Algorithms
THIRD EDITION
IN C

Part 5

GRAPH ALGORITHMS

ROBERT SEDGEWICK

IPR2023-00124
KASPERSKY EX 1036 Page 1

Algorithms Third Edition

in C

IPR2023-00124
KASPERSKY EX 1036 Page 2

Part 5 Graphs

Robert Sedgewick
Princeton University

Addison-Wesley

Boston • San Francisco • New York • Toronto • Montreal
London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

IPR2023-00124
KASPERSKY EX 1036 Page 3

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book and we were aware of a trademark claim, the designations have been printed in initial capital letters or all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

Copyright © 2002 by Addison-Wesley

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher. Printed in the United States of America. Published simultaneously in Canada.

The publisher offers discounts on this book when ordered in quantity for special sales. For more information, please contact: Pearson Education Corporate Sales Division, One Lake Street, Upper Saddle River, NJ 07458, (800) 382-3419, corpsales@pearson-techgroup.com.

Visit usonthe Web at www.awl.com/cseng/ (http://www.awl.com/cseng/).

Library of Congress Cataloging-in-Publication Data

Sedgewick, Robert, 1946 –
Algorithms in C / Robert Sedgewick.—3d ed.
500 p. 24 cm.
Includes bibliographical references and index.
Contents: v. 2, pt. 5. Graph algorithms
1. C (Computer program language) 2. Computer algorithms.
I. Title.
QA76.73.C15S43 2002
005.13’3—dc21 97-23418
CIP

ISBN 0-201-31663-3

Text printed on recycled paper
1 2 3 4 5 6 7 8 9 10 – **DOC** – 0504030201
First printing, August 2001

Please buy me:
http://www.amazon.com/s/?field-keywords=Algorithms-Part-Graph-Edition-Pt-5

Preface

GRAPHS AND GRAPH algorithms are pervasive in modern computing applications. This book describes the most important known methods for solving the graph-processing problems that arise in practice. Its primary aim is to make these methods and the basic principles behind them accessible to the growing number of people in need of knowing them. The material is developed from first principles, starting with basic information and working through classical methods up through modern techniques that are still under development. Carefully chosen examples, detailed figures, and complete implementations supplement thorough descriptions of algorithms and applications.

ALGORITHMS

This book is the second of three volumes that are intended to survey the most important computer algorithms in use today. The first volume (Parts 1–4) covers fundamental concepts (Part 1), data structures (Part 2), sorting algorithms (Part 3), and searching algorithms (Part 4); this volume (Part 5) covers graphs and graph algorithms; and the (yet to be published) third volume (Parts 6–8) covers strings (Part 6), computational geometry (Part 7), and advanced algorithms and applications (Part 8).

The books are useful as texts early in the computer science curriculum, after students have acquired basic programming skills and familiarity with computer systems, but before they have taken specialized courses in advanced areas of computer science or computer applications. The books also are useful for self-study or as a reference for people engaged in the development of computer systems or applications programs because they contain implementations of useful algorithms and detailed information on these algorithms' performance characteristics. The broad perspective taken makes the series an appropriate introduction to the field.

Together the three volumes comprise the *Third Edition* of a book that has been widely used by students and programmers around the world for many years. I have completely rewritten the text for this edition, and I have added thousands of new exercises, hundreds of new figures, dozens of new programs, and detailed commentary on all the figures and programs. This new material provides both coverage of new topics and fuller explanations of many of the classic algorithms. A new emphasis on abstract data types throughout the books makes the programs more broadly useful and relevant in modern object-oriented programming environments. People who have read previous editions will find a wealth of new information throughout; all readers will find a wealth of pedagogical material that provides effective access to essential concepts.

IPR2023-00124
KASPERSKY EX 1036 Page 6

These books are not just for programmers and computer-science students. Nearly everyone who uses a computer wants it to run faster or to solve larger problems. The algorithms that we consider represent a body of knowledge developed during the last 50 years that has become indispensable in the efficient use of the computer for a broad variety of applications. From *N* -body simulation problems in physics to genetic-sequencing problems in molecular biology, the basic methods described here have become essential in scientific research; and from database systems to Internet search engines, they have become essential parts of modern software systems. As the scope of computer applications becomes more widespread, so grows the impact of basic algorithms, particularly the fundamental graph algorithms covered in this volume. The goal of this book is to serve as a resource so that students and professionals can know and make intelligent use of graph algorithms as the need arises in whatever computer application they might undertake.

SCOPE

This book, *Algorithms in C, Third Edition, Part 5: Graph Algorithms*, contains six chapters that cover graph properties and types, graph search, directed graphs, minimal spanning trees, shortest paths, and networks. The descriptions here are intended to give readers an understanding of the basic properties of as broad a range of fundamental graph algorithms as possible.

You will most appreciate the material here if you have had a course covering basic principles of algorithm design and analysis and programming experience in a high-level language such as C, Java, or C++. *Algorithms in C, Third Edition, Parts 1–4* is certainly adequate preparation. This volume assumes basic knowledge about arrays, linked lists, and ADT design, and makes uses of priority-queue, symbol-table, and union-find ADTs—all of which are described in detail in Parts 1–4 (and in many other introductory texts on algorithms and data structures).

Basic properties of graphs and graph algorithms are developed from first principles, but full understanding of the properties of the algorithms can lead to deep and difficult mathematics. Although the discussion of advanced mathematical concepts is brief, general, and descriptive, you certainly need a higher level of mathematical maturity to appreciate graph algorithms than you do for the topics in Parts 1–4. Still, readers at various levels of mathematical maturity will be able to profit from this book. The topic dictates this approach: some elementary graph algorithms that should be understood and used by everyone differ only slightly from some advanced algorithms that are not understood by anyone. The primary intent here is to place important algorithms in context with other methods throughout the book, not to teach all of the mathematical material. But the rigorous treatment demanded by good mathematics often leads us to good programs, so I have

IPR2023-00124
KASPERSKY EX 1036 Page 7

tried to provide a balance between the formal treatment favored by theoreticians and the coverage needed by practitioners, without sacrificing rigor.

USE IN THE CURRICULUM

There is a great deal of flexibility in how the material here can be taught, depending on the taste of the instructor and the preparation of the students. The algorithms described have found widespread use for years, and represent an essential body of knowledge for both the practicing programmer and the computer science student. There is sufficient coverage of basic material for the book to be used in a course on data structures and algorithms, and there is sufficient detail and coverage of advanced material for the book to be used for a course on graph algorithms. Some instructors may wish to emphasize implementations and practical concerns; others may wish to emphasize analysis and theoretical concepts.

For a more comprehensive course, this book is also available in a special bundle with Parts 1–4; thereby instructors can cover fundamentals, data structures, sorting, searching, and graph algorithms in one consistent style. A complete set of slide masters for use in lectures, sample programming assignments, interactive exercises for students, and other course materials may be found by accessing the book's home page.

The exercises—nearly all of which are new to this edition—fall into several types. Some are intended to test understanding of material in the text, and simply ask readers to work through an example or to apply concepts described in the text. Others involve implementing and putting together the algorithms, or running empirical studies to compare variants of the algorithms and to learn their properties. Still other exercises are a repository for important information at a level of detail that is not appropriate for the text. Reading and thinking about the exercises will pay dividends for every reader.

ALGORITHMS OF PRACTICAL USE

Anyone wanting to use a computer more effectively can use this book for reference or for self-study. People with programming experience can find information on specific topics throughout the book. To a large extent, you can read the individual chapters in the book independently of the others, although, in some cases, algorithms in one chapter make use of methods from a previous chapter.

The orientation of the book is to study algorithms likely to be of practical use. The book provides information about the tools of the trade to the point that readers can confidently implement, debug, and put to work algorithms to solve a problem or to provide function-

ality in an application. Full implementations of the methods discussed are included, as are descriptions of the operations of these programs on a consistent set of examples. Because we work with real code, rather than write pseudo-code, the programs can be put to practical use quickly. Program listings are available from the book's home page.

Indeed, one practical application of the algorithms has been to produce the hundreds of figures throughout the book. Many algorithms are brought to light on an intuitive level through the visual dimension provided by these figures.

Characteristics of the algorithms and of the situations in which they might be useful are discussed in detail. Although not emphasized, connections to the analysis of algorithms and theoretical computer science are developed in context. When appropriate, empirical and analytic results are presented to illustrate why certain algorithms are preferred. When interesting, the relationship of the practical algorithms being discussed to purely theoretical results is described. Specific information on performance characteristics of algorithms and implementations is synthesized, encapsulated, and discussed throughout the book.

PROGRAMMING LANGUAGE

The programming language used for all of the implementations is C (versions of the book in C++ and Java are under development). Any particular language has advantages and disadvantages; we use C in this book because it is widely available and provides the features needed for the implementations here. The programs can be translated easily to other modern programming languages because relatively few constructs are unique to C. We use standard C idioms when appropriate, but this book is not intended to be a reference work on C programming.

We strive for elegant, compact, and portable implementations, but we take the point of view that efficiency matters, so we try to be aware of the code's performance characteristics at all stages of development. There are many new programs in this edition, and many of the old ones have been reworked, primarily to make them more readily useful as abstract-data-type implementations. Extensive comparative empirical tests on the programs are discussed throughout the book.

A goal of this book is to present the algorithms in as simple and direct a form as possible. The style is consistent whenever possible so that similar programs look similar. For many of the algorithms, the similarities remain regardless of which language is used: Dijkstra's algorithm (to pick one prominent example) is Dijkstra's algorithm, whether expressed in Algol-60, Basic, Fortran, Smalltalk, Ada, Pascal, C, C++, Modula-3, PostScript, Java, or

IPR2023-00124
KASPERSKY EX 1036 Page 9

any of the countless other programming languages and environments in which it has proved to be an effective graph-processing method.

ACKNOWLEDGMENTS

Many people gave me helpful feedback on earlier versions of this book. In particular, hundreds of students at Princeton and Brown have suffered through preliminary drafts over the years. Special thanks are due to Trina Avery and Tom Freeman for their help in producing the first edition; to Janet Incerpi for her creativity and ingenuity in persuading our early and primitive digital computerized typesetting hardware and software to produce the first edition; to Marc Brown for his part in the algorithm visualization research that was the genesis of so many of the figures in the book; to Dave Hanson for his willingness to answer all of my questions about C; and to Kevin Wayne, for patiently answering my basic questions about networks. I would also like to thank the many readers who have provided me with detailed comments about various editions, including Guy Almes, Jon Bentley, Marc Brown, Jay Gischer, Allan Heydon, Kennedy Lemke, Udi Manber, Dana Richards, John Reif, M. Rosenfeld, Stephen Seidman, Michael Quinn, and William Ward.

To produce this new edition, I have had the pleasure of working with Peter Gordon and Helen Goldstein at Addison-Wesley, who have patiently shepherded this project as it has evolved from a standard update to a massive rewrite. It has also been my pleasure to work with several other members of the professional staff at Addison-Wesley. The nature of this project made the book a somewhat unusual challenge for many of them, and I much appreciate their forbearance.

I have gained two new mentors in writing this book, and particularly want to express my appreciation to them. First, Steve Summit carefully checked early versions of the manuscript on a technical level, and provided me with literally thousands of detailed comments, particularly on the programs. Steve clearly understood my goal of providing elegant, efficient, and effective implementations, and his comments not only helped me to provide a measure of consistency across the implementations, but also helped me to improve many of them substantially. Second, Lyn Dupre also provided me with thousands of detailed comments on the manuscript, which were invaluable in helping me not only to correct and avoid grammatical errors, but also—more important—to find a consistent and coherent writing style that helps bind together the daunting mass of technical material here. I am extremely grateful for the opportunity to learn from Steve and Lyn—their input was vital in the development of this book.

Much of what I have written here I have learned from the teaching and writings of Don Knuth, my advisor at Stanford. Although Don had no direct influence on this work, his

presence may be felt in the book, for it was he who put the study of algorithms on the scientific footing that makes a work such as this possible. My friend and colleague Philippe Flajolet, who has been a major force in the development of the analysis of algorithms as a mature research area, has had a similar influence on this work.

I am deeply thankful for the support of Princeton University, Brown University, and the Institut National de Recherce en Informatique et Automatique (INRIA), where I did most of the work on the books; and of the Institute for Defense Analyses and the Xerox Palo Alto Research Center, where I did some work on the books while visiting. Many parts of these books are dependent on research that has been generously supported by the National Science Foundation and the Office of Naval Research. Finally, I thank Bill Bowen, Aaron Lemonick, and Neil Rudenstine for their support in building an academic environment at Princeton in which I was able to prepare this book, despite my numerous other responsibilities.

Robert Sedgewick
Marly-le-Roi, France, February, 1983
Princeton, New Jersey, January, 1990
Jamestown, Rhode Island, May, 2001

To Adam, Andrew, Brett, Robbie, and especially Linda

Notes on Exercises

Classifying exercises is an activity fraught with peril, because readers of a book such as this come to the material with various levels of knowledge and experience. Nonetheless, guidance is appropriate, so many of the exercises carry one of four annotations, to help you decide how to approach them.

Exercises that *test your understanding* of the material are marked with an open triangle, as follows:

▷ **17.2** Consider the graph

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

Draw the its DFS tree and use the tree to find the graph's bridges and edge-connected components.

Most often, such exercises relate directly to examples in the text. They should present no special difficulty, but working them might teach you a fact or concept that may have eluded you when you read the text.

Exercises that *add new and thought-provoking* information to the material are marked with an open circle, as follows:

○ **18.2** Write a program that counts the number of different possible results of topologically sorting a given DAG.

Such exercises encourage you to think about an important concept that is related to the material in the text, or to answer a question that may have occurred to you when you read the text. You may find it worthwhile to read these exercises, even if you do not have the time to work them through.

Exercises that are intended to *challenge you* are marked with a black dot, as follows:

● **19.2** Describe how you would find the MST of a graph so large that only V edges can fit into main memory at once.

Such exercises may require a substantial amount of time to complete, depending upon your experience. Generally, the most productive approach is to work on them in a few different sittings.

A few exercises that are *extremely difficult* (by comparison with most others) are marked

with two black dots, as follows:

●● **20.2** Develop a reasonable generator for random graphs with V vertices and E edges such that the running time of the PFS implementation of Dijkstra's algorithm is nonlinear.

These exercises are similar to questions that might be addressed in the research literature, but the material in the book may prepare you to enjoy trying to solve them (and perhaps succeeding).

The annotations are intended to be neutral with respect to your programming and mathematical ability. Those exercises that require expertise in programming or in mathematical analysis are self-evident. All readers are encouraged to test their understanding of the algorithms by implementing them. Still, an exercise such as this one is straightforward for a practicing programmer or a student in a programming course, but may require substantial work for someone who has not recently programmed:

● **17.2** Write a program that generates V random points in the plane, then builds a network with edges (in both directions) connecting all pairs of points within a given distance d of one another (see Program 3.20), setting each edge's weight to the distance between the two points that it connects. Determine how to set d so that the expected number of edges is E.

In a similar vein, all readers are encouraged to strive to appreciate the analytic underpinnings of our knowledge about properties of algorithms. Still, an exercise such as this one is straightforward for a scientist or a student in a discrete mathematics course, but may require substantial work for someone who has not recently done mathematical analysis:

○ **18.2** How many digraphs correspond to each undirected graph with V vertices and E edges?

There are far too many exercises for you to read and assimilate them all; my hope is that there are enough exercises here to stimulate you to strive to come to a broader understanding on the topics that interest you than you can glean by simply reading the text.

Contents

Graph Algorithms

Chapter 17. Graph Properties and Types **1**

17.1 Glossary - 4

17.2 Graph ADT - 17

17.3 Adjacency-Matrix Representation - 23

17.4 Adjacency-Lists Representation - 30

17.5 Variations, Extensions, and Costs - 35

17.6 Graph Generators - 44

17.7 Simple, Euler, and Hamilton Paths - 58

17.8 Graph-Processing Problems - 77

Chapter 18. Graph Search **88**

18.1 Exploring a Maze - 89

18.2 Depth-First Search - 94

18.3 Graph-Search ADT Functions - 102

18.4 Properties of DFS Forests - 108

18.5 DFS Algorithms - 117

18.6 Separability and Biconnectivity - 125

18.7 Breadth-First Search - 135

18.8 Generalized Graph Search - 145

18.9 Analysis of Graph Algorithms - 155

Chapter 19. Digraphs and DAGs **162**

19.1 Glossary and Rules of the Game - 166

19.2 Anatomy of DFS in Digraphs - 178

19.3 Reachability and Transitive Closure - 190

19.4 Equivalence Relations and Partial Orders - 205

19.5 DAGs - 209

19.6 Topological Sorting - 216

19.7 Reachability in DAGs - 229

19.8 Strong Components in Digraphs - 234

19.9 Transitive Closure Revisited - 245

19.10 Perspective - 249

Chapter 20. Minimum Spanning Trees **255**

20.1 Representations - 259

20.2 Underlying Principles of MST Algorithms - 266

20.3 Prim’s Algorithm and Priority-First Search - 275

20.4 Kruskal’s Algorithm - 288

20.5 Boruvka’s Algorithm - 295

20.6 Comparisons and Improvements - 301

20.7 Euclidean MST - 308

Chapter 21. Shortest Paths **313**

21.1 Underlying Principles - 322

21.2 Dijkstra’s algorithm - 330

21.3 All-Pairs Shortest Paths - 341

21.4 Shortest Paths in Acyclic Networks - 351

21.5 Euclidean Networks - 360

21.6 Reduction - 367

21.7 Negative Weights - 386

21.8 Perspective - 407

Chapter 22. Network Flows **409**

22.1 Flow Networks - 417

22.2 Augmenting-Path Maxflow Algorithms - 432

22.3 Preflow-Push Maxflow Algorithms - 459

22.4 Maxflow Reductions - 477

22.5 Mincost Flows - 502

22.6 Network Simplex Algorithm - 512

22.7 Mincost-Flow Reductions - 533

22.8 Perspective - 545

References for Part Five **549**

Index **/dev/null**

IPR2023-00124
KASPERSKY EX 1036 Page 20

Chapter Seventeen
Graph Properties and Types

MANY COMPUTATIONAL APPLICATIONS naturally involve not just a set of *items*, but also a set of *connections* between pairs of those items. The relationships implied by these connections lead immediately to a host of natural questions: Is there a way to get from one item to another by following the connections? How many other items can be reached from a given item? What is the best way to get from this item to this other item?

To model such situations, we use abstract objects called *graphs*. In this chapter, we examine basic properties of graphs in detail, setting the stage for us to study a variety of algorithms that are useful for answering questions of the type just posed. These algorithms make effective use of many of the computational tools that we considered in Parts 1 through 4. They also serve as the basis for attacking problems in important applications whose solution we could not even contemplate without good algorithmic technology.

Graph theory, a major branch of combinatorial mathematics, has been studied intensively for hundreds of years. Many important and useful properties of graphs have been proved, yet many difficult problems remain unresolved. In this book, while recognizing that there is much still to be learned, we draw from this vast body of knowledge about graphs what we need to understand and use a broad variety of useful and fundamental algorithms.

Like so many of the other problem domains that we have studied, the algorithmic investigation of graphs is relatively recent. Although a few of the fundamental algorithms are old, the majority of the interesting ones have been discovered within the last few decades. Even the simplest graph algorithms lead to useful computer programs, and the nontrivial algorithms that we examine are among the most elegant and interesting algorithms known.

To illustrate the diversity of applications that involve graph processing, we begin our exploration of algorithms in this fertile area by considering several examples.

Maps A person who is planning a trip may need to answer questions such as, “What is the *least expensive* way to get from Princeton to San Jose?” A person more interested in time than in money may need to know the answer to the question “What is the *fastest* way to get from Princeton to San Jose?” To answer such questions, we process information about *connections* (travel routes) between *items* (towns and cities).

Hypertexts When we browse the Web, we encounter documents that contain references (links) to other documents, and we move from document to document by clicking on the

links. The entire web is a graph, where the items are documents and the connections are links. Graph-processing algorithms are essential components of the search engines that help us locate information on the web.

Circuits An electric circuit comprises elements such as transistors, resistors, and capacitors that are intricately wired together. We use computers to control machines that make circuits, and to check that the circuits perform desired functions. We need to answer simple questions such as, "Is a short-circuit present?" as well as complicated questions such as, "Can we lay out this circuit on a chip without making any wires cross?" In this case, the answer to the first question depends on only the properties of the connections (wires), whereas the answer to the second question requires detailed information about the wires, the items that those wires connect, and the physical constraints of the chip.

Schedules A manufacturing process requires a variety of tasks to be performed, under a set of constraints that specifies that certain tasks cannot be started until certain other tasks have been completed. We represent the constraints as connections between the tasks (items), and we are faced with a classical *scheduling* problem: How do we schedule the tasks such that we both respect the given constraints and complete the whole process in the least amount of time?

Transactions A telephone company maintains a database of telephone-call traffic. Here the connections represent telephone calls. We are interested in knowing about the nature of the interconnection structure because we want to lay wires and build switches that can handle the traffic efficiently. As another example, a financial institution tracks buy/sell orders in a market. A connection in this situation represents the transfer of cash between two customers. Knowledge of the nature of the connection structure in this instance may enhance our understanding of the nature of the market.

Matching Students apply for positions in selective institutions such as social clubs, universities, or medical schools. Items correspond to the students and the institutions; connections correspond to the applications. We want to discover methods for matching interested students with available positions.

Networks A computer network consists of interconnected sites that send, forward, and receive messages of various types. We are interested not just in knowing that it is possible to get a message from every site to every other site, but also in maintaining this connectivity for all pairs of sites as the network changes. For example, we might wish to check a given network to be sure that no small set of sites or connections is so critical that losing it would disconnect any remaining pair of sites.

Program structure A compiler builds graphs to represent the call structure of a large

software system. The items are the various functions or modules that comprise the system; connections are associated either with the possibility that one function might call another (static analysis) or with actual calls while the system is in operation (dynamic analysis). We need to analyze the graph to determine how best to allocate resources to the program most efficiently.

These examples indicate the range of applications for which graphs are the appropriate abstraction, and also the range of computational problems that we might encounter when we work with graphs. Such problems will be our focus in this book. In many of these applications as they are encountered in practice, the volume of data involved is truly huge, and efficient algorithms make the difference between whether or not a solution is at all feasible.

We have already encountered graphs, briefly, in Part 1. Indeed, the first algorithms that we considered in detail, the union-find algorithms in Chapter 1, are prime examples of graph algorithms. We also used graphs in Chapter 3 as an illustration of applications of two-dimensional arrays and linked lists, and in Chapter 5 to illustrate the relationship between recursive programs and fundamental data structures. Any linked data structure is a representation of a graph, and some familiar algorithms for processing trees and other linked structures are special cases of graph algorithms. The purpose of this chapter is to provide a context for developing an understanding of graph algorithms ranging from the simple ones in Part 1 to the sophisticated ones in Chapters 18 through 22.

As always, we are interested in knowing which are the most efficient algorithms that solve a particular problem. The study of the performance characteristics of graph algorithms is challenging because

- The cost of an algorithm depends not just on properties of the set of items, but also on numerous properties of the set of connections (and global properties of the graph that are implied by the connections).

- Accurate models of the types of graphs that we might face are difficult to develop.

We often work with worst-case performance bounds on graph algorithms, even though they may represent pessimistic estimates on actual performance in many instances. Fortunately, as we shall see, a number of algorithms are *optimal* and involve little unnecessary work. Other algorithms consume the same resources on all graphs of a given size. We can predict accurately how such algorithms will perform in specific situations. When we cannot make such accurate predictions, we need to pay particular attention to properties of the various types of graphs that we might expect in practical applications and must assess how these properties might affect the performance of our algorithms.

We begin by working through the basic definitions of graphs and the properties of graphs, examining the standard nomenclature that is used to describe them. Following that, we define the basic ADT (abstract data type) interfaces that we use to study graph algorithms and the two most important data structures for representing graphs—the *adjacency-matrix* representation and the *adjacency-lists* representation, and various approaches to implementing basic ADT functions. Then, we consider client programs that can generate random graphs, which we can use to test our algorithms and to learn properties of graphs. All this material provides a basis for us to introduce graph-processing algorithms that solve three classical problems related to finding paths in graphs, which illustrate that the difficulty of graph problems can differ dramatically even when they might seem similar. We conclude the chapter with a review of the most important graph-processing problems that we consider in this book, placing them in context according to the difficulty of solving them.

17.1 GLOSSARY

A substantial amount of nomenclature is associated with graphs. Most of the terms have straightforward definitions, and, for reference, it is convenient to consider them in one place: here. We have already used some of these concepts when considering basic algorithms in Part 1; others of them will not become relevant until we address associated advanced algorithms in Chapters 18 through 22.

Definition 17.1*A* **graph***is a set of* **vertices***plus a set of* **edges***that connect pairs of distinct vertices (with at most one edge connecting any pair of vertices).*

We use the names 0 through V-1 for the vertices in a V-vertex graph. The main reason that we choose this system is that we can access quickly information corresponding to each vertex, using array indexing. In Section 17.6, we consider a program that uses a symbol table to establish a 1–1 mapping to associate V arbitrary vertex names with the V integers between 0 and $V - 1$. With that program in hand, we can use indices as vertex names (for notational convenience) without loss of generality. We sometimes assume that the set of vertices is defined implicitly, by taking the set of edges to define the graph and considering only those vertices that are included in at least one edge. To avoid cumbersome usage such as "the ten-vertex graph with the following set of edges," we do not explicitly mention the number of vertices when that number is clear from the context. By convention, we always denote the number of vertices in a given graph by V, and denote the number of edges by E.

We adopt as standard this definition of a graph (which we first encountered in Chapter 5), but note that it embodies two technical simplifications. First, it disallows duplicate

edges (mathematicians sometimes refer to such edges as *parallel* edges, and a graph that can contain them as a *multigraph*). Second, it disallows edges that connect vertices to themselves; such edges are called *self-loops*. Graphs that have no parallel edges or self-loops are sometimes referred to as *simple graphs*.

We use simple graphs in our formal definitions because it is easier to express their basic properties and because parallel edges and self-loops are not needed in many applications. For example, we can bound the number of edges in a simple graph with a given number of vertices.

Property 17.1 *A graph with V vertices has at most* $V(V-1)/2$ *edges.*

Proof: The total of V^2 possible pairs of vertices includes V self-loops and accounts twice for each edge between distinct vertices, so the number of edges is at most $(V^2 - V)/2 = V(V-1)/2$. ■

No such bound holds if we allow parallel edges: a graph that is not simple might consist of two vertices and billions of edges connecting them (or even a single vertex and billions of self-loops).

For some applications, we might consider the elimination of parallel edges and self-loops to be a data-processing problem that our implementations must address. For other applications, ensuring that a given set of edges represents a simple graph may not be worth the trouble. Throughout the book, whenever it is more convenient to address an application or to develop an algorithm by using an extended definition that includes parallel edges or self-loops, we shall do so. For example, self-loops play a critical role in a classical algorithm that we will examine in Section 17.4; and parallel edges are common in the applications that we address in Chapter 22. Generally, it is clear from the context whether we intend the term "graph" to mean "simple graph" or "multigraph" or "multigraph with self-loops."

Mathematicians use the words *vertex* and *node* interchangeably, but we generally use *vertex* when discussing graphs and *node* when discussing representations—for example, in C data structures. We normally assume that a vertex can have a name and can carry other associated information. Similarly, the words *arc, edge*, and *link* are all widely used by mathematicians to describe the abstraction embodying a connection between two vertices, but we consistently use *edge* when discussing graphs and *link* when discussing C data structures.

When there is an edge connecting two vertices, we say that the vertices are *adjacent to* one another and that the edge is *incident on* both vertices. The *degree* of a vertex is the

IPR2023-00124
KASPERSKY EX 1036 Page 25

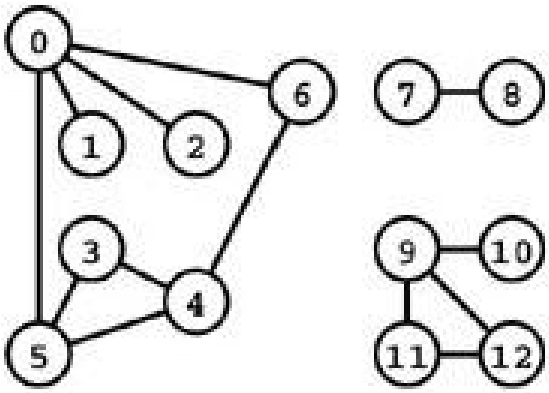
number of edges incident on it. We use the notation `v-w` to represent an edge that connects `v` and `w`; the notation `w-v` is an alternative way to represent the same edge.

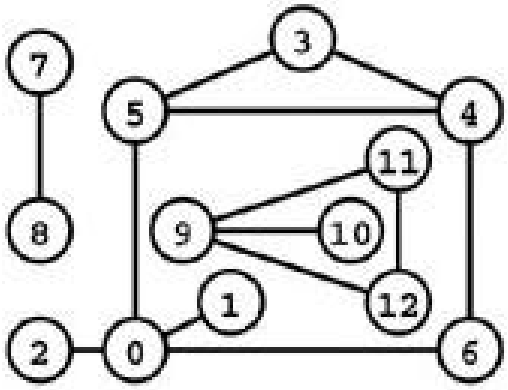
A *subgraph* is a subset of a graph's edges (and associated vertices) that constitutes a graph. Many computational tasks involve identifying subgraphs of various types. If we identify a subset of a graph's vertices, we call that subset, together with all edges that connect two of its members, the *induced subgraph* associated with those vertices.

We can draw a graph by marking points for the vertices and drawing lines connecting them for the edges. A drawing gives us intuition about the structure of the graph; but this intuition can be misleading, because the graph is defined independently of the representation. For example, the two drawings in *Figure* 17.1 and the list of edges represent the same graph, because the graph is only its (unordered) set of vertices and its (unordered) set of edges (pairs of vertices)—nothing more. Although it suffices to consider a graph simply as a set of edges, we examine other representations that are particularly suitable as the basis for graph data structures in Section 17.4.

Placing the vertices of a given graph on the plane and drawing them and the edges that connect them is known as *graph drawing*. The possible vertex placements, edge-drawing styles, and aesthetic constraints on the drawing are limitless. Graph-drawing algorithms that respect various natural constraints have been studied heavily and have many successful applications (*see reference section*). For example, one of the simplest constraints is to insist that edges do not intersect. A *planar graph* is one that can be drawn in the plane without any edges crossing. Determining whether or not a graph is planar is a fascinating algorithmic problem that we discuss briefly in Section 17.8. Being able to produce a helpful visual representation is a useful goal, and graph drawing is a fascinating field of study, but successful drawings are often difficult to realize. Many graphs that have huge numbers of vertices and edges are abstract objects for which no suitable drawing is feasible.

Figure 17.1 Three Different Representations of the Same Graph *A graph is defined by its vertices and its edges, not by the way that we choose to draw it. These two drawings depict the same graph, as does the list of edges (bottom), given the additional information that the graph has 13 vertices labeled 0 through 12.*






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

For some applications, such as graphs that represent maps or circuits, a graph drawing can carry considerable information because the vertices correspond to points in the plane and the distances between them are relevant. We refer to such graphs as *Euclidean graphs*. For many other applications, such as graphs that represent relationships or schedules, the graphs simply embody connectivity information, and no particular geometric placement of vertices is ever implied. We consider examples of algorithms that exploit the geometric information in Euclidean graphs in Chapters 20 and 21, but we primarily work with algorithms that make no use of any geometric information, and stress that graphs are generally independent of any particular representation in a drawing or in a computer.

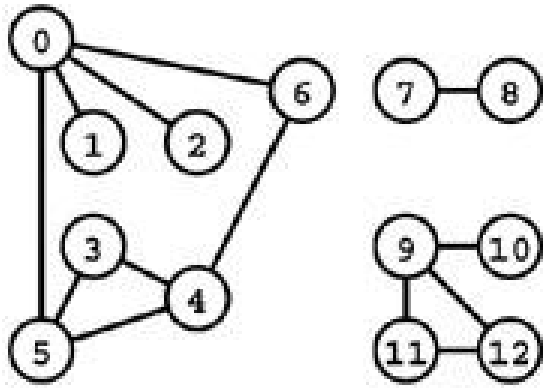
Focusing solely on the connections themselves, we might wish to view the vertex labels as merely a notational convenience, and to regard two graphs as being the same if they differ

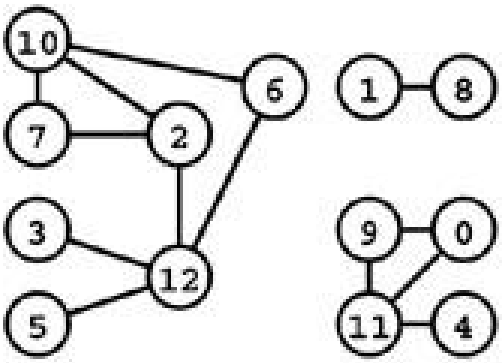
in only the vertex labels. Two graphs are *isomorphic* if we can change the vertex labels on one to make its set of edges identical to the other. Determining whether or not two graphs are isomorphic is a difficult computational problem (see *Figure* 17.2 and Exercise 17.5). It is challenging because there are $V!$ possible ways to label the vertices—far too many for us to try all the possibilities. Therefore, despite the potential appeal of reducing the number of different graph structures that we have to consider by treating isomorphic graphs as identical structures, we rarely do so.

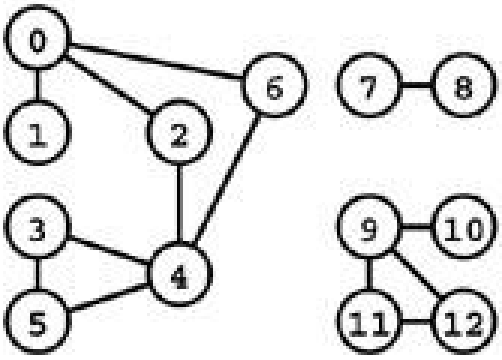
As we saw with trees in Chapter 5, we are often interested in basic structural properties that we can deduce by considering specific sequences of edges in a graph.

Definition 17.2 *A* **path** *in a graph is a sequence of vertices in which each successive vertex (after the first) is adjacent to its predecessor in the path. In a* **simple path**, *the vertices and edges are distinct. A* **cycle** *is a* `path` *that is simple except that the first and final vertices are the same.*

Figure 17.2 Graph Isomorphism Examples *The top two graphs are isomorphic because we can relabel the vertices to make the two sets of edges identical (to make the middle graph the same as the top graph, change* `10` *to* `4, 7` *to* `3, 2` *to* `5, 3` *to* `1, 12` *to* `0, 5` *to* `2, 9` *to* `11, 0` *to* `12, 11` *to* `9, 1` *to* `7`*, and* `4` *to* `10`*). The bottom graph is not isomorphic to the others because there is no way to relabel the vertices to make its set of edges identical to either.*



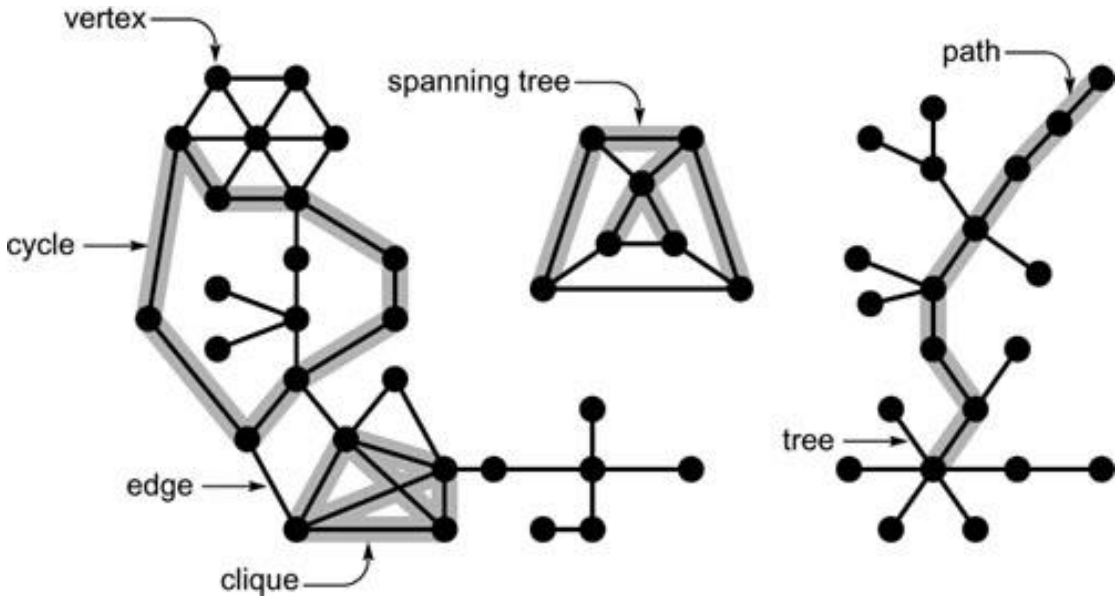




We sometimes use the term *cyclic path* to refer to a `path` whose first and final vertices are the same (and is otherwise not necessarily simple); and we use the term *tour* to refer to a cyclic `path` that includes every vertex. An equivalent way to define a `path` is as the sequence of *edges* that connect the successive vertices. We emphasize this in our notation by connecting vertex names in a `path` in the same way as we connect them in an edge. For example, the simple paths in *Figure* 17.1 include `3-4-6-0-2`, and `9-12-11`, and the cycles in the graph include `0-6-4-3-5-0` and `5-4-3-5`. We define the *length* of a `path` or a cycle to be its number of edges.

Figure 17.3Graph Terminology *This graph has 55 vertices, 70 edges, and 3*

connected components. One of the connected components is a tree (right). The graph has many cycles, one of which is highlighted in the large connected component (left). The diagram also depicts a spanning tree in the small connected component (center). The graph as a whole does not have a spanning tree, because it is not connected.



We adopt the convention that each single vertex is a `path` of length 0 (a `path` from the vertex to itself with no edges on it, which is different from a self-loop). Apart from this convention, in a graph with no parallel edges and no self-loops, a pair of vertices uniquely determines an edge, paths must have on them at least two distinct vertices, and cycles must have on them at least three distinct edges and three distinct vertices.

We say that two simple paths are *disjoint* if they have no vertices in common other than, possibly, their endpoints. Placing this condition is slightly weaker than insisting that the paths have no vertices at all in common, and is useful because we can combine simple disjoint paths from s to t and t to u to get a simple disjoint `path` from s to u if s and u are different (and to get a cycle if s and u are the same). The term *vertex disjoint* is sometimes used to distinguish this condition from the stronger condition of *edge disjoint*, where we require that the paths have no *edge* in common.

Definition 17.3 *A graph is a* **connected graph** *if there is a* `path` *from every vertex to every other vertex in the graph. A graph that is not connected consists of a set of* **connected components**, *which are maximal connected subgraphs.*

The term *maximal connected subgraph* means that there is no `path` from a subgraph vertex to any vertex in the graph that is not in the subgraph. Intuitively, if the vertices were physical objects, such as knots or beads, and the edges were physical connections, such as strings or wires, a connected graph would stay in one piece if picked up by any

vertex, and a graph that is not connected comprises two or more such pieces.

Definition 17.4 *An acyclic connected graph is called a* **tree** (*see Chapter 4). A set of trees is called a* **forest**. *A* **spanning tree** *of a connected graph is a subgraph that contains all of that graph's vertices and is a single tree. A* **spanning forest** *of a graph is a subgraph that contains all of that graph's vertices and is a forest.*

For example, the graph illustrated in *Figure* 17.1 has three connected components, and is spanned by the forest `7-8 9-10 9-11 9-12 0-1 0-2 0-5 5-3 5-4 4-6` (there are many other spanning forests). *Figure* 17.3 highlights these and other features in a larger graph.

We explore further details about trees in Chapter 4, and look at various equivalent definitions. For example, a graph G with V vertices is a tree if and only if it satisfies *any* of the following four conditions:

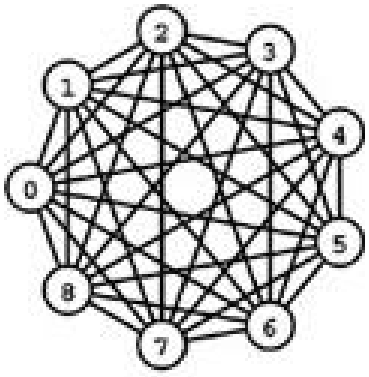
- G has $V - 1$ edges and no cycles.
- G has $V - 1$ edges and is connected.
- Exactly one simple `path` connects each pair of vertices in G.
- G is connected, but removing any edge disconnects it.

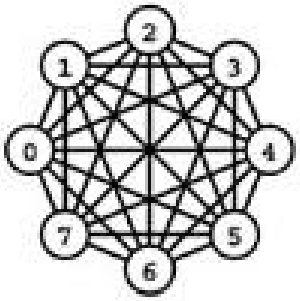
Any one of these conditions is necessary and sufficient to prove the other three, and we can develop other combinations of facts about trees from them (see Exercise 17.1). Formally, we should choose one condition to serve as a definition; informally, we let them collectively serve as the definition, and freely engage in usage such as the "acyclic connected graph" choice in Definition 17.4.

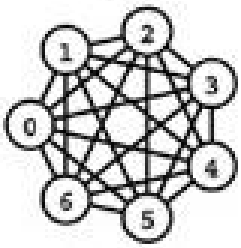
Graphs with all edges present are called *complete graphs* (see *Figure* 17.4). We define the *complement* of a graph G by starting with a complete graph that has the same set of vertices as the original graph, and removing the edges of G. The *union* of two graphs is the graph induced by the union of their sets of edges. The union of a graph and its complement is a complete graph. All graphs that have V vertices are subgraphs of the complete graph that has V vertices. The total number of different graphs that have V vertices is $2^{V(V-1)/2}$ (the number of different ways to choose a subset from the $V(V-1)/2$ possible edges). A complete subgraph is called a *clique*.

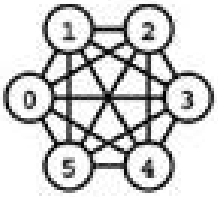
Figure 17.4 Complete Graphs *These complete graphs, with every vertex connected to every other vertex, have 10, 15, 21, 28, and 36 edges (bottom to top). Every graph with between 5 and 9 vertices (there are more than 68 billion such graphs) is a subgraph of one of these graphs.*

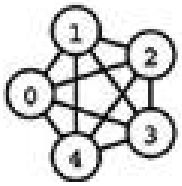
IPR2023-00124
KASPERSKY EX 1036 Page 31











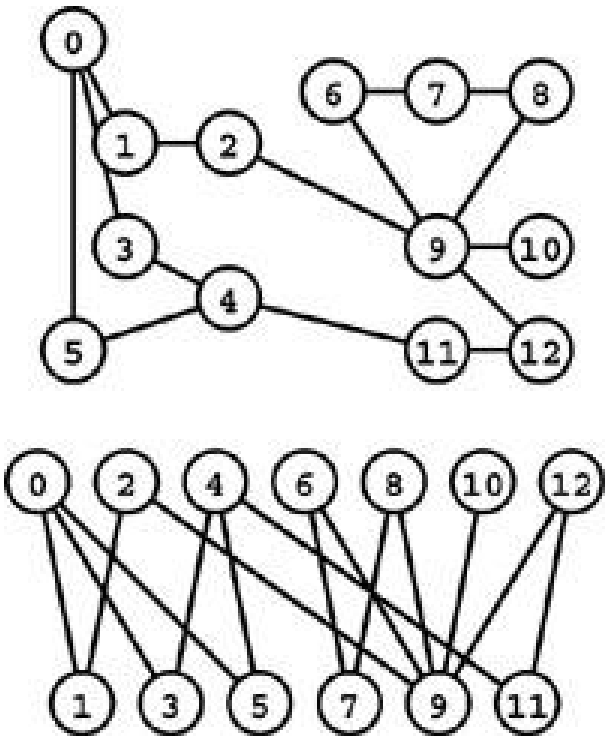
Most graphs that we encounter in practice have relatively few of the possible edges present. To quantify this concept, we define the *density* of a graph to be the average vertex degree, or $2E/V$. A *dense graph* is a graph whose average vertex degree is proportional to V; a *sparse graph* is a graph whose complement is dense. In other words, we consider a graph to be dense if E is proportional to V^2 and sparse otherwise. This asymptotic definition is not necessarily meaningful for a particular graph, but the distinction is generally clear: A graph that has millions of vertices and tens of millions of edges is certainly sparse, and a graph that has thousands of vertices and millions of edges is certainly dense. We might contemplate processing a sparse graph with billions of vertices, but a dense graph with billions of vertices would have an overwhelming number of edges.

Knowing whether a graph is sparse or dense is generally a key factor in selecting an efficient algorithm to process the graph. For example, for a given problem, we might develop one algorithm that takes about V^2 steps and another that takes about $E \lg E$ steps. These formulas tell us that the second algorithm would be better for sparse graphs, whereas the first would be preferred for dense graphs. For example, a dense graph with millions of edges might have only thousands of vertices: in this case V^2 and E would be comparable in value, and the V^2 algorithm would be 20 times faster than the $E \lg E$ algorithm. On the other hand, a sparse graph with millions of edges also has millions of vertices, so the $E \lg E$ algorithm could be *millions* of times faster than the V^2 algorithm. We could make specific tradeoffs on the basis of analyzing these formulas in more detail, but it generally suffices in practice to use the terms *sparse* and *dense* informally to help us understand fundamental performance characteristics.

When analyzing graph algorithms, we assume that V/E is bounded by above a small constant, so that we can abbreviate expressions such as $V(V + E)$ to VE. This assumption comes into play only when the number of edges is tiny in comparison to the number of vertices—a rare situation. Typically, the number of edges far exceeds the number of vertices (V/E is much less than 1).

A *bipartite graph* is a graph whose vertices we can divide into two sets such that all edges connect a vertex in one set with a vertex in the other set. *Figure* 17.5 gives an example of a bipartite graph. Bipartite graphs arise in a natural way in many situations, such as the matching problems described at the beginning of this chapter. Any subgraph of a bipartite graph is bipartite.

Figure 17.5 A Bipartite Graph *All edges in this graph connect odd-numbered vertices with even-numbered ones, so it is bipartite. The bottom diagram makes the property obvious.*



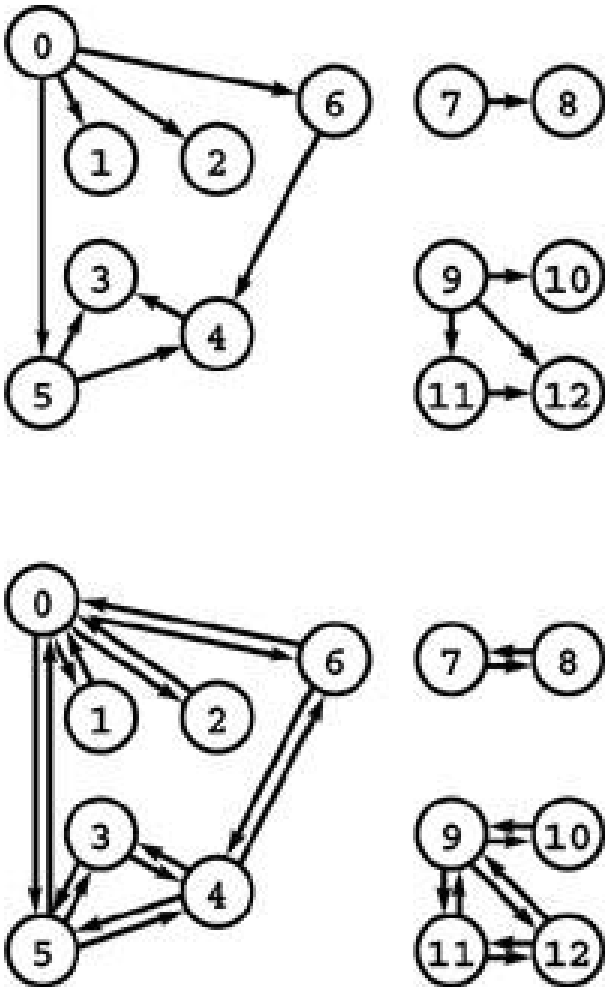
Graphs as defined to this point are called *undirected graphs*. In *directed graphs*, also known as *digraphs*, edges are one-way: we consider the pair of vertices that defines each edge to be an *ordered* pair that specifies a one-way adjacency where we think about having the ability to get from the first vertex to the second but not from the second vertex to the first. Many applications (for example, graphs that represent the web, scheduling constraints or telephone-call transactions) are naturally expressed in terms of digraphs.

We refer to edges in digraphs as *directed edges*, though that distinction is generally obvious in context (some authors reserve the term *arc* for directed edges). The first vertex in a directed edge is called the *source*; the second vertex is called the *destination*. (Some authors use the terms *head* and *tail*, respectively, to distinguish the vertices in directed edges, but we avoid this usage because of overlap with our use of the same terms in data-structure implementations.) We draw directed edges as arrows pointing from source to destination, and often say that the edge *points to* the destination. When we use the notation `v-w` in a digraph, we mean it to represent an edge that points from `v` to `w`; it is different from `w-v`, which represents an edge that points from `w` to `v`. We speak of the *indegree* and *outdegree* of a vertex (the number of edges where it is the destination and the number of edges where it is the source, respectively).

Sometimes, we are justified in thinking of an undirected graph as a digraph that has two directed edges (one in each direction); other times, it is useful to think of undirected graphs simply in terms of connections. Normally, as discussed in detail in Section 17.4, we use the same representation for directed and undirected graphs (see *Figure* 17.6).

That is, we generally maintain two representations of each edge for undirected graphs, one pointing in each direction, so that we can immediately answer questions such as, "Which vertices are connected to vertex v ?"

Figure 17.6 Two Digraphs *The drawing at the top is a representation of the example graph in Figure 17.1 interpreted as a directed graph, where we take the edges to be ordered pairs and represent them by drawing an arrow from the first vertex to the second. It is also a DAG. The drawing at the bottom is a representation of the undirected graph from Figure 17.1 that indicates the way that we usually represent undirected graphs: as digraphs with two edges corresponding to each connection (one in each direction).*



Chapter 19 is devoted to exploring the structural properties of digraphs; they are generally more complicated than the corresponding properties for undirected graphs. A *directed cycle* in a digraph is a cycle in which all adjacent vertex pairs appear in the order indicated by (directed) graph edges. A *directed acyclic graph (DAG)*, is a digraph that has no directed cycles. A DAG (an acyclic digraph) is not the same as a tree (an acyclic undirected graph). Occasionally, we refer to the *underlying undirected graph* of a digraph, meaning the undirected graph defined by the same set of edges, but where these edges

are not interpreted as directed.

Chapters 20 through 22 are generally concerned with algorithms for solving various computational problems associated with graphs in which other information is associated with the vertices and edges. In *weighted graphs*, we associate numbers (*weights*) with each edge, which generally represents a distance or cost. We also might associate a weight with each vertex, or multiple weights with each vertex and edge. In Chapter 20 we work with weighted undirected graphs; in Chapters 21 and 22 we study weighted digraphs, which we also refer to as *networks*. The algorithms in Chapter 22 solve classic problems that arise from a particular interpretation of networks known as *flow networks*.

As was evident even in Chapter 1, the combinatorial structure of graphs is extensive. This extent of this structure is all the more remarkable because it springs forth from a simple mathematical abstraction. This underlying simplicity will be reflected in much of the code that we develop for basic graph processing. However, this simplicity sometimes masks complicated dynamic properties that require deep understanding of the combinatorial properties of graphs themselves. It is often far more difficult to convince ourselves that a graph algorithm works as intended than the compact nature of the code might suggest.

Exercises

17.1 Prove that any acyclic connected graph that has V vertices has $V - 1$ edges.

▷ **17.2** Give all the connected subgraphs of the graph

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

▷ **17.3** Write down a list of the nonisomorphic cycles of the graph in *Figure* 17.1. For example, if your list contains `3-4-5-3`, it should not contain `3-5-4-3`, `4-5-3-4`, `4-3-5-4`, `5-3-4-5`, or `5-4-3-5`.

17.4 Consider the graph

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

Determine the number of connected components, give a spanning forest, list all the simple paths with at least three vertices, and list all the nonisomorphic cycles (see Exercise 17.3).

○ **17.5** Consider the graphs defined by the following four sets of edges:

```
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
```

Which of these graphs are isomorphic to one another? Which of them are planar?

17.6 Consider the more than 68 billion graphs referred to in the caption to *Figure* 17.4. What percentage of them has fewer than nine vertices?

▷ **17.7** How many different subgraphs are there in a given graph with V vertices and E edges?

17.8 Give tight upper and lower bounds on the number of connected components in graphs that have V vertices and E edges.

○ **17.9** How many different undirected graphs are there that have V vertices and E edges?

●●● **17.10** If we consider two graphs to be different only if they are not isomorphic, how many different graphs are there that have V vertices and E edges?

17.11 How many V-vertex graphs are bipartite?

17.2 GRAPH ADT

We develop our graph-processing algorithms within the context of an ADT that defines the tasks of interest, using the standard mechanisms considered in Chapter 4. *Program* 17.1 is the nucleus of the ADT interface that we use for this purpose. Basic graph representations and implementations for this ADT are the topic of Sections 17.3 through 17.5. Later in the book, whenever we consider a new graph-processing problem, we consider the algorithms that solve it and their implementations in the context of new ADT functions for this interface. This scheme allows us to address graph-processing tasks ranging from elementary maintenance functions to sophisticated solutions of difficult problems.

The interface is based on our standard mechanism that hides representations and implementations from client programs (see Section 4.8). It also includes a simple structure type definition that allows our programs to manipulate edges in a uniform way. The interface provides the basic mechanisms that allows clients to build graphs (by initializing the graph and then adding the edges), to maintain the graphs (by removing some edges and adding others), and to retrieve the graphs (in the form of an array of edges).

IPR2023-00124
KASPERSKY EX 1036 Page 37

Program 17.1 Graph ADT interface

This interface is a starting point for implementing and testing graph algorithms. Throughout this and the next several chapters, we shall add functions to this interface for solving various graph-processing problems. Various assumptions that simplify the code and other issues surrounding the design of a general-purpose graph-processing interface are discussed in the text.

The interface defines two data types: a simple `Edge` data type, including a constructor function `EDGE` that makes an `Edge` from two vertices; and a `Graph` data type, which is defined with the standard representation-independent construction from Chapter 4. The basic operations that we use to process graphs are ADT functions to create, copy, and destroy them; to add and delete edges; and to extract an edge list.

```
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);
```

The ADT in *Program* 17.1 is primarily a vehicle to allow us to develop and test algorithms; it is not a general-purpose interface. As usual, we work with the simplest interface that supports the basic graph-processing operations that we wish to consider. Defining such an interface for use in practical applications involves making numerous tradeoffs among simplicity, efficiency, and generality. We consider a few of these tradeoffs next; we address many others in the context of implementations and applications throughout this book.

We assume for simplicity that graph representations include integers V and E that contain the number of vertices and edges, respectively, so that we can refer directly to those

values by name in ADT implementations. When convenient, we make other, similar, assumptions about variables in graph representations, primarily to keep implementations compact. For convenience, we also provide the maximum possible number of vertices in the graph as an argument to the `GRAPHinit` ADT function so that implementations can allocate memory accordingly. We adopt these conventions solely to make the code compact and readable.

A slightly more general interface might provide the capability to add and remove vertices as well as edges (and might include functions that return the number of vertices and edges), making no assumptions about implementations. This design would allow for ADT implementations that grow and shrink their data structures as the graph grows and shrinks. We might also choose to work at an intermediate level of abstraction, and consider the design of interfaces that support higher-level abstract operations on graphs that we can use in implementations. We revisit this idea briefly in Section 17.5, after we consider several concrete representations and implementations.

A general graph ADT needs to take into account parallel edges and self-loops, because nothing prevents a client program from calling `GRAPHinsertE` with an edge that is already present in the graph (parallel edge) or with an edge whose two vertex indices are the same (self-loop). It might be necessary to disallow such edges in some applications, desirable to include them in other applications, and possible to ignore them in still other applications. Self-loops are trivial to handle, but parallel edges can be costly to handle, depending on the graph representation. In certain situations, adding a *remove parallel edges* ADT function might be appropriate; then, implementations can let parallel edges collect, and clients can remove or otherwise process parallel edges when warranted.

Program 17.1 includes a function for implementations to return a graph's set of edges to a client, in an array. A graph is nothing more nor less than its set of edges, and we often need a way to retrieve a graph in this form, regardless of its internal representation. We might even consider an *array of edges* representation as the basis for an ADT implementation (see Exercise 17.15). That representation, however, does not provide the flexibility that we need to perform efficiently the basic graph-processing operations that we shall be studying.

In this book, we generally work with *static* graphs, which have a fixed number of vertices V and edges E. Generally, we build the graphs by executing E calls to `GRAPHinsertE`, then process them by calling some ADT function that takes a graph as argument and returns some information about that graph. *Dynamic* problems, where we intermix graph processing with edge and vertex insertion and removal, take us into the realm of *online algorithms* (also known as *dynamic algorithms*), which present a different set of challenges. For example, the union-find problem that we considered in Chapter 1 is an exam-

ple of an online algorithm, because we can get information about the connectivity of a graph as we insert edges. The ADT in *Program* 17.1 supports *insert edge* and *remove edge* operations, so clients are free to use them to make changes in graphs, but there may be performance penalties for certain sequences of operations. For example, union-find algorithms are effective for only those clients that do not use *remove edge*.

The ADT might also include a function that takes an array of edges as an argument for use in initializing the graph. We could easily implement this function by calling `GRAPHinsert` for each of the edges (see Exercise 17.13) or, depending on the graph representation, we might be able to craft a more efficient implementation.

We might also provide graph-traversal functions that call client-supplied functions for each edge or each vertex in the graph. For some simple problems, using the array returned by `GRAPHedges` might suffice. Most of our implementations, however, do more complicated traversals that reveal information about the graph's structure, while implementing functions that provide a higher level of abstraction to clients.

In Sections 17.3 through 17.5, we examine the primary classical graph representations and implementations of the ADT functions in *Program* 17.1. These implementations provide a basis for us to expand the interface to include the graph-processing tasks that are our focus for the next several chapters.

When we consider a new graph-processing problem, we extend the ADT as appropriate to encompass functions that implement algorithms of interest for solving the problem. Generally these tasks fall into one of two broad categories:

- Compute the value of some measure of the graph.
- Compute some subset of the edges of the graph.

Program 17.2 Example of a graph-processing client

This program takes `V` and `E` from standard input, generates a random graph with `V` vertices and `E` edges, prints the graph if it is small, and computes (and prints) the number of connected components. It uses the ADT functions GRAPHrand (see *Program* 17.8), `GRAPHshow` (see Exercise 17.16 and *Program* 17.5), and `GRAPHcc` (see *Program* 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));
  }
```

Examples of the former are the number of connected components and the length of the shortest `path` between two given vertices in the graph; examples of the latter are a spanning tree and the longest cycle containing a given vertex. Indeed, the terms that we defined in Section 17.1 immediately bring to mind a host of computational problems.

Program 17.2 is an example of a graph-processing client program. It uses the basic ADT of *Program* 17.1, augmented by a *generate random graph* ADT function that returns a random graph that contains a given number of vertices and edges (see Section 17.6), and a *connected components* ADT function that returns the number of connected components in a given graph (see Section 18.4). We use similar but more sophisticated clients to generate other types of graphs, to test algorithms, and to learn properties of graphs. The basic interface is amenable for use in any graph-processing application.

The first decision that we face in developing an ADT implementation is which graph representation to use. We have three basic requirements. First, we must be able to accommodate the types of graphs that we are likely to encounter in applications (and we also would prefer not to waste space). Second, we should be able to construct the requisite data structures efficiently. Third, we want to develop efficient algorithms to solve our graph-processing problems without being unduly hampered by any restrictions imposed by the representation. Such requirements are standard ones for any domain that we consider—we emphasize them again them here because, as we shall see, different representations give rise to huge performance differences for even the simplest of problems.

Most graph-processing applications can be handled reasonably with one of two straightforward classical representations that are only slightly more complicated than the array-of-edges representation: the *adjacency-matrix* or the *adjacency-lists* representation. These representations, which we consider in detail in Sections 17.3 and 17.4, are based on elementary data structures (indeed, we discussed them both in Chapters 3 and 5 as example applications of sequential and linked allocation). The choice between the two depends

primarily on whether the graph is dense or sparse, although, as usual, the nature of the operations to be performed also plays an important role in the decision on which to use.

Exercises

▷ **17.12** Write a program that builds a graph by reading edges (pairs of integers between 0 and $V - 1$) from standard input.

17.13 Write a representation-independent graph-initialization ADT function that, given an array of edges, returns a graph.

17.14 Write a representation-independent graph ADT function that uses `GRAPHedges` to print out all the edges in the graph, in the format used in this text (vertex numbers separated by a hyphen).

17.15 Provide an implementation of the ADT functions in *Program* 17.1 that uses an array of edges to represent the graph. Modify `GRAPHinit` to take the maximum number of edges allowed as its second argument, for use in allocating the edge array. Use a brute-force implementation of `GRAPHremoveE` that removes an edge `v-w` by scanning the array to find `v-w` or `w-v`, and then exchanges the edge found with the final one in the array. Disallow parallel edges by doing a similar scan in `GRAPHinsertE`.

Figure 17.7 Adjacency-Matrix Graph Representation *This matrix is another representation of the graph depicted in Figure 17.1. It has a 1 in row v and column w whenever there is an edge connecting vertex v and vertex w. The array is symmetric about the diagonal. For example, the sixth row (and the sixth column) says that vertex 6 is connected to vertices 0 and 4. For some applications, we will adopt the convention that each vertex is connected to itself, and assign 1s on the main diagonal. The large blocks of 0s in the upper right and lower left corners are artifacts of the way we assigned vertex numbers for this example, not characteristic of the graph (except that they do indicate the graph to be sparse).*

IPR2023-00124
KASPERSKY EX 1036 Page 42

	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

17.3 ADJACENCY-MATRIX REPRESENTATION

An *adjacency-matrix* representation of a graph is a V-by-V array of Boolean values, with the entry in row v and column w defined to be `1` if there is an edge connecting vertex v and vertex w in the graph, and to be `0` otherwise. *Program* 17.3 is an implementation of the graph ADT that uses a direct representation of this matrix. The implementation maintains a two-dimensional array of integers with the entry `a[v][w]` set to `1` if there is an edge connecting `v` and `w` in the graph, and set to `0` otherwise. In an undirected graph, each edge is actually represented by *two* entries: the edge `v-w` is represented by `1` values in *both* `a[v][w]` and `a[w][v]`, as is the edge `w-v`.

Program 17.3 Graph ADT implementation (adjacency matrix)

This implementation of the interface in *Program* 17.1 uses a two-dimensional array. An implementation of the function `MATRIXint`, which allocates memory for the array and initializes it, is given in *Program* 17.4. The rest of the code is straightfor-

ward: An edge `i-j` is present in the graph if and only if `a[i][j]` and `a[j][i]` are both `1`. Edges are inserted and removed in constant time, and duplicate edges are silently ignored. Initialization and extracting all edges each take time proportional to V^2.

```
#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;
  }
```

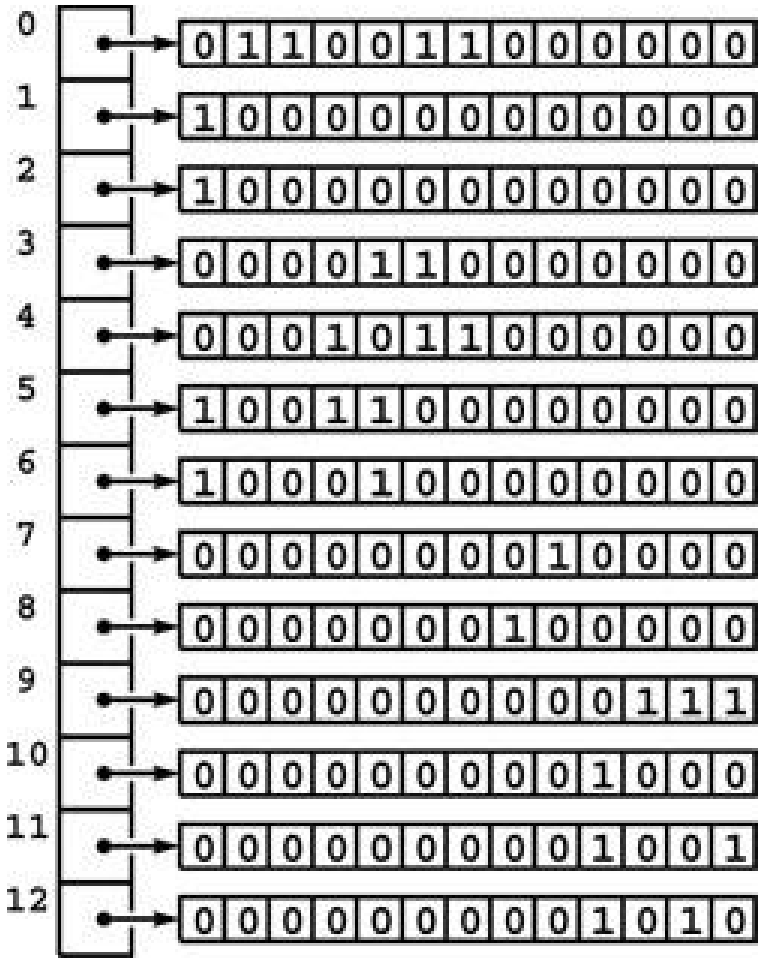
Program 17.4 Adjacency-matrix allocation and initialization

This program uses the standard C array-of-arrays representation for the two-dimensional adjacency matrix (see Section 3.7). It allocates `r` rows with c integers each, then initializes all entries to the value `val`. Thecall `MATRIXint (V, V, 0)` in *Pro-*

gram 17.3 takes time proportional to V^2 to create a matrix that represents a V-vertex graph with no edges. For small V, the cost of V calls to `malloc` might predominate.

```
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;
  }
```

Figure 17.8 Adjacency Matrix Data Structure *This figure depicts the C representation of the graph in Figure 17.1, as an array of arrays.*



As mentioned in Section 17.2, we generally assume that the number of vertices is known to the client when the graph is initialized. For many applications, we might set the number of vertices as a compile-time constant and use statically allocated arrays, but *Program* 17.3 takes the slightly more general approach of allocating dynamically the space for the adjacency matrix. *Program* 17.4 is an implementation of the standard method of dynamically allocating a two-dimensional array in C, as an array of pointers, as depicted in *Figure* 17.8. *Program* 17.4 also includes code that initializes the graph by setting the array entries all to a given value. This operation takes time proportional to V^2. Error checks for insufficient memory are not included in *Program* 17.4 for brevity—it is prudent programming practice to add them before using this code (see Exercise 17.22).

To add an edge, we set the two indicated array entries to `1`. We do not allow parallel edges in this representation: If an edge is to be inserted for which the array entries are already `1`, the code has no effect. In some ADT designs, it might be preferable to inform the client of the attempt to insert a parallel edge, perhaps using a return code from `GRAPHinsertE`. We do allow self-loops in this representation: An edge `v-v` is represented by a nonzero entry in `a[v][v]`.

To remove an edge, we set the two indicated array entries to `0`. If a nonexistent edge (one for which the array entries are already `0`) is to be removed, the code has no effect. Again, in some ADT designs, we might wish to arrange to inform the client of such conditions.

If we are processing huge graphs or huge numbers of small graphs, or space is otherwise tight, there are several ways to save space. For example, adjacency matrices that represent undirected graphs are symmetric: `a[v][w]` is always equal to `a[w][v]`. In C, it is easy to save space by storing only one-half of this symmetric matrix (see Exercise 17.20). At the extreme, we might consider using an array of bits (in this way, for instance, we could represent graphs of up to about 64,000 vertices in about 64 million 64-bit words) (see Exercise 17.21). These implementations have the slight complication that we need to add an ADT operation to test for the existence of an edge (see Exercise 17.19). (We do not use such an operation in our implementations because the code is slightly easier to understand when we test for the existence on an edge `v-w` by simply testing `a[v][w]`.) Such space-saving techniques are effective, but come at the cost of extra overhead that may fall in the inner loop in time-critical applications.

Many applications involve associating other information with each edge—in such cases, we can generalize the adjacency matrix to be an array that holds any information whatever. We reserve at least one value in the data type that we use for the array elements, to indicate that the indicated edge is absent. In Chapters 20 and 21, we explore the representation of such graphs.

Use of adjacency matrices depends on associating vertex names with integers between 0 and $V - 1$. This assignment might be done in one of many ways—for example, we consider a program that does so in Section 17.6). Therefore, the specific matrix of 0-1 values that we represent with a two-dimensional array in C is but one possible representation of any given graph as an adjacency matrix, because another program might give a different assignment of vertex names to the indices we use to specify rows and columns. Two arrays that appear to be markedly different could represent the same graph (see Exercise 17.17). This observation is a restatement of the graph isomorphism problem: Although we might like to determine whether or not two different arrays represent the same graph, no one has devised an algorithm that can always do so efficiently. This difficulty is fundamental. For example, our ability to find an efficient solution to various important graph-processing problems depends completely on the way in which the vertices are numbered (see, for example, Exercise 17.25).

Program 17.5 Graph ADT output (adjacency-lists format)

Printing out the full adjacency matrix is unwieldy for sparse graphs, so we might choose to simply print out, for each vertex, the vertices that are connected to that vertex by an edge.

```
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");
      }
  }
```

Figure 17.9 Adjacency Lists Format *This table illustrates yet another way to represent the graph in Figure 17.1: we associate each vertex with its set of adjacent vertices (those connected to it by a single edge). Each edge affects two sets: for every edge u-v in the graph, u appears in v's set and v appears in u's set.*

```
 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
```

Developing an ADT function that prints out the adjacency-matrix representation of a graph is a simple exercise (see Exercise 17.16). *Program* 17.5 illustrates a different implementation that may be preferred for sparse graphs: It just prints out the vertices adjacent to each vertex, as illustrated in *Figure* 17.9. These programs (and, specifically, their output) clearly illustrate a basic performance tradeoff. To print out the array, we need space for all V^2 entries; to print out the lists, we need room for just $V + E$ numbers. For sparse graphs, when V^2 is huge compared to $V + E$, we prefer the lists; for dense graphs, when E and V^2 are comparable, we prefer the array. As we shall soon see, we make the same basic tradeoff when we compare the adjacency-matrix representation with its primary alternative: an explicit representation of the lists.

The adjacency-matrix representation is not satisfactory for huge sparse graphs: The array requires V^2 bits of storage and V^2 steps just to initialize. In a dense graph, when the number of edges (the number of 1 bits in the matrix) is proportional to V^2, this cost may be acceptable, because time proportional to V^2 is required to process the edges no matter what representation we use. In a sparse graph, however, just initializing the array could be the dominant factor in the running time of an algorithm. Moreover, we may not even have enough space for the matrix. For example, we may be faced with graphs with millions of vertices and tens of millions of edges, but we may not want—or be able—to pay the price of reserving space for *trillions* of 0 entries in the adjacency matrix.

On the other hand, when we do need to process a huge dense graph, then the 0-entries that represent absent edges increase our space needs by only a constant factor and provide us with the ability to determine whether any particular edge is present with a single

array access. For example, disallowing parallel edges is automatic in an adjacency matrix but is costly in some other representations. If we do have space available to hold an adjacency matrix, and either V^2 is so small as to represent a negligible amount of time or we will be running a complex algorithm that requires more than V^2 steps to complete, the adjacency-matrix representation may be the method of choice, no matter how dense the graph.

Exercises

▷ **17.16** Give an implementation of `GRAPHshow` for inclusion in the adjacency-lists graph ADT implementation (*Program* 17.3) that prints out a two-dimensional array of 0s and 1s like the one illustrated in *Figure* 17.7.

▷ **17.17** Give the adjacency-matrix representations of the three graphs depicted in *Figure* 17.2.

17.18 Given a graph, consider another graph that is identical to the first, except that the names of (integers corresponding to) two vertices are interchanged. How do the adjacency matrices of these two graphs differ?

▷ **17.19** Add a function `GRAPHedge` to the graph ADT that allows clients to test whether there is an edge connecting two given vertices, and provide an implementation for the adjacency-matrix representation.

▷ **17.20** Modify *Program* 17.3, augmented as described in Exercise 17.19, to cut its space requirements about in half by not including array entries `a[v][w]` for `w` greater than `v`.

17.21 Modify *Program* 17.3, augmented as described in Exercise 17.19, to use an array of bits, rather than of integers. That is, if your computer has B bits per word, your implementation should be able to represent a graph with V vertices in about V^2/B words (as opposed to V^2). Do empirical tests to assess the effect of using a bit array on the time required for the ADT operations.

17.22 Modify *Program* 17.4 to check `malloc` return codes and return `0` if there is insufficient memory available to represent the matrix.

17.23 Write a version of *Program* 17.4 that uses a single call to `malloc`.

17.24 Add implementations of `GRAPHcopy` and `GRAPHdestroy` to *Program* 17.3.

○ **17.25** Suppose that all k vertices in a group have consecutive indices. How can you determine from the adjacency matrix whether or not that group of vertices constitutes a

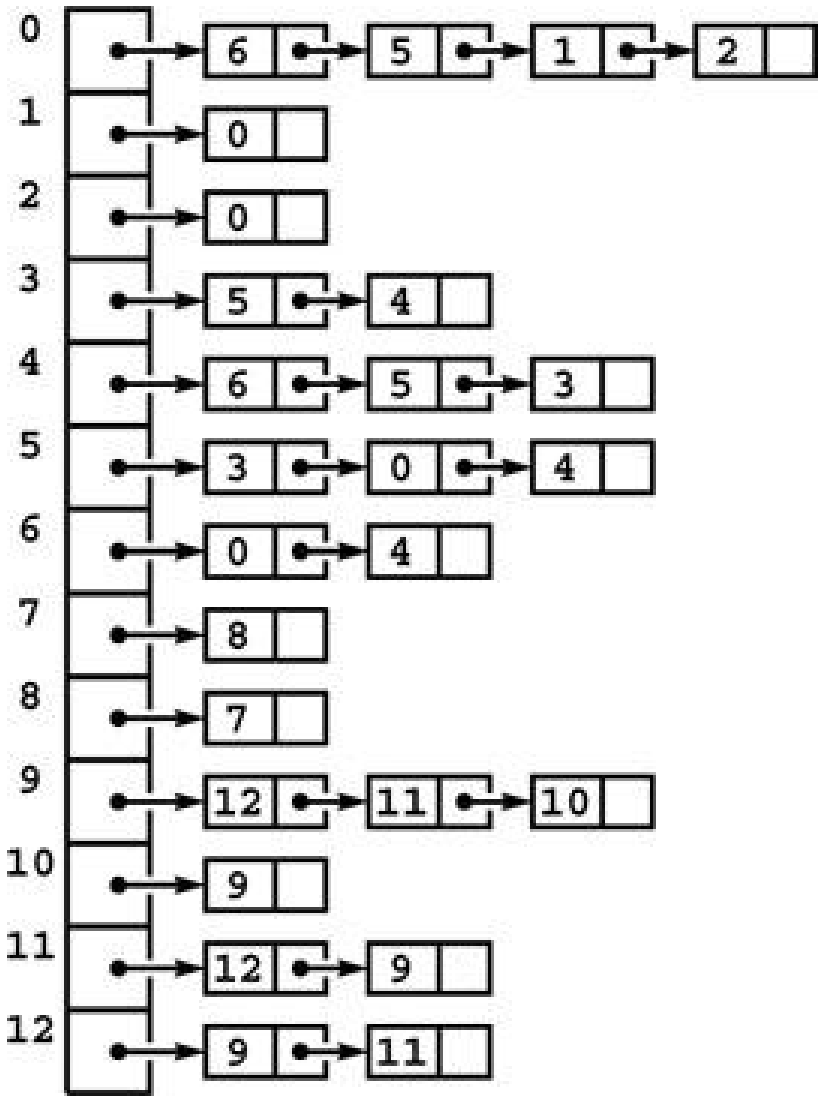
clique? Add a function to the adjacency-matrix implementation of the graph ADT (*Program* 17.3) that finds, in time proportional to V^2, the largest group of vertices with consecutive indices that constitutes a clique.

17.4 ADJACENCY-LISTS REPRESENTATION

The standard representation that is preferred for graphs that are not dense is called the *adjacency-lists* representation, where we keep track of all the vertices connected to each vertex on a linked list that is associated with that vertex. We maintain an array of lists so that, given a vertex, we can immediately access its list; we use linked lists so that we can add new edges in constant time.

Program 17.6 is an implementation of the ADT interface in *Program* 17.1 that is based on this approach, and *Figure* 17.10 depicts an example. To add an edge connecting `v` and `w` to this representation of the graph, we add `w` to v's adjacency list and `v` to w's adjacency list. In this way, we still can add new edges in constant time, but the total amount of space that we use is proportional to the number of vertices plus the number of edges (as opposed to the number of vertices squared, for the adjacency-matrix representation). We again represent each edge in two different places: an edge connecting `v` and `w` is represented as nodes on both adjacency lists. It is important to include both; otherwise, we could not answer efficiently simple questions such as, "Which vertices are connected directly to vertex v?"

Figure 17.10 Adjacency-Lists Data Structure *This figure depicts a representation of the graph in Figure 17.1 as an array of linked lists. The space used is proportional to the number of nodes plus the number of edges. To find the indices of the vertices connected to a given vertex v, we look at the v th position in an array, which contains a pointer to a linked list containing one node for each vertex connected to v. The order in which the nodes appear on the lists depends on the method that we use to construct the lists.*



Program 17.6 Graph ADT implementation (adjacency lists)

This implementation of the interface in *Program* 17.1 uses an array of lists, one corresponding to each vertex. An edge `v-w` is represented by a node for `w` on list `v` and a node for `v` on list `w`. As in *Program* 17.3, `GRAPHedges` puts just one of the two representations of each edge into the output array. Implementations of `GRAPHcopy`, `GRAPHdestroy`, and `GRAPHremoveE` are omitted. The `GRAPHinsertE` code keeps insertion time constant by not checking for duplicate edges.

IPR2023-00124
KASPERSKY EX 1036 Page 51

```
#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;
  }
```

By contrast to *Program* 17.3, *Program* 17.6 builds multigraphs, because it does not remove parallel edges. Checking for duplicate edges in the adjacency-lists structure would necessitate searching through the lists and could take time proportional to V. Similarly, *Program* 17.6 does not include an implementation of the *remove edge* operation. Again, adding such an implementation is an easy exercise (see Exercise 17.28), but each deletion might take time proportional to V, to search through the two lists for the nodes to remove. These costs make the basic adjacency-lists representation unsuitable for applications involving huge graphs where parallel edges cannot be tolerated, or applications involving heavy use of the *remove edge* operation. In Section 17.5, we discuss the use of elementary data-structure techniques to augment adjacency lists such that they support constant-time *remove edge* and *parallel-edge detection* operations.

If a graph's vertex names are not integers, then (as with adjacency matrices) two different

programs might associate vertex names with the integers from 0 to $V - 1$ in two different ways, leading to two different adjacency-list structures (see, for example, *Program* 17.10). We cannot expect to be able to tell whether two different structures represent the same graph because of the difficulty of the graph isomorphism problem.

Moreover, with adjacency lists, there are numerous representations of a given graph even for a given vertex numbering. No matter in what order the edges appear on the adjacency lists, the adjacency-list structure represents the same graph (see Exercise 17.31). This characteristic of adjacency lists is important to know because the order in which edges appear on the adjacency lists affects, in turn, the order in which edges are processed by algorithms. That is, the adjacency-list structure determines how our various algorithms see the graph. Although an algorithm should produce a correct answer no matter how the edges are ordered on the adjacency lists, it might get to that answer by different sequences of computations for different orderings. If an algorithm does not need to examine all the graph's edges, this effect might affect the time that it takes. And, if there is more than one correct answer, different input orderings might lead to different output results.

The primary advantage of the adjacency-lists representation over the adjacency-matrix representation is that it always uses space proportional to $E + V$, as opposed to V^2 in the adjacency matrix. The primary disadvantage is that testing for the existence of specific edges can take time proportional to V, as opposed to constant time in the adjacency matrix. These differences trace, essentially, to the difference between using linked lists and arrays to represent the set of vertices incident on each vertex.

Thus, we see again that an understanding of the basic properties of linked data structures and arrays is critical if we are to develop efficient graph ADT implementations. Our interest in these performance differences is that we want to avoid implementations that are inappropriately inefficient under unexpected circumstances when a wide range of operations is to be included in the ADT. In Section 17.5, we discuss the application of basic symbol-table algorithmic technology to realize many of the theoretical benefits of both structures. Nonetheless, because *Program* 17.6 is a simple implementation with the essential characteristics that we need to learn efficient algorithms for processing sparse graphs, we use it as the basis for many implementations in this book.

Exercises

▷ **17.26** Show, in the style of *Figure* 17.10, the adjacency-lists structure produced when you insert the edges in the graph

IPR2023-00124
KASPERSKY EX 1036 Page 53

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

(in that order) into an initially empty graph, using *Program* 17.6.

17.27 Give implementations of `GRAPHshow` that have the same functionality as Exercise 17.16 and *Program* 17.5, for inclusion in the adjacency-lists graph ADT implementation (*Program* 17.6).

17.28 Provide an implementation of the *remove edge* function `GRAPHremoveE` for the adjacency-lists graph ADT implementation (*Program* 17.6). *Note*: Remember the possibility of duplicates.

17.29 Add implementations of `GRAPHcopy` and `GRAPHdestroy` to the adjacency-lists graph ADT implementation (*Program* 17.6).

○ **17.30** Give a simple example of an adjacency-lists graph representation that could not have been built by repeated addition of edges by *Program* 17.6.

17.31 How many different adjacency-lists representations represent the same graph as the one depicted in *Figure* 17.10?

17.32 Write a version of *Program* 17.6 that keeps the adjacency lists in sorted order of vertex index. Describe a situation where this approach would be useful.

▷ **17.33** Add a function declaration to the graph ADT (*Program* 17.1) that removes self-loops and parallel edges. Provide the trivial implementation of this function for the adjacency-matrix–based ADT implementation (Pro-gram 17.3), and provide an implementation of the function for the adjacency-list–based ADT implementation (*Program* 17.6) that uses time proportional to E and extra space proportional to V.

17.34 Extend your solution to Exercise 17.33 to also remove degree-0 (isolated) vertices. *Note*: To remove vertices, you need to rename the other vertices and rebuild the data structures—you should do so just once.

17.35 Write an ADT function for the adjacency-lists representation (Pro-gram 17.6) that collapses paths that consist solely of degree-2 vertices. Specifically, every degree-2 vertex in a graph with no parallel edges appears on some `path` `u-...-w` where `u` and `w` are either equal or not of degree 2. Replace any such `path` with `u-w` and then remove all unused degree-2 vertices as in Exercise 17.34. *Note*: This operation may introduce self-loops and parallel edges, but it preserves the degrees of vertices that are not removed.

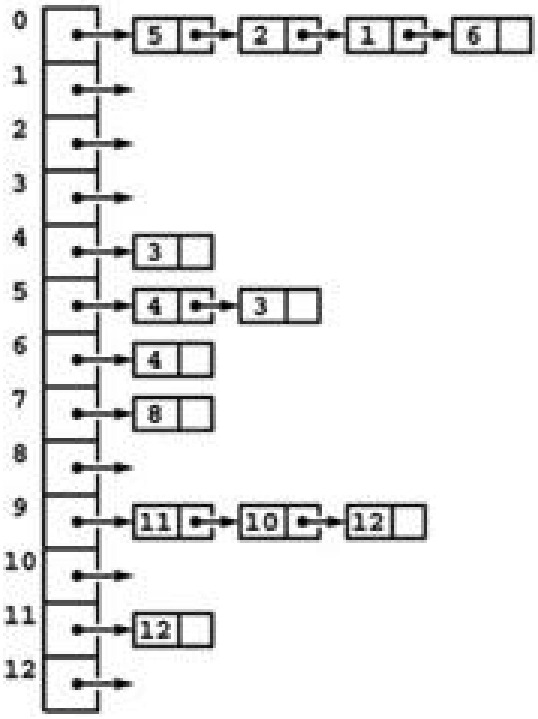
▷ **17.36** Give a (multi)graph that could result from applying the transformation described in Exercise 17.35 on the sample graph in *Figure* 17.1.

17.5 VARIATIONS, EXTENSIONS, AND COSTS

In this section, we describe a number of options for improving the graph representations discussed in Sections 17.3 and 17.5. The modifications fall into one of two categories. First, the basic adjacency-matrix and adjacency-lists mechanisms extend readily to allow us to represent other types of graphs. In the relevant chapters, we consider these extensions in detail, and give examples; here, we look at them briefly. Second, we often need to modify or augment the basic data structures to make certain operations more efficient. We do so as a matter of course in the chapters that follow; in this section, we discuss the application of data-structure design techniques to enable efficient implementation of several basic functions.

Figure 17.11 Digraph Representations *The adjacency-array and adjacency-lists representations of a digraph have only one representation of each edge, as illustrated in the adjacency array (top) and adjacency lists (bottom) representation of the set of edges in Figure 17.1 interpreted as a digraph (see Figure 17.6, top).*

	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



For *digraphs*, we represent each edge just once, as illustrated in *Figure* 17.11. An edge `v-w` in a digraph is represented by a `1` in the entry in row `v` and column `w` in the adjacency array or by the appearance of `w` on v's adjacency list in the adjacency-lists representation. These representations are simpler than the corresponding representations that we have been considering for undirected graphs, but the asymmetry makes digraphs more complicated combinatorial objects than undirected graphs, as we see in Chapter 19. For example, the standard adjacency-lists representation gives no direct way to find all edges coming into a vertex in a digraph, so we must make appropriate modifications if that operation needs to be supported.

For *weighted graphs* and *networks*, we fill the adjacency matrix with weights instead of Boolean values (using some nonexistent weight to represent the absence of an edge); in the adjacency-lists representation, we include a vertex weight in the adjacency structure or an edge weight in adjacency-list elements.

It is often necessary to associate still more information with the vertices or edges of a graph, to allow that graph to model more complicated objects. We can associate extra information with each edge by extending the `Edge` type in *Program* 17.1 as appropriate, then using instances of that type in the adjacency matrix, or in the list nodes in the adjacency lists. Or, since vertex names are integers between 0 and $V - 1$, we can use vertex-indexed arrays to associate extra information for vertices, perhaps using an appropriate ADT. Alternatively, we can simply use a separate symbol table ADT to associate extra information with each vertex and edge (see Exercise 17.46 and *Program* 17.10).

To handle various specialized graph-processing problems, we often need to add specialized auxiliary data structures to the graph ADT. The most common such data structure is a vertex-indexed array, as we saw already in Chapter 1, where we used vertex-indexed arrays to answer connectivity queries. We use vertex-indexed arrays in numerous implementations throughout the book.

As an example, suppose that we wish to know whether a vertex v in a graph is isolated. Is v of degree 0? For the adjacency-lists representation, we can find this information immediately, simply by checking whether `adj[v]` is null. But for the adjacency-matrix representation, we need to check all V entries in the row or column corresponding to `v` to know that each one is not connected to any other vertex; and for the array-of-edges representation, we have no better approach than to check all E edges to see whether there are any that involve v. Instead of these potentially time-consuming computations, we could implement a simple online algorithm that maintains a vertex-indexed array such that we can find the degree of any vertex in constant time (see Exercise 17.40). Such a substantial performance differential for such a simple problem is typical in graph processing.

Table 17.1 Worst-Case Cost of Graph-Processing Operations The performance characteristics of basic graph-processing ADT operations for different graph representations vary widely, even for simple tasks, as indicated in this table of the worst-case costs (all within a constant factor for large V and E). These costs are for the simple implementations we have described in previous sections; various modifications that affect the costs are described in the text of this section.

	array of edges	adjacency matrix	adjacency lists
space	E	V^2	$V + E$
initialize empty	1	V^2	V
copy	E	V^2	E
destroy	1	V	E
insert edge	1	1	1
find/remove edge	E	1	V
is v isolated?	E	V	1
path from u to v?	$E \lg^* V$	V^2	$V + E$

Table 17.1 shows the dependence of the cost of various simple graph-processing operations on the representation that we use. This table is worth examining before we consider the implementation of more complicated operations; it will help you to develop an intuition for the difficulty of various primitive operations. Most of the costs listed follow immediately from inspecting the code, with the exception of the bottom row, which we consider in detail at the end of this section.

In several cases, we can modify the representation to make simple operations more efficient, although we have to take care that doing so does not increase costs for other simple operations. For example, the entry for adjacency-matrix *destroy* is an artifact of the C array-of-arrays allocation scheme for two-dimensional arrays (see Section 3.7). It is not difficult to reduce this cost to be constant (see Exercise 17.23). On the other hand, if graph edges are sufficiently complex structures that the matrix entries are pointers, then to *destroy* an adjacency matrix would take time proportional to V^2.

Because of their frequent use in typical applications, we consider the *find edge* and *remove edge* operations in detail. In particular, we need a *find edge* operation to be able to remove or disallow parallel edges. As we saw in Section 17.3, these operations are trivial if we are using an adjacency-matrix representation—we need only to check or set an array entry that we can index directly. But how can we implement these operations efficiently in the adjacency-lists representation? One approach is described next, and another is described in Exercise 17.48. Both approaches are based on symbol-table implementations. If we use, for example, dynamic hash tables (see Section 14.5), both approaches take space proportional to E andallowustoperform either operation in constant time (on the average, amortized).

Specifically, to implement *find edge* when we are using adjacency lists, we could use an auxiliary symbol table for the edges. We can assign an edge `v-w` the integer key `v*V+w`

and use any of the symbol-table implementations from Part 4. (For undirected graphs, we might assign the same key to `v-w` and `w-v`.) We can insert each edge into the symbol table, after first checking whether it has already been inserted. If we wish, we can disallow parallel edges (see Exercise 17.47), maintain duplicate records in the symbol table for parallel edges, or build a graph ADT implementation based entirely on these operations (see Exercise 17.46). In the present context, our main interest in this technique is that it provides a constant-time *find edge* implementation for adjacency lists.

To remove edges, we can put a pointer in the symbol-table record for each edge that refers to its representation in the adjacency-lists structure. Even this information, however, is not sufficient to allow us to remove the edge in constant time unless the lists are doubly linked (see Section 3.4). Furthermore, in undirected graphs, it is not sufficient to remove the node from the adjacency list, because each edge appears on two different adjacency lists. One solution to this difficulty is to put both pointers in the symbol table; another is to link together the two list nodes that correspond to a particular edge (see Exercise 17.44). With either of these solutions, we can remove an edge in constant time.

Removing vertices is more expensive. In the adjacency-matrix representation, we essentially need to remove a row and a column from the array, which is not much less expensive than starting over again with a smaller array. If we are using an adjacency-lists representation, we see immediately that it is not sufficient to remove nodes from the vertex's adjacency list, because each node on the adjacency list specifies another vertex whose adjacency list we must search to remove the other node that represents the same edge. We need the extra links to support constant-time edge removal as described in the previous paragraph if we are to remove a vertex in time proportional to V.

We omit implementations of these operations here because they are straightforward programming exercises using basic techniques from Part 1, because maintaining complex structures with multiple pointers per node is not justified in typical applications that involve static graphs, and because we wish to avoid getting bogged down in the details of maintaining the other pointers when implementing graph-processing algorithms that do not otherwise use them. In Chapter 22, we do consider implementations of a similar structure that plays an essential role in the powerful general algorithms that we consider in that chapter.

For clarity in describing and developing implementations of algorithms of interest, we use the simplest appropriate representation. Generally, we strive to make a link or an auxiliary array in a piece of code directly relevant to the task at hand. Many programmers practice this kind of minimalism as a matter of course, knowing that maintaining the integrity of a data structure with multiple disparate components can be a challenging task, indeed.

IPR2023-00124
KASPERSKY EX 1036 Page 59

We might also modify the basic data structures in a performance-tuning process to save space or time, particularly when processing huge graphs (or huge numbers of small graphs). For example, we can dramatically improve the performance of algorithms that process huge static graphs represented with adjacency lists by stripping down the representation to use arrays of varying length instead of linked lists to represent the set of vertices incident on each vertex. With this technique, we can ultimately represent a graph with just $2E$ integers less than V and V integers less than V^2 (see Exercises 17.51 and 17.53). Such representations are attractive for processing huge static graphs.

Many of the algorithms that we consider adapt readily to all the variations that we have discussed in this section, because they are based on a few high-level abstract operations such as "perform the following operation for each edge adjacent to vertex v." Indeed, several of the programs that we consider differ *only* in the way that such abstract operations are implemented.

Why not develop these algorithms at a higher level of abstraction, then discuss different options for representing the data and implementing the associated operations, as we have done in so many instances throughout this book? The answer to this question is not clearcut. Often, we are presented with a clear choice between adjacency lists, for sparse graphs, and adjacency matrices, for dense graphs. We choose to work directly with one of these two particular representations in our primary implementations because the performance characteristics of the algorithms are clearly exposed in code that works with the low-level representation, and that code is generally no more difficult to read and understand than code written in terms of higher-level abstractions.

In some instances, our algorithm-design decisions depend on certain properties of the representation. Working at a higher level of abstraction might obscure our knowledge of that dependence. If we *know* that one representation would lead to poor performance but another would not, we would be taking an unnecessary risk were we to consider the algorithm at the wrong level of abstraction. As usual, our goal is to craft implementations such that we can make precise statements about performance.

It is possible to address these issues with a rigorous abstract approach, where we build layers of abstraction that culminate in the abstract operations that we need for our algorithms. Adding an ADT operation to test for the existence of an edge is one example (see Exercise 17.19), and we could also arrange to have representation-independent code for processing each of the vertices adjacent to a given vertex (see Exercise 17.60). Such an approach is worthwhile in many situations. In this book, however, we keep one eye on performance by using code that directly accesses adjacency matrices when working with dense graphs and adjacency lists when working with sparse graphs, augmenting the basic structures as appropriate to suit the task at hand.

IPR2023-00124
KASPERSKY EX 1036 Page 60

All of the operations that we have considered so far are simple, albeit necessary, data-processing functions; and the essential net result of the discussion in this section is that basic algorithms and data structures from Parts 1 through 3 are effective for handling them. As we develop more sophisticated graph-processing algorithms, we face more difficult challenges in finding the best implementations for specific practical problems. To illustrate this point, we consider the last row in Table 17.1, which gives the costs of determining whether there is a `path` connecting two given vertices.

In the worst case, the simple algorithm in Section 17.7 examines all E edges in the graph (as do several other methods that we consider in Chapter 18). The entries in the center and right column on the bottom row in Table 17.1 indicate, respectively, that the algorithm may examine all V^2 entries in an adjacency-matrix representation, and all V list heads and all E nodes on the lists in an adjacency-lists representation. These facts imply that the algorithm's running time is linear in the size of the graph representation, but they also exhibit two anomalies: The worst-case running time is *not* linear in the number of edges in the graph if we are using an adjacency-matrix representation for a sparse graph or either representation for an extremely sparse graph (one with a huge number of isolated vertices). To avoid repeatedly considering these anomalies, we assume throughout that *the size of the representation that we use is proportional to the number of edges in the graph*. This point is moot in the majority of applications because they involve huge sparse graphs and thus require an adjacency-lists representation.

The left column on the bottom row in Table 17.1 derives from the use of the union-find algorithms in Chapter 1 (see Section 1.3). This method is attractive because it only requires space proportional to V, but has the drawback that it cannot exhibit the path. This entry highlights the importance of completely and precisely specifying graph-processing problems.

Even after taking all of these factors into consideration, one of the most significant challenges that we face when developing practical graph-processing algorithms is assessing the extent to which the results of worst-case performance analyses, such as those in Table 17.1, overestimate performance on graphs that we encounter in practice. Most of the literature on graph algorithms describes performance in terms of such worst-case guarantees, and, while this information is helpful in identifying algorithms that can have unacceptably poor performance, it may not shed much light on which of several simple, direct programs may be most suitable for a given application. This situation is exacerbated by the difficulty of developing useful models of average-case performance for graph algorithms, leaving us with (perhaps unreliable) benchmark testing and (perhaps overly conservative) worst-case performance guarantees to work with. For example, the graph-search methods that we discuss in Chapter 18 are all effective linear-time algorithms for finding a `path`

IPR2023-00124
KASPERSKY EX 1036 Page 61

between two given vertices, but their performance characteristics differ markedly, depending both upon the graph being processed and its representation. When using graph-processing algorithms in practice, we constantly fight this disparity between the worst-case performance guarantees that we can prove and the actual performance characteristics that we can expect. This theme will recur throughout the book.

Exercises

17.37 Develop an adjacency-matrix representation for dense multigraphs, and provide an ADT implementation for *Program* 17.1 that uses it.

○ **17.38** Why not use a direct representation for graphs (a data structure that models the graph exactly, with vertices represented as allocated records and edge lists containing links to vertices instead of vertex names)?

17.39 Write a representation-independent ADT function that returns a pointer to a vertex-indexed array giving the degree of each vertex in the graph. *Hint*: Use `GRAPHedges`.

17.40 Modify the adjacency-matrix ADT implementation (*Program* 17.3) to include in the graph representation a vertex-indexed array that holds the degree of each vertex. Add an ADT function `GRAPHdeg` that returns the degree of a given vertex.

17.41 Do Exercise 17.40 for the adjacency-lists representation.

▷ **17.42** Give a row to add to Table 17.1 for the problem of determining the number of isolated vertices in a graph. Support your answer with function implementations for each of the three representations.

○ **17.43** Give a row to add to Table 17.1 for the problem of determining whether a given digraph has a vertex with indegree V and outdegree 0. Support your answer with function implementations for each of the three representations. *Note*: Your entry for the adjacency-matrix representation should be V.

17.44 Use doubly-linked adjacency lists with cross links as described in the text to implement a constant-time *remove edge* function `GRAPHremoveE` for the adjacency-lists graph ADT implementation (*Program* 17.6).

17.45 Add a *remove vertex* function `GRAPHremoveV` to the doubly-linked adjacency-lists graph ADT implementation described in the previous exercise.

○ **17.46** Modify your solution to Exercise 17.15 to use a dynamic hash table, as described in the text, such that *insert edge* and *remove edge* take constant amortized time.

17.47 Modify the adjacency-lists graph ADT implementation (*Program* 17.6) to use a symbol table to ignore duplicate edges, such that it is functionally equivalent to the adjacency-matrix graph ADT (*Program* 17.3). Use dynamic hashing for your symbol-table implementation so that your implementation uses space proportional to E and can insert and remove edges in constant time (on the average, amortized).

17.48 Develop a graph ADT implementation based on an array-of-symbol-tables representation (with one symbol table for each vertex, which contains its list of adjacent edges). Use dynamic hashing for your symbol-table implementation so that your implementation uses space proportional to E and can insert and remove edges in constant time (on the average, amortized).

17.49 Develop a graph ADT intended for static graphs, based upon a function `GRAPHconstruct` that takes an array of edges as an argument and builds a graph. Develop an implementation of `GRAPHconstruct` that uses `GRAPHinit` and `GRAPHinsert` from *Program* 17.1. (Such an implementation might be useful for performance comparisons with the implementations described in Exercises 17.51 through 17.54.)

17.50 Develop an implementation of `GRAPHinit` and `GRAPHconstruct` from *Program* 17.1 that uses the ADT described in the previous exercise. (Such an implementation might be useful for backwards compatibility with driver programs such as *Program* 17.2.)

17.51 Develop an implementation for the `GRAPHconstruct` function described in Exercise 17.49 that use a compact representation based on the following data structures:

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

A graph is a vertex count, an edge count, and an array of vertices. A vertex contains an edge count and an array with one vertex index corresponding to each adjacent edge. Implement `GRAPHshow` for this representation.

17.52 Add to your solution to Exercise 17.51 a function that eliminates self-loops and parallel edges, as in Exercise 17.33.

17.53 Develop an implementation for the static-graph ADT described in Exercise 17.49 that uses just two arrays to represent the graph: one array of E vertices, and another of V indices or pointers into the first array. Implement `GRAPHshow` for this representation.

17.54 Add to your solution to Exercise 17.53 a function that eliminates self-loops and

parallel edges, as in Exercise 17.33.

17.55 Develop a graph ADT interface that associates (x, y) coordinates with each vertex, so that you can work with graph drawings. Modify `GRAPHinit` as appropriate and add functions `GRAPHdrawV` and `GRAPHdrawE` to initialize, to draw a vertex, and to draw an edge, respectively.

17.56 Write a client program that uses your interface to produce drawings of edges that are being added to a small graph.

17.57 Develop an implementation of your interface from Exercise 17.55 that produces a PostScript program with drawings as output (see Section 4.3).

17.58 Find an appropriate graphics interface that allows you to develop an implementation of your interface from Exercise 17.55 that directly draws graphs in a window on your display.

17.59 Extend your solution to Exercises 17.55 and 17.58 to include functions to erase vertices and edges and to draw them in different styles, so that you can write client programs that provide dynamic graphical animations of graph algorithms in operation.

▷ **17.60** Define a graph ADT function that will allow the clients to visit (run a client-supplied function that takes an edge as argument for) all edges adjacent to a given vertex. Provide implementations of the function for both the adjacency-matrix and adjacency-lists representations. To test your function, use it to implement a representation-independent version of `GRAPHshow`.

17.6 GRAPH GENERATORS

To develop further appreciation for the diverse nature of graphs as combinatorial structures, we now consider detailed examples of the types of graphs that we use later to test the algorithms that we study. Some of these examples are drawn from applications. Others are drawn from mathematical models that are intended both to have properties that we might find in real graphs and to expand the range of input trials available for testing our algorithms.

To make the examples concrete, we present them as extensions to the interface of *Program* 17.1, so that we can put them to immediate use when we test implementations of the graph algorithms that we consider. In addition, we consider a program that reads a sequence of pairs of arbitrary names from standard input and builds a graph with vertices corresponding to the names and edges corresponding to the pairs.

The implementations that we consider in this section are based upon the interface of *Program* 17.1, so they function properly, in theory, for any graph representation. In practice, however, some combinations of interface and representation can have unacceptably poor performance, as we shall see.

As usual, we are interested in having "random problem instances," both to exercise our programs with arbitrary inputs and to get an idea of how the programs might perform in real applications.

Program 17.7 Random graph generator (random edges)

This ADT function builds a graph by generating `E` random pairs of integers between `0` and `V-1`, interpreting the integers as vertex labels and the pairs of vertex labels as edges. It leaves the decision about the treatment of parallel edges and self-loops to the implementation of `GRAPHinsertE` and assumes that the ADT implementation maintains counts of the number of graph edges and vertices in `G->E` and `G->V`, repectively. This method is generally not suitable for generating huge dense graphs because of the number of parallel edges that it generates.

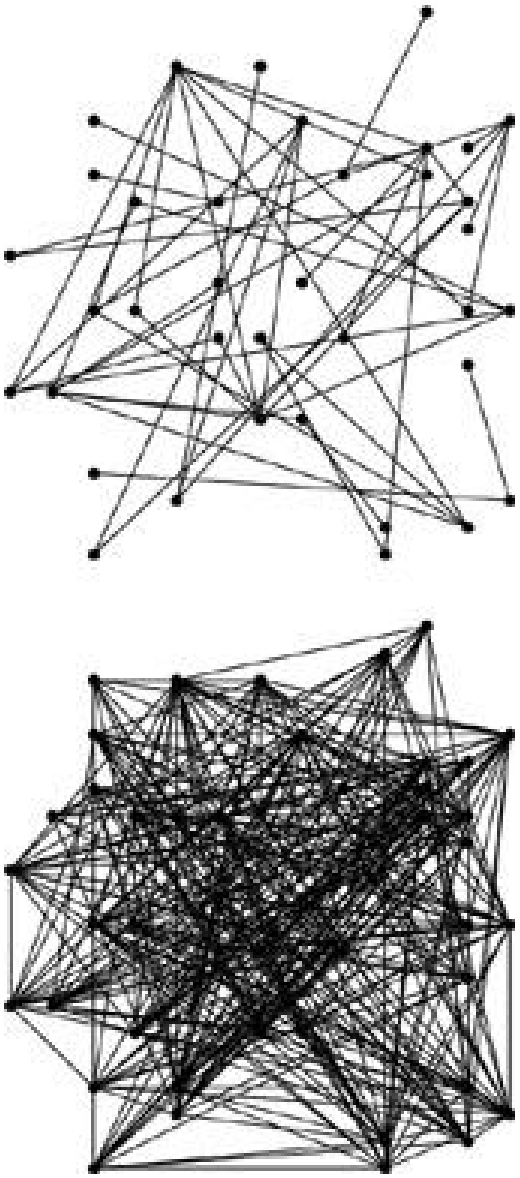
```
int randV (Graph G)
  { return G->V * (rand () / (RAND_MAX + 1.0)); }
Graph GRAPHrand (int V, int E)
  { Graph G = GRAPHinit (V);
    while (G->E < E)
      GRAPHinsertE (G, EDGE (randV (G), randV (G)));
    return G;
  }
```

For graphs, the latter goal is more elusive than for other domains that we have considered, although it is still a worthwhile objective. We shall encounter various different models of randomness, starting with these two:

Random edges This model is simple to implement, as indicated by the generator given in *Program* 17.7. For a given number of vertices V, we generate random edges by generating pairs of numbers between 0 and $V - 1$. The result is likely to be a random multigraph

with self-loops, rather than a graph as defined in Definition 17.1. A given pair could have two identical numbers (hence, self-loops could occur); and any pair could be repeated multiple times (hence, parallel edges could occur). *Program* 17.7 generates edges until the graph is known to have E edges, and leaves to the implementation the decision of whether to eliminate parallel edges. If parallel edges are eliminated, the number of edges generated is substantially higher than then number of edges used (E) for dense graphs (see Exercise 17.62); so this method is normally used for sparse graphs.

Figure 17.12 Two Random Graphs *Both of these random graphs have 50 vertices. The sparse graph at the top has 50 edges, while the dense graph at the bottom has 500 edges. The sparse graph is not connected, with each vertex connected only to a few others; the dense graph is certainly connected, with each vertex connected to 20 others, on the average. These diagrams also indicate the difficulty of developing algorithms that can draw arbitrary graphs (the vertices here are placed in random position).*



Random graph The classic mathematical model for random graphs is to consider all possible edges and to include each in the graph with a fixed probability p. If we want the expected number of edges in the graph to be E, we can choose $p = 2E/V(V-1)$. *Program* 17.8 is a function that uses this model to generate random graphs. This model precludes duplicate edges, but the number of edges in the graph is only equal to E *on the average*. This implementation is well-suited for dense graphs, but not for sparse graphs, since it runs in time proportional to $V(V-1)/2$ to generate just $E = pV(V-1)/2$ edges. That is, for sparse graphs, the running time of *Program* 17.8 is quadratic in the size of the graph (see Exercise 17.67).

Program 17.8 Random graph generator (random graph)

Like *Program* 17.7, this graph client generates random pairs of integers between 0 and V-1 to create a random graph, but it uses a different probabilistic model where each possible edge occurs independently with some probability p. The value of p is calculated such that the expected number of edges ($pV(V-1)/2$) is equal to E. The number of edges in any particular graph generated by this code will be close to E but is unlikely to be precisely equal to E. This method is primarily suitable for dense graphs, because its running time is proportional to V^2.

```
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;
  }
```

These models are well-studied and are not difficult to implement, but they do not necessarily generate graphs with properties similar to the ones that we see in practice. In particular, graphs that model maps, circuits, schedules, transactions, networks, and other practical situations usually not only are sparse, but also exhibit a *locality* property—edges are much more likely to connect a given vertex to vertices in a particular set than to ver-

tices that are not in the set. We might consider many different ways of modeling locality, as illustrated in the following examples.

***k*-neighbor graph** The graph depicted at the top in *Figure* 17.13 is drawn from a simple modification to a random-edges graph generator, where we randomly pick the first vertex v, then randomly pick the second from among those whose indices are within a fixed constant *k* of v (wrapping around from $V - 1$ to 0, when the vertices are arranged in a circle as depicted). Such graphs are easy to generate and certainly exhibit locality not found in random graphs.

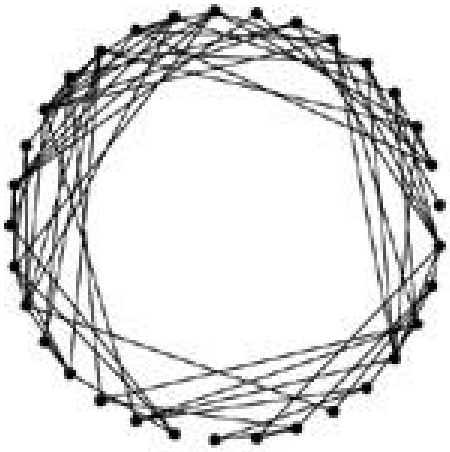
Euclidean neighbor graph The graph depicted at the bottom in *Figure* 17.13 is drawn from a generator that generates *V* points in the plane with random coordinates between 0 and 1, and then generates edges connecting any two points within distance *d* of one another. If *d* is small, the graph is sparse; if *d* is large, the graph is dense (see Exercise 17.73). This graph models the types of graphs that we might expect when we process graphs from maps, circuits, or other applications where vertices are associated with geometric locations. They are easy to visualize, exhibit properties of algorithms in an intuitive manner, and exhibit many of the structural properties that we find in such applications.

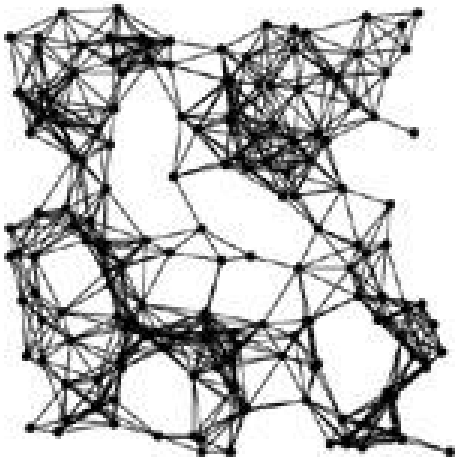
One possible defect in this model is that the graphs are not likely to be connected when they are sparse; other difficulties are that the graphs are unlikely to have high-degree vertices and that they do not have any long edges. We can change the models to handle such situations, if desired, or we can consider numerous similar examples to try to model other situations (see, for example, Exercises 17.71 and 17.72).

Or, we can test our algorithms on real graphs. In many applications, there is no shortage of problem instances drawn from actual data that we can use to test our algorithms. For example, huge graphs drawn from actual geographic data are easy to find; two more examples are listed in the next two paragraphs. The advantage of working with real data instead of a random graph model is that we can see solutions to real problems as algorithms evolve. The disadvantage is that we may lose the benefit of being able to predict the performance of our algorithms through mathematical analysis. We return to this topic when we are ready to compare several algorithms for the same task, at the end of Chapter 18.

Figure 17.13 Random Neighbor Graphs *These figures illustrate two models of sparse graphs. The neighbor graph at the top has 33 vertices and 99 edges, with each edge restricted to connect vertices whose indices differ by less than 10 (modulo V). The Euclidean neighbor graph at the bottom models the types of graphs that we might find in applications where vertices are tied to geometric locations. Vertices are random points in the plane; edges connect any pair of vertices within a specified distance d of each other. This graph is sparse (177 vertices and 1001 edges); by adjusting d, we can generate graphs of any desired density.*

IPR2023-00124
KASPERSKY EX 1036 Page 68





Transaction graph *Figure* 17.14 illustrates a tiny piece of a graph that we might find in a telephone company's computers. It has a vertex defined for each phone number, and an edge for each pair i and j with the property that i made a telephone call to j within some fixed period. This set of edges represents a huge multigraph. It is certainly sparse, since each person places calls to only a tiny fraction of the available telephones. It is representative of many other applications. For example, a financial institution's credit card and merchant account records might have similar information.

Function call graph We can associate a graph with any computer program with functions as vertices and an edge connecting X and Y whenever function X calls function Y. We can instrument the program to create such a graph (or have a compiler do it). Two completely different graphs are of interest: the static version, where we create edges at compile time corresponding to the function calls that appear in the program text of each function; and a dynamic version, where we create edges at run time when the calls actually happen. We use static function call graphs to study program structure and dynamic ones to study program behavior. These graphs are typically huge and sparse.

In applications such as these, we face massive amounts of data, so we might prefer to

study the performance of algorithms on real sample data rather than on random models. We might choose to try to avoid degenerate situations by randomly ordering the edges, or by introducing randomness in the decision making in our algorithms, but that is a different matter from generating a random graph. Indeed, in many applications, learning the properties of the graph structure is a goal in itself.

Figure 17.14 Transaction Graph *A sequence of pairs of numbers like this one might represent a list of telephone calls in a local exchange, or financial transfers between accounts, or any similar situation involving transactions between entities with unique identifiers. The graphs are hardly random—some phones are far more heavily used than others and some accounts are far more active than others.*

```
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
```

IPR2023-00124
KASPERSKY EX 1036 Page 70

In several of these examples, vertices are natural named objects, and edges appear as pairs of named objects. For example, a transaction graph might be built from a sequence of pairs of telephone numbers, and a Euclidean graph might be built from a sequence of pairs of cities or towns. *Program* 17.9 is a client program that we can use to build a graph in this common situation. For the client's convenience, it takes the set of edges as defining the graph, and deduces the set of vertex names from their use in edges. Specifically, the program reads a sequence of pairs of symbols from standard input, uses a symbol table to associate the vertex numbers 0 to $V - 1$ to the symbols (where V is the number of different symbols in the input), and builds a graph by inserting the edges, as in Programs 17.7 and 17.8. We could adapt any symbol-table implementation to support the needs of *Program* 17.9; *Program* 17.10 is an example that uses ternary search trees (TSTs) (see Chapter 14). These programs make it easy for us to test our algorithms on real graphs that may not be characterized accurately by any probabilistic model.

Program 17.9 Building a graph from pairs of symbols

This function uses a symbol table to build a graph by reading pairs of symbols from standard input. The symbol-table ADT function `STindex` associates an integer with each symbol: on unsuccessful search in a table of size `N` it adds the symbol to the table with associated integer `N+1`; on successful search, it simply returns the integer previously associated with the symbol. Any of the symbol-table methods in Part 4 can be adapted for this use; for example, see *Program* 17.10. The code to check that the number of edges does not exceed `Emax` is omitted.

```
#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;
  }
```

Program 17.9 is also significant because it validates the assumption we have made in all of our algorithms that the vertex names are integers between 0 and $V - 1$. If we have a graph that has some other set of vertex names, then the first step in representing a graph is to use a program such as *Program* 17.9 to map the vertex names to integers between 0 and $V - 1$.

Some graphs are based on implicit connections among items. We do not focus on such graphs, but we note their existence in the next few examples and devote a few exercises to them. When faced with processing such a graph, we can certainly write a program to construct explicit graphs by enumerating all the edges; but there also may be solutions to specific problems that do not require that we enumerate all the edges and therefore can run in sublinear time.

Program 17.10 Symbol indexing for vertex names

This implementation of the symbol-table indexing function for string keys that is described in the commentary for *Program* 17.9 accomplishes the task by adding an index field to each node in an existence-table TST (see *Program* 15.8). The index associated with each key is kept in the index field in the node corresponding to its end-of-string character. When we reach the end of a search key, we set its index if necessary and set a global variable which is returned to the caller after all recursive calls to the function have returned.

```
#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; }
```

Degrees-of-separation graph Consider a collection of subsets drawn from V items. We define a graph with one vertex corresponding to each element in the union of the subsets and edges between two vertices if both vertices appear in some subset (see *Figure* 17.15). If desired, the graph might be a multigraph, with edge labels naming the appropriate subsets. All items incident on a given item v are said to be 1 *degree of separation* from v. Otherwise, all items incident on any item that is i degrees of separation from v (that are not already known to be i or fewer degrees of separation from v) are (i+1) degrees of separation from v. This construction has amused people ranging from mathematicians (Erdös number) to movie buffs (separation from Kevin Bacon).

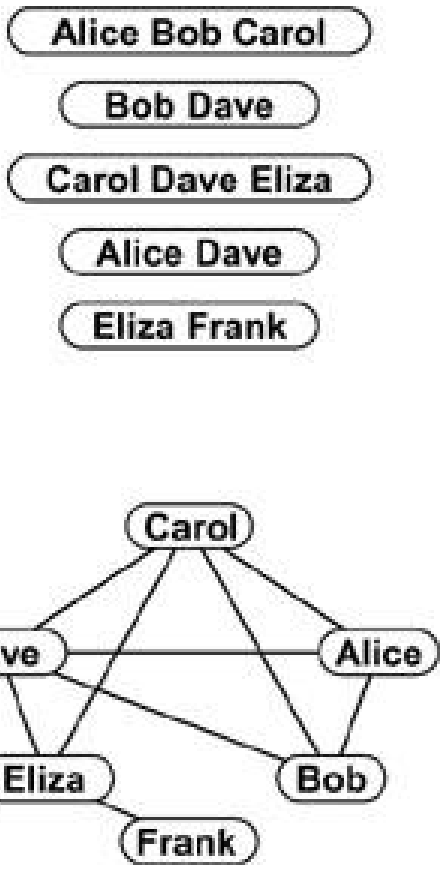
Interval graph Consider a collection of V intervals on the real line (pairs of real numbers). We define a graph with one vertex corresponding to each interval, with edges between vertices if the corresponding intervals intersect (have any points in common).

de Bruijn graph Suppose that V is a power of 2. We define a digraph with one vertex corresponding to each nonnegative integer less than V, with edges from each vertex i to 2i and (2i +1) mod lg V. These graphs are useful in the study of the sequence of values that can occur in a fixed-length shift register for a sequence of operations where we repeatedly shift all the bits one position to the left, throw away the leftmost bit, and fill the rightmost bit with 0 or 1. *Figure* 17.16 depicts the de Bruijn graphs with 8, 16, 32, and 64 vertices.

The various types of graphs that we have considered in this section have a wide variety of different characteristics. However, they all look the same to our programs: They are simply collections of edges. As we saw in Chapter 1, learning even the simplest facts about them can be a computational challenge. In this book, we consider numerous ingenious algorithms that have been developed for solving practical problems related to many types of graphs.

IPR2023-00124
KASPERSKY EX 1036 Page 73

Figure 17.15 Degrees-of-Separation Graph *The graph at the bottom is defined by the groups at the top, with one vertex for each person and an edge connecting a pair of people whenever they are in the same group. Shortest* `path` *lengths in the graph correspond to degrees of separation. For example, Frank is three degrees of separation from Alice and Bob.*



Based just on the few examples presented in this section, we can see that graphs are complex combinatorial objects, far more complex than those underlying other algorithms that we studied in Parts 1 through 4. In many instances, the graphs that we need to consider in applications are difficult or impossible to characterize. Algorithms that perform well on random graphs are often of limited applicability because it is often difficult to be persuaded that random graphs have structural characteristics the same as those of the graphs that arise in applications. The usual approach to overcome this objection is to design algorithms that perform well in the worst case. While this approach is successful in some instances, it falls short (by being too conservative) in others.

While we are often not justified in assuming that performance studies on graphs generated from one of the random graph models that we have discussed will give information sufficiently accurate to allow us to predict performance on real graphs, the graph generators that we have considered in this section are useful in helping us to test implementations and to understand our algorithms' performance. Before we even attempt to predict performance for applications, we must at least verify any assumptions that we might have

IPR2023-00124
KASPERSKY EX 1036 Page 74

made about the relationship between the application's data and whatever models or sample data we may have used. While such verfication is wise when we are working in any applications domain, it is particularly important when we are processing graphs, because of the broad variety of types of graphs that we encounter.

Exercises

▷ **17.61** When we use *Program* 17.7 to generate random graphs of density αV, what fraction of edges produced are self-loops?

17.62 Calculate the expected number of parallel edges produced when we use *Program* 17.7 to generate random graphs with V vertices of density. Use the result of your calculation to draw plots showing the fraction of parallel edges produced as a function of, for V = 10, 100, and 1000.

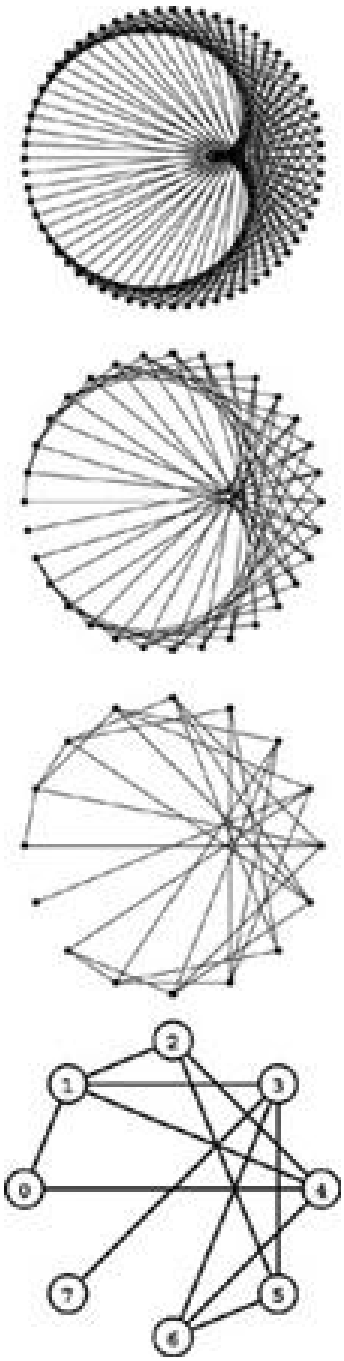
17.63 Find a large undirected graph somewhere online—perhaps based on network-connectivity information, or a separation graph defined by coauthors in a set of bibliographic lists or by actors in movies.

▷ **17.64** Write a client program that generates sparse random graphs for a well-chosen set of values of V and E, and prints the amount of space that it used for the graph representation and the amount of time that it took to build it. Test your program with a sparse-graph ADT implementation (*Program* 17.6) and with the random-graph generator (*Program* 17.7), so that you can do meaningful empirical tests on graphs drawn from this model.

▷ **17.65** Write a client program that generates dense random graphs for a well-chosen set of values of V and E, and prints the amount of space that it used for the graph representation and the amount of time that it took to build it.

Figure 17.16 De Bruijn Graphs *A de Bruijn digraph of order n has 2^n vertices with edges from i to 2i mod n and (2i +1) mod n, for all i. Pictured here are the underlying undirected de Bruijn graphs of order 6, 5, 4, and 3 (top to bottom).*

IPR2023-00124
KASPERSKY EX 1036 Page 75



Test your program with a dense-graph ADT implementation (*Program* 17.3) and with the random-graph generator (*Program* 17.8), so that you can do meaningful empirical tests on graphs drawn from this model.

● **17.66** Give the standard deviation of the number of edges produced by *Program* 17.8.

17.67 Write a program that produces each possible graph with precisely the same probability as does *Program* 17.8, but uses time and space proportional to only $V + E$, not V^2. Test your program as described in Exercise 17.64.

○ **17.68** Write a program that produces each possible graph with precisely the same probability as does *Program* 17.7, but uses time proportional to E, even when the density is close to 1. Test your program as described in Exercise 17.65.

17.69 Write a program that produces, with equal likelihood, each of the possible graphs with V vertices and E edges (see Exercise 17.9). Test your program as described in Exercise 17.64 (for low densities) and as described in Exercise 17.65 (for high densities).

○ **17.70** Write a program that generates random graphs by connecting vertices arranged in a $\sqrt{V}$-by-$\sqrt{V}$ grid to their neighbors (see **Figure** 1.2), with k extra edges connecting each vertex to a randomly chosen destination vertex (each destination vertex equally likely). Determine how to set k such that the expected number of edges is E. Test your program as described in Exercise 17.64.

○ **17.71** Write a program that generates random digraphs by randomly connecting vertices arranged in a $\sqrt{V}$-by-$\sqrt{V}$ grid to their neighbors, with each of the possible edges occurring with probability p (see **Figure** 1.2). Determine how to set p such that the expected number of edges is E. Test your program as described in Exercise 17.64.

○ **17.72** Augment your program from Exercise 17.71 to add R extra random edges, computed as in *Program* 17.7. For large R, shrink the grid so that the total number of edges remains about V.

17.73 Write a program that generates V random points in the plane, then builds a graph consisting of edges connecting all pairs of points within a given distance d of one another (see *Figure* 17.13 and Program 3.20). Determine how to set d such that the expected number of edges is E. Test your program as described in Exercise 17.64 (for low densities) and as described in Exercise 17.65 (for high densities).

17.74 Write a program that generates V random intervals in the unit interval, all of length d, then builds the corresponding interval graph. Determine how to set d such that the expected number of edges is E. Test your program as described in Exercise 17.64 (for low densities) and as described in Exercise 17.65 (for high densities). *Hint*: Use a BST.

17.75 Write a program that chooses V vertices and E edges from the real graph that you found for Exercise 17.63. Test your program as described in Exercise 17.64 (for low densities) and as described in Exercise 17.65 (for high densities).

○ **17.76** One way to define a transportation system is with a set of sequences of vertices, each sequence defining a `path` connecting the vertices. For example, the sequence `0-9-3-2` defines the edges `0-9, 9-3`, and `3-2`. Write a program that builds a graph from

IPR2023-00124
KASPERSKY EX 1036 Page 77

an input file consisting of one sequence per line, using symbolic names. Develop input suitable to allow you to use your program to build a graph corresponding to the Paris metro system.

17.77 Extend your solution to Exercise 17.76 to include vertex coordinates, along the lines of Exercise 17.59, so that you can work with graphical representations.

○ **17.78** Apply the transformations described in Exercises 17.33 through 17.35 to various graphs (see Exercises 17.63–76), and tabulate the number of vertices and edges removed by each transformation.

○ **17.79** Design an appropriate extension that allows you to use *Program* 17.1 to build separation graphs without having to call a function for each implied edge. That is, the number of function calls required to build a graph should be proportional to the sum of the sizes of the groups. (Implementations of graph-processing functions are left with the problem of efficiently handling implied edges.)

17.80 Give a tight upper bound on the number of edges in any separation graph with N different groups of k people.

▷ **17.81** Draw graphs in the style of *Figure* 17.16 that, for $V = 8$, 16, and 32, have V vertices numbered from 0 to $V - 1$ and an edge connecting each vertex i with $[i/2]$.

17.82 Modify the ADT interface in *Program* 17.1 to allow clients to use symbolic vertex names and edges to be pairs of instances of a generic Vertex type. Hide the vertex-index representation and the symbol-table ADT usage completely from clients.

17.83 Add a function to the ADT interface from Exercise 17.82 that supports a *join* operation for graphs, and provide implementations for the adjacency-matrix and adjacency-lists representations. *Note*: Any vertex or edge in either graph should be in the *join*, but vertices that are in both graphs appear only once in the *join*, and you should remove parallel edges.

17.7 SIMPLE, EULER, AND HAMILTON PATHS

Our first nontrivial graph-processing algorithms solve fundamental problems concerning paths in graphs. They introduce the general recursive paradigm that we use throughout the book, and they illustrate that apparently similar graph-processing problems can range widely in difficulty.

IPR2023-00124
KASPERSKY EX 1036 Page 78

These problems take us from local properties such as the existence of edges or the degrees of vertices to global properties that tell us about a graph's structure. The most basic such property is whether two vertices are connected.

Simple path Given two vertices, is there a simple `path` in the graph that connects them? In some applications, we might be satisfied to know of the *existence* of the path; in other applications, we might need an algorithm that can supply a specific path.

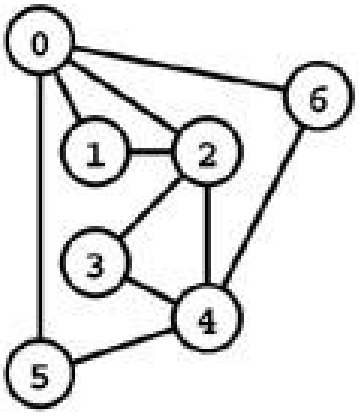
Program 17.11 is a direct solution that finds a path. It is based on *depth-first search*, a fundamental graph-processing paradigm that we considered briefly in Chapters 3 and 5 and shall study in detail in Chapter 18. The algorithm is based on a recursive function that determines whether there is a simple `path` from `v` to `w` by checking, for each edge `v-t` incident on `v`, whether there is a simple `path` from `t` to `w` that does not go through `v`. It uses a vertex-indexed array to mark `v` so that no `path` through `v` will be checked in the recursive call.

The code in *Program* 17.11 simply tests for the existence of a path. How can we augment it to print the path's edges? Thinking recursively suggests an easy solution:

- Add a statement to print `t-v` just after the recursive call in `pathR` finds a `path` from `t` to `w`.

- Switch `w` and `v` in the call on `pathR` in `GRAPHpath`.

Figure 17.17 Trace for Simple `Path` Search *This trace shows the operation of the recursive function in Program 17.11 for the call* `pathR (G, 2, 6)` *to find a simple* `path` *from 2 to 6 in the graph shown at the top. There is a line in the trace for each edge considered, indented one level for each recursive call. To check 2-0, we call* `pathR (G, 0, 6)`*. This call causes us to check 0-1, 0-2, and 0-5. To check 0-1, we call* `pathR (G, 1, 6)`*, which causes us to check 1-0 and 1-2, which do not lead to recursive calls because 0 and 2 are marked. For this example, the function discovers the* `path` *2-0-5-4-6.*




```
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)
```

Alone, the first change would cause the `path` from `v` to `w` to be printed in reverse order: If the call to `pathR (G, t, w)` finds a `path` from `t` to `w` (and prints that path's edges in reverse order), then printing `t-v` completes the job for the `path` from `v` to `w`. The second change reverses the order: To print the edges on the `path` from `v` to `w`, we print the edges on the `path` from `w` to `v` in reverse order. (This trick does not work for digraphs.) We could use this same strategy to implement an ADT function that calls a client-supplied function for each of the path's edges (see Exercise 17.87).

Figure 17.17 gives an example of the dynamics of the recursion. As with any recursive program (indeed, any program with function calls at all), such a trace is easy to produce: To modify *Program* 17.11 to produce one, we can add a variable `depth` that is incremented on entry and decremented on exit to keep track of the depth of the recursion, then add

code at the beginning of the recursive function to print out `depth` spaces followed by the appropriate information (see Exercises 17.85 and 17.86).

Program 17.11 Simple `path` search (adjacency matrix)

The function `GRAPHpath` tests for the existence of a simple `path` connecting two given vertices. It uses a recursive depth-first search function `pathR`, which, given two vertices `v` and `w`, checks each edge `v-t` adjacent to `v` to see whether it could be the first edge on a `path` to `w`. The vertex-indexed array `visited` keeps the function from revisiting any vertex, and therefore keeps paths simple.

To have the function print the edges of the `path` (in reverse order), add the statement `printf ("%d-%d", t, v);` just before the `return 1` near the end of `pathR` (*see text*).

```
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);
  }
```

Program 17.11 is centered around checking all edges adjacent to a given vertex. As we have noted, this operation is also easy to implement if we use the adjacency-lists representation for sparse graphs (see Exercise 17.89) or any of the variations that we have dis-

cussed.

Property 17.2 *We can find a* `path` *connecting two given vertices in a graph in linear time.*

The recursive depth-first search function in *Program* 17.11 immediately implies a proof by induction that the ADT function determines whether or not a `path` exists. Such a proof is easily extended to establish that, in the worst case, *Program* 17.11 checks all the entries in the adjacency matrix exactly once. Similarly, we can show that the analogous program for adjacency lists checks all of the graph edges exactly twice (once in each direction), in the worst case.■

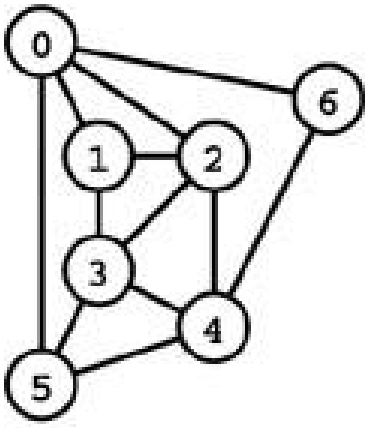
We use the phrase *linear* in the context of graph algorithms to mean a quantity whose value is within a constant factor of $V + E$, the size of the graph. As discussed at the end of Section 17.5, such a value is also normally within a constant factor of the size of the graph representation. Property 17.2 is worded so as to allow for the use of the adjacency-lists representation for sparse graphs and the adjacency-matrix representation for dense graphs, our general practice. It is not appropriate to use the term "linear" to describe an algorithm that uses an adjacency matrix and runs in time proportional to V^2 (even though it is linear in the size of the graph representation) *unless* the graph is dense. Indeed, if we use the adjacency-matrix representation for a sparse graph, we cannot have a linear-time algorithm for *any* graph-processing problem that could require examination of all the edges.

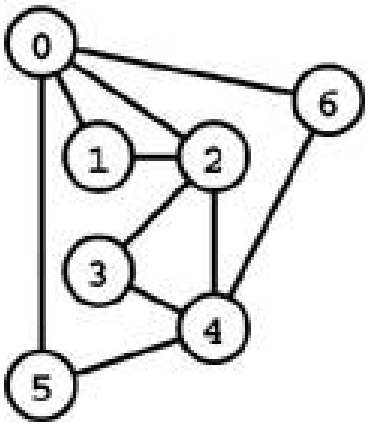
We study depth-first search in detail in a more general setting in the next chapter, and we consider several other connectivity algorithms there. For example, a slightly more general version of *Program* 17.11 gives us a way to pass through all the edges in the graph, building a vertex-indexed array that allows a client to test in constant time whether there exists a `path` connecting any two vertices.

Property 17.2 can substantially overestimate the actual running time of *Program* 17.11, because it might find a `path` after examining only a few edges. For the moment, our interest is in knowing a method that is guaranteed to find in linear time a `path` connecting any two vertices in any graph. By contrast, other problems that appear similar are much more difficult to solve. For example, consider the following problem, where we seek paths connecting pairs of vertices, but add the restriction that they visit all the other vertices in the graph, as well.

Figure 17.18 Hamilton Tour *The graph at the top has the Hamilton tour 0-6-4-2-1-3-5-0, which visits each vertex exactly once and returns to the start vertex, but the graph at the bottom has no such tour.*

IPR2023-00124
KASPERSKY EX 1036 Page 82





Hamilton path Given two vertices, is there a simple `path` connecting them that visits every vertex in the graph exactly once? If the `path` is from a vertex back to itself, this problem is known as the *Hamilton tour* problem. Is there a cycle that visits every vertex in the graph exactly once?

Program 17.12 Hamilton path

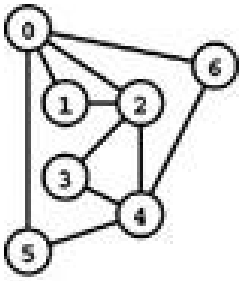
The function `GRAPHpathH` searches for a Hamilton `path` from `v` to `w`. It uses a recursive function that differs from the one in *Program* 17.11 in just two respects: First, it returns successfully only if it finds a `path` of length `V`; second, it resets the `visited` marker before returning unsuccessfully. *Do not expect this function to finish* except for tiny graphs (*see text*).

```
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);
  }
```

At first blush, this problem seems to admit a simple solution: We can write down the simple recursive program for finding a Hamilton `path` that is shown in *Program* 17.12. But this program is not likely to be useful for many graphs, because its worst-case running time is *exponential* in the number of vertices in the graph.

Property 17.3 *A recursive search for a Hamilton tour could take exponential time.*

Figure 17.19 Hamilton-Tour–search Trace *This trace shows the edges checked by Program 17.12 when discovering that the graph shown at the top has no Hamilton tour. For brevity, edges to marked vertices are omitted.*




```
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
```

Proof: Consider a graph where vertex V-1 is isolated and the edges, with the other $V-1$ vertices, constitute a complete graph. Program 17.12 will never find a Hamilton path, but it is easy to see by induction that it will examine all of the $(V-1)!$ paths in the complete graph, all of which involve $V-1$ recursive calls. The total number of recursive calls is therefore about $V!$, or about $(V/e)^V$, which is higher than any constant to the Vth power. ■

Our implementations *Program* 17.11 for finding simple paths and *Program* 17.12 for finding Hamilton paths are extremely similar, and both programs terminate when all the elements of the `visited` array are set to 1. Why are the running times so dramatically dif-

ferent? *Program* 17.11 is guaranteed to finish quickly because it sets at least one element of the `visited` array to `1` each time `pathR` is called. *Program* 17.12, on the other hand, can set `visited` elements back to `0`, so we cannot guarantee that it will finish quickly.

When searching for simple paths, in *Program* 17.11, we know that, if there is a `path` from `v` to `w`, we will find it by taking one of the edges `v-t` from `v`, and the same is true for Hamilton paths. But there this similarity ends. If we cannot find a simple `path` from `t` to `w`, then we can conclude that there is no simple `path` from `v` to `w` that goes through `t`; but the analogous situation for Hamilton paths does not hold. It could be that case that there is no Hamilton `path` to `w` that starts with `v-t`, but there is one that starts with `v-x-t` for some other vertex `x`. We have to make a recursive call from `t` corresponding to every `path` that leads to it from `v`. In short, we may have to check every `path` in the graph.

It is worthwhile to reflect on just how slow a factorial-time algorithm is. If we could process a graph with 15 vertices in 1 second, it would take 1 day to process a graph with 19 vertices, over 1 year for 21 vertices, and over 6 centuries for 23 vertices. Faster computers do not help much, either. A computer that is 200,000 times faster than our original one would still take more than a day to solve that 23-vertex problem. The cost to process graphs with 100 or 1000 vertices is too high to contemplate, let alone graphs of the size that we might encounter in practice. It would take millions of pages in this book just to write down the number of centuries required to process a graph that contained millions of vertices.

In Chapter 5, we examined a number of simple recursive programs that are similar in character to *Program* 17.12 but that could be drastically improved with top-down dynamic programming. This recursive program is entirely different in character: The number of intermediate results that would have to be saved is exponential. Despite many people doing an extensive amount of work on the problem, no one has been able to find any algorithm that can promise reasonable performance for large (or even medium-sized) graphs.

Now, suppose that we change the restriction from having to visit all the *vertices* to having to visit all the *edges*. Is this problem easy, like finding a simple path, or hopelessly difficult, like finding a Hamilton path?

Euler path Is there a `path` connecting two given vertices that uses each *edge* in the graph exactly once? The `path` need not be simple—vertices may be visited multiple times. If the `path` is from a vertex back to itself, we have the *Euler tour* problem. Is there a cyclic `path` that uses each edge in the graph exactly once? We prove in the corollary to Property 17.4 that the `path` problem is equivalent to the tour problem in a graph formed

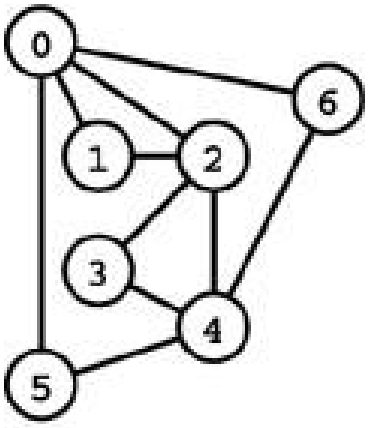
by adding an edge connecting the two vertices. *Figure* 17.20 gives two small examples.

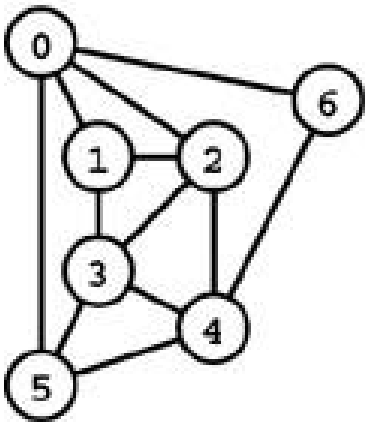
This classical problem was first studied by L. Euler in 1736. Indeed, some people trace the origin of the study of graphs and graph theory to Euler's work on this problem, starting with a special case known as the *bridges of Königsberg* problem (see *Figure* 17.21). The Swiss town of Königsberg had seven bridges connecting riverbanks and islands, and people in the town found that they could not seem to cross all the bridges without crossing one of them twice. Their problem amounts to the Euler tour problem.

These problems are familiar to puzzle enthusiasts. They are commonly seen in the form of puzzles where you are to draw a given figure without lifting your pencil from the paper, perhaps under the constraint that you must start and end at particular points. It is natural for us to consider Euler paths when developing graph-processing algorithms, because a Euler `path` is an efficient representation of the graph (putting the edges in a particular order) that we might consider as the basis for developing efficient algorithms.

Euler showed that it is easy to determine whether or not a `path` exists, because all that we need to do is to check the degree of each of the vertices. The property is easy to state and apply, but the proof is a tricky exercise in graph theory.

Figure 17.20 Euler Tour and `Path` Examples *The graph at the top has the Euler tour* `0-1-2-0-6-4-3-2-4-5-0`*, which uses all the edges exactly once. The graph at the bottom no such tour, but it has the Euler* `path` `1-2-0-1-3-4-2-3-5-4-6-0-5`.





Property 17.4 *A graph has a Euler tour if and only if it is connected and all its vertices are of even degree.*

Proof: To simplify the proof, we allow self-loops and parallel edges, though it is not difficult to modify the proof to show that this property also holds for simple graphs (see Exercise 17.94).

If a graph has a Euler tour, then it must be connected because the tour defines a `path` connecting each pair of vertices. Also, any given vertex v must be of even degree because when we traverse the tour (starting anywhere else), we enter v on one edge and leave on a different edge (neither of which appear again on the tour); so the number of edges incident upon v must be twice the number of times we visit v when traversing the tour, an even number.

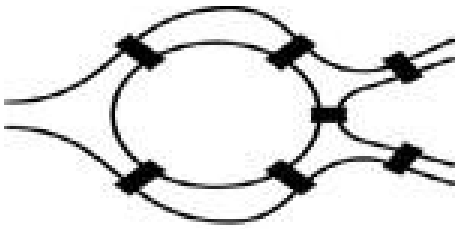
To prove the converse, we use induction on the number of edges. The claim is certainly true for graphs with no edges. Consider any connected graph that has more than one edge, with all vertices of even degree. Suppose that, starting at any vertex `v`, we follow and remove any edge, and we continue doing so until arriving at a vertex that has no more edges. This process certainly must terminate, since we delete an edge at every step, but

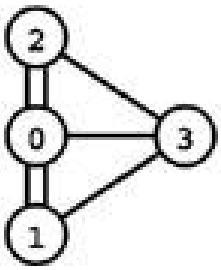
what are the possible outcomes? *Figure* 17.22 illustrates examples. Immediately, we see that we must end up back at v, because we end at a vertex other than v if and only if that vertex had an odd degree when we started.

One possibility is that we trace the full tour; if so, we are done. Otherwise, all the vertices in the graph that remains have even degree, but it may not be connected. Still, each connected component has a Euler tour by the inductive hypothesis. Moreover, the cyclic `path` just removed connects those tours together into a Euler tour for the original graph: traverse the cyclic path, taking detours to do the Euler tours for the connected components defined by deleting the edges on the cyclic `path` (each detour is a proper Euler tour that takes us back to the vertex on the cyclic `path` where it started). Note that a detour may touch the cyclic `path` multiple times (see Exercise 17.99). In such a case, we take the detour only once (say, when we first encounter it).■

Corollary *A graph has a Euler* `path` *if and only if it is connected and exactly two of its vertices are of odd degree.*

Figure 17.21 Bridges of Königsberg *A well-known problem studied by Euler has to do with town of Königsberg, in which there is an island at the point where the river Pregel forks. There are seven bridges connecting the island with the two banks of the river and the land between the forks, as shown in the diagram at top. Is there a way to cross the seven bridges in a continuous walk through the town, without recrossing any of them? If we label the island 0, the banks 1 and 2, and the land between the forks 3 and define an edge corresponding to each bridge, we get the multigraph shown at the bottom. The problem is to find a* `path` *through this graph that uses each edge exactly once.*





IPR2023-00124
KASPERSKY EX 1036 Page 89

Program 17.13 Euler path existence

This function, which is based upon the corollary to Property 17.4, tests whether there is an Euler path from v to w in a connected graph, using the GRAPHdeg ADT function from Exercise 17.40. It takes time proportional to V, not including preprocessing time to check connectivity and to build the vertex-degree table used by GRAPHdeg.

```
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;
  }
```

Proof: This statement is equivalent to Property 17.4 in the graph formed by adding an edge connecting the two vertices of odd degree (the ends of the path).■

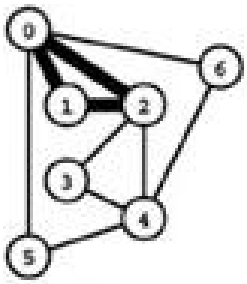
Therefore, for example, there is no way for anyone to traverse all the bridges of Königsberg in a continuous path without retracing their steps, since all four vertices in the corresponding graph have odd degree (see *Figure* 17.21).

As discussed in Section 17.5, we can find all the vertex degrees in time proportional to E for the adjacency-lists or set-of-edges representation and in time proportional to V^2 for the adjacency-matrix representation, or we can maintain a vertex-indexed array with vertex degrees as part of the graph representation (see Exercise 17.40). Given the array, we can check whether Property 17.4 is satisfied in time proportional to V. *Program* 17.13 implements this strategy, and demonstrates that determining whether a given graph has a Euler path is an easy computational problem. This fact is significant because we have little intuition to suggest that the problem should be easier than determining whether a given graph has a Hamilton path.

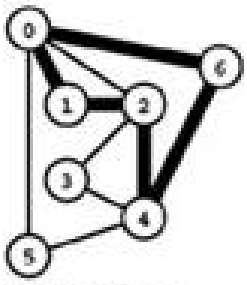
Figure 17.22 Partial Tours *Following edges from any vertex in a graph that has an Euler tour always takes us back to that vertex, as shown in these examples. The*

IPR2023-00124
KASPERSKY EX 1036 Page 90

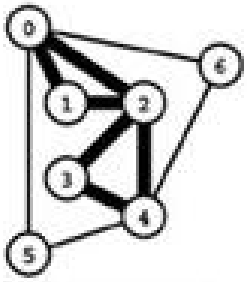
cycle does not necessarily use all the edges in the graph.



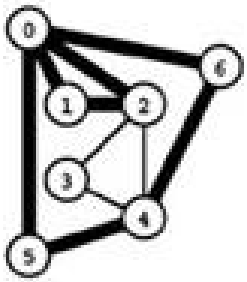
1-0-2-1



6-0-1-2-4-6



2-0-1-2-3-4-2



0-6-4-5-0-2-1-0

Now, suppose that we actually wish to find a Euler path. We are treading on thin ice because a direct recursive implementation (find a `path` by trying an edge, then making a recursive call to find a `path` for the rest of the graph) will have the same kind of factorial-time performance as *Program* 17.12. We expect not to have to live with such performance because it is so easy to test whether a `path` exists, so we seek a better algorithm. It is possible to avoid factorial-time blowup with a fixed-cost test for determining whether or not to use an edge (rather than unknown costs from the recursive call), but we leave this approach as an exercise (see Exercises 17.96 and 17.97).

IPR2023-00124
KASPERSKY EX 1036 Page 91

Program 17.14 Linear-time Euler path (adjacency lists)

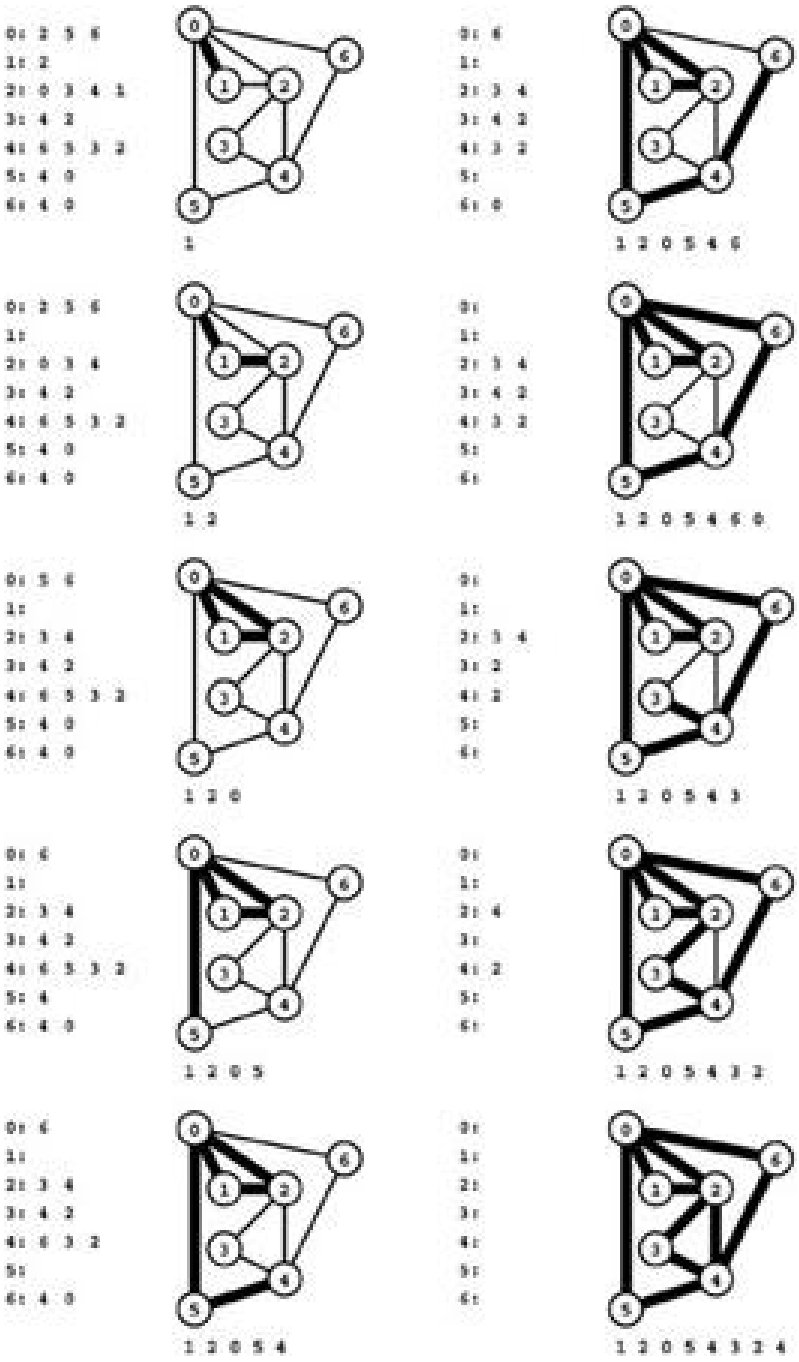
The function `pathEshow` prints an Euler `path` between `v` and `w`. With a constant-time implementation of `GRAPHremoveE` (see Exercise 17.44), it runs in linear time. The auxiliary function `path` follows and removes edges on a cyclic `path` and pushes vertices onto a stack, to be checked for side loops (*see text*). The main loop calls `path` as long as there are vertices with side loops to traverse.

```
#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");
  }
```

Another approach is suggested by the proof of Property 17.4. Traverse a cyclic path, deleting the edges encountered and pushing onto a stack the vertices that it encounters, so that (*i*) we can trace back along that path, printing out its edges, and (*ii*) we can check each vertex for additional side paths (which can be spliced into the main path). This process is illustrated in *Figure* 17.23.

Figure 17.23 Euler Tour by Removing Cycles *This figure shows how Program 17.14 discovers an Euler tour from 0 back to 0 in a sample graph. Thick black edges are those on the tour, the stack contents are listed below each diagram, and adjacency lists for the non-tour edges are shown at left. First, the program adds the edge* `0-1` *to the tour and removes it from the adjacency lists (in two places) (top left, lists at left). Second, it adds* `1-2` *to the tour in the same way*

(left, second from top). Next, it winds up back at 0 but continues to do another cycle `0-5-4-6-0`*, winding up back at* `0` *with no more edges incident upon* `0` *(right, second from top). Then it pops the isolated vertices* `0` *and* `6` *from the stack until* `4` *is at the top and starts a tour from* `4` *(right, third from from top), which takes it to* `3, 2`*, and back to* `4`*, whereupon it pops all the now-isolated vertices* `4, 2, 3`*, and so forth. The sequence of vertices popped from the stack defines the Euler tour* `0-6-4-2-3-4-5-0-2-1-0` *of the whole graph.*



Program 17.14 is an implementation along these lines, for an adjacency-lists graph ADT. It assumes that a Euler `path` exists, and it destroys the graph representation; so the client has responsibility to use `GRAPHpathE, GRAPHcopy`, and `GRAPHdestroy` as appropriate.

IPR2023-00124
KASPERSKY EX 1036 Page 93

The code is tricky—novices may wish to postpone trying to understand it until gaining more experience with graph-processing algorithms in the next few chapters. Our purpose in including it here is to illustrate that good algorithms and clever implementations can be very effective for solving some graph-processing problems.

Property 17.6 *We can find a Euler tour in a graph, if one exists, in linear time.*

We leave a full induction proof as an exercise (see Exercise 17.101). Informally, after the first call on `path`, the stack contains a `path` from `v` to `w`, and the graph that remains (after removal of isolated vertices) consists of the smaller connected components (sharing at least one vertex with the `path` so far found) that also have Euler tours. We pop isolated vertices from the stack and use `path` to find Euler tours that contain the nonisolated vertices, in the same manner. Each edge in the graph is pushed onto (and popped from) the stack exactly once, so the total running time is proportional to E.■

Despite their appeal as a systematic way to traverse all the edges and vertices, we rarely use Euler tours in practice because few graphs have them. Instead, we typically use depth-first search to explore graphs, as described in detail in Chapter 18. Indeed, as we shall see, doing depth-first search in an undirected graph amounts to computing a *two-way Euler tour*: a `path` that traverses each edge exactly *twice*, once in each direction.

In summary, we have seen in this section that it is easy to find simple paths in graphs, that it is even easier to know whether we can tour all the *edges* of a large graph without revisiting any of them (by just checking that all vertex degrees are even), and that there is even a clever algorithm to find such a tour; but that it is practically impossible to know whether we can tour all the graph's *vertices* without revisiting any. We have simple recursive solutions to all these problems, but the potential for exponential growth in the running time renders some of the solutions useless in practice. Others provide insights that lead to fast practical algorithms.

This range of difficulty among apparently similar problems that is illustrated by these examples is typical in graph processing, and is fundamental to the theory of computing. As discussed briefly in Section 17.8 and in detail in Part 8, we must acknowledge what seems to be an insurmountable barrier between problems that seem to require exponential time (such as the Hamilton tour problem and many other commonly encountered problems) and problems for which we know algorithms that can guarantee to find a solution in polynomial time (such as the Euler tour problem and many other commonly encountered problems). In this book, our primary objective is to develop efficient algorithms for problems in the latter class.

IPR2023-00124
KASPERSKY EX 1036 Page 94

Exercises

▷ **17.84** Show, in the style of *Figure* 17.17, the trace of recursive calls (and vertices that are skipped), when *Program* 17.11 finds a `path` from `0` to `5` in the graph

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

17.85 Modify the recursive function in *Program* 17.11 to print out a trace like *Figure* 17.17, using a global variable as described in the text.

17.86 Do Exercise 17.85 by adding an argument to the recursive function to keep track of the depth.

17.87 Using the method described in the text, give an implementation of GRAPHpath that calls a client-supplied function for each edge on a `path` from `v` to `w`, if any such `path` exists.

○ **17.88** Modify *Program* 17.11 such that it takes a third argument `d` and tests the existence of a `path` connecting `u` and `v` of length greater than `d`. In particular, `GRAPHpath (v, v, 2)` should be nonzero if and only if `v` is on a cycle.

▷ **17.89** Modify `GRAPHpath` to use the adjacency-lists graph representation (*Program* 17.6).

● **17.90** Run experiments to determine empirically the probability that *Program* 17.11 finds a `path` between two randomly chosen vertices for various graphs (see Exercises 17.63–76), and to calculate the average length of the paths found for each type of graph.

○ **17.91** Consider the graphs defined by the following four sets of edges:

```
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
```

Which of these graphs have Euler tours? Which of them have Hamilton tours?

○ **17.92** Give necessary and sufficient conditions for a *directed* graph to have a (directed) Euler tour.

17.93 Prove that every connected undirected graph has a two-way Euler tour.

17.94 Modify the proof of Property 17.4 to make it work for graphs with parallel edges and self-loops.

▷ **17.95** Show that adding one more bridge could give a solution to the bridges-of-Königsberg problem.

● **17.96** Prove that a connected graph has a Euler `path` from `v` to `w` only if it has an edge incident on `v` whose removal does not disconnect the graph (except possibly by isolating `v`).

● **17.97** Use Exercise 17.96 to develop an efficient recursive method for finding a Euler tour in a graph that has one. Beyond the basic graph ADT functions, you may use the ADT functions `GRAPHdeg` (see Exercise 17.40) and `GRAPHpath` (see *Program* 17.11). Implement and test your program for the adjacency-matrix graph representation.

17.98 Develop a representation-independent version of *Program* 17.14 that uses a general interface for visiting all edges adjacent to a given vertex (see Exercise 17.60). *Note*: Be careful of interactions between your code and the `GRAPHremoveE` function. Make sure that your implementation works properly in the presence of parallel edges and self-loops.

▷ **17.99** Give an example where the graph remaining after the first call to `path` in *Program* 17.14 is not connected (in a graph that has a Euler tour).

▷ **17.100** Describe how to modify *Program* 17.14 so that it can be used to detect whether or not a given graph has a Euler tour, in linear time.

17.101 Give a complete proof by induction that the linear-time Euler tour algorithm described in the text and implemented in *Program* 17.14 properly finds a Euler tour.

○ **17.102** Find the number of V-vertex graphs that have a Euler tour, for as large a value of V as you can feasibly afford to do the computation.

● **17.103** Run experiments to determine empirically the average length of the `path` found by the first call to `path` in *Program* 17.14 for various graphs (see Exercises 17.63–76). Calculate the probability that this `path` is cyclic.

○ **17.104** Write a program that computes a sequence of $2^n + n - 1$ bits in which no two pairs of n consecutive bits match. (For example, for $n = 3$, the sequence `0001110100` has this property.) *Hint*: Find a Euler tour in a de Bruijn digraph.

▷ **17.105** Show, in the style of *Figure* 17.19, the trace of recursive calls (and vertices that are skipped), when *Program* 17.11 finds a Hamilton tour in the graph

IPR2023-00124
KASPERSKY EX 1036 Page 96

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

○ **17.106** Modify *Program* 17.12 to print out the Hamilton tour if it finds one.

● **17.107** Find a Hamilton tour of the graph

```
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
```

or show that none exists.

●● **17.108** Determine how many V-vertex graphs have a Hamilton tour, for as large a value of V as you can feasibly afford to do the computation.

17.8 GRAPH-PROCESSING PROBLEMS

Armed with the basic tools developed in this chapter, we consider in Chapters 18 through 22 a broad variety of algorithms for solving graph-processing problems. These algorithms are fundamental ones and are useful in many applications, but they serve as only an introduction to the subject of graph algorithms. Many interesting and useful algorithms have been developed that are beyond the scope of this book, and many interesting problems have been studied for which good algorithms have still not yet been invented. As is true in any domain, the first challenge that we face is determining how difficult to solve a given problem is. For graph processing, this decision might be far more difficult than we might imagine, even for problems that appear to be simple to solve. Moreover, our intuition is not always helpful in distinguishing easy problems from difficult or hitherto unsolved ones. In this section, we describe briefly important classical problems and the state of our knowledge of them.

Given a new graph-processing problem, what type of challenge do we face in developing an implementation to solve it? The unfortunate truth is that there is no good method to answer this question for any problem that we might encounter, but we can provide a general description of the difficulty of solving various classical graph-processing problems. To this end, we will roughly categorize the problems according to the difficulty of solving them, as follows:

• Easy

• Tractable

IPR2023-00124
KASPERSKY EX 1036 Page 97

• Intractable

• Unknown

These terms are intended to convey information relative to one another and to the current state of knowledge about graph algorithms.

As indicated by the terminology, our primary reason for categorizing problems in this way is that there are many graph problems, such as the Hamilton tour problem, that no one knows how to solve efficiently. We will eventually see (in Part 8) how to make that statement meaningful in a precise technical sense; at this point, we can at least be warned that we face significant obstacles to writing an efficient program to solve these problems.

We defer full context on many of the graph-processing problems until later in the book. Here, we present brief statements that are easily understood, to introduce the general issue of classifying the difficulty of graph-processing problems.

An *easy* graph-processing problem is one that can be solved by the kind of elegant and efficient short programs to which we have grown accustomed in Parts 1 through 4. We often find the running time to be linear in the worst case, or bounded by a small-degree polynomial in the number of vertices or the number of edges. Generally, as we have done in many other domains, we can establish that a problem is easy by developing a brute-force solution that, although it may be too slow for huge graphs, is useful for small and even intermediate-sized problems. Then, once we know that the problem is easy, we look for efficient solutions that we might use in practice, and try to identify the best among those. The Euler tour problem that we considered in Section 17.7 is a prime example of such a problem, and we shall see many others in Chapters 18 through 22 including, most notably, the following.

Simple connectivity Is a given graph connected? That is, is there a `path` connecting every pair of vertices? Is there a cycle in the graph, or is it a forest? Given two vertices, are they on a cycle? We first considered these basic graph-processing question in Chapter 1. We consider numerous solutions to such problems in Chapter 18. Some are trivial to implement in linear time; others have rather sophisticated linear-time solutions that bear careful study.

Strong connectivity in digraphs Is there a *directed* `path` connecting every pair of vertices in a digraph? Given two vertices, are they connected by directed paths in both directions (are they on a directed cycle)? Implementing efficient solutions to these problems is a much more challenging task than for the corresponding simple-connectivity problem in undirected graphs, and much of Chapter 19 is devoted to studying them. De-

spite the clever intricacies involved in solving them, we classify the problems as easy because we can write a compact, efficient, and useful implementation.

Transitive closure What set of vertices can be reached by following directed edges from each vertex in a digraph? This problem is closely related to strong connectivity and to other fundamental computational problems. We study classical solutions that amount to a few lines of code in Chapter 19.

Minimum spanning tree In a weighted graph, find a minimum-weight set of edges that connects all the vertices. This is one of the oldest and best-studied graph-processing problems; Chapter 20 is devoted to the study of various classical algorithms to solve it. Researchers still seek faster algorithms for this problem.

Single-source shortest paths What are the shortest paths connecting a given vertex v with each other vertex in a weighted digraph (network)? Chapter 21 is devoted to the study of this problem, which is extremely important in numerous applications. The problem is decidedly *not* easy if edge weights can be negative.

A *tractable* graph-processing problem is one for which an algorithm is known whose time and space requirements are guaranteed to be bounded by a polynomial function of the size of the graph ($V + E$). All easy problems are tractable, but we make a distinction because many tractable problems have the property that developing an efficient practical program to solve is an extremely challenging, if not impossible, task. Solutions may be too complicated to present in this book, because implementations might require hundreds or even thousands of lines of code. The following examples are two of the most important problems in this class.

Planarity Can we draw a given graph without any of the lines that represent edges intersecting? We have the freedom to place the vertices anywhere, so we can solve this problem for many graphs, but it is impossible to solve for many other graphs. A remarkable classical result known as *Kuratowski's theorem* provides an easy test for determining whether a graph is planar: it says that the only graphs that cannot be drawn with no edge intersections are those that contain some subgraph that, after removing vertices of degree 2, is isomorphic to one of the graphs in *Figure* 17.24. A straightforward implementation of that test, even without taking the vertices of degree 2 into consideration, would be too slow for large graphs (see Exercise 17.111), but in 1974 R. Tarjan developed an ingenious (but intricate) algorithm for solving the problem in *linear* time, using a depth-first search scheme that extends those that we consider in Chapter 18. Tarjan's algorithm does not necessarily give a practical layout; it just certifies that a layout exists. As discussed in Section 17.1, developing a visually pleasing layout in applications where vertices do not necessarily relate directly to the physical world has turned out to be a challenging research

problem.

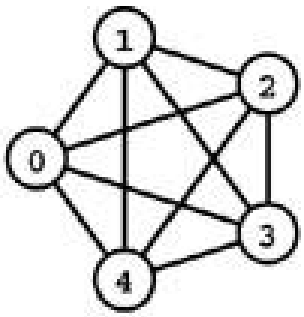
Matching Given a graph, what is the largest subset of its edges with the property that no two are connected to the same vertex? This classical problem is known to be solvable in time proportional to a polynomial function of V and E, but a fast algorithm that is suitable for huge graphs is still an elusive research goal. The problem is easier to solve when restricted in various ways. For example, the problem of matching students to available positions in selective institutions is an example of *bipartite matching*: We have two different types of vertices (students and institutions) and we are concerned with only those edges that connect a vertex of one type with a vertex of the other type. We see a solution to this problem in Chapter 22.

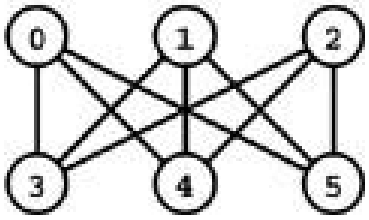
The solutions to some tractable problems have never been written down as programs, or have running times so high that we could not contemplate using them in practice. The following example is in this class. It also demonstrates the capricious nature of the mathematical reality of the difficulty of graph processing.

Even cycles in digraphs Does a given digraph have a cycle of even length? This question would seem simple to resolve because the corresponding question for undirected graphs is easy to solve (see Section 18.4), as is the question of whether a digraph has a cycle of odd length. However, for many years, the problem was not sufficiently well understood for us even to know whether or not there exists an efficient algorithm for solving it (*see reference section*). A theorem establishing the existence of an efficient algorithm was proved in 1999, but the method is so complicated that no mathematician or programmer would attempt to implement it.

Figure 17.24 Forbidden Subgraphs in Planar Graphs *Neither of these graphs can be drawn in the plane without intersecting edges, nor can any graph that contains either of these graphs as a subgraph (after we remove vertices of degree two); but all other graphs can be so drawn.*

IPR2023-00124
KASPERSKY EX 1036 Page 100





One of the important themes of Chapter 22 is that many tractable graph problems are best handled by algorithms that can solve a whole class of problems in a general setting. The *shortest-paths* algorithms of Chapter 21, the *network-flow* algorithms of Chapter 22, and the powerful *network-simplex* algorithm of Chapter 22 are capable of solving many graph problems that otherwise might present a significant challenge. Examples of such problems include the following.

Assignment This problem, also known as *bipartite weighted matching*, is to find a perfect matching of minimum weight in a bipartite graph. It is easily solved with network-flow algorithms. Specific methods that attack the problem directly are known, but they have been shown to be essentially equivalent to network-flow solutions.

General connectivity What is the minimum number of edges whose removal will separate a graph into two disjoint parts (edge connectivity)? What is the minimum number of vertices whose removal will separate a graph into two disjoint parts (vertex connectivity)? As we see in Chapter 22, these problems, although difficult to solve directly, can both be solved with network-flow algorithms.

Mail carrier Given a graph, find a tour with a minimal number of edges that uses every edge in the graph *at least* once (but is allowed to use edges multiple times). This problem is much more difficult than the Euler tour problem but much less difficult than the Hamilton tour problem.

The step from convincing yourself that a problem is tractable to providing software that can be used in practical situations can be a large step, indeed. On the one hand, when proving that a problem is tractable, researchers tend to brush past numerous details that have to be dealt with in an implementation; on the other hand, they have to account for

numerous potential situations even though they may not arise in practice. This gap between theory and practice is particularly acute when investigators are considering graph algorithms, both because mathematical research is filled with deep results describing a bewildering variety of structural properties that we may need to take into account when processing graphs, and because the relationships between those results and the properties of graphs that arise in practice are little understood. The development of general schemes such as the network-simplex algorithm has been an extremely effective approach to dealing with such problems.

An *intractable* graph-processing problem is one for which there is no known algorithm that is guaranteed to solve the problem in a reasonable amount of time. Many such problems have the characteristic that we can use a brute-force method where we can try all possibilities to compute the solution—we consider them to be intractable because there are far too many possibilities to consider. This class of problems is extensive, and includes many important problems that we would like to know how to solve. The term *NP-hard* describes the problems in this class. Most experts believe that no efficient algorithms exist for these problems. We consider the bases for this terminology and this belief in more detail in Part 8. The Hamilton `path` problem that we discussed in Section 17.7 is a prime example of an NP-hard graph-processing problem, as are those on the following list.

Longest path What is the longest simple `path` connecting two given vertices in a graph? Despite its apparent similarity to shortest-paths problems, this problem is a version of the Hamilton tour problem, and is NP-hard.

Colorability Is there a way to assign one of k colors to each of the vertices of a graph such that no edge connects two vertices of the same color? This classical problem is easy for $k = 2$ (see Section 18.4), but it is NP-hard for $k = 3$.

Independent set What is the size of the largest subset of the vertices of a graph with the property that no two are connected by an edge? Just as we saw when contrasting the Euler and Hamilton tour problems, this problem is NP-hard, despite its apparent similarity to the matching problem, which is solvable in polynomial time.

Clique What is size of the largest clique (complete subgraph) in a given graph? This problem generalizes part of the planarity problem, because if the largest clique has more than four nodes, the graph is not planar.

These problems are formulated as *existence* problems—we are asked to determine whether or not a particular subgraph exists. Some of the problems ask for the size of the largest subgraph of a particular type, which we can do by framing an existence problem

where we test for the existence of a subgraph of size k that satisfies the property of interest, then use binary search to find the largest. In practice, we actually often want to find a complete solution, which is potentially much harder to do. Four example, the famous *four-color theorem* tells us that it is possible use just four colors to color all the vertices of a planar graph such that no edge connects two vertices of the same color. But the theorem does not tell us how to do so for a particular planar graph: knowing that a coloring exists does not help us find a complete solution to the problem. Another famous example is the *traveling salesperson* problem, which asks us to find the minimum-length tour through all the vertices of a weighted graph. This problem is related to the Hamilton `path` problem, but it is certainly no easier: if we cannot find an efficient solution to the Hamilton `path` problem, we cannot expect to find one for the traveling salesperson problem. As a rule, when faced with difficult problems, we work with the simplest version that we cannot solve. Existence problems are within the spirit of this rule, but they also play an essential role in the theory, as we shall see in Part 8.

The problems just listed are but a few of the thousands of NP-hard problems that have been identified. They arise in all types of computational applications, as we shall discuss in Part 8, but they are particularly prevalent in graph processing; so we have to be aware of their existence throughout this book.

Note that we are insisting that our algorithms *guarantee* efficiency, in the worst case. Perhaps we should instead settle for algorithms that are efficient for typical inputs (but not necessarily in the worst case). Similarly, many of the problems involve *optimization*. Perhaps we should instead settle for a long `path` (not necessarily the longest) or a large clique (not necessarily the maximum). For graph processing, it might be easy to find a good answer for graphs that arise in practice, and we may not even be interested in looking for an algorithm that could find an optimal solution in fictional graphs that we will never see. Indeed, intractable problems can often be attacked with straightforward or general-purpose algorithms similar to *Program* 17.12 that, although they have exponential running time in the worst case, can quickly find a solution (or a good approximation) for specific problem instances that arise in practice. We would be reluctant to use a program that will crash or produce a bad answer for certain inputs, but we do sometimes find ourselves using programs that run in exponential time for certain inputs. We consider this situation in Part 8.

There are also many research results proving intractable problems to remain that way even when we relax various restrictions. Moreover, there are many specific practical problems that we cannot solve because no one knows a sufficiently fast algorithm. In this part of the book, we will label problems NP-hard when we encounter them and interpret this label as meaning, at the very least, that we are not going to expect to find efficient al-

IPR2023-00124
KASPERSKY EX 1036 Page 103

gorithms to solve them and that we are not going to attack them without using advanced techniques of the type discussed in Part 8 (except perhaps to use brute-force methods to solve tiny problems).

There are some graph-processing problems whose difficulty is *unknown*. Neither is there an efficient algorithm known for them, nor are they known to be NP-hard. It is possible, as our knowledge of graph-processing algorithms and properties of graphs expands, that some of these problems will turn out to be tractable, or even easy. The following important natural problem, which we have already encountered (see *Figure* 17.2), is the best-known problem in this class.

Graph isomorphism Can we make two given graphs identical by renaming vertices? Efficient algorithms are known for this problem for many special types of graphs, but the difficulty of the general problem remains open.

The number of significant problems whose intrinsic difficulty is unknown is very small in comparison to the other categories that we have considered, because of intense research in this field over the past several decades. Certain problems in this class, such as graph isomorphism, are of immense practical interest; other problems in this class are of significance primarily by virtue of having resisted classification.

For the class of easy algorithms, we are used to the idea of comparing algorithms with different worst-case performance characteristics and trying to predict performance through analysis and empirical tests. For graph processing, these tasks are particularly challenging because of the difficulty of characterizing the types of graphs that might arise in practice. Fortunately, many of the important classical algorithms have optimal or near-optimal worst-case performance, or have a running time that depends on only the number of vertices and edges, rather than on the graph structure; so we can concentrate on streamlining implementations and still can predict performance with confidence.

Table 17.2 Difficulty of classical graph-processing problems This table summarizes the discussion in the text about the relative difficulty of various classical graph-processing problems, comparing them in rough subjective terms. These examples indicate not only the range of difficulty of the problems but also that classifying a given problem can be a challenging task.

IPR2023-00124
KASPERSKY EX 1036 Page 104

	E	T	I	?
undirected graphs				
connectivity	*			
general connectivity		*		
Euler tour	*			
Hamilton tour			*	
bipartite matching	*			
maximum matching		*		
planarity		*		
maximum clique			*	
2-colorability	*			
3-colorability			*	
shortest paths	*			
longest paths			*	
vertex cover			*	
isomorphism				*
digraphs				
transitive closure	*			
strong connectivity	*			
odd-length cycle	*			
even-length cycle		*		
weighted graphs				
minimum spanning tree	*			
traveling salesperson			*	
networks				
shortest paths (nonnegative weights)	*			
shortest paths (negative weights)			*	
maximum flow	*			
assignment		*		
minimum-cost flow		*		

Key:
- E Easy—efficient classical algorithm known (see reference)
- T Tractable—solution exists (implementation difficult)
- I Intractable—no efficient solution known (NP-hard)
- ? Unknown

In summary, there is a wide spectrum of problems and algorithms known for graph processing. Table 17.2 summarizes some of the information that we have discussed. Every problem comes in different versions for different types of graphs (directed, weighted, bipartite, planar, sparse, dense), and there are thousands of problems and algorithms to consider. We certainly cannot expect to solve every problem that we might encounter, and some problems that appear to be simple are still baffling the experts. Despite a natural a priori expectation that we should have no problem distinguishing easy problems from intractable ones, the many examples that we have discussed illustrate that placing a problem even into these rough categories can turn into a significant research challenge.

IPR2023-00124
KASPERSKY EX 1036 Page 105

As our knowledge about graphs and graph algorithms expands, given problems may move among these categories. Despite a flurry of research activity in the 1970s and intensive work by many researchers since then, the possibility still remains that all the problems that we are discussing will someday be categorized as "easy" (solvable by an algorithm that is compact, efficient, and possibly ingenious).

Having developed this context, we shall press on to consider numerous useful graph-processing algorithms. Problems that we can solve do arise often, the graph algorithms that we study serve well in a great variety of applications, and these algorithms serve as the basis for attacking numerous other problems that we need to handle even if we cannot guarantee efficient solutions.

Exercises

● **17.109** Prove that neither of the two graphs depicted in *Figure* 17.24 is planar.

17.110 Write a function for the adjacency-lists representation (*Program* 17.6) that determines whether or not a graph contains one of the graphs depicted in *Figure* 17.24, using a brute-force algorithm where you test all possible subsets of five vertices for the clique and all possible subsets of six vertices for the complete bipartite graph. *Note*: This test does not suffice to show whether the graph is planar, because it ignores the condition that removing vertices of degree 2 in some subgraph might give one of the two forbidden subgraphs.

17.111 Give a drawing of the graph

```
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
```

that has no intersecting edges, or prove that no such drawing exists.

17.112 Find a way to assign three colors to the vertices of the graph

```
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
```

such that no edge connects two vertices of the same color, or show that it is not possible to do so.

17.113 Solve the independent-set problem for the graph

```
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 What is the size of the largest clique in a de Bruijn graph of order n?

IPR2023-00124
KASPERSKY EX 1036 Page 107

Chapter Eighteen
Graph Search

WE OFTEN LEARN properties of a graph by systematically examining each of its vertices and each of its edges. Determining some simple graph properties—for example, computing the degrees of all the vertices—is easy if we just examine each edge (in any order whatever). Many other graph properties are related to paths, so a natural way to learn them is to move from vertex to vertex along the graph's edges. Nearly all the graph-processing algorithms that we consider use this basic abstract model. In this chapter, we consider the fundamental *graph-search* algorithms that we use to move through graphs, learning their structural properties as we go.

Graph searching in this way is equivalent to exploring a maze. Specifically, passages in the maze correspond to edges in the graph, and points where passages intersect in the maze correspond to vertices in the graph. When a program changes the value of a variable from vertex `v` to vertex `w` because of an edge `v-w`, we view it as equivalent to a person in a maze moving from point `v` to point `w`. We begin this chapter by examining a systematic exploration of a maze. By correspondence with this process, we see precisely how the basic graph-search algorithms proceed through every edge and every vertex in a graph.

In particular, the recursive *depth-first search* (DFS) algorithm corresponds precisely to the particular maze-exploration strategy of Section 18.1. DFS is a classic and versatile algorithm that we use to solve connectivity and numerous other graph-processing problems. The basic algorithm admits two simple implementations: one that is recursive, and another that uses an explicit stack. Replacing the stack with a FIFO queue leads to another classic algorithm, *breadth-first search* (BFS), which we use to solve another class of graph-processing problems related to shortest paths.

The main topics of this chapter are DFS, BFS, their related algorithms, and their application to graph processing. We briefly considered DFS and BFS in Chapter 5; we treat them from first principles here, in the context of graph-search ADT functions that we can extend to solve various graph-processing problems, and use them to demonstrate relationships among various graph algorithms. In particular, we consider a generalized graph-search method that encompasses a number of classical graph-processing algorithms, including both DFS and BFS.

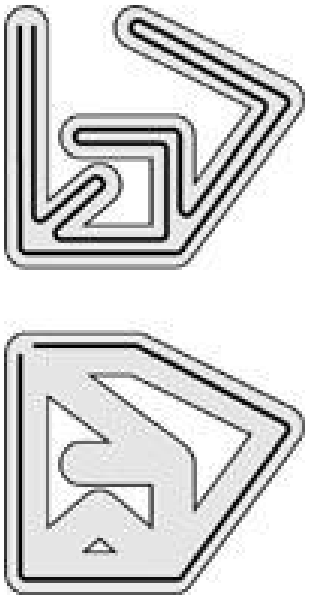
As illustrations of the application of these basic graph-searching methods to solve more complicated problems, we consider algorithms for finding connected components, biconnected components, spanning trees, and shortest paths, and for solving numerous other graph-processing problems. These implementations exemplify the approach that we shall

IPR2023-00124
KASPERSKY EX 1036 Page 108

use to solve more difficult problems in Chapters 19 through 22.

We conclude the chapter by considering the basic issues involved in the analysis of graph algorithms, in the context of a case study comparing several different algorithms for finding the number of connected components in a graph.

Figure 18.1 Exploring a Maze *We can explore every passageway in a simple maze by following a simple rule such as "keep your right hand on the wall." Following this rule in the maze at the top, we explore the whole maze, going through each passage once in each direction. But if we follow this rule in a maze with a cycle, we return to the starting point without exploring the whole maze, as illustrated in the maze at the bottom.*



18.1 EXPLORING A MAZE

It is instructive to think about the process of searching through a graph in terms of an equivalent problem that has a long and distinguished history (*see reference section*): finding our way through a maze that consists of passages connected by intersections. This section presents a detailed study of a basic method for exploring every passage in any given maze. Some mazes can be handled with a simple rule, but most mazes require a more sophisticated strategy (see *Figure* 18.1). Using the terminology *maze* instead of *graph*, *passage* instead of *edge*, and *intersection* instead of *vertex* is making mere semantic distinctions, but, for the moment, doing so will help to give us an intuitive feel for the problem.

One trick for exploring a maze without getting lost that has been known since antiquity (dating back at least to the legend of Theseus and the Minotaur) is to unroll a ball of

string behind us. The string guarantees that we can always find a way out, but we are also interested in being sure that we have explored every part of the maze, and we do not want to retrace our steps unless we have to. To accomplish these goals, we need some way to mark places that we have been. We could use the string for this purpose as well, but we use an alternative approach that models a computer implementation more closely.

We assume that there are lights, initially off, in every intersection, and doors, initially closed, at both ends of every passage. We further assume that the doors have windows and that the lights are sufficiently strong and the passages sufficiently straight that we can determine, by opening the door at one end of a passage, whether or not the intersection at the other end is lit (even if the door at the other end is closed). Our goals are to turn on all the lights and to open all the doors. To reach them, we need a set of rules to follow, systematically. The following maze-exploration strategy, which we refer to as *Trémaux exploration*, has been known at least since the nineteenth century (*see reference section*):

(*i*) If there are no closed doors at the current intersection, go to step (*iii*). Otherwise, open any closed door to any passage leading out of the current intersection (and leave it open).

(*ii*) If you can see that the intersection at the other end of that passage is already lighted, try another door at the current intersection (step (*i*)). Otherwise (if you can see that the intersection at the other end of the passage is dark), follow the passage to that intersection, unrolling the string as you go, turn on the light, and go to step (*i*).

(*iii*) If all the doors at the current intersection are open, check whether you are back at the start point. If so, stop. If not, use the string to go back down the passage that brought you to this intersection for the first time, rolling the string back up as you go, and look for another closed door there (that is, return to step (*i*)).

Figures 18.2 and 18.3 depict a traversal of a sample graph and show that, indeed, every light is lit and every door is opened for that example. The figures depict just one of many possible outcomes of the exploration, because we are free to open the doors in any order at each intersection. Convincing ourselves that this method is always effective is an interesting exercise in mathematical induction.

Figure 18.2 Trémaux Maze Exploration Example *In this diagram, places that we have not visited are shaded (dark) and places that we have visited are white (light). We assume that there are lights in the intersections, and that, when we have opened doors into lighted intersections on both ends of a passage, the passage is lighted. To explore the maze, we begin at* `0` *and take the passage to* `2` *(left, top). Then we proceed to* `6, 4, 3` *and* `5`*, opening the doors to the passages, lighting the intersections as we proceed through them, and leaving a string trailing behind us (left). When we open the door that leads to* `0` *from* `5`*, we can see that* `0` *is*

lighted, so we skip that passage (top right). Similarly, we skip the passage from 5 *to* 4 (right, second from top), *leaving us with nowhere to go from* 5 *except back to* 3 *and then back to* 4, *rolling up our ball of string. When we open the doorway from the passage from* 4 *to* 5, *we can see that* 5 *is lighted through the open door at the other end, and we therefore skip that passage (right, bottom). We never walked down the passage connecting* 4 *and* 5, *but we lighted it by opening the doors at both ends.*

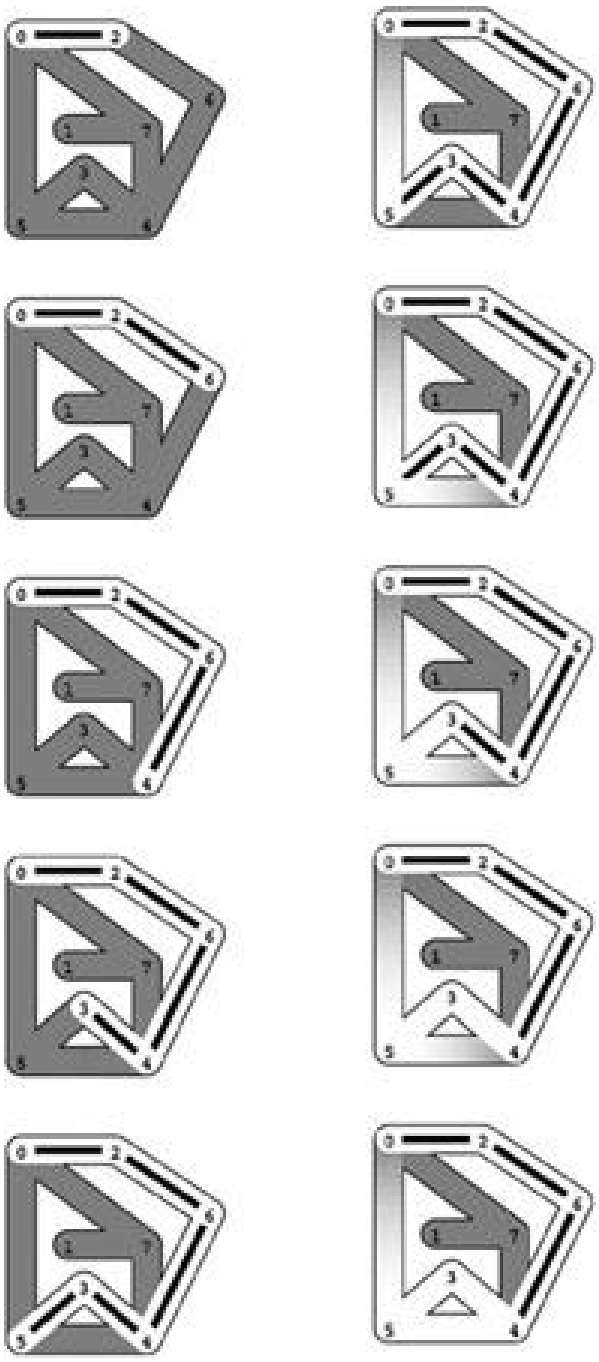
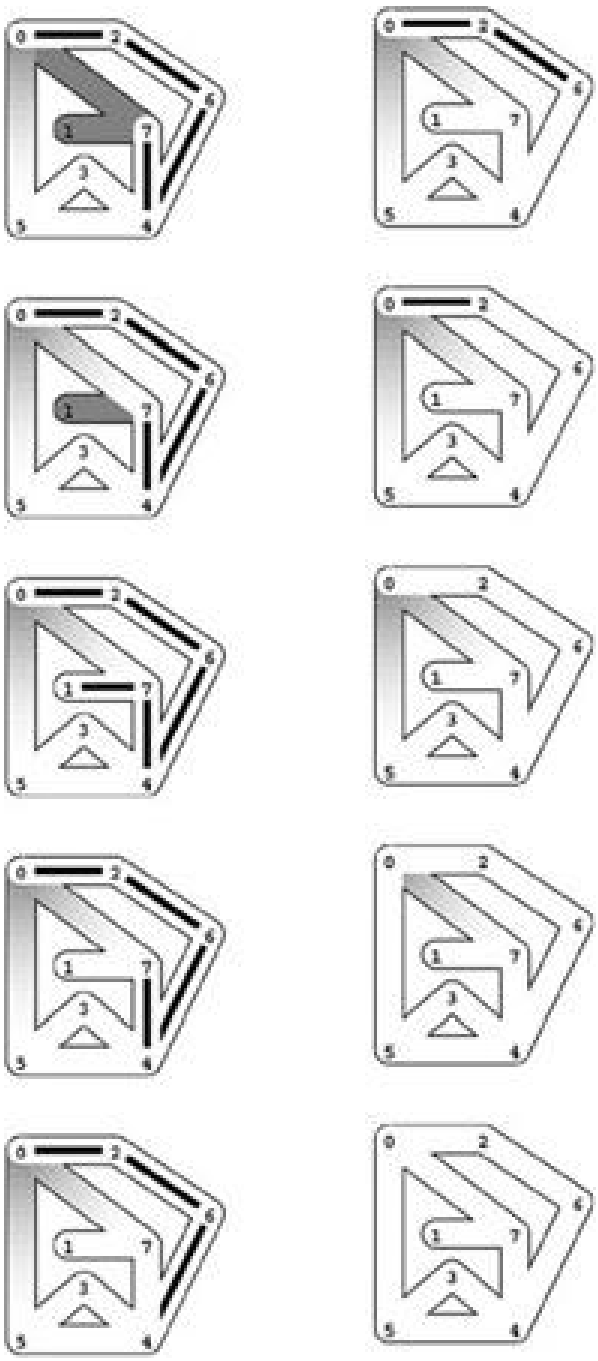


Figure 18.3 Trémaux Maze Exploration Example (continued) *Next, we proceed to* 7 (top left), *open the door to see that* 0 *is lighted* (left, second from top), *and then proceed to* 1 (*left, third from top*). *At this point, most of the maze is traversed, and we use our string to take us back to the beginning, moving from* 1 *to* 7

IPR2023-00124
KASPERSKY EX 1036 Page 111

to 4 *to* 6 *to* 2 *to* 0. *Back at* 0, *we complete our exploration by checking the passages to* 5 (right, second from bottom) *and* 7 (bottom right), *leaving all passages and intersections lighted. Again, the passages connecting* 0 *to* 5 *and* 0 *to* 7 *are both lighted because we opened the doors at both ends, but we did not walk through them.*

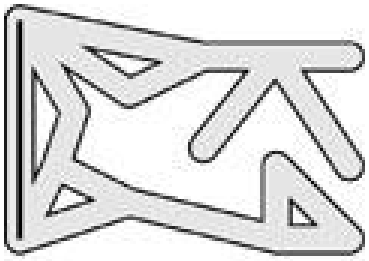


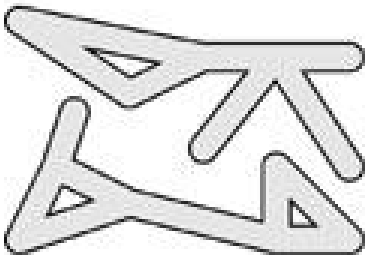
Property 18.1 *When we use Trémaux maze exploration, we light all lights and open all doors in the maze, and end up back where we started.*

Proof: To prove this assertion by induction, we first note that it holds, trivially, for a maze that contains one intersection and no passages—we just turn on the light. For any maze

that contains more than one intersection, we assume the property to be true for all mazes with fewer intersections. It suffices to show that we visit all intersections, since we open all the doors at every intersection that we visit. Now, consider the first passage that we take from the first intersection, and divide the intersections into two subsets: (*i*) those that we can reach by taking that passage without returning to the start, and (*ii*) those that we *cannot* reach from that passage without returning to the start. Applying the inductive hypothesis, we know that we visit all intersections in (*i*) (ignoring any passages back to the start intersection, which is lit) and end up back at the start intersection. Then, applying the the inductive hypothesis again, we know that we visit all intersections in (*ii*) (ignoring the passages from the start to intersections in (*i*), which are lit).■

Figure 18.4 Decomposing a Maze *To prove by induction that Trémaux exploration takes us everywhere in a maze* (*top*)*, we break it into two smaller pieces, by removing all edges connecting the first intersection with any intersection that can be reached from the first passage without going back through the first intersection* (bottom).





From the detailed example in Figures 18.2 and 18.3, we see that there are four different possible situations that arise for each passage that we consider taking:

(*i*) The passage is dark, so we take it.

(*ii*) The passage is the one that we used to enter (it has our string in it), so we use it to exit (and we roll up the string).

(*iii*) The door at the other end of the passage is closed (but the intersection is lit), so we skip the passage.

(*iv*) The door at the other end of the passage is open (and the intersection is lit), so we

skip it.

The first and second situations apply to any passage that we traverse, first at one end and then at the other end. The third and fourth situations apply to any passage that we skip, first at one end and then at the other end. Next, we see how this perspective on maze exploration translates directly to graph search.

Exercises

▷ **18.1** Assume that intersections 6 and 7 (and all the hallways connected to them) are removed from the maze in Figures 18.2 and 18.3, and a hallway is added that connects 1 and 2. Show a Trémaux exploration of the resulting maze, in the style of Figures 18.2 and 18.3.

○ **18.2** Which of the following could *not* be the order in which lights are turned on at the intersections during a Trémaux exploration of the maze depicted in Figures 18.2 and 18.3?

```
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** How many different ways are there to traverse the maze depicted in Figures 18.2 and 18.3 with a Trémaux exploration?

18.2 DEPTH-FIRST SEARCH

Our interest in Trémaux exploration is that this technique leads us immediately to the classic recursive function for traversing graphs: To visit a vertex, we mark it as having been visited, then (recursively) visit all the vertices that are adjacent to it and that have not yet been marked. This method, which we considered briefly in Chapters 3 and 5 and used to solve `path` problems in Section 17.7, is called *depth-first search (DFS)*. It is one of the most important algorithms that we shall encounter. DFS is deceptively simple because it is based on a familiar concept and is easy to implement; in fact, it is a subtle and powerful algorithm that we put to use for numerous difficult graph-processing tasks.

Program 18.1 is a DFS implementation that visits all the vertices and examines all the edges in a connected graph. It uses an adjacency-matrix representation. As usual, the corresponding function for the adjacency-lists representation differs only in the mechanics of accessing the edges (see *Program* 18.2). Like the simple path-search functions that we

considered in Section 17.7, the implementation is based on a recursive function that uses a global array and an incrementing counter to mark vertices by recording the order in which they are visited. *Figure* 18.5 is a trace that shows the order in which *Program* 18.1 visits the edges and vertices for the example depicted in Figures 18.2 and 18.3 (see also *Figure* 18.17). *Figure* 18.6 depicts the same process using standard graph drawings.

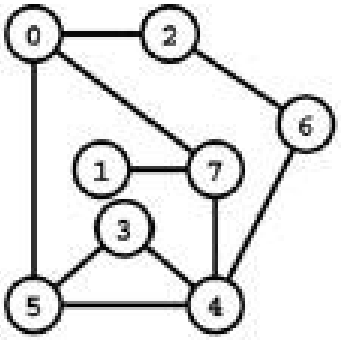
Program 18.1 Depth-first search (adjacency-matrix)

This code is intended for use with a generic graph-search ADT function that initializes a counter `cnt` to `0` and all of the entries in the vertex-indexed array `pre` to `-1`, then calls `search` once for each connected component (see *Program* 18.3), assuming that the call `search (G, EDGE (v, v))` marks all vertices in the same connected component as `v` (by setting their `pre` entries to be nonnegative).

Here, we implement `search` with a recursive function `dfsR` that visits all the vertices connected to `e.w` by scanning through its row in the adjacency matrix and calling itself for each edge that leads to an unmarked vertex.

```
#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));
  }
```

Figure 18.5 DFS Trace *This trace shows the order in which DFS checks the edges and vertices for the adjacency-matrix representation of the graph corresponding to the example in Figures 18.2 and* 18.3 (top) *and traces the contents of the* `pre` *array* (right) *as the search progresses (asterisks represent* `-1`*, for unseen vertices). There are two lines in the trace for every graph edge (once for each orientation). Indentation indicates the level of recursion.*




```
                    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
```

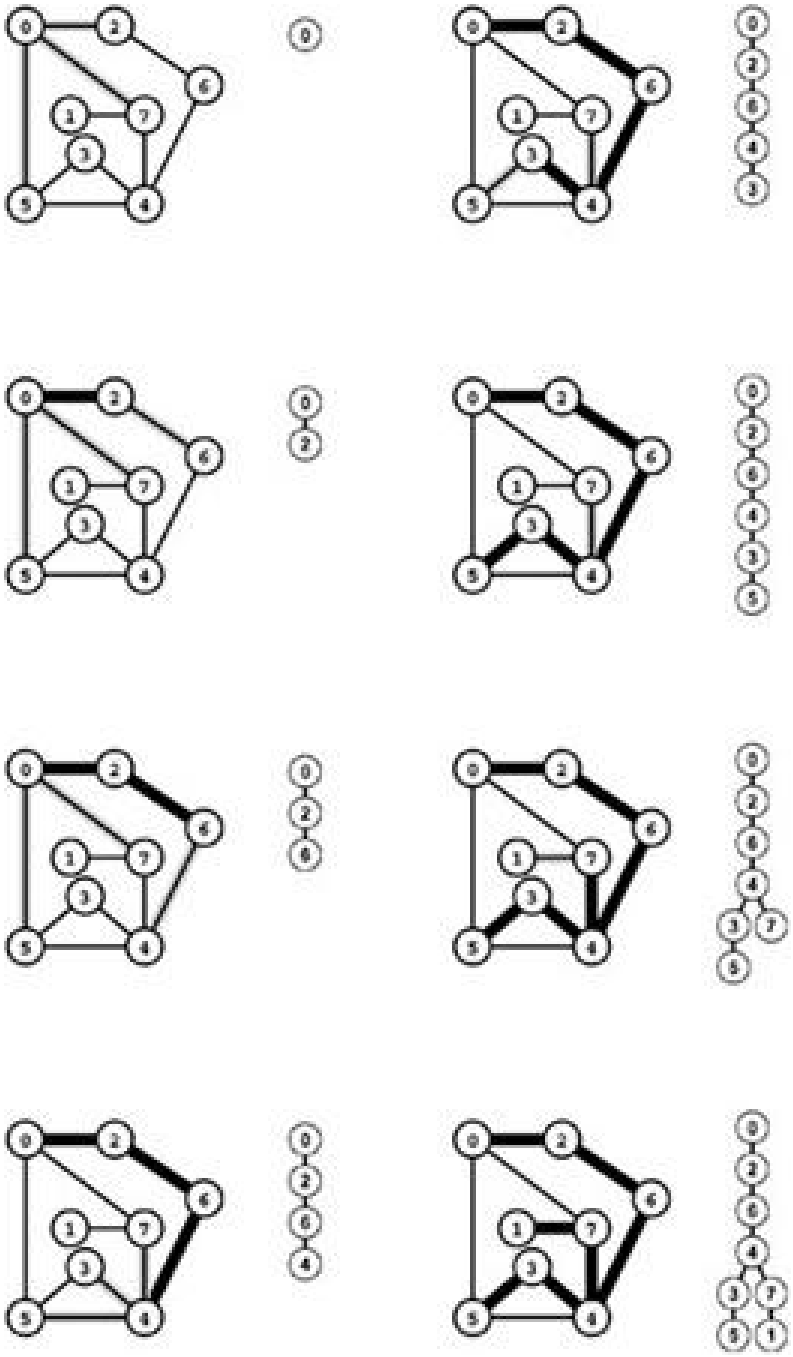
These figures illustrate the dynamics of a recursive DFS and show the correspondence with Trémaux exploration of a maze. First, the vertex-indexed array corresponds to the lights in the intersections: When we encounter an edge to a vertex that we have already visited (see a light at the end of the passage), we do not make a recursive call to follow that edge (go down that passage). Second, the function call–return mechanism in the program corresponds to the string in the maze: When we have processed all the edges adjacent to a vertex (explored all the passages leaving an intersection), we “return” (in both

senses of the word).

In the same way that we encounter each passage in the maze twice (once at each end), we encounter each edge in the graph twice (once at each of its vertices). In Trémaux exploration, we open the doors at each end of each passage; in DFS of an undirected graph, we check each of the two representations of each edge. If we encounter an edge `v-w`, we either do a recursive call (if `w` is not marked) or skip the edge (if `w` is marked). The second time that we encounter the edge, in the opposite orientation `w-v`, we always ignore it, because the destination vertex `v` has certainly already been visited (the first time that we encountered the edge).

Figure 18.6 Depth-First Search *These diagrams are a graphical view of the process depicted in Figure 18.5, showing the DFS recursive-call tree as it evolves. Thick black edges in the graph correspond to edges in the DFS tree shown to the right of each graph diagram. Shaded edges are the candidates to be added to the tree next. In the early stages (left) the tree grows down in a straight line, as we make recursive calls for* `0`, `2`, `6`, *and* `4` *(left). Then we make recursive calls for* `3`, *then* `5` *(right, top two diagrams); and return from those calls to make a recursive call for* `7` *from* `4` *(right, second from bottom) and to* `1` *from* `7` *(right, bottom).*

IPR2023-00124
KASPERSKY EX 1036 Page 117



One difference between DFS as implemented in *Program* 18.1 and Trémaux exploration as depicted in Figures 18.2 and 18.3, although it is inconsequential in many contexts, is worth taking the time to understand. When we move from vertex `v` to vertex `w`, we have not examined any of the entries in the adjacency matrix that correspond to edges from `w` to other vertices in the graph. In particular, we know that there is an edge from `w` to `v` and that we will ignore that edge when we get to it (because `v` is marked as visited). That decision happens at a time different from in the Trémaux exploration, where we open the doors corresponding to the edge from `v` to `w` when we go to `w` for the first time, from `v`. If we were to *close* those doors on the way in and open them on the way out (having identi-

fied the passage with the string), then we would have a precise correspondence between DFS and Trémaux exploration.

We pass an *edge* to the recursive procedure, instead of passing its destination vertex, because the edge tells us how we reached the vertex. Knowing the edge corresponds to knowing which passage led to a particular intersection in a maze. This information is useful in many DFS functions. When we are simply keeping track of which vertices we have visited, this information is of little consequence; but more interesting problems require that we always know from whence we came.

Figure 18.6 also depicts the tree corresponding to the recursive calls as it evolves, in correspondence with *Figure* 18.5. This recursive-call tree, which is known as the *DFS tree*, is a structural description of the search process. As we see in Section 18.4, the DFS tree, properly augmented, can provide a full description of the search dynamics, in addition to just the call structure.

The same basic scheme is effective for an adjacency-lists graph representation, as illustrated in *Program* 18.2. As usual, instead of scanning a row of an adjacency matrix to find the vertices adjacent to a given vertex, we scan through its adjacency list. As before, we traverse (via a recursive call) all edges to vertices that have not yet been visited.

Program 18.2 Depth-first search (adjacency-lists)

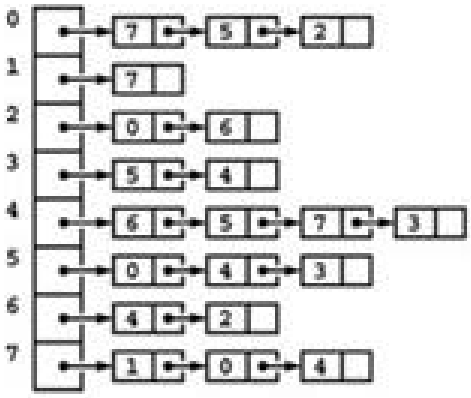
This implementation of `dfsR` is DFS for graphs represented with adjacency lists. The algorithm is the same as for the adjacency-matrix representation (*Program* 18.1): to visit a vertex, we mark it and then scan through its incident edges, making a recursive call whenever we encounter an edge to an unmarked vertex.

```
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));
  }
```

The DFS in *Program* 18.2 ignores self-loops and duplicate edges if they are present, so that we do not necessarily have to take the trouble to remove them from the adjacency lists.

For the adjacency-matrix representation, we examine the edges incident on each vertex in numerical order; for the adjacency-lists representation, we examine them in the order that they appear on the lists. This difference leads to a different recursive search dynamic, as illustrated in *Figure* 18.7. The order in which DFS discovers the edges and vertices in the graph depends entirely on the order in which the edges appear in the adjacency lists in the graph representation. We might also consider examining the entries in each row in the adjacency matrix in some different order, which would lead to another different search dynamic (see Exercise 18.5).

Figure 18.7 DFS Trace (Adjacency Lists) *This trace shows the order in which DFS checks the edges and vertices for the adjacency-lists representation of the same graph as in Figure 18.5.*




```
                    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
```

Despite all of these possibilities, the critical fact remains that DFS visits all the edges and all the vertices connected to the start vertex, *regardless of in what order* it examines the edges incident on each vertex. This fact is a direct consequence of Property 18.1, since the proof of that property does not depend on the order in which the doors are opened at any given intersection. All the DFS-based algorithms that we examine have this same essential property. Although the dynamics of their operation might vary substantially depending on the graph representation and details of the implementation of the search, the recursive structure gives us a way to make relevant inferences about the graph itself, no matter how it is represented and no matter which order we choose to examine the edges incident upon each vertex.

Exercises

▷**18.4** Show, in the style of *Figure* 18.5, a trace of the recursive function calls made for a standard adjacency-matrix DFS of the graph

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

Draw the corresponding DFS recursive-call tree.

▷**18.5** Show, in the style of *Figure* 18.6, the progress of the search if the search function is modified to scan the vertices in reverse order (from `V-1` down to `0`).

○**18.6** Implement DFS using your representation-independent ADT function for processing edge lists from Exercise 17.60.

18.3 GRAPH-SEARCH ADT FUNCTIONS

DFS and the other graph-search methods that we consider later in this chapter all involve following graph edges from vertex to vertex, with the goal of systematically visiting every vertex and every edge in the graph. But following graph edges from vertex to vertex can lead us to all the vertices in only the same connected component as the starting vertex. In general, of course, graphs may not be connected, so we need one call on a search function for each connected component. We will typically use a generic graph-search function that performs the following steps until all of the vertices of the graph have been marked as having been visited:

• Find an unmarked vertex (a *start* vertex).

• Visit (and mark as visited) all the vertices in the connected component that contains the start vertex.

The method for marking vertices is not specified in this description, but we most often use the same method that we used for the DFS implementations in Section 18.2: We initialize all entries in a global vertex-indexed array to a negative integer, and mark vertices by setting their corresponding entry to a positive value. Using this procedure corresponds to using a single bit (the sign bit) for the mark; most implementations are also concerned with keeping other information associated with marked vertices in the array (such as, for the DFS implementations in Section 18.2, the order in which vertices are marked).

IPR2023-00124
KASPERSKY EX 1036 Page 122

Program 18.3 Graph search

We typically use code like this to process graphs that may not be connected. The function `GRAPHsearch` assumes that `search`, when called with a self-loop to `v` as its second argument, sets the `pre` entries corresponding to each vertex connected to `v` to nonnegative values. Under this assumption, this implementation calls `search` once for each connected component in the graph—we can use it for any graph representation and any `search` implementation.

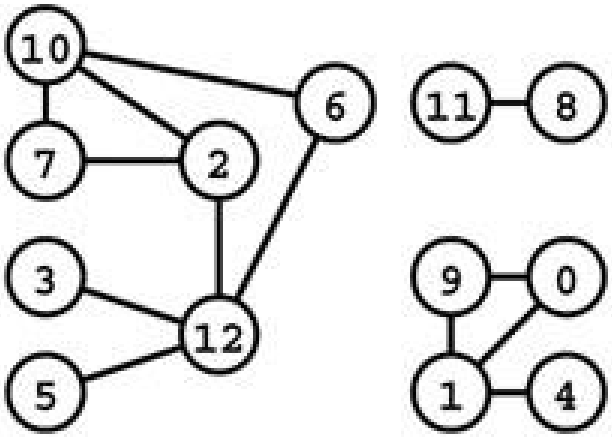
Together with *Program* 18.1 or *Program* 18.2, this code computes the order in which vertices are visited in `pre`. Other DFS-based implementations do other computations, but use the same general scheme of interpreting nonnegative entries in a vertex-indexed array as vertex marks.

```
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));
  }
```

The method for looking for a vertex in the next connected component is also not specified, but we most often use a scan through the array in order of increasing index.

Figure 18.8 Graph Search *The table at the bottom shows vertex marks (contents of the* `pre` *array) during a typical search of the graph at the top. Initially, the function* `GRAPHsearch` *in* *Program* 18.3 *unmarks all vertices by setting the marks all to* `-1` *(indicated by an asterisk). Then it calls* `search` *for the dummy edge* `0-0`*, which marks all of the vertices in the same connected component as* `0` (second row) *by setting them to a nonnegative values (indicated by* `0`*s). In this example, it marks* `0, 1, 4`*, and* `9` *with the values* `0` *through* `3` *in that order. Next, it scans from left to right to find the unmarked vertex* `2` *and calls* `search` *for the dummy edge* `2-2` (third row)*, which marks the seven vertices in the same con-*

nected component as 2. *Continuing the left-to-right scan, it calls* search *for* 8-8 *to mark* 8 *and* 11 (*bottom row*). *Finally,* GRAPHsearch *completes the search by discovering that* 9 *through* 12 *are all marked.*



	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

The GRAPHsearch ADT function in *Program* 18.3 is an implementation that illustrates these choices. *Figure* 18.8 illustrates the effect on the pre array of this function, in conjunction with *Program* 18.1 or *Program* 18.2 (or any graph-search function that marks all of the vertices in the same connected component as its argument). The graph-search functions that we consider also examine all edges indicent upon each vertex visited, so knowing that we visit all vertices tells us that we visit all edges as well, as in Trémaux traversal.

In a connected graph, the ADT function is nothing more than a wrapper that calls search once for 0-0 and then finds that all the other vertices are marked. In a graph with more than one connected component, the ADT function checks all the connected components in a straightforward manner. DFS is the first of several methods that we consider for searching a connected component. No matter which method (and no matter what graph representation) we use, *Program* 18.3 is an effective method for visiting all the graph vertices.

Property 18.2 *A graph-search function checks each edge and marks each vertex in a graph if and only if the search function that it uses marks each vertex and checks each*

edge in the connected component that contains the start vertex.

Proof: By induction on the number of connected components.■

Graph-search functions provide a systematic way of processing each vertex and each edge in a graph. Generally, our implementations are designed to run in linear or near-linear time, by doing a fixed amount of processing per edge. We prove this fact now for DFS, noting that the same proof technique works for several other search strategies.

Property 18.3 *DFS of a graph represented with an adjacency matrix requires time proportional to* V^2.

Proof: An argument similar to the proof of Property 18.1 shows that `dfsR` not only marks all vertices connected to the start vertex, but also calls itself exactly once for each such vertex (to mark that vertex). An argument similar to the proof of Property 18.2 shows that a call to `GRAPHsearch` leads to exactly one call to `dfsR` for each graph vertex. In `dfsR`, we check every entry in the vertex's row in the adjacency matrix. In other words, we check each entry in the adjacency matrix precisely once.■

Property 18.4 *DFS of a graph represented with adjacency lists requires time proportional to* $V + E$.

Proof: From the argument just outlined, it follows that we call the recursive function precisely V times (hence the V term), and we examine each entry on each adjacency list (hence the E term).■

The primary implication of Properties 18.3 and 18.4 is that they establish the running time of DFS to be *linear* in the size of the data structure used to represent the graph. In most situations, we are also justified in thinking of the running time of DFS as being linear in the size of the graph, as well: If we have a dense graph (with the number of edges proportional to V^2) then either representation gives this result; if we have a sparse graph, then we assume use of an adjacency-lists representation. Indeed, we normally think of the running time of DFS as being linear in E. That statement is technically not true if we are using adjacency matrices for sparse graphs or for extremely sparse graphs with $E << V$ and most vertices isolated, but we can usually avoid the former situation and we can remove isolated vertices (see Exercise 17.33) in the latter situation.

As we shall see, these arguments all apply to any algorithm that has a few of the same essential features of DFS. If the algorithm marks each vertex and examines all the latter's incident vertices (and does any other work that takes time per vertex bounded by a constant), then these properties apply. More generally, if the time per vertex is bounded by some function $f(V, E)$, then the time for the search is guaranteed to be proportional to E

+ $Vf(V, E)$. In section 18.8, we see that DFS is one of a family of algorithms that has just these characteristics; in Chapters 19 through 22, we see that algorithms from this family serve as the basis for a substantial fraction of the code that we consider in this book.

Much of the graph-processing code that we examine is ADT-implementation code for some particular task, where we extend a basic search to compute structural information in other vertex-indexed arrays. In essence, we re-implement the search each time that we build an algorithm implementation around it. We adopt this approach because many of our algorithms are best understood as augmented graph-search functions. Typically, we uncover a graph's structure by searching it. We normally extend the search function with code that is executed when each vertex is marked, instead of working with a more generic search (for example, one that calls a specified function each time a vertex is visited), solely to keep the code compact and self-contained. Providing a more general ADT mechanism for clients to process all the vertices with a client-supplied function is a worthwhile exercise (see Exercises 18.12 and 18.13).

In Sections 18.5 and 18.6, we examine numerous graph-processing functions that are based on DFS in this way. In Sections 18.7 and 18.8, we look at other implementations of `search` and at some graph-processing functions that are based on them. Although we do not build this layer of abstraction into our code, we take care to identify the basic graph-search strategy underlying each algorithm that we develop. For example, we use the term *DFS function* to refer to any implementation that is based on the recursive DFS scheme. The simple-path–search function *Program* 17.11 is an example of a DFS function.

Many graph-processing functions are based on the use of vertex-indexed arrays. We typically include such arrays in one of three places in implementations:

• As global variables

• In the graph representation

• As function parameters, supplied by the client

We use the first alternative when gathering information about the search to learn facts about the structure of graphs that help us solve various problems of interest. An example of such an array is the `pre` array used in Programs 18.1 through 18.3. We use the second alternative when implementing preprocessing functions that compute information about the graph that enable efficient implementation of associated ADT functions. For example, we might maintain such an array to support an ADT function that returns the degree of any given vertex. When implementing ADT functions whose purpose is to compute a vertex-indexed array, we might use either the second or the third alternative, depending on

IPR2023-00124
KASPERSKY EX 1036 Page 126

the context.

Our convention in graph-search functions is to initialize all entries in vertex-indexed arrays to `-1`, and to set the entries corresponding to each vertex visited to nonnegative values in the search function. Any such array can play the role of the `pre` array (marking vertices as visited) in Programs 18.1 through 18.3. When a graph-search function is based on using or computing a vertex-indexed array, we use that array to mark vertices, rather than bothering with the `pre` array.

The specific outcome of a graph search depends not just on the nature of the search function, but also on the graph representation and even the order in which `GRAPHsearch` examines the vertices. For the examples and exercises in this book, we use the term *standard adjacency-lists DFS* to refer to the process of inserting a sequence of edges into a graph ADT implemented with an adjacency-lists representation (*Program* 17.6), then doing a DFS with Programs 18.3 and 18.2. For the adjacency-matrix representation, the order of edge insertion does not affect search dynamics, but we use the parallel term *standard adjacency-matrix DFS* to refer to the process of inserting a sequence of edges into a graph ADT implemented with an adjacency-matrix representation (*Program* 17.3), then doing a DFS with Programs 18.3 and 18.1.

Exercises

18.7 Show, in the style of *Figure* 18.5, a trace of the recursive function calls made for a standard adjacency-matrix DFS of the graph

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

18.8 Show, in the style of *Figure* 18.7, a trace of the recursive function calls made for a standard adjacency-lists DFS of the graph

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

18.9 Modify the adjacency-matrix graph ADT implementation in *Program* 17.3 to use a dummy vertex that is connected to all the other vertices. Then, provide a simplified DFS implementation that takes advantage of this change.

18.10 Do Exercise 18.9 for the adjacency-lists ADT implementation in *Program* 17.6.

● **18.11** There are 13! different permutations of the vertices in the graph depicted in *Figure* 18.8. How many of these permutations could specify the order in which vertices are

visited by *Program* 18.3?

18.12 Define an ADT function that calls a client-supplied function for each vertex in the graph. Provide implementations for the adjacency-matrix and the adjacency-lists graph representations.

18.13 Define an ADT function that calls a client-supplied function for each *edge* in the graph. Provide implementations for the adjacency-matrix and the adjacency-lists graph representations. (Such a function might be a reasonable alternative to `GRAPHedges` in an ADT design.)

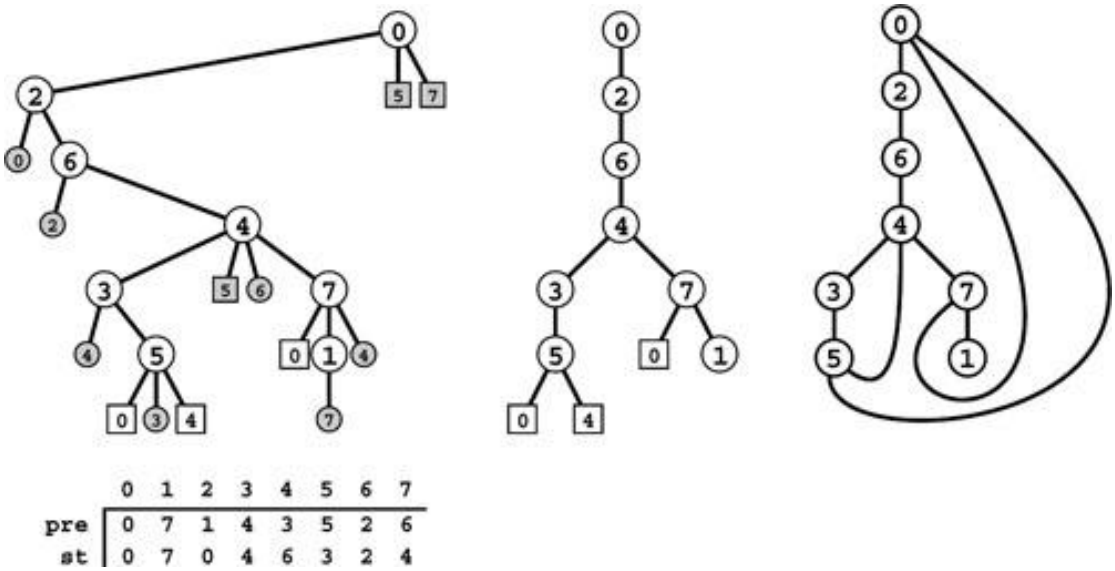
18.4 PROPERTIES OF DFS FORESTS

As noted in Section 18.2, the trees that describe the recursive structure of DFS function calls give us the key to understanding how DFS operates. In this section, we examine properties of the algorithm by examining properties of DFS trees.

The `pre` array in our DFS implementations is the *preorder numbering* of the internal nodes of the DFS tree. It is also easy to compute an explicit *parent-link representation* of the DFS tree: We initialize all entries of a vertex-indexed array `st` to `-1` and include the statement `st[e.w] = e.v` at the beginning of the recursive DFS function `dfsR` in Programs 18.1 and 18.2.

Figure 18.9 DFS Tree Representations *If we augment the DFS recursive-call tree to represent edges that are checked but not followed, we get a complete description of the DFS process* (*left*). *Each tree node has a child representing each of the nodes adjacent to it, in the order they were considered by the DFS, and a preorder traversal gives the same information as Figure 18.5: first we follow* `0-0`, *then* `0-2`, *then we skip* `2-0`, *then we follow* `2-6`, *then we skip* `6-2`, *then we follow* `6-4`, *then* `4-3`, *and so forth. The* `pre` *array specifies the order in which we visit tree vertices during this preorder walk, which is the same as the order in which we visit graph vertices in the DFS. The* `st` *array is a parent-link representation of the DFS recursive-call tree* (see *Figure* 18.6). *There are two links in the tree for every edge in the graph, one for each of the two times it encounters the edge. The first is to an unshaded node and either corresponds to making a recursive call (if it is to an internal node) or to skipping a recursive call because it goes to an ancestor for which a recursive call is in progress (if it is to an external node). The second is to a shaded external node and always corresponds to skipping a recursive call, either because it goes back to the parent (circles) or because it goes to a descendent of the parent for which a recursive call is in progress (squares). If we eliminate shaded nodes* (center), *then replace the external nodes with edges, we get another drawing of the graph* (right).

IPR2023-00124
KASPERSKY EX 1036 Page 128



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

If we add external nodes to the DFS tree to record the moments when we skipped recursive calls for vertices that had already been visited, we get the compact representation of the dynamics of DFS illustrated in *Figure* 18.9. This representation is worthy of careful study. The tree is a representation of the graph, with a vertex corresponding to each graph vertex and an edge corresponding to each graph edge. We can choose to show the two representations of the edge that we process (one in each direction), as shown in the left part of the figure, or just one representation of each edge, as shown in the center and right parts of the figure. The former is useful in understanding that the algorithm processes each and every edge; the latter is useful in understanding that the DFS tree is simply another graph representation. Traversing the internal nodes of the tree in preorder gives the vertices in the order in which DFS visits them; moreover, the order in which we visit the *edges* of the tree as we traverse it in preorder is the same as the order in which DFS examines the *edges* of the graph.

Indeed, the DFS tree in *Figure* 18.9 contains the same information as the trace in *Figure* 18.5 or the step-by-step illustration of Trémaux traversal in Figures 18.2 and 18.3. Edges to internal nodes represent edges (passages) to unvisited vertices (intersections), edges to external nodes represent occasions where DFS checks edges that lead to previously visited vertices (intersections), and shaded nodes represent edges to vertices for which a recursive DFS is in progress (when we open a door to a passage where the door at the other end is already open). With these interpretations, a preorder traversal of the tree tells the same story as that of the detailed maze-traversal scenario.

To study more intricate graph properties, we classify the edges in a graph according to the role that they play in the search. We have two distinct edge classes:

• Edges representing a recursive call (*tree* edges)

IPR2023-00124
KASPERSKY EX 1036 Page 129

• Edges connecting a vertex with an ancestor in its DFS tree that is not its parent (*back* edges)

When we study DFS trees for digraphs in Chapter 19, we examine other types of edges, not just to take the direction into account, but also because we can have edges that go across the tree, connecting nodes that are neither ancestors nor descendants in the tree.

Since there are two representations of each graph edge that each correspond to a link in the DFS tree, we divide the tree links into four classes. We refer to a link from `v` to `w` in a DFS tree that represents a tree edge as

• A *tree* link if `w` is unmarked

• A *parent* link if `st[w]` is `v` and a link from `v` to `w` that represents a back edge as

• A *back* link if `pre[w] < pre[v]`

• A *down* link if `pre[w] > pre[v]`

Each tree edge in the graph corresponds to a tree link and a parent link in the DFS tree, and each back edge in the graph corresponds to a back link and a down link in the DFS tree.

In the graphical DFS representation illustrated in *Figure* 18.9, tree links point to unshaded circles, parent links point to shaded circles, back links point to unshaded squares, and down links point to shaded squares. Each graph edge is represented either as one tree link and one parent link or as one down link and one back link. These classifications are tricky and worthy of study. For example, note that even though parent links and back links both point to ancestors in the tree, they are quite different: A parent link is just the other representation of a tree link, but a back link gives us new information about the structure of the graph.

The definitions just given provide sufficient information to distinguish among tree, parent, back, and down links in a DFS function implementation. Note that parent links and back links both have `pre[w] < pre[v]`, so we have also to know that `st[w]` is not `v` to know that `v-w` is a back link. *Figure* 18.10 depicts the result of printing out the classification of the DFS tree link for each graph edge as that edge is encountered during a sample DFS. It is yet another complete representation of the basic search process that is an intermediate step between *Figure* 18.5 and *Figure* 18.9.

Figure 18.10 DFS Trace (Tree Link Classifications) *This version of Figure 18.5 shows the classification of the DFS tree link corresponding to each graph edge representation encountered. Tree edges (which correspond to recursive calls) are*

IPR2023-00124
KASPERSKY EX 1036 Page 130

represented as tree links on the first encounter and parent links on the second encounter. Back edges are back links on the first encounter and down links on the second encounter.

```
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
```

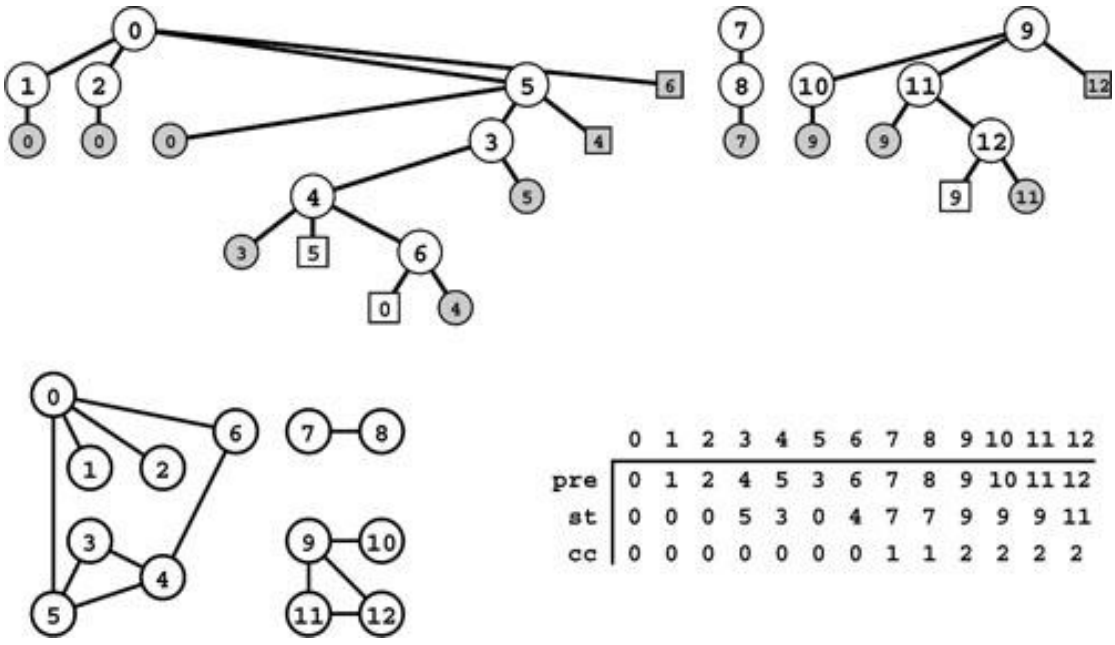
The four types of tree links correspond to the four different ways in which we treat edges during a DFS, as described (in maze-exploration terms) at the end of Section 18.1. A tree link corresponds to DFS encountering the first of the two representations of a tree edge, leading to a recursive call (to as-yet-unseen vertices); a parent link corresponds to DFS encountering the other representation of the tree edge (when going through the adjacency list on that first recursive call) and ignoring it. A back link corresponds to DFS encountering the first of the two representations of a back edge, which points to a vertex for which the recursive search function has not yet completed; a down link corresponds to DFS encountering a vertex for which the recursive search *has* completed at the time that the edge is encountered. In *Figure* 18.9, tree links and back links connect unshaded nodes, represent the first encounter with the corresponding edge, and constitute a representation of the graph; parent links and down links go to shaded nodes and represent the second encounter with the corresponding edge.

We have considered this tree representation of the dynamic characteristics of recursive DFS in detail not just because it provides a complete and compact description of both the

graph and the operation of the algorithm, but also because it gives us a basis for understanding numerous important graph-processing algorithms. In the remainder of this chapter, and in the next several chapters, we consider a number of examples of graph-processing problems that draw conclusions about a graph's structure from the DFS tree.

Search in a graph is a generalization of tree traversal. When invoked on a tree, DFS is precisely equivalent to recursive tree traversal; for graphs, using it corresponds to traversing a tree that spans the graph and that is discovered as the search proceeds. As we have seen, the particular tree traversed depends on how the graph is represented. DFS corresponds to preorder tree traversal. In Section 18.6, we examine the graph-searching analog to level-order tree traversal and explore its relationship to DFS; in section 18.7, we examine a general schema that encompasses any traversal method.

Figure 18.11 DFS Forest *The DFS forest at the top represents a DFS of an adjacency-matrix representation of the graph at the bottom right. The graph has three connected components, so the forest has three trees. The* `pre` *array is a preorder numbering of the nodes in the tree (the order in which they are examined by the DFS) and the* `st` *array is a parent-link representation of the forest. The* `cc` *array associates each vertex with a connected-component index (see Program 18.4). As in Figure 18.9, edges to circles are tree edges; edges that go to squares are back edges; and shaded nodes indicate that the incident edge was encountered earlier in the search, in the other direction.*



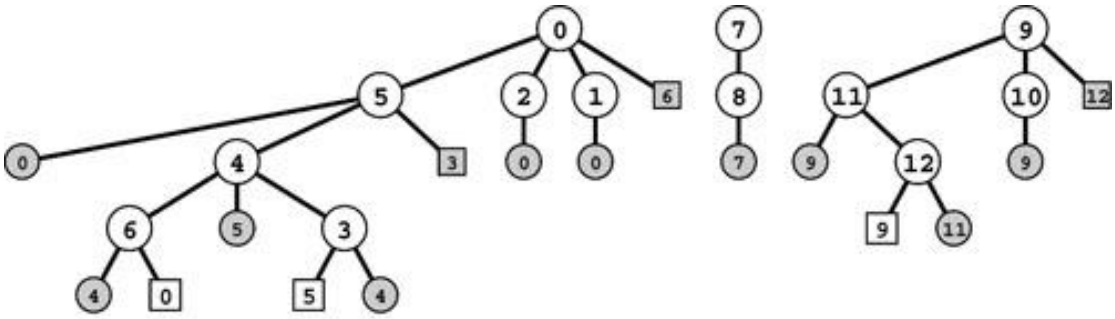
When traversing graphs, we have been assigning preorder numbers to the vertices in the order that we start processing them (just after entering the recursive search function). We can also assign *postorder numbers* to vertices, in the order that we *finish* processing them (just before returning from the recursive search function). When processing a graph, we do more than simply traverse the vertices—as we shall see, the preorder and

postorder numbering give us knowledge about global graph properties that helps us to accomplish the task at hand. Preorder numbering suffices for the algorithms that we consider in this chapter, but we use postorder numbering in later chapters.

We describe the dynamics of DFS for a general undirected graph with a *DFS forest* that has one DFS tree for each connected component. An example of a DFS forest is illustrated in *Figure* 18.11.

With an adjacency-lists representation, we visit the edges connected to each vertex in an order different from that for the adjacency-matrix representation, so we get a different DFS forest, as illustrated in *Figure* 18.12. DFS trees and forests are graph representations that describe not only the dynamics of DFS but also the internal representation of the graph. For example, by reading the children of any node in *Figure* 18.12 from left to right, we see the order in which they appear on the adjacency list of the vertex corresponding to that node. We can have many different DFS forests for the same graph—each ordering of nodes on adjacency lists leads to a different forest.

Figure 18.12 Another DFS Forest *This forest describes depth-first search of the same graph as Figure* 18.9*, but using Program* 18.2*, so the search order is different because it is determined by the order that nodes appear in adjacency lists. Indeed, the forest itself tells us that order: it is the order in which children are listed for each node in the tree. For instance, the nodes on* `0`*'s adjacency list were found in the order* `5 2 1 6`*, the nodes on* `4`*'s list are in the order* `6 5 3`*, and so forth. As before, all vertices and edges in the graph are examined during the search, in a manner that is precisely described by a preorder walk of the tree. The* `pre` *and* `st` *arrays depend upon the graph representation and the search dynamics and are different from Figure* 18.9*, but the array* `cc` *depends on graph properties and is the same.*



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

Details of the structure of a particular forest inform our understanding of how DFS operates for particular graphs, but most of the important DFS properties that we consider de-

pend on graph properties that are independent of the structure of the forest. For example, the forests in Figures 18.11 and 18.12 both have three trees (as would any other DFS forest for the same graph) because they are just different representations of the same graph, which has three connected components. Indeed, a direct consequence of the basic proof that DFS visits all the nodes and edges of a graph (see Properties 18.2 through 18.4) is that the number of connected components in the graph is equal to the number of trees in the DFS forest. This example illustrates the basis for our use of graph search throughout the book: A broad variety of graph ADT function implementations are based on learning graph properties by processing a particular graph representation (a forest corresponding to the search).

Potentially, we could analyze DFS tree structures with the goal of improving algorithm performance. For example, should we attempt to speed up an algorithm by rearranging the adjacency lists before starting the search? For many of the important classical DFS-based algorithms, the answer to this question is no, because they are optimal—their worst-case running time depends on *neither* the graph structure *nor* the order in which edges appear on the adjacency lists (they essentially process each edge exactly once). Still, DFS forests have a characteristic structure that is worth understanding because it distinguishes them from other fundamental search schema that we consider later in this chapter.

Figure 18.13 shows a DFS tree for a larger graph that illustrates the basic characteristics of DFS search dynamics. The tree is tall and thin, and it demonstrates several characteristics of the graph being searched and of the DFS process:

• There exists at least one long `path` that connects a substantial fraction of the nodes.

• During the search, most vertices have at least one adjacent vertex that we have not yet seen.

• We rarely make more than one recursive call from any node.

• The depth of the recursion is proportional to the number of vertices in the graph.

This behavior is typical for DFS, though these characteristics are not guaranteed for all graphs. Verifying facts of this kind for graph models of interest and various types of graphs that arise in practice requires detailed study. Still, this example gives an intuitive feel for DFS-based algorithms that is often borne out in practice. *Figure* 18.13 and similar figures for other graph-search algorithms (see Figures 18.24 and 18.29) help us understand differences in their behavior.

Exercises

18.14 Draw the DFS forest that results from a standard adjacency-matrix DFS of the graph

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

18.15 Draw the DFS forest that results from a standard adjacency-lists DFS of the graph

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

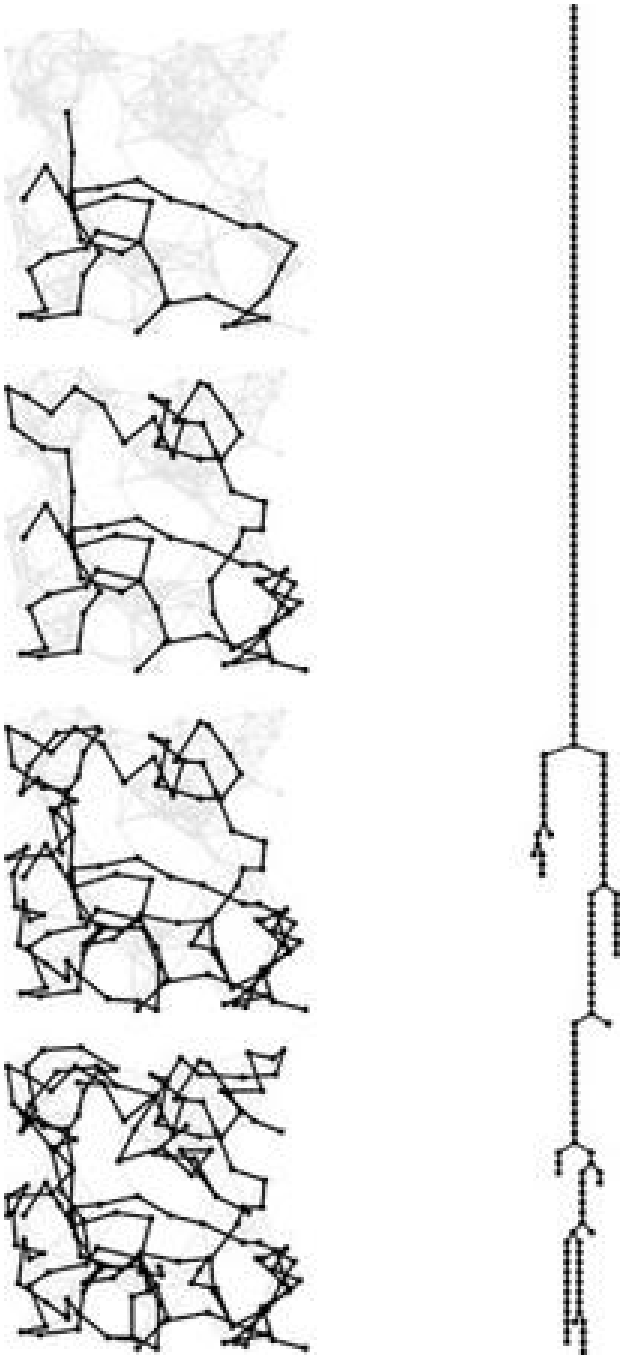
18.16 Write a DFS trace program for the adjacency-lists representation to produce output that classifies each of the two representations of each graph edge as corresponding to a tree, parent, back, or down link in the DFS tree, in the style of *Figure* 18.10.

○ **18.17** Write a program that computes a parent-link representation of the full DFS tree (including the external nodes), using an array of E integers between `0` and $V - 1$. *Hint*: The first V entries in the array should be the same as those in the `st` array described in the text.

○ **18.18** Instrument our DFS implementation for the adjacency-lists representation (*Program* 18.2) to print out the height of the DFS tree, the number of back edges, and the percentage of edges processed to see every vertex.

Figure 18.13 Depth-First Search *This figure illustrates the progress of DFS in a random Euclidean near-neighbor graph (left). The figures show the DFS tree vertices and edges in the graph as the search progresses through 1/4, 1/2, 3/4, and all of the vertices (top to bottom). The DFS tree (tree edges only) is shown at the right. As is evident from this example, the search tree for DFS tends to be quite tall and thin for this type of graph (as it is for many other types of graphs commonly encountered in practice). We normally find a vertex nearby that we have not seen before.*

IPR2023-00124
KASPERSKY EX 1036 Page 135



18.19 Run experiments to determine empirically the average values of the quantities described in Exercise 18.18 for graphs of various sizes, drawn from various graph models (see Exercises 17.63–76).

18.20 Write a function that builds a graph by inserting edges from a given array into an initially empty graph, in random order. Using this function with an adjacency-lists implementation of the graph ADT, run experiments to determine empirically properties of the *distribution* of the quantities described in Exercise 18.18 for all the adjacency-lists representations of large sample graphs of various sizes, drawn from various graph models (see Exercises 17.63–76).

IPR2023-00124
KASPERSKY EX 1036 Page 136

18.5 DFS ALGORITHMS

Regardless of the graph structure or the representation, any DFS forest allows us to identify edges as tree or back edges and gives us dependable insights into graph structure that allow us to use DFS as a basis for solving numerous graph-processing problems. We have already seen, in Section 17.7, basic examples related to finding paths. In this section, we consider DFS-based ADT function implementations for these and other typical problems; in the remainder of this chapter and in the next several chapters, we look at numerous solutions to much more difficult problems.

Cycle detection Does a given graph have any cycles? (Is the graph a forest?) This problem is easy to solve with DFS because any back edge in a DFS tree belongs to a cycle consisting of the edge plus the tree `path` connecting the two nodes (see *Figure* 18.9). Thus, we can use DFS immediately to check for cycles: A graph is acyclic if and only if we encounter no back (or down!) edges during a DFS. For example, to test this condition in *Program* 18.1, we simply add an `else` clause to the `if` statement to test whether `t` is equal to `v`. If it is, we have just encountered the parent link `w-v` (the second representation of the edge `v-w` that led us to `w`). If it is not, `w-t` completes a cycle with the edges from `t` down to `w` in the DFS tree. Moreover, we do not need to examine all the edges: We know that we must find a cycle or finish the search without finding one before examining V edges, because any graph with V or more edges must have a cycle. Thus, we can test whether a graph is acyclic in time proportional to V with the adjacency-lists representation, although we may need time proportional to V^2 (to find the edges) with the adjacency-matrix representation.

Program 18.4 Graph connectivity (adjacency lists)

The DFS function `GRAPHcc` computes, in linear time, the number of connected components in a graph and stores a component index associated with each vertex in the vertex-indexed array `G->cc` in the graph representation. (Since it does not need structural information, the recursive function uses a vertex as its second argument instead of an edge as in *Program* 18.2.) After calling `GRAPHcc`, clients can test whether any pair of vertices are connected in constant time (`GRAPHconnect`).

IPR2023-00124
KASPERSKY EX 1036 Page 137

```
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]; }
```

Simple path Given two vertices, is there a `path` in the graph that connects them? We saw in Section 17.7 that a DFS function that can solve this problem in linear time is easy to devise.

Simple connectivity As discussed in Section 18.3, we determine whether or not a graph is connected whenever we use DFS, in linear time. Indeed, our basic graph-search strategy is based upon calling a search function for each connected component. In a DFS, the graph is connected if and only if the graph-search function calls the recursive DFS function just once (see *Program* 18.3). The number of connected components in the graph is precisely the number of times that the recursive function is called from `GRAPHsearch`, so we can find the number of connected components by simply keeping track of the number of such calls.

More generally, *Program* 18.4 illustrates a DFS-based ADT implementation for the adjacency-lists representation that supports constant-time connectivity queries after a linear-time preprocessing step. Each tree in the DFS forest identifies a connected component, so we can arrange to decide quickly whether two vertices are in the same component by including a vertex-indexed array in the graph representation, to be filled in by a DFS and accessed for connectivity queries. In the recursive DFS function, we assign the current value of the component counter to the entry corresponding to each vertex visited. Then, we know that two vertices are in the same component if and only if their entries in this array are equal. Again, note that this array reflects structural properties of the graph, rather than artifacts of the graph representation or of the search dynamics.

Program 18.4 typifies the basic approach that we shall use in solving numerous graph-processing problems. We invest preprocessing time and extend the graph ADT to compute structural graph properties that help us to provide efficient implementations of important ADT functions. In this case, we preprocess with a (linear-time) DFS and keep an array cc that allows us to answer connectivity queries in constant time. For other graph-processing problems, we might use more space, preprocessing time, or query time. As usual, our focus is on minimizing such costs, although doing so is often challenging. For example, much of Chapter 19 is devoted to solving the connectivity problem for digraphs, where achieving the same performance characteristics as *Program* 18.4 is an elusive goal.

How does the DFS-based solution for graph connectivity in *Program* 18.4 compare with the union-find approach that we considered in Chapter 1 for the problem of determining whether a graph is connected, given an edge list? In theory, DFS is faster than union-find because it provides a constant-time guarantee, which union- find does not; in practice, this difference is negligible, and union-find is faster because it does not have to build a full representation of the graph. More important, union-find is an online algorithm (we can check whether two vertices are connected in near-constant time at any point), whereas the DFS solution preprocesses the graph to answer connectivity queries in constant time. Therefore, for example, we prefer union-find when

Program 18.5 Two-way Euler tour

This DFS function for the adjacency-matrix representation prints each edge twice, once in each orientation, in a two-way–Euler-tour order. We go back and forth on back edges and ignore down edges (*see text*).

```
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);
```

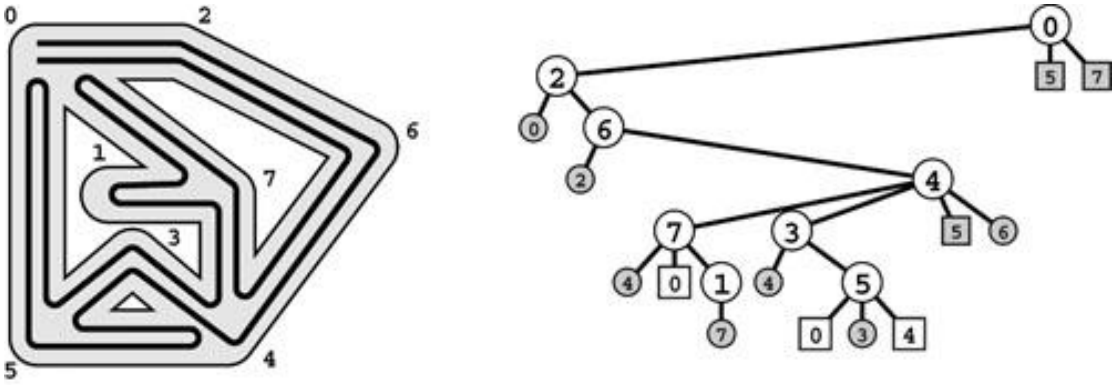
```
        else printf ("\n");
      }
```

determining connectivity is our only task or when we have a large number of queries intermixed with edge insertions but may find the DFS solution more appropriate for use in a graph ADT because it makes efficient use of existing infrastructure. Neither approach handles efficiently huge numbers of intermixed edge insertions, edge deletions, and connectivity queries; both require a separate DFS to compute the path. These considerations illustrate the complications that we face when analyzing graph algorithms—we explore them in detail in Section 18.9.

Two-way Euler tour *Program* 18.5 uses DFS to solve the problem of finding a `path` that uses all the edges in a graph exactly twice—once in each direction (see Section 17.7). The `path` corresponds to a Trémaux exploration in which we take our string with us everywhere that we go, check for the string instead of using lights (so we have to go down the passages that lead to intersections that we have already visited), and first arrange to go back and forth on each back link (the first time that we encounter each back edge), then ignore down links (the second time that we encounter each back edge). We might also choose to ignore the back links (first encounter) and to go back and forth on down links (second encounter) (see Exercise 18.23).

Figure 18.14 A Two-Way Euler Tour *Depth-first search gives us a way to explore any maze, traversing both passages in each direction. We modify Trémaux exploration to take the string with us wherever we go and take a back-and-forth trip on passages without any string in them that go to intersections that we have already visited. This figure shows a different traversal order than shown in Figures 18.2 and 18.3, primarily so that we can draw the tour without crossing itself. This ordering might result, for example, if the edges were processed in some different order when building an adjacency-lists representation of the graph, or, we might explicitly modify DFS to take the geometric placement of the nodes into account (see Exercise 18.24). Moving along the lower track leading out of* `0`*, we move from* `0` *to* `2` *to* `6` *to* `4` *to* `7`*, then take a trip from* `7` *to* `0` *and back because* `pre[0]` *is less than* `pre[7]`*. Then we go to* `1`*, back to* `7`*, back to* `4`*, to* `3`*, to* `5`*, from* `5` *to* `0` *and back, from* `5` *to* `4` *and back, back to* `3`*, back to* `4`*, back to* `6`*, back to* `2`*, and back to* `0`*. This* `path` *may be obtained by a recursive pre- and postorder walk of the DFS tree (ignoring the shaded vertices that represent the second time we encounter the edges) where we print out the vertex name, recursively visit the subtrees, then print out the vertex name again.*

IPR2023-00124
KASPERSKY EX 1036 Page 140



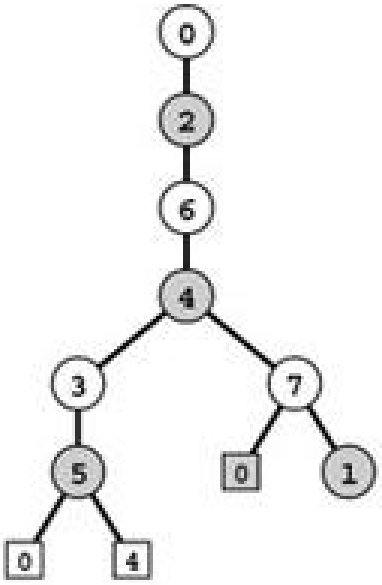
Spanning tree Given a connected graph with V vertices, find a set of $V - 1$ edges that connects the vertices. DFS solves this problem because any DFS tree is a spanning tree. Our DFS implementations make precisely $V - 1$ recursive calls for a connected graph, one for each edge on a spanning tree, and can be easily instrumented to produce a parent-link representation of the tree, as discussed at the beginning of Section 18.4. If the graph has C connected components, we get a spanning forest with $V - C$ edges. In an ADT function implementation, we might choose to add the parent-link array to the graph representation, in the style of *Program* 18.4, or to compute it in a client-supplied array.

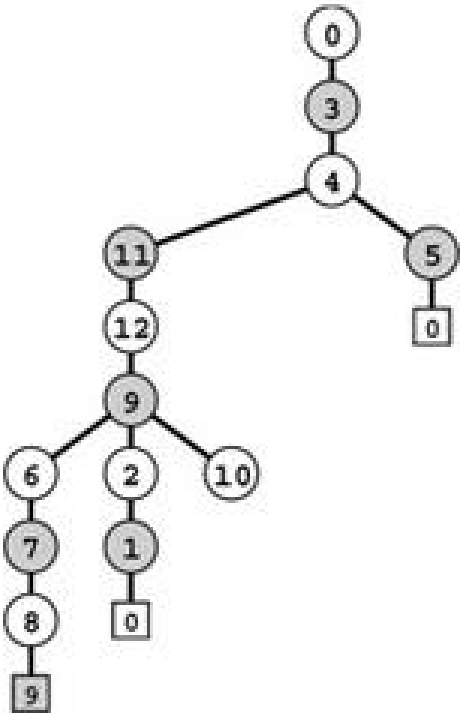
Vertex search How many vertices are in the same connected component as a given vertex? We can solve this problem easily by starting a DFS at the vertex and counting the number of vertices marked. In a dense graph, we can speed up the process considerably by stopping the DFS after we have marked V vertices—at that point, we know that no edge will take us to a vertex that we have not yet seen, so we will be ignoring all the rest of the edges. This improvement is likely to allow us to visit all vertices in time proportional to $V \log V$, not E (see section 18.8).

Two-colorability, bipartiteness, odd cycle Is there a way to assign one of two colors to each vertex of a graph such that no edge connects two vertices of the same color? Is a given graph bipartite (see Section 17.1)? Does a given graph have a cycle of odd length? These three problems are all equivalent: The first two are different nomenclature for the same problem; any graph with an odd cycle is clearly not two-colorable, and *Program* 18.6 demonstrates that any graph that is free of odd cycles can be two-colored. The program is a DFS-based ADT function implementation that tests whether a graph is bipartite, two-colorable, and free of odd cycles. The recursive function is an outline for a proof by induction that the program two-colors any graph with no odd cycles (or finds an odd cycle as evidence that a graph that is not free of odd cycles cannot be two-colored). To two-color a graph with a given color assigned to a vertex v, two-color the remaining graph, assigning the other color to each vertex adjacent to v. This process is equivalent to

assigning alternate colors on levels as we proceed down the DFS tree, checking back edges for consistency in the coloring, as illustrated in *Figure* 18.15. Any back edge connecting two vertices of the same color is evidence of an odd cycle.

Figure 18.15 Two-Coloring a DFS Tree *To two-color a graph, we alternate colors as we move down the DFS tree, then check the back edges for inconsistencies. In the tree at the top, a DFS tree for the sample graph illustrated in Figure* 18.9*, the back edges* `5-4` *and* `7-0` *prove that the graph is not two-colorable because of the odd-length cycles* `4-3-5-4` *and* `0-2-6-4-7-0`*, respectively. In the tree at the bottom, a DFS tree for the bipartite graph illustrated in Figure* 17.5*, there are no such inconsistencies, and the indicated shading is a two-coloring of the graph.*





These basic examples illustrate ways in which DFS can give us insight into the structure of a graph. They illustrate that we can learn various important graph properties in a single linear-time sweep through the graph, where we examine every edge twice, once in each direction. Next, we consider an example that shows the utility of DFS in discovering more intricate details about the graph structure, still in linear time.

Exercises

○ **18.21** Give an ADT function implementation for the adjacency-lists graph representation that, in time proportional to V, finds a cycle and prints it, or reports that none exists.

IPR2023-00124
KASPERSKY EX 1036 Page 143

18.22 Describe a family of graphs with V vertices for which a standard adjacency-matrix DFS requires time proportional to V^2 for cycle detection.

▷**18.23** Specify a modification to *Program* 18.5 that will produce a two-way Euler tour that does the back-and-forth traversal on down edges instead of back edges.

● **18.24** Modify *Program* 18.5 such that it always produces a two-way Euler tour that, like the one in *Figure* 18.14, can be drawn such that it does not cross itself at any vertex. For example, if the search in *Figure* 18.14 were to take the edge `4-3` before the edge `4-7`, then the tour would have to cross itself; your task is to ensure that the algorithm avoids such situations.

18.25 Develop a version of *Program* 18.5 for the adjacency-lists graph representation (*Program* 17.6) that sorts the edges of a graph in order of a two-way Euler tour. Your program should return a link to a circular linked list of nodes that corresponds to a two-way Euler tour.

Program 18.6 Two-colorability

This DFS function assigns the values `0` or `1` to the vertex-indexed array `G->color` and indicates in the return value whether or not it was able to do the assignment such that, for each graph edge `v-w`, `G->color[v]` and `G->color[w]` are different.

```
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;
        }
```

18.26 Modify your solution to Exercise 18.25 such that you can use it for huge graphs, where you might not have the space to make a copy of the list nodes corresponding to each edge. That is, use the list nodes that were allocated to build the graph, and destroy the original adjacency-lists graph representation.

18.27 Prove that a graph is two-colorable if and only if it contains no odd cycle. *Hint*: Prove by induction that *Program* 18.6 determines whether or not any given graph is two-colorable.

18.28 Explain why the approach taken in *Program* 18.6 does not generalize to give an efficient method for determining whether a graph is three-colorable.

18.29 Most graphs are not two-colorable, and DFS tends to discover that fact quickly. Run empirical tests to study the number of edges examined by *Program* 18.6, for graphs of various sizes, drawn from various graph models (see Exercises 17.63–76).

18.30 Prove that every connected graph has a vertex whose removal will not disconnect the graph, and write a DFS function that finds such a vertex. *Hint*: Consider the leaves of the DFS tree.

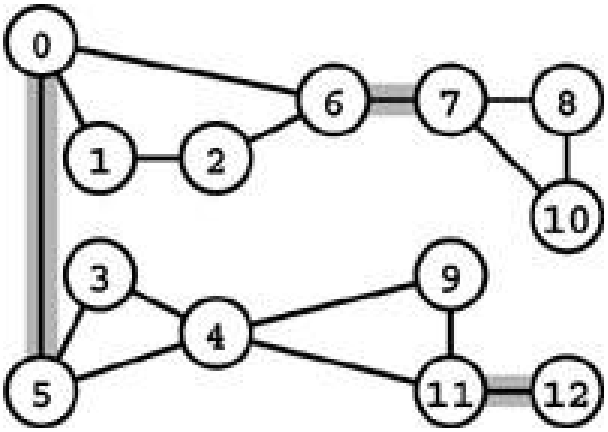
18.31 Prove that every graph with more than one vertex has at least two vertices whose removal will not increase the number of connected components.

18.6 SEPARABILITY AND BICONNECTIVITY

To illustrate the power of DFS as the basis for graph-processing algorithms, we turn to problems related to generalized notions of connectivity in graphs. We study questions such as the following: Given two vertices, are there two different paths connecting them?

Figure 18.16 An Edge-Separable Graph *This graph is not edge connected. The edges* `0-5`, `6-7`, *and* `11-12` *(shaded) are separating edges (bridges). The graph has* `4` *edge-connected components: one comprising vertices* `0, 1, 2`, *and* `6`; *another comprising vertices* `3, 4, 9`, *and* `11`; *another comprising vertices* `7, 8`, *and* `10`; *and the single vertex* `12`.

IPR2023-00124
KASPERSKY EX 1036 Page 145



If it is important that a graph *be* connected in some situation, it might also be important that it *stay* connected when an edge or a vertex is removed. That is, we may want to have more than one route between each pair of vertices in a graph, so as to handle possible failures. For example, we can fly from New York to San Francisco even if Chicago is snowed in by going through Denver instead. Or, we might imagine a wartime situation where we want to arrange our railroad network such that an enemy must destroy at least two stations to cut our rail lines. Similarly, we might expect the main communications lines in an integrated circuit or a communications network to be connected such that the rest of the circuit still can function if one wire is broken or one link is down.

These examples illustrate two distinct concepts: In the circuit and in the communications network, we are interested in staying connected if an *edge* is removed; in the air or train routes, we are interested in staying connected if a *vertex* is removed. We begin by examining the former in detail.

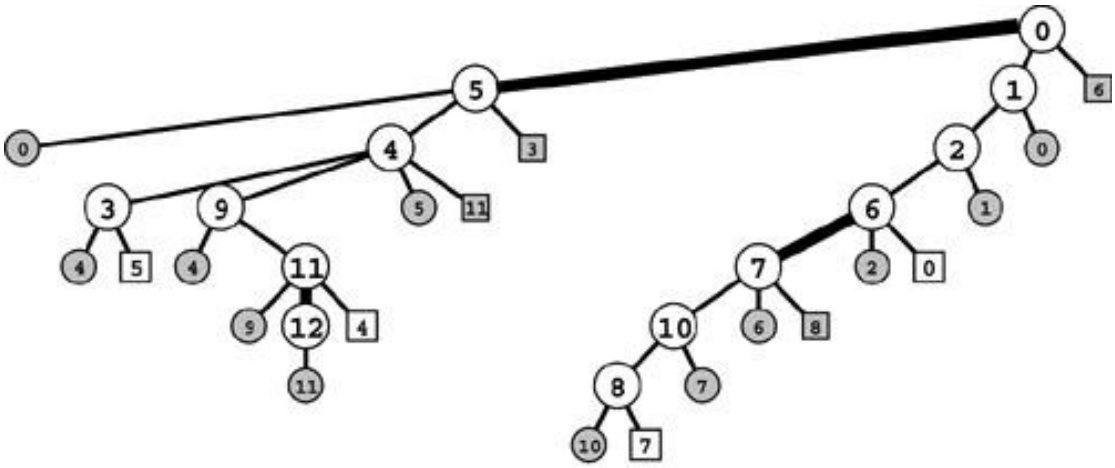
Definition 18.5 A bridge *in a graph is an edge that, if removed, would separate a connected graph into two disjoint subgraphs. A graph that has no bridges is said to be* **edge-connected**.

When we speak of *removing* an edge, we mean to delete that edge from the set of edges that define the graph, even when that action might leave one or both of the edge's vertices isolated. An edge-connected graph remains connected when we remove any single edge. In some contexts, it is more natural to emphasize our ability to disconnect the graph rather than the graph's ability to stay connected, so we freely use alternate terminology that provides this emphasis: We refer to a graph that is not edge-connected as an *edge-separable* graph, and we call bridges *separation edges*. If we remove all the bridges in an edge-separable graph, we divide it into *edge-connected components* or *bridge-connected components*: maximal subgraphs with no bridges. *Figure* 18.16 is a small example that illustrates these concepts.

Figure 18.17 DFS Tree for Finding Bridges *Nodes* `5`, `7`, *and* `12` *in this DFS*

IPR2023-00124
KASPERSKY EX 1036 Page 146

tree for the graph in Figure 18.16 all have the property that no back edge connects a descendant with an ancestor, and no other nodes have that property. Therefore, as indicated, breaking the edge between one of these nodes and its parent would disconnect the subtree rooted at that node from the rest of the graph. That is, the edges `0-5`, `11-12`, *and* `6-7` *are bridges. We use the vertex-indexed array* `low` *to keep track of the lowest preorder number referenced by any back edge in the subtree rooted at the vertex. For example, the value of* `low[9]` *is* `2` *because one of the back edges in the subtree rooted at* `9` *points to* `4` *(the vertex with preorder number* `2`*), and no other back edge points higher in the tree. Nodes* `5`, `7`, *and* `12` *are the ones for which the* `low` *value is equal to the* `pre` *value.*



Finding the bridges in a graph seems, at first blush, to be a nontrivial graph-processing problem, but it actually is an application of DFS where we can exploit basic properties of the DFS tree that we have already noted. Specifically, back edges cannot be bridges because we know that the two nodes they connect are also connected by a `path` in the DFS tree. Moreover, we can add a simple test to our recursive DFS function to test whether or not tree edges are bridges. The basic idea, stated formally next, is illustrated in *Figure 18.17*.

Property 18.5 *In any DFS tree, a tree edge* `v-w` *is a bridge if and only if there are no back edges that connect a descendant of* `w` *to an ancestor of* `w`.

Proof: If there is such an edge, `v-w` cannot be a bridge. Conversely, if `v-w` is not a bridge, then there has to be some `path` from `w` to `v` in the graph other than `w-v` itself. Every such `path` has to have some such edge.■

Program 18.7 Edge connectivity (adjacency lists)

This recursive DFS function prints and counts the bridges in a graph. It assumes that *Program* 18.3 is augmented with a counter `bnct` and a vertex-indexed array `low` that are initialized in the same way as `cnt` and `pre`, respectively. The `low` array keeps track of the lowest preorder number that can be reached from each vertex by a sequence of tree edges followed by one back edge.

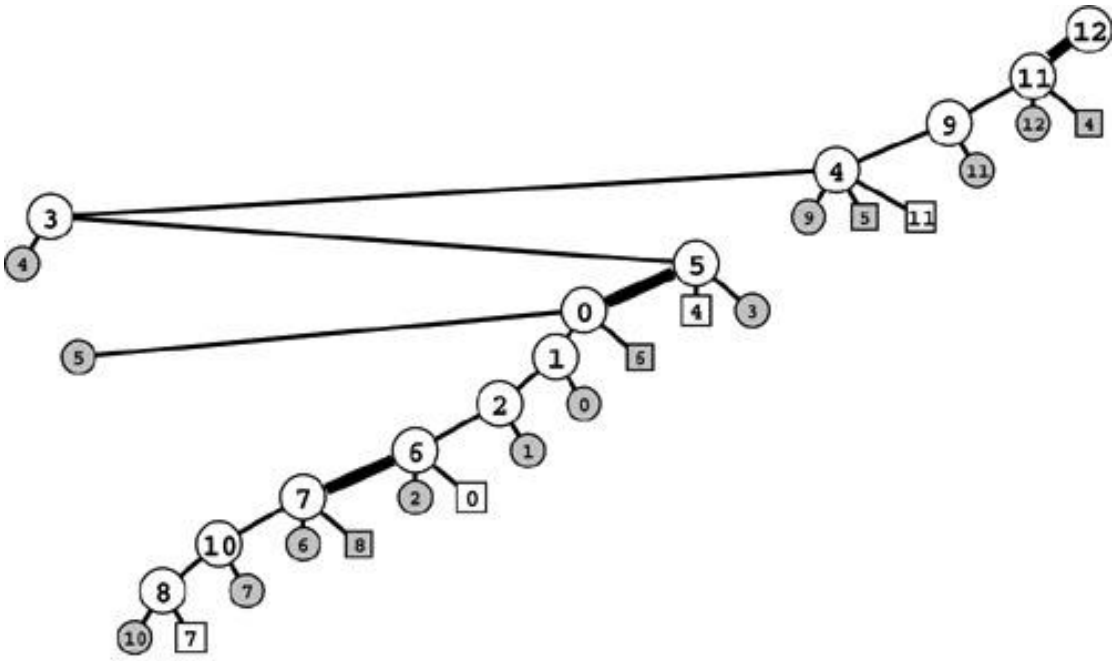
```
void bridgeR (Graph G, Edge e)
  { link t; int v, w = e.w;
    pre[w] = cnt++; low[w] = pre[w];
    for (t = G->adj[w]; t != NULL; t = t->next)
      if (pre[v = t->v] == -1)
        {
          bridgeR (G, EDGE (w, v));
          if (low[w] > low[v]) low[w] = low[v];
          if (low[v] == pre[v])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

Asserting this property is equivalent to saying that the only link in the subtree rooted at `w` that points to a node not in the subtree is the parent link from `w` back to `v`. This condition holds if and only if every `path` connecting any of the nodes in `w`'s subtree to any node that is not in `w`'s subtree includes `v-w`. In other words, removing `v-w` would disconnect from the rest of the graph the subgraph corresponding to `w`'s subtree.

Program 18.7 shows how we can augment DFS to identify bridges in a graph, using Property 18.5. For every vertex `v`, we use the recursive function to compute the lowest preorder number that can be reached by a sequence of zero or more tree edges followed by a single back edge from any node in the subtree rooted at `v`. If the computed number is equal to `v`'s preorder number, then there is no edge connecting a descendant with an ancestor, and we have identified a bridge. The computation for each vertex is straightforward: We proceed through the adjacency list, keeping track of the minimum of the numbers that we can reach by following each edge. For tree edges, we do the computation re-

cursively; for back edges, we use the preorder number of the adjacent vertex. If the call to the recursive function for an edge `w-v` does not uncover a `path` to a node with a preorder number less than `v`'s preorder number, then `w-v` is a bridge.

Figure 18.18 Another DFS Tree for Finding Bridges *This diagram shows a different DFS tree for than the one in Figure 18.17 for the graph in Figure 18.16, where we starting the search at a different node. Although we visit the nodes and edges in a completely different order, we still find the same bridges (of course). In this tree,* `0`, `7`, *and* `11` *are the ones for which the* `low` *value is equal to the* `pre` *value, so the edges connecting each of them to their parents* (`12-11`, `5-0`, *and* `6-7`, *respectively) are bridges.*



	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

Property 18.6 *We can find a graph's bridges in linear time.*

Proof: *Program* 18.7 is a minor modification to DFS that involves adding a few constant-time tests, so it follows directly from Properties 18.3 and 18.4 that finding the bridges in a graph requires time proportional to V^2 for the adjacency-matrix representation and to $V + E$ for the adjacency-lists representation.■

In *Program* 18.7, we use DFS to discover properties of the graph. The graph representation certainly affects the order of the search, but it does not affect the results because bridges are a characteristic of the graph rather than of the way that we choose to represent or search the graph. As usual, any DFS tree is simply another representation of the graph, so all DFS trees have the same connectivity properties. The correctness of the algo-

rithm depends on this fundamental fact. For example, *Figure* 18.18 illustrates a different search of the graph, starting from a different vertex, that (of course) finds the same bridges. Despite Property 18.6, when we examine different DFS trees for the same graph, we see that some search costs may depend not just on properties of the graph, but also on properties of the DFS tree. For example, the amount of space needed for the stack to support the recursive calls is larger for the example in *Figure* 18.18 than for the example in *Figure* 18.17.

As we did for regular connectivity in *Program* 18.4, wemaywish to use *Program* 18.7 to build an ADT function for testing whether a graph is edge-connected or to count the number of edge-connected components. If desired, we can proceed as for *Program* 18.4 to create ADT functions that gives clients the ability to call a linear-time preprocessing function, then respond in constant time to queries that ask whether two vertices are in the same edge-connected component (see Exercise 18.35).

We conclude this section by considering other generalizations of connectivity, including the problem of determining which *vertices* are critical to keeping a graph connected. By including this material here, we keep in one place the basic background material for the more complex algorithms that we consider in Chapter 22. If you are new to connectivity problems, you may wish to skip to section 18.7 and return here when you study Chapter 22.

When we speak of it removing a vertex, we also mean that we remove all its incident edges. As illustrated in *Figure* 18.19, removing either of the vertices on a bridge would disconnect a graph (unless the bridge were the only edge incident on one or both of the vertices), but there are also other vertices, not associated with bridges, that have the same property.

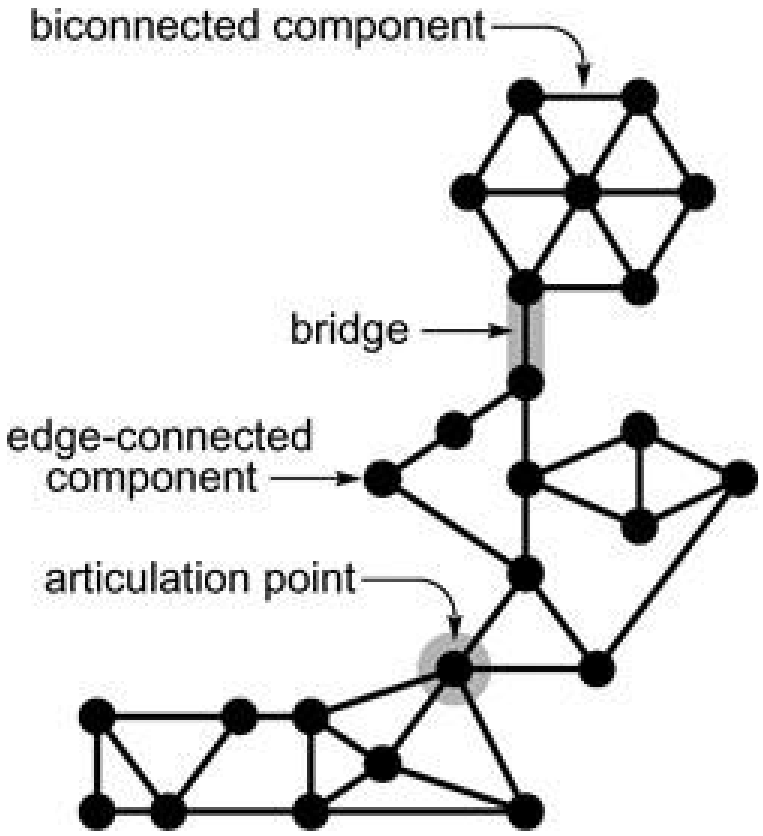
Definition 18.6 *An* **articulation point** *in a graph is an vertex that, if removed, would separate a connected graph into at least two disjoint subgraphs.*

We also refer to articulation points as *separation vertices* or *cut vertices*. We might use the term “vertex connected” to describe a graph that has no separation vertices, but we use different terminology based on a related characterization that turns out to be equivalent.

Definition 18.7 *A graph is said to be* **biconnected** *if every pair of vertices is connected by two disjoint paths.*

Figure 18.19 Graph Separability Terminology *This graph has two edge-connected components and one bridge. The edge-connected component above the bridge is also biconnected; the one below the bridge consists of two biconnected*

components that are joined at an articulation point.



The requirement that the paths be *disjoint* is what distinguishes biconnectivity from edge connectivity. An alternate definition of edge connectivity is that every pair of vertices is connected by two *edge-disjoint* paths—these paths can have a vertex (but no edge) in common. Biconnectivity is a stronger condition: An edge-connected graph remains connected if we remove any edge, but a biconnected graph remains connected if we remove any vertex (and all that vertex's incident edges). Every biconnected graph is edge-connected, but an edge-connected graph need not be biconnected. We also use the term *separable* to refer to graphs that are not biconnected, because they can be separated into two pieces by removal of just one vertex. The separation vertices are the key to biconnectivity.

Property 18.7 *A graph is biconnected if and only if it has no separation vertices (articulation points).*

Proof: Assume that a graph has a separation vertex. Let *s* and *t* be vertices that would be in two different pieces if the separation vertex were removed. All paths between *s* and *t* must contain the separation vertex, therefore the graph is not biconnected. The proof in the other direction is more difficult and is a worthwhile exercise for the mathematically inclined reader (see Exercise 18.39).■

We have seen that we can partition the edges of a graph that is not connected into a set of

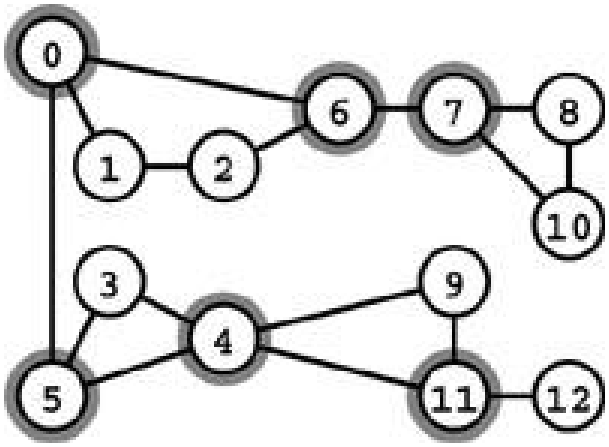
IPR2023-00124
KASPERSKY EX 1036 Page 151

connected subgraphs, and that we can partition the edges of a graph that is not edge-connected into a set of bridges and edge-connected subgraphs (which are connected by bridges). Similarly, we can divide any graph that is not biconnected into a set of bridges and *biconnected components*, which are each biconnected subgraphs. The biconnected components and bridges are not a proper partition of the graph because articulation points may appear on multiple biconnected components (see, for example, *Figure* 18.20). The biconnected components are connected at articulation points, perhaps by bridges.

A connected component of a graph has the property that there exists a `path` between any two vertices in the graph. Analogously, a biconnected component has the property that there exist *two* disjoint paths between any pair of vertices.

We can use the same DFS-based approach that we used in *Program* 18.7 to determine whether or not a graph is biconnected and to identify the articulation points. We omit the code because it is very similar to *Program* 18.7, with an extra test to check whether the root of the DFS tree is an articulation point (see Exercise 18.42). Developing code to print out the biconnected components is also a worthwhile exercise that is only slightly more difficult than the corresponding code for edge connectivity (see Exercise 18.43).

Figure 18.20 Articulation Points (Separation Vertices) *This graph is not biconnected. The vertices* `0, 4, 5, 6, 7,` *and* `11` *(shaded) are articulation points. The graph has five biconnected components: one comprising edges* `4-9, 9-11,` *and* `4-11`*; another comprising edges* `7-8, 8-10,` *and* `7-10`*; another comprising edges* `0-1, 1-2, 2-6,` *and* `6-0`*; another comprising edges* `3-5, 4-5,` *and* `3-4`*; and the single vertex* `12`*. Adding an edge connecting* `12` *to* `7, 8,` *or* `10` *would biconnect the graph.*



Property 18.8 *We can find a graph's articulation points and biconnected components in linear time.*

Proof: As for Property 18.7, this fact follows from the observation that the solutions to Exercises 18.42 and 18.43 involve minor modifications to DFS that amount to adding a few

constant-time tests per edge.■

Biconnectivity generalizes simple connectivity. Further generalizations have been the subjects of extensive studies in classical graph theory and in the design of graph algorithms. These generalizations indicate the scope of graph-processing problems that we might face, many of which are easily posed but less easily solved.

Definition 18.8 *A graph is* **k-connected** *if there are at least k vertex-disjoint paths connecting every pair of vertices in the graph. The* **vertex connectivity** *of a graph is the minimum number of vertices that need to be removed to separate it into two pieces.*

In this terminology, "1-connected" is the same as "connected" and "2-connected" is the same as "biconnected." A graph with an articulation point has vertex connectivity 1 (or 0), so Property 18.7 says that a graph is 2-connected if and only if its vertex connectivity is not less than 2. It is a special case of a classical result from graph theory, known as *Whitney's theorem*, which says that a graph is *k*-connected if and only if its vertex connectivity is not less than *k*. Whitney's theorem follows directly from *Menger's theorem* (see section 22.7), which says that the minimum number of vertices whose removal disconnects two vertices in a graph is equal to the maximum number of vertex-disjoint paths between the two vertices (to prove Whitney's theorem, apply Menger's theorem to every pair of vertices).

Definition 18.9 *A graph is* **k–edge-connected** *if there are at least k edge-disjoint paths connecting every pair of vertices in the graph. The* **edge connectivity** *of a graph is the minimum number of edges that need to be removed to separate it into two pieces.*

In this terminology, "2–edge-connected" is the same as "edge-connected" (that is, an edge-connected graph has edge connectivity greater than 1) and a graph with at least one bridge has edge connectivity 1. Another version of Menger's theorem says that the minimum number of vertices whose removal disconnects two vertices in a graph is equal to the maximum number of vertex-disjoint paths between the two vertices, which implies that a graph is *k* –edge-connected if and only if its edge connectivity is *k*.

With these definitions, we are led to generalize the connectivity problems that we considered at the beginning of this section.

st-connectivity What is the minimum number of edges whose removal will separate two given vertices *s* and *t* in a given graph? What is the minimum number of vertices whose removal will separate two given vertices *s* and *t* in a given graph?

General connectivity Is a given graph *k*-connected? Is a given graph *k* –edge-connected? What is the edge connectivity and the vertex connectivity of a given graph?

Although these problems are much more difficult to solve than are the simple connectivity problems that we have considered in this section, they are members of a large class of graph-processing problems that we can solve using the general algorithmic tools that we consider in Chapter 22 (with DFS playing an important role); we consider specific solutions in section 22.7.

Exercises

▷**18.32** If a graph is a forest, all its edges are separation edges; but which vertices are separation vertices?

▷**18.33** Consider the graph

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

Draw the standard adjacency-lists DFS tree. Use it to find the bridges and the edge-connected components.

18.34 Prove that every vertex in any graph belongs to exactly one edge-connected component.

○**18.35** Expand *Program* 18.7, in the style of *Program* 18.4, to support an ADT function for testing whether two vertices are in the same edge-connected component.

▷**18.36** Consider the graph

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

Draw the standard adjacency-lists DFS tree. Use it to find the articulation points and the biconnected components.

▷**18.37** Do the previous exercise using the standard adjacency-matrix DFS tree.

18.38 Prove that every edge in a graph either is a bridge or belongs to exactly one biconnected component.

●**18.39** Prove that any graph with no articulation points is biconnected. *Hint*: Given a pair of vertices *s* and *t* and a `path` connecting them, use the fact that none of the vertices on the `path` are articulation points to construct two disjoint paths connecting *s* and *t*.

18.40 Modify *Program* 18.3 to derive a program for determining whether a graph is bi-

connected, using a brute-force algorithm that runs in time proportional to $V(V + E)$. *Hint*: If you mark a vertex as having been seen before you start the search, you effectively remove it from the graph.

○ **18.41** Extend your solution to Exercise 18.40 to derive a program that determines whether a graph is 3-connected. Give a formula describing the approximate number of times your program examines a graph edge, as a function of V and E.

18.42 Prove that the root of a DFS tree is an articulation point if and only if it has two or more (internal) children.

● **18.43** Write an ADT function for the adjacency-lists representation that prints the biconnected components of the graph.

18.44 What is the minimum number of edges that must be present in any biconnected graph with V vertices?

18.45 Modify Programs 18.3 and 18.7 to implement an ADT function that determines whether a graph is edge-connected (returning as soon as it identifies a bridge if the graph is not). Run empirical tests to study the number of edges examined by your function, for graphs of various sizes, drawn from various graph models (see Exercises 17.63–76).

○ **18.46** Instrument Programs 18.3 and 18.7 to print out the number of articulation points, bridges, and biconnected components.

● **18.47** Run experiments to determine empirically the average values of the quantities described in Exercise 18.46 for graphs of various sizes, drawn from various graph models (see Exercises 17.63–76).

18.48 Give the edge connectivity and the vertex connectivity of the graph

```
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 BREADTH-FIRST SEARCH

Suppose that we want to find a *shortest path* between two specific vertices in a graph—a `path` connecting the vertices with the property that no other `path` connecting those vertices has fewer edges. The classical method for accomplishing this task, called *breadth-first search (BFS)*, is also the basis of numerous algorithms for processing graphs, so we consider it in detail in this section. DFS offers us little assistance in solving this problem, because the order in which it takes us through the graph has no relation-

ship to the goal of finding shortest paths. In contrast, BFS is based on this goal. To find a shortest `path` from v to w, we start at v and check for w among all the vertices that we can reach by following one edge, then we check all the vertices that we can reach by following two edges, and so forth.

When we come to a point during a graph search where we have more than one edge to traverse, we choose one and save the others to be explored later. In DFS, we use a pushdown stack (that is managed by the system to support the recursive search function) for this purpose. Using the LIFO rule that characterizes the pushdown stack corresponds to exploring passages that are close by in a maze: We choose, of the passages yet to be explored, the one that was most recently encountered. In BFS, we want to explore the vertices in order of their distance from the start. For a maze, doing the search in this order might require a search team; within a computer program, however, it is easily arranged: We simply use a *FIFO queue* instead of a stack.

Program 18.8 is an implementation of BFS for graphs represented with an adjacency matrix. It is based on maintaining a queue of all edges that connect a visited vertex with an unvisited vertex. We put a dummy self-loop to the start vertex on the queue, then perform the following steps until the queue is empty:

- Take edges from the queue until finding one that points to an unvisited vertex.

- Visit that vertex; put onto the queue all edges that go from that vertex to unvisited vertices.

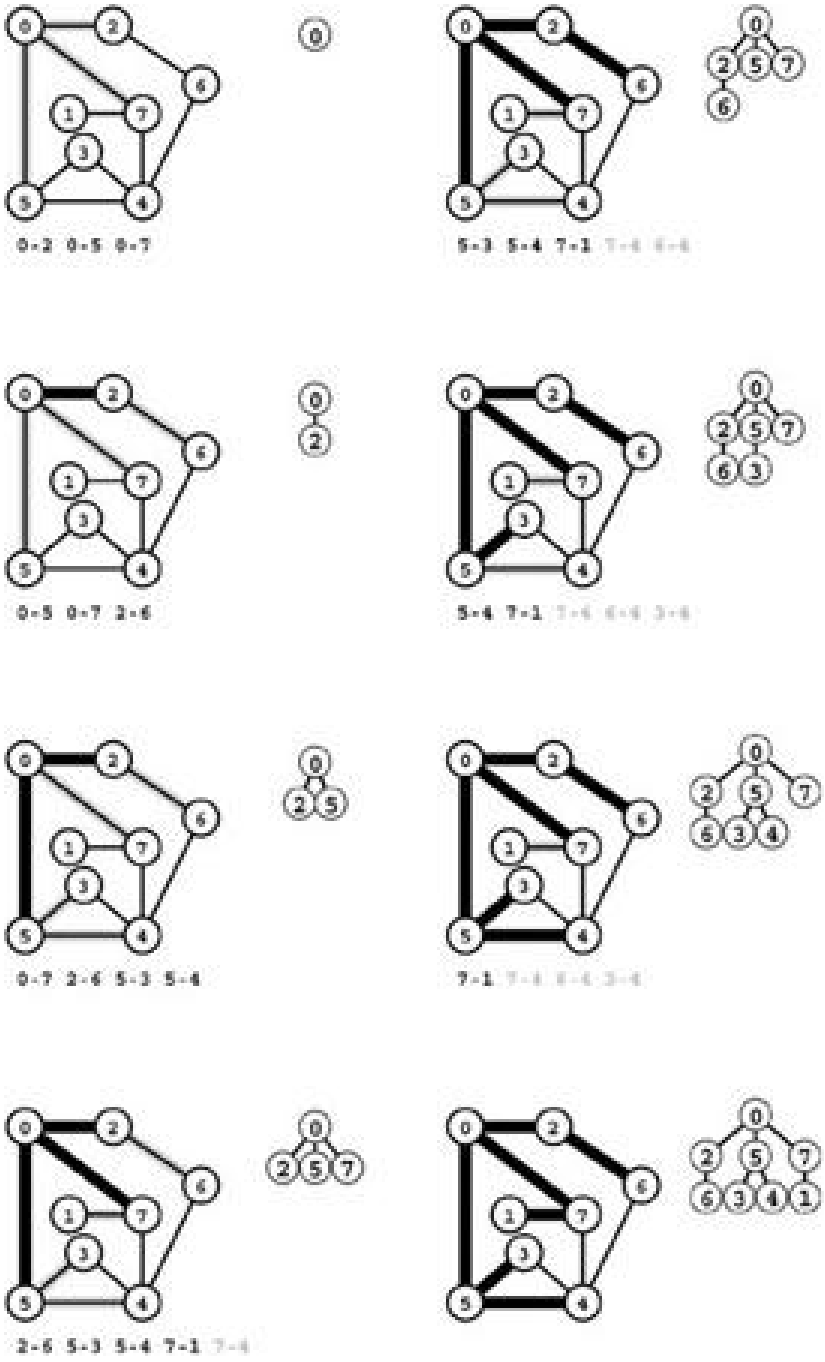
Figure 18.21 shows the step-by-step development of BFS on a sample graph.

As we saw in Section 18.4, DFS is analogous to one person exploring a maze. BFS is analogous to a group of people exploring by fanning out in all directions. Although DFS and BFS are different in many respects, there is an essential underlying relationship between the two methods—one that we noted when we briefly considered the methods in Chapter 5. In section 18.8, we consider a generalized graph-searching method that we can specialize to include these two algorithms and a host of others. Each algorithm has particular dynamic characteristics that we use to solve associated graph-processing problems. For BFS, the distance from each vertex to the start vertex (the length of a shortest `path` connecting the two) is the key property of interest.

Figure 18.21 Breadth-First Search *This figure traces the operation of BFS on our sample graph. We begin with all the edges adjacent to the start vertex on the queue (top left). Next, we move edge* `0-2` *from the queue to the tree and process its incident edges* `2-0` *and* `2-6` *(second from top, left). We do not put* `2-0` *on the queue because* `0` *is already on the tree. Third, we move edge* `0-5` *from the queue to*

IPR2023-00124
KASPERSKY EX 1036 Page 156

the tree; again `5`*'s incident edge (to* `0`*) leads nowhere new, but we add* `5-3` *and* `5-4` *to the queue (third from top, left). Next, we add* `0-7` *to the tree and put* `7-1` *on the queue (bottom left). The edge* `7-4` *is printed in gray because we could also avoid putting it on the queue, since there is another edge that will take us to* `4` *that is already on the queue. To complete the search, we take the remaining edges off the queue, completely ignoring the gray edges when they come to the front of the queue (right). Edges enter and leave the queue in order of their distance from* `0`.



Program 18.8 Breadth-first search (adjacency matrix)

Property 18.9 *During BFS, vertices enter and leave the FIFO queue in order of their distance from the start vertex.*

Proof: A stronger property holds: The queue always consists of zero or more vertices of distance k from the start, followed by zero or more vertices of distance $k + 1$ from the start, for some integer k. This stronger property is easy to prove by induction.■

For DFS, we understood the dynamic characteristics of the algorithm with the aid of the DFS search forest that describes the recursive-call structure of the algorithm. An essential property of that forest is that the forest represents the paths from each vertex back to the place that the search started for its connected component. As indicated in the implementation and shown in *Figure* 18.22, such a spanning tree also helps us to understand BFS. As with DFS, we have a forest that characterizes the dynamics of the search, one tree for each connected component, one tree node for each graph vertex, and one tree edge for each graph edge. BFS corresponds to traversing each of the trees in this forest in level order. As with DFS, we use a vertex-indexed array to represent explicitly the forest with parent links. For BFS, this forest carries essential information about the graph structure:

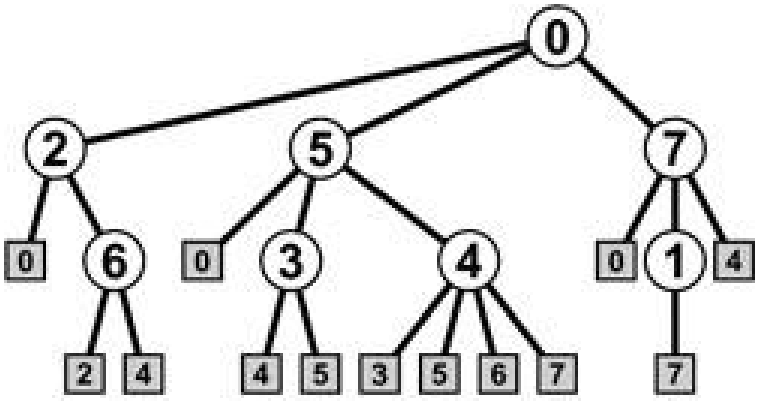
Property 18.10 *For any node w in the BFS tree rooted at v, the tree* `path` *from v to w corresponds to a shortest* `path` *from v to w in the corresponding graph.*

Proof: The tree-path lengths from nodes coming off the queue to the root are nondecreasing, and all nodes closer to the root than w are on the queue; so no shorter `path` to w was found *before* it comes off the queue, and no `path` to w that is discovered *after* it comes off the queue can be shorter than w's tree `path` length.■

As indicated in *Figure* 18.21 and noted in Chapter 5, there is no need to put an edge on the queue with the same destination vertex as any edge already on the queue, since the FIFO policy ensures that we will process the old queue edge (and visit the vertex) before we get to the new edge. One way to implement this policy is to use a queue ADT implementation where such duplication is disallowed by an ignore-the-new-item policy (see Section 4.7). Another choice is to use the global vertex-marking array for this purpose: Instead of marking a vertex as having been visited when we take it *off* the queue, we do so when we put it *on* the queue. Testing whether a vertex is marked (whether its entry has changed from its initial sentinel value) then stops us from putting any other edges that point to the same vertex on the queue. This change, shown in *Program* 18.9, gives a BFS implementation where there are never more than V edges on the queue (one edge pointing to each vertex, at most).

Figure 18.22 BFS Tree *This tree provides a compact description of the dynamic properties of BFS, in a manner similar to the tree depicted in Figure 18.9. Traversing the tree in level order tells us how the search proceeds, step by step: first we*

visit `0`*; then we visit* `2, 5`*, and* `7`*; then we check from* `2` *that* `0` *was visited and visit* `6`*; and so forth. Each tree node has a child representing each of the nodes adjacent to it, in the order they were considered by the BFS. As in Figure 18.9, links in the BFS tree correspond to edges in the graph: if we replace edges to external nodes by lines to the indicated node, we have a drawing of the graph. Links to external nodes represent edges that were not put onto the queue because they led to marked nodes: they are either parent links or cross links that point to a node either on the same level or one level closer to the root. The st array is a parent-link representation of the tree, which we can use to find a shortest* `path` *from any node to the root. For example,* `3-5-0` *is a* `path` *in the graph from* `3` *to* `0`*, since* `st[3]` *is* `5` *and* `st[5]` *is* `0`*. No other* `path` *from* `3` *to* `0` *is shorter.*



	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

Program 18.9 Improved BFS

To guarantee that the queue that we use during BFS has at most V entries, we mark the vertices as we put them on the queue.

```
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 &#60; G->V; v++)
              if ( (G->adj[w][v] == 1) && (pre[v] == -1))
               { QUEUEput (EDGE (w, v)); pre[v] = cnt++; }
          }
      }
```

The code corresponding to *Program* 18.9 for BFS in graphs represented with adjacency lists is straightforward and derives immediately from the generalized graph search that we consider in section 18.8, so we do not include it here. As we did for DFS, we consider BFS to be a linear-time algorithm.

Property 18.11 *BFS visits all the vertices and edges in a graph in time proportional to* V^2 *for the adjacency-matrix representation and to V + E for the adjacency-lists representation.*

Proof: As we did in proving the analogous DFS properties, we note by inspecting the code that we check each entry in the adjacency-matrix row or in the adjacency list precisely once for every vertex that we visit, so it suffices to show that we visit each vertex. Now, for each connected component, the algorithm preserves the following invariant: All vertices that can be reached from the start vertex (*i*) are on the BFS tree, (*ii*) are on the queue, or (*iii*) can be reached from a vertex on the queue. Each vertex moves from (*iii*) to (*ii*) to (*i*), and the number of vertices in (*i*) increases on each iteration of the loop, so that the BFS tree eventually contains all the vertices that can be reached from the start vertex.■

With BFS, we can solve the spanning tree, connected components, vertex search, and several other basic connectivity problems that we described in Section 18.4, since the solutions that we considered depend on only the ability of the search to examine every node and edge connected to the starting point. More important, as mentioned at the outset of this section, BFS is the natural graph-search algorithm for applications where we want to know a shortest `path` between two specified vertices. Next, we consider a specific solution to this problem and its extension to solve two related problems.

Shortest path Find a shortest `path` in the graph from *v* to *w*. We can accomplish this task by starting a BFS that maintains the parent-link representation st of the search tree at *v*, then stopping when we reach *w*. The `path` up the tree from *w* to *v* is a shortest path. For example, the following code prints out the `path` connecting `w` to `v`:

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

IPR2023-00124
KASPERSKY EX 1036 Page 160

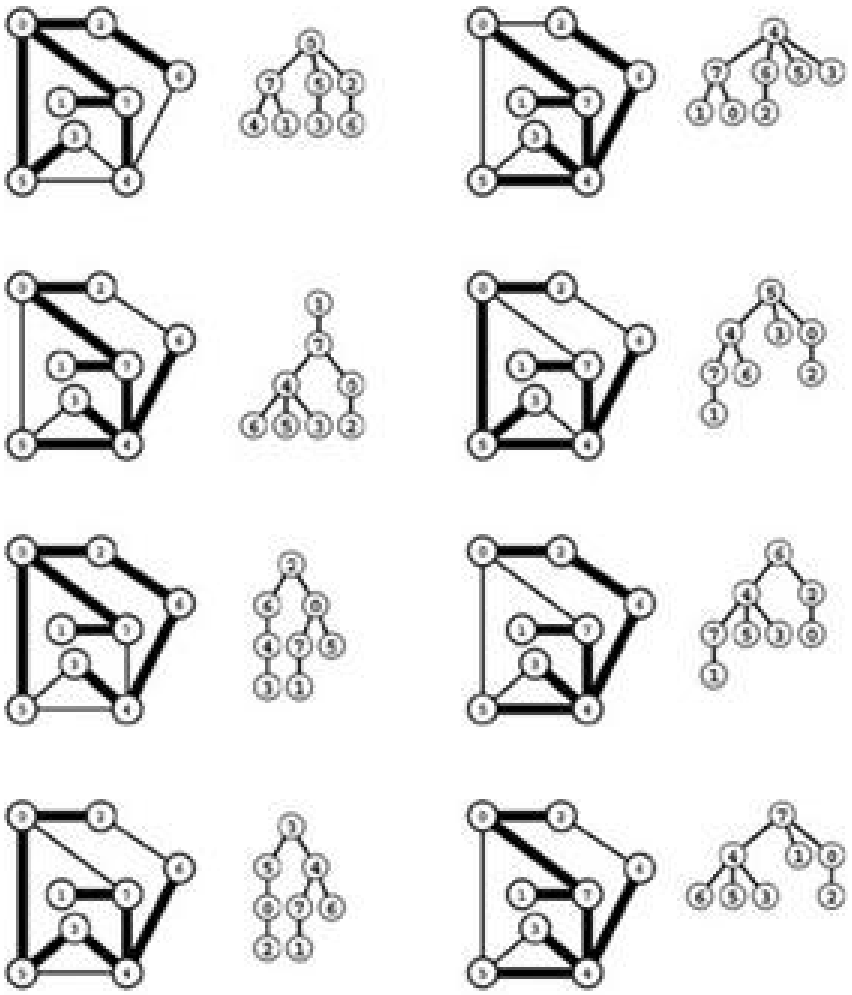
```
      printf ("%d\n", t);
```

If we want the `path` from `v` to `w`, we can replace the printf operations in this code by stack pushes, then go into a loop that prints the vertex indices as we pop them from the stack. Or, we start the search at `w` and stop at `v` in the first place.

Single-source shortest paths Find shortest paths connecting a given vertex v with each other vertex in the graph. The full BFS tree rooted at v provides a way to accomplish this task: The `path` from each vertex to the root is a shortest `path` to the root. Therefore, to solve the problem, we run BFS to completion starting at `v`. The st array that results from this computation is a parent-link prepresentation of the BFS tree, and the code in the previous paragraph will give the shortest `path` to any other vertex `w`.

All-pairs shortest paths Find shortest paths connecting each pair of vertices in the graph. The way to accomplish this task is to run BFS to solve the single-source problem for each vertex in the graph and, to support ADT functions that can handle huge numbers of shortest-path queries efficiently, store the `path` lengths and parent-link tree representations for each vertex (see *Figure* 18.23). This preprocessing requires time proportional to VE and space proportional to V^2, a potentially prohibitive cost for huge sparse graphs. However, it allows us to build an ADT with optimal performance: After investing in the preprocessing (and the space to hold the results), we can return shortest-path lengths in constant time and the paths themselves in time proportional to their length (see Exercise 18.53).

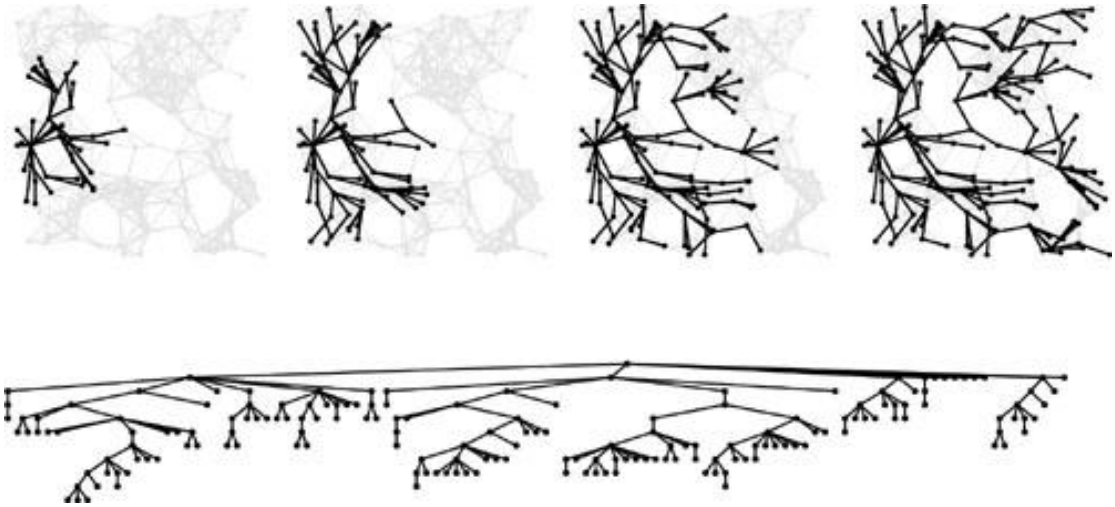
Figure 18.23 All-Pairs Shortest Paths Example *These figures depict the result of doing BFS from each vertex, thus computing the shortest paths connecting all pairs of vertices. Each search gives a BFS tree that defines the shortest paths connecting all graph vertices to the vertex at the root. The results of all the searches are summarized in the two matrices at the bottom. In the left matrix, the entry in row* `v` *and column* `w` *gives the length of the shortest* `path` *from* `v` *to* `w` *(the depth of* `v` *in* `w`*'s tree). Each row of the right matrix contains the st array for the corresponding search. For example, the shortest* `path` *from 3 to 2 has three edges, as indicated by the entry in row 3 and column 2 of the left matrix. The third BFS tree from the top on the left tells us that the* `path` *is 3-4-6-2, and this information is encoded in row 2 in the right matrix. The matrix is not necessarily symmetric when there is more than one shortest path, because the paths found depend on the BFS search order. For example, the BFS tree at the bottom on the left and row 3 of the right matrix tell us that the shortest* `path` *from 2 to 3 is 2-0-5-3.*



	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	3	2	1	2
3	2	3	3	0	1	1	3	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	6	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

Figure 18.24 Breadth-First Search *This figure illustrates the progress of BFS in random Euclidean near-neighbor graph (left), in the same style as Figure 18.13. As is evident from this example, the search tree for BFS tends to be quite short and wide for this type of graph (and many other types of graphs commonly encountered in practice). That is, vertices tend to be connected to one another by rather short paths. The contrast between the shapes of the DFS and BFS trees is striking testimony to the differing dynamic properties of the algorithms.*



These BFS-based solutions are effective, but we do not consider implementations in any further detail here because they are special cases of algorithms that we consider in detail in Chapter 21. The term *shortest paths* in graphs is generally taken to describe the corresponding problems for digraphs and networks. Chapter 21 is devoted to this topic. The solutions that we examine there are strict generalizations of the BFS-based solutions described here.

The basic characteristics of BFS search dynamics contrast sharply with those for DFS search, as illustrated in the large graph depicted in *Figure* 18.24, which you should compare with *Figure* 18.13. The tree is shallow and broad, and demonstrates a set of facts about the graph being searched different from those shown by DFS. For example,

- There exists a relatively short `path` connecting each pair of vertices in the graph.
- During the search, most vertices are adjacent to numerous unvisited vertices.

Again, this example is typical of the behavior that we expect from BFS, but verifying facts of this kind for graph models of interest and graphs that arise in practice requires detailed analysis.

DFS wends its way through the graph, storing on the stack the points where other paths branch off; BFS sweeps through the graph, using a queue to remember the frontier of visited places. DFS explores the graph by looking for new vertices far away from the start point, taking closer vertices only when dead ends are encountered; BFS completely covers the area close to the starting point, moving farther away only when everything nearby has been examined. The order in which the vertices are visited depends on the graph structure and representation, but these global properties of the search trees are more informed by the algorithms than by the graphs or their representations.

The key to understanding graph-processing algorithms is to realize that not only are various different search strategies effective ways to learn various different graph properties, but also we can implement many of them uniformly. For example, the DFS illustrated in *Figure* 18.13 tells us that the graph has a long path, and the BFS illustrated in *Figure* 18.24 tells us that it has many short paths. Despite these marked dynamic differences, DFS and BFS are similar, essentially differing in only the data structure that we use to save edges that are not yet explored (and the fortuitous circumstance that we can use a recursive implementation for DFS, to have the system maintain an implicit stack for us). Indeed, we turn next to a generalized graph-search algorithm that encompasses DFS, BFS, and a host of other useful strategies, and will serve as the basis for solutions to numerous classic graph-processing problems.

Exercises

18.49 Draw the BFS forest that results from a standard adjacency-lists BFS of the graph

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

18.50 Draw the BFS forest that results from a standard adjacency-matrix BFS of the graph

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

18.51 Give the all-shortest-path matrices (in the style of *Figure* 18.23) for the graph

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

assuming that you use the adjacency-matrix representation.

18.52 Give a BFS implementation (a version of *Program* 18.9) that uses a standard FIFO queue of vertices. Include a test in the BFS search code to ensure that no duplicates go on the queue.

18.53 Develop ADT functions for the adjacency-lists implementation of the graph ADT (*Program* 17.6) that supports shortest-path queries after preprocessing to compute all shortest paths. Specifically, add two array pointers to the graph representation, and write a preprocessing function `GRAPHshortpaths` that assigns values to all their entries as illustrated in *Figure* 18.23. Then, add two query functions `GRAPHshort(v, w)` (that returns the shortest-path length between `v` and `w`) and `GRAPHpath(v, w)` (that returns the vertex adjacent to `v` that is on a shortest `path` between `v` and `w`).

▷**18.54** What does the BFS tree tell us about the distance from v to w when neither is at the root?

18.55 Develop a graph ADT function that returns the `path` length that suffices to connect any pair of vertices. (This quantity is known as the graph's *diameter*). *Note*: You need to define a convention for the return value in the case that the graph is not connected.

18.56 Give a simple optimal recursive algorithm for finding the diameter of a *tree* (see Exercise 18.55).

○**18.57** Instrument a BFS implementation for the adjacency-lists representation (see *Program* 18.10) to print out the height of the BFS tree and the percentage of edges that must be processed for every vertex to be seen.

●**18.58** Run experiments to determine empirically the average values of the quantities described in Exercise 18.57 for graphs of various sizes, drawn from various graph models (see Exercises 17.63–76).

18.8 GENERALIZED GRAPH SEARCH

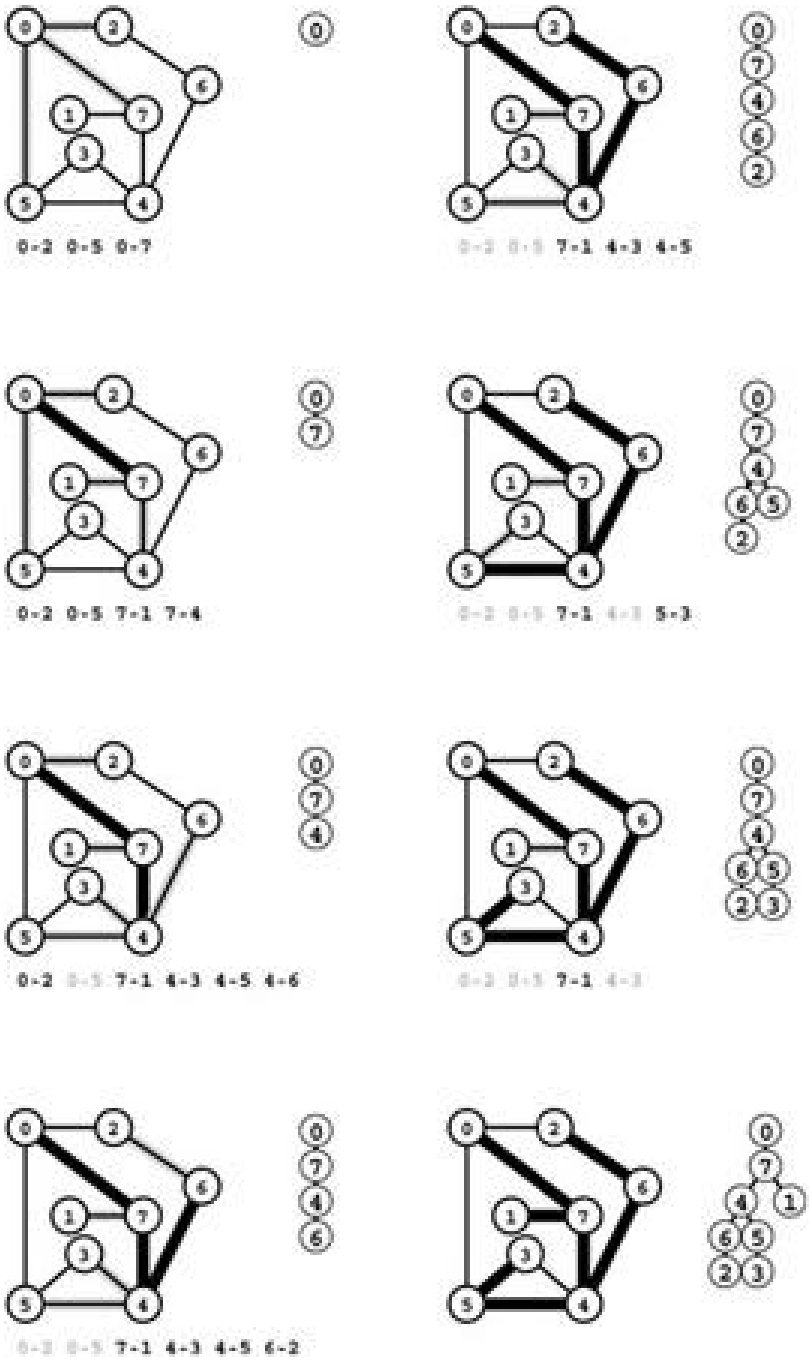
DFS and BFS are fundamental and essential graph-traversal methods that lie at the heart of numerous graph-processing algorithms. Knowing their essential properties, we now move to a higher level of abstraction, where we see that both methods are special cases of a generalized strategy for moving through a graph, one that is suggested by our BFS implementation (*Program* 18.9).

The basic idea is simple: We revisit our description of BFS from Section 18.6, but we use the term generic term *fringe*, instead of *queue*, to describe the set of edges that are possible candidates for being next added to the tree. We are led immediately to a general strategy for searching a connected component of a graph. Starting with a self-loop to a start vertex on the fringe and an empty tree, perform the following operation until the fringe is empty:

Figure 18.25 Stack-Based DFS *Together with Figure 18.21, this figure illustrates that BFS and DFS differ only in the underlying data structure. For BFS, we used a queue; for DFS we use a stack. We begin with all the edges adjacent to the start vertex on the stack (top left). Second, we move edge* `0-7` *from the stack to the tree and push onto the stack its incident edges that go to vertices not yet on the tree* `7-1`, `7-4`, *and* `7-6` *(second from top, left). The LIFO stack discipline implies that, when we put an edge on the stack, any edges that points to the same vertex are obsolete and will be ignored when they reach the top of the stack. Such edges are printed in gray. Third, we move edge* `7-6` *from the stack to the tree and push its incident edges on the stack (third from top, left). Next, we pop* `4-6` *push its inci-*

IPR2023-00124
KASPERSKY EX 1036 Page 165

dent edges on the stack, two of which will take us to new vertices (bottom left). To complete the search, we take the remaining edges off the stack, completely ignoring the gray edges when they come to the top of the stack (right).



Move an edge from the fringe to the tree. If the vertex to which it leads is unvisited, visit that vertex, and put onto the fringe all edges that lead from that vertex to unvisited vertices.

This strategy describes a family of search algorithms that will visit all the vertices and edges of a connected graph, *no matter what* type of generalized queue we use to hold the fringe edges.

When we use a queue to implement the fringe, we get BFS, the topic of Section 18.6. When we use a stack to implement the fringe, we get DFS. *Figure* 18.25, which you should compare with Figures 18.6 and 18.21, illustrates this phenomenon in detail. Proving this equivalence between recursive and stack-based DFS is an interesting exercise in recursion removal, where we essentially transform the stack underlying the recursive program into the stack implementing the fringe (see Exercise 18.61). The search order for the DFS depicted in *Figure* 18.25 differs from the one depicted in *Figure* 18.6 only because the stack discipline implies that we check the edges incident on each vertex in the reverse of the order that we encounter them in the adjacency matrix (or the adjacency lists). The basic fact remains that, if we change the data structure used by *Program* 18.8 to be a stack instead of a queue (which is trivial to do because the ADT interfaces of the two data structures differ in only the function names), then we change that program from BFS to DFS.

Now, as we discussed in section 18.7, this general method may not be as efficient as we would like, because the fringe becomes cluttered up with edges that point to vertices that are moved to the tree during the time that the edge is on the fringe. For FIFO queues, we avoid this situation by marking destination vertices when we put edges on the queue. We ignore edges to fringe vertices because we know that they will never be used: The old one will come off the queue (and the vertex visited) before the new one does (see *Program* 18.9). For a stack implementation, we want the opposite: When an edge is to be added to the fringe that has the same destination vertex as the one already there, we know that the *old* edge will never be used, because the new one will come off the stack (and the vertex visited) before the old one. To encompass these two extremes and to allow for fringe implementations that can use some other policy to disallow edges on the fringe that point to the same vertex, we modify our general scheme as follows:

Figure 18.26 Graph Search Terminology *During a graph search, we maintain a search tree (black) and a fringe (gray) of edges that are candidates to be next added to the tree. Each vertex is either on the tree (black), the fringe (gray), or not yet seen (white). Tree vertices are connected by tree edges, and each fringe vertex is connected by a fringe edge to a tree vertex.*

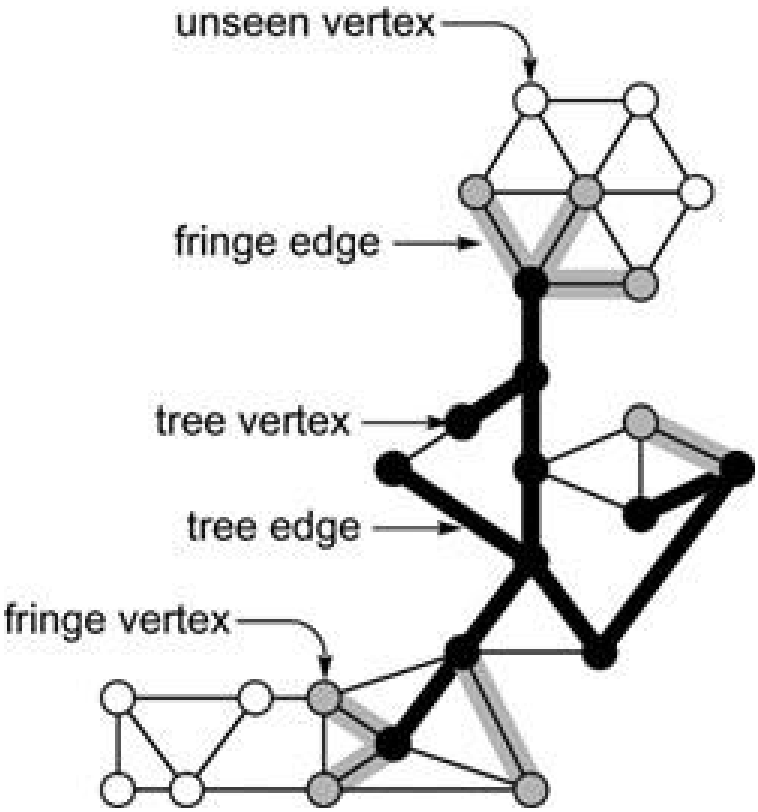
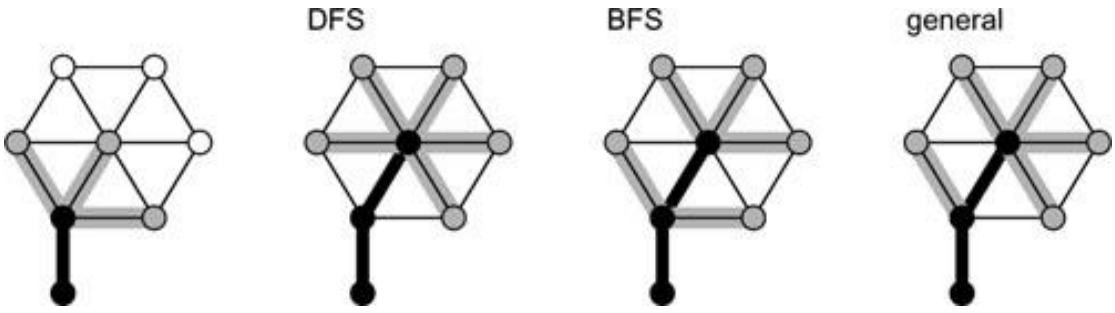


Figure 18.27 Graph Search Strategies *This figure illustrates different possibilities when we take a next step in the search illustrated in Figure 18.26. We move a vertex from the fringe to the tree (in the center of the wheel at the top right) and check all its edges, putting those to unseen vertices on the fringe and using an algorithm-specific replacement rule to decide whether those to fringe vertices should be skipped or should replace the fringe edge to the same vertex. In DFS, we always replace the old edges; in BFS, we always skip the new edges; and in general, we might replace some and skip others.*



Move an edge from the fringe to the tree. Visit the vertex that it leads to, and put all edges that lead from that vertex to unvisited vertices onto the fringe, using a replacement policy on the fringe that ensures that no two edges on the fringe point to the same vertex.

The no-duplicate-destination-vertex policy on the fringe guarantees that we do not need to test whether the destination vertex of the edge coming off the queue has been visited.

For BFS, we use a queue implementation with an ignore-the-new-item policy; for DFS, we need a stack with a forget-the-old-item policy; but any generalized queue and any replacement policy at all will still yield an effective method for visiting all the vertices and edges of the graph in linear time and extra space proportional to *V*. *Figure* 18.27 is a schematic illustration of these differences. We have a family of graph-searching strategies that includes both DFS and BFS and whose members differ only in the generalized-queue implementation that they use. As we shall see, this family encompasses numerous other classical graph-processing algorithms.

Program 18.10 is an implementation based on these ideas for graphs represented with adjacency lists. It puts fringe edges on a generalized queue, and uses the usual vertex-indexed arrays to identify fringe vertices, so that it can use an explicit *update* ADT operation whenever it encounters another edge to a fringe vertex. The ADT implementation can choose to ignore the new edge or to replace the old one.

Property 18.12 *Generalized graph searching visits all the vertices and edges in a graph in time proportional to* V^2 *for the adjacency-matrix representation and to V + E for the adjacency-lists representation plus, in the worst case, the time required for V insert, V remove, and E update operations in a generalized queue of size V.*

Program 18.10 Generalized graph search

This implementation of search for *Program* 18.3 generalizes BFS and DFS and supports numerous other graph-processing algorithms (see section 21.2 for a discussion of these algorithms and alternate implementations). It maintains a generalized queue of edges called the *fringe*. We initialize the fringe with a self-loop to the start vertex; then, while the fringe is not empty, we move an edge e from the fringe to the tree (attached at P|e.v|) and scan `e.w`'s adjacency list, moving unseen vertices to the fringe and calling `GQupdate` for new edges to fringe vertices.

This code makes judicious use of pre and st to guarantee that no two edges on the fringe point to the same vertex. A vertex `v` is the destination vertex of a fringe edge if and only if it is marked (`pre[v]` is nonnegative) but it is not yet on the tree (`st[v]` is `-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));
    }
}
```

Proof: The proof of Property 18.12 does not depend on the queue implementation, and therefore applies. The stated extra time requirements for the generalized-queue operations follow immediately from the implementation.■

There are many other effective ADT designs for the fringe that we might consider. For example, as with our first BFS implementation, we could stick with our first general scheme and simply put all the edges on the fringe, then ignore those that go to tree vertices when we take them off. The disadvantage of this approach, as with BFS, is that the maximum queue size has to be E instead of V. Or, we could handle updates implicitly in the ADT implementation, just by specifying that no two edges with the same destination vertex can be on the queue. But the simplest way for the ADT implementation to do so is essentially equivalent to using a vertex-indexed array (see Exercises 4.51 and 4.54), so the test fits more comfortably into the client graph-search program.

Program 18.11 Random queue implementation

When we remove an item from this data structure, it is equally likely to be any one of the items currently in the data structure. We can use this code to implement the generalized-queue ADT for graph searching to search a graph in a “random” fashion (*see text*).

```
#include &#60;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];
  }
```

The combination of *Program* 18.10 and the generalized-queue abstraction gives us a general and flexible graph-search mechanism. To illustrate this point, we now consider briefly two interesting and useful alternatives to BFS and DFS.

The first alternative strategy is based on *randomized queues* (see Section 4.6). In a randomized queue, we *delete* items randomly: Each item on the data structure is equally likely to be the one removed. *Program* 18.11 is an implementation that provides this functionality. If we use this code to implement the generalized queue ADT for *Program* 18.10, then we get a randomized graph-searching algorithm, where each vertex on the fringe is equally likely to be the next one added to the tree. The edge (to that vertex) that is added to the tree depends on the implementation of the *update* operation. The implementation in *Program* 18.11 does no updates, so each fringe vertex is added to the tree with the edge that caused it to be moved to the fringe. Alternatively, we might choose to always do updates (which results in the most recently encountered edge to each fringe vertex being added to the tree), or to make a random choice.

Another strategy, which is critical in the study of graph-processing algorithms because it serves as the basis for a number of the classical algorithms that we address in Chapters 20 through 22, is to use a *priority-queue* ADT (see Chapter 9) for the fringe: We assign priority values to each edge on the fringe, update them as appropriate, and choose the highest-priority edge as the one to be added next to the tree. We consider this formulation in detail in Chapter 20. The queue-maintenance operations for priority queues are more costly than are those for stacks and queues because they involve implicit compar-

isons among items on the queue, but they can support a much broader class of graph-search algorithms. As we shall see, several critical graph-processing problems can be addressed simply with judicious choice of priority assignments in a priority-queue–based generalized graph search.

All generalized graph-searching algorithms examine each edge just once and take extra space proportional to V in the worst case; they do differ, however, in some performance measures. For example, *Figure* 18.28 shows the size of the fringe as the search progresses for DFS, BFS, and randomized search; *Figure* 18.29 shows the tree computed by randomized search for the same example as *Figure* 18.13 and *Figure* 18.24. Randomized search has neither the long paths of DFS nor the high-degree nodes of BFS. The shapes of these trees and the fringe plots depend on the structure of the particular graph being searched, but they also characterize the different algorithms.

Figure 18.28 Fringe Sizes for DFS, Randomized Graph Search, and BFS
These plots of the fringe size during the searches illustrated in Figures 18.13, 18.24, and 18.29 indicate the dramatic effects that the choice of data structure for the fringe can have on graph searching. When we use a stack, in DFS (top), we fill up the fringe early in the search as we find new nodes at every step, then we end the search by removing everything. When we use a randomized queue (center), the maximum queue size is much lower. When we use a FIFO queue in BFS (bottom), the maximum queue size is still lower, and we discover new nodes throughout the search.

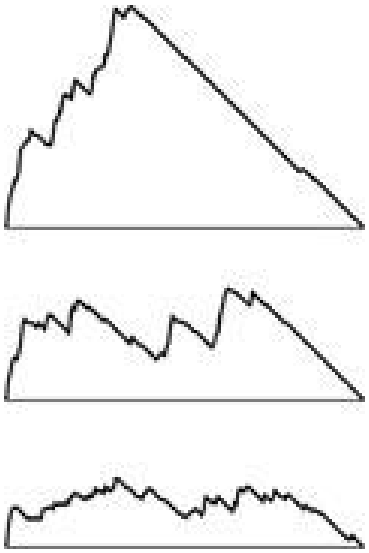
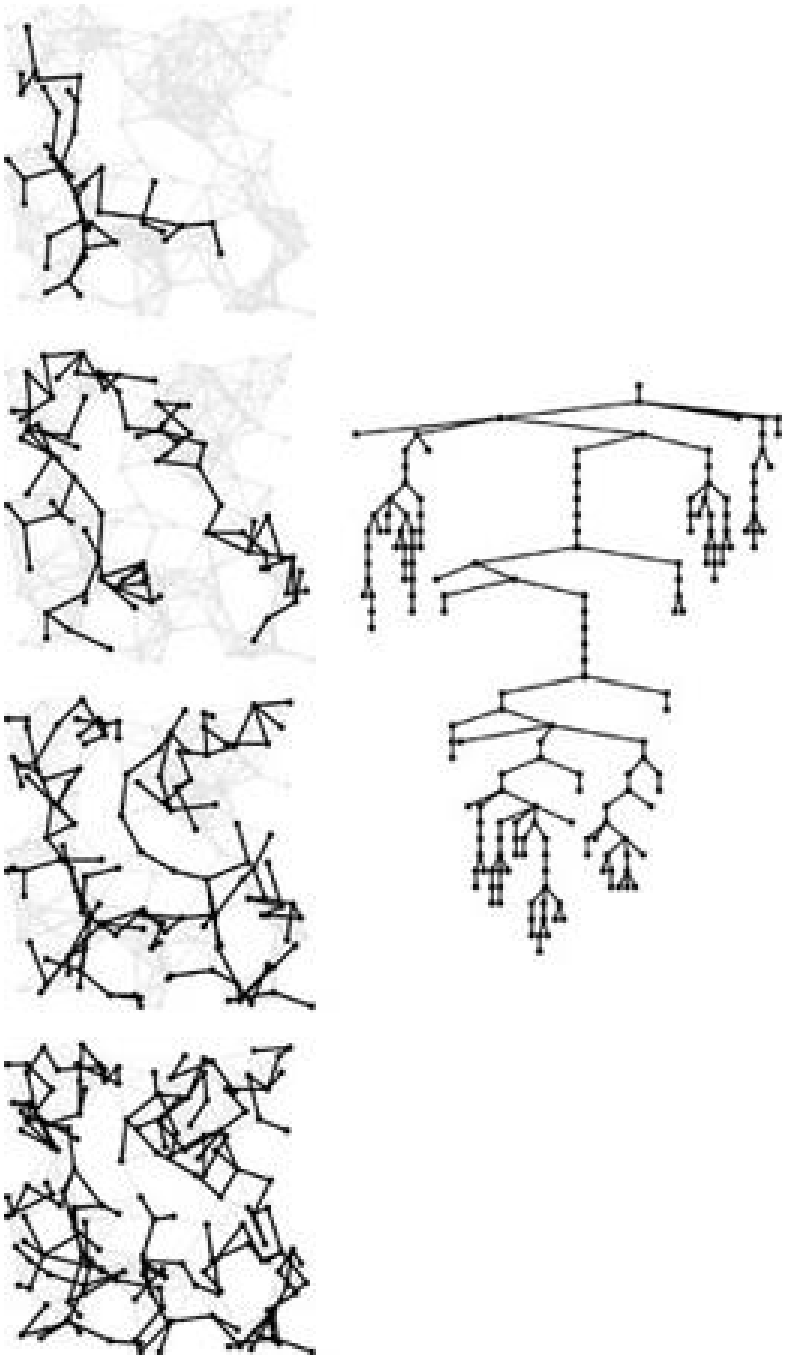


Figure 18.29 Randomized Graph Search *This figure illustrates the progress of randomized graph searching (left), in the same style as Figures 18.13 and 18.24. The search tree shape falls somewhere between the BFS and DFS shapes. The dynamics of these three algorithms, which differ only in the data structure for work to be completed, could hardly be more different.*



We could generalize graph searching still further by working with a forest (not necessarily a tree) during the search. Although we stop short of working at this level of generality throughout, we consider a few algorithms of this sort in Chapter 20.

Exercises

○ **18.59** Discuss the advantages and disadvantages of a generalized graph-searching implementation that is based on the following policy: "Move an edge from the fringe to the tree. If the vertex that it leads to is unvisited, visit that vertex and put all its incident edges onto the fringe."

IPR2023-00124
KASPERSKY EX 1036 Page 173

18.60 Modify the adjacency-lists ADT to include edges (in addition to destination vertices) on the lists, then implement a graph search based on the strategy described in Exercise 18.59 that visits every edge but destroys the graph, taking advantage of the fact that you can move all of a vertex's edges to the fringe with a single link change.

●**18.61** Prove that recursive DFS (Programs 18.3 and 18.2) is equivalent to generalized graph search using a stack (*Program* 18.10), in the sense that both programs will visit all vertices in precisely the same order for all graphs if and only if the programs scan the adjacency lists in opposite orders.

18.62 Give three different possible traversal orders for randomized search through the graph

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

18.63 Could randomized search visit the vertices in the graph

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

in numerical order of their indices? Prove your answer.

18.64 Provide a generalized-queue implementation for graph edges that disallows edges with duplicate vertices on the queue, using an ignore-the-new-item policy.

○**18.65** Develop a randomized graph search that chooses each *edge* on the fringe with equal likelihood. *Hint*: See *Program* 18.8.

○ **18.66** Describe a maze-traversal strategy that corresponds to using a standard push-down stack for generalized graph searching (see Section 18.1).

○ **18.67** Instrument generalized graph searching (see *Program* 18.10) to print out the height of the tree and the percentage of edges processed for every vertex to be seen.

● **18.68** Run experiments to determine empirically the average values of the quantities described in Exercise 18.67 for generalized graph search with a random queue in graphs of various sizes, drawn from various graph models (see Exercises 17.63–76).

● **18.69** Write a client program that does dynamic graphical animations of generalized graph search for graphs that have (x, y) coordinates associated with each vertex (see Exercises 17.55 through 17.59). Test your program on random Euclidean neighbor graphs, using as many points as you can process in a reasonable amount of time. Your program

should produce images like the snapshots shown in Figures 18.13, 18.24, and 18.29, although you should feel free to use colors instead of shades of gray to denote tree, fringe, and unseen vertices and edges.

18.9 ANALYSIS OF GRAPH ALGORITHMS

We have for our consideration a broad variety of graph-processing problems and methods for solving them, so we do not always compare numerous different algorithms for the same problem, as we have in other domains. Still, it is always valuable to gain experience with our algorithms by testing them on real data, or on artificial data that we understand and that have relevant characteristics that we might expect to find in actual applications.

As we discussed briefly in Chapter 2, we seek—ideally—natural input models that have three critical properties:

- They reflect reality to a sufficient extent that we can use them to predict performance.
- They are sufficiently simple that they are amenable to mathematical analysis.
- We can write generators that provide problem instances that we can use to test our algorithms.

With these three components, we can enter into a design-analysis-implementation-test scenario that leads to efficient algorithms for solving practical problems.

For domains such as sorting and searching, we have seen spectacular success along these lines in Parts 3 and 4. We can analyze algorithms, generate random problem instances, and refine implementations to provide extremely efficient programs for use in a host of practical situations. For some other domains that we study, various difficulties can arise. For example, mathematical analysis at the level that we would like is beyond our reach for many geometric problems, and developing an accurate model of the input is a significant challenge for many string-processing algorithms (indeed, doing so is an essential part of the computation). Similarly, graph algorithms take us to a situation where, for many applications, we are on thin ice with respect to all three properties listed in the previous paragraph:

- The mathematical analysis is challenging, and many basic analytic questions remain unanswered.
- There is a huge number of different types of graphs, and we cannot reasonably test our algorithms on all of them.

IPR2023-00124
KASPERSKY EX 1036 Page 175

● Characterizing the types of graphs that arise in practical problems is, for the most part, a poorly understood problem.

Graphs are sufficiently complicated that we often do not fully understand the essential properties of the ones that we see in practice *or* of the artificial ones that we can perhaps generate and analyze.

The situation is perhaps not as bleak as just described for one primary reason: Many of the graph algorithms that we consider are *optimal* in the worst case, so predicting performance is a trivial exercise. For example, *Program* 18.7 finds the bridges after examining each edge and each vertex just once. This cost is the same as the cost of building the graph data structure, and we can confidently predict, for example, that doubling the number of edges will double the running time, no matter what kind of graphs we are processing.

When the running time of an algorithm depends on the structure of the input graph, predictions are much harder to come by. Still, when we need to process huge numbers of huge graphs, we want efficient algorithms for the same reasons that we want them for any other problem domain, and we will continue to pursue the goals of understanding the essential properties of algorithms and applications, striving to identify those methods that can best handle the graphs that might arise in practice.

To illustrate some of these issues, we revisit the study of graph connectivity, a problem that we considered already in Chapter 1 (!). Connectivity in random graphs has fascinated mathematicians for years, and it has been the subject of an extensive literature. That literature is beyond the scope of this book, but it provides a backdrop that validates our use of the problem as the basis for some experimental studies that help us understand the basic algorithms that we use and the types of graphs that we are considering.

For example, growing a graph by adding random edges to a set of initially isolated vertices (essentially, the process behind *Program* 17.7) is a well-studied process that has served as the basis for classical random graph theory. It is well known that, as the number of edges grows, the graph coalesces into just one giant component. The literature on random graphs gives extensive information about the nature of this process, for example:

Table 18.1 Connectivity in two random graph models This table shows the number of connected components and the size of the largest connected component for 100000-vertex graphs drawn from two different distributions. For the random graph model, these experiments support the well-known fact that the graph is highly likely to consist primarily of one giant component if the average vertex degree is larger than a small constant. The right two columns give experimental results when we restrict the edges to be chosen from those that connect each vertex to just one of 10 specified neighbors.

IPR2023-00124
KASPERSKY EX 1036 Page 176

	random edges		random 10-neighbors	
E	C	L	C	L
1000	99000	5	99003	3
2000	98000	4	98010	4
5000	95000	6	95075	5
10000	90000	8	90300	7
20000	80002	16	81381	9
50000	50003	1701	57986	27
100000	16236	79633	28721	151
200000	1887	98049	3818	6797
500000	4	99997	19	99979
1000000	1	100000	1	100000

Key:
C number of connected components
L size of largest connected component

Property 18.13 *If* $E > \frac{1}{2} V \ln V + \mu V$ *(with* μ *positive), a random graph with* V *vertices and* E *edges consists of a single connected component and an average of less than* $e^{-2\mu}$ *isolated vertices, with probability approaching 1 as* V *approaches infinity.*

Proof: This fact was established by seminal work of Erdös and Renyi in 1960. The proof is beyond the scope of this book (*see reference section*).■

This property tells us that we can expect large nonsparse random graphs to be connected.

For example, if $V > 1000$ and $E > 10\,V$, then $\mu > 10 - \frac{1}{2} \ln 1000 > 6.5$ and the average number of vertices not in the giant component is (almost surely) less than $e^{-13} < .000003$. If we generate a million random 1000-vertex graphs of density greater than 10, we might get a few with a single isolated vertex, but the rest will all be connected.

Figure 18.30 compares random graphs with random neighbor graphs, where we allow only edges that connect vertices whose indices are within a small constant of one another. The neighbor-graph model yields graphs that are evidently substantially different in character from random graphs. We eventually get a giant component, but it appears suddenly, when two large components merge.

Table 18.2 shows that these structural differences between random graphs and neighbor graphs persist for V and E in ranges of practical interest. Such structural differences certainly may be reflected in the performance of our algorithms.

Table 18.3 gives empirical results for the cost of finding the number of connected components in a random graph, using various algorithms. Although the algorithms perhaps are not subject to direct comparison for this specific task because they are designed to handle different tasks, these experiments do validate a subset of the general conclusions that we have drawn.

First, it is plain from the table that we should not use the adjacency-matrix representation for large sparse graphs (and cannot use it for huge ones), not just because the cost of initializing the array is prohibitive, but also because the algorithm inspects every entry in the array, so its running time is proportional to the size (V^2) of the array rather than to the number of 1s in it (E). The table shows, for example, that it takes about as long to process a graph that contains 1000 edges as it does to process one that contains 100000 edges when we are using an adjacency matrix.

Second, it is also plain from Table 18.3 that the cost of allocating memory for the list nodes is significant when we build adjacency lists for large sparse graphs. The cost of building the lists is more than five times the cost of traversing them. In the typical situation where we are going to perform numerous searches of various types after building the graph, this cost is acceptable. Otherwise, we might consider preallocating the array to reduce the cost of memory allocation, as discussed in Chapter 2.

Figure 18.30 Connectivity in Random Graphs *This figures show the evolution of two types of random graphs at 10 equal increments as a total of 2E edges are added to initially empty graphs. Each plot is a histogram of the number of vertices in components of size 1 through V – 1 (left to right). We start out with all vertices in components of size 1 and end with nearly all vertices in a giant component. The plot at left is for a standard random graph: the giant component forms quickly, and all other components are small. The plot at right is for a random neighbor graph: components of various sizes persist for a longer time.*

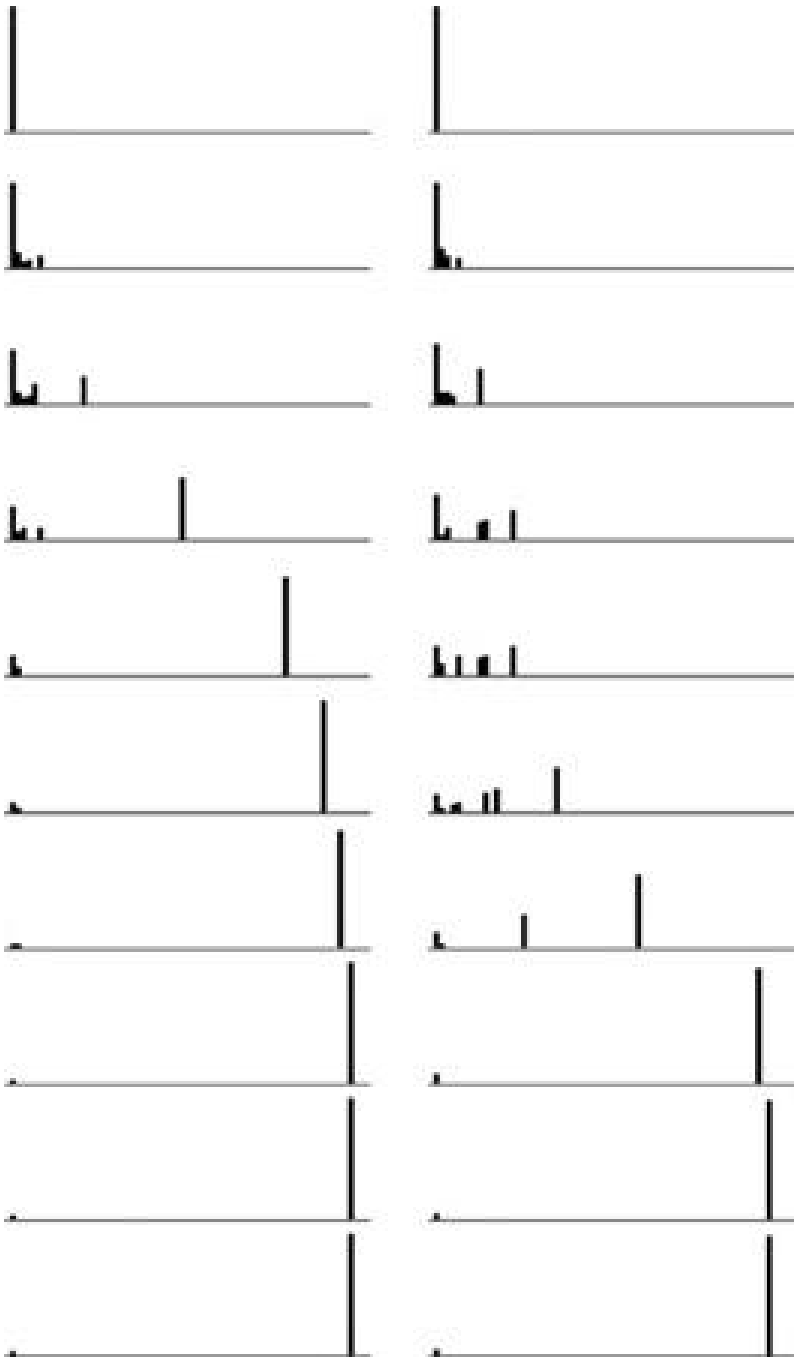


Table 18.2 Empirical study of graph-search algorithms This table shows relative timings for various algorithms for the task of determining the number of connected components (and the size of the largest one) for graphs with various numbers of vertices and edges. As expected, algorithms that use the adjacency-matrix representation are slow for sparse graphs, but competitive for dense graphs. For this specialized task, the union-find algorithms that we considered in Chapter 1 are the fastest, because they build a data structure tailored to solve the problem and do not need otherwise to represent the graph. Once the data structure representing the graph has been built, however, DFS and BFS are faster, and more flexible. Adding a test to stop when the graph is known to consist of a single connected component significantly speeds up DFS and union-find (but not BFS) for dense graphs.

IPR2023-00124
KASPERSKY EX 1036 Page 179

			adjacency matrix			adjacency lists				
E	U	U*	I	D	D*	I	D	D*	B	B*
5000 vertices										
500	1	0	255	312	356	1	0	0	0	1
1000	0	1	255	311	354	1	0	0	0	1
5000	1	2	258	312	353	2	2	1	2	1
10000	3	3	258	314	358	5	2	1	2	1
50000	12	6	270	315	202	25	6	4	5	6
100000	23	7	286	314	181	52	9	2	10	11
500000	117	5	478	248	111	267	54	16	56	47
100000 vertices										
5000	5	3				3	8	7	24	24
10000	4	5				6	7	7	24	24
50000	18	18				26	12	12	28	28
100000	34	35				51	28	24	34	34
500000	133	137				259			88	89

Key:
U weighted quick union with halving (Program 1.4)
I initial construction of the graph representation
D recursive DFS (Programs 18.1 and 18.2)
B BFS (Programs 18.9 and 18.10)
* exit when graph is found to be fully connected

Third, the absence of numbers in the DFS columns for large sparse graphs is significant. These graphs cause excessive recursion depth, which (eventually) cause the program to crash. If we want to use DFS on such graphs, we need to use the nonrecursive version that we discussed in section 18.7.

Fourth, the table shows that the union-find–based method from Chapter 1 is faster than DFS or BFS, primarily because it does not have to represent the entire graph. Without such a representation, however, we cannot answer simple queries such as "Is there an edge connecting `v` and `w`?" so union-find–based methods are not suitable if we want to do more than what they are designed to do (answer "Is there a `path` between `v` and `w`?" queries intermixed with adding edges). Once the internal representation of the graph has been built, it is not worthwhile to implement a union-find algorithm just to determine whether it is connected, because DFS or BFS can provide the answer about as quickly.

When we run empirical tests that lead to tables of this sort, various anomalies might require further explanation. For example, on many computers, the cache architecture and other features of the memory system might have dramatic impact on performance for large graphs. Improving performance in critical applications may require detailed knowledge of the machine architecture in addition to all the factors that we are considering.

Careful study of these tables will reveal more properties of these algorithms than we are able to address. Our aim is not to do an exhaustive comparison but to illustrate that, despite the many challenges that we face when we compare graph algorithms, we can and should run empirical studies and make use of any available analytic results, both to get a feeling for the algorithms' important characteristics and to predict performance.

Exercises

○ **18.70** Do an empirical study culminating in a table like Table 18.2 for the problem of determining whether or not a graph is bipartite (two-colorable).

18.71 Do an empirical study culminating in a table like Table 18.2 for the problem of determining whether or not a graph is biconnected.

○ **18.72** Do an empirical study to find the expected size of the second largest connected component in sparse graphs of various sizes, drawn from various graph models (see Exercises 17.63–76).

18.73 Write a program that produces plots like those in *Figure* 18.30, and test it on graphs of various sizes, drawn from various graph models (see Exercises 17.63–76).

○ **18.74** Modify your program from Exercise 18.73 to produce similar histograms for the sizes of edge-connected components.

● ● **18.75** The numbers in the tables in this section are results for only one sample. We might wish to prepare a similar table where we run 1000 experiments for each entry and give the sample mean and standard deviation, but we probably could not include nearly as many entries. Would this approach be a better use of computer time? Defend your answer.

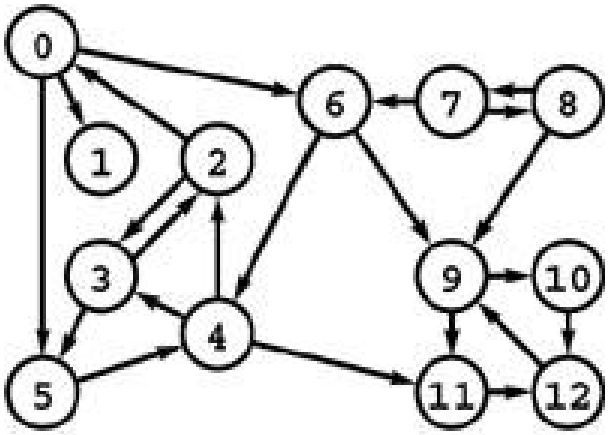
IPR2023-00124
KASPERSKY EX 1036 Page 181

Chapter Nineteen
Digraphs and DAGs

WHEN WE ATTACH significance to the order in which the two vertices are specified in each edge of a graph, we have an entirely different combinatorial object known as a *digraph*, or *directed graph*. *Figure* 19.1 shows a sample digraph. In a digraph, the notation `s-t` describes an edge that goes from `s` to `t`, but provides no information about whether or not there is an edge from `t` to `s`. There are four different ways in which two vertices might be related in a digraph: no edge; an edge `s-t` from `s` to `t`; an edge `t-s` from `t` to `s`; or two edges `s-t` and `t-s`, which indicate connections in both directions. The one-way restriction is natural in many applications, easy to enforce in our implementations, and seems innocuous; but it implies added combinatorial structure that has profound implications for our algorithms and makes working with digraphs quite different from working with undirected graphs. Processing digraphs is akin to traveling around in a city where *all* the streets are one-way, with the directions not necessarily assigned in any uniform pattern. We can imagine that getting from one point to another in such a situation could be a challenge indeed.

We interpret edge directions in digraphs in many ways. For example, in a telephone-call graph, we might consider an edge to be directed from the caller to the person receiving the call. In a transaction graph, we might have a similar relationship where we interpret an edge as representing cash, goods, or information flowing from one entity to another. We find a modern situation that fits this classic model on the Internet, with vertices representing Web pages and edges the links between the pages. In section 19.4, we examine other examples, many of which model situations that are more abstract.

Figure 19.1 A Directed Graph (Digraph) *A digraph is defined by a list of nodes and edges* (*bottom*)*, with the order that we list the nodes when specifying an edge implying that the edge is directed from the first node to the second. When drawing a digraph, we use arrows to depict directed edges* (*top*).



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		

One common situation is for the edge direction to reflect a precedence relationship. For example, a digraph might model a manufacturing line: Vertices correspond to jobs to be done and an edge exists from vertex *s* to vertex *t* if the job corresponding to vertex *s* must be done before the job corresponding to vertex *t*. Another way to model the same situation is to use a *PERT chart*: edges represent jobs and vertices implicitly specify the precedence relationships (at each vertex, all incoming jobs must complete before any outgoing jobs can begin). How do we decide when to perform each of the jobs so that none of the precedence relationships are violated? This is known as a *scheduling* problem. It makes no sense if the digraph has a cycle so, in such situations, we are working with *directed acyclic graphs (DAGs)*. We shall consider basic properties of DAGs and algorithms for this simple scheduling problem, which is known as *topological sorting*, in Sections 19.5 through 19.7. In practice, scheduling problems generally involve weights on the vertices or edges that model the time or cost of each job. We consider such problems in Chapters 21 and 22.

The number of possible digraphs is truly huge. Each of the V^2 possible directed edges (including self-loops) could be present or not, so the total number of different digraphs is 2^{V^2}. As illustrated in *Figure* 19.2, this number grows very quickly, even by comparison with the number of different undirected graphs and even when V is small. As with undirected graphs, there is a much smaller number of classes of digraphs that are isomorphic to each other (the vertices of one can be relabeled to make it identical to the other), but we cannot take advantage of this reduction because we do not know an efficient algorithm

for digraph isomorphism.

Figure 19.2 Graph Enumeration *While the number of different undirected graphs with V vertices is huge, even when V is small, the number of different digraphs with V vertices is much larger. For undirected graphs, the number is given by the formula* $2^{V(V+1)/2}$; *for digraphs, the formula is* 2^{V^2}.

V	undirected graphs	digraphs
2	8	16
3	64	512
4	1024	65536
5	32768	33554432
6	2097152	68719476736
7	268435456	562949953421312

Certainly, any program will have to process only a tiny fraction of the possible digraphs; indeed, the numbers are so large that we can be certain that virtually all digraphs will not be among those processed by any given program. Generally, it is difficult to characterize the digraphs that we might encounter in practice, so we design our algorithms such that they can handle any possible digraph as input. On the one hand, this situation is not new to us (for example, virtually none of the 1000! permutations of 1000 elements have ever been processed by any sorting program). On the other hand, it is perhaps unsettling to know that, for example, even if all the electrons in the universe could run supercomputers capable of processing 10^{10} graphs per second for the estimated lifetime of the universe, those supercomputers would see far fewer than 10^{-100} percent of the 10-vertex digraphs (see Exercise 19.9).

This brief digression on graph enumeration perhaps underscores several points that we cover whenever we consider the analysis of algorithms and indicates their particular relevance to the study of digraphs. Is it important to design our algorithms to perform well in the worst case, when we are so unlikely to see any particular worst-case digraph? Is it useful to choose algorithms on the basis of average-case analysis, or is that a mathematical fiction? If our intent is to have implementations that perform efficiently on digraphs that we see in practice, we are immediately faced with the problem of characterizing those digraphs. Mathematical models that can convincingly describe the digraphs that we might expect in applications are even more difficult to develop than are models for undirected graphs.

In this chapter, we revisit, in the context of digraphs, a subset of the basic graph-processing problems that we considered in Chapter 17, and we examine several problems that are specific to digraphs. In particular, we look at DFS and several of its applications, includ-

IPR2023-00124
KASPERSKY EX 1036 Page 184

ing *cycle detection* (to determine whether a digraph is a DAG); *topological sort* (to solve, for example, the scheduling problem for DAGs that was just described); and computation of the *transitive closure* and the *strong components* (which have to do with the basic problem of determining whether or not there is a directed `path` between two given vertices). As in other graph-processing domains, these algorithms range from the trivial to the ingenious; they are both informed by and give us insight into the complex combinatorial structure of digraphs.

EXERCISES

● **19.1** Find a large digraph somewhere online—perhaps a transaction graph in some online system, or a digraph defined by links on Web pages.

● **19.2** Find a large DAG somewhere online—perhaps one defined by function-definition dependencies in a large software system or by directory links in a large file system.

19.3 Make a table like *Figure* 19.2, but exclude from the counts graphs and digraphs with self-loops.

19.4 How many digraphs are there that contain V vertices and E edges?

○ **19.5** How many digraphs correspond to each undirected graph that contains V vertices and E edges?

▷ **19.6** How many digits do we need to express the number of digraphs that have V vertices as a base-10 number?

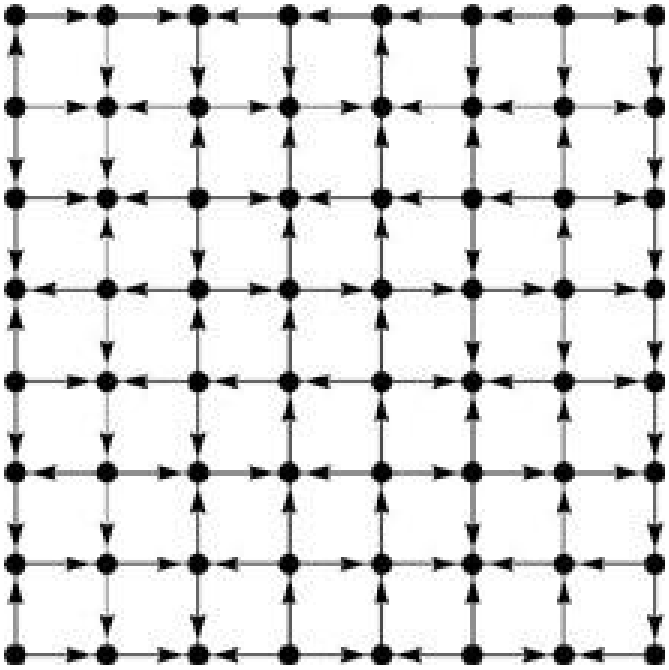
○ **19.7** Draw the nonisomorphic digraphs that contain 3 vertices.

●●● **19.8** How many different digraphs are there with V vertices and E edges if we consider two digraphs to be different only if they are not isomorphic?

○ **19.9** Compute an upper bound on the percentage of 10-vertex digraphs that could ever be examined by any computer, under the assumptions described in the text and the additional ones that the universe has less than 10^{80} electrons and that the age of the universe will be less than 10^{20} years.

Figure 19.3 A Grid Digraph *This small digraph is similar to the large grid network that we first considered in Chapter 1, except that it has a directed edge on every grid line, with the direction randomly chosen. Even though the graph has relatively few nodes, its connectivity properties are not readily apparent. Is there a directed* `path` *from the upper left corner to the lower right corner?*

IPR2023-00124
KASPERSKY EX 1036 Page 185



19.1 GLOSSARY AND RULES OF THE GAME

Our definitions for digraphs are nearly identical to those in Chapter 17 for undirected graphs (as are some of the algorithms and programs that we use), but they are worth restating. The slight differences in the wording to account for edge directions imply structural properties that will be the focus of this chapter.

Definition 19.1 *A* **digraph** (*or* **directed graph**) *is a set of* **vertices** *plus a set of* **directed edges** *that connect ordered pairs of vertices (with no duplicate edges). We say that an edge goes* **from** *its first vertex* **to** *its second vertex.*

As we did with undirected graphs, we disallow duplicate edges in this definition but reserve the option of allowing them when convenient for various applications and implementations. We explicitly allow self-loops in digraphs (and usually adopt the convention that every vertex has one) because they play a critical role in the basic algorithms.

Definition 19.2 *A* **directed path** *in a digraph is a list of vertices in which there is a (directed) digraph edge connecting each vertex in the list to its successor in the list. We say that a vertex t is* **reachable** *from a vertex s if there is a directed* `path` *from s to t.*

We adopt the convention that each vertex is reachable from itself and normally implement that assumption by ensuring that self-loops are present in our digraph representations.

Understanding many of the algorithms in this chapter requires understanding the con-

IPR2023-00124
KASPERSKY EX 1036 Page 186

nectivity properties of digraphs and the effect of these properties on the basic process of moving from one vertex to another along digraph edges. Developing such an understanding is more complicated for digraphs than for undirected graphs. For example, we might be able to tell at a glance whether a small undirected graph is connected or contains a cycle; these properties are not as easy to spot in digraphs, as indicated in the typical example illustrated in *Figure* 19.3.

While examples like this highlight differences, it is important to note that what a human considers difficult may or may not be relevant to what a program considers difficult—for instance, writing a DFS function to find cycles in digraphs is no more difficult than for undirected graphs. More important, digraphs and graphs have essential structural differences. For example, the fact that *t* is reachable from *s* in a digraph indicates nothing about whether *s* is reachable from *t*. This distinction is obvious, but critical, as we shall see.

As mentioned in Section 17.3, the representations that we use for digraphs are essentially the same as those that we use for undirected graphs. Indeed, they are more straightforward because we represent each edge just once, as illustrated in *Figure* 19.4. In the adjacency-list representation, an edge `s-t` is represented as a list node containing `t` in the linked list corresponding to `s`. In the adjacency-matrix representation, we need to maintain a full *V* -by- *V* array and to represent an edge `s-t` by a 1 bit in row `s` and column `t`. We do not put a 1 bit in row `t` and column `s` unless there is also an edge `t-s`. In general, the adjacency matrix for a digraph is not symmetric about the diagonal. In both representations, we typically include self-loops (representations of `s-s` for each vertex `s`).

There is no difference in these representations between an undirected graph and a directed graph with self-loops at every vertex and two directed edges for each edge connecting distinct vertices in the undirected graph (one in each direction). Thus, we can use the algorithms that we develop in this chapter for digraphs to process undirected graphs, provided that we interpret the results appropriately. In addition, we use the programs that we considered in Chapter 17 as the basis for our digraph programs, taking care to remove references to and implicit assumptions about the second representation of each edge that is not a self-loop. For example, to adapt for digraphs our programs for generating, building, and showing graphs (Programs 17.1 through 17.9), we remove the statement

Figure 19.4 Digraph Representations *The adjacency-array and adjacency-lists representations of a digraph have only one representation of each edge, as illustrated in the adjacency array (top) and adjacency lists (bottom) representation of the graph depicted in Figure 19.1. These representations both include self-loops at every vertex, which is typical for digraph-processing algorithms.*

IPR2023-00124
KASPERSKY EX 1036 Page 187

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

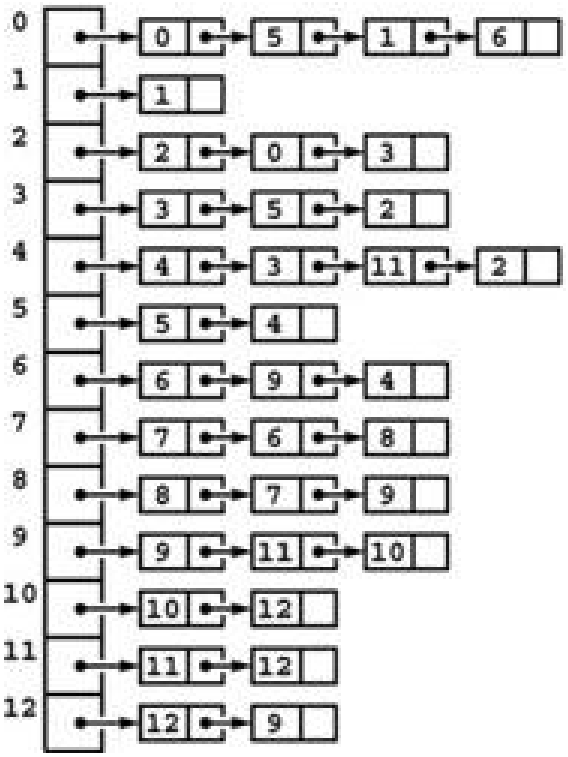
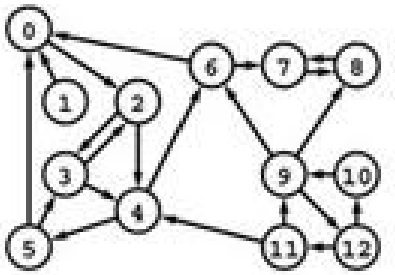
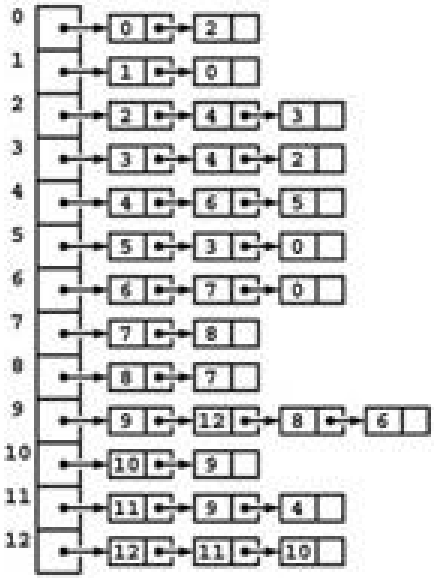


Figure 19.5 Digraph Reversal *Reversing the edges of a digraph corresponds to transposing the adjacency matrix but requires rebuilding the adjacency lists* (see Figures 19.1 and 19.4).



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



Program 19.1 Reversing a digraph (adjacency lists)

Given a digraph represented with adjacency lists, this function creates a new adjacency-lists representation of a digraph that has the same vertices and edges but with the edge directions reversed.

```
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;
  }
```

```
G->adj[w] = NEW (v, G->adj[w]);
```

from the function `GRAPHinsertE` in the adjacency-lists version (Pro-gram 17.6), remove the references to `G->adj[w][v]` from the functions `GRAPHinsertE` and `GRAPHremoveE` in the adjacency-matrix version (*Program* 17.3), and make the appropriate adjustments to `GRAPHedges` in both versions.

The *indegree* of a vertex in a digraph is the number of directed edges that lead to that vertex. The *outdegree* of a vertex in a digraph is the number of directed edges that emanate from that vertex. No vertex is reachable from a vertex of outdegree 0, which is called a *sink*; a vertex of indegree 0, which is called a *source*, is not reachable from any other vertex. A digraph where self-loops are allowed and every vertex has outdegree 1 is called a *map* (a function from the set of integers from 0 to $V - 1$ onto itself). We can easily compute the indegree and outdegree of each vertex, and find sources and sinks, in linear time and space proportional to V, using vertex-indexed arrays (see Exercise 19.19).

The *reverse* of digraph is the digraph that we obtain by switching the direction of all the edges. *Figure* 19.5 shows the reverse and its representations for the digraph of *Figure* 19.1. We use the reverse in digraph algorithms when we need to know from where edges come because our standard representations tell us only where the edges go. For example, indegree and outdegree change roles when we reverse a digraph.

For an adjacency-matrix representation, we could compute the reverse by making a copy of the array and transposing it (interchanging its rows and columns). If we know that the graph is not going to be modified, we can actually use the reverse without any extra computation by simply interchanging our *references* to the rows and columns when we want to refer to the reverse. That is, an edge `s-t` in a digraph `G` is indicated by a `1` in `G->adj[s][t]`, so, if we were to compute the reverse `R` of `G`, it would have a `1` in `R->adj[t][s];` we do not need to do so, however, because, if we want to check whether there is an edge from `s` to `t` in the reverse of `G`, we can just check `G->adj[t][s]`. This

opportunity may seem obvious, but it is often overlooked. For an adjacency-lists representation, the reverse is a completely different data structure, and we need to take time proportional to the number of edges to build it, as shown in *Program* 19.1.

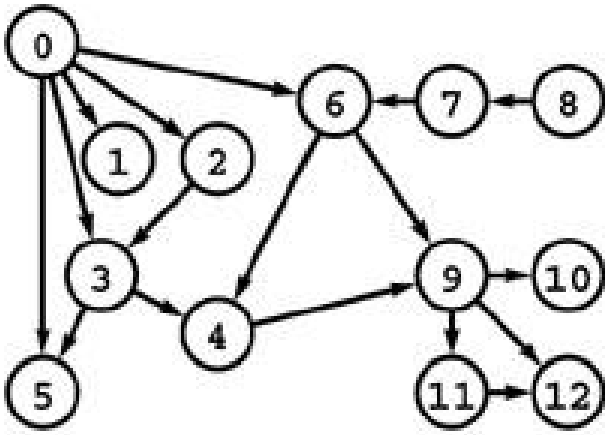
Yet another option, which we shall address in Chapter 22, is to maintain two representations of each edge, in the same manner as we do for undirected graphs (see Section 17.3) but with an extra bit that indicates edge direction. For example, to use this method in an adjacency-lists representation we would represent an edge `s-t` by a node for `t` on the adjacency list for `s` (with the direction bit set to indicate that to move from `s` to `t` is a forward traversal of the edge) and a node for `s` on the adjacency list for `t` (with the direction bit set to indicate that to move from `t` to `s` is a backward traversal of the edge). This representation supports algorithms that need to traverse edges in digraphs in both directions. It is also generally convenient to include pointers connecting the two representations of each edge in such cases. We defer considering this representation in detail to Chapter 22, where it plays an essential role.

In digraphs, by analogy to undirected graphs, we speak of directed cycles, which are directed paths from a vertex back to itself, and simple directed paths and cycles, where the vertices and edges are distinct. Note that `s-t-s` is a cycle of length 2 in a digraph but that cycles in undirected graphs must have at least three distinct vertices.

In many applications of digraphs, we do not expect to see any cycles, and we work with yet another type of combinatorial object.

Figure 19.6 A Directed Acyclic Graph (DAG) *This digraph has no cycles, a property that is not immediately apparent from the edge list or even from examining its drawing.*

IPR2023-00124
KASPERSKY EX 1036 Page 191



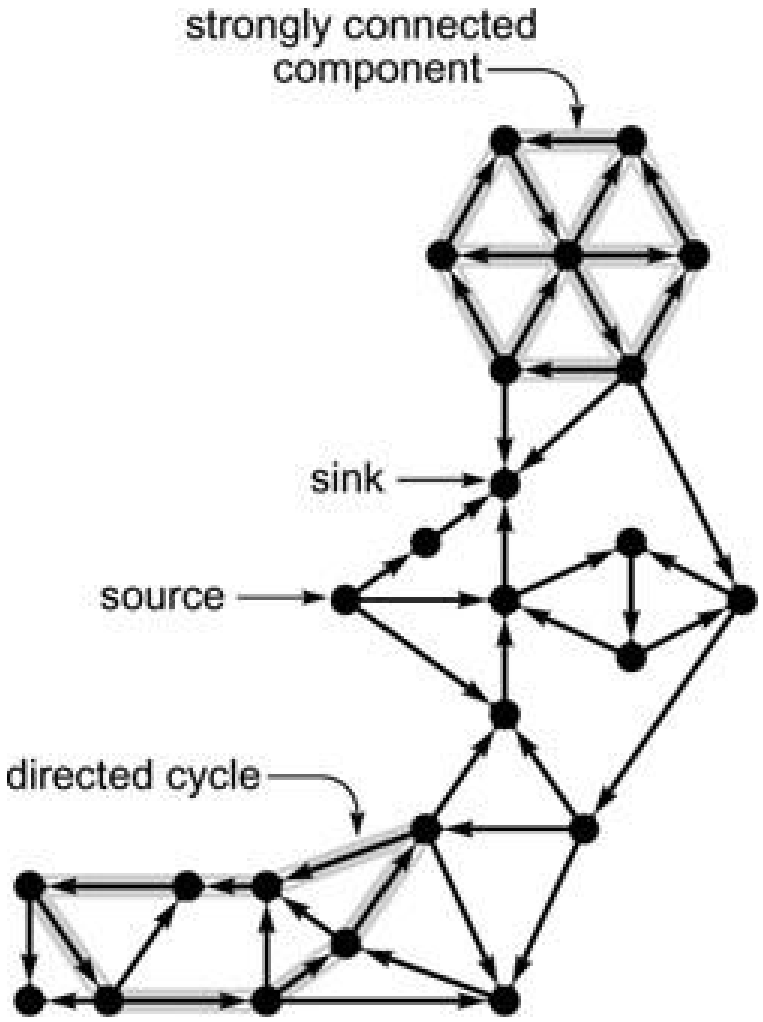

```
2-3  11-12  3-5  6-4
0-6   9-12  8-7  6-9
0-1   9-10  5-4  7-6
2-0   9-11  0-5
```

Definition 19.3 *A* **directed acyclic graph (DAG)** *is a digraph with no directed cycles.*

We expect DAGs, for example, in applications where we are using digraphs to model precedence relationships. DAGs not only arise naturally in these and other important applications, but also, as we shall see, in the study of the structure of general digraphs. A sample DAG is illustrated in *Figure* 19.6.

Directed cycles are therefore the key to understanding connectivity in digraphs that are not DAGs. An undirected graph is connected if there is a `path` from every vertex to every other vertex; for digraphs, we modify the definition as follows:

Figure 19.7 Digraph Terminology *Sources (vertices with no edges coming in) and sinks (vertices with no edges going out) are easy to identify in digraph drawings like this one, but directed cycles and strongly connected components are more difficult to identify. What is the longest directed cycle in this digraph? How many strongly connected components with more than one vertex are there?*



Definition 19.4 *A digraph is* **strongly connected** *if every vertex is reachable from every vertex.*

The graph in *Figure* 19.1 is not strongly connected because, for example, there are no directed paths from vertices 9 through 12 to any of the other vertices in the graph.

As indicated by *strong*, this definition implies a stronger relationship between each pair of vertices than reachability. In any digraph, we say that a pair of vertices `s` and `t` are *strongly connected* or *mutually reachable* if there is a directed `path` from `s` to `t` *and* a directed `path` from `t` to `s`. (Our convention that each vertex is reachable from itself implies that each vertex is strongly connected to itself.) A digraph is strongly connected if and only if all pairs of vertices are strongly connected. The defining property of strongly connected digraphs is one that we take for granted in connected undirected graphs: if there is a `path` from `s` to `t`, then there is a `path` from `t` to `s`. In the case of undirected graphs, we know this fact because the *same* `path` fits the bill, traversed in the other direction; in digraphs, it must be a different path.

IPR2023-00124
KASPERSKY EX 1036 Page 193

Another way of saying that a pair of vertices is strongly connected is to say that they lie on some directed cyclic path. Recall that we use the term *cyclic path* instead of *cycle* to indicate that the `path` does not need to be simple. For example, in *Figure* 19.1, `5` and `6` are strongly connected because `6` is reachable from `5` via the directed `path` `5-4-2-0-6` and `5` is reachable from `6` via the directed `path` `6-4-3-5`; and these paths imply that `5` and `6` lie on the directed cyclic `path` `5-4-2-0-6-4-3-5`, but they do not lie on any (simple) directed cycle. Note that no DAG that contains more than one vertex is strongly connected.

Like simple connectivity in undirected graphs, this relation is transitive: If `s` is strongly connected to `t` and `t` is strongly connected to u, then `s` is strongly connected to u. Strong connectivity is an equivalence relation that divides the vertices into equivalence classes containing vertices that are strongly connected to each other. (See section 19.4 for a detailed discussion of equivalence relations.) Again, strong connectivity provides a property for digraphs that we take for granted with respect to connectivity in undirected graphs.

Property 19.1 *A digraph that is not strongly connected comprises a set of* **strongly connected components** (**strong components**, *for short), which are maximal strongly connected subgraphs, and a set of directed edges that go from one component to another.*

Proof: Like components in undirected graphs, strong components in digraphs are induced subgraphs of subsets of vertices: Each vertex is in exactly one strong component. To prove this fact, we first note that every vertex belongs to at least one strong component, which contains (at least) the vertex itself. Then we note that every vertex belongs to at most one strong component: If a vertex were to belong to two different ones, then there would be paths through that vertex connecting vertices in those components to each other, in both directions, which contradicts the maximality of both components.■

For example, a digraph that consists of a single directed cycle has just one strong component. At the other extreme, each vertex in a DAG is a strong component, so each edge in a DAG goes from one component to another. In general, not all edges in a digraph are in the strong components. This situation is in contrast to the analogous situation for connected components in undirected graphs, where every vertex *and* every edge belongs to some connected component, but similar to the analogous situation for edge-connected components in undirected graphs. The strong components in a digraph are connected by edges that go from a vertex in one component to a vertex in another, but do not go back again.

Property 19.2 *Given a digraph D, define another digraph K (D) with one vertex corresponding to each strong component of D and one edge in K (D) corresponding to each edge in D that connects vertices in different strong components (connecting the vertices*

in K that correspond to the strong components that it connects in D). Then, K (D) is a DAG (which we call the **kernel DAG** *of D*).

Proof: If *K* (*D*) were to have a directed cycle, then vertices in two different strong components of *D* would fall on a directed cycle, and that would be a contradiction.■

Figure 19.8 shows the strong components and the kernel DAG for a sample digraph. We look at algorithms for finding strong components and building kernel DAGs in section 19.6.

From these definitions, properties, and examples, it is clear that we need to be precise when referring to paths in digraphs. We need to consider at least the following three situations:

Connectivity We reserve the term *connected* for undirected graphs. In digraphs, we might say that two vertices are connected if they are connected in the undirected graph defined by ignoring edge directions, but we generally avoid such usage.

Reachability In digraphs, we say that vertex *t* is reachable from vertex *s* if there is a directed `path` from *s* to *t*. We generally avoid the term *reachable* when referring to undirected graphs, although we might consider it to be equivalent to *connected* because the idea of one vertex being reachable from another is intuitive in certain undirected graphs (for example, those that represent mazes).

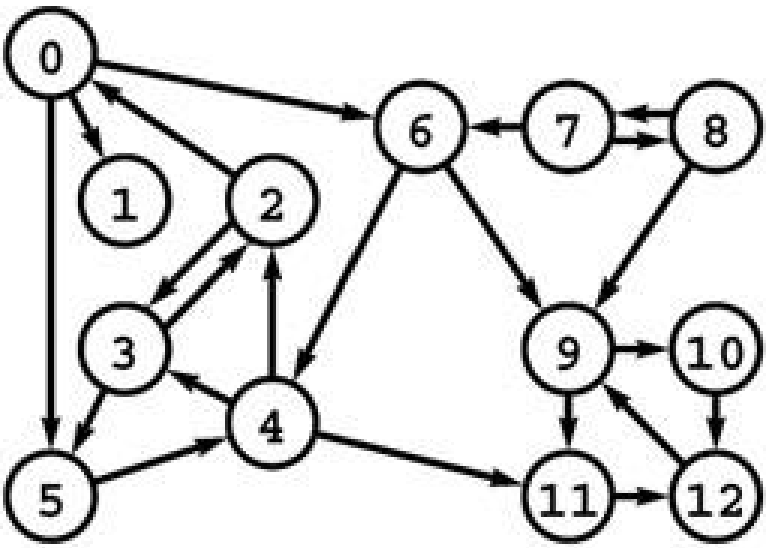
Strong connectivity Two vertices in a digraph are strongly connected if they are mutually reachable; in undirected graphs, two connected vertices imply the existence of paths from each to the other. Strong connectivity in digraphs is similar in certain ways to edge connectivity in undirected graphs.

We wish to support digraph ADT operations that take two vertices `s` and `t` as arguments and allow us to test whether

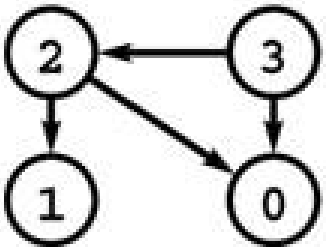
- `t` is reachable from `s`
- `s` and `t` are strongly connected (mutually reachable)

What resource requirements are we willing to expend for these operations? As we saw in Section 17.5, DFS provides a simple solution for connectivity in undirected graphs that takes time proportional to *V*, but if we are willing to invest preprocessing time proportional to *V* + *E* and space proportional to *V*, we can answer connectivity queries in constant time. Later in this chapter, we examine algorithms for strong connectivity that have these same performance characteristics.

Figure 19.8 Strong Components And Kernel Dag *This digraph (top) consists of four strong components, as identified (with arbitrary integer labels) by the vertex-indexed array* `sc` *(center). Component* `0` *consists of the vertices* `9, 10, 11` *and* `12`; *component* `1` *consists of the single vertex* `1`; *component* `2` *consists of the vertices* `0, 2, 3, 4, 5`, *and* `6`; *and component* `3` *consists of the vertices* `7` *and* `8`. *If we draw the graph defined by the edges between different components, we get a DAG (bottom).*



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



But our primary aim is to address the fact that *reachability* queries in digraphs are more difficult to handle than connectivity or strong connectivity queries. In this chapter, we examine classical algorithms that require preprocessing time proportional to VE and space proportional to V^2, develop implementations that can achieve constant-time reachability queries with linear space and preprocessing time for some di-graphs, and study the difficulty of achieving this optimal performance for all digraphs.

Exercises

IPR2023-00124
KASPERSKY EX 1036 Page 196

▷ **19.10** Give the adjacency-lists structure that is built by *Program* 17.6, modified as described in the text to process digraphs, for the digraph

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

▷ **19.11** Implement an ADT for sparse digraphs, based on *Program* 17.6, modified as described in the text. Include a random-digraph generator based on *Program* 17.7. Write a client program to generate random digraphs for a well-chosen set of values of V and E, such that you can use it to run meaningful empirical tests on digraphs drawn from this model.

▷ **19.12** Implement an ADT for dense digraphs, based on *Program* 17.3, modified as described in the text. Include a random-digraph generator based on *Program* 17.8. Write a client program to generate random graphs for a well-chosen set of values of V and E, such that you can use it to run meaningful empirical tests on graphs drawn from this model.

○ **19.13** Write a program that generates random digraphs by connecting vertices arranged in a $\sqrt{V}$-by-$\sqrt{V}$ grid to their neighbors, with edge directions randomly chosen (see *Figure* 19.3).

○ **19.14** Augment your program from Exercise 19.13 to add R extra random edges (all possible eges equally likely). For large R, shrink the grid so that the total number of edges remains about V. Test your program as described in Exercise 19.11.

○ **19.15** Modify your program from Exercise 19.14 such that an extra edge goes from a vertex s to a vertex t with probability inversely proportional to the Euclidean distance between s and t.

○ **19.16** Write a program that generates V random intervals in the unit interval, all of length d, then builds a digraph with an edge from interval s to interval t if and only if at least one of the endpoints of s falls within t (see Exercise 17.73). Determine how to set d so that the expected number of edges is E. Test your program as described in Exercise 19.11 (for low densities) and as described in Exercise 19.12 (for high densities).

● **19.17** Write a program that chooses V vertices and E edges from the real digraph that you found for Exercise 19.1. Test your program as described in Exercise 19.11 (for low densities) and as described in Exercise 19.12 (for high densities).

● **19.18** Write a program that produces each of the possible digraphs with V vertices and E edges with equal likelihood (see Exercise 17.69). Test your program as described in Exercise 19.11 (for low densities) and as described in Exercise 19.12 (for high densities).

IPR2023-00124
KASPERSKY EX 1036 Page 197

▷ **19.19** Add digraph ADT functions that return the number of sources and sinks in the digraph. Modify the adjacency-lists ADT implementation (see Exercise 19.11) such that you can implement the functions in constant time.

○ **19.20** Use your program from Exercise 19.19 to find the average number of sources and sinks in various types of digraphs (see Exercise 19.11–18).

▷ **19.21** Show the adjacency-lists structure that is produced when you use *Program* 19.1 to find the reverse of the digraph

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

○ **19.22** Characterize the reverse of a map.

19.23 Design a digraph ADT that explicitly provides clients with the capability to refer to both a digraph and its reverse, and provide an implementation, using an adjacency-matrix representation.

19.24 Provide an implementation for your ADT in Exercise 19.23 that maintains adjacency-lists representations of both the digraph and its reverse.

▷ **19.25** Describe a family of strongly connected digraphs with V vertices and no (simple) directed cycles of length greater than 2.

○ **19.26** Give the strong components and a kernel DAG of the digraph

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

● **19.27** Give a kernel DAG of the grid digraph shown in *Figure* 19.3.

19.28 How many digraphs have V vertices, all of outdegree k ?

● **19.29** What is the expected number of different adjacency-list representations of a random digraph? *Hint*: Divide the total number of possible representations by the total number of digraphs.

19.2 ANATOMY OF DFS IN DIGRAPHS

We can use our DFS code for undirected graphs from Chapter 18 to visit each edge and each vertex in a digraph. The basic principle behind the recursive algorithm holds: To visit every vertex that can be reached from a given vertex, we mark the vertex as having

IPR2023-00124
KASPERSKY EX 1036 Page 198

been visited, then (recursively) visit all the vertices that can be reached from each of the vertices on its adjacency list.

In an undirected graph, we have two representations of each edge, but the second representation that is encountered in a DFS always leads to a marked vertex and is ignored (see Section 18.2). In a digraph, we have just one representation of each edge, so we might expect DFS algorithms to be more straightforward. But digraphs themselves are more complicated combinatorial objects than undirected graphs, so this expectation is not justified. For example, the search trees that we use to understand the operation of the algorithm have a more complicated structure for digraphs than for undirected graphs. This complication makes digraph-processing algorithms more difficult to devise. For example, as we will see, it is more difficult to make inferences about directed paths in digraphs than it is to make inferences about paths in graphs.

As we did in Chapter 18, we use the term *standard adjacency-lists DFS* to refer to the process of inserting a sequence of edges into a digraph ADT implemented with an adjacency-lists representation (*Program* 17.6, modified to insert just one directed edge), then doing a DFS with Programs 18.3 and 18.2 and the parallel term *standard adjacency-matrix DFS* to refer to the process of inserting a sequence of edges into a digraph ADT implemented with an adjacency-matrix representation (*Program* 17.3, modified to insert just one directed edge), then doing a DFS with Programs 18.3 and 18.1.

For example, *Figure* 19.9 shows the recursive-call tree that describes the operation of a standard adjacency-matrix DFS on the sample digraph in *Figure* 19.1. Just as for undirected graphs, such trees have internal nodes that correspond to calls on the recursive DFS function for each vertex, with links to external nodes that correspond to edges that take us to vertices that have already been seen. Classifying the nodes and links gives us information about the search (and the digraph), but the classification for digraphs is quite different from the classification for undirected graphs.

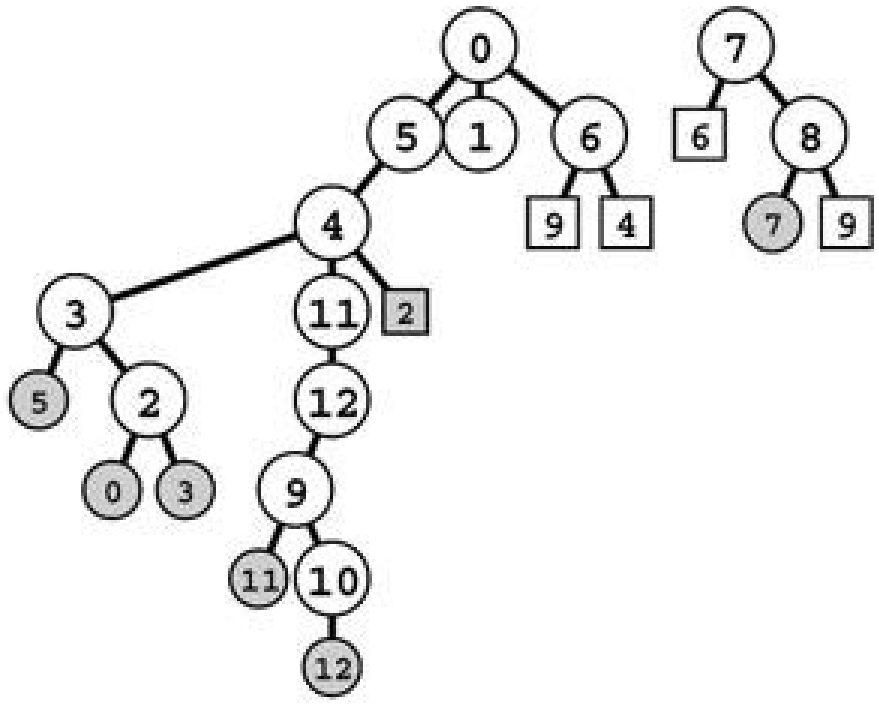
In undirected graphs, we assigned each link in the DFS tree to one of four classes according to whether it corresponded to a graph edge that led to a recursive call and to whether it corresponded to the first or second representation of the edge encountered by the DFS. In digraphs, there is a one-to-one correspondence between tree links and graph edges, and they fall into four distinct classes:

Figure 19.9 DFS Forest for a Digraph *This forest describes a standard adjacency-lists DFS of the sample digraph in Figure 19.1. External nodes represent previously visited internal nodes with the same label; otherwise the forest is a representation of the digraph, with all edges pointing down. There are four types of edges: tree edges, to internal nodes; back edges, to external nodes representing ancestors (shaded circles); down edges, to external nodes representing descen-*

dants (shaded squares); and cross edges, to external nodes representing nodes that are neither ancestors nor descendants (white squares). We can determine the type of edges to visited nodes, by comparing the preorder and postorder numbers (bottom) *of their source and destination:*

pre	post	example	type
`<`	`>`	`4-2`	`down`
`>`	`<`	`2-0`	`back`
`>`	`>`	`7-6`	`cross`

For example, `7-6` *is a cross edge because* `7`*'s preorder and postorder numbers are both larger than* `6`*'s.*



	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

- Those representing a recursive call (*tree* edges)
- Those from a vertex to an ancestor in its DFS tree (*back* edges)
- Those from a vertex to a descendant in its DFS tree (*down* edges)
- Those from a vertex to another vertex that is neither an ancestor nor a descendant in its DFS tree (*cross* edges)

A tree edge is an edge to an unvisited vertex, corresponding to a recursive call in the DFS. Back, cross, and down edges go to visited vertices. To identify the type of a given edge, we use preorder and postorder numbering.

Property 19.3 *In a DFS forest corresponding to a digraph, an edge to a visited node is a back edge if it leads to a node with a higher postorder number; otherwise, it is a cross edge if it leads to a node with a lower preorder number and a down edge if it leads to a node with a higher preorder number.*

Proof: These facts follow from the definitions. A node's ancestors in a DFS tree have lower preorder numbers and higher postorder numbers; its descendants have higher preorder numbers and lower postorder numbers. Both numbers are lower in previously vis-

ited nodes in other DFS trees and both numbers are higher in yet-to-be-visited nodes in other DFS trees but we do not need code that tests for these cases.■

Program 19.2 is a prototype DFS digraph search function that identifies the type of each edge in the digraph. *Figure* 19.10 illustrates its operation on the example digraph of *Figure* 19.1. During the search, testing to see whether an edge leads to a node with a higher postorder number is equivalent to testing whether a postorder number has yet been assigned. Any node for which a preorder number has been assigned but for which a postorder number has not yet been assigned is an ancestor in the DFS tree and will therefore have a postorder number higher than that of the current node.

As we saw in Chapter 17 for undirected graphs, the edge types are properties of the dynamics of the search, rather than of only the graph. Indeed, different DFS forests of the same graph can differ remarkably in character, as illustrated in *Figure* 19.11. For example, even the number of trees in the DFS forest depends upon the start vertex.

Program 19.2 DFS of a digraph

This DFS function for digraphs represented with adjacency lists is instrumented to show the role that each edge in the graph plays in the DFS. It assumes that *Program* 18.3 is augmented to declare and initialize the array post and the counter cntP in the same way as pre and cnt, respectively. See *Figure* 19.10 for sample output and a discussion about implementing show.

```
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++;
  }
```

Despite these differences, several classical digraph-processing algorithms are able to determine digraph properties by taking appropriate action when they encounter the various types of edges during a DFS. For example, consider the following basic problem:

Directed cycle detection Does a given digraph have any directed cycles? (Is the digraph a DAG?) In undirected graphs, any edge to a visited vertex indicates a cycle in the graph; in digraphs, we must restrict our attention to back edges.

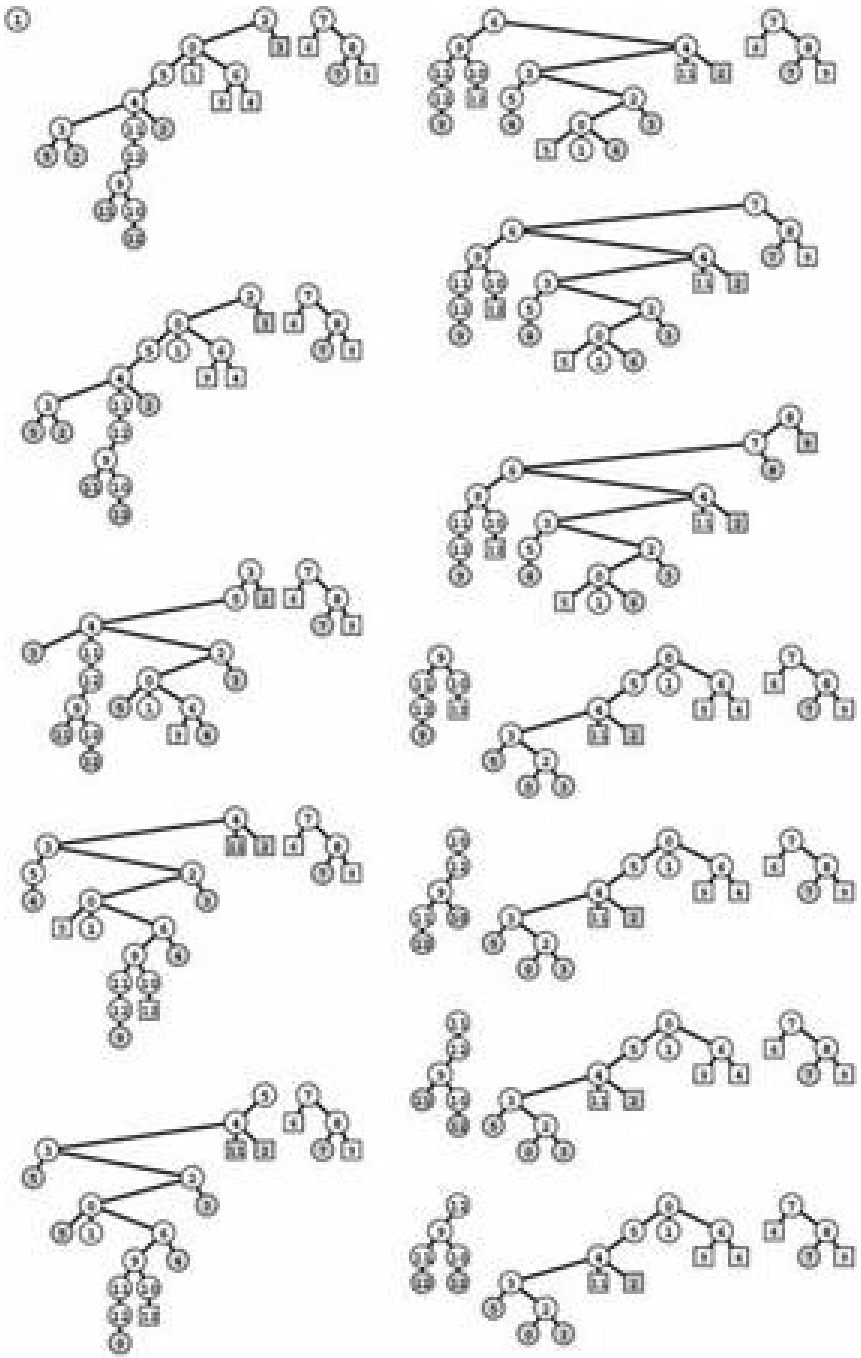
Property 19.4 *A digraph is a DAG if and only if we encounter no back edges when we use DFS to examine every edge.*

Proof: Any back edge belongs to a directed cycle that consists of the edge plus the tree `path` connecting the two nodes, so we will find no back edges when using DFS on a DAG. To prove the converse, we show that, if the digraph has a cycle, then the DFS encounters a back edge. Suppose that `v` is the first of the vertices on the cycle that is visited by the DFS. That vertex has the lowest preorder number of all the vertices on the cycle. The edge that points to it will therefore be a back edge: It will be encountered during the recursive call for `v` (for a proof that it must be, see Property 19.5); and it points from some node on the cycle to `v`, a node with a lower preorder number (see Property 19.3).■

Figure 19.10 Digraph DFS Trace *This DFS trace for the example di-graph in Figure 19.1 corresponds precisely to a preorder walk of the DFS tree depicted in Figure 19.9. As for Figure 17.17 and other similar traces, we can modify Program 19.2 to produce precisely this output by adding a global variable depth that keeps track of the depth of the recursion and an implementation of* `show` *that prints out* `depth` *spaces followed by an appropriate* `printf` *of its arguments.*

```
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
```

Figure 19.11 DFS Forests for a Digraph *These forests describes depth-first search of the same graph as Figure 19.9, when the graph search function checks the vertices (and calls the recursive function for the unvisited ones) in the order* `s, s+1, ..., V-1, 0, 1, ..., s-1` *for each* `s`. *The forest structure is determined both by the search dynamics and the graph structure. Each node has the same children (the nodes on its adjacency list, in order) in every forest. The leftmost tree in each forest contains all the nodes reachable from its root, but reachability inferences about other nodes are complicated because of back, cross, and down edges. Even the number of trees in the forest depends on the starting node, so we do not necessarily have a direct correspondence between trees in the forest and strong components, the way that we did for components in undirected graphs. For example, we see that all vertices are reachable from* `8` *only when we start the DFS at* `8`.



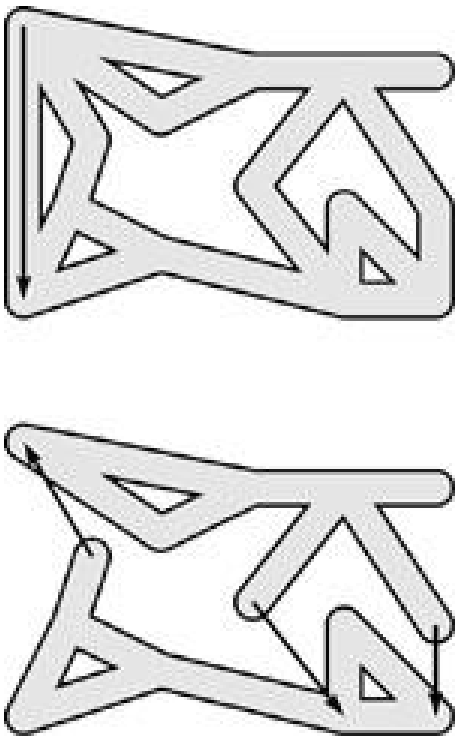
We can convert any digraph into a DAG by doing a DFS and removing any graph edges that correspond to back edges in the DFS. For example, *Figure* 19.9 tells us that removing the edges `2-0, 3-5, 2-3, 9-11, 10-12, 4-2`, and `8-7` makes the digraph in *Figure* 19.1 a DAG. The specific DAG that we get in this way depends on the graph representation and the associated implications for the dynamics of the DFS (see Exercise 19.38). This method is a useful way to generate large arbitrary DAGs randomly (see Exercise 19.79) for use in testing DAG-processing algorithms.

Directed cycle detection is a simple problem, but contrasting the solution just described with the solution that we considered in Chapter 18 for undirected graphs gives insight

into the necessity to consider the two types of graphs as different combinatorial objects, even though their representations are similar and the same programs work on both types for some applications. By our definitions, we seem to be using the same method to solve this problem as for cycle detection in undirected graphs (look for back edges), but the implementation that we used for undirected graphs would not work for digraphs. For example, in Section 18.5 we were careful to distinguish between parent links and back links, since the existence of a parent link does not indicate a cycle (cycles in undirected graphs must involve at least three vertices). But to ignore links back to a node's parents in digraphs would be incorrect; we do consider a doubly-connected pair of vertices in a digraph to be a cycle. Theoretically, we could have defined back edges in undirected graphs in the same way as we have done here, but then we would have needed an explicit exception for the two-vertex case. More important, we can detect cycles in undirected graphs in time proportional to V (see Section 18.5), but we may need time proportional to E to find a cycle in a digraph (see Exercise 19.33).

The essential purpose of DFS is to provide a systematic way to visit all the vertices and all the edges of a graph. It therefore gives us a basic approach for solving reachability problems in digraphs, although, again, the situation is more complicated than for undirected graphs.

Figure 19.12 Decomposing a Digraph *To prove by induction that DFS takes us everywhere reachable from a given node in a digraph, we use essentially the same proof as for Trémaux exploration. The key step is depicted here as a maze* (*top*), *for comparison with Figure 18.4. We break the graph into two smaller pieces* (bottom), *induced by two sets of vertices: those vertices that can be reached by following the first edge from the start vertex without revisiting it* (bottom piece); *and those vertices that cannot be reached by following the first edge without going back through the start vertex* (*top piece*). *Any edge that goes from a vertex in the first set to the start vertex is skipped during the search of the first set because of the mark on the start vertex. Any edge that goes from a vertex in the second set to a vertex in the first set is skipped because all vertices in the first set are marked before the search of the second subgraph begins.*



Single-source reachability Which vertices in a given digraph can be reached from a given start vertex *s*? How many such vertices are there?

Property 19.5 *With a recursive DFS starting at s, we can solve the single-source reachability problem for a vertex s in time proportional to the number of edges in the subgraph induced by the reachable vertices.*

Proof: This proof is essentially the same as the proof of Property 18.1, but it is worth restating to underline the distinction between reachability in digraphs and connectivity in undirected graphs. The property is certainly true for a digraph that has one vertex and no edges. For any digraph that has more than one vertex, we assume the property to be true for all digraphs that have fewer vertices. Now, the first edge that we take from *s* divides the digraph into the subgraphs induced by two subsets of vertices (see *Figure* 19.12): (*i*) the vertices that we can reach by directed paths that begin with that edge and do not otherwise include *s*; and (*ii*) the vertices that we cannot reach with a directed `path` that begins with that edge without returning to *s*. We apply the inductive hypothesis to these subgraphs, noting that there are no directed edges from a vertex in the first subgraph to any vertex other than *s* in the second subgraph (such an edge would be a contradiction because its destination vertex should be in the first subgraph), that directed edges to *s* will be ignored because it has the lowest preorder number, and that all the vertices in the first subgraph have lower preorder numbers than any vertex in the second subgraph, so all directed edges from a vertex in the second subgraph to a vertex in the first subgraph will be ignored.■

IPR2023-00124
KASPERSKY EX 1036 Page 207

By contrast with undirected graphs, a DFS on a digraph does *not* give full information about reachability from any vertex other than the start node, because tree edges are directed and because the search structures have cross edges. When we leave a vertex to travel down a tree edge, we cannot assume that there is a way to get back to that vertex via digraph edges; indeed, there is not, in general. For example, there is no way to get back to 4 after we take the tree edge 4-11 in *Figure* 19.9. Moreover, when we ignore cross and forward edges (because they lead to vertices that have been visited and are no longer active), we are ignoring information that they imply (the set of vertices that are reachable from the destination). For example, following the cross edge 6-9 in *Figure* 19.9 is the only way for us to find out that 10, 11, and 12 are reachable from 6.

To determine which vertices are reachable from another vertex, we apparantly need to start over with a new DFS from that vertex (see *Figure* 19.11). Can we make use of information from previous searches to make the process more efficient for later ones? We consider such reachability questions in section 19.7.

To determine connectivity in undirected graphs, we rely on knowing that vertices are connected to their ancestors in the DFS tree, through (at least) the `path` in the tree. By contrast, the tree `path` goes in the wrong direction in a digraph: There is a directed `path` from a vertex in a digraph to an ancestor only if there is a back edge from a descendant to that or a more distant ancestor. Moreover, connectivity in undirected graphs for each vertex is restricted to the DFS tree rooted at that vertex; in contrast, in digraphs, cross edges can take us to any previously visited part of the search structure, even one in another tree in the DFS forest. For undirected graphs, we were able to take advantage of these properties of connectivity to identify each vertex with a connected component in a single DFS, then to use that information as the basis for a constant-time ADT operation to determine whether any two vertices are connected. For digraphs, as we see in this chapter, this goal is elusive.

We have emphasized throughout this and the previous chapter that different ways of choosing unvisited vertices lead to different search dynamics for DFS. For digraphs, the structural complexity of the DFS trees leads to differences in search dynamics that are even more pronounced than those we saw for undirected graphs. For example, *Figure* 19.11 illustrates that we get marked differences for digraphs even when we simply vary the order in which the vertices are examined in the top-level search function. Only a tiny fraction of even these possibilities is shown in the figure—in principle, each of the $V!$ different orders of examining vertices might lead to different results. In section 19.7, we shall examine an important algorithm that specifically takes advantage of this flexibility, processing the unvisited vertices at the top level (the roots of the DFS trees) in a particular order that immediately exposes the strong components.

Exercises

▷ **19.30** Add code to *Program* 19.2 to print out an indented DFS trace, as described in the commentary to *Figure* 19.10.

19.31 Draw the DFS forest that results from a standard adjacency-lists DFS of the digraph

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

19.32 Draw the DFS forest that results from a standard adjacency-matrix DFS of the digraph

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

○ **19.33** Describe a family of digraphs with V vertices and E edges for which a standard adjacency-lists DFS requires time proportional to E for cycle detection.

▷ **19.34** Show that, during a DFS in a digraph, no edge connects a node to another node whose preorder and postorder numbers are both smaller.

○ **19.35** Show all possible DFS forests for the digraph

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

Tabulate the number of tree, back, cross, and down edges for each forest.

19.36 If we denote the number of tree, back, cross, and down edges by t, b, c, and d, respectively, then we have $t + b + c + d = E$ and $t < V$ for any DFS of any digraph with V vertices and E edges. What other relationships among these variables can you infer? Which of the values are dependent solely on graph properties and which are dependent on dynamic properties of the DFS?

▷ **19.37** Prove that every source in a digraph must be a root of some tree in the forest corresponding to any DFS of that digraph.

○ **19.38** Construct a connected DAG that is a subgraph of *Figure* 19.1 by deleting five edges (see *Figure* 19.11).

19.39 Define a digraph ADT function that provides the capability for a client to check that a digraph is indeed a DAG, and provide a DFS-based implementation for the adja-

IPR2023-00124
KASPERSKY EX 1036 Page 209

cency-matrix representation.

19.40 Use your solution to Exercise 19.39 to estimate (empirically) the probability that a random digraph with V vertices and E edges is a DAG for various types of digraphs (see Exercise 19.11–18).

19.41 Run empirical studies to determine the relative percentages of tree, back, cross, and down edges when we run DFS on various types of digraphs (see Exercise 19.11–18).

19.42 Describe how to construct a sequence of directed edges on V vertices for which there will be no cross or down edges and for which the number of back edges will be proportional to V^2 in a standard adjacency-lists DFS.

○ **19.43** Describe how to construct a sequence of directed edges on V vertices for which there will be no back or down edges and for which the number of cross edges will be proportional to V^2 in a standard adjacency-lists DFS.

19.44 Describe how to construct a sequence of directed edges on V vertices for which there will be no back or cross edges and for which the number of down edges will be proportional to V^2 in a standard adjacency-lists DFS.

○ **19.45** Give rules corresponding to Trémaux traversal for a maze where all the passages are one-way.

● **19.46** Extend your solutions to Exercises 17.55 through 17.60 to include arrows on edges (see the figures in this chapter for examples).

19.3 REACHABILITY AND TRANSITIVE CLOSURE

To develop efficient solutions to reachability problems in digraphs, we begin with the following fundamental definition.

Definition 19.5 *The* **transitive closure** *of a digraph is a digraph with the same vertices but with an edge from s to t in the transitive closure if and only if there is a directed* `path` *from s to t in the given digraph.*

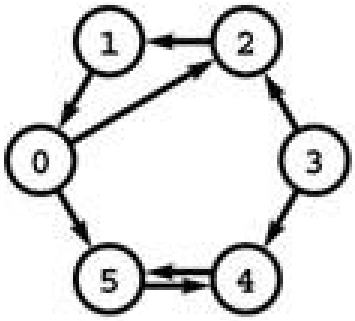
In other words, the transitive closure has an edge from each vertex to all the vertices reachable from that vertex in the digraph. Clearly, the transitive closure embodies all the requisite information for solving reachability problems. *Figure* 19.13 illustrates a small example.

IPR2023-00124
KASPERSKY EX 1036 Page 210

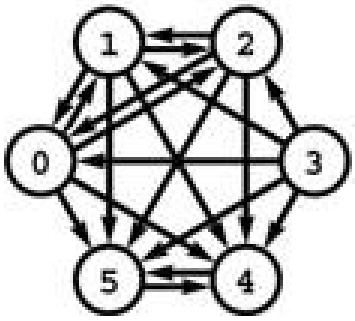
One appealing way to understand the transitive closure is based on adjacency-matrix digraph representations, and on the following basic computational problem.

Boolean matrix multiplication A *Boolean matrix* is a matrix whose entries are all binary values, either `0` or `1`. Given two Boolean matrices *A* and *B*, compute a Boolean product matrix *C*, using the logical *and* and *or* operations instead of the arithmetic operations + and *, respectively.

Figure 19.13 Transitive Closure *This digraph* (*top*) *has just eight directed edges, but its transitive closure* (bottom) *shows that there are directed paths connecting 19 of the 30 pairs of vertices. Structural properties of the digraph are reflected in the transitive closure. For example, rows* `0`, `1`, *and* `2` *in the adjacency matrix for the transitive closure are identical (as are columns* `0`, `1`, *and* `2`*) because those vertices are on a directed cycle in the digraph.*



	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

The textbook algorithm for computing the product of two *V* -by- *V* matrices computes, for each *s* and *t*, the dot product of row *s* in the first matrix and row *t* in the second matrix, as follows:

```
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];
```

In matrix notation, we write this operation simply as $C = A * B$. This operation is defined

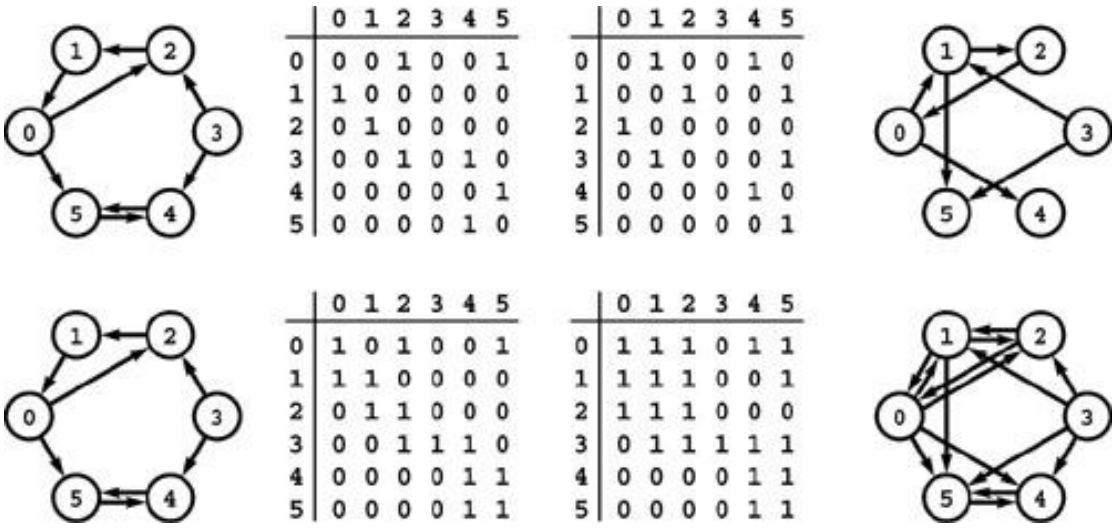
for matrices comprising any type of entry for which `0`, `+`, and `*` are defined. In particular, if we interpret `a+b` to be the logical *or* operation and `a*b` to be the logical *and* operation, then we have Boolean matrix multiplication. In C, we can use the following version:

```
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;
```

To compute `C[s][t]` in the product, we initialize it to `0`, then set it to `1` if we find some value `i` for which both `A[s][i]` and `B[i][t]` are both `1`. Running this computation is equivalent to setting `C[s][t]` to `1` if and only if the result of a bitwise logical *and* of row `s` in `A` with column `t` in `B` has a nonzero entry.

Now suppose that `A` is the adjacency matrix of a digraph A and that we use the preceding code to compute $C = A * A \equiv A^2$ (simply by changing the reference to `B` in the code into a reference to `A`). Reading the code in terms of the interpretation of the adjacency-matrix entries immediately tells us what it computes: For each pair of vertices s and t, we put an edge from s to t in C if and only if there is some vertex i for which there is both a `path` from s to i and a `path` from i to t in A. In other words, directed edges in A^2 correspond precisely to directed paths of length `2` in A. If we include self-loops at every vertex in A, then A^2 also has the edges of A; otherwise, it does not. This relationship between Boolean matrix multiplication and paths in digraphs is illustrated in *Figure* 19.14. It leads immediately to an elegant method for computing the transitive closure of any digraph.

Figure 19.14 Squaring an Adjacency Matrix *If we put 0s on the diagonal of a digraph's adjacency matrix, the square of the matrix represents a graph with an edge corresponding to each* `path` *of length 2* (*top*). *If we put 1s on the diagonal, the square of the matrix represents a graph with an edge corresponding to each* `path` *of length 1 or 2* (bottom).



Property 19.6 *We can compute the transitive closure of a digraph by constructing the latter's adjacency matrix A, adding self-loops for every vertex, and computing* A^V.

Proof: Continuing the argument in the previous paragraph, A^3 has an edge for every `path` of length less than or equal to 3 in the digraph, A^4 has an edge for every `path` of length less than or equal to 4 in the digraph, and so forth. We do not need to consider paths of length greater than V because of the pigeonhole principle: Any such `path` must revisit some vertex (since there are only V of them) and therefore adds no information to the transitive closure because the same two vertices are connected by a directed `path` of length less than V (which we could obtain by removing the cycle to the revisited vertex).■

Figure 19.15 shows the adjacency-matrix powers for a sample digraph converging to transitive closure. This method takes V matrix multiplications, each of which takes time proportional to V^3, for a grand total of V^4. We can actually compute the transitive closure for any digraph with just [lg V] Boolean matrix-multiplication operations: We compute A^2, A^4, A^8,... until we reach an exponent greater than or equal to V. As shown in the proof of Property 19.6, $A^t = A^V$ for any $t > V$; so the result of this computation, which requires time proportional to V^3 lg V, is A^V—the transitive closure.

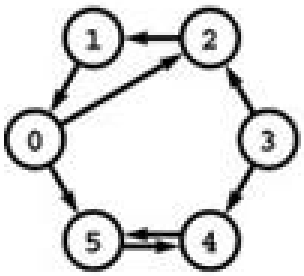
Although the approach just described is appealing in its simplicity, an even simpler method is available. We can compute the transitive closure with just *one* operation of this kind, building up the transitive closure from the adjacency matrix in place, as follows:

```
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;
```

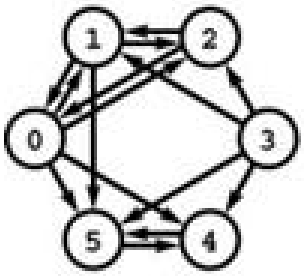
Figure 19.15 Adjacency Matrix Powers and Directed Paths *This sequence*

IPR2023-00124
KASPERSKY EX 1036 Page 213

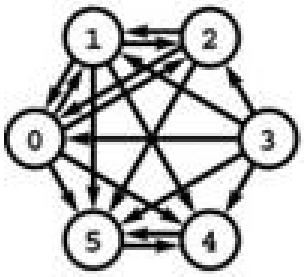
shows the first, second, third, and fourth powers (right, top to bottom) *of the adjacency matrix at the top right, which gives graphs with edges for each of the paths of lengths less than 1, 2, 3, and 4, respectively,* (left, top to bottom) *in the graph that the matrix represents. The bottom graph is the transitive closure for this example, since there are no paths of length greater than 4 that connect vertices not connected by shorter paths.*



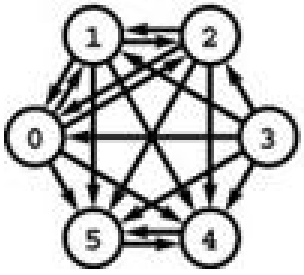
	0	1	2	3	4	5
0	1	0	1	0	0	1
1	1	1	0	0	0	0
2	0	1	1	0	0	0
3	0	0	1	1	1	0
4	0	0	0	0	1	1
5	0	0	0	0	1	1



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

This classical method, invented by S. Warshall in 1962, is the method of choice for computing the transitive closure of dense digraphs. The code is similar to the code that we might try to use to square a Boolean matrix in place: The difference (which is significant!) lies in the order of the for loops.

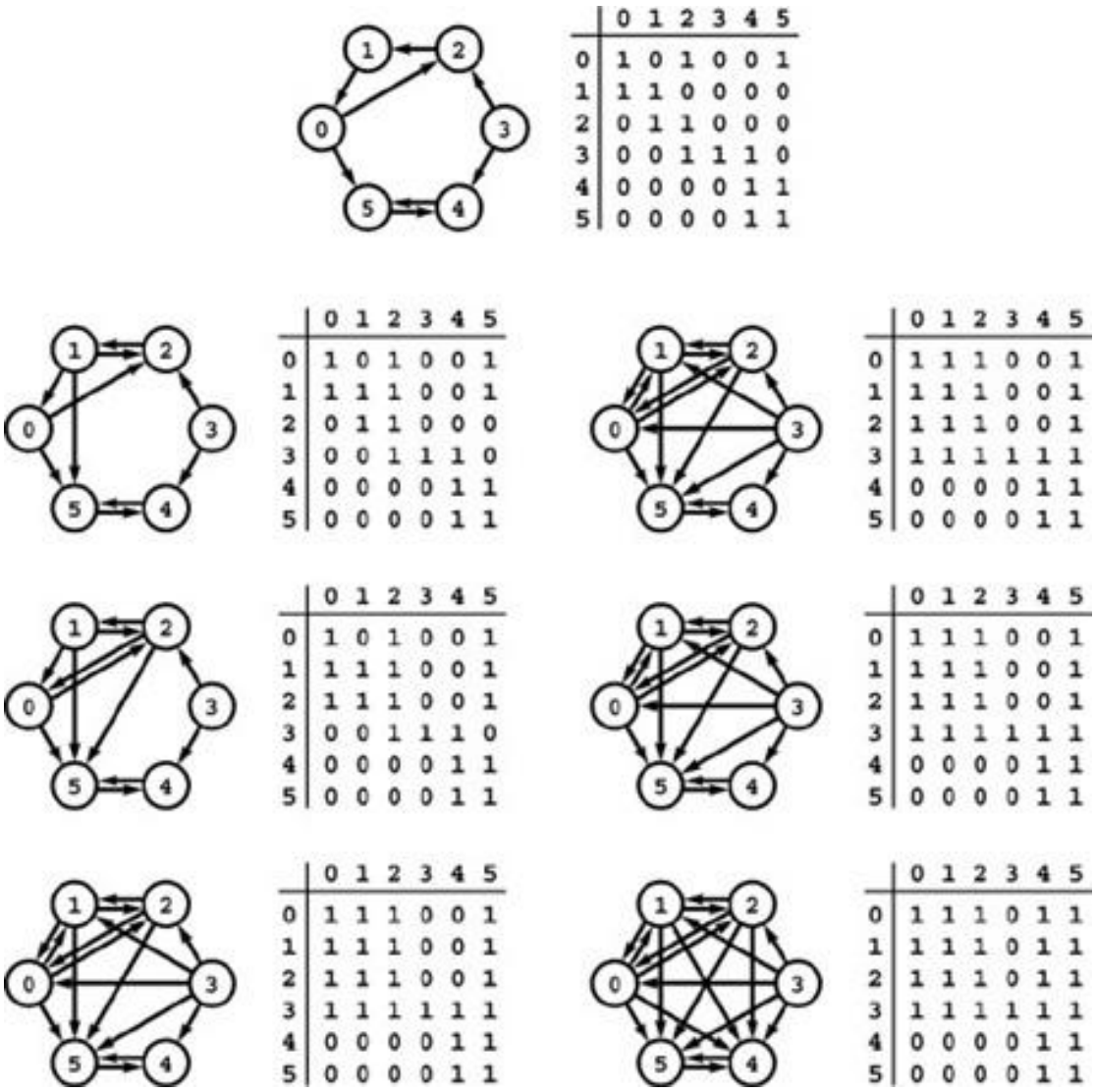
Property 19.7 *With Warshall's algorithm, we can compute the transitive closure of a digraph in time proportional to* V^3.

Proof: The running time is immediately evident from the structure of the code. We prove that it computes the transitive closure by induction on `i`. After the first iteration of the loop, the matrix has a `1` in row `s` and column `t` if and only if we have either the paths `s-t` or `s-0-t`. The second iteration checks all the paths between `s` and `t` that include `1` and perhaps `0`, such as `s-1-t`, `s-1-0-t`, and `s-0-1-t`. We are led to the following inductive hypothesis: the ith iteration of the loop sets the bit in row `s` and column `t` in the matrix to `1` if and only if there is a directed `path` from `s` to `t` in the digraph that does not include any vertices with indices greater than `i` (except possibly the endpoints `s` and `t`). As just argued, the condition is true when `i` is `0`, after the first iteration of the loop. Assuming that it is true for the ith iteration of the loop, there is a `path` from `s` to `t` that does not include any vertices with indices greater than `i+1` if and only if (*i*) there is a `path` from `s` to `t` that does not include any vertices with indices greater than `i`, in which case `A[s][t]` was set on a previous iteration of the loop (by the inductive hypothesis); or (*ii*) there is a `path` from `s` to `i+1` and a `path` from `i+1` to `t`, neither of which includes any vertices with indices greater than `i` (except endpoints), in which case `A[s][i+1]` and `A[i+1][t]` were previously set to `1` (by hypothesis), so the inner loop sets `A[s][t]`.■

We can improve the performance of Warshall's algorithm with a simple transformation of the code: We move the test of `A[s][i]` out of the inner loop because its value does not change as `t` varies. This move allows us to avoid executing the `t` loop entirely when `A[s][i]` is zero. The savings that we achieve from this improvement depends on the digraph and is substantial for many digraphs (see Exercise 19.54 and 19.55). *Program* 19.3 implements this improvement and packages Warshall's method as a pair of ADT functions for digraphs such that clients can preprocess a digraph (compute the transitive closure), then compute the answer to any reachability query in constant time.

Figure 19.16 Warshall's Algorithm *This sequence shows the development of the transitive closure* (bottom) *of an example digraph* (*top*) *as computed with Warshall's algorithm. The first iteration of the loop* (left column, top) *adds the edges* `1-2` *and* `1-5` *because of the paths* `1-0-2` *and* `1-0-5`, *which include vertex* `0` (*but no vertex with a higher number); the second iteration of the loop* (left column, second from top) *adds the edges* `2-0` *and* `2-5` *because of the paths* `2-1-0` *and* `2-1-0-5`, *which include vertex* `1` (*but no vertex with a higher number); and the third iteration of the loop* (left column, bottom) *adds the edges* `0-1, 3-0, 3-1`, *and* `3-5` *because of the paths* `0-2-1, 3-2-1-0, 3-2-1`, *and* `3-2-1-0-5`, *which include vertex* `2` (*but no vertex with a higher number). The right column shows the edges added when paths through* `3, 4`, *and* `5` *are considered. The last iteration of the loop* (right column, bottom) *adds the edges from* `0, 1`, *and* `2`, *to* `4`, *because the only directed paths from those nodes to* `4` *include* `5`, *the*

highest-numbered vertex.



We are interested in pursuing more efficient solutions, particularly for sparse digraphs. We would like to reduce both the preprocessing time and the space because both make the use of Warshall's method prohibitively costly for huge sparse digraphs.

In modern applications, abstract data types provide us with the ability to separate out the idea of an operation from any particular implementation, so we can focus on efficient implementations. For the transitive closure, this point of view leads to a recognition that we do not necessarily need to compute the entire matrix to provide clients with the transitive-closure abstraction. One possibility might be that the transitive closure is a huge sparse matrix, so an adjacency-lists representation is called for because we cannot store the array representation. Even when the transitive closure is dense, client programs might test only a tiny fraction of possible pairs of edges, so computing the whole matrix is wasteful.

Program 19.3 Warshall's algorithm

This ADT implementation of Warshall's algorithm provides clients with the capability to test whether any vertex in a digraph is reachable from any other vertex, after calling GRAPHtc to compute the transitive closure. The array pointer tc in the graph representation is used to store the transitive closure matrix.

```
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]; }
```

We use the term *abstract transitive closure* to refer to an ADT that provides clients with the ability to test reachability after preprocessing a digraph, like *Program* 19.3. In this context, we need to measure an algorithm not just by its cost to compute the transitive closure (preprocessing cost), but also by the space required and the query time achieved. That is, we rephrase Property 19.7 as follows:

Property 19.8 *We can support constant-time reachability testing (abstract transitive closure) for a digraph, using space proportional to V^2 and time proportional to V^3 for preprocessing.*

This property follows immediately from the basic performance characteristics of Warshall's algorithm.■

For most applications, our goal is not just to compute the transitive closure of a digraph quickly, but also to support constant query time for the abstract transitive closure using

far less space and far less preprocessing time than specified in Property 19.8. Can we find an implementation that will allow us to build clients that can afford to handle such digraphs? We return to this question in section 19.8.

There is an intimate relationship between the problem of computing the transitive closure of a digraph and a number of other fundamental computational problems, and that relationship can helps us to understand this problem's difficulty. We conclude this section by considering two examples of such problems.

First, we consider the relationship between the transitive closure and the *all-pairs shortest-paths* problem (see section 18.7). For di-graphs, the problem is to find, for each pair of vertices, a directed `path` with a minimal number of edges. The BFS-based solution for undirected graphs that we considered in section 18.7 also works for digraphs (appropriately modified), but in the present context we are interested in adapting Warshall's algorithm to solve this problem. Shortest paths are the subject of Chapter 21, so we defer considering detailed performance comparisons until then.

Given a digraph, we initialize a V-by-V integer matrix `A` by setting `A[s][t]` to `1` if there is an edge from `s` to `t` andtothe sentinel value `V` if there is no such edge. Our goal is to set `A[s][t]` equal to the length of (the number of edges on) a shortest directed `path` from `s` to `t`, using the sentinel value `V` to indicate that there is no such path. The following code accomplishes this objective:

```
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];
```

This code differs from the version of Warshall's algorithm that we saw just before Property 19.7 in only the if statement in the inner loop. Indeed, in the proper abstract setting, the computations are precisely the same (see Exercise 19.56 and 19.57). Converting the proof of Property 19.7 into a direct proof that this method accomplishes the desired objective is straightforward. This method is a special case of *Floyd's algorithm* for finding shortest paths in weighted graphs (see Chapter 21).

Second, as we have seen, the transitive-closure problem is also closely related to the Boolean matrix-multiplication problem. The basic algorithms that we have seen for both problems require time proportional to V^3, using similar computational schema. Boolean matrix multiplication is known to be a difficult computational problem: Algorithms that are asymptotically faster than the straightforward method are known, but it is debatable

whether the savings are sufficiently large to justify the effort of implementing any of them. This fact is signifi-cant in the present context because we could use a fast algorithm for Boolean matrix multiplication to develop a fast transitive-closure algorithm (slower by just a factor of lg *V*) using the repeated-squaring method illustrated in *Figure* 19.15. Conversely, we have a lower bound on the difficulty of computing the transitive closure:

Property 19.9 *We can use any transitive-closure algorithm to compute the product of two Boolean matrices with at most a constant-factor difference in running time.*

Proof: Given two *V*-by-*V* Boolean matrices *A* and *B*, we construct the following 3 *V* -by-3 *V* matrix:

$$\begin{pmatrix} I & A & 0 \\ 0 & I & B \\ 0 & 0 & I \end{pmatrix}$$

Here, 0 denotes the *V* -by- *V* matrix with all entries equal to 0, and *I* denotes the *V* -by- *V* identity matrix with all entries equal to 0 except those on the diagonal, which are equal to 1. Now, we consider this matrix to be the adjacency matrix for a digraph and compute its transitive closure by repeated squaring:

$$\begin{pmatrix} I & A & 0 \\ 0 & I & B \\ 0 & 0 & I \end{pmatrix}^2 = \begin{pmatrix} I & A & A*B \\ 0 & I & B \\ 0 & 0 & I \end{pmatrix}$$

The matrix on the right-hand side of this equation is the transitive closure because further multiplications give back the same matrix. But this matrix has the *V* -by- *V* product *A* * *B* in its upper-right corner. Whatever algorithm we use to solve the transitive-closure problem, we can use it to solve the Boolean matrix-multiplication problem at the same cost (to within a constant factor).■

The significance of this property depends on the conviction of experts that Boolean matrix multiplication is difficult: Mathematicians have been working for decades to try to learn precisely how difficult it is, and the question is unresolved, with the best known results saying that the running time should be proportional to about $V^{2.5}$ (*see reference section*). Now, if we could find a linear-time (proportional to V^2) solution to the transitive-closure problem, then we would have a linear-time solution to the Boolean matrix-multiplication problem as well. This relationship between problems is known as *reduction*: We say that the Boolean matrix-multiplication problem *reduces to* the transitive-closure

problem (see section 21.6 and Part 8). Indeed, the proof actually shows that Boolean matrix multiplication reduces to finding the paths of length 2 in a digraph.

Despite a great deal of research by many people, no one has been able to find a linear-time Boolean matrix-multiplication algorithm, so we cannot present a simple linear-time transitive-closure algorithm. On the other hand, no one has proved that no such algorithm exists, so we hold open that possibility for the future. In short, we take Property 19.9 to mean that, barring a research breakthrough, we cannot expect the worst-case running time of any transitive-closure algorithm that we can concoct to be proportional to V^2. Despite this conclusion, we can develop fast algorithms for certain classes of digraphs. For example, we have already touched on a simple method for computing the transitive closure that is much faster than Warshall's algorithm for sparse digraphs.

Property 19.10 *With DFS, we can support constant query time for the abstract transitive closure of a digraph, with space proportional to V^2 and time proportional to $V(E + V)$ for preprocessing (computing the transitive closure).*

Proof: As we observed in the previous section, DFS gives us all the vertices reachable from the start vertex in time proportional to E, if we use the adjacency-lists representation (see Property 19.5 and *Figure* 19.11). Therefore, if we run DFS V times, once with each vertex as the start vertex, then we can compute the set of vertices reachable from each vertex—the transitive closure—in time proportional to $V(E + V)$.The same argument holds for any linear-time generalized search (see section 18.8 and Exercise 19.69).■

Program 19.4 DFS-based transitive closure

This program is functionally equivalent to *Program* 19.3. It computes the transitive closure by doing a separate DFS starting at each vertex to compute its set of reachable nodes. Each call on the recursive procedure adds an edge from the start vertex and makes recursive calls to fill the corresponding row in the transitive-closure matrix. The matrix also serves to mark the visited vertices during the DFS.

```
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]; }
```

Program 19.4 is an implementation of this search-based transitive-closure algorithm. The result of running this program on the sample digraph in *Figure* 19.1 is illustrated in the first tree in each forest in *Figure* 19.11. The implementation is packaged in the same way as we packaged Warshall's algorithm in *Program* 19.3: a preprocessing function that computes the transitive closure, and a function that can determine whether any vertex is reachable from any other in constant time by testing the indicated entry in the transitive-closure array.

For sparse digraphs, this search-based approach is the method of choice. For example, if E is proportional to V, then *Program* 19.4 computes the transitive closure in time proportional to V^2. How can it do so, given the reduction to Boolean matrix multiplication that we just considered? The answer is that this transitive-closure algorithm does indeed give an optimal way to multiply *certain types* of Boolean matrices (those with $O(V)$ nonzero entries). The lower bound tells us that we should not expect to find a transitive-closure algorithm that runs in time proportional to V^2 for *all* digraphs, but it does not preclude the possibility that we might find algorithms, like this one, that are faster for *certain classes* of digraphs. If such graphs are the ones that we need to process, the relationship between transitive closure and Boolean matrix multiplication may not be relevant to us.

It is easy to extend the methods that we have described in this section to provide clients with the ability to find a specific `path` connecting two vertices by keeping track of the search tree, as described in Section 17.8. We consider specific ADT implementations of this sort in the context of the more general shortest-paths problems in Chapter 21.

Table 19.1 shows empirical results comparing the elementary transitive-closure algorithms described in this section. The adjacency-lists implementation of the search-based solution is by far the fastest method for sparse digraphs. The implementations all compute an adjacency matrix (of size V^2), so none of them are suitable for huge sparse digraphs.

IPR2023-00124
KASPERSKY EX 1036 Page 221

For sparse digraphs whose transitive closure is also sparse, we might use an adjacency-lists implementation for the closure so that the size of the output is proportional to the number of edges in the transitive closure. This number certainly is a lower bound on the cost of computing the transitive closure, which we can achieve for certain types of digraphs using various algorithmic techniques (see Exercise 19.67 and 19.68). Despite this possibility, we generally view the objective of a transitive-closure computation to be the adjacency-matrix representation, so we can easily answer reachability queries, and we regard transitive-closure algorithms that compute the matrix in time proportional to V^2 as being optimal since they take time proportional to the size of their output.

If the adjacency matrix is symmetric, it is equivalent to an undirected graph, and finding the transitive closure is the same as finding the connected components—the transitive closure is the union of complete graphs on the vertices in the connected components (see Exercise 19.49). Our connectivity algorithms in Section 18.5 amount to an abstract–transitive-closure computation for symmetric digraphs (undirected graphs) that uses space proportional to V and still supports constant-time reachability queries. Can we do as well in general digraphs? Can we reduce the preprocessing time still further? For what types of graphs can we compute the transitive closure in linear time? To answer these questions, we need to study the structure of digraphs in more detail, including, specifically, that of DAGs.

Table 19.1 Empirical study of transitive-closure algorithms This table shows running times that exhibit dramatic performance differences for various algorithms for computing the transitive closure of random digraphs, both dense and sparse. For all but the adjacency-lists DFS, the running time goes up by a factor of 8 when we double V, which supports the conclusion that it is essentially proportional to V^3. The adjacency-lists DFS takes time proportional to VE, which explains the running time roughly increasing by a factor of 4 when we double both V and E (sparse graphs) and by a factor of about 2 when we double E (dense graphs), except that list-traversal overhead degrades performance for high-density graphs.

IPR2023-00124
KASPERSKY EX 1036 Page 222

sparse ($10V$ edges)					dense (250 vertices)				
V	W	W*	A	L	E	W	W*	A	L
25	0	0	1	0	5000	289	203	177	23
50	3	1	2	1	10000	300	214	184	38
125	35	24	23	4	25000	309	226	200	97
250	275	181	178	13	50000	315	232	218	337
500	2222	1438	1481	54	100000	326	246	235	784

Key:
W Warshall's algorithm (Section 19.3)
W* Improved Warshall's algorithm (Program 19.3)
A DFS, adjacency-matrix representation (Exercise 19.64)
L DFS, adjacency-lists representation (Program 19.4)

Exercises

▷ **19.47** What is the transitive closure of a digraph that consists solely of a directed cycle with V vertices?

19.48 How many edges are there in the transitive closure of a digraph that consists solely of a simple directed `path` with V vertices?

▷ **19.49** Give the transitive closure of the undirected graph

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

● **19.50** Show how to construct a digraph with V vertices and E edges with the property that the number of edges in the transitive closure is proportional to t, for any t between E and V^2. As usual, assume that $E > V$.

19.51 Give a formula for the number of edges in the transitive closure of a digraph that is a directed forest as a function of structural properties of the forest.

19.52 Show, in the style of *Figure* 19.15, the process of computing the transitive closure of the digraph

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

through repeated squaring.

19.53 Show, in the style of *Figure* 19.16, the process of computing the transitive closure of the digraph

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

with Warshall's algorithm.

○ **19.54** Give a family of sparse digraphs for which the improved version of Warshall's algorithm for computing the transitive closure (*Program* 19.3) runs in time proportional to *VE*.

○ **19.55** Find a sparse digraph for which the improved version of Warshall's algorithm for computing the transitive closure (*Program* 19.3) runs in time proportional to V^3.

○ **19.56** Develop an ADT for integer matrices with appropriate implementations such that we can have a single client program that encompasses both Warshall's algorithm and Floyd's algorithm. (This exercise is a version of Exercise 19.57 for people who are more familiar with abstract data types than with abstract algebra.)

● **19.57** Use abstract algebra to develop a generic algorithm that encompasses both Warshall's algorithm and Floyd's algorithm. (This exercise is a version of Exercise 19.56 for people who are more familiar with abstract algebra than with abstract data types.)

○ **19.58** Show, in the style of *Figure* 19.16, the development of the all-shortest paths matrix for the example graph in the figure as computed with Floyd's algorithm.

19.59 Is the Boolean product of two symmetric matrices symmetric? Explain your answer.

19.60 Modify Programs 19.3 and 19.4 to provide implementations of a di-graph ADT function that returns the number of edges in the transitive closure of the digraph.

19.61 Implement the function described in Exercise 19.60 by maintaining the count as part of the digraph ADT, and modifying it when edges are added or deleted. Give the cost of adding and deleting edges with your implementation.

▷ **19.62** Add a digraph ADT function for use with Programs 19.3 and 19.4 that returns a vertex-indexed array that indicates which vertices are reachable from a given vertex. You may require the client to have called GRAPHtc for preprocessing.

IPR2023-00124
KASPERSKY EX 1036 Page 224

○ **19.63** Run empirical studies to determine the number of edges in the transitive closure, for various types of digraphs (see Exercise 19.11–18).

▷ **19.64** Implement the DFS-based transitive-closure algorithm for the adjacency-matrix representation.

● **19.65** Consider the bit-array graph representation described in Exercise 17.21. Which method can you speed up by a factor of B (where B is the number of bits per word on your computer): Warshall's algorithm or the DFS-based algorithm? Justify your answer by developing an implementation that does so.

○ **19.66** Develop a representation-independent abstract digraph ADT (see Exercise 17.60), including a transitive-closure function. *Note*: The transitive closure itself should be an abstract digraph without reference to any particular representation.

○ **19.67** Give a program that computes an adjacency-lists representation of the transitive closure of a digraph that is a directed forest in time proportional to the number of edges in the transitive closure.

○ **19.68** Implement an abstract–transitive-closure algorithm for sparse graphs that uses space proportional to T and can answer reachability requests in constant time after preprocessing time proportional to $VE + T$, where T is the number of edges in the transitive closure. *Hint*: Use dynamic hashing.

▷ **19.69** Provide a version of *Program* 19.4 that is based on generalized graph search (see section 18.8).

19.4 EQUIVALENCE RELATIONS AND PARTIAL ORDERS

This section is concerned with basic concepts in set theory and their relationship to abstract–transitive-closure algorithms. Its purposes are to put the ideas that we are studying into a larger context and to demonstrate the wide applicability of the algorithms that we are considering. Mathematically inclined readers who are familiar with set theory may wish to skip to section 19.5 because the material that we cover is elementary (although our brief review of terminology may be helpful); readers who are not familiar with set theory may wish to consult an elementary text on discrete mathematics because our treatment is rather succinct. The connections between digraphs and these fundamental mathematical concepts are too important for us to ignore.

Given a set, a *relation* among its objects is defined to be a set of ordered pairs of the ob-

jects. Except possibly for details relating to parallel edges and self-loops, this definition is the same as our definition of a digraph: Relations and digraphs are different representations of the same abstraction. The mathematical concept is somewhat more powerful because the sets may be infinite, whereas our computer programs all work with finite sets, but we ignore this difference for the moment.

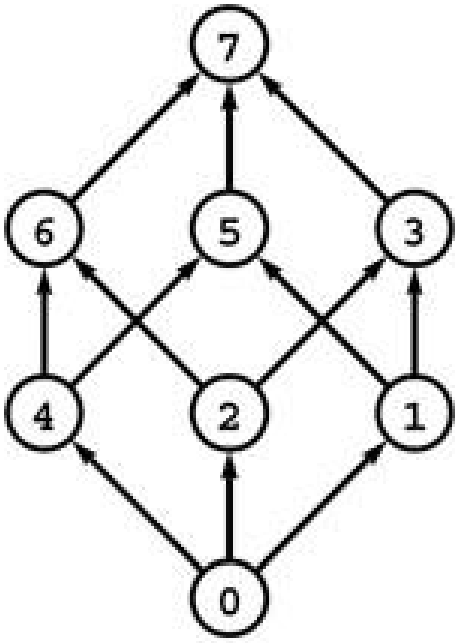
Typically, we choose a symbol R and use the notation sRt as shorthand for the statement "the ordered pair (s, t) is in the relation R." For example, we use the symbol "<" to represent the "less than" relation among numbers. Using this terminology, we can characterize various properties of relations. For example, a relation R is said to be *symmetric* if sRt implies that tRs for all s and t; itissaidtobe *reflexive* if sRs for all s. Symmetric relations are the same as undirected graphs. Reflexive relations correspond to graphs in which all vertices have self-loops; relations that correspond to graphs where no vertices have self-loops are said to be *irreflexive*.

A relation R is said to be *transitive* when sRt and tRu implies that sRu for all s, t, and u. The *transitive closure* of a relation is a well-defined concept; but instead of redefining it in set-theoretic terms, we appeal to the definition that we gave for digraphs in section 19.3. Any relation is equivalent to a digraph, and the transitive closure of the relation is equivalent to the transitive closure of the digraph. The transitive closure of any relation is transitive.

In the context of graph algorithms, we are particularly interested in two particular transitive relations that are defined by further constraints. These two types, which are widely applicable, are known as *equivalence relations* and *partial orders*.

An *equivalence relation* $\equiv$ is a transitive relation that is also reflexive and symmetric. Note that a symmetric, transitive relation that includes each object in some ordered pair must be an equivalence relation: If $s \equiv t$, then $t \equiv s$ (by symmetry) and $s \equiv s$ (by transitivity). Equivalence relations divide the objects in a set into subsets known as *equivalence classes*. Two objects s and t are in the same equivalence class if and only if $s \equiv t$. The following examples are typical equivalence relations:

Figure 19.17 Set-Inclusion DAG *In the DAG at the top, we interpret vertex indices to represent subsets of a set of 3 elements, as shown in the table at the bottom. The transitive closure of this DAG represents the subset inclusion partial order: there is a directed* `path` *between two nodes if and only if the subset represented by the first is included in the subset represented by the second.*



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	$\{a, b, c\}$

Modular arithmetic Any positive integer k defines an equivalence relation on the set of integers, with $s \equiv t \pmod{k}$ if and only if the remainder that results when we divide s by k is equal to the the remainder that results when we divide t by k. The relation is obviously symmetric; a short proof establishes that it is also transitive (see Exercise 19.70) and therefore is an equivalence relation.

Connectivity in graphs The relation "is in the same connected component as" among vertices is an equivalence relation because it is symmetric and transitive. The equivalence classes correspond to the connected components in the graph.

When we build a graph ADT that gives clients the ability to test whether two vertices are in the same connected component, we are implementing an equivalence-relation ADT that provides clients with the ability to test whether two objects are equivalent. In practice, this correspondence is significant because the graph is a succinct representation of the equivalence relation (see Exercise 19.74). In fact, as we saw in Chapters 1 and 18, to

build such an ADT we need to maintain only a single vertex-indexed array.

A *partial order* $\prec$ is a transitive relation that is also irreflexive. As a direct consequence of transitivity and irreflexivity, it is trivial to prove that partial orders are also *asymmetric*: If $s \prec t$ and $t \prec s$, then $s \prec s$ (by transitivity), which contradicts irreflexivity, so we cannot have both $s \prec t$ and $t \prec s$. Moreover, extending the same argument shows that a partial order cannot have a cycle, such as $s \prec t$, $t \prec u$, and $u \prec s$. The following examples are typical partial orders:

Subset inclusion The relation "includes but is not equal to" ($\subset$) among subsets of a given set is a partial order—it is certainly irreflexive, and if $s \subset t$ and $t \subset u$, then certainly $s \subset u$.

Paths in DAGs The relation "can be reached by a nonempty directed `path` from" is a partial order on vertices in DAGs with no self-loops because it is transitive and irreflexive. Like equivalence relations and undirected graphs, this particular partial order is significant for many applications because a DAG provides a succinct implicit representation of the partial order. For example, *Figure* 19.17 illustrates DAGs for subset containment partial orders whose number of edges is only a fraction of the cardinality of the partial order (see Exercise 19.76).

Indeed, we rarely define partial orders by enumerating all their ordered pairs, because there are too many of such pairs. Instead, we generally specify an irreflexive relation (a DAG) and consider its transitive closure. This usage is a primary reason for considering abstract–transitive-closure ADT implementations for DAGs. Using DAGs, we consider examples of partial orders in section 19.5.

A *total order T* is a partial order where either *sTt* or *tT s* for all $s \neq t$. Familiar examples of total orders are the "less than" relation among integers or real numbers and lexicographic ordering among strings of characters. Our study of sorting and searching algorithms in Parts 3 and 4 was based on a total-order ADT implementation for sets. In a total order, there is one and only one way to arrange the elements in the set such that *sT t* whenever *s* is before *t* in the arrangement; in a partial order that is not total, there are many ways to do so. In section 19.5, we examine algorithms for this task.

In summary, the following correspondences between sets and graph models help us to understand the importance and wide applicability of fundamental graph algorithms.

- Relations and digraphs
- Symmetric relations and undirected graphs

- Transitive relations and paths in graphs
- Equivalence relations and paths in undirected graphs
- Partial orders and paths in DAGs

This context places in perspective the types of graphs and algorithms that we are considering and provides one motivation for us to move on to consider basic properties of DAGs and algorithms for processing those DAGs.

Exercises

19.70 Show that "has the same remainder after dividing by k" is a transitive relation (and therefore is an equivalence relation) on the set of integers.

19.71 Show that "is in the same edge-connected component as" is an equivalence relation among vertices in any graph.

19.72 Show that "is in the same biconnected component as" is not an equivalence relation among vertices in all graphs.

19.73 Prove that the transitive closure of an equivalence relation is an equivalence relation, and that the transitive closure of a partial order is a partial order.

▷ **19.74** The cardinality of a relation is its number of ordered pairs. Prove that the cardinality of an equivalence relation is equal to the sum of the squares of the cardinalities of that relation's equivalence classes.

○ **19.75** Using an online dictionary, build a graph that represents the equivalence relation "has k letters in common with" among words. Determine the number of equivalence classes for $k = 1$ through 5.

19.76 The cardinality of a partial order is its number of ordered pairs. What is the cardinality of the subset containment partial order for an n-element set?

▷ **19.77** Show that "is a factor of" is a partial order among integers.

19.5 DAGS

In this section, we consider various applications of directed acyclic graphs (DAGs). We have two reasons to do so. First, because they serve as implicit models for partial orders, we work directly with DAGs in many applications and need efficient algorithms to

IPR2023-00124
KASPERSKY EX 1036 Page 229

process them. Second, these various applications give us insight into the nature of DAGs, and understanding DAGs is essential to understanding general digraphs.

Since DAGs are a special type of digraph, all DAG-processing problems trivially reduce to digraph-processing problems. Although we expect processing DAGs to be easier than processing general di-graphs, we know when we encounter a problem that is difficult to solve on DAGs we should not expect to do any better solving the same problem on general digraphs. As we shall see, the problem of computing the transitive closure lies in this category. Conversely, understanding the difficulty of processing DAGs is important because every digraph has a kernel DAG (see Property 19.2), so we encounter DAGs even when we work with digraphs that are not DAGs.

The prototypical application where DAGs arise directly is called *scheduling*. Generally, solving scheduling problems has to do with arranging for the completion of a set of *tasks*, under a set of *constraints*, by specifying when and how the tasks are to be performed. Constraints might involve functions of the time taken or other resources consumed by the tasks. The most important type of constraints are *precedence constraints*, which specify that certain tasks must be performed before certain others, thus comprising a partial order among the tasks. Different types of additional constraints lead to many different types of scheduling problems, of varying difficulty. Literally thousands of different problems have been studied, and researchers still seek better algorithms for many of them. Perhaps the simplest nontrivial scheduling problem may be stated as follows:

Scheduling Given a set of tasks to be completed, with a partial order that specifies that certain tasks have to be completed before certain other tasks are begun, how can we schedule the tasks such that they are all completed while still respecting the partial order?

In this basic form, the scheduling problem is called *topological sorting*; it is not difficult to solve, as we shall see in the next section by examining two algorithms that do so. In more complicated practical applications, we might need to add other constraints on how the tasks might be scheduled, and the problem can become much more difficult. For example, the tasks might correspond to courses in a student's schedule, with the partial order specifying prerequisites. Topological sorting gives a feasible course schedule that meets the prerequisite requirements, but perhaps not one that respects other constraints that need to be added to the model, such as course conflicts, limitations on enrollments, and so forth. As another example, the tasks might be part of a manufacturing process, with the partial order specifying sequential requirements of the particular process. Topological sorting gives us a way to schedule the tasks, but perhaps there is another way to do so that uses less time, money, or some other resources not included in the model. We examine versions of the scheduling problem that capture more general situations such as these in Chapters 21 and 22.

IPR2023-00124
KASPERSKY EX 1036 Page 230

Despite many similarities with digraphs, a separate ADT for DAGs is appropriate when we are implementing DAG-specific algorithms. In such cases, we use an ADT interface like *Program* 17.1, substituting `DAG` for `GRAPH` everywhere. Sections 19.6 and 19.7 are devoted to implementations of the ADT functions for topological sorting (`DAGts`) and reachability in DAGs (`DAGtc` and `DAGreach`); *Program* 19.13 is an example of a client of this ADT.

Often, our first task is to check whether or not a given DAG indeed has no directed cycles. As we saw in Section 19.2, we can test whether a general digraph is a DAG in linear time, by running a standard DFS and checking that the DFS forest has no back edges. A separate ADT for DAGs should include an ADT function allowing client programs to perform such a check (see Exercise 19.78).

In a sense, DAGs are part tree, part graph. We can certainly take advantage of their special structure when we process them. For example, we can view a DAG almost as we view a tree, if we wish. The following simple program is like a recursive tree traversal:

```
void traverseR (Dag D, int w)
 { link t;
   visit (w);
   for (t = D->adj[w]; t != NULL; t = t->next)
     traverseR (D, t->v);
 }
```

The result of this program is to traverse the vertices of the DAG D as though it were a tree rooted at `w`. For example, the result of traversing the two DAGs in *Figure* 19.18 with this program would be the same. We rarely use a full traversal, however, because we normally want to take advantage of the same economies that save space in a DAG to save time in traversing it (for example, by marking visited nodes in a normal DFS). The same idea applies to a *search*, where we make a recursive call for only one link incident on each vertex. In such an algorithm, the search cost will be the same for the DAG and the tree, but the DAG uses far less space.

Because they provide a compact way to represent trees that have identical subtrees, we often use DAGs instead of trees when we represent computational abstractions. In the context of algorithm design, the distinction between the DAG representation and the tree representation of a program in execution is the essential distinction behind dynamic programming (see, for example, *Figure* 19.18 and Exercise 19.81). DAGs are also widely used in compilers as intermediate representations of arithmetic expressions and programs (see, for example, *Figure* 19.19), and in circuit-design systems as intermediate representations of combinational circuits.

Along these lines, an important example that has many applications arises when we consider binary trees. We can apply the same restriction to DAGs that we applied to trees to define binary trees:

Figure 19.18 DAG Model of Fibonacci Computation *The tree at the top shows the dependence of computing each Fibonacci number on computing its two predecessors. The DAG at the bottom shows the same dependence with only a fraction of the nodes.*

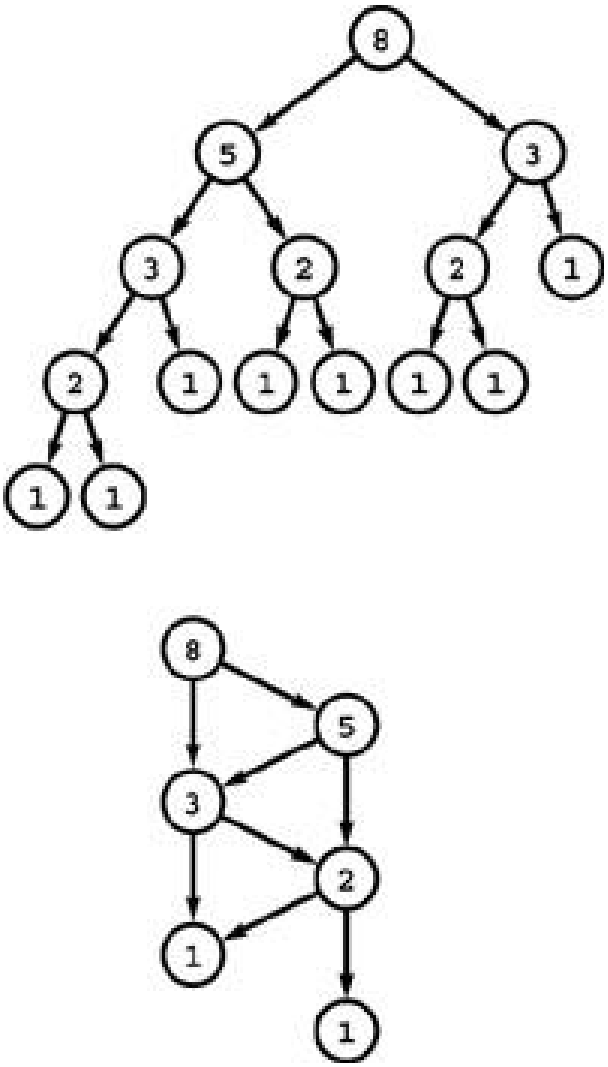
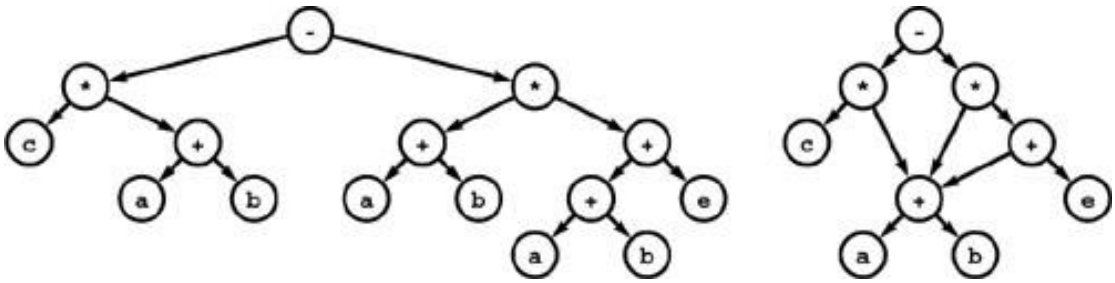


Figure 19.19 DAG Representation of an Arithmetic Expression *Both of these DAGs are representations of the arithmetic expression* `(c* (a+b))- ( (a+b))* ( (a+b)+e))`. *In the binary parse tree at left, leaf nodes represent operands and internal nodes each represent operators to be applied to the expressions represented by their two subtrees (see Figure 5.31). The DAG at right is a more compact representation of the same tree. More important, we can compute the value of the expression in time proportional to the size of the DAG, which is typically significantly less than the size of the tree (see* Exercise 19.114 *and* 19.115).



Definition 19.6 *A* **binary DAG** *is a directed acyclic graph with two edges leaving each node, identified as the left edge and the right edge, either or both of which may be null.*

The distinction between a binary DAG and a binary tree is that in the binary DAG we can have more than one link pointing to a node. As did our definition for binary trees, this definition models a natural representation, where each node is a structure with a left link and a right link that point to other nodes (or are null), subject to only the global restriction that no directed cycles are allowed. Binary DAGs are significant because they provide a compact way to represent binary trees in certain applications. For example, we can compress an existence trie into a binary DAG without changing the search implementation, as shown *Figure* 19.20 and *Program* 19.5.

An equivalent application is to view the trie keys as corresponding to rows in the truth table of a Boolean function for which the function is true (see Exercise 19.87 through 19.90). The binary DAG is a model for an economical circuit that computes the function. In this application, binary DAGs are known as *binary decision diagrams (BDD)*s.

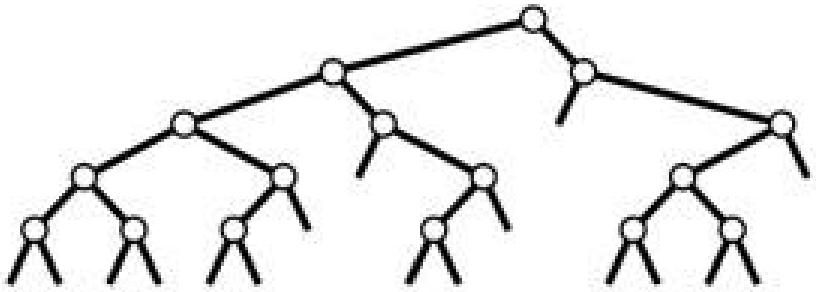
Motivated by these applications, we turn, in the next two sections, to the study of DAG-processing algorithms. Not only do these algorithms lead to efficient and useful DAG ADT function implementations, but also they provide insight into the difficulty of processing digraphs. As we shall see, even though DAGs would seem to be substantially simpler structures than general digraphs, some basic problems are apparently no easier to solve.

Exercises

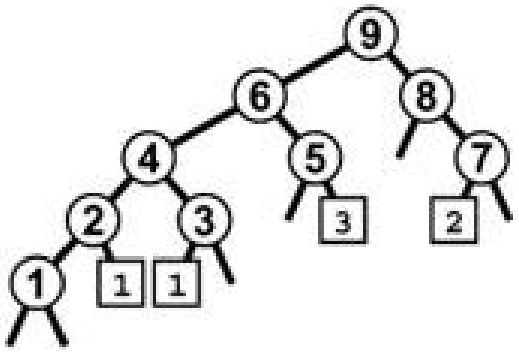
▷ **19.78** Define an ADT interface that is suitable for processing DAGs, and build adjacency-lists and adjacency-matrix implementations. Include an ADT function for verifying that the DAG has no cycles, implemented with DFS.

Program 19.5 Representing a binary tree with a binary DAG

Figure 19.20 Binary Tree Compression *The table of nine pairs of integers at the bottom left is a compact representation of a binary DAG* (bottom right) *that is a compressed version of the binary tree structure at top. Node labels are not explicitly stored in the data structure: The table represents the eighteen edges* `1-0, 1-0, 2-1, 2-1, 3-1, 3-2`, *and so forth, but designates a left edge and a right edge leaving each node (as in a binary tree) and leaves the source vertex for each edge implicit in the table index. An algorithm that depends only upon the tree shape will work effectively on the DAG. For example, suppose that the tree is an existence trie for binary keys corresponding to the leaf nodes, so it represents the keys* `0000, 0001, 0010, 0110, 1100`, *and* `1101`. *A successful search for the key* `1101` *in the trie moves right, right, left, and right to end at a leaf node. In the DAG, the same search goes from* `9` *to* `8` *to* `7` *to* `2` *to* `1`.



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



○ **19.79** Write a program that generates random DAGs by generating random digraphs, doing a DFS from a random starting point, and throwing out the back edges (see Exercise 19.41). Run experiments to decide how to set parameters in your program to expect DAGs with E edges, given V.

▷ **19.80** How many nodes are there in the tree and in the DAG corresponding to *Figure*

IPR2023-00124
KASPERSKY EX 1036 Page 234

19.18 for F_N, the N th Fibonacci number?

19.81 Give the DAG corresponding to the dynamic-programming example for the knapsack model from Chapter 5 (see Figure 5.17).

○ **19.82** Develop an ADT for binary DAGs.

19.83 Can every DAG be represented as a binary DAG (see Property 5.4)?

○ **19.84** Write a function that performs an inorder traversal of a single-source binary DAG. That is, the function should visit all vertices that can be reached via the left edge, then visit the source, then visit all the vertices that can be reached via the right edge.

▷ **19.85** In the style of *Figure* 19.20, give the existence trie and corresponding binary DAG for the keys `01001010 10010101 00100001 11101100 01010001 00100001 00000111 01010011`.

19.86 Implement an ADT based on building an existence trie from a set of 32-bit keys, compressing it as a binary DAG, then using that data structure to support existence queries.

○ **19.87** Draw the BDD for the truth table for the odd parity function of four variables, which is 1 if and only if the number of variables that have the value 1 is odd.

19.88 Write a function that takes a 2^n-bit truth table as argument and returns the corresponding BDD. For example, given the input `1110001000001100`, your program should return a representation of the binary DAG in *Figure* 19.20.

19.89 Write a function that takes a 2^n-bit truth table as argument, computes every permutation of its argument variables, and, using your solution to Exercise 19.88, finds the permutation that leads to the smallest BDD.

● **19.90** Run empirical studies to determine the effectiveness of the strategy of Exercise 19.90 for various Boolean functions, both standard and randomly generated.

19.91 Write a program like *Program* 19.5 that supports common subexpression removal: Given a binary tree that represents an arithmetic expression, compute a binary DAG that represents the same expression with common subexpressions removed.

○ **19.92** Draw all the nonisomorphic DAGs with two, three, four, and five vertices.

●● **19.93** How many different DAGs are there with V vertices and E edges?

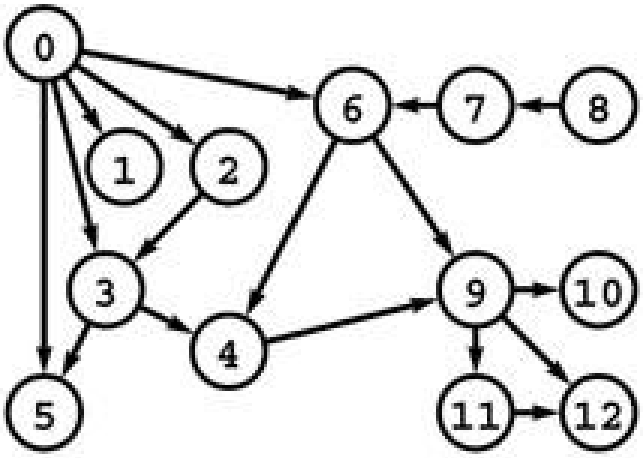
●●● **19.94** How many different DAGs are there with V vertices and E edges, if we consider two DAGs to be different only if they are not isomorphic?

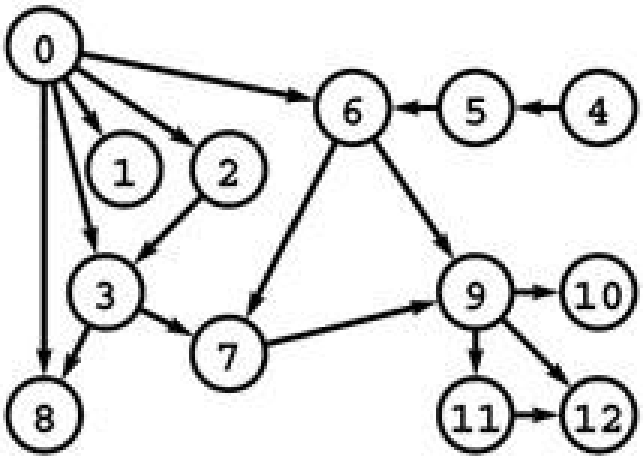
19.6 TOPOLOGICAL SORTING

The goal of topological sorting is to be able to process the vertices of a DAG such that every vertex is processed before all the vertices to which it points. There are two natural ways to define this basic operation; they are essentially equivalent. Both tasks call for a permutation of the integers `0` through `V-1`, which we put in vertex-indexed arrays, as usual.

Topological sort (relabel) Given a DAG, relabel its vertices such that every directed edge points from a lower-numbered vertex to a higher-numbered one (see *Figure* 19.21).

Figure 19.21 Topological Sort (Relabeling) *Given any DAG (top), topological sorting allows us to relabel its vertices so that every edge points from a lower-numbered vertex to a higher-numbered one (bottom). In this example, we relabel* `4, 5, 7` *and* `8` *to* `7, 8, 5` *and* `4`*, respectively, as indicated in the array* `tsI`*. There are many possible labelings that achieve the desired result.*

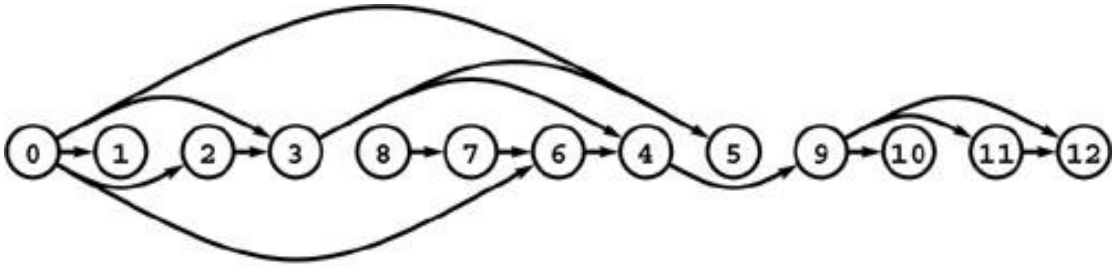




	0	1	2	3	4	5	6	7	8	9	10	11	12
tsI	0	1	2	3	7	8	6	5	4	9	10	11	12

Figure 19.22 Topological Sorting (Rearrangement). *This diagram shows another way to look at the topological sort in Figure 19.21, where we specify a way to rearrange the vertices, rather than relabel them. When we place the vertices in the order specified in the array* `ts`*, from left to right, then all directed edges point from left to right. The inverse of the permutation* `ts` *is the permutation* `tsI` *that specifies the relabeling described in Figure 19.21.*

IPR2023-00124
KASPERSKY EX 1036 Page 237



	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

Topological sort (rearrange) Given a DAG, rearrange its vertices on a horizontal line such that all the directed edges point from left to right (see *Figure* 19.22).

As indicated in *Figure* 19.22, it is easy to establish that the relabeling and rearrangement permutations are inverses of one another: Given a rearrangement, we can obtain a relabeling by assigning the label `0` to the first vertex on the list, `1` to the second label on the list, and so forth. For example, if an array `ts` has the vertices in topologically sorted order, then the loop

```
for (i = 0; i < V; i++) tsI[ts[i]] = i;
```

defines a relabeling in the vertex-indexed array `tsI`. Conversely, if we have the relabeling in an array `tsI`, then we can get the rearrangement with the loop

```
for (i = 0; i < V; i++) ts[tsI[i]] = i;
```

which puts the vertex that would have label `0` first in the list, the vertex that would have label `1` second in the list, and so forth. Most often, we use the term *topological sort* to refer to the rearrangement version of the problem.

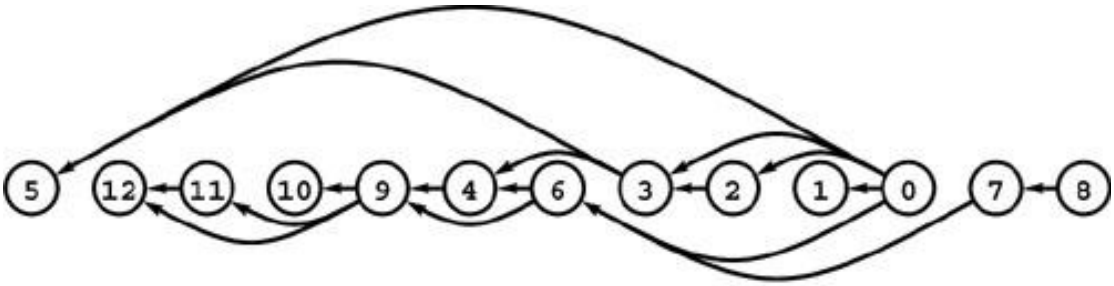
In general, the vertex order produced by a topological sort is not unique. For example,

```
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
```

are all topological sorts of the example DAG in *Figure* 19.6 (and there are many others). In a scheduling application, this situation arises whenever one task has no direct or indirect dependence on another and thus they can be performed either before or after the other (or even in parallel). The number of possible schedules grows exponentially with

the number of such pairs of tasks.

Figure 19.23 Reverse Topological Sort. *In this reverse topological sort of our sample digraph, the edges all point from right to left. Numbering the vertices as specified by the inverse permutation* `tsI` *gives a graph where every edge points from a higher-numbered vertex to a lower-numbered vertex.*



	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

As we have noted, it is sometimes useful to interpret the edges in a digraph the other way around: We say that an edge directed from `s` to `t` means that vertex `s` "depends" on vertex `t`. For example, the vertices might represent terms to be defined in a book, with an edge from `s` to `t` if the definition of `s` uses `t`. In this case, it would be useful to find an ordering with the property that every term is defined before it is used in another definition. Using this ordering corresponds to positioning the vertices in a line such that edges all go from right to left—a *reverse topological sort*. *Figure* 19.23 illustrates a reverse topological sort of our sample DAG.

Now, it turns out that we have already seen an algorithm for reverse topological sorting: our standard recursive DFS! When the input graph is a DAG, a *postorder numbering* puts the vertices in reverse topological order. That is, we number each vertex as the final action of the recursive DFS function, as in the post array in the DFS implementation in *Program* 19.2. As illustrated in *Figure* 19.24, using this numbering is equivalent to numbering the nodes in the DFS forest in postorder. Taking the vertices in this example in order of their postorder numbers, we get the vertex order in *Figure* 19.23—a reverse topological sort of the DAG.

Property 19.11 *Postorder numbering in DFS yields a reverse topological sort for any DAG.*

Proof: Suppose that `s` and `t` are two vertices such that `s` appears before `t` in the postorder numbering even though there is a directed edge `s-t` in the graph. Since we are finished with the recursive DFS for `s` at the time that we assign `s` its number, we have examined, in particular, the edge `s-t`. But if `s-t` were a tree, down, or cross edge, the recursive DFS

for `t` would be complete, and `t` would have a lower number; however, `s-t` cannot be a back edge because that would imply a cycle. This contradiction implies that such an edge `s-t` cannot exist.■

Thus, we can easily adapt a standard DFS to do a topological sort, as shown in *Program 19.6*. Depending on the application, we might wish to package an ADT function for topological sorting that fills in the client-supplied vertex-indexed array with the postorder numbering, or we might wish to return the inverse of that permutation, which has the vertex indices in topologically sorted order. In either case, we might wish to do a forward or reverse topological sort, for a total of four different possibilities that we might wish to handle.

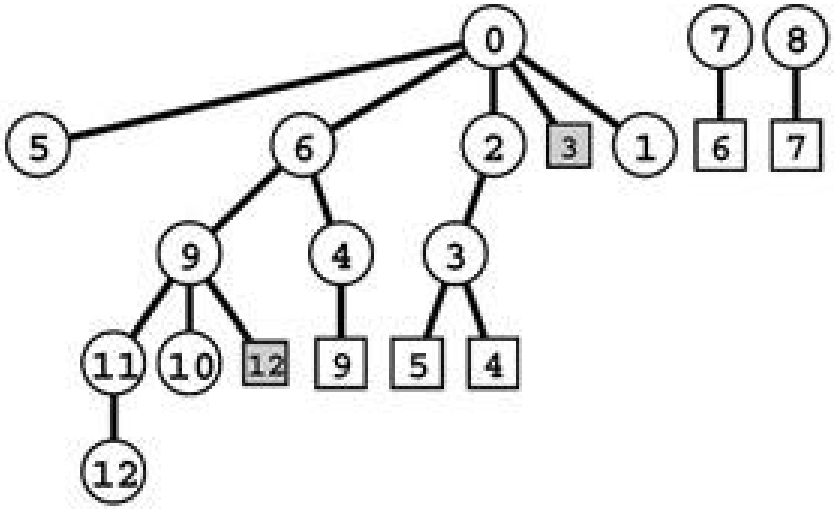
Computationally, the distinction between topological sort and reverse topological sort is not crucial. We have at least three ways to modify DFS with postorder numbering if we want it to produce a proper topological sort:

- Do a reverse topological sort on the reverse of the given DAG.

- Rather than using it as an index for postorder numbering, push the vertex number on a stack as the final act of the recursive procedure. After the search is complete, pop the vertices from the stack. They come off the stack in topological order.

- Number the vertices in reverse order (start at $V - 1$ and count down to 0). If desired, compute the inverse of the vertex numbering to get the topological order.

Figure 19.24 DFS Forest For a DAG *A DFS forest of a digraph has no back edges (edges to nodes with a higher postorder number) if and only if the digraph is a DAG. The non-tree edges in this DFS forest for the DAG of Figure 19.21 are either down edges (shaded squares) or cross edges (unshaded squares). The order in which vertices are encountered in a postorder walk of the forest, shown at the bottom, is a reverse topological sort (see Figure 19.23).*



	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

The proofs that these changes give a proper topological ordering are left for you to do as an exercise (see Exercise 19.99).

To implement the first of the options listed in the previous paragraph for sparse graphs (represented with adjacency lists), we would need to use *Program* 19.1 to compute the reverse graph. Doing so essentially doubles our space usage, which may thus become onerous for huge graphs. For dense graphs (represented with an adjacency matrix), as noted in Section 19.1, we can do DFS on the reverse without using any extra space or doing any extra work, simply by exchanging rows and columns when referring to the matrix, as illustrated in *Program* 19.7.

Program 19.6 Reverse topological sort (adjacency lists)

This version of DFS returns an array that contains vertex indices of a DAG such that the source vertex of every edge appears to the right of the destination vertex (a reverse topological sort). Comparing the last line of `TSdfsR` with, for example, the last line of `dfsR` in *Program* 19.2 is an instructive way to understand the computation of the inverse of the postorder-numbering permutation (see *Figure* 19.23).

```
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;
  }
```

Next, we consider an alternative classical method for topological sorting that is more like breadth-first search (BFS) (see section 18.7). It is based on the following property of DAGs.

Property 19.12 *Every DAG has at least one source and at least one sink.*

Proof: Suppose that we have a DAG that has no sinks. Then, starting at any vertex, we can build an arbitrarily long directed `path` by following any edge from that vertex to any other vertex (there is at least one edge, since there are no sinks), then following another edge from that vertex, and so on. But once we have been to $V + 1$ vertices, we must have seen a directed cycle, by the pigeonhole principle (see Property 19.6), which contradicts the assumption that we have a DAG. Therefore, every DAG has at least one sink. It follows that every DAG also has at least one source: its reverse's sink.■

Program 19.7 Topological sort (adjacency matrix)

This adjacency-array DFS computes a topological sort (not the reverse) because we replace the reference to `a[v][w]` in the DFS by `a[w][v]`, thus processing the reverse graph (*see text*).

```
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;
  }
```

From this fact, we can derive a (relabel) topological-sort algorithm: Label any source with the smallest unused label, then remove it and label the rest of the DAG, using the same algorithm. *Figure* 19.25 is a trace of this algorithm in operation for our sample DAG.

Implementing this algorithm efficiently is a classic exercise in data-structure design (*see reference section*). First, there may be multiple sources, so we need to maintain a queue to keep track of them (any generalized queue will do). Second, we need to identify the sources in the DAG that remains when we remove a source. We can accomplish this task by maintaining a vertex-indexed array that keeps track of the indegree of each vertex. Vertices with indegree 0 are sources, so we can initialize the queue with one scan through the DAG (using DFS or any other method that examines all of the edges). Then, we perform the following operations until the source queue is empty:

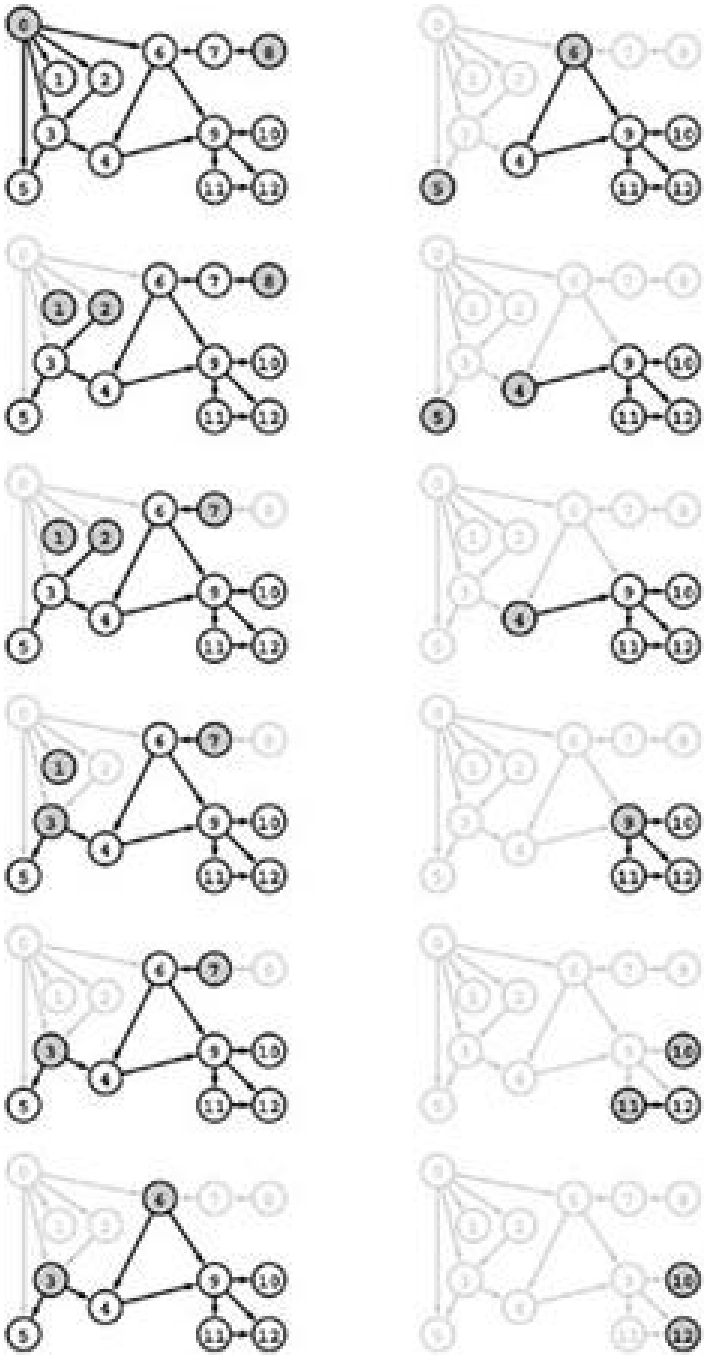
- Remove a source from the queue and label it.
- Decrement the entries in the indegree array corresponding to the destination vertex of each of the removed vertex's edges.
- If decrementing any entry causes it to become 0, insert the corresponding vertex onto the source queue.

Program 19.8 is an implementation of this method, using a FIFO queue, and *Figure* 19.26 illustrates its operation on our sample DAG, providing the details behind the dynamics of the example in *Figure* 19.25.

Figure 19.25 Topologically Sorting a DAG by Removing Sources *Since it is a source (no edges point to it),* `0` *can appear first in a topological sort of this example graph (left, top). If we remove* `0` *(and all the edges that point from it to other vertices), then* `1` *and* `2` *become sources in the resulting DAG* (left, second from top), *which we can then sort using the same algorithm. This figure illustrates the operation of Program 19.8, which picks from among the sources (the shaded nodes in each diagram) using the FIFO discipline, though any of the sources could be chosen at each step. See Figure 19.26 for the contents of the data structures that con-*

IPR2023-00124
KASPERSKY EX 1036 Page 243

trol the specific choices that the algorithm makes. The result of the topological sort illustrated here is the node order `0 8 2 1 7 3 6 5 4 9 11 10 12`.



Program 19.8 Source-queue–based topological sort

IPR2023-00124
KASPERSKY EX 1036 Page 244

This implementation maintains a queue of sources and uses a table that keeps track of the indegree of each vertex in the DAG induced by the vertices that have not been removed from the queue. When we remove a source from the queue, we decrement the indegree entries corresponding to each of the vertices on its adjacency list (and put on the queue any vertices corresponding to entries that become 0). Vertices come off the queue in topologically sorted order.

```
#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);
      }
  }
```

Figure 19.26 Indegree Table and Queue Contents *This sequence depicts the contents of the indegree table* (left) *and the source queue* (right) *during the execution of Program 19.8 on the sample DAG corresponding to Figure 19.25. At any given point in time, the source queue contains the nodes with indegree* `0`*. Reading from top to bottom, we remove the leftmost node from the source queue, decrement the indegree entry corresponding to every edge leaving that node, and add any vertices whose entries become* `0` *to the source queue. For example, the second line of the table reflects the result of removing* `0` *from the source queue, then (because the DAG has the edges* `0-1, 0-2, 0-3, 0-5`*, and* `0-6`*) decrementing the indegree entries corresponding to* `1, 2, 3, 5`*, and* `6` *and adding* `2` *and* `1` *to the source queue (because decrementing made their indegree entries* `0`*). Reading the leftmost entries in the source queue from top to bottom gives a topological ordering for the graph.*

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

The source queue does not empty until every vertex in the DAG is labeled, because the subgraph induced by the vertices not yet labeled is always a DAG, and every DAG has at least one source. Indeed, we can use the algorithm to test whether a graph is a DAG by inferring that there must be a cycle in the subgraph induced by the vertices not yet labeled if the queue empties before all the vertices are labeled (see Exercise 19.106).

Processing vertices in topologically sorted order is a basic technique in processing DAGs. A classic example is the problem of finding the length of the longest `path` in a DAG. Considering the vertices in reverse topologically sorted order, the length of the longest `path` originating at each vertex v is easy to compute: add one to the maximum of the lengths of the longest paths originating at each of the vertices reachable by a single edge from v. The topological sort ensures that all those lengths are known when v is processed, and that no other paths from v will be found afterwards. For example, taking a left-to-right scan of the reverse topological sort shown in *Figure* 19.23 we can quickly compute the following table of lengths of the longest paths originating at each vertex in the sample graph in *Figure* 19.21.

```
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
```

For example, the 6 corresponding to 0 (third column from the right) says that there is a path of length 6 originating at 0, which we know because there is an edge 0-2, we previously found the length of the longest path from 2 to be 5, and no other edge from 0 leads to a node having a longer path.

Whenever we use topological sorting for such an application, we have several choices in developing an implementation:

- Use DAGts in a DAG ADT, then proceed through the array it computes to process the vertices.
- Process the vertices after the recursive calls in a DFS.
- Process the vertices as they come off the queue in a source-queue–based topological sort.

All of these methods are used in DAG-processing implementations in the literature, and it is important to know that they are all equivalent. We will consider other topological-sort applications in Exercise 19.113 and 19.116 and in Sections 19.7 and 21.4.

Exercises

▷ **19.95** Add a topological sort function to your DAG ADT from Exercise 19.78, then add an ADT function that checks whether or not a given permutation of a DAG's vertices is a proper topological sort of that DAG.

19.96 How many different topological sorts are there of the DAG that is depicted in *Figure* 19.6?

▷ **19.97** Give the DFS forest and the reverse topological sort that results from doing a standard adjacency-lists DFS (with postorder numbering) of the DAG

```
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** Give the DFS forest and the topological sort that results from building a standard adjacency-lists representation of the DAG

```
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,
```

then using *Program* 19.1 to build the reverse, then doing a standard adjacency-lists DFS with postorder numbering.

● **19.99** Prove the correctness of each of the three suggestions given in the text for modifying DFS with postorder numbering such that it computes a topological sort instead of a reverse topological sort.

▷ **19.100** Give the DFS forest and the topological sort that results from doing a standard adjacency-matrix DFS with implicit reversal (and postorder numbering) of the DAG

```
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
```

(see *Program* 19.7).

● **19.101** Given a DAG, does there exist a topological sort that cannot result from applying a DFS-based algorithm, no matter what order the vertices adjacent to each vertex are chosen? Prove your answer.

▷ **19.102** Show, in the style of *Figure* 19.26, the process of topologically sorting the DAG

```
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
```

with the source-queue algorithm (*Program* 19.8).

▷ **19.103** Give the topological sort that results if the data structure used in the example depicted in *Figure* 19.25 is a stack rather than a queue.

● **19.104** Given a DAG, does there exist a topological sort that cannot result from applying the source-queue algorithm, no matter what queue discipline is used? Prove your answer.

19.105 Modify the source-queue topological-sort algorithm to use a generalized queue. Use your modified algorithm with a LIFO queue, a stack, and a randomized queue.

▷ **19.106** Use *Program* 19.8 to provide an implementation for the ADT function for verifying that a DAG has no cycles (see Exercise 19.78).

○ **19.107** Convert the source-queue topological-sort algorithm into a sink-queue algorithm for reverse topological sorting.

19.108 Write a program that generates all possible topological orderings of a given DAG, or, if the number of such orderings exceeds a bound taken as an argument, prints that number.

19.109 Write a program that converts any digraph with V vertices and E edges into a

DAG by doing a DFS-based topological sort and changing the orientation of any back edge encountered. Prove that this strategy always produces a DAG.

●● **19.110** Write a program that produces each of the possible DAGs with V vertices and E edges with equal likelihood (see Exercise 17.69).

19.111 Give necessary and sufficient conditions for a DAG to have just one possible topologically sorted ordering of its vertices.

19.112 Run empirical tests to compare the topological-sort algorithms given in this section for various DAGs (see Exercise 19.2, Exercise 19.79, Exercise 19.109, and Exercise 19.110). Test your program as described in Exercise 19.11 (for low densities) and as described in Exercise 19.12 (for high densities).

▷ **19.113** Modify *Program* 19.8 so that it computes the number of different simple paths from any source to each vertex in a DAG.

○ **19.114** Write a program that evaluates DAGs that represent arithmetic expressions (see *Figure* 19.19). Use the adjacency-lists graph ADT, extended to include a `double` corresponding to each vertex (to hold its value). Assume that values corresponding to leaves have been established.

○ **19.115** Describe a family of arithmetic expressions with the property that the size of the expression tree is exponentially larger than the size of the corresponding DAG (so the running time of your program from Exercise 19.114 for the DAG is proportional to the logarithm of the running time for the tree).

○ **19.116** Write a program that finds the longest simple directed `path` in a DAG, in time proportional to V. Use your program to implement an ADT function that finds a Hamilton `path` in a given DAG, if it has one.

19.7 REACHABILITY IN DAGS

To conclude our study of DAGs, we consider the problem of computing the transitive closure of a DAG. Can we develop algorithms for DAGs that are more efficient than the algorithms for general digraphs that we considered in section 19.3?

Any method for topological sorting can serve as the basis for a transitive-closure algorithm for DAGs, as follows: We proceed through the vertices in reverse topological order, computing the reachability vector for each vertex (its row in the transitive-closure matrix) from the rows corresponding to its adjacent vertices. The reverse topological sort ensures

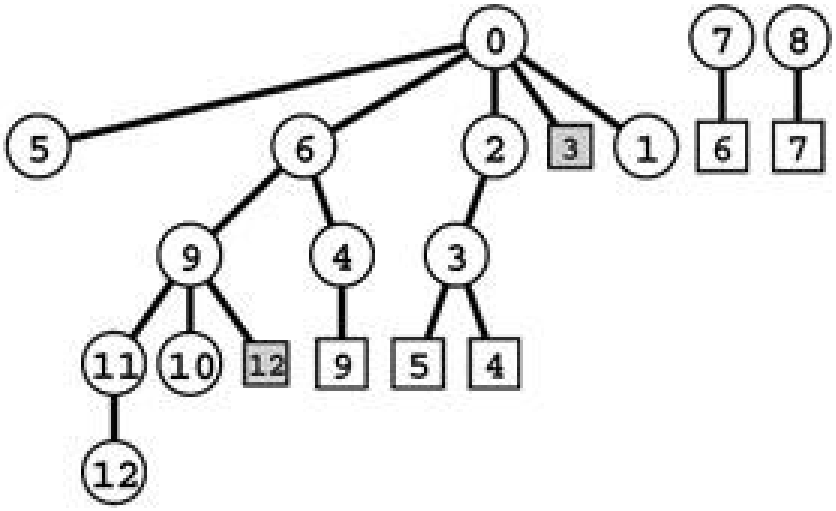
IPR2023-00124
KASPERSKY EX 1036 Page 249

that all those rows have already been computed. In total, we check each of the V entries in the vector corresponding to the destination vertex of each of the E edges, for a total running time proportional to VE. Although it is simple to implement, this method is no more efficient for DAGs than for general digraphs.

When we use a standard DFS for the topological sort (see *Program* 19.7), we can improve performance for some DAGs, as shown in *Program* 19.9. Since there are no cycles in a DAG, there are no back edges in any DFS. More important, both cross edges and down edges point to nodes for which the DFS has completed. To take advantage of this fact, we develop a recursive function to compute all vertices reachable from a given start vertex, but (as usual in DFS) we make no recursive calls for vertices for which the reachable set has already been computed. In this case, the reachable vertices are represented by a row in the transitive closure, and the recursive function takes the logical *or* of all the rows associated with its adjacent edges. For tree edges, we do a recursive call to compute the row; for cross edges, we can skip the recursive call because we know that the row has been computed by a previous recursive call; for down edges, we can skip the whole computation, because any reachable nodes that would add have already been accounted for in the set of reachable nodes for the destination vertex (lower and earlier in the DFS tree).

Using this version of DFS might be characterized as using dynamic programming to compute the transitive closure, because we make use of results that have already been computed to avoid making unnecessary recursive calls. *Figure* 19.27 illustrates the computation of the transitive closure for the sample DAG in *Figure* 19.6.

Figure 19.27 Transitive Closure of a DAG *This sequence of row vectors is the transitive closure of the DAG in Figure* 19.21*, with rows created in reverse topological order, computed as the last action in a recursive DFS function (see Program* 19.9*). Each row is the logical or of the rows for adjacent vertices, which appear earlier in the list. For example, to compute the row for* `0` *we take the logical or of the rows for* `5`, `2`, `1`, *and* `6` (*and put a 1 corresponding to* `0` *itself) because the edges* `0-5`, `0-2`, `0-1`, *and* `0-6` *take us from* `0` *to any vertex that is reachable from any of those vertices. We can ignore down edges because they add no new information. For example, we ignore the edge from* `0` *to* `3` *because the vertices reachable from* `3` *are already accounted for in the row corresponding to* `2`.




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

Property 19.13 *With dynamic programming and DFS, we can support constant query time for the abstract transitive closure of a DAG with space proportional to V^2 and time proportional to $V^2 + VX$ for preprocessing (computing the transitive closure), where X is the number of cross edges in the DFS forest.*

Proof: The proof is immediate by induction from the recursive function in *Program* 19.9. We visit the vertices in reverse topological order. Every edge points to a vertex for which we have already computed all reachable vertices, and we can therefore compute the set of reachable vertices of any vertex by merging together the sets of reachable vertices associated with the destination vertex of each edge. Taking the logical *or* of the specified rows

in the adjacency matrix accomplishes this merge. We access a row of size V for each tree edge and each cross edge. There are no back edges, and we can ignore down edges because we accounted for any vertices they reach when we processed any ancestors of both nodes earlier in the search.■

Program 19.9 Transitive closure of a DAG

This code computes the transitive closure of a DAG with a single DFS. We recursively compute the reachable vertices from each vertex from the reachable vertices of each of its children in the DFS tree. We make recursive calls for tree edges, use previously computed values for cross edges, and ignore down edges.

```
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]; }
```

If our DAG has no down edges (see Exercise 19.43), the running time of *Program* 19.9 is proportional to VE and represents no improvement over the transitive-closure algorithms that we examined for general digraphs in section 19.3 (such as, for example, *Program* 19.4) or the approach based on topological sorting that is described at the beginning of

this section. On the other hand, if the number of down edges is large (or, equivalently, the number of cross edges is small), *Program* 19.9 will be significantly faster than these methods.

The problem of finding an optimal algorithm (one that is guaranteed to finish in time proportional to V^2) for computing the transitive closure of dense DAGs is still unsolved. The best known worst-case performance bound is *VE*. However, we are certainly better off using an algorithm that runs faster for a large class of DAGs, such as *Program* 19.9, than we are using one that always runs in time proportional to *VE*, such as *Program* 19.4. As we see in section 19.9, this performance improvement for DAGs has direct implications for our ability to compute the transitive closure of general digraphs, as well.

Exercises

○ **19.117** Show, in the style of *Figure* 19.27, the reachability vectors that result when we use *Program* 19.9 to compute the transitive closure of the DAG

```
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 Develop a version of *Program* 19.9 for an adjacency-matrix DAG representation.

○ **19.119** Develop a version of *Program* 19.9 that uses an adjacency-lists representation of the transitive closure and that runs in time proportional to $V^2 + \sum_e v(e)$, where the sum is over all edges in the DAG and v (e) is the number of vertices reachable from the destination vertex of edge e. This cost will be significantly less than *VE* for some sparse DAGs (see Exercise 19.68).

○ **19.120** Develop an ADT implementation for the abstract transitive closure for DAGs that uses extra space at most proportional to V (and is suitable for huge DAGs). Use topological sorting to provide quick response when the vertices are not connected, and use a source-queue implementation to return the length of the `path` when they are connected.

○ **19.121** Develop a transitive-closure implementation based on a sink-queue–based reverse topological sort (see Exercise 19.107).

○ **19.122** Does your solution to Exercise 19.121 require that you examine all edges in the DAG, or are there edges that can be ignored, such as the down edges in DFS? Give an example that requires examination of all edges, or characterize the edges that can be skipped.

IPR2023-00124
KASPERSKY EX 1036 Page 253

19.8 STRONG COMPONENTS IN DIGRAPHS

Undirected graphs and DAGs are both simpler structures than general digraphs because of the structural symmetry that characterizes the reachability relationships among the vertices: In an undirected graph, if there is a `path` from *s* to *t*, then we know that there is also a `path` from *t* to *s*; in a DAG, if there is a directed `path` from *s* to *t*, then we know that there is *no* directed `path` from *t* to *s*. For general digraphs, knowing that *t* is reachable from *s* gives no information about whether *s* is reachable from *t*.

To understand the structure of digraphs, we consider *strong connectivity*, which has the symmetry that we seek. If *s* and *t* are strongly connected (each reachable from the other), then, by definition, so are *t* and *s*. As discussed in Section 19.1, this symmetry implies that the vertices of the digraph divide into *strong components*, which consist of mutually reachable vertices. In this section, we discuss three algorithms for finding the strong components in a digraph.

We use the same interface as for connectivity in our general graph-searching algorithms for undirected graphs (see *Program* 18.3). The goal of our algorithms is to assign component numbers to each vertex in a vertex-indexed array, using the labels 0, 1,..., for the strong components. The highest number assigned is the number of strong components, and we can use the component numbers to provide a constant-time test of whether two vertices are in the same strong component.

A brute-force algorithm to solve the problem is simple to develop. Using an abstract–transitive-closure ADT, check every pair of vertices *s* and *t* to see whether *t* is reachable from *s* and *s* is reachable from *t*. Define an undirected graph with an edge for each such pair: The connected components of that graph are the strong components of the digraph. This algorithm is simple to describe and to implement, and its running time is dominated by the costs of the abstract–transitive-closure implementation, as described by, say, Property 19.10.

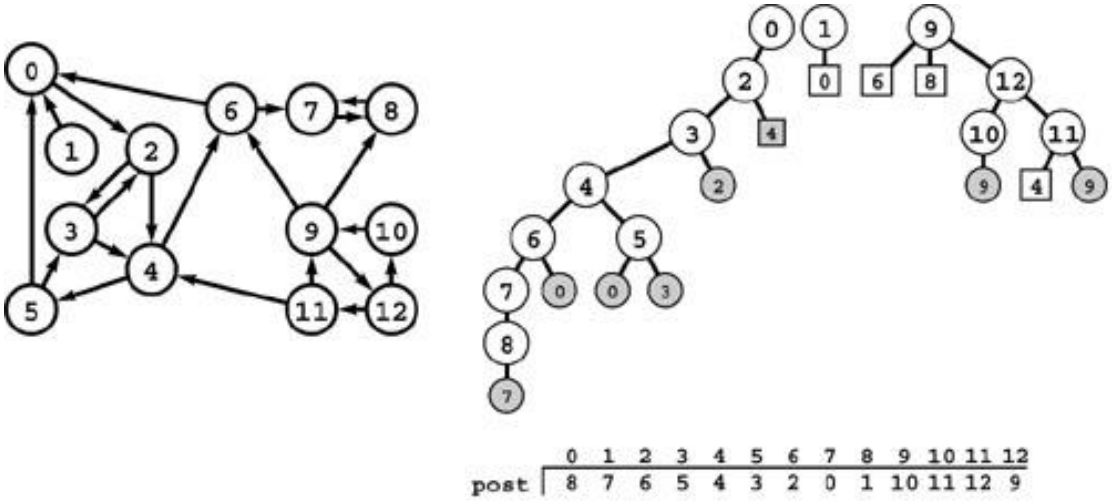
The algorithms that we consider in this section are triumphs of modern algorithm design that can find the strong components of any graph in *linear* time, a factor of *V* faster than the brute-force algorithm. For 100 vertices, these algorithms will be 100 times faster than the brute-force algorithm; for 1000 vertices, they will be 1000 times faster; and we can contemplate addressing problems involving billions of vertices. This problem is a prime example of the power of good algorithm design, one which has motivated many people to study graph algorithms closely. Where else might we contemplate reducing resource usage by a factor of 1 billion or more with an elegant solution to an important practical problem?

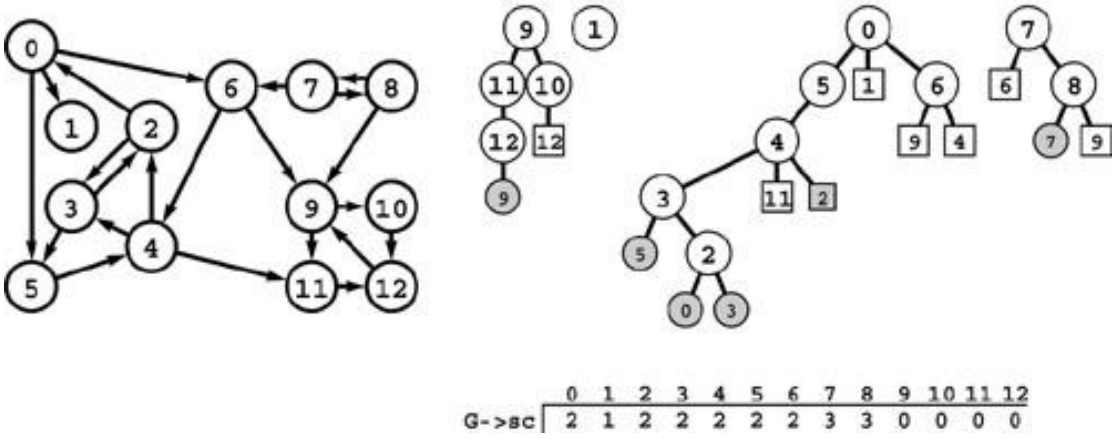
IPR2023-00124
KASPERSKY EX 1036 Page 254

The history of this problem is instructive (*see reference section*). In the 1950s and 1960s, mathematicians and computer scientists began to study graph algorithms in earnest in a context where the analysis of algorithms itself was under development as a field of study. The broad variety of graph algorithms to be considered—coupled with ongoing developments in computer systems, languages, and our understanding of performing computations efficiently—left many difficult problems unsolved. As computer scientists began to understand many of the basic principles of the analysis of algorithms, they began to understand which graph problems could be solved efficiently and which could not and then to develop increasingly efficient algorithms for the former set of problems. Indeed, R. Tarjan introduced linear-time algorithms for strong connectivity and other graph problems in 1972, the same year that R. Karp documented the intractability of the traveling-salesperson problem and many other graph problems. Tarjan's algorithm has been a staple of advanced courses in the analysis of algorithms for many years because it solves an important practical problem using simple data structures. In the 1980s, R. Kosaraju took a fresh look at the problem and developed a new solution; people later realized that a paper that describes essentially the same method appeared in the Russian scientific literature in 1972. Then, in 1999, H. Gabow found a simple implementation of one of the first approaches tried in the 1960s, giving a third linear-time algorithm for this problem.

The point of this story is not just that difficult graph-processing problems can have simple solutions, but also that the abstractions that we are using (DFS and adjacency lists) are more powerful than we might realize. As we become more accustomed to using these and similar tools, we should not be surprised to discover simple solutions to other important graph problems, as well. Researchers still seek concise implementations like these for numerous other important graph algorithms; many such algorithms remain to be discovered.

Figure 19.28 Computing Strong Components (Kosaraju's Algorithm) *To compute the strong components of the digraph at the lower left, we first do a DFS of its reverse (top left), computing a postorder array that gives the vertex indices in the order in which the recursive DFS completed (top). This order is equivalent to a postorder walk of the DFS forest (top right). Then we use the reverse of that order to do a DFS of the original digraph (bottom). First we check all nodes reachable from* 9*, then we scan from right to left through the array to find that* 1 *is the rightmost unvisited vertex, so we do the recursive call for* 1*, and so forth. The trees in the DFS forest that results from this process define the strong components: all vertices in each tree have the same value in the vertex-indexed id array* (bottom).





Kosaraju's method is simple to explain and implement. To find the strong components of a graph, first run DFS on its reverse, computing the permutation of vertices defined by the postorder numbering. (This process constitutes a topological sort if the digraph is a DAG.) Then, run DFS again on the graph, but to find the next vertex to search (when calling the recursive search function, both at the outset and each time that the recursive search function returns to the top-level search function), *use the unvisited vertex with the highest postorder number*.

The magic of the algorithm is that, when the unvisited vertices are checked according to the topological sort in this way, the trees in the DFS forest define the strong components just as trees in a DFS forest define the connected components in undirected graphs—two vertices are in the same strong component if and only if they belong to the same tree in this forest. *Figure* 19.28 illustrates this fact for our example, and we will prove it in a moment. Therefore, we can assign component numbers as we did for undirected graphs, incrementing the component number each time that the recursive function returns to the top-level search function. *Program* 19.10 is a full implementation of the method.

Program 19.10 Strong components (Kosaraju's algorithm)

This implementation finds the strong components of a digraph represented with adjacency lists. Like our solutions to the connectivity problem for undirected graphs in Section 18.5, it sets values in the vertex-indexed array sc such that the entries corresponding to any pair of vertices are equal if and only if they are in the same strong component.

First, we build the reverse digraph and do a DFS to compute a postorder permutation. Next, we do a DFS of the original digraph, using the reverse of the postorder from the first DFS in the search loop that calls the recursive function. Each recursive call in the second DFS visits all the vertices in a strong component.

```
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]; }
```

Property 19.14 *Kosaraju's method finds the strong components of a graph in linear time and space.*

Proof: The method consists of minor modifications to two DFS procedures, so the running time is certainly proportional to V^2 for dense graphs and $V + E$ for sparse graphs (using an adjacency-lists representation), as usual. To prove that it computes the strong components properly, we have to prove that two vertices *s* and *t* are in the same tree in the DFS forest for the second search if and only if they are mutually reachable.

If *s* and *t* are mutually reachable, they certainly will be in the same DFS tree because when the first of the two is visited, the second is unvisited and is reachable from the first and so will be visited before the recursive call for the root terminates.

To prove the converse, we assume that *s* and *t* are in the same tree, and let *r* be the root of the tree. The fact that *s* is reachable from *r* (through a directed `path` of tree edges) implies that there is a directed `path` from *s* to *r* in the reverse digraph. Now, the key to the proof is that there must also be a `path` from *r* to *s* in the reverse digraph because *r* has a higher postorder number than *s* (since *r* was chosen first in the second DFS at a time when both were unvisited) and there is a `path` from *s* to *r*: If there were no `path` from *r* to *s*, then the `path` from *s* to *r* in the reverse would leave *s* with a higher postorder number. Therefore, there are directed paths from *s* to *r* and from *r* to *s* in the digraph and its reverse: *s* and *r* are strongly connected. The same argument proves that *t* and *r* are strongly connected, and therefore *s* and *t* are strongly connected.■

The implementation for Kosaraju's algorithm for the adjacency-matrix digraph representation is even simpler than *Program* 19.10 because we do not need to compute the reverse explicitly; that problem is left as an exercise (see Exercise 19.128).

Program 19.10 is packaged as an ADT that represents an optimal solution to the strong reachability problem that is analogous to our solutions for connectivity in Chapter 18. In section 19.9, we examine the task of extending this solution to compute the transitive closure and to solve the reachability (abstract transitive closure) problem for digraphs.

Program 19.11 Strong components (Tarjan's algorithm)

With this implementation for the recursive DFS function, a standard adjacency-lists digraph DFS will result in strong components being identified in the vertex-indexed

array sc, according to our conventions.

We use a stack s (with stack pointer N) to hold each vertex until determining that all the vertices down to a certain point at the top of the stack belong to the same strong component. The vertex-indexed array low keeps track of the lowest preorder number reachable via a series of down links followed by one up link from each node (*see text*).

```
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++;
  }
```

First, however, we consider Tarjan's algorithm and Gabow's algorithm—ingenious methods that require only a few simple modifications to our basic DFS procedure. They are preferable to Kosaraju's algorithm because they use only one pass through the graph and because they do not require computation of the reverse for sparse graphs.

Tarjan's algorithm is similar to the program that we studied in Chapter 17 for finding bridges in undirected graphs (see *Program* 18.7). The method is based on two observations that we have already made in other contexts. First, we consider the vertices in reverse topological order so that when we reach the end of the recursive function for a vertex we know we will not encounter any more vertices in the same strong component (because all the vertices that can be reached from that vertex have been processed). Second, the back links in the tree provide a second `path` from one vertex to another and bind together the strong components.

IPR2023-00124
KASPERSKY EX 1036 Page 259

The recursive DFS function uses the same computation as *Program* 18.7 to find the highest vertex reachable (via a back edge) from any descendant of each vertex. It also uses a vertex-indexed array to keep track of the strong components and a stack to keep track of the current search path. It pushes the vertex names onto a stack on entry to the recursive function, then pops them and assigns component numbers after visiting the final member of each strong component. The algorithm is based on our ability to identify this moment with a simple test (based on keeping track of the highest ancestor reachable via one up link from all descendants of each node) at the end of the recursive procedure that tells us that all vertices encountered since entry (except those already assigned to a component) belong to the same strong component.

The implementation in *Program* 19.11 is a succinct and complete description of the algorithm that fills in the details missing from the brief sketch just given. *Figure* 19.29 illustrates the operation of the algorithm for our sample digraph from *Figure* 19.1.

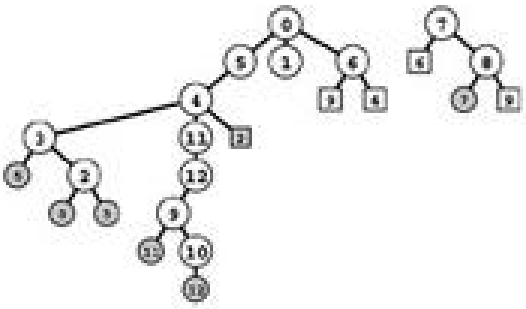
Property 19.15 *Tarjan's algorithm finds the strong components of a digraph in linear time.*

Proof sketch: If a vertex *s* has no descendants or up links in the DFS tree, or if it has a descendant in the DFS tree with an up link that points to *s* and no descendants with up links that point higher up in the tree, then it and all its descendants (except those vertices that satisfy the same property and their descendants) constitute a strong component. To establish this fact, we note that every descendant *t* of *s* that does not satisfy the stated property has some descendant that has an up link pointing higher than *t* in the tree. There is a `path` from *s* to *t* down through the tree and we can find a `path` from *t* to *s* as follows: go down from *t* to the vertex with the up link that reaches past *t*, then continue the same process from that vertex until reaching *s*.

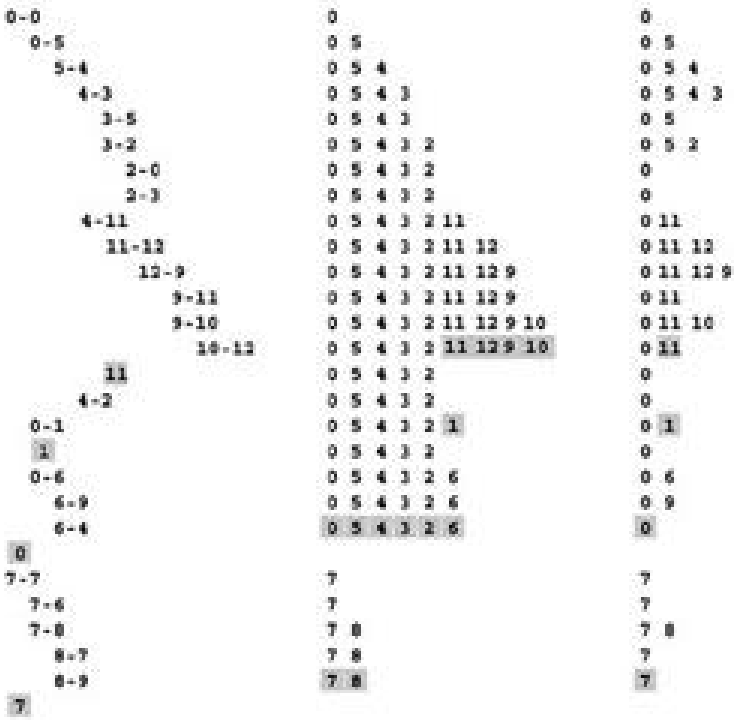
As usual, the method is linear time because it consists of adding a few constant-time operations to a standard DFS.■

Figure 19.29 Computing Strong Components (Tarjan and Gabow Algorithms) *Tarjan's algorithm is based on a recursive DFS, augmented to push vertices on a stack. It computes a component index for each vertex in a vertex-indexed array* `G->sc`*, using auxiliary arrays* `pre` *and* `low` *(center). The DFS tree for our sample graph is shown at the top and an edge trace at the bottom left. In the center at the bottom is the main stack: we push vertices reached by tree edges. Using a DFS to consider the vertices in reverse topological order, we compute, for each v, the highest point reachable via a back link from an ancestor* (`low[v]`)*. When a vertex v has* `pre[v] = low[v]` *(vertices* `11, 1, 0,` *and* `7` *here) we pop it and all the vertices above it* (shaded) *and assign them all the next component number. In Gabow's algorithm, we push vertices on the main stack, just as in Tarjan's algorithm, but we also keep a second stack* (bottom right) *with vertices on the search*

`path` *that are known to be in different strong components, by popping all vertices after the destination of each back edge. When we complete a vertex* `v` *with* `v` *at the top of this second stack* (shaded), *we know that all vertices above* `v` *on the main stack are in the same strong component.*



	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



Program 19.12 Strong components (Gabow's algorithm)

This program performs the same computation as *Program* 19.11, but uses a second stack `path` instead of the vertex-indexed array low to decide when to pop the vertices in each strong component from the main stack (*see text*).

```
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++;
  }
```

In 1999 Gabow discovered the version of Tarjan's algorithm in *Program* 19.12. The algorithm maintains the same stack of vertices in the same way as does Tarjan's algorithm, but it uses a second stack (instead of a vertex-indexed array of preorder numbers) to decide when to pop all the vertices in each strong component from the main stack. The second stack contains vertices on the search path. When a back edge shows that a sequence of such vertices all belong to the same strong component, we pop that stack to leave only the destination vertex of the back edge, which is nearer the root of the tree than are any of the other vertices. After processing all the edges for each vertex (making recursive calls for the tree edges, popping the `path` stack for the back edges, and ignoring the down edges), we check to see whether the current vertex is at the top of the `path` stack. If it is, it and all the vertices above it on the main stack make a strong component, and we pop them and assign the next strong component number to them, as we did in Tarjan's algorithm.

The example in *Figure* 19.29 also shows the contents of this second stack. Thus, this figure also illustrates the operation of Gabow's algorithm.

Property 19.16 *Gabow's algorithm finds the strong components of a digraph in linear time.*

Formalizing the argument just outlined and proving the relationship between the stack contents that it depends upon is an instructive exercise for mathematically inclined read-

ers (see Exercise 19.136). As usual, the method is linear time, because it consists of adding a few constant-time operations to a standard DFS.■

The strong-components algorithms that we have considered in this section are all ingenious and are deceptively simple. We have considered all three because they are testimony to the power of fundamental data structures and carefully crafted recursive programs. From a practical standpoint, the running time of all the algorithms is proportional to the number of edges in the digraph, and performance differences are likely to be dependent upon implementation details. For example, pushdown-stack ADT operations constitute the inner loop of Tarjan's and Gabow's algorithm. Our implementations use explicitly coded stacks; implementations that use a stack ADT may be slower. The implementation of Kosaraju's algorithm is perhaps the simplest of the three, but it suffers the slight disadvantage (for sparse digraphs) of requiring three passes through the edges (one to make the reverse and two DFS passes).

Next, we consider a key application of computing strong components: building an efficient reachability (abstract transitive closure) ADT for digraphs.

Exercises

▷ **19.123** Describe what happens when you use Kosaraju's algorithm to find the strong components of a DAG.

▷ **19.124** Describe what happens when you use Kosaraju's algorithm to find the strong components of a digraph that consists of a single cycle.

●● **19.125** Can we avoid computing the reverse of the digraph in the adjacency-lists version of Kosaraju's method (*Program* 19.10) by using one of the three techniques mentioned in section 19.4 for avoiding the reverse computation when doing a topological sort? For each technique, give either a proof that it works or a counterexample that shows that it does not work.

○ **19.126** Show, in the style of *Figure* 19.28, the DFS forests and the contents of the auxiliary vertex-indexed arrays that result when you use Kosaraju's algorithm to compute the strong components of the reverse of the digraph in *Figure* 19.5. (You should have the same strong components.)

19.127 Show, in the style of *Figure* 19.28, the DFS forests and the contents of the auxiliary vertex-indexed arrays that result when you use Kosaraju's algorithm to compute the strong components of the digraph

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

○ **19.128** Implement Kosaraju's algorithm for finding the strong components of a digraph for an ADT that uses the adjacency-matrix representation. Do not explicitly compute the reverse. *Hint*: Consider using two different recursive DFS functions.

▷ **19.129** Describe what happens when you use Tarjan's algorithm to find the strong components of a DAG.

▷ **19.130** Describe what happens when you use Tarjan's algorithm to find the strong components of a digraph that consists of a single cycle.

▷ **19.131** Show, in the style of *Figure* 19.29, the DFS forest, stack contents during the execution of the algorithm, and the final contents of the auxiliary vertex-indexed arrays that result when you use Tarjan's algorithm to compute the strong components of the reverse of the digraph in *Figure* 19.5. (You should have the same strong components.)

19.132 Show, in the style of *Figure* 19.29, the DFS forest, stack contents during the execution of the algorithm, and the final contents of the auxiliary vertex-indexed arrays that result when you use Tarjan's algorithm to compute the strong components of the digraph

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

○ **19.133** Modify the implementations of Tarjan's algorithm in *Program* 19.11 and of Gabow's algorithm in *Program* 19.12 such that they use sentinel values to avoid the need to check explicitly for cross links.

19.134 Modify the implementations of Tarjan's algorithm in *Program* 19.11 and of Gabow's algorithm in *Program* 19.12 such that they use a stack ADT.

19.135 Show, in the style of *Figure* 19.29, the DFS forest, contents of both stacks during the execution of the algorithm, and the final contents of the auxiliary vertex-indexed arrays that result when you use Gabow's algorithm to compute the strong components of the digraph

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

● **19.136** Give a full proof of Property 19.16.

○ **19.137** Develop a version of Gabow's algorithm that finds bridges and edge-connected

IPR2023-00124
KASPERSKY EX 1036 Page 264

components in undirected graphs.

● **19.138** Develop a version of Gabow's algorithm that finds articulation points and biconnected components in undirected graphs.

19.139 Develop a table in the spirit of Table 18.1 to study strong connectivity in random digraphs (see Table 19.2). Let S be the set of vertices in the largest strong component. Keep track of the size of S and study the percentages of edges in the following four classes: those connecting two vertices in S, those pointing out of S, those pointing in to S, those connecting two vertices not in S.

19.140 Run empirical tests to compare the brute-force method for computing strong components described at the beginning of this section, Kosaraju's algorithm, Tarjan's algorithm, and Gabow's algorithm, for various types of digraphs (see Exercise 19.11–18).

●●● **19.141** Develop a linear-time algorithm for *strong 2-connectivity*: Determine whether a strongly connected digraph has the property that it remains strongly connected after deleting any vertex (and all its incident edges).

19.9 TRANSITIVE CLOSURE REVISITED

By putting together the results of the previous two sections, we can develop an algorithm to solve the abstract–transitive-closure problem for digraphs that—although it offers no improvement over a DFS-based solution in the worst case—will provide an optimal solution in many situations.

The algorithm is based on preprocessing the digraph to build the latter's kernel DAG (see Property 19.2). The algorithm is efficient if the kernel DAG is small relative to the size of the original digraph. If the digraph is a DAG (and therefore is identical to its kernel DAG) or if it has just a few small cycles, we will not see any significant cost savings; however, if the digraph has large cycles or large strong components (and therefore a small kernel DAG), we can develop optimal or near-optimal algorithms. For clarity, we assume the kernel DAG to be sufficiently small that we can afford an adjacency-matrix representation, although the basic idea is still effective for larger kernel DAGs.

To implement the abstract transitive closure, we preprocess the digraph as follows:

- Find its strong components
- Build its kernel DAG

IPR2023-00124
KASPERSKY EX 1036 Page 265

● Compute the transitive closure of the kernel DAG

We can use Kosaraju's, Tarjan's, or Gabow's algorithm to find the strong components; a single pass through the edges to build the kernel DAG (as described in the next paragraph); and DFS (*Program* 19.9)to compute its transitive closure. After this preprocessing, we can immediately access the information necessary to determine reachability.

Once we have a vertex-indexed array with the strong components of a digraph, building an adjacency-array representation of its kernel DAG is a simple matter. The vertices of the DAG are the component numbers in the digraph. For each edge `s-t` in the original digraph, we simply set `D->adj[sc[s]][sc[t]]` to `1`. We would have to cope with duplicate edges in the kernel DAG if we were using an adjacency-lists representation—in an adjacency array, duplicate edges simply correspond to setting an array entry to `1` that has already been set to `1`. This small point is significant because the number of duplicate edges is potentially huge (relative to the size of the kernel DAG) in this application.

Property 19.17 *Given two vertices s and t in a digraph D, let sc* (*s*) *and sc* (*t*)*, respectively, be their corresponding vertices in D's kernel DAG K. Then, t is reachable from s in D if and only if sc* (*t*) *is reachable from sc* (*s*) *in K.*

This simple fact follows from the definitions. In particular, this property assumes the convention that a vertex is reachable from itself (all vertices have self-loops). If the vertices are in the same strong component (*sc* (*s*) = *sc*(*t*)), then they are mutually reachable.■

We determine whether a given vertex *t* is reachable from a given vertex *s* in the same way as we built the kernel DAG: We use the vertex-indexed array computed by the strong-components algorithm to get component numbers *sc* (*s*) and *sc* (*t*) (in constant time), then use those numbers to index into the transitive closure of the kernel DAG (in constant time), which tells us the result. *Program* 19.13 is an implementation of the abstract–transitive-closure ADT that embodies these ideas.

We use an *abstract*–transitive-closure interface for the kernel DAG as well. For purposes of analysis, we suppose that we use an adjacency-matrix representation for the kernel DAG because we expect the kernel DAG to be small, if not also dense.

Property 19.18 *We can support constant query time for the abstract transitive closure of a digraph with space proportional to* $V + v^2$ *and time proportional to* $E + v^2 + vx$ *for preprocessing (computing the transitive closure), where v is the number of vertices in the kernel DAG and x is the number of cross edges in its DFS forest.*

Program 19.13 Strong-component–based transitive closure

This program computes the abstract transitive closure of a digraph by computing its strong components, kernel DAG, and the transitive closure of the kernel DAG (see *Program* 19.9). The vertex-indexed array sc gives the strong component index for each vertex, or its corresponding vertex index in the kernel DAG. A vertex `t` is reachable from a vertex `s` in the digraph if and only if `sc[t]` is reachable from `sc[s]` in the kernel DAG.

```
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]); }
```

Proof: Immediate from Property 19.13.■

If the digraph is a DAG, then the strong-components computation provides no new information, and this algorithm is the same as *Program* 19.9; in general digraphs that have cycles, however, this algorithm is likely to be significantly faster than Warshall's algorithm or the DFS-based solution. For example, Property 19.18 immediately implies the following result.

Property 19.19 *We can support constant query time for the abstract transitive closure of any digraph whose kernel DAG has less than* $\sqrt[3]{V}$ *vertices with space proportional to V and time proportional to E + V for preprocessing.*

Proof: Take $v < \sqrt[3]{V}$ in Property 19.18. Then, $xv < V$ since $x < v^2$.■

Table 19.2 Properties of Random Digraphs This table shows the numbers of edges and vertices in the kernel DAGs for random digraphs generated from two dif-

ferent models (the directed versions of the models in Table 18.1). In both cases, the kernel DAG becomes small (and is sparse) as the density increases.

		random edges		random 10-neighbors	
	E	v	e	v	e
1000 vertices					
	1000	983	981	916	755
	2000	424	621	713	1039
	5000	13	13	156	313
	10000	1	1	8	17
	20000	1	1	1	1
10000 vertices					
	50000	144	150	1324	150
	100000	1	1	61	123
	200000	1	1	1	1

Key:
v Number of vertices in kernel DAG
e Number of edges in kernel DAG

We might consider other variations on these bounds. For example, if we are willing to use space proportional to E, we can achieve the same time bounds when the kernel DAG has up to $\sqrt[3]{E}$ vertices. Moreover, these time bounds are conservative because they assume that the kernel DAG is dense with cross edges—and certainly it need not be so.

The primary limiting factor in the applicability of this method is the size of the kernel DAG. The more similar our digraph is to a DAG (the larger its kernel DAG), the more difficulty we face in computing its transitive closure. Note that (of course) we still have not violated the lower bound implicit in Property 19.9, since the algorithm runs in time proportional to V^3 for dense DAGs; we have, however, significantly broadened the class of graphs for which we can avoid this worst-case performance. Indeed, constructing a random-digraph model that produces digraphs for which the algorithm is slow is a challenge (see Exercise 19.146).

Table 19.2 displays the results of an empirical study; it shows that random digraphs have small kernel DAGs even for moderate densities and even in models with severe restrictions on edge placement. Although there can be no guarantees in the worst case, we can

expect to see huge digraphs with small kernel DAGs in practice. When we do have such a digraph, we can provide an efficient implementation of the abstract–transitive-closure ADT.

Exercises

● **19.142** Develop a version of the implementation of the abstract transitive closure for digraphs based on using an adjacency-lists representation of the kernel DAG. Your challenge is to eliminate duplicates on the list without using an excessive amount of time or space (see Exercise 19.68).

▷ **19.143** Show the kernel DAG computed by *Program* 19.13 and its transitive closure for the digraph

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

○ **19.144** Convert the strong-component–based abstract–transitive-closure implementation (*Program* 19.13) into an efficient program that computes the adjacency matrix of the transitive closure for a digraph represented with an adjacency matrix, using Gabow's algorithm to compute the strong components and the improved Warshall's algorithm to compute the transitive closure of the DAG.

19.145 Do empirical studies to estimate the expected size of the kernel DAG for various types of digraphs (see Exercise 19.11–18).

●● **19.146** Develop a random-digraph model that generates digraphs that have large kernel DAGs. Your generator must generate edges one at a time, but it must not make use of any structural properties of the resulting graph.

19.147 Develop an implementation of the abstract transitive closure in a digraph by finding the strong components and building the kernel DAG, then answering reachability queries in the affirmative if the two vertices are in the same strong component, and doing a DFS in the DAG to determine reachability otherwise.

19.10 PERSPECTIVE

In this chapter, we have considered algorithms for solving the topological-sorting, transitive-closure, and shortest-paths problems for digraphs and for DAGs, including fundamental algorithms for finding cycles and strong components in digraphs. These algorithms have numerous important applications in their own right and also serve as the ba-

IPR2023-00124
KASPERSKY EX 1036 Page 269

sis for the more difficult problems involving weighted graphs that we consider in the next two chapters. Worst-case running times of these algorithms are summarized in Table 19.3.

In particular, a common theme through the chapter has been the solution of the abstract–transitive-closure problem, where we wish to support an ADT that can determine quickly, after preprocessing, whether there is a directed `path` from one given vertex to another. Despite a lower bound that implies that our worst-case preprocessing costs are significantly higher than V^2, the method discussed in section 19.7 melds the basic methods from throughout the chapter into a simple solution that provides optimal performance for many types of digraphs—the significant exception being dense DAGs. The lower bound suggests that better guaranteed performance on all graphs will be difficult to achieve, but we can use these methods to get good performance on practical graphs.

The goal of developing an algorithm with performance characteristics similar to the union-find algorithms of Chapter 1 for dense di-graphs remains elusive. Ideally, we would like to define an ADT where we can add directed edges or test whether one vertex is reachable from another and to develop an implementation where we can support all the operations in constant time (see Exercise 19.157 through 19.159). As discussed in Chapter 1, we can come close to that goal for undirected graphs, but comparable solutions for di-graphs or DAGs are still not known. (Note that deleting edges presents a challenge even for undirected graphs.) Not only does this *dynamic reachability* problem have both fundamental appeal and direct application in practice, but also it plays a critical role in the development of algorithms at a higher level of abstraction. For example, reachability lies at the heart of the problem of implementing the network simplex algorithm for the min-cost flow problem, a problem-solving model of wide applicability that we consider in Chapter 22.

Table 19.3 Worst-case cost of digraph-processing operations

This table summarizes the cost (worst-case running time) of algorithms for various digraph-processing problems considered in this chapter, for random graphs and graphs where edges randomly connect each vertex to one of 10 specified neighbors. All costs assume use of the adjacency-list representation; for the adjacency-matrix representation the E entries become V^2 entries, so, for example, the cost of computing all shortest paths is V^3. The linear-time algorithms are optimal, so the costs will reliably predict the running time on any input; the others may be overly conservative estimates of cost, so the running time may be lower for certain types of graphs. Performance characteristics of the fastest algorithms for computing the transitive closure of a digraph depend on the digraph's structure, particularly the size of its kernel DAG.

IPR2023-00124
KASPERSKY EX 1036 Page 270

	problem	cost	algorithm
digraphs			
	cycle detect	E	DFS
	transitive closure	V(E+V)	DFS from each vertex
	single-source shortest paths	E	DFS
	all shortest paths	V(E+V)	DFS from each vertex
	strong components	E	Kosaraju, Tarjan, or Gabow
	transitive closure	E + v(v + x)	kernel DAG
DAGs			
	acyclic verify	E	DFS or source queue
	topological sort	E	DFS or source queue
	transitive closure	V(V+E)	DFS
	transitive closure	V(V+X)	DFS/dynamic programming

Many other algorithms for processing digraphs and DAGs have important practical applications and have been studied in detail, and many digraph-processing problems still call for the development of efficient algorithms. The following list is representative.

Dominators Given a DAG with all vertices reachable from a single source *r*, a vertex *s* *dominates* a vertex *t* if every `path` from *r* to *t* contains *s*. (In particular, each vertex dominates itself.) Every vertex *v* other than the source has an *immediate dominator* that dominates *v* but does not dominate any dominator of *v* but *v* and itself. The set of immediate dominators is a tree that spans all vertices reachable from the source. This structure is important in compiler implementations. The dominator tree can be computed in linear time with a DFS-based approach that uses several ancillary data structures, although a slightly slower version is typically used in practice.

Transitive reduction Given a digraph, find a digraph that has the same transitive closure and the smallest number of edges among all such digraphs. This problem is tractable (see Exercise 19.154); but if we restrict it to insist that the result be a subgraph of the original graph, it is NP-hard.

Directed Euler path Given a digraph, is there a directed `path` connecting two given vertices that uses each edge in the digraph exactly once? This problem is easy by essentially the same arguments as we used for the corresponding problem for undirected graphs, which we considered in Section 17.7 (see Exercise 17.92).

Directed mail carrier Given a digraph, find a directed tour with a minimal number of

edges that uses every edge in the graph at least once (but is allowed to use edges multiple times). As we shall see in section 22.7, this problem reduces to the mincost-flow problem and is therefore tractable.

Directed Hamilton path Find the longest simple directed `path` in a digraph. This problem is NP-hard, but it is easy if the digraph is a DAG (see Exercise 19.116).

Uniconnected subgraph A digraph is said to be *uniconnected* if there is at most one directed `path` between any pair of vertices. Given a digraph and an integer k, determine whether there is a uniconnected subgraph with at least k edges. This problem is known to be NP-hard for general k.

Feedback vertex set Decide whether a given digraph has a subset of at most k vertices that contains at least one vertex from every directed cycle in G. This problem is known to be NP-hard.

Even cycle Decide whether a given digraph has a cycle of even length. As mentioned in Section 17.8, this problem, while not intractable, is so difficult to solve that no one has yet devised an algorithm that is useful in practice.

Just as for undirected graphs, myriad digraph-processing problems have been studied, and knowing whether a problem is easy or intractable is often a challenge (see Section 17.8). As indicated throughout this chapter, some of the facts that we have discovered about digraphs are expressions of more general mathematical phenomena, and many of our algorithms have applicability at levels of abstraction different from that at which we have been working. On the one hand, the concept of intractability tells us that we might encounter fundamental roadblocks in our quest for efficient algorithms that can guarantee efficient solutions to some problems. On the other hand, the classic algorithms described in this chapter are of fundamental importance and have broad applicability, as they provide efficient solutions to problems that arise frequently in practice and would otherwise be difficult to solve.

Exercises

19.148 Adapt Programs 17.13 and 17.14 to implement an ADT function for printing an Euler `path` in a digraph, if one exists. Explain the purpose of any additions or changes that you need to make in the code.

▷ **19.149** Draw the dominator tree of the digraph

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

IPR2023-00124
KASPERSKY EX 1036 Page 272

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

●● **19.150** Write an ADT function that uses DFS to create a parent-link representation of the dominator tree of a given digraph (*see reference section*).

○ **19.151** Find a transitive reduction of the digraph

```
        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** Find a subgraph of the digraph

```
        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
```

that has the same transitive closure and the smallest number of edges among all such subgraphs.

○ **19.153** Prove that every DAG has a unique transitive reduction, and give an efficient ADT function implementation for computing the transitive reduction of a DAG.

● **19.154** Write an efficient ADT function for digraphs that computes a transitive reduction.

19.155 Give an algorithm that determines whether or not a given digraph is unconnected. Your algorithm should have a worst-case running time proportional to VE.

19.156 Find the largest uniconnected subgraph in the digraph

```
        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** Develop a package of digraph ADT functions for constructing a graph from an array of edges, inserting an edge, deleting an edge, and testing whether two vertices are in the same strong component, such that construction, insertion, and deletion all take linear time and strong-connectivity queries take constant time, in the worst case.

○ **19.158** Solve Exercise 19.157, in such a way that insertion, deletion, and strong-connectivity queries all take time proportional to $\log V$ in the worst case.

●●● **19.159** Solve Exercise 19.157, in such a way that insertion, deletion, and strong-

connectivity queries all take near-constant time (as they do for the union-find algorithms for connectivity in undirected graphs).

Chapter Twenty
Minimum Spanning Trees

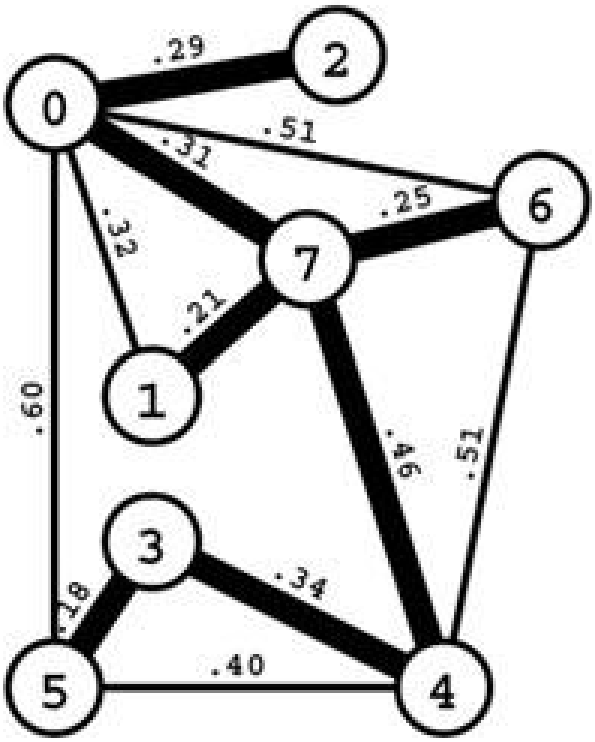
APPLICATIONS OFTEN CALL for a graph model where we associate *weights* or *costs* with each edge. In an airline map where edges represent flight routes, these weights might represent distances or fares. In an electric circuit where edges represent wires, the weights might represent the length of the wire, its cost, or the time that it takes a signal to propagate through it. In a job-scheduling problem, weights might represent time or the cost of performing tasks or of waiting for tasks to be performed.

Questions that entail cost minimization naturally arise for such situations. We examine algorithms for two such problems: (*i*) find the lowest-cost way to connect all of the points, and (*ii*) find the lowest-cost `path` between two given points. The first type of algorithm, which is useful for undirected graphs that represent objects such as electric circuits, finds a *minimum spanning tree*; it is the subject of this chapter. The second type of algorithm, which is useful for digraphs that represent objects such as an airline route map, finds the *shortest paths*; it is the subject of Chapter 21. These algorithms have broad applicability beyond circuit and map applications, extending to a variety of problems that arise on weighted graphs.

Figure 20.1 A Weighted Undirected Graph and its MST *A weighted undirected graph is a set of weighted edges. The MST is a set of edges of minimal total weight that connects the vertices* (black in the edge list, thick edges in the graph drawing). *In this particular graph, the weights are proportional to the distances between the vertices, but the basic algorithms that we consider are appropriate for general graphs and make no assumptions about the weights (see Figure 20.2).*

IPR2023-00124
KASPERSKY EX 1036 Page 275

```
0-6  .51
0-1  .32
0-2  .29
4-3  .34
5-3  .18
7-4  .46
5-4  .40
0-5  .60
6-4  .51
7-0  .31
7-6  .25
7-1  .21
```



When we study algorithms that process weighted graphs, our intuition is often supported by thinking of the weights as distances: we speak of "the vertex closest to *x*," and so forth. Indeed, the term "shortest path" embraces this bias. Despite numerous applications where we actually do work with distance and despite the benefits of geometric intuition in understanding the basic algorithms, it is important to remember that the weights do not need to be proportional to a distance at all; they might represent time or cost or an entirely different variable. Indeed, as we see in Chapter 21, weights in shortest-paths problems can even be *negative*.

To appeal to intuition in describing algorithms and examples while still retaining general applicability, we use ambiguous terminology where we refer interchangeably to edge lengths and weights. When we refer to a "short" edge, we mean a "low-weight" edge, and so forth. For most of the examples in this chapter, we use weights that are proportional to the distances between the vertices, as shown in *Figure* 20.1. Such graphs are more convenient for examples, because we do not need to carry the edge labels and can still tell at a glance that longer edges have weights higher than those of shorter edges. When the weights do represent distances, we can consider algorithms that gain efficiency by taking into account geometric properties (Sections 20.7 and 21.5). With that exception, the algorithms that we consider simply process the edges and do not take advantage of any implied geometric information (see *Figure* 20.2).

The problem of finding the minimum spanning tree of an arbitrary weighted undirected graph has numerous important applications, and algorithms to solve it have been known

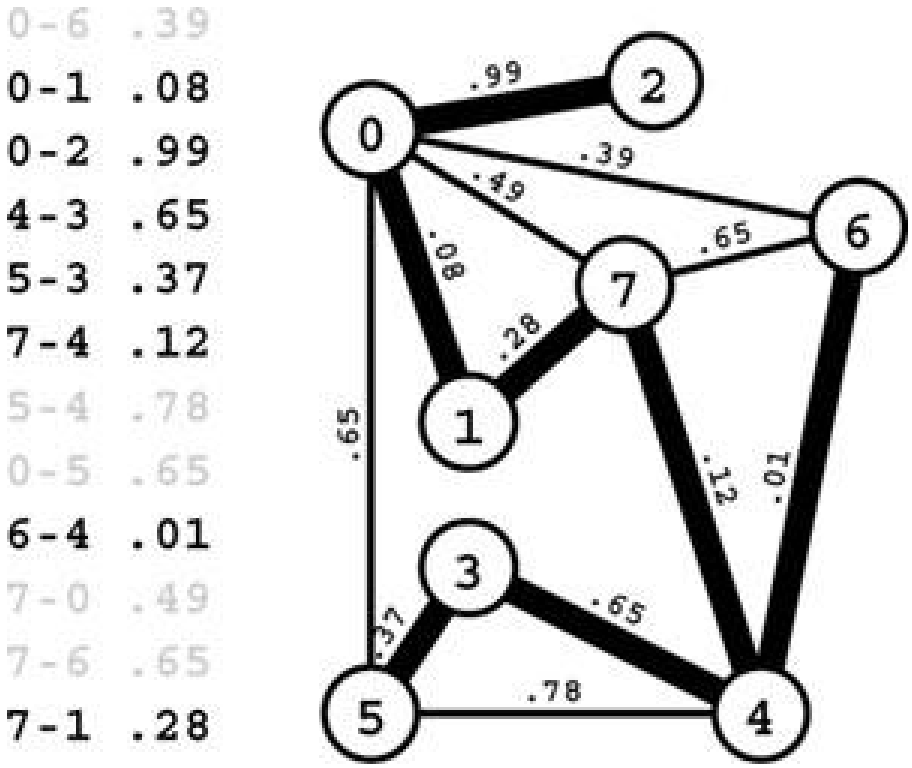
IPR2023-00124
KASPERSKY EX 1036 Page 276

since at least the 1920s; but the efficiency of implementations varies widely, and researchers still seek better methods. In this section, we examine three classical algorithms that are easily understood at a conceptual level; in Sections 20.3 through 20.5, we examine implementations of each in detail; and in Section 20.6, we consider comparisons of and improvements on these basic approaches.

Definition 20.1 *A* **minimum spanning tree (MST)** *of a weighted graph is a spanning tree whose weight (the sum of the weights of its edges) is no larger than the weight of any other spanning tree.*

If the edge weights are all positive, it suffices to define the MST as the set of edges with minimal total weight that connects all the vertices, as such a set of edges must form a spanning tree. The spanning-tree condition in the definition is included so that it applies for graphs that may have negative edge weights (see Exercise 20.2 and 20.3).

Figure 20.2 Arbitrary Weights *In this example, the edge weights are arbitrary and do not relate to the geometry of the drawn graph representation at all. This example also illustrates that the MST is not necessarily unique if edge weights may be equal: we get one MST by including* `3-4` (shown) *and a different MST by including* `0-5` *instead (although* `7-6`*, which has the same weight as those two edges, is not in any MST).*



If edges can have equal weights, the minimum spanning tree may not be unique. For example, *Figure* 20.2 shows a graph that has two different MSTs. The possibility of equal

weights also complicates the descriptions and correctness proofs of some of our algorithms. We have to consider equal weights carefully, because they are not unusual in applications and we want to know that our algorithms operate correctly when they are present.

Not only might there be more than one MST, but also the nomenclature does not capture precisely the concept that we are minimizing the weight rather than the tree itself. The proper adjective to describe a specific tree is *minimal* (one having the smallest weight). For these reasons, many authors use more accurate terms like *minimal spanning tree* or *minimum-weight spanning tree*. The abbreviation *MST*, which we shall use most often, is universally understood to capture the basic concept.

Still, to avoid confusion when describing algorithms for networks that may have edges with equal weights, we do take care to be precise to use the term "minimal" to refer to "an edge of minimum weight" (among all edges in some specified set) and "maximal" to refer to "an edge of maximum weight." That is, if edge weights are distinct, a minimal edge is the shortest edge (and is the only minimal edge); but if there is more than one edge of minimum weight, any one of them might be a minimal edge.

We work exclusively with undirected graphs in this chapter. The problem of finding a minimum-weight directed spanning tree in a digraph is different, and is more difficult.

Several classical algorithms have been developed for the MST problem. These methods are among the oldest and most well-known algorithms in this book. As we have seen before, the classical methods provide a general approach, but modern algorithms and data structures can give us compact and efficient implementations. Indeed, these implementations provide a compelling example of the effectiveness of careful ADT design and proper choice of fundamental ADT data structure and algorithm implementations in solving increasingly difficult algorithmic problems.

EXERCISES

20.1 Assume that the weights in a graph are positive. Prove that you can rescale them by adding a constant to all of them or by multiplying them all by a constant without affecting the MSTs, provided only that the rescaled weights are positive.

20.2 Show that, if edge weights are positive, a set of edges that connects all the vertices whose weights sum to a quantity no larger than the sum of the weights of any other set of edges that connects all the vertices is an MST.

IPR2023-00124
KASPERSKY EX 1036 Page 278

20.3 Show that the property stated in Exercise 20.2 holds for graphs with negative weights, provided that there are no cycles whose edges all have non-positive weights.

○ **20.4** How would you find a *maximum* spanning tree of a weighted graph?

▷ **20.5** Show that if a graph's edges all have distinct weights, the MST is unique.

▷ **20.6** Consider the assertion that a graph has a unique MST only if its edge weights are distinct. Give a proof or a counterexample.

● **20.7** Assume that a graph has $t < V$ edges with equal weights and that all other weights are distinct. Give upper and lower bounds on the number of different MSTs that the graph might have.

20.1 REPRESENTATIONS

In this chapter, we concentrate on weighted undirected graphs—the most natural setting for MST problems. Extending the basic graph representations from Chapter 17 to represent weighted graphs is straightforward: In the adjacency-matrix representation, the matrix can contain edge weights rather than Boolean values; in the adjacency-lists representation, we add a field to the list elements that represent edges, for the weights.

In our examples, we generally assume that edge weights are real numbers between 0 and 1. This decision does not conflict with various alternatives that we might need in applications, because we can explicitly or implicitly rescale the weights to fit this model (see Exercise 20.1 and 20.8). For example, if the weights are positive integers less than a known maximum value, we can divide them all by the maximum value to convert them to real numbers between 0 and 1.

We use the same basic graph ADT interface that we used in Chapter 17 (see *Program* 17.1), except that we add a weight field to the edge data type, as follows:

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

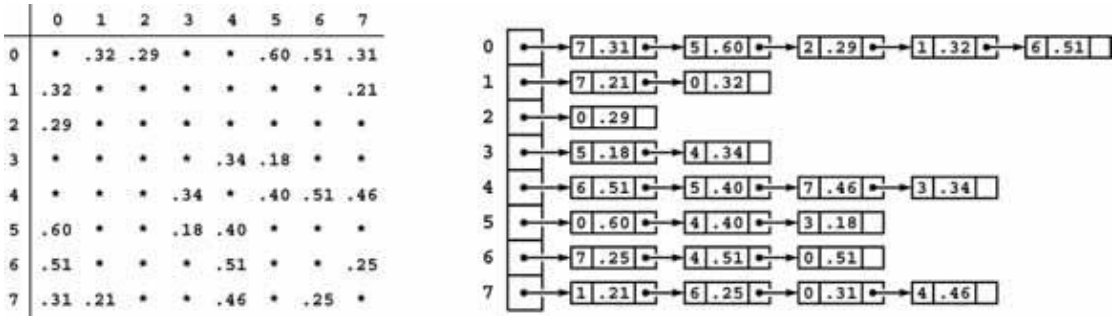
To avoid proliferation of simple types, we use double for edge weights throughout this chapter and Chapter 21. If we wanted to do so, we could build a more general ADT interface and use any data type that supports addition, subtraction, and comparisons, since we do little more with the weights than to accumulate sums and to make decisions based on their values. In Chapter 22, our algorithms are concerned with comparing linear combinations of edge weights, and the running time of some algorithms depends on arithmetic

IPR2023-00124
KASPERSKY EX 1036 Page 279

properties of the weights, so we switch to integer weights to allow us to more easily analyze the algorithms.

We use sentinel weights to indicate the absence of an edge. Another straightforward approach would be to use the standard adjacency matrix to indicate the existence of edges and a parallel matrix to hold weights. With sentinels, many of our algorithms do not need to explicitly test whether or not an edge exists. The adjacency-matrix representation of our sample graph is shown in *Figure* 20.3; *Program* 20.1 gives the implementation details of the weighted-graph ADT for an adjacency-matrix representation. It uses an auxiliary function that allocates a matrix of weights and fills it with the sentinel weight. Inserting an edge amounts to storing the weight in two places in the matrix—one for each orientation of the edge. The sentinel weight value that indicates the absence of an edge is larger than all other weights, *not* 0, which represents an edge of length 0 (an alternative would be to disallow edges of length 0). As is true of algorithms that use the adjacency-matrix representation for unweighted graphs, the running time of any algorithm that uses this representation is proportional to V^2 (to initialize the matrix) or higher.

Figure 20.3 Weighted-Graph Representations (Undirected) *The two standard representations of weighted undirected graphs include weights with each edge representation, as illustrated in the adjacency-matrix* (*left*) *and adjacency-lists* (right) *representation of the graph depicted in Figure 20.1. The adjacency matrix is symmetric and the adjacency lists contain two nodes for each edge, as in unweighted directed graphs. Nonexistent edges are represented by a sentinel value in the matrix* (indicated by asterisks in the figure) *and are not present at all in the lists. Self-loops are absent in both of the representations illustrated here because MST algorithms are simpler without them; other algorithms that process weighted graphs use them (see Chapter 21).*



Similarly, *Program* 20.2 gives the implementation details of the weighted-graph ADT for an adjacency-lists representation. A vertex-indexed array associates each vertex with a linked list of that vertex's incident edges. Each list node represents an edge, and contains a weight. An adjacency-lists representation of our sample graph is also shown in *Figure* 20.3.

IPR2023-00124
KASPERSKY EX 1036 Page 280

Program 20.1 Weighted-graph ADT (adjacency matrix)

For dense weighted undirected graphs, we use a matrix of weights, with the entries in row `v` and column `w` and in row `w` and column `v` containing the weight of the edge `v-w`. The sentinel value `maxWT` indicates the absence of an edge. This code assumes that edge weights are of type double, and uses an auxiliary routine `MATRIXdouble` to allocate a `V`-by-`V` array of weights with all entries initialized to `maxWT` (see *Program* 17.4). To adapt this code for use as a weighted *digraph* ADT implementation (see Chapter 21), remove the last line of `GRAPHinsertE`.

```
#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;
  }
```

As with our undirected-graph representations, we do not explicitly test for parallel edges in either representation. Depending upon the application, we might alter the adjacency-matrix representation to keep the parallel edge of lowest or highest weight, or to effectively coalesce parallel edges to a single edge with the sum of their weights. In the adjacency-lists representation, we can allow parallel edges to remain in the data structure, or build more powerful data structures to eliminate them using one of the rules just mentioned for adjacency matrices (see Exercise 17.47).

IPR2023-00124
KASPERSKY EX 1036 Page 281

Program 20.2 Weighted-graph ADT (adjacency lists)

This adjacency-lists representation is appropriate for sparse undirected weighted graphs. As with undirected unweighted graphs, we represent each edge with two list nodes, one on the adjacency list for each of the edge's vertices. To represent the weights, we add a weight field to the list nodes.

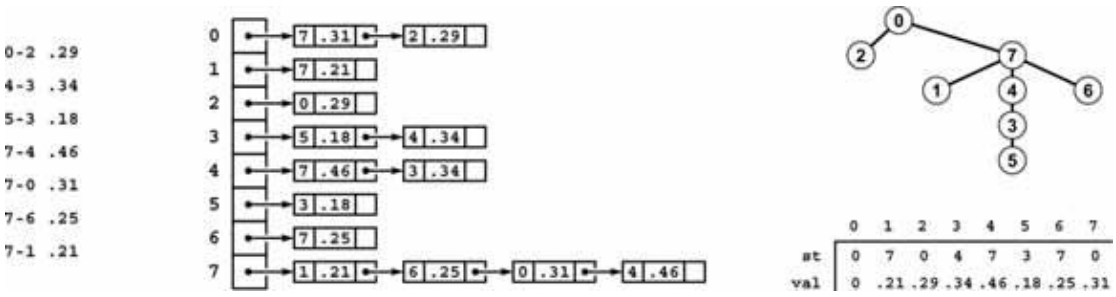
This implementation does not check for duplicate edges. Beyond the factors that we considered for unweighted graphs, a design decision has to be made about whether or not to allow multiple edges of differing weights connecting the same pair of nodes, which might be appropriate in certain applications.

To adapt this code for use as a weighted *digraph* ADT implementation, (see Chapter 21), remove the line that adds the edge from `w` to `v` (the next-to-last line of `GRAPHinsertE`).

```
#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++;
      }
```

Figure 20.4 MST Representations *This figure depicts various representations of the MST in Figure 20.1. The most straightforward is a list of its edges, in no particular order (left). The MST is also a sparse graph, and might be represented with adjacency lists (center). The most compact is a parent-link representation: we choose one of the vertices as the root and keep two vertex-indexed arrays, one with the parent of each vertex in the tree, the other with the weight of the edge from each vertex to its parent (right). The orientation of the tree (choice of root vertex) is arbitrary, not a property of the MST. We can convert from any one of these representations to any other in linear time.*



How should we represent the MST itself? The MST of a graph *G* is a subgraph of *G* that is also a tree, so we have numerous options. Chief among them are

- A graph
- A linked list of edges
- An array of edges
- A vertex-indexed array with parent links

Figure 20.4 illustrates these options for the example MST in *Figure* 20.1. Another alternative is to define and use an ADT for trees.

The same tree might have many different representations in any of the schemes. In what order should the edges be presented in the list-of-edges representation? Which node should be chosen as the root in the parent-link representation (see Exercise 20.15)? Generally speaking, when we run an MST algorithm, the particular MST representation that results is an artifact of the algorithm used, rather than reflecting any important features

IPR2023-00124
KASPERSKY EX 1036 Page 283

of the MST.

From an algorithmic point of view, the choice of MST representation is of little consequence, because we can convert easily from each of these representations to any of the others. To convert from the graph representation to an array of edges, we can use the `GRAPHedges` function in the graph ADT. To convert from the parent-link representation in an array `st` to an array of edges in an array `mst`, we can use the simple loop

```
for (k = 1; k < G->V; k++) mst[k] = EDGE (k, st[k]);
```

This code is for the typical case where the MST is rooted at `0`, and it does not put the dummy edge `0-0` onto the MST edge list.

These two conversions are trivial, but how do we convert from the array-of-edges representation to the parent-link representation? We have the basic tools to accomplish this task easily as well: We convert to the graph representation using a loop like the one just given (changed to call `GRAPHinsert` for each edge), then run a a DFS starting at any vertex to compute the parent-link representation of the DFS tree, in linear time.

In short, the choice of MST representation is a matter of convenience for our algorithms. Dependent on the needs of applications, we can add wrapper functions to give client programs the flexibility that they might need. The parent-link representation is more natural for some of the algorithms that we consider; other representations are more natural for other algorithms. Our goal in this chapter is to develop efficient implementations that can support a graph ADT function `GRAPHmst`; we do not specify details of the interface, so we leave flexibility for simple wrapper functions that can meet the needs of clients while allowing implementations that produce MSTs in a manner natural for each algorithm (see, for example, Exercise 20.18 through 20.20).

Exercises

▷ **20.8** Build a graph ADT that uses integer weights, but keep track of the minimum and maximum weights in the graph and include an ADT function that always returns weights that are numbers between 0 and 1.

▷ **20.9** Modify the sparse-random-graph generator in *Program* 17.7 to assign a random weight (between 0 and 1) to each edge.

▷ **20.10** Modify the dense-random-graph generator in *Program* 17.8 to assign a random weight (between 0 and 1) to each edge.

20.11 Write a program that generates random weighted graphs by connecting vertices arranged in a $\sqrt{V}$-by-$\sqrt{V}$ grid to their neighbors (as in *Figure* 19.3, but undirected) with random weights (between 0 and 1) assigned to each edge.

20.12 Write a program that generates a random complete graph that has weights chosen from a Gaussian distribution.

● **20.13** Write a program that generates V random points in the plane, then builds a weighted graph by connecting each pair of points within a given distance d of one another with an edge whose weight is the distance. (see Exercise 17.60). Determine how to set d so that the expected number of edges is E.

● **20.14** Find a large weighted graph online—perhaps a map with distances, telephone connections with costs, or an airline rate schedule.

20.15 Write down an 8-by-8 matrix that contains parent-link representations of all the orientations of the MST of the graph in *Figure* 20.1. Put the parent-link representation of the tree rooted at `i` in the `i`th row of the matrix.

▷ **20.16** Add `GRAPHscan, GRAPHshow`, and `GRAPHedges` functions to the adjacency-matrix and adjacency-lists implementations in Programs 20.1 and 20.2 (see Programs 17.5 and 17.6).

▷ **20.17** Provide an implementation for the `MATRIXdouble` function that is used in *Program* 20.1 (see *Program* 17.4).

▷ **20.18** Assume that a function `GRAPHmstE` produces an array-of-edges representation of an MST in an array `mst`. Add a wrapper ADT function `GRAPHmst` to the graph ADT that calls `GRAPHmstE` but puts a parent-link representation of the MST into an array that is passed as an argument by the client.

○ **20.19** Assume that a function `GRAPHmstV` produces a parent-link representation of an MST in an array `st` with corresponding weights in another array `wt`. Add a wrapper ADT function `GRAPHmst` to the graph ADT that calls `GRAPHmstV` but puts an array-of-edges representation of the MST into an array that is passed as an argument by the client.

▷ **20.20** Define a `Tree` ADT. Then, under the assumptions of Exercise 20.19, add a wrapper ADT function to the graph ADT that calls `GRAPHmst` but returns the MST in a `Tree`.

IPR2023-00124
KASPERSKY EX 1036 Page 285

20.2 UNDERLYING PRINCIPLES OF MST ALGORITHMS

The MST problem is one of the most heavily studied problems that we encounter in this book. Basic approaches to solving it were studied long before the development of modern data structures and modern techniques for studying the performance of algorithms, at a time when finding the MST of a graph that contained, say, thousands of edges was a daunting task. As we shall see, several new MST algorithms differ from old ones essentially in their use and implementation of modern algorithms and data structures for basic tasks, which (coupled with modern computing power) makes it possible for us to compute MSTs with millions or even billions of edges.

One of the defining properties of a tree (see Section 5.4) is that adding an edge to a tree creates a unique cycle. This property is the basis for proving two fundamental properties of MSTs, which we now consider. All the algorithms that we encounter are based on one or both of these two properties.

The first property, which we refer to as the *cut property*, has to do with identifying edges that must be in an MST of a given graph. The few basic terms from graph theory that we define next make possible a concise statement of this property, which follows.

Definition 20.2 *A* **cut** *in a graph is a partition of the vertices into two disjoint sets. A* **crossing edge** *is one that connects a vertex in one set with a vertex in the other.*

We sometimes specify a cut by specifying a set of vertices, leaving implicit the assumption that the cut comprises that set and its complement. Generally, we use cuts where both sets are nonempty—otherwise, for example, there are no crossing edges.

Property 20.1 (Cut property) *Given any cut in a graph, every minimal crossing edge belongs to some MST of the graph, and every MST contains a minimal crossing edge.*

Proof: The proof is by contradiction. Suppose that e is a minimal crossing edge that is not in any MST and let T be any MST; or suppose that T is an MST that contains no minimal crossing edge and let e be any minimal crossing edge. In either case, T is an MST that does not contain the minimal crossing edge e. Now consider the graph formed by adding e to T. This graph has a cycle that contains e, and that cycle must contain at least one other crossing edge—say, f, which is equal or higher weight than e (since e is minimal). We can get a spanning tree of equal or lower weight by deleting f and adding e, contradicting either the minimality of T or the assumption that e is not in T.■

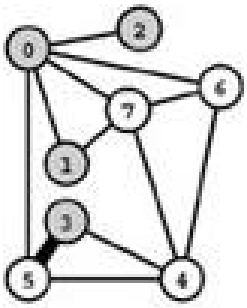
If a graph's edge weights are distinct, it has a unique MST; and the cut property says that

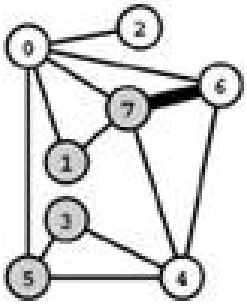
IPR2023-00124
KASPERSKY EX 1036 Page 286

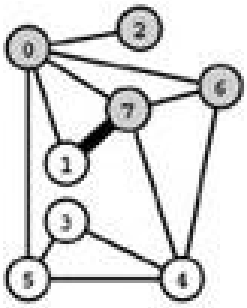
the shortest crossing edge for every cut must be in the MST. When equal weights are present, we may have multiple minimal crossing edges. At least one of them will be in any given MST and the others may be present or absent.

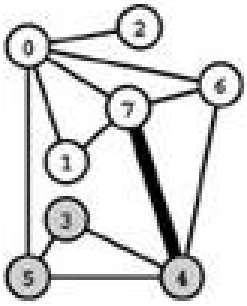
Figure 20.5 illustrates several examples of this cut property. Note that there is no requirement that the minimal edge be the *only* MST edge connecting the two sets; indeed, for typical cuts there are several MST edges that connect a vertex in one set with a vertex in the other. If we could be sure that there were only one such edge, we might be able to develop divide-and-conquer algorithms based on judicious selection of the sets; but that is not the case.

Figure 20.5 Cut Property *These four examples illustrate Property 20.1. If we color one set of vertices gray and another set white, then the shortest edge connecting a gray vertex with a white one belongs to an MST.*









We use the cut property as the basis for algorithms to find MSTs, and it also can serve as an *optimality condition* that characterizes MSTs. Specifically, the cut property implies that every edge in an MST is a minimal crossing edge for the cut defined by the vertices in the two subtrees connected by the edge.

The second property, which we refer to as the *cycle property*, has to do with identifying edges that do *not* have to be in a graph's MST. That is, if we ignore these edges, we can still find an MST.

Property 20.2 (Cycle property) *Given a graph G, consider the graph G′ defined by*

adding an edge e to G. Adding e to an MST of G and deleting a maximal edge on the resulting cycle gives an MST of G′.

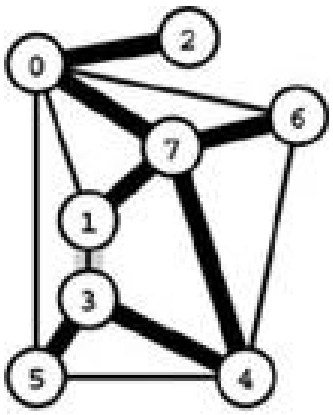
Proof: If *e* is longer than all the other edges on the cycle, it cannot be on an MST of *G′*, because of Property 20.1: Removing *e* from any such MST would split the latter into two pieces, and *e* would not be the shortest edge connecting vertices in each of those two pieces, because some other edge on the cycle must do so. Otherwise, let *t* be a maximal edge on the cycle created by adding *e* to the MST of *G*. Removing *t* would split the original MST into two pieces, and edges of *G* connecting those pieces are no shorter than *t*; so *e* is a minimal edge in *G′* connecting vertices in those two pieces. The subgraphs induced by the two subsets of vertices are identical for *G* and *G′*, so an MST for *G′* consists of *e* and the MSTs of those two subsets.

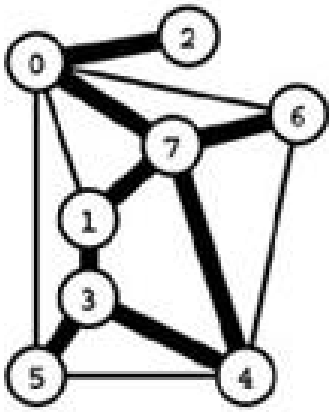
In particular, note that if *e* is maximal on the cycle, then we have shown that there exists an MST of *G′* that does not contain *e* (the MST of *G*).■

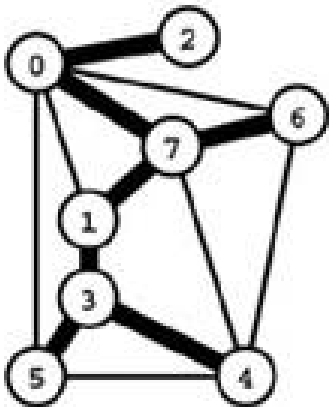
Figure 20.6 illustrates this cycle property. Note that the process of taking any spanning tree, adding an edge that creates a cycle, and then deleting a maximal edge on that cycle gives a spanning tree of weight less than or equal to the original. The new tree weight will be less than the original if and only if the added edge is shorter than some edge on the cycle.

The cycle property also serves as the basis for an optimality condition that characterizes MSTs: It implies that every edge in a graph that is not in a given MST is a maximal edge on the cycle that it forms with MST edges.

Figure 20.6 Cycle Property *Adding the edge* `1-3` *to the graph in Figure 20.1 invalidates the MST* (*top*). *To find the MST of the new graph, we add the new edge to the MST of the old graph, which creates a cycle* (*center*). *Deleting the longest edge on the cycle* (`4-7`) *yields the MST of the new graph* (*bottom*). *One way to verify that a spanning tree is minimal is to check that each edge not on the MST has the largest weight on the cycle that it forms with tree edges. For example, in the bottom graph,* `4-6` *has the largest weight on the cycle* `4-6-7-1-3-4`.







The cut property and the cycle property are the basis for the classical algorithms that we consider for the MST problem. We consider edges one at a time, using the cut property to accept them as MST edges or the cycle property to reject them as not needed. The algorithms differ in their approaches to efficiently identifying cuts and cycles.

The first approach to finding the MST that we consider in detail is to build the MST one edge at a time: start with any vertex as a single-vertex MST, then add $V - 1$ edges to it, always taking next a minimal edge that connects a vertex on the MST to a vertex not yet on the MST. This method is known as *Prim's algorithm*; it is the subject of Section 20.3.

IPR2023-00124
KASPERSKY EX 1036 Page 290

Property 20.3 *Prim's algorithm computes an MST of any connected graph.*

Proof: As described in detail in Section 20.2, the method is a generalized graph-search method. Implicit in the proof of Property 18.12 is the fact that the edges chosen are a spanning tree. To show that they are an MST, apply the cut property, using vertices on the MST as the first set, vertices not on the MST as the second set.■

Another approach to computing the MST is to apply the cycle property repeatedly: We add edges one at a time to a putative MST, deleting a maximal edge on the cycle if one is formed (see Exercise 20.28 and 20.66). This method has received less attention than the others that we consider because of the comparative difficulty of maintaining a data structure that supports efficient implementation of the "delete the longest edge on the cycle" operation.

The second approach to finding the MST that we consider in detail is to process the edges in order of their length (shortest first), adding to the MST each edge that does not form a cycle with edges previously added, stopping after $V - 1$ edges have been added. This method is known as *Kruskal's algorithm*; it is the subject of Section 20.4.

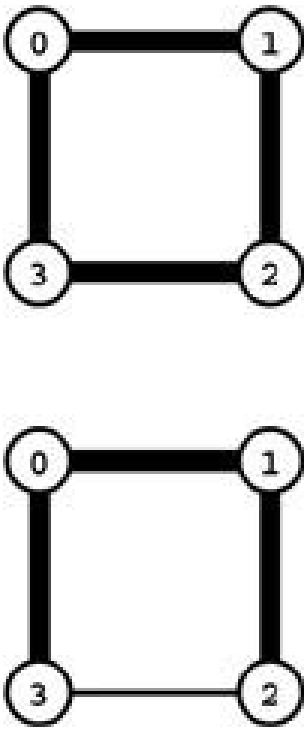
Property 20.4 *Kruskal's algorithm computes an MST of any connected graph.*

Proof: We prove by induction that the method maintains a forest of MST subtrees. If the next edge to be considered would create a cycle, it is a maximal edge on the cycle (since all others appeared before it in the sorted order), so ignoring it still leaves an MST, by the cycle property. If the next edge to be considered does not form a cycle, apply the cut property, using the cut defined by the set of vertices connected to one of the edge's vertex by MST edges (and its complement). Since the edge does not create a cycle, it is the only crossing edge, and since we consider the edges in sorted order, it is a minimal edge and therefore in an MST. The basis for the induction is the V individual vertices; once we have chosen $V - 1$ edges, we have one tree (the MST). No unexamined edge is shorter than an MST edge, and all would create a cycle, so ignoring all of the rest of the edges leaves an MST, by the cycle property.■

The third approach to building an MST that we consider in detail is known as *Boruvka's algorithm*; it is the subject of Section 20.4. The first step is to add to the MST the edges that connect each vertex to its closest neighbor. If edge weights are distinct, this step creates a forest of MST subtrees (we prove this fact and consider a refinement that does so even when equal-weight edges are present in a moment). Then, we add to the MST the edges that connect each tree to a closest neighbor (a minimal edge connecting a vertex in one tree with a vertex in any other), and iterate the process until we are left with just one tree.

IPR2023-00124
KASPERSKY EX 1036 Page 291

Figure 20.7 Cycles in Boruvka's Algorithm In the graph of four vertices and four edges shown here, the edges are all the same length. When we connect each vertex to a nearest neighbor, we have to make a choice between minimal edges. In the example at the top, we choose `1` from `0, 2` from `1, 3` from `2`, and `0` from `3`, which leads to a cycle in the putative MST. Each of the edges are in some MST, but not all are in every MST. To avoid this problem, we adopt a tie-breaking rule, as shown in the bottom: choose the minimal edge to the vertex with the lowest index. Thus, we choose `1` from `0, 0` from `1, 1` from `2`, and `0` from `3`, which yields an MST. The cycle is broken because highest-numbered vertex `3` is not chosen from either of its neighbors `2` or `1`, and it can choose only one of them (`0`).



Property 20.5 *Boruvka's algorithm computes the MST of any connected graph.*

First, suppose that the edge weights are distinct. In this case, each vertex has a unique closest neighbor, the MST is unique, and we know that each edge added is an MST edge by applying the cut property (it is the shortest edge crossing the cut from a vertex to all the other vertices). Since every edge chosen is from the unique MST, there can be no cycles, each edge added merges two trees from the forest into a bigger tree, and the process continues until a single tree, the MST, remains.

If edge weights are not distinct, there may be more than one closest neighbor, and a cycle could form when we add the edges to closest neighbors (see *Figure* 20.7). Put another way, we might include two edges from the set of minimal crossing edges for some vertex, when only one belongs on the MST. To avoid this problem, we need an appropriate tie-breaking rule. One choice is to choose, among the minimal neighbors, the one with the lowest vertex number. Then any cycle would present a contradiction: if v is the highest-

numbered vertex in the cycle, then neither neighbor of v would have led to its choice as the closest, and v would have led to the choice only one of its lower-numbered neighbors, not both.■

These algorithms are all special cases of a general paradigm that is still being used by researchers seeking new MST algorithms. Specifically, we can apply *in arbitrary order* the cut property to accept an edge as an MST edge or the cycle property to reject an edge, continuing until neither can increase the number of accepted or rejected edges. At that point, any division of the graph's vertices into two sets has an MST edge connecting them (so applying the cut property cannot increase the number of MST edges), and all graph cycles have at least one non-MST edge (so applying the cycle property cannot increase the number of non-MST edges). Together, these properties imply that a complete MST has been computed.

More specifically, the three algorithms that we consider in detail can be unified with a generalized algorithm where we begin with a forest of single-vertex MST subtrees (each with no edges) and perform the step of adding to the MST a minimal edge connecting any two subtrees in the forest, continuing until $V - 1$ edges have been added and a single MST remains. By the cut property, no edge that causes a cycle need be considered for the MST, since some other edge was previously a minimal edge crossing a cut between MST subtrees containing each of its vertices. With Prim's algorithm, we grow a single tree an edge at a time; with Kruskal's and Boruvka's algorithms, we coalesce trees in a forest.

As described in this section and in the classical literature, the algorithms involve certain high-level abstract operations, such as the following:

- Find a minimal edge connecting two subtrees.
- Determine whether adding an edge would create a cycle.
- Delete the longest edge on a cycle.

Our challenge is to develop algorithms and data structures that implement these operations efficiently. Fortunately, this challenge presents us with an opportunity to put to good use basic algorithms and data structures that we developed earlier in this book.

MST algorithms have a long and colorful history that is still evolving; we discuss that history as we consider them in detail. Our evolving understanding of different methods of implementing the basic abstract operations has created some confusion surrounding the origins of the algorithms over the years. Indeed, the methods were first described in the 1920s, pre-dating the development of computers as we know them, as well as pre-dating our basic knowledge about sorting and other algorithms. As we now know, the choices of

underlying algorithm and data structure can have substantial influences on performance, even when we are implementing the most basic schemes. In recent years, research on the MST problem has concentrated on such implementation issues, still using the classical schemes. For consistency and clarity, we refer to the basic approaches by the names listed here, although abstract versions of the algorithms were considered earlier, and modern implementations use algorithms and data structures invented long after these methods were first contemplated.

As yet unsolved in the design and analysis of algorithms is the quest for a linear-time MST algorithm. As we shall see, many of our implementations are linear-time in a broad variety of practical situations, but they are subject to a nonlinear worst case. The development of an algorithm that is guaranteed to be linear-time for sparse graphs is still a research goal.

Beyond our normal quest in search of the best algorithm for this fundamental problem, the study of MST algorithms underscores the importance of understanding the basic performance characteristics of fundamental algorithms. As programmers continue to use algorithms and data structures at increasingly higher levels of abstraction, situations of this sort become increasingly common. Our ADT implementations have varying performance characteristics—as we use higher-level ADTs as components when solving more yet higher-level problems, the possibilities multiply. Indeed, we often use algorithms that are based on using MSTs and similar abstractions (enabled by the efficient implementations that we consider in this chapter) to help us solve other problems at a yet higher level of abstraction.

Exercises

20.21 Label the following points in the plane 0 through 5, respectively:

```
(1, 3) (2, 1) (6, 5) (3, 4) (3, 7) (5, 3).
```

Taking edge lengths to be weights, give an MST of the graph defined by the edges

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

20.22 Suppose that a graph has distinct edge weights. Does its shortest edge have to belong to the MST? Prove that it does or give a counterexample.

20.23 Answer Exercise 20.22 for the graph's *longest* edge.

20.24 Give a counterexample that shows why the following strategy does not necessarily

IPR2023-00124
KASPERSKY EX 1036 Page 294

find the MST: “Start with any vertex as a single-vertex MST, then add $V - 1$ edges to it, always taking next a minimal edge incident upon the vertex most recently added to the MST.”

20.25 Suppose that a graph has distinct edge weights. Does a minimal edge on every cycle have to belong to the MST? Prove that it does or give a counterexample.

20.26 Given an MST for a graph G, suppose that an edge in G is deleted. Describe how to find an MST of the new graph in time proportional to the number of edges in G.

20.27 Show the MST that results when you repeatedly apply the cycle property to the graph in *Figure* 20.1, taking the edges in the order given.

20.28 Prove that repeated application of the cycle property gives an MST.

20.29 Describe how each of the algorithms described in this section can be adapted (if necessary) to the problem of finding a *minimal spanning forest* of a weighted graph (the union of the MSTs of its connected components).

20.3 PRIM'S ALGORITHM AND PRIORITY-FIRST SEARCH

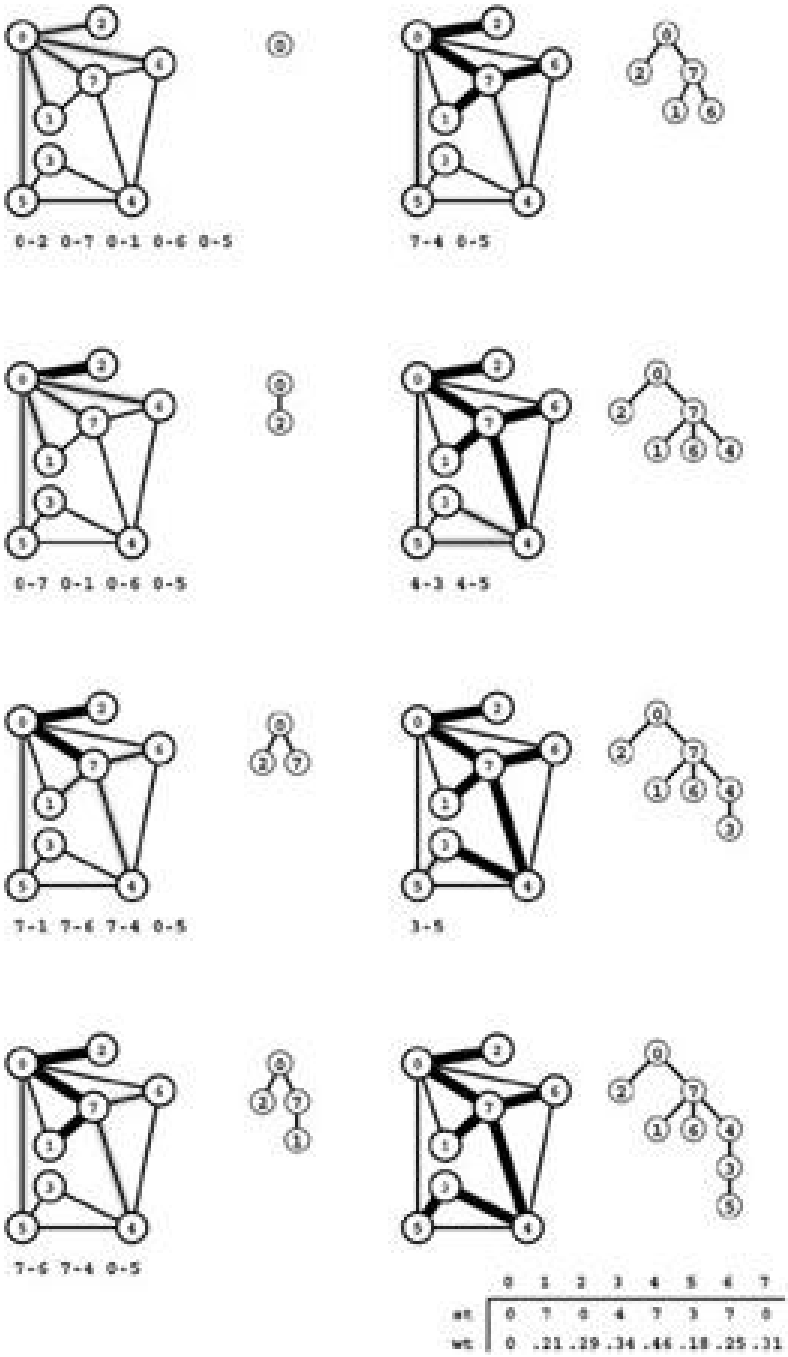
Prim's algorithm is perhaps the simplest MST algorithm to implement, and it is the method of choice for dense graphs. We maintain a cut of the graph that is comprised of *tree* vertices (those chosen for the MST) and *nontree* vertices (those not yet chosen for the MST). We start by putting any vertex on the MST, then put a minimal crossing edge on the MST (which changes its nontree vertex to a tree vertex) and repeat the same operation $V - 1$ times, to put all vertices on the tree.

A brute-force implementation of Prim's algorithm follows directly from this description. To find the edge to add next to the MST, we could examine all the edges that go from a tree vertex to a nontree vertex, then pick the shortest of the edges found to put on the MST. We do not consider this implementation here because it is overly expensive (see Exercise 20.30 through 20.32). Adding a simple data structure to eliminate excessive recomputation makes the algorithm both simpler and faster.

Adding a vertex to the MST is an incremental change: To implement Prim's algorithm, we focus on the nature of that incremental change. The key is to note that our interest is in the shortest distance from each nontree vertex to the tree. When we add a vertex `v` to the tree, the only possible change for each nontree vertex `w` is that adding `v` brings `w` closer than before to the tree. In short, we do not need to check the distance from `w` to all tree

vertices—we just need to keep track of the minimum and check whether the addition of `v` to the tree necessitates that we update that minimum.

Figure 20.8 Prim's MST Algorithm *The first step in computing the MST with Prim's algorithm is to add* `0` *to the tree. Then we find all the edges that connect* `0` *to other vertices (which are not yet on the tree) and keep track of the shortest* (*top left*). *The edges that connect tree vertices with nontree vertices (the fringe) are shadowed in gray and listed below each graph drawing. For simplicity in this figure, we list the fringe edges in order of their length, so that the shortest is the first in the list. Different implementations of Prim's algorithm use different data structures to maintain this list and to find the minimum. The second step is to move the shortest edge* `0-2` (*along with the vertex that it takes us to*) *from the fringe to the tree* (*second from top, left*). *Third, we move* `0-7` *from the fringe to the tree, replace* `0-1` *by* `7-1` *and* `0-6` *by* `7-6` *on the fringe (because adding* `7` *to the tree brings* `1` *and* `6` *closer to the tree*), *and add* `7-4` *to the fringe* (*because adding* `7` *to the tree makes* `7-4` *an edge that connects a tree vertex with a nontree vertex*) (*third from top, left*). *Next, we move edge* `7-1` *to the tree* (*bottom, left*). *To complete the computation, we take* `7-6, 7-4, 4-3`, *and* `3-5` *off the queue, updating the fringe after each insertion to reflect any shorter or new paths discovered (right, top to bottom). An oriented drawing of the growing MST is shown at the right of each graph drawing. The orientation is an artifact of the algorithm: we generally view the MST itself as a set of edges, unordered and unoriented.*



To implement this idea, we need data structures that can give us the following information:

- For each tree vertex, its parent in the MST
- For each nontree vertex, the closest tree vertex
- For each tree vertex, the length of its parent link
- For each nontree vertex, its distance to the tree

The simplest implementation for each of these data structures is a vertex-indexed array, though we have various options to save space by combining arrays or using structures.

Program 20.3 is an implementation of Prim's algorithm for an adjacency-matrix graph ADT implementation. It uses the arrays `st, fr`, and `wt` for these four data structures, with `st` and `fr` for the first two (respectively), and `wt` for both the third and fourth. For a tree vertex `v`, the entry `wt[v]` is the length of the parent link (corresponding to `st[v]`); for a nontree vertex `w`, the entry `wt[w]` is the distance to the tree (corresponding to `fr[w]`). The implementation is packaged as a function `GRAPHmstV` that takes `st` and `wt` as client-supplied arrays. If desired, we could add a wrapper function `GRAPHmst` to build an edge list (or some other) MST representation, as discussed in Section 20.1.

After adding a new edge (and vertex) to the tree, we have two tasks to accomplish:

- Check to see whether adding the new edge brought any nontree vertex closer to the tree.
- Find the next edge to add to the tree.

The implementation in *Program* 20.3 accomplishes both of these tasks with a single scan through the nontree vertices, updating `wt[w]` and `fr[w]` if `v-w` brings `w` closer to the tree, then updating the current minimum if `wt[w]` (the distance from `w` to `fr[w]` indicates that `w` is closer to the tree than any other vertex with a lower index.

Property 20.6 *Using Prim's algorithm, we can find the MST of a dense graph in linear time.*

Program 20.3 Prim's MST algorithm

This implementation of Prim's algorithm is the method of choice for dense graphs. The

```
static int fr[maxV];    #define P G->adj[v][w]    void GRAPHmstV (Graph
```

Proof: It is immediately evident from inspection of the program that the running time is

proportional to V^2 and therefore is linear for dense graphs.■

Figure 20.8 shows an example MST construction with Prim's algorithm; *Figure* 20.9 shows the evolving MST for a larger example.

Program 20.3 is based upon the observation that we can interleave the *find the minimum* and update operations in a single loop where we examine all the nontree edges. In a dense graph, the number of edges that we may have to examine to update the distance from the nontree vertices to the tree is proportional to V, so looking at all the nontree edges to find the one that is closest to the tree does not represent excessive extra cost. But in a sparse graph, we can expect to use substantially fewer than V steps to perform each of these operations. The crux of the strategy that we will use to do so is to focus on the set of potential edges to be added next to the MST—a set that we call the *fringe*. The number of fringe edges is typically substantially smaller than the number of nontree edges, and we can recast our description of the algorithm as follows. Starting with a self loop to a start vertex on the fringe and an empty tree, we perform the following operation until the fringe is empty:

Move a minimal edge from the fringe to the tree. Visit the vertex that it leads to, and put onto the fringe any edges that lead from that vertex to an nontree vertex, replacing the longer edge when two edges on the fringe point to the same vertex.

From this formulation, it is clear that Prim's algorithm is nothing more than a generalized graph search (see section 18.8), where the fringe is a priority queue based on a *delete the minimum* operation (see Chapter 9). We refer to generalized graph searching with priority queues as *priority-first search (PFS)*. With edge weights for priorities, PFS implements Prim's algorithm.

This formulation encompasses a key observation that we made already in connection with implementing BFS in section 18.7. An even simpler general approach is to simply keep on the fringe all of the edges that are incident upon tree vertices, letting the priority-queue mechanism find the shortest one and ignore longer ones (see Exercise 20.37). As we saw with BFS, this approach is unattractive because the fringe data structure becomes unnecessarily cluttered with edges that will never get to the MST. The size of the fringe could grow to be proportional to E (with whatever attendant costs having a fringe this size might involve), while the PFS approach just outlined ensures that the fringe will never have more than V entries.

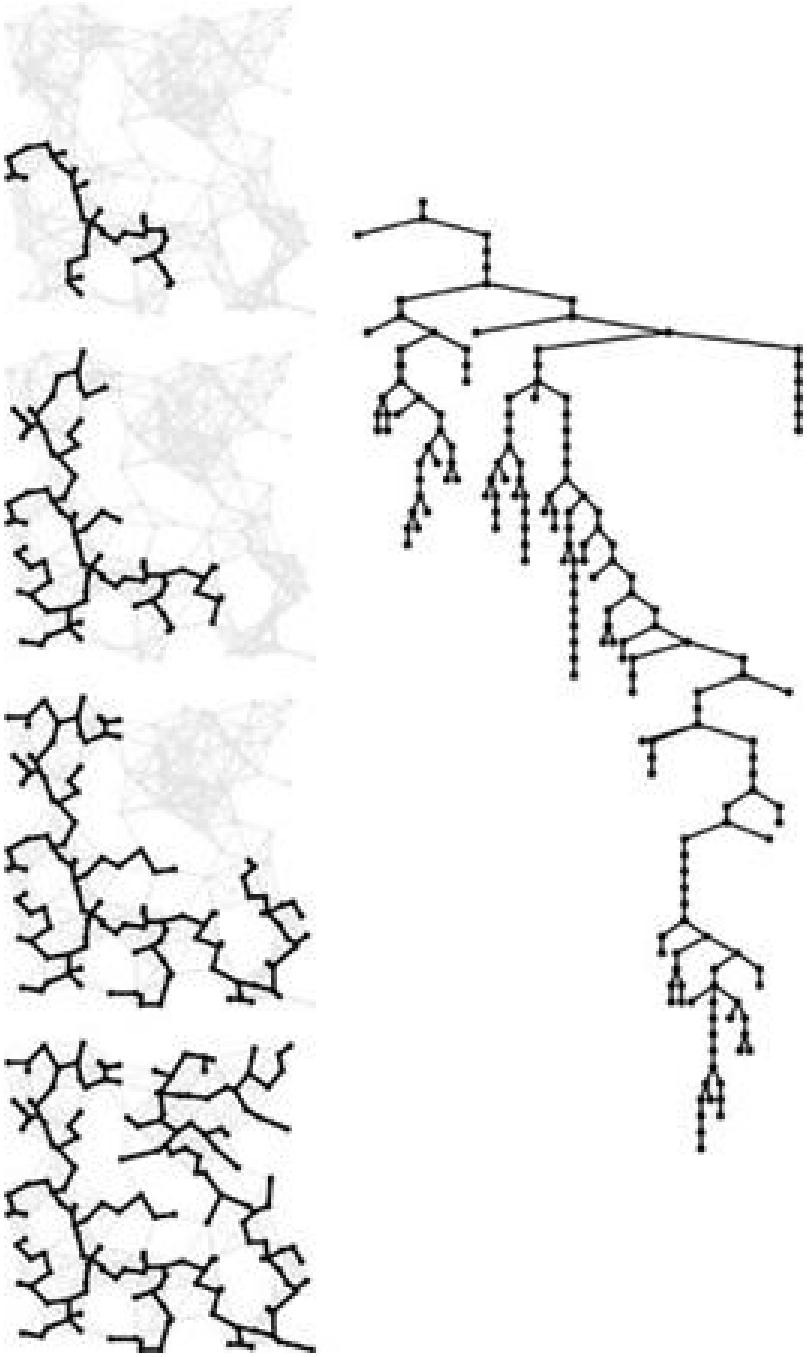
As with any implementation of a general algorithm, we have a number of available approaches for interfacing with priority-queue ADTs. One approach is to use a priority queue of edges, as in our generalized graph-search implementation of *Program* 18.10.

IPR2023-00124
KASPERSKY EX 1036 Page 299

Program 20.4 is an implementation that is essentially equivalent to *Program* 18.10 but uses a vertex-based approach so that it can use the index priority-queue interface of Section 9.6. We identify the *fringe vertices*, the subset of nontree vertices that are connected by fringe edges to tree vertices, and keep the same vertex-indexed arrays `st`, `fr`, and `wt` as in *Program* 20.3. The priority queue contains the index of each fringe vertex, and that entry gives access to the fringe vertex's closest tree vertex and distance to the tree, through the second and third arrays.

Figure 20.9 Prim's MST Algorithm *This sequence shows how the MST grows as Prim's algorithm discovers* 1/4, 1/2, 3/4, *and all of the edges in the MST* (*top to bottom*). *An oriented representation of the full MST is shown at the right*.

IPR2023-00124
KASPERSKY EX 1036 Page 300



Since we seek an MST, *Program* 20.4 assumes the graph to be connected. It acutally finds the MST in the connected component that contains vertex 0. To augment it to find minimal spanning forests in graphs that are not connected, we can add a loop as in *Program* 18.1 (see Exercise 20.29).

Property 20.7 *Using a PFS implementation of Prim's algorithm that uses a heap for the priority-queue implementation, we can compute an MST in time proportional to* $E \lg V$.

Proof: The algorithm directly implements the generic idea of Prim's algorithm (add next

IPR2023-00124
KASPERSKY EX 1036 Page 301

to the MST a minimal edge that connects a vertex on the MST with a vertex not on the MST). Each priority-queue operation requires less than lg *V* steps. Each vertex is chosen with a *delete the minimum* operation; and, in the worst case, each edge might require a *change priority* operation.■

Priority-first search is a proper generalization of breadth-first and depth-first search, because those methods also can be derived through appropriate priority settings. For example, we can (somewhat artificially) use a variable `cnt` to assign a unique priority `cnt++` to each vertex when we put that vertex on the priority queue. If we define `P` to be `cnt`, we get preorder numbering and DFS, because newly encountered nodes have the highest priority. If we define `P` to be `V-cnt`, we get BFS, because old nodes have the highest priority. These priority assignments make the priority queue operate like a stack and a queue, respectively. This equivalence is purely of academic interest, since the priority-queue operations are unnecessary for DFS and BFS. Also, as discussed in section 18.8, a formal proof of equivalence would require a precise attention to replacement rules, to obtain the same sequence of vertices as result from the classical algorithms.

As we shall see, PFS encompasses not just DFS, BFS, and Prim's MST algorithm, but also several other classical algorithms. The various algorithms differ only in their priority functions. Thus, the running times of all these algorithms depend on the performance of the priority-queue ADT. Indeed, we are led to a general result that encompasses not just the two implementations of Prim's algorithms that we have examined in this section, but also a broad class of fundamental graph-processing algorithms.

Program 20.4 Priority-first search (adjacency lists)

This program is a generalized graph search that uses a priority queue to manage the fringe (see section 18.8). The priority `P` is defined such that the ADT function `GRAPHpfs` implements Prim's MST algorithm for sparse (connected) graphs. Other priority definitions implement different graph-processing algorithms.

The program moves the highest priority (lowest weight) edge from the fringe to the tree, then checks every edge adjacent to the new tree vertex to see whether it implies changes in the fringe. Edges to vertices not on the fringe or the tree are added to the fringe; shorter edges to fringe vertices replace corresponding fringe edges.

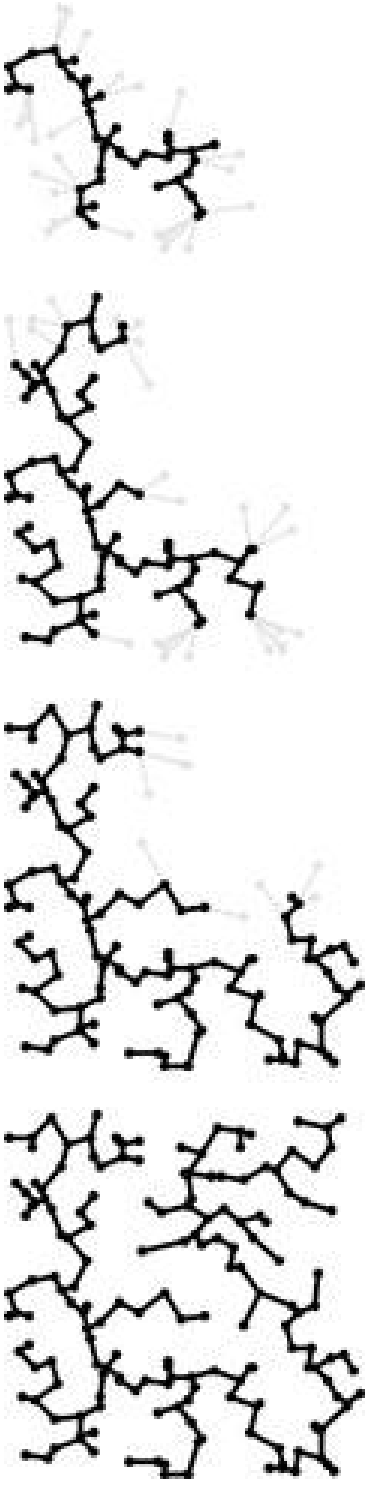
Property 20.8 *For all graphs and all priority functions, we can compute a spanning tree with PFS in linear time plus time proportional to the time required for V* **insert**, *V* **delete the minimum**, *and E* **decrease key** *operations in a priority queue of size at most V.*

Proof: The proof of Property 20.7 establishes this more general result. We have to examine all the edges in the graph; hence the "linear time" clause. The algorithm never increases the priority (it changes the priority to only a lower value); by more precisely specifying what we need from the priority-queue ADT (*decrease key*, not necessarily *change priority*), we strengthen this statement about performance.■

In particular, use of an unordered-array priority-queue implementation gives an optimal solution for dense graphs that has the same worst-case performance as the classical implementation of Prim's algorithm (*Program* 20.3). That is, Properties 20.6 and 20.7 are special cases of Property 20.8; throughout this book we shall see numerous other algorithms that essentially differ in only their choice of priority function and their priority-queue implementation.

Figure 20.10 PFS Implementation of Prim's MST Algorithm *With PFS, Prim's algorithm processes just the vertices and edges closest to the MST* (in gray).

IPR2023-00124
KASPERSKY EX 1036 Page 303



Property 20.7 is an important general result: The time bound stated is a worst-case upper bound that guarantees performance within a factor of lg *V* of optimal (linear time) for a large class of graph-processing problems. Still, it is somewhat pessimistic for many of the graphs that we encounter in practice, for two reasons. First, the lg *V* bound for priority-queue operation holds only when the number of vertices on the fringe is proportional to *V*, and even then is just an upper bound. For a real graph in a practical application, the fringe might be small (see Figures 20.10 and 20.11), and some priority-queue operations

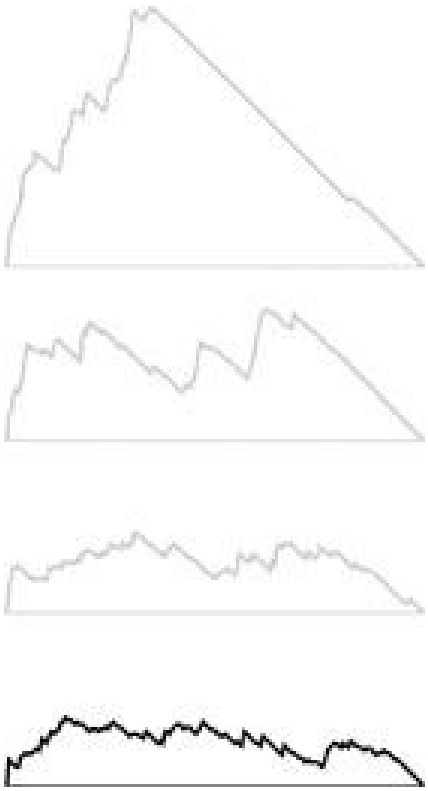
IPR2023-00124
KASPERSKY EX 1036 Page 304

might take many fewer than lg V steps. Although noticeable, this effect is likely to account for only a small constant factor in the running time; for example, a proof that the fringe never has more than $\sqrt{V}$ vertices on it would improve the bound by only a factor of 2. More important, we generally perform many fewer than E *decrease key* operations, since we do that operation only when we find an edge to a fringe node that is shorter than the current best-known edge to a fringe node. This event is relatively rare: Most edges have no effect on the priority queue (see Exercise 20.35). It is reasonable to regard PFS as an essentially linear-time algorithm unless V lg V is significantly greater than E.

The priority-queue ADT and generalized graph-searching abstractions make it easy for us to understand the relationships among various algorithms. Since these abstractions (and software mechanisms to support their use) were developed many years after the basic methods, relating the algorithms to classical descriptions of them becomes an exercise for historians. However, knowing basic facts about the history is useful when we encounter descriptions of MST algorithms in the research literature or in other texts, and understanding how these few simple abstractions tie together the work of numerous researchers over a time span of decades is persuasive evidence of their value and power; so we consider briefly the origins of these algorithms here.

An MST implementation for dense graphs essentially equivalent to *Program* 20.3 was first presented by Prim in 1961, and, independently, by Dijkstra soon thereafter. It is usually referred to as *Prim's algorithm*, although Dijkstra's presentation was more general, so some scholars refer to the MST algorithm as a special case of Dijkstra's algorithm. But the basic idea was also presented by Jarnik in 1939, so some authors refer to the method as *Jarnik's algorithm*, thus characterizing Prim's (or Dijkstra's) role as finding an efficient implementation of the algorithm for dense graphs. As the priority-queue ADT came into use in the early 1970s, its application to finding MSTs of sparse graphs was straightforward; the fact that MSTs of sparse graphs could be computed in time proportional to E lg V became widely known without attribution to any particular researcher. Since that time, as we discuss in Section 20.6, many researchers have concentrated on finding efficient priority-queue implementations as the key to finding efficient MST algorithms for sparse graphs.

Figure 20.11 Fringe Size for PFS Implementation of Prim's Algorithm
The plot at the bottom shows the size of the fringe as the PFS proceeds for the example in Figure 20.10. For comparison, the corresponding plots for DFS, randomized search, and BFS from Figure 18.28 are shown above in gray.



Exercises

▷ **20.30** Analyze the performance of the brute-force implementation of Prim's algorithm mentioned at the beginning of this section for a complete weighted graph with V vertices.

Hint: The following combinatorial sum might be useful: $\sum_{1 \le k < V} k\,(V - k) = (V + 1)V(V - 1)/6$.

○ **20.31** Answer Exercise 20.30 for graphs in which all vertices have the same fixed degree t.

○ **20.32** Answer Exercise 20.30 for general sparse graphs that have V vertices and E edges. Since the running time depends on the weights of the edges and on the degrees of the vertices, do a worst-case analysis. Exhibit a family of graphs for which your worst-case bound is confirmed.

20.33 Show, in the style of *Figure* 20.8, the result of computing the MST of the network defined in Exercise 20.21 with Prim's algorithm.

● **20.34** Describe a family of graphs with V vertices and E edges for which the worst-case running time of the PFS implementation of Prim's algorithm is confirmed.

●● **20.35** Develop a reasonable generator for random graphs with V vertices and E

edges such that the running time of the PFS implementation of Prim's algorithm (Program 20.4) is nonlinear.

20.36 Convert *Program* 20.4 for use in an adjacency-matrix graph ADT implementation.

○ **20.37** Modify *Program* 20.4 so that it works like *Program* 18.8, in that it keeps on the fringe all edges incident upon tree vertices. Run empirical studies to compare your implementation with *Program* 20.4, for various weighted graphs (see Exercise 20.9–14).

● **20.38** Provide an implementation of Prim's algorithm that makes use of your representation-independent graph ADT from Exercise 17.51.

20.39 Suppose that you use a priority-queue implementation that maintains a sorted list. What would be the worst-case running time for graphs with V vertices and E edges, to within a constant factor? When would this method be appropriate, if ever? Defend your answer.

▷ **20.40** An MST edge whose deletion from the graph would cause the MST weight to increase is called a *critical edge*. Show how to find all critical edges in a graph in time proportional to $E \lg V$.

20.41 Run empirical studies to compare the performance of *Program* 20.3 to that of *Program* 20.4, using an unordered array implementation for the priority queue, for various weighted graphs (see Exercise 20.9–14).

● **20.42** Run empirical studies to determine the effect of using an index-heap–tournament (see Exercise 9.53) priority-queue implementation instead of Program 9.12 in *Program* 20.4, for various weighted graphs (see Exercise 20.9–14).

20.43 Run empirical studies to analyze tree weights as a function of V, for various weighted graphs (see Exercise 20.9–14).

20.44 Run empirical studies to analyze maximum fringe size as a function of V, for various weighted graphs (see Exercise 20.9–14).

20.45 Run empirical studies to analyze tree height as a function of V, for various weighted graphs (see Exercise 20.9–14).

20.46 Run empirical studies to study the dependence of the results of Exercise 20.44 and 20.45 on the start vertex. Would it be worthwhile to use a random starting point?

● **20.47** Write a client program that does dynamic graphical animations of Prim's algo-

IPR2023-00124
KASPERSKY EX 1036 Page 307

rithm. Your program should produce images like *Figure* 20.10 (see Exercises 17.55 through 17.59). Test your program on random Euclidean neighbor graphs and on grid graphs (see Exercise 20.11 and 20.13), using as many points as you can process in a reasonable amount of time.

20.4 KRUSKAL'S ALGORITHM

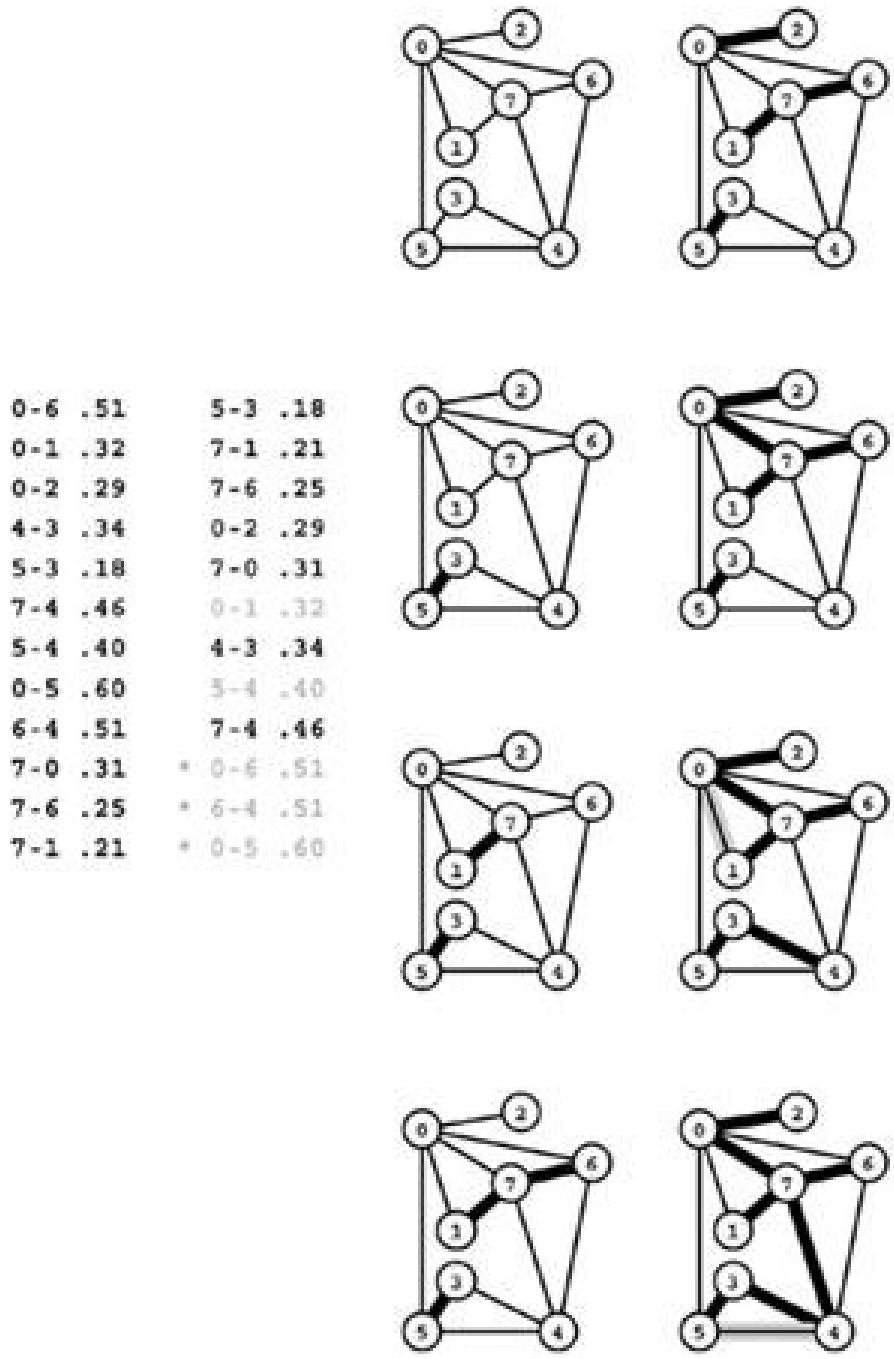
Prim's algorithm builds the MST one edge at a time, finding a new edge to attach to a single growing tree at each step. Kruskal's algorithm also builds the MST one edge at a time; but, by contrast, it finds an edge that connects two trees in a spreading forest of growing MST subtrees. We start with a degenerate forest of V single-vertex trees and perform the operation of combining two trees (using the shortest edge possible) until there is just one tree left: the MST.

Figure 20.12 shows a step-by-step example of the operation of Kruskal's algorithm; *Figure* 20.13 illustrates the algorithm's dynamic characteristics on a larger example. The disconnected forest of MST subtrees evolve gradually into a tree. Edges are added to the MST in order of their length, so the forests comprise vertices that are connected to one another by relatively short edges. At any point during the execution of the algorithm, each vertex is closer to some vertex in its subtree than to any vertex not in its subtree.

Kruskal's algorithm is simple to implement, given the basic algorithmic tools that we have considered in this book. Indeed, we can use any sort from Part 3 to sort the edges by weight, and any of the connectivity algorithms from Chapter 1 to eliminate those that cause cycles! *Program* 20.5 is an implementation along these lines of an MST function for a graph ADT that is functionally equivalent to the other MST implementations that we consider in this chapter. The implementation does not depend on the graph representation: It calls an ADT function to return an array that contains the graph's edges, then computes the MST from that array.

Figure 20.12 Kruskal's MST Algorithm *Given a list of a graph's edges in arbitrary order* (left edge list), *the first step in Kruskal's algorithm is to sort them by weight (right edge list). Then we go through the edges on the list in order of their weight, adding edges that do not create cycles to the MST. We add* `5-3` (*the shortest edge*), then `7-1`, *then* `7-6` (left), *then* `0-2` (right, top) *and* `0-7` (*right, second from top*). *The edge with the next largest weight*, `0-1`, *creates a cycle and is not included. Edges that we do not add to the MST are shown in gray on the sorted list. Then we add* `4-3` (*right, third from top*). *Next, we reject* `5-4` *because it causes a cycle, then we add* `7-4` (*right, bottom*). *Once the MST is complete, any edges with larger weights would cause cycles and be rejected (we stop when we have added V* – 1 *edges to the MST). These edges are marked with asterisks on the sorted list.*

IPR2023-00124
KASPERSKY EX 1036 Page 308



With Kruskal's algorithm, an array-of-edges representation is a more natural choice to use for the MST than the parent-link representation that we used for Prim's algorithm; so *Program* 20.5 is packaged to return the MST in a user-supplied array. Modifying the interface to be compatible with Programs 20.3 and 20.4 and the implementation to use DFS to compute a parent-link tree ADT for the MST is a trivial matter, as discussed in Section 20.1 (see Exercise 20.18 and 20.19).

Note that there are two ways in which Kruskal's algorithm can terminate. If we find $V - 1$ edges, then we have a spanning tree and can stop. If we examine all the edges without finding $V - 1$ tree edges, then we have determined that the graph is not connected, pre-

IPR2023-00124
KASPERSKY EX 1036 Page 309

cisely as we did in Chapter 1.

Analyzing the running time of Kruskal's algorithm is a simple matter because we know the running time of its constituent ADT operations.

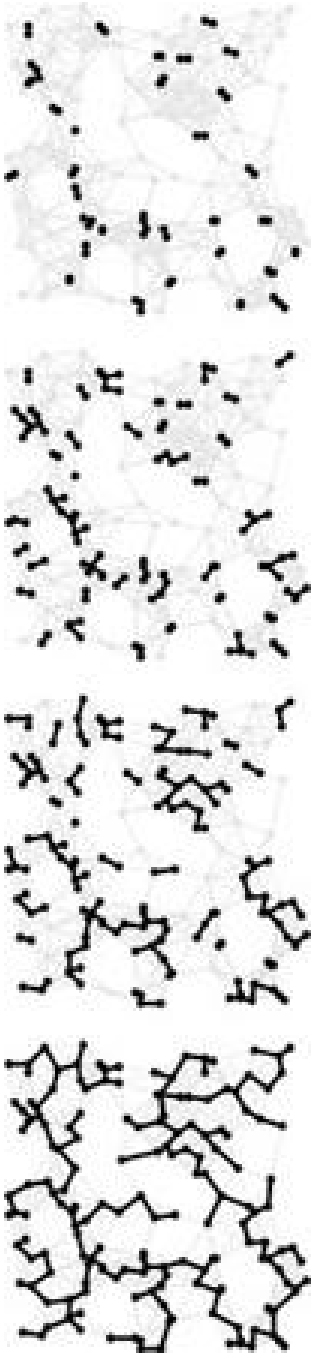
Property 20.9 *Kruskal's algorithm computes the MST of a graph in time proportional to* $E \lg E$.

Proof: This property is a consequence of the more general observation that the running time of *Program* 20.5 is proportional to the cost of sorting E numbers plus the cost of E *find* and $V - 1$ *union* operations. If we use standard ADT implementations such as merge-sort and weighted union-find with halving, the cost of sorting dominates.■

We consider the comparative performance of Kruskal's and Prim's algorithm in Section 20.6. For the moment, note that a running time proportional to $E \lg E$ is not necessarily worse than $E \lg V$, because E is at most V^2, so $\lg E$ is at most $2 \lg V$. Performance differences for specific graphs are due to what the properties of the implementations are and to whether the actual running time approaches these worst-case bounds.

Figure 20.13 Kruskal's MST Algorithm *This sequence shows* 1/4, 1/2, 3/4, *and the full MST as it evolves.*

IPR2023-00124
KASPERSKY EX 1036 Page 310



In practice, we might use quicksort or a fast system sort (which is likely to be based on quicksort). Although this approach may give the usual unappealing (in theory) quadratic worst case for the sort, it is likely to lead to the fastest run time. Indeed, on the other hand, we could use a radix sort to do the sort in linear time (under certain conditions on the weights) so the cost of the *E find* operations dominates, and then adjust Property 20.9 to say that the running time of Kruskal's algorithm is within a constant factor of $E \lg^* E$ under those conditions on the weights (see Chapter 2). Recall that the function $\lg^* E$ is the number of iterations of the binary logarithm function before the result is less than 1, which is less than 5 if E is less than 2^{65536}. In other words, these adjustments make Kruskal's algorithm effectively *linear* in most practical circumstances.

Program 20.5 Kruskal's MST algorithm

This implementation uses our sorting ADT from Chapter 6 and our union-find ADT from Chapter 4 to find the MST by considering the edges in order of their weights, discarding edges that create cycles until finding $V - 1$ edges that comprise a spanning tree.

The implementation is packaged as a function `GRAPHmstE` that returns the MST in a user-supplied array of edges. If desired, we can add a wrapper function `GRAPHmst` to build a parent-link (or some other representation) of the MST, as discussed in Section 20.1.

```
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];
        }
  }
```

Typically, the cost of finding the MST with Kruskal's algorithm is even lower than the cost of processing all edges, because the MST is complete well before a substantial fraction of the (long) graph edges is ever considered. We can take this fact into account to reduce the running time significantly in many practical situations, by keeping edges that are longer than the longest MST edge entirely out of the sort. One easy way to accomplish this objective is to use a priority queue, with an implementation that does the *construct* operation in linear time and the *delete the minimum* operation in logarithmic time.

For example, we can achieve these performance characteristics with a standard heap implementation, using bottom-up construction (see Section 9.4). Specifically, we make the

following changes to *Program* 20.5: First, we change the call on `sort` to a call on `PQconstruct`, to build a heap in time proportional to *E*. Second, we change the inner loop to take the shortest edge off the priority queue with `e = PQdelmin()` and to change all references to `a[i]` to refer to `e`.

Property 20.10 *A priority-queue–based version of Kruskal's algorithm computes the MST of a graph in time proportional to* $E + X \lg V$*, where X is the number of graph edges not longer than the longest edge in the MST.*

Proof: See the preceding discussion, which shows the cost to be the cost of building a priority queue of size *E* plus the cost of running the *X delete the minimum, X find*, and $V - 1$ *union* operations. Note that the priority-queue–construction costs dominate (and the algorithm is linear time) unless *X* is greater than $E / \lg V$.■

We can also apply the same idea to reap similar benefits in a quicksort-based implementation. Consider what happens when we use a straight recursive quicksort, where we partition at `i`, then recursively sort the subfile to the left of `i` and the subfile to the right of i. We note that, by construction of the algorithm, the first i elements are in sorted order after completion of the first recursive call (see Program 9.2). This obvious fact leads immediately to a fast implementation of Kruskal's algorithm: If we put the check for whether the edge `a[i]` causes a cycle between the recursive calls, then we have an algorithm that, by construction, has checked the first `i` edges, in sorted order, after completion of the first recursive call! If we include a test to return when we have found `V-1` MST edges, then we have an algorithm that sorts only as many edges as we need to compute the MST, with a few extra partitioning stages involving larger elements (see Exercise 20.52). Like straight sorting implementations, this algorithm could run in quadratic time in the worst case, but we can provide a probabilistic guarantee that the worst-case running time will not come close to this limit. Also, like straight sorting implementations, this program is likely to be faster than a heap-based implementation because of its shorter inner loop.

If the graph is not connected, the partial-sort versions of Kruskal's algorithm offer no advantage, because all the edges have to be considered. Even for a connected graph, the longest edge in the graph might be in the MST, so any implementation of Kruskal's method would still have to examine all the edges. For example, the graph might consist of tight clusters of vertices all connected together by short edges, with one outlier connected to one of the vertices by a long edge. Despite such anomalies, the partial-sort approach is probably worthwhile because it offers significant gain when it applies and incurs little if any extra cost.

Historical perspective is relevant and instructive here as well. Kruskal presented this algorithm in 1956, but, again, the relevant ADT implementations were not carefully studied

IPR2023-00124
KASPERSKY EX 1036 Page 313

for many years, so the performance characteristics of implementations such as the priority-queue version of *Program* 20.5 were not well understood until the 1970s. Other interesting historical notes are that Kruskal's paper mentioned a version of Prim's algorithm (see Exercise 20.54) and that Boruvka mentioned both approaches. Efficient implementations of Kruskal's method for sparse graphs preceded implementations of Prim's method for sparse graphs because union-find (and sort) ADTs came into use before priority-queue ADTs. Generally, as was true of implementations of Prim's algorithm, advances in the state of the art for Kruskal's algorithm are attributed primarily to advances in ADT performance. On the other hand, the applicability of the union-find abstraction to Kruskal's algorithm and the applicability of the priority-queue abstraction to Prim's algorithm have been prime motivations for many researchers to seek better implementations of those ADTs.

Exercises

▷ **20.48** Show, in the style of *Figure* 20.12, the result of computing the MST of the network defined in Exercise 20.21 with Kruskal's algorithm.

○ **20.49** Run empirical studies to analyze the length of the longest edge in the MST and the number of graph edges that are not longer than that one, for various weighted graphs (see Exercise 20.9–14).

● **20.50** Develop an implementation of the union-find ADT that implements *find* in constant time and *union* in time proportional to lg *V*.

20.51 Run empirical tests to compare your ADT implementation from Exercise 20.50 to weighted union-find with halving (Program 1.4) when Kruskal's algorithm is the client, for various weighted graphs (see Exercise 20.9–14). Separate out the cost of sorting the edges so that you can study the effects of the change both on the total cost and on the part of the cost associated with the union-find ADT.

20.52 Develop an implementation based on the idea described in the text where we integrate Kruskal's algorithm with quicksort so as to check MST membership of each edge as soon as we know that all smaller edges have been checked.

○ **20.53** Adapt Kruskal's algorithm to implement two ADT functions that fill a client-supplied vertex-indexed array classifying vertices into *k* clusters with the property that no edge of length greater than *d* connects two vertices in different clusters. For the first function, take *k* as an argument and return *d*; for the second, take *d* as an argument and return *k*. Test your program on random Euclidean neighbor graphs and on grid graphs (see Exercise 20.11 and 20.13) of various sizes for various values of *k* and *d*.

IPR2023-00124
KASPERSKY EX 1036 Page 314

20.54 Develop an implementation of Prim's algorithm that is based on pre-sorting the edges.

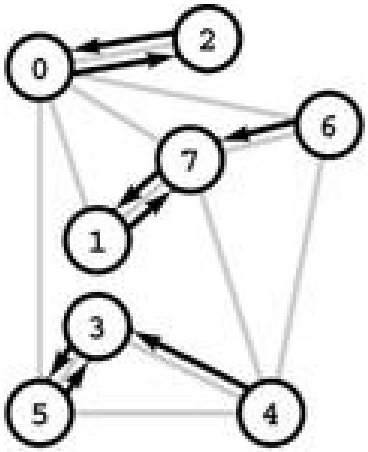
● **20.55** Write a client program that does dynamic graphical animations of Kruskal's algorithm (see Exercise 20.47). Test your program on random Euclidean neighbor graphs and on grid graphs (see Exercise 20.11 and 20.13), using as many points as you can process in a reasonable amount of time.

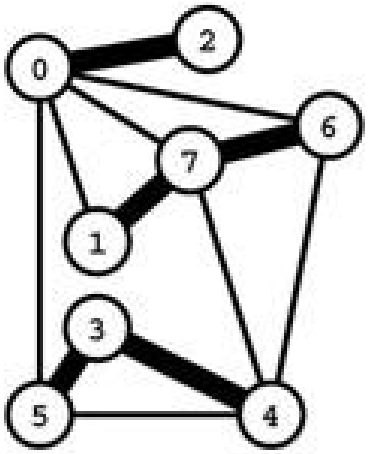
20.5 BORUVKA'S ALGORITHM

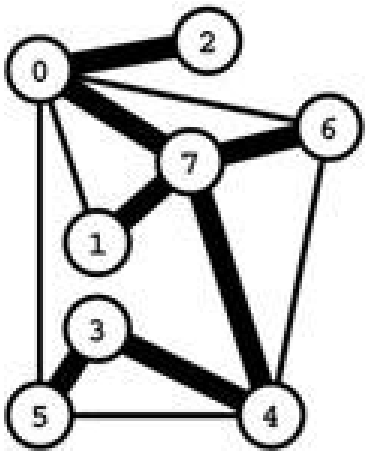
The next MST algorithm that we consider is also the oldest. Like Kruskal's algorithm, we build the MST by adding edges to a spreading forest of MST subtrees; but we do so in stages, adding several MST edges at each stage. At each stage, we find the shortest edge that connects each MST subtree with a different one, then add all such edges to the MST.

Again, our union-find ADT from Chapter 1 leads to an efficient implementation. For this problem, it is convenient to extend the interface to make the *find* operation available to clients. We use this function to associate an index with each subtree, so that we can tell quickly to which subtree a given vertex belongs. With this capability, we can implement efficiently each of the necessary operations for Boruvka's algorithm.

Figure 20.14 Boruvka's MST Algorithm The diagram at the top shows a directed edge from each vertex to its closest neighbor. These edges show that `0-2`, `1-7`, *and* `3-5` `are each the shortest edge incident on both their vertices`, `6-7` *is* 6*'s shortest edge, and* `4-3` *is* 4*'s shortest edge. These edges all belong to the MST and comprise a forest of MST subtrees* (*center*), *as computed by the first phase of Boruvka's algorithm. In the second phase, the algorithm completes the MST computation* (*bottom*) *by adding the edge* `0-7`, *which is the shortest edge incident on any of the vertices in the subtrees it connects, and the edge* `4-7`, *which is the shortest edge incident on any of the vertices in the bottom subtree.*







First, we maintain a vertex-indexed array that identifies, for each MST subtree, the nearest neighbor. Then, we perform the following operations on each edge in the graph:

- If it connects two vertices in the same tree, discard it.
- Otherwise, check the nearest-neighbor distances between the two trees the edge connects and update them if appropriate.

After this scan of all the graph edges, the nearest-neighbor array has the information that we need to connect the subtrees. For each vertex index, we perform a *union* operation to connect it with its nearest neighbor. In the next stage, we discard all the longer edges that connect other pairs of vertices in the now-connected MST subtrees. Figures 20.14 and 20.15 illustrate this process on our sample algorithm

Program 20.6 is a direct implementation of Boruvka's algorithm. The implementation is packaged as a function `GRAPHmstE` that uses an array of edges. It can be called from a `GRAPHmst` function that builds some other representation, as discussed in Section 20.1. There are three major factors that make this implementation efficient:

- The cost of each *find* operation is essentially constant.
- Each stage decreases the number of MST subtrees in the forest by at least a factor of 2.
- A substantial number of edges is discarded during each stage.

It is difficult to quantify precisely all these factors, but the following bound is easy to establish.

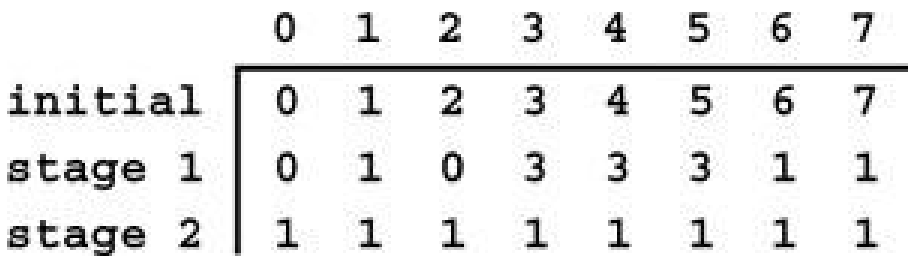
Property 20.11 *The running time of Boruvka's algorithm for computing the MST of a graph is* $O(E \lg V \lg^* E)$.

Proof: Since the number of trees in the forest is halved at each stage, the number of stages is no larger than $\lg V$. The time for each stage is at most proportional to the cost of E *find* operations, which is less than $E \lg^* E$, or linear for practical purposes.■

The running time given in Property 20.11 is a conservative upper bound, since it does not take into account the substantial reduction in the number of edges during each stage. The *find* operations take constant time in the early passes, and there are very few edges in the later passes. Indeed, for many graphs, the number of edges decreases exponentially with the number of vertices, and the total running time is proportional to E. For example, as illustrated in *Figure* 20.16, the algorithm finds the MST of our larger sample graph in just four stages.

It is possible to remove the $\lg^* E$ factor to lower the theoretical bound on the running time of Boruvka's algorithm to be proportional to $E \lg V$, by representing MST subtrees with doubly-linked lists instead of using the *union* and *find* operations. However, this improvement is sufficiently more complicated to implement and the potential performance improvement sufficiently marginal that it is not likely to be worth considering for use in practice (see Exercise 20.61 and 20.62).

Figure 20.15 Union-Find Array in Boruvka's Algorithm *This figure depicts the contents of the union-find array corresponding to the example depicted in Figure 20.14. Initially, each entry contains its own index, indicating a forest of singleton vertices. After the first stage, we have three components, represented by the indices* `0`, `1`, *and* `3` *(the union-find trees are all flat for this tiny example). After the second stage, we have just one component, represented by* `1`.



	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

Program 20.6 Boruvka's MST algorithm

This implementation of Boruvka's MST algorithm uses the union-find ADT from Chapter 4 to associate indices with MST subtrees as they are built. It assumes that find (which returns the same index for all the vertices belonging to each subtree) is in the interface.

Each phase corresponds to checking all the remaining edges; those that connect vertices in different components are kept for the next phase. The array a is used to store edges not yet discarded and not yet in the MST. The index `N` is used to store those being saved for the next phase (the code resets `E` from `N` at the end of each phase) and the index `h` is used to access the next edge to be checked. Each component's nearest neighbor is kept in the array `nn` with `find` component numbers as indices. At the end of each phase, we unite each component with its nearest neighbor and add the nearest-neighbor edges to the MST.

```
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; }
              }
          }
      }
```

As we mentioned, Boruvka's is the oldest of the algorithms that we consider: It was originally conceived in 1926, for a power-distribution application. The method was rediscovered by Sollin in 1961; it later attracted attention as the basis for MST algorithms with efficient asymptotic performance and as the basis for parallel MST algorithms.

Exercises

▷ **20.56** Show, in the style of *Figure* 20.14, the result of computing the MST of the network defined in Exercise 20.21 with Boruvka's algorithm.

○ **20.57** Why does *Program* 20.6 do a *find* test before doing the *union* operation? *Hint*: Consider equal-length edges.

○ **20.58** Why does *Program* 20.6 test that `v != G->V` before doing the *union* operation?

● **20.59** Describe a family of graphs with V vertices and E edges for which the number of edges that survive each stage of Boruvka's algorithm is sufficiently large that the worst-case running time is achieved.

20.60 Develop an implementation of Boruvka's algorithm that is based on a presorting of the edges.

○ **20.61** Develop an implementation of Boruvka's algorithm that uses doubly-linked circular lists to represent MST subtrees, so that subtrees can be merged and renamed in

time proportional to E during each stage (and the equivalence-relations ADT is therefore not needed).

○ **20.62** Do empirical studies to compare your implementation of Boruvka's algorithm in Exercise 20.61 with the implementation in the text (*Program* 20.6), for various weighted graphs (see Exercise 20.9–14).

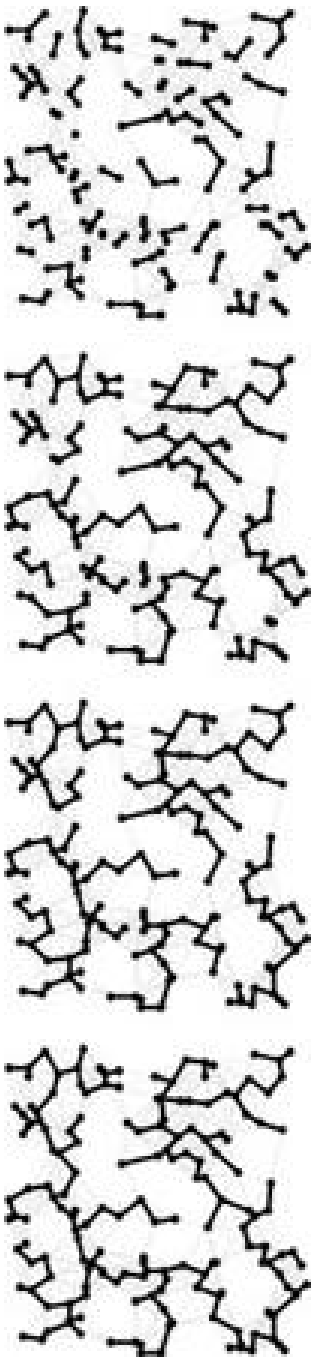
● **20.63** Do empirical studies to tabulate the number of stages and the number of edges processed per stage in Boruvka's algorithm, for various weighted graphs (see Exercise 20.9–14).

20.64 Develop an implementation of Boruvka's algorithm that constructs a new graph (one vertex for each tree in the forest) at each stage.

● **20.65** Write a client program that does dynamic graphical animations of Boruvka's algorithm (see Exercise 20.47 and 20.55). Test your program on random Euclidean neighbor graphs and on grid graphs (see Exercise 20.11 and 20.13), using as many points as you can process in a reasonable amount of time.

Figure 20.16 Boruvka's MST Algorithm *The MST evolves in just four stages for this example (top to bottom).*

IPR2023-00124
KASPERSKY EX 1036 Page 320



20.6 COMPARISONS AND IMPROVEMENTS

Table 20.1 summarizes the running times of the basic MST algorithms that we have considered; Table 20.2 presents the results of an empirical study comparing the algorithms. From these tables, we can conclude that the adjacency-matrix implementation of Prim's algorithm is the method of choice for dense graphs, that all the other methods perform within a small constant factor of the best possible (the time that it takes to extract the edges) for graphs of intermediate density, and that Kruskal's method essentially reduces the problem to sorting for sparse graphs.

IPR2023-00124
KASPERSKY EX 1036 Page 321

In short, we might consider the MST problem to be "solved" for practical purposes. For most graphs, the cost of finding the MST is only slightly higher than the cost of extracting the graph's edges. This rule holds except for huge graphs that are extremely sparse, but the available performance improvement over the best known algorithms even in this case is approximately a factor of 10 at best. The results in Table 20.2 are dependent on the model used to generate graphs, but they are borne out for many other graph models as well (see, for example, Exercise 20.74). Still, the theoretical results do not deny the existence of an algorithm that is guaranteed to run in linear time for all graphs; here we take a look at the extensive research on improved implementations of these methods.

First, much research has gone into developing better priority-queue implementations. The *Fibonacci heap* data structure, an extension of the binomial queue, achieves the theoretically optimal performance of taking constant time for *decrease key* operations and logarithmic time for *delete the minimum* operations, which behavior translates, by Property 20.8, to a running time proportional to $E + V \lg V$ for Prim's algorithm. Fibonacci heaps are more complicated than binomial queues and are somewhat unwieldy in practice, but some simpler priority-queue implementations have similar performance characteristics (*see reference section*).

One effective approach is to use radix methods for the priority-queue implementation. Performance of such methods is typically equivalent to that of radix sorting for Kruskal's method, or even to that of using a radix quicksort for the partial-sorting method that we discussed in Section 20.4.

Another simple early approach, proposed by D. Johnson in 1977, is one of the most effective: Implement the priority queue for Prim's algorithm with d-ary heaps, instead of with standard binary heaps (see *Figure* 20.17). For this priority-queue implementation, *decrease key* takes less than $\log_d V$ steps, and *delete the minimum* takes time proportional to $d \log_d V$. By Property 20.8, this behavior leads to a running time proportional to $Vd \log_d V + E \log_d V$ for Prim's algorithm, which is linear for graphs that are not sparse.

Table 20.1 Cost of MST algorithms This table summarizes the cost (worst-case running time) of various MST algorithms considered in this chapter. The formulas are based on the assumptions that an MST exists (which implies that $E \geq V - 1$) and that there are X edges not longer than the longest edge in the MST (see Property 20.10). These worst-case bounds may be too conservative to be useful in predicting performance on real graphs. The algorithms run in near-linear time in a broad variety of practical situations.

IPR2023-00124
KASPERSKY EX 1036 Page 322

algorithm	worst-case cost	comment
Prim (standard)	V^2	optimal for dense graphs
Prim (PFS, heap)	$E \lg V$	conservative upper bound
Prim (PFS, d-heap)	$E \log_d V$	linear unless exremely sparse
Kruskal	$E \lg E$	sort cost dominates
Kruskal (partial sort)	$E + X \lg V$	cost depends on longest edge
Boruvka	$E \lg V$	conservative upper bound

Property 20.12 *Given a graph with V vertices and E edges, let d denote the density E/V. If d* < 2*, then the running time of Prim's algorithm is proportional to V* lg *V. Otherwise, we can use a* ⌈*E/V*⌉*ary heap for the priority queue, which improves the worst-case running time by a factor of* lg (*E/V*).

Proof: Continuing the discussion in the previous paragraph, the number of steps is $Vd \log_d V + E \log_d V$, so the running time is at most proportional to $E \log_d V = (E \lg V)/\lg d$. ■

When E is proportional to $V^{1+\varepsilon}$, Property 20.12 leads to a worst-case running time proportional to E/ε and that value is linear for any constant ε. For example, if the number of edges is proportional to $V^{3/2}$, the cost is less than $2E$; if the number of edges is proportional to $V^{4/3}$, the cost is less than $3E$; and if the number of edges is proportional to $V^{5/4}$, the cost is less than $4E$. For a graph with 1 million vertices, the cost is less than $6E$ unless the density is less than 10.

Table 20.2 Empirical study of MST algorithms

This table shows relative timings for various algorithms for finding the MST, for random weighted graphs of various density. For low densities, Kruskal's algorithm is best because it amounts to a fast sort. For high densities, the classical implementation of Prim's algorithm is best because it does not incur list-processing overhead. For intermediate densities, the PFS implementation of Prim's algorithm runs within a small factor of the time that it takes to examine each graph edge.

IPR2023-00124
KASPERSKY EX 1036 Page 323

E	V	C	H	J	P	K	K*	e/E	B	e/E
density 2										
20000	10000	2	22	27		9	11	1.00	14	3.3
50000	25000	8	69	84		24	31	1.00	38	3.3
100000	50000	15	169	203		49	66	1.00	89	3.8
200000	100000	30	389	478		108	142	1.00	189	3.6
density 20										
20000	1000	2	5	4	20	6	5	.20	9	4.2
50000	2500	12	12	13	130	16	15	.28	25	4.6
100000	5000	14	27	28		34	31	.30	55	4.6
200000	10000	29	61	61		73	68	.35	123	5.0
density 100										
100000	1000	14	17	17	24	30	19	.06	51	4.6
250000	2500	36	44	44	130	81	53	.05	143	5.2
500000	5000	73	93	93		181	113	.06	312	5.5
1000000	10000	151	204	198		377	218	.06	658	5.6
density $V/2.5$										
400000	1000	61	60	59	20	137	78	.02	188	4.5
2500000	2500	597	409	400	128	1056	687	.01	1472	5.5

Key:

- C extract edges only
- H Prim's algorithm (adjacency lists/indexed heap)
- J Johnson's version of Prim's algorithm (d-heap priority queue)
- P Prim's algorithm (adjacency-matrix representation)
- K Kruskal's algorithm
- K* Partial-sort version of Kruskal's algorithm
- B Boruvka's algorithm
- e edges examined (*union* operations)

Program 20.7 Multiway heap PQ implementation

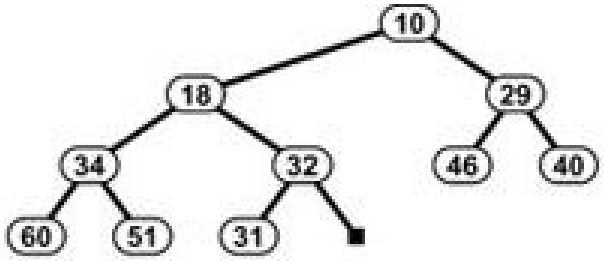
These `fixUp` and `fixDown` functions for the heap PQ implementation (see Program 9.5), maintain a d-way heap; so *delete the minimum* takes time proportional to d, but *decrease key* requires less than $\log_d V$ steps. For $d = 2$, these functions are equivalent to Programs 9.3 and 9.4, respectively.

IPR2023-00124
KASPERSKY EX 1036 Page 324

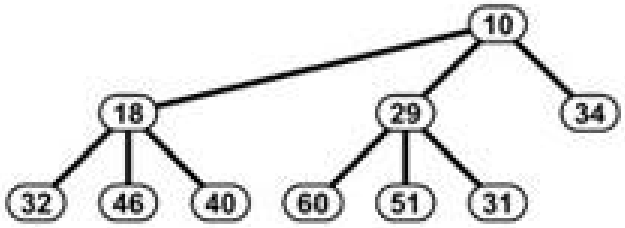
The temptation to minimize the bound on the worst-case running time in this way needs to be tempered with the realization that the $Vd \log_d V$ part of the cost is not likely to be avoided (for *delete the minimum*, we have to examine d successors in the heap as we sift down) but the $E \lg d$ part is not likely to be achieved (since most edges will not require a priority-queue update, as we showed in the discussion following Property 20.8).

Figure 20.17 2-, 3-, and 4-ary heaps *When we store a standard binary heap-ordered complete tree in an array (top), we use implicit links to take us from a node i down the tree to its children $2i$ and $2i + 1$ and up the tree to its parent $i/2$. In a 3-ary heap (center), implicit links for i are to its children $3i - 1$, $3i$, and $3i + 1$ and to its parent $\lfloor (i + 1)/3 \rfloor$; and in a 4-ary heap (bottom), implicit links for i are to its children $4i - 2$, $4i - 1$, $4i$, and $4i + 1$ and to its parent $\lfloor (i + 2)/4 \rfloor$. Increasing the branching factor in an implicit heap implementation can be valuable in applications, like Prim's algorithm, that require a significant number of decrease key operations.*

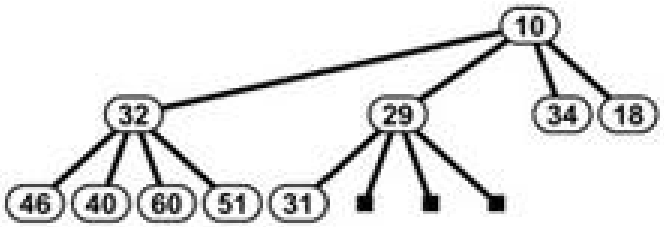
IPR2023-00124
KASPERSKY EX 1036 Page 325



1	2	3	4	5	6	7	8	9	10
10	18	29	34	32	46	40	60	51	31



1	2	3	4	5	6	7	8	9	10
10	18	29	34	32	46	40	60	51	31



1	2	3	4	5	6	7	8	9	10
10	32	29	34	18	46	40	60	51	31

For typical graphs such as those in the experiments in Table 20.2, decreasing *d* has no effect on the running time, and using a large value of *d* can slow down the implementation slightly. Still, the slight protection offered for worst-case performance makes the method worthwhile, since it is so easy to implement. In principle, we could tune the implementation to pick the best value of *d* for certain types of graphs (choose the largest value that does not slow down the algorithm), but a small fixed value (such as 4, 5, or 6) will be fine except possibly for some particular huge classes of graphs that have atypical characteristics.

Using *d* -heaps is not effective for sparse graphs because *d* has to be an integer greater

than or equal to 2, a condition that implies that we cannot bring the asymptotic running time lower than $V \lg V$. Ifthe density is a small constant, then a linear-time MST algorithm would have to run in time proportional to V.

The goal of developing practical algorithms for computing the MST of sparse graphs in linear time remains elusive. A great deal of research has been done on variations of Boruvka's algorithm as the basis for nearly linear-time MST algorithms for extremely sparse graphs (*see reference section*). Such research still holds the potential to lead us eventually to a practical linear-time MST algorithm and has even shown the existence of a randomized linear-time algorithm. While these algorithms are generally quite complicated, simplified versions of some of them may yet be shown to be useful in practice. In the meantime, we can use the basic algorithms that we have considered here to compute the MST in linear time in most practical situations, perhaps paying an extra factor of $\lg V$ for some sparse graphs.

Exercises

○ **20.66** [V. Vyssotsky] Develop an implementation of the algorithm discussed in Section 20.2 that builds the MST by adding edges one at a time and deleting the longest edges on the cycle formed (see Exercise 20.28). Use a parent-link representation of a forest of MST subtrees. *Hint*: Reverse pointers when traversing paths in trees.

20.67 Run empirical tests to compare the running time of your implementation in Exercise 20.66 with that of Kruskal's algorithm, for various weighted graphs (see Exercise 20.9–14). Check whether randomizing the order in which the edges are considered affects your results.

● **20.68** Describe how you would find the MST of a graph so large that only V edges can fit into main memory at once.

○ **20.69** Develop a priority-queue implementation for which *delete the minimum* and *find the minimum* are constant-time operations, and for which *decrease key* takes time proportional to the logarithm of the priority-queue size. Compare your implementation with 4-heaps when you use Prim's algorithm to find the MST of sparse graphs, for various weighted graphs (see Exercise 20.9–14).

20.70 Develop an implementation that generalizes Boruvka's algorithm to maintain a generalized queue containing the forest of MST subtrees. (Using *Program* 20.6 corresponds to using a FIFO queue.) Experiment with other generalized-queue implementations, for various weighted graphs (see Exercise 20.9–14).

● **20.71** Develop a generator for random connected cubic graphs (each vertex of degree

3) that have random weights on the edges. Fine-tune for this case the MST algorithms that we have discussed, then determine which is the fastest.

○ **20.72** For $V = 10^6$, plot the ratio of the upper bound on the cost for Prim's algorithm with d-heaps to E as a function of the density d, for d in the range from 1 to 100.

○ **20.73** Table 20.2 suggests that the standard implementation of Kruskal's algorithm is significantly faster than the partial-sort implementation for low-density graphs. Explain this phenomenon.

● **20.74** Run an empirical study, in the style of Table 20.2, for random complete graphs that have Gaussian weights (see Exercise 20.12).

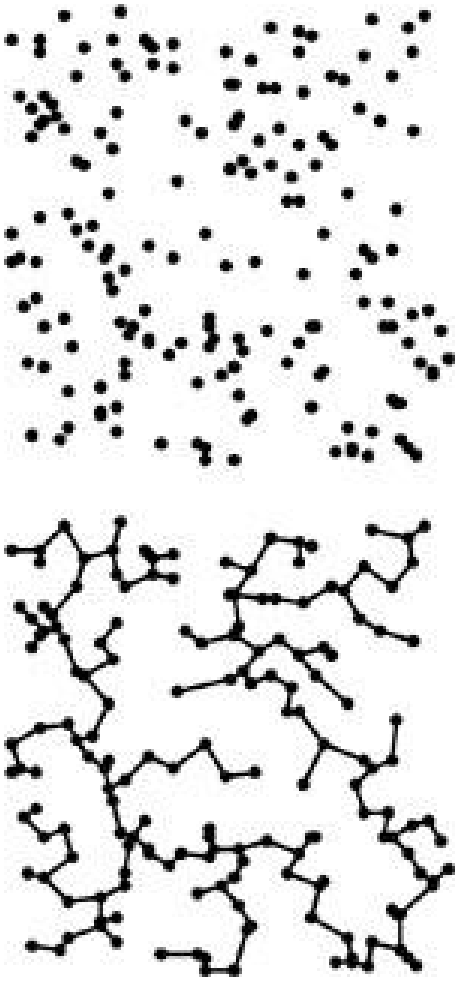
20.7 EUCLIDEAN MST

Suppose that we are given N points in the plane and we want to find the shortest set of lines connecting all the points. This geometric problem is called the *Euclidean MST* problem (see *Figure* 20.18). One way to solve it is to build a complete graph with N vertices and $N(N-1)/2$ edges—one edge connecting each pair of vertices weighted with the distance between the corresponding points. Then, we can use Prim's algorithm to find the MST in time proportional to N^2.

This solution is generally too slow. The Euclidean problem is somewhat different from the graph problems that we have been considering because all the edges are implicitly defined. The size of the input is just proportional to N, so the solution that we have sketched is a *quadratic* algorithm for the problem. Research has proved that it is possible to do better. The geometric structure makes most of the edges in the complete graph irrelevant to the problem, and we do not need to add most of them to the graph before we construct the minimum spanning tree.

Figure 20.18 Euclidean MST *Given a set on N points in the plane (top), the Euclidean MST is the shortest set of lines connecting them together (bottom). This problem is not just a graph-processing problem, because we need to make use of global geometric information about the points to avoid having to process all N^2 implicit edges connecting the points.*

IPR2023-00124
KASPERSKY EX 1036 Page 328



Property 20.13 *We can find the Euclidean MST of N points in time proportional to N* log *N*.

This fact is a direct consequence of two basic facts about points in the plane that we discuss in detail in Part 7. First, a graph known as the *Delauney triangulation* contains the MST, by definition. Second, the Delauney triangulation is a planar graph whose number of edges is proportional to *N*.■

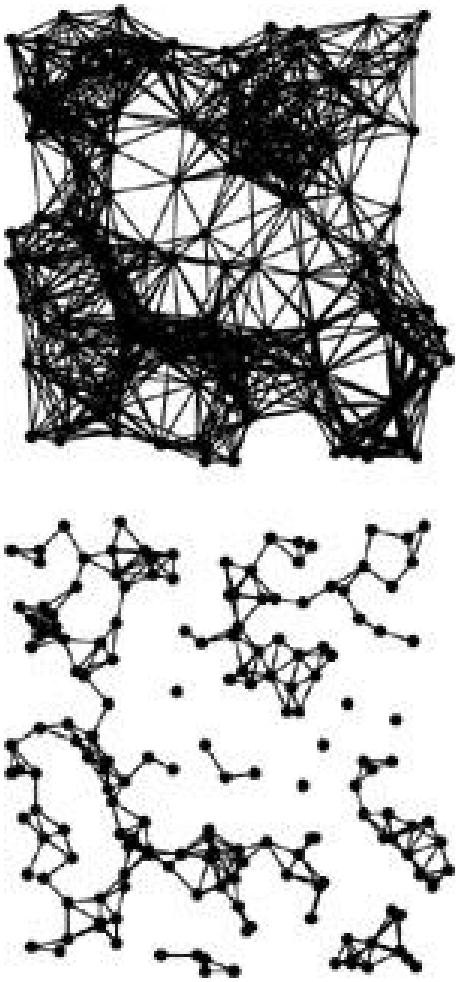
In principle, then, we could compute the Delauney triangulation in time proportional to *N* log *N*, then run either Kruskal's algorithm or the priority-first search method to find the Euclidean MST, in time proportional to *N* log *N*. But writing a program to compute the Delauney triangulation is a challenge for even an experienced programmer, so this approach may be overkill for this problem in practice.

Other approaches derive from the geometric algorithms that we consider in Part 7. For randomly distributed points, we can divide up the plane into squares such that each square is likely to contain about lg *N*/2 points, like we did for the closest-point computation in Program 3.20. Then, even if we include in the graph only the edges connecting

each point to the points in the neighboring squares, we are likely (but are not guaranteed) to get all the edges in the minimum spanning tree; in that case, we could use Kruskal's algorithm or the PFS implementation of Prim's algorithm to finish the job efficiently. The example that we have used in *Figure* 20.10, *Figure* 20.13, *Figure* 20.16 and similar figures was created in this way (see *Figure* 20.19). Or, we could develop a version of Prim's algorithm based on using near-neighbor algorithms to avoid updating distant vertices.

With all the possible choices that we have for approaching this problem and with the possibility of linear algorithms for the general MST problem, it is important to note that there is a simple lower bound on the best that we could do.

Figure 20.19 Euclidean Near-Neighbor Graphs *One way to compute the Euclidean MST is to generate a graph with edges connecting every pair of points within a distance d, as in the graph in Figure 20.8 et al. However, this method yields too many edges if d is too large* (top) *and is not guaranteed to have edges connecting all the points, if d is smaller than the longest edge in the MST* (bottom).



Property 20.14 *Finding the Euclidean MST of N points is no easier than sorting N numbers.*

Proof: Given a list of numbers to be sorted, convert the list into a list of points where the x coordinate is taken from the corresponding number of the list and the y coordinate is 0. Find the MST of that list of points. Then (as we did for Kruskal's algorithm), put the points into a graph ADT and run DFS to produce a spanning tree, starting at the point with the lowest x coordinate. That spanning tree amounts to a linked-list sort of the numbers in order; thus, we have solved the sorting problem.■

Precise interpretations of this lower bound are complicated because the basic operations used for the two problems (comparisons of coordinates for the sorting problem, distances for the MST problem) are different and because there is a possibility of using methods such as radix sort and grid methods. However, we may interpret the bound to mean that, as we do sorting, we should consider a Euclidean MST algorithm that uses $N \lg N$ comparisons to be optimal unless we exploit numerical properties of the coordinates, in which case we might expect to it to be linear time (*see reference section*).

It is interesting to reflect on the relationship between graph and geometric algorithms that is brought out by the Euclidean MST problem. Many of the practical problems that we might encounter could be formulated either as geometric problems or as graph problems. If the physical placement of objects is a dominating characteristic, then the geometric algorithms of Part 7 may be called for; but if interconnections between objects are of fundamental importance, then the graph algorithms of this section may be better.

The Euclidean MST seems to fall at the interface between these two approaches (the input involves geometry and the output involves interconnections), and the development of simple, straightforward methods for the Euclidean MST remains an elusive goal. In Chapter 21, we see a similar problem that falls at this interface, but where a Euclidean approach admits substantially faster algorithms than do the corresponding graph problems.

Exercises

▷ **20.75** Give a counterexample to show why the following method for finding the Euclidean MST does not work: "Sort the points on their x coordinates, then find the minimum spanning trees of the first half and the second half, then find the shortest edge that connects them."

○ **20.76** Develop a fast version of Prim's algorithm for computing the Euclidean MST of a uniformly distributed set of points in the plane based on ignoring distant points until the tree approaches them.

●● **20.77** Develop an algorithm that, given a set of N points in the plane, finds a set of edges of cardinality proportional to N that is certain to contain the MST and is sufficiently

easy to compute that you can develop a concise and efficient implementation of your algorithm.

○ **20.78** Given a random set of N points in the unit square (uniformly distributed), empirically determine a value of d, to within two decimal places, such that the set of edges defined by all pairs of points within distance d of one another is 99 percent certain to contain the MST.

○ **20.79** Work Exercise 20.78 for points where each coordinate is drawn from a Gaussian distribution with mean 0.5 and standard deviation 0.1.

● **20.80** Describe how you would improve the performance of Kruskal's and Boruvka's algorithm for sparse Euclidean graphs.

IPR2023-00124
KASPERSKY EX 1036 Page 332

Chapter Twenty-One
Shortest Paths

EVERY PATH IN a weighted digraph has an associated *path weight*, the value of which is sum of the weights of that path's edges. This essential measure allows us to formulate such problems as "find the lowest-weight `path` between two given vertices." These *shortest-paths problems* are the topic of this chapter. Not only are shortest-paths problems intuitive for many direct applications, but they also take us into a powerful and general realm where we seek efficient algorithms to solve general problems that can encompass a broad variety of specific applications.

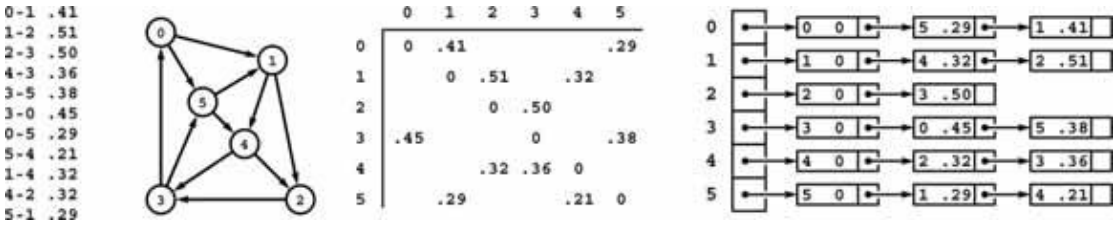
Several of the algorithms that we consider in this chapter relate directly to various algorithms that we examined in Chapters 17 through 20. Our basic graph-search paradigm applies immediately, and several of the specific mechanisms that we used in Chapters 17 and 19 to address connectivity in graphs and digraphs provide the basis for us to solve shortest-paths problems.

For economy, we refer to weighted digraphs as *networks*. *Figure* 21.1 shows a sample network, with standard representations. It is a simple matter to derive the basic ADT functions that we need to process networks from corresponding functions for the undirected representations that we considered in Section 20.1—we just keep one representation of each edge, precisely as we did when deriving digraph representations in Chapter 19 from the undirected graph representations in Chapter 17 (see Programs 20.1 and 20.2). In applications or systems for which we need all types of graphs, it is a textbook exercise in software engineering to define a network ADT from which ADTs for the unweighted undirected graphs of Chapters 17 and 18, the un-weighted digraphs of Chapter 19, or the weighted undirected graphs of Chapter 20 can be derived (see Exercise 21.9).

Figure 21.1 Sample Network and Representations *This network (weighted digraph) is shown in four representations: list of edges, drawing, adjacency matrix, and adjacency lists (left to right). As we did for MST algorithms, we often use edge weights that are proportional to their lengths in the drawing, but we do not insist on this rule because most shortest-paths algorithms handle arbitrary nonnegative weights (negative weights do present special challenges). The adjacency-matrix and adjacency-lists representations include weights with each edge representation, as in weighted undirected graphs. The adjacency matrix is not symmetric, and the adjacency lists contain one node for each edge (as in un-weighted digraphs). Nonexistent edges are represented by a sentinel value in the matrix (blank in the figure) and are not present at all in the lists. Self-loops of length 0 are present because they simplify our implementations of shortest-paths algorithms. They are omitted from the list of edges at left for economy and to indicate the typical scenario where we add them by convention when we create an adjacency-ma-*

IPR2023-00124
KASPERSKY EX 1036 Page 333

trix or adjacency-lists representation.



When we work with networks, it is generally convenient to keep self-loops in all the representations. This convention allows algorithms the flexibility to use a sentinel maximum-value weight to indicate that a vertex cannot be reached from itself. In our examples, we use self-loops of weight 0, although positive-weight self-loops certainly make sense in many applications. Many applications also call for parallel edges, perhaps with differing weights. As we mentioned in Section 20.1, various options for ignoring or combining such edges are appropriate in various different applications. In this chapter, for simplicity, none of our examples use parallel edges, and we do not allow parallel edges in the adjacency-matrix representation; we also do not check for parallel edges or remove them in adjacency lists.

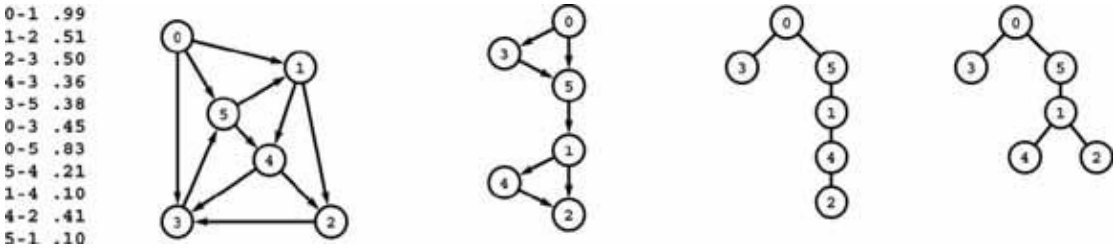
All the connectivity properties of digraphs that we considered in Chapter 19 are relevant in networks. In that chapter, we wished to know whether it is *possible* to get from one vertex to another; in this chapter, we take weights into consideration—we wish to find the *best* way to get from one vertex to another.

Definition 21.1 *A* **shortest `path`** *between two vertices s and t in a network is a directed simple* `path` *from s to t with the property that no other such* `path` *has a lower weight*.

This definition is succinct, but its brevity masks points worth examining. First, if *t* is not reachable from *s*, there is no `path` at all, and therefore there is no shortest path. For convenience, the algorithms that we consider often treat this case as equivalent to one in which there exists an infinite-weight `path` from *s* to *t*. Second, as we did for MST algorithms, we use networks where edge weights are proportional to edge lengths in examples, but the definition has no such requirement and our algorithms (other than the one in section 21.5) do not make this assumption. Indeed, shortest-paths algorithms are at their best when they discover counterintuitive shortcuts, such as a `path` between two vertices that passes through several other vertices but has total weight smaller than that of a direct edge connecting those vertices. Third, there may be multiple paths of the same weight from one vertex to another; we typically are content to find one of them. *Figure* 21.2 shows an example with general weights that illustrates these points.

IPR2023-00124
KASPERSKY EX 1036 Page 334

Figure 21.2 Shortest-Path Trees *A shortest-path tree (SPT) defines shortest paths from the root to other vertices* (*see Definition* 21.2). *In general, different paths may have the same length, so there may be multiple SPTs defining the shortest paths from a given vertex. In the example network shown at left, all shortest paths from* `0` *are subgraphs of the DAG shown to the right of the network. A tree rooted at* `0` *spans this DAG if and only if it is an SPT for* `0`. *The two trees at right are such trees.*



The restriction in the definition to simple paths is unnecessary in networks that contain edges that have nonnegative weight, because any cycle in a `path` in such a network can be removed to give a `path` that is no longer (and is shorter unless the cycle comprises zero-weight edges). But when we consider networks with edges that could have negative weight, the need for the restriction to simple paths is readily apparent: Otherwise, the concept of a shortest `path` is meaningless if there is a cycle in the network that has negative weight. For example, suppose that the edge `3-5` in the network in *Figure* 21.1 were to have weight `-.38`, and edge `5-1` were to have weight `-.31`. Then, the weight of the cycle `1-4-3-5-1` would be `.32 + .36 - .38 - .31 = -.01`, and we could spin around that cycle to generate arbitrarily short paths. Note carefully that, as is true in this example, it is *not* necessary for all the edges on a negative-weight cycle to be of negative weight; what counts is the *sum* of the edge weights. For brevity, we use the term *negative cycle* to refer to directed cycles whose total weight is negative.

In the definition, suppose that some vertex on a `path` from *s* to *t* is also on a negative cycle. In this case, the existence of a (nonsimple) shortest `path` from *s* to *t* would be a contradiction, because we could use the cycle to construct a `path` that had a weight lower than any given value. To avoid this contradiction, we include in the definition the restriction to simple paths, so that the concept of a shortest `path` is well defined in any network. However, we do not consider negative cycles in networks until section 21.7, because, as we see there, they present a truly fundamental barrier to the solution of shortest-paths problems.

To find shortest paths in a weighted undirected graph, we build a network with the same vertices and with two edges (one in each direction) corresponding to each edge in the graph. There is a one-toone correspondence between simple paths in the network and simple paths in the graph, and the costs of the paths are the same; so shortest-paths

problems are equivalent. Indeed, we build precisely such a network when we build the standard adjacency-lists or adjacency-matrix representation of a weighted undirected graph (see, for example, *Figure* 20.3). This construction is not helpful if weights can be negative, because it gives negative cycles in the network, and we do not know how to solve shortest-paths problems in networks that have negative cycles (see section 21.7). Otherwise, the algorithms for networks that we consider in this chapter also work for weighted undirected graphs.

In certain applications, it is convenient to have weights on vertices instead of, or in addition to, weights on edges; and we might also consider more complicated problems where both the number of edges on the `path` and the overall weight of the `path` play a role. We can handle such problems by recasting them in terms of edge-weighted networks (see, for example, Exercise 21.3) or by slightly extending the basic algorithms (see, for example, Exercise 21.52).

Because the distinction is clear from the context, we do not introduce special terminology to distinguish shortest paths in weighted graphs from shortest paths in graphs that have no weights (where a path's weight is simply its number of edges (see Section 17.7)). The usual nomenclature refers to (edge-weighted) networks, as used in this chapter, since the special cases presented by undirected or unweighted graphs are handled easily by algorithms that process networks (see, for example, Exercise 21.9).

We are interested in the same basic problems that we defined for undirected and unweighted graphs in section 18.7. We restate them here, noting that Definition 21.1 implicitly generalizes them to take weights into account in networks.

Figure 21.3 All Shortest Paths *This table gives all the shortest paths in the network of Figure 21.1 and their lengths. This network is strongly connected, so there exist paths connecting each pair of vertices. The goal of a source-sink shortest-path algorithm is to compute one of the entries in this table; the goal of a single-source shortest-paths algorithm is to compute one of the rows in this table; and the goal of an all-pairs shortest-paths algorithm is to compute the whole table. Generally, we use more compact representations, which contain essentially the same information and allow clients to trace any* `path` *in time proportional to its number of edges (see Figure 21.8).*

	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

Source-sink shortest path Given a start vertex *s* and a finish vertex *t*, find a shortest `path` in the graph from *s* to *t*. We refer to the start vertex as the *source* and to the finish vertex as the *sink*, except in contexts where this usage conflicts with the definition of

sources (vertices with no incoming edges) and sinks (vertices with no outgoing edges) in digraphs.

Single-source shortest paths Given a start vertex *s*, find shortest paths from *s* to each other vertex in the graph.

All-pairs shortest paths Find shortest paths connecting each pair of vertices in the graph. For brevity, we sometimes use the term *all shortest paths* to refer to this set of V^2 paths.

If there are multiple shortest paths connecting any given pair of vertices, we are content to find any one of them. Since paths have varying number of edges, our implementations provide ADT functions that allow clients to trace paths in time proportional to the paths' lengths. Any shortest `path` also implicitly gives us the shortest-path length, but our implementations explicitly provide lengths. In summary, to be precise, when we say "find a shortest path" in the problem statements just given, we mean "compute the shortest-path length and a way to trace a specific `path` in time proportional to that path's length."

Figure 21.3 illustrates shortest paths for the example network in *Figure* 21.1. In networks with V vertices, we need to specify V paths to solve the single-source problem, and to specify V^2 paths to solve the all-pairs problem. In our implementations, we use a representation more compact than these lists of paths; we first noted it in section 18.7, and we consider it in detail in Section 21.1.

In modern implementations, we build our algorithmic solutions to these problems into ADT implementations that allow us to build efficient client programs that can solve a variety of practical graph-processing problems. For example, as we see in section 21.3, an attractive way to package a solution to the all-pairs shortest-paths problem is as a preprocessing function in an ADT interface that provides a constant-time shortest-path query implementation. We might also provide a preprocessing function to solve single-source problems, so that clients that need to compute shortest paths from a specific vertex (or a small set of them) can avoid the expense of computing shortest paths for other vertices. Careful consideration of these issues and proper use of the algorithms that we examine can mean the difference between an efficient solution to a practical problem and a solution that is so costly that no client could afford to use it.

Shortest-paths problems arise in various guises in numerous applications. Many of the applications appeal immediately to geometric intuition, but many others involve arbitrary cost structures. As we did with minimum spanning trees (MSTs) in Chapter 20, we sometimes take advantage of geometric intuition to help develop an understanding of algorithms that solve the problems, but stay cognizant that our algorithms operate properly in

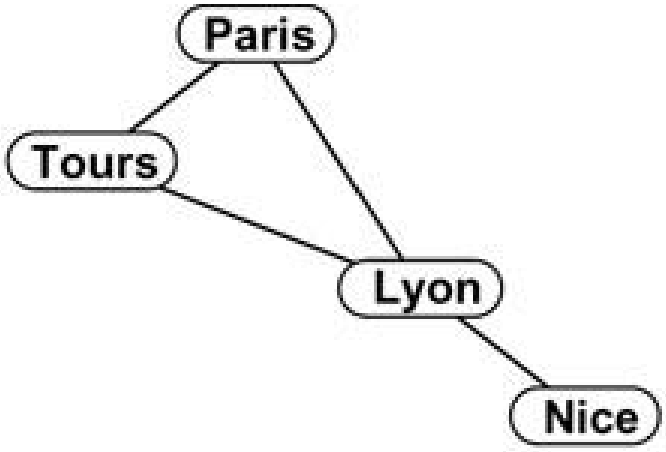
more general settings. In section 21.5, we do consider specialized algorithms for Euclidean networks. More important, in Sections 21.6 and 21.7, we see that the basic algorithms are effective for numerous applications where networks represent an abstract model of the computation.

Road maps Tables that give distances between all pairs of major cities are a prominent feature of many road maps. We presume that the map maker took the trouble to be sure that the distances are the shortest ones, but our assumption is not necessarily always valid (see, for example, Exercise 21.10). Generally, such tables are for undirected graphs that we should treat as networks with edges in both directions corresponding to each road, though we might contemplate handling one-way streets for city maps and some similar applications. As we see in section 21.3, it is not difficult to provide other useful information, such as a table that tells how to execute the shortest paths (see *Figure* 21.4). In modern applications, embedded systems provide this kind of capability in cars and transportation systems. Maps are Euclidean graphs; in section 21.4, we examine shortest-paths algorithms that take into account the vertex position when they seek shortest paths.

Airline routes Route maps and schedules for airlines or other transportation systems can be represented as networks for which various shortest-paths problems are of direct importance. For example, we might wish to minimize the time that it takes to fly between two cities, or to minimize the cost of the trip. Costs in such networks might involve functions of time, of money, or of other complicated resources. For example, flights between two cities typically take more time in one direction than the other, because of prevailing winds. Air travelers also know that the fare is not necessarily a simple function of the distance between the cities—situations where it is cheaper to use a circuitous route (or endure a stopover) than to take a direct flight are all too common. Such complications can be handled by the basic shortest-paths algorithms that we consider in this chapter; these algorithms are designed to handle any positive costs.

Figure 21.4 Distances and Paths *Road maps typically contain distance tables like the one in the center, for this tiny subset of French cities connected by highways as shown in the graph at the top. Though rarely found in maps, a table like the one at the bottom would also be useful, as it tells what signs to follow to execute the shortest path. For example, to decide how to get from Paris to Nice, we can check the table, which says to begin by following signs to Lyon.*

IPR2023-00124
KASPERSKY EX 1036 Page 338



	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	

The fundamental shortest-paths computations suggested by these applications only scratch the surface of the applicability of shortest-paths algorithms. In section 21.6, we consider problems from applications areas that appear unrelated to these natural ones, in the context of a discussion of *reduction*, a formal mechanism for proving relationships among problems. We solve problems for these applications by transforming them into abstract shortest-paths problems that do not have the intuitive geometric feel of the problems just described. Indeed, some applications lead us to consider shortest-paths problems in networks with negative weights. Such problems can be far more difficult to solve than are problems where negative weights cannot occur. Shortest-paths problems for such applications not only bridge a gap between elementary algorithms and unsolved al-

gorithmic challenges, but also lead us to powerful and general problem-solving mechanisms.

As we did with MST algorithms in Chapter 20, we often mix the weight, cost, and distance metaphors. Again, we normally exploit the natural appeal of geometric intuition even when working in more general settings with arbitrary edge weights; thus we refer to the "length" of paths and edges when we should say "weight" and to one `path` as "shorter" than another when we should say that it "has lower weight." We also might say that v is "closer" to s than w when we should say that "the lowest-weight directed `path` from s to v has weight lower than that of the lowest-weight directed `path` s to w," and so forth. This usage is inherent in the standard use of the term "shortest paths" and is natural even when weights are not related to distances (see *Figure* 21.2); however, when we expand our algorithms to handle negative weights in section 21.6, we must abandon such usage.

This chapter is organized as follows. After introducing the basic underlying principles in Section 21.1, we introduce basic algorithms for the single-source and all-pairs shortest-paths problems in Sections 21.2 and 21.3. Then, we consider acyclic networks (or, in a clash of shorthand terms, weighted DAGs) in section 21.4 and ways of exploiting geometric properties for the source–sink problem in Euclidean graphs in section 21.5. We then cast off in the other direction to look at more general problems in Sections 21.6 and 21.7, where we explore shortest-paths algorithms, perhaps involving networks with negative weights, as a high-level problem-solving tools.

EXERCISES

▷ **21.1** Label the following points in the plane 0 through 5, respectively:

```
(1, 3) (2, 1) (6, 5) (3, 4) (3, 7) (5, 3).
```

Taking edge lengths to be weights, consider the network defined by the edges

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

Draw the network and give the adjacency-lists structure that is built by *Program* 17.8, modified as appropriate to process networks.

21.2 Show, in the style of *Figure* 21.3, all shortest paths in the network defined in Exercise 21.1.

○ **21.3** Show that shortest-paths computations in networks with nonnegative weights on *both* vertices and edges (where the weight of a `path` is defined to be the sum of the weights of the vertices and the edges on the path) can be handled by building a network ADT that has weights on only the edges.

21.4 Find a large network online—perhaps a geographic database with entries for roads that connect cities or an airline or railroad schedule that contains distances or costs.

21.5 Implement a network ADT for sparse networks with weights between 0 and 1, based on *Program* 17.6. Include a random-network generator based on *Program* 17.7. Use a separate ADT to generate edge weights, and write two implementations: one that generates uniformly distributed weights and another that generates weights according to a Gaussian distribution. Write client programs to generate random networks for both weight distributions with a well-chosen set of values of V and E, so that you can use them to run empirical tests on graphs drawn from various distributions of edge weights.

○ **21.6** Implement a network ADT for dense graphs with weights between 0 and 1, based on *Program* 17.3. Include a random-network generator based on *Program* 17.8 and edge-weight generators as described in Exercise 21.5. Write client programs to generate random networks for both weight distributions with a well-chosen set of values of V and E, so that you can use them to run empirical tests on graphs drawn from these models.

21.7 Implement a representation-independent network ADT function that builds a network by taking edges with weights (pairs of integers between 0 and $V - 1$ with weights between 0 and 1) from standard input.

● **21.8** Write a program that generates V random points in the plane, then builds a network with edges (in both directions) connecting all pairs of points within a given distance d of one another (see Exercise 17.72), setting each edge's weight to the distance between the two points that that edge connects. Determine how to set d so that the expected number of edges is E.

○ **21.9** Write functions that use a network ADT to implement ADTs for undirected graphs (weighted and unweighted) and for unweighted digraphs.

▷ **21.10** The following table from a published road map purports to give the length of the shortest routes connecting the cities. It contains an error. Correct the table. Also, add a table that shows how to execute the shortest routes, in the style of *Figure* 21.4.

IPR2023-00124
KASPERSKY EX 1036 Page 341

	Providence	Westerly	New London	Norwich
Providence	–	53	54	48
Westerly	53	–	18	101
New London	54	18	–	12
Norwich	48	101	12	–

21.1 UNDERLYING PRINCIPLES

Our shortest-paths algorithms are based on a simple operation known as *relaxation*. We start a shortest-paths algorithm knowing only the network's edges and weights. As we proceed, we gather information about the shortest paths that connect various pairs of vertices. Our algorithms all update this information incrementally, making new inferences about shortest paths from the knowledge gained so far. At each step, we test whether we can find a `path` that is shorter than some known path. The term "relaxation" is commonly used to describe this step, which *relaxes* constraints on the shortest path. We can think of a rubber band stretched tight on a `path` connecting two vertices: A successful relaxation operation allows us to relax the tension on the rubber band along a shorter path.

Our algorithms are based on applying repeatedly one of two types of relaxation operations:

- *Edge relaxation:* Test whether traveling along a given edge gives a new shortest `path` to its destination vertex.

- *Path relaxation:* Test whether traveling through a given vertex gives a new shortest `path` connecting two other given vertices.

Edge relaxation is a special case of `path` relaxation; we consider the operations separately, however, because we use them separately (the former in single-source algorithms; the latter in all-pairs algorithms). In both cases, the prime requirement that we impose on the data structures that we use to represent the current state of our knowledge about a network's shortest paths is that we can update them easily to reflect changes implied by a relaxation operation.

First, we consider edge relaxation, which is illustrated in *Figure* 21.5. All the single-source shortest-paths algorithms that we consider are based on this step: Does a given edge lead us to consider a shorter `path` to its destination from the source?

The data structures that we need to support this operation are straightforward. First, we have the basic requirement that we need to compute the shortest-paths lengths from the

IPR2023-00124
KASPERSKY EX 1036 Page 342

source to each of the other vertices. Our convention will be to store in a vertex-indexed array `wt` the lengths of the shortest known paths from the source to each vertex. Second, to record the paths themselves as we move from vertex to vertex, our convention will be the same as the one that we used for other graph-search algorithms that we examined in Chapters 18 through 20: We use a vertex-indexed array `st` to record the previous vertex on a shortest `path` from the source to each vertex. This array is the parent-link representation of a tree.

With these data structures, implementing edge relaxation is a straightforward task. In our single-source shortest-paths code, we use the following code to relax along an edge e from `v` to `w`:

```
if (wt[w] > wt[v] + e.wt)
 { wt[w] = wt[v] + e.wt; st[w] = v; }
```

This code fragment is both simple and descriptive; we include it in this form in our implementations, rather than defining relaxation as a higher-level abstract operation.

Definition 21.2 *Given a network and a designated vertex s, a* **shortest-paths tree (SPT)** *for s is a subnetwork containing s and all the vertices reachable from s that forms a directed tree rooted at s such that every tree* `path` *is a shortest* `path` *in the network.*

Figure 21.5 Edge Relaxation *These diagrams illustrate the relaxation operation that underlies our single-source shortest-paths algorithms. We keep track of the shortest known* `path` *from the source s to each vertex and ask whether an edge* `v-w` *gives us a shorter* `path` *to* `w`*. In the top example, it does not; so we would ignore it. In the bottom example, it does; so we would update our data structures to indicate that the best known way to get to* `w` *from* `s` *is to go to* `v`*, then take* `v-w`*.*

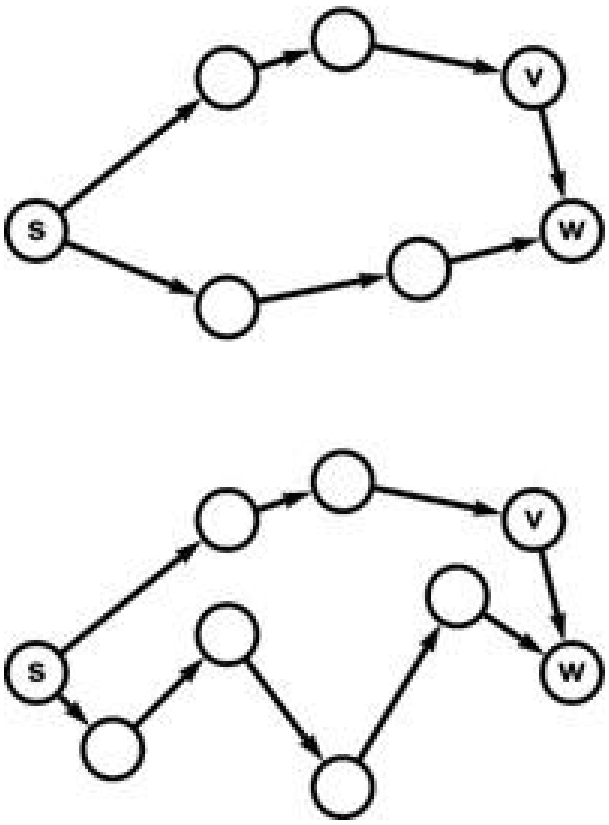
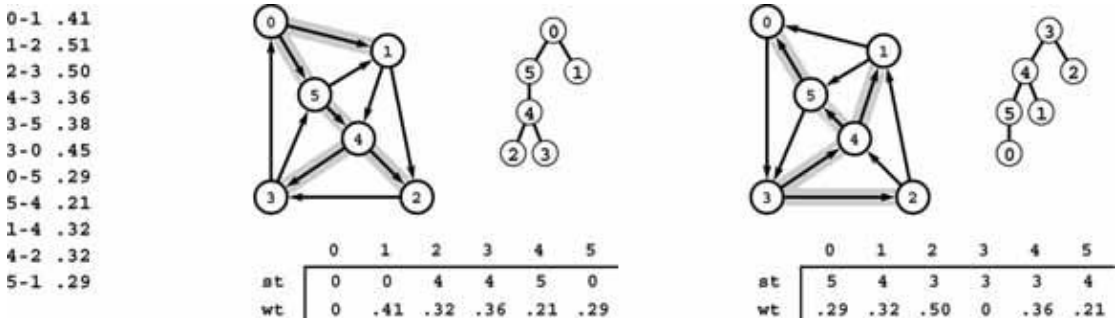


Figure 21.6 Shortest Paths Trees *The shortest paths from* `0` *to the other nodes in this network are* `0-1`, `0-5-4-2`, `0-5-4-3`, `0-5-4`, *and* `0-5`, *respectively. These paths define a spanning tree, which is depicted in three representations (gray edges in the network drawing, oriented tree, and parent links with weights) in the center. Links in the parent-link representation (the one that we typically compute) run in the opposite direction than links in the digraph, so we sometimes work with the reverse digraph. The spanning tree defined by shortest paths from* `3` *to each of the other nodes in the reverse is depicted on the right. The parent-link representation of this tree gives the shortest paths from each of the other nodes to* `2` *in the original graph. For example, we can find the shortest path* `0-5-4-3` *from* `0` *to* `3` *by following the links* `st[0] = 5`, `st[5] = 4`, *and* `st[4] = 3`.



There may be multiple paths of the same length connecting a given pair of nodes, so SPTs are not necessarily unique. In general, as illustrated in *Figure* 21.2, if we take shortest paths from a vertex *s* to every vertex reachable from *s* in a network and from the subnetwork induced by the edges in the paths, we may get a DAG. Different shortest paths con-

IPR2023-00124
KASPERSKY EX 1036 Page 344

necting pairs of nodes, may each appear as a subpath in some longer path containing both nodes. Because of such effects, we generally are content to compute any SPT for a given digraph and start vertex.

Our algorithms generally initialize the entries in the wt array with the sentinel value maxWT. That value needs to be sufficiently small that the addition in the relaxation test does not cause overflow and sufficiently large that no simple path has a larger weight. For example, if edge weights are between 0 and 1, we can use the value V. Note that we have to take extra care to check our assumptions when using sentinels in networks that could have negative weights. For example, if both vertices have the sentinel value, the relaxation code just given takes no action if e.wt is nonnegative (which is probably what we intend in most implementations), but it will change wt[w] and st[w] if the weight is negative.

The st array is a parent-link representation of the shortest-paths tree, with the links pointing in the direction opposite from that of the links in the network, as illustrated in *Figure* 21.6. We can compute the shortest path from s to t by traveling up the tree from t to s, visiting the vertices on the path in reverse order (t, st[t], st[st[t]], and so forth). In some situations, reverse order is precisely what we want. For example, if we are to return a linked-list representation of the path, we can (adhering to our usual conventions for linked lists, where NEW is a function that allocates memory for a node, fills in the node's fields from its arguments, and returns a link to the node) use code like the following:

```
p = NEW (t, null);
while (t != s)
  { t = st[t]; p = NEW (t, p); }
return p;
```

Another option is to use similar code to push the vertices on the path onto a stack—then the client program can visit the vertices on the path in order by popping them from the stack.

On the other hand, if we simply want to print or otherwise process the vertices on the path, reverse order is inconvenient because we have to go all the way through the path in reverse order to get to the first vertex, then go back through the path to process the vertices. One approach to get around this difficulty is to work with the reverse graph, as illustrated in *Figure* 21.6.

Next, we consider path relaxation, which is the basis of some of our all-pairs algorithms: Does going through a given vertex lead us to a shorter path that connects two other given

vertices? For example, suppose that, for three vertices *s*, *x*, and *t*, we wish to know whether it is better to go from *s* to *x* and then from *x* to *t* or to go from *s* to *t* without going through *x*. For straight-line connections in a Euclidean space, the triangle inequality tells us that the route through *x* cannot be shorter than the direct route from *s* to *t*, but for paths in a network, it could be (see *Figure* 21.7). To determine which, we need to know the lengths of paths from *s* to *x*, *x* to *t*, and of those from *s* to *t* (that do not include *x*). Then, we simply test whether or not the sum of the first two is less than the third; if it is, we update our records accordingly.

Path relaxation is appropriate for all-pairs solutions where we maintain the lengths of the shortest paths that we have encountered between all pairs of vertices. Specifically, in all-pairs–shortest-paths code of this kind, we maintain an array `d` such that `d[s][t]` is the shortest-path length from `s` to `t`, and we also maintain an array `p` such that `p[s][t]` is the next vertex on a shortest `path` from `s` to `t`. We refer to the former as the *distances* matrix and the latter as the *paths* matrix. *Figure* 21.8 shows the two matrices for our example network. The distances matrix is a prime objective of the computation, and we use the paths matrix because it is clearly more compact than, but carries the same information as, the full list of paths that is illustrated in *Figure* 21.3.

Figure 21.7 `Path` Relaxation *These diagrams illustrate the relaxation operation that underlies our all-pairs shortest-paths algorithms. We keep track of the best known* `path` *between all pairs of vertices and ask whether a vertex* `i` *is evidence that the shortest known* `path` *from* `s` *to* `t` *could be improved. In the top example, it is not; in the bottom example, it is. Whenever we encounter a vertex* `i` *such that the length of the shortest known* `path` *from* `s` *to* `i` *plus the length of the shortest known* `path` *from* `i` *to* `t` *is smaller than the length of the shortest known* `path` *from* `s` *to* `t`, *then we update our data structures to indicate that we now know a shorter* `path` *from* `s` *to* `t` (*head towards* `i` *first*).

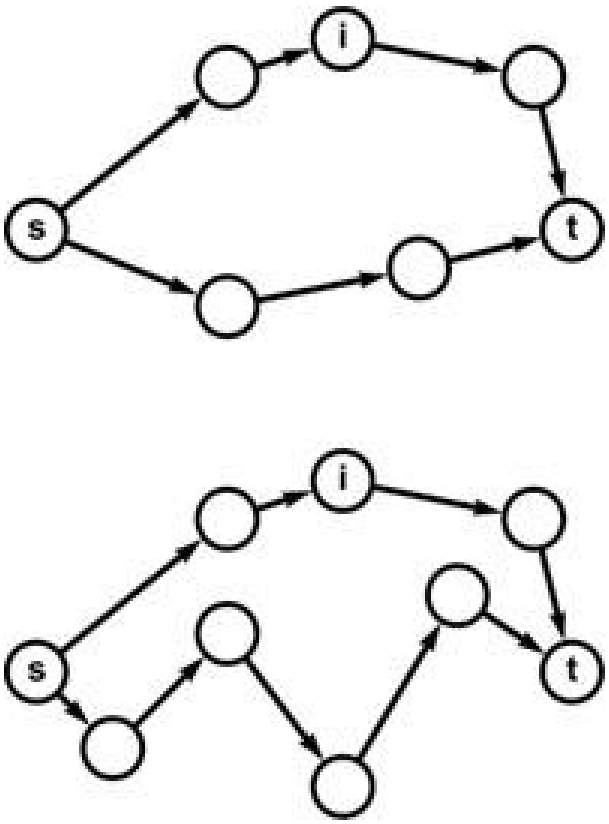
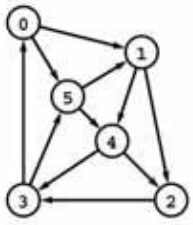


Figure 21.8 All Shortest Paths *The two matrices on the right are compact representations of all the shortest paths in the sample network on the left, containing the same information in the exhaustive list in Figure 21.3. The distances matrix on the left contains the shortest-path length: the entry in row* `s` *and column* `t` *is the length of the shortest* `path` *from* `s` *to* `t`. *The paths matrix on the right contains the information needed to execute the path: the entry in row* `s` *and column* `t` *is the next vertex on the* `path` *from* `s` *to* `t`.

```
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

In terms of these data structures, `path` relaxation amounts to the following code:

```
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]; }
```

Like edge relaxation, this code reads as a restatement of the informal description that we have given, so we use it directly in our implementations. More formally, `path` relaxation reflects the following:

IPR2023-00124
KASPERSKY EX 1036 Page 347

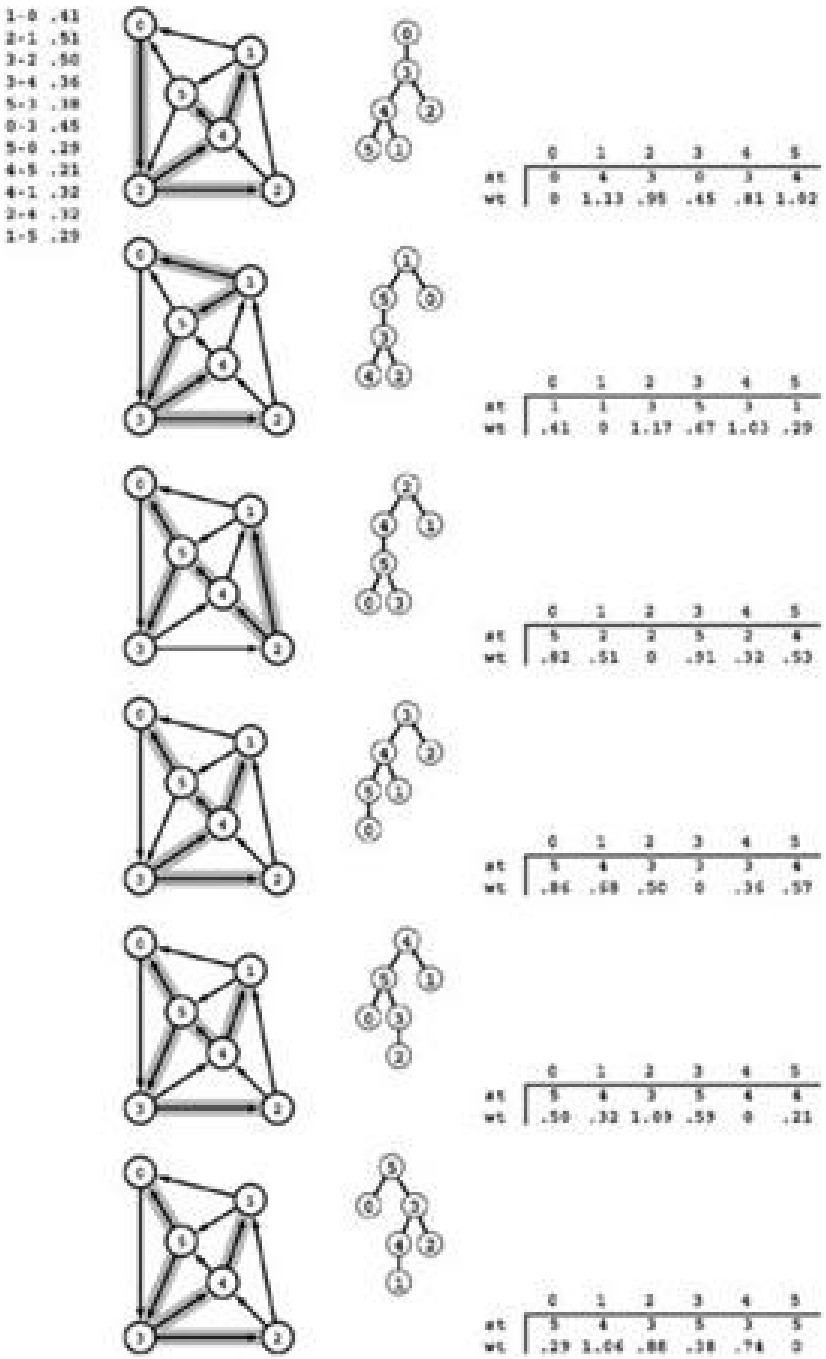
Property 21.1 *If a vertex* `x` *is on a shortest* `path` *from s to t, then that* `path` *consists of a shortest* `path` *from s to x followed by a shortest* `path` *from x to t.*

Proof: By contradiction. We could use any shorter `path` from *s* to *x* or from *x* to *t* to build a shorter `path` from *s* to *t*.■

We encountered the path-relaxation operation when we discussed transitive-closure algorithms, in section 19.3. If the edge and `path` weights are either `1` or infinite (that is, a path's weight is `1` only if all that path's edges have weight `1`), then `path` relaxation is the operation that we used in Warshall's algorithm (if we have a `path` from `s` to `x` and a `path` from `x` to `t`, then we have a `path` from `s` to `t`). If we define a path's weight to be the number of edges on that path, then Warshall's algorithm generalizes to Floyd's algorithm for finding all shortest paths in unweighted digraphs; it further generalizes to apply to networks, as we see in section 21.3.

From a mathematician's perspective, it is important to note that these algorithms all can be cast in a general algebraic setting that unifies and helps us to understand them. From a programmer's perspective, it is important to note that we can implement each of these algorithms using an abstract + operator (to compute `path` weights from edge weights) and an abstract < operator (to compute the minimum value in a set of `path` weights), both solely in the context of the relaxation operation (see Exercise 19.53 and 19.54).

Figure 21.9 All Shortest Paths in a Network *These diagrams depict the SPTs for each vertex in the reverse of the network in Figure* 21.8 (`0` *to* `5`, top to bottom), *as network subtrees* (*left*), *oriented trees* (*center*), *and parent-link representation including a vertex-indexed array for* `path` *length* (*right*). *Putting the arrays together to form* `path` *and distance matrices* (*where each array becomes a column*) *gives the solution to the all-pairs shortest-paths problem illustrated in Figure* 21.8.



Property 21.1 implies that a shortest `path` from *s* to *t* contains shortest paths from *s* to every other vertex along the `path` to *t*. Most shortest-paths algorithms also compute shortest paths from *s* to every vertex that is closer to *s* than to *t* (whether or not the vertex is on the `path` from *s* to *t*), although that is not a requirement (see Exercise 21.16). Solving the source–sink shortest-paths problem with such an algorithm when *t* is the vertex that is farthest from *s* is equivalent to solving the single-source shortest-paths problem for *s*. Conversely, we could use a solution to the single-source shortest-paths problem from *s* as a method for finding the vertex that is farthest from *s*.

The paths array that we use in our implementations for the all-pairs problem is a repre-

IPR2023-00124
KASPERSKY EX 1036 Page 349

sentation of the shortest-paths trees for each of the vertices. We defined `p[s][t]` to be the vertex that follows `s` on a shortest `path` from `s` to `t`. It is thus the same as the vertex that precedes `s` on the shortest `path` from `t` to `s` in the reverse network. In other words, *column* `t` in the paths matrix of a network is a vertex-indexed array that represents the SPT for vertex `t` in its reverse. Conversely, we can build the paths matrix for a network by filling each column with the vertex-indexed array representation of the SPT for the appropriate vertex in the reverse. This correspondence is illustrated in *Figure* 21.9.

We defer to section 21.4 detailed consideration of ADT design, where we see it in the context of solutions to the all-pairs problem. In section 21.2, we consider the single-source problem and use edge relaxation to compute the parent-link representation of the SPT for any given source.

Exercises

▷ **21.11** Draw the SPT from `0` for the network defined in Exercise 21.1 and for its reverse. Give the parent-link representation of both trees.

21.12 Consider the edges in the network defined in Exercise 21.1 to be *undirected* edges, such that each edge corresponds to equal-weight edges in both directions in the network. Answer Exercise 21.11 for this corresponding network.

▷ **21.13** Change the direction of edge `0-2` in *Figure* 21.2. Draw two different SPTs that are rooted at `2` for this modified network.

▷ **21.14** Write a code fragment that, using a parent-link representation of an SPT, prints out each of the paths to the root.

▷ **21.15** Write a code fragment that, using a paths-matrix representation of all shortest paths in a network, prints out all of those paths, in the style of *Figure* 21.3.

21.16 Give an example that shows how we could know that a `path` from `s` to `t` is shortest without knowing the length of a shorter `path` from `s` to `x` for some `x`.

21.2 DIJKSTRA'S ALGORITHM

In Section 20.3, we discussed Prim's algorithm for finding the minimum spanning tree (MST) of a weighted undirected graph: We build it one edge at a time, always taking next the shortest edge that connects a vertex on the MST to a vertex not yet on the MST. We can use a nearly identical scheme to compute an SPT. We begin by putting the source on the SPT; then, we build the SPT one edge at a time, always taking next the edge that gives

a shortest path from the source to a vertex not on the SPT. In other words, we add vertices to the SPT in order of their distance (through the SPT) to the start vertex. This method is known as *Dijkstra's algorithm*.

As usual, we need to make a distinction between the algorithm at the level of abstraction in this informal description and various concrete implementations (such as *Program* 21.1) that differ primarily in graph representation and priority-queue implementations, even though such a distinction is not always made in the literature. We shall consider other implementations and discuss their relationships with *Program* 21.1 after establishing that Dijkstra's algorithm correctly performs the single-source shortest-paths computation.

Property 21.2 *Dijkstra's algorithm solves the single-source shortest-paths problem in networks that have nonnegative weights*.

Proof: Given a source vertex *s*, we have to establish that the tree path from the root *s* to each vertex *x* in the tree computed by Dijkstra's algorithm corresponds to a shortest path in the graph from *s* to *x*. This fact follows by induction. Assuming that the subtree so far computed has the property, we need only to prove that adding a new vertex *x* adds a shortest path to that vertex. But all other paths to *x* must begin with a tree path followed by an edge to a vertex not on the tree. By construction, all such paths are longer than the one from *s* to *x* that is under consideration.

The same argument shows that Dijkstra's algorithm solves the source–sink shortest-paths problem, if we start at the source and stop when the sink comes off the priority queue.■

The proof breaks down if the edge weights could be negative, because it assumes that a path's length does not decrease when we add more edges to the path. In a network with negative edge weights, this assumption is not valid because *any* edge that we encounter might lead to some tree vertex and might have a sufficiently large negative weight to give a path to that vertex shorter than the tree path. We consider this defect in section 21.7 (see *Figure* 21.28).

Figure 21.10 shows the evolution of an SPT for a sample graph when computed with Dijkstra's algorithm; *Figure* 21.11 shows an oriented drawing of a larger SPT tree. Although Dijkstra's algorithm differs from Prim's MST algorithm in only the choice of priority, SPT trees are different in character from MSTs. They are rooted at the start vertex and all edges are directed away from the root, whereas MSTs are unrooted and undirected. We represent MSTs as directed, rooted trees when we use Prim's algorithm, but such structures are still different in character from SPTs (compare the oriented drawing in *Figure*

20.9 with the drawing in *Figure* 21.11). Indeed, the nature of the SPT somewhat depends on the choice of start vertex, as well, as depicted in *Figure* 21.12.

Dijkstra's original implementation, which is suitable for dense graphs, is precisely like Prim's MST algorithm. Specifically, we simply change the definition of P in *Program* 20.3 from

```
#define P G->adj[v][w]
```

(the edge weight) to

```
#define P wt[v] + G->adj[v][w]
```

(the distance from the source to the edge's destination). This change gives the classical implementation of Dijkstra's algorithm: we grow an SPT one edge at a time, each time checking all the nontree vertices to find an edge to move to the tree whose destination vertex is a nontree vertex of minimal distance from the source.

Figure 21.10 Dijkstra's Algorithm *This sequence depicts the construction of a shortest-paths spanning tree rooted at vertex* `0` *by Dijkstra's algorithm for a sample network. Thick black edges in the network diagrams are tree edges, and thick gray edges are fringe edges. Oriented drawings of the tree as it grows are shown in the center, and a list of fringe edges is given on the right. The first step is to add* `0` *to the tree and the edges leaving it,* `0-1` *and* `0-5`*, to the fringe (top). Second, we move the shortest of those edges,* `0-5`*, from the fringe to the tree and check the edges leaving it: the edge* `5-4` *is added to the fringe and the edge* `5-1` *is discarded because it is not part of a shorter* `path` *from* `0` *to* `1` *than the known path* `0-1` *(second from top). The priority of* `5-4` *on the fringe is the length of the* `path` *from* `0` *that it represents,* `0-5-4`*. Third, we move* `0-1` *from the fringe to the tree, add* `1-2` *to the fringe, and discard* `1-4` *(third from top). Fourth, we move* `5-4` *from the fringe to the tree, add* `4-3` *to the fringe, and replace* `1-2` *with* `4-2` *because* `0-5-4-2` *is a shorter* `path` *than* `0-1-2` *(fourth from top). We keep at most one edge to any vertex on the fringe, choosing the one on the shortest* `path` *from* `0`*. We complete the computation by moving* `4-2` *and then* `4-3` *from the fringe to the tree (bottom).*

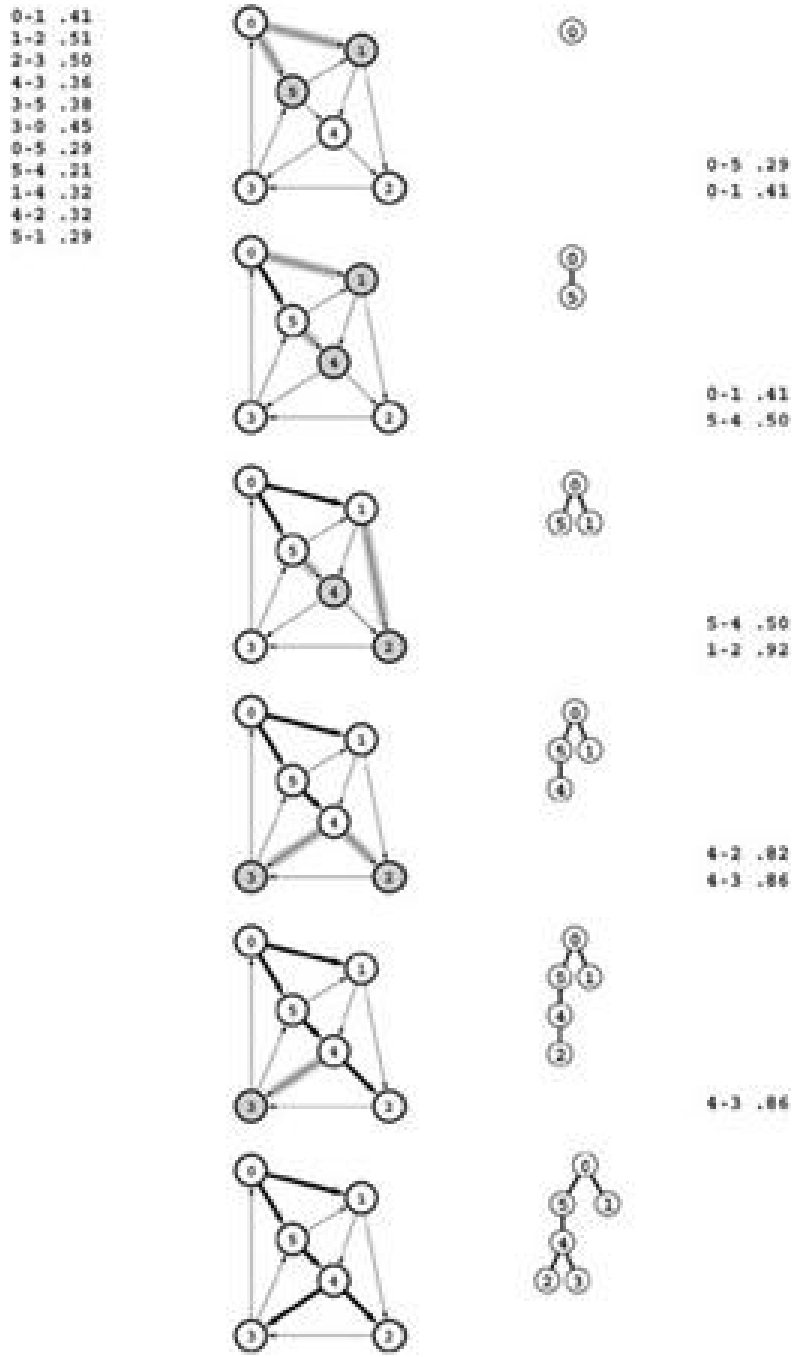
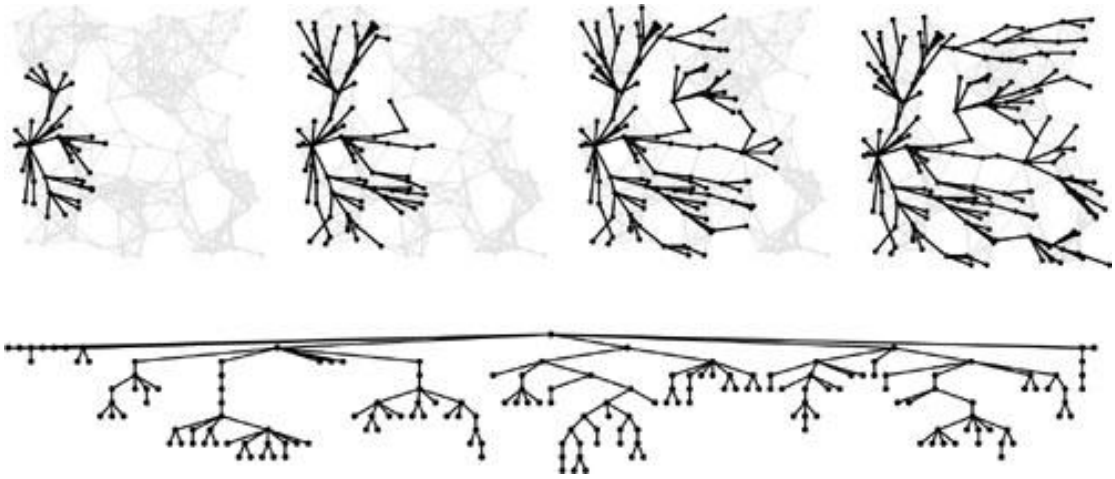


Figure 21.11 Shortest-Paths Spanning Tree *This figure illustrates the progress of Dijkstra's algorithm in solving the single-source shortest-paths problem in a random Euclidean near-neighbor digraph (with directed edges in both directions corresponding to each line drawn), in the same style as Figures 18.13, 18.24, and 20.9. The search tree is similar in character to BFS because vertices tend to be connected to one another by short paths, but it is slightly deeper and less broad because distances lead to slightly longer paths than* `path` *lengths.*

IPR2023-00124
KASPERSKY EX 1036 Page 353



Property 21.3 *With Dijkstra's algorithm, we can find any SPT in a dense network in linear time.*

Proof: As for Prim's MST algorithm, it is immediately clear, from inspection of the code of *Program* 20.3, that the running time is proportional to V^2, which is linear for dense graphs.■

For sparse graphs, we can do better, by viewing Dijkstra's algorithm as a generalized graph-searching method that differs from depth-first search (DFS), from breadth-first search (BFS), and from Prim's MST algorithm in only the rule used to add edges to the tree. As in Chapter 20, we keep edges that connect tree vertices to nontree vertices on a generalized queue called the *fringe*, use a priority queue to implement the generalized queue, and provide for updating priorities so as to encompass DFS, BFS, and Prim's algorithm in a single implementation (see Section 20.3). This priority-first search (PFS) scheme also encompasses Dijkstra's algorithm. That is, changing the definition of P in *Program* 20.4 to

```
#define P wt[v] + t->wt
```

(the distance from the source to the edge's destination) gives an implementation of Dijkstra's algorithm that is suitable for sparse graphs.

Program 21.1 is an alternative PFS implementation for sparse graphs that is slightly simpler than *Program* 20.4 and that directly matches the informal description of Dijkstra's algorithm given at the beginning of this section. It differs from *Program* 20.4 in that it initializes the priority queue with all the vertices in the network and maintains the queue with the aid of a sentinel value for those vertices that are neither on the tree nor on the fringe (unseen vertices with sentinel values); in contrast, *Program* 20.4 keeps on the priority queue only those vertices that are reachable by a single edge from the tree. Keeping

all the vertices on the queue simplifies the code but can incur a small performance penalty for some graphs (see Exercise 21.31).

Program 21.1 Dijkstra's algorithm (adjacency lists)

This implementation of Dijkstra's algorithm uses a priority queue of vertices (in order of their distance from the source) to compute an SPT. We initialize the queue with priority `0` for the source and priority `maxWT` for the other vertices, then enter a loop where we move a lowest-priority vertex from the queue to the SPT and relax along its incident edges.

The indirect priority-queue interface code is the same as in *Program* 20.4 and is omitted. It defines a static variable `priority` and a function `less`, which allow the priority-queue functions to manipulate vertex names (indices) and to use `less` to compare the priorities that are maintained by this code in the `wt` array.

This code is a generalized graph search, a PFS implementation. The definition of `P` implements Dijkstra's algorithm; other definitions implement other PFS algorithms (*see text*).

```
#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; }
}
```

The general results that we considered concerning the performance of priority-first search (PFS) in Chapter 20 give us specific information about the performance of these implementations of Dijkstra's algorithm for sparse graphs (*Program* 21.1 and *Program* 20.4, suitably modified). For reference, we restate those results in the present context. Since the proofs do not depend on the priority function, they apply without modification. They are worst-case results that apply to both programs, although *Program* 20.4 may be more efficient for many classes of graphs because it maintains a smaller fringe.

Property 21.4 *For all networks and all priority functions, we can compute a spanning tree with PFS in time proportional to the time required for* ***V insert, V*** **delete the minimum**, *and E* **decrease key** *operations in a priority queue of size at most V.*

Figure 21.12 SPT Examples *These three examples show growing SPTs for three different source locations: left edge* (*top*), *upper left corner* (*center*), *and center* (*bottom*).

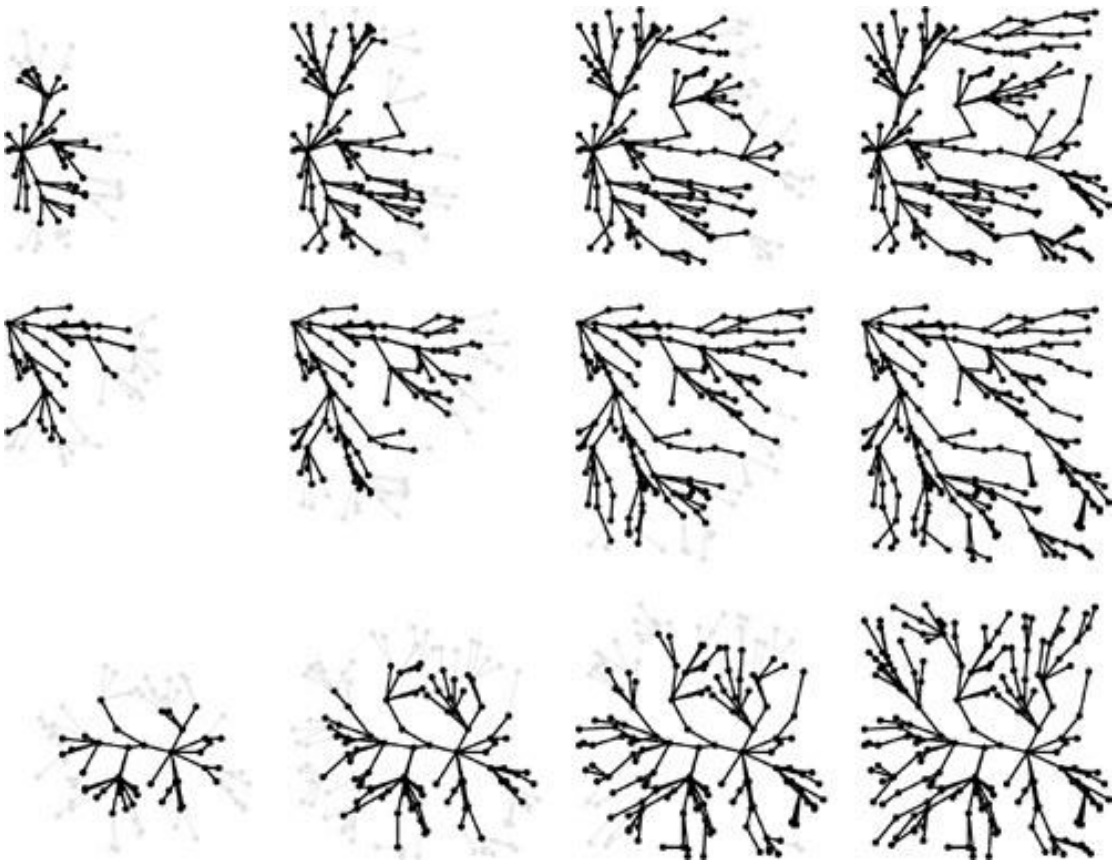


Table 21.1 Priority-First Search Algorithms These four classical graph-processing algorithms all can be implemented with PFS, a generalized priority-queue–based graph search that builds graph spanning trees one edge at a time. Details of search dynamics depend upon graph representation, priority-queue implementation, and PFS implementation; but the search trees generally characterize the various algorithms, as illustrated in the figures referenced in the fourth column.

algorithm	priority	result	Figure
DFS	reverse preorder	recursion tree	18.13
BFS	preorder	SPT (edges)	18.24
Prim	edge weight	MST	20.8
Dijkstra	path weight	SPT	21.9

Proof: This fact is immediate from the priority-queue–based implementations in *Program* 20.4 or *Program* 21.1. It represents a conservative upper bound because the size of the priority queue is often much smaller than V, particularly for *Program* 20.4■

Property 21.5 *With a PFS implementation of Dijkstra's algorithm that uses a heap for the priority-queue implementation, we can compute any SPT in time proportional to E* lg *V*.

Proof: This result is a direct consequence of Property 21.4.■

Property 21.6 *Given a graph with V vertices and E edges, let d denote the density E/V. If d < 2, then the running time of Dijkstra's algorithm is proportional to V* lg *V. Otherwise, we can use a [E/V]-ary heap for the priority queue to improve the worst-case running time by a factor of* lg (*E/V*), *to O* (*E* $\lg_d V$), *which is linear if E is at least* $V^{1+\epsilon}$.

Proof: This result directly mirrors Property 20.12 and the multiway-heap priority-queue implementation discussed directly thereafter.■

Table 21.1 summarizes pertinent information about the four major PFS algorithms that we have considered. They differ in only the priority function used, but this difference leads to spanning trees that are entirely different from one another in character (as required). For the example in the figures referred to in the table (and for many other graphs), the DFS tree is tall and thin, the BFS tree is short and fat, the SPT is like the BFS tree but neither quite as short nor quite as fat, and the MST is neither short and fat nor tall and thin.

We have also considered four different implementations of PFS. The first is the classical dense-graph implementation that encompasses Dijkstra's algorithm and Prim's MST algorithm (*Program* 20.3); the other three are sparse-graph implementations that differ in priority-queue contents:

- Fringe edges (*Program* 18.10)
- Fringe vertices (*Program* 20.4)

● All vertices (*Program* 21.1)

Of these, the first is primarily of pedagogical value; the second is the most refined of the three; the third is perhaps the simplest. This framework already describes 16 different implementations of classical graph-search algorithms—when we factor in different priority-queue implementations, the possibilities multiply further. This proliferation of networks, algorithms, and implementations underscores the utility of the general statements about performance in Properties 21.4 through 21.6, which are also summarized in Table 21.2.

As is true of MST algorithms, actual running times of shortest-paths algorithms are likely to be lower than these worst-case time bounds suggest, primarily because most edges do not necessitate *decrease key* operations. In practice, except for the sparsest of graphs, we regard the running time as being linear.

The name *Dijkstra's algorithm* is commonly used to refer both to the abstract method of building an SPT by adding edges to vertices in order of their distance from the source and to its implementation as the V^2 algorithm for the adjacency-matrix representation, because Dijkstra presented both in his 1959 paper (and also showed that the same approach could compute the MST). Performance improvements for sparse graphs are dependent on later improvements in ADT technology and priority-queue implementations that are not specific to the shortest-paths problem. Improved performance of Dijkstra's algorithm is one of the most important applications of that technology (*see reference section*). As we do for MSTs, we use terminology such as the "PFS implementation of Dijkstra's algorithm using *d*-heaps" to identify specific combinations.

Table 21.2 Cost of implementations of Dijkstra's algorithm This table summarizes the cost (worst-case running time) of various implementations of Dijkstra's algorithm. With appropriate priority-queue implementations, the algorithm runs in linear time (time proportional to V^2 for dense networks, E for sparse networks) except for networks that are extremely sparse.

algorithm	worst-case cost	comment
classical	V^2	optimal for dense graphs
PFS, full heap	$E \lg V$	simplest ADT code
PFS, fringe heap	$E \lg V$	conservative upper bound
PFS, d-heap	$E \lg_d V$	linear unless extremely sparse

We saw in section 18.8 that, in unweighted undirected graphs, using preorder numbering for priorities causes the priority queue to operate as a FIFO queue and leads to a BFS.

Dijkstra's algorithm gives us another realization of BFS: When all edge weights are 1, it visits vertices in order of the number of edges on the shortest `path` to the start vertex. The priority queue does not operate precisely as a FIFO queue would in this case, because items with equal priority do not necessarily come out in the order in which they went in.

Each of these implementations builds a parent-link representation of the SPT from vertex `0` in the vertex-indexed argument array `st`, with the shortest-path length to each vertex in the SPT in the vertex-indexed argument array `wt`. As usual, we can build various convenient ADT functions around this general scheme (see Exercise 21.19 through 21.28).

Exercises

▷ **21.17** Show, in the style of *Figure* 21.10, the result of using Dijkstra's algorithm to compute the SPT of the network defined in Exercise 21.1 with start vertex `0`.

○ **21.18** How would you find a *second* shortest `path` from *s* to *t* in a network?

▷ **21.19** Add a function to a standard network ADT that uses `GRAPHspt` to compute the length of a shortest `path` connecting two given vertices `s` and `t`.

21.20 Add a function to a standard network ADT that uses `GRAPHspt` to find the most distant vertex from a given vertex `s` (the vertex whose shortest `path` from `s` is the longest).

21.21 Add a function to a standard network ADT that uses `GRAPHspt` to compute the average of the lengths of the shortest paths from a given vertex to each of the vertices reachable from it.

○ **21.22** Add a function to a standard adjacency-lists network ADT that solves the source–sink shortest-paths problem by using `GRAPHspt` to compute a linked-list representation of a shortest `path` connecting two given vertices `s` and `t`.

○ **21.23** Add a function to a standard network ADT that solves the source–sink shortest-paths problem by using `GRAPHspt` to fill the initial entries in a given argument array with the successive vertex indices on a shortest `path` connecting two given vertices `s` and `t`.

21.24 Develop an interface and implementation based on *Program* 21.1 to push a shortest `path` connecting two given vertices `s` and `t` onto a user-supplied stack.

21.25 Add a function to a standard adjacency-lists network ADT that finds all vertices within a given distance *d* of a given vertex in a given network. The running time of your

function should be proportional to the size of the subgraph induced by those vertices and the vertices incident on them.

21.26 Develop an algorithm for finding an edge whose removal causes maximal increase in the shortest-path length from one given vertex to another given vertex in a given network.

● **21.27** Add a function to a standard adjacency-matrix network ADT that performs a *sensitivity analysis* on the network's edges with respect to a given pair of vertices `s` and `t`: Compute a `V`-by-`V` array such that, for every `u` and `v`, the entry in row `u` and column `v` is `1` if `u-v` is an edge in the network whose weight can be increased without the shortest-path length from `s` to `t` being increased and is `0` otherwise.

○ **21.28** Add a function to a standard adjacency-lists network ADT that finds a shortest `path` connecting one given set of vertices with another given set of vertices in a given network.

21.29 Use your solution from Exercise 21.28 to implement a function that finds a shortest `path` from the left edge to the right edge in a random grid network (see Exercise 20.7).

21.30 Show that an MST of an undirected graph is equivalent to a *bottleneck SPT* of the graph: For every pair of vertices v and w, it gives the `path` connecting them whose longest edge is as short as possible.

21.31 Run empirical studies to compare the performance of the two versions of Dijkstra's algorithm for the sparse graphs that are described in this section (*Program* 21.1 and *Program* 20.4, with suitable priority definition), for various networks (see Exercise 21.4–8). Use a standard-heap priority-queue implementation.

21.32 Run empirical studies to learn the best value of d when using a d-heap priority-queue implementation (see *Program* 20.7) for each of the three PFS implementations that we have discussed (*Program* 18.10, *Program* 20.4 and *Program* 21.1), for various networks (see Exercise 21.4–8).

● **21.33** Run empirical studies to determine the effect of using an index-heap-tournament priority-queue implementation (see Exercise 9.53) in *Program* 21.1, for various networks (see Exercise 21.4–8).

IPR2023-00124
KASPERSKY EX 1036 Page 360

○ **21.34** Run empirical studies to analyze height and average `path` length in SPTs, for various networks (see Exercise 21.4– 8).

21.35 Develop an implementation for the source–sink shortest-paths problem that is based on initializing the priority queue with *both* the source and the sink. Doing so leads to the growth of an SPT from each vertex; your main task is to decide precisely what to do when the two SPTs collide.

● **21.36** Describe a family of graphs with V vertices and E edges for which the worst-case running time of Dijkstra's algorithm is achieved.

●● **21.37** Develop a reasonable generator for random graphs with V vertices and E edges for which the running time of the heap-based PFS implementation of Dijkstra's algorithm is superlinear.

● **21.38** Write a client program that does dynamic graphical animations of Dijkstra's algorithm. Your program should produce images like *Figure* 21.11 (see Exercises 17.55 through 17.59). Test your program on random Euclidean networks (see Exercise 21.8).

21.3 ALL-PAIRS SHORTEST PATHS

In this section, we consider an ADT and two basic implementations for the all-pairs shortest-paths problem. The algorithms that we implement directly generalize two basic algorithms that we considered in section 19.3 for the transitive-closure problem. The first method is to run Dijkstra's algorithm from each vertex to get the shortest paths from that vertex to each of the others. If we implement the priority queue with a heap, the worst-case running time for this approach is proportional to $VE \lg V$, and we can improve this bound to VE for many types of networks by using a d-ary heap. The second method, which allows us to solve the problem directly in time proportional to V^3, is an extension of Warshall's algorithm that is known as *Floyd's algorithm*.

Program 21.2 All-pairs shortest-paths ADT

Our solutions to the all-pairs shortest-paths problem provide clients with a preprocessing function `GRAPHspALL` and two query functions: one that returns the length of the shortest `path` from the first argument to the second (`GRAPHspDIST`); and an-

IPR2023-00124
KASPERSKY EX 1036 Page 361

other that returns the next vertex on the shortest `path` from the first argument to the second (`GRAPHspPATH`). By convention, if there is no such path, `GRAPHspPATH` returns `G->V` and `GRAPHspDIST` returns a sentinel value greater than the length of the longest path.

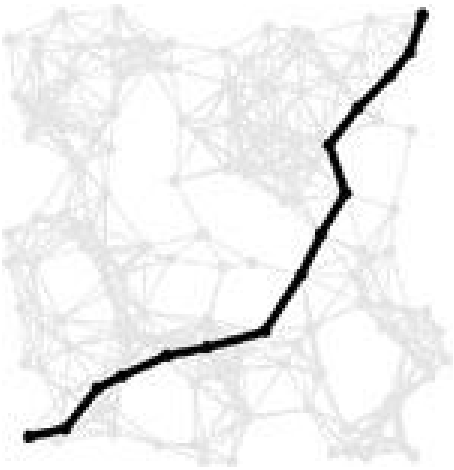
```
      void GRAPHspALL (Graph G);
double GRAPHspDIST (Graph G, int s, int t);
   int GRAPHspPATH (Graph G, int s, int t);
```

We can use either of the algorithms to implement a preprocessing function in support of an *abstract shortest-paths* function in a network ADT that can return the shortest-path length between any two vertices in constant time, just as we built ADTs to handle connectivity queries based on computing the transitive closure, in Chapter 19. *Program* 21.2 is an interface that describes three functions that we can add to the standard network ADT to support finding shortest distances and paths in this manner. The first is a preprocessing function, the second is a query function that returns the shortest-path length from one given vertex to another, and the third is a query function that returns the next vertex on the shortest `path` from one given vertex to another. Support for such an ADT is a primary reason to use all-pairs shortest-paths algorithms in practice.

Program 21.3 is a sample client program that uses the shortest-path ADT functions to find the *weighted diameter* of a network. It calls the preprocessing function, then checks all pairs of vertices to find the one for which the shortest-path length is longest; then, it traverses the path, vertex by vertex. *Figure* 21.13 shows the `path` computed by this program for our Euclidean network example.

Figure 21.13 Diameter of a Network *The largest entry in a network's all-shortest-paths matrix is the diameter of the network: the length of the longest of the shortest paths, depicted here for our sample Euclidean network.*

IPR2023-00124
KASPERSKY EX 1036 Page 362



The goal of the algorithms in this section is to support constant-time implementations of the query functions. Typically, we expect to have a huge number of such requests, so we are willing to invest substantial resources in memory and preprocessing to be able to satisfy the requests quickly.

Accordingly, our implementations solve the all-pairs shortest-paths problem in the preprocessing function, then simply access the solutions in the query functions. We assume that the standard network ADT is augmented to include pointers to two matrices: a V-by-V array `G->dist` for the distances matrix, and another V-by-V array `G->path` for the paths table. For every pair of vertices `s` and `t`, the preprocessing functions of both of the algorithms that we consider set `G->dist[s][t]` to the shortest-path length from `s` to `t` and set `G->path[s][t]` to the index of the next vertex on the shortest `path` from `s` to `t`; the other two ADT functions just return these values, in constant time.

The primary disadvantage of this general approach is that, for a huge network, we may not be able to afford the time for preprocessing or have space available for the tables. In principle, our interface provides us with the latitude to trade off preprocessing time and space for query time. If we expect only a few queries, we can do no preprocessing and simply run a single-source algorithm for each query, but intermediate situations require more advanced algorithms (see Exercise 21.48 through 21.50). This problem generalizes one that challenged us for much of Chapter 19: the problem of supporting constant-time reachability queries in limited space.

The first all-pairs shortest-paths ADT function implementation that we consider solves the problem by using Dijkstra's algorithm to solve the single-source problem for each vertex. For each vertex `s`, we fill row `s` in the distances matrix with the `wt` array for the solution to the single-source shortest-paths problem for `s`. This method generalizes the BFS-based method for unweighted undirected graphs that we considered in Section 17.7. It is also similar to our use of a DFS that starts at each vertex to compute the transitive

closure of unweighted digraphs, in *Program* 19.4.

As illustrated in Figures 21.8 and 21.9, computing the paths matrix is slightly trickier, because the parent-link SPT representation in the array `st` gives us the edges in the wrong direction. This difficulty does not arise for undirected graphs, because all edges run in both directions. To address this problem, we work with the reverse network and fill *column* `t` in the paths table with the parent-link representation of the SPT (the `st` array) for `t` in the reverse network. Since the lengths of shortest paths are the same in both directions, we can also fill column `t` in the distances matrix with the `wt` array to give the solution to the single-source shortest-paths problem for `t`.

Program 21.3 Computing the diameter of a network

This client program illustrates the use of the interface in *Program* 21.2. It finds the longest of the shortest paths in the given network, prints its weight (the diameter of the network), and prints the path.

```
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);
  }
```

Program 21.4 is an ADT implementation based on these ideas. It can be used with either the adjacency-matrix or the adjacency-lists representation, but it is intended for use with sparse graphs, because it takes advantage of the efficiency of Dijkstra's algorithm for such

graphs.

Property 21.7 *With Dijkstra's algorithm, we can find all shortest paths in a network that has nonnegative weights in time proportional to* $V E \log_d V$*, where* $d = 2$ *if* $E < 2V$*, and* $d = E/V$ *otherwise.*

Proof: Immediate from Property 21.6.■

Program 21.4 Dijkstra's algorithm for all shortest paths

This implementation of the interface in *Program* 21.2 uses Dijkstra's algorithm (see, for example, *Program* 21.1) on the reverse network to find all shortest paths to each vertex (*see text*).

The network data type is assumed to have a pointer dist for the distances array and a pointer `path` for the paths array. The `wt` and `st` arrays computed by Dijkstra's algorithm are columns in the distances and paths matrices, respectively (see *Figure* 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]; }
```

As are our bounds for the single-source shortest-paths and the MST problems, this bound is conservative; and a running time of VE is likely for typical graphs.

For dense graphs, we could use an adjacency-matrix representation and avoid computing the reverse graph by implicitly transposing the matrix (interchanging the row and column indices), as in *Program* 19.8. Developing an implementation along these lines is an interesting programming exercise and leads to a compact implementation (see Exercise 21.43); however, a different approach, which we consider next, admits an even more compact implementation.

Program 21.5 Floyd's algorithm for all shortest paths

This code for `GRAPHspALL` (along with the one-line implementations of `GRAPHspDIST` and `GRAPHspPATH` from *Program* 21.4) implements the interface in *Program* 21.2 using Floyd's algorithm, a generalization of Warshall's algorithm (see *Program* 19.4) that finds the shortest paths instead of just testing for the existence of paths.

After initializing the distances and paths matrices with the graph's edges, we do a series of relaxation operations to compute the shortest paths. The algorithm is simple to implement, but verifying that it computes the shortest paths is more complicated (*see text*).

```
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;
      }
```

The method of choice for solving the all-pairs shortest-paths problem in dense graphs, which was developed by R. Floyd, is precisely the same as Warshall's method, except that, instead of using the logical *or* operation to keep track of the existence of paths, it checks distances for each edge to determine whether that edge is part of a new shorter path. Indeed, as we have noted, Floyd's and Warshall's algorithms are identical in the proper abstract setting (see Sections 19.3 and 21.1). *Program* 21.5 is an all-pairs shortest-paths ADT function that implements Floyd's algorithm.

Property 21.8 *With Floyd's algorithm, we can find all shortest paths in a network in time proportional to* V^3.

Proof: The running time is immediate from inspection of the code. We prove that the algorithm is correct by induction in precisely the same way as we did for Warshall's algorithm. The `ith` iteration of the loop computes a shortest `path` from `s` to `t` in the network that does not include any vertices with indices greater than `i` (except possibly the endpoints `s` and `t`). Assuming this fact to be true for the `ith` iteration of the loop, we prove it to be true for the `(i+1)st` iteration of the loop. A shortest `path` from `s` to `t` that does not include any vertices with indices greater than `i+1` is either (*i*) a `path` from `s` to `t` that does not include any vertices with indices greater than `i`, of length `d[s][t]`, that was found on a previous iteration of the loop, by the inductive hypothesis; or (*ii*) comprising a `path` from `s` to `i` and a `path` from `i` to `t`, neither of which includes any vertices with indices greater than `i`, in which case the inner loop sets `d[s][t]`.■

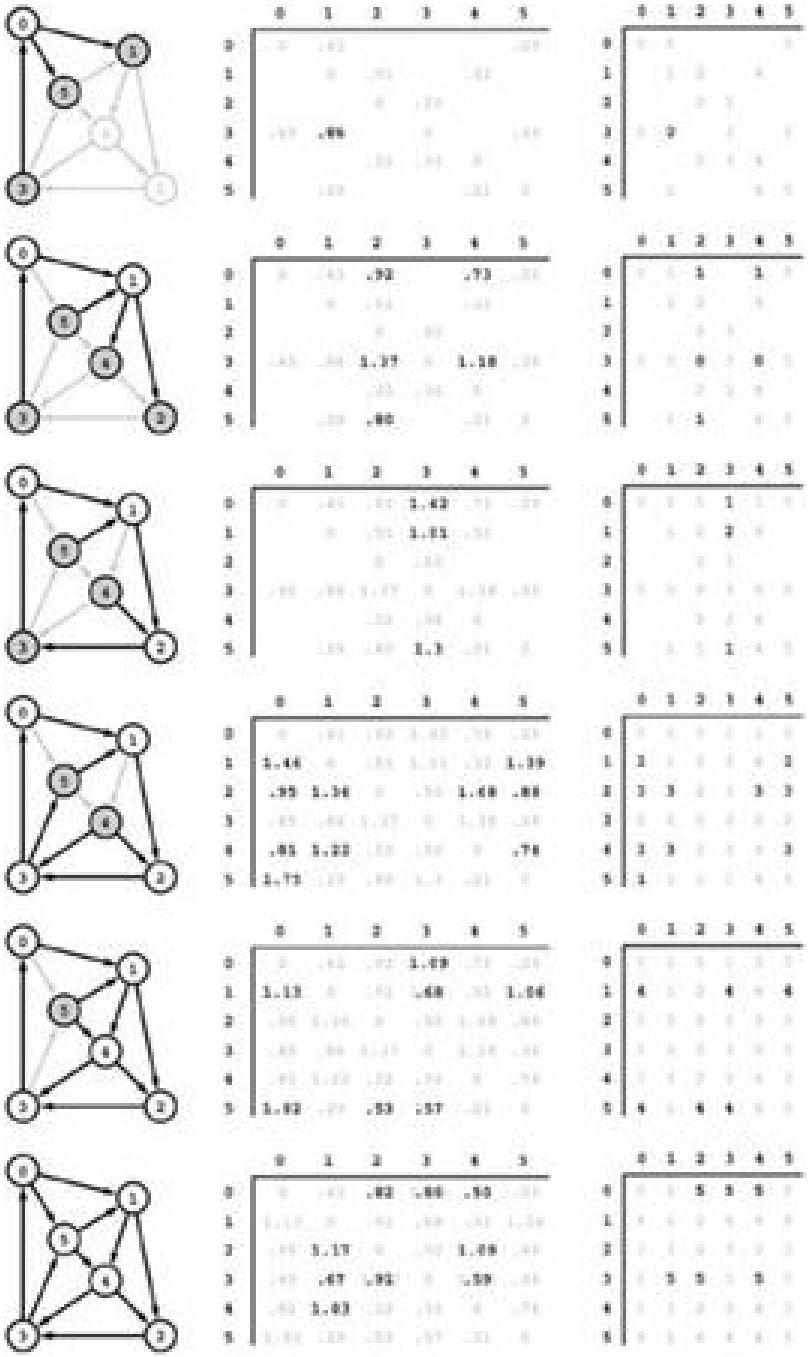
Figure 21.14 is a detailed trace of Floyd's algorithm on our sample network. If we convert each blank entry to `0` (to indicate the absence of an edge) and convert each nonblank entry to `1` (to indicate the presence of an edge), then these matrices describe the operation of Warshall's algorithm in precisely the same manner as we did in *Figure* 19.15. For Floyd's algorithm, the nonblank entries indicate more than the existence of a path; they give information about the shortest known path. An entry in the distance matrix has the length of the shortest known `path` connecting the vertices corresponding to the given row and column; the corresponding entry in the paths matrix gives the next vertex on that path. As the matrices become filled with nonblank entries, running Warshall's algorithm amounts to just doublechecking that new paths connect pairs of vertices already known to

IPR2023-00124
KASPERSKY EX 1036 Page 367

be connected by a path; in contrast, Floyd's algorithm must compare (and update if necessary) each new `path` to see whether the new `path` leads to shorter paths.

Comparing the worst-case bounds on the running times of Dijkstra's and Floyd's algorithms, we can draw the same conclusion for these all-pairs shortest-paths algorithms as we did for the corresponding transitive-closure algorithms in section 19.3. Running Dijkstra's algorithm on each vertex is clearly the method of choice for sparse networks, because the running time is close to VE. As density increases, Floyd's algorithm—which always takes time proportional to V^3—becomes competitive (see Exercise 21.67); it is widely used because it is so simple to implement.

Figure 21.14 Floyd's Algorithm *This sequence shows the construction of the all-pairs shortest-paths matrices with Floyd's algorithm. For* `i` *from* `0` *to* `5` (top to bottom), *we consider, for all* `s` *and* `t`, *all of the paths from* `s` *to* `t` *having no intermediate vertices greater than* `i` (*the shaded vertices*). *Initially, the only such paths are the network's edges, so the distances matrix* (*center*) *is the graph's adjacency matrix and the paths matrix* (right) *is set with* `p[s][t] = t` *for each edge* `s-t`. *For vertex* `0` (top), *the algorithm finds that* `3-0-1` *is shorter than the sentinel value that is present because there is no edge* `3-1` *and updates the matrices accordingly. It does not do so for paths such as* `3-0-5`, *which is not shorter than the known path* `3-5`. *Next the algorithm considers paths through* `0` *and* `1` (*second from top*), *and finds the new shorter paths* `0-1-2, 0-1-4, 3-0-1-2, 3-0-1-4`, *and* `5-1-2`. *The third row from the top shows the updates corresponding to shorter paths through* `0, 1`, *and* `2`; *and so forth. Black numbers overstriking gray ones in the matrices indicate situations where the algorithm finds a shorter* `path` *than one it found earlier. For example,* `.91` *overstrikes* `1.37` *in row* `3` *and column* `2` *in the bottom diagram because the algorithm discovered that* `3-5-4-2` *is shorter than* `3-0-1-2`.



A more fundamental distinction between the algorithms, which we examine in detail in section 21.7, is that Floyd's algorithm is effective in even those networks that have negative weights (provided that there are no negative cycles). As we noted in section 21.2, Dijkstra's method does not necessarily find shortest paths in such graphs.

The classical solutions to the all-pairs shortest-paths problem that we have described presume that we have space available to hold the distances and paths matrices. Huge sparse graphs, where we cannot afford to have any V-by-V arrays, present another set of challenging and interesting problems. As we saw in Chapter 19, it is an open problem to reduce this space cost to be proportional to *V* while still supporting constant-time shortest-

path-length queries. We found the analogous problem to be difficult even for the simpler reachability problem (where we are satisfied with learning in constant time whether there is *any* `path` connecting a given pair of vertices), so we cannot expect a simple solution for the all-pairs shortest-paths problem. Indeed, the number of different shortest `path` lengths is, in general, proportional to V^2 even for sparse graphs. That value, in some sense, measures the amount of information that we need to process, and perhaps indicates that, when we do have restrictions on space, we must expect to spend more time on each query (see Exercise 21.48 through 21.50).

Exercises

▷ **21.39** Estimate, to within a factor of 10, the largest graph (measured by its number of vertices) that your computer and programming system could handle if you were to use Floyd's algorithm to compute all its shortest paths in 10 seconds.

▷ **21.40** Estimate, to within a factor of 10, the largest graph of density 10 (measured by its number of edges) that your computer and programming system could handle if you were to use Dijkstra's algorithm to compute all its shortest paths in 10 seconds.

21.41 Show, in the style of *Figure* 21.9, the result of using Dijkstra's algorithm to compute all shortest paths of the network defined in Exercise 21.1.

21.42 Show, in the style of *Figure* 21.14, the result of using Floyd's algorithm to compute all shortest paths of the network defined in Exercise 21.1.

○ **21.43** Combine *Program* 20.3 and *Program* 21.4 to make an implementation of the all-pairs shortest-paths ADT interface (based on Dijkstra's algorithm) for dense networks that does not require explicit computation of the reverse network. Do not define a separate function for `GRAPHpfs`—put the code from *Program* 20.3 directly in the inner loop, eliminate the argument arrays `wt` and `st`, and put results directly in `G->dist` and `G->path` (or use local arrays `d` and `p` as in *Program* 21.5).

21.44 Run empirical tests, in the style of Table 20.2, to compare Dijkstra's algorithm (*Program* 21.4 and Exercise 21.43) and Floyd's algorithm (*Program* 21.5) for various networks (see Exercise 21.4–8).

21.45 Run empirical tests to determine the number of times that Floyd's and Dijkstra's algorithms update the values in the distances matrix, for various networks (see Exercise 21.4–8).

IPR2023-00124
KASPERSKY EX 1036 Page 370

21.46 Give a matrix in which the entry in row *s* and column *t* is equal to the number of different simple directed paths connecting *s* and *t* in *Figure* 21.1.

21.47 Implement a network ADT function that can compute the path-count matrix that is described in Exercise 21.46.

21.48 Develop an implementation of the abstract shortest-paths ADT for sparse graphs that cuts the space cost to be proportional to V, by increasing the query time to be proportional to V.

● **21.49** Develop an implementation of the abstract shortest-paths ADT for sparse graphs that uses substantially less than $O(V^2)$ space but supports queries in substantially less than $O(V)$ time. *Hint*: Compute all shortest paths for a subset of the vertices.

● **21.50** Develop an implementation of the abstract shortest-paths ADT for sparse graphs that uses substantially less than $O(V^2)$ space and (using randomization) supports queries in constant *expected* time.

○ **21.51** Develop an implementation of the shortest-paths ADT that takes the *lazy* approach of using Dijkstra's algorithm to build the SPT (and associated distance vector) for each vertex `s` the first time that the client issues a shortest-path query from `s`, then references the information on subsequent queries.

21.52 Modify the shortest-paths ADT and Dijkstra's algorithm to handle shortest-paths computations in networks that have weights on both vertices and edges. Do not rebuild the graph representation (the method described in Exercise 21.3); modify the code instead.

● **21.53** Build a small model of airline routes and connection times, perhaps based upon some flights that you have taken. Use your solution to Exercise 21.52 to compute the fastest way to get from one of the served destinations to another. Then test your program on real data (see Exercise 21.4).

21.4 SHORTEST PATHS IN ACYCLIC NETWORKS

In Chapter 19, we found that, despite our intuition that DAGs should be easier to process than general digraphs, developing algorithms with substantially better performance for DAGs than for general digraphs is an elusive goal. For shortest-paths problems, we do have algorithms for DAGs that are simpler and faster than the priority-queue–based methods that we have considered for general digraphs. Specifically, in this section we

IPR2023-00124
KASPERSKY EX 1036 Page 371

consider algorithms for acyclic networks that

- Solve the single-source problem in linear time.
- Solve the all-pairs problem in time proportional to VE.
- Solve other problems, such as finding longest paths.

In the first two cases, we cut the logarithmic factor from the running time that is present in our best algorithms for sparse networks; in the third case, we have simple algorithms for problems that are intractable for general networks. These algorithms are all straightforward extensions to the algorithms for reachability and transitive closure in DAGs that we considered in Chapter 19.

Since there are no cycles at all, there are no negative cycles; so negative weights present no difficulty in shortest-paths problems on DAGs. Accordingly, we place no restrictions on edge-weight values throughout this section.

Next, a note about terminology: We might choose to refer to directed graphs with weights on the edges and no cycles either as *weighted DAGs* or as *acyclic networks*. We use both terms interchangeably to emphasize their equivalence and to avoid confusion when we refer to the literature, where both are widely used. It is sometimes convenient to use the former to emphasize differences from unweighted DAGs that are implied by weights and the latter to emphasize differences from general networks that are implied by acyclicity.

The four basic ideas that we applied to derive efficient algorithms for unweighted DAGs in Chapter 19 are even more effective for weighted DAGs:

- Use DFS to solve the single-source problem.
- Use a source queue to solve the single-source problem.
- Invoke either method, once for each vertex, to solve the all-pairs problem.
- Use a single DFS (with dynamic programming) to solve the all-pairs problem.

These methods solve the single-source problem in time proportional to E and the all-pairs problem in time proportional to VE. They are all effective because of topological ordering, which allows us compute shortest paths for each vertex without having to revisit any decisions. We consider one implementation for each problem in this section; we leave the others for exercises (see Exercise 21.62 through 21.65).

We begin with a slight twist. Every DAG has at least one source but could have several, so

IPR2023-00124
KASPERSKY EX 1036 Page 372

it is natural to consider the following shortest-paths problem:

Multisource shortest paths Given a set of start vertices, find, for each other vertex w, a shortest `path` among the shortest paths from each start vertex to w.

This problem is essentially equivalent to the single-source shortest-paths problem. We can convert a multisource problem into a single-source problem by adding a dummy source vertex with zero-length edges to each source in the network. Conversely, we can convert a single-source problem to a multisource problem by working with the induced subnetwork defined by all the vertices and edges reachable from the source. We rarely construct such subnetworks explicitly, because our algorithms automatically process them if we treat the start vertex as though it were the only source in the network (even when it is not).

Topological sorting immediately presents a solution to the multi-source shortest-paths problem and to numerous other problems. We maintain a vertex-indexed array `wt` that gives the weight of the shortest known `path` from any source to each vertex. To solve the multisource shortest-paths problem, we initialize the `wt` array to `0` for sources and `MAXwt` for all the other vertices. Then, we process the vertices in topological order. To process a vertex `v`, we perform a relaxation operation for each outgoing edge `v-w` that updates the shortest `path` to `w` if `v-w` gives a shorter `path` from a source to `w` (through `v`). This process checks all paths from any source to each vertex in the graph; the relaxation operation keeps track of the minimum-length such path, and the topological sort ensures that we process the vertices in an appropriate order.

We can implement this method directly in one of two ways. The first is to add a few lines of code to the topological sort code in *Program* 19.8: just after we remove a vertex `v` from the source queue, we perform the indicated relaxation operation for each of its edges (see Exercise 21.56). The second is to put the vertices in topological order, then to scan through them and to perform the relaxation operations precisely as described in the previous paragraph. These same processes (with other relaxation operations) can solve many graph-processing problems. For example, *Program* 21.6 is an implementation of the second approach (sort, then scan) for solving the *multisource longest-paths* problem: For each vertex in the network, what is a longest `path` from some source to that vertex?

We interpret the `wt` entry associated with each vertex to be the length of the *longest* known `path` from any source to that vertex, initialize all of the weights to `0`, and change the sense of the comparison in the relaxation operation. *Figure* 21.15 traces the operation of *Program* 21.6 on a sample acyclic network.

Property 21.9 *We can solve the multisource shortest-paths problem and the multi-*

IPR2023-00124
KASPERSKY EX 1036 Page 373

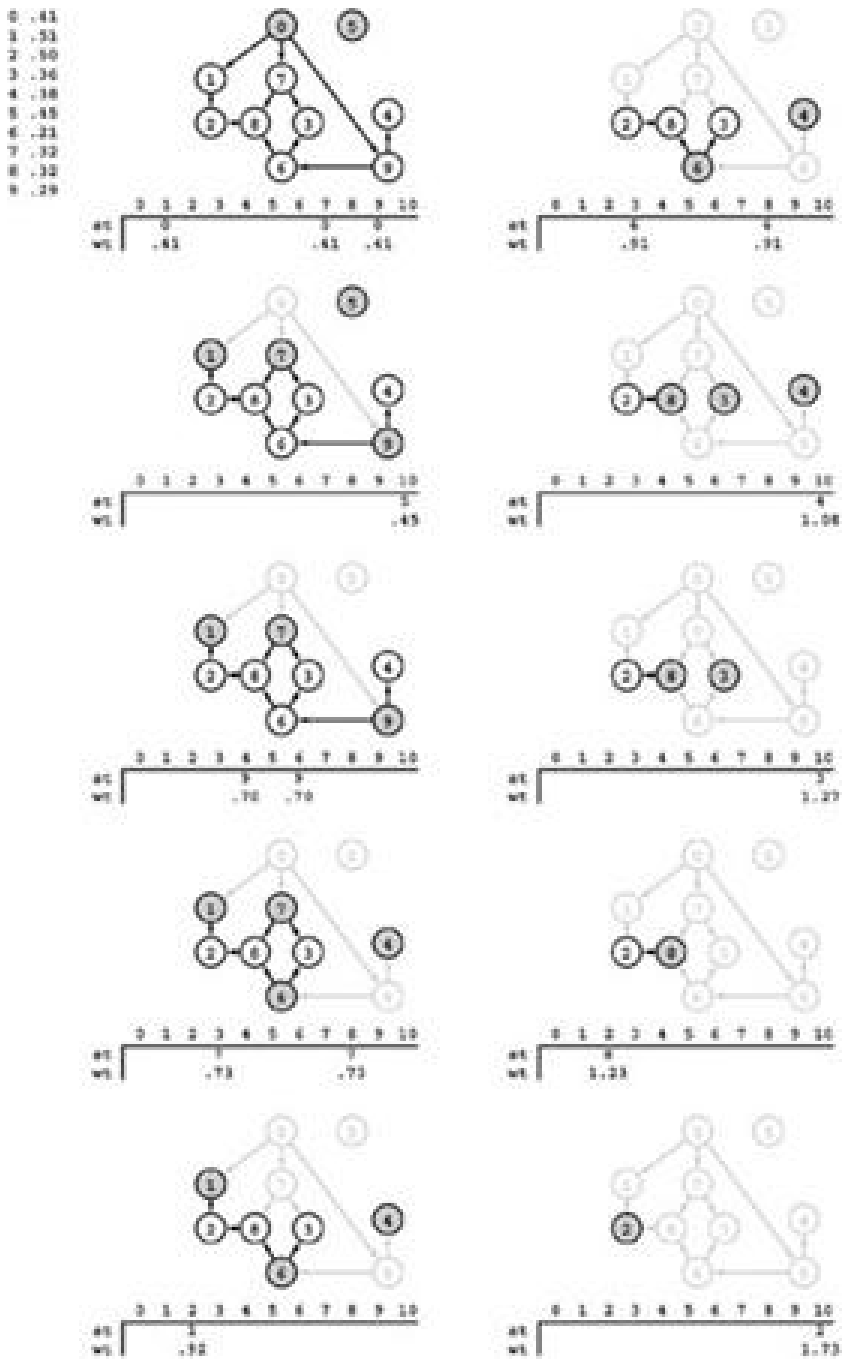
source longest-paths problem in acyclic networks in linear time.

Proof : The same proof holds for longest path, shortest path, and many other `path` properties. To match *Program* 21.6, we state the proof for longest paths. We show by induction on the loop variable `i` that, for all vertices `v = ts[j]` with `j < i` that have been processed, `wt[v]` is the length of the longest `path` from a source to `v`. When `v = ts[i]`, let `t` be the vertex preceding `v` on any `path` from a source to `v`. Since vertices in the ts array are in reverse topologically sorted order, `t` must have been processed already. By the induction hypothesis, `wt[t]` is the length of the longest `path` to `t`, and the relaxation step in the code checks whether that `path` gives a longer `path` to `v` through `t`. The induction hypothesis also implies that all paths to `v` are checked in this way as `v` is processed.■

This property is significant because it tells us that processing acyclic networks is considerably easier than processing networks that have cycles. For shortest paths, the source-queue method is faster than Dijkstra's algorithm by a factor proportional to the cost of the priority-queue operations in Dijkstra's algorithm. For longest paths, we have a linear algorithm for acyclic networks but an intractable problem for general networks. Moreover, negative weights present no special difficulty here, but they present formidable barriers for algorithms on general networks, as discussed in section 21.7.

Figure 21.15 Computing Longest Paths in an Acyclic Network *In this network, each edge has the weight associated with the vertex that it leads from, listed at the top left. Sinks have edges to a dummy vertex* `10`*, which is not shown in the drawings. The* `wt` *array contains the length of the longest known* `path` *to each vertex from some source, and the* `st` *array contains the previous vertex on the longest path. This figure illustrates the operation of Program* 21.6*, which picks from among the sources (the shaded nodes in each diagram) using the FIFO discipline, though any of the sources could be chosen at each step. We begin by removing* `0` *and checking each of its incident edges, discovering one-edge paths of length* `.41` *to* `1`*,* `7`*, and* `9`*. Next, we remove* `5` *and record the one-edge* `path` *from* `5` *to* `10` *(left, second from top). Next, we remove* `9` *and record the paths* `0-9-4` *and* `0-9-6`*, of length* `.70` *(left, third from top). We continue in this way, changing the arrays whenever we find longer paths. For example, when we remove* `7` *(left, second from bottom) we record paths of length* `.73` *to* `8` *and* `3`*; then, later, when we remove* `6`*, we record longer paths (of length* `.91`*) to* `8` *and* `3` *(right, top). The point of the computation is to find the longest* `path` *to the dummy node* `10`*. In this case, the result is the path* `0-9-6-8-2`*, of length* `1.73`*.*

IPR2023-00124
KASPERSKY EX 1036 Page 374



Program 21.6 Longest paths in an acyclic network

To find the *longest* paths in an acyclic network, we consider the vertices in topological order, keeping the weight of the longest known `path` to each vertex in a vertex-indexed array `wt` by doing a relaxation step for each edge. This program also computes a spanning forest of longest paths from a source to each vertex, in a vertex-in-

dexed array `st`.

```
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; }
  }
```

The method just described depends on only the fact that we process the vertices in topological order. Therefore, any topological-sorting algorithm can be adapted to solve shortest- and longest-paths problems and other problems of this type (see, for example, Exercise 21.56 and 21.62).

As we know from Chapter 19, the DAG abstraction is a general one that arises in many applications. For example, we see an application in section 21.6 that seems unrelated to networks but that can be addressed directly with *Program* 21.6.

Next, we turn to the all-pairs shortest-paths problem for acyclic networks. As in section 19.3, one method that we could use to solve this problem is to run a single-source algorithm for each vertex (see Exercise 21.65). The equally effective approach that we consider here is to use a single DFS with dynamic programming, just as we did for computing the transitive closure of DAGs in section 19.5 (see *Program* 19.9). When we consider the vertices in reverse topological order, we can derive the shortest-path vector for each vertex from the shortest-path vectors for each adjacent vertex, simply by using each edge in a relaxation step.

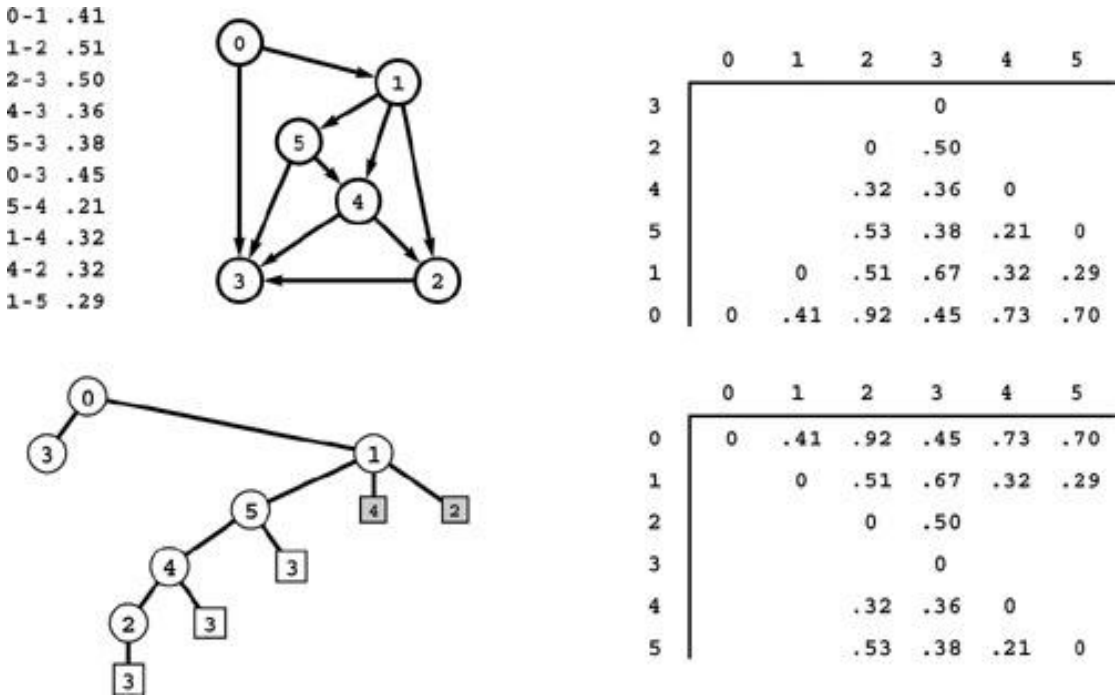
Program 21.7 All shortest paths in an acyclic network

This implementation of GRAPHspALL for weighted DAGs is derived by adding appropriate relaxation operations to the dynamic-programming– based transitive-closure function in *Program* 19.10.

IPR2023-00124
KASPERSKY EX 1036 Page 376

Program 21.7 is an implementation along these lines. The operation of this program on a sample weighted DAG is illustrated in *Figure* 21.16. Beyond the generalization to include relaxation, there is one important difference between this computation and the transitive-closure computation for DAGs: In *Program* 19.9, we had the choice of ignoring down edges in the DFS tree because they provide no new information about reachability; in *Program* 21.7, however, we need to consider all edges, because any edge might lead to a shorter path.

Figure 21.16 Shortest Paths in an Acyclic Network *This diagram depicts the computation of the all-shortest-distances matrix (bottom right) for a sample weighted DAG (top left), computing each row as the last action in a recursive DFS function. Each row is computed from the rows for adjacent vertices, which appear earlier in the list, because the rows are computed in reverse topological order (postorder traversal of the DFS tree shown at the bottom left). The array on the top right shows the rows of the matrix in the order that they are computed. For example, to compute each entry in the row for* `0` *we add* `.41` *to the corresponding entry in the row for* `1` *(to get the distance to it from* `0` *after taking* `0-1`*), then add* `.45` *to the corresponding entry in the row for* `3` *(to get the distance to it from* `0` *after taking* `0-3`*), and take the smaller of the two. The computation is essentially the same as computing the transitive closure of a DAG (see, for example, Figure 19.23). The most signifi-cant difference between the two is that the transitive closure algorithm could ignore down edges (such as* `1-2` *in this example) because they go to vertices known to be reachable, while the shortest-paths algorithm has to check whether paths associated with down edges are shorter than known paths. If we were to ignore* `1-2` *in this example, we would miss the shortest paths* `0-1-2` *and* `1-2`*.*



Property 21.10 *We can solve the all-pairs shortest-paths problem in acyclic networks*

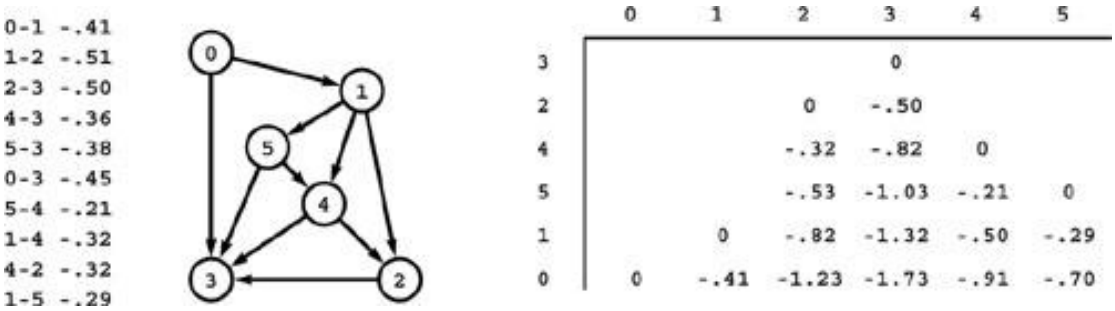
with a single DFS in time proportional to V E.

Proof : This fact follows immediately from the strategy of solving the single-source problem for each vertex (see Exercise 21.65). We can also establish it by induction, from *Program* 21.7. After the recursive calls for a vertex v, we know that we have computed all shortest paths for each vertex on v's adjacency list, so we can find shortest paths from v to each vertex by checking each of v's edges. We do *V* relaxation steps for each edge, for a total of *V E* relaxation steps■

Thus, for acyclic networks, reverse topological ordering allows us to avoid the cost of the priority queue in Dijkstra's algorithm. Like Floyd's algorithm, *Program* 21.7 also solves problems more general than those solved by Dijkstra's algorithm, because, unlike Dijkstra's (see section 21.7), this algorithm works correctly even in the presence of negative edge weights. If we run the algorithm after negating all the weights in an acyclic network, it finds all *longest* paths, as depicted in *Figure* 21.17. Or, we can find longest paths by reversing the inequality test in the relaxation algorithm, as in *Program* 21.6.

The other algorithms for finding shortest paths in acyclic networks that are mentioned at the beginning of this section generalize the methods from Chapter 19 in a manner similar to the other algorithms that we have examined in this chapter. Developing implementations of them is a worthwhile way to cement your understanding of both DAGs and shortest paths (see Exercise 21.62 through 21.65). All the methods run in time proportional to *V E* in the worst case, with actual costs dependent on the structure of the DAG. In principle, we might do even better for certain sparse weighted DAGs (see Exercise 19.119).

Figure 21.17 All Longest Paths in an Acyclic Network *Our method for computing all shortest paths in acyclic networks works even if the weights are negative. Therefore, we can use it to compute longest paths, simply by first negating all the weights, as illustrated here for the network in Figure 21.16. The longest simple* `path` *in this network is* `0-1-5-4-2-3`*, of weight* `1.73`.



Exercises

▷ **21.54** Give the solutions to the multisource shortest- and longest-paths problems for

the network defined in Exercise 21.1, with the directions of edges `2-3` and `1-0` reversed.

▷ **21.55** Modify *Program* 21.6 such that it solves the multisource shortest-paths problem for acyclic networks.

▷ **21.56** Give an implementation of `GRAPHlpt` that is derived from the source-queue–based topological-sorting code of *Program* 19.8, performing the relaxation operations for each vertex just after that vertex is removed from the source queue.

○ **21.57** Define an ADT for the relaxation operation, provide implementations, and modify *Program* 21.6 to use your ADT, such that you can use *Program* 21.6 to solve the multisource shortest-paths problem, the multisource longest-paths problem, and other problems, just by changing the relaxation implementation.

21.58 Use your generic implementation from Exercise 21.57 to implement ADT operations that return the length of the longest paths from any source to any other vertex in a DAG, the length of the shortest such path, and the number of vertices reachable via paths whose lengths fall within a given range.

● **21.59** Define properties of relaxation such that you can modify the proof of Property 21.9 to apply an abstract version of *Program* 21.6 (such as the one described in Exercise 21.57).

▷ **21.60** Show, in the style of *Figure* 21.16, the computation of the all-pairs shortest-paths matrices for the network defined in Exercise 21.54 by *Program* 21.7.

○ **21.61** Give an upper bound on the number of edge weights accessed by *Program* 21.7, as a function of basic structural properties of the network. Write a program to compute this function, and use it to estimate the accuracy of the $V E$ bound, for various acyclic networks (add weights as appropriate to the models in Chapter 19).

○ **21.62** Write a DFS-based solution to the multisource shortest-paths problem for acyclic networks. Does your solution work correctly in the presence of negative edge weights? Explain your answer.

▷ **21.63** Extend your solution to Exercise 21.62 to provide an implementation of the all-pairs shortest-paths ADT interface for acyclic networks that builds the all-paths and all-distances arrays in time proportional to $V E$.

21.64 Show, in the style of *Figure* 21.9, the computation of all shortest paths of the network defined in Exercise 21.54 using the DFS-based method of Exercise 21.63.

IPR2023-00124
KASPERSKY EX 1036 Page 379

▷ **21.65** Modify *Program* 21.6 such that it solves the single-source shortest-paths problem in acyclic networks, then use it to develop an implementation of the all-pairs shortest-paths ADT interface for acyclic networks that builds the all-paths and all-distances arrays in time proportional to VE.

○ **21.66** Work Exercise 21.61 for the DFS-based (Exercise 21.63) and for the topological-sort–based (Exercise 21.65) implementations of the all-pairs shortest-paths ADT. What inferences can you draw about the comparative costs of the three methods?

21.67 Run empirical tests, in the style of Table 20.2, to compare the three programs for the all-pairs shortest-paths problem described in this section (see *Program* 21.7, Exercise 21.63, and Exercise 21.65), for various acyclic networks (add weights as appropriate to the models in Chapter 19).

21.5 EUCLIDEAN NETWORKS

In applications where networks model maps, our primary interest is often in finding the best route from one place to another. In this section, we examine a strategy for this problem: a fast algorithm for the source–sink shortest-path problem in *Euclidean networks*, which are networks whose vertices are points in the plane and whose edge weights are defined by the geometric distances between the points.

These networks satisfy two important properties that do not necessarily hold for general edge weights. First, the distances satisfy the triangle inequality: The distance from s to d is never greater than the distance from s to x plus the distance from x to d. Second, vertex positions give a lower bound on `path` length: No `path` from s to d will be shorter than the distance from s to d. The algorithm for the source–sink shortest-paths problem that we examine in this section takes advantage of these two properties to improve performance.

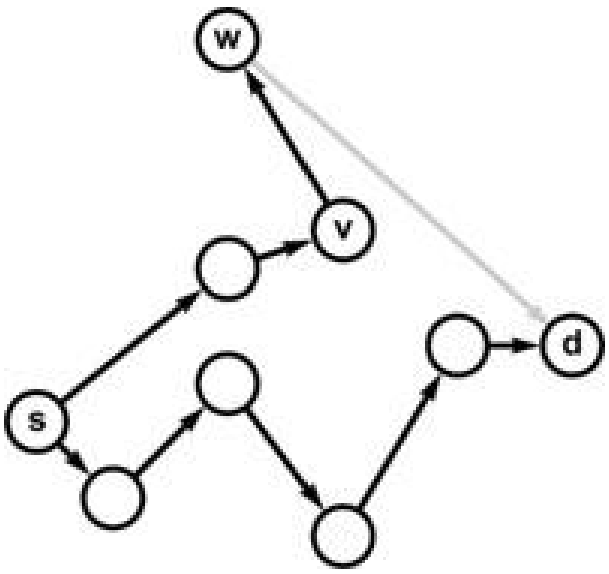
Often, Euclidean networks are also *symmetric*: Edges run in both directions. As mentioned at the beginning of the chapter, such networks arise immediately if, for example, we interpret the adjacency-matrix or adjacency-lists representation of an undirected weighted Euclidean graph (see Section 20.7) as a weighted digraph (network). When we draw an undirected Euclidean network, we assume this interpretation, to avoid proliferation of arrowheads in the drawings.

The basic idea is straightforward: Priority-first search provides us with a general mechanism to search for paths in graphs. With Dijkstra's algorithm, we examine paths in order of their distance from the start vertex. This ordering ensures that, when we reach the sink, we have examined all paths in the graph that are shorter, none of which took us to

IPR2023-00124
KASPERSKY EX 1036 Page 380

the sink. But in a Euclidean graph, we have additional information: If we are looking for a `path` from a source `s` to a sink `d` and we encounter a third vertex `v`, then we know that not only do we have to take the `path` that we have found from `s` to `v`, but also the best that we could possibly do in traveling from `v` to `d` is first to take an edge `v-w` and then to find a `path` whose length is the straight-line distance from `w` to `d` (see *Figure* 21.18). With priority-first search, we can easily take into account this extra information to improve performance. We use the standard algorithm, but we use the sum of the following three quantities as the priority of each edge `v-w`: the length of the known `path` from `s` to `v`, the weight of the edge `v-w`, and the distance from `w` to `t`. If we always pick the edge for which this number is smallest, then, when we reach `t`, we are still assured that there is no shorter `path` in the graph from `s` to `t`. Furthermore, in typical networks, we reach this conclusion after doing far less work than we would were we using Dijkstra's algorithm.

Figure 21.18 Edge Relaxation (Euclidean) *In a Euclidean graph, we can take distances to the destination into account in the relaxation operation as we compute shortest paths. In this example, we could deduce that the* `path` *depicted from* `s` *to* `v` *plus* `v-w` *cannot lead to a shorter* `path` *from* `s` *to* `d` *than the one already found because the length of any such* `path` *must be at least the length of the* `path` *from* `s` *to* `v` *plus the length of* `v-w` *plus the straight-line distance from* `w` *to* `d`*, which is greater than the length of the known* `path` *from* `s` *to* `d`*. Tests like this one can significantly reduce the number of paths that we have to consider.*



To implement this approach, we use a standard PFS implementation of Dijkstra's algorithm (*Program* 21.1, since Euclidean graphs are normally sparse, but also see Exercise 21.73) with two changes: First, instead of initializing `wt[s]` at the beginning of the search to `0.0`, we set it to the quantity `dist (s, d)`, where dist is a function that returns the distance between two vertices. Second, we define the priority `P` to be the function

```
      (wt[v] + t->wt + dist (t->v, d) - dist (v, d))
```

instead of the function (`wt[v] + t->w`) that we used in *Program* 21.1. These changes, to which we refer as the *Euclidean heuristic*, maintain the invariant that the quantity `wt[v] - dist (v, d)` is the length of the shortest `path` through the network from `s` to `v`, for every tree vertex `v` (and therefore `wt[v]` is a lower bound on the length of the shortest possible `path` through `v` from `s` to `d`). We compute `wt[t->v]` by adding to this quantity the edge weight (the distance to `t->v`) plus the distance from `t->v` to the sink `d`.

Property 21.11 *Priority-first search with the Euclidean heuristic solves the source–sink shortest-paths problem in Euclidean graphs*.

Proof: The proof of Property 21.2 applies: At the time that we add a vertex `x` to the tree, the addition of the distance from `x` to `d` to the priority does not affect the reasoning that the tree `path` from `s` to `x` is a shortest `path` in the graph from `s` to `x`, since the same quantity is added to the length of all paths to `x`. When `d` is added to the tree, we know that no other `path` from `s` to `d` is shorter than the tree path, because any such `path` must consist of a tree `path` followed by an edge to some vertex `w` that is not on the tree, followed by a `path` from `w` to `d` (whose length cannot be shorter than the distance from `w` to `d`); and, by construction, we know that the length of the `path` from `s` to `w` plus the distance from `w` to `d` is no smaller than the length of the tree `path` from `s` to `d`. ■

In section 21.6, we discuss another simple way to implement the Euclidean heuristic. First, we make a pass through the graph to change the weight of each edge: For each edge `v-w`, we add the quantity `dist (w, d) - dist (v, d)`. Then, we run a standard shortest-path algorithm, starting at `s` (with `wt[s]` initialized to `dist (s, d)`) and stopping when we reach `d`. This method is computationally equivalent to the method that we have described (which essentially computes the same weights on the fly) and is a specific example of a basic operation known as *reweighting* a network. Reweighting plays an essential role in solving the shortest-paths problems with negative weights; we discuss it in detail in section 21.6.

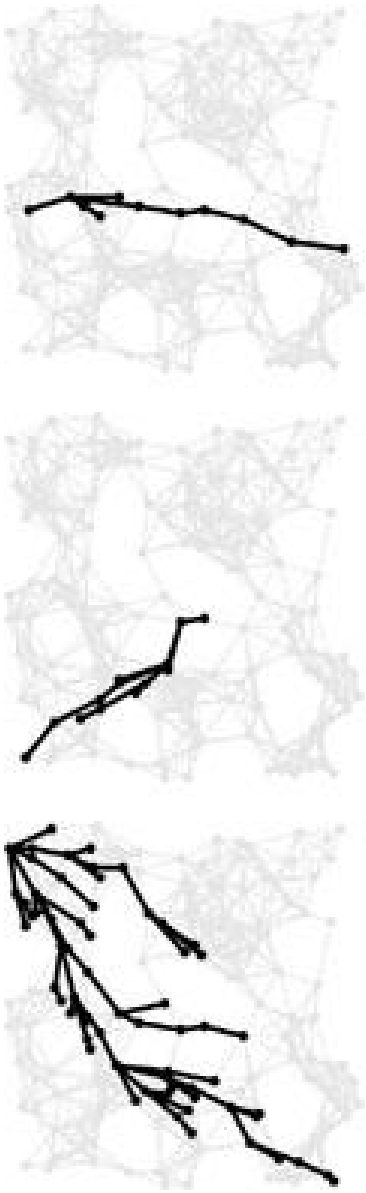
The Euclidean heuristic affects the performance, but not the correctness, of Dijkstra's algorithm for the source–sink shortest-paths computation. As discussed in the proof of Property 21.2, using the standard algorithm to solve the source–sink problem amounts to building an SPT that has all vertices closer to the start than the sink `d`. With the Euclidean heuristic, the SPT contains just the vertices whose `path` from `s` *plus* distance to `d` is smaller than the length of the shortest `path` from `s` to `d`. We expect this tree to be substantially smaller for many applications because the heuristic prunes a substantial num-

ber of long paths. The precise savings is dependent on the structure of the graph and the geometry of the vertices. *Figure* 21.19 shows the operation of the Euclidean heuristic on our sample graph, where the savings are substantial. We refer to the method as a heuristic because there is no guarantee that there will be any savings at all: It could always be the case that the only `path` from source to sink is a long one that wanders arbitrarily far from the source before heading back to the sink (see Exercise 21.80).

Figure 21.20 illustrates the basic underlying geometry that describes the intuition behind the Euclidean heuristic: If the shortest-path length from `s` to `d` is z, then vertices examined by the algorithm fall roughly within the ellipse defined as the locus of points `x` for which the distance from `s` to `x` plus the distance from `x` to `d` is equal to z. For typical Euclidean graphs, we expect the number of vertices in this ellipse to be far smaller than the number of vertices in the circle of radius z that is centered at the source (those that would be examined by Dijkstra's algorithm).

Precise analysis of the savings is a difficult analytic problem and depends on models of both random point sets and random graphs (*see reference section*). For typical situations, we expect that, if the standard algorithm examines X vertices in computing a source–sink shortest path, the Euclidean heuristic will cut the cost to be proportional to $\sqrt{X}$, which leads to an expected running time proportional to V for dense graphs and proportional to $\sqrt{V}$ for sparse graphs. This example illustrates that the difficulty of developing an appropriate model or analyzing associated algorithms should not dissuade us from taking advantage of the substantial savings that are available in many applications, particularly when the implementation (add a term to the priority) is trivial.

Figure 21.19 Shortest `Path` in a Euclidean Graph *When we direct the shortest-path search towards the destination vertex, we can restrict the search to vertices within a relatively small ellipse around the path, as illustrated in these three examples, which show SPT subtrees from the examples in Figure 21.12.*

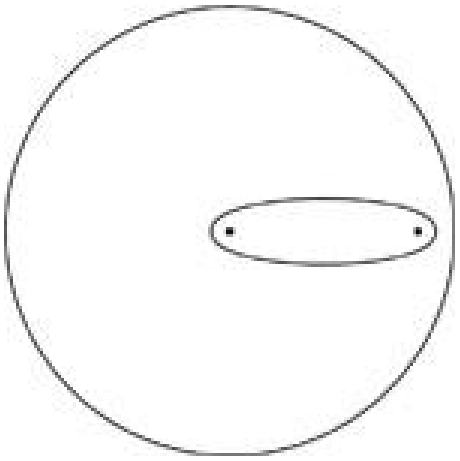


The proof of Property 21.11 applies for *any* function that gives a lower bound on the distance from each vertex to d. Might there be other functions that will cause the algorithm to examine even fewer vertices than the Euclidean heuristic? This question has been studied in a general setting that applies to a broad class of combinatorial search algorithms. Indeed, the Euclidean heuristic is a specific instance of an algorithm called *A* * (pronounced “ay-star”). This theory tells us that using the best available lower-bound function is *optimal*; stated another way, the better the bound function, the more efficient the search. In this case, the optimality of A* tells us that the Euclidean heuristic will certainly examine fewer vertices than Dijkstra's algorithm (which is A* with a lower bound of 0). The analytic results just described give more precise information for specific random network models.

We can also use properties of Euclidean networks to help build efficient implementations

of the abstract shortest-paths ADT, trading time for space more effectively than we can for general networks (see Exercise 21.48 through 21.50). Such algorithms are important in applications such as map processing, where networks are huge and sparse. For example, suppose that we want to develop a navigation system based on shortest paths for a map with millions of roads. We perhaps can store the map itself in a small onboard computer, but the distances and paths matrices are much too large to be stored (see Exercise 21.39 and 21.40); therefore, the all-paths algorithms of section 21.3 are not effective. Dijkstra's algorithm also may not give sufficiently short response times for huge maps. Exercise 21.77 through 21.78 explore strategies whereby we can invest a reasonable amount of preprocessing and space to provide fast responses to source– sink shortest-paths queries.

Figure 21.20 Euclidean Heuristic Cost Bounds *When we direct the shortest-path search towards the destination vertex, we can restrict the search to vertices within an ellipse around the path, as compared to the circle centered at* `s` *that is required by Dijkstra's algorithm. The radius of the circle and the shape of the ellipse are determined by the length of the shortest path.*



Exercises

● **21.68** Find a large Euclidean graph online—perhaps a map with an underlying table of locations and distances between them, telephone connections with costs, or airline routes and rates.

21.69. Using the strategies described in Exercises 17.13 through 17.15, write programs that generate random Euclidean graphs by connecting vertices arranged in a $\sqrt{V}$ -by-$\sqrt{V}$ grid.

▷ **21.70** Show that the partial SPT computed by the Euclidean heuristic is independent of the value that we use to initialize `wt[s]`. Explain how to compute the shortest-path lengths from the initial value.

IPR2023-00124
KASPERSKY EX 1036 Page 385

▷ **21.71** Show, in the style of *Figure* 21.10, what is the result when you use the Euclidean heuristic to compute a shortest `path` from `0` to `6` in the network defined in Exercise 21.1.

○ **21.72** Describe what happens if the function `dist (s, t)`, used for the Euclidean heuristic, returns the actual shortest-path length from `s` to `t` for all pairs of vertices.

21.73 Develop an ADT implementation for shortest paths in dense Euclidean graphs that is based on an adjacency-matrix network ADT and an implementation of Dijkstra's algorithm (*Program* 20.3, with an appropriate priority function).

21.74 Run empirical studies to test the effectiveness of the Euclidean shortest-path heuristic, for various Euclidean networks (see Exercise 21.8, 21.68, 21.69, and 21.80). For each graph, generate $V/10$ random pairs of vertices, and print a table that shows the average distance between the vertices, the average length of the shortest `path` between the vertices, the average ratio of the number of vertices examined with the Euclidean heuristic to the number of vertices examined with Dijkstra's algorithm, and the average ratio of the area of the ellipse associated with the Euclidean heuristic with the area of the circle associated with Dijkstra's algorithm.

21.75 Develop an implementation for the source–sink shortest-paths problem in Euclidean graphs that is based on the bidirectional search described in Exercise 21.35.

○ **21.76** Use a geometric interpretation to provide an estimate of the ratio of the number of vertices in the SPT produced by Dijkstra's algorithm for the source–sink problem to the number of vertices in the SPTs produced in the two-way version described in Exercise 21.75.

21.77 Develop an ADT implementation for shortest paths in Euclidean graphs that performs the following preprocessing step: Divide the map region into a W-by-W grid, and then use Floyd's all-pairs shortest-paths algorithm to compute a W^2-by-W^2 array, where row i and column j contain the length of a shortest `path` connecting any vertex in grid square i to any vertex in grid square j. Then, use these shortest-path lengths as lower bounds to improve the Euclidean heuristic. Experiment with a few different values of W such that you expect a small constant number of vertices per grid square.

21.78 Develop an implementation of the all-pairs shortest-paths ADT for Euclidean graphs that combines the ideas in Exercise 21.75 and 21.77.

21.79 Run empirical studies to compare the effectiveness of the heuristics described in Exercise 21.75 through 21.78, for various Euclidean networks (see Exercise 21.8, 21.68, 21.69, and 21.80).

IPR2023-00124
KASPERSKY EX 1036 Page 386

21.80 Expand your empirical studies to include Euclidean graphs that are derived by removal of all vertices and edges from a circle of radius r in the center, for r = 0.1, 0.2, 0.3, and 0.4. (These graphs provide a severe test of the Euclidean heuristic.)

21.81 Give a direct implementation of Floyd's algorithm for an implementation of the network ADT for implicit Euclidean graphs defined by N points in the plane with edges that connect points within d of each other. Do not explicitly represent the graph; rather, given two vertices, compute their distance to determine whether an edge exists and, if one does, what its length is.

21.82 Develop an implementation for the scenario described in Exercise 21.81 that builds a neighbor graph and then uses Dijkstra's algorithm from each vertex (see *Program* 21.1).

21.83 Run empirical studies to compare the time and space needed by the algorithms in Exercise 21.81 and 21.82, for d = 0.1, 0.2, 0.3, and 0.4.

● **21.84** Write a client program that does dynamic graphical animations of the Euclidean heuristic. Your program should produce images like *Figure* 21.19 (see Exercise 21.38). Test your program on various Euclidean networks (see Exercise 21.8, 21.68, 21.69, and 21.80).

21.6 REDUCTION

It turns out that shortest-paths problems—particularly the general case, where negative weights are allowed (the topic of section 21.7)—represent a general mathematical model that we can use to solve a variety of other problems that seem unrelated to graph processing. This model is the first among several such general models that we encounter. As we move to more difficult problems and increasingly general models, one of the challenges that we face is to characterize precisely relationships among various problems. Given a new problem, we ask whether we can solve it easily by transforming it to a problem that we know how to solve. If we place restrictions on the problem, will we be able to solve it more easily? To help answer such questions, we digress briefly in this section to discuss the technical language that we use to describe these types of relationships among problems.

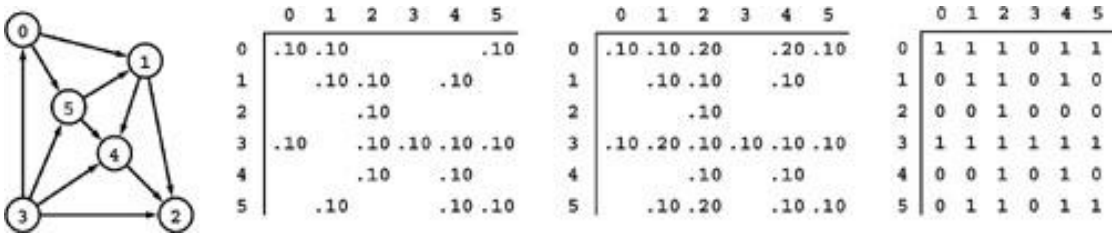
Definition 21.3 *We say that a problem A* **reduces to** *another problem B if we can use an algorithm that solves B to develop an algorithm that solves A, in a total amount of time that is, in the worst case, no more than a constant times the worst-case running time of B. We say that two problems are* **equivalent** *if they reduce to each other.*

We postpone until Part 8 a rigorous definition of what it means to "use" one algorithm to

IPR2023-00124
KASPERSKY EX 1036 Page 387

"develop" another. For most applications, we are content with the following simple approach. We show that *A* reduces to *B* by demonstrating that we can solve any instance of *A* in three steps:

Figure 21.21 Transitive-Closure Reduction *Given a digraph* (*left*), *we can transform its adjacency matrix (with self-loops) into an adjacency matrix representing a network by assigning an arbitrary weight to each edge* (*left matrix*). *As usual, blank entries in the matrix represent a sentinel value that indicates the absence of an edge. Given the all-pairs shortest-paths-lengths matrix of that network* (center matrix), *the transitive closure of the digraph* (right matrix) *is simply the matrix formed by subsituting* `0` *for each sentinel and* `1` *for all other entries.*



- Transform it to an instance of *B*.
- Solve that instance of *B*.
- Transform the solution of *B* to be a solution of *A*.

As long as we can perform the transformations (and solve *B*) efficiently, we can solve *A* efficiently. To illustrate this proof technique, we consider two examples.

Property 21.12 *The transitive-closure problem reduces to the all-pairs shortest-paths problem with nonnegative weights.*

Proof: We have already pointed out the direct relationship between Warshall's algorithm and Floyd's algorithm. Another way to consider that relationship, in the present context, is to imagine that we need to compute the transitive closure of digraphs using a library function that computes all shortest paths in networks. To do so, we add self-loops if they are not present in the digraph; then, we build a network directly from the adjacency matrix of the digraph, with an arbitrary weight (say `0.1`) corresponding to each `1` and the sentinel weight corresponding to each `0`. Then, we call the all-pairs shortest-paths function. Next, we can easily compute the transitive closure from the all-pairs shortest-paths matrix that the function computes: Given any two vertices `u` and `v`, there is a `path` from `u` to `v` in the digraph if and only if the length of the `path` from `u` to `v` in the network is nonzero (see *Figure* 21.21)■

This property is a formal statement that the transitive-closure problem is no more diffi-

cult than the all-pairs shortest-paths problem. Since we happen to know algorithms for transitive closure that are even faster than the algorithms that we know for all-pairs shortest-paths problems, this information is no surprise. Reduction is more interesting when we use it to establish a relationship between problems that we do not know how to solve, or between such problems and other problems that we can solve.

Property 21.13 *In networks with no constraints on edge weights, the longest-path and shortest-path problems (single-source or all-pairs) are equivalent.*

Proof: Given a shortest-path problem, negate all the weights. A longest `path` (a `path` with the highest weight) in the modified network is a shortest `path` in the original network. An identical argument shows that the shortest-path problem reduces to the longest-path problem.

This proof is trivial, but this property also illustrates that care is justified in stating and proving reductions, because it is easy to take reductions for granted and thus to be misled. For example, it is decidedly *not* true that the longest-path and shortest-path problems are equivalent in networks with nonnegative weights.

At the beginning of this chapter, we outlined an argument that shows that the problem of finding shortest paths in undirected weighted graphs reduces to the problem of finding shortest paths in networks, so we can use our algorithms for networks to solve shortest-paths problems in undirected weighted graphs. Two further points about this reduction are worth contemplating in the present context. First, the converse does not hold: knowing how to solve shortest-paths problems in undirected weighted graphs does not help us to solve them in networks. Second, we saw a flaw in the argument: If edge weights could be negative, the reduction gives networks with negative cycles; and we do not know how to find shortest paths in such networks. Even though the reduction fails, it turns out to be still possible to find shortest paths in undirected weighted graphs with no negative cycles with an unexpectedly complicated algorithm (*see reference section*). Since this problem does not reduce to the directed version, this algorithm does not help us to solve the shortest-path problem in general networks.

The concept of reduction essentially describes the process of using one ADT to implement another, as is done routinely by modern systems programmers. If two problems are equivalent, we know that, if we can solve either of them efficiently, we can solve the other efficiently. We often find simple one-to-one correspondences, such as the one in Property 21.13, that show two problems to be equivalent. In this case, we have not yet discussed how to solve either problem, but it is useful to know that, if we could find an efficient solution to one of them, we could use that solution to solve the other one. We saw another example in Chapter 17: When faced with the problem of determining whether or not a

IPR2023-00124
KASPERSKY EX 1036 Page 389

graph has an odd cycle, we noted that the problem is equivalent to determining whether or not the graph is two-colorable.

Reduction has two primary applications in the design and analysis of algorithms. First, it helps us to classify problems according to their difficulty at an appropriate abstract level without necessarily developing and analyzing full implementations. Second, we often do reductions to establish lower bounds on the difficulty of solving various problems, to help indicate when to stop looking for better algorithms. We have seen examples of these uses in Sections 19.3 and 20.7;we see others later in this section.

Beyond these direct practical uses, the concept of reduction also has widespread and profound implications for the theory of computation; implications that are important for us to understand as we tackle increasingly difficult problems. We discuss this topic briefly at the end of this section and consider it in full formal detail in Part 8.

The constraint that the cost of the transformations should not dominate is a natural one and often applies. In many cases, however, we might choose to use reduction even when the cost of the transformations does dominate. One of the most important uses of reduction is to provide efficient solutions to problems that might otherwise seem intractable, by performing a transformation to a well-understood problem that we know how to solve efficiently. Reducing A to B, even if computing the transformations is much more expensive than is solving B, may give us a much more efficient algorithm for solving A than we could otherwise devise. There are many other possibilities. Perhaps we are interested in expected cost, rather than worst-case. Perhaps we need to solve two problems B and C to solve A. Perhaps we need to solve multiple instances of B. We leave further discussion of such variations until Part 8, because all the examples that we consider before then are of the simple type just discussed.

In the particular case where we solve a problem A by simplifying another problem B, we know that A reduces to B, but not necessarily vice versa. For example, selection reduces to sorting because we can find the k th smallest element in a file by sorting the file and then indexing (or scanning) to the k th position, but this fact certainly does not imply that sorting reduces to selection. In the present context, the shortest-paths problem for weighted DAGs and the shortest-paths problem for networks with positive weights both reduce to the general shortest-paths problem. This use of reduction corresponds to the intuitive notion of one problem being more general than another. Any sorting algorithm solves any selection problem, and, if we can solve the shortest-paths problem in general networks, we certainly can use that solution for networks with various restrictions; but the converse is not necessarily true.

This use of reduction is helpful, but the concept becomes more useful when we use it to

IPR2023-00124
KASPERSKY EX 1036 Page 390

gain information about the relationships between problems in different domains. For example, consider the following problems, which seem at first blush to be far removed from graph processing. Through reduction, we can develop specific relationships between these problems and the shortest-paths problem.

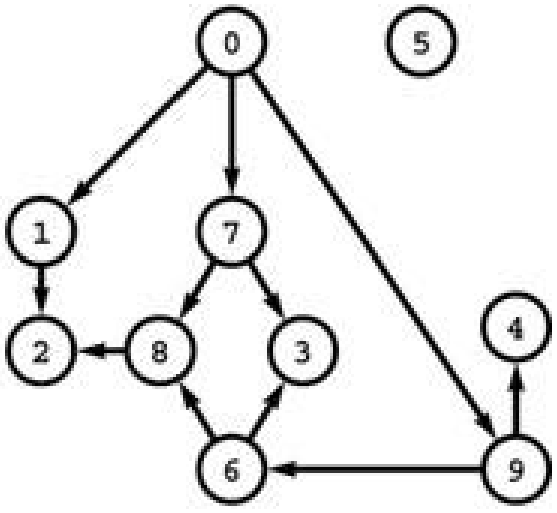
Job scheduling A large set of jobs, of varying durations, needs to be performed. We can be working on any number of jobs at a given time, but a set of precedence relationships specify, for a set of pairs of jobs, that the first must be completed before the second can be started. What is the minimum amount of time required to complete all the jobs while satisfying all the precedence constraints? Specifically, given a set of jobs (with durations) and a set of precedence constraints, schedule the jobs (find a start time for each) so as to achieve this minimum.

Figure 21.22 depicts an example instance of the job-scheduling problem. It uses a natural network representation, which we use in a moment as the basis for a reduction. This version of the problem is perhaps the simplest of literally hundreds of versions that have been studied—versions that involve other job characteristics and other constraints, such as the assignment of personnel or other resources to the jobs, other costs associated with specific jobs, deadlines, and so forth. In this context, the version that we have described is commonly called *precedence-constrained scheduling with unlimited parallelism*; we use the term *job scheduling* as shorthand.

To help us to develop an algorithm that solves the job-scheduling problem, we consider the following problem, which is widely applicable in its own right.

Figure 21.22 Job Scheduling *In this network, vertices represent jobs to be completed (with weights indicating the amount of time required) and edges represent precedence relationships between them. For example, the edges from* 7 *to* 8 *and* 3 *mean that job* 7 *must be finished before job* 8 *or job* 3 *can be started. What is the minimum amount of time required to complete all the jobs?*

```
0  .41
1  .51
2  .50
3  .36
4  .38
5  .45
6  .21
7  .32
8  .32
9  .29
```



Difference constraints Assign nonnegative values to a set variables x_0 through x_n that minimize the value of x_n while satisfying a set of *difference constraints* on the variables, each of which specifies that the difference between two of the variables must be greater than or equal to a given constant.

Figure 21.23 depicts an example instance of this problem. It is a purely abstract mathematical formulation that can serve as the basis for solving numerous practical problems (*see reference section*).

The difference-constraint problem is a special case of a much more general problem where we allow general linear combinations of the variables in the equations:

Linear programming Assign nonnegative values to a set of variables x_0 through x_n that minimize the value of a specified linear combination of the variables, subject to a set of constraints on the variables, each of which specifies that a given linear combination of the variables must be greater than or equal to a given constant.

Linear programming is a widely used general approach to solving a broad class of optimization problems that we will not consider it in detail until Part 8. Clearly, the difference-constraints problem reduces to linear programming, as do many other problems. For the moment, our interest is in the relationships among the difference-constraints, job-scheduling, and shortest-paths problems.

Property 21.14 *The job-scheduling problem reduces to the difference-constraints problem.*

Proof: Add a dummy job and a precedence constraint for each job saying that the job must finish before the dummy job starts. Given a job-scheduling problem, define a system of difference equations where each job `i` corresponds to a variable x_i, and the constraint that `j` cannot start until `i` finishes corresponds to the equation $x_j \geq x_i + c_i$, where c_i is the length of job `i`. The solution to the difference-constraints problem gives precisely a solution to the job-scheduling problem, with the value of each variable specifying the start time of the corresponding job.■

Figure 21.23 Difference Constraints *Finding an assignment of nonnegative values to the variables that minimizes the value of x_{10} subject to this set of inequalities is equivalent to the job-scheduling problem instance illustrated in Figure 21.22. For example, the equation $x_8 \geq x_7 + .32$means that job 8 cannot start until job 7 is completed.*

IPR2023-00124
KASPERSKY EX 1036 Page 392

$$
\begin{aligned}
x_1 - x_0 &\geq .41 \\
x_7 - x_0 &\geq .41 \\
x_9 - x_0 &\geq .41 \\
x_2 - x_1 &\geq .51 \\
x_8 - x_6 &\geq .21 \\
x_3 - x_6 &\geq .21 \\
x_8 - x_7 &\geq .32 \\
x_3 - x_7 &\geq .32 \\
x_2 - x_8 &\geq .32 \\
x_4 - x_9 &\geq .29 \\
x_6 - x_9 &\geq .29 \\
x_{10} - x_2 &\geq .50 \\
x_{10} - x_3 &\geq .36 \\
x_{10} - x_4 &\geq .38 \\
x_{10} - x_5 &\geq .45
\end{aligned}
$$

Figure 21.23 illustrates the system of difference equations created by this reduction for the job-scheduling problem in *Figure* 21.22. The practical significance of this reduction is that we can use to solve job-scheduling problems any algorithm that can solve difference-constraint problems.

It is instructive to consider whether we can use this construction in the opposite way: Given a job-scheduling algorithm, can we use it to solve difference-constraints problems? The answer to this question is that the correspondence in the proof of Property 21.14 does *not* help us to show that the difference-constraints problem reduces to the job-scheduling problem, because the systems of difference equations that we get from job-scheduling problems have a property that does not necessarily hold in every difference-constraints problem. Specifically, if two equations have the same second variable, then they have the same constant. Therefore, an algorithm for job scheduling does not immediately give a direct way to solve a system of difference equations that contains two equations $x_i - x_j \geq a$ and $x_k - x_j \geq b$, where $a \neq b$. When proving reductions, we need to be aware of situations like this: A proof that A reduces to B must show that we can use an algorithm for solving B to solve *any* instance of A.

By construction, the constants in the difference-constraints problems produced by the construction in the proof of Property 21.14 are always nonnegative. This fact turns out to be significant:

Property 21.15 *The difference-constraints problem with positive constants is equivalent to the single-source longest-paths problem in an acyclic network.*

Proof: Given a system of difference equations, build a network where each variable x_i corresponds to a vertex `i` and each equation $x_i - x_j \geq c$ corresponds to an edge `i-j` of weight `c`. For example, assigning to each edge in the digraph of *Figure* 21.22 the weight of its source vertex gives the network corresponding to the set of difference equations in *Figure* 21.23. Add a dummy vertex to the network, with a zero-weight edge to every other vertex. If the network has a cycle, the system of difference equations has no solution (because the positive weights imply that the values of the variables corresponding to each vertex strictly decrease as we move along a path, and, therefore, a cycle would imply that some variable is less than itself), so report that fact. Otherwise, the network has no cycle, so solve the single-source longest-paths problem from the dummy vertex. There exists a longest `path` for every vertex because the network is acyclic (see section 21.4). Assign to each variable the length of the longest `path` to the corresponding vertex in the network from the dummy vertex. For each variable, this `path` is evidence that its value satisfies the constraints and that no smaller value does so.

Unlike the proof of Property 21.14, this proof does extend to show that the two problems are equivalent because the construction works in both directions. We have no constraint that two equations with the same second variable in the equation must have the same constants, and no constraint that edges leaving any given vertex in the network must have the same weight. Given any acyclic network with positive weights, the same correspondence gives a system of difference constraints with positive constants whose solution directly yields a solution to the single-source longest-paths problem in the network. Details of this proof are left as an exercise (see Exercise 21.90).■

The network in *Figure* 21.22 depicts this correspondence for our sample problem, and *Figure* 21.15 shows the computation of the longest paths in the network, using *Program* 21.6 (the dummy start vertex is implicit in the implementation). The schedule that is computed in this way is shown in *Figure* 21.24.

Program 21.8 is an implementation that shows the application of this theory in a practical setting. It transforms any instance of the job-scheduling problem into an instance of the longest-path problem in acyclic networks, then uses *Program* 21.6 to solve it.

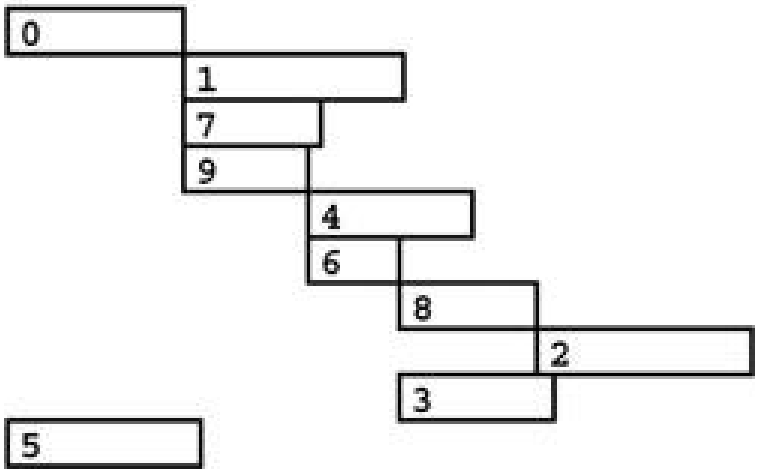
We have been implicitly assuming that a solution exists for any instance of the job-scheduling problem; however, if there is a cycle in the set of precedence constraints, then there is no way to schedule the jobs to meet them. Before looking for longest paths, we should check for this condition by checking whether the corresponding network has a cycle (see Exercise 21.100). Such a situation is typical, and a specific technical term is normally used

to describe it.

Definition 21.4 *A problem instance that admits no solution is said to be* **infeasible**.

Figure 21.24 Job Schedule *This figure illustrates the solution to the job-scheduling problem of Figure 21.22, derived from the correspondence between longest paths in weighted DAGs and job schedules. The longest* `path` *lengths in the* `wt` *array that is computed by the longest paths algorithm in Program 21.6 (see Figure 21.15) are precisely the required job start times (top, right column). We start jobs* `0` *and* `5` *at time* `0`*, jobs* `1`*,* `7`*, and* `9` *at time* `.41`*, jobs* `4` *and* `6` *at time* `.70`*, and so forth.*

```
 0    0
 1    .41
 2   1.23
 3    .91
 4    .70
 5    0
 6    .70
 7    .41
 8    .91
 9    .41
10   1.73
```



In other words, for job-scheduling problems, the question of determining whether a job-scheduling problem instance is feasible reduces to the problem of determining whether a digraph is acyclic. As we move to ever-more-complicated problems, the question of feasibility becomes an ever-more-important (and ever-more-difficult!) part of our computational burden.

Program 21.8 Job scheduling

This implementation reads a list of jobs with lengths followed by a list of precedence constraints from standard input, then prints on standard output a list of job starting times that satisfy the constraints. It solves the job-scheduling problem by reducing it to the longest-paths problem for acyclic networks, using Properties 21.14 and 21.15 and *Program* 21.6. Minor adjustments to fit the interface (e.g., we do not use the `st` array here) are typical artifacts of implementing a reduction by using an existing implementation.

```
#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]);
}
```

We have now considered three interrelated problems. We might have shown directly that the job-scheduling problem reduces to the single-source longest-paths problem in acyclic networks, but we have also shown that we can solve any difference-constraints problem (with positive constants) in a similar manner (see Exercise 21.94), as well as any other problem that reduces to a difference-constraints problem or a job-scheduling problem. We could, alternatively, develop an algorithm to solve the difference-constraints problem and use that algorithm to solve the other problems, but we have *not* shown that a solution to the job-scheduling problem would give us a way to solve the others.

These examples illustrate the use of reduction to broaden the applicability of proven implementations. Indeed, modern systems programming emphasizes the need to reuse soft-

IPR2023-00124
KASPERSKY EX 1036 Page 396

ware by developing new interfaces and using existing software resources to build implementations. This important process, which is sometimes referred to as *library programming*, is a practical realization of the idea of reduction.

Library programming is extremely important in practice, but it represents only part of the story of the implications of reduction. To illustrate this point, we consider the following version of the job-scheduling problem.

Job scheduling with deadlines Allow an additional type of constraint in the job-scheduling problem, to specify that a job must begin *before* a specified amount of time has elapsed, relative to another job. (Conventional deadlines are relative to the start job.) Such constraints are commonly needed in time-critical manufacturing processes and in many other applications, and they can make the job-scheduling problem considerably more difficult to solve.

Suppose that we need to add a constraint to our example of Figures 21.22 through 21.24 that job 2 must start earlier than a certain number c of time units after job 4 starts. If c is greater than .53, then the schedule that we have computed fits the bill, since it says to start job 2 at time 1.23, which is .53 after the end time of job 4 (which starts at .70). If c is less than .53, we can shift the start time of 4 later to meet the constraint. If job 4 were a long job, this change could increase the finish time of the whole schedule. Worse, if there are other constraints on job 4, we may not be able to shift its start time. Indeed, we may find ourselves with constraints that no schedule can meet: for instance, we could not satisfy a constraint in our example that job 2 must start earlier than d time units after the start of job 6 for d less than .53 because the constraints that 2 must follow 8 and 8 must follow 6 imply that 2 must start *later* than .53 time units after the start of 6.

If we add both of the two constraints described in the previous paragraph to the example, then both of them affect the time that 4 can be scheduled, the finish time of the whole schedule, and whether a feasible schedule exists, depending on the values of c and d. Adding more constraints of this type multiplies the possibilities, and turns an easy problem into a difficult one. Therefore, we are justified in seeking the approach of reducing the problem to a known problem.

Property 21.16 *The job-scheduling-with-deadlines problem reduces to the shortest-paths problem (with negative weights allowed).*

Proof: Convert precedence constraints to inequalities using the same reduction described in Property 21.14. For any deadline constraint, add an inequality $x_i - x_j \le d_j$, or, equivalently $x_j - x_i \ge -d_j$, where d_j is a positive constant. Convert the set of inequalities to a network using the same reduction described in Property 21.15. Negate all the weights. By the

same construction given in the proof of Property 21.15, any shortest-path tree rooted at 0 in the network corresponds to a schedule■

This reduction takes us to the realm of shortest paths with negative weights. It says that if we can find an efficient solution to the shortest-paths problem with negative weights, then we can find an efficient solution to the job-scheduling problem with deadlines. (Again, the correspondence in the proof of Property 21.16 does *not* establish the converse (see Exercise 21.91).)

Adding deadlines to the job-scheduling problem corresponds to allowing negative constants in the difference-constraints problem and negative weights in the shortest-paths problem. (This change also requires that we modify the difference-constraints problem to properly handle the analog of negative cycles in the shortest paths problem.) These more general versions of these problems are more difficult to solve than the versions that we first considered, but they are also likely to be more useful as more general models. A plausible approach to solving all of them would seem to be to seek an efficient solution to the shortest-paths problem with negative weights.

Unfortunately, there is a fundamental difficulty with this approach; it illustrates the other part of the story in the use of reduction to assess the relative difficulty of problems. We have been using reduction in a positive sense, to expand the applicability of solutions to general problems; but it also applies in a negative sense, to show the limits on such expansion.

The difficulty is that the general shortest-paths problem is too hard to solve. We see next how the concept of reduction helps us to make this statement with precision and conviction. In Section 17.8, we discussed a set of problems, known as the NP-hard problems, that we consider to be intractable because all known algorithms for solving them require exponential time in the worst case. We show here that the general shortest-paths problem is NP-hard.

As mentioned briefly in Section 17.8 and discussed in detail in Part 8, we generally take the fact that a problem is NP-hard to mean not just that no efficient algorithm is known that is guaranteed to solve the problem, but also that we have little hope of finding one. In this context, we use the term *efficient* to refer to algorithms whose running time is bounded by some polynomial function of the size of the input, in the worst case. We assume that the discovery of an efficient algorithm to solve any NP-hard problem would be a stunning research breakthrough. The concept of NP-hardness is important in identifying problems that are difficult to solve, because it is often easy to prove that a problem is NP-hard, using the following technique:

Property 21.17 *A problem is NP-hard if there is an efficient reduction to it from any NP-hard problem.*

This property depends on the precise meaning of an *efficient reduction* from one problem *A* to another problem *B*. We defer such definitions to Part 8 (two different definitions are commonly used). For the moment, we simply use the term to cover the case where we have efficient algorithms both to transform an instance of *A* to an instance of *B* and to transform a solution of *B* to a solution of *A*.

Now, suppose that we have an efficient reduction from an NP-hard problem *A* to a given problem *B*. The proof is by contradiction: If we have an efficient algorithm for *B*, then we could use it to solve any instance of *A* in polynomial time, by reduction (transform the given instance of *A* to an instance of *B*, solve that problem, then transform the solution). But no known algorithm can make such a guarantee for *A* (because *A* is NP-hard), so the assumption that there exists a polynomial-time algorithm for *B* is incorrect: *B* is also NP-hard■

This technique is extremely important because people have used it to show a huge number of problems to be NP-hard, giving us a broad variety of problems from which to choose when we want to develop a proof that a new problem is NP-hard. For example, we encountered one of the classic NP-hard problems in Section 17.7. The Hamilton-path problem, which asks whether there is a simple `path` containing all the vertices in a given graph, was one of the first problems shown to be NP-hard (*see reference section*). It is easy to formulate as a shortest-paths problem, so Property 21.17 implies that the shortest-paths problem itself is NP-hard:

Property 21.18 *In networks with edge weights that could be negative, shortest-paths problems are NP-hard.*

Proof: Our proof consists of reducing the Hamilton-path problem to the shortest-paths problem. That is, we show that we could use any algorithm that can find shortest paths in networks with negative edge weights to solve the Hamilton-path problem. Given an undirected graph, we build a network with edges in both directions corresponding to each edge in the graph and with all edges having weight `-1`. The shortest (simple) `path` starting at any vertex in this network is of length $-(V-1)$ if and only if the graph has a Hamilton path. Note that this network is replete with negative cycles. Not only does every cycle in the graph correspond to a negative cycle in the network, but also every *edge* in the graph corresponds to a cycle of weight `-2` in the network.

The implication of this construction is that the shortest-paths problem is NP-hard, because, if we could develop an efficient algorithm for the shortest-paths problem in net-

works, then we would have an efficient algorithm for the Hamilton-path problem in graphs■

One response to the discovery that a given problem is NP-hard is to seek versions of that problem that we *can* solve. For shortest-paths problems, we are caught between having a host of efficient algorithms for acyclic networks or for networks in which edge weights are nonnegative and having no good solution for networks that could have cycles and negative weights. Are there other kinds of networks that we can address? That is the subject of section 21.7. There, for example, we see that the job-scheduling-with-deadlines problem reduces to a version of the shortest-paths problem that we can solve efficiently. This situation is typical: as we address ever-more-difficult computational problems, we find ourselves working to identify the versions of those problems that we can expect to solve.

As these examples illustrate, reduction is a simple technique that is helpful in algorithm design, and we use it frequently. Either we can solve a new problem by proving that it reduces to a problem that we know how to solve, or we can prove that the new problem will be difficult by proving that a problem that we know to be difficult reduces to the problem in question.

Table 21.3 Reduction implications This table summarizes some implications of reducing a problem *A* to another problem *B*, with examples that we have discussed in this section. The profound implications of cases 9 and 10 are so far-reaching that we generally assume that it is not possible to prove such reductions (see Part 8). Reduction is most useful in cases 1, 6, 11, and 16, to learn a new algorithm for *A* or prove a lower bound on *B*; cases 13-15, to learn new algorithms for *A*; and case 12, to learn the difficulty of *B*.

IPR2023-00124
KASPERSKY EX 1036 Page 400

	A	B	$A \Rightarrow B$ implication	example
1	easy	easy	new B lower bound	sorting $\Rightarrow$ EMST
2	easy	tractable	none	TC $\Rightarrow$ APSP(+)
3	easy	intractable	none	SSSP(DAG) $\Rightarrow$ SSSP($\pm$)
4	easy	unknown	none	
5	tractable	easy	A easy	
6	tractable	tractable	new A solution	DC(+) $\Rightarrow$ SSSP(DAG)
7	tractable	intractable	none	
8	tractable	unknown	none	
9	intractable	easy	profound	
10	intractable	tractable	profound	
11	intractable	intractable	same as 1 or 6	SSLP($\pm$) $\Rightarrow$ SSSP($\pm$)
12	intractable	unknown	B intractable	HP $\Rightarrow$ SSSP($\pm$)
13	unknown	easy	A easy	JS $\Rightarrow$ SSSP(DAG)
14	unknown	tractable	A tractable	
15	unknown	intractable	A solvable	
16	unknown	unknown	same as 1 or 6	JSWD $\Rightarrow$ SSSP($\pm$)

Key:

EMST	Euclidean minimum spanning tree
TC	transitive closure
APSP	all-pairs shortest paths
SSSP	single-source shortest paths
SSLP	single-source longest paths
(+)	(in networks with nonnegative weights)
($\pm$)	(in networks with weights that could be negative)
(DAG)	(in acyclic networks)
DC	difference constraints
HP	Hamilton paths
JS(WD)	job scheduling (with deadlines)

Table 21.3 gives us a more detailed look at the various implications of reduction results among the four general problem classes that we discussed in Chapter 17. Note that there are several cases where a reduction provides no new information; for example, although selection reduces to sorting and the problem of finding longest paths in acyclic networks reduces to the problem of finding shortest paths in general networks, these facts shed no new light on the relative difficulty of the problems. In other cases, the reduction may or may not provide new information; in still other cases, the implications of a reduction are truly profound. To develop these concepts, we need a precise and formal description of reduction, as we discuss in detail in Part 8; here, we summarize informally the most important uses of reduction in practice, with examples that we have already seen.

IPR2023-00124
KASPERSKY EX 1036 Page 401

Upper bounds If we have an efficient algorithm for a problem *B* and can prove that *A* reduces to *B*, then we have an efficient algorithm for *A*. There may exist some other better algorithm for *A*, but *B* 's performance is an upper bound on the best that we can do for *A*. For example, our proof that job scheduling reduces to longest paths in acyclic networks makes our algorithm for the latter an efficient algorithm for the former.

Lower bounds If we know that any algorithm for problem *A* has certain resource requirements, and we can prove that *A* reduces to *B*, then we know that *B* has at least those same resource requirements, because a better algorithm for *B* would imply the existence of a better algorithm for *A* (as long as the cost of the reduction is lower than the cost of *B*). That is,*A* 's performance is a lower bound on the best that we can do for *B*. For example, we used this technique in section 19.3 to show that computing the transitive closure is as difficult as Boolean matrix multiplication, and we used it in Section 20.7 to show that computing the Euclidean MST is as difficult as sorting.

Intractability In particular, we can prove a problem to be intractable by showing that an intractable problem reduces to it. For example, Property 21.18 shows that the shortest-paths problem is intractable because the Hamilton-path problem reduces to the shortest-paths problem.

Beyond these general implications, it is clear that more detailed information about the performance of specific algorithms to solve specific problems can be directly relevant to other problems that reduce to the first ones. When we find an upper bound, we can analyze the associated algorithm, run empirical studies, and so forth to determine whether it represents a better solution to the problem. When we develop a good general-purpose algorithm, we can invest in developing and testing a good implementation and then develop associated ADTs that expand its applicability.

We use reduction as a basic tool in this and the next chapter. We emphasize the general relevance of the problems that we consider, and the general applicability of the algorithms that solve them, by reducing other problems to them. It is also important to be aware of a hierarchy among increasingly general problem-formulation models. For example, linear programming is a general formulation that is important not just because many problems reduce to it but also because it is *not* known to be NP-hard. In other words, there is no known way to reduce the general shortest-paths problem (or any other NP-hard problem) to linear programming. We discuss such issues in Part 8.

Not all problems can be solved, but good general models have been devised that are suitable for broad classes of problems that we do know how to solve. Shortest paths in networks is our first example of such a model. As we move to ever-more-general problem domains, we enter the field of *operations research (OR*), the study of mathematical meth-

ods of decision making, where developing and studying such models is central. One key challenge in OR is find the model that is most appropriate for solving a problem and to fit the problem to the model. This activity is sometimes known as *mathematical programming* (a name given to it before the advent of computers and the new use of the word "programming"). Reduction is a modern concept that is in the same spirit as mathematical programming and is the basis for our understanding of the cost of computation in a broad variety of applications.

Exercises

▷ **21.85** Use the reduction of Property 21.12 to develop a transitive-closure implementation (the ADT function `GRAPHtc` of section 19.3) using the all-pairs shortest-paths ADT of section 21.3.

21.86 Show that the problem of computing the number of strong components in a digraph reduces to the all-pairs shortest-paths problem with nonnegative weights.

21.87 Give the difference-constraints and shortest-paths problems that correspond—according to the constructions of Properties 21.14 and 21.15—to the job-scheduling problem, where jobs 0 to 7 have lengths

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

and constraints

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

respectively.

▷ **21.88** Give a solution to the job-scheduling problem of Exercise 21.87.

○ **21.89** Suppose that the jobs in Exercise 21.87 also have the constraints that job 1 must start before job 6 ends, and job 2 must start before job 4 ends. Give the shortest-paths problem to which this problem reduces, using the construction described in the proof of Property 21.16.

21.90 Show that the all-pairs longest-paths problem in acyclic networks with positive weights reduces to the difference-constraints problem with positive constants.

▷ **21.91** Explain why the correspondence in the proof of Property 21.16 does not extend to show that the shortest-paths problem reduces to the job-scheduling-with-deadlines

IPR2023-00124
KASPERSKY EX 1036 Page 403

problem.

21.92 Extend *Program* 21.8 to use symbolic names instead of integers to refer to jobs (see *Program* 17.10).

21.93 Design an ADT interface that provides clients with the ability to pose and solve difference-constraints problems.

21.94 Provide ADT function implementations for your interface from Exercise 21.93, basing your solution to the difference-constraints problem on a reduction to the shortest-paths problem in acyclic networks.

21.95 Provide an implementation for `GRAPHsp` (which is intended for use in acyclic networks) that is based on a reduction to the difference-constraints problem and uses your interface from Exercise 21.93.

○ **21.96** Your solution to the shortest-paths problem in acyclic networks for Exercise 21.95 assumes the existence of an implementation that solves the difference-constraints problem. What happens if you use the implementation from Exercise 21.94, which assumes the existence of an implementation to the shortest-paths problem in acyclic networks?

○ **21.97** Prove the equivalence of any two NP-hard problems (that is, choose to problems and prove that they reduce to each other).

●● **21.98** Give an explicit construction that reduces the shortest-paths problem in networks with integer weights to the Hamilton-path problem.

● **21.99** Use reduction to implement an ADT function that uses a network ADT that solves the single-source shortest paths problem to solve the following problem: Given a digraph, a vertex-indexed array of positive weights, and a start vertex `v`, find the paths from `v` to each other vertex such that the sum of the weights of the vertices on the `path` is minimized.

○ **21.100** *Program* 21.8 does not check whether the job-scheduling problem that it takes as input is feasible (has a cycle). Characterize the schedules that it prints out for infeasible problems.

21.101 Design an ADT interface that gives clients the ability to pose and solve job-scheduling problems. Provide ADT function implementations for your interface, basing your solution to the job-scheduling problem on a reduction to the shortest-paths problem in acyclic networks, as in *Program* 21.8.

IPR2023-00124
KASPERSKY EX 1036 Page 404

○ **21.102** Add a function to your ADT interface (and provide an implementation) that prints out a longest `path` in the schedule. (Such a `path` is known as a *critical path*.)

21.103 Provide an ADT function implementation for your interface from Exercise 21.101 that outputs a PostScript program that draws the schedule in the style of *Figure* 21.24 (see Section 4.3).

○ **21.104** Develop a model for generating job-scheduling problems. Use this model to test your implementations of Exercise 21.101 and 21.103 for a reasonable set of problem sizes.

21.105 Provide ADT function implementations for your interface from Exercise 21.101, basing your solution to the job-scheduling problem on a reduction to the difference-constraints problem.

○ **21.106** A PERT (performance-evaluation-review-technique) chart is a network that represents a job-scheduling problem, with *edges* representing jobs, as described in *Figure* 21.25. Develop an implementation of your job-scheduling interface of Exercise 21.101 that is based on PERT charts.

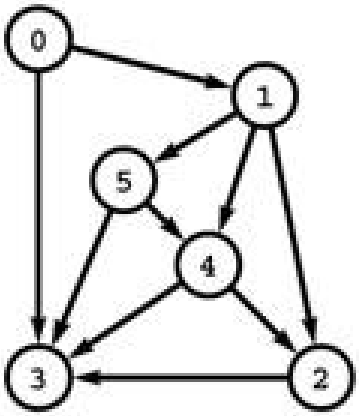
21.107 How many vertices are there in a PERT chart for a job-scheduling problem with V jobs and E constraints?

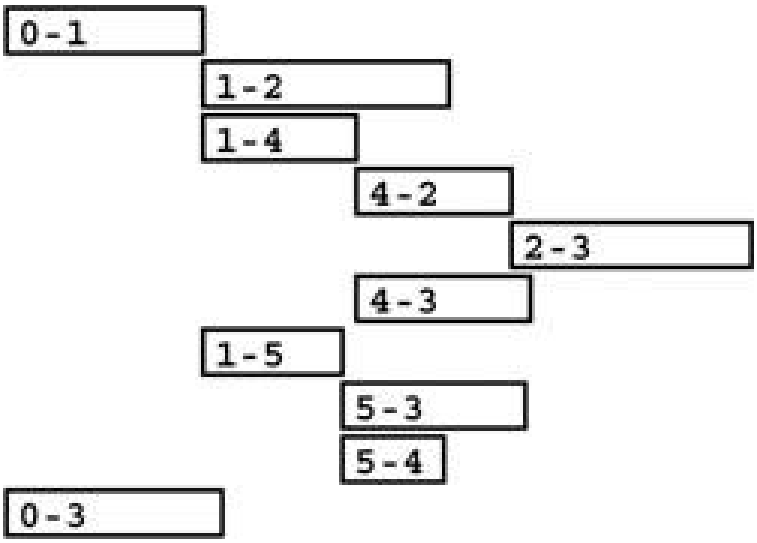
21.108 Write programs to convert between the edge-based job-scheduling representation (PERT chart) discussed in Exercise 21.106 and the vertex-based representation used in the text (see *Figure* 21.22).

Figure 21.25 A PERT Chart *A PERT chart is a network representation for job-scheduling problems where we represent jobs by edges. The network at the top is a representation of the job-scheduling problem depicted in Figure 21.22, where jobs* `0` *through* `9` *in Figure* 21.22 *are represented by edges* `0-1, 1-2, 2-3, 4-3, 5-3, 0-3, 5-4, 1-4, 4-2`, *and* `1-5`, *respectively, here. The critical* `path` *in the schedule is the longest* `path` *in the network.*

IPR2023-00124
KASPERSKY EX 1036 Page 405

```
0-1  .41
1-2  .51
2-3  .50
4-3  .36
5-3  .38
0-3  .45
5-4  .21
1-4  .32
4-2  .32
1-5  .29
```





21.7 NEGATIVE WEIGHTS

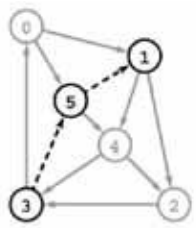
We now turn to the challenge of coping with negative weights in shortest-paths problems. Perhaps negative edge weights seem unlikely given our focus through most of this chapter on intuitive examples, where weights represent distances or costs; however, we also saw in section 21.6 that negative edge weights arise in a natural way when we reduce other problems to shortest-paths problems. Negative weights are not merely a mathematical curiosity; on the contrary, they significantly extend the applicability of the shortest-paths problems as a model for solving other problems. This potential utility is our motivation to search for efficient algorithms to solve network problems that involve negative weights.

Figure 21.26 A Sample Network with Negative Edges *This sample network is the same as the network depicted in Figure 21.1, except that the edges* `3-5` *and* `5-1` *are negative. Naturally, this change dramatically affects the shortest paths*

IPR2023-00124
KASPERSKY EX 1036 Page 406

structure, as we can easily see by comparing the distance and `path` *matrices at the right with their counterparts in Figure 21.9. For example, the shortest* `path` *from* `0` *to* `1` *in this network is* `0-5-1`*, which is of length* `0`*; and the shortest* `path` *from* `2` *to* `1` *is* `2-3-5-1`*, which is of length* -.17.

```
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

Figure 21.26 is a small example that illustrates the effects of introducing negative weights on a network's shortest paths. Perhaps the most important effect is that, when negative weights are present, *low-weight shortest paths tend to have more edges than higher-weight paths*. For positive weights, our emphasis was on looking for *shortcuts*; but when negative weights are present, we seek *detours* that use as many edges with negative weights as we can find. This effect turns our intuition in seeking "short" paths into a liability in understanding the algorithms, so we need to suppress that line of intuition and consider the problem on a basic abstract level.

The relationship shown in the proof of Property 21.18 between shortest paths in networks and Hamilton paths in graphs ties in with our observation that finding paths of low weight (which we have been calling "short") is tantamount to finding paths with a high number of edges (which we might consider to be "long"). With negative weights, we are looking for long paths rather than short paths.

The first idea that suggests itself to remedy the situation is to find the smallest (most negative) edge weight, then to add the absolute value of that number to all the edge weights to transform the network into one with no negative weights. This naive approach does not work at all, because shortest paths in the new network bear little relation to shortest paths in the old one. For example, in the network illustrated in *Figure* 21.26, the shortest `path` from `4` to `2` is `4-3-5-1-2`. If we add .38 to all the edge weights in the graph to make them all positive, the weight of this `path` grows from `.20` to `1.74`. But the weight of `4-2` grows from `.32` to just `.70`, so that edge becomes the shortest `path` from `4` to `2`. The more edges a `path` has, the more it is penalized by this transformation; that result, from the observation in the previous paragraph, is precisely the opposite of what we need. Even though this naive idea does not work, the goal of transforming the network into an equivalent one with no negative weights but the same shortest paths is worthy; at the end of the section, we consider an algorithm that achieves this goal.

Our shortest-paths algorithms to this point have all placed one of two restrictions on the

shortest-paths problem so that they can offer an efficient solution: They either disallow cycles or disallow negative weights. Is there a less stringent restriction that we could impose on networks that contain both cycles and negative weights that would still lead to tractable shortest-paths problems? We touched on an answer to this question at the beginning of the chapter, when we had to add the restriction that paths be simple so that the problem would make sense if there were negative cycles. Perhaps we should restrict attention to networks that have no such cycles?

Shortest paths in networks with no negative cycles Given a network that may have negative edge weights but does not have any negative-weight cycles, solve one of the following problems: find a shortest `path` connecting two given vertices (shortest-path problem), find shortest paths from a given vertex to all the other vertices (single-source problem), or find shortest paths connecting all pairs of vertices (all-pairs problem).

The proof of Property 21.18 leaves the door open for the possibility of efficient algorithms for solving this problem because it breaks down if we disallow negative cycles. To solve the Hamilton-path problem, we would need to be able to solve shortest-paths problems in networks that have huge numbers of negative cycles.

Moreover, many practical problems reduce precisely to the problem of finding shortest paths in networks that contain no negative cycles. We have already seen one such example.

Property 21.19 *The job-scheduling-with-deadlines problem reduces to the shortest-paths problem in networks that contain no negative cycles*.

Proof: The argument that we used in the proof of Property 21.15 shows that the construction in the proof of Property 21.16 leads to networks that contain no negative cycles. From the job-scheduling problem, we construct a difference-constraints problem with variables that correspond to job start times; from the difference-constraints problem, we construct a network. We negate all the weights to convert from a longest-paths problem to a shortest-paths problem—a transformation that corresponds to reversing the sense of all the inequalities. Any simple `path` from `i` to `j` in the network corresponds to a sequence of inequalities involving the variables. The existence of the `path` implies, by collapsing these inequalities, that $x_i - x_j \leq W_{ij}$, where w_{ij} is the sum of the weights on the `path` from `i` to `j`. A negative cycle corresponds to `0` on the left side of this inequality and a negative value on the right, so the existence of such a cycle is a contradiction■

As we noted when we first discussed the job-scheduling problem in section 21.6, this statement implicitly assumes that our job-scheduling problems are feasible (have a solution). In practice, we would not make such an assumption, and part of our computational

burden would be determining whether or not a job-scheduling-with-deadlines problem is feasible. In the construction in the proof of Property 21.19, a negative cycle in the network implies an infeasible problem, so this task corresponds to the following problem:

Negative cycle detection Does a given network have a negative cycle? If it does, find one such cycle.

On the one hand, this problem is not necessarily easy (a simple cycle-checking algorithm for digraphs does not apply); on the other hand, it is not necessarily difficult (the reduction of Property 21.16 from the Hamilton-path problem does not apply). Our first challenge will be to develop an algorithm for this task.

In the job-scheduling-with-deadlines application, negative cycles correspond to error conditions that are presumably rare, but for which we need to check. We might even develop algorithms that remove edges to break the negative cycle and iterate until there are none. In other applications, detecting negative cycles is the prime objective, as in the following example.

Arbitrage Many newspapers print tables showing conversion rates among the world's currencies (see, for example, *Figure* 21.27). We can view such tables as adjacency-matrix representations of complete networks. An edge `s-t` with weight x means that we can convert 1 unit of currency `s` into x units of currency `t`. Paths in the network specify multistep conversions. For example, if there is also an edge `t-w` with weight y, then the `path` `s-t-w` represents a way to convert 1 unit of currency `s` into xy units of currency `w`. We might expect xy to be equal to the weight of `s-w` in all cases, but such tables represent a complex dynamic system where such consistency cannot be guaranteed. If we find a case where xy is smaller than the weight of `s-w`, then we may be able to outsmart the system. Suppose that the weight of `w-s` is z and $xyz > 1$, then the cycle `s-t-w-s` gives a way to convert `1` unit of currency `s` into more than `1` units (xyz) of currency `s`. That is, we can make a 100 ($xyz - 1$) percent profit by converting from `s` to `t` to `w` back to `s`. This situation is an example of an *arbitrage* opportunity that would allow us to make unlimited profits were it not for forces outside the model, such as limitations on the size of transactions. To convert this problem to a shortest-paths problem, we take the logarithm of all the numbers so that `path` weights correspond to adding edge weights instead of multiplying them, and then we take the negative to invert the comparison. Then the edge weights might be negative or positive, and a shortest `path` from `s` to `t` gives a best way of converting from currency `s` to currency `t`. The lowest-weight cycle is the best arbitrage opportunity, but any negative cycle is of interest.

Figure 21.27 Arbitrage *The table at the top specifies conversion factors from one currency to another. For example, the second entry in the top row says that $1*

IPR2023-00124
KASPERSKY EX 1036 Page 409

buys 1.631 units of currency P. Converting $1000 to currency P and back again would yield $1000 (1.631)* (0.613) = $999, a loss of $1. But converting $1000 to currency P then to currency Y and back again yields $1000* (1.631)* (0.411)* (1.495) = $1002, a. 2% arbitrage opportunity. If we take the negative of the logarithm of all the numbers in the table (bottom), we can consider it to be the adjacency matrix for a complete network with edge weights that could be positive or negative. In this network, nodes correspond to currencies, edges to conversions, and paths to sequences of conversions. The conversion just described corresponds to the cycle* `$-P-Y-$` *in the graph, which has weight* `−0.489 + 0.890 − 0.402 = −.002`*. The best arbitrage opportunity is the shortest cycle in the graph.*

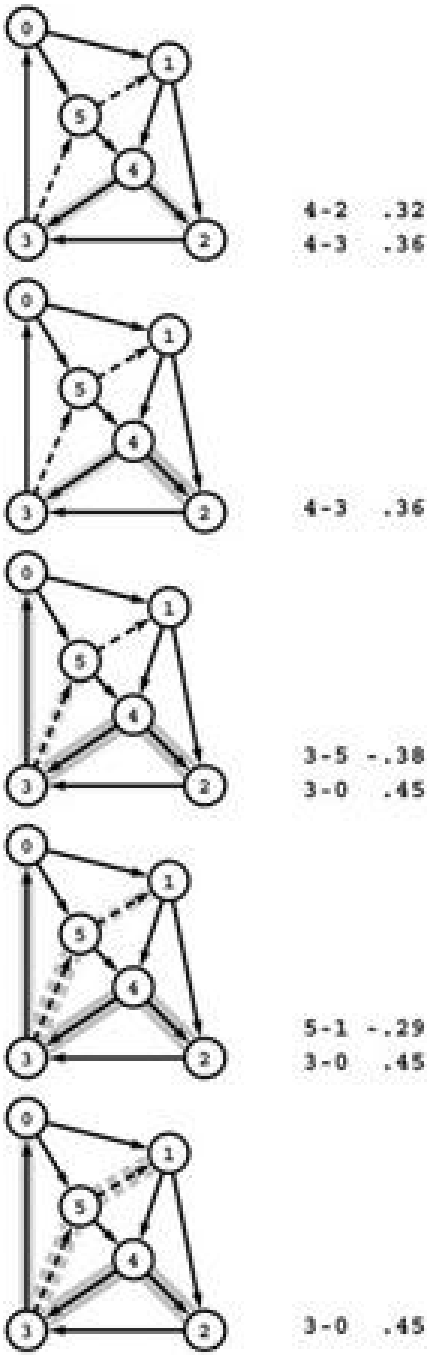
	$	P	Y	C	S
$	1.0	1.631	0.669	0.008	0.686
P	0.613	1.0	0.411	0.005	0.421
Y	1.495	2.436	1.0	0.012	1.027
C	120.5	197.4	80.82	1.0	82.91
S	1.459	2.376	0.973	0.012	1.0

	$	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

Can we detect negative cycles in a network or find shortest paths in networks that contain no negative cycles? The existence of efficient algorithms to solve these problems does not contradict the NP-hardness of the general problem that we proved in Property 21.18, because no reduction from the Hamilton-path problem to either problem is known. Specifically, the reduction of Property 21.18 says that what we cannot do is to craft an algorithm that can guarantee to find efficiently the lowest-weight `path` in any given network when negative edge weights are allowed. That problem statement is too general. But we can solve the restricted versions of the problem just mentioned, albeit not as easily as we can the other restricted versions of the problem (positive weights and acyclic networks) that we studied earlier in this chapter.

In general, as we noted in section 21.2, Dijkstra's algorithm does not work in the presence of negative weights, even when we restrict attention to networks that contain no negative cycles. *Figure* 21.28 illustrates this fact. The fundamental difficulty is that the algorithm depends on examining paths in increasing order of their length. The proof that the algorithm is correct (see Property 21.2) assumes that adding an edge to a `path` makes that `path` longer.

Figure 21.28 Failure of Dijkstra's Algorithm (Negative Weights) *In this example, Dijkstra's algorithm decides that* `4-2` *is the shortest* `path` *from* `4` *to* `2` (*of length .32), and misses the shorter path* `4-3-5-1-2` (*of length .20*).



Floyd's algorithm makes no such assumption and is effective even when edge weights may be negative. If there are no negative cycles, it computes shortest paths; remarkably enough, if there are negative cycles, it detects at least one of them.

Property 21.20 *Floyd's algorithm solves the negative-cycle-detection problem and the all-pairs shortest-paths problem in networks that contain no negative cycles, in time proportional to* V^3.

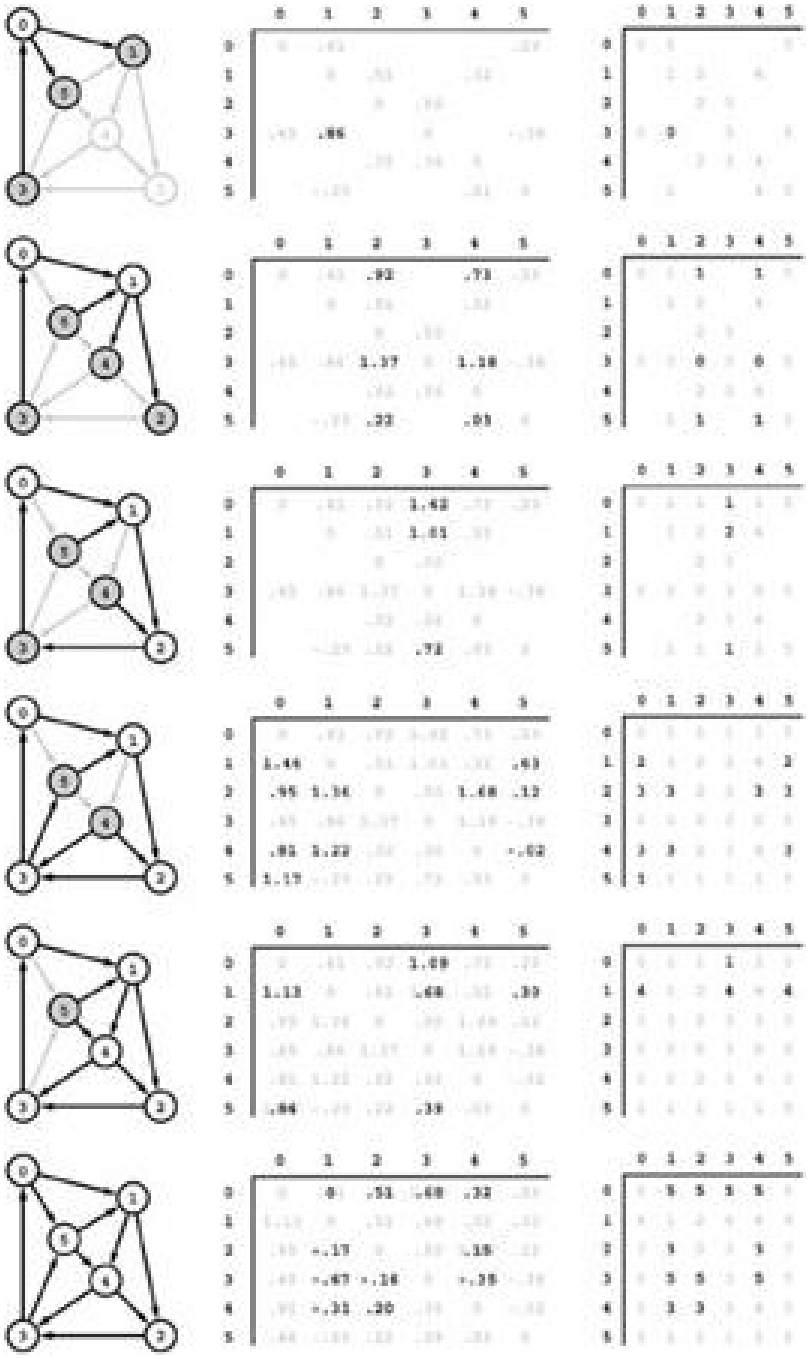
Proof: The proof of Property 21.8 does not depend on whether or not edge weights are negative; however, we need to interpret the results differently when negative edge weights are present. Each entry in the matrix is evidence of the algorithm having discovered a `path` of that length; in particular, any negative entry on the diagonal of the distances matrix is evidence of the presence of at least one negative cycle. In the presence of negative cycles, we cannot directly infer any further information, because the paths that the algorithm implicitly tests are not necessarily simple: Some may involve one or more trips around one or more negative cycles. However, if there are no negative cycles, then the paths that are computed by the algorithm are simple, because any `path` with a cycle would imply the existence of a `path` that has fewer edges and is not of higher weight that connects the same two points (the same `path` with the cycle removed)■

The proof of Property 21.20 does not give specific information about how to find a specific negative cycle from the distances and paths matrices computed by Floyd's algorithm. We leave that task for an exercise (see Exercise 21.122).

Floyd's algorithm solves the all-pairs shortest-paths problem for graphs that contain no negative cycles. Given the failure of Dijkstra's algorithm in networks that contain weights that could be negative, we could also use Floyd's algorithm to solve the the all-pairs problem for sparse networks that contain no negative cycles, in time proportional to V^3. If we have a single-source problem in such networks, then we can use this V^3 solution to the all-pairs problem that, although it amounts to overkill, is the best that we have yet seen for the single-source problem. Can we develop faster algorithms for these problems—ones that achieve the running times that we achieve with Dijkstra's algorithm when edge weights are positive ($E \lg V$ for single-source shortest paths and $V E \lg V$ for all-pairs shortest paths)? We can answer this question in the affirmative for the all-pairs problem, and can bring down the worst-case cost to $V E$ for the single-source problem, but breaking the $V E$ barrier for the general single-source shortest-paths problem is a longstanding open problem.

Figure 21.29 Floyd's Algorithm (Negative Weights) *This sequence shows the construction of the all-shortest paths matrices for a digraph with negative weights, using Floyd's algorithm. The first step are the same as depicted in Figure 21.14. Then the negative edge* `5-1` *comes into play in the second step, where the*

paths `5-1-2` *and* `5-1-4` *are discovered. The algorithm involves precisely the same sequence of relaxation steps for any edge weights, but the outcome differs.*



The following approach, known as the *Bellman–Ford algorithm*, provides a simple and effective basis for attacking single-source shortest-paths problems in networks that contain no negative cycles. To compute shortest paths from a vertex `s`, we maintain (as usual) a vertex-indexed array `wt` such that `wt[t]` contains the shortest-path length from `s` to `t`. We initialize `wt[s]` to `0` and all other `wt` entries to a large sentinel value, then compute shortest paths as follows:

Considering the network's edges in any order, relax along each edge. Make V such passes.

We use the term *Bellman–Ford algorithm* to refer to the generic method of making *V* passes through the edges, considering the edges in any order. Certain authors use the term to describe a more general method (see Exercise 21.130).

Property 21.21 *With the Bellman–Ford algorithm, we can solve the single-source shortest-paths problem in networks that contain no negative cycles in time proportional to V E.*

Proof: Wemake *V* passes through all *E* edges, so the total time is proportional to *V E*. To show that the computation achieves the desired result, we show by induction on `i` that, after the ith pass, `wt[v]` is no greater than the length of the shortest `path` from `s` to `v` that contains `i` or fewer edges (or `maxWT` if there is no such path), for all vertices v. The claim is certainly true if `i` is `0`. Assuming the claim to be true for `i`, there are two cases for each given vertex `v`: Among the paths from `s` to `v` with `i+1` or fewer edges, there may or may not be a shortest `path` with `i+1` edges. If the shortest of the paths with `i+1` or fewer edges from `s` to `v` is of length `i` or less, then `wt[v]` will not change and will remain valid. Otherwise, there is a `path` from `s` to `v` with `i+1` edges that is shorter than any `path` from `s` to `v` with `i` or fewer edges. That `path` must consist of a `path` with `i` edges from `s` to some vertex `w` plus the edge `w-v`. By the induction hypothesis, `wt[w]` is an upper bound on the shortest distance from `s` to `w`, and the `(i+1)`st pass checks whether each edge constitutes the final edge in a new shortest `path` to that edge's destination. In particular, it checks the edge `w-v`.

After `V-1` iterations, then, `wt[v]` is a lower bound on the length of any shortest `path` with `V-1` or fewer edges from `s` to `v`, for all vertices `v`. We can stop after `V-1` iterations because any `path` with `V` or more edges must have a (positive- or zero-cost) cycle and we could find a `path` with `V-1` or fewer edges that is the same length or shorter by removing the cycle. Since `wt[v]` is the length of *some* `path` from `s` to `v`, it is also an upper bound on the shortest-path length, and therefore must be equal to the shortest-path length.

Although we did not consider it explicitly, the same proof shows that the `st` array is a parent-link representation of the shortest-paths tree rooted at `s`■

For example, in a graph represented with adjacency lists, we could implement the Bellman–Ford algorithm to find the shortest paths from a start vertex `s` as follows:

```
    for (v = 0; v < G->V; v++)
```

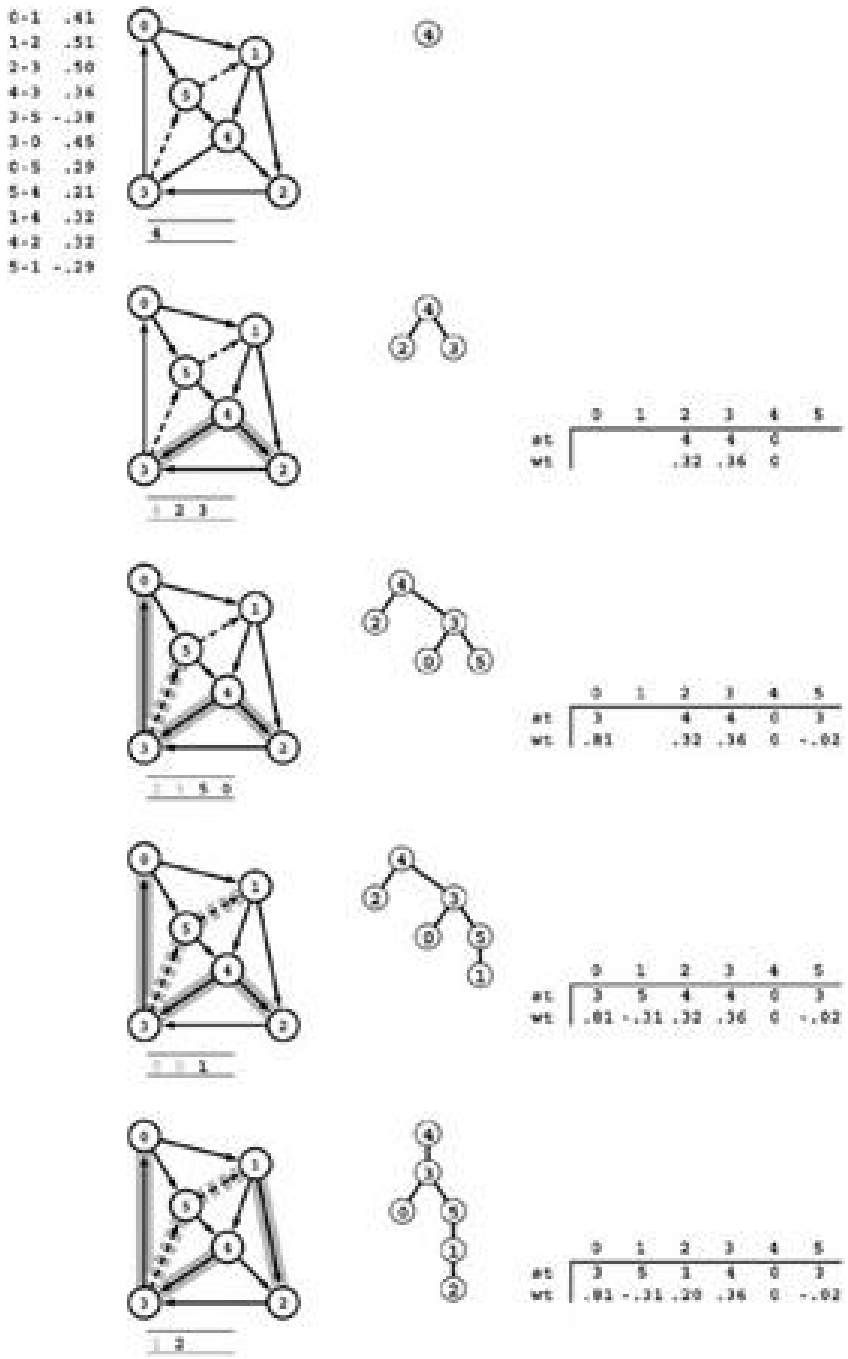
```
      { st[v] = -1; wt[v] = maxWT; }
    wt[s] = 0; st[s] = 0;
    for (i = 0; i < G->V; i++)
      for (v = 0; v < G->V; v++)
       if (wt[v] < maxWT)
         for (t = G->adj[v]; t != NULL; t = t->next)
           if (wt[t->v] > wt[v] + t->wt)
             { wt[t->v] = wt[v] + t->wt; st[t->v] = v; }
```

This code exhibits the simplicity of the basic method. It is not used in practice, however, because simple modifications yield implementations that are more efficient for most graphs, as we soon see.

For typical graphs, examining every edge on every pass is wasteful. Indeed, we can easily determine a priori that numerous edges are not going to lead to a successful relaxation in any given pass. In fact, the only edges that could lead to a change are those emanating from a vertex whose value changed on the previous pass.

Program 21.9 is a straightforward implementation where we use a FIFO queue to hold these edges, so that they are the only ones examined on each pass. *Figure* 21.30 shows an example of this algorithm in operation.

Figure 21.30 Bellman-Ford Algorithm (with Negative Weights) *This figure shows the result of using the Bellman-Ford algorithm to find the shortest paths from vertex* `4` *in the network depicted in Figure 21.26. The algorithm operates in passes, where we examine all edges emanating from all vertices on a FIFO queue. The contents of the queue are shown below each graph drawing, with the shaded entries representing the contents of the queue for the previous pass. When we find an edge that can reduce the length of a* `path` *from* `4` *to its destination, we do a relaxation operation that puts the destination vertex on the queue and the edge on the SPT. The gray edges in the graph drawings comprise the SPT after each stage, which is also shown in oriented form in the center (all edges pointing down). We begin with an empty SPT and* `4` *on the queue (top). In the second pass, we relax along* `4-2` *and* `4-3`*, leaving* `2` *and* `3` *on the queue. In the third pass, we examine but do not relax along* `2-3` *and then relax along* `3-0` *and* `3-5`*, leaving* `0` *and* `5` *on the queue. In the fourth pass, we relax along 5-1 and then examine but do not relax along* `1-0` *and* `1-5`*, leaving* `1` *on the queue. In the last pass (bottom), we relax along* `1-2`*. The algorithm initially operates like BFS, but, unlike all of our other graph search methods, it might change tree edges, as in the last step.*



Program 21.9 Bellman–Ford algorithm

This implementation of the Bellman–Ford algorithm maintains a FIFO queue of all vertices for which relaxing along an outgoing edge could be effective. We take a vertex off the queue and relax along all of its edges. If any of them leads to a shorter `path` to some vertex, we put that on the queue. The sentinel value `G->V` separates

the current batch of vertices (which changed on the last iteration) from the next batch (which change on this iteration), and allows us to stop after `G->V` passes.

```
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; }
}
```

Program 21.9 is effective for solving the single-source shortest-paths problem in networks that arise in practice, but its worst-case performance is still proportional to VE. For dense graphs, the running time is not better than for Floyd's algorithm, which finds all shortest paths, rather than just those from a single source. For sparse graphs, the implementation of the Bellman–Ford algorithm in *Program* 21.9 is up to a factor of V faster than Floyd's algorithm but is nearly a factor of V slower than the worst-case running time that we can achieve with Dijkstra's algorithm for networks with no negative-weight edges (see Table 19.2).

Other variations of the Bellman–Ford algorithm have been studied, some of which are faster for the single-source problem than the FIFO-queue version in *Program* 21.9, but all take time proportional to at least VE in the worst-case (see, for example, Exercise 21.132). The basic Bellman–Ford algorithm was developed decades ago; and, despite the dramatic strides in performance that we have seen for many other graph problems, we have not yet seen algorithms with better worst-case performance for networks with negative weights.

The Bellman–Ford algorithm is also a more efficient method than Floyd's algorithm for detecting whether a network has negative cycles:

Property 21.22 *With the Bellman–Ford algorithm, we can solve the negative-cycle–detection problem in time proportional to VE.*

Proof: The basic induction in the proof of Property 21.21 is valid even in the presence of negative cycles. If we run a `V` th iteration of the algorithm and any relaxation step succeeds, then we have found a shortest `path` with `V` edges that connects `s` to some vertex in the network. Any such `path` must have a cycle (connecting some vertex `w` to itself) and that cycle must be negative, by the inductive hypothesis, since the `path` from `s` to the second occurrence of `w` must be shorter than the `path` from `s` to the first occurrence of `w` for `w` to be included on the `path` the second time. The cycle will also be present in the parent-link array; thus, we could also detect cycles by periodically checking the st links (see Exercise 21.134).

This argument holds for only those vertices that are in the same strongly connected component as the source `s`. To detect negative cycles in general we can either compute the strongly connected components and initialize weights for one vertex in each component to `0` (see Exercise 21.126), or add a dummy vertex with edges to every other vertex (see Exercise 21.127)■

To conclude this section, we consider the all-pairs shortest-paths problem. Can we do better than Floyd's algorithm, which runs in time proportional to V^3? Using the Bellman–Ford algorithm to solve the all-pairs shortest-paths problem by solving the single-source problem at each vertex exposes us to a worst-case running time that is proportional to V^2E. We do not consider this solution in more detail because there is a way to guarantee that we can solve the all-paths problem in time proportional to $VE \log V$. It is based on an idea that we considered at the beginning of this section: transforming the network into a network that has only nonnegative weights and that has the same shortest-paths structure.

In fact, we have a great deal of flexibility in transforming any network to another one with different edge weights but the same shortest paths. Suppose that a vertex-indexed array `wt` contains an arbitrary assignment of weights to the vertices of a network G. With these weights, we define the operation of *reweighting* the graph to be the following:

- To reweight an edge, add to that edge's weight the difference between the weights of the edge's source and destination.

- To reweight a network, reweight all of that network's edges

For example, the following code reweights a network that is represented with adjacency lists, using our standard conventions:

```
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]
```

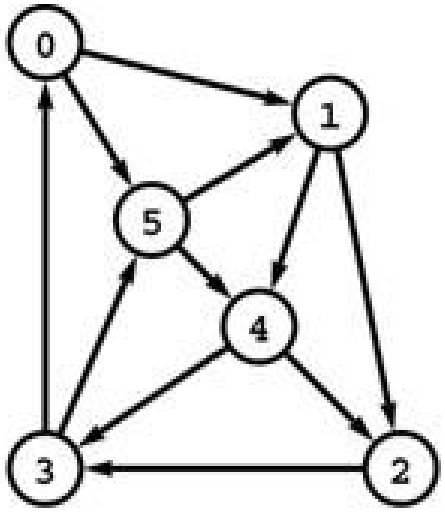
This operation is a simple linear-time process that is well-defined for all networks, regardless of the weights. Remarkably, the shortest paths in the transformed network are the same as the shortest paths in the original network.

Property 21.23 *Reweighting a network does not affect its shortest paths*.

Proof: Given any two vertices `s` and `t`, reweighting changes the weight of any *path* from `s` to `t`, precisely by adding the difference between the weights of `s` and `t`. This fact is easy to prove by induction on the length of the path. The weight of *every* `path` from `s` to `t` is changed by the same amount when we reweight the network, long paths and short paths alike. In particular, this fact implies immediately that the shortest-path length between any two vertices in the transformed network is the same as the shortest-path length between them in the original network■

Since paths between different pairs of vertices are reweighted differently, reweighting could affect questions that involve comparing shortest-path lengths (for example, computing the network's diameter). In such cases, we need to invert the reweighting after completing the shortest-paths computation but before using the result.

Figure 21.31 Reweighting a Network *Given any assignment of weights to vertices (top), we can reweight all of the edges in a network by adding to each edge's weight the difference of the weights of its source and destination vertices. Reweighting does not affect the shortest paths because it makes the same change to the weights of all paths connecting each pair of vertices. For example, consider the path* `0-5-4-2-3`: *its weight is* `.29 + .21 + .32 + .50 = 1.32`; *its weight in the reweighted network is* `1.12 + .19 + .12 + .34 = 1.77`; *these weights differ by* `.45 = .81 - .36`, *the difference of the weights of* `0` *and* `3`; *and the weights of all paths between* `0` *and* `3` *change by this same amount.*



	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
```

Reweighting is no help in networks with negative cycles: the operation does not change the weight of any cycle, so we cannot remove negative cycles by reweighting. But for networks with no negative cycles, we can seek to discover a set of vertex weights such that reweighting leads to edge weights that are nonnegative, no matter what the original edge weights. With nonnegative edge weights, we can then solve the all-pairs shortest-paths problem with the all-pairs version of Dijkstra's algorithm. For example, *Figure* 21.31 gives such an example for our sample network and *Figure* 21.32 shows the shortest-paths computation with Dijkstra's algorithm on the transformed network with no negative edges. The following property shows that we can always find such a set of weights.

Property 21.24 *In any network with no negative cycles, pick any vertex* s *and assign*

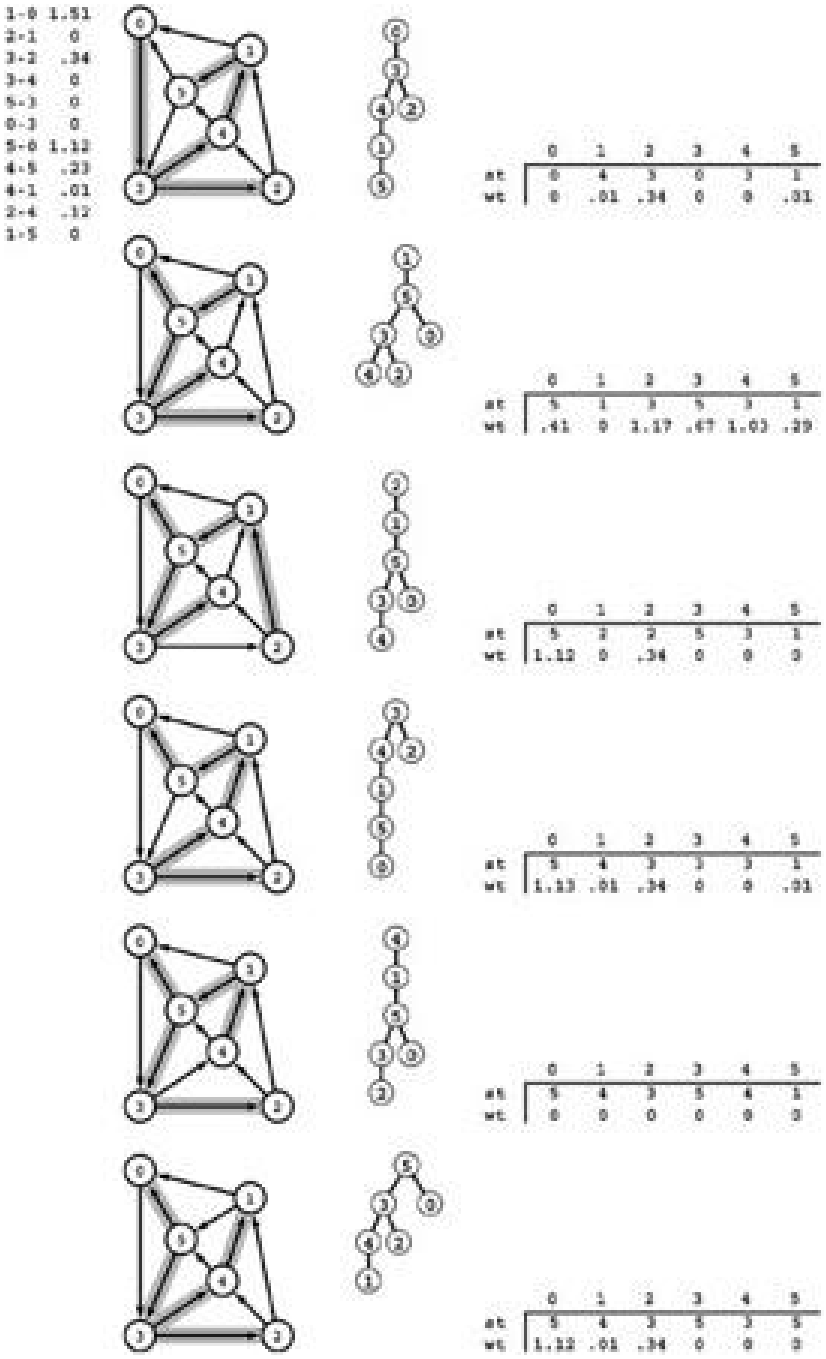
IPR2023-00124
KASPERSKY EX 1036 Page 420

to each vertex v *a weight equal to the length of a shortest* path *to* v *from* s. *Reweighting the network with these vertex weights yields nonnegative edge weights for each edge that connects vertices reachable from* s.

Proof: Given any edge v-w, the weight of v is the length of a shortest path to v, and the weight of w is the length of a shortest path to w. If v-w is the final edge on a shortest path to w, then the difference between the weight of w and the weight of v is precisely the weight of v-w. In other words, reweighting the edge will give a weight of 0. If the shortest path through w does not go through v, then the weight of v plus the weight of v-w must be greater than or equal to the weight of w. In other words, reweighting the edge will give a positive weight■

Just as we did when we used the Bellman–Ford algorithm to detect negative cycles, we have two ways to proceed to make every edge weight nonnegative in an arbitrary network with no negative cycles. Either we can begin with a source from each strongly connected component, or we can add a dummy vertex with an edge of length 0 to every network vertex. In either case, the result is a shortest-paths spanning forest that we can use to assign weights to every vertex (weight of the path from the root to the vertex in its SPT).

Figure 21.32 All Shortest Paths in a Reweighted Network *These diagrams depict the SPTs for each vertex in the reverse of the reweighted network from Figure 21.31, as could be computed with Dijkstra's algorithm to give us shortest paths in the original network in Figure 21.26. The paths are the same as for the network before reweighting, so, as in Figure 21.9, the st vectors in these diagrams are the columns of the paths matrix in Figure 21.26. The wt vectors in this diagram correspond to the columns in the distances matrix, but we have to undo the reweighting for each entry by subtracting the weight of the source vertex and adding the weight of the final vertex in the* path *(see Figure 21.31). For example, from the third row from the bottom here we can see that the shortest* path *from* 0 *to* 3 *is* 0-5-1-4-3 *in both networks, and its length is 1.13 in the reweighted network shown here. Consulting Figure 21.31, we can calculate its length in the original network by subtracting the weight of* 0 *and adding the weight of* 3 *to get the result* 1.13 - .81 + .36 = .68, *the entry in row* 0 *and column* 3 *of the distances matrix in Figure 21.26. All shortest paths to* 4 *in this network are of length* 0 *because we used those paths for reweighting.*



For example, the weight values chosen in *Figure* 21.31 are precisely the lengths of shortest paths from 4, so the edges in the shortest-paths tree rooted at 4 have weight `0` in the reweighted network.

In summary, we can solve the all-pairs shortest-paths problem in networks that contain negative edge weights but no negative cycles by proceeding as follows:

- Apply the Bellman–Ford algorithm to find a shortest-paths forest in the original network.

- If the algorithm detects a negative cycle, report that fact and terminate.

- Reweight the network from the forest.

- Apply the all-pairs version of Dijkstra's algorithm to the reweighted network.

After this computation, the paths matrix gives shortest paths in both networks, and the distances matrix give the `path` lengths in the reweighted network. This series of steps is sometimes known as *John-son's algorithm* (*see reference section*).

Property 21.25 *With Johnson's algorithm, we can solve the all-pairs shortest-paths problem in networks that contain no negative cycles in time proportional to* $V E \log_d V$, *where* $d = 2$ *if* $E < 2V$, *and* $d = E/V$ *otherwise*.

Proof: See Properties 21.22 through 21.24 and the summary in the previous paragraph. The worst-case bound on the running time is immediate from Properties 21.7 and 21.22.

To implement Johnson's algorithm, we combine the implementation of *Program* 21.9, the reweighting code that we gave just before Property 21.23, and the all-pairs shortest-paths implementation of Dijkstra's algorithm in *Program* 21.4 (or, for dense graphs, *Program* 20.3). As noted in the proof of Property 21.22, we have to make appropriate modifications to the Bellman–Ford algorithm for networks that are not strongly connected (see Exercise 21.135 through 21.137). To complete the implementation of the all-pairs shortest-paths interface, we can either compute the true `path` lengths by subtracting the weight of the start vertex and adding the weight of the destination vertex (undoing the reweighting operation for the paths) when copying the two vectors into the distances and paths matrices in Dijkstra's algorithm, or we can put that computation in `GRAPHdist` in the ADT implementation.

For networks with no negative weights, the problem of detecting cycles is easier to solve than is the problem of computing shortest paths from a single source to all other vertices,; the latter problem is easier to solve than is the problem of computing shortest paths connecting all pairs of vertices. These facts match our intuition. By contrast, the analogous facts for networks that contain negative weights are counterintuitive: The algorithms that we have discussed in this section show that, for networks that have negative weights, the best known algorithms for these three problems have similar worst-case performance characteristics. For example, it is nearly as difficult, in the worst case, to determine whether a network has a single negative cycle as it is to find all shortest paths in a network of the same size that has no negative cycles.

Exercises

▷ **21.109** Modify your random-network generators from Exercise 21.5 and 21.6 to generate weights between *a* and *b* (where *a* and *b* are both between -1 and 1), by rescaling.

▷ **21.110** Modify your random-network generators from Exercise 21.5 and 21.6 to generate negative weights, by negating a fixed percentage (whose value is supplied by the client) of the edge weights.

○ **21.111** Develop client programs that use your generators from Exercise 21.109 and 21.110 to produce networks that have a large percentage of negative weights but have at most a few negative cycles, for as large a range of values of *V* and *E* as possible.

21.112 Find a currency-conversion table online or in a newspaper. Use it to build an arbitrage table. *Note*: Avoid tables that are derived (calculated) from a few values and that therefore do not give sufficiently accurate conversion information to be interesting. *Extra credit*: Make a killing in the money-exchange market!

● **21.113** Build a sequence of arbitrage tables using the source for conversion that you found for Exercise 21.112 (any source publishes different tables periodically). Find all the arbitrage opportunities that you can in the tables, and try to find patterns among them. For example, do opportunities persist day after day, or are they fixed quickly after they arise?

21.114 Develop a model for generating random arbitrage problems. Your goal is to generate tables that are as similar as possible to the tables that you used in Exercise 21.113.

21.115 Develop a model for generating random job-scheduling problems that include deadlines. Your goal is to generate nontrivial problems that are likely to be feasible.

21.116 Modify your interface and implementations from Exercise 21.101 to give clients the ability to pose and solve job-scheduling problems that include deadlines, using a reduction to the shortest-paths problem.

○ **21.117** Explain why the following argument is invalid: The shortest-paths problem reduces to the difference-constraints problem by the construction used in the proof of Property 21.15, and the difference-constraints problem reduces trivially to linear programming, so, by Property 21.17, linear programming is NP-hard.

21.118 Does the shortest-paths problem in networks with no negative cycles reduce to the job-scheduling problem with deadlines? (Are the two problems equivalent?) Prove your answer.

○ **21.119** Find the lowest-weight cycle (best arbitrage opportunity) in the example

IPR2023-00124
KASPERSKY EX 1036 Page 424

shown in *Figure* 21.27.

▷ **21.120** Prove that finding the lowest-weight cycle in a network that may have negative edge weights is NP-hard.

▷ **21.121** Show that Dijkstra's algorithm does work correctly for a network in which edges that leave the source are the only edges with negative weights.

▷ **21.122** Develop an ADT implementation based on Floyd's algorithm that provides clients with the capability to test networks for the existence of negative cycles.

21.123 Show, in the style of *Figure* 21.29, the computation of all shortest paths of the network defined in Exercise 21.1, with the weights on edges `5-1` and `4-2` negated, using Floyd's algorithm.

● **21.124** Is Floyd's algorithm optimal for complete networks (networks with V^2 edges)? Prove your answer.

21.125 Show, in the style of Figures 21.30 through 21.32, the computation of all shortest paths of the network defined in Exercise 21.1, with the weights on edges `5-1` and `4-2` negated, using the Bellman–Ford algorithm.

▷ **21.126** Develop an ADT implementation based on the Bellman–Ford algorithm that provides clients with the capability to test networks for the existence of negative cycles, using the method of starting with a source in each strongly connected component.

▷ **21.127** Develop an ADT implementation based on the Bellman–Ford algorithm that provides clients with the capability to test networks for the existence of negative cycles, using a dummy vertex with edges to all the network vertices.

○ **21.128** Give a family of graphs for which *Program* 21.9 takes time proportional to VE to find negative cycles.

▷ **21.129** Show the schedule that is computed by *Program* 21.9 for the job-scheduling-with-deadlines problem in Exercise 21.89.

○ **21.130** Prove that the following generic algorithm solves the single-source shortest-paths problem: "Relax *any* edge; continue until there are no edges that can be relaxed."

21.131 Modify the implementation of the Bellman–Ford algorithm in *Program* 21.9 to use a randomized queue rather than a FIFO queue. (The result of Exercise 21.130 proves

that this method is correct.)

○ **21.132** Modify the implementation of the Bellman–Ford algorithm in *Program* 21.9 to use a deque rather than a FIFO queue, such that edges are put onto the deque according to the following rule: If the edge has previously been on the deque, put it at the beginning (as in a stack); if it is being encountered for the first time, put it at the end (as in a queue).

21.133 Run empirical studies to compare the performance of the implementations in Exercise 21.131 and 21.132 with *Program* 21.9, for various general networks (see Exercise 21.109 through 21.111).

○ **21.134** Modify the implementation of the Bellman–Ford algorithm in *Program* 21.9 to implement a function `GRAPHnegcycle`, which returns the index of any vertex on any negative cycle or `-1` if the network has no negative cycle. When a negative cycle is present, the function should also leave the `st` array such that following links in the array in the normal way (starting with the return value) traces through the cycle.

○ **21.135** Modify the implementation of the Bellman–Ford algorithm in *Program* 21.9 to set vertex weights as required for Johnson's algorithm, using the following method. Each time that the queue empties, scan the `st` array to find a vertex whose weight is not yet set and rerun the algorithm with that vertex as source (to set the weights for all vertices in the same strong component as the new source), continuing until all strong components have been processed.

▷ **21.136** Develop an implementation of the all-pairs shortest-paths ADT interface for the adjacency-lists representation of general networks (based on Johnson's algorithm) by making appropriate modifications to Programs 21.9 and 21.4.

21.137 Develop an implementation of the all-pairs shortest-paths ADT interface for dense networks (based on Johnson's algorithm) (see Exercise 21.136 and 21.43). Run empirical studies to compare your implementation with Floyd's algorithm (*Program* 21.5), for various general networks (see Exercise 21.109 through 21.111).

● **21.138** Add an ADT function to your solution to Exercise 21.137 that allows a client to decrease the cost of an edge. Return a flag that indicates whether that action creates a negative cycle. If it does not, update the paths and distances matrices to reflect any new shortest paths. Your function should take time proportional to V^2.

● **21.139** Implement network ADT functions like the one described in Exercise 21.138 that allow clients to insert and delete edges.

IPR2023-00124
KASPERSKY EX 1036 Page 426

● **21.140** Develop an algorithm that breaks the VE barrier for the single-source shortest-paths problem in general networks, for the special case where the weights are known to be bounded in absolute value by a constant.

21.8 PERSPECTIVE

Table 21.4 summarizes the algorithms that we have discussed in this chapter and gives their worst-case performance characteristics. These algorithms are broadly applicable because, as discussed in section 21.6, shortest-paths problems are related to a large number of other problems in a specific technical sense that directly leads to efficient algorithms for solving the entire class, or at least indicates such algorithms exist.

The general problem of finding shortest paths in networks where edge weights could be negative is intractable. Shortest-paths problems are a good illustration of the fine line that often separates intractable problems from easy ones, since we have numerous algorithms to solve the various versions of the problem when we restrict the edge weights to be positive or acyclic, or even when we restrict to subproblems where there are negative edge weights but no negative cycles. Several of the algorithms are optimal or nearly so, although there are significant gaps between the best known lower bound and the best known algorithm for the single-source problem in networks that contain no negative cycles and for the all-pairs problem in networks that contain nonnegative weights.

The algorithms are all based on a small number of abstract operations and can be cast in a general setting. Specifically, the only operations that we perform on edge weights are addition and comparison: any setting in which these operations make sense can serve as the platform for shortest-paths algorithms. As we have noted, this point of view unifies our algorithms for computing the transitive closure of digraphs with our algorithms for finding shortest paths in networks. The difficulty presented by negative edge weights corresponds to a monotonicity property on these abstract operations: If we can ensure that the sum of two weights is never less than either of the weights, then we can use the algorithms in Sections 21.2 through 21.4; if we cannot make such a guarantee, we have to use the algorithms from section 21.7. Encapsulating these considerations in an ADT is easily done and expands the utility of the algorithms.

Table 21.4 **Costs of shortest-paths algorithms** This table summarizes the cost (worst-case running time) of various shortest-paths algorithms considered in this chapter. The worst-case bounds marked as conservative may not be useful in predicting performance on real networks, particularly the Bellman–Ford algorithm, which typically runs in linear time.

IPR2023-00124
KASPERSKY EX 1036 Page 427

weight constraint	algorithm	cost	comment
single-source			
nonnegative	Dijkstra	V^2	optimal (dense networks)
nonnegative	Dijkstra (PFS)	$E \lg V$	conservative bound
acyclic	source queue	E	optimal
no negative cycles	Bellman–Ford	VE	room for improvement?
none	open	?	NP-hard
all-pairs			
nonnegative	Floyd	V^3	same for all networks
nonnegative	Dijkstra (PFS)	$VE \lg V$	conservative bound
acyclic	DFS	VE	same for all networks
no negative cycles	Floyd	V^3	same for all networks
no negative cycles	Johnson	$VE \lg V$	conservative bound
none	open	?	NP-hard

Shortest-paths problems put us at a crossroads, between elementary graph-processing algorithms on the one hand and problems that we cannot solve on the other. They are the first of several other classes of problems with a similar character that we consider, including *network-flow problems* and *linear programming*. As there is in shortest paths, there is a fine line between easy and intractable problems in those areas. Not only are numerous efficient algorithms available when various restrictions are appropriate, but also there are numerous opportunities where better algorithms have yet to be invented and numerous occasions where we are faced with the certainty of NP-hard problems.

Many such problems were studied in detail as OR problems, before the advent of computers or computer algorithms. Historically, OR has focused on general mathematical and algorithmic models, whereas computer science has focused on specific algorithmic solutions and basic abstractions that can both admit efficient implementations and help to build general solutions. As models from OR and basic algorithmic abstractions from computer science have both been applied to develop implementations on computers that can solve huge practical problems, the line between OR and computer science has blurred in some areas: for example, researchers in both fields seek efficient solutions to problems such as shortest-paths problems. As we address more difficult problems, we draw on classical methods from both fields of research.

IPR2023-00124
KASPERSKY EX 1036 Page 428

Chapter Twenty-Two
Network Flow

GRAPHS, DIGRAPHS, AND networks are not just mathematical abstractions; they are useful in practice because they help us to solve numerous important problems. In this chapter, we extend the network problem-solving model to encompass a dynamic situation where we imagine material flowing through the network, with different costs attached to different routes. These extensions allow us to tackle a surprisingly broad variety of problems with a long list of applications.

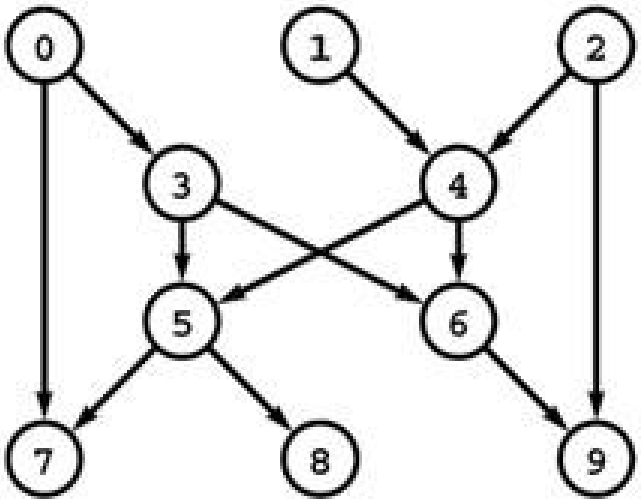
We see that these problems and applications can be handled within a few natural models that we can relate to one another through reduction. There are several different ways, all of which are technically equivalent, to formulate the basic problems. To implement algorithms that solve them all, we settle on two specific problems, develop efficient algorithms to solve them, then develop algorithms that solve other problems by finding reductions to the known problems.

In real life, we do not always have the freedom of choice that this idealized scenario suggests, because not all pairs of reduction relationships between these problems have been proved, and because few optimal algorithms for solving any of the problems are known. Perhaps no efficient direct solution to a given problem has yet been invented, and perhaps no efficient reduction that directly relates a given pair of problems has yet been devised. The network-flow formulation that we cover in this chapter has been successful not only because simple reductions to it are easy to define for many problems, but also because numerous efficient algorithms for solving the basic network-flow problems have been devised.

Figure 22.1 Distribution Problem *In this instance of the distribution problem, we have three supply vertices (0 through 2), four distribution points (3 through 6), three demand vertices (7 through 9), and twelve channels. Each supply vertex has a rate of production; each demand vertex a rate of consumption; and each channel a maximum capacity and a cost per unit distributed. The problem is to minimize costs while distributing material through the channels (without exceeding capacity anywhere) such that the total rate of material leaving each supply vertex equals its rate of production; the total rate at which material arrives at each demand vertex equals its rate of consumption; and the total rate at which material arrives at each distribution point equals the total rate at which material leaves.*

IPR2023-00124
KASPERSKY EX 1036 Page 429

```
supply          channels   cost cap
    0:  4          0-3:      2   2
    1:  4          0-7:      1   3
    2:  6          1-4:      5   5
distribution       2-4:      3   4
    3              2-9:      1   4
    4              3-5:      3   2
    5              3-6:      4   1
    6              4-5:      2   5
demand             4-6:      1   4
    7:  7          5-7:      6   6
    8:  3          5-8:      3   4
    9:  4          6-9:      4   3
```



The following examples illustrate the range of problems that we can handle with network-flow models, algorithms, and implementations. They fall into general categories known as *distribution* problems, *matching* problems, and *cut* problems, each of which we examine in turn. We indicate several different related problems, rather than lay out specific details in these examples. Later in the chapter, when we undertake to develop and implement algorithms, we give rigorous descriptions of many of the problems mentioned here.

In *distribution* problems, we are concerned with moving objects from one place to another within a network. Whether we are distributing hamburger and chicken to fast-food outlets or toys and clothes to discount stores along highways throughout the country—or

IPR2023-00124
KASPERSKY EX 1036 Page 430

software to computers or bits to display screens along communications networks throughout the world—the essential problems are the same. Distribution problems typify the challenges that we face in managing a large and complex operation. Algorithms to solve them are broadly applicable, and are critical in numerous applications.

Merchandise distribution A company has factories, where goods are produced; distribution centers, where the goods are stored temporarily; and retail outlets, where the goods are sold. The company must distribute the goods from factories through distribution centers to retail outlets on a regular basis, using distribution channels that have varying capacities and unit distribution costs. Is it possible to get the goods from the warehouses to the retail outlets such that supply meets demand everywhere? What is the least-cost way to do so? *Figure* 22.1 illustrates a distribution problem.

Figure 22.2 illustrates the *transportation* problem, a special case of the merchandise-distribution problem where we eliminate the distribution centers and the capacities on the channels. This version is important in its own right and is significant (as we see in section 22.7) not just because of important direct applications but also because it turns out not to be a "special case" at all—indeed, it is equivalent in difficulty to the general version of the problem.

Communications A communications network has a set of requests to transmit messages between servers that are connected by channels (abstract wires) that are capable of transferring information at varying rates. What is the maximum rate at which information can be transferred between two specified servers in the network? If there are costs associated with the channels, what is the cheapest way to send the information at a given rate that is less than the maximum?

Traffic flow A city government needs to formulate a plan for evacuating people from the city in an emergency. What is the minimum amount of time that it would take to evacuate the city, if we suppose that we can control traffic flow so as to realize the minimum? Traffic planners also might formulate questions like this when deciding which new roads, bridges, or tunnels might alleviate rush-hour or vacation-weekend traffic problems.

In *matching* problems, the network represents the possible ways to connect pairs of vertices, and our goal is to choose among the connections (according to a specified criterion) without including any vertex twice. In other words, the chosen set of edges defines a way to pair vertices with one another. We might be matching students to colleges, applicants to jobs, courses to available hours for a school, or members of Congress to committee assignments. In each of these situations, we might imagine a variety of criteria defining the characteristics of the matches sought.

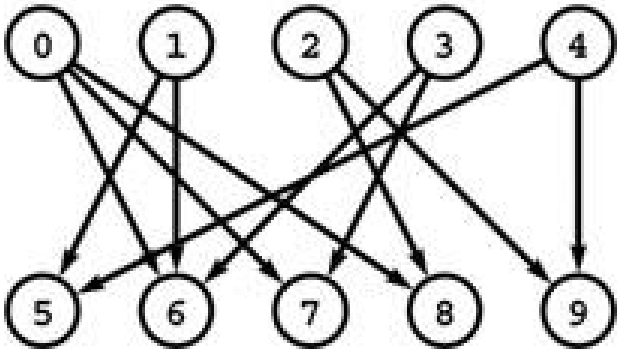
Job placement A job-placement service arranges interviews for a set of students with a set of companies; these interviews result in a set of job offers. Assuming that an interview followed by a job offer represents mutual interest in the student taking a job at the company, it is in everyone's best interests to maximize the number of job placements. *Figure 22.3* is an example illustrating that this task can be complicated.

Minimum-distance point matching Given two sets of N points, find the set of N line segments, each with one endpoint from each of the point sets, with minimum total length. One application of this purely geometric problem is in radar tracking systems. Each sweep of the radar gives a set of points that represent planes. We assume that the planes are kept sufficiently well spaced that solving this problem allows us to associate each plane's position on one sweep to its position on the next, thus giving us the paths of all the planes. Other data-sampling applications can be cast in this framework.

Figure 22.2 Transportation Problem *The transportation problem is like the distribution problem, but with no channel-capacity restrictions and no distribution points. In this instance, we have five supply vertices (0 through 4), five demand vertices (5 through 9), and twelve channels. The problem is to find the lowest-cost way to distribute material through the channels such that supply exactly meets demand everywhere. Specifically, we require an assignment of weights (distribution rates) to the edges such that the sum of weights on outgoing edges equals the supply at each supply vertex; the sum of weights on ingoing edges equals the demand at each demand vertex; and the total cost (sum of weight times cost for all edges) is minimized over all such assignments.*

IPR2023-00124
KASPERSKY EX 1036 Page 432

```
supply          channels   cost
    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
demand             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
```



In *cut* problems, such as the one illustrated in *Figure* 22.4, we remove edges to cut networks into two or more pieces. Cut problems are directly related to fundamental questions of graph connectivity that we first examined in Chapter 18. In this chapter, we discuss a central theorem that demonstrates a surprising connection between cut and flow problems, substantially expanding the reach of network-flow algorithms.

Figure 22.3 Job Placement *Suppose that we have six students, each needing jobs, and six companies, each needing to hire a student. These two lists (one sorted by student, the other sorted by company) give a list of job offers, which indicate mutual interest in matching students and jobs. Is there some way to match students to jobs so that every job is filled and every student gets a job? If not, what is the maximum number of jobs that can be filled?*

IPR2023-00124
KASPERSKY EX 1036 Page 433

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

Network reliability A simplified model considers a telephone network as consisting of a set of wires that connect telephones through switches, such that there is the possibility of a switched `path` through trunk lines connecting any two given telephones. What is the minimum number of trunk lines that could be cut without any pair of switches being disconnected?

Cutting supply lines A country at war moves supplies from depots to troops along an interconnected highway system. An enemy can cut off the troops from the supplies by bombing roads, with the number of bombs required to destroy a road proportional to that road's width. What is the minimum number of bombs that the enemy must drop to ensure that no troops can get supplies?

Each of the applications just cited immediately suggests numerous related questions, and there are still other related models, such as the job-scheduling problems that we considered in Chapter 21. We consider further examples throughout this chapter, yet still treat only a small fraction of the important, directly related practical problems.

The network-flow model that we consider in this chapter is important not just because it provides us with two simply stated problems to which many of the practical problems re-

duce, but also because we have efficient algorithms for solving the two problems. This breadth of applicability has led to the development of numerous algorithms and implementations. The solutions that we consider illustrate the tension between our quest for implementations of general applicability and our quest for efficient solutions to specific problems. The study of network-flow algorithms is fascinating because it brings us tantalizingly close to compact and elegant implementations that achieve both goals.

We consider two particular problems within the network-flow model: the *maxflow* problem and the *mincost-flow* problem. We see specific relationships among these problem-solving models, the shortest-path model of Chapter 21, the linear-programming (LP) model of Part 8, and numerous specific problem models including some of those just discussed.

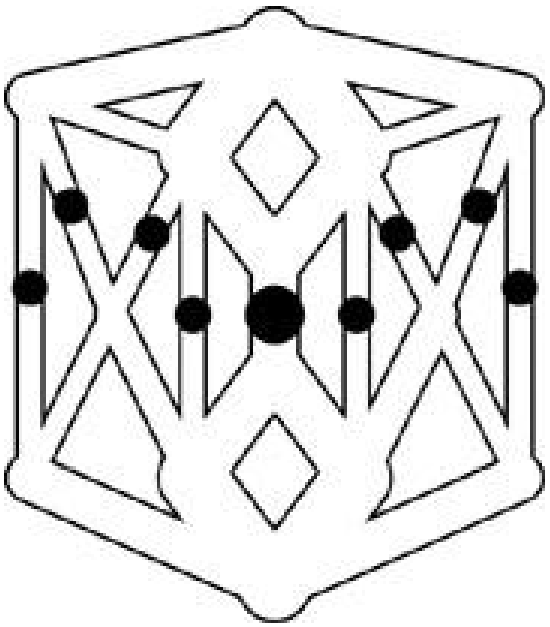
At first blush, many of these problems might seem to be completely different from network-flow problems. Determining a given problem's relationship to known problems is often the most important step in developing a solution to that problem. Moreover, this step is often significant because, as is usual with graph algorithms, we must understand the fine line between trivial and intractable problems before we attempt to develop implementations. The infrastructure of problems and the relationships among the problems that we consider in this chapter provides a helpful context for addressing such issues.

In the rough categorization that we began with in Chapter 17, the algorithms that we examine in this chapter demonstrate that network-flow problems are "easy," because we have straightforward implementations that are guaranteed to run in time proportional to a polynomial in the size of the network. Other implementations, although not guaranteed to run in polynomial time in the worst case, are compact and elegant, and have been proved to solve a broad variety of other practical problems, such as the ones discussed here. We consider them in detail because of their utility. Researchers still seek faster algorithms for these problems, to enable huge applications and to save costs in critical ones. Ideal optimal algorithms that are guaranteed to be as fast as possible are yet to be discovered for network-flow problems.

On the one hand, some of the problems that we reduce to network-flow problems are known to be easier to solve with specialized algorithms. In principle, we might consider implementing and improving these specialized algorithms. Although that approach is productive in some situations, efficient algorithms for solving many of the problems (other than through reduction to network flow) are not known. Even when specialized algorithms are known, developing implementations that can outperform good network-flow codes can be a significant challenge. Moreover, researchers are still improving network-flow algorithms, and there remains the possibility that a good network-flow algorithm might outperform known specialized methods for a given practical problem.

IPR2023-00124
KASPERSKY EX 1036 Page 435

Figure 22.4 Cutting Supply Lines *This diagram represents the roads connecting an army's supply depot at the top to the troops at the bottom. The black dots represent an enemy bombing plan that would separate troops from supplies. The enemy's goal is to minimize the cost of bombing (perhaps assuming that the cost of cutting an edge is proportional to its width), and the army's goal is to design its road network to maximize the enemy's minimum cost. The same model is useful in improving the reliability of communications networks and many other applications.*



On the other hand, network-flow problems are special cases of the even more general LP problems that we discuss in Part 8. Although we could (and people often do) use an algorithm that solves LP problems to solve network-flow problems, the network-flow algorithms that we consider are simpler and more efficient than are those that solve LP problems. But researchers are still improving LP solvers, and there remains the possibility that a good algorithm for LP problems might—when used for practical network-flow problems—outperform all the algorithms that we consider in this chapter.

The classical solutions to network-flow problems are closely related to the other graph algorithms that we have been examining, and we can write surprisingly concise programs that solve them, using the algorithmic tools we have developed. As we have seen in many other situations, good algorithms and data structures can achieve substantial reductions in running times. Development of better implementations of classical generic algorithms is still being studied, and new approaches continue to be discovered.

In section 22.1 we consider basic properties of *flow networks*, where we interpret a network's edge weights as *capacities* and consider properties of *flows*, which are a second set of edge weights that satisfy certain natural constraints. Next, we consider the *maxflow* problem, which is to compute a flow that is best in a specific technical sense. In Sections

22.2 and 22.3, we consider two approaches to solving the maxflow problem, and examine a variety of implementations. Many of the algorithms and data structures that we have considered are directly relevant to the development of efficient solutions of the maxflow problem. We do not yet have the best possible algorithms to solve the maxflow problem, but we consider specific useful implementations. In section 22.4, to illustrate the reach of the maxflow problem, we consider different formulations, as well as other reductions involving other problems.

Maxflow algorithms and implementations prepare us to discuss an even more important and general model known as the *mincostflow problem*, where we assign *costs* (another set of edge weights) and define flow costs, then look for a solution to the maxflow problem that is of minimal cost. We consider a classic generic solution to the mincost-flow problem known as the *cycle-canceling* algorithm; then, in section 22.6, we give a particular implementation of the cycle-canceling algorithm known as the *network simplex* algorithm. In section 22.7, we discuss reductions to the mincost-flow problem that encompass, among others, all the applications that we just outlined.

Network-flow algorithms are an appropriate topic to conclude this book for several reasons. They represent a payoff on our investment in learning basic algorithmic tools such as linked lists, priority queues, and general graph-search methods. The ADT implementations that we have learned lead immediately to compact and efficient ADT implementations for network-flow problems. These implementations take us to a new level of problem-solving power and are immediately useful in numerous practical applications. Furthermore, studying their applicability and understanding their limitations sets the context for our examination of better algorithms and harder problems—the undertaking of Part 8.

22.1 FLOW NETWORKS

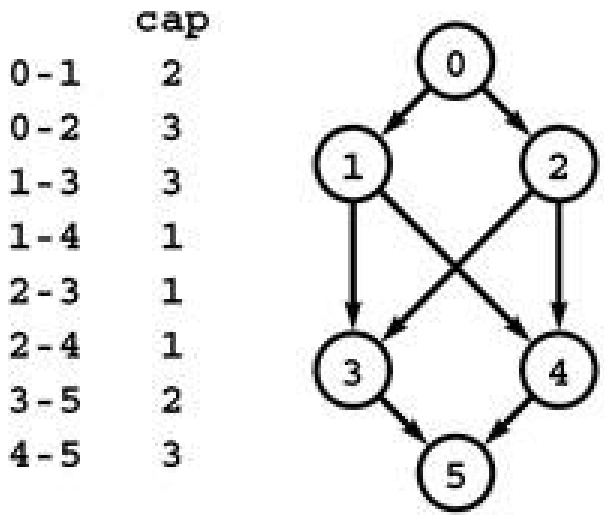
To describe network-flow algorithms, we begin with an idealized physical model in which several of the basic concepts are intuitive. Specifically, we imagine a collection of interconnected oil pipes of varying sizes, with switches controlling the direction of flow at junctions, as in the example illustrated in *Figure* 22.5. We suppose further that the network has a single source (say, an oil field) and a single sink (say, a large refinery) to which all the pipes ultimately connect. At each vertex, the flowing oil reaches an equilibrium where the amount of oil flowing in is equal to the amount flowing out. We measure both flow and pipe capacity in the same units (say, gallons per second).

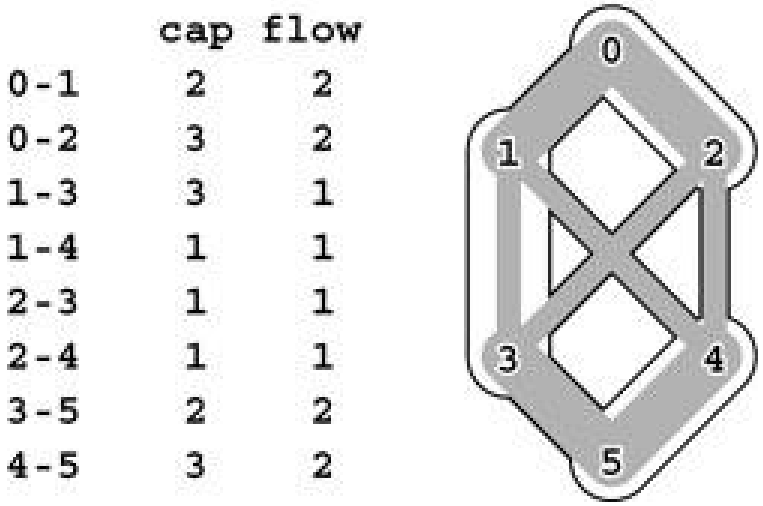
If every switch has the property that the total capacity of the ingoing pipes is equal to the total capacity of the outgoing pipes, then there is no problem to solve: We simply fill all

IPR2023-00124
KASPERSKY EX 1036 Page 437

pipes to full capacity. Otherwise, not all pipes are full, but oil flows through the network, controlled by switch settings at the junctions, such that the amount of oil flowing into each junction is equal to the amount of oil flowing out. But this local equilibrium at the junctions implies an equilibrium in the network as a whole: We prove in Property 22.1 that the amount of oil flowing into the sink is equal to the amount flowing out of the source. Moreover, as illustrated in *Figure* 22.6, the switch settings at the junctions of this amount of flow from source to sink have nontrivial effects on the flow through the network. Given these facts, we are interested in the following question: What switch settings will maximize the amount of oil flowing from source to sink?

Figure 22.5 Network Flow *A flow network is a weighted network where we interpret edge weights as capacities (top). Our objective is to compute a second set of edge weights, bounded by the capacities, which we call the flow. The bottom drawing illustrates our conventions for drawing flow networks. Each edge's width is proportional to its capacity; the amount of flow in each edge is shaded in gray; the flow is always directed down the page from a single source at the top to a single sink at the bottom; and intersections (such as* `1-4` *and* `2-3` *in this example) do not represent vertices unless labelled as such. Except for the source and the sink, flow in is equal to flow out at every vertex: for example, vertex 2 has 2 units of flow coming in (from 0) and 2 units of flow going out (1 unit to 3 and 1 unit to 4).*



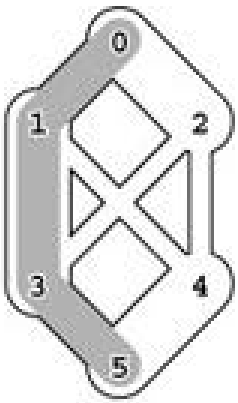


We can model this situation directly with a network (a weighted digraph, as defined in Chapter 21) that has a single source and a single sink. The edges in the network correspond to the oil pipes, the vertices correspond to the junctions with switches that control how much oil goes into each outgoing edge, and the weights on the edges correspond to the capacity of the pipes. We assume that the edges are directed, specifying that oil can flow in only one direction in each pipe. Each pipe has a certain amount of flow, which is less than or equal to its capacity, and every vertex satisfies the equilibrium condition that the flow in is equal to the flow out.

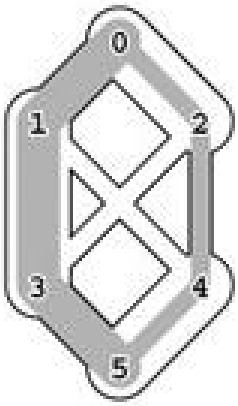
Figure 22.6 Controlling Flow in a Network *We might initialize the flow in this network by opening the switches along the* `path` `0-1-3-5`*, which can handle 2 units of flow (top), and by opening switches along the* `path` `0-2-4-5` *to get another 1 unit of flow in the network (center). Asterisks indicate full edges. Since* `0-1`*,* `2-4`*, and* `3-5` *are full, there is no direct way to get more flow from 0 to 5, but if we change the switch at 1 to redirect enough flow to fill* `1-4`*, we open up enough capacity in* `3-5` *to allow us to add flow on* `0-2-3-5`*, giving a maxflow for this network (bottom).*

IPR2023-00124
KASPERSKY EX 1036 Page 439

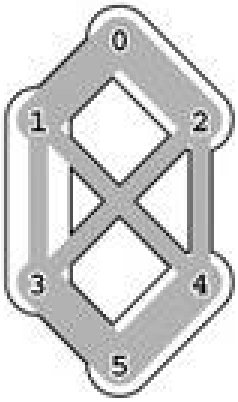
	cap	flow
0-1	2	2*
0-2	3	0
1-3	3	2
1-4	1	0
2-3	1	0
2-4	1	0
3-5	2	2*
4-5	3	0



	cap	flow
0-1	2	2*
0-2	3	1
1-3	3	2
1-4	1	0
2-3	1	0
2-4	1	1*
3-5	2	2*
4-5	3	1



	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



This flow-network abstraction is a useful problem-solving model that applies directly to a variety of applications and indirectly to still more. We sometimes appeal to the idea of oil flowing through pipes for intuitive support of basic ideas, but our discussion applies equally well to goods moving through distribution channels and to numerous other situations.

The flow model directly applies to a distribution scenario: we interpret the flow values as rates of flow, so that a flow network describes the flow of goods in a manner precisely analogous to the flow of oil. For example, we can interpret the flow in *Figure* 22.5 as specifying that we should be sending two items per time unit from 0 to 1 and from 0 to 2,

one item per time unit from 1 to 3 and from 1 to 4, and so forth.

Another way to interpret the flow model for a distribution scenario is to interpret flow values as amounts of goods, so that a flow network describes a one-time transfer of goods. For example, we can interpret the flow in *Figure* 22.5 as describing the transfer of four items from 0 to 5 in the following three-step process: First, send two items from 0 to 1 and two items from 0 to 2, leaving two items at each of those vertices. Second, send one item each from 1 to 3, 1 to 4, 2 to 3, and 2 to 4, leaving two items each at 3 and 4. Third, complete the transfer by sending two items from 3 to 5 and two items from 4 to 5.

As with our use of distance in shortest-paths algorithms, we are free to abandon any physical intuition when convenient because all the definitions, properties, and algorithms that we consider are based entirely on an abstract model that does not necessarily obey physical laws. Indeed, a prime reason for our interest in the network-flow model is that it allows us to solve numerous other problems through reduction, as we see in Sections 22.4 and 22.6. Because of this broad applicability, it is worthwhile to consider precise statements of the terms and concepts that we have just informally introduced.

Definition 22.1 *We refer to a network with a designated source s and a designated sink t as an* **st-network**.

We use the modifier "designated" here to mean that *s* does not necessarily have to be a source (vertex with no incoming edges) and *t* does not necessarily have to be a sink (vertex with no outgoing edges), but that we nonetheless treat them as such, because our discussion (and our algorithms) will ignore edges directed into *s* and edges directed out of *t*. To avoid confusion, we use networks with a single source and a single sink in examples; we consider more general situations in section 22.4. We refer to *s* and *t* as "the source" and "the sink," respectively, in the *st*-network because those are the roles that they play in the network. We also refer to the other vertices in the network as the *internal* vertices.

Definition 22.2 *A* **flow network** *is an st-network with positive edge weights, which we refer to as* **capacities**. *A* **flow** *in a flow network is a set of nonnegative edge weights —which we refer to as* **edge flows***—satisfying the conditions that no edge's flow is greater than that edge's capacity and that the total flow into each internal vertex is equal to the total flow out of that vertex.*

We refer to the total flow into a vertex (the sum of the flows on its incoming edges) as the vertex's *inflow* and the total flow out of a vertex (the sum of the flows on its outgoing edges) as the vertex's *outflow*. By convention, we set the flow on edges into the source and edges out of the sink to zero, and in Property 22.1 we prove that the source's outflow is always equal to the sink's inflow, which we refer to as the network's *value*. With these

definitions, the formal statement of our basic problem is straightforward.

Maximum flow Given an *st*-network, find a flow such that no other flow from *s* to *t* has larger value. For brevity, we refer to such a flow as a *maxflow* and the problem of finding one in a network as the *maxflow problem*. In some applications, we might be content to know just the maxflow value, but we generally want to know a flow (edge flow values) that achieves that value.

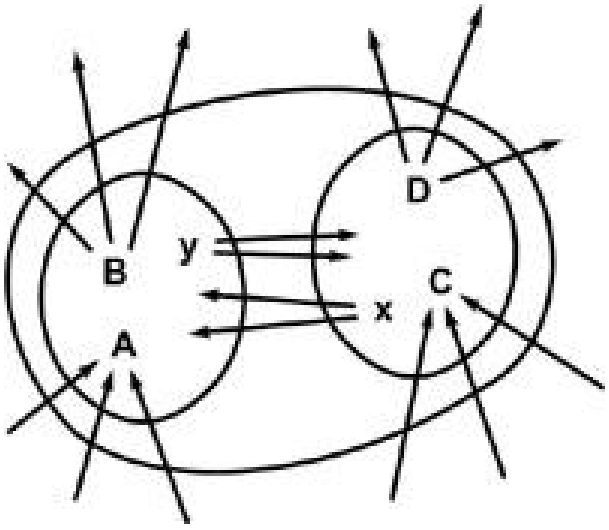
Variations on the problem immediately come to mind. Can we allow multiple sources and sinks? Should we be able to handle networks with no sources or sinks? Can we allow flow in either direction in the edges? Can we have capacity restrictions for the vertices instead of or in addition to the restrictions for the edges? As is typical with graph algorithms, separating restrictions that are trivial to handle from those that have profound implications can be a challenge. We investigate this challenge, and give examples of reducing to maxflow a variety of problems that seem different in character, after we consider algorithms to solve the basic problem, in Sections 22.2 and 22.3.

Figure 22.7 Flow Equilibrium *This diagram illustrates the preservation of flow equilibrium when we merge sets of vertices. The two smaller figures represent any two disjoint sets of vertices, and the letters represent flow in sets of edges as indicated: A is the amount of flow into the set on the left from outside the set on the right,x is the amount of flow into the set on the left from the set on the right, and so forth. Now, if we have flow equilibrium in the two sets, then we must have*

$A + x = B + y$ for the set on the left and

$C + y = D + x$ for the set on the right. Adding these two equations and cancelling the $x + y$ terms, we conclude that

$A + C = B + D$, or inflow is equal to outflow for the union of the two sets.



The characteristic property of flows is the local equilibrium condition that inflow be equal to outflow at each internal vertex. There is no such constraint on capacities; indeed, the imbalance between total capacity of incoming edges and total capacity of outgoing edges is what characterizes the maxflow problem. The equilibrium constraint has to hold at each and every internal vertex, and it turns out that this local property determines global movement through the network, as well. Although this idea is intuitive, it needs to be proved:

Property 22.1 *Any st-flow has the property that outflow from s is equal to the inflow to t.*

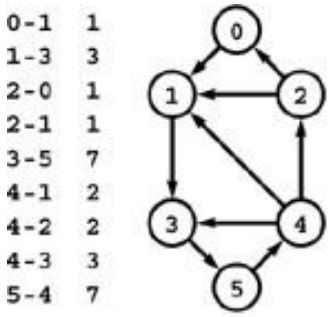
Proof: (We use the term *st-flow* to mean “flow in an *st*-network.") Augment the network with an edge from a dummy vertex into *s*, with flow and capacity equal to the outflow from *s*, and with an edge from *t* to another dummy vertex, with flow and capacity equal to the inflow to *t*. Then, we can prove a more general property by induction: Inflow is equal to outflow for any *set* of vertices (not including the dummy vertices).

This property is true for any single vertex, by local equilibrium. Now, assume that it is true for a given set of vertices S and that we add a single vertex v to make the set $S' = S \cup \{v\}$. To compute inflow and outflow for S', note that each edge from v to some vertex in S reduces outflow (from v) by the same amount as it reduces inflow (to S); each edge to v from some vertex in S reduces inflow (to v) by the same amount as it reduces outflow (from S); and all other edges provide inflow or outflow for S' if and only if they do so for S or v. Thus, inflow and outflow are equal for S', and the value of the flow is equal to the sum of the values of the flows of v and S minus sum of the flows on the edges connecting v to a vertex in S (in either direction).

Applying this property to the set of all the network’s vertices, we find that the source’s inflow from its associated dummy vertex (which is equal to the source’s outflow) is equal to the sink’s outflow to its associated dummy vertex (which is equal to the sink’s inflow).■

Figure 22.8 Cycle Flow Representation *This figure demonstrates that the circulation at left decomposes into the four cycles* `1-3-5-4-1`, `0-1-3-5-4-2-0`, `1-3-5-4-2-1`, `3-5-4-3`, *with weights 2, 1, 1, and 3, respectively. Each cycle’s edges appear in its respective column, and summing each edge’s weight from each cycle in which it appears (across its respective row) gives its weight in the circulation.*

IPR2023-00124
KASPERSKY EX 1036 Page 443



Corollary *The value of the flow for the union of two sets of vertices is equal to the sum of the values of the flows for the two sets minus the sum of the weights of the edges that connect a vertex in one to a vertex in the other*.

Proof: The proof just given for a set *S* and a vertex *v* still works if we replace *v* by a set *T* (which is disjoint from *S*) in the proof. An example of this property is illustrated in *Figure* 22.7.■

We can dispense with the dummy vertices in the proof of Property 22.1, augment any flow network with an edge from t to s with flow and capacity equal to the network's value, and know that inflow is equal to outflow for any set of nodes in the augmented network. Such a flow is called a *circulation*, and this construction demonstrates that the maxflow problem reduces to the problem of finding a circulation that maximizes the flow along a given edge. This formulation simplifies our discussion in some situations. For example, it leads to an interesting alternate representation of flows as a set of cycles, as illustrated in *Figure* 22.8.

Given a set of cycles and a flow value for each cycle, it is easy to compute the corresponding circulation, by following through each cycle and adding the indicated flow value to each edge. The converse property is more surprising: We can find a set of cycles (with a flow value for each) that is equivalent to any given circulation.

Property 22.2 (Flow decomposition theorem) *Any circulation can be represented as flow along a set of at most E directed cycles*.

Proof: A simple algorithm establishes this result. Iterate the following process as long as there is any edge that has flow: Starting with any edge that has flow, follow any edge leaving that edge's destination vertex that has flow and continue until encountering a vertex that has already been visited (a cycle has been detected). Go back around the cycle to find an edge with minimal flow; then reduce the flow on every edge in the cycle by that amount. Each iteration of this process reduces the flow on at least one edge to 0, so there are at most *E* cycles.■

Figure 22.9 Cycle Flow Decomposition Process *To decompose any circulation into a set of cycles, we iterate the following process: follow any* `path` *until encountering a node for the second time, then find the minimum weight on the indicated cycle, then subtract that weight from each edge on the cycle and remove any edge whose weight becomes 0. For example, the first iteration is to follow the* `path` `0-1-3-5-4-1` *to find the cycle* `1-3-5-4-1`*, then subtract 1 from the weights of each of the edges on the cycle, which causes us to remove* `4-1` *because its weight becomes 0. In the second iteration, we remove* `0-1` *and* `2-0`*; in the third iteration, we remove* `1-3`*,* `4-2`*, and* `2-1`*; and in the fourth iteration, we remove* `3-5`*,* `5-4`*, and* `4-3`*.*

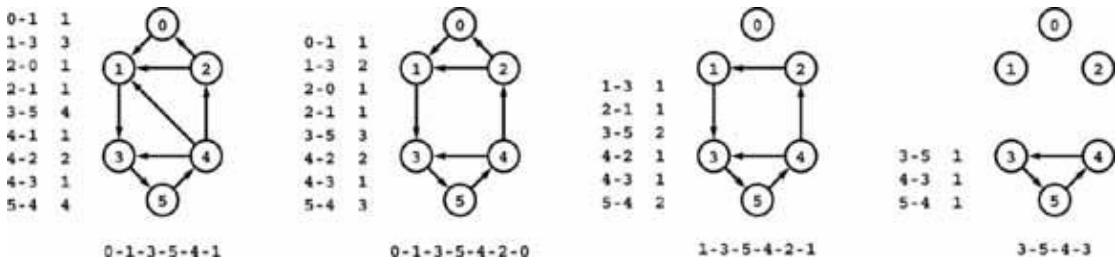


Figure 22.9 illustrates the process described in the proof. For *st*-flows, applying this property to the circulation created by the addition of an edge from t to s gives the result that any *st*-flow can be represented as flow along a set of at most *E* directed paths, each of which is either a `path` from *s* to *t* or a cycle.

Corollary *Any st-network has a maxflow such that the subgraph induced by nonzero flow values is acyclic.*

Proof: Cycles that do not contain `t-s` do not contribute to the value of the flow, so we can change the flow to 0 along any such cycle without changing the value of the flow.■

Corollary *Any st-network has a maxflow that can be represented as flow along a set of at most E directed paths from s to t.*

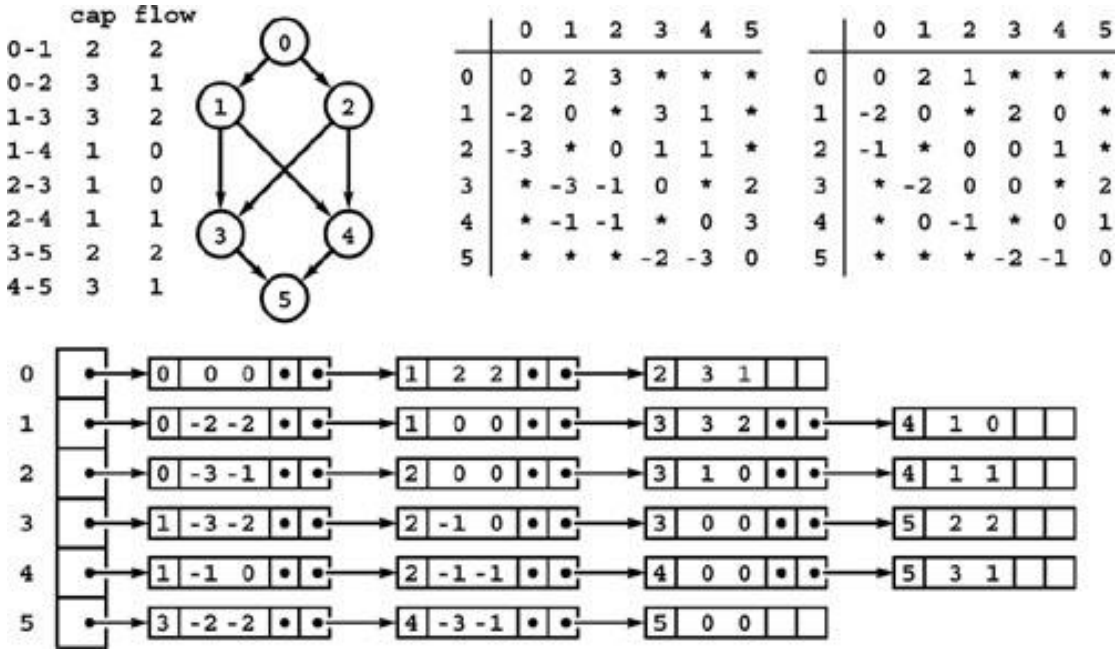
Proof: Immediate.■

This representation provides a useful insight into the nature of flows that is helpful in the design and analysis of maxflow algorithms.

On the one hand, we might consider a more general formulation of the maxflow problem where we allow for multiple sources and sinks. Doing so would allow our algorithms to be used for a broader range of applications. On the other hand, we might consider special cases, such as restricting attention to acyclic networks. Doing so might make the problem easier to solve. In fact, as we see in section 22.4, these variants are equivalent in difficulty to the version that we are considering. Therefore, in the first case, we can adapt our algorithms and implementations to the broader range of applications; in the second case, we

cannot expect an easier solution. In our figures, we use acyclic networks because the examples are easier to understand when they have an implicit flow direction (down the page), but our implementations allow networks with cycles.

Figure 22.10 Flow-Network Representations *In the adjacency-array representation of a flow network, we use parallel arrays (top): one for capacities (left) and one for flow values (right). We represent an edge s-t of capacity c by putting c in row s and column t and -c in row t and column s of the array on the left. We use an identical scheme for flow values in the array on the right. Including both representations in this way simplifies our code because our algorithms need to traverse edges in both directions. We use a similar scheme in the adjacency-lists representation (right). In order to make it possible to change flow values in constant time, links connecting the two representations of each edge are needed. These links are omitted in this diagram.*



To implement maxflow algorithms, we use standard natural representations of flow networks that we derive by extending either the adjacency-matrix or the adjacency-list representation that we used in previous chapters. Instead of the single weight that we used in Chapters 20 and 21, we associate two weights with each edge, cap (capacity) and flow. Even though networks are directed graphs, the algorithms that we examine need to traverse edges in the both directions, so we use a representation like the one that we use for undirected graphs: If there is an edge from *x* to *y* with capacity c and flow f, we also keep an edge from *y* to *x* with capacity -c and flow -f. *Figure* 22.10 shows an adjacency-matrix representation of the example in *Figure* 22.5.

In the network representations of Chapters 20 and 21, we used the convention that weights are real numbers between 0 and 1. In this chapter, we assume that the weights (capacities and flows) are all *m*-bit integers (between 0 and $2^m - 1$). We do so for two pri-

IPR2023-00124
KASPERSKY EX 1036 Page 446

mary reasons. First, we frequently need to test for equality among linear combinations of weights, and doing so can be inconvenient in floating-point representations. Second, the running times of our algorithms can depend on the relative values of the weights, and the parameter $M = 2^m$ gives us a convenient way to bound weight values. For example, the ratio of the largest weight to the smallest nonzero weight is less than M. The use of integer weights is but one of many possible alternatives (see, for example, Exercise 20.8) that we could choose to address these problems.

Program 22.1 Flow-network ADT implementation

This ADT implementation for flow networks extends the adjacency-lists implementation for undirected graphs from Chapter 17 by adding to each list node a field `cap` for edge capacities, `flow` for edge flow, and dup to link the two representations of each edge. Appropriate extensions to other graph-processing code are assumed (for example, the `Edge` type and `EDGE` constructor must have `cap` and `flow` fields). The flow fields are initially all `0`; our network-flow implementations are cast as ADT functions that compute appropriate values for them.

```
#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++;
      }
```

Program 22.2 Flow check and value computation

This ADT function returns 0 if ingoing flow is not equal to outgoing flow at some internal node or if some flow value is negative; the flow value otherwise. Despite our confidence as mathematicians in Property 22.1, our paranoia as programmers dictates that we also check that the flow out of the source is equal to the flow into the sink.

```
    static int flowV (Graph G, int v)
      { link t; intx=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;
      }
```

We use an adjacency-lists representation for most of our programs, for best efficiency with sparse networks, which are most common in applications. Extending the weighted-network representation of Chapter 21, we keep capacities and flows in the list nodes that correspond to edges. As with the adjacency-matrix representation, we keep both a for-

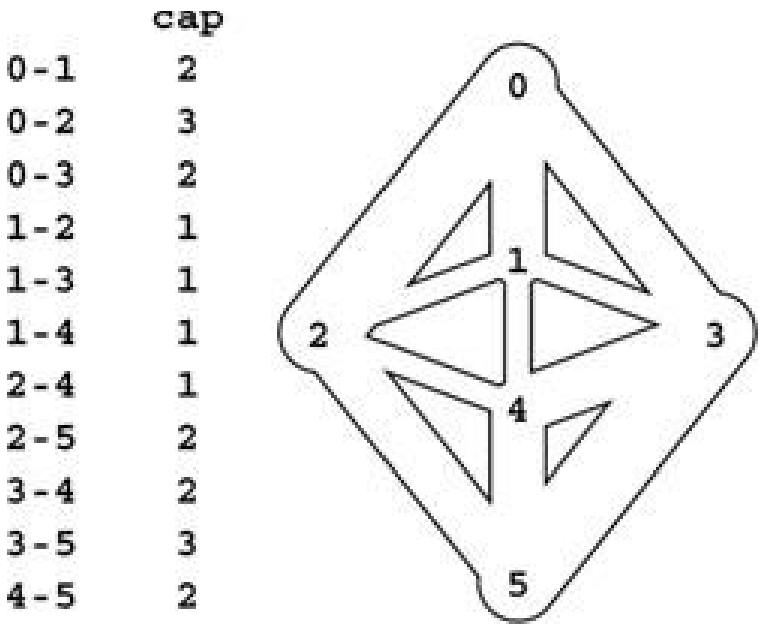
ward and a backward representation of each edge, in a manner similar to our undirected-graph representation of Chapter 17, but with the capacity and flow negated in the backward representation. To avoid excessive list traversals, we also maintain links connecting the two list nodes that represent each edge, so that, when we change the flow in one, we can update it in the other. *Figure* 22.10 shows an adjacency-lists representation of the example in *Figure* 22.5. The extra fields and links make the data structure seem more complicated than it is: The implementation in *Program* 22.1 differs in only a few lines of code from its counterpart for undirected graphs in *Program* 17.4.

As in previous chapters, we often use, for convenience, an EDGE constructor function similar to the one defined in *Program* 17.1, but modified to include flow and capacity.

We sometimes refer to edges as having infinite capacity, or, equivalently, as being *uncapacitated*. That might mean that we do not compare flow against capacity for such edges, or we might use a sentinel value that is guaranteed to be larger than any flow value.

Program 22.2 is an ADT function for checking whether a flow satisfies the equilibrium condition at every node, and returning that flow's value if the flow does. Typically, we might include a call to this function as the final action of a maxflow algorithm. It might also be prudent to check that no edge's flow exceeds that edge's capacity and that the data structures are internally consistent (see Exercise 22.13).

Figure 22.11 Flow Network for Exercises *This flow Network is the Subject of Several Exercises Throughout the Chapter.*



Exercises

▷ **22.1** Find two different maxflows in the flow network shown in *Figure* 22.11.

22.2 Under our assumption that capacities are positive integers less than M, what is the maximum possible flow value for any st-network with V vertices and E edges? Give two answers, depending on whether or not parallel edges are allowed.

▷ **22.3** Give an algorithm to solve the maxflow problem for the case that the network forms a tree if the sink is removed.

○ **22.4** Give a family of networks with E edges having circulations where the process described in the proof of Property 22.2 produces E cycles.

22.5 Modify *Program* 22.1 to represent capacities and flows as real numbers between 0 and 1 that are expressed with d digits to the right of the decimal point, where d is a fixed constant.

▷ **22.6** Give the adjacency-matrix and the adjacency-lists representations of the flow network shown in *Figure* 22.11, in the style of *Figure* 22.10.

▷ **22.7** Write a representation-independent program that builds a flow network by reading edges (pairs of integers between 0 and V-1) with integer capacities from standard input. Assume that the capacity upper bound M is less than 2^{20}.

22.8 Extend your solution to Exercise 22.7 to use symbolic names instead of integers to refer to vertices (see *Program* 17.10).

○ **22.9** Find a large network online that you can use as a vehicle for testing flow algorithms on realistic data. Possibilities include transportation networks (road, rail, or air), communications networks (telephone or computer connections), or distribution networks. If capacities are not available, devise a reasonable model to add them. Write a program that uses the ADT of *Program* 22.1 to build flow networks from your data, perhaps using your solution to Exercise 22.8. If warranted, develop additional ADT functions to clean up the data, as described in Exercises 17.33–35.

22.10 Implement a flow-network ADT for sparse networks with capacities between 0 and 2^{20}, based on *Program* 17.6. Include a random-network generator based on *Program* 17.7. Use a separate ADT to generate capacities, and develop two implementations: one that generates uniformly distributed capacities and another that generates capacities according to a Gaussian distribution. Implement client programs that generate random networks for both weight distributions with a well-chosen set values of V and E, so that you can use them to run empirical tests on graphs drawn from various distributions of edge weights.

IPR2023-00124
KASPERSKY EX 1036 Page 450

22.11 Implement a flow-network ADT for dense networks with capacities between 0 and 2^{20}, using an adjacency-matrix representation. Include a random-network generator based on *Program* 17.8 and edge-capacity generators as described in Exercise 22.10. Write client programs to generate random networks for both weight distributions with a well-chosen set values of V and E, so that you can use them to run empirical tests on graphs drawn from these models.

●**22.12** Write a program that generates V random points in the plane, then builds a flow network with edges (in both directions) connecting all pairs of points within a given distance d of each other (see Program 3.20), setting each edge's capacity using one of the random models described in Exercise 22.10. Determine how to set d so that the expected number of edges is E.

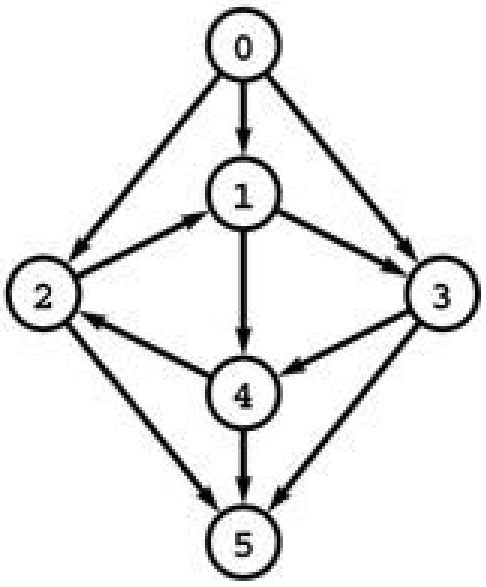
22.13 Modify *Program* 22.2 to check that flow and capacity are both positive or both negative for all edges, that the flow of `u-v` and the flow of `v-u` sum to zero for all `u` and `v`, that the same holds true for capacity, and that there are precisely two list nodes, which are mutually linked together, corresponding to each edge.

22.14 Write an ADT function like the one described in Exercise 22.13 for the adjacency-matrix representation.

▷**22.15** Find all the maxflows in the network depicted in *Figure* 22.12. Give cycle representations for each of them.

Figure 22.12 Flow Network with Cycle *This flow network is like the one depicted in Figure 22.11, but with the direction of two of the edges reversed, so there are two cycles. It is also the subject of several exercises through out the chapter.*

	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



IPR2023-00124
KASPERSKY EX 1036 Page 451

22.16 Write an ADT function for the adjacency-lists representation that reads values and cycles (one per line, in the format illustrated in *Figure* 22.8) and computes the corresponding flow.

22.17 Write an ADT function for the adjacency-lists representation that finds the cycle representation of a network's flow using the method described in the proof of Property 22.2 and prints values and cycles (one per line, in the format illustrated in *Figure* 22.8).

○ **22.18** Write an ADT function for the adjacency-lists representation that removes cycles from an *st*-flow.

○ **22.19** Write a program that assigns integer flows to each edge in any given digraph that contains no sinks and no sources such that the digraph is a flow network that is a circulation.

○ **22.20** Suppose that a flow represents goods to be transferred by trucks between cities, with the flow on edge `u-v` representing the amount to be taken from city `u` to `v` in a given day. Write an ADT function that prints out daily orders for truckers, telling them how much and where to pick up and how much and where to drop off. Assume that there are no limits on the supply of truckers and that nothing leaves a given distribution point until everything has arrived.

22.2 AUGMENTING-PATH MAXFLOW ALGORITHMS

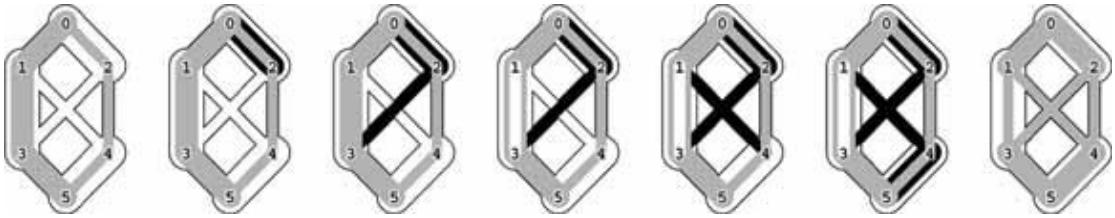
An effective approach to solving maxflow problems was developed by L. R. Ford and D. R. Fulkerson in 1962. It is a generic method for increasing flows incrementally along paths from source to sink that serves as the basis for a family of algorithms. It is known as the *Ford– Fulkerson method* in the classical literature; the more descriptive term *augmenting-path method* is also widely used.

Consider any directed `path` (not necessarily a simple one) from source to sink through an *st*-network. Let x be the minimum of the unused capacities of the edges on the path. We can increase the network's flow value by at least x, by increasing the flow in all edges on the `path` by that amount. Iterating this action, we get a first attempt at computing flow in a network: find another path, increase the flow along that path, and continue until all paths from source to sink have at least one full edge (so that we can no longer increase flow in this way). This algorithm will compute the maxflow in some cases, but will fall short in other cases. *Figure* 22.6 illustrates a case where it fails.

Figure 22.13 Augmenting Flow along a Path *This sequence shows the process*

IPR2023-00124
KASPERSKY EX 1036 Page 452

of increasing flow in a network along a `path` *of forward and backward edges. Starting with the flow depicted at the left and reading from left to right, we increase the flow in* `0-2` *and then* `2-3` *(additional flow is shown in black). Then we decrease the flow in* `1-3` *(shown in white) and divert it to* `1-4` *and then* `4-5`*, resulting in the flow at the right.*



To improve the algorithm such that it always finds a maxflow, we consider a more general way to increase the flow, along any `path` from source to sink through the network's underlying undirected graph. The edges on any such `path` are either *forward* edges, which go with the flow (when we traverse the `path` from source to sink, we traverse the edge from its source vertex to its destination vertex) or *backward* edges, which go against the flow (when we traverse the `path` from source to sink, we traverse the edge from its destination vertex to its source vertex). Now, for any `path` with no full forward edges and no empty backward edges, we can increase the amount of flow in the network by increasing flow in forward edges and *decreasing* flow in backward edges. The amount by which the flow can be increased is limited by the minimum of the unused capacities in the forward edges and the flows in the backward edges. *Figure* 22.13 depicts an example. In the new flow, at least one of the forward edges along the `path` becomes full or at least one of the backward edges along the `path` becomes empty.

The process just sketched is the basis for the classical Ford–Fulkerson maxflow algorithm (augmenting-path method). We summarize it as follows:

Start with zero flow everywhere. Increase the flow along any `path` *from source to sink with no full forward edges or empty backward edges, continuing until there are no such paths in the network.*

Remarkably, this method always finds a maxflow, *no matter how* we choose the paths. Like the MST method discussed in Section 20.1 and the Bellman–Ford shortest-paths method discussed in section 21.7, it is a generic algorithm that is useful because it establishes the correctness of a whole family of more specific algorithms. We are free to use any method whatever to choose the path.

Figure 22.14 Augmenting-Path Sequences *In these three examples, we augment a flow along different sequences of augmenting paths until no augmenting* `path` *can be found. The flow that results in each case is a maximum flow. The key classical theorem in the theory of network flows states that we get a maximum*

flow in any network, no matter what sequence of paths we use (see Property 22.5).

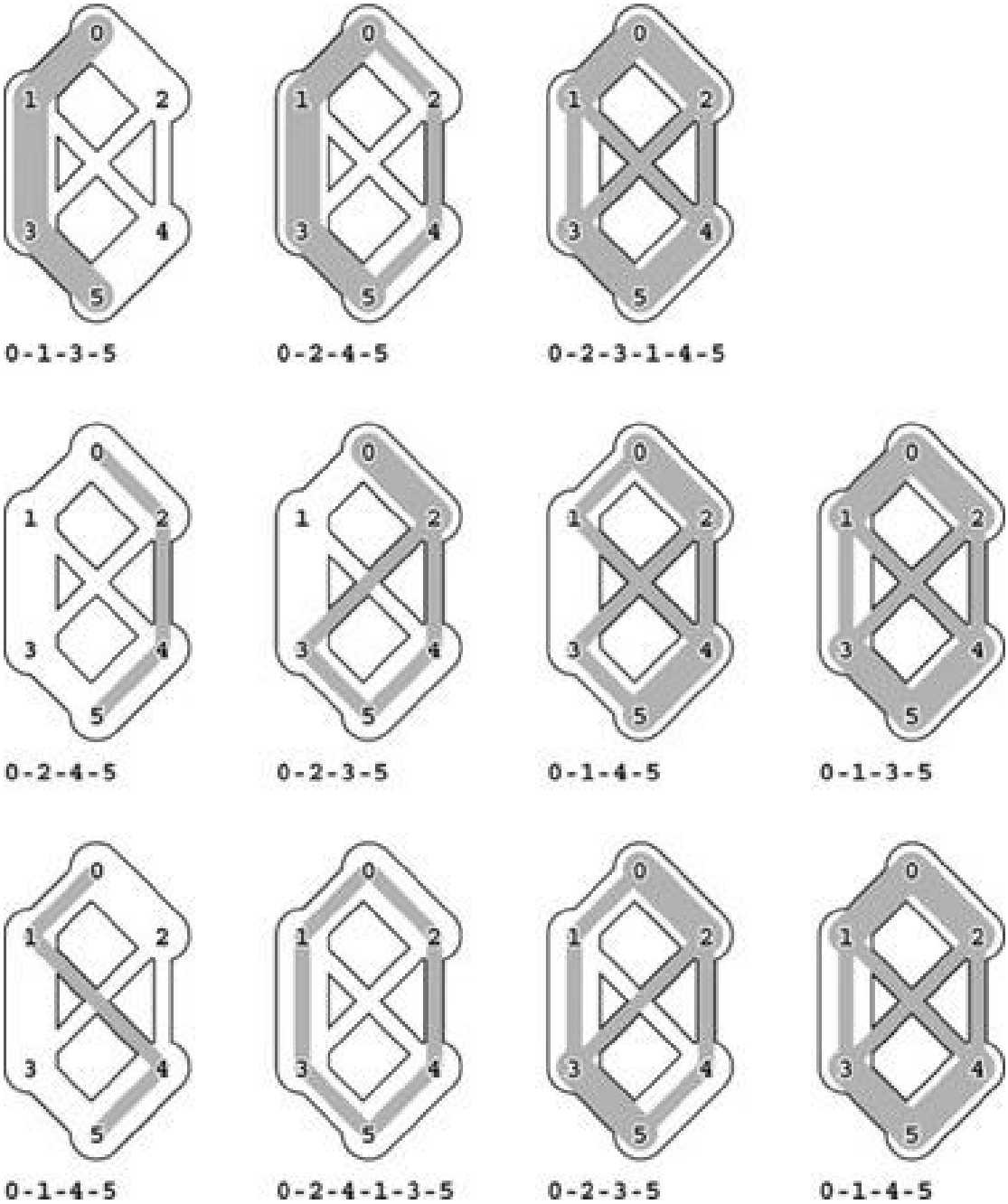


Figure 22.14 illustrates several different sequences of augmenting paths that all lead to a maxflow for a sample network. Later in this section, we examine several algorithms that compute sequences of augmenting paths, all of which lead to a maxflow. The algorithms differ in the number of augmenting paths they compute, the lengths of the paths, and the costs of finding each path, but they all implement the Ford–Fulkerson algorithm and find a maxflow.

To show that any flow computed by any implementation of the Ford–Fulkerson algorithm indeed has maximal value, we show that this fact is equivalent to a key fact known

IPR2023-00124
KASPERSKY EX 1036 Page 454

as the *maxflow–mincut theorem*. Understanding this theorem is a crucial step in understanding network-flow algorithms. As suggested by its name, the theorem is based on a direct relationship between flows and *cuts* in networks, so we begin by defining terms that relate to cuts.

Recall from Section 20.1 that a *cut* in a graph is a partition of the vertices into two disjoint sets, and a *crossing edge* is an edge that connects a vertex in one set to a vertex in the other set. For flow networks, we refine these definitions as follows (see *Figure* 22.15).

Definition 22.3 *An st-cut is a cut that places vertex s in one of its sets and vertex t in the other.*

Each crossing edge corresponding to an *st*-cut is either an *st*-edge that goes from a vertex in the set containing *s* to a vertex in the set containing *t*, or a *ts*-edge that goes in the other direction. We sometimes refer to the set of crossing edges as a **cut set**. The **capacity** of an *st*-cut in a flow network is the sum of the capacities of that cut's *st*-edges, and the **flow across** an *st*-cut is the difference between the sum of the flows in that cut's *st*-edges and the sum of the flows in that cut's *ts*-edges.

Removing a cut set divides a connected graph into two connected components, leaving no `path` connecting any vertex in one to any vertex in the other. Removing all the edges in an *st*-cut of a network leaves no `path` connecting *s* to *t* in the underlying undirected graph, but adding any one of them back could create such a path.

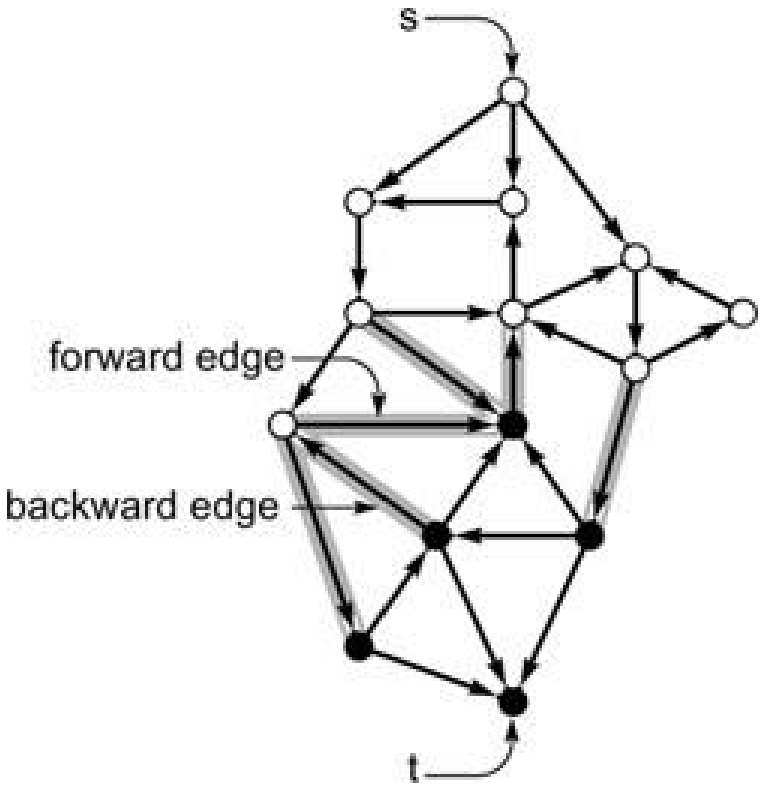
Cuts are the appropriate abstraction for the application mentioned at the beginning of the chapter where a flow network describes the movement of supplies from a depot to the troops of an army. To cut off supplies completely and in the most economical manner, an enemy might solve the following problem:

Minimum cut Given an *st*-network, find an *st*-cut such that the capacity of no other cut is smaller. For brevity, we refer to such a cut as a *mincut*, and to the problem of finding one in a network as the *mincut problem*.

The mincut problem is a generalization of the connectivity problems that we discussed briefly in Section 18.6. We analyze specific relationships in detail in section 22.4.

Figure 22.15 St-cut Terminology *An st-network has one source s and one sink t. An st-cut is a partition of the vertices into a set containing s* (white) *and another set containing t* (*black*). *The edges that connect a vertex in one set with a vertex in the other* (high-lighted in gray) *are known as a cut set. A forward edge goes from a vertex in the set containing s to a vertex in the set containing t; a backward edge goes the other way. There are four forward edges and two backward edges in the cut set shown here.*

IPR2023-00124
KASPERSKY EX 1036 Page 455



The statement of the mincut problem includes no mention of flows, and these definitions might seem to digress from our discussion of the augmenting-path algorithm. On the surface, computing a mincut (a set of edges) seems easier than computing a maxflow (an assignment of weights to all the edges). On the contrary, the key fact of this chapter is that the maxflow and mincut problems are intimately related. The augmenting-path method itself, in conjunction with two facts about flows and cuts, provides a proof.

Property 22.3 *For any st-flow, the flow across each st-cut is equal to the value of the flow.*

Proof: This property is an immediate consequence of the generalization of Property 22.1 that we discussed in the associated proof (see *Figure* 22.7). Add an edge `t-s` with flow equal to the value of the flow, such that inflow is equal to outflow for any set of vertices. Then, for any *st*-cut where C_s is the vertex set containing *s* and C_t is the vertex set containing *t*, the inflow to C_s is the inflow to *s* (the value of the flow) plus the sum of the flows in the backward edges across the cut; and the outflow from C_s is the sum of the flows in the forward edges across the cut. Setting these two quantities equal establishes the desired result.■

Property 22.4 *No st-flow's value can exceed the capacity of any st-cut.*

Proof: The flow across a cut certainly cannot exceed that cut's capacity, so this result is immediate from Property 22.3.■

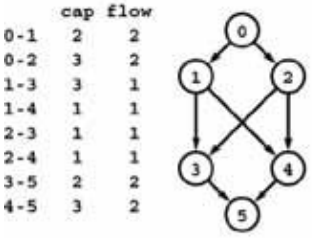
IPR2023-00124
KASPERSKY EX 1036 Page 456

In other words, cuts represent bottlenecks in networks. In our military application, an enemy that is not able to cut off army troops completely from their supplies could still be sure that supply flow is restricted to at most the capacity of any given cut. We certainly might imagine that the cost of making a cut is proportional to its capacity in this application, thus motivating the invading army to find a solution to the mincut problem. More important, these facts also imply, in particular, that no flow can have value higher than the capacity of any minimum cut.

Property 22.5 (Maxflow–mincut theorem) *The maximum value among all st-flows in a network is equal to the minimum capacity among all st-cuts.*

Proof: It suffices to exhibit a flow and a cut such that the value of the flow is equal to the capacity of the cut. The flow has to be a maxflow because no other flow value can exceed the capacity of the cut and the cut has to be a minimum cut because no other cut capacity can be lower than the value of the flow (by Property 22.4). The Ford–Fulkerson algorithm gives precisely such a flow and cut: When the algorithm terminates, identify the first full forward or empty backward edge on every `path` from s to t in the graph. Let C_s be the set of all vertices that can be reached from s with an undirected `path` that does not contain a full forward or empty backward edge, and let C_t be the remaining vertices. Then, t must be in C_t, so (C_s,C_t) is an st-cut, whose cut set consists entirely of full forward or empty backward edges. The flow across this cut is equal to the cut's capacity (since forward edges are full and the backward edges are empty) and also to the value of the network flow (by Property 22.3).■

Figure 22.16 All *st*-cuts *This list gives, for all the st-cuts of the network at left, the vertices in the set containing s, the vertices in the set containing t, forward edges, backward edges, and capacity (sum of capacities of the forward edges). For any flow, the flow across all the cuts (flow in forward edges minus flow in backward edges) is the same. For example, for the flow in the network at left, the flow across the cut separating* `0 1 3` *and* `2 4 5` *is* `2 + 1 + 2` *(the flow in* `0-2`*,* `1-4`*, and* `3-5`*, respectively) minus* `1` *(the flow in* `2-3`*), or* `4`*. This calculation also results in the value* `4` *for every other cut in the network, and the flow is a maximum flow because its value is equal to the capacity of the minimum cut (see Property 22.5). There are two minimum cuts in this network.*



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

IPR2023-00124
KASPERSKY EX 1036 Page 457

This proof also establishes explicitly that the Ford–Fulkerson algorithm finds a maxflow. No matter what method we choose to find an augmenting path, and no matter what paths we find, we always end up with a cut whose flow is equal to its capacity, and therefore also is equal to the value of the network's flow, which therefore must be a maxflow.

Another implication of the correctness of the Ford–Fulkerson algorithm is that, for any flow network with integer capacities, there exists a maxflow solution where the flows are all integers. Each augmenting `path` increases the flow by a positive integer (the minimum of the unused capacities in the forward edges and the flows in the backward edges, all of which are always positive integers). This fact justifies our decision to restrict our attention to integer capacities and flows. It is possible to design a maxflow with noninteger flows, even when capacities are all integers (see Exercise 22.25), but we do not need to consider such flows. This restriction is important: Generalizing to allow capacities and flows that are real numbers can lead to unpleasant anomalous situations. For example, the Ford–Fulkerson algorithm might lead to an infinite sequence of augmenting paths that does not even converge to the maxflow value (*see reference section*).

The generic Ford–Fulkerson algorithm does not specify any particular method for finding an augmenting path. Perhaps the most natural way to proceed is to use the generalized graph-search strategy of section 18.8. To this end, we begin with the following definition:

Definition 22.4 *Given a flow network and a flow, the* **residual network** *for the flow has the same vertices as the original and one or two edges in the residual network for each edge in the original, defined as follows: For each edge* `u-v` *in the original, let* `f` *be the flow and* `c` *the capacity. If* `f` *is positive, include an edge* `v-u` *in the residual with capacity* `f` *; and if* `f` *is less than* `c`*, include an edge* `u-v` *in the residual with capacity* `c-f`.

If `u-v` is empty (f is equal to `0`), there is a single corresponding edge `u-v` with capacity `c` in the residual; if `u-v` is full (f is equal to `c`), there is a single corresponding edge `v-u` with capacity `f` in the residual; and if `u-v` is neither empty nor full, both `u-v` and `v-u` are in the residual with their respective capacities.

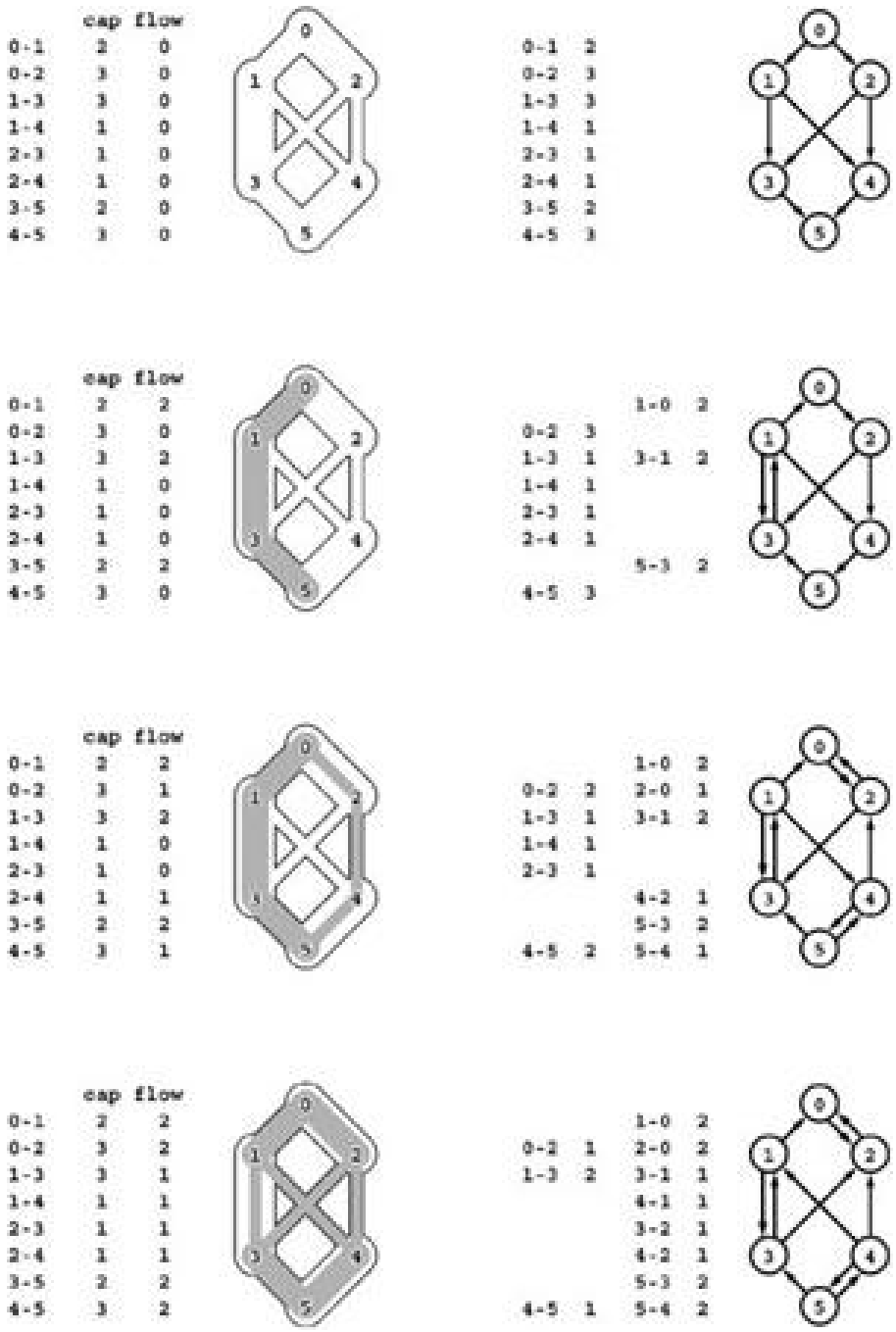
Residual networks allow us to use any generalized graph search (see section 18.8) to find an augmenting path, since any `path` from source to sink in the residual network corresponds directly to an augmenting `path` in the original network. Increasing the flow along the `path` implies making changes in the residual network: for example, at least one edge on the `path` becomes full or empty, so at least one edge in the residual network changes direction or disappears. *Figure* 22.17 shows a sequence of augmenting paths and the corresponding residual networks for an example.

As we have seen, the classical graph-search methods differ in only the choice of the data

structure to hold unexplored edges. For the maxflow problem, choosing a data structure corresponds immediately to implementing a particular strategy for finding augmenting paths. In particular we examine the use of a FIFO queue, a priority queue, a stack, and a randomized queue.

Program 22.3 is a priority-queue–based implementation that encompasses all these possibilities, using a modified version of our PFS graph-search implementation from *Program* 21.1. This implementation allows us to choose among several different classical implementations of the Ford–Fulkerson algorithm, simply by setting priorities so as to implement various data structures for the fringe.

Figure 22.17 Residual Networks (Augmenting Paths) *Finding augmenting paths in a flow network is equivalent to finding directed paths in the residual network that is defined by the flow. For each edge in the flow network, we create an edge in each direction in the residual network: one in the direction of the flow with weight equal to the unused capacity and one in the opposite direction with weight equal to the flow. We do not include edges of weight* `0` *in either case. Initially (top), the residual network is the same as the flow network with weights equal to capacities. When we augment along the* `path` `0-1-3-5` *(second from top), we fill edges* `0-1` *and* `3-5` *to capacity, so they switch direction in the residual network, we reduce the weight of* `1-3` *to correspond to the remaining flow, and we add the edge* `3-1` *of weight* `2`*. Similary, when we augment along the* `path` `0-2-4-5`*, we fill* `2-4` *to capacity, so it switches direction, and we have edges in either direction between* `0` *and* `2` *and between* `4` *and* `5` *to represent flow and unused capacity. After we augment along* `0-2-3-1-4-5` *(bottom), no directed paths from source to sink remain in the residual network, so there are no augmenting paths.*



Program 22.3 Augmenting-paths maxflow implementation

This ADT function implements the generic augmenting-paths (Ford– Fulkerson) maxflow algorithm, using a PFS implementation and priority-queue interface based on the one that we used for Dijkstra's algorithm (Program 21.1), modified to represent the spanning tree with links to the network's edges (*see text*). The Q macro

IPR2023-00124
KASPERSKY EX 1036 Page 460

gives the capacity in the residual network of u's edge. Appropriate definitions of P yield various different maxflow algorithms.

```
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; }
  }
```

The key to understanding *Program* 22.3 is to understand the two ways in which it takes advantage of the fact that the adjacency-lists representation contains two representations of each flow-network edge (see *Figure* 22.10). First, it does not need to explicitly construct the residual network: a simple macro suffices. Second, it represents the PFS search tree with the network's edges, to enable flow augmentation: In *Program* 21.1, when we traverse an edge `v-w`, we set `st[w] = v`; In *Program* 22.3, when we traverse an edge `v-w` where we have a link `u` on `v`'s adjacency list with `u->v = w`, we `set st[w] = u`. This gives access to both the flow (in `st[w]`->flow) and the parent vertex (in `st[w]`->dup->v).

As discussed in section 21.2, using a priority queue to implement a stack, queue, or randomized queue for the fringe data structure incurs an extra factor of lg V in the cost of fringe operations. Since we could avoid this cost by using a generalized-queue ADT in an implementation like Program 18.10 with direct implementations of these ADTs, we assume when analyzing the algorithms that the costs of fringe operations are constant in these cases. By using the single implementation in *Program* 22.3, we emphasize the direct relationship between various Ford–Fulkerson implementations.

Although it is general, *Program* 22.3 does not encompass all implementations of the Ford–Fulkerson algorithm (see, for example, Exercises 22.38 and 22.40). Researchers continue to develop new ways to implement the algorithm. But the family of algorithms encompassed by *Program* 22.3 is widely used, gives us a basis for understanding computation of maxflows, and introduces us to straightforward implementations that perform well on practical networks.

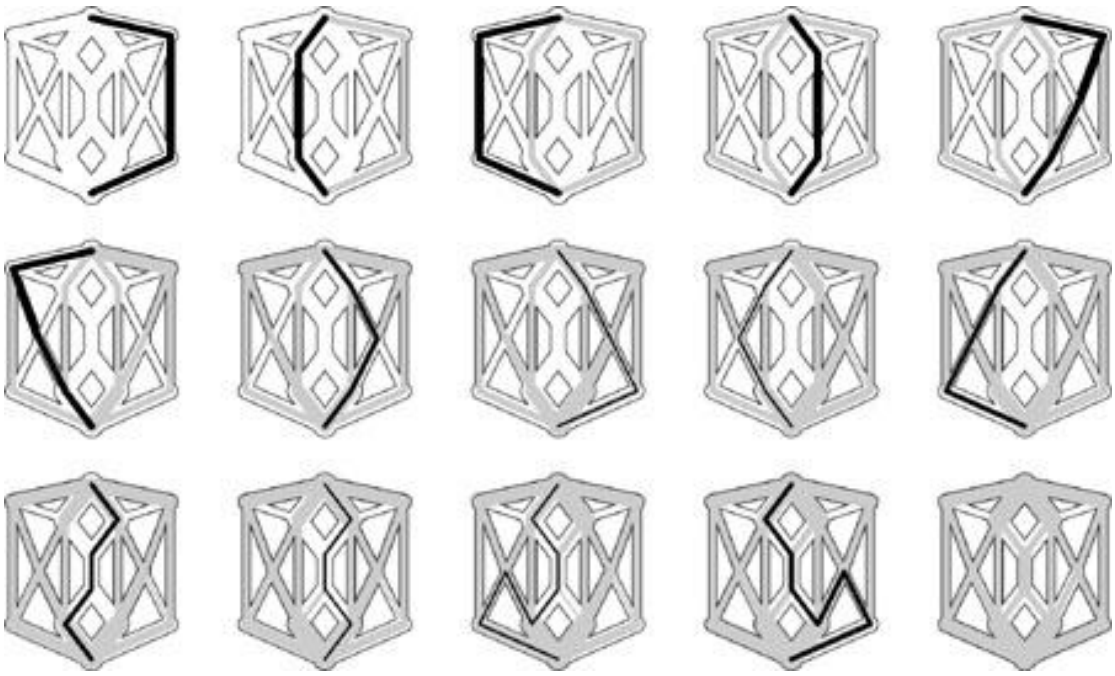
As we soon see, these basic algorithmic tools get us simple (and useful, for many applications) solutions to the network-flow problem. A complete analysis establishing which specific method is best is a complex task, however, because their running times depend on

• The number of augmenting paths needed to find a maxflow

• The time needed to find each augmenting path

Figure 22.18 Shortest Augmenting Paths *This sequence illustrates how the shortest-augmenting-path implementation of the Ford–Fulkerson method finds a maximum flow in a sample network.* `Path` *lengths increase as the algorithm progresses: The first four paths in the top row are of length* `3`*; the last* `path` *in the top row and all of the paths in the second row are of length* `4`*; the first two paths in the bottom row are of length 5; and the process finishes with two paths of length* `7` *that each have a backward edge.*

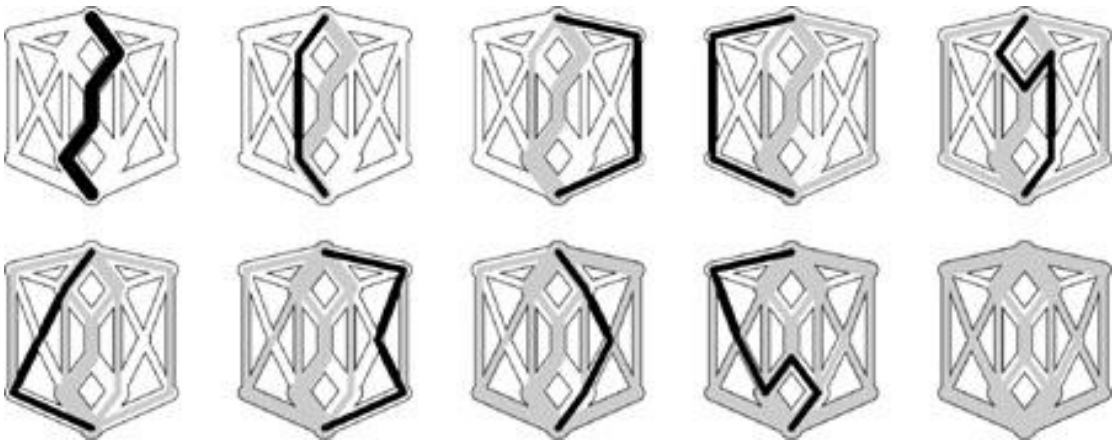
IPR2023-00124
KASPERSKY EX 1036 Page 462



These quantities can vary widely, depending on the network being processed and on the graph-search strategy (fringe data structure).

Perhaps the simplest Ford–Fulkerson implementation uses the *shortest* augmenting `path` (as measured by the number of edges on the path, not flow or capacity). This method was suggested by Edmonds and Karp in 1972. To implement it, we use a queue for the fringe, either by using the value of an increasing counter for P or by using a queue ADT instead of a priority-queue ADT in Program22.3. In this case, the search for an augmenting `path` amounts to breadth-first search (BFS) in the residual network, precisely as described in Sections 18.8 and 21.2. *Figure* 22.18 shows this implementation of the Ford–Fulkerson method in operation on a sample network. For brevity, we refer to this method as the *shortest-augmenting-path* maxflow algorithm. As is evident from the figure, the lengths of the augmenting paths form a nondecreasing sequence. Our analysis of this method, in Property 22.7, proves that this property is characteristic.

Figure 22.19 Maximum-Capacity Augmenting Paths *This sequence illustrates how the maximum-capacity–augmenting-path implementation of the Ford-Fulkerson method finds a maxflow in a sample network.* `Path` *capacities decrease as the algorithm progresses, but their lengths may increase or decrease. The method needs only nine augmenting paths to compute the same maxflow as the one depicted in Figure 22.18.*



Another Ford–Fulkerson implementation suggested by Edmonds and Karp is the following: *Augment along the* `path` *that increases the flow by the largest amount*. To implement this method with our generic PFS implementation, we use the priority

```
#define P ( Q > wt[v] ? wt[v] : Q )
```

in *Program* 22.3. This priority makes the algorithm choose edges from the fringe to give the maximum amount of flow that can be pushed through a forward edge or diverted from a backward edge. For brevity, we refer to this method as the *maximum-capacity augmenting-path* maxflow algorithm. *Figure* 22.19 illustrates the algorithm on the same flow network as that in *Figure* 22.18.

These are but two examples (ones we can analyze!) of Ford– Fulkerson implementations. At the end of this section, we consider others. Before doing so, we consider the task of analyzing augmenting-path methods, so as to learn their properties and, ultimately, to decide which one will have the best performance.

In trying to choose among the family of algorithms represented by *Program* 22.3, we are in a familiar situation. Should we focus on worst-case performance guarantees, or do those represent a mathematical fiction that may not relate to networks that we encounter in practice? This question is particularly relevant in this context, because the classical worst-case performance bounds that we can establish are much higher than the actual performance results that we see for typical graphs. Many factors further complicate the situation. For example, the worst-case running time for several versions depends not just on V and E, but also on the values of the edge capacities in the network. Developing a maxflow algorithm with fast guaranteed performance has been a tantalizing problem for several decades, and numerous methods have been proposed. Evaluating all these methods for all the types of networks that are likely to be encountered in practice, with sufficient precision to allow us to choose among them, is not as clear-cut as is the same task for other situations that we have studied, such as typical practical applications of sorting

IPR2023-00124
KASPERSKY EX 1036 Page 464

or searching algorithms.

Keeping these difficulties in mind, we now consider the classical results about the worst-case performance of the Ford–Fulkerson method: One general bound and two specific bounds, one for each of the two augmenting-path algorithms that we have examined. These results serve more to give us insight into characteristics of the algorithms than to allow us to predict performance to a sufficient degree of accuracy for meaningful comparison. We discuss empirical comparisons of the methods at the end of the section.

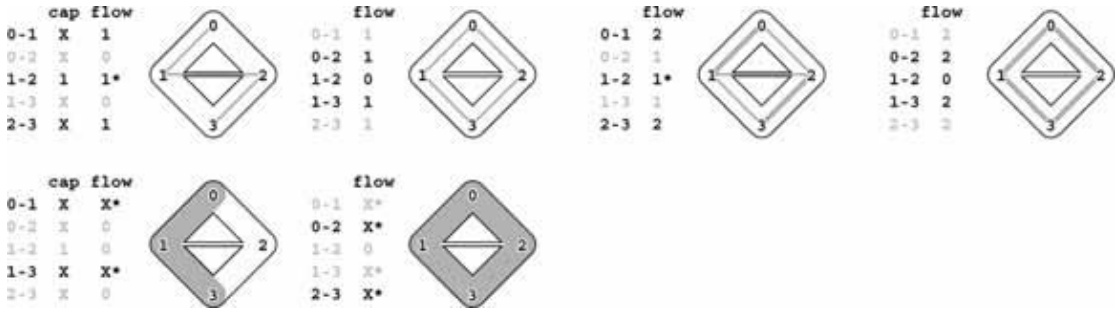
Property 22.6 *Let M be the maximum edge capacity in the network. The number of augmenting paths needed by any implementation of the Ford–Fulkerson algorithm is at most equal to VM.*

Proof: Any cut has at most V edges, of capacity M, for a total capacity of VM. Every augmenting `path` increases the flow through every cut by at least 1, so the algorithm must terminate after VM passes, since all cuts must be filled to capacity after that many augmentations.■

This bound is tight. For example, suppose that we use a *longest* augmenting-path algorithm (perhaps based on the intuition that the longer the path, the more flow we put on the network's edges). Since we are counting iterations, we ignore, for the moment, the cost of computing such a path. The (classical) example shown in *Figure* 22.20 shows a network for which choosing longest paths makes the bound in Property 20.6 tight: It is a network for which the number of iterations of a longest augmenting-path algorithm is equal to the maximum edge capacity. This example tells us that we must undertake a more detailed scrutiny to know whether other specific implementations use substantially fewer iterations than are indicated by Property 22.6. It also gives an upper bound on the running time of any Ford–Fulkerson implementation that is useful in many practical situations:

Figure 22.20 Two Scenarios for the Ford-Fulkerson Algorithm This network illustrates that the number of iterations used by the Ford-Fulkerson algorithm depends on the capacities of the edges in the network and the sequence of paths chosen by the implementation. It consists of four edges of capacity X and one of capacity 1. The scenario depicted at the top shows that an implementation that alternates between using `0-1-2-3` and `0-2-1-3` as augmenting paths (for example, one that prefers long paths) would require X pairs of iterations like the two pairs shown, each pair incrementing the total flow by 2. The scenario depicted at the bottom shows that an implementation that chooses `0-1-3` and then `0-2-3` as augmenting paths (for example, one that prefers short paths) finds the maximum flow in just two iterations. If edge capacities are, say, 32-bit integers, the scenario depicted at the top would be billions of times slower than the scenario depicted the bottom.

IPR2023-00124
KASPERSKY EX 1036 Page 465



Corollary *The time required to find a maxflow is O (VEM), which is O ($V^2 M$) for sparse networks.*

Proof: Immediate from the basic result that generalized graph search is linear in the size of the graph representation (Property 18.12). As mentioned, we need an extra lg *V* factor if we are using a priority-queue fringe implementation.■

For sparse networks and networks with small integer capacities, this bound is reasonably low. The proof actually establishes that the factor of *M* can be replaced by the ratio between the largest and smallest nonzero capacities in the network (see Exercise 22.27). When this ratio is low, the bound tells us that any Ford–Fulkerson implementation will find a maxflow in time proportional to the time required to (for example) solve the all-shortest-paths problem, in the worst case. There are many situations where the capacities are indeed low and the factor of *M* is of no concern. We will see an example in section 22.4.

When *M* is large, the *VEM* worst-case bound is high; but it is pessimistic, as we obtained it by multiplying together worst-case bounds that derive from contrived examples. Actual costs on practical networks are typically much lower.

From a theoretical standpoint, our first goal is to discover, using the rough subjective categorizations of Section 17.8, whether or not the maximum-flow problem for networks with large integer weights is tractable (solvable by a polynomial-time algorithm). The bounds just derived do *not* resolve this question, because the maximum weight $M = 2^m$ could grow exponentially with *V* and *E*. From a practical standpoint, we seek better performance guarantees. To pick a typical practical example, suppose that we use 32-bit integers ($m = 32$) to represent edge weights. In a graph with hundreds of vertices and thousands of edges, the corollary to Property 22.6 says that we might have to perform hundreds of trillions of operations in an augmenting-path algorithm. If we are dealing with millions of vertices, this point is moot, not only because we will not have weights as large $2^{1000000}$, but also because V^3 and *VE* are so large as to make the bound meaningless. We are interested both in finding a polynomial bound to resolve the tractability question and in finding better bounds that are relevant for situations that we might encounter in prac-

tice.

Property 22.6 is general: It applies to any Ford–Fulkerson implementation at all. The generic nature of the algorithm leaves us with a substantial amount of flexibility to consider a number of simple implementations in seeking to improve performance. We expect that specific implementations might be subject to better worst-case bounds. Indeed, that is one of our primary reasons for considering them in the first place! Now, as we have seen, implementing and using a large class of these implementations is trivial: We just substitute different generalized-queue implementations or priority definitions in *Program* 22.3. Analyzing differences in worst-case behavior is more challenging, as indicated by the classical results that we consider next for the two basic augmenting-path implementations that we have considered.

First, we analyze the shortest-augmenting-path algorithm. This method is not subject to the problem illustrated in *Figure* 22.20. Indeed, we can use it to replace the factor of M in the worst-case running time with $VE/2$, thus establishing that the network-flow problem is tractable. We might even classify it as being easy (solvable in polynomial time on practical cases by a simple, if clever, implementation).

Property 22.7 *The number of augmenting paths needed in the shortest-augmenting-path implementation of the Ford–Fulkerson algorithm is at most VE/2.*

Proof: First, as is apparent from the example in *Figure* 22.18, no augmenting `path` is shorter than a previous one. To establish this fact, we show by contradiction that a stronger property holds: No augmenting `path` can decrease the length of the shortest `path` from the source `s` to any vertex in the residual network. Suppose that some augmenting `path` does so, and that `v` is the first such vertex on the path. There are two cases to consider: Either no vertex on the new shorter `path` from `S` to `v` appears anywhere on the augmenting `path` or some vertex `W` on the new shorter `path` from `S` to `V` appear`S` somewhere between `V` and `t` on the augmenting path. Both situations contradict the minimality of the augmenting path.

Now, by construction, every augmenting `path` has at least one *critical edge*: an edge that is deleted from the residual network because it corresponds either to a forward edge that becomes filled to capacity or a backward edge that is emptied. Suppose that an edge `u-v` is a critical edge for an augmenting `path` P of length `d`. The next augmenting `path` for which it is a critical edge has to be of length at least `d+2`, because that `path` has to go from `s` to `v`, then along `v-u`, then from `u` to `t`. The first segment is of length at least `1` greater than the distance from `s` to `u` in P, and the final segment is of length at least `1` greater than the distance from `v` to `t` in P, so the `path` is of length at least `2` greater than P.

IPR2023-00124
KASPERSKY EX 1036 Page 467

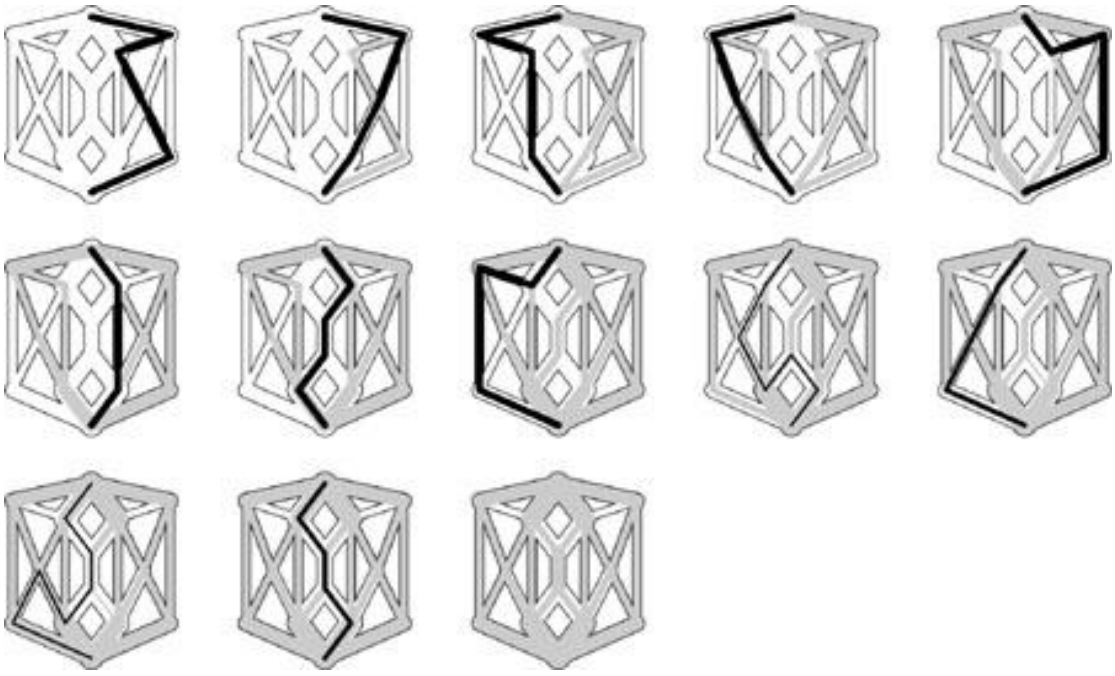
Since augmenting paths are of length at most V, these facts imply that each edge can be the critical edge on at most $V/2$ augmenting paths, so the total number of augmenting paths is at most $EV/2$.■

Corollary *The time required to find a maxflow in a sparse network is* $O(V^3)$.

Proof: The time required to find an augmenting `path` is $O(E)$, so the total time is $O(VE^2)$. The stated bound follows immediately.■

The quantity V^3 is sufficiently high that it does not provide a guarantee of good performance on huge networks. But that fact should not preclude us from using the algorithm on a huge network, because it is a worst-case performance result that may not be useful for predicting performance in a practical application. For example, as just mentioned, the maximum capacity M (or the maximum ratio between capacities) might be much less than V, so the corollary to Property 22.6 would provide a better bound. Indeed, in the best case, the number of augmenting paths needed by the Ford–Fulkerson method is the smaller of the outdegree of `s` or the indegree of `t`, which again might be far smaller than V. Given this range between best- and worst-case performance, comparing augmenting-path algorithms solely on the basis of worst-case bounds is not wise.

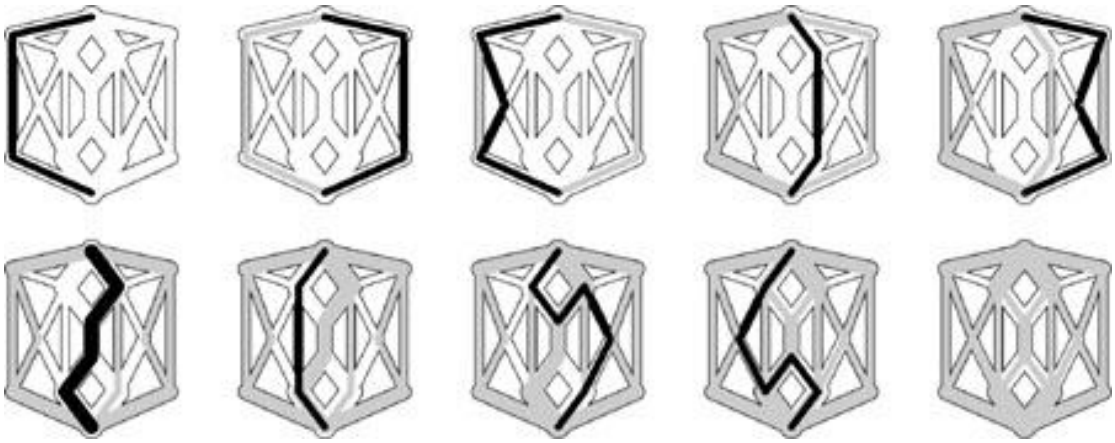
Figure 22.21 Stack-Based Augmenting-Path Search *This illustrates the result of using a stack for the generalized queue in our implementation of the Ford–Fulkerson method, so that the* `path` *search operates like DFS. In this case, the method does about as well as BFS, but its somewhat erratic behavior is rather sensitive to the network representation and has not been analyzed.*



Still, other implementations that are nearly as simple as the shortest-augmenting-path method might admit better bounds or be preferred in practice (or both). For example, the maximum-augmenting-path algorithm used far fewer paths to find a maxflow than did the shortest-augmenting-path algorithm in the example illustrated in Figures 22.18 and 22.19. We now turn to the worst-case analysis of that algorithm.

First, just as for Prim's algorithm and for Dijkstra's algorithm (see Sections 20.6 and 21.2), we can implement the priority queue such that the algorithm takes time proportional to V^2 (for dense graphs) or $(E + V) \log V$ (for sparse graphs) per iteration in the worst case, although these estimates are pessimistic because the algorithm stops when it reaches the sink. We also have seen that we can do slightly better with advanced data structures. The more important and more challenging question is how many augmenting paths are needed.

Figure 22.22 Randomized Augmenting-Path Search *This sequence the result of using a randomized queue for the fringe data structure in the augmenting-path search in the Ford-Fulkerson method. In this example, we happen upon the short high-capacity* `path` *and therefore need relatively few augmenting paths. While predicting the performance characteristics of this method is a challenging task, it performs well in many situations.*



Property 22.8 *The number of augmenting paths needed in the maximal-augmenting-path implementation of the Ford–Fulkerson algorithm is at most $2E \lg M$.*

Proof: Given a network, let F be its maxflow value. Let v be the value of the flow at some point during the algorithm as we begin to look for an augmenting path. Applying Property 22.2 to the residual network, we can decompose the flow into at most E directed paths that sum to F - v, so the flow in at least one of the paths is at least $(F - v)/E$. Now, either we find the maxflow sometime before doing another $2E$ augmenting paths or the value of the augmenting `path` after that sequence of $2E$ paths is less than $(F - v)/2E$, which is less than one-half of the value of the maximum before that sequence of $2E$ paths.

That is, in the worst case, we need a sequence of $2E$ paths to decrease the `path` value by a factor of 2. The first `path` value is at most M, which we need to decrease by a factor of 2 at most lg M times, so we have a total of at most lg M sequences of $2E$ paths.■

Corollary *The time required to find a maxflow in a sparse network is* $O(V^2 \lg M \lg V)$.

Proof: Immediate from the use of a heap-based priority-queue implementation, as for Properties 20.7 and 21.5.■

For values of M and V that are typically encountered in practice, this bound is significantly lower than the $O(V^3)$ bound of the corollary to Property 22.7. In many practical situations, the maximum-augmenting-path algorithm uses significantly fewer iterations than does the shortest-augmenting-path algorithm, at the cost of a slightly higher bound on the work to find each path.

There are many other variations to consider, as reflected by the extensive literature on maxflow algorithms. Algorithms with better worst-case bounds continue to be discovered, and no nontrivial lower bound has been proved—that is, the possibility of a simple linear-time algorithm remains. Although they are important from a theoretical standpoint, many of the algorithms are primarily designed to lower the worst-case bounds for dense graphs, so they do not offer substantially better performance than the maximum-augmenting-path algorithm for the kinds of sparse networks that we encounter in practice. Still, there remain many options to explore in pursuit of better practical maxflow algorithms. We briefly consider two more augmenting-path algorithms next; in section 22.3, we consider another family of algorithms.

One easy augmenting-path algorithm is to use the value of a decreasing counter for `P` or a stack implementation for the generalized queue in *Program* 22.3, making the search for augmenting paths like depth-first search. *Figure* 22.21 shows the flow computed for our small example by this algorithm. The intuition is that the method is fast, is easy to implement, and appears to put flow throughout the network. As we will see, its performance varies remarkably, from extremely poor on some networks to reasonable on others.

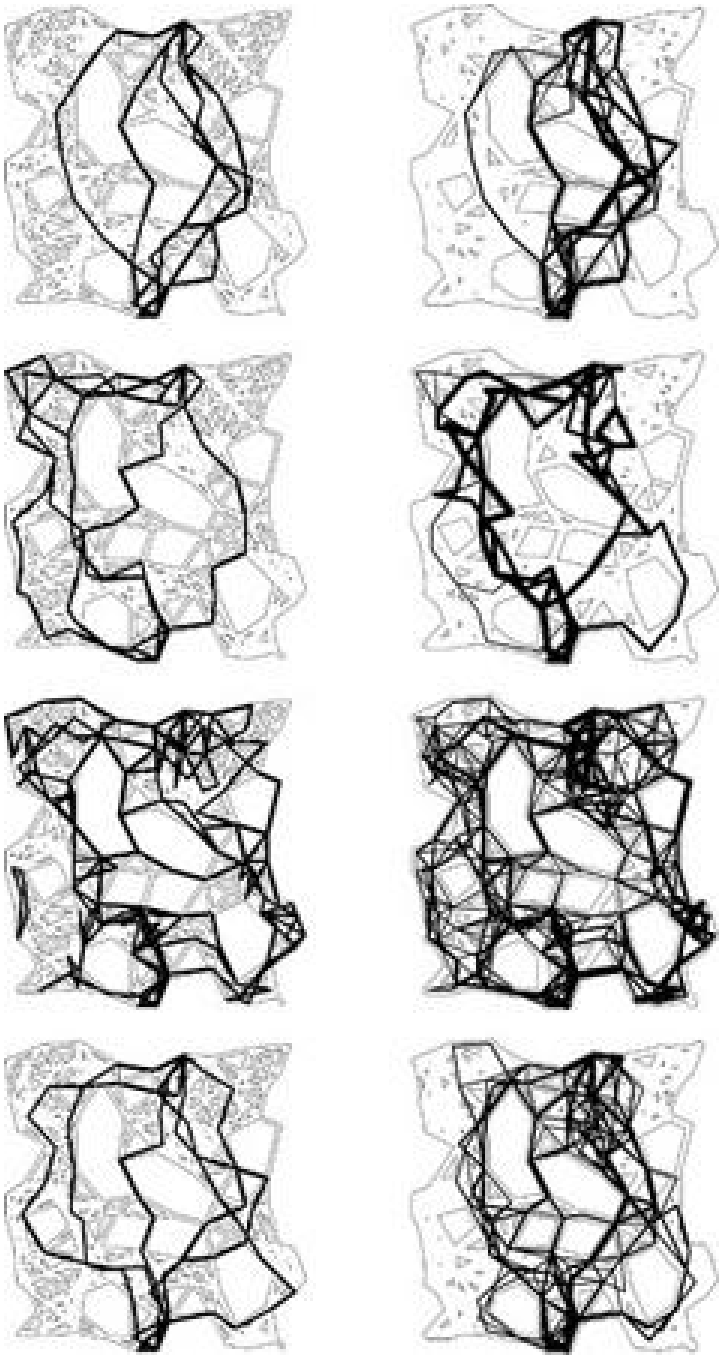
Another alternative is to use a randomized-queue implementation for the generalized queue, so that the search for augmenting paths is a randomized search. *Figure* 22.22 shows the flow computed for our small example by this algorithm. This method is also fast and easy to implement; in addition, as we noted in section 18.8, it may embody good features of both breadth-first and depth-first search. Randomization is a powerful tool in algorithm design, and this problem represents a reasonable situation in which to consider using it.

IPR2023-00124
KASPERSKY EX 1036 Page 470

We conclude this section by looking more closely at the methods that we have examined, to indicate the difficulty of comparing them or attempting to predict performance for practical applications.

Figure 22.23 Random Flow Networks *This figure depicts maxflow computations on our random Euclidean graph, with two different capacity models. On the left, all edges are assigned unit capacities; on the right edges are assigned random capacities. The source is near the middle at the top and the sink near the middle at the bottom. Illustrated top to bottom are the flows computed by the shortest-path, maximum-capacity, stack-based, and randomized algorithms, respectively. Since the vertices are not of high degree and the capacities are small integers, there are many different flows that achieve the maximum for these examples. The indegree of the sink is 6, so all the algorithms find the flow in the unit-capacity model on the left with 6 augmenting paths. The methods find augmenting paths that differ dramatically in character for the random-weight model on the right. In particular, the stack-based method finds long paths on low weight, and even produces a flow with a disconnected cycle.*

IPR2023-00124
KASPERSKY EX 1036 Page 471



As a start in understanding the quantitative differences that we might expect, we use two flow-network models that are based on the Euclidean graph model that we have been using to compare other graph algorithms. Both models use a graph drawn from V points in the plane with random coordinates between 0 and 1 with edges connecting any two points within a fixed distance of each other. They differ in the assignment of capacities to the edges.

The first model simply assigns the same constant value to each capacity. As discussed in section 22.4, this type of network-flow problem is known to be easier than the general problem. For our Euclidean graphs, flows are limited by the outdegree of the source and

the indegree of the sink, so the algorithms each need only a few augmenting paths. But the paths differ substantially for the various algorithms, as we soon see.

The second model assigns random weights from some fixed range of values to the capacities. This model generates the type of networks that people typically envision when thinking about the problem, and the performance of the various algorithms on such networks is certainly instructive.

Both of these models are illustrated in *Figure* 22.23, along with the flows that are computed by the four methods on the two networks. Perhaps the most noticeable characteristic of these examples is that the flows themselves are different in character. All have the same value, but the networks have many maxflows, and the different algorithms make different choices when computing them. This situation is typical in practice. We might try to impose other conditions on the flows that we want to compute, but such changes to the problem can make it more difficult. The mincost-flow problem that we consider in Sections 22.5 through 22.7 is one way to formalize such situations.

Table 22.1 gives more detailed quantitative results for use of the four methods to compute the flows in *Figure* 22.23. An augmenting-path algorithm's performance depends not just on the number of augmenting paths but also on the lengths of such paths and on the cost of finding them. In particular, the running time is proportional to the number of edges examined in the inner loop of *Program* 22.3. Asusual, this number might vary widely, even for a given graph, depending on properties of the representation; but we can still characterize different algorithms. For example, Figures 22.24 and 22.25 show the search trees for the maximum-capacity and shortest-path algorithms, respectively. These examples help support the general conclusion that the shortest-path method expends more effort to find augmenting paths with less flow than the maximum-capacity algorithm, thus helping to explain why the latter is preferred.

Table 22.1 Empirical study of augmenting-path algorithms *This table shows performance parameters for various augmenting-path network-flow algorithms for our sample Euclidean neighbor network (with random capacities with maxflow value 286) and with unit capacities (with maxflow value 6). The maximum-capacity algorithm outperforms the others for both types of networks. The random search algorithm finds augmenting paths that are not much longer than the shortest, and examines fewer nodes. The stack-based algorithm peforms very well for random weights but, though it has very long paths, is competetive for unit weights.*

	paths	mean length	total edges
random capacity 1-50			
shortest	37	10.8	76394
maximum capacity	7	19.3	15660
depth-first	286	123.5	631392
random	35	12.8	58016
capacity 1			
shortest	6	10.5	13877
maximum capacity	6	14.7	10736
depth-first	6	110.8	12291
random	6	12.2	11223

Perhaps the most important lesson that we can learn from studying particular networks in detail in this way is that the gap between the upper bounds of Properties 22.6 through 22.8 and the actual number of augmenting paths that the algorithms need for a given application might be enormous. For example, the flow network illustrated in *Figure* 22.24 has 177 vertices and 2000 edges of capacity less than 100, so the value of the quantity $2E$ lg M in Property 22.8 is more than 25,000; but the maximum-capacity algorithm finds the maxflow with only seven augmenting paths. Similarly, the value of the quantity $VE/2$ in Property 22.7 for this network is 177,000, but the shortest-path algorithm needs only 37 paths.

Figure 22.24 Maximum-Capacity Augmenting Paths (Larger Example)
This figure depicts the augmenting paths computed by the maximum-capacity algorithm for the Euclidean network with random weights that is shown in Figure 22.23, along with the edges in the graph-search spanning tree (in gray). The resulting flow is shown at the bottom right.

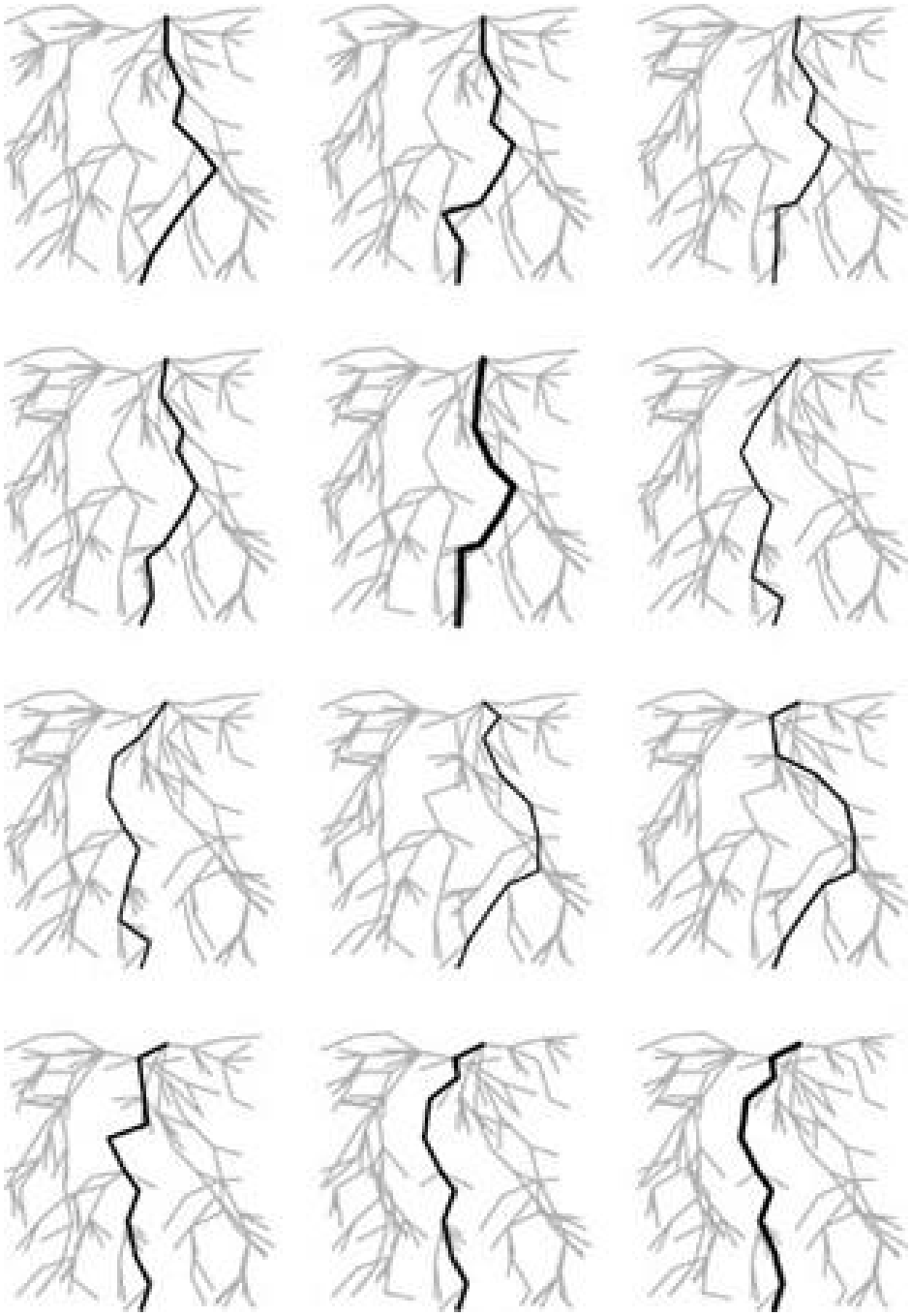
IPR2023-00124
KASPERSKY EX 1036 Page 474



As we have already indicated, the relatively low node degree and the locality of the connections partially explain these differences between theoretical and actual performance in this case. We can prove more accurate performance bounds that account for such details; but such disparities are the rule, not the exception, in flow-network models and in practical networks. On the one hand, we might take these results to indicate that these networks are not sufficiently general to represent the networks that we encounter in practice; on the other hand, perhaps the worst-case analysis is more removed from practice than these kinds of networks.

Large gaps like this certainly provide strong motivation for researchers seeking to lower the worst-case bounds. There are many other possible implementations of augmenting-path algorithms to consider that might lead to better worst-case performance or better practical performance than the methods that we have considered (see Exercises 22.57 through 22.61). Numerous methods that are more sophisticated and have been shown to have improved worst-case performance can be found in the research literature (*see reference section*).

Figure 22.25 Shortest Augmenting Paths (Larger Example) *This figure depicts the augmenting paths computed by the shortest-paths algorithm for the Euclidean network with random weights that is shown in Figure 22.23, along with the edges in the graph-search spanning tree (in gray). This algorithm is much slower than the maximum-capacity algorithm depicted in Figure 22.24 in this case both because it requires a large number of augmenting paths (the paths shown are just the first 12 out of a total of 37) and because the spanning trees are larger (usually containing nearly all of the vertices).*



Another important complication follows when we consider the numerous other problems that reduce to the maxflow problem. When such reductions apply, the flow networks that result may have some special structure that some particular algorithm may be able to exploit for improved performance. For example, in section 22.8 we will examine a reduction that gives flow networks with unit capacities on all edges.

Even when we restrict attention just to augmenting-path algorithms, we see that the study of maxflow algorithms is both an art and a science. The art lies in picking the strategy that is most effective for a given practical situation; the science lies in understanding the essential nature of the problem. Are there new data structures and algorithms that

can solve the maxflow problem in linear time, or can we prove that none exist? In section 22.3, we see that no augmenting-path algorithm can have linear worst-case performance, and we examine a different generic family of algorithms that might.

Exercises

22.21 Show, in the style of *Figure* 22.14, as many different sequences of augmenting paths as you can find for the flow network shown in *Figure* 22.11.

22.22 Show, in the style of *Figure* 22.16, all the cuts for the flow network shown in *Figure* 22.11, their cut sets, and their capacities.

▷**22.23** Find a minimum cut in the flow network shown in *Figure* 22.12.

○**22.24** Suppose that capacities are in equilibrium in a flow network (for every internal node, the total capacity of incoming edges is equal to the total capacity of outgoing edges). Does the Ford–Fulkerson algorithm ever use a backward edge? Prove that it does or give a counterexample.

22.25 Give a maxflow for the flow network shown in *Figure* 22.5 with at least one flow that is not an integer.

▷**22.26** Develop an implementation of the Ford–Fulkerson algorithm that uses a generalized queue instead of a priority queue (see section 18.8).

▷**22.27** Prove that the number of augmenting paths needed by any implementation of the Ford–Fulkerson algorithm is no more than V times the smallest integer larger than the ratio of the largest edge capacity to the smallest edge capacity.

22.28 Prove a linear-time *lower bound* for the maxflow problem: that, for any values of V and E, any maxflow algorithm might have to examine every edge in some network with V vertices and E edges.

▷**22.29** Give a network like *Figure* 22.20 for which the shortest-augmenting-path algorithm has the worst-case behavior that is illustrated.

22.30 Give an adjacency-lists representation of the network in *Figure* 22.20 for which our implementation of the stack-based search (Programs 22.1 and 22.3, using a stack for the generalized queue) has the worst-case behavior that is illustrated.

22.31 Show, in the style of *Figure* 22.17, the flow and residual networks after each augmenting `path` when we use the shortest-augmenting-path algorithm to find a maxflow in the flow network shown in *Figure* 22.11. Also include the graph-search trees for each aug-

IPR2023-00124
KASPERSKY EX 1036 Page 477

menting path. When more than one `path` is possible, show the one that is chosen by the implementations given in this section.

22.32 Do Exercise 22.31 for the maximum-capacity augmenting-path algorithm.

22.33 Do Exercise 22.31 for the stack-based augmenting-path algorithm.

○ **22.34** Exhibit a family of networks for which the maximum-augmenting-path algorithm needs $2E \lg M$ augmenting paths.

● **22.35** Can you arrange the edges such that our implementations take time proportional to E to find each `path` in your example in Exercise 22.34? If necessary, modify your example to achieve this goal. Describe the adjacency-lists representation that is constructed for your example. Explain how the worst case is achieved.

22.36 Run empirical studies to determine the number of augmenting paths and the ratio of the running time to V for each of the four algorithms described in this section, for various networks (see Exercises 22.7–12).

22.37 Develop and test an implementation of the augmenting-path method that uses the source–sink shortest-path heuristic for Euclidean networks of section 21.5.

22.38 Develop and test an implementation of the augmenting-path method that is based on alternately growing search trees rooted at the source and at the sink (see Exercise 21.35 and 21.75).

● **22.39** The implementation of *Program* 22.3 stops the graph search when it finds the first augmenting `path` from source to sink, augments, then starts the search all over again. Alternatively, it could go on with the search and find another path, continuing until all vertices have been marked. Develop and test this second approach.

22.40 Develop and test implementations of the augmenting-path method that use paths that are not simple.

▷ **22.41** Give a sequence of simple augmenting paths that produces a flow with a cycle in the network depicted in *Figure* 22.12.

○ **22.42** Give an example showing that not all maxflows can be the result of starting with an empty network and augmenting along a sequence of simple paths from source to sink.

22.43 Experiment with hybrid methods that use one augmenting-path method at the beginning, then switch to a different augmenting `path` to finish up (part of your task is to

decide what are appropriate criteria for when to switch). Run empirical studies for various networks (see Exercises 22.7–12) to compare these to the basic methods, studying methods that perform better than others in more detail.

22.44 Experiment with hybrid methods that alternate between two or more different augmenting-path methods. Run empirical studies for various networks (see Exercises 22.7–12) to compare these to the basic methods, studying variations that perform better than others in more detail.

○ **22.45** Experiment with hybrid methods that choose randomly among two or more different augmenting-path methods. Run empirical studies for various networks (see Exercises 22.7–12) to compare these to the basic methods, studying variations that perform better than others in more detail.

22.46 [Gabow] Develop a maxflow implementation that uses $m = \lg M$ phases, where the ith phase solves the maxflow problem using the leading i bits of the capacities. Start with zero flow everywhere; then, after the first phase, initialize the flow by doubling the flow found during the previous phase. Run empirical studies for various networks (see Exercises 22.7–12) to compare this implementation to the basic methods.

●**22.47** Prove that the running time of the algorithm described in Exercise 22.46 is $O(VE \lg M)$.

○ **22.48** Write a flow-network ADT function that, given an integer `c`, finds an edge for which increasing the capacity of that edge by `c` increases the maxflow by the maximum amount. Your function may assume that the client has already called `GRAPHmaxflow` to compute a maximum flow.

●● **22.49** Suppose that you are given a mincut for a network. Does this information make it easier to compute a maxflow? Develop an algorithm that uses a given mincut to speed up substantially the search for maximum-capacity augmenting paths.

● **22.50** Write a client program that does dynamic graphical animations of augmenting-path algorithms. Your program should produce images like *Figure* 22.18 and the other figures in this section (see Exercises 17.55–59). Test your implementation for the Euclidean networks among Exercises 22.7–12.

22.3 PREFLOW-PUSH MAXFLOW ALGORITHMS

In this section, we consider another approach to solving the maxflow problem. Using a

IPR2023-00124
KASPERSKY EX 1036 Page 479

generic method known as the *preflow-push* method, we incrementally move flow along the outgoing edges of vertices that have more inflow than outflow. The preflow-push approach was developed by A. Goldberg and R. E. Tarjan in 1986 on the basis of various earlier algorithms. It is widely used because of its simplicity, flexibility, and efficiency.

Figure 22.26 Preflow-Push Example *In the preflow-push algorithm, we maintain a list of the active nodes that have more incoming than outgoing flow (shown below each network). One version of the algorithm is a loop that chooses an active node from the list and pushes flow along outgoing edges until it is no longer active, perhaps creating other active nodes in the process. In this example, we push flow along* `0-1`*, which makes* `1` *active. Next, we push flow along* `1-2` *and* `1-3`*, which makes* `1` *inactive but* `2` *and* `3` *both active. Then, we push flow along* `2-4`*, which makes* `2` *inactive. But* `3-4` *does not have sufficient capacity for us to push flow along it to make* `3` *inactive, so we also push flow back along* `3-1` *to do so, which makes* `1` *active. Then we can push the flow along* `1-2` *and then* `2-4`*, which makes all nodes inactive and leaves a maxflow.*



As defined in section 22.1, a flow must satisfy the equilibrium conditions that the outflow from the source is equal to the inflow to the sink and that inflow is equal to the outflow at each of the internal nodes. We refer to such a flow as a *feasible* flow. An augmenting-path algorithm always maintains a feasible flow: It increases the flow along augmenting paths until a maxflow is achieved. By contrast, the preflow-push algorithms that we consider in this section maintain maxflows that are not feasible because some vertices have more inflow than outflow: They push flow through such vertices until a feasible flow is achieved (no such vertices remain).

Definition 22.5 *In a flow network, a* **preflow** *is a set of positive edge flows satisfying the conditions that the flow on each edge is no greater than that edge's capacity and that inflow is no smaller than outflow for every internal vertex. An* **active** *vertex is an internal vertex whose inflow is larger than its outflow (by convention, the source and sink are never active).*

We refer to the difference between an active vertex's inflow and outflow as that vertex's *excess*. To change the set of active vertices, we choose one and *push* its excess along an outgoing edge, or, if there is insufficient capacity to do so, push the excess back along an incoming edge. If the push equalizes the vertex's inflow and outflow, the vertex becomes inactive; the flow pushed to another vertex may activate that vertex. The preflow-push method provides a systematic way to push excess out of active vertices repeatedly, such

that the process terminates in a maxflow, with no active vertices. We keep active vertices on a generalized queue. As for the augmenting-path method, this decision gives a generic algorithm that encompasses a whole family of more specific algorithms.

Figure 22.27 Bad Case for the Ford-Fulkerson Algorithm *This network represents a family of networks with V vertices for which any augmenting-path algorithm requires V/2 paths of length V/2 (since every augmenting* `path` *has to include the long vertical path), for a total running time proportional to* V^2. *Pre-flow-push algorithms find maxflows in such networks in linear time.*

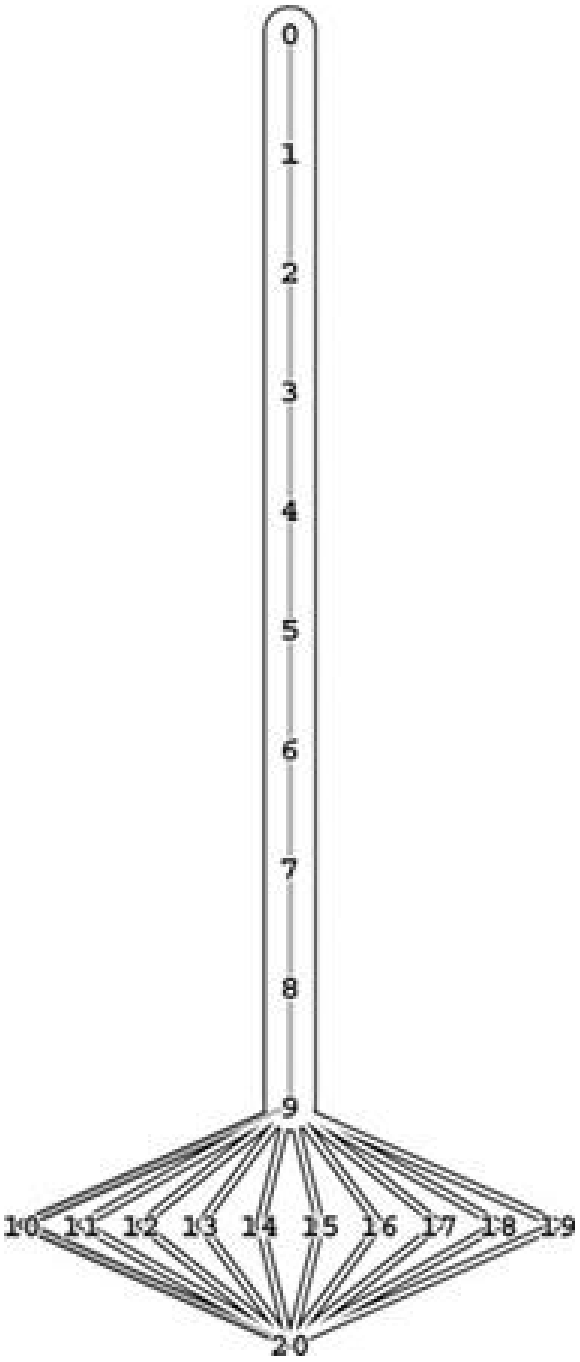


Figure 22.26 is a small example that illustrates the basic operations used in preflow-push

algorithms, in terms of the metaphor that we have been using, where we imagine that flow can go only down the page. Either we push excess flow out of an active vertex down an outgoing edge or we imagine that the vertex temporarily moves up so we can push excess flow back down an incoming edge.

Figure 22.27 is an example that illustrates why the preflow-push approach might be preferred to the augmenting-paths approach. In this network, any augmenting-path method successively puts a tiny amount of flow through a long `path` over and over again, slowly filling up the edges on the `path` until finally the maxflow is reached. By contrast, the preflow-push method fills up the edges on the long `path` as it traverses that `path` for the first time, then distributes that flow directly to the sink without traversing the long `path` again.

As we did in augmenting-path algorithms, we use the residual network (see Definition 22.4) to keep track of the edges that we might push flow through. Every edge in the residual network represents a potential place to push flow. If a residual-network edge is in the same direction as the corresponding edge in the flow network, we increase the flow; if it is in the opposite direction, we decrease the flow. If the increase fills the edge or the decrease empties the edge, the corresponding edge disappears from the residual network. For preflow-push algorithms, we use an additional mechanism to help decide which of the edges in the residual network can help us to eliminate active vertices.

Definition 22.6 A height function *for a given flow in a flow network is a set of nonnegative vertex weights h(0)... h (V-1) such that h (t) = 0 for the sink* `t` *and h (u) ≤ h (v) + 1 for every edge* `u-v` *in the residual network for the flow. An* **eligible edge** *is an edge* `u-v` *in the residual network with h (u) = h (v) + 1.*

A trivial height function, for which there are no eligible edges, is $h(0) = h(1) = \ldots = h(V-1) = 0$. If we set $h(s) = 1$, then any edge that emanates from the source and has flow corresponds to an eligible edge in the residual network.

We define a more interesting height function by assigning to each vertex the latter's shortest-path distance to the sink (its distance to the root in any BFS tree for the reverse of the network rooted at `t`, as illustrated in *Figure* 22.28). This height function is valid because $h(t) = 0$, and, for any pair of vertices u and v connected by an edge `u-v`, any shortest `path` to t starting with `u-v` is of length $h(v) + 1$; so the shortest-path length from `u` to `t`, or $h(u)$, must be less than or equal to that value. This function plays a special role because it puts each vertex at the maximum possible height. Working backward, we see that `t` has to be at height `0`; the only vertices that could be at height `1` are those with an edge directed to `t` in the residual network; the only vertices that could be at height `2` are

those that have edges directed to vertices that could be at height 1, and so forth.

Property 22.9 *For any flow and associated height function, a vertex's height is no larger than the length of the shortest* `path` *from that vertex to the sink in the residual network.*

Proof: For any given vertex `u`, let `d` be the shortest-path length from `u` to `t`, and let `u` = u_1 u_2`,...,` u_d `= t` be a shortest path. Then

$$\begin{aligned} h(u) = h(u_1) &\leq h(u_2) + 1 \\ &\leq h(u_3) + 2 \\ &\vdots \\ &\leq h(u_d) + d = h(t) + d = d \end{aligned}$$

■

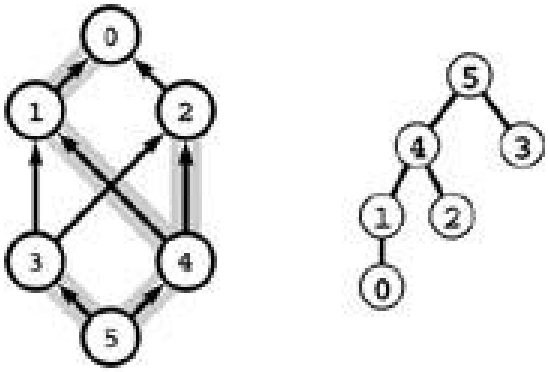
The intuition behind height functions is the following: When an active node's height is less than the height of the source, it is possible that there is some way to push flow from that node down to the sink; when an active node's height exceeds the height of the source, we know that that node's excess needs to be pushed back to the source. To establish this latter fact, we reorient our view of Property 22.9, where we thought about the length of the shortest `path` to the sink as an upper bound on the height; instead, we think of the height as a lower bound on the shortest-path length:

Corollary *If a vertex's height is greater than V, then there is no* `path` *from that vertex to the sink in the residual network.*

Proof: If there is a `path` from the vertex to the sink, the implication of Property 22.9 would be that the path's length is greater than *V*, but that cannot be true because the network has only *V* vertices.■

Figure 22.28 Initial Height Function *The tree at the right is a BFS tree rooted at 5 for the reverse of our sample network (left). The vertex-indexed array h gives the distance from each vertex to the root and is a valid height function: for every edge* `u-v` *in the network,* `h[u]` *is less than or equal to* `h[v]+1`.

IPR2023-00124
KASPERSKY EX 1036 Page 483



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

Now that we understand these basic mechanisms, the generic preflow-push algorithm is simple to describe. We start with any height function and assign zero flow to all edges except those connected to the source, which we fill to capacity. Then, we repeat the following step until no active vertices remain:

Choose an active vertex. Push flow through some eligible edge leaving that vertex (if any). If there are no such edges, increment the vertex's height.

We do not specify what the initial height function is, how to choose the active vertex, how to choose the eligible edge, or how much flow to push. We refer to this generic method as the *edge-based* preflow-push algorithm.

The algorithm depends on the height function to identify eligible edges. We also use the height function to prove that the algorithm computes a maxflow, and to analyze performance. Therefore, it is critical to ensure that the height function remains valid throughout the execution of the algorithm.

Property 22.10 *The edge-based preflow-push algorithm preserves the validity of the height function.*

Proof: We increment $h(u)$ only if there are no edges `u-v` with $h(u) = h(v)+1$. That is, $h(u) < h(v)+1$ for all edges `u-v` before incrementing $h(u)$, so $h(u) \le h(v) + 1$ afterward. For any incoming edges `w-u`, incrementing $h(u)$ certainly preserves the inequality $h(w) \le h(u) + 1$. Incrementing $h(u)$ does not affect inequalities corresponding to any other edge, and we never increment `h(t)` (or `h(s)`). Together, these observations imply the stated result.■

All the excess flow emanates from the source. Informally, the generic preflow-push algorithm tries to push the excess flow to the sink; if it cannot do so, it eventually pushes the

excess flow back to the source. It behaves in this manner because nodes with excess always stay connected to the source in the residual network.

Property 22.11 *While the preflow-push algorithm is in execution on a flow network, there exists a (directed)* `path` *in that flow network's residual network from each active vertex to the source, and there are no (directed) paths from source to sink in the residual network*.

Proof: By induction. Initially, the only flow is in the edges leaving the source, which are filled to capacity, so the destination vertices of those edges are the only active vertices. Since the edges are filled to capacity, there is an edge in the residual network from each of those vertices to the source and there are no edges in the residual network leaving the source. Thus, the stated property holds for the initial flow.

The source is reachable from every active vertex because the only way to add to the set of active vertices is to push flow from an active vertex down an eligible edge. This operation leaves an edge in the residual network from the receiving vertex back to the active vertex, from which the source is reachable, by the inductive hypothesis.

Initially, no other nodes are reachable from the source in the residual network. The first time that another node `u` becomes reachable from the source is when flow is pushed back along `u-s` (thus causing `s-u` to be added to the residual network). But this can happen only when `h(u)` is greater than `h(s)`, which can happen only after `h(u)` has been incremented, because there are no edges in the residual network to vertices with lower height. The same argument shows that all nodes reachable from the source have greater height. But the sink's height is always `0`, so it cannot be reachable from the source.■

Corollary *During the preflow-push algorithm, vertex heights are always less than* $2V$.

Proof: We need to consider only active vertices, since the height of each inactive vertex is either the same as or 1 greater than it was the last time that the vertex was active. By the same argument as in the proof of Property 22.9, the `path` from a given active vertex to the source implies that that vertex's height is at most $V - 2$ greater than the height of the source (`t` cannot be on the path). The height of the source never changes, and it is initially no greater than V. Thus, active vertices are of height at most $2V - 2$, and no vertex has height $2V$ or greater.■

The generic preflow-push algorithm is simple to state and implement. Less clear, perhaps, is why it computes a maxflow. The height function is the key to establishing that it does so.

Property 22.12 *The preflow-push algorithm computes a maxflow.*

Proof: First, we have to show that the algorithm terminates. There must come a point where there are no active vertices. Once we push all the excess flow out of a vertex, that vertex cannot become active again until some of that flow is pushed back; and that push-back can happen only if the height of the vertex is increased. If we have a sequence of active vertices of unbounded length, some vertex has to appear an unbounded number of times; and it can do so only if its height grows without bound, contradicting the corollary to Property 22.9.

When there are no active vertices, the flow is feasible. Since, by Property 22.11, there is also no `path` from source to sink in the residual network, the flow is a maxflow, by the same argument as that in the proof of Property 22.5.■

It is possible to refine the proof that the algorithm terminates to give an $O(V^2E)$ bound on its worst-case running time. We leave the details to exercises (see Exercises 22.67 through 22.68), in favor of the simpler proof in Property 22.13, which applies to a less general version of the algorithm. Specifically, the implementations that we consider are based on the following more specialized instructions for the iteration:

Choose an active vertex. Increase the flow along an eligible edge leaving that vertex (filling it if possible), continuing until the vertex becomes inactive or no eligible edges remain. In the latter case, increment the vertex's height.

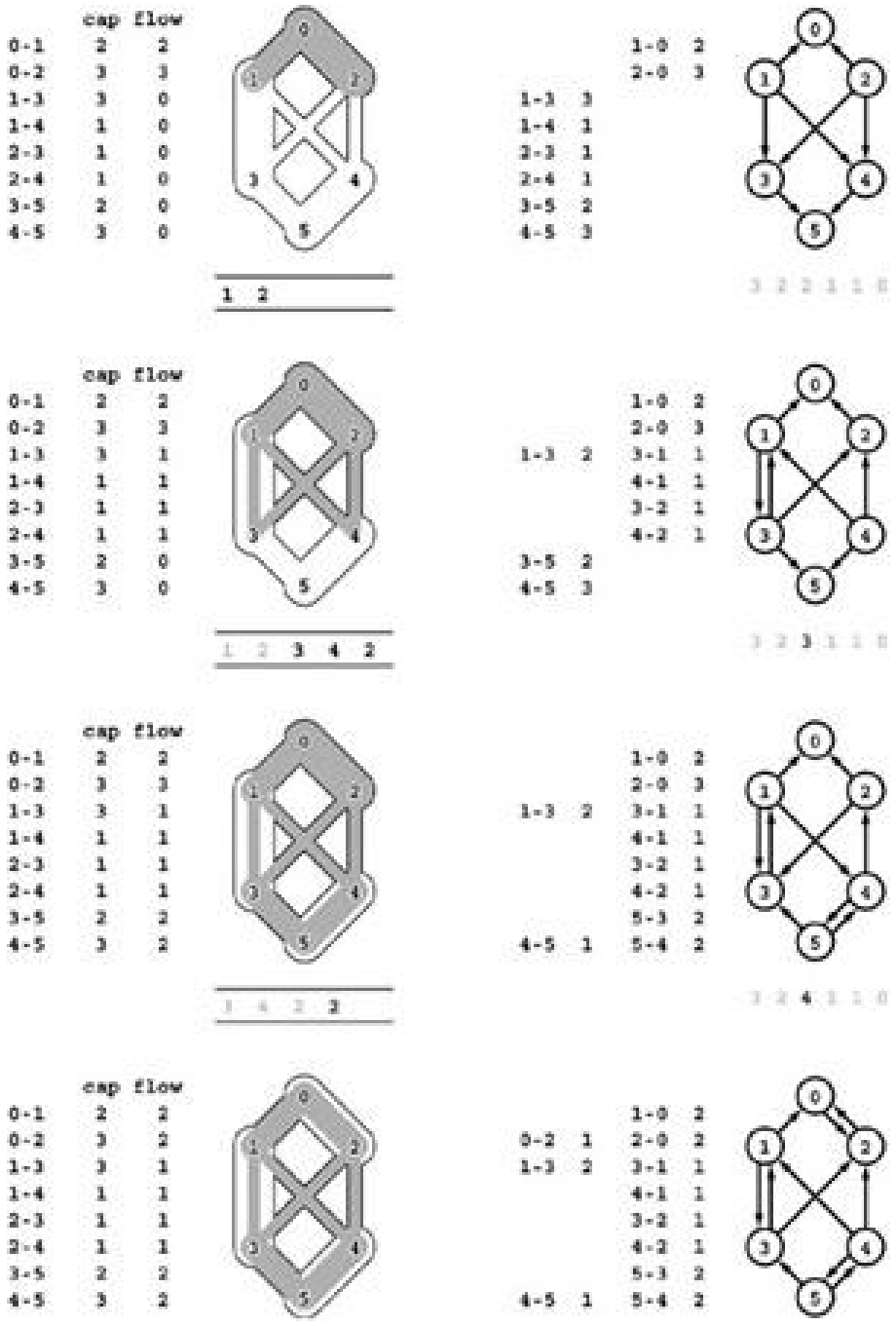
That is, once we have chosen a vertex, we push out of it as much flow as possible. If we get to the point that the vertex still has excess flow but no eligible edges remain, we increment the vertex's height. We refer to this generic method as the *vertex-based* preflow-push algorithm. It is a special case of the edge-based generic algorithm, where we keep choosing the same active vertex until it becomes inactive or we have used up all the eligible edges leaving it. The correctness proof in Property 22.12 applies to any implementation of the edge-based generic algorithm, so it immediately implies that the vertex-based algorithm computes a maxflow.

Program 22.4 is an implementation of the vertex-based generic algorithm that uses a generalized queue for the active vertices. It is a direct implementation of the method just described and represents a family of algorithms that differ only in their generalized-queue ADT implementation. This implementation assumes that the ADT disallows duplicate vertices on the queue; alternatively, we could add code to *Program* 22.4 to avoid enqueueing duplicates (see Exercises 22.62 and 22.63).

Figure 22.29 Residual Networks (FIFO Preflow-Push) *This figure shows the flow networks (left) and the residual networks (right) for each phase of the*

IPR2023-00124
KASPERSKY EX 1036 Page 486

FIFO preflow-push algorithm operating on our sample network. Queue contents are shown below the flow networks and distance labels below the residual networks. In the initial phase, we push flow through `0-1` *and* `0-2`*, thus making* `1` *and* `2` *active. In the second phase, we push flow from these two vertices to* `3` *and* `4`*, which makes them active and* `1` *inactive (*`2` *remains active and its distance label is incremented). In the third phase, we push flow through* `3` *and* `4` *to* `5`*, which makes them inactive (*`2` *still remains active and its distance label is again incremented). In the fourth phase,* `2` *is the only active node, and the edge* `2-0` *is admissible because of the distance-label increments, and one unit of flow is pushed back along* `2-0` *to complete the computation.*



Program 22.4 Preflow-push maxflow implementation

This ADT function implements the generic vertex-based preflow-push maxflow algorithm, using a generalized queue that disallows duplicates for active nodes.

The shortest-paths function GRAPHdist is used to initialize vertex heights in the array h, to shortest-paths distances from the sink. The wt array contains each vertex's excess flow and therefore implicitly defines the set of active vertices. By convention, s is initially active but never goes back on the queue and t is never active.

The main loop chooses an active vertex v, then pushes flow through each of its eligible edges (adding vertices that receive the flow to the active list, if necessary), until either v becomes inactive or all its edges have been considered. In the latter case, v's height is incremented and it goes back onto the queue.

```
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); }
```

```
            }
        }
```

Perhaps the simplest data structure to use for active vertices is a FIFO queue. *Figure 22.29* shows the operation of the algorithm on a sample network. As illustrated in the figure, it is convenient to break up the sequence of active vertices chosen into a sequence of *phases*, where a phase is defined to be the contents of the queue after all the vertices that were on the queue in the previous phase have been processed. Doing so helps us to bound the total running time of the algorithm.

Property 22.13 *The worst-case running time of the FIFO queue implementation of the preflow-push algorithm is proportional to* V^2E.■

Proof: We bound the number of phases using a *potential function*. This argument is a simple example of a powerful technique in the analysis of algorithms and data structures that we examine in more detail in Part 8.

Define the quantity φ to be 0 if there are no active vertices and to be the maximum height of the active vertices otherwise, then consider the effect of each phase on the value of φ. Let $h_0(s)$ be the initial height of the source. At the beginning, $\varphi = h_0(s)$; at the end, $\varphi = 0$.

First, we note that the number of phases where the height of some vertex increases is no more than $2V^2 - h_0(s)$, since each of the V vertex heights can be increased to a value of at most $2V$, by the corollary to Property 22.11. Since φ can increase only if the height of some vertex increases, the number of phases where φ increases is no more than $2V^2 - h_0(s)$.

If, however, no vertex's height is incremented during a phase, then φ must decrease by at least 1, since the effect of the phase was to push all excess flow from each active vertex to vertices that have smaller height.

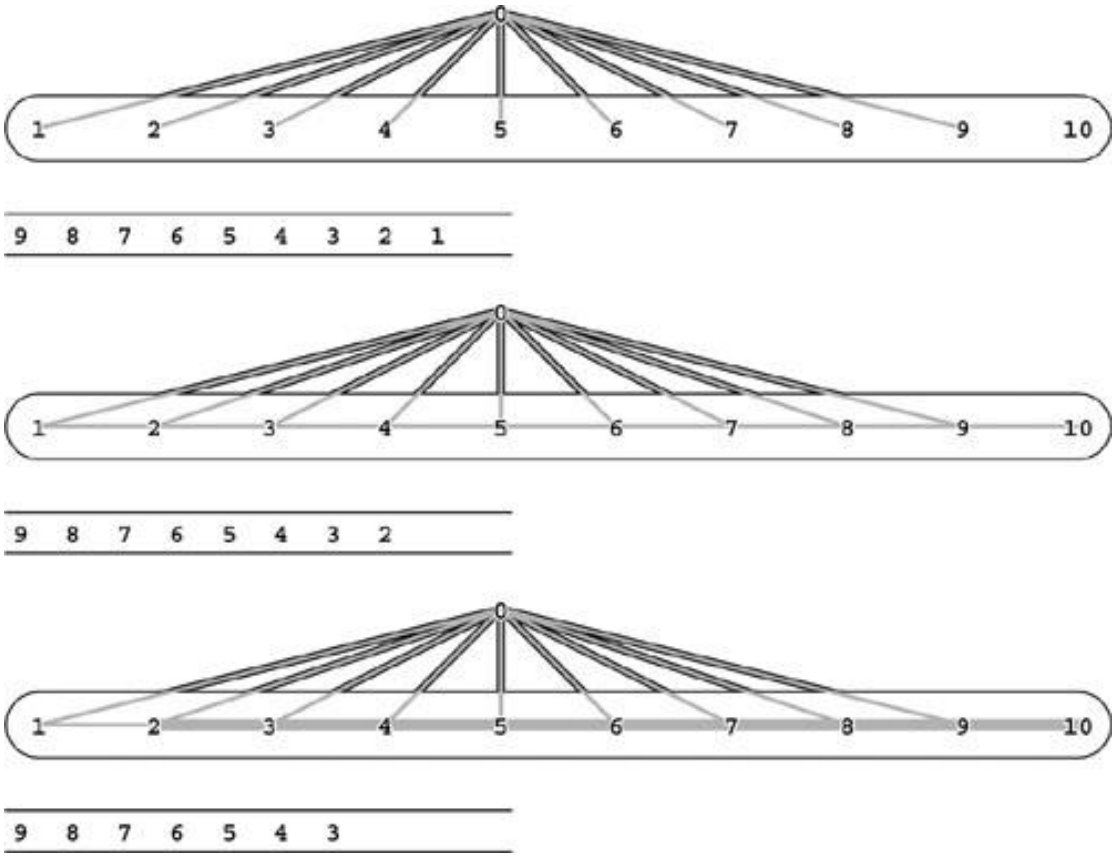
Together, these facts imply that the number of phases must be less than $4V^2$: The value of φ is $h_0(s)$ at the beginning and can be incremented at most $2V^2 - h_0(s)$ times and therefore can be decremented at most $2V^2$ times. The worst case for each phase is that all vertices are on the queue and all of their edges are examined, leading to the stated bound on the total running time.

This bound is tight. *Figure* 22.30 illustrates a family of flow networks for which the number of phases used by the preflow-push algorithm is proportional to V^2.

Figure 22.30 FIFO Preflow-Push Worst Case *This network represents a family of networks with V vertices such that the total running time of the preflow-push algorithm is proportional to* V^2. *It consists of unit-capacity edges emanating from the source (vertex 0) and horizontal edges of capacity* $v - 2$ *running from left to right towards the sink (vertex 10). In the initial phase of the preflow-push algorithm (top), we push one unit of flow out each edge from the source, making all the vertices active except the source and the sink. In a standard adjacency-lists repre-*

IPR2023-00124
KASPERSKY EX 1036 Page 490

sentation, they appear on the FIFO queue of active vertices in reverse order, as shown below the network. In the second phase (center), *we push one unit of flow from* 9 *to* 10, *making* 9 *inactive (temporarily); then we push one unit of flow from* 8 *to* 9, *making* 8 *inactive (temporarily) and making* 9 *active; then we push one unit of flow from* 7 *to* 8, *making* 7 *inactive (temporarily) and making* 8 *active; and so forth. Only* 1 *is left inactive. In the third phase* (bottom), *we go through a similar process to make* 2*inactive, and the same process continues for* $V - 2$ *phases.*



Because our implementations maintain an implicit representation of the residual network, they examine edges leaving a vertex even when those edges are not in the residual network (to test whether or not they are there). It is possible to show that we can reduce the bound in Property 22.13 from V^2E to V^3 for an implementation that eliminates this cost by maintaining an explicit representation of the residual network. Although the theoretical bound is the lowest that we have seen for the maxflow problem, this change may not be worth the trouble, particularly for the sparse graphs that we see in practice (see Exercises 22.64 through 22.66).

Again, these worst-case bounds tend to be pessimistic and thus not necessarily useful for predicting performance on real networks (though the gap is not as excessive as we found for augmenting-path algorithms). For example, the FIFO algorithm finds the flow in the network illustrated in *Figure* 22.31 in 15 phases, whereas the bound in the proof of Property 22.13 says only that it must do so in fewer than 182.

IPR2023-00124
KASPERSKY EX 1036 Page 491

To improve performance, we might try using a stack, a randomized queue, or any other generalized queue in *Program* 22.4. One approach that has proved to do well in practice is to implement the generalized queue such that `GQget` returns the highest active vertex. We refer to this method as the *highest-vertex preflow-push* maxflow algorithm. We can implement this strategy with a priority queue, although it is also possible to take advantage of the particular properties of heights to implement the generalized-queue operations in constant time. A worst-case time bound of $V^2\sqrt{E}$ (which is $V^{5/2}$ for sparse graphs) has been proved for this algorithm (*see reference section*); as usual, this bound is pessimistic. Many other preflow-push variants have been proposed, several of which reduce the worst-case time bound to be close to VE (*see reference section*).

Figure 22.31 Preflow-Push Algorithm (FIFO) *This sequence illustrates how the FIFO implementation of the preflow-push method finds a maximum flow in a sample network. It proceeds in phases: First it pushes as much flow as it can from the source along the edges leaving the source (top left). Then, it pushes flow from each of those nodes, continuing until all nodes are in equilibrium.*

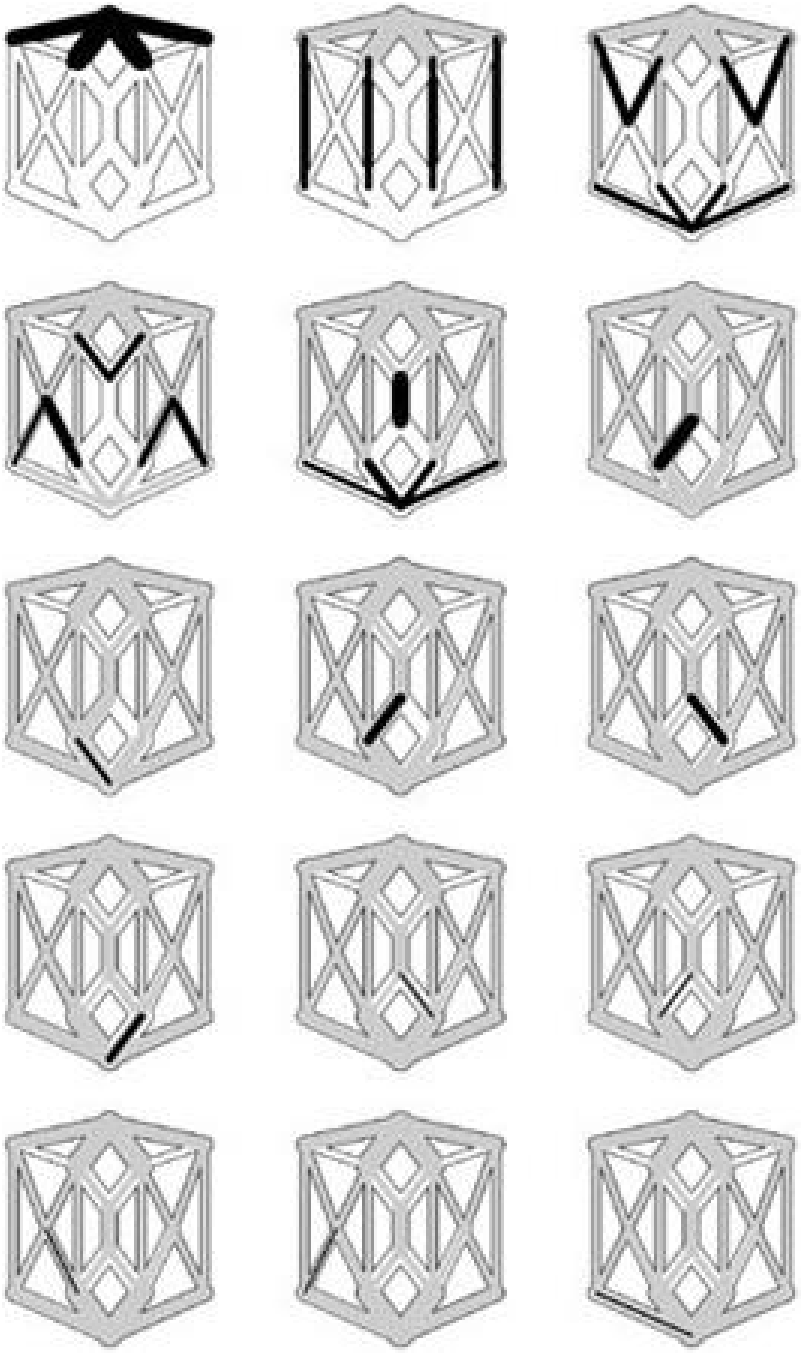


Table 22.2 Empirical study of preflow-push algorithms This table shows performance parameters (number of vertices expanded and number of adjacency-list nodes touched) for various preflow-push network-flow algorithms for our sample Euclidean neighbor network (with random capacities with maxflow value 286) and with unit capacities (with maxflow value 6). Differences among the methods are minimal for both types of networks. For random capacities, the number of edges examined is about the same as for the random augmenting-path algorithm (see Table 22.1). For unit capacities, the augmenting-path algorithms examine substantially fewer edges for these networks.

IPR2023-00124
KASPERSKY EX 1036 Page 493

	vertices	edges
random capacity 1-50		
shortest	2450	57746
depth-first	2476	58258
random	2363	55470
capacity 1		
shortest	1192	28356
depth-first	1234	29040
random	1390	33018

Table 22.2 shows performance results for preflow-push algorithms corresponding to those for augmenting-path algorithms in Table 22.1, for the two network models discussed in Section 22.2. These experiments show much less performance variation for the various preflow-push methods than we observed for augmenting-path methods.

There are many options to explore in developing preflow-push implementations. We have already discussed three major choices:

- Edge-based versus vertex-based generic algorithm
- Generalized queue implementation
- Initial assignment of heights

There are several other possibilities to consider and many options to try for each, leading to a multitude of different algorithms to study (see, for example, Exercises 22.57 through 22.61). The dependence of an algorithm's performance on characteristics of the input network further multiplies the possibilities.

The two generic algorithms that we have discussed (augmenting-path and preflow-push) are among the most important from an extensive research literature on maxflow algorithms. The quest for better maxflow algorithms is still a potentially fruitful area for further research. Researchers are motivated to develop and study new algorithms and implementations by the reality of faster algorithms for practical problems and by the possibility that a simple linear algorithm exists for the maxflow problem. Until one is discovered, we

IPR2023-00124
KASPERSKY EX 1036 Page 494

can work confidently with the algorithms and implementations that we have discussed; numerous studies have shown them to be effective for a broad range of practical maxflow problems.

Exercises

▷**22.51** Describe the operation of the preflow-push algorithm in a network whose capacities are in equilibrium.

22.52 Use the concepts described in this section (height functions, eligible edges, and pushing of flow through edges) to describe augmenting-path maxflow algorithms.

22.53 Show, in the style of *Figure* 22.29, the flow and residual networks after each phase when you use the FIFO preflow-push algorithm to find a maxflow in the flow network shown in *Figure* 22.11.

22.54 Do Exercise 22.53 for the highest-vertex preflow-push algorithm.

○**22.55** Modify *Program* 22.4 to implement the highest-vertex preflow-push algorithm, by implementing the generalized queue as a priority queue. Run empirical tests so that you can add a line to Table 22.2 for this variant of the algorithm.

22.56 Plot the number of active vertices and the number of vertices and edges in the residual network as the FIFO preflow-push algorithm proceeds, for specific instances of various networks (see Exercises 22.7–12).

○**22.57** Implement the generic edge-based preflow-push algorithm, using a generalized queue of eligible edges. Run empirical studies for various networks (see Exercises 22.7–12) to compare these to the basic methods, studying generalized-queue implementations that perform better than others in more detail.

22.58 Modify *Program* 22.4 to recalculate the vertex heights periodically to be shortest-path lengths to the sink in the residual network.

○**22.59** Evaluate the idea of pushing excess flow out of vertices by spreading it evenly among the outgoing edges, rather than perhaps filling some and leaving others empty.

22.60 Run empirical tests to determine whether the shortest-paths computation for the initial height function is justified in *Program* 22.4 by comparing its performance as given for various networks (see Exercises 22.7–12) with its performance when the vertex heights are just initialized to zero.

○**22.61** Experiment with hybrid methods involving combinations of the ideas above.

IPR2023-00124
KASPERSKY EX 1036 Page 495

Run empirical studies for various networks (see Exercises 22.7–12) to compare these to the basic methods, studying variations that perform better than others in more detail.

22.62 Modify the implementation of *Program* 22.4 such that it explicitly disallows duplicate vertices on the generalized queue. Run empirical tests for various networks (see Exercises 22.7–12) to determine the effect of your modification on actual running times.

22.63 What effect does allowing duplicate vertices on the generalized queue have on the worst-case running-time bound of Property 22.13?

22.64 Modify the implementation of *Program* 22.4 to maintain an explicit representation of the residual network.

○ **22.65** Sharpen the bound in Property 22.13 to $O(V^3)$ for the implementation of Exercise 22.64. *Hint*: Prove separate bounds on the number of pushes that correspond to deletion of edges in the residual network and on the number of pushes that do not result in full or empty edges.

22.66 Run empirical studies for various networks (see Exercises 22.7–12) to determine the effect of using an explicit representation of the residual network (see Exercise 22.64) on actual running times.

22.67 For the edge-based generic preflow-push algorithm, prove that the number of pushes that correspond to deleting an edge in the residual network is less than $2VE$. Assume that the implementation keeps an explicit representation of the residual network.

● **22.68** For the edge-based generic preflow-push algorithm, prove that the number of pushes that do not correspond to deleting an edge in the residual network is less than $4V^2(V + E)$. *Hint*: Use the sum of the heights of the active vertices as a potential function.

● **22.69** Run empirical studies to determine the actual number of edges examined and the ratio of the running time to V for several versions of the preflow-push algorithm for various networks (see Exercises 22.7–12). Consider various algorithms described in the text and in the previous exercises, and concentrate on those that perform the best on huge sparse networks. Compare your results with your result from Exercise 22.36.

● **22.70** Write a client program that does dynamic graphical animations of preflow-push algorithms. Your program should produce images like *Figure* 22.31 and the other figures in this section (see Exercise 22.50). Test your implementation for the Euclidean networks among Exercises 22.7–12.

IPR2023-00124
KASPERSKY EX 1036 Page 496

22.4 MAXFLOW REDUCTIONS

In this section, we consider a number of reductions to the maxflow problem, to demonstrate that the maxflow algorithms of Sections 22.2 and 22.3 are important in a broad context. We can remove various restrictions on the network and solve other flow problems; we can solve other network- and graph-processing problems; and we can solve problems that are not network problems at all. This section is devoted to examples of such uses—to establish maxflow as a general problem-solving model.

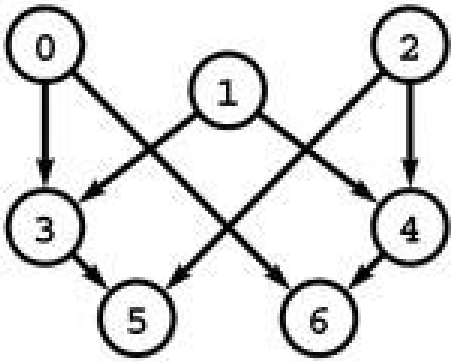
We also consider relationships between the maxflow problem and problems that are more difficult, to set the context for considering those problems later on. In particular, we note that the maxflow problem is a special case of the mincost-flow problem that is the subject of Sections 22.5 and 22.6, and we describe how to formulate maxflow problems as LP problems, which we will address in Part 8. Mincost flow and LP represent problem-solving models that are more general than the maxflow model. Although we normally can solve maxflow problems more easily with the specialized algorithms of Sections 22.2 and 22.3 than with algorithms that solve these more general problems, it is important to be cognizant of the relationships among problem-solving models as we progress to more powerful ones.

We use the term *standard maxflow problem* to refer to the version of the problem that we have been considering (maxflow in edge-capacitated *st*-networks). This usage is solely for easy reference in this section. Indeed, we begin by considering reductions that show the restrictions in the standard problem to be essentially immaterial, because several other flow problems reduce to or are equivalent to the standard problem. We could adopt any of the equivalent problems as the "standard" problem. A simple example of such a problem, already noted as a consequence of Property 22.1, is that of finding a circulation in a network that maximizes the flow in a specified edge. Next, we consider other ways to pose the problem, in each case noting its relationship to the standard problem.

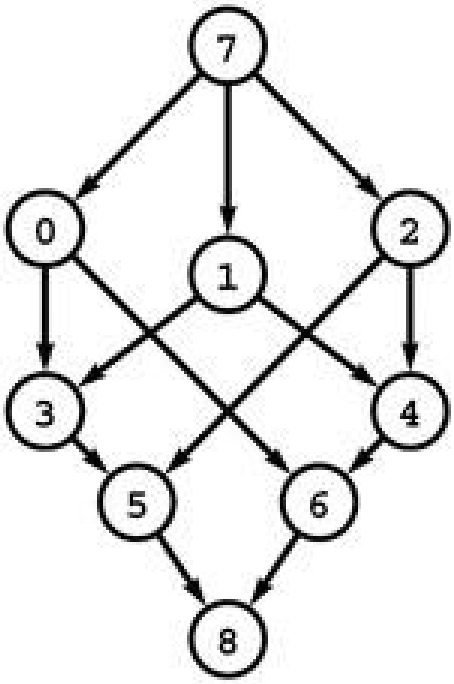
Figure 22.32 Reduction from Multiple Sources and Sinks *The network at the top has three sources* (0, 1, *and* 2) *and two sinks* (5 *and* 6). *To find a flow that maximizes the total flow out of the sources and into the sinks, we find a maxflow in the st-network illustrated at the bottom. This network is a copy of the original network, with the addition of a new source* 7 *and a new sink* 8. *There is an edge from* 7 *to each original-network source with capacity equal to the sum of the capacities of that source's outgoing edges and an edge from each original-network sink to* 8 *with capacity equal to the sum of the capacities of that sink's incoming edges*.

IPR2023-00124
KASPERSKY EX 1036 Page 497

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



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



Maxflow in general networks Find the flow in a network that maximizes the total outflow from its sources (and therefore the total inflow to its sinks). By convention, define the flow to be zero if there are no sources or no sinks.

Property 22.14 *The maxflow problem for general networks is equivalent to the maxflow problem for st-networks.*

Proof: A maxflow algorithm for general networks will clearly work for *st*-networks, so we just need to establish that the general problem reduces to the *st*-network problem. To do so, first find the sources and sinks (using, for example, the method that we used to initialize the queue in *Program* 19.8), and return zero if there are none of either. Then, add a dummy source vertex s and edges from s to each source in the network (with each such

edge's capacity set to that edge's destination vertex's outflow) and a dummy sink vertex `t` and edges from each sink in the network to `t` (with each such edge's capacity set to that edge's source vertex's inflow) *Figure* 22.32 illustrates this reduction. Any maxflow in the *st*-network corresponds directly to a maxflow in the original network.■

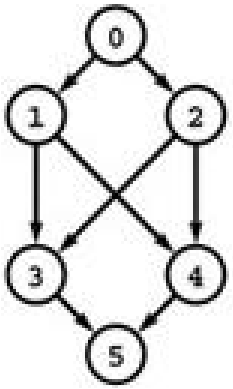
Vertex-capacity constraints Given a flow network, find a maxflow satisfying additional constraints specifying that the flow through each vertex must not exceed some fixed capacity.

Property 22.15 *The maxflow problem for flow networks with capacity constraints on vertices is equivalent to the standard maxflow problem.*

Proof: Again, we could use any algorithm that solves the capacity-constraint problem to solve a standard problem (by setting capacity constraints at each vertex to be larger than its inflow or outflow), so we need only to show a reduction to the standard problem. Given a flow network with capacity constraints, construct a standard flow network with two vertices `u` and `u*` corresponding to each original vertex `u`, with all incoming edges to the original vertex going to `u`, all outgoing edges coming from `u*`, and an edge `u-u*` of capacity equal to the vertex capacity. This construction is illustrated in *Figure* 22.33. The flows in the edges of the form `u*-v` in any maxflow for the transformed network give a maxflow for the original network that must satisfy the vertex-capacity constraints because of the edges of the form `u-u*`.■

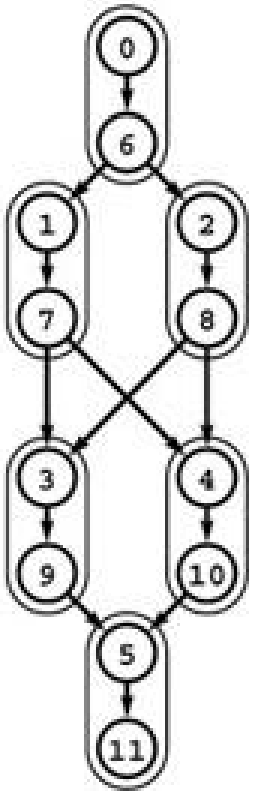
Figure 22.33 Removing Vertex Capacities *To solve the problem of finding a maxflow in the network at the top such that flow through each vertex does not exceed the capacity bound given in the vertex-indexed array* `capV`*, we build the standard network at the bottom: Associate a new vertex* `u*` (*where* `u*` *denotes* `u+V`*) with each vertex* `u`*, add an edge* `u-u*` *whose capacity is the capacity of* `u`*, and include an edge* `u*-v` *for each edge* `u-v`*. Each* `u-u*` *pair is encircled in the diagram. Any flow in the bottom network corresponds directly to a flow in the top network that satisfies the vertex-capacity constraints.*

	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



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

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



Allowing multiple sinks and sources or adding capacity constraints seem to generalize the maxflow problem; the interest of Properties 22.14 and 22.15 is that these problems are actually no more difficult than the standard problem. Next, we consider a version of the problem that might seem to be easier to solve:

Acyclic networks Find a maxflow in an acyclic network. Does the presence of cycles in a flow network make more difficult the task of computing a maxflow?

We have seen many examples of digraph-processing problems that are much more difficult to solve in the presence of cycles. Perhaps the most prominent example is the short-

est-paths problem in weighted digraphs with edge weights that could be negative (see section 21.7), which is simple to solve in linear time if there are no cycles but NP-complete if cycles are allowed. But the maxflow problem, remarkably, is no easier for acyclic networks.

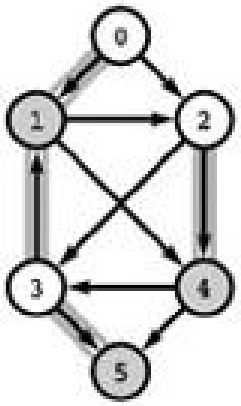
Property 22.16 *The maxflow problem for acyclic networks is equivalent to the standard maxflow problem.*

Proof: Again, we need only to show that the standard problem reduces to the acyclic problem. Given any network with V vertices and E edges, we construct a network with $2V + 2$ vertices and $E + 3V$ edges that is not just acyclic but has a simple structure.

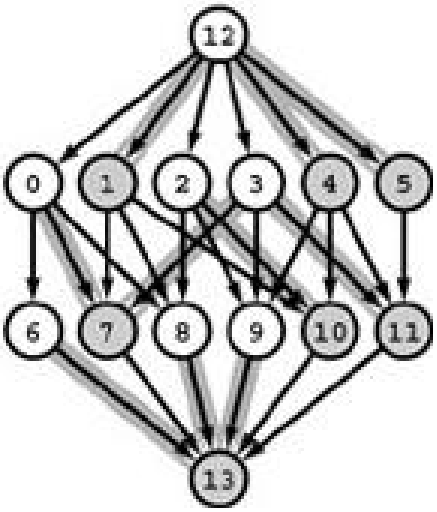
Let `u*` denote `u+V`, and build a bipartite digraph consisting of two vertices `u` and `u*` corresponding to each vertex `u` in the original network and one edge `u-v*` corresponding to each edge `u-v` in the original network with the same capacity. Now, add to the bipartite digraph a source `s` and a sink `t` and, for each vertex `u` in the original graph, an edge `s-u` and an edge `u*-t`, both of capacity equal to the sum of the capacities of `u`'s outgoing edges in the original network. Also, let X be the sum of the capacities of the edges in the original network, and add edges from `u` to `u*`, with capacity $X + 1$. This construction is illustrated in *Figure* 22.34.

Figure 22.34 Reduction to Acyclic Network *Each vertex* `u` *in the top network corresponds to two vertices* `u` and `u*` (*where* `u*` *denotes* `u+V`*) in the bottom network and each edge* `u-v` *in the top network corresponds to an edge* `u-v*` *in the bottom network. Additionally, the bottom network has uncapacitated edges* `u-u*`, *a source* `s` *with an edge to each unstarred vertex and a sink* `t` *with an edge from each starred vertex. The shaded and unshaded vertices (and edges which connect shaded to unshaded) illustrate the direct relationship among cuts in the two networks (see text).*

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




```
0-6   25    12-0  5    6-13  5
1-7   25    12-1  5    7-13  5
2-8   25    12-2  3    8-13  3
3-9   25    12-3  5    9-13  5
4-10  25    12-4  6   10-13  6
5-11  25    12-5  0   11-13  0
0-7   2
0-8   3
1-8   3
1-10  2
2-9   2
2-10  1
3-7   3
3-11  2
4-9   3
4-11  3
```



To show that any maxflow in the original network corresponds to a maxflow in the transformed network, we consider cuts rather than flows. Given any *st*-cut of size *c* in the original network, we show how to construct an *st*-cut of size $c + X$ in the transformed network; and, given any minimal *st*-cut of size $c + X$ in the transformed network, we show how to construct an *st*-cut of size *c* in the original network. Thus, given a minimal cut in the transformed network, the corresponding cut in the original network is minimal. Moreover, our construction gives a flow whose value is equal to the minimal-cut capacity, so it is a maxflow.

Given any cut of the original network that separates the source from the sink, let *S* be the

source's vertex set and *T* the sink's vertex set. Construct a cut of the transformed network by putting vertices in *S* in a set with `s` and vertices in *T* in a set with `t` and putting `u` and `u*` on the same side of the cut for all `u`, as illustrated in *Figure* 22.34. For every `u`, either `s-u` or `P|u*-t|` is in the cut set, and `u-v*` is in the cut set if and only if `u-v` is in the cut set of the original network; so the total capacity of the cut is equal to the capacity of the cut in the original network plus *X*.

Given any minimal *st*-cut of the transformed network, let *S** be `s`'s vertex set and *T* `t`'s vertex set. Our goal is to construct a cut of the same capacity with `u` and `u*` both in the same cut vertex set for all `u`, so that the correspondence of the previous paragraph gives a cut in the original network, completing the proof. First, if `u` is in *S** and `u*` in *T*, then `u-u*` must be a crossing edge, which is a contradiction: `u-u*` cannot be in any minimal cut, because a cut consisting of all the edges corresponding to the edges in the original graph is of lower cost. Second, if `u` is in *T* and `u*` is in *S**, then `s-u` must be in the cut, because that is the only edge connecting `s` *to* `u`. But we can create a cut of equal cost by substituting all the edges directed out of `u` for `s-u`, moving `u` to *S**.

Given any flow in the transformed network of value $c + X$, we simply assign the same flow value to each corresponding edge in the original network to get a flow with value *c*. The cut transformation at the end of the previous paragraph does not affect this assignment, because it manipulates edges with flow value zero.■

The result of the reduction not only is an acyclic network, but also has a simple bipartite structure. The reduction says that we could, if we wished, adopt these simpler networks, rather than general networks, as our standard. It would seem that perhaps this special structure would lead to faster maxflow algorithms. But the reduction shows that we could use any algorithm that we found for these special acyclic networks to solve maxflow problems in general networks, at only modest extra cost. Indeed, the classical maxflow algorithms exploit the flexibility of the general network model: Both the augmenting-path and preflow-push approaches that we have considered use the concept of a residual network, which involves introducing cycles into the network. When we have a maxflow problem for an acyclic network, we typically use the standard algorithm for general networks to solve it.

The construction of Property 22.16 is elaborate, and it illustrates that reduction proofs can require care, if not ingenuity. Such proofs are important because not all versions of the maxflow problem are equivalent to the standard problem, and we need to know the extent of the applicability of our algorithms. Researchers continue to explore this topic because reductions relating various natural problems have not yet been established, as illustrated by the following example.

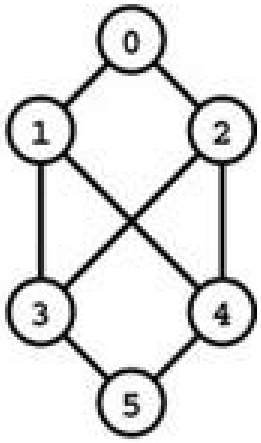
Maxflow in undirected networks An undirected flow network is a weighted graph with integer edge weights that we interpret to be capacities. A circulation in such a network is an assignment of weights and directions to the edges satisfying the conditions that the flow on each edge is no greater than that edge's capacity and that the total flow into each vertex is equal to the total flow out of that vertex. The undirected maxflow problem is to find a circulation that maximizes the flow in specified direction in a specified edge (that is, from some vertex `s` to some other vertex `t`). This problem perhaps corresponds more naturally than the standard problem to our liquid-flowing-through-pipes model: It corresponds to allowing liquid to flow through a pipe in either direction.

Property 22.17 *The maxflow problem for undirected st-networks reduces to the maxflow problem for st-networks.*

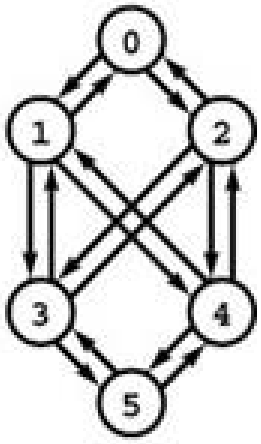
Figure 22.35 Reduction from Undirected Networks *To solve a maxflow problem in an undirected network, we can consider it to be a directed network with edges in each direction.*

IPR2023-00124
KASPERSKY EX 1036 Page 504

	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



Proof: Given an undirected network, construct a directed network with the same vertices and two directed edges corresponding to each edge, one in each direction, both with the capacity of the undirected edge. Any flow in the original network certainly corresponds to a flow with the same value in the transformed network. The converse is also true: If u-v has flow f and v-u flow g in the undirected network, then we can put the flow $f - g$ in u-v in the directed network if $f \geq g$; $g - f$ in v-u otherwise. Thus, any maxflow in the directed network is a maxflow in the undirected network: The construction gives a flow, and any flow with a higher value in the directed network would correspond to some flow with a higher value in the undirected network; but no such flow exists.■

This proof does not establish that the problem for undirected networks is equivalent to the standard problem. That is, it leaves open the possibility that finding maxflows in undirected networks is easier than finding maxflows in standard networks (see Exercise 22.83).

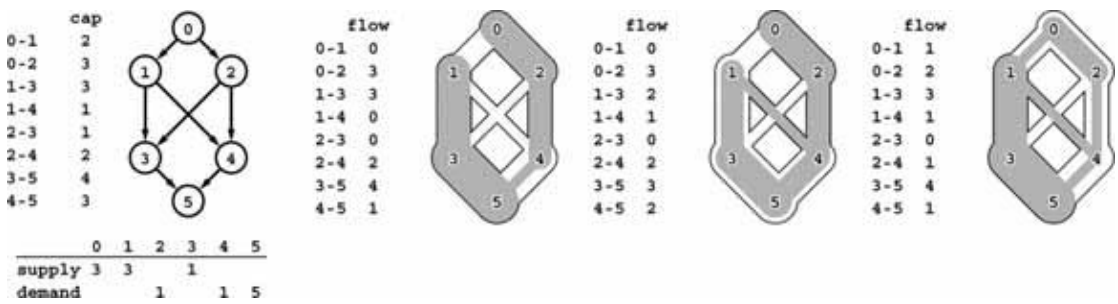
In summary, we can handle networks with multiple sinks and sources, undirected networks, networks with capacity constraints on vertices, and many other types of networks (see, for example, Exercise 22.81) with the maxflow algorithms for *st*-networks in the previous two sections. In fact, Property 22.16 says that we could solve all these problems with even an algorithm that works for only acyclic networks.

Next, we consider a problem that is not an explicit maxflow problem but that we can reduce to the maxflow problem and therefore solve with maxflow algorithms. It is one way to formalize a basic version of the merchandise-distribution problem described at the beginning of this chapter.

Feasible flow Suppose that a weight is assigned to each vertex in a flow network, to be interpreted as supply (if positive) or demand (if negative), with the sum of the vertex weights equal to zero. Define a flow to be *feasible* if the difference between each vertex's outflow and inflow is equal to that vertex's weight (supply if positive and demand if negative). Given such a network, determine whether or not a feasible flow exists. *Figure* 22.36 illustrates a feasible-flow problem.

Supply vertices correspond to warehouses in the merchandise-distribution problem; demand vertices correspond to retail outlets; and edges correspond to roads on the trucking routes. The feasible-flow problem answers the basic question of whether it is possible to find a way to ship the goods such that supply meets demand everywhere.

Figure 22.36 Feasible Flow *In a feasible-flow problem, we specify supply and demand constraints at the vertices in addition to the capacity constraints on the edges. We seek any flow for which outflow equals supply plus inflow at supply vertices and inflow equals outflow plus demand at demand vertices. Three solutions to the feasible-flow problem at left are shown on the right.*



Property 22.18 *The feasible-flow problem reduces to the maxflow problem.*

Proof: Given a feasible-flow problem, construct a network with the same vertices and edges but with no weights on the vertices. Instead, add a source vertex `s` that has an edge to each supply vertex with weight equal to that vertex's supply and a sink vertex `t` that has an edge from each demand vertex with weight equal to the negation of that vertex's demand (so that the edge weight is positive). Solve the maxflow problem on this network. The original network has a feasible flow if and only if all the edges out of the source and all the edges into the sink are filled to capacity in this flow. *Figure* 22.37 illustrates an example of this reduction.■

Developing ADT functions that implement reductions of the type that we have been considering can be a challenging software-engineering task, primarily because the objects that we are manipulating are represented with complicated data structures. To reduce another problem to a standard maxflow problem, should we create a new network? Some of the problems require extra data, such as vertex capacities or supply and demand, so creating a standard network without these data might be justified. But if we do so and then compute a maxflow, what are we to do with the result? Transferring the computed flow (a weight on each edge) from one network to another when both are represented with adjacency lists is not a trivial computation. If we do not make a copy of the network, we certainly may disturb it by adding dummy vertices and computing a maxflow; so we need to take care to restore the network to its original state after the computation. *Program* 22.5 is an implementation that illustrates some of these considerations in a network ADT function for the feasible-flow problem using the reduction of Property 22.16.

Program 22.5 Feasible flow via reduction to maxflow

This ADT function solves the feasible-flow problem by reduction to maxflow, using the construction illustrated in *Figure* 22.37. It assumes that the graph ADT representation has a vertex-indexed array `sd` such that `sd[i]` represents, if it is positive, the supply at vertex `i` and, if it is negative, the demand at vertex `i`.

As indicated in the construction, we add edges from a dummy node `s` to the supply nodes and from the demand nodes to another dummy node `t`, then find a maxflow, then check whether all the extra edges are filled to capacity, then remove all the extra edges.

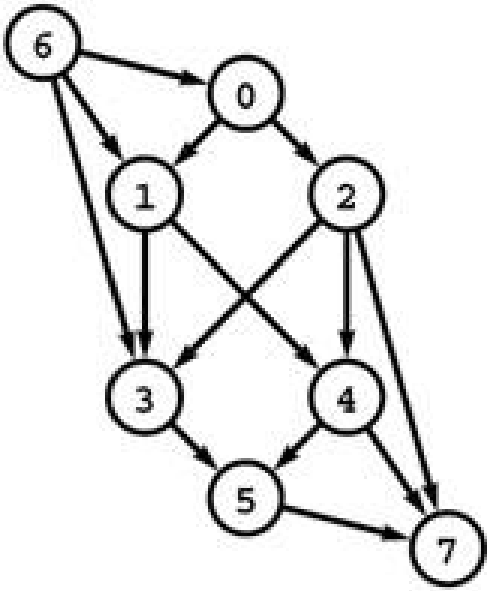
IPR2023-00124
KASPERSKY EX 1036 Page 507

This code assumes that the vertex-indexed arrays in the graph representation are large enough to accommodate the dummy nodes. It also uses a shortcut that depends upon the representation to remove the edges and does not free the memory for the list nodes corresponding to the edges that are added and removed (see Exercise 22.79).

```
  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;
  }
```

Figure 22.37 Reduction from Feasible Flow *This network is a standard network constructed from the feasible-flow problem in Figure 22.36 by adding edges from a new source vertex to the supply vertices (each with capacity equal to the amount of the supply) and edges to a new sink vertex from the demand vertices (each with capacity equal to the amount of the demand). The network in Figure 22.36 has a feasible flow if and only if this network has a flow (a maxflow) that fills all the edges from the sink and all the edges to the source.*

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



A canonical example of a flow problem that we cannot handle with the maxflow model, and that is the subject of Sections 22.5 and 22.6, is an extension of the feasible-flow problem. We add a second set of edge weights that we interpret as costs, define flow costs in terms of these weights, and ask for a feasible flow of minimal cost. This model formalizes the general merchandise-distribution problem. We are interested not just in whether it is possible to move the goods, but also in what is the lowest-cost way to move them.

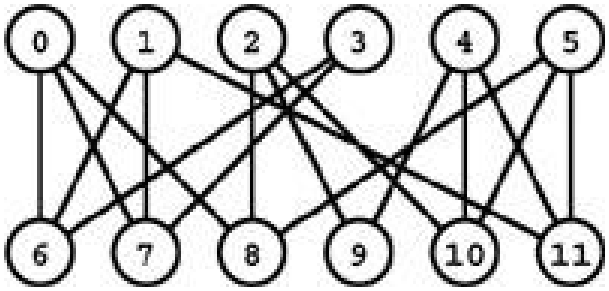
All the problems that we have considered so far in this section have the same basic goal (computing a flow in a flow network), so it is perhaps not surprising that we can handle them with a flow-network problem-solving model. As we saw with the maxflow–mincut theorem, we can use maxflow algorithms to solve graph-processing problems that seem to have little to do with flows. We now turn to examples of this kind.

Maximum-cardinality bipartite matching Given a bipartite graph, find a set of edges of maximum cardinality such that each vertex is connected to at most one other vertex.

For brevity, we refer to this problem simply as the *bipartite-matching* problem except in contexts where we need to distinguish it from similar problems. It formalizes the job-placement problem discussed at the beginning of this chapter. Vertices correspond to individuals and employers; edges correspond to a "mutual interest in the job" relation. A solution to the bipartite-matching problem maximizes total employment. *Figure* 22.38 illustrates the bipartite graph that models the example problem in *Figure* 22.3.

It is an instructive exercise to think about finding a direct solution to the bipartite-matching problem, without using the graph model. For example, the problem amounts to the following combinatorial puzzle: "Find the largest subset of a set of pairs of integers (drawn from disjoint sets) with the property that no two pairs have the same integer." The example depicted in *Figure* 22.38 corresponds to solving this puzzle on the pairs `0-6`, `0-7`, `0-8`, `1-6`, and so forth. The problem seems straightforward at first, but, as was true of the Hamilton-path problem that we considered in Section 17.7 (and many other problems), a naive approach that chooses pairs in some systematic way until finding a contradiction might require exponential time. That is, there are far too many subsets of the pairs for us to try all possibilities; a solution to the problem must be clever enough to examine only a few of them. Solving specific matching puzzles like the one just given or developing algorithms that can solve efficiently any such puzzle are nontrivial tasks that help to demonstrate the power and utility of the network-flow model, which provides a reasonable way to do bipartite matching.

Figure 22.38 Bipartite Matching *This instance of the bipartite-matching problem formalizes the job-placement example depicted in Figure 22.3. Finding the best way to match the students to the jobs in that example is equivalent to finding the maximum number of vertex-disjoint edges in this bipartite graph.*



Property 22.19 *The bipartite-matching problem reduces to the maxflow problem.*

Proof: Given a bipartite-matching problem, construct an instance of the maxflow problem by directing all edges from one set to the other, adding a source vertex with edges directed to all the members of one set in the bipartite graph, and adding a sink vertex with edge directed from all the members of the other set. To make the resulting digraph a network, assign each edge capacity 1. *Figure* 22.39 illustrates this construction.

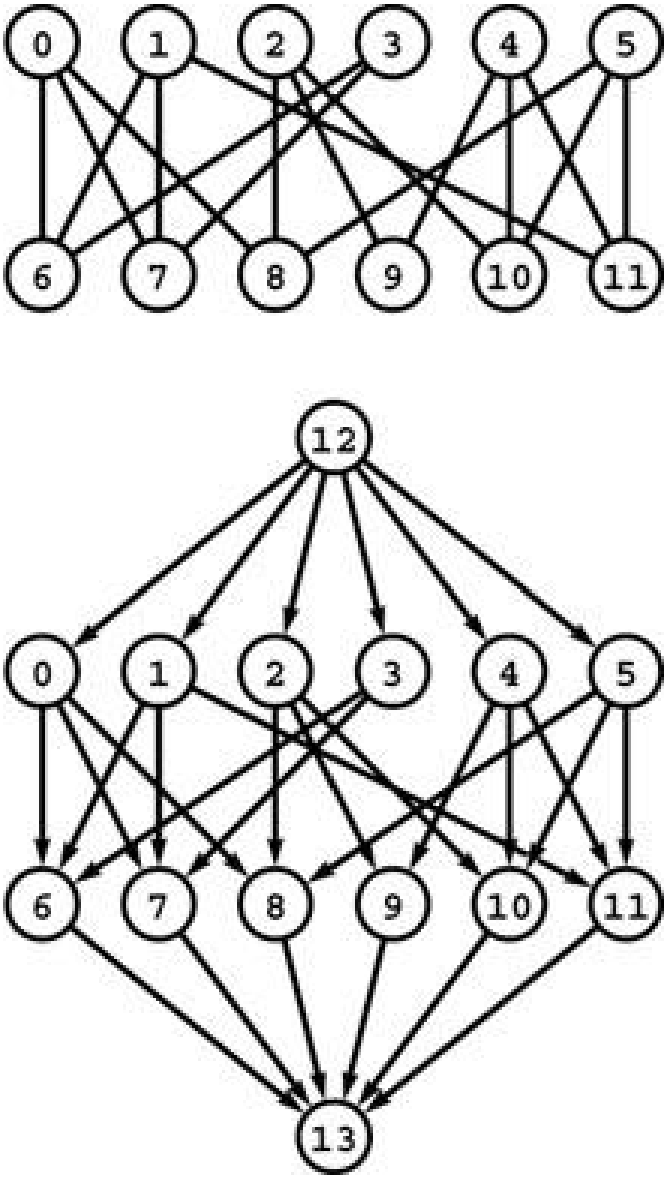
Now, any solution to the maxflow problem for this network provides a solution to the corresponding bipartite-matching problem. The matching corresponds exactly to those edges between vertices in the two sets that are filled to capacity by the maxflow algorithm. First, the network flow always gives a legal matching: since each vertex has an edge of capacity one either coming in (from the sink) or going out (to the source), at most one unit of flow can go through each vertex, implying in turn that each vertex will be in-

cluded at most once in the matching. Second, no matching can have more edges, since any such matching would lead directly to a better flow than that produced by the maxflow algorithm.■

For example, in *Figure* 22.39, an augmenting-path maxflow algorithm might use the paths `s-0-6-t`, `s-1-7-t`, `s-2-8-t`, `s-4-9-t`, `s-5-10-t`, and `s-3-6-0-7-1-11-t` to compute the matching `0-7`, `1-11`, `2-8`, `3-6`, `0-7`, and 1-11. Thus, there is a way to match all the students to jobs in *Figure* 22.3.

Program 22.6 is a client program that reads a bipartite-matching problem from standard input and uses the reduction described in this proof to solve it. What is the running time of this program for huge networks? Certainly, the running time depends on the maxflow algorithm and implementation that we use. Also, we need to take into account that the networks that we build have a special structure (unit-capacity bipartite flow networks)—not only do the running times of the various maxflow algorithms that we have considered not necessarily approach their worst-case bounds, but also we can substantially reduce the bounds. For example, the first bound that we considered, for the generic augmenting-path algorithm, provides a quick answer:

Figure 22.39 Reduction from Bipartite Matching *To find a maximum matching in a bipartite graph* (*top*), *we construct an st-network* (bottom) *by directing all the edges from the top row to the bottom row, adding a new source with edges to each vertex on the top row, adding a new sink with edges to each vertex on the bottom row, and assigning capacity 1 to all edges. In any flow, at most one outgoing edge from each vertex on the top row can be filled and at most one incoming edge to each vertex on the bottom row can be filled, so a solution to the maxflow problem on this network gives a maximum matching for the bipartite graph.*



Program 22.6 Bipartite matching via reduction to maxflow

This client reads a bipartite matching problem with $V + V$ vertices and E edges from standard input, then constructs a flow network corresponding to the bipartite matching problem, finds the maximum flow in the network, and uses the solution to print a maximum bipartite matching.

```
#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);
  }
```

Corollary *The time required to find a maximum-cardinality matching in a bipartite graph is O (V E).*

Proof: Immediate from Property 22.6.■

Table 22.3 Empirical Study for Bipartite Matching This table shows performance parameters (number of vertices expanded and number of adjacency-list nodes touched) when various augmenting-path maxflow algorithms are used to compute a maximum bipartite matching for graphs with 2000 pairs of vertices and 500 edges (top) and 4000 edges (bottom). For this problem, depth-first search is the most effective method.

	vertices	edges
500 edges, match cardinality 347		
shortest	1071	945599
maximum capacity	1067	868407
depth-first	1073	477601
random	1073	644070
4000 edges, match cardinality 971		
shortest	3483	8280585
maximum capacity	6857	6573560
depth-first	34109	1266146
random	3569	4310656

The operation of augmenting-path algorithms on unit-capacity bipartite networks is simple to describe. Each augmenting `path` fills one edge from the source and one edge into the sink. These edges are never used as back edges, so there are at most V augmenting paths. The VE upper bound holds for any algorithm that finds augmenting paths in time proportional to E.

Table 22.3 shows performance results for solving random bipartite-matching problems using various augmenting-path algorithms. It is clear from this table that actual running times for this problem are closer to the VE worst case than to the optimal (linear) time. It is possible, with judicious choice and tuning of the maxflow implementation, to speed up this method by a factor of $\sqrt{V}$ (see Exercises 22.93 and 22.94).

This problem is representative of a situation that we face more frequently as we examine new problems and more general problem-solving models, demonstrating the effectiveness of reduction as a practical problem-solving tool. If we can find a reduction to a known general model such as the maxflow problem, we generally view that as a major step toward developing a practical solution, because it at least indicates not only that the problem is tractable, but also that we have numerous efficient algorithms for solving the problem. In many situations, it is reasonable to use an existing maxflow ADT function to solve the problem, and move on to the next problem. If performance remains a critical issue, we can study the relative performance of various maxflow algorithms or implementa-

IPR2023-00124
KASPERSKY EX 1036 Page 514

tions, or we can use their behavior as the starting point to develop a better, special-purpose algorithm. The general problem-solving model provides both an upper bound that we can choose either to live with or to strive to improve, and a host of implementations that have proved effective on a variety of other problems.

Next, we discuss problems relating to connectivity in graphs. Before considering the use of maxflow algorithms to solve connectivity problems, we examine the use of the maxflow–mincut theorem to take care of a piece of unfinished business from Chapter 18: the proofs of the basic theorems relating to paths and cuts in undirected graphs. These proofs are further testimony to the fundamental importance of the maxflow–mincut theorem.

Property 22.20 (Menger's Theorem) *The minimum number of edges whose removal disconnects two vertices in a digraph is equal to the maximum number of edge-disjoint paths between the two vertices.*

Proof: Given a digraph, define a flow network with the same vertices and edges with all edge capacities defined to be 1. By Property 22.2, we can represent any *st*-flow as a set of edge-disjoint paths from *s* to *t*, with the number of such paths equal to the value of the flow. The capacity of any *st*-cut is equal to that cut's cardinality. Given these facts, the maxflow–mincut theorem implies the stated result.■

The corresponding results for undirected graphs, and for vertex connectivity for digraphs and for undirected graphs, involve reductions similar to those considered here and are left for exercises (see Exercises 22.96 through 22.98).

Now we turn to algorithmic implications of the direct connection between flows and connectivity that is established by the maxflow–mincut theorem. Property 22.5 is perhaps the most important algorithmic implication (the mincut problem reduces to the maxflow problem), but the converse is *not* known to be true (see Exercise 22.49). Intuitively, it seems as though knowing a mincut should make easier the task of finding a maxflow, but no one has been able to demonstrate how. This basic example highlights the need to proceed with care when working with reductions among problems.

Still, we can also use maxflow algorithms to handle numerous connectivity problems. For example, they help solve the first nontrivial graph-processing problems that we encountered, in Chapter 18.

Edge connectivity What is the minimum number of edges that need to be removed to separate a given graph into two pieces? Find a set of edges of minimal cardinality that does this separation.

Vertex connectivity What is the minimum number of vertices that need to be removed to separate a given graph into two pieces? Find a set of vertices of minimal cardinality that does this separation.

These problems also are relevant for digraphs, so there are a total of four problems to consider. As we did for Menger's theorem, we consider one of them in detail (edge connectivity in undirected graphs) and leave the others for exercises.

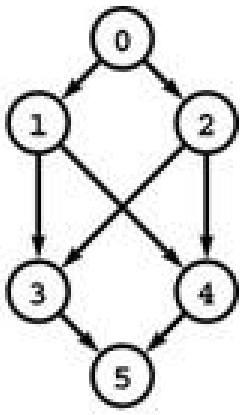
Property 22.21 *The time required to determine the edge connectivity of an undirected graph is* $O(E^2)$.

Proof: We can compute the minimum size of any cut that separates two given vertices by computing the maxflow in the *st*-network formed from the graph by assigning unit capacity to each edge. The edge connectivity is equal to the minimum of these values over all pairs of vertices.

We do not need to do the computation for all pairs of vertices, however. Let `s*` be a vertex of minimal degree in the graph. Note that the degree of `s*` can be no greater than $2E/V$. Consider any minimum cut of the graph. By definition, the number of edges in the cut set is equal to the graph's edge connectivity. The vertex `s*` appears in one of the cut's vertex sets, and the other set must have some vertex `t`, so the size of any minimal cut separating `s*` and `t` must be equal to the graph's edge connectivity. Therefore, if we solve `V-1` maxflow problems (using `s*` as the source and each other vertex as the sink), the minimum flow value found is the edge connectivity of the network.

Figure 22.40 LP Formulation of a Maxflow Problem *This linear program is equivalent to the maxflow problem for the sample network of Figure 22.5. There is one inequality for each edge, which specifies that flow cannot exceed capacity, and one equality for each vertex, which specifies that flow in must equal flow out. We use a dummy edge from sink to source to capture the flow in the network, as described in the discussion following Property 22.2.*

	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



Maximize x_{50}
subject to the constraints

$$
\begin{aligned}
x_{01} &\leq 2 \\
x_{02} &\leq 3 \\
x_{13} &\leq 3 \\
x_{14} &\leq 1 \\
x_{23} &\leq 1 \\
x_{24} &\leq 1 \\
x_{35} &\leq 2 \\
x_{45} &\leq 3 \\
x_{50} &= x_{01} + x_{02} \\
x_{01} &= x_{13} + x_{14} \\
x_{02} &= x_{23} + x_{24} \\
x_{13} + x_{23} &= x_{35} \\
x_{14} + x_{24} &= x_{45} \\
x_{35} + x_{45} &= x_{50}
\end{aligned}
$$

Now, any augmenting-path maxflow algorithm with s* as the source uses at most $2E/V$ paths; so, if we use any method that takes at most E steps to find an augmenting path, we have a total of at most $(V - 1)\ (2E/V)E$ steps to find the edge connectivity and that implies the stated result.■

This method, unlike all the other examples of this section, is not a direct reduction of one problem to another, but it does give a practical algorithm for computing edge connectivity. Again, with a careful maxflow implementation tuned to this specific problem, we can improve performance—it is possible to solve the problem in time proportional to VE (*see reference section*). The proof of Property 22.21 is an example of the more general concept

IPR2023-00124
KASPERSKY EX 1036 Page 517

of *efficient* (polynomial-time) reductions that we first encountered in section 21.7 and that plays an essential role in the theory of algorithms discussed in Part 8. Such a reduction both proves the problem to be tractable and provides an algorithm for solving it—significant first steps in coping with a new combinatorial problem.

We conclude this section by considering a strictly mathematical formulation of the maxflow problem, using linear programming (LP) (see section 21.6). This exercise is useful because it helps us to see relationships to other problems that can be so formulated.

The formulation is straightforward: We consider a system of inequalities that involve one variable corresponding to each edge, two inequalities corresponding to each edge, and one equation corresponding to each vertex. The value of the variable is the edge flow, the inequalities specify that the edge flow must be between 0 and the edge's capacity, and the equations specify that the total flow on the edges that go into each vertex must be equal to the total flow on the edges that go out of that vertex.

Figure 22.40 illustrates an example of this construction. Any maxflow problem can be cast as a LP problem in this way. LP is a versatile approach to solving combinatorial problems, and a great number of the problems that we study can be formulated as linear programs. The fact that maxflow problems are easier to solve than LP problems may be explained by the fact that the constraints in the LP formulation of maxflow problems have a specific structure not necessarily found in all LP problems.

Even though the LP problem is much more difficult in general than are specific problems such as the maxflow problem, there are powerful algorithms that can solve LP problems efficiently. The worst-case running time of these algorithms certainly exceeds the worst-case running of the specific algorithms that we have been considering, but an immense amount of practical experience over the past several decades has shown them to be effective for solving problems of the type that arise in practice.

The construction illustrated in *Figure* 22.40 indicates a proof that the maxflow problem reduces to the LP problem, unless we insist that flow values be integers. When we examine LP in detail in Part 8, we describe a way to overcome the difficulty that the LP formulation does not carry the constraint that the results have integer values.

This context gives us a precise mathematical framework that we can use to address ever more general problems and to create ever more powerful algorithms to solve those problems. The maxflow problem is easy to solve and also is versatile in its own right, as indicated by the examples in this section. Next, we examine a harder problem (still easier than LP) that encompasses a still-wider class of practical problems. We discuss ramifications of building solutions to these increasingly general problem-solving models at the

IPR2023-00124
KASPERSKY EX 1036 Page 518

end of this chapter, setting the stage for a full treatment in Part 8.

Exercises

▷ **22.71** Define a network ADT function that finds a circulation with maximal flow in a specified edge. Provide an implementation that uses GRAPHmaxflow.

22.72 Define a network ADT function that finds a maxflow in a network with no constraint on the number of sources or sinks. Provide an implementation that uses `GRAPHmaxflow`.

22.73 Define a network ADT function that finds a maxflow in an undirected network. Provide an implementation that uses GRAPHmaxflow.

22.74 Define a network ADT function that finds a maxflow in a network with capacity constraints on vertices. Provide an implementation that uses GRAPHmaxflow.

22.75 Develop an ADT for feasible-flow problems. Include an ADT function allowing clients to initialize supply–demand values and an auxiliary function to check that flow values are properly related at each vertex.

22.76 Do Exercise 22.20 for the case that each distribution point has limited capacity (that is, there is a limit on the amount of goods that can be stored there at any given time).

▷ **22.77** Show that the maxflow problem reduces to the feasible-flowproblem—that the two problems are equivalent.

22.78 Find a feasible flow for the flow network shown in *Figure* 22.11, given the additional constraints that `0, 2`, and `3` are supply vertices with weight `4`, and that `1, 4`, and `5` are supply vertices with weights `1, 3`, and `5`, respectively.

22.79 Modify *Program* 22.5 to free the memory for the list nodes corresponding to the edges that are added and removed.

○ **22.80** Write a program that takes as input a sports league's schedule and current standings and determines whether a given team is eliminated. Assume that there are no ties. *Hint*: Reduce to a feasible-flow problem with one source node that has a supply value equal to the total number of games remaining to play in the season, sink nodes that correspond to each pair of teams having a demand value equal to the number of remaining games between that pair, and distribution nodes that correspond to each team. Edges should connect the supply node to each team's distribution node (of capacity equal to the

IPR2023-00124
KASPERSKY EX 1036 Page 519

number of games that team would have to win to beat X if X were to win all its remaining games), and there should be an (uncapacitated) edge connecting each team's distribution node to each of the demand nodes involving that team.

● **22.81** Prove that the maxflow problem for networks with lower bounds on edges reduces to the standard maxflow problem.

▷ **22.82** Prove that, for networks with lower bounds on edges, the problem of finding a *minimal* flow (that respects the bounds) reduces to the maxflow problem (see Exercise 22.81).

●●● **22.83** Prove that the maxflow problem for *st*-networks reduces to the maxflow problem for undirected networks, or find a maxflow algorithm for undirected networks that has a worst-case running time substantially better than those of the algorithms in Sections 22.2 and 22.3.

▷ **22.84** Find all the matchings with five edges for the bipartite graph in *Figure* 22.38.

22.85 Extend *Program* 22.6 to use symbolic names instead of integers to refer to vertices (see *Program* 17.10).

○ **22.86** Prove that the bipartite-matching problem is equivalent to the problem of finding maxflows in networks where all edges are of unit capacity.

22.87 We might interpret the example in *Figure* 22.3 as describing student preferences for jobs and employer preferences for students, the two of which may not be mutual. Does the reduction described in the text apply to the *directed* bipartite-matching problem that results from this interpretation, where edges in the bipartite graph are directed (in either direction) from one set to the other? Prove that it does or provide a counterexample.

○ **22.88** Construct a family of bipartite-matching problems where the average length of the augmenting paths used by any augmenting-path algorithm to solve the corresponding maxflow problem is proportional to E.

22.89 Show, in the style of *Figure* 22.29, the operation of the FIFO preflow-push network-flow algorithm on the bipartite-matching network shown in *Figure* 22.39.

○ **22.90** Extend Table 22.3 to include various preflow-push algorithms.

● **22.91** Suppose that the two sets in a bipartite-matching problem are of size S and T, with $S << T$. Give as sharp a bound as you can for the worst-case running time to solve

IPR2023-00124
KASPERSKY EX 1036 Page 520

this problem, for the reduction of Property 22.19 and the maximal-augmenting-path implementation of the Ford–Fulkerson algorithm (see Property 22.8).

● **22.92** Exercise 22.91 for the FIFO-queue implementation of the preflow-push algorithm (see Property 22.13).

22.93 Extend Table 22.3 to include implementations that use the all-augmenting-paths approach described in Exercise 22.39.

●● **22.94** Prove that the running time of the method described in Exercise 22.93 is $O(\sqrt{V}E)$ for BFS.

○ **22.95** Do empirical studies to plot the expected number of edges in a maximal matching in random bipartite graphs with $V + V$ vertices and E edges, for a reasonable set of values for V and sufficient values of E to plot a smooth curve that goes from zero to V.

○ **22.96** Prove Menger's theorem (Property 22.20) for undirected graphs.

● **22.97** Prove that the minimum number of vertices whose removal disconnects two vertices in a digraph is equal to the maximum number of vertex-disjoint paths between the two vertices. *Hint*: Use a vertex-splitting transformation, similar to the one illustrated in *Figure* 22.33.

● **22.98** Extend your proof for Exercise 22.97 to apply to undirected graphs.

22.99 Implement the edge-connectivity algorithm described in this section as an ADT function for the graph ADT of Chapter 17 that returns a given graph's connectivity.

22.100 Extend your solution to Exercise 22.99 to put in a user-supplied array a minimal set of edges that separates the graph. How big an array should the user allocate?

● **22.101** Develop an algorithm for computing the edge connectivity of digraphs (the minimal number of edges whose removal leaves a digraph that is not strongly connected). Implement your algorithm as an ADT function for the digraph ADT of Chapter 19.

● **22.102** Develop algorithms based on your solutions to Exercises 22.97 and 22.98 for computing the vertex connectivity of digraphs and undirected graphs. Implement your algorithms as ADT functions for the digraph ADT of Chapter 19 and the graph ADT of Chapter 17, respectively (see Exercises 22.99 and 22.100).

22.103 Describe how to find the vertex connectivity of a digraph by solving $V \lg V$ unit-capacity maxflow problems. *Hint*: Use Menger's theorem and binary search.

IPR2023-00124
KASPERSKY EX 1036 Page 521

○ **22.104** Run empirical studies based on your solution to Exercise 22.99 to determine edge connectivity of various graphs (see Exercises 17.63–76).

▷ **22.105** Give an LP formulation for the problem of finding a maxflow in the flow network shown in *Figure* 22.11.

○ **22.106** Formulate as an LP problem the bipartite-matching problem in *Figure* 22.38.

22.5 MINCOST FLOWS

It is not unusual for there to be numerous solutions to a given maxflow problem. This fact leads to the question of whether we might wish to impose some additional criterion to choose among them. For example, there are clearly many solutions to the unit-capacity flow problems shown in *Figure* 22.23; perhaps we would prefer the one that uses the fewest edges or the one with the shortest paths, or perhaps we would like to know whether there exists one comprising disjoint paths. Such problems are more difficult than the standard maxflow problem; they fall into a more general model known as the *mincost-flow problem*, which is the subject of this section.

As there are for the maxflow problem, there are numerous equivalent ways to pose the mincost-flow problem. We consider one standard formulation in detail in this section, then consider various reductions in section 22.7.

Specifically, we use the *mincost-maxflow* model: We extend the flow networks defined in section 22.1 to allow for integer costs on the edges, use the edge costs to define a flow cost in a natural way, then ask for a maximal flow of minimal cost. As we learn, not only do we have efficient and effective algorithms for this problem, but also the problem-solving model is of broad applicability.

Definition 22.8 *The* **flow cost** *of an edge in a flow network with edge costs is the product of that edge's flow and cost. The* **cost** *of a flow is the sum of the flow costs of that flow's edges.*

We continue to assume that capacities are positive integers less than M. We also assume edge costs to be nonnegative integers less than C. (Disallowing negative costs is primarily a matter of convenience, as discussed in section 22.7.) As before, we assign names to these upper-bound values because the running times of some algorithms depend on the latter. With these basic assumptions, the problem that we wish to solve is trivial to define:

Figure 22.41 Maxflows in Flow Networks with Costs *These flows all have the same (maximal) value, but their costs (the sum of the products of edge flows and edge costs) differ. The maxflow in the center has minimal cost (no maxflow*

has lower cost).

	cap	cost	flow
0-1	3	3	1
0-2	3	1	3
1-3	2	1	1
1-4	2	1	0
2-3	1	4	1
2-4	2	2	2
3-5	2	2	2
4-5	2	1	2



cost: 21

flow
2
2
2
0
0
2
2
2



cost: 20

flow
2
2
1
1
1
1
2
2



cost: 22

Mincost maxflow Given a flow network with edge costs, find a maxflow such that no other maxflow has lower cost.

Figure 22.41 illustrates different maxflows in a flow network with costs, including a min-cost maxflow. Certainly, the computational burden of minimizing cost is no less challenging than the burden of maximizing flow with which we were concerned in Sections 22.2 and 22.3. Indeed, costs add an extra dimension that presents significant new challenges. Even so, we can shoulder this burden with a generic algorithm that is similar to the augmenting-path algorithm for the maxflow problem.

Numerous other problems reduce to or are equivalent to the mincost-maxflow problem. For example, the following formulation is of interest because it encompasses the merchandise-distribution problem that we considered at the beginning of the chapter.

Mincost feasible flow Suppose that a weight is assigned to each vertex in a flow network with edge costs, to be interpreted as supply (if positive) or demand (if negative), with the sum of the vertex weights equal to zero. Recall that we define a flow to be *feasible* if the difference between each vertex's outflow and inflow is equal to that vertex's weight. Given such a network, find a feasible flow of minimal cost.

To describe the network model for the mincost–feasible-flow problem, we use the term *distribution network* for brevity, to mean "capacitated flow network with edge costs and supply or demand weights on vertices."

In the merchandise-distribution application, supply vertices correspond to warehouses, demand vertices to retail outlets, edges to trucking routes, supply or demand values to the amount of material to be shipped or received, and edge capacities to the number and capacity of the trucks available for the various routes. A natural way to interpret each edge's cost is as the cost of moving a unit of flow through that edge (the cost of sending a unit of material in a truck along the corresponding route). Given a flow, an edge's flow cost is the part of the cost of moving the flow through the network that we can attribute to that edge.

IPR2023-00124
KASPERSKY EX 1036 Page 523

Given an amount of material that is to be shipped along a given edge, we can compute the cost of shipping it by multiplying the cost per unit by the amount. Doing this computation for each edge and adding the results together gives us the total shipping cost, which we would like to minimize.

Property 22.22 *The mincost–feasible-flow problem and the mincost-maxflow problems are equivalent*.

Proof: Immediate, by the same correspondence as Property 22.18 (see also Exercise 22.77).■

Because of this equivalence and because the mincost–feasible-flow problem directly models merchandise-distribution problems and many other applications, we use the term *mincost flow* to refer to both problems in contexts where we could refer to either. We consider other reductions in section 22.7.

To implement edge costs in flow networks, we add an integer `cost` field to the `Edge` data type from section 22.1. *Program* 22.7 is an ADT function that computes the cost of a flow. As it is when we work with maxflows, it is also prudent to implement an ADT function to check that inflow and outflow values are properly related at each vertex and that the data structures are consistent (see Exercise 22.107).

The first step in developing algorithms to solve the mincost-flow problem is to extend the definition of residual networks to include costs on the edges.

Definition 22.9 *Given a flow in a flow network with edge costs, the* **residual network** *for the flow has the same vertices as the original and one or two edges in the residual network for each edge in the original, defined as follows: For each edge* `u-v` *in the original, let* `f` *be the flow*, `c` *the capacity, and* `x` *the cost. If* `f` *is positive, include an edge* `v-u` *in the residual with capacity* `f` *and cost* `-x`; *if* `f` *is less than* `c`, *include an edge* `u-v` *in the residual with capacity* `c-f` *and cost* `x`.

Program 22.7 Computing flow cost

This ADT function returns the cost of a network's flow. It adds together cost times flow for all positive-capacity edges.

IPR2023-00124
KASPERSKY EX 1036 Page 524

```
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;
  }
```

This definition is nearly identical to Definition 22.4, but the difference is crucial. Edges in the residual network that represent backward edges have *negative* cost. Traversing those edges corresponds to removing flow in the corresponding edge in the original network, so the cost has to be reduced accordingly. Because of the negative edge costs, these networks can have negative-cost cycles. The concept of negative cycles, which seemed artificial when we first considered it in the context of shortest-paths algorithms, plays a critical role in mincost-flow algorithms, as we now see. We consider two algorithms, both based on the following optimality condition.

Property 22.23 *A maxflow is a* **mincost maxflow** *if and only if its residual network contains no negative-cost (directed) cycle.*

Proof: Suppose that we have a mincost maxflow whose residual network has a negative-cost cycle. Let x be the capacity of a minimal-capacity edge in the cycle. Augment the flow by adding x to edges in the flow corresponding to positive-cost edges in the residual network (forward edges) and subtracting x from edges corresponding to negative-cost edges in the residual network (backward edges). These changes do not affect the difference between inflow and outflow at any vertex, so the flow remains a maxflow, but they change the network's cost by x times the cost of the cycle, which is negative, thereby contradicting the assertion that the cost of the original flow was minimal.

To prove the converse, suppose that we have a maxflow F with no negative-cost cycles whose cost is not minimal, and consider any mincost maxflow M. By an argument identical to the flow decomposition theorem (Property 22.2), we can find at most E directed cycles such that adding those cycles to the flow F gives the flow M. But, since F has no negative cycles, this operation cannot lower the cost of F, a contradiction. In other words, we should be able to convert F to M by augmenting along cycles, but we cannot do so because we have no negative-cost cycles to use to lower the flow cost.■

This property leads immediately to a simple generic algorithm for solving the mincost-flow problem, called the *cycle-canceling* algorithm:

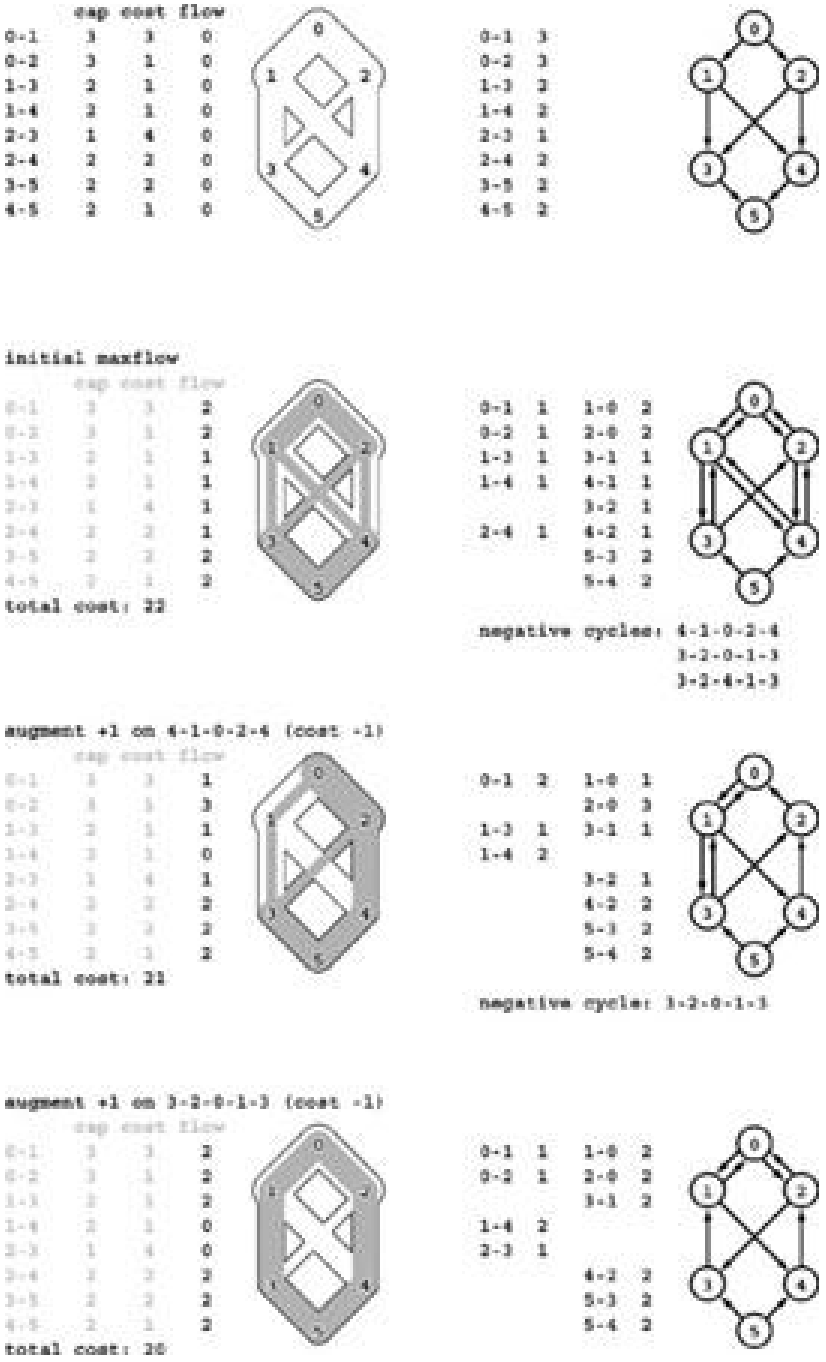
Find a maxflow. Augment the flow along any negative-cost cycle in the residual network, continuing until none remain.

This method brings together machinery that we have developed over this chapter and the previous one to provide effective algorithms for solving the wide class of problems that fit into the mincost-flow model. Like several other generic methods that we have seen, it admits several different implementations, since the methods for finding the initial maxflow and for finding the negative-cost cycles are not specified. *Figure* 22.42 shows an example mincost-maxflow computation that uses cycle canceling.

Since we have already developed algorithms for computing a maxflow and for finding negative cycles, we immediately have the implementation of the cycle-canceling algorithm given in *Program* 22.8. We use any maxflow implementation to find the initial maxflow and the Bellman–Ford algorithm to find negative cycles (*Program* 21.9). To these two implementations, we need to add only a loop to augment flow along the cycles.

We can eliminate the initial maxflow computation in the cycle-canceling algorithm, by adding a dummy edge from source to sink and assigning to it a cost that is higher than the cost of any source–sink `path` in the network (for example, $VC + 1$) and a flow that is higher than the maxflow (for example, higher than the source's outflow). With this initial setup, cycle canceling moves as much flow as possible out of the dummy edge, so the resulting flow is a maxflow. A mincost-flow computation using this technique is illustrated in *Figure* 22.43. In the figure, we use an initial flow equal to the maxflow to make plain that the algorithm is simply computing another flow of the same value but lower cost (in general, we do not know the flow value, so there is some flow left in the dummy edge at the end, which we ignore). As is evident from the figure, some augmenting cycles include the dummy edge and increase flow in the network; others do not include the dummy edge and reduce cost. Eventually, we reach a maxflow; at that point, all the augmenting cycles reduce cost without changing the value of the flow, as when we started with a maxflow.

Figure 22.42 Residual Networks (Cycle Canceling) *Each of the flows depicted here is a maxflow for the flow network depicted at the top, but only the one at the bottom is a mincost maxflow. To find it, we start with any maxflow and augment flow around negative cycles. The initial maxflow* (second from top) *has a cost of* `22`, *which is not a mincost maxflow because the residual network* (shown at right) *has three negative cycles. In this example, we augment along* `4-1-0-2-4` *to get a maxflow of cost* `21` (third from top), *which still has one negative cycle. Augmenting along that cycle gives a mincost flow (bottom). Note that augmenting along* `3-2-4-1-3` *in the first step would have brought us to the mincost flow in one step.*



Program 22.8 Cycle canceling

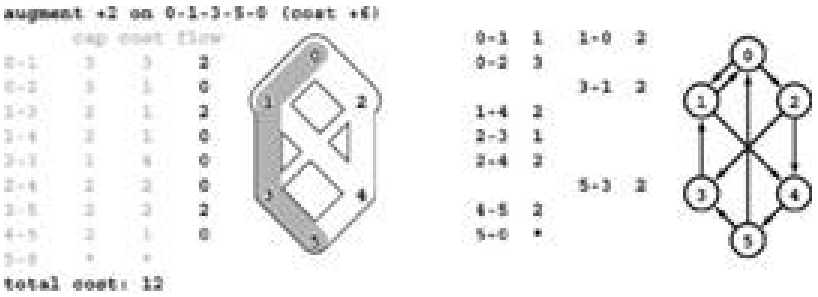
This ADT function solves the mincost-maxflow problem by canceling negative-cost cycles. It uses the ADT function GRAPHmaxflow to find a maxflow, then finds negative cycles with an ADT function `GRAPHnegcycle`, which is a version of the function described in Exercise 21.134, modified to adhere to the convention used in this

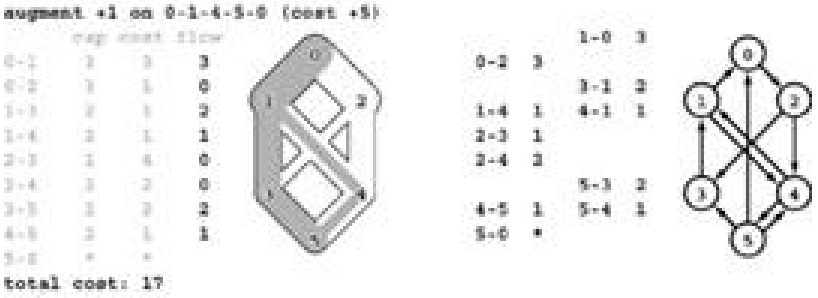
IPR2023-00124
KASPERSKY EX 1036 Page 527

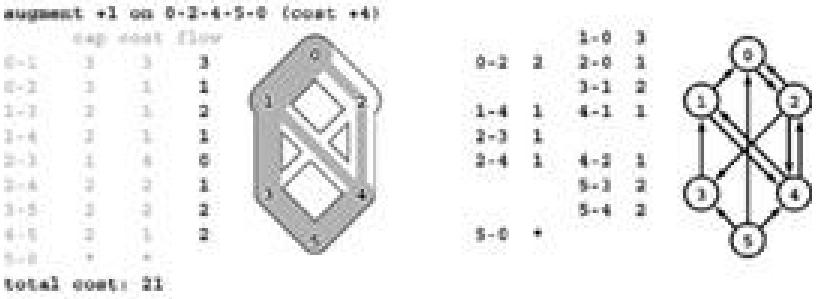
chapter where the st array contains links to nodes instead of indices. While a negative cycle exists, this code finds one, computes the maximum amount of flow to push through it, and does so. This process reduces the cost to the minimum value.

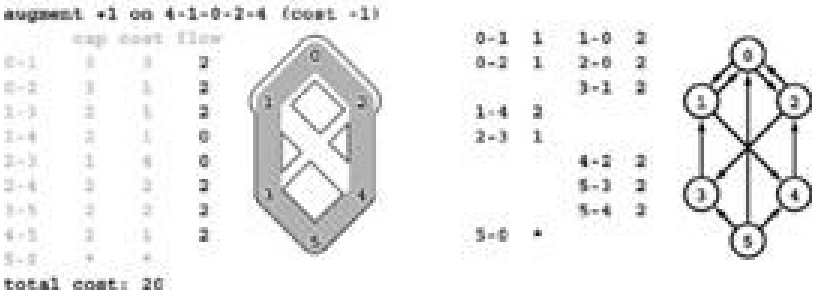
```
void addflow (link u, int d)
  { u->flow += d; u->dup->flow -=d; }
int GRAPHmincost (Graph G, int s, int t)
  { int d, x, w; link u, st[maxV];
    GRAPHmaxflow (G, s, t);
    while ( (x = GRAPHnegcycle (G, st)) != -1)
      {
       u = st[x]; d = Q;
       for (w = u->dup->v; w != x; w = u->dup->v)
         { 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);
  }
```

Figure 22.43 Cycle Canceling Without Initial Maxflow *This sequence illustrates the computation of a mincost maxflow from an initially empty flow with the cycle-canceling algorithm, by using a dummy edge from sink to source in the residual network with infinite capacity and infinite negative cost. The dummy edge makes any augmenting* `path` *from* `0` *to* `5` *a negative cycle (but we ignore it when augmenting and computing the cost of the flow). Augmenting along such a* `path` *increases the flow, as in augmenting-path algorithms (top three rows). When there are no cycles involving the dummy edge, there are no paths from source to sink in the residual network, so we have a maxflow (third from top). At that point, augmenting along a negative cycle decreases the cost without changing the flow value (bottom). In this example, we compute a maxflow, then decrease its cost; but that need not be the case. For example, the algorithm might have augmented along the negative cycle* `1-4-5-3-1` *instead of* `0-1-4-5-0` *in the second step. Since every augmentation either increases the flow or reduces the cost, we always wind up with a mincost maxflow.*









Technically, using a dummy-flow initialization is neither more nor less generic than using a maxflow initialization for cycle canceling. The former does encompass all augmenting-path maxflow algorithms, but not all maxflows can be computed with an augmenting-path algorithm (see Exercise 22.42). On the one hand, by using this technique, we may be giving up the benefits of a sophisticated maxflow algorithm; on the other hand, we may be better off reducing costs during the process of building a maxflow. In practice, dummy-flow initialization is widely used because it is so simple to implement.

As for maxflows, the existence of this generic algorithm guarantees that every min-cost-flow problem (with capacities and costs that are integers) has a solution where flows

are all integers; and the algorithm computes such a solution (see Exercise 22.110). Given this fact, it is easy to establish an upper bound on the amount of time that any cycle-canceling algorithm will require.

Property 22.24 *The number of augmenting cycles needed in the generic cycle-canceling algorithm is less than ECM*.

Proof: In the worst case, each edge in the initial maxflow has capacity M, cost C, and is filled. Each cycle reduces this cost by at least 1.■

Corollary *The time required to solve the mincost-flow problem in a sparse network is* $O(V^3CM)$.

Proof: Immediate by multiplying the worst-case number of augmenting cycles by the worst-case cost of the Bellman–Ford algorithm for finding them (see Property 21.22).■

Like that of augmenting-path methods, this running time is extremely pessimistic, as it assumes not only that we have a worst-case situation where we need to use a huge number of cycles to minimize cost, but also that we have another worst-case situation where we have to examine a huge number of edges to find each cycle. In many practical situations, we use relatively few cycles that are relatively easy to find, and the cycle-canceling algorithm is effective.

It is possible to develop a strategy that finds negative-cost cycles and ensures that the number of negative-cost cycles used is less than VE (*see reference section*). This result is significant because it establishes the fact that the mincost-flow problem is tractable (as are all the problems that reduce to it). However, practitioners typically use implementations that admit a bad worst case (in theory) but use substantially fewer iterations on the problems that arise in practice than predicted by the worst-case bounds.

The mincost-flow problem represents the most general problem-solving model that we have yet examined, so it is perhaps surprising that we can solve it with such a simple implementation. Because of the importance of the model, numerous other implementations of the cycle-canceling method and numerous other different methods have been developed and studied in detail. *Program* 22.8 is a remarkably simple and effective starting point, but it suffers from two defects that can potentially lead to poor performance. First, each time that we seek a negative cycle, we start from scratch. Can we save intermediate information during the search for one negative cycle that can help us find the next? Second, *Program* 22.8 just takes the first negative cycle that the Bellman–Ford algorithm finds. Can we direct the search towards negative cycles with particular properties? In section 22.6, we consider an improved implementation, still generic, that represents a re-

sponse to both of these questions.

Exercises

22.107 Modify your solution to Exercise 22.13 to check that the source's outflow equals the sink's inflow and that outflow equals inflow at every internal vertex. Also check that the cost is the same sign as flow and capacity for all edges and that the cost of `u-v` and the cost of `v-u` sum to zero for all `u` and `v`.

22.108 Expand your ADT for feasible flows from Exercise 22.75 to include costs. Include an ADT function for solving the mincost–feasible-flow problem that uses the standard flow network ADT and calls `GRAPHmincost`.

▷ **22.109** Given a flow network whose edges are not all maximal capacity and cost, give an upper bound better than *ECM* on the cost of a maxflow.

22.110 Prove that, if all capacities and costs are integers, then the mincost-flow problem has a solution where all flow values are integers.

▷ **22.111** Modify *Program* 22.8 to initialize with flow in a dummy edge instead of computing a flow.

○ **22.112** Give all possible sequences of augmenting cycles that might have been depicted in *Figure* 22.42.

○ **22.113** Give all possible sequences of augmenting cycles that might have been depicted in *Figure* 22.43.

22.114 Show, in the style of *Figure* 22.42, the flow and residual networks after each augmentation when you use the cycle-canceling implementation *of Program* 22.8 to find a mincost flow in the flow network shown in *Figure* 22.11, with cost `2` assigned to `0-2` and `0-3`; cost `3` assigned to `2-5` and `3-5`; cost `4` assigned to `1-4`; and cost `1` assigned to all of the other edges. Assume that the maxflow is computed with the shortest-augmenting-path algorithm.

22.115 Answer Exercise 22.114, but assume that the program is modified to start with a maxflow in a dummy edge from source to sink, as in *Figure* 22.43.

22.116 Extend your solutions to Exercises 22.7 and 22.8 to handle costs in flow networks.

22.117 Extend your solutions to Exercises 22.10 through 22.12 to include costs in the net-

works. Take each edge's cost to be roughly proportional to the Euclidean distance between the vertices that the edge connects.

22.6 NETWORK SIMPLEX ALGORITHM

The running time of the cycle-canceling algorithm is based on not just the number of negative-cost cycles that the algorithm uses to reduce the flow cost but also the time that the algorithm uses to find each of the cycles. In this section, we consider a basic approach that both dramatically decreases the cost of identifying negative cycles and admits methods for reducing the number of iterations. This implementation of the cycle-canceling algorithm is known as the *network simplex* algorithm. It is based on maintaining a tree data structure and reweighting costs such that negative cycles can be identified quickly.

To describe the network simplex algorithm, we begin by noting that, with respect to any flow, each network edge `u-v` is in one of three states (see *Figure* 22.44):

- *Empty*, so flow can be pushed from only `u` to `v`
- *Full*, so flow can be pushed from only `v` to `u`
- *Partial* (neither empty nor full), so flow can pushed either way

This classification is familiar from our use of residual networks throughout this chapter. If `u-v` is an empty edge, then `u-v` is in the residual network, but `v-u` is not; if `u-v` is a full edge, then `v-u` is in the residual network, but `u-v` is not; if `u-v` is a partial edge, then both `u-v` and `v-u` are in the residual network.

Definition 22.10 *Given a maxflow with no cycle of partial edges, a* **feasible spanning tree** *for the maxflow is any spanning tree of the network that contains all the flow's partial edges.*

> **Figure 22.44 Edge Classification** *With respect to any flow, every edge is either empty, full, or partial (neither empty nor full). In this flow, edge* `1-4` *is empty; edges* `0-2, 2-3, 2-4, 3-5`*, and* `4-5` *are full; and edges* `0-1` *and* `1-3` *are partial. Our conventions in figures give two ways to identify an edge's state: In the flow column,* `0` *entries are empty edges; starred entries are full edges; and entries that are neither* `0` *nor starred are partial edges. In the residual network (bottom), empty edges appear only in the left column; full edges appear only in the right column; and partial edges appear in both columns.*

IPR2023-00124
KASPERSKY EX 1036 Page 532

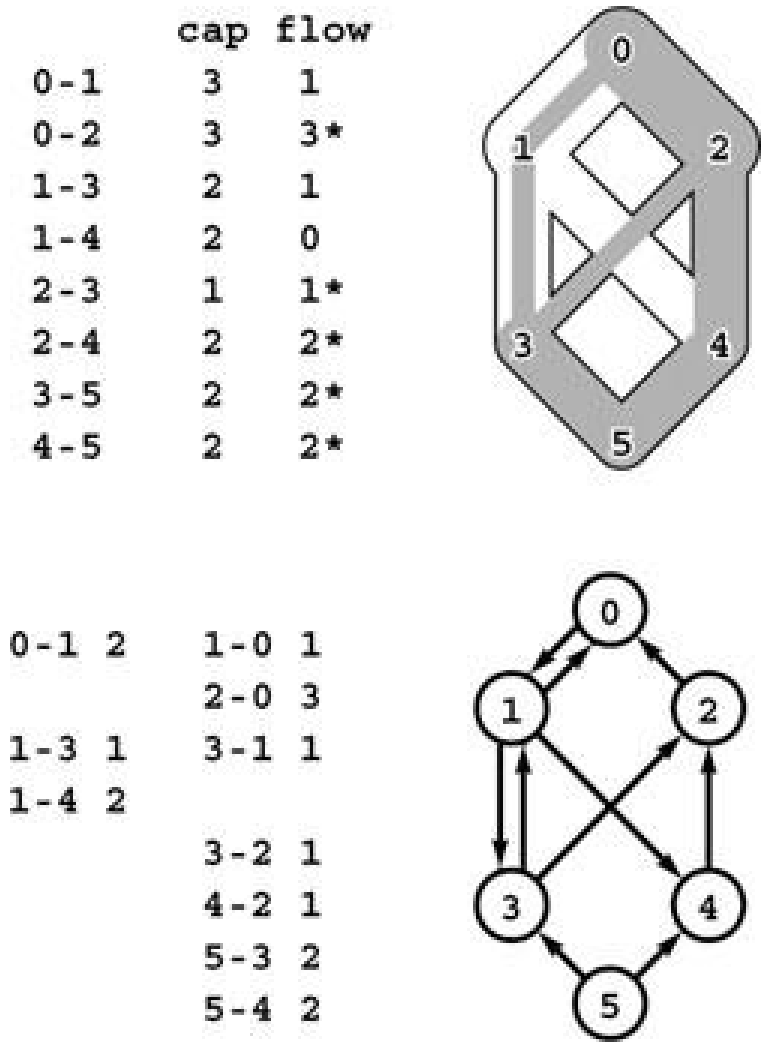
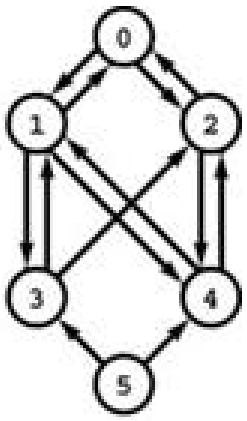
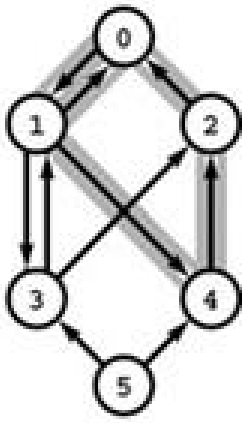


Figure 22.45 Maxflow Spanning Tree *Given any maxflow (top), we can construct a maxflow that has a spanning tree such that no nontree edges are partial by the two-step process illustrated in this example. First, we break cycles of partial edges: in this case, we break the cycle* `0-2-4-1-0` *by pushing 1 unit of flow along it. We can always fill or empty at least one edge in this way; in this case, we empty* `1-4` and fill both `0-2` *and* `2-4` (*center). Second, we add empty or full edges to the set of partial edges to make a spanning tree: in this case, we add* `0-2, 1-4` *and* `3-5` (bottom).

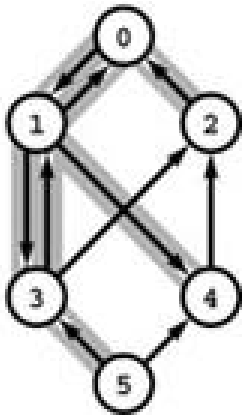
	cap	flow
0-1	3	2
0-2	3	2
1-3	2	1
1-4	2	1
2-3	1	1*
2-4	2	1
3-5	2	2*
4-5	2	2*



	cap	flow
0-1	3	1
0-2	3	3*
1-3	2	1
1-4	2	0
2-3	1	1*
2-4	2	2*
3-5	2	2*
4-5	2	2*



	cap	flow
0-1	3	1
0-2	3	3*
1-3	2	1
1-4	2	0
2-3	1	1*
2-4	2	2*
3-5	2	2*
4-5	2	2*



In this context, we ignore edge directions in the spanning tree. That is, any set of V - 1 directed edges that connects the network's V vertices (ignoring edge directions) constitutes a spanning tree, and a spanning tree is feasible if all nontree edges are either full or empty.

The first step in the network simplex algorithm is to build a spanning tree. One way to build it is to compute a maxflow, to break cycles of partial edges by augmenting along each cycle to fill or empty one of its edges, then to add full or empty edges to the remaining partial edges to make a spanning tree. An example of this process is illustrated in *Figure* 22.45. Another option is to start with the maxflow in a dummy edge from source to

sink. Then, this edge is the only possible partial edge, and we can build a spanning tree for the flow with any graph search. An example of such a spanning tree is illustrated in *Figure* 22.46.

Now, adding any nontree edge to a spanning tree creates a cycle. The basic mechanism behind the network simplex algorithm is a set of vertex weights that allows immediate identification of the edges that, when added to the tree, create negative-cost cycles in the residual network. We refer to these vertex weights as *potentials* and use the notation φ (*v*) to refer to the potential associated with `v`. Depending on context, we refer to potentials as a function defined on the vertices, or as a set of integer weights with the implicit assumption that one is assigned to each vertex, or as a vertex-indexed array (since we always store them that way in implementations).

Definition 22.11 *Given a flow in a flow network with edge costs, let c (u, v) denote the cost of* `u-v` *in the flow's residual network. For any potential function φ, the* **reduced cost** *of an edge* `u-v` *in the residual network with respect to φ, which we denote by c* (u, v), is defined to be the value c (u, v) – (φ (u) – φ (v))*.

In other words, the reduced cost of every edge is the difference between that edge's actual cost and the difference of the potentials of the edge's vertices. In the merchandise distribution application, we can see the intuition behind node potentials: If we interpret the potential *φ* (*u*) as the cost of buying a unit of material at node *u*, the full cost *c** (*u, v*) + *φ* (*u*) – *φ* (*v*) is the cost of buying at *u*, shipping to *v* and selling at *v*.

We use code like the following to compute reduced costs:

```
#define costR (e) = e.cost - (phi[e.u] - phi[e.v])
```

That is, while we maintain a vertex-indexed array `phi` for the vertex potentials, there is no need to store the reduced edge costs anywhere, because it is so easy to compute them.

In the network simplex algorithm, we use feasible spanning trees to define vertex potentials such that reduced edge costs with respect to those potentials give direct information about negative-cost cycles. Specifically, we maintain a feasible spanning tree throughout the execution of the algorithm and we set the values of the vertex potentials such that all tree edges have reduced cost zero.

Property 22.25 *We say that a set of vertex potentials is* **valid** *with respect to a spanning tree if all tree edges have zero reduced cost. All valid vertex potentials for any given spanning tree imply the same reduced costs for each network edge.*

IPR2023-00124
KASPERSKY EX 1036 Page 535

Proof: Given two different potential functions φ and φ' that are both valid with respect to a given spanning tree, we show that they differ by an additive constant: that $\varphi(u) = \varphi'(u) + \Delta$ for all u and some constant Δ. Then, $\varphi(u) - \varphi(v) = \varphi'(u) - \varphi'(v)$ for all u and v, implying that all reduced costs are the same for the two potential functions.

For any two vertices u and v that are connected by a tree edge, we must have $\varphi(v) = \varphi(u) - c(u, v)$, by the following argument. If `u-v` is a tree edge, then $\varphi(v)$ must be equal to $\varphi(u) - c(u, v)$, to make the reduced cost $c(u, v) - \varphi(u) + \varphi(v)$ equal to zero; if `v-u` is a tree edge, then $\varphi(v)$ must be equal to $\varphi(u) + c(v, u) = \varphi(u) - c(u, v)$, to make the reduced cost $c(v, u) - \varphi(v) + \varphi(u)$ equal to zero. The same argument holds for φ', so we must also have $\varphi'(v) = \varphi'(u) - c(u, v)$.

Subtracting, we find that $\varphi(v) - \varphi'(v) = \varphi(u) - \varphi'(u)$ for any u and v connected by a tree edge. Denoting this difference by Δ for any vertex and applying the equality along the edges of any search tree of the spanning tree, immediately gives the desired result that $\varphi(u) = \varphi'(u) + \Delta$ for all u.■

Another way to imagine the process of defining a set of valid vertex potentials is that we start by fixing one value, then compute the values for all vertices connected to that vertex by tree edges, then compute them for all vertices connected to those vertices, and so forth. No matter where we start the process, the potential difference between any two vertices is the same, determined by the structure of the tree. *Figure* 22.47 depicts an example. We consider the details of the task of computing potentials after we examine the relationship between reduced costs on nontree edges and negative-cost cycles.

Figure 22.46 Spanning Tree for Dummy Maxflow

If we start with flow on a dummy edge from source to sink, then that is the only possible partial edge, so we can use any spanning tree of the remaining nodes to build a spanning tree for the flow. In the example, the edges `0-5, 0-1, 0-2, 1-3`, *and* `1-4` *comprise a spanning tree for the initial maxflow. All of the nontree edges are empty.*

	cap	cost	flow
0-1	3	3	0
0-2	3	1	0
1-3	2	1	0
1-4	2	1	0
2-3	1	4	0
2-4	2	2	0
3-5	2	2	0
4-5	2	1	0
0-5	6	9	4

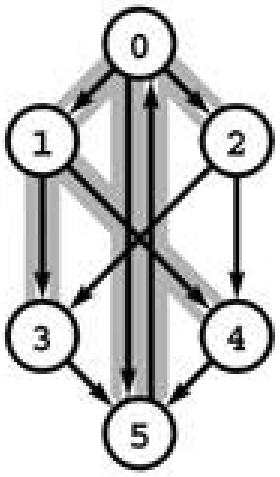
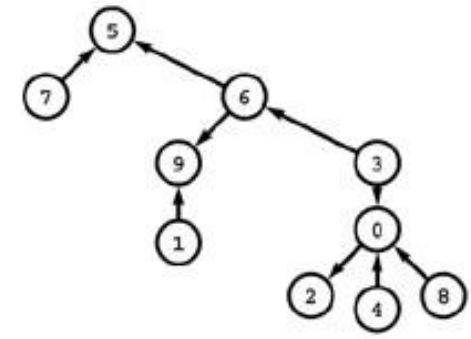
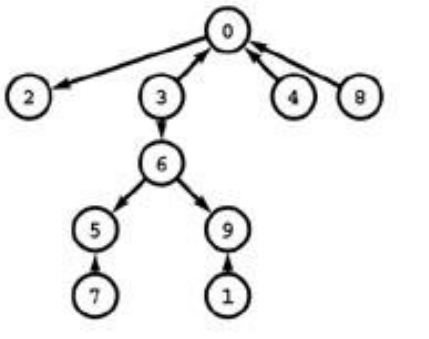


Figure 22.47 Vertex Potentials *Vertex potentials are determined by the structure of the spanning tree and by an initial assignment of a potential value to any vertex. At left is a set of edges that comprise a spanning tree of the ten vertices* `0` *through* `9`*. In the center is a representation of that tree with* `5` *at the root, vertices connected to* `5` *one level lower, and so forth. When we assign the root the potential value* `0`*, there is a unique assignment of potentials to the other nodes that make the difference between the potentials of each edge's vertices equal to its cost. At right is a different representation of the same tree with* `0` *at the root. The potentials that we get by assigning* `0` *the value* `0` *differ from those in the center by a constant offset. All our computations use the difference between two potentials: this difference is the same for any pair of potentials no matter what vertex we start with (and no matter what value we assign it), so our choice of starting vertex and value is immaterial.*

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



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



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

Property 22.26 *We say that a nontree edge is* **eligible** *if the cycle that it creates with tree edges is a negative-cost cycle in the residual network. An edge is eligible if and only if it is a full edge with positive reduced cost or an empty edge with negative reduced cost.*

Proof: Suppose that the edge `u-v` creates the cycle `t1-t2-t3-...-td-t1` with tree edges

`t1-t2, t2-t3`, ..., where `v` is `t1` and `u` is `td`. The reduced cost definitions of each edge imply the following:

$$
\begin{aligned}
c(u, v) &= c^*(u, v) + \phi(u) - \phi(t_1) \\
c(t_1, t_2) &= \phi(t_1) - \phi(t_2) \\
c(t_2, t_3) &= \phi(t_2) - \phi(t_3) \\
&\vdots \\
c(t_{d-1}, u) &= \phi(t_{d-1}) - \phi(u)
\end{aligned}
$$

The left-hand side of the sum of these equations gives the total cost of the cycle and the right-hand side collapses to $c^*(u, v)$. In other words, the edge's reduced cost gives the cycle cost, so only the edges described can give a negative-cost cycle■

Property 22.27 *If we have a flow and a feasible spanning tree with no eligible edges, the flow is a mincost flow.*■

Proof: If there are no eligible edges, then there are no negative-cost cycles in the residual network, so the optimality condition of Property 22.23 implies that the flow is a mincost flow.

An equivalent statement is that if we have a flow and a set of vertex potentials such that reduced costs of tree edges are all zero, full nontree edges are all nonnegative, and empty nontree edges are all nonpositive, then the flow is a mincost flow.

If we have eligible edges, we can choose one and augment along the cycle that it creates with tree edges to get a lower-cost flow. As we did with the cycle-canceling implementation in section 22.5, we go through the cycle to find the maximum amount of flow that we can push, then go through the cycle again to push that amount, which fills or empties at least one edge. If that is the eligible edge that we used to create the cycle, it becomes ineligible (its reduced cost stays the same, but it switches from full to empty or empty to full). Otherwise, it becomes partial. By adding it to the tree and removing a full edge or an empty edge from the cycle, we maintain the invariant that no nontree edges are partial and that the tree is a feasible spanning tree. Again, we consider the mechanics of this computation later in this section.

In summary, feasible spanning trees give us vertex potentials, which give us reduced

costs, which give us eligible edges, which give us negative-cost cycles. Augmenting along a negative-cost cycle reduces the flow cost and also implies changes in the tree structure. Changes in the tree structure imply changes in the vertex potentials; changes in the vertex potentials imply changes in reduced edge costs; and changes in reduced costs imply changes in the set of eligible edges. After making all these changes, we can pick another eligible edge and start the process again. This generic implementation of the cycle-canceling algorithm for solving the mincost-flow problem is called the *network simplex* algorithm:

Build a feasible spanning tree and maintain vertex potentials such that all tree vertices have zero reduced cost. Add an eligible edge to the tree, augment the flow along the cycle that it makes with tree edges, and remove from the tree an edge that is filled or emptied, continuing until no eligible edges remain.

This implementation is a generic one because the initial choice of spanning tree, the method of maintaining vertex potentials, and the method of choosing eligible edges are not specified. The strategy for choosing eligible edges determines the number of iterations, which trades off against the differing costs of implementing various strategies and recalculating the vertex potentials.

Property 22.28 *If the generic network simplex algorithm terminates, it computes a mincost flow.*

Proof: If the algorithm terminates, it does so because there are no negative-cost cycles in the residual network, so by Property 22.23 the maxflow is of mincost.■

The condition that the algorithm might not terminate derives from the possibility that augmenting along a cycle might fill or empty multiple edges, thus leaving edges in the tree through which no flow can be pushed. If we cannot push flow, we cannot reduce the cost, and we could get caught in an infinite loop adding and removing edges to make a fixed sequence of spanning trees. Several ways to avoid this problem have been devised; we discuss them later in this section after we look in more detail at implementations.

The first choice that we face in developing an implementation of the network simplex algorithm is what representation to use for the spanning tree. We have three primary computational tasks that involve the tree:

- Computing the vertex potentials
- Augmenting along the cycle (and identifying an empty or a full edge on it)
- Inserting a new edge and removing an edge on the cycle formed

Each of these tasks is an interesting exercise in data structure and algorithm design. There are several data structures and numerous algorithms that we might consider, with varied performance characteristics. We begin by considering perhaps the simplest available data structure—which we first encountered in Chapter 1 (!)—the parent-link tree representation. After we examine algorithms and implementations that are based on the parent-link representation for the tasks just listed and describe their use in the context of the network simplex algorithm, we discuss alternative data structures and algorithms.

Program 22.9 Vertex potential calculation

The recursive function phiR follows parent links up the tree until finding one whose potential is valid (by convention the potential of the root is always `-C` and valid), then computes potentials for vertices on the `path` on the way back down as the last actions of the recursive invocations. It marks each vertex whose potential it computes by setting its mark entry to the current value of valid.

The `ST` macro allows our tree-processing functions to maintain the simplicity of the parent-tree link abstraction while at the same time following links through the network's adjacency-lists data structure.

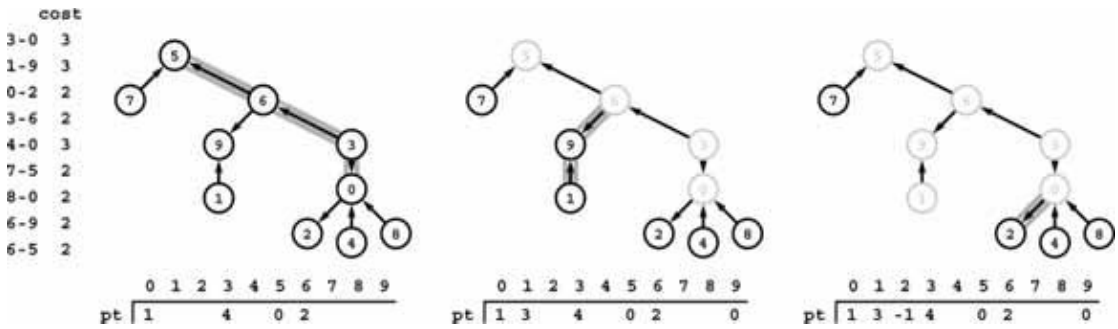
```
#define ST (i) st[i]->dup->v
static int valid, phi[maxV];
int phiR (link st[], int v)
  {
    if (ST (v) == v)
      { mark[v] = valid; return -C; }
    if (mark[v] != valid)
      phi[v] =phiR (st, ST (v)) - st[v]->cost;
    mark[v] = valid;
    return phi[v];
  }
```

As we did in several other implementations in this chapter, beginning with the augmenting-path maxflow implementation, we keep links into the network representation, rather

than simple indices in the tree representation, to allow us to have access to the flow values that need to be changed without losing access to the vertex name.

Program 22.9 is an implementation that assigns vertex potentials in time proportional to *V*. It is based on the following idea, also illustrated in *Figure* 22.48. We start with any vertex and recursively compute the potentials of its ancestors, following parent links up to the root, to which, by convention, we assign potential `0`. Then, we pick another vertex and use parent links to compute recursively the potentials of its ancestors. The recursion terminates when we reach an ancestor whose potential is known; then, on the way out of the recursion, we travel back down the path, computing each node's potential from its parent. We continue this process until we have computed all potential values. Once we have traveled along a path, we do not revisit any of its edges, so this process runs in time proportional to *V*.

Figure 22.48 Computing Potentials Through Parent Links *We start at* `0`, *follow the* `path` *to the root, set* `pt[5]` *to* `0`, *then work down the path, first setting* `6` *to make* `pt[6] - pt[5]` *equal to the cost of* `6-5`, *then setting* `p[3]` *to make* `p[3] - p[6]` *equal to the cost of* `3-6`, *and so forth* (*left*). *Then we start at* `1` *and follow parent links until hitting a vertex whose potential is known* (`6` *in this case) and work down the* `path` *to compute potentials on* `9` *and* `1` (*center*). *When we start at* `2`, *we can compute its potential from its parent* (right); *when we start at* `3`, *we see that its potential is already known, and so forth. In this example, when we try each vertex after* `1`, *we either find that its potential is already done or we can compute the value from its parent. We never retrace an edge, no matter what the tree structure, so the total running time is linear*.



Given two nodes in a tree, their *least common ancestor (LCA)* is the root of the smallest subtree that contains them both. The cycle that we form by adding an edge connecting two nodes consists of that edge plus the edges on the two paths from the two nodes to their LCA. To augment along a cycle, we do not need to consider edges in order around the cycle; it suffices to consider them all (in either direction). Accordingly, we can augment along the cycle by simply following paths from each of the two nodes to their LCA. To augment along the cycle formed by the addition of `u-v`, we find the LCA of `u` and `v` (say, `t`) and push flow from `u` to `v`, from `v` along the `path` to `t`, and from `u` along the `path`

to t, but in reverse direction for each edge. To compute the amount of flow to push, we first traverse the edges on the cycle in the same manner to determine the maximum amount that can be pushed. *Program* 22.10 is an implementation of this idea, in the form of a function that augments a cycle and also returns an edge that is emptied or filled by the augmentation.

The implementation in *Program* 22.10 uses a simple technique to avoid paying the cost of initializing all the marks each time that we call it. We maintain the marks as global variables, initialized to zero. Each time that we seek an LCA, we increment a global counter and mark vertices by setting their corresponding entry in a vertex-indexed array to that counter. After initialization, this technique allows us to perform the computation in time proportional to the length of the cycle. In typical problems, we might augment along a large number of small cycles, so the time saved can be substantial. As we learn, the same technique is useful in saving time in other parts of the implementation.

Program 22.10 Augmenting along a cycle

The function lca finds the least common ancestor of its argument vertices by moving up the tree simultaneously from the two vertices, marking each node encountered, until hitting a marked node that is not the root. (If the root is the LCA, the loop terminates with u equal to v.)

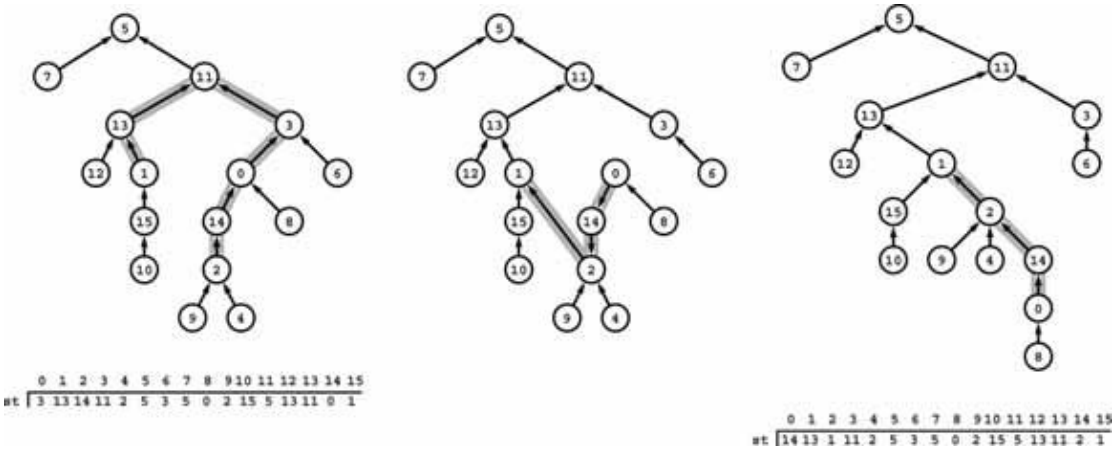
All the nodes on the path from the two nodes to their LCA are on the cycle formed by adding an edge connecting the two nodes. To augment along the cycle, we traverse the paths once to find the maximum amount of flow that we can push through their edges. Then we traverse the two paths again, pushing flow in opposite directions on one of them.

The key performance characteristic of these functions is that they run in time proportional to the number of nodes on the cycle.

IPR2023-00124
KASPERSKY EX 1036 Page 542

```
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; }
   }
```

Figure 22.49 Spanning Tree Substitution *This example illustrates the basic tree-manipulation operation in the network simplex algorithm for the parent-link representation. At left is a sample tree with links all pointing upwards, as indicated by the parent-link structure in the array* `st`*. Our code actually maintains the parent-link values implicitly through links into the network structure, so tree links can represent network edges in either orientation (see text). Adding the edge* `1-2` *creates a cycle with the paths from* `1` and `2` to their LCA, `11`*. If we then delete one of those edges, say* `0-3`*, the structure remains a tree. To update the parent-link array to reflect the change, we switch the directions of all the links from* `2` *up to* `3` *(center). The tree at right is the same tree with node positions changed so that links all point up, as indicated by the parent-link array that represents the tree* (bottom right).



Our third tree-manipulation task is to substitute an edge `u-v` for another edge in the cycle that it creates with tree edges. *Program* 22.11 is an implementation of a function that accomplishes this task for the parent-link representation. Again, the LCA of `u` and `v` is important, because the edge to be removed is either on the `path` from `u` to the LCA or on the `path` from `v` to the LCA. Removing an edge detaches all its descendents from the tree, but we can repair the damage by reversing the links between `u-v` and the removed edge, as illustrated in *Figure* 22.49.

These three implementations support the basic operations underlying the network simplex algorithm: we can choose an eligible edge by examining reduced costs and flows; we can use the parent-link representation of the spanning tree to augment along the negative cycle formed with tree edges and the chosen eligible edge; and we can update the tree and recalculate potentials. These operations are illustrated for an example flow network in Figures 22.50 and 22.51.

Figure 22.50 illustrates initialization of the data structures using a dummy edge with the-maxflow on it, as in *Figure* 22.43. Shown there are an initial feasible spanning tree with its parent-link representation, the corresponding vertex potentials, the reduced costs for the nontree edges, and the initial set of eligible edges. Also, rather than computing the maxflow value in the implementation, we use the outflow from the source, which is guaranteed to be no less than the maxflow value; we use the maxflow value here to make the operation of the algorithm easier to trace.

***Program* 22.11 Spanning tree substitution**

The function update adds an edge to the spanning tree and removes an edge on the cycle thus created. The edge to be removed is on the `path` from one of the two vertices on the edge added to their LCA. This implementation uses the function onpath to find the edge removed and the function reverse to reverse the edges on the `path` between it and the edge added.

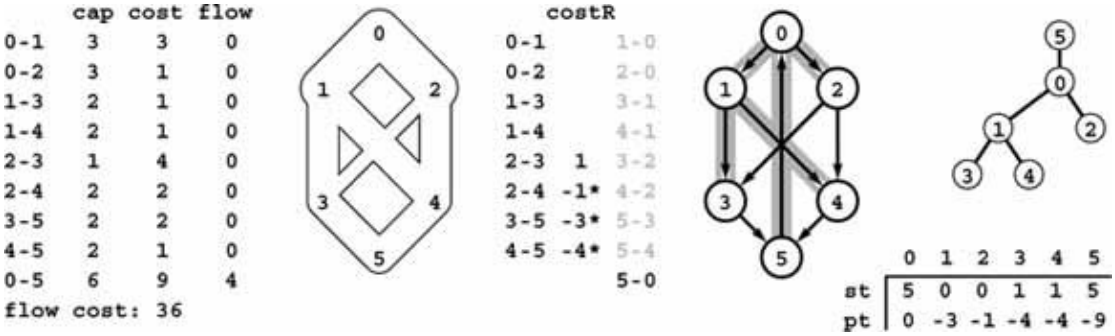
```
int onpath (link st [], int a, int b, int c)
  { int i;
    for (i = a; i != c; i = ST (i))
      if (i == b) return 1;
    return 0;
  }
int reverse (link st[], int u, int x)
  { int i;
    while (i != st[x]->v)
      { i = st[u]->v; st[i] = st[u]->dup; u = i; }
  }
int update (link st[], link w, 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; }
  }
```

Figure 22.51 illustrates the changes in the data structures for each of a sequence of eligible edges and augmentations around negative-cost cycles. The sequence does not reflect any particular method for choosing eligible edges; it represents the choices that make the augmenting paths the same as depicted in *Figure* 22.43. These figures show all vertex potentials and all reduced costs after each cycle augmentation, even though many of these numbers are implicitly defined and are not necessarily computed explicitly by typical implementations. The purpose of these two figures is to illustrate the overall progress of the algorithm and the state of the data structures as the algorithm moves from one feasible spanning tree to another simply by adding an eligible edge and removing a tree edge on the cycle that is formed.

Figure 22.50 Network Simplex Initialization *To initialize the data structures for the network simplex algorithm, we start with zero flow on all edges* (left)*, then add a dummy edge* `0-5` *from source to sink with flow no less than the maxflow*

IPR2023-00124
KASPERSKY EX 1036 Page 545

value (for clarity, we use a value equal to the maxflow value here). The cost value `9` *for the dummy edge is greater than the cost of any cycle in the network; in the implementation, we use the value CV. The dummy edge is not shown in the flow network, but it is included in the residual network* (center). We initialize the spanning tree with the sink at the root, the source as its only child, and a search tree of the graph induced by the remaining nodes in the residual network. The implementation uses the parent-link representation of the tree in the array st; our figures depict this representation and two others: the rooted representation shown on the right and the set of shaded edges in the residual network. The vertex potentials are in the `pt` *array and are computed from the tree structure so as to make the difference of a tree edge's vertex potentials equal to its cost. The column labeled* `costR` *in the center gives the reduced costs for nontree edges, which are computed for each edge by adding the difference of its vertex potentials to its cost. Reduced costs for tree edges are zero and left blank. Empty edges with negative reduced cost and full edges with positive reduced cost (eligible edges) are marked with asterisks.*



One critical fact that is illustrated in the example in *Figure* 22.51 is that the algorithm might not even terminate, because full or empty edges on the spanning tree can stop us from pushing flow along the negative cycle that we identify. That is, we can identify an eligible edge and the negative cycle that it makes with spanning tree edges, but the maximum amount of flow that we can push along the cycle may be `0`. In this case, we still substitute the eligible edge for an edge on the cycle, but we make no progress in reducing the cost of the flow. To ensure that the algorithm terminates we need to prove that we cannot end up in an endless sequence of zero-flow augmentations.

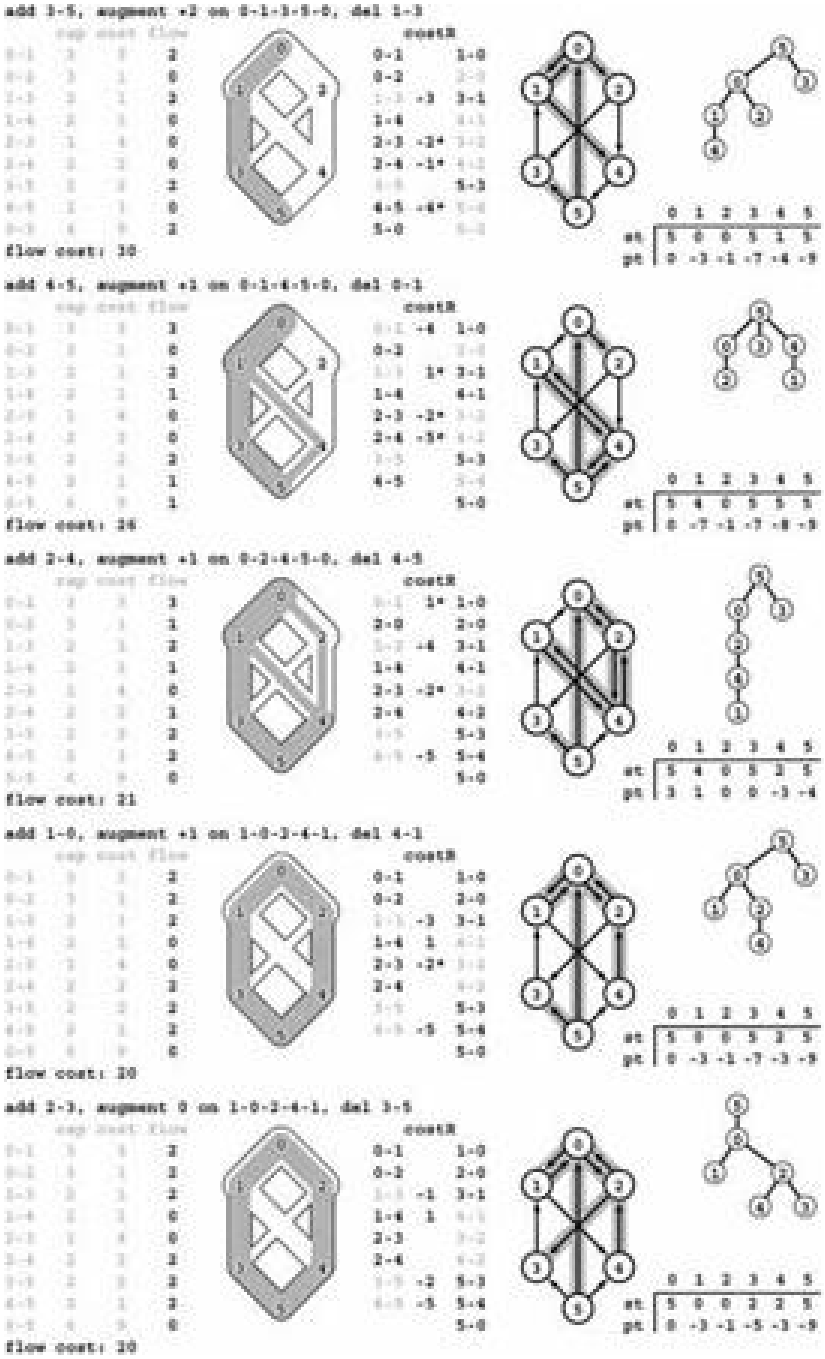
If there is more than one full or empty edge on the augmenting cycle, the substitution algorithm in *Program* 22.11 always deletes from the tree the one closest to the LCA of the eligible edge's two vertices. Fortunately, it has been shown that this particular strategy for choosing the edge to remove from the cycle suffices to ensure that the algorithm terminates (*see reference section*).

The final important choice that we face in developing an implementation of the network simplex algorithm is a strategy for identifying eligible edges and choosing one to add to the tree. Should we maintain a data structure containing eligible edges? If so, how sophisticated a data structure is appropriate? The answer to these questions depends somewhat

IPR2023-00124
KASPERSKY EX 1036 Page 546

on the application and the dynamics of solving particular instances of the problem. If the total number of eligible edges is small, then it is worthwhile to maintain a separate data structure; if most edges are eligible most of the time, it is not. Maintaining a separate data structure could spare us the expense of searching for eligible edges, but also could require costly update computations. What criterion should we use to pick from among the eligible edges? Again, there are many that we might adopt. We consider examples in our implementations, then we discuss alternatives.

Figure 22.51 Residual Networks and Spanning Trees (Network Simplex)
Each row in this figure corresponds to an iteration of the network simplex algorithm following the initialization depicted in *Figure* 22.50: On each iteration, it chooses an eligible edge, augments along a cycle, and updates the data structures as follows: First, the flow is augmented, including implied changes in the residual network. Second, the tree structure st is changed by adding an eligible edge and deleting an edge on the cycle that it makes with tree edges. Third, the table of potentials pt is updated to reflect the changes in the tree structure. Fourth, the reduced costs of the nontree edges (column marked costR in the center) are updated to reflect the potential changes, and these values used to identify empty edges with negative reduced cost and full edges with positive reduced costs as eligible edges (marked with asterisks on reduced costs). Implementations need not actually make all these computations (they only need to compute potential changes and reduced costs sufficient to identify an eligible edge), but we include all the numbers here to provide a full illustration of the algorithm. The final augmentation for this example is degenerate. It does not increase the flow, but it leaves no eligible edges, which guarantees that the flow is a mincost maxflow.



Program 22.12 Network simplex (basic implementation)

This ADT function uses the network simplex algorithm to solve the mincost flow problem. It is a straightforward implementation that recomputes all the vertex potentials and traverses the entire edge list to find an eligible edge of maximal reduced cost at each iteration. The R macro gives the reduced cost of a link's edge.

Program 22.12 is a full implementation of the network simplex algorithm that uses the strategy of choosing the eligible edge giving a negative cycle whose cost is highest in absolute value. The implementation depends on the tree-manipulation functions of Programs 22.9 through 22.11, but the comments that we made regarding our first cycle-canceling implementation (*Program* 22.8) apply: It is remarkable that such a simple piece of code is sufficiently powerful to provide useful solutions in the context of a general problem-solving model with the reach of the mincost-flow problem.

The worst-case performance bound for *Program* 22.12 is at least a factor of V lower than that for the cycle-canceling implementation in *Program* 22.8, because the time per iteration is just E (to find the eligible edge) rather than VE (to find a negative cycle). Although we might suspect that using the maximum augmentation will result in fewer augmentations than just taking the first negative cycle provided by the Bellman- Ford algorithm, that suspicion has not been proved valid. Specific bounds on the number of augmenting cycles used are difficult to develop, and, as usual, these bounds are far higher than the numbers that we see in practice. As mentioned earlier, there are theoretical results demonstrating that certain strategies can guarantee that the number of augmenting cycles is bounded by a polynomial in the number of edges, but practical implementations typically admit an exponential worst case.

In light of these considerations, there are many options to consider in pursuit of improved performance. For example, *Program* 22.13 is another implementation of the network simplex algorithm. The straightforward implementation in *Program* 22.12 always takes time proportional to V to revalidate the tree potentials and always takes time proportional to E to find the eligible edge with the largest reduced cost. The implementation in *Program* 22.13 is designed to eliminate both of these costs in typical networks.

First, even if choosing the maximum edge leads to the fewest number of iterations, expending the effort of examining every edge to find the maximum edge may not be worthwhile. We could do numerous augmentations along short cycles in the time that it takes to scan all the edges. Accordingly, it is worthwhile to consider the strategy of using *any* eligible edge, rather than taking the time to find a particular one. In the worst case, we might have to examine all the edges or a substantial fraction of them to find an eligible edge, but we typically expect to need to examine relatively few edges to find an eligible one. One approach is to start from the beginning each time; another is to pick a random starting point (see Exercise 22.127). This use of randomness also makes an artificially long sequence of augmenting paths unlikely.

IPR2023-00124
KASPERSKY EX 1036 Page 549

Program 22.13 Network simplex (improved implementation)

This network simplex implementation saves time on each iteration by calculating potentials only when needed, and takes the first eligible edge that it finds.

```
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);
   }
```

Second, we adopt a *lazy* approach to computing potentials. Rather than compute all the potentials in the vertex-indexed array `phi`, then refer to them when we need them, we call the function `phiR` to get each potential value; it travels up the tree to find a valid potential, then computes the necessary potentials on that path. To implement this approach, we simply change the macro that defines the cost to use the function call `phiR(u)`, instead of the array access `phi[u]`. In the worst case, we calculate all the potentials in the same way as before; but if we examine only a few eligible edges, then we calculate only those potentials that we need to identify them.

Such changes do not affect the worst-case performance of the algorithm, but they certainly speed it up in practical applications. Several other ideas for improving the performance of the network simplex algorithm are explored in the exercises (see Exercises 22.127 through 22.131); those represent only a small sample of those that have been proposed.

As we have emphasized throughout this book, the task of analyzing and comparing graph algorithms is complex. With the network simplex algorithm, the task is further compli-

IPR2023-00124
KASPERSKY EX 1036 Page 550

cated by the variety of different implementation approaches and the broad array of types of applications that we might encounter (see section 22.5). Which implementation is best? Should we compare implementations based on the worst-case performance bounds that we can prove? How accurately can we quantify performance differences of various implementations, for specific applications. Should we use multiple implementations, each tailored to specific applications?

Readers are encouraged to gain computational experience with various network-simplex implementations and to address some of these questions by running empirical studies of the kind that we have emphasized throughout this book. When seeking to solve mincost flow problems, we are faced with the familiar fundamental challenges, but the experience that we have gained in tackling increasingly difficult problems throughout this book provides ample background to develop efficient implementations that can effectively solve a broad variety of important practical problems. Some such studies are described in the exercises at the end of this and the next section, but these exercises should be viewed as a starting point. Each reader can craft a new empirical study that sheds light on some particular implementation/application pair of interest.

The potential to improve performance dramatically for critical applications through proper deployment of classic data structures and algorithms (or development of new ones) for basic tasks makes the study of network-simplex implementations a fruitful research area, and there is a large literature on network-simplex implementations. In the past, progress in this research has been crucial, because of the huge cost of solving network simplex problems. People tend to rely on carefully crafted libraries to attack these problems, and that is still appropriate in many situations. However, it is difficult for such libraries to keep up with recent research and to adapt to the variety of problems that arise in new applications. With the speed and size of modern computers, accessible implementations like Programs 22.12 and 22.13 can be starting points for the development of effective problem-solving tools for numerous applications.

Exercises

▷ **22.118** Give a maxflow with associated feasible spanning tree for the flow network shown in *Figure* 22.11.

22.119 Implement a function for the adjacency-lists representation of the flow ADT (*Program* 22.1) that removes cycles of partial edges from the flow and builds a feasible spanning tree for the resulting flow, as illustrated in *Figure* 22.45. Your function should take a network and an array as argument and should return a parent-link representation of the tree (using our standard convention with links to list nodes instead of indices) in the array.

IPR2023-00124
KASPERSKY EX 1036 Page 551

22.120 In the example in *Figure* 22.47, show the effect of reversing the direction of the edge connecting 6 and 5 on the potential tables.

○ **22.121** Construct a flow network and exhibit a sequence of augmenting edges such that the generic network simplex algorithm does not terminate.

▷ **22.122** Show, in the style of *Figure* 22.48, the process of computing potentials for the tree rooted at 0 in *Figure* 22.47.

22.123 Show, in the style of *Figure* 22.51, the process of computing a min-cost maxflow in the flow network shown in *Figure* 22.11, starting with the basic maxflow and associated basic spanning tree that you found in Exercise 22.118.

○ **22.124** Suppose that all nontree edges are empty. Write a function that takes as arguments two vertex-indexed arrays (a parent-link representation of a basic spanning tree and an output array) and computes the flows in the tree edges, putting the flow in the edge connecting v and its parent in the tree in the vth entry of the output array.

○ **22.125** Do Exercise 22.124 for the case where some nontree edges may be full.

22.126 Use *Program* 22.11 as the basis for an MST algorithm. Run empirical tests comparing your implementation with the three basic MST algorithms described in Chapter 20 (see Exercise 20.66).

22.127 Describe how to modify *Program* 22.13 such that it starts the scan for an eligible edge at a random edge rather than at the beginning each time.

22.128 Modify your solution to Exercise 22.127 such that each time it searches for an eligible edge it starts where it left off in the previous search.

22.129 Modify the auxiliary functions in this section to maintain a triply linked tree structure that includes each node's parent, leftmost child, and right sibling (see Section 5.4). Your functions to augment along a cycle and substitute an eligible edge for a tree edge should take time proportional to the length of the augmenting cycle and your function to compute potentials should take time proportional to the size of the smaller of the two subtrees created when the tree edge is deleted.

○ **22.130** Modify the auxiliary functions in this section to maintain, in addition to the parent-link array, two other vertex-indexed arrays: one containing each vertex's distance to the root, the other containing each vertex's successor in a DFS. Your functions to augment along a cycle and substitute an eligible edge for a tree edge should take time proportional to the length of the augmenting cycle and your function to compute potentials

should take time proportional to the size of the smaller of the two subtrees created when the tree edge is deleted.

●**22.131** Explore the idea of maintaining a generalized queue of eligible edges. Consider various generalized-queue implementations and various improvements to avoid excessive edge-cost calculations, such as restricting attention to a subset of the eligible edges so as to limit the size of the queue or possibly allowing some ineligible edges to remain on the queue.

●**22.132** Run empirical studies to determine the number of iterations, the number of vertex potentials calculated, and the ratio of the running time to E for several versions of the network simplex algorithm, for various networks (see Exercises 22.7–12). Consider various algorithms described in the text and in the previous exercises, and concentrate on those that perform the best on huge sparse networks.

●**22.133** Write a client program that does dynamic graphical animations of network simplex algorithms. Your program should produce images like those in *Figure* 22.51 and the other figures in this section (see Exercise 22.50). Test your implementation for the Euclidean networks among Exercises 22.7–12.

22.7 MINCOST-FLOW REDUCTIONS

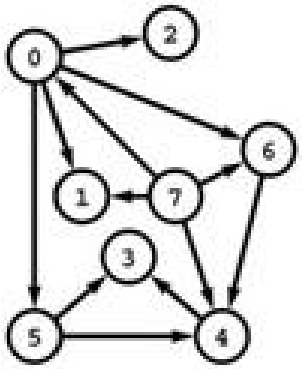
Mincost flow is a general problem-solving model that can encompass a variety of useful practical problems. In this section, we substantiate this claim by proving specific reductions from a variety of problems to mincost flow.

The mincost-flow problem is obviously more general than the maxflow problem, since any mincost maxflow is an acceptable solution to the maxflow problem. Specifically, if we assign to the dummy edge a cost 1 and to all other edges a cost 0 in the construction of *Figure* 22.43, any mincost maxflow minimizes the flow in the dummy edge and therefore maximizes the flow in the original network. Therefore, all the problems discussed in section 22.4 that reduce to the maxflow problem also reduce to the mincost-flow problem. This set of problems includes bipartite matching, feasible flow, and mincut, among numerous others.

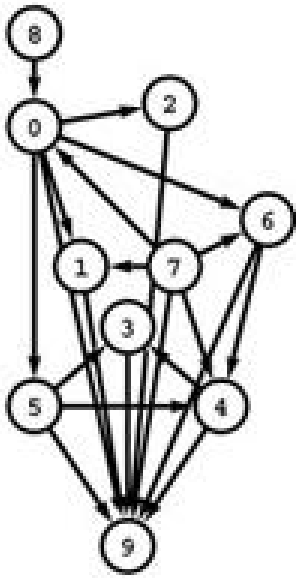
Figure 22.52 Reduction from Shortest Paths Finding a single-source shortest paths tree in the network at the top is equivalent to solving the mincost-maxflow problem in the flow network at the bottom.

IPR2023-00124
KASPERSKY EX 1036 Page 553

```
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
```



More interesting, we can examine properties of our algorithms for the mincost-flow problem to develop new generic algorithms for the maxflow problem. We have already noted that the generic cycle-canceling algorithm for the mincost-maxflow problem gives a generic augmenting-path algorithm for the maxflow problem. In particular, this approach leads to an implementation that finds augmenting paths without having to search the network (see Exercises 22.134 and 22.135). On the other hand, the algorithm might produce zero-flow augmenting paths, so its performance characteristics are difficult to evaluate (*see reference section*).

The mincost-flow problem is also more general than the shortest-paths problem, by the

following simple reduction:

Property 22.29 *The single-source shortest-paths problem (in networks with no negative cycles) reduces to the mincost–feasible-flow problem.*

Proof: Given a single-source shortest-paths problem (a network and a source vertex `s`), build a flow network with the same vertices, edges, and edge costs; and give each edge unlimited capacity. Add a new source vertex with an edge to `s` that has cost zero and capacity `V-1` and a new sink vertex with edges from each of the other vertices with costs zero and capacities 1. This construction is illustrated in *Figure* 22.52.

Solve the mincost–feasible-flow problem on this network. If necessary, remove cycles in the solution to produce a spanning-tree solution. This spanning tree corresponds directly to a shortest-paths spanning tree of the original network. Detailed proof of this fact is left as an exercise (see Exercise 22.139).■

Thus, all the problems discussed in section 21.6 that reduce to the single-source shortest-paths problem also reduce to the mincost-flow problem. This set of problems includes job scheduling with deadlines and difference constraints, among numerous others.

As we found when studying maxflow problems, it is worthwhile to consider the details of the operation of the network simplex algorithm when solving a shortest-paths problem using the reduction of Property 22.29. In this case, the algorithm maintains a spanning tree rooted at the source, much like the search-based algorithms that we considered in Chapter 21, but the node potentials and reduced costs provide increased flexibility in developing methods to choose the next edge to add to the tree.

We do not generally exploit the fact that the mincost-flow problem is a proper generalization of both the maxflow and the shortest-paths problems, because we have specialized algorithms with better performance guarantees for both problems. If such implementations are not available, however, a good implementation of the network simplex algorithm is likely to produce quick solutions to particular instances of both problems. Of course, we must avoid reduction loops when using or building network-processing systems that take advantage of such reductions. For example, the cycle-canceling implementation in *Program* 22.8 uses both maxflow and shortest paths to solve the mincost-flow problem (see Exercise 21.96).

Next, we consider equivalent network models. First, we show that assuming that costs are nonnegative is not restrictive, as we can transform networks with negative costs into networks without them.

Property 22.30 *In mincost-flow problems, we can assume, without loss of generality,*

that edge costs are nonnegative.

Proof: We prove this fact for feasible mincost flows in distribution networks. The same result is true for mincost maxflows because of the equivalence of the two problems proved in Property 22.22 (see Exercises 22.144 and 22.145).

Given a distribution network, we replace any edge u-v that has cost $x < 0$ and capacity c by an edge v-u of the same capacity that has cost $-x$ (a positive number). Furthermore, we can decrement the supply-demand value of u by c and increment the supply-demand value of v by c. This operation corresponds to pushing c units of flow from u to v and adjusting the network accordingly.

For negative-cost edges, if a solution to the mincost-flow problem for the transformed network puts flow f in the edge v-u, we put flow $c - f$ in u-v in the original network; for positive-cost edges, the transformed network has the same flow as in the original. This flow assignment preserves the supply or demand constraint at all the vertices.

The flow in v-u in the transformed network contributes fx to the cost and the flow in u-v in the original network contributes $-cx + fx$ to the cost. Since the first term in this expression does not depend on the flow, the cost of any flow in the transformed network is equal to the cost of the corresponding flow in the original network plus the sum of the products of the capacities and costs of all the negative-cost edges (which is a negative quantity). Any minimal-cost flow in the transformed network is a minimal-cost flow in the original network.■

This reduction shows that we can restrict attention to positive costs, but we generally do not bother to do so in practice because our implementations in Sections 22.5 and 22.6 work exclusively with residual networks and handle negative costs with no difficulty. It is important to have *some* lower bound on costs in some contexts, but that bound does not need to be zero (see Exercise 22.146).

Next, we show, as we did for the maxflow problem, that we could, if we wanted, restrict attention to acyclic networks. Moreover, we can also assume that edges are uncapacitated (there is no upper bound on the amount of flow in the edges). Combining these two variations leads to the following classic formulation of the mincost-flow problem.

Transportation Solve the mincost-flow problem for a bipartite distribution network where all edges are directed from a supply vertex to a demand vertex and have unlimited capacity. As discussed at the beginning of this chapter (see *Figure* 22.2), the usual way to think of this problem is as modeling the distribution of goods from warehouses (supply vertices) to retail outlets (demand vertices) along distribution channels (edges) at a cer-

tain cost per unit amount of goods.

Property 22.31 *The transportation problem is equivalent to the mincost-flow problem.*

Proof: Given a transportation problem, we can solve it by assigning a capacity for each edge higher than the supply or demand values of the vertices that it connects and solving the resulting mincost–feasible-flow problem on the resulting distribution network. Therefore, we need only to establish a reduction from the standard problem to the transportation problem.

For variety, we describe a new transformation, which is linear only for sparse networks. A construction similar to the one that we used in the proof of Property 22.16 establishes the result for nonsparse networks (see Exercise 22.149).

Given a standard distribution network with V vertices and E edges, build a transportation network with V supply vertices, E demand vertices and $2E$ edges, as follows. For each vertex in the original network, include a vertex in the bipartite network with supply or demand value set to the original value plus the sum of the capacities of the outgoing edges; and for each edge `u-v` in the original network with capacity c, include a vertex in the bipartite network with supply or demand value -c (we use the notation `[u-v]` to refer to this vertex). For each edge `u-v` in the original network, include two edges in the bipartite network: one from u to `[u-v]` with the same cost, and one from v to `[u-v]` with cost `0`.

The following one-to-one correspondence preserves costs between flows in the two networks: An edge `u-v` has flow value f in the original network if and only if edge `u-[u-v]` has flow value f, and edge `v-[u-v]` has flow value `c-f` in the bipartite network (those two flows must sum to c because of the supply–demand constraint at vertex `[u-v]`. Thus, any mincost flow in one network corresponds to a mincost flow in the other.■

Since we have not considered direct algorithms for solving the transportation problem, this reduction is of academic interest only. To use it, we would have to convert the resulting problem back to a (different) mincost-flow problem, using the simple reduction mentioned at the beginning of the proof of Property 22.31. Perhaps such networks admit more efficient solutions in practice; perhaps they do not. The point of studying the equivalence between the transportation problem and the mincost-flow problem is to understand that removing capacities and restricting attention to bipartite networks would seem to simplify the mincost-flow problem substantially; however, that is not the case.

We need to consider another classical problem in this context. It generalizes the bipartite-matching problem that is discussed in detail in section 22.4. Like that problem, it is deceptively simple:

Assignment Given a weighted bipartite graph, find a set of edges of minimum total weight such that each vertex is connected to exactly one other vertex.

For example, we might generalize our job-placement problem to include a way for each company to quantify its desire for each applicant (by, say, assigning an integer to each applicant, with lower integers going to the better applicants) and for each applicant to quantify his or her desire for each company. Then, a solution to the assignment problem would provide a reasonable way to take these relative preferences into account.

Property 22.32 *The assignment problem reduces to the mincost-flow problem.*

This result can be established via a simple reduction to the transportation problem. Given an assignment problem, construct a transportation problem with the same vertices and edges, with all vertices in one of the sets designated as supply vertices with value 1 and all vertices in the other set designated as demand vertices with value 1. Assign capacity 1 to each edge and assign a cost corresponding to that edge's weight in the assignment problem. Any solution to this instance of the transportation problem is simply a set of edges of minimal total cost that each connect a supply vertex to a demand vertex and therefore corresponds directly to a solution to the original assignment problem.

Reducing this instance of the transportation problem to the mincost-maxflow problem gives a construction that is essentially equivalent to the construction that we used to reduce the bipartite-matching problem to the maxflow problem (see Exercise 22.159).■

This relationship is *not* known to be an equivalence. There is no known way to reduce a general mincost-flow problem to the assignment problem. Indeed, like the single-source shortest-paths problem and the maxflow problem, the assignment problem seems to be easier than the mincost-flow problem in the sense that algorithms that solve it are known that have better asymptotic performance than the best known algorithms for the mincost-flow problem. Still, the network simplex algorithm is sufficiently well refined that a good implementation of it is a reasonable choice for solving assignment problems. Moreover, as with maxflow and shortest paths, we can tailor the network simplex algorithm to get improved performance for the assignment problem (*see reference section*).

Our next reduction to the mincost-flow problem brings us back to a basic problem related to paths in graphs like the ones that we first considered in Section 17.7. As we did in the Euler-path problem, we want a `path` that includes all the edges in a graph. Recognizing that not all graphs have such a path, we relax the restriction that edges must appear only once.

Mail carrier Given a network (weighted digraph), find a cyclic `path` of minimal weight

that includes each edge at least once (see *Figure* 22.53). Recall that our basic definitions in Chapter 17 make the distinction between cyclic paths (which may revisit vertices and edges) and cycles (which consist of distinct vertices, except the first and the final, which are the same).

The solution to this problem would describe the best route for a mail carrier (who has to cover all the streets on her route) to follow. A solution to this problem might also describe the route that a snow-plow should take during a snowstorm, and there are many similar applications.

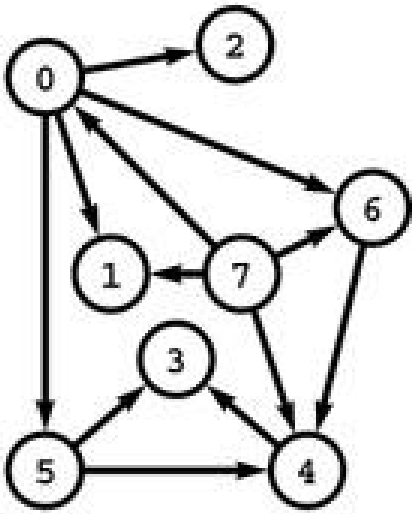
The mail carrier's problem is an extension of the Euler tour problem that we discussed in Section 17.7: The solution to Exercise 17.92 is a simple test for whether a digraph has an Euler tour, and *Program* 17.14 is an effective way to find an Euler tour for such digraphs. That tour solves the mail carrier's problem because it includes each edge exactly once—no `path` could have lower weight. The problem becomes more difficult when indegrees and outdegrees are not necessarily equal. In the general case, some edges must be traversed more than once: The problem is to minimize the total weight of all the multiply traversed edges.

Property 22.33 *The mail carrier's problem reduces to the mincost-flow problem.*

Proof: Given an instance of the mail carrier's problem (a weighted digraph), define a distribution network on the same vertices and edges, with all the vertex supply or demand values set to 0, edge costs set to the weights of the corresponding edge, and no upper bounds on edge capacities, but all edge capacities constrained to be greater than 1. We interpret a flow value f in an edge `u-v` as saying that the mail carrier needs to traverse `u-v` a total of f times.

Figure 22.53 Mail Carrier's Problem. Finding the shortest `path` that includes each edge at least once is a challenge even for this small network, but the problem can be solved efficiently through reduction to the mincost-flow problem.

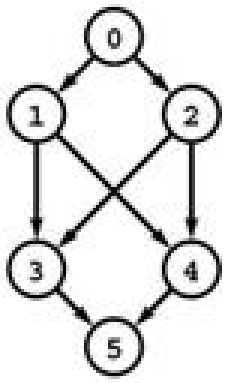
```
0-1   4
1-2   5
2-3   5
4-3   3
3-5   3
3-0   4
0-5   2
5-4   2
1-4   3
4-2   3
5-1   2
```



Find a mincost flow for this network by using the transformation of Exercise 22.147 to remove the lower bound on edge capacities. The flow-decomposition theorem says that we can express the flow as a set of cycles, so we can build a cyclic `path` from this flow in the same way that we built an Euler tour in an Eulerian graph: We traverse any cycle, taking a detour to traverse another cycle whenever we encounter a node that is on another cycle.■

Figure 22.54 LP Formulation of a Mincost-Maxflow Problem This linear program is equivalent to the mincost-maxflow problem for the sample network of *Figure* 22.41. The edge equalities and vertex inequalities are the same as in *Figure* 22.40, but the objective is different. The variable c represents the total cost, which is a linear combination of the other variables. In this case, $c = -9x_{50} + 3x_{01} + x_{02} + x_{13} + x_{14} + 4x_{23} + 2x_{24} + 2x_{35} + x_{45}$.

	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



Maximize $-c$
subject to the constraints

$$
\begin{aligned}
x_{01} &\leq 2 \\
x_{02} &\leq 3 \\
x_{13} &\leq 3 \\
x_{14} &\leq 1 \\
x_{23} &\leq 1 \\
x_{24} &\leq 1 \\
x_{35} &\leq 2 \\
x_{45} &\leq 3 \\
x_{50} &= x_{01} + x_{02} \\
x_{01} &= x_{13} + x_{14} \\
x_{02} &= x_{23} + x_{24} \\
x_{13} + x_{23} &= x_{35} \\
x_{14} + x_{24} &= x_{45} \\
x_{35} + x_{45} &= x_{50}
\end{aligned}
$$

A careful look at the mail carrier's problem illustrates yet again the fine line between trivial and intractable in graph algorithms. Suppose that we consider the two-way version of the problem where the network is undirected and the mail carrier must travel in both directions along each edge. Then, as we noted in Section 18.5, depth-first search (or any graph search) will provide an immediate solution. If, however, it suffices to traverse each undirected edge in either direction, then a solution is significantly more difficult to formulate than is the simple reduction to mincost flow that we just examined, but the problem is still tractable. If some edges are directed and others undirected, the problem becomes NP-hard (*see reference section*).

These are only a few of the scores of practical problems that have been formulated as mincost-flow problems. The mincost-flow problem is even more versatile than the maxflow problem or shortest-paths problems, and the network simplex algorithm effectively solves all problems encompassed by the model.

As we did when we studied maxflow, we can examine how any mincost-flow problem can be cast as an LP problem, as illustrated in *Figure* 22.54. The formulation is a straightforward extension of the maxflow formulation: We add equations that set a dummy variable to be equal to the cost of the flow, then set the objective so as to minimize that variable. LP models allow addition of arbitrary (linear) constraints. Some constraints may lead to problems that still may be equivalent to mincost-flow problems, but others do not. That is, many problems do not reduce to mincost-flow problems: In particular, LP encompasses a much broader set of problems. The mincost-flow problem is a next step toward that general problem-solving model, which we consider in Part 8.

There are other models that are even more general than the LP model; but LP has the additional virtue that, while LP problems are in general more difficult than mincost-flow problems, effective and efficient algorithms have been invented to solve them. Indeed, perhaps the most important such algorithm is known as the *simplex* method: the network simplex method is a specialized version of the simplex method that applies to the subset of LP problems that correspond to mincost-flow problems, and understanding the network simplex algorithm is a helpful first step in understanding the simplex algorithm.

Exercises

○ **22.134** Show that, when the network simplex algorithm is computing a maxflow, the spanning tree is the union of `t-s`, a tree containing s and a tree containing t.

22.135 Develop a maxflow implementation based on Exercise 22.134. Choose an eligible edge at random.

22.136 Show, in the style of *Figure* 22.51, the process of computing a maxflow in the flow network shown in *Figure* 22.11 using the reduction described in the text and the network-simplex implementation of *Program* 22.12.

22.137 Show, in the style of *Figure* 22.51, the process of finding shortest paths from 0 in the flow network shown in *Figure* 22.11 using the reduction described in the text and the network-simplex implementation of *Program* 22.12.

○ **22.138** Prove that all edges in the spanning tree described in the proof of Property 22.29 are on paths directed from the source to leaves.

IPR2023-00124
KASPERSKY EX 1036 Page 562

22.139 Prove that the spanning tree described in the proof of Property 22.29 corresponds to a shortest-paths tree in the original network.

22.140 Suppose that you use the network simplex algorithm to solve a problem created by a reduction from the single-source shortest-paths problem as described in the proof of Property 22.29. (*i*) Prove that the algorithm never uses a zero-cost augmenting path. (*ii*) Show that the edge that leaves the cycle is always the parent of the destination vertex of the edge that is added to the cycle. (*iii*) As a consequence of Exercise 22.139, the network simplex algorithm does not need to maintain edge flows. Provide a full implementation that takes advantage of this observation. Choose the new tree edge at random.

22.141 Suppose that we assign a positive cost to each edge in a network. Prove that the problem of finding a single-source shortest-paths tree of minimal cost reduces to the mincost-maxflow problem.

22.142 Suppose that we modify the job-scheduling-with-deadlines problem in section 21.6 to stipulate that jobs can miss their deadlines, but that they incur a fixed positive cost if they do. Show that this modified problem reduces to the mincost-maxflow problem.

22.143 Implement an ADT function that finds mincost maxflows in distribution networks with negative costs using your solution to Exercise 22.108, which assumes that costs are all nonnegative.

22.144 Suppose that the costs of 0-2 and 1-3 in *Figure* 22.41 are -1, instead of 1. Show how to find a mincost maxflow by transforming the network to a network with positive costs and finding a mincost maxflow of the new network.

22.145 Implement an ADT function that finds mincost maxflows in networks with negative costs using `GRAPHmincost`, which assumes that costs are all nonnegative.

22.146 Do the implementations in Sections 22.5 and 22.6 depend in a fundamental way on costs being nonnegative? If they do, explain how; if they do not, explain what fixes (if any) are required to make them work properly for networks with negative costs or explain why no such fixes are possible.

22.147 Extend your feasible-flow ADT from Exercise 22.75 to include lower bounds on the capacities of edges. Provide an ADT implementation that computes a mincost maxflow that respects these bounds or prove that none exists.

22.148 Give the result of using the reduction in the text to reduce the flow network described in Exercise 22.114 to the transportation problem.

IPR2023-00124
KASPERSKY EX 1036 Page 563

○**22.149** Show that the mincost-maxflow problem reduces to the transportation problem with just V extra vertices and edges, by using a construction similar to the one used in the proof of Property 22.16.

▷**22.150** Develop an ADT implementation for the transportation problem that is based on the simple reduction to the mincost-flow problem given in the proof of Property 22.30.

○**22.151** Develop an ADT implementation for the mincost-flow problem that is based on the reduction to the transportation problem described in the proof of Property 22.31.

○**22.152** Develop an ADT implementation for the mincost-flow problem that is based on the reduction to the transportation problem described in Exercise 22.149.

22.153 Write a program to generate random instances of the transportation problem, then use them as the basis for empirical tests on various algorithms and implementations to solve that problem.

22.154 Find a large instance of the transportation problem online.

22.155 Run empirical studies to compare the two different methods of reducing arbitrary mincost-flow problems to the transportation problem that are discussed in the proof of Property 22.31.

22.156 Write a program to generate random instances of the assignment problem, then use them as the basis for empirical tests on various algorithms and implementations to solve that problem.

22.157 Find a large instance of the assignment problem online.

22.158 The job-placement problem described in the text favors the employers (their total weights are maximized). Formulate a version of the problem such that applicants also express their wishes. Explain how to solve your version.

22.159 Do empirical studies to compare the performance of the two network-simplex implementations in section 22.6 for solving random instances of the assignment problem (see Exercise 22.156) with V vertices and E edges, for a reasonable set of values for V and E.

22.160 The mail carrier's problem clearly has no solution for networks that are not strongly connected (the mail carrier can visit only those vertices that are in the strong component where she starts), but that fact is not mentioned in the reduction of Property 22.33. What happens when we use the reduction on a network that is not strongly con-

nected?

22.161 Run empirical studies for various weighted graphs (see Exercise 21.4–8) to determine average length of the mail carrier's path.

22.162 Give a direct proof that the single-source shortest-paths problem reduces to the assignment problem.

22.163 Describe how to formulate an arbitrary assignment problem as an LP problem.

○ **22.164** Do Exercise 22.20 for the case where the cost value associated with each edge is -1 (so you minimize unused space in the trucks).

○ **22.165** Devise a cost model for Exercise 22.20 such that the solution is a maxflow that takes a minimal number of days.

22.8 PERSPECTIVE

Our study of graph algorithms appropriately culminates in the study of network-flow algorithms for four reasons. First, the network-flow model validates the practical utility of the graph abstraction in countless applications. Second, the maxflow and mincost-flow algorithms that we have examined are natural extensions of graph algorithms that we studied for simpler problems. Third, the implementations exemplify the important role of fundamental algorithms and data structures in achieving good performance. Fourth, the maxflow and mincost-flow models illustrate the utility of the approach of developing increasingly general problem-solving models and using them to solve broad classes of problems. Our ability to develop efficient algorithms that solve these problems leaves the door open for us to develop more general models and to seek algorithms that solve those problems.

Before considering these issues in further detail, we develop further context by listing important problems that we have *not* covered in this chapter, even though they are closely related to familiar problems.

Maximum matching In a graph with edge weights, find a subset of edges in which no vertex appears more than once and whose total weight is such that no other such set of edges has a higher total weight. We can reduce the *maximum-cardinality matching* problem in unweighted graphs immediately to this problem, by setting all edge weights to 1.

The assignment problem and maximum-cardinality bipartite-matching problems reduce

IPR2023-00124
KASPERSKY EX 1036 Page 565

to maximum matching for general graphs. On the other hand, maximum matching does not reduce to mincost flow, so the algorithms that we have considered do not apply. The problem is tractable, although the computational burden of solving it for huge graphs remains significant. Treating the many techniques that have been tried for matching on general graphs would fill an entire volume: The problem is one of those studied most extensively in graph theory. We have drawn the line in this book at mincost flow, but we revisit maximum matching in Part 8.

Multicommodity flow Suppose that we need to compute a *second* flow such that the sum of an edge's two flows is limited by that edge's capacity, both flows are in equilibrium, and the total cost is minimized. This change models the presence of two different types of material in the merchandise-distribution problem; for example, should we put more hamburger or more potatoes in the truck bound for the fast-food restaurant? This change also makes the problem much more difficult and requires more advanced algorithms than those considered here; for example, no analogue to the maxflow–mincut theorem is known to hold for the general case. Formulating the problem as an LP problem is a straightforward extension of the example shown in *Figure* 22.54, so the problem is tractable (because LP is tractable).

Convex and nonlinear costs The simple cost functions that we have been considering are linear combinations of variables, and our algorithms for solving them depend in an essential way on the simple mathematical structure underlying these functions. Many applications call for more complicated functions. For example, when we minimize distances, we are led to sums of squares of costs. Such problems cannot be formulated as LP problems, so they require problem-solving models that are even more powerful. Many such problems are not tractable.

Scheduling We have presented a few scheduling problems as examples. They are barely representative of the hundreds of different scheduling problems that have been posed. The research literature is replete with the study of relationships among these problems and the development of algorithms and implementations to solve the problems (*see reference section*). Indeed, we might have chosen to use scheduling rather than network-flow algorithms to develop the idea for defining general problem-solving models and implementing reductions to solve particular problems (the same might be said of matching). Many scheduling problems reduce to the mincost-flow model.

The scope of combinatorial computing is vast, indeed, and the study of problems of this sort is certain to occupy researchers for many years to come. We revisit many of these problems in Part 8, in the context of coping with intractability.

We have presented only a fraction of the studied algorithms that solve maxflow and min-

IPR2023-00124
KASPERSKY EX 1036 Page 566

cost-flow problems. As indicated in the exercises throughout this chapter, combining the many options available for different parts of various generic algorithms leads to a large number of different algorithms. Algorithms and data structures for basic computational tasks play a significant role in the efficacy of many of these approaches; indeed, some of the important general-purpose algorithms that we have studied were developed in the quest for efficient implementations of network-flow algorithms. This topic is still being studied by many researchers. The development of better algorithms for network-flow problems certainly depends on intelligent use of basic algorithms and data structures.

The broad reach of network-flow algorithms and our extensive use of reductions to extend this reach makes this section an appropriate place to consider some implications of the concept of reduction. For a large class of combinatorial algorithms, these problems represent a watershed in our studies of algorithms, where we stand between the study of efficient algorithms for particular problems and the study of general problem-solving models. There are important forces pulling in both directions.

We are drawn to develop as general a model as possible, because the more general the model, the more problems it encompasses, thereby increasing the usefulness of an efficient algorithm that can solve any problem that reduces to the model. Developing such an algorithm may be a significant, if not impossible, challenge. Even if we do not have an algorithm that is guaranteed to be reasonably efficient, we typically have good algorithms that perform well for specific classes of problems that are of interest. Specific analytic results are often elusive, but we often have persuasive empirical evidence. Indeed, practitioners typically will try the most general model available (or one that has a well-developed solution package) and will look no further if the model works in reasonable time. However, we certainly should strive to avoid using overly general models that lead us to spend excessive amounts of time solving problems for which more specialized models can be effective.

We are also drawn to seek better algorithms for important specific problems, particularly for huge problems or huge numbers of instances of smaller problems where computational resources are a critical bottleneck. As we have seen for numerous examples throughout this book and in Parts 1 through 4, we often can find a clever algorithm that can reduce resource costs by factors of hundreds or thousands or more, which is extremely significant if we are measuring costs in hours or dollars. The general outlook described in Chapter 2, which we have used successfully in so many domains, remains extremely valuable in such situations, and we can look forward to the development of clever algorithms throughout the spectrum of graph algorithms and combinatorial algorithms. Perhaps the most important drawback to depending too heavily on a specialized algorithm is that often a small change to the model will invalidate the algorithm. When we use an overly general model and an algorithm that gets our problem solved, we are less

IPR2023-00124
KASPERSKY EX 1036 Page 567

vulnerable to this defect.

Software libraries that encompass many of the algorithms that we have addressed may be found in many programming environments. Such libraries certainly are important resources to consider for specific problems. However, libraries may be difficult to use, obsolete, or poorly matched to the problem at hand. Experienced programmers know the importance of considering the tradeoff between taking advantage of a library resource and becoming overly dependent on that resource (if not subject to premature obsolescence). Some of the implementations that we have considered are efficient, simple to develop, and broad in scope. Adapting and tuning such implementations to address problems at hand can be the proper approach in many situations.

The tension between theoretical studies that are restricted to what we can prove and empirical studies that are relevant to only the problems at hand becomes ever more pronounced as the difficulty of the problems that we address increases. The theory provides the guidance that we need to gain a foothold on the problem, and practical experience provides the guidance that we need to develop implementations. Moreover, experience with practical problems suggests new directions for the theory, perpetuating the cycle that expands the class of practical problems that we can solve.

Ultimately, whichever approach we pursue, the goal is the same: We want a broad spectrum of problem-solving models, effective algorithms for solving problems within those models, and efficient implementations of those algorithms that can handle practical problems. The development of increasingly general problem-solving models (such as the shortest paths, maxflow, and mincost-flow problems), the increasingly powerful generic algorithms (such as the Bellman–Ford algorithm for the shortest-paths problem, the augmenting-path algorithm for the maxflow problem, and the network simplex algorithm for the mincost-maxflow problem) brought us a long way towards the goal. Much of this work was done in the 1950s and 1960s. The subsequent emergence of fundamental data structures (Parts 1 through 4) and of algorithms that provide effective implementations of these generic methods (this book) has been an essential force leading to our current ability to solve such a large class of huge problems.

IPR2023-00124
KASPERSKY EX 1036 Page 568

References for Part Five

The algorithms textbooks listed below cover most of the basic graph-processing algorithms in Chapters 17 through 21. These books are basic references that give careful treatments of fundamental and advanced graph algorithms, with extensive references to the recent literature. The book by Even and the monograph by Tarjan are devoted to thorough coverage of many of the same topics that we have discussed. Tarjan's original paper on the application of depth-first search to solve strong connectivity and other problems merits further study. The source-queue topological sort implementation in Chapter 19 is from Knuth's book. Original references for some of the other specific algorithms that we have covered are listed below.

The algorithms for minimum spanning trees in dense graphs in Chapter 20 are quite old, but the original papers by Dijkstra, Prim, and Kruskal still make interesting reading. The survey by Graham and Hell provides a thorough and entertaining history of the problem. The paper by Chazelle is the state of the art in the quest for a linear MST algorithm.

The book by Ahuja, Magnanti, and Orlin is a comprehensive treatment of network flow algorithms (and shortest paths algorithms). Further information on nearly every topic covered in Chapters 21 and 22 may be found in that book. Another source for further material is the classic book by Papadimitriou and Steiglitz. Though most of that book is about much more advanced topics, it carefully treats many of the algorithms that we have discussed. Both books have extensive and detailed information about source material from the research literature. The classic work by Ford and Fulkerson is still worthy of study, as it introduced many of the fundamental concepts.

We have briefly introduced numerous advanced topics from (the yet to be published) Part 8, including reducibility, intractability, and linear programming, among several others. This reference list is focused on the material that we cover in detail and cannot do justice to these advanced topics. The algorithms texts treat many of them, and the book by Papadimitriou and Steiglitz provides a thorough introduction. There are numerous other books and a vast research literature on these topics.

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. Cormen, C. L. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, MIT Press and McGraw-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 strong and biconnected components,” *Information Processing Letters*, 74 (2000).

R. L. Graham and P. Hell, “On the history of the minimum spanning tree problem,” *Annals 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.