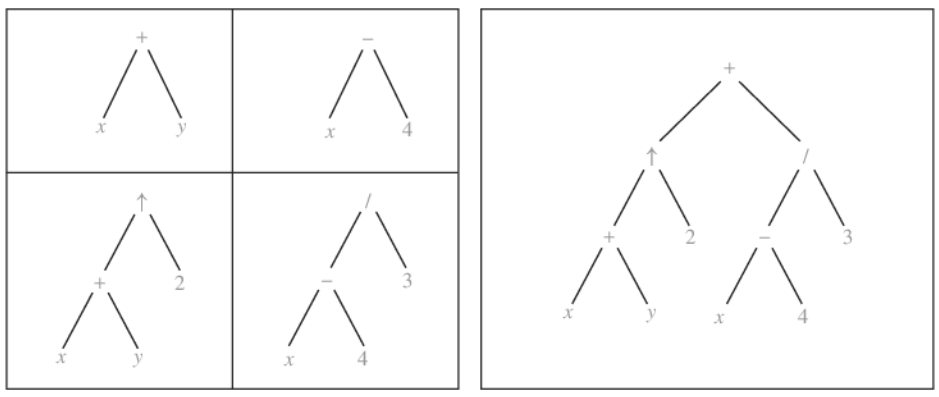


FIGURE 10 **A Binary Tree Representing $((x + y) \uparrow 2) + ((x - 4)/3)$.**

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 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 in preorder, 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

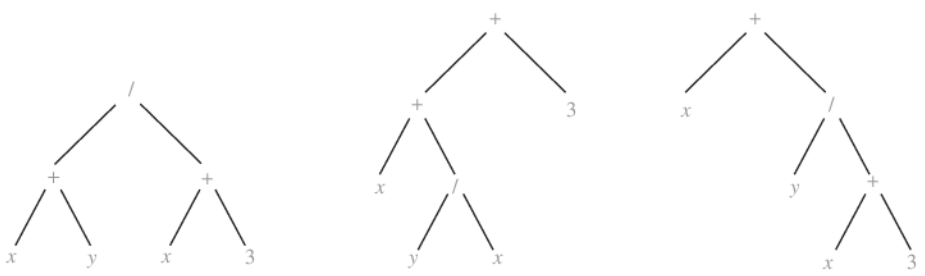


FIGURE 11 **Rooted Trees Representing $(x + y)/(x + 3)$, $(x + (y/x)) + 3$, and $x + (y/(x + 3))$.**

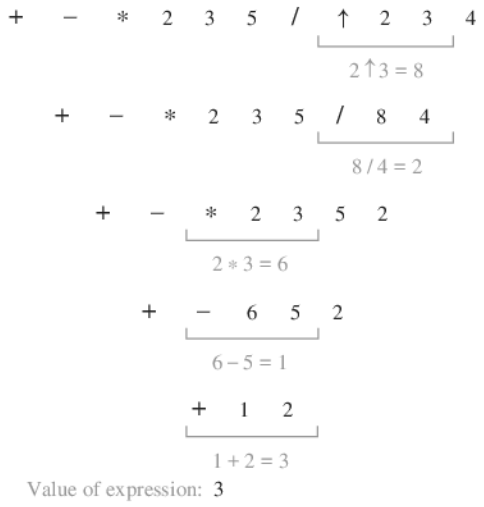


FIGURE 12 Evaluating a Prefix Expression.

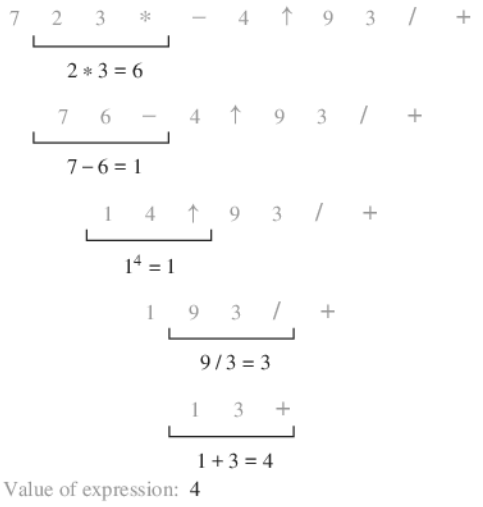


FIGURE 13 Evaluating a Postfix Expression.

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. ◀

Links

Reverse polish notation was first proposed in 1954 by Burks, Warren, and Wright.

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. Reverse polish notation was extensively used in electronic calculators in the 1970s and 1980s.

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.

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. ◀

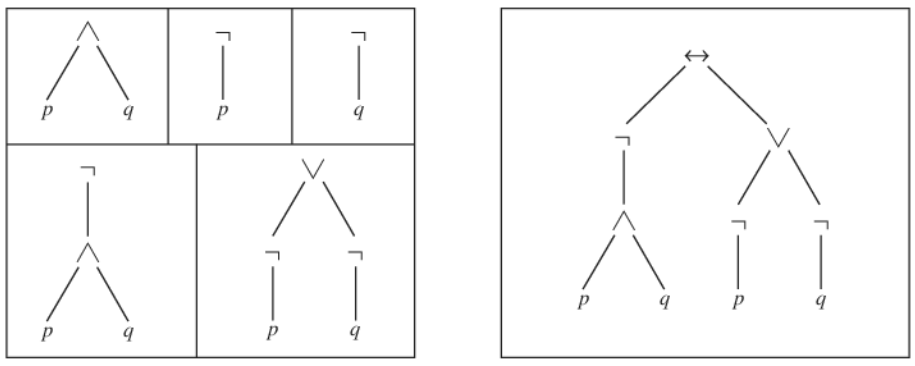


FIGURE 14 **Constructing the Rooted Tree for a Compound Proposition.**

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.

Extra Examples

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.

Links

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.

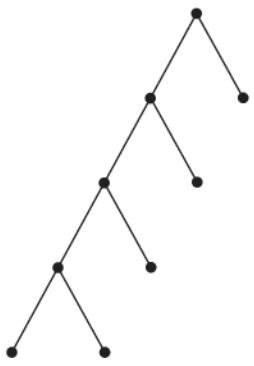
Łukasiewicz worked on mathematical logic throughout his career. His work on a three-valued logic was an important contribution to the subject. Nevertheless, he is best known in the mathematical and computer science communities for his introduction of parenthesis-free notation, now called Polish notation.

Exercises

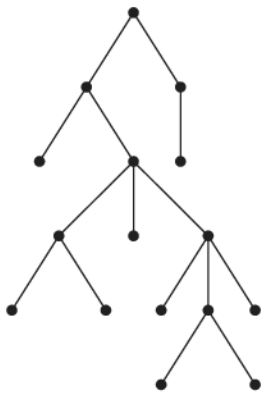
In Exercises 1–3 construct the universal address system for the given ordered rooted tree. Then use this to order its vertices using the lexicographic order of their labels.

1.

2.



3.



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.

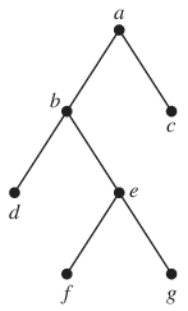
a) 1.1.1, 1.1.2, 1.2, 2.1.1.1, 2.1.2, 2.1.3, 2.2, 3.1.1, 3.1.2.1, 3.1.2.2, 3.2

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

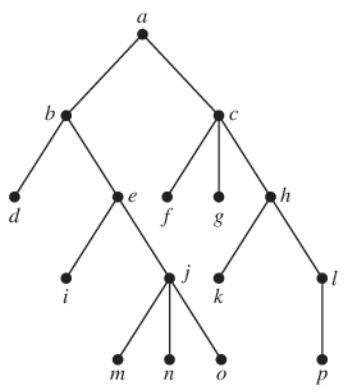
c) 1.1, 1.2.1, 1.2.2, 1.2.2.1, 1.3, 1.4, 2, 3.1, 3.2, 4.1.1.1

In Exercises 7–9 determine the order in which a preorder traversal visits the vertices of the given ordered rooted tree.

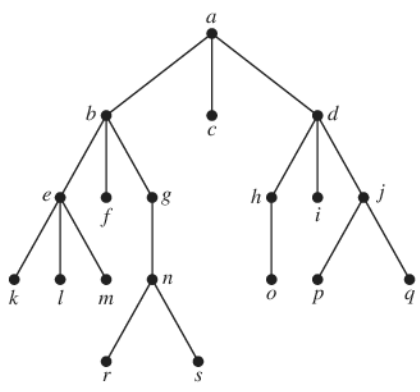
7.



8.



9.



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) $* + 3 + 3 \uparrow 3 + 3\ 3\ 3$

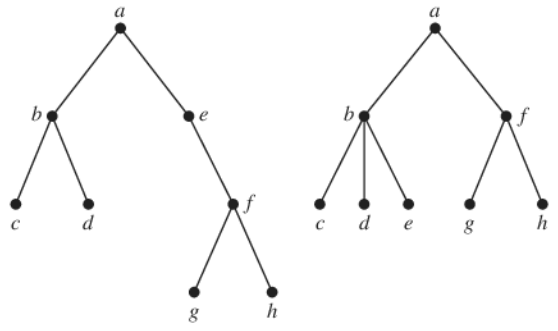
24. What is the value of each of these postfix expressions?
a) $5\ 2\ 1 - - 3\ 1\ 4 + + *$
b) $9\ 3 / 5 + 7\ 2 - *$
c) $3\ 2 * 2 \uparrow 5\ 3 - 8\ 4 / * -$

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.

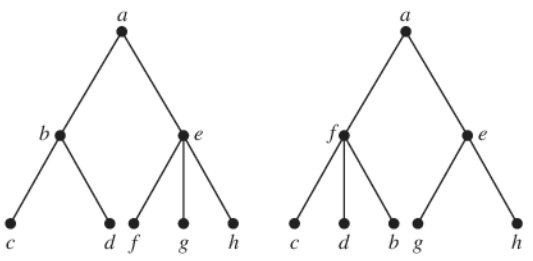
***26.** Show that an ordered rooted tree is uniquely determined when a list of vertices generated by a preorder traversal of the tree and the number of children of each vertex are specified.

***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.

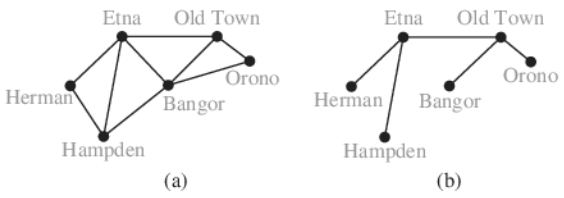


FIGURE 1 **(a) A Road System and (b) a Set of Roads to Plow.**

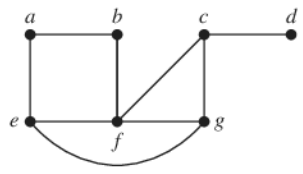


FIGURE 2 **The Simple Graph *G*.**

11.4 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.

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.

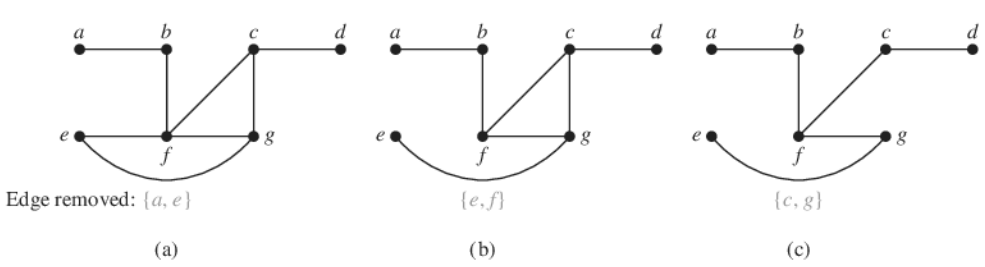


FIGURE 3 **Producing a Spanning Tree for *G* by Removing Edges That Form Simple Circuits.**

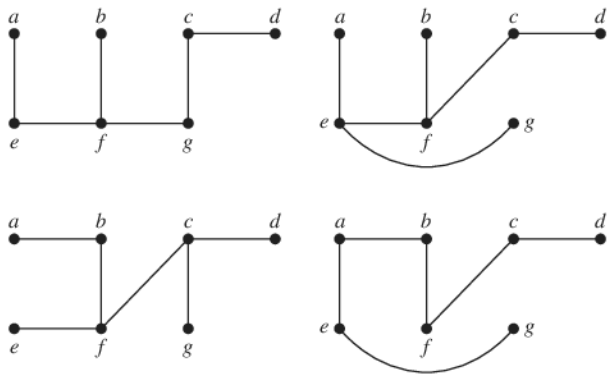


FIGURE 4 **Spanning Trees of G.**

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 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 17 in Section 6.1.)

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

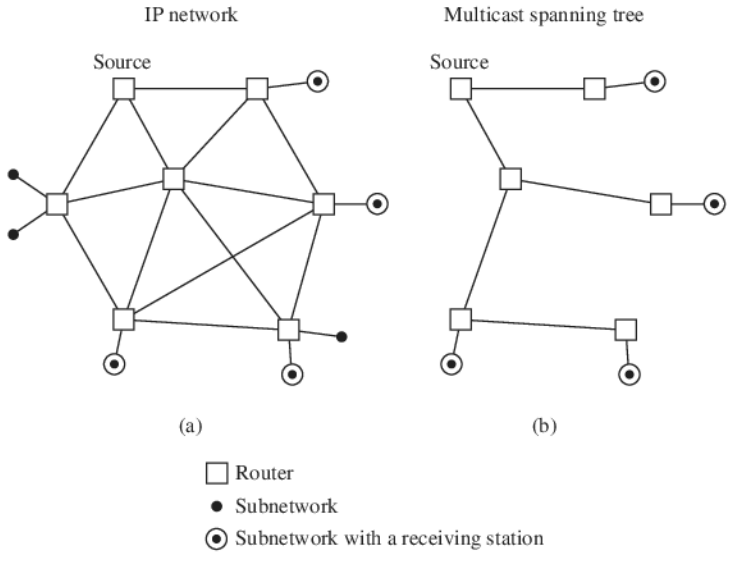


FIGURE 5 A Multicast Spanning Tree.

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

Links

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. 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.

Demo

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.

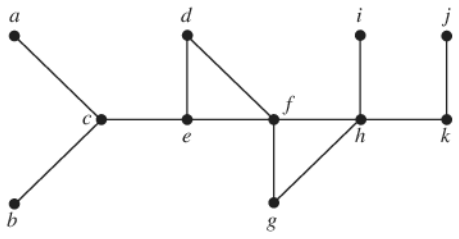


FIGURE 6 **The Graph G.**

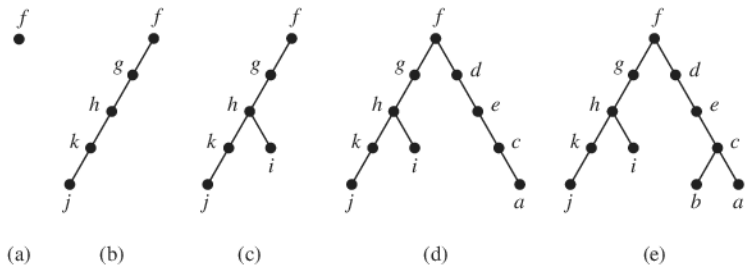


FIGURE 7 **Depth-First Search of G.**

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.

Extra Examples

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 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 43 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

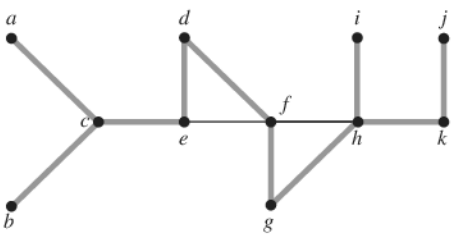


FIGURE 8 **The Tree Edges and Back Edges of the Depth-First Search in Example 4.**

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 $v_1, \ldots, v_n$ 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 ever added that connects two vertices 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.

ALGORITHM 1 Depth-First Search.

procedure *DFS*(G: connected graph with vertices $v_1, v_2, \ldots, v_n$)
$T :=$ 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
  add vertex w and edge $\{v, w\}$ to T
  visit(w)

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 10.3), 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 [GrYe05], [Ma89], and [CoLeRiSt09] 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

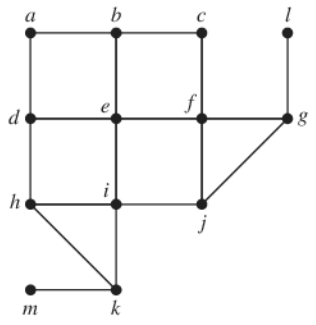


FIGURE 9 **A Graph *G*.**

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.

EXAMPLE 5 Use breadth-first search to find a spanning tree for the graph shown in Figure 9.

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 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 $v_1, v_2, \ldots, v_n$. 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.

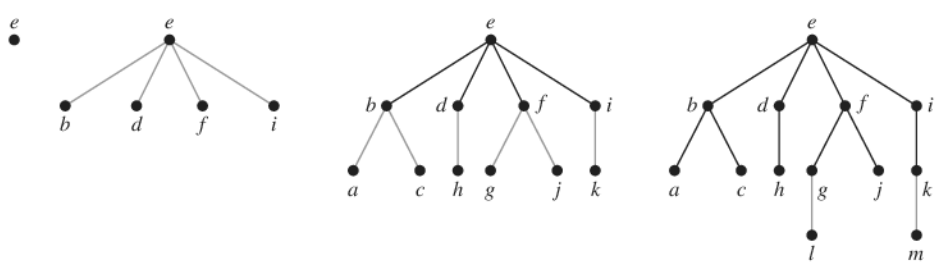


FIGURE 10 **Breadth-First Search of *G*.**

ALGORITHM 2 **Breadth-First Search.**

procedure *BFS* (G: connected graph with vertices $v_1, v_2, \ldots, v_n$)
$T :=$ 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
  remove the first vertex, v, from L
  for each neighbor w of v
    if w is not in L and not in T **then**
      add w to the end of the list L
      add w and edge $\{v, w\}$ to T

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.

Breadth-first search is one of the most useful algorithms in graph theory. In particular, it can serve as the basis for algorithms that solve a wide variety of problems. For example, algorithms that find the connected components of a graph, that determine whether a graph is bipartite, and that find the path with the fewest edges between two vertices in a graph can all be built using breadth-first search.

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

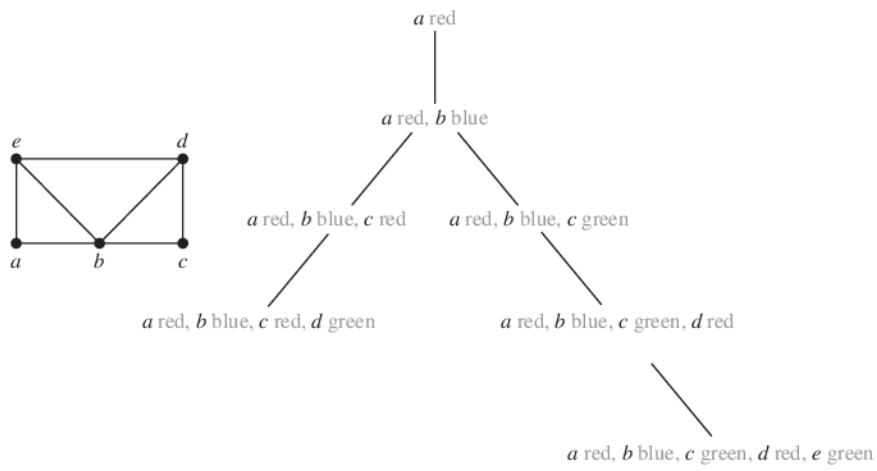


FIGURE 11 **Coloring a Graph Using Backtracking.**

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 Links **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 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 kth 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. ◀

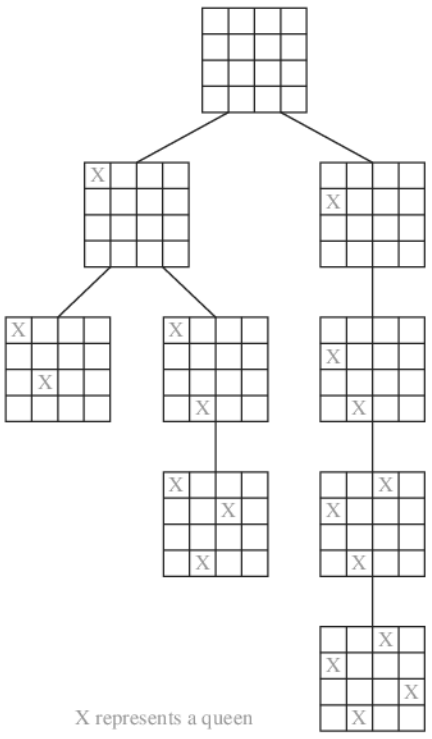


FIGURE 12 **A Backtracking Solution of the Four-Queens Problem.**

EXAMPLE 8 **Sums of Subsets** Consider this problem. Given a set of positive integers $x_1, x_2, \ldots, x_n$, 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. ◀

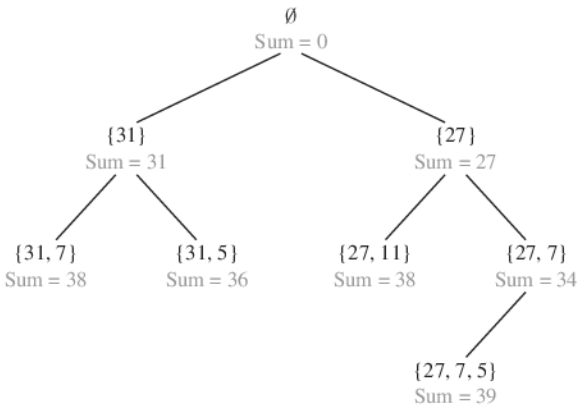


FIGURE 13 **Find a Sum Equal to 39 Using Backtracking.**

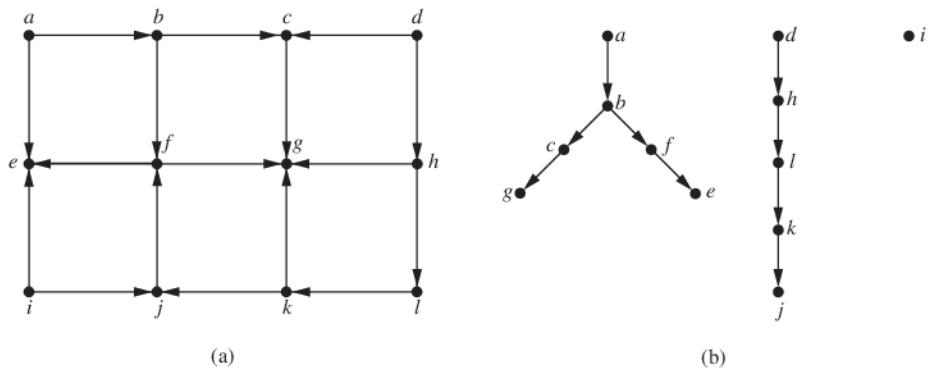


FIGURE 14 **Depth-First Search of a Directed Graph.**

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 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 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 [GrYe05], [Ma89], and [CoLeRiSt09]). 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 5 in Section 10.1, 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

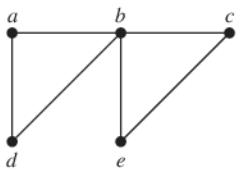
links at the previous level to look for new links, and so on. (Because of practical limitations, Web spiders have limits to the depth they search in depth-first search.) Using breadth-first search, an initial Web page is selected and a link on this page is followed to a second Web page, then a second 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

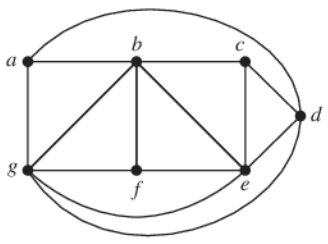
1. How many edges must be removed from a connected graph with n vertices and m edges to produce a spanning tree?

In Exercises 2–6 find a spanning tree for the graph shown by removing edges in simple circuits.

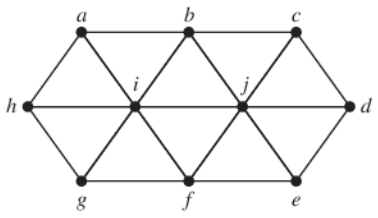
2.



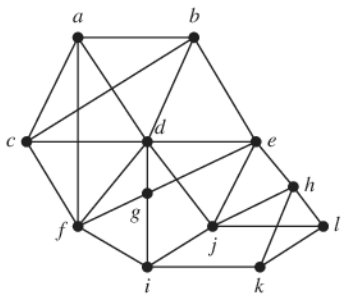
3.



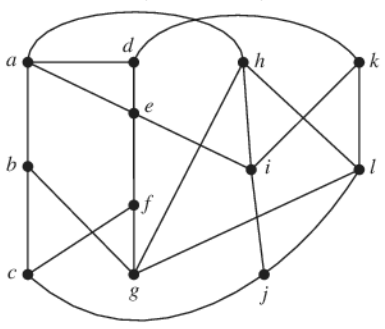
4.



5.



6.

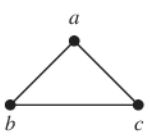


7. Find a spanning tree for each of these graphs.

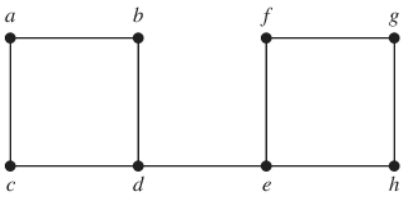
a) K_5 **b)** $K_{4,4}$ **c)** $K_{1,6}$
d) Q_3 **e)** C_5 **f)** W_5

In Exercises 8–10 draw all the spanning trees of the given simple graphs.

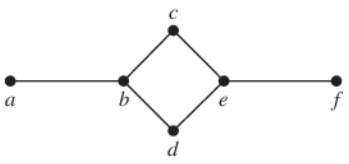
8.



9.



10.



***11.** How many different spanning trees does each of these simple graphs have?

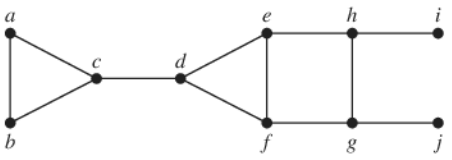
a) K_3 **b)** K_4 **c)** $K_{2,2}$ **d)** C_5

***12.** How many nonisomorphic spanning trees does each of these simple graphs have?

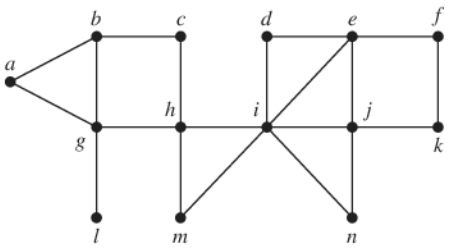
a) K_3 **b)** K_4 **c)** K_5

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.

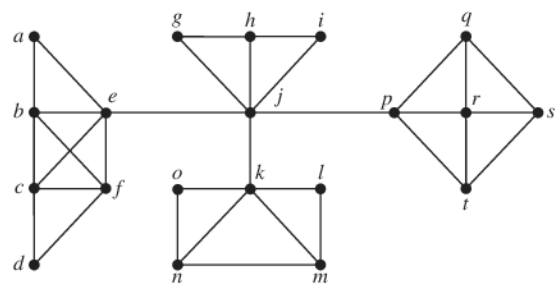
13.



14.



15.



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 10.2), 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.

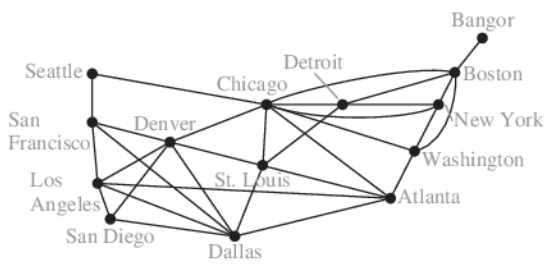
19. Describe the trees produced by breadth-first search and depth-first search of the wheel graph W_n, starting at the vertex of degree n, where n is an integer with $n \geq 3$. (See Example 7 of Section 10.2.) Justify your answers.

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. Describe the tree produced by breadth-first search and depth-first search for the n-cube graph Q_n, where n is a positive integer.

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. Explain how breadth-first search or depth-first search can be used to order the vertices of a connected graph.

***25.** 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.

26. Use backtracking to try to find a coloring of each of the graphs in Exercises 7–9 of Section 10.8 using three colors.

27. Use backtracking to solve the n-queens problem for these values of n.

a) $n = 3$ **b)** $n = 5$ **c)** $n = 6$

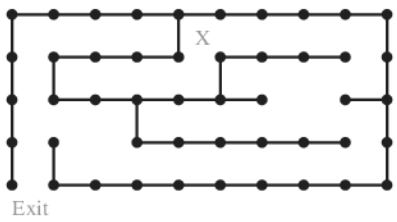
28. Use backtracking to find a subset, if it exists, of the set $\{27, 24, 19, 14, 11, 8\}$ with sum

a) 20. **b)** 41. **c)** 60.

29. Explain how backtracking can be used to find a Hamilton path or circuit in a graph.

30. 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 a 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.

31. Show that every finite simple graph has a spanning forest.

32. How many trees are in the spanning forest of a graph?

33. How many edges must be removed to produce the spanning forest of a graph with n vertices, m edges, and c connected components?

34. 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.

35. Explain how to use breadth-first search to find the length of a shortest path between two vertices in an undirected graph.

36. Devise an algorithm based on breadth-first search that determines whether a graph has a simple circuit, and if so, finds one.

37. Devise an algorithm based on breadth-first search for finding the connected components of a graph.

38. Explain how breadth-first search and how depth-first search can be used to determine whether a graph is bipartite.

39. Which connected simple graphs have exactly one spanning tree?

40. Devise an algorithm for constructing the spanning forest of a graph based on deleting edges that form simple circuits.

41. Devise an algorithm for constructing the spanning forest of a graph based on depth-first searching.

42. Devise an algorithm for constructing the spanning forest of a graph based on breadth-first searching.

43. 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.

44. When must an edge of a connected simple graph be in every spanning tree for this graph?

45. 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.

46. Use Exercise 43 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.

47. Use mathematical induction to prove that breadth-first search visits vertices in order of their level in the resulting spanning tree.

48. Use pseudocode to describe a variation of depth-first search that assigns the integer n to the nth 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.

49. Use pseudocode to describe a variation of breadth-first search that assigns the integer m to the mth vertex visited in the search.

***50.** Suppose that G is a directed graph and T is a spanning tree constructed using breadth-first search. Show that every edge of G has endpoints that are at the same level or one level higher or lower.

51. 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.

***52.** 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.

53. Find the distance between each pair of spanning trees shown in Figures 3(c) and 4 of the graph G shown in Figure 2.

***54.** 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.

****55.** 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.

***56.** 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.

57. For each of the directed graphs in Exercises 18–23 of Section 10.5 either find a rooted spanning tree of the graph or determine that no such tree exists.

***58.** 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.]

***59.** 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.

***60.** 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 51) relative to the spanning tree T.

***61.** Use Exercise 60 to construct an algorithm for determining whether a directed graph contains a circuit.

11.5 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

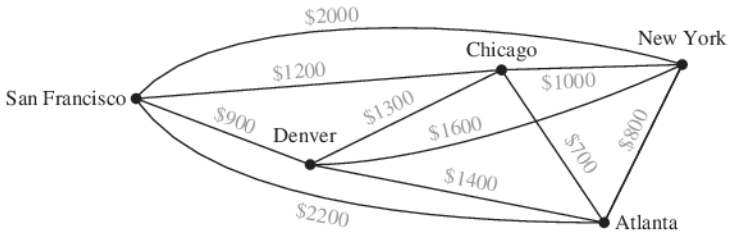


FIGURE 1 A Weighted Graph Showing Monthly Lease Costs for Lines in a Computer Network.

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**.

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.

Demo

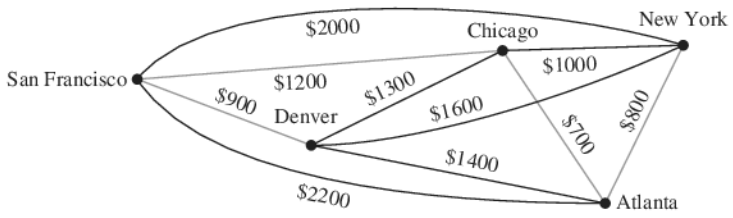
We will present two algorithms for constructing minimum spanning trees. Both proceed by successively adding edges of smallest weight from those edges with a specified property that have not already been used. Both are greedy algorithms. Recall from Section 3.1 that a greedy algorithm is a procedure that 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.

Links

The first algorithm that we will discuss was originally discovered by the Czech mathematician Vojtěch Jarník in 1930, who described it in a paper in an obscure Czech journal. The algorithm became well known when it was rediscovered in 1957 by Robert Prim. Because of this, it is known as **Prim's algorithm** (and sometimes as the **Prim-Jarník 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, never 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.

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.



Choice	Edge	Cost
1	{Chicago, Atlanta}	$ 700
2	{Atlanta, New York}	$ 800
3	{Chicago, San Francisco}	$ 1200
4	{San Francisco, Denver}	$ 900
	Total:	$ 3600

FIGURE 2 A Minimum Spanning Tree for the Weighted Graph in Figure 1.

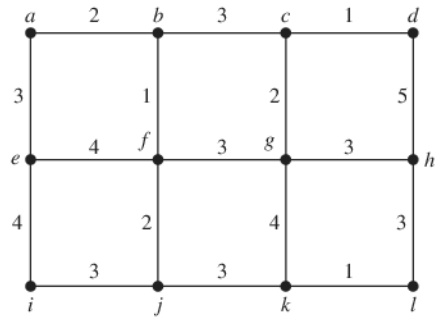


FIGURE 3 A Weighted Graph.

ALGORITHM 1 Prim's Algorithm.

procedure *Prim*(G: weighted connected undirected graph with n vertices)
T := a minimum-weight edge
for $i := 1$ **to** $n - 2$
  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
return T {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 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.

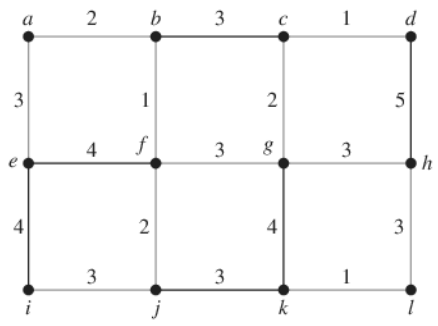
Solution: We solve this problem by finding a minimum spanning tree in the graph in Figure 1. Prim's algorithm is carried out by choosing an initial edge of minimum weight and successively adding edges of minimum weight that are incident to a vertex in the tree and that do not form simple circuits. The 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. ◀

Links

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.



(a)

Choice	Edge	Weight
1	$\{b, f\}$	1
2	$\{a, b\}$	2
3	$\{f, j\}$	2
4	$\{a, e\}$	3
5	$\{i, j\}$	3
6	$\{f, g\}$	3
7	$\{c, g\}$	2
8	$\{c, d\}$	1
9	$\{g, h\}$	3
10	$\{h, l\}$	3
11	$\{k, l\}$	1
	Total:	24

(b)

FIGURE 4 **A Minimum Spanning Tree Produced Using Prim's Algorithm.**

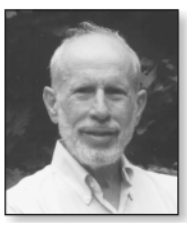
Successively add edges with minimum weight that do not form a simple circuit with those edges already chosen. Stop after $n - 1$ edges have been selected.

The proof that Kruskal's algorithm produces a minimum spanning tree for every connected weighted graph is left as an exercise. Pseudocode for Kruskal's algorithm is given in Algorithm 2.

ALGORITHM 2 **Kruskal's Algorithm.**

procedure *Kruskal*(G: weighted connected undirected graph with n vertices)
T := empty graph
for $i := 1$ **to** $n - 1$
  e := any edge in G with smallest weight that does not form a simple circuit when added to T
  $T := T$ with e added
return T {T is a minimum spanning tree of G}

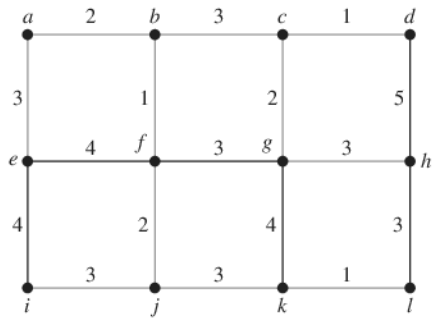
Links



JOSEPH BERNARD KRUSKAL (1928–2010) Joseph Kruskal was born in New York City, where his father was a fur dealer and his mother promoted the art of origami on early television. Kruskal 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, where he worked until his retirement in the late 1990s. 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. His research interests included statistical linguistics and psychometrics. Besides his work on minimum spanning trees, Kruskal is also known for contributions to multidimensional scaling. It is noteworthy that Joseph Kruskal's two brothers, Martin and William, also were well known mathematicians.

Links

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, and as mentioned in the text what is now called Prim's algorithm was discovered by Vojtěch Jarník in 1930.



(a)

Choice	Edge	Weight
1	$\{c, d\}$	1
2	$\{k, l\}$	1
3	$\{b, f\}$	1
4	$\{c, g\}$	2
5	$\{a, b\}$	2
6	$\{f, j\}$	2
7	$\{b, c\}$	3
8	$\{j, k\}$	3
9	$\{g, h\}$	3
10	$\{i, j\}$	3
11	$\{a, e\}$	3
	Total:	24

(b)

FIGURE 5 A Minimum Spanning Tree Produced by Kruskal's Algorithm.

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.

Extra Examples

Solution: A minimum spanning tree and the choices of edges at each stage of Kruskal's algorithm are shown in Figure 5. ◀

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 $e_1, e_2, \ldots, e_{n-1}$. Let S be the tree with $e_1, e_2, \ldots, e_{n-1}$ as its edges, and let S_k be the tree with $e_1, e_2, \ldots, e_k$ as its edges. Let T be a minimum spanning tree of G containing the edges $e_1, e_2, \ldots, e_k$, 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 $e_1, e_2, \ldots, e_k$, 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 $e_1, \ldots, e_k$, 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 $e_1, e_2, \ldots, e_k$.

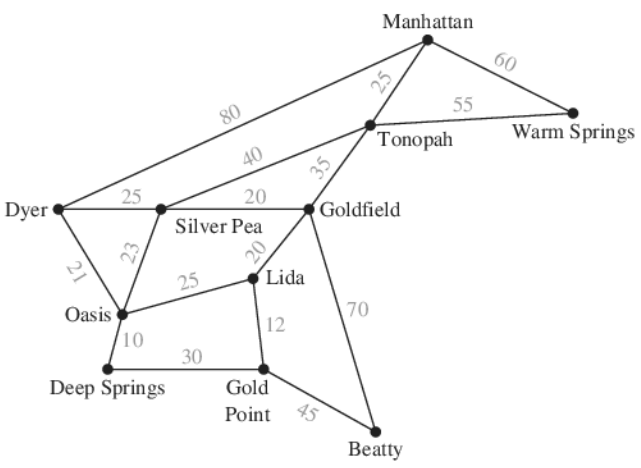
By deleting e from T and adding e_{k+1}, we obtain a tree T' with $n - 1$ edges (it is a tree because it has no simple circuits). Note that the tree T' contains $e_1, e_2, \ldots, e_k, e_{k+1}$. Furthermore, because e_{k+1} was chosen by Prim's algorithm at the $(k + 1)$st step, and e was also available 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 $e_1, \ldots, e_k$. Hence, $k = n - 1$, and $S = T$. It follows that Prim's algorithm produces a minimum spanning tree. ◁

It can be shown (see [CoLeRiSt09]) that to find a minimum spanning tree of a graph with m edges and n vertices, Kruskal's algorithm can be carried out using $O(m \log m)$ operations and Prim's algorithm can be carried out using $O(m \log n)$ operations. Consequently, it is preferable to use Kruskal's algorithm for graphs that are **sparse**, that is, where m is very small compared to $C(n, 2) = n(n-1)/2$, the total number of possible edges in an undirected graph with n vertices. Otherwise, there is little difference in the complexity of these two algorithms.

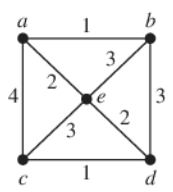
Exercises

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.)

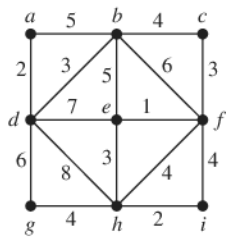


In Exercises 2–4 use Prim's algorithm to find a minimum spanning tree for the given weighted graph.

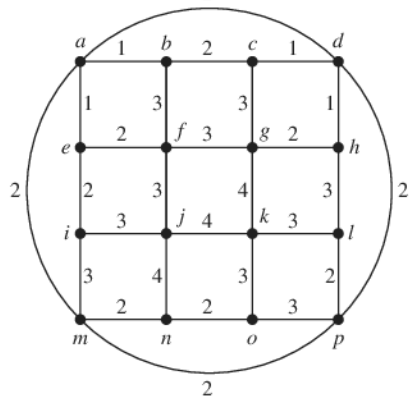
2.



3.



4.



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 when the input is an undirected graph with n vertices.

***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 when given as input an undirected graph with n vertices, 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.

33. Show that if G is a weighted graph with distinct edge weights, then for every simple circuit of G, the edge of maximum weight in this circuit does not belong to any minimum spanning tree of G.

When Kruskal invented the algorithm that finds minimum spanning trees by adding edges in order of increasing weight as long as they do not form a simple circuit, he also invented another algorithm sometimes called the **reverse-delete algorithm**. This algorithm proceeds by successively deleting edges of maximum weight from a connected graph as long as doing so does not disconnect the graph.

34. Express the reverse-delete algorithm in pseudocode.

35. Prove that the reverse-delete algorithm always produces a minimum spanning tree when given as input a weighted graph with distinct edge weights. [*Hint:* Use Exercise 33.]

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 every 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 of a problem
game tree: a rooted tree where vertices represents the possible positions of a game as it progresses and edges represent legal moves between these positions
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

RESULTS

A graph is a tree if and only if there is a unique simple path between every pair 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 11.1)
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 so 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.
c) Form a binary search tree for the words *vireo, warbler, egret, grosbeak, nuthatch*, and *kingfisher*.

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 to sort a list of n elements 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 part (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 $d_1, d_2, \ldots, d_n$ 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 $d_1, d_2, \ldots, d_n$.

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, $i = 0, 1, 2, \ldots$, 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 root 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-1}-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 $T_0, T_1, \ldots, T_{k-1}$, with roots $r_0, r_1, \ldots, r_{k-1}$, respectively, where v is not in any of these trees, where T_i is an S_i-tree for $i = 0, 1, \ldots, k-1$, by connecting v to r_0 and r_i to r_{i+1} for $i = 0, 1, \ldots, k-2$.

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 11.3 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.

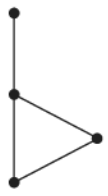
A **cut set** of a graph is a set of edges such that the removal of these edges produces a subgraph with more connected components than in the original graph, but no proper subset of this set of edges has this property.

26. Show that a cut set of a graph must have at least one edge in common with any spanning tree of this graph.

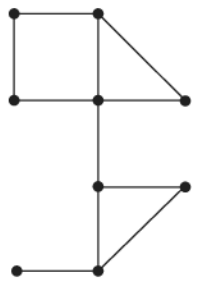
A **cactus** is a connected graph in which no edge is in more than one simple circuit not passing through any vertex other than its initial vertex more than once or its initial vertex other than at its terminal vertex (where two circuits that contain the same edges are not considered different).

27. Which of these graphs are cacti?

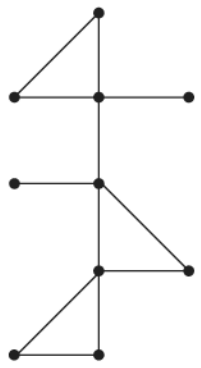
a)



b)



c)



28. Is a tree necessarily a cactus?

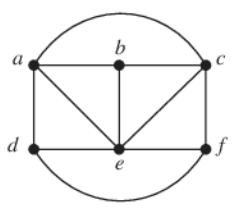
29. Show that a cactus is formed if we add a circuit containing new edges beginning and ending at a vertex of a tree.

***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.

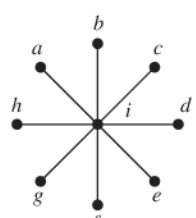
A **degree-constrained spanning tree** of a simple graph G is a spanning tree with the property that the degree of a vertex in this tree cannot exceed some specified bound. Degree-constrained spanning trees are useful in models of transportation systems where the number of roads at an intersection is 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.

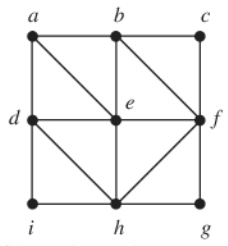
31.



32.



33.

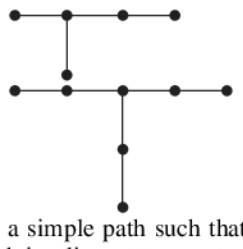


34. Show that a degree-constrained spanning tree of a simple graph in which each vertex has degree not exceeding 2 consists of a single Hamilton path in the graph.

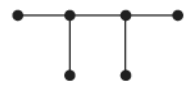
35. A tree with n vertices is called **graceful** if its vertices can be labeled with the integers $1, 2, \ldots, n$ such that the absolute values of the difference of the labels of adjacent vertices are all different. Show that these trees are graceful.

a)

b)



c)



d)

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.

39. 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.

a) 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?

b) 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?

40. 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.

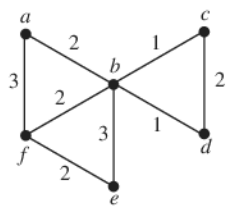
***41.** Suppose that e is an edge in a weighted graph that is incident to a vertex v such that the weight of e does not exceed the weight of any other edge incident to v. Show that there exists a minimum spanning tree containing this edge.

42. 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.

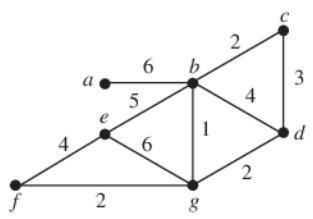
***43.** Show that if no two edges in a weighted graph have the same weight, then the edge with least weight incident to a vertex v is included in every minimum spanning tree.

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.

a)



b)



Let $G = (V, E)$ be a directed graph and let r be a vertex in G. An **arborescence** of G rooted at r is a subgraph $T = (V, F)$ of G such that the underlying undirected graph of T is a spanning tree of the underlying undirected graph of G and for every vertex $v \in V$ there is a path from r to v in T (with directions taken into account).

45. Show that a subgraph $T = (V, F)$ of the graph $G = (V, E)$ is an arborescence of G rooted at r if and only if T contains r, T has no simple circuits, and for every vertex $v \in V$ other than r, $\deg^-(v) = 1$ in T.

46. Show that a directed graph $G = (V, E)$ has an arborescence rooted at the vertex r if and only if for every vertex $v \in V$, there is a directed path from r to v.

47. In this exercise we will develop an algorithm to find the strong components of a directed graph $G = (V, E)$. Recall that a vertex $w \in V$ is **reachable** from a vertex $v \in V$ if there is a directed path from v to w.

a) Explain how to use breadth-first search in the directed graph G to find all the vertices reachable from a vertex $v \in G$.

b) Explain how to use breadth-first search in G^{conv} to find all the vertices from which a vertex $v \in G$ is reachable. (Recall that G^{conv} is the directed graph obtained from G by reversing the direction of all its edges.)

c) Explain how to use parts (a) and (b) to construct an algorithm that finds the strong components of a directed graph G, and explain why your algorithm is correct.

Computer Projects

Write programs with these input and output.

1. Given the adjacency matrix of an undirected simple graph, determine whether the graph is a tree.
2. 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.
3. 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.
4. Given a list of items, construct a binary search tree containing these items.
5. Given a binary search tree and an item, locate or add this item to the binary search tree.
6. Given the ordered list of edges of an ordered rooted tree, find the universal addresses of its vertices.
7. Given the ordered list of edges of an ordered rooted tree, list its vertices in preorder, inorder, and postorder.
8. Given an arithmetic expression in prefix form, find its value.
9. Given an arithmetic expression in postfix form, find its value.
10. Given the frequency of symbols, use Huffman coding to find an optimal code for these symbols.
11. Given an initial position in the game of nim, determine an optimal strategy for the first player.
12. Given the adjacency matrix of a connected undirected simple graph, find a spanning tree for this graph using depth-first search.
13. Given the adjacency matrix of a connected undirected simple graph, find a spanning tree for this graph using breadth-first search.
14. 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.
17. 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.
18. 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, and 10,000 from the set of positive integers less than 1,000,000, 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. Explain how trees are used to represent ancestral relations in the study of evolution.
3. Discuss hierarchical cluster trees and how they are used.
4. Define *AVL-trees* (sometimes also known as *height-balanced trees*). Describe how and why AVL-trees are used in a variety of different algorithms.

5. 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.
6. Define a *heap* and explain how trees can be turned into heaps. Why are heaps useful in sorting?
7. Describe dynamic algorithms for data compression based on letter frequencies as they change as characters are successively read, such as adaptive Huffman coding.
8. Explain how *alpha-beta pruning* can be used to simplify the computation of the value of a game tree.
9. Describe the techniques used by chess-playing programs such as Deep Blue.
10. 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.
11. Discuss the algorithms used in IP multicasting to avoid loops between routers.
12. Describe an algorithm based on depth-first search for finding the articulation points of a graph.
13. Describe an algorithm based on depth-first search to find the strongly connected components of a directed graph.
14. Describe the search techniques used by the crawlers and spiders in different search engines on the Web.
15. 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.
16. Compare and contrast some of the most important sorting algorithms in terms of their complexity and when they are used.
17. Discuss the history and origins of algorithms for constructing minimum spanning trees.
18. Describe algorithms for producing random trees.

CHAPTER

12 Boolean Algebra

The circuits in computers and other electronic devices have inputs, each of which is either a 0 or a 1, and produce outputs that are also 0s and 1s. Circuits can be constructed using any basic element that has two different states. Such elements include switches that can be in either the on or the off position and optical devices that can be either lit or unlit. In 1938 Claude Shannon showed how the basic rules of logic, first given by George Boole in 1854 in his *The Laws of Thought*, could be used to design circuits. These rules form the basis for Boolean algebra. In this chapter we develop the basic properties of Boolean algebra. The operation of a circuit is defined by a Boolean function that specifies the value of an output for each set of inputs. The first step in constructing a circuit is to represent its Boolean function by an expression built up using the basic operations of Boolean algebra. We will provide an algorithm for producing such expressions. The expression that we obtain may contain many more operations than are necessary to represent the function. Later in the chapter we will describe methods for finding an expression with the minimum number of sums and products that represents a Boolean function. The procedures that we will develop, Karnaugh maps and the Quine–McCluskey method, are important in the design of efficient circuits.

12.1 Boolean Functions

Introduction

Boolean algebra provides the operations and the rules for working with the set $\{0, 1\}$. Electronic and optical switches can be studied using this set and the rules of Boolean algebra. The three operations in Boolean algebra that we will use most are complementation, the Boolean sum, and the Boolean product. The **complement** of an element, denoted with a bar, is defined by $\overline{0} = 1$ and $\overline{1} = 0$. The Boolean sum, denoted by $+$ or by *OR*, has the following values:

$$1 + 1 = 1, \qquad 1 + 0 = 1, \qquad 0 + 1 = 1, \qquad 0 + 0 = 0.$$

The Boolean product, denoted by $\cdot$ or by *AND*, has the following values:

$$1 \cdot 1 = 1, \qquad 1 \cdot 0 = 0, \qquad 0 \cdot 1 = 0, \qquad 0 \cdot 0 = 0.$$

When there is no danger of confusion, the symbol $\cdot$ can be deleted, just as in writing algebraic products. Unless parentheses are used, the rules of precedence for Boolean operators are: first, all complements are computed, followed by all Boolean products, followed by all Boolean sums. This is illustrated in Example 1.

EXAMPLE 1 Find the value of $1 \cdot 0 + \overline{(0 + 1)}$.

Solution: Using the definitions of complementation, the Boolean sum, and the Boolean product, it follows that

$$\begin{aligned} 1 \cdot 0 + \overline{(0 + 1)} &= 0 + \overline{1} \\ &= 0 + 0 \\ &= 0. \end{aligned}$$

The complement, Boolean sum, and Boolean product correspond to the logical operators, $\neg$, $\vee$, and $\wedge$, respectively, where 0 corresponds to **F** (false) and 1 corresponds to **T** (true). Equalities in Boolean algebra can be directly translated into equivalences of compound propositions. Conversely, equivalences of compound propositions can be translated into equalities in Boolean algebra. We will see later in this section why these translations yield valid logical equivalences and identities in Boolean algebra. Example 2 illustrates the translation from Boolean algebra to propositional logic.

EXAMPLE 2 Translate $1 \cdot 0 + \overline{(0 + 1)} = 0$, the equality found in Example 1, into a logical equivalence.

Solution: We obtain a logical equivalence when we translate each 1 into a **T**, each 0 into an **F**, each Boolean sum into a disjunction, each Boolean product into a conjunction, and each complementation into a negation. We obtain

$$(\mathbf{T} \wedge \mathbf{F}) \vee \neg(\mathbf{T} \vee \mathbf{F}) \equiv \mathbf{F}.$$

◀

Example 3 illustrates the translation from propositional logic to Boolean algebra.

EXAMPLE 3 Translate the logical equivalence $(\mathbf{T} \wedge \mathbf{T}) \vee \neg\mathbf{F} \equiv \mathbf{T}$ into an identity in Boolean algebra.

Solution: We obtain an identity in Boolean algebra when we translate each **T** into a 1, each **F** into a 0, each disjunction into a Boolean sum, each conjunction into a Boolean product, and each negation into a complementation. We obtain

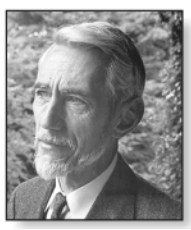
$$(1 \cdot 1) + \overline{0} = 1.$$

◀

Boolean Expressions and Boolean Functions

Let $B = \{0, 1\}$. Then $B^n = \{(x_1, x_2, \ldots, x_n) \mid x_i \in B \text{ for } 1 \leq i \leq n\}$ is the set of all possible n-tuples of 0s and 1s. The variable x is called a **Boolean variable** if it assumes values only from B, that is, if its only possible values are 0 and 1. A function from B^n to B is called a **Boolean function of degree *n***.

Links



CLAUDE ELWOOD SHANNON (1916–2001) Claude Shannon was born in Petoskey, Michigan, and grew up in Gaylord, Michigan. His father was a businessman and a probate judge, and his mother was a language teacher and a high school principal. Shannon attended the University of Michigan, graduating in 1936. He continued his studies at M.I.T., where he took the job of maintaining the differential analyzer, a mechanical computing device consisting of shafts and gears built by his professor, Vannevar Bush. Shannon's master's thesis, written in 1936, studied the logical aspects of the differential analyzer. This master's thesis presents the first application of Boolean algebra to the design of switching circuits; it is perhaps the most famous master's thesis of the twentieth century. He received his Ph.D. from M.I.T. in 1940. Shannon joined Bell Laboratories in 1940, where he worked on transmitting data efficiently. He was one of the first people to use bits to represent information. At Bell Laboratories he worked on determining the amount of traffic that telephone lines can carry. Shannon made many fundamental contributions to information theory. In the early 1950s he was one of the founders of the study of artificial intelligence. He joined the M.I.T. faculty in 1956, where he continued his study of information theory.

Shannon had an unconventional side. He is credited with inventing the rocket-powered Frisbee. He is also famous for riding a unicycle down the hallways of Bell Laboratories while juggling four balls. Shannon retired when he was 50 years old, publishing papers sporadically over the following 10 years. In his later years he concentrated on some pet projects, such as building a motorized pogo stick. One interesting quote from Shannon, published in *Omni Magazine* in 1987, is "I visualize a time when we will be to robots what dogs are to humans. And I am rooting for the machines."

EXAMPLE 4 The function $F(x, y) = x\overline{y}$ from the set of ordered pairs of Boolean variables to the set $\{0, 1\}$ is a Boolean function of degree 2 with $F(1, 1) = 0$, $F(1, 0) = 1$, $F(0, 1) = 0$, and $F(0, 0) = 0$. We display these values of F in Table 1. ◀

TABLE 1

x	y	$F(x, y)$
1	1	0
1	0	1
0	1	0
0	0	0

Boolean functions can be represented using expressions made up from variables and Boolean operations. The **Boolean expressions** in the variables $x_1, x_2, \ldots, x_n$ are defined recursively as

$0, 1, x_1, x_2, \ldots, x_n$ are Boolean expressions;
if E_1 and E_2 are Boolean expressions, then $\overline{E}_1$, (E_1E_2), and $(E_1 + E_2)$ are Boolean expressions.

Each Boolean expression represents a Boolean function. The values of this function are obtained by substituting 0 and 1 for the variables in the expression. In Section 12.2 we will show that every Boolean function can be represented by a Boolean expression.

EXAMPLE 5 Find the values of the Boolean function represented by $F(x, y, z) = xy + \overline{z}$.

Solution: The values of this function are displayed in Table 2. ◀

TABLE 2

x	y	z	xy	$\overline{z}$	$F(x, y, z) = xy + \overline{z}$
1	1	1	1	0	1
1	1	0	1	1	1
1	0	1	0	0	0
1	0	0	0	1	1
0	1	1	0	0	0
0	1	0	0	1	1
0	0	1	0	0	0
0	0	0	0	1	1

Note that we can represent a Boolean function graphically by distinguishing the vertices of the n-cube that correspond to the n-tuples of bits where the function has value 1.

EXAMPLE 6 The function $F(x, y, z) = xy + \overline{z}$ from B^3 to B from Example 5 can be represented by distinguishing the vertices that correspond to the five 3-tuples $(1, 1, 1)$, $(1, 1, 0)$, $(1, 0, 0)$, $(0, 1, 0)$, and $(0, 0, 0)$, where $F(x, y, z) = 1$, as shown in Figure 1. These vertices are displayed using solid black circles. ◀

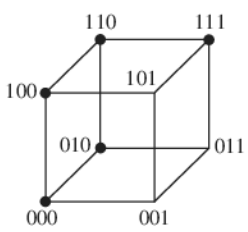


FIGURE 1

Boolean functions F and G of n variables are equal if and only if $F(b_1, b_2, \ldots, b_n) = G(b_1, b_2, \ldots, b_n)$ whenever $b_1, b_2, \ldots, b_n$ belong to B. Two different Boolean expressions that represent the same function are called **equivalent**. For instance, the Boolean expressions xy, $xy + 0$, and $xy \cdot 1$ are equivalent. The **complement** of the Boolean function F is the function $\overline{F}$, where $\overline{F}(x_1, \ldots, x_n) = \overline{F(x_1, \ldots, x_n)}$. Let F and G be Boolean functions of degree n. The **Boolean sum** $F + G$ and the **Boolean product** FG are defined by

$$(F + G)(x_1, \ldots, x_n) = F(x_1, \ldots, x_n) + G(x_1, \ldots, x_n),$$
$$(FG)(x_1, \ldots, x_n) = F(x_1, \ldots, x_n)G(x_1, \ldots, x_n).$$

A Boolean function of degree two is a function from a set with four elements, namely, pairs of elements from $B = \{0, 1\}$, to B, a set with two elements. Hence, there are 16 different Boolean functions of degree two. In Table 3 we display the values of the 16 different Boolean functions of degree two, labeled $F_1, F_2, \ldots, F_{16}$.

TABLE 3 The 16 Boolean Functions of Degree Two.

x	y	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}	F_{16}
1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0
1	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0
0	1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0
0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0

EXAMPLE 7 How many different Boolean functions of degree n are there?

Solution: From the product rule for counting, it follows that there are 2^n different n-tuples of 0s and 1s. Because a Boolean function is an assignment of 0 or 1 to each of these 2^n different n-tuples, the product rule shows that there are 2^{2^n} different Boolean functions of degree n. ◀

Table 4 displays the number of different Boolean functions of degrees one through six. The number of such functions grows extremely rapidly.

TABLE 4 The Number of Boolean Functions of Degree n.

Degree	*Number*
1	4
2	16
3	256
4	65,536
5	4,294,967,296
6	18,446,744,073,709,551,616

Identities of Boolean Algebra

There are many identities in Boolean algebra. The most important of these are displayed in Table 5. These identities are particularly useful in simplifying the design of circuits. Each of the identities in Table 5 can be proved using a table. We will prove one of the distributive laws in this way in Example 8. The proofs of the remaining properties are left as exercises for the reader.

EXAMPLE 8 Show that the distributive law $x(y + z) = xy + xz$ is valid.

Solution: The verification of this identity is shown in Table 6. The identity holds because the last two columns of the table agree. ◀

The reader should compare the Boolean identities in Table 5 to the logical equivalences in Table 6 of Section 1.3 and the set identities in Table 1 in Section 2.2. All are special cases of the same set of identities in a more abstract structure. Each collection of identities can be obtained by making the appropriate translations. For example, we can transform each of the identities in Table 5 into a logical equivalence by changing each Boolean variable into a propositional variable, each 0 into a **F**, each 1 into a **T**, each Boolean sum into a disjunction, each Boolean product into a conjunction, and each complementation into a negation, as we illustrate in Example 9.

Compare these Boolean identities with the logical equivalences in Section 1.3 and the set identities in Section 2.2.

TABLE 5 Boolean Identities.

Identity	*Name*
$\overline{\overline{x}} = x$	Law of the double complement
$x + x = x$ $x \cdot x = x$	Idempotent laws
$x + 0 = x$ $x \cdot 1 = x$	Identity laws
$x + 1 = 1$ $x \cdot 0 = 0$	Domination laws
$x + y = y + x$ $xy = yx$	Commutative laws
$x + (y + z) = (x + y) + z$ $x(yz) = (xy)z$	Associative laws
$x + yz = (x + y)(x + z)$ $x(y + z) = xy + xz$	Distributive laws
$\overline{(xy)} = \overline{x} + \overline{y}$ $\overline{(x + y)} = \overline{x}\,\overline{y}$	De Morgan's laws
$x + xy = x$ $x(x + y) = x$	Absorption laws
$x + \overline{x} = 1$	Unit property
$x\overline{x} = 0$	Zero property

EXAMPLE 9 Translate the distributive law $x + yz = (x + y)(x + z)$ in Table 5 into a logical equivalence.

Solution: To translate a Boolean identity into a logical equivalence, we change each Boolean variable into a propositional variable. Here we will change the Boolean variables x, y, and z into the propositional variables p, q, and r. Next, we change each Boolean sum into a disjunction and each Boolean product into a conjunction. (Note that 0 and 1 do not appear in this identity and

TABLE 6 Verifying One of the Distributive Laws.

x	y	z	$y + z$	xy	xz	$x(y + z)$	$xy + xz$
1	1	1	1	1	1	1	1
1	1	0	1	1	0	1	1
1	0	1	1	0	1	1	1
1	0	0	0	0	0	0	0
0	1	1	1	0	0	0	0
0	1	0	1	0	0	0	0
0	0	1	1	0	0	0	0
0	0	0	0	0	0	0	0

complementation also does not appear.) This transforms the Boolean identity into the logical equivalence

$$p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r).$$

This logical equivalence is one of the distributive laws for propositional logic in Table 6 in Section 1.3. ◀

Identities in Boolean algebra can be used to prove further identities. We demonstrate this in Example 10.

EXAMPLE 10 Prove the **absorption law** $x(x + y) = x$ using the other identities of Boolean algebra shown in Table 5. (This is called an absorption law because absorbing $x + y$ into x leaves x unchanged.)

Extra Examples

Solution: We display steps used to derive this identity and the law used in each step:

$$\begin{aligned} x(x + y) &= (x + 0)(x + y) && \text{Identity law for the Boolean sum} \\ &= x + 0 \cdot y && \text{Distributive law of the Boolean sum over the Boolean product} \\ &= x + y \cdot 0 && \text{Commutative law for the Boolean product} \\ &= x + 0 && \text{Domination law for the Boolean product} \\ &= x && \text{Identity law for the Boolean sum.} \end{aligned}$$

◀

Duality

The identities in Table 5 come in pairs (except for the law of the double complement and the unit and zero properties). To explain the relationship between the two identities in each pair we use the concept of a dual. The **dual** of a Boolean expression is obtained by interchanging Boolean sums and Boolean products and interchanging 0s and 1s.

EXAMPLE 11 Find the duals of $x(y + 0)$ and $\overline{x} \cdot 1 + (\overline{y} + z)$.

Solution: Interchanging $\cdot$ signs and $+$ signs and interchanging 0s and 1s in these expressions produces their duals. The duals are $x + (y \cdot 1)$ and $(\overline{x} + 0)(\overline{y}z)$, respectively. ◀

The dual of a Boolean function F represented by a Boolean expression is the function represented by the dual of this expression. This dual function, denoted by F^d, does not depend on the particular Boolean expression used to represent F. An identity between functions represented by Boolean expressions remains valid when the duals of both sides of the identity are taken. (See Exercise 30 for the reason why this is true.) This result, called the **duality principle**, is useful for obtaining new identities.

EXAMPLE 12 Construct an identity from the absorption law $x(x + y) = x$ by taking duals.

Solution: Taking the duals of both sides of this identity produces the identity $x + xy = x$, which is also called an absorption law and is shown in Table 5. ◀

The Abstract Definition of a Boolean Algebra

In this section we have focused on Boolean functions and expressions. However, the results we have established can be translated into results about propositions or results about sets. Because of this, it is useful to define Boolean algebras abstractly. Once it is shown that a particular structure is a Boolean algebra, then all results established about Boolean algebras in general apply to this particular structure.

Boolean algebras can be defined in several ways. The most common way is to specify the properties that operations must satisfy, as is done in Definition 1.

DEFINITION 1

A *Boolean algebra* is a set B with two binary operations $\vee$ and $\wedge$, elements 0 and 1, and a unary operation $^-$ such that these properties hold for all x, y, and z in B:

$$\left.\begin{aligned} x \vee 0 &= x \\ x \wedge 1 &= x \end{aligned}\right\} \quad \text{Identity laws}$$

$$\left.\begin{aligned} x \vee \overline{x} &= 1 \\ x \wedge \overline{x} &= 0 \end{aligned}\right\} \quad \text{Complement laws}$$

$$\left.\begin{aligned} (x \vee y) \vee z &= x \vee (y \vee z) \\ (x \wedge y) \wedge z &= x \wedge (y \wedge z) \end{aligned}\right\} \quad \text{Associative laws}$$

$$\left.\begin{aligned} x \vee y &= y \vee x \\ x \wedge y &= y \wedge x \end{aligned}\right\} \quad \text{Commutative laws}$$

$$\left.\begin{aligned} x \vee (y \wedge z) &= (x \vee y) \wedge (x \vee z) \\ x \wedge (y \vee z) &= (x \wedge y) \vee (x \wedge z) \end{aligned}\right\} \quad \text{Distributive laws}$$

Using the laws given in Definition 1, it is possible to prove many other laws that hold for every Boolean algebra, such as idempotent and domination laws. (See Exercises 35–42.)

From our previous discussion, $B = \{0, 1\}$ with the *OR* and *AND* operations and the complement operator, satisfies all these properties. The set of propositions in n variables, with the $\vee$ and $\wedge$ operators, **F** and **T**, and the negation operator, also satisfies all the properties of a Boolean algebra, as can be seen from Table 6 in Section 1.3. Similarly, the set of subsets of a universal set U with the union and intersection operations, the empty set and the universal set, and the set complementation operator, is a Boolean algebra as can be seen by consulting Table 1 in Section 2.2. So, to establish results about each of Boolean expressions, propositions, and sets, we need only prove results about abstract Boolean algebras.

Boolean algebras may also be defined using the notion of a lattice, discussed in Chapter 9. Recall that a lattice L is a partially ordered set in which every pair of elements x, y has a least upper bound, denoted by $\text{lub}(x, y)$ and a greatest lower bound denoted by $\text{glb}(x, y)$. Given two elements x and y of L, we can define two operations $\vee$ and $\wedge$ on pairs of elements of L by $x \vee y = \text{lub}(x, y)$ and $x \wedge y = \text{glb}(x, y)$.

For a lattice L to be a Boolean algebra as specified in Definition 1, it must have two properties. First, it must be **complemented**. For a lattice to be complemented it must have a least element 0 and a greatest element 1 and for every element x of the lattice there must exist an element $\overline{x}$ such that $x \vee \overline{x} = 1$ and $x \wedge \overline{x} = 0$. Second, it must be **distributive**. This means that for every x, y, and z in L, $x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$ and $x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$. Showing that a complemented, distributive lattice is a Boolean algebra has been left as Supplementary Exercise 39 in Chapter 9.

Exercises

1. Find the values of these expressions.
a) $1 \cdot \overline{0}$ **b)** $1 + \overline{1}$ **c)** $\overline{0} \cdot 0$ **d)** $\overline{(1 + 0)}$

2. Find the values, if any, of the Boolean variable x that satisfy these equations.
a) $x \cdot 1 = 0$ **b)** $x + x = 0$
c) $x \cdot 1 = x$ **d)** $x \cdot \overline{x} = 1$

3. a) Show that $(1 \cdot 1) + (\overline{0 \cdot 1} + 0) = 1$.
b) Translate the equation in part (a) into a propositional equivalence by changing each 0 into an **F**, each 1 into a **T**, each Boolean sum into a disjunction, each Boolean product into a conjunction, each complementation into a negation, and the equals sign into a propositional equivalence sign.

4. a) Show that $(1 \cdot 0) + (1 \cdot 0) = 1$.
b) Translate the equation in part (a) into a propositional equivalence by changing each 0 into an **F**, each 1 into a **T**, each Boolean sum into a disjunction, each Boolean product into a conjunction, each complementation into a negation, and the equals sign into a propositional equivalence sign.

5. Use a table to express the values of each of these Boolean functions.
a) $F(x, y, z) = \overline{x}y$
b) $F(x, y, z) = x + yz$
c) $F(x, y, z) = x\overline{y} + \overline{(xyz)}$
d) $F(x, y, z) = x(yz + \overline{y}\,\overline{z})$

6. Use a table to express the values of each of these Boolean functions.
a) $F(x, y, z) = \overline{z}$
b) $F(x, y, z) = \overline{x}y + \overline{y}z$
c) $F(x, y, z) = x\overline{y}z + \overline{(xyz)}$
d) $F(x, y, z) = \overline{y}(xz + \overline{x}\,\overline{z})$

7. Use a 3-cube Q_3 to represent each of the Boolean functions in Exercise 5 by displaying a black circle at each vertex that corresponds to a 3-tuple where this function has the value 1.

8. Use a 3-cube Q_3 to represent each of the Boolean functions in Exercise 6 by displaying a black circle at each vertex that corresponds to a 3-tuple where this function has the value 1.

9. What values of the Boolean variables x and y satisfy $xy = x + y$?

10. How many different Boolean functions are there of degree 7?

11. Prove the absorption law $x + xy = x$ using the other laws in Table 5.

☞ **12.** Show that $F(x, y, z) = xy + xz + yz$ has the value 1 if and only if at least two of the variables x, y, and z have the value 1.

13. Show that $x\overline{y} + y\overline{z} + \overline{x}z = \overline{x}y + \overline{y}z + x\overline{z}$.

Exercises 14–23 deal with the Boolean algebra $\{0, 1\}$ with addition, multiplication, and complement defined at the beginning of this section. In each case, use a table as in Example 8.

14. Verify the law of the double complement.

15. Verify the idempotent laws.

16. Verify the identity laws.

17. Verify the domination laws.

18. Verify the commutative laws.

19. Verify the associative laws.

20. Verify the first distributive law in Table 5.

21. Verify De Morgan's laws.

22. Verify the unit property.

23. Verify the zero property.

The Boolean operator $\oplus$, called the *XOR* operator, is defined by $1 \oplus 1 = 0$, $1 \oplus 0 = 1$, $0 \oplus 1 = 1$, and $0 \oplus 0 = 0$.

24. Simplify these expressions.
a) $x \oplus 0$ **b)** $x \oplus 1$
c) $x \oplus x$ **d)** $x \oplus \overline{x}$

25. Show that these identities hold.
a) $x \oplus y = (x + y)\overline{(xy)}$
b) $x \oplus y = (x\overline{y}) + (\overline{x}y)$

26. Show that $x \oplus y = y \oplus x$.

27. Prove or disprove these equalities.
a) $x \oplus (y \oplus z) = (x \oplus y) \oplus z$
b) $x + (y \oplus z) = (x + y) \oplus (x + z)$
c) $x \oplus (y + z) = (x \oplus y) + (x \oplus z)$

28. Find the duals of these Boolean expressions.
a) $x + y$ **b)** $\overline{x}\,\overline{y}$
c) $xyz + \overline{x}\,\overline{y}\,\overline{z}$ **d)** $x\overline{z} + x \cdot 0 + \overline{x} \cdot 1$

***29.** Suppose that F is a Boolean function represented by a Boolean expression in the variables $x_1, \ldots, x_n$. Show that $F^d(x_1, \ldots, x_n) = \overline{F(\overline{x}_1, \ldots, \overline{x}_n)}$.

***30.** Show that if F and G are Boolean functions represented by Boolean expressions in n variables and $F = G$, then $F^d = G^d$, where F^d and G^d are the Boolean functions represented by the duals of the Boolean expressions representing F and G, respectively. [*Hint:* Use the result of Exercise 29.]

***31.** How many different Boolean functions $F(x, y, z)$ are there such that $F(\overline{x}, \overline{y}, \overline{z}) = F(x, y, z)$ for all values of the Boolean variables x, y, and z?

***32.** How many different Boolean functions $F(x, y, z)$ are there such that $F(\overline{x}, y, z) = F(x, \overline{y}, z) = F(x, y, \overline{z})$ for all values of the Boolean variables x, y, and z?

33. Show that you obtain De Morgan's laws for propositions (in Table 6 in Section 1.3) when you transform De Morgan's laws for Boolean algebra in Table 6 into logical equivalences.

34. Show that you obtain the absorption laws for propositions (in Table 6 in Section 1.3) when you transform the absorption laws for Boolean algebra in Table 6 into logical equivalences.

In Exercises 35–42, use the laws in Definition 1 to show that the stated properties hold in every Boolean algebra.

35. Show that in a Boolean algebra, the **idempotent laws** $x \vee x = x$ and $x \wedge x = x$ hold for every element x.

36. Show that in a Boolean algebra, every element x has a unique complement $\overline{x}$ such that $x \vee \overline{x} = 1$ and $x \wedge \overline{x} = 0$.

37. Show that in a Boolean algebra, the complement of the element 0 is the element 1 and vice versa.

38. Prove that in a Boolean algebra, the **law of the double complement** holds; that is, $\overline{\overline{x}} = x$ for every element x.

39. Show that **De Morgan's laws** hold in a Boolean algebra. That is, show that for all x and y, $\overline{(x \vee y)} = \overline{x} \wedge \overline{y}$ and $\overline{(x \wedge y)} = \overline{x} \vee \overline{y}$.

40. Show that in a Boolean algebra, the **modular properties** hold. That is, show that $x \wedge (y \vee (x \wedge z)) = (x \wedge y) \vee (x \wedge z)$ and $x \vee (y \wedge (x \vee z)) = (x \vee y) \wedge (x \vee z)$.

41. Show that in a Boolean algebra, if $x \vee y = 0$, then $x = 0$ and $y = 0$, and that if $x \wedge y = 1$, then $x = 1$ and $y = 1$.

42. Show that in a Boolean algebra, the **dual** of an identity, obtained by interchanging the $\vee$ and $\wedge$ operators and interchanging the elements 0 and 1, is also a valid identity.

43. Show that a complemented, distributive lattice is a Boolean algebra.

12.2 Representing Boolean Functions

Two important problems of Boolean algebra will be studied in this section. The first problem is: Given the values of a Boolean function, how can a Boolean expression that represents this function be found? This problem will be solved by showing that any Boolean function can be represented by a Boolean sum of Boolean products of the variables and their complements. The solution of this problem shows that every Boolean function can be represented using the three Boolean operators $\cdot$, $+$, and $\overline{\ }$. The second problem is: Is there a smaller set of operators that can be used to represent all Boolean functions? We will answer this question by showing that all Boolean functions can be represented using only one operator. Both of these problems have practical importance in circuit design.

Sum-of-Products Expansions

We will use examples to illustrate one important way to find a Boolean expression that represents a Boolean function.

EXAMPLE 1 Find Boolean expressions that represent the functions $F(x, y, z)$ and $G(x, y, z)$, which are given in Table 1.

TABLE 1

x	y	z	F	G
1	1	1	0	0
1	1	0	0	1
1	0	1	1	0
1	0	0	0	0
0	1	1	0	0
0	1	0	0	1
0	0	1	0	0
0	0	0	0	0

Solution: An expression that has the value 1 when $x = z = 1$ and $y = 0$, and the value 0 otherwise, is needed to represent F. Such an expression can be formed by taking the Boolean product of x, $\overline{y}$, and z. This product, $x\overline{y}z$, has the value 1 if and only if $x = \overline{y} = z = 1$, which holds if and only if $x = z = 1$ and $y = 0$.

To represent G, we need an expression that equals 1 when $x = y = 1$ and $z = 0$, or $x = z = 0$ and $y = 1$. We can form an expression with these values by taking the Boolean sum of two different Boolean products. The Boolean product $xy\overline{z}$ has the value 1 if and only if $x = y = 1$ and $z = 0$. Similarly, the product $\overline{x}y\overline{z}$ has the value 1 if and only if $x = z = 0$ and $y = 1$. The Boolean sum of these two products, $xy\overline{z} + \overline{x}y\overline{z}$, represents G, because it has the value 1 if and only if $x = y = 1$ and $z = 0$, or $x = z = 0$ and $y = 1$. ◀

Example 1 illustrates a procedure for constructing a Boolean expression representing a function with given values. Each combination of values of the variables for which the function has the value 1 leads to a Boolean product of the variables or their complements.

DEFINITION 1 A *literal* is a Boolean variable or its complement. A *minterm* of the Boolean variables $x_1, x_2, \ldots, x_n$ is a Boolean product $y_1 y_2 \cdots y_n$, where $y_i = x_i$ or $y_i = \overline{x}_i$. Hence, a minterm is a product of n literals, with one literal for each variable.

A minterm has the value 1 for one and only one combination of values of its variables. More precisely, the minterm $y_1 y_2 \ldots y_n$ is 1 if and only if each y_i is 1, and this occurs if and only if $x_i = 1$ when $y_i = x_i$ and $x_i = 0$ when $y_i = \overline{x}_i$.

EXAMPLE 2 Find a minterm that equals 1 if $x_1 = x_3 = 0$ and $x_2 = x_4 = x_5 = 1$, and equals 0 otherwise.

Solution: The minterm $\overline{x}_1 x_2 \overline{x}_3 x_4 x_5$ has the correct set of values. ◀

By taking Boolean sums of distinct minterms we can build up a Boolean expression with a specified set of values. In particular, a Boolean sum of minterms has the value 1 when exactly one of the minterms in the sum has the value 1. It has the value 0 for all other combinations of values of the variables. Consequently, given a Boolean function, a Boolean sum of minterms can be formed that has the value 1 when this Boolean function has the value 1, and has the value 0 when the function has the value 0. The minterms in this Boolean sum correspond to those combinations of values for which the function has the value 1. The sum of minterms that represents the function is called the **sum-of-products expansion** or the **disjunctive normal form** of the Boolean function.

Links

(See Exercise 42 in Section 1.3 for the development of disjunctive normal form in propositional calculus.)

EXAMPLE 3 Find the sum-of-products expansion for the function $F(x, y, z) = (x + y)\overline{z}$.

Extra Examples

Solution: We will find the sum-of-products expansion of $F(x, y, z)$ in two ways. First, we will use Boolean identities to expand the product and simplify. We find that

$$\begin{aligned} F(x, y, z) &= (x + y)\overline{z} \\ &= x\overline{z} + y\overline{z} && \text{Distributive law} \\ &= x1\overline{z} + 1y\overline{z} && \text{Identity law} \\ &= x(y + \overline{y})\overline{z} + (x + \overline{x})y\overline{z} && \text{Unit property} \\ &= xy\overline{z} + x\overline{y}\,\overline{z} + xy\overline{z} + \overline{x}y\overline{z} && \text{Distributive law} \\ &= xy\overline{z} + x\overline{y}\,\overline{z} + \overline{x}y\,\overline{z}. && \text{Idempotent law} \end{aligned}$$

Second, we can construct the sum-of-products expansion by determining the values of F for all possible values of the variables x, y, and z. These values are found in Table 2. The sum-of-products expansion of F is the Boolean sum of three minterms corresponding to the three rows of this table that give the value 1 for the function. This gives

$$F(x, y, z) = xy\overline{z} + x\overline{y}\,\overline{z} + \overline{x}y\overline{z}.$$ ◀

It is also possible to find a Boolean expression that represents a Boolean function by taking a Boolean product of Boolean sums. The resulting expansion is called the **conjunctive normal form** or **product-of-sums expansion** of the function. These expansions can be found from sum-of-products expansions by taking duals. How to find such expansions directly is described in Exercise 10.

TABLE 2

x	y	z	$x + y$	$\overline{z}$	$(x + y)\overline{z}$
1	1	1	1	0	0
1	1	0	1	1	1
1	0	1	1	0	0
1	0	0	1	1	1
0	1	1	1	0	0
0	1	0	1	1	1
0	0	1	0	0	0
0	0	0	0	1	0

Functional Completeness

Every Boolean function can be expressed as a Boolean sum of minterms. Each minterm is the Boolean product of Boolean variables or their complements. This shows that every Boolean function can be represented using the Boolean operators $\cdot$, $+$, and $^-$. Because every Boolean function can be represented using these operators we say that the set $\{\cdot, +, ^-\}$ is **functionally complete**. Can we find a smaller set of functionally complete operators? We can do so if one of the three operators of this set can be expressed in terms of the other two. This can be done using one of De Morgan's laws. We can eliminate all Boolean sums using the identity

$$x + y = \overline{\overline{x}\,\overline{y}},$$

which is obtained by taking complements of both sides in the second De Morgan law, given in Table 5 in Section 12.1, and then applying the double complementation law. This means that the set $\{\cdot, ^-\}$ is functionally complete. Similarly, we could eliminate all Boolean products using the identity

$$xy = \overline{\overline{x} + \overline{y}},$$

which is obtained by taking complements of both sides in the first De Morgan law, given in Table 5 in Section 12.1, and then applying the double complementation law. Consequently $\{+, ^-\}$ is functionally complete. Note that the set $\{+, \cdot\}$ is not functionally complete, because it is impossible to express the Boolean function $F(x) = \overline{x}$ using these operators (see Exercise 19).

We have found sets containing two operators that are functionally complete. Can we find a smaller set of functionally complete operators, namely, a set containing just one operator? Such sets exist. Define two operators, the $\mid$ or ***NAND*** operator, defined by $1 \mid 1 = 0$ and $1 \mid 0 = 0 \mid 1 = 0 \mid 0 = 1$; and the $\downarrow$ or ***NOR*** operator, defined by $1 \downarrow 1 = 1 \downarrow 0 = 0 \downarrow 1 = 0$ and $0 \downarrow 0 = 1$. Both of the sets $\{\mid\}$ and $\{\downarrow\}$ are functionally complete. To see that $\{\mid\}$ is functionally complete, because $\{\cdot, ^-\}$ is functionally complete, all that we have to do is show that both of the operators $\cdot$ and $^-$ can be expressed using just the $\mid$ operator. This can be done as

$$\overline{x} = x \mid x,$$
$$xy = (x \mid y) \mid (x \mid y).$$

The reader should verify these identities (see Exercise 14). We leave the demonstration that $\{\downarrow\}$ is functionally complete for the reader (see Exercises 15 and 16).

Exercises

1. Find a Boolean product of the Boolean variables x, y, and z, or their complements, that has the value 1 if and only if

a) $x = y = 0, z = 1$. **b)** $x = 0, y = 1, z = 0$.
c) $x = 0, y = z = 1$. **d)** $x = y = z = 0$.

2. Find the sum-of-products expansions of these Boolean functions.

a) $F(x, y) = \overline{x} + y$ **b)** $F(x, y) = x\,\overline{y}$
c) $F(x, y) = 1$ **d)** $F(x, y) = \overline{y}$

3. Find the sum-of-products expansions of these Boolean functions.

a) $F(x, y, z) = x + y + z$
b) $F(x, y, z) = (x + z)y$
c) $F(x, y, z) = x$
d) $F(x, y, z) = x\,\overline{y}$

4. Find the sum-of-products expansions of the Boolean function $F(x, y, z)$ that equals 1 if and only if

a) $x = 0$. **b)** $xy = 0$.
c) $x + y = 0$. **d)** $xyz = 0$.

5. Find the sum-of-products expansion of the Boolean function $F(w, x, y, z)$ that has the value 1 if and only if an odd number of w, x, y, and z have the value 1.

6. Find the sum-of-products expansion of the Boolean function $F(x_1, x_2, x_3, x_4, x_5)$ that has the value 1 if and only if three or more of the variables x_1, x_2, x_3, x_4, and x_5 have the value 1.

Another way to find a Boolean expression that represents a Boolean function is to form a Boolean product of Boolean sums of literals. Exercises 7–11 are concerned with representations of this kind.

7. Find a Boolean sum containing either x or $\overline{x}$, either y or $\overline{y}$, and either z or $\overline{z}$ that has the value 0 if and only if

a) $x = y = 1, z = 0$. **b)** $x = y = z = 0$.
c) $x = z = 0, y = 1$.

8. Find a Boolean product of Boolean sums of literals that has the value 0 if and only if $x = y = 1$ and $z = 0$, $x = z = 0$ and $y = 1$, or $x = y = z = 0$. [*Hint:* Take the Boolean product of the Boolean sums found in parts (a), (b), and (c) in Exercise 7.]

9. Show that the Boolean sum $y_1 + y_2 + \cdots + y_n$, where $y_i = x_i$ or $y_i = \overline{x}_i$, has the value 0 for exactly one combination of the values of the variables, namely, when $x_i = 0$ if $y_i = x_i$ and $x_i = 1$ if $y_i = \overline{x}_i$. This Boolean sum is called a **maxterm**.

10. Show that a Boolean function can be represented as a Boolean product of maxterms. This representation is called the **product-of-sums expansion** or **conjunctive normal form** of the function. [*Hint:* Include one maxterm in this product for each combination of the variables where the function has the value 0.]

11. Find the product-of-sums expansion of each of the Boolean functions in Exercise 3.

12. Express each of these Boolean functions using the operators $\cdot$ and $^-$.

a) $x + y + z$ **b)** $x + \overline{y}(\overline{x} + z)$
c) $\overline{x + \overline{y}}$ **d)** $\overline{x}(x + \overline{y} + \overline{z})$

13. Express each of the Boolean functions in Exercise 12 using the operators $+$ and $^-$.

14. Show that

a) $\overline{x} = x \mid x$. **b)** $xy = (x \mid y) \mid (x \mid y)$.
c) $x + y = (x \mid x) \mid (y \mid y)$.

15. Show that

a) $\overline{x} = x \downarrow x$.
b) $xy = (x \downarrow x) \downarrow (y \downarrow y)$.
c) $x + y = (x \downarrow y) \downarrow (x \downarrow y)$.

16. Show that $\{ \downarrow \}$ is functionally complete using Exercise 15.

17. Express each of the Boolean functions in Exercise 3 using the operator $\mid$.

18. Express each of the Boolean functions in Exercise 3 using the operator $\downarrow$.

19. Show that the set of operators $\{+, \cdot\}$ is not functionally complete.

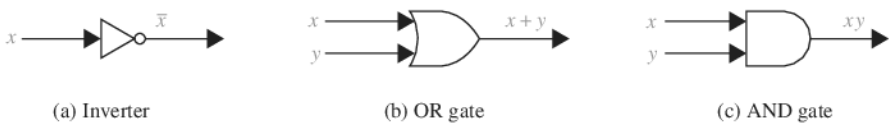
20. Are these sets of operators functionally complete?
a) $\{+, \oplus\}$ **b)** $\{^-, \oplus\}$ **c)** $\{\cdot, \oplus\}$

12.3 Logic Gates

Introduction

Links

Boolean algebra is used to model the circuitry of electronic devices. Each input and each output of such a device can be thought of as a member of the set $\{0, 1\}$. A computer, or other electronic device, is made up of a number of circuits. Each circuit can be designed using the rules of Boolean algebra that were studied in Sections 12.1 and 12.2. The basic elements of circuits



(a) Inverter (b) OR gate (c) AND gate

FIGURE 1 Basic Types of Gates.

are called **gates**, and were introduced in Section 1.2. Each type of gate implements a Boolean operation. In this section we define several types of gates. Using these gates, we will apply the rules of Boolean algebra to design circuits that perform a variety of tasks. The circuits that we will study in this chapter give output that depends only on the input, and not on the current state of the circuit. In other words, these circuits have no memory capabilities. Such circuits are called **combinational circuits** or **gating networks**.

We will construct combinational circuits using three types of elements. The first is an **inverter**, which accepts the value of one Boolean variable as input and produces the complement of this value as its output. The symbol used for an inverter is shown in Figure 1(a). The input to the inverter is shown on the left side entering the element, and the output is shown on the right side leaving the element.

The next type of element we will use is the **OR gate**. The inputs to this gate are the values of two or more Boolean variables. The output is the Boolean sum of their values. The symbol used for an OR gate is shown in Figure 1(b). The inputs to the OR gate are shown on the left side entering the element, and the output is shown on the right side leaving the element.

The third type of element we will use is the **AND gate**. The inputs to this gate are the values of two or more Boolean variables. The output is the Boolean product of their values. The symbol used for an AND gate is shown in Figure 1(c). The inputs to the AND gate are shown on the left side entering the element, and the output is shown on the right side leaving the element.

We will permit multiple inputs to AND and OR gates. The inputs to each of these gates are shown on the left side entering the element, and the output is shown on the right side. Examples of AND and OR gates with n inputs are shown in Figure 2.

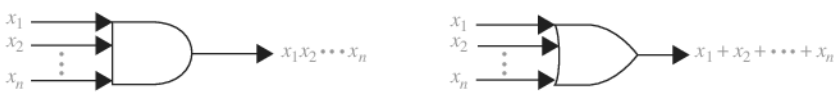


FIGURE 2 Gates with *n* Inputs.

Combinations of Gates

Combinational circuits can be constructed using a combination of inverters, OR gates, and AND gates. When combinations of circuits are formed, some gates may share inputs. This is shown in one of two ways in depictions of circuits. One method is to use branchings that indicate all the gates that use a given input. The other method is to indicate this input separately for each gate. Figure 3 illustrates the two ways of showing gates with the same input values. Note also that output from a gate may be used as input by one or more other elements, as shown in Figure 3. Both drawings in Figure 3 depict the circuit that produces the output $xy + \overline{x}y$.

EXAMPLE 1 Construct circuits that produce the following outputs: (a) $(x + y)\overline{x}$, (b) $\overline{x}\,\overline{(y + \overline{z})}$, and (c) $(x + y + z)(\overline{x}\,\overline{y}\,\overline{z})$.

Solution: Circuits that produce these outputs are shown in Figure 4. ◀

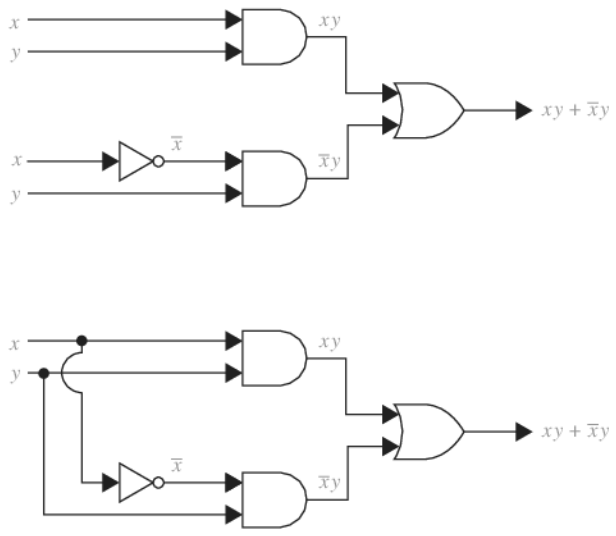


FIGURE 3 **Two Ways to Draw the Same Circuit.**

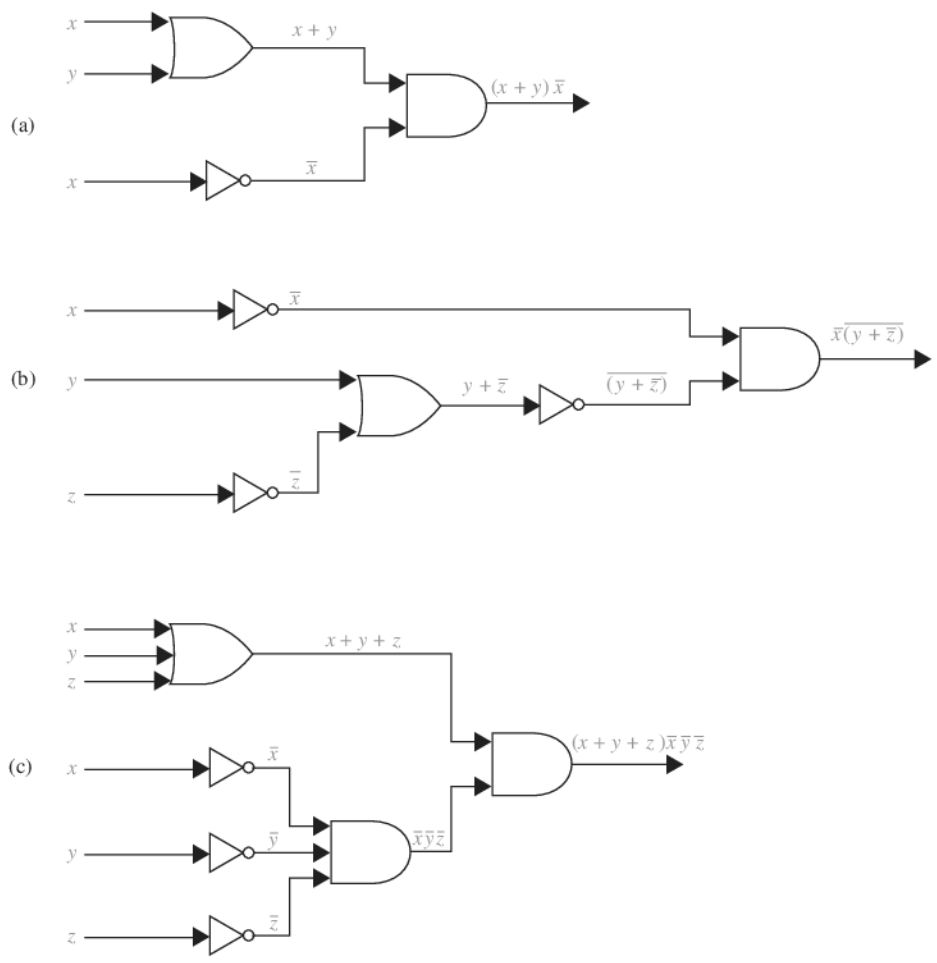


FIGURE 4 **Circuits that Produce the Outputs Specified in Example 1.**

Examples of Circuits

We will give some examples of circuits that perform some useful functions.

EXAMPLE 2 A committee of three individuals decides issues for an organization. Each individual votes either yes or no for each proposal that arises. A proposal is passed if it receives at least two yes votes. Design a circuit that determines whether a proposal passes.

Extra Examples

Solution: Let $x = 1$ if the first individual votes yes, and $x = 0$ if this individual votes no; let $y = 1$ if the second individual votes yes, and $y = 0$ if this individual votes no; let $z = 1$ if the third individual votes yes, and $z = 0$ if this individual votes no. Then a circuit must be designed that produces the output 1 from the inputs x, y, and z when two or more of x, y, and z are 1. One representation of the Boolean function that has these output values is $xy + xz + yz$ (see Exercise 12 in Section 12.1). The circuit that implements this function is shown in Figure 5. ◀

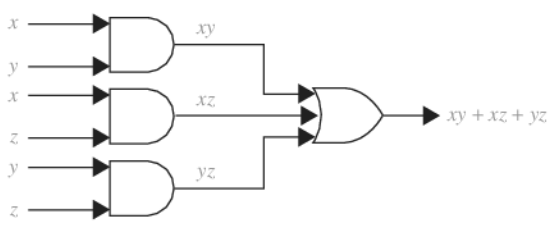


FIGURE 5 **A Circuit for Majority Voting.**

EXAMPLE 3 Sometimes light fixtures are controlled by more than one switch. Circuits need to be designed so that flipping any one of the switches for the fixture turns the light on when it is off and turns the light off when it is on. Design circuits that accomplish this when there are two switches and when there are three switches.

TABLE 1

x	y	$F(x, y)$
1	1	1
1	0	0
0	1	0
0	0	1

Solution: We will begin by designing the circuit that controls the light fixture when two different switches are used. Let $x = 1$ when the first switch is closed and $x = 0$ when it is open, and let $y = 1$ when the second switch is closed and $y = 0$ when it is open. Let $F(x, y) = 1$ when the light is on and $F(x, y) = 0$ when it is off. We can arbitrarily decide that the light will be on when both switches are closed, so that $F(1, 1) = 1$. This determines all the other values of F. When one of the two switches is opened, the light goes off, so $F(1, 0) = F(0, 1) = 0$. When the other switch is also opened, the light goes on, so $F(0, 0) = 1$. Table 1 displays these values. Note that $F(x, y) = xy + \overline{x}\,\overline{y}$. This function is implemented by the circuit shown in Figure 6.

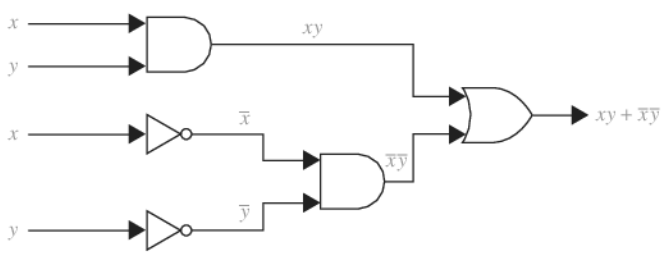


FIGURE 6 **A Circuit for a Light Controlled by Two Switches.**

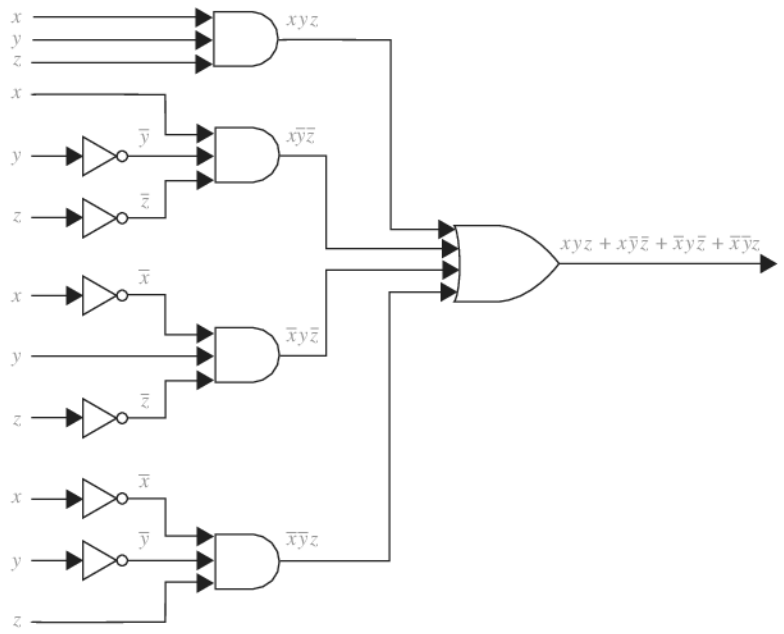


FIGURE 7 **A Circuit for a Fixture Controlled by Three Switches.**

TABLE 2			
x	y	z	$F(x, y, z)$
1	1	1	1
1	1	0	0
1	0	1	0
1	0	0	1
0	1	1	0
0	1	0	1
0	0	1	1
0	0	0	0

We will now design a circuit for three switches. Let x, y, and z be the Boolean variables that indicate whether each of the three switches is closed. We let $x = 1$ when the first switch is closed, and $x = 0$ when it is open; $y = 1$ when the second switch is closed, and $y = 0$ when it is open; and $z = 1$ when the third switch is closed, and $z = 0$ when it is open. Let $F(x, y, z) = 1$ when the light is on and $F(x, y, z) = 0$ when the light is off. We can arbitrarily specify that the light be on when all three switches are closed, so that $F(1, 1, 1) = 1$. This determines all other values of F. When one switch is opened, the light goes off, so $F(1, 1, 0) = F(1, 0, 1) = F(0, 1, 1) = 0$. When a second switch is opened, the light goes on, so $F(1, 0, 0) = F(0, 1, 0) = F(0, 0, 1) = 1$. Finally, when the third switch is opened, the light goes off again, so $F(0, 0, 0) = 0$. Table 2 shows the values of this function.

The function F can be represented by its sum-of-products expansion as $F(x, y, z) = xyz + x\overline{y}\,\overline{z} + \overline{x}y\overline{z} + \overline{x}\,\overline{y}z$. The circuit shown in Figure 7 implements this function. ◀

Adders

Links

TABLE 3 **Input and Output for the Half Adder.**			
Input		*Output*	
x	y	s	c
1	1	0	1
1	0	1	0
0	1	1	0
0	0	0	0

We will illustrate how logic circuits can be used to carry out addition of two positive integers from their binary expansions. We will build up the circuitry to do this addition from some component circuits. First, we will build a circuit that can be used to find $x + y$, where x and y are two bits. The input to our circuit will be x and y, because these each have the value 0 or the value 1. The output will consist of two bits, namely, s and c, where s is the sum bit and c is the carry bit. This circuit is called a **multiple output circuit** because it has more than one output. The circuit that we are designing is called the **half adder**, because it adds two bits, without considering a carry from a previous addition. We show the input and output for the half adder in Table 3. From Table 3 we see that $c = xy$ and that $s = x\overline{y} + \overline{x}y = (x + y)\overline{(xy)}$. Hence, the circuit shown in Figure 8 computes the sum bit s and the carry bit c from the bits x and y.

We use the **full adder** to compute the sum bit and the carry bit when two bits and a carry are added. The inputs to the full adder are the bits x and y and the carry c_i. The outputs are the sum bit s and the new carry c_{i+1}. The inputs and outputs for the full adder are shown in Table 4.

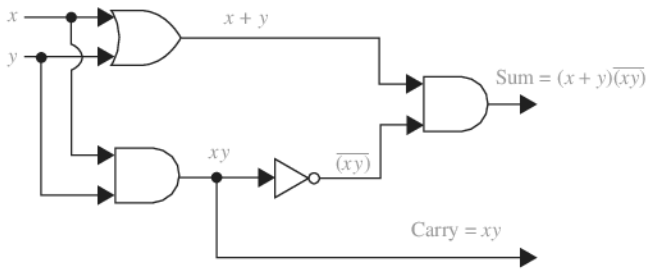


FIGURE 8 **The Half Adder.**

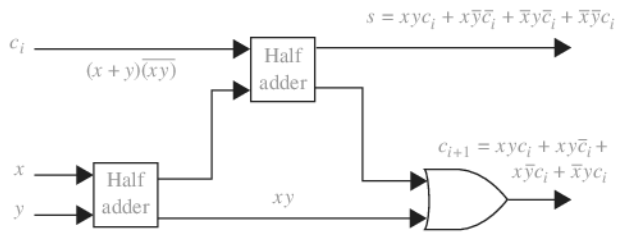


FIGURE 9 **A Full Adder.**

TABLE 4 Input and Output for the Full Adder.

Input			*Output*	
x	y	c_i	s	c_{i+1}
1	1	1	1	1
1	1	0	0	1
1	0	1	0	1
1	0	0	1	0
0	1	1	0	1
0	1	0	1	0
0	0	1	1	0
0	0	0	0	0

The two outputs of the full adder, the sum bit s and the carry c_{i+1}, are given by the sum-of-products expansions $xyc_i + x\overline{y}\,\overline{c}_i + \overline{x}y\overline{c}_i + \overline{x}\,\overline{y}c_i$ and $xyc_i + xy\overline{c}_i + x\overline{y}c_i + \overline{x}yc_i$, respectively. However, instead of designing the full adder from scratch, we will use half adders to produce the desired output. A full adder circuit using half adders is shown in Figure 9.

Finally, in Figure 10 we show how full and half adders can be used to add the two three-bit integers $(x_2x_1x_0)_2$ and $(y_2y_1y_0)_2$ to produce the sum $(s_3s_2s_1s_0)_2$. Note that s_3, the highest-order bit in the sum, is given by the carry c_2.

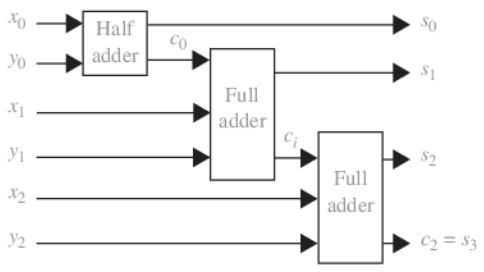
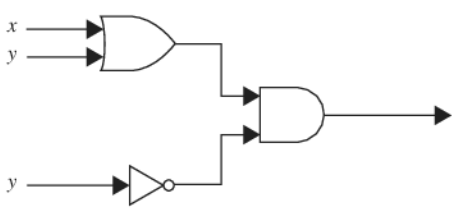


FIGURE 10 **Adding Two Three-Bit Integers with Full and Half Adders.**

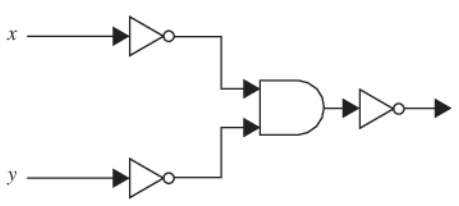
Exercises

In Exercises 1–5 find the output of the given circuit.

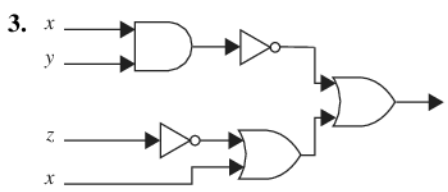
1.



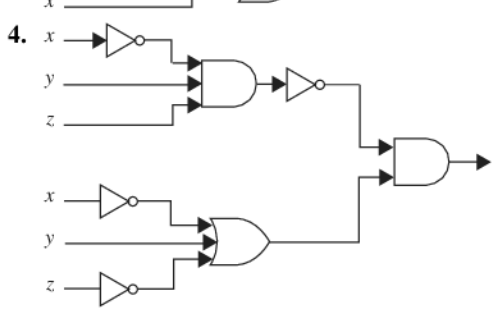
2.

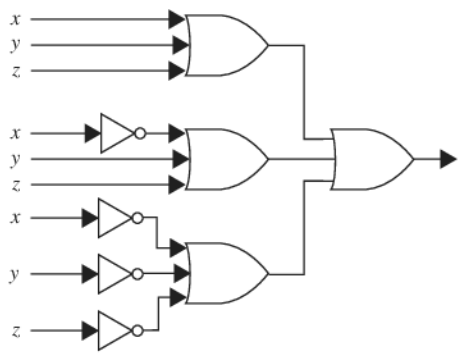


3.



4.



5. 

6. Construct circuits from inverters, AND gates, and OR gates to produce these outputs.
a) $\overline{x} + y$ **b)** $\overline{(x + y)}x$
c) $xyz + \overline{x}\,\overline{y}\,\overline{z}$ **d)** $\overline{(\overline{x} + z)(y + \overline{z})}$

7. Design a circuit that implements majority voting for five individuals.

8. Design a circuit for a light fixture controlled by four switches, where flipping one of the switches turns the light on when it is off and turns it off when it is on.

9. Show how the sum of two five-bit integers can be found using full and half adders.

10. Construct a circuit for a half subtractor using AND gates, OR gates, and inverters. A **half subtractor** has two bits as input and produces as output a difference bit and a borrow.

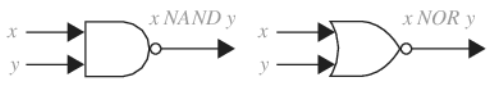
11. Construct a circuit for a full subtractor using AND gates, OR gates, and inverters. A **full subtractor** has two bits and a borrow as input, and produces as output a difference bit and a borrow.

12. Use the circuits from Exercises 10 and 11 to find the difference of two four-bit integers, where the first integer is greater than the second integer.

***13.** Construct a circuit that compares the two-bit integers $(x_1x_0)_2$ and $(y_1y_0)_2$, returning an output of 1 when the first of these numbers is larger and an output of 0 otherwise.

***14.** Construct a circuit that computes the product of the two-bit integers $(x_1x_0)_2$ and $(y_1y_0)_2$. The circuit should have four output bits for the bits in the product.

Two gates that are often used in circuits are NAND and NOR gates. When NAND or NOR gates are used to represent circuits, no other types of gates are needed. The notation for these gates is as follows:



***15.** Use NAND gates to construct circuits with these outputs.
a) $\overline{x}$ **b)** $x + y$
c) xy **d)** $x \oplus y$

***16.** Use NOR gates to construct circuits for the outputs given in Exercise 15.

***17.** Construct a half adder using NAND gates.

***18.** Construct a half adder using NOR gates.

A **multiplexer** is a switching circuit that produces as output one of a set of input bits based on the value of control bits.

19. Construct a multiplexer using AND gates, OR gates, and inverters that has as input the four bits x_0, x_1, x_2, and x_3 and the two control bits c_0 and c_1. Set up the circuit so that x_i is the output, where i is the value of the two-bit integer $(c_1c_0)_2$.

The **depth** of a combinatorial circuit can be defined by specifying that the depth of the initial input is 0 and if a gate has n different inputs at depths $d_1, d_2, \ldots, d_n$, respectively, then its outputs have depth equal to $\max(d_1, d_2, \ldots, d_n) + 1$; this value is also defined to be the depth of the gate. The depth of a combinatorial circuit is the maximum depth of the gates in the circuit.

20. Find the depth of
a) the circuit constructed in Example 2 for majority voting among three people.
b) the circuit constructed in Example 3 for a light controlled by two switches.
c) the half adder shown in Figure 8.
d) the full adder shown in Figure 9.

12.4 Minimization of Circuits

Introduction

The efficiency of a combinational circuit depends on the number and arrangement of its gates. The process of designing a combinational circuit begins with the table specifying the output for each combination of input values. We can always use the sum-of-products expansion of a circuit to find a set of logic gates that will implement this circuit. However, the sum-of-products expansion

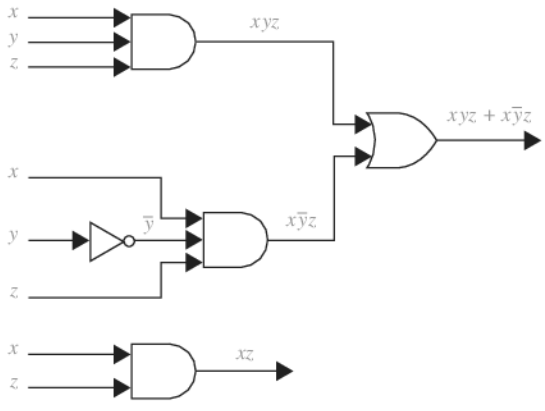


FIGURE 1 Two Circuits with the Same Output.

may contain many more terms than are necessary. Terms in a sum-of-products expansion that differ in just one variable, so that in one term this variable occurs and in the other term the complement of this variable occurs, can be combined. For instance, consider the circuit that has output 1 if and only if $x = y = z = 1$ or $x = z = 1$ and $y = 0$. The sum-of-products expansion of this circuit is $xyz + x\overline{y}z$. The two products in this expansion differ in exactly one variable, namely, y. They can be combined as

$$\begin{aligned} xyz + x\overline{y}z &= (y + \overline{y})(xz) \\ &= 1 \cdot (xz) \\ &= xz. \end{aligned}$$

Hence, xz is a Boolean expression with fewer operators that represents the circuit. We show two different implementations of this circuit in Figure 1. The second circuit uses only one gate, whereas the first circuit uses three gates and an inverter.

This example shows that combining terms in the sum-of-products expansion of a circuit leads to a simpler expression for the circuit. We will describe two procedures that simplify sum-of-products expansions.

The goal of both procedures is to produce Boolean sums of Boolean products that represent a Boolean function with the fewest products of literals such that these products contain the fewest literals possible among all sums of products that represent a Boolean function. Finding such a sum of products is called **minimization of the Boolean function**. Minimizing a Boolean function makes it possible to construct a circuit for this function that uses the fewest gates and fewest inputs to the *AND* gates and *OR* gates in the circuit, among all circuits for the Boolean expression we are minimizing.

Until the early 1960s logic gates were individual components. To reduce costs it was important to use the fewest gates to produce a desired output. However, in the mid-1960s, integrated circuit technology was developed that made it possible to combine gates on a single chip. Even though it is now possible to build increasingly complex integrated circuits on chips at low cost, minimization of Boolean functions remains important.

Reducing the number of gates on a chip can lead to a more reliable circuit and can reduce the cost to produce the chip. Also, minimization makes it possible to fit more circuits on the same chip. Furthermore, minimization reduces the number of inputs to gates in a circuit. This reduces the time used by a circuit to compute its output. Moreover, the number of inputs to a gate may be limited because of the particular technology used to build logic gates.

The first procedure we will introduce, known as Karnaugh maps (or K-maps), was designed in the 1950s to help minimize circuits by hand. K-maps are useful in minimizing circuits with up to six variables, although they become rather complex even for five or six variables. The