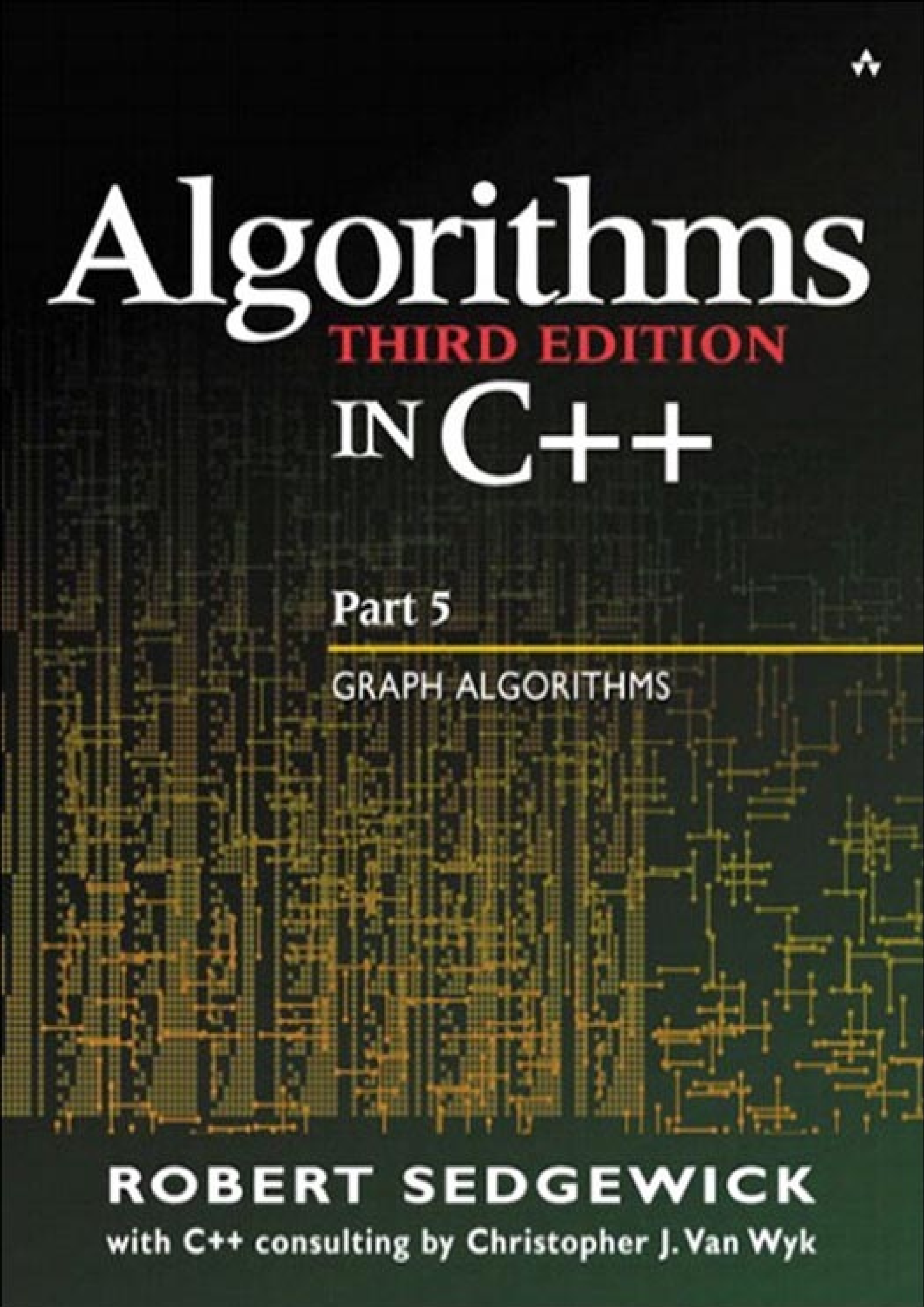
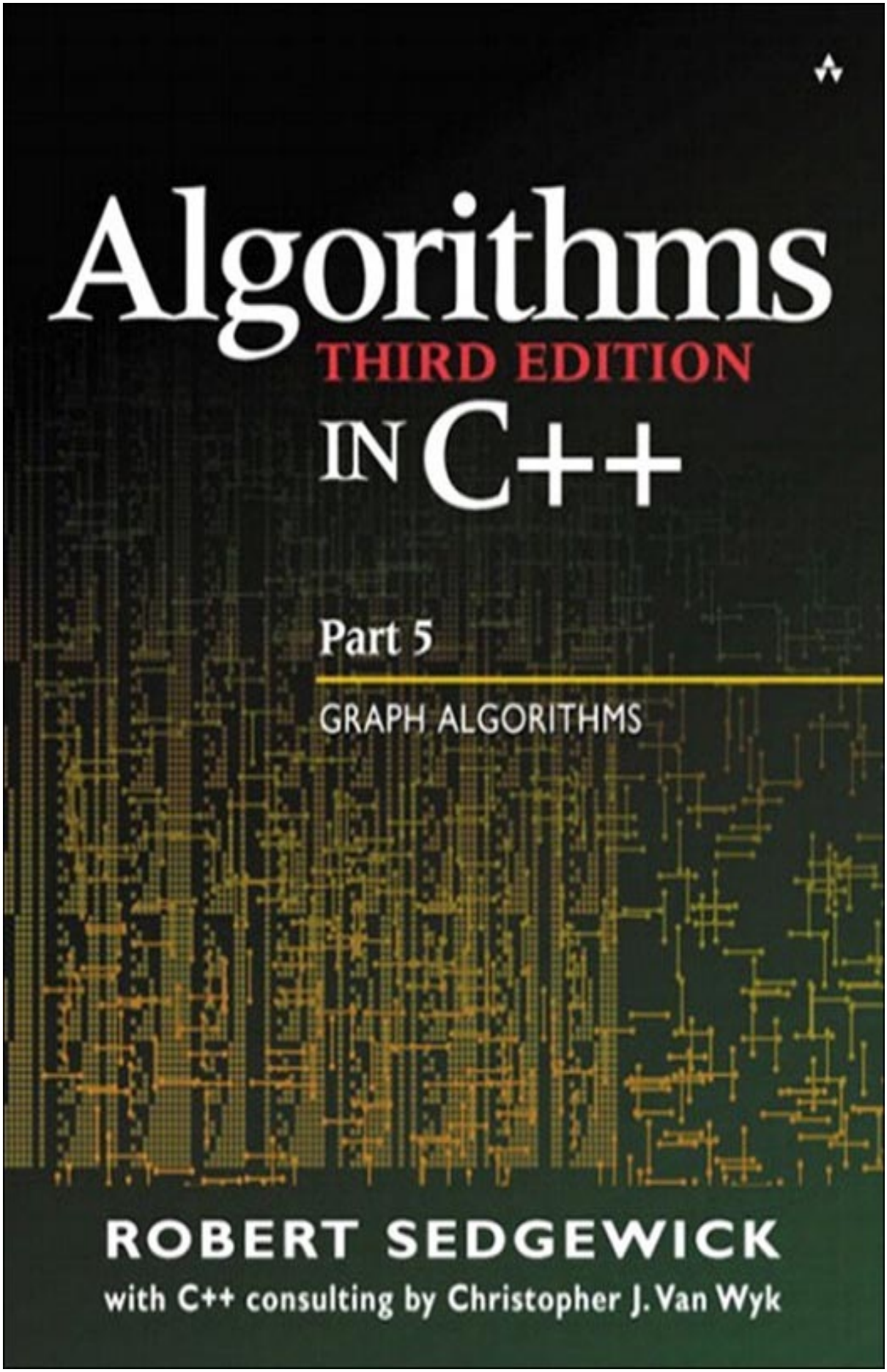

Algorithms
THIRD EDITION
IN C++
Part 5
GRAPH ALGORITHMS
ROBERT SEDGEWICK
with C++ consulting by Christopher J. Van Wyk


Algorithms
THIRD EDITION
IN C++
Part 5
GRAPH ALGORITHMS
ROBERT SEDGEWICK
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

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
1. C++ (Computer program language) 2. Computer algorithms.
I. Title.
QA76.73.C15S38 2002
005.13’3—dc20 92-901
CIP

ISBN 0-201-36118-3

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

Preface

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

Algorithms

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

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

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

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 low-level 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

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

Notes on Exercises

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

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

- **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

PART FIVE
Graph Algorithms

CHAPTER SEVENTEEN
Graph Properties and Types

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

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

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

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

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

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

Hypertexts When we browse the Web, we encounter documents that contain references (links) to other documents, and we move from document to document by clicking on the links. The entire web is a graph, where the items are documents and the connections are links. Graph-processing algorithms are essential components of the search engines that help us locate information on the web.

Circuits An electric circuit comprises elements such as transistors, resistors, and

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

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

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

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

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

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

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