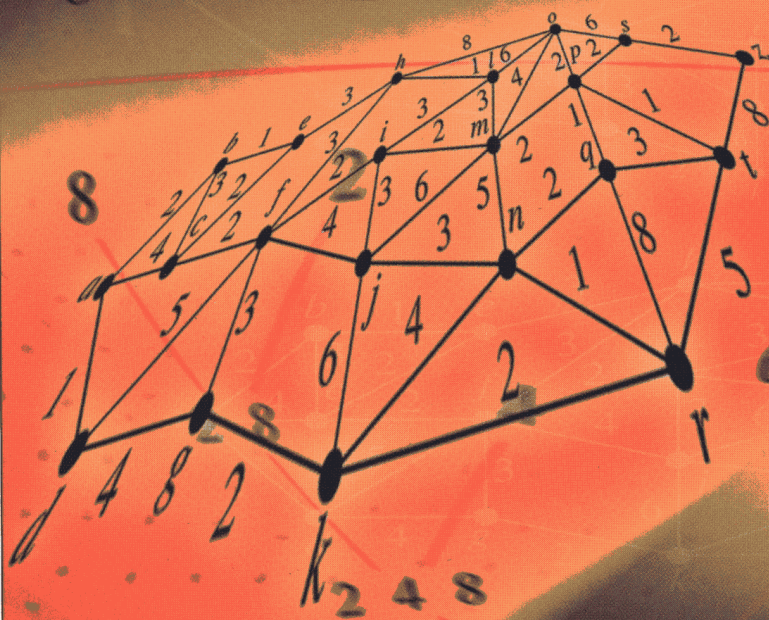


DISCRETE MATHEMATICS AND ITS APPLICATIONS

SIXTH EDITION



Kenneth H. Rosen

McGraw-Hill International Edition

LIST OF SYMBOLS

TOPIC	SYMBOL	MEANING	PAGE	
LOGIC	$\neg p$	negation of p	3	
	$p \wedge q$	conjunction of p and q	4	
	$p \vee q$	disjunction of p and q	4	
	$p \oplus q$	exclusive or of p and q	5	
	$p \rightarrow q$	the implication p implies q	6	
	$p \leftrightarrow q$	biconditional of p and q	9	
	$p \equiv q$	equivalence of p and q	22	
	T	tautology	24	
	F	contradiction	24	
	$P(x_1, \dots, x_n)$	propositional function	32	
	$\forall x P(x)$	universal quantification of $P(x)$	34	
	$\exists x P(x)$	existential quantification of $P(x)$	36	
	$\exists! x P(x)$	uniqueness quantification of $P(x)$	37	
	\therefore	therefore	63	
	$p\{S\}q$	partial correctness of S	323	
	SETS	$x \in S$	x is a member of S	112
		$x \notin S$	x is not a member of S	112
$\{a_1, \dots, a_n\}$		list of elements of a set	112	
$\{x \mid P(x)\}$		set builder notation	112	
N		set of natural numbers	112	
Z		set of integers	112	
Z⁺		set of positive integers	113	
Q		set of rational numbers	113	
R		set of real numbers	113	
$S = T$		set equality	113	
\emptyset		the empty (or null) set	114	
$S \subseteq T$		S is a subset of T	114	
$S \subset T$		S is a proper subset of T	115	
$ S $		cardinality of S	116	
$P(S)$		the power set of S	116	
(a_1, \dots, a_n)		n -tuple	117	
(a, b)		ordered pair	117	
$A \times B$		Cartesian product of A and B	118	
$A \cup B$		union of A and B	121	
$A \cap B$		intersection of A and B	121	
$A - B$		the difference of A and B	123	
\overline{A}		complement of A	123	
$\bigcup_{i=1}^n A_i$		union of $A_i, i = 1, 2, \dots, n$	127	
$\bigcap_{i=1}^n A_i$		intersection of $A_i, i = 1, 2, \dots, n$	128	
$A \oplus B$		symmetric difference of A and B	131	

TOPIC	SYMBOL	MEANING	PAGE
FUNCTIONS	$f(a)$	value of the function f at a	133
	$f:A \rightarrow B$	function from A to B	133
	$f_1 + f_2$	sum of the functions f_1 and f_2	135
	$f_1 f_2$	product of the functions f_1 and f_2	135
	$f(S)$	image of the set S under f	136
	$\iota_A(s)$	identity function on A	138
	$f^{-1}(x)$	inverse of f	139
	$f \circ g$	composition of f and g	140
	$\lfloor x \rfloor$	floor function of x	143
	$\lceil x \rceil$	ceiling function of x	143
	a_n	term of $\{a_i\}$ with subscript n	150
	$\sum_{i=1}^n a_i$	sum of a_1, a_2, \dots, a_n	153
	$\sum_{\alpha \in S} a_\alpha$	sum of a_α over $\alpha \in S$	156
	$\prod_{i=1}^n a_n$	product of a_1, a_2, \dots, a_n	162
	$f(x)$ is $O(g(x))$	$f(x)$ is big- O of $g(x)$	180
	$n!$	n factorial	185
	$f(x)$ is $\Omega(g(x))$	$f(x)$ is big- Ω of $g(x)$	189
	$f(x)$ is $\Theta(g(x))$	$f(x)$ is big- Θ of $g(x)$	189
	\sim	asymptotic	192
	$\min(x, y)$	minimum of x and y	216
	$\max(x, y)$	maximum of x and y	217
\approx	approximately equal to	395	
INTEGERS	$a \mid b$	a divides b	201
	$a \nmid b$	a does not divide b	201
	$a \text{ div } b$	quotient when a is divided by b	202
	$a \text{ mod } b$	remainder when a is divided by b	202
	$a \equiv b \pmod{m}$	a is congruent to b modulo m	203
	$a \not\equiv b \pmod{m}$	a is not congruent to b modulo m	203
	$\gcd(a, b)$	greatest common divisor of a and b	215
	$\text{lcm}(a, b)$	least common multiple of a and b	217
	$(a_k a_{k-1} \dots a_1 a_0)_b$	base b representation	219
	MATRICES	$[a_{ij}]$	matrix with entries a_{ij}
$\mathbf{A} + \mathbf{B}$		matrix sum of \mathbf{A} and \mathbf{B}	247
\mathbf{AB}		matrix product of \mathbf{A} and \mathbf{B}	248
\mathbf{I}_n		identity matrix of order n	251
\mathbf{A}^t		transpose of \mathbf{A}	251
$\mathbf{A} \vee \mathbf{B}$		join of \mathbf{A} and \mathbf{B}	252
$\mathbf{A} \wedge \mathbf{B}$		the meet of \mathbf{A} and \mathbf{B}	252
$\mathbf{A} \odot \mathbf{B}$		Boolean product of \mathbf{A} and \mathbf{B}	253
$\mathbf{A}^{[n]}$		n th Boolean power of \mathbf{A}	254

(List of Symbols continued at back of book)

Discrete Mathematics and Its Applications

Sixth Edition

Kenneth H. Rosen

AT&T Laboratories



Boston Burr Ridge, IL Dubuque, IA New York San Francisco St. Louis
Bangkok Bogotá Caracas Kuala Lumpur Lisbon London Madrid Mexico City
Milan Montreal New Delhi Santiago Seoul Singapore Sydney Taipei Toronto



DISCRETE MATHEMATICS AND ITS APPLICATIONS, SIXTH EDITION
International Edition 2007

Exclusive rights by McGraw-Hill Education (Asia), for manufacture and export. This book cannot be re-exported from the country to which it is sold by McGraw-Hill. The International Edition is not available in North America.

Published by McGraw-Hill, a business unit of The McGraw-Hill Companies, Inc., 1221 Avenue of the Americas, New York, NY 10020. Copyright © 2007 by Kenneth H. Rosen. All rights reserved. No part of this publication may be reproduced or distributed in any form or by any means, or stored in a database or retrieval system, without the prior written consent of The McGraw-Hill Companies, Inc., including, but not limited to, in any network or other electronic storage or transmission, or broadcast for distance learning.

Some ancillaries, including electronic and print components, may not be available to customers outside the United States.

10 09 08 07 06 05 04 03 02 01
20 09 08 07 06
CTF BJE

The credits section for this book begins on page C-1 and is considered an extension of the copyright page.

When ordering this title, use ISBN-13: 978-007-124474-9 or ISBN-10: 007-124474-3

Printed in Singapore

www.mhhe.com

Contents

Preface vii

The MathZone Companion Website xviii

To the Student xx

1	The Foundations: Logic and Proofs	1
1.1	Propositional Logic	1
1.2	Propositional Equivalences	21
1.3	Predicates and Quantifiers	30
1.4	Nested Quantifiers	50
1.5	Rules of Inference	63
1.6	Introduction to Proofs	75
1.7	Proof Methods and Strategy	86
	End-of-Chapter Material	104
2	Basic Structures: Sets, Functions, Sequences, and Sums	111
2.1	Sets	111
2.2	Set Operations	121
2.3	Functions	133
2.4	Sequences and Summations	149
	End-of-Chapter Material	163
3	The Fundamentals: Algorithms, the Integers, and Matrices	167
3.1	Algorithms	167
3.2	The Growth of Functions	180
3.3	Complexity of Algorithms	193
3.4	The Integers and Division	200
3.5	Primes and Greatest Common Divisors	210
3.6	Integers and Algorithms	219
3.7	Applications of Number Theory	231
3.8	Matrices	246
	End-of-Chapter Material	257
4	Induction and Recursion	263
4.1	Mathematical Induction	263
4.2	Strong Induction and Well-Ordering	283
4.3	Recursive Definitions and Structural Induction	294
4.4	Recursive Algorithms	311

4.5	Program Correctness	322
	End-of-Chapter Material	328
5	Counting.....	335
5.1	The Basics of Counting	335
5.2	The Pigeonhole Principle.....	347
5.3	Permutations and Combinations	355
5.4	Binomial Coefficients.....	363
5.5	Generalized Permutations and Combinations.....	370
5.6	Generating Permutations and Combinations.....	382
	End-of-Chapter Material	386
6	Discrete Probability.....	393
6.1	An Introduction to Discrete Probability.....	393
6.2	Probability Theory	400
6.3	Bayes' Theorem.....	417
6.4	Expected Value and Variance	426
	End-of-Chapter Material	442
7	Advanced Counting Techniques.....	449
7.1	Recurrence Relations	449
7.2	Solving Linear Recurrence Relations.....	460
7.3	Divide-and-Conquer Algorithms and Recurrence Relations	474
7.4	Generating Functions	484
7.5	Inclusion–Exclusion	499
7.6	Applications of Inclusion–Exclusion	505
	End-of-Chapter Material	513
8	Relations.....	519
8.1	Relations and Their Properties.....	519
8.2	n -ary Relations and Their Applications	530
8.3	Representing Relations	537
8.4	Closures of Relations	544
8.5	Equivalence Relations	555
8.6	Partial Orderings	566
	End-of-Chapter Material	581
9	Graphs.....	589
9.1	Graphs and Graph Models.....	589
9.2	Graph Terminology and Special Types of Graphs	597
9.3	Representing Graphs and Graph Isomorphism.....	611
9.4	Connectivity.....	621

9.5	Euler and Hamilton Paths	633
9.6	Shortest-Path Problems	647
9.7	Planar Graphs	657
9.8	Graph Coloring	666
	End-of-Chapter Material	675
10	Trees.....	683
10.1	Introduction to Trees.....	683
10.2	Applications of Trees	695
10.3	Tree Traversal.....	710
10.4	Spanning Trees	724
10.5	Minimum Spanning Trees.....	737
	End-of-Chapter Material	743
11	Boolean Algebra.....	749
11.1	Boolean Functions.....	749
11.2	Representing Boolean Functions	757
11.3	Logic Gates.....	760
11.4	Minimization of Circuits	766
	End-of-Chapter Material	781
12	Modeling Computation.....	785
12.1	Languages and Grammars	785
12.2	Finite-State Machines with Output	796
12.3	Finite-State Machines with No Output.....	804
12.4	Language Recognition	817
12.5	Turing Machines	827
	End-of-Chapter Material	838
	Appendixes.....	A-1
A-1	Axioms for the Real Numbers and the Positive Integers	A-1
A-2	Exponential and Logarithmic Functions	A-7
A-3	Pseudocode	A-10
	<i>Suggested Readings</i>	B-1
	<i>Answers to Odd-Numbered Exercises</i>	S-1
	<i>Photo Credits</i>	C-1
	<i>Index of Biographies</i>	I-1
	<i>Index</i>	I-2

9.2 Graph Terminology and Special Types of Graphs

Introduction



We introduce some of the basic vocabulary of graph theory in this section. We will use this vocabulary later in this chapter when we solve many different types of problems. One such problem involves determining whether a graph can be drawn in the plane so that no two of its edges cross. Another example is deciding whether there is a one-to-one correspondence between the vertices of two graphs that produces a one-to-one correspondence between the edges of the graphs. We will also introduce several important families of graphs often used as examples and in models. Several important applications will be described where these special types of graphs arise.

Basic Terminology

First, we give some terminology that describes the vertices and edges of undirected graphs.

DEFINITION 1 Two vertices u and v in an undirected graph G are called *adjacent* (or *neighbors*) in G if u and v are endpoints of an edge of G . If e is associated with $\{u, v\}$, the edge e is called *incident with the vertices u and v* . The edge e is also said to *connect u and v* . The vertices u and v are called *endpoints* of an edge associated with $\{u, v\}$.

To keep track of how many edges are incident to a vertex, we make the following definition.

DEFINITION 2 The *degree of a vertex in an undirected graph* is the number of edges incident with it, except that a loop at a vertex contributes twice to the degree of that vertex. The degree of the vertex v is denoted by $\deg(v)$.

EXAMPLE 1 What are the degrees of the vertices in the graphs G and H displayed in Figure 1?

Solution: In G , $\deg(a) = 2$, $\deg(b) = \deg(c) = \deg(f) = 4$, $\deg(d) = 1$, $\deg(e) = 3$, and $\deg(g) = 0$. In H , $\deg(a) = 4$, $\deg(b) = \deg(e) = 6$, $\deg(c) = 1$, and $\deg(d) = 5$. ◀

A vertex of degree zero is called **isolated**. It follows that an isolated vertex is not adjacent to any vertex. Vertex g in graph G in Example 1 is isolated. A vertex is **pendant** if and only if it has degree one. Consequently, a pendant vertex is adjacent to exactly one other vertex. Vertex d in graph G in Example 1 is pendant.

Examining the degrees of vertices in a graph model can provide useful information about the model, as Example 2 shows.

EXAMPLE 2 What does the degree of a vertex in a niche overlap graph (introduced in Example 1 in Section 9.1) represent? Which vertices in this graph are pendant and which are isolated? Use the niche overlap graph shown in Figure 6 of Section 9.1 to interpret your answers.

Solution: There is an edge between two vertices in a niche overlap graph if and only if the two species represented by these vertices compete. Hence, the degree of a vertex in a niche overlap graph is the number of species in the ecosystem that compete with the species represented by this vertex. A vertex is pendant if the species competes with exactly one other species in the

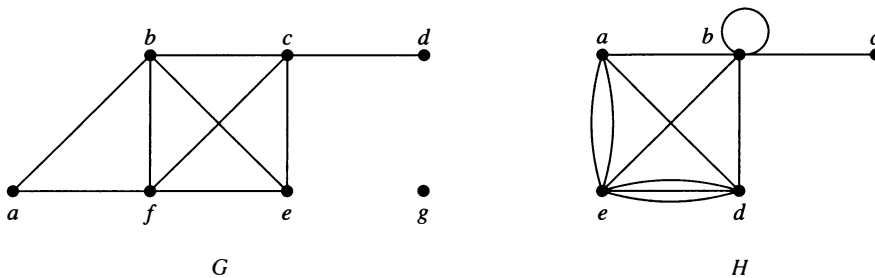


FIGURE 1 The Undirected Graphs G and H .

ecosystem. Finally, the vertex representing a species is isolated if this species does not compete with any other species in the ecosystem.

For instance, the degree of the vertex representing the squirrel in Figure 6 in Section 9.1 is four, because the squirrel competes with four other species: the crow, the opossum, the raccoon, and the woodpecker. In the graph in Figure 6, the mouse is the only species represented by a pendant vertex, because the mouse competes only with the shrew and all other species compete with at least two other species. The vertex representing a species is pendant if this species competes with only one other species. There are no isolated vertices in the graph in Figure 6 because every species in this ecosystem competes with at least one other species. ◀

What do we get when we add the degrees of all the vertices of a graph $G = (V, E)$? Each edge contributes two to the sum of the degrees of the vertices because an edge is incident with exactly two (possibly equal) vertices. This means that the sum of the degrees of the vertices is twice the number of edges. We have the result in Theorem 1, which is sometimes called the Handshaking Theorem, because of the analogy between an edge having two endpoints and a handshake involving two hands.

THEOREM 1 THE HANDSHAKING THEOREM Let $G = (V, E)$ be an undirected graph with e edges. Then

$$2e = \sum_{v \in V} \deg(v).$$

(Note that this applies even if multiple edges and loops are present.)

EXAMPLE 3 How many edges are there in a graph with 10 vertices each of degree six?

Solution: Because the sum of the degrees of the vertices is $6 \cdot 10 = 60$, it follows that $2e = 60$. Therefore, $e = 30$. ◀

Theorem 1 shows that the sum of the degrees of the vertices of an undirected graph is even. This simple fact has many consequences, one of which is given as Theorem 2.

THEOREM 2 An undirected graph has an even number of vertices of odd degree.

Proof: Let V_1 and V_2 be the set of vertices of even degree and the set of vertices of odd degree, respectively, in an undirected graph $G = (V, E)$. Then

$$2e = \sum_{v \in V} \deg(v) = \sum_{v \in V_1} \deg(v) + \sum_{v \in V_2} \deg(v).$$

Because $\deg(v)$ is even for $v \in V_1$, the first term in the right-hand side of the last equality is even. Furthermore, the sum of the two terms on the right-hand side of the last equality is even, because this sum is $2e$. Hence, the second term in the sum is also even. Because all the terms in this sum are odd, there must be an even number of such terms. Thus, there are an even number of vertices of odd degree. ◀

Terminology for graphs with directed edges reflects the fact that edges in directed graphs have directions.

DEFINITION 3 When (u, v) is an edge of the graph G with directed edges, u is said to be *adjacent to* v and v is said to be *adjacent from* u . The vertex u is called the *initial vertex* of (u, v) , and v is called the *terminal* or *end vertex* of (u, v) . The initial vertex and terminal vertex of a loop are the same.

Because the edges in graphs with directed edges are ordered pairs, the definition of the degree of a vertex can be refined to reflect the number of edges with this vertex as the initial vertex and as the terminal vertex.

DEFINITION 4 In a graph with directed edges the *in-degree of a vertex* v , denoted by $\deg^-(v)$, is the number of edges with v as their terminal vertex. The *out-degree of* v , denoted by $\deg^+(v)$, is the number of edges with v as their initial vertex. (Note that a loop at a vertex contributes 1 to both the in-degree and the out-degree of this vertex.)

EXAMPLE 4 Find the in-degree and out-degree of each vertex in the graph G with directed edges shown in Figure 2.

Solution: The in-degrees in G are $\deg^-(a) = 2$, $\deg^-(b) = 2$, $\deg^-(c) = 3$, $\deg^-(d) = 2$, $\deg^-(e) = 3$, and $\deg^-(f) = 0$. The out-degrees are $\deg^+(a) = 4$, $\deg^+(b) = 1$, $\deg^+(c) = 2$, $\deg^+(d) = 2$, $\deg^+(e) = 3$, and $\deg^+(f) = 0$. ◀

Because each edge has an initial vertex and a terminal vertex, the sum of the in-degrees and the sum of the out-degrees of all vertices in a graph with directed edges are the same. Both of these sums are the number of edges in the graph. This result is stated as Theorem 3.

THEOREM 3 Let $G = (V, E)$ be a graph with directed edges. Then

$$\sum_{v \in V} \deg^-(v) = \sum_{v \in V} \deg^+(v) = |E|.$$

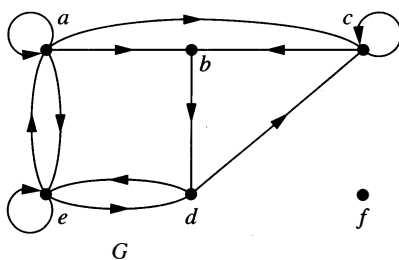


FIGURE 2 The Directed Graph G .

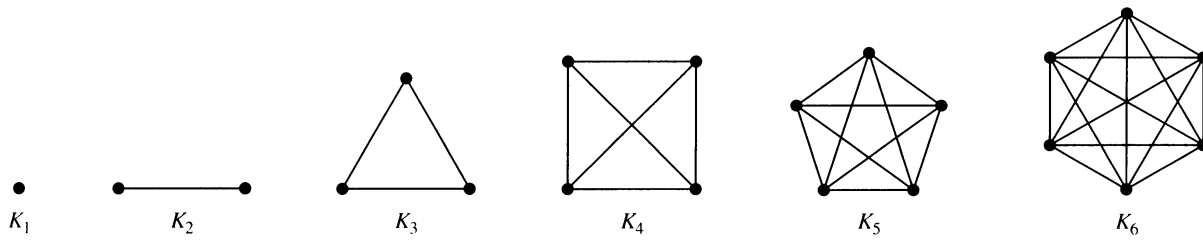


FIGURE 3 The Graphs K_n for $1 \leq n \leq 6$.

There are many properties of a graph with directed edges that do not depend on the direction of its edges. Consequently, it is often useful to ignore these directions. The undirected graph that results from ignoring directions of edges is called the **underlying undirected graph**. A graph with directed edges and its underlying undirected graph have the same number of edges.

Some Special Simple Graphs

We will now introduce several classes of simple graphs. These graphs are often used as examples and arise in many applications.

EXAMPLE 5 Complete Graphs The **complete graph on n vertices**, denoted by K_n , is the simple graph that contains exactly one edge between each pair of distinct vertices. The graphs K_n , for $n = 1, 2, 3, 4, 5, 6$, are displayed in Figure 3. ◀

EXAMPLE 6 Cycles The **cycle C_n** , $n \geq 3$, consists of n vertices v_1, v_2, \dots, v_n and edges $\{v_1, v_2\}, \{v_2, v_3\}, \dots, \{v_{n-1}, v_n\}$, and $\{v_n, v_1\}$. The cycles C_3, C_4, C_5 , and C_6 are displayed in Figure 4. ◀

EXAMPLE 7 Wheels We obtain the **wheel W_n** when we add an additional vertex to the cycle C_n , for $n \geq 3$, and connect this new vertex to each of the n vertices in C_n , by new edges. The wheels W_3, W_4, W_5 , and W_6 are displayed in Figure 5. ◀

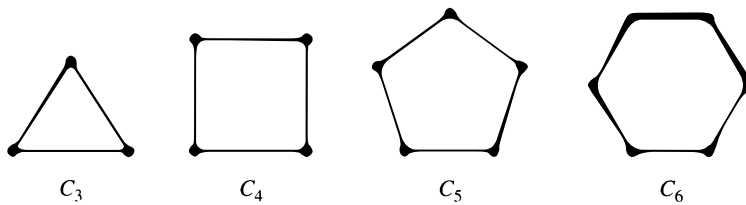


FIGURE 4 The Cycles C_3, C_4, C_5 , and C_6 .

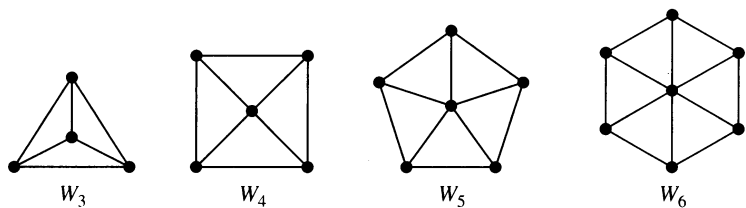


FIGURE 5 The Wheels W_3, W_4, W_5 , and W_6 .

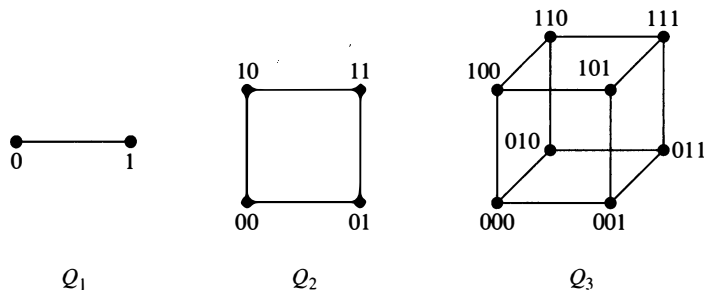


FIGURE 6 The n -cube Q_n for $n = 1, 2,$ and 3 .

EXAMPLE 8 *n -Cubes* The n -dimensional hypercube, or n -cube, denoted by Q_n , is the graph that has vertices representing the 2^n bit strings of length n . Two vertices are adjacent if and only if the bit strings that they represent differ in exactly one bit position. The graphs Q_1 , Q_2 , and Q_3 are displayed in Figure 6. Note that you can construct the $(n + 1)$ -cube Q_{n+1} from the n -cube Q_n by making two copies of Q_n , prefacing the labels on the vertices with a 0 in one copy of Q_n and with a 1 in the other copy of Q_n , and adding edges connecting two vertices that have labels differing only in the first bit. In Figure 6, Q_3 is constructed from Q_2 by drawing two copies of Q_2 as the top and bottom faces of Q_3 , adding 0 at the beginning of the label of each vertex in the bottom face and 1 at the beginning of the label of each vertex in the top face. (Here, by *face* we mean a face of a cube in three-dimensional space. Think of drawing the graph Q_3 in three-dimensional space with copies of Q_2 as the top and bottom faces of a cube and then drawing the projection of the resulting depiction in the plane.) ◀

Bipartite Graphs



Sometimes a graph has the property that its vertex set can be divided into two disjoint subsets such that each edge connects a vertex in one of these subsets to a vertex in the other subset. For example, consider the graph representing marriages between men and women in a village, where each person is represented by a vertex and a marriage is represented by an edge. In this graph, each edge connects a vertex in the subset of vertices representing males and a vertex in the subset of vertices representing females. This leads us to Definition 5.

DEFINITION 5 A simple graph G is called *bipartite* if its vertex set V can be partitioned into two disjoint sets V_1 and V_2 such that every edge in the graph connects a vertex in V_1 and a vertex in V_2 (so that no edge in G connects either two vertices in V_1 or two vertices in V_2). When this condition holds, we call the pair (V_1, V_2) a *bipartition* of the vertex set V of G .

In Example 9 we will show that C_6 is bipartite, and in Example 10 we will show that K_3 is not bipartite.

EXAMPLE 9 C_6 is bipartite, as shown in Figure 7, because its vertex set can be partitioned into the two sets $V_1 = \{v_1, v_3, v_5\}$ and $V_2 = \{v_2, v_4, v_6\}$, and every edge of C_6 connects a vertex in V_1 and a vertex in V_2 . ◀

EXAMPLE 10 K_3 is not bipartite. To verify this, note that if we divide the vertex set of K_3 into two disjoint sets, one of the two sets must contain two vertices. If the graph were bipartite, these two vertices

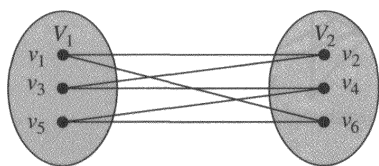


FIGURE 7 Showing That C_6 Is Bipartite.

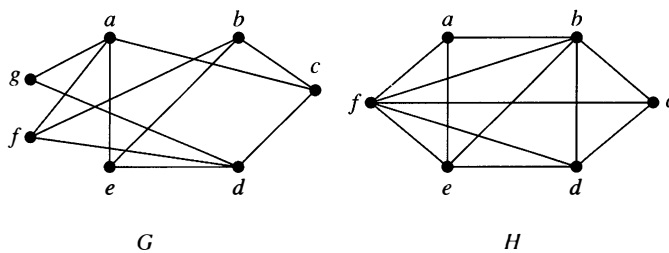


FIGURE 8 The Undirected Graphs G and H .

could not be connected by an edge, but in K_3 each vertex is connected to every other vertex by an edge. ◀

EXAMPLE 11 Are the graphs G and H displayed in Figure 8 bipartite?

Solution: Graph G is bipartite because its vertex set is the union of two disjoint sets, $\{a, b, d\}$ and $\{c, e, f, g\}$, and each edge connects a vertex in one of these subsets to a vertex in the other subset. (Note that for G to be bipartite it is not necessary that every vertex in $\{a, b, d\}$ be adjacent to every vertex in $\{c, e, f, g\}$. For instance, b and g are not adjacent.)

Graph H is not bipartite because its vertex set cannot be partitioned into two subsets so that edges do not connect two vertices from the same subset. (The reader should verify this by considering the vertices $a, b,$ and f .) ◀

Theorem 4 provides a useful criterion for determining whether a graph is bipartite.

THEOREM 4 A simple graph is bipartite if and only if it is possible to assign one of two different colors to each vertex of the graph so that no two adjacent vertices are assigned the same color.

Proof: First, suppose that $G = (V, E)$ is a bipartite simple graph. Then $V = V_1 \cup V_2$, where V_1 and V_2 are disjoint sets and every edge in E connects a vertex in V_1 and a vertex in V_2 . If we assign one color to each vertex in V_1 and a second color to each vertex in V_2 , then no two adjacent vertices are assigned the same color.

Now suppose that it is possible to assign colors to the vertices of the graph using just two colors so that no two adjacent vertices are assigned the same color. Let V_1 be the set of vertices assigned one color and V_2 be the set of vertices assigned the other color. Then, V_1 and V_2 are disjoint and $V = V_1 \cup V_2$. Furthermore, every edge connects a vertex in V_1 and a vertex in V_2 because no two adjacent vertices are either both in V_1 or both in V_2 . Consequently, G is bipartite. ◀

We illustrate how Theorem 4 can be used to determine whether a graph is bipartite in Example 12.

EXAMPLE 12 Use Theorem 4 to determine whether the graphs in Example 11 are bipartite.

Solution: We first consider the graph G . We will try to assign one of two colors, say red and blue, to each vertex in G so that no edge in G connects a red vertex and a blue vertex. Without loss of generality we begin by arbitrarily assigning red to a . Then, we must assign blue to $c, e, f,$ and g , because each of these vertices is adjacent to a . To avoid having an edge with two blue endpoints, we must assign red to all the vertices adjacent to either $c, e, f,$ or g . This means that

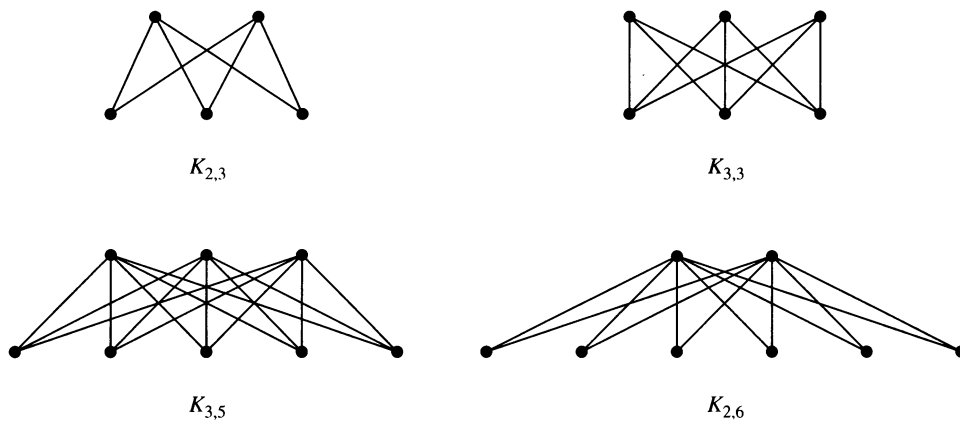


FIGURE 9 Some Complete Bipartite Graphs.

we must assign red to both b and d (and means that a must be assigned red, which it already has been). We have now assigned colors to all vertices, with a , b , and d red and c , e , f , and g blue. Checking all edges, we see that every edge connects a red vertex and a blue vertex. Hence, by Theorem 4 the graph G is bipartite.

Next, we will try to assign either red or blue to each vertex in H so that no edge in H connects a red vertex and a blue vertex. Without loss of generality we arbitrarily assign red to a . Then, we must assign blue to b , e , and f , because each is adjacent to a . But this is not possible because e and f are adjacent, so both cannot be assigned blue. This argument shows that we cannot assign one of two colors to each of the vertices of H so that no adjacent vertices are assigned the same color. It follows by Theorem 4 that H is not bipartite. ◀

Theorem 4 is an example of a result in the part of graph theory known as graph colorings. Graph colorings is an important part of graph theory with important applications. We will study graph colorings further in Section 9.8.

Another useful criterion for determining whether a graph is bipartite is based on the notion of a path, a topic we study in Section 9.4. A graph is bipartite if and only if it is not possible to start at a vertex and return to this vertex by traversing an odd number of distinct edges. We will make this notion more precise when we discuss paths and circuits in graphs in Section 9.4 (see Exercise 53 in that section).

EXAMPLE 13 Complete Bipartite Graphs The **complete bipartite graph** $K_{m,n}$ is the graph that has its vertex set partitioned into two subsets of m and n vertices, respectively. There is an edge between two vertices if and only if one vertex is in the first subset and the other vertex is in the second subset. The complete bipartite graphs $K_{2,3}$, $K_{3,3}$, $K_{3,5}$, and $K_{2,6}$ are displayed in Figure 9. ◀

Some Applications of Special Types of Graphs

We will show how bipartite graphs and special types of graphs are used in models in Examples 14–16.

EXAMPLE 14 Job Assignments Suppose that there are m employees in a group and j different jobs that need to be done where $m \leq j$. Each employee is trained to do one or more of these j jobs. We can use a graph to model employee capabilities. We represent each employee by a vertex and each job by a vertex. For each employee, we include an edge from the vertex representing that employee to the vertices representing all jobs that the employee has been trained to do.

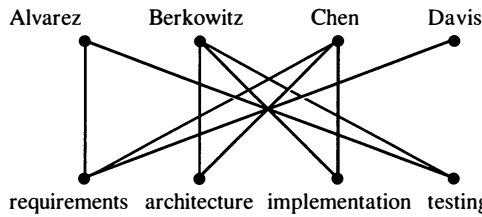


FIGURE 10 Modeling the Jobs for Which Employees Have Been Trained.

Note that the vertex set of this graph can be partitioned into two disjoint sets, the set of vertices representing employees and the set of vertices representing jobs, and each edge connects a vertex representing an employee to a vertex representing a job. Consequently, this graph is bipartite.

For example, suppose that a group has four employees: Alvarez, Berkowitz, Chen, and Davis; and suppose that four jobs need to be done to complete a project: requirements, architecture, implementation, and testing. Suppose that Alvarez has been trained to do requirements and testing; Berkowitz has been trained to do architecture, implementation, and testing; Chen has been trained to do requirements, architecture, and implementation; and Davis has only been trained to do requirements. We can model these capabilities of employees using the bipartite graph shown in Figure 10.

To complete the project, we must assign jobs to the employees so that every job has an employee assigned to it and no employee is assigned more than one job. In this case, we can assign Alvarez to do testing, Berkowitz to do implementation, Chen to do architecture, and Davis to do requirements, as shown in Figure 10 (where colored lines show this assignment of jobs). ◀

Finding an assignment of jobs to employees can be thought of as finding a matching in the graph model. A **matching** in a simple graph is a subset of the set of edges of the graph such that no two edges are incident with the same vertex; a **maximal matching** is a matching with the largest number of edges. In other words, a matching is a subset of edges such that if $\{s, t\}$ and $\{u, v\}$ are edges of the matching, then $s, t, u,$ and v are distinct. To assign jobs to employees so that the largest number of employees are assigned jobs, we seek a maximum matching in the graph that models employee capabilities. (The interested reader can find more about matchings in books about graph theory, including [GrYe06].)

EXAMPLE 15



Local Area Networks The various computers in a building, such as minicomputers and personal computers, as well as peripheral devices such as printers and plotters, can be connected using a *local area network*. Some of these networks are based on a *star topology*, where all devices are connected to a central control device. A local area network can be represented using a complete bipartite graph $K_{1,n}$, as shown in Figure 11(a). Messages are sent from device to device through the central control device.

Other local area networks are based on a *ring topology*, where each device is connected to exactly two others. Local area networks with a ring topology are modeled using n -cycles, C_n , as shown in Figure 11(b). Messages are sent from device to device around the cycle until the intended recipient of a message is reached.

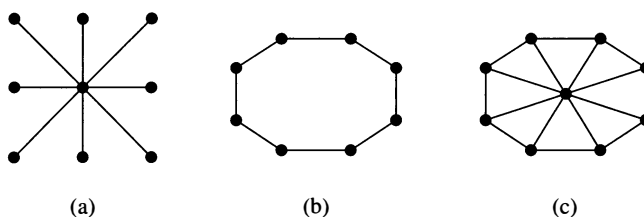


FIGURE 11 Star, Ring, and Hybrid Topologies for Local Area Networks.

Finally, some local area networks use a hybrid of these two topologies. Messages may be sent around the ring, or through a central device. This redundancy makes the network more reliable. Local area networks with this redundancy can be modeled using wheels W_n , as shown in Figure 11(c). ◀

EXAMPLE 16 Interconnection Networks for Parallel Computation For many years, computers executed programs one operation at a time. Consequently, the algorithms written to solve problems were designed to perform one step at a time; such algorithms are called **serial**. (Almost all algorithms described in this book are serial.) However, many computationally intense problems, such as weather simulations, medical imaging, and cryptanalysis, cannot be solved in a reasonable amount of time using serial operations, even on a supercomputer. Furthermore, there is a physical limit to how fast a computer can carry out basic operations, so there will always be problems that cannot be solved in a reasonable length of time using serial operations.

Parallel processing, which uses computers made up of many separate processors, each with its own memory, helps overcome the limitations of computers with a single processor. **Parallel algorithms**, which break a problem into a number of subproblems that can be solved concurrently, can then be devised to rapidly solve problems using a computer with multiple processors. In a parallel algorithm, a single instruction stream controls the execution of the algorithm, sending subproblems to different processors, and directs the input and output of these subproblems to the appropriate processors.

When parallel processing is used, one processor may need output generated by another processor. Consequently, these processors need to be interconnected. We can use the appropriate type of graph to represent the interconnection network of the processors in a computer with multiple processors. In the following discussion, we will describe the most commonly used types of interconnection networks for parallel processors. The type of interconnection network used to implement a particular parallel algorithm depends on the requirements for exchange of data between processors, the desired speed, and, of course, the available hardware.

The simplest, but most expensive, network-interconnecting processors include a two-way link between each pair of processors. This network can be represented by K_n , the complete graph on n vertices, when there are n processors. However, there are serious problems with this type of interconnection network because the required number of connections is so large. In reality, the number of direct connections to a processor is limited, so when there are a large number of processors, a processor cannot be linked directly to all others. For example, when there are 64 processors, $C(64, 2) = 2016$ connections would be required, and each processor would have to be directly connected to 63 others.

On the other hand, perhaps the simplest way to interconnect n processors is to use an arrangement known as a **linear array**. Each processor P_i , other than P_1 and P_n , is connected to its neighbors P_{i-1} and P_{i+1} via a two-way link. P_1 is connected only to P_2 , and P_n is connected only to P_{n-1} . The linear array for six processors is shown in Figure 12. The advantage of a linear array is that each processor has at most two direct connections to other processors. The disadvantage is that it is sometimes necessary to use a large number of intermediate links, called **hops**, for processors to share information.

The **mesh network** (or **two-dimensional array**) is a commonly used interconnection network. In such a network, the number of processors is a perfect square, say $n = m^2$. The n processors are labeled $P(i, j)$, $0 \leq i \leq m - 1$, $0 \leq j \leq m - 1$. Two-way links connect processor $P(i, j)$ with its four neighbors, processors $P(i \pm 1, j)$ and $P(i, j \pm 1)$, as long as these are processors in the mesh. (Note that four processors, on the corners of the mesh, have only two adjacent processors, and other processors on the boundaries have only three neighbors. Sometimes a variant of a mesh network in which every processor has exactly four connections is used; see Exercise 66 at the end of this section.) The mesh network limits the number of links for each processor. Communication between some pairs of processors requires $O(\sqrt{n}) = O(m)$

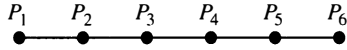


FIGURE 12 A Linear Array for Six Processors.

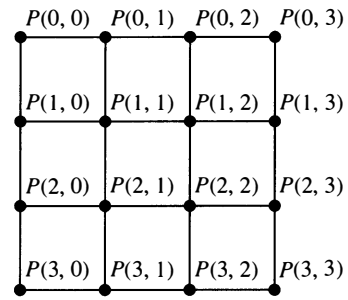


FIGURE 13 A Mesh Network for 16 Processors.

intermediate links. (See Exercise 67 at the end of this section.) The graph representing the mesh network for 16 processors is shown in Figure 13.

One important type of interconnection network is the hypercube. For such a network, the number of processors is a power of 2, $n = 2^m$. The n processors are labeled P_0, P_1, \dots, P_{n-1} . Each processor has two-way connections to m other processors. Processor P_i is linked to the processors with indices whose binary representations differ from the binary representation of i in exactly one bit. The hypercube network balances the number of direct connections for each processor and the number of intermediate connections required so that processors can communicate. Many computers have been built using a hypercube network, and many parallel algorithms have been devised that use a hypercube network. The graph Q_m , the m -cube, represents the hypercube network with $n = 2^m$ processors. Figure 14 displays the hypercube network for eight processors. (Figure 14 displays a different way to draw Q_3 than was shown in Figure 6.)

New Graphs from Old

Sometimes we need only part of a graph to solve a problem. For instance, we may care only about the part of a large computer network that involves the computer centers in New York, Denver, Detroit, and Atlanta. Then we can ignore the other computer centers and all telephone lines not linking two of these specific four computer centers. In the graph model for the large network, we can remove the vertices corresponding to the computer centers other than the four of interest, and we can remove all edges incident with a vertex that was removed. When edges and vertices are removed from a graph, without removing endpoints of any remaining edges, a smaller graph is obtained. Such a graph is called a **subgraph** of the original graph.

DEFINITION 6 A *subgraph* of a graph $G = (V, E)$ is a graph $H = (W, F)$, where $W \subseteq V$ and $F \subseteq E$. A subgraph H of G is a *proper subgraph* of G if $H \neq G$.

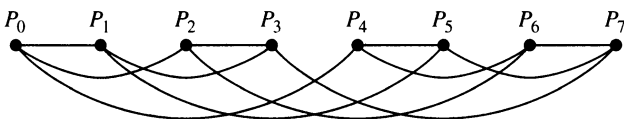


FIGURE 14 A Hypercube Network for Eight Processors.

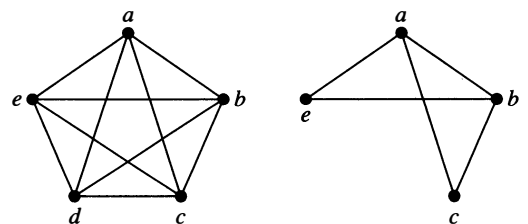


FIGURE 15 A Subgraph of K_5 .

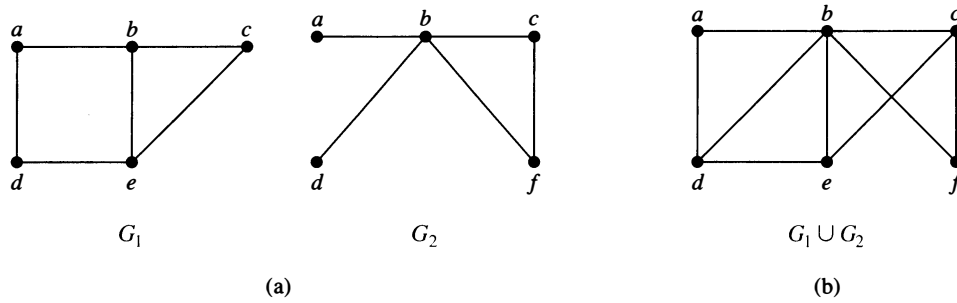


FIGURE 16 (a) The Simple Graphs G_1 and G_2 ; (b) Their Union $G_1 \cup G_2$.

EXAMPLE 17 The graph G shown in Figure 15 is a subgraph of K_5 . ◀

Two or more graphs can be combined in various ways. The new graph that contains all the vertices and edges of these graphs is called the **union** of the graphs. We will give a more formal definition for the union of two simple graphs.

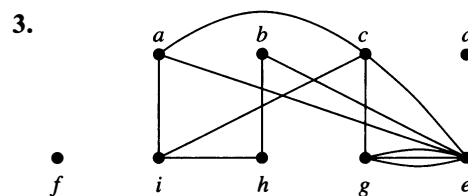
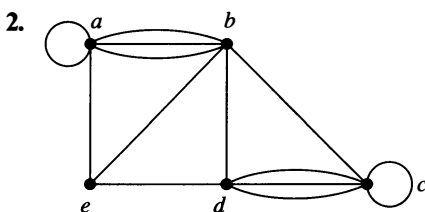
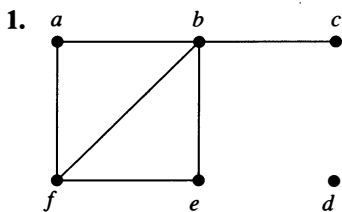
DEFINITION 7 The *union* of two simple graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ is the simple graph with vertex set $V_1 \cup V_2$ and edge set $E_1 \cup E_2$. The union of G_1 and G_2 is denoted by $G_1 \cup G_2$.

EXAMPLE 18 Find the union of the graphs G_1 and G_2 shown in Figure 16(a). ◀

Solution: The vertex set of the union $G_1 \cup G_2$ is the union of the two vertex sets, namely, $\{a, b, c, d, e, f\}$. The edge set of the union is the union of the two edge sets. The union is displayed in Figure 16(b).

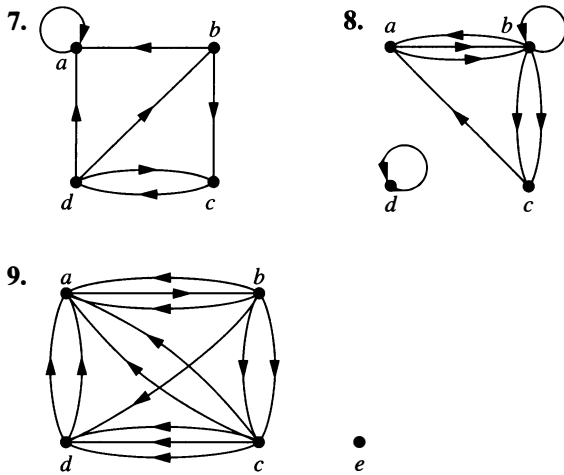
Exercises

In Exercises 1–3 find the number of vertices, the number of edges, and the degree of each vertex in the given undirected graph. Identify all isolated and pendant vertices.

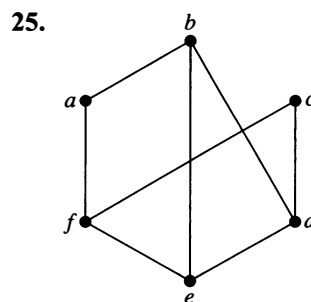
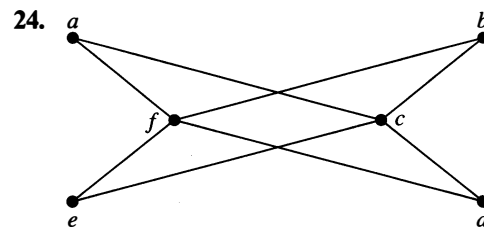
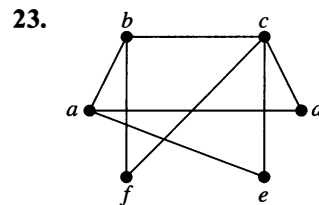
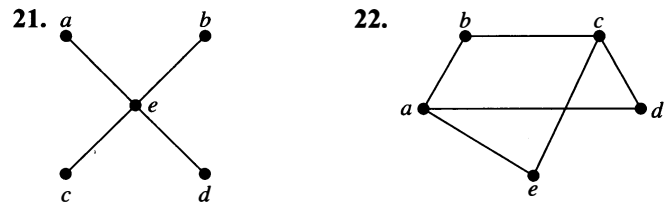


- Find the sum of the degrees of the vertices of each graph in Exercises 1–3 and verify that it equals twice the number of edges in the graph.
- Can a simple graph exist with 15 vertices each of degree five?
- Show that the sum, over the set of people at a party, of the number of people a person has shaken hands with, is even. Assume that no one shakes his or her own hand.

In Exercises 7–9 determine the number of vertices and edges and find the in-degree and out-degree of each vertex for the given directed multigraph.



red or blue to each vertex so that no two adjacent vertices are assigned the same color.



- 10. For each of the graphs in Exercises 7–9 determine the sum of the in-degrees of the vertices and the sum of the out-degrees of the vertices directly. Show that they are both equal to the number of edges in the graph.
- 11. Construct the underlying undirected graph for the graph with directed edges in Figure 2.
- 12. What does the degree of a vertex represent in the acquaintanceship graph, where vertices represent all the people in the world? What do isolated and pendant vertices in this graph represent? In one study it was estimated that the average degree of a vertex in this graph is 1000. What does this mean in terms of the model?
- 13. What does the degree of a vertex represent in a collaboration graph? What do isolated and pendant vertices represent?
- 14. What does the degree of a vertex in the Hollywood graph represent? What do the isolated and pendant vertices represent?
- 15. What do the in-degree and the out-degree of a vertex in a telephone call graph, as described in Example 7 of Section 9.1, represent? What does the degree of a vertex in the undirected version of this graph represent?
- 16. What do the in-degree and the out-degree of a vertex in the Web graph, as described in Example 8 of Section 9.1, represent?
- 17. What do the in-degree and the out-degree of a vertex in a directed graph modeling a round-robin tournament represent?
- 18. Show that in a simple graph with at least two vertices there must be two vertices that have the same degree.
- 19. Use Exercise 18 to show that in a group, there must be two people who know the same number of other people in the group.
- 20. Draw these graphs.
 - a) K_7 b) $K_{1,8}$ c) $K_{4,4}$
 - d) C_7 e) W_7 f) Q_4
- 21. a b
 c d
 e
- 22. b c
 a d
 e
- 23. b c
 a d
 f e
- 24. a b
 f c
 e d
- 25. b
 a c
 f d
 e
- 26. For which values of n are these graphs bipartite?
 a) K_n b) C_n c) W_n d) Q_n
- 27. Suppose that a new company has five employees: Zamora, Agraharam, Smith, Chou, and Macintyre. Each employee will assume one of six responsibilities: planning, publicity, sales, marketing, development, and industry relations. Each employee is capable of doing one or more of these jobs: Zamora could do planning, sales, marketing, or industry relations; Agraharam could do planning or development; Smith could do publicity, sales, or industry relations; Chou could do planning, sales, or industry relations; and Macintyre could do planning, publicity, sales, or industry relations.
 - a) Model the capabilities of these employees using a bipartite graph.
 - b) Find an assignment of responsibilities such that each employee is assigned a responsibility.
- 28. Suppose that there are five young women and six young men on an island. Each woman is willing to marry some

In Exercises 21–25 determine whether the graph is bipartite. You may find it useful to apply Theorem 4 and answer the question by determining whether it is possible to assign either

of the men on the island and each man is willing to marry any woman who is willing to marry him. Suppose that Anna is willing to marry Jason, Larry, and Matt; Barbara is willing to marry Kevin and Larry; Carol is willing to marry Jason, Nick, and Oscar; Diane is willing to marry Jason, Larry, Nick, and Oscar; and Elizabeth is willing to marry Jason and Matt.

- a) Model the possible marriages on the island using a bipartite graph.
 - b) Find a matching of the young women and the young men on the island such that each young woman is matched with a young man whom she is willing to marry.
29. How many vertices and how many edges do these graphs have?
- a) K_n b) C_n c) W_n
 - d) $K_{m,n}$ e) Q_n

The **degree sequence** of a graph is the sequence of the degrees of the vertices of the graph in nonincreasing order. For example, the degree sequence of the graph G in Example 1 in this section is 4, 4, 4, 3, 2, 1, 0.

30. Find the degree sequences for each of the graphs in Exercises 21–25.
31. Find the degree sequence of each of the following graphs.
- a) K_4 b) C_4 c) W_4
 - d) $K_{2,3}$ e) Q_3

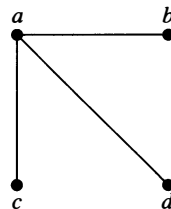
32. What is the degree sequence of the bipartite graph $K_{m,n}$ where m and n are positive integers? Explain your answer.
33. What is the degree sequence of K_n , where n is a positive integer? Explain your answer.
34. How many edges does a graph have if its degree sequence is 4, 3, 3, 2, 2? Draw such a graph.
35. How many edges does a graph have if its degree sequence is 5, 2, 2, 2, 2, 1? Draw such a graph.

A sequence d_1, d_2, \dots, d_n is called **graphic** if it is the degree sequence of a simple graph.

36. Determine whether each of these sequences is graphic. For those that are, draw a graph having the given degree sequence.
- a) 5, 4, 3, 2, 1, 0 b) 6, 5, 4, 3, 2, 1
 - c) 2, 2, 2, 2, 2, 2 d) 3, 3, 3, 2, 2, 2
 - e) 3, 3, 2, 2, 2, 2 f) 1, 1, 1, 1, 1, 1
 - g) 5, 3, 3, 3, 3, 3 h) 5, 5, 4, 3, 2, 1
37. Determine whether each of these sequences is graphic. For those that are, draw a graph having the given degree sequence.
- a) 3, 3, 3, 3, 2 b) 5, 4, 3, 2, 1
 - c) 4, 4, 3, 2, 1 d) 4, 4, 3, 3, 3
 - e) 3, 2, 2, 1, 0 f) 1, 1, 1, 1, 1

38. Suppose that d_1, d_2, \dots, d_n is a graphic sequence. Show that there is a simple graph with vertices v_1, v_2, \dots, v_n such that $\deg(v_i) = d_i$ for $i = 1, 2, \dots, n$ and v_1 is adjacent to v_2, \dots, v_{d_1} .

- *39. Show that a sequence d_1, d_2, \dots, d_n of nonnegative integers in nonincreasing order is a graphic sequence if and only if the sequence obtained by reordering the terms of the sequence $d_2 - 1, \dots, d_{d_1+1} - 1, d_{d_1+2}, \dots, d_n$ so that the terms are in nonincreasing order is a graphic sequence.
- *40. Use Exercise 39 to construct a recursive algorithm for determining whether a nonincreasing sequence of positive integers is graphic.
41. Show that every nonincreasing sequence of nonnegative integers with an even sum of its terms is the degree sequence of a pseudograph, that is, an undirected graph where loops are allowed. [Hint: Construct such a graph by first adding as many loops as possible at each vertex. Then add additional edges connecting vertices of odd degree. Explain why this construction works.]
42. How many subgraphs with at least one vertex does K_2 have?
43. How many subgraphs with at least one vertex does K_3 have?
44. How many subgraphs with at least one vertex does W_3 have?
45. Draw all subgraphs of this graph.



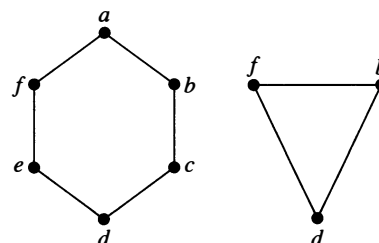
46. Let G be a graph with v vertices and e edges. Let M be the maximum degree of the vertices of G , and let m be the minimum degree of the vertices of G . Show that
- a) $2e/v \geq m$.
 - b) $2e/v \leq M$.

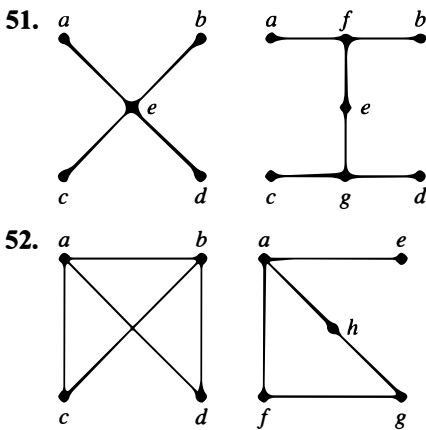
A simple graph is called **regular** if every vertex of this graph has the same degree. A regular graph is called **n -regular** if every vertex in this graph has degree n .

47. For which values of n are these graphs regular?
- a) K_n b) C_n c) W_n d) Q_n
48. For which values of m and n is $K_{m,n}$ regular?
49. How many vertices does a regular graph of degree four with 10 edges have?

In Exercises 50–52 find the union of the given pair of simple graphs. (Assume edges with the same endpoints are the same.)

- 50.





53. The **complementary graph** \overline{G} of a simple graph G has the same vertices as G . Two vertices are adjacent in \overline{G} if and only if they are not adjacent in G . Describe each of these graphs.
 a) $\overline{K_n}$ b) $\overline{K_{m,n}}$ c) $\overline{C_n}$ d) $\overline{Q_n}$
54. If G is a simple graph with 15 edges and \overline{G} has 13 edges, how many vertices does G have?
55. If the simple graph G has v vertices and e edges, how many edges does \overline{G} have?
56. If the degree sequence of the simple graph G is 4, 3, 3, 2, 2, what is the degree sequence of \overline{G} ?
57. If the degree sequence of the simple graph G is d_1, d_2, \dots, d_n , what is the degree sequence of \overline{G} ?
58. Show that if G is a bipartite simple graph with v vertices and e edges, then $e \leq v^2/4$.
59. Show that if G is a simple graph with n vertices, then the union of G and \overline{G} is K_n .

- *60. Describe an algorithm to decide whether a graph is bipartite based on the fact that a graph is bipartite if and only if it is possible to color its vertices two different colors so that no two vertices of the same color are adjacent.
- The **converse** of a directed graph $G = (V, E)$, denoted by G^{conv} , is the directed graph (V, F) , where the set F of edges of G^{conv} is obtained by reversing the direction of each edge in E .
61. Draw the converse of each of the graphs in Exercises 7–9 in Section 9.1.
62. Show that $(G^{conv})^{conv} = G$ whenever G is a directed graph.
63. Show that the graph G is its own converse if and only if the relation associated with G (see Section 8.3) is symmetric.
64. Show that if a bipartite graph $G = (V, E)$ is n -regular for some positive integer n (see the preamble to Exercise 47) and (V_1, V_2) is a bipartition of V , then $|V_1| = |V_2|$. That is, show that the two sets in a bipartition of the vertex set of an n -regular graph must contain the same number of vertices.
65. Draw the mesh network for interconnecting nine parallel processors.
66. In a variant of a mesh network for interconnecting $n = m^2$ processors, processor $P(i, j)$ is connected to the four processors $P((i \pm 1) \bmod m, j)$ and $P(i, (j \pm 1) \bmod m)$, so that connections wrap around the edges of the mesh. Draw this variant of the mesh network for 16 processors.
67. Show that every pair of processors in a mesh network of $n = m^2$ processors can communicate using $O(\sqrt{n}) = O(m)$ hops between directly connected processors.

9.3 Representing Graphs and Graph Isomorphism

Introduction

There are many useful ways to represent graphs. As we will see throughout this chapter, in working with a graph it is helpful to be able to choose its most convenient representation. In this section we will show how to represent graphs in several different ways.

Sometimes, two graphs have exactly the same form, in the sense that there is a one-to-one correspondence between their vertex sets that preserves edges. In such a case, we say that the two graphs are **isomorphic**. Determining whether two graphs are isomorphic is an important problem of graph theory that we will study in this section.

Representing Graphs

One way to represent a graph without multiple edges is to list all the edges of this graph. Another way to represent a graph with no multiple edges is to use **adjacency lists**, which specify the vertices that are adjacent to each vertex of the graph.

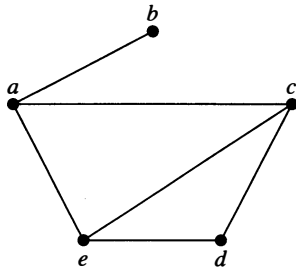


FIGURE 1 A Simple Graph.

Vertex	Adjacent Vertices
a	b, c, e
b	a
c	a, d, e
d	c, e
e	a, c, d

EXAMPLE 1 Use adjacency lists to describe the simple graph given in Figure 1.

Solution: Table 1 lists those vertices adjacent to each of the vertices of the graph. ◀

EXAMPLE 2 Represent the directed graph shown in Figure 2 by listing all the vertices that are the terminal vertices of edges starting at each vertex of the graph.

Solution: Table 2 represents the directed graph shown in Figure 2. ◀

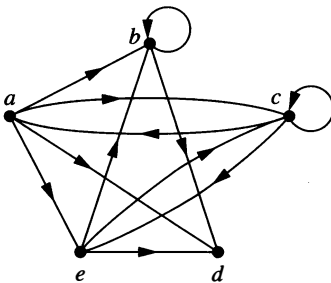


FIGURE 2 A Directed Graph.

Initial Vertex	Terminal Vertices
a	b, c, d, e
b	b, d
c	a, c, e
d	
e	b, c, d

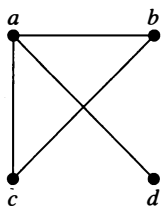


FIGURE 3 Simple Graph.

Adjacency Matrices

Carrying out graph algorithms using the representation of graphs by lists of edges, or by adjacency lists, can be cumbersome if there are many edges in the graph. To simplify computation, graphs can be represented using matrices. Two types of matrices commonly used to represent graphs will be presented here. One is based on the adjacency of vertices, and the other is based on incidence of vertices and edges.



Suppose that $G = (V, E)$ is a simple graph where $|V| = n$. Suppose that the vertices of G are listed arbitrarily as v_1, v_2, \dots, v_n . The **adjacency matrix** A (or A_G) of G , with respect to this listing of the vertices, is the $n \times n$ zero-one matrix with 1 as its (i, j) th entry when v_i and v_j are adjacent, and 0 as its (i, j) th entry when they are not adjacent. In other words, if its adjacency matrix is $A = [a_{ij}]$, then

$$a_{ij} = \begin{cases} 1 & \text{if } \{v_i, v_j\} \text{ is an edge of } G, \\ 0 & \text{otherwise.} \end{cases}$$

EXAMPLE 3 Use an adjacency matrix to represent the graph shown in Figure 3.

Solution: We order the vertices as a, b, c, d . The matrix representing this graph is

$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}.$$

EXAMPLE 4 Draw a graph with the adjacency matrix

$$\begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}$$

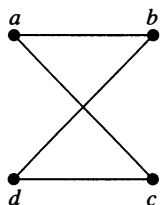


FIGURE 4
A Graph with the
Given Adjacency
Matrix.

with respect to the ordering of vertices a, b, c, d .

Solution: A graph with this adjacency matrix is shown in Figure 4.

Note that an adjacency matrix of a graph is based on the ordering chosen for the vertices. Hence, there are as many as $n!$ different adjacency matrices for a graph with n vertices, because there are $n!$ different orderings of n vertices.

The adjacency matrix of a simple graph is symmetric, that is, $a_{ij} = a_{ji}$, because both of these entries are 1 when v_i and v_j are adjacent, and both are 0 otherwise. Furthermore, because a simple graph has no loops, each entry a_{ii} , $i = 1, 2, 3, \dots, n$, is 0.

Adjacency matrices can also be used to represent undirected graphs with loops and with multiple edges. A loop at the vertex a_i is represented by a 1 at the (i, i) th position of the adjacency matrix. When multiple edges are present, the adjacency matrix is no longer a zero–one matrix, because the (i, j) th entry of this matrix equals the number of edges that are associated to $\{a_i, a_j\}$. All undirected graphs, including multigraphs and pseudographs, have symmetric adjacency matrices.

EXAMPLE 5 Use an adjacency matrix to represent the pseudograph shown in Figure 5.

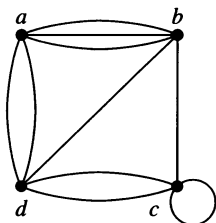


FIGURE 5
A Pseudograph.

Solution: The adjacency matrix using the ordering of vertices a, b, c, d is

$$\begin{bmatrix} 0 & 3 & 0 & 2 \\ 3 & 0 & 1 & 1 \\ 0 & 1 & 1 & 2 \\ 2 & 1 & 2 & 0 \end{bmatrix}.$$

We used zero–one matrices in Chapter 8 to represent directed graphs. The matrix for a directed graph $G = (V, E)$ has a 1 in its (i, j) th position if there is an edge from v_i to v_j , where v_1, v_2, \dots, v_n is an arbitrary listing of the vertices of the directed graph. In other words, if $\mathbf{A} = [a_{ij}]$ is the adjacency matrix for the directed graph with respect to this listing of the vertices, then

$$a_{ij} = \begin{cases} 1 & \text{if } (v_i, v_j) \text{ is an edge of } G, \\ 0 & \text{otherwise.} \end{cases}$$

The adjacency matrix for a directed graph does not have to be symmetric, because there may not be an edge from a_j to a_i when there is an edge from a_i to a_j .

Adjacency matrices can also be used to represent directed multigraphs. Again, such matrices are not zero–one matrices when there are multiple edges in the same direction connecting two vertices. In the adjacency matrix for a directed multigraph, a_{ij} equals the number of edges that are associated to (v_i, v_j) .

TRADE-OFFS BETWEEN ADJACENCY LISTS AND ADJACENCY MATRICES When a simple graph contains relatively few edges, that is, when it is **sparse**, it is usually preferable to use adjacency lists rather than an adjacency matrix to represent the graph. For example, if each vertex has degree not exceeding c , where c is a constant much smaller than n , then each adjacency list contains c or fewer vertices. Hence, there are no more than cn items in all these adjacency lists. On the other hand, the adjacency matrix for the graph has n^2 entries. Note, however, that the adjacency matrix of a sparse graph is a **sparse matrix**, that is, a matrix with few nonzero entries, and there are special techniques for representing, and computing with, sparse matrices.

Now suppose that a simple graph is **dense**, that is, suppose that it contains many edges, such as a graph that contains more than half of all possible edges. In this case, using an adjacency matrix to represent the graph is usually preferable over using adjacency lists. To see why, we compare the complexity of determining whether the possible edge $\{v_i, v_j\}$ is present. Using an adjacency matrix, we can determine whether this edge is present by examining the (i, j) th entry in the matrix. This entry is 1 if the graph contains this edge and is 0 otherwise. Consequently, we need make only one comparison, namely, comparing this entry with 0, to determine whether this edge is present. On the other hand, when we use adjacency lists to represent the graph, we need to search the list of vertices adjacent to either v_i or v_j to determine whether this edge is present. This can require $\Theta(|V|)$ comparisons when many edges are present.

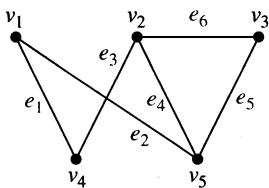
Incidence Matrices

Another common way to represent graphs is to use **incidence matrices**. Let $G = (V, E)$ be an undirected graph. Suppose that v_1, v_2, \dots, v_n are the vertices and e_1, e_2, \dots, e_m are the edges of G . Then the incidence matrix with respect to this ordering of V and E is the $n \times m$ matrix $M = [m_{ij}]$, where

$$m_{ij} = \begin{cases} 1 & \text{when edge } e_j \text{ is incident with } v_i, \\ 0 & \text{otherwise.} \end{cases}$$

EXAMPLE 6 Represent the graph shown in Figure 6 with an incidence matrix.

Solution: The incidence matrix is



$$\begin{array}{c} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{array} \begin{bmatrix} e_1 & e_2 & e_3 & e_4 & e_5 & e_6 \\ \left[\begin{array}{cccccc} 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 \end{array} \right] \end{bmatrix}$$

FIGURE 6 An Undirected Graph.

Incidence matrices can also be used to represent multiple edges and loops. Multiple edges are represented in the incidence matrix using columns with identical entries, because these edges are incident with the same pair of vertices. Loops are represented using a column with exactly one entry equal to 1, corresponding to the vertex that is incident with this loop.

EXAMPLE 7 Represent the pseudograph shown in Figure 7 using an incidence matrix.

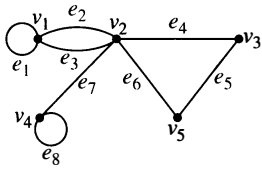


FIGURE 7
A Pseudograph.

Solution: The incidence matrix for this graph is

$$\begin{matrix} & e_1 & e_2 & e_3 & e_4 & e_5 & e_6 & e_7 & e_8 \\ \begin{matrix} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{matrix} & \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \end{bmatrix} \end{matrix}$$

Isomorphism of Graphs

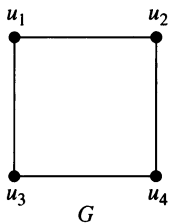
We often need to know whether it is possible to draw two graphs in the same way. For instance, in chemistry, graphs are used to model compounds. Different compounds can have the same molecular formula but can differ in structure. Such compounds will be represented by graphs that cannot be drawn in the same way. The graphs representing previously known compounds can be used to determine whether a supposedly new compound has been studied before.

There is a useful terminology for graphs with the same structure.

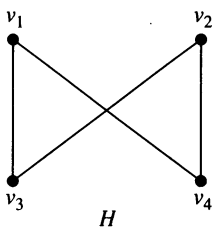
DEFINITION 1 The simple graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are *isomorphic* if there is a one-to-one and onto function f from V_1 to V_2 with the property that a and b are adjacent in G_1 if and only if $f(a)$ and $f(b)$ are adjacent in G_2 , for all a and b in V_1 . Such a function f is called an *isomorphism*.*

In other words, when two simple graphs are isomorphic, there is a one-to-one correspondence between vertices of the two graphs that preserves the adjacency relationship. Isomorphism of simple graphs is an equivalence relation. (We leave the verification of this as Exercise 45 at the end of this section.)

EXAMPLE 8 Show that the graphs $G = (V, E)$ and $H = (W, F)$, displayed in Figure 8, are isomorphic.



Solution: The function f with $f(u_1) = v_1$, $f(u_2) = v_4$, $f(u_3) = v_3$, and $f(u_4) = v_2$ is a one-to-one correspondence between V and W . To see that this correspondence preserves adjacency, note that adjacent vertices in G are u_1 and u_2 , u_1 and u_3 , u_2 and u_4 , and u_3 and u_4 , and each of the pairs $f(u_1) = v_1$ and $f(u_2) = v_4$, $f(u_1) = v_1$ and $f(u_3) = v_3$, $f(u_2) = v_4$ and $f(u_4) = v_2$, and $f(u_3) = v_3$ and $f(u_4) = v_2$ are adjacent in H .



It is often difficult to determine whether two simple graphs are isomorphic. There are $n!$ possible one-to-one correspondences between the vertex sets of two simple graphs with n vertices. Testing each such correspondence to see whether it preserves adjacency and nonadjacency is impractical if n is at all large.

Sometimes it is not hard to show that two graphs are not isomorphic. In particular, we can show that two graphs are not isomorphic if we can find a property only one of the two graphs has, but that is preserved by isomorphism. A property preserved by isomorphism of graphs is called a **graph invariant**. For instance, isomorphic simple graphs must have the same number of vertices, because there is a one-to-one correspondence between the sets of vertices of the graphs.

FIGURE 8 The Graphs G and H .

*The word *isomorphism* comes from the Greek roots *isos* for “equal” and *morphe* for “form.”

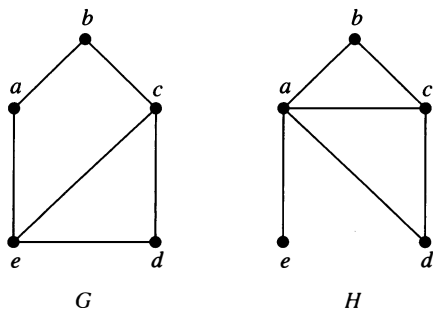


FIGURE 9 The Graphs G and H .

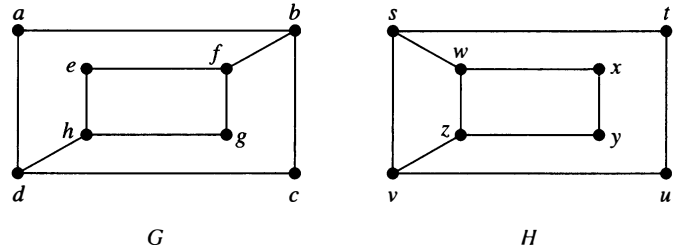


FIGURE 10 The Graphs G and H .



Isomorphic simple graphs also must have the same number of edges, because the one-to-one correspondence between vertices establishes a one-to-one correspondence between edges. In addition, the degrees of the vertices in isomorphic simple graphs must be the same. That is, a vertex v of degree d in G must correspond to a vertex $f(v)$ of degree d in H , because a vertex w in G is adjacent to v if and only if $f(v)$ and $f(w)$ are adjacent in H .

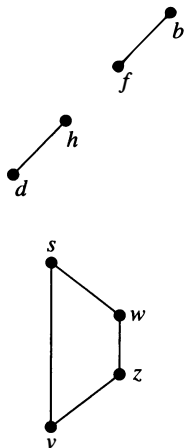
EXAMPLE 9 Show that graphs displayed in Figure 9 are not isomorphic.



Solution: Both G and H have five vertices and six edges. However, H has a vertex of degree one, namely, e , whereas G has no vertices of degree one. It follows that G and H are not isomorphic. ◀

The number of vertices, the number of edges, and the number of vertices of each degree are all invariants under isomorphism. If any of these quantities differ in two simple graphs, these graphs cannot be isomorphic. However, when these invariants are the same, it does not necessarily mean that the two graphs are isomorphic. There are no useful sets of invariants currently known that can be used to determine whether simple graphs are isomorphic.

EXAMPLE 10 Determine whether the graphs shown in Figure 10 are isomorphic.



Solution: The graphs G and H both have eight vertices and 10 edges. They also both have four vertices of degree two and four of degree three. Because these invariants all agree, it is still conceivable that these graphs are isomorphic.

However, G and H are not isomorphic. To see this, note that because $\deg(a) = 2$ in G , a must correspond to either $t, u, x,$ or y in H , because these are the vertices of degree two in H . However, each of these four vertices in H is adjacent to another vertex of degree two in H , which is not true for a in G .

Another way to see that G and H are not isomorphic is to note that the subgraphs of G and H made up of vertices of degree three and the edges connecting them must be isomorphic if these two graphs are isomorphic (the reader should verify this). However, these subgraphs, shown in Figure 11, are not isomorphic. ◀

FIGURE 11 The Subgraphs of G and H Made Up of Vertices of Degree Three and the Edges Connecting Them.

To show that a function f from the vertex set of a graph G to the vertex set of a graph H is an isomorphism, we need to show that f preserves the presence and absence of edges. One helpful way to do this is to use adjacency matrices. In particular, to show that f is an isomorphism, we can show that the adjacency matrix of G is the same as the adjacency matrix of H , when rows and columns are labeled to correspond to the images under f of the vertices in G that are the labels of these rows and columns in the adjacency matrix of G . We illustrate how this is done in Example 11.

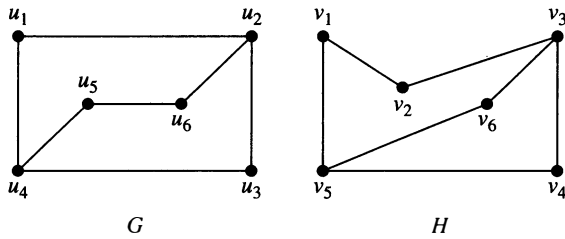


FIGURE 12 Graphs G and H .

EXAMPLE 11 Determine whether the graphs G and H displayed in Figure 12 are isomorphic.

Solution: Both G and H have six vertices and seven edges. Both have four vertices of degree two and two vertices of degree three. It is also easy to see that the subgraphs of G and H consisting of all vertices of degree two and the edges connecting them are isomorphic (as the reader should verify). Because G and H agree with respect to these invariants, it is reasonable to try to find an isomorphism f .

We now will define a function f and then determine whether it is an isomorphism. Because $\deg(u_1) = 2$ and because u_1 is not adjacent to any other vertex of degree two, the image of u_1 must be either v_4 or v_6 , the only vertices of degree two in H not adjacent to a vertex of degree two. We arbitrarily set $f(u_1) = v_6$. [If we found that this choice did not lead to isomorphism, we would then try $f(u_1) = v_4$.] Because u_2 is adjacent to u_1 , the possible images of u_2 are v_3 and v_5 . We arbitrarily set $f(u_2) = v_3$. Continuing in this way, using adjacency of vertices and degrees as a guide, we set $f(u_3) = v_4$, $f(u_4) = v_5$, $f(u_5) = v_1$, and $f(u_6) = v_2$. We now have a one-to-one correspondence between the vertex set of G and the vertex set of H , namely, $f(u_1) = v_6$, $f(u_2) = v_3$, $f(u_3) = v_4$, $f(u_4) = v_5$, $f(u_5) = v_1$, $f(u_6) = v_2$. To see whether f preserves edges, we examine the adjacency matrix of G ,

$$\mathbf{A}_G = \begin{matrix} & \begin{matrix} u_1 & u_2 & u_3 & u_4 & u_5 & u_6 \end{matrix} \\ \begin{matrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \\ u_6 \end{matrix} & \begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 \end{bmatrix} \end{matrix},$$

and the adjacency matrix of H with the rows and columns labeled by the images of the corresponding vertices in G ,

$$\mathbf{A}_H = \begin{matrix} & \begin{matrix} v_6 & v_3 & v_4 & v_5 & v_1 & v_2 \end{matrix} \\ \begin{matrix} v_6 \\ v_3 \\ v_4 \\ v_5 \\ v_1 \\ v_2 \end{matrix} & \begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 \end{bmatrix} \end{matrix}.$$

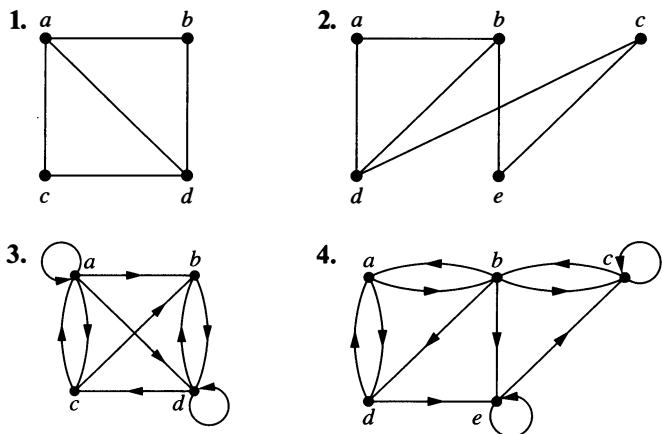
Because $A_G = A_H$, it follows that f preserves edges. We conclude that f is an isomorphism, so G and H are isomorphic. Note that if f turned out not to be an isomorphism, we would *not* have established that G and H are not isomorphic, because another correspondence of the vertices in G and H may be an isomorphism. ◀



The best algorithms known for determining whether two graphs are isomorphic have exponential worst-case time complexity (in the number of vertices of the graphs). However, linear average-case time complexity algorithms are known that solve this problem, and there is some hope that an algorithm with polynomial worst-case time complexity for determining whether two graphs are isomorphic can be found. The best practical algorithm, called NAUTY, can be used to determine whether two graphs with as many as 100 vertices are isomorphic in less than 1 second on a modern PC. The software for NAUTY can be downloaded over the Internet and experimented with.

Exercises

In Exercises 1–4 use an adjacency list to represent the given graph.



5. Represent the graph in Exercise 1 with an adjacency matrix.
6. Represent the graph in Exercise 2 with an adjacency matrix.
7. Represent the graph in Exercise 3 with an adjacency matrix.
8. Represent the graph in Exercise 4 with an adjacency matrix.
9. Represent each of these graphs with an adjacency matrix.
 - a) K_4
 - b) $K_{1,4}$
 - c) $K_{2,3}$
 - d) C_4
 - e) W_4
 - f) Q_3

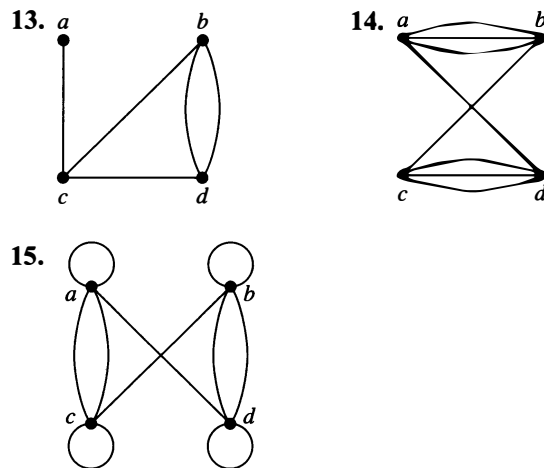
In Exercises 10–12 draw a graph with the given adjacency matrix.

10.
$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

11.
$$\begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

12.
$$\begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

In Exercises 13–15 represent the given graph using an adjacency matrix.



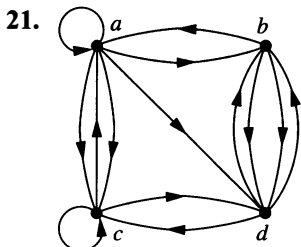
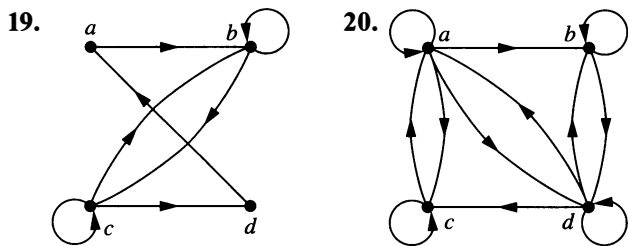
In Exercises 16–18 draw an undirected graph represented by the given adjacency matrix.

16.
$$\begin{bmatrix} 1 & 3 & 2 \\ 3 & 0 & 4 \\ 2 & 4 & 0 \end{bmatrix}$$

17.
$$\begin{bmatrix} 1 & 2 & 0 & 1 \\ 2 & 0 & 3 & 0 \\ 0 & 3 & 1 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix}$$

18.
$$\begin{bmatrix} 0 & 1 & 3 & 0 & 4 \\ 1 & 2 & 1 & 3 & 0 \\ 3 & 1 & 1 & 0 & 1 \\ 0 & 3 & 0 & 0 & 2 \\ 4 & 0 & 1 & 2 & 3 \end{bmatrix}$$

In Exercises 19–21 find the adjacency matrix of the given directed multigraph.



In Exercises 22–24 draw the graph represented by the given adjacency matrix.

22. $\begin{bmatrix} 1 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix}$ 23. $\begin{bmatrix} 1 & 2 & 1 \\ 2 & 0 & 0 \\ 0 & 2 & 2 \end{bmatrix}$ 24. $\begin{bmatrix} 0 & 2 & 3 & 0 \\ 1 & 2 & 2 & 1 \\ 2 & 1 & 1 & 0 \\ 1 & 0 & 0 & 2 \end{bmatrix}$

25. Is every zero–one square matrix that is symmetric and has zeros on the diagonal the adjacency matrix of a simple graph?

26. Use an incidence matrix to represent the graphs in Exercises 1 and 2.

27. Use an incidence matrix to represent the graphs in Exercises 13–15.

*28. What is the sum of the entries in a row of the adjacency matrix for an undirected graph? For a directed graph?

*29. What is the sum of the entries in a column of the adjacency matrix for an undirected graph? For a directed graph?

30. What is the sum of the entries in a row of the incidence matrix for an undirected graph?

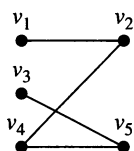
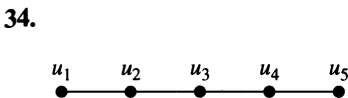
31. What is the sum of the entries in a column of the incidence matrix for an undirected graph?

*32. Find an adjacency matrix for each of these graphs.

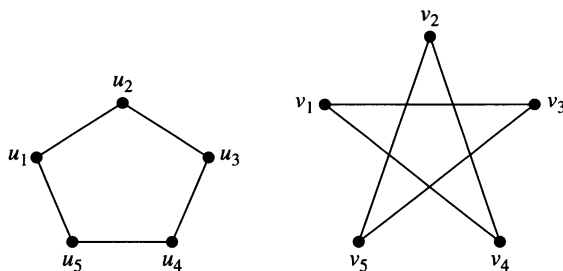
- a) K_n b) C_n c) W_n d) $K_{m,n}$ e) Q_n

*33. Find incidence matrices for the graphs in parts (a)–(d) of Exercise 32.

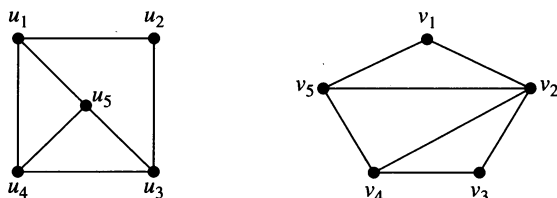
In Exercises 34–44 determine whether the given pair of graphs is isomorphic. Exhibit an isomorphism or provide a rigorous argument that none exists.



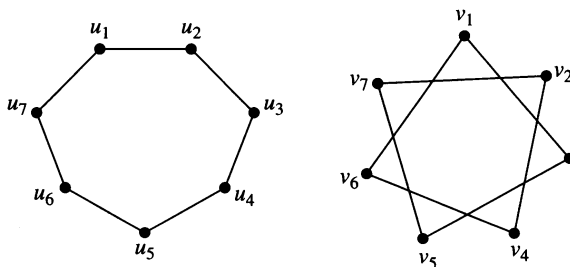
35.



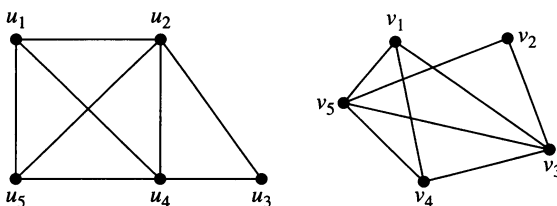
36.



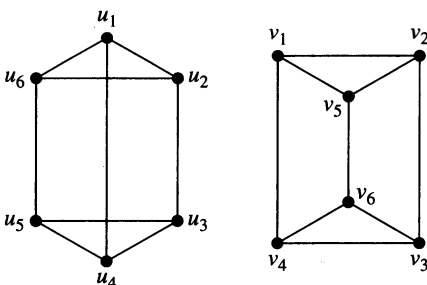
37.



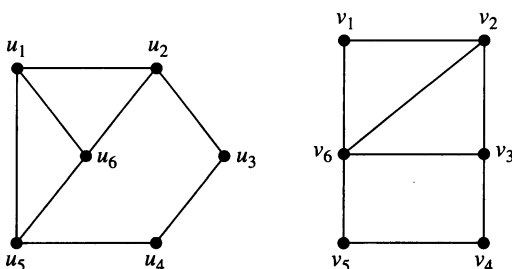
38.

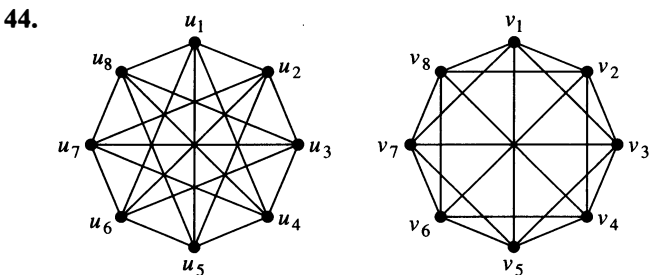
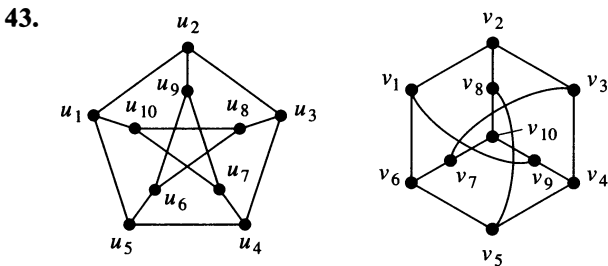
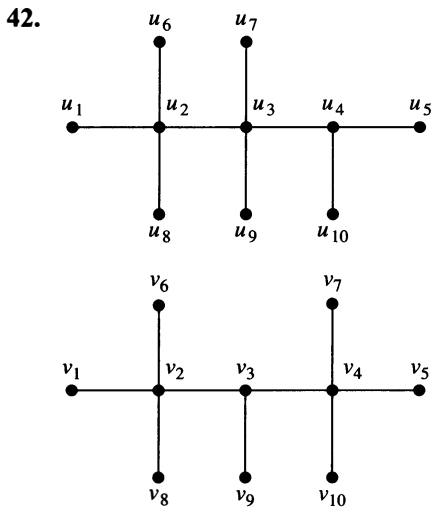
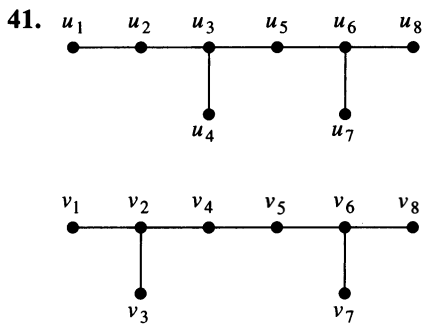


39.



40.





45. Show that isomorphism of simple graphs is an equivalence relation.
 46. Suppose that G and H are isomorphic simple graphs. Show that their complementary graphs \overline{G} and \overline{H} are also isomorphic.

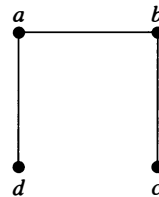
47. Describe the row and column of an adjacency matrix of a graph corresponding to an isolated vertex.
 48. Describe the row of an incidence matrix of a graph corresponding to an isolated vertex.
 49. Show that the vertices of a bipartite graph with two or more vertices can be ordered so that its adjacency matrix has the form

$$\begin{bmatrix} \mathbf{0} & \mathbf{A} \\ \mathbf{B} & \mathbf{0} \end{bmatrix},$$

where the four entries shown are rectangular blocks.

A simple graph G is called **self-complementary** if G and \overline{G} are isomorphic.

50. Show that this graph is self-complementary.



51. Find a self-complementary simple graph with five vertices.
 *52. Show that if G is a self-complementary simple graph with v vertices, then $v \equiv 0$ or $1 \pmod{4}$.
 53. For which integers n is C_n self-complementary?
 54. How many nonisomorphic simple graphs are there with n vertices, when n is
 a) 2? b) 3? c) 4?
 55. How many nonisomorphic simple graphs are there with five vertices and three edges?
 56. How many nonisomorphic simple graphs are there with six vertices and four edges?
 57. Are the simple graphs with the following adjacency matrices isomorphic?

a) $\begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}$

b) $\begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$

c) $\begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix}$

58. Determine whether the graphs without loops with these incidence matrices are isomorphic.

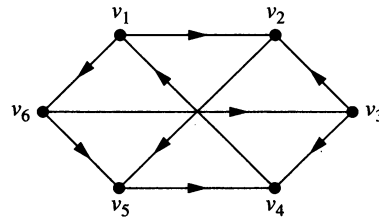
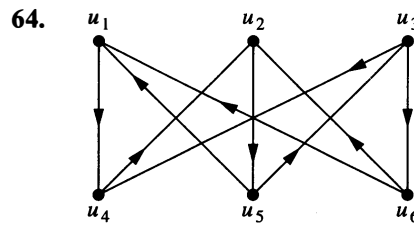
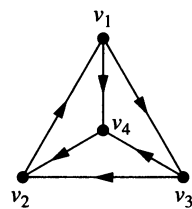
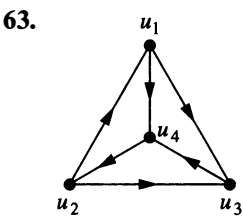
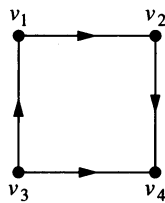
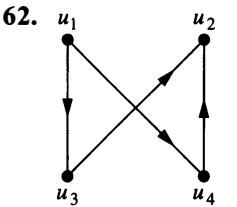
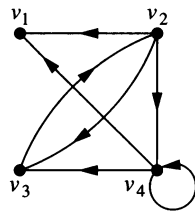
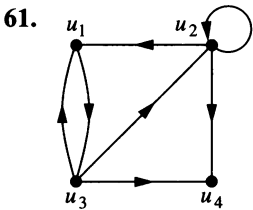
a) $\begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 0 \end{bmatrix}, \begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix}$

b) $\begin{bmatrix} 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \end{bmatrix}$

59. Extend the definition of isomorphism of simple graphs to undirected graphs containing loops and multiple edges.

60. Define isomorphism of directed graphs.

In Exercises 61–64 determine whether the given pair of directed graphs are isomorphic. (See Exercise 60.)



65. Show that if G and H are isomorphic directed graphs, then the converses of G and H (defined in the preamble of Exercise 61 of Section 9.2) are also isomorphic.

66. Show that the property that a graph is bipartite is an isomorphic invariant.

67. Find a pair of nonisomorphic graphs with the same degree sequence such that one graph is bipartite, but the other graph is not bipartite.

*68. How many nonisomorphic directed simple graphs are there with n vertices, when n is

- a) 2? b) 3? c) 4?

*69. What is the product of the incidence matrix and its transpose for an undirected graph?

*70. How much storage is needed to represent a simple graph with v vertices and e edges using

- a) adjacency lists?
- b) an adjacency matrix?
- c) an incidence matrix?

A **devil's pair** for a purported isomorphism test is a pair of nonisomorphic graphs that the test fails to show are not isomorphic.

71. Find a devil's pair for the test that checks the degree sequence (defined in the preamble to Exercise 30 in Section 9.2) in two graphs to make sure they agree.

9.4 Connectivity

Introduction

Many problems can be modeled with paths formed by traveling along the edges of graphs. For instance, the problem of determining whether a message can be sent between two computers using intermediate links can be studied with a graph model. Problems of efficiently planning routes for mail delivery, garbage pickup, diagnostics in computer networks, and so on can be solved using models that involve paths in graphs.

Paths

Informally, a **path** is a sequence of edges that begins at a vertex of a graph and travels from vertex to vertex along edges of the graph.

A formal definition of paths and related terminology is given in Definition 1.

DEFINITION 1 Let n be a nonnegative integer and G an undirected graph. A *path* of length n from u to v in G is a sequence of n edges e_1, \dots, e_n of G such that e_1 is associated with $\{x_0, x_1\}$, e_2 is associated with $\{x_1, x_2\}$, and so on, with e_n associated with $\{x_{n-1}, x_n\}$, where $x_0 = u$ and $x_n = v$. When the graph is simple, we denote this path by its vertex sequence x_0, x_1, \dots, x_n (because listing these vertices uniquely determines the path). The path is a *circuit* if it begins and ends at the same vertex, that is, if $u = v$, and has length greater than zero. The path or circuit is said to *pass through* the vertices x_1, x_2, \dots, x_{n-1} or *traverse* the edges e_1, e_2, \dots, e_n . A path or circuit is *simple* if it does not contain the same edge more than once.

When it is not necessary to distinguish between multiple edges, we will denote a path e_1, e_2, \dots, e_n , where e_i is associated with $\{x_{i-1}, x_i\}$ for $i = 1, 2, \dots, n$ by its vertex sequence x_0, x_1, \dots, x_n . This notation identifies a path only up to the vertices it passes through. There may be more than one path that passes through this sequence of vertices. Note that a path of length zero consists of a single vertex.

Remark: There is considerable variation of terminology concerning the concepts defined in Definition 1. For instance, in some books, the term **walk** is used instead of *path*, where a walk is defined to be an alternating sequence of vertices and edges of a graph, $v_0, e_1, v_1, e_2, \dots, v_{n-1}, e_n, v_n$, where v_{i-1} and v_i are the endpoints of e_i for $i = 1, 2, \dots, n$. When this terminology is used, **closed walk** is used instead of *circuit* to indicate a walk that begins and ends at the same vertex, and **trail** is used to denote a walk that has no repeated edge (replacing the term *simple path*). When this terminology is used, the terminology **path** is often used for a trail with no repeated vertices, conflicting with the terminology in Definition 1. Because of this variation in terminology, you will need to make sure which set of definitions are used in a particular book or article when you read about traversing edges of a graph. The text [GrYe06] is a good reference for the alternative terminology described in this remark.

EXAMPLE 1 In the simple graph shown in Figure 1, a, d, c, f, e is a simple path of length 4, because $\{a, d\}$, $\{d, c\}$, $\{c, f\}$, and $\{f, e\}$ are all edges. However, d, e, c, a is not a path, because $\{e, c\}$ is not an edge. Note that b, c, f, e, b is a circuit of length 4 because $\{b, c\}$, $\{c, f\}$, $\{f, e\}$, and $\{e, b\}$ are edges, and this path begins and ends at b . The path a, b, e, d, a, b , which is of length 5, is not simple because it contains the edge $\{a, b\}$ twice. ◀

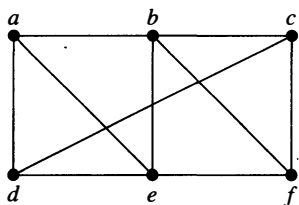


FIGURE 1 A Simple Graph.

Paths and circuits in directed graphs were introduced in Chapter 8. We now provide more general definitions.

DEFINITION 2 Let n be a nonnegative integer and G a directed graph. A *path* of length n from u to v in G is a sequence of edges e_1, e_2, \dots, e_n of G such that e_1 is associated with (x_0, x_1) , e_2 is associated with (x_1, x_2) , and so on, with e_n associated with (x_{n-1}, x_n) , where $x_0 = u$ and $x_n = v$. When there are no multiple edges in the directed graph, this path is denoted by its vertex sequence $x_0, x_1, x_2, \dots, x_n$. A path of length greater than zero that begins and ends at the same vertex is called a *circuit* or *cycle*. A path or circuit is called *simple* if it does not contain the same edge more than once.

Remark: Terminology other than that given in Definition 2 is often used for the concepts defined there. In particular, the alternative terminology that uses *walk*, *closed walk*, *trail*, and *path* (described in the remarks following Definition 1) may be used for directed graphs. See [GrYe06] for details.

Note that the terminal vertex of an edge in a path is the initial vertex of the next edge in the path. When it is not necessary to distinguish between multiple edges, we will denote a path e_1, e_2, \dots, e_n , where e_i is associated with (x_{i-1}, x_i) for $i = 1, 2, \dots, n$ by its vertex sequence x_0, x_1, \dots, x_n . The notation identifies a path only up to the vertices it passes through. There may be more than one path that passes through this sequence of vertices.

Paths represent useful information in many graph models, as Examples 2–4 demonstrate.

EXAMPLE 2 **Paths in Acquaintanceship Graphs** In an acquaintanceship graph there is a path between two people if there is a chain of people linking these people, where two people adjacent in the chain know one another. For example, in Figure 7 in Section 9.1, there is a chain of six people linking Kamini and Ching. Many social scientists have conjectured that almost every pair of people in the world are linked by a small chain of people, perhaps containing just five or fewer people. This would mean that almost every pair of vertices in the acquaintanceship graph containing all people in the world is linked by a path of length not exceeding four. The play *Six Degrees of Separation* by John Guare is based on this notion. ◀



EXAMPLE 3 **Paths in Collaboration Graphs** In a collaboration graph two vertices a and b , which represent authors, are connected by a path when there is a sequence of authors beginning at a and ending at b such that the two authors represented by the endpoints of each edge have written a joint paper. In the collaboration graph of all mathematicians, the **Erdős number** of a mathematician m (defined in terms of relations in Supplementary Exercise 14 in Chapter 8) is the length of the shortest path between m and the vertex representing the extremely prolific mathematician Paul Erdős (who died in 1996). That is, the Erdős number of a mathematician is the length of the shortest chain of mathematicians that begins with Paul Erdős and ends with this mathematician, where each adjacent pair of mathematicians have written a joint paper. The number of mathematicians with each Erdős number as of early 2006, according to the Erdős Number Project, is shown in Table 1. ◀



EXAMPLE 4 **Paths in the Hollywood Graph** In the Hollywood graph (see Example 4 in Section 9.1) two vertices a and b are linked when there is a chain of actors linking a and b , where every two actors adjacent in the chain have acted in the same movie. In the Hollywood graph, the **Bacon number** of an actor c is defined to be the length of the shortest path connecting c and the well-known actor Kevin Bacon. As new movies are made, including new ones with Kevin Bacon, the Bacon



<i>Erdős Number</i>	<i>Number of People</i>
0	1
1	504
2	6,593
3	33,605
4	83,642
5	87,760
6	40,014
7	11,591
8	3,146
9	819
10	244
11	68
12	23
13	5

<i>Bacon Number</i>	<i>Number of People</i>
0	1
1	1,902
2	160,463
3	457,231
4	111,310
5	8,168
6	810
7	81
8	14

number of actors can change. In Table 2 we show the number of actors with each Bacon number as of early 2006 using data from the Oracle of Bacon website. ◀

Connectedness In Undirected Graphs

When does a computer network have the property that every pair of computers can share information, if messages can be sent through one or more intermediate computers? When a graph is used to represent this computer network, where vertices represent the computers and edges represent the communication links, this question becomes: When is there always a path between two vertices in the graph?

DEFINITION 3 An undirected graph is called *connected* if there is a path between every pair of distinct vertices of the graph.

Thus, any two computers in the network can communicate if and only if the graph of this network is connected.

EXAMPLE 5 The graph G_1 in Figure 2 is connected, because for every pair of distinct vertices there is a path between them (the reader should verify this). However, the graph G_2 in Figure 2 is not connected. For instance, there is no path in G_2 between vertices a and d . ◀

We will need the following theorem in Chapter 10.

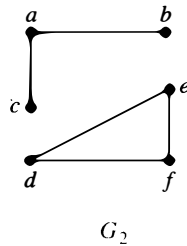
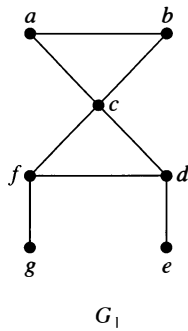


FIGURE 2 The Graphs G_1 and G_2 .

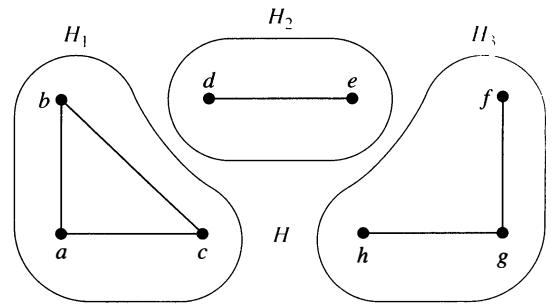


FIGURE 3 The Graph H and Its Connected Components $H_1, H_2,$ and H_3 .

THEOREM 1 There is a simple path between every pair of distinct vertices of a connected undirected graph.

Proof: Let u and v be two distinct vertices of the connected undirected graph $G = (V, E)$. Because G is connected, there is at least one path between u and v . Let x_0, x_1, \dots, x_n , where $x_0 = u$ and $x_n = v$, be the vertex sequence of a path of least length. This path of least length is simple. To see this, suppose it is not simple. Then $x_i = x_j$ for some i and j with $0 \leq i < j$. This means that there is a path from u to v of shorter length with vertex sequence $x_0, x_1, \dots, x_{i-1}, x_j, \dots, x_n$ obtained by deleting the edges corresponding to the vertex sequence x_i, \dots, x_{j-1} . ◀

A **connected component** of a graph G is a connected subgraph of G that is not a proper subgraph of another connected subgraph of G . That is, a connected component of a graph G is a maximal connected subgraph of G . A graph G that is not connected has two or more connected components that are disjoint and have G as their union.



EXAMPLE 6 What are the connected components of the graph H shown in Figure 3?

Solution: The graph H is the union of three disjoint connected subgraphs $H_1, H_2,$ and H_3 , shown in Figure 3. These three subgraphs are the connected components of H . ◀

EXAMPLE 7 Connected Components of Call Graphs Two vertices x and y are in the same component of a telephone call graph (see Example 7 in Section 9.1) when there is a sequence of telephone calls beginning at x and ending at y . When a call graph for telephone calls made during a particular day in the AT&T network was analyzed, this graph was found to have 53,767,087 vertices, more than 170 million edges, and more than 3.7 million connected components. Most of these components were small; approximately three-fourths consisted of two vertices representing pairs of telephone numbers that called only each other. This graph has one huge connected component with 44,989,297 vertices comprising more than 80% of the total. Furthermore, every vertex in this component can be linked to any other vertex by a chain of no more than 20 calls. ◀



Sometimes the removal of a vertex and all edges incident with it produces a subgraph with more connected components than in the original graph. Such vertices are called **cut vertices** (or **articulation points**). The removal of a cut vertex from a connected graph produces a subgraph that is not connected. Analogously, an edge whose removal produces a graph with more connected components than in the original graph is called a **cut edge** or **bridge**.

EXAMPLE 8 Find the cut vertices and cut edges in the graph G shown in Figure 4.

Solution: The cut vertices of G are b , c , and e . The removal of one of these vertices (and its adjacent edges) disconnects the graph. The cut edges are $\{a, b\}$ and $\{c, e\}$. Removing either one of these edges disconnects G . ◀

Connectedness in Directed Graphs

There are two notions of connectedness in directed graphs, depending on whether the directions of the edges are considered.

DEFINITION 4 A directed graph is *strongly connected* if there is a path from a to b and from b to a whenever a and b are vertices in the graph.

For a directed graph to be strongly connected there must be a sequence of directed edges from any vertex in the graph to any other vertex. A directed graph can fail to be strongly connected but still be in “one piece.” Definition 5 makes this notion precise.

DEFINITION 5 A directed graph is *weakly connected* if there is a path between every two vertices in the underlying undirected graph.

That is, a directed graph is weakly connected if and only if there is always a path between two vertices when the directions of the edges are disregarded. Clearly, any strongly connected directed graph is also weakly connected.

EXAMPLE 9 Are the directed graphs G and H shown in Figure 5 strongly connected? Are they weakly connected?

Solution: G is strongly connected because there is a path between any two vertices in this directed graph (the reader should verify this). Hence, G is also weakly connected. The graph H is not strongly connected. There is no directed path from a to b in this graph. However, H is weakly connected, because there is a path between any two vertices in the underlying undirected graph of H (the reader should verify this). ◀

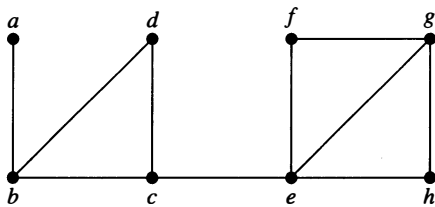


FIGURE 4 The Graph G .

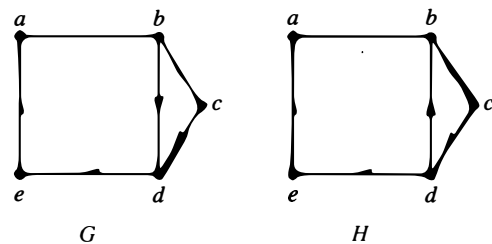


FIGURE 5 The Directed Graphs G and H .

The subgraphs of a directed graph G that are strongly connected but not contained in larger strongly connected subgraphs, that is, the maximal strongly connected subgraphs, are called the **strongly connected components** or **strong components** of G .

EXAMPLE 10 The graph H in Figure 5 has three strongly connected components, consisting of the vertex a ; the vertex e ; and the graph consisting of the vertices b, c , and d and edges (b, c) , (c, d) , and (d, b) . ◀

EXAMPLE 11 **The Strongly Connected Components of the Web Graph** The Web graph introduced in Example 8 of Section 9.1 represents Web pages with vertices and links with directed edges. A snapshot of the Web in 1999 produced a Web graph with over 200 million vertices and over 1.5 billion edges. (See [Br00] for details.) The underlying undirected graph of this Web graph is not connected and has a connected component that includes approximately 90% of the vertices in the graph. The subgraph of the original directed graph corresponding to this connected component of the underlying undirected graph (that is, with the same vertices and all directed edges connecting vertices in this graph) has one very large strongly connected component and many small ones. The former is called the **giant strongly connected component (GSCC)** of the directed graph. A Web page in this component can be reached following links starting at any other page in this component. The GSCC in the Web graph produced by this study was found to have over 53 million vertices. The remaining vertices in the large connected component of the undirected graph represent three different types of Web pages: pages that can be reached from a page in the GSCC, but do not link back to these pages following a series of links; pages that link back to pages in the GSCC following a series of links, but cannot be reached by following links on pages in the GSCC; and pages that cannot reach pages in the GSCC and cannot be reached from pages in the GSCC following a series of links. In this study, each of these three other sets was found to have approximately 44 million vertices. (It is rather surprising that these three sets are close to the same size.) ◀



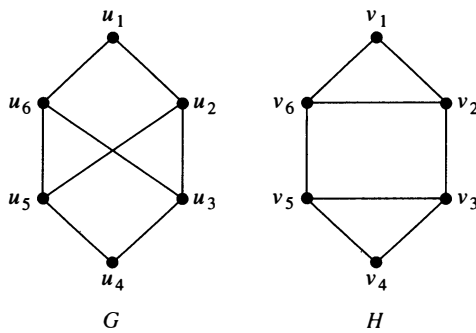
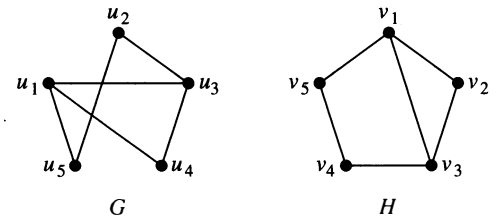
Paths and Isomorphism

There are several ways that paths and circuits can help determine whether two graphs are isomorphic. For example, the existence of a simple circuit of a particular length is a useful invariant that can be used to show that two graphs are not isomorphic. In addition, paths can be used to construct mappings that may be isomorphisms.

As we mentioned, a useful isomorphic invariant for simple graphs is the existence of a simple circuit of length k , where k is a positive integer greater than 2. (The proof that this is an invariant is left as Exercise 50 at the end of this section.) Example 12 illustrates how this invariant can be used to show that two graphs are not isomorphic.

EXAMPLE 12 Determine whether the graphs G and H shown in Figure 6 are isomorphic.

Solution: Both G and H have six vertices and eight edges. Each has four vertices of degree three, and two vertices of degree two. So, the three invariants—number of vertices, number of edges, and degrees of vertices—all agree for the two graphs. However, H has a simple circuit of length three, namely, v_1, v_2, v_6, v_1 , whereas G has no simple circuit of length three, as can be determined by inspection (all simple circuits in G have length at least four). Because the existence of a simple circuit of length three is an isomorphic invariant, G and H are not isomorphic. ◀

FIGURE 6 The Graphs G and H .FIGURE 7 The Graphs G and H .

We have shown how the existence of a type of path, namely, a simple circuit of a particular length, can be used to show that two graphs are not isomorphic. We can also use paths to find mappings that are potential isomorphisms.

EXAMPLE 13 Determine whether the graphs G and H shown in Figure 7 are isomorphic.

Solution: Both G and H have five vertices and six edges, both have two vertices of degree three and three vertices of degree two, and both have a simple circuit of length three, a simple circuit of length four, and a simple circuit of length five. Because all these isomorphic invariants agree, G and H may be isomorphic. To find a possible isomorphism, we can follow paths that go through all vertices so that the corresponding vertices in the two graphs have the same degree. For example, the paths u_1, u_4, u_3, u_2, u_5 in G and v_3, v_2, v_1, v_5, v_4 in H both go through every vertex in the graph; start at a vertex of degree three; go through vertices of degrees two, three, and two, respectively; and end at a vertex of degree two. By following these paths through the graphs, we define the mapping f with $f(u_1) = v_3$, $f(u_4) = v_2$, $f(u_3) = v_1$, $f(u_2) = v_5$, and $f(u_5) = v_4$. The reader can show that f is an isomorphism, so G and H are isomorphic, either by showing that f preserves edges or by showing that with the appropriate orderings of vertices the adjacency matrices of G and H are the same. ◀

Counting Paths Between Vertices

The number of paths between two vertices in a graph can be determined using its adjacency matrix.

THEOREM 2 Let G be a graph with adjacency matrix \mathbf{A} with respect to the ordering v_1, v_2, \dots, v_n (with directed or undirected edges, with multiple edges and loops allowed). The number of different paths of length r from v_i to v_j , where r is a positive integer, equals the (i, j) th entry of \mathbf{A}^r .

Proof: The theorem will be proved using mathematical induction. Let G be a graph with adjacency matrix \mathbf{A} (assuming an ordering v_1, v_2, \dots, v_n of the vertices of G). The number of paths from v_i to v_j of length 1 is the (i, j) th entry of \mathbf{A} , because this entry is the number of edges from v_i to v_j .

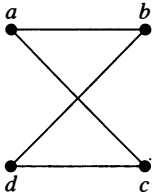
Assume that the (i, j) th entry of \mathbf{A}^r is the number of different paths of length r from v_i to v_j . This is the induction hypothesis. Because $\mathbf{A}^{r+1} = \mathbf{A}^r \mathbf{A}$, the (i, j) th entry of \mathbf{A}^{r+1} equals

$$b_{i1}a_{1j} + b_{i2}a_{2j} + \cdots + b_{in}a_{nj},$$

where b_{ik} is the (i, k) th entry of A^r . By the induction hypothesis, b_{ik} is the number of paths of length r from v_i to v_k .

A path of length $r + 1$ from v_i to v_j is made up of a path of length r from v_i to some intermediate vertex v_k , and an edge from v_k to v_j . By the product rule for counting, the number of such paths is the product of the number of paths of length r from v_i to v_k , namely, b_{ik} , and the number of edges from v_k to v_j , namely, a_{kj} . When these products are added for all possible intermediate vertices v_k , the desired result follows by the sum rule for counting. ◀

EXAMPLE 14 How many paths of length four are there from a to d in the simple graph G in Figure 8?



Solution: The adjacency matrix of G (ordering the vertices as a, b, c, d) is

$$A = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}.$$

FIGURE 8 The Graph G .

Hence, the number of paths of length four from a to d is the $(1, 4)$ th entry of A^4 . Because

$$A^4 = \begin{bmatrix} 8 & 0 & 0 & 8 \\ 0 & 8 & 8 & 0 \\ 0 & 8 & 8 & 0 \\ 8 & 0 & 0 & 8 \end{bmatrix},$$



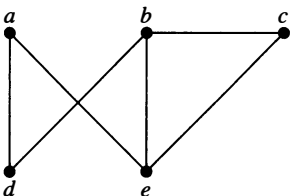
there are exactly eight paths of length four from a to d . By inspection of the graph, we see that a, b, a, b, d ; a, b, a, c, d ; a, b, d, b, d ; a, b, d, c, d ; a, c, a, b, d ; a, c, a, c, d ; a, c, d, b, d ; and a, c, d, c, d are the eight paths from a to d . ◀

Theorem 2 can be used to find the length of the shortest path between two vertices of a graph (see Exercise 46), and it can also be used to determine whether a graph is connected (see Exercises 51 and 52).

Exercises

1. Does each of these lists of vertices form a path in the following graph? Which paths are simple? Which are circuits? What are the lengths of those that are paths?

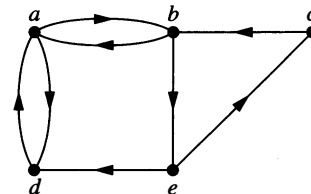
- a) a, e, b, c, b b) a, e, a, d, b, c, a
 c) e, b, a, d, b, e d) c, b, d, a, e, c



2. Does each of these lists of vertices form a path in the following graph? Which paths are simple? Which are circuits? What are the lengths of those that are paths?

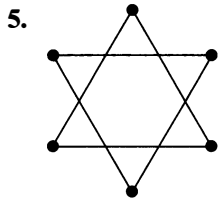
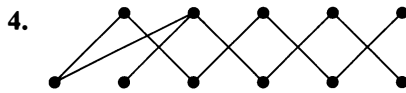
- a) a, b, e, c, b b) a, d, a, d, a

- c) a, d, b, e, a d) a, b, e, c, b, d, a

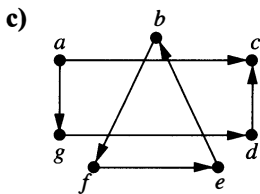
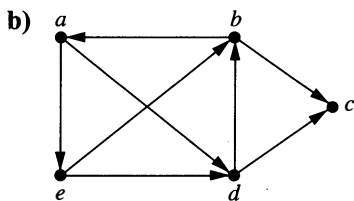
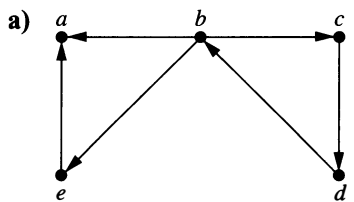


In Exercises 3–5 determine whether the given graph is connected.

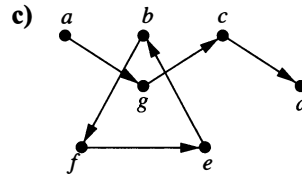
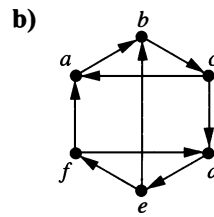
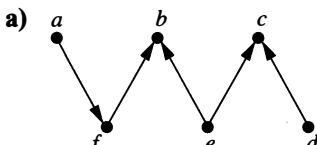




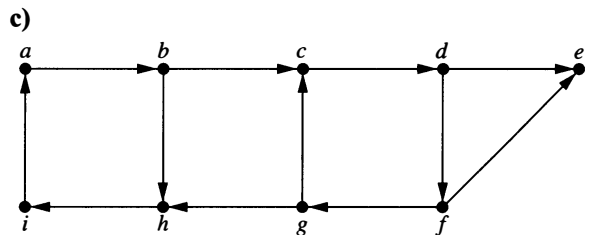
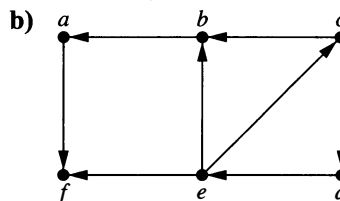
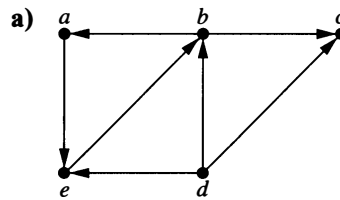
6. How many connected components does each of the graphs in Exercises 3–5 have? For each graph find each of its connected components.
7. What do the connected components of acquaintanceship graphs represent?
8. What do the connected components of a collaboration graph represent?
9. Explain why in the collaboration graph of mathematicians a vertex representing a mathematician is in the same connected component as the vertex representing Paul Erdős if and only if that mathematician has a finite Erdős number.
10. In the Hollywood graph (see Example 4 in Section 9.1), when is the vertex representing an actor in the same connected component as the vertex representing Kevin Bacon?
11. Determine whether each of these graphs is strongly connected and if not, whether it is weakly connected.



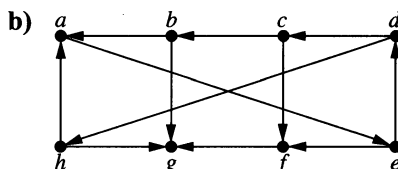
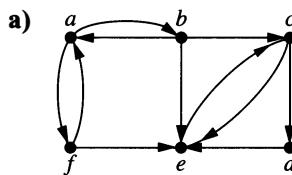
12. Determine whether each of these graphs is strongly connected and if not, whether it is weakly connected.

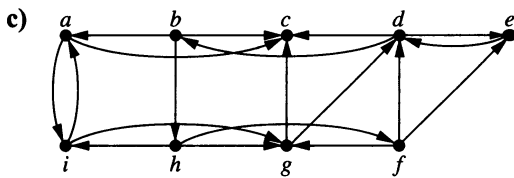


13. What do the strongly connected components of a telephone call graph represent?
14. Find the strongly connected components of each of these graphs.

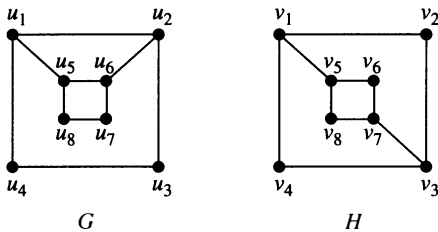


15. Find the strongly connected components of each of these graphs.

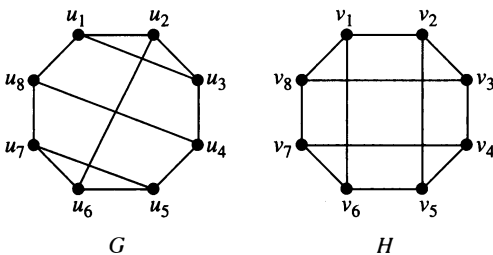




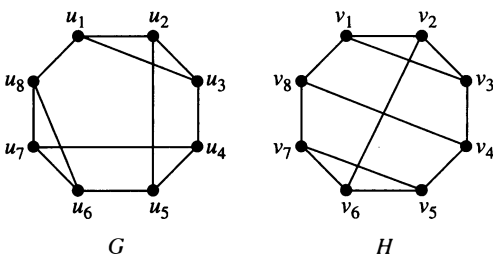
- 16. Show that all vertices visited in a directed path connecting two vertices in the same strongly connected component of a directed graph are also in this strongly connected component.
- 17. Find the number of paths of length n between two different vertices in K_4 if n is
 - a) 2. b) 3. c) 4. d) 5.
- 18. Use paths either to show that these graphs are not isomorphic or to find an isomorphism between these graphs.



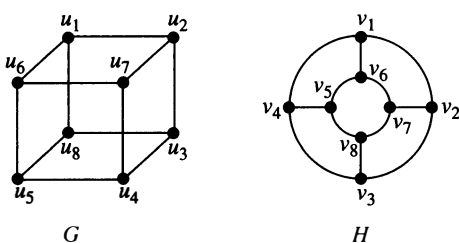
- 19. Use paths either to show that these graphs are not isomorphic or to find an isomorphism between them.



- 20. Use paths either to show that these graphs are not isomorphic or to find an isomorphism between them.



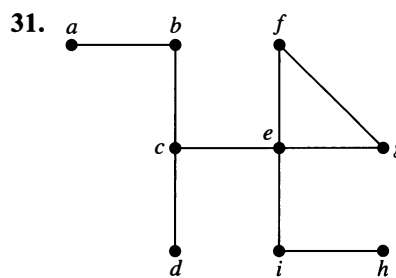
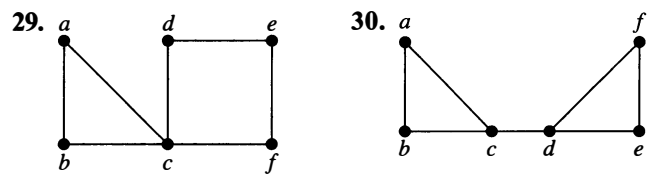
- 21. Use paths either to show that these graphs are not isomorphic or to find an isomorphism between them.



- 22. Find the number of paths of length n between any two adjacent vertices in $K_{3,3}$ for the values of n in Exercise 17.
- 23. Find the number of paths of length n between any two non-adjacent vertices in $K_{3,3}$ for the values of n in Exercise 17.
- 24. Find the number of paths between c and d in the graph in Figure 1 of length
 - a) 2. b) 3. c) 4. d) 5. e) 6. f) 7.
- 25. Find the number of paths from a to e in the directed graph in Exercise 2 of length
 - a) 2. b) 3. c) 4. d) 5. e) 6. f) 7.
- *26. Show that every connected graph with n vertices has at least $n - 1$ edges.
- 27. Let $G = (V, E)$ be a simple graph. Let R be the relation on V consisting of pairs of vertices (u, v) such that there is a path from u to v or such that $u = v$. Show that R is an equivalence relation.

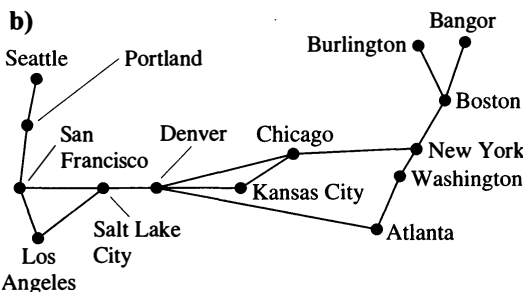
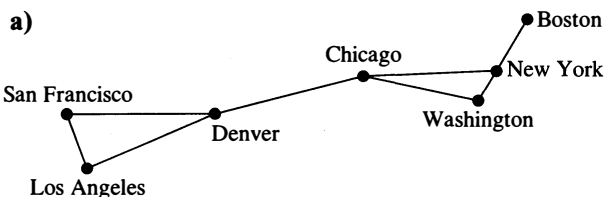
- *28. Show that in every simple graph there is a path from any vertex of odd degree to some other vertex of odd degree.

In Exercises 29–31 find all the cut vertices of the given graph.



- 29. Find all the cut edges in the graphs in Exercises 29–31.
- *33. Suppose that v is an endpoint of a cut edge. Prove that v is a cut vertex if and only if this vertex is not pendant.
- *34. Show that a vertex c in the connected simple graph G is a cut vertex if and only if there are vertices u and v , both different from c , such that every path between u and v passes through c .
- *35. Show that a simple graph with at least two vertices has at least two vertices that are not cut vertices.
- *36. Show that an edge in a simple graph is a cut edge if and only if this edge is not part of any simple circuit in the graph.
- 37. A communications link in a network should be provided with a backup link if its failure makes it impossible for some message to be sent. For each of the communications

networks shown here in (a) and (b), determine those links that should be backed up.



A **vertex basis** in a directed graph is a set of vertices such that there is a path to every vertex in the directed graph not in the set from some vertex in this set and there is no path from any vertex in the set to another vertex in the set.

- 38. Find a vertex basis for each of the directed graphs in Exercises 7–9 of Section 9.2.
- 39. What is the significance of a vertex basis in an influence graph (described in Example 3 of Section 9.1)? Find a vertex basis in the influence graph in this example.
- 40. Show that if a connected simple graph G is the union of the graphs G_1 and G_2 , then G_1 and G_2 have at least one common vertex.
- *41. Show that if a simple graph G has k connected components and these components have n_1, n_2, \dots, n_k vertices, respectively, then the number of edges of G does not exceed

$$\sum_{i=1}^k C(n_i, 2).$$

- *42. Use Exercise 41 to show that a simple graph with n vertices and k connected components has at most $(n - k)(n - k + 1)/2$ edges. [Hint: First show that

$$\sum_{i=1}^k n_i^2 \leq n^2 - (k - 1)(2n - k),$$

where n_i is the number of vertices in the i th connected component.]

- *43. Show that a simple graph G with n vertices is connected if it has more than $(n - 1)(n - 2)/2$ edges.
- 44. Describe the adjacency matrix of a graph with n connected components when the vertices of the graph are listed so that vertices in each connected component are listed successively.

- 45. How many nonisomorphic connected simple graphs are there with n vertices when n is
 - a) 2? b) 3? c) 4? d) 5?

- 46. Explain how Theorem 2 can be used to find the length of the shortest path from a vertex v to a vertex w in a graph.
- 47. Use Theorem 2 to find the length of the shortest path between a and f in the graph in Figure 1.

- 48. Use Theorem 2 to find the length of the shortest path from a to c in the directed graph in Exercise 2.

- ↳ 49. Let P_1 and P_2 be two simple paths between the vertices u and v in the simple graph G that do not contain the same set of edges. Show that there is a simple circuit in G .

- 50. Show that the existence of a simple circuit of length k , where k is a positive integer greater than 2, is an isomorphic invariant.

- 51. Explain how Theorem 2 can be used to determine whether a graph is connected.

- 52. Use Exercise 51 to show that the graph G_1 in Figure 2 is connected whereas the graph G_2 in that figure is not connected.

- 53. Show that a simple graph G is bipartite if and only if it has no circuits with an odd number of edges.

- 54. In an old puzzle attributed to Alcuin of York (735–804), a farmer needs to carry a wolf, a goat, and a cabbage across a river. The farmer only has a small boat, which can carry the farmer and only one object (an animal or a vegetable). He can cross the river repeatedly. However, if the farmer is on the other shore, the wolf will eat the goat, and, similarly, the goat will eat the cabbage. We can describe each state by listing what is on each shore. For example, we can use the pair (FG, WC) for the state where the farmer and goat are on the first shore and the wolf and cabbage are on the other shore. [The symbol \emptyset is used when nothing is on a shore, so that $(FWGC, \emptyset)$ is the initial state.]

- a) Find all allowable states of the puzzle, where neither the wolf and the goat nor the goat and the cabbage are left on the same shore without the farmer.
- b) Construct a graph such that each vertex of this graph represents an allowable state and the vertices representing two allowable states are connected by an edge if it is possible to move from one state to the other using one trip of the boat.
- c) Explain why finding a path from the vertex representing $(FWGC, \emptyset)$ to the vertex representing $(\emptyset, FWGC)$ solves the puzzle.
- d) Find two different solutions of the puzzle, each using seven crossings.
- e) Suppose that the farmer must pay a toll of one dollar whenever he crosses the river with an animal. Which solution of the puzzle should the farmer use to pay the least total toll?

- *55. Use a graph model and a path in your graph, as in Exercise 54, to solve the **jealous husbands problem**. Two married couples, each a husband and a wife, want to cross

a river. They can only use a boat that can carry one or two people from one shore to the other shore. Each husband is extremely jealous and is not willing to leave his wife with the other husband, either in the boat or on shore. How can these four people reach the opposite shore?

56. Suppose that you have a three-gallon jug and a five-gallon jug, and you may fill either jug from a water tap, you may

empty either jug, and you may transfer water from either jug into the other jug. Use a path in a directed graph model to show that you can end up with a jug containing exactly one gallon. [Hint: Use an ordered pair (a, b) to indicate how much water is in each of the jugs and represent these ordered pairs by vertices. Add edges corresponding to the allowable operations with the jugs.]

9.5 Euler and Hamilton Paths

Introduction

Can we travel along the edges of a graph starting at a vertex and returning to it by traversing each edge of the graph exactly once? Similarly, can we travel along the edges of a graph starting at a vertex and returning to it while visiting each vertex of the graph exactly once? Although these questions seem to be similar, the first question, which asks whether a graph has an *Euler circuit*, can be easily answered simply by examining the degrees of the vertices of the graph, while the second question, which asks whether a graph has a *Hamilton circuit*, is quite difficult to solve for most graphs. In this section we will study these questions and discuss the difficulty of solving them. Although both questions have many practical applications in many different areas, both arose in old puzzles. We will learn about these old puzzles as well as modern practical applications.

Euler Paths and Circuits



The town of Königsberg, Prussia (now called Kaliningrad and part of the Russian republic), was divided into four sections by the branches of the Pregel River. These four sections included the two regions on the banks of the Pregel, Kneiphof Island, and the region between the two branches of the Pregel. In the eighteenth century seven bridges connected these regions. Figure 1 depicts these regions and bridges.

The townspeople took long walks through town on Sundays. They wondered whether it was possible to start at some location in the town, travel across all the bridges without crossing any bridge twice, and return to the starting point.

The Swiss mathematician Leonhard Euler solved this problem. His solution, published in 1736, may be the first use of graph theory. Euler studied this problem using the multigraph obtained when the four regions are represented by vertices and the bridges by edges. This multigraph is shown in Figure 2.

The problem of traveling across every bridge without crossing any bridge more than once can be rephrased in terms of this model. The question becomes: Is there a simple circuit in this multigraph that contains every edge?

DEFINITION 1 An *Euler circuit* in a graph G is a simple circuit containing every edge of G . An *Euler path* in G is a simple path containing every edge of G .

Examples 1 and 2 illustrate the concept of Euler circuits and paths.

EXAMPLE 1 Which of the undirected graphs in Figure 3 have an Euler circuit? Of those that do not, which have an Euler path?

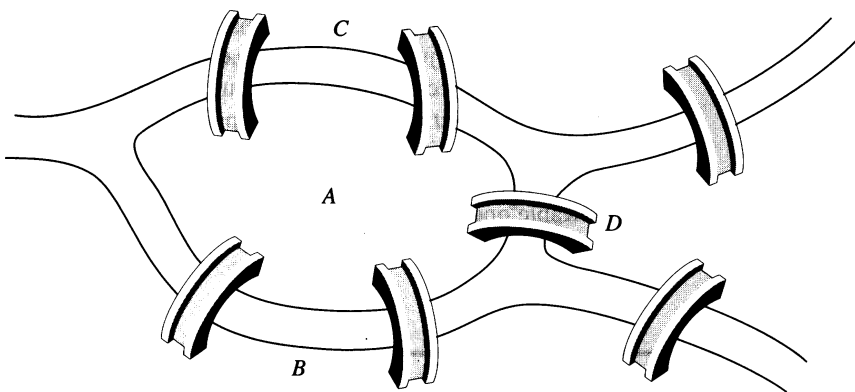


FIGURE 1 The Seven Bridges of Königsberg.

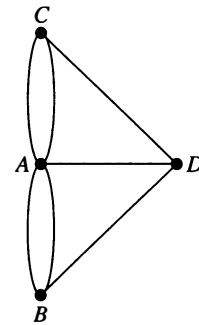


FIGURE 2 Multigraph Model of the Town of Königsberg.

Solution: The graph G_1 has an Euler circuit, for example, a, e, c, d, e, b, a . Neither of the graphs G_2 or G_3 has an Euler circuit (the reader should verify this). However, G_3 has an Euler path, namely, a, c, d, e, b, d, a, b . G_2 does not have an Euler path (as the reader should verify). ◀

EXAMPLE 2 Which of the directed graphs in Figure 4 have an Euler circuit? Of those that do not, which have an Euler path?



Solution: The graph H_2 has an Euler circuit, for example, $a, g, c, b, g, e, d, f, a$. Neither H_1 nor H_3 has an Euler circuit (as the reader should verify). H_3 has an Euler path, namely, c, a, b, c, d, b , but H_1 does not (as the reader should verify). ◀

NECESSARY AND SUFFICIENT CONDITIONS FOR EULER CIRCUITS AND PATHS
 There are simple criteria for determining whether a multigraph has an Euler circuit or an Euler path. Euler discovered them when he solved the famous Königsberg bridge problem. We will assume that all graphs discussed in this section have a finite number of vertices and edges.

What can we say if a connected multigraph has an Euler circuit? What we can show is that every vertex must have even degree. To do this, first note that an Euler circuit begins with a vertex a and continues with an edge incident with a , say $\{a, b\}$. The edge $\{a, b\}$ contributes one to $\deg(a)$. Each time the circuit passes through a vertex it contributes two to the vertex's degree, because the circuit enters via an edge incident with this vertex and leaves via another such edge. Finally, the circuit terminates where it started, contributing one to $\deg(a)$. Therefore, $\deg(a)$ must be even, because the circuit contributes one when it begins, one when it ends, and two every time it passes through a (if it ever does). A vertex other than a has even degree because the circuit contributes two to its degree each time it passes through the vertex. We conclude that if a connected graph has an Euler circuit, then every vertex must have even degree.

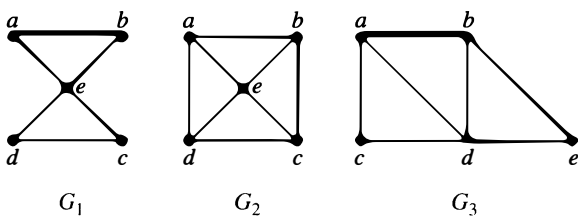


FIGURE 3 The Undirected Graphs G_1 , G_2 , and G_3 .

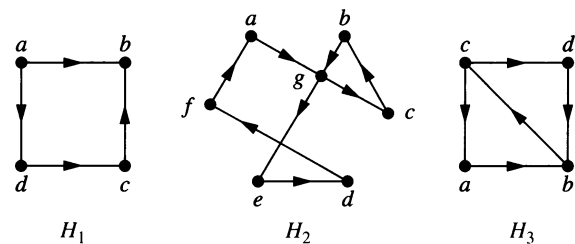


FIGURE 4 The Directed Graphs H_1 , H_2 , and H_3 .

Is this necessary condition for the existence of an Euler circuit also sufficient? That is, must an Euler circuit exist in a connected multigraph if all vertices have even degree? This question can be settled affirmatively with a construction.

Suppose that G is a connected multigraph with at least two vertices and the degree of every vertex of G is even. We will form a simple circuit that begins at an arbitrary vertex a of G . Let $x_0 = a$. First, we arbitrarily choose an edge $\{x_0, x_1\}$ incident with a which is possible because G is connected. We continue by building a simple path $\{x_0, x_1\}, \{x_1, x_2\}, \dots, \{x_{n-1}, x_n\}$, adding edges to the path until we cannot add another edge to the path. This happens when we reach a vertex for which we have already included all edges incident with that vertex in the path. For instance, in the graph G in Figure 5 we begin at a and choose in succession the edges $\{a, f\}$, $\{f, c\}$, $\{c, b\}$, and $\{b, a\}$.

The path we have constructed must terminate because the graph has a finite number of edges, so we are guaranteed to eventually reach a vertex for which no edges are available to add to the path. The path begins at a with an edge of the form $\{a, x\}$, and we now show that it must terminate at a with an edge of the form $\{y, a\}$. To see that the path must terminate at a , note that each time the path goes through a vertex with even degree, it uses only one edge to enter this vertex, so because the degree must be at least two, at least one edge remains for the path to leave the vertex. Furthermore, every time we enter and leave a vertex of even degree, there are an even number of edges incident with this vertex that we have not yet used in our path. Consequently, as we form the path, every time we enter a vertex other than a , we can leave it. This means that path can end only at a . Next, note that the path we have constructed may use all the edges of the graph, or it may not if we have returned to a for the last time before using all the edges.

An Euler circuit has been constructed if all the edges have been used. Otherwise, consider the subgraph H obtained from G by deleting the edges already used and vertices that are not incident with any remaining edges. When we delete the circuit a, f, c, b, a from the graph in Figure 5, we obtain the subgraph labeled as H .

Because G is connected, H has at least one vertex in common with the circuit that has been deleted. Let w be such a vertex. (In our example, c is the vertex.)

Every vertex in H has even degree (because in G all vertices had even degree, and for each vertex, pairs of edges incident with this vertex have been deleted to form H). Note that H may not be connected. Beginning at w , construct a simple path in H by choosing edges as long as possible, as was done in G . This path must terminate at w . For instance, in Figure 5, c, d, e, c is a path in H . Next, form a circuit in G by splicing the circuit in H with the original circuit in G (this can be done because w is one of the vertices in this circuit). When this is done in the graph in Figure 5, we obtain the circuit a, f, c, d, e, c, b, a .

Continue this process until all edges have been used. (The process must terminate because there are only a finite number of edges in the graph.) This produces an Euler circuit. The



LEONHARD EULER (1707–1783) Leonhard Euler was the son of a Calvinist minister from the vicinity of Basel, Switzerland. At 13 he entered the University of Basel, pursuing a career in theology, as his father wished. At the university Euler was tutored by Johann Bernoulli of the famous Bernoulli family of mathematicians. His interest and skills led him to abandon his theological studies and take up mathematics. Euler obtained his master's degree in philosophy at the age of 16. In 1727 Peter the Great invited him to join the Academy at St. Petersburg. In 1741 he moved to the Berlin Academy, where he stayed until 1766. He then returned to St. Petersburg, where he remained for the rest of his life.

Euler was incredibly prolific, contributing to many areas of mathematics, including number theory, combinatorics, and analysis, as well as its applications to such areas as music and naval architecture. He wrote over 1100 books and papers and left so much unpublished work that it took 47 years after he died for all his work to be published. During his life his papers accumulated so quickly that he kept a large pile of articles awaiting publication. The Berlin Academy published the papers on top of this pile so later results were often published before results they depended on or superseded. Euler had 13 children and was able to continue his work while a child or two bounced on his knees. He was blind for the last 17 years of his life, but because of his fantastic memory this did not diminish his mathematical output. The project of publishing his collected works, undertaken by the Swiss Society of Natural Science, is ongoing and will require more than 75 volumes.

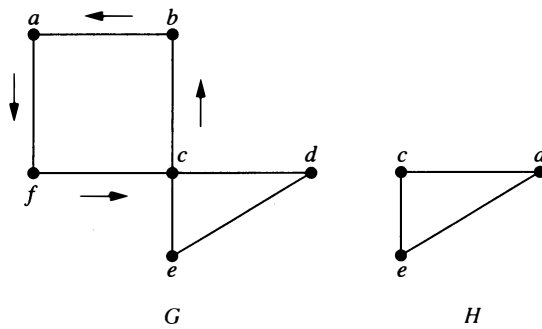


FIGURE 5 Constructing an Euler Circuit in G .

construction shows that if the vertices of a connected multigraph all have even degree, then the graph has an Euler circuit.

We summarize these results in Theorem 1.

THEOREM 1 A connected multigraph with at least two vertices has an Euler circuit if and only if each of its vertices has even degree.

We can now solve the Königsberg bridge problem. Because the multigraph representing these bridges, shown in Figure 2, has four vertices of odd degree, it does not have an Euler circuit. There is no way to start at a given point, cross each bridge exactly once, and return to the starting point.

Algorithm 1 gives the constructive procedure for finding Euler circuits given in the discussion preceding Theorem 1. (Because the circuits in the procedure are chosen arbitrarily, there is some ambiguity. We will not bother to remove this ambiguity by specifying the steps of the procedure more precisely.)

ALGORITHM 1 Constructing Euler Circuits.

```

procedure Euler( $G$ : connected multigraph with all vertices of
    even degree)
  circuit := a circuit in  $G$  beginning at an arbitrarily chosen
    vertex with edges successively added to form a path that
    returns to this vertex
   $H := G$  with the edges of this circuit removed
  while  $H$  has edges
  begin
    subcircuit := a circuit in  $H$  beginning at a vertex in  $H$  that
      also is an endpoint of an edge of circuit
     $H := H$  with edges of subcircuit and all isolated vertices
      removed
    circuit := circuit with subcircuit inserted at the appropriate
      vertex
  end {circuit is an Euler circuit}
  
```

Example 3 shows how Euler paths and circuits can be used to solve a type of puzzle.

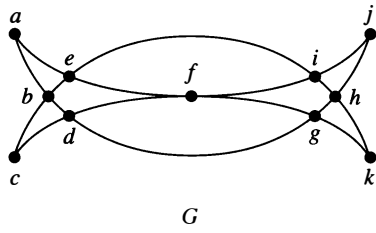


FIGURE 6 Mohammed’s Scimitars.

EXAMPLE 3 Many puzzles ask you to draw a picture in a continuous motion without lifting a pencil so that no part of the picture is retraced. We can solve such puzzles using Euler circuits and paths. For example, can *Mohammed’s scimitars*, shown in Figure 6, be drawn in this way, where the drawing begins and ends at the same point?

Solution: We can solve this problem because the graph G shown in Figure 6 has an Euler circuit. It has such a circuit because all its vertices have even degree. We will use Algorithm 1 to construct an Euler circuit. First, we form the circuit $a, b, d, c, b, e, i, f, e, a$. We obtain the subgraph H by deleting the edges in this circuit and all vertices that become isolated when these edges are removed. Then we form the circuit $d, g, h, j, i, h, k, g, f, d$ in H . After forming this circuit we have used all edges in G . Splicing this new circuit into the first circuit at the appropriate place produces the Euler circuit $a, b, d, g, h, j, i, h, k, g, f, d, c, b, e, i, f, e, a$. This circuit gives a way to draw the scimitars without lifting the pencil or retracing part of the picture. ◀

Another algorithm for constructing Euler circuits, called Fleury’s algorithm, is described in the exercises at the end of this section.

We will now show that a connected multigraph has an Euler path (and not an Euler circuit) if and only if it has exactly two vertices of odd degree. First, suppose that a connected multigraph does have an Euler path from a to b , but not an Euler circuit. The first edge of the path contributes one to the degree of a . A contribution of two to the degree of a is made every time the path passes through a . The last edge in the path contributes one to the degree of b . Every time the path goes through b there is a contribution of two to its degree. Consequently, both a and b have odd degree. Every other vertex has even degree, because the path contributes two to the degree of a vertex whenever it passes through it.

Now consider the converse. Suppose that a graph has exactly two vertices of odd degree, say a and b . Consider the larger graph made up of the original graph with the addition of an edge $\{a, b\}$. Every vertex of this larger graph has even degree, so there is an Euler circuit. The removal of the new edge produces an Euler path in the original graph. Theorem 2 summarizes these results.

THEOREM 2 A connected multigraph has an Euler path but not an Euler circuit if and only if it has exactly two vertices of odd degree.

EXAMPLE 4 Which graphs shown in Figure 7 have an Euler path?

Solution: G_1 contains exactly two vertices of odd degree, namely, b and d . Hence, it has an Euler path that must have b and d as its endpoints. One such Euler path is d, a, b, c, d, b . Similarly,

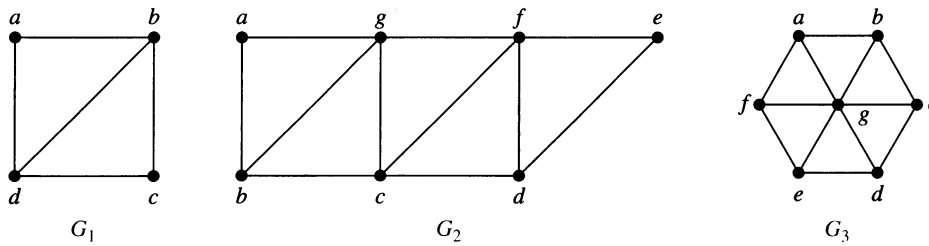


FIGURE 7 Three Undirected Graphs.

G_2 has exactly two vertices of odd degree, namely, b and d . So it has an Euler path that must have b and d as endpoints. One such Euler path is $b, a, g, f, e, d, c, g, b, c, f, d$. G_3 has no Euler path because it has six vertices of odd degree. ◀

Returning to eighteenth-century Königsberg, is it possible to start at some point in the town, travel across all the bridges, and end up at some other point in town? This question can be answered by determining whether there is an Euler path in the multigraph representing the bridges in Königsberg. Because there are four vertices of odd degree in this multigraph, there is no Euler path, so such a trip is impossible.

Necessary and sufficient conditions for Euler paths and circuits in directed graphs are discussed in the exercises at the end of this section.



Euler paths and circuits can be used to solve many practical problems. For example, many applications ask for a path or circuit that traverses each street in a neighborhood, each road in a transportation network, each connection in a utility grid, or each link in a communications network exactly once. Finding an Euler path or circuit in the appropriate graph model can solve such problems. For example, if a postman can find an Euler path in the graph that represents the streets the postman needs to cover, this path produces a route that traverses each street of the route exactly once. If no Euler path exists, some streets will have to be traversed more than once. This problem is known as the *Chinese postman problem* in honor of Guan Meigu, who posed it in 1962. See [MiRo91] for more information on the solution of the Chinese postman problem when no Euler path exists.

Among the other areas where Euler circuits and paths are applied is in layout of circuits, in network multicasting, and in molecular biology, where Euler paths are used in the sequencing of DNA.

Hamilton Paths and Circuits



We have developed necessary and sufficient conditions for the existence of paths and circuits that contain every edge of a multigraph exactly once. Can we do the same for simple paths and circuits that contain every vertex of the graph exactly once?

DEFINITION 2

A simple path in a graph G that passes through every vertex exactly once is called a *Hamilton path*, and a simple circuit in a graph G that passes through every vertex exactly once is called a *Hamilton circuit*. That is, the simple path $x_0, x_1, \dots, x_{n-1}, x_n$ in the graph $G = (V, E)$ is a Hamilton path if $V = \{x_0, x_1, \dots, x_{n-1}, x_n\}$ and $x_i \neq x_j$ for $0 \leq i < j \leq n$, and the simple circuit $x_0, x_1, \dots, x_{n-1}, x_n, x_0$ (with $n > 0$) is a Hamilton circuit if $x_0, x_1, \dots, x_{n-1}, x_n$ is a Hamilton path.

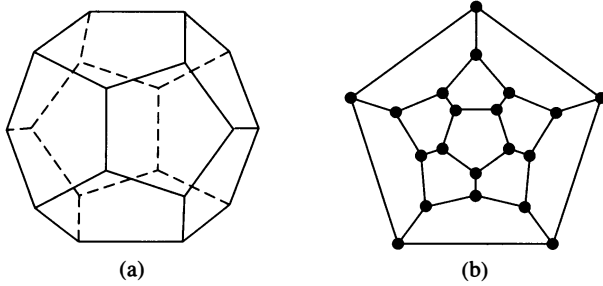


FIGURE 8 Hamilton’s “A Voyage Round the World” Puzzle.

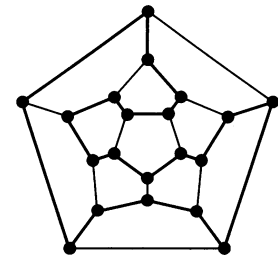


FIGURE 9 A Solution to the “A Voyage Round the World” Puzzle.

This terminology comes from a game, called the *Icosian puzzle*, invented in 1857 by the Irish mathematician Sir William Rowan Hamilton. It consisted of a wooden dodecahedron [a polyhedron with 12 regular pentagons as faces, as shown in Figure 8(a)], with a peg at each vertex of the dodecahedron, and string. The 20 vertices of the dodecahedron were labeled with different cities in the world. The object of the puzzle was to start at a city and travel along the edges of the dodecahedron, visiting each of the other 19 cities exactly once, and end back at the first city. The circuit traveled was marked off using the strings and pegs.

Because the author cannot supply each reader with a wooden solid with pegs and string, we will consider the equivalent question: Is there a circuit in the graph shown in Figure 8(b) that passes through each vertex exactly once? This solves the puzzle because this graph is isomorphic to the graph consisting of the vertices and edges of the dodecahedron. A solution of Hamilton’s puzzle is shown in Figure 9.

EXAMPLE 5 Which of the simple graphs in Figure 10 have a Hamilton circuit or, if not, a Hamilton path?



Solution: G_1 has a Hamilton circuit: a, b, c, d, e, a . There is no Hamilton circuit in G_2 (this can be seen by noting that any circuit containing every vertex must contain the edge $\{a, b\}$ twice), but G_2 does have a Hamilton path, namely, a, b, c, d . G_3 has neither a Hamilton circuit nor a Hamilton path, because any path containing all vertices must contain one of the edges $\{a, b\}$, $\{e, f\}$, and $\{c, d\}$ more than once. ◀

Is there a simple way to determine whether a graph has a Hamilton circuit or path? At first, it might seem that there should be an easy way to determine this, because there is a simple way to answer the similar question of whether a graph has an Euler circuit. Surprisingly, there are no known simple necessary and sufficient criteria for the existence of Hamilton circuits. However, many theorems are known that give sufficient conditions for the existence of Hamilton circuits.

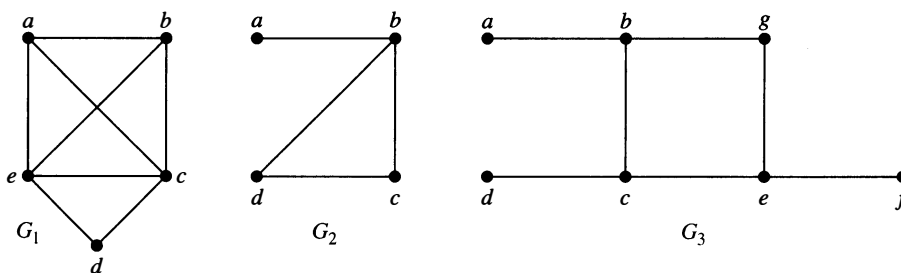


FIGURE 10 Three Simple Graphs.

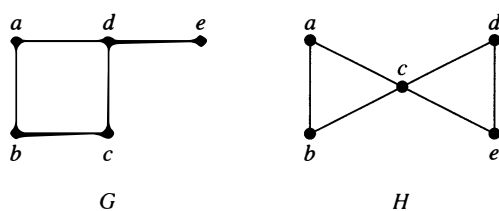
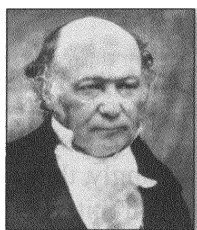


FIGURE 11 Two Graphs That Do Not Have a Hamilton Circuit.

Also, certain properties can be used to show that a graph has no Hamilton circuit. For instance, a graph with a vertex of degree one cannot have a Hamilton circuit, because in a Hamilton circuit, each vertex is incident with two edges in the circuit. Moreover, if a vertex in the graph has degree two, then both edges that are incident with this vertex must be part of any Hamilton circuit. Also, note that when a Hamilton circuit is being constructed and this circuit has passed through a vertex, then all remaining edges incident with this vertex, other than the two used in the circuit, can be removed from consideration. Furthermore, a Hamilton circuit cannot contain a smaller circuit within it.

EXAMPLE 6 Show that neither graph displayed in Figure 11 has a Hamilton circuit.

Solution: There is no Hamilton circuit in G because G has a vertex of degree one, namely, e . Now consider H . Because the degrees of the vertices a , b , d , and e are all two, every edge incident with these vertices must be part of any Hamilton circuit. It is now easy to see that no Hamilton circuit can exist in H , for any Hamilton circuit would have to contain four edges incident with c , which is impossible. ◀



WILLIAM ROWAN HAMILTON (1805–1865) William Rowan Hamilton, the most famous Irish scientist ever to have lived, was born in 1805 in Dublin. His father was a successful lawyer, his mother came from a family noted for their intelligence, and he was a child prodigy. By the age of 3 he was an excellent reader and had mastered advanced arithmetic. Because of his brilliance, he was sent off to live with his uncle James, a noted linguist. By age 8 Hamilton had learned Latin, Greek, and Hebrew; by 10 he had also learned Italian and French and he began his study of oriental languages, including Arabic, Sanskrit, and Persian. During this period he took pride in knowing as many languages as his age. At 17, no longer devoted to learning new languages and having mastered calculus and much mathematical astronomy, he began original work in optics, and he also found an important mistake in Laplace's work on celestial mechanics.

Before entering Trinity College, Dublin, at 18, Hamilton had not attended school; rather, he received private tutoring. At Trinity, he was a superior student in both the sciences and the classics. Prior to receiving his degree, because of his brilliance he was appointed the Astronomer Royal of Ireland, beating out several famous astronomers for the post. He held this position until his death, living and working at Dunsink Observatory outside of Dublin. Hamilton made important contributions to optics, abstract algebra, and dynamics. Hamilton invented algebraic objects called quaternions as an example of a noncommutative system. He discovered the appropriate way to multiply quaternions while walking along a canal in Dublin. In his excitement, he carved the formula in the stone of a bridge crossing the canal, a spot marked today by a plaque. Later, Hamilton remained obsessed with quaternions, working to apply them to other areas of mathematics, instead of moving to new areas of research.

In 1857 Hamilton invented "The Icosian Game" based on his work in noncommutative algebra. He sold the idea for 25 pounds to a dealer in games and puzzles. (Because the game never sold well, this turned out to be a bad investment for the dealer.) The "Traveler's Dodecahedron," also called "A Voyage Round the World," the puzzle described in this section, is a variant of that game.

Hamilton married his third love in 1833, but his marriage worked out poorly, because his wife, a semi-invalid, was unable to cope with his household affairs. He suffered from alcoholism and lived reclusively for the last two decades of his life. He died from gout in 1865, leaving masses of papers containing unpublished research. Mixed in with these papers were a large number of dinner plates, many containing the remains of desiccated, uneaten chops.

EXAMPLE 7 Show that K_n has a Hamilton circuit whenever $n \geq 3$.

Solution. We can form a Hamilton circuit in K_n beginning at any vertex. Such a circuit can be built by visiting vertices in any order we choose, as long as the path begins and ends at the same vertex and visits each other vertex exactly once. This is possible because there are edges in K_n between any two vertices. ◀

Although no useful necessary and sufficient conditions for the existence of Hamilton circuits are known, quite a few sufficient conditions have been found. Note that the more edges a graph has, the more likely it is to have a Hamilton circuit. Furthermore, adding edges (but not vertices) to a graph with a Hamilton circuit produces a graph with the same Hamilton circuit. So as we add edges to a graph, especially when we make sure to add edges to each vertex, we make it increasingly likely that a Hamilton circuit exists in this graph. Consequently, we would expect there to be sufficient conditions for the existence of Hamilton circuits that depend on the degrees of vertices being sufficiently large. We state two of the most important sufficient conditions here. These conditions were found by Gabriel A. Dirac in 1952 and Oystein Ore in 1960.

THEOREM 3 **DIRAC'S THEOREM** If G is a simple graph with n vertices with $n \geq 3$ such that the degree of every vertex in G is at least $n/2$, then G has a Hamilton circuit.

THEOREM 4 **ORE'S THEOREM** If G is a simple graph with n vertices with $n \geq 3$ such that $\deg(u) + \deg(v) \geq n$ for every pair of nonadjacent vertices u and v in G , then G has a Hamilton circuit.

The proof of Ore's Theorem is outlined in Exercise 65 at the end of this section. Dirac's Theorem can be proved as a corollary to Ore's Theorem because the conditions of Dirac's Theorem imply those of Ore's Theorem.

Both Ore's Theorem and Dirac's Theorem provide sufficient conditions for a connected simple graph to have a Hamilton circuit. However, these theorems do not provide necessary conditions for the existence of a Hamilton circuit. For example, the graph C_5 has a Hamilton circuit but does not satisfy the hypotheses of either Ore's Theorem or Dirac's Theorem, as the reader can verify.



The best algorithms known for finding a Hamilton circuit in a graph or determining that no such circuit exists have exponential worst-case time complexity (in the number of vertices of the graph). Finding an algorithm that solves this problem with polynomial worst-case time complexity would be a major accomplishment because it has been shown that this problem is NP-complete (see Section 3.3). Consequently, the existence of such an algorithm would imply that many other seemingly intractable problems could be solved using algorithms with polynomial worst-case time complexity.

Hamilton paths and circuits can be used to solve practical problems. For example, many applications ask for a path or circuit that visits each road intersection in a city, each place pipelines intersect in a utility grid, or each node in a communications network exactly once. Finding a Hamilton path or circuit in the appropriate graph model can solve such problems. The famous **traveling salesman problem** asks for the shortest route a traveling salesman should take to visit a set of cities. This problem reduces to finding a Hamilton circuit in a complete graph such that the total weight of its edges is as small as possible. We will return to this question in Section 9.6.

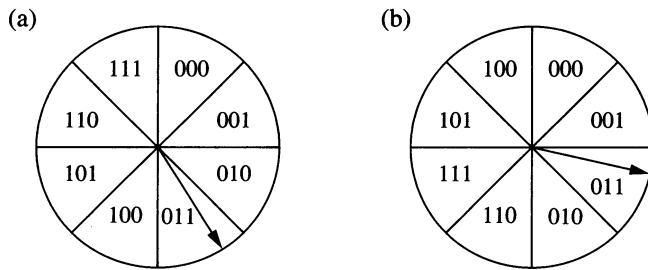


FIGURE 12 Converting the Position of a Pointer into Digital Form.

We now describe a less obvious application of Hamilton circuits to coding.

EXAMPLE 8 Gray Codes The position of a rotating pointer can be represented in digital form. One way to do this is to split the circle into 2^n arcs of equal length and to assign a bit string of length n to each arc. Two ways to do this using bit strings of length three are shown in Figure 12.

The digital representation of the position of the pointer can be determined using a set of n contacts. Each contact is used to read one bit in the digital representation of the position. This is illustrated in Figure 13 for the two assignments from Figure 12.

When the pointer is near the boundary of two arcs, a mistake may be made in reading its position. This may result in a major error in the bit string read. For instance, in the coding

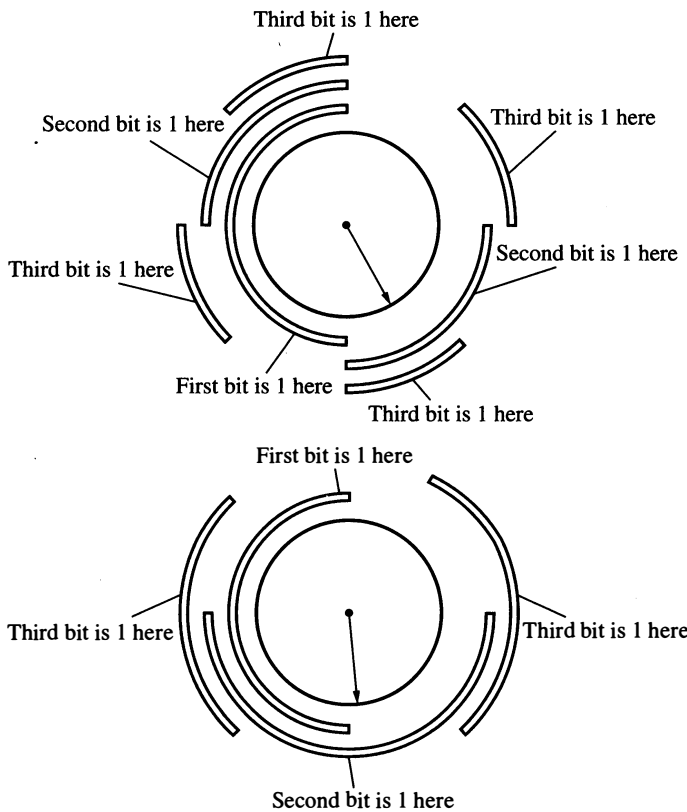


FIGURE 13 The Digital Representation of the Position of the Pointer.

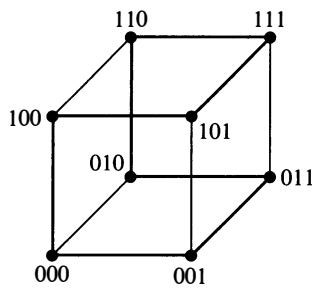


FIGURE 14 A Hamilton Circuit for Q_3 .

scheme in Figure 12(a), if a small error is made in determining the position of the pointer, the bit string 100 is read instead of 011. All three bits are incorrect! To minimize the effect of an error in determining the position of the pointer, the assignment of the bit strings to the 2^n arcs should be made so that only one bit is different in the bit strings represented by adjacent arcs. This is exactly the situation in the coding scheme in Figure 12(b). An error in determining the position of the pointer gives the bit string 010 instead of 011. Only one bit is wrong.

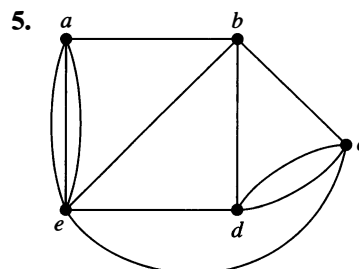
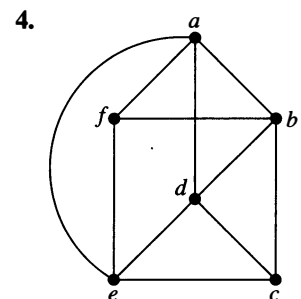
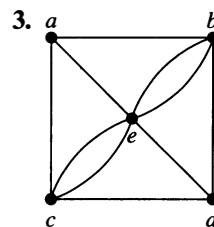
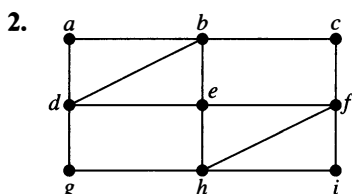
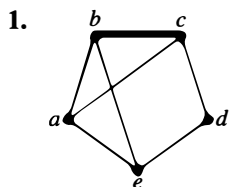


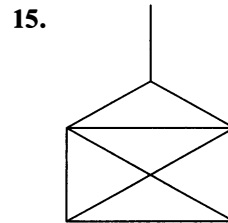
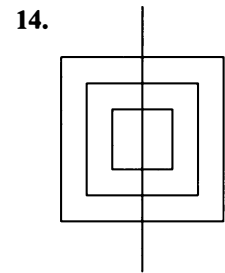
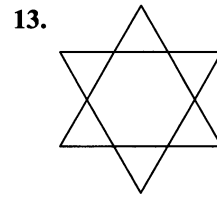
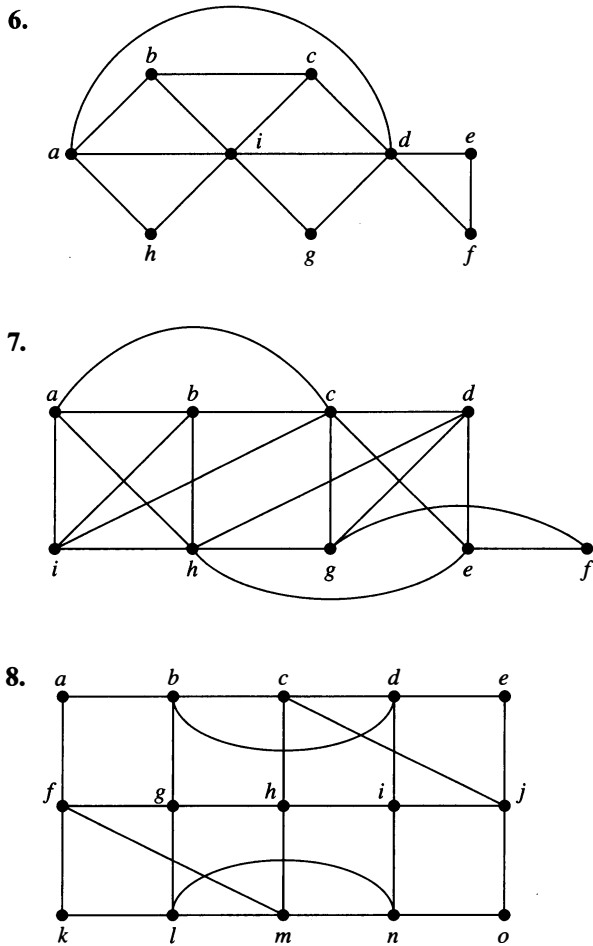
A **Gray code** is a labeling of the arcs of the circle such that adjacent arcs are labeled with bit strings that differ in exactly one bit. The assignment in Figure 12(b) is a Gray code. We can find a Gray code by listing all bit strings of length n in such a way that each string differs in exactly one position from the preceding bit string, and the last string differs from the first in exactly one position. We can model this problem using the n -cube Q_n . What is needed to solve this problem is a Hamilton circuit in Q_n . Such Hamilton circuits are easily found. For instance, a Hamilton circuit for Q_3 is displayed in Figure 14. The sequence of bit strings differing in exactly one bit produced by this Hamilton circuit is 000, 001, 011, 010, 110, 111, 101, 100.

Gray codes are named after Frank Gray, who invented them in the 1940s at AT&T Bell Laboratories to minimize the effect of errors in transmitting digital signals. ◀

Exercises

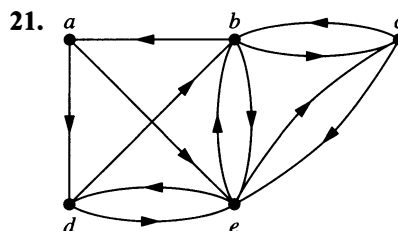
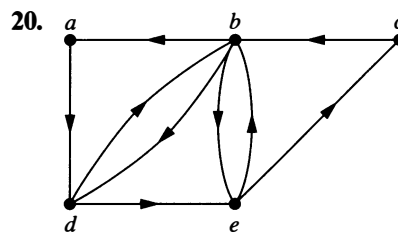
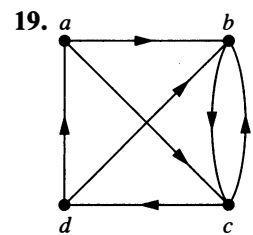
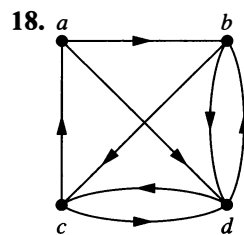
In Exercises 1–8 determine whether the given graph has an Euler circuit. Construct such a circuit when one exists. If no Euler circuit exists, determine whether the graph has an Euler path and construct such a path if one exists.





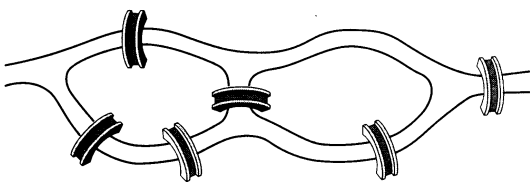
- *16. Show that a directed multigraph having no isolated vertices has an Euler circuit if and only if the graph is weakly connected and the in-degree and out-degree of each vertex are equal.
- *17. Show that a directed multigraph having no isolated vertices has an Euler path but not an Euler circuit if and only if the graph is weakly connected and the in-degree and out-degree of each vertex are equal for all but two vertices, one that has in-degree one larger than its out-degree and the other that has out-degree one larger than its in-degree.

In Exercises 18–23 determine whether the directed graph shown has an Euler circuit. Construct an Euler circuit if one exists. If no Euler circuit exists, determine whether the directed graph has an Euler path. Construct an Euler path if one exists.



9. In Kaliningrad (the Russian name for Königsberg) there are two additional bridges, besides the seven that were present in the eighteenth century. These new bridges connect regions *B* and *C* and regions *B* and *D*, respectively. Can someone cross all nine bridges in Kaliningrad exactly once and return to the starting point?

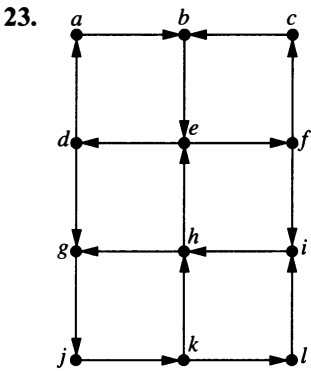
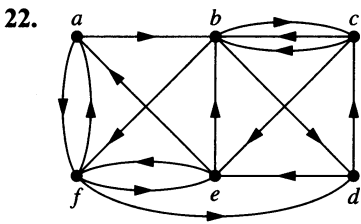
10. Can someone cross all the bridges shown in this map exactly once and return to the starting point?



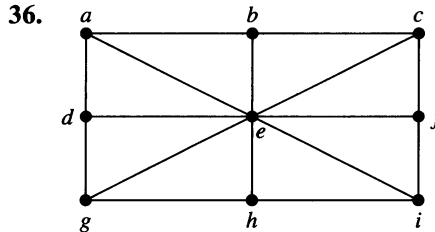
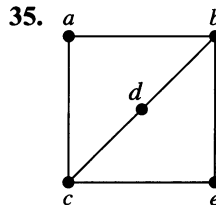
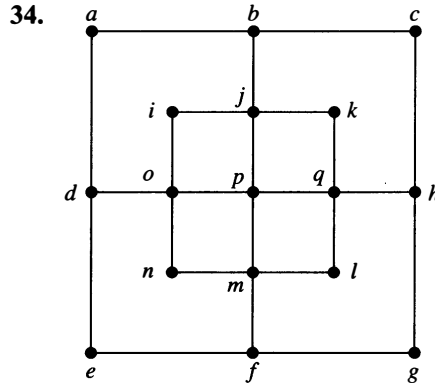
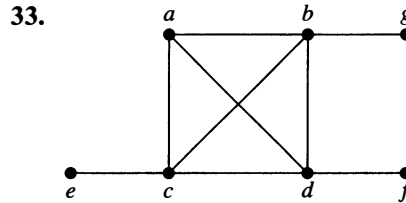
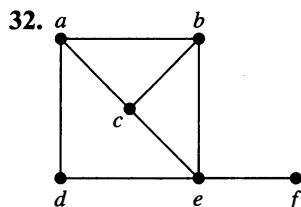
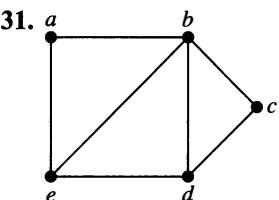
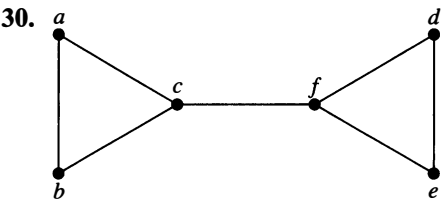
11. When can the centerlines of the streets in a city be painted without traveling a street more than once? (Assume that all the streets are two-way streets.)

12. Devise a procedure, similar to Algorithm 1, for constructing Euler paths in multigraphs.

In Exercises 13–15 determine whether the picture shown can be drawn with a pencil in a continuous motion without lifting the pencil or retracing part of the picture.

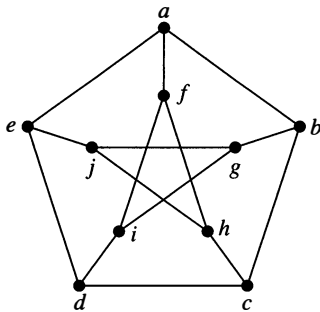


- *24. Devise an algorithm for constructing Euler circuits in directed graphs.
25. Devise an algorithm for constructing Euler paths in directed graphs.
26. For which values of n do these graphs have an Euler circuit?
- a) K_n b) C_n c) W_n d) Q_n
27. For which values of n do the graphs in Exercise 26 have an Euler path but no Euler circuit?
28. For which values of m and n does the complete bipartite graph $K_{m,n}$ have an
- a) Euler circuit?
b) Euler path?
29. Find the least number of times it is necessary to lift a pencil from the paper when drawing each of the graphs in Exercises 1–7 without retracing any part of the graph.
- In Exercises 30–36 determine whether the given graph has a Hamilton circuit. If it does, find such a circuit. If it does not, give an argument to show why no such circuit exists.

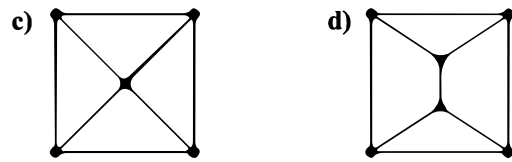
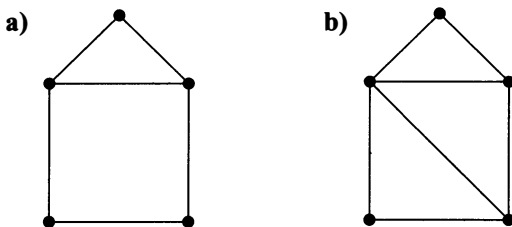


37. Does the graph in Exercise 30 have a Hamilton path? If so, find such a path. If it does not, give an argument to show why no such path exists.
38. Does the graph in Exercise 31 have a Hamilton path? If so, find such a path. If it does not, give an argument to show why no such path exists.
39. Does the graph in Exercise 32 have a Hamilton path? If so, find such a path. If it does not, give an argument to show why no such path exists.
40. Does the graph in Exercise 33 have a Hamilton path? If so, find such a path. If it does not, give an argument to show why no such path exists.
- *41. Does the graph in Exercise 34 have a Hamilton path? If so, find such a path. If it does not, give an argument to show why no such path exists.
42. Does the graph in Exercise 35 have a Hamilton path? If so, find such a path. If it does not, give an argument to show why no such path exists.
43. Does the graph in Exercise 36 have a Hamilton path? If so, find such a path. If it does not, give an argument to show why no such path exists.

- 44. For which values of n do the graphs in Exercise 26 have a Hamilton circuit?
- 45. For which values of m and n does the complete bipartite graph $K_{m,n}$ have a Hamilton circuit?
- *46. Show that the **Petersen graph**, shown here, does not have a Hamilton circuit, but that the subgraph obtained by deleting a vertex v , and all edges incident with v , does have a Hamilton circuit.



- 47. For each of these graphs, determine (i) whether Dirac's Theorem can be used to show that the graph has a Hamilton circuit, (ii) whether Ore's Theorem can be used to show that the graph has a Hamilton circuit, and (iii) whether the graph has a Hamilton circuit.



- 48. Can you find a simple graph with n vertices with $n \geq 3$ that does not have a Hamilton circuit, yet the degree of every vertex in the graph is at least $(n - 1)/2$?
- *49. Show that there is a Gray code of order n whenever n is a positive integer, or equivalently, show that the n -cube Q_n , $n > 1$, always has a Hamilton circuit. [Hint: Use mathematical induction. Show how to produce a Gray code of order n from one of order $n - 1$.]
- Fleury's algorithm** for constructing Euler circuits begins with an arbitrary vertex of a connected multigraph and forms a circuit by choosing edges successively. Once an edge is chosen, it is removed. Edges are chosen successively so that each edge begins where the last edge ends, and so that this edge is not a cut edge unless there is no alternative.
- 50. Use Fleury's algorithm to find an Euler circuit in the graph G in Figure 5.
- *51. Express Fleury's algorithm in pseudocode.
- **52. Prove that Fleury's algorithm always produces an Euler circuit.
- *53. Give a variant of Fleury's algorithm to produce Euler paths.
- 54. A diagnostic message can be sent out over a computer network to perform tests over all links and in all devices. What sort of paths should be used to test all links? To test all devices?
- 55. Show that a bipartite graph with an odd number of vertices does not have a Hamilton circuit.



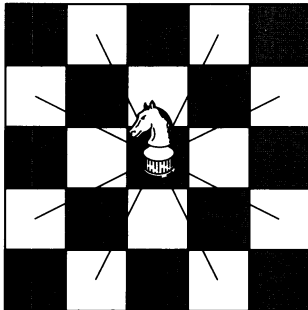
JULIUS PETER CHRISTIAN PETERSEN (1839–1910) Julius Petersen was born in the Danish town of Sorø. His father was a dyer. In 1854 his parents were no longer able to pay for his schooling, so he became an apprentice in an uncle's grocery store. When this uncle died, he left Petersen enough money to return to school. After graduating, he began studying engineering at the Polytechnical School in Copenhagen, later deciding to concentrate on mathematics. He published his first textbook, a book on logarithms, in 1858. When his inheritance ran out, he had to teach to make a living. From 1859 until 1871 Petersen taught at a prestigious private high school in Copenhagen. While teaching high school he continued his studies, entering Copenhagen University in 1862. He married Laura Bertelsen in 1862; they had three children, two sons and a daughter.

Petersen obtained a mathematics degree from Copenhagen University in 1866 and finally obtained his doctorate in 1871 from that school. After receiving his doctorate, he taught at a polytechnic and military academy. In 1887 he was appointed to a professorship at the University of Copenhagen. Petersen was well known in Denmark as the author of a large series of textbooks for high schools and universities. One of his books, *Methods and Theories for the Solution of Problems of Geometrical Construction*, was translated into eight languages, with the English language version last reprinted in 1960 and the French version reprinted as recently as 1990, more than a century after the original publication date.

Petersen worked in a wide range of areas, including algebra, analysis, cryptography, geometry, mechanics, mathematical economics, and number theory. His contributions to graph theory, including results on regular graphs, are his best-known work. He was noted for his clarity of exposition, problem-solving skills, originality, sense of humor, vigor, and teaching. One interesting fact about Petersen was that he preferred not to read the writings of other mathematicians. This led him often to rediscover results already proved by others, often with embarrassing consequences. However, he was often angry when other mathematicians did not read his writings!

Petersen's death was front-page news in Copenhagen. A newspaper of the time described him as the Hans Christian Andersen of science—a child of the people who made good in the academic world.

A **knight** is a chess piece that can move either two spaces horizontally and one space vertically or one space horizontally and two spaces vertically. That is, a knight on square (x, y) can move to any of the eight squares $(x \pm 2, y \pm 1)$, $(x \pm 1, y \pm 2)$, if these squares are on the chessboard, as illustrated here.



A **knight's tour** is a sequence of legal moves by a knight starting at some square and visiting each square exactly once. A knight's tour is called **reentrant** if there is a legal move that takes the knight from the last square of the tour back to where the tour began. We can model knight's tours using the graph that has a vertex for each square on the board, with an edge connecting two vertices if a knight can legally move between the squares represented by these vertices.

56. Draw the graph that represents the legal moves of a knight on a 3×3 chessboard.
57. Draw the graph that represents the legal moves of a knight on a 3×4 chessboard.
58. a) Show that finding a knight's tour on an $m \times n$ chessboard is equivalent to finding a Hamilton path on the graph representing the legal moves of a knight on that board.
 b) Show that finding a reentrant knight's tour on an $m \times n$ chessboard is equivalent to finding a Hamilton circuit on the corresponding graph.
- *59. Show that there is a knight's tour on a 3×4 chessboard.
- *60. Show that there is no knight's tour on a 3×3 chessboard.
- *61. Show that there is no knight's tour on a 4×4 chessboard.
62. Show that the graph representing the legal moves of a knight on an $m \times n$ chessboard, whenever m and n are positive integers, is bipartite.
63. Show that there is no reentrant knight's tour on an $m \times n$ chessboard when m and n are both odd. [Hint: Use Exercises 55, 58b, and 62.]
- *64. Show that there is a knight's tour on an 8×8 chessboard. [Hint: You can construct a knight's tour using a method invented by H. C. Warnsdorff in 1823: Start in any square, and then always move to a square connected to the fewest number of unused squares. Although this method may not always produce a knight's tour, it often does.]
65. The parts of this exercise outline a proof of Ore's Theorem. Suppose that G is a simple graph with n vertices, $n \geq 3$, and $\deg(x) + \deg(y) \geq n$ whenever x and y are nonadjacent vertices in G . Ore's Theorem states that under these conditions, G has a Hamilton circuit.
- Show that if G does not have a Hamilton circuit, then there exists another graph H with the same vertices as G , which can be constructed by adding edges to G such that the addition of a single edge would produce a Hamilton circuit in H . [Hint: Add as many edges as possible at each successive vertex of G without producing a Hamilton circuit.]
 - Show that there is a Hamilton path in H .
 - Let v_1, v_2, \dots, v_n be a Hamilton path in H . Show that $\deg(v_1) + \deg(v_n) \geq n$ and that there are at most $\deg(v_1)$ vertices not adjacent to v_n (including v_n itself).
 - Let S be the set of vertices preceding each vertex adjacent to v_1 in the Hamilton path. Show that S contains $\deg(v_1)$ vertices and $v_n \notin S$.
 - Show that S contains a vertex v_k , which is adjacent to v_n , implying that there are edges connecting v_1 and v_{k+1} and v_k and v_n .
 - Show that part (e) implies that $v_1, v_2, \dots, v_{k-1}, v_k, v_n, v_{n-1}, \dots, v_{k+1}, v_1$ is a Hamilton circuit in G . Conclude from this contradiction that Ore's Theorem holds.

9.6 Shortest-Path Problems

Introduction

Many problems can be modeled using graphs with weights assigned to their edges. As an illustration, consider how an airline system can be modeled. We set up the basic graph model by representing cities by vertices and flights by edges. Problems involving distances can be modeled by assigning distances between cities to the edges. Problems involving flight time can be modeled by assigning flight times to edges. Problems involving fares can be modeled by assigning fares to the edges. Figure 1 displays three different assignments of weights to the edges of a graph representing distances, flight times, and fares, respectively.

Graphs that have a number assigned to each edge are called **weighted graphs**. Weighted graphs are used to model computer networks. Communications costs (such as the monthly cost

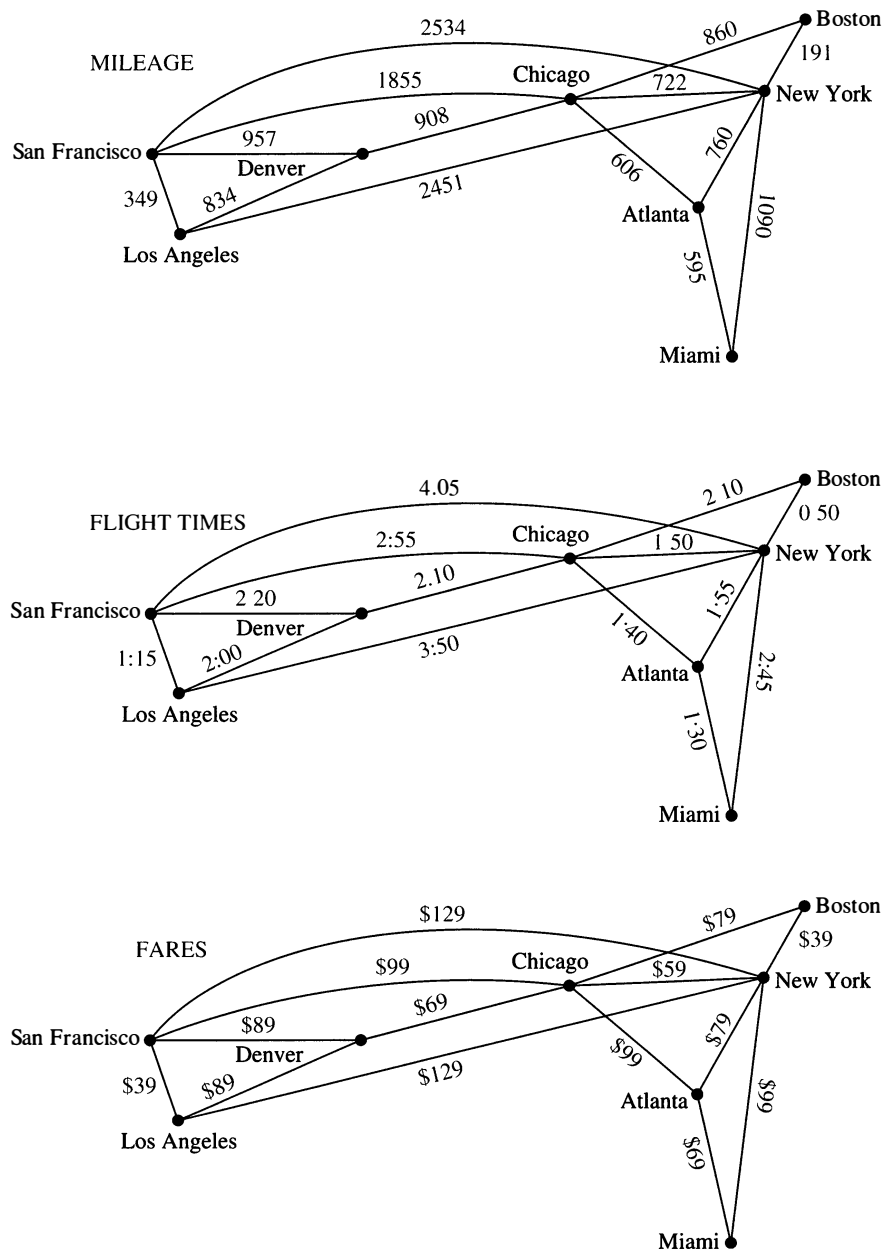


FIGURE 1 Weighted Graphs Modeling an Airline System.

of leasing a telephone line), the response times of the computers over these lines, or the distance between computers, can all be studied using weighted graphs. Figure 2 displays weighted graphs that represent three ways to assign weights to the edges of a graph of a computer network, corresponding to distance, response time, and cost.

Several types of problems involving weighted graphs arise frequently. Determining a path of least length between two vertices in a network is one such problem. To be more specific, let the **length** of a path in a weighted graph be the sum of the weights of the edges of this path. (The reader should note that this use of the term *length* is different from the use of *length* to denote the number of edges in a path in a graph without weights.) The question is: What is a shortest path, that is, a path of least length, between two given vertices? For instance, in the airline system represented by the weighted graph shown in Figure 1, what is a shortest path in air distance between Boston and Los Angeles? What combinations of flights has the smallest total

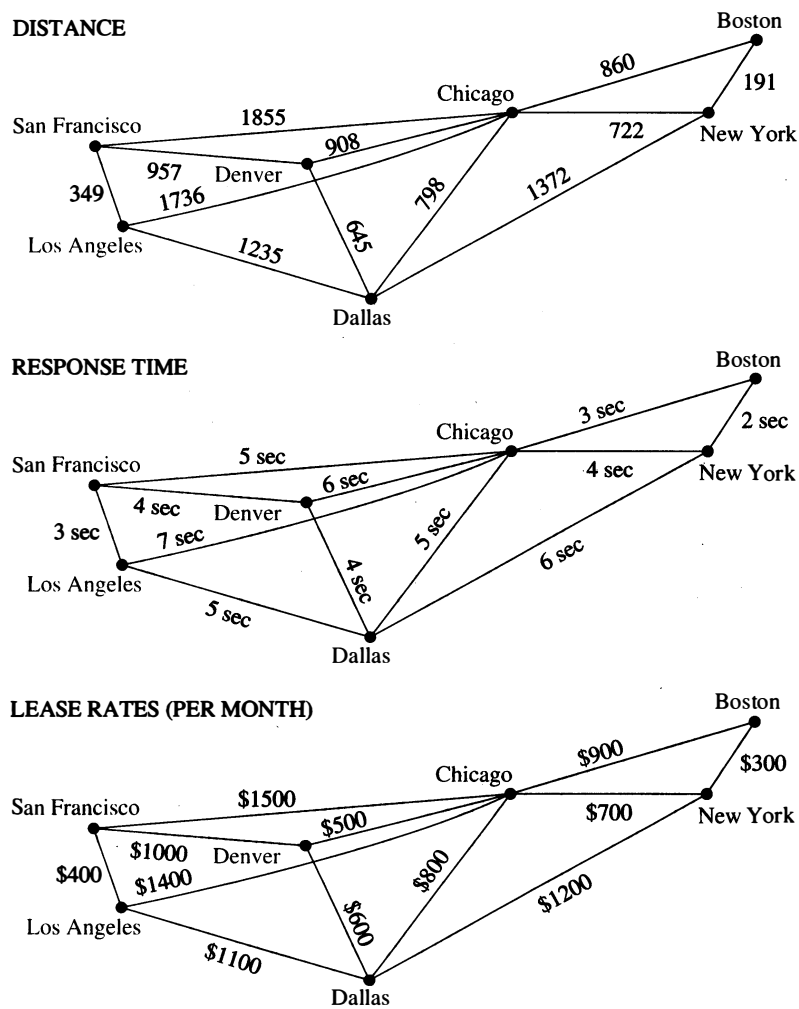


FIGURE 2 Weighted Graphs Modeling a Computer Network.

flight time (that is, total time in the air, not including time between flights) between Boston and Los Angeles? What is the cheapest fare between these two cities? In the computer network shown in Figure 2, what is a least expensive set of telephone lines needed to connect the computers in San Francisco with those in New York? Which set of telephone lines gives a fastest response time for communications between San Francisco and New York? Which set of lines has a shortest overall distance?

Another important problem involving weighted graphs asks for a circuit of shortest total length that visits every vertex of a complete graph exactly once. This is the famous *traveling salesman problem*, which asks for an order in which a salesman should visit each of the cities on his route exactly once so that he travels the minimum total distance. We will discuss the traveling salesman problem later in this section.

A Shortest-Path Algorithm



There are several different algorithms that find a shortest path between two vertices in a weighted graph. We will present an algorithm discovered by the Dutch mathematician Edsger Dijkstra in 1959. The version we will describe solves this problem in undirected weighted graphs where all the weights are positive. It is easy to adapt it to solve shortest-path problems in directed graphs.

Before giving a formal presentation of the algorithm, we will give a motivating example.

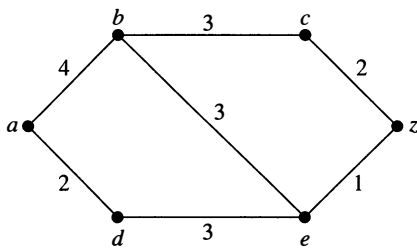


FIGURE 3 A Weighted Simple Graph.

EXAMPLE 1 What is the length of a shortest path between a and z in the weighted graph shown in Figure 3?

Solution: Although a shortest path is easily found by inspection, we will develop some ideas useful in understanding Dijkstra's algorithm. We will solve this problem by finding the length of a shortest path from a to successive vertices, until z is reached.

The only paths starting at a that contain no vertex other than a (until the terminal vertex is reached) are a, b and a, d . Because the lengths of a, b and a, d are 4 and 2, respectively, it follows that d is the closest vertex to a .

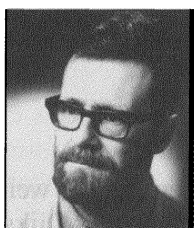
We can find the next closest vertex by looking at all paths that go through only a and d (until the terminal vertex is reached). The shortest such path to b is still a, b , with length 4, and the shortest such path to e is a, d, e , with length 5. Consequently, the next closest vertex to a is b .

To find the third closest vertex to a , we need to examine only paths that go through only a, d , and b (until the terminal vertex is reached). There is a path of length 7 to c , namely, a, b, c , and a path of length 6 to z , namely, a, d, e, z . Consequently, z is the next closest vertex to a , and the length of a shortest path to z is 6. ◀

Example 1 illustrates the general principles used in Dijkstra's algorithm. Note that a shortest path from a to z could have been found by inspection. However, inspection is impractical for both humans and computers for graphs with large numbers of edges.

We will now consider the general problem of finding the length of a shortest path between a and z in an undirected connected simple weighted graph. Dijkstra's algorithm proceeds by finding the length of a shortest path from a to a first vertex, the length of a shortest path from a to a second vertex, and so on, until the length of a shortest path from a to z is found.

The algorithm relies on a series of iterations. A distinguished set of vertices is constructed by adding one vertex at each iteration. A labeling procedure is carried out at each iteration. In this labeling procedure, a vertex w is labeled with the length of a shortest path from a to w that contains only vertices already in the distinguished set. The vertex added to the distinguished set is one with a minimal label among those vertices not already in the set.



EDSGER WYBE DIJKSTRA (1930–2002) Edsger Dijkstra, born in the Netherlands, began programming computers in the early 1950s while studying theoretical physics at the University of Leiden. In 1952, realizing that he was more interested in programming than in physics, he quickly completed the requirements for his physics degree and began his career as a programmer, even though programming was not a recognized profession. (In 1957, the authorities in Amsterdam refused to accept “programming” as his profession on his marriage license. However, they did accept “theoretical physicist” when he changed his entry to this.)

Dijkstra was one of the most forceful proponents of programming as a scientific discipline. He has made fundamental contributions to the areas of operating systems, including deadlock avoidance; programming languages, including the notion of structured programming; and algorithms. In 1972 Dijkstra received the Turing Award from the Association for Computing Machinery, one of the most prestigious awards in computer science. Dijkstra became a Burroughs Research Fellow in 1973, and in 1984 he was appointed to a chair in Computer Science at the University of Texas,

We now give the details of Dijkstra's algorithm. It begins by labeling a with 0 and the other vertices with ∞ . We use the notation $L_0(a) = 0$ and $L_0(v) = \infty$ for these labels before any iterations have taken place (the subscript 0 stands for the "0th" iteration). These labels are the lengths of shortest paths from a to the vertices, where the paths contain only the vertex a . (Because no path from a to a vertex different from a exists, ∞ is the length of a shortest path between a and this vertex.)

Dijkstra's algorithm proceeds by forming a distinguished set of vertices. Let S_k denote this set after k iterations of the labeling procedure. We begin with $S_0 = \emptyset$. The set S_k is formed from S_{k-1} by adding a vertex u not in S_{k-1} with the smallest label. Once u is added to S_k , we update the labels of all vertices not in S_k , so that $L_k(v)$, the label of the vertex v at the k th stage, is the length of a shortest path from a to v that contains vertices only in S_k (that is, vertices that were already in the distinguished set together with u).

Let v be a vertex not in S_k . To update the label of v , note that $L_k(v)$ is the length of a shortest path from a to v containing only vertices in S_k . The updating can be carried out efficiently when this observation is used: A shortest path from a to v containing only elements of S_k is either a shortest path from a to v that contains only elements of S_{k-1} (that is, the distinguished vertices not including u), or it is a shortest path from a to u at the $(k-1)$ st stage with the edge (u, v) added. In other words,

$$L_k(a, v) = \min\{L_{k-1}(a, v), L_{k-1}(a, u) + w(u, v)\}.$$

This procedure is iterated by successively, adding vertices to the distinguished set until z is added. When z is added to the distinguished set, its label is the length of a shortest path from a to z . Dijkstra's algorithm is given in Algorithm 1. Later we will give a proof that this algorithm is correct.

ALGORITHM 1 Dijkstra's Algorithm.

```

procedure Dijkstra( $G$ : weighted connected simple graph, with
    all weights positive)
  { $G$  has vertices  $a = v_0, v_1, \dots, v_n = z$  and weights  $w(v_i, v_j)$ 
    where  $w(v_i, v_j) = \infty$  if  $\{v_i, v_j\}$  is not an edge in  $G$ }
  for  $i := 1$  to  $n$ 
     $L(v_i) := \infty$ 
   $L(a) := 0$ 
   $S := \emptyset$ 
  {the labels are now initialized so that the label of  $a$  is 0 and all
    other labels are  $\infty$ , and  $S$  is the empty set}
  while  $z \notin S$ 
  begin
     $u :=$  a vertex not in  $S$  with  $L(u)$  minimal
     $S := S \cup \{u\}$ 
    for all vertices  $v$  not in  $S$ 
      if  $L(u) + w(u, v) < L(v)$  then  $L(v) := L(u) + w(u, v)$ 
      {this adds a vertex to  $S$  with minimal label and updates the
        labels of vertices not in  $S$ }
  end { $L(z)$  = length of a shortest path from  $a$  to  $z$ }
  
```

Example 2 illustrates how Dijkstra's algorithm works. Afterward, we will show that this algorithm always produces the length of a shortest path between two vertices in a weighted graph.

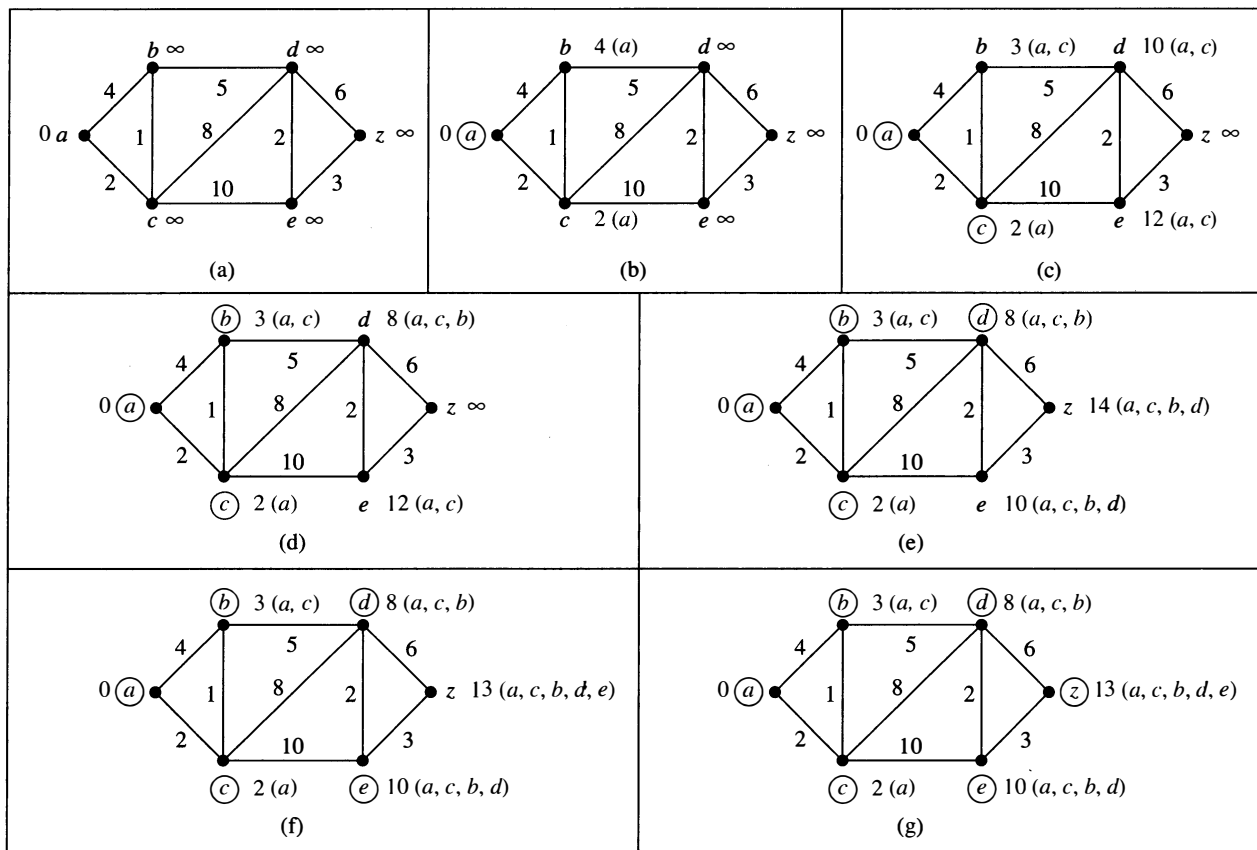


FIGURE 4 Using Dijkstra's Algorithm to Find a Shortest Path from a to z .

EXAMPLE 2 Use Dijkstra's algorithm to find the length of a shortest path between the vertices a and z in the weighted graph displayed in Figure 4(a).

Solution: The steps used by Dijkstra's algorithm to find a shortest path between a and z are shown in Figure 4. At each iteration of the algorithm the vertices of the set S_k are circled. A shortest path from a to each vertex containing only vertices in S_k is indicated for each iteration. The algorithm terminates when z is circled. We find that a shortest path from a to z is a, c, b, d, e, z , with length 13. ◀

Remark: In performing Dijkstra's algorithm it is sometimes more convenient to keep track of labels of vertices in each step using a table instead of redrawing the graph for each step.

Next, we use an inductive argument to show that Dijkstra's algorithm produces the length of a shortest path between two vertices a and z in an undirected connected weighted graph. Take as the induction hypothesis the following assertion: At the k th iteration

- (i) the label of every vertex v in S is the length of a shortest path from a to this vertex, and
- (ii) the label of every vertex not in S is the length of a shortest path from a to this vertex that contains only (besides the vertex itself) vertices in S .

When $k = 0$, before any iterations are carried out, $S = \emptyset$, so the length of a shortest path from a to a vertex other than a is ∞ . Hence, the basis case is true.

Assume that the inductive hypothesis holds for the k th iteration. Let v be the vertex added to S at the $(k + 1)$ st iteration, so v is a vertex not in S at the end of the k th iteration with the smallest label (in the case of ties, any vertex with smallest label may be used).

From the inductive hypothesis we see that the vertices in S before the $(k + 1)$ st iteration are labeled with the length of a shortest path from a . Also, v must be labeled with the length of a shortest path to it from a . If this were not the case, at the end of the k th iteration there would be a path of length less than $L_k(v)$ containing a vertex not in S [because $L_k(v)$ is the length of a shortest path from a to v containing only vertices in S after the k th iteration]. Let u be the first vertex not in S in such a path. There is a path with length less than $L_k(v)$ from a to u containing only vertices of S . This contradicts the choice of v . Hence, (i) holds at the end of the $(k + 1)$ st iteration.

Let u be a vertex not in S after $k + 1$ iterations. A shortest path from a to u containing only elements of S either contains v or it does not. If it does not contain v , then by the inductive hypothesis its length is $L_k(u)$. If it does contain v , then it must be made up of a path from a to v of shortest possible length containing elements of S other than v , followed by the edge from v to u . In this case, its length would be $L_k(v) + w(v, u)$. This shows that (ii) is true, because $L_{k+1}(u) = \min\{L_k(u), L_k(v) + w(v, u)\}$.

Theorem 1 has been proved.

THEOREM 1 Dijkstra's algorithm finds the length of a shortest path between two vertices in a connected simple undirected weighted graph.

We can now estimate the computational complexity of Dijkstra's algorithm (in terms of additions and comparisons). The algorithm uses no more than $n - 1$ iterations, because one vertex is added to the distinguished set at each iteration. We are done if we can estimate the number of operations used for each iteration. We can identify the vertex not in S_k with the smallest label using no more than $n - 1$ comparisons. Then we use an addition and a comparison to update the label of each vertex not in S_k . It follows that no more than $2(n - 1)$ operations are used at each iteration, because there are no more than $n - 1$ labels to update at each iteration. Because we use no more than $n - 1$ iterations, each using no more than $2(n - 1)$ operations, we have Theorem 2.

THEOREM 2 Dijkstra's algorithm uses $O(n^2)$ operations (additions and comparisons) to find the length of a shortest path between two vertices in a connected simple undirected weighted graph with n vertices.

The Traveling Salesman Problem



We now discuss an important problem involving weighted graphs. Consider the following problem: A traveling salesman wants to visit each of n cities exactly once and return to his starting point. For example, suppose that the salesman wants to visit Detroit, Toledo, Saginaw, Grand Rapids, and Kalamazoo (see Figure 5). In which order should he visit these cities to travel the minimum total distance? To solve this problem we can assume the salesman starts in Detroit (because this must be part of the circuit) and examine all possible ways for him to visit the other four cities and then return to Detroit (starting elsewhere will produce the same circuits). There are a total of 24 such circuits, but because we travel the same distance when we travel a circuit in reverse order, we need only consider 12 different circuits to find the minimum total distance he must travel. We list these 12 different circuits and the total distance traveled for each circuit. As can be seen from the list, the minimum total distance of 458 miles is traveled using the circuit Detroit–Toledo–Kalamazoo–Grand Rapids–Saginaw–Detroit (or its reverse).

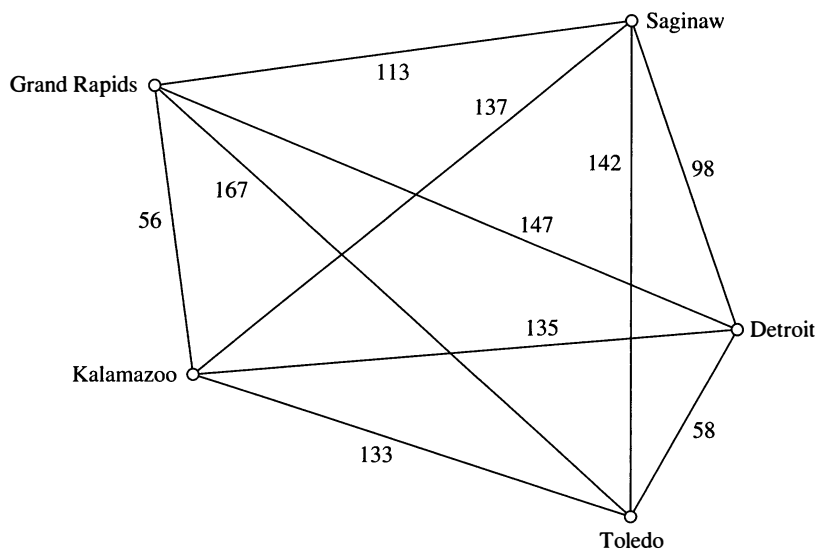


FIGURE 5 The Graph Showing the Distances between Five Cities.

<i>Route</i>	<i>Total Distance (miles)</i>
Detroit–Toledo–Grand Rapids–Saginaw–Kalamazoo–Detroit	610
Detroit–Toledo–Grand Rapids–Kalamazoo–Saginaw–Detroit	516
Detroit–Toledo–Kalamazoo–Saginaw–Grand Rapids–Detroit	588
Detroit–Toledo–Kalamazoo–Grand Rapids–Saginaw–Detroit	458
Detroit–Toledo–Saginaw–Kalamazoo–Grand Rapids–Detroit	540
Detroit–Toledo–Saginaw–Grand Rapids–Kalamazoo–Detroit	504
Detroit–Saginaw–Toledo–Grand Rapids–Kalamazoo–Detroit	598
Detroit–Saginaw–Toledo–Kalamazoo–Grand Rapids–Detroit	576
Detroit–Saginaw–Kalamazoo–Toledo–Grand Rapids–Detroit	682
Detroit–Saginaw–Grand Rapids–Toledo–Kalamazoo–Detroit	646
Detroit–Grand Rapids–Saginaw–Toledo–Kalamazoo–Detroit	670
Detroit–Grand Rapids–Toledo–Saginaw–Kalamazoo–Detroit	728

We just described an instance of the **traveling salesman problem**. The traveling salesman problem asks for the circuit of minimum total weight in a weighted, complete, undirected graph that visits each vertex exactly once and returns to its starting point. This is equivalent to asking for a Hamilton circuit with minimum total weight in the complete graph, because each vertex is visited exactly once in the circuit.

The most straightforward way to solve an instance of the traveling salesman problem is to examine all possible Hamilton circuits and select one of minimum total length. How many circuits do we have to examine to solve the problem if there are n vertices in the graph? Once a starting point is chosen, there are $(n - 1)!$ different Hamilton circuits to examine, because there are $n - 1$ choices for the second vertex, $n - 2$ choices for the third vertex, and so on. Because a Hamilton circuit can be traveled in reverse order, we need only examine $(n - 1)!/2$ circuits to find our answer. Note that $(n - 1)!/2$ grows extremely rapidly. Trying to solve a traveling salesman problem in this way when there are only a few dozen vertices is impractical. For example, with 25 vertices, a total of $24!/2$ (approximately 3.1×10^{23}) different Hamilton circuits would have to be considered. If it took just one nanosecond (10^{-9} second) to examine each Hamilton circuit, a total of approximately ten million years would be required to find a minimum-length Hamilton circuit in this graph by exhaustive search techniques.

Because the traveling salesman problem has both practical and theoretical importance, a great deal of effort has been devoted to devising efficient algorithms that solve it. However, no algorithm with polynomial worst-case time complexity is known for solving this problem. Furthermore, if a polynomial worst-case time complexity algorithm were discovered for the traveling salesman problem, many other difficult problems would also be solvable using polynomial worst-case time complexity algorithms (such as determining whether a proposition in n variables is a tautology, discussed in Chapter 1). This follows from the theory of NP-completeness. (For more information about this, consult [GaJo79].)

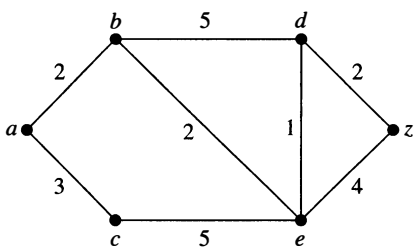
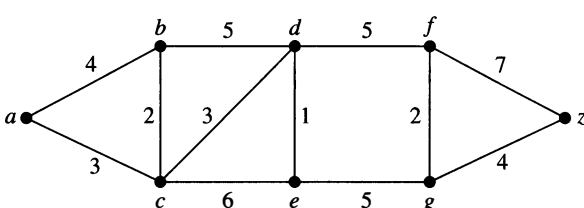
A practical approach to the traveling salesman problem when there are many vertices to visit is to use an **approximation algorithm**. These are algorithms that do not necessarily produce the exact solution to the problem but instead are guaranteed to produce a solution that is close to an exact solution. That is, they may produce a Hamilton circuit with total weight W' such that $W \leq W' \leq cW$, where W is the total length of an exact solution and c is a constant. For example, there is an algorithm with polynomial worst-case time complexity that works if the weighted graph satisfies the triangle inequality such that $c = 3/2$. For general weighted graphs for every positive real number k no algorithm is known that will always produce a solution at most k times a best solution. If such an algorithm existed, this would show that the class P would be the same as the class NP, perhaps the most famous open question about the complexity of algorithms (see Section 3.3).

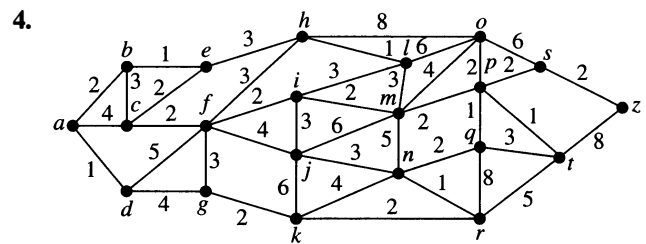
In practice, algorithms have been developed that can solve traveling salesman problems with as many as 1000 vertices within 2% of an exact solution using only a few minutes of computer time. For more information about the traveling salesman problem, including history, applications, and algorithms, see the chapter on this topic in *Applications of Discrete Mathematics* [MiRo91] also available on the website for this book.

Exercises

1. For each of these problems about a subway system, describe a weighted graph model that can be used to solve the problem.
 - a) What is the least amount of time required to travel between two stops?
 - b) What is the minimum distance that can be traveled to reach a stop from another stop?
 - c) What is the least fare required to travel between two stops if fares between stops are added to give the total fare?

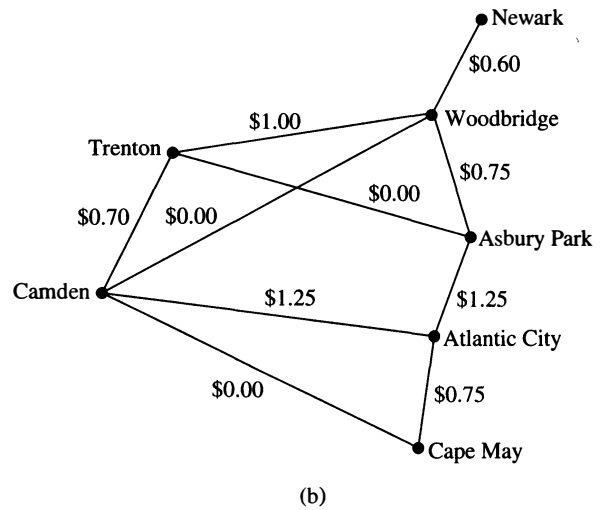
In Exercises 2–4 find the length of a shortest path between a and z in the given weighted graph.

2. 
3. 



4. Find a shortest path between a and z in each of the weighted graphs in Exercises 2–4.
5. Find the length of a shortest path between these pairs of vertices in the weighted graph in Exercise 3.
 - a) a and d
 - b) a and f
 - c) c and f
 - d) b and z
6. Find shortest paths in the weighted graph in Exercise 3 between the pairs of vertices in Exercise 6.
7. Find a shortest path (in mileage) between each of the following pairs of cities in the airline system shown in Figure 1.
 - a) New York and Los Angeles
 - b) Boston and San Francisco
 - c) Miami and Denver
 - d) Miami and Los Angeles

9. Find a combination of flights with the least total air time between the pairs of cities in Exercise 8, using the flight times shown in Figure 1.
10. Find a least expensive combination of flights connecting the pairs of cities in Exercise 8, using the fares shown in Figure 1.
11. Find a shortest route (in distance) between computer centers in each of these pairs of cities in the communications network shown in Figure 2.
 - a) Boston and Los Angeles
 - b) New York and San Francisco
 - c) Dallas and San Francisco
 - d) Denver and New York
12. Find a route with the shortest response time between the pairs of computer centers in Exercise 11 using the response times given in Figure 2.
13. Find a least expensive route, in monthly lease charges, between the pairs of computer centers in Exercise 11 using the lease charges given in Figure 2.
14. Explain how to find a path with the least number of edges between two vertices in an undirected graph by considering it as a shortest path problem in a weighted graph.
15. Extend Dijkstra's algorithm for finding the length of a shortest path between two vertices in a weighted simple connected graph so that the length of a shortest path between the vertex a and every other vertex of the graph is found.
16. Extend Dijkstra's algorithm for finding the length of a shortest path between two vertices in a weighted simple connected graph so that a shortest path between these vertices is constructed.
17. The weighted graphs in the figures here show some major roads in New Jersey. Part (a) shows the distances between cities on these roads; part (b) shows the tolls.

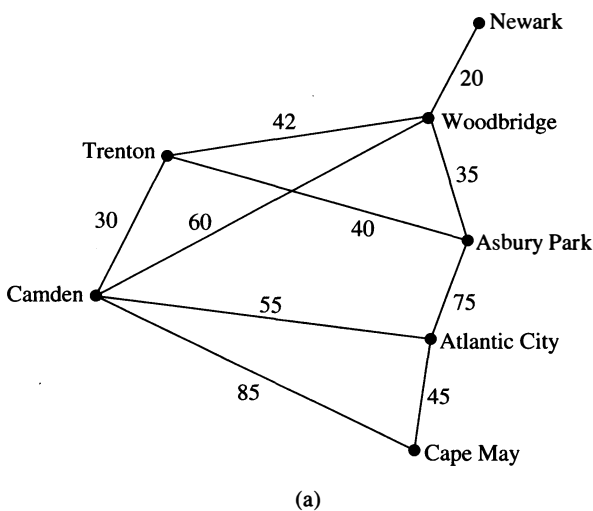


- a) Find a shortest route in distance between Newark and Camden, and between Newark and Cape May, using these roads.
 - b) Find a least expensive route in terms of total tolls using the roads in the graph between the pairs of cities in part (a) of this exercise.
18. Is a shortest path between two vertices in a weighted graph unique if the weights of edges are distinct?
 19. What are some applications where it is necessary to find the length of a longest simple path between two vertices in a weighted graph?
 20. What is the length of a longest simple path in the weighted graph in Figure 4 between a and z ? Between c and z ?



Floyd's algorithm, displayed as Algorithm 2, can be used to find the length of a shortest path between all pairs of vertices in a weighted connected simple graph. However, this algorithm cannot be used to construct shortest paths. (We assign an infinite weight to any pair of vertices not connected by an edge in the graph.)

21. Use Floyd's algorithm to find the distance between all pairs of vertices in the weighted graph in Figure 4(a).
- *22. Prove that Floyd's algorithm determines the shortest distance between all pairs of vertices in a weighted simple graph.
- *23. Give a big- O estimate of the number of operations (comparisons and additions) used by Floyd's algorithm to determine the shortest distance between every pair of vertices in a weighted simple graph with n vertices.
- *24. Show that Dijkstra's algorithm may not work if edges can have negative weights.

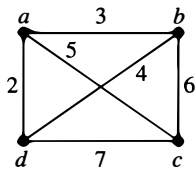


ALGORITHM 2 Floyd's Algorithm.

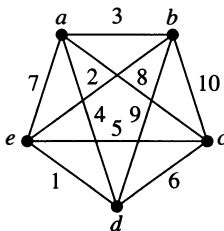
```

procedure Floyd( $G$ : weighted simple graph)
{ $G$  has vertices  $v_1, v_2, \dots, v_n$  and weights  $w(v_i, v_j)$ 
with  $w(v_i, v_j) = \infty$  if  $(v_i, v_j)$  is not an edge}
  for  $i := 1$  to  $n$ 
    for  $j := 1$  to  $n$ 
       $d(v_i, v_j) := w(v_i, v_j)$ 
  for  $i := 1$  to  $n$ 
    for  $j := 1$  to  $n$ 
      for  $k := 1$  to  $n$ 
        if  $d(v_j, v_i) + d(v_i, v_k) < d(v_j, v_k)$ 
          then  $d(v_j, v_k) :=$ 
             $d(v_j, v_i) + d(v_i, v_k)$ 
{ $d(v_i, v_j)$  is the length of a shortest path between  $v_i$ 
and  $v_j$ }
    
```

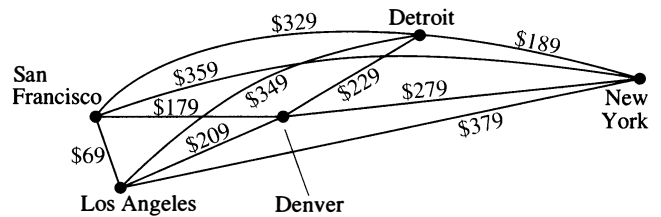
25. Solve the traveling salesman problem for this graph by finding the total weight of all Hamilton circuits and determining a circuit with minimum total weight.



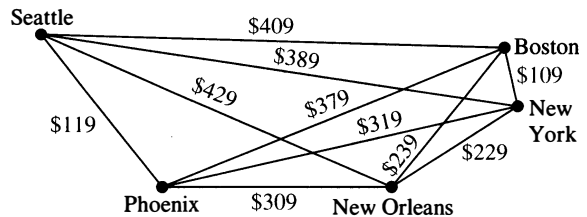
26. Solve the traveling salesman problem for this graph by finding the total weight of all Hamilton circuits and determining a circuit with minimum total weight.



27. Find a route with the least total airfare that visits each of the cities in this graph, where the weight on an edge is the least price available for a flight between the two cities.



28. Find a route with the least total airfare that visits each of the cities in this graph, where the weight on an edge is the least price available for a flight between the two cities.



29. Construct a weighted undirected graph such that the total weight of a circuit that visits every vertex at least once is minimized for a circuit that visits some vertices more than once. [Hint: There are examples with three vertices.]
30. Show that the problem of finding a circuit of minimum total weight that visits every vertex of a weighted graph at least once can be reduced to the problem of finding a circuit of minimum total weight that visits each vertex of a weighted graph exactly once. Do so by constructing a new weighted graph with the same vertices and edges as the original graph but whose weight of the edge connecting the vertices u and v is equal to the minimum total weight of a path from u to v in the original graph.

9.7 Planar Graphs

Introduction



Consider the problem of joining three houses to each of three separate utilities, as shown in Figure 1. Is it possible to join these houses and utilities so that none of the connections cross? This problem can be modeled using the complete bipartite graph $K_{3,3}$. The original question can be rephrased as: Can $K_{3,3}$ be drawn in the plane so that no two of its edges cross?

In this section we will study the question of whether a graph can be drawn in the plane without edges crossing. In particular, we will answer the houses-and-utilities problem.

There are always many ways to represent a graph. When is it possible to find at least one way to represent this graph in a plane without any edges crossing?

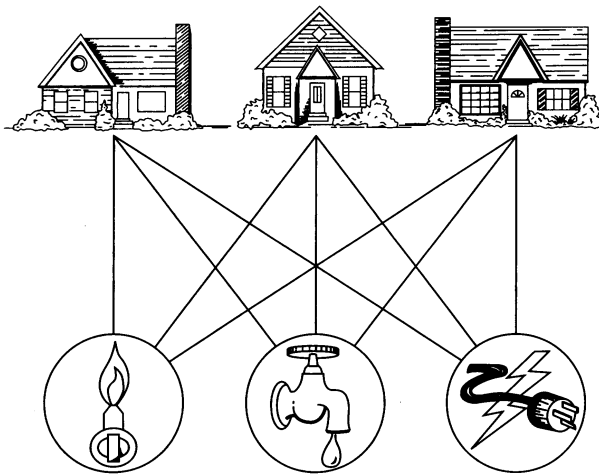


FIGURE 1 Three Houses and Three Utilities.

DEFINITION 1 A graph is called *planar* if it can be drawn in the plane without any edges crossing (where a crossing of edges is the intersection of the lines or arcs representing them at a point other than their common endpoint). Such a drawing is called a *planar representation* of the graph.

A graph may be planar even if it is usually drawn with crossings, because it may be possible to draw it in a different way without crossings.

EXAMPLE 1 Is K_4 (shown in Figure 2 with two edges crossing) planar?

Solution: K_4 is planar because it can be drawn without crossings, as shown in Figure 3. ◀

EXAMPLE 2 Is Q_3 , shown in Figure 4, planar?

Solution: Q_3 is planar, because it can be drawn without any edges crossing, as shown in Figure 5. ◀

We can show that a graph is planar by displaying a planar representation. It is harder to show that a graph is nonplanar. We will give an example to show how this can be done in an ad hoc fashion. Later we will develop some general results that can be used to do this.

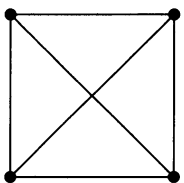


FIGURE 2 The Graph K_4 .

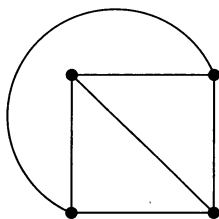


FIGURE 3 K_4 Drawn with No Crossings.

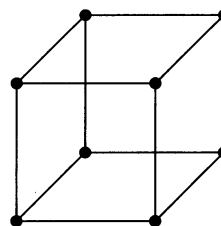


FIGURE 4 The Graph Q_3 .

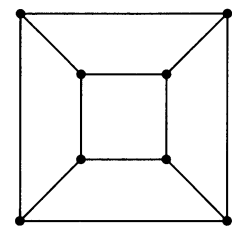
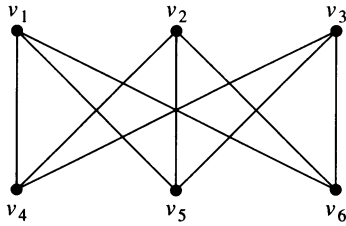
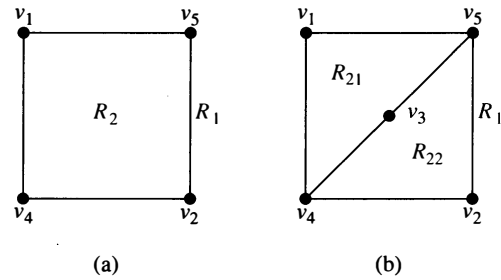


FIGURE 5 A Planar Representation of Q_3 .

FIGURE 6 The Graph $K_{3,3}$.FIGURE 7 Showing that $K_{3,3}$ Is Nonplanar.

EXAMPLE 3 Is $K_{3,3}$, shown in Figure 6, planar?

Solution: Any attempt to draw $K_{3,3}$ in the plane with no edges crossing is doomed. We now show why. In any planar representation of $K_{3,3}$, the vertices v_1 and v_2 must be connected to both v_4 and v_5 . These four edges form a closed curve that splits the plane into two regions, R_1 and R_2 , as shown in Figure 7(a). The vertex v_3 is in either R_1 or R_2 . When v_3 is in R_2 , the inside of the closed curve, the edges between v_3 and v_4 and between v_3 and v_5 separate R_2 into two subregions, R_{21} and R_{22} , as shown in Figure 7(b).

Next, note that there is no way to place the final vertex v_6 without forcing a crossing. For if v_6 is in R_1 , then the edge between v_6 and v_3 cannot be drawn without a crossing. If v_6 is in R_{21} , then the edge between v_2 and v_6 cannot be drawn without a crossing. If v_6 is in R_{22} , then the edge between v_1 and v_6 cannot be drawn without a crossing.

A similar argument can be used when v_3 is in R_1 . The completion of this argument is left for the reader (see Exercise 10 at the end of this section). It follows that $K_{3,3}$ is not planar. ◀

Example 3 solves the utilities-and-houses problem that was described at the beginning of this section. The three houses and three utilities cannot be connected in the plane without a crossing. A similar argument can be used to show that K_5 is nonplanar. (See Exercise 11 at the end of this section.)

Planarity of graphs plays an important role in the design of electronic circuits. We can model a circuit with a graph by representing components of the circuit by vertices and connections between them by edges. We can print a circuit on a single board with no connections crossing if the graph representing the circuit is planar. When this graph is not planar, we must turn to more expensive options. For example, we can partition the vertices in the graph representing the circuit into planar subgraphs. We then construct the circuit using multiple layers. (See the preamble to Exercise 30 to learn about the thickness of a graph.) We can construct the circuit using insulated wires whenever connections cross. In this case, drawing the graph with the fewest possible crossings is important. (See the preamble to Exercise 26 to learn about the crossing number of a graph.)

Euler's Formula

A planar representation of a graph splits the plane into **regions**, including an unbounded region. For instance, the planar representation of the graph shown in Figure 8 splits the plane into six regions. These are labeled in the figure. Euler showed that all planar representations of a graph split the plane into the same number of regions. He accomplished this by finding a relationship among the number of regions, the number of vertices, and the number of edges of a planar graph.

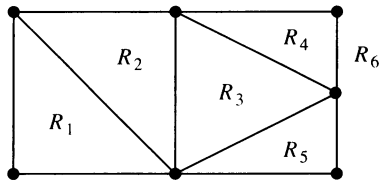


FIGURE 8 The Regions of the Planar Representation of a Graph.

THEOREM 1 EULER'S FORMULA Let G be a connected planar simple graph with e edges and v vertices. Let r be the number of regions in a planar representation of G . Then $r = e - v + 2$.

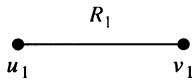


FIGURE 9 The Basis Case of the Proof of Euler's Formula.

Proof: First, we specify a planar representation of G . We will prove the theorem by constructing a sequence of subgraphs $G_1, G_2, \dots, G_e = G$, successively adding an edge at each stage. This is done using the following inductive definition. Arbitrarily pick one edge of G to obtain G_1 . Obtain G_n from G_{n-1} by arbitrarily adding an edge that is incident with a vertex already in G_{n-1} , adding the other vertex incident with this edge if it is not already in G_{n-1} . This construction is possible because G is connected. G is obtained after e edges are added. Let r_n, e_n , and v_n represent the number of regions, edges, and vertices of the planar representation of G_n induced by the planar representation of G , respectively.

The proof will now proceed by induction. The relationship $r_1 = e_1 - v_1 + 2$ is true for G_1 , because $e_1 = 1, v_1 = 2$, and $r_1 = 1$. This is shown in Figure 9.

Now assume that $r_n = e_n - v_n + 2$. Let $\{a_{n+1}, b_{n+1}\}$ be the edge that is added to G_n to obtain G_{n+1} . There are two possibilities to consider. In the first case, both a_{n+1} and b_{n+1} are already in G_n . These two vertices must be on the boundary of a common region R , or else it would be impossible to add the edge $\{a_{n+1}, b_{n+1}\}$ to G_n without two edges crossing (and G_{n+1} is planar). The addition of this new edge splits R into two regions. Consequently, in this case, $r_{n+1} = r_n + 1, e_{n+1} = e_n + 1$, and $v_{n+1} = v_n$. Thus, each side of the formula relating the number of regions, edges, and vertices increases by exactly one, so this formula is still true. In other words, $r_{n+1} = e_{n+1} - v_{n+1} + 2$. This case is illustrated in Figure 10(a).

In the second case, one of the two vertices of the new edge is not already in G_n . Suppose that a_{n+1} is in G_n but that b_{n+1} is not. Adding this new edge does not produce any new regions, because b_{n+1} must be in a region that has a_{n+1} on its boundary. Consequently, $r_{n+1} = r_n$. Moreover, $e_{n+1} = e_n + 1$ and $v_{n+1} = v_n + 1$. Each side of the formula relating the number

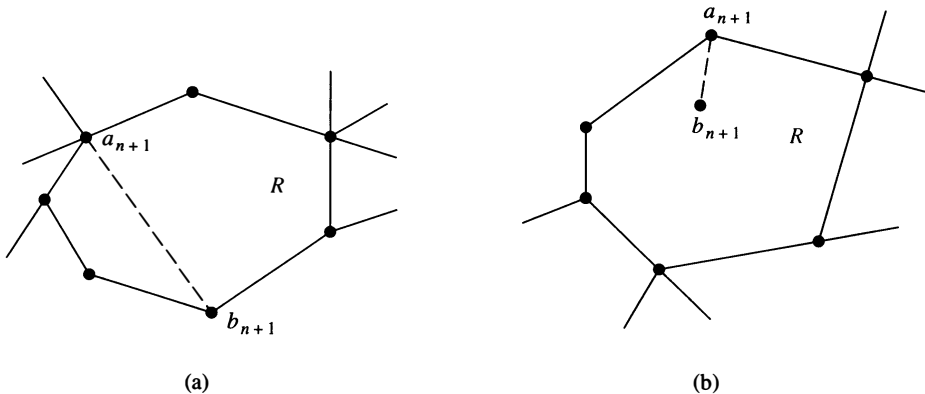


FIGURE 10 Adding an Edge to G_n to Produce G_{n+1} .

of regions, edges, and vertices remains the same, so the formula is still true. In other words, $r_{n+1} = e_{n+1} - v_{n+1} + 2$. This case is illustrated in Figure 10(b).

We have completed the induction argument. Hence, $r_n = e_n - v_n + 2$ for all n . Because the original graph is the graph G_e , obtained after e edges have been added, the theorem is true. ◀

Euler's formula is illustrated in Example 4.

EXAMPLE 4 Suppose that a connected planar simple graph has 20 vertices, each of degree 3. Into how many regions does a representation of this planar graph split the plane?

Solution: This graph has 20 vertices, each of degree 3, so $v = 20$. Because the sum of the degrees of the vertices, $3v = 3 \cdot 20 = 60$, is equal to twice the number of edges, $2e$, we have $2e = 60$, or $e = 30$. Consequently, from Euler's formula, the number of regions is

$$r = e - v + 2 = 30 - 20 + 2 = 12. \quad \blacktriangleleft$$

Euler's formula can be used to establish some inequalities that must be satisfied by planar graphs. One such inequality is given in Corollary 1.

COROLLARY 1 If G is a connected planar simple graph with e edges and v vertices, where $v \geq 3$, then $e \leq 3v - 6$.

Before we prove Corollary 1 we will use it to prove the following useful result.

COROLLARY 2 If G is a connected planar simple graph, then G has a vertex of degree not exceeding five.

Proof: If G has one or two vertices, the result is true. If G has at least three vertices, by Corollary 1 we know that $e \leq 3v - 6$, so $2e \leq 6v - 12$. If the degree of every vertex were at least six, then because $2e = \sum_{v \in V} \deg(v)$ (by the Handshaking Theorem), we would have $2e \geq 6v$. But this contradicts the inequality $2e \leq 6v - 12$. It follows that there must be a vertex with degree no greater than five. ◀

The proof of Corollary 1 is based on the concept of the **degree** of a region, which is defined to be the number of edges on the boundary of this region. When an edge occurs twice on the boundary (so that it is traced out twice when the boundary is traced out), it contributes two to the degree. The degrees of the regions of the graph shown in Figure 11 are displayed in the figure.

The proof of Corollary 1 can now be given.

Proof: A connected planar simple graph drawn in the plane divides the plane into regions, say r of them. The degree of each region is at least three. (Because the graphs discussed here are simple graphs, no multiple edges that could produce regions of degree two, or loops that could produce regions of degree one, are permitted.) In particular, note that the degree of the unbounded region is at least three because there are at least three vertices in the graph.

Note that the sum of the degrees of the regions is exactly twice the number of edges in the graph, because each edge occurs on the boundary of a region exactly twice (either in two

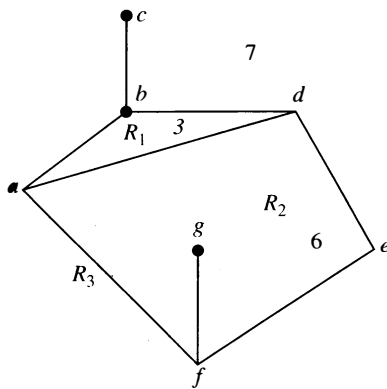


FIGURE 11 The Degrees of Regions.

different regions, or twice in the same region). Because each region has degree greater than or equal to three, it follows that

$$2e = \sum_{\text{all regions } R} \deg(R) \geq 3r.$$

Hence,

$$(2/3)e \geq r.$$

Using $r = e - v + 2$ (Euler's formula), we obtain

$$e - v + 2 \leq (2/3)e.$$

It follows that $e/3 \leq v - 2$. This shows that $e \leq 3v - 6$. ◁

This corollary can be used to demonstrate that K_5 is nonplanar.

EXAMPLE 5 Show that K_5 is nonplanar using Corollary 1.

Solution: The graph K_5 has five vertices and 10 edges. However, the inequality $e \leq 3v - 6$ is not satisfied for this graph because $e = 10$ and $3v - 6 = 9$. Therefore, K_5 is not planar. ◀

It was previously shown that $K_{3,3}$ is not planar. Note, however, that this graph has six vertices and nine edges. This means that the inequality $e = 9 \leq 12 = 3 \cdot 6 - 6$ is satisfied. Consequently, the fact that the inequality $e \leq 3v - 6$ is satisfied does *not* imply that a graph is planar. However, the following corollary of Theorem 1 can be used to show that $K_{3,3}$ is nonplanar.

COROLLARY 3 If a connected planar simple graph has e edges and v vertices with $v \geq 3$ and no circuits of length three, then $e \leq 2v - 4$.

The proof of Corollary 3 is similar to that of Corollary 1, except that in this case the fact that there are no circuits of length three implies that the degree of a region must be at least four. The details of this proof are left for the reader (see Exercise 15 at the end of this section).

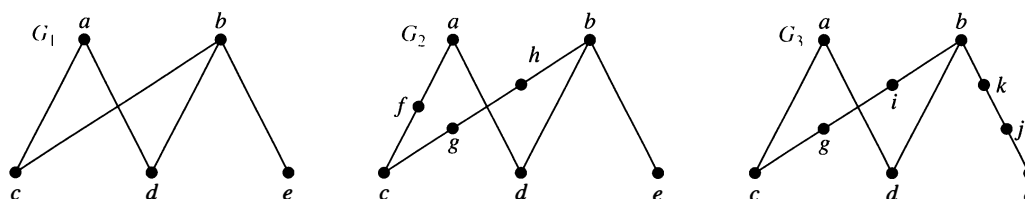


FIGURE 12 Homeomorphic Graphs.

EXAMPLE 6 Use Corollary 3 to show that $K_{3,3}$ is nonplanar.

Solution: Because $K_{3,3}$ has no circuits of length three (this is easy to see because it is bipartite), Corollary 3 can be used. $K_{3,3}$ has six vertices and nine edges. Because $e = 9$ and $2v - 4 = 8$, Corollary 3 shows that $K_{3,3}$ is nonplanar. \blacktriangleleft

Kuratowski's Theorem

We have seen that $K_{3,3}$ and K_5 are not planar. Clearly, a graph is not planar if it contains either of these two graphs as a subgraph. Surprisingly, all nonplanar graphs must contain a subgraph that can be obtained from $K_{3,3}$ or K_5 using certain permitted operations.

If a graph is planar, so will be any graph obtained by removing an edge $\{u, v\}$ and adding a new vertex w together with edges $\{u, w\}$ and $\{w, v\}$. Such an operation is called an **elementary subdivision**. The graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are called **homeomorphic** if they can be obtained from the same graph by a sequence of elementary subdivisions.

EXAMPLE 7 Show that the graphs G_1 , G_2 , and G_3 displayed in Figure 12 are all homeomorphic.

Solution: These three graphs are homeomorphic because all three can be obtained from G_1 by elementary subdivisions. G_1 can be obtained from itself by an empty sequence of elementary subdivisions. To obtain G_2 from G_1 we can use this sequence of elementary subdivisions: (i) remove the edge $\{a, c\}$, add the vertex f , and add the edges $\{a, f\}$ and $\{f, c\}$; (ii) remove the edge $\{b, c\}$, add the vertex g , and add the edges $\{b, g\}$ and $\{g, c\}$; and (iii) remove the edge $\{b, g\}$, add the vertex h , and add the edges $\{g, h\}$ and $\{b, h\}$. We leave it to the reader to determine the sequence of elementary subdivisions needed to obtain G_3 from G_1 . \blacktriangleleft

The Polish mathematician Kazimierz Kuratowski established Theorem 2 in 1930, which characterizes planar graphs using the concept of graph homeomorphism.



KAZIMIERZ KURATOWSKI (1896–1980) Kazimierz Kuratowski, the son of a famous Warsaw lawyer, attended secondary school in Warsaw. He studied in Glasgow, Scotland, from 1913 to 1914 but could not return there after the outbreak of World War I. In 1915 he entered Warsaw University, where he was active in the Polish patriotic student movement. He published his first paper in 1919 and received his Ph.D. in 1921. He was an active member of the group known as the Warsaw School of Mathematics, working in the areas of the foundations of set theory and topology. He was appointed associate professor at the Lwów Polytechnical University, where he stayed for seven years, collaborating with the important Polish mathematicians Banach and Ulam. In 1930, while at Lwów, Kuratowski completed his work characterizing planar graphs.

In 1934 he returned to Warsaw University as a full professor. Until the start of World War II, he was active in research and teaching. During the war, because of the persecution of educated Poles, Kuratowski went into hiding under an assumed name and taught at the clandestine Warsaw University. After the war he helped revive Polish mathematics, serving as director of the Polish National Mathematics Institute. He wrote over 180 papers and three widely used textbooks.

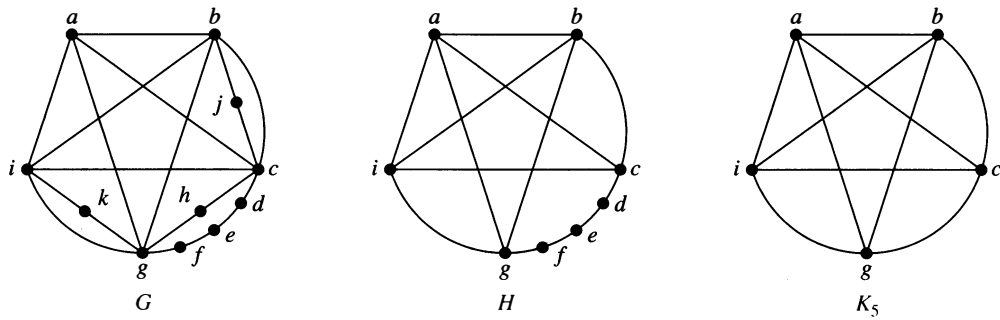


FIGURE 13 The Undirected Graph G , a Subgraph H Homeomorphic to K_5 , and K_5 .

THEOREM 2 A graph is nonplanar if and only if it contains a subgraph homeomorphic to $K_{3,3}$ or K_5 .

It is clear that a graph containing a subgraph homeomorphic to $K_{3,3}$ or K_5 is nonplanar. However, the proof of the converse, namely that every nonplanar graph contains a subgraph homeomorphic to $K_{3,3}$ or K_5 , is complicated and will not be given here. Examples 8 and 9 illustrate how Kuratowski's Theorem is used.

EXAMPLE 8 Determine whether the graph G shown in Figure 13 is planar.



Solution: G has a subgraph H homeomorphic to K_5 . H is obtained by deleting h , j , and k and all edges incident with these vertices. H is homeomorphic to K_5 because it can be obtained from K_5 (with vertices a , b , c , g , and i) by a sequence of elementary subdivisions, adding the vertices d , e , and f . (The reader should construct such a sequence of elementary subdivisions.) Hence, G is nonplanar. ◀

EXAMPLE 9 Is the Petersen graph, shown in Figure 14(a), planar? (The Danish mathematician Julius Petersen studied this graph in 1891; it is often used to illustrate various theoretical properties of graphs.)

Solution: The subgraph H of the Petersen graph obtained by deleting b and the three edges that have b as an endpoint, shown in Figure 14(b), is homeomorphic to $K_{3,3}$, with vertex sets

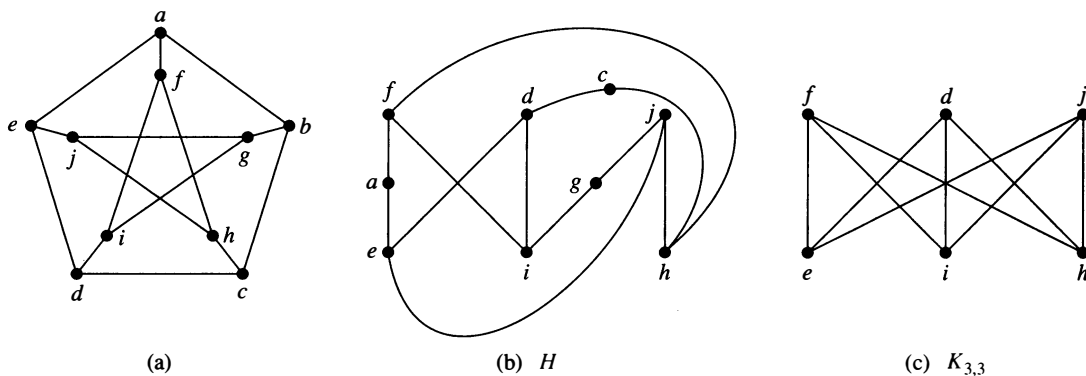


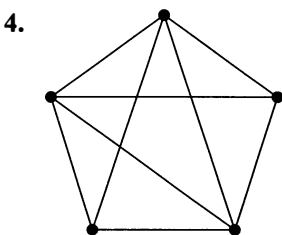
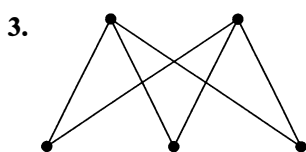
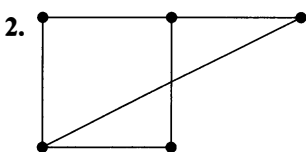
FIGURE 14 (a) The Petersen Graph, (b) a Subgraph H Homeomorphic to $K_{3,3}$, and (c) $K_{3,3}$.

$\{f, d, j\}$ and $\{e, i, h\}$, because it can be obtained by a sequence of elementary subdivisions, deleting $\{d, h\}$ and adding $\{c, h\}$ and $\{c, d\}$, deleting $\{e, f\}$ and adding $\{a, e\}$ and $\{a, f\}$, and deleting $\{i, j\}$ and adding $\{g, i\}$ and $\{g, j\}$. Hence, the Petersen graph is not planar. ◀

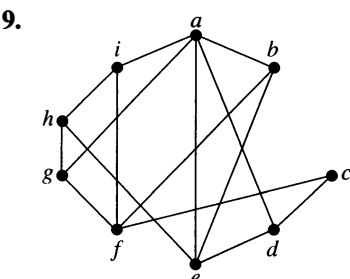
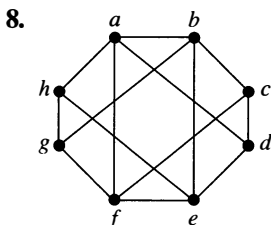
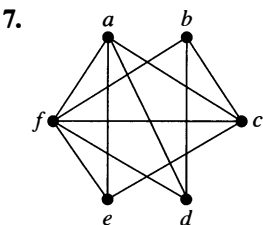
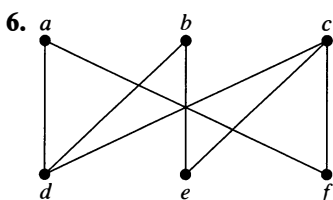
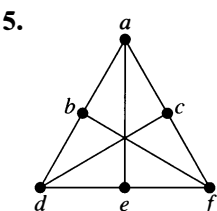
Exercises

1. Can five houses be connected to two utilities without connections crossing?

In Exercises 2–4 draw the given planar graph without any crossings.

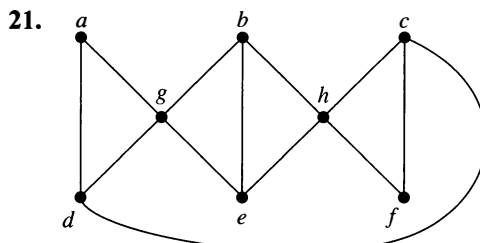
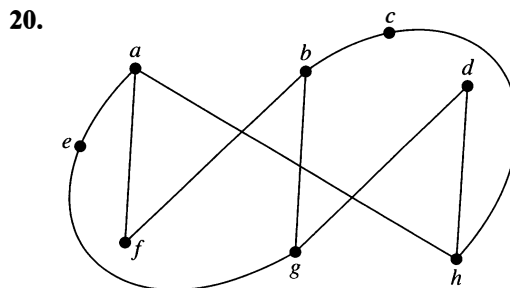


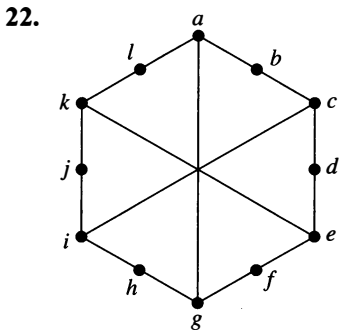
In Exercises 5–9 determine whether the given graph is planar. If so, draw it so that no edges cross.



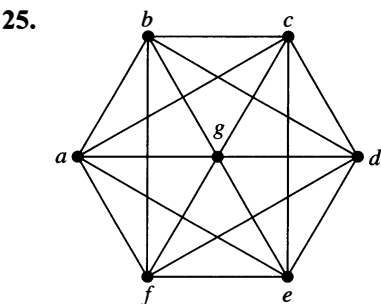
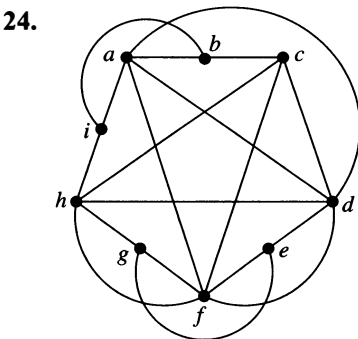
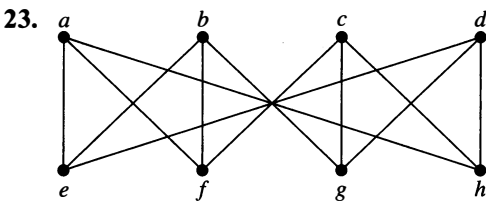
- 10. Complete the argument in Example 3.
- 11. Show that K_5 is nonplanar using an argument similar to that given in Example 3.
- 12. Suppose that a connected planar graph has eight vertices, each of degree three. Into how many regions is the plane divided by a planar representation of this graph?
- 13. Suppose that a connected planar graph has six vertices, each of degree four. Into how many regions is the plane divided by a planar representation of this graph?
- 14. Suppose that a connected planar graph has 30 edges. If a planar representation of this graph divides the plane into 20 regions, how many vertices does this graph have?
- 15. Prove Corollary 3.
- 16. Suppose that a connected bipartite planar simple graph has e edges and v vertices. Show that $e \leq 2v - 4$ if $v \geq 3$.
- *17. Suppose that a connected planar simple graph with e edges and v vertices contains no simple circuits of length 4 or less. Show that $e \leq (5/3)v - (10/3)$ if $v \geq 4$.
- 18. Suppose that a planar graph has k connected components, e edges, and v vertices. Also suppose that the plane is divided into r regions by a planar representation of the graph. Find a formula for r in terms of e , v , and k .
- 19. Which of these nonplanar graphs have the property that the removal of any vertex and all edges incident with that vertex produces a planar graph?
 - a) K_5 b) K_6 c) $K_{3,3}$ d) $K_{3,4}$


In Exercises 20–22 determine whether the given graph is homeomorphic to $K_{3,3}$.





In Exercises 23–25 use Kuratowski’s Theorem to determine whether the given graph is planar.



 The **crossing number** of a simple graph is the minimum number of crossings that can occur when this graph is drawn in the plane where no three arcs representing edges are permitted to cross at the same point.

26. Show that $K_{3,3}$ has 1 as its crossing number.

**27. Find the crossing numbers of each of these nonplanar graphs.

- a) K_5 b) K_6 c) K_7
- d) $K_{3,4}$ e) $K_{4,4}$ f) $K_{5,5}$

*28. Find the crossing number of the Petersen graph.

*29. Show that if m and n are even positive integers, the crossing number of $K_{m,n}$ is less than or equal to $mn(m - 2)(n - 2)/16$. [Hint: Place m vertices along the x -axis so that they are equally spaced and symmetric about the origin and place n vertices along the y -axis so that they are equally spaced and symmetric about the origin. Now connect each of the m vertices on the x -axis to each of the vertices on the y -axis and count the crossings.]

The **thickness** of a simple graph G is the smallest number of planar subgraphs of G that have G as their union.

30. Show that $K_{3,3}$ has 2 as its thickness.

*31. Find the thickness of the graphs in Exercise 27.

32. Show that if G is a connected simple graph with v vertices and e edges, then the thickness of G is at least $\lceil e/(3v - 6) \rceil$.

*33. Use Exercise 32 to show that the thickness of K_n is at least $\lceil (n + 7)/6 \rceil$ whenever n is a positive integer.

34. Show that if G is a connected simple graph with v vertices and e edges and no circuits of length three, then the thickness of G is at least $\lceil e/(2v - 4) \rceil$.

35. Use Exercise 34 to show that the thickness of $K_{m,n}$ is at least $\lceil mn/(2m + 2n - 4) \rceil$ whenever m and n are positive integers.

*36. Draw K_5 on the surface of a torus (a doughnut-shaped solid) so that no edges cross.

*37. Draw $K_{3,3}$ on the surface of a torus so that no edges cross.

9.8 Graph Coloring

Introduction



Problems related to the coloring of maps of regions, such as maps of parts of the world, have generated many results in graph theory. When a map* is colored, two regions with a common border are customarily assigned different colors. One way to ensure that two adjacent regions never have the same color is to use a different color for each region. However, this is inefficient,

*We will assume that all regions in a map are connected. This eliminates any problems presented by such geographical entities

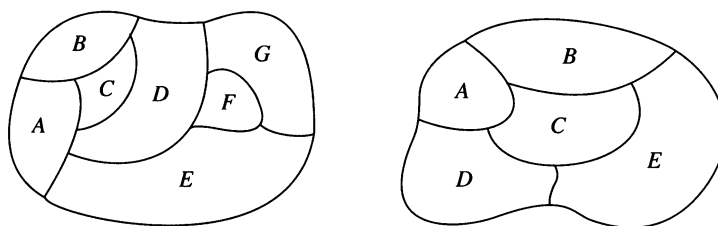


FIGURE 1 Two Maps.

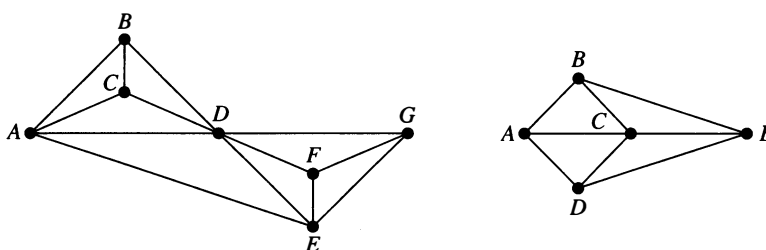


FIGURE 2 Dual Graphs of the Maps in Figure 1.

and on maps with many regions it would be hard to distinguish similar colors. Instead, a small number of colors should be used whenever possible. Consider the problem of determining the least number of colors that can be used to color a map so that adjacent regions never have the same color. For instance, for the map shown on the left in Figure 1, four colors suffice, but three colors are not enough. (The reader should check this.) In the map on the right in Figure 1, three colors are sufficient (but two are not).

Each map in the plane can be represented by a graph. To set up this correspondence, each region of the map is represented by a vertex. Edges connect two vertices if the regions represented by these vertices have a common border. Two regions that touch at only one point are not considered adjacent. The resulting graph is called the **dual graph** of the map. By the way in which dual graphs of maps are constructed, it is clear that any map in the plane has a planar dual graph. Figure 2 displays the dual graphs that correspond to the maps shown in Figure 1.

The problem of coloring the regions of a map is equivalent to the problem of coloring the vertices of the dual graph so that no two adjacent vertices in this graph have the same color. We now define a graph coloring.

DEFINITION 1 A *coloring* of a simple graph is the assignment of a color to each vertex of the graph so that no two adjacent vertices are assigned the same color.

A graph can be colored by assigning a different color to each of its vertices. However, for most graphs a coloring can be found that uses fewer colors than the number of vertices in the graph. What is the least number of colors necessary?

DEFINITION 2 The *chromatic number* of a graph is the least number of colors needed for a coloring of this graph. The chromatic number of a graph G is denoted by $\chi(G)$. (Here χ is the Greek letter *chi*.)

Note that asking for the chromatic number of a planar graph is the same as asking for the minimum number of colors required to color a planar map so that no two adjacent regions are assigned the same color. This question has been studied for more than 100 years. The answer is provided by one of the most famous theorems in mathematics.

THEOREM 1 THE FOUR COLOR THEOREM The chromatic number of a planar graph is no greater than four.



The Four Color Theorem was originally posed as a conjecture in the 1850s. It was finally proved by the American mathematicians Kenneth Appel and Wolfgang Haken in 1976. Prior to 1976, many incorrect proofs were published, often with hard-to-find errors. In addition, many futile attempts were made to construct counterexamples by drawing maps that require more than four colors. (Proving the Five Color Theorem is not that difficult; see Exercise 36.)

Perhaps the most notorious fallacious proof in all of mathematics is the incorrect proof of the Four Color Theorem published in 1879 by a London barrister and amateur mathematician, Alfred Kempe. Mathematicians accepted his proof as correct until 1890, when Percy Heawood found an error that made Kempe's argument incomplete. However, Kempe's line of reasoning turned out to be the basis of the successful proof given by Appel and Haken. Their proof relies on a careful case-by-case analysis carried out by computer. They showed that if the Four Color Theorem were false, there would have to be a counterexample of one of approximately 2000 different types, and they then showed that none of these types exists. They used over 1000 hours of computer time in their proof. This proof generated a large amount of controversy, because computers played such an important role in it. For example, could there be an error in a computer program that led to incorrect results? Was their argument really a proof if it depended on what could be unreliable computer output?

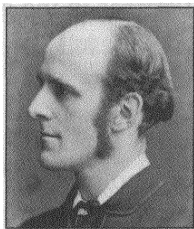
Note that the Four Color Theorem applies only to planar graphs. Nonplanar graphs can have arbitrarily large chromatic numbers, as will be shown in Example 2.

Two things are required to show that the chromatic number of a graph is k . First, we must show that the graph can be colored with k colors. This can be done by constructing such a coloring. Second, we must show that the graph cannot be colored using fewer than k colors. Examples 1–4 illustrate how chromatic numbers can be found.

EXAMPLE 1 What are the chromatic numbers of the graphs G and H shown in Figure 3?



Solution: The chromatic number of G is at least three, because the vertices a , b , and c must be assigned different colors. To see if G can be colored with three colors, assign red to a , blue to b , and green to c . Then, d can (and must) be colored red because it is adjacent to b and c . Furthermore, e can (and must) be colored green because it is adjacent only to vertices colored red and blue, and f can (and must) be colored blue because it is adjacent only to vertices colored red and green. Finally, g can (and must) be colored red because it is adjacent only to vertices



ALFRED BRAY KEMPE (1849–1922) Kempe was a barrister and a leading authority on ecclesiastical law. However, having studied mathematics at Cambridge University, he retained his interest in it, and later in life he devoted considerable time to mathematical research. Kempe made contributions to kinematics, the branch of mathematics dealing with motion, and to mathematical logic. However, Kempe is best remembered for his fallacious proof of the Four Color Theorem.

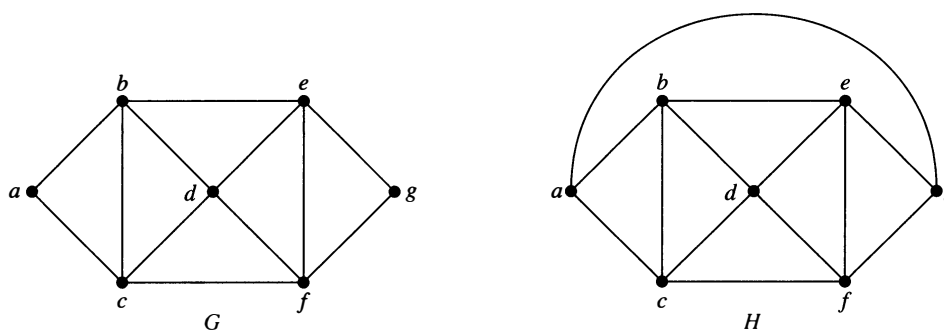


FIGURE 3 The Simple Graphs G and H .

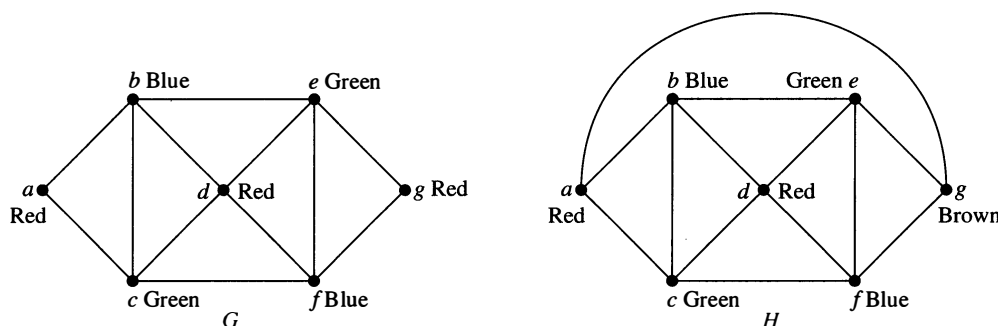


FIGURE 4 Colorings of the Graphs G and H .

colored blue and green. This produces a coloring of G using exactly three colors. Figure 4 displays such a coloring.

The graph H is made up of the graph G with an edge connecting a and g . Any attempt to color H using three colors must follow the same reasoning as that used to color G , except at the last stage, when all vertices other than g have been colored. Then, because g is adjacent (in H) to vertices colored red, blue, and green, a fourth color, say brown, needs to be used. Hence, H has a chromatic number equal to 4. A coloring of H is shown in Figure 4. ◀

EXAMPLE 2 What is the chromatic number of K_n ?

Solution: A coloring of K_n can be constructed using n colors by assigning a different color to each vertex. Is there a coloring using fewer colors? The answer is no. No two vertices can be assigned the same color, because every two vertices of this graph are adjacent. Hence, the chromatic number of $K_n = n$. That is, $\chi(K_n) = n$. (Recall that K_n is not planar when $n \geq 5$, so

HISTORICAL NOTE In 1852, an ex-student of Augustus De Morgan, Francis Guthrie, noticed that the counties in England could be colored using four colors so that no adjacent counties were assigned the same color. On this evidence, he conjectured that the Four Color Theorem was true. Francis told his brother Frederick, at that time a student of De Morgan, about this problem. Frederick in turn asked his teacher De Morgan about his brother's conjecture. De Morgan was extremely interested in this problem and publicized it throughout the mathematical community. In fact, the first written reference to the conjecture can be found in a letter from De Morgan to Sir William Rowan Hamilton. Although De Morgan thought Hamilton would be interested in this problem, Hamilton apparently was not interested in it, because it had nothing to do with quaternions.



HISTORICAL NOTE Although a simpler proof of the Four Color Theorem was found by Robertson, Sanders, Seymour, and Thomas in 1996, reducing the computational part of the proof to examining 633 configurations, no proof that does not rely on extensive computation has yet been found.

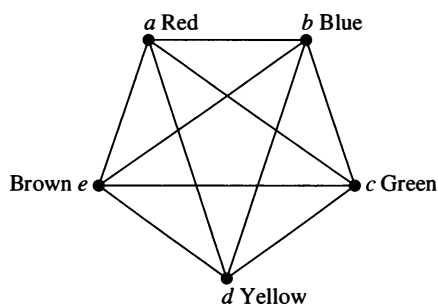


FIGURE 5 A Coloring of K_5 .

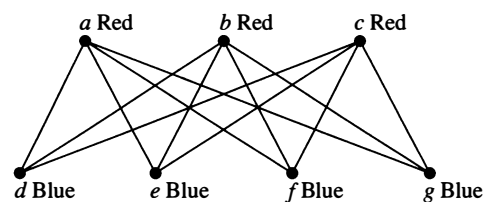


FIGURE 6 A Coloring of $K_{3,4}$.

this result does not contradict the Four Color Theorem.) A coloring of K_5 using five colors is shown in Figure 5. ◀

EXAMPLE 3 What is the chromatic number of the complete bipartite graph $K_{m,n}$, where m and n are positive integers?

Solution: The number of colors needed may seem to depend on m and n . However, as Theorem 4 in Section 9.2 tells us, only two colors are needed, because $K_{m,n}$ is a bipartite graph. Hence, $\chi(K_{m,n}) = 2$. This means that we can color the set of m vertices with one color and the set of n vertices with a second color. Because edges connect only a vertex from the set of m vertices and a vertex from the set of n vertices, no two adjacent vertices have the same color. A coloring of $K_{3,4}$ with two colors is displayed in Figure 6. ◀

EXAMPLE 4 What is the chromatic number of the graph C_n , where $n \geq 3$? (Recall that C_n is the cycle with n vertices.)

Solution: We will first consider some individual cases. To begin, let $n = 6$. Pick a vertex and color it red. Proceed clockwise in the planar depiction of C_6 shown in Figure 7. It is necessary to assign a second color, say blue, to the next vertex reached. Continue in the clockwise direction; the third vertex can be colored red, the fourth vertex blue, and the fifth vertex red. Finally, the sixth vertex, which is adjacent to the first, can be colored blue. Hence, the chromatic number of C_6 is 2. Figure 7 displays the coloring constructed here.

Next, let $n = 5$ and consider C_5 . Pick a vertex and color it red. Proceeding clockwise, it is necessary to assign a second color, say blue, to the next vertex reached. Continuing in the clockwise direction, the third vertex can be colored red, and the fourth vertex can be colored blue.

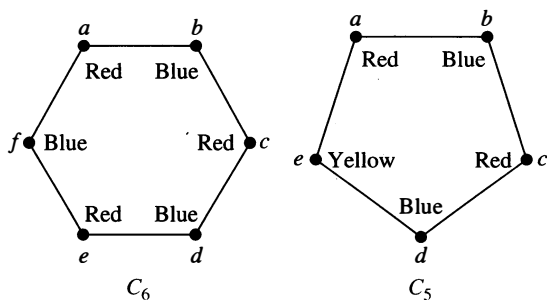


FIGURE 7 Colorings of C_5 and C_6 .

The fifth vertex cannot be colored either red or blue, because it is adjacent to the fourth vertex and the first vertex. Consequently, a third color is required for this vertex. Note that we would have also needed three colors if we had colored vertices in the counterclockwise direction. Thus, the chromatic number of C_5 is 3. A coloring of C_5 using three colors is displayed in Figure 7.

In general, two colors are needed to color C_n when n is even. To construct such a coloring, simply pick a vertex and color it red. Proceed around the graph in a clockwise direction (using a planar representation of the graph) coloring the second vertex blue, the third vertex red, and so on. The n th vertex can be colored blue, because the two vertices adjacent to it, namely the $(n - 1)$ st and the first vertices, are both colored red.

When n is odd and $n > 1$, the chromatic number of C_n is 3. To see this, pick an initial vertex. To use only two colors, it is necessary to alternate colors as the graph is traversed in a clockwise direction. However, the n th vertex reached is adjacent to two vertices of different colors, namely, the first and $(n - 1)$ st. Hence, a third color must be used.

We have shown that $\chi(C_n) = 2$ if n is an even positive integer with $n \geq 4$ and $\chi(C_n) = 3$ if n is an odd positive integer with $n \geq 3$. ◀



The best algorithms known for finding the chromatic number of a graph have exponential worst-case time complexity (in the number of vertices of the graph). Even the problem of finding an approximation to the chromatic number of a graph is difficult. It has been shown that if there were an algorithm with polynomial worst-case time complexity that could approximate the chromatic number of a graph up to a factor of 2 (that is, construct a bound that was no more than double the chromatic number of the graph), then an algorithm with polynomial worst-case time complexity for finding the chromatic number of the graph would also exist.

Applications of Graph Colorings

Graph coloring has a variety of applications to problems involving scheduling and assignments. (Note that because no efficient algorithm is known for graph coloring, this does not lead to efficient algorithms for scheduling and assignments.) Examples of such applications will be given here. The first application deals with the scheduling of final exams.

EXAMPLE 5 Scheduling Final Exams How can the final exams at a university be scheduled so that no student has two exams at the same time?

Solution: This scheduling problem can be solved using a graph model, with vertices representing courses and with an edge between two vertices if there is a common student in the courses they represent. Each time slot for a final exam is represented by a different color. A scheduling of the exams corresponds to a coloring of the associated graph.

For instance, suppose there are seven finals to be scheduled. Suppose the courses are numbered 1 through 7. Suppose that the following pairs of courses have common students: 1 and 2, 1 and 3, 1 and 4, 1 and 7, 2 and 3, 2 and 4, 2 and 5, 2 and 7, 3 and 4, 3 and 6, 3 and 7, 4 and 5, 4 and 6, 5 and 6, 5 and 7, and 6 and 7. In Figure 8 the graph associated with this set of classes is shown. A scheduling consists of a coloring of this graph.

Because the chromatic number of this graph is 4 (the reader should verify this), four time slots are needed. A coloring of the graph using four colors and the associated schedule are shown in Figure 9. ◀

Now consider an application to the assignment of television channels.

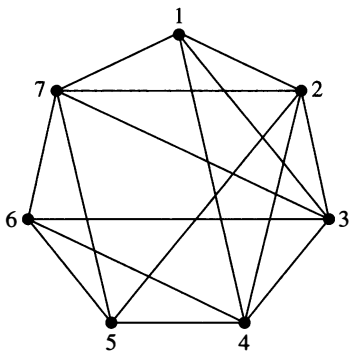
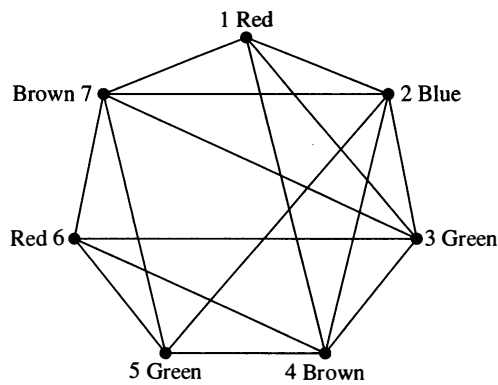


FIGURE 8 The Graph Representing the Scheduling of Final Exams.



Time Period	Courses
I	1, 6
II	2
III	3, 5
IV	4, 7

FIGURE 9 Using a Coloring to Schedule Final Exams.

EXAMPLE 6 Frequency Assignments Television channels 2 through 13 are assigned to stations in North America so that no two stations within 150 miles can operate on the same channel. How can the assignment of channels be modeled by graph coloring?

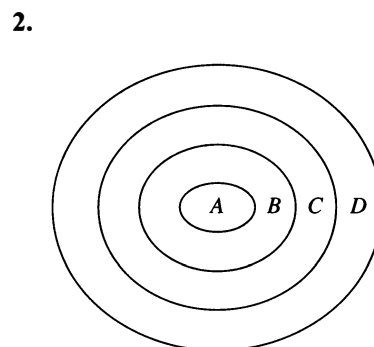
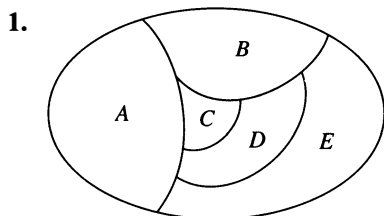
Solution: Construct a graph by assigning a vertex to each station. Two vertices are connected by an edge if they are located within 150 miles of each other. An assignment of channels corresponds to a coloring of the graph, where each color represents a different channel. ◀

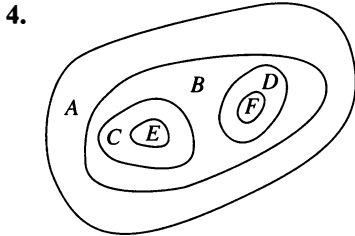
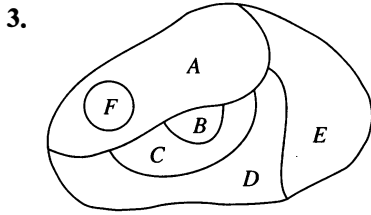
An application of graph coloring to compilers is considered in Example 7.

EXAMPLE 7 Index Registers In efficient compilers the execution of loops is speeded up when frequently used variables are stored temporarily in index registers in the central processing unit, instead of in regular memory. For a given loop, how many index registers are needed? This problem can be addressed using a graph coloring model. To set up the model, let each vertex of a graph represent a variable in the loop. There is an edge between two vertices if the variables they represent must be stored in index registers at the same time during the execution of the loop. Thus, the chromatic number of the graph gives the number of index registers needed, because different registers must be assigned to variables when the vertices representing these variables are adjacent in the graph. ◀

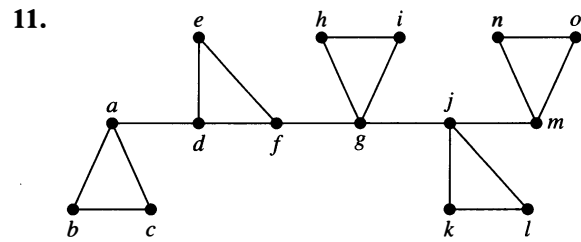
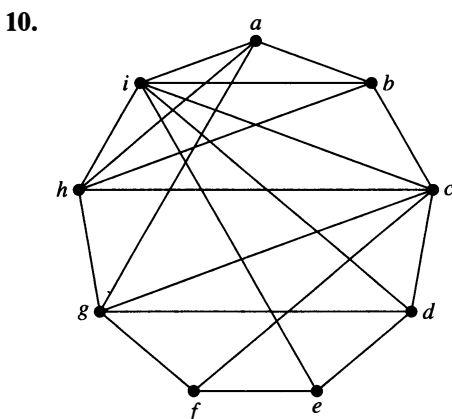
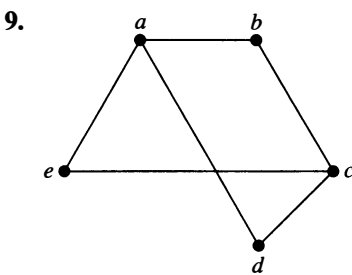
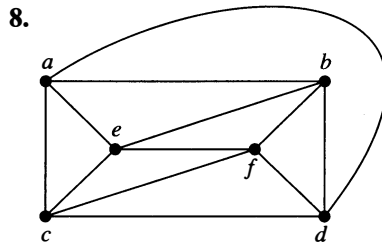
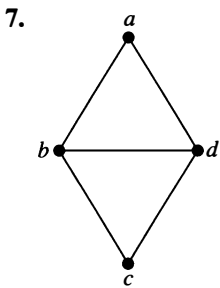
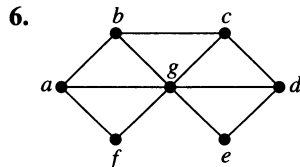
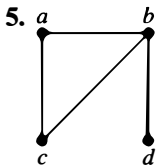
Exercises

In Exercises 1–4 construct the dual graph for the map shown. Then find the number of colors needed to color the map so that no two adjacent regions have the same color.





In Exercises 5–11 find the chromatic number of the given graph.



12. For the graphs in Exercises 5–11, decide whether it is possible to decrease the chromatic number by removing a single vertex and all edges incident with it.
13. Which graphs have a chromatic number of 1?
14. What is the least number of colors needed to color a map of the United States? Do not consider adjacent states that meet only at a corner. Suppose that Michigan is one region. Consider the vertices representing Alaska and Hawaii as isolated vertices.
15. What is the chromatic number of W_n ?
16. Show that a simple graph that has a circuit with an odd number of vertices in it cannot be colored using two colors.
17. Schedule the final exams for Math 115, Math 116, Math 185, Math 195, CS 101, CS 102, CS 273, and CS 473, using the fewest number of different time slots, if there are no students taking both Math 115 and CS 473, both Math 116 and CS 473, both Math 195 and CS 101, both Math 195 and CS 102, both Math 115 and Math 116, both Math 115 and Math 185, and both Math 185 and Math 195, but there are students in every other combination of courses.
18. How many different channels are needed for six stations located at the distances shown in the table, if two stations cannot use the same channel when they are within 150 miles of each other?

	1	2	3	4	5	6
1	—	85	175	200	50	100
2	85	—	125	175	100	160
3	175	125	—	100	200	250
4	200	175	100	—	210	220
5	50	100	200	210	—	100
6	100	160	250	220	100	—

19. The mathematics department has six committees each meeting once a month. How many different meeting times must be used to ensure that no member is scheduled to attend two meetings at the same time if the committees are $C_1 = \{\text{Arlinghaus, Brand, Zaslavsky}\}$, $C_2 = \{\text{Brand, Lee, Rosen}\}$, $C_3 = \{\text{Arlinghaus, Rosen, Zaslavsky}\}$, $C_4 = \{\text{Lee, Rosen, Zaslavsky}\}$, $C_5 = \{\text{Arlinghaus, Brand}\}$, and $C_6 = \{\text{Brand, Rosen, Zaslavsky}\}$?

20. A zoo wants to set up natural habitats in which to exhibit its animals. Unfortunately, some animals will eat some of the others when given the opportunity. How can a graph model and a coloring be used to determine the number of different habitats needed and the placement of the animals in these habitats?

An **edge coloring** of a graph is an assignment of colors to edges so that edges incident with a common vertex are assigned different colors. The **edge chromatic number** of a graph is the smallest number of colors that can be used in an edge coloring of the graph.

21. Find the edge chromatic number of each of the graphs in Exercises 5–11.

22. Find the edge chromatic numbers of

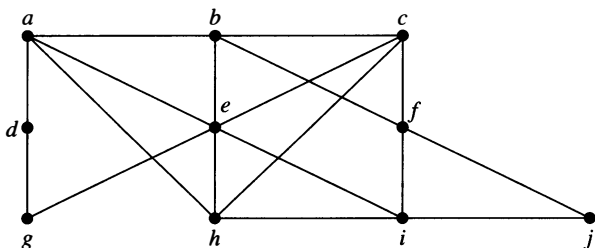
- a) K_n . b) $K_{m,n}$.
 c) C_n . d) W_n .

23. Seven variables occur in a loop of a computer program. The variables and the steps during which they must be stored are t : steps 1 through 6; u : step 2; v : steps 2 through 4; w : steps 1, 3, and 5; x : steps 1 and 6; y : steps 3 through 6; and z : steps 4 and 5. How many different index registers are needed to store these variables during execution?

24. What can be said about the chromatic number of a graph that has K_n as a subgraph?

This algorithm can be used to color a simple graph: First, list the vertices $v_1, v_2, v_3, \dots, v_n$ in order of decreasing degree so that $\deg(v_1) \geq \deg(v_2) \geq \dots \geq \deg(v_n)$. Assign color 1 to v_1 and to the next vertex in the list not adjacent to v_1 (if one exists), and successively to each vertex in the list not adjacent to a vertex already assigned color 1. Then assign color 2 to the first vertex in the list not already colored. Successively assign color 2 to vertices in the list that have not already been colored and are not adjacent to vertices assigned color 2. If uncolored vertices remain, assign color 3 to the first vertex in the list not yet colored, and use color 3 to successively color those vertices not already colored and not adjacent to vertices assigned color 3. Continue this process until all vertices are colored.

25. Construct a coloring of the graph shown using this algorithm.



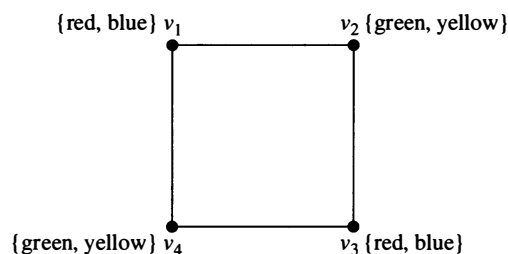
26. Use pseudocode to describe this coloring algorithm.
 27. Show that the coloring produced in this algorithm may use more colors than are necessary to color a graph.

A connected graph G is called **chromatically k -critical** if the chromatic number of G is k , but for every edge e of G , the

chromatic number of the graph obtained by deleting this edge from G is $k - 1$.

28. Show that C_n is chromatically 3-critical whenever n is an odd positive integer, $n \geq 3$.
 29. Show that W_n is chromatically 4-critical whenever n is an odd integer, $n \geq 3$.
 30. Show that W_4 is not chromatically 3-critical.
 31. Show that if G is a chromatically k -critical graph, then the degree of every vertex of G is at least $k - 1$.

A **k -tuple coloring** of a graph G is an assignment of a set of k different colors to each of the vertices of G such that no two adjacent vertices are assigned a common color. We denote by $\chi_k(G)$ the smallest positive integer n such that G has a k -tuple coloring using n colors. For example, $\chi_2(C_4) = 4$. To see this, note that using only four colors we can assign two colors to each vertex of C_4 , as illustrated, so that no two adjacent vertices are assigned the same color. Furthermore, no fewer than four colors suffice because the vertices v_1 and v_2 each must be assigned two colors, and a common color cannot be assigned to both v_1 and v_2 . (For more information about k -tuple coloring, see [MiRo91].)



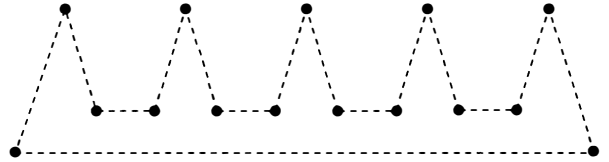
32. Find these values:
 a) $\chi_2(K_3)$ b) $\chi_2(K_4)$ c) $\chi_2(W_4)$
 d) $\chi_2(C_5)$ e) $\chi_2(K_{3,4})$ f) $\chi_3(K_5)$
 *g) $\chi_3(C_5)$ h) $\chi_3(K_{4,5})$
- *33. Let G and H be the graphs displayed in Figure 3. Find
 a) $\chi_2(G)$. b) $\chi_2(H)$.
 c) $\chi_3(G)$. d) $\chi_3(H)$.
34. What is $\chi_k(G)$ if G is a bipartite graph and k is a positive integer?
35. Frequencies for mobile radio (or cellular) telephones are assigned by zones. Each zone is assigned a set of frequencies to be used by vehicles in that zone. The same frequency cannot be used in zones where interference is a problem. Explain how a k -tuple coloring can be used to assign k frequencies to each mobile radio zone in a region.
- *36. Show that every planar graph G can be colored using six or fewer colors. [Hint: Use mathematical induction on the number of vertices of the graph. Apply Corollary 2 of Section 9.7 to find a vertex v with $\deg(v) \leq 5$. Consider the subgraph of G obtained by deleting v and all edges incident with it.]
- **37. Show that every planar graph G can be colored using five or fewer colors. [Hint: Use the hint provided for Exercise 36.]

The famous Art Gallery Problem asks how many guards are needed to see all parts of an art gallery, where the gallery is the interior and boundary of a polygon with n sides. To state this problem more precisely, we need some terminology. A point x inside or on the boundary of a simple polygon P **covers** or **sees** a point y inside or on P if all points on the line segment xy are in the interior or on the boundary of P . We say that a set of points is a **guarding set** of a simple polygon P if for every point y inside P or on the boundary of P there is a point x in this guarding set that sees y . Denote by $G(P)$ the minimum number of points needed to guard the simple polygon P . The **Art Gallery Problem** asks for the function $g(n)$, which is the maximum value of $G(P)$ over all simple polygons with n vertices. That is, $g(n)$ is the minimum positive integer for which it is guaranteed that a simple polygon with n vertices can be guarded with $g(n)$ or fewer guards.

38. Show that $g(3) = 1$ and $g(4) = 1$ by showing that all triangles and quadrilaterals can be guarded using one point.
39. Show that $g(5) = 1$. That is, show that all pentagons can be guarded using one point. [Hint: Show that there are either 0, 1, or 2 vertices with an interior angle greater than 180 degrees and that in each case, one guard suffices.]
40. Show that $g(6) = 2$ by first using Exercises 38 and 39 as well as Lemma 1 in Section 4.2 to show that $g(6) \leq 2$

and then find a simple hexagon for which two guards are needed.

- *41. Show that $g(n) \geq \lfloor n/3 \rfloor$. [Hint: Consider the polygon with $3k$ vertices that resembles a comb with k prongs, such as the polygon with 15 sides shown here.]



- *42. Solve the Art Gallery Problem by proving the **Art Gallery Theorem**, which states that at most $\lfloor n/3 \rfloor$ guards are needed to guard the interior and boundary of a simple polygon with n vertices. [Hint: Use Theorem 1 in Section 4.2 to triangulate the simple polygon into $n - 2$ triangles. Then show that it is possible to color the vertices of the triangulated polygon using three colors so that no two adjacent vertices have the same color. Use induction and Exercise 23 in Section 4.2. Finally, put guards at all vertices that are colored red, where red is the color used least in the coloring of the vertices. Show that placing guards at these points is all that is needed.]

Key Terms and Results

TERMS

- undirected edge:** an edge associated to a set $\{u, v\}$, where u and v are vertices
- directed edge:** an edge associated to an ordered pair (u, v) , where u and v are vertices
- multiple edges:** distinct edges connecting the same vertices
- multiple directed edges:** distinct directed edges associated with the same ordered pair (u, v) , where u and v are vertices
- loop:** an edge connecting a vertex with itself
- undirected graph:** a set of vertices and a set of undirected edges each of which is associated with a set of one or two of these vertices
- simple graph:** an undirected graph with no multiple edges or loops
- multigraph:** an undirected graph that may contain multiple edges but no loops
- pseudograph:** an undirected graph that may contain multiple edges and loops
- directed graph:** a set of vertices together with a set of directed edges each of which is associated with an ordered pair of vertices
- directed multigraph:** a graph with directed edges that may contain multiple directed edges
- simple directed graph:** a directed graph without loops or multiple directed edges
- adjacent:** two vertices are adjacent if there is an edge between them

incident: an edge is incident with a vertex if the vertex is an endpoint of that edge

deg(v) (degree of the vertex v in an undirected graph): the number of edges incident with v with loops counted twice

deg⁻(v) (the in-degree of the vertex v in a graph with directed edges): the number of edges with v as their terminal vertex

deg⁺(v) (the out-degree of the vertex v in a graph with directed edges): the number of edges with v as their initial vertex

underlying undirected graph of a graph with directed edges: the undirected graph obtained by ignoring the directions of the edges

K_n (complete graph on n vertices): the undirected graph with n vertices where each pair of vertices is connected by an edge

bipartite graph: a graph with vertex set that can be partitioned into subsets V_1 and V_2 such that each edge connects a vertex in V_1 and a vertex in V_2

$K_{m,n}$ (complete bipartite graph): the graph with vertex set partitioned into a subset of m elements and a subset of n elements such that two vertices are connected by an edge if and only if one is in the first subset and the other is in the second subset

C_n (cycle of size n), $n \geq 3$: the graph with n vertices v_1, v_2, \dots, v_n and edges $\{v_1, v_2\}, \{v_2, v_3\}, \dots, \{v_{n-1}, v_n\}, \{v_n, v_1\}$

W_n (wheel of size n), $n \geq 3$: the graph obtained from C_n by adding a vertex and edges from this vertex to the original vertices in C_n

Q_n (n -cube), $n \geq 1$: the graph that has the 2^n bit strings of length n as its vertices and edges connecting every pair of bit strings that differ by exactly one bit

isolated vertex: a vertex of degree zero

pendant vertex: a vertex of degree one

regular graph: a graph where all vertices have the same degree

subgraph of a graph $G = (V, E)$: a graph (W, F) , where W is a subset of V and F is a subset of E

$G_1 \cup G_2$ (union of G_1 and G_2): the graph $(V_1 \cup V_2, E_1 \cup E_2)$, where $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$

adjacency matrix: a matrix representing a graph using the adjacency of vertices

incidence matrix: a matrix representing a graph using the incidence of edges and vertices

isomorphic simple graphs: the simple graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are isomorphic if there is a one-to-one correspondence f from V_1 to V_2 such that $\{f(v_1), f(v_2)\} \in E_2$ if and only if $\{v_1, v_2\} \in E_1$ for all v_1 and v_2 in V_1

invariant: a property that isomorphic graphs either both have or both do not have

path from u to v in an undirected graph: a sequence of edges e_1, e_2, \dots, e_n , where e_i is associated to $\{x_i, x_{i+1}\}$ for $i = 0, 1, \dots, n$, where $x_0 = u$ and $x_{n+1} = v$

path from u to v in a graph with directed edges: a sequence of edges e_1, e_2, \dots, e_n , where e_i is associated to (x_i, x_{i+1}) for $i = 0, 1, \dots, n$, where $x_0 = u$ and $x_{n+1} = v$

simple path: a path that does not contain an edge more than once

circuit: a path of length $n \geq 1$ that begins and ends at the same vertex

connected graph: an undirected graph with the property that there is a path between every pair of vertices

connected component of a graph G : a maximal connected subgraph of G

strongly connected directed graph: a directed graph with the property that there is a directed path from every vertex to every vertex

strongly connected component of a directed graph G : a maximal strongly connected subgraph of G

Euler circuit: a circuit that contains every edge of a graph exactly once

Euler path: a path that contains every edge of a graph exactly once

Hamilton path: a path in a simple graph that passes through each vertex exactly once

Hamilton circuit: a circuit in a simple graph that passes through each vertex exactly once

weighted graph: a graph with numbers assigned to its edges

shortest-path problem: the problem of determining the path in a weighted graph such that the sum of the weights of the edges in this path is a minimum over all paths between specified vertices

traveling salesman problem: the problem that asks for the circuit of shortest total length that visits every vertex of the graph exactly once

planar graph: a graph that can be drawn in the plane with no crossings

regions of a representation of a planar graph: the regions the plane is divided into by the planar representation of the graph

elementary subdivision: the removal of an edge $\{u, v\}$ of an undirected graph and the addition of a new vertex w together with edges $\{u, w\}$ and $\{w, v\}$

homeomorphic: two undirected graphs are homeomorphic if they can be obtained from the same graph by a sequence of elementary subdivisions

graph coloring: an assignment of colors to the vertices of a graph so that no two adjacent vertices have the same color

chromatic number: the minimum number of colors needed in a coloring of a graph

RESULTS

There is an Euler circuit in a connected multigraph if and only if every vertex has even degree.

There is an Euler path in a connected multigraph if and only if at most two vertices have odd degree.

Dijkstra's algorithm: a procedure for finding a shortest path between two vertices in a weighted graph (see Section 9.6).

Euler's formula: $r = e - v + 2$ where r , e , and v are the number of regions of a planar representation, the number of edges, and the number of vertices, respectively, of a planar graph.

Kuratowski's Theorem: A graph is nonplanar if and only if it contains a subgraph homeomorphic to $K_{3,3}$ or K_5 . (Proof beyond scope of this book.)

The Four Color Theorem: Every planar graph can be colored using no more than four colors. (Proof far beyond the scope of this book!)

Review Questions

1. a) Define a simple graph, a multigraph, a pseudograph, a directed graph, and a directed multigraph.
b) Use an example to show how each of the types of graph in part (a) can be used in modeling. For example,

explain how to model different aspects of a computer network or airline routes.

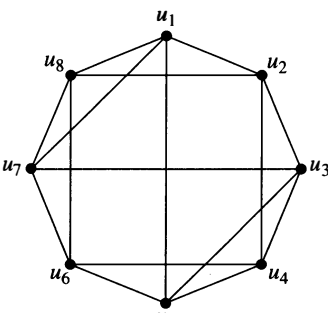
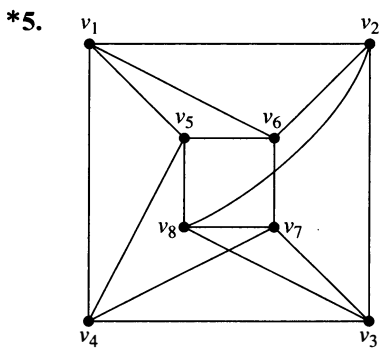
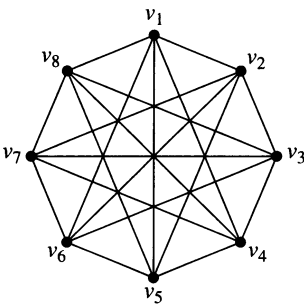
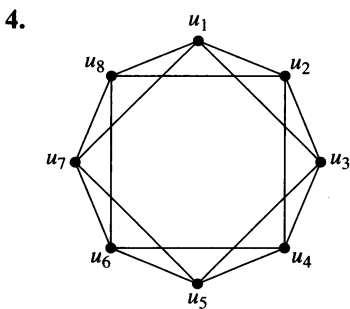
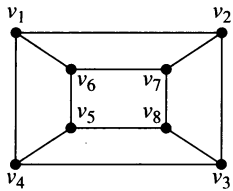
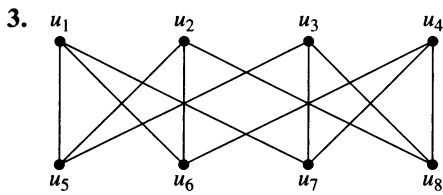
2. Give at least four examples of how graphs are used in modeling.

3. What is the relationship between the sum of the degrees of the vertices in an undirected graph and the number of edges in this graph? Explain why this relationship holds.
4. Why must there be an even number of vertices of odd degree in an undirected graph?
5. What is the relationship between the sum of the in-degrees and the sum of the out-degrees of the vertices in a directed graph? Explain why this relationship holds.
6. Describe the following families of graphs.
 - a) K_n , the complete graph on n vertices
 - b) $K_{m,n}$, the complete bipartite graph on m and n vertices
 - c) C_n , the cycle with n vertices
 - d) W_n , the wheel of size n
 - e) Q_n , the n -cube
7. How many vertices and how many edges are there in each of the graphs in the families in Question 6?
8.
 - a) What is a bipartite graph?
 - b) Which of the graphs K_n , C_n , and W_n are bipartite?
 - c) How can you determine whether an undirected graph is bipartite?
9.
 - a) Describe three different methods that can be used to represent a graph.
 - b) Draw a simple graph with at least five vertices and eight edges. Illustrate how it can be represented using the methods you described in part (a).
10.
 - a) What does it mean for two simple graphs to be isomorphic?
 - b) What is meant by an invariant with respect to isomorphism for simple graphs? Give at least five examples of such invariants.
 - c) Give an example of two graphs that have the same numbers of vertices, edges, and degrees of vertices, but that are not isomorphic.
 - d) Is a set of invariants known that can be used to efficiently determine whether two simple graphs are isomorphic?
11.
 - a) What does it mean for a graph to be connected?
 - b) What are the connected components of a graph?
12.
 - a) Explain how an adjacency matrix can be used to represent a graph.
 - b) How can adjacency matrices be used to determine whether a function from the vertex set of a graph G to the vertex set of a graph H is an isomorphism?
- c) How can the adjacency matrix of a graph be used to determine the number of paths of length r , where r is a positive integer, between two vertices of a graph?
13.
 - a) Define an Euler circuit and an Euler path in an undirected graph.
 - b) Describe the famous Königsberg bridge problem and explain how to rephrase it in terms of an Euler circuit.
 - c) How can it be determined whether an undirected graph has an Euler path?
 - d) How can it be determined whether an undirected graph has an Euler circuit?
14.
 - a) Define a Hamilton circuit in a simple graph.
 - b) Give some properties of a simple graph that imply that it does not have a Hamilton circuit.
15. Give examples of at least two problems that can be solved by finding the shortest path in a weighted graph.
16.
 - a) Describe Dijkstra's algorithm for finding the shortest path in a weighted graph between two vertices.
 - b) Draw a weighted graph with at least 10 vertices and 20 edges. Use Dijkstra's algorithm to find the shortest path between two vertices of your choice in the graph.
17.
 - a) What does it mean for a graph to be planar?
 - b) Give an example of a nonplanar graph.
18.
 - a) What is Euler's formula for planar graphs?
 - b) How can Euler's formula for planar graphs be used to show that a simple graph is nonplanar?
19. State Kuratowski's Theorem on the planarity of graphs and explain how it characterizes which graphs are planar.
20.
 - a) Define the chromatic number of a graph.
 - b) What is the chromatic number of the graph K_n when n is a positive integer?
 - c) What is the chromatic number of the graph C_n when n is a positive integer greater than 2?
 - d) What is the chromatic number of the graph $K_{m,n}$ when m and n are positive integers?
21. State the Four Color Theorem. Are there graphs that cannot be colored with four colors?
22. Explain how graph coloring can be used in modeling. Use at least two different examples.

Supplementary Exercises

1. How many edges does a 50-regular graph with 100 vertices have?
2. How many nonisomorphic subgraphs does K_3 have?

In Exercises 3–5 determine whether the given pair of graphs is isomorphic.



The **complete m -partite graph** K_{n_1, n_2, \dots, n_m} has vertices partitioned into m subsets of n_1, n_2, \dots, n_m elements each, and vertices are adjacent if and only if they are in different subsets in the partition.

6. Draw these graphs.

- a) $K_{1,2,3}$ b) $K_{2,2,2}$ c) $K_{1,2,2,3}$

*7. How many vertices and how many edges does the complete m -partite graph K_{n_1, n_2, \dots, n_m} have?

*8. a) Prove or disprove that there are always two vertices with the same degree in a finite simple graph having at least two vertices.

b) Do the same as in part (a) for finite multigraphs.

Let $G = (V, E)$ be a simple graph. The **subgraph induced** by a subset W of the vertex set V is the graph (W, F) , where the edge set F contains an edge in E if and only if both endpoints of this edge are in W .

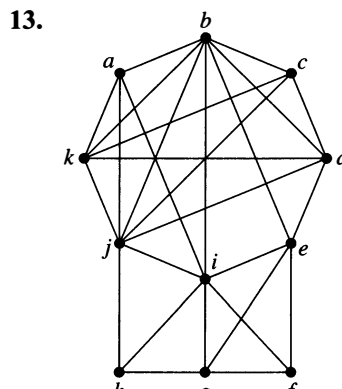
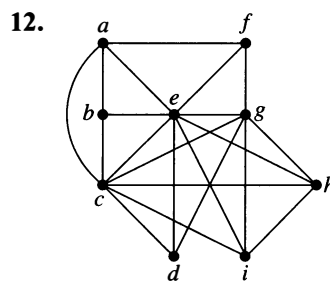
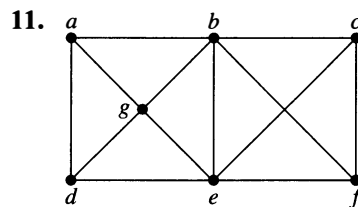
9. Consider the graph shown in Figure 3 of Section 9.4. Find the subgraphs induced by

- a) $\{a, b, c\}$. b) $\{a, e, g\}$.
c) $\{b, c, f, g, h\}$.

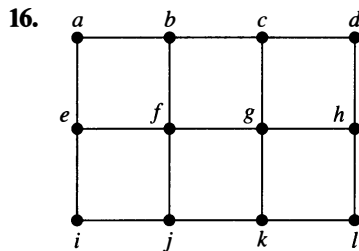
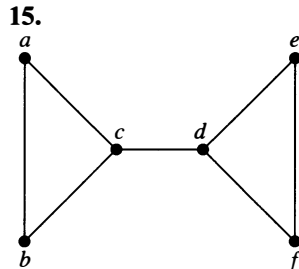
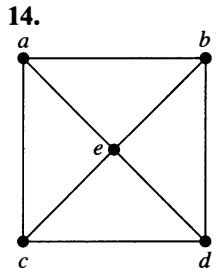
10. Let n be a positive integer. Show that a subgraph induced by a nonempty subset of the vertex set of K_n is a complete graph.



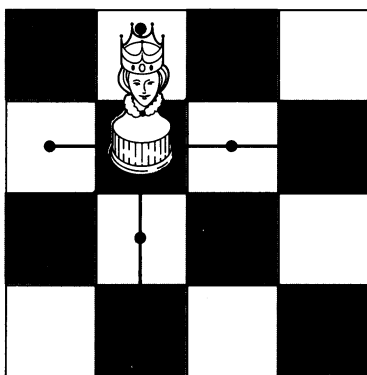
A **clique** in a simple undirected graph is a complete subgraph that is not contained in any larger complete subgraph. In Exercises 11–13 find all cliques in the graph shown.



A **dominating set** of vertices in a simple graph is a set of vertices such that every other vertex is adjacent to at least one vertex of this set. A dominating set with the least number of vertices is called a **minimum dominating set**. In Exercises 14–16 find a minimum dominating set for the given graph.



A simple graph can be used to determine the minimum number of queens on a chessboard that control the entire chessboard. An $n \times n$ chessboard has n^2 squares in an $n \times n$ configuration. A queen in a given position controls all squares in the same row, the same column, and on the two diagonals containing this square, as illustrated. The appropriate simple graph has n^2 vertices, one for each square, and two vertices are adjacent if a queen in the square represented by one of the vertices controls the square represented by the other vertex.



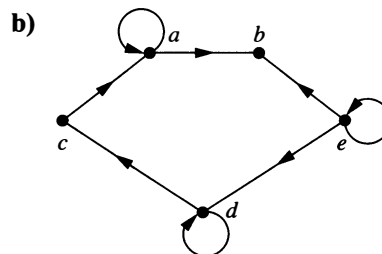
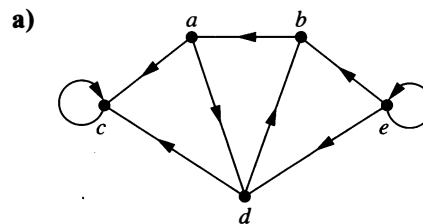
The Squares Controlled by a Queen

17. Construct the simple graph representing the $n \times n$ chessboard with edges representing the control of squares by queens for
- a) $n = 3$. b) $n = 4$.
18. Explain how the concept of a minimum dominating set applies to the problem of determining the minimum number of queens controlling an $n \times n$ chessboard.

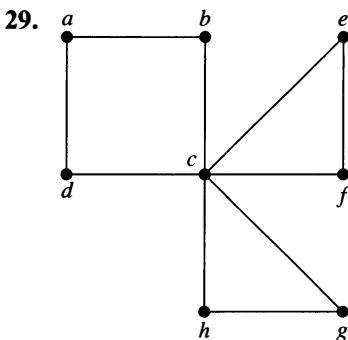
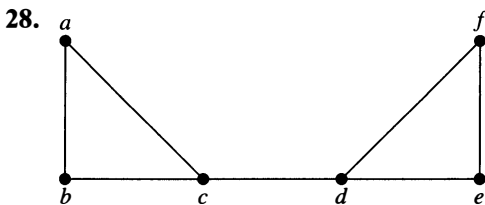
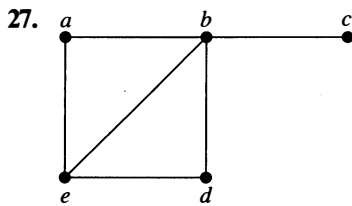
- *19. Find the minimum number of queens controlling an $n \times n$ chessboard for
- a) $n = 3$. b) $n = 4$. c) $n = 5$.
20. Suppose that G_1 and H_1 are isomorphic and that G_2 and H_2 are isomorphic. Prove or disprove that $G_1 \cup G_2$ and $H_1 \cup H_2$ are isomorphic.
21. Show that each of these properties is an invariant that isomorphic simple graphs either both have or both do not have.
- a) connectedness
 b) the existence of a Hamilton circuit
 c) the existence of an Euler circuit
 d) having crossing number C
 e) having n isolated vertices
 f) being bipartite
22. How can the adjacency matrix of \overline{G} be found from the adjacency matrix of G , where G is a simple graph?
23. How many nonisomorphic connected bipartite simple graphs are there with four vertices?
- *24. How many nonisomorphic simple connected graphs with five vertices are there
- a) with no vertex of degree more than two?
 b) with chromatic number equal to four?
 c) that are nonplanar?

A directed graph is **self-converse** if it is isomorphic to its converse.

25. Determine whether the following graphs are self-converse.



26. Show that if the directed graph G is self-converse and H is a directed graph isomorphic to G , then H is also self-converse.
- An **orientation** of an undirected simple graph is an assignment of directions to its edges such that the resulting directed graph is strongly connected. When an orientation of an undirected graph exists, this graph is called **orientable**. In Exercises 27–29 determine whether the given simple graph is orientable.



30. Because traffic is growing heavy in the central part of a city, traffic engineers are planning to change all the streets, which are currently two-way, into one-way streets. Explain how to model this problem.

- *31. Show that a graph is not orientable if it has a cut edge.
- A **tournament** is a simple directed graph such that if u and v are distinct vertices in the graph, exactly one of (u, v) and (v, u) is an edge of the graph.
- 32. How many different tournaments are there with n vertices?
- 33. What is the sum of the in-degree and out-degree of a vertex in a tournament?
- *34. Show that every tournament has a Hamilton path.
- 35. Given two chickens in a flock, one of them is dominant. This defines the **pecking order** of the flock. How can a tournament be used to model pecking order?
- 36. Suppose that G is a connected multigraph with $2k$ vertices of odd degree. Show that there exist k subgraphs that have G as their union, where each of these subgraphs has an Euler path and where no two of these subgraphs have an edge in common. [Hint: Add k edges to the graph connecting pairs of vertices of odd degree and use an Euler circuit in this larger graph.]
- *37. Let G be a simple graph with n vertices. The **band width** of G , denoted by $B(G)$, is the minimum, over all permutations, a_1, a_2, \dots, a_n of the vertices of G , of $\max\{|i - j| \mid a_i \text{ and } a_j \text{ are adjacent}\}$. That is, the band

width is the minimum over all listings of the vertices of the maximum difference in the indices assigned to adjacent vertices. Find the band widths of these graphs.

- a) K_5 b) $K_{1,3}$ c) $K_{2,3}$
- d) $K_{3,3}$ e) Q_3 f) C_5

*38. The **distance** between two distinct vertices v_1 and v_2 of a connected simple graph is the length (number of edges) of the shortest path between v_1 and v_2 . The **radius** of a graph is the minimum overall vertices v of the maximum distance from v to another vertex. The **diameter** of a graph is the maximum distance between two distinct vertices. Find the radius and diameter of

- a) K_6 . b) $K_{4,5}$. c) Q_3 . d) C_6 .

- *39. a) Show that if the diameter of the simple graph G is at least four, then the diameter of its complement \bar{G} is no more than two.
- b) Show that if the diameter of the simple graph G is at least three, then the diameter of its complement \bar{G} is no more than three.
- *40. Suppose that a multigraph has $2m$ vertices of odd degree. Show that any circuit that contains every edge of the graph must contain at least m edges more than once.
- 41. Find the second shortest path between the vertices a and z in Figure 3 of Section 9.6.
- 42. Devise an algorithm for finding the second shortest path between two vertices in a simple connected weighted graph.
- 43. Find the shortest path between the vertices a and z that passes through the vertex e in the weighted graph in Figure 4 in Section 9.6.
- 44. Devise an algorithm for finding the shortest path between two vertices in a simple connected weighted graph that passes through a specified third vertex.
- *45. Show that if G is a simple graph with at least 11 vertices, then either G or \bar{G} , the complement of G , is nonplanar.
- A set of vertices in a graph is called **independent** if no two vertices in the set are adjacent. The **independence number** of a graph is the maximum number of vertices in an independent set of vertices for the graph.
- *46. What is the independence number of
 - a) K_n ? b) C_n ? c) Q_n ? d) $K_{m,n}$?
- 47. Show that the number of vertices in a simple graph is less than or equal to the product of the independence number and the chromatic number of the graph.
- 48. Show that the chromatic number of a graph is less than or equal to $v - i + 1$, where v is the number of vertices in the graph and i is the independence number of this graph.
- 49. Suppose that to generate a random simple graph with n vertices we first choose a real number p with $0 \leq p \leq 1$. For each of the $C(n, 2)$ pairs of distinct vertices we generate a random number x between 0 and 1. If $0 \leq x \leq p$, we connect these two vertices with an edge; otherwise these vertices are not connected.

- a) What is the probability that a graph with m edges where $0 \leq m \leq C(n, 2)$ is generated?
- b) What is the expected number of edges in a randomly generated graph with n vertices if each edge is included with probability p ?
- c) Show that if $p = 1/2$ then every simple graph with n vertices is equally likely to be generated.

A property retained whenever additional edges are added to a simple graph (without adding vertices) is called **monotone increasing**, and a property that is retained whenever edges are removed from a simple graph (without removing vertices) is called **monotone decreasing**.

50. For each of these properties, determine whether it is monotone increasing and determine whether it is monotone decreasing.

- a) The graph G is connected.
- b) The graph G is not connected.
- c) The graph G has an Euler circuit.
- d) The graph G has a Hamilton circuit.
- e) The graph G is planar.
- f) The graph G has chromatic number four.
- g) The graph G has radius three.
- h) The graph G has diameter three.

51. Show that the graph property P is monotone increasing if and only if the graph property Q is monotone decreasing where Q is the property of not having property P .

- **52. Suppose that P is a monotone increasing property of simple graphs. Show that the probability a random graph with n vertices has property P is a monotonic nondecreasing function of p , the probability an edge is chosen to be in the graph.

Computer Projects

Write programs with these input and output.

1. Given the vertex pairs associated to the edges of an undirected graph, find the degree of each vertex.
2. Given the ordered pairs of vertices associated to the edges of a directed graph, determine the in-degree and out-degree of each vertex.
3. Given the list of edges of a simple graph, determine whether the graph is bipartite.
4. Given the vertex pairs associated to the edges of a graph, construct an adjacency matrix for the graph. (Produce a version that works when loops, multiple edges, or directed edges are present.)
5. Given an adjacency matrix of a graph, list the edges of this graph and give the number of times each edge appears.
6. Given the vertex pairs associated to the edges of an undirected graph and the number of times each edge appears, construct an incidence matrix for the graph.
7. Given an incidence matrix of an undirected graph, list its edges and give the number of times each edge appears.
8. Given a positive integer n , generate an undirected graph by producing an adjacency matrix for the graph so that all simple graphs are equally likely to be generated.
9. Given a positive integer n , generate a directed graph by producing an adjacency matrix for the graph so that all directed graphs are equally likely to be generated.
10. Given the lists of edges of two simple graphs with no more than six vertices, determine whether the graphs are isomorphic.
11. Given an adjacency matrix of a graph and a positive integer n , find the number of paths of length n between two vertices. (Produce a version that works for directed and undirected graphs.)
- *12. Given the list of edges of a simple graph, determine whether it is connected and find the number of connected components if it is not connected.
13. Given the vertex pairs associated to the edges of a multigraph, determine whether it has an Euler circuit and, if not, whether it has an Euler path. Construct an Euler path or circuit if it exists.
- *14. Given the ordered pairs of vertices associated to the edges of a directed multigraph, construct an Euler path or Euler circuit, if such a path or circuit exists.
- **15. Given the list of edges of a simple graph, produce a Hamilton circuit, or determine that the graph does not have such a circuit.
- **16. Given the list of edges of a simple graph, produce a Hamilton path, or determine that the graph does not have such a path.
17. Given the list of edges and weights of these edges of a weighted connected simple graph and two vertices in this graph, find the length of a shortest path between them using Dijkstra's algorithm. Also, find a shortest path.
18. Given the list of edges of an undirected graph, find a coloring of this graph using the algorithm given in the exercise set of Section 9.8.
19. Given a list of students and the courses that they are enrolled in, construct a schedule of final exams.
20. Given the distances between pairs of television stations, assign frequencies to these stations.

Computations and Explorations

Use a computational program or programs you have written to do these exercises.

1. Display all the simple graphs with four vertices.
2. Display a full set of nonisomorphic simple graphs with six vertices.
3. Display a full set of nonisomorphic directed graphs with four vertices.
4. Generate at random 10 different simple graphs each with 20 vertices so that each such graph is equally likely to be generated.
5. Construct a Gray code where the code words are bit strings of length six.
6. Construct knight's tours on chessboards of various sizes.
7. Determine whether each of the graphs you generated in Exercise 4 of this set is planar. If you can, determine the thickness of each of the graphs that are not planar.
8. Determine whether each of the graphs you generated in Exercise 4 of this set is connected. If a graph is not connected, determine the number of connected components of the graph.
9. Generate at random simple graphs with 10 vertices. Stop when you have constructed one with an Euler circuit. Display an Euler circuit in this graph.
10. Generate at random simple graphs with 10 vertices. Stop when you have constructed one with a Hamilton circuit. Display a Hamilton circuit in this graph.
11. Find the chromatic number of each of the graphs you generated in Exercise 4 of this set.
- **12. Find the shortest path a traveling salesperson can take to visit each of the capitals of the 50 states in the United States, traveling by air between cities in a straight line.
- *13. Estimate the probability that a randomly generated simple graph with n vertices is connected for each positive integer n not exceeding ten by generating a set of random simple graphs and determining whether each is connected.
- **14. Work on the problem of determining whether the crossing number of $K(7, 7)$ is 77, 79, or 81. It is known that it equals one of these three values.

Writing Projects

Respond to these questions with essays using outside sources.

1. Describe the origins and development of graph theory prior to the year 1900.
2. Discuss the applications of graph theory to the study of ecosystems.
3. Discuss the applications of graph theory to sociology and psychology.
4. Discuss what can be learned by investigating the properties of the Web graph.
5. Describe algorithms for drawing a graph on paper or on a display given the vertices and edges of the graph. What considerations arise in drawing a graph so that it has the best appearance for understanding its properties?
6. What are some of the capabilities that a software tool for inputting, displaying, and manipulating graphs should have? Which of these capabilities do available tools have?
7. Describe some of the algorithms available for determining whether two graphs are isomorphic and the computational complexity of these algorithms. What is the most efficient such algorithm currently known?
8. Describe how Euler paths can be used to help determine DNA sequences.
9. Define *de Bruijn sequences* and discuss how they arise in applications. Explain how de Bruijn sequences can be constructed using Euler circuits.
10. Describe the *Chinese postman problem* and explain how to solve this problem.
11. Describe some of the different conditions that imply that a graph has a Hamilton circuit.
12. Describe some of the strategies and algorithms used to solve the traveling salesman problem.
13. Describe several different algorithms for determining whether a graph is planar. What is the computational complexity of each of these algorithms?
14. In modeling, very large scale integration (VLSI) graphs are sometimes embedded in a book, with the vertices on the spine and the edges on pages. Define the *book number* of a graph and find the book number of various graphs including K_n for $n = 3, 4, 5,$ and 6 .
15. Discuss the history of the Four Color Theorem.
16. Describe the role computers played in the proof of the Four Color Theorem. How can we be sure that a proof that relies on a computer is correct?
17. Describe and compare several different algorithms for coloring a graph, in terms of whether they produce a coloring with the least number of colors possible and in terms of their complexity.
18. Explain how graph multicolorings can be used in a variety of different models.
19. Explain how the theory of random graphs can be used in nonconstructive existence proofs of graphs with certain properties.

- 10.1 Introduction to Trees
- 10.2 Applications of Trees
- 10.3 Tree Traversal
- 10.4 Spanning Trees
- 10.5 Minimum Spanning Trees

A connected graph that contains no simple circuits is called a tree. Trees were used as long ago as 1857, when the English mathematician Arthur Cayley used them to count certain types of chemical compounds. Since that time, trees have been employed to solve problems in a wide variety of disciplines, as the examples in this chapter will show.

Trees are particularly useful in computer science, where they are employed in a wide range of algorithms. For instance, trees are used to construct efficient algorithms for locating items in a list. They can be used in algorithms, such as Huffman coding, that construct efficient codes saving costs in data transmission and storage. Trees can be used to study games such as checkers and chess and can help determine winning strategies for playing these games. Trees can be used to model procedures carried out using a sequence of decisions. Constructing these models can help determine the computational complexity of algorithms based on a sequence of decisions, such as sorting algorithms.

Procedures for building trees containing every vertex of a graph, including depth-first search and breadth-first search, can be used to systematically explore the vertices of a graph. Exploring the vertices of a graph via depth-first search, also known as backtracking, allows for the systematic search for solutions to a wide variety of problems, such as determining how eight queens can be placed on a chessboard so that no queen can attack another.

We can assign weights to the edges of a tree to model many problems. For example, using weighted trees we can develop algorithms to construct networks containing the least expensive set of telephone lines linking different network nodes.

10.1 Introduction to Trees



In Chapter 9 we showed how graphs can be used to model and solve many problems. In this chapter we will focus on a particular type of graph called a **tree**, so named because such graphs resemble trees. For example, *family trees* are graphs that represent genealogical charts. Family trees use vertices to represent the members of a family and edges to represent parent-child relationships. The family tree of the male members of the Bernoulli family of Swiss mathematicians is shown in Figure 1. The undirected graph representing a family tree (restricted to people of just one gender and with no inbreeding) is an example of a tree.

DEFINITION 1 A *tree* is a connected undirected graph with no simple circuits.

Because a tree cannot have a simple circuit, a tree cannot contain multiple edges or loops. Therefore any tree must be a simple graph.

EXAMPLE 1 Which of the graphs shown in Figure 2 are trees?

Solution: G_1 and G_2 are trees, because both are connected graphs with no simple circuits. G_3 is not a tree because e, b, a, d, e is a simple circuit in this graph. Finally, G_4 is not a tree because it is not connected. ◀

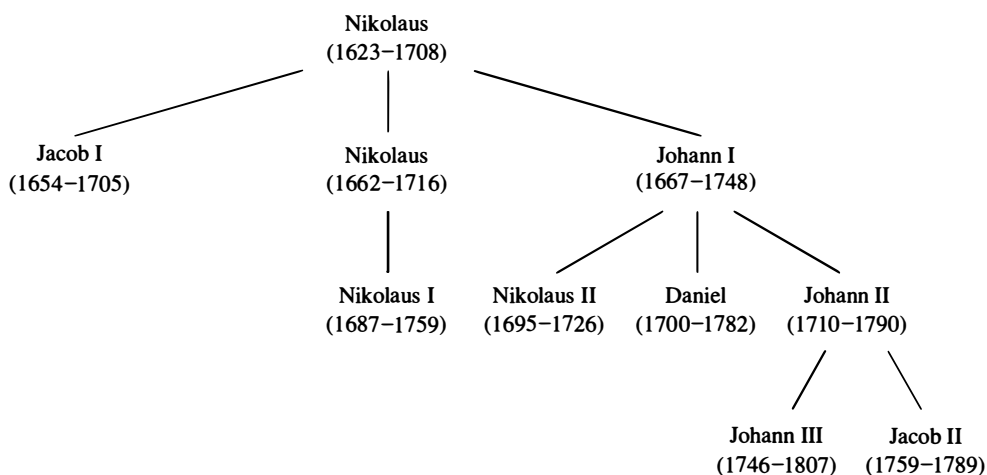


FIGURE 1 The Bernoulli Family of Mathematicians.

Any connected graph that contains no simple circuits is a tree. What about graphs containing no simple circuits that are not necessarily connected? These graphs are called **forests** and have the property that each of their connected components is a tree. Figure 3 displays a forest.

Trees are often defined as undirected graphs with the property that there is a unique simple path between every pair of vertices. Theorem 1 shows that this alternative definition is equivalent to our definition.

THEOREM 1 An undirected graph is a tree if and only if there is a unique simple path between any two of its vertices.

Proof: First assume that T is a tree. Then T is a connected graph with no simple circuits. Let x and y be two vertices of T . Because T is connected, by Theorem 1 of Section 9.4 there is a simple path between x and y . Moreover, this path must be unique, for if there were a second such path, the path formed by combining the first path from x to y followed by the path from y to x obtained by reversing the order of the second path from x to y would form a circuit. This

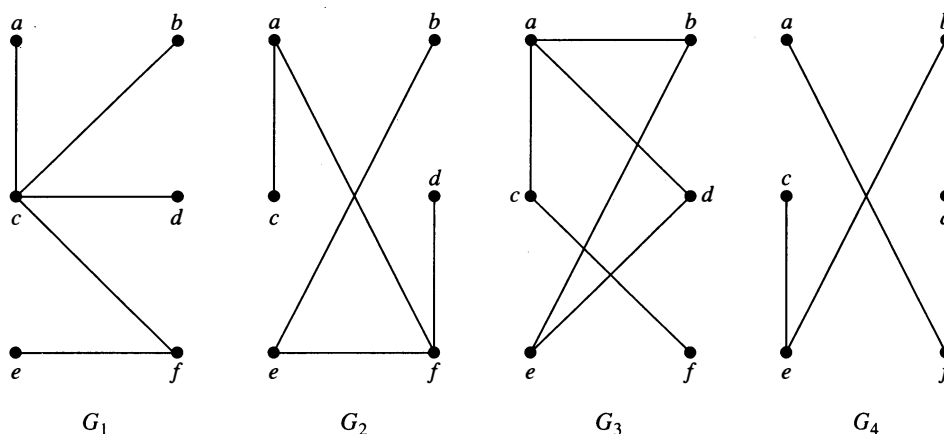


FIGURE 2 Examples of Trees and Graphs That Are Not Trees.

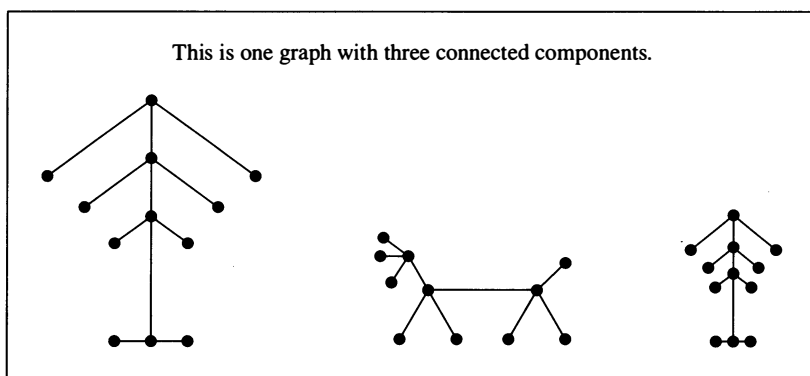


FIGURE 3 Example of a Forest.

implies, using Exercise 49 of Section 9.4, that there is a simple circuit in T . Hence, there is a unique simple path between any two vertices of a tree.

Now assume that there is a unique simple path between any two vertices of a graph T . Then T is connected, because there is a path between any two of its vertices. Furthermore, T can have no simple circuits. To see that this is true, suppose T had a simple circuit that contained the vertices x and y . Then there would be two simple paths between x and y , because the simple circuit is made up of a simple path from x to y and a second simple path from y to x . Hence, a graph with a unique simple path between any two vertices is a tree. ◁

In many applications of trees, a particular vertex of a tree is designated as the **root**. Once we specify a root, we can assign a direction to each edge as follows. Because there is a unique path from the root to each vertex of the graph (by Theorem 1), we direct each edge away from the root. Thus, a tree together with its root produces a directed graph called a **rooted tree**.

DEFINITION 2 A *rooted tree* is a tree in which one vertex has been designated as the root and every edge is directed away from the root.

Rooted trees can also be defined recursively. Refer to Section 4.3 to see how this can be done. We can change an unrooted tree into a rooted tree by choosing any vertex as the root. Note that different choices of the root produce different rooted trees. For instance, Figure 4 displays the rooted trees formed by designating a to be the root and c to be the root, respectively, in the tree T . We usually draw a rooted tree with its root at the top of the graph. The arrows indicating the

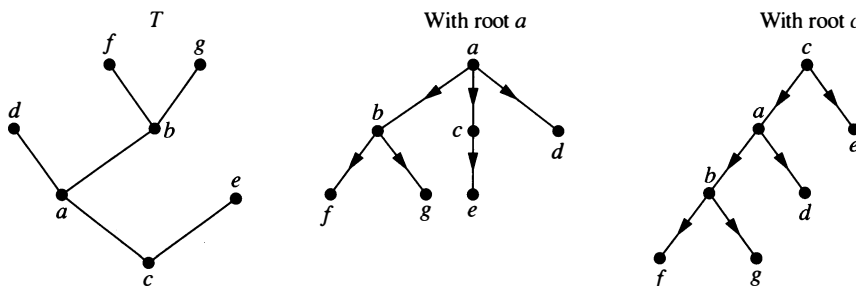
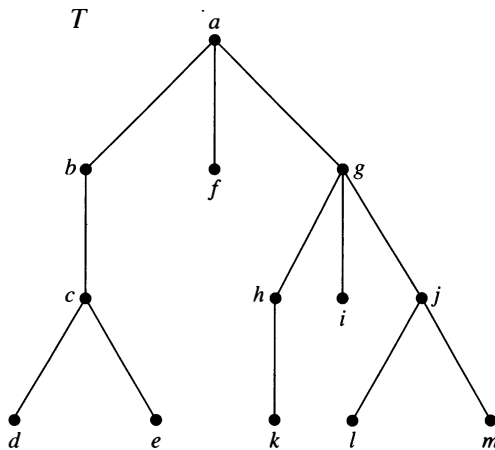
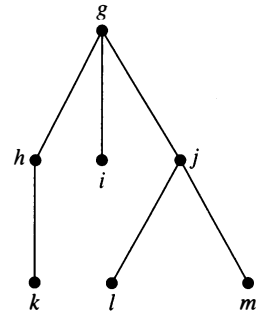


FIGURE 4 A Tree and Rooted Trees Formed by Designating Two Roots.

FIGURE 5 A Rooted Tree T .FIGURE 6 The Subtree Rooted at g .

directions of the edges in a rooted tree can be omitted, because the choice of root determines the directions of the edges.

The terminology for trees has botanical and genealogical origins. Suppose that T is a rooted tree. If v is a vertex in T other than the root, the **parent** of v is the unique vertex u such that there is a directed edge from u to v (the reader should show that such a vertex is unique). When u is the parent of v , v is called a **child** of u . Vertices with the same parent are called **siblings**. The **ancestors** of a vertex other than the root are the vertices in the path from the root to this vertex, excluding the vertex itself and including the root (that is, its parent, its parent's parent, and so on, until the root is reached). The **descendants** of a vertex v are those vertices that have v as an ancestor. A vertex of a tree is called a **leaf** if it has no children. Vertices that have children are called **internal vertices**. The root is an internal vertex unless it is the only vertex in the graph, in which case it is a leaf.

If a is a vertex in a tree, the **subtree** with a as its root is the subgraph of the tree consisting of a and its descendants and all edges incident to these descendants.

EXAMPLE 2 In the rooted tree T (with root a) shown in Figure 5, find the parent of c , the children of g , the siblings of h , all ancestors of e , all descendants of b , all internal vertices, and all leaves. What is the subtree rooted at g ?



Solution: The parent of c is b . The children of g are h , i , and j . The siblings of h are i and j . The ancestors of e are c , b , and a . The descendants of b are c , d , and e . The internal vertices are a , b , c , g , h , and j . The leaves are d , e , f , i , k , l , and m . The subtree rooted at g is shown in Figure 6. ◀

Rooted trees with the property that all of their internal vertices have the same number of children are used in many different applications. Later in this chapter we will use such trees to study problems involving searching, sorting, and coding.

DEFINITION 3



A rooted tree is called an *m -ary tree* if every internal vertex has no more than m children. The tree is called a *full m -ary tree* if every internal vertex has exactly m children. An *m -ary tree* with $m = 2$ is called a *binary tree*.

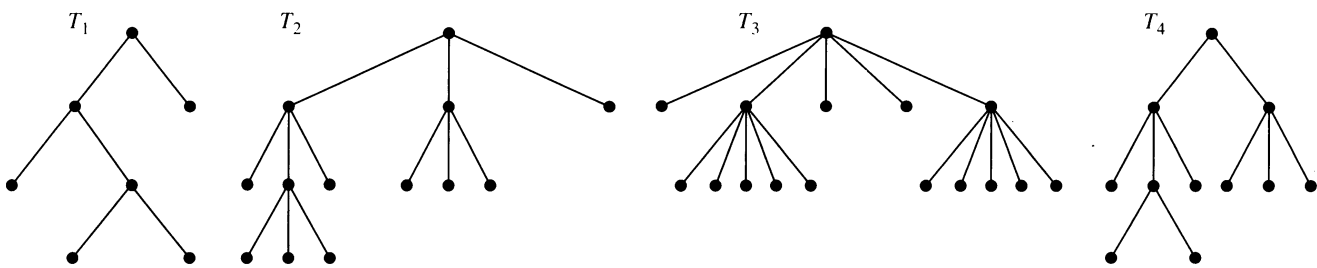


FIGURE 7 Four Rooted Trees.

EXAMPLE 3 Are the rooted trees in Figure 7 full m -ary trees for some positive integer m ?

Solution: T_1 is a full binary tree because each of its internal vertices has two children. T_2 is a full 3-ary tree because each of its internal vertices has three children. In T_3 each internal vertex has five children, so T_3 is a full 5-ary tree. T_4 is not a full m -ary tree for any m because some of its internal vertices have two children and others have three children. ◀

An **ordered rooted tree** is a rooted tree where the children of each internal vertex are ordered. Ordered rooted trees are drawn so that the children of each internal vertex are shown in order from left to right. Note that a representation of a rooted tree in the conventional way determines an ordering for its edges. We will use such orderings of edges in drawings without explicitly mentioning that we are considering a rooted tree to be ordered.

In an ordered binary tree (usually called just a **binary tree**), if an internal vertex has two children, the first child is called the **left child** and the second child is called the **right child**. The tree rooted at the left child of a vertex is called the **left subtree** of this vertex, and the tree rooted at the right child of a vertex is called the **right subtree** of the vertex. The reader should note that for some applications every vertex of a binary tree, other than the root, is designated as a right or a left child of its parent. This is done even when some vertices have only one child. We will make such designations whenever it is necessary, but not otherwise.

Ordered rooted trees can be defined recursively. Binary trees, a type of ordered rooted trees, were defined this way in Section 4.3.

EXAMPLE 4 What are the left and right children of d in the binary tree T shown in Figure 8(a) (where the order is that implied by the drawing)? What are the left and right subtrees of c ?

Solution: The left child of d is f and the right child is g . We show the left and right subtrees of c in Figures 8(b) and 8(c), respectively. ◀

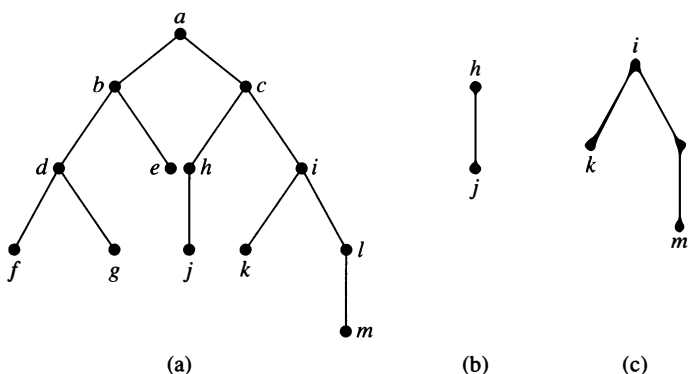


FIGURE 8 A Binary Tree T and Left and Right Subtrees of the Vertex c .

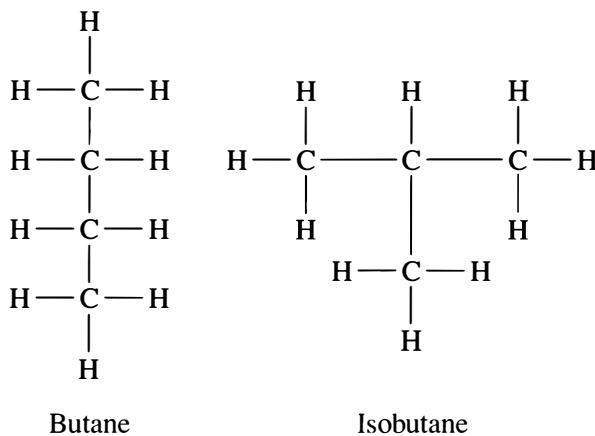


FIGURE 9 The Two Isomers of Butane.

Just as in the case of graphs, there is no standard terminology used to describe trees, rooted trees, ordered rooted trees, and binary trees. This nonstandard terminology occurs because trees are used extensively throughout computer science, which is a relatively young field. The reader should carefully check meanings given to terms dealing with trees whenever they occur.

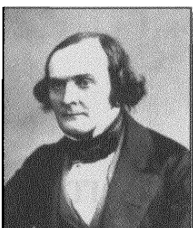
Trees as Models

Trees are used as models in such diverse areas as computer science, chemistry, geology, botany, and psychology. We will describe a variety of such models based on trees.

EXAMPLE 5 Saturated Hydrocarbons and Trees Graphs can be used to represent molecules, where atoms are represented by vertices and bonds between them by edges. The English mathematician Arthur Cayley discovered trees in 1857 when he was trying to enumerate the isomers of compounds of the form C_nH_{2n+2} , which are called *saturated hydrocarbons*.

In graph models of saturated hydrocarbons, each carbon atom is represented by a vertex of degree 4, and each hydrogen atom is represented by a vertex of degree 1. There are $3n + 2$ vertices in a graph representing a compound of the form C_nH_{2n+2} . The number of edges in such a graph is half the sum of the degrees of the vertices. Hence, there are $(4n + 2n + 2)/2 = 3n + 1$ edges in this graph. Because the graph is connected and the number of edges is one less than the number of vertices, it must be a tree (see Exercise 15 at the end of this section).

The nonisomorphic trees with n vertices of degree 4 and $2n + 2$ of degree 1 represent the different isomers of C_nH_{2n+2} . For instance, when $n = 4$, there are exactly two nonisomorphic trees of this type (the reader should verify this). Hence, there are exactly two different isomers of C_4H_{10} . Their structures are displayed in Figure 9. These two isomers are called butane and isobutane. ◀



ARTHUR CAYLEY (1821–1895) Arthur Cayley, the son of a merchant, displayed his mathematical talents at an early age with amazing skill in numerical calculations. Cayley entered Trinity College, Cambridge, when he was 17. While in college he developed a passion for reading novels. Cayley excelled at Cambridge and was elected to a 3-year appointment as Fellow of Trinity and assistant tutor. During this time Cayley began his study of n -dimensional geometry and made a variety of contributions to geometry and to analysis. He also developed an interest in mountaineering, which he enjoyed during vacations in Switzerland. Because no position as a mathematician was available to him, Cayley left Cambridge, entering the legal profession and gaining admittance to the bar in 1849. Although Cayley limited his legal work to be able to continue his mathematics research, he developed a reputation as a legal specialist. During his legal career he was able

write more than 300 mathematical papers. In 1863 Cambridge University established a new post in mathematics and offered it to Cayley. He took this job, even though it paid less money than he made as a lawyer.

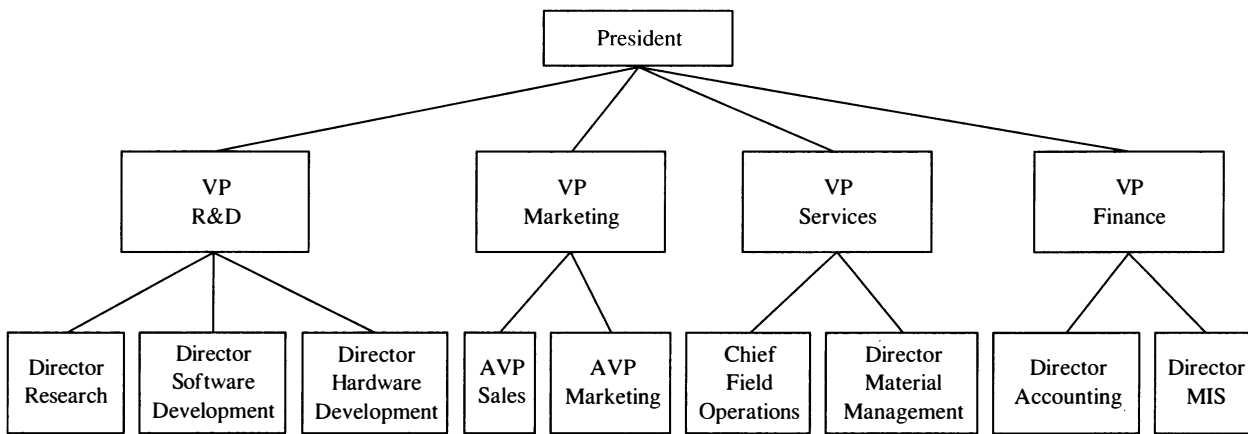


FIGURE 10 An Organizational Tree for a Computer Company.

EXAMPLE 6 Representing Organizations The structure of a large organization can be modeled using a rooted tree. Each vertex in this tree represents a position in the organization. An edge from one vertex to another indicates that the person represented by the initial vertex is the (direct) boss of the person represented by the terminal vertex. The graph shown in Figure 10 displays such a tree. In the organization represented by this tree, the Director of Hardware Development works directly for the Vice President of R&D. ◀

EXAMPLE 7 Computer File Systems Files in computer memory can be organized into directories. A directory can contain both files and subdirectories. The root directory contains the entire file system. Thus, a file system may be represented by a rooted tree, where the root represents the root directory, internal vertices represent subdirectories, and leaves represent ordinary files or empty directories. One such file system is shown in Figure 11. In this system, the file *chr* is in the directory *rje*. (Note that links to files where the same file may have more than one pathname can lead to circuits in computer file systems.) ◀

EXAMPLE 8 Tree-Connected Parallel Processors In Example 16 of Section 9.2 we described several interconnection networks for parallel processing. A **tree-connected network** is another important

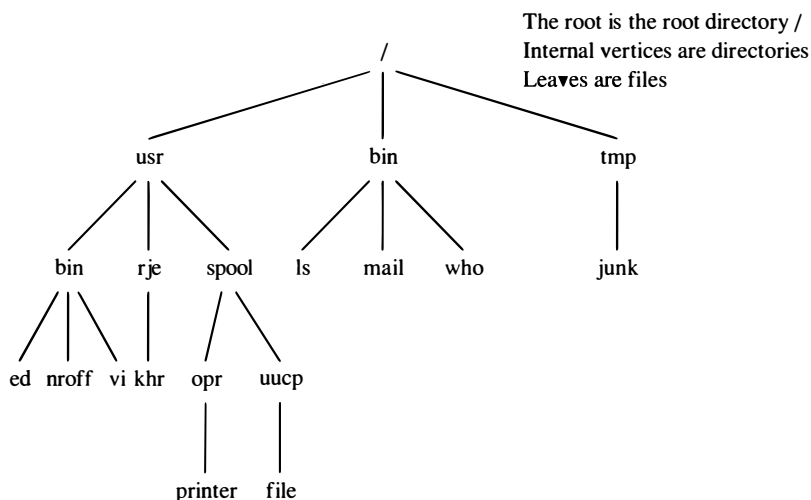


FIGURE 11 A Computer File System.

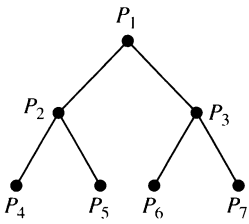


FIGURE 12 A Tree-Connected Network of Seven Processors.

way to interconnect processors. The graph representing such a network is a complete binary tree. Such a network interconnects $n = 2^k - 1$ processors, where k is a positive integer. A processor represented by the vertex v that is not a root or a leaf has three two-way connections—one to the processor represented by the parent of v and two to the processors represented by the two children of v . The processor represented by the root has two two-way connections to the processors represented by its two children. A processor represented by a leaf v has a single two-way connection to the parent of v . We display a tree-connected network with seven processors in Figure 12.

We will illustrate how a tree-connected network can be used for parallel computation. In particular, we will show how the processors in Figure 12 can be used to add eight numbers, using three steps. In the first step, we add x_1 and x_2 using P_4 , x_3 and x_4 using P_5 , x_5 and x_6 using P_6 , and x_7 and x_8 using P_7 . In the second step, we add $x_1 + x_2$ and $x_3 + x_4$ using P_2 and $x_5 + x_6$ and $x_7 + x_8$ using P_3 . Finally, in the third step, we add $x_1 + x_2 + x_3 + x_4$ and $x_5 + x_6 + x_7 + x_8$ using P_1 . The three steps used to add eight numbers compares favorably to the seven steps required to add eight numbers serially, where the steps are the addition of one number to the sum of the previous numbers in the list. ◀

Properties of Trees

We will often need results relating the numbers of edges and vertices of various types in trees.

THEOREM 2 A tree with n vertices has $n - 1$ edges.



Proof: We will use mathematical induction to prove this theorem. Note that for all the trees here we can choose a root and consider the tree rooted.

BASIS STEP: When $n = 1$, a tree with $n = 1$ vertex has no edges. It follows that the theorem is true for $n = 1$.

INDUCTIVE STEP: The induction hypothesis states that every tree with k vertices has $k - 1$ edges, where k is a positive integer. Suppose that a tree T has $k + 1$ vertices and that v is a leaf of T (which must exist because the tree is finite), and let w be the parent of v . Removing from T the vertex v and the edge connecting w to v produces a tree T' with k vertices, because the resulting graph is still connected and has no simple circuits. By the induction hypothesis, T' has $k - 1$ edges. It follows that T has k edges because it has one more edge than T' , the edge connecting v and w . This completes the induction step. ◀

The number of vertices in a full m -ary tree with a specified number of internal vertices is determined, as Theorem 3 shows. As in Theorem 2, we will use n to denote the number of vertices in a tree.

THEOREM 3 A full m -ary tree with i internal vertices contains $n = mi + 1$ vertices.

Proof: Every vertex, except the root, is the child of an internal vertex. Because each of the i internal vertices has m children, there are mi vertices in the tree other than the root. Therefore, the tree contains $n = mi + 1$ vertices. ◀

Suppose that T is a full m -ary tree. Let i be the number of internal vertices and l the number of leaves in this tree. Once one of n , i , and l is known, the other two quantities are determined. Theorem 4 explains how to find the other two quantities from the one that is known.

THEOREM 4 A full m -ary tree with

- (i) n vertices has $i = (n - 1)/m$ internal vertices and $l = [(m - 1)n + 1]/m$ leaves,
- (ii) i internal vertices has $n = mi + 1$ vertices and $l = (m - 1)i + 1$ leaves,
- (iii) l leaves has $n = (ml - 1)/(m - 1)$ vertices and $i = (l - 1)/(m - 1)$ internal vertices.

Proof: Let n represent the number of vertices, i the number of internal vertices, and l the number of leaves. The three parts of the theorem can all be proved using the equality given in Theorem 3, that is, $n = mi + 1$, together with the equality $n = l + i$, which is true because each vertex is either a leaf or an internal vertex. We will prove part (i) here. The proofs of parts (ii) and (iii) are left as exercises for the reader.

Solving for i in $n = mi + 1$ gives $i = (n - 1)/m$. Then inserting this expression for i into the equation $n = l + i$ shows that $l = n - i = n - (n - 1)/m = [(m - 1)n + 1]/m$. ◀

Example 9 illustrates how Theorem 4 can be used.

EXAMPLE 9 Suppose that someone starts a chain letter. Each person who receives the letter is asked to send it on to four other people. Some people do this, but others do not send any letters. How many people have seen the letter, including the first person, if no one receives more than one letter and if the chain letter ends after there have been 100 people who read it but did not send it out? How many people sent out the letter?

Solution: The chain letter can be represented using a 4-ary tree. The internal vertices correspond to people who sent out the letter, and the leaves correspond to people who did not send it out. Because 100 people did not send out the letter, the number of leaves in this rooted tree is $l = 100$. Hence, part (iii) of Theorem 4 shows that the number of people who have seen the letter is $n = (4 \cdot 100 - 1)/(4 - 1) = 133$. Also, the number of internal vertices is $133 - 100 = 33$, so 33 people sent out the letter. ◀

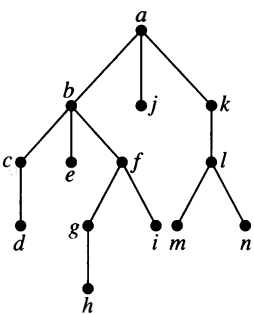


FIGURE 13 A Rooted Tree.

It is often desirable to use rooted trees that are “balanced” so that the subtrees at each vertex contain paths of approximately the same length. Some definitions will make this concept clear. The **level** of a vertex v in a rooted tree is the length of the unique path from the root to this vertex. The level of the root is defined to be zero. The **height** of a rooted tree is the maximum of the levels of vertices. In other words, the height of a rooted tree is the length of the longest path from the root to any vertex.

EXAMPLE 10 Find the level of each vertex in the rooted tree shown in Figure 13. What is the height of this tree?

Solution: The root a is at level 0. Vertices b , j , and k are at level 1. Vertices c , e , f , and l are at level 2. Vertices d , g , i , m , and n are at level 3. Finally, vertex h is at level 4. Because the largest level of any vertex is 4, this tree has height 4. ◀

A rooted m -ary tree of height h is **balanced** if all leaves are at levels h or $h - 1$.

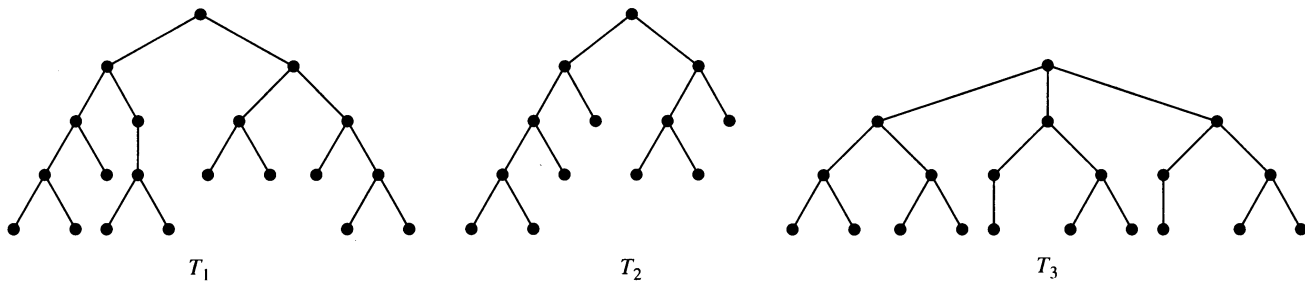


FIGURE 14 Some Rooted Trees.

EXAMPLE 11 Which of the rooted trees shown in Figure 14 are balanced?

Solution: T_1 is balanced, because all its leaves are at levels 3 and 4. However, T_2 is not balanced, because it has leaves at levels 2, 3, and 4. Finally, T_3 is balanced, because all its leaves are at level 3. ◀

The results in Theorem 5 relate the height and the number of leaves in m -ary trees.

THEOREM 5 There are at most m^h leaves in an m -ary tree of height h .

Proof: The proof uses mathematical induction on the height. First, consider m -ary trees of height 1. These trees consist of a root with no more than m children, each of which is a leaf. Hence, there are no more than $m^1 = m$ leaves in an m -ary tree of height 1. This is the basis step of the inductive argument.

Now assume that the result is true for all m -ary trees of height less than h ; this is the inductive hypothesis. Let T be an m -ary tree of height h . The leaves of T are the leaves of the subtrees of T obtained by deleting the edges from the root to each of the vertices at level 1, as shown in Figure 15.

Each of these subtrees has height less than or equal to $h - 1$. So by the inductive hypothesis, each of these rooted trees has at most m^{h-1} leaves. Because there are at most m such subtrees, each with a maximum of m^{h-1} leaves, there are at most $m \cdot m^{h-1} = m^h$ leaves in the rooted tree. This finishes the inductive argument. ◀

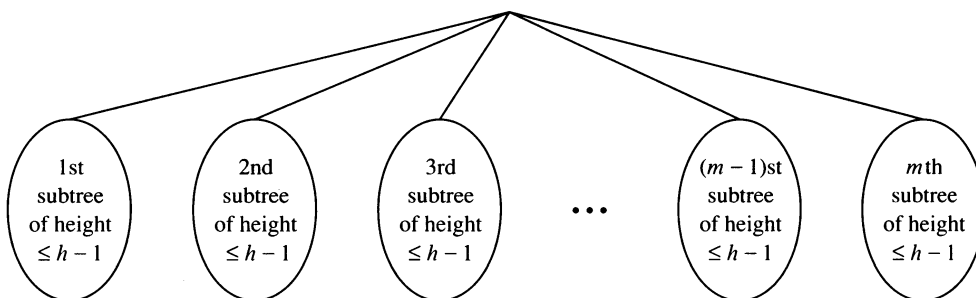


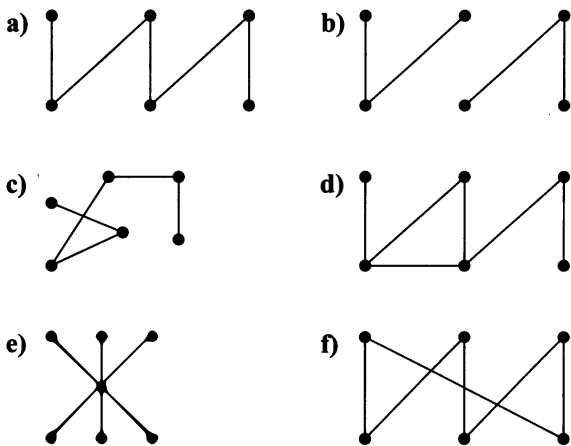
FIGURE 15 The Inductive Step of the Proof.

COROLLARY 1 If an m -ary tree of height h has l leaves, then $h \geq \lceil \log_m l \rceil$. If the m -ary tree is full and balanced, then $h = \lceil \log_m l \rceil$. (We are using the ceiling function here. Recall that $\lceil x \rceil$ is the smallest integer greater than or equal to x .)

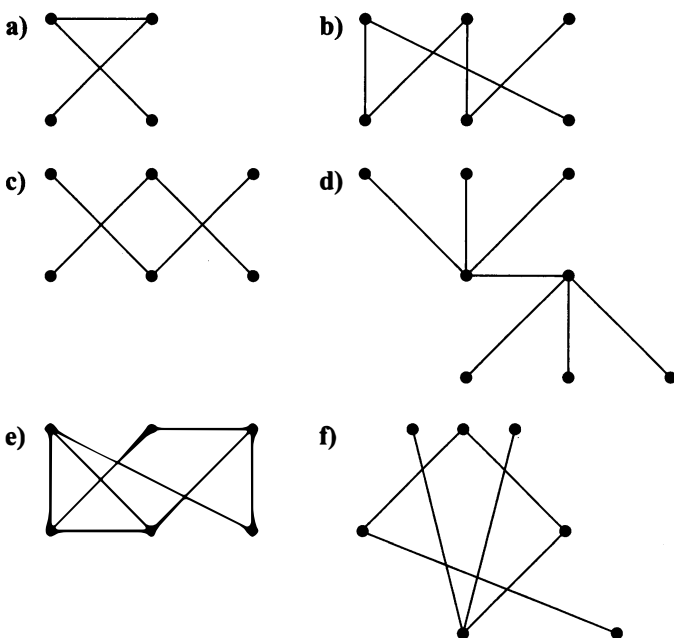
Proof: We know that $l \leq m^h$ from Theorem 5. Taking logarithms to the base m shows that $\log_m l \leq h$. Because h is an integer, we have $h \geq \lceil \log_m l \rceil$. Now suppose that the tree is balanced. Then each leaf is at level h or $h - 1$, and because the height is h , there is at least one leaf at level h . It follows that there must be more than m^{h-1} leaves (see Exercise 30 at the end of this section). Because $l \leq m^h$, we have $m^{h-1} < l \leq m^h$. Taking logarithms to the base m in this inequality gives $h - 1 < \log_m l \leq h$. Hence, $h = \lceil \log_m l \rceil$. \triangleleft

Exercises

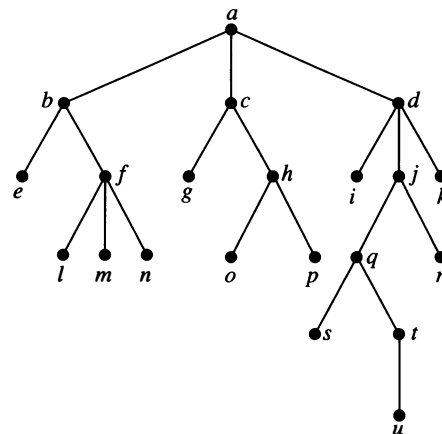
1. Which of these graphs are trees?



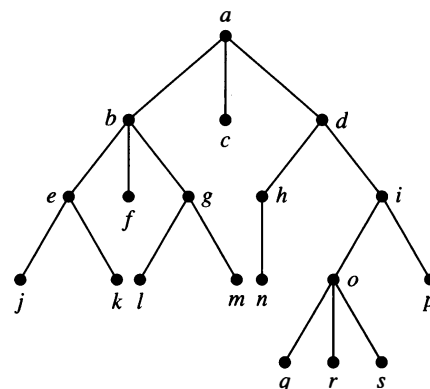
2. Which of these graphs are trees?



3. Answer these questions about the rooted tree illustrated.



- Which vertex is the root?
 - Which vertices are internal?
 - Which vertices are leaves?
 - Which vertices are children of j ?
 - Which vertex is the parent of h ?
 - Which vertices are siblings of o ?
 - Which vertices are ancestors of m ?
 - Which vertices are descendants of b ?
4. Answer the same questions as listed in Exercise 3 for the rooted tree illustrated.

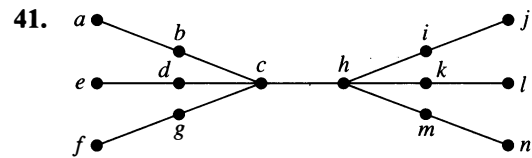
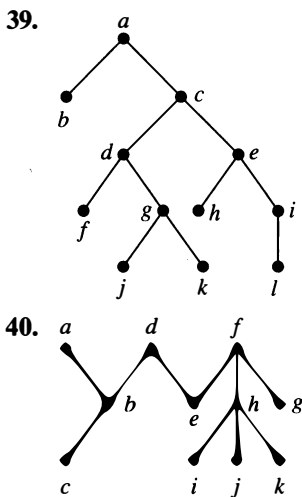


5. Is the rooted tree in Exercise 3 a full m -ary tree for some positive integer m ?
6. Is the rooted tree in Exercise 4 a full m -ary tree for some positive integer m ?
7. What is the level of each vertex of the rooted tree in Exercise 3?
8. What is the level of each vertex of the rooted tree in Exercise 4?
9. Draw the subtree of the tree in Exercise 3 that is rooted at
 - a) a .
 - b) c .
 - c) e .
10. Draw the subtree of the tree in Exercise 4 that is rooted at
 - a) a .
 - b) c .
 - c) e .
11.
 - a) How many nonisomorphic unrooted trees are there with three vertices?
 - b) How many nonisomorphic rooted trees are there with three vertices (using isomorphism for directed graphs)?
- *12.
 - a) How many nonisomorphic unrooted trees are there with four vertices?
 - b) How many nonisomorphic rooted trees are there with four vertices (using isomorphism for directed graphs)?
- *13.
 - a) How many nonisomorphic unrooted trees are there with five vertices?
 - b) How many nonisomorphic rooted trees are there with five vertices (using isomorphism for directed graphs)?
- *14. Show that a simple graph is a tree if and only if it is connected, but the deletion of any of its edges produces a graph that is not connected.
- *15. Let G be a simple graph with n vertices. Show that G is a tree if and only if G is connected and has $n - 1$ edges. [Hint: For the "if" part, use Exercise 14 and Theorem 2.]
16. Which complete bipartite graphs $K_{m,n}$, where m and n are positive integers, are trees?
17. How many edges does a tree with 10,000 vertices have?
18. How many vertices does a full 5-ary tree with 100 internal vertices have?
19. How many edges does a full binary tree with 1000 internal vertices have?
20. How many leaves does a full 3-ary tree with 100 vertices have?
21. Suppose 1000 people enter a chess tournament. Use a rooted tree model of the tournament to determine how many games must be played to determine a champion, if a player is eliminated after one loss and games are played until only one entrant has not lost. (Assume there are no ties.)
22. A chain letter starts when a person sends a letter to five others. Each person who receives the letter either sends it to five other people who have never received it or does not send it to anyone. Suppose that 10,000 people send out the letter before the chain ends and that no one receives more than one letter. How many people receive the letter, and how many do not send it out?
23. A chain letter starts with a person sending a letter out to 10 others. Each person is asked to send the letter out to 10 others, and each letter contains a list of the previous six people in the chain. Unless there are fewer than six names in the list, each person sends one dollar to the first person in this list, removes the name of this person from the list, moves up each of the other five names one position, and inserts his or her name at the end of this list. If no person breaks the chain and no one receives more than one letter, how much money will a person in the chain ultimately receive?
- *24. Either draw a full m -ary tree with 76 leaves and height 3, where m is a positive integer, or show that no such tree exists.
- *25. Either draw a full m -ary tree with 84 leaves and height 3, where m is a positive integer, or show that no such tree exists.
- *26. A full m -ary tree T has 81 leaves and height 4.
 - a) Give the upper and lower bounds for m .
 - b) What is m if T is also balanced?

A **complete m -ary tree** is a full m -ary tree, where every leaf is at the same level.
27. Construct a complete binary tree of height 4 and a complete 3-ary tree of height 3.
28. How many vertices and how many leaves does a complete m -ary tree of height h have?
29. Prove
 - a) part (ii) of Theorem 4.
 - b) part (iii) of Theorem 4.
- ☞ 30. Show that a full m -ary balanced tree of height h has more than m^{h-1} leaves.
31. How many edges are there in a forest of t trees containing a total of n vertices?
32. Explain how a tree can be used to represent the table of contents of a book organized into chapters, where each chapter is organized into sections, and each section is organized into subsections.
33. How many different isomers do these saturated hydrocarbons have?
 - a) C_3H_8
 - b) C_5H_{12}
 - c) C_6H_{14}
34. What does each of these represent in an organizational tree?
 - a) the parent of a vertex
 - b) a child of a vertex
 - c) a sibling of a vertex
 - d) the ancestors of a vertex
 - e) the descendants of a vertex
 - f) the level of a vertex
 - g) the height of the tree
35. Answer the same questions as those given in Exercise 34 for a rooted tree representing a computer file system.
36.
 - a) Draw the complete binary tree with 15 vertices that represents a tree-connected network of 15 processors.

- b) Show how 16 numbers can be added using the 15 processors in part (a) using four steps.
37. Let n be a power of 2. Show that n numbers can be added in $\log n$ steps using a tree-connected network of $n - 1$ processors.
- *38. A **labeled tree** is a tree where each vertex is assigned a label. Two labeled trees are considered isomorphic when there is an isomorphism between them that preserves the labels of vertices. How many nonisomorphic trees are there with three vertices labeled with different integers from the set $\{1, 2, 3\}$? How many nonisomorphic trees are there with four vertices labeled with different integers from the set $\{1, 2, 3, 4\}$?

The **eccentricity** of a vertex in an unrooted tree is the length of the longest simple path beginning at this vertex. A vertex is called a **center** if no vertex in the tree has smaller eccentricity than this vertex. In Exercises 39–41 find every vertex that is a center in the given tree.



42. Show that a center should be chosen as the root to produce a rooted tree of minimal height from an unrooted tree.
- *43. Show that a tree has either one center or two centers that are adjacent.
44. Show that every tree can be colored using two colors.

The **rooted Fibonacci trees** T_n are defined recursively in the following way. T_1 and T_2 are both the rooted tree consisting of a single vertex, and for $n = 3, 4, \dots$, the rooted tree T_n is constructed from a root with T_{n-1} as its left subtree and T_{n-2} as its right subtree.

45. Draw the first seven rooted Fibonacci trees.
- *46. How many vertices, leaves, and internal vertices does the rooted Fibonacci tree T_n have, where n is a positive integer? What is its height?
47. What is wrong with the following “proof” using mathematical induction of the statement that every tree with n vertices has a path of length $n - 1$. *Basis step:* Every tree with one vertex clearly has a path of length 0. *Inductive step:* Assume that a tree with n vertices has a path of length $n - 1$, which has u as its terminal vertex. Add a vertex v and the edge from u to v . The resulting tree has $n + 1$ vertices and has a path of length n . This completes the induction step.

- *48. Show that the average depth of a leaf in a binary tree with n vertices is $\Omega(\log n)$.

10.2 Applications of Trees

Introduction

We will discuss three problems that can be studied using trees. The first problem is: How should items in a list be stored so that an item can be easily located? The second problem is: What series of decisions should be made to find an object with a certain property in a collection of objects of a certain type? The third problem is: How should a set of characters be efficiently coded by bit strings?

Binary Search Trees



Searching for items in a list is one of the most important tasks that arises in computer science. Our primary goal is to implement a searching algorithm that finds items efficiently when the

items are totally ordered. This can be accomplished through the use of a **binary search tree**, which is a binary tree in which each child of a vertex is designated as a right or left child, no vertex has more than one right child or left child, and each vertex is labeled with a key, which is one of the items. Furthermore, vertices are assigned keys so that the key of a vertex is both larger than the keys of all vertices in its left subtree and smaller than the keys of all vertices in its right subtree.

This recursive procedure is used to form the binary search tree for a list of items. Start with a tree containing just one vertex, namely, the root. The first item in the list is assigned as the key of the root. To add a new item, first compare it with the keys of vertices already in the tree, starting at the root and moving to the left if the item is less than the key of the respective vertex if this vertex has a left child, or moving to the right if the item is greater than the key of the respective vertex if this vertex has a right child. When the item is less than the respective vertex and this vertex has no left child, then a new vertex with this item as its key is inserted as a new left child. Similarly, when the item is greater than the respective vertex and this vertex has no right child, then a new vertex with this item as its key is inserted as a new right child. We illustrate this procedure with Example 1.

EXAMPLE 1 Form a binary search tree for the words *mathematics*, *physics*, *geography*, *zoology*, *meteorology*, *geology*, *psychology*, and *chemistry* (using alphabetical order).

Solution: Figure 1 displays the steps used to construct this binary search tree. The word *mathematics* is the key of the root. Because *physics* comes after *mathematics* (in alphabetical order), add a right child of the root with key *physics*. Because *geography* comes before *mathematics*, add a left child of the root with key *geography*. Next, add a right child of the vertex with key *physics*, and assign it the key *zoology*, because *zoology* comes after *mathematics* and after *physics*. Similarly, add a left child of the vertex with key *physics* and assign this new vertex the key *meteorology*. Add a right child of the vertex with key *geography* and assign this new vertex the key *geology*. Add a left child of the vertex with key *zoology* and assign it the key *psychology*.

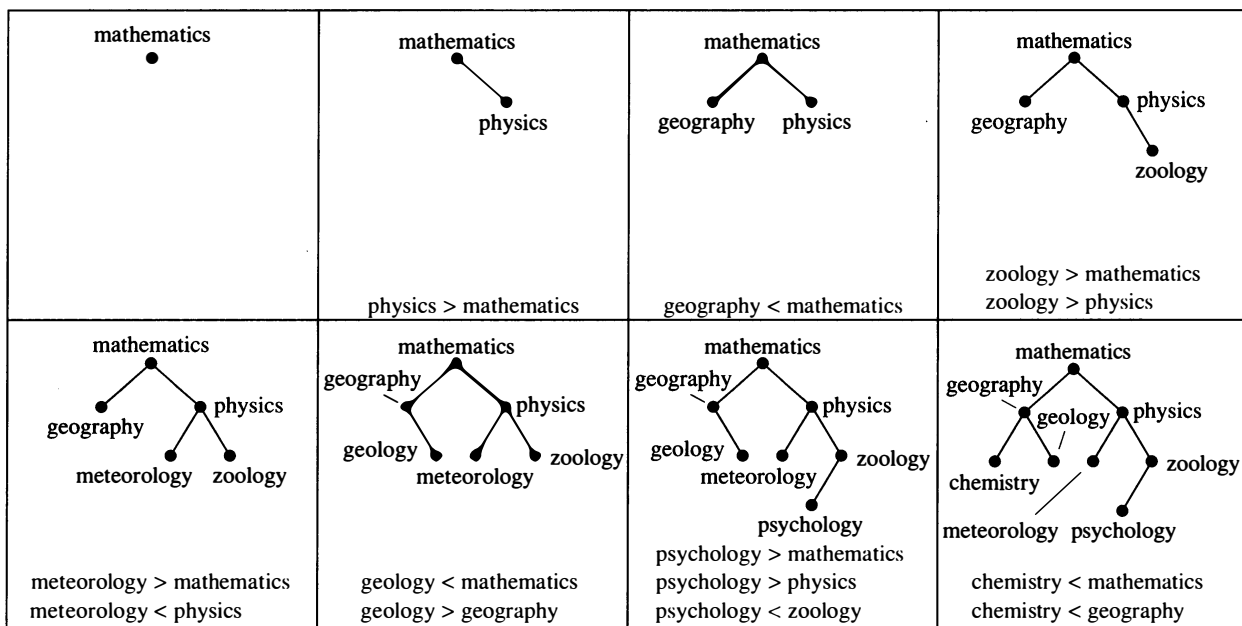


FIGURE 1 Constructing a Binary Search Tree.

Add a left child of the vertex with key *geography* and assign it the key *chemistry*. (The reader should work through all the comparisons needed at each step.) ◀

Once we have a binary search tree, we need a way to locate items in the binary search tree, as well as a way to add new items. Algorithm 1, an insertion algorithm, actually does both of these tasks, even though it may appear that it is only designed to add vertices to a binary search tree. That is, Algorithm 1 is a procedure that locates an item x in a binary search tree if it is present, and adds a new vertex with x as its key if x is not present. In the pseudocode, v is the vertex currently under examination and $label(v)$ represents the key of this vertex. The algorithm begins by examining the root. If the x equals the key of v , then the algorithm has found the location of x and terminates; if x is less than the key of v , we move to the left child of v and repeat the procedure; and if x is greater than the key of v , we move to the right child of v and repeat the procedure. If at any step we attempt to move to a child that is not present, we know that x is not present in the tree, and we add a new vertex as this child with x as its key.

ALGORITHM 1 Locating and Adding Items to a Binary Search Tree.

```

procedure insertion( $T$ : binary search tree,  $x$ : item)
 $v :=$  root of  $T$ 
{a vertex not present in  $T$  has the value null }
while  $v \neq null$  and  $label(v) \neq x$ 
begin
  if  $x < label(v)$  then
    if left child of  $v \neq null$  then  $v :=$  left child of  $v$ 
    else add new vertex as a left child of  $v$  and set  $v := null$ 
  else
    if right child of  $v \neq null$  then  $v :=$  right child of  $v$ 
    else add new vertex as a right child of  $v$  to  $T$  and set  $v := null$ 
end
if root of  $T = null$  then add a vertex  $v$  to the tree and label it with  $x$ 
else if  $v$  is null or  $label(v) \neq x$  then label new vertex with  $x$  and let  $v$  be this new vertex
{ $v =$  location of  $x$ }

```

Example 2 illustrates the use of Algorithm 1 to insert a new item into a binary search tree.

EXAMPLE 2 Use Algorithm 1 to insert the word *oceanography* into the binary search tree in Example 1.

Solution: Algorithm 1 begins with v , the vertex under examination, equal to the root of T , so $label(v) = mathematics$. Because $v \neq null$ and $label(v) = mathematics < oceanography$, we next examine the right child of the root. This right child exists, so we set v , the vertex under examination, to be this right child. At this step we have $v \neq null$ and $label(v) = physics > oceanography$, so we examine the left child of v . This left child exists, so we set v , the vertex under examination, to this left child. At this step, we also have $v \neq null$ and $label(v) = meteorology < oceanography$, so we try to examine the right child of v . However, this right child does not exist, so we add a new vertex as the right child of v (which at this point is the vertex with the key *meteorology*) and we set $v := null$. We now exit the **while** loop because $v = null$. Because the root of T is not *null* and $v = null$, we use the **else if** statement at the end of the algorithm to label our new vertex with the key *oceanography*. ◀

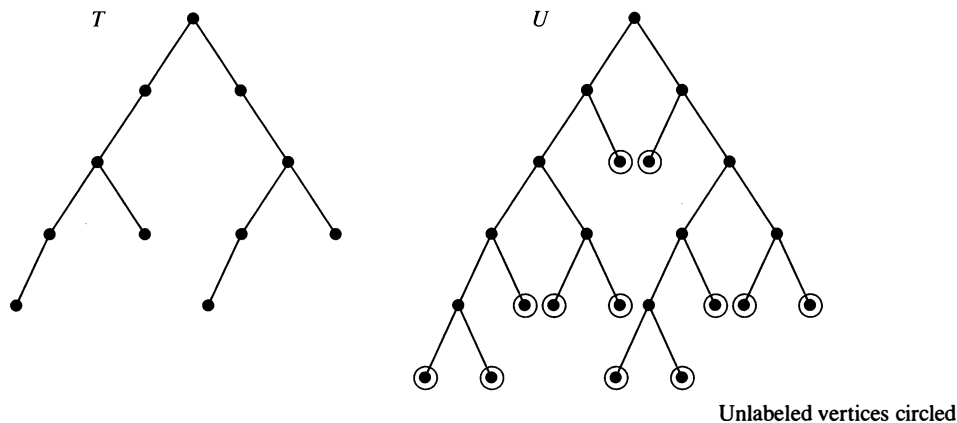


FIGURE 2 Adding Unlabeled Vertices to Make a Binary Search Tree Full.

We will now determine the computational complexity of this procedure. Suppose we have a binary search tree T for a list of n items. We can form a full binary tree U from T by adding unlabeled vertices whenever necessary so that every vertex with a key has two children. This is illustrated in Figure 2. Once we have done this, we can easily locate or add a new item as a key without adding a vertex.

The most comparisons needed to add a new item is the length of the longest path in U from the root to a leaf. The internal vertices of U are the vertices of T . It follows that U has n internal vertices. We can now use part (ii) of Theorem 4 in Section 10.1 to conclude that U has $n + 1$ leaves. Using Corollary 1 of Section 10.1, we see that the height of U is greater than or equal to $h = \lceil \log(n + 1) \rceil$. Consequently, it is necessary to perform at least $\lceil \log(n + 1) \rceil$ comparisons to add some item. Note that if U is balanced, its height is $\lceil \log(n + 1) \rceil$ (by Corollary 1 of Section 10.1). Thus, if a binary search tree is balanced, locating or adding an item requires no more than $\lceil \log(n + 1) \rceil$ comparisons. A binary search tree can become unbalanced as items are added to it. Because balanced binary search trees give optimal worst-case complexity for binary searching, algorithms have been devised that rebalance binary search trees as items are added. The interested reader can consult references on data structures for the description of such algorithms.

Decision Trees



Rooted trees can be used to model problems in which a series of decisions leads to a solution. For instance, a binary search tree can be used to locate items based on a series of comparisons, where each comparison tells us whether we have located the item, or whether we should go right or left in a subtree. A rooted tree in which each internal vertex corresponds to a decision, with a subtree at these vertices for each possible outcome of the decision, is called a **decision tree**. The possible solutions of the problem correspond to the paths to the leaves of this rooted tree. Example 3 illustrates an application of decision trees.

EXAMPLE 3 Suppose there are seven coins, all with the same weight, and a counterfeit coin that weighs less than the others. How many weighings are necessary using a balance scale to determine which of the eight coins is the counterfeit one? Give an algorithm for finding this counterfeit coin.



Solution: There are three possibilities for each weighing on a balance scale. The two pans can have equal weight, the first pan can be heavier, or the second pan can be heavier. Consequently, the decision tree for the sequence of weighings is a 3-ary tree. There are at least eight leaves in

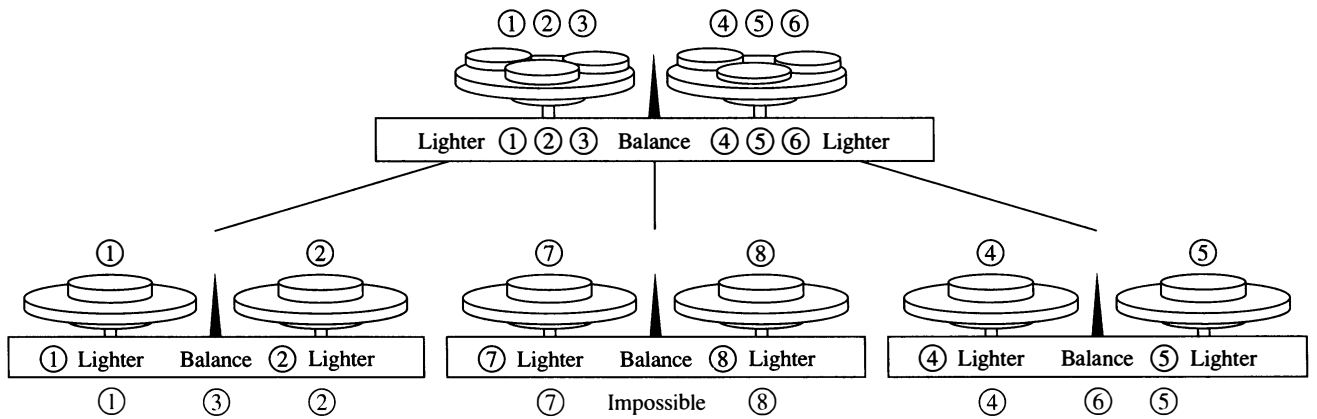


FIGURE 3 A Decision Tree for Locating a Counterfeit Coin. The counterfeit coin is shown in color below each final weighing.

the decision tree because there are eight possible outcomes (because each of the eight coins can be the counterfeit lighter coin), and each possible outcome must be represented by at least one leaf. The largest number of weighings needed to determine the counterfeit coin is the height of the decision tree. From Corollary 1 of Section 10.1 it follows that the height of the decision tree is at least $\lceil \log_3 8 \rceil = 2$. Hence, at least two weighings are needed.

It is possible to determine the counterfeit coin using two weighings. The decision tree that illustrates how this is done is shown in Figure 3. ◀

THE COMPLEXITY OF SORTING ALGORITHMS Many different sorting algorithms have been developed. To decide whether a particular sorting algorithm is efficient, its complexity is determined. Using decision trees as models, a lower bound for the worst-case complexity of sorting algorithms can be found.

We can use decision trees to model sorting algorithms and to determine an estimate for the worst-case complexity of these algorithms. Note that given n elements, there are $n!$ possible orderings of these elements, because each of the $n!$ permutations of these elements can be the correct order. The sorting algorithms studied in this book, and most commonly used sorting algorithms, are based on binary comparisons, that is, the comparison of two elements at a time. The result of each such comparison narrows down the set of possible orderings. Thus, a sorting algorithm based on binary comparisons can be represented by a binary decision tree in which each internal vertex represents a comparison of two elements. Each leaf represents one of the $n!$ permutations of n elements.

EXAMPLE 4 We display in Figure 4 a decision tree that orders the elements of the list a, b, c . ◀

The complexity of a sort based on binary comparisons is measured in terms of the number of such comparisons used. The largest number of binary comparisons ever needed to sort a list with n elements gives the worst-case performance of the algorithm. The most comparisons used equals the longest path length in the decision tree representing the sorting procedure. In other words, the largest number of comparisons ever needed is equal to the height of the decision tree. Because the height of a binary tree with $n!$ leaves is at least $\lceil \log n! \rceil$ (using Corollary 1 in Section 10.1), at least $\lceil \log n! \rceil$ comparisons are needed, as stated in Theorem 1.

THEOREM 1 A sorting algorithm based on binary comparisons requires at least $\lceil \log n! \rceil$ comparisons.

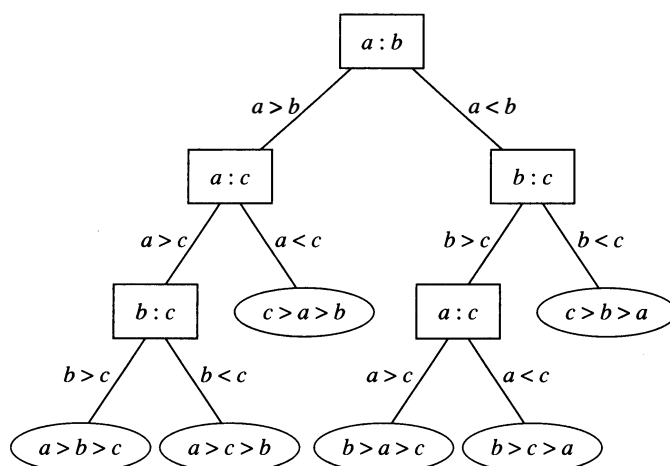


FIGURE 4 A Decision Tree for Sorting Three Distinct Elements.

We can use Theorem 1 to provide a big-Omega estimate for the number of comparisons used by a sorting algorithm based on binary comparison. We need only note that by Exercise 62 in Section 3.2 we know that $\lceil \log n! \rceil$ is $\Theta(n \log n)$, one of the commonly used reference functions for the computational complexity of algorithms. Corollary 1 is a consequence of this estimate.

COROLLARY 1 The number of comparisons used by a sorting algorithm to sort n elements based on binary comparisons is $\Omega(n \log n)$.

A consequence of Corollary 1 is that a sorting algorithm based on binary comparisons that uses $\Theta(n \log n)$ comparisons, in the worst case, to sort n elements is optimal, in the sense that no other such algorithm has better worst-case complexity. Note that by Theorem 1 in Section 4.4 we see that the merge sort algorithm is optimal in this sense.

We can also establish a similar result for the average-case complexity of sorting algorithms. The average number of comparisons used by a sorting algorithm based on binary comparisons is the average depth of a leaf in the decision tree representing the sorting algorithm. By Exercise 48 in Section 10.1 we know that the average depth of a leaf in a binary tree with N vertices is $\Omega(\log N)$. We obtain the following estimate when we let $N = n!$ and note that a function that is $\Omega(\log n!)$ is also $\Omega(n \log n)$ because $\log n!$ is $\Theta(n \log n)$.

THEOREM 2 The average number of comparisons used by a sorting algorithm to sort n elements based on binary comparisons is $\Omega(n \log n)$.

Prefix Codes

Consider the problem of using bit strings to encode the letters of the English alphabet (where no distinction is made between lowercase and uppercase letters). We can represent each letter with a bit string of length five, because there are only 26 letters and there are 32 bit strings of length five. The total number of bits used to encode data is five times the number of characters in the text when each character is encoded with five bits. Is it possible to find a coding scheme

of these letters such that, when data are coded, fewer bits are used? We can save memory and reduce transmittal time if this can be done.

Consider using bit strings of different lengths to encode letters. Letters that occur more frequently should be encoded using short bit strings, and longer bit strings should be used to encode rarely occurring letters. When letters are encoded using varying numbers of bits, some method must be used to determine where the bits for each character start and end. For instance, if e were encoded with 0, a with 1, and t with 01, then the bit string 0101 could correspond to eat , tea , $eaea$, or tt .

One way to ensure that no bit string corresponds to more than one sequence of letters is to encode letters so that the bit string for a letter never occurs as the first part of the bit string for another letter. Codes with this property are called **prefix codes**. For instance, the encoding of e as 0, a as 10, and t as 11 is a prefix code. A word can be recovered from the unique bit string that encodes its letters. For example, the string 10110 is the encoding of ate . To see this, note that the initial 1 does not represent a character, but 10 does represent a (and could not be the first part of the bit string of another letter). Then, the next 1 does not represent a character, but 11 does represent t . The final bit, 0, represents e .

A prefix code can be represented using a binary tree, where the characters are the labels of the leaves in the tree. The edges of the tree are labeled so that an edge leading to a left child is assigned a 0 and an edge leading to a right child is assigned a 1. The bit string used to encode a character is the sequence of labels of the edges in the unique path from the root to the leaf that has this character as its label. For instance, the tree in Figure 5 represents the encoding of e by 0, a by 10, t by 110, n by 1110, and s by 1111.

The tree representing a code can be used to decode a bit string. For instance, consider the word encoded by 11111011100 using the code in Figure 5. This bit string can be decoded by starting at the root, using the sequence of bits to form a path that stops when a leaf is reached. Each 0 bit takes the path down the edge leading to the left child of the last vertex in the path, and each 1 bit corresponds to the right child of this vertex. Consequently, the initial 1111 corresponds to the path starting at the root, going right four times, leading to a leaf in the graph that has s as its label, because the string 1111 is the code for s . Continuing with the fifth bit, we reach a leaf next after going right then left, when the vertex labeled with a , which is encoded by 10, is visited. Starting with the seventh bit, we reach a leaf next after going right three times and then left, when the vertex labeled with n , which is encoded by 1110, is visited. Finally, the last bit, 0, leads to the leaf that is labeled with e . Therefore, the original word is $sane$.

We can construct a prefix code from any binary tree where the left edge at each internal vertex is labeled by 0 and the right edge by a 1 and where the leaves are labeled by characters. Characters are encoded with the bit string constructed using the labels of the edges in the unique path from the root to the leaves.

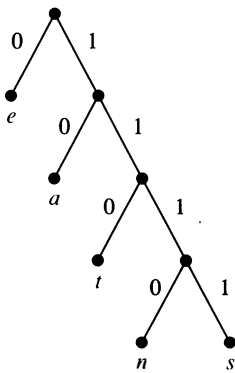


FIGURE 5 The Binary Tree with a Prefix Code.



HUFFMAN CODING We now introduce an algorithm that takes as input the frequencies (which are the probabilities of occurrences) of symbols in a string and produces as output a prefix code that encodes the string using the fewest possible bits, among all possible binary prefix codes for these symbols. This algorithm, known as **Huffman coding**, was developed by David Huffman in a term paper he wrote in 1951 while a graduate student at MIT. (Note that this algorithm assumes that we already know how many times each symbol occurs in the string, so we can compute the frequency of each symbol by dividing the number of times this symbol occurs by the length of the string.) Huffman coding is a fundamental algorithm in *data compression*, the subject devoted to reducing the number of bits required to represent information. Huffman coding is extensively used to compress bit strings representing text and it also plays an important role in compressing audio and image files.

Algorithm 2 presents the Huffman coding algorithm. Given symbols and their frequencies, our goal is to construct a rooted binary tree where the symbols are the labels of the leaves. The algorithm begins with a forest of trees each consisting of one vertex, where each vertex has a symbol as its label and where the weight of this vertex equals the frequency of the symbol that



is its label. At each step, we combine two trees having the least total weight into a single tree by introducing a new root and placing the tree with larger weight as its left subtree and the tree with smaller weight as its right subtree. Furthermore, we assign the sum of the weights of the two subtrees of this tree as the total weight of the tree. (Although procedures for breaking ties by choosing between trees with equal weights can be specified, we will not specify such procedures here.) The algorithm is finished when it has constructed a tree, that is, when the forest is reduced to a single tree.

ALGORITHM 2 Huffman Coding.

```

procedure Huffman(C: symbols  $a_i$  with frequencies  $w_i, i = 1, \dots, n$ )
   $F :=$  forest of  $n$  rooted trees, each consisting of the single vertex  $a_i$  and assigned weight  $w_i$ 
  while  $F$  is not a tree
  begin
    Replace the rooted trees  $T$  and  $T'$  of least weights from  $F$  with  $w(T) \geq w(T')$  with a tree
    having a new root that has  $T$  as its left subtree and  $T'$  as its right subtree. Label the new
    edge to  $T$  with 0 and the new edge to  $T'$  with 1.
    Assign  $w(T) + w(T')$  as the weight of the new tree.
  end
  {the Huffman coding for the symbol  $a_i$  is the concatenation of the labels of the edges in the
  unique path from the root to the vertex  $a_i$ }

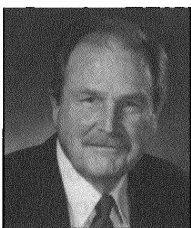
```

Example 5 illustrates how Algorithm 2 is used to encode a set of five symbols.

EXAMPLE 5


Use Huffman coding to encode the following symbols with the frequencies listed: A: 0.08, B: 0.10, C: 0.12, D: 0.15, E: 0.20, F: 0.35. What is the average number of bits used to encode a character?

Solution: Figure 6 displays the steps used to encode these symbols. The encoding produced encodes A by 111, B by 110, C by 011, D by 010, E by 10, and F by 00. The average number of bits used to encode a symbol using this encoding is $3 \cdot 0.08 + 3 \cdot 0.10 + 3 \cdot 0.12 + 3 \cdot 0.15 + 2 \cdot 0.20 + 2 \cdot 0.35 = 2.45$. ◀



DAVID A. HUFFMAN (1925–1999) David Huffman grew up in Ohio. At the age of 18 he received his B.S. in electrical engineering from The Ohio State University. Afterward he served in the U.S. Navy as a radar maintenance officer on a destroyer that had the mission of clearing mines in Asian waters after World War II. Later, he earned his M.S. from Ohio State and his Ph.D. in electrical engineering from MIT. Huffman joined the MIT faculty in 1953, where he remained until 1967 when he became the founding member of the computer science department at the University of California at Santa Cruz. He played an important role in developing this department and spent the remainder of his career there, retiring in 1994.

Huffman is noted for his contributions to information theory and coding, signal designs for radar and for communications, and design procedures for asynchronous logical circuits. His work on surfaces with zero curvature led him to develop original techniques for folding paper and vinyl into unusual shapes considered works of art by many and publicly displayed in several exhibits. However, Huffman is best known for his development of what is now called Huffman coding, a result of a term paper he wrote during his graduate work at MIT.

Huffman enjoyed exploring the outdoors, hiking, and traveling extensively. He became certified as a scuba diver when he was in his late 60s. He kept poisonous snakes as pets.

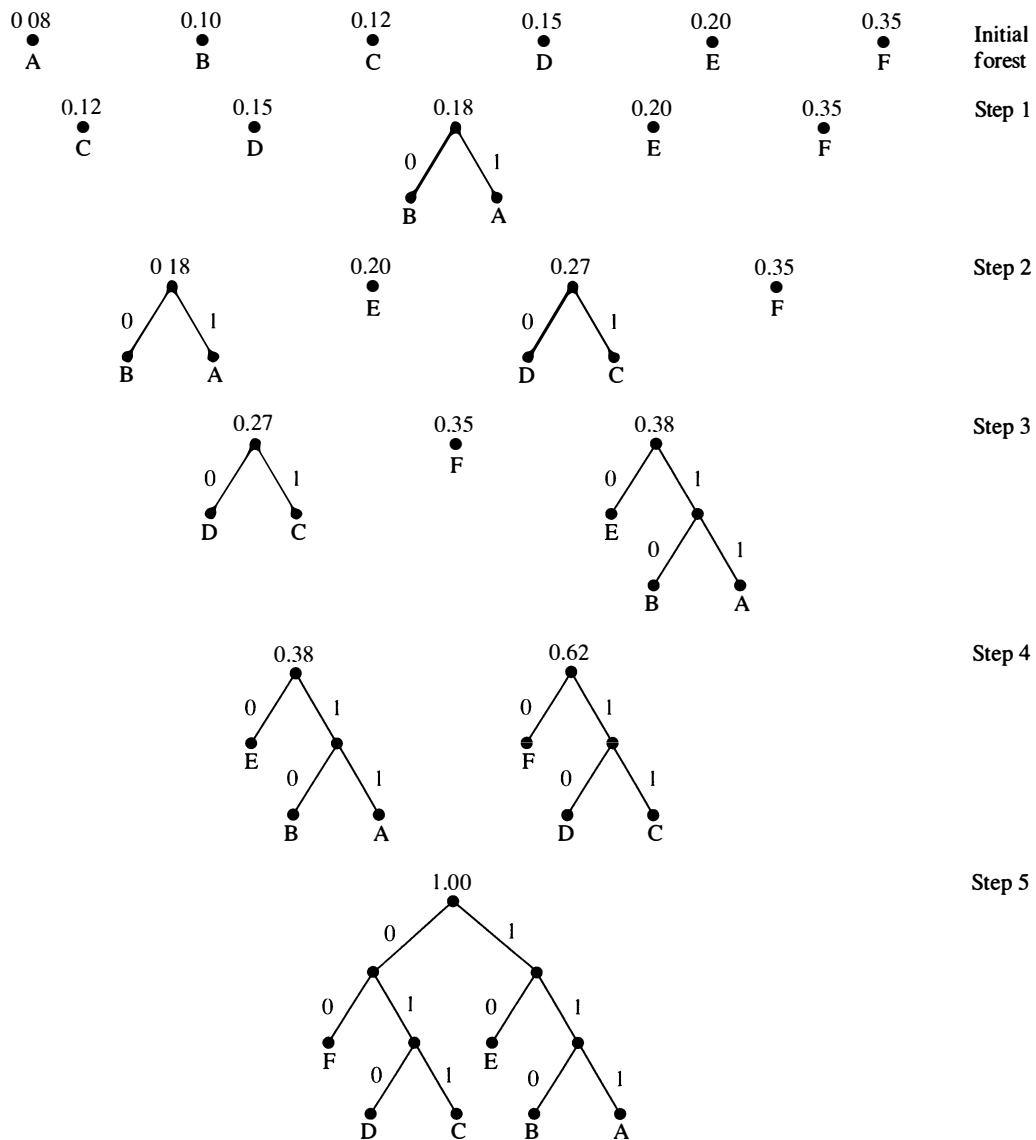


FIGURE 6 Huffman Coding of Symbols in Example 4.

Note that Huffman coding is a greedy algorithm. Replacing the two subtrees with the smallest weight at each step leads to an optimal code in the sense that no binary prefix code for these symbols can encode these symbols using fewer bits. We leave the proof that Huffman codes are optimal as Exercise 32 at the end of this section.

There are many variations of Huffman coding. For example, instead of encoding single symbols, we can encode blocks of symbols of a specified length, such as blocks of two symbols. Doing so may reduce the number of bits required to encode the string (see Exercise 30 at the end of this section). We can also use more than two symbols to encode the original symbols in the string (see the preamble to Exercise 28 at the end of this section). Furthermore, a variation known as adaptive Huffman coding (see [Sa00]) can be used when the frequency of each symbol in a string is not known in advance, so that encoding is done at the same time the string is being read.

Game Trees



Trees can be used to analyze certain types of games such as tic-tac-toe, nim, checkers, and chess. In each of these games, two players take turns making moves. Each player knows the moves made by the other player and no element of chance enters into the game. We model such games using **game trees**; the vertices of these trees represent the positions that a game can be in as it progresses; the edges represent legal moves between these positions. Because game trees are usually large, we simplify game trees by representing all symmetric positions of a game by the same vertex. However, the same position of a game may be represented by different vertices if different sequences of moves lead to this position. The root represents the starting position. The usual convention is to represent vertices at even levels by boxes and vertices at odd levels by circles. When the game is in a position represented by a vertex at an even level, it is the first player's move; when the game is in a position represented by a vertex at an odd level, it is the second player's move. Game trees may be infinite when the games they represent never end, such as games that can enter infinite loops, but for most games there are rules that lead to finite game trees.

The leaves of a game tree represent the final positions of a game. We assign a value to each leaf indicating the payoff to the first player if the game terminates in the position represented by this leaf. For games that are win-lose, we label a terminal vertex represented by a circle with a 1 to indicate a win by the first player and we label a terminal vertex represented by a box with a -1 to indicate a win by the second player. For games where draws are allowed, we label a terminal vertex corresponding to a draw position with a 0. Note that for win-lose games, we have assigned values to terminal vertices so that the larger the value, the better the outcome for the first player.

In Example 6 we display a game tree for a well-known and well-studied game.

EXAMPLE 6 Nim In a version of the game of **nim**, at the start of a game there are a number of piles of stones. Two players take turns making moves; a legal move consists of removing one or more stones from one of the piles, without removing all the stones left. A player without a legal move loses. (Another way to look at this is that the player removing the last stone loses because the position with no piles of stones is not allowed.) The game tree shown in Figure 7 represents

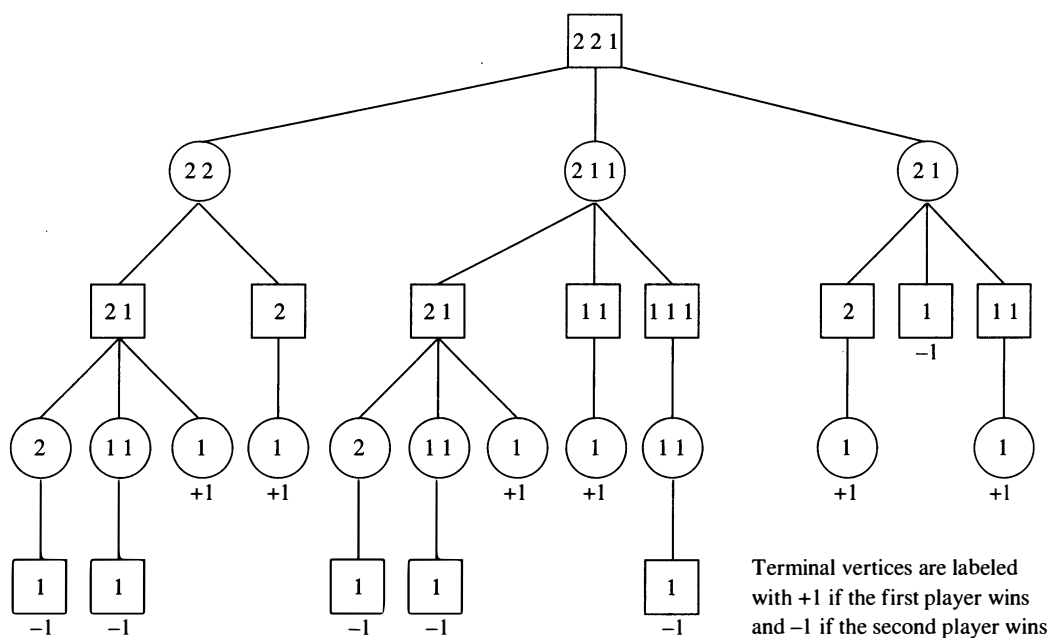


FIGURE 7 The Game Tree for a Game of Nim.

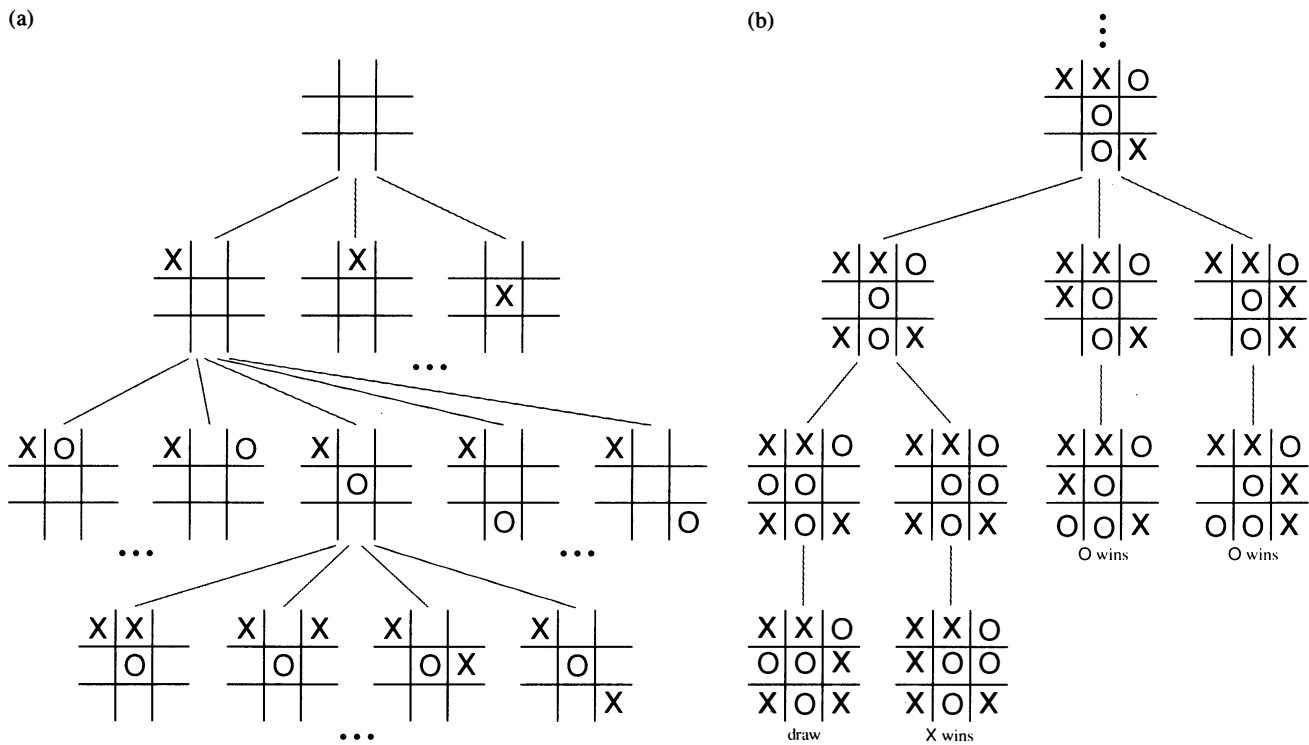


FIGURE 8 Some of the Game Tree for Tic-Tac-Toe.

this version of nim given the starting position where there are three piles of stones containing two, two, and one stone each, respectively. We represent each position with an unordered list of the number of stones in the different piles (the order of the piles does not matter). The initial move by the first player can lead to three possible positions because this player can remove one stone from a pile with two stones (leaving three piles containing one, one, and two stones); two stones from a pile containing two stones (leaving two piles containing two stones and one stone); or one stone from the pile containing one stone (leaving two piles of two stones). When only one pile with one stone is left, no legal moves are possible, so such positions are terminal positions. Because nim is a win-lose game, we label the terminal vertices with +1 when they represent wins for the first player and -1 when they represent wins for the second player. ◀

EXAMPLE 7 Tic-tac-toe The game tree for tic-tac-toe is extremely large and cannot be drawn here, although a computer could easily build such a tree. We show a portion of the game tic-tac-toe in Figure 8(a). Note that by considering symmetric positions equivalent, we need only consider three possible initial moves, as shown in Figure 8(a). We also show a subtree of this game tree leading to terminal positions in Figure 8(b), where a player who can win makes a winning move. ◀

We can recursively define the values of all vertices in a game tree in a way that enables us to determine the outcome of this game when both players follow optimal strategies. By a **strategy** we mean a set of rules that tells a player how to select moves to win the game. An optimal strategy for the first player is a strategy that maximizes the payoff to this player and for the second player is a strategy that minimizes this payoff. We now recursively define the value of a vertex.

DEFINITION 1 The *value of a vertex in a game tree* is defined recursively as:

- (i) the value of a leaf is the payoff to the first player when the game terminates in the position represented by this leaf.
- (ii) the value of an internal vertex at an even level is the maximum of the values of its children, and the value of an internal vertex at an odd level is the minimum of the values of its children.

The strategy where the first player moves to a position represented by a child with maximum value and the second player moves to a position of a child with minimum value is called the **minmax strategy**. We can determine who will win the game when both players follow the minmax strategy by calculating the value of the root of the tree; this value is called the **value** of the tree. This is a consequence of Theorem 3.

THEOREM 3 The value of a vertex of a game tree tells us the payoff to the first player if both players follow the minmax strategy and play starts from the position represented by this vertex.

Proof: We will use induction to prove this theorem.

BASIS STEP: If the vertex is a leaf, by definition the value assigned to this vertex is the payoff to the first player.

INDUCTIVE STEP: The inductive hypothesis is the assumption that the values of the children of a vertex are the payoffs to the first player, assuming that play starts at each of the positions represented by these vertices. We need to consider two cases, when it is the first player's turn and when it is the second player's turn.

When it is the first player's turn, this player follows the minmax strategy and moves to the position represented by the child with the largest value. By the inductive hypothesis, this value is the payoff to the first player when play starts at the position represented by this child and follows the minmax strategy. By the recursive step in the definition of the value of an internal vertex at an even level (as the maximum value of its children), the value of this vertex is the payoff when play begins at the position represented by this vertex.

When it is the second player's turn, this player follows the minmax strategy and moves to the position represented by the child with the least value. By the inductive hypothesis, this value is the payoff to the first player when play starts at the position represented by this child and both players follow the minmax strategy. By the recursive definition of the value of an internal vertex at an odd level as the minimum value of its children, the value of this vertex is the payoff when play begins at the position represented by this vertex. ◁

Remark: By extending the proof of Theorem 3, it can be shown that the minmax strategy is the optimal strategy for both players.

Example 8 illustrates how the minmax procedure works. It displays the values assigned to the internal vertices in the game tree from Example 6. Note that we can shorten the computation required by noting that for win–lose games, once a child of a square vertex with value +1 is found, the value of the square vertex is also +1 because +1 is the largest possible payoff.

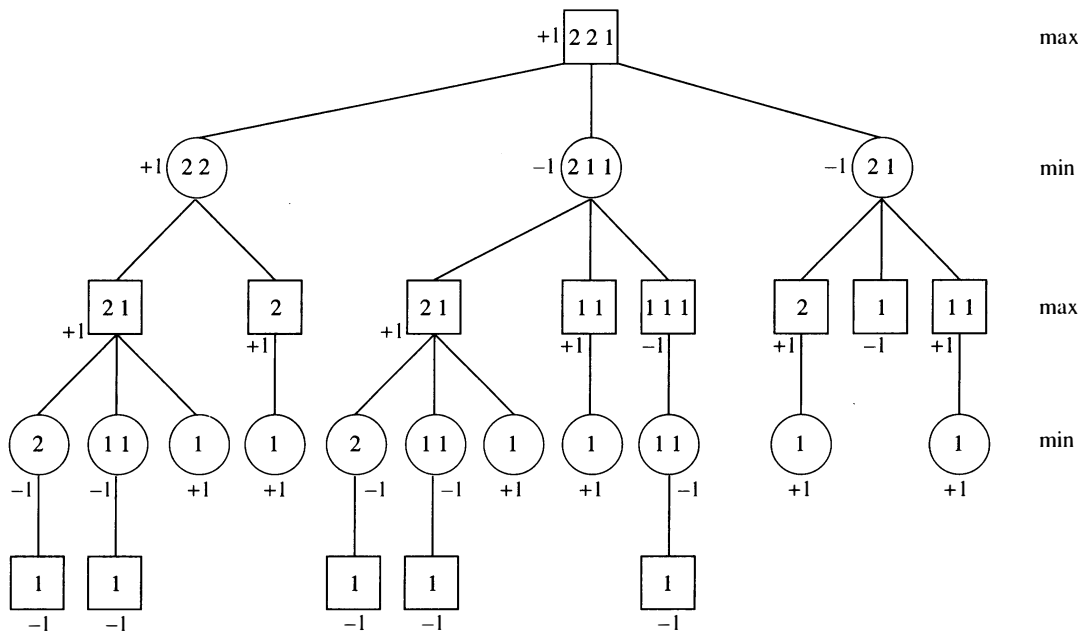


FIGURE 9 Showing the Values of Vertices in the Game of Nim.

Similarly, once a child of a circle vertex with value -1 is found, this is the value of the circle vertex also.

EXAMPLE 8 In Example 6 we constructed the game tree for nim with a starting position where there are three piles containing two, two, and one stones. In Figure 9 we show the values of the vertices of this game tree. The values of the vertices are computed using the values of the leaves and working one level up at a time. In the right margin of this figure we indicate whether we use the maximum or minimum of the values of the children to find the value of an internal vertex at each level. For example, once we have found the values of the three children of the root, which are 1 , -1 , and -1 , we find the value of the root by computing $\max(1, -1, -1) = 1$. Because the value of the root is 1 , it follows that the first player wins when both players follow a minmax strategy. ◀

Game trees for some well-known games can be extraordinarily large, because these games have many different possible moves. For example, the game tree for chess has been estimated to have as many as 10^{100} vertices! It may be impossible to use Theorem 3 directly to study a game because of the size of the game tree. Therefore, various approaches have been devised to help determine good strategies and to determine the outcome of such games. One useful technique, called *alpha-beta pruning*, eliminates much computation by pruning portions of the game tree that cannot affect the values of ancestor vertices. (For information about alpha-beta pruning, consult [Gr90].) Another useful approach is to use *evaluation functions*, which estimate the value of internal vertices in the game tree when it is not feasible to compute these values exactly. For example, in the game of tic-tac-toe, as an evaluation function for a position, we may use the number of files (rows, columns, and diagonals) containing no Os (used to indicate moves of the second player) minus the number of files containing no Xs (used to indicate moves of the first player). This evaluation function provides some indication of which player has the advantage in the game. Once the values of an evaluation function are inserted, the value of the game can be computed following the rules used for the minmax strategy. Computer programs created to play chess, such as the famous Deep Blue program, are based on sophisticated evaluation functions. For more information about how computers play chess see [Le91].

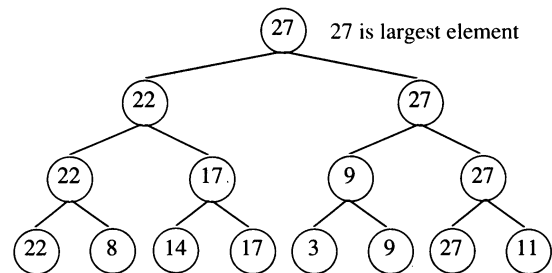


Exercises

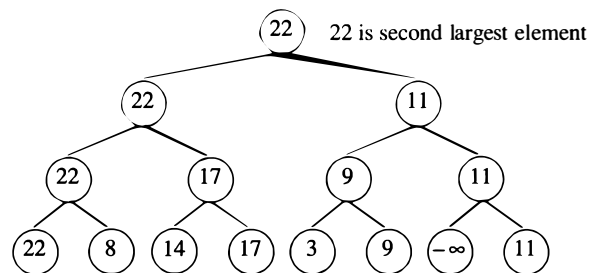
- Build a binary search tree for the words *banana*, *peach*, *apple*, *pear*, *coconut*, *mango*, and *papaya* using alphabetical order.
- Build a binary search tree for the words *oenology*, *phrenology*, *campanology*, *ornithology*, *ichthyology*, *limnology*, *alchemy*, and *astrology* using alphabetical order.
- How many comparisons are needed to locate or to add each of these words in the search tree for Exercise 1, starting fresh each time?
 - pear*
 - banana*
 - kumquat*
 - orange*
- How many comparisons are needed to locate or to add each of the words in the search tree for Exercise 2, starting fresh each time?
 - palmistry*
 - etymology*
 - paleontology*
 - glaciology*
- Using alphabetical order, construct a binary search tree for the words in the sentence “*The quick brown fox jumps over the lazy dog.*”
- How many weighings of a balance scale are needed to find a lighter counterfeit coin among four coins? Describe an algorithm to find the lighter coin using this number of weighings.
- How many weighings of a balance scale are needed to find a counterfeit coin among four coins if the counterfeit coin may be either heavier or lighter than the others? Describe an algorithm to find the counterfeit coin using this number of weighings.
- How many weighings of a balance scale are needed to find a counterfeit coin among eight coins if the counterfeit coin is either heavier or lighter than the others? Describe an algorithm to find the counterfeit coin using this number of weighings.
- How many weighings of a balance scale are needed to find a counterfeit coin among 12 coins if the counterfeit coin is lighter than the others? Describe an algorithm to find the lighter coin using this number of weighings.
- One of four coins may be counterfeit. If it is counterfeit, it may be lighter or heavier than the others. How many weighings are needed, using a balance scale, to determine whether there is a counterfeit coin, and if there is, whether it is lighter or heavier than the others? Describe an algorithm to find the counterfeit coin and determine whether it is lighter or heavier using this number of weighings.
- Find the least number of comparisons needed to sort four elements and devise an algorithm that sorts these elements using this number of comparisons.
- Find the least number of comparisons needed to sort five elements and devise an algorithm that sorts these elements using this number of comparisons.

The **tournament sort** is a sorting algorithm that works by building an ordered binary tree. We represent the elements to

be sorted by vertices that will become the leaves. We build up the tree one level at a time as we would construct the tree representing the winners of matches in a tournament. Working left to right, we compare pairs of consecutive elements, adding a parent vertex labeled with the larger of the two elements under comparison. We make similar comparisons between labels of vertices at each level until we reach the root of the tree that is labeled with the largest element. The tree constructed by the tournament sort of 22, 8, 14, 17, 3, 9, 27, 11 is illustrated in part (a) of the figure. Once the largest element has been determined, the leaf with this label is relabeled by $-\infty$, which is defined to be less than every element. The labels of all vertices on the path from this vertex up to the root of the tree are recalculated, as shown in part (b) of the figure. This produces the second largest element. This process continues until the entire list has been sorted.



(a)

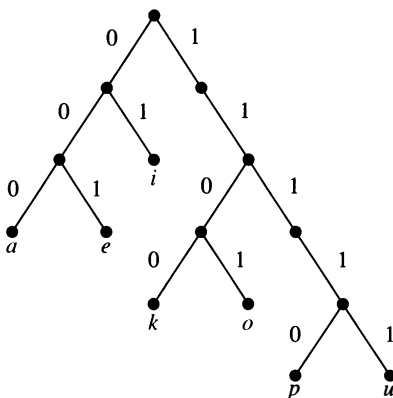


(b)

- Complete the tournament sort of the list 22, 8, 14, 17, 3, 9, 27, 11. Show the labels of the vertices at each step.
- Use the tournament sort to sort the list 17, 4, 1, 5, 13, 10, 14, 6.
- Describe the tournament sort using pseudocode.
- Assuming that n , the number of elements to be sorted, equals 2^k for some positive integer k , determine the number of comparisons used by the tournament sort to find the largest element of the list using the tournament sort.
- How many comparisons does the tournament sort use to find the second largest, the third largest, and so on, up to the $(n - 1)$ st largest (or second smallest) element?
- Show that the tournament sort requires $\Theta(n \log n)$ comparisons to sort a list of n elements. [Hint: By inserting the appropriate number of dummy elements defined to be

smaller than all integers, such as $-\infty$, assume that $n = 2^k$ for some positive integer k .]

19. Which of these codes are prefix codes?
- $a: 11, e: 00, t: 10, s: 01$
 - $a: 0, e: 1, t: 01, s: 001$
 - $a: 101, e: 11, t: 001, s: 011, n: 010$
 - $a: 010, e: 11, t: 011, s: 1011, n: 1001, i: 10101$
20. Construct the binary tree with prefix codes representing these coding schemes.
- $a: 11, e: 0, t: 101, s: 100$
 - $a: 1, e: 01, t: 001, s: 0001, n: 00001$
 - $a: 1010, e: 0, t: 11, s: 1011, n: 1001, i: 10001$
21. What are the codes for $a, e, i, k, o, p,$ and u if the coding scheme is represented by this tree?



22. Given the coding scheme $a: 001, b: 0001, e: 1, r: 0000, s: 0100, t: 011, x: 01010$, find the word represented by
- 01110100011.
 - 0001110000.
 - 0100101010.
 - 01100101010.
23. Use Huffman coding to encode these symbols with given frequencies: $a: 0.20, b: 0.10, c: 0.15, d: 0.25, e: 0.30$. What is the average number of bits required to encode a character?
24. Use Huffman coding to encode these symbols with given frequencies: $A: 0.10, B: 0.25, C: 0.05, D: 0.15, E: 0.30, F: 0.07, G: 0.08$. What is the average number of bits required to encode a symbol?
25. Construct two different Huffman codes for these symbols and frequencies: $t: 0.2, u: 0.3, v: 0.2, w: 0.3$.
26. a) Use Huffman coding to encode these symbols with frequencies $a: 0.4, b: 0.2, c: 0.2, d: 0.1, e: 0.1$ in two different ways by breaking ties in the algorithm differently. First, among the trees of minimum weight select two trees with the largest number of vertices to combine at each stage of the algorithm. Second, among the trees of minimum weight select two trees with the smallest number of vertices at each stage.
- b) Compute the average number of bits required to encode a symbol with each code and compute the variances of this number of bits for each code. Which tie-breaking procedure produced the smaller variance in the number of bits required to encode a symbol?

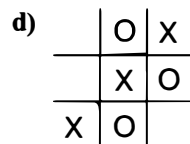
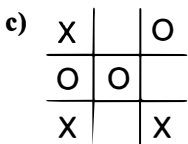
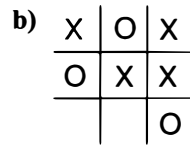
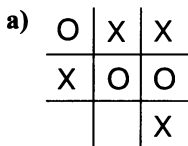
27. Construct a Huffman code for the letters of the English alphabet where the frequencies of letters in typical English text are as shown in this table.

Letter	Frequency	Letter	Frequency
A	0.0817	N	0.0662
B	0.0145	O	0.0781
C	0.0248	P	0.0156
D	0.0431	Q	0.0009
E	0.1232	R	0.0572
F	0.0209	S	0.0628
G	0.0182	T	0.0905
H	0.0668	U	0.0304
I	0.0689	V	0.0102
J	0.0010	W	0.0264
K	0.0080	X	0.0015
L	0.0397	Y	0.0211
M	0.0277	Z	0.0005

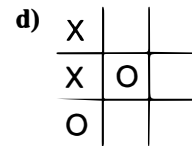
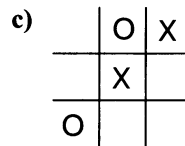
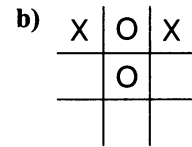
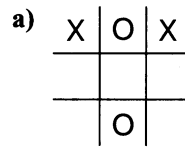
Suppose that m is a positive integer with $m \geq 2$. An m -ary Huffman code for a set of N symbols can be constructed analogously to the construction of a binary Huffman code. At the initial step, $((N - 1) \bmod (m - 1)) + 1$ trees consisting of a single vertex with least weights are combined into a rooted tree with these vertices as leaves. At each subsequent step, the m trees of least weight are combined into an m -ary tree.

28. Describe the m -ary Huffman coding algorithm in pseudocode.
29. Using the symbols 0, 1, and 2 use ternary ($m = 3$) Huffman coding to encode these letters with the given frequencies: $A: 0.25, E: 0.30, N: 0.10, R: 0.05, T: 0.12, Z: 0.18$.
30. Consider the three symbols A, B, and C with frequencies $A: 0.80, B: 0.19, C: 0.01$.
- Construct a Huffman code for these three symbols.
 - Form a new set of nine symbols by grouping together blocks of two symbols, AA, AB, AC, BA, BB, BC, CA, CB, and CC. Construct a Huffman code for these nine symbols, assuming that the occurrences of symbols in the original text are independent.
 - Compare the average number of bits required to encode text using the Huffman code for the three symbols in part (a) and the Huffman code for the nine blocks of two symbols constructed in part (b). Which is more efficient?
31. Given $n + 1$ symbols $x_1, x_2, \dots, x_n, x_{n+1}$ appearing $1, f_1, f_2, \dots, f_n$ times in a symbol string, respectively, where f_j is the j th Fibonacci number, what is the maximum number of bits used to encode a symbol when all possible tie-breaking selections are considered at each stage of the Huffman coding algorithm?

- 32. Show that Huffman codes are optimal in the sense that they represent a string of symbols using the fewest bits among all binary prefix codes.
- 33. Draw a game tree for nim if the starting position consists of two piles with two and three stones, respectively. When drawing the tree represent by the same vertex symmetric positions that result from the same move. Find the value of each vertex of the game tree. Who wins the game if both players follow an optimal strategy?
- 34. Draw a game tree for nim if the starting position consists of three piles with one, two, and three stones, respectively. When drawing the tree represent by the same vertex symmetric positions that result from the same move. Find the value of each vertex of the game tree. Who wins the game if both players follow an optimal strategy?
- 35. Suppose that we vary the payoff to the winning player in the game of nim so that the payoff is n dollars when n is the number of legal moves made before a terminal position is reached. Find the payoff to the first player if the initial position consists of
 - a) two piles with one and three stones, respectively.
 - b) two piles with two and four stones, respectively.
 - c) three piles with one, two, and three stones, respectively.
- 36. Suppose that in a variation of the game of nim we allow a player to either remove one or more stones from a pile or merge the stones from two piles into one pile as long as at least one stone remains. Draw the game tree for this variation of nim if the starting position consists of three piles containing two, two, and one stone, respectively. Find the values of each vertex in the game tree and determine the winner if both players follow an optimal strategy.
- 37. Draw the subtree of the game tree for tic-tac-toe beginning at each of these positions. Determine the value of each of these subtrees.



- 38. Suppose that the first four moves of a tic-tac-toe game are as shown. Does the first player (whose moves are marked by Xs) have a strategy that will always win?



- 39. Show that if a game of nim begins with two piles containing the same number of stones, as long as this number is at least two, then the second player wins when both players follow optimal strategies.
- 40. Show that if a game of nim begins with two piles containing different numbers of stones, the first player wins when both players follow optimal strategies.
- 41. How many children does the root of the game tree for checkers have? How many grandchildren does it have?
- 42. How many children does the root of the game tree for nim have and how many grandchildren does it have if the starting position is
 - a) piles with four and five stones, respectively.
 - b) piles with two, three, and four stones, respectively.
 - c) piles with one, two, three, and four stones, respectively.
 - d) piles with two, two, three, three, and five stones, respectively.
- 43. Draw the game tree for the game of tic-tac-toe for the levels corresponding to the first two moves. Assign the value of the evaluation function mentioned in the text that assigns to a position the number of files containing no Os minus the number of files containing no Xs as the value of each vertex at this level and compute the value of the tree for vertices as if the evaluation function gave the correct values for these vertices.
- 44. Use pseudocode to describe an algorithm for determining the value of a game tree when both players follow a minmax strategy.

10.3 Tree Traversal

Introduction



Ordered rooted trees are often used to store information. We need procedures for visiting each vertex of an ordered rooted tree to access data. We will describe several important algorithms for visiting all the vertices of an ordered rooted tree. Ordered rooted trees can also be used to represent various types of expressions, such as arithmetic expressions involving numbers,

variables, and operations. The different listings of the vertices of ordered rooted trees used to represent expressions are useful in the evaluation of these expressions.

Universal Address Systems

Procedures for traversing all vertices of an ordered rooted tree rely on the orderings of children. In ordered rooted trees, the children of an internal vertex are shown from left to right in the drawings representing these directed graphs.

We will describe one way we can totally order the vertices of an ordered rooted tree. To produce this ordering, we must first label all the vertices. We do this recursively:

1. Label the root with the integer 0. Then label its k children (at level 1) from left to right with $1, 2, 3, \dots, k$.
2. For each vertex v at level n with label A , label its k_v children, as they are drawn from left to right, with $A.1, A.2, \dots, A.k_v$.

Following this procedure, a vertex v at level n , for $n \geq 1$, is labeled $x_1.x_2.\dots.x_n$, where the unique path from the root to v goes through the x_1 st vertex at level 1, the x_2 nd vertex at level 2, and so on. This labeling is called the **universal address system** of the ordered rooted tree.

We can totally order the vertices using the lexicographic ordering of their labels in the universal address system. The vertex labeled $x_1.x_2.\dots.x_n$ is less than the vertex labeled $y_1.y_2.\dots.y_m$ if there is an i , $0 \leq i \leq n$, with $x_1 = y_1, x_2 = y_2, \dots, x_{i-1} = y_{i-1}$, and $x_i < y_i$; or if $n < m$ and $x_i = y_i$ for $i = 1, 2, \dots, n$.

EXAMPLE 1

We display the labelings of the universal address system next to the vertices in the ordered rooted tree shown in Figure 1. The lexicographic ordering of the labelings is



$$0 < 1 < 1.1 < 1.2 < 1.3 < 2 < 3 < 3.1 < 3.1.1 < 3.1.2 < 3.1.2.1 < 3.1.2.2 < 3.1.2.3 < 3.1.2.4 < 3.1.3 < 3.2 < 4 < 4.1 < 5 < 5.1 < 5.1.1 < 5.2 < 5.3 \quad \blacktriangleleft$$

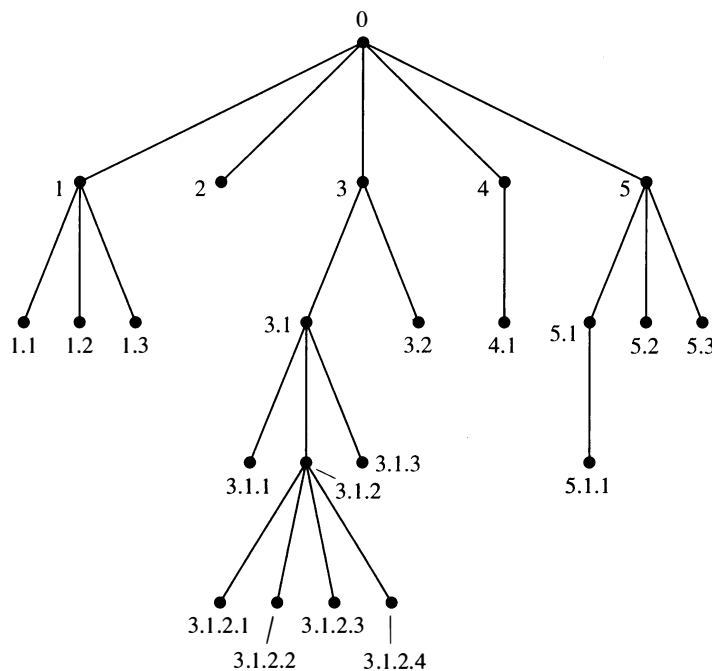


FIGURE 1 The Universal Address System of an Ordered Rooted Tree.

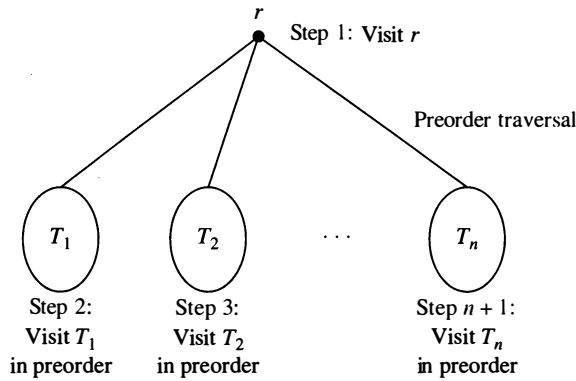


FIGURE 2 Preorder Traversal.

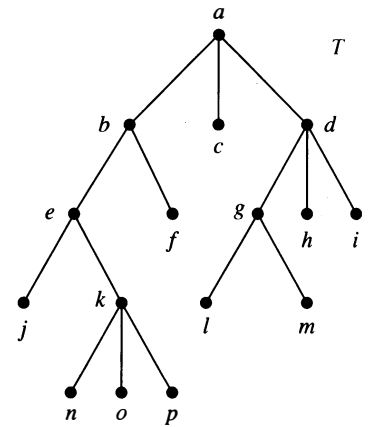


FIGURE 3 The Ordered Rooted Tree T .

Traversal Algorithms

Procedures for systematically visiting every vertex of an ordered rooted tree are called **traversal algorithms**. We will describe three of the most commonly used such algorithms, **preorder traversal**, **inorder traversal**, and **postorder traversal**. Each of these algorithms can be defined recursively. We first define preorder traversal.

DEFINITION 1 Let T be an ordered rooted tree with root r . If T consists only of r , then r is the *preorder traversal* of T . Otherwise, suppose that T_1, T_2, \dots, T_n are the subtrees at r from left to right in T . The *preorder traversal* begins by visiting r . It continues by traversing T_1 in preorder, then T_2 in preorder, and so on, until T_n is traversed in preorder.

The reader should verify that the preorder traversal of an ordered rooted tree gives the same ordering of the vertices as the ordering obtained using a universal address system. Figure 2 indicates how a preorder traversal is carried out.

Example 2 illustrates preorder traversal.

EXAMPLE 2 In which order does a preorder traversal visit the vertices in the ordered rooted tree T shown in Figure 3?



Solution: The steps of the preorder traversal of T are shown in Figure 4. We traverse T in preorder by first listing the root a , followed by the preorder list of the subtree with root b , the preorder list of the subtree with root c (which is just c) and the preorder list of the subtree with root d .

The preorder list of the subtree with root b begins by listing b , then the vertices of the subtree with root e in preorder, and then the subtree with root f in preorder (which is just f). The preorder list of the subtree with root d begins by listing d , followed by the preorder list of the subtree with root g , followed by the subtree with root h (which is just h), followed by the subtree with root i (which is just i).

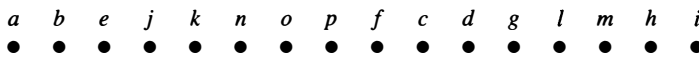
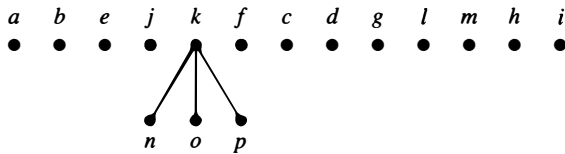
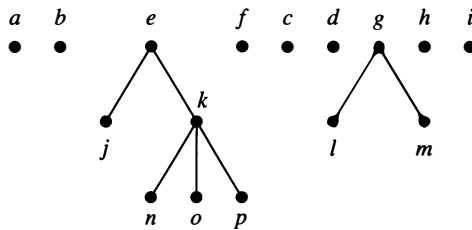
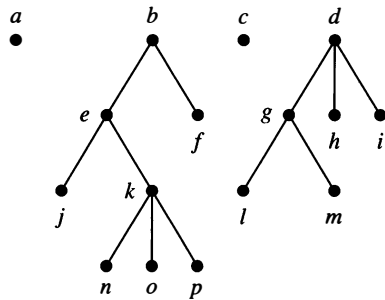
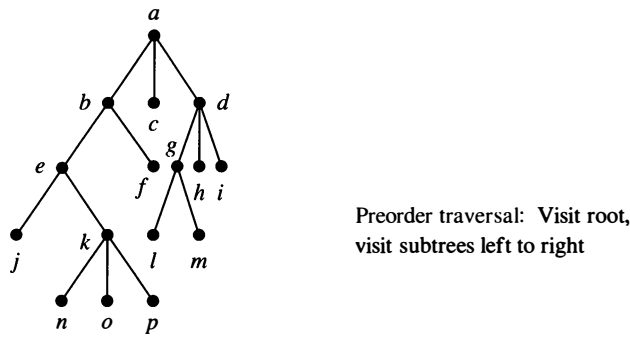


FIGURE 4 The Preorder Traversal of T .

The preorder list of the subtree with root e begins by listing e , followed by the preorder listing of the subtree with root j (which is just j), followed by the preorder listing of the subtree with root k . The preorder listing of the subtree with root g is g followed by l , followed by m . The preorder listing of the subtree with root k is k, n, o, p . Consequently, the preorder traversal of T is $a, b, e, j, k, n, o, p, f, c, d, g, l, m, h, i$. ◀

We will now define inorder traversal.

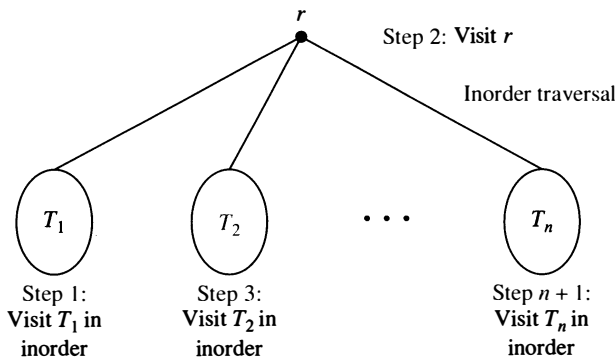


FIGURE 5 Inorder Traversal.

DEFINITION 2 Let T be an ordered rooted tree with root r . If T consists only of r , then r is the *inorder traversal* of T . Otherwise, suppose that T_1, T_2, \dots, T_n are the subtrees at r from left to right. The *inorder traversal* begins by traversing T_1 in inorder, then visiting r . It continues by traversing T_2 in inorder, then T_3 in inorder, \dots , and finally T_n in inorder.

Figure 5 indicates how inorder traversal is carried out. Example 3 illustrates how inorder traversal is carried out for a particular tree.

EXAMPLE 3 In which order does an inorder traversal visit the vertices of the ordered rooted tree T in Figure 3?



Solution: The steps of the inorder traversal of the ordered rooted tree T are shown in Figure 6. The inorder traversal begins with an inorder traversal of the subtree with root b , the root a , the inorder listing of the subtree with root c , which is just c , and the inorder listing of the subtree with root d .

The inorder listing of the subtree with root b begins with the inorder listing of the subtree with root e , the root b , and f . The inorder listing of the subtree with root d begins with the inorder listing of the subtree with root g , followed by the root d , followed by h , followed by i .

The inorder listing of the subtree with root e is j , followed by the root e , followed by the inorder listing of the subtree with root k . The inorder listing of the subtree with root g is l, g, m . The inorder listing of the subtree with root k is n, k, o, p . Consequently, the inorder listing of the ordered rooted tree is $j, e, n, k, o, p, b, f, a, c, l, g, m, d, h, i$. ◀

We now define postorder traversal.

DEFINITION 3 Let T be an ordered rooted tree with root r . If T consists only of r , then r is the *postorder traversal* of T . Otherwise, suppose that T_1, T_2, \dots, T_n are the subtrees at r from left to right. The *postorder traversal* begins by traversing T_1 in postorder, then T_2 in postorder, \dots , then T_n in postorder, and ends by visiting r .

Figure 7 illustrates how postorder traversal is done. Example 4 illustrates how postorder traversal works.

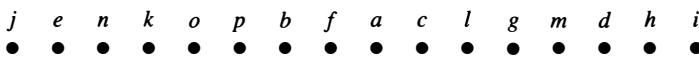
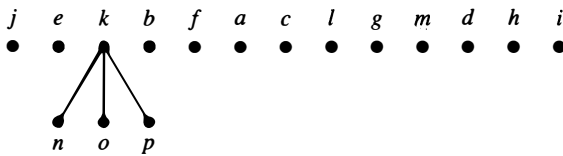
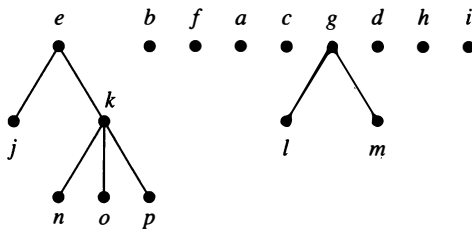
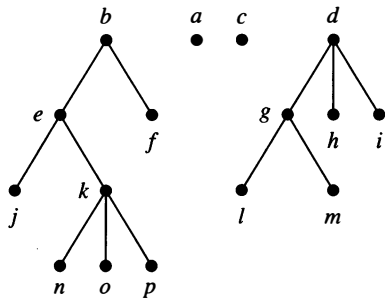
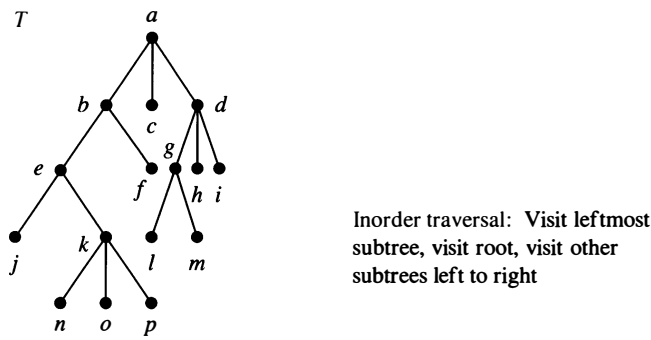


FIGURE 6 The Inorder Traversal of T .

EXAMPLE 4 In which order does a postorder traversal visit the vertices of the ordered rooted tree T shown in Figure 3?



Solution: The steps of the postorder traversal of the ordered rooted tree T are shown in Figure 8. The postorder traversal begins with the postorder traversal of the subtree with root b , the postorder traversal of the subtree with root c , which is just c , the postorder traversal of the subtree with root d , followed by the root a .

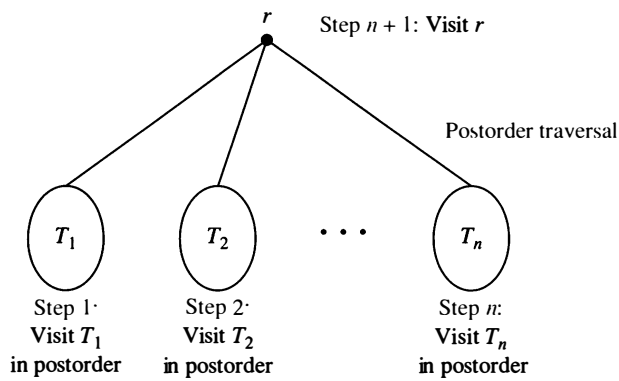


FIGURE 7 Postorder Traversal.

The postorder traversal of the subtree with root b begins with the postorder traversal of the subtree with root e , followed by f , followed by the root b . The postorder traversal of the rooted tree with root d begins with the postorder traversal of the subtree with root g , followed by h , followed by i , followed by the root d .

The postorder traversal of the subtree with root e begins with j , followed by the postorder traversal of the subtree with root k , followed by the root e . The postorder traversal of the subtree with root g is l, m, g . The postorder traversal of the subtree with root k is n, o, p, k . Therefore, the postorder traversal of T is $j, n, o, p, k, e, f, b, c, l, m, g, h, i, d, a$. ◀

There are easy ways to list the vertices of an ordered rooted tree in preorder, inorder, and postorder. To do this, first draw a curve around the ordered rooted tree starting at the root, moving along the edges, as shown in the example in Figure 9. We can list the vertices in preorder by listing each vertex the first time this curve passes it. We can list the vertices in inorder by listing a leaf the first time the curve passes it and listing each internal vertex the second time the curve passes it. We can list the vertices in postorder by listing a vertex the last time it is passed on the way back up to its parent. When this is done in the rooted tree in Figure 9, it follows that the preorder traversal gives $a, b, d, h, e, i, j, c, f, g, k$, the inorder traversal gives $h, d, b, i, e, j, a, f, c, k, g$; and the postorder traversal gives $h, d, i, j, e, b, f, k, g, c, a$.

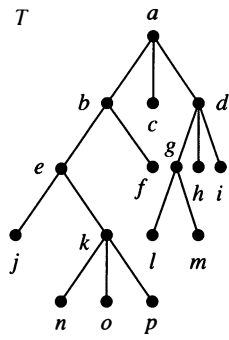
Algorithms for traversing ordered rooted trees in preorder, inorder, or postorder are most easily expressed recursively.

ALGORITHM 1 Preorder Traversal.

```

procedure preorder( $T$ : ordered rooted tree)
 $r$  := root of  $T$ 
list  $r$ 
for each child  $c$  of  $r$  from left to right
begin
 $T(c)$  := subtree with  $c$  as its root
preorder( $T(c)$ )
end

```



Postorder traversal: Visit subtrees left to right; visit root

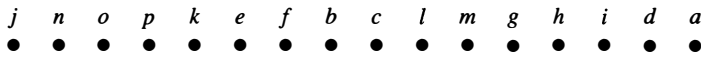
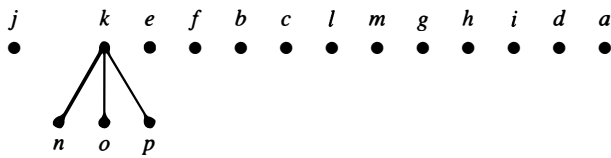
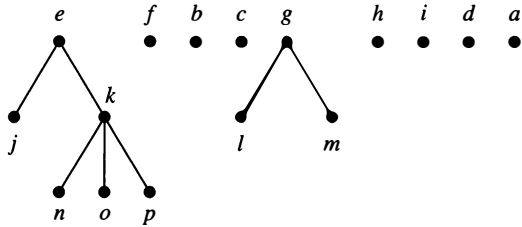
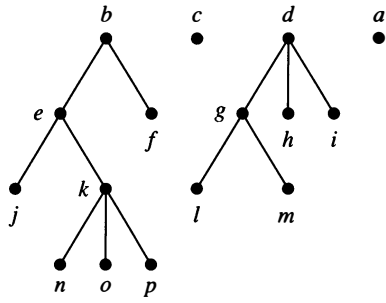


FIGURE 8 The Postorder Traversal of *T*.

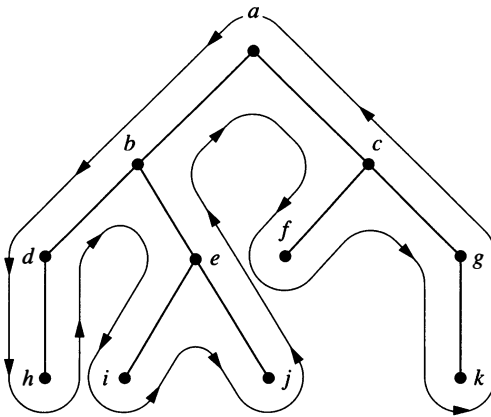


FIGURE 9 A Shortcut for Traversing an Ordered Rooted Tree in Preorder, Inorder, and Postorder.

ALGORITHM 2 Inorder Traversal.

```

procedure inorder(T: ordered rooted tree)
  r := root of T
  if r is a leaf then list r
  else
  begin
    l := first child of r from left to right
    T(l) := subtree with l as its root
    inorder(T(l))
    list r
    for each child c of r except for l from left to right
      T(c) := subtree with c as its root
      inorder(T(c))
  end

```

ALGORITHM 3 Postorder Traversal.

```

procedure postorder(T: ordered rooted tree)
  r := root of T
  for each child c of r from left to right
  begin
    T(c) := subtree with c as its root
    postorder(T(c))
  end
  list r

```

Note that both the preorder traversal and the postorder traversal encode the structure of an ordered rooted tree when the number of children of each vertex is specified. That is, an ordered rooted tree is uniquely determined when we specify a list of vertices generated by a preorder traversal or by a postorder traversal of the tree, together with the number of children of each vertex

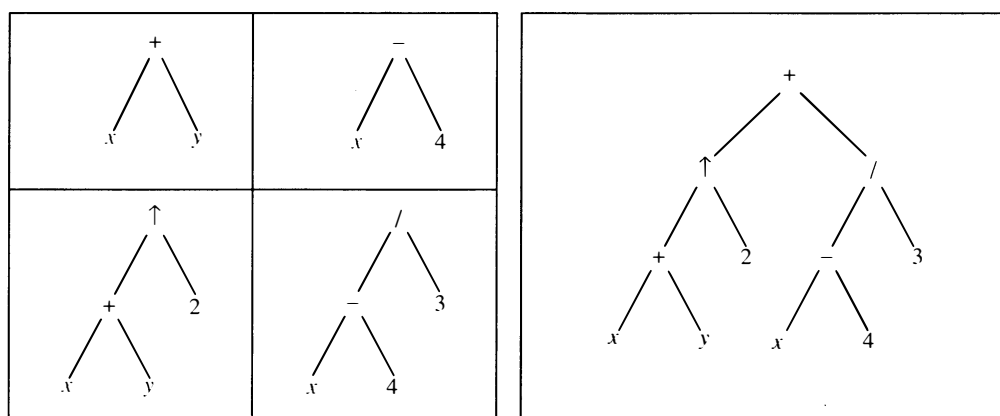


FIGURE 10 A Binary Tree Representing $((x + y) \uparrow 2) + ((x - 4)/3)$.

(see Exercises 26 and 27). In particular, both a preorder traversal and a postorder traversal encode the structure of a full ordered m -ary tree. However, when the number of children of vertices is not specified, neither a preorder traversal nor a postorder traversal encodes the structure of an ordered rooted tree (see Exercises 28 and 29).

Infix, Prefix, and Postfix Notation

We can represent complicated expressions, such as compound propositions, combinations of sets, and arithmetic expressions using ordered rooted trees. For instance, consider the representation of an arithmetic expression involving the operators $+$ (addition), $-$ (subtraction), $*$ (multiplication), $/$ (division), and \uparrow (exponentiation). We will use parentheses to indicate the order of the operations. An ordered rooted tree can be used to represent such expressions, where the internal vertices represent operations, and the leaves represent the variables or numbers. Each operation operates on its left and right subtrees (in that order).

EXAMPLE 5 What is the ordered rooted tree that represents the expression $((x + y) \uparrow 2) + ((x - 4)/3)$?

Solution: The binary tree for this expression can be built from the bottom up. First, a subtree for the expression $x + y$ is constructed. Then this is incorporated as part of the larger subtree representing $(x + y) \uparrow 2$. Also, a subtree for $x - 4$ is constructed, and then this is incorporated into a subtree representing $(x - 4)/3$. Finally the subtrees representing $(x + y) \uparrow 2$ and $(x - 4)/3$ are combined to form the ordered rooted tree representing $((x + y) \uparrow 2) + ((x - 4)/3)$. These steps are shown in Figure 10. ◀

An inorder traversal of the binary tree representing an expression produces the original expression with the elements and operations in the same order as they originally occurred, except for unary operations, which instead immediately follow their operands. For instance, inorder traversals of the binary trees in Figure 11, which represent the expressions $(x + y)/(x + 3)$, $(x + (y/x)) + 3$, and $x + (y/(x + 3))$, all lead to the infix expression $x + y/x + 3$. To make such expressions unambiguous it is necessary to include parentheses in the inorder traversal whenever we encounter an operation. The fully parenthesized expression obtained in this way is said to be in **infix form**.

We obtain the **prefix form** of an expression when we traverse its rooted tree in preorder. Expressions written in prefix form are said to be in **Polish notation**, which is named after the Polish logician Jan Łukasiewicz. An expression in prefix notation (where each operation has

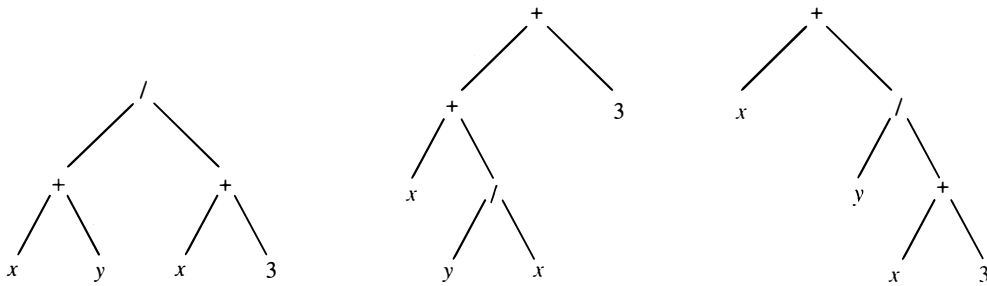


FIGURE 11 Rooted Trees Representing $(x + y)/(x + 3)$, $(x + (y/x)) + 3$, and $x + (y/(x + 3))$.

a specified number of operands), is unambiguous, so no parentheses are needed in such an expression. The verification of this is left as an exercise for the reader.

EXAMPLE 6 What is the prefix form for $((x + y) \uparrow 2) + ((x - 4)/3)$?

Solution: We obtain the prefix form for this expression by traversing the binary tree that represents it, shown in Figure 10. This produces $+ \uparrow + x y 2 / - x 4 3$. ◀

In the prefix form of an expression, a binary operator, such as $+$, precedes its two operands. Hence, we can evaluate an expression in prefix form by working from right to left. When we encounter an operator, we perform the corresponding operation with the two operands immediately to the right of this operand. Also, whenever an operation is performed, we consider the result a new operand.

EXAMPLE 7 What is the value of the prefix expression $+ - * 2 3 5 / \uparrow 2 3 4$?

Solution: The steps used to evaluate this expression by working right to left, and performing operations using the operands on the right, are shown in Figure 12. The value of this expression is 3. ◀



We obtain the **postfix form** of an expression by traversing its binary tree in postorder. Expressions written in postfix form are said to be in **reverse Polish notation**. Expressions in reverse Polish notation are unambiguous, so parentheses are not needed. The verification of this is left to the reader.

EXAMPLE 8 What is the postfix form of the expression $((x + y) \uparrow 2) + ((x - 4)/3)$?

Solution: The postfix form of the expression is obtained by carrying out a postorder traversal of the binary tree for this expression, shown in Figure 10. This produces the postfix expression: $x y + 2 \uparrow x 4 - 3 / +$. ◀

In the postfix form of an expression, a binary operator follows its two operands. So, to evaluate an expression from its postfix form, work from left to right, carrying out operations whenever an operator follows two operands. After an operation is carried out, the result of this operation becomes a new operand.

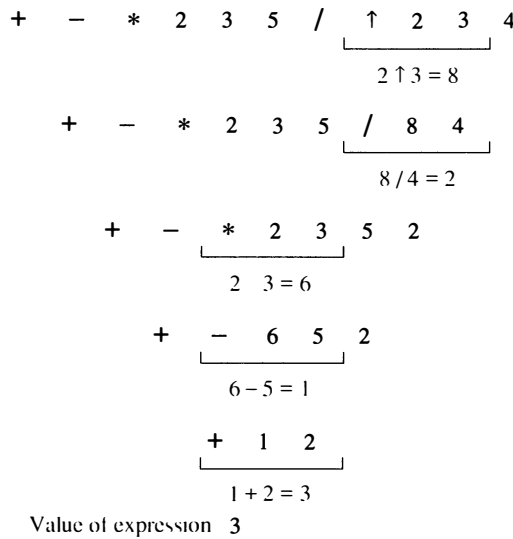


FIGURE 12 Evaluating a Prefix Expression.

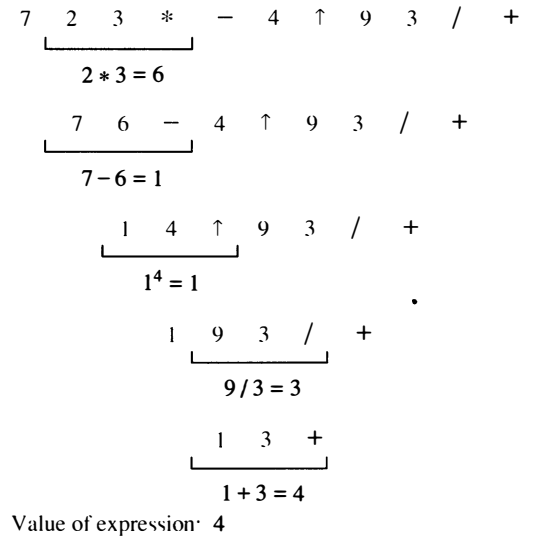


FIGURE 13 Evaluating a Postfix Expression.

EXAMPLE 9 What is the value of the postfix expression $7 \ 2 \ 3 \ * \ - \ 4 \ \uparrow \ 9 \ 3 \ / \ +$?

Solution: The steps used to evaluate this expression by starting at the left and carrying out operations when two operands are followed by an operator are shown in Figure 13. The value of this expression is 4. ◀

Rooted trees can be used to represent other types of expressions, such as those representing compound propositions and combinations of sets. In these examples unary operators, such as the negation of a proposition, occur. To represent such operators and their operands, a vertex representing the operator and a child of this vertex representing the operand are used.

EXAMPLE 10 Find the ordered rooted tree representing the compound proposition $(\neg(p \wedge q)) \leftrightarrow (\neg p \vee \neg q)$. Then use this rooted tree to find the prefix, postfix, and infix forms of this expression.



JAN ŁUKASIEWICZ (1878–1956) Jan Łukasiewicz was born into a Polish-speaking family in Lvov. At that time Lvov was part of Austria, but it is now in the Ukraine. His father was a captain in the Austrian army. Łukasiewicz became interested in mathematics while in high school. He studied mathematics and philosophy at the University of Lvov at both the undergraduate and graduate levels. After completing his doctoral work he became a lecturer there, and in 1911 he was appointed to a professorship. When the University of Warsaw was reopened as a Polish university in 1915, Łukasiewicz accepted an invitation to join the faculty. In 1919 he served as the Polish Minister of Education. He returned to the position of professor at Warsaw University where he remained from 1920 to 1939, serving as rector of the university twice.

Łukasiewicz was one of the cofounders of the famous Warsaw School of Logic. He published his famous text, *Elements of Mathematical Logic*, in 1928. With his influence, mathematical logic was made a required course for mathematics and science undergraduates in Poland. His lectures were considered excellent, even attracting students of the humanities.

Łukasiewicz and his wife experienced great suffering during World War II, which he documented in a posthumously published autobiography. After the war they lived in exile in Belgium. Fortunately, in 1949 he was offered a position at the Royal Irish Academy in Dublin.

Łukasiewicz worked on mathematical logic throughout his career. His work on a three-valued logic was an important contribution to the subject. Nevertheless, he is best known in the mathematical and computer science communities for his introduction of parenthesis-free notation, now called Polish notation.

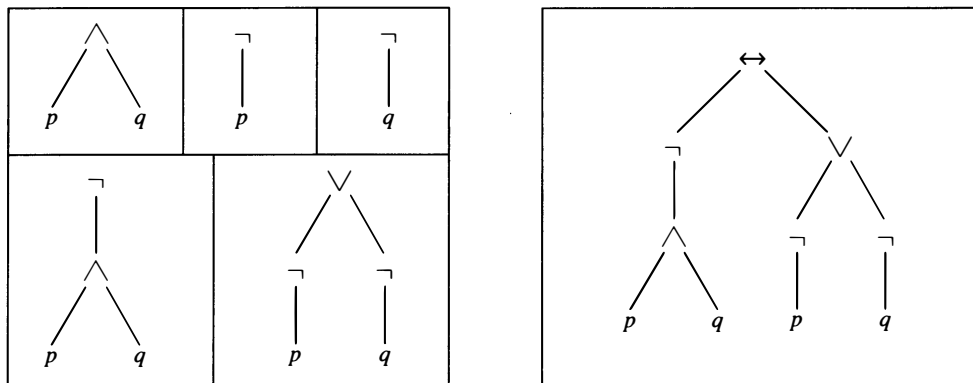


FIGURE 14 Constructing the Rooted Tree for a Compound Proposition.



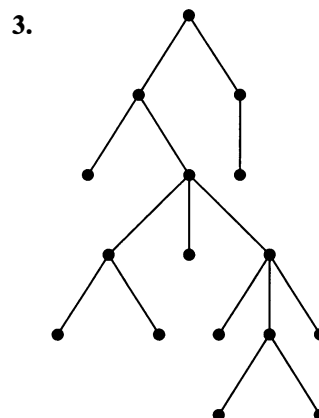
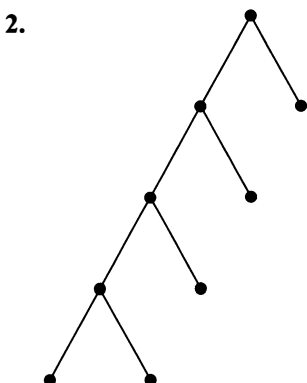
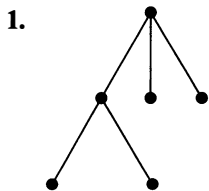
Solution: The rooted tree for this compound proposition is constructed from the bottom up. First, subtrees for $\neg p$ and $\neg q$ are formed (where \neg is considered a unary operator). Also, a subtree for $p \wedge q$ is formed. Then subtrees for $\neg(p \wedge q)$ and $(\neg p) \vee (\neg q)$ are constructed. Finally, these two subtrees are used to form the final rooted tree. The steps of this procedure are shown in Figure 14.

The prefix, postfix, and infix forms of this expression are found by traversing this rooted tree in preorder, postorder, and inorder (including parentheses), respectively. These traversals give $\leftrightarrow \neg \wedge pq \vee \neg p \neg q$, $pq \wedge \neg p \neg q \neg \vee \leftrightarrow$, and $(\neg(p \wedge q)) \leftrightarrow ((\neg p) \vee (\neg q))$, respectively. ◀

Because prefix and postfix expressions are unambiguous and because they can be evaluated easily without scanning back and forth, they are used extensively in computer science. Such expressions are especially useful in the construction of compilers.

Exercises

In Exercises 1–3 construct the universal address system for the given ordered rooted tree. Then use this to order its vertices using the lexicographic order of their labels.

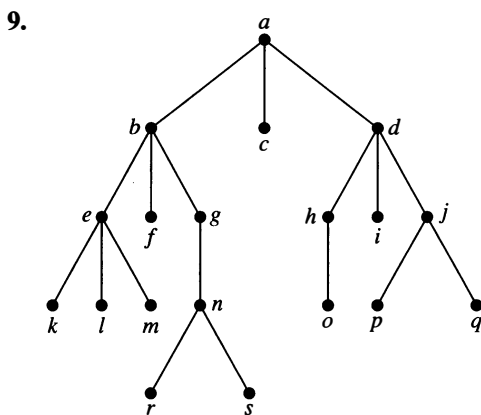
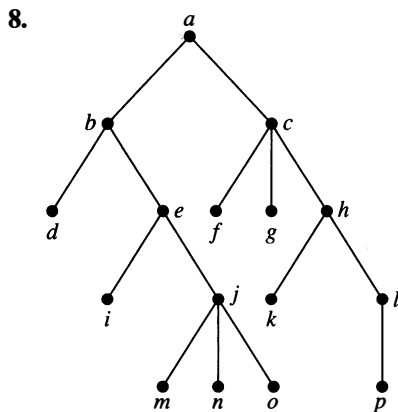
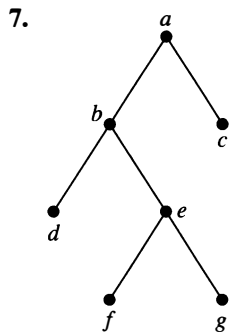


4. Suppose that the address of the vertex v in the ordered rooted tree T is 3.4.5.2.4.
 - a) At what level is v ?
 - b) What is the address of the parent of v ?
 - c) What is the least number of siblings v can have?
 - d) What is the smallest possible number of vertices in T if v has this address?
 - e) Find the other addresses that must occur.
5. Suppose that the vertex with the largest address in an

ordered rooted tree T has address 2.3.4.3.1. Is it possible to determine the number of vertices in T ?

6. Can the leaves of an ordered rooted tree have the following list of universal addresses? If so, construct such an ordered rooted tree.
 - a) 1.1.1, 1.1.2, 1.2, 2.1.1.1, 2.1.2, 2.1.3, 2.2, 3.1.1, 3.1.2.1, 3.1.2.2, 3.2
 - b) 1.1, 1.2.1, 1.2.2, 1.2.3, 2.1, 2.2.1, 2.3.1, 2.3.2, 2.4.2.1, 2.4.2.2, 3.1, 3.2.1, 3.2.2
 - c) 1.1, 1.2.1, 1.2.2, 1.2.2.1, 1.3, 1.4, 2, 3.1, 3.2, 4.1.1.1

In Exercises 7–9 determine the order in which a preorder traversal visits the vertices of the given ordered rooted tree.

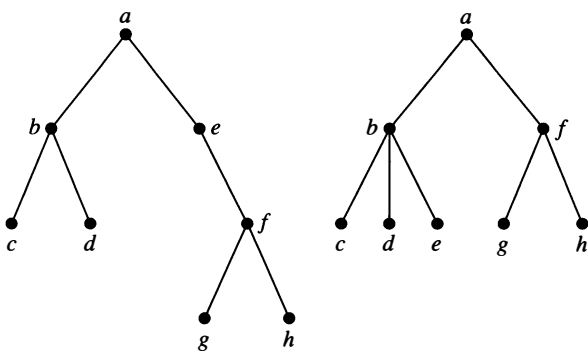


10. In which order are the vertices of the ordered rooted tree in Exercise 7 visited using an inorder traversal?
11. In which order are the vertices of the ordered rooted tree in Exercise 8 visited using an inorder traversal?

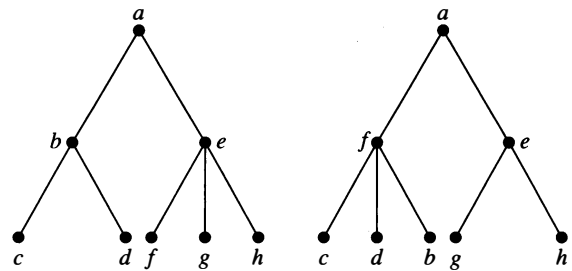
12. In which order are the vertices of the ordered rooted tree in Exercise 9 visited using an inorder traversal?
13. In which order are the vertices of the ordered rooted tree in Exercise 7 visited using a postorder traversal?
14. In which order are the vertices of the ordered rooted tree in Exercise 8 visited using a postorder traversal?
15. In which order are the vertices of the ordered rooted tree in Exercise 9 visited using a postorder traversal?
16. a) Represent the expression $((x + 2) \uparrow 3) * (y - (3 + x)) - 5$ using a binary tree. Write this expression in
 - b) prefix notation.
 - c) postfix notation.
 - d) infix notation.
17. a) Represent the expressions $(x + xy) + (x/y)$ and $x + ((xy + x)/y)$ using binary trees. Write these expressions in
 - b) prefix notation.
 - c) postfix notation.
 - d) infix notation.
18. a) Represent the compound propositions $\neg(p \wedge q) \leftrightarrow (\neg p \vee \neg q)$ and $(\neg p \wedge (q \leftrightarrow \neg p)) \vee \neg q$ using ordered rooted trees. Write these expressions in
 - b) prefix notation.
 - c) postfix notation.
 - d) infix notation.
19. a) Represent $(A \cap B) - (A \cup (B - A))$ using an ordered rooted tree. Write this expression in
 - b) prefix notation.
 - c) postfix notation.
 - d) infix notation.
- *20. In how many ways can the string $\neg p \wedge q \leftrightarrow \neg p \vee \neg q$ be fully parenthesized to yield an infix expression?
- *21. In how many ways can the string $A \cap B - A \cap B - A$ be fully parenthesized to yield an infix expression?
22. Draw the ordered rooted tree corresponding to each of these arithmetic expressions written in prefix notation. Then write each expression using infix notation.
 - a) $+ * + - 5 3 2 1 4$
 - b) $\uparrow + 2 3 - 5 1$
 - c) $* / 9 3 + * 2 4 - 7 6$
23. What is the value of each of these prefix expressions?
 - a) $- * 2 / 8 4 3$
 - b) $\uparrow - * 3 3 * 4 2 5$
 - c) $+ - \uparrow 3 2 \uparrow 2 3 / 6 - 4 2$
 - d) $* + 3 + 3 \uparrow 3 + 3 3 3$
24. What is the value of each of these postfix expressions?
 - a) $5 2 1 - - 3 1 4 + + *$
 - b) $9 3 / 5 + 7 2 - *$
 - c) $3 2 * 2 \uparrow 5 3 - 8 4 / * -$
25. Construct the ordered rooted tree whose preorder traversal is $a, b, f, c, g, h, i, d, e, j, k, l$, where a has four children,

c has three children, j has two children, b and e have one child each, and all other vertices are leaves.

- *26. Show that an ordered rooted tree is uniquely determined when a list of vertices generated by a preorder traversal of the tree and the number of children of each vertex are specified.
- *27. Show that an ordered rooted tree is uniquely determined when a list of vertices generated by a postorder traversal of the tree and the number of children of each vertex are specified.
- 28. Show that preorder traversals of the two ordered rooted trees displayed below produce the same list of vertices. Note that this does not contradict the statement in Exercise 26, because the numbers of children of internal vertices in the two ordered rooted trees differ.



- 29. Show that postorder traversals of these two ordered rooted trees produce the same list of vertices. Note that this does not contradict the statement in Exercise 27, because the numbers of children of internal vertices in the two ordered rooted trees differ.



Well-formed formulae in prefix notation over a set of symbols and a set of binary operators are defined recursively by these rules:

- (i) if x is a symbol, then x is a well-formed formula in prefix notation;
 - (ii) if X and Y are well-formed formulae and $*$ is an operator, then $* XY$ is a well-formed formula.
- 30. Which of these are well-formed formulae over the symbols $\{x, y, z\}$ and the set of binary operators $\{\times, +, \circ\}$?
 - a) $\times + + x y x$
 - b) $\circ x y \times x z$
 - c) $\times \circ x z \times \times x y$
 - d) $\times + \circ x x \circ x x x$
 - *31. Show that any well-formed formula in prefix notation over a set of symbols and a set of binary operators contains exactly one more symbol than the number of operators.
 - 32. Give a definition of well-formed formulae in postfix notation over a set of symbols and a set of binary operators.
 - 33. Give six examples of well-formed formulae with three or more operators in postfix notation over the set of symbols $\{x, y, z\}$ and the set of operators $\{+, \times, \circ\}$.
 - 34. Extend the definition of well-formed formulae in prefix notation to sets of symbols and operators where the operators may not be binary.

10.4 Spanning Trees

Introduction

Consider the system of roads in Maine represented by the simple graph shown in Figure 1(a). The only way the roads can be kept open in the winter is by frequently plowing them. The highway department wants to plow the fewest roads so that there will always be cleared roads connecting any two towns. How can this be done?

At least five roads must be plowed to ensure that there is a path between any two towns. Figure 1(b) shows one such set of roads. Note that the subgraph representing these roads is a tree, because it is connected and contains six vertices and five edges.

This problem was solved with a connected subgraph with the minimum number of edges containing all vertices of the original simple graph. Such a graph must be a tree.

DEFINITION 1 Let G be a simple graph. A *spanning tree* of G is a subgraph of G that is a tree containing every vertex of G .

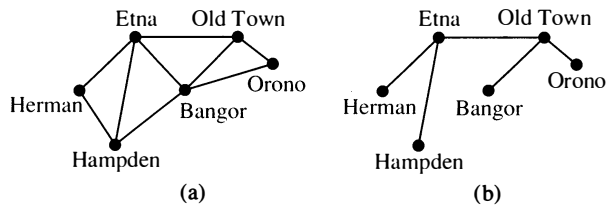


FIGURE 1 (a) A Road System and (b) a Set of Roads to Plow.

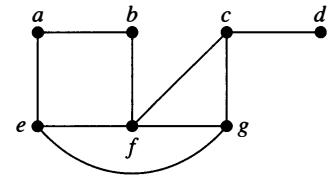


FIGURE 2 The Simple Graph G .

A simple graph with a spanning tree must be connected, because there is a path in the spanning tree between any two vertices. The converse is also true; that is, every connected simple graph has a spanning tree. We will give an example before proving this result.

EXAMPLE 1 Find a spanning tree of the simple graph G shown in Figure 2.

Solution: The graph G is connected, but it is not a tree because it contains simple circuits. Remove the edge $\{a, e\}$. This eliminates one simple circuit, and the resulting subgraph is still connected and still contains every vertex of G . Next remove the edge $\{e, f\}$ to eliminate a second simple circuit. Finally, remove edge $\{c, g\}$ to produce a simple graph with no simple circuits. This subgraph is a spanning tree, because it is a tree that contains every vertex of G . The sequence of edge removals used to produce the spanning tree is illustrated in Figure 3.

The tree shown in Figure 3 is not the only spanning tree of G . For instance, each of the trees shown in Figure 4 is a spanning tree of G . ◀

THEOREM 1 A simple graph is connected if and only if it has a spanning tree.

Proof: First, suppose that a simple graph G has a spanning tree T . T contains every vertex of G . Furthermore, there is a path in T between any two of its vertices. Because T is a subgraph of G , there is a path in G between any two of its vertices. Hence, G is connected.

Now suppose that G is connected. If G is not a tree, it must contain a simple circuit. Remove an edge from one of these simple circuits. The resulting subgraph has one fewer edge but still contains all the vertices of G and is connected. This subgraph is still connected because when two vertices are connected by a path containing the removed edge, they are connected by a path not containing this edge. We can construct such a path by inserting into the original path, at

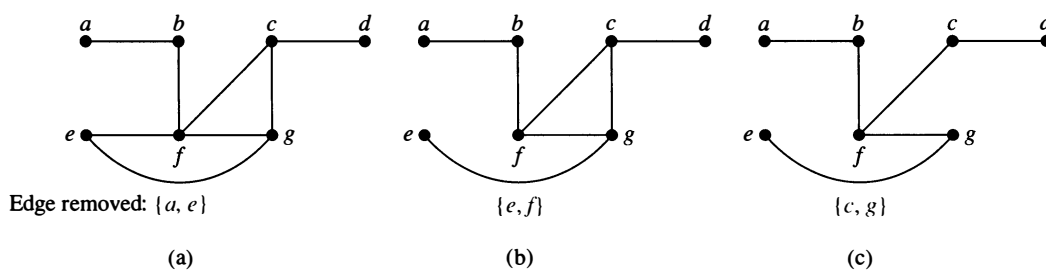


FIGURE 3 Producing a Spanning Tree for G by Removing Edges That Form Simple Circuits.

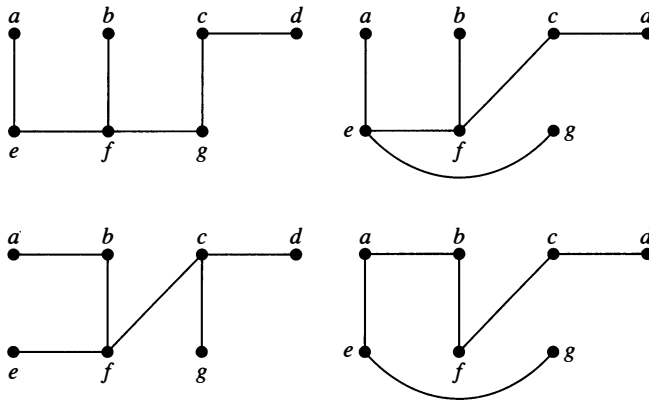


FIGURE 4 Spanning Trees of G .

the point where the removed edge once was, the simple circuit with this edge removed. If this subgraph is not a tree, it has a simple circuit; so as before, remove an edge that is in a simple circuit. Repeat this process until no simple circuits remain. This is possible because there are only a finite number of edges in the graph. The process terminates when no simple circuits remain. A tree is produced because the graph stays connected as edges are removed. This tree is a spanning tree because it contains every vertex of G . ◀

Spanning trees are important in data networking, as Example 2 shows.

EXAMPLE 2 IP Multicasting Spanning trees play an important role in multicasting over Internet Protocol (IP) networks. To send data from a source computer to multiple receiving computers, each of which is a subnetwork, data could be sent separately to each computer. This type of networking, called unicasting, is inefficient, because many copies of the same data are transmitted over the network. To make the transmission of data to multiple receiving computers more efficient, IP multicasting is used. With IP multicasting, a computer sends a single copy of data over the network, and as data reaches intermediate routers, the data are forwarded to one or more other routers so that ultimately all receiving computers in their various subnetworks receive these data. (Routers are computers that are dedicated to forwarding IP datagrams between subnetworks in a network. In multicasting, routers use Class D addresses, each representing a session that receiving computers may join; see Example 16 in Section 5.1.)



For data to reach receiving computers as quickly as possible, there should be no loops (which in graph theory terminology are circuits or cycles) in the path that data take through the network. That is, once data have reached a particular router, data should never return to this router. To avoid loops, the multicast routers use network algorithms to construct a spanning tree in the graph that has the multicast source, the routers, and the subnetworks containing receiving computers as vertices, with edges representing the links between computers and/or routers. The root of this spanning tree is the multicast source. The subnetworks containing receiving computers are leaves of the tree. (Note that subnetworks not containing receiving stations are not included in the graph.) This is illustrated in Figure 5. ◀

Depth-First Search



The proof of Theorem 1 gives an algorithm for finding spanning trees by removing edges from simple circuits. This algorithm is inefficient, because it requires that simple circuits be identified.

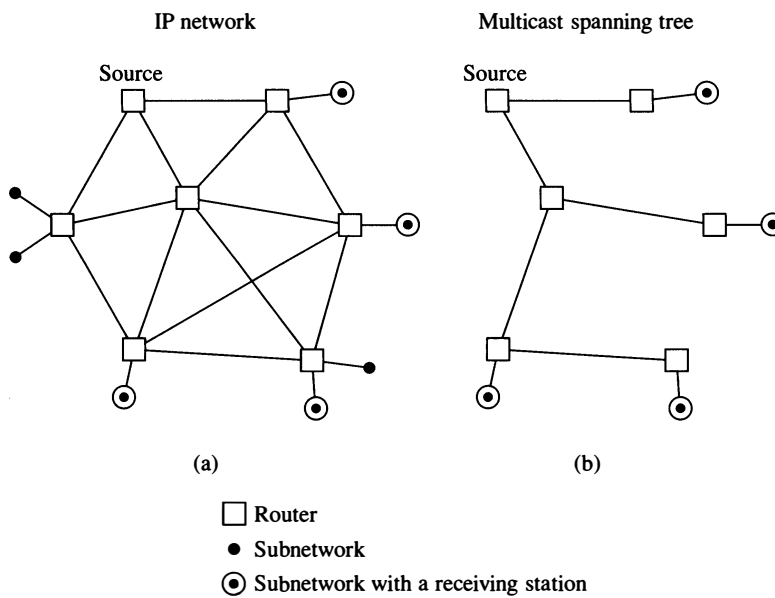


FIGURE 5 A Multicast Spanning Tree.

Instead of constructing spanning trees by removing edges, spanning trees can be built up by successively adding edges. Two algorithms based on this principle will be presented here.



We can build a spanning tree for a connected simple graph using **depth-first search**. We will form a rooted tree, and the spanning tree will be the underlying undirected graph of this rooted tree. Arbitrarily choose a vertex of the graph as the root. Form a path starting at this vertex by successively adding vertices and edges, where each new edge is incident with the last vertex in the path and a vertex not already in the path. Continue adding vertices and edges to this path as long as possible. If the path goes through all vertices of the graph, the tree consisting of this path is a spanning tree. However, if the path does not go through all vertices, more vertices and edges must be added. Move back to the next to last vertex in the path, and, if possible, form a new path starting at this vertex passing through vertices that were not already visited. If this cannot be done, move back another vertex in the path, that is, two vertices back in the path, and try again.

Repeat this procedure, beginning at the last vertex visited, moving back up the path one vertex at a time, forming new paths that are as long as possible until no more edges can be added. Because the graph has a finite number of edges and is connected, this process ends with the production of a spanning tree. Each vertex that ends a path at a stage of the algorithm will be a leaf in the rooted tree, and each vertex where a path is constructed starting at this vertex will be an internal vertex.

The reader should note the recursive nature of this procedure. Also, note that if the vertices in the graph are ordered, the choices of edges at each stage of the procedure are all determined when we always choose the first vertex in the ordering that is available. However, we will not always explicitly order the vertices of a graph.

Depth-first search is also called **backtracking**, because the algorithm returns to vertices previously visited to add paths. Example 3 illustrates backtracking.

EXAMPLE 3 Use depth-first search to find a spanning tree for the graph G shown in Figure 6.



Solution: The steps used by depth-first search to produce a spanning tree of G are shown in Figure 7. We arbitrarily start with the vertex f . A path is built by successively adding edges incident with vertices not already in the path, as long as this is possible. This produces a path f, g, h, k, j (note that other paths could have been built). Next, backtrack to k . There is no path

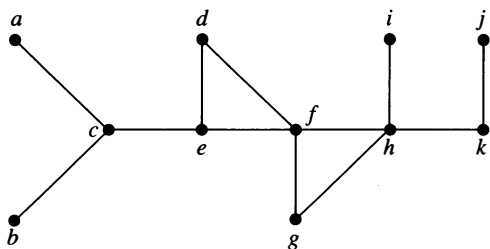


FIGURE 6 The Graph G .

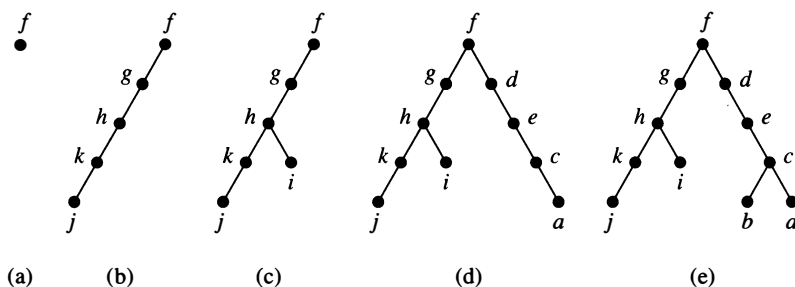


FIGURE 7 Depth-First Search of G .

beginning at k containing vertices not already visited. So we backtrack to h . Form the path h, i . Then backtrack to h , and then to f . From f build the path f, d, e, c, a . Then backtrack to c and form the path c, b . This produces the spanning tree. ◀

The edges selected by depth-first search of a graph are called **tree edges**. All other edges of the graph must connect a vertex to an ancestor or descendant of this vertex in the tree. These edges are called **back edges**. (Exercise 39 asks for a proof of this fact.)

EXAMPLE 4 In Figure 8 we highlight the tree edges found by depth-first search starting at vertex f by showing them with heavy colored lines. The back edges (e, f) and (f, h) are shown with thinner black lines. ◀

We have explained how to find a spanning tree of a graph using depth-first search. However, our discussion so far has not brought out the recursive nature of depth-first search. To help make the recursive nature of the algorithm clear, we need a little terminology. We say that we *explore* from a vertex v when we carry out the steps of depth-first search beginning when v is added to the tree and ending when we have backtracked back to v for the last time. The key observation needed to understand the recursive nature of the algorithm is that when we add an edge connecting a vertex v to a vertex w , we finish exploring from w before we return to v to complete exploring from v .

In Algorithm 1 we construct the spanning tree of a graph G with vertices v_1, \dots, v_n by first selecting the vertex v_1 to be the root. We initially set T to be the tree with just this one vertex. At each step we add a new vertex to the tree T together with an edge from a vertex already in T to this new vertex and we explore from this new vertex. Note that at the completion of the algorithm, T contains no simple circuits because no edge is added that connects a vertex already in the tree. Moreover, T remains connected as it is built. (These last two observations can be easily proved via mathematical induction.) Because G is connected, every vertex in G is visited by the algorithm and is added to the tree (as the reader should verify). It follows that T is a spanning tree of G .

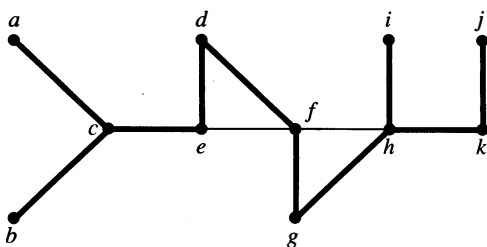


FIGURE 8 The Tree Edges and Back Edges of the Depth-First Search in Example 4.

ALGORITHM 1 Depth-First Search.

```

procedure DFS(G: connected graph with vertices  $v_1, v_2, \dots, v_n$ )
  T := tree consisting only of the vertex  $v_1$ 
  visit( $v_1$ )
  procedure visit(v: vertex of G)
  for each vertex w adjacent to v and not yet in T
  begin
    add vertex w and edge  $\{v, w\}$  to T
    visit(w)
  end

```

We now analyze the computational complexity of the depth-first search algorithm. The key observation is that for each vertex v , the procedure $visit(v)$ is called when the vertex v is first encountered in the search and it is not called again. Assuming that the adjacency lists for G are available (see Section 9.3), no computations are required to find the vertices adjacent to v . As we follow the steps of the algorithm, we examine each edge at most twice to determine whether to add this edge and one of its endpoints to the tree. Consequently, the procedure DFS constructs a spanning tree using $O(e)$, or $O(n^2)$, steps where e and n are the number of edges and vertices in G , respectively. [Note that a step involves examining a vertex to see whether it is already in the spanning tree as it is being built and adding this vertex and the corresponding edge if the vertex is not already in the tree. We have also made use of the inequality $e \leq n(n-1)/2$, which holds for any simple graph.]

Depth-first search can be used as the basis for algorithms that solve many different problems. For example, it can be used to find paths and circuits in a graph, it can be used to determine the connected components of a graph, and it can be used to find the cut vertices of a connected graph. As we will see, depth-first search is the basis of backtracking techniques used to search for solutions of computationally difficult problems. (See [GrYe99], [Ma89], and [CoLeRiSt01] for a discussion of algorithms based on depth-first search.)

Breadth-First Search



Demo

We can also produce a spanning tree of a simple graph by the use of **breadth-first search**. Again, a rooted tree will be constructed, and the underlying undirected graph of this rooted tree forms the spanning tree. Arbitrarily choose a root from the vertices of the graph. Then add all edges incident to this vertex. The new vertices added at this stage become the vertices at level 1 in the spanning tree. Arbitrarily order them. Next, for each vertex at level 1, visited in order, add each edge incident to this vertex to the tree as long as it does not produce a simple circuit. Arbitrarily order the children of each vertex at level 1. This produces the vertices at level 2 in the tree. Follow the same procedure until all the vertices in the tree have been added. The procedure ends because there are only a finite number of edges in the graph. A spanning tree is produced because we have produced a tree containing every vertex of the graph. An example of breadth-first search is given in Example 5.



Links

EXAMPLE 5 Use breadth-first search to find a spanning tree for the graph shown in Figure 9.



Extra Examples

Solution: The steps of the breadth-first search procedure are shown in Figure 10. We choose the vertex e to be the root. Then we add edges incident with all vertices adjacent to e , so edges from e to b , d , f , and i are added. These four vertices are at level 1 in the tree. Next, add the edges from these vertices at level 1 to adjacent vertices not already in the tree. Hence, the edges

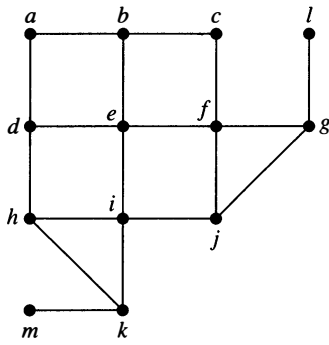


FIGURE 9 A Graph G .

from b to a and c are added, as are edges from d to h , from f to j and g , and from i to k . The new vertices a, c, h, j, g , and k are at level 2. Next, add edges from these vertices to adjacent vertices not already in the graph. This adds edges from g to l and from k to m . ◀

We describe breadth-first search in pseudocode as Algorithm 2. In this algorithm, we assume the vertices of the connected graph G are ordered as v_1, v_2, \dots, v_n . In the algorithm we use the term “process” to describe the procedure of adding new vertices, and corresponding edges, to the tree adjacent to the current vertex being processed as long as a simple circuit is not produced.

ALGORITHM 2 Breadth-First Search.

```

procedure BFS ( $G$ : connected graph with vertices  $v_1, v_2, \dots, v_n$ )
   $T :=$  tree consisting only of vertex  $v_1$ 
   $L :=$  empty list
  put  $v_1$  in the list  $L$  of unprocessed vertices
  while  $L$  is not empty
  begin
    remove the first vertex,  $v$ , from  $L$ 
    for each neighbor  $w$  of  $v$ 
      if  $w$  is not in  $L$  and not in  $T$  then
        begin
          add  $w$  to the end of the list  $L$ 
          add  $w$  and edge  $\{v, w\}$  to  $T$ 
        end
      end
    end
  end
  
```

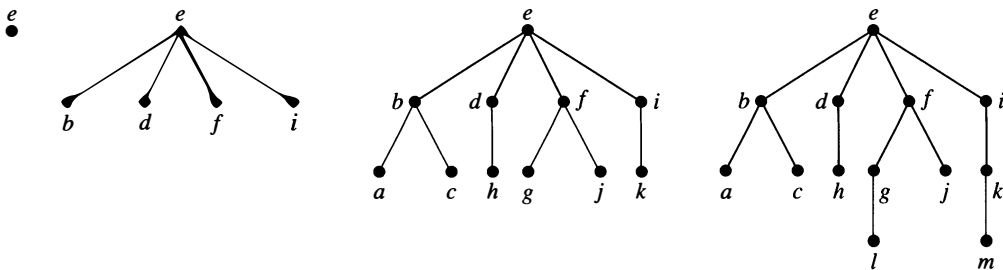


FIGURE 10 Breadth-First Search of G .

We now analyze the computational complexity of breadth-first search. For each vertex v in the graph we examine all vertices adjacent to v and we add each vertex not yet visited to the tree T . Assuming we have the adjacency lists for the graph available, no computation is required to determine which vertices are adjacent to a given vertex. As in the analysis of the depth-first search algorithm, we see that we examine each edge at most twice to determine whether we should add this edge and its endpoint not already in the tree. It follows that the breadth-first search algorithm uses $O(e)$ or $O(n^2)$ steps.

Backtracking Applications

There are problems that can be solved only by performing an exhaustive search of all possible solutions. One way to search systematically for a solution is to use a decision tree, where each internal vertex represents a decision and each leaf a possible solution. To find a solution via backtracking, first make a sequence of decisions in an attempt to reach a solution as long as this is possible. The sequence of decisions can be represented by a path in the decision tree. Once it is known that no solution can result from any further sequence of decisions, backtrack to the parent of the current vertex and work toward a solution with another series of decisions, if this is possible. The procedure continues until a solution is found, or it is established that no solution exists. Examples 6 to 8 illustrate the usefulness of backtracking.

EXAMPLE 6 Graph Colorings How can backtracking be used to decide whether a graph can be colored using n colors?

Solution: We can solve this problem using backtracking in the following way. First pick some vertex a and assign it color 1. Then pick a second vertex b , and if b is not adjacent to a , assign it color 1. Otherwise, assign color 2 to b . Then go on to a third vertex c . Use color 1, if possible, for c . Otherwise use color 2, if this is possible. Only if neither color 1 nor color 2 can be used should color 3 be used. Continue this process as long as it is possible to assign one of the n colors to each additional vertex, always using the first allowable color in the list. If a vertex is reached that cannot be colored by any of the n colors, backtrack to the last assignment made and change the coloring of the last vertex colored, if possible, using the next allowable color in the list. If it is not possible to change this coloring, backtrack farther to previous assignments, one step back at a time, until it is possible to change a coloring of a vertex. Then continue assigning colors of additional vertices as long as possible. If a coloring using n colors exists, backtracking will produce it. (Unfortunately this procedure can be extremely inefficient.)

In particular, consider the problem of coloring the graph shown in Figure 11 with three colors. The tree shown in Figure 11 illustrates how backtracking can be used to construct a 3-coloring. In this procedure, red is used first, then blue, and finally green. This simple example can obviously be done without backtracking, but it is a good illustration of the technique.

In this tree, the initial path from the root, which represents the assignment of red to a , leads to a coloring with a red, b blue, c red, and d green. It is impossible to color e using any of the three colors when a , b , c , and d are colored in this way. So, backtrack to the parent of the vertex representing this coloring. Because no other color can be used for d , backtrack one more level. Then change the color of c to green. We obtain a coloring of the graph by then assigning red to d and green to e . ◀

EXAMPLE 7 The n -Queens Problem The n -queens problem asks how n queens can be placed on an $n \times n$ chessboard so that no two queens can attack one another. How can backtracking be used to solve the n -queens problem?



Solution: To solve this problem we must find n positions on an $n \times n$ chessboard so that no two of these positions are in the same row, same column, or in the same diagonal [a diagonal consists of all positions (i, j) with $i + j = m$ for some m , or $i - j = m$ for some m]. We will

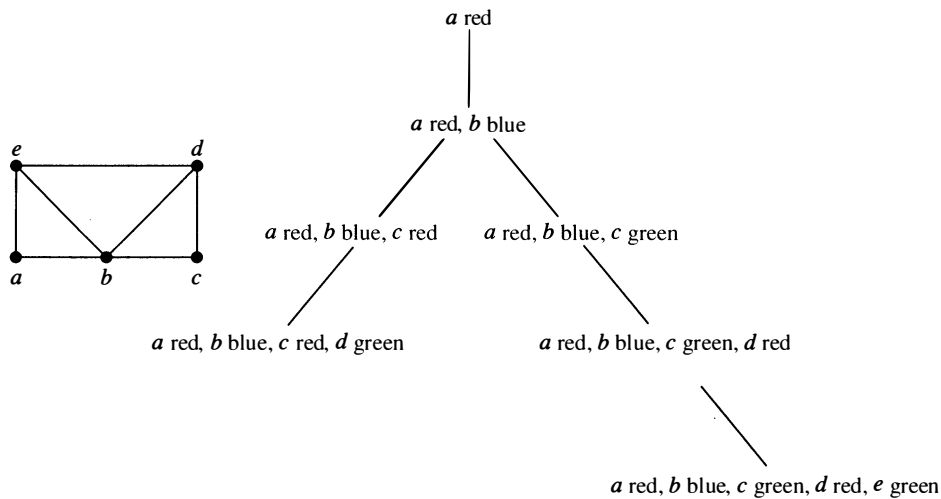


FIGURE 11 Coloring a Graph Using Backtracking.

use backtracking to solve the n -queens problem. We start with an empty chessboard. At stage $k + 1$ we attempt putting an additional queen on the board in the $(k + 1)$ st column, where there are already queens in the first k columns. We examine squares in the $(k + 1)$ st column starting with the square in the first row, looking for a position to place this queen so that it is not in the same row or on the same diagonal as a queen already on the board. (We already know it is not in the same column.) If it is impossible to find a position to place the queen in the $(k + 1)$ st column, backtrack to the placement of the queen in the k th column, and place this queen in the next allowable row in this column, if such a row exists. If no such row exists, backtrack further.

In particular, Figure 12 displays a backtracking solution to the four-queens problem. In this solution, we place a queen in the first row and column. Then we put a queen in the third row of the second column. However, this makes it impossible to place a queen in the third column. So we backtrack and put a queen in the fourth row of the second column. When we do this, we can place a queen in the second row of the third column. But there is no way to add a queen to the fourth column. This shows that no solution results when a queen is placed in the first row and column. We backtrack to the empty chessboard, and place a queen in the second row of the first column. This leads to a solution as shown in Figure 12. ◀

EXAMPLE 8 **Sums of Subsets** Consider this problem. Given a set of positive integers x_1, x_2, \dots, x_n , find a subset of this set of integers that has M as its sum. How can backtracking be used to solve this problem?

Solution: We start with a sum with no terms. We build up the sum by successively adding terms. An integer in the sequence is included if the sum remains less than M when this integer is added to the sum. If a sum is reached such that the addition of any term is greater than M , backtrack by dropping the last term of the sum.

Figure 13 displays a backtracking solution to the problem of finding a subset of $\{31, 27, 15, 11, 7, 5\}$ with the sum equal to 39. ◀

Depth-First Search in Directed Graphs

We can easily modify both depth-first search and breadth-first search so that they can run given a directed graph as input. However, the output will not necessarily be a spanning tree, but rather a spanning forest. In both algorithms we can add an edge only when it is directed away from

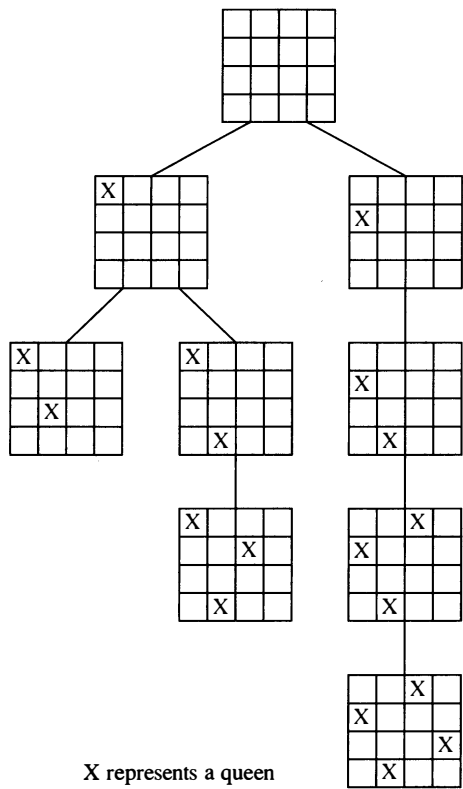


FIGURE 12 A Backtracking Solution of the Four-Queens Problem.

the vertex that is being visited and to a vertex not yet added. If at a stage of either algorithm we find that no edge exists starting at a vertex already added to one not yet added, the next vertex added by the algorithm becomes the root of a new tree in the spanning forest. This is illustrated in Example 9.

EXAMPLE 9 What is the output of depth-first search given the graph G shown in Figure 14(a) as input?

Solution: We begin the depth-first search at vertex a and add vertices b , c , and g and the corresponding edges where we are blocked. We backtrack to c but we are still blocked, and then

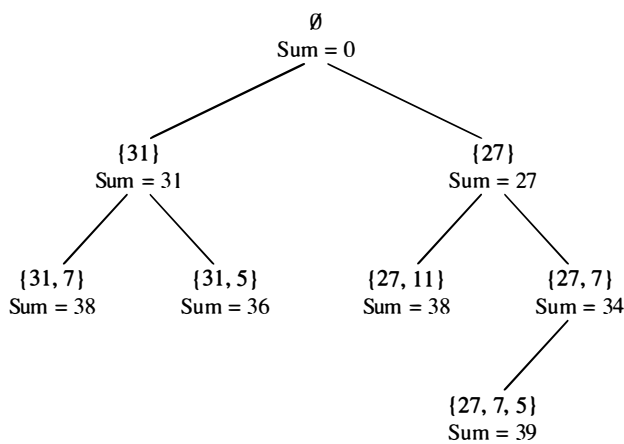


FIGURE 13 Find a Sum Equal to 39 Using Backtracking.

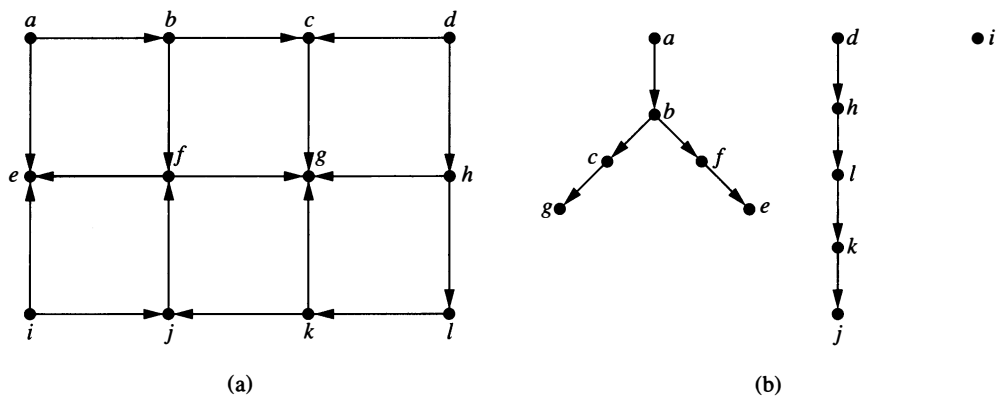


FIGURE 14 Depth-First Search of a Directed Graph.

backtrack to b , where we add vertices f and e and the corresponding edges. Backtracking takes us all the way back to a . We then start a new tree at d and add vertices h , l , k , and j and the corresponding edges. We backtrack to k , then l , then h , and back to d . Finally, we start a new tree at i , completing the depth-first search. The output is shown in Figure 14(b). ◀

Depth-first search in directed graphs is the basis of many algorithms (see [GrYe99], [Ma89], and [CoLeRiSt01]). It can be used to determine whether a directed graph has a circuit, it can be used to carry out a topological sort of a graph, and it can also be used to find the strongly connected components of a directed graph.

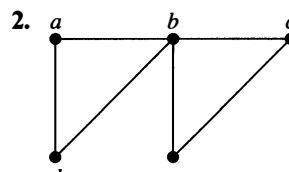
We conclude this section with an application of depth-first search and breadth-first search to search engines on the Web.

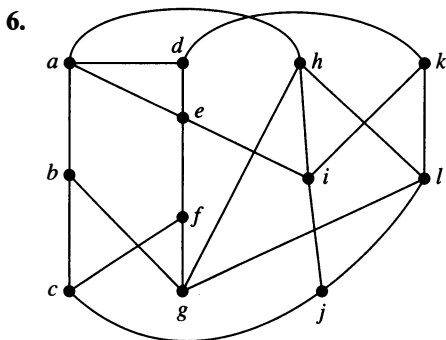
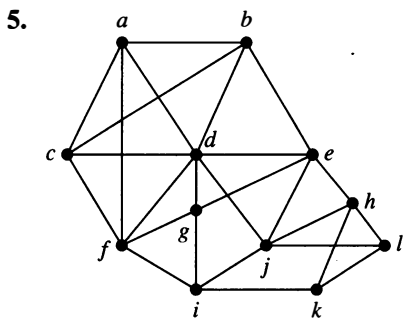
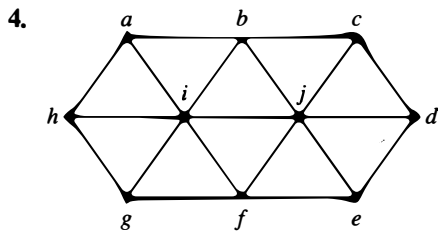
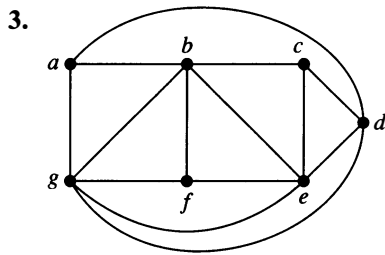
EXAMPLE 10 Web Spiders To index websites, search engines such as Google and Yahoo systematically explore the Web starting at known sites. These search engines use programs called Web spiders (or crawlers or bots) to visit websites and analyze their contents. Web spiders use both depth-first searching and breadth-first searching to create indices. As described in Example 8 in Section 9.1, Web pages and links between them can be modeled by a directed graph called the Web graph. Web pages are represented by vertices and links are represented by directed edges. Using depth-first search, an initial Web page is selected, a link is followed to a second Web page (if there is such a link), a link on the second Web page is followed to a third Web page, if there is such a link, and so on, until a page with no new links is found. Backtracking is then used to examine links at the previous level to look for new links, and so on. (Because of practical limitations, Web spiders have limits to the depth they search in depth-first search.) Using breadth-first search, an initial Web page is selected and a link on this page is followed to a second Web page, then a second link on the initial page is followed (if it exists), and so on, until all links of the initial page have been followed. Then links on the pages one level down are followed, page by page, and so on. ◀

Exercises

- How many edges must be removed from a connected graph with n vertices and m edges to produce a spanning tree?

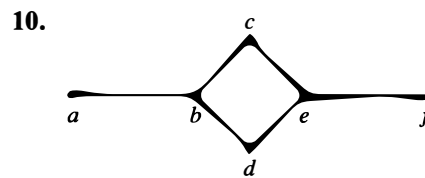
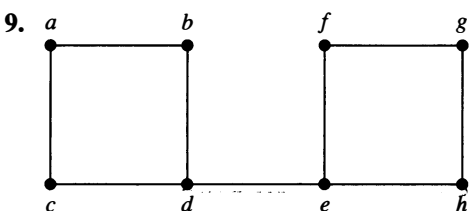
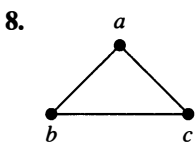
In Exercises 2–6 find a spanning tree for the graph shown by removing edges in simple circuits.





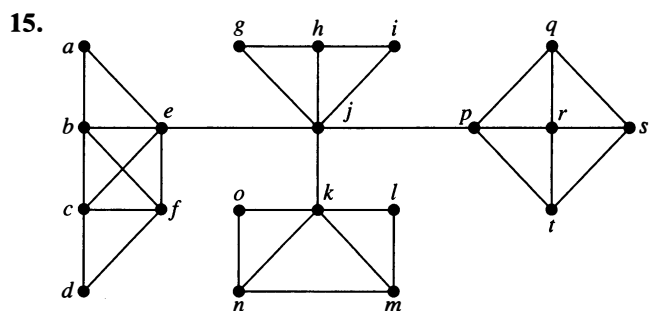
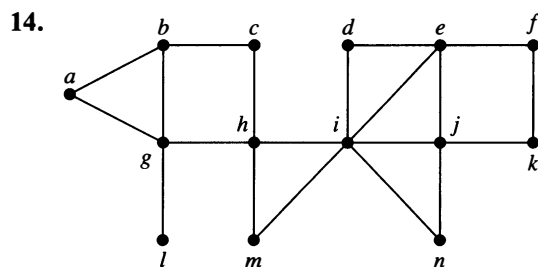
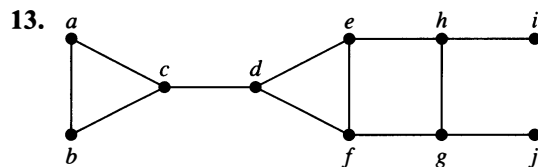
7. Find a spanning tree for each of these graphs.
 a) K_5 b) $K_{4,4}$ c) $K_{1,6}$
 d) Q_3 e) C_5 f) W_5

In Exercises 8–10 draw all the spanning trees of the given simple graphs.



- *11. How many different spanning trees does each of these simple graphs have?
 a) K_3 b) K_4 c) $K_{2,2}$ d) C_5
 *12. How many nonisomorphic spanning trees does each of these simple graphs have?
 a) K_3 b) K_4 c) K_5

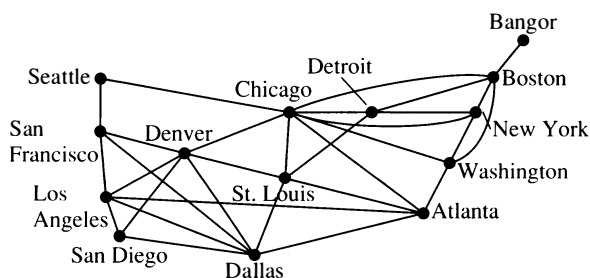
In Exercises 13–15 use depth-first search to produce a spanning tree for the given simple graph. Choose a as the root of this spanning tree and assume that the vertices are ordered alphabetically.



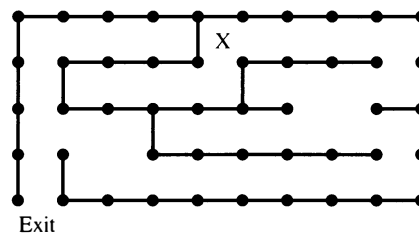
16. Use breadth-first search to produce a spanning tree for each of the simple graphs in Exercises 13–15. Choose a as the root of each spanning tree.
 17. Use depth-first search to find a spanning tree of each of these graphs.
 a) W_6 (see Example 7 of Section 9.2), starting at the vertex of degree 6
 b) K_5
 c) $K_{3,4}$, starting at a vertex of degree 3
 d) Q_3
 18. Use breadth-first search to find a spanning tree of each of the graphs in Exercise 17.
 19. Describe the trees produced by breadth-first search and depth-first search of the wheel graph W_n , starting at the

vertex of degree n , where n is an integer with $n \geq 3$. (See Example 7 of Section 9.2.) Justify your answers.

20. Describe the trees produced by breadth-first search and depth-first search of the complete graph K_n , where n is a positive integer. Justify your answers.
21. Describe the trees produced by breadth-first search and depth-first search of the complete bipartite graph $K_{m,n}$, starting at a vertex of degree m , where m and n are positive integers. Justify your answers.
22. Explain how breadth-first search and how depth-first search can be used to determine whether a graph is bipartite.
23. Suppose that an airline must reduce its flight schedule to save money. If its original routes are as illustrated here, which flights can be discontinued to retain service between all pairs of cities (where it may be necessary to combine flights to fly from one city to another)?



24. When must an edge of a connected simple graph be in every spanning tree for this graph?
25. Which connected simple graphs have exactly one spanning tree?
26. Explain how breadth-first search or depth-first search can be used to order the vertices of a connected graph.
- *27. Show that the length of the shortest path between vertices v and u in a connected simple graph equals the level number of u in the breadth-first spanning tree of G with root v .
28. Use backtracking to try to find a coloring of each of the graphs in Exercises 7–9 of Section 9.8 using three colors.
29. Use backtracking to solve the n -queens problem for these values of n .
 a) $n = 3$ b) $n = 5$ c) $n = 6$
30. Use backtracking to find a subset, if it exists, of the set $\{27, 24, 19, 14, 11, 8\}$ with sum
 a) 20. b) 41. c) 60.
31. Explain how backtracking can be used to find a Hamilton path or circuit in a graph.
32. a) Explain how backtracking can be used to find the way out of a maze, given a starting position and the exit position. Consider the maze divided into positions, where at each position the set of available moves includes one to four possibilities (up, down, right, left).
 b) Find a path from the starting position marked by X to the exit in this maze.



A **spanning forest** of graph G is a forest that contains every vertex of G such that two vertices are in the same tree of the forest when there is a path in G between these two vertices.

33. Show that every finite simple graph has a spanning forest.
34. How many trees are in the spanning forest of a graph?
35. How many edges must be removed to produce the spanning forest of a graph with n vertices, m edges, and c connected components?
36. Devise an algorithm for constructing the spanning forest of a graph based on deleting edges that form simple circuits.
37. Devise an algorithm for constructing the spanning forest of a graph based on depth-first searching.
38. Devise an algorithm for constructing the spanning forest of a graph based on breadth-first searching.
39. Let G be a connected graph. Show that if T is a spanning tree of G constructed using depth-first search, then an edge of G not in T must be a back edge, that is, it must connect a vertex to one of its ancestors or one of its descendants in T .
40. Let G be a connected graph. Show that if T is a spanning tree of G constructed using breadth-first search, then an edge of G not in T must connect vertices at the same level or at levels that differ by 1 in this spanning tree.
41. For which graphs do depth-first search and breadth-first search produce identical spanning trees no matter which vertex is selected as the root of the tree? Justify your answer.
42. Use Exercise 39 to prove that if G is a connected, simple graph with n vertices and G does not contain a simple path of length k then it contains at most $(k - 1)n$ edges.
43. Use mathematical induction to prove that breadth-first search visits vertices in order of their level in the resulting spanning tree.
44. Use pseudocode to describe a variation of depth-first search that assigns the integer n to the n th vertex visited in the search. Show that this numbering corresponds to the numbering of the vertices created by a preorder traversal of the spanning tree.
45. Use pseudocode to describe a variation of breadth-first search that assigns the integer m to the m th vertex visited in the search.
- *46. Suppose that G is a directed graph and T is a spanning tree constructed using breadth-first search. Show that every edge of G connects two vertices at the same level, a vertex to a vertex at one level lower, or a vertex to a vertex at some higher level.

47. Show that if G is a directed graph and T is a spanning tree constructed using depth-first search, then every edge not in the spanning tree is a **forward edge** connecting an ancestor to a descendant, a **back edge** connecting a descendant to an ancestor, or a **cross edge** connecting a vertex to a vertex in a previously visited subtree.
- *48. Describe a variation of depth-first search that assigns the smallest available positive integer to a vertex when the algorithm is totally finished with this vertex. Show that in this numbering, each vertex has a larger number than its children and that the children have increasing numbers from left to right.
- Let T_1 and T_2 be spanning trees of a graph. The **distance** between T_1 and T_2 is the number of edges in T_1 and T_2 that are not common to T_1 and T_2 .
49. Find the distance between each pair of spanning trees shown in Figures 3(c) and 4 of the graph G shown in Figure 2.
- *50. Suppose that T_1 , T_2 , and T_3 are spanning trees of the simple graph G . Show that the distance between T_1 and T_3 does not exceed the sum of the distance between T_1 and T_2 and the distance between T_2 and T_3 .
- **51. Suppose that T_1 and T_2 are spanning trees of a simple graph G . Moreover, suppose that e_1 is an edge in T_1 that is not in T_2 . Show that there is an edge e_2 in T_2 that is not in T_1 such that T_1 remains a spanning tree if e_1 is removed from it and e_2 is added to it, and T_2 remains a spanning tree if e_2 is removed from it and e_1 is added to it.
- *52. Show that it is possible to find a sequence of spanning trees leading from any spanning tree to any other by successively removing one edge and adding another.
- A **rooted spanning tree** of a directed graph is a rooted tree containing edges of the graph such that every vertex of the graph is an endpoint of one of the edges in the tree.
53. For each of the directed graphs in Exercises 18–23 of Section 9.5 either find a rooted spanning tree of the graph or determine that no such tree exists.
- *54. Show that a connected directed graph in which each vertex has the same in-degree and out-degree has a rooted spanning tree. [Hint: Use an Euler circuit.]
- *55. Give an algorithm to build a rooted spanning tree for connected directed graphs in which each vertex has the same in-degree and out-degree.
- *56. Show that if G is a directed graph and T is a spanning tree constructed using depth-first search, then G contains a circuit if and only if G contains a back edge (see Exercise 47) relative to the spanning tree T .
- *57. Use Exercise 56 to construct an algorithm for determining whether a directed graph contains a circuit.

10.5 Minimum Spanning Trees

Introduction



A company plans to build a communications network connecting its five computer centers. Any pair of these centers can be linked with a leased telephone line. Which links should be made to ensure that there is a path between any two computer centers so that the total cost of the network is minimized? We can model this problem using the weighted graph shown in Figure 1, where vertices represent computer centers, edges represent possible leased lines, and the weights on edges are the monthly lease rates of the lines represented by the edges. We can solve this problem by finding a spanning tree so that the sum of the weights of the edges of the tree is minimized. Such a spanning tree is called a **minimum spanning tree**.

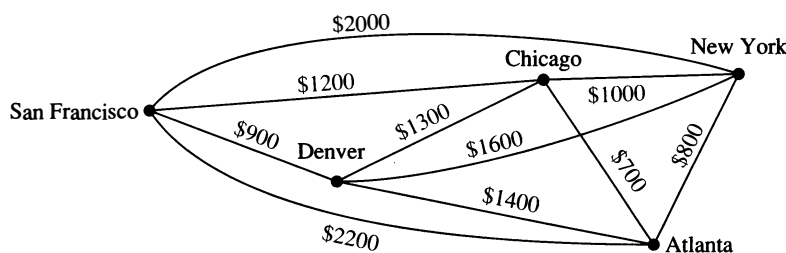


FIGURE 1 A Weighted Graph Showing Monthly Lease Costs for Lines in a Computer Network.

Algorithms for Minimum Spanning Trees

A wide variety of problems are solved by finding a spanning tree in a weighted graph such that the sum of the weights of the edges in the tree is a minimum.

DEFINITION 1 A *minimum spanning tree* in a connected weighted graph is a spanning tree that has the smallest possible sum of weights of its edges.



We will present two algorithms for constructing minimum spanning trees. Both proceed by successively adding edges of smallest weight from those edges with a specified property that have not already been used. Both are greedy algorithms. Recall from Section 3.1 that a greedy algorithm is a procedure that makes an optimal choice at each of its steps. Optimizing at each step does not guarantee that the optimal overall solution is produced. However, the two algorithms presented in this section for constructing minimum spanning trees are greedy algorithms that do produce optimal solutions.



The first algorithm that we will discuss was given by Robert Prim in 1957, although the basic ideas of this algorithm have an earlier origin. To carry out **Prim's algorithm**, begin by choosing any edge with smallest weight, putting it into the spanning tree. Successively add to the tree edges of minimum weight that are incident to a vertex already in the tree and not forming a simple circuit with those edges already in the tree. Stop when $n - 1$ edges have been added.

Later in this section, we will prove that this algorithm produces a minimum spanning tree for any connected weighted graph. Algorithm 1 gives a pseudocode description of Prim's algorithm.

ALGORITHM 1 Prim's Algorithm.

```

procedure Prim( $G$ : weighted connected undirected graph with  $n$  vertices)
 $T :=$  a minimum-weight edge
for  $i := 1$  to  $n - 2$ 
begin
   $e :=$  an edge of minimum weight incident to a vertex in  $T$  and not forming a
  simple circuit in  $T$  if added to  $T$ 
   $T := T$  with  $e$  added
end { $T$  is a minimum spanning tree of  $G$ }
  
```

Note that the choice of an edge to add at a stage of the algorithm is not determined when there is more than one edge with the same weight that satisfies the appropriate criteria. We need to order the edges to make the choices deterministic. We will not worry about this in the remainder of the section. Also note that there may be more than one minimum spanning tree for a given



ROBERT CLAY PRIM (BORN 1921) Robert Prim, born in Sweetwater, Texas, received his B.S. in electrical engineering in 1941 and his Ph.D. in mathematics from Princeton University in 1949. He was an engineer at the General Electric Company from 1941 until 1944, an engineer and mathematician at the United States Naval Ordnance Lab from 1944 until 1949, and a research associate at Princeton University from 1948 until 1949. Among the other positions he has held are director of mathematics and mechanics research at Bell Telephone Laboratories from 1958 until 1961 and vice president of research at Sandia Corporation. He is currently retired.

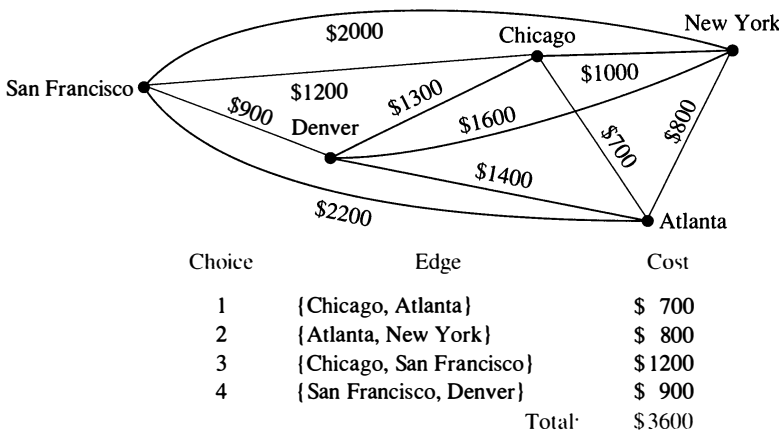


FIGURE 2 A Minimum Spanning Tree for the Weighted Graph in Figure 1.

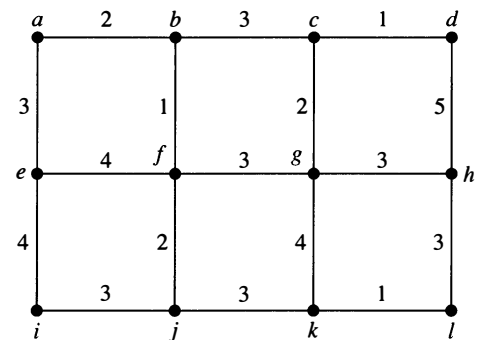


FIGURE 3 A Weighted Graph.

connected weighted simple graph. (See Exercise 9.) Examples 1 and 2 illustrate how Prim's algorithm is used.

EXAMPLE 1 Use Prim's algorithm to design a minimum-cost communications network connecting all the computers represented by the graph in Figure 1.

Solution: We solve this problem by finding a minimum spanning tree in the graph in Figure 1. Prim's algorithm is carried out by choosing an initial edge of minimum weight and successively adding edges of minimum weight that are incident to a vertex in the tree and that do not form simple circuits. The edges in color in Figure 2 show a minimum spanning tree produced by Prim's algorithm, with the choice made at each step displayed. ◀

EXAMPLE 2 Use Prim's algorithm to find a minimum spanning tree in the graph shown in Figure 3.

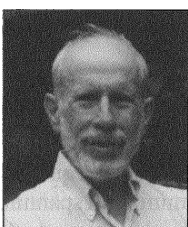
Solution: A minimum spanning tree constructed using Prim's algorithm is shown in Figure 4. The successive edges chosen are displayed. ◀



The second algorithm we will discuss was discovered by Joseph Kruskal in 1956, although the basic ideas it uses were described much earlier. To carry out **Kruskal's algorithm**, choose an edge in the graph with minimum weight.

Successively add edges with minimum weight that do not form a simple circuit with those edges already chosen. Stop after $n - 1$ edges have been selected.

The proof that Kruskal's algorithm produces a minimum spanning tree for every connected weighted graph is left as an exercise at the end of this section. Pseudocode for Kruskal's algorithm is given in Algorithm 2.



JOSEPH BERNARD KRUSKAL (BORN 1928) Joseph Kruskal, born in New York City, attended the University of Chicago and received his Ph.D. from Princeton University in 1954. He was an instructor in mathematics at Princeton and at the University of Wisconsin, and later he was an assistant professor at the University of Michigan. In 1959 he became a member of the technical staff at Bell Laboratories, a position he continues to hold. His current research interests include statistical linguistics and psychometrics. Besides his work on minimum spanning trees, Kruskal is also known for contributions to multidimensional scaling. Kruskal discovered his algorithm for producing minimum spanning trees when he was a second-year graduate student. He was not sure his $2\frac{1}{2}$ -page paper on this subject was worthy of publication, but was convinced by others to submit it.

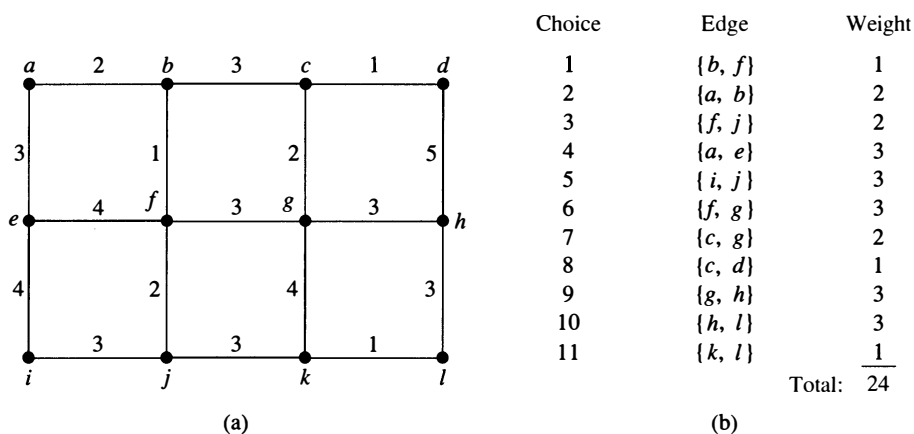


FIGURE 4 A Minimum Spanning Tree Produced Using Prim's Algorithm.

ALGORITHM 2 Kruskal's Algorithm.

```

procedure Kruskal( $G$ : weighted connected undirected graph with  $n$  vertices)
 $T :=$  empty graph
for  $i := 1$  to  $n - 1$ 
begin
   $e :=$  any edge in  $G$  with smallest weight that does not form a simple circuit
  when added to  $T$ 
   $T := T$  with  $e$  added
end ( $T$  is a minimum spanning tree of  $G$ )
  
```

The reader should note the difference between Prim's and Kruskal's algorithms. In Prim's algorithm edges of minimum weight that are incident to a vertex already in the tree, and not forming a circuit, are chosen; whereas in Kruskal's algorithm edges of minimum weight that are not necessarily incident to a vertex already in the tree, and that do not form a circuit, are chosen. Note that as in Prim's algorithm, if the edges are not ordered, there may be more than one choice for the edge to add at a stage of this procedure. Consequently, the edges need to be ordered for the procedure to be deterministic. Example 3 illustrates how Kruskal's algorithm is used.

EXAMPLE 3 Use Kruskal's algorithm to find a minimum spanning tree in the weighted graph shown in Figure 3.



Solution: A minimum spanning tree and the choices of edges at each stage of Kruskal's algorithm are shown in Figure 5. ◀

HISTORICAL NOTE Joseph Kruskal and Robert Prim developed their algorithms for constructing minimum spanning trees in the mid-1950s. However, they were not the first people to discover such algorithms. For example, the work of the anthropologist Jan Czekanowski, in 1909, contains many of the ideas required to find minimum spanning trees. In 1926, Otakar Boruvka described methods for constructing minimum spanning trees in work relating to the construction of electric power networks.

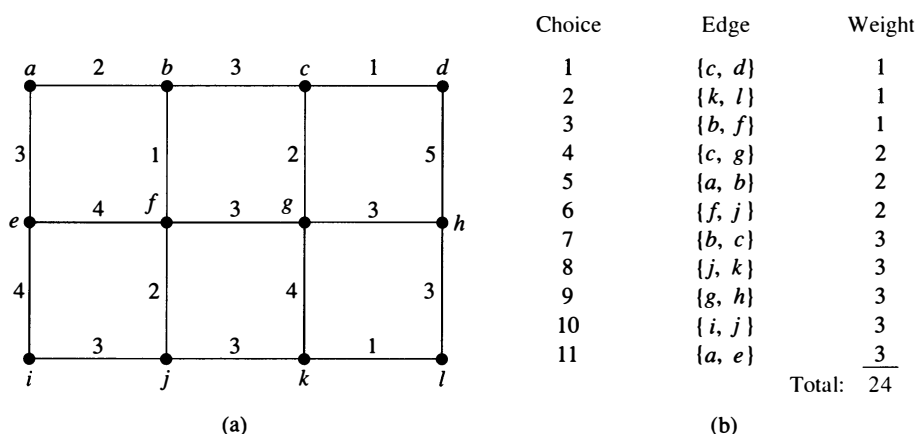


FIGURE 5 A Minimum Spanning Tree Produced by Kruskal's Algorithm.

We will now prove that Prim's algorithm produces a minimum spanning tree of a connected weighted graph.

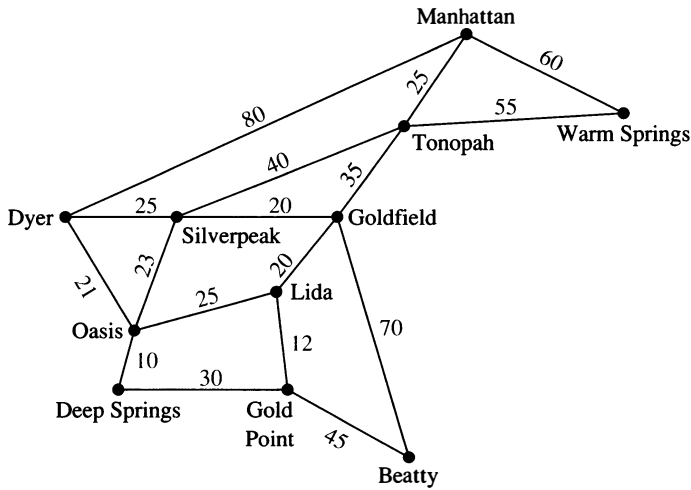
Proof: Let G be a connected weighted graph. Suppose that the successive edges chosen by Prim's algorithm are e_1, e_2, \dots, e_{n-1} . Let S be the tree with e_1, e_2, \dots, e_{n-1} as its edges, and let S_k be the tree with e_1, e_2, \dots, e_k as its edges. Let T be a minimum spanning tree of G containing the edges e_1, e_2, \dots, e_k , where k is the maximum integer with the property that a minimum spanning tree exists containing the first k edges chosen by Prim's algorithm. The theorem follows if we can show that $S = T$.

Suppose that $S \neq T$, so that $k < n - 1$. Consequently, T contains e_1, e_2, \dots, e_k , but not e_{k+1} . Consider the graph made up of T together with e_{k+1} . Because this graph is connected and has n edges, too many edges to be a tree, it must contain a simple circuit. This simple circuit must contain e_{k+1} because there was no simple circuit in T . Furthermore, there must be an edge in the simple circuit that does not belong to S_{k+1} because S_{k+1} is a tree. By starting at an endpoint of e_{k+1} that is also an endpoint of one of the edges e_1, \dots, e_k , and following the circuit until it reaches an edge not in S_{k+1} , we can find an edge e not in S_{k+1} that has an endpoint that is also an endpoint of one of the edges e_1, e_2, \dots, e_k . By deleting e from T and adding e_{k+1} , we obtain a tree T' with $n - 1$ edges (it is a tree because it has no simple circuits). Note that the tree T' contains $e_1, e_2, \dots, e_k, e_{k+1}$. Furthermore, because e_{k+1} was chosen by Prim's algorithm at the $(k + 1)$ st step, and e was also available at that step, the weight of e_{k+1} is less than or equal to the weight of e . From this observation it follows that T' is also a minimum spanning tree, because the sum of the weights of its edges does not exceed the sum of the weights of the edges of T . This contradicts the choice of k as the maximum integer such that a minimum spanning tree exists containing e_1, \dots, e_k . Hence, $k = n - 1$, and $S = T$. It follows that Prim's algorithm produces a minimum spanning tree. ◁

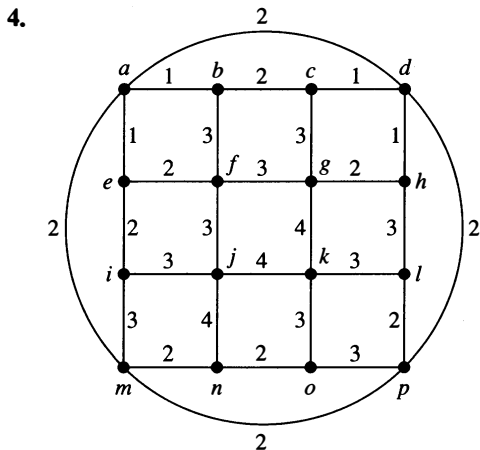
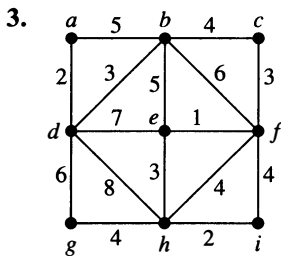
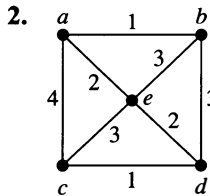
It can be shown (see [CoLeRiSt01]) that to find a minimum spanning tree of a graph with e edges and v vertices, Kruskal's algorithm can be carried out using $O(e \log e)$ operations and Prim's algorithm can be carried out using $O(e \log v)$ operations. Consequently, it is preferable to use Kruskal's algorithm for graphs that are **sparse**, that is, where e is very small compared to $C(v, 2) = v(v - 1)/2$, the total number of possible edges in an undirected graph with v vertices. Otherwise, there is little difference in the complexity of these two algorithms.

Exercises

1. The roads represented by this graph are all unpaved. The lengths of the roads between pairs of towns are represented by edge weights. Which roads should be paved so that there is a path of paved roads between each pair of towns so that a minimum road length is paved? (*Note:* These towns are in Nevada.)



In Exercises 2–4 use Prim’s algorithm to find a minimum spanning tree for the given weighted graph.



5. Use Kruskal’s algorithm to design the communications network described at the beginning of the section.
6. Use Kruskal’s algorithm to find a minimum spanning tree for the weighted graph in Exercise 2.
7. Use Kruskal’s algorithm to find a minimum spanning tree for the weighted graph in Exercise 3.
8. Use Kruskal’s algorithm to find a minimum spanning tree for the weighted graph in Exercise 4.
9. Find a connected weighted simple graph with the fewest edges possible that has more than one minimum spanning tree.
10. A **minimum spanning forest** in a weighted graph is a spanning forest with minimal weight. Explain how Prim’s and Kruskal’s algorithms can be adapted to construct minimum spanning forests.
- A **maximum spanning tree** of a connected weighted undirected graph is a spanning tree with the largest possible weight.
11. Devise an algorithm similar to Prim’s algorithm for constructing a maximum spanning tree of a connected weighted graph.
12. Devise an algorithm similar to Kruskal’s algorithm for constructing a maximum spanning tree of a connected weighted graph.
13. Find a maximum spanning tree for the weighted graph in Exercise 2.
14. Find a maximum spanning tree for the weighted graph in Exercise 3.
15. Find a maximum spanning tree for the weighted graph in Exercise 4.
16. Find the second least expensive communications network connecting the five computer centers in the problem posed at the beginning of the section.
- *17. Devise an algorithm for finding the second shortest spanning tree in a connected weighted graph.
- *18. Show that an edge with smallest weight in a connected weighted graph must be part of any minimum spanning tree.
19. Show that there is a unique minimum spanning tree in a connected weighted graph if the weights of the edges are all different.
20. Suppose that the computer network connecting the cities in Figure 1 must contain a direct link between New York and Denver. What other links should be included so that there is a link between every two computer centers and the cost is minimized?
21. Find a spanning tree with minimal total weight containing the edges $\{e, i\}$ and $\{g, k\}$ in the weighted graph in Figure 3.
22. Describe an algorithm for finding a spanning tree with minimal weight containing a specified set of edges in a connected weighted undirected simple graph.

23. Express the algorithm devised in Exercise 22 in pseudocode.

Sollin's algorithm produces a minimum spanning tree from a connected weighted simple graph $G = (V, E)$ by successively adding groups of edges. Suppose that the vertices in V are ordered. This produces an ordering of the edges where $\{u_0, v_0\}$ precedes $\{u_1, v_1\}$ if u_0 precedes u_1 or if $u_0 = u_1$ and v_0 precedes v_1 . The algorithm begins by simultaneously choosing the edge of least weight incident to each vertex. The first edge in the ordering is taken in the case of ties. This produces a graph with no simple circuits, that is, a forest of trees (Exercise 24 asks for a proof of this fact). Next, simultaneously choose for each tree in the forest the shortest edge between a vertex in this tree and a vertex in a different tree. Again the first edge in the ordering is chosen in the case of ties. (This produces a graph with no simple circuits containing fewer trees than were present before this step; see Exercise 24.) Continue the process of simultaneously adding edges connecting trees until $n - 1$ edges have been chosen. At this stage a minimum spanning tree has been constructed.

*24. Show that the addition of edges at each stage of Sollin's algorithm produces a forest.

25. Use Sollin's algorithm to produce a minimum spanning tree for the weighted graph shown in

- a) Figure 1.
- b) Figure 3.

*26. Express Sollin's algorithm in pseudocode.

**27. Prove that Sollin's algorithm produces a minimum spanning tree in a connected undirected weighted graph.

*28. Show that the first step of Sollin's algorithm produces a forest containing at least $\lceil n/2 \rceil$ edges.

*29. Show that if there are r trees in the forest at some intermediate step of Sollin's algorithm, then at least $\lceil r/2 \rceil$ edges are added by the next iteration of the algorithm.

*30. Show that no more than $\lfloor n/2^k \rfloor$ trees remain after the first step of Sollin's algorithm has been carried out and the second step of the algorithm has been carried out $k - 1$ times.

*31. Show that Sollin's algorithm requires at most $\log n$ iterations to produce a minimum spanning tree from a connected undirected weighted graph with n vertices.

32. Prove that Kruskal's algorithm produces minimum spanning trees.

Key Terms and Results

TERMS

tree: a connected undirected graph with no simple circuits

forest: an undirected graph with no simple circuits

rooted tree: a directed graph with a specified vertex, called the root, such that there is a unique path to any other vertex from this root

subtree: a subgraph of a tree that is also a tree

parent of v in a rooted tree: the vertex u such that (u, v) is an edge of the rooted tree

child of a vertex v in a rooted tree: any vertex with v as its parent

sibling of a vertex v in a rooted tree: a vertex with the same parent as v

ancestor of a vertex v in a rooted tree: any vertex on the path from the root to v

descendant of a vertex v in a rooted tree: any vertex that has v as an ancestor

internal vertex: a vertex that has children

leaf: a vertex with no children

level of a vertex: the length of the path from the root to this vertex

height of a tree: the largest level of the vertices of a tree

m -ary tree: a tree with the property that every internal vertex has no more than m children

full m -ary tree: a tree with the property that every internal vertex has exactly m children

binary tree: an m -ary tree with $m = 2$ (each child may be designated as a left or a right child of its parent)

ordered tree: a tree in which the children of each internal vertex are linearly ordered

balanced tree: a tree in which every leaf is at level h or $h - 1$, where h is the height of the tree

binary search tree: a binary tree in which the vertices are labeled with items so that a label of a vertex is greater than the labels of all vertices in the left subtree of this vertex and is less than the labels of all vertices in the right subtree of this vertex

decision tree: a rooted tree where each vertex represents a possible outcome of a decision and the leaves represent the possible solutions

prefix code: a code that has the property that the code of a character is never a prefix of the code of another character

minmax strategy: the strategy where the first player and second player move to positions represented by a child with maximum and minimum value, respectively

value of a vertex in a game tree: for a leaf, the payoff to the first player when the game terminates in the position represented by this leaf; for an internal vertex, the maximum or minimum of the values of its children, for an internal vertex at an even or odd level, respectively

tree traversal: a listing of the vertices of a tree

preorder traversal: a listing of the vertices of an ordered rooted tree defined recursively—the root is listed, followed by the first subtree, followed by the other subtrees in the order they occur from left to right

inorder traversal: a listing of the vertices of an ordered rooted tree defined recursively—the first subtree is listed, followed by the root, followed by the other subtrees in the order they occur from left to right

postorder traversal: a listing of the vertices of an ordered rooted tree defined recursively—the subtrees are listed in

the order they occur from left to right, followed by the root

infix notation: the form of an expression (including a full set of parentheses) obtained from an inorder traversal of the binary tree representing this expression

prefix (or Polish) notation: the form of an expression obtained from a preorder traversal of the tree representing this expression

postfix (or reverse Polish) notation: the form of an expression obtained from a postorder traversal of the tree representing this expression

spanning tree: a tree containing all vertices of a graph

minimum spanning tree: a spanning tree with smallest possible sum of weights of its edges

greedy algorithm: an algorithm that optimizes by making the optimal choice at each step

RESULTS

A graph is a tree if and only if there is a unique simple path between any of its vertices.

A tree with n vertices has $n - 1$ edges.

A full m -ary tree with i internal vertices has $mi + 1$ vertices.

The relationships among the numbers of vertices, leaves, and internal vertices in a full m -ary tree (see Theorem 4 in Section 10.1)

There are at most m^h leaves in an m -ary tree of height h .

If an m -ary tree has l leaves, its height h is at least $\lceil \log_m l \rceil$. If the tree is also full and balanced, then its height is $\lceil \log_m l \rceil$.

Huffman coding: a procedure for constructing an optimal binary code for a set of symbols, given the frequencies of these symbols

depth-first search, or backtracking: a procedure for constructing a spanning tree by adding edges that form a path until this is not possible, and then moving back up the path until a vertex is found where a new path can be formed

breadth-first search: a procedure for constructing a spanning tree that successively adds all edges incident to the last set of edges added, unless a simple circuit is formed

Prim's algorithm: a procedure for producing a minimum spanning tree in a weighted graph that successively adds edges with minimal weight among all edges incident to a vertex already in the tree such that no edge produces a simple circuit when it is added

Kruskal's algorithm: a procedure for producing a minimum spanning tree in a weighted graph that successively adds edges of least weight that are not already in the tree such that no edge produces a simple circuit when it is added

Review Questions

1. a) Define a tree. b) Define a forest.
2. Can there be two different simple paths between the vertices of a tree?
3. Give at least three examples of how trees are used in modeling.
4. a) Define a rooted tree and the root of such a tree.
b) Define the parent of a vertex and a child of a vertex in a rooted tree.
c) What are an internal vertex, a leaf, and a subtree in a rooted tree?
d) Draw a rooted tree with at least 10 vertices, where the degree of each vertex does not exceed 3. Identify the root, the parent of each vertex, the children of each vertex, the internal vertices, and the leaves.
5. a) How many edges does a tree with n vertices have?
b) What do you need to know to determine the number of edges in a forest with n vertices?
6. a) Define a full m -ary tree.
b) How many vertices does a full m -ary tree have if it has i internal vertices? How many leaves does the tree have?
7. a) What is the height of a rooted tree?
b) What is a balanced tree?
c) How many leaves can an m -ary tree of height h have?
8. a) What is a binary search tree?
b) Describe an algorithm for constructing a binary search tree.
9. a) What is a prefix code?
b) How can a prefix code be represented by a binary tree?
10. a) Define preorder, inorder, and postorder tree traversal.
b) Give an example of preorder, postorder, and inorder traversal of a binary tree of your choice with at least 12 vertices.
11. a) Explain how to use preorder, inorder, and postorder traversals to find the prefix, infix, and postfix forms of an arithmetic expression.
b) Draw the ordered rooted tree that represents $((x - 3) + ((x/4) + (x - y) \uparrow 3))$.
c) Find the prefix and postfix forms of the expression in part (b).
12. Show that the number of comparisons used by a sorting algorithm is at least $\lceil \log n! \rceil$.
13. a) Describe the Huffman coding algorithm for constructing an optimal code for a set of symbols, given the frequency of these symbols.
b) Use Huffman coding to find an optimal code for these symbols and frequencies: A: 0.2, B: 0.1, C: 0.3, D: 0.4.
14. Draw the game tree for nim if the starting position consists of two piles with one and four stones, respectively. Who wins the game if both players follow an optimal strategy?
15. a) What is a spanning tree of a simple graph?

- b) Which simple graphs have spanning trees?
 c) Describe at least two different applications that require that a spanning tree of a simple graph be found.
16. a) Describe two different algorithms for finding a spanning tree in a simple graph.
 b) Illustrate how the two algorithms you described in (a) can be used to find the spanning tree of a simple graph, using a graph of your choice with at least eight vertices and 15 edges.
17. a) Explain how backtracking can be used to determine whether a simple graph can be colored using n colors.
 b) Show, with an example, how backtracking can be used to show that a graph with a chromatic number equal to 4 cannot be colored with three colors, but can be colored with four colors.
18. a) What is a minimum spanning tree of a connected weighted graph?
 b) Describe at least two different applications that require that a minimum spanning tree of a connected weighted graph be found.
19. a) Describe Kruskal's algorithm and Prim's algorithm for finding minimum spanning trees.
 b) Illustrate how Kruskal's algorithm and Prim's algorithm are used to find a minimum spanning tree, using a weighted graph with at least eight vertices and 15 edges.

Supplementary Exercises

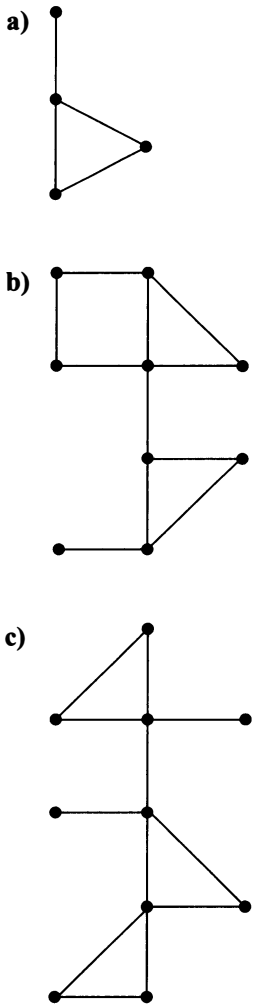
- *1. Show that a simple graph is a tree if and only if it contains no simple circuits and the addition of an edge connecting two nonadjacent vertices produces a new graph that has exactly one simple circuit (where circuits that contain the same edges are not considered different).
- *2. How many nonisomorphic rooted trees are there with six vertices?
3. Show that every tree with at least one edge must have at least two pendant vertices.
4. Show that a tree with n vertices that has $n - 1$ pendant vertices must be isomorphic to $K_{1,n-1}$.
5. What is the sum of the degrees of the vertices of a tree with n vertices?
- *6. Suppose that d_1, d_2, \dots, d_n are n positive integers with sum $2n - 2$. Show that there is a tree that has n vertices such that the degrees of these vertices are d_1, d_2, \dots, d_n .
7. Show that every tree is a planar graph.
8. Show that every tree is bipartite.
9. Show that every forest can be colored using two colors.
- A **B-tree of degree k** is a rooted tree such that all its leaves are at the same level, its root has at least two and at most k children unless it is a leaf, and every internal vertex other than the root has at least $\lceil k/2 \rceil$, but no more than k , children. Computer files can be accessed efficiently when B-trees are used to represent them.
10. Draw three different B-trees of degree 3 with height 4.
11. Give an upper bound and a lower bound for the number of leaves in a B-tree of degree k with height h .
12. Give an upper bound and a lower bound for the height of a B-tree of degree k with n leaves.
- The **binomial trees** B_i , $i = 0, 1, 2, \dots$, are ordered rooted trees defined recursively:
- Basis step:* The binomial tree B_0 is the tree with a single vertex.
- Recursive step:* Let k be a nonnegative integer. To construct the binomial tree B_{k+1} , add a copy of B_k to a second copy of B_k by adding an edge that makes the root of the first copy of B_k the leftmost child of the second copy of B_k .
13. Draw B_k for $k = 0, 1, 2, 3, 4$.
14. How many vertices does B_k have? Prove that your answer is correct.
15. Find the height of B_k . Prove that your answer is correct.
16. How many vertices are there in B_k at depth j , where $0 \leq j \leq k$? Justify your answer.
17. What is the degree of the root of B_k ? Prove that your answer is correct.
18. Show that the vertex of largest degree in B_k is the root. A rooted tree T is called an **S_k -tree** if it satisfies this recursive definition. It is an S_0 -tree if it has one vertex. For $k > 0$, T is an S_k -tree if it can be built from two S_{k-1} -trees by making the root of one the root of the S_k -tree and making the root of the other the child of the root of the first S_k -tree.
19. Draw an S_k -tree for $k = 0, 1, 2, 3, 4$.
20. Show that an S_k -tree has 2^k vertices and a unique vertex at level k . This vertex at level k is called the **handle**.
- *21. Suppose that T is an S_k -tree with handle v . Show that T can be obtained from disjoint trees T_0, T_1, \dots, T_{k-1} , where v is not in any of these trees, where T_i is an S_i -tree for $i = 0, 1, \dots, k - 1$, by connecting v to r_0 and r_i to r_{i+1} for $i = 0, 1, \dots, k - 2$.
- The listing of the vertices of an ordered rooted tree in **level order** begins with the root, followed by the vertices at level 1 from left to right, followed by the vertices at level 2 from left to right, and so on.
22. List the vertices of the ordered rooted trees in Figures 3 and 9 of Section 10.3 in level order.
23. Devise an algorithm for listing the vertices of an ordered rooted tree in level order.
- *24. Devise an algorithm for determining if a set of universal addresses can be the addresses of the leaves of a rooted tree.
25. Devise an algorithm for constructing a rooted tree from the universal addresses of its leaves.

A **cut set** of a graph is a set of edges such that the removal of these edges produces a subgraph with more connected components than in the original graph, but no proper subset of this set of edges has this property.

26. Show that a cut set of a graph must have at least one edge in common with any spanning tree of this graph.

A **cactus** is a connected graph in which no edge is in more than one simple circuit not passing through any vertex other than its initial vertex more than once or its initial vertex other than at its terminal vertex (where two circuits that contain the same edges are not considered different).

27. Which of these graphs are cacti?



28. Is a tree necessarily a cactus?

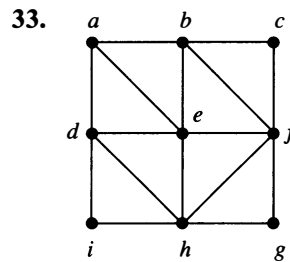
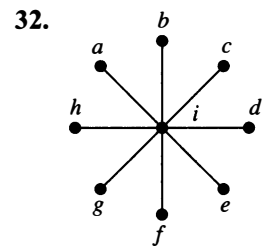
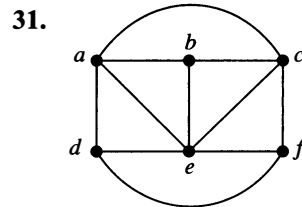
29. Show that a cactus is formed if we add a circuit containing new edges beginning and ending at a vertex of a tree.

*30. Show that if every circuit not passing through any vertex other than its initial vertex more than once in a connected graph contains an odd number of edges, then this graph must be a cactus.

A **degree-constrained spanning tree** of a simple graph G is a spanning tree with the property that the degree of a vertex

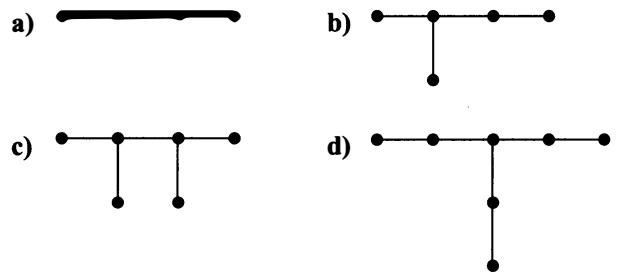
in this tree cannot exceed some specified bound. Degree-constrained spanning trees are useful in models of transportation systems where the number of roads at an intersection is limited, models of communications networks where the number of links entering a node is limited, and so on.

In Exercises 31–33 find a degree-constrained spanning tree of the given graph where each vertex has degree less than or equal to 3, or show that such a spanning tree does not exist.



34. Show that a degree-constrained spanning tree of a simple graph in which each vertex has degree not exceeding 2 consists of a single Hamilton path in the graph.

35. A tree with n vertices is called **graceful** if its vertices can be labeled with the integers $1, 2, \dots, n$ such that the absolute values of the difference of the labels of adjacent vertices are all different. Show that these trees are graceful.



A **caterpillar** is a tree that contains a simple path such that every vertex not contained in this path is adjacent to a vertex in the path.

36. Which of the graphs in Exercise 35 are caterpillars?

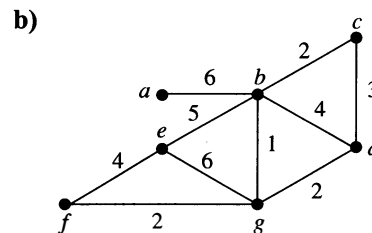
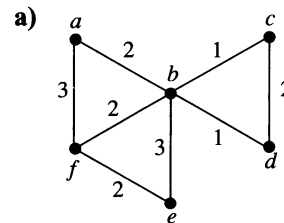
37. How many nonisomorphic caterpillars are there with six vertices?

**38. a) Prove or disprove that all trees whose edges form a single path are graceful.
b) Prove or disprove that all caterpillars are graceful.

9. Suppose that in a long bit string the frequency of occurrence of a 0 bit is 0.9 and the frequency of a 1 bit is 0.1 and bits occur independently.
- Construct a Huffman code for the four blocks of two bits, 00, 01, 10, and 11. What is the average number of bits required to encode a bit string using this code?
 - Construct a Huffman code for the eight blocks of three bits. What is the average number of bits required to encode a bit string using this code?
10. Suppose that G is a directed graph with no circuits. Describe how depth-first search can be used to carry out a topological sort of the vertices of G .
1. Suppose that e is an edge in a weighted graph that is incident to a vertex v such that the weight of e does not exceed the weight of any other edge incident to v . Show that there exists a minimum spanning tree containing this edge.
2. Three couples arrive at the bank of a river. Each of the wives is jealous and does not trust her husband when he is with one of the other wives (and perhaps with other people), but not with her. How can six people cross to the other side of the river using a boat that can hold no more than two people so that no husband is alone

with a woman other than his wife? Use a graph theory model.

- *43. Show that if no two edges in a weighted graph have the same weight, then the edge with least weight incident to a vertex v is included in every minimum spanning tree.
44. Find a minimum spanning tree of each of these graphs where the degree of each vertex in the spanning tree does not exceed 2.



Computer Projects

Write programs with these input and output.

- Given the adjacency matrix of an undirected simple graph, determine whether the graph is a tree.
- Given the adjacency matrix of a rooted tree and a vertex in the tree, find the parent, children, ancestors, descendants, and level of this vertex.
- Given the list of edges of a rooted tree and a vertex in the tree, find the parent, children, ancestors, descendants, and level of this vertex.
- Given a list of items, construct a binary search tree containing these items.
- Given a binary search tree and an item, locate or add this item to the binary search tree.
- Given the ordered list of edges of an ordered rooted tree, find the universal addresses of its vertices.
- Given the ordered list of edges of an ordered rooted tree, list its vertices in preorder, inorder, and postorder.
- Given an arithmetic expression in prefix form, find its value.
- Given an arithmetic expression in postfix form, find its value.
- Given the frequency of symbols, use Huffman coding to find an optimal code for these symbols.
- Given an initial position in the game of nim, determine an optimal strategy for the first player.
- Given the adjacency matrix of a connected undirected simple graph, find a spanning tree for this graph using depth-first search.
- Given the adjacency matrix of a connected undirected simple graph, find a spanning tree for this graph using breadth-first search.
- Given a set of positive integers and a positive integer N , use backtracking to find a subset of these integers that have N as their sum.
- *15. Given the adjacency matrix of an undirected simple graph, use backtracking to color the graph with three colors, if this is possible.
- *16. Given a positive integer n , solve the n -queens problem using backtracking.
- Given the list of edges and their weights of a weighted undirected connected graph, use Prim's algorithm to find a minimum spanning tree of this graph.
- Given the list of edges and their weights of a weighted undirected connected graph, use Kruskal's algorithm to find a minimum spanning tree of this graph.

Computations and Explorations

Use a computational program or programs you have written to do these exercises.

1. Display all trees with six vertices.
2. Display a full set of nonisomorphic trees with seven vertices.
- *3. Construct a Huffman code for the symbols with ASCII codes given the frequency of their occurrence in representative input.
4. Compute the number of different spanning trees of K_n for $n = 1, 2, 3, 4, 5, 6$. Conjecture a formula for the number of such spanning trees whenever n is a positive integer.
5. Compare the number of comparisons needed to sort lists of n elements for $n = 100, 1000, \text{ and } 10,000$, where the elements are randomly selected positive integers, using the selection sort, the insertion sort, the merge sort, and the quick sort.
6. Compute the number of different ways n queens can be arranged on an $n \times n$ chessboard so that no two queens can attack each other for all positive integers n not exceeding 10.
- *7. Find a minimum spanning tree of the graph that connects the capital cities of the 50 states in the United States to each other where the weight of each edge is the distance between the cities.
8. Draw the complete game tree for a game of checkers on a 4×4 board.

Writing Projects

Respond to these with essays using outside sources.

1. Explain how Cayley used trees to enumerate the number of certain types of hydrocarbons.
2. Define *AVL-trees* (sometimes also known as *height-balanced trees*). Describe how and why AVL-trees are used in a variety of different algorithms.
3. Define *quad trees* and explain how images can be represented using them. Describe how images can be rotated, scaled, and translated by manipulating the corresponding quad tree.
4. Define a *heap* and explain how trees can be turned into heaps. Why are heaps useful in sorting?
5. Describe dynamic algorithms for data compression based on letter frequencies as they change as characters are successively read, such as adaptive Huffman coding.
6. Explain how *alpha-beta pruning* can be used to simplify the computation of the value of a game tree.
7. Describe the techniques used by chess-playing programs such as Deep Blue.
8. Define the type of graph known as a *mesh of trees*. Explain how this graph is used in applications to very large system integration and parallel computing.
9. Discuss the algorithms used in IP multicasting to avoid loops between routers.
10. Describe an algorithm based on depth-first search for finding the articulation points of a graph.
11. Describe an algorithm based on depth-first search to find the strongly connected components of a directed graph.
12. Describe the search techniques used by the crawlers and spiders in different search engines on the Web.
13. Describe an algorithm for finding the minimum spanning tree of a graph such that the maximum degree of any vertex in the spanning tree does not exceed a fixed constant k .
14. Compare and contrast some of the most important sorting algorithms in terms of their complexity and when they are used.
15. Discuss the history and origins of algorithms for constructing minimum spanning trees.
16. Describe algorithms for producing random trees.

Index

- Absorption laws
 - for Boolean algebra, 753, 754
 - for lattices, 585
 - for propositions, 24
 - for sets, 124
- Abstract definition of Boolean algebra, 755
- Academic papers, 593
- Academy, Plato's, 2
- Acceptance, deferred, 179
- Accepted language, 806
- Accepted strings, 806
- Ackermann, Wilhelm, 310
- Ackermann's function, 310
- Acquaintanceship graph, 592–593
 - paths in, 623
- Actors, 593
- Ada, Augusta (Countess of Lovelace), 25, 27
- Adapting existing proofs, 96
- Adders, 764–765
 - full, 764, 781
 - half, 764, 781
- Addition
 - of functions, 135–136
 - of geometric progressions, 270–271
 - of integers, 222–224
 - of matrices, 247–248
 - of multisets, 132
 - of subsets, 732
- Addition rule of inference, 66
- Additive Compatibility Law, A–2
- Address system, universal, 711
- Adjacency list, 611
- Adjacency matrices, 612–614, 676
 - counting paths between vertices by, 628–629
- Adjacent cells, 768
- Adjacent vertices, 598, 600, 675
- Adjective, 786
- Adleman, Leonard, 241, 242
- Adverb, 786
- Affine transformation, 208
- Airline system, weighted graphs modeling, 648
- Alcuin of York, 632
- Aleph null, 158
- Alexander the Great, 2
- Algebra, Boolean, 5, 28
 - abstract definition of, 755
 - definition of, 781
 - identities of, 752–754, 755
- ALGOL, 792
- ALGOL 60, 792
- Algorithm(s), 167–177
 - approximation, 655
 - for base b expansion, 219, 221
 - for binary search tree, 695–707
 - for Boolean product of zero-one matrices, 252–254
 - bubble sort, 173
 - change-making, 174–176
 - for coloring graphs, 671, 674
 - complexity of, 193–199
 - average-case, 195, 430–433
 - of binary search algorithms, 194–195
 - of breadth-first search algorithms, 731
 - of bubble sort, 194–195
 - computational, 193
 - of Dijkstra's algorithm, 653
 - constant, 197
 - of depth-first search algorithms, 729
 - exponential, 197
 - factorial, 197
 - of finding maximum element of set, 194
 - of insertion sort, 196
 - linear, 197
 - of linear search algorithms, 194
 - logarithmic, 197
 - of merge sort, 479
 - polynomial, 197
 - of searching algorithms, 194–195, 197
 - of sorting algorithms, 195–196
 - space, 193
 - time, 193–196
 - worst-case, 194, 195–196
 - computer time used by, 198–199
 - for computing **div** and **mod**, 225–226
 - deferred acceptance, 179, 293
 - definition of, 168
 - Dijkstra's, 649–653, 676
 - divide-and-conquer, 474, 514
 - division, 202–203, 257
 - estimating number of operations of, 180–182
 - Euclidean, 227–229, 258, 298
 - extended, 232, 246
 - for Euler circuits, 636–637
 - for Euler paths, 637–638
 - for evaluating polynomials, 199
 - for fast integer multiplication, 475–476
 - for fast matrix multiplication, 475
 - for Fibonacci numbers, 316–317
 - for finding maximum element in sequence, 169
 - for finding minimum element in sequence, 259
 - Fleury's, 637, 646
 - Floyd's, 656–657
 - for generating combinations, 384–385
 - for generating permutations, 382–384
 - greedy, 174–176, 275–276, 703, 738, 744
 - history of word, 168
 - Huffman coding, 702–703
 - inorder traversal, 718
 - insertion sort, 174
 - for integer addition, 222–224
 - for integer multiplication, 224–226
 - for integer operations, 222–226
 - integers and, 219–229
 - iterative, for Fibonacci numbers, 317
 - Kruskal's, 739–740, 744
 - for matrix multiplication, 249–250
 - merge sort, 317–321
 - for minimum spanning trees, 738–741
 - for modular exponentiation, 226
 - Monte Carlo, 411–413
 - NAUTY, 618
 - optimal, 200
 - parallel, 606
 - postorder traversal, 718
 - preorder traversal, 716
 - Prim's, 738–739, 744
 - probabilistic, 393, 411–413, 442
 - properties of, 169
 - recursive, 311–321, 329
 - binary search, 314
 - for computing a^n , 312
 - for computing greatest common divisor, 313
 - correctness of, 315–317
 - for factorials, 197
 - for Fibonacci numbers, 316
 - linear search, 314
 - modular exponentiation, 313
 - searching, 170, 257
 - binary, 171, 257, 474
 - Boolean, 13
 - breadth-first, 729–731, 744
 - complexity of, 194–195, 197
 - depth-first, 726–729, 744
 - applications with, 727–729
 - in directed graphs, 732–734
 - linear, 170, 256, 257
 - average-case complexity of, 431–432
 - complexity of, 194, 197
 - recursive sequential, 314
 - ternary, 178
 - Web page, 13
 - serial, 606
 - shortest-path, 649–653
 - Sollin's, 743
 - sorting, 172–174
 - complexity of, 195–196
 - topological, 576–578, 582
 - for spanning trees, 738–741
 - topological sorting, 577
 - for transitive closure, 550
 - traversal, 712–719
 - Warshall's, 550–553, 582
 - Alice in Wonderland* (Carroll), 44
 - Alpha-beta pruning, 707, 748
 - Alphabet, 787, 838
 - Alphanumeric character, 340
 - Ambiguous grammar, 840
 - Analytic Engine, 27
 - Ancestors of vertex, 686, 743
 - AND, 13, 15–16
 - AND gate, 761, 781
 - Antecedent, 6
 - Antichain, 585
 - Antisymmetric relation, 523–524, 581
 - representing
 - using digraphs, 542
 - using matrices, 538–539
 - Appel, Kenneth, 668, 669
 - Application(s)
 - of backtracking, 731–732
 - of Bayes' Theorem, 419–421
 - of congruences, 205–207
 - of graph theory, 592–595, 604–607, 623–625, 627, 638, 643, 671–672
 - of graphs colorings, 671–672
 - of inclusion-exclusion, 505–512
 - of number theory, 231–244
 - of pigeonhole principle, 351–353
 - of trees, 695–707
 - Approximation algorithm, 655
 - Archimedean property, A–5
 - Archimedes, A–4
 - Archimedes' screw, A–4
 - Argument, 44, 63
 - Cantor diagonalization, 160
 - form, 64
 - valid, 64
 - Aristotle, 2
 - Arithmetic
 - computer, with large integers, 237–238
 - Fundamental Theorem of, 211, 257, 285–286
 - matrix, 247–248
 - modular, 203–205
 - Arithmetic mean, 95, 282
 - Arithmetic progression, 151
 - Arithmeticon Libri Duo* (Maurolico), 265

- Array
 - linear, 606, 607
 - two-dimensional, 606
- Arrow, Peirce, 29
- Ars Conjectandi* (Bernoulli), 407
- Art Gallery
 - problem, 675
 - Theorem, 675
- Art of Computer Programming, The* (Knuth), 172
- Article, 786
- Articulation points, 625
- Artificial intelligence, 332
 - fuzzy logic in, 19
 - fuzzy sets in, 132–133
- Assertion
 - final, 323, 329
 - initial, 323, 329
- Assignment of jobs, 513
- Assignment, stable, 179, 293
 - optimal, 293
- Assignment statements, A–10 – A–11
- Assignments, frequency, 672
- Associated homogenous recurrence relations, 467
- Associative laws
 - for addition, A–1
 - for Boolean algebra, 753, 755
 - for lattices, 585
 - for multiplication, A–1
 - for propositions, 24
 - for sets, 124
- Asymmetric relation, 528
- Asymptotic functions, 146, 192
- Asynchronous transfer mode (ATM), 144
- ATM cells, 144
- AT&T network, 625
- Automated theorem proving, 108
- Automaton
 - finite-state, 805–811
 - deterministic, 807–811, 839
 - nondeterministic, 811–814, 820–821, 822, 839
 - regular sets recognized by, 819–821
 - set not recognized by, 824–825
 - linear bounded, 825
 - pushdown, 825
 - quotient, 817
- Average-case complexity of algorithms, 195, 430–433
- Avian influenza, 424
- AVL-trees, 748
- Axiom(s), 75, A–1 – A–5
 - Completeness, A–2
 - field, A–1 – A–2
 - of Mathematical Induction, A–5
 - order, A–2
 - in proving basic facts, A–2 – A–5
 - for real numbers, A–1 – A–2
 - for set of positive integers, A–5
- Babbage, Charles, 27
- Bachmann, Paul Gustav Heinrich, 182
- Back edges, 728
- Backtracking, 726–729, 744
 - applications of, 731–732
 - in directed graphs, 733
- Backus, John, 792
- Backus-Naur form (BNF), 792–793, 839
- Backward differences, 460
- Backward reasoning, 94
- Bacon, Kevin, 623–624
- Bacon number, 623
- Bacteria, 457
- Balanced rooted trees, 691, 743
- Balanced strings of parentheses, 332
- Balanced ternary expansion, 230
- Bandwidth of graph, 680
- Bar
 - chocolate, 292
 - Ziegler's Giant, 184
- Barber paradox, 19–20
- Base b
 - Miller's test for, 245
 - strong pseudoprime to, 245
- Base b expansion, 219, 221
- Base conversion, 220
- BASIC, 340
- Basis step, 264, 284, 299, 328, 690, 706
- Basis, vertex, 632
- Basketball, 382
- Bayes, Thomas, 418, 420
- Bayes' Theorem, 417–425
 - generalized, 420
- Bayesian spam filters, 421–423
- Beaver, busy, 835
- Begging the question, 84
- Begin** statements, A–11, A–14
- Bell, E. T., 566
- Bell numbers, 566
- Bernoulli family tree, 684
- Bernoulli, James, 406, 407
- Bernoulli trial, 428, 442
 - and binomial distribution, 406–408
 - mutually independent, 378
- Bernoulli's inequality, 280
- Biconditional, 9–10
 - logical equivalences for, 25
- Bicycle racers, 424
- Bidirectional bubble sort, 173
- Big- O notation, 180–186
- Big-Omega notation, 182, 189–190
- Big-Theta notation, 182, 189–190
- Bigit, 16
- Bijective function, 138, 163
- Binary coded decimal, 231
- Binary coded decimal expansion, 774
- Binary digit
 - origin of term, 15
- Binary expansion, 219
- Binary insertion sort, 172, 174, 179
- Binary relation, 519, 581
- Binary search algorithm, 171, 257, 474
 - complexity of, 194–195, 197
 - recursive, 314
- Binary search trees, 695–698, 743
- Binary trees, 302–304, 686, 687
 - definition of, 743
 - extended, 302
 - full, 304–305
 - height of, 306
 - number of vertices of, 306
 - with prefix code, 700–701
- Binding variables, 38–39
- Binit, 16
- Binomial coefficients, 357, 363–368, 387
 - extended, 486
- Binomial distribution, 406–408
- Binomial expression, 363
- Binomial Theorem, 363–365, 387
 - extended, 486
- Binomial trees, 745
- Bipartite graphs, 602–604, 675
 - coloring of, 603
- Bird flu, 424
- Birthday problem, 409–411
- Bit, 104
 - origin of term, 15, 16
- Bit operations, 15, 104
- Bit strings, 15, 104, 129–130
 - with an odd number of 1s and ending with at least two 0s, 809
 - counting, 336
 - without consecutive 0s, 454–455
 - decoding, 700–701
 - generating next largest, 385
 - in interstellar communication, 424
 - length of, 15, 454–455
- Bitwise operators, 15–16, 104
- Blocks of statements, A–11
- BNF (Backus-Naur form), 792–793, 839
- Boat, 632
- Bonferroni's inequality, 415
- Book number of graph, 682
- Boole, George, 3, 5, 420, 749
- Boolean algebra, 5, 28, 752–755
 - abstract definition of, 755
 - definition of, 781
 - identities of, 752–754, 755
- Boolean expressions, 750–752, 781
 - dual of, 754, 781
- Boolean functions, 749–755, 781
 - dual of, 754, 781
 - functionally complete set of operators for, 759
 - implicant of, 770
 - minimization of, 766–779, 781
 - representing, 757–759
 - self-dual, 782
 - threshold, 783
- Boolean power (of zero-one matrices), 252–254
- Boolean product, 749–750, 751, 757–759, 781
 - of zero-one matrices, 252–254, 257
- Boolean searches, 13
- Boolean sum, 749–750, 751, 757–759, 781
- Boolean variable, 15, 104, 750, 758, 781
- Boole's inequality, 415
- Boruvka, Otakar, 740
- Bots, 734
- Bottom-up parsing, 791
- Bound
 - greatest lower, 582
 - of poset, 574
 - least upper, 582, A–2
 - of poset, 574
 - lower
 - of lattice, 586
 - of poset, 574, 582
 - upper, A–2
 - of lattice, 586
 - of poset, 574, 582
- Bound variable, 38–39
- Bounded lattices, 586
- Boxes, distributing objects into, 376–379
 - distinguishable, 376
 - indistinguishable, 376
- Breadth-first search, 729–731, 744
- Bridge, in graphs, 625
- Bridge problem, Königsberg, 633–634, 636, 638
- B-tree of degree k , 745
- Bubble sort, 173, 257
 - bidirectional, 173, 259
 - variant of, 173, 219
 - worst-case complexity of, 194–195
- Bug, computer, 811
- Buoyancy, principle of, A–4
- Busy beaver function, 835
- Butane, 688
- Bytes, 220
- C programming language, 346, 559
- Cabbage, 632
- Cactus, 746
- CAD programs, 772
- Caesar, Julius, 207
- Caesar cipher, 207–208
- Calculus
 - predicate, 31
 - propositional, 2
- Call graphs, 594
 - connected components of, 625
- Canterbury Puzzles, The* (Dudeney), 454
- Cantor diagonalization argument, 160
- Cantor digits, 386
- Cantor expansion, 231, 386
- Cantor, Georg, 111, 113, 160
- Card hands, 358–359, 376–377
- Cardinality
 - aleph null, 158
 - of set, 113, 116, 158–160

- Carmichael, Robert Daniel, 240
 Carmichael numbers, 240–241, 257
 Carroll, Lewis (Charles Dodgson), 44–45, 50
 Carry, 222
 Cartesian products, 117–118
 Cases, proofs by, 86–90, 105
 Cash box, 372–373
 Catalan numbers, 456
 Caterpillar, 746
 Cayley, Arthur, 683, 688
 Ceiling function, 143–145, 163
 Celebrity problem, 282
 Cells, adjacent, 768
 Center, 695
 Chain, 585
 Chain letter, 691
 Change-making algorithm, 174–176
 Characteristic equation, 461
 Characteristic function, 148
 Characteristic roots, 461
 Chebyshev, Pafnuty Lvovich, 438
 Chebyshev's inequality, 438–439, 442
 Checkerboard, tiling of, 97–102, 277, 283
 Cheese, in Reve's puzzle, 454
 Chen, J. R., 215
 Chess, game tree for, 707
 Chessboard
 knight's tour on, 647
 n -queens problem on, 731–732
 squares controlled by queen on, 679
 Chi (Greek letter), 667
 Chickens, 680
 Child of vertex, 686, 743
 Chinese meal, preparing, 585
 Chinese postman problem, 638, 682
 Chinese Remainder Theorem, 235–237, 258
 Chocolate bar, 292
 Chomp, 91, 292, 586
 Chomsky, Avram Noam, 785, 789, 790
 Chomsky classification of grammars, 789–790
 Chromatic number, 668, 676
 edge, 674
 of graph, 667
 k -critical, 674
 Chung-Wu Ho, 290
 Church, Alonzo, 833
 Church-Turing thesis, 833
 Cipher
 affine, 208
 Caesar, 207–208
 RSA, 241–244
 shift, 208
 Circuits
 combinational, 761–763
 depth of, 766
 examples of, 763–764
 minimization of, 766–779
 in directed graph, 546, 582
 Euler, 633–638, 676
 in graph, 622
 Hamilton, 638–643, 676
 multiple output, 764
 simple, 622
 Circular reasoning, 84, 105
 Circular relation, 584
 Civil War, 32
 Class
 congruence, 204
 equivalence, 558–559, 582
 definition of, 558
 and partitions, 559–562
 representative of, 558
 NP, 197
 P, 197
 Class A addresses, 341
 Class B addresses, 341
 Class C addresses, 341
 Class D addresses, 341
 Class E addresses, 341
 Clauses, 68
 Clique, 678
 Closed walk, 622
 Closest-Pair Problem, 479–482
 Closure, Kleene, 805, 817
 Closure laws
 for addition, A–1
 for multiplication, A–1
 Closures of relations, 544–553, 582
 reflexive, 544, 582
 symmetric, 545, 582
 transitive, 544, 547–550, 582
 computing, 550–553
 Coast Survey, U. S., 32
 COBOL, 792, 811
 Codes, Gray, 642–643
 Codes, prefix, 700–703, 743
 Codeword enumeration, 455–456
 Coding, Huffman, 701–703, 744
 Codomain of function, 134, 149, 163
 Coefficient(s)
 binomial, 357, 363–368, 387
 extended, 486
 constant
 linear homogenous recurrent relations with,
 460, 461–467, 514
 linear nonhomogenous recurrent relations
 with, 467–471, 514
 multinomial, 382
 Coins, change using fewest, 175–176
 Collaboration graphs, 593–594
 paths in, 623
 Collatz problem, 101–102
 Collision, 206
 in hashing functions, probability of, 410–411
 Coloring
 of bipartite graphs, 603
 chromatic number in, 667
 of graphs, 666–672, 676
 of maps, 666–667
 Column of matrix, 247
 Combinations, 357–360
 of events, 396–398, 403–404
 generating, 384–385
 linear, 232
 with repetition, 371–375
 Combinational circuits, 761–763
 depth of, 766
 examples of, 763–764
 minimization of, 766–779
 Combinatorial identities, 364–368
 Combinatorial proof, 359, 386
 Combinatorics, 335, 352, 378, 386
 Comments in pseudocode, A–11 – A–12
 Common difference, 151
 Common divisor, greatest, 215–217
 Common errors
 in exhaustive proofs, 90
 in proof by cases, 90
 Common multiple, least, 216, 217
 Common ratio, 150
 Commutative laws
 for addition, A–1
 for Boolean algebra, 753, 755
 for lattices, 585
 for multiplication, A–1
 for propositions, 26
 for sets, 124
 Comparable elements in poset, 567, 582
 Compatible total ordering, 576, 582
 Compatibility laws, A–2
 Compilers, 559, 810
 Complement, 749, 781
 of Boolean function, 751
 double, law of, in Boolean algebra, 753,
 755, 757
 of fuzzy set, 132
 one's, 230
 of set, 123, 163
 two's, 230–231
 Complement laws, for sets, 124
 Complementary event, 396
 Complementary graph, 611
 Complementary relation, 528
 Complementation law, for sets, 124
 Complemented lattice, 586, 755
 Complete bipartite graph, 604, 675
 Complete graphs, 601, 675
 Complete induction, 284
 Complete m -ary tree, 694
 Complete m -partite graph, 678
 Complexity of algorithms, 193–199
 average-case, 109, 430–433
 of binary search algorithms, 194–195
 of breadth-first search algorithms, 731
 of bubble sort, 194–195
 computational, 193
 of Dijkstra's algorithm, 653
 constant, 197
 of depth-first search algorithms, 729
 exponential, 197
 factorial, 197
 of finding maximum element of set, 194
 of insertion sort, 196
 linear, 197
 of linear search algorithms, 194
 logarithmic, 197
 of merge sort, 479
 polynomial, 197
 of searching algorithms, 194–195, 197
 of sorting algorithms, 195–196
 space, 193
 time, 193–196
 worst-case, 194, 195–196
 Complexity of merge sort, 479
 Components of graphs, connected, 625–626
 strongly, 627, 676
 Composite integers, 189–190, 210, 211,
 217, 257
 Composite key, 532, 582
 Composite of relations, 526–527, 581
 Composition rule, 324
 Compositions of functions, 139–141, 163
 Compound interest, 451
 Compound propositions, 3, 10–11, 21, 22, 104
 consistent, 104
 disjunctive normal form of, 29
 dual of, 29
 equivalences of, in Boolean algebra, 750
 satisfiable, 30
 well-formed formula for, 301
 Computable function, 163, 835
 Computation, models of, 785–837
 Computational complexity of algorithms, 193
 of Dijkstra's algorithm, 653
 Turing machine in making precise, 833
 Computational geometry, 288–290, 479–481
 Computer arithmetic, with large integers,
 237–238
 Computer debugging, 811
 Computer file systems, 689
 Computer network, 590
 with diagnostic lines, 591
 interconnection networks, 606–607
 local area networks, 605–606
 multicasting over, 726
 with multiple lines, 590
 with multiple one-way lines, 592
 with one-way lines, 591
 Computer programming, 792
 Computer programming languages, 68, 135
 Computer representation, of sets, 129–130
 Computer science, 792
 Computer time used by algorithms, 198–199
 Computer virus transmission, 389
 Computer-aided design (CAD) programs, 772
 Concatenation, 300, 804, 839

- Conclusion, 6, 44, 64
 - fallacy of affirming, 69
- Concurrent processing, 595
- Conditional constructions, A-12 – A-13
- Conditional probability, 400, 404–405, 442
- Conditional statement, 6
 - for program correctness, 324–326
- Conditions
 - don't care*, 774–775
 - initial, 450, 513
- Congruence class, 204
- Congruence modulo m , 203–204, 556, 558, 582
 - inverse of, 234
- Congruences, 205–207
 - linear, 234–235, 257
- Conjecture, 75, 105
 - $3x + 1$, 101–102
 - and counterexamples, 96–97
 - Frame's, 460
 - Goldbach's, 214–215
 - and proof, 96–97
 - twin prime, 215
- Conjunction, 4–5, 104, 781
 - distributive law of disjunction over, 23
- Conjunction rule of inference, 66
- Conjunctive normal form, 758
- Connected components of graphs, 625–626
 - strongly, 627, 676
- Connected graphs, 621–629
 - directed, 626–627
 - planar simple, 659–663
 - strongly, 626
 - undirected, 624–626
 - weakly, 626
- Connecting vertices, 598
- Connectives, 4
- Connectivity relation, 547, 582
- Consecutive composite integers, 217
- Consequence, 6
- Consistency, of system specifications, 12
- Consistent compound propositions, 104
- Constant coefficients
 - linear homogenous recurrent relations with, 460, 461–467, 513
 - linear nonhomogenous recurrent relations with, 467–471, 514
- Constant complexity of algorithms, 197
- Construction of the real numbers, A-5
- Constructions
 - conditional, A-12 – A-13
 - loop, A-13 – A-14
- Constructive existence proof, 91, 105
- Contain, 112
- Context-free grammars, 789, 839
- Context-free language, 789
- Context-sensitive grammars, 789
- Contingency, 21, 104
- Contradiction, 21, 80
 - proofs by, 80–83
- Contrapositive, of implication, 8
- Converse
 - of directed graph, 611
 - of implication, 8, 104
- Convex polygon, 288
- Cookie, 373
- Corollary, 75, 105
- Correctness
 - of programs, 322–328, 329
 - conditional statements for, 324–326
 - loop invariants for, 326–327
 - partial, 323
 - program verification for, 323
 - rules of inference for, 323–324
 - of recursive algorithms, 315–316
- Correspondences, one-to-one, 136, 160
- Countability, of rational numbers, 158, 163, 218
- Countable set, 158
- Counterexamples, 34, 83
 - conjecture and, 96–97
- Counting, 335–391, 449–518
 - basic principles of, 335–340
 - bit strings, 336
 - without consecutive 0s, 454–455
 - Boolean functions, 751
 - combinations, 357–360
 - derangements, 510, 514
 - by distributing objects into boxes, 376–379
 - functions, 336–337
 - generating functions for, 488–493
 - Internet addresses, 341
 - one-to-one functions, 337
 - onto functions, 509–510, 514
 - passwords, 340
 - paths between vertices, 628–629
 - permutations, 355–357
 - pigeonhole principle and, 347–353
 - reflexive relations, 524
 - relations, 522
 - with repetition, 370–371
 - subsets of finite set, 338
 - telephone numbers, 337
 - tree diagrams for, 343–344
 - variable names, 340
- Contrapositive, of implication, 104
- Course in Pure Mathematics, A* (Hardy), 93
- Covariance, 442
- Covering relation, 579
- Covers, 579
- CPM (Critical Path Method), 587
- Crawlers, 734
- Cricket, 93
- Critical Path Method (CPM), 587
- Crossing number, 666
- Cryptography, 207–208
 - private key, 241
 - public key, 231, 241–244
- Cunningham numbers, 214
- Cut edge, 625
- Cut set of graph, 746
- Cut vertices, 625
- Cycle
 - in directed graph, 546, 582
 - with n vertices, 601
- Cylinder, 769
- Czekanowski, Jan, 740
- Data compression, 701
- Database
 - composite key of, 532
 - intension of, 531
 - primary key of, 531
 - records in, 531
 - relational model of, 531–532, 581
- Database query language, 535–536
- Datagrams, 346–347
- Datalogy, 792
- Datatype, 113
- de Bruijn sequences, 682
- de Méré, Chevalier, 400
- De Morgan, Augustus, 22, 25, 668, 669
- De Morgan's laws
 - for Boolean algebra, 753, 757
 - for propositions, 22, 25–26
 - proving by mathematical induction, 274–275
 - for quantifiers, 40–41
 - for sets, 124–126
- Debugging, 811
- Decimal, binary coded, 231
- Decimal notation, 168, 219
- Decision problems, 834
- Decision trees, 698–700, 743
- Decreasing function, 137
- Decryption, 208, 231, 257
 - RSA, 243
- Deductive reasoning, 264
- Deep-Blue, 707, 748
- Deferred acceptance algorithm, 179, 293
- Definition
 - domain of, 149
 - recursive, 263, 294–308
- Degree
 - of linear homogenous recurrence relations, 461
 - of membership, 132
 - of n -ary relations, 530
 - of region, 661
 - of vertex in undirected graph, 598
- Degree-constrained spanning tree, 746
- Delay machine, 799–800
- Dense
 - graph, 614
 - poset, 581
- Dependency notation, 784
- Depth of combinatorial circuit, 766
- Depth-first search, 726–729, 744
 - applications with, 732–734
 - in directed graphs, 732–734
- Derangements, 510–512, 514
 - number of, 510, 514
- Derivable form, 787
- Derivation, 787
- Derivation trees, 790–792, 839
- Descartes, René, 117
- Descendants of vertex, 686, 743
- Detachment, law of, 65
- Deterministic finite-state automata, 807–811, 839
- Deviation, standard, 436
- Devil's pair, 621
- Diagnostic test results, 419–420
- Diagonal matrix, 255
- Diagonal of a polygon, 288
- Diagonal relation, 545
- Diagonalization argument, 160
- Diagrams
 - Hasse, 571–572, 574, 575, 582
 - state, for finite-state machine, 798
 - tree, 343–344, 386
 - Venn, 114, 115, 122, 123, 163
- Diameter of graph, 680
- Dice, 394
- Dictionary ordering, 382
- Die, 416, 436–437
 - dodecahedral, 443
 - octahedral, 443
- Difference, A-6
 - backward, 460
 - common, 151
 - forward, 516
 - of multisets, 132
 - of sets, 123, 163
 - symmetric, 131, 163
- Difference equation, 460
- Digits
 - binary
 - origin of term, 15, 16
 - Cantor, 386
- Digraphs, 582, 591, 675
 - circuit (cycle) in, 546, 582
 - connectedness in, 626–627
 - converse of, 611
 - depth-first search in, 732–734
 - edges of, 591, 600–601
 - Euler circuit of, 634
 - loops in, 541, 582
 - paths in, 546–547, 582, 676
 - representing relations using, 541–542
 - self-converse, 679
 - vertex of, 590, 600
- Dijkstra, Edsger Wybe, 649
- Dijkstra's algorithm, 649–653, 676
- Dimes, 175–176
- Diophantus, 101, 239
- Dirac, G. A., 641
- Dirac's Theorem, 641
- Direct proof, 76–77, 105
- Directed edges, 592, 600–601, 675

- Directed graphs, 582, 591, 675
 - circuit (cycle) in, 546, 582
 - connectedness in, 626–627
 - converse of, 611
 - depth-first search in, 732–734
 - edges of, 591, 600–601
 - Euler circuit of, 634
 - loops in, 541, 582
 - paths in, 546–547, 582, 676
 - representing relations using, 541–542
 - self-converse, 679
 - vertex of, 590, 600
- Directed multigraph, 591, 675
 - paths in, 623
- Directly derivable form, 787
- Dirichlet drawer principle, 348
- Dirichlet, G. Lejeune, 348
- Discours* (Descartes), 117
- Discourse, universe of, 34
- Discrete mathematics, definition of, vii, viii
- Discrete probability, 393–447, 442
 - assigning, 401–403
 - of collision in hashing functions, 410–411
 - of combinations of events, 396–398, 403–404
 - conditional, 400, 404–405, 442
 - finite, 394–396
 - Laplace's definition of, 393
- Disjoint set, 122
- Disjunction, 4–5, 104, 781
 - associative law of, 24
 - distributive law of, over conjunction, 23
- Disjunctive normal form
 - for Boolean variables, 758
 - for propositions, 29
- Disjunctive syllogism, 66
- Distance
 - between distinct vertices, 680
 - between spanning trees, 737
- Distinguishable
 - boxes, 376
 - objects, 376
- Distinguishable strings, 827
- Distributing objects into boxes, 376–379
- Distribution
 - binomial, 406–408
 - probability, 401
 - of random variable, 408, 442
 - geometric, 433–434, 442
 - uniform, 402, 442
- Distributive lattice, 586, 755
- Distributive laws
 - for Boolean algebra, 752, 753, 755
 - for propositions, 23, 24
 - for sets, 124
- div**, 202, 225–226
- Divide-and-conquer
 - algorithms, 474, 514
 - recurrence relations, 474–482
- Dividend, 214
- Divides, 201
- Divine Benevolence* (Bayes), 420
- Divisibility facts, proving, by mathematical induction, 273
- Divisibility relation, 567
- Division
 - of integers, 200–208
 - trial, 214
- Division algorithm, 202–203
- Divisor, 214
 - greatest common, 215–217, 232–234, 258
- DNA sequencing, 638
- Dodecahedral die, 443
- Dodgson, Charles (Lewis Carroll), 44–45
- Dogs, 17
- Domain
 - of definition, 149
 - of discourse, 34
 - of a function, 134, 149, 163
 - of n -ary relation, 530
 - of a quantifier, 34
 - of relation, 530
 - universe of, 34
- Dominating set, 679
- Domination laws
 - for Boolean algebra, 753
 - for propositions, 27
 - for sets, 124
- Dominoes, 97, 266
- Don't care* conditions, 774–775
- Double complement, law of, in Boolean algebra, 753, 757
- Double negation law, for propositions, 26
- Double summations, 156
- Drug testing, 424
- Dual
 - of Boolean expression, 754
 - of Boolean function, 754
 - of compound proposition, 29
 - of poset, 579
- Dual graph, 667
- Duality in lattice, 587
- Duality principle, for Boolean identities, 754
- Dudeney, Henry, 454
- Ear(s), 292
 - nonoverlapping, 292
- Earth, 424
- EBNF (Extended Backus-Naur form), 796
- Eccentricity of vertex, 695
- Ecology, niche overlap graph in, 592
- Edge chromatic number, 674
- Edge coloring, 674
- Edge vertex, 541
- Edges
 - cut, 625
 - directed, 592, 600, 675
 - of directed graph, 591, 600–601
 - of directed multigraph, 591
 - endpoints of, 598
 - incident, 598, 676
 - of multigraph, 590
 - multiple, 590, 591, 675
 - of pseudograph, 591
 - of simple graph, 590, 611
 - undirected, 592, 675
 - of undirected graph, 592, 600
- Egyptian (unit) fraction, 331
- Einstein, Albert, 21
- Electronic mail, 421
- Elementary subdivision, 663, 676
- Elements, 112, 163
 - comparable, in partially ordered set, 567, 582
 - equivalent, 556
 - fixed, 442
 - greatest, of partially ordered set, 573, 582
 - incomparable, in partially ordered set, 567, 582
 - least, of partially ordered set, 573, 582
 - of matrix, 247
 - maximal, of partially ordered set, 572, 582
 - minimal, of partially ordered set, 572, 573, 582
- Elements* (Euclid), 212
- Elements of Mathematical Logic* (Lukasiewicz), 721
- Ellipses (...), 112
- Empty folder, 114
- Empty set, 114, 163
- Empty string, 151, 787
- Encryption, 207, 231, 257
 - RSA, 242–243
- Encyclopedia of Integers Sequences, The* (Sloane and Plouffe), 153
- End** statements, A–11, A–14
- Endpoints of edge, 598
- English sentences
 - translating logical expressions to, 42–43
 - translating to logical expressions, 11–12, 42–43, 56–57
- Entry of matrix, 247
- Enumeration, 335, 386
 - codeword, 455–456
- Equality, of sets, 113, 163
- Equation
 - characteristic, 461
 - difference, 460
- Equivalence classes, 558–559, 582
 - definition of, 558
 - and partitions, 559–562
 - representative of, 558
- Equivalence, proofs of, 82–83
- Equivalence relations, 555–562, 582
 - definition of, 555
- Equivalent Boolean expressions, 750, 751
- Equivalent elements, 556
- Equivalent finite-state automata, 806, 809–811
- Equivalent propositions, 8, 21–30
- Eratosthenes, 507–508
 - sieve of, 210, 514
- Erdős number, 584, 623, 624
- Erdős Number Project, 623
- Erdős, Paul, 584, 623
- Errors, in proofs, 83–84
 - by mathematical induction, 278–279
- Essential prime implicant, 770, 781
- Euclid, 212, 227, 228
- Euclidean algorithm, 227–229, 258, 298
 - extended, 232, 246
- Euler circuits, 633–638, 676
- Euler ϕ -function, 218
- Euler, Leonhard, 214, 633, 635
- Euler paths, 633–638, 676
- Euler's formula, 659–663, 676
- "Eureka," A–4
- Evaluation functions, 707
- Even integer, 77
- Event(s), 394
 - combinations of, 396–398, 403–404
 - complementary, 396
 - independent, 399, 405–406, 442
 - mutually independent, 444
- Exams, scheduling, 671, 672
- Exclusion rule, 299
- Exclusive or, 5, 104
- Exhaustive proof, 87
- Existence proofs, 91–92, 105
 - constructive, 91
 - nonconstructive, 91
- Existential generalization, 70, 71
- Existential instantiation, 70, 71
- Existential quantification, 36, 105
 - negation of, 39–41
- Existential quantifier, 36–37
- Expansion
 - balanced ternary, 230
 - base b , 219, 221
 - binary, 219, 257
 - binary coded decimal, 774
 - Cantor, 231
 - hexadecimal, 219–220, 257
 - octal, 220, 257
 - one's complement, 230
 - two's complement, 230–231
- Expected values, 426–429, 442
 - in hatcheck problem, 430
 - of inversions in permutation, 430–431
 - linearity of, 429–431, 442
- Experiment, 394
- Exponential complexity of algorithms, 197
- Exponential functions, A–7
- Exponential generating function, 498
- Exponentiation, modular, 226–227
 - recursive, 313
- Expression(s)
 - binomial, 363
 - Boolean, 750–752, 781
 - infix form of, 719

- logical
 - translating English sentences into, 11, 42–43, 56–57
 - translating to English sentences, 55
- postfix form of, 721
- prefix form of, 719
- regular, 818, 839
- Extended Backus-Naur form (EBNF), 796
- Extended binary trees, 302
- Extended binomial coefficients, 486
- Extended Binomial Theorem, 487
- Extended Euclidean algorithm, 232, 246
- Exterior of simple polygon, 288
- Factor, of integers, 201
- Factorial complexity of algorithms, 197
- Factorial function, 145
- Factorials, recursive procedure for, 311–312
- Factorization into primes, 211–212
 - for finding greatest common divisor, 216
 - for finding least common multiple, 216, 217
 - uniqueness of, 233–234
- Facts, 75
- Failure, 406
- Fallacy, 63, 69, 105
 - of affirming conclusion, 69
 - of begging the question, 84, 105
 - of circular reasoning, 84, 105
 - of denying hypothesis, 69
- False
 - negative, 419, 421
 - positive, 419, 421
- Family trees, 683, 684
- Farmer, 632
- Fast multiplication
 - of integers, 475–476
 - of matrices, 476–478
- Fermat, Pierre de, 239
- Fermat's Last Theorem, 100–101, 299
- Fermat's Little Theorem, 239, 258
- Fibonacci, 298
- Fibonacci numbers, 297
 - formula for, 463–464
 - and Huffman coding, 709
 - iterative algorithm for, 317
 - rabbits and, 451–452
 - recursive algorithms for, 316
- Fibonacci trees, rooted, 695
- Field axioms, A-1 – A-2
- Fields, 531
- Filter, spam, 421–423
- Final assertion, 323, 329
- Final exams, scheduling, 671, 672
- Final value, A-13
- Finite graph, 589
- Finite probability, 394–396
- Finite sets, 116, 158, 163, 500, 514
 - subsets of
 - counting, 338
 - number of, 273–274
 - union of three, number of elements in, 501–503, 514
 - union of two, number of elements in, 500, 514
- Finite-state automata, 805–811
 - deterministic, 807–811, 839
 - equivalent, 809–811
 - nondeterministic, 811–814, 820–821, 822, 839
 - regular sets recognized by, 819–821
 - set not recognized by, 824–825
- Finite-state machines, 785, 796–814, 839
 - with no output, 804–814
 - with outputs, 798–801
 - transition function extension in, 805
- First difference, 460
- First forward difference, 516
- Fixed elements, 442
- Fixture controlled by three switches, circuit for, 764
- Flavius Josephus, 459
- Fleury's algorithm, 637, 646
- Flip-flops, 784
- Floor function, 143–145, 163
- Floyd's algorithm, 656–657
- Folder, empty, 114
- Forests, 684, 685
 - definition of, 742
 - minimum spanning, 742
 - spanning, 736
- Form(s)
 - Backus-Naur, 792–793, 839
 - conjunctive normal, 758
 - disjunctive normal
 - for Boolean variables, 758
 - for propositions, 29
 - infix, 719
 - postfix, 720
 - prefix, 719
 - prenex normal, 62
- Formal language, 786
- Formal power series, 485
- Formula(s)
 - Euler's, 659–663, 676
 - for Fibonacci numbers, 463–464
 - Stirling's, 146
 - summation, 157, 267
 - well-formed, 301
 - for compound propositions, 301
 - of operators and operands, 301
- FORTRAN, 792
- Fortune cookie, 388
- Forward differences, 516
- Forward reasoning, 94
- Four Color Theorem, 668–671, 676
- Fraction, unit (Egyptian), 331
- Frame's conjecture, 460
- Free variable, 38, 105
- Frend
 - Sophia, 25
 - William, 27
- Frequency assignments, 672
- Frisbee, rocket-powered, 750
- Full adder, 764, 781
- Full binary trees, 304–305
 - height of, 306
 - number of vertices of, 306
- Full m -ary tree, 686, 690, 743
 - complete, 694
- Full subtractor, 766
- Function(s), 133–146, 163
 - Ackermann's, 310
 - addition of, 135–136
 - asymptotic, 192
 - bijective, 138, 163
 - Boolean, 749–755, 781
 - dual of, 754
 - functionally complete set of operators for, 759
 - implicant of, 770
 - minimization of, 766–779, 781
 - representing, 757–759
 - self-dual, 782
 - threshold, 783
 - busy beaver, 835
 - ceiling, 143–145, 163
 - characteristic, 148
 - codomain of, 134, 149, 163
 - compositions of, 139–141, 163
 - computable, 163, 835
 - counting, 336–337
 - decreasing, 137
 - definition of, 133, 163
 - domain of, 134, 149, 163
 - Euler ϕ , 218
 - evaluation, 707
 - exponential, A-7
 - factorial, 145
 - floor, 143–145, 163
 - generating, 484–495, 514
 - for counting, 488–493
 - exponential, 498
 - probability, 499
 - for proving identities, 495
 - for recurrence relations, 493–495
 - graphs of, 142
 - greatest integer, 143–145
 - growth of, 180–190
 - growth of combinations of, 186–189
 - hashing, 205–206
 - collision in, probability of, 410–411
 - identity, 138
 - increasing, 137
 - inverse, 139–140, 163
 - invertible, 140
 - iterated, 311
 - iterated logarithm, 311
 - logarithmic, A-7 – A-9
 - McCarthy 91, 331
 - multiplication of, 135–136
 - number-theoretic, 831
 - one-to-one (injective), 136, 138, 163
 - counting, 337
 - onto (surjective), 137–138, 163
 - number of, 509–510, 514
 - partial, 149
 - product of, 135–136
 - propositional, 30, 31, 105
 - existential quantification of, 36
 - universal quantification of, 33
 - range of, 134, 163
 - recursive, 133, 295–299, 328
 - as relations, 520
 - strictly decreasing, 137
 - strictly increasing, 137
 - sum of, 135–136
 - threshold, 783
 - total, 149
 - transition, 798
 - Turing machines computing, 831–832
 - undefined, 149
- Functional completeness, 759
- Functional decomposition, 784
- Functionally complete set of operators
 - for Boolean functions, 759, 781
 - for propositions, 29
- Fundamental Theorem of Arithmetic, 211, 257
- Fuzzy logic, 19
- Fuzzy sets, 132–133
 - complement of, 132
 - intersection of, 133
 - union of, 133
- Gambling, 393
- Game of
 - Chomp, 91
- Game trees, 704–707, 743
- Gates, logic, 760–765
 - AND, 761, 781
 - combination of, 761–763
 - NAND, 766
 - NOR, 766
 - OR, 761, 781
 - threshold, 783
- Gating networks, 761–763
 - depth of, 766
 - examples of, 763–764
 - minimization of, 766–779
- Gauss, Karl Friedrich, 204, 213
- GCD (greatest common divisor), 215–217, 232–234, 258
- Generalization
 - existential, 70, 71
 - universal, 70, 71
- Generalized combinations, 371–375
- Generalized induction, 307–308
- Generalized permutations, 370–371
- Generalized pigeonhole principle, 349–351, 387

- Generating functions, 484–495, 514
 - for counting, 488–493
 - exponential, 498
 - probability, 499
 - for proving identities, 495
 - for recurrence relations, 493–495
- Geometric distribution, 433–434, 442
- Geometric mean, 95, 282
- Geometric progression(s), 150, 152
 - sums of, 270–271
- Geometric series, 155
- Geometry, computational, 288–290, 479–481
- Giant strongly connected components (GSCC), 627
- GIMPS (Great Internet Mersenne Prime Search), 213
- Goat, 632
- Gödel, Escher, Bach* (Hofstadter), 333
- Goldbach, Christian, 214, 215
- Goldbach's conjecture, 214–215
- Golf, hole-in-one, 279
- Golomb, Solomon, 99
- Golomb's self-generating sequence, 333
- Google, 13, 734
- Gossip problem, by mathematical induction, 282
- Graceful trees, 746
- Graham, Ron, 584
- Grammars, 785–793
 - ambiguous, 840
 - Backus-Naur form of, 792–793, 839
 - context-free (type 2), 789, 839
 - context-sensitive, 789
 - monotonic, 790
 - noncontracting, 790
 - phrase-structure, 787–790, 838
 - productions of, 787, 838
 - regular (type 3), 789, 817, 821–824, 839
 - type 0, 789, 838
 - type 1, 789, 838–839
- Graph(s), 589–682
 - acquaintanceship, 592–593
 - bandwidth of, 680
 - bipartite, 602–604, 675
 - book number of, 682
 - call, 594
 - connected components of, 625
 - chromatic number of, 667
 - chromatically k -critical, 674
 - collaboration, 593–594
 - coloring, 666–672, 676
 - complementary, 611
 - complete, 601, 675
 - complete bipartite, 604, 675
 - complete m -partite, 678
 - connected components of, 625–626, 676
 - connectedness in, 621–629
 - cut set of, 746
 - definition of, 591
 - diameter of, 680
 - directed, 582, 591, 675
 - circuit (cycle) in, 546, 582
 - connectedness in, 626–627
 - converse of, 611
 - dense, 614
 - depth-first search in, 732–734
 - edges of, 591, 600–601
 - Euler circuit of, 634
 - loops in, 541, 582
 - paths in, 546–547, 582, 676
 - representing relations using, 541–542
 - self-converse, 679
 - simple, 591
 - vertex of, 590, 600
 - directed multigraph, 592, 675
 - dual, 667
 - finite, 589
 - of functions, 142
 - Hollywood, 593
 - homeomorphic, 663–665, 676
 - independence number of, 680
 - infinite, 589
 - influence, 593, 594
 - intersection, 596
 - isomorphic, 611, 615–618, 676
 - paths and, 627–628
 - matching in, 605
 - mixed, 591
 - models, 592–595
 - monotone decreasing property of, 681
 - monotone increasing property of, 681
 - multigraphs, 590, 675
 - niche overlap, 592, 593
 - nonplanar, 663–665
 - n -regular, 610
 - orientable, 679
 - paths in, 622–624
 - planar, 657–665, 676
 - precedence, 595
 - pseudograph, 590, 592, 675
 - radius of, 680
 - regular, 610, 676
 - representing, 611–618
 - in roadmap modeling, 595
 - self-complementary, 620
 - simple, 590, 601–602, 675
 - coloring of, 667
 - connected planar, 659–663
 - crossing number of, 666
 - dense, 614
 - edges of, 590, 608
 - isomorphic, 615–618, 675
 - orientation of, 679
 - paths in, 725
 - random, 680–681
 - self-complementary, 620
 - with spanning trees, 725–726
 - sparse, 614
 - thickness of, 666
 - vertices of, 590, 608
 - simple directed, 591
 - sparse, 614, 741
 - strongly directed connected, 626
 - subgraph of, 607, 676
 - terminology of, 597–601
 - undirected, 592, 599, 676
 - connectedness in, 624–626
 - Euler circuit of, 634
 - Euler path of, 638
 - orientation of, 679
 - paths in, 622, 676
 - underlying, 601, 675
 - union of, 661, 676
 - very large scale integration, 682
 - Web, 594–595
 - strongly connected components of, 627
 - weighted, 676
 - shortest path between, 647–653
 - traveling salesman problem with, 653–655
 - wheel, 601, 676
- Gray codes, 642–643
- Gray, Frank, 643
- Great Internet Mersenne Prime Search (GIMPS), 213
- “Greater than or equal” relation, 566–567
- Greatest common divisor (GCD), 215–217, 232–234, 258
- Greatest element of poset, 573, 582
- Greatest integer function, 143–145, 163
- Greatest lower bound, 582
 - of poset, 574
- Greedy algorithms, 174–176, 275–276, 703, 738, 744
- Greedy change-making algorithm, 175
- Growth of functions, 180–190
- GSCC (giant strongly connected components), 627
- Guarding set, 675
- Guare, John, 623
- Guthrie, Francis, 669
- Hadamard, Jacques, 213
- Haken, Wolfgang, 668
- Half adder, 764, 781
- Half subtractor, 766
- Halting problem, 176–177, 834–835
- Hamilton circuits, 636–643, 676
- Hamilton paths, 638–643, 676
- Hamilton, William Rowan, 640, 669
- Hamilton's “Voyage Around the World” Puzzle, 639
- Handle, 745
- Handshaking Theorem, 599
- Hanoi, tower of, 452–454
- Hardware systems, 12
- Hardy, Godfrey Harold, 93
- Harmonic mean, 103
- Harmonic number(s), 192
 - inequality of, 272–273
- Harmonic series, 273
- Hashing functions, 205–206
 - collision in, probability of, 410–411
- Hasse diagrams, 571–572, 574, 575, 582
- Hasse, Helmut, 571
- Hasse's algorithm, 101–102
- Hatchcheck problem, 430, 510
- Hazard-free switching circuits, 784
- Heaps, 748
- Heawood, Percy, 668
- Height, star, 840
- Height of full binary tree, 306
- Height of rooted tree, 691, 743
- Height-balanced trees, 748
- Hermeas, 2
- Hexadecimal expansion, 219–220
- Hexagon identity, 369
- Hilbert's Tenth Problem, 835
- HIV, 424
- Ho, Chung-Wu, 290
- Hoare, C. Anthony R., 323, 324
- Hoare triple, 323
- Hofstadter, Douglas, 333
- Hole-in-one, 279
- Hollywood graph, 593
 - paths in, 623–624
- Homeomorphic graphs, 663–665, 676
- Hopper, Grace Brewster Murray, 811
- Hops, 606
- Horner's method, 199
- Horse races, 27
- Host number (hostid), 341
- Hotbot, 734
- HTML, 796
- Huffman coding, 701–703, 744
 - variations of, 703
- Huffman, David A., 701, 702
- Husbands, jealous, 632–633
- Hybrid topology for local area network, 605
- Hydrocarbons, trees modeling, 688
- Hypercube, 607
- Hypothesis, 6
 - fallacy of denying, 69
 - inductive, 265
- Hypothetical syllogism, 66
- “Icosian Game,” 639, 640
- Idempotent laws
 - for Boolean algebra, 753, 757
 - for lattices, 585
 - for propositions, 24
 - for sets, 124
- Identifier, 559, 793
- Identities, combinatorial, 364–368
- Identity
 - combinatorial, proof of, 359, 386
 - hexagon, 369
 - Pascal's, 366–367, 387

- proving, generating functions for, 495
- Vandermonde's, 367–368
- Identity elements axiom, A-1
- Identity function, 138
- Identity laws
 - additive, A-1
 - for Boolean algebra, 752–754, 755
 - multiplicative, A-1
 - for propositions, 26
 - for sets, 124
- Identity matrix, 251, 257
- “If and only if” statement, 10
- If then statement, 6, 8
- Image, 136, 163
 - inverse, of set, 147
 - of set, 136
- Implicant, 770
 - prime, 770
 - essential, 770
- Implications, 6–10, 105
 - logical equivalences for, 26–27
- Incidence matrices, 614–615, 676
- Incident edge, 598
- Inclusion relation, 567
- Inclusion-exclusion principle, 122, 341–343, 500–504, 514
 - alternative form of, 506–507
 - applications with, 505–512
- Inclusive or, 5
- Incomparable elements in poset, 567, 582
- Incomplete induction, 284
- Increasing function, 137
- Increment, 206
- In-degree of vertex, 600, 675
- Independence number, 680
- Independent events, 399, 400, 405–406, 442
- Independent random variables, 434–436, 442
- Independent set of vertices, 680
- Index, of summation, 153
- Index registers, 672
- Indicator random variable, 440
- Indirect proof, 77
- Indistinguishable
 - boxes, 376–379
 - objects, 375, 376–379
 - objects, permutations with, 375
- Indistinguishable strings, 827
- Induced subgraph, 678
- Induction
 - complete, 284
 - generalized, 307–308
 - incomplete, 284
 - mathematical, 25, 263–279
 - principle of, 328
 - proofs by, 266–278
 - second principle of, 284
 - strong, 283–285, 328
 - structural, 304–306, 329
 - validity of, 278
 - well-ordered, 568–569, 582
- Inductive definitions, 295–308
 - of factorials, 296
 - of functions, 295–299, 328
 - of sets, 299–307, 329
 - of strings, 300
 - of structures, 299–307
- Inductive hypothesis, 265
- Inductive loading, 282, 293, 331
- Inductive reasoning, 264
- Inductive step, 264, 690, 706
- Inequality
 - Bernoulli's, 280
 - Bonferroni's, 415
 - Boole's, 415
 - Chebyshev's, 438–439, 442
 - of harmonic numbers, 272–273
 - Markov's, 441
 - proving by mathematical induction, 271
 - triangle, 102
- Inference, rules of, 64–67, 105
 - for program correctness, 323–324
 - for statements, 71–72
- Infinite graph, 589
- Infinite ladder, 263–264
 - by strong induction, 284
- Infinite series, 157
- Infinite set, 116, 158, 163
- Infix form, 719
- Infix notation, 719–721, 744
- Influence graphs, 593, 594
- Information flow, lattice model of, 575–576
- Initial assertion, 323, 329
- Initial conditions, 450, 513
- Initial position of a Turing machine, 829
- Initial state, 798, 805
- Initial term, 151
- Initial value, A-13
- Initial vertex, 541, 600
- Injection, 136
- Injective (one-to-one) function, 136, 139, 163
 - counting, 337
- Inorder traversal, 712, 714, 715, 718, 743
- Input alphabet, 798
- Insertion sort, 172, 174, 257
 - average-case complexity of, 432–433
 - worst-case complexity of, 196
- Instantiation
 - existential, 70, 71
 - universal, 70, 71
- Integer sequences, 151–153
- Integers, 112
 - addition of, algorithms for, 222–224
 - applications with, 231–244
 - axioms for, A-5
 - composite, 210, 211, 217, 257
 - division of, 200–208
 - even, 77
 - finding maximum element in finite sequence of, 170
 - greatest common divisor of, 215–217
 - large, computer arithmetic with, 237–238
 - least common multiple of, 216, 217
 - multiplication of
 - algorithms for, 224–226
 - fast, 475–476
 - odd, 77
 - partition of, 310
 - perfect, 218
 - pseudoprimes, 238–241, 245, 257
 - representations of, 219–222, 230–231
 - set of, 112
 - squarefree, 513
- Intension, of database, 531
- Interconnection networks for parallel computation, 606–607
- Interest, compound, 451
- Interior of simple polygon, 288
- Interior vertices, 551
- Internal vertices, 686, 743
- International Standard Book Number (ISBN), 209
- Internet
 - search engines on, 734
 - searching, 13
- Internet addresses, counting, 341
- Internet datagram, 346–347
- Internet Movie Database, 593
- Internet Protocol (IP) multicasting, 726
- Intersection
 - of fuzzy sets, 133
 - of multisets, 132
 - of sets, 121–122, 126–128, 163
- Intersection graph, 596
- Interval, open, 282
- Intractable problem, 197, 836
- Invariant for graph isomorphism, 615, 676
- Invariants, loop, 326–327, 329
- Inverse
 - of congruence modulo m , 234
 - of functions, 139–140, 163
 - of implication, 8, 105
 - of matrix, 255
- Inverse image of set, 147
- Inverse law
 - for addition, A-2
 - for multiplication, A-2
- Inverse relation, 528
- Inverse, multiplicative, 54
- Inversions, in permutation, expected number of, 430–431
- Inverter, 761, 781
- Invertible function, 140
- Invertible matrix, 255
- IP multicasting, 726
- Irrational numbers, 218
- Irrationality of $\sqrt{2}$, 80–81, 292, 332
- Irreflexive relation, 528
- ISBN (International Standard Book Number), 209
- Isobutane, 688
- Isolated vertex, 598, 676
- Isomorphic graphs, 611, 615–618, 676
 - paths and, 627–628
- Iterated function, 311
- Iterated logarithm, 311
- Iteration, 311
- Iterative algorithm, for Fibonacci numbers, 317
- Iterative procedure, for factorials, 317
- Iwaniec, Henryk, 215
- Jacobean rebellion, 145
- Jacquard loom, 27
- Java, 135
- Jealous husband problem, 632–633
- Jewish-Roman wars, 459
- Jigsaw puzzle, 292
- Jobs, assignment of, 513, 604
- Join, in lattice, 585
- Join of n -ary relations, 533–534
- Join of zero-one matrices, 252
- Joint authorship, 593–594
- Jordan Curve Theorem, 288
- Josephus, Flavius, 459
- Josephus problem, 459
- Jug, 103, 633
- Kakutani's problem, 102
- Kaliningrad, Russia, 633
- Karnaugh maps, 768–774, 781
- Karnaugh, Maurice, 767, 768
- Kempe, Alfred Bray, 668
- Key
 - composite, 532
 - for hashing, 205
 - primary, 531–532
- al-Khowarizmi, Abu Ja'far Mohammed ibn Musa, 168
- Kissing problem, 154
- Kleene closure, 804–805, 839
- Kleene, Stephen Cole, 805, 817
- Kleene's Theorem, 819–821, 839
- K-maps, 768–774, 781
- Kneiphof Island, 633
- Knights and knaves, 13–14
- Knight's tour, 647
 - reentrant, 647
- Knuth, Donald E., 172, 182, 184, 189
- Königsberg bridge problem, 633–634, 636, 638
- Kruskal, Joseph Bernard, 739, 740
- Kruskal's algorithm, 739–740, 744
- k -tuple graph coloring, 674
- Kuratowski, Kazimierz, 663
- Kuratowski's Theorem, 663–665, 676
- Labeled tree, 695
- Ladder, infinite, 263–264, 284
- Lagarias, Jeffrey, 102

- Lamé, Gabriel, 299
 Lamé's Theorem, 298
 Landau, Edmund, 183
 Landau symbol, 182
 Language, 785–793, 838
 context-free, 789
 formal, 786
 generated by grammar, 788
 natural, 785–786
 recognized by finite-state automata, 806–814, 839
 regular, 789
 Language recognizer, 801
 Lady Byron, 27
 Laplace, Pierre Simon, 393, 394, 420
 Laplace's definition of probability, 394
 Large numbers, law of, 407
 Lattice model of information flow, 575–576
 Lattice point, 330
 Lattices, 574–576, 582
 absorption laws for, 585
 associative laws for, 585
 bounded, 586
 commutative laws for, 585
 complemented, 586
 distributive, 586
 duality in, 587
 idempotent laws for, 585
 join in, 585
 meet in, 585
 modular, 587
 Law(s)
 absorption
 for Boolean algebra, 753, 754
 for lattices, 585
 for propositions, 24
 for sets, 124
 associative
 for addition, A–1
 for Boolean algebra, 753, 755
 for lattices, 585
 for multiplication, A–1
 for propositions, 24
 for sets, 124
 closure
 for addition, A–1
 for multiplication, A–1
 commutative
 for addition, A–1
 for Boolean algebra, 753, 755
 for lattices, 585
 for multiplication, A–1
 for propositions, 26
 for sets, 124
 compatibility
 additive, A–2
 multiplicative, A–2
 complement, for sets, 124
 complementation, for sets, 124
 completeness, A–2
 De Morgan's
 for Boolean algebra, 753, 757
 for propositions, 22, 25–26
 proving by mathematical induction, 274–275
 for quantifiers, 40–41
 for sets, 124
 of detachment, 65
 distributive, A–2
 for Boolean algebra, 752, 753, 755
 for propositions, 23, 24
 for sets, 124
 domination
 for Boolean algebra, 753
 for propositions, 27
 for sets, 124
 of double complement, in Boolean algebra, 753, 755
 idempotent
 for Boolean algebra, 753, 757
 for lattices, 585
 for propositions, 24
 for sets, 124
 identity
 additive, A–1
 for Boolean algebra, 752–754, 755
 multiplicative, A–1
 for propositions, 26
 for sets, 124
 inverse
 for addition, A–2
 for multiplication, A–2
 of large numbers, 407
 of mathematical induction, A–5
 negation, for propositions, 26
 used to prove basic facts, A–2 – A–5
 transitivity, A–2
 trichotomy, A–2
Laws of Thought, The (Boole), 3, 5, 420, 749
 Leaf, 686, 743
 Least common multiple (LCM), 216–217, 257
 Least element of poset, 573, 582
 Least upper bound, 582, A–2
 of poset, 574
 Left child of vertex, 687
 Left subtree, 687
 Lemma, 75, 105
 Length of bit string, 15, 454–455
 Length of path
 in directed graph, 546
 in weighted graph, 648
 Length of string, 151
 recursive definition of, 301
 “Less than or equals” relation, 567
 Level of vertex, 691, 743
 Level order of vertex, 745
 Lexicographic ordering, 307, 382–384, 385, 569–570
Liber Abaci (Fibonacci), 298
 Light fixture controlled by three switches, circuit for, 764
 Limit, definition of, 55
 Linear array, 606, 607
 Linear bounded automata, 825
 Linear combination, gcd as, 232
 Linear complexity of algorithms, 197
 Linear congruences, 234–235, 257
 Linear congruential method, 206
 Linear equations, simultaneous, 256
 Linear homogenous recurrence relations, 460, 461–467, 513
 Linear nonhomogenous recurrence relations, 467–471, 514
 Linear ordering, 568, 582
 Linear search algorithm, 170, 257
 average-case complexity of, 430–431
 complexity of, 194, 197
 recursive, 314
 Linearity of expectations, 429–431, 442
 Linearly ordered set, 568, 582
 Lists, merging two, 319
 Literal, 758, 781
 Little-*o* notation, 192
 Littlewood, John E., 93
 Loading, inductive, 282, 293, 331
 Loan, 457
 Lobsters, 471
 Local area networks, 605–606
 Logarithm, iterated, 311
 Logarithmic complexity of algorithms, 197
 Logarithmic function, A–7 – A–9
 Logic, 1–58, 104–105
 fuzzy, 19
 propositional, 2
 rules of inference for, 64–67
 Logic gates, 760–765
 AND, 761, 781
 combination of, 761–763
 NAND, 766
 NOR, 766
 OR, 761, 781
 threshold, 783
 Logic Problem, 74
 Logic programming, 45–46
 Logic puzzles, 13–15
 Logical connectives, 4
 Logical equivalences, 22–27, 39, 105
 in predicate calculus, 39
 in propositional calculus, 22–27
 Logical expressions
 translating English sentences into, 10–11, 42–43, 56–57
 translating to English sentences, 42–43
 Logical operators, 104
 functionally complete, 29
 precedence of, 10–11
 Long-distance telephone network, 594
 Loom, Jacquard, 27
 Loop constructions, A–13 – A–14
 Loop invariants, 326–327, 329
 Loops
 in directed graphs, 541, 582
 within loops, A–14
 nested, 51–52
 Lottery, 394–395
 Lovelace, Countess of (Augusta Ada), 25, 27
 Lower bound
 of lattice, 586
 of poset, 574, 582
 Lower limit of summation, 153
 Lucas, Edouard, 452
 Lucas numbers, 330, 471
 Lucas-Lehmer test, 213
 Lucky numbers, 518
 Łukasiewicz, Jan, 719, 744
 Lyceum, 2
 Lycos, 734
 McCarthy, John, 332
 McCarthy 91 function, 331
 McCluskey, Edward J., 775
 Machine minimization, 810
 Machines
 delay, 799–800
 finite-state, 785, 798, 839
 with no output, 804–814
 with outputs, 798–801
 Mealy, 801
 Moore, 801
 Turing, 785, 825, 826, 827–832, 839
 computing functions with, 831–832
 definition of, 828–830
 nondeterministic, 832
 sets recognized by, 830–831
 types of, 832
 vending, 797–798
MAD Magazine, 184
 Magic tricks, 14
 Majority voting, circuit for, 763
 Maps
 coloring of, 666–667
 Karnaugh, 768–774
 Markov's inequality, 441
 Markup languages, 796
m-ary tree, 686, 743
 complete, 694
 full, 686, 690
 height of, 692–693
 Master Theorem, 479
 Matching, 605
 maximal, 605
Mathematical Analysis of Logic, The (Boole), 5
 Mathematical induction, 25, 263–279
 Axiom of, A–5
 errors in, 278–279
 generalized, 307–308
 inductive loading with, 282
 principle of, 328

- proofs
 - by, 266–278
 - of divisibility facts, 273
 - errors in, 278–279
 - of inequalities, 271
 - of results about algorithms, 275
 - of results about sets, 273–275
 - of summation formulae, 267
- second principle of, 284
- strong, 283–285, 328
- structural, 304–306, 329
- validity of, 278
- Mathematician's Apology, A* (Hardy), 93
- Matrix (matrices), 246–254
 - addition of, 247–248
 - adjacency, 612–614, 676
 - counting paths between vertices by, 628–629
 - Boolean product of, 252–254, 257
 - column of, 247
 - definition of, 257
 - diagonal, 255
 - identity, 251, 257
 - incidence, 614–615, 676
 - inverse of, 255
 - invertible, 255
 - multiplication of, 249–250
 - algorithms for, 249–250
 - fast, 476–478
 - powers of, 251–252
 - representing relations using, 538–540
 - row of, 247
 - sparse, 614
 - square, 251
 - symmetric, 251, 257
 - transposes of, 251, 257
 - upper triangular, 260
 - zero-one, 252–254, 257
 - Boolean product of, 252–254
 - join of, 252
 - meet of, 252
 - representing relations using, 538–540
 - of transitive closure, 549–550
 - Warshall's algorithm and, 550–553
- Matrix-chain multiplication, 250
- Maurolico, Francesco, 265
- Maximal element of poset, 572, 582
- Maximal matching, 605
- Maximum element
 - in finite sequence, 169
 - time complexity of finding, 194
- Maximum, of sequence, 475
- Maximum satisfiability problems, 445
- Maximum spanning tree, 742
- Maxterm, 760
- Mealy, G. H., 801
- Mealy machines, 801, 839
- Mean, 178
 - arithmetic, 95, 282
 - deviation from, 439
 - geometric, 95, 282
 - harmonic, 103
 - quadratic, 103
- Median, 178
- Meet, in lattice, 585
- Meet of zero-one matrices, 252
- Meigu, Guan, 638
- Members, 112, 163
- Membership, degree of, 132
- Membership tables, 126, 163
- Ménages, problème des, 518
- Merge sort, 172, 317–321, 329, 475
 - complexity of, 479
 - recursive, 318
- Merging two lists, 319
- Mersenne, Marin, 212, 213
- Mersenne primes, 212, 257
- Mesh network, 606–607
- Mesh of trees, 748
- Metacharacters, 796
- Metafont, 184
- Method(s)
 - Critical Path, 587
 - Horner's, 199
 - linear congruential, 206
 - probabilistic, 413–414, 442
 - proof, 75–102, 105
 - Quine-McCluskey, 767, 775–779
- Millbanke, Annabella, 27
- Millennium problems, 836
- Miller's test for base b , 245
- Minimal element of poset, 572, 582
- Minimization
 - of Boolean functions, 766–779, 781
 - of combinational circuits, 766–779
- Minimum dominating set, 679
- Minimum, of sequence, 475
- Minimum spanning forest, 742
- Minimum spanning trees, 737–741, 744
- Minmax strategy, 706, 743
- Minterm, 758, 781
- Mistakes, in proofs, 83–84
- Mixed graph, 591
- mod, 202, 203, 205–207, 225–226
- Mode, 178
- Modeling
 - computation, 785–837
 - with graphs, 592–595
 - with recurrence relations, 450–456
 - with trees, 688–690
- Modular arithmetic, 203–205
- Modular exponentiation, 226–227
 - recursive, 313
- Modular lattice, 587
- Modular properties, in Boolean algebra, 757
- Modulus, 206
- Modus ponens, 65, 66
- Modus tollens, 66
- Mohammed's scimitars, 637
- Molecules, trees modeling, 688
- Monotone decreasing property of graph, 681
- Monotone increasing property of graph, 681
- Monotonic grammars, 790
- Monte Carlo algorithms, 411–413
- Montmort, Pierre Raymond de, 511, 518
- Monty Hall Three Door Puzzle, 398, 425
- Moore, E. F., 801
- Moore machine, 801
- Moth, 811
- Motorized pogo stick, 750
- m -tuple, 533
- Muddy children puzzle, 14–15
- Multicasting, 726
- Multigraphs, 590, 675
 - Euler circuit of, 637
 - Euler path of, 637
 - undirected, 592
- Multinomial coefficient, 382
- Multinomial Theorem, 382
- Multiple
 - of integers, 201
 - least common, 216–217, 257
 - Multiple edges, 590, 591, 592, 675
 - Multiple output circuit, 764
 - Multiplexer, 766
- Multiplication
 - of functions, 135–136
 - of integers
 - algorithms for, 224–226
 - fast, 475–476
 - of matrices, 249–250
 - algorithms for, 249–250
 - fast, 476–478
 - matrix-chain, 250
- Multiplicative Compatibility Law, A–2
- Multiplicative inverse, 54
- Multiplicity of membership in multisets, 132
- Multiplier, 206
- Multisets, 132
- Mutually independent events, 444
- Mutually independent trials, 406
- Mutually relative prime, 260
- Naive set theory, 112
- Namagiri, 94
- NAND, 29, 759, 781
- NAND gate, 766
- n -ary relations, 530–536, 581
 - domain of, 530
 - operations on, 532–535
- Natural language, 785–786
- Natural numbers, 112
- Naur, Peter, 792
- NAUTY, 618
- Naval Ordnance Laboratory, 811
- Navy WAVES, 811
- n -cubes, 602
- Necessary, 6
 - and sufficient, 9
- Negation
 - of existential quantification, 39
 - of nested quantifiers, 57–58
 - of propositions, 3–4, 104
 - of universal quantification, 39
- Negation laws, for propositions, 26
- Negation operator, 4
- Negative
 - false, 419
 - true, 419
- Neighbors in graphs, 598
- Neptune, 424
- Nested loops, 51–52
- Nested quantifiers, 50–58
 - negating, 57–58
- Network
 - computer, 590
 - with diagnostic lines, 591
 - interconnection networks, 606–607
 - local area networks, 605–606
 - multicasting over, 726
 - with multiple lines, 590
 - with multiple one-way lines, 592
 - with one-way lines, 591
 - gating, 761–763
 - depth of, 766
 - examples of, 763–764
 - minimization of, 766–779
 - tree-connected, 689–690
- Network number (netid), 341
- New Foundations of Mathematical Logic* (Quine), 776
- New Jersey crags, 44
- Newton-Pepys problem, 447
- Niche overlap graph, 592, 593
- Nickels, 175–176
- Nim, 704–705, 707
- Nodes, 541, 589
- Nonconformists, 420
- Nonconstructive existence proof, 91, 105
- Noncontracting grammar, 790
- Nondeterministic finite-state automata, 811–814, 820, 822, 839
- Nondeterministic Turing machine, 832
- Nonoverlapping ears, 292
- Nonplanar graphs, 663–665
- Nonterminals, 787
- NOR, 29, 759, 781
- NOR gate, 766
- NOT, 13
- Notation
 - big- O , 180–186
 - big- Ω , 182, 189–190
 - big- Θ , 182, 189–190
 - dependency, 784
 - infix, 719–721, 744
 - little- o notation, 192
 - one's complement, 230
 - Polish, 719, 744

- Notation—*Cont.*
 postfix, 719–721, 744
 prefix, 719–721, 744
 for products, 162
 well-formed formula in, 724
 reverse Polish, 720, 744
 set builder, 112
 summation, 153
 two's complement, 230–231
- Noun, 786
- Noun phrase, 786
- Nova*, 101
- NP, class, 197, 836
- NP-complete problems, 197, 655, 768, 836
- n -queens problem, 731–732
- n -regular graph, 610
- n -tuples, 531–532
 ordered, 117
- Null quantification, 49
- Null set, 114, 163
- Null string, 151, 787
- Number(s)
 Bacon, 623
 Bell, 566
 Carmichael, 240–241, 257
 Catalan, 456
 chromatic, 667, 668, 676
 edge, 674
 crossing, 666
 Cunningham, 214
 Erdős, 584, 623, 624
 Fibonacci, 297–298, 316–317, 709
 formula for, 463–464
 iterative algorithm for, 317
 rabbits and, 451–452
 recursive algorithms for, 316
 harmonic, 192
 inequality of, 272–273
 independence, 680
 irrational, 218
 large, law of, 407
 Lucas, 330, 471
 lucky, 518
 natural, 112
 perfect, 218
 pseudorandom, 206–207
 Ramsey, 352
 rational, 105, 113
 countability of, 158, 163
 real, 113, A-1 – A-5
 Stirling, of the second kind, 378
 Ten Mosted Wanted, 214
 Ulam, 165
- Number theory, 200–244
- Numbering plan, 337
- Number-theoretic functions, 831
- Object(s), 111–112
 distinguishable, 376
 and distinguishable boxes, 376–377
 and indistinguishable boxes, 377–378
 indistinguishable, 376
 and indistinguishable boxes, 377, 378–379
 unlabeled, 376
- Octahedral die, 443
- Octal expansion, 220, 257
- Odd integer, 77
- Odd pie fights, 276
- Odlyzko, Andrew, 154
- One's complement expansion, 230
- One-to-one correspondence, 139, 160, 163
- One-to-one (injective) function, 136, 139, 163
 counting, 337
- “Only if” statement, 9
- Onto (surjective) function, 137–138, 139, 163
 number of, 509–510, 514
- Open interval, 282
- Open problems, 214–215
- Operands, well-formed formula of, 301
- Operations
 bit, 15
 estimating number of, of algorithms, 180–182
 integer, algorithms for, 222–226
 on n -ary relations, 532–535
 on set, 121–130
- Operators
 bitwise, 15–16, 104
 functionally complete set of, for
 propositions, 29
 logical, 104
 functionally complete, 29
 precedence of, 10–11
 negation, 4
 selection, 532
 well-formed formula of, 301
- Opium, 424
- Optimal algorithm, 200
- Optimal for suitors, stable assignment, 293
- Optimization problems, 174–176
- OR, 13, 15–16
- OR gate, 761, 781
- Oracle of Bacon, 624
- Order, 190
 of quantifiers, 52–54
 same, 182
- Ordered n -tuples, 117
- Ordered pairs, 117, 120
- Ordered rooted tree, 687, 743, 745
- Ordering
 dictionary, 382
 lexicographic, 307, 382–384, 569–570
 linear, 568
 partial, 566–578, 582
 quasi-ordering, 585
 total, 568, 582
- Ordinary generating function, 484
- Ore, O., 641
- Ore's Theorem, 641, 647
- Organizational tree, 689
- Orientable graphs, 679
- Orientation of undirected graph, 679
- Quantifiers theorems stated as propositions
 involving, 10, 23, 24, 64
- Out-degree of vertex, 600, 675
- Output alphabet, 798
- Outputs
 finite-state machines with, 798–801
 finite-state machines without, 804–814
- P, class, 197, 836
- Pair
 devil's, 621
 ordered, 117
- Pairwise relatively prime integers, 216, 257
- Palindrome, 177, 346, 795
- Paradox, 112
 barber's, 19–20
 Russell's, 121
 St. Petersburg, 444
- Parallel algorithms, 606
- Parallel edges, 590
- Parallel processing, 199, 606
 tree-connected, 689–690
- Parent of vertex, 686, 743
- Parent relation, 526
- Parentheses, balanced strings of, 332
- Parse tree, 790–792, 839
- Parsing
 bottom-up, 791
 top-down, 791
- Partial correctness, 323
- Partial function, 149
- Partial orderings, 566–578, 582
 compatible total ordering from, 582
- Partially ordered set, 566, 582
 antichain in, 585
 chain in, 585
 comparable elements in, 568, 582
 dense, 581
 dual of, 579
 greatest element of, 573, 582
 Hasse diagram of, 571–572, 573, 582
 incomparable elements in, 568, 582
 least element of, 573, 582
 lower bound of, 574, 582
 maximal element of, 572, 582
 minimal element of, 572, 573, 582
 upper bound of, 574, 582
 well-founded, 581
- Partition, 499
 of positive integer, 310, 379
 refinement of, 565
 of set, 559–562, 582
- Pascal, 135
- Pascal, Blaise, 239, 366, 393, 400
- Pascal's identity, 366–367, 387
- Pascal's triangle, 366, 386
- Passwords, 63, 340
- Paths, 622–624
 in acquaintanceship graphs, 623
 in collaboration graphs, 623
 counting between vertices, 628–629
 in directed graphs, 546–547, 582, 676
 in directed multigraphs, 623
 Euler, 633–638, 676
 and graph isomorphism, 627–628
 Hamilton, 638–643, 676
 in Hollywood graph, 623–624
 of length n , 623
 length of, in weighted graph, 648
 shortest, 647–655, 676
 in simple graphs, 622
 terminology of, 622
 in undirected graphs, 622
- Payoff, 704
- Pearl Harbor, 811
- Pecking order, 680
- Peirce, Charles Sanders, 29, 32
- Peirce arrow, 29
- Pelc, A., 483
- Pendant vertex, 598, 676
- Pennies, 175–176
- Perfect
 integers, 218
 number, 218
 power, 87
- Peripatetics, 2
- Permutations, 355–357, 386
 generating, 382–384
 generating random, 445
 with indistinguishable objects, 375–376
 inversions in, expected number of,
 430–431
 with repetition, 371
- PERT (Program Evaluation and Review
 Technique), 587
- Petersen, Julius Peter Christian, 646
- Phrase-structure grammars, 787–790, 838
- Pick's Theorem, 292
- Pie fights, odd, 276
- Pigeonhole principle, 347–353, 387
 applications with, 351–353
 generalized, 349–351, 387
- Planar graphs, 657–665, 676
- Plato, 2
- Plato's Academy, 2
- Plouffe, Simon, 153
- PNF (prenex normal form), 62
- $P=NP$ problem, 836
- $P(n, r)$, 386
- Pogo stick, motorized, 750
- Pointer, position of, digital representation of,
 642–643
- Poisonous snakes, 702
- Poker hands, 358–359
- Polish notation, 719, 744
- Pólya, George, 102

- Polygon, 288
 - convex, 288
 - diagonal of, 288
 - exterior of, 288
 - interior of, 288
 - nonconvex, 288
 - with nonoverlapping ears, 292
 - sides of, 288
 - simple, 288
 - vertices of, 288
- Polynomial complexity of algorithms, 197
- Polynomials, rook, 518
- Polynomial-time problems, 836
- Polyominoes, 99, 277
- Poset, 566, 582
 - antichain in, 585
 - chain in, 585
 - comparable elements in, 567, 582
 - dense, 581
 - dual of, 579
 - greatest element of, 573, 582
 - Hasse diagram of, 571–572, 573, 582
 - incomparable elements in, 568, 582
 - least element of, 573, 582
 - lower bound of, 574, 582
 - maximal element of, 572, 582
 - minimal element of, 572, 573, 582
 - upper bound of, 574, 582
 - well-founded, 581
- Postcondition, 33, 323
- Positive
 - false, 419
 - true, 419
- Positive integers, 113
 - axioms for, A–5
- Positive rational numbers, countability of, 158
- Postfix form, 720
- Postfix notation, 719–721, 744
- Postorder traversal, 712, 715–716, 717, 718, 743
- Postulates, 75
- Power, perfect, 88
- Power, Boolean (of zero-one matrices), 252–254
- Power series, 485–488
 - formal, 485
- Power set, 116–117, 163
- Powers
 - of matrices, 251–252
 - of relation, 527, 581
- Pre-image, 147, 163
- Precedence graphs, 595
- Precedence levels, 11, 38
- Precondition, 33, 323
- Predicate, 30, 31
- Predicate calculus, 33
 - logical equivalence in, 39
- Prefix codes, 700–703, 743
- Prefix form, 719
- Prefix notation, 719–721, 744
 - well-formed formula in, 724
- Premise, 6, 44, 63
- Prenex normal form (PNF), 62
- Preorder traversal, 712–713, 718, 743
- Prim, Robert Clay, 738, 740
- Primality testing, 412–413
- Primary key, 531–532, 581
- Prime factorization, 211–212
 - for finding greatest common divisor, 216
 - for finding least common multiple, 216–217
 - uniqueness of, 233–234
- Prime implicant, 770, 781
 - essential, 770
- Prime Number Theorem, 213, 584
- Primes, 210–215
 - definition of, 257
 - distribution of, 213
 - infinite of, 212
 - Mersenne, 212, 257
 - mutually relative, 260
 - pseudoprimes, 238–241, 245, 257
 - relatively, 216, 257
 - sieve of Eratosthenes and, 210
 - twin, 215
- Prim's algorithm, 738–739, 744
- Princess of Parallelograms, 27
- Principia Mathematica* (Whitehead and Russell), 28, 114
- Principle(s)
 - of buoyancy, A–4
 - of counting, 335–340
 - duality, for Boolean identities, 754
 - of inclusion-exclusion, 122, 341–343, 500–504, 514
 - alternative form of, 506–507
 - applications with, 505–512
 - of mathematical induction, 328
 - pigeonhole, 347–353, 387
 - applications with, 351–353
 - generalized, 349–351, 387
 - of well-founded induction, 585
 - of well-ordered induction, 568–569
- Private key cryptography, 241
- Probabilistic algorithms, 393, 411–413, 442
- Probabilistic method, 413–414, 442
- Probabilistic primality testing, 412–413
- Probabilistic reasoning, 398
- Probability, discrete, 393–447
 - assigning, 401–403
 - of collision in hashing functions, 410–411
 - of combinations of events, 396–398, 403–404
 - conditional, 400, 404–405, 442
 - finite, 394–396
 - Laplace's definition of, 394
 - in medical test results, 419–420
- Probability distribution, 401
- Probability generating function, 499
- Probability theory, 393, 400–416
- Problem(s)
 - art gallery, 675
 - birthday, 409–411
 - bridge, 633–634, 636, 638
 - celebrity, 282
 - Chinese postman, 638
 - classes of, 835
 - Closest-Pair, 479–482
 - Collatz, 101–102
 - decision, 834
 - halting, 176–177, 834–835
 - hatcheck, 430, 510
 - Hilbert's tenth, 835
 - intractable, 197, 836
 - jealous husband, 632–633
 - Josephus, 459
 - Kakutani's, 102
 - kissing, 154
 - Logic, 74
 - maximum satisfiability, 445
 - Millennium, 836
 - Newton-Pepys, 447
 - NP-complete, 197, 655, 768
 - n -queens, 731–732
 - open, 214–215
 - optimization, 174–176
 - $P=NP$, 836
 - polynomial-time, 836
 - scheduling greedy algorithm for, 179
 - final exams, 671, 672
 - shortest-path, 647–655, 676
 - solvable, 197, 834
 - Syracuse, 102
 - tiling, 835
 - tractable, 197, 836
 - traveling salesman, 641, 649, 653–655, 676
 - Ulam's, 101–102
 - unsolvable, 197, 834, 835
 - utilities-and-houses, 657–659
 - yes-or-no, 834
- Problème de rencontres*, 511, 518
- Problème des ménages*, 518
- Problems, Millennium, 836
- Procedure statements, A–10, A–14 – A–15
- Processing
 - concurrent, 595
 - parallel, 199, 606
 - tree-connected, 689–690
- Product
 - Boolean, 749–750, 751, 757–759, 781
 - of zero-one matrices, 252–254, 257
 - Cartesian, 117–118
- Product notation, 162
- Product rule, 336
- Productions of grammar, 787
- Product-of-sums expansion, 760
- Program correctness, 322–328, 329
 - conditional statements for, 324–326
 - loop invariants for, 326–327
 - partial, 323
 - program verification for, 323
 - rules of inference for, 323–324
- Program Evaluation and Review Technique (PERT), 587
- Program verification, 323
- Programming, logic, 45–46
- Programming languages, 68, 135, 346, 559
- Progression(s)
 - arithmetic, 151
 - geometric, 150, 152
 - sums of, 270–271
- Projection of n -ary relations, 533, 582
- Prolog, 45
- Prolog facts, 45
- Prolog rules, 45
- Proof(s)
 - adapting existing, 96
 - by cases, 86–90, 105
 - combinatorial, 359, 386
 - conjecture and, 96–97
 - by contradiction, 80–83, 105
 - definition of, 75, 105
 - direct, 76–77, 105
 - of equivalence, 82–83
 - exhaustive, 87
 - existence, 91–92, 105
 - indirect, 77, 105
 - by mathematical induction, 266–278
 - methods of, 75–102, 105
 - mistakes in, 83–84
 - of recursive algorithms, 315–317
 - by structural induction, 304
 - trivial, 78, 79, 105
 - uniqueness, 92–93, 105
 - vacuous, 78, 105
- Proof strategy, 79, 86–102
- Proper subset, 115, 163
- Property
 - Archimedean, A–5
 - Completeness, A–2
 - Well-ordering, 290–291, 328, A–5
- Proposition(s), 1–6, 75
 - compound, 3, 10–11, 21, 23, 104
 - consistent, 104
 - disjunctive normal form of, 29
 - dual of, 29
 - satisfiable, 30
 - well-formed formula for, 301
 - conjunction of, 4–5, 104
 - definition of, 2, 104
 - disjunction of, 4–5, 104
 - equivalent, 8, 21–30, 104
 - functionally complete set of operators for, 29
 - negation of, 3–4, 104
 - rules of inference for, 64–67
 - truth value of, 3
- Propositional equivalences, 21–30
- Propositional function, 30, 31, 105
 - existential quantification of, 36
 - universal quantification of, 33
- Propositional logic, 2

- Protestant Nonconformists, 420
 Pseudocode, 121, 169, A-10 – A-15
 Pseudograph, 590, 592, 675
 Pseudoprimes, 238–239, 245, 257
 Pseudorandom numbers, 206–207
 Public key cryptography, 231, 241–244, 257
 Pure multiplicative generator, 207
 Pushdown automaton, 825
 Puzzle(s)
 Birthday Problem, 409–411
 Icosian, 639, 640
 jigsaw, 292
 logic, 13–15
 Monty Hall Three Door, 398, 425
 muddy children, 14–15
 Reve's, 454
 river crossing, 632
 Tower of Hanoi, 452–454
 "Voyage Around the World," 639
 zebra, 21
 Quad trees, 748
 Quadratic mean, 103
 Quadratic residue, 246
 Quality control, 412
 Quantification
 existential, 36, 105
 negation of, 39
 as loops, 51–52
 null, 49
 universal, 34, 105
 negation of, 39
 Quantifiers, 34–37
 De Morgan's laws for quantifiers, 40–41
 domain of, 34
 existential, 36–37
 negating, 39–41
 nested, 50–58
 negating, 57–58
 order of, 52–54
 precedence of, 38
 rules of inference for, 70
 truth set of, 119
 uniqueness, 37
 universal, 34–36
 using in system specifications, 43–44
 using set notation with, 119
 with restricted domains, 38
 Quarters, 175–176
 Quasi-ordering, 585
 Queens on chessboard, 679
 Question, begging, 84, 105
 Quick sort, 172, 322
 Quine, Willard van Orman, 775, 776
 Quine-McCluskey method, 768, 775–779
 Quotient, 202, A-6
 Automaton, 817
 Rabbits, 451–452
 Races, horse, 27
 Radius of graph, 680
 Rado, Tibor, 838
 Ramanujan, Srinivasa, 93, 94
 Ramaré, O., 215
 Ramsey, Frank Plumpton, 352
 Ramsey number, 352
 Ramsey theory, 352
 Random permutation, generating, 445
 Random simple graph, 680–681
 Random variables, 402, 408–409
 covariance of, 442
 definition of, 442
 distribution of, 408, 442
 geometric, 433–434
 expected values of, 402–403, 426–429, 442
 independent, 434–436, 442
 indicator, 440
 standard deviation of, 436
 variance of, 426–429, 442
 Range of function, 134, 163
 Ratio, common, 150
 Rational numbers, 105, 113
 countability of, 158, 163, 218
r-combination, 357, 386
 Reachable state, 841
 Real numbers, 113
 constructing, A-5
 least upper bound, A-2
 uncountability of, 160
 upper bound, A-2
 Reasoning
 backward, 94
 circular, 84, 105
 deductive, 264
 forward, 94
 inductive, 264
 probabilistic, 398
 Recognized language, 806–814, 839
 Recognized strings, 806
 Records, 531
 Recurrence relations, 449–460
 associated homogenous, 467
 definition of, 450, 513
 divide-and-conquer, 474–482
 initial condition for, 450
 linear homogenous, 460, 461–467, 513
 linear nonhomogenous, 467–471, 514
 modeling with, 450–456
 simultaneous, 472
 solving, 460–471
 generating functions for, 493–495
 Recursion, 294
 Recursive algorithms, 311–321, 329
 for binary search, 314
 for computing a^n , 312
 for computing greatest common divisor, 313
 correctness of, 315
 for factorials, 311–312
 for Fibonacci numbers, 316
 for linear search, 314
 for modular exponentiation, 313
 proving correct, 315–317
 trace of, 312, 313
 Recursive definitions, 263, 295–308
 of extended binary trees, 302
 of factorials, 311–312
 of full binary trees, 33
 of functions, 295–299, 328
 of sets, 299–307, 329
 of strings, 300
 of structures, 299–307
 Recursive merge sort, 318
 Recursive modular exponentiation, 313
 Recursive sequential search algorithm, 314
 Recursive step, 299
 Reentrant knight's tour, 647
 Refinement, of partition, 565
 Reflexive closure of relation, 544, 582
 Reflexive relation, 522, 581
 representing
 using digraphs, 541–542
 using matrices, 538
 Regions of planar representation graphs, 659, 676
 Regular expressions, 818, 839
 Regular grammars, 789, 817, 821–824, 839
 Regular graph, 610
 Regular language, 789
 Regular sets, 817–819, 820, 839
 Relation(s), 519–587
 antisymmetric, 523–524, 582
 asymmetric, 528
 binary, 519, 581
 circular, 584
 closures of, 544–553, 582
 reflexive, 544, 582
 symmetric, 545, 582
 transitive, 544, 547–550, 582
 computing, 550–553
 combining, 525–527
 complementary, 528
 composite of, 526–527, 581
 connectivity, 547, 582
 counting, 522
 covering, 579
 diagonal, 545
 divide-and-conquer recurrence, 474
 divisibility, 567
 domains of, 530
 equivalence, 555–562, 582
 functions as, 520
 "greater than or equal," 566–567
 inclusion, 566
 inverse, 528, 581
 irreflexive, 528
 "less than or equals," 567
 n-ary, 530–536, 581
 domain of, 531
 operations on, 532–535
 parent, 526
 paths in, 546–547
 powers of, 527, 581
 properties of, 522–525
 reflexive, 522, 581
 representing
 using digraphs, 541–542
 using matrices, 538–540
 on set, 118, 521–522
 symmetric, 523–524, 582
 transitive, 524–525, 527, 582
 Relational database model, 531–532, 581
 Relatively prime integers, 216, 257
 Remainder, 202, 203, 257
 Rencontres, 511, 518
 Repetition
 combinations with, 371–375
 permutations with, 371
 Replacement
 sampling with, 396
 sampling without, 396
 Representations
 binary, 219
 decimal, 168, 219
 hexadecimal, 219–220
 octal, 220
 one's complement, 230
 unary, 831
 Representative, of equivalence class, 558
 Resolution, 66, 68
 Results, 75
 Reverse Polish notation, 720, 744
 Reve's puzzle, 454
 Right child of vertex, 687
 Right subtree, 687
 Right triominoes, 100, 277, 283
 Ring topology for local area network, 605
 River crossing puzzle, 632
 Rivest, Ronald, 241, 242
 Roadmaps, 595
 Rocket-powered Frisbee, 750
 Rook polynomials, 518
 Rooted Fibonacci trees, 695
 Rooted spanning tree, 737
 Rooted trees, 302, 685–688
 balanced, 691, 743
 binomial, 745
 B-tree of degree k , 745
 decision trees, 698–700
 definition of, 743
 height of, 691, 743
 level order of vertices of, 745
 ordered, 687, 743, 745
 S_k -tree, 745
 Roots, characteristic, 461
 Round-robin tournaments, 291, 593, 594
 Row of matrix, 247
 Roy, Bernard, 551
 Roy-Warshall algorithm, 550–553

- r -permutation, 355, 386
- RSA system, 241–244
 - decryption, 243
 - encryption, 242–243
- Rule
 - product, 336
 - sum, 338, 386–387
- Rules of inference, 64–67, 70, 105
 - for program correctness, 323–324
 - for propositions, 64–67
 - for quantifiers, 70
 - for statements, 71–72
- Run, 440
- Russell, Bertrand, 112, 114
- Russell's paradox, 121

- St. Petersburg Paradox, 444
- Same order, 182
- Samos, Michael, 480
- Sample space, 394
- Sampling
 - with replacement, 396
 - without replacement, 396
- Sanskrit, 792
- Satisfiability problem, maximum, 445
- Satisfiable compound propositions, 30
- Saturated hydrocarbons, trees modeling, 688
- Scheduling problems, 179
 - final exams, 671, 672
 - greedy algorithm for, 275–276
 - software projects, 581
 - tasks, 577–578
- Scimitars, Mohammed's, 637
- Scope of a quantifier, 38
- Screw, Archimedes, A–4
- Search engines, 734
- Search trees, binary, 695–698, 743
- Searching algorithms, 170–172, 257
 - binary, 171–172, 257, 474
 - Boolean, 13
 - breadth-first, 729–731, 744
 - complexity of, 194–195, 197
 - depth-first, 726–729, 744
 - applications with, 732–734
 - in directed graphs, 732–734
 - linear, 170, 257
 - average-case complexity of, 430–431
 - complexity of, 195, 197
 - recursive, 314
 - recursive binary, 314
 - recursive linear, 314
 - recursive sequential, 314
 - ternary, 178
 - Web, 13
- Second principle of mathematical induction, 284
- Secrets, 265–266
- Seed, 206
- Sees, 675
- Selection operator, 532, 582
- Selection sort, 172, 178–179
- Self-complementary graph, 620
- Self-converse directed graph, 679
- Self-dual, 782
- Self-generating sequences, 333
- Semantics, 785
- Sentence, 786, 787
- Sequence(s), 149–153
 - de Bruijn, 682
 - definition of, 149
 - finding maximum and minimum of, 475
 - generating functions for, 484
 - integer, 151–153
 - self-generating, 333
 - strictly decreasing, 351
 - strictly increasing, 351
- Sequential search, 170
- Serial algorithms, 606
- Series
 - geometric, 155
 - harmonic, 273
 - infinite, 157
 - power, 485–488
 - formal, 485
- Set(s), 111–121
 - cardinality of, 116, 158–160, 163
 - complement of, 123, 163
 - computer representation of, 129–130
 - countable, 158, 218
 - cut, 746
 - definition of, 111, 163
 - description of, 111–112
 - difference of, 123, 163
 - symmetric, 131, 163
 - disjoint, 122
 - dominating, 679
 - elements of, 112, 163
 - empty (null), 114, 163
 - equal, 113, 163
 - finite, 116, 158, 163, 500, 514
 - combinations of, 384–385
 - subsets of
 - counting, 338
 - number of, 273–274
 - union of three, number of elements in, 501–503, 514
 - union of two, number of elements in, 500, 514
 - fuzzy, 132–133
 - complement of, 132
 - intersection of, 133
 - union of, 133
 - guarding, 675
 - identities of, 124–126
 - image of, 136
 - infinite, 116, 159, 163
 - intersection of, 121–122, 126–128, 163
 - inverse image of, 147
 - linearly ordered, 568, 582
 - members of, 112, 163
 - not recognized by finite-set automata, 824–825
 - operations on, 121–130
 - partially ordered, 566, 582
 - antichain in, 585
 - chain in, 585
 - comparable elements in, 567, 582
 - dense, 581
 - dual of, 579
 - greatest element of, 573, 582
 - Hasse diagram of, 571–572, 573, 582
 - incomparable elements in, 567, 568, 582
 - least element of, 573, 582
 - lower bound of, 574, 582
 - maximal element of, 572, 582
 - minimal element of, 572, 573, 582
 - upper bound of, 574, 582
 - well-founded, 581
 - partition of, 560–561, 582
 - power, 116–117, 163
 - proofs of facts about, by mathematical induction, 273
 - recognized by finite-set automata, 819–821
 - recognized by Turing machines, 830–831
 - recursively defined, 299–307, 329
 - regular, 817–819, 820, 821–824, 839
 - relation on, 118, 521–522
 - singleton, 114
 - successor of, 132
 - symmetric difference of, 131, 163
 - totally ordered, 568, 582
 - truth, 119
 - type, 113
 - uncountable, 160
 - union of, 121, 126–128, 163
 - universal, 113, 163
 - well-ordered, 332, 568, 582
- Set builder, 112
- Set notation, with quantifiers, 119

- Sex, 406
- Shaker sort, 172, 259
- Shamir, Adi, 241, 242
- Shannon, Claude Elwood, 749, 750
- Sheffer, Henry Maurice, 28, 29
- Sheffer stroke, 28, 29
- Shift cipher, 208
- Shift register, 784
- Shifting, 224
- Shortest-path algorithm, 649–653
- Shortest-path problems, 647–655, 676
- Sibling of vertex, 686, 743
- Sides of a polygon, 288
- Sieve of Eratosthenes, 210, 507–509, 514
- Simple circuit, 622
- Simple directed graph, 591
- Simple graphs, 590, 601–604, 675
 - coloring of, 667
 - connected planar, 659–663
 - crossing number of, 666
 - dense, 614
 - edges of, 590, 608
 - isomorphic, 615–618, 676
 - orientation of, 679
 - paths in, 622
 - random, 680–681
 - self-complementary, 620
 - with spanning trees, 725–726
 - sparse, 614
 - thickness of, 666
 - vertices of, 590, 608
- Simple polygon, 288
 - triangulation of, 288
- Simplification rule of inference, 66
- Simultaneous linear equations, 256
- Singleton set, 114
- Sink, 841
- Six Degrees of Separation* (Guare), 623
- S_k -tree, 745
- Sloane, Neil, 154
- Smullyan, Raymond, 13, 14
- Snakes, poisonous, 702
- Sneakers* (movie), 242
- Soccer players, 424
- Socks, 353
- Software systems, 12
- Sollin's algorithm, 743
- Solution of a recurrence relation, 450
- Solvable problem, 197
- Solve a decision problem, 836
- Solving using generating functions,
 - counting problems, 488–493
 - recurrence relations, 493–495
- Somerville, Mary, 27
- Sort
 - binary insertion, 172, 179
 - bubble, 173, 257
 - bidirectional, 173
 - worst-case complexity of, 194–195
 - insertion, 172, 174, 257
 - average-case complexity of, 432–433
 - worst-case complexity of, 196
 - merge, 172, 317–321, 329, 475
 - complexity of, 479
 - recursive, 318
 - quick, 172, 322
 - selection, 172, 178–179
 - shaker, 172, 259
 - tournament, 172, 708
- Sorting algorithms, 172–174
 - complexity of, 195–196
 - topological, 576–578, 582
- Space complexity of algorithms, 193
- Space probe, 424
- Space, sample, 394
- Spam, 421–423
- Spam filters, Bayesian, 421–423
- Spanning forest, 736
 - minimum, 744

- Spanning trees, 724–734, 744
 - building
 - by breadth-first search, 729–731
 - by depth-first search, 726–729
 - definition of, 724
 - degree-constrained, 746
 - distance between, 737
 - in IP multicasting, 726
 - maximum, 742
 - minimum, 737–741, 744
 - rooted, 737
- Sparse graphs, 614, 741
- Sparse matrix, 614
- Specifications, system, 12, 43–44
- Spiders, Web, 734
- SQL, 535–536
- Square matrix, 251
- Squarefree integer, 513
- Stable assignment, 179, 293
- Standard deviation, 436
- Star height, 840
- Star topology for local area network, 605
- Start symbol, 787
- State diagram for finite-state machine, 798
- State table for finite-state machine, 798
- State tables, 802
- Statement(s), A–10
 - assignment, A–10 – A–11
 - begin, A–11, A–14
 - blocks of, A–11
 - conditional, 6
 - for program correctness, 324–326
 - end, A–11, A–14
 - “if and only if,” 8, 9
 - if then, 6, 8
 - procedure, A–10
 - rules of inference for, 71–72
- States, 798
 - reachable, 841
 - transient, 841
- Steroids, 424
- Stirling, James, 145
- Stirling numbers of the second kind, 378
- Stirling’s formula, 146
- Straight triomino, 100
- Strategies
 - minmax, 706, 743
- Strategy, proof, 79, 86–102
- Strictly decreasing function, 137
- Strictly decreasing sequence, 351
- Strictly increasing function, 137
- Strictly increasing sequence, 351
- Strings, 151
 - bit, 15, 104, 129–130
 - concatenation of, 300, 804, 839
 - counting, 336
 - without consecutive 0s, 454–455
 - decoding, 700–701
 - distinguishable, 827
 - empty, 151, 787
 - generating next largest, 385
 - indistinguishable, 827
 - length of, 15, 151, 454–455
 - recursive definition of, 301
 - lexicographic ordering of, 568
 - of parentheses, balanced, 332
 - recognized (accepted), 806, 836
 - recursively defined, 300
 - ternary, 458
- Strong induction, 283–285, 328
- Strongly connected components of graphs, 627, 676
- Strongly connected graphs, 626
- Structural induction, 304–306, 329
 - proof by, 304
- Structured query language, 535–536
- Structures, recursively defined, 299–307
- Subgraph, 607, 678
 - induced, 678
- Subsequence, 351
- Subsets, 114, 163
 - of finite set
 - counting, 338
 - number of, 273–274
 - proper, 81, 163
 - sums of, 732
- Subtractors
 - full, 766
 - half, 766
- Subtree, 686, 687, 743
- Success, 406
- Successor
 - of integer, A–5
 - of set, 132
- Sufficient, 6
 - necessary and, 9
- Suites, 179
- Suitors, 179, 293
 - optimal for, 293
- Sum(s)
 - Boolean, 749–750, 751, 757–759, 781
 - of first n positive integers, 157, 162, 267–268
 - of functions, 135–136
 - of geometric progressions, 270–271
 - of matrices, 247–248
 - of multisets, 132
 - of subsets, 732
- Sum rule, 338, 386–387
- Summation formulae
 - proving by mathematical induction, 267–270
- Summation notation, 153
- Summations, 153–158
 - double, 156
 - index of, 153
 - lower limit of, 153
 - proving, by mathematical induction, 267
 - upper limit of, 153
- Sum-of-products expansions, 757–759, 781
 - simplifying, 766–779
- Sun-Tsu, 235
- Surjective (onto) function, 137–138, 139, 163
 - number of, 509–510, 514
- Switching circuits, hazard-free, 784
- Syllogism
 - disjunctive, 66
 - hypothetical, 66
- Symbol(s), 787
 - Landau, 182
 - start, 787
 - terminal, 787
- Symbolic Logic* (Carroll), 44
- Symbolic Logic* (Venn), 115
- Symmetric closure of relation, 545, 582
- Symmetric difference, of sets, 131, 163
- Symmetric matrix, 251, 257
- Symmetric relation, 523–524, 581
 - representing
 - using digraphs, 542
 - using matrices, 538–539
- Syntax, 785
- Syracuse problem, 102
- System specifications, 12, 43–44
 - consistent, 12
 - using quantifiers in, 43–44
- Tables
 - membership, 126
 - state, 798, 802
 - truth, 3
- Tautology, 21, 27, 65, 66, 104
- Tee-shirts, 344
- Telephone call graph, 625
- Telephone calls, 594
- Telephone lines
 - computer network with diagnostic, 591
 - computer network with multiple, 590
 - computer network with multiple one-way, 592
 - computer network with one-way, 591
- Telephone network, 594, 625
- Telephone number, 594
- Telephone numbering plan, 337
- Telescoping sum, 162
- “Ten Most Wanted” numbers, 214
- Term, initial, 151
- Terminal vertex, 541, 600
- Terminals, 787
- Ternary expansion, balanced, 230
- Ternary search algorithm, 178
- Ternary string, 458
- Test
 - Lucas-Lehmer, 212
 - Miller’s, 245
 - primality, 412–413
 - probabilistic primality, 412–413
 - Turing, 27
- Tetrominoes, 104
- TeX, 184
- Theorem(s), 75, 105
 - Archimedean property, A–5
 - Art Gallery, 675
 - Bayes’, 417–425
 - Binomial, 363–365, 387
 - Chinese remainder, 235–237, 258
 - Dirac’s, 641
 - Extended Binomial, 487
 - Fermat’s Last, 100–101, 299
 - Fermat’s Little, 239, 258
 - Four Color, 668–671, 676
 - Fundamental, Theorem of Arithmetic, 211, 257, 285–286
 - Handshaking, 599
 - Jordan Curve, 288
 - Kleene’s, 819–821, 839
 - Kuratowski’s, 663–665, 676
 - Lamé’s, 298
 - Master, 479
 - methods of proving, 76–83
 - Multinomial, 382
 - Ore’s, 641, 647
 - Pick’s, 292
 - Prime Number, 213
 - Proving, automated, 108
 - Wilson’s, 244
- Theory, Ramsey, 352
- Thesis, Church-Turing, 833
- Thickness of a graph, 666
- $3x + 1$ conjecture, 101–102
- Threshold function, 783
- Threshold gate, 783
- Threshold value, 783
- Tic-tac-toe, 705
- Tiling of checkerboard, 97–102, 277
- Time complexity of algorithms, 193–196
- Top-down parsing, 791
- Topological sorting, 576–578, 582
- Topology for local area network
 - hybrid, 605
 - ring, 605
 - star, 605
- Torus, 666
- Total function, 149
- Total ordering, 567, 582
 - compatible, 576, 582
- Totally ordered set, 568, 582
- Tournament, 680
 - round-robin, 416, 593
- Tournament sort, 172, 708
- Tower of Hanoi, 452–454
- Trace of recursive algorithm, 312, 313
- Tractable problem, 197, 836
- Trail, 622
- Transfer mode, asynchronous, 144
- Transient state, 841
- Transition function, 798
 - extending, 805
- Transitive closure of relation, 544, 547–550, 582
- computing, 550–553

- Transitive relation, 524–525, 527, 581
 - representing, using digraphs, 542
- Transitivity law, A–2
- Translating
 - English sentences to logical expressions, 10–11, 42–43, 56–57
 - logical expressions to English sentences, 42–43
- Transposes of matrices, 251, 257
- “Traveler’s Dodecahedron,” 639, 640
- Traveling salesman problem, 641, 649, 653–655, 676
- Traversal of tree, 710–722, 743
 - inorder, 712, 714, 715, 718, 743
 - level-order, 745
 - postorder, 712, 715–716, 717, 718, 743–744
 - preorder, 712–713, 718, 743
- Tree(s), 683–741
 - applications of, 695–707
 - AVL, 748
 - binary, 302–304, 686, 687
 - extended, 302
 - full, 304–305
 - binary search, 695–698, 743
 - binomial, 745
 - caterpillar, 746
 - decision, 698–700, 743
 - definition of, 683
 - derivation, 790–792, 839
 - extended binary, 302
 - family, 683, 684
 - full binary, 303
 - full m -ary, 686, 690, 743
 - game, 704–707
 - graceful, 746
 - height-balanced, 748
 - labeled, 695
 - m -ary, 686, 743
 - complete, 694
 - height of, 692–693
 - mesh of, 748
 - as models, 688–690
 - properties of, 690–693
 - quad, 748
 - rooted, 302, 685–688
 - balanced, 691, 743
 - binomial, 745
 - B-tree of degree k , 745
 - decision trees, 698–700
 - definition of, 743
 - height of, 691, 743
 - level order of vertices of, 745
 - ordered, 687, 743, 745
 - S_k -tree, 745
 - rooted Fibonacci, 695
 - spanning, 724–734, 744
 - building
 - by breadth-first search, 729–731
 - by depth-first search, 726–729
 - definition of, 724
 - degree-constrained, 746
 - distance between, 737
 - in IP multicasting, 726
 - maximum, 742
 - minimum, 737–741, 744
 - rooted, 737
 - Tree diagrams, 343–344, 386
 - Tree edges, 728
 - Tree traversal, 710–722, 743
 - inorder, 712, 714, 715, 718, 743
 - postorder, 712, 715–716, 717, 718, 743–744
 - preorder, 712–713, 718, 743
 - Tree-connected network, 689–690
 - Tree-connected parallel processors, 689–690
 - Trial division, 214
 - Triangle inequality, 102
 - Triangle, Pascal’s, 366, 386
 - Triangulation, 288
 - Trichotomy Law, A–2
 - Triominoes, 100, 277, 283
 - right, 100, 283
 - straight, 100
 - Trivial proof, 78, 79, 105
 - True negative, 419
 - True positive, 419
 - Truth set, 119
 - Truth table, 3–6, 10, 104
 - for biconditional statement, 9
 - for conjunction of two propositions, 4
 - for disjunction of two propositions, 4
 - for exclusive or of two propositions, 6
 - for implication, 6
 - for logical equivalences, 24
 - for negation of proposition, 3
 - Truth value, 104
 - of implication, 6
 - of proposition, 3
 - T-shirts, 344
 - Tukey, John Wilder, 16
 - Turing, Alan Mathison, 27, 176, 197, 825, 826
 - Turing machines, 785, 825, 826, 827–832, 839
 - in computational complexity, 833
 - computing functions with, 831–832
 - definition of, 828–830
 - nondeterministic, 832
 - sets recognized by, 830–831
 - types of, 832
 - Twin prime conjecture, 215
 - Twin primes, 215
 - Two-dimensional array, 606–607
 - Two’s complement expansion, 230–231
 - Type, 113
 - Type 0 grammar, 789, 838
 - Type 1 grammar, 789, 838–839
 - Type 2 grammar, 789, 839
 - Type 3 grammar, 789, 817, 821–824, 839
 - Ulam, Stanislaw, 483
 - Ulam numbers, 165
 - Ulam’s problem, 101–102, 483
 - Unary representations, 831
 - Uncountable set, 160
 - Undefined function, 149
 - Underlying undirected graph, 601, 675
 - Undirected edges, 592, 675
 - of simple graph, 590
 - Undirected graphs, 592, 599, 675
 - connectedness in, 624–626
 - Euler circuit of, 634
 - Euler path of, 638
 - orientation of, 679
 - paths in, 622, 676
 - underlying, 601, 675
 - Unicasting, 726
 - Unicorns, 17
 - Unicycle, 750
 - Uniform distribution, 402, 442
 - Union
 - of fuzzy sets, 133
 - of graphs, 661, 767
 - of multisets, 132
 - of sets, 121, 126–128, 163
 - of three finite sets, number of elements in, 501–503, 514
 - of two finite sets, number of elements in, 500, 514
 - Uniqueness proofs, 92–93, 105
 - Uniqueness quantifier, 37
 - Unit (Egyptian) fraction, 331
 - Unit property, in Boolean algebra, 753
 - Unit-delay machine, 799–800
 - United States Coast Survey, 32
 - UNIVAC, 811
 - Universal address system, 711
 - Universal generalization, 70, 71
 - Universal instantiation, 70, 71
 - Universal quantification, 34, 105
 - negation of, 39–41
 - Universal quantifier, 34–36
 - Universal set, 113, 163
 - Universal transitivity, 74
 - Universe of discourse, 34, 105
 - Unlabeled
 - boxes, 376
 - objects, 376
 - Unless, 6
 - Unsolvable problem, 197
 - Upper bound, A–2
 - of lattice, 586
 - of poset, 574, 582
 - Upper limit of summation, 153
 - Upper triangular matrix, 260
 - U. S. Coast Survey, 32
 - Utilities-and-houses problem, 657–659
 - Vacuous proof, 78, 105
 - Valeé-Poussin, Charles-Jean-Gustave-Nicholas de la, 213
 - Valid
 - argument, 64
 - argument form, 64
 - Value(s)
 - expected, 402–403, 426–429, 442
 - in hatcheck problem, 403
 - of inversions in permutation, 430–431
 - linearity of, 429–431, 442
 - final, A–13
 - initial, A–13
 - threshold, 783
 - of tree, 706
 - of vertex in game tree, 706–707, 743
 - Vandermonde, Alexandre-Théophile, 368
 - Vandermonde’s identity, 367–368
 - Variable(s)
 - binding, 38–39
 - Boolean, 15, 104, 750, 758, 781
 - free, 38, 105
 - random, 402, 408–409
 - covariance of, 442
 - definition of, 442
 - distribution of, 408, 442
 - geometric, 433–434
 - expected values of, 402–403, 426–429, 442
 - independent, 434–436, 442
 - indicator, 440
 - standard deviation of, 436
 - variance of, 426–429, 442
 - Variance, 426, 436–438, 442
 - Veitch, E. W., 768
 - Vending machine, 797–798
 - Venn diagrams, 113, 115, 122, 123, 163
 - Venn, John, 115
 - Verb, 786
 - Verb phrase, 786
 - Vertex, 288
 - bipartition of, 602
 - Vertex basis, 632
 - Vertex (vertices), 541
 - adjacent, 598, 600, 675
 - ancestor of, 686, 743
 - child of, 686, 687, 743
 - connecting, 598
 - counting paths between, 628–629
 - cut, 625
 - degree of, in undirected graph, 598
 - descendant of, 686, 743
 - of directed graph, 591, 600
 - of directed multigraph, 592
 - distance between, 680
 - eccentricity of, 695
 - end, 600
 - in-degree of, 600, 675
 - independent set of, 680
 - initial, 541, 600
 - interior, 551
 - internal, 686, 743
 - isolated, 598, 676
 - level of, 691, 743

- Vertex (vertices)—*Cont.*
 - level order of, 745
 - of multigraph, 590
 - number of, of full binary tree, 306
 - out-degree of, 600, 675
 - parent of, 686, 743
 - pendant, 598, 676
 - of polygon, 288
 - of pseudograph, 590
 - sibling of, 686, 743
 - of simple graph, 590, 612
 - terminal, 541, 600
 - of undirected graph, 592, 598, 600
 - degree of, 598
 - value of, in game tree, 706–707, 743
- Very large scale integration (VLSI) graphs, 682
- Vocabulary, 787, 838
- “Voyage Around the World” Puzzle, 639

- Walk, 622
 - closed, 622
- Warshall, Stephen, 551
- Warshall’s algorithm, 550–553
- WAVES, Navy, 811
- Weakly connected graphs, 626
- Web crawlers, 595
- Web graph, 594–595, 734
 - strongly connected components of, 627

- Web page searching, 13
- Web pages, 13, 595, 627, 734
- Web spiders, 734
- Website for this book, xviii, xxi
- Weighted graphs, 676
 - minimum spanning tree for, 738–741
 - shortest path between, 647–653
 - traveling salesman problem with, 653–655
- Well-formed expressions, 306
- Well-formed formula, 301
 - for compound propositions, 301
 - of operators and operands, 301
 - in prefix notation, 724
 - structural induction and, 305
- Well-founded induction, principle of, 585
- Well-founded poset, 581
- Well-ordered induction, 582
 - principle of, 568–569
- Well-ordered set, 332, 568, 582
- Well-ordering property, 290–291, 328, A–5
- WFF’N PROOF; The Game of Modern Logic* (Allen), 74
- Wheels, 601, 676
- Whitehead, Alfred North, 114
- Wiles, Andrew, 101, 239
- Wilson’s Theorem, 244

- Without loss of generality, 89
- Witnesses, 181
- WLOG (without loss of generality), 89
- Word, 787
- Word and Object* (Quine), 776
- World Cup soccer tournament, 362
- World Wide Web graph, 594–595
- World’s record, for twin primes, 215
- Worst-case complexity of algorithms, 194, 195–196, 197

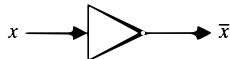

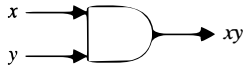
- XML, 796
- XOR, 15–16

- Yes-or-no problems, 834

- Zebra puzzle, 21
- Zero property, in Boolean algebra, 753
- Zero-one matrices, 252–254, 257
 - Boolean product of, 252–254
 - join of, 252
 - meet of, 252
 - representing relations using, 538–540
 - of transitive closure, 549–550
 - Warshall’s algorithm and, 550–553
- Ziegler’s Giant Bar, 184
- Zodiac, signs of, 388

LIST OF SYMBOLS

TOPIC	SYMBOL	MEANING	PAGE	
COUNTING	$P(n, r)$	number of r -permutations of a set with n elements	355	
	$C(n, r)$	number of r -combinations of a set with n elements	357	
	$\binom{n}{r}$	binomial coefficient n choose r	357	
	$C(n; n_1, n_2, \dots, n_m)$	multinomial coefficient	382	
	$p(E)$	probability of E	394	
	$p(E F)$	conditional probability of E given F	404	
	$E(X)$	expected value of the random variable X	426	
	$V(X)$	variance of the random variable X	436	
	C_n	Catalan number	456	
	$N(P_{i_1} \dots P_{i_n})$	number of elements having properties $P_{i_j}, j = 1, \dots, n$	506	
	$N(P'_{i_1} \dots P'_{i_n})$	number of elements not having properties $P_{i_j}, j = 1, \dots, n$	506	
	D_n	number of derangements of n objects	510	
RELATIONS	$S \circ R$	composite of the relations R and S	526	
	R^n	n th power of the relation R	526	
	R^{-1}	inverse relation	528	
	s_C	select operator for condition C	532	
	P_{i_1, i_2, \dots, i_m}	projection	533	
	$J_p(R, S)$	join	534	
	Δ	diagonal relation	545	
	R^*	connectivity relation of R	547	
	$[a]_R$	equivalence class of a with respect to R	558	
	$[a]_m$	congruence class modulo m	558	
	(S, R)	poset consisting of the set S and partial ordering R	566	
	$a < b$	a is less than b	567	
	$a > b$	a is greater than b	567	
	$a \leq b$	a is less than or equal to b	567	
	$a \geq b$	a is greater than or equal to b	567	
	GRAPHS AND TREES	(u, v)	directed edge	541
	$G = (V, E)$	graph with vertex set V and edge set E	589	
$\{u, v\}$	undirected edge	590		
$\deg(v)$	degree of the vertex v	598		
$\deg^-(v)$	in-degree of the vertex v	600		
$\deg^+(v)$	out-degree of the vertex v	600		
K_n	complete graph on n vertices	601		
C_n	cycle of size n	601		
W_n	wheel of size n	601		
Q_n	n -cube	602		
$K_{m,n}$	complete bipartite graph of size m, n	604		

TOPIC	SYMBOL	MEANING	PAGE
GRAPHS AND TREES (cont.)	$G_1 \cup G_2$	union of G_1 and G_2	608
	$a, x_1, \dots, x_{n-1}, b$	path from a to b	623
	$a, x_1, \dots, x_{n-1}, a$	circuit	623
	r	number of regions of the plane	660
	$\text{deg}(R)$	degree of the region R	661
	n	number of vertices of a rooted tree	690
	i	number of internal vertices of a rooted tree	691
	l	number of leaves of a rooted tree	691
	m	greatest number of children of an internal vertex in a rooted tree	686
	h	height of a rooted tree	692
BOOLEAN ALGEBRA	\bar{x}	complement of the Boolean variable x	749
	$x + y$	Boolean sum of x and y	749
	$x \cdot y$ (or xy)	Boolean product of x and y	749
	B	$\{0, 1\}$	750
	F^d	dual of F	754
	$x \downarrow y$	x NAND y	759
	$x \uparrow y$	x NOR y	759
		inverter	761
		OR gate	761
		AND gate	761
LANGUAGES	λ	the empty string	151
AND	xy	concatenation of x and y	300
FINITE-STATE MACHINES	$l(x)$	length of the string x	301
	w^R	reversal of w	309
	(V, T, S, P)	phrase-structure grammar	787
	S	start symbol	787
	$w \rightarrow w_1$	production	787
	$w_1 \Rightarrow w_2$	w_2 is directly derivable from w_1	787
	$w_1 \xRightarrow{*} w_2$	w_2 is derivable from w_1	787
	$\langle A \rangle ::= \langle B \rangle c \mid d$	Backus-Naur form	792
	(S, I, O, f, g, s_0)	finite-state machine with output	798
	s_0	start state	798
	AB	concatenation of the sets A and B	804
	A^*	Kleene closure of A	804
	(S, I, f, s_0, F)	finite-state machine with no output	805
	(S, I, f, s_0)	Turing machine	828