Algorithms

Part 5

GRAPH ALGORITHMS

ROBERT SEDGEWICK with C++ consulting by Christopher J. Van Wyk

Algorithms THIRD EDITION in C++

PART 5 GRAPH ALGORITHMS

Robert Sedgewick Princeton University

♣ Addison-Wesley

Boston • San Francisco • New York • Toronto • Montreal London • Munich • Paris • Madrid Capetown • Sydney • Tokyo • Singapore • Mexico City 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 Addison-Wesley was 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.

The publisher offers discounts on this book when ordered in quantity for special sales. For more information, please contact: Pearson Education Corporate Sales Division, 201 W. 103rd Street, Indianapolis, IN 46290, (800) 428-5331, corpsales@pearsontechgroup.com.

Visit AW on the Web at www.awl.com/cseng/ .

Library of Congress Cataloging-in-Publication Data

Sedgewick, Robert, 1946 –

Algorithms in C++ / Robert Sedgewick. — 3d ed.
p. cm.

Includes bibliographical references and index.
Contents: v. 2, pt. 5. Graph algorithms

C++ (Computer program language) 2. Computer algorithms.
I. Title.

QA76.73.C15S38 2002

Q05.13'3—dc20
92-901
CIP

Copyright © 2002 by Pearson Education, Inc.

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.

ISBN 0-201-36118-3

Text printed on recycled paper 7 8 9 10 – DOH – 070605 Seventh printing, February 2006

Preface

G RAPHS 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.

These books are not just for programmers and computer science students. 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 is the basis for 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 use 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 often 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 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 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 functionality 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. 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++. The programs use a wide range of standard C++ idioms, and the text includes concise descriptions of each construct.

Chris Van Wyk and I developed a style of C++ programming based on classes, templates, and overloaded operators that we feel is an effective way to present the algorithms and data structures as real programs. We have striven for elegant, compact, efficient, and portable implementations. The style is consistent whenever possible, so that programs that are similar look similar.

A goal of this book is to present the algorithms in as simple and direct a form as possible. 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 any of the countless other programming languages and environments in which it has proved to be an effective graph-processing method. On the one hand, our code is informed by experience with implementing algorithms in these and numerous other languages (a C version of this book is also available, and a Java version will appear soon); on the other hand, some of the properties of some of these languages are informed by their designers' experience with some of the algorithms and data structures that we consider in this book. In the end, we feel that the code presented in the book both precisely defines the algorithms and is useful in practice.

Acknowledgments

Many people gave me helpful feedback on earlier versions of this book. In particular, thousands 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 the figures in the book; to Dave Hanson and Andrew Appel for their willingness to answer my questions about programming languages; and to Kevin Wayne, for patiently answering my basic questions about networks. Kevin urged me to include the network simplex algorithm in this book, but I was not persuaded that it would be possible to do so until I saw a presentation by Ulrich Lauther at Dagstuhl of the ideas on which the implementations in Chapter 22 are based. 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 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. In particular, Marilyn Rash did an outstanding job managing the book's production within a very tightly compressed schedule.

I have gained three 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 Dupré 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. Third, Chris Van Wyk implemented and debugged all my algorithms in C++, answered numerous questions about C++, helped to develop an appropriate C++ programming style, and carefully read the manuscript twice. Chris also patiently stood by as I took apart many of his C++ programs and then, as I learned more and more about C++ from him, had to put them back together much as he had written them. I am extremely grateful for the opportunity to learn from Steve, Lyn, and Chris—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, 1983 Princeton, New Jersey, 1990 Jamestown, Rhode Island, 2001

C++ Consultant's Preface

Bob Sedgewick and I wrote many versions of most of these programs in our quest to implement graph algorithms in clear and natural programs. Because there are so many kinds of graphs and so many different questions to ask about them, we agreed early on not to pursue a single class scheme that would work across the whole book. Remarkably, we ended up using only two schemes: a simple one in Chapters 17 through 19, where the edges of a graph are either present or absent; and an approach similar to STL containers in Chapters 20 through 22, where more information is associated with edges.

C++ classes offer many advantages for presenting graph algorithms. We use classes to collect useful generic functions on graphs (like input/output). In Chapter 18, we use classes to factor out the operations common to several different graph-search methods. Throughout the book, we use an iterator class on the edges emanating from a vertex so that the programs work no matter how the graph is stored. Most important, we package graph algorithms in classes whose constructor processes the graph and whose member functions give us access to the answers discovered. This organization allows graph algorithms to readily use other graph algorithms as subroutines—see, for example, Program 19.13 (transitive closure via strong components), Program 20.8 (Kruskal's algorithm for minimum spanning tree), Program 21.4 (all shortest paths via Dijkstra's algorithm), Program 21.6 (longest path in a directed acyclic graph). This trend culminates in Chapter 22, where most of the programs are built at a higher level of abstraction, using classes that are defined earlier in the book.

For consistency with Algorithms in C++, Third Edition, Parts 1– 4 our programs rely on the stack and queue classes defined there, and we write explicit pointer operations on singly-linked lists in two lowlevel implementations. We have adopted two stylistic changes from Parts 1–4: Constructors use initialization rather than assignment and we use STL vectors instead of arrays. Here is a summary of the STL vector functions we use in our programs:

- The default constructor creates an empty vector.
- The constructor vec(n) creates a vector of n elements.
- The constructor vec(n, x) creates a vector of n elements each initialized to the value x.

- Member function vec.assign(n, x) makes vec a vector of n elements each initialized to the value x.
- Member function vec.resize(n) grows or shrinks vec to have capacity n.
- Member function vec.resize(n, x) grows or shrinks vec to have capacity n and initializes any new elements to the value x.

The STL also defines the assignment operator, copy constructor, and destructor needed to make vectors first-class objects.

Before I started working on these programs, I had read informal descriptions and pseudocode for many of the algorithms, but had only implemented a few of them. I have found it very instructive to work out the details needed to turn algorithms into working programs, and fun to watch them in action. I hope that reading and running the programs in this book will also help you to understand the algorithms better.

Thanks: to Jon Bentley, Brian Kernighan, and Tom Szymanski, from whom I learned much of what I know about programming; to Debbie Lafferty, who asked whether I would be interested in this project; and to Drew University, Lucent Technologies, and Princeton University, for institutional support.

> Christopher Van Wyk Chatham, New Jersey, 2001

This page intentionally left blank

To Adam, Andrew, Brett, Robbie, and especially Linda This page intentionally left blank

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:

▷ **18.34** 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 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:

• **19.106** 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:

• 20.73 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 on 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:

• 20.37 Develop a reasonable generator for random graphs with V vertices and E edges such that the running time of the heap-based PFS implementation of Dijkstra's algorithm is superlinear.

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:

• 17.74 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:

• **19.5** 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 of the topics that interest you than you could glean by simply reading the text.

Contents

Graph Algorithms

Chapter	17. Graph Properties and Types	3
17.1	Glossary · 7	
17.2	Graph ADT · 16	
17.3	Adjacency-Matrix Representation · 25	
17.4	Adjacency-Lists Representation · 31	
17.5	Variations, Extensions, and Costs · 36	
17.6	Graph Generators · 46	
17.7	Simple, Euler, and Hamilton Paths · 56	
17.8	Graph-Processing Problems · 70	
Chapter	18. Graph Search	81
18.1	Exploring a Maze · 82	
18.2	Depth-First Search · 87	
18.3	Graph-Search ADT Functions · 91	
18.4	Properties of DFS Forests · 98	

- 18.5 DFS Algorithms · 105 18.6 Separability and Biconnectivity · 112 18.7 Breadth-First Search · 121 18.8 Generalized Graph Search · 131 18.9 Analysis of Graph Algorithms · 140 Chapter 19. Digraphs and DAGs 149 19.1 Glossary and Rules of the Game · 152 19.2 Anatomy of DFS in Digraphs · 160 19.3 Reachability and Transitive Closure · 169 19.4 Equivalence Relations and Partial Orders · 182 19.5 DAGs · 186 19.6 Topological Sorting · 191 19.7 Reachability in DAGs · 201 19.8 Strong Components in Digraphs · 205 19.9 Transitive Closure Revisited · 216 19.10 Perspective · 221 Chapter 20. Minimum Spanning Trees 227 20.1 Representations · 230 20.2 Underlying Principles of MST Algorithms · 240 20.3 Prim's Algorithm and Priority-First Search · 247 20.4 Kruskal's Algorithm · 258 20.5 Boruvka's Algorithm · 264
 - 20.6 Comparisons and Improvements · 267
 - 20.7 Euclidean MST · 274

Chapter 21. Shortest Paths	277
21.1 Underlying Principles · 285	
21.2 Dijkstra's Algorithm · 293	
21.3 All-Pairs Shortest Paths · 304	
21.4 Shortest Paths in Acyclic Networks · 313	
21.5 Euclidean Networks · 322	
21.6 Reduction · 328	
21.7 Negative Weights · 345	
21.8 Perspective · 363	
Chapter 22. Network Flow	
22.1 Flow Networks · 373	
22.2 Augmenting-Path Maxflow Algorithms · 382	
22.3 Preflow-Push Maxflow Algorithms · 410	
22.4 Maxflow Reductions · 425	
22.5 Mincost Flows · 443	
22.6 Network Simplex Algorithm · 453	
22.7 Mincost-Flow Reductions · 472	
22.8 Perspective · 482	
References for Part Five	
Index	489

This page intentionally left blank

PART FIVE

Graph Algorithms

This page intentionally left blank

CHAPTER SEVENTEEN

Graph Properties and Types

M ANY 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 I 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?

GRAPH PROPERTIES AND TYPES

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 twodimensional 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 and 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 vector 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 selfloops 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 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.

§17.1

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.

For some applications, such as graphs that represent maps or circuits, a graph drawing can carry considerable information because



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.



§17.1

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. 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.*

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.

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

Figure 17.3 Graph 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.



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. 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 then 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*.

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.

§17.1

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 above by 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 evennumbered ones, so it is bipartite. The bottom diagram makes the property obvious.



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). 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 *tail* and *head*, 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?"

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.

 \triangleright 17.2 Give all the connected subgraphs of the graph

 \triangleright 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:

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?

- \triangleright 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.
- \circ **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

§17.2

We develop our graph-processing algorithms using an ADT that defines the fundamental tasks, using the standard mechanisms introduced in Chapter 4. Program 17.1 is 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 client programs and ADTs that access graphs through 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 allow clients to build graphs (by constructing the graph and then adding the edges), to maintain the

Program 17.1 Graph ADT interface

This interface is a starting point for implementing and testing graph algorithms. It defines two data types: a trivial Edge data type, including a constructor function that creates an Edge from two given vertices; and a GRAPH data type, which is defined with the standard representationindependent ADT interface methodology from Chapter 4.

The GRAPH constructor takes two arguments: an integer giving the number of vertices and a boolean that tells whether the graph is undirected or directed (a digraph), with undirected the default.

The basic operations that we use to process graphs and digraphs are ADT functions to create and destroy them; to report the number of vertices and edges; and to add and delete edges. The iterator class adjlterator allows clients to process each of the vertices adjacent to any given vertex. Programs 17.2 and 17.3 illustrate its use.

```
struct Edge
 \{ int v, w; \}
    Edge(int v = -1, int w = -1) : v(v), w(w) { }
 };
class GRAPH
  { private:
      // Implementation-dependent code
   public:
      GRAPH(int, bool);
      ~GRAPH();
      int V() const;
      int E() const;
      bool directed() const:
      int insert(Edge);
      int remove(Edge);
      bool edge(int, int);
      class adjIterator
        {
          public:
            adjIterator(const GRAPH &, int);
            int beg();
            int nxt();
            bool end();
        };
 };
```

graphs (by removing some edges and adding others), and to examine the graphs (using an iterator for processing the vertices adjacent to any given vertex).

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.

The graph constructor takes the maximum possible number of vertices in the graph as an argument, so that implementations can allocate memory accordingly. We adopt this convention solely to make the code compact and readable. A more general graph ADT might include in its interface the capability to add and remove vertices as well as edges; this would impose more demanding requirements on the data structures used to implement the ADT. 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 insert 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, including 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. We will revisit these issues when we examine graph representations in Sections 17.3 and 17.4.

Program 17.2 is a function that illustrates the use of the iterator class in the graph ADT. It is a function that extracts a graph's

§17.2

Program 17.2 Example of a graph-processing client function

This function shows one way to use the graph ADT to implement a basic graph-processing operation in a manner independent of the representation. It returns all the graph's edges in a vector.

This implementation illustrates the basis for most of the programs that we consider: we process each edge in the graph by checking all the vertices adjacent to each vertex. We generally do not call beg, end, and nxt *except* as illustrated here, so that we can better understand the performance characteristics of our implementations (see Section 17.5).

```
template <class Graph>
vector <Edge> edges(Graph &G)
{ int E = 0;
    vector <Edge> a(G.E());
    for (int v = 0; v < G.V(); v++)
        {
            typename Graph::adjIterator A(G, v);
            for (int w = A.beg(); !A.end(); w = A.nxt())
            if (G.directed() || v < w)
                a[E++] = Edge(v, w);
        }
    return a;
}</pre>
```

set of edges and returns it in a C++ Standard Template Library (STL) vector. 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. The order in which the edges appear in the vector is immaterial and will differ from implementation to implementation. We use a template for such functions to allow for using multiple implementations of the graph ADT.

Program 17.3 is another example of the use of the iterator class in the graph ADT, to print out a table of the vertices adjacent to each vertex, as shown in Figure 17.7. The code in these two examples is quite similar and is similar to the code in numerous graph-processing algorithms. Remarkably, we can build all of the algorithms that we consider in this book on this basic abstraction of processing all the vertices adjacent to each vertex (which is equivalent to processing all the edges in the graph), as in these functions.

Program 17.3 A client function that prints a graph

This implementation of the show function from the io class of Program 17.4 uses the graph ADT to print a table of the vertices adjacent to each graph vertex. The order in which the vertices appear depends upon the graph representation and the ADT implementation (see Figure 17.7).

```
template <class Graph>
void IO<Graph>::show(const Graph &G)
{
   for (int s = 0; s < G.V(); s++)
      {
      cout.width(2); cout << s << ":";
      typename Graph::adjIterator A(G, s);
      for (int t = A.beg(); !A.end(); t = A.nxt())
        { cout.width(2); cout << t << " "; }
      cout << endl;
    }
}</pre>
```

As discussed in Section 17.5, we often package related graphprocessing functions into a single class. Program 17.4 is an interface for such a class. It defines the show function of Program 17.3 and two functions for inserting into a graph edges taken from standard input (see Exercise 17.12 and Program 17.14 for implementations of these functions).

Generally, the graph-processing tasks that we consider in this book fall into one of three broad categories:

- Compute the value of some measure of the graph.
- Compute some subset of the edges of the graph.
- Answer queries about some property of the graph.

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

Figure 17.7 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, uappears in v's set and v appears in u's set.

Program 17.4 Graph-processing input/output interface

This class illustrates how we might package related graph-processing functions together in a single class. It defines functions for printing a graph (see Program 17.3); inserting edges defined by pairs of integers on standard input (see Exercise 17.12); and inserting edges defined by pairs of symbols on standard input (see Program 17.14).

```
template <class Graph>
class I0
  {
    public:
        static void show(const Graph &);
        static void scanEZ(Graph &);
        static void scan(Graph &);
    };
```

Our convention for addressing such tasks will be to build ADTs that are clients of the basic ADT in Program 17.1, but that, in turn, allow us to separate client programs that need to solve the problem from implementations. For example, Program 17.5 is an interface for a graph-connectivity ADT. We can write client programs that use this ADT to create objects that can provide the number of connected components in the graph and that can test whether or not any two vertices are in the same connected component. We describe implementations of this ADT and their performance characteristics in Section 18.5, and we develop similar ADTs throughout the book. Typically, such ADTs include a *preprocessing* public member function (usually the constructor), private data members that keep information learned during the preprocessing, and *query* public member functions that use this information to provide clients with information about the graph.

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 insert, then process them either by calling some ADT function that takes a graph as argument and returns some information about that graph, or by using objects of the kind just described to preprocess the graph so as to be able to efficiently answer queries about it. In either case, changing the graph by calling insert or remove necessitates reprocessing the graph. Dynamic problems,

Program 17.5 Connectivity interface

This ADT interface illustrates a typical paradigm that we use for implementing graph-processing algorithms. It allows a client to construct an object that processes a graph so that it can answer queries about the graph's connectivity. The count member function returns the number of connected components and the connect member function tests whether two given vertices are connected. Program 18.4 is an implementation of this interface.

```
template <class Graph>
class CC
  {
    private:
        // implementation-dependent code
    public:
        CC(const Graph &);
        int count();
        bool connect(int, int);
    };
```

where we want to 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 connectivity problem that we solved with union-find algorithms in Chapter I is an example 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 may require reprocessing the whole graph if a client uses *remove edge*. For most of the graph-processing problems that we consider, adding or deleting a few edges can dramatically change the nature of the graph and thus necessitate reprocessing it.

One of our most important challenges in graph processing is to have a clear understanding of performance characteristics of implementations and to make sure that client programs make appropriate use of them. As with the simpler problems that we considered in

Program 17.6 Example of a graph-processing client program

This program illustrates the use of the graph-processing ADTs described in this section, using the ADT conventions described in Section 4.5. It constructs a graph with V vertices, inserts edges taken from standard input, prints the resulting graph if it is small, and computes (and prints) the number of connected components. It assumes that Program 17.1, Program 17.4, and Program 17.5 (with implementations) are in the files GRAPH.cc, 10.cc, and CC.cc (respectively).

```
#include <iostream.h>
#include <stdlib.h>
#include "GRAPH.cc"
#include "IO.cc"
#include "CC.cc"
main(int argc, char *argv[])
{ int V = atoi(argv[1]);
    GRAPH G(V);
    IO<GRAPH>::scan(G);
    if (V < 20) IO<GRAPH>::show(G);
    cout << G.E() << " edges ";
    CC<GRAPH> Gcc(G);
    cout << Gcc.count() << " components" << endl;
}</pre>
```

Parts 1 through 4, our use of ADTs makes it possible to address such issues in a coherent manner.

Program 17.6 is an example of a graph-processing client program. It uses the basic ADT of Program 17.1, the input-output class of Program 17.4 to read the graph from standard input and print it to standard output, and the connectivity class of Program 17.5 to find its number of connected components. We use similar but more sophisticated clients to generate other types of graphs, to test algorithms, to learn other properties of graphs, and to use graphs to solve other problems. The basic scheme is amenable for use in any graph-processing application.

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.

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.

For example, we might consider a *vector of edges* representation as the basis for an ADT implementation (see Exercise 17.16). That direct representation is simple, but it does not allow us to perform efficiently the basic graph-processing operations that we shall be studying. As we will see, most graph-processing applications can be handled reasonably with one of two straightforward classical representations that are only slightly more complicated than the vector-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

- \triangleright 17.12 Implement the scanEZ function from Program 17.4: write a function that builds a graph by reading edges (pairs of integers between 0 and V 1) from standard input.
- ▷ 17.13 Write an ADT client that adds all the edges in a given vector to a given graph.
- ▷ 17.14 Write a function that calls edges and prints out all the edges in the graph, in the format used in this text (vertex numbers separated by a hyphen).

- 17.15 Develop an implementation for the connectivity ADT of Program 17.5, using a union-find algorithm (see Chapter 1).
- 17.16 Provide an implementation of the ADT functions in Program 17.1 that uses a vector of edges to represent the graph. Use a brute-force implementation of remove that removes an edge v-w by scanning the vector to find v-w or w-v and then exchanges the edge found with the final one in the vector. Use a similar scan to implement the iterator. *Note*: Reading Section 17.3 first might make this exercise easier.

17.3 Adjacency-Matrix Representation

An *adjacency-matrix* representation of a graph is a V-by-V matrix 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. Figure 17.8 depicts an example.

Program 17.7 is an implementation of the graph ADT interface that uses a direct representation of this matrix, built as a vector of vectors, as depicted in Figure 17.9. It is a two-dimensional existence table with the entry adj[v][w] set to true if there is an edge connecting v and w in the graph, and set to false otherwise. Note that maintaining this property in an undirected graph requires that each edge be represented by *two* entries: the edge v-w is represented by true values in *both* adj[v][w] and adj[w][v], as is the edge w-v.

The name DenseGRAPH in Program 17.7 emphasizes that the implementation is more suited for dense graphs than for sparse ones, and distinguishes it from other implementations. Clients may use typedef to make this type equivalent to GRAPH or use DenseGRAPH explicitly.

In the adjacency matrix that represents a graph G, row v is a vector that is an existence table whose ith entry is true if vertex i is adjacent to v (the edge v-i is in G). Thus, to provide clients with the capability to process the vertices adjacent to v, we need only provide code that scans through this vector to find true entries, as shown in Program 17.8. We need to be mindful that, with this implementation, processing all of the vertices adjacent to a given vertex requires (at least) time proportional to V, no matter how many such vertices exist.

As mentioned in Section 17.2, our interface requires that the number of vertices is known to the client when the graph is initialized. If desired, we could allow for inserting and deleting vertices (see Exercise 17.21). A key feature of the constructor in Program 17.7 is



Figure 17.8 Adjacency-matrix graph representation

This Boolean matrix is another representation of the graph depicted in Figure 17.1. It has a 1 (true) in row v and column w if there is an edge connecting vertex vand vertex w and a 0 (false) in row v and column w if there is no such edge. The matrix 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).

Program 17.7 Graph ADT implementation (adjacency matrix)

This class is a straightforward implementation of the interface in Program 17.1 that is based on representing the graph with a vector of boolean vectors (see Figure 17.9). Edges are inserted and removed in constant time. Duplicate edge insert requests are silently ignored, but clients can use edge to test whether an edge exists. Constructing the graph takes time proportional to V^2 .

```
class DenseGRAPH
{ int Vcnt, Ecnt; bool digraph;
  vector <vector <bool> > adj;
public:
  DenseGRAPH(int V, bool digraph = false) :
    adj(V), Vcnt(V), Ecnt(0), digraph(digraph)
    {
      for (int i = 0; i < V; i++)
        adj[i].assign(V, false);
    }
  int V() const { return Vcnt; }
  int E() const { return Ecnt; }
  bool directed() const { return digraph; }
  void insert(Edge e)
    { int v = e.v, w = e.w;
      if (adj[v][w] == false) Ecnt++;
      adj[v][w] = true;
      if (!digraph) adj[w][v] = true;
    }
  void remove(Edge e)
    { int v = e.v, w = e.w;
      if (adj[v][w] == true) Ecnt--;
      adj[v][w] = false;
      if (!digraph) adj[w][v] = false;
    }
  bool edge(int v, int w) const
    { return adj[v][w]; }
  class adjIterator;
  friend class adjIterator;
};
```

Program 17.8 Iterator for adjacency-matrix representation

This implementation of the iterator for Program 17.7 uses an index i to scan past false entries in row v of the adjacency matrix (adj[v]). A call to beg() followed by a sequence of calls to nxt() (checking that end() is false before each call) gives a sequence of the vertices adjacent to v in G in order of their vertex index.

```
class DenseGRAPH::adjIterator
{ const DenseGRAPH &G;
  int i, v;
public:
  adjIterator(const DenseGRAPH &G, int v) :
    G(G), v(v), i(-1) \{ \}
  int beg()
    { i = -1; return nxt(); }
  int nxt()
    ſ
      for (i++; i < G.V(); i++)</pre>
        if (G.adj[v][i] == true) return i;
      return -1;
    }
  bool end()
    { return i >= G.V(); }
};
```

that it initializes the graph by setting the matrix entries all to false. We need to be mindful that this operation takes time proportional to V^2 , no matter how many edges are in the graph. Error checks for insufficient memory are not included in Program 17.7 for brevity—it is prudent programming practice to add them before using this code (see Exercise 17.24).

To add an edge, we set the indicated matrix entries to true (one for digraphs, two for undirected graphs). This representation does not allow parallel edges: If an edge is to be inserted for which the matrix 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 insert. This



Figure 17.9 Adjacency matrix data structure

This figure depicts the C++ representation of the graph in Figure 17.1, as an vector of vectors.

representation does allow self-loops: An edge v-v is represented by a nonzero entry in a[v][v].

To remove an edge, we set the indicated matrix entries to false. If a nonexistent edge (one for which the matrix entries are already false) 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]. Thus, we could save space by storing only one-half of this symmetric matrix (see Exercise 17.22). Another way to save a significant amount of space is to use a matrix of bits (assuming that vector<bool> does not do so). 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 17.23). These implementations have the slight complication that we need to add an ADT operation to test for the existence of an edge (see Exercise 17.20). (We do not use such an operation in our implementations because we can test for the existence of 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 hold any information whatever, not just bools. Whatever data type that we use for the matrix elements, we need to include an indication whether the indicated edge is present or absent. In Chapters 20 and 21, we explore such representations.

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 vector of vectors in C++ is but one possible representation of any given graph as an adjacency matrix, because another program might assign different vertex names to the indices we use to specify rows and columns. Two matrices 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 matrices 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 17.26).

Program 17.3, which we considered in Section 17.2, prints out a table with the vertices adjacent to each vertex. When used with the implementation in Program 17.7, it prints the vertices in order of their vertex index, as in Figure 17.7. Notice, though, that it is not part of the definition of adjIterator that it visits vertices in index order, so developing an ADT client that prints out the adjacency-matrix representation of a graph is not a trivial task (see Exercise 17.18). The output produced by these programs are themselves graph representations that clearly illustrate a basic performance tradeoff. To print out the matrix, we need room on the page 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 matrix. 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: We need at least V^2 bits of storage and V^2 steps just to construct the representation. 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 matrix 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 in constant time. 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

- ▷ 17.17 Give the adjacency-matrix representations of the three graphs depicted in Figure 17.2.
- 0 17.18 Give an implementation of show for the representation-independent io package of Program 17.4 that prints out a two-dimensional matrix of 0s and 1s like the one illustrated in Figure 17.8. Note: You cannot depend upon the iterator producing vertices in order of their indices.

17.19 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 are the adjacency matrices of these two graphs related?

- ▷ 17.20 Add a function edge 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.
- 17.21 Add functions to the graph ADT that allow clients to insert and delete vertices, and provide implementations for the adjacency-matrix representation.
- ▷ 17.22 Modify Program 17.7, augmented as described in Exercise 17.20, to cut its space requirements about in half by not including array entries a [v] [w] for w greater than v.

17.23 Modify Program 17.7, augmented as described in Exercise 17.20, to ensure that, if your computer has *B* bits per word, a graph with *V* vertices is represented in about V^2/B words (as opposed to V^2). Do empirical tests to assess the effect of packing bits into words on the time required for the ADT operations.

17.24 Describe what happens if there is insufficient memory available to represent the matrix when the constructor in Program **17.7** is invoked, and suggest appropriate modifications to the code to handle this situation.

17.25 Develop a version of Program 17.7 that uses a single vector with V^2 entries.

 \circ 17.26 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? Write a client ADT function 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 a vector 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 17.9 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). For undirected graphs, 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 adjacent to vertex v?" Program 17.10 implements the iterator that answers this question for clients, in time proportional to the number of such vertices.

The implementation in Programs 17.9 and 17.10 is a low-level one. An alternative is to use the STL list to implement each linked list (see Exercise 17.30). The disadvantage of doing so is that STL list implementations need to support many more operations than we need and therefore typically carry extra overhead that might affect the performance of all of our algorithms (see Exercise 17.31). Indeed, all of our graph algorithms use the Graph ADT interface, so this implementation is an appropriate place to encapuslate all the low-level operations and concentrate on efficiency without affecting our other code. Another advantage of using the linked-list representation is that it provides a concrete basis for understanding the performance characteristics of our implementations.

But an important factor to consider is that the linked-list-based implementation in Programs 17.9 and 17.10 is incomplete, because it lacks a destructor and a copy constructor. For many applications, this defect could lead to unexpected results or severe performance prob-



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 vth 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.