NETWORK FLOWS

Theory, Algorithms, and Applications

RAVINDRA K. AHUJA
Department of Industrial & Management Engineering
Indian Institute of Technology, Kanpur

THOMAS L. MAGNANTI
Sloan School of Management
Massachusetts Institute of Technology, Cambridge

JAMES B. ORLIN
Sloan School of Management
Massachusetts Institute of Technology, Cambridge

PRENTICE HALL, Upper Saddle River, New Jersey 07458

Library of Congress Cataloging-in-Publication Data

Ahuja, Ravindra K. (date)
Network flows : theory, algorithms, and applications / Ravindra K.
Ahuja, Thomas L. Magnanti, James B. Orlin.
p. cm.
Includes bibliographical references and index.
ISBN 0-13-617549-X
1. Network analysis (Planning) 2. Mathematical optimization.
I. Magnanti, Thomas L. II. Orlin, James B., (date). III. Title.
T57.85.A37 1993
658.4'032—dc20 92-26702
CIP

Acquisitions editor: Pete Janzow
Production editor: Merrill Peterson
Cover designer: Design Source
Prepress buyer: Linda Behrens
Manufacturing buyer: David Dickey
Editorial assistant: Phyllis Morgan

The author and publisher of this book have used their best efforts in preparing this book. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness. The author and publisher make no warranty of any kind, expressed or implied, with regard to these programs or the documentation contained in this book. The author and publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of the furnishing, performance, or use of these programs.

© 1993 by Prentice-Hall, Inc.
Upper Saddle River, New Jersey 07458

All rights reserved. No part of this book may be
reproduced, in any form or by any means,
without permission in writing from the publisher.

Printed in the United States of America

16 17 18 19

ISBN 0-13-617549-X

Prentice-Hall International (UK) Limited, *London*
Prentice-Hall of Australia Pty. Limited, *Sydney*
Prentice-Hall Canada Inc., *Toronto*
Prentice-Hall Hispanoamericana, S.A., *Mexico*
Prentice-Hall of India Private Limited, *New Delhi*
Prentice-Hall of Japan, Inc., *Tokyo*
Editora Prentice-Hall do Brasil, Ltda., *Rio de Janeiro*

Ravi dedicates this book to his spiritual master,
Revered Sri Ranaji Saheb.

Tom dedicates this book to his favorite network,
Beverly and Randy.

Jim dedicates this book to Donna,
who inspired him in so many ways.

Collectively, we offer this book as a tribute
to Lester Ford and Ray Fulkerson, whose pioneering research
and seminal text in network flows have been an enduring
inspiration to us and to a generation
of researchers and practitioners.

CONTENTS

PREFACE, xi

1 INTRODUCTION, 1

1.1 Introduction, *1*
1.2 Network Flow Problems, *4*
1.3 Applications, *9*
1.4 Summary, *18*
Reference Notes, *19*
Exercises, *20*

2 PATHS, TREES, AND CYCLES, 23

2.1 Introduction, *23*
2.2 Notation and Definitions, *24*
2.3 Network Representations, *31*
2.4 Network Transformations, *38*
2.5 Summary, *46*
Reference Notes, *47*
Exercises, *47*

3 ALGORITHM DESIGN AND ANALYSIS, 53

3.1 Introduction, *53*
3.2 Complexity Analysis, *56*
3.3 Developing Polynomial-Time Algorithms, *66*
3.4 Search Algorithms, *73*
3.5 Flow Decomposition Algorithms, *79*
3.6 Summary, *84*
Reference Notes, *85*
Exercises, *86*

4 SHORTEST PATHS: LABEL-SETTING ALGORITHMS, 93

4.1 Introduction, *93*
4.2 Applications, *97*
4.3 Tree of Shortest Paths, *106*
4.4 Shortest Path Problems in Acyclic Networks, *107*
4.5 Dijkstra's Algorithm, *108*
4.6 Dial's Implementation, *113*
4.7 Heap Implementations, *115*
4.8 Radix Heap Implementation, *116*

4.9 Summary, *121*
Reference Notes, *122*
Exercises, *124*

5 *SHORTEST PATHS: LABEL-CORRECTING ALGORITHMS, 133*

5.1 Introduction, *133*
5.2 Optimality Conditions, *135*
5.3 Generic Label-Correcting Algorithms, *136*
5.4 Special Implementations of the Modified Label-Correcting Algorithm, *141*
5.5 Detecting Negative Cycles, *143*
5.6 All-Pairs Shortest Path Problem, *144*
5.7 Minimum Cost-to-Time Ratio Cycle Problem, *150*
5.8 Summary, *154*
Reference Notes, *156*
Exercises, *157*

6 *MAXIMUM FLOWS: BASIC IDEAS, 166*

6.1 Introduction, *166*
6.2 Applications, *169*
6.3 Flows and Cuts, *177*
6.4 Generic Augmenting Path Algorithm, *180*
6.5 Labeling Algorithm and the Max-Flow Min-Cut Theorem, *184*
6.6 Combinatorial Implications of the Max-Flow Min-Cut Theorem, *188*
6.7 Flows with Lower Bounds, *191*
6.8 Summary, *196*
Reference Notes, *197*
Exercises, *198*

7 *MAXIMUM FLOWS: POLYNOMIAL ALGORITHMS, 207*

7.1 Introduction, *207*
7.2 Distance Labels, *209*
7.3 Capacity Scaling Algorithm, *210*
7.4 Shortest Augmenting Path Algorithm, *213*
7.5 Distance Labels and Layered Networks, *221*
7.6 Generic Preflow-Push Algorithm, *223*
7.7 FIFO Preflow-Push Algorithm, *231*
7.8 Highest-Label Preflow-Push Algorithm, *233*
7.9 Excess Scaling Algorithm, *237*
7.10 Summary, *241*
Reference Notes, *241*
Exercises, *243*

8 *MAXIMUM FLOWS: ADDITIONAL TOPICS, 250*

8.1 Introduction, *250*
8.2 Flows in Unit Capacity Networks, *252*
8.3 Flows in Bipartite Networks, *255*
8.4 Flows in Planar Undirected Networks, *260*
8.5 Dynamic Tree Implementations, *265*

8.6 Network Connectivity, *273*
8.7 All-Pairs Minimum Value Cut Problem, *277*
8.8 Summary, *285*
Reference Notes, *287*
Exercises, *288*

9 *MINIMUM COST FLOWS: BASIC ALGORITHMS,* *294*

9.1 Introduction, *294*
9.2 Applications, *298*
9.3 Optimality Conditions, *306*
9.4 Minimum Cost Flow Duality, *310*
9.5 Relating Optimal Flows to Optimal Node Potentials, *315*
9.6 Cycle-Canceling Algorithm and the Integrality Property, *317*
9.7 Successive Shortest Path Algorithm, *320*
9.8 Primal–Dual Algorithm, *324*
9.9 Out-of-Kilter Algorithm, *326*
9.10 Relaxation Algorithm, *332*
9.11 Sensitivity Analysis, *337*
9.12 Summary, *339*
Reference Notes, *341*
Exercises, *344*

10 *MINIMUM COST FLOWS: POLYNOMIAL ALGORITHMS,* *357*

10.1 Introduction, *357*
10.2 Capacity Scaling Algorithm, *360*
10.3 Cost Scaling Algorithm, *362*
10.4 Double Scaling Algorithm, *373*
10.5 Minimum Mean Cycle-Canceling Algorithm, *376*
10.6 Repeated Capacity Scaling Algorithm, *382*
10.7 Enhanced Capacity Scaling Algorithm, *387*
10.8 Summary, *395*
Reference Notes, *396*
Exercises, *397*

11 *MINIMUM COST FLOWS: NETWORK SIMPLEX ALGORITHMS,* *402*

11.1 Introduction, *402*
11.2 Cycle Free and Spanning Tree Solutions, *405*
11.3 Maintaining a Spanning Tree Structure, *409*
11.4 Computing Node Potentials and Flows, *411*
11.5 Network Simplex Algorithm, *415*
11.6 Strongly Feasible Spanning Trees, *421*
11.7 Network Simplex Algorithm for the Shortest Path Problem, *425*
11.8 Network Simplex Algorithm for the Maximum Flow Problem, *430*
11.9 Related Network Simplex Algorithms, *433*
11.10 Sensitivity Analysis, *439*
11.11 Relationship to Simplex Method, *441*
11.12 Unimodularity Property, *447*
11.13 Summary, *450*
Reference Notes, *451*
Exercises, *453*

12 ***ASSIGNMENTS AND MATCHINGS, 461***

12.1 Introduction, *461*
12.2 Applications, *463*
12.3 Bipartite Cardinality Matching Problem, *469*
12.4 Bipartite Weighted Matching Problem, *470*
12.5 Stable Marriage Problem, *473*
12.6 Nonbipartite Cardinality Matching Problem, *475*
12.7 Matchings and Paths, *494*
12.8 Summary, *498*
Reference Notes, *499*
Exercises, *501*

13 ***MINIMUM SPANNING TREES, 510***

13.1 Introduction, *510*
13.2 Applications, *512*
13.3 Optimality Conditions, *516*
13.4 Kruskal's Algorithm, *520*
13.5 Prim's Algorithm, *523*
13.6 Sollin's Algorithm, *526*
13.7 Minimum Spanning Trees and Matroids, *528*
13.8 Minimum Spanning Trees and Linear Programming, *530*
13.9 Summary, *533*
Reference Notes, *535*
Exercises, *536*

14 ***CONVEX COST FLOWS, 543***

14.1 Introduction, *543*
14.2 Applications, *546*
14.3 Transformation to a Minimum Cost Flow Problem, *551*
14.4 Pseudopolynomial-Time Algorithms, *554*
14.5 Polynomial-Time Algorithm, *556*
14.6 Summary, *560*
Reference Notes, *561*
Exercises, *562*

15 ***GENERALIZED FLOWS, 566***

15.1 Introduction, *566*
15.2 Applications, *568*
15.3 Augmented Forest Structures, *572*
15.4 Determining Potentials and Flows for an Augmented Forest Structure, *577*
15.5 Good Augmented Forests and Linear Programming Bases, *582*
15.6 Generalized Network Simplex Algorithm, *583*
15.7 Summary, *591*
Reference Notes, *591*
Exercises, *593*

16 *LAGRANGIAN RELAXATION AND NETWORK OPTIMIZATION,* **598**

16.1 Introduction, *598*
16.2 Problem Relaxations and Branch and Bound, *602*
16.3 Lagrangian Relaxation Technique, *605*
16.4 Lagrangian Relaxation and Linear Programming, *615*
16.5 Applications of Lagrangian Relaxation, *620*
16.6 Summary, *635*
Reference Notes, *637*
Exercises, *638*

17 *MULTICOMMODITY FLOWS,* **649**

17.1 Introduction, *649*
17.2 Applications, *653*
17.3 Optimality Conditions, *657*
17.4 Lagrangian Relaxation, *660*
17.5 Column Generation Approach, *665*
17.6 Dantzig–Wolfe Decomposition, *671*
17.7 Resource-Directive Decomposition, *674*
17.8 Basis Partitioning, *678*
17.9 Summary, *682*
Reference Notes, *684*
Exercises, *686*

18 *COMPUTATIONAL TESTING OF ALGORITHMS,* **695**

18.1 Introduction, *695*
18.2 Representative Operation Counts, *698*
18.3 Application to Network Simplex Algorithm, *702*
18.4 Summary, *713*
Reference Notes, *713*
Exercises, *715*

19 *ADDITIONAL APPLICATIONS,* **717**

19.1 Introduction, *717*
19.2 Maximum Weight Closure of a Graph, *719*
19.3 Data Scaling, *725*
19.4 Science Applications, *728*
19.5 Project Management, *732*
19.6 Dynamic Flows, *737*
19.7 Arc Routing Problems, *740*
19.8 Facility Layout and Location, *744*
19.9 Production and Inventory Planning, *748*
19.10 Summary, *755*
Reference Notes, *759*
Exercises, *760*

APPENDIX A ***DATA STRUCTURES,*** ***765***

A.1 Introduction, *765*
A.2 Elementary Data Structures, *766*
A.3 *d*-Heaps, *773*
A.4 Fibonacci Heaps, *779*
Reference Notes, *787*

APPENDIX B ***$\mathcal{NP}$-COMPLETENESS,*** ***788***

B.1 Introduction, *788*
B.2 Problem Reductions and Transformations, *790*
B.3 Problem Classes $\mathcal{P}$, $\mathcal{NP}$, $\mathcal{NP}$-Complete, and $\mathcal{NP}$-Hard, *792*
B.4 Proving $\mathcal{NP}$-Completeness Results, *796*
B.5 Concluding Remarks, *800*
Reference Notes, *801*

APPENDIX C ***LINEAR PROGRAMMING,*** ***802***

C.1 Introduction, *802*
C.2 Graphical Solution Procedure, *804*
C.3 Basic Feasible Solutions, *805*
C.4 Simplex Method, *810*
C.5 Bounded Variable Simplex Method, *814*
C.6 Linear Programming Duality, *816*
Reference Notes, *820*

REFERENCES, ***821***

INDEX, ***840***

1

INTRODUCTION

Begin at the beginning . . . and go on till you come to the end: then stop.
—Lewis Carroll

Chapter Outline

1.1 Introduction
1.2 Network Flow Problems
1.3 Applications
1.4 Summary

1.1 INTRODUCTION

Everywhere we look in our daily lives, networks are apparent. Electrical and power networks bring lighting and entertainment into our homes. Telephone networks permit us to communicate with each other almost effortlessly within our local communities and across regional and international borders. National highway systems, rail networks, and airline service networks provide us with the means to cross great geographical distances to accomplish our work, to see our loved ones, and to visit new places and enjoy new experiences. Manufacturing and distribution networks give us access to life's essential foodstock and to consumer products. And computer networks, such as airline reservation systems, have changed the way we share information and conduct our business and personal lives.

In all of these problem domains, and in many more, we wish to move some entity (electricity, a consumer product, a person or a vehicle, a message) from one point to another in an underlying network, and to do so as efficiently as possible, both to provide good service to the users of the network and to use the underlying (and typically expensive) transmission facilities effectively. In the most general sense, this objective is what this book is all about. We want to learn how to model application settings as mathematical objects known as network flow problems and to study various ways (algorithms) to solve the resulting models.

Network flows is a problem domain that lies at the cusp between several fields of inquiry, including applied mathematics, computer science, engineering, management, and operations research. The field has a rich and long tradition, tracing its roots back to the work of Gustav Kirchhof and other early pioneers of electrical engineering and mechanics who first systematically analyzed electrical circuits. This early work set the foundations of many of the key ideas of network flow theory and established networks (graphs) as useful mathematical objects for representing many

physical systems. Much of this early work was descriptive in nature, answering such questions as: If we apply a set of voltages to a given network, what will be the resulting current flow? The set of questions that we address in this book are a bit different: If we have alternative ways to use a network (i.e., send flow), which alternative will be most cost-effective? Our intellectual heritage for answering such questions is much more recent and can be traced to the late 1940s and early 1950s when the research and practitioner communities simultaneously developed optimization as an independent field of inquiry and launched the computer revolution, leading to the powerful instruments we know today for performing scientific and managerial computations.

For the most part, in this book we wish to address the following basic questions:

1. *Shortest path problem.* What is the best way to traverse a network to get from one point to another as cheaply as possible?
2. *Maximum flow problem.* If a network has capacities on arc flows, how can we send as much flow as possible between two points in the network while honoring the arc flow capacities?
3. *Minimum cost flow problem.* If we incur a cost per unit flow on a network with arc capacities and we need to send units of a good that reside at one or more points in the network to one or more other points, how can we send the material at minimum possible cost?

In the sense of traditional applied and pure mathematics, each of these problems is trivial to solve. It is not very difficult (but not at all obvious for the later two problems) to see that we need only consider a finite number of alternatives for each problem. So a traditional mathematician might say that the problems are well solved: Simply enumerate the set of possible solutions and choose the one that is best. Unfortunately, this approach is far from pragmatic, since the number of possible alternatives can be very large—more than the number of atoms in the universe for many practical problems! So instead, we would like to devise algorithms that are in a sense "good," that is, whose computation time is small, or at least reasonable, for problems met in practice. One way to ensure this objective is to devise algorithms whose running time is guaranteed not to grow very fast as the underlying network becomes larger (the computer science, operations research, and applied mathematics communities refer to the development of algorithms with such performance guarantees as *worst-case analysis*). Developing algorithms that are good in this sense is another major theme throughout this book, and our development builds heavily on the theory of computational complexity that began to develop within computer science, applied mathematics, and operations research circles in the 1970s, and has flourished ever since.

The field of computational complexity theory combines both craftsmanship and theory; it builds on a confluence of mathematical insight, creative algorithm design, and the careful, and often very clever use of data structures to devise solution methods that are provably good in the sense that we have just mentioned. In the field of network flows, researchers devised the first, seminal contributions of this nature in the 1950s before the field of computational complexity theory even existed as a separate discipline as we know it today. And throughout the last three decades,

researchers have made a steady stream of innovations that have resulted in new solution methods and in improvements to known methods. In the past few years, however, researchers have made contributions to the design and analysis of network flow algorithms with improved worst-case performance guarantees at an explosive, almost dizzying pace; moreover, these contributions were very surprising: Throughout the 1950s, 1960s, and 1970s, network flows had evolved into a rather mature field, so much so that most of the research and practitioner communities believed that the core models that we study in this book were so very well understood that further innovations would be hard to come by and would be few and far between. As it turns out, nothing could have been further from the truth.

Our presentation is intended to reflect these new developments; accordingly, we place a heavy emphasis on designing and analyzing good algorithms for solving the core optimization models that arise in the context of network flows. Our intention is to bring together and synthesize the many new contributions concerning efficient network flow algorithms with traditional material that has evolved over the past four decades. We have attempted to distill and highlight some of the essential core ideas (e.g., scaling and potential function arguments) that underlie many of the recent innovations and in doing so to give a unified account of the many algorithms that are now available. We hope that this treatment will provide our readers not only with an accessible entrée to these exciting new developments, but also with an understanding of the most recent and advanced contributions from the literature. Although we are bringing together ideas and methodologies from applied mathematics, computer science, and operations research, our approach has a decidedly computer science orientation as applied to certain types of models that have traditionally arisen in the context of managing a variety of operational systems (the foodstuff of operations research).

We feel that a full understanding of network flow algorithms and a full appreciation for their use requires more than an in-depth knowledge of good algorithms for core models. Consequently, even though this topic is our central thrust, we also devote considerable attention to describing applications of network flow problems. Indeed, we feel that our discussion of applications throughout the text, in the exercises, and in a concluding chapter is one of the major distinguishing features of our coverage.

We have not adopted a linear programming perspective throughout the book, however, because we feel there is much to be gained from a more direct approach, and because we would like the material we cover to be readily accessible to readers who are not optimization specialists. Moreover, we feel that an understanding of network flow problems from first principles provides a useful concrete setting from which to draw considerable insight about more general linear programs.

Similarly, since several important variations of the basic network flow problems are important in practice, or in placing network flows in the broader context of the field of combinatorial optimization, we have also included several chapters on additional topics: assignments and matchings, minimum spanning trees, models with convex (instead of linear) costs, networks with losses and gains, and multicommodity flows. In each of these chapters we have not attempted to be comprehensive, but rather, have tried to provide an introduction to the essential ideas of the topics.

The Lagrangian relaxation chapter permits us to show how the core network

models arise in broader problem contexts and how the algorithms that we have developed for the core models can be used in conjunction with other methods to solve more complex problems that arise frequently in practice. In particular, this discussion permits us to introduce and describe the basic ideas of decomposition methods for several important network optimization models—constrained shortest paths, the traveling salesman problem, vehicle routing problem, multicommodity flows, and network design.

Since the proof of the pudding is in the eating, we have also included a chapter on some aspects of computational testing of algorithms. We devote much of our discussion to devising the best possible algorithms for solving network flow problems, in the theoretical sense of computational complexity theory. Although the theoretical model of computation that we are using has proven to be a valuable guide for modeling and predicting the performance of algorithms in practice, it is not a perfect model, and therefore algorithms that are not theoretically superior often perform best in practice. Although empirical testing of algorithms has traditionally been a valuable means for investigating algorithmic ideas, the applied mathematics, computer science, and operations research communities have not yet reached a consensus on how to measure algorithmic performance empirically. So in this chapter we not only report on computational experience with an algorithm we have presented, but also offer some thoughts on how to measure computational performance and compare algorithms.

1.2 NETWORK FLOW PROBLEMS

In this section we introduce the network flow models we study in this book, and in the next section we present several applications that illustrate the practical importance of these models. In both the text and exercises throughout the remaining chapters, we introduce many other applications. In particular, Chapter 19 contains a more comprehensive summary of applications with illustrations drawn from several specialties in applied mathematics, engineering, logistics, manufacturing, and the physical sciences.

Minimum Cost Flow Problem

The minimum cost flow model is the most fundamental of all network flow problems. Indeed, we devote most of this book to the minimum cost flow problem, special cases of it, and several of its generalizations. The problem is easy to state: We wish to determine a least cost shipment of a commodity through a network in order to satisfy demands at certain nodes from available supplies at other nodes. This model has a number of familiar applications: the distribution of a product from manufacturing plants to warehouses, or from warehouses to retailers; the flow of raw material and intermediate goods through the various machining stations in a production line; the routing of automobiles through an urban street network; and the routing of calls through the telephone system. As we will see later in this chapter and in Chapters 9 and 19, the minimum cost flow model also has many less transparent applications.

In this section we present a mathematical programming formulation of the minimum cost flow problem and then describe several of its specializations and

variants as well as other basic models that we consider in later chapters. We assume our readers are familiar with the basic notation and definitions of graph theory; those readers without this background might consult Section 2.2 for a brief account of this material.

Let $G = (N, A)$ be a directed network defined by a set N of n *nodes* and a set A of m *directed arcs*. Each arc $(i, j) \in A$ has an associated *cost* c_{ij} that denotes the cost per unit flow on that arc. We assume that the flow cost varies linearly with the amount of flow. We also associate with each arc $(i, j) \in A$ a *capacity* u_{ij} that denotes the maximum amount that can flow on the arc and a *lower bound* l_{ij} that denotes the minimum amount that must flow on the arc. We associate with each node $i \in N$ an integer number $b(i)$ representing its supply/demand. If $b(i) > 0$, node i is a *supply node*; if $b(i) < 0$, node i is a *demand node* with a demand of $-b(i)$; and if $b(i) = 0$, node i is a *transshipment node*. The decision variables in the minimum cost flow problem are arc flows and we represent the flow on an arc $(i, j) \in A$ by x_{ij}. The minimum cost flow problem is an optimization model formulated as follows:

$$\text{Minimize} \sum_{(i,j)\in A} c_{ij}x_{ij} \tag{1.1a}$$

subject to

$$\sum_{\{j:(i,j)\in A\}} x_{ij} - \sum_{\{j:(j,i)\in A\}} x_{ji} = b(i) \qquad \text{for all } i \in N, \tag{1.1b}$$

$$l_{ij} \le x_{ij} \le u_{ij} \qquad \text{for all } (i,j) \in A, \tag{1.1c}$$

where $\sum_{i=1}^{n} b(i) = 0$. In matrix form, we represent the minimum cost flow problem as follows:

$$\text{Minimize } cx \tag{1.2a}$$

subject to

$$\mathcal{N}x = b, \tag{1.2b}$$

$$l \le x \le u. \tag{1.2c}$$

In this formulation, $\mathcal{N}$ is an $n \times m$ matrix, called the *node–arc incidence matrix* of the minimum cost flow problem. Each column $\mathcal{N}_{ij}$ in the matrix corresponds to the variable x_{ij}. The column $\mathcal{N}_{ij}$ has a $+1$ in the ith row, a -1 in the jth row; the rest of its entries are zero.

We refer to the constraints in (1.1b) as *mass balance constraints*. The first term in this constraint for a node represents the total *outflow* of the node (i.e., the flow emanating from the node) and the second term represents the total *inflow* of the node (i.e., the flow entering the node). The mass balance constraint states that the outflow minus inflow must equal the supply/demand of the node. If the node is a supply node, its outflow exceeds its inflow; if the node is a demand node, its inflow exceeds its outflow; and if the node is a transshipment node, its outflow equals its inflow. The flow must also satisfy the lower bound and capacity constraints (1.1c), which we refer to as *flow bound constraints*. The flow bounds typically model physical capacities or restrictions imposed on the flows' operating ranges. In most applications, the lower bounds on arc flows are zero; therefore, if we do not state lower bounds for any problem, we assume that they have value zero.

In most parts of the book we assume that the data are integral (i.e., all arc capacities, arc costs, and supplies/demands of nodes are integral). We refer to this assumption as the *integrality assumption*. The integrality assumption is not restrictive for most applications because we can always transform rational data to integer data by multiplying them by a suitably large number. Moreover, we necessarily need to convert irrational numbers to rational numbers to represent them on a computer.

The following special versions of the minimum cost flow problem play a central role in the theory and applications of network flows.

Shortest path problem. The shortest path problem is perhaps the simplest of all network flow problems. For this problem we wish to find a path of minimum cost (or length) from a specified *source node* s to another specified *sink node* t, assuming that each arc $(i, j) \in A$ has an associated cost (or length) c_{ij}. Some of the simplest applications of the shortest path problem are to determine a path between two specified nodes of a network that has minimum length, or a path that takes least time to traverse, or a path that has the maximum reliability. As we will see in our later discussions, this basic model has applications in many different problem domains, such as equipment replacement, project scheduling, cash flow management, message routing in communication systems, and traffic flow through congested cities. If we set $b(s) = 1$, $b(t) = -1$, and $b(i) = 0$ for all other nodes in the minimum cost flow problem, the solution to the problem will send 1 unit of flow from node s to node t along the shortest path. The shortest path problem also models situations in which we wish to send flow from a single-source node to a single-sink node in an uncapacitated network. That is, if we wish to send v units of flow from node s to node t and the capacity of each arc of the network is at least v, we would send the flow along a shortest path from node s to node t. If we want to determine shortest paths from the source node s to every other node in the network, then in the minimum cost flow problem we set $b(s) = (n - 1)$ and $b(i) = -1$ for all other nodes. [We can set each arc capacity u_{ij} to any number larger than $(n - 1)$.] The minimum cost flow solution would then send unit flow from node s to every other node i along a shortest path.

Maximum flow problem. The maximum flow problem is in a sense a complementary model to the shortest path problem. The shortest path problem models situations in which flow incurs a cost but is not restricted by any capacities; in contrast, in the maximum flow problem flow incurs no costs but is restricted by flow bounds. The maximum flow problem seeks a feasible solution that sends the maximum amount of flow from a specified source node s to another specified sink node t. If we interpret u_{ij} as the maximum flow rate of arc (i, j), the maximum flow problem identifies the maximum steady-state flow that the network can send from node s to node t per unit time. Examples of the maximum flow problem include determining the maximum steady-state flow of (1) petroleum products in a pipeline network, (2) cars in a road network, (3) messages in a telecommunication network, and (4) electricity in an electrical network. We can formulate this problem as a minimum cost flow problem in the following manner. We set $b(i) = 0$ for all $i \in N$, $c_{ij} = 0$ for all $(i, j) \in A$, and introduce an additional arc (t, s) with cost $c_{ts} = -1$ and flow bound $u_{ts} = \infty$. Then the minimum cost flow solution maximizes the flow on arc (t, s); but

since any flow on arc (t, s) must travel from node s to node t through the arcs in A [since each $b(i) = 0$], the solution to the minimum cost flow problem will maximize the flow from node s to node t in the original network.

Assignment problem. The data of the assignment problem consist of two equally sized sets N_1 and N_2 (i.e., $|N_1| = |N_2|$), a collection of pairs $A \subseteq N_1 \times N_2$ representing possible assignments, and a cost c_{ij} associated with each element $(i, j) \in A$. In the assignment problem we wish to pair, at minimum possible cost, each object in N_1 with exactly one object in N_2. Examples of the assignment problem include assigning people to projects, jobs to machines, tenants to apartments, swimmers to events in a swimming meet, and medical school graduates to available internships. The assignment problem is a minimum cost flow problem in a network $G = (N_1 \cup N_2, A)$ with $b(i) = 1$ for all $i \in N_1$, $b(i) = -1$ for all $i \in N_2$, and $u_{ij} = 1$ for all $(i, j) \in A$.

Transportation problem. The transportation problem is a special case of the minimum cost flow problem with the property that the node set N is partitioned into two subsets N_1 and N_2 (of possibly unequal cardinality) so that (1) each node in N_1 is a supply node, (2) each node N_2 is a demand node, and (3) for each arc (i,j) in A, $i \in N_1$ and $j \in N_2$. The classical example of this problem is the distribution of goods from warehouses to customers. In this context the nodes in N_1 represent the warehouses, the nodes in N_2 represent customers (or, more typically, customer zones), and an arc (i, j) in A represents a distribution channel from warehouse i to customer j.

Circulation problem. The circulation problem is a minimum cost flow problem with only transshipment nodes; that is, $b(i) = 0$ for all $i \in N$. In this instance we wish to find a feasible flow that honors the lower and upper bounds l_{ij} and u_{ij} imposed on the arc flows x_{ij}. Since we never introduce any exogenous flow into the network or extract any flow from it, all the flow circulates around the network. We wish to find the circulation that has the minimum cost. The design of a routing schedule of a commercial airline provides one example of a circulation problem. In this setting, any airplane circulates among the airports of various cities; the lower bound l_{ij} imposed on an arc (i, j) is 1 if the airline needs to provide service between cities i and j, and so must dispatch an airplane on this arc (actually, the nodes will represent a combination of both a physical location and a time of day so that an arc connects, for example, New York City at 8 A.M. with Boston at 9 A.M.).

In this book, we also study the following generalizations of the minimum cost flow problem.

Convex cost flow problems. In the minimum cost flow problem, we assume that the cost of the flow on any arc varies linearly with the amount of flow. Convex cost flow problems have a more general cost structure: The cost is a convex function of the amount of flow. Flow costs vary in a convex manner in numerous problem settings, including (1) power losses in an electrical network due to resistance, (2) congestion costs in a city transportation network, and (3) expansion costs of a communication network.

Generalized flow problems. In the minimum cost flow problem, arcs conserve flows (i.e., the flow entering an arc equals the flow leaving the arc). In generalized flow problems, arcs might "consume" or "generate" flow. If x_{ij} units of flow enter an arc (i, j), then $\mu_{ij}x_{ij}$ units arrive at node j; μ_{ij} is a positive *multiplier* associated with the arc. If $0 < \mu_{ij} < 1$, the arc is *lossy*, and if $1 < \mu_{ij} < \infty$, the arc is *gainy*. Generalized network flow problems arise in several application contexts: for example, (1) power transmission through electric lines, with power lost with distance traveled, (2) flow of water through pipelines or canals that lose water due to seepage or evaporation, (3) transportation of a perishable commodity, and (4) cash management scenarios in which arcs represent investment opportunities and multipliers represent appreciation or depreciation of an investment's value.

Multicommodity flow problems. The minimum cost flow problem models the flow of a single commodity over a network. Multicommodity flow problems arise when several commodities use the same underlying network. The commodities may either be differentiated by their physical characteristics or simply by their origin–destination pairs. Different commodities have different origins and destinations, and commodities have separate mass balance constraints at each node. However, the sharing of the common arc capacities binds the different commodities together. In fact, the essential issue addressed by the multicommodity flow problem is the allocation of the capacity of each arc to the individual commodities in a way that minimizes overall flow costs. Multicommodity flow problems arise in many practical situations, including (1) the transportation of passengers from different origins to different destinations within a city; (2) the routing of nonhomogeneous tankers (nonhomogeneous in terms of speed, carrying capability, and operating costs); (3) the worldwide shipment of different varieties of grains (such as corn, wheat, rice, and soybeans) from countries that produce grains to those that consume it; and (4) the transmission of messages in a communication network between different origin–destination pairs.

Other Models

In this book we also study two other important network models: the *minimum spanning tree problem* and the *matching problem*. Although these two models are not flow problems per se, because of their practical and mathematical significance and because of their close connection with several flow problems, we have included them as part of our treatment of network flows.

Minimum spanning tree problem. A spanning tree is a tree (i.e., a connected acyclic graph) that spans (touches) all the nodes of an undirected network. The cost of a spanning tree is the sum of the costs (or lengths) of its arcs. In the minimum spanning tree problem, we wish to identify a spanning tree of minimum cost (or length). The applications of the minimum spanning tree problem are varied and include (1) constructing highways or railroads spanning several cities; (2) laying pipelines connecting offshore drilling sites, refineries, and consumer markets; (3) designing local access networks; and (4) making electric wire connections on a control panel.

Matching problems. A *matching* in a graph $G = (N, A)$ is a set of arcs with the property that every node is incident to at most one arc in the set; thus a matching induces a pairing of (some of) the nodes in the graph using the arcs in A. In a matching, each node is matched with at most one other node, and some nodes might not be matched with any other node. The *matching problem* seeks a matching that optimizes some criteria. Matching problems on a bipartite graphs (i.e., those with two sets of nodes and with arcs that join only nodes between the two sets, as in the assignment and transportation problems) are called *bipartite matching problems*, and those on nonbipartite graphs are called *nonbipartite matching problems*. There are two additional ways of categorizing matching problems: *cardinality matching problems*, which maximize the number of pairs of nodes matched, and *weighted matching problems*, which maximize or minimize the weight of the matching. The weighted matching problem on a bipartite graph is also known as the *assignment problem*. Applications of matching problems arise in matching roommates to hostels, matching pilots to compatible airplanes, scheduling airline crews for available flight legs, and assigning duties to bus drivers.

1.3 APPLICATIONS

Networks are pervasive. They arise in numerous application settings and in many forms. Physical networks are perhaps the most common and the most readily identifiable classes of networks; and among physical networks, transportation networks are perhaps the most visible in our everyday lives. Often, these networks model homogeneous facilities such as railbeds or highways. But on other occasions, they correspond to composite entities that model, for example, complex distribution and logistics decisions. The traditional operations research "transportation problem" is illustrative. In the transportation problem, a shipper with inventory of goods at its warehouses must ship these goods to geographically dispersed retail centers, each with a given customer demand, and the shipper would like to meet these demands incurring the minimum possible transportation costs. In this setting, a transportation link in the underlying network might correspond to a complex distribution channel with, for example, a trucking shipment from the warehouse to a railhead, a rail shipment, and another trucking leg from the destination rail yard to the customer's site.

Physical networks are not limited to transportation settings; they also arise in several other disciplines of applied science and engineering, such as mathematics, chemistry, and electrical, communications, mechanical, and civil engineering. When physical networks occur in these different disciplines, their nodes, arcs, and flows model many different types of physical entities. For example, in a typical communication network, nodes will represent telephone exchanges and transmission facilities, arcs will denote copper cables or fiber optic links, and flow would signify the transmission of voice messages or of data. Figure 1.1 shows some typical associations for the nodes, arcs, and flows in a variety of physical networks.

Network flow problems also arise in surprising ways for problems that on the surface might not appear to involve networks at all. Sometimes these applications are linked to a physical entity, and at other times they are not. Sometimes the nodes and arcs have a temporal dimension that models activities that take place over time.

Applications	Physical analog of nodes	Physical analog of arcs	Flow
Communication systems	Telephone exchanges, computers, transmission facilities, satellites	Cables, fiber optic links, microwave relay links	Voice messages, data, video transmissions
Hydraulic systems	Pumping stations, reservoirs, lakes	Pipelines	Water, gas, oil, hydraulic fluids
Integrated computer circuits	Gates, registers, processors	Wires	Electrical current
Mechanical systems	Joints	Rods, beams, springs	Heat, energy
Transportation systems	Intersections, airports, rail yards	Highways, railbeds, airline routes	Passengers, freight, vehicles, operators

Figure 1.1 Ingredients of some common physical networks.

Many scheduling applications have this flavor. In any event, networks model a variety of problems in project, machine, and crew scheduling; location and layout theory; warehousing and distribution; production planning and control; and social, medical, and defense contexts. Indeed, these various applications of network flow problems seem to be more widespread than are the applications of physical networks. We present many such applications throughout the text and in the exercises; Chapter 19, in particular, brings together and summarizes many applications. In the following discussion, to set a backdrop for the next few chapters, we describe several sample applications that are intended to illustrate a range of problem contexts and to be suggestive of how network flow problems arise in practice. This set of applications provides at least one example of each of the network models that we introduced in the preceding section.

Application 1.1 Reallocation of Housing

A housing authority has a number of houses at its disposal that it lets to tenants. Each house has its own particular attributes. For example, a house might or might not have a garage, it has a certain number of bedrooms, and its rent falls within a particular range. These variable attributes permit us to group the house into several categories, which we index by $i = 1, 2, \ldots, n$.

Over a period of time a number of tenants will surrender their tenancies as they move or choose to live in alternative accommodations. Furthermore, the requirements of the tenants will change with time (because new families arrive, children leave home, incomes and jobs change, and other considerations). As these changes occur, the housing authority would like to relocate each tenant to a house of his or her choice category. While the authority can often accomplish this objective by simple exchanges, it will sometimes encounter situations requiring multiple moves: moving one tenant would replace another tenant from a house in a different category, who, in turn, would replace a tenant from a house in another category, and so on, thus creating a cycle of changes. We call such a change a *cyclic change*. The decision

problem is to identify a cyclic change, if it exists, or to show that no such change exists.

To solve this problem as a network problem, we first create a *relocation graph* G whose nodes represent various categories of houses. We include arc (i, j) in the graph whenever a person living in a house of category i wishes to move to a house of category j. A directed cycle in G specifies a cycle of changes that will satisfy the requirements of one person in each of the categories contained in the cycle. Applying this method iteratively, we can satisfy the requirements of an increasing number of persons.

This application requires a method for identifying directed cycles in a network, if they exist. A well-known method, known as *topological sorting*, will identify such cycles. We discuss topological sorting in Section 3.4. In general, many tenant reassignments might be possible, so the relocation graph G might contain several cycles. In that case the authority's management would typically want to find a cycle containing as few arcs as possible, since fewer moves are easier to handle administratively. We can solve this problem using a shortest path algorithm (see Exercise 5.38).

Application 1.2 Assortment of Structural Steel Beams

In its various construction projects, a construction company needs structural steel beams of a uniform cross section but of varying lengths. For each $i = 1, \ldots, n$, let $D_i > 0$ denote the demand of the steel beam of length L_i, and assume that $L_1 < L_2 < \cdots < L_n$. The company could meet its needs by maintaining and drawing upon an inventory of stock containing exactly D_i units of the steel beam of length L_i. It might not be economical to carry all the demanded lengths in inventory, however, because of the high cost of setting up the inventory facility to store and handle each length. In that case, if the company needs a beam of length L_i not carried in inventory, it can cut a beam of longer length down to the desired length. The cutting operation will typically produce unusable steel as scrap. Let K_i denote the cost for setting up the inventory facility to handle beams of length L_i, and let C_i denote the cost of a beam of length L_i. The company wants to determine the lengths of beams to be carried in inventory so that it will minimize the total cost of (1) setting up the inventory facility, and (2) discarding usable steel lost as scrap.

We formulate this problem as a shortest path problem as follows. We construct a directed network G on $(n + 1)$ nodes numbered $0, 1, 2, \ldots, n$; the nodes in this network correspond to various beam lengths. Node 0 corresponds to a beam of length zero and node n corresponds to the longest beam. For each node i, the network contains a directed arc to every node $j = i + 1, i + 2, \ldots, n$. We interpret the arc (i, j) as representing a storage strategy in which we hold beams of length L_j in inventory and use them to satisfy the demand of all the beams of lengths L_{i+1}, $L_{i+2}, \ldots, L_j$. The cost c_{ij} of the arc (i, j) is

$$c_{ij} = K_j + C_j \sum_{k=i+1}^{j} D_k.$$

The cost of arc (i, j) has two components: (1) the fixed cost K_j of setting up the inventory facility to handle beams of length L_j, and (2) the cost of using beams of length L_j to meet the demands of beams of lengths $L_{i+1}, \ldots, L_j$. A directed path

from node 0 to node n specifies an assortment of beams to carry in inventory and the cost of the path equals the cost associated with this inventory scheme. For example, the path 0–4–6–9 corresponds to the situation in which we set up the inventory facility for handling beams of lengths L_4, L_6, and L_9. Consequently, the shortest path from node 0 to node n would prescribe the least cost assortment of structural steel beams.

Application 1.3 Tournament Problem

Consider a round-robin tournament between n teams, assuming that each team plays against every other team c times. Assume that no game ends in a draw. A person claims that α_i for $1 \le i \le n$ denotes the number of victories accrued by the ith team at the end of the tournament. How can we determine whether the given set of non-negative integers $\alpha_1, \alpha_2, \ldots, \alpha_n$ represents a possible winning record for the n teams?

Define a directed network $G = (N, A)$ with node set $N = \{1, 2, \ldots, n\}$ and arc set $A = \{(i, j) \in N \times N : i < j\}$. Therefore, each node i is connected to the nodes $i + 1, i + 2, \ldots, n$. Let x_{ij} for $i < j$ represent the number of times team i defeats team j. Observe that the total number of times team i beats teams $i + 1, i + 2, \ldots, n$ is $\sum_{\{j:(i,j)\in A\}} x_{ij}$. Also observe that the number of times that team i beats a team $j < i$ is $c - x_{ji}$. Consequently, the total number of times that team i beats teams $1, 2, \ldots, i - 1$ is $(i - 1)c - \sum_{\{j:(j,i)\in A\}} x_{ji}$. The total number of wins of team i must equal the total number of times it beats the teams $1, 2, \ldots, n$. The preceding observations show that

$$\sum_{\{j:(i,j)\in A\}} x_{ij} - \sum_{\{j:(j,i)\in A\}} x_{ji} = \alpha_i - (i - 1)c \qquad \text{for all } i \in N. \tag{1.3}$$

In addition, a possible winning record must also satisfy the following lower and upper bound conditions:

$$0 \le x_{ij} \le c \qquad \text{for all } (i, j) \in A. \tag{1.4}$$

This discussion shows that the record α_i is a possible winning record if the constraints defined by (1.3) and (1.4) have a feasible solution x. Let $b(i) = \alpha_i - (i - 1)c$. Observe that the expressions $\sum_{i\in N}\alpha_i$ and $\sum_{i\in N}(i - 1)c$ are both equal to $cn(n - 1)/2$, which is the total number of games played. Consequently, $\sum_{i\in N}b(i) = 0$. The problem of finding a feasible solution of a network flow system like (1.3) and (1.4) is called a *feasible flow problem* and can be solved by solving a maximum flow problem (see Section 6.2).

Application 1.4 Leveling Mountainous Terrain

This application was inspired by a common problem facing civil engineers when they are building road networks through hilly or mountainous terrain. The problem concerns the distribution of earth from high points to low points of the terrain to produce a leveled roadbed. The engineer must determine a plan for leveling the route by

specifying the number of truckloads of earth to move between various locations along the proposed road network.

We first construct a *terrain graph*: it is an undirected graph whose nodes represent locations with a demand for earth (low points) or locations with a supply of earth (high points). An arc of this graph represents an available route for distributing the earth, and the cost of this arc represents the cost per truckload of moving earth between the two points. (A *truckload* is the basic unit for redistributing the earth.) Figure 1.2 shows a portion of the terrain graph.

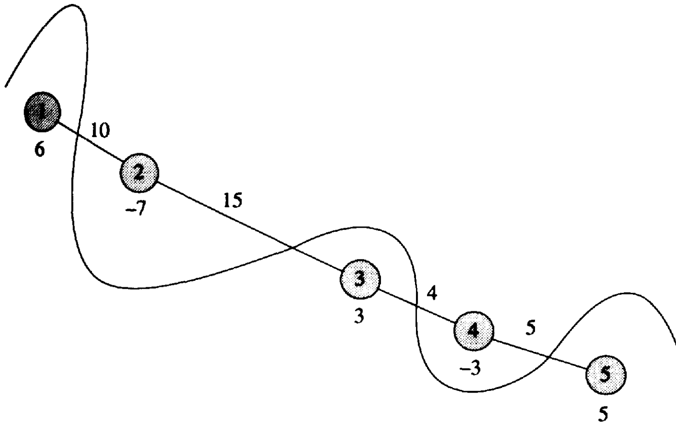

Figure 1.2 Portion of the terrain graph.

A leveling plan for a terrain graph is a flow (set of truckloads) that meets the demands at nodes (levels the low points) by the available supplies (by earth obtained from high points) at the minimum cost (for the truck movements). This model is clearly a minimum cost flow problem in the terrain graph.

Application 1.5 Rewiring of Typewriters

For several years, a company had been using special electric typewriters to prepare punched paper tapes to enter data into a digital computer. Because the typewriter is used to punch a six-hole paper tape, it can prepare $2^6 = 64$ binary hole/no-hole patterns. The typewriters have 46 characters, and each punches one of the 64 patterns. The company acquired a new digital computer that uses a different coding hole/no-hole patterns to represent characters. For example, using 1 to represent a hole and 0 to represent a no-hole, the letter A is 111100 in the code for the old computer and 011010 in the code for the new computer. The typewriter presently punches the former and must be modified to punch the latter.

Each key in the typewriter is connected to a steel code bar, so changing the code of that key requires mechanical changes in the steel bar system. The extent of the changes depends on how close the new and old characters are to each other. For the letter A, the second, third, and sixth bits are identical in the old and new codes and no changes need be made for these bits; however, the first, fourth, and fifth bits are different, so we would need to make three changes in the steel code bar connected to the A-key. Each change involves removing metal at one place and

adding metal at another place. When a key is pressed, its steel code bar activates six cross-bars (which are used by all the keys) that are connected electrically to six hole punches. If we interchange the fourth and fifth wires of the cross-bars to the hole punches (which is essentially equivalent to interchanging the fourth and fifth bits of all characters in the old code), we would reduce the number of mechanical changes needed for the A-key from three to one. However, this change of wires might increase the number of changes for some of the other 45 keys. The problem, then, is how to optimally connect the wires from the six cross-bars to the six punches so that we can minimize the number of mechanical changes on the steel code bars.

We formulate this problem as an assignment problem as follows. Define a network $G = (N_1 \cup N_2, A)$ with node sets $N_1 = \{1, 2, \ldots, 6\}$ and $N_2 = \{1', 2', \ldots, 6'\}$, and an arc set $A = N_1 \times N_2$; the cost of the arc $(i, j') \in A$ is the number of keys (out of 46) for which the ith bit in the old code differs from the jth bit in the new code. Thus if we assign cross-bar i to the punch j, the number of mechanical changes needed to print the ith bit of each symbol correctly is c_{ij}. Consequently, the minimum cost assignment will minimize the number of mechanical changes.

Application 1.6 Pairing Stereo Speakers

As a part of its manufacturing process, a manufacturer of stereo speakers must pair individual speakers before it can sell them as a set. The performance of the two speakers depends on their frequency response. To measure the quality of the pairs, the company generates matching coefficients for each possible pair. It calculates these coefficients by summing the absolute differences between the responses of the two speakers at 20 discrete frequencies, thus giving a matching coefficient value between 0 and 30,000. Bad matches yield a large coefficient, and a good pairing produces a low coefficient.

The manufacturer typically uses two different objectives in pairing the speakers: (1) finding as many pairs as possible whose matching coefficients do not exceed a specification limit, or (2) pairing speakers within specification limits to minimize the total sum of the matching coefficients. The first objective minimizes the number of pairs outside specification, and so the number of speakers that the firm must sell at a reduced price. This model is an application of the nonbipartite cardinality matching problem on an undirected graph: the nodes of this graph represent speakers and arcs join two nodes if the matching coefficients of the corresponding speakers are within the specification limit. The second model is an application of the nonbipartite weighted matching problem.

Application 1.7 Measuring Homogeneity of Bimetallic Objects

This application shows how a minimum spanning tree problem can be used to determine the degree to which a bimetallic object is homogeneous in composition. To use this approach, we measure the composition of the bimetallic object at a set of sample points. We then construct a network with nodes corresponding to the sample

points and with an arc connecting physically adjacent sample points. We assign a cost with each arc (i, j) equal to the product of the physical (Euclidean) distance between the sample points i and j and a homogeneity factor between 0 and 1. This homogeneity factor is 0 if the composition of the corresponding samples is exactly alike, and is 1 if the composition is very different; otherwise, it is a number between 0 and 1. Note that this measure gives greater weight to two points if they are different and are far apart. The cost of the minimum spanning tree is a measure of the homogeneity of the bimetallic object. The cost of the tree is 0 if all the sample points are exactly alike, and high cost values imply that the material is quite nonhomogeneous.

Application 1.8 Electrical Networks

The electrical network shown in Figure 1.3 has eight resistors, two current sources (at nodes 1 and 6), and one current sink (at node 7). In this network we wish to determine the equilibrium current flows through the resistors. A popular method for solving this problem is to introduce a variable x_{ij} representing the current flow on the arc (i, j) of the electrical network and write a set of equilibrium relationships for these flows; that is, the voltage–current relationship equations (using Ohm's law) and the current balance equations (using Kirchhof's law). The solution of these equations gives the arc currents x_{ij}. An alternative, and possibly more efficient approach is to formulate this problem as a convex cost flow problem. This formulation uses the well-known result that the equilibrium currents on resistors are those flows for which the resistors dissipate the least amount of the total power supplied by the voltage sources (i.e., the electric current follows the path of least resistance). Ohm's law shows that a resistor of resistance r_{ij} dissipates $r_{ij}x_{ij}^2$ watts of power. Therefore, we can obtain the optimal currents by solving the following convex cost flow problem:

$$\text{Minimize} \sum_{(i,j)\in A} r_{ij}x_{ij}^2$$

subject to

$$\sum_{\{j:(i,j)\in A\}} x_{ij} - \sum_{\{j:(j,i)\in A\}} x_{ji} = b(i) \qquad \text{for each node } i \in N,$$

$$x_{ij} \geq 0 \qquad \text{for each arc } (i,j) \in A.$$

In this model $b(i)$ represents the supply/demand of a current source or sink.

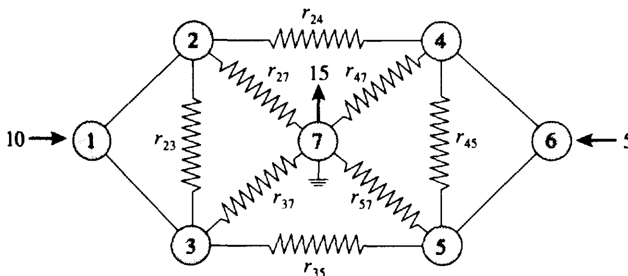

Figure 1.3 Electrical network.

The formulation of a set of equilibrium conditions as an equivalent optimization model is a powerful idea in the physical sciences, dating from the last century, which has become known as so-called *variational principles*. The term "variational" arises because the equilibrium conditions are the "optimality conditions" for the equivalent optimization model that tell us that we cannot improve the optimal solution by varying (hence the term "variational") the optimal solution to this optimization model.

Application 1.9 Determining an Optimal Energy Policy

As part of their national planning effort, most countries need to decide on an energy policy (i.e., how to utilize the available raw materials to satisfy their energy needs). Assume, for simplicity, that a particular country has four basic raw materials: crude oil, coal, uranium, and hydropower; and it has four basic energy needs: electricity, domestic oil, petroleum, and gas. The country has the technological base and infrastructure to convert each raw material into one or more energy forms. For example, it can convert crude oil into domestic oil or petrol, coal into electricity, and so on. The available technology base specifies the efficiency and the cost of each conversion. The objective is to satisfy, at the least possible cost of energy conversion, a certain annual consumption level of various energy needs from a given annual production of raw materials.

Figure 1.4 shows the formulation of this problem as a generalized network flow problem. The network has three types of arcs: (1) *source arcs* (s, i) emanating from the source node s, (2) *sink arcs* (j, t) entering the sink node t, and (3) *conversion arcs* (i, j). The source arc (s, i) has a capacity equal to the availability $\alpha(i)$ of the raw material i and a flow multiplier of value 1. The sink arc (j, t) has capacity equal to the demand $\beta(j)$ of type j energy need and flow multiplier of value 1. Each conversion arc (i, j) represents the conversion of raw material i into the energy form j; the multiplier of this arc is the efficiency of the conversion (i.e., units of energy j obtained from 1 unit of raw material i); and the cost of the arc (i, j) is the cost of this conversion. In this model, since $\alpha(i)$ is an upper bound on the use of raw material

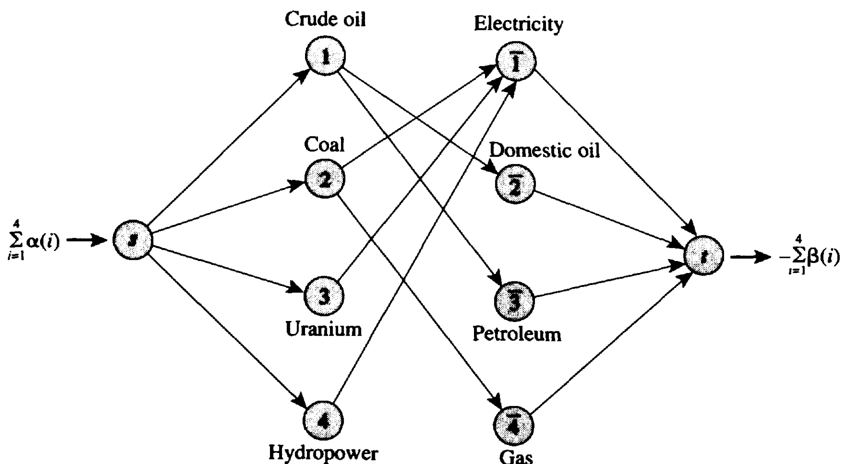

Figure 1.4 Energy problem as a generalized network flow problem.

i, $\sum_{i=1}^{4} \alpha(i)$ is an upper bound on the flow out of node s. Similarly, $\sum_{i=1}^{4} \beta(i)$ is a lower bound on the flow into node t. In Exercise 15.29, we show how to convert this problem into a standard form without bounds on supplies and demands.

Application 1.10 Racial Balancing of Schools

In 1968, the U.S. Supreme Court ruled that all school systems in the country should begin admitting students to schools on a nondiscriminatory basis and should employ faster techniques to promote desegregated schools across the nation. This decision made it necessary for many school systems to develop radically different procedures for assigning students to schools. Since the Supreme Court did not specify what constitutes an acceptable racial balance, the individual school boards used their own best judgments to arrive at acceptable criteria on which to base their desegregation plans. This application describes a multicommodity flow model for determining an optimal assignment of students to schools that minimizes the total distance traveled by the students, given a specification of lower and upper limits on the required racial balance in each school.

Suppose that a school district has S schools and school j has capacity u_j. For the purpose of this formulation, we divide the school district into L population centers. These locations might, for example, be census tracts, bus stops, or city blocks. The only restriction on the population centers is that they be finite in number and that a single distance measure reasonably approximates the distance any student at center i must travel if he or she is assigned to school j. Let S_{ik} denote the available number of students of the kth ethnic group at the ith population center. The objective is to assign students to schools in a way that achieves the desired ethnic composition for each school and minimizes the total distance traveled by the students. Each school j has the ethnic requirement that it must enroll at least l_{jk} and no more than u_{jk} students from the kth ethnic group.

We can model this problem as a multicommodity flow problem on an appropriately defined network. Figure 1.5 shows this network representation for a problem with three population centers and three schools. This network has one node for each population center and for each school as well as a "source" and a "sink" node for each ethnic group. The flow commodities represent the students of different ethnic groups. The students of the kth ethnic group flow from source a_k to sink e_k via population center and school nodes. We set the upper bound on arc (a_k, b_i) connecting the kth ethnic group source node and the ith population center equal to S_{ik} and the cost of the arc (b_i, c_j) connecting the ith population center and jth school equal to f_{ij}, the distance between that population center and that school. By setting the capacity of the arc (c_j, d_j) equal to u_j, we ensure that the total number of students (of all ethnic groups) allocated to the jth school does not exceed the maximum student population for this school. The students of all ethnic groups must share the capacity of each school. Finally, we incorporate constraints on the ethnic compositions of the schools by setting the lower and upper bounds on the arc (d_j, e_k) equal to l_{jk} and u_{jk}. It is fairly easy to verify that the multicommodity flow problem models the racial balancing problem, so a minimum multicommodity flow will specify an optimal assignment of students to the schools.

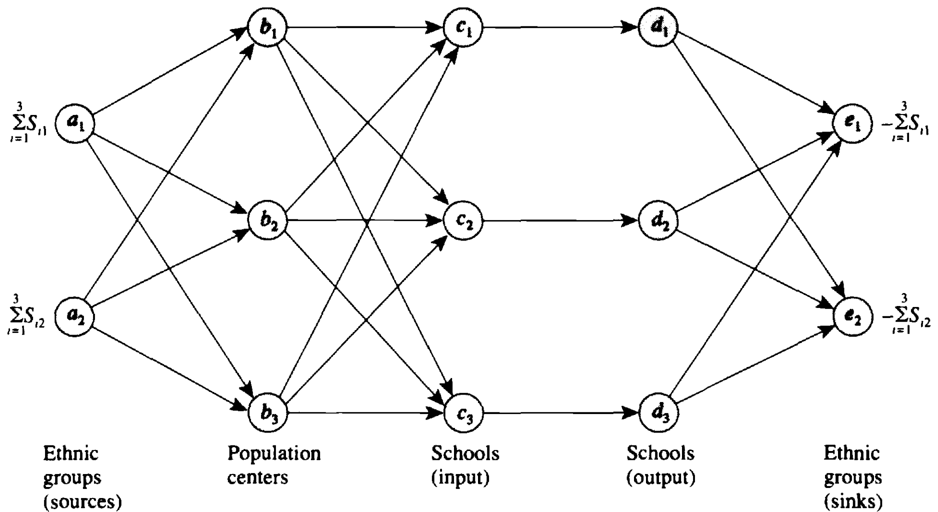

Figure 1.5 Formulating the racial balancing problem as a multicommodity flow problem.

1.4 SUMMARY

In this chapter we introduced the network flow problems that we study in this book and described a few scenarios in which these problems arise. We began by giving a linear programming formulation of the minimum cost flow problem and identifying several special cases: the shortest path problem, the maximum flow problem, the assignment problem, the transportation problem, and the circulation problem. We next described several generalizations of the minimum cost flow problem: the convex cost flow problem, the generalized network flow problem, and the multicommodity flow problem. Finally, we described two other important network models: the minimum spanning tree problem and the matching problem. Although these two problems are not network flow problems per se, we have included them in this book because they are closely related to several network flow problems and because they arise often in the context of network optimization.

Networks are pervasive and arise in numerous application settings. Physical networks, which are the most readily identifiable classes of networks, arise in many applications in many different types of systems: communications, hydraulic, mechanical, electronic, and transportation. Network flow problems also arise in surprising ways in optimization problems that on the surface might not appear to involve networks at all. We described several of these "indirect" applications of network flow problems, in such problem settings as urban housing, production planning, electrical networks, racial balancing, leveling mountainous terrain, evaluating tournaments, matching stereo speakers, wiring typewriters, assessing the homogeneity of physical materials, and energy planning. The applications we have considered offer only a brief glimpse of the wide-ranging practical importance of network flows; although our discussion of applications in this chapter is limited, it does provide at least one example of each of the network models that we have introduced in this chapter.

REFERENCE NOTES

The study of network flow models predates the development of linear programming. The first studies in this problem domain, conducted by Kantorovich [1939], Hitchcock [1941], and Koopmans [1947], considered the transportation problem, a special case of the minimum cost flow problem. These studies provided insight into the problem structure and yielded algorithmic approaches. Interest in the network flow problems grew with the advent of the simplex method by Dantzig in 1947, who also specialized this algorithm for the transportation problem (see Dantzig [1951]).

During the 1950s, researchers began to exhibit increasing interest in the minimum cost flow problem and its specializations—the shortest path problem, the maximum flow problem, and the assignment problem—mainly because of the importance of these models in real-world applications. Soon researchers developed special algorithms for solving these problems. Dantzig, Ford, and Fulkerson pioneered these efforts. Whereas Dantzig focused on the simplex-based methods, Ford and Fulkerson developed primal–dual combinatorial algorithms. The landmark books by Dantzig [1962] and Ford and Fulkerson [1962] present thorough discussions of these early contributions.

In the years following this groundbreaking work, network flow problems and their generalizations emerged as major research topics in thousands of papers and numerous text and reference books. The following books summarize developments in the field and serve as a guide to the literature:

1. *Flows in Networks* (Ford and Fulkerson [1962])
2. *Programming, Games and Transportation Networks* (Berge and Ghouila-Houri [1962])
3. *Finite Graphs and Networks* (Busacker and Saaty [1965])
4. *Network Flow, Transportation and Scheduling* (Iri [1969])
5. *Integer Programming and Network Flows* (Hu [1969])
6. *Communication, Transmission, and Transportation Networks* (Frank and Frisch [1971])
7. *Flows in Transportation Networks* (Potts and Oliver [1972])
8. *Graph Theory: An Algorithmic Approach* (Christophides [1975])
9. *Flow Algorithms* (Adel'son-Vel'ski, Dinics, and Karzanov [1975])
10. *Graph Theory with Applications* (Bondy and Murty [1976])
11. *Combinatorial Optimization: Networks and Matroids* (Lawler [1976])
12. *Optimization Algorithms for Networks and Graphs* (Minieka [1978])
13. *Graph Algorithms* (Even [1979])
14. *Algorithms for Network Programming* (Kennington and Helgason [1980])
15. *Network Flow Programming* (Jensen and Barnes [1980])
16. *Fundamentals of Network Analysis* (Phillips and Garcia-Diaz [1981])
17. *Combinatorial Optimization: Algorithms and Complexity* (Papadimitriou and Steiglitz [1982])
18. *Discrete Optimization Algorithms* (Syslo, Deo, and Kowalik [1983])
19. *Data Structures and Network Algorithms* (Tarjan [1983])

20. *Graphs and Algorithms* (Gondran and Minoux [1984])
21. *Network Flows and Monotropic Optimization* (Rockafellar [1984])
22. *Linear Programming and Network Models* (Gupta [1985])
23. *Programming in Networks and Graphs* (Derigs [1988])
24. *Linear Programming and Network Flows*, 2nd ed. (Bazaraa, Jarvis, and Sherali [1990])

As an additional source of references, the reader might consult the bibliographies on network optimization prepared by Golden and Magnanti [1977], Ahuja, Magnanti, and Orlin [1989, 1991], Bazaraa, Jarvis, and Sherali [1990], and the extensive set of references on integer programming compiled by researchers at the University of Bonn (Kastning [1976], Hausman [1978], and Von Randow [1982, 1985]).

Since the applications of network flow models are so pervasive, no single source provides a comprehensive account of network flow models and their impact on practice. Several researchers have prepared general surveys of selected application areas. Notable among these are the papers by Bennington [1974], Glover and Klingman [1976], Bodin, Golden, Assad, and Ball [1983], Aronson [1989], and Glover, Klingman, and Phillips [1990]. The book by Gondran and Minoux [1984] also describes a variety of applications of network flow problems. In this book we describe or cite over 150 selected applications of network flow problems. We provide the references for these problems in the reference notes given at the end of Chapters 4, 6, 9, 12, 13, 14, 15, 16, 17, and 19. We have adapted many of these applications from the paper of Ahuja, Magnanti, Orlin, and Reddy [1992].

The applications we present in Section 1.3 are adapted from the following references:

1. Reallocation of housing (Wright [1975])
2. Assortment of structural steel beams (Frank [1965])
3. Tournament problem (Ford and Johnson [1959])
4. Leveling mountainous terrain (Farley [1980])
5. Rewiring of typewriters (Machol [1961])
6. Pairing stereo speakers (Mason and Philpott [1988])
7. Measuring homogeneity of bimetallic objects (Shier [1982])
8. Electrical networks (Hu [1966])
9. Determining an optimal energy policy (Gondran and Minoux [1984])
10. Racial balancing of schools (Clarke and Surkis [1968])

EXERCISES

1.1. Formulate the following problems as circulation problems: (1) the shortest path problem; (2) the assignment problem; and (3) the transportation problem.

1.2. Consider a variant of the transportation problem for which (1) the sum of demands exceeds the sum of supplies, and (2) we incur a penalty p_j for every unit of unfulfilled demand at demand node j. Formulate this problem as a standard transportation problem with total supply equal to total demand.

1.3. In this exercise we examine a generalization of Application 1.2, concerning assortment of structural steel beams. In the discussion of that application, we assumed that if we must cut a beam of length 5 units to a length of 2 units, we obtain a single beam of length 2 units; the remaining 3 units have no value. However, in practice, from a beam of length 5 we can cut two beams of length 2; the remaining length of 1 unit will have some scrap value. Explain how we might incorporate the possibility of cutting multiple beam lengths (of the same length) from a single piece and assigning some salvage value to the scrap. Assume that the scrap has a value of β per unit length.

1.4. **Large-scale personnel assignment.** A recurring problem in the U.S. armed forces is efficient distribution and utilization of skilled personnel. Each month thousands of individuals in the U.S. military vacate jobs, and thousands of personnel become available for assignment. Each job has particular characteristics and skill requirements, while each person from the pool of available personnel has specific skills and preferences. Suppose that we use this information to compute the utility (or desirability) d_{ij} of each possible assignment of a person to a job. The decision problem is to assign personnel to the vacancies in a way that maximizes the total utility of all the assignments. Explain how to formulate this problem as a network flow problem.

1.5. **Dating problem.** A dating service receives data from p men and p women. These data determine what pairs of men and women are mutually compatible. Since the dating service's commission is proportional to the number of dates it arranges, it would like to determine the maximum number of compatible couples that can be formed. Formulate this problem as a matching problem.

1.6. **Pruned chessboard problem.** A chessboard consists of 64 squares arranged in eight rows and eight columns. A *domino* is a wooden or plastic piece consisting of two squares joined on a side. Show that it is possible to fully cover the chessboard using 32 dominos (i.e., each domino covers two squares of the board, no two dominos overlap, and some domino covers each square). A *pruned board* is a chessboard with some squares removed.

(a) Suppose that we want to know whether it is possible to fully cover a pruned board, and if not, to find the maximum number of dominos we can place on the pruned board so that each domino covers two squares and no two dominos overlap. Formulate this problem as a bipartite cardinality matching problem.

(b) Suppose that we prune only two diagonally opposite corners of the chessboard. Show that we cannot cover the resulting board with 31 dominos.

1.7. **Paragraph problem.** The well-known document processing program TeX uses an optimization procedure to decompose a paragraph into several lines so that when lines are left- and right-adjusted, the appearance of the paragraph will be the most attractive. Suppose that a paragraph consists of n words and that each word is assigned a sequence number. Let c_{ij} denote the attractiveness of a line if it begins with the word i and ends with the word $j - 1$. The program TeX uses formulas to compute the value of each c_{ij}. Given the c_{ij}'s, show how to formulate the problem of decomposing the paragraph into several lines of text in order to maximize the total attractiveness (of all lines) as a shortest path problem.

1.8. **Seat-sharing problem.** Several families are planning a shared car trip on scenic drives in the White Mountains, New Hampshire. To minimize the possibility of any quarrels, they want to assign individuals to cars so that no two members of a family are in the same car. Formulate this problem as a network flow problem.

1.9. **Police patrol problem** (Khan [1979]). A police department in a small city consists of three precincts denoted p_1, p_2, and p_3. Each precinct is assigned a number of patrol cars equipped with two-way radios and first-aid equipment. The department operates with three shifts. Figure 1.6(a) and (b) shows the minimum and maximum number of patrol cars needed in each shift. Administrative constraints require that (1) shifts 1, 2, and 3 have, respectively, at least cars 10, 20, and 18 cars available; and (2) precincts p_1, p_2, and p_3 are, respectively, allocated at least 10, 14, and 13 cars. The police department wants to determine an allocation of patrol units that will meet all the require-

	Shift 1	Shift 2	Shift 3
p_1	2	4	3
p_2	3	6	5
p_3	5	7	6

(a)

	Shift 1	Shift 2	Shift 3
p_1	3	7	5
p_2	5	7	10
p_3	8	12	10

(b)

Figure 1.6 Patrol car requirements: (a) minimum required per shift; (b) maximum required per shift.

ments with the fewest possible units committed to the field. Formulate this problem as a circulation problem.

1.10. Forest scheduling problem. Paper and wood products companies need to define cutting schedules that will maximize the total wood yield of their forests over some planning period. Suppose that a company with control of p forest units wants to identify the best cutting schedule over a planning horizon of k years. Forest unit i has a total acreage of a_i units, and studies that the company has undertaken predict that this unit will have w_{ij} tons of woods available for harvesting in the jth year. Based on its prediction of economic conditions, the company believes that it should harvest at least l_j tons of wood in year j. Due to the availability of equipment and personnel, the company can harvest at most u_j tons of wood in year j. Formulate the problem of determining a schedule with maximum wood yield as a network flow problem.

2

PATHS, TREES, AND CYCLES

I hate definitions.
—Benjamin Disraeli

Chapter Outline

2.1 Introduction
2.2 Notation and Definitions
2.3 Network Representations
2.4 Network Transformations
2.5 Summary

2.1 INTRODUCTION

Because graphs and networks arise everywhere and in a variety of alternative forms, several professional disciplines have contributed important ideas to the evolution of network flows. This diversity has yielded numerous benefits, including the infusion of many rich and varied perspectives. It has also, however, imposed costs: For example, the literature on networks and graph theory lacks unity and authors have adopted a wide variety of conventions, customs, and notation. If we so desired, we could formulate network flow problems in several different standard forms and could use many alternative sets of definitions and terminology. We have chosen to adopt a set of common, but not uniformly accepted, definitions: for example, arcs and nodes instead of edges and vertices (or points). We have also chosen to use models with capacitated arcs and with exogenous supplies and demands at the nodes. The circulation problem we introduced in Chapter 1, without exogenous supplies and demands, is an alternative model and so is the capacitated transportation problem. Another special case is the uncapacitated network flow problem. In Chapter 1 we viewed each of these models as special cases of the minimum cost network flow problem. Perhaps somewhat surprisingly, we could have started with any of these models and shown that all the others were special cases. In this sense, each of these models offers another way to capture the mathematical essence of network flows.

In this chapter we have three objectives. First, we bring together many basic definitions of network flows and graph theory, and in doing so, we set the notation that we will be using throughout this book. Second, we introduce several different data structures used to represent networks within a computer and discuss the relative advantages and disadvantages of each of these structures. In a very real sense, data structures are the life blood of most network flow algorithms, and choosing among alternative data structures can greatly influence the efficiency of an algorithm, both

in practice and in theory. Consequently, it is important to have a good understanding of the various available data structures and an idea of how and when to use them. Third, we discuss a number of different ways to transform a network flow problem and obtain an equivalent model. For example, we show how to eliminate flow bounds and formulate any model as an uncapacitated problem. As another example, we show how to formulate the minimum cost flow problem as a transportation problem (i.e., how to define it over a bipartite graph). This discussion is of theoretical interest, because it establishes the equivalence between several alternative models and therefore shows that by developing algorithms and theory for any particular model, we will have at hand algorithms and theory for several other models. That is, our results enjoy a certain universality. This development is also of practical value since on various occasions throughout our discussion in this book we will find it more convenient to work with one modeling assumption rather than another—our discussion of network transformations shows that there is no loss in generality in doing so. Moreover, since algorithms developed for one set of modeling assumptions also apply to models formulated in other ways, this discussion provides us with one very reassuring fact: We need not develop separate computer implementations for every alternative formulation, since by using the transformations, we can use an algorithm developed for any one model to solve any problem formulated as one of the alternative models.

We might note that many of the definitions we introduce in this chapter are quite intuitive, and much of our subsequent discussion does not require a complete understanding of all the material in this chapter. Therefore, the reader might simply wish to skim this chapter on first reading to develop a general overview of its content and then return to the chapter on an "as needed" basis later as we draw on the concepts introduced at this point.

2.2 NOTATION AND DEFINITIONS

In this section we give several basic definitions from graph theory and present some basic notation. We also state some elementary properties of graphs. We begin by defining directed and undirected graphs.

Directed Graphs and Networks: A *directed graph* $G = (N, A)$ consists of a set N of nodes and a set A of arcs whose elements are ordered pairs of distinct nodes. Figure 2.1 gives an example of a directed graph. For this graph, $N = \{1, 2, 3, 4, 5, 6, 7\}$ and $A = \{(1, 2), (1, 3), (2, 3), (2, 4), (3, 6), (4, 5), (4, 7), (5, 2), (5, 3), (5, 7), (6, 7)\}$. A *directed network* is a directed graph whose nodes and/or arcs have associated numerical values (typically,

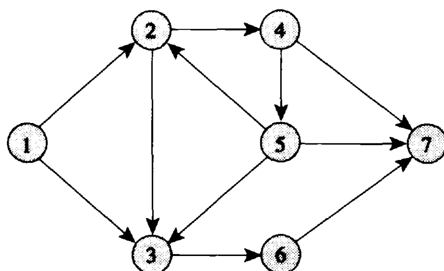

Figure 2.1 Directed graph.

costs, capacities, and/or supplies and demands). In this book we often make no distinction between graphs and networks, so we use the terms "graph" and "network" synonymously. As before, we let n denote the number of nodes and m denote the number of arcs in G.

Undirected Graphs and Networks: We define an undirected graph in the same manner as we define a directed graph except that arcs are unordered pairs of distinct nodes. Figure 2.2 gives an example of an undirected graph. In an undirected graph, we can refer to an arc joining the node pair i and j as either (i, j) or (j, i). An undirected arc (i, j) can be regarded as a two-way street with flow permitted in both directions: either from node i to node j or from node j to node i. On the other hand, a directed arc (i, j) behaves like a one-way street and permits flow only from node i to node j.

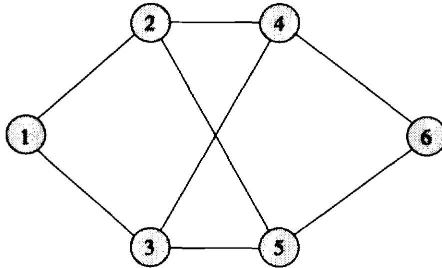

Figure 2.2 Undirected graph.

In most of the material in this book, we assume that the underlying network is directed. Therefore, we present our subsequent notation and definitions for directed networks. The corresponding definitions for undirected networks should be transparent to the reader; nevertheless, we comment briefly on some definitions for undirected networks at the end of this section.

Tails and Heads: A directed arc (i, j) has two *endpoints* i and j. We refer to node i as the *tail* of arc (i, j) and node j as its *head*. We say that the arc (i, j) *emanates* from node i and *terminates* at node j. An arc (i, j) is *incident to* nodes i and j. The arc (i, j) is an *outgoing arc* of node i and an *incoming arc* of node j. Whenever an arc $(i, j) \in A$, we say that node j is *adjacent* to node i.

Degrees: The *indegree* of a node is the number of incoming arcs of that node and its *outdegree* is the number of its outgoing arcs. The *degree* of a node is the sum of its indegree and outdegree. For example, in Figure 2.1, node 3 has an indegree of 3, an outdegree of 1, and a degree of 4. It is easy to see that the sum of indegrees of all nodes equals the sum of outdegrees of all nodes and both are equal to the number of arcs m in the network.

Adjacency List: The *arc adjacency list* $A(i)$ of a node i is the set of arcs emanating from that node, that is, $A(i) = \{(i, j) \in A : j \in N\}$. The *node adjacency list* $A(i)$ is the set of nodes adjacent to that node; in this case, $A(i) = \{j \in N: (i, j) \in A\}$. Often, we shall omit the terms "arc" and "node" and simply refer to the adjacency list; in all cases it will be clear from context whether we mean arc adjacency list or node adjacency list. We assume that arcs in the adjacency list $A(i)$ are arranged so that the head nodes of arcs are in increasing order. Notice that $| A(i) |$ equals the outdegree of node i. Since the sum of all node outdegrees equals m, we immediately obtain the following property:

Property 2.1. $\sum_{i \in N} | A(i) | = m.$

Multiarcs and Loops: *Multiarcs* are two or more arcs with the same tail and head nodes. A *loop* is an arc whose tail node is the same as its head node. In most of the chapters in this book, we assume that graphs contain no multiarcs or loops.

Subgraph: A graph $G' = (N', A')$ is a *subgraph* of $G = (N, A)$ if $N' \subseteq N$ and $A' \subseteq A$. We say that $G' = (N', A')$ is the subgraph of G *induced* by N' if A' contains each arc of A with both endpoints in N'. A graph $G' = (N', A')$ is a *spanning subgraph* of $G = (N, A)$ if $N' = N$ and $A' \subseteq A$.

Walk: A *walk* in a directed graph $G = (N, A)$ is a subgraph of G consisting of a sequence of nodes and arcs $i_1 - a_1 - i_2 - a_2 - \cdots - i_{r-1} - a_{r-1} - i_r$ satisfying the property that for all $1 \le k \le r - 1$, either $a_k = (i_k, i_{k+1}) \in A$ or $a_k = (i_{k+1}, i_k) \in A$. Alternatively, we shall sometimes refer to a walk as a set of (sequence of) arcs (or of nodes) without any explicit mention of the nodes (without explicit mention of arcs). We illustrate this definition using the graph shown in Figure 2.1. Figure 2.3(a) and (b) illustrates two walks in this graph: 1–2–5–7 and 1–2–4–5–2–3.

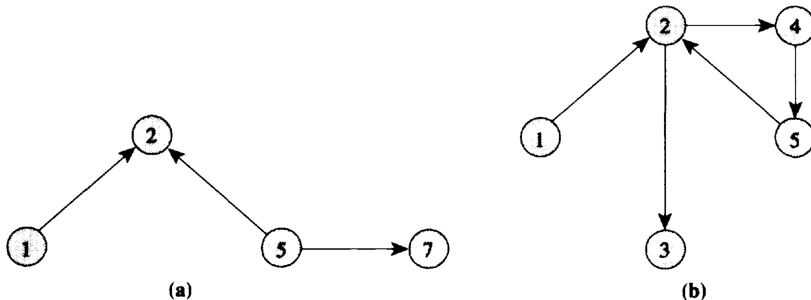

Figure 2.3 Examples of walks.

Directed Walk: A *directed walk* is an "oriented" version of a walk in the sense that for any two consecutive nodes i_k and i_{k+1} on the walk, $(i_k, i_{k+1}) \in A$. The walk shown in Figure 2.3(a) is not directed; the walk shown in Figure 2.3(b) is directed.

Path: A *path* is a walk without any repetition of nodes. The walk shown in Figure 2.3(a) is also a path, but the walk shown in Figure 2.3(b) is not because it repeats node 2 twice. We can partition the arcs of a path into two groups: forward arcs and backward arcs. An arc (i, j) in the path is a *forward arc* if the path visits node i prior to visiting node j, and is a *backward arc* otherwise. For example, in the path shown in Figure 2.3(a), the arcs (1, 2) and (5, 7) are forward arcs and the arc (5, 2) is a backward arc.

Directed Path: A *directed path* is a directed walk without any repetition of nodes. In other words, a directed path has no backward arcs. We can store a path (or a directed path) easily within a computer by defining a *predecessor* index *pred(j)* for every node j in the path. If i and j are two consecutive nodes on the path (along its orientation), pred(j) = i. For the path 1–2–5–7 shown in Figure 2.3(a), pred(7) = 5, pred(5) = 2, pred(2) = 1, and pred(1) = 0. (Frequently, we shall use the convention of setting the predecessor index of the initial node of a path equal to zero to indicate the beginning of the path.) Notice that we cannot use predecessor indices to store a walk since a walk may visit a node more than once, and a single predecessor index of a node cannot store the multiple predecessors of any node that a walk visits more than once.

Cycle: A *cycle* is a path $i_1 - i_2 - \cdots - i_r$ together with the arc (i_r, i_1) or (i_1, i_r). We shall often refer to a cycle using the notation $i_1 - i_2 - \cdots - i_r - i_1$. Just as we did for paths, we can define forward and backward arcs in a cycle. In Figure 2.4(a) the arcs (5, 3) and (3, 2) are forward arcs and the arc (5, 2) is a backward arc of the cycle 2–5–3.

Directed Cycle: A *directed cycle* is a directed path $i_1 - i_2 - \cdots - i_r$ together with the arc (i_r, i_1). The graph shown in Figure 2.4(a) is a cycle, but not a directed cycle; the graph shown in Figure 2.4(b) is a directed cycle.

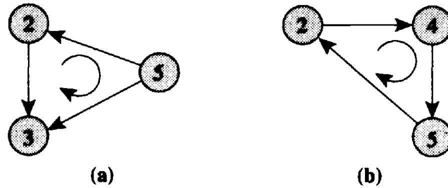

Figure 2.4 Examples of cycles.

Acyclic Graph: A graph is a *acyclic* if it contains no directed cycle.

Connectivity: We will say that two nodes i and j are *connected* if the graph contains at least one path from node i to node j. A graph is *connected* if every pair of its nodes is connected; otherwise, the graph is *disconnected*. We refer to the maximal connected subgraphs of a disconnected network as its *components*. For instance, the graph shown in Figure 2.5(a) is connected, and the graph shown in Figure 2.5(b) is disconnected. The latter graph has two components consisting of the node sets {1, 2, 3, 4} and {5, 6}. In Section 3.4 we describe a method for determining whether a graph is connected or not, and in Exercise 3.41 we discuss a method for identifying all components of a graph.

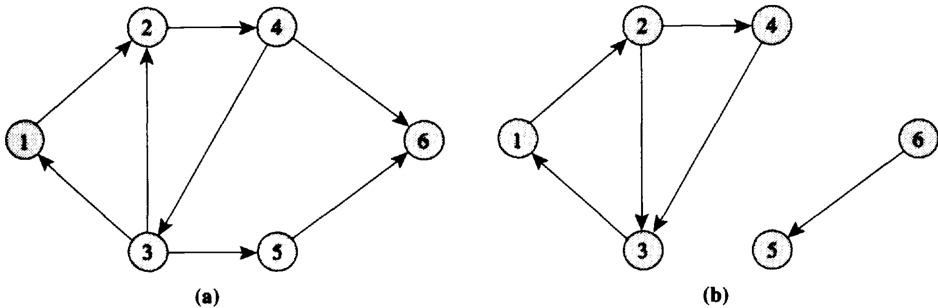

Figure 2.5 (a) Connected and (b) disconnected graphs.

Strong Connectivity: A connected graph is *strongly connected* if it contains at least one *directed* path from every node to every other node. In Figure 2.5(a) the component [see Figure 2.5(b)] defined on the node set {1, 2, 3, 4} is strongly connected; the component defined by the node set {5, 6} is not strongly connected because it contains no directed path from node 5 to node 6. In Section 3.4 we describe a method for determining whether or not a graph is strongly connected.

Cut: A *cut* is a partition of the node set N into two parts, S and $\bar{S} = N - S$. Each cut defines a set of arcs consisting of those arcs that have one endpoint in S and another endpoint in $\bar{S}$. Therefore, we refer to this set of arcs as a cut and represent it by the notation $[S, \bar{S}]$. Figure 2.6 illustrates a cut with $S = \{1, 2, 3\}$ and $\bar{S} = \{4, 5, 6, 7\}$. The set of arcs in this cut are {(2, 4), (5, 2), (5, 3), (3, 6)}.

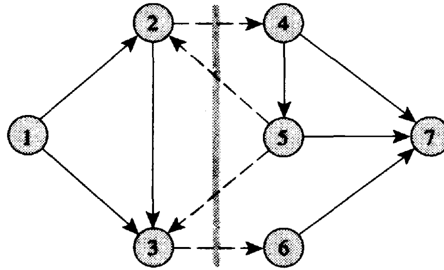

Figure 2.6 Cut.

***s–t* Cut:** An *s–t cut* is defined with respect to two distinguished nodes s and t, and is a cut $[S, \bar{S}]$ satisfying the property that $s \in S$ and $t \in \bar{S}$. For instance, if $s = 1$ and $t = 6$, the cut depicted in Figure 2.6 is an s–t cut; but if $s = 1$ and $t = 3$, this cut is not an s–t cut.

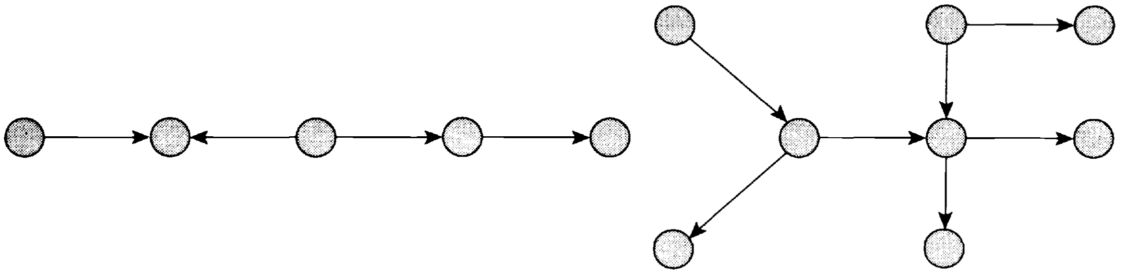

Figure 2.7 Example of two trees.

Tree. A *tree* is a connected graph that contains no cycle. Figure 2.7 shows two examples of trees.

A tree is a very important graph theoretic concept that arises in a variety of network flow algorithms studied in this book. In our subsequent discussion in later chapters, we use some of the following elementary properties of trees.

Property 2.2

(a) *A tree on n nodes contains exactly* $n - 1$ *arcs.*

(b) *A tree has at least two leaf nodes* (*i.e., nodes with degree* 1).

(c) *Every two nodes of a tree are connected by a unique path.*

Proof. See Exercise 2.13.

Forest: A graph that contains no cycle is a *forest*. Alternatively, a forest is a collection of trees. Figure 2.8 gives an example of a forest.

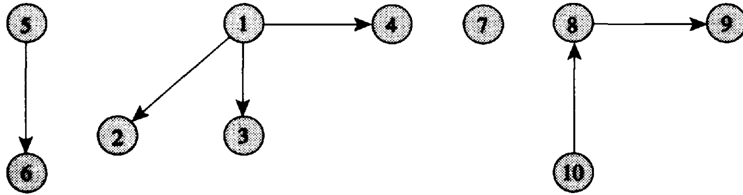

Figure 2.8 Forest.

Subtree: A connected subgraph of a tree is a *subtree.*

Rooted Tree: A rooted tree is a tree with a specially designated node, called its *root*; we regard a rooted tree as though it were hanging from its root. Figure 2.9 gives an instance of a rooted tree; in this instance, node 1 is the root node.

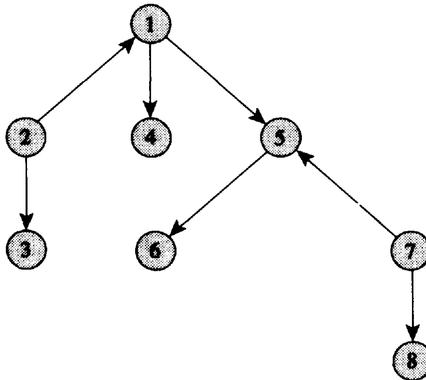

Figure 2.9 Rooted tree.

We often view the arcs in a rooted tree as defining predecessor–successor (or parent–child) relationships. For example, in Figure 2.9, node 5 is the predecessor of nodes 6 and 7, and node 1 is the predecessor of nodes 2, 4, and 5. Each node i (except the root node) has a unique predecessor, which is the next node on the unique path in the tree from that node to the root; we store the predecessor of node i using a predecessor index *pred(i).* If j = pred(i), we say that node j is the predecessor of node i and node i is a successor of node j. These predecessor indices uniquely define a rooted tree and also allow us to trace out the unique path from any node back to the root. The *descendants* of a node i consist of the node itself, its successors, successors of its successors, and so on. For example, in Figure 2.9 the node set {5, 6, 7, 8} is the set of descendants of node 5. We say that a node is an *ancestor* of all of its descendants. For example, in the same figure, node 2 is an ancestor of itself and node 3.

In this book we occasionally use two special type of rooted trees, called a *directed in-tree* and a *directed out-tree.*

Directed-Out-Tree: A tree is a *directed out-tree routed at node s* if the unique path in the tree from node s to every other node is a directed path. Figure 2.10(a) shows an instance of a directed out-tree rooted at node 1. Observe that every node in the directed out-tree (except node 1) has indegree 1.

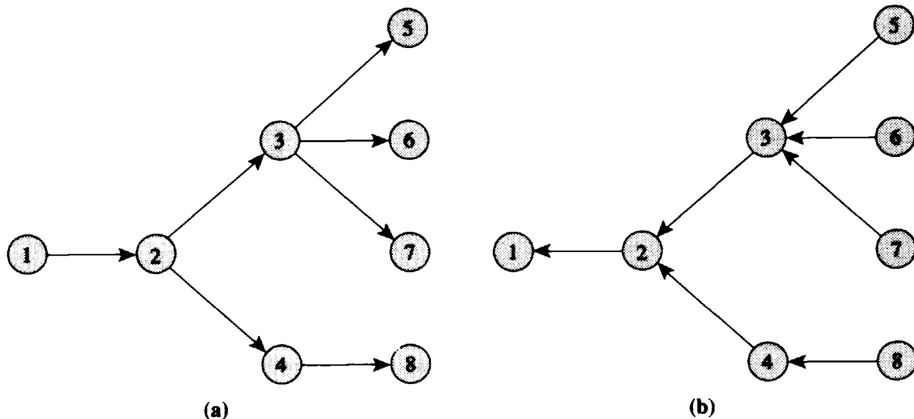

Figure 2.10 Instances of directed out-tree and directed in-tree.

Directed-In-Tree: A tree is a *directed in-tree routed at node s* if the unique path in the tree from any node to node s is a directed path. Figure 2.10(b) shows an instance of a directed in-tree rooted at node 1. Observe that every node in the directed in-tree (except node 1) has outdegree 1.

Spanning Tree: A tree T is a spanning tree of G if T is a spanning subgraph of G. Figure 2.11 shows two spanning trees of the graph shown in Figure 2.1. Every spanning tree of a connected n-node graph G has $(n - 1)$ arcs. We refer to the arcs belonging to a spanning tree T as *tree arcs* and arcs not belonging to T as *nontree arcs*.

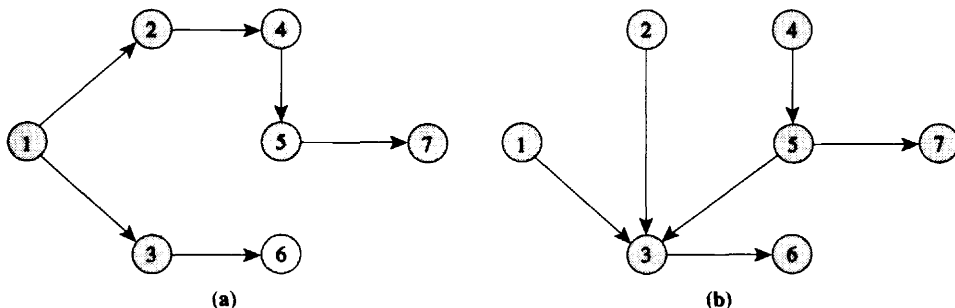

Figure 2.11 Two spanning trees of the network in Figure 2.1.

Fundamental Cycles: Let T be a spanning tree of the graph G. The addition of any nontree arc to the spanning tree T creates exactly one cycle. We refer to any such cycle as a *fundamental cycle* of G with respect to the tree T. Since the network contains $m - n + 1$ nontree arcs, it has $m - n + 1$ fundamental cycles. Observe that if we delete any arc in a fundamental cycle, we again obtain a spanning tree.

Fundamental Cuts: Let T be a spanning tree of the graph G. The deletion of any tree arc of the spanning tree T produces a disconnected graph containing two subtrees T_1 and T_2. Arcs whose endpoints belong to the different subtrees constitute a cut. We refer to any such cut as a *fundamental cut* of G with respect to the tree T. Since a spanning tree contains $n - 1$ arcs, the network has $n - 1$ fundamental cuts with respect to any tree. Observe that when we add any arc in the fundamental cut to the two subtrees T_1 and T_2, we again obtain a spanning tree.

Bipartite Graph: A graph $G = (N, A)$ is a *bipartite graph* if we can partition its node set into two subsets N_1 and N_2 so that for each arc (i, j) in A either (i) $i \in N_1$ and $j \in N_2$, or (ii) $i \in N_2$ and $j \in N_1$. Figure 2.12 gives two examples of bipartite graphs. Although it might not be immediately evident whether or not the graph in Figure 2.12(b) is bipartite, if we define $N_1 = \{1, 2, 3, 4\}$ and $N_2 = \{5, 6, 7, 8\}$, we see that it is.

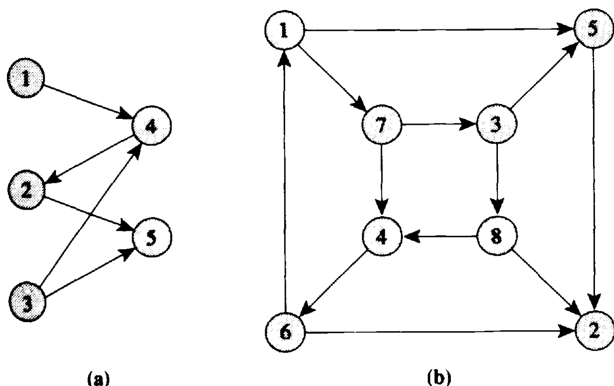

Figure 2.12 Examples of bipartite graphs.

Frequently, we wish to discover whether or not a given graph is bipartite. Fortunately, there is a very simple method for resolving this issue. We discuss this method in Exercise 3.42, which is based on the following well-known characterization of bipartite graphs.

Property 2.3. *A graph G is a bipartite graph if and only if every cycle in G contains an even number of arcs.*

Proof. See Exercise 2.21.

Definitions for undirected networks. The definitions for directed networks easily translate into those for undirected networks. An undirected arc (i, j) has two endpoints, i and j, but its tail and head nodes are undefined. If the network contains the arc (i, j), node i is adjacent to node j, and node j is adjacent to node i. The arc adjacency list (as well as the node adjacency list) is defined similarly except that arc (i, j) appears in $A(i)$ as well as $A(j)$. Consequently, $\sum_{i \in N} |A(i)| = 2m$. The degree of a node is the number of nodes adjacent to node i. Each of the graph theoretic concepts we have defined so far—walks, paths, cycles, cuts and trees—has essentially the same definition for undirected networks except that we do not distinguish between a path and a directed path, a cycle and a directed cycle, and so on.

2.3 NETWORK REPRESENTATIONS

The performance of a network algorithm depends not only on the algorithm, but also on the manner used to represent the network within a computer and the storage scheme used for maintaining and updating the intermediate results. By representing

a network more cleverly and by using improved data structures, we can often improve the running time of an algorithm. In this section we discuss some popular ways of representing a network. In representing a network, we typically need to store two types of information: (1) the network topology, that is, the network's node and arc structure; and (2) data such as costs, capacities, and supplies/demands associated with the network's nodes and arcs. As we will see, usually the scheme we use to store the network's topology will suggest a natural way for storing the associated node and arc information. In this section we describe in detail representations for directed graphs. The corresponding representations for undirected networks should be apparent to the reader. At the end of the section, however, we briefly discuss representations for undirected networks.

Node–Arc Incidence Matrix

The *node–arc incidence matrix* representation, or simply the *incidence matrix* representation, represents a network as the constraint matrix of the minimum cost flow problem that we discussed in Section 1.2. This representation stores the network as an $n \times m$ matrix $\mathcal{N}$ which contains one row for each node of the network and one column for each arc. The column corresponding to arc (i, j) has only two nonzero elements: It has a $+1$ in the row corresponding to node i and a -1 in the row corresponding to node j. Figure 2.14 gives this representation for the network shown in Figure 2.13.

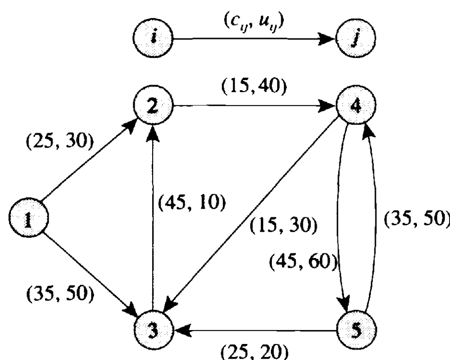

Figure 2.13 Network example.

	(1, 2)	(1, 3)	(2, 4)	(3, 2)	(4, 3)	(4, 5)	(5, 3)	(5, 4)
1	1	1	0	0	0	0	0	0
2	−1	0	1	−1	0	0	0	0
3	0	−1	0	1	−1	0	−1	0
4	0	0	−1	0	1	1	0	−1
5	0	0	0	0	0	−1	1	1

Figure 2.14 Node–arc incidence matrix of the network example.

The node–arc incidence matrix has a very special structure: Only $2m$ out of its nm entries are nonzero, all of its nonzero entries are $+1$ or -1, and each column has exactly one $+1$ and one -1. Furthermore, the number of $+1$'s in a row equals the outdegree of the corresponding node and the number of -1's in the row equals the indegree of the node.

Because the node–arc incidence matrix $\mathcal{N}$ contains so few nonzero coefficients, the incidence matrix representation of a network is not space efficient. More efficient schemes, such as those that we consider later in this section would merely keep track of the nonzero entries in the matrix. Because of its inefficiency in storing the underlying network topology, use of the node–arc incidence matrix rarely produces efficient algorithms. This representation is important, however, because it represents the constraint matrix of the minimum cost flow problem and because the node–arc incidence matrix possesses several interesting theoretical properties. We study some of these properties in Sections 11.11 and 11.12.

Node–Node Adjacency Matrix

The node–node adjacency matrix representation, or simply the adjacency matrix representation, stores the network as an $n \times n$ matrix $\mathcal{H} = \{h_{ij}\}$. The matrix has a row and a column corresponding to every node, and its ijth entry h_{ij} equals 1 if $(i, j) \in A$ and equals 0 otherwise. Figure 2.15 specifies this representation for the network shown in Figure 2.13. If we wish to store arc costs and capacities as well as the network topology, we can store this information in two additional $n \times n$ matrices, $\mathcal{C}$ and $\mathcal{U}$.

The adjacency matrix has n^2 elements, only m of which are nonzero. Consequently, this representation is space efficient only if the network is sufficiently dense; for sparse networks this representation wastes considerable space. Nevertheless, the simplicity of the adjacency representation permits us to use it to implement most network algorithms rather easily. We can determine the cost or capacity of any arc (i, j) simply by looking up the ijth element in the matrix $\mathcal{C}$ or $\mathcal{U}$. We can obtain the arcs emanating from node i by scanning row i: If the jth element in this row has a nonzero entry, (i, j) is an arc of the network. Similarly, we can obtain the arcs entering node j by scanning column j: If the ith element of this column has a nonzero entry, (i, j) is an arc of the network. These steps permit us to identify all the outgoing or incoming arcs of a node in time proportional to n. For dense networks we can usually afford to spend this time to identify the incoming or outgoing arcs, but for

	1	2	3	4	5
1	0	1	1	0	0
2	0	0	0	1	0
3	0	1	0	0	0
4	0	0	1	0	1
5	0	0	1	1	0

Figure 2.15 Node–node adjacency matrix of the network example.

sparse networks these steps might be the bottleneck operations for an algorithm. The two representations we discuss next permit us to identify the set of outgoing arcs $A(i)$ of any node in time proportional to $|A(i)|$.

Adjacency Lists

Earlier we defined the *arc adjacency list* $A(i)$ of a node i as the set of arcs emanating from that node, that is, the set of arcs $(i, j) \in A$ obtained as j ranges over the nodes of the network. Similarly, we defined the *node adjacency list* of a node i as the set of nodes j for which $(i, j) \in A$. The *adjacency list representation* stores the node adjacency list of each node as a singly linked list (we refer the reader to Appendix A for a description of singly linked lists). A linked list is a collection of cells each containing one or more fields. The node adjacency list for node i will be a linked list having $|A(i)|$ cells and each cell will correspond to an arc $(i, j) \in A$. The cell corresponding to the arc (i, j) will have as many fields as the amount of information we wish to store. One data field will store node j. We might use two other data fields to store the arc cost c_{ij} and the arc capacity u_{ij}. Each cell will contain one additional field, called the *link*, which stores a pointer to the next cell in the adjacency list. If a cell happens to be the last cell in the adjacency list, by convention we set its link to value zero.

Since we need to be able to store and access n linked lists, one for each node, we also need an array of pointers that point to the first cell in each linked list. We accomplish this objective by defining an n-dimensional array, *first*, whose element *first(i)* stores a pointer to the first cell in the adjacency list of node i. If the adjacency list of node i is empty, we set *first(i)* = 0. Figure 2.16 specifies the adjacency list representation of the network shown in Figure 2.13.

In this book we sometimes assume that whenever arc (i, j) belongs to a network, so does the reverse arc (j, i). In these situations, while updating some information about arc (i, j), we typically will also need to update information about arc (j, i). Since we will store arc (i, j) in the adjacency list of node i and arc (j, i) in the adjacency list of node j, we can carry out any operation on both arcs efficiently if we know where to find the reversal (j, i) of each arc (i, j). We can access both arcs

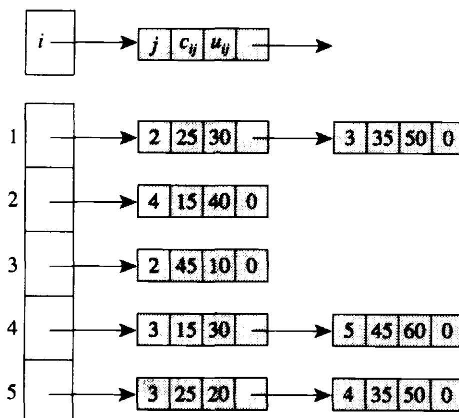

Figure 2.16 Adjacency list representation of the network example.

easily if we define an additional field, *mate*, that contains a pointer to the cell containing data for the reversal of each arc. The mate of arc (i, j) points to the cell of arc (j, i) and the mate of arc (j, i) points to the cell of arc (i, j).

Forward and Reverse Star Representations

The *forward star representation* of a network is similar to the adjacency list representation in the sense that it also stores the node adjacency list of each node. But instead of maintaining these lists as linked lists, it stores them in a single array. To develop this representation, we first associate a unique sequence number with each arc, thus defining an ordering of the arc list. We number the arcs in a specific order: first those emanating from node 1, then those emanating from node 2, and so on. We number the arcs emanating from the same node in an arbitrary fashion. We then sequentially store information about each arc in the arc list. We store the tails, heads, costs, and capacities of the arcs in four arrays: *tail*, *head*, *cost*, and *capacity*. So if arc (i, j) is arc number 20, we store the tail, head, cost, and capacity data for this arc in the array positions tail(20), head(20), cost(20), and capacity(20). We also maintain a pointer with each node i, denoted by *point(i)*, that indicates the smallest-numbered arc in the arc list that emanates from node i. [If node i has no outgoing arcs, we set point(i) equal to point(i + 1).] Therefore, the forward star representation will store the outgoing arcs of node i at positions point(i) to (point(i + 1) − 1) in the arc list. If point(i) > point(i + 1) − 1, node i has no outgoing arc. For consistency, we set point(1) = 1 and point(n + 1) = m + 1. Figure 2.17(a) specifies the forward star representation of the network given in Figure 2.13.

The forward star representation provides us with an efficient means for determining the set of outgoing arcs of any node. To determine, simultaneously, the set of incoming arcs of any node efficiently, we need an additional data structure known as the *reverse star representation*. Starting from a forward star representation, we can create a reverse star representation as follows. We examine the nodes i = 1 to n in order and sequentially store the heads, tails, costs, and capacities of the incoming arcs at node i. We also maintain a reverse pointer with each node i, denoted by *rpoint(i)*, which denotes the first position in these arrays that contains information about an incoming arc at node i. [If node i has no incoming arc, we set rpoint(i) equal to rpoint(i + 1).] For sake of consistency, we set rpoint(1) = 1 and rpoint(n + 1) = m + 1. As before, we store the incoming arcs at node i at positions rpoint(i) to (rpoint(i + 1) − 1). This data structure gives us the representation shown in Figure 2.17(b).

Observe that by storing both the forward and reverse star representations, we will maintain a significant amount of duplicate information. We can avoid this duplication by storing arc numbers in the reverse star instead of the tails, heads, costs, and capacities of the arcs. As an illustration, for our example, arc (3, 2) has arc number 4 in the forward star representation and arc (1, 2) has an arc number 1. So instead of storing the tails, costs, and capacities of the arcs, we simply store arc numbers; and once we know the arc numbers, we can always retrieve the associated information from the forward star representation. We store arc numbers in an array *trace* of size m. Figure 2.18 gives the complete *trace* array of our example.

In our discussion of the adjacency list representation, we noted that sometimes

	point
1	1
2	3
3	4
4	5
5	7
6	9

	tail	head	cost	capacity
1	1	2	25	30
2	1	3	35	50
3	2	4	15	40
4	3	2	45	10
5	4	3	15	30
6	4	5	45	60
7	5	3	25	20
8	5	4	35	50

(a)

cost	capacity	tail	head	
45	10	3	2	1
25	30	1	2	2
35	50	1	3	3
15	30	4	3	4
25	20	5	3	5
35	50	5	4	6
15	40	2	4	7
45	60	4	5	8

rpoint	
1	1
1	2
3	3
6	4
8	5
9	6

(b)

Figure 2.17 (a) Forward star and (b) reverse star representations of the network example.

while updating data for an arc (i, j), we also need to update data for its reversal (j, i). Just as we did in the adjacency list representation, we can accomplish this task by defining an array *mate* of size m, which stores the arc number of the reversal of an arc. For example, the forward star representation shown in Figure 2.17(a) assigns the arc number 6 to arc (4, 5) and assigns the arc number 8 to arc (5, 4).

	point
1	1
2	3
3	4
4	5
5	7
6	9

	tail	head	cost	capacity
1	1	2	25	30
2	1	3	35	50
3	2	4	15	40
4	3	2	45	10
5	4	3	15	30
6	4	5	45	60
7	5	3	25	20
8	5	4	35	50

trace	
4	1
1	2
2	3
5	4
7	5
8	6
3	7
6	8

rpoint	
1	1
1	2
3	3
6	4
8	5
9	6

Figure 2.18 Compact forward and reverse star representation of the network example.

Therefore, if we were using the *mate* array, we would set *mate(6)* = 8 and *mate(8)* = 6.

Comparison of Forward Star and Adjacency List Representations

The major advantage of the forward star representation is its space efficiency. It requires less storage than does the adjacency list representation. In addition, it is much easier to implement in languages such as FORTRAN that have no natural provisions for using linked lists. The major advantage of adjacency list representation is its ease of implementation in languages such as Pascal or C that are able to manipulate linked lists efficiently. Further, using an adjacency list representation, we can add or delete arcs (as well as nodes) in constant time. On the other hand, in the forward star representation these steps require time proportional to m, which can be too time consuming.

Storing Parallel Arcs

In this book we assume that the network does not contain parallel arcs; that is, no two arcs have the same tail and head nodes. By allowing parallel arcs, we encounter some notational difficulties, since (i, j) will not specify the arc uniquely. For networks with parallel arcs, we need more complex notation to specify arcs, arc costs, and capacities. This difficulty is merely notational, however, and poses no problems computationally: both the adjacency list representation and the forward star representation data structures are capable of handling parallel arcs. If a node i has two

outgoing arcs with the same head node but (possibly) different costs and capacities, the linked list of node i will contain two cells corresponding to these two arcs. Similarly, the forward star representation allows several entries with the same tail and head nodes but different costs and capacities.

Representing Undirected Networks

We can represent undirected networks using the same representations we have just described for directed networks. However, we must remember one fact: Whenever arc (i, j) belongs to an undirected network, we need to include both of the pairs (i, j) and (j, i) in the representations we have discussed. Consequently, we will store each arc (i, j) of an undirected network twice in the adjacency lists, once in the list for node i and once in the list for node j. Some other obvious modifications are needed. For example, in the node–arc incidence matrix representation, the column corresponding to arc (i, j) will have $+1$ in both rows i and j. The node–node adjacency matrix will have $+1$ in position h_{ij} and h_{ji} for every arc $(i, j) \in A$. Since this matrix will be symmetric, we might as well store half of the matrix. In the adjacency list representation, the arc (i, j) will be present in the linked lists of both nodes i and j. Consequently, whenever we update information for one arc, we must update it for the other arc as well. We can accomplish this task by storing for each arc the address of its other occurrence in an additional mate array. The forward star representation requires this additional storage as well. Finally, observe that undirected networks do not require the reverse star representation.

2.4 NETWORK TRANSFORMATIONS

Frequently, we require network transformations to simplify a network, to show equivalences between different network problems, or to state a network problem in a standard form required by a computer code. In this section, we describe some of these important transformations. In describing these transformations, we assume that the network problem is a minimum cost flow problem as formulated in Section 1.2. Needless to say, these transformations also apply to special cases of the minimum cost flow problem, such as the shortest path, maximum flow, and assignment problems, wherever the transformations are appropriate. We first recall the formulation of the minimum cost flow problem for convenience in discussing the network transformations.

$$\text{Minimize} \sum_{(i,j)\in A} c_{ij}x_{ij} \tag{2.1a}$$

subject to

$$\sum_{\{j:(i,j)\in A\}} x_{ij} - \sum_{\{j:(j,i)\in A\}} x_{ji} = b(i) \qquad \text{for all } i \in N, \tag{2.1b}$$

$$l_{ij} \le x_{ij} \le u_{ij} \qquad \text{for all } (i, j) \in A. \tag{2.1c}$$

Undirected Arcs to Directed Arcs

Sometimes minimum cost flow problems contain undirected arcs. An undirected arc (i, j) with cost $c_{ij} \geq 0$ and capacity u_{ij} permits flow from node i to node j and also from node j to node i; a unit of flow in either direction costs c_{ij}, and the total flow (i.e., from node i to node j plus from node j to node i) has an upper bound u_{ij}. That is, the undirected model has the constraint $x_{ij} + x_{ji} \leq u_{ij}$ and the term $c_{ij}x_{ij} + c_{ij}x_{ji}$ in the objective function. Since the cost $c_{ij} \geq 0$, in some optimal solution one of x_{ij} and x_{ji} will be zero. We refer to any such solution as non-overlapping.

For notational convenience, in this discussion we refer to the undirected arc (i, j) as $\{i, j\}$. We assume (with some loss of generality) that the arc flow in either direction on arc $\{i, j\}$ has a lower bound of value 0; our transformation is not valid if the arc flow has a nonzero lower bound or the arc cost c_{ij} is negative (why?). To transform the undirected case to the directed case, we replace each undirected arc $\{i, j\}$ by two directed arcs, (i, j) and (j, i), both with cost c_{ij} and capacity u_{ij}. To establish the correctness of this transformation, we show that every non-overlapping flow in the original network has an associated flow in the transformed network with the same cost, and vice versa. If the undirected arc $\{i, j\}$ carries α units of flow from node i to node j, in the transformed network $x_{ij} = \alpha$ and $x_{ji} = 0$. If the undirected arc $\{i, j\}$ carries α units of flow from node j to node i, in the transformed network $x_{ij} = 0$ and $x_{ji} = \alpha$. Conversely, if x_{ij} and x_{ji} are the flows on arcs (i, j) and (j, i) in the directed network, $x_{ij} - x_{ji}$ or $x_{ji} - x_{ij}$ is the associated flow on arc $\{i, j\}$ in the undirected network, whichever is positive. If $x_{ij} - x_{ji}$ is positive, the flow from node i to node j on arc $\{i, j\}$ equals this amount. If $x_{ji} - x_{ij}$ is positive, the flow from node j to node i on arc $\{i, j\}$ equals $x_{ji} - x_{ij}$. In either case, the flow in the opposite direction is zero. If $x_{ji} - x_{ij}$ is zero, the flow on arc $\{i, j\}$ is 0.

Removing Nonzero Lower Bounds

If an arc (i, j) has a nonzero lower bound l_{ij} on the arc flow x_{ij}, we replace x_{ij} by $x'_{ij} + l_{ij}$ in the problem formulation. The flow bound constraint then becomes $l_{ij} \leq x'_{ij} + l_{ij} \leq u_{ij}$, or $0 \leq x'_{ij} \leq (u_{ij} - l_{ij})$. Making this substitution in the mass balance constraints decreases $b(i)$ by l_{ij} units and increases $b(j)$ by l_{ij} units [recall from Section 1.2 that the flow variable x_{ij} appears in the mass balance constraint (2.1b) of only nodes i and j]. This substitution changes the objective function value by a constant that we can record separately and then ignore when solving the problem. Figure 2.19 illustrates this transformation graphically. We can view this transformation as a two-step flow process: We begin by sending l_{ij} units of flow on arc (i, j), which decreases $b(i)$ by l_{ij} units and increases $b(j)$ by l_{ij} units, and then we measure (by the variable x'_{ij}) the incremental flow on the arc beyond the flow value l_{ij}.

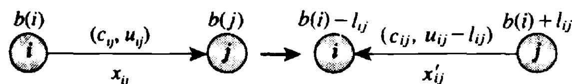

Figure 2.19 Removing nonzero lower bounds.

Arc Reversal

The arc reversal transformation is typically used to remove arcs with negative costs. Let u_{ij} denote the capacity of the arc (i, j) or an upper bound on the arc's flow if the arc is uncapacitated. In this transformation we replace the variable x_{ij} by $u_{ij} - x_{ji}$. Doing so replaces the arc (i, j), which has an associated cost c_{ij}, by the arc (j, i) with an associated cost $-c_{ij}$. As shown in Figure 2.20, the transformation has the following network interpretation. We first send u_{ij} units of flow on the arc (which decreases $b(i)$ by u_{ij} units and increases $b(j)$ by u_{ij} units) and then we replace arc (i, j) by arc (j, i) with cost $-c_{ij}$. The new flow x_{ji} measures the amount of flow we "remove" from the "full capacity" flow of u_{ij}.

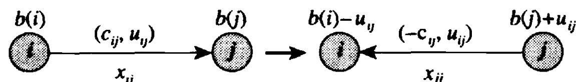

Figure 2.20 Arc reversal transformation.

Removing Arc Capacities

If an arc (i, j) has a positive capacity u_{ij}, we can remove the capacity, making the arc *uncapacitated*, by using the following idea: We introduce an additional node so that the capacity constraint on arc (i, j) becomes the mass balance constraint of the new node. Suppose that we introduce a slack variable $s_{ij} \geq 0$, and write the capacity constraint $x_{ij} \leq u_{ij}$ in an equality form as $x_{ij} + s_{ij} = u_{ij}$. Multiplying both sides of the equality by -1, we obtain

$$-x_{ij} - s_{ij} = -u_{ij} \tag{2.2}$$

We now treat constraint (2.2) as the mass balance constraint of an additional node k. Observe that the flow variable x_{ij} now appears in three mass balance constraints and s_{ij} in only one. By subtracting (2.2) from the mass balance constraint of node j (which contains the flow variable x_{ij} with a negative sign), we assure that each of x_{ij} and s_{ij} appears in exactly two constraints—in one with a positive sign and in the other with a negative sign. These algebraic manipulations correspond to the network transformation shown in Figure 2.21.

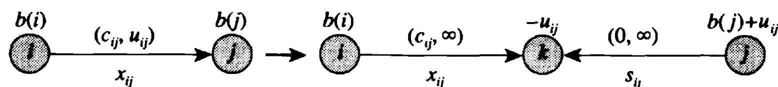

Figure 2.21 Transformation for removing an arc capacity.

To see the relationships between the flows in the original and transformed networks, we make the following observations. If x_{ij} is the flow on arc (i, j) in the original network, the corresponding flow in the transformed network is $x'_{ik} = x_{ij}$ and $x'_{jk} = u_{ij} - x_{ij}$. Notice that both the flows x and x' have the same cost. Similarly, a flow x'_{ik}, x'_{jk} in the transformed network yields a flow $x_{ij} = x'_{ik}$ of the same cost in the original network. Furthermore, since $x'_{ik} + x'_{jk} = u_{ij}$ and x'_{ik} and x'_{jk} are both nonnegative, $x_{ij} = x'_{ik} \leq u_{ij}$. Therefore, the flow x_{ij} satisfies the arc capacity, and the transformation does correctly model arc capacities.

Suppose that every arc in a given network $G = (N, A)$ is capacitated. If we apply the preceding transformation to every arc, we obtain a bipartite uncapacitated network G' (see Figure 2.22 for an illustration). In this network (1) each node i on the left corresponds to a node $i \in N$ of the original network and has a supply equal to $b(i) + \sum_{\{k:(k,i) \in A\}} u_{ki}$, and (2) each node i–j on the right corresponds to an arc $(i, j) \in A$ in the original network and has a demand equal to u_{ij}; this node has exactly two incoming arcs, originating at nodes i and j from the left. Consequently, the transformed network has $(n + m)$ nodes and $2m$ arcs.

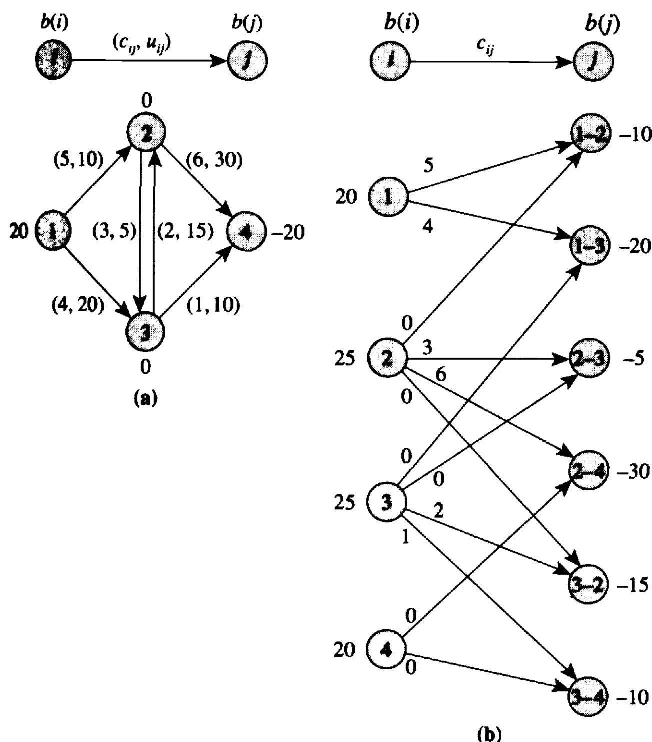

Figure 2.22 Transformation for removing arc capacities: (a) original network; (b) transformed network with uncapacitated arcs.

At first glance we might be tempted to believe that this technique for removing arc capacities would be unattractive computationally since the transformation substantially increases the number of nodes in the network. However, on most occasions the original and transformed networks have algorithms with the same complexity, because the transformed network possesses a special structure that permits us to design more efficient algorithms.

Node Splitting

The node splitting transformation splits each node i into two nodes i' and i'' corresponding to the node's *output* and *input* functions. This transformation replaces each original arc (i, j) by an arc (i', j'') of the same cost and capacity. It also adds an arc (i'', i') of zero cost and with infinite capacity for each i. The input side of

node i (i.e., node i'') receives all the node's inflow, the output side (i.e., node i') sends all the node's outflow, and the additional arc (i'', i') carries flow from the input side to the output side. Figure 2.23 illustrates the resulting network when we carry out the node splitting transformation for all the nodes of a network. We define the supplies/demands of nodes in the transformed network in accordance with the following three cases:

1. If $b(i) > 0$, then $b(i'') = b(i)$ and $b(i') = 0$.
2. If $b(i) < 0$, then $b(i'') = 0$ and $b(i') = b(i)$.
3. If $b(i) = 0$, then $b(i') = b(i'') = 0$.

It is easy to show a one-to-one correspondence between a flow in the original network and the corresponding flow in the transformed network; moreover, the flows in both networks have the same cost.

The node splitting transformation permits us to model numerous applications in a variety of practical problem domains, yet maintain the form of the network flow model that we introduced in Section 1.2. For example, we can use the transformation to handle situations in which nodes as well as arcs have associated capacities and costs. In these situatioı , each flow unit passing through a node i incurs a cost c_i and the maximum flow that can pass through the node is u_i. We can reduce this problem to the standard "arc flow" form of the network flow problem by performing the node splitting transformation and letting c_i and u_i be the cost and capacity of arc

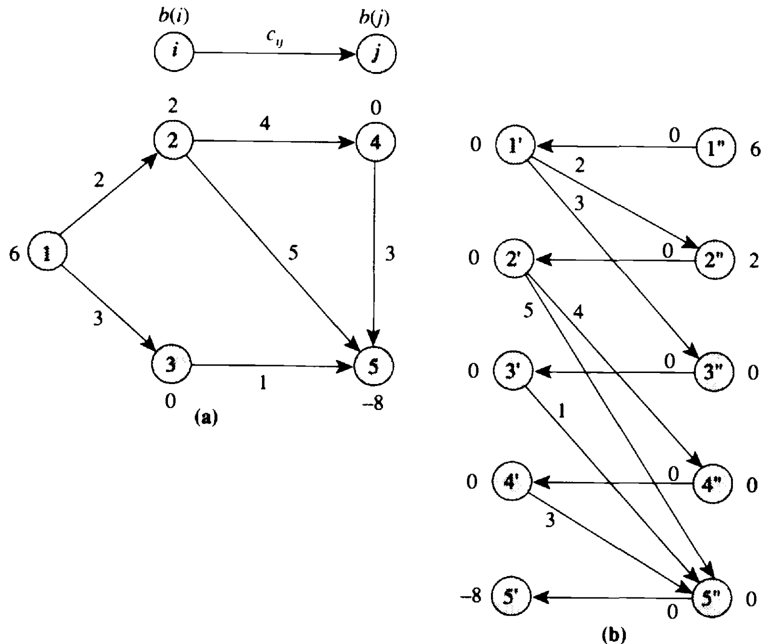

Figure 2.23 Node splitting transformation: (a) original network; (b) transformed network.

(i'', i'). We shall study more applications of the node splitting transformation in Sections 6.6 and 12.7 and in several exercises.

Working with Reduced Costs

In many of the network flow algorithms discussed in this book, we measure the cost of an arc relative to "imputed" costs associated with its incident nodes. These imputed costs typically are intermediate data that we compute within the context of an algorithm. Suppose that we associate with each node $i \in N$ a number $\pi(i)$, which we refer to as the *potential* of that node. With respect to the node potentials $\pi = (\pi(1), \pi(2), \ldots, \pi(n))$, we define the *reduced cost* c_{ij}^{π} of an arc (i, j) as

$$c_{ij}^{\pi} = c_{ij} - \pi(i) + \pi(j). \tag{2.3}$$

In many algorithms discussed later, we often work with reduced costs c_{ij}^{π} instead of the actual costs c_{ij}. Consequently, it is important to understand the relationship between the objective functions $z(\pi) = \sum_{(i,j)\in A} c_{ij}^{\pi} x_{ij}$ and $z(0) = \sum_{(i,j)\in A} c_{ij} x_{ij}$. Suppose, initially, that $\pi = 0$ and we then increase the node potential of node k to $\pi(k)$. The definition (2.3) of reduced costs implies that this change reduces the reduced cost of each unit of flow leaving node k by $\pi(k)$ and increases the reduced cost of each flow unit entering node k by $\pi(k)$. Thus the total decrease in the objective function equals $\pi(k)$ times the outflow of node k minus the inflow of node k. By definition (see Section 1.2), the outflow minus inflow equals the supply/demand of the node. Consequently, increasing the potential of node k by $\pi(k)$ decreases the objective function value by $\pi(k)b(k)$ units. Repeating this argument iteratively for each node establishes that

$$z(0) - z(\pi) = \sum_{i\in N} \pi(i)b(i) = \pi b.$$

For a given node potential π, πb is a constant. Therefore, a flow that minimizes $z(\pi)$ also minimizes $z(0)$. We formalize this result for easy future reference.

Property 2.4. *Minimum cost flow problems with arc costs c_{ij} or c_{ij}^{π} have the same optimal solutions. Moreover, $z(\pi) = z(0) - \pi b$.*

We next study the effect of working with reduced costs on the cost of cycles and paths. Let W be a directed cycle in G. Then

$$\begin{aligned}\sum_{(i,j)\in W} c_{ij}^{\pi} &= \sum_{(i,j)\in W} (c_{ij} - \pi(i) + \pi(j)), \\ &= \sum_{(i,j)\in W} c_{ij} + \sum_{(i,j)\in W} (\pi(j) - \pi(i)), \\ &= \sum_{(i,j)\in W} c_{ij}.\end{aligned}$$

The last equality follows from the fact that for any directed cycle W, the expression $\sum_{(i,j)\in W} (\pi(j) - \pi(i))$ sums to zero because for each node i in the cycle W, $\pi(i)$ occurs once with a positive sign and once with a negative sign. Similarly, if P

is a directed path from node k to node l, then

$$\begin{aligned}\sum_{(i,j)\in P} c_{ij}^{\pi} &= \sum_{(i,j)\in P} (c_{ij} - \pi(i) + \pi(j)), \\ &= \sum_{(i,j)\in P} c_{ij} - \sum_{(i,j)\in P} (\pi(i) - \pi(j)), \\ &= \sum_{(i,j)\in P} c_{ij} - \pi(k) + \pi(l),\end{aligned}$$

because all $\pi(\cdot)$ corresponding to the nodes in the path, other than the terminal nodes k and l, cancel each other in the expression $\sum_{(i,j)\in P} (\pi(i) - \pi(j))$. We record these results for future reference.

Property 2.5

(a) *For any directed cycle W and for any node potentials π, $\sum_{(i,j)\in W} c_{ij}^{\pi} = \sum_{(i,j)\in W} c_{ij}$.*

(b) *For any directed path P from node k to node l and for any node potentials π, $\sum_{(i,j)\in P} c_{ij}^{\pi} = \sum_{(i,j)\in P} c_{ij} - \pi(k) + \pi(l)$.*

Working with Residual Networks

In designing, developing, and implementing network flow algorithms, it is often convenient to measure flow not in absolute terms, but rather in terms of incremental flow about some given feasible solution—typically, the solution at some intermediate point in an algorithm. Doing so leads us to define a new, ancillary network, known as the *residual network*, that functions as a "remaining flow network" for carrying the incremental flow. We show that formulations of the problem in the original network and in the residual network are equivalent in the sense that they give a one-to-one correspondence between feasible solutions to the two problems that preserves the value of the cost of solutions.

The concept of residual network is based on the following intuitive idea. Suppose that arc (i, j) carries x_{ij}° units of flow. Then we can send an additional $u_{ij} - x_{ij}^{\circ}$ units of flow from node i to node j along arc (i, j). Also notice that we can send up to x_{ij}° units of flow from node j to node i over the arc (i, j), which amounts to canceling the existing flow on the arc. Whereas sending a unit flow from node i to node j on arc (i, j) increases the flow cost by c_{ij} units, sending flow from node j to node i on the same arc decreases the flow cost by c_{ij} units (since we are saving the cost that we used to incur in sending the flow from node i to node j).

Using these ideas, we define the residual network with respect to a given flow x° as follows. We replace each arc (i, j) in the original network by two arcs, (i, j) and (j, i): the arc (i, j) has cost c_{ij} and *residual capacity* $r_{ij} = u_{ij} - x_{ij}^{\circ}$, and the arc (j, i) has cost $-c_{ij}$ and *residual capacity* $r_{ji} = x_{ij}^{\circ}$ (see Figure 2.24). The residual network consists of only the arcs with a positive residual capacity. We use the notation $G(x^{\circ})$ to represent the residual network corresponding to the flow x°.

In general, the concept of residual network poses some notational difficulties. If for some pair i and j of nodes, the network G contains both the arcs (i, j) and

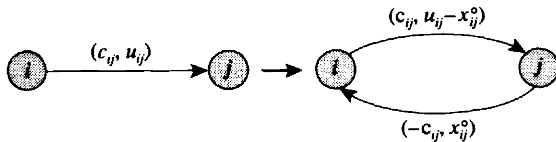

Figure 2.24 Constructing the residual network $G(x^0)$.

(j, i), the residual network may contain two (parallel) arcs from node i to node j with different costs and residual capacities, and/or two (parallel) arcs from node j to node i with different costs and residual capacities. In these instances, any reference to arc (i, j) will be ambiguous and will not define a unique arc cost and residual capacity. We can overcome this difficulty by assuming that for any pair of nodes i and j, the graph G does not contain both arc (i, j) and arc (j, i); then the residual network will contain no parallel arcs. We might note that this assumption is merely a notational convenience; it does not impose any loss of generality, because by suitable transformations we can always define a network that is equivalent to any given network and that will satisfy this assumption (see Exercise 2.47). However, we need not actually make this transformation in practice, since the network representations described in Section 2.3 are capable of handling parallel arcs.

We note further that although the construction and use of the residual network poses some notational difficulties for the general minimum cost flow problem, the difficulties might not arise for some special cases. In particular, for the maximum flow problem, the parallel arcs have the same cost (of zero), so we can merge both of the parallel arcs into a single arc and set its residual capacity equal to the sum of the residual capacities of the two arcs. For this reason, in our discussion of the maximum flow problem, we will permit the underlying network to contain arcs joining any two nodes in both directions.

We now show that every flow x in the network G corresponds to a flow x' in the residual network $G(x^{\circ})$. We define the flow $x' \geq 0$ as follows:

$$x'_{ij} - x'_{ji} = x_{ij} - x^{\circ}_{ij}, \tag{2.4}$$

and

$$x'_{ij}x'_{ji} = 0. \tag{2.5}$$

The condition (2.5) implies that x'_{ij} and x'_{ji} cannot both be positive at the same time. If $x_{ij} \geq x^{\circ}_{ij}$, we set $x'_{ij} = (x_{ij} - x^{\circ}_{ij})$ and $x'_{ji} = 0$. Notice that if $x_{ij} \leq u_{ij}$, then $x'_{ij} \leq u_{ij} - x^{\circ}_{ij} = r_{ij}$. Therefore, the flow x'_{ij} satisfies the flow bound constraints. Similarly, if $x_{ij} < x^{\circ}_{ij}$, we set $x'_{ij} = 0$ and $x'_{ji} = x^{\circ}_{ij} - x_{ij}$. Observe that $0 \leq x'_{ji} \leq x^{\circ}_{ij} = r_{ji}$, so the flow x'_{ji} also satisfies the flow bound constraints. These observations show that if x is a feasible flow in G, its corresponding flow x' is a feasible flow in $G(x^{\circ})$.

We next establish a relationship between the cost of a flow x in G and the cost of the corresponding flow x' in $G(x^{\circ})$. Let c' denote the arc costs in the residual network. Then for every arc $(i, j) \in A$, $c'_{ij} = c_{ij}$ and $c'_{ji} = -c_{ij}$. For a flow x_{ij} on arc (i, j) in the original network G, the cost of flow on the pair of arcs (i, j) and (j, i) in the residual network $G(x^{\circ})$ is $c'_{ij}x'_{ij} + c'_{ji}x'_{ji} = c'_{ij}(x'_{ij} - x'_{ji}) = c_{ij}x_{ij} - c_{ij}x^{\circ}_{ij}$; the last equality follows from (2.4). We have thus shown that

$$c'x' = cx - cx^{\circ}.$$

Similarly, we can show the converse result that if x' is a feasible flow in the residual network $G(x^\circ)$, the solution given by $x_{ij} = (x'_{ij} - x'_{ji}) + x^\circ_{ij}$ is a feasible flow in G. Moreover, the costs of these two flows is related by the equality $cx = c'x' + cx^\circ$. We ask the reader to prove these results in Exercise 2.48. We summarize the preceding discussion as the following property.

Property 2.6. *A flow x is a feasible flow in the network G if and only if its corresponding flow x', defined by $x'_{ij} - x'_{ji} = x_{ij} - x^\circ_{ij}$ and $x'_{ij}x'_{ji} = 0$, is feasible in the residual network $G(x^\circ)$. Furthermore, $cx = c'x' + cx^\circ$.*

One important consequence of Property 2.6 is the flexibility it provides us. Instead of working with the original network G, we can work with the residual network $G(x^\circ)$ for some x°: Once we have determined an optimal solution in the residual network, we can immediately convert it into an optimal solution in the original network. Many of the maximum flow and minimum cost flow algorithms discussed in the subsequent chapters use this result.

2.5 SUMMARY

In this chapter we brought together many basic definitions of network flows and graph theory and presented basic notation that we will use throughout this book. We defined several common graph theoretic terms, including adjacency lists, walks, paths, cycles, cuts, and trees. We also defined acyclic and bipartite networks.

Although networks are often geometric entities, optimization algorithms require computer representations of them. The following four representations are the most common: (1) the node–arc incidence matrix, (2) the node–node adjacency matrix, (3) adjacency lists, and (4) forward and reverse star representations. Figure 2.25 summarizes the basic features of these representations.

Network representations	Storage space	Features
Node–arc incidence matrix	nm	1. Space inefficient 2. Too expensive to manipulate 3. Important because it represents the constraint matrix of the minimum cost flow problem
Node–node adjacency matrix	kn^2 for some constant k	1. Suited for dense networks 2. Easy to implement
Adjacency list	$k_1n + k_2m$ for some constants k_1 and k_2	1. Space efficient 2. Efficient to manipulate 3. Suited for dense as well as sparse networks
Forward and reverse star	$k_3n + k_4m$ for some constants k_3 and k_4	1. Space efficient 2. Efficient to manipulate 3. Suited for dense as well as sparse networks

Figure 2.25 Comparison of various network representations.

The field of network flows is replete with transformations that allow us to transform one problem to another, often transforming a problem that appears to include new complexities into a simplified "standard" format. In this chapter we described some of the most common transformations: (1) transforming undirected networks to directed networks, (2) removing nonzero lower flow bounds (which permits us to assume, without any loss of generality, that flow problems have zero lower bounds on arc flows), (3) performing arc reversals (which often permits us to assume, without any loss of generality, that arcs have nonnegative arc costs), (4) removing arc capacities (which allows us to transform capacitated networks to uncapacitated networks), (5) splitting nodes (which permits us to transform networks with constraints and/or cost associated with "node flows" into our formulation with all data and constraints imposed upon arc flows), and (6) replacing costs with reduced costs (which permits us to alter the cost coefficients, yet retain the same optimal solutions).

The last transformation we studied in this chapter permits us to work with residual networks, which is a concept of critical importance in the development of maximum flow and minimum cost flow algorithms. With respect to an existing flow x, the residual network $G(x)$ represents the capacity and cost information in the network for carrying incremental flows on the arcs. As our discussion has shown, working with residual networks is equivalent to working with the original network.

REFERENCE NOTES

The applied mathematics, computer science, engineering, and operations research communities have developed no standard notation of graph concepts; different researchers and authors use different names to denote the same object (e.g., some authors refer to nodes as vertices or points). The notation and definitions we have discussed in Section 2.2 and adopted throughout this book are among the most popular in the literature. The network representations and transformation that we described in Sections 2.3 and 2.4 are part of the folklore; it is difficult to pinpoint their origins. The books by Aho, Hopcroft, and Ullman [1974], Gondran and Minoux [1984], and Cormen, Leiserson, and Rivest [1990] contain additional information on network representations. The classic book by Ford and Fulkerson [1962] discusses many transformations of network flow problems.

EXERCISES

Note: If any of the following exercises does not state whether a graph is undirected or directed, assume either option, whichever is more convenient.

2.1 Consider the two graphs shown in Figure 2.26.

- **(a)** List the indegree and outdegree of every node.
- **(b)** Give the node adjacency list of each node. (Arrange each list in the increasing order of node numbers.)
- **(c)** Specify a directed walk containing six arcs. Also, specify a walk containing eight arcs.
- **(d)** Specify a cycle containing nine arcs and a directed cycle containing seven arcs.

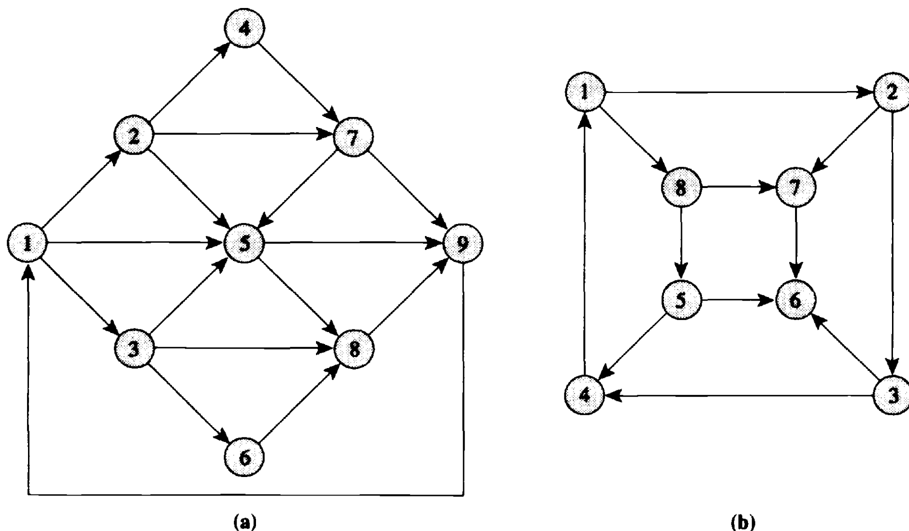

Figure 2.26 Example networks for Exercises 2.1 to 2.4.

2.2. Specify a spanning tree of the graph in Figure 2.26(a) with six leaves. Specify a cut of the graph in Figure 2.26(a) containing six arcs.

2.3. For the graphs shown in Figure 2.26, answer the following questions.
 (a) Are the graphs acyclic?
 (b) Are the graphs bipartite?
 (c) Are the graphs strongly connected?

2.4. Consider the graphs shown in Figure 2.26.
 (a) Do the graphs contain a directed in-tree for some root node?
 (b) Do the graphs contain a directed out-tree for some root node?
 (c) In Figure 2.26(a), list all fundamental cycles with respect to the following spanning tree $T = \{(1, 5), (1, 3), (2, 5), (4, 7), (7, 5), (7, 9), (5, 8), (6, 8)\}$.
 (d) For the spanning tree given in part (c), list all fundamental cuts. Which of these are the s–t cuts when $s = 1$ and $t = 9$?

2.5. (a) Construct a directed strongly connected graph with five nodes and five arcs.
 (b) Construct a directed bipartite graph with six nodes and nine arcs.
 (c) Construct an acyclic directed graph with five nodes and ten arcs.

2.6. **Bridges of Königsberg.** The first paper on graph theory was written by Leonhard Euler in 1736. In this paper, he started with the following mathematical puzzle: The city of Königsburg has seven bridges, arranged as shown in Figure 2.27. Is it possible to start at some place in the city, cross every bridge exactly once, and return to the starting place? Either specify such a tour or prove that it is impossible to do so.

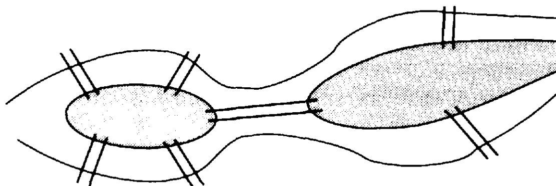

Figure 2.27 Bridges of Königsberg.

2.7. At the beginning of a dinner party, several participants shake hands with each other. Show that the participants that shook hands an odd number of times must be even in number.

2.8. Show that in a directed strongly connected graph containing more than one node, no node can have a zero indegree or a zero outdegree.

2.9. Suppose that every node in a directed graph has a positive indegree. Show that the graph must contain a directed cycle.

2.10. Show that a graph G remains connected even after deleting an arc (i, j) if and only if arc (i, j) belongs to some cycle in G.

2.11. Show that an undirected graph $G = (N, A)$ is connected if and only if for every partition of N into subsets N_1 and N_2, some arc has one endpoint in N_1 and the other endpoint in N_2.

2.12. Let $d_{\min}$ denote the minimum degree of a node in an undirected graph. Show that the graph contains a path containing at least $d_{\min}$ arcs.

2.13. Prove the following properties of trees.
- **(a)** A tree on n nodes contains exactly $(n - 1)$ arcs.
- **(b)** A tree has at least two leaf nodes (i.e., nodes with degree 1).
- **(c)** Every two nodes of a tree are connected by a unique path.

2.14. Show that every tree is a bipartite graph.

2.15. Show that a forest consisting of k components has $m = n - k$ arcs.

2.16. Let $d_{\max}$ denote the maximum degree of a node in a tree. Show that the tree contains at least $d_{\max}$ nodes of degree 1. (*Hint*: Use the fact that the sum of the degrees of all nodes in a tree is $2m = 2n - 2$.)

2.17. Let Q be any cut of a connected graph and T be any spanning tree. Show that $Q \cap T$ is nonempty.

2.18. Show that a closed directed walk containing an odd number of arcs contains a directed cycle having an odd number of arcs. Is it true that a closed directed walk containing an even number of arcs also contains a directed cycle having an even number of arcs?

2.19. Show that any cycle of a graph G contains an even number of arcs (possibly zero) in common with any cut of G.

2.20. Let $d_{\min}$ denote the minimum degree of a node in an undirected graph G. Show that if $d_{\min} \geq 2$, then G must contain a cycle.

2.21.
- **(a)** Show that in a bipartite graph every cycle contains an even number of arcs.
- **(b)** Show that a (connected) graph, in which every cycle contains an even number of arcs, must be bipartite. Conclude that a graph is bipartite if and only if every cycle has an even number of arcs.

2.22. The *k-color problem* on an undirected graph $G = (N, A)$ is defined as follows: Color all the nodes in N using at most k colors so that for every arc $(i, j) \in A$, nodes i and j have a different color.
- **(a)** Given a world map, we want to color countries using at most k colors so that the countries having common boundaries have a different color. Show how to formulate this problem as a k-color problem.
- **(b)** Show that a graph is bipartite if and only if it is *2-colorable* (i.e., can be colored using at most two colors).

2.23. Two undirected graphs $G = (N, A)$ and $G' = (N', A')$ are said to be *isomorphic* if we can number the nodes of the graph G so that G becomes identical to G'. Equivalently, G is isomorphic to G' if some one-to-one function f maps N onto N' so that (i, j) is an arc in A if and only if $(f(i), f(j))$ is an arc in A'. Give several necessary conditions for two undirected graphs to be isomorphic. (*Hint*: For example, they must have the same number of nodes and arcs.)

2.24.
- **(a)** List all nonisomorphic trees having four nodes.
- **(b)** List all nonisomorphic trees having five nodes. (*Hint:* There are three such trees.)

2.25. For any undirected graph $G = (N, A)$, we define its *complement* $G^c = (N, A^c)$ as follows: If $(i, j) \in A$, then $(i, j) \notin A^c$, and if $(i, j) \notin A$, then $(i, j) \in A^c$. Show that if the graph G is disconnected, its complement G^c is connected.

2.26. Let $G = (N, A)$ be an undirected graph. We refer to a subset $N_1 \subseteq N$ as *independent* if no two nodes in N_1 are adjacent. Let $\beta(G)$ denote the maximum cardinality of any independent set of G. We refer to a subset $N_2 \subseteq N$ as a *node cover* if each arc in A has at least one of its endpoints in N_2. Let $\eta(G)$ denote the minimum cardinality of any node cover G. Show that $\beta(G) + \eta(G) = n$. (*Hint:* Show that the complement of an independent set is a node cover.)

2.27. **Problem of queens.** Consider the problem of determining the maximum number of queens that can be placed on a chessboard so that none of the queens can be taken by another. Show how to transform this problem into an independent set problem defined in Exercise 2.26.

2.28. Consider a directed graph $G = (N, A)$. For any subset $S \subseteq N$, let *neighbor(S)* denote the set of neighbors of S [i.e., neighbor$(S) = \{j \in N$: for some $i \in S$, $(i, j) \in A$ and $j \notin S\}$]. Show that G is strongly connected if and only if for every proper nonempty subset $S \subset N$, neighbor$(S) \neq \emptyset$.

2.29. A subset $N_1 \subseteq N$ of nodes in an undirected graph $G = (N, A)$ is said to be a *clique* if every pair of nodes in N_1 is connected by an arc. Show that the set N_1 is a clique in G if and only if N_1 is independent in its complement G^c.

2.30. Specify the node–arc incidence matrix and the node–node adjacency matrix for the graph shown in Figure 2.28.

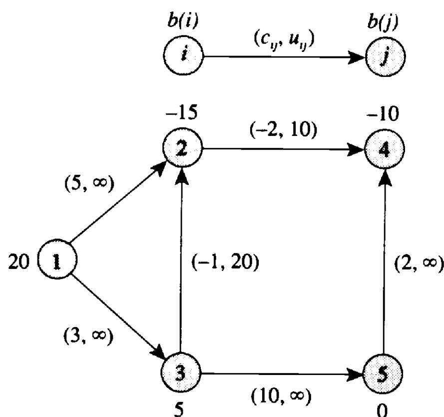

Figure 2.28 Network example.

2.31. **(a)** Specify the forward star representation of the graph shown in Figure 2.28.
(b) Specify the forward and reverse star representations of the graph shown in Figure 2.28.

2.32. Let $\mathcal{N}$ denote the node–arc incidence matrix of an undirected graph and let $\mathcal{N}^T$ denote its transpose. Let "·" denote the operation of taking a product of two matrices. Show how to interpret the diagonal elements of $\mathcal{N} \cdot \mathcal{N}^T$?

2.33. Let $\mathcal{H}$ denote the node–node adjacency matrix of a directed network, and let $\mathcal{N}$ denote the node–arc incidence matrix of this network. Can $\mathcal{H} = \mathcal{N} \cdot \mathcal{N}^T$?

2.34. Let $\mathcal{H}$ be the node–node adjacency matrix of a directed graph $G = (N, A)$. Let $\mathcal{H}^T$ be the transpose of $\mathcal{H}$, and let G^T be the graph corresponding to $\mathcal{H}^T$. How is the graph G^T related to G?

2.35. Let G be a bipartite graph. Show that we can always renumber the nodes of G so that the node–node adjacency matrix $\mathcal{H}$ of G has the following form:

0	F
E	0

2.36. Show that a directed graph G is acyclic if and only if we can renumber its nodes so that its node–node adjacency matrix is a lower triangular matrix.

2.37. Let $\mathcal{H}$ denote the node–node adjacency matrix of a network G. Define $\mathcal{H}^k = \mathcal{H} \cdot \mathcal{H}^{k-1}$ for each $k = 2, 3, \ldots, n$. Show that the ijth entry of the matrix $\mathcal{H}^2$ is the number of directed paths consisting of two arcs from node i to node j. Then using induction, show that the ijth entry of matrix $\mathcal{H}^k$ is the number of distinct walks from node i to node j containing exactly k arcs. In making this assessment, assume that two walks are *distinct* if their sequences of arcs are different (even if the unordered set of arcs are the same).

2.38. Let $\mathcal{H}$ denote the node–node adjacency matrix of a network G. Show that G is strongly connected if and only if the matrix $\mathcal{R}$ defined by $\mathcal{R} = \mathcal{H} + \mathcal{H}^2 + \mathcal{H}^3 + \cdots + \mathcal{H}^n$ has no zero entry.

2.39. Write a pseudocode that takes as an input the node–node adjacency matrix representation of a network and produces as an output the forward and reverse star representations of the network. Your pseudocode should run in $O(n^2)$ time.

2.40. Write a pseudocode that accepts as an input the forward star representation of a network and produces as an output the network's node–node adjacency matrix representation.

2.41. Write a pseudocode that takes as an input the forward star representation of a network and produces the reverse star representation. Your pseudocode should run in $O(m)$ time.

2.42. Consider the minimum cost flow problem shown in Figure 2.28. Suppose that arcs (1, 2) and (3, 5) have lower bounds equal to $l_{12} = l_{35} = 5$. Transform this problem to one where all arcs have zero lower bounds.

2.43. In the network shown in Figure 2.28, some arcs have finite capacities. Transform this problem to one where all arcs are uncapacitated.

2.44. Consider the minimum cost flow problem shown in Figure 2.28 (note that some arcs have negative arc costs). Modify the problem so that all arcs have nonnegative arc costs.

2.45. Construct the residual network for the minimum cost flow problem shown in Figure 2.28 with respect to the following flow: $x_{12} = x_{13} = x_{32} = 10$ and $x_{24} = x_{35} = x_{54} = 5$.

2.46. For the minimum cost flow problem shown in Figure 2.28, specify a vector π of node potentials so that $c_{ij}^{\pi} \geq 0$ for every arc $(i, j) \in A$. Compute cx, $c^{\pi}x$, and πb for the flow given in Exercise 2.45 and verify that $cx = c^{\pi}x + \pi b$.

2.47. Suppose that a minimum cost flow problem contains both arcs (i, j) and (j, i) for some pair of nodes. Transform this problem to one in which the network contains either arc (i, j) or arc (j, i), but not both.

2.48. Show that if x' is a feasible flow in the residual network $G(x^{\circ})$, the solution given by $x_{ij} = (x'_{ij} - x'_{ji}) + x^{\circ}_{ij}$ is a feasible flow in G and satisfies $cx = c'x' + cx^{\circ}$.

2.49. Suppose that you are given a minimum cost flow code that requires that its input data be specified so that $l_{ij} = u_{ij}$ for no arc (i, j). How would you eliminate such arcs?

2.50. Show how to transform a minimum cost flow problem stated in (2.1) into a circulation problem. Establish a one-to-one correspondence between the feasible solutions of these two problems. (*Hint:* Introduce two new nodes and some arcs.)

2.51. Show that by adding an extra node and appropriate arcs, we can formulate any minimum cost flow problem with one or more inequalities for supplies and demands (i.e., the mass balance constraints are stated as "$\leq b(i)$" for a supply node i, and/or "$\geq b(j)$" for a demand node j) into an equivalent problem with all equality constriants (i.e., "$= b(k)$" for all nodes k).

3

ALGORITHM DESIGN AND ANALYSIS

Numerical precision is the very soul of science.
—Sir D'Arcy Wentworth Thompson

Chapter Outline

3.1 Introduction
3.2 Complexity Analysis
3.3 Developing Polynomial-Time Algorithms
3.4 Search Algorithms
3.5 Flow Decomposition Algorithms
3.6 Summary

3.1 INTRODUCTION

Scientific computation is a unifying theme that cuts across many disciplines, including computer science, operations research, and many fields within applied mathematics and engineering. Within the realm of computational problem solving, we almost always combine three essential building blocks: (1) a recipe, or algorithm, for solving a particular class of problems; (2) a means for encoding this procedure in a computational device (e.g., a calculator, a computer, or even our own minds); and (3) the application of the method to the data of a specific problem. For example, to divide one number by another, we might use the iterative algorithm of long division, which is a systematic procedure for dividing any two numbers. To solve a specific problem, we could use a calculator that has this algorithm already built into its circuitry. As a first step, we would enter the data into storage locations on the calculator; then we would instruct the calculator to apply the algorithm to our data.

Although dividing two numbers is an easy task, the essential steps required to solve this very simple problem—designing, encoding, and applying an algorithm—are similar to those that we need to address when solving complex network flow problems. We need to develop an algorithm, or a mathematical prescription, for solving a class of network flow problems that contains our problem—for example, to solve a particular shortest path problem, we might use an algorithm that is known to solve any shortest path problem with nonnegative arc lengths. Since solving a network flow problem typically requires the solution of an optimization model with hundreds or thousands of variables, equations, and inequalities, we will invariably solve the problem on a computer. Doing so requires that we not only express the mathematical steps of the algorithm as a computer program, but that we also develop data structures for manipulating the large amounts of information required to rep-

resent the problem. We also need a method for entering the data into the computer and for performing the necessary operations on it during the course of the solution procedure.

In Chapter 2 we considered the lower-level steps of the computational problem-solving hierarchy; that is, we saw how to represent network data and therefore how to encode and manipulate the data within a computer. In this chapter we consider the highest level of the solution hierarchy: How do we design algorithms, and how do we measure their effectiveness? Although the idea of an algorithm is an old one—Chinese mathematicians in the third century B.C. had already devised algorithms for solving small systems of simultaneous equations—researchers did not begin to explore the notion of algorithmic efficiency as discussed in this book in any systematic and theoretical sense until the early 1970s. This particular subject matter, known as computational complexity theory, provides a framework and a set of analysis tools for gauging the work performed by an algorithm as measured by the elementary operations (e.g., addition, multiplication) it performs. One major stream of research in computational complexity theory has focused on developing performance guarantees or worst-case analyses that address the following basic question: When we apply an algorithm to a class of problems, can we specify an upper bound on the amount of computations that the algorithm will require? Typically, the performance guarantee is measured with respect to the size of the underlying problem: for example, for network flow problems, the number n of nodes and the number m of arcs in the underlying graph. For example, we might state that the complexity of an algorithm for solving shortest path problems with nonnegative arc lengths is $2n^2$, meaning that the number of computations grow no faster than twice the square of the number of nodes. In this case we say that the algorithm is "good" because its computations are bounded by a polynomial in the problem size (as measured by the number of nodes). In contrast, the computational time for a "bad" algorithm would grow exponentially when applied to a certain class of problems. With the theoretical worst-case bound in hand, we can now assess the amount of work required to solve (nonnegative length) shortest path problems as a function of their size. We also have a tool for comparing any two algorithms: the one with the smaller complexity bound is preferred from the viewpoint of a worst-case analysis.

Network optimization problems have been the core and influential subject matter in the evolution of computational complexity theory. Researchers and analysts have developed many creative ideas for designing efficient network flow algorithms based on the concepts and results emerging in the study of complexity theory; at the same time, many ideas originating in the study of network flow problems have proven to be useful in developing and analyzing a wide variety of algorithms in many other problem domains. Although network optimization has been a constant subject of study throughout the years, researchers have developed many new results concerning complexity bounds for network flow algorithms at a remarkable pace in recent years. Many of these recent innovations draw on a small set of common ideas, which are simultaneously simple and powerful.

Our intention in this chapter is to bring together some of the most important of these ideas. We begin by reviewing the essential ingredients of computational complexity theory, including the definition and computational implications of good

algorithms. We then describe several key ideas that appear to be mainstays in the development and analysis of good network flow algorithms. One idea is an approximation strategy, known as *scaling*, that solves a sequence of "simple" approximate versions of a given problem (determined by scaling the problem data) in such a way that the problems gradually become better approximations of the original problem. A second idea is a *geometric improvement argument* that is quite useful in analyzing algorithms; it shows that whenever we make sufficient (i.e., fixed percentage) improvements in the objective function at every iteration, an algorithm is good.

We also describe some important tools that can be used in analyzing or streamlining algorithms: (1) a *potential function method* that provides us with a scalar integer-valued function that summarizes the progress of an algorithm in such a way that we can use it to bound the number of steps that the algorithm takes, and (2) a *parameter balancing technique* that permits us to devise an algorithm based on some underlying parameter and then to set the parameter so that we minimize the number of steps required by the algorithm. Next, we introduce the idea of *dynamic programming*, which is a useful algorithmic strategy for developing good algorithms. The dynamic programming technique decomposes the problem into stages and uses a recursive relationship to go from one stage to another. Finally, we introduce the *binary search* technique, another well-known technique for obtaining efficient algorithms. Binary search performs a search over the feasible values of the objective function and solves an easier problem at each search point.

In this chapter we also describe important and efficient (i.e., good) algorithms that we use often within the context of network optimization: *search algorithms* that permit us to find all the nodes in a network that satisfy a particular property. Often in the middle of a network flow algorithm, we need to discover all nodes that share a particular attribute; for example, in solving a maximum flow problem, we might want to find all nodes that are reachable from the designated source node along a directed path in the residual network. Search algorithms provide us with a mechanism to perform these important computations efficiently. As such, they are essential, core algorithms used to design other more complex algorithms.

Finally, we study *network decomposition algorithms* that permit us to decompose a solution to a network flow problem, formulated in terms of arc flows, into a set of flows on paths and cycles. In our treatment of network flow problems, we have chosen to use a model with flows defined on arcs. An alternative modeling approach is to view all flows as being carried along paths and cycles in the network. In this model, the variables are the amount of flow that we send on any path or cycle. Although the arc flow formulation suffices for most of the topics that we consider in this book, on a few occasions such as our discussion of multicommodity flows in Chapter 17, we will find it more convenient to work with a path and cycle flow model. Moreover, even if we do not use the path and cycle flow formulation per se, understanding this model provides additional insight about the nature of network flow problems. The network decomposition algorithms show that the arc flow model and the path and cycle flow model are equivalent, so we could use any of these models for formulating network flow problems; in addition, these algorithms provide us with an efficient computational procedure for finding a set of path and cycle flows that is equivalent to any given set of arc flows.

3.2 COMPLEXITY ANALYSIS

An algorithm is a step-by-step procedure for solving a problem. By a *problem* we mean a generic model such as the shortest path problem or the minimum cost flow problem. Problems can be subsets of one another: For example, not only does the set of all shortest path problems define a problem, but so does the class of all shortest path problems with nonnegative arc costs. An *instance* is a special case of a problem with data specified for all the problem parameters. For example, to define an instance of the shortest path problem we would need to specify the network topology $G = (N, A)$, the source and destination nodes, and the values of the arc costs. An algorithm is said to *solve* a problem P if when applied to any instance of P, the algorithm is guaranteed to produce a solution. Generally, we are interested in finding the most "efficient" algorithm for solving a problem. In the broadest sense, the notion of efficiency involves all the various computing resources needed for executing an algorithm. However, in this book since time is often a dominant computing resource, we use the time taken by an algorithm as our metric for measuring the "most efficient" algorithm.

Different Complexity Measures

As already stated, an algorithm is a step-by-step procedure for solving a problem. The different steps an algorithm typically performs are (1) assignment steps (such as assigning some value to a variable), (2) arithmetic steps (such as addition, subtraction, multiplication, and division), and (3) logical steps (such as comparison of two numbers). The number of steps performed (or taken) by the algorithm is said to be the sum total of all steps it performs. The number of steps taken by an algorithm, which to a large extent determines the time it requires, will differ from one instance of the problem to another. Although an algorithm might solve some "good" instances of the problem quickly, it might take a long time to solve some "bad" instances. This range of possible outcomes raises the question of how we should measure the performance of an algorithm so that we can select the "best" algorithm from among several competing algorithms for solving a problem. The literature has widely adopted three basic approaches for measuring the performance of an algorithm:

1. *Empirical analysis.* The objective of empirical analysis is to estimate how algorithms behave in practice. In this analysis we write a computer program for the algorithm and test the performance of the program on some classes of problem instances.
2. *Average-case analysis.* The objective of average-case analysis is to estimate the expected number of steps an algorithm takes. In this analysis we choose a probability distribution for the problem instances and using statistical analysis derive asymptotic expected running times for the algorithm.
3. *Worst-case analysis.* Worst-case analysis provides upper bounds on the number of steps that a given algorithm can take on *any* problem instance. In this analysis we count the largest possible number of steps; consequently, this analysis provides a "guarantee" on the number of steps an algorithm will take to solve any problem instance.

Each of these three performance measures has its relative merits and drawbacks. Empirical analysis has several major drawbacks: (1) an algorithm's performance depends on the programming language, compiler, and computer used for the computational experiments, as well as the skills of the programmer who wrote the program; (2) often this analysis is too time consuming and expensive to perform; and (3) the comparison of algorithms is often inconclusive in the sense that different algorithms perform better on different classes of problem instances and different empirical studies report contradictory results.

Average-case analysis has major drawbacks as well: (1) the analysis depends crucially on the probability distribution chosen to represent the problem instances, and different choices might lead to different assessments as to the relative merits of the algorithms under consideration; (2) it is often difficult to determine appropriate probability distributions for problems met in practice; and (3) the analysis often requires quite intricate mathematics even for assessing the simplest type of algorithm—the analysis typically is extremely difficult to carry out for more complex algorithms. Furthermore, the prediction of an algorithm's performance, based on its average-case analysis, is tailored for situations in which the analyst needs to solve a large number of problem instances; it does not provide information about the distribution of outcomes. In particular, although the average-case performance of an algorithm might be good, we might encounter exceptions with little statistical significance on which the algorithm performs very badly.

Worst-case analysis avoids many of these drawbacks. The analysis is independent of the computing environment, is relatively easier to perform, provides a guarantee on the steps (and time) taken by an algorithm, and is definitive in the sense that it provides conclusive proof that an algorithm is superior to another for the worst possible problem instances that an analyst might encounter. Worst-case analysis is not perfect, though: One major drawback of worst-case analysis is that it permits "pathological" instances to determine the performance of an algorithm, even though they might be exceedingly rare in practice. However, the advantages of the worst-case analysis have traditionally outweighed its shortcomings, and this analysis has become the most popular method for measuring algorithmic performance in the scientific literature. The emergence of the worst-case analysis as a tool for assessing algorithms has also had a great impact on the field of network flows, stimulating considerable research and fostering many algorithmic innovations. In this book, too, we focus primarily on worst-case analysis. We also try to provide insight about the empirical performance, particularly in Chapter 18, since we believe that the empirical behavior of algorithms provides important information for guiding the use of algorithms in practice.

Problem Size

To express the time requirement of an algorithm, we would like to define some measure of the "complexity" of the problem instances we encounter. Having a single performance measure for all problem instances rarely makes sense since as the problem instances become larger, they typically become more difficult to solve (i.e., take more time); often the effort required to solve problem instances varies roughly

with their size. Hence to measure the complexity of problem instances, we must consider the "size" of the problem instance. But what is the size of a problem?

Before we address this question, let us discuss what is the size of a data item whose value is x. We can make one of the two plausible assumptions: (1) assume that the size of the data item is x, or (2) assume that the size of the data item is $\log x$. Of these, for several reasons the second assumption is more common. The primary reason is that $\log x$ reflects the way that computers work. Most modern computers represent numbers in binary form (i.e., in bits) and store them in memory locations of fixed bit size. The binary representation of item x requires $\log x$ bits, and hence the space required to store x is proportional to $\log x$.

The size of a network problem is a function of how the problem is stated. For a network problem, the input might be in the form of one of the representations discussed in Section 2.3. Suppose that we specify the network in the adjacency list representation, which is the most space-efficient representation we could use. Then the size of the problem is the number of bits needed to store its adjacency list representation. Since the adjacency list representation stores one pointer for each node and arc, and one data element for each arc cost coefficient and each arc capacity, it requires approximately $n \log n + m \log m + m \log C + m \log U$ bits to store all of the problem data for a minimum cost network flow problem (recall that C represents the largest arc cost and U represents the largest arc capacity). Since $m \leq n^2$, $\log m \leq \log n^2 = 2 \log n$. For this reason, when citing the size of problems using a "big O" complexity notation that ignores constants (see the subsection entitled "big O" to follow), we can (and usually do) replace each occurrence of $\log m$ by the term $\log n$.

In principle, we could express the running time of an algorithm as a function of the problem size; however, that would be unnecessarily awkward. Typically, we will express the running time more simply and more directly as a function of the network parameters n, m, $\log C$, and $\log U$.

Worst-Case Complexity

The time taken by an algorithm, which is also called the *running time* of the algorithm, depends on both the nature and size of the input. Larger problems require more solution time, and different problems of the same size typically require different solution times due to differences in the data. A *time complexity function* for an algorithm is a function of the problem size and specifies the largest amount of time needed by the algorithm to solve any problem instance of a given size. In other words, the time complexity function measures the rate of growth in solution time as the problem size increases. For example, if the time complexity function of a network algorithm is cnm for some constant $c \geq 0$, the running time needed to solve *any* network problem with n nodes and m arcs is at most cnm. Notice that the time complexity function accounts for the dependence of the running time on the problem size by measuring the *largest* time needed to solve any problem instance of a given size; at this level of detail in measuring algorithmic performance, the complexity function provides a performance guarantee that depends on the appropriate measure of the problem's input data. Accordingly, we also refer to the time complexity function as the *worst-case complexity* (or, simply, the *complexity*) of the algorithm. We

also refer to the worst-case complexity of an algorithm as its *worst-case bound*, for it states an upper bound on the time taken by the algorithm.

Big O Notation

To define the complexity of an algorithm completely, we need to specify the values for one or more constants. In most cases the determination of these constants is a nontrivial task; moreover, the determination might depend heavily on the computer, and other factors. Consider, for example, the following segment of an algorithm, which adds two $p \times q$ arrays:

```
for i: = 1 to p do
  for j: = 1 to q do
    c_ij: = a_ij + b_ij;
```

At first glance, this program segment seems to perform exactly pq additions and the same number of assignments of values to the computer locations storing the values of the variables c_{ij}. This accounting, however, ignores many computations that the computer would actually perform. A computer generally stores a two-dimensional array of size $p \times q$ as a single array of length pq and so would typically store the element a_{ij} at the location $(i - 1)q + j$ of the array a. Thus each time we retrieve the value of a_{ij} and b_{ij} we would need to perform one subtraction, one multiplication, and one addition. Further, whenever, the computer would increment the index i (or j), it would perform a comparison to determine whether $i > p$ (or $j > q$). Needless to say, such a detailed analysis of an algorithm is very time consuming and not particularly illuminating.

The dependence of the complexity function on the constants poses yet another problem: How do we compare an algorithm that performs $5n$ additions and $3n$ comparisons with an algorithm that performs n multiplications and $2n$ subtractions? Different computers perform mathematical and logical operations at different speeds, so neither of these algorithms might be universally better.

We can overcome these difficulties by ignoring the constants in the complexity analysis. We do so by using "big O" notation, which has become commonplace in computational mathematics, and replace the lengthy and somewhat awkward expression "the algorithm required cnm time for some constant c" by the equivalent expression "the algorithm requires $O(nm)$ time." We formalize this definition as follows:

An algorithm is said to run in $O(f(n))$ time if for some numbers c and n_0, the time taken by the algorithm is at most $cf(n)$ for all $n \geq n_0$.

Although we have stated this definition in terms of a single measure n of a problem-size parameter, we can easily incorporate other size parameters m, C, and U in the definition.

The big O notation has several implications. The complexity of an algorithm is an upper bound on the running time of the algorithm for sufficiently large values of n. Therefore, this complexity measure states the asymptotic growth rate of the running time. We can justify this feature of the complexity measure from practical

considerations since we are more interested about the behavior of the algorithm on very large inputs, as these inputs determine the limits of applicability of the algorithm. Furthermore, the big O notation indicates only the most dominant term in the running time, because for sufficiently large n, terms with a smaller growth rate become insignificant as compared to terms with a higher growth rate. For example, if the running time of an algorithm is $100n + n^2 + 0.0001n^3$, then for all $n \geq 100$, the second term dominates the first term, and for all $n \geq 10{,}000$, the third term dominates the second term. Therefore, the complexity of the algorithm is $O(n^3)$.

Another important implication of ignoring constants in the complexity analysis is that we can assume that each elementary mathematical operation, such as addition, subtraction, multiplication, division, assignment, and logical operations, requires an equal amount of time. A computer typically performs these operations at different speeds, but the variation in speeds can typically be bounded by a constant (provided the numbers are not too large), which is insignificant in big O notation. For example, a computer typically multiplies two numbers by repeated additions and the number of such additions are equal to number of bits in the smaller number. Assuming that the largest number can have 32 bits, the multiplication can be at most 32 times more expensive than addition. These observations imply that we can summarize the running time of an algorithm by recording the number of elementary mathematical operations it performs, viewing every operation as requiring an equivalent amount of time.

Similarity Assumption

The assumption that each arithmetic operation takes one step might lead us to underestimate the asymptotic running time of arithmetic operations involving very large numbers on real computers since, in practice, a computer must store such numbers in several words of its memory. Therefore, to perform each operation on very large numbers, a computer must access a number of words of data and thus take more than a constant number of steps. Thus the reader should be forewarned that the running times are misleading if the numbers are exponentially large. To avoid this systematic underestimation of the running time, in comparing two running times, we will sometimes assume that both C (i.e., the largest arc cost) and U (i.e., the largest arc capacity) are polynomially bounded in n [i.e., $C = O(n^k)$ and $U = O(n^k)$, for some constant k]. We refer to this assumption as the *similarity assumption*.

Polynomial- and Exponential-Time Algorithms

We now consider the question of whether or not an algorithm is "good." Ideally, we would like to say that an algorithm is good if it is sufficiently efficient to be usable in practice, but this definition is imprecise and has no theoretical grounding. An idea that has gained wide acceptance in recent years is to consider a network algorithm "good" if its worst-case complexity is bounded by a polynomial function of the problem's parameters (i.e., it is a polynomial function of n, m, $\log C$, and $\log U$). Any such algorithm is said to be a *polynomial-time algorithm*. Some examples of polynomial-time bounds are $O(n^2)$, $O(nm)$, $O(m + n \log C)$, $O(nm \log(n^2/m))$, and $O(nm + n^2 \log U)$. (Note that $\log n$ is polynomially bounded because

its growth rate is slower than n.) A polynomial-time algorithm is said to be a *strongly polynomial-time algorithm* if its running time is bounded by a polynomial function in only n and m, and does not involve $\log C$ or $\log U$, and is a *weakly polynomial-time algorithm* otherwise. Some strongly polynomial time bounds are $O(n^2 m)$ and $O(n \log n)$. In principle, strongly polynomial-time algorithms are preferred to weakly polynomial-time algorithms because they can solve problems with arbitrary large values for the cost and capacity data.

Note that in this discussion we have said that an algorithm is polynomial time if its running time is bounded by a polynomial in the network parameters n, m, $\log C$, and $\log U$. Typically, in computational complexity we say that an algorithm is polynomial time if its running time is bounded by a polynomial in the problem size, in this case $n \log n + m \log m + n \log C + m \log U$; however, it is easy to see that the running time of a network problem is bounded by a polynomial in its problem size if and only if it is also bounded by a polynomial in the problem parameters. For example, if the running time is bounded by n^{100}, it is strictly less than the problem size to the 100th power. Similarly, if the running time is bounded by the problem size to the 100th power, it is less than $(n \log n + m \log m + n \log C + m \log U)^{100}$, which in turn is bounded by $(n^2 + m^2 + n \log C + m \log U)^{100}$, which is a polynomial in n, m, $\log C$, and $\log U$.

An algorithm is said to be an *exponential-time algorithm* if its worst-case running time grows as a function that cannot be polynomially bounded by the input length. Some examples of exponential time bounds are $O(nC)$, $O(2^n)$, $O(n!)$, and $O(n^{\log n})$. (Observe that nC cannot be bounded by a polynomial function of n and $\log C$.) We say that an algorithm is a *pseudopolynomial-time algorithm* if its running time is polynomially bounded in n, m, C, and U. The class of pseudopolynomial-time algorithms is an important subclass of exponential-time algorithms. Some examples of pseudopolynomial-time bounds are $O(m + nC)$ and $O(mC)$. For problems that satisfy the similarity assumption, pseudopolynomial-time algorithms become polynomial-time algorithms, but the algorithms will not be attractive if C and U are high-degree polynomials in n.

There are several reasons for preferring polynomial-time algorithms to exponential-time algorithms. Any polynomial-time algorithm is asymptotically superior to any exponential-time algorithm, even in extreme cases. For example, n^{4000} is smaller than $n^{0.1 \log n}$ if n is sufficiently large (i.e., $n \geq 2^{100,000}$). Figure 3.1 illustrates the growth rates of several typical complexity functions. The exponential complexity functions have an explosive growth rate and, in general, they are able to solve only small problems. Further, much practical experience has shown that the polynomials encountered in practice typically have a small degree, and generally, polynomial-time algorithms perform better than exponential-time algorithms.

n	$\log n$	$n^{0.5}$	n^2	n^3	2^n	$n!$
10	3.32	3.16	10^2	10^3	10^3	3.6×10^6
100	6.64	10.00	10^4	10^6	1.27×10^{30}	9.33×10^{157}
1000	9.97	31.62	10^6	10^9	1.07×10^{301}	$4.02 \times 10^{2,567}$
10,000	13.29	100.00	10^8	10^{12}	$0.99 \times 10^{3,010}$	$2.85 \times 10^{35,659}$

Figure 3.1 Growth rates of some polynomial and exponential functions.

A brief examination of the effects of improved computer technology on algorithms is even more revealing in understanding the impact of various complexity functions. Consider a polynomial-time algorithm whose complexity is $O(n^2)$. Suppose that the algorithm is able to solve a problem of size n_1 in 1 hour on a computer with speed of s_1 instructions per second. If we increase the speed of the computer to s_2, then $(n_2/n_1)^2 = s_2/s_1$ specifies the size n_2 of the problem that the algorithm can solve in the same time. Consequently, a 100-fold increase in computer speed would allow us to solve problems that are 10 *times* larger. Now consider an exponential-time algorithm with a complexity of $O(2^n)$. As before, let n_1 and n_2 denote the problem sizes solved on a computer with speeds s_1 and s_2 in 1 hour of computation time. Then $s_2/s_1 = 2^{n_2}/2^{n_1}$. Alternatively, $n_2 = n_1 + \log(s_2/s_1)$. In this case, a 100-fold increase in computer speed would allow us to solve problems that are only about 7 *units* larger. This discussion shows that a substantial increase in computer speed allows us to solve problems by polynomial-time algorithms that are larger by a multiplicative factor; for exponential-time algorithms we obtain only additive improvements. Consequently, improved hardware capabilities of computers can have only a marginal impact on the problem-solving ability of exponential-time algorithms.

Let us pause to summarize our discussion of polynomial and exponential-time algorithms. In the realm of complexity theory, our objective is to obtain polynomial-time algorithms, and within this domain our objective is to obtain an algorithm with the smallest possible growth rate, because an algorithm with smaller growth rate is likely to permit us to solve larger problems in the same amount of computer time (depending on the associated constants). For example, we prefer $O(\log n)$ to $O(n^k)$ for any $k > 0$, and we prefer $O(n^2)$ to $O(n^3)$. However, running times involving more than one parameter, such as $O(n\, m \log n)$ and $O(n^3)$, might not be comparable. If $m < n^2/\log n$, then $O(n\, m \log n)$ is superior; otherwise, $O(n^3)$ is superior.

Can we say that a polynomial-time algorithm with a smaller growth rate would run faster in practice, or even that a polynomial-time algorithm would empirically outperform an exponential-time algorithm? Although this statement is generally true, there are many exceptions to the rule. A classical exception is provided by the simplex method and Khachian's "ellipsoid" algorithm for solving linear programming problems. The simplex algorithm is known to be an exponential-time algorithm, but in practice it runs much faster than Khachian's polynomial-time algorithm. Many of these exceptions can be explained by the fact that the worst-case complexity is greatly inferior to the average complexity of some algorithms, while for other algorithms the worst-case complexity and the average complexity might be comparable. As a consequence, considering worst-case complexity as synonymous with average complexity can lead to incorrect conclusions.

Sometimes, we might not succeed in developing a polynomial-time algorithm for a problem. Indeed, despite their best efforts spanning several decades, researchers have been unable to develop polynomial-time alogorithms for a huge collectin of important combinatorial problems; all known algorithms for these problems are exponential-time algorithms. However, the research community has been able to show that most of these problems belong to a class of problems, called *$\mathcal{NP}$-complete problems*, that are equivalent in the sense that if there exists a polynomial-time algorithm for one problem, there exists a polynomial-time algorithm for every other $\mathcal{NP}$-complete problem. Needless to say, developing a polynomial-time algorithm

for some $\mathcal{NP}$-complete problem is one of the most challenging and intriguing issues facing the research community; the available evidence suggests that no such algorithm exists. We discuss the theory of $\mathcal{NP}$-completeness in greater detail in Appendix B.

Big Ω and Big Θ Notation

The big O notation that we introduced earlier in this section is but one of several convenient notational devices that researchers use in the analysis of algorithms. In this subsection we introduce two related notational constructs: the big Ω (big omega) notation and the big Θ (big theta) notation.

Just as the big O notation specifies an upper bound on an algorithm's performance, the big Ω notation specifies a lower bound on the running time.

An algorithm is said to be $\Omega(f(n))$ if for some numbers c' and n_0 and all $n \geq n_0$, the algorithm takes at least $c'f(n)$ time on some problem instance.

The reader should carefully note that the big O notation and the big Ω notation are defined in somewhat different ways. If an algorithm runs in $O(f(n))$ time, *every* instance of the problem of size n takes *at most* $cf(n)$ time for a constant c. On the other hand, if an algorithm runs in $\Omega(f(n))$ time, *some* instance of size n takes *at least* $c'f(n)$ time for a constant c'.

The big Θ (big theta) notation provides both a lower and an upper bound on an algorithm's performance.

An algorithm is said to be $\Theta(f(n))$ if the algorithm is both $O(f(n))$ and $\Omega(f(n))$.

We generally prove an algorithm to be an $O(f(n))$ algorithm and then try to see whether it is also an $\Omega(f(n))$ algorithm. Notice that the proof that the algorithm requires $O(f(n))$ time does not imply that it would actually take $cf(n)$ time to solve all classes of problems of the type we are studying. The upper bound of $cf(n)$ could be "too loose" and might never be achieved. There is always a distinct possibility that by conducting a more clever analysis of the algorithm we might be able to improve the upper bound of $cf(n)$, replacing it by a "tighter" bound. However, if we prove that the algorithm is also $\Omega(f(n))$, we know that the upper bound of $cf(n)$ is "tight" and cannot be improved by more than a constant factor. This result would imply that the algorithm can actually achieve its upper bound and no tighter bound on the algorithm's running time is possible.

Potential Functions and Amortized Complexity

An algorithm typically performs some basic operations repetitively with each operation performing a sequence of steps. To bound the running time of the algorithm we must bound the running time of each of its basic operations. We typically bound the total number of steps associated with an operation using the following approach: We obtain a bound on the number of steps per operation, obtain a bound on the number of operations, and then take a product of the two bounds. In some of the

algorithms that we study in this book, the time required for a certain operation might vary depending on the problem data and/or the stage the algorithm is in while solving a problem. Although the operation might be easy to perform most of the time, occasionally it might be quite expensive. When this happens and we consider the time for the operation corresponding to the worst-case situation, we could greatly overestimate the running time of the algorithm. In this situation, a more global analysis is required to obtain a "tighter" bound on the running time of the operation. Rather than bounding the number of steps per operation and the number of operations executed in the algorithm, we should try to bound the total number of steps over all executions of these operations. We often carry out this type of worst-case analysis using a *potential function* technique.

We illustrate this concept on a problem of inserting and removing data from a data structure known as a *stack* (see Appendix A for a discussion of this data structure). On a stack S, we perform two operations:

push(x, S). Add element x to the *top* of the stack S.

popall(S). Pop (i.e., take out) every element of S.

The operation *push(x, S)* requires $O(1)$ time and the operation popall(S) requires $O(|S|)$ time. Now assume that starting with an empty stack, we perform a sequence of n operations in which push and popall operations occur in a random order. What is the worst-case complexity of performing this sequence of n operations?

A naive worst-case analysis of this problem might proceed as follows. Since we require at most n *push* operations, and each push takes $O(1)$ time, the push operations require a total of $O(n)$ time. A popall requires $O(|S|)$ time and since $|S| \leq n$, the complexity of this operation is $O(n)$. Since our algorithm can invoke at most n popall operations, these operations take a total of $O(n^2)$ time. Consequently, a random sequence of n push and popall operations has a worst-case complexity of $O(n^2)$.

However, if we look closely at the arguments we will find that the bound of $O(n^2)$ is a substantial overestimate of the algorithm's computational requirements. A popall operation pops $|S|$ items from the stack, one by one until the stack becomes empty. Now notice that any element that is popped from the stack must have been pushed into the stack at some point, and since the number of push operations is at most n, the total number of elements popped out of the stack must be at most n. Consequently, the total time taken by all popall operations is $O(n)$. We can therefore conclude that a random sequence of n push and popall operations has a worst-case complexity of $O(n)$.

Let us provide a formal framework, using *potential functions*, for conducting the preceding arguments. Potential function techniques are general-purpose techniques for establishing the complexity of an algorithm by analyzing the effects of different operations on an appropriately defined function. The use of potential functions enables us to define an "accounting" relationship between the occurrences of various operations of an algorithm so that we can obtain a bound on the operations that might be difficult to obtain using other arguments.

Let $\phi(k) = |S|$ denote the number of items in the stack at the end of the kth

step; for the purpose of this argument we define a step as either a push or a popall operation. We assume that we perform the popall step on a nonempty stack; for otherwise, it requires $O(1)$ time. Initially, $\phi(0) = 0$. Each push operation increases $\phi(k)$ by 1 unit and takes 1 unit of time. Each popall step decreases $\phi(k)$ by at least 1 unit and requires time proportional to $\phi(k)$. Since the total increase in ϕ is at most n (because we invoke at most n push steps), the total decrease in ϕ is also at most n. Consequently, the total time taken by all push and popall steps is $O(n)$.

This argument is fairly representative of the potential function arguments. Our objective was to bound the time for the popalls. We did so by defining a potential function that decreases whenever we perform a popall. The potential increases only when we perform a push. Thus we can bound the total decrease by the total increase in ϕ. In general, we bound the number of steps of one type by using known bounds on the number of steps of other types.

The analysis we have just discussed is related to the concept known as *amortized complexity*. An operation is said to be of amortized complexity $O(f(n))$ if the time to perform a sequence of k operations is $O(kf(n))$ for sufficiently large k. In our preceding example, the worst-case complexity of performing k popalls for $k \geq n$ is $O(k)$; hence the amortized complexity of the popall operation is $O(1)$. Roughly speaking, the amortized complexity of an operation is the "average" worst-case complexity of the operation so that the total obtained using this average will indeed be an upper bound on the number of steps performed by the algorithm.

Parameter Balancing

We frequently use the parameter balancing technique in situations when the running time of an algorithm is a function of a parameter k and we wish to determine the value of k that gives the smallest running time. To be more specific, suppose that the running time of an algorithm is $O(f(n, m, k) + g(n, m, k))$ and we wish to determine an optimal value of k. We shall assume that $f(n, m, k) \geq 0$ and $g(n, m, k) \geq 0$ for all feasible values of k. The optimization problem is easy to solve if the functions $f(n, m, k)$ and $g(n, m, k)$ are both either monotonically increasing or monotonically decreasing in k. In the former case, we set k to the smallest possible value, and in the latter case, we set k to the largest possible value. Finding the optimal value of k is more complex if one function is monotonically decreasing and the other function is monotonically increasing. So let us assume that $f(n, m, k)$ is monotonically decreasing in k and $g(n, m, k)$ is monotonically increasing in k.

One method for selecting the optimal value of k is to use differential calculus. That is, we differentiate $f(n, m, k) + g(n, m, k)$ with respect to k, set the resulting expression equal to zero, and solve for k. A major drawback of this approach is that finding a value of k that will set the expression to value zero, and so determine the optimal value of k, is often a difficult task. Consider, for example, a shortest path algorithm (which we discuss in Section 4.7) that runs in time $O(m \log_k n + nk \log_k n)$. In this case, choosing the optimal value of k is not trivial. We can restate the algorithm's time bound as $O((m \log n + nk \log n)/\log k)$. The derivative of this expression with respect to k is

$$(nk \log n \log k - m \log n - nk \log n)/k(\log k)^2.$$

Setting this expression to zero, we obtain

$$m + nk - nk \log k = 0.$$

Unfortunately, we cannot solve this equation in closed form.

The parameter balancing technique is an alternative method for determining the "optimal value" of k and is based on the idea that it is not necessary to select a value of k that minimizes $f(n, m, k) + g(n, m, k)$. Since we are evaluating the performance of algorithms in terms of their worst-case complexity, it is sufficient to select a value of k for which $f(n, m, k) + g(n, m, k)$ is within a constant factor of the optimal value. The parameter balancing technique determines a value of k so that $f(n, m, k) + g(n, m, k)$ is at most twice the minimum value.

In the parameter balancing technique, we select k^* so that $f(n, m, k^*) = g(n, m, k^*)$. Before giving a justification of this approach, we illustrate it on two examples. We first consider the $O(m \log_k n + nk \log_k n)$ time shortest path algorithm that we mentioned earlier. We first note that $m \log_k n$ is a decreasing function of k and $nk \log_k n$ is an increasing function of k. Therefore, the parameter balancing technique is appropriate. We set $m \log_{k^*} n = nk^* \log_{k^*} n$, which gives $k^* = m/n$. Consequently, we achieve the best running time of the algorithm, $O(m \log_{m/n} n)$, by setting $k = m/n$.

Our second example concerns a maximum flow algorithm whose running time is $O((n^3/k)(\log k) + nm(\log k))$. We set

$$\frac{n^3}{k^*} \log k^* = nm \log k^*,$$

which gives $k^* = n^2/m$. Therefore, the best running time of this maximum flow algorithm is $O(nm \log(n^2/m))$. In Exercise 3.13 we discuss more examples of the parameter balancing technique.

We now justify the parameter balancing technique. Suppose we select k^* so that $f(n, m, k^*) = g(n, m, k^*)$. Let $\lambda^* = f(n, m, k^*) + g(n, m, k^*)$. Then for any $k < k^*$,

$$f(n, m, k) + g(n, m, k) \geq f(n, m, k) \geq f(n, m, k^*) = \lambda^*/2. \tag{3.1}$$

The second inequality follows from the fact that the function $f(n, m, k)$ is monotonically decreasing in k. Similarly, for any $k > k^*$,

$$f(n, m, k) + g(n, m, k) \geq g(n, m, k) \geq g(n, m, k^*) = \lambda^*/2. \tag{3.2}$$

The expressions (3.1) and (3.2) imply that for any k,

$$f(n, m, k) + g(n, m, k) \geq \lambda^*/2.$$

This result establishes the fact that $\lambda^* = f(n, m, k^*) + g(n, m, k^*)$ is within a factor of 2 of the minimum value of $f(n, m, k) + g(n, m, k)$.

3.3 DEVELOPING POLYNOMIAL-TIME ALGORITHMS

Researchers frequently employ four important approaches for obtaining polynomial-time algorithms for network flow problems: (1) *a geometric improvement* approach, (2) *a scaling* approach, (3) *a dynamic programming* approach, and (4) *a binary search*

approach. In this section we briefly outline the basic ideas underlying these four approaches.

Geometric Improvement Approach

The geometric improvement approach permits us to show that an algorithm runs in polynomial time if at every iteration it makes an improvement in the objective function value proportional to the difference between the objective values of the current and optimal solutions. Let H be the difference between the maximum and minimum objective function values of an optimization problem. For most network problems, H is a function of n, m, C, and U. For example, in the maximum flow problem $H = mU$, and in the minimum cost flow problem $H = mCU$. We also assume that the optimal objective function value is integer.

Theorem 3.1. *Suppose that z^k is the objective function value of some solution of a minimization problem at the kth iteration of an algorithm and z^* is the minimum objective function value. Furthermore, suppose that the algorithm guarantees that for every iteration k,*

$$(z^k - z^{k+1}) \geq \alpha(z^k - z^*) \tag{3.3}$$

(i.e., the improvement at iteration $k + 1$ is at least α times the total possible improvement) for some constant α with $0 < \alpha < 1$ (which is independent of the problem data). Then the algorithm terminates in $O((\log H)/\alpha)$ iterations.

Proof. The quantity $(z^k - z^*)$ represents the total possible improvement in the objective function value after the kth iteration. Consider a consecutive sequence of $2/\alpha$ iterations starting from iteration k. If each iteration of the algorithm improves the objective function value by at least $\alpha(z^k - z^*)/2$ units, the algorithm would determine an optimal solution within these $2/\alpha$ iterations. Suppose, instead, that at some iteration $q + 1$, the algorithm improves the objective function value by less than $\alpha(z^k - z^*)/2$ units. In other words,

$$z^q - z^{q+1} \leq \alpha(z^k - z^*)/2. \tag{3.4}$$

The inequality (3.3) implies that

$$\alpha(z^q - z^*) \leq z^q - z^{q+1}. \tag{3.5}$$

The inequalities (3.4) and (3.5) imply that

$$(z^q - z^*) \leq (z^k - z^*)/2,$$

so the algorithm has reduced the total possible improvement $(z^k - z^*)$ by a factor at least 2. We have thus shown that within $2/\alpha$ consecutive iterations, the algorithm either obtains an optimal solution or reduces the total possible improvement by a factor of at least 2. Since H is the maximum possible improvement and every objective function value is an integer, the algorithm must terminate within $O((\log H)/\alpha)$ iterations. ◆

We have stated this result for the minimization version of optimization problems. A similar result applies to the maximization problems.

The geometric improvement approach might be summarized by the statement "network algorithms that have a geometric convergence rate are polynomial-time algorithms." To develop polynomial-time algorithms using this approach, we look for local improvement techniques that lead to large (i.e., fixed percentage) improvements in the objective function at every iteration. The maximum augmenting path algorithm for the maximum flow problem discussed in Section 7.3 and the maximum improvement algorithm for the minimum cost flow problem discussed in Section 9.6 provide two examples of this approach.

Scaling Approach

Researchers have used scaling methods extensively to derive polynomial-time algorithms for a wide variety of network and combinatorial optimization problems. Indeed, for problems that satisfy the similarity assumption, the scaling-based algorithms achieve the best worst-case running time for most of the network optimization problems we consider in this book.

We shall describe the simplest form of scaling, which we call *bit-scaling*. In the bit-scaling technique, we represent the data as binary numbers and solve a problem P parametrically as a sequence of problems $P_1, P_2, P_3, \ldots, P_K$: The problem P_1 approximates data to the first most significant bit, the problem P_2 approximates data to the first two most significant bits, and each successive problem is a better approximation, until $P_K = P$. Moreover, for each $k = 2, \ldots, K$, the optimal solution of problem P_{k-1} serves as the starting solution for problem P_k. The scaling technique is useful whenever reoptimization from a good starting solution is more efficient than solving the problem from scratch.

For example, consider a network flow problem whose largest arc capacity has value U. Let $K = \lceil \log U \rceil$ and suppose that we represent each arc capacity as a K-bit binary number, adding leading zeros if necessary to make each capacity K bits long. Then the problem P_k would consider the capacity of each arc as the k leading bits in its binary representation. Figure 3.2 illustrates an example of this type of scaling.

The manner of defining arc capacities easily implies the following property.

Property 3.2. *The capacity of an arc in P_k is twice that in P_{k-1} plus* 0 *or* 1.

The algorithm shown in Figure 3.3 encodes a generic version of the bit-scaling technique.

This approach is very robust, and variants of it have led to improved algorithms for both the maximum flow and minimum cost flow problems. This approach works well for these applications, in part, for the following reasons:

1. The problem P_1 is generally easy to solve.
2. The optimal solution of problem P_{k-1} is an excellent starting solution for problem P_k since P_{k-1} and P_k are quite similar. Therefore, we can easily reoptimize the problem starting from the optimal solution of P_{k-1} to obtain an optimal solution of P_k.

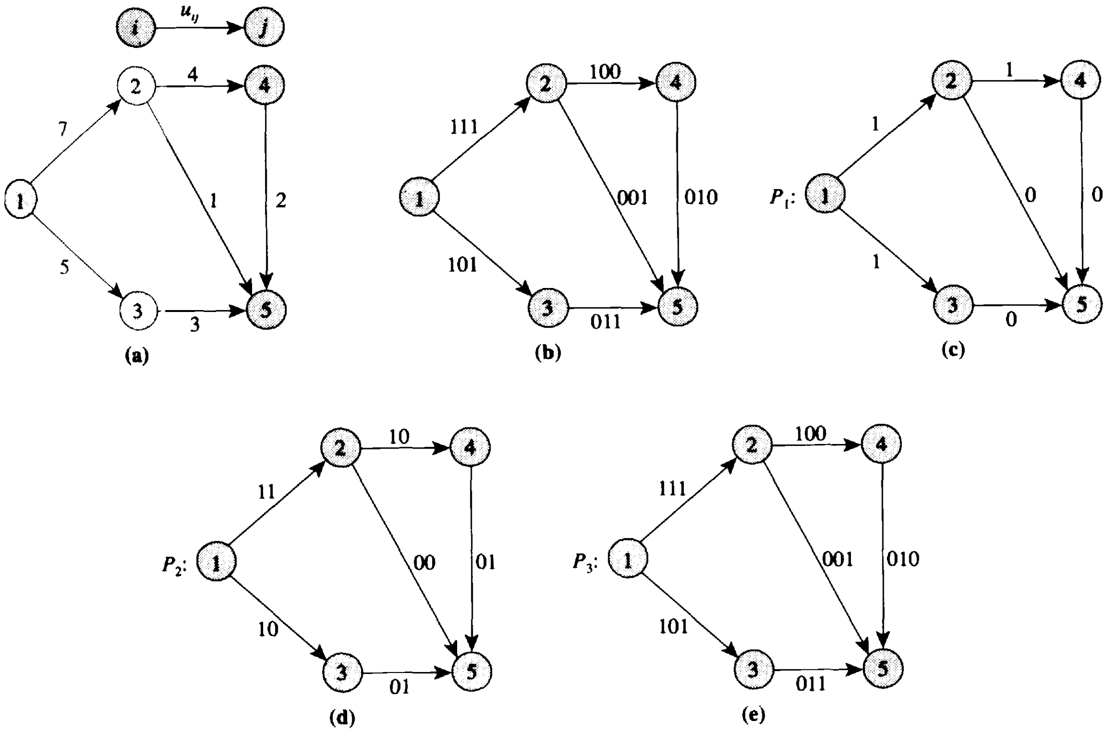

Figure 3.2 Examples of a bit-scaling technique: (a) network with arc capacities; (b) network with binary expansions of arc capacities; (c)–(e) problems P_1, P_2, and P_3.

3. The number of reoptimization problems we solve is $O(\log C)$ or $O(\log U)$. Thus for this approach to work, reoptimization needs to be only a little more efficient (i.e., by a factor of $\log C$ or $\log U$) than optimization.

Consider, for example, the maximum flow problem. Let v_k denote the maximum flow value for problem P_k and let x_k denote an arc flow corresponding to v_k. In the problem P_k, the capacity of an arc is twice its capacity in P_{k-1} plus 0 or 1. If we multiply the optimal flow x_{k-1} of P_{k-1} by 2, we obtain a feasible flow for P_k. Moreover, $v_k - 2v_{k-1} \leq m$ because multiplying the flow x_{k-1} by 2 takes care of the doubling of the capacities and the additional 1's can increase the maximum flow value by at most m units (if we add 1 to the capacity of any arc, we increase the

```
algorithm bit-scaling;
begin
    obtain an optimal solution of P_1;
    for k: = 2 to K do
    begin
        reoptimize using the optimal solution of P_{k-1} to obtain an optimal solution of P_k;
    end;
end;
```

Figure 3.3 Typical bit-scaling algorithm.

maximum flow from the source to the sink by at most 1). In general, it is easier to reoptimize such a maximum flow problem than to solve a general problem from scratch. For example, the classical labeling algorithm as discussed in Section 6.5 would perform the reoptimization in at most m augmentations, requiring $O(m^2)$ time. Therefore, the scaling version of the labeling algorithm runs in $O(m^2 \log U)$ time, improving on the running time $O(nmU)$ of the nonscaling version. The former time bound is polynomial and the latter bound is only pseudopolynomial. Thus this simple bit-scaling algorithm improves the running time dramatically.

An alternative approach to scaling considers a sequence of problems $P(1)$, $P(2), \ldots, P(K)$, each involving the original data, but in this case we do not solve the problem $P(k)$ optimally, but solve it approximately, with an error of Δ_k. Initially, Δ_1 is quite large, and it subsequently converges geometrically to 0. Usually, we can interpret an error of Δ_k as follows. From the current nearly optimal solution x^k, there is a way of modifying some or all of the data by at most Δ_k units so that the resulting solution is optimal. Our discussion of the capacity scaling algorithm for the maximum flow problem in Section 7.3 illustrates this type of scaling.

Dynamic Programming

Researchers originally conceived of dynamic programming as a stagewise optimization technique. However, for our purposes in this book, we prefer to view it as a "table-filling" approach in which we complete the entries of a two-dimensional tableau using a recursive relationship. Perhaps the best way to explain this approach is through several illustrations.

Computing Binomial Coefficients

In many application of combinatorics, for example in elementary probability, we frequently wish to determine the number ${}^pC_q$ of different combinations of p objects taken q at a time for some given values of p and q ($p \geq q$). As is well known, ${}^pC_q = p!/((p - q)!q!)$. Suppose that we wish to make this computation using only the mathematical operation of addition and using the fact that the combination function ${}^pC_q$ satisfies the following recursive relationship:

$$ {}^iC_j = {}^{i-1}C_j + {}^{i-1}C_{j-1}. \tag{3.6}$$

To solve this problem, we define a lower triangular table $D = \{d(i, j)\}$ with p rows and q columns: Its entries, which we would like to compute, will be $d(i, j) = {}^iC_j$ for $i \geq j$. We will fill in the entries in the table by scanning the rows in the order 1 through p; when scanning each row i, we scan its columns in the order 1 through i. Note that we can start the computations by setting the ith entry $d(i, 1) = {}^iC_1$ in the first column to value i since there are exactly i ways to select one object from a collection of i objects. Observe that whenever we scan the element (i, j) in the table, we have already computed the entries ${}^{i-1}C_j$ and ${}^{i-1}C_{j-1}$, and their sum yields $d(i, j)$. So we always have the available information to compute the entries in the table as we reach them. When we have filled the entire table, the entry $d(p, q)$ gives us the desired answer to our problem.

Knapsack Problem

We can also illustrate the dynamic programming approach on another problem, known as the *knapsack problem*, which is a classical model in the operations research literature. A hiker must decide which goods to include in her knapsack on a forthcoming trip. She must choose from among p objects: Object i has weight w_i (in pounds) and a utility u_i to the hiker. The objective is to maximize the utility of the hiker's trip subject to the weight limitation that she can carry no more than W pounds. This knapsack problem has the following formulation as an integer program:

$$\text{Maximize } \sum_{i=1}^{p} u_i x_i \tag{3.7a}$$

subject to

$$\sum_{i=1}^{p} w_i x_i \leq W, \tag{3.7b}$$

$$x_i = \{0, 1\} \qquad \text{for all } i. \tag{3.7c}$$

To solve the knapsack problem, we construct a $p \times W$ table D whose elements $d(i, j)$ are defined as follows:

$d(i, j)$: The maximum utility of the selected items if we restrict our selection to the items 1 through i and impose a weight restriction of j.

Clearly, our objective is to determine $d(p, W)$. We determine this value by computing $d(i, j)$ for increasing values of i and, for a fixed value of i, for increasing values of j. We now develop the recursive relationship that would allow us to compute $d(i, j)$ from those elements of the tableau that we have already computed. Note that any solution restricted to the items 1 through i, either (1) does not use item i, or (2) uses this item. In case (1), $d(i, j) = d(i - 1, j)$. In case (2), $d(i, j) = u_i + d(i - 1, j - w_i)$ for the following reason. The first term in this expression represents the value of including item i in the knapsack and the second term denotes the optimal value obtained by allocating the remaining capacity of $j - w_i$ among the items 1 through $i - 1$. We have thus shown that

$$d(i, j) = \max\{d(i - 1, j), u_i + d(i - 1, j - w_i)\}.$$

When carrying out these computations, we also record the decision corresponding to each $d(i, j)$ (i.e., whether $x_i = 0$ or $x_i = 1$). These decisions allow us to construct the solution for any $d(i, j)$, including the desired solution for $d(p, W)$.

In both these illustrations of dynamic programming, we scanned rows of the table in ascending order and for each fixed row, we scanned columns in ascending order. In general, we could scan the rows and columns of the table in either ascending or descending order as long as the recursive relationship permits us to determine the entries needed in the recursion from those we have already computed.

To conclude this brief discussion, we might note that much of the traditional literature in dynamic programming views the problem as being composed of "stages" and "states" (or possible outcomes within each state). Frequently, the stages cor-

respond to points in time (this is the reason that this topic has become known as dynamic programming). To reconceptualize our tabular approach in this stage and state framework, we would view each row as a stage and each column within each row as a possible state at that stage. For both the binomial coefficient and knapsack applications that we have considered, each stage corresponds to a restricted set of objects (items): In each case stage i corresponds to a restricted problem containing only the first i objects. In the binomial coefficient problem, the states are the number of elements in a subset of the i objects; in the knapsack problem, the states are the possible weights that we could hold in a knapsack containing only the first i items.

Binary Search

Binary search is another popular technique for obtaining polynomial-time algorithms for a variety of network problems. Analysts use this search technique to find, from among a set of feasible solutions, a solution satisfying "desired properties." At every iteration, binary search eliminates a fixed percentage (as the name binary implies, typically, 50 percent) of the solution set, until the solution set becomes so small that each of its feasible solutions is guaranteed to be a solution with the desired properties.

Perhaps the best way to describe the binary search technique is through examples. We describe two examples. In the first example, we wish to find the telephone number of a person, say James Morris, in a phone book. Suppose that the phone book contains p pages and we wish to find the page containing James Morris's phone number. The following "divide and conquer" search strategy is a natural approach. We open the phone book to the middle page, which we suppose is page x. By viewing the first and last names on this page, we reach one of the following three conclusions: (1) page x contains James Morris's telephone number, (2) the desired page is one of pages 1 through $x - 1$, or (3) the desired page is one of pages $x + 1$ to p. In the second case, we would next turn to the middle of the pages 1 through $x - 1$, and in the third case, we would next turn to the middle of the pages $x + 1$ through p. In general, at every iteration, we maintain an interval $[a, b]$ of pages that are guaranteed to contain the desired phone number. Our next trial page is the middle page of this interval, and based on the information contained on this page, we eliminate half of the pages from further consideration. Clearly, after $O(\log p)$ iterations, we will be left with just one page and our search would terminate. If we are fortunate, the search would terminate even earlier.

As another example, suppose that we are given a continuous function $f(x)$ satisfying the properties that $f(0) < 0$ and $f(1) > 0$. We want to determine an interval of size at most $\epsilon > 0$ that contains a *zero* of the function, that is, a value of x for which $f(x) = 0$ (to within the accuracy of the computer we are using). In the first iteration, the interval $[0, 1]$ contains a zero of the function $f(x)$, and we evaluate the function at the midpoint of this interval, that is, at the point 0.5. Three outcomes are possible: (1) $f(0.5) = 0$, (2) $f(0.5) < 0$, and (3) $f(0.5) > 0$. In the first case, we have found a zero x and we terminate the search. In the second case, the continuity property of the function $f(x)$ implies that the interval $[0.5, 1]$ contains a zero of the function, and in the third case the interval $[0, 0.5]$ contains a zero. In the second and third cases, our next trial point is the midpoint of the resulting interval. We repeat this process, and eventually, when the interval size is less than ϵ, we dis-

continue the search. As the reader can verify, this method will terminate within $O(\log(1/\epsilon))$ iterations.

In general, we use the binary search technique to identify a desired value of a parameter among an interval of possible values. The interval $[l, u]$ is defined by a lower limit l and an upper limit u. In the phone book example, we wanted to identify a page that contains a specific name, and in the zero value problem we wanted to identify a value of x in the range [0, 1] for which $f(x)$ is zero. At every iteration we perform a test at the midpoint $(l + u)/2$ of the interval, and determine whether the desired parameter lies in the range $[l, (l + u)/2]$ or in the range $[(l + u)/2, u]$. In the former case, we reset the upper limit to $(l + u)/2$, and in the latter case, we reset the lower limit to $(l + u)/2$. We might note that eliminating one-half of the interval requires that the problem satisfy certain properties. For instance, in the phone book example, we used the fact that the names in the book are arranged alphabetically, and in the zero-value problem we used the fact that the function $f(x)$ is continuous. We repeat this process with the reduced interval and keep reapplying the procedure until the interval becomes so small that it contains only points that are desired solutions. If $w_{\max}$ denotes the maximum (i.e., starting) width of the interval (i.e., $u - l$) and $w_{\min}$ denotes the minimum width of the interval, the binary search technique required $O(\log(w_{\max}/w_{\min}))$ iterations.

In most applications of the binary search technique, we perform a single test and eliminate half of the feasible interval. The worst-case complexity of the technique remains the same, however, even if we perform several, but a constant number, of tests at each step and eliminate a constant portion (not necessarily 50 percent) of the feasible interval (in Exercise 3.23 we discuss one such application). Although we typically use the binary search technique to perform a search over a single parameter, a generalized version of the method would permit us to search over multiple parameters.

3.4 SEARCH ALGORITHMS

Search algorithms are fundamental graph techniques that attempt to find all the nodes in a network satisfying a particular property. Different variants of search algorithms lie at the heart of many maximum flow and minimum cost flow algorithms. The applications of search algorithms include (1) finding all nodes in a network that are reachable by directed paths from a specific node, (2) finding all the nodes in a network that can reach a specific node t along directed paths, (3) identifying all connected components of a network, and (4) determining whether a given network is bipartite. To illustrate some of the basic ideas of search algorithms, in this section we discuss only the first two of these applications; Exercises 3.41 and 3.42 consider the other two applications.

Another important application of search algorithms is to identify a directed cycle in a network, and if the network is acyclic, to reorder the nodes $1, 2, \ldots, n$ so that for each arc $(i, j) \in A$, $i < j$. We refer to any such order as a *topological ordering*. Topological orderings prove to be essential constructs in several applications, such as project scheduling (see Chapter 19). They are also useful in the design of certain algorithms (see Section 10.5). We discuss topological ordering later in this section.

To illustrate the basic ideas of search algorithms, suppose that we wish to find all the nodes in a network $G = (N, A)$ that are reachable along directed paths from a distinguished node s, called the *source*. A search algorithm *fans out* from the source and identifies an increasing number of nodes that are reachable from the source. At every intermediate point in its execution, the search algorithm designates all the nodes in the network as being in one of the two states: *marked* or *unmarked*. The marked nodes are known to be reachable from the source, and the status of unmarked nodes has yet to be determined. Note that if node i is marked, node j is unmarked, and the network contains the arc (i, j), we can mark node j; it is reachable from source via a directed path to node i plus arc (i, j). Let us refer to arc (i, j) as *admissible* if node i is marked and node j is unmarked, and refer to it as *inadmissible* otherwise. Initially, we mark only the source node. Subsequently, by examining admissible arcs, the search algorithm will mark additional nodes. Whenever the procedure marks a new node j by examining an admissible arc (i, j), we say that node i is a *predecessor* of node j [i.e., $pred(j) = i$]. The algorithm terminates when the network contains no admissible arcs.

The search algorithm traverses the marked nodes in a certain order. We record this traversal order in an array *order*: the entry *order(i)* is the order of node i in the traversal. Figure 3.4 gives a formal description of the search algorithm. In the algorithmic description, *LIST* represents the set of marked nodes that the algorithm has yet to examine in the sense that some admissible arcs might emanate from them. When the algorithm terminates, it has marked all the nodes in G that are reachable from s via a directed path. The predecessor indices define a tree consisting of marked nodes. We call this tree a *search tree*. Figure 3.5(b) and (c), respectively, depict two search trees for the network shown in Figure 3.5(a).

To identify admissible arcs, we need to be able to access the arcs of the network and determine whether or not they connect a marked and unmarked node. To do so we must design a data structure for storing the arcs and assessing the status of

```
algorithm search;
begin
    unmark all nodes in N;
    mark node s;
    pred(s) : = 0;
    next : = 1;
    order(s): = s;
    LIST : = {s}
    while LIST ≠ Ø do
    begin
        select a node i in LIST;
        if node i is incident to an admissible arc (i, j) then
        begin
            mark node j;
            pred(j) : = i;
            next : = next + 1;
            order(j) : = next;
            add node j to LIST;
        end
        else delete node i from LIST;
    end;
end;
```

Figure 3.4 Search algorithm.

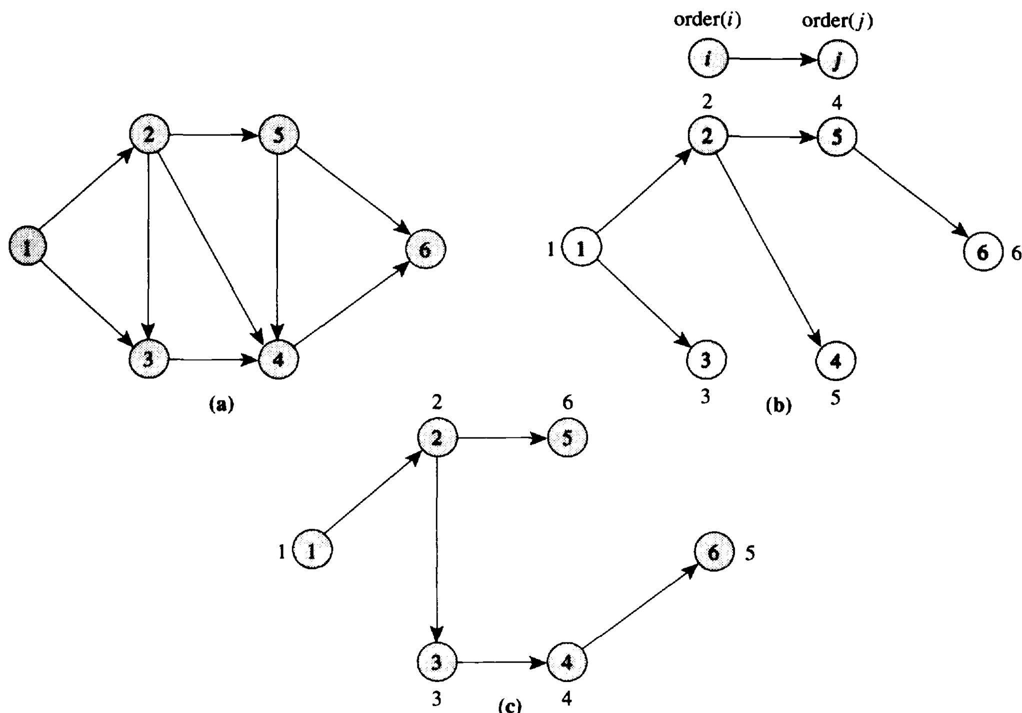

Figure 3.5 Two search trees of a network.

their incident nodes. In later chapters, too, we need the same data structure to implement maximum flow and minimum cost flow algorithms. We use the *current-arc* data structure, defined as follows, for this purpose. We maintain with each node i the adjacency list $A(i)$ of arcs emanating from it (see Section 2.2 for the definition of adjacency list). For each node i, we define a *current arc* (i, j), which is the next candidate arc that we wish to examine. Initially, the current arc of node i is the first arc in $A(i)$. The search algorithm examines the list $A(i)$ sequentially: At any stage, if the current arc is inadmissible, the algorithm designates the next arc in the arc list as the current arc. When the algorithm reaches the end of the arc list, it declares that the node has no admissible arc. Note that the order in which the algorithm examines the nodes depends on how we have arranged the arcs in the arc adjacency lists $A(i)$. We assume here, as well as elsewhere in this book, that we have ordered the arcs in $A(i)$ in the increasing order of their head nodes [i.e., if (i, j) and (i, k) are two consecutive arcs in $A(i)$, then $j < k$].

It is easy to show that the search algorithm runs in $O(m + n) = O(m)$ time. Each iteration of the while loop either finds an admissible arc or does not. In the former case, the algorithm marks a new node and adds it to LIST, and in the latter case it deletes a marked node from LIST. Since the algorithm marks any node at most once, it executes the while loop at most $2n$ times. Now consider the effort spent in identifying the admissible arcs. For each node i, we scan the arcs in $A(i)$ at most once. Therefore, the search algorithm examines a total of $\sum_{i \in N} |A(i)| = m$ arcs, and thus terminates in $O(m)$ time.

The algorithm, as described, does not specify the manner for examining the nodes or for adding the nodes to LIST. Different rules give rise to different search techniques. Two data structures have proven to be the most popular for maintaining LIST—a *queue* and a *stack* (see Appendix A for a discussion of these data structures)—and they give rise to two fundamental search strategies: *breadth-first search* and *depth-first search.*

Breadth-First Search

If we maintain the set LIST as a queue, we always select nodes from the front of LIST and add them to the rear. In this case the search algorithm selects the marked nodes in a first-in, first-out order. If we define the distance of a node i as the minimum number of arcs in a directed path from node s to node i, this kind of search first marks nodes with distance 1, then those with distance 2, and so on. Therefore, this version of search is called a *breadth-first search* and the resulting search tree is a *breadth-first search tree.* Figure 3.5(b) specifies the breadth-first search tree for the network shown in Figure 3.5(a). In subsequent chapters we use the following property of the breadth-first search tree whose proof is left as an exercise (see Exercise 3.30).

Property 3.3. *In the breadth-first search tree, the tree path from the source node s to any node i is a shortest path (i.e., contains the fewest number of arcs among all paths joining these two nodes).*

Depth-First Search

If we maintain the set LIST as a stack, we always select the nodes from the front of LIST and also add them to the front. In this case the search algorithm selects the marked node in a last-in, first-out order. This algorithm performs a deep probe, creating a path as long as possible, and backs up one node to initiate a new probe when it can mark no new node from the tip of the path. Consequently, we call this version of search a *depth-first search* and the resulting tree a *depth-first search tree.* The depth-first traversal of a network is also called its *preorder traversal.* Figure 3.5(c) gives the depth-first search tree for the network shown in Figure 3.5(a).

In subsequent chapters we use the following property of the depth-first search tree, which can be easily proved using induction arguments (see Exercise 3.32).

Property 3.4

(a) *If node j is a descendant of node i and $j \neq i$, then $order(j) > order(i)$.*
(b) *All the descendants of any node are ordered consecutively in sequence.*

Reverse Search Algorithm

The search algorithm described in Figure 3.4 allows us to identify all the nodes in a network that are reachable from a given node s by directed paths. Suppose that we wish to identify all the nodes in a network from which we can reach a given node t along directed paths. We can solve this problem by using the algorithm we have

just described with three slight changes: (1) we initialize LIST as LIST $= \{t\}$; (2) while examining a node, we scan the incoming arcs of the node instead of its outgoing arcs; and (3) we designate an arc (i, j) as admissible if i is unmarked and j is marked. We subsequently refer to this algorithm as a *reverse search algorithm*. Whereas the (forward) search algorithm gives us a directed out-tree rooted at node s, the reverse search algorithm gives us a directed in-tree rooted at node t.

Determining Strong Connectivity

Recall from Section 2.2 that a network is strongly connected if for every pair of nodes i and j, the network contains a directed path from node i to node j. This definition implies that a network is strongly connected if and only if for any arbitrary node s, every node in G is reachable from s along a directed path and, conversely, node s is reachable from every other node in G along a directed path. Clearly, we can determine the strong connectivity of a network by two applications of the search algorithm, once applying the (forward) search algorithm and then the reverse search algorithm.

We next consider the problem of finding a topological ordering of the nodes of an acyclic network. We will show how to solve this problem by using a minor modification of the search algorithm.

Topological Ordering

Let us label the nodes of a network $G = (N, A)$ by distinct numbers from 1 through n and represent the labeling by an array *order* [i.e., order(i) gives the label of node i]. We say that this labeling is a *topological ordering* of nodes if every arc joins a lower-labeled node to a higher-labeled node. That is, for every arc $(i, j) \in A$, order(i) < order(j). For example, for the network shown in Figure 3.6(a), the labeling shown in Figure 3.6(b) is not a topological ordering because (5, 4) is an arc and order(5) > order(4). However, the labelings shown in Figure 3.6(c) and (d) are topological orderings. As shown in this example, a network might have several topological orderings.

Some networks cannot be topologically ordered. For example, the network shown in Figure 3.7 has no such ordering. This network is cyclic because it contains a directed cycle and for any directed cycle W we can never satisfy the condition order(i) < order(j) for each $(i, j) \in W$. Indeed, acyclic networks and topological ordering are closely related. A network that contains a directed cycle has no topological ordering, and conversely, we shall show next that a network that does not contain any negative cycle can be topologically ordered. This observation shows that a network is acyclic if an only if it possesses a topological ordering of its nodes.

By using a search algorithm, we can either detect the presence of a directed cycle or produce a topological ordering of the nodes. The algorithm is fairly easy to describe. In the network G, select any node of zero indegree. Give it a label of 1, and then delete it and all the arcs emanating from it. In the remaining subnetwork select *any* node of zero indegree, give it a label of 2, and then delete it and all arcs emanating from it. Repeat this process until no node has a zero indegree. At this point, if the remaining subnetwork contains some nodes and arcs, the network G

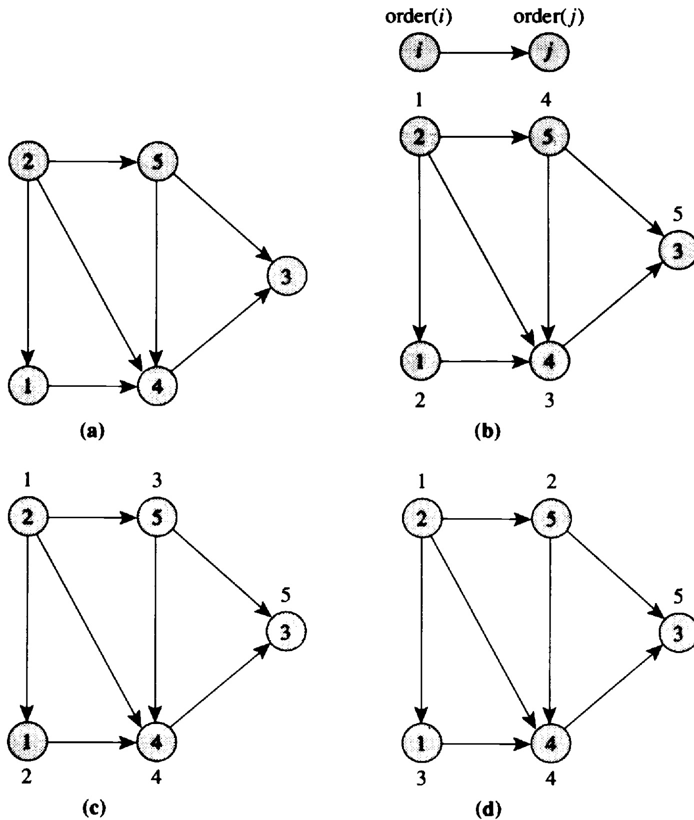

Figure 3.6 Topological ordering of nodes.

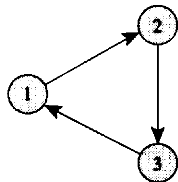

Figure 3.7 Network without a topological ordering of the nodes.

contains a directed cycle (see Exercise 3.38). Otherwise, the network is acyclic and we have assigned labels to all the nodes. Now notice that whenever we assign a label to a node at an iteration, the node has only outgoing arcs and they all must necessarily point to nodes that will be assigned higher labels in subsequent iterations. Consequently, this labeling gives a topological ordering of nodes.

We now describe an efficient implementation of this algorithm that runs in $O(m)$ time. Figure 3.8 specifies this implementation. This algorithm first computes the indegrees of all nodes and forms a set LIST consisting of all nodes with zero indegrees. At every iteration we select a node i from LIST, for every arc $(i, j) \in A(i)$ we reduce the indegree of node j by 1 unit, and if indegree of node j becomes zero, we add node j to the set LIST. [Observe that deleting the arc (i, j) from the

```
algorithm topological ordering;
begin
    for all i ∈ N do indegree(i) : = 0;
    for all (i, j) ∈ A do indegree(j) : = indegree(j) + 1;
    LIST : = Ø;
    next : = 0;
    for all i ∈ N do
        if indegree(i) = 0 then LIST : = LIST ∪ {i};
    while LIST ≠ Ø do
    begin
        select a node i from LIST and delete it;
        next : = next + 1;
        order(i) : = next;
        for all (i, j) ∈ A(i) do
        begin
            indegree(j) : = indegree(j) - 1;
            if indegree(j) = 0 then LIST : = LIST ∪ {j};
        end;
    end;
    if next < n then the network contains a directed cycle
    else the network is acyclic and the array order gives a topological order of nodes;
end;
```

Figure 3.8 Topological ordering algorithm.

network is equivalent to decreasing the indegree of node j by 1 unit.] Since the algorithm examines each node and each arc of the network $O(1)$ times, it runs in $O(m)$ time.

3.5 FLOW DECOMPOSITION ALGORITHMS

In formulating network flow problems, we can adopt either of two equivalent modeling approaches: We can define flows on arcs (as discussed in Section 1.2) or define flows on paths and cycles. For example, the arc flow shown in Figure 3.9(a) sends 7 units of flow from node 1 to node 6. Figure 3.9(b) shows a path and cycle flow

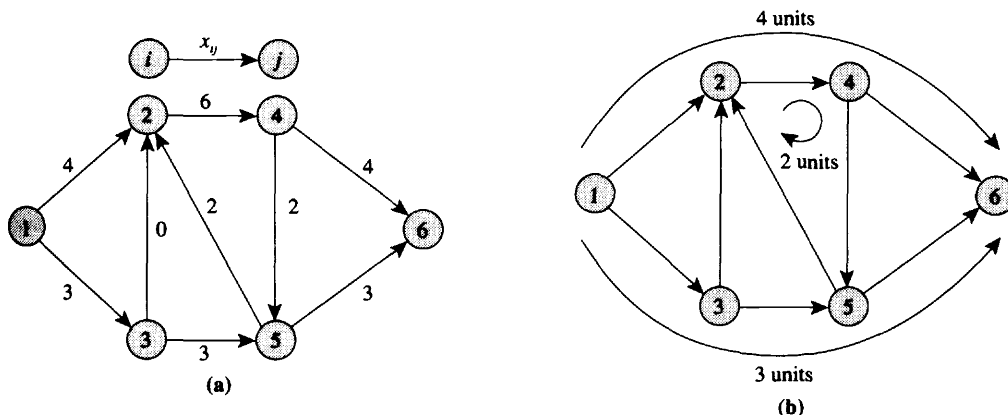

Figure 3.9 Two ways to express flows in a network: (a) using arc flows; (b) using path and cycle flows.

corresponding to this arc flow: In the path and cycle flow, we send 4 units along the path 1–2–4–6, 3 units along the path 1–3–5–6, and 2 units along the cycle 2–4–5–2. Throughout most of this book, we use the arc flow formulation; on a few occasions, however, we need to use the path and cycle flow formulation or results that stem from this modeling perspective. In this section we develop several connections between these two alternative formulations.

In this discussion, by an "arc flow" we mean a vector $x = \{x_{ij}\}$ that satisfies the following constraints:

$$\sum_{\{j:(i,j)\in A\}} x_{ij} - \sum_{\{j:(j,i)\in A\}} x_{ji} = -e(i) \qquad \text{for all } i \in N, \tag{3.8a}$$

$$0 \le x_{ij} \le u_{ij} \qquad \text{for all } (i, j) \in A. \tag{3.8b}$$

where $\sum_{i=1}^{n} e(i) = 0$. Notice that in this model we have replaced the supply/demand $b(i)$ of node i by another term, $-e(i)$; we refer to $e(i)$ as the node's *imbalance*. We have chosen this alternative modeling format purposely because some of the maximum flow and minimum cost flow algorithms described in this book maintain a solution that satisfies the flow bound constraints, but not necessarily the supply/demand constraints. The term $e(i)$ represents the inflow minus outflow of node i. If the inflow is more than outflow, $e(i) > 0$ and we say that node i is an *excess node*. If inflow is less than the outflow, $e(i) < 0$ and we say that node i is a *deficit node*. If the inflow equals outflow, we say that node i is a *balanced* node. Observe that if $e = -b$, the flow x is feasible for the minimum cost flow problem.

In the arc flow formulation discussed in Section 1.2, the basic decision variables are flows x_{ij} on the arcs $(i, j) \in A$. The path and cycle flow formulation starts with an enumeration of all directed paths P between any pair of nodes and all directed cycles W of the network. We let $\mathcal{P}$ denote the collection of all paths and $\mathcal{W}$ the collection of all cycles. The decision variables in the path and cycle flow formulation are $f(P)$, the flow on path P, and $f(W)$, the flow on cycle W; we define these variables for every directed path P in $\mathcal{P}$ and every directed cycle W in $\mathcal{W}$.

Notice that every set of path and cycle fows uniquely determines arc flows in a natural way: The flow x_{ij} on arc (i, j) equals the sum of the flows $f(P)$ and $f(W)$ for all paths P and cycles W that contain this arc. We formalize this observation by defining some new notation: $\delta_{ij}(P)$ equals 1 if arc (i, j) is contained in the path P, and is 0 otherwise. Similarly, $\delta_{ij}(W)$ equals 1 if arc (i, j) is contained in the cycle W, and is 0 otherwise. Then

$$x_{ij} = \sum_{P\in\mathcal{P}} \delta_{ij}(P)f(P) + \sum_{W\in\mathcal{W}} \delta_{ij}(W)f(W).$$

Thus each path and cycle flow determines arc flows uniquely. Can we reverse this process? That is, can we decompose any arc flow into (i.e., represent it as) path and cycle flow? The following theorem provides an affirmative answer to this question.

Theorem 3.5 (Flow Decomposition Theorem). *Every path and cycle flow has a unique representation as nonnegative arc flows. Conversely, every nonnegative arc flow x can be represented as a path and cycle flow (though not necessarily uniquely) with the following two properties:*

(a) *Every directed path with positive flow connects a deficit node to an excess node.*
(b) *At most $n + m$ paths and cycles have nonzero flow; out of these, at most m cycles have nonzero flow.*

Proof. In the light of our previous observations, we need to establish only the converse assertions. We give an algorithmic proof to show how to decompose any arc flow x into a path and cycle flow. Suppose that i_0 is a deficit node. Then some arc (i_0, i_1) carries a positive flow. If i_1 is an excess node, we stop; otherwise, the mass balance constraint (3.8a) of node i_1 implies that some other arc (i_1, i_2) carries positive flow. We repeat this argument until we encounter an excess node or we revisit a previously examined node. Note that one of these two cases will occur within n steps. In the former case we obtain a directed path P from the deficit node i_0 to some excess node i_k, and in the latter case we obtain a directed cycle W. In either case the path or the cycle consists solely of arcs with positive flow. If we obtain a directed path, we let $f(P) = \min\{-e(i_0), e(i_k), \min\{x_{ij} : (i, j) \in P\}\}$ and redefine $e(i_0) = e(i_0) + f(P)$, $e(i_k) = e(i_k) - f(P)$, and $x_{ij} = x_{ij} - f(P)$ for each arc (i, j) in P. If we obtain a directed cycle W, we let $f(W) = \min\{x_{ij} : (i, j) \in W\}$ and redefine $x_{ij} = x_{ij} - f(W)$ for each (i, j) in W.

We repeat this process with the redefined problem until all node imbalances are zero. Then we select any node with at least one outgoing arc with a positive flow as the starting node, and repeat the procedure, which in this case must find a directed cycle. We terminate when $x = 0$ for the redefined problem. Clearly, the original flow is the sum of flows on the paths and cycles identified by this method. Now observe that each time we identify a directed path, we reduce the excess/deficit of some node to zero or the flow on some arc to zero; and each time we identify a directed cycle, we reduce the flow on some arc to zero. Consequently, the path and cycle representation of the given flow x contains at most $n + m$ directed paths and cycles, and at most m of these are directed cycles. ◆

Let us consider a flow x for which $e(i) = 0$ for all $i \in N$. Recall from Section 1.2 that we call any such flow a circulation. When we apply the flow decomposition algorithm to a circulation, each iteration discovers a directed cycle consisting solely of arcs with positive flow, and subsequently reduces the flow on at least one arc to zero. Consequently, a circulation decomposes into flows along at most m directed cycles.

Property 3.6. *A circulation x can be represented as cycle flow along at most m directed cycles.*

We illustrate the flow decomposition algorithm on the example shown in Figure 3.10(a). Initially, nodes 1 and 5 are deficit nodes. Suppose that the algorithm selects node 5. We would then obtain the directed path 5–3–2–4–6 and the flow on this path is 3 units. Removing this path flow gives the flow given in Figure 3.10(b). The algorithm selects node 1 as the starting node and obtains the path flow of 2 units along the directed path 1–2–4–5–6. In the third iteration, the algorithm identifies a cycle flow of 4 units along the directed cycle 5–3–4–5. Now the flow becomes zero and the algorithm terminates.

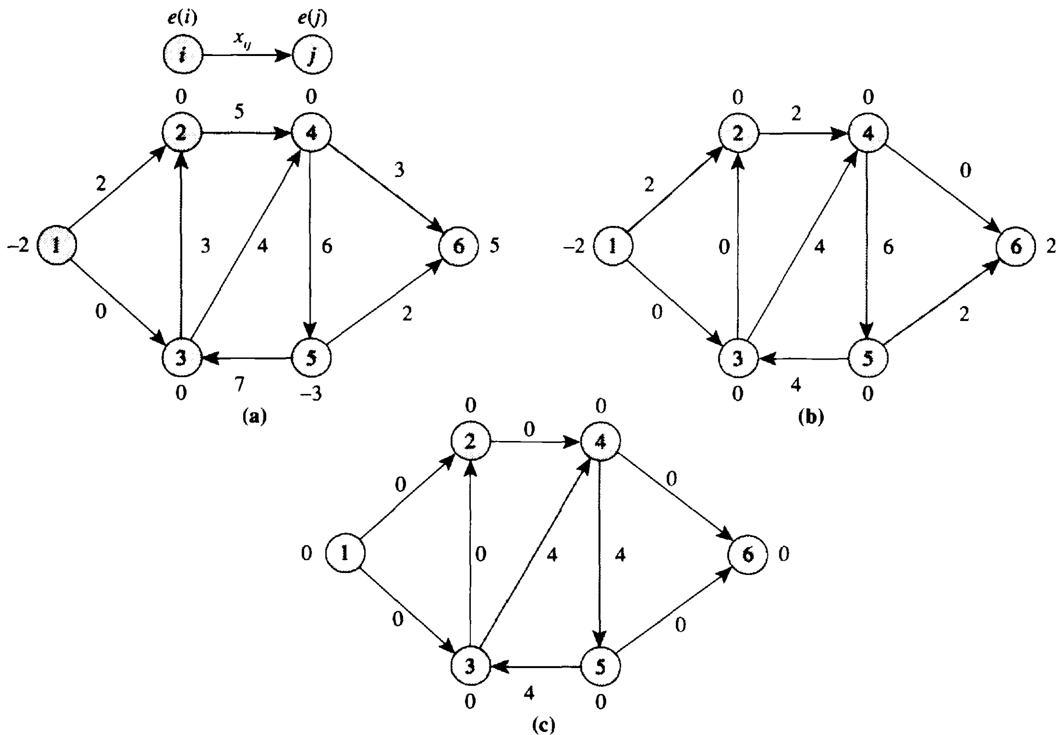

Figure 3.10 Illustrating the flow decomposition theorem.

What is the time required for the flow decomposition algorithm described in the proof of Theorem 3.5? In the algorithm, we first construct a set LIST of deficit nodes. We maintain LIST as a doubly linked list (see Appendix A for a description of this data structure) so that selection of an element as well as addition and deletion of an element require $O(1)$ time. As the algorithm proceeds, it removes nodes from LIST. When LIST eventually becomes empty, we initialize it as the set of arcs with positive flow. Consider now another basic operation in the flow decomposition algorithm: identifying an arc with positive flow emanating from a node. We refer to such arcs as *admissible* arcs. We use the *current-arc* data structure (described in Section 3.4) to identify an admissible arc emanating from a node. Notice that in any iteration, the flow decomposition algorithm requires $O(n)$ time plus the time spent in scanning arcs to identify admissible arcs. Also notice that since arc flows are nonincreasing, an arc found to be inadmissible in one iteration remains inadmissible in subsequent iterations. Consequently, we preserve the current arcs of the nodes in the current-arc data structure when we proceed from one iteration to the next. Since the current-arc data structure requires a total of $O(m)$ time in arc scanning to identify admissible arcs and the algorithm performs at most $(n + m)$ iterations, the flow decomposition algorithm runs in $O(m + (n + m)n) = O(nm)$ time.

The flow decomposition theorem has a number of important consequences. As one example, it enables us to compare any two solutions of a network flow problem in a particularly convenient way and to show how we can build one solution from

another by a sequence of simple operations. The augmenting cycle theorem, to be discussed next, highlights these ideas.

We begin by introducing the concept of augmenting cycles with respect to a flow x. A cycle W (not necessarily directed) in G is called an *augmenting cycle* with respect to the flow x if by augmenting a positive amount of flow $f(W)$ around the cycle, the flow remains feasible. The augmentation increases the flow on forward arcs in the cycle W and decreases the flow on backward arcs in the cycle. Therefore, a cycle W is an augmenting cycle in G if $x_{ij} < u_{ij}$ for every forward arc (i, j) and $x_{ij} > 0$ for every backward arc (i, j). We next extend the notation of $\delta_{ij}(W)$ for cycles that are not necessarily directed. We define $\delta_{ij}(W)$ equal to 1 if arc (i, j) is a forward arc in the cycle W, $\delta_{ij}(W)$ equal to -1 if arc (i, j) is a backward arc in the cycle W, and equal to 0 otherwise.

Notice that in terms of residual networks (defined in Section 2.4), each augmenting cycle W with respect to a flow x corresponds to a directed cycle W in $G(x)$, and vice versa. We define the cost of an augmenting cycle W as $c(W) = \sum_{(i,j)\in W} c_{ij}\delta_{ij}(W)$. The cost of an augmenting cycle represents the change in the cost of a feasible solution if we augment 1 unit of flow along the cycle. The change in flow cost for augmenting $f(W)$ units along the cycle W is $c(W)f(W)$.

We next use the flow decomposition theorem to prove an augmenting cycle theorem formulated in terms of residual networks. Suppose that x and x° are any two feasible solutions of the minimum cost flow problem. We have seen earlier that some feasible circulation x^1 in $G(x^\circ)$ satisfies the property that $x = x^\circ + x^1$. Property 3.6 implies that we can represent the circulation x^1 as cycle flows $f(W_1)$, $f(W_2), \ldots, f(W_r)$, with $r \le m$. Notice that each of the cycles $W_1, W_2, \ldots, W_r$ is an augmenting cycle in $G(x^\circ)$. Furthermore, we see that

$$\begin{aligned}\sum_{(i,j)\in A} c_{ij}x_{ij} &= \sum_{(i,j)\in A} c_{ij}x^\circ_{ij} + \sum_{(i,j)\in G(x^\circ)} c_{ij}x^1_{ij} \\ &= \sum_{(i,j)\in A} c_{ij}x^\circ_{ij} + \sum_{(i,j)\in G(x^\circ)} c_{ij}\left[\sum_{k=1}^{r} \delta_{ij}(W_k)f(W_k)\right] \\ &= \sum_{(i,j)\in A} c_{ij}x^\circ_{ij} + \sum_{k=1}^{r} c(W_k)f(W_k).\end{aligned}$$

We have thus established the following result:

Theorem 3.7 (Augmenting Cycle Theorem). *Let x and x° be any two feasible solutions of a network flow problem. Then x equals x° plus the flow on at most m directed cycles in $G(x^\circ)$. Furthermore, the cost of x equals the cost of x° plus the cost of flow on these augmenting cycles.* ◆

In Section 9.3 we see that the augmenting cycle theorem permits us to obtain the following novel characterization of the optimal solutions of the minimum cost flow problem.

Theorem 3.8 (Negative Cycle Optimality Theorem). *A feasible solution x^* of the minimum cost flow problem is an optimal solution if and only if the residual network $G(x^*)$ contains no negative cost directed cycle.*

3.6 SUMMARY

The design and analysis of algorithms is an expansive topic that has grown in importance over the past 30 years as computers have become more central to scientific and administrative computing. In this chapter we described several fundamental techniques that are widely used for this purpose. Having some way to measure the performance of algorithms is critical for comparing algorithms and for determining how well they perform. The research community has adopted three basic approaches for measuring the performance of an algorithm: empirical analysis, average-case analysis, and worst-case analysis. Each of these three performance measures has its own merits and drawbacks. Worst-case analysis has become a widely used approach, due in large part to the simplicity and theoretical appeal of this type of analysis. A worst-case analysis typically assumes that each arithmetic and logical operation requires unit time, and it provides an upper bound on the time taken by an algorithm (correct to within a constant factor) for solving *any* instance of a problem. We refer to this bound, which we state in big O notation as a function of the problem's size parameters n, m, $\log C$, and $\log U$, as the worst-case complexity of the algorithm. This bound gives the growth rate (in the worst case) that the algorithm requires for solving successively larger problems. If the worst-case complexity of an algorithm is a polynomial function of n, m, $\log C$, and $\log U$, we say that the algorithm is a polynomial-time algorithm; otherwise, we say that it is an exponential-time algorithm. Polynomial-time algorithms are preferred to exponential-time algorithms because polynomial-time algorithms are asymptotically (i.e., for sufficiently large networks) faster than exponential-time algorithms. Among several polynomial-time algorithms for the same problem, we prefer an algorithm with the least order polynomial running time because this algorithm will be asymptotically fastest.

A commonly used approach for obtaining the worst-case complexity of an iterative algorithm is to obtain a bound on the number of iterations, a bound on the number of steps per iteration, and take the product of these two bounds. Sometimes this method overestimates the actual number of steps, especially when an iteration might be easy most of the time, but expensive occasionally. In these situations, arguments based on potential functions (see Section 3.3) often allow us to obtain a tighter bound on an algorithm's required computations.

In this chapter we described four important approaches that researchers frequently use to obtain polynomial-time algorithms for network flow problems: (1) geometric improvement, (2) scaling, (3) dynamic programming, and (4) binary search. Researchers have recently found the scaling approach to be particularly useful for solving network flow problems efficiently, and currently many of the fastest network flow algorithms use scaling as an algorithmic strategy.

Search algorithms lie at the core of many network flow algorithms. We described search algorithms for performing the following tasks: (1) identifying all nodes that are reachable from a specified source node via directed paths, (2) identifying all nodes that can reach a specified sink node via directed paths, and (3) identifying whether a network is strongly connected. Another important application of search algorithms is to determine whether a given directed network is acyclic and, if so, to number the nodes in a topological order [i.e., so that $i < j$ for every arc $(i, j) \in A$]. This algorithm is a core subroutine in methods for project planning (so called

CPM/PERT models) that practitioners use extensively in many industrial settings. All of these search algorithms run in $O(m)$ time. Other $O(m)$ search algorithms are able (1) to identify whether a network is disconnected and if so to identify all of its components, and (2) to identify whether a network is bipartite. We discuss these algorithms in the exercises for this chapter.

We concluded this chapter by studying flow decomposition theory. This theory shows that we can formulate flows in a network in two alternative ways: (1) flows on arcs, or (2) flows along directed paths and directed cycles. Although we use the arc flow formulation throughout most of this book, sometimes we need to rely on the path and cycle flow formulation. Given a path and cycle flow, we can obtain the corresponding arc flow in a straightforward manner (to obtain the flow on any arc, add the flow on this arc in each path and cycle); finding path and cycle flows that corresponds to a set of given arc flows is more difficult. We described an $O(nm)$ algorithm that permits us to find these path and cycle flows. One important consequence of flow decomposition theory is the fact that we can transform any feasible flow of the minimum cost flow problem into any other feasible flow by sending flows along at most m augmenting cycles. We used this result to derive a negative cycle optimality condition for characterizing optimal solutions for the minimum cost flow problem. These conditions state that a flow x is optimal if and only if the residual network $G(x)$ contains no negative cost augmenting cycle.

REFERENCE NOTES

Over the past two decades, worst-case complexity (see Section 3.2) has become a very popular approach for analyzing algorithms. A number of books provide excellent treatments of this topic. The book by Garey and Johnson [1979] is an especially good source of information concerning the topics we have considered. Books by Aho, Hopcroft, and Ullman [1974], Papadimitriou and Steiglitz [1982], Tarjan [1983], and Cormen, Leiserson, and Rivest [1990] provide other valuable treatments of this subject matter.

The techniques used to develop polynomial-time algorithms (see Section 3.3) fall within the broad domain of algorithm design. Books on algorithms and data structures offer extensive coverage of this topic. Edmonds and Karp [1972] and Dinic [1973] independently discovered the scaling technique and its use for obtaining polynomial-time algorithms for the minimum cost flow problem. Gabow [1985] popularized the scaling technique by developing scaling-based algorithms for the shortest path, maximum flow, assignment, and matching problems. This book is the first that emphasizes scaling as a generic algorithmic tool. The geometric improvement technique is a combinatorial analog of linear convergence in the domain of nonlinear programming. For a study of linear convergence, we refer the reader to any book in nonlinear programming. Dynamic programming, which was first developed by Richard Bellman, has proven to be a very successful algorithmic tool. Some important sources of information on dynamic programming are books by Bellman [1957], Bertsekas [1976], and Denardo [1982]. Binary search is a standard technique in searching and sorting; Knuth [1973b] and many other books on data structures and algorithms develop this subject.

Search algorithms are important subroutines for network optimization algo-

rithms. The books by Aho, Hopcroft, and Ullman [1974], Even [1979], Tarjan [1983], and Cormen, Leiserson, and Rivest [1990] present insightful treatments of search algorithms. Ford and Fulkerson [1962] developed flow decomposition theory; their book contains additional material on this topic.

EXERCISES

3.1. Write a pseudocode that, for any integer n, computes n^n by performing at most 2 log n multiplications. Assume that multiplying two numbers, no matter how large, requires one operation.

3.2. Compare the following functions for various values of n and determine the approximate values of n when the second function becomes larger than the first.
(a) $1000n^2$ and $2^n/100$.
(b) $(\log n)^3$ and $n^{0.001}$.
(c) $10{,}000n$ and $0.1n^2$.

3.3. Rank the following functions in increasing order of their growth rates.
(a) $2^{\log \log n}$, $n!$, n^2, 2^n, $(1.5)^{(\log n)^2}$.
(b) $1000(\log n)^2$, $0.005n^{0.0001}$, $\log \log n$, $(\log n)(\log \log n)$.

3.4. Rank the following functions in increasing order of their growth rates for two cases: (1) when a network containing n nodes and m arcs is connected and very sparse [i.e., $m = O(n)$]; and (2) when the network is very dense [i.e., $m = \Omega(n^2)$].
(a) $n^2m^{1/2}$, $nm + n^2 \log n$, $nm \log n$, $nm \log(n^2/m)$.
(b) n^2, $m \log n$, $m + n \log n$, $m \log \log n$.
(c) $n^3 \log n$, $(m \log n)(m + n \log n)$, $nm(\log \log n)\log n$.

3.5. We say that a function $f(n)$ is $O(g(n))$ if for some numbers c and n_0, $f(n) \le cg(n)$ for all $n \ge n_0$. Similarly, we say that a function is $\Omega(g(n))$ if for some numbers c' and n_0, $f(n) \ge c'g(n)$ for infinitely many $n \ge n_0$. Finally, we say that a function $f(n)$ is $\Theta(g(n))$ if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$. For each of the functions $f(n)$ and $g(n)$ specified below, indicate whether $f(n)$ is $O(g(n))$, $\Omega(g(n))$, $\Theta(g(n))$, or none of these.
(a) $f(n) = \begin{cases} n & \text{if } n \text{ is odd} \\ n^2 & \text{if } n \text{ is even} \end{cases}$; $g(n) = \begin{cases} n & \text{if } n \text{ is even} \\ n^2 & \text{if } n \text{ is odd} \end{cases}$
(b) $f(n) = \begin{cases} n & \text{if } n \text{ is odd} \\ n^2 & \text{if } n \text{ is even} \end{cases}$; $g(n) = \begin{cases} n & \text{if } n \text{ is prime} \\ n^2 & \text{if } n \text{ is not prime} \end{cases}$
(c) $f(n) = 3 + 1/(\log n)$; $g(n) = (n + 4)/(n + 3)$

3.6. Are the following statements true or false?
(a) $(\log n)^{100} = O(n^\epsilon)$ for any $\epsilon > 0$.
(b) $2^{n+1} = O(2^n)$.
(c) $f(n) + g(n) = O(\max(f(n), g(n)))$.
(d) If $f(n) = O(g(n))$, then $g(n) = \Omega(f(n))$.

3.7. Let $g(n, m) = m \log_d n$, where $d = \lceil m/n + 2 \rceil$. Show that for any $\epsilon > 0$, $g(n, m) = O(m^{1+\epsilon})$.

3.8. Show that if $f(n) = O(g(n))$ and $g(n) = O(h(n))$, then $f(n) = O(h(n))$. Is it true that if $f(n) = \Omega(g(n))$ and $g(n) = \Omega(h(n))$, then $f(n) = \Omega(h(n))$? Prove or disprove this statement.

3.9. **Bubble sort.** The *bubble sort algorithm* is a popular method for sorting n numbers in nondecreasing order of their magnitudes. The algorithm maintains an ordered set of the numbers $\{a_1, a_2, \ldots, a_n\}$ that it rearranges through a sequence of several passes over the set. In each pass, the algorithm examines every pair of elements (a_k, a_{k+1}) for each $k = 1,\ldots,(n_{-1})$, and if the pair is out of order (i.e., $a_k > a_{k+1}$), it swaps the positions of these elements. The algorithm terminates when it makes no swap during one entire pass. Show that the algorithm performs at most n passes and runs in $O(n^2)$ time. For every n, construct a sorting problem (i.e., the initial ordered set of numbers $\{a_1,$

$a_2, \ldots, a_n\}$ so that the algorithm performs $\Omega(n^2)$ operations. Conclude that the bubble sort is a $\Theta(n^2)$ algorithm.

3.10. **Bin packing problem.** The bin packing problem requires that we pack n items of lengths $a_1, a_2, \ldots, a_n$ (assume that each $a_i \leq 1$) into bins of unit length using the minimum possible number of bins. Several approximate methods, called *heuristics*, are available for solving the bin packing problem. The *first-fit heuristic* is one of the more popular of these heuristics. It works as follows. Arrange items in an arbitrary order and examine them one by one in this order. For an item being examined, scan the bins one by one and put the item in the bin where it fits first. If an items fits in none of the bins that currently contain an item, we introduce a new bin and place the item in it. Write a pseudocode for the first-fit heuristic and show that it runs in $O(n^2)$ time. For every n, construct an instance of the bin packing problem for which your first-fit heuristic runs in $\Omega(n^2)$ time. Conclude that the first-fit heuristic runs in $\Theta(n^2)$ time.

3.11. Consider a queue of elements on which we perform two operations: (1) *insert(i)*, which adds an element i to the rear of the queue; and (2) *delete(k)*, which deletes the k frontmost elements from the queue. Show that an arbitrary sequence of n insert and delete operations, starting with an empty queue, requires a total of $O(n)$ time.

3.12. An algorithm performs three different operations. The first and second operations are executed $O(nm)$ and $O(n^2)$ times respectively and the number of executions of the third operation is yet to be determined. These operations have the following impact on an appropriately defined potential function ϕ: Each execution of operation 1 increases ϕ by at most n units, each execution of operation 2 increases ϕ by 1 unit, and each execution of operation 3 decreases ϕ by at least 1 unit. Suppose we know that $1 \leq \phi \leq n^2$. Obtain a bound on the number of executions of the third operation.

3.13. **Parameter balancing.** For each of the time bounds stated below as a function of the parameter k, use the parameter balancing technique to determine the value of k that yields the minimum time bound. Also try to determine the optimal value of k using differential calculus.

(a) $O\left(\frac{n^3}{k} + knm\right)$

(b) $O\left(nk + \frac{m}{k}\right)$

(c) $O\left(\frac{m \log n}{\log k} + \frac{n\, k \log n}{\log k}\right)$

3.14. **Generalized parameter balancing.** In Section 3.3 we discussed the parameter balancing technique for situations when the time bound contains two expressions. In this exercise we generalize the technique to bounds containing three expressions. Suppose that the running time of an algorithm is $O(f(n, k) + g(n, k) + h(n, k))$ and we wish to determine the optimal value of k—that is, the value of k producing the smallest possible overall time. Assume that for all k, $f(n, k)$, $g(n, k)$, and $h(n, k)$ are all nonnegative, $f(n, k)$ is monotonically increasing, and both $g(n, k)$ and $h(n, k)$ are monotonically decreasing. Show how to obtain the optimal value of k and prove that your method is valid. Illustrate your technique on the following time bounds: (1) $kn^2 + n^3/k + n^4/k^2$; (2) $nm/k + kn^2 + n^2 \log_k U$.

3.15. In each of the algorithms described below, use Theorem 3.1 to obtain an upper bound on the total number of iterations the algorithm performs.

(a) Let v^* denote the maximum flow value and v the flow value of the current solution in a maximum flow algorithm. This algorithm increases the flow value by an amount $(v^* - v)/m$ at each iteration. How many iterations will this algorithm perform?

(b) Let z^* and z represent the optimal objective function value and objective function value of the current solution in an application of the some algorithm for solving the shortest path problem. Suppose that this algorithm ensures that each iteration decreases the objective function value by at least $(z - z^*)/2n^2$. How many iterations will the algorithm perform?

3.16. Consider a function $f(n, m)$, defined inductively as follows:

$$f(n, 0) = n, \quad f(0, m) = 2m, \quad \text{and}$$

$$f(n, m) = f(n - 1, m) + f(n, m - 1) - f(n - 1, m - 1).$$

Derive the values of $f(n, m)$ for all values of $n, m \le 4$. Simplify the definition of $f(n, m)$ and prove your result using inductive arguments.

3.17. In Section 3.3 we described a dynamic programming algorithm for the 0–1 knapsack problem. Generalize this approach so that it can be used to solve a knapsack problem in which we can place more than one item of the same type in the knapsack.

3.18. **Shortest paths in layered networks.** We say that a directed network $G = (N, A)$ with a specified source node s and a specified sink node t is *layered* if we can partition its node set N into k layers $N_1, N_2, \ldots, N_k$ so that $N_1 = \{s\}$, $N_k = \{t\}$, and for every arc $(i, j) \in A$, nodes i and j belong to adjacent layers (i.e., $i \in N_l$ and $j \in N_{l+1}$ for some $1 \le l \le k - 1$). Suggest a dynamic programming algorithm for solving the shortest path problem in a layered network. What is the running time of your algorithm? (*Hint*: Examine nodes in the layers $N_1, N_2, \ldots, N_k$, in order and compute shortest path distances.)

3.19. Let $G = (N, A)$ be a directed network. We want to determine whether G contains an odd-length directed cycle passing through node i. Show how to solve this problem using dynamic programming. [*Hint:* Define $d^k(j)$ as equal to 1 if the network contains a walk from node i to node j with exactly k arcs, and as 0 otherwise. Use recursion on k.]

3.20. Now consider the problem of determining whether a network contains an even-length directed cycle passing through node i. Explain why the approach described in Exercise 3.19 does not work in this case.

3.21. Consider a network with a length c_{ij} associated with each arc (i, j). Give a dynamic programming algorithm for finding a shortest walk (i.e., of minimum total length) containing exactly k arcs from a specified node s to every other node j in a network. Does this algorithm work in the presence of negative cycles? [*Hint*: Define $d^k(j)$ as the length of the shortest walk from node s to node j containing exactly k arcs and write a recursive relationship for $d^k(j)$ in terms of $d^{k-1}(j)$ and c_{ij}'s.]

3.22. Professor May B. Wright suggests the following sorting method utilizing a binary search technique. Consider a list of n numbers and suppose that we have already sorted the first k numbers in the list (i.e., arranged them in the nondecreasing order). At the $(k + 1)$th iteration, select the $(k + 1)$th number in the list, perform binary search over the first k numbers to identify the position of this number, and then insert it to produce the sorted list of the first $k + 1$ elements. Professor Wright claims that this method runs in $O(n \log n)$ time because it performs n iterations and each binary search requires $O(\log n)$ time. Unfortunately, Professor Wright's claim is false and it is not possible to implement the algorithm in $O(n \log n)$ time. Explain why. (*Hint*: Work out the details of this implementation including the required data structures.)

3.23. Given a convex function $f(x)$ of the form shown in Figure 3.11, suppose that we want to find a value of x that minimizes $f(x)$. Since locating the exact minima is a difficult task, we allow some approximation and wish to determine a value x so that the interval $(x - \epsilon, x + \epsilon)$ contains a value that minimizes $f(x)$. Suppose that we know that $f(x)$

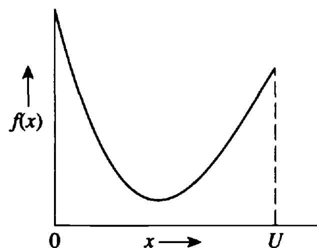

Figure 3.11 Convex function.

attains its minimum value in the interval $[0, U]$. Develop a binary search algorithm for solving this problem that runs in $O(\log(U/\epsilon))$ time. (*Hint*: At any iteration when $[a, b]$ is the feasible interval, evaluate $f(x)$ at the points $(a + b)/4$ and $3(a + b)/4$, and exclude the region $[a, (a + b)/4]$ or $[3(a + b)/4, b]$.)

3.24. (a) Determine the breadth-first and depth-first search trees with $s = 1$ as the source node for the graph shown in Figure 3.12.

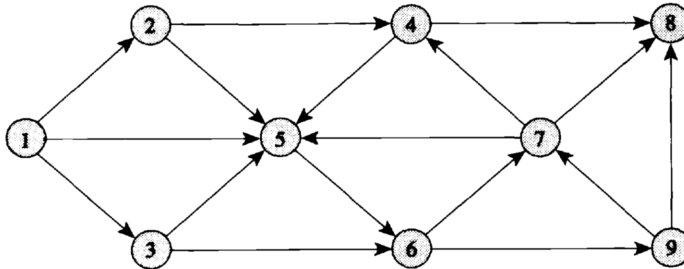

Figure 3.12 Example for Exercise 3.24.

(b) Is the graph shown in Figure 3.12 acyclic? If not, what is the minimum number of arcs whose deletion will produce an acyclic graph? Determine a topological ordering of the nodes in the resulting graph. Is the topological ordering unique?

3.25. Knight's tour problem. Consider the chessboard shown in Figure 3.13. Note that some squares are shaded. We wish to determine a knight's tour, if one exists, that starts at the square designated by s and, after visiting the minimum number of squares, ends at the square designated by t. The tour must not visit any shaded square. Formulate this problem as a reachability problem on an appropriately defined graph.

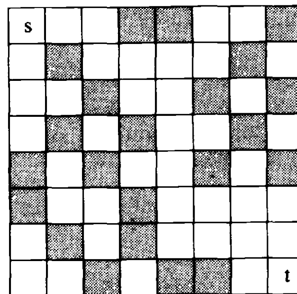

Figure 3.13 Chessboard.

3.26. Maze problem. Show how to formulate a maze problem as a reachability problem in a directed network. Illustrate your method on the maze problem shown in Figure 3.14. (*Hint*: Define rectangular segments in the maze as *cords* and represent cords by nodes.)

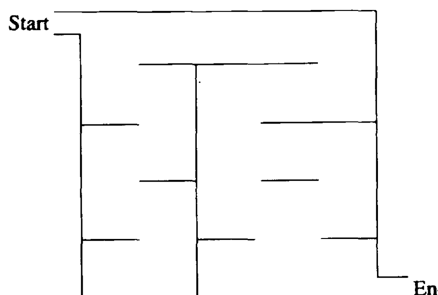

Figure 3.14 Maze.

3.27. **Wine division problem.** Two men have an 8-gallon jug full of wine and two empty jugs with a capacity of 5 and 3 gallons. They want to divide the wine into two equal parts. Suppose that when shifting the wine from one jug to another, in order to know how much they have transferred, the men must always empty out the first jug or fill the second, or both. Formulate this problem as a reachability problem in an appropriately defined graph. (*Hint*: Let a, b, and c, respectively, denote a partitioning of the 8 gallons of wine into the jugs of 8, 5, and 3 gallons capacity. Refer to any such partitioning as a feasible state of the jugs. Since at least one of the jugs is always empty or full, we can define 16 possible feasible states. Suppose that we represent these states by nodes and connect two nodes by an arc when we can permissibly move wine from one jug to another to move from one state to the other.)

3.28. Give a five-node network for which a breadth-first traversal examines the nodes in the same order as a depth-first traversal.

3.29. Let T be a depth-first search tree of an undirected graph G. Show that for every nontree arc (k, l) in G, either node k is an ancestor of node l in T or node l is an ancestor of node k in T. Show by a counterexample that a breadth-first search tree need not satisfy this property.

3.30. Show that in a breadth-first search tree, the tree path from the source node to any node i is a shortest path (i.e., contains the fewest number of arcs among all paths joining these two nodes). (*Hint*: Use induction on the number of labeled nodes.)

3.31. In an undirected graph $G = (N, A)$, a set of nodes $S \subseteq N$ defines a *clique* if for every pair of nodes i, j in S, $(i, j) \in A$. Show that in the depth-first tree of G, all nodes in any clique S appear on one path. Do all the nodes in a clique S appear consecutively on the path?

3.32. Show that a depth-first order of a network satisfies the following properties.
(a) If node j is a descendant of node i, order(j) > order(i).
(b) All the descendants of any node are ordered consecutively in the order sequence.

3.33. Show that a directed network G is either strongly connected or contains a cut $[S, \bar{S}]$ having no arc (i, j) with $i \in S$ and $j \in \bar{S}$.

3.34. We define the diameter of a graph as a longest path (i.e., one containing the largest number of arcs) in the graph: The path can start and end at any node. Construct a graph whose diameter equals the longest path in a depth-first search tree (you can select any node as the source node). Construct another graph whose diameter is strictly less than the longest path in some depth-first search tree.

3.35. **Transitive closure.** A *transitive closure* of a graph $G = (N, A)$ is a matrix $\tau = \{\tau_{ij}\}$ defined as follows:

$$\tau_{ij} = \begin{cases} 1 & \text{if the graph G contains a directed path from node } i \text{ to node } j \\ 0 & \text{otherwise.} \end{cases}$$

Give an $O(nm)$ algorithm for constructing the transitive closure of a (possibly cyclic) graph G.

3.36. Let $\mathcal{H} = \{h_{ij}\}$ denote the node–node adjacency matrix of a graph G. Consider the following set of statements:

```
for l: = 1 to n - 1 do
  for k: = 1 to n do
    for j: = 1 to n do
      for i: = 1 to n do
        h_ij: = max{h_ij, h_ik, h_kj};
```

Show that at the end of these computations, the matrix $\mathcal{H}$ represents the transitive closure of G.

3.37. Given the transitive closure of a graph G, describe an $O(n^2)$ algorithm for determining all strongly connected components of the graph.

3.38. Show that in a directed network, if each node has indegree at least one, the network contains a directed cycle.

3.39. Show through an example that a network might have several topological orderings of its nodes. Show that the topological ordering of a network is unique if and only if the network contains a simple directed path passing through all of its nodes.

3.40. Given two n-vectors $(\alpha(1), \alpha(2), \ldots, \alpha(n))$ and $(\beta(1), \beta(2), \ldots, \beta(n))$, we say that α is *lexicographically smaller* than β (i.e., $\alpha \leq \beta$) if for the first index k for which $\alpha(k) \neq \beta(k)$, $\alpha(k)$ is less than $\beta(k)$. [For example, (2, 4, 8) is lexicographically smaller than (2, 5, 1).] Modify the algorithm given in Figure 3.8 so that it gives the lexico-minimum topological ordering of its nodes (i.e., a topological ordering that is lexicographically smaller than every other topological ordering).

3.41. Suggest an $O(m)$ algorithm for identifying all components of a (possibly) disconnected graph. Design the algorithm so that it will assign a label 1 to all nodes in the first component, a label 2 to all nodes in the second component, and so on. (*Hint*: Maintain a doubly linked list of all unlabeled node.)

3.42. Consider an (arbitrary) spanning tree T of a graph G. Show how to label each node in T as 0 or 1 so that whenever arc (i, j) is contained in the tree, nodes i and j have different labels. Using this result, prove that G is bipartite if and only if for every nontree arc (k, l), nodes k and l have different labels. Using this characterization, describe an $O(m)$ algorithm for determining whether a graph is bipartite or not.

3.43. In an acyclic network $G = (N, A)$ with a specified source node s, let $\alpha(i)$ denote the number of distinct paths from node s to node i. Give an $O(m)$ algorithm that determines $\alpha(i)$ for all $i \in N$. (*Hint*: Examine nodes in a topological order.)

3.44. For an acyclic network G with a specified source node s, outline an algorithm that enumerates *all* distinct directed paths from the source node to every other node in the network. The running time of your algorithm should be proportional to the total length of all the paths enumerated (i.e., linear in terms of the output length.) (*Hint*: Extend your method developed in Exercise 3.43.)

3.45. In an undirected connected graph $G = (N, A)$, an *Euler tour* is a walk that starts at some node, visits each arc exactly once, and returns to the starting node. A graph is *Eulerian* if it contains an Euler tour. Show that in an Eulerian graph, the degree of every node is even. Next, show that if every node in a connected graph has an even degree, the graph is Eulerian. Establish the second result by describing an $O(m)$ algorithm for determining whether a graph is Eulerian and, if so, will construct an Euler tour. (*Hint*: Describe an algorithm that decomposes any graph with only even-degree nodes into a collection of arc-disjoint cycles, and then converts the cycles into an Euler tour.)

3.46. Let T be a depth-first search tree of a graph. Let $D(i)$ denote an ordered set of descendants of the node $i \in T$, arranged in the same order in which the depth-first search method labeled them. Define *last(i)* as the last element in the set $D(i)$. Modify the depth-first search algorithm so that while computing the depth-first traversal of the network G, it also computes the last index of every node. Your algorithm should run in $O(m)$ time.

3.47. **Longest path in a tree** (Handler, 1973). A longest path in an undirected tree T is a path containing the maximum number of arcs. The longest path can start and end anywhere. Show that we can determine a longest path in T as follows: Select any node i and use a search algorithm to find a node k farthest from node i. Then use a search algorithm to find a node l farthest from node k. Show that the tree path from node k to node l is a longest path in T. (*Hint*: Consider the midmost node or arc on any longest path in the tree depending on whether the path contains an even or odd number or arcs. Need the longest path starting from any node j pass through this node or arc?)

3.48. Consider the flow given in Figure 3.15(a). Compute the imbalance $e(i)$ for each node $i \in N$ and decompose the flow into a path and cycle flow. Is this decomposition unique?

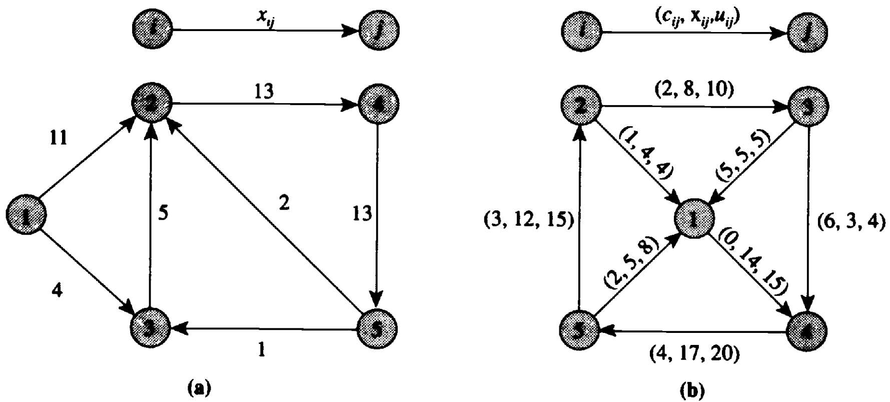

Figure 3.15 Examples for Exercises 3.48 and 3.49.

3.49. Consider the circulation given in Figure 3.15(b). Decompose this circulation into flows along directed cycles. Draw the residual network and use Theorem 3.8 to check whether the flow is an optimal solution of the minimum cost flow problem.

3.50. Consider the circulation shown in Figure 3.16. Show that there are $k!$ distinct flow decompositions of this circulation.

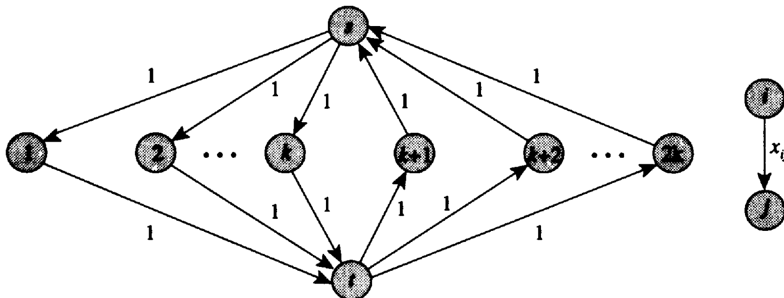

Figure 3.16 Example for Exercise 3.50.

3.51. Show that a unit flow along directed walk from node i to node $j (i \neq j)$ containing any arc at most once can be decomposed into a directed path from node i to node j plus some arc-disjoint directed cycles. Next, show that a unit flow along a closed directed walk can be decomposed into unit flows along arc-disjoint directed cycles.

3.52. Show that if an undirected connected graph $G = (N, A)$ contains exactly $2k$ odd-degree nodes, the graph contains k arc-disjoint walks $P_1, P_2, \ldots, P_k$ satisfying the property that $A = P_1 \cup P_2 \cup \cdots \cup P_k$.

3.53. Let $G = (N, A)$ be a connected network in which every arc $(i, j) \in A$ has positive lower bound $l_{ij} > 0$ and an infinite upper bound $u_{ij} = \infty$. Show that G contains a feasible circulation (i.e., a flow in which the inflow equals the outflow for every node) if and only if G is strongly connected.

3.54. Show that a solution x satisfying the flow bound constraints is a circulation if and only if the net flow across any cut is zero.

Simplex method,
  for bounded variables, 814–15
  for generalized flows, 583–89
  for linear programming, 810–19
  for maximum flows, 430–33
  for minimum cost flows, 415–21
  for shortest paths, 425–30
  generalized upper bounding, 666–67
  revised, 813–14
Simplex multipliers
  for linear programs, 808
  for minimum cost flows, 445–46
Ski instructor's problem, 501
Small-capacity networks, 289
Sollin's algorithm, 526–28, 534
Solving systems of equations, 199
Sorting, 86, 521, 774, 778
Spanning subgraph, 26
Spanning tree, 30
Spanning tree solutions, 405–09
Spanning tree structures, 408–09
Stable marriage problem, 473–75
Stable matchings, 475
Stable university admissions, 507
Stacks
  applications, 64–65
Statistical security of data, 199, 283–85
Steiner tree problem, 642
Stick percolation problem, 550–51
Storage policy for libraries, 344–45
Strong connectivity
  algorithm, 77
  definition, 27
Strong duality theorem
  for linear programs, 818–19
  for minimum cost flows, 312–13
Strongly feasible solutions, 421–25, 432, 457, 590
  and perturbation, 457
Subgradient optimization
  application to multicommodity flows, 663–65
  technique, 611–15
Subgraph, 26
Subset systems, 528–30
Subtour breaking constraints, 626
Successive shortest path algorithm
  applications, 360, 437, 471, 556, 639, 701
  basic approach, 320–24, 340
Succint certificate, 794
Symmetric difference, 477
System of difference constraints, 103–05, 127, 726–28

Tail nodes, 25
Tanker scheduling problems, 176–77, 347, 656
Telephone operator scheduling, 105–06, 127
Teleprocessing design problem, 632
Temporarily labeled nodes, 109
Terminal assignment problem, 346
Thread index, 410–14, 443–46
Threshold algorithm, 161
Time complexity function, 58
Time-cost trade-off problem, 735–37
Time-expanded networks, 737–40
Topological ordering
  algorithm, 77–79
  applications, 11, 107–08, 371–72
Totally unimodular matrices, 448–49
Tournament problem, 12
Traffic flows, 547
Tramp steamer problem, 103, 150
Transfers in communication networks, 547–48
Transformations
  for removing arc capacities, 40
  for removing nonzero lower bounds, 39
  for removing undirected arcs, 39
  node splitting, 41–43
Transitive closure, 90, 91
Transportation problem, 7, 9, 20, 294
Travelling salesman problem. See TSP
Tree arcs, 30
Tree indices, 410–14, 419, 576
Tree of shortest paths, 106, 139
Trees, 28–30
Triangularity property, 443–47
Triple operation, 147
Truck scheduling problem, 763
TSP, 623–25, 643–44, 790–91, 794, 797

Uncapacitated networks, 40–41
Undirected networks
  definitions, 25, 31
  representations, 38
  transformation, 39
Unimodular matrices, 447–49
Unimodularity property, 447–49
Union-find operation, 522
Unique label property, 481–82
Unit capacity networks
  and bipartite matchings, 469–70
  and minimum cost flows, 399
  and network connectivity, 188–91, 274
  maximum flows in, 252–55, 285, 289
Unstable roommates, 507

Validity conditions, 209
Variable splitting, 630
Variational principle, 16, 547
Vehicle fleet planning, 344
Vehicle routing, 625–27, 645–47
Virtual running times, 707–09
Vital arcs, 128–29, 244

Walk, 26
Warehousing problem, 570, 655
Wave algorithm, 246
Weak duality theorem
  for Lagrangian relaxation, 606
  for linear programs, 817–18
  for minimum cost flow, 312
Wine division problem, 90
Worst-case complexity, 56–66

Zero length cycle, 151, 160
Zoned warehousing, 345