# PART IV

Advanced Data Structures

11

Graphs

Graphs provide the ultimate in data structure flexibility. Graphs can model both real-world systems and abstract problems, so they are used in hundreds of applications. Here is a small sampling of the range of problems that graphs are routinely applied to.

1. Modeling connectivity in computer and communications networks.
2. Representing a map as a set of locations with distances between locations; used to compute shortest routes between locations.
3. Modeling flow capacities in transportation networks.
4. Finding a path from a starting condition to a goal condition; for example, in artificial intelligence problem solving.
5. Modeling computer algorithms, showing transitions from one program state to another.
6. Finding an acceptable order for finishing subtasks in a complex activity, such as constructing large buildings.
7. Modeling relationships such as family trees, business or military organizations, and scientific taxonomies.

We begin in Section 11.1 with some basic graph terminology and then define two fundamental representations for graphs, the adjacency matrix and adjacency list. Section 11.2 presents a graph ADT and simple implementations based on the adjacency matrix and adjacency list. Section 11.3 presents the two most commonly used graph traversal algorithms, called depth-first and breadth-first search, with application to topological sorting. Section 11.4 presents algorithms for solving some problems related to finding shortest routes in a graph. Finally, Section 11.5 presents algorithms for finding the minimum-cost spanning tree, useful for determining lowest-cost connectivity in a network. Besides being useful and interesting in their own right, these algorithms illustrate the use of some data structures presented in earlier chapters.
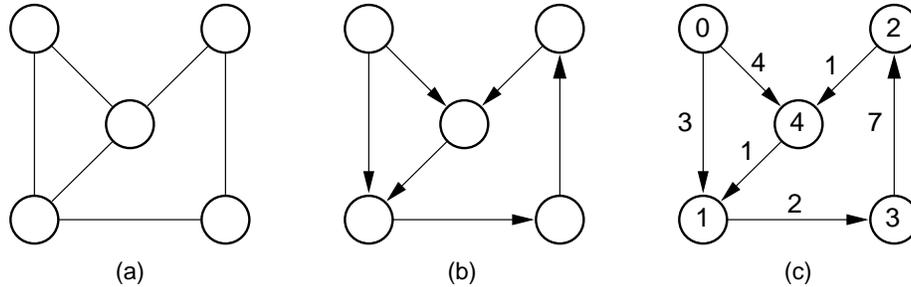
Figure 11.1 Examples of graphs and terminology. (a) A graph. (b) A directed graph (digraph). (c) A labeled (directed) graph with weights associated with the edges. In this example, there is a simple path from Vertex 0 to Vertex 3 containing Vertices 0, 1, and 3. Vertices 0, 1, 3, 2, 4, and 1 also form a path, but not a simple path because Vertex 1 appears twice. Vertices 1, 3, 2, 4, and 1 form a simple cycle.

11.1 Terminology and Representations

A graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ consists of a set of vertices $\mathbf{V}$ and a set of edges $\mathbf{E}$, such that each edge in $\mathbf{E}$ is a connection between a pair of vertices in $\mathbf{V}$.[1] The number of vertices is written $|\mathbf{V}|$, and the number of edges is written $|\mathbf{E}|$. $|\mathbf{E}|$ can range from zero to a maximum of $|\mathbf{V}|^2 - |\mathbf{V}|$. A graph with relatively few edges is called **sparse**, while a graph with many edges is called **dense**. A graph containing all possible edges is said to be **complete**.

A graph with edges directed from one vertex to another (as in Figure 11.1(b)) is called a **directed graph** or **digraph**. A graph whose edges are not directed is called an **undirected graph** (as illustrated by Figure 11.1(a)). A graph with labels associated with its vertices (as in Figure 11.1(c)) is called a **labeled graph**. Two vertices are **adjacent** if they are joined by an edge. Such vertices are also called **neighbors**. An edge connecting Vertices U and V is written (U, V). Such an edge is said to be **incident** on Vertices U and V. Associated with each edge may be a cost or **weight**. Graphs whose edges have weights (as in Figure 11.1(c)) are said to be **weighted**.

A sequence of vertices v_1, v_2, ..., v_n forms a **path** of length $n - 1$ if there exist edges from v_i to v_{i+1} for $1 \leq i < n$. A path is **simple** if all vertices on the path are distinct. The **length** of a path is the number of edges it contains. A **cycle** is a path of length three or more that connects some vertex v_1 to itself. A cycle is **simple** if the path is simple, except for the first and last vertices being the same.

---

[1]Some graph applications require that a given pair of vertices can have multiple or parallel edges connecting them, or that a vertex can have an edge to itself. However, the applications discussed in this book do not require either of these special cases, so for simplicity we will assume that they cannot occur.
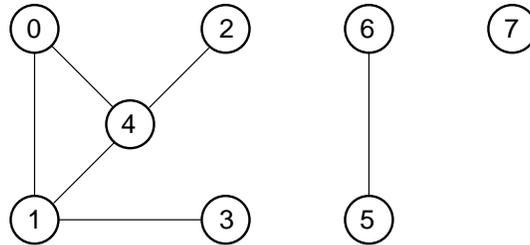
Figure 11.2 An undirected graph with three connected components. Vertices 0, 1, 2, 3, and 4 form one connected component. Vertices 5 and 6 form a second connected component. Vertex 7 by itself forms a third connected component.

A **subgraph S** is formed from graph **G** by selecting a subset $\mathbf{V}_s$ of **G**'s vertices and a subset $\mathbf{E}_s$ of **G**'s edges such that for every edge E in $\mathbf{E}_s$, both of E's vertices are in $\mathbf{V}_s$.

An undirected graph is **connected** if there is at least one path from any vertex to any other. The maximally connected subgraphs of an undirected graph are called **connected components**. For example, Figure 11.2 shows an undirected graph with three connected components.

A graph without cycles is called **acyclic**. Thus, a directed graph without cycles is called a **directed acyclic graph** or DAG.

A **free tree** is a connected, undirected graph with no simple cycles. An equivalent definition is that a free tree is connected and has $|\mathbf{V}| - 1$ edges.

There are two commonly used methods for representing graphs. The **adjacency matrix** is illustrated by Figure 11.3(b). The adjacency matrix for a graph is a $|\mathbf{V}| \times |\mathbf{V}|$ array. Assume that $|\mathbf{V}| = n$ and that the vertices are labeled from v_0 through v_{n-1}. Row i of the adjacency matrix contains entries for Vertex v_i. Column j in row i is marked if there is an edge from v_i to v_j and is not marked otherwise. Thus, the adjacency matrix requires one bit at each position. Alternatively, if we wish to associate a number with each edge, such as the weight or distance between two vertices, then each matrix position must store that number. In either case, the space requirements for the adjacency matrix are $\Theta(|\mathbf{V}|^2)$.

The second common representation for graphs is the **adjacency list**, illustrated by Figure 11.3(c). The adjacency list is an array of linked lists. The array is $|\mathbf{V}|$ items long, with position i storing a pointer to the linked list of edges for Vertex v_i. This linked list represents the edges by the vertices that are adjacent to Vertex v_i. The adjacency list is therefore a generalization of the "list of children" representation for trees described in Section 6.3.1.

Example 11.1 The entry for Vertex 0 in Figure 11.3(c) stores 1 and 4 because there are two edges in the graph leaving Vertex 0, with one going
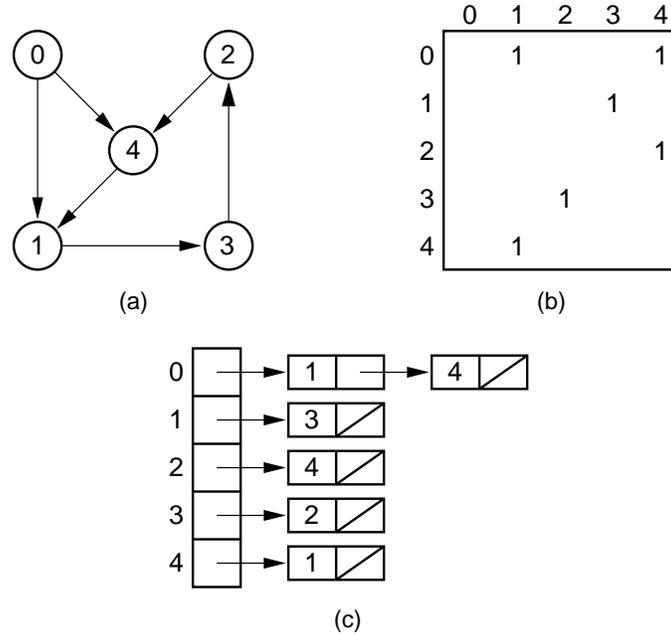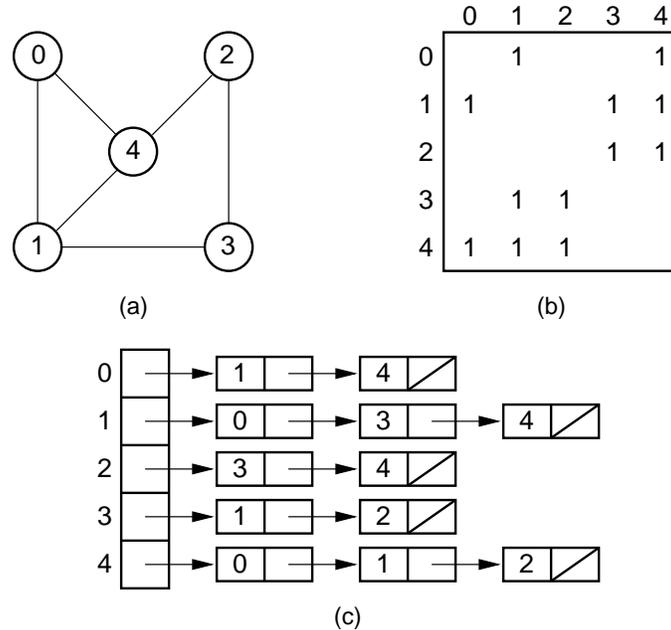
Figure 11.3 Two graph representations. (a) A directed graph. (b) The adjacency matrix for the graph of (a). (c) The adjacency list for the graph of (a).

to Vertex 1 and one going to Vertex 4. The list for Vertex 2 stores an entry for Vertex 4 because there is an edge from Vertex 2 to Vertex 4, but no entry for Vertex 3 because this edge comes into Vertex 2 rather than going out.

The storage requirements for the adjacency list depend on both the number of edges and the number of vertices in the graph. There must be an array entry for each vertex (even if the vertex is not adjacent to any other vertex and thus has no elements on its linked list), and each edge must appear on one of the lists. Thus, the cost is $\Theta(|\mathbf{V}| + |\mathbf{E}|)$.

Both the adjacency matrix and the adjacency list can be used to store directed or undirected graphs. Each edge of an undirected graph connecting Vertices U and V is represented by two directed edges: one from U to V and one from V to U. Figure 11.4 illustrates the use of the adjacency matrix and the adjacency list for undirected graphs.

Which graph representation is more space efficient depends on the number of edges in the graph. The adjacency list stores information only for those edges that actually appear in the graph, while the adjacency matrix requires space for each potential edge, whether it exists or not. However, the adjacency matrix requires no overhead for pointers, which can be a substantial cost, especially if the only

Figure 11.4 Using the graph representations for undirected graphs. (a) An undirected graph. (b) The adjacency matrix for the graph of (a). (c) The adjacency list for the graph of (a).

information stored for an edge is one bit to indicate its existence. As the graph becomes denser, the adjacency matrix becomes relatively more space efficient. Sparse graphs are likely to have their adjacency list representation be more space efficient.

Example 11.2 Assume that a vertex index requires two bytes, a pointer requires four bytes, and an edge weight requires two bytes. Then the adjacency matrix for the graph of Figure 11.3 requires $2|\mathbf{V}^2| = 50$ bytes while the adjacency list requires $4|\mathbf{V}| + 6|\mathbf{E}| = 56$ bytes. For the graph of Figure 11.4, the adjacency matrix requires the same space as before, while the adjacency list requires $4|\mathbf{V}| + 6|\mathbf{E}| = 92$ bytes (because there are now 12 edges instead of 6).

The adjacency matrix often requires a higher asymptotic cost for an algorithm than would result if the adjacency list were used. The reason is that it is common for a graph algorithm to visit each neighbor of each vertex. Using the adjacency list, only the actual edges connecting a vertex to its neighbors are examined. However, the adjacency matrix must look at each of its $|\mathbf{V}|$ potential edges, yielding a total cost of $\Theta(|\mathbf{V}^2|)$ time when the algorithm might otherwise require only $\Theta(|\mathbf{V}|+|\mathbf{E}|)$

time. This is a considerable disadvantage when the graph is sparse, but not when the graph is closer to full.

11.2 Graph Implementations

We next turn to the problem of implementing a general-purpose graph class. Figure 11.5 shows an abstract class defining an ADT for graphs. Vertices are defined by an integer index value. In other words, there is a Vertex 0, Vertex 1, and so on. We can assume that a graph application stores any additional information of interest about a given vertex elsewhere, such as a name or application-dependent value. Note that this ADT is not implemented using a generic, because it is the **Graph** class users' responsibility to maintain information related to the vertices themselves. The **Graph** class need have no knowledge of the type or content of the information associated with a vertex, only the index number for that vertex.

Abstract class **Graph** has methods to return the number of vertices and edges (methods **n** and **e**, respectively). Function **weight** returns the weight of a given edge, with that edge identified by its two incident vertices. For example, calling **weight(0, 4)** on the graph of Figure 11.1 (c) would return 4. If no such edge exists, the weight is defined to be 0. So calling **weight(0, 2)** on the graph of Figure 11.1 (c) would return 0.

Functions **setEdge** and **delEdge** set the weight of an edge and remove an edge from the graph, respectively. Again, an edge is identified by its two incident vertices. **setEdge** does not permit the user to set the weight to be 0, because this value is used to indicate a non-existent edge, nor are negative edge weights permitted. Functions **getMark** and **setMark** get and set, respectively, a requested value in the **Mark** array (described below) for Vertex *V*.

Nearly every graph algorithm presented in this chapter will require visits to all neighbors of a given vertex. Two methods are provided to support this. They work in a manner similar to linked list access functions. Function **first** takes as input a vertex *V*, and returns the edge to the first neighbor for *V* (we assume the neighbor list is sorted by vertex number). Function **next** takes as input Vertices *V1* and *V2* and returns the index for the vertex forming the next edge with *V1* after *V2* on *V1*'s edge list. Function **next** will return a value of $n = |\mathbf{V}|$ once the end of the edge list for *V1* has been reached. The following line appears in many graph algorithms:

```
for (w = G=>first(v); w < G->n(); w = G->next(v,w))
```

This **for** loop gets the first neighbor of **v**, then works through the remaining neighbors of **v** until a value equal to **G->n()** is returned, signaling that all neighbors of **v** have been visited. For example, **first(1)** in Figure 11.4 would return 0. **next(1, 0)** would return 3. **next(0, 3)** would return 4. **next(1, 4)** would return 5, which is not a vertex in the graph.

```
/** Graph ADT */
public interface Graph {            // Graph class ADT

  /** Initialize the graph
      @param n The number of vertices */
  public void Init(int n);

  /** @return The number of vertices */
  public int n();

  /** @return The current number of edges */
  public int e();

  /** @return v's first neighbor */
  public int first(int v);

 /** @return v's next neighbor after w */
  public int next(int v, int w);

  /** Set the weight for an edge
      @param i,j The vertices
      @param wght Edge weight */
  public void setEdge(int i, int j, int wght);

  /** Delete an edge
      @param i,j The vertices */
  public void delEdge(int i, int j);

  /** Determine if an edge is in the graph
      @param i,j The vertices
      @return true if edge i,j has non-zero weight */
  public boolean isEdge(int i, int j);


  /** @return The weight of edge i,j, or zero
      @param i,j The vertices */
  public int weight(int i, int j);

  /** Set the mark value for a vertex
      @param v The vertex
      @param val The value to set */
  public void setMark(int v, int val);

  /** Get the mark value for a vertex
      @param v The vertex
      @return The value of the mark */
  public int getMark(int v);
}
```

Figure 11.5 A graph ADT. This ADT assumes that the number of vertices is fixed when the graph is created, but that edges can be added and removed. It also supports a mark array to aid graph traversal algorithms.

It is reasonably straightforward to implement our graph and edge ADTs using either the adjacency list or adjacency matrix. The sample implementations presented here do not address the issue of how the graph is actually created. The user of these implementations must add functionality for this purpose, perhaps reading the graph description from a file. The graph can be built up by using the **setEdge** function provided by the ADT.

Figure 11.6 shows an implementation for the adjacency matrix. Array **Mark** stores the information manipulated by the **setMark** and **getMark** functions. The edge matrix is implemented as an integer array of size $n \times n$ for a graph of n vertices. Position (i, j) in the matrix stores the weight for edge (i, j) if it exists. A weight of zero for edge (i, j) is used to indicate that no edge connects Vertices i and j.

Given a vertex *V*, function **first** locates the position in **matrix** of the first edge (if any) of *V* by beginning with edge (*V*, 0) and scanning through row *V* until an edge is found. If no edge is incident on *V*, then **first** returns n.

Function **next** locates the edge following edge (i, j) (if any) by continuing down the row of Vertex i starting at position $j + 1$, looking for an edge. If no such edge exists, **next** returns n. Functions **setEdge** and **delEdge** adjust the appropriate value in the array. Function **weight** returns the value stored in the appropriate position in the array.

Figure 11.7 presents an implementation of the adjacency list representation for graphs. Its main data structure is an array of linked lists, one linked list for each vertex. These linked lists store objects of type **Edge**, which merely stores the index for the vertex pointed to by the edge, along with the weight of the edge.

```
/** Edge class for Adjacency List graph representation */
class Edge {
  private int vert, wt;

  public Edge(int v, int w) // Constructor
    { vert = v;   wt = w; }

  public int vertex() { return vert; }
  public int weight() { return wt; }
}
```

Implementation for **Graphl** member functions is straightforward in principle, with the key functions being **setEdge**, **delEdge**, and **weight**. They simply start at the beginning of the adjacency list and move along it until the desired vertex has been found. Note that **isEdge** checks to see if *j* is already the current neighbor in *i*'s adjacency list, since this will often be true when processing the neighbors of each vertex in turn.

```
/** Graph: Adjacency matrix */
class Graphm implements Graph {
  private int[][] matrix;              // The edge matrix
  private int numEdge;                 // Number of edges
  private int[] Mark;                   // The mark array

  public Graphm() {}                   // Constructors
  public Graphm(int n) {
    Init(n);
  }

  public void Init(int n) {
    Mark = new int[n];
    matrix = new int[n][n];
    numEdge = 0;
  }

  public int n() { return Mark.length; } // # of vertices
  public int e() { return numEdge; }      // # of edges

  /** @return v's first neighbor */
  public int first(int v) {
    for (int i=0; i<Mark.length; i++)
      if (matrix[v][i] != 0) return i;
    return Mark.length;  // No edge for this vertex
  }

 /** @return v's next neighbor after w */
  public int next(int v, int w) {
    for (int i=w+1; i<Mark.length; i++)
      if (matrix[v][i] != 0)
        return i;
    return Mark.length;  // No next edge;
  }

  /** Set the weight for an edge */
  public void setEdge(int i, int j, int wt) {
    assert wt!=0 : "Cannot set weight to 0";
    if (matrix[i][j] == 0) numEdge++;
    matrix[i][j] = wt;
  }

  /** Delete an edge */
  public void delEdge(int i, int j) { // Delete edge (i, j)
    if (matrix[i][j] != 0) numEdge--;
    matrix[i][j] = 0;
  }

  /** Determine if an edge is in the graph */
  public boolean isEdge(int i, int j)
    { return matrix[i][j] != 0; }
```

Figure 11.6 An implementation for the adjacency matrix implementation.

```
  /** @return an edge's weight */
  public int weight(int i, int j) {
    return matrix[i][j];
  }

  /** Set/Get the mark value for a vertex */
  public void setMark(int v, int val) { Mark[v] = val; }
  public int getMark(int v) { return Mark[v]; }
}
```

Figure 11.6 (continued)

11.3   Graph Traversals

Often it is useful to visit the vertices of a graph in some specific order based on the graph's topology. This is known as a **graph traversal** and is similar in concept to a tree traversal. Recall that tree traversals visit every node exactly once, in some specified order such as preorder, inorder, or postorder. Multiple tree traversals exist because various applications require the nodes to be visited in a particular order. For example, to print a BST's nodes in ascending order requires an inorder traversal as opposed to some other traversal. Standard graph traversal orders also exist. Each is appropriate for solving certain problems. For example, many problems in artificial intelligence programming are modeled using graphs. The problem domain may consist of a large collection of states, with connections between various pairs of states. Solving the problem may require getting from a specified start state to a specified goal state by moving between states only through the connections. Typically, the start and goal states are not directly connected. To solve this problem, the vertices of the graph must be searched in some organized manner.

Graph traversal algorithms typically begin with a start vertex and attempt to visit the remaining vertices from there. Graph traversals must deal with a number of troublesome cases. First, it may not be possible to reach all vertices from the start vertex. This occurs when the graph is not connected. Second, the graph may contain cycles, and we must make sure that cycles do not cause the algorithm to go into an infinite loop.

Graph traversal algorithms can solve both of these problems by maintaining a **mark bit** for each vertex on the graph. At the beginning of the algorithm, the mark bit for all vertices is cleared. The mark bit for a vertex is set when the vertex is first visited during the traversal. If a marked vertex is encountered during traversal, it is not visited a second time. This keeps the program from going into an infinite loop when it encounters a cycle.

Once the traversal algorithm completes, we can check to see if all vertices have been processed by checking the mark bit array. If not all vertices are marked, we can continue the traversal from another unmarked vertex. Note that this process

```
/** Adjacency list graph implementation */
class Graphl implements Graph {
  private GraphList[] vertex;              // The vertex list
  private int numEdge;                     // Number of edges
  private int[] Mark;                       // The mark array

  public Graphl() {}
  public Graphl(int n)                     // Constructor
    { Init(n); }

  public void Init(int n) {
    Mark = new int[n];
    vertex = new GraphList[n];
    for (int i=0; i<n; i++)
      vertex[i] = new GraphList();
    numEdge = 0;
  }

  public int n() { return Mark.length; } // # of vertices
  public int e() { return numEdge; }     // # of edges

  /** @return v's first neighbor */
  public int first(int v) {
    if (vertex[v].length() == 0)
      return Mark.length;    // No neighbor
    vertex[v].moveToStart();
    Edge it = vertex[v].getValue();
    return it.vertex();
  }

 /** @return v's next neighbor after w */
  public int next(int v, int w) {
    Edge it = null;
    if (isEdge(v, w)) {
      vertex[v].next();
      it = vertex[v].getValue();
    }
    if (it != null)
      return it.vertex();
    return Mark.length; // No neighbor
  }
```

Figure 11.7 An implementation for the adjacency list.

```java
/** Set the weight for an edge */
public void setEdge(int i, int j, int weight) {
  assert weight != 0 : "May not set weight to 0";
  Edge currEdge = new Edge(j, weight);
  if (isEdge(i, j)) { // Edge already exists in graph
    vertex[i].remove();
    vertex[i].insert(currEdge);
  }
  else { // Keep neighbors sorted by vertex index
    numEdge++;
    for (vertex[i].moveToStart();
         vertex[i].currPos() < vertex[i].length();
         vertex[i].next())
      if (vertex[i].getValue().vertex() > j) break;
    vertex[i].insert(currEdge);
  }
}

/** Delete an edge */
public void delEdge(int i, int j)
  { if (isEdge(i, j)) { vertex[i].remove(); numEdge--; } }

/** Determine if an edge is in the graph */
public boolean isEdge(int v, int w) {
  Edge it = vertex[v].getValue();
  // Check if j is the current neighbor in the list
  if ((it != null) && (it.vertex() == w)) return true;
  for (vertex[v].moveToStart();
       vertex[v].currPos() < vertex[v].length();
       vertex[v].next())                    // Check whole list
    if (vertex[v].getValue().vertex() == w) return true;
  return false;
}

/** @return an edge's weight */
public int weight(int i, int j) {
  if (isEdge(i, j)) return vertex[i].getValue().weight();
  return 0;
}

/** Set/Get the mark value for a vertex */
public void setMark(int v, int val) { Mark[v] = val; }
public int getMark(int v) { return Mark[v]; }
}
```

Figure 11.7 (continued)

works regardless of whether the graph is directed or undirected. To ensure visiting all vertices, **graphTraverse** could be called as follows on a graph **G**:

```
void graphTraverse(Graph G) {
  int v;
  for (v=0; v<G.n(); v++)
    G.setMark(v, UNVISITED); // Initialize
  for (v=0; v<G.n(); v++)
    if (G.getMark(v) == UNVISITED)
      doTraverse(G, v);
}
```

Function "**doTraverse**" might be implemented by using one of the graph traversals described in this section.
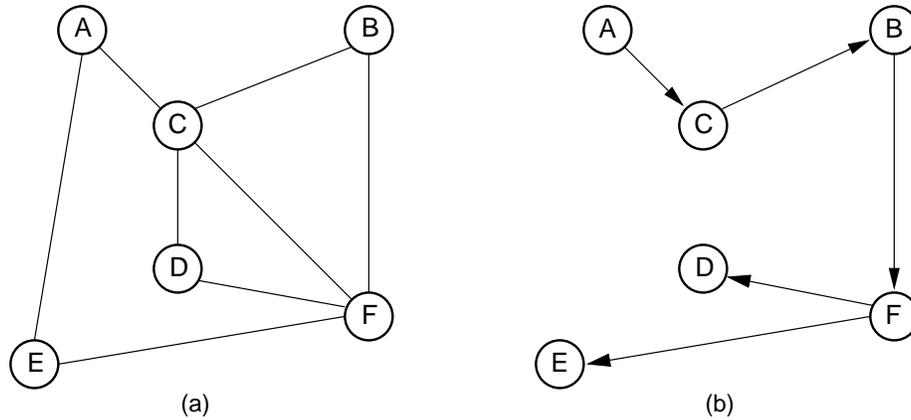
11.3.1 Depth-First Search

The first method of organized graph traversal is called **depth-first search** (DFS). Whenever a vertex V is visited during the search, DFS will recursively visit all of V's unvisited neighbors. Equivalently, DFS will add all edges leading out of v to a stack. The next vertex to be visited is determined by popping the stack and following that edge. The effect is to follow one branch through the graph to its conclusion, then it will back up and follow another branch, and so on. The DFS process can be used to define a **depth-first search tree**. This tree is composed of the edges that were followed to any new (unvisited) vertex during the traversal, and leaves out the edges that lead to already visited vertices. DFS can be applied to directed or undirected graphs. Here is an implementation for the DFS algorithm:

```
/** Depth first search */
static void DFS(Graph G, int v) {
  PreVisit(G, v);                      // Take appropriate action
  G.setMark(v, VISITED);
  for (int w = G.first(v); w < G.n() ; w = G.next(v, w))
    if (G.getMark(w) == UNVISITED)
      DFS(G, w);
  PostVisit(G, v);                     // Take appropriate action
}
```

This implementation contains calls to functions **PreVisit** and **PostVisit**. These functions specify what activity should take place during the search. Just as a preorder tree traversal requires action before the subtrees are visited, some graph traversals require that a vertex be processed before ones further along in the DFS. Alternatively, some applications require activity *after* the remaining vertices are processed; hence the call to function **PostVisit**. This would be a natural opportunity to make use of the visitor design pattern described in Section 1.3.2.

Figure 11.8 shows a graph and its corresponding depth-first search tree. Figure 11.9 illustrates the DFS process for the graph of Figure 11.8(a).

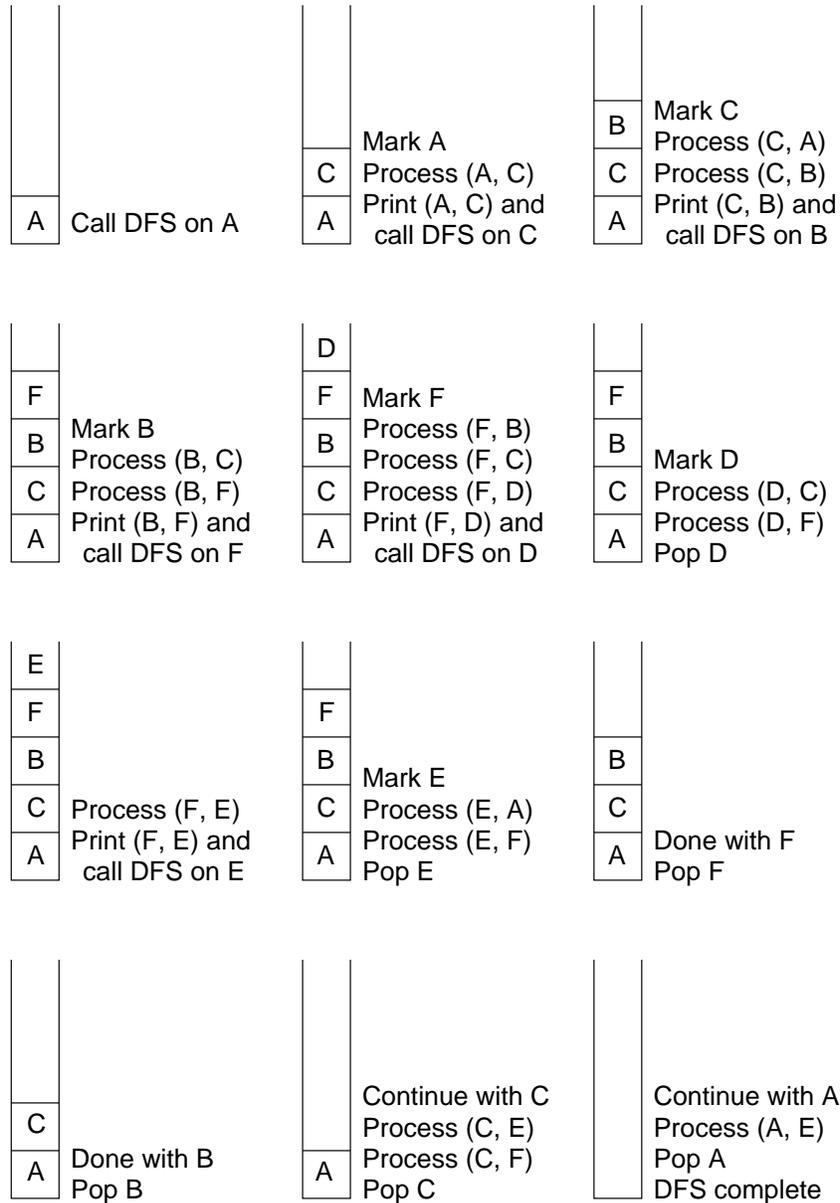**Figure 11.8** (a) A graph. (b) The depth-first search tree for the graph when starting at Vertex *A*.

DFS processes each edge once in a directed graph. In an undirected graph, DFS processes each edge from both directions. Each vertex must be visited, but only once, so the total cost is $\Theta(|\mathbf{V}| + |\mathbf{E}|)$.

11.3.2 Breadth-First Search

Our second graph traversal algorithm is known as a **breadth-first search** (BFS). BFS examines all vertices connected to the start vertex before visiting vertices further away. BFS is implemented similarly to DFS, except that a queue replaces the recursion stack. Note that if the graph is a tree and the start vertex is at the root, BFS is equivalent to visiting vertices level by level from top to bottom. Figure 11.10 provides an implementation for the BFS algorithm. Figure 11.11 shows a graph and the corresponding breadth-first search tree. Figure 11.12 illustrates the BFS process for the graph of Figure 11.11(a).

11.3.3 Topological Sort

Assume that we need to schedule a series of tasks, such as classes or construction jobs, where we cannot start one task until after its prerequisites are completed. We wish to organize the tasks into a linear order that allows us to complete them one at a time without violating any prerequisites. We can model the problem using a DAG. The graph is directed because one task is a prerequisite of another — the vertices have a directed relationship. It is acyclic because a cycle would indicate a conflicting series of prerequisites that could not be completed without violating at least one prerequisite. The process of laying out the vertices of a DAG in a linear order to meet the prerequisite rules is called a **topological sort**. Figure 11.14

A	Call DFS on A

C	Mark A
A	Process (A, C)
	Print (A, C) and
	call DFS on C

B	Mark C
C	Process (C, A)
A	Process (C, B)
	Print (C, B) and
	call DFS on B

F	
B	Mark B
C	Process (B, C)
A	Process (B, F)
	Print (B, F) and
	call DFS on F

D	
F	Mark F
B	Process (F, B)
C	Process (F, C)
A	Process (F, D)
	Print (F, D) and
	call DFS on D

F	
B	Mark D
C	Process (D, C)
A	Process (D, F)
	Pop D

E	
F	
B	
C	Process (F, E)
A	Print (F, E) and
	call DFS on E

F	
B	Mark E
C	Process (E, A)
A	Process (E, F)
	Pop E

B	
C	
A	Done with F
	Pop F

C	Done with B
A	Pop B

A	Continue with C
	Process (C, E)
	Process (C, F)
	Pop C

	Continue with A
	Process (A, E)
	Pop A
	DFS complete

Figure 11.9 A detailed illustration of the DFS process for the graph of Figure 11.8(a) starting at Vertex *A*. The steps leading to each change in the recursion stack are described.

```
/** Breadth first (queue-based) search */
static void BFS(Graph G, int start) {
  Queue<Integer> Q = new AQueue<Integer>(G.n());
  Q.enqueue(start);
  G.setMark(start, VISITED);
  while (Q.length() > 0) {      // Process each vertex on Q
    int v = Q.dequeue();
    PreVisit(G, v);             // Take appropriate action
    for (int w = G.first(v); w < G.n(); w = G.next(v, w))
      if (G.getMark(w) == UNVISITED) { // Put neighbors on Q
        G.setMark(w, VISITED);
        Q.enqueue(w);
      }
    PostVisit(G, v);           // Take appropriate action
  }
}
```

Figure 11.10 Implementation for the breadth-first graph traversal algorithm
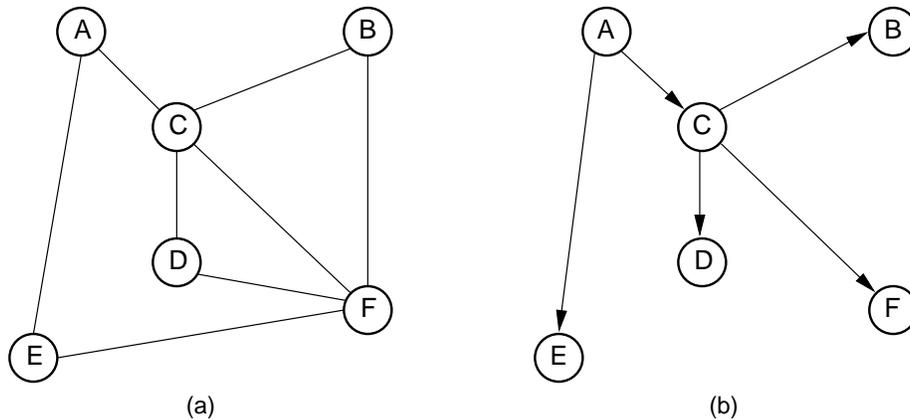
Figure 11.11 (a) A graph. (b) The breadth-first search tree for the graph when starting at Vertex *A*.

illustrates the problem. An acceptable topological sort for this example is *J1*, *J2*, *J3*, *J4*, *J5*, *J6*, *J7*.

A topological sort may be found by performing a DFS on the graph. When a vertex is visited, no action is taken (i.e., function **PreVisit** does nothing). When the recursion pops back to that vertex, function **PostVisit** prints the vertex. This yields a topological sort in reverse order. It does not matter where the sort starts, as long as all vertices are visited in the end. Figure 11.13 shows an implementation for the DFS-based algorithm.

Using this algorithm starting at *J1* and visiting adjacent neighbors in alphabetic order, vertices of the graph in Figure 11.14 are printed out in the order *J7*, *J5*, *J4*, *J6*, *J2*, *J3*, *J1*. Reversing this yields the topological sort *J1*, *J3*, *J2*, *J6*, *J4*, *J5*, *J7*.

A		

Initial call to BFS on A.
Mark A and put on the queue.

C	E	

Dequeue A.
Process (A, C).
Mark and enqueue C. Print (A, C)
Process (A, E).
Mark and enqueue E. Print(A, E).
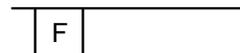
E	B	D	F	

Dequeue C.
Process (C, A). Ignore.
Process (C, B).
Mark and enqueue B. Print (C, B).
Process (C, D).
Mark and enqueue D. Print (C, D).
Process (C, F).
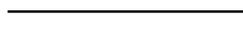Mark and enqueue F. Print (C, F).

B	D	F	

Dequeue E.
Process (E, A). Ignore.
Process (E, F). Ignore.

D	F	

Dequeue B.
Process (B, C). Ignore.
Process (B, F). Ignore.

F	

Dequeue D.
Process (D, C). Ignore.
Process (D, F). Ignore.

Dequeue F.
Process (F, B). Ignore.
Process (F, C). Ignore.
Process (F, D). Ignore.
BFS is complete.

Figure 11.12 A detailed illustration of the BFS process for the graph of Figure 11.11(a) starting at Vertex *A*. The steps leading to each change in the queue are described.

```
/** Recursive topological sort */
static void topsort(Graph G) {
  for (int i=0; i<G.n(); i++)   // Initialize Mark array
    G.setMark(i, UNVISITED);
  for (int i=0; i<G.n(); i++)   // Process all vertices
    if (G.getMark(i) == UNVISITED)
      tophelp(G, i);            // Recursive helper function
}

/** Topsort helper function */
static void tophelp(Graph G, int v) {
  G.setMark(v, VISITED);
  for (int w = G.first(v); w < G.n(); w = G.next(v, w))
    if (G.getMark(w) == UNVISITED)
      tophelp(G, w);
  printout(v);                  // PostVisit for Vertex v
}
```

Figure 11.13 Implementation for the recursive topological sort.
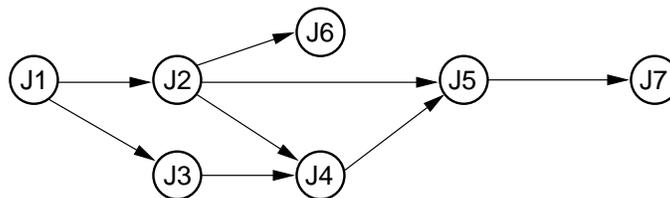
Figure 11.14 An example graph for topological sort. Seven tasks have dependencies as shown by the directed graph.

We can implement topological sort using a queue instead of recursion, as follows. First visit all edges, counting the number of edges that lead to each vertex (i.e., count the number of prerequisites for each vertex). All vertices with no prerequisites are placed on the queue. We then begin processing the queue. When Vertex *V* is taken off of the queue, it is printed, and all neighbors of *V* (that is, all vertices that have *V* as a prerequisite) have their counts decremented by one. Place on the queue any neighbor whose count becomes zero. If the queue becomes empty without printing all of the vertices, then the graph contains a cycle (i.e., there is no possible ordering for the tasks that does not violate some prerequisite). The printed order for the vertices of the graph in Figure 11.14 using the queue version of topological sort is **J1**, **J2**, **J3**, **J6**, **J4**, **J5**, **J7**. Figure 11.15 shows an implementation for the algorithm.

11.4   Shortest-Paths Problems

On a road map, a road connecting two towns is typically labeled with its distance. We can model a road network as a directed graph whose edges are labeled with

```
static void topsort(Graph G) { // Topological sort: Queue
  Queue<Integer> Q = new AQueue<Integer>(G.n());
  int[] Count = new int[G.n()];
  int v;
  for (v=0; v<G.n(); v++) Count[v] = 0; // Initialize
  for (v=0; v<G.n(); v++)          // Process every edge
    for (int w = G.first(v); w < G.n(); w = G.next(v, w))
        Count[w]++;                // Add to v's prereq count
  for (v=0; v<G.n(); v++)          // Initialize Queue
    if (Count[v] == 0)            // V has no prerequisites
      Q.enqueue(v);
  while (Q.length() > 0) {        // Process the vertices
    v = Q.dequeue().intValue();
    printout(v);                   // PreVisit for Vertex V
    for (int w = G.first(v); w < G.n(); w = G.next(v, w)) {
      Count[w]--;                  // One less prerequisite
      if (Count[w] == 0)          // This vertex is now free
        Q.enqueue(w);
    }
  }
}
```

Figure 11.15 A queue-based topological sort algorithm.

real numbers. These numbers represent the distance (or other cost metric, such as travel time) between two vertices. These labels may be called **weights**, **costs**, or **distances**, depending on the application. Given such a graph, a typical problem is to find the total length of the shortest path between two specified vertices. This is not a trivial problem, because the shortest path may not be along the edge (if any) connecting two vertices, but rather may be along a path involving one or more intermediate vertices. For example, in Figure 11.16, the cost of the path from A to B to D is 15. The cost of the edge directly from A to D is 20. The cost of the path from A to C to B to D is 10. Thus, the shortest path from A to D is 10 (not along the edge connecting A to D). We use the notation $d(A, D) = 10$ to indicate that the shortest distance from A to D is 10. In Figure 11.16, there is no path from E to B, so we set $d(E, B) = \infty$. We define $w(A, D) = 20$ to be the weight of edge (A, D), that is, the weight of the direct connection from A to D. Because there is no edge from E to B, $w(E, B) = \infty$. Note that $w(D, A) = \infty$ because the graph of Figure 11.16 is directed. We assume that all weights are positive.

11.4.1   Single-Source Shortest Paths

This section presents an algorithm to solve the **single-source shortest-paths** problem. Given Vertex S in Graph **G**, find a shortest path from S to every other vertex in **G**. We might want only the shortest path between two vertices, S and T. However in the worst case, while finding the shortest path from S to T, we might find the shortest paths from S to every other vertex as well. So there is no better alg-
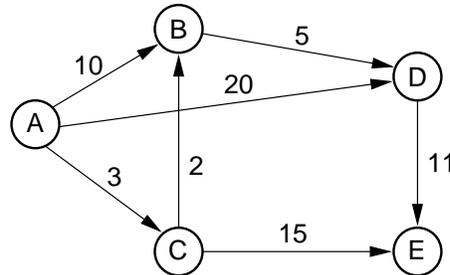
Figure 11.16 Example graph for shortest-path definitions.

orithm (in the worst case) for finding the shortest path to a single vertex than to find shortest paths to all vertices. The algorithm described here will only compute the distance to every such vertex, rather than recording the actual path. Recording the path requires modifications to the algorithm that are left as an exercise.

Computer networks provide an application for the single-source shortest-paths problem. The goal is to find the cheapest way for one computer to broadcast a message to all other computers on the network. The network can be modeled by a graph with edge weights indicating time or cost to send a message to a neighboring computer.

For unweighted graphs (or whenever all edges have the same cost), the single-source shortest paths can be found using a simple breadth-first search. When weights are added, BFS will not give the correct answer.

One approach to solving this problem when the edges have differing weights might be to process the vertices in a fixed order. Label the vertices v_0 to v_{n-1}, with $S = v_0$. When processing Vertex v_1, we take the edge connecting v_0 and v_1. When processing v_2, we consider the shortest distance from v_0 to v_2 and compare that to the shortest distance from v_0 to v_1 to v_2. When processing Vertex v_i, we consider the shortest path for Vertices v_0 through v_{i-1} that have already been processed. Unfortunately, the true shortest path to v_i might go through Vertex v_j for $j > i$. Such a path will not be considered by this algorithm. However, the problem would not occur if we process the vertices in order of distance from S. Assume that we have processed in order of distance from S to the first $i - 1$ vertices that are closest to S; call this set of vertices **S**. We are now about to process the ith closest vertex; call it X. A shortest path from S to X must have its next-to-last vertex in **S**. Thus,

$$\mathrm{d}(S, X) = \min_{U \in \mathbf{S}}(\mathrm{d}(S, U) + \mathrm{w}(U, X)).$$

In other words, the shortest path from S to X is the minimum over all paths that go from S to U, then have an edge from U to X, where U is some vertex in **S**.

This solution is usually referred to as Dijkstra's algorithm. It works by maintaining a distance estimate $\mathbf{D}(X)$ for all vertices X in **V**. The elements of **D** are ini-

```
// Compute shortest path distances from s, store them in D
static void Dijkstra(Graph G, int s, int[] D) {
  for (int i=0; i<G.n(); i++)     // Initialize
    D[i] = Integer.MAX_VALUE;
  D[s] = 0;
  for (int i=0; i<G.n(); i++) {  // Process the vertices
    int v = minVertex(G, D);       // Find next-closest vertex
    G.setMark(v, VISITED);
    if (D[v] == Integer.MAX_VALUE) return; // Unreachable
    for (int w = G.first(v); w < G.n(); w = G.next(v, w))
      if (D[w] > (D[v] + G.weight(v, w)))
        D[w] = D[v] + G.weight(v, w);
  }
}
```

Figure 11.17 An implementation for Dijkstra's algorithm.

tialized to the value **INFINITE**. Vertices are processed in order of distance from
S. Whenever a vertex *V* is processed, $\mathbf{D}(X)$ is updated for every neighbor *X* of *V*.
Figure 11.17 shows an implementation for Dijkstra's algorithm. At the end, array *D*
will contain the shortest distance values.

There are two reasonable solutions to the key issue of finding the unvisited
vertex with minimum distance value during each pass through the main **for** loop.
The first method is simply to scan through the list of $|\mathbf{V}|$ vertices searching for the
minimum value, as follows:

```
static int minVertex(Graph G, int[] D) {
  int v = 0;  // Initialize v to any unvisited vertex;
  for (int i=0; i<G.n(); i++)
    if (G.getMark(i) == UNVISITED) { v = i; break; }
  for (int i=0; i<G.n(); i++)  // Now find smallest value
    if ((G.getMark(i) == UNVISITED) && (D[i] < D[v]))
      v = i;
  return v;
}
```

Because this scan is done $|\mathbf{V}|$ times, and because each edge requires a constant-
time update to D, the total cost for this approach is $\Theta(|\mathbf{V}|^2 + |\mathbf{E}|) = \Theta(|\mathbf{V}|^2)$,
because $|\mathbf{E}|$ is in $O(|\mathbf{V}|^2)$.

The second method is to store unprocessed vertices in a min-heap ordered by
distance values. The next-closest vertex can be found in the heap in $\Theta(\log |\mathbf{V}|)$
time. Every time we modify $\mathbf{D}(X)$, we could reorder *X* in the heap by deleting
and reinserting it. This is an example of a priority queue with priority update, as
described in Section 5.5. To implement true priority updating, we would need to
store with each vertex its array index within the heap. A simpler approach is to
add the new (smaller) distance value for a given vertex as a new record in the heap.
The smallest value for a given vertex currently in the heap will be found first, and
greater distance values found later will be ignored because the vertex will already
be marked as **VISITED**. The only disadvantage to repeatedly inserting distance

```
/** Dijkstra's shortest-paths: priority queue version */
static void Dijkstra(Graph G, int s, int[] D) {
  int v;                                // The current vertex
  DijkElem[] E = new DijkElem[G.e()];   // Heap for edges
  E[0] = new DijkElem(s, 0);            // Initial vertex
  MinHeap<DijkElem> H = new MinHeap<DijkElem>(E, 1, G.e());
  for (int i=0; i<G.n(); i++)           // Initialize distance
    D[i] = Integer.MAX_VALUE;
  D[s] = 0;
  for (int i=0; i<G.n(); i++) {         // For each vertex
    do { v = (H.removemin()).vertex(); }    // Get position
      while (G.getMark(v) == VISITED);
    G.setMark(v, VISITED);
    if (D[v] == Integer.MAX_VALUE) return;  // Unreachable
    for (int w = G.first(v); w < G.n(); w = G.next(v, w))
      if (D[w] > (D[v] + G.weight(v, w))) { // Update D
        D[w] = D[v] + G.weight(v, w);
        H.insert(new DijkElem(w, D[w]));
      }
  }
}
```

Figure 11.18 An implementation for Dijkstra's algorithm using a priority queue.

values is that it will raise the number of elements in the heap from $\Theta(|\mathbf{V}|)$ to $\Theta(|\mathbf{E}|)$ in the worst case. The time complexity is $\Theta((|\mathbf{V}| + |\mathbf{E}|)\log|\mathbf{E}|)$, because for each edge we must reorder the heap. Because the objects stored on the heap need to know both their vertex number and their distance, we create a simple class for the purpose called **DijkElem**, as follows. **DijkElem** is quite similar to the **Edge** class used by the adjacency list representation.

```
class DijkElem implements Comparable<DijkElem> {
  private int vertex;
  private int weight;

  public DijkElem(int inv, int inw)
    { vertex = inv; weight = inw; }
  public DijkElem() {vertex = 0; weight = 0; }

  public int key() { return weight; }
  public int vertex() { return vertex; }
  public int compareTo(DijkElem that) {
    if (weight < that.key()) return -1;
    else if  (weight == that.key()) return 0;
    else return 1;
  }
}
```

Figure 11.18 shows an implementation for Dijkstra's algorithm using the priority queue.

Using **MinVertex** to scan the vertex list for the minimum value is more efficient when the graph is dense, that is, when $|\mathbf{E}|$ approaches $|\mathbf{V}|^2$. Using a prior-

	A	**B**	**C**	**D**	**E**
Initial	0	∞	∞	∞	∞
Process A	0	10	3	20	∞
Process C	0	5	3	20	18
Process B	0	5	3	10	18
Process D	0	5	3	10	18
Process E	0	5	3	10	18

Figure 11.19 A listing for the progress of Dijkstra's algorithm operating on the graph of Figure 11.16. The start vertex is *A*.

ity queue is more efficient when the graph is sparse because its cost is $\Theta((|\mathbf{V}| + |\mathbf{E}|) \log |\mathbf{E}|)$. However, when the graph is dense, this cost can become as great as $\Theta(|\mathbf{V}|^2 \log |\mathbf{E}|) = \Theta(|V|^2 \log |V|)$.

Figure 11.19 illustrates Dijkstra's algorithm. The start vertex is *A*. All vertices except *A* have an initial value of ∞. After processing Vertex *A*, its neighbors have their D estimates updated to be the direct distance from *A*. After processing *C* (the closest vertex to *A*), Vertices *B* and *E* are updated to reflect the shortest path through *C*. The remaining vertices are processed in order *B*, *D*, and *E*.

11.5 Minimum-Cost Spanning Trees

The **minimum-cost spanning tree** (MST) problem takes as input a connected, undirected graph **G**, where each edge has a distance or weight measure attached. The MST is the graph containing the vertices of **G** along with the subset of **G**'s edges that (1) has minimum total cost as measured by summing the values for all of the edges in the subset, and (2) keeps the vertices connected. Applications where a solution to this problem is useful include soldering the shortest set of wires needed to connect a set of terminals on a circuit board, and connecting a set of cities by telephone lines in such a way as to require the least amount of cable.

The MST contains no cycles. If a proposed MST did have a cycle, a cheaper MST could be had by removing any one of the edges in the cycle. Thus, the MST is a free tree with $|\mathbf{V}| - 1$ edges. The name "minimum-cost spanning tree" comes from the fact that the required set of edges forms a tree, it spans the vertices (i.e., it connects them together), and it has minimum cost. Figure 11.20 shows the MST for an example graph.

11.5.1 Prim's Algorithm

The first of our two algorithms for finding MSTs is commonly referred to as Prim's algorithm. Prim's algorithm is very simple. Start with any Vertex *N* in the graph, setting the MST to be *N* initially. Pick the least-cost edge connected to *N*. This
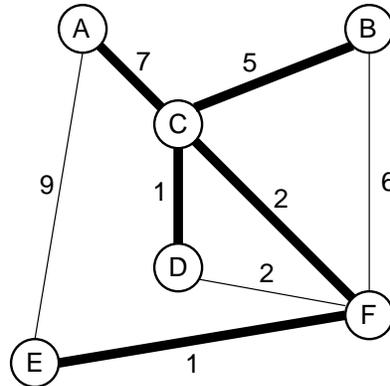
Figure 11.20 A graph and its MST. All edges appear in the original graph. Those edges drawn with heavy lines indicate the subset making up the MST. Note that edge (*C*, *F*) could be replaced with edge (*D*, *F*) to form a different MST with equal cost.

edge connects *N* to another vertex; call this *M*. Add Vertex *M* and Edge (*N*, *M*) to the MST. Next, pick the least-cost edge coming from either *N* or *M* to any other vertex in the graph. Add this edge and the new vertex it reaches to the MST. This process continues, at each step expanding the MST by selecting the least-cost edge from a vertex currently in the MST to a vertex not currently in the MST.

Prim's algorithm is quite similar to Dijkstra's algorithm for finding the single-source shortest paths. The primary difference is that we are seeking not the next closest vertex to the start vertex, but rather the next closest vertex to any vertex currently in the MST. Thus we replace the lines

```
if (D[w] > (D[v] + G.weight(v, w)))
  D[w] = D[v] + G.weight(v, w);
```

in Djikstra's algorithm with the lines

```
if (D[w] > G.weight(v, w))
  D[w] = G.weight(v, w);
```

in Prim's algorithm.

Figure 11.21 shows an implementation for Prim's algorithm that searches the distance matrix for the next closest vertex. For each vertex *I*, when *I* is processed by Prim's algorithm, an edge going to *I* is added to the MST that we are building. Array **V[I]** stores the previously visited vertex that is closest to Vertex *I*. This information lets us know which edge goes into the MST when Vertex *I* is processed. The implementation of Figure 11.21 also contains calls to **AddEdgetoMST** to indicate which edges are actually added to the MST.

Alternatively, we can implement Prim's algorithm using a priority queue to find the next closest vertex, as shown in Figure 11.22. As with the priority queue version of Dijkstra's algorithm, the heap's **Elem** type stores a **DijkElem** object.

```
/** Compute a minimal-cost spanning tree */
static void Prim(Graph G, int s, int[] D, int[] V) {
  for (int i=0; i<G.n(); i++)    // Initialize
    D[i] = Integer.MAX_VALUE;
  D[s] = 0;
  for (int i=0; i<G.n(); i++) { // Process the vertices
    int v = minVertex(G, D);
    G.setMark(v, VISITED);
    if (v != s) AddEdgetoMST(V[v], v);
    if (D[v] == Integer.MAX_VALUE) return; // Unreachable
    for (int w = G.first(v); w < G.n(); w = G.next(v, w))
      if (D[w] > G.weight(v, w)) {
        D[w] = G.weight(v, w);
        V[w] = v;
      }
  }
}
```

Figure 11.21 An implementation for Prim's algorithm.

```
/** Prims's MST algorithm: priority queue version */
static void Prim(Graph G, int s, int[] D, int[] V) {
  int v;                              // The current vertex
  DijkElem[] E = new DijkElem[G.e()]; // Heap for edges
  E[0] = new DijkElem(s, 0);          // Initial vertex
  MinHeap<DijkElem> H = new MinHeap<DijkElem>(E, 1, G.e());
  for (int i=0; i<G.n(); i++)         // Initialize
    D[i] = Integer.MAX_VALUE;         //   distances
  D[s] = 0;
  for (int i=0; i<G.n(); i++) {       // Now, get distances
    do { v = (H.removemin()).vertex(); } // Get position
      while (G.getMark(v) == VISITED);
    G.setMark(v, VISITED);
    if (v != s) AddEdgetoMST(V[v], v); // Add edge to MST
    if (D[v] == Integer.MAX_VALUE) return; // Unreachable
    for (int w = G.first(v); w < G.n(); w = G.next(v, w))
      if (D[w] > G.weight(v, w)) {    // Update D
        D[w] = G.weight(v, w);
        V[w] = v;                     // Where it came from
        H.insert(new DijkElem(w, D[w]));
      }
  }
}
```

Figure 11.22 An implementation of Prim's algorithm using a priority queue.

Prim's algorithm is an example of a greedy algorithm. At each step in the `for` loop, we select the least-cost edge that connects some marked vertex to some unmarked vertex. The algorithm does not otherwise check that the MST really should include this least-cost edge. This leads to an important question: Does Prim's algorithm work correctly? Clearly it generates a spanning tree (because each pass through the `for` loop adds one edge and one unmarked vertex to the spanning tree until all vertices have been added), but does this tree have minimum cost?

**Theorem 11.1** *Prim's algorithm produces a minimum-cost spanning tree.*

Proof: We will use a proof by contradiction. Let $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ be a graph for which Prim's algorithm does *not* generate an MST. Define an ordering on the vertices according to the order in which they were added by Prim's algorithm to the MST: $v_0, v_1, ..., v_{n-1}$. Let edge e_i connect (v_x, v_i) for some $x < i$ and $i \geq 1$. Let e_j be the lowest numbered (first) edge added by Prim's algorithm such that the set of edges selected so far *cannot* be extended to form an MST for $\mathbf{G}$. In other words, e_j is the first edge where Prim's algorithm "went wrong." Let $\mathbf{T}$ be the "true" MST. Call v_p $(p < j)$ the vertex connected by edge e_j, that is, $e_j = (v_p, v_j)$.

Because $\mathbf{T}$ is a tree, there exists some path in $\mathbf{T}$ connecting v_p and v_j. There must be some edge e' in this path connecting vertices v_u and v_w, with $u < j$ and $w \geq j$. Because e_j is not part of $\mathbf{T}$, adding edge e_j to $\mathbf{T}$ forms a cycle. Edge e' must be of lower cost than edge e_j, because Prim's algorithm did not generate an MST. This situation is illustrated in Figure 11.23. However, Prim's algorithm would have selected the least-cost edge available. It would have selected e', not e_j. Thus, it is a contradiction that Prim's algorithm would have selected the wrong edge, and thus, Prim's algorithm must be correct.   □

Example 11.3 For the graph of Figure 11.20, assume that we begin by marking Vertex *A*. From *A*, the least-cost edge leads to Vertex *C*. Vertex *C* and edge (*A*, *C*) are added to the MST. At this point, our candidate edges connecting the MST (Vertices *A* and *C*) with the rest of the graph are (*A*, *E*), (*C*, *B*), (*C*, *D*), and (*C*, *F*). From these choices, the least-cost edge from the MST is (*C*, *D*). So we add Vertex *D* to the MST. For the next iteration, our edge choices are (*A*, *E*), (*C*, *B*), (*C*, *F*), and (*D*, *F*). Because edges (*C*, *F*) and (*D*, *F*) happen to have equal cost, it is an arbitrary decision as to which gets selected. Say we pick (*C*, *F*). The next step marks Vertex *E* and adds edge (*F*, *E*) to the MST. Following in this manner, Vertex *B* (through edge (*C*, *B*)) is marked. At this point, the algorithm terminates.
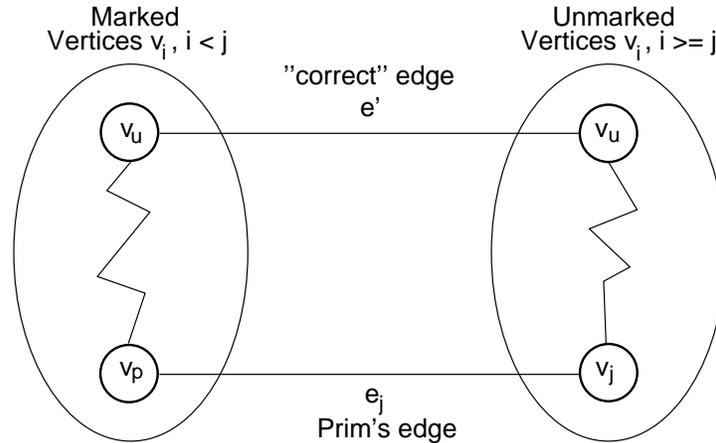
Figure 11.23 Prim's MST algorithm proof. The left oval contains that portion of
the graph where Prim's MST and the "true" MST **T** agree. The right oval contains
the rest of the graph. The two portions of the graph are connected by (at least)
edges e_j (selected by Prim's algorithm to be in the MST) and e' (the "correct"
edge to be placed in the MST). Note that the path from v_w to v_j cannot include
any marked vertex v_i, $i \leq j$, because to do so would form a cycle.
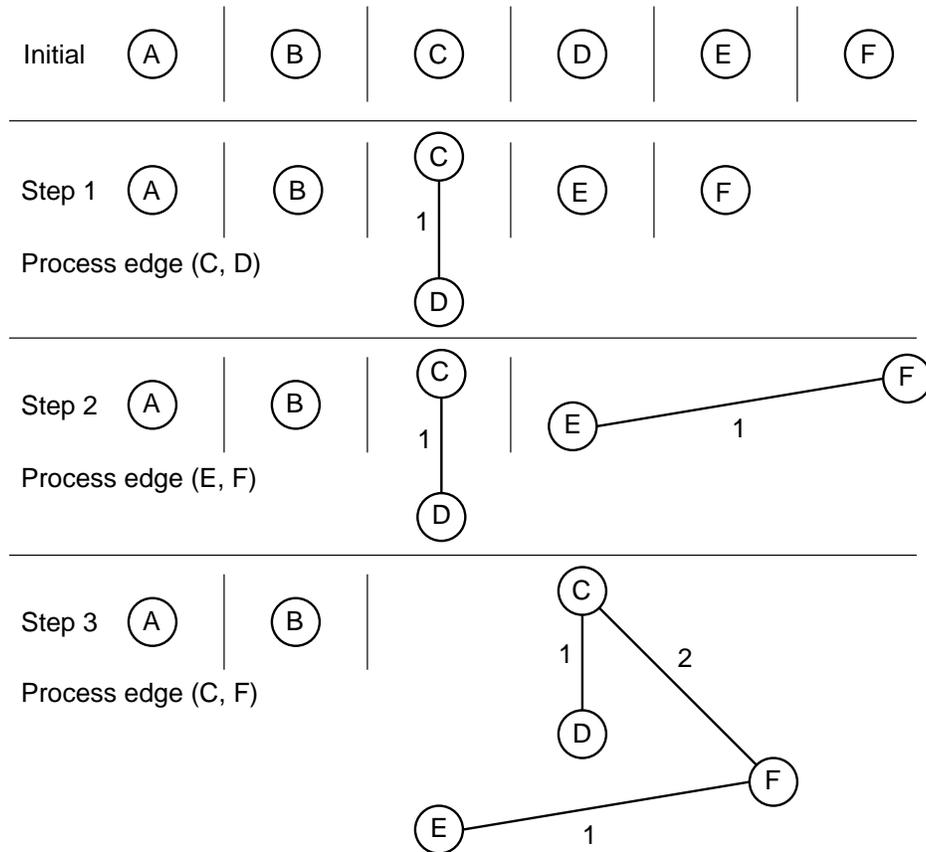
11.5.2 Kruskal's Algorithm

Our next MST algorithm is commonly referred to as Kruskal's algorithm. Kruskal's
algorithm is also a simple, greedy algorithm. First partition the set of vertices into
$|\mathbf{V}|$ equivalence classes (see Section 6.2), each consisting of one vertex. Then pro-
cess the edges in order of weight. An edge is added to the MST, and two equiva-
lence classes combined, if the edge connects two vertices in different equivalence
classes. This process is repeated until only one equivalence class remains.

Example 11.4 Figure 11.24 shows the first three steps of Kruskal's Alg-
orithm for the graph of Figure 11.20. Edge (C, D) has the least cost, and
because C and D are currently in separate MSTs, they are combined. We
next select edge (E, F) to process, and combine these vertices into a single
MST. The third edge we process is (C, F), which causes the MST contain-
ing Vertices C and D to merge with the MST containing Vertices E and F.
The next edge to process is (D, F). But because Vertices D and F are cur-
rently in the same MST, this edge is rejected. The algorithm will continue
on to accept edges (B, C) and (A, C) into the MST.

The edges can be processed in order of weight by using a min-heap. This is
generally faster than sorting the edges first, because in practice we need only visit
a small fraction of the edges before completing the MST. This is an example of
finding only a few smallest elements in a list, as discussed in Section 7.6.

Figure 11.24 Illustration of the first three steps of Kruskal's MST algorithm as applied to the graph of Figure 11.20.

The only tricky part to this algorithm is determining if two vertices belong to the same equivalence class. Fortunately, the ideal algorithm is available for the purpose — the UNION/FIND algorithm based on the parent pointer representation for trees described in Section 6.2. Figure 11.25 shows an implementation for the algorithm. Class **KruskalElem** is used to store the edges on the min-heap.

Kruskal's algorithm is dominated by the time required to process the edges. The **differ** and **UNION** functions are nearly constant in time if path compression and weighted union is used. Thus, the total cost of the algorithm is $\Theta(|\mathbf{E}| \log |\mathbf{E}|)$ in the worst case, when nearly all edges must be processed before all the edges of the spanning tree are found and the algorithm can stop. More often the edges of the spanning tree are the shorter ones,and only about $|\mathbf{V}|$ edges must be processed. If so, the cost is often close to $\Theta(|\mathbf{V}| \log |\mathbf{E}|)$ in the average case.

```
/** Heap element implementation for Kruskal's algorithm */
class KruskalElem implements Comparable<KruskalElem> {
  private int v, w, weight;

  public KruskalElem(int inweight, int inv, int inw)
    { weight = inweight; v = inv; w = inw; }
  public int v1() { return v; }
  public int v2() { return w; }
  public int key() { return weight; }
  public int compareTo(KruskalElem that) {
    if (weight < that.key()) return -1;
    else if  (weight == that.key()) return 0;
    else return 1;
  }
}

/** Kruskal's MST algorithm */
static void Kruskal(Graph G) {
  ParPtrTree A = new ParPtrTree(G.n()); // Equivalence array
  KruskalElem[] E = new KruskalElem[G.e()]; // Minheap array
  int edgecnt = 0; // Count of edges

  for (int i=0; i<G.n(); i++)    // Put edges in the array
    for (int w = G.first(i); w < G.n(); w = G.next(i, w))
      E[edgecnt++] = new KruskalElem(G.weight(i, w), i, w);
  MinHeap<KruskalElem> H =
             new MinHeap<KruskalElem>(E, edgecnt, edgecnt);
  int numMST = G.n();                  // Initially n classes
  for (int i=0; numMST>1; i++) { // Combine equiv classes
    KruskalElem temp = H.removemin(); // Next cheapest
    int v = temp.v1();  int u = temp.v2();
    if (A.differ(v, u)) {         // If in different classes
      A.UNION(v, u);              // Combine equiv classes
      AddEdgetoMST(v, u);  // Add this edge to MST
      numMST--;                   // One less MST
    }
  }
}
```

Figure 11.25 An implementation for Kruskal's algorithm.

11.6   Further Reading

Many interesting properties of graphs can be investigated by playing with the programs in the Stanford Graphbase. This is a collection of benchmark databases and graph processing programs. The Stanford Graphbase is documented in [Knu94].

11.7   Exercises

11.1 Prove by induction that a graph with n vertices has at most $n(n-1)/2$ edges.

11.2 Prove the following implications regarding free trees.

(a) IF an undirected graph is connected and has no simple cycles, THEN the graph has $|\mathbf{V}| - 1$ edges.

(b) IF an undirected graph has $|\mathbf{V}| - 1$ edges and no cycles, THEN the graph is connected.

11.3 (a) Draw the adjacency matrix representation for the graph of Figure 11.26.

(b) Draw the adjacency list representation for the same graph.

(c) If a pointer requires four bytes, a vertex label requires two bytes, and an edge weight requires two bytes, which representation requires more space for this graph?

(d) If a pointer requires four bytes, a vertex label requires one byte, and an edge weight requires two bytes, which representation requires more space for this graph?

11.4 Show the DFS tree for the graph of Figure 11.26, starting at Vertex 1.

11.5 Write a pseudocode algorithm to create a DFS tree for an undirected, connected graph starting at a specified vertex V.

11.6 Show the BFS tree for the graph of Figure 11.26, starting at Vertex 1.

11.7 Write a pseudocode algorithm to create a BFS tree for an undirected, connected graph starting at a specified vertex V.

11.8 The BFS topological sort algorithm can report the existence of a cycle if one is encountered. Modify this algorithm to print the vertices possibly appearing in cycles (if there are any cycles).

11.9 Explain why, in the worst case, Dijkstra's algorithm is (asymptotically) as efficient as any algorithm for finding the shortest path from some vertex I to another vertex J.

11.10 Show the shortest paths generated by running Dijkstra's shortest-paths algorithm on the graph of Figure 11.26, beginning at Vertex 4. Show the D values as each vertex is processed, as in Figure 11.19.

11.11 Modify the algorithm for single-source shortest paths to actually store and return the shortest paths rather than just compute the distances.

11.12 The root of a DAG is a vertex R such that every vertex of the DAG can be reached by a directed path from R. Write an algorithm that takes a directed graph as input and determines the root (if there is one) for the graph. The running time of your algorithm should be $\Theta(|\mathbf{V}| + |\mathbf{E}|)$.

11.13 Write an algorithm to find the longest path in a DAG, where the length of the path is measured by the number of edges that it contains. What is the asymptotic complexity of your algorithm?

11.14 Write an algorithm to determine whether a directed graph of $|\mathbf{V}|$ vertices contains a cycle. Your algorithm should run in $\Theta(|\mathbf{V}| + |\mathbf{E}|)$ time.

11.15 Write an algorithm to determine whether an undirected graph of $|\mathbf{V}|$ vertices contains a cycle. Your algorithm should run in $\Theta(|\mathbf{V}|)$ time.
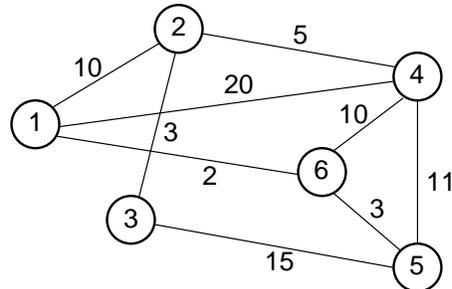
Figure 11.26 Example graph for Chapter 11 exercises.

11.16 The **single-destination shortest-paths** problem for a directed graph is to find the shortest path *from* every vertex to a specified vertex *V*. Write an algorithm to solve the single-destination shortest-paths problem.

11.17 List the order in which the edges of the graph in Figure 11.26 are visited when running Prim's MST algorithm starting at Vertex 3. Show the final MST.

11.18 List the order in which the edges of the graph in Figure 11.26 are visited when running Kruskal's MST algorithm. Each time an edge is added to the MST, show the result on the equivalence array, (e.g., show the array as in Figure 6.7).

11.19 Write an algorithm to find a **maximum** cost spanning tree, that is, the spanning tree with highest possible cost.

11.20 When can Prim's and Kruskal's algorithms yield different MSTs?

11.21 Prove that, if the costs for the edges of Graph **G** are distinct, then only one MST exists for **G**.

11.22 Does either Prim's or Kruskal's algorithm work if there are negative edge weights?

11.23 Consider the collection of edges selected by Dijkstra's algorithm as the shortest paths to the graph's vertices from the start vertex. Do these edges form a spanning tree (not necessarily of minimum cost)? Do these edges form an MST? Explain why or why not.

11.24 Prove that a tree is a bipartite graph.

11.25 Prove that any tree (i.e., a connected, undirected graph with no cycles) can be two-colored. (A graph can be two colored if every vertex can be assigned one of two colors such that no adjacent vertices have the same color.)

11.26 Write an algorithm that determines if an arbitrary undirected graph is a bipartite graph. If the graph is bipartite, then your algorithm should also identify the vertices as to which of the two partitions each belongs to.

11.8   Projects

11.1 Design a format for storing graphs in files. Then implement two functions: one to read a graph from a file and the other to write a graph to a file. Test your functions by implementing a complete MST program that reads an undirected graph in from a file, constructs the MST, and then writes to a second file the graph representing the MST.

11.2 An undirected graph need not explicitly store two separate directed edges to represent a single undirected edge. An alternative would be to store only a single undirected edge (I, J) to connect Vertices I and J. However, what if the user asks for edge (J, I)? We can solve this problem by consistently storing the edge such that the lesser of I and J always comes first. Thus, if we have an edge connecting Vertices 5 and 3, requests for edge (5, 3) and (3, 5) both map to (3, 5) because $3 < 5$.

Looking at the adjacency matrix, we notice that only the lower triangle of the array is used. Thus we could cut the space required by the adjacency matrix from $|\mathbf{V}|^2$ positions to $|\mathbf{V}|(|\mathbf{V}|-1)/2$ positions. Read Section 12.2 on triangular matrices. The re-implement the adjacency matrix representation of Figure 11.6 to implement undirected graphs using a triangular array.

11.3 While the underlying implementation (whether adjacency matrix or adjacency list) is hidden behind the graph ADT, these two implementations can have an impact on the efficiency of the resulting program. For Dijkstra's shortest paths algorithm, two different implementations were given in Section 11.4.1 that provide different ways for determining the next closest vertex at each iteration of the algorithm. The relative costs of these two variants depend on who sparse or dense the graph is. They might also depend on whether the graph is implemented using an adjacency list or adjacency matrix.

Design and implement a study to compare the effects on performance for three variables: (i) the two graph representations (adjacency list and adjacency matrix); (ii) the two implementations for Djikstra's shortest paths algorithm (searching the table of vertex distances or using a priority queue to track the distances), and (iii) sparse versus dense graphs. Be sure to test your implementations on a variety of graphs that are sufficiently large to generate meaningful times.

11.4 The example implementations for DFS and BFS show calls to functions **PreVisit** and **PostVisit**. Re-implement the BFS and DFS functions to make use of the visitor design pattern to handle the pre/post visit functionality.

11.5 Write a program to label the connected components for an undirected graph. In other words, all vertices of the first component are given the first component's label, all vertices of the second component are given the second

component's label, and so on. Your algorithm should work by defining any two vertices connected by an edge to be members of the same equivalence class. Once all of the edges have been processed, all vertices in a given equivalence class will be connected. Use the UNION/FIND implementation from Section 6.2 to implement equivalence classes.