

# Graphs

# 12

---

**T**HIS CHAPTER introduces important mathematical structures called graphs that have applications in subjects as diverse as sociology, chemistry, geography, and electrical engineering. We shall study methods to represent graphs with the data structures available to us and shall construct several important algorithms for processing graphs. Finally, we look at the possibility of using graphs themselves as data structures.

---

- 12.1 Mathematical Background 570**
  - 12.1.1 Definitions and Examples 570
  - 12.1.2 Undirected Graphs 571
  - 12.1.3 Directed Graphs 571
- 12.2 Computer Representation 572**
  - 12.2.1 The Set Representation 572
  - 12.2.2 Adjacency Lists 574
  - 12.2.3 Information Fields 575
- 12.3 Graph Traversal 575**
  - 12.3.1 Methods 575
  - 12.3.2 Depth-First Algorithm 577
  - 12.3.3 Breadth-First Algorithm 578
- 12.4 Topological Sorting 579**
  - 12.4.1 The Problem 579
  - 12.4.2 Depth-First Algorithm 580
  - 12.4.3 Breadth-First Algorithm 581
- 12.5 A Greedy Algorithm: Shortest Paths 583**
  - 12.5.1 The Problem 583
  - 12.5.2 Method 584
  - 12.5.3 Example 585
  - 12.5.4 Implementation 586
- 12.6 Minimal Spanning Trees 587**
  - 12.6.1 The Problem 587
  - 12.6.2 Method 589
  - 12.6.3 Implementation 590
  - 12.6.4 Verification of Prim's Algorithm 593
- 12.7 Graphs as Data Structures 594**
  - Pointers and Pitfalls 596
  - Review Questions 597
  - References for Further Study 597

## 12.1 MATHEMATICAL BACKGROUND

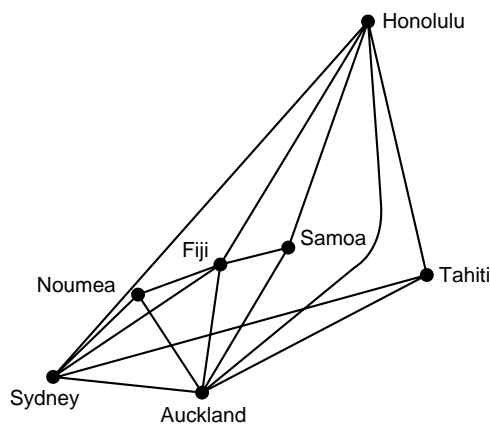
### 12.1.1 Definitions and Examples



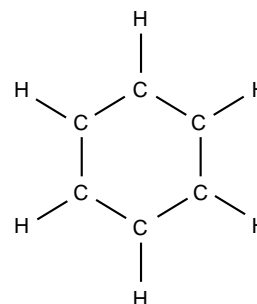
graphs and directed  
graphs

drawings

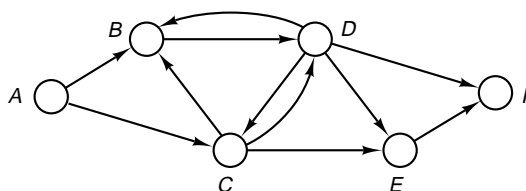
A **graph**  $G$  consists of a set  $V$ , whose members are called the **vertices** of  $G$ , together with a set  $E$  of pairs of distinct vertices from  $V$ . These pairs are called the **edges** of  $G$ . If  $e = (\nu, w)$  is an edge with vertices  $\nu$  and  $w$ , then  $\nu$  and  $w$  are said to *lie on*  $e$ , and  $e$  is said to be **incident** with  $\nu$  and  $w$ . If the pairs are unordered, then  $G$  is called an **undirected graph**; if the pairs are ordered, then  $G$  is called a **directed graph**. The term *directed graph* is often shortened to **digraph**, and the unqualified term *graph* usually means *undirected graph*. The natural way to picture a graph is to represent vertices as points or circles and edges as line segments or arcs connecting the vertices. If the graph is directed, then the line segments or arcs have arrowheads indicating the direction. Figure 12.1 shows several examples of graphs.



Selected South Pacific air routes



Benzene molecule



Message transmission in a network

Figure 12.1. Examples of graphs

The places in the first part of Figure 12.1 are the vertices of the graph, and the air routes connecting them are the edges. In the second part, the hydrogen and carbon atoms (denoted H and C) are the vertices, and the chemical bonds are the edges. The third part of Figure 12.1 shows a directed graph, where the nodes of the network ( $A, B, \dots, F$ ) are the vertices and the edges from one to another have the directions shown by the arrows.

applications



Graphs find their importance as models for many kinds of processes or structures. Cities and the highways connecting them form a graph, as do the components on a circuit board with the connections among them. An organic chemical compound can be considered a graph with the atoms as the vertices and the bonds between them as edges. The people living in a city can be regarded as the vertices of a graph with the relationship *is acquainted with* describing the edges. People working in a corporation form a directed graph with the relation “supervises” describing the edges. The same people could also be considered as an undirected graph, with different edges describing the relationship “works with.”

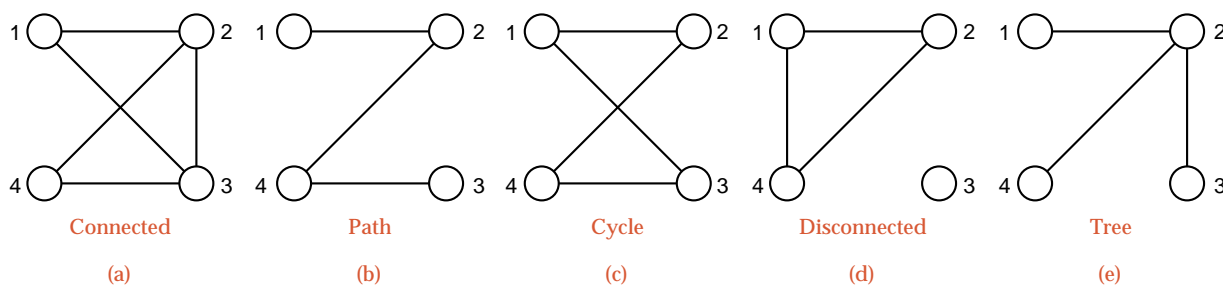


Figure 12.2. Various kinds of undirected graphs

### 12.1.2 Undirected Graphs



paths, cycles, connected

Several kinds of undirected graphs are shown in Figure 12.2. Two vertices in an undirected graph are called **adjacent** if there is an edge from one to the other. Hence, in the undirected graph of part (a), vertices 1 and 2 are adjacent, as are 3 and 4, but 1 and 4 are not adjacent. A **path** is a sequence of distinct vertices, each adjacent to the next. Part (b) shows a path. A **cycle** is a path containing at least three vertices such that the last vertex on the path is adjacent to the first. Part (c) shows a cycle. A graph is called **connected** if there is a path from any vertex to any other vertex; parts (a), (b), and (c) show connected graphs, and part (d) shows a disconnected graph. If a graph is disconnected, we shall refer to a maximal subset of connected vertices as a **component**. For example, the disconnected graph in part (c) has two components: The first consists of vertices 1, 2, and 4, and the second has just the vertex 3. Part (e) of Figure 12.2 shows a connected graph with no cycles. You will notice that this graph is, in fact, a tree, and we take this property as the definition: A **free tree** is defined as a connected undirected graph with no cycles.

free tree

### 12.1.3 Directed Graphs

directed paths and cycles

For directed graphs, we can make similar definitions. We require all edges in a path or a cycle to have the same direction, so that following a path or a cycle means always moving in the direction indicated by the arrows. Such a path (cycle) is called a **directed** path (cycle). A directed graph is called **strongly connected** if there is a directed path from any vertex to any other vertex. If we suppress the direction of the edges and the resulting undirected graph is connected, we call the directed graph **weakly connected**. Figure 12.3 illustrates directed cycles, strongly connected directed graphs, and weakly connected directed graphs.

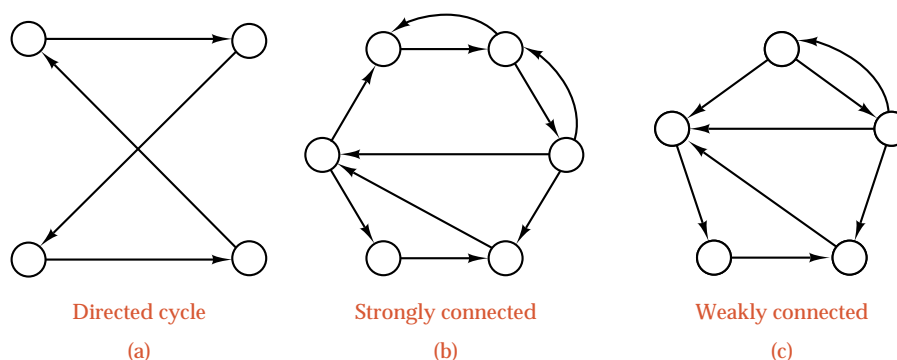


Figure 12.3. Examples of directed graphs

*multiple edges* The directed graphs in parts (b) and (c) of Figure 12.3 show pairs of vertices with directed edges going both ways between them. Since directed edges are ordered pairs and the ordered pairs  $(v, w)$  and  $(w, v)$  are distinct if  $v \neq w$ , such pairs of edges are permissible in directed graphs. Since the corresponding unordered pairs are not distinct, however, in an undirected graph there can be at most one edge connecting a pair of vertices. Similarly, since the vertices on an edge are required to be distinct, there can be no edge from a vertex to itself. We should remark, however, that (although we shall not do so) sometimes these requirements are relaxed to allow multiple edges connecting a pair of vertices and self-loops connecting a vertex to itself.

*self-loops*

## 12.2 COMPUTER REPRESENTATION

If we are to write programs for solving problems concerning graphs, then we must first find ways to represent the mathematical structure of a graph as some kind of data structure. There are several methods in common use, which differ fundamentally in the choice of abstract data type used to represent graphs, and there are several variations depending on the implementation of the abstract data type. In other words, we begin with one mathematical system (a *graph*), then we study how it can be described in terms of abstract data types (*sets*, *tables*, and *lists* can all be used, as it turns out), and finally we choose implementations for the abstract data type that we select.

### 12.2.1 The Set Representation

Graphs are defined in terms of sets, and it is natural to look first to sets to determine their representation as data. First, we have a set of vertices, and, second, we have the edges as a set of pairs of vertices. Rather than attempting to represent this set of pairs directly, we divide it into pieces by considering the set of edges attached to each vertex separately. In other words, we can keep track of all the edges in the graph by keeping, for all vertices  $v$  in the graph, the set  $E_v$  of edges containing  $v$ , or, equivalently, the set  $A_v$  of all vertices adjacent to  $v$ . In fact, we can use this idea to produce a new, equivalent definition of a graph:



## Definition

A **digraph**  $G$  consists of a set  $V$ , called the **vertices** of  $G$ , and, for all  $v \in V$ , a subset  $A_v$  of  $V$ , called the set of vertices **adjacent** to  $v$ .

From the subsets  $A_v$  we can reconstruct the edges as ordered pairs by the following rule: The pair  $(v, w)$  is an edge if and only if  $w \in A_v$ . It is easier, however, to work with sets of vertices than with pairs. This new definition, moreover, works for both directed and undirected graphs. The graph is undirected means that it satisfies the following symmetry property:  $w \in A_v$  implies  $v \in A_w$  for all  $v, w \in V$ . This property can be restated in less formal terms: It means that an undirected edge between  $v$  and  $w$  can be regarded as made up of two directed edges, one from  $v$  to  $w$  and the other from  $w$  to  $v$ .

### 1. Implementation of Sets

There are two general ways for us to implement sets of vertices in data structures and algorithms. One way is to represent the set as a *list* of its elements; this method we shall study presently. The other implementation, often called a **bit string**, keeps a Boolean value for each potential element of the set to indicate whether or not it is in the set. For simplicity, we shall consider that the potential elements of a set are indexed with the integers from 0 to  $\text{max\_set} - 1$ , where  $\text{max\_set}$  denotes the maximum number of elements that we shall allow. This latter strategy is easily implemented either with the standard template library class `std::bitset<max_set>` or with our own class template that uses a template parameter to give the maximal number of potential members of a set.

sets as Boolean arrays

```
template <int max_set>
struct Set {
    bool is_element[max_set];
};
```

We can now fully specify a first representation of a graph:

first implementation:  
sets

```
template <int max_size>
class Digraph {
    int count; // number of vertices, at most max_size
    Set<max_size> neighbors[max_size];
};
```

In this implementation, the vertices are identified with the integers from 0 to  $\text{count} - 1$ . If  $v$  is such an integer, the array entry `neighbors[v]` is the set of all vertices adjacent to the vertex  $v$ .

### 2. Adjacency Tables

sets as arrays

In the foregoing implementation, the structure `Set` is essentially implemented as an array of `bool` entries. Each entry indicates whether or not the corresponding vertex is a member of the set. If we substitute this array for a set of neighbors, we find that the array `neighbors` in the definition of class `Graph` can be changed to an array of arrays, that is, to a two-dimensional array, as follows:

*second implementation: adjacency table*

```
template <int max_size>
class Digraph {
    int count; // number of vertices, at most max_size
    bool adjacency[max_size][max_size];
};
```

*meaning* The adjacency table has a natural interpretation:  $\text{adjacency}[v][w]$  is true if and only if vertex  $v$  is adjacent to vertex  $w$ . If the graph is directed, we interpret  $\text{adjacency}[v][w]$  as indicating whether or not the edge from  $v$  to  $w$  is in the graph. If the graph is undirected, then the adjacency table must be symmetric; that is,  $\text{adjacency}[v][w] = \text{adjacency}[w][v]$  for all  $v$  and  $w$ . The representation of a graph by adjacency sets and by an adjacency table is illustrated in Figure 12.4.

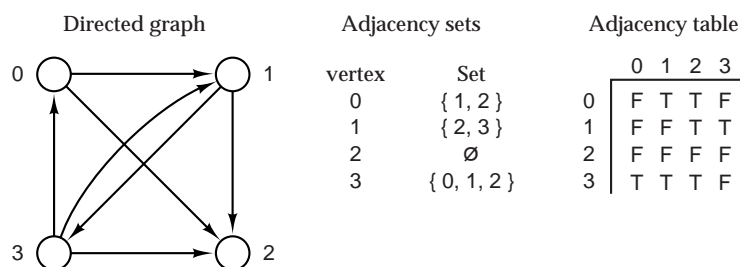


Figure 12.4. Adjacency set and an adjacency table

## 12.2.2 Adjacency Lists

Another way to represent a set is as a *list* of its elements. For representing a graph, we shall then have both a list of vertices and, for each vertex, a list of adjacent vertices. We can consider implementations of graphs that use either contiguous lists or simply linked lists. For more advanced applications, however, it is often useful to employ more sophisticated implementations of lists as binary or multiway search trees or as heaps. Note that, by identifying vertices with their indices in the previous representations, we have *ipso facto* implemented the vertex set as a contiguous list, but now we should make a deliberate choice concerning the use of contiguous or linked lists.



### 1. List-based Implementation

We obtain list-based implementations by replacing our earlier sets of neighbors by lists. This implementation can use either contiguous or linked lists. The contiguous version is illustrated in part (b) of Figure 12.5, and the linked version is illustrated in part (c) of Figure 12.5.

*third implementation: lists*

```
typedef int Vertex;
template <int max_size>
class Digraph {
    int count; // number of vertices, at most max_size
    List<Vertex> neighbors[max_size];
};
```

## 2. Linked Implementation

Greatest flexibility is obtained by using linked objects for both the vertices and the adjacency lists. This implementation is illustrated in part (a) of Figure 12.5 and results in a definition such as the following:

```

fourth      class Edge;           // forward declaration
implementation: class Vertex {
linked vertices and   Edge *first_edge;       // start of the adjacency list
edges         Vertex *next_vertex; // next vertex on the linked list
};
class Edge {
  Vertex *end_point; // vertex to which the edge points
  Edge *next_edge;   // next edge on the adjacency list
};
class Digraph {
  Vertex *first_vertex; // header for the list of vertices
};

```

### 12.2.3 Information Fields

Many applications of graphs require not only the adjacency information specified in the various representations but also further information specific to each vertex or each edge. In the linked representations, this information can be included as additional members within appropriate records, and, in the contiguous representations, it can be included by making array entries into records. An especially important case is that of a **network**, which is defined as a graph in which a numerical **weight** is attached to each edge. For many algorithms on networks, the best representation is an adjacency table, where the entries are the weights rather than Boolean values. We shall return to this topic later in the chapter.

*networks, weights*

## 12.3 GRAPH TRAVERSAL

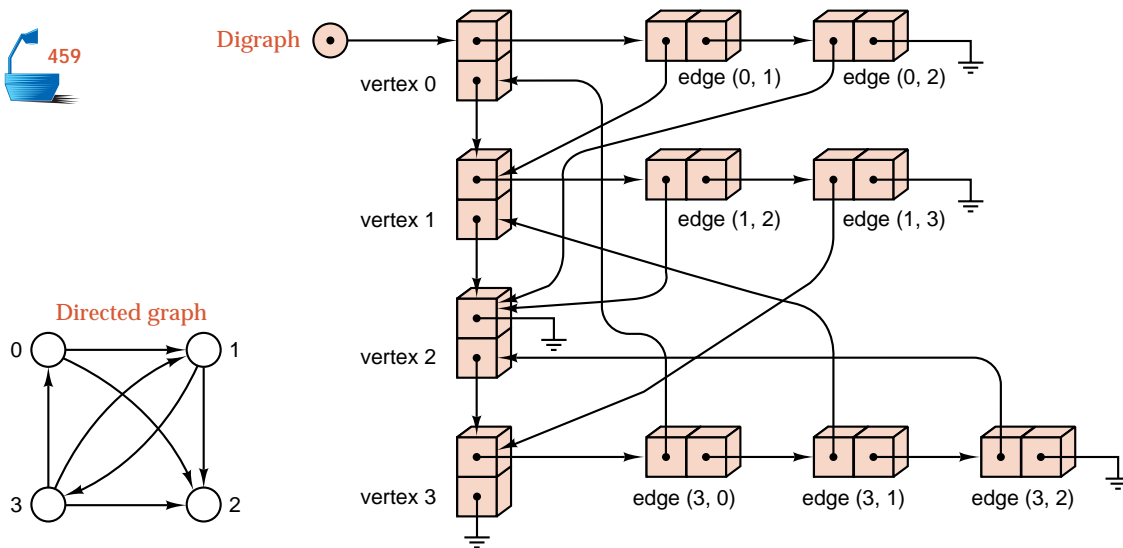
---

### 12.3.1 Methods



*depth-first*

In many problems, we wish to investigate all the vertices in a graph in some systematic order, just as with binary trees, where we developed several systematic traversal methods. In tree traversal, we had a root vertex with which we generally started; in graphs, we often do not have any one vertex singled out as special, and therefore the traversal may start at an arbitrary vertex. Although there are many possible orders for visiting the vertices of the graph, two methods are of particular importance. **Depth-first traversal** of a graph is roughly analogous to preorder traversal of an ordered tree. Suppose that the traversal has just visited a vertex  $v$ , and let  $w_1, w_2, \dots, w_k$  be the vertices adjacent to  $v$ . Then we shall next visit  $w_1$  and keep  $w_2, \dots, w_k$  waiting. After visiting  $w_1$ , we traverse all the vertices to which



(a) Linked lists

count = 4

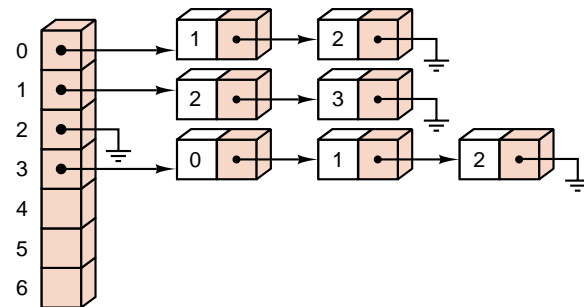
vertex adjacency list

0	1	2	-	-	-	-
1	2	3	-	-	-	-
2	-	-	-	-	-	-
3	0	1	2	-	-	-
4	-	-	-	-	-	-
5	-	-	-	-	-	-
6	-	-	-	-	-	-

(b) Contiguous lists

count = 4

first\_edge



(c) Mixed

Figure 12.5. Implementations of a graph with lists

*breadth-first* it is adjacent before returning to traverse  $w_2, \dots, w_k$ . **Breadth-first traversal** of a graph is roughly analogous to level-by-level traversal of an ordered tree. If the traversal has just visited a vertex  $v$ , then it next visits *all* the vertices adjacent to  $v$ , putting the vertices adjacent to these in a waiting list to be traversed after all vertices adjacent to  $v$  have been visited. Figure 12.6 shows the order of visiting the vertices of one graph under both depth-first and breadth-first traversals.

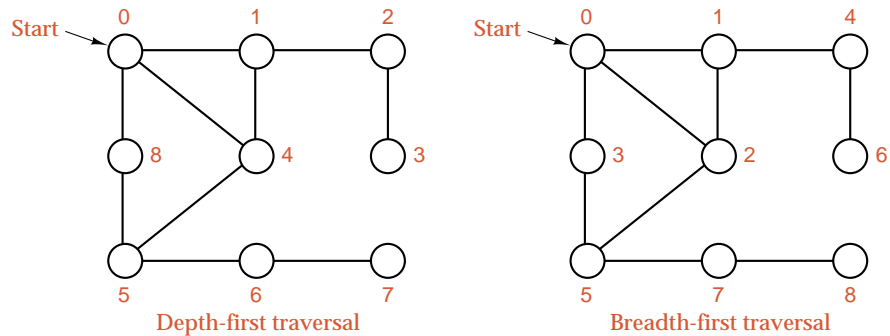


Figure 12.6. Graph traversal

### 12.3.2 Depth-First Algorithm

Depth-first traversal is naturally formulated as a recursive algorithm. Its action, when it reaches a vertex  $v$ , is:

```
visit(v);
for (each vertex w adjacent to v)
    traverse(w);
```

*complications*

In graph traversal, however, two difficulties arise that cannot appear for tree traversal. First, the graph may contain cycles, so our traversal algorithm may reach the same vertex a second time. To prevent infinite recursion, we therefore introduce a `bool` array `visited`. We set `visited[v]` to `true` immediately before visiting  $v$ , and check the value of `visited[w]` before processing  $w$ . Second, the graph may not be connected, so the traversal algorithm may fail to reach all vertices from a single starting point. Hence we enclose the action in a loop that runs through all vertices to make sure that we visit all components of the graph. With these refinements, we obtain the following outline of depth-first traversal. Further details depend on the choice of implementation of graphs and vertices, and we postpone them to application programs.



*main function outline*

```
template <int max_size>
void Digraph<max_size>::depth_first(void (*visit)(Vertex &)) const
/* Post: The function *visit has been performed at each vertex of the Digraph in
depth-first order.
Uses: Method traverse to produce the recursive depth-first order. */
{
    bool visited[max_size];
    Vertex v;
    for (all v in G) visited[v] = false;
    for (all v in G) if (!visited[v])
        traverse(v, visited, visit);
}
```

The recursion is performed in an auxiliary function `traverse`. Since `traverse` needs access to the internal structure of a graph, it should be a member function of the class `Digraph`. Moreover, since `traverse` is merely an auxiliary function, used in the construction of the method `depth_first`, it should be private to the class `Digraph`.

*recursive traversal  
outline*

```
template <int max_size>
void Digraph<max_size> :: traverse(Vertex &v, bool visited[ ],
                                void (*visit)(Vertex &)) const
/* Pre:  v is a vertex of the Digraph.
   Post: The depth-first traversal, using function *visit, has been completed for v
         and for all vertices that can be reached from v.
   Uses: traverse recursively. */
{ Vertex w;
  visited[v] = true;
  (*visit)(v);
  for (all w adjacent to v)
    if (!visited[w])
      traverse(w, visited, visit);
}
```

### 12.3.3 Breadth-First Algorithm



*stacks and queues*

Since using recursion and programming with stacks are essentially equivalent, we could formulate depth-first traversal with a Stack, pushing all unvisited vertices adjacent to the one being visited onto the Stack and popping the Stack to find the next vertex to visit. The algorithm for breadth-first traversal is quite similar to the resulting algorithm for depth-first traversal, except that a Queue is needed instead of a Stack. Its outline follows.

*breadth-first traversal  
outline*

```
template <int max_size>
void Digraph<max_size> :: breadth_first(void (*visit)(Vertex &)) const
/* Post: The function *visit has been performed at each vertex of the Digraph in
         breadth-first order.
   Uses: Methods of class Queue. */
{ Queue q;
  bool visited[max_size];
  Vertex v, w, x;
  for (all v in G) visited[v] = false;
  for (all v in G)
    if (!visited[v]) {
      q.append(v);
      while (!q.empty()){
        q.retrieve(w);
        if (!visited[w]) {
          visited[w] = true;
          (*visit)(w);
          for (all x adjacent to w)
            q.append(x);
        }
        q.serve();
      }
    }
}
```

## 12.4 TOPOLOGICAL SORTING

### 12.4.1 The Problem

#### topological order

If  $G$  is a directed graph with no directed cycles, then a **topological order** for  $G$  is a sequential listing of all the vertices in  $G$  such that, for all vertices  $v, w \in G$ , if there is an edge from  $v$  to  $w$ , then  $v$  precedes  $w$  in the sequential listing. Throughout this section, we shall consider only directed graphs that have no directed cycles. The term **acyclic** is often used to mean that a graph has no cycles.

#### applications

Such graphs arise in many problems. As a first application of topological order, consider the courses available at a university as the vertices of a directed graph, where there is an edge from one course to another if the first is a prerequisite for the second. A topological order is then a listing of all the courses such that all prerequisites for a course appear before it does. A second example is a glossary of technical terms that is ordered so that no term is used in a definition before it is itself defined. Similarly, the author of a textbook uses a topological order for the topics in the book. Two different topological orders of a directed graph are shown in Figure 12.7. As an example of algorithms for graph traversal, we shall develop functions that produce a topological ordering of the vertices of a directed graph that has no cycles. We shall develop two methods: first, using depth-first traversal, and, then, using breadth-first traversal. Both methods apply to an object of a class `Digraph` that uses the list-based implementation. Thus we shall assume the following class specification:



463

#### graph representation

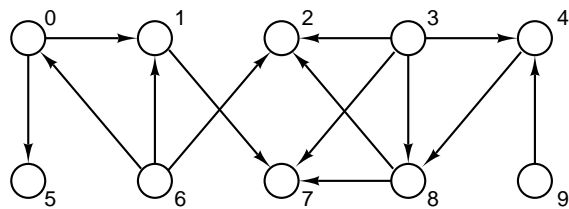


464

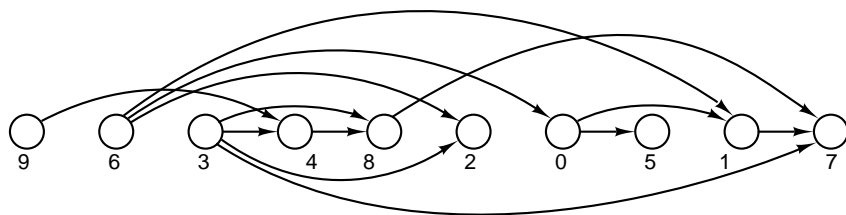
```
typedef int Vertex;

template <int graph_size>
class Digraph {
public:
    Digraph();
    void read();
    void write();
    // methods to do a topological sort
    void depth_sort(List<Vertex> &topological_order);
    void breadth_sort(List<Vertex> &topological_order);
private:
    int count;
    List <Vertex> neighbors[graph_size];
    void recursive_depth_sort(Vertex v, bool visited[ ],
                               List<Vertex> &topological_order);
};
```

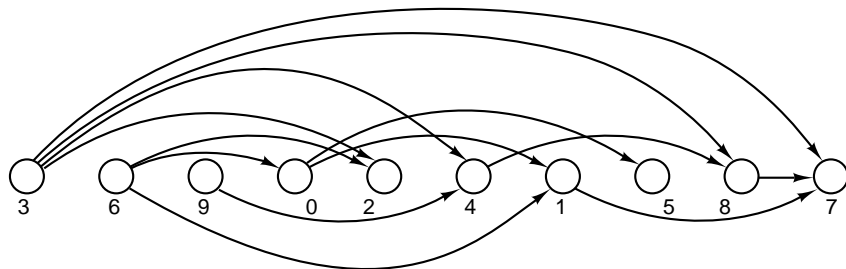
The auxiliary member function `recursive_depth_sort` will be used by the method `depth_sort`. Both sorting methods should create a list giving a topological order of the vertices.



Directed graph with no directed cycles



Depth-first ordering



Breadth-first ordering

Figure 12.7. Topological orderings of a directed graph

### 12.4.2 Depth-First Algorithm

*strategy* In a topological order, each vertex must appear before all the vertices that are its successors in the directed graph. For a depth-first topological ordering, we therefore start by finding a vertex that has no successors and place it last in the order. After we have, by recursion, placed all the successors of a vertex into the topological order, then we can place the vertex itself in a position before any of its successors. Since we first order the last vertices, we can repeatedly add vertices to the beginning of the List `topological_order`. The method is a direct implementation of the general depth first traversal procedure developed in the last section.



```

template <int graph_size>
void Digraph<graph_size> :: depth_sort(List<Vertex> &topological_order)
/* Post: The vertices of the Digraph are placed into List topological_order with a
depth-first traversal of those vertices that do not belong to a cycle.
Uses: Methods of class List, and function recursive_depth_sort to perform depth-
first traversal. */
{
    bool visited[graph_size];
    Vertex v;
    for (v = 0; v < count; v++) visited[v] = false;
    topological_order.clear();
    for (v = 0; v < count; v++)
        if (!visited[v]) // Add v and its successors into topological order.
            recursive_depth_sort(v, visited, topological_order);
}

```

The auxiliary function `recursive_depth_sort` that performs the recursion, based on the outline for the general function `traverse`, first places all the successors of `v` into their positions in the topological order and then places `v` into the order.

```

template <int graph_size>
void Digraph<graph_size> :: recursive_depth_sort(Vertex v, bool *visited,
List<Vertex> &topological_order)
/* Pre: Vertex v of the Digraph does not belong to the partially completed List
topological_order.
Post: All the successors of v and finally v itself are added to topological_order
with a depth-first search.
Uses: Methods of class List and the function recursive_depth_sort. */
{
    visited[v] = true;
    int degree = neighbors[v].size();
    for (int i = 0; i < degree; i++) {
        Vertex w; // A (neighboring) successor of v
        neighbors[v].retrieve(i, w);
        if (!visited[w]) // Order the successors of w.
            recursive_depth_sort(w, visited, topological_order);
    }
    topological_order.insert(0, v); // Put v into topological_order.
}

```

*performance* Since this algorithm visits each node of the graph exactly once and follows each edge once, doing no searching, its running time is  $O(n + e)$ , where  $n$  is the number of vertices and  $e$  is the number of edges in the graph.

### 12.4.3 Breadth-First Algorithm

*method* In a breadth-first topological ordering of a directed graph with no cycles, we start by finding the vertices that should be first in the topological order and then apply

the fact that every vertex must come before its successors in the topological order. The vertices that come first are those that are not successors of any other vertex. To find these, we set up an array `predecessor_count` whose entry at index  $v$  is the number of immediate predecessors of vertex  $v$ . The vertices that are not successors are those with no predecessors. We therefore initialize the breadth-first traversal by placing these vertices into a Queue of vertices to be visited. As each vertex is visited, it is removed from the Queue, assigned the next available position in the topological order (starting at the beginning of the order), and then removed from further consideration by reducing the predecessor count for each of its immediate successors by one. When one of these counts reaches zero, all predecessors of the corresponding vertex have been visited, and the vertex itself is then ready to be processed, so it is added to the Queue. We thereby obtain the following function:



```

template <int graph_size>
void Digraph<graph_size> :: breadth_sort(List<Vertex> &topological_order)
/* Post: The vertices of the Digraph are arranged into the List topological_order
    which is found with a breadth-first traversal of those vertices that do not
    belong to a cycle.
    Uses: Methods of classes Queue and List. */
{
    topological_order.clear();
    Vertex v, w;
    int predecessor_count[graph_size];
    for (v = 0; v < count; v++) predecessor_count[v] = 0;
    for (v = 0; v < count; v++)
        for (int i = 0; i < neighbors[v].size(); i++) {
            // Loop over all edges v — w.
            neighbors[v].retrieve(i, w);
            predecessor_count[w]++;
        }
    Queue ready_to_process;
    for (v = 0; v < count; v++)
        if (predecessor_count[v] == 0)
            ready_to_process.append(v);
    while (!ready_to_process.empty()) {
        ready_to_process.retrieve(v);
        topological_order.insert(topological_order.size(), v);
        for (int j = 0; j < neighbors[v].size(); j++) { // Traverse successors of v.
            neighbors[v].retrieve(j, w);
            predecessor_count[w]--;
            if (predecessor_count[w] == 0)
                ready_to_process.append(w);
        }
        ready_to_process.serve();
    }
}

```

*performance*

This algorithm requires one of the packages for processing queues. The queue can be implemented in any of the ways described in Chapter 3 and Chapter 4. Since the entries in the Queue are to be vertices, we should add a specification: `typedef Vertex Queue_entry;` before the implementation of `breadth_sort`. As with depth-first traversal, the time required by the breadth-first function is  $O(n + e)$ , where  $n$  is the number of vertices and  $e$  is the number of edges in the directed graph.

## 12.5 A GREEDY ALGORITHM: SHORTEST PATHS

### 12.5.1 The Problem

*shortest path*

As another application of graphs, one requiring somewhat more sophisticated reasoning, we consider the following problem. We are given a directed graph  $G$  in which every edge has a nonnegative *weight* attached: In other words,  $G$  is a **directed network**. Our problem is to find a path from one vertex  $v$  to another  $w$  such that the sum of the weights on the path is as small as possible. We call such a path a **shortest path**, even though the weights may represent costs, time, or some quantity other than distance. We can think of  $G$  as a map of airline routes, for example, with each vertex representing a city and the weight on each edge the cost of flying from one city to the second. Our problem is then to find a routing from city  $v$  to city  $w$  such that the total cost is a minimum. Consider the directed graph shown in Figure 12.8. The shortest path from vertex 0 to vertex 1 goes via vertex 2 and has a total cost of 4, compared to the cost of 5 for the edge directly from 0 to 1 and the cost of 8 for the path via vertex 4.

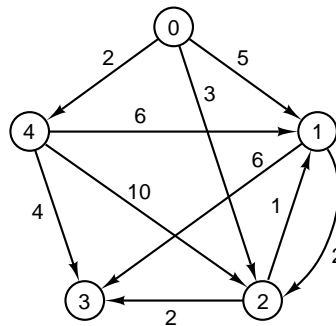


Figure 12.8. A directed graph with weights

*source*

It turns out that it is just as easy to solve the more general problem of starting at one vertex, called the **source**, and finding the shortest path to every other vertex, instead of to just one destination vertex. In our implementation, the source vertex will be passed as a parameter. Our problem then consists of finding the shortest path from vertex `source` to every vertex in the graph. We require that the weights are all nonnegative.

## 12.5.2 Method



468

*distance table**greedy algorithm**verification**end of proof*

The algorithm operates by keeping a set  $S$  of those vertices whose shortest distance from source is known. Initially, source is the only vertex in  $S$ . At each step, we add to  $S$  a remaining vertex for which the shortest path from source has been found. The problem is to determine which vertex to add to  $S$  at each step. Let us think of the vertices already in  $S$  as having been labeled with some color, and think of the edges making up the shortest paths from source to these vertices as also colored.

We shall maintain a table distance that gives, for each vertex  $v$ , the distance from source to  $v$  along a path all of whose edges are colored, except possibly the last one. That is, if  $v$  is in  $S$ , then  $\text{distance}[v]$  gives the shortest distance to  $v$  and all edges along the corresponding path are colored. If  $v$  is not in  $S$ , then  $\text{distance}[v]$  gives the length of the path from source to some vertex  $w$  in  $S$  plus the weight of the edge from  $w$  to  $v$ , and all the edges of this path except the last one are colored. The table distance is initialized by setting  $\text{distance}[v]$  to the weight of the edge from source to  $v$  if it exists and to infinity if not.

To determine what vertex to add to  $S$  at each step, we apply the **greedy** criterion of choosing the vertex  $v$  with the smallest distance recorded in the table distance, such that  $v$  is not already in  $S$ . We must prove that, for this vertex  $v$ , the distance recorded in distance really is the length of the shortest path from source to  $v$ . For suppose that there were a shorter path from source to  $v$ , such as shown in Figure 12.9. This path first leaves  $S$  to go to some vertex  $x$ , then goes on to  $v$  (possibly even reentering  $S$  along the way). But if this path is shorter than the colored path to  $v$ , then its initial segment from source to  $x$  is also shorter, so that the greedy criterion would have chosen  $x$  rather than  $v$  as the next vertex to add to  $S$ , since we would have had  $\text{distance}[x] < \text{distance}[v]$ .

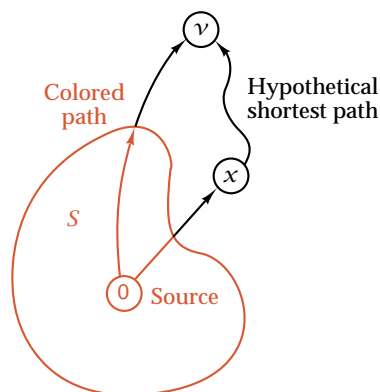


Figure 12.9. Finding a shortest path

When we add  $v$  to  $S$ , we think of  $v$  as now colored and also color the shortest path from source to  $v$  (every edge of which except the last was actually already colored). Next, we must update the entries of distance by checking, for each vertex  $w$  not in  $S$ , whether a path through  $v$  and then directly to  $w$  is shorter than the previously recorded distance to  $w$ . That is, we replace  $\text{distance}[w]$  by  $\text{distance}[v]$  plus the weight of the edge from  $v$  to  $w$  if the latter quantity is smaller.

*maintain the invariant*

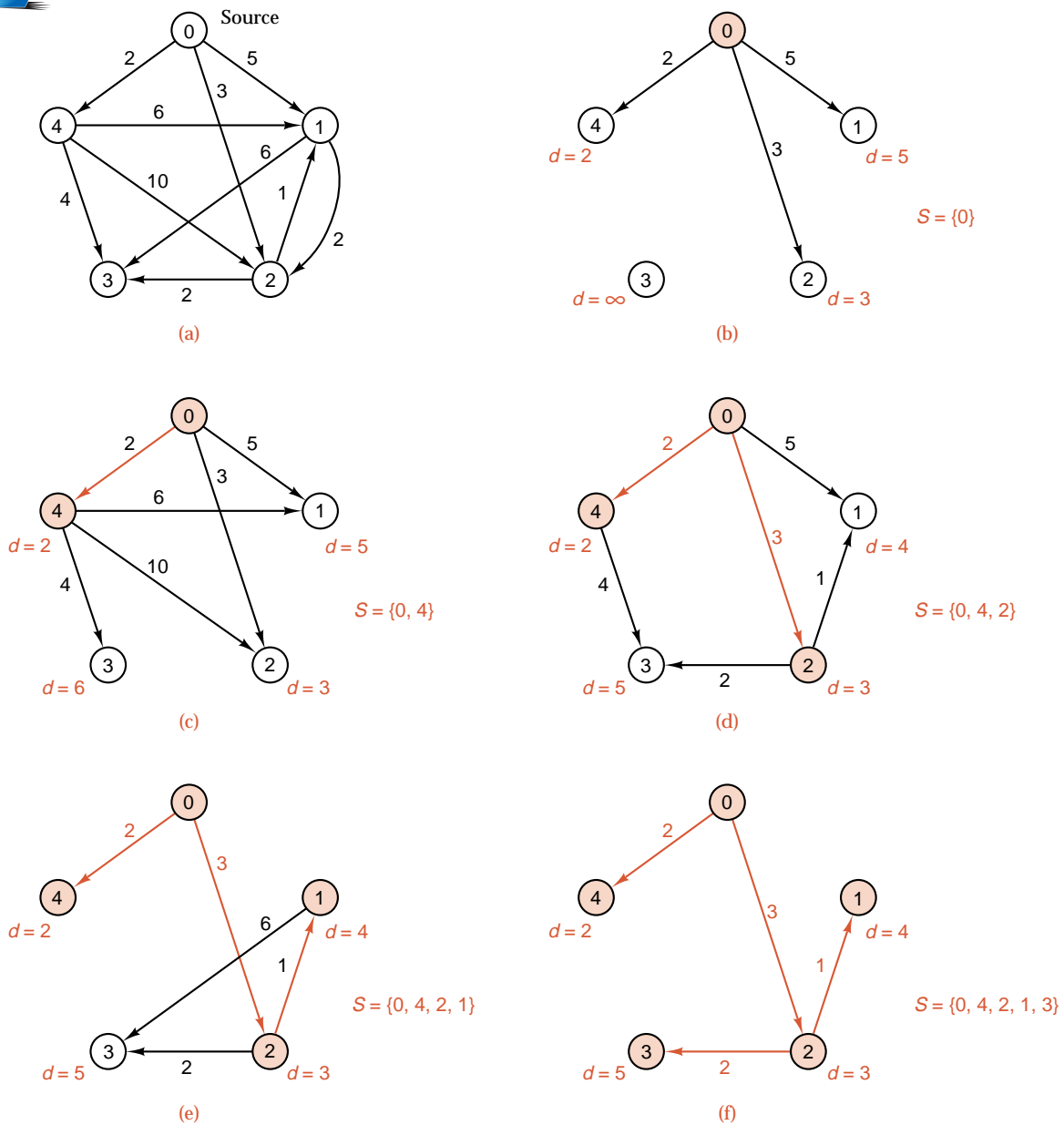


Figure 12.10. Example of shortest paths

### 12.5.3 Example

Before writing a formal function incorporating this method, let us work through the example shown in Figure 12.10. For the directed graph shown in part (a), the initial situation is shown in part (b): The set  $S$  (colored vertices) consists of source, vertex 0, alone, and the entries of the distance table distance are shown as numbers

in color beside the other vertices. The distance to vertex 4 is shortest, so 4 is added to  $S$  in part (c), and the distance `distance[3]` is updated to the value 6. Since the distances to vertices 1 and 2 via vertex 4 are greater than those already recorded in  $T$ , their entries remain unchanged. The next closest vertex to source is vertex 2, and it is added in part (d), which also shows the effect of updating the distances to vertices 1 and 3, whose paths via vertex 2 are shorter than those previously recorded. The final two steps, shown in parts (e) and (f), add vertices 1 and 3 to  $S$  and yield the paths and distances shown in the final diagram.

### 12.5.4 Implementation

For the sake of writing a function to embody this algorithm for finding shortest distances, we must choose an implementation of the directed graph. Use of the adjacency-table implementation facilitates random access to all the vertices of the graph, as we need for this problem. Moreover, by storing the weights in the table, we can use the table to give weights as well as adjacencies. In the following Digraph specification, we add a template parameter to allow clients to specify the type of weights to be used. For example, a client using our class Digraph to model a network of airline routes might wish to use either integer or real weights for the cost of an airline route.



```
template <class Weight, int graph_size>
class Digraph {
public:
    // Add a constructor and methods for Digraph input and output.
    void set_distances(Vertex source, Weight distance[ ]) const;
protected:
    int count;
    Weight adjacency[graph_size][graph_size];
};
```

The data member `count` records the number of vertices in a particular Digraph. In applications, we would need to flesh out this class by adding methods for input and output, but since these will not be needed in the implementation of the method `set_distances`, which calculates shortest paths, we shall leave the additional methods as exercises.

We shall assume that the class `Weight` has comparison operators. Moreover, we shall expect clients to declare a largest possible `Weight` value called `infinity`. For example, client code working with integer weights could make use of the information in the ANSI C++ standard library `<limits>` and use a global definition:

```
const Weight infinity = numeric_limits<int>::max();
```

We shall place the value `infinity` in any position of the adjacency table for which the corresponding edge does not exist. The method `set_distances` that we now write will calculate the table of closest distances into its output parameter `distance[ ]`.

*shortest distance  
procedure*

```

template <class Weight, int graph_size>
void Digraph<Weight, graph_size>::set_distances(Vertex source,
                                               Weight distance[ ]) const
/* Post: The array distance gives the minimal path weight from vertex source to
each vertex of the Digraph. */
{
    Vertex v, w;
    bool found[graph_size]; // Vertices found in S
    for (v = 0; v < count; v++) {
        found[v] = false;
        distance[v] = adjacency[source][v];
    }
    found[source] = true; // Initialize with vertex source alone in the set S.
    distance[source] = 0;
    for (int i = 0; i < count; i++) { // Add one vertex v to S on each pass.
        Weight min = infinity;
        for (w = 0; w < count; w++) if (!found[w])
            if (distance[w] < min) {
                v = w;
                min = distance[w];
            }
        found[v] = true;
        for (w = 0; w < count; w++) if (!found[w])
            if (min + adjacency[v][w] < distance[w])
                distance[w] = min + adjacency[v][w];
    }
}

```

*performance*

To estimate the running time of this function, we note that the main loop is executed  $n - 1$  times, where  $n$  is the number of vertices, and within the main loop are two other loops, each executed  $n - 1$  times, so these loops contribute a multiple of  $(n - 1)^2$  operations. Statements done outside the loops contribute only  $O(n)$ , so the running time of the algorithm is  $O(n^2)$ .

## 12.6 MINIMAL SPANNING TREES

---

### 12.6.1 The Problem

The shortest-path algorithm of the last section applies without change to networks and graphs as well as to directed networks and digraphs. For example, in Figure 12.11 we illustrate the result of its application to find shortest paths (shown in color) from a source vertex, labeled 0, to the other vertices of a network.

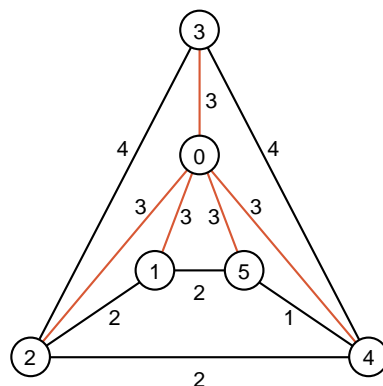


Figure 12.11. Finding shortest paths in a network

If the original network is based on a connected graph  $G$ , then the shortest paths from a particular source vertex link that source to all other vertices in  $G$ . Therefore, as we can see in Figure 12.11, if we combine the computed shortest paths together, we obtain a tree that links up all the vertices of  $G$ . In other words, we obtain a connected tree that is built up out of all the vertices and some of the edges of  $G$ . We shall refer to any such tree as a **spanning tree** of  $G$ . As in the previous section, we can think of a network on a graph  $G$  as a map of airline routes, with each vertex representing a city and the weight on each edge the cost of flying from one city to the second. A spanning tree of  $G$  represents a set of routes that will allow passengers to complete any conceivable trip between cities. Of course, passengers will frequently have to use several flights to complete journeys. However, this inconvenience for the passengers is offset by lower costs for the airline and cheaper tickets. In fact, spanning trees have been commonly adopted by airlines as hub-spoke route systems. If we imagine the network of Figure 12.11 as representing a hub-spoke system, then the source vertex corresponds to the hub airport, and the paths emerging from this vertex are the spoke routes. It is important for an airline running a hub-spoke system to minimize its expenses by choosing a system of routes whose costs have a minimal sum. For example, in Figure 12.12, where a pair of spanning trees of a network are illustrated with colored edges, an airline would prefer the second spanning tree, because the sum of its labels is smaller. To model an optimal hub-spoke system, we make the following definition:

**Definition**  
*minimal spanning tree*

A **minimal spanning tree** of a connected network is a spanning tree such that the sum of the weights of its edges is as small as possible.

Although it is not difficult to compare the two spanning trees of Figure 12.12, it is much harder to see whether there are any other, cheaper spanning trees. Our problem is to devise a method that determines a minimal spanning tree of a connected network.

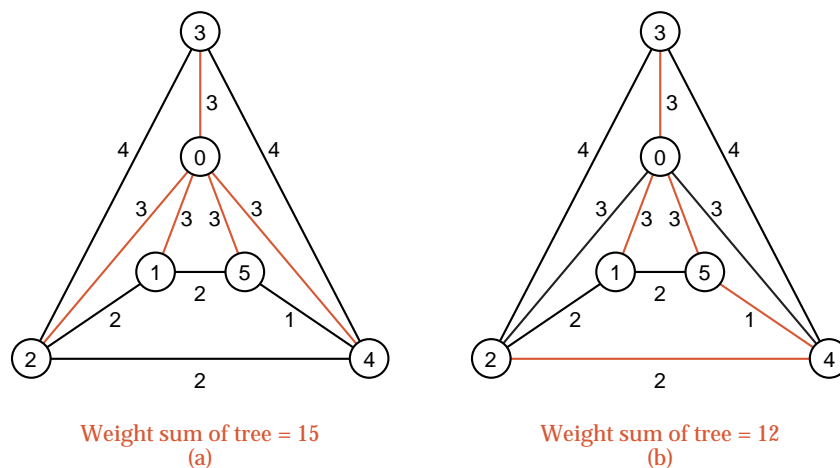


Figure 12.12. Two spanning trees in a network

## 12.6.2 Method



We already know an algorithm for finding a spanning tree of a connected graph, since the shortest path algorithm will do this. It turns out that we can make a small change to our shortest path algorithm to obtain a method, first implemented in 1957 by R. C. PRIM, that finds a minimal spanning tree.

We start out by choosing a starting vertex, that we call source, and, as we proceed through the method, we keep a set  $X$  of those vertices whose paths to the source in the minimal spanning tree that we are building have been found. We also need to keep track of the set  $Y$  of edges that link the vertices in  $X$  in the tree under construction. Thus, over the course of time, we can visualize the vertices in  $X$  and edges in  $Y$  as making up a small tree that grows to become our final spanning tree. Initially, source is the only vertex in  $X$ , and the edge set  $Y$  is empty. At each step, we add an additional vertex to  $X$ : This vertex is chosen so that an edge back to  $X$  has as small as possible a weight. This minimal edge back to  $X$  is added to  $Y$ .

It is quite tricky to prove that Prim's algorithm does give a minimal spanning tree, and we shall postpone this verification until the end of this section. However, we can understand the selection of the new vertex that we add to  $X$  and the new edge that we add to  $Y$  by noting that eventually we must incorporate an edge linking  $X$  to the other vertices of our network into the spanning tree that we are building. The edge chosen by Prim's criterion provides the cheapest way to accomplish this linking, and so according to the greedy criterion, we should use it. When we come to implement Prim's algorithm, we shall maintain a list of vertices that belong to  $X$  as the entries of a Boolean array component. It is convenient for us to store the edges in  $Y$  as the edges of a graph that will grow to give the output tree from our program.

*neighbor table*  
*distance table*

We shall maintain an auxiliary table *neighbor* that gives, for each vertex  $v$ , the vertex of  $X$  whose edge to  $v$  has minimal cost. It is convenient to maintain a second table *distance* that records these minimal costs. If a vertex  $v$  is not joined by an edge to  $X$  we shall record its distance as the value infinity. The table *neighbor*

is initialized by setting `neighbor[v]` to `source` for all vertices  $v$ , and `distance` is initialized by setting `distance[v]` to the weight of the edge from `source` to  $v$  if it exists and to infinity if not.

To determine what vertex to add to  $X$  at each step, we choose the vertex  $v$  with the smallest value recorded in the table `distance`, such that  $v$  is not already in  $X$ . After this we must update our tables to reflect the change that we have made to  $X$ . We do this by checking, for each vertex  $w$  not in  $X$ , whether there is an edge linking  $v$  and  $w$ , and if so, whether this edge has a weight less than `distance[w]`. In case there is an edge  $(v, w)$  with this property, we reset `neighbor[w]` to  $v$  and `distance[w]` to the weight of the edge.

*maintain the invariant*



For example, let us work through the network shown in part (a) of Figure 12.13. The initial situation is shown in part (b): The set  $X$  (colored vertices) consists of `source` alone, and for each vertex  $w$  the vertex `neighbor[w]` is visualized by following any arrow emerging from  $w$  in the diagram. (The value of `distance[w]` is the weight of the corresponding edge.) The distance to vertex 1 is among the shortest, so 1 is added to  $X$  in part (c), and the entries in tables `distance` and `neighbor` are updated for vertices 2 and 5. The other entries in these tables remain unchanged. Among the next closest vertices to  $X$  is vertex 2, and it is added in part (d), which also shows the effect of updating the `distance` and `neighbor` tables. The final three steps are shown in parts (e), (f), and (g).

### 12.6.3 Implementation



To implement Prim's algorithm, we must begin by choosing a C++ class to represent a network. The similarity of the algorithm to the shortest path algorithm of the last section suggests that we should base a class `Network` on our earlier class `Digraph`.

```
template <class Weight, int graph_size>
class Network: public Digraph<Weight, graph_size> {
public:
    Network();
    void read(); // overridden method to enter a Network
    void make_empty(int size = 0);
    void add_edge(Vertex v, Vertex w, Weight x);
    void minimal_spanning(Vertex source,
                          Network<Weight, graph_size> &tree) const;
};
```

Here, we have overridden an input method, `read`, to make sure that the weight of any edge  $(v, w)$  matches that of the edge  $(w, v)$ : In this way, we preserve our data structure from the potential corruption of undirected edges. The new method `make_empty(int size)` creates a `Network` with `size` vertices and no edges. The other new method, `add_edge`, adds an edge with a specified weight to a `Network`. Just as in the last section, we shall assume that the class `Weight` has comparison operators. Moreover, we shall expect clients to declare a largest possible `Weight` value called `infinity`. The method `minimal_spanning` that we now write will calculate a minimal spanning tree into its output parameter `tree`. Although the method can only compute a spanning tree when applied to a connected `Network`, it will always

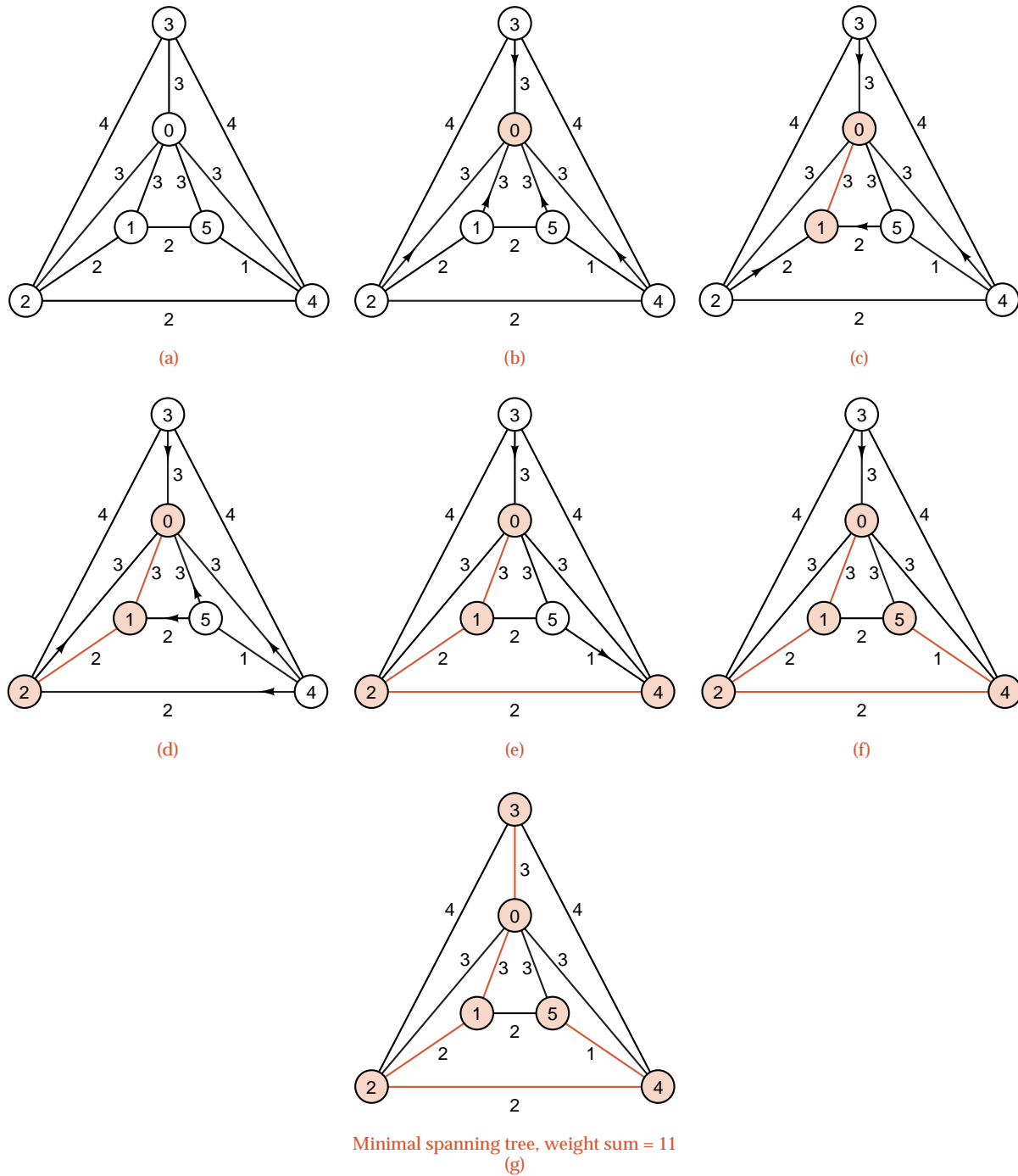


Figure 12.13. Example of Prim's algorithm



compute a spanning tree for the connected component determined by the vertex source in a Network.

```

template <class Weight, int graph_size>
void Network < Weight, graph_size > ::minimal_spanning(Vertex source,
    Network<Weight, graph_size> &tree) const
    /* Post: The Network tree contains a minimal spanning tree for the connected
        component of the original Network that contains vertex source. */
{
    tree.make_empty(count);
    bool component[graph_size]; // Vertices in set X
    Weight distance[graph_size]; // Distances of vertices adjacent to X
    Vertex neighbor[graph_size]; // Nearest neighbor in set X
    Vertex w;
    for (w = 0; w < count; w++) {
        component[w] = false;
        distance[w] = adjacency[source][w];
        neighbor[w] = source;
    }
    component[source] = true; // source alone is in the set X.
    for (int i = 1; i < count; i++) {
        Vertex v; // Add one vertex v to X on each pass.
        Weight min = infinity;
        for (w = 0; w < count; w++) if (!component[w])
            if (distance[w] < min) {
                v = w;
                min = distance[w];
            }
        if (min < infinity) {
            component[v] = true;
            tree.add_edge(v, neighbor[v], distance[v]);
            for (w = 0; w < count; w++) if (!component[w])
                if (adjacency[v][w] < distance[w]) {
                    distance[w] = adjacency[v][w];
                    neighbor[w] = v;
                }
        }
        else break; // finished a component in disconnected graph
    }
}

```

**performance** To estimate the running time of this function, we note that the main loop is executed  $n - 1$  times, where  $n$  is the number of vertices, and within the main loop are two other loops, each executed  $n - 1$  times, so these loops contribute a multiple of  $(n - 1)^2$  operations. Statements done outside the loops contribute only  $O(n)$ , so the running time of the algorithm is  $O(n^2)$ .

### 12.6.4 Verification of Prim's Algorithm



We must prove that, for a connected graph  $G$ , the spanning tree  $S$  that is produced by Prim's algorithm has a smaller edge-weight sum than any other spanning tree of  $G$ . Prim's algorithm determines a sequence of edges  $s_1, s_2, \dots, s_n$  that make up the tree  $S$ . Here, as shown in Figure 12.14,  $s_1$  is the first edge added to the set  $Y$  in Prim's algorithm,  $s_2$  is the second edge added to  $Y$ , and so on.

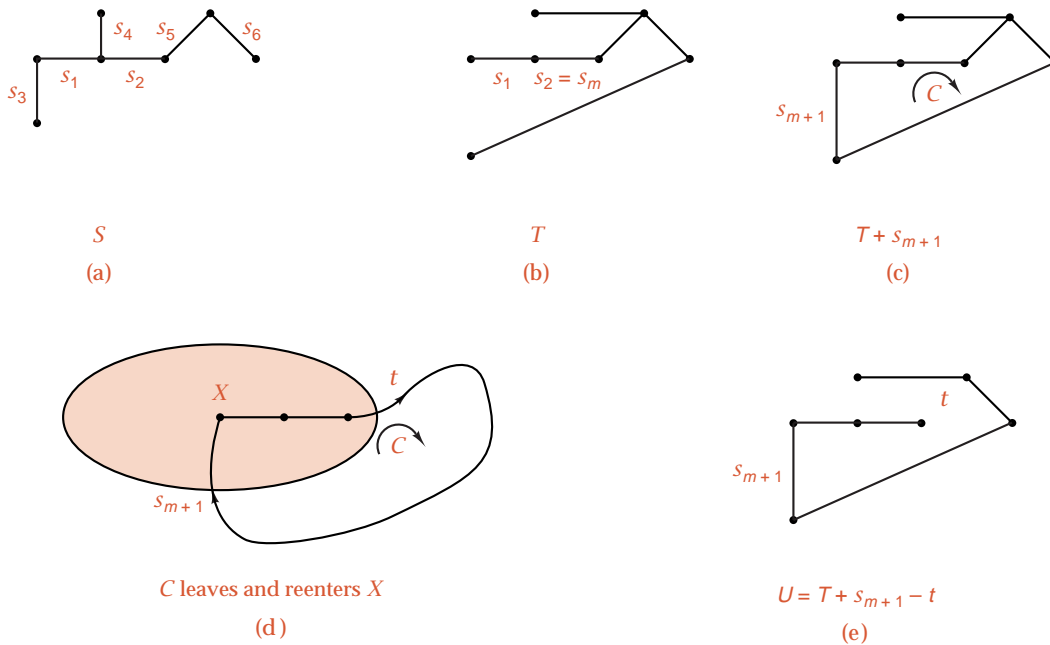


Figure 12.14. Optimality of the output tree of Prim's algorithm



*induction: base case*

*final case*

In order to show that  $S$  is a minimal spanning tree, we prove instead that if  $m$  is an integer with  $0 \leq m \leq n$ , then there is a minimal spanning tree that contains the edges  $s_i$  with  $i \leq m$ . We can work by induction on  $m$  to prove this result. The base case, where  $m = 0$ , is certainly true, since any minimal spanning tree does contain the requisite empty set of edges. Moreover, once we have completed the induction, the final case with  $m = n$  shows that there is a minimal spanning tree that contains all the edges of  $S$ , and therefore agrees with  $S$ . (Note that adding any edge to a spanning tree creates a cycle, so any spanning tree that does contain all the edges of  $S$  must be  $S$  itself). In other words, once we have completed our induction, we will have shown that  $S$  is a minimal spanning tree.

*inductive step*



We must therefore establish the inductive step, by showing that if  $m < n$  and  $T$  is a minimal spanning tree that contains the edges  $s_i$  with  $i \leq m$ , then there is a minimal spanning tree  $U$  with these edges and  $s_{m+1}$ . If  $s_{m+1}$  already belongs to  $T$ , we can simply set  $U = T$ , so we shall also suppose that  $s_{m+1}$  is not an edge of  $T$ . See part (b) of Figure 12.14.

Let us write  $X$  for the set of vertices of  $S$  belonging to the edges  $s_1, s_2, \dots, s_m$  and  $R$  for the set of remaining vertices of  $S$ . We recall that, in Prim's algorithm, the selected edge  $s_{m+1}$  links a vertex of  $X$  to  $R$ , and  $s_{m+1}$  is at least as cheap as any other edge between these sets. Consider the effect of adding  $s_{m+1}$  to  $T$ , as illustrated in part (c) of Figure 12.14. This addition must create a cycle  $C$ , since the connected network  $T$  certainly contains a multi-edge path linking the endpoints of  $s_{m+1}$ . The cycle  $C$  must contain an edge  $t \neq s_{m+1}$  that links  $X$  to  $R$ , since as we move once around the closed path  $C$  we must enter the set  $X$  exactly as many times as we leave it. See part (d) of Figure 12.14. Prim's algorithm guarantees that the weight of  $s_{m+1}$  is less than or equal to the weight of  $t$ . Therefore, the new spanning tree  $U$  (see part (e) of Figure 12.14), obtained from  $T$  by deleting  $t$  and adding  $s_{m+1}$ , has a weight sum no greater than that of  $T$ . We deduce that  $U$  must also be a minimal spanning tree of  $G$ , but  $U$  contains the sequence of edges  $s_1, s_2, \dots, s_m, s_{m+1}$ . This completes our induction.

*end of proof*

## 12.7 GRAPHS AS DATA STRUCTURES

In this chapter, we have studied a few applications of graphs, but we have hardly begun to scratch the surface of the broad and deep subject of graph algorithms. In many of these algorithms, graphs appear, as they have in this chapter, as mathematical structures capturing the essential description of a problem rather than as computational tools for its solution. Note that in this chapter we have spoken of graphs as mathematical structures, and not as data structures, for we have used graphs to formulate mathematical problems, and, to write algorithms, we have then implemented the graphs within data structures like tables and lists. Graphs, however, can certainly be regarded as data structures themselves, data structures that embody relationships among the data more complicated than those describing a list or a tree. Because of their generality and flexibility, graphs are powerful data structures that prove valuable in more advanced applications such as the design of data base management systems. Such powerful tools are meant to be used, of course, whenever necessary, but they must always be used with care so that their power is not turned to confusion. Perhaps the best safeguard in the use of powerful tools is to insist on regularity; that is, to use the powerful tools only in carefully defined and well-understood ways. Because of the generality of graphs, it is not always easy to impose this discipline on their use. In this world, nonetheless, irregularities will always creep in, no matter how hard we try to avoid them. It is the bane of the systems analyst and programmer to accommodate these irregularities while trying to maintain the integrity of the underlying system design. Irregularity even occurs in the very systems that we use as models for the data structures we devise, models such as the family trees whose terminology we have always used. An excellent illustration of what can happen is the following classic story, quoted by N. WIRTH<sup>1</sup> from a Zurich newspaper of July 1922.

I married a widow who had a grown-up daughter. My father, who visited us quite often, fell in love with my step-daughter and married her. Hence, my father became

*mathematical structures and data structures*

*flexibility and power*

*irregularity*



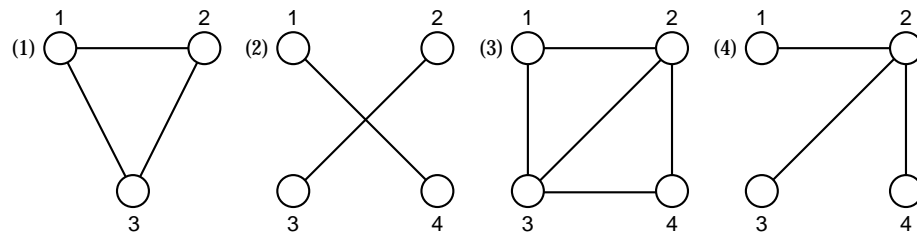
<sup>1</sup> *Algorithms + Data Structures = Programs*, Prentice Hall, Englewood Cliffs, N. J., 1976, page 170.



my son-in-law, and my step-daughter became my mother. Some months later, my wife gave birth to a son, who became the brother-in-law of my father as well as my uncle. The wife of my father, that is my step-daughter, also had a son. Thereby, I got a brother and at the same time a grandson. My wife is my grandmother, since she is my mother's mother. Hence, I am my wife's husband and at the same time her step-grandson; in other words, I am my own grandfather.

## Exercises 12.7

- E1.** (a) Find all the cycles in each of the following graphs. (b) Which of these graphs are connected? (c) Which of these graphs are free trees?



- E2.** For each of the graphs shown in Exercise E1, give the implementation of the graph as (a) an adjacency table, (b) a linked vertex list with linked adjacency lists, (c) a contiguous vertex list of contiguous adjacency lists.
- E3.** A graph is *regular* if every vertex has the same valence (that is, if it is adjacent to the same number of other vertices). For a regular graph, a good implementation is to keep the vertices in a linked list and the adjacency lists contiguous. The length of all the adjacency lists is called the *degree* of the graph. Write a C++ class specification for this implementation of regular graphs.
- E4.** The topological sorting functions as presented in the text are deficient in error checking. Modify the (a) depth-first and (b) breadth-first functions so that they will detect any (directed) cycles in the graph and indicate what vertices cannot be placed in any topological order because they lie on a cycle.
- E5.** How can we determine a maximal spanning tree in a network?
- E6.** Kruskal's algorithm to compute a minimal spanning tree in a network works by considering all edges in increasing order of weight. We select edges for a spanning tree, by adding edges to an initially empty set. An edge is selected if together with the previously selected edges it creates no cycle. Prove that the edges chosen by Kruskal's algorithm do form a minimal spanning tree of a connected network.
- E7.** Dijkstra's algorithm to compute a minimal spanning tree in a network works by considering all edges in any convenient order. As in Kruskal's algorithm, we select edges for a spanning tree, by adding edges to an initially empty set. However, each edge is now selected as it is considered, but if it creates a cycle together with the previously selected edges, the most expensive edge in this cycle is deselected. Prove that the edges chosen by Dijkstra's algorithm also form a minimal spanning tree of a connected network.

## Programming Projects 12.7

- P1.** Write Digraph methods called `read` that will read from the terminal the number of vertices in an undirected graph and lists of adjacent vertices. Be sure to include error checking. The graph is to be implemented with
- (a) an adjacency table;
  - (b) a linked vertex list with linked adjacency lists;
  - (c) a contiguous vertex list of linked adjacency lists.
- P2.** Write Digraph methods called `write` that will write pertinent information specifying a graph to the terminal. The graph is to be implemented with
- (a) an adjacency table;
  - (b) a linked vertex list with linked adjacency lists;
  - (c) a contiguous vertex list of linked adjacency lists.
- P3.** Use the methods `read` and `write` to implement and test the topological sorting functions developed in this section for
- (a) depth-first order and
  - (b) breadth-first order.
- P4.** Write Digraph methods called `read` and `write` that will perform input and output for the implementation of [Section 12.5](#). Make sure that the method `write()` also applies to the derived class `Network` of [Section 12.6](#).
- P5.** Implement and test the method for determining shortest distances in directed graphs with weights.
- P6.** Implement and test the methods of Prim, Kruskal, and Dijkstra for determining minimal spanning trees of a connected network.



## POINTERS AND PITFALLS



1. Graphs provide an excellent way to describe the essential features of many applications, thereby facilitating specification of the underlying problems and formulation of algorithms for their solution. Graphs sometimes appear as data structures but more often as mathematical abstractions useful for problem solving.
2. Graphs may be implemented in many ways by the use of different kinds of data structures. Postpone implementation decisions until the applications of graphs in the problem-solving and algorithm-development phases are well understood.
3. Many applications require graph traversal. Let the application determine the traversal method: depth first, breadth first, or some other order. Depth-first traversal is naturally recursive (or can use a stack). Breadth-first traversal normally uses a queue.
4. Greedy algorithms represent only a sample of the many paradigms useful in developing graph algorithms. For further methods and examples, consult the references.

## REVIEW QUESTIONS

- 12.1 1. In the sense of this chapter, what is a *graph*? What are *edges* and *vertices*?  
 2. What is the difference between an *undirected* and a *directed* graph?  
 3. Define the terms *adjacent*, *path*, *cycle*, and *connected*.  
 4. What does it mean for a directed graph to be strongly connected? Weakly connected?
- 12.2 5. Describe three ways to implement graphs in computer memory.
- 12.3 6. Explain the difference between *depth-first* and *breadth-first* traversal of a graph.  
 7. What data structures are needed to keep track of the waiting vertices during (a) depth-first and (b) breadth-first traversal?
- 12.4 8. For what kind of graphs is *topological sorting* defined?  
 9. What is a *topological order* for such a graph?
- 12.5 10. Why is the algorithm for finding shortest distances called *greedy*?
- 12.6 11. Explain how Prim's algorithm for minimal spanning trees differs from Kruskal's algorithm.

## REFERENCES FOR FURTHER STUDY

The study of graphs and algorithms for their processing is a large subject and one that involves both mathematics and computing science. Three books, each of which contains many interesting algorithms, are

R. E. TARJAN, *Data Structures and Network Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, 1983, 131 pages.

SHIMON EVEN, *Graph Algorithms*, Computer Science Press, Rockville, Md., 1979, 249 pages.

E. M. REINGOLD, J. NIEVERGELT, N. DEO, *Combinatorial Algorithms: Theory and Practice*, Prentice-Hall, Englewood Cliffs, N. J., 1977, 433 pages.

The original reference for the greedy algorithm determining the shortest paths in a graph is

E. W. DIJKSTRA, "A note on two problems in connexion with graphs," *Numerische Mathematik* 1 (1959), 269–271.

Prim's algorithm for minimal spanning trees is reported in

R. C. PRIM, "Shortest connection networks and some generalizations," *Bell System Technical Journal* 36 (1957), 1389–1401.

Kruskal's algorithm is described in

J. B. KRUSKAL, "On the shortest spanning tree of a graph and the traveling salesman problem," *Proceedings of the American Mathematical Society* 7 (1956), 48–50.

The original reference for Dijkstra's algorithm for minimal spanning trees is

E. W. DIJKSTRA, "Some theorems on spanning subtrees of a graph," *Indagationes Mathematicæ* 28 (1960), 196–199.