22-8　循环流表示

此图表明了左边的循环流可以分解为 4 个循环 1-3-5-4-1，0-1-3-5-4-2-0，1-3-5-4-2-1，3-5-4-3，它们的权值分别为 2，1，1 和 3。每个环的边出现在各自的列中，对每个环中出现的每条边上的权值求和（穿过其各自的行），即得在循环流中的权值。

给定一个环的集合，以及每个环上的一个流值，通过每个环并向每条边增加指示的流值很容易计算对应的循环流。相反的性质更令人惊讶：我们可以找出与任何给定的循环流等价的一个环集合（每个环一个流值）。

性质 22.2（流分解定理）　任何循环流可以表示为至多有 E 个有向环的集合上的流。

证明　由一个简单算法即可得出这个结论。只要还有一个边上有流，就重复以下过程：从任何有流的一条边开始，沿着离开那条边的目的顶点的任何有流的一条边，继续该过程直到遇到一个已经被访问过的顶点（检测出环）为止。沿环返回找出具有最小流的一条边；接着将环中的每一条边的流减去此量。该过程每次迭代都会使至少一条边上的流为 0，因而至多有 E 个环。 ■

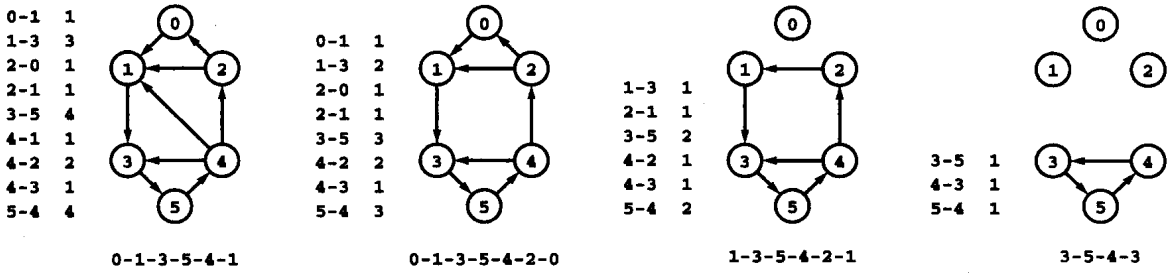


图 22-9　循环流分解过程

要把任何一个循环流分解为环的集合，我们对以下过程迭代：沿着任一路径行进，直到再次遇到一个结点，接着找出指示环上的最小权值，然后从环中的每一条边上减去这个最小权值，并去掉环中权值为 0 的那些边。例如，首次迭代的路径是 0-1-3-5-4-1，环是 1-3-5-4-1，接着从该环中每条边上的权值减去 1，由于 4-1 的权值为 0，故被去掉。第二次迭代中，去掉 0-1 和 2-0；第三次迭代中，去掉 1-3、4-2 和 2-1；第四次迭代中，去掉 3-5、5-4 和 4-3。

图 22-9 说明了证明中所描述的过程。对于 st 流，把这个性质应用到添加从 t 到 s 的边所创建的循环流得到下述结果，任何 st 流可以表示为一个至多有 E 条有向路径的集合上的流，每条路径要么是从 s 到 t 的一条路径，要么是一个环。

推论　任何 st 网都有一个最大流，满足非零流值所导出的子图是一个环。

证明　不含 $t-s$ 的环不会对流值有所贡献，因而我们可以在这样的环中把流变为 0，而不改变流值。 ■

推论 任何 st 网都有一个最大流，它能表示为从 s 到 t 的至多 E 个有向路径集合上的流。

证明 直接得证。 ■

这种表示为深刻理解流的本质提供了很好的基础。它在最大流的算法设计和分析中非常有用。

一方面，我们可能考虑更一般形式的最大流问题，允许多个源点和汇点。这样做可以使算法的应用范围更广。另一方面，我们也可以考虑最大流问题的特例，比如只关注无环网。这样简化之后，使问题比较容易求解。事实上，如我们在 22.4 节中看到的，这些变形在难度上与我们正考虑的版本是等价的。于是，在第一种情况下，我们可以把我们的算法和实现应用到更大的范围；第二种情况，我们不期望有更简单的解决方案。在图示中使用了无环网，因为它若有一个隐含的流向（向下），就更容易理解，但是我们的实现允许网中有环。

程序 22.1 流网络 ADT 实现

这个流网络的 ADT 实现只是扩展了第 17 章的无向图的邻接表实现，在每个表结点中添加一个表示边容量的 cap 域，一个表示边流的 flow 域，以及链接每条边的两种表示的 dup 域。假设对另一个图处理代码的做了相应扩展（例如，Edge 类型和 EDGE 构造器必定含有 cap 域和 flow 域）。flow 域初始化为 0；我们的网络流实现作为 ADT 函数用来计算它们的相应值。

```
#include <stdlib.h>
#include "GRAPH.h"
typedef struct node *link;
struct node
  { int v; int cap; int flow; link dup; link next;};
struct graph
  { int V; int E; link *adj; };
link NEW(int v, int cap, int flow, link next)
  { link x = malloc(sizeof *x);
    x->v = v; x->cap = cap; x->flow = flow;
    x->next = next;
    return x;
  }
Graph GRAPHinit(int V)
  { int i;
    Graph G = malloc(sizeof *G);
    G->adj = malloc(V*sizeof(link));
    G->V = V; G->E = 0;
    for (i = 0; i < V; i++) G->adj[i] = NULL;
    return G;
  }
void GRAPHinsertE(Graph G, Edge e)
  { int v = e.v, w = e.w;
    G->adj[v] = NEW(w, e.cap, e.flow, G->adj[v]);
    G->adj[w] = NEW(v, -e.cap, -e.flow, G->adj[w]);
    G->adj[v]->dup = G->adj[w];
    G->adj[w]->dup = G->adj[v];
    G->E++;
  }
```

为了实现最大流算法，我们使用流网络的标准自然表示，只需把前面几章中使用的邻接矩阵或邻接表表示进行扩展。不是像在第 20 章和第 21 章中那样只使用单个权值，我们在每条边上关联两个权值，cap（容量）和 flow（流）。即使网络是有向图，我们考察的算法也需要遍历两个方向的边，因而我们使用像在无向图中那样的表示：如果从 x 到 y 有一条边，其上容量为 c，流为 f，我们也从 y 到 x 保存一条边，其上容量为 $-c$，流为 $-f$。图 22-10 显示了图 22-5 中示例的邻接矩阵表示。

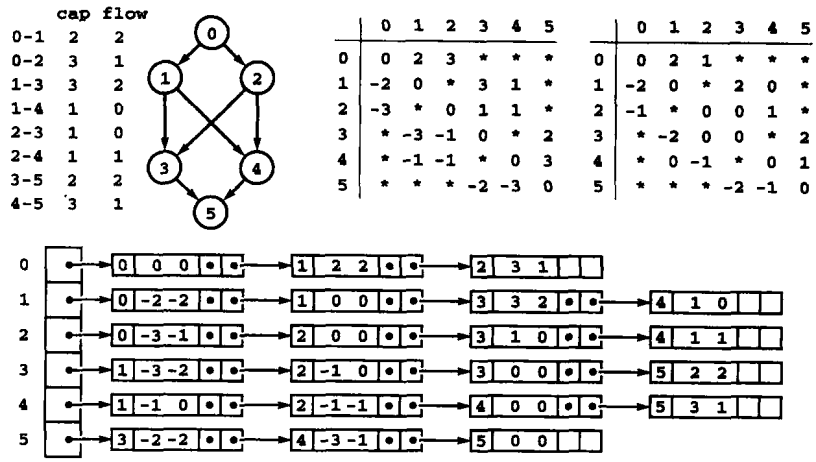


图 22-10　流网络的表示

在流网络的邻接数组表示中，我们使用平行数组（上图）：一个数组表示容量（上图左），另一个表示流值（上图右）。我们把 c 放在 s 行 t 列处，把 $-c$ 放在 t 行 s 列处来表示 s-t 边的容量 c。在右边的数组中我们使用相同的模式表示流值。这种方式包括了两种表示，简化了代码，因为我们的算法需要遍历两个方向上的边。

在邻接表表示中（下图）我们使用一种类似的方法。为了使它能够在常量时间内改变流值，需要连接每条边的两种表示的链接。这些链接在此图中省略。

程序 22.2　流的检查和流值计算

如果在某个内部结点上进入的流不等于出去的流或者某个流值为负，那么这个 ADT 函数返回 0；否则该函数返回流值。尽管在性质 22.1 中数学上已有保证，但作为程序员还应该检查是否从源点出来的流等于进入汇点的流。

```
static int flowV(Graph G, int v)
  { link t; int x = 0;
    for (t = G->adj[v]; t != NULL; t = t->next)
      x += t->flow;
    return x;
  }
int GRAPHflow(Graph G, int s, int t)
  { int v, val = flowV(G, s);
    for (v = 0; v < G->V; v++)
      if ((v != s) && (v != t))
        if (flowV(G, v) != 0) return 0;
    if (val + flowV(G, t) != 0) return 0;
    if (val <= 0) return 0;
    return val;
  }
```

在第 20 章和第 21 章的网络表示中，我们使用约定：权值为 0 到 1 之间的实数。在这一

章里，我们假设权值（容量和流）都是 m 位的整数（$0 \sim 2^m - 1$ 之间）。这样做有两个主要原因。第一，通常需要测试权值线性组合是否相等，采用浮点表示法不方便这样做。第二，算法的运行时间可能依赖于权的相对值，而参数 $M = 2^m$ 使我们很方便界定权值的界限。例如，最大权值和最小非零权值之比小于 M。对于解决这些问题（例如，见练习 20.8），使用整数权值只是多种可选方案中的一种。

在大多数程序中，我们使用邻接表表示图。因为对于稀疏网它是最有效的，同时在应用中很常用。扩展第 21 章中的加权网的表示，把容量和流保存在对应边的表结点中。和邻接矩阵表示一样，我们把每条边的前向和后向表示保存起来，方法类似于第 17 章中的无向图表示，但把容量和流放在后向表示中。为了避免过度的遍历链表，我们还维护着连接两个表的结点的一些链接，这些结点表示每条边。因而，当我们改变一个表中的流时，就要更新另一个表中的流。图 22-10 中显示了图 22-5 中示例的一个邻接表表示。所添加的域和链接使得数据结构看起来更复杂：程序 22.1 的实现与它所对应的程序 17.4 的无向图的代码只有几行代码不同。

如在前面一些章中那样，我们常常使用类似于程序 17.1 中定义的 EDGE 构造器函数，但做了一些修改，包含了流和容量。

我们有时称边有无限容量，或等价地说，不限容量（uncapacitated）。那意味着我们并不能比较这些边上的流和容量，或者我们可能使用一个观察哨来保证比任何流都大的一个值。

程序 22.2 是一个 ADT 函数，它用来检查是否在每个结点上的流满足平衡性条件，如果满足则返回那个流的值。一般情况下，在最大流算法的最后一步，我们可能包含对这个函数的一个调用。一种谨慎的做法是检查没有一条边上的流会超过那条边上的容量，并且数据结构能够做到内部一致（见练习 22.13）。

练习

▷ **22.1** 在图 22-11 所显示的流网络中，找出两个不同的最大流。

22.2 假设容量为小于 M 的正整数，对于一个 V 个顶点、E 条边的 *st* 网，其最大可能流值是多少？给出依赖于是否允许平行边存在的两种答案。

▷ **22.3** 对于汇点被删除后所构成一棵树这样的网，给出求解最大流问题的一个算法。

○ **22.4** 给出带有 E 条边且有循环流的一类网，满足在性质 22.2 的证明中所描述的过程产生 E 个环。

22.5 修改程序 22.1 把容量和流表示为 0 和 1 之间的实数，且小数点后有 d 位数字，d 是一个固定常数。

	cap
0-1	2
0-2	3
0-3	2
1-2	1
1-3	1
1-4	1
2-4	1
2-5	2
3-4	2
3-5	3
4-5	2

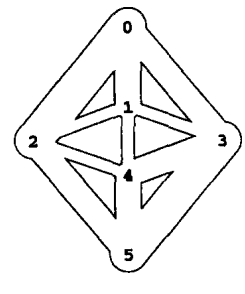


图 22-11 流网络的练习

这个流网络是贯穿本章的几个练习的主题。

▷ **22.6** 给出图 22-11 所示的流网络的邻接矩阵和邻接表表示，风格如图 22-10。

▷ **22.7** 编写一个独立于表示的程序，通过从标准输入读入边（0 到 $V-1$ 之间的整数对）和整数容量值来构建一个流网络。假设容量上界 M 小于 2^{20}。

22.8 扩展练习 22.7 中的解，使用符号名而不是整数来引用顶点（见程序 17.10）。

○ **22.9** 找出一个大型在线网络，你可用它作为测试针对实际数据的流算法的工具。可以包括运输网（公路、铁路或航空）、通信网（电话或计算机连接）或配送网。如果未提供容量，设计一个合理模型把它们添加进去。编写一个程序，使用程序 22.1 中的 ADT 来为你的

数据构建流网络，在此也可能使用练习 22.8 中的解决方案。如果允许，另开发一些 ADT 函数来清除数据，如练习 17.33 ~ 17.35 中所述。

22.10 基于程序 17.6，实现稀疏网的流网络 ADT，容量介于在 0 和 2^{20} 之间。其中包括根据程序 17.7 的一个随机网络生成器。使用另一个 ADT 产生容量，并开发两个实现：一个实现用于产生均匀分布的容量，另一个用于产生高斯分布的容量。对于这两种权值分布，用经过精心选择的集合 V 和 E，实现产生随机网的客户程序，从而可以利用这些客户程序对由各种边权值分布所得的图进行测试。

22.11 使用邻接矩阵表示，实现稠密网的流网络 ADT，容量介于在 0 和 2^{20} 之间。其中包括根据程序 17.8 的一个随机网络生成器和根据练习 22.10 描述的边容量生成器。对于这两种权值分布，用经过精心选择的集合 V 和 E，编写一个产生随机网的客户程序，从而可以利用这些客户程序对由这些模型所得的图进行测试。

• 22.12 编写一个程序产生平面上的 V 个随机点，然后构建一个流网络，其中的边（两个方向）连接给定距离 d 内的所有点对（见程序 3.20），使用练习 22.10 所描述的任何一个随机模型来设置每条边的容量。确定如何设置 d 以使边的期望数为 E。

22.13 修改程序 22.2 来检查所有边上的流和容量是否都为正值或都为负值；u-v 上的流与 v-u 上的流之和是否为 0；对于容量是否也有这个结论；对应每条边是否只有两个相互链接的链表结点。

22.14 对于邻接矩阵表示，编写一个类似练习 22.13 中描述的 ADT 函数。

▷ 22.15 找出图 22-12 所示网络中的所有最大流。给出各个最大流的环表示。

22.16 编写一个邻接表表示的 ADT 函数，能够读取流值和环（每行一个环，按照图 22-8 中描述的格式），且计算出所对应的流。

22.17 编写一个邻接表表示的 ADT 函数，使用性质 22.2 的证明所描述的方法找出一个网络流的环表示。并输出流值和环（每行一个环，按照图 22-8 中描述的格式）。

○ 22.18 编写一个邻接表表示的 ADT 函数，使它删除一个 st 流中的环。

○ 22.19 编写一个程序，对于任何给定的有向图，为不包括汇点及源点的每条边赋予一个整数流，使得该有向图是一个循环流的流网络。

	cap
0-1	2
0-2	3
0-3	2
1-3	1
1-4	1
2-1	1
2-5	2
3-4	2
3-5	3
4-2	1
4-5	2

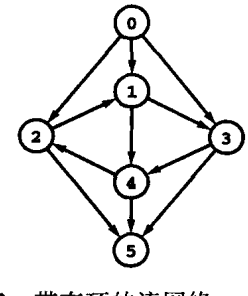


图 22-12 带有环的流网络

这个流网络与图 22-11 中描述的流网络类似，但边上的两个方向反向，因而有两个环。它也是贯穿本章的几个练习的主题。

○ 22.20 假设流表示要用卡车在城市间运输的商品，边 u-v 上的流表示从城市 u 到城市 v 在给定的某天内运输的量。编写一个 ADT 函数，输出卡车司机的日常订单，通知司机装货和卸货的地点和数量。假设每天对卡车司机的供应没有限制，而且对于一个给定的配送点，只有货物到达后，才会配送货物。

22.2 增大路径最大流算法

L. R. Ford 和 D. R. Fulkerson 于 1962 年提出了解决最大流问题的一种有效方法。它是一种沿着从源点到汇点的路径逐渐增加流的一般性方法。在经典的文献中称它为 Ford-Fulkerson 法；现在都广泛使用术语*增大路径法*（augmenting-path method）。

考虑穿越 st 网的从源点到汇点的一条有向路径（不必是简单路径）。设 x 是这条路径上

的边的未用容量。我们可以增加的网络流值至少为 x，在这条路径的所有边上增加这个数量的流。重复这个过程，首先得到网络中计算的流，找出另一条路径，沿着这条路径增加流，继续这一过程，直到从源点到汇点的所有路径至少有一条满边（从而不能再用这种方式增加流）。这个算法可以计算出某些情况下的最大流，有些情况该算法不能工作。图 22-6 说明了这个算法会失效的一种情况。

为了改进算法使它对任意情况都能找出最大流，我们考虑增大流的更一般的方法，沿着穿越网络的基本无向图的从源点到汇点一条路径增大流。在任何一条这样路径上的边要么是前向（forward）边，随着流行进（遍历从源点到汇点的路径时，遍历从其源顶点到其目的顶点的边），要么是后向（backward）边，沿着流逆行（遍历从源点到汇点的路径时，遍历从其目的顶点到其源顶点的边）。因而，对于没有任何前向满边和后向空边的路径，我们可以通过增加前向边上的流和减少后向边上的流来增加网络中的流量。所增加的流量受到前向边上未用容量最小值和后向边上流的限制。图 22-13 描述了一个例子。在新流中，这条路径上至少有一条前向边变成满边或至少有一条后项边变成空边。

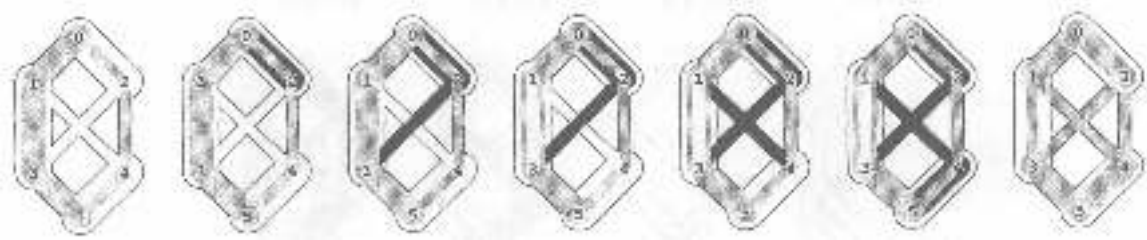


图 22-13 沿着路径增大流

这一序列显示了沿着前向边和后向边的路径在网络中增大流的过程。从左边的描述的流开始，并从左向右读取，先增加 0-2 上的流，再增加 2-3 上的流（增加的流用黑色显示）。然后减少 1-3 上的流（用白色显示），并使此流转到 1-4，然后到 4-5，最后得到右图的流。

刚才描述的这个过程是经典的 Ford-Fulkerson 最大流算法（增大路径法）。总结如下：

> 从任何一个零流开始，沿着从源点到汇点的任何一条路径上的未满前向边或空的后向边增大此流，继续这个过程直到网络中不存在这样的路径。

显然，不论我们如何选择路径，这种方法总是能够找出最大流。就像在 20.1 节中讨论的 MST 方法和 21.7 中讨论的 Bellman-Ford 最短路径方法，它是一种非常有用的通用方法，因为它确立了一类更为特定算法的正确性。我们可以使用任何方法来选择路径。

图 22-14 说明了几种不同的增大路径序列，都能得到示例网的最大路径。本节稍后，我们考察几种计算增大路径序列的算法，所有这些方法都可得到最大流。算法不同之处在于它们计算的增大路径数目、路径长度以及找出每条路径的开销不同。但是它们都实现了 Ford-Fulkerson 算法，并找出了最大流。

为了说明 Ford-Fulkerson 算法的任何实际实现所计算出的流都有最大流值，我们表明这个事实等价于一个称为最大流 - 最小割定理（maxflow-mincut theorem）的重要事实。理解这个定理是理解最大流算法的关键一步。顾名思义，该定理是基于网络中流和割之间的直接关系的，因而我们首先定义有关割（cut）的术语。

回忆 20.1 节中的内容，图中的割是把顶点分成两个不相交集合的一个划分，并且交叉边（crossing edge）就是连接这两个集合中顶点的边。对于流网络，我们完善这些定义，如下（见图 22-15）。

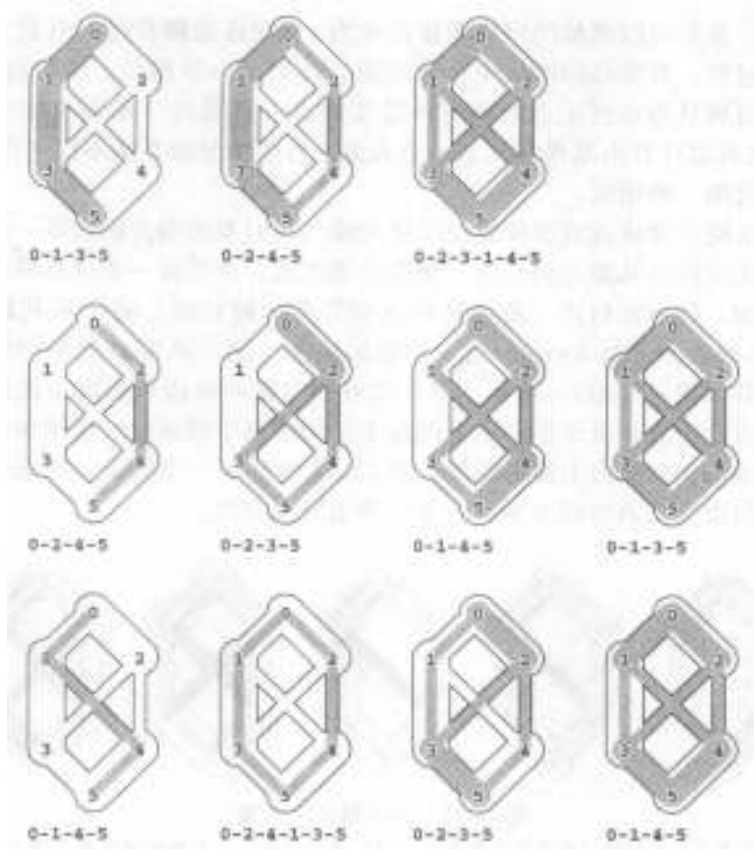


图 22-14　增大路径序列

在这三个例子中，我们沿着不同增大路径序列来增大流，直到没有增大路径存在。每种方法所得到的流是一个最大流。网络流理论中的关键经典定理指出：不论使用哪一个路径序列，我们都可以得到任何一个网的最大流（见性质 22.5）。

定义 22.3　*st* 割是一个将顶点 *s* 放在其中一个集合，将顶点 *t* 放在另一个集合中的割。

与 *st* 割对应的每条交叉边要么是一条 *st* 边，它连接包含 *s* 的集合中的一个顶点与包含 *t* 的集合中的一个顶点，要么是一条 *ts* 边，边的方向相反。我们有时把交叉边的集合称为割集（cut set）。流网络中的一个 *st* 割的容量（capacity）就是该割的 *st* 边的容量之和，穿越（flow across）一个 *st* 割的流为此割中的 *st* 边的流之和与为此割中的 *ts* 边的流之和的差。

删除割集把一个连通图划分为两个连通分量，使得不存在路径连接这两个集合中的顶点。删除网的一个 *st* 割中的所有边，致使在基本无向图中不存在连接 *s* 到 *t* 的路径，但恢复任何一条边就可能形成这样一条路径。

对于本章开始提到的那些问题，割是合适的抽象，其中流网络描述了从仓库到部队补给的转移。为了以最经济的方式完全切断补给，敌方可能会解决以下问题：

最小割（minmum cut）　给定一个 *st* 网，找出一个使

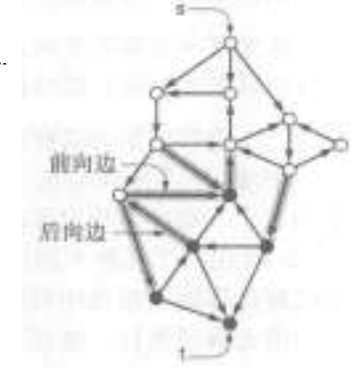


图 22-15　*st* 割的术语

一个 *st* 网有一个源点 *s* 和一个汇点 *t*。*st* 割是把顶点分成包含源点的集合（白色顶点）及包含汇点的集合（黑色顶点）的一个划分。称连接这两个集合中的顶点（其中一个集合用灰色强调）的边为割集。前向边从包含 *s* 的集合中的顶点到包含 *t* 的集合中的顶点；后向边的方向则相反。在如图所示的割集中，有 4 条前向边和 2 条后向边。

割的容量达到最小的 st 割。为简明起见，我们把这个割称为最小割（mincut），找出网络中最小割的问题称为最小割问题（mincut problem）。

最小割问题是对我们在18.6节所讨论的连通性问题的一个推广。在22.4节详细我们分析它们的特定关系。

在对最小割问题的阐述中，没有提到流的概念。这些定义似乎有些偏离我们讨论的增大路径算法。表面上看，计算一个最小割（边集）似乎是要比计算一个最大流（在所有边上分配权值）要容易。相反，本章的关键事实是最大流问题和最小割问题紧密相关。结合关于流和割的两个事实，增大路径法自身给出了一个证明。

性质22.3 *对于任何 st 流，穿越每个 st 割的流等于该流值。*

证明 这个性质是把我们在相关证明（见图22-7）中讨论的性质22.1的一个直接推广结果。增加与流值相等的流的边 t-s，使对任何顶点其流入量等于流出量。然后，对于任何 st 割，C_s是包含 s 的顶点集合，C_t是包含 t 的顶点集合，到 C_s的流入量为到 s 的流入量（流值）加上穿越该割的后向边上的流之和；从 C_s的流出量为穿越该割的前向边上的流之和。设置这两个量相等就确立了所需要的结果。■

性质22.4 *st 流的值都不超过任何 st 割的容量。*

证明 穿越一个割的流肯定不会超过那个割的容量，因而该结果由性质22.3直接而得。■

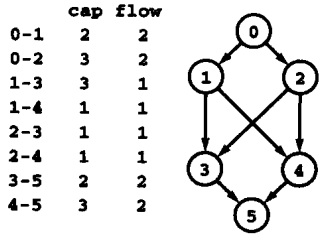
换句话说，割表示了网络中的瓶颈。在军事应用中，不能完全切断部队补给的敌方仍然能够确信补给流会受到任一给定割的容量的限制。我们肯定可以想象在这个应用中，建立一个割的成本与它的容量成正比，因而促使敌方找出最小割问题的一个解。更重要的是，这些事实还蕴含着不存在流值大于任一最小割的容量。

性质22.5 *（最大流－最小割定理）网络中的所有 st 流的最大值等于所有 st 割的最小容量。*

证明 只要展示一个流和一个割，满足流值等于割的容量就足够了。该流必须是一个最大流，因为任何其他流值都不超过此割的容量；而且此割必须是一个最小割，因为其他割的容量都不会小于这个流值（由性质22.4可知）。Ford-Fulkerson算法正是给出了这样一个流和割：算法终止时，找出图中从 s 到 t 的每条路径上的第一条前向满边或后向空边。令 C_s为不含前向满边或后向空边的无向路径中从 s 可达的所有顶点集合，令 C_t是其余顶点。那么，t 必定在 C_t中，因而（C_s，C_t）是一个 st 割，其割集完全由前向满边或后向空边组成。穿越此割的流等于此割的容量（因为前向边是满的，且后向边是空的），且等于网络流值（由性质22.3可知）。■

这个证明同时还明确地确立了Ford-Fulkerson算法可以找出一个最大流。无论选择何种方法来找出一条增大路径，也不论我们找出哪一条路径，我们总是结束在其流等于其容量的一个割上，而且等于网络流的值，因此，它必定是一个最大流。

Ford-Fulkerson算法的正确性还有另一个含义，那就是对于任何具有整数容量的流网络，存在一个最大流解，其中流都是整数。每条增大路径使流增加一个正整数（前向边中的最小未用容量，及后向边中的流，它们都总是正整数）。这个事实说明了我们仅关注整数容量和流的决定是正确的。即使是在容量都为整数时，也可能设计一个有非整数流的最大流（见练习22.25），但我们需要不考虑这样的流。这个限制是重要的：推广到允许容量和流为实数的情况，可以得到我们不希望的异常情况。例如，Ford-Fulkerson算法可能导致甚至不含最大流值的一条无限的增大路径序列（见第五部分参考文献）。



	cap	flow
0-1	2	2
0-2	3	2
1-3	3	1
1-4	1	1
2-3	1	1
2-4	1	1
3-5	2	2
4-5	3	2

0	1 2 3 4 5	0-1 0-2		5
0 1	2 3 4 5	0-2 1-3 1-4		7
0 2	1 3 4 5	0-1 2-3 2-4		4
0 3	1 2 4 5	0-1 0-2 3-5	1-3 2-3	6
0 4	1 2 3 5	0-1 0-2 4-5	1-4 2-4	8
0 1 2	3 4 5	1-3 1-4 2-3 2-4		6
0 1 3	2 4 5	0-2 1-4 3-5	2-3	6
0 1 4	2 3 5	0-2 1-3 4-5	2-4	9
0 2 3	1 4 5	0-1 2-4 3-5	1-3	5
0 2 4	1 3 5	0-1 2-3 4-5	1-4	6
0 3 4	1 2 5	0-1 0-2	1-3 1-4 2-3 2-4	5
0 1 2 3	4 5	1-4 2-4 3-5		4
0 1 2 4	3 5	1-3 4-5		6
0 2 3 4	1 5	0-1 3-5 4-5	1-3 1-4	7
0 1 2 3 4	5	3-5 4-5		5

图 22-16 所有 *st* 割

这个表在左边列出了网络中的所有 *st* 割、包含 *s* 的集合中的顶点、包含 *t* 的集合中的顶点、前向边、后向边以及容量（前向边容量之和）。对于任何流，穿越所有割的流（即前向边中的流减去后向边中的流）都相同。例如，对于左边网络中的流，穿越此割（分割 0 1 3 和 2 4 5）的流为 2 + 1 + 2（分别为 0-2、1-4 和 3-5 中的流）减 1（2-3 中的流），即为 4。这个计算结果 4 也可由网络中的其他割得到，而且这个流是一个最大流，因为它的值等于最小割的容量（见性质 22.5）。这个网络中有两个最小割。

通用的 Ford-Fulkerson 算法并没有指定找出一条增大路径的特定方法。也许找增大路径的最自然的一种方式是使用 18.8 节中的推广的图搜索策略。为此，我们先从以下定义开始：

定义 22.4 给定一个流网络和一个流，该流的残量网络（residual network）有着和原始网络一样的顶点，且对于原网络中的每条边，残量网络中都有一条或两条边，定义如下：对于原网络中的每条边 *u-v*，令 *f* 是流，*c* 是容量。如果 *f* 是正的，则在残量网络中包含容量为 *f* 的一条边 *v-u*；且如果 *f* 小于 *c*，在残量网络中包含容量为 *c-f* 的一条边 *u-v*。

如果 *u-v* 为空（*f* 等于 0），则在残量网络中有一条容量为 *c* 的边 *u-v*；如果 *u-v* 为满（*f* 等于 *c*），则在残量网络中有一条容量为 *f* 的边 *v-u*；如果 *u-v* 即不空也不满，则在残量网络中包含带有各自容量的 *u-v* 和 *v-u* 的边。

残量网络使我们可以使用任何广义图搜索方法来找出一条增大路径（见 18.8 节），因为残量网络中从源点到汇点的任意路径直接对应原网络中的一条增大路径。沿着这条路径增大流意味着在残量网络中进行改变：例如，在这条路径上至少有一条边变成满或空，因而在残量网络中至少有一条边改变方向或者消失。图 22-17 显示了一个例子的一条增大路径序列以及所对应的残量网络。

正如我们看到的，经典的图搜索方法仅在选择何种数据结构来存放未探索的边有所不同。对于最大流问题，选择一个数据结构直接对应实现找出增大路径的一种特殊策略。尤其是在我们考虑使用 FIFO 队列、优先队列、栈和随机队列进行实现时。

程序 22.3 是一种基于优先队列的实现，它包括了所有这些可能性，使用程序 21.1 图搜索实现的 PFS 的一个改进版本。这一实现可使我们在 Ford-Fulkerson 的几种不同的经典实现间做出选择，只要设置优先级就可以为边缘集实现不同数据结构。

理解程序 22.3 的关键是理解其中的两种方式，在这两种方式中，它利用了这样的事实：邻接表表示包含流网络的每条边的两种表示（见图 22-10）。首先，它不需要显式地构造残量网络：一个简单的宏就足够了。第二，它表示带有网络边的 PFS 搜索树，以支持流增大：在程序 21.1 中，当我们遍历边 *v-w* 时，设置 st[w] = v；在程序 22.3 中，当我们遍历边 *v-w* 时，此时在 v 的邻接表中有一个链接 u，使 u -> v = w，我们设置 st[w] = u。这使得我们可以访问流（用 st[w]-> flow 访问）和父结点（用 st[w]-> dup -> v）。

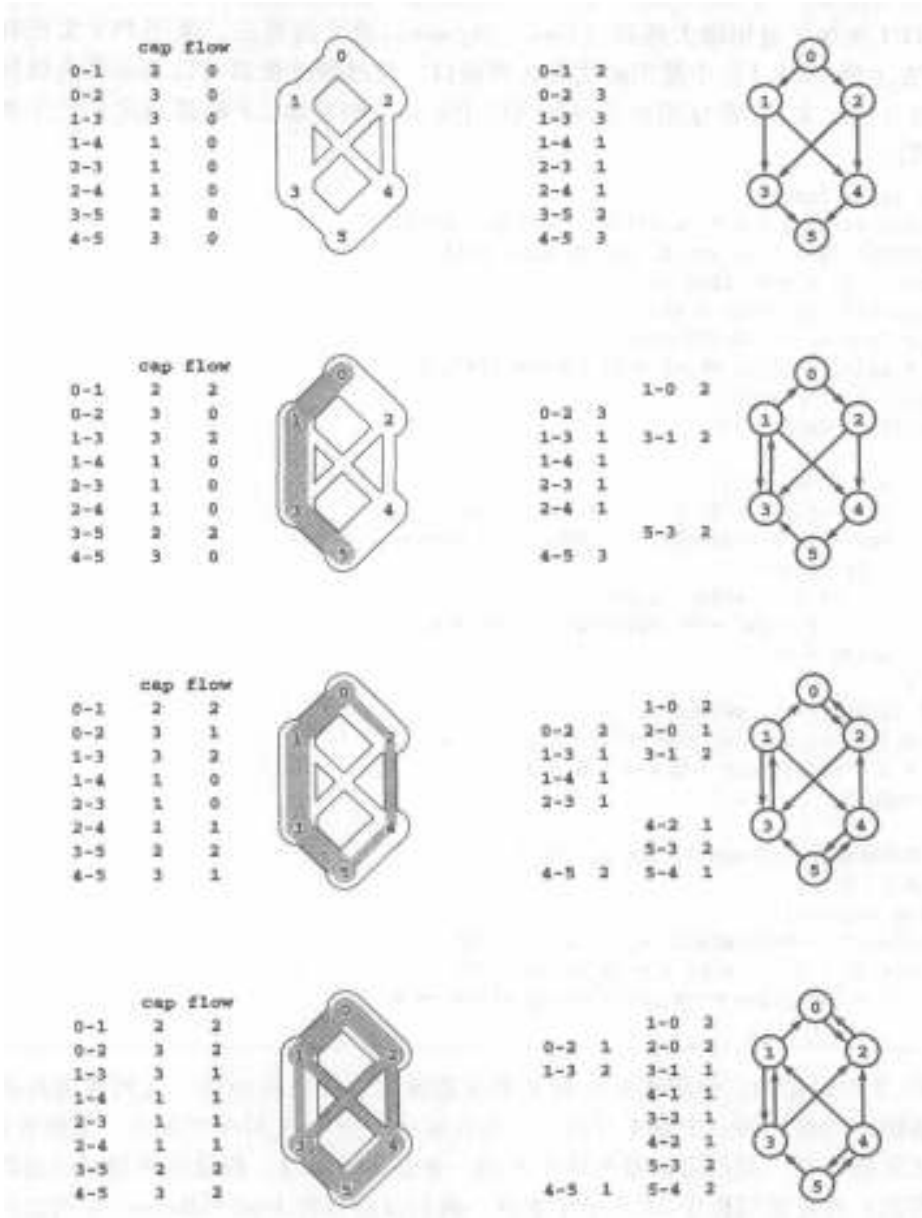


图 22-17 残量网络（增大路径）

找一个流网络中的增大路径等价于找该流所定义的残量网络中的有向路径。对于流网络中的每条边，我们在残量网络中创建每个方向的一条边：一条边是在流的方向，其权值等于未用容量，另一条边在流的相反方向，其权值等于该流。两种情况都不含权值为零的边。初始时（上图），残量网络就是权值为容量的流网络。当我们沿着路径 0-1-3-5 增大时（从上数第二个），我们将 0-1 和 3-5 充满至其容量，因而它们在残量网络中改变了方向，我们把 1-3 的权值减少使其与残流对应，并增加权值为 2 的边 3-1。类似地，当我们沿着 0-2-4-5 增大路径时，将 2-4 充满至其容量，使它改变方向，且在 0 和 2、4 和 5 之间的两个方向都有表示流和未用容量的边。沿着路径 0-2-3-1-4-5 增大之后（下图），在残量网络中，不再有从源点到汇点的有向路径，因而不会再有增大路径。

程序 22.3 增大路径最大流实现

这个 ADT 实现了通用增大路径（Ford-Fulkerson）最大流算法，使用 PFS 实现和我们在 Dijkstra 算法（程序 21.1）中使用的优先队列接口，经过修改使其可以表示带有链接到网络边的生成树（见正文）。宏 Q 给出了残量网络中 u 的边的容量。P 的适当定义产生各种不同的最大流算法。

```
static int wt[maxV];
#define Q (u->cap < 0 ? -u->flow : u->cap - u->flow)
int GRAPHpfs(Graph G, int s, int t, link st[])
  { int v, w, d = M; link u;
    PQinit(); priority = wt;
    for (v = 0; v < G->V; v++)
      { st[v] = NULL; wt[v] = 0; PQinsert(v); }
    wt[s] = M; PQinc(s);
    while (!PQempty())
      {
        v = PQdelmax();
        if ((wt[v] == 0) || (v == t)) break;
        for (u = G->adj[v]; u != NULL; u = u->next)
          if (Q > 0)
            if (P > wt[w = u->v])
              { wt[w] = P; PQinc(w); st[w] = u; }
        wt[v] = M;
      }
    if (wt[t] == 0) return 0;
    for (w = t; w != s; w = st[w]->dup->v)
      { u = st[w]; d = ( Q > d ? d : Q ); }
    return d;
  }
void GRAPHmaxflow(Graph G, int s, int t)
  { int x, d;
    link st[maxV];
    while ((d = GRAPHpfs(G, s, t, st)) != 0)
      for (x = t; x != s; x = st[x]->dup->v)
        { st[x]->flow += d; st[x]->dup->flow -= d; }
  }
```

如在 21.2 中讨论的，使用优先队列来实现边缘集数据结构的栈、队列或随机队列可以导致边缘集操作的额外的一个 $\lg V$ 开销。因为在实现中把这些 ADT 实现为一个像程序 18.10 那样的广义队列 ADT，我们可以避免这个开销。在分析算法时，假设这些情况的边缘集操作的开销为常量。通过使用程序 22.3 单个实现，我们强调各种 Ford-Fulkerson 实现之间的直接关系。

尽管它是通用的，程序 22.3 并没有包含 Ford-Fulkerson 算法的所有实现（例如，见练习 22.38 和练习 22.40）。研究人员继续开发实现算法的新方法。但是程序 22.3 包含的算法类已得到广泛应用，使我们对理解网络流的计算建立了一个基础，并且引入了实际中执行很好的直接实现。

正如所见到的，这些基本算法学工具给出了网络流问题的简单解（且应用广泛）。然而对确定哪一种是最好的方法的完整分析是一件复杂的任务，因为它们的运行时间取决于：

- 找出一个最大流所需的增大路径数。
- 找出每个增大路径所需的时间。

这些数量变化很大，与所处理的网络有关，还与图搜索策略有关（边缘集数据结构）。

也许最简单的 Ford-Fulkerson 实现使用了最短（shortest）增大路径（根据路径上边数度量，而不是根据流或容量度量）。这种方法是由 Edmonds 和 Karp 于 1972 年提出的。为了实现这种方法，我们使用了队列表示边缘集，在程序 22.3 中要么使用增量计数器的值表示 P，要么使用队列 ADT 代替优先队列 ADT。在这种情况下，在残量网络中搜索一条增大路径相当于进行广度优先搜索（BFS），正如 18.8 节和 21.2 节中所描述的。图 22-18 通过一个示例显示了 Ford-Fulkerson 方法实现的过程。为简明起见，我们把这种方法称为最短增大路径（shortest-augmenting path）最大流算法。由图可知，增大路径长度形成一个非降序列。在性质 22.7 中对此方法的分析证明了这个性质很特殊。

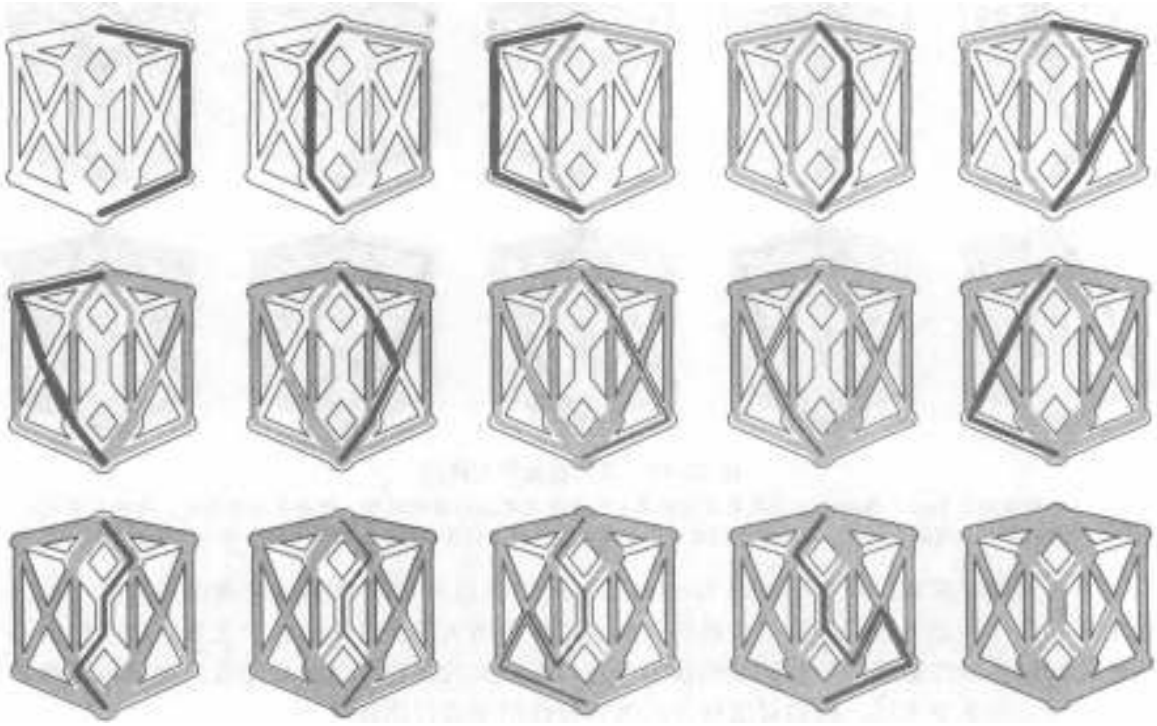


图 22-18　最短增大路径

这个序列显示了 Ford-Fulkerson 方法在一个网络中找最大流的一种最短增大路径实现。随着算法的进行，路径长度增加：最上一行的前 4 条路径长度为 3；最上一行的最后一条路径长度为 4；最下一行前两条路径长度为 5；在两条长度为 7 的路径都各有一条后向边时，算法结束。

Edmonds 和 Karp 提出了另一种 Ford-Fulkerson 的实现：沿着使流增加最大量的路径增大。为了使用一般 PFS 来实现这一方法，在程序 22.3 中我们使用优先队列

```
#define P ( Q > wt[v] ? wt[v] : Q )
```

这个优先级使得算法从边缘集中选择边，从而给出通过前向边可以推进或后向边可以流出的最大流量。为了简明起见，我们把这种方法称为最大容量增大路径（maximum-capacity augmenting-path）最大流算法。图 22-19 说明了算法对于图 22-18 中的同一流网络的工作过程。

这些只是 Ford-Fulkerson 实现的两个例子（也是我们所能分析的例子）。本节末尾，我们还会考虑其他例子。在此之前，我们先对增大路径方法进行分析，以便了解其性质，并最终确定哪一种方法性能最佳。

为了在程序 22.3 中表示的算法类中做出选择，我们又遇到类似的情况。应该关注最坏情况的性能保证，还是强调把问题表示成为一种与实际中遇到的网络无关的数学猜想呢？这个问题与上下文尤为相关，因为我们建立的经典的最坏情况下的性能界限比我们所见的典型图的实际性能结果高很多。很多因素使这种情况更复杂。例如，集中版本算法的最坏情况下的运行时间不仅取决于 V 和 E，而且取决于网络中边上容量的值。开发有可保证性能的快速最大流算法一直是数十年来吸引人的问题，并且大量方法已提出。使用在实际中可能遇到的网络，对这些方法做出评价以选出足够精度的方法，并不像是我们研究的其他情况下的同一任务（比如说排序或搜索算法的典型应用问题）那样明确。

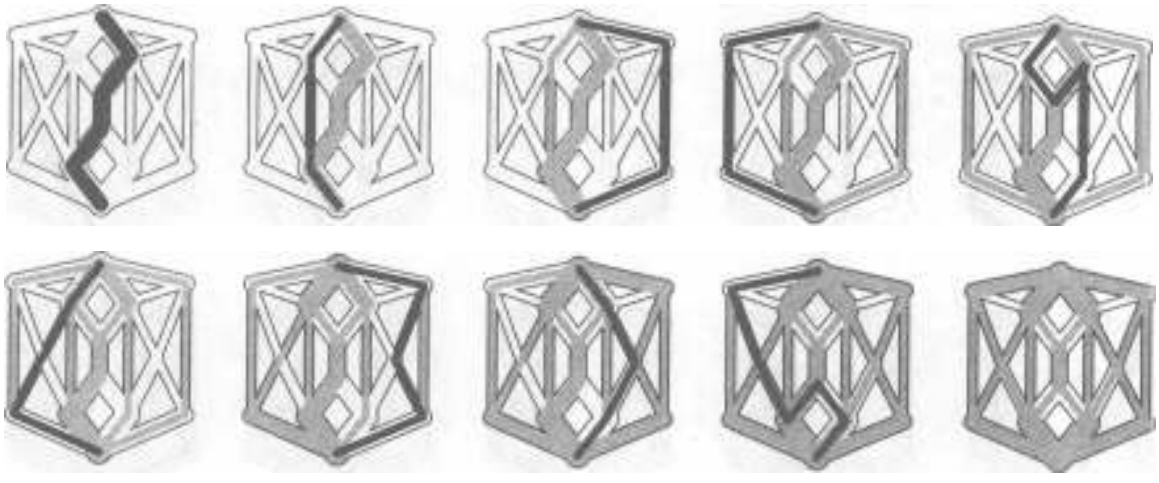


图 22-19 最大容量增大路径

这个序列描绘了 Ford-Fulkerson 方法找最大流的最大容量增大路径的实现过程。随着算法的进行，路径容量减小，但是它们的长度可能增加或减小。这种方法只需要 9 次增大路径就能计算出图 22-18 中所示的同一网络流。

考虑到这些困难，现在来考虑 Ford-Fulkerson 法的最坏情况性能的经典结果：有一个一般上界和两个特定上界，分别对应我们考察过的两种增大路径算法的各个上界。这些结果不仅使我们对算法性能进行足够精确的预测以进行有意义的比较，而且可使我们深刻地理解算法的特性。在本节末尾，我们对这些方法的实验性结果进行比较。

性质 22.6 *设 M 是网络中的最大边容量。Ford-Fulkerson 算法的实现所需的增大路径数目至多为 VM。*

证明 任何割至多有 V 条边，其容量最大为 M，则总容量为 VM。每条增大路径使通过每个割的流至少增 1，因此算法必定在 VM 遍后终止，因为在多次增大之后，所有割必定会充满至容量。 ■

这个上界是一个紧致界，例如，假设我们使用最长增大路径算法（基于直觉，路径越长，放在网络边上的流越多）。因为是统计迭代次数，这里暂且忽略计算这样一条路径的开销。图 22-20 所示的（经典）例子表明了在网络中选择最长路径使性质 20.6 中的上界成为紧致界：作为这样一个网络，它的最长增大路径算法的迭代次数等于边容量的最大值。这个例子表明我们必须进行详细的考察，以便知道是否还有其他的实现会使用比性质 22.6 中表明的更少的次数。同时它还给出了很多实际情况所使用的 Ford-Fulkerson 算法实现的运行时间的上界。

推论 *找出一个最大流所需要的时间为 $O(VEM)$，对于稀疏网络，即为 $O(V^2M)$。*

证明 由于广义图搜索是图表示大小的线性函数这一基本结果（性质 18.12），可以直接得到结论。如已经表明的，如果我们使用优先队列边缘实现，则还需 lg V 倍的因子。 ■

对于稀疏网络和有小规模整数容量的网络，这个上界是合理的。证明过程实际上确定了 M 因子可用网络中的最大与最小非零容量之间的一个比值替代（见练习 22.27）。当这个比值较小时，这个上界表明，最坏情况下 Ford-Fulkerson 的任何实现将会在与求解所有点对最短路径问题所需时间成正比的时间内找到一个最大流。有很多情况其中的容量的确很小，因子 M 也可以忽略。我们将在 22.4 节看到一个例子。

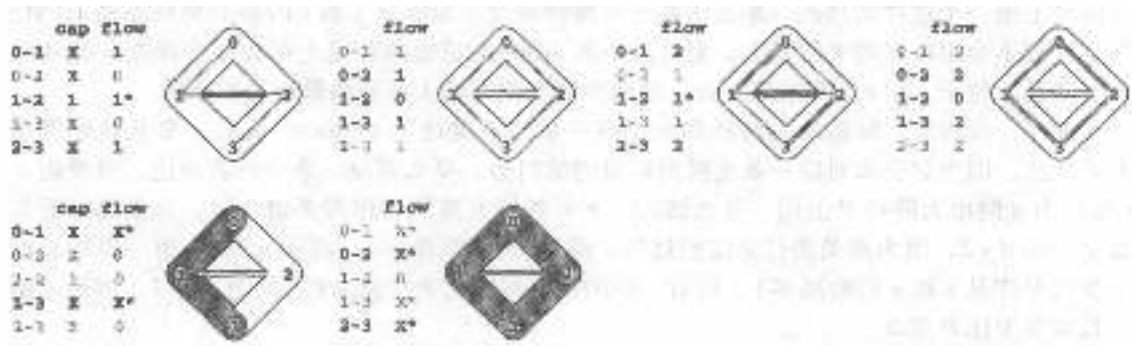


图 22-20 Ford-Fulkerson 算法的两种情况

这个网络说明了 Ford-Fulkerson 算法所使用的迭代次数依赖于网络中边上容量的大小以及实现所选择的路径序列。它由容量为 X 的 4 条边、容量为 1 的 1 条边组成。上图中描述的情况表明交替使用路径 0-1-2-3 和路径 0-2-1-3 作为增大路径（例如，有人偏爱长路径）的实现将要求 X 对的迭代，就像所显示的两对那样，每一对都使总流增加 2。下图描述的情况表明先选择 0-1-3，再选择 0-2-3 作为增大路径的实现（例如，有人偏爱短路径）只用两次迭代就找出了最大流。

如果边上容量（比如说）是 32-位的整数，上图所描述的情况将会比下图描述的情况慢数十亿倍。

当 M 较大时，VEM 最坏情况下的界限很大；但它是一种悲观的结论。因为我们是通过把最坏情况下的界限相乘得到这个结果的，这些界限是所构造的例子的界限。真实网络的实际开销一般会低得多。

从理论的观点，我们的首选目标是：使用 17.8 节中的粗略分类来发现带有较大整数权值的网络的最大流问题是否是易解问题（多项式时间算法可解）。所导出的这个上界并不能解决这个问题，因为最大权值 $M=2^m$ 可能呈 V 和 E 的指数增长。从实际的观点来看，我们可以选择更好的性能保证。要选择一个典型的实际例子，假设使用 32-位的整数（$m=32$）来表示边上的权值。在一个有数百个顶点和数千条边的图中，性质 22.6 的推论表明在一个增大路径的算法中我们必须进行数百万亿次的操作。如果要处理数百万个顶点，这个结果是没有实际意义的，不仅由于我们将不会有大到 $2^{1\,000\,000}$ 的权值，而且还由于 V^3 和 VE 是如此大以至于使得这个上界毫无意义。我们对找解决这个易解问题的多项式的上界感兴趣，也对在实际中可能遇到的相关情况的更好上界感兴趣。

性质 22.6 具有一般性：它可应用到任何 Ford-Fulkerson 的实现。算法的通用性质给我们留下很大的灵活性，使我们可以考虑大量简单实现以改进性能。我们期望特定的实现可能是更好最坏情况界限的一个研究主题。实际上，这是我们把它放在第一位考虑的主要原因！现在，如我们已看到的，实现并使用一大类的这些实现是很容易的：我们只需要替代不同广义队列实现或程序 22.3 中的优先定义即可。分析最坏情况下的性能差异具有挑战性，就如我们接下来考虑的两种基本增大路径实现的经典结果。

首先，我们分析最短增大路径算法。这种方法不是图 22-20 中描述的问题的主题。实际

上，我们可以使用它，将最坏情况下的运行时间中的 M 因子用 $VE/2$ 代替。由此得出网络流是易解问题。我们甚至可能对它比较容易地分类（如果聪明的话，通过一种简单实现可在多项式时间解决实际问题）。

性质 22.7 *Ford-Fulkerson 算法的最短增大路径实现所需的增大路径数目至多为 $VE/2$。*

证明 首先，由图 22-18 中的例子可显然看出，任何增大路径都不会比前一条增大路径更短。为了确定这一点，我们用反证法来说明一个更强的性质成立：任何增大路径都不会减少从源点 s 到残量网络中的任一顶点的最短路径长度。假设存在这样的扩展路径，而且 v 是此路径上第一个这样的顶点。那么需要考虑两种情况：要么从 s 到 v 的新的更短路径上的任何顶点都不会出现在增大路径上，要么这条从 s 到 v 的更短的路径上存在某个顶点 w 出现在增大路径上位于 v 和 t 之间的某一处。这两种情况都与增大路径的最小性相矛盾。

现在，由构造，每条增大路径都至少有一条“关键边”（critical edge）：要从残量网络删除此边，因为它要么对应一条充满到容量的前向边，要么对应一条空的后向边。假设边 u-v 是长为 d 的增大路径 P 上的一条关键边。下一条增大路径它作为关键边时，这条路径的长度至少为 $d+2$，因为那条路径必定经过从 s 到 v，然后沿着 v-u，再从 u 到 t。第一段的长度至少比 P 中从 s 到 u 的距离多 1，最后一段的长度至少比 P 中从 v 到 t 的距离多 1，因而该路径长度至少比 P 多 2。

由于增大路径的长度至多为 V，这些事实蕴含着每条边都可能是至多 $V/2$ 条增大路径上的关键边，因而增大路径总数至多为 $EV/2$。 ■

推论 *找出稀疏网中的一个最大流所需时间为 $O(V^3)$。*

证明 找出一条增大路径所需时间为 $O(E)$，因而总时间为 $O(VE^2)$。由此可得陈述的上界。 ■

V^3相当大，它并不能为大型网络提供一个良好的性能界限，但是这个事实并不会排除把该算法用于大型网络。因为它是一种最坏情况下的性能结果，对于预测实际应用的性能并不是很有用。例如，正如所提到的，最大容量 M（或者容量之间的最大比率）可能要比 V 小得多，因而性质 22.6 的推论会给出一个更好的界限。实际上，在最好的情况下，Ford-Fulkerson 方法所需的增大路径数目是 s 的出度或 t 的入度中的较小者，它比 V 小得多。给定这个最好和最坏性能的范围，只根据最坏情况的上界来比较增大路径算法是不明智的。

不过，还有一些与最短增大路径方法一样简单的其他实现，它们可能有更好的上界，或在实际中更被偏爱（或者兼而有之）。例如，对于图 22-18 和图 22-19 所示的例子，最大增大路径算法找出一个最大流时所用的路径数远远少于最短增大路径算法。我们现在转到该算法的最坏情况分析。

首先，正如 Prim 算法和 Dijkstra 算法（见 20.6 和 21.2 节），我们可以实现优先队列，使算法在最坏情况下每次迭代的时间与 V^2成正比（对于稠密图），或与 $(E+V)\log V$ 成正比（对于稀疏图）。不过这些估计是悲观的，因为算法到达汇点时会终止。我们还看到如果使用高级数据结构还可以做得更好一些。越来越重要的挑战问题是需要多少增大路径。

性质 22.8 *在 Ford-Fulkerson 算法的最大增大路径实现中所需的增大路径数目至多为 $2E \lg M$。*

证明 给定一个网络，令 F 是它的最大流值。令 v 是算法寻找最大路径的过程中某处的流值。把性质 22.2 应用到残量网络，我们可以把流分解为至多 E 条有向路径，其总和为 $E-v$，因而至少有一条路径中的流至少为 $(F-v)/E$。现在，要么在对另外 $2E$ 条增大路径处理完之前的某个时候找到最大值，要么在处理 $2E$ 条路径之后增大路径的值小于

$(F-v)/2E$，而这比处理$2E$条路径序列之前的最大值减少一半。也就是说，在最坏情况下，我们需要$2E$条路径的序列使路径值降低一倍。第一条路径值至多为M，为此最多需要$\lg M$次使路径值每次降低一倍。因此总共至多有$\lg M$个$2E$条路径的序列。 ■

推论 *在一个稀疏网中找一个最大流所需的时间为* $O(V^2 \lg M \lg V)$。

证明 与性质20.7和性质21.5中的证明一样，使用基于堆的优先队列实现直接而得。 ■

对于实际中典型遇到的M和V值，这个上界要比性质22.7的推论的$O(V^3)$上界小得多。在许多实际情况下，最大增大路径算法使用的迭代次数要比最短增大路径算法使用的迭代次数少得多。不过发现每条路径的代价要高一些。

还有许多其他的变型需要考虑，这在有关最大流算法的大量文献中反映出来。有更好最坏情况下上界的算法还在继续被发现，也没有非平凡下界得以证明。也就是说，仍然可能存在线性时间的算法。尽管从理论的角度它们是重要的，但对于稠密图，很多算法主要被设计为降低最坏情况下的上界。因而，与实际中我们遇到的稀疏图的最大增大路径算法相比，它们并不能够提供实质上的更好性能。不过，仍然有很多选择可以利用来追求更好的实用最大流算法。在22.3节，我们还将考虑另一类算法。

一种简单的增大路径算法是为P使用一个递减计数器的值，或者用一个栈实现来代替程序22.3中的广义队列，使搜索增大路径就像深度优先搜索。图22-21显示了这个算法对于这个小例子所计算出的流。直觉看来该方法很快，易于实现，且似乎可以把流放在网络中。如我们将要看到的，它的性能变化很大，对于某些网络性能很差，而对另一些网络则是合理的。

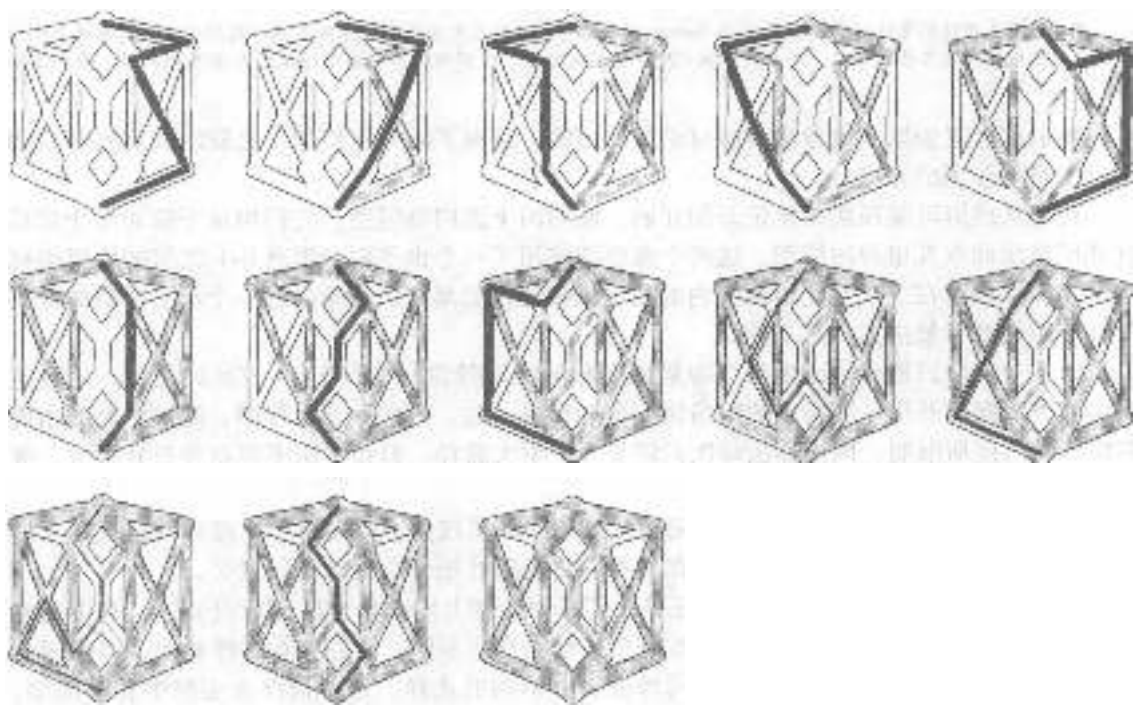


图22-21 基于栈的增大路径搜索

此图说明了在Ford-Fulkerson方法的实现中，利用栈代替广义队列的结果，因而搜索路径的过程就像DFS。在这种情况下，该方法的效果与BFS的效果一样，但是其行为对网络的表示相当敏感，尚未得到分析。

另一种方法是使用一个随机化的队列实现来代替广义队列。因而搜索增大路径是一个随机搜索。图 22-22 显示了这个算法对于这个小例子所计算出的流。该方法也很快，易于实现。此外，如我们在 18.8 节所提到的，它可能包含了广度优先搜索和深度优先搜索的良好特性。随机化在算法设计中是一种强大的工具。这个问题表示一种我们考虑使用的一种合理情况。

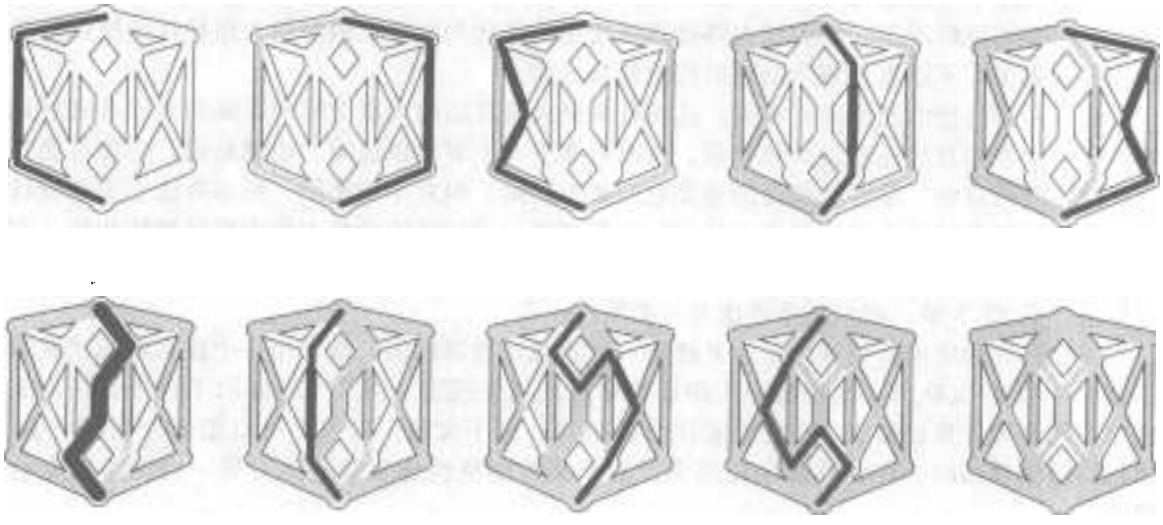


图 22-22　随机化的增大路径搜索

这个序列是用随机化队列代替在 Ford-Fulkerson 方法的增大路径搜索使用的边缘集数据结构所的结果。在这个例子中，我们发现短的高容量的路径，并且需要相对较少的增大路径。但预测这个方法的性能特征具有挑战性，在很多情况下，它效果很好。

我们最后更为深入地考察前面讨论过的方法，从而了解对它们加以比较以及对实际应用的性能进行预测的难度。

我们从理解可能预期的定量差别开始，使用两个流网络模型，它们均基于前面用于比较其他图算法的欧几里得图模型。这两个模型都使用了一个由平面上带有 0-1 之间的随机坐标的 V 个顶点以及在一个给定距离之内的两个点相连接的某些边所绘制的一个图。它们的差别仅在于对边的容量指定有所不同。

第一个模型只是简单地为每个容量赋以相同的常量值。如在 22.4 节所讨论的，这种类型的网络流问题要比一般类型的网络流问题要简单一些。对于欧几里得图，流被源点的出度和汇点的入度所限制，因而算法每次只需要几个增大路径。但是算法不同路径差异很大，我们很快会看到这一点。

第二个模型则为容量赋以某个固定值范围内的随机权值。此模型所生成的网络正是人们思考该问题所想到的类型，各种算法在这些网络上的性能肯定很有指导意义。

两种模型如图 22-23 所示，此外还显示了在两个网上用四种方法计算出的流。对于这些例子，可能最值得注意的特性就是流本身在特征上有所不同。它们都有同样的值，但是网有很多最大流，在计算它们的时候，不同算法会有不同的选择。这种情况在实际中相当典型。我们可能会对希望计算的流设置其他条件，但是对问题的改变将使之更为困难。我们在第 22.5 节至第 22.7 节考虑的最小成本流问题是形式化这种情况的一种方法。

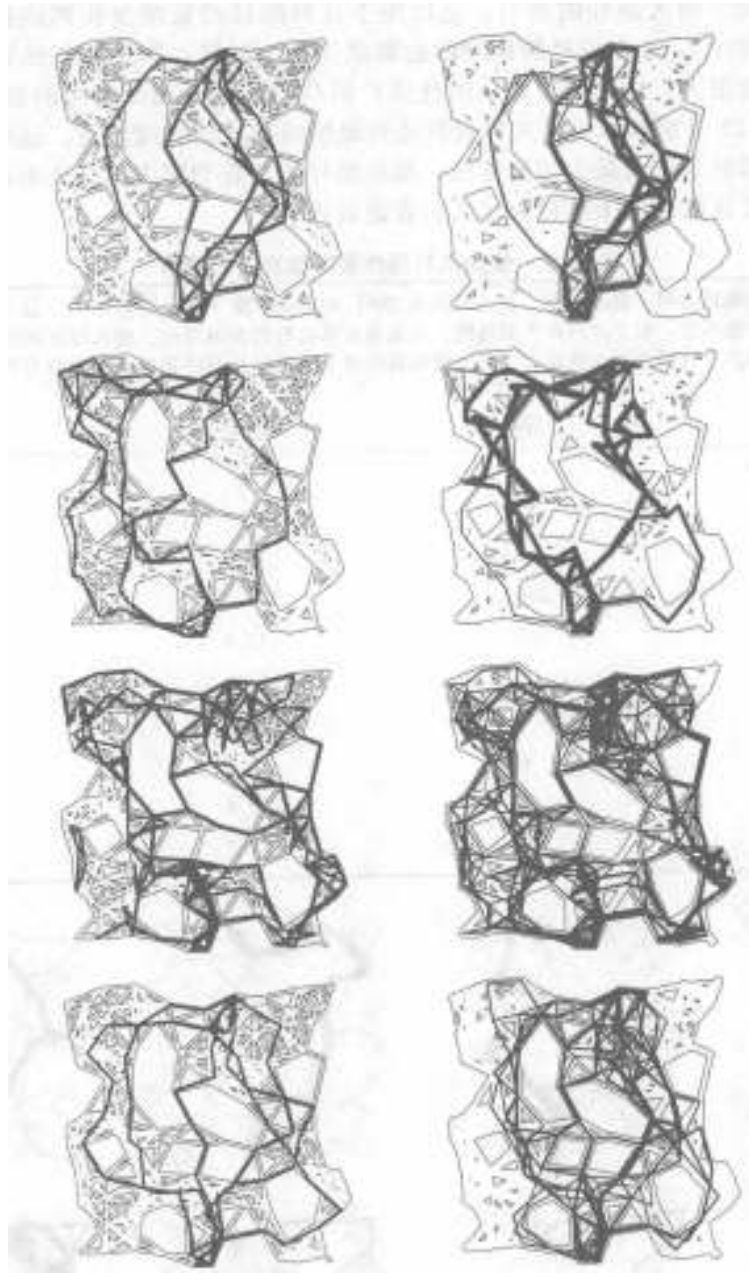


图 22-23 随机流网络

此图描绘了在随机欧几里得图上计算最大流的过程，使用两个不同的容量模型。左图中，所有边被赋以单位容量；右图中，所有边被赋以随机容量。源点在图的中上部，汇点在图的中下部。自上而下描绘的分别是最短路径、最大容量、基于堆栈和随机算法所计算出的流值。因为顶点的度不大，容量是小整数，有很多不同的流可以达到这些例子中的最大流。

汇点的入度为6，因为所有算法都可利用6条增大路径找到左图的单位容量模型中的流。

对于右图的随机权值模型来说，寻找增大路径的方法在特征上则有明显的不同。特别是，基于栈的方法会找出小权值的长路径，甚至会产生一个带有不连通环的流。

表 22-1 给出了使用四种方法计算图 22-23 中的流的更为详尽的定量结果。增大路径算法的性能不仅取决于增大路径的数目，还取决于这种路径的长度及找到路径的开销。特别是，运行时间与程序 22.3 内循环所检测的边数成正比。如常，即使对于给定的图，这个数目的变化范围可能很大，这取决于表示的性质；但我们仍可以刻画不同的算法特征。例如，图 22-24 和图 22-25 分别显示了最大容量算法和最短路径算法的搜索树。这些例子可以支持一般性的结论，那就是比起最大容量算法，最短路径方法花费更多的功夫来找出具有更少流的增大路径，因此这就有助于解释为什么后者是首选的。

表 22-1 优先队列操作最坏情况下的开销

对于所示的欧几里得近邻网（随机容量，最大流值为 286）和单位容量（最大流值为 6），这个表格显示了各种增大路径网络流算法的性能参数。对于这两种类型的网，最大容量算法胜过其他算法。随机搜索算法找出的最大路径不比最短路径找出的长很多，且检查更少的结点。基于栈的算法对于随机图执行得很好，尽管它有很长的路径，但仍可与单位权值竞争。

	路径	均值长度	总边数
随机容量 1-50			
最短	37	10.8	76 394
最大容量	7	19.3	15 660
深度优先	286	123.5	631 392
随机	35	12.8	58 016
容量 1			
最短	6	10.5	13 877
最大容量	6	14.7	10 736
深度优先	6	110.8	12 291
随机	6	12.2	11 223

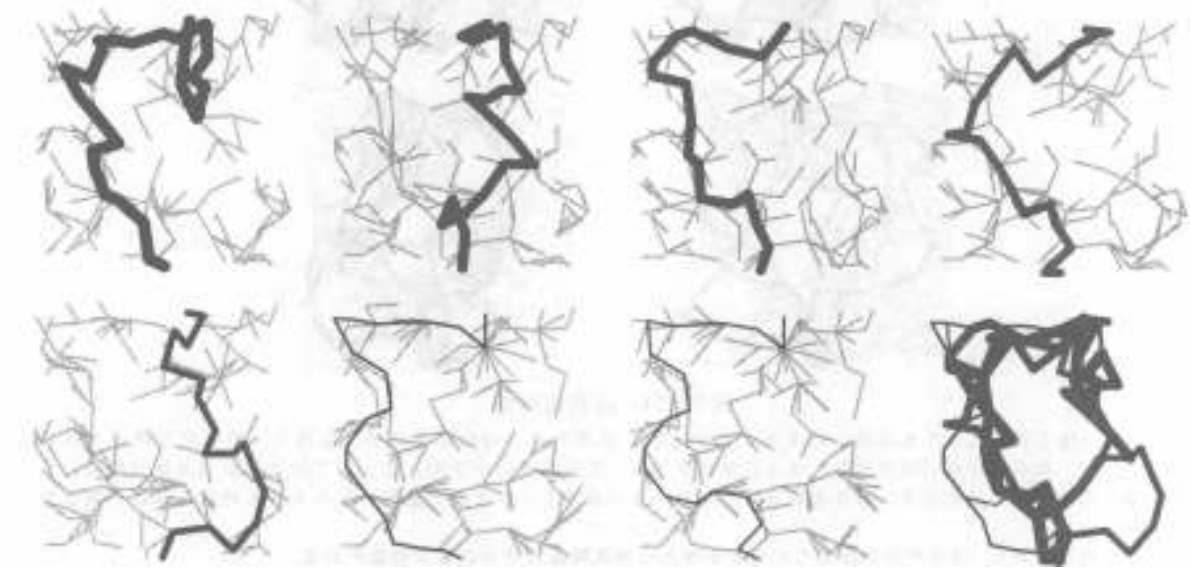


图 22-24 最大容量增大路径（大型例子）

对于图 22-23 中显示的具有随机权值的欧几里得网络，此图描绘了最大容量算法计算出的增大路径，以及图搜索生成树的边（灰色显示）。所得流在右下方显示。

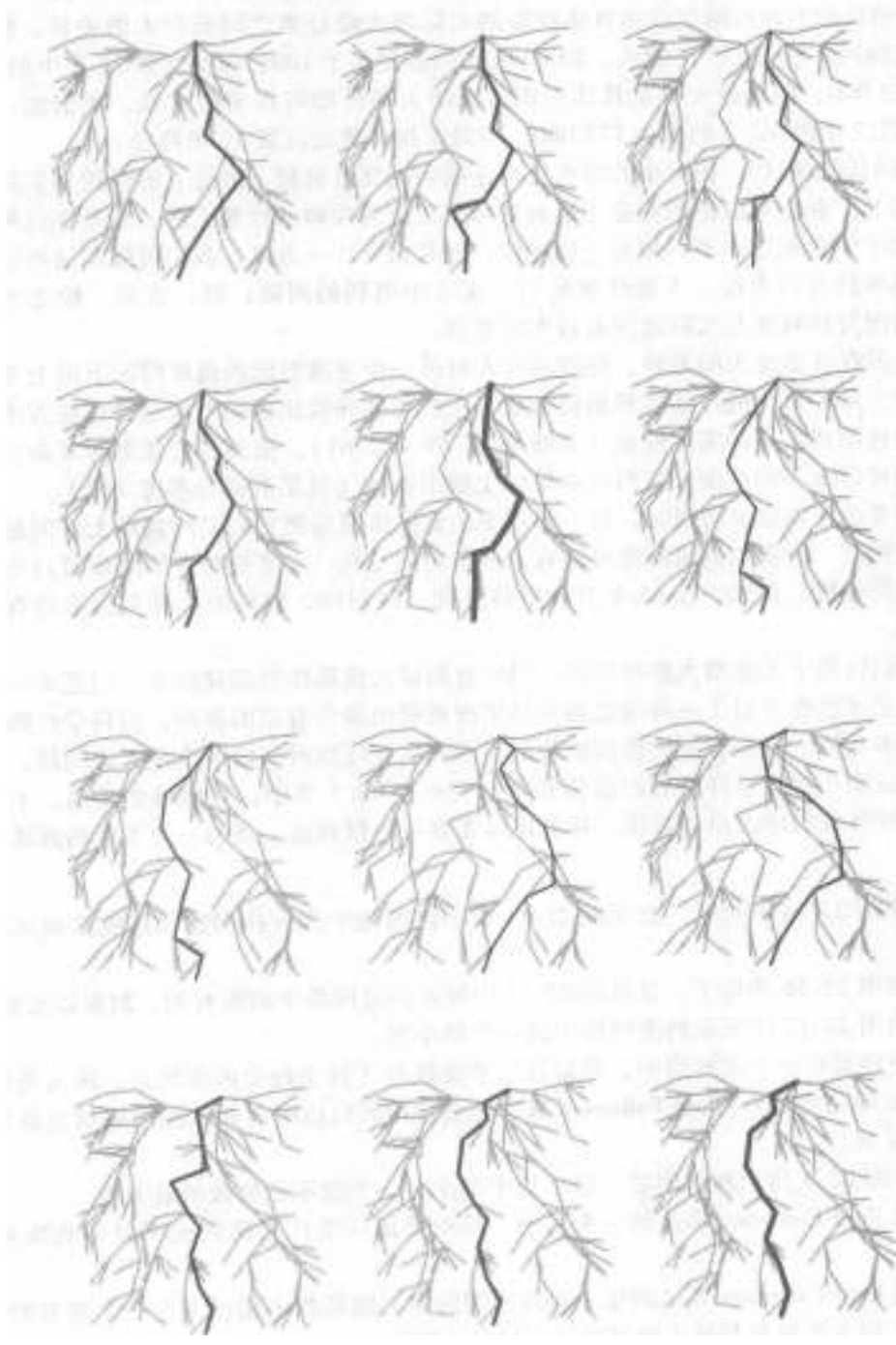


图 22-25 最短增大路径（大型例子）

对于图 22-23 中显示的具有随机权值的欧几里得网络，此图描绘了最短路径算法计算出的增大路径，以及图搜索生成树的边（灰色显示）。这个算法要比图 22-24 描述的最大容量算法慢得多。原因是它需要大量的增大路径（所显示的只是总数为 37 条路径中的前 12 条）而且生成树较大（通常几乎包含所有顶点）。

使用这种方法详尽地研究特定的网络，我们可以学到的最重要的收获是性质 22.6 到性质 22.8 中给出的上界与给定应用算法所需的实际增大路径数之间有巨大的差异。例如，图 22-24 描述的网络中有 177 个顶点，2 000 条边的容量小于 100，因而性质 22.8 中的 $2E \lg M$ 的值大于 25 000；但是最大容量算法只需 7 条增大路径即可找出最大流。类似地，对于此网，性质 22.7 中的 $VE/2$ 的值为 177 000，但最短路径算法仅需 37 条路径。

正如我们所指明的，相对小的结点度和连接的局部性解释了理论上的性能和实际性能之间的这些差异。我们可以证明对应于这些具体情况更为准确的性能上界；但是在流网络模型和实际模型中，存在这些不一致是一般规律，而非例外。一方面，我们可能用这些结果来说明这些网络不具有代表性，不能代表我们在实际中遇到的网络；另一方面，较之于这些网络，最坏情况分析可能与实际情况有较大的差异。

由于上界存在如此大的差异，促使研究人员进一步寻求更低的最坏情况下的上界。还有很多其他增大路径算法的可能实现值得考虑，它们可能导致比我们考虑过的那些方法有更好的最坏情况性能或更好的实际性能（见练习 22.57 ~ 22.61）。很多方法都更为复杂，已经表明改进了最坏情况下的性能，它们可在研究文献中查到（见第五部分参考文献）。

在我们考虑其他很多问题时，另一种重要的复杂情况是把它们归约到最大流问题。当这样的归约应用时，所得到的流网络可能有一些特定的结构，而某些特定算法则可以充分利用此结构来提高性能。例如，在 22.8 节我们将研究一种归约，它可给出所有边上均有单位容量的流网络。

即使我们仅限于关注增大路径算法，仍可看到最大流算法的研究既是一门艺术，也是一门科学。其艺术性在于对于一种给定的实际情况选择出最为有效的策略；而科学性则在于理解问题的基本实质。是否有新的数据结构和算法可以在线性时间内解决最大流问题，或者我们是否能够证明不存在这样的数据结构和算法呢？在 22.3 节中，我们将会看到，不存在最坏情况下线性性能的增大路径算法，我们还将考察可能做到这一点的一类不同的算法。

练习

22.21 按照图 22-14 的样子，显示图 22-11 所示流网络中你所找到的尽可能多的不同增大路径序列。

22.22 按照图 22-16 的样子，显示图 22-11 中所示的流网络中的所有割、割集以及容量。

▷ **22.23** 找出图 22-12 中所示的流网络中的一个最小割。

○ **22.24** 假设容量在一个流网络中，容量处于平衡状态（对于每个内部结点，流入边的总容量等于流出边的总容量）。Ford-Fulkerson 算法还会使用后向边吗？如果使用则给出证明，否则给出一个反例。

22.25 对于图 22-5 所示的流网络，给出其中至少有一个流不是整数的最大流。

▷ **22.26** 开发 Ford-Fulkerson 算法的一个实现，其中使用一个广义队列而不是优先队列（见 18.8 节）。

▷ **22.27** 证明 Ford-Fulkerson 算法的任一实现所需的增大路径数不超过 V 与一个整数的乘积，该整数是大于最大边容量与最小边容量比率的最小整数。

22.28 证明最大流问题的一个线性时间下界（lower bound）：即，对于任何 V 和 E 值，任一最大流算法必定检查包含 V 个顶点和 E 条边的某个网络中的每条边。

▷ **22.29** 给出一个类似图 22-20 的网，使得对于此网最短增大路径算法有最坏情况下的性能。

22.30 给出图 22-20 中网的邻接表示，使得基于栈的搜索实现（程序 22.1 和程序 22.3，

使用栈表示广义队列）有图中所示的最坏情况下的行为。

22.31 按照图22-17中的样子，使用最短增大路径算法找出图22-11所示的流网络中的一个最大流，显示每次增大路径后的流和残量网络。

22.32 使用最大容量增大路径算法做练习22.31。

22.33 使用基于栈的增大路径算法做练习22.31。

○ 22.34 展示一类网，对于这类网，最大增大路径算法需要 $2E \lg M$ 条增大路径。

● 22.35 对于练习22.34中的例子，你能否对边进行排列，使得我们的实现找出每条路径所需时间与 E 成正比。如果需要，可以修改例子来达到这个目标。描述对于此例所构造的邻接表表示，并解释如何达到最坏情况。

22.36 对于各种网（见练习22.7～22.12）进行实验性研究，对于本节描述的4种算法，确定增大路径数和运行时间与 V 的比值。

22.37 对于21.5节的欧几里得网，开发并测试使用源点－汇点最短路径启发式方法的增大路径方法的一种实现。

22.38 开发并测试基于交替使用源点和汇点为根的增长搜索树（见练习21.35和练习21.75）的增大路径方法的一种实现。

● 22.39 程序22.3的实现在找到从源点到汇点的第一条增大路径时终止，然后再次开始全面搜索。还有一种方法，继续进行搜索找出另一条路径，继续这个过程直到所有顶点被加上标志。开发并测试第二种方法。

22.40 开发并测试使用非简单路径的增大路径方法的实现。

▷ 22.41 给出简单增大路径的一个序列，使其产生图22-12所示的网络中带有一个环的流。

○ 22.42 给出一个例子，显示并非所有最大流都可从空网络开始沿着从源点到汇点的简单路径序列增大而得。

22.43 使用开始时利用一次增大路径法，接着切换到另一不同的增大路径法来结束的混合方法进行实验（你的部分任务是确定切换的合适准则）。对于各种网络（见练习22.7～22.12）进行实验研究，将这些方法与基本方法进行比较，深入研究那些比其他方法更好的方法。

22.44 使用交替利用两种或更多不同增大路径方法的混合方法进行实验。对于各种网络（见练习22.7～22.12）进行实验研究，将这些方法与基本方法进行比较，深入研究那些比其他方法更好的方法。

○ 22.45 使用随机利用两种或更多不同增大路径方法的混合方法进行实验。对于各种网络（见练习22.7～22.12）进行实验研究，将这些方法与基本方法进行比较，深入研究那些比其他方法更好的方法。

22.46 ［Gabow］开发一个使用 $m = \lg M$ 步的最大流实现，其中第 i 步使用容量的前 i 位来求解最大流问题。从各处的0流开始；接着在第一步之后，使在前一步中找到的流加倍来初始化流。对于各种网络（见练习22.7～22.12）进行实验研究，将这一实现与基本方法进行比较。

● 22.47 证明练习22.46中描述的算法的运行时间为 $O(V E \lg M)$。

○ 22.48 给定整数 c，编写一个流网络ADT函数，找出一条这样的边，使得该边上的容量增加 c，就会使最大流增加一个最大量。你的函数可以假设客户已经调用GRAPHmaxflow来计算一个最大流。

●● 22.49 假设已知网络的一个最小割。试问此信息对于计算最大流是否更容易？使用给定最

小割，开发一个实质性地加速搜索最大容量增大路径的算法。

● **22.50**　编写一个客户程序，对于增大路径算法进行动态可视化模拟。你的程序应该产生类似图 22-18 和本节其他图形的图像（见练习 17.55 ~ 17.59）。使用练习 22.7 ~ 22.12 中的欧几里得网来测试你的实现。

22.3　预流 - 推进最大流算法

在这一节里，我们考虑另一种求解最大流问题的方法。使用称为预流 - 推进（preflow-push）法，沿着顶点的出边逐渐移动流，使其流入量多于流出量。预流 - 推进法是由 A. Goldberg 和 R. E. Tarjan 在 1986 年根据更早期的一些算法提出的。由于它的简单、灵活和高效，已得到广泛应用。

如 22.1 节所定义的，一个流必定满足平衡条件，即从源点的流出量等于到汇点的流入量，且在每个内部结点上，流入量等于流出量。我们称这样的流为可行（feasible）流。增大路径算法总是维持一个可行流：它沿着增大路径增加流，最终得到最大流。对比之下，本节考虑的预流 - 推进算法维持的最大流不是可行流，因为某些顶点的流入量比流出量要大：它们通过这些顶点推进流，直到达到一个可行流（也即不再存在这样的顶点）。

定义 22.5　*在一个流网络中，预流（preflow）是一个正边流的集合，满足以下条件：每条边上的流不大于该边上的容量，且对于每个内部结点，其上的流入量不小于它的流出量。活动（active）顶点是一个其流入量大于流出量的内部顶点（按照约定，源点和汇点永不会是活动顶点）。*

我们把活动顶点的流入量与流出量的差称为该顶点的盈残量（excess）。为了改变活动顶点的集合，选择一个活动顶点，沿着一条发出边推进（push）它的盈残量，或者，如果容量不足，则沿着进入边倒推进这个盈残量。如果推进使该顶点的流入量和流出量相等，顶点就变成不活动的顶点；推入另一顶点中的流可能激活那个顶点。预流 - 推进方法为反复推进活动顶点的盈残量提供了一种系统的方法。我们把活动顶点保存在一个广义队列中。正如在增大路径方法的所做的，这个决定给出了一种通用的算法，可以包含整个一类更为确定的算法。

图 22-26 是一个描述预流 - 推进算法中使用的基本操作的小例子，按照我们使用过的术语来说明，我们认为流在页面中只能从上而下。可以想象是把一个活动结点的盈残量沿着发出边推进，或者沿这个顶点暂时回退，沿着它的进入边倒推进盈残量。

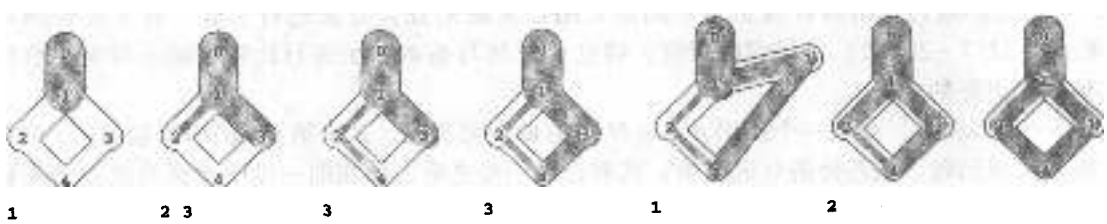


图 22-26　预流 - 推进示例

在预流 - 推进算法中，我们维持一个活动结点表，其中结点的流入量比流出量大（每个网络下显示出）。该算法的一种版本是使用一个循环，从该表中选择一个活动结点，并沿着流出边推进流，直到表中不再有活动结点。在这个过程中还可能创建其他的活动结点。在这个例子中，我们沿着 0-1 推进流，使结点 1 成为活动结点。接下来，沿着 1-2 和 1-3 推进流，使结点 1 成为不活动的结点，而 2 和 3 成为活动结点。然后，沿着 2-4 推进流，使结点 2 成为不活动结点。但 3-4 没有足够容量可以沿着它推进流，使结点 3 成为不活动的结点。因而，我们还沿着 3-1 倒推进流，使结点 1 成为活动结点。然后，沿着 1-2 以及 2-4 推进流，这使得所有结点成为不活动结点，且得到最大流。

图 22-27 使用一个例子，描述为什么预流 - 推进法优于增大路径方法。在这个网络中，任何增大路径法可以反复地把微量的流放进一条长的路径中，缓慢地充满这条路径上的边，直到最终得到最大流。对比之下，预流 - 推进法遍历那条路径时首先充满那条长路径上的边，然后直接把那个流分布到汇点中，无需再次遍历那条长的路径。

正如在增大路径算法中所做的那样，我们使用残量网络（见定义 22.4）来保存可能推进流的边。残量网络中的每条边表示可能推进流的地方。如果一条残量网络的边与流网络中的对应边在同一个方向，则增加该流；如果是在反方向，则减少该流。如果增加使边充满或者减少使边变空，那么对应的边就从残量网络中消失。对于预流 - 推进算法，我们使用另一种机制来帮助决定残量网络中的哪些边可以帮助我们消除活动顶点。

定义 22.6 *在一个流网络中，给定流的高度函数*（height function）*是一个非负顶点权值集合* $h(0)...h(V-1)$，*满足对于汇点* t，$h(t)=0$；*对于流的残量网络的中的每条边* u-v，$h(u) \leqslant h(v)+1$。*一条合格边*（eligible edge）*是残量网络中满足* $h(u)=h(v)+1$ *的一条边* u-v。

没有合格边的一个平凡的高度函数为 $h(0)=h(1)=\ldots=h(V-1)=0$。如果我们设置 $h(s)=1$，那么从源点出发且有流的任何边都对应着残量网络中的一条合格边。

通过为每个顶点赋以到汇点的最短路径距离（即以 t 为根的逆网的任何 BFS 树中到根结点的距离，如图 22-28 所示），来定义一个更有趣的高度函数。这个高度函数是有效的，因为 $h(t)=0$，且对于边 u-v 连接的任一对顶点 u 和 v，从 u-v 开始的到 t 的任一最短路径的长度均为 $h(v)+1$；因而，从 u 到 t 的最短路径长度 $h(u)$ 必定小于等于那个值。这个函数起着特殊的作用，因为它为每个顶点设置了最大可能的高度。反过来，我们看到 t 的高度必定为 0；在高度为 1 处的顶点是残量网络中那些顶点的边直接指向 t 的顶点；在高度为 2 处的顶点是那些顶点的边直接指向高度为 1 的顶点，以此类推。

性质 22.9 *对于任何流及所关联的高度函数，一个顶点的高度不大于残量网络中从该顶点到汇点的最短路径的长度。*

证明 对于任一给定顶点 u，令 d 是从 u 到 t 的最短路径长度，$u=u_1, u_2, \ldots, u_d$ 是最短路

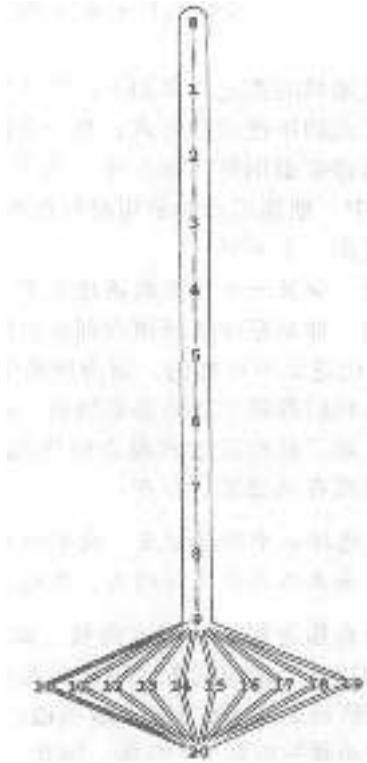


图 22-27 Ford-Fulkerson 算法的较坏情况

这个网络表示了 V 个顶点的网络族，其中的任何增大路径算法需要长为 $V/2$ 的 $V/2$ 条路径（因为每条增大路径必须包括这条长的垂直路径），总运行时间与 V^2 成正比。预流 - 推进算法在此网中找到最大流的时间为线性时间。

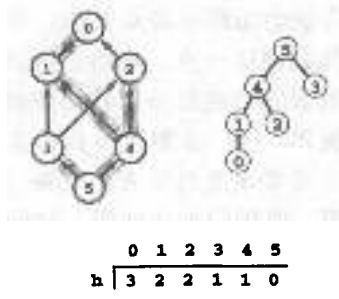


图 22-28 初始高度函数

右图的树是一棵根为 5 的 BFS 树，是左图倒过来的图。顶点索引数组 h 给出了从每个顶点到根结点的距离，是一个有效的高度函数：对于网络中的每条边 u-v，$h[u]$ 小于等于 $h[v]+1$。

径。那么

$$\begin{aligned} h(u) = h(u_1) &\leqslant h(u_2) + 1 \\ &\leqslant h(u_3) + 2 \\ &\cdots \\ &\leqslant h(u_d) + d = h(t) + d = d \end{aligned}$$

■

高度函数的真正含义如下：当一个活动结点的高度小于源点的高度时，可能存在从那个结点到汇点的推进流的方式；当一个活动结点的高度大于源点的高度时，我们知道那个结点的超出量需要被倒推入源点中。为了确定后一事实，我们从另一角度看待性质 22.9，在性质 22.9 中，把到汇点的最短路径长度看作高度一个上界；而在这里，我们把高度看作最短路径长度的一个下界。

推论 *如果一个顶点的高度大于 V，那么在残量网络中不存在从该顶点到汇点的路径。*

证明 如果存在从该顶点到汇点的一条路径，那么性质 22.9 蕴含着那条路径的长度会大于 V，但这是不可能的，因为网络中只有 V 个顶点。■

现在我们理解了这些基本机制，通用的预流 - 推进算法很容易描述。我们从任一高度函数开始，除了那些连接到源点的边充满至其容量外，其他的边赋以 0 流。然后重复以下步骤，直到没有活动顶点存在：

> *选择一个活动顶点。向某些离开那个顶点的合格边（如果存在这样的边）推进流。如果不存在这样的边，则增加顶点的高度。*

我们并没有指定初始的高度函数、如何选择活动顶点、如何选择合格边，或是推进多大的流。我们把这个通用的方法称之为基于边的（edge-based）预流 - 推进算法。

算法依赖高度函数来识别合格边。我们也使用高度函数来证明算法计算了一个最大流，且使用高度函数分析算法的性能。因此，关键的是在算法执行过程中要保证高度函数是有效的。

性质 22.10 *基于边的预流 - 推进算法保持了高度函数的有效性。*

证明 我们仅在不存在满足 $h(u) = h(v) + 1$ 的边 u-v 时才增加 $h(u)$ 的值。也就是说，如果在增加 $h(u)$ 之前，对于所有边 u-v，$h(u) < h(v) + 1$，因而增加之后，$h(u) \leqslant h(v) + 1$。对于任何流入边 w-u，增加 $h(u)$ 并不会影响对应任何其他边上的不等式，我们永远不会增加 $h(t)$（或 $h(s)$）。总之，由这些观察可得声明结果成立。■

所有盈余流都从源点发出。非形式地，通用预流 - 推进算法试图把盈余流推入汇点；如果它不能做到这一点，最终会把该盈余流倒推进源点。它表现为这种形式，是因为在残量网络中有盈余的结点总是保持与源点的连通。

性质 22.11 *当预流 - 推进算法在一个流网络中执行时，在那个流网络的残量网络中存在从每个活动顶点到源点的一条（有向）路径，不存在从源点到汇点的（有向）路径。*

证明 使用归纳法证明。初始时，离开源点的边中有一个流，它被充满至容量，因而那些边的目的顶点是唯一的活动顶点。因为这些边被充满至容量，在残量网络中存在从每个这样的顶点到源点的一条边，且在残量网络中不存在离开源点的边。因此，所声明的性质对于初始流成立。

从每个活动顶点都可达源点，是因为向活动顶点集中添加的唯一方式是把流从活动顶点推进一条合格边中。这个操作在残量网络中留下一条从接收顶点回到活动顶点的边。由归纳假设，从这条边可达源点。

初始时，残量网络中不存在从源点可达的其他结点。第一次结点 u 成为源点可达的结点是在流被沿着 u-s 倒推进时（此时致使 s-u 被添加到残量网络）。但这种情况只能发生在 $h(u)$ 大于 $h(s)$ 时，仅在 $h(u)$ 增加之后发生。因为残量网络中不存在到顶点高度更小的边。同理可证由源点可达的所有结点有更大的高度。但是汇点的高度总是0，因而它由源点不可达。 ■

推论 *在预流－推进算法执行中，顶点高度总是小于 $2V$。*

证明 我们只需要考虑活动顶点，因为每个不活动的顶点的高度要么相同，要么比它上次处于活动时的高度大1。同理采用性质22.9中的方法，从一给定活动顶点到源点的路径蕴含着顶点的高度至多比源点的高度大 $V-2$（t 不可能在这条路径上）。源点的高度不变，且它初始时不会大于 V。因此，活动顶点的高度至多为 $2V-2$，不存在顶点的高度为 $2V$ 或更大。 ■

通用预流－推进算法阐述简单，易于实现。但可能不那么直观的是为什么它能计算出一个最大流。高度函数是确定它可以计算出最大流的关键所在。

性质22.12 *预流－推进算法计算一个最大流。*

证明 首先，我们必须证明算法能够终止。必须表明在某一点不存在活动顶点。一旦我们把一个顶点的所有盈余流推进，直到那个流的某些流被倒推回之前，那个顶点就不能再次成为活动顶点。仅当该顶点的高度增加时才会发生倒推流的情形。如果我们有一个长度无限的活动顶点序列，某些顶点必定出现无限次；原因是如果高度也能无限增长。这与性质22.9的推论相矛盾。

当没有活动顶点时，流是可行的。因为由性质22.11可得，在残量网络中也没有从源点到汇点的路径，根据性质22.5的同样论证，该流是最大流。 ■

可以细化这个证明过程。算法终止且最坏情况下的运行时间为 $O(V^2E)$。我们把细节留作练习（见练习22.67～22.68）。与性质22.13中的更简单的证明类似，可以应用到算法的一个具体的版本。具体地说，我们考虑的实现是基于以下对于迭代更为具体的说明：

> *选择一个活动顶点。向某些离开那个顶点的合格边增加流（如果可能则充满），继续这个过程直到顶点变成不活动的，或者不存在这样的合格边。在后一种情况下，增加顶点的高度。*

也就是说，一旦我们选择一个顶点，就尽可能的推进它的所有流。如果到了仍有盈余流的顶点，但是没有合格边，就增加顶点的高度。我们把这种通用方法称之为*基于顶点的*（vertex-based）预流－推进算法。它是基于边的通用算法的一种特例，在基于边的算法中，我们保持选择相同的活动顶点，直到该顶点变为不活动的，或者用尽了所有离开它的合格边。性质22.12的正确性证明可应用到基于边的通用算法的任何实现中，因而直接可得基于顶点的算法计算一个最大流。

程序22.24是基于顶点的通用算法的一种实现，它对活动顶点使用了一个广义队列。这是上面描述方法的一种直接实现，表示了仅在广义队列ADT实现上有所不同的一类算法。这一实现假设ADT不允许队列中有重复顶点；另外，我们假设可以向程序22.4中添加代码以避免重复入队（见练习22.62～22.63）。

也许对于活动顶点所使用的最简单的数据结构是一个FIFO队列。图22-29显示了算法在示例网络上的运行过程。如同图中所述，非常便利把活动顶点序列分为由*阶段*（phase）组成的序列，其中每一步骤定义为在上一步的队列中的所有顶点被处理完之后队列中的内容。这样做有助于我们界定算法的总运行时间。

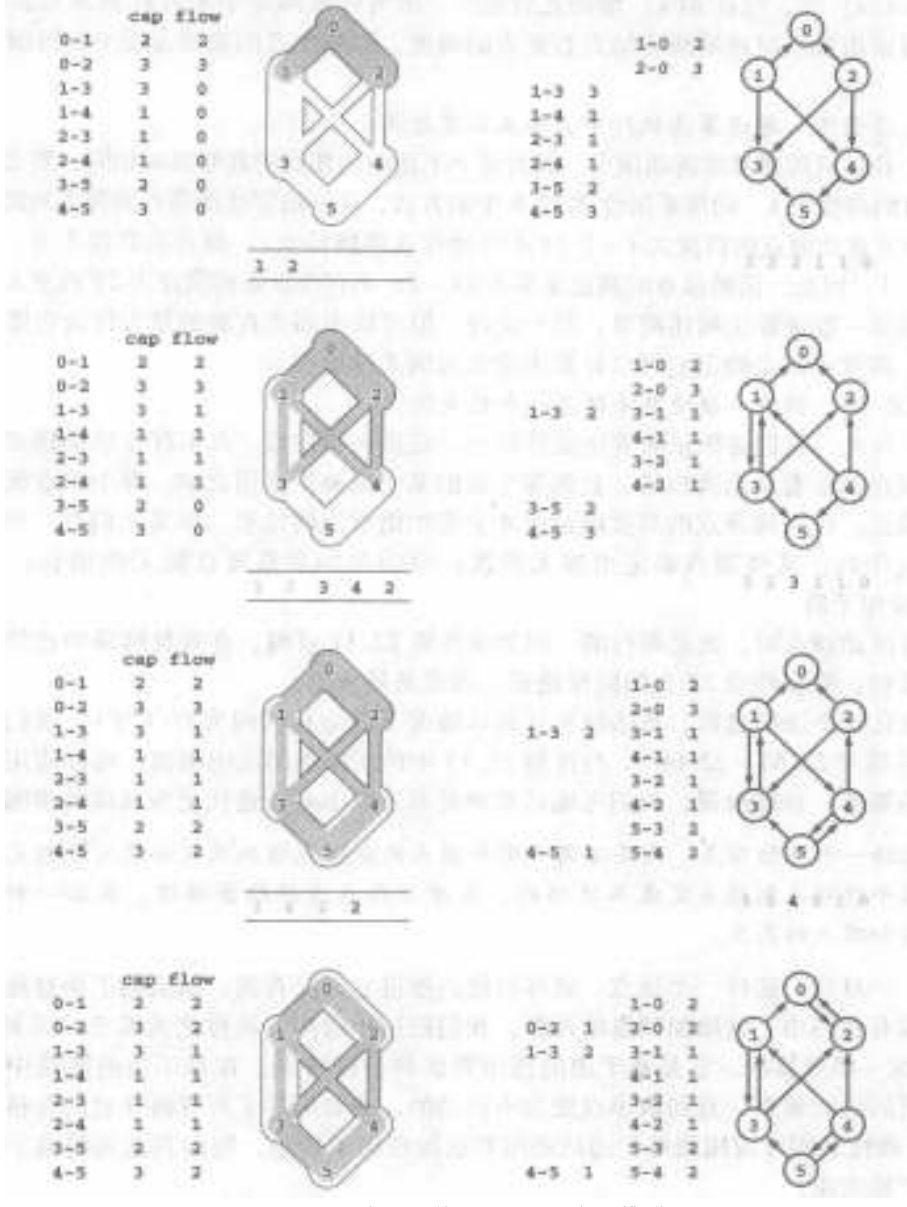


图 22-29 残量网络（FIFO 预流 - 推进）

此图通过一个例子显示了 FIFO 预流 - 推进算法运行过程中的流网络（左图）和残量网络（右图）。队列中的内容显示在流网络的下方，距离标记显示在残量网络的下方。在初始步中，我们通过 0-1 和 0-2 推进流，使得 1 和 2 成为活动顶点。在第二步中，从到 3 和 4 的这两个顶点推进流，使 3 和 4 成为活动结点，1 成为不活动的结点（2 仍然是活动的，且其距离标记增加）。在第三步中，我们通过 3 和 4 把流推进 5，并使 3 和 4 成为不活动的结点（2 仍然是活动的结点，它的距离标记再次增加）。在第四步中，2 是唯一的活动结点，而且由于边 2-0 上的距离标记增加，2-0 是可允许的。沿着 2-0 倒推进一个单位的流来完成计算。

性质22.13 预流－推进算法的FIFO队列实现的最坏情况下的运行时间与V^2E成正比。

证明 我们使用势函数（potential function）来界定步骤数。证明过程是展示我们在第8部分更详细考虑的算法分析和数据结构的强大技术的一个示例。

如果没有活动顶点，则定义数量ϕ为0，否则为活动顶点的最大高度。然后考虑每一步骤对于ϕ值的影响。令$h_0(s)$为源点的初始高度。开始时，$\phi=h_0(s)$；最后，$\phi=0$。

首先，我们注意到顶点高度增加的步数不超过$2V^2-h_0(s)$，由性质22.11的推论，可得V个顶点的高度分别可增加的值至多为$2V$。因为ϕ仅在某个顶点的高度增加时才增加，因而ϕ增加的步骤数不会超过$2V^2-h_0(s)$。

然而，如果在一个步骤中，没有顶点的高度增加，那么ϕ值必定至少减少1，因为步骤的作用是把每个活动顶点的所有盈余流推进到有更小高度的顶点中。

综上，这些事实蕴含着步骤数必定小于$4V^2$：ϕ值开始时为$h_0(s)$，至多增加$2V^2-h_0(s)$次，因而至多减少$2V^2$次。每一步骤的最坏情况是所有顶点都在队列中，要检查它们的所有边。这会导致总运行时间的上界。

这个上界是一个紧致界。图22-30描述了一类流网络，其中预流－推进算法使用的步骤数与V^2成正比。 ■

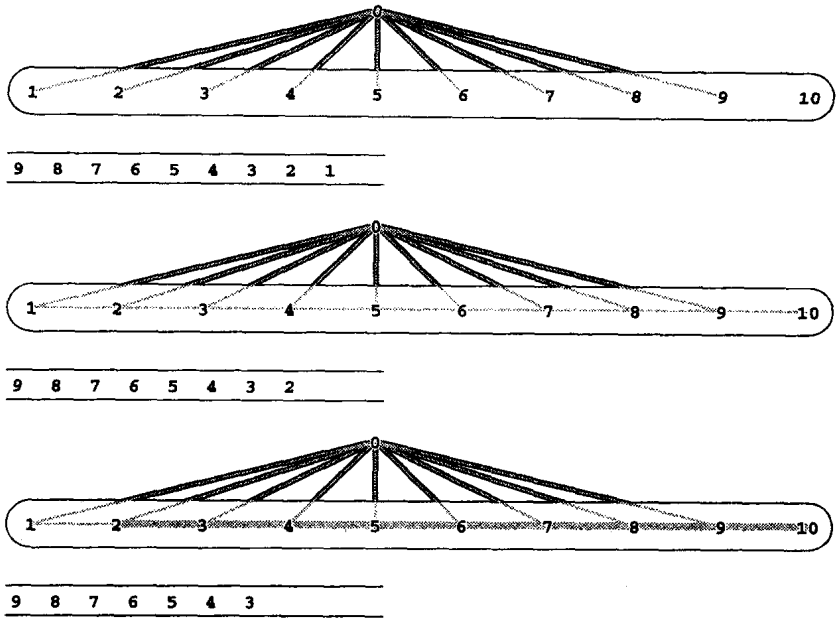


图22-30 FIFO预流－推进最坏情况

这个网络表示V的顶点组成的一类网络，满足预流－推进算法的总运行时间与V^2成正比。它由源点（顶点0）发出的单位容量边和朝向汇点（顶点10）的从左到右容量为$v-2$的水平边组成。在预流－推进算法的初始步（上图），我们从源点向每条边推进一个单位的流，使除源点和汇点之外的所有顶点变成活动状态。在标准的邻接表表示中，它们按照逆序出现在活动顶点的FIFO队列中，显示在网下方。在步骤2中，我们把从9到10推进一个单位的流，使9成为（暂时）不活动的；然后，从8到9推进一个单位的流，使8成为（暂时）不活动的并使9成为活动的；然后，从7到8推进一个单位的流，使7成为（暂时）不活动的并使8成为活动的，依此类推。只剩1是不活动的。在步骤3中（下图），经过一个类似的过程使2成为不活动的，同样的过程继续$V-2$步。

程序 22.4 预流 - 推进最大流实现

这个 ADT 函数实现了通用基于顶点的预流 - 推进最大流算法，对于活动结点使用一个不允许重复的广义队列。

最短路径函数 GRAPHdist 被用于初始化顶点的高度数组 h 和距汇点的最短路径距离。wt 数组包含每个顶点的盈余流，隐含地定义了活动顶点集。按照常规，s 被初始化为活动结点，但不会再加入队列，t 永远不为活动结点。

主循环选择一个活动顶点 v，接着通过它的每条合格边推进流（如果需要，就把接收流的顶点添加到活动表中），直到 v 变成不活动的顶点，或者它的所有边已经都被考虑到。在后一种情况下，v 的高度被增加，并且它返回到队列。

```
static int h[maxV], wt[maxV];
#define P ( Q > wt[v] ? wt[v] : Q )
#define Q (u->cap < 0 ? -u->flow : u->cap - u->flow)
int GRAPHmaxflow(Graph G, int s, int t)
  { int v, w, x;  link u;
    GRAPHdist(G, t, h);
    GQinit();
    for (v = 0; v < G->V; v++) wt[v] = 0;
    GQput(s); wt[s] = maxWT; wt[t] = -maxWT;
    while (!GQempty())
      {
        v = GQget();
        for (u = G->adj[v]; u != NULL; u = u->next)
          if (P > 0 && v == s || h[v] == h[u->v]+1)
            {
              w = u->v; x = P;
              u->flow += x; u->dup->flow -= x;
              wt[v] -= x; wt[w] += x;
              if ((w != s) && (w != t)) GQput(w);
            }
        if ((v != s) && (v != t))
          if (wt[v] > 0) { h[v]++; GQput(v); }
      }
  }
```

因为我们实现维持着残量网络的一个隐含表示，它们检查离开顶点的边，即使当这些边不在残量网络中（测试是否它们在）。我们通过维护一个残量网络的显示表示，来取消这个测试开销，表明可能把性质 22.13 中的上界从V^2E降低为 V^3。虽然理论上的上界是我们见到的最大流问题的最坏情况，这种变化并不太糟，特别是对于实际中看到的稀疏图（见练习 22.64 ~ 22.66）。

这些最坏情况上界往往是悲观的，因而不必用于预测实际网络的性能（不过这种差距不像我们在增大路径算法中所发现的那么大）。例如，FIFO 算法使用了 15 步找出了图 22-31 中描述的网络中的流，而性质 22.13 中证明的上界指出可以在少于 182 步中找出这个流。

为了改进性能，在程序 22.4 中可以使用一个栈、一个随机队列或其他任何广义队列。在实际中被证明很好的一种方法是实现使 GQget 返回最高活动顶点的广义队列。我们把这种方法称为最高顶点（highest-vertex）预流 - 推进最大流算法。我们可以用一个优先队列实现

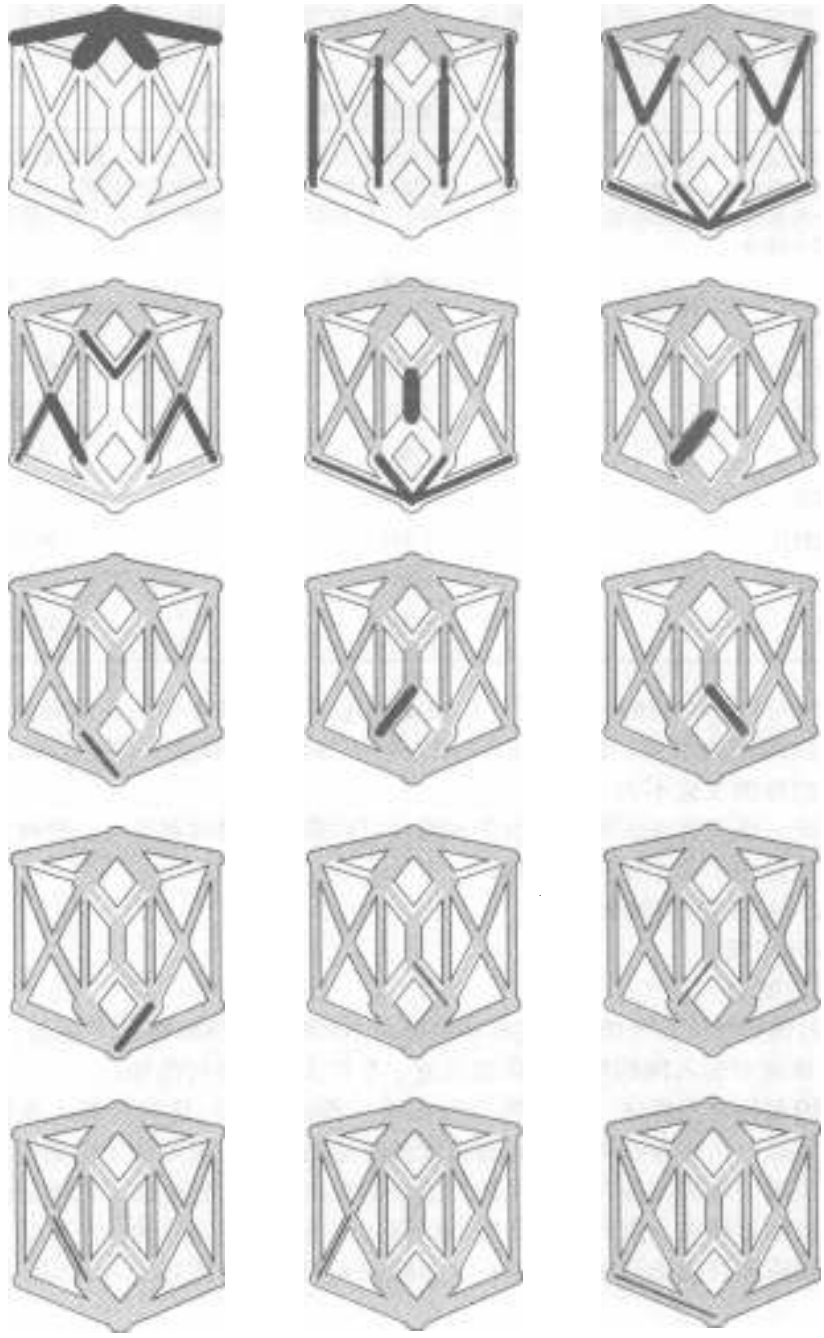


图 22-31 预流 - 推进算法（FIFO）

这个序列描述了使用预流 - 推进方法找出示例网络中最大流的 FIFO 实现。它由 2 步组成：沿着离开源点的边从源点推进尽可能多的流（上图左）。然后，从每个这样的结点推进流，直到所有结点处于平衡。

这一策略，虽然也可能利用高度的特殊性质在常量时间内来实现广义队列操作。已经证明这个算法（见第4部分参看文献）的最坏情况下的上界为$V^2\sqrt{E}$（对于稀疏图此上界为$V^{5/2}$）；如常，这个上界是悲观的。很多其他预流－推进的变型都已提出，其中有几个把最坏情况下的上界降低为VE（见第4部分参考文献）。

表22-2 预流－推进算法的实验研究

此表显示了对于示例欧几里得近邻网（随机容量且最大流值为286）和单位容量网（最大流值为6）各种预流－推进网络流算法的性能参数（扩展的顶点数和涉及的邻接表结点数）。对于这两种类型的网络，这些方法的差异很小。对于随机容量，所考察的边数与随机增大路径算法大约相同（见表22-1）。对于单位容量，增大路径算法对于这些网络所考察的边数要少得多。

	顶点数	边 数
随机容量1-50		
最短路径	2 450	57 746
深度优先	2 476	58 258
随机	2 363	55 470
单位容量		
最短路径	1 192	28 356
深度优先	1 234	29 040
随机	1 390	33 018

对于22.2节讨论的两种网络模型，表22-2显示了与表22-1中增大路径算法性能对应的预流－推进算法的性能结果。比起我们在增大路径方法中的性能，这些实验显示出各种预流－推进算法的性能变化不大。

在开发预流－推进算法实现中有很多可能性可以探讨，同时对于每一种可能性也有很多选择。我们已经讨论了3个主要的选择：

- 基于边与基于顶点的通用算法
- 广义队列实现
- 初始高度赋值

还有几个其他选择可以考虑，它们都导致很多不同的算法值得研究（例如，练习22.57～22.61）。算法性能对输入网的特征的依赖性进一步扩大了这种可能性。

我们讨论的两种通用算法（增大路径和预流－推进算法）是关于最大流算法的众多文献中最重要的算法。由于存在对于更好的最大流算法的需求，因此对于它的研究仍然是一个可能出成果的领域。解决实际问题的快速算法的实现以及存在解决最大流问题的简单线性算法的可能性都激励着研究人员开发和研究新的算法。直到发现这样的一个算法，我们才能够很有信心地使用前面讨论的算法和实现；大量研究表明这些算法对于大量实际的最大流问题是有效的。

练习

▷ **22.51** 对于容量处于平衡状态的网络，描述预流－推进算法的执行过程。

22.52 使用本节中描述的概念（高度函数、合格边，以及通过边推进流），描述增大路径最大流算法。

22.53 按照图22-29的样式，显示使用FIFO预流－推进算法来找图22-11中所示的流网络

中的一个最大流时，每步的流网络和残量网络。

22.54 对于最高顶点预流 - 推进算法做练习 22.53。

○ 22.55 修改程序 22.4，把广义队列实现为优先队列来实现最高顶点预流 - 推进算法。进行实验测试在表 22-2 中增加一行关于这个变型算法的信息。

22.56 对于各种特定网络实例（见练习 22.7 ~ 22.12），当执行 FIFO 预流 - 推进算法时，画出残量网络中活动顶点数、顶点数和边数的图。

○ 22.57 使用合格边的广义队列，实现通用基于边的预流 - 推进算法。对于各种网络进行实验研究（见练习 22.7 ~ 22.12），来比较这些算法与基本方法，并对性能更好的算法进行更为详尽的研究。更详尽地研究性能更好的算法的广义队列实现。

22.58 修改程序 22.4，从而定期地重新计算残量网络中顶点高度为到汇点的最短路径长度。

○ 22.59 把顶点的盈残量平均到其发出边中，而不是只使某些边充满，使其他边为空的思想，评价这一推进顶点盈残量的思路。

22.60 对于不同的网络（见练习 22.7 ~ 22.12）进行实验研究，通过对程序 22.4 所给出的性能以及高度函数均初始化为 0 时的性能加以比较，从而确定程序 22.4 中初始高度函数的最短路径计算是否正确。

○ 22.61 对结合以上思路的混合方法进行实验研究。对于不同的网络（见练习 22.7 ~ 22.12）进行实验，将这种方法与基本方法进行比较，并对性能更好的算法进行更为详尽的研究。

22.62 修改程序 22.4 的实现，使其显式地禁止广义队列中的重复顶点。进行实验研究测试各种网络（见练习 22.7 ~ 22.12），以确定你的修改对于实际运行时间的影响。

22.63 如果广义队列中允许重复顶点，这对于性质 22.13 中的最坏情况下的运行时间上界有何影响？

22.64 修改程序 22.4 中的实现，使之维持残量网络的一种显式表示。

○ 22.65 对于练习 22.64 中的实现，把性质 22.13 的上界缩小到 $O(V^3)$。提示：证明以下两种情况的各自上界，一是与残量网络中边删除所对应的推进数，二是不能导致满边或空边的推进数。

22.66 对于各种网络（见练习 22.7 ~ 22.12）进行实验研究，来确定使用残量网络的显式表示（见练习 22.64）对于实际运行时间的影响。

22.67 对于基于边的通用预流 - 推进算法，证明与残量网络中边删除所对应的推进数小于 $2VE$。假设实现保存残量网络的显式表示。

● 22.68 对于基于边的通用预流 - 推进算法，证明不与残量网络中边删除对应的推进数小于 $4V^2(V+E)$。提示：使用活动顶点的高度和作为势函数。

● 22.69 对于各种网络以及预流 - 推进算法的几个版本（见练习 22.7 ~ 22.12）进行实验研究，来确定实际考虑的边数以及运行时间与 V 的比率。考虑书中以及前面练习中描述的几种算法，并主要集中在那些对于大型稀疏网性能最佳的算法上。把你的结果与练习 22.36 中的结果进行比较。

● 22.70 编写一个客户程序，动态模拟预流 - 推进算法的执行过程。你的程序应该产生类似图 22-31 中的图和本节中的其他形式的图（见练习 22.50）。对于练习 22.7 ~ 22.12 中的欧几里得网测试你的实现。

22.4 最大流归约

在这一节里，我们考虑对最大流问题的一些归约，来说明22.2节和22.3中的最大流算法在广泛的领域中是很重要的。我们可以去掉对于网的各种限制求解其他流问题；可以求解其他网处理问题和图处理问题；还能求解不是网问题的问题。本节专门讨论这些用法的一些例子，把最大流确立为求解问题的一般模型。

我们还讨论最大流问题和更困难问题之间的关系，从而为以后讨论这些问题奠定基础。特别地，我们指出最大流问题是最小成本流的问题的一个特例，最小成本流问题是22.5节和22.6节的主题，另外我们将描述如何将最大流问题形式化为LP问题，我们将在第8部分讨论。最小成本流和LP表示求解问题的模型。与求解这个更一般问题的算法相比，可以用22.2节和22.3节的专门算法更容易地求解最大流问题。随着我们研制出更强大算法，认识到求解问题的模型之间的关系就很重要。

我们使用术语*标准最大流问题*（standard maxflow problem）来表示我们所讨论的问题版本（边上有容量的、*st*网中的最大流）。这种用法只是在本节中易于引用。实际上，我们从考虑一些归约开始，这些归约在标准问题中所做的限制基本上无关紧要，因为几个其他的流问题可以归约到或等价于该标准问题。我们可以把任何一个等价问题作为“标准”问题。作为性质22.1的一个结论，这个问题的一个简单例子就是找出网中使某条特定边上流最大的一个环流。接下来，我们将讨论提出此问题的其他方法，在每种情况下，指出它与标准问题的关系。

一般网中的最大流（maxflow in general network） 找出网中使从源点总流出量（相应地使到汇点总流入量）最大的流。按照约定，如果没有源点或汇点，则将此流定义为0。

性质22.14 *一般网的最大流问题等价于st网的最大流问题。*

证明 一般网的最大流算法显然那可以用于*st*网，因此我们只需确定一般问题可以归约为*st*网问题即可。为了做到这一点，首先找出源点和汇点（比如说，可以使用我们程序19.8中初始化队列的方法），如果没有源点或汇点则返回0。然后，添加一个虚拟源点s，以及从s到网中每个源点的边（每条边上的容量设置为那条边的目的顶点的流出量），另外增加一个虚拟汇点t，以及网中每个汇点到t的边（每条边上的容量设置为该边源点的流入量）。图22-32描述了这一归约过程。*st*网的最大流直接对应于原网络中的一个最大流。 ■

顶点容量约束（vertex-capacity constraint） 给定一个流网络，找出满足额外约束条件的一个最大流，使得通过每个顶点的流必定不超过某个固定容量。

性质22.15 *对于顶点上有容量约束的流网络，其最大流问题等价于标准最大流问题。*

证明 同样，我们可以使用求解容量约束问题的任何算法，来求解标准问题（可以把每个顶点上的容量约束设置为大于它的流入量或流出量），因而我们只需证明可归约到标准问题即可。给定一个带有容量约束的流网络，构造一个带有两个顶点u和u^*的标准流网络，其中顶点u和u^*都对应原顶点u，到达原顶点的所有进入边将到达u，所有发出边来自u^*，以及边u-u^*的容量等于该顶点的容量。图22-3描述了这个构造过程。对于转换后的网中的任一最大流，形如u^*-v的边上的流给出原网中的一个最大流，必定满足顶点－容量约束条件，这是由于存在形如u-u^*这样的边。 ■

允许多汇点和多源点或增加容量约束似乎可以推广最大流问题；性质22.14和性质22.15的意义在于，这些问题实际上都不会比标准问题更难。接下来，我们考虑问题的一个可能更容易求解的版本。

无环网（acyclic network） 找出无环网中的最大流问题。计算含有环的流网络的一个最大流是一个更困难的问题吗？

我们已经考察过很多图处理问题的例子，它们在包含环时更难以求解。也许最突出的例子是边上权值可为负数的加权有向图的最短路径问题（见21.7节）。如果图中不存在环，它是一个线性时间可解的简单问题，但是如果图中允许环存在，则它是一个NP-完全问题。但值得注意的是，最大流问题对于无环网不会更容易。

性质22.16 无环网的最大流问题等价于标准最大流问题。

证明 同样，我们只需要说明标准问题可归约为无环问题即可。给定一个有 V 个顶点和 E 条边的网络，我们可以构造一个有 $2V+2$ 个顶点和 $E+3V$ 条边的网络，它不仅不含环，还有一个简单结构。

令 u^* 表示 $u+V$，构造一个二分有向图，其中对应于原网中的各个顶点 u 有两个顶点 u 和 u^*，以及对应原网中的每条边 u-v 则有一条与之有相同容量的边 u-v^*。现在，向二分有向图中添加一个源点 s 和一个汇点 t，且对于原图中的每个顶点 u，边 s-u 和边 u^*-t，其上容量均等于原网中 u 的发出边的容量之和。另外，另 X 是原网中边上的容量之和，增加从 u 到 u^* 的边，其容量为 $X+1$。此构造如图22-34所示。

为了说明原网中任何最大流都对应于转换后的网中的一个最大流，我们要讨论割而不是流。给定原网中规模为 c 的任意 st 割，可以说明如何在转换后的网中构造一个规模为 $c+X$ 的最小割，也可说明如何在原网中构造一个规模为 c 的 st 割。这样在转换后的网中给定一个最小割，原网中的相应割也是最小的割。此外，我们的构造还给出了一个其流值等于最小割容量的流。因此这是一个最大流。

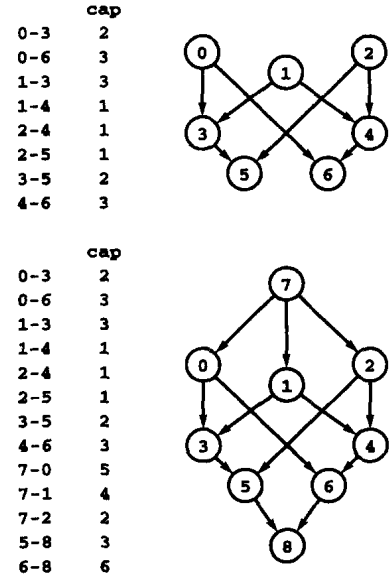


图22-32 从多源点和多汇点进行归约

上图中的网中有3个源点（0、1和2），两个汇点（5和6）。为了找出使源点流出量和汇点流入量最大的流，我们找出下图所描述的 st 网中的一个最大流。这个网除了新添加的源点7和汇点8，其他与原网络一样。从7到原网中的每个源点的边上的容量等于那个源点的流出边上容量之和，从原网中的每个汇点到8的边上的容量等于那个汇点的流入边上容量之和。

给定原网中的任意割，该割把源点与汇点分开。令 S 为源点的顶点集，T 为汇点的顶点集。为转换后的网构造一个割，其中将 S 中的顶点放在一个含有 s 的集合中，将 T 中的顶点放在一个含有 t 的集合中，另外对于所有 u，将 u 和 u^* 置于割的同一边，如图22-34所示。对于每个 u，要么 s-u 在割集中，要么 $P|u^*$-$t|$ 在割集中，u-v^* 在割集中当且仅当 u-v 在原网中的割集中；因而割的总容量等于原网中割的容量加上 X。

给定转换后的网的任意最小 st 割，令 S^* 为 s 所在的顶点集，T^* 为 t 所在的顶点集。我们的目标是构造一个有相同容量的割，且对于所有 u，u 和 u^* 都在同一个割顶点集中，这样由前一段中的对应性可以得到原网中的一个割，从而完成证明。首先，如果 u 在 S^* 中且 u^* 在 T^* 中，那么 u-u^* 必定是一条交叉边，这是一个矛盾：u-u^* 不会在任何最小割中，因为由对应原图中边的所有边组成的割具有较小成本。其次，如果 u 在 T^* 中且 u^* 在 S^* 中，那么 s-u 必定在割中，因为那是连接 s 到 u 的唯一的边。但是我们可以用 s-u 代替由 u 发出的所有边，从而将 u 移至 S^* 来建立有相同成本的割。

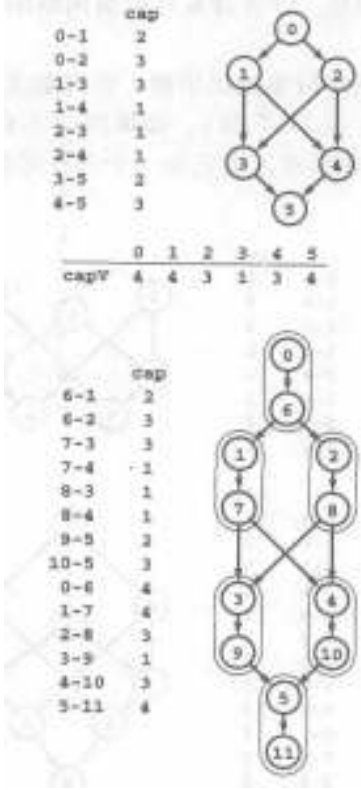


图 22-33 删除顶点容量

为了求解找出上图网的最大流问题，使通过每个顶点的流不超过顶点索引数组 capV 给出的容量界限，我们构建下图的标准网：将一个新的 u^*（这里 u^* 表示 $u+V$）与每个顶点 u 关联，添加一条边 u-u^*，其容量是 u 的容量，且对于每条边 u-v，加上一条边 u^*-v。每一对 u-u^* 在图中均由环所包围。下图的网中，任何流都直接对应于上图网中的满足顶点容量约束的一个流。

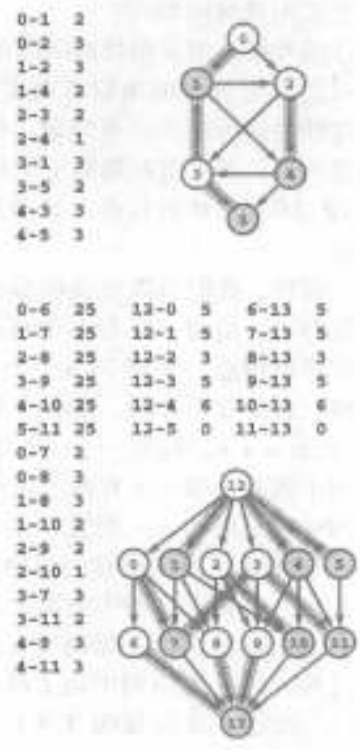


图 22-34 归约到无环网

上图中网中的每个顶点 u 对应下图中网中的两个顶点 u 和 u^*（u^* 表示 $u+V$），上图中网中的每条边 u-v 对应下图中网中的边 u-v^*。此外，下图中网有未加容量的边 u-u^*，源点 s 到每个未加星号的顶点都有一条边，源点 t 从每个未加星号的顶点都有一条边。阴影和非阴影的顶点（以及将阴影与非阴影连接的边）显示了两个网中割之间的直接关系（见正文）。

给定转换网中值为 $c+X$ 的任意流，可以简单地将相同的流值赋给原网中对应的各条边，来得到值为 c 的一个流。前一段最后的割转换并不影响这个赋值，因为它处理流值为 0 的边。■

该归约的结果不仅是一个无环网，而是还是一个简单二分结构。归约指出如果我们愿意，那么完全可以采用这些更为简单的网，而不是一般网作为我们的标准。似乎这种特殊结构会得到一个更快的最大流算法。但是归约表明，我们可以将在这些特殊无环网中找到的任何算法用于求解一般网的最大流问题。实际上，经典最大流算法利用了一般网模型的灵活性：我们讨论过的增大路径方法和预流 - 推进方法使用残量网络的概念，把环引入网中。在对于无环网中的最大流问题，我们一般使用一般网中的标准算法来求解它。

性质 22.16 的构造相当精巧，且它说明了要完成归约证明，即便不是天才，也需要非常仔细。这种证明很重要，原因在于并非所有版本的最大流问题都等价于标准问题，且需要知道我们的算法的应用程度如何。研究人员还在研究这个内容，因为将不同的自然问题相关联

的归约尚未建立，如下例所示。

无向网中的最大流（maxflow in undirected networks） 一个无向流网络是一个加权图，其中边上整数权值解释为容量。这种网中的循环流是指定边上的权值和方向满足以下条件：各条边上的流不大于该边上的容量。进入每个顶点上的总流量等于从该顶点出去的总流量。无向最大流问题是要找出一个循环流，从而使某条边（也就是说，从某个顶点 s 到某个顶点 t）上指定方向的流达到最大。这个问题可能比标准问题更自然地对应于通过管道内的液体流动模型：它对应于允许液体在管道内在任意方向上流动。

性质 22.17 *无向 st 网中的最大流问题可归约到 st 网的最大流问题。*

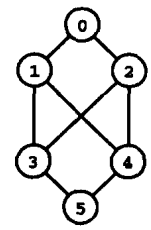
证明 给定一个无向网，构造一个有相同顶点的有向图，而且对应于原网的各条边，在两个方向上都有边，其上容量均为原无向边上的容量。原网中的任何流必定对应于转换后网中的具有相同值的一个流。反过来也成立：如果在无向网中，u-v 有流 f，且 v-u 有流 g，那么，如果 $f \geqslant g$，则有向网中 u-v 的流为 $f-g$；否则 v-u 的流为 $g-f$。因此，有向网中的任一最大流都是无向网中的一个最大流：构造过程给出了一个最大流，并且在有向网中的具有更大值的任何流都将对应于无向网中有更大值的某个流；但是并不存在这样的流。 ■

这个证明并没有确立无向网中的问题与标准问题等价。也就是说，找到一个无向网中的最大流是否比找到标准网中最大流问题要简单，是一个开放的问题（见练习 22.83）。

总之，利用前两节中关于 *st* 网的最大流算法，我们可以处理有多个汇点和源点的网、无向网、在顶点上由容量约束的网，另外还可以处理许多其他类型的网（例如，见练习 22.81）。事实上，性质 22.16 说明了即便使用一个仅无环网的算法，我们也可以解决所有这些问题。

接下来，我们将讨论另一个问题，它并不是一个显式的最大流问题，但是可以将它归约到最大流问题，因此可以利用最大流算法来解决。这是对本章开始处描述的商品配送问题进行形式化描述的一种方法。

	cap
0-1	2
0-2	3
1-3	3
1-4	1
2-3	1
2-4	1
3-5	2
4-5	3



	cap
0-1	2
1-0	2
0-2	3
2-0	3
1-3	3
3-1	3
1-4	1
4-1	1
2-3	1
3-2	1
2-4	1
4-2	1
3-5	2
5-3	2
4-5	3
5-4	3

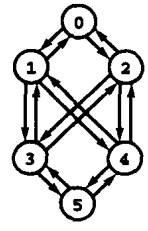


图 22-35 从无向网中归约

为了求解无向网中的最大流问题，我们可以将此网看作是每个方向都有边的有向网。

可行流（feasible flow） 假设在一个流网络中为每个顶点赋予一个权值，并将此解释为供应（如果为正）或需求（如果为负），而且顶点权值之和为 0。定义一个流为可行流（feasible），如果每个顶点的流出量与流入量之差等于那个顶点的权值（如果为正，则为供应；为负，则为需求）。给定这样的一个网，确定是否存在一个可行流。图 22-36 显示了一个可行流问题。

供应顶点对应于商品配送问题中的仓库：需求顶点对应于零售店，而边则对应于运输线路中的道路。可行流问题可以回答一个基本问题：是否有可能找出一种运货的方法从而无论如何都可使供应满足需求。

性质 22.18 *可行流问题可归约到最大流问题。*

证明 给定一个可行流问题，构造一个具有相同顶点和边但是顶点上无权值的网。此外，添加一个与每个供应顶点都有边相连的源点 s，边上的权值为该顶点的供应量，添加一个到每

个需求顶点都有边相连的汇点 t，边上的权值为该顶点的需求量取负（从而边权值为正）。求解这个网中的最大流问题。原网中有一个可行流，当且仅当在一个流中从源点的所有流出边以及到汇点的所有流入边都被充满至容量。图 22-37 显示了这个归约的一个例子。 ■

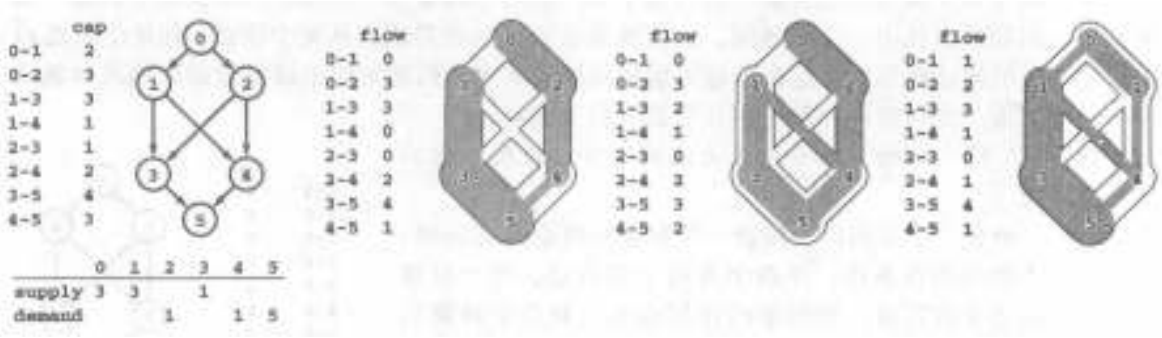


图 22-36 可行流

在一个可行流问题中，除了边上的容量约束之外，我们还指定顶点上的供应和需求约束。在供应顶点上寻找其流入量等于供应加上流入量的任何流，在需求顶点上寻找其流入量等于流出量加上需求的任何流。对于左图的这个可行流问题有三个解，都在右图中显示出来。

开发实现我们所考虑的这种类型归约的 ADT 函数是软件工程中的一项挑战性的任务。主要原因是我们所操纵的对象是用复杂数据结构表示的。为了把另一个问题归约为标准最大流问题，需要建立一个新的网吗？某些问题要求额外数据，比如说顶点容量或供应量和需求量，没有这些数据时建立一个标准网可能是合理的。但如果我们这样做并且计算出一个最大流，结果将如何处理？把一个计算出的流从一个网转换到另一个网，当这两个网都是用邻接表表示时并不是一个简单的计算。如果我们不做网的复制，而向网中添加虚拟顶点并计算一个最大流肯定会破坏原有的结构；因而我们需要在计算之后，仔细把网恢复到它的原始状态。程序 22.5 使用性质 22.16 中的归约，对于可行流问题用网 ADT 函数描述了这些考虑的一种实现。

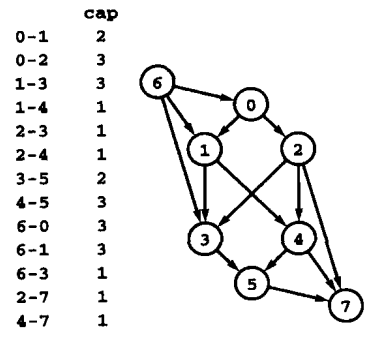


图 22-37 从可行流归约

这个网是一个标准的网，由图 22-36 中的可行流问题通过添加从一个新源点到供应顶点的边（每条边上的容量等于供应量）和从需求顶点到一个新汇点的边（每条边上的容量等于需求量）构造而成。图 22-36 中的网存在一个可行流，当且仅当此网中有一个流（最大流）使从汇点发出和到源点的所有边都充满。

用最大流模型我们不能处理的流问题的一个典型例子是可行流问题的一个扩展，这也是我们在 22.5 节和 22.6 节讨论的主题。再添加一个解释为成本的边权值集合，根据这些权值来定义流成本，要求一个最小成本可行流。这个模型形式化了一般的商品配送问题。我们不仅对它是否可转运商品感兴趣，还对转运它们的最小成本的方法感兴趣。

到目前为止，本节所讨论的所有问题都有一个相同的基本目标（即计算一个流网络中的一个流），因此可以利用一个解问题的流网络模型来处理这些问题。正如我们在最大流 - 最小割定理中所见的那样，可以使用最大流算法来解决看上去与流无关的图处理问题。我们现在来考虑这种类型的例子。

最大基数二份匹配（maximum-cardinality bipartite matching） 给定一个二分图，找出最大基数的的一个边集，满足每个顶点至多连接到一个其他顶点。

为简明起见，除了需要与类似的问题加以区分时，我们把这个问题都简单称为二分匹配（bipartite-matching）问题。它将本章开始所讨论的职位安置问题加以形式化。顶点对应个体和雇员；边对应“职位中相互感兴趣的”关系。二分匹配问题的一个解是总雇用最大。图22-38显示了对图22-3中的示例问题建模的二分图。

程序22.5 归约到最大流的可行流

使用图22-37中描述的构造方法，这个ADT函数通过归约到最大流问题来求解可行流问题。它假设图的ADT表示有一个顶点索引的数组sd，满足sd为正，则表示顶点i的供应量，若为负，则表示顶点i的需求量。

如在构造中所指出的，我们添加从虚拟结点s到供应结点和从需求结点到另一虚拟结点t的边，接着找出一个最大流，然后检查是否所有额外边被填充到容量，再去除所有额外边。

这段代码假设图表示的顶点索引的数组大得足以容纳虚拟结点。它还使用了一个捷径，就是依赖于这种表示法来去除边，而不释放对应所添加和删除的边的表结点的内存空间。

```
void insertSTlinks(Graph G, int s, int t)
  { int i, sd;
    for (i = 0; i < G->V; i++)
      if ((sd = G->sd[i]) >= 0)
        GRAPHinsertE(G, EDGE(s, i, sd, 0, 0));
    for (i = 0; i < G->V; i++)
      if ((sd = G->sd[i]) < 0)
        GRAPHinsertE(G, EDGE(i, t, -sd, 0, 0));
  }
void removeSTlinks(Graph G)
  { int i;
    for (i = 0; i < G->V; i++)
      G->adj[i] = G->adj[i]->next;
  }
int GRAPHfeasible(Graph G)
  { int s = G->V, t = G->V+1, sd = 0; link u;
    insertSTlinks(G, s, t); G->V += 2;
    GRAPHmaxflow(G, s, t);
    for (u = G->adj[s]; u != NULL; u = u->next)
      sd += u->cap - u->flow;
    for (u = G->adj[t]; u != NULL; u = u->next)
      sd += u->cap - u->flow;
    G->V -= 2; removeSTlinks(G);
    return sd;
  }
```

为二分匹配问题找出一个直接解，而不是用图模型，是一个很有意义的练习。例如，问题相当于以下组合谜题：“找出整数对集合的最大子集（可由不相交集而得），满足性质：其中任何两对都没有相同的整数”。图22-38中所示的例子对应于基于数对0-6、0-7、0-8、1-6等求解此谜题。问题乍一看似乎很简单，但是正如17.7节中所讨论的哈密顿回路问题（及许多其他问题一样），系统地选择整数对的简单方法直到遇到矛盾为止，可能需要指数时间。也就是说，如果要穷尽所有可能性，则有太多的整数对要检查；因而，聪明的解决方法是只需检查其中很少一部分。要解决类似于以上给出的特定匹配谜题，或者开发可以高效地解决此类谜题的算法，这些都不是简单的任务，它们有助于说明网络流的强大功能和用

途，这也为进行二分匹配提供了一个合理的方法。

性质 22.19 二分匹配问题可归约到最大流问题。

证明 给定一个二分匹配问题，通过为所有边指定从一个集合到另一个集合的方向，并添加一个源点以及该源点指向二分图中其中一个集合中的所有顶点的边，再添加一个汇点以及从二分图中另一个集合中的所有顶点指向该汇点的边，来构造最大流问题的一个实例。为使结果有向图是一个网，为每条边赋以容量 1。图 22-39 显示了这个构造。

现在，对于此网的最大流问题的解给出了对应的二分匹配问题的一个解。该匹配恰好对应这两个集合中的顶点之间的由最大流算法充满到容量的那些边。首先，网络流总是提供一个合法的匹配：因为每个顶点有一条容量为 1 的边，要么是进入边（来自源点），要么是发出边（到达汇点）。因此，通过每个顶点至多只有一个单位的流，这反过来说明每个顶点在匹配中最多被包含一次。其次，不存在有多条边的匹配，因为较之于最大流算法所生成的流，任何这样的匹配都将直接导致一个更好的流。 ■

例如，在图 22-39 中，一个增大路径最大流算法可能使用路径 s-0-6-t，s-1-7-t，s-2-8-t，s-4-9-t，s-5-10-t 和 s-3-6-0-7-1-11-t 来计算匹配 0-7，1-11，2-8，3-6，0-7 和 1-11。因此，在图 22-3 中有一种方法可使所有学生和职位匹配。

程序 22.6 是一个客户程序，从标准输入读取一个二分匹配问题，并使用证明中描述的归约方法来求解它。对于大型网此程序的运行时间是多少？可以肯定，运行时间依赖于我们使用的最大流算法及其实现。同时，我们需要考虑我们构建的网具有特殊的结构（单位容量二分流网络）——不仅可以使我们考虑过的各种流网络的运行时间不一定接近其最坏情况的上界，而且可大大地降低这个界限。例如，对于一般增大路径算法，我们考虑的第一个界限给出了一个答案：

推论 在一个二分图中找最大基数匹配所需要的时间为 $O(VE)$。

证明 由性质 22.6 直接可得。 ■

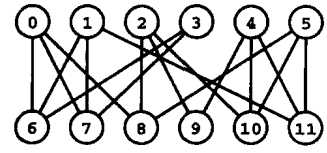
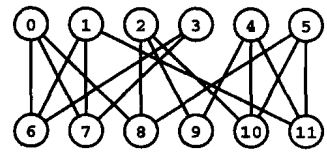


图 22-38 二分匹配

这个二分匹配问题的实例是对图 22-3 中所示的职位安置例子的形式化描述。在该例中找出一种最佳方法将学生和职位加以匹配，这等价于在此二分图找出顶点不相交的边的最大数目。



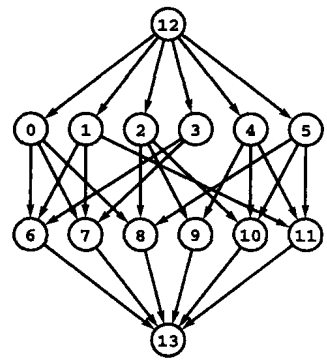


图 22-39 从二分匹配归约

为了找出一个二分图中的最大匹配（上图），通过使所有边均为从上面一行指向下面一行，添加一个新源点以及到上面一行中每个顶点的边，再添加一个汇点以及从下面一行中每个顶点到该汇点的边，并为所有边上的容量赋值为 1，来构造一个 st 网（下图）。在任何一个流中，上面一行中的每个顶点至多有一个发出边能被充满，下面一行中的每个顶点至多有一个进入边能被充满，因此此网的最大流问题的一个解给出了二分图的一个最大匹配。

程序 22.6 经由归约到最大流的二分匹配

这个客户程序从标准输入读取一个有 $V+V$ 个顶点和 E 条边的二分匹配问题，然后构造一个对应这个二分匹配问题的流网络，找出网中的最大流，并使用这个解打印一个最大二分匹配。

```
#include <stdio.h>
#include "GRAPH.h"
main(int argc, char *argv[])
  { Graph G; int i, v, w, E, V = atoi(argv[1]);
    G = GRAPHinit(2*V+2);
    for (i = 1; i <= V; i++)
      GRAPHinsertE(G, EDGE(0, i, 1, 0));
    while (scanf("%d %d", &v, &w) != EOF)
      GRAPHinsertE(G, EDGE(v, w, 1, 0));
    for (i = V+1; i <= V+V; i++)
      GRAPHinsertE(G, EDGE(i, V+V+1, 1, 0));
    if (GRAPHmaxflow(G, 0, V+V+1) == 0) return;
    E = GRAPHedges(a, G);
    for (i = 0; i < E; i++)
      if ((a[i].v != 0) && (a[i].w != V+V+1))
        if (a[i].flow == 1)
          printf("%d-%d\n", a[i].v, a[i].w);
  }
```

当增大路径算法用于单位容量的二分网上时，它的操作描述很简单。每条增大路径将从源点发出的一条边和进入汇点的一条边充满。这些边从来不用做回边，因而至多有 V 条增大路径。对于任何与边 E 成正比的时间找到增大路径的算法，此 VE 上界都成立。

表 22-3 显示了使用各种增大路径算法求解随机二分匹配问题的性能结果。由此表可得，此问题的实际运行时间近似于最坏情况时间 VE，而不是最优时间（线性时间）。可能通过明智的选择以及优化的最大流算法的实现，来使这种方法的速度提高一个$\sqrt{V}$因子（见练习 22.93 和练习 22.94）。

表 22-3 二分匹配算法的实验研究

此表对于有 2 000 个顶点，500 条边（上面）和 4 000 条边（下面）的图，显示了使用各种增大路径最大流算法来计算一个最大二分匹配的性能参数（扩展的顶点数和涉及的邻接表结点数）。对于这个问题，深度优先搜索是最有效的方法。

	顶点数	边 数
500 条边，匹配基数 347		
最短路径	1 071	945 599
最大容量	1 067	868 407
深度优先	1 073	477 601
随机	1 073	644 070
4 000 条边，匹配基数 971		
最短路径	3 483	8 280 585
最大容量	6 857	6 573 560
深度优先	34 109	1 266 146
随机	3 569	4 310 656

此问题代表了我们考察新问题时经常会遇到的一种情况和更一般的问题求解模型，也表明归约作为实际求解问题工具的有效性。如果能够找到一个到已知一般模型（比如说最大流问题的模型）的归约，我们会将其看作为开发一个实用解的一个主要步骤，因为它至少

表明了不仅那个问题是易解的，而且还有很多求解这个问题的高效算法。在很多情况下，使用现有最大流 ADT 函数来求解这个问题是合理的，再转向下一个问题。如果性能依然是一个关键的问题，我们可以研究各种最大流算法或实现的相对性能，也可以使用它们的行为作为开发更好算法、特殊用途的算法的起点。一般求解问题的模型为我们提供了可选择的或改进的上界，此外还提供了大量实现。已经证实这些实现对于大量其他问题都是有效的。

接下来，我们将讨论与图的连通性有关的问题。在考虑将最大流算法用于求解连通性问题之前，我们考察将最大流 - 最小割定理用于第 18 章中未完成的工作：与无向图中路径和割有关的基本定理的证明。这些证明是进一步表明了最大流 - 最小割定理的重要性。

性质 22.20（Menger 定理） *在有向图中删除某条边使两个顶点不连通的最少边数等于这两个顶点之间边不相交的路径的最大数目。*

证明 给定一个有向图，用同一组顶点和边定义一个流网络，其中所有边的容量均定义为 1。由性质 22.2 可得，我们可以把任何一个 *st* 流表示为从 s 到 t 的边不相交的路径集合，这样的路径数目等于流值。任何一个 *st* 割的容量等于那个割的基数。给出以上事实，最大流 - 最小割定理蕴含了定理所述结果。 ■

无向图中的相应结果，以及有向图和无向图中的顶点连通性都涉及这里所讨论的归约，这些我们留作练习（见练习 22.96 到练习 22.98）。

现在我们转到流和连通性之间的直接关系得算法学含义（最小割问题归约到最大流问题），这是由最大流定理所确定的。性质 22.5 可能是最重要的算法学含义，但是反过来还不不确定是否成立（见练习 22.49）。直观来看，如果知道一个最小割会使找出一个最大流的任务变得更容易一些。但没有人能够证明如何做到这一点。这个基本例子强调了处理问题之间的归约时务必谨慎。

然而，我们还可以使用最大流算法来处理很多连通性问题。例如，它们有助于解决在第 18 章中遇到的第一个非平凡的图处理问题。

边连通度（edge connectivity） 把一个给定的图分割为两部分所需去除的最少边数是多少？找出进行此分割的一个最小基数的边集。

顶点连通度（vertex connectivity） 把一个给定的图分割为两部分所需去除的最少顶点数是多少？找出进行此分割的一个最小基数的顶点集。

这些问题对于有向图也是相关的，因此共有 4 个问题要考虑。如我们在 Menger 定理中所做的那样，我们详细讨论其中的一个（无向图中边的连通性），其他三个问题留作练习。

性质 22.21 *确定无向图的边的连通性所需要的时间为 $O(E^2)$。*

证明 我们通过计算一个 *st* 网的最大流来计算分割两组给定顶点的任一最小割的大小。该网是由对图的每条边上赋以单位容量构造而得。边的连通度等于所有顶点对的值的最小值。

但我们并不需要计算出所有顶点对。令 s^* 是图中度最小的顶点。注意 s^* 的度不大于 $2E/V$。考虑图的任一最小割。按照定义，割集合中的边数等于图的边连通度。顶点 s^* 出现在割的某个顶点集中，另一集合必定含有某个顶点 t，因而任何一个分割 s^* 和 t 的最小割的大小必定等于该图的边连通度。于是，如果我们可以求解 $V-1$ 的最大流问题（使用 s^* 作为源点，其他一个顶点作为汇点），所找到的最小流值就是网的边连通度。

现在，任何一个以 s^* 作为源点的增大路径最大流算法至多使用 $2E/V$ 条路径；因此，如果我们使用任何至多采用 E 步来找出一条增大路径的方法，就至多使用 $(V-1)(2E/V)E$ 步来找出边连通度，这蕴含着结果成立。 ■

这种方法不像本节中的其他例子，并不是直接把一个问题归约到另一个问题，而是给出了一种计算边连通度的实用算法。而且仔细实现最大流可以调节到这个特殊的问题，我们可以改进性能，可以在与 VE 成正比的时间内解决这个问题（见本部分参考文献）。性质 22.21 的证明是我们在 21.7 节首次遇到的有效归约（多项式时间）的更一般概念的例子。它在第 8 部分所讨论的算法理论中起着重要的作用。这样的归约及证明了问题是易解的，同时又给出了一种求解它的算法。这是处理新的组合问题的重要一步。

我们使用线性规划（LP，见 21.6 节）来考虑最大流问题的一个严格的形式化描述来结束本节的讨论。这个练习是有用的，因为它有助于我们弄清楚与那些能这样形式化的其他问题的关系。

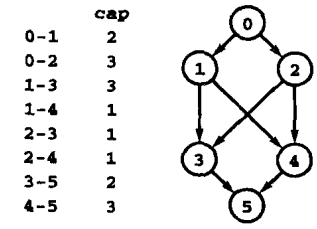
形式化是直接的：我们考虑一个不等式组，其中每个变量对应一条边，两个不等式对应一条边，一个方程对应一个顶点。变量的值是边流，不等式指定了边流必定位于 0 和边的容量之间，等式指定了进入那个顶点的边上的总流必定等于离开那个顶点的边上的总流。

图 22-40 显示了这个构造过程的一个例子。任何最大流问题都可以按照这种方法转换成一个 LP 问题。LP 是一种求解组合问题的很有用的方法，我们研究的大量问题都可以形式化为线性规划问题。最大流问题比线性规划问题更容易求解的事实可以解释为：最大流问题的 LP 形式化中的约束条件具有特殊的结构，它们并不必在所有 LP 问题中出现。

即使一般的 LP 问题比诸如最大流问题等特定问题更为困难，仍有一些强大的算法可以高效地解决 LP 问题。这些算法的最坏情况运行时间肯定超过我们所讨论的特定算法的最坏情况运行时间，但是由数十年来的大量实践经验可知，它们对于解决实际中出现的各种类型的问题是有效的。

图 22-40 描述的构造过程给出了一个证明：最大流问题可归约到 LP 问题，除非我们认为流值是整数。当在第 8 部分详细讨论 LP 问题时，我们会描述一种方法，能够克服 LP 形式化时不支持结果有整数值的限制的困难。

以上内容为我们提供了精确的数学框架以解决更为一般的问题，也可以用来创建功能更为强大的算法来解决这些问题。最大流问题很容易解决，而且其本身使用也相当广泛，正如在本书中的例子所示。接下来，我们将讨论一个更难的问题（仍比 LP 要简单），它涵盖了一类更为广泛的问题。我们将在本章最后讨论这些更为一般的求解问题模型的构建方法，并在第 8 部分进行更为全面的考虑。



最大值 x_{50}
约束条件

$$
\begin{aligned}
x_{01} &\le 2 \\
x_{02} &\le 3 \\
x_{13} &\le 3 \\
x_{14} &\le 1 \\
x_{23} &\le 1 \\
x_{24} &\le 1 \\
x_{35} &\le 2 \\
x_{45} &\le 3 \\
x_{50} &= x_{01} + x_{02} \\
x_{01} &= x_{13} + x_{14} \\
x_{02} &= x_{23} + x_{24} \\
x_{13} + x_{23} &= x_{35} \\
x_{14} + x_{24} &= x_{45} \\
x_{35} + x_{45} &= x_{50}
\end{aligned}
$$

图 22-40 最大流问题的 LP 形式化

这个线性规划等价于图 22-5 中的示例网络的最大流问题。每个不等式对应一条边，指定了流不能超过的容量，每个等式对应一个顶点，指定了每个顶点上的流入量必须等于流出量。我们使用一个从汇点到源点的虚拟边来捕获网中的流，如后面的性质 22.2 中所述。

练习

▷ **22.71** 定义一个网的 ADT 函数，找出一个循环流，其中某条边上具有最大流。使用

GRAPHmaxflow 给出一种实现。

22.72 定义一个网的 ADT 函数，找出网中的一个最大流，其中对于源点或汇点数不加限制。使用 GRAPHmaxflow 给出一种实现。

22.73 定义一个网的 ADT 函数，找出无向网中的一个最大流。使用 GRAPHmaxflow 给出一种实现。

22.74 定义一个网的 ADT 函数，找出网中的一个最大流，其中对于顶点有容量约束。使用 GRAPHmaxflow 给出一种实现。

22.75 开发可行流问题的一个 ADT。使 ADT 函数允许客户来初始化供应 - 需求值，以及一个辅助函数来检查流值是否正确地关联到每个顶点。

22.76 对于每个分布点有有限容量的情况（也就是说，在任何给定时间所存储的商品量有所限制），做练习 22.20。

▷ 22.77 证明最大流问题归约到可行流问题，这两个问题是等价的。

22.78 找出图 22-11 中所示的流网络中的一个可行流，给定额外约束条件：0、2 和 3 是权值为 4 的供应顶点，1、4 和 5 是权值分别为 1、3 和 5 的供应顶点。

22.79 修改程序 22.5，释放对应被添加或删除的边的表结点的内存。

○ 22.80 编写一个程序，以体育联赛的赛程和当前排名作为输入，来确定一个给定的小组是否被淘汰。假设不存在平局。提示：可归约为可行流问题，其中有一个源点其供应值等于此赛季剩余的比赛场数，一些对应于某些赛对的汇点，其需求值等于该赛对剩余的比赛数，还有一些对应于各个赛对的配送结点。边将供应结点和每个赛对的配送结点连接起来，如果 X 要赢得剩余所有比赛，此边上的容量等于该对必须取胜 X 的比赛数目。还会有一条容量不限的边将每个赛队的配送结点与涉及那个赛队的需求结点连接起来。

● 22.81 证明边上有下界的网的最大流问题可以归约为标准最大流问题。

▷ 22.82 证明对于边上有下界的网，找（关于那个下界的）最小流的问题可以归约为最大流问题（见练习 22.81）。

●●● 22.83 证明 *st* 网的最大流问题可归约到无向网的最大流问题，或找出无向网的一个最大流的最坏情况下的运行时间要比那些 22.2 节和 22.3 节中的算法好很多。

▷ 22.84 对于图 22-38 中的二分图，找出有 5 条边的所有匹配。

22.85 扩展程序 22.6，使用符号名替代整数来引用顶点（见程序 17.10）。

○ 22.86 证明二分匹配问题等价于找其上所有边上有单位容量的网的最大流问题。

22.87 我们可能将图 22-3 中的例子解释为描述学生找工作时的偏好，以及雇主对学生的偏好，这两者可能不是相互的。书中描述的归约可以应用到有此解释而得的有向二分匹配问题吗？其中二分图中从一个集合到另一个集合的边是有向的（两个方向的任意一个方向）。证明它可以，或者给出一个反例。

○ 22.88 构造一系列二分匹配问题，其中用于求解对应的最大流问题的任一增大路径算法所使用的增大路径的平均长度与 E 成正比。

22.89 按照图 22-29 的风格，显示在图 22-39 所示的二分匹配网上，FIFO 预流 - 推进网络流算法的操作过程。

○ 22.90 扩展表 22-3，使其包括各种预流推进算法。

● 22.91 假设二分匹配问题中的两个集合大小分别为 S 和 T，$S \ll T$。对于性质 22.19 的归约和 Ford-Fulkerson 算法的最大增大路径实现（见性质 22.8），给出求解此问题的最坏情况运行时间的尽可能精确的一个上界。

• 22.92 对于预流 - 推进算法的 FIFO - 队列实现（见性质 22.13），完成练习 22.91。

22.93 扩展表 22-3，使其包括练习 22.39 中描述的所有增大路径方法的实现。

•• 22.94 证明练习 22.93 中描述的方法对于 BFS 的运行时间为 $O(\sqrt{VE})$。

○ 22.95 进行实验研究，对于一个 $V+V$ 个顶点和 E 条边的随机二分图，画出最大匹配的期望边数的变化曲线，对于 V 的合理大小值以及足够的 E，画出一条从 0 到 V 的光滑曲线。

○ 22.96 对于无向图，证明 Menger 定理（性质 22.20）。

• 22.97 若有向图中删除某些顶点使图中的两个顶点不连接，证明这种顶点的最小数目等于两个顶点之间的顶点不相交路径的最大数。提示：使用顶点分裂变换，类似于在图 22.33 中描述的方法。

• 22.98 扩展你的练习 22.97 中的证明，使其应用到无向图上。

22.99 实现本节描述的边连通度算法，作为第 17 章中图 ADT 的一个 ADT 函数，返回一个给定的图的连通度。

22.100 扩展你在练习 22.99 中的解，放入一个用户提供的最小数组，它是分割图的边的最小集合。用户应该为数组分配多大空间？

• 22.101 开发一个计算有向图的边连通度的算法（即所需删除的边的最小数目，删除边之后使得有向图不是强连通的）。将你的算法实现为第 19 章的有向图 ADT 的一个 ADT 函数。

• 22.102 基于练习 22.97 和练习 22.98 中的解，开发一个计算有向图和无向图的顶点连通度的算法。将你的算法分别实现为第 19 章的有向图 ADT 和第 17 章的图 ADT 的 ADT 函数（见练习 22.99 和练习 22.100）。

22.103 描述如何通过求解 $V \lg V$ 的单位容量的最大流问题，找出一个有向图的顶点连通度。提示：使用 Menger 定理和二叉查找。

○ 22.104 根据练习 22.99 的解来进行实验研究，确定各种图的边连通度（见练习 17.63 ~ 76）。

▷ 22.105 给出图 22-11 中所示的流网络的找最大流问题的 LP 形式化描述。

○ 22.106 将图 22-38 中的二分匹配问题形式化为 LP 问题。

22.5 最小成本流

对于给定的一个最大流问题有很多种解决方案是很正常的。这一事实导致一个问题，我们是否希望强加一些额外准则来选择其中的一个。例如，对于图 22-23 中所示的单位容量的流问题显然有很多种求解方法；也许我们希望选择出使用最少边数或最短路径的一种方法，或是想要知道是否存在包含不相交路径的一个解。这样的问题要比标准最大流问题更难一些；它们属于一种称之为*最小成本流问题*（mincost-flow problem）的更一般模型。这是本节的所讨论的主题。

与最大流问题一样，有很多等价的方式来提出最小成本流问题。我们在本节详细考虑一种标准的形式化描述，然后在 22.7 节考虑各种归约。

具体地说，我们使用*最小成本流模型*（mincost-maxflow model）：扩展在 22.1 节中定义的流网络，允许边上有整数成本，并以自然的方式使用边的成本来定义流成本，然后求出具有最小成本的最大流。我们知道，此问题不仅有高效且有效的算法，而且此问题解决模型也有广泛的应用性。

定义 22.7 *在一个带有边成本的流网络中，一条边上的流成本（flow cost）为那条边上的流和成本的乘积。一个流的成本为该流中边的流成本之和。*

我们仍假设容量为小于 M 的正整数。我们还假设边成本为小于 C 的非负整数。（不允许

负成本的主要原因是基于方便的考虑，像在22.7节讨论的那样）。与以往一样，我们为这些上界值指定了名字，因为一些算法的运行时间取决于后者。有了这些基本假设，将很容易定义所要解决的问题。

最小成本最大流（mincost maxflow） 给定一个带有边成本的流网络，找出一个最大流使得不存在有更小成本的其他任何最大流。

对于一个带有成本的流网络，图22-41显示了带有成本的一个流网络中的不同的最大流。可以肯定地说，较之于我们在22.2节和22.3节中讨论的最大流的计算负担，最小化成本的计算负担并不比前者难度更小。实际上，成本增加了表示重要的新的挑战的另一面。即便如此，利用一个类似于对于最大流问题的增大路径算法的通用算法我们可以承受这个负担。

许多其他问题可以归约或等价于最小成本流问题。例如，以下的形式化描述令我们感兴趣，因为它涵盖了本章开始我们所讨论的商品配送问题。

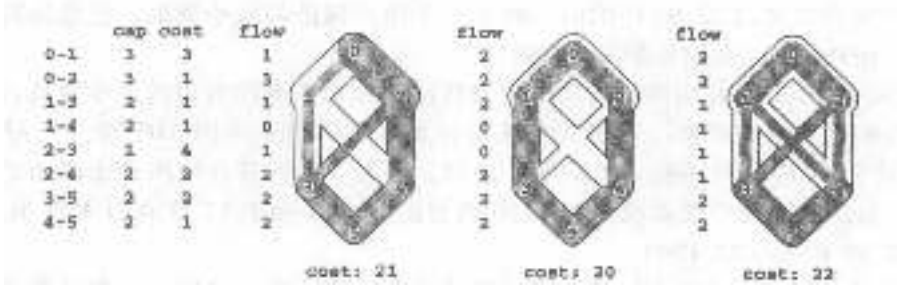


图22-41 带有成本的流网络中的最大流

这些流的（最大）流值都相同，但是它们的成本（边上流和成本乘积之和）不同。中图的最大流成本最小（不存在更小成本的最大流）。

最小成本可行流（mincost feasible flow） 假设在一个带有边成本的流网络中，每个顶点被赋以一个权值，解释为供应（为正时）或需求（为负时），且顶点上的权值之和为0。回忆我们对于流的定义：如果每个顶点上的流出量和流入量之差等于该顶点的权值，则称流是可行的（feasible）。给定这样一个网络，找出具有最小成本的一个可行流。

为了描述最小成本可行流问题的网络模型，我们使用术语配送网来简洁表示“边带有成本和顶点带有供应或需求权值的有容量的流网络”。

在商品配送应用中，供应顶点对应仓库，需求顶点对应零售店，边对应运输路线，供应或需求值对应允送或接受的货物量，另外边容量对应为不同路线上可用的卡车数和载货量。一种解释每条边上成本的自然方式是单位流通过该边上的成本（沿着相应的路径发送一个单位货物量的成本）。给定一个流，它的成本是网络中移动的成本的一部分，是能够归到那条边上的成本。给定要沿着一条给定边运送的货物量，可以通过将每单位成本与数量相乘而计算出运送这些货物的成本。对于每条边上进行这一计算，并将结果相加得到总运送成本，我们希望使这个量达到最小。

性质22.22 最小成本可行流问题和最小成本最大流问题是等价的。

证明 由性质22.18及练习22.77直接而得。 ■

由于这种等价性，同时由于最小成本可行流问题能够直接建立商品配送问题和许多其他应用的模型，因此在关于这两个问题的上下文中，我们使用术语最小成本流（minicost flow）代表这两个问题。我们在22.7节考虑才其他的归约。

为了实现流网络中的边成本，我们为22.1节中的EDGE类增加了一个整数型的cost域。程序22.7是计算一个流成本的ADT函数。和在最大流中的作用一样，也是需要谨慎实现的一个ADT函数，用来检查每个顶点上的流入量值和流出量值是否正确地关联以及数据结构是否一致（见练习22.107）。

开发求解最小成本流问题算法的第一步是扩展残量网络的定义，使边上含有成本。

定义22.8 给定带有边成本的流网络的一个流，该流的残量网络（residual network）中的顶点和原网中一样，而且对于原网络中的每条边，在残量网络中有一或两条边。定义如下：对于原网中的每条边u-v，令f是该流，c是容量，x是成本。如果f为正，则在残量网络中有一条带有容量f、成本$-x$的边；如果f为负，则在残量网络中有一条带有容量c-f、成本x的边。

程序22.7 计算流成本

这个ADT函数返回一个网络流的成本。它将正容量的每条边上的成本与流相乘再求和。

```
int GRAPHcost(Graph G)
  { int i; link u; int cost = 0;
    for (i = 0; i < G->V; i++)
      for (u = G->adj[i]; u != NULL; u = u->next)
        if ((u->cap > 0) && (u->cost != C))
          cost += (u->flow)*(u->cost);
    return cost;
  }
```

这一定义几乎等同于定义22.4，但是差异是很关键的。残量网络中表示后向边的边有负（negative）成本。遍历这些边对应于删除原网络中对应边上的流，因而成本一定会相应地减小。由于边上的负成本，这些网络可以有负成本的环。当我们在最短路径算法中首次遇到负环的概念时，似乎是一个人工的概念。而在最小成本流算法中将起着关键的作用。我们考虑两种算法，它们都是基于以下优化条件。

性质22.23 最大流是一个最小成本的最大流，当且仅当它的残量网络中不含负成本的（有向）环。

证明 假设我们有一个其残量网络包含负成本环的最小成本流。令x是那个环中具有最小容量的边上的容量。将流中对应残量网络中正成本的边（前向边）增加x，并将对应于残量网络中负成本的边（后向边）减去x，从而扩展流。这些调整不会影响到任何顶点上的流入量与流出量之差，因而流仍是一个最大流，但它们将网中的成本改变为x乘以环的成本，此为负值，与前面断言原始流的成本最小相矛盾。

为证明充分性，假设有一个不含负成本环的最大流F，其成本不是最小的，考虑任何一个最小成本最大流M。使用流分解定理（性质22.2）一样的论证，我们可以找到至多E个有向环，使得向流F中添加这些环得到流M。但由于F不含负环，这个操作不能减小F的成本，这是一个矛盾。换句话说，我们应该能够通过沿着环增大将F转换成M，但实际上做不到。因为我们没有负成本环用来使此流成本减小。 ■

这个性质直接得到一个简单求解最小成本流问题的通用算法，称为消环（cycle-canceling）算法。

找一个最大流。在残量网中沿着任何负成本的环来增大流，继续这一过程，直到不存在负成本的环。

这种方法将我们在本章以及以前开发的算法集成在一起，提供了求解最小成本流模型的更广泛一类问题的有效算法。就像我们已经看到的几种其他的通用方法，它允许几种不同的实现，因为找初始最大流和找负成本环的方法不是特定的。图 22-42 显示了一个使用消环算法来计算最小成本 - 最大流的例子。

我们可以通过添加一条从源点到汇点的虚拟边，并给它赋以大于网络中任何源点 - 汇点路径上的成本（例如 $VC+1$）及大于最大流的一个流（例如，大于源点的流出量）来去掉消环算法中初始最大流的计算。经过这样的初始设置，消环就能从虚拟边去掉尽可能多的流，因此所得流是一个最大流。图 22-43 描述了使用这种技术计算最小成本流的过程。在图中，我们使用了一个等于最大流的初始流，从而表示此算法只不过是计算另一个有相同值的流，但它有更小的成本（一般而言，我们并不知道此流值，因此最后在虚拟边上还会有一些流，我们将忽略它）。从图中可以明显看出，有些增大的环包含了此虚拟边并会增加网中的流，而另一些不包括虚拟边，并会减少成本。最终，我们得到一个最大流；所有增大的环减少了成本而不改变流的值，与开始的最大流相同。

程序 22.8 消环算法

这个 ADT 函数通过消去具有负成本的环来求解最小成本 - 最大流问题。它使用了 ADT 函数 GRAPHmaxflow 来找出一个最大流，然后用 ADT 函数 GRAPHnegcycle 找出负环，此函数是练习 21.134 中描述函数的一个版本。这里做了修改以适合本章使用的约定，而 st 数组中是到结点的链接而不是索引。在负环存在时，这段代码找出这个负环，计算出推进它的最大流量，并完成操作。这个过程减少最小值的成本。

```
void addflow(link u, int d)
  { u->flow += d; u->dup->flow -=d; }
int GRAPHmincost(Graph G, int s, int t)
  { int d, x, w; link u, st[maxV];
    GRAPHmaxflow(G, s, t);
    while ((x = GRAPHnegcycle(G, st)) != -1)
      {
        u = st[x]; d = Q;
        for (w = u->dup->v; w != x; w = u->dup->v)
          { u = st[w]; d = ( Q > d ? d : Q ); }
        u = st[x]; addflow(u, d);
        for (w = u->dup->v; w != x; w = u->dup->v)
          { u = st[w]; addflow(u, d); }
      }
    return GRAPHcost(G);
  }
```

技术上，使用一个虚拟流初始化较之使用一个最大流初始化进行消环，既不会增加也不会减少一般性。前者的确包含所有增大路径最大流算法，但使用一个最大流算法并不能计算出所有的最大流（见练习 22.42）。一方面，通过使用这项技术，我们可能放弃了复杂最大流算法的一些好处；另一方面，在构建最大流的过程中，可以更好地降低成本。实际上，虚拟流初始化得到广泛应用是因为它实现起来很简单。

至于最大流，这种通用算法的存在性保证了每个最小成本流问题（其中容量和成本都是整数）的解，其中的流都是整数；并且算法计算出这样的一个解（见练习 22.110）。给定这一事实，很容易建立任意消环算法需要的时间量的一个上界。

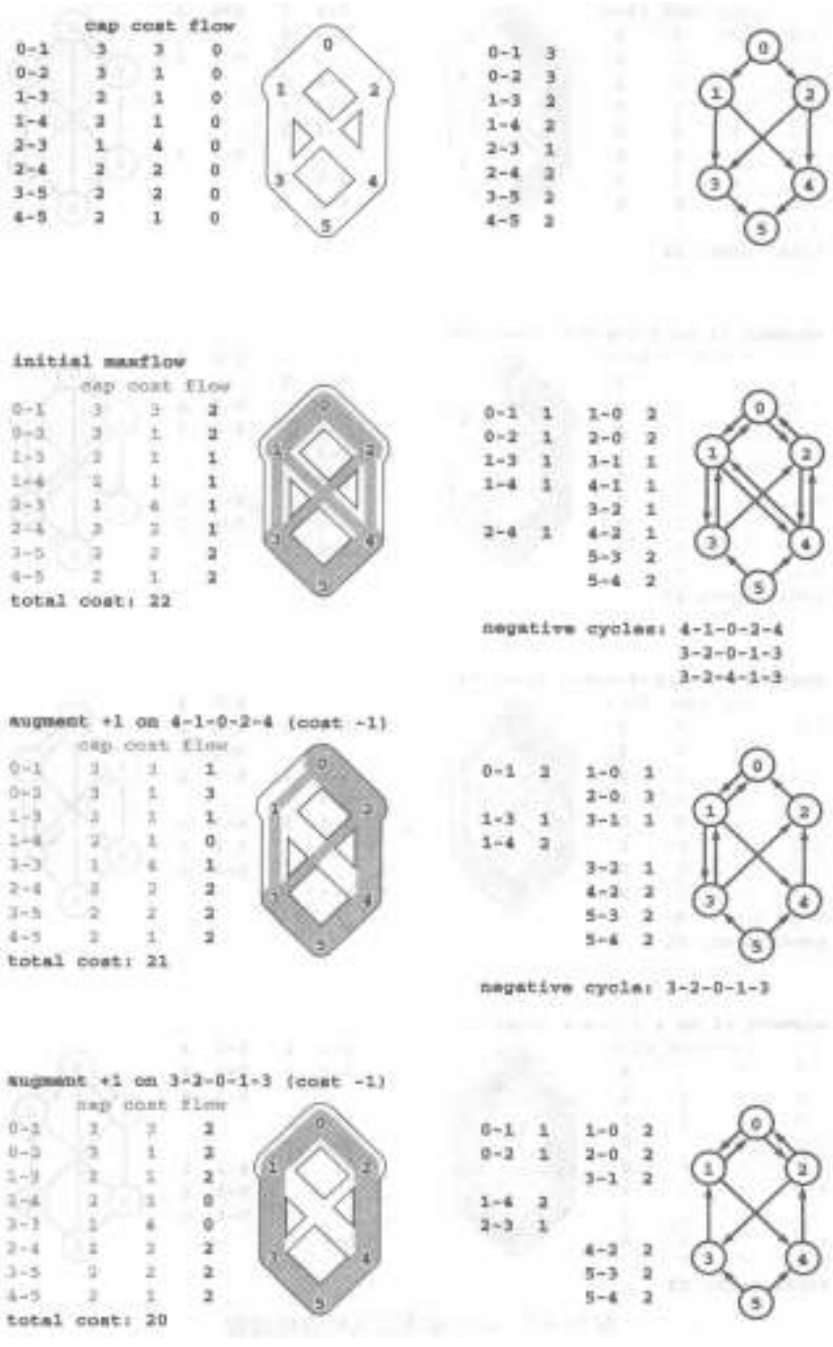


图 22-42 残量网络（消环过程）

这里描述的每个流都是上图描述的流网络的一个最大流，但只有最下面的那个有最小成本最大流。为了找出这个流，我们从任何最大流开始，并沿着负环增大流。初始最大流（从上图数第二个图）的成本为22，它不是一个最小成本最大流，因为残量网络（右端显示出）中有三个负环。在这个例子中，我们沿着4-1-0-2-4增大得到一个成本为21的最大流（从上图数第三个图），它仍然有一个负环。沿着那个环增大得到一个最小成本流（下图）。注意在第一步中沿着3-2-4-1-3增大会在一步之内得到一个最小成本的流。

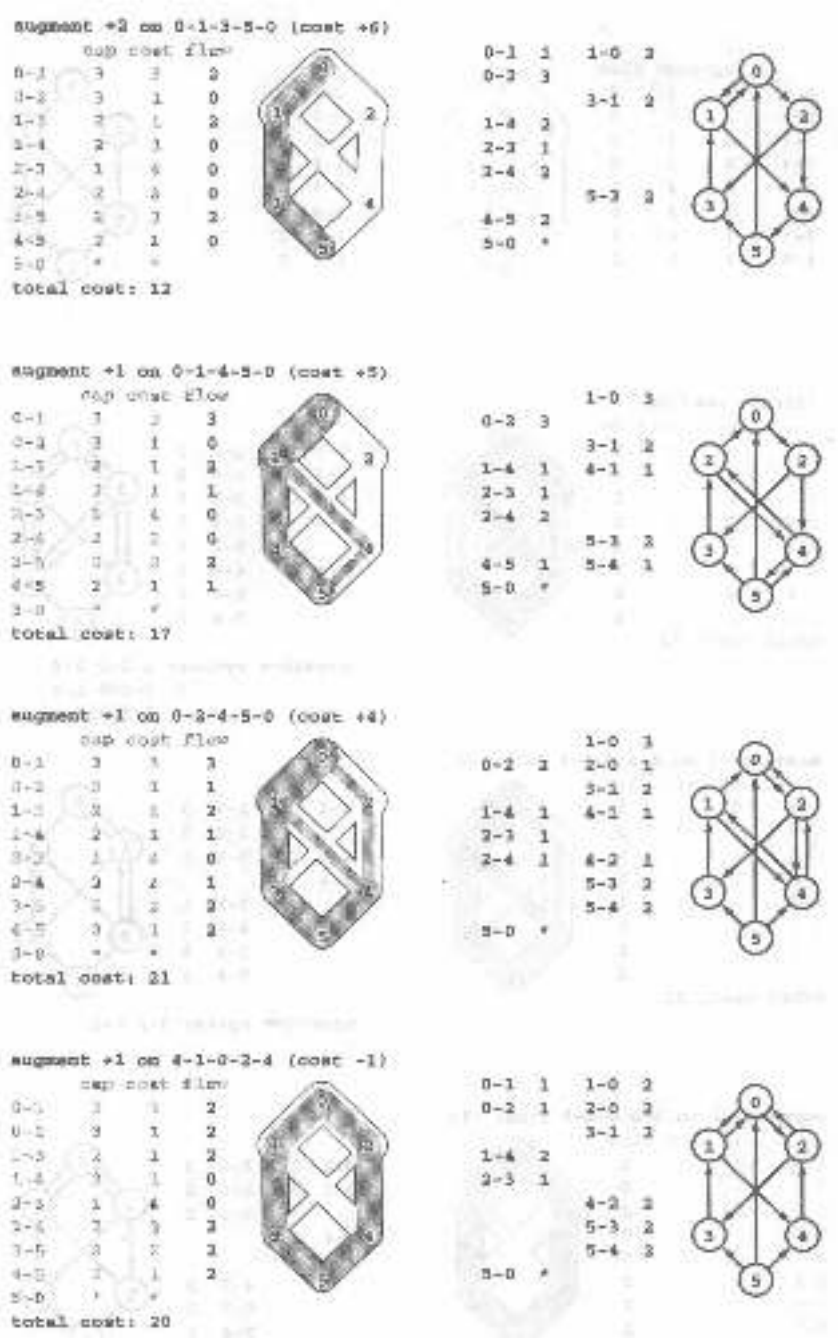


图 22-43 无初始最大流的消环过程

使用消环算法，这个序列描述了从初始为空的一个流计算最小成本最大流的过程。在残量网中使用一条带有无限容量和无限负成本的从汇点到源点的虚拟边。这条虚拟边使得从 0 到 5 的任意增大路径成为一个负环（但在增大并计算此流的成本时忽略它）。这与增大路径算法类似（前三行），沿着这样一条路径增大使流增加。当不存在涉及虚拟边的环时，则在残量网络中不存在从源点到汇点的路径。因而我们得到一个最大流（从上数第三个图）。此时，沿着负环增加使成本较小，同时不改变流值（下图）。在这个例子中，我们计算一个最大流，然后减小它的成本；但是并非总是如此。例如，算法在第二步中可能沿着负环 1-4-5-3-1 而不是 0-1-4-5-0 增大。因为每次增大要么增加流，要么减小成本。我们最终总是得到一个最小成本最大流。

性质22.24 在通用消环算法中所需的增大环的数目少于 ECM。

证明 最坏情况下，在初始最大流中的每条边容量为 M，成本 C，且被充满。每个环将这个成本至少减少1。 ■

推论 在一个稀疏网中求解最小成本流问题所需的时间为 $O(V^3CM)$。

证明 将最坏情况增大环的数目乘以用 Bellman-Ford 算法找出这些环的最坏情况成本直接可得结论（见性质21.22）。 ■

就像增大路径方法，这个运行时间非常悲观。因为它不仅假设使成本达到最小需要使用的大量环的最坏情况，而且还假设了我们必须检查大量的边以找出每个环的另一种最坏情况。在实际很多情况下，我们会使用相对容易找到的相对少量的环，而且消环算法很有效。

可能开发一种寻找负成本环的策略，并且保证所使用的负成本环的数目小于 VE（见第5部分参考文献）。这个结果非常重要，因为它确立了最小成本流问题是易解问题的事实（所有问题可归约到这个问题）。然而，开发人员一般使用允许（理论上）最坏情况的实现，但对于实际问题，迭代次数要比预期的最坏情界限少得多。

最小成本流问题表示我们考察过的求解最一般问题的模型，因此非常令人惊讶能有如此简单的实现来解决这个问题。由于模型的重要性，消环方法以及其他许多不同的方法的很多其他实现都已详尽开发出并得到深入研究。程序22.8一种极其简单且有效的起点，但它有两个缺陷，可能导致坏的性能。第一，每次寻找一个负环时，我们都从头开始。可以在寻找一个负环的过程中将一些中间信息存储起来，以帮助寻找下一个负环吗？第二，程序22.8只是取了 Bellman-Ford 算法找到的第一条负环。可以直接查找带有特殊性质的负环吗？在22.6节中，我们考虑一种改进的实现，仍然是通用实现，但表示了对这两个问题的一种回答。

练习

22.107 修改练习22.13的解，使其检查源点的流出量等于汇点的流入量，以及在每个内部结点上的流出量等于流入量。同时检查对于所有边，成本与流及容量符号相同。对于所有 u 和 v，u-v 的成本和 v-u 的成本之和为0。

22.108 扩展练习22.75中可行流的 ADT，使其包括成本。在求解最小成本－可行流问题中加上一个 ADT 函数，使用标准流网络 ADT 并调用 GRAPHmincost。

▷ 22.109 给定一个其所有边都不是最大容量和成本的流网络，对于最大流成本，给出一个比 ECM 更好的上界。

22.110 证明如果所有容量和成本是整数，那么最小成本流问题有所有流值都为整数的解。

▷ 22.111 修改程序22.8，使用虚拟边上的流进行初始化，而不是计算一个流。

○ 22.112 给出图22-42中所描述的增大环的所有可能序列。

○ 22.113 给出图22-43中所描述的增大环的所有可能序列。

22.114 按照图22-42中的风格，使用程序22.8中的消环算法找出图22-11中显示的流网络的一个最小成本流，显示每次增大后的流和残量网络，其中0-2和0-3的成本为2；2-5和3-5的成本为3；1-4的成本为4；其余边的成本为1。假设使用最短增大路径算法来计算最大流。

22.115 假设修改了程序22.8，如图22-43所示，以从源点到汇点的一条虚拟边上的最大流开始，做练习22.114。

22.116 扩展练习22.7和练习22.8的解使其能够处理流网络中的成本。

22.117 扩展练习22.10到练习22.12，使其在网络中包括成本。每条边的成本取为那条边所连接的顶点之间的欧氏距离。

22.6　网络单纯形算法

消环算法的运行时间不仅基于算法用于降低流成本的负成本环的数目，而且基于算法用于找出每个环的时间。在这一节里，我们考虑一种基本的方法，它不仅极大地减少了识别环的开销，而且允许有降低迭代次数的方法。消环算法的这一实现被称为网络单纯形算法（network simplex algorithm）。它所基于的是维护一个树形数据结构，对成本进行重新加权以使负环能够快速识别。

为了描述网络单纯形算法，我们注意到，对于任意一个流，每个网络边 u-v 处于以下三种状态之一（见图 22-44）：

- 空，因此流只能从 u 到 v 推入
- 满，因此流只能从 v 到 u 推入
- 部分（即不空也不满），流可以双向推入

这个分类类似于我们贯穿本章的残量网络的使用。如果 u-v 是一条空边，那么 u-v 在残量网中，但 v-u 不在残量网中；如果 u-v 是一条满边，那么 v-u 在残量网中，但 u-v 不在残量网中；如果 u-v 是一条部分边，那么 u-v 和 v-u 都在残量网中。

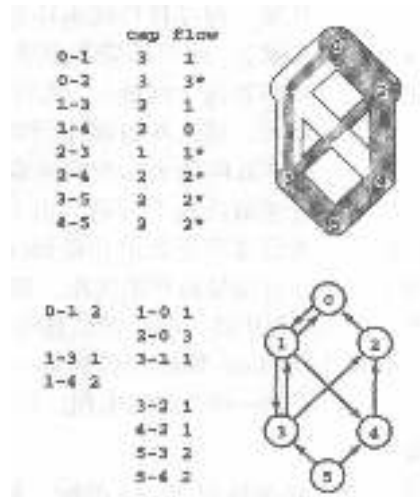


图 22-44　边分类

对于任意流，每条边要么为空，要么为满，要么为部分（即不空也不满）。在这个流中，边 1-4 为空；边 0-2、2-3、2-4、3-5 和边 4-5 为满；边 0-1 和 1-3 为部分。在图中我们的约定给出了识别一条边的状态的两种方法：在流所在列中，0 元素为空边；加星元素为满边；既非 0 也非星的元素为部分边。在残量网中（下图），空边只出现在左列中；满边只出现在右列中；部分边在两列中均出现。

定义 22.9　*给定一个不含部分边环的最大流，这个最大流的一个可行生成树（feasible spanning tree）是网络的一棵生成树，它包含了所有流的部分边。*

在这个定义中，我们忽略了生成树中的方向。也就是说，连接网中 V 个顶点的任意 $V-1$ 条有向边的集合（忽略边的方向）构成了一棵生成树。如果所有非树边要么为空、要么为满，则这棵生成时是可行的。

网络单纯形算法的第一步是构造一棵生成树。构建生成树的一种方法是计算一个最大流，通过沿着每个环增大从而使其边为空或为满，来破坏部分边的环，然后向余下的部分边中添加满边或空边，从而构建一棵生成树。图 22-45 描述了这个过程的一个例子。另一种方法是从源点到汇点的一条虚拟边的最大流开始。接下来，此边为唯一可能的部分边，我们可以使用任何图搜索方法构建这个流的一棵生成树。图 22-46 描述了这样生成树的一个例子。

现在，向一棵生成树中添加任何非树边会造成一个环。网络单纯形算法基本原理是一个顶点权值的集合，这些权值可以很快识别出一些边，当把它们添加到树中时，将在残量网中创建负环。我们称这些顶点权值为势（potential），并用 $\phi(v)$ 表示关联顶点 v 的势。取决于上下文，我们将势表示为定义在顶点上的一个函数，或者是一组整数权值，其中隐含假设了对于每个顶点指定有一个权值，或者也可以表示为顶点索引的数组（因为我们总是按照实现的方式存储它们）。

定义 22.10　*给定带有边成本的流网络中的一个流，令 $c(u, v)$ 表示该流的残量网络中 u-v 的成本。对于任何势函数 ϕ，该残量网络中的边 u-v 关于 ϕ 的降低的成本（reduced cost）用 $c^*(u, v)$ 表示，定义为值 $c(u, v) - (\phi(u) - (\phi v))$。*

换句话说，每条边上降低的成本为该边实际成本与该边顶点势差的差。在商品配送应用中，我们可以看到结点势的直观性：如果将势 $\phi(u)$ 解释为在结点 u 处购买一个单位的商品的成本，那么完全成本 $c^*(u,v)+\phi(u)-\phi(v)$ 就是在结点 u 处购买、运送到 v 并在 v 处销售的成本。

我们使用如下的代码来计算降低的成本：

```
#define costR(e) = e.cost - (phi[e.u] - phi[e.v])
```

也就是说，在维护一个顶点索引的数组 phi 用于存放顶点的势时，不需要存储降低的边成本。因为非常容易计算出它们。

在网络单纯形算法中，我们使用可行生成树来定义顶点的势，使得关于这些势的降低的边成本直接给出关于负成本环的信息。具体地说，我们通过执行算法和设置顶点势的值（使得所有树边可降低的成本为0）来维护一棵可行生成树。

性质 22.25 我们称顶点势的集合关于一棵生成树是有效的（valid），如果所有树边可降低的成本为0。对于任意给定的生成树，所有有效的顶点势隐含着每条网络边具有相同的降低成本。

证明 给定两个不同的势函数 ϕ 和 ϕ'，它们关于一棵给定的生成树都是有效的。我们证明它们之差是一个可加的常数：对于所有 u 及某个常数 Δ，$\phi(u)=\phi'(u)+\Delta$。那么对于所有 u 和 v，$\phi(u)-\phi(v)=\phi'(u)-\phi'(v)$，这表明对于这两个势函数，所有降低的成本都相同。

对于任意通过树边连接的两个顶点 u 和 v，必定有 $\phi(v)=\phi(u)-c(u,v)$。以下证明。如果 u-v 是一条树边，为使降低的成本 $c(u,v)-\phi(u)+\phi(v)$ 为0，那么 $\phi(v)$ 必定等于 $\phi(u)-c(u,v)$；如果 v-u 是一条树边，为使降低的成本 $c(v,u)-\phi(v)+\phi(u)$ 为0，那么 $\phi(v)$ 必定等于 $\phi(u)+c(v,u)=\phi(u)-c(u,v)$。同理对于 ϕ' 也成立。因此我们可得 $\phi'(v)=\phi'(u)-c(u,v)$。

两式相减，对于任何树边连接的 u 和 v，我们可得 $\phi'(v)-\phi'(v)=\phi'(u)-\phi(u)$。对于任何顶点，用 Δ 表示这个差，沿着该生成树的任何搜索树边应用该等式，直接可得结论：对于所有 u，$\phi(u)=\phi'(u)+\Delta$。 ■

还有一种方法来想象定义有效顶点势的集合的过程，那就是从固定的一个值开始，接着计算由树边连接到那个顶点的所有顶点的值，然后计算连接到这些

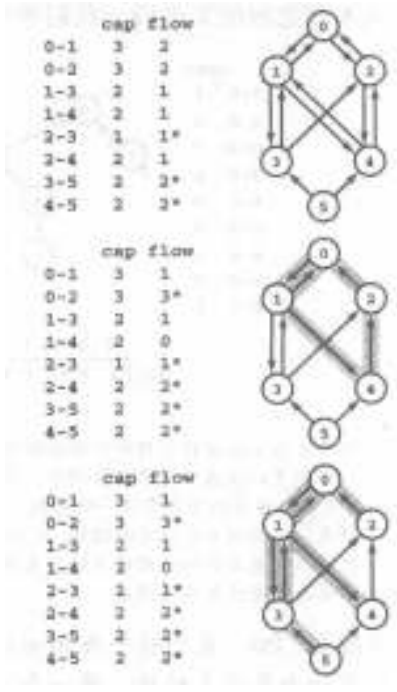


图 22-45 最大流生成树

给定任何最大流（上图）。利用此例描述的两步过程我们可以构造一个有一棵生成树的最大流，满足其中没有非树边是部分边。首先，打破部分边的环：在这种情况下，我们沿着环 0-2-4-1-0 把一个单位的流推入以打破环。使用这种方式总是可以至少是一条边为满或为空；在这种情况下，使 1-4 为空，使 0-2 和 2-4 为满（中图）。第二，我们向部分边的集合中添加空边或满边，以形成一棵生成树：在这种情况下，我们添加 0-2、1-4 和 3-5（下图）。

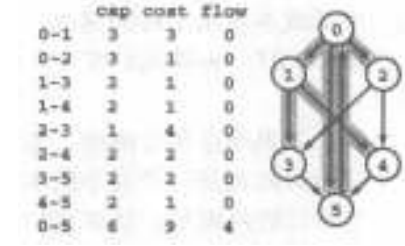


图 22-46 虚拟最大流的生成树

如果从源点到汇点的一条虚拟边上的流开始，那么它是唯一可能的部分边，因此我们可以使用剩余结点的任何生成树来构建此流的一棵生成树。在本例中，边 0-5、0-1、0-2、1-3 和 1-4 包含了初始最大流的一棵生成树。所有非树边都是空的。

顶点的所有顶点的值，依次类推。无论从哪里开始这一过程，任何两个顶点之间的势差都是一样的，它是由树的结构确定的。图 22-47 描述了一个例子。在考察完非树边上降低的成本和负成本环之间的关系后，我们考虑计算势的细节。

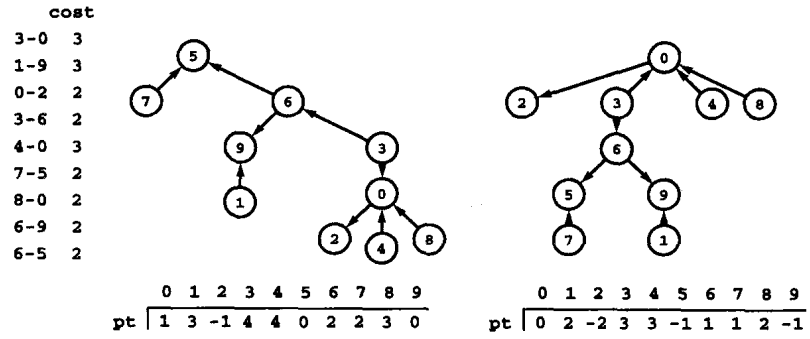


图 22-47　顶点的势

顶点的势是由生成树的结构和初始分配给任意顶点的势值决定的。左图是包含编号 0 到 9 的 10 个顶点的一棵生成树的边集。中图是以顶点 5 为根的该树的一种表示。连接到 5 的顶点要低一层，其他依次类推。我们指定根结点的势值为 0，对于其他顶点的势只有惟一的势值，使得每条边的顶点的势差就等于此成本。右图是以顶点 0 为根的同一棵树的另一种表示。为顶点 0 指定势值 0，我们得到的势值与中图的顶点的势值只差一个常数。我们的所有计算使用了两个势值之差：该差与任何一对顶点的势值差相同，与开始顶点的选择以及为顶点指定的初始势值无关。因此开始顶点和势值的选择不是根本性的问题。

性质 22.26　我们称一条非树边是合格的（eligible），如果在残量网络中，它与树边创建的环是一个负成本的环。称一条边是合格的，当且仅当它是带有可降低成本为正的一条满边，或带有负可降低成本的一条空边。

证明　假设边 u-v 形成的环 t_1-t_2-t_3-…-t_d-t_1，其树边为 t_1-t_2，t_2-t_3，…，其中 v 是 t_1，u 是 t_d。由每条边的降低的成本定义，可得以下结果：

$$
\begin{aligned}
c(u, v) &= c^*(u, v) + \phi(u) - \phi(t_1) \\
c(t_1, t_2) &= \phi(t_1) - \phi(t_2) \\
c(t_2, t_3) &= \phi(t_2) - \phi(t_3) \\
&\cdots \\
c(t_{d-1}, u) &= \phi(t_{d-1}) - \phi(u)
\end{aligned}
$$

这些方程的左边之和给出这个环的总开销，方程的右边之和结果为 $c^*(u, v)$。换句话说，该边的降低成本给出了环的成本，因此只有所描述的这些边才会给出一个负成本的环。 ■

性质 22.27　如果我们有一个流和一棵没有合格边的可行生成树，那么该流是一个最小成本流。

证明　如果不存在合格边，那么在残量网中不存在负成本的环，因此由性质 22.23 的最优性条件可得此流是一个最小成本流。 ■

一个等价的声明是：如果我们有一个流和顶点的势集合，满足树边的降低的成本均为 0，满的非树边都是非负的，空的非树边都是非正的，那么此流是一个最小成本流。

如果我们有合格边，我们可以选择一条边，并用树边沿着它所形成的环增大，来得到一个更小成本的流。如我们在 22.5 节中的消环算法实现的那样，经过环来找到所能推入的最大流量，然后再次经过该环推入那个流量，这至少会使一条边为满或为空。如果这是曾经用于形成环的那条合格边，那么它变成了不合格的边（它的降低成本没变，但是它一个满边变成空边，或从一个空边变成满边）。否则，它成为一条部分边。通过将它添加到树中，并

删除环中的所有满边或空边，可以维持不变式：没有一条非树边是部分边，该树是可行生成树。同样，我们在本节还将考虑这个计算原理。

总之，可行生成树提供了顶点的势，而顶点的势则提供了降低的成本，降低的成本则提供了合格边，合格边则使我们得到负成本环。沿着负成本环增大减少了流成本，同时也蕴含着树结构的改变。树结构的改变则隐含着顶点势的改变；做出所有这些改变之后，我们可以选择另一条合格边，再次开始这一过程。这个求解最小成本流问题的消环算法的通用实现被称为是网络单纯形法（network simplex algorithm）：

> 构建一棵可行生成树并维持顶点的势，使其满足所有树顶点的降低成本为0。向树中添加一条合格边，并沿着用该边与树边形成的环来增大流，从该树中删除满边或空边，继续这一过程，直到不存在合格边。

这一实现是一个通用实现。因为初始生成树的选择，维持顶点势的方法，以及选择合格边的方法均未指定。选择合格边的策略决定了迭代的次数，它是在实现各种策略的不同成本和重新计算顶点势之间的一种权衡。

性质22.28 如果通用网络单纯形算法终止，那么它计算出一个最小成本流。

证明 如果算法终止，之所以这样是由于在残量网中不含负成本的环，因此由性质22.23可得最大流是最小成本的。 ■

算法可能不终止的条件源于这样一种可能性，就是沿着环增大时使很多边满或空，因此留下树中的一些边，使得没有留能够通过这些边推入。如果我们不能推入流，就不能减少成本，还可能使添加和删除边来形成生成树的一个固定序列陷入无限循环。已有几种方法被设计出来以避免这一问题；我们在本节稍后更详细地考虑实现后讨论这些方法。

开发网络单纯形算法的实现的所面临的第一种选择就是如何表示生成树。我们有三种涉及树的主要计算任务：

- 计算顶点的势
- 沿着该环增大（并识别其上的空边或满边）
- 在形成的环上，插入一条新边和删除一条边

每个任务都是数据结构和算法设计的一个有趣的练习。我们可能会考虑几种数据结构和大量算法，它们的性能特征不同。从考虑最简单可用的数据结构开始，就是我们在第一章首次遇到的父链接树表示。我们先讨论基于这种父链接树表示上面列出的任务的算法和实现，然后描述算法在网络单纯形算法中的应用，之后讨论其他一些数据结构和算法。

与本章中的其他几种实现一样，从增大路径最大流的实现开始，我们将链接保存在网络表示中，而不是树表示中的简单索引。这使得在流需要改变时可以访问到流值，同时不会访问不到顶点名。

程序22.9是一种实现，它用与V呈线性的时间指定顶点的势。基于以下思路，图22-48中也做了描述。我们从任意顶点开始，递归地计算其祖先的势，沿着父链接直到树根，按照约定指定根的势为0。然后选择另一个顶点，并使用父链接来递归计算其祖先的势。当我们到达一个其势已知的祖先顶点时，递归终止；然后在跳出递归的过程中，我们沿着路径向下行进，通过相应的父结点来计算每个结点的势。继续这一过程，直到计算了所有结点的势值。一旦遍历完成一条路径，我们不会再次访问这条路径上的边，因此这一过程在与V成正比的时间内运行。

程序 22.9 顶点势的计算

递归函数 phiR 沿着父链接向树上方行进，直到找到其势有效的顶点（约定根的势为 $-C$，总是有效），然后在返回的路径上计算顶点的势，作为递归调用的最后操作。对于计算出势的顶点，通过设置其标志元素为 valid 的当前值来做出标记。

ST 宏允许将树处理函数用于维持父 - 树链接抽象的简洁性，同时可通过网络的邻接表数据结构跟踪链接。

```
#define ST(i) st[i]->dup->v
static int valid, phi[maxV];
int phiR(link st[], int v)
  {
    if (ST(v) == v)
      { mark[v] = valid; return -C; }
    if (mark[v] != valid)
      phi[v] =phiR(st, ST(v)) - st[v]->cost;
    mark[v] = valid;
    return phi[v];
  }
```

给定树中的两个结点，它们的最小公共祖先（least common ancestor，LCA）是包含这两个结点的最小子树的根。通过添加连接两个结点的边所形成的环，它由那条边以及这两个结点到其 LCA 的两条路径上的边组成。为了沿着一个环增大路径，我们并不需要按照顺序考虑环上的边；只要边都考虑到就足够了（两个方向）。因此可以通过这两个结点到其 LCA 的任一路径，沿着环增大。为了沿着通过添加 u-v 所形成的环增大，我们找到 u 和 v 的 LCA（不妨称为 t），并从 u 到 v，从 v 沿着到到 t 的路径，以及从 u 沿着到 t 的路径推入流，但在相反方向每条边都推入流。为了计算推入的流量，我们用同样的方法首先遍历环上的边，来确定能被推入的最大量。程序 22.10 是这种思想的一种实现。其中函数增大一个环，并返回增大过程中被填满或腾空的一条边。

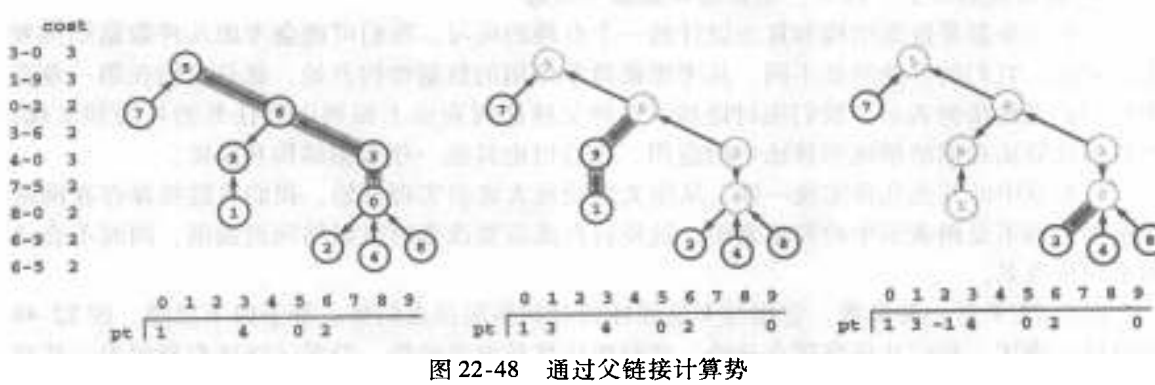


图 22-48 通过父链接计算势

从 0 开始，沿着路径到达根，设置 pt[5] 为 0，然后沿着该路径下行，首先设置 pt[6]，使 pt[6]-pt[5] 等于 6-5 的成本，再设置 p[3]，使 pt[3]-pt[6] 等于 3-6 的成本，以此类推（左图）。然后从顶点 1 开始，沿着父链接直至到达一个其势已知道的顶点（在这种情况下，即为 6），并沿着路径下行，从而计算出顶点 9 和 1 的势（中图）。当从顶点 2 开始时，可以通过其父结点计算出它的势（右图）；当从顶点 3 开始时，我们看到它的势已经已知，以此类推。在这里例子中，当尝试 1 之后的每个顶点时，要么发现它的势已经确定，要么可以从其父结点计算出其值。无论采用何种树结构，我们从不会对某条边回溯，因此总运行时间是线性的。

程序 22.10 沿着环增大

函数 lca 通过从两个顶点同时沿树上移，找到其参数顶点的最小公共祖先，并对遇到的每个顶点做出标志，直到到达一个不是根结点的已标记的顶点。（如果根结点是 LCA，那么循环终止时，u 等于 v。）

从这两个结点到其 LCA 的路径上的所有结点都在由添加连接这两个结点的边所形成的环上。为了沿着这条环增大，我们遍历这两条路径一次，以找到通过其边能够推入的最大流量。然后我们再次遍历这两条路径，在其中的一条路径上，按照反方向推入这个流。

这些函数的主要性能特征是它们的运行时间与环中的结点数成正比。

```
int lca(link st[], int u, int v)
  { int i, j;
    mark[u] = ++valid; mark[v] = valid;
    while (u != v)
    {
      u = ST(u); v = ST(v);
      if (u != ST(u) && mark[u] == valid) return u;
      mark[u] = valid;
      if (v != ST(v) && mark[v] == valid) return v;
      mark[v] = valid;
    }
    return u;
  }
link augment(link st[], link x)
{ link u, cyc[maxV]; int d, N;
  int t, i = x->v, j = x->dup->v;
  t = lca(st, i, j);
  cyc[0] = x; N = 1;
  while (i != t)
    { cyc[N++] = st[i]->dup; i = ST(i); }
  while (j != t)
    { cyc[N++] = st[j]; j = ST(j); }
  for (i = 0, d = C; i < N; i++)
    { u = cyc[i]; d = Q > d ? d : Q; }
  for (i = 0; i < N; i++) addflow(cyc[i], d);
  for (i = 0; i < N-1; i++)
    { u = cyc[N-1-i]; if (Q == 0) return u; }
  }
```

程序 22.10 中的实现使用了一种简单技术来避免每次调用时初始化所有标志所付出的开销。我们将标志维护为全部变量，初始化为 0。每次在寻找一个 LCA 时，我们就使全局计数器增加 1，并通过设置其顶点索引的数组中的对应元素为那个计数器的值来对顶点进行标志。初始化后，这项技术可使我们在与环的长度成正比的时间执行计算。在一般情况下，可能沿着大量较小的环进行增大，因此节省的时间是很可观的。我们将了解到，同样的技术在其他部分的实现中所节省的时间也是很有用的。

我们的第三个树处理任务是用 u-v 与树边所创建的环中的另一条边来代替边 u-v。程序 22.11 是完成这个任务的基于父链接表示的一种实现。再次强调 u 和 v 的 LCA 是重要的，因为要删除的边要么在从 u 到 LCA 的路径上，要么在从 v 到 LCA 的路径上。删除一条边使得它的所有后代与树分离，但我们可以通过将 u-v 之间的链接反向并删除边来修补这个损害，如图 22-49 所示。

这三个实现支持了网络单纯形算法的基本操作：可以通过检查降低的成本和流来选择一条合格边；可以使用父链接表示的生成树，沿着所选合格边与树边所形成的负环来增大；也可以更新树并重新计算势。这些操作在图 22-50 和 22-51 中的网络流的例子中作了描述。

图 22-50 描述了使用带有最大流的一条虚拟边来初始化数据结构，如图 22-43 所示。这里所显示的是带有父链接表示、相应顶点势、非树边降低的成本以及初始合格边集的初始可行生成树。而且，不是在实现中计算出最大流值，我们使用源点的流出量，它保证了不小于最大流值；我们使用最大流值来使得算法的运行易于跟踪。

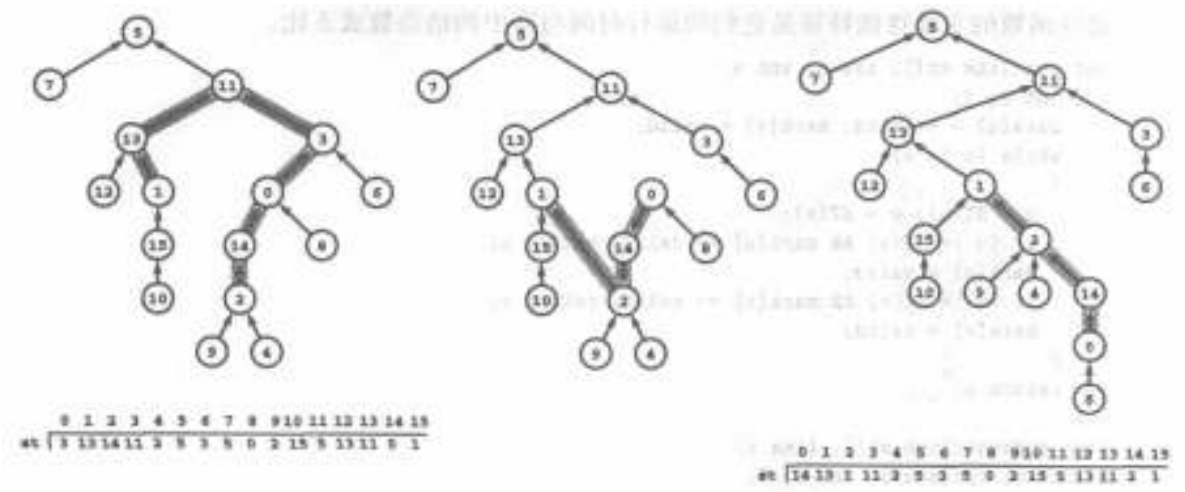


图 22-49　生成树替换

这个例子描述了基于父链接表示的网络单纯形算法中的基本树处理操作。左图是一棵示例树，所有链接指向上方，如在数组表示的父链接结构中所示。通过指向网络结构的链接隐含地维持父链接的值，因此树链接可以在两个方向表示网络的边（见正文）。添加边 1-2 造成了从 1 和 2 到其 LCA（即 11）的路径的环。如果我们接着删除其中的一条边，比如说0-3，结构仍然是一棵树。为了更新父链接数组来反应这种变化，我们将从 2 直到 3 的所有链接改变方向（中图）。右图的树是与改变的结点位置相同的一棵树，因此所有链接向上。如表示这棵树的父链接数组所示（右下图）。

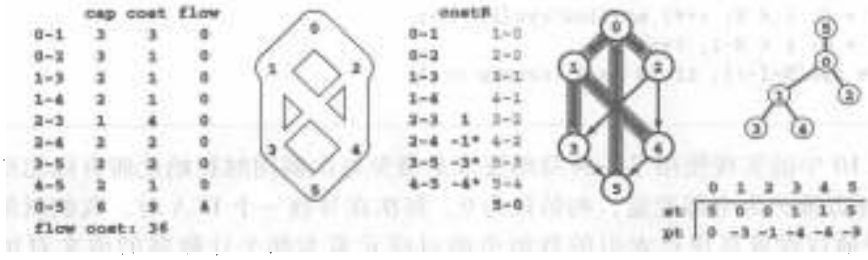


图 22-50　网路单纯形的初始化

为了初始化网络单纯形算法的数据结构，从所有边上的 0 流值开始（左图），然后添加一条从源点到汇点的虚拟边 0-5，其上流值不超过最大流值（为简洁，这里使用等于最大流值的值）。虚拟边的成本值 9 大于网络中任意环的成本；在实现中，我们使用值 *CV*。在流网络中虚拟边未显示出，但是它包括在残量网中（中图）。

我们将汇点作为根结点、源点作为它的唯一子结点，以及由残量网中的其余结点所导出的图的搜索树来初始化生成树。这一实现在数组中使用树的父链接表示法；我们给出了这种表示法以及其他两种表示法：右图中表示有根表示法，以及残量网中阴影边集。

顶点的势放在数组 pt 中，由树结构可计算出，从而使一棵树边的顶点势差等于与其成本。

中图中的列标识的 costR 给出了非树边的降低成本，通过将顶点的势差添加到其成本中来计算每条边的降低成本。树边的降低成本为 0，并且为空。带有负的降低成本的空边和带有正的降低成本的满边（合格边）用星号标出。

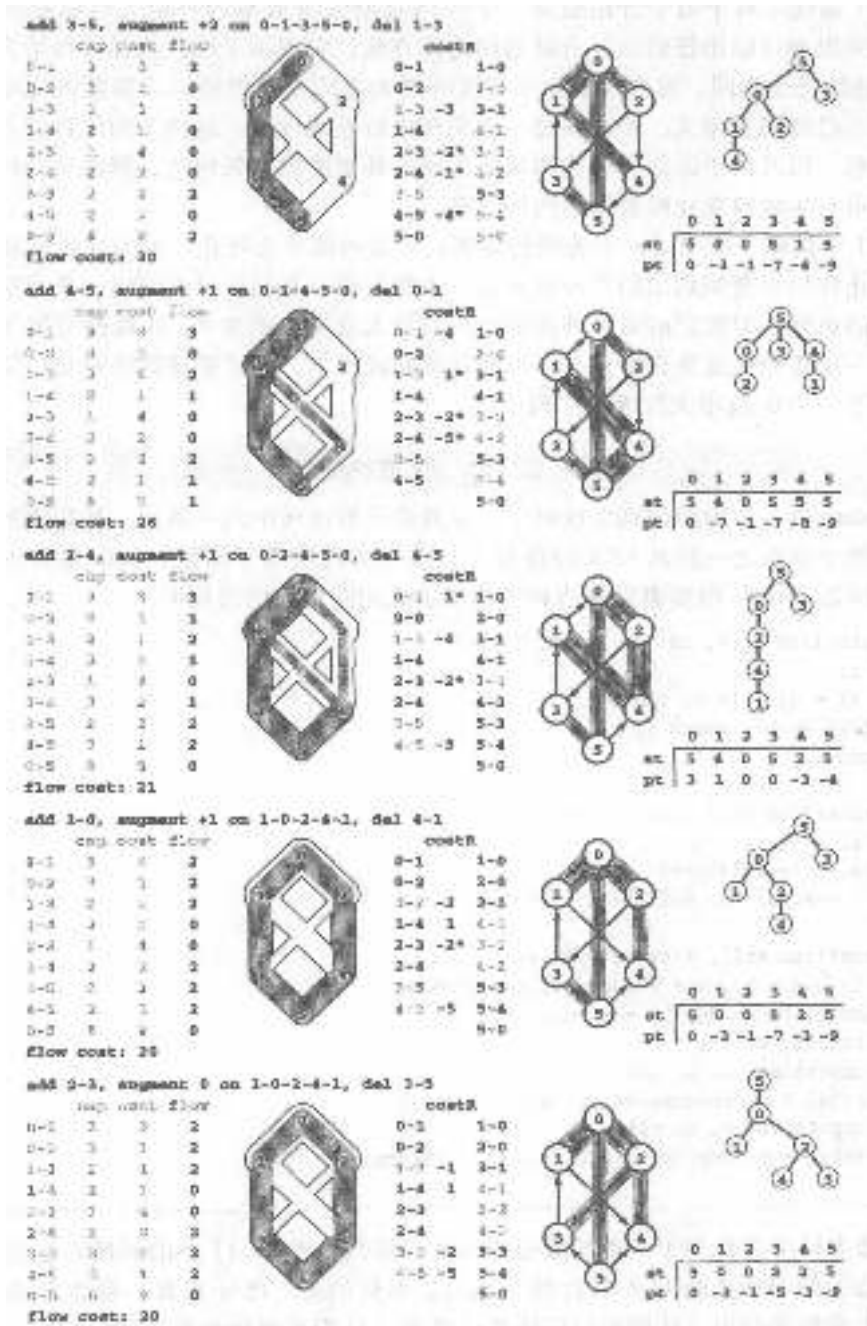


图 22-51 残量网络和生成树（网络单纯形算法）

在图 22-50 描述的初始化之后，此图中的每一行对应网络单纯形算法的一次迭代：在每次迭代中，它选择一条合格边，沿着一个环增大，并如下更新数据结构：首先，此流被增大，包括了残量网中隐含的变化。第二，通过添加一条合格边和从该边与树边形成的环中删除一条边改变了树结构 st。第三，势表 pt 被更新，以反映树结构的变化。第四，非树边的降低成本（中图中标以 costR 的列）被更新，以反映势的变化。这些值常用于将带有负降低成本的空边和带有正降低成本的满边识别为合格边（在降低成本上用星号标出）。实现时不需要进行所有计算（只需要计算势的变化和减低的成本就足以识别一条合格边），但我们这里将所有数都包含进来，以提供算法的一个完整描述。

该例的最终增大是退化的。它并没有增大流，但它不再有合格边。这保证了该流是一个最小成本最大流。

图 22-51 描述了对于每个合格边的一个序列且沿着负成本的环增大时数据结构中的变化。该序列并没有反应出任何选择合格边的特殊方法；它表示了这些选择使得增大路径与图 22-43 中描述的完全相同。这些图显示了每次环增大之后顶点的势以及降低的成本。即使这些数中的很多已经隐含定义，不必通过一般实现进行显式计算。这两个图的目的是来描述算法的完整过程，以及由于添加合格边和从所形成的环中删除一条树边，算法从一棵可行生成树向另一棵可行生成树变化时数据结构的状态。

图 22-51 中的例子说明了一个关键的事实：算法可能不会终止，因为在生成树中的满边或空边会阻止我们沿着识别出的负环推入流。也就是说，我们可以识别出一条合格边和它与生成树形成的负环，但我们沿着该环能够推入的最大流量可能为 0。在这种情况下，我们仍然用环中的一条边替换这条合格边，不再能够降低流成本。为了保证算法终止，需要证明算法不会结束于一个 0-流增大的无限序列中。

程序 22.11 生成树替换

函数 update 将一条边添加到生成树中，并删除所形成环中的一条边。被删除的边是在所添加边上的两个顶点之一到其 LCA 的路径上。这一实现使用了函数 onpath 来找出要删除的边，并使用函数 reverse 将要删除的边和所添加边之间路径上的边反向。

```
int onpath(link st[], int a, int b, int c)
  { int i;
    for (i = a; i != c; i = ST(i))
      if (i == b) return 1;
    return 0;
  }
int reverse(link st[], int u, int x)
  { int i;
    while (i != st[x]->v)
      { i = st[u]->v; st[i] = st[u]->dup; u = i; }
  }
int update(link st[], link w, link y)
  { int t, u = y->v, v = y->dup->v, x = w->v;
    if (st[x] != w->dup) x = w->dup->v;
    t = lca(st, u, v);
    if (onpath(st, u, x, t))
      { st[u] = y; reverse(st, u, x); return; }
    if (onpath(st, v, x, t))
      { st[v] = y->dup; reverse(st, v, x); return; }
  }
```

如果在增大环中存在多于一条的满边或空边，那么程序 22.11 中的替换算法总是从树中删除与合格边的两个顶点的 LCA 最近的一条边。幸运的是，已证实有一种特殊的策略可用于选择从环中要删除的边，从而足以保证算法终止（见第五部分参考文献）。

在开发网络单纯形算法的实现中，我们面临的最后一个重要选择是识别合格边并选择向树中添加边的策略。应该维护包含合格边的一个数据结构吗？如果是这样，这个数据结构要多复杂才合适呢？这些问题的答案从一定程度上依赖于应用和求解特定问题实例的动态性。如果合格边的总数较小，那么维护单独的数据结构是值得的；如果大多数时间内多数边都是合格的，答案则是否定的。维护单独的数据结构可能在搜索合格边的开销上有所降低，但同时可能要求代价昂贵的更新计算。在这些合格边中选择一条边的准则是什么呢？同样有很多

种做法可以采用。我们将在实现中考虑一些例子，然后再讨论另一些候选方法。

程序22.12 网络单纯形（基本实现）

该ADT函数使用网络单纯形算法来求解最小成本流问题。它是一种直接实现。重新计算所有顶点的势，并在每次迭代中遍历整个边表来找出最大降低成本的一条合格边。宏R给出了一条链接边的降低成本。

```
#define R(u)  u->cost - phi[u->dup->v] + phi[u->v]
void addflow(link u, int d)
  { u->flow += d; u->dup->flow -=d; }
int GRAPHmincost(Graph G, int s, int t)
  { int v; link u, x, st[maxV];
    GRAPHinsertE(G, EDGE(s, t, M, M, C));
    initialize(G, s, t, st);
    for (valid = 1; valid++; )
    {
      for (v = 0; v < G->V; v++)
        phi[v] = phiR(st, v);
      for (v = 0, x = G->adj[v]; v < G->V; v++)
        for (u = G->adj[v]; u != NULL; u = u->next)
          if (Q > 0)
            if (R(u) < R(x)) x = u;
      if (R(x) == 0) break;
      update(st, augment(st, x), x);
    }
    return GRAPHcost(G);
  }
```

程序22.12是网络单纯形算法的一个完整实现，它使用了选择合格边的策略，给出负环的成本是绝对值最高的成本。实现依赖于程序22.9～22.11中的树处理函数，但对于第一个消环实现（程序22.8）所作的注解也适用：这样一段简单代码的功能非常强大，可以为求解最小成本流问题提供一般求解问题的模型，非常令人注目。

程序22.12的最坏情况下的性能上界要比程序22.28中的消环实现至少小V倍，因为每次迭代的时间仅为E（找出合格边），而不是VE（找出负环）。虽然我们怀疑，使用最大增大而不是使用Bellman-Ford算法所提供的第一个负环将导致更小的增大，但这个结果还未被证明是有效的。所使用的增大环数的某个界限很难得到，这些界限要比我们实际中看到的高很多。如早先提到的那样，一些理论上的结果表明，某些策略可以保证增大环的数目的界限为边数目的一个多项式。但是，实际实现最坏情况下为指数。

根据这些考虑，可以考虑很多因素来改进性能。例如，程序22.13是网络单纯形算法的另一种实现。程序22.12中的直接实现，重新使树有效所花费的时间与V成正比，找出具有最大归约成本的合格边所花费的时间与E成正比。程序22.13中的实现设计用来消除一般网络中的这些开销。

首先，即使选择最大边会导致最少的迭代数，但检查每条边来找出最大边的花费，这可能是不值得的。我们可以沿着一个短环进行无数次的增大，所需时间为扫描所有边的时间。因此，考虑使用任一合格边的策略是值得的，不是花费时间来找出一条特殊边。在最坏情况下，可能必须检查所有的边或其中的相当部分来找出一条合格边，但是我们期望检查相对少量的边来找出一条合格边。一种方法是每次从头开始；另一种方法是从一个随机点开始

(见练习 22.127)。使用随机化不太可能人为地使增大路径过长。

程序 22.13 网络单纯形(改进实现)

这个网络单纯形的实现在每次的迭代中将时间保存起来,只在需要时计算势,并取它找到的第一条合格边。

```
int R(link st[], link u)
  { return u->cost
      - phiR(st, u->dup->v) + phiR(st, u->v); }
int GRAPHmincost(Graph G, int s, int t)
  { int v, old = 0; link u, x, st[maxV];
    GRAPHinsertE(G, EDGE(s, t, M, M, C));
    initialize(G, s, t, st);
    for (valid = 1; valid != old; old = valid)
      for (v = 0; v < G->V; v++)
        for (u = G->adj[v]; u != NULL; u = u->next)
          if ((Q > 0) && (R(st, u) < 0))
           { update(st, augment(st, u), u); valid++; }
    return GRAPHcost(G);
  }
```

第二,我们采用一种懒方法来计算势。不是计算顶点索引数组 phi 的所有势,在需要时参考它们,我们通过调用函数 phiR 来得到每个势值;它遍历这棵树来找出有效的势,然后计算出那条路径上所需的势。为了实现这种方法,我们简单地将定义成本的宏改变为使用函数调用 phiR(u),不使用数组访问 phi[u]。在最坏情况下,我们像以前那样计算所有势;但如果只检查几条合格边,那么我们只计算那些需要识别它们的势。

这样的改变并不影响算法的最坏情况下的性能,但在实际应用中它们肯定会更快。在练习(见练习 22.127 ~ 22.131)中,探索了几种单纯形算法的改进思想;这些只代表了已经提出算法的一小部分。

正如在本书中所强调的,图算法的分析和比较是复杂的。有了网络单纯形算法,任务就更复杂了,因为有各种不同的实现方法以及我们可能遇到各种类型的应用(见 22.5 节)。哪一种实现是最好的?我们可以根据所证明的最坏情况下的性能上界来比较这些实现吗?对于各种不同实现的量化性能的精度又如何?如果使用可多种实现,每一种实现都有特定应用吗?

鼓励读者通过各种网络单纯形法的实现来获得计算经历。并通过本书中强调的实验解决这些问题。当寻求解决最小成本流问题时,我们面对类似地基础性的挑战,但是在处理不断增加的困难问题所获得的经验为我们开发有效实现提供了丰富的背景,而这些有效实现可以解决一大类实际问题。本节和下一节的练习中描述了这样的一些研究,但这些练习只能看作一个起点。每个读者可以完成一个新的实验研究,从而明确实现与应用之间的关系。

通过适当地部署基本任务的经典数据结构和算法(或开发新的),有可能极大地改进关键应用的性能,这使得网络单纯形实现的研究成为一个成果丰富的领域,而且存在大量网络单纯形算法的实现。在以往,此研究的进展是至关重要的,因为它有助于减少解决网络单纯形问题的巨大开销。人们试图依赖精心设计的库来解决这些问题,且在很多情况下,这样做是合适的。然而,这些库很难跟得上目前研究的脚步,而且也无法适应于新应用中出现的大量问题。基于现代计算机的速度和规模,为了对大量应用开法有效的求解问题的工具,可以将诸如程序 22.12 和程序 22.13 等可访问的实现作为起点。

练习

▷ **22.118** 对于图22-11所示的流网络，给出它的一个有相关可行生成树的最大流。

22.119 实现邻接表表示的流ADT（程序22.1）的函数，它可以删除流中部分边上的环，并构建结果流的一棵可行生成树，如图22-45所示。你的函数以网络和数组作为参数，并在数组中返回树的父-链接表示（按照标准约定，使用链接到表结点而不是下标的表示）。

22.120 在图22-47中的例子中，将连接6和5的边的反向后，说明它对势表的影响。

○ **22.121** 构造一个流网络，并展示一系列扩展边，从而使通用单纯形网络算法对此无法终止。

▷ **22.122** 按照图22-48的风格，显示计算图22-47中以0为根的树的势的过程。

22.123 按照图22-51的风格，显示计算图22-11中所示的流网络的一个最小成本最大流的过程，从你在练习22.118中找到的基本最大流和相关基本生成树开始。

○ **22.124** 假设所有非树边为空。编写一个以两个顶点索引的数组（一个基本生成树的父-链接表示和一个输出数组）作为参数的函数，并计算树边上的流。并将连接v与其父结点的边上的流和它在树中的父结点放在输出数组的第v个元素中。

○ **22.125** 对于某些非树边可能为满的情况，重做练习22.124。

22.126 将程序22.11作为MST算法的基础。进行实验对第20章中描述的三个基本MST算法进行比较。

22.127 描述如何修改程序22.13，使其以随机的方式扫描一条合格边，而不是每次从开始扫描。

22.128 修改练习22.127的解，使当每次搜索时，从上次搜索停止的地方开始搜索一条合格边。

22.129 修改本节的辅助函数，维护一个三叉链表树结构，包含每个结点的父结点，最左端孩子结点和右兄弟结点（见5.4节）。沿着环增大且用一条合格边代替一条树边的函数所花费的时间应该与增大环长度成正比，计算势的函数所花费的时间应该与树边被删除时所建立的两棵子树中的较小者的大小成正比。

○ **22.130** 修改本节的辅助函数使其除了父-连接数组之外，还可以维持其他两个父-链接数组：一个包含每个顶点到根结点的距离，另一个包含每个顶点在DFS中的后继。沿着环增大且用一条合格边代替一条树边的函数所花费的时间应该与增大环的长度成正比，计算势的函数所花费的时间应该与树边被删除时所建立的两棵子树中的较小者的大小成正比。

● **22.131** 探索维护合格边的广义队列的想法。考虑各种广义队列的实现和各种避免过度边成本计算改进方法，诸如限制计算为合格边的一个子集，从而限制队列大小；或者可能允许某些不合格的边保留在队列中。

● **22.132** 对于各种版本的网络单纯形算法（见练习22.7～22.12），进行实验研究来确定迭代数，所计算的顶点的势的数目，以及运行时间与E的比值。考虑本文和上一练习中描述的各种算法，重点放在对于大规模稀疏网络有最好性能的算法上。

● **22.133** 编写一个客户端程序，可以动态地可视化网络单纯形算法的运行过程。你的程序应该产生类似图22-51中的图像，以及本节中的其他图像（见练习22.50）。对于练习22.7～22.12，使用欧几里得网络测试你的实现。

22.7 最小成本流归约

最小成本流是一个求解问题的一般模型，它可以涵盖大量有用的实际问题。本节，我们

将通过对大量问题到最小成本流的特定归约来证实这一点。

最小成本流问题显然比最大流问题更具有一般性，因为任何最小成本最大流都是最大流问题的一个可接受的解。具体地说，在图 22-43 构造的过程中，如果我们将虚拟边的成本指定为 1，并将所有其他边的成本指定为 0，那么任何最小成本最大流都会使虚拟边中的流最小化，相应地使原网络中的流最大化。因此，在 22.4 节中讨论的可归约为最大流问题的所有问题也可归约为最小成本流问题。这类问题包括二分匹配、可行流和最小割，还有很多其他问题。

更有趣的是，我们可以分析最小成本流问题的算法性质，来开发最大流问题新的通用算法。我们已经注意到，最小成本最大流问题的通用消环算法给出了最大流问题的一种通用增大路径算法。特别地，这种方法可以得到一种实现，将无需搜索网络就能找出增大路径（见练习 22.134 和 22.135）。另一方面，该算法可能产生 0 流的增大路径，因而很难评价其性能特征。

根据以下简单归约，最小成本流问题比最短路径问题也更具有一般性。

性质 22.29（无负环的）　*单源点最短路径问题可归约为最小成本可行流问题。*

证明　给定一个单源点最短路径问题（一个网和一个源点 s），构建一个具有相同顶点、边和边成本的流网络，其中每条边上有无限容量。增加一个新的源点，其到 s 的边上的成本为 0，容量为 $V-1$，另增加一个汇点，从各个其他顶点到该汇点的边上的成本为 0，容量为 1。图 22-52 显示了这个构造过程。

求解该网的最小成本最大流问题。如果必要，在构造最小生成树的过程中去掉环。这棵生成树直接对应着原网中的最短路径生成树。这一结论的详细证明留作练习（见练习 22.139）。■

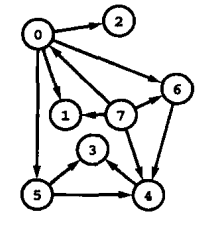
因此，在 21.6 节讨论的可归约为单源点路径问题的所有问题也可归约为最小成本流问题。这类问题包括具有截止期的作业调度问题、差分约束问题，还有其他大量问题。

正如我们在研究最大流问题时所发现的，在使用性质 22.29 的归约求解一个最短路径问题时，值得考虑网络单纯性算法操作的一些细节。在这种情况下，算法维护一个以源点为根的生成树，更像是我们在第 21 章中考虑过的基于搜索的算法，但在开发选择加入树的下一条边的方法时，结点的势和可归约的成本提供了越来越多的灵活性。

我们并没有认识到最小成本流问题是最大流问题和最短路径问题的一种适当的推广，因为对于这两个问题均有特定的算法，而它们都具有更好的性能保证。然而，如果没有这种实现，网络单纯形算法的良好实现则有可能对这两个问题的特殊实例提供快速的解法。当然，我们在使用或构建利用了这些归约的网络处理系统时，必须避免归约进入循环。例如，程序 22.8 中的消环算法实现使用了最大流和最短路径来求解最小成本流问题（见练习 21.96）。

下面将讨论等价的网模型。首先，我们将说明假设成

```
0-1  41
1-2  51
2-3  50
4-3  36
3-5  38
3-0  45
0-5  29
5-4  21
1-4  32
4-2  32
5-1  29
```




```
     cost  cap
0-1   41
1-2   51
2-3   50
4-3   36
3-5   38
3-0   45
0-5   29
5-4   21
1-4   32
4-2   32
5-1   29
8-0    0    7
1-9    0    1
2-9    0    1
3-9    0    1
4-9    0    1
5-9    0    1
6-9    0    1
7-9    0    1
```

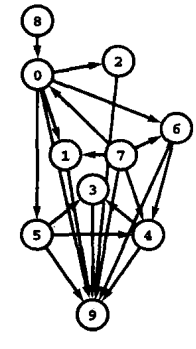


图 22-52　由最短路径进行归约

在上图的网中找一条单源点的最短路径树等价于解下图流网络中的最小成本最大流问题。

本非负并不是一个限制性条件，因为可以将有负成本的网转换成无负成本的网。

性质22.30 *在最小成本流问题中，可以不失一般性假设边成本是非负的。*

证明 我们将对配送网中的可行最小成本流来证明这一点。对于最小成本最大流问题，结论同样成立。因为由性质22.22的证明可得这两个问题是等价的（见练习22.144和22.145）。

给定一个配送网络，将成本 $x < 0$ 且容量为 c 的任何边 u-v 替换为一条有同样容量的边 v-u，但其成本为 $-x$（一个正数）。此外，我们可以使 u 的供需值降低 c，使 v 的供需值增加 c。这个操作对应着向 u 至 v 推进 c 个单位的流，并相应地调整网络。

对于负成本的边，如果对于转换后的网的最小成本流问题的解决方案在边 v-u 上放置流 f，那么我们在原网中的边 u-v 上放置流 c-f；对于正成本边，转换后的网与原网有相同的流。这种流分配可以保持所有顶点上的供应或需求约束条件。

转换后的网中，v-u 中的流对成本的贡献为 fx，而原网中 v-u 中的流对成本的贡献则为 $-cx+fx$。因为此表达式中的第一项与流无关，转换后的网中的任何流的成本等于原网中对应流的成本加上所有负成本边上的容量与成本之积的和（这是一个负值）。转换后的网中的最小成本流是原网中的一个最小成本流。 ■

这一归约表明我们可以只考虑正成本，但在实际中我们并不这样做，因为在22.5和22.6节中的实现只在残量网上工作，且毫不困难就可以处理负成本。在某些情况下需要成本有下界这很重要，但此下界又不必为0（见练习22.146）。

接下来，我们将表明就像在最大流问题中所作的那样，如果需要，可以只考虑无环网。此外，我们还可以假设边上的容量是无限的（边上的流量没有上界）。将这两点组合在一起就得到以下最小成本流问题的经典形式化描述。

运输问题 求解二分配送网的最小成本流问题，其中所有边都由一个供应顶点指向一个需求顶点，且有无限容量。如本章开始所讨论的那样（见图22-2），通常求解此问题的思路是，建立沿着配送渠道（边）从仓库（供应顶点）到零售店（需求顶点）的商品配送问题的模型，其中每单位货物量有一个特定的成本。

性质22.31 *运输问题等价于最小最小成本流问题。*

证明 给定一个运输问题，我们可以通过为每条边上的容量指定大于包含它的顶点上的供给量或需求量的值，并且求解所得配送网中的最小成本可行流问题，来解决这个运输问题。因此，我们只需建立从标准问题到运输问题的一个归约。

为简单起见，我们描述了一个新的转换，其仅对稀疏网是线性的。构造过程类似于我们在性质22.16中使用的方法，建立了非稀疏网的结果（见练习22.149）。

给定一个有 V 个顶点、E 条边的标准配送网，如下构建一个有 V 个供应顶点、E 个需求顶点且有 $2E$ 条边的运输网。对于原网中的每个顶点，在二分网中加入一个顶点，其上的供应值或需求值设为原值加上发出边上的容量之和；对于原网中容量为 c 的每条边 u-v，在二分网中加入一个顶点，其上的供应值或需求值为 $-c$（我们使用 $[u$-$v]$ 表示此顶点）。对于原网中的每条边 u-v，在二分网中加入两条边：一条边从 u 到 $[u$-$v]$，并有相同成本，另一条边从 v 到 $[u$-$v]$，成本为0。

以下的一一对应保持了两个网中流之间的成本：原网中一条边 u-v 有流值 f 当且仅当二分网中边 u-$[u$-$v]$ 有流值 f，且边 v-$[u$-$v]$ 有流值 c-f（这两个流之和必为 c，这是由于顶点 $[u$-$v]$ 上的供-需约束）。因此一个网中的任何最小成本流都对应于另一个网中的一个最小成本流。 ■

由于我们并未讨论解决运输问题的直接算法，此归约仅有理论价值。为了使用此归约，我们就要使用在性质 22.31 中的一开始提到的简单归约，将所得的问题转换回一个（不同的）最小成本流问题。也许这样的网络在实际中会产生更有效的解；也许不会。研究运输问题和最小成本流问题之间的等价关系的关键是理解：去除容量且仅关注二分网络看上去会大大简化最小成本流问题；然而实际上并非如此。

我们需要考虑另一个经典问题。它推广了在 22.4 节中详细讨论的二分匹配问题。像那个问题一样，给人以简单的假象。

分配问题 给定一个加权二分图，找出一组有最小总权值的边，使得每个顶点只与另外一个其他顶点相连接。

例如，我们可以推广职位安置问题，针对每个应用，使每个公司量化它的需求（比如说，为每个求职者指定一个整数，求职者越好整数越小）。那么，对于分配问题的一个解决方案就可以提供一种合理的方法，将这些相对优先关系考虑在内。

性质 22.32 *分配问题可归约到最小成本流问题。*

使用简单归约到运输问题可以得到这个结果。给定一个分配问题，可以构造一个具有相同顶点和边的运输问题，其中任一集合中的所有顶点指定为值为 1 的供应顶点，另一个集合中的所有顶点指定为值为 1 的需求顶点。每条边上的容量指定为 1，并为分配问题中对应的那条边上的权值指定一个成本。

将这个运输问题的实例归约到最小成本流的过程给出了一种构造过程。它本质上等价于我们在将二分匹配问题归约到最大流问题时所使用的构造过程（见练习 22.159）。 ■

这种关系还不能称为是等价的。尚没有将最小成本流问题归约为分配问题的方法。实际上，就像单源点的最短路径问题和最大流问题，分配问题看起来要比最小成本流问题更简单一些，因为从求解它的算法来看，它比现有求解最小成本流的最好算法有更好的渐近性能。不过，网络单纯形算法得到了充分的改进。它的一种良好的实现是用于求解分配问题的一个合理的选择。此外，与最大流问题和最短路径问题一样，我们可以调整网络单纯形算法来得到分配问题的改进性能（见第五部分参考文献）。

接下来归约到最小成本流问题会产生图中路径有关的一个基本问题，就像我们在 17.7 节一开始考虑的问题。如在欧拉路径问题中的做法，希望有那么一条路径能够包括图中的所有边。我们知道并不是所有的图都存在这样的路径，因此放宽限制，图中的边只出现一次。

邮差问题 给定一个网（加权有向图），找出一条有最小权值的回路，其中将每条边至少包含一次（见图 22-53）。回忆我们在第 17 章中关于回路的基本定义，其中对于回路（可能重新访问顶点和边）和环（除了第一个顶点和最后一个顶点相同外，其余顶点都不同）做了区别。

此问题的解将描述出一个邮差所行的最佳路线（邮差必须遍及其路线上的所有街道）。此问题的一个解也可以描述一个扫雪机在雪天的路线，另外还有很多类似的应用。

邮差问题是我们在 17.7 节讨论的欧拉路径问题的一个扩展：练习 17.92 的解就是对有向图是否存在欧拉路径的一个简单测试。程序 17.14 是找这类有向图中的欧拉路径的一种有效的方法。这条路径解决了邮差问题，因为它包含每条边仅一次，没有其他路径会有更小的权值。如果入度和出度

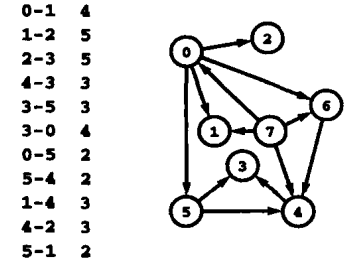


图 22-53 邮差问题

找出包括每条边至少一次的最短路径是一个挑战性的问题，即使是在对于这个小的网络也是如此。但是可以将问题归约到最小成本流问题来有效地求解此问题。

不相等，此问题将变得更为困难。在一般情况下，某些边必定会被遍历多于一次：此问题就是使所有多次遍历边的总权值达到最小。

性质22.33 *邮差问题可归约为最小成本流问题。*

证明 给定邮差问题的一个实例（一个加权有向图），在同样的顶点和边上定义配送网络，其中所有顶点的供应或需求值设置为0，边的成本设置为相应边上的权值。边上的容量无限。但是所有边的容量都要求大于1。我们将一条边 u-v 上的流值 f 解释为邮差需要遍历 u-v 共 f 次。

通过使用练习22.147的转换来去除边容量的下界，从而为此网络找出一个最小成本流。由流分解定理可得，我们可以将流表示为一个环的集合。因而，我们可以由此流构建一条回路，所用方法与在欧拉图中构建欧拉路径一样：我们遍历任何环，一旦遇到一个在另一环上的顶点，则迂回遍历到另一个环。 ■

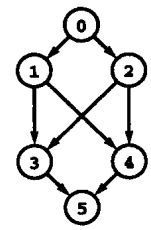
仔细研究邮差问题再次表明：图算法中的平凡问题和困难问题之间微妙的界限。假设我们讨论此问题的双向版本，网是无限的，邮差可以沿着每条边的两个方向行走。这样，如我们在18.5节所指出的，深度优先搜索（或任何图搜索）将会提供直接的解法。然而，如果可以遍历每条无向边的任何一个方向即可满足，那么较之于简单归约到最小成本流问题，对上述问题的形式化要困难多。如果某些边是有向的，而其他边是无向的，那么问题就变成了NP-难问题（见第五部分参考文献）。

只有少量的实际问题可以被形式化为最小成本流问题。最小成本流问题实际上甚至比最大流问题或最短路径问题用途更多。网络单纯形算法可以有效地解决这个模型所涵盖的所有问题。

正如我们在研究最大流时的做法，可以考察最小成本流问题是如何转换到一个LP问题的，如图22-54所示。这种形式化是最大流形式化的一种直接扩展：我们增加了虚拟变量等于此流的成本的等式，然后设置目标函数，并使该变量达到最小。LP模型允许我们任意增加（线性）约束条件。有些约束条件使问题仍然可能与最小成本流问题等价，但有些约束条件却不能。也就是说，很多问题并不能归约到最小成本流问题：特别是，LP涵盖范围更广的问题。最小成本流问题是求解问题的一般模型的下一步，我们将在第八部分讨论。

此外还有比LP模型更具一般性的模型；然而，LP还有一个优点，即尽管LP问题通常比最小成本流问题更困难，但已经为此发明了有效而高效的算法。实际上，这种算法中可能最重要的是称为“单纯形”的方法：网络单纯形方法是单纯形方法的一个特定的版本，应用于LP问题的一个子集中，该子集对应着最小成本流问题。理解网络单纯形算法有助于我们理解这个单纯形算法。

	cap	cost
0-1	2	3
0-2	3	1
1-3	3	1
1-4	1	1
2-3	1	4
2-4	1	2
3-5	2	2
4-5	3	1
0-5	*	9



最大值 c

约束条件

$$x_{01} \leqslant 2$$
$$x_{02} \leqslant 3$$
$$x_{13} \leqslant 3$$
$$x_{14} \leqslant 1$$
$$x_{23} \leqslant 1$$
$$x_{24} \leqslant 1$$
$$x_{35} \leqslant 2$$
$$x_{45} \leqslant 3$$
$$x_{50} = x_{01} + x_{02}$$
$$x_{01} = x_{13} + x_{14}$$
$$x_{02} = x_{23} + x_{24}$$
$$x_{13} + x_{23} = x_{35}$$
$$x_{14} + x_{24} = x_{45}$$
$$x_{35} + x_{45} = x_{50}$$

图22-54 最小成本流问题的LP形式化

这个线性规划等价于图22-41中的实例网的最小成本流问题。边等式和顶点不等式与图22-40中的相同，但是目标函数不同。变量 c 表示总成本，它是其他变量的线性组合。在这种情况下，$c = -9x_{50} + 3x_{01} + x_{02} + x_{13} + x_{14} + 4x_{23} + 2x_{24} + 2x_{35} + x_{45}$。

练习

○ **22.134** 说明在网络单纯形算法计算一个最大流时，其生成树是两棵树的并 t-s，其中一棵树为 s，另一棵树为 t。

22.135 开发一个基于练习 22.134 的最大流实现，随机选择一条合格边。

22.136 按照图 22-51 的样子，使用课文中所描述的归约和程序 22.12 的网络单纯形实现，显示图 22-11 中所示的流网络的最大流的计算过程。

22.137 按照图 22-51 的样子，使用课文中所描述的归约和程序 22.12 的网络单纯形实现，显示图 22-11 中所示的流网络的从 0 开始的最短路径的计算过程。

○ **22.138** 证明在性质 22.29 的证明中所描述的生成树的所有边位于从源点到叶结点的有向路径上。

○ **22.139** 证明在性质 22.29 的证明中所描述的生成树对应着原网络中的一条最短路径树。

22.140 假设你使用网络单纯形算法来求解对性质 22.29 中所描述的单源点最短路径问题进行归约所得问题。(ⅰ) 证明该算法绝不会使用 0-成本的最大路径。(ⅱ) 说明离开环的顶点总是添加到此环中该边的目的顶点的父结点。(ⅲ) 作为练习 22.139 的一个结论，网络单纯形算法并不需要维持边上的流。给出利用了这一点的一种完全实现。随机地选择新的树边。

22.141 假设我们对网络中的每条边指定一个正成本。证明找最小成本的单源点的最短路径树问题可归约到最小成本最大流问题。

22.142 假设修改 21.6 节中的具有截止期的作业调度问题，使得作业可以不在截止期内完成。但如果这样，将会带来一个固定的正成本。说明这个修改过的问题可以归约到最小成本最大流问题。

22.143 使用练习 22.108 的方案（假设成本都是非负的），实现一个 ADT 函数，找出带有负成本的配送网中的最小成本最大流。

○ **22.144** 假设图 22-41 中 0-2 和 1-3 的成本为 -1，而不是 1。说明如何通过将网转换具有正成本的网络并找出新网的一个最小成本最大流来找出最小成本最大流。

22.145 实现一个 ADT 函数，使用 GRAPHmincost（假设成本均非负），找出带有负成本的网中的最小成本最大流。

○ **22.146** 22.5 节和 22.6 节中的实现依赖于成本非负吗？如果依赖于，解释实现是如何依赖于成本非负的；否则，解释需做什么修改（如果需要）才能使之对于有负成本的网也能正确工作，或解释为什么不可能做这样的修改。

22.147 扩展练习 22.75 中的可行流使其包含边上容量的下界。给出一个 ADT 的实现，用来计算关于这些下界的最小成本最大流，或者证明它的最小成本最大流不存在。

22.148 使用文本中的归约，给出将练习 22.114 中描述的流网络归约到运输问题的结果。

○ **22.149** 使用类似于性质 22.16 的证明中的构造过程，说明最小成本最大流问题可归约到只有 V 个额外顶点和边的运输问题。

▷ **22.150** 根据归约到性质 22.30 的证明中所给出的最小成本流问题，开发运输问题的一个 ADT 实现。

○ **22.151** 根据归约到性质 22.31 的证明中所描述的运输问题，开发最小成本流的一个 ADT 实现。

○ **22.152** 根据归约到练习 22.149 中所描述的证明中所描述的运输问题，开发最小成本流的一个 ADT 实现。

22.153 编写一个产生运输问题各种随机实例的程序，然后使用这些实例作为基础，进行实验测试求解该问题的各种算法及实现。

22.154 在线找出运输问题的一个更大实例。

22.155 进行实验研究，比较将任意的最小成本流问题归约到运输问题的两种不同方法。这些方法在性质22.31的证明中讨论过。

22.156 编写一个产生分配问题各种随机实例的程序，然后使用这些实例作为基础，进行实验测试求解该问题的各种算法及实现。

22.157 在线找出分配问题的较大实例。

22.158 正文中所描述的职位安置问题偏向雇主（其总权值最大）。形式化此问题的一个版本，使得求职者也能表达其愿望。解释如何解决此版本的问题。

22.159 进行实验研究，比较22.6节中求解有V个顶点、E条边的分配问题的随机实例的两种网络单纯形算法的实现性能。

22.160 邮差问题对于并非强连通的网络（邮差只能访问其在开始所在强连通分量中的那些顶点）显然是无解的，但是这个结果在性质22.33的归约中并未提到。当我们在一个不是强连通的网上使用归约时会怎样？

22.161 对于各种加权图（见练习21.4～21.8）进行实验研究，确定邮差路径的平均长度。

22.162 给出单源点最短路径问题归约到分配问题的一种直接证明。

22.163 描述如何将任一分配问题形式化为LP问题。

○ 22.164 对于每条边上的成本值为-1的情况，做练习22.20（从而可以最小化卡车中的未用空间）。

○ 22.165 设计练习22.20的一个成本模型，满足解是一个取天数最小的最大流。

22.8 展望

我们对于图算法的研究在研究网络流算法是达到了顶峰，其中有4个原因。首先，网络流模型证实了图抽象在不计其数的应用中的适用性。其次，我们所讨论的最大流和最小成本流算法都是对于更简单的问题所研究的图算法的自然扩展。第三，这些实现充分体现了基本算法和数据结构在获得好的性能所具有的重要作用。第四，最大流和最小成本流模型说明了开发求解问题的更为一般的模型以及用这些模型来求解更广泛问题的实用性。我们有能力开发出求解这些问题的有效算法，这样就为开发出更为一般的模型和寻求解决这些问题的有效算法敞开了大门。

在进一步讨论这些问题之前，我们先列出在本章没有涵盖的一些问题，即使它们与我们所熟知的问题紧密相关。然后进一步讨论这些问题。

最大匹配 在带有边权值的图中，找边的一个子集，其中任何顶点都不会出现多于一次，而且任何其他边集都不会比此子集的总权值更大。通过将无权图中的边上的权值设置为1，可将该图的*最大基数匹配*（maximum-cardinality matching）问题直接归约到此问题。

分配问题和最大基数二分匹配问题可归约到一般图的最大匹配问题。一方面，最大匹配不能归约到最小成本流，因而我们讨论的算法不适用。但问题是易解的，虽然对于较大规模的图，求解该问题的计算开销依然很大。如果讨论试图用于一般图的匹配的处理技术会充满这本书：该问题是图论中研究最为广泛的问题之一。我们在本书中只讨论到最小成本流，到第八部分再来讨论最大流问题。

多商品流（multicommodity flow） 假设我们需要计算第二个流，满足一条边行的两个流之和受到该边上的容量限制，这两个流都要处于平衡状态，并使总成本最小。这一改变对于商品配送问题中存在两种不同类型商品的情况建立模型；例如，对于快餐店，能否在其相应卡车中放入更多的汉堡或是更多的土豆？这种修改还将使问题更为困难，并且较之于在此所讨论的方法，还需要更为高级的算法；例如，对于一般情况，没有类似于最大流－最小割定理对于一般情况的结论。将此问题形式化为 LP 问题是对图 22-54 所示的例子的直接扩展。因而问题是易解的（因为 LP 是易解的）。

凸包和非线性成本（convex and nonlinear cost） 我们所讨论过的简单成本函数是变量的线性组合，求解它们的算法本质上依赖于这些函数的基本数学结构。很多应用需要更为复杂的函数。例如，当我们最小化距离时，可得成本平方之和。这样的问题不能被形式化为 LP 问题，因而它们需要求解问题的模型更为强大。很多这样的问题不是易解的问题。

调度问题 我们已经给出了几种调度问题的例子。它们只是数百种不同调度问题的几个代表。对于这些问题之间的关系以及解决这类问题的算法和实现的开发，相关研究可见大量研究文献中（见第五部分参考文献）。实际上，我们可能会选择使用调度而非网络流算法来定义一般求解问题模型并实现归约来解决某些特定问题的思路（同样可以说是匹配）。很多调度问题可归约到最小成本流模型。

组合计算的领域相当广泛，而且对于此类问题的研究，自然要求研究人员还要做多年的努力。我们将在第 8 部分重新讨论这些问题，涉及难解性问题。

我们只给出了求解最大流和最小成本流的已有研究算法的一部分。正如贯穿在本章中的那些练习所指明的，将各种通用算法不同部分的可用部分组合起来就能得到大量不同的算法。此时基本计算任务的算法和数据结构在这些方法的效率方面起着至关重要的作用；实际上，我们研究的一些重要的通用算法在开发时要求有效地实现网络流算法。许多研究人员仍然在对这一主题进行研究。要开发对于网络流问题的更好的算法肯定要依靠对于基本算法和数据结构的智慧使用。

由于网络流算法的覆盖面如此之广，而且我们对于归约的扩展使用也将触角延伸到这一领域，使得本节适合于讨论归约这一概念的含义。对于大量的组合算法，我们是研究某些问题的有效算法，还是研究求解一般问题的模型，这些问题代表了在算法研究中的分水岭。无论哪一方面都有很重要的拉动力量。

我们往往倾向于开发尽可能一般的模型，因为模型越具有一般性，其涵盖的问题就越多，相应地，对于能够归约为此模型的任何问题，解决所有这些问题的有效算法的有用性将有所增加。开发这样一个算法尽管不是完全没有可能，但也有相当大的难度。即使并没有一个能够保证相当有效的算法，但对于我们感兴趣的特定类型的问题，通常仍有许多好的算法能够完成得很好。具体的分析结果往往很晦涩难懂。但我们常常有很有说服力的实验结果。实际上，具体开发人员常常可用的最一般模型（或者有一个完善的解决方案包），如果模型可在合理时间内完成则不再做更多的努力。然而，我们当然要避免使用过于一般的模型，这些模型会导致花费过多的时间来解决一些采用更为专用的模型就很有效的问题。

我们已经在为一些重要的特定问题寻求更好的算法，特别是大型问题或较小问题的大量实例，其中计算资源是一个关键瓶颈。我们通常可以找到一种精巧的算法，它可以使资源开销减少一个数百、数千甚至更多的因子，在以时间或金钱来衡量开销时，这一点尤为重要。第 2 章中介绍的一般观点，已经成功地用于很多领域，在这种情况下，也相当有价值。而且我们还希望在图算法和组合算法方面开发出更为精巧的算法。过度依赖特定算法最重要的缺

点是，如果对模型稍作改变，算法就会失效。在使用一个相当一般的模型和可使问题得到解决的算法时，这一问题所带来的影响则很小。

在许多编程环境中都可以看到一些软件库，它们涵盖了我们所谈到的许多算法。这些库对于讨论特定问来说当然是很重要的资源。然而，这些库可能很难使用、可能会过时，也可能与当前问题不能很好地匹配。有经验的程序员都懂得对充分利用库资源和过于依赖次资源（不会过早过时）加以权衡的重要性。我们所讨论的某些实现很高效，易于开发，且应用范围很广。在许多情况下，将这些实现加以调整和优化从而解决当前问题都是一个适当的方法。

随着我们要解决的问题的难度逐渐增大，理论研究（仅限于所能证明的范围）和实验研究（仅与当前问题相关的领域）之间的对立也日益突出。理论为我们提供了关于问题的一个立足点的指导，而实际经验为我们提供了开发实现的指导。此外，具有实际问题的经验则为理论提供了新的方向，如此循环，扩展了我们所能求解问题的范畴。

最后，无论研究哪种方法，我们的目标都是希望得到求解大量问题的模型；对于这些模型范围内的问题，还希望得到可将其加以解决的有效算法；另外对于可处理实际问题的那些算法，还希望得到其有效实现。对于开发更加一般的求解问题的模型（如最短路径问题、最大流、最小成本流问题），逐步增加的强大的通用算法（如最短路径问题的 Bellman-Ford 算法、最大流问题的增大路径算法，以及最小成本 - 最大流问题的网络单纯形算法）都使我们朝着这个目标前进了一大步。其中很多工作是 20 世纪 50 年代到 60 年代的成果。后来又出现了基本数据结构（第一部分至第四部分）和为这些通用方法提供有效实现的算法（本书），对于我们目前能够处理如此众多的大型问题都起到了根本的作用。

第五部分参考文献

下列的算法书籍涵盖了第 17 章到第 21 章中的大部分基本图处理算法。这些书籍可作为基本图算法和高级图算法的参考文献，所给出的文献涉及广泛，已引用到最近的文献。Even 的教材和 Tarjan 的专著涵盖了我们讨论的很多同样的专题。Tarjan 的使用深度优先搜索求解强连通问题和其他问题的原著论文值得进一步研究。第 19 章中的源 - 队列拓扑排序实现源自 Knuth 的著作。某些其他特定算法的引用出处也在下面列出。

第 20 章的稠密图的最小生成树算法相当陈旧，但是 Dijkstra、Prim 和 Kruskal 的原始论文仍然值得一读。Graham 和 Hell 给出了该问题的一种全面而又有趣的历史。Chazelle 的论文是线性 MST 算法的艺术呈现。

Ahuja、Mangnanti 和 Orlin 的著作完整地论述了网络流算法（和最短路径算法）。关于第 21 章和第 22 章的中所涵盖的几乎每个专题的进一步的信息也可在那本书中找到。还有一些资料源于 Papadimitriou 和 Steiglitz 的经典著作。书中大部分的内容更多的是关于高级专题。它详细地论述了我们讨论的大部分算法。这两本书都很涵盖了研究文献的非常广泛和深入的信息。Ford 和 Fulkerson 的经典著作作为引人基本概念仍然值得一读。

我们已经对第 8 部分的大量的高级专题作了入门性的介绍，这些专题包括可归约性、难解性和线型规划，还有一些其他的专题。该参考文献列表注重对内容的详尽介绍，而非注重内容的公平性。算法的正文对它们作了说明，Papadimitriou 和 Steiglitz 的著作作了深入的介绍。关于这些专题还有很多其他著作和大量的文献。

R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, *Network Flows: Theory, Algorithms, and Applications*, Prentice Hall, 1993.

B. Chazelle, "A minimum spanning tree algorithm with inverse-Ackermann type complexity," *Journal of the ACM*, 47 (2000).

T. H. Corman, C. L. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, MIT Press and McGram-Hill, 1990.

E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, 1 (1959).

P. Erdös and A. Renyi, "On the evolution of random graphs," *Magyar Tud. Akad, Mat. Kutato Int Kozl*, 5 (1960).

S. Even, *Graph Algorithms*, Computer Science Press, 1979.

L. R. Ford and D. R. Fulkerson, *Flows in Networks*, Princeton University Press, 1962.

H. N. Gabow, "Path-based depth-first search for string and biconnected components," *Information Processing Letters*, 74 (2000).

R. L. Graham and P. Hell, "On the history of the minimum spanning tree problem," *Annual of the History of Computing*, 7 (1985).

D. B. Johnson, "Efficient shortest path algorithms," Journal of the ACM, 24 (1977).

D. E. Knuth, *The Art of Computer Programming. Volume 1: Fundamental Algorithms*, third edition, Addison-Wesley, 1997.

J. R. Kruskal Jr. , "On the shortest spanning subtree of a graph and the traveling salesman problem," *Proceedings AMS*, 7, 1 (1956).

K. Mehlhorn, *Data Structures and Algorithms* 2: *NP-Completeness and Graph Algorithms*, Springer-Verlag, 1984.

C. H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization*: *Algorithms and Complexity*, Prentice-Hall, 1982.

R. C. Prim, "Shortest connection networks and some generalizations," *Bell System Technical Journal*, 36 (1957).

R. E. Tarjan, "Depth-first search and linear graph algorithms," *SIAM Journal on Computing*, 1, 2 (1972).

R. E. Tarjan, *Data Structures and Network Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.

经典推荐



算法导论（原书第2版）

作　者：Thomas H.Cormen 等
译　者：潘金贵 顾铁成 等
书　号：7-111-18777-6
定　价：85.00元
■2006、2007　CSDN、《程序员》杂志评选的十大IT好书之一，算法中的经典权威之作



编译原理（第2版）

作　者：Alfred V.Aho,Monica S.Lam,
Ravi Sethi,Jeffrey D.Ullman
译　者：赵建华 等
书　号：978-7-111-25121-7
■编译领域无可替代的经典著作，被广大计算机专业人士誉为"龙书"



自动机理论、语言和计算导论（原书第3版）

作　者：John E.Hopcroft,Rajeev Motwani,
Jeffrey D.Ullman
译　者：孙家骕 等
中文版：978-7-111-24035-8，49.00元
英文版：978-7-111-22392-4，59.00元
■1996年图灵奖得主经典巨著升级版



分布式系统：概念与设计（原书第4版）

作　者：George Coulouris，Jean Dollimore，
Tim Kindberg
译　者：金蓓弘 曹冬磊
中文版：978-7-111-22438-9，69.00元
英文版：7-111-17366-X，89.00元
■本书是衡量所有其他分布式系统教材的标准



数据库系统概念（原书第5版）

作　者：Abraham Silberschatz，
Henry F.Korth，S. Sudarshan
译　者：杨冬青 马秀莉 唐世渭
中文版：7-111-19687-2，69.50元
本科教学版：978-7-111-23422-7，45.00元
■数据库系统方面的经典教材，被美誉为"帆船书"



软件工程：实践者的研究方法（原书第6版）

作　者：Roger S.Pressman
译　者：郑人杰 等
中文版：7-111-19400-4，69.00元
本科教学版：978-7-111-23443-2，49.00元
英文精编版：978-7-111-24138-6，65.00元
■全球上百所大学和学院采用，最受欢迎的软件工程指南

教师服务登记表

尊敬的老师：

您好！感谢您购买我们出版的_____教材。

机械工业出版社华章公司为了进一步加强与高校教师的联系与沟通，更好地为高校教师服务，特制此表，请您填妥后发回给我们，我们将定期向您寄送华章公司最新的图书出版信息！感谢合作！

个人资料（请用正楷完整填写）

<table>
<tr><td>教师姓名</td><td></td><td>□先生
□女士</td><td>出生年月</td><td></td><td>职务</td><td></td><td colspan="2">职称：□教授 □副教授
□讲师 □助教 □其他</td></tr>
<tr><td>学校</td><td colspan="2"></td><td>学院</td><td colspan="2"></td><td>系别</td><td colspan="2"></td></tr>
<tr><td rowspan="2">联系
电话</td><td colspan="3" rowspan="2">办公：
宅电：
移动：</td><td>联系地址
及邮编</td><td colspan="3"></td></tr>
<tr><td>E-mail</td><td colspan="3"></td></tr>
<tr><td>学历</td><td></td><td>毕业院校</td><td colspan="2"></td><td>国外进修及讲学经历</td><td colspan="3"></td></tr>
<tr><td>研究领域</td><td colspan="8"></td></tr>
<tr><td colspan="2">主讲课程</td><td colspan="3">现用教材名</td><td>作者及
出版社</td><td>共同授
课教师</td><td colspan="2">教材满意度</td></tr>
<tr><td colspan="2">课程：
□专 □本 □研
人数： 学期：□春□秋</td><td colspan="3"></td><td></td><td></td><td colspan="2">□满意 □一般
□不满意 □希望更换</td></tr>
<tr><td colspan="2">课程：
□专 □本 □研
人数： 学期：□春□秋</td><td colspan="3"></td><td></td><td></td><td colspan="2">□满意 □一般
□不满意 □希望更换</td></tr>
<tr><td colspan="9">样书申请</td></tr>
<tr><td>已出版著作</td><td colspan="3"></td><td>已出版译作</td><td colspan="4"></td></tr>
<tr><td colspan="3">是否愿意从事翻译/著作工作 □是 □否</td><td>方向</td><td colspan="5"></td></tr>
<tr><td>意见和建议</td><td colspan="8"></td></tr>
</table>

填妥后请选择以下任何一种方式将此表返回：（如方便请赐名片）

地 址：北京市西城区百万庄南街1号 华章公司营销中心 邮编：100037

电 话：(010) 68353079 88378995 传真：(010)68995260

E-mail:hzedu@hzbook.com markerting@hzbook.com 图书详情可登录http://www.hzbook.com网站查询