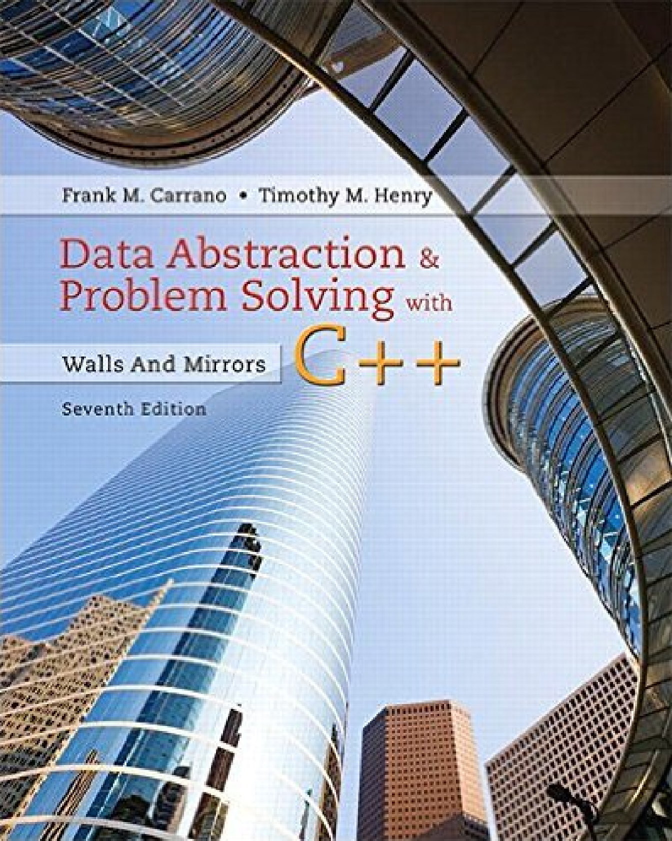
Frank M. Carrano • Timothy M. Henry

Data Abstraction & Problem Solving with C++

Walls And Mirrors

Seventh Edition



Chapter 20 Graphs

Contents

Prerequistes

Chapter 5	Recursion as a Problem-Solving Technique
Chapter 6	Stacks
Chapter 15	Trees

Graphs are an important mathematical concept with significant applications not only in computer science, but also in many other fields. You can view a graph as a mathematical construct, a data structure, or an abstract data type. This chapter provides an introduction to graphs that allows you to view a graph in any of these three ways. It also presents the major operations and applications of graphs that are relevant to the computer scientist.

20.1 Terminology

VideoNote

Graph concepts

You are undoubtedly familiar with graphs: Line graphs, bar graphs, and pie charts are in common use. The simple line graph in Figure 20-1 is an example of the type of graph that this chapter considers: a set of points that are joined by lines. Clearly, graphs provide a way to illustrate data. However, graphs also represent the relationships among data items, and it is this feature of graphs that is important here.

G = {*V*, *E*}; that is, a graph is a set of vertices and edges

A **graph** *G* consists of two sets: a set *V* of vertices, or nodes, and a set *E* of edges that connect the vertices. For example, the campus map in Figure 20-2a is a graph whose vertices represent buildings and whose edges represent the sidewalks between the buildings. This definition of a graph is more general than the definition of a line graph. In fact, a line graph, with its points and lines, is a special case of the general definition of a graph.

FIGURE 20-1 An ordinary line graph

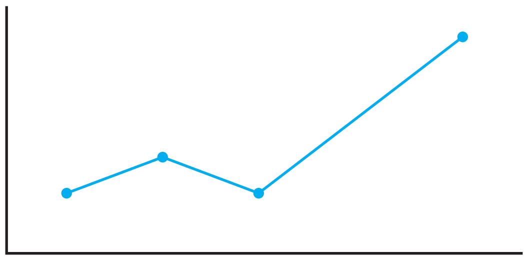
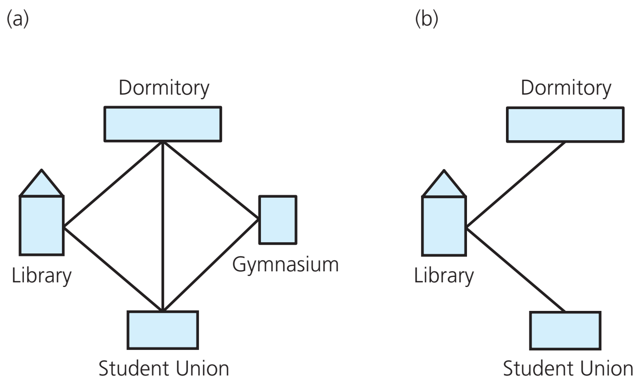


FIGURE 20-2 (a) A campus map as a graph; (b) a subgraph

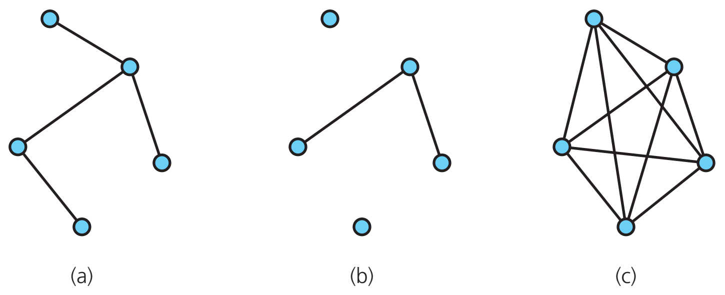


Adjacent vertices are joined by an edge

A path between two vertices is a sequence of edges

A **subgraph** consists of a subset of a graph's vertices and a subset of its edges. Figure 20-2b shows a subgraph of the graph in Figure 20-2a. Two vertices of a graph are **adjacent** if they are joined by an edge. In Figure 20-2b, the Library and the Student Union are adjacent. A **path** between two vertices is a sequence of edges that begins at one vertex and ends at another vertex. For example, there is a path in Figure 20-2a that begins at the Dormitory, leads first to the Library, then to the Student

FIGURE 20-3 Graphs that are (a) connected; (b) disconnected; and (c) complete



Union, and finally back to the Library. Although a path may pass through the same vertex more than once, as the path just described does, a **simple path** may not. The path Dormitory–Library–Student Union is a simple path. A **cycle** is a path that begins and ends at the same vertex; a **simple cycle** is a cycle that does not pass through other vertices more than once. The path Library–Student Union–Gymnasium–Dormitory–Library is a simple cycle in the graph in Figure 20-2a. A graph is **connected** if each pair of distinct vertices has a path between them. That is, in a connected graph you can get from any vertex to any other vertex by following a path. Figure 20-3a shows a connected graph. Notice that a connected graph does not necessarily have an edge between every pair of vertices. Figure 20-3b shows a **disconnected** graph.

A simple path passes through a vertex only once

A cycle is a path that begins and ends at the same vertex

A connected graph has a path between each pair of distinct vertices

In a **complete graph**, each pair of distinct vertices has an edge between them. The graph in Figure 20-3c is complete. Clearly, a complete graph is also connected, but the converse is not true; notice that the graph in Figure 20-3a is connected but is not complete.

A complete graph has an edge between each pair of distinct vertices

A complete graph is connected

Because a graph has a *set* of edges, a graph cannot have duplicate edges between vertices. However, a **multigraph**, as illustrated in Figure 20-4a, does allow multiple edges. Thus, a multigraph is not a graph. A graph's edges cannot begin and end at the same vertex. Figure 20-4b shows such an edge, which is called a **self edge** or **loop**.

A multigraph has multiple edges and so is not a graph

You can label the edges of a graph. When these labels represent numeric values, the graph is called a **weighted graph**. The graph in Figure 20-5a is a weighted graph whose edges are labeled with the distances between cities.

The edges of a weighted graph have numeric labels

All of the previous graphs are examples of **undirected graphs**, because the edges do not indicate a direction. That is, you can travel in either direction along the edges between the vertices of an undirected graph. In contrast, each edge in a **directed graph**, or **digraph**, has a direction and is called a

Each edge in a directed graph has a direction

FIGURE 20-4 (a) A multigraph is not a graph; (b) a self edge is not allowed in a graph

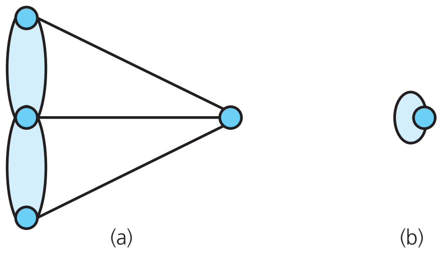
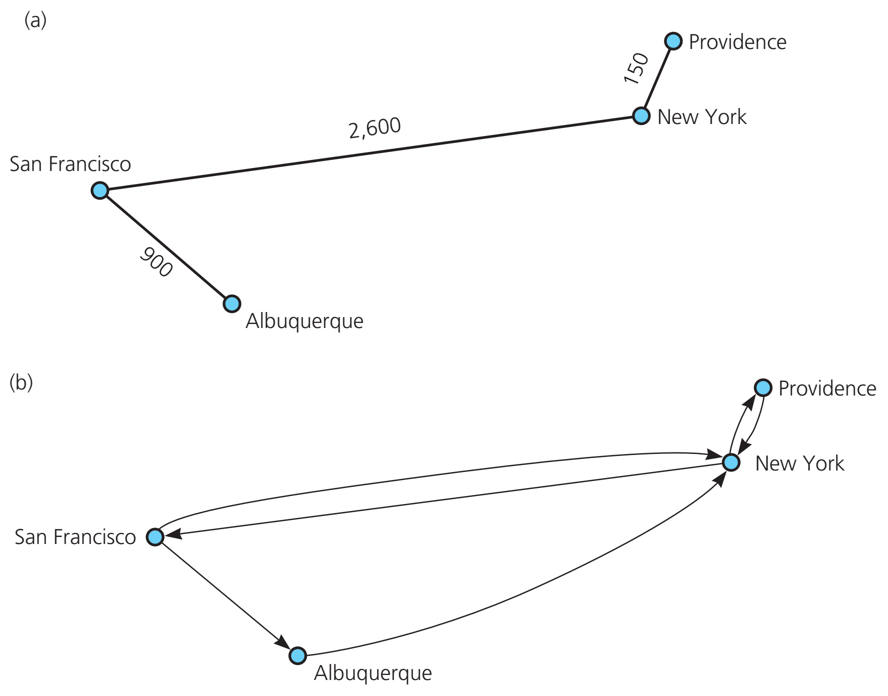


FIGURE 20-5 (a) A weighted graph; (b) a directed graph



directed edge. Although each distinct pair of vertices in an undirected graph has only one edge between them, a directed graph can have two edges between a pair of vertices, one in each direction. For example, the airline flight map in Figure 20-5b is a directed graph. There are flights in both directions between Providence and New York, but, although there is a flight from San Francisco to Albuquerque, there is no flight from Albuquerque to San Francisco. You can convert an undirected graph to a directed graph by replacing each edge with two directed edges that point in opposite directions.

In a directed graph, vertex *y* is adjacent to vertex *x* if there is a directed edge from *x* to *y*

The definitions just given for undirected graphs apply also to directed graphs, with changes that account for direction. For example, a **directed path** is a sequence of directed edges between two vertices, such as the directed path in Figure 20-5b that begins in Providence, goes to New York, and ends in San Francisco. However, the definition of adjacent vertices is not quite as obvious for a digraph. If there is a directed edge from vertex *x* to vertex *y*, then *y* is adjacent to *x*. (Alternatively, *y* is a **successor** of *x*, and *x* is a **predecessor** of *y*.) It does not necessarily follow, however, that *x* is adjacent to *y*. Thus, in Figure 20-5b, Albuquerque is adjacent to San Francisco, but San Francisco is not adjacent to Albuquerque.

20.2 Graphs as ADTs

Vertices can have values

You can treat graphs as abstract data types. Insertion and removal operations are somewhat different for graphs than for other ADTs that you have studied, in that they apply to either vertices or edges. You can define the ADT graph so that its vertices either do or do not contain values. A graph whose vertices do not contain values represents only the relationships among vertices. Such graphs are not unusual, because many problems have no need for vertex values. However, the following ADT graph operations do assume that the graph’s vertices contain values.

Note: ADT graph operations

- Test whether a graph is empty.
- Get the number of vertices in a graph.
- Get the number of edges in a graph.
- See whether an edge exists between two given vertices.
- Insert a vertex in a graph whose vertices have distinct values that differ from the new vertex's value.
- Insert an edge between two given vertices in a graph.
- Remove a particular vertex from a graph and any edges between the vertex and other vertices.
- Remove the edge between two given vertices in a graph.
- Retrieve from a graph the vertex that contains a given value.

Several variations of this ADT are possible. For example, if the graph is directed, you can replace occurrences of "edges" in the previous operations with "directed edges." You can also add traversal operations to the ADT. Graph-traversal algorithms are discussed in Section 20.3.

Listing 20-1 contains an interface that specifies in more detail the ADT operations for an undirected graph.

LISTING 20-1 A C++ interface for undirected, connected graphs

```
/** An interface for the ADT undirected, connected graph.
 @file GraphInterface.h */
#ifndef _GRAPH_INTERFACE
#define _GRAPH_INTERFACE

template<class LabelType>
class GraphInterface
{
public:
   /** Gets the number of vertices in this graph.
    @pre  None.
    @return  The number of vertices in the graph. */
   virtual int getNumVertices() const = 0;

   /** Gets the number of edges in this graph.
    @pre  None.
    @return  The number of edges in the graph. */
   virtual int getNumEdges() const = 0;

  /** Creates an undirected edge in this graph between two vertices
      that have the given labels. If such vertices do not exist, creates
      them and adds them to the graph before creating the edge.
    @param start  A label for the first vertex.
    @param end  A label for the second vertex.
    @param edgeWeight  The integer weight of the edge.
    @return  True if the edge is created, or false otherwise. */
   virtual bool add(LabelType start, LabelType end, int edgeWeight) = 0;

   /** Removes an edge from this graph. If a vertex has no other edges,
       it is removed from the graph since this is a connected graph.
```

(continues)

```
      @pre  None.
      @param start  A label for the first vertex.
      @param end  A label for the second vertex.
      @return  True if the edge is removed, or false otherwise. */
   virtual bool remove(LabelType start, LabelType end) = 0;

   /** Gets the weight of an edge in this graph.
      @return  The weight of the specified edge.
         If no such edge exists, returns a negative integer. */
   virtual int getEdgeWeight(LabelType start, LabelType end) const = 0;

   /** Performs a depth-first search of this graph beginning at the given
       vertex and calls a given function once for each vertex visited.
      @param start  A label for the first vertex.
      @param visit  A client-defined function that performs an operation on
        or with each visited vertex. */
   virtual void depthFirstTraversal(LabelType start, void visit(LabelType&)) = 0;

   /** Performs a breadth-first search of this graph beginning at the given
       vertex and calls a given function once for each vertex visited.
      @param start  A label for the first vertex.
      @param visit  A client-defined function that performs an operation on
        or with each visited vertex. */
   virtual void breadthFirstTraversal(LabelType start, void visit(LabelType&)) = 0;
}; // end GraphInterface
#endif
```

20.2.1 Implementing Graphs

The two most common implementations of a graph are the adjacency matrix and the adjacency list.

Adjacency matrix

An **adjacency matrix** for a graph with n vertices numbered 0, 1, . . ., $n - 1$ is an n by n array `matrix` such that `matrix[i][j]` is 1 (true) if there is an edge from vertex i to vertex j, and 0 (false) otherwise. Figure 20-6 shows a directed graph and its adjacency matrix. Notice that the diagonal entries `matrix[i][i]` are 0, although sometimes it can be useful to set these entries to 1. You should choose the value that is most convenient for your application.

When the graph is weighted, you can let `matrix[i][j]` be the weight that labels the edge from vertex i to vertex j, instead of simply 1, and let `matrix[i][j]` equal ∞ instead of 0 when there is no edge from vertex i to vertex j. For example, Figure 20-7 shows a weighted undirected graph and its adjacency matrix. Notice that the adjacency matrix for an undirected graph is symmetrical; that is, `matrix[i][j]` equals `matrix[j][i]`.

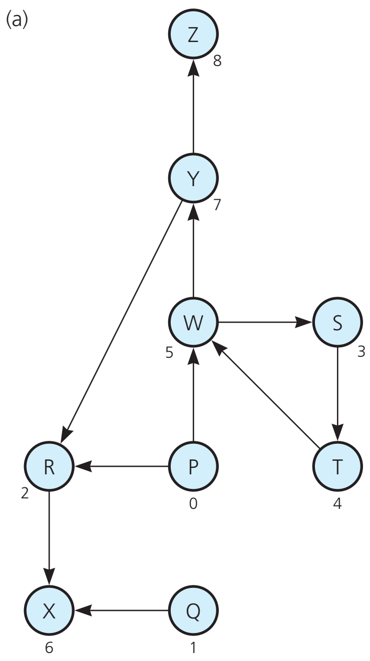
Our definition of an adjacency matrix does not mention the value, if any, in a vertex. If you need to associate values with vertices, you can use a second array, `values`, to represent the n vertex values. The array `values` is one-dimensional, and `values[i]` is the value in vertex i.

Adjacency list

An **adjacency list** for a graph with n vertices numbered 0, 1, . . ., $n - 1$ consists of n linked chains. The i^{th} linked chain has a node for vertex j if and only if the graph contains an edge from vertex i to vertex j. This node can contain the vertex j's value, if any. If the vertex has no value, the node needs to contain some indication of the vertex's identity. Figure 20-8 shows a directed graph and its adjacency list. You can see, for example, that vertex 0 (P) has edges to vertex 2 (R) and vertex 5 (W). Thus, the first linked chain in the adjacency chain contains nodes for R and W.

Figure 20-9 shows an undirected graph and its adjacency list. The adjacency list for an undirected graph treats each edge as if it were two directed edges in opposite directions. Thus, the edge

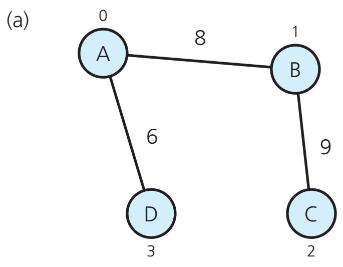
FIGURE 20-6 (a) A directed graph and (b) its adjacency matrix



(b)

		0	1	2	3	4	5	6	7	8
		P	Q	R	S	T	W	X	Y	Z
0	P	0	0	1	0	0	1	0	0	0
1	Q	0	0	0	0	0	0	1	0	0
2	R	0	0	0	0	0	0	1	0	0
3	S	0	0	0	0	1	0	0	0	0
4	T	0	0	0	0	0	1	0	0	0
5	W	0	0	0	1	0	0	0	1	0
6	X	0	0	0	0	0	0	0	0	0
7	Y	0	0	1	0	0	0	0	0	1
8	Z	0	0	0	0	0	0	0	0	0

FIGURE 20-7 (a) A weighted undirected graph and (b) its adjacency matrix



(b)

		0	1	2	3
		A	B	C	D
0	A	∞	8	∞	6
1	B	8	∞	9	∞
2	C	∞	9	∞	∞
3	D	6	∞	∞	∞

between *A* and *B* in Figure 20-9a appears as edges from *A* to *B* and from *B* to *A* in Figure 20-9b. The graph in 20-9a happens to be weighted; you can include the edge weights in the nodes of the adjacency list, as shown in Figure 20-9b.

Which of these two implementations of a graph—the adjacency matrix or the adjacency list—is better? The answer depends on how your particular application uses the graph. For example, the two most commonly performed graph operations are

Two common operations on graphs

1. Determine whether there is an edge from vertex *i* to vertex *j*
2. Find all vertices adjacent to a given vertex *i*

The adjacency matrix supports the first operation somewhat more efficiently than does the adjacency list. To determine whether there is an edge from *i* to *j* by using an adjacency matrix, you need

FIGURE 20-8 (a) A directed graph and (b) its adjacency list

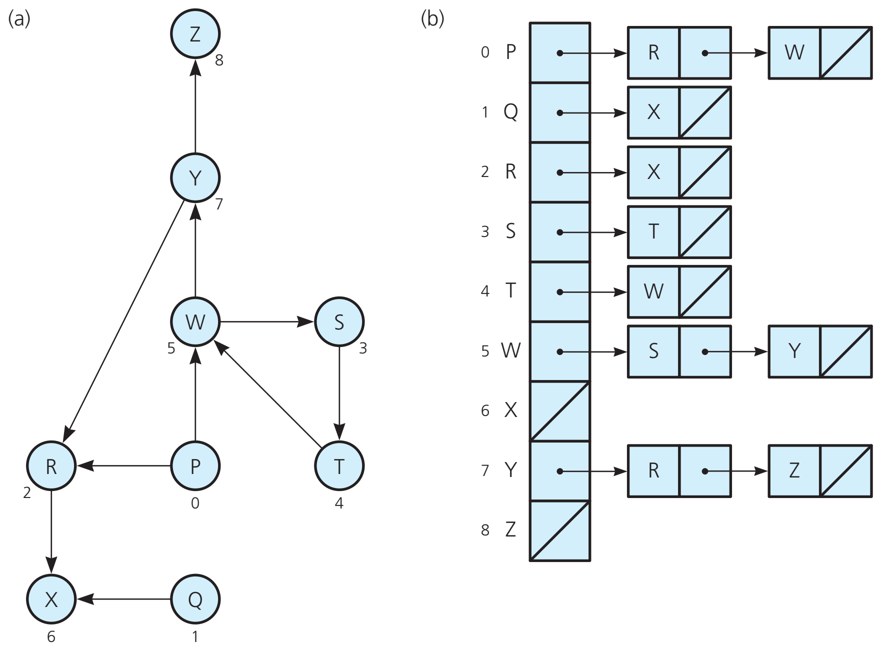
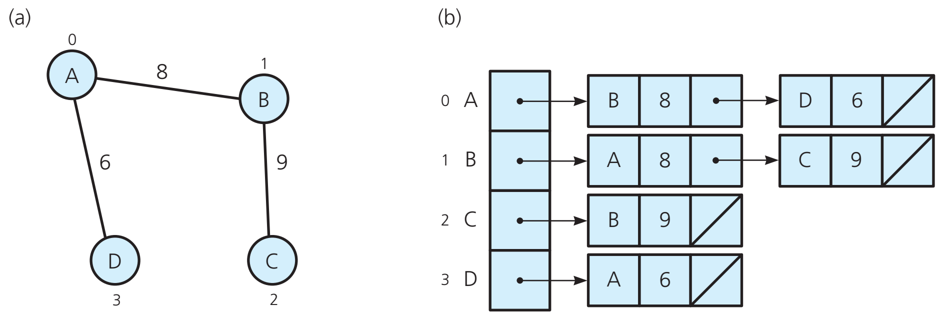


FIGURE 20-9 (a) A weighted undirected graph and (b) its adjacency list



An adjacency list often requires less space than an adjacency matrix

An adjacency matrix supports operation 1 more efficiently

only examine the value of `matrix[i][j]`. If you use an adjacency list, however, you must traverse the *i*th linked chain to determine whether a vertex corresponding to vertex *j* is present.

The second operation, on the other hand, is supported more efficiently by the adjacency list. To determine all vertices adjacent to a given vertex *i*, given the adjacency matrix, you must traverse the *i*th row of the array; however, given the adjacency list, you need only traverse the *i*th linked chain. For a graph with *n* vertices, the *i*th row of the adjacency matrix always has *n* entries, whereas the *i*th linked chain has only as many nodes as there are vertices adjacent to vertex *i*, a number typically far less than *n*.

Consider now the space requirements of the two implementations. On the surface it might appear that the adjacency matrix requires less memory than the adjacency list, because each entry in the matrix is simply an integer, whereas each list node contains both a value to identify the vertex and a pointer. The adjacency matrix, however, always has n^2 entries, whereas the number of nodes in an adjacency list equals the number of edges in a directed graph or twice that number for an undirected graph. Even though the adjacency list also has n head pointers, it often requires less storage than an adjacency matrix.

An adjacency list supports operation 2 more efficiently

Thus, when choosing a graph implementation for a particular application, you must consider such factors as what operations you will perform most frequently on the graph and the number of edges that the graph is likely to contain. For example, Chapters 5 and 6 presented the HPAir problem, which was to determine whether an airline provided a sequence of flights from an origin city to a destination city. The flight map for that problem is in fact a directed graph and appeared earlier in this chapter in Figure 20-8a. Figures 20-6b and 20-8b show, respectively, the adjacency matrix and adjacency list for this graph. Because the most frequent operation was to find all cities (vertices) adjacent to a given city (vertex), the adjacency list would be the more efficient implementation of the flight map. The adjacency list also requires less storage than the adjacency matrix, which you can demonstrate as an exercise.

20.3 Graph Traversals

VideoNote

Graph operations

The solution to the HPAir problem in Chapter 6 involved an exhaustive search of the graph in Figure 20-8a to determine a directed path from the origin vertex (city) to the destination vertex (city). The algorithm `searchS` started at a given vertex and traversed edges to other vertices until it either found the desired vertex or determined that no (directed) path existed between the two vertices.

What distinguishes `searchS` from a standard graph traversal is that `searchS` stops when it first encounters the designated destination vertex. A **graph-traversal** algorithm, on the other hand, will not stop until it has visited all of the vertices *that it can reach*. That is, a graph traversal that starts at vertex v will visit all vertices w for which there is a path between v and w. Unlike a tree traversal, which always visits *all of the nodes* in a tree, a graph traversal does not necessarily visit all of the vertices in the graph unless the graph is connected. In fact, a graph traversal visits every vertex in the graph if and only if the graph is connected, regardless of where the traversal starts. (See Exercise 18.) Thus, you can use a graph traversal to determine whether a graph is connected.

A graph traversal visits all of the vertices that it can reach

A graph traversal visits all vertices if and only if the graph is connected

If a graph is not connected, a graph traversal that begins at vertex v will visit only a subset of the graph's vertices. This subset is called the **connected component** containing v. You can determine all of the connected components of a graph by repeatedly starting a traversal at an unvisited vertex.

A connected component is the subset of vertices visited during a traversal that begins at a given vertex

If a graph contains a cycle, a graph-traversal algorithm can loop indefinitely. To prevent such a misfortune, the algorithm must mark each vertex during a visit and must never visit a vertex more than once.

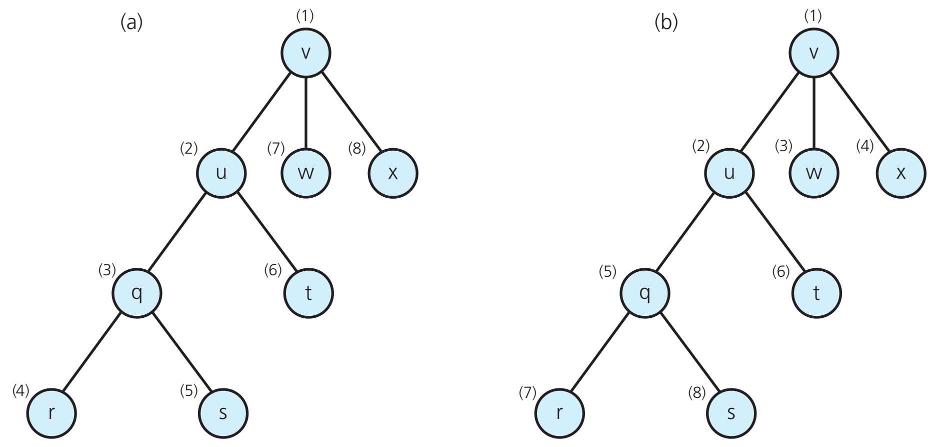
Two basic graph-traversal algorithms, which apply to either directed or undirected graphs, are presented next. These algorithms visit the vertices in different orders, but if they both start at the same vertex, they will visit the same set of vertices. Figure 20-10 shows the traversal order for the two algorithms when they begin at vertex v.

20.3.1 Depth-First Search

From a given vertex v, the **depth-first search (DFS)** strategy of graph traversal proceeds along a path from v as deeply into the graph as possible before backing up. That is, after visiting a vertex, a DFS visits, if possible, an unvisited adjacent vertex.

DFS traversal goes as far as possible from a vertex before backing up

FIGURE 20-10 Visitation order for (a) a depth-first search; (b) a breadth-first search



The DFS strategy has a simple recursive form:

Recursive DFS traversal algorithm

```
// Traverses a graph beginning at vertex v by using a
// depth-first search: Recursive version.
dfs(v: Vertex)

   Mark v as visited
   for (each unvisited vertex u adjacent to v)
      dfs(u)
```

Choose an order in which to visit adjacent vertices

The depth-first search algorithm does not completely specify the order in which it should visit the vertices adjacent to *v*. One possibility is to visit the vertices adjacent to *v* in sorted (that is, alphabetic or numerically increasing) order. This possibility is natural either when an adjacency matrix represents the graph or when the nodes in each linked chain of an adjacency list are linked in sorted order.

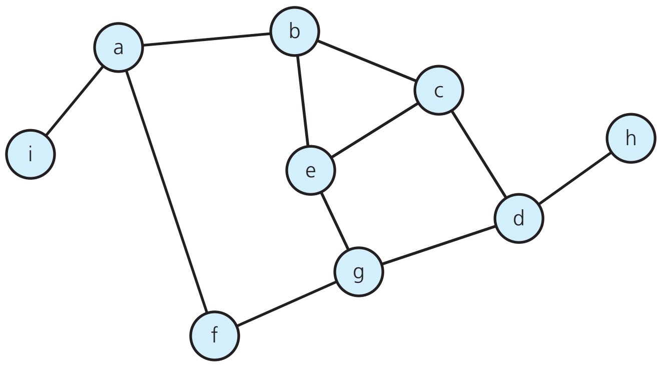
As Figure 20-10a illustrates, the DFS traversal algorithm marks and then visits each of the vertices *v*, *u*, *q*, and *r*. When the traversal reaches a vertex—such as *r*—that has no unvisited adjacent vertices, it backs up and visits, if possible, an unvisited adjacent vertex. Thus, the traversal backs up to *q* and then visits *s*. Continuing in this manner, the traversal visits vertices in the order given in the figure.

An iterative DFS traversal algorithm uses a stack

An iterative version of the DFS algorithm is also possible by using a stack:

```
// Traverses a graph beginning at vertex v by using a
// depth-first search: Iterative version.
dfs(v: Vertex)

   s= a new empty stack

   // Push v onto the stack and mark it
   s.push(v)
   Mark v as visited

   // Loop invariant: there is a path from vertex v at the
   // bottom of the stack s to the vertex at the top of s
   while (!s.isEmpty())
```

FIGURE 20-11 A connected graph with cycles



```
{
   if (no unvisited vertices are adjacent to the vertex on the top of the stack)
      s.pop()  // Backtrack

   else
   {
      Select an unvisited vertex u adjacent to the vertex on the top of the stack
      s.push(u)
      Mark u as visited
   }
}
```

The `dfs` algorithm is similar to `searchS` of Chapter 6, but the `while` statement in `searchS` terminates when the top of the stack is `destinationCity`.

For another example of a DFS traversal, consider the graph in Figure 20-11. Figure 20-12 shows the contents of the stack as the previous function `dfs` visits vertices in this graph, beginning at vertex *a*. Because the graph is connected, a DFS traversal will visit every vertex. In fact, the traversal visits the vertices in this order: *a, b, c, d, g, e, f, h, i*.

The vertex from which a depth-first traversal embarks is the vertex that it visited most recently. This *last visited, first explored* strategy is reflected both in the explicit stack of vertices that the iterative `dfs` uses and in the implicit stack of vertices that the recursive `dfs` generates with its recursive calls.

20.3.2 Breadth-First Search

BFS traversal visits all vertices adjacent to a vertex before going forward

After visiting a given vertex *v,* the **breadth-first search (BFS)** strategy of graph traversal visits every vertex adjacent to *v* that it can before visiting any other vertex. As Figure 20-10b illustrates, after marking and visiting *v*, the BFS traversal algorithm marks and then visits each of the vertices *u*, *w*, and *x*. Since no other vertices are adjacent to *v*, the BFS algorithm visits, if possible, all unvisited vertices adjacent to *u*. Thus, the traversal visits *q* and *t*. Continuing in this manner, the traversal visits vertices in the order given in the figure.

A BFS traversal will not embark from any of the vertices adjacent to *v* until it has visited all possible vertices adjacent to *v.* Whereas a DFS is a *last visited*, *first explored* strategy, a BFS is a *first visited*, *first explored* strategy. It is not surprising, then, that a breadth-first search uses a queue. An iterative version of this algorithm follows.

FIGURE 20-12 The results of a depth-first traversal, beginning at vertex *a*, of the graph in Figure 20-11

Node visited	Stack (bottom to top)
a	a
b	a b
c	a b c
d	a b c d
g	a b c d g
e	a b c d g e
(backtrack)	a b c d g
f	a b c d g f
(backtrack)	a b c d g
(backtrack)	a b c d
h	a b c d h
(backtrack)	a b c d
(backtrack)	a b c
(backtrack)	a b
(backtrack)	a
i	a i
(backtrack)	a
(backtrack)	*(empty)*

An iterative BFS traversal algorithm uses a queue

```
// Traverses a graph beginning at vertex v by using a
// breadth-first search: Iterative version.
bfs(v: Vertex)

   q = a new empty queue

   // Add v to queue and mark it
   q.enqueue(v)
   Mark v as visited

   while (!q.isEmpty())
   {
      q.dequeue(w)

      // Loop invariant: there is a path from vertex w to every vertex in the queue q
      for (each unvisited vertex u adjacent to w)
      {
         Mark u as visited
         q.enqueue(u)
      }
   }
```

Figure 20-13 shows the contents of the queue as `bfs` visits vertices in the graph in Figure 20-11, beginning at vertex *a*. In general, a breadth-first search will visit the same vertices as a depth-first search, but in a different order. In this example, the BFS traversal visits all of the vertices in this order: *a*, *b*, *f*, *i*, *c*, *e*, *g*, *d*, *h*.

A recursive version of BFS traversal is not as simple as the recursive version of DFS traversal. Exercise 19 at the end of this chapter asks you to think about why this is so.

FIGURE 20-13 The results of a breadth-first traversal, beginning at vertex *a*, of the graph in Figure 20-11

Node visited	Queue (front to back)
a	a
	(empty)
b	b
f	b f
i	b f i
	f i
c	f i c
e	f i c e
	i c e
g	i c e g
	c e g
	e g
d	e g d
	g d
	d
	(empty)
h	h
	(empty)

20.4 Applications of Graphs

There are many useful applications of graphs. This section surveys some of these common applications.

20.4.1 Topological Sorting

A directed graph without cycles, such as the one in Figure 20-14, has a natural order. For example, vertex *a* precedes *b,* which precedes *c*. Such a graph has significance in ordinary life. If the vertices represent academic courses, the graph represents the prerequisite structure for the courses. For example, course *a* is a prerequisite to course *b,* which is a prerequisite to both courses *c* and *e*. In what order should you take all seven courses so that you will satisfy all prerequisites? There is a linear order, called a **topological order**, of the vertices in a directed graph without cycles that answers this question. In a list of vertices in topological order, vertex *x* precedes vertex *y* if there is a directed edge from *x* to *y* in the graph.

FIGURE 20-14 A directed graph without cycles

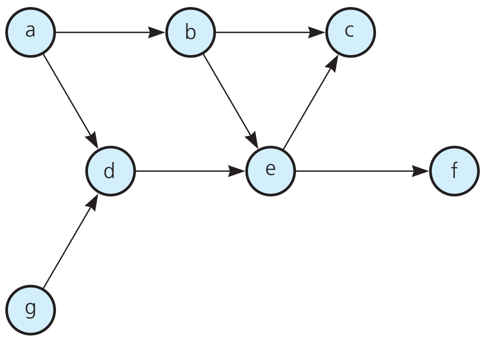
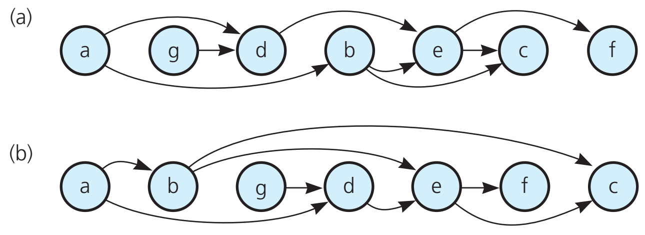


FIGURE 20-15 The graph in Figure 20-14 arranged according to the topological orders (a) *a, g, d, b, e, c, f* and (b) *a, b, g, d, e, f, c*



The vertices in a given graph may have several topological orders. For example, two topological orders for the vertices in Figure 20-14 are

a, g, d, b, e, c, f

and

a, b, g, d, e, f, c

If you arrange the vertices of a directed graph linearly and in a topological order, the edges will all point in one direction. Figure 20-15 shows two versions of the graph in Figure 20-14 that correspond to the two topological orders just given.

Arranging the vertices into a topological order is called **topological sorting**. There are several simple algorithms for finding a topological order. First, you could find a vertex that has no successor. You remove from the graph this vertex and all edges that lead to it, and add it to the beginning of a list of vertices. You add each subsequent vertex that has no successor to the beginning of the list. When the graph is empty, the list of vertices will be in topological order. The following pseudocode describes this algorithm:

A simple topological sorting algorithm

```
// Arranges the vertices in graph theGraph into a
// topological order and places them in list aList.
topSort1(theGraph: Graph, aList: List)

   n = number of vertices in theGraph
   for (step = 1 through n)
   {
      Select a vertex v that has no successors
      aList.insert(1, v)
      Remove from theGraph vertex v and its edges
   }
```

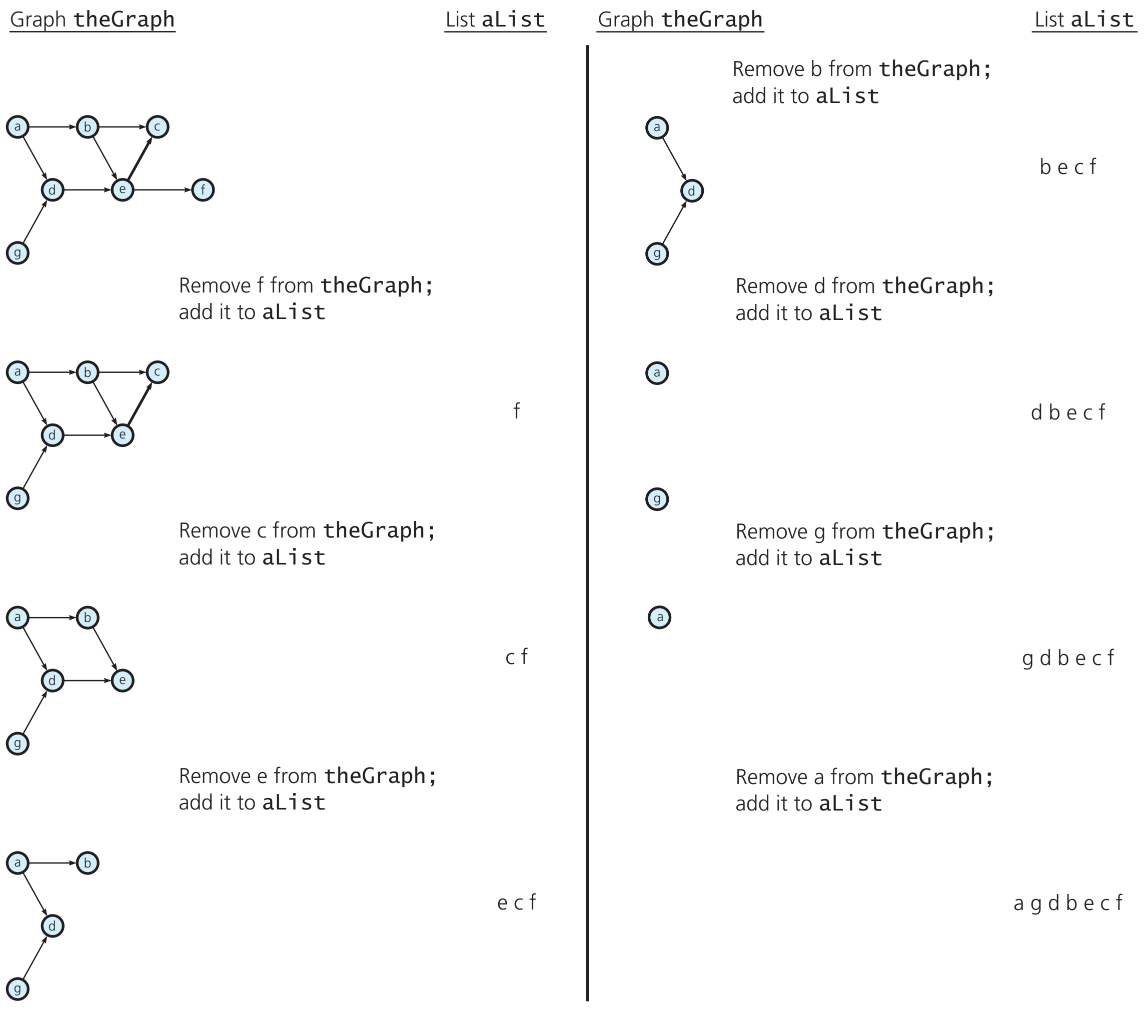
When the traversal ends, the list `aList` of vertices will be in topological order. Figure 20-16 traces this algorithm for the graph in Figure 20-14. The resulting topological order is the one that Figure 20-15a represents.

Another algorithm is a simple modification of the iterative depth-first search algorithm. First you push all vertices that have no predecessor onto a stack. Each time you pop a vertex from the stack, you add it to the beginning of a list of vertices. The pseudocode for this algorithm is

The DFS topological sorting algorithm

```
// Arranges the vertices in graph theGraph into a
// topological order and places them in list aList.
topSort2(theGraph: Graph, aList: List)
```

FIGURE 20-16 A trace of `topSort1` for the graph in Figure 20-14



Remove f from `theGraph;` add it to `aList` — f

Remove c from `theGraph;` add it to `aList` — c f

Remove e from `theGraph;` add it to `aList` — e c f

Remove b from `theGraph;` add it to `aList` — b e c f

Remove d from `theGraph;` add it to `aList` — d b e c f

Remove g from `theGraph;` add it to `aList` — g d b e c f

Remove a from `theGraph;` add it to `aList` — a g d b e c f

```
s = a new empty stack
for (all vertices v in the graph)
   if (v has no predecessors)
   {
      s.push(v)
      Mark v as visited
   }
while (!s.isEmpty())
{
   if (all vertices adjacent to the vertex on the top of the stack have been visited)
   {
      s.pop(v)
      aList.insert(1, v)
   }
   else
   {
```

```
            Select an unvisited vertex u adjacent to the vertex on the top of the stack
            s.push(u)
            Mark u as visited
        }
    }
```

When the traversal ends, the list `aList` of vertices will be in topological order. Figure 20-17 traces this algorithm for the graph in Figure 20-14. The resulting topological order is the one that Figure 20-15b represents.

FIGURE 20-17 A trace of `topSort2` for the graph in Figure 20-14

Action	Stack `s` (bottom to top)	List `aList` (beginning to end)
Push a	a	
Push g	a g	
Push d	a g d	
Push e	a g d e	c
Push c	a g d e c	c
Pop c, add c to `aList`	a g d e	f c
Push f	a g d e f	e f c
Pop f, add f to `aList`	a g d e	d e f c
Pop e, add e to `aList`	a g d	g d e f c
Pop d, add d to `aList`	a g	g d e f c
Pop g, add g to `aList`	a	b g d e f c
Push b	a b	a b g d e f c
Pop b, add b to `aList`	a	
Pop a, add a to `aList`	(empty)	

20.4.2 Spanning Trees

A tree is an undirected connected graph without cycles

A tree is a special kind of undirected graph, one that is connected but that has no cycles. Each vertex in the graph in Figure 20-3a could be the root of a different tree. Although all trees are graphs, not all graphs are trees. The nodes (vertices) of a tree have a hierarchical arrangement that is not required of all graphs.

A **spanning tree** of a connected undirected graph *G* is a subgraph of *G* that contains all of *G*'s vertices and enough of its edges to form a tree. For example, Figure 20-18 shows a spanning tree for the graph in Figure 20-11. The dashed lines in Figure 20-18 indicate edges that were omitted from the graph to form the tree. There may be several spanning trees for a given graph.

If you have a connected undirected graph with cycles and you remove edges until there are no cycles, you will obtain a spanning tree for the graph. It is relatively simple to determine whether a graph contains a cycle. One way to make this determination is based on the following observations about undirected graphs:

Observations about undirected graphs that enable you to detect a cycle

1. **A connected undirected graph that has *n* vertices must have at least *n* – 1 edges.** To establish this fact, recall that a connected graph has a path between every pair of vertices. Suppose that, beginning with *n* vertices, you choose one vertex and draw an edge between it and any other vertex. Next, draw an edge between this second vertex and any other unattached vertex. If you continue this process until you run out of unattached vertices, you will get a connected graph like the ones in Figure 20-19. If the graph has *n* vertices, it has *n* – 1 edges. In addition, if you remove an edge, the graph will not be connected.

FIGURE 20-18 A spanning tree for the graph in Figure 20-11

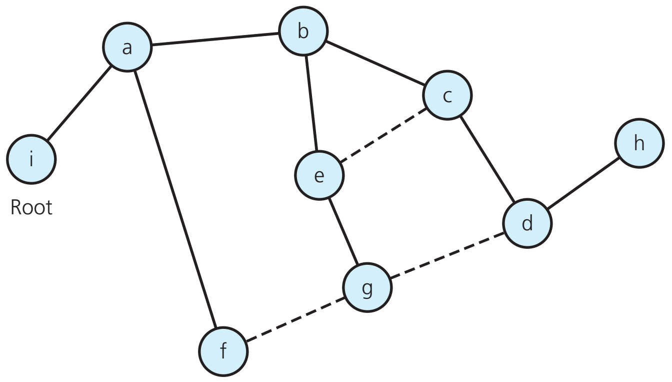
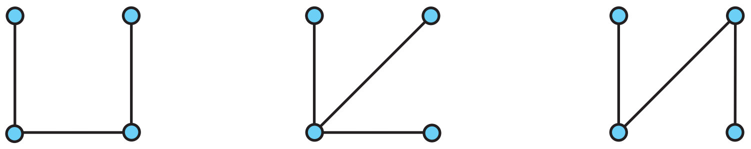


FIGURE 20-19 Connected graphs that each have four vertices and three edges



2. **A connected undirected graph that has *n* vertices and exactly *n* – 1 edges cannot contain a cycle.** To see this, begin with the previous observation: To be connected, a graph with *n* vertices must have at least *n* – 1 edges. If a connected graph did have a cycle, you could remove any edge along that cycle and still have a connected graph. Thus, if a connected graph with *n* vertices and *n* – 1 edges did contain a cycle, removing an edge along the cycle would leave you with a connected graph with only *n* – 2 edges, which is impossible according to observation 1.
3. **A connected undirected graph that has *n* vertices and more than *n* – 1 edges must contain at least one cycle.** For example, if you add an edge to any of the graphs in Figure 20-19, you will create a cycle within the graph. This fact is harder to establish and is left as an exercise. (See Exercise 17 at the end of this chapter.)

Thus, you can determine whether a connected graph contains a cycle simply by counting its vertices and edges.

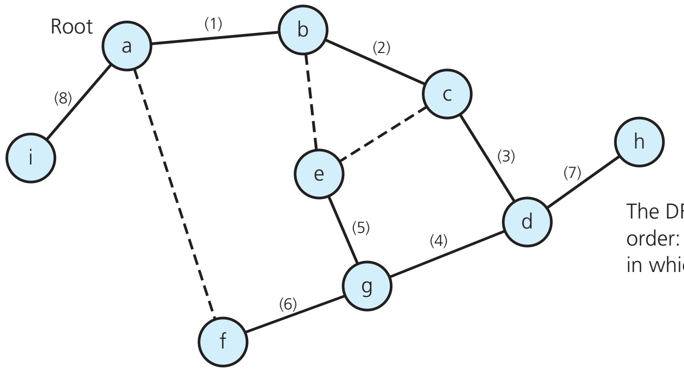
It follows, then, that a tree, which is a connected undirected graph without cycles, must connect its *n* nodes with *n* – 1 edges. Thus, to obtain the spanning tree of a connected graph of *n* vertices, you must remove edges along cycles until *n* – 1 edges are left.

Two algorithms for determining a spanning tree of a graph are based on the previous traversal algorithms and are presented next. In general, these algorithms will produce different spanning trees for any particular graph.

Count a graph's vertices and edges to determine whether it contains a cycle

The DFS spanning tree. One way to determine a spanning tree for a connected undirected graph is to traverse the graph's vertices by using a depth-first search. As you traverse the graph, mark the edges that you follow. After the traversal is complete, the graph's vertices and marked edges form a spanning

FIGURE 20-20 The DFS spanning tree rooted at vertex *a* for the graph in Figure 20-11



The DFS spanning tree algorithm visits vertices in this order: a, b, c, d, g, e, f, h, i. Numbers indicate the order in which the algorithm marks edges.

tree, which is called the **depth-first search (DFS) spanning tree**. (Alternatively, you can remove the unmarked edges from the graph to form the spanning tree.) Simple modifications to the previous iterative and recursive versions of `dfs` result in algorithms to create a DFS spanning tree. For example, the recursive algorithm follows:

DFS spanning tree algorithm

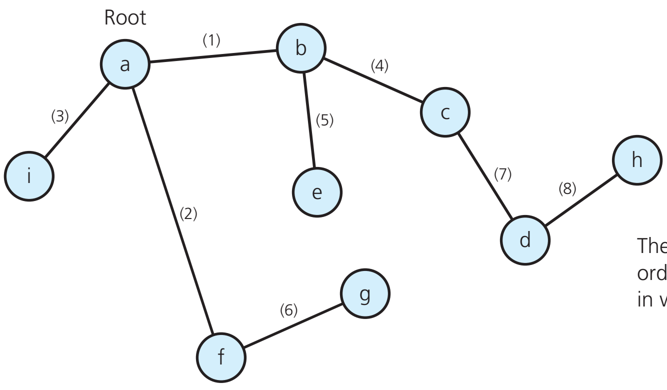
```
// Forms a spanning tree for a connected undirected graph
// beginning at vertex v by using depth-first search:
// Recursive version.
dfsTree(v: Vertex)

   Mark v as visited

   for (each unvisited vertex u adjacent to v)
   {
      Mark the edge from u to v
      dfsTree(u)
   }
```

When you apply this algorithm to the graph in Figure 20-11, you get the DFS spanning tree rooted at vertex *a* shown in Figure 20-20. The figure indicates the order in which the algorithm visits vertices and marks edges. You should reproduce these results by tracing the algorithm.

The BFS spanning tree. Another way to determine a spanning tree for a connected undirected graph is to traverse the graph's vertices by using a breadth-first search. As you traverse the graph, mark the edges that you follow. After the traversal is complete, the graph's vertices and marked edges form a spanning tree, which is called the **breadth-first search (BFS) spanning tree**. (Alternatively, you can remove the unmarked edges from the graph to form the spanning tree.) You can modify the previous iterative version of `bfs` by marking the edge between `w` and `u` before you add `u` to the queue. The result is the following iterative algorithm to create a BFS spanning tree.

```
// Forms a spanning tree for a connected undirected graph
// beginning at vertex v by using breadth-first search:
// Iterative version.
bfsTree(v: Vertex)

   q = a new empty queue

   // Add v to queue and mark it
```

FIGURE 20-21 The BFS spanning tree rooted at vertex *a* for the graph in Figure 20-11



The BFS spanning tree algorithm visits vertices in this order: a, b, f, i, c, e, g, d, h. Numbers indicate the order in which the algorithm marks edges.

```
q.enqueue(v)
Mark v as visited

while (!q.isEmpty())
{
   q.dequeue(w)

   // Loop invariant: there is a path from vertex w to
   // every vertex in the queue q
   for (each unvisited vertex u adjacent to w)
   {
      Mark u as visited
      Mark edge between w and u
      q.enqueue(u)
   }
}
```

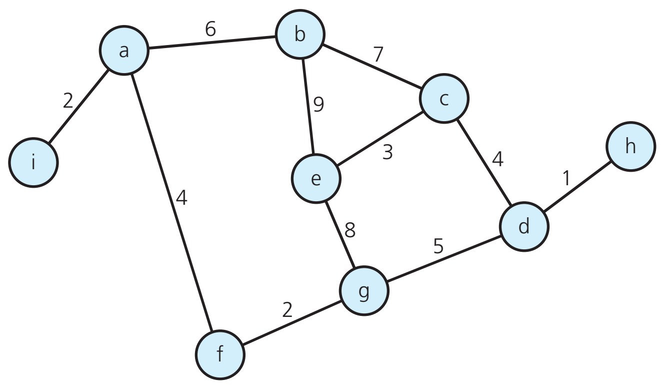
When you apply this algorithm to the graph in Figure 20-11, you get the BFS spanning tree rooted at vertex *a* shown in Figure 20-21. The figure indicates the order in which the algorithm visits vertices and marks edges. You should reproduce these results by tracing the algorithm.

20.4.3 Minimum Spanning Trees

Imagine that a developing country hires you to design its telephone system so that all the cities in the country can call one another. Obviously, one solution is to place telephone lines between every pair of cities. However, your engineering team has determined that due to the country's mountainous terrain, it is impossible to put lines between certain pairs of cities. The team's report contains the weighted undirected graph in Figure 20-22. The vertices in the graph represent *n* cities. An edge between two vertices indicates that it is feasible to place a telephone line between the cities that the vertices represent, and each edge's weight represents the installation cost of the telephone line. Note that if this graph is not connected, you will be unable to link all of the cities with a network of telephone lines. The graph in Figure 20-22 is connected, however, making the problem feasible.

If you install a telephone line between each pair of cities that is connected by an edge in the graph, you will certainly solve the problem. However, this solution may be too costly. From observation 1 in

FIGURE 20-22 A weighted, connected, undirected graph



the previous section, you know that $n - 1$ is the minimum number of edges necessary for a graph of n vertices to be connected. Thus, $n - 1$ is the minimum number of lines that can connect n cities.

If the cost of installing each line is the same, the problem is reduced to one of finding any spanning tree of the graph. The total installation cost—that is, the **cost of the spanning tree**—is the sum of the costs of the edges in the spanning tree. However, as the graph in Figure 20-22 shows, the cost of installing each line varies. Because there may be more than one spanning tree, and because the cost of different trees may vary, you need to solve the problem by selecting a spanning tree with the least cost; that is, you must select a spanning tree for which the sum of the edge weights (costs) is minimal. Such a tree is called the **minimum spanning tree**, and it need not be unique. Although there may be several minimum spanning trees for a particular graph, their costs are equal.

A minimum spanning tree of a connected undirected graph has a minimal edge-weight sum

One simple algorithm, called Prim's algorithm, finds a minimum spanning tree that begins at any vertex. Initially, the tree contains only the starting vertex. At each stage, the algorithm selects a least-cost edge from among those that begin with a vertex in the tree and end with a vertex not in the tree. The latter vertex and least-cost edge are then added to the tree. The following pseudocode describes this algorithm:

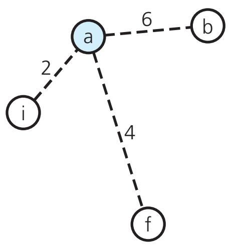
Minimum spanning tree algorithm

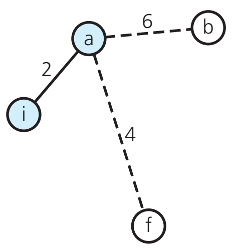
```
// Determines a minimum spanning tree for a weighted,
// connected, undirected graph whose weights are
// nonnegative, beginning with any vertex v.
primsAlgorithm(v: Vertex)

   Mark vertex v as visited and include it in the minimum spanning tree
   while (there are unvisited vertices)
   {
      Find the least-cost edge (v, u) from a visited vertex v to some unvisited vertex u
      Mark u as visited
      Add the vertex u and the edge (v, u) to the minimum spanning tree
   }
```

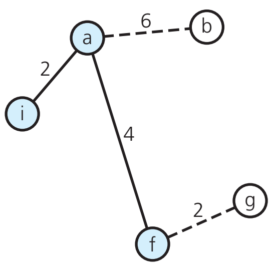
Figure 20-23 traces `primsAlgorithm` for the graph in Figure 20-22, beginning at vertex *a*. Edges added to the tree appear as solid lines, while edges under consideration appear as dashed lines.

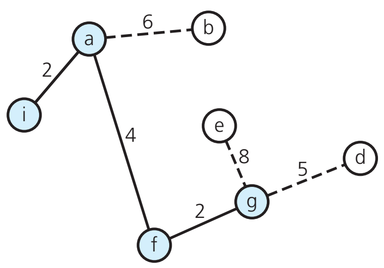
It is not obvious that the spanning tree that `primsAlgorithm` determines will be minimal. However, the proof that `primsAlgorithm` is correct is beyond the scope of this book.

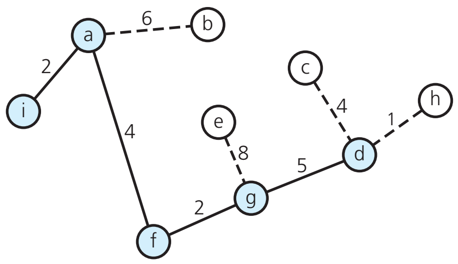
FIGURE 20-23 A trace of `primsAlgorithm` for the graph in Figure 20-22, beginning at vertex `a`

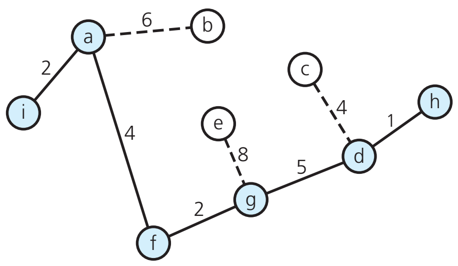

(a) Mark a, consider edges from a

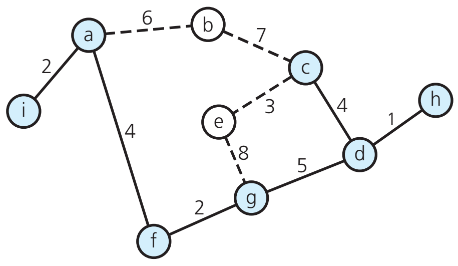

(b) Mark i, include edge (a, i)

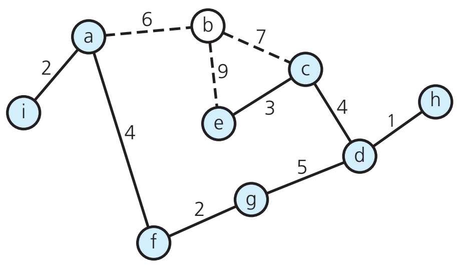

(c) Mark f, include edge (a, f)

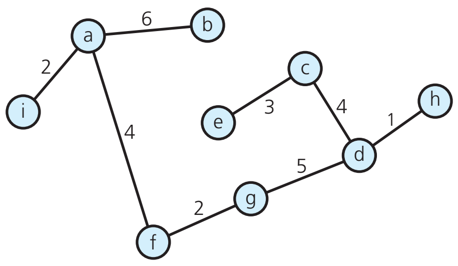

(d) Mark g, include edge (f, g)


(e) Mark d, include edge (g, d)


(f) Mark h, include edge (d, h)


(g) Mark c, include edge (d, c)


(h) Mark e, include edge (c, e)


(i) Mark b, include edge (a, b)

20.4.4 Shortest Paths

Consider once again a map of airline routes. A weighted directed graph can represent this map: The vertices are cities, and the edges indicate existing flights between cities. The edge weights represent the mileage between cities (vertices); as such, the weights are not negative. For example, you could combine the two graphs in Figure 20-5 to get such a weighted directed graph.

The shortest path between two vertices in a weighted graph has the smallest edge-weight sum

Often for weighted directed graphs you need to know the shortest path between two particular vertices. The **shortest path** between two given vertices in a weighted graph is the path that has the smallest sum of its edge weights. Although we use the term "shortest," realize that the weights could be a measure other than distance, such as the cost of each flight in dollars or the duration of each flight in hours. The sum of the weights of the edges of a path is called the path's **length** or **weight** or **cost**.

For example, the shortest path from vertex 0 to vertex 1 in the graph in Figure 20-24a is not the edge between 0 and 1—its cost is 8—but rather the path from 0 to 4 to 2 to 1, with a cost of 7. For convenience, the starting vertex, or origin, is labeled 0 and the other vertices are labeled from 1 to $n - 1$. Notice the graph's adjacency matrix in Figure 20-24b.

Finding the shortest paths between vertex 0 and all other vertices

The following algorithm, which is attributed to E. Dijkstra, actually determines the shortest paths between a given origin and *all* other vertices. The algorithm uses a set *vertexSet* of selected vertices and an array *weight,* where *weight*[*v*] is the weight of the shortest (cheapest) path from vertex 0 to vertex *v* that passes through vertices in *vertexSet*.

If *v* is in *vertexSet,* the shortest path involves only vertices in *vertexSet*. However, if *v* is not in *vertexSet,* then *v* is the only vertex along the path that is not in *vertexSet.* That is, the path ends with an edge from a vertex in *vertexSet* to *v*.

Initially, *vertexSet* contains only vertex 0, and *weight* contains the weights of the single-edge paths from vertex 0 to all other vertices. That is, *weight*[*v*] equals *matrix*[0][*v*] for all *v*, where *matrix* is the adjacency matrix. Thus, initially *weight* is the first row of *matrix*.

After this initialization step, you find a vertex *v* that is not in *vertexSet* and that minimizes *weight*[*v*]. You add *v* to *vertexSet*. For all (unselected) vertices *u* not in *vertexSet*, you check the values *weight*[*u*] to ensure that they are indeed minimums. That is, can you reduce *weight*[*u*]—the weight of a path from vertex 0 to vertex *u*—by passing through the newly selected vertex *v*?

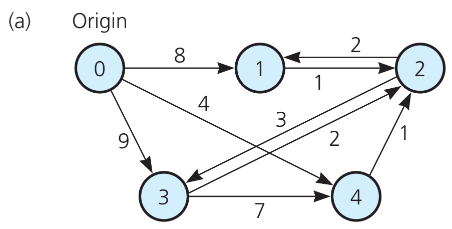
To make this determination, break the path from 0 to *u* into two pieces and find their weights as follows:

weight[*v*] = weight of the shortest path from 0 to *v*
matrix[*v*][*u*] = weight of the edge from *v* to *u*

Then compare *weight*[*u*] with *weight*[*v*] + *matrix*[*v*][*u*] and let

weight[*u*] = the smaller of the values *weight*[*u*] and *weight*[*v*] + *matrix*[*v*][*u*]

FIGURE 20-24 (a) A weighted directed graph and (b) its adjacency matrix



(b)

	0	1	2	3	4
0	∞	8	∞	9	4
1	∞	∞	1	∞	∞
2	∞	2	∞	3	∞
3	∞	∞	2	∞	7
4	∞	∞	1	∞	∞

The pseudocode for Dijkstra's shortest-path algorithm is as follows:

The shortest-path algorithm

```
// Finds the minimum-cost paths between an origin vertex
// (vertex 0) and all other vertices in a weighted directed
// graph theGraph; theGraph's weights are nonnegative.
shortestPath(theGraph: Graph, weight: WeightArray)

   // Step 1: initialization
   Create a set vertexSet that contains only vertex 0
   n = number of vertices in theGraph
   for (v = 0 through n - 1)
      weight[v] = matrix[0][v]

   // Steps 2 through n
   // Invariant: For v not in vertexSet, weight[v] is the
   // smallest weight of all paths from 0 to v that pass
   // through only vertices in vertexSet before reaching
   // v. For v in vertexSet, weight[v] is the smallest
   // weight of all paths from 0 to v (including paths
   // outside vertexSet), and the shortest path
   // from 0 to v lies entirely in vertexSet.
   for (step = 2 through n)
   {
      Find the smallest weight[v] such that v is not in vertexSet
      Add v to vertexSet

      // Check weight[u] for all u not in vertexSet
      for (all vertices u not in vertexSet)
         if (weight[u] > weight[v] + matrix[v][u])
            weight[u] = weight[v] + matrix[v][u]
    }
```

The loop invariant states that once a vertex *v* is placed in *vertexSet, weight*[*v*] is the weight of the absolutely shortest path from 0 to *v* and will not change.

Figure 20-25 traces the algorithm for the graph in Figure 20-24a. The algorithm takes the following steps:

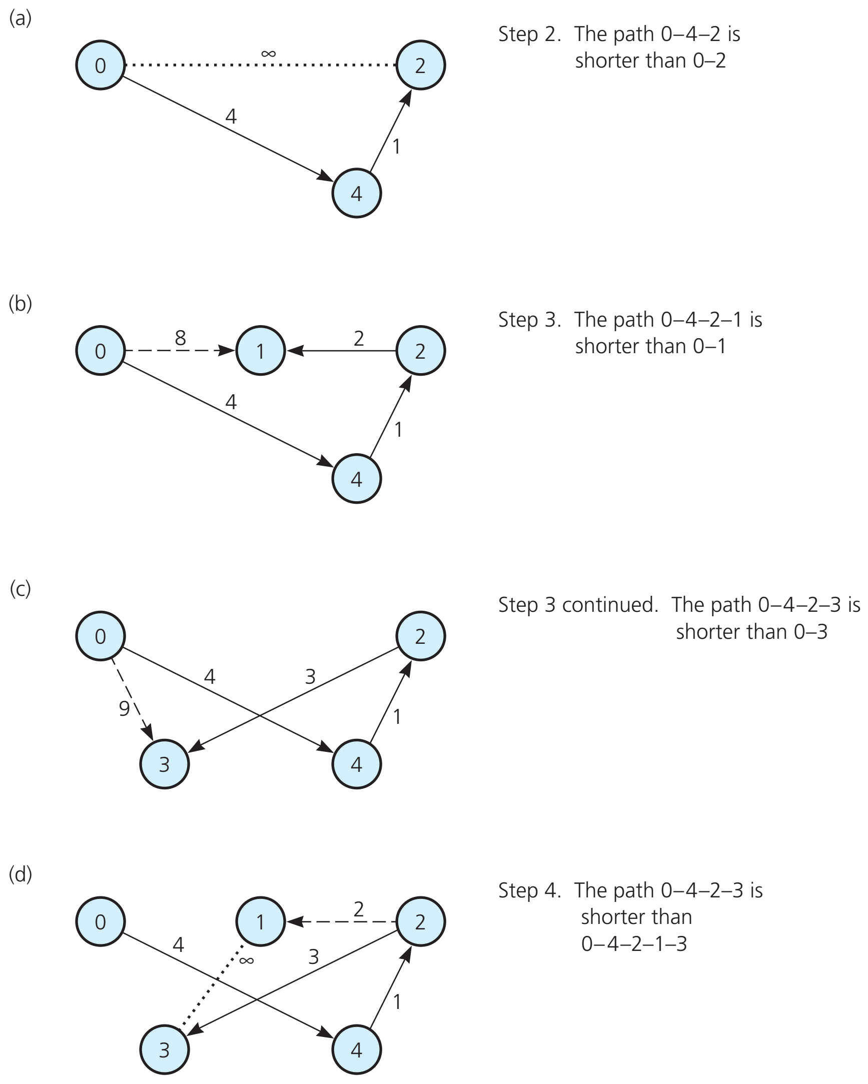
Step 1. *vertexSet* initially contains vertex 0, and *weight* is initially the first row of the graph's adjacency matrix, shown in Figure 20-24b.

Step 2. *weight*[4] = 4 is the smallest value in *weight,* ignoring *weight*[0] because 0 is in *vertexSet*. Thus, *v* = 4, so add 4 to *vertexSet*. For vertices not in *vertexSet*—that is, for *u* = 1, 2, and 3—check whether it is shorter to go from 0 to 4 and then along an edge to *u* instead of directly from 0 to *u* along an edge. For vertices 1 and 3, it is not shorter to

FIGURE 20-25 A trace of the shortest-path algorithm applied to the graph in Figure 20-24a

			weight				
Step	v	vertexSet	[0]	[1]	[2]	[3]	[4]
1	–	0	0	8	∞	9	4
2	4	0, 4	0	8	5	9	4
3	2	0, 4, 2	0	7	5	8	4
4	1	0, 4, 2, 1	0	7	5	8	4
5	3	0, 4, 2, 1, 3	0	7	5	8	4

FIGURE 20-26 Checking *weight*[u] by examining the graph: (a) *weight*[2] in step 2; (b) *weight*[1] in step 3; (c) *weight*[3] in step 3; (d) *weight*[3] in step 4


(a)
Step 2. The path 0–4–2 is shorter than 0–2
(b)
Step 3. The path 0–4–2–1 is shorter than 0–1
(c)
Step 3 continued. The path 0–4–2–3 is shorter than 0–3
(d)
Step 4. The path 0–4–2–3 is shorter than 0–4–2–1–3

include vertex 4 in the path. However, for vertex 2 notice that *weight*[2] = ∞ > *weight*[4] + *matrix*[4][2] = 4 + 1 = 5. Therefore, replace *weight*[2] with 5. You can also verify this conclusion by examining the graph directly, as Figure 20-26a shows.

Step 3. *weight*[2] = 5 is the smallest value in *weight,* ignoring *weight*[0] and *weight*[4] because 0 and 4 are in *vertexSet*. Thus, *v* = 2, so add 2 to *vertexSet*. For vertices not in *vertex-Set*—that is, for *u* = 1 and 3—check whether it is shorter to go from 0 to 2 and then along an edge to *u* instead of directly from 0 to *u* along an edge. (See Figures 20-26b and 20-26c.)

Notice that

weight[1] = 8 > *weight*[2] + *matrix*[2][1] = 5 + 2 = 7. Therefore, replace *weight*[1] with 7.

weight[3] = 9 > *weight*[2] + *matrix*[2][3] = 5 + 3 = 8. Therefore, replace *weight*[3] with 8.

Step 4. *weight*[1] = 7 is the smallest value in *weight,* ignoring *weight*[0], *weight*[2], and *weight*[4] because 0, 2, and 4 are in *vertexSet.* Thus, $v = 1$, so add 1 to *vertexSet.* For vertex 3, which is the only vertex not in *vertexSet,* notice that *weight*[3] = 8 < *weight*[1] + *matrix*[1][3] = 7 + ∞, as Figure 20-26d shows. Therefore, leave *weight*[3] as it is.

Step 5. The only remaining vertex not in *vertexSet* is 3, so add it to *vertexSet* and stop.

The final values in *weight* are the weights of the shortest paths. These values appear in the last line of Figure 20-25. For example, the shortest path from vertex 0 to vertex 1 has a cost of *weight*[1], which is 7. This result agrees with our earlier observation about Figure 20-24. We saw then that the shortest path is from 0 to 4 to 2 to 1. Also, the shortest path from vertex 0 to vertex 2 has a cost of *weight*[2], which is 5. This path is from 0 to 4 to 2.

The weights in *weight* are the smallest possible, as long as the algorithm's loop invariant is true. The proof that the loop invariant is true is by induction on `step`, and is left as a difficult exercise. (See Exercise 20.)

20.4.5 Circuits

A **circuit** is simply another name for a type of cycle that is common in the statement of certain problems. Recall that a cycle in a graph is a path that begins and ends at the same vertex. Typical circuits either visit every vertex once or visit every edge once.

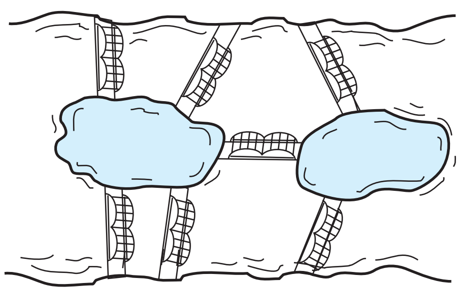
Probably the first application of graphs occurred in the early 1700s when Euler proposed a bridge problem. Two islands in a river are joined to each other and to the river banks by several bridges, as Figure 20-27a illustrates. The bridges correspond to the edges in the multigraph in Figure 20-27b, and the land masses correspond to the vertices. The problem asked whether you can begin at a vertex *v,* pass through every edge exactly once, and terminate at *v*. Euler demonstrated that no solution exists for this particular configuration of edges and vertices.

For simplicity, we will consider an undirected graph rather than a multigraph. A path in an undirected graph that begins at a vertex *v,* passes through every edge in the graph exactly once, and terminates at *v* is called an **Euler circuit**. Euler showed that an Euler circuit exists if and only if each vertex touches an even number of edges. Intuitively, if you arrive at a vertex along one edge, you must be able to leave the vertex along another edge. If you cannot, you will not be able to reach all of the vertices.

An Euler circuit begins at a vertex *v*, passes through every edge exactly once, and terminates at *v*

FIGURE 20-27 (a) Euler's bridge problem and (b) its multigraph representation

(a)



(b)

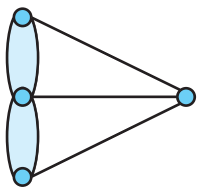


FIGURE 20-28 Pencil and paper drawings

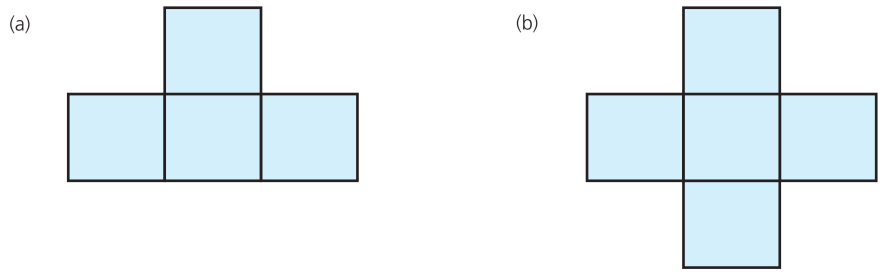
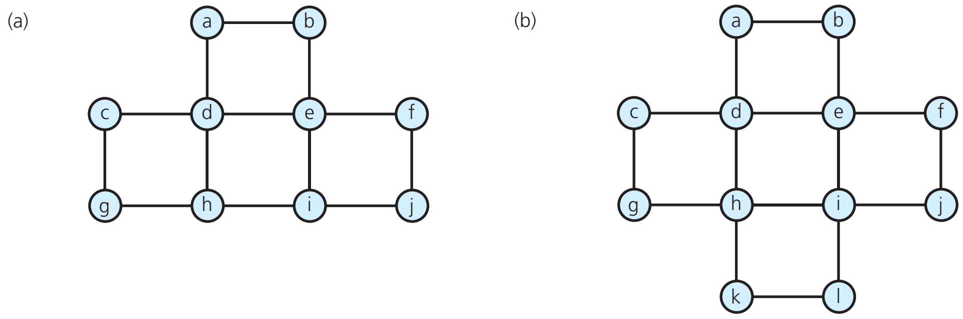


FIGURE 20-29 Connected undirected graphs based on the drawings in Figure 20-28



Finding an Euler circuit is like drawing each of the diagrams in Figure 20-28 without lifting your pencil or redrawing a line, and ending at your starting point. No solution is possible for Figure 20-28a, but you should be able to find one easily for Figure 20-28b. Figure 20-29 contains undirected graphs based on Figure 20-28. In Figure 20-29a, vertices *h* and *i* each touch an odd number of edges (three), so no Euler circuit is possible. On the other hand, each vertex in Figure 20-29b touches an even number of edges, making an Euler circuit feasible. Notice also that the graphs are connected. If a graph is not connected, a path through *all* of the vertices would not be possible.

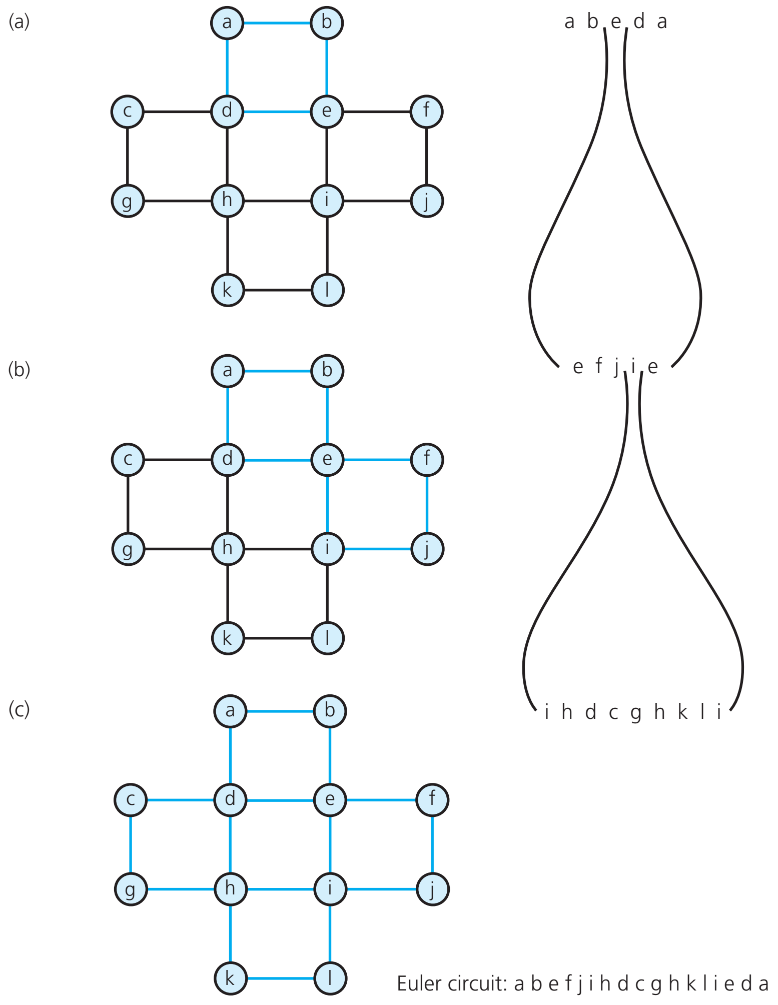
Let's find an Euler circuit for the graph in Figure 20-29b, starting arbitrarily at vertex *a*. The strategy uses a depth-first search that marks edges instead of vertices as they are traversed. Recall that a depth-first search traverses a path from *a* as deeply into the graph as possible. By marking edges instead of vertices, you will return to the starting vertex; that is, you will find a cycle. In this example, the cycle is *a*, *b*, *e*, *d*, *a* if we visit the vertices in alphabetical order, as Figure 20-30a shows. Clearly this is not the desired circuit, because we have not visited every edge. We are not finished, however.

To continue, find the first vertex along the cycle *a*, *b*, *e*, *d*, *a* that touches an unvisited edge. In our example, the desired vertex is *e*. Apply our modified depth-first search, beginning with this vertex. The resulting cycle is *e*, *f*, *j*, *i*, *e*. Next you join this cycle with the one you found previously. That is, when you reach *e* in the first cycle, you travel along the second cycle before continuing in the first cycle. The resulting path is *a*, *b*, *e*, *f*, *j*, *i*, *e*, *d*, *a*, as Figure 20-30b shows.

The first vertex along our combined cycle that touches an unvisited edge is *i*. Beginning at *i*, our algorithm determines the cycle *i*, *h*, *d*, *c*, *g*, *h*, *k*, *l*, *i*. Joining this to our combined cycle results in the Euler circuit *a*, *b*, *e*, *f*, *j*, *i*, *h*, *d*, *c*, *g*, *h*, *k*, *l*, *i*, *e*, *d*, *a*. (See Figure 20-30c.)

FIGURE 20-30 The steps to determine an Euler circuit for the graph in Figure 20-29b

Loop invariant



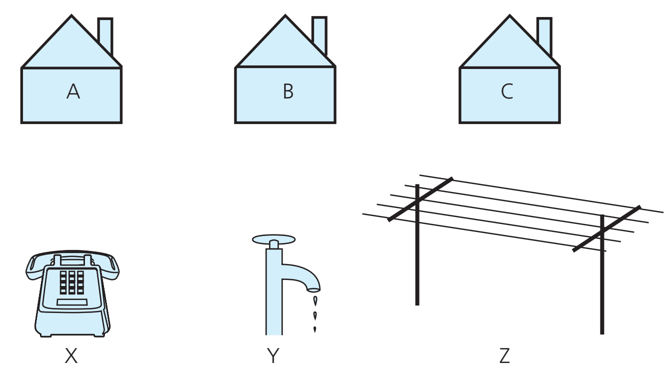
20.4.6 Some Difficult Problems

The next three applications of graphs have solutions that are beyond the scope of this book.

A Hamilton circuit begins at a vertex *v*, passes through every vertex exactly once, and terminates at *v*

The traveling salesperson problem. A **Hamilton circuit** is a path that begins at a vertex *v*, passes through every vertex in the graph exactly once, and terminates at *v*. Determining whether an arbitrary graph contains a Hamilton circuit can be difficult. A well-known variation of this problem—the traveling salesperson problem—involves a weighted graph that represents a road map. Each edge has an associated cost, such as the mileage between cities or the time required to drive from one city to the next. The salesperson must begin at an origin city, visit every other city exactly once, and return to the origin city. However, the circuit traveled must be the least expensive.

FIGURE 20-31 The three utilities problem



Unfortunately for this traveler, solving the problem is no easy task. Although a solution does exist, it is quite slow, and no better solution is known.

The three utilities problem. Imagine three houses *A, B,* and *C* and three utilities *X, Y,* and *Z* (such as telephone, water, and electricity), as Figure 20-31 illustrates. If the houses and the utilities are vertices in a graph, is it possible to connect each house to each utility with edges that do not cross one another? The answer to this question is no.

A planar graph can be drawn so that no two edges cross

A graph is **planar** if you can draw it in a plane in at least one way so that no two edges cross. The generalization of the three utilities problem determines whether a given graph is planar. Making this determination has many important applications. For example, a graph can represent an electronic circuit where the vertices represent components and the edges represent the connections between components. Is it possible to design the circuit so that the connections do not cross? The solutions to these problems are also beyond the scope of this book.

The four-color problem. Given a planar graph, can you color the vertices so that no adjacent vertices have the same color, if you use at most four colors? For example, the graph in Figure 20-11 is planar because none of its edges cross. You can solve the coloring problem for this graph by using only three colors. Color vertices *a, c, g,* and *h* red, color vertices *b, d, f,* and *i* blue, and color vertex *e* green.

The answer to our question is yes, but it is difficult to prove. In fact, this problem was posed more than a century before it was solved in the 1970s with the use of a computer.

? CHECK POINT

Question 1 Describe the graphs in Figure 20-32. For example, are they directed? Connected? Complete? Weighted?

Question 2 Use the depth-first strategy and the breadth-first strategy to traverse the graph in Figure 20-32a, beginning with vertex 0. List the vertices in the order in which each traversal visits them.

Question 3 Write the adjacency matrix for the graph in Figure 20-32a.

Question 4 Add an edge to the directed graph in Figure 20-14 that runs from vertex *d* to vertex *b*. Write all possible topological orders for the vertices in this new graph.

Question 5 Is it possible for a connected undirected graph with five vertices and four edges to contain a simple cycle? Explain.

Question 6 Draw the DFS spanning tree whose root is vertex 0 for the graph in Figure 20-33.

Question 7 Draw the minimum spanning tree whose root is vertex 0 for the graph in Figure 20-33.

Question 8 What are the shortest paths from vertex 0 to each vertex of the graph in Figure 20-24a? (Note the weights of these paths in Figure 20-25.)

FIGURE 20-32 Graphs for Checkpoint Questions 1, 2, and 3

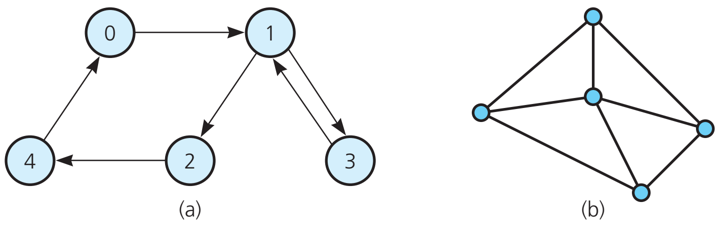
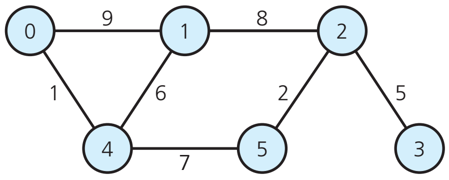


FIGURE 20-33 A graph for Checkpoint Questions 6 and 7 and for Exercises 1 and 4



Summary

1. The two most common implementations of a graph are the adjacency matrix and the adjacency list. Each has its relative advantages and disadvantages. The choice should depend on the needs of the given application.
2. Graph searching is an important application of stacks and queues. Depth-first search (DFS) is a graph-traversal algorithm that uses a stack to keep track of the sequence of visited vertices. It goes as deep into the graph as it can before backtracking. Breadth-first search (BFS) uses a queue to keep track of the sequence of visited vertices. It visits all possible adjacent vertices before traversing further into the graph.
3. Topological sorting produces a linear order of the vertices in a directed graph without cycles. Vertex *x* precedes vertex *y* if there is a directed edge from *x* to *y* in the graph.
4. Trees are connected undirected graphs without cycles. A spanning tree of a connected undirected graph is a subgraph that contains all of the graph's vertices and enough of its edges to form a tree. DFS and BFS traversals produce DFS and BFS spanning trees.
5. A minimum spanning tree for a weighted undirected graph is a spanning tree whose edge-weight sum is minimal. Although a particular graph can have several minimum spanning trees, their edge-weight sums will be the same.
6. The shortest path between two vertices in a weighted directed graph is the path that has the smallest sum of its edge weights.

7. An Euler circuit in an undirected graph is a cycle that begins at vertex *v*, passes through every edge in the graph exactly once, and terminates at *v*.

8. A Hamilton circuit in an undirected graph is a cycle that begins at vertex *v*, passes through every vertex in the graph exactly once, and terminates at *v*.

Exercises

When given a choice of vertices to visit, the traversals in the following exercises should visit vertices in sorted order.

1. Give the adjacency matrix and adjacency list for

a. The weighted graph in Figure 20-33
b. The directed graph in Figure 20-34

2. Show that the adjacency list in Figure 20-8b requires less memory than the adjacency matrix in Figure 20-6b.

3. Consider Figure 20-35 and answer the following:

a. Will the adjacency matrix be symmetrical?
b. Provide the adjacency matrix.
c. Provide the adjacency list.

4. Use both the depth-first strategy and the breadth-first strategy to traverse the graph in Figure 20-33, beginning with vertex 0, and the graph in Figure 20-36, beginning with vertex *a*. List the vertices in the order in which each traversal visits them.

5. By modifying the DFS traversal algorithm, write pseudocode for an algorithm that determines whether a graph contains a cycle.

6. Using the topological sorting algorithm `topSort1`, as given in this chapter, write the topological order of the vertices for each graph in Figure 20-37.

7. Trace the DFS topological sorting algorithm `topSort2`, and indicate the resulting topological order of the vertices for each graph in Figure 20-37.

8. Revise the topological sorting algorithm `topSort1` by removing predecessors instead of successors. Trace the new algorithm for each graph in Figure 20-37.

9. Trace the DFS and BFS spanning tree algorithms, beginning with vertex *a* of the graph in Figure 20-11, and show that the spanning trees are the trees in Figures 20-20 and 20-21, respectively.

FIGURE 20-34 A graph for Exercise 1

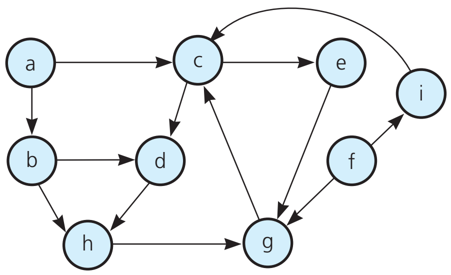


FIGURE 20-35 A graph for Exercise 3

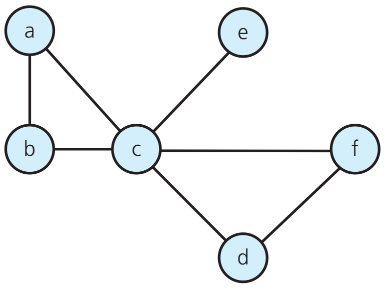
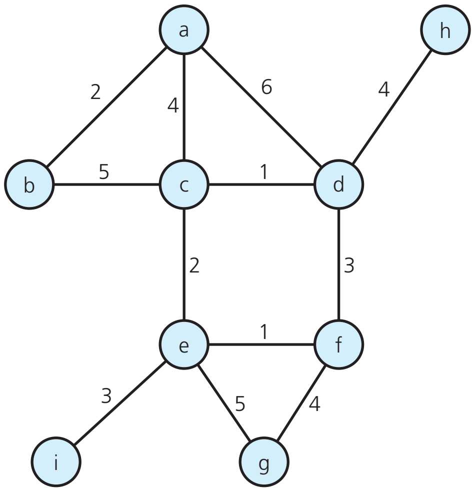


FIGURE 20-36 A graph for Exercises 4 and 10



10. Draw the DFS and BFS spanning trees rooted at *a* for the graph in Figure 20-36. Then draw the minimum spanning tree rooted at *a* for this graph.

11. For the graph in Figure 20-38,

a. Draw all the possible spanning trees.
b. Draw the minimum spanning tree.

12. Write pseudocode for an iterative algorithm that determines a DFS spanning tree for an undirected graph. Base your algorithm on the traversal algorithm `dfs`.

13. Draw the minimum spanning tree for the graph in Figure 20-22 when you start with

a. Vertex *g*
b. Vertex *c*

***14.** Trace the shortest-path algorithm for the graph in Figure 20-39, letting vertex 0 be the origin.

***15.** Implement the shortest-path algorithm in C++. How can you modify this algorithm so that any vertex can be the origin?

FIGURE 20-37 Graphs for Exercises 6, 7, and 8

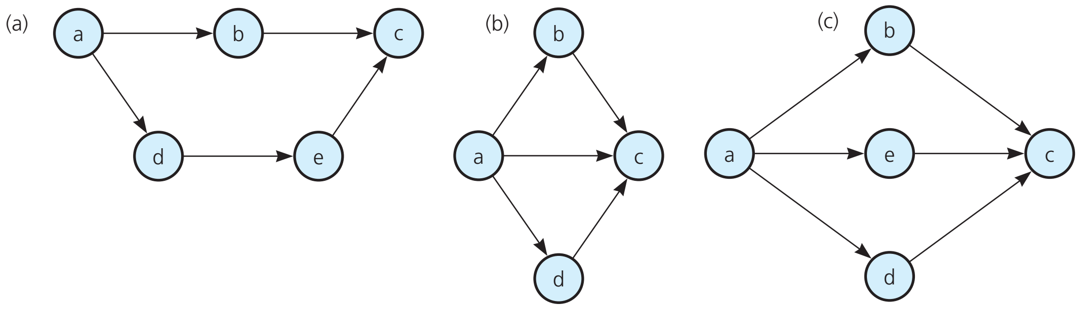


FIGURE 20-38 A graph for Exercise 11

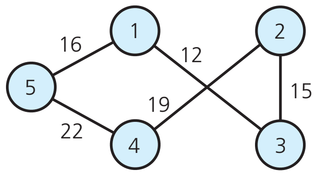
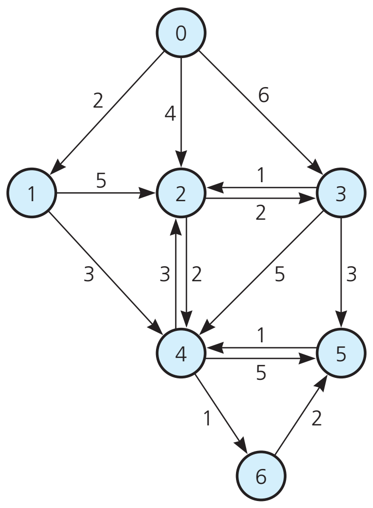


FIGURE 20-39 A graph for Exercise 14



***16.** Determine an Euler circuit for the graph in Figure 20-40. Why is one possible?

***17.** Prove that a connected undirected graph with n vertices and more than $n - 1$ edges must contain at least one simple cycle. (See observation 3 in Section 20.4.2.)

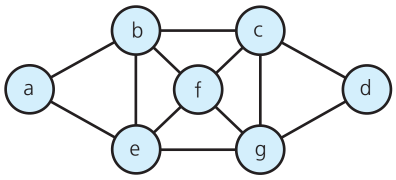
***18.** Prove that a graph-traversal algorithm visits every vertex in the graph if and only if the graph is connected, regardless of where the traversal starts.

***19.** Although the DFS traversal algorithm has a simple recursive form, a recursive BFS traversal algorithm is not straightforward.

- **a.** Explain why this statement is true.
- **b.** Write the pseudocode for a recursive version of the BFS traversal algorithm.

***20.** Prove that the loop invariant of Dijkstra's shortest-path algorithm is true by using a proof by induction on `step`.

FIGURE 20-40 A graph for Exercise 16



Programming Problems

1. Write a C++ class derived from `GraphInterface`, as given in Listing 20-1. Use an adjacency matrix to represent the graph.

2. Repeat the previous programming problem, but represent the graph using an adjacency list instead of an adjacency matrix.

3. Repeat Programming Problems 1 and 2, but allow the graph to be either weighted or unweighted and either directed or undirected.

4. Extend Programming Problem 3 by adding ADT operations such as `isConnected` and `hasCycle`. Also, include operations that perform a topological sort for a directed graph without cycles, determine the DFS and BFS spanning trees for a connected graph, and determine a minimum spanning tree for a connected undirected graph.

5. The HPAir problem was the subject of Programming Problems 11 through 14 of Chapter 6. Revise these problems by implementing the ADT flight map as a derived class of the graph class that you wrote for Programming Problem 3.