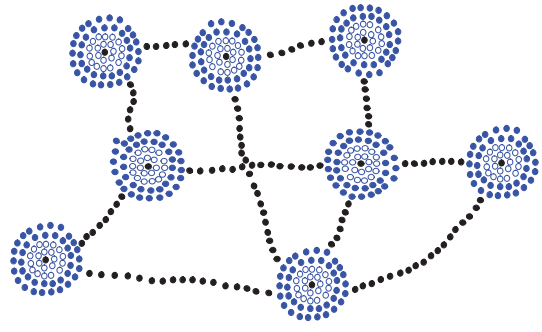
Chapter

13 Graph Algorithms



Contents

13.1 Graphs

A ***graph*** is a way of representing relationships that exist between pairs of objects. That is, a graph is a set of objects, called vertices, together with a collection of pairwise connections between them. This notion of a "graph" should not be confused with bar charts and function plots, as these kinds of "graphs" are unrelated to the topic of this chapter. Graphs have applications in a host of different domains, including mapping, transportation, electrical engineering, and computer networks.

Viewed abstractly, a ***graph*** G is simply a set V of ***vertices*** and a collection E of pairs of vertices from V, called ***edges***. Thus, a graph is a way of representing connections or relationships between pairs of objects from some set V. Some books use different terminology for graphs and refer to what we call vertices as ***nodes*** and what we call edges as ***arcs***. We use the terms "vertices" and "edges."

Edges in a graph are either ***directed*** or ***undirected***. An edge (u,v) is said to be ***directed*** from u to v if the pair (u,v) is ordered, with u preceding v. An edge (u,v) is said to be ***undirected*** if the pair (u,v) is not ordered. Undirected edges are sometimes denoted with set notation, as $\{u,v\}$, but for simplicity we use the pair notation (u,v), noting that in the undirected case (u,v) is the same as (v,u). Graphs are typically visualized by drawing the vertices as ovals or rectangles and the edges as segments or curves connecting pairs of ovals and rectangles. The following are some examples of directed and undirected graphs.

Example 13.1: *We can visualize collaborations among the researchers of a certain discipline by constructing a graph whose vertices are associated with the researchers themselves, and whose edges connect pairs of vertices associated with researchers who have coauthored a paper or book. (See Figure 13.1.) Such edges are undirected because coauthorship is a* **symmetric** *relation; that is, if A has coauthored something with B, then B necessarily has coauthored something with A.*

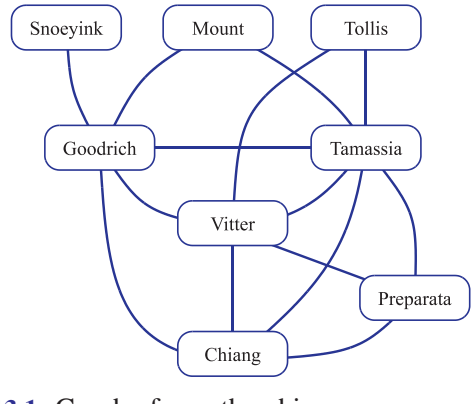


Figure 13.1: Graph of coauthorship among some authors.

Example 13.2: *An object-oriented program can be associated with a graph whose vertices represent the classes defined in the program and whose edges indicate inheritance between classes. There is an edge from a vertex v to a vertex u if the class for v extends the class for u. Such edges are directed because the inheritance relation only goes in one direction (that is, it is* **asymmetric***).*

If all the edges in a graph are undirected, then we say the graph is an ***undirected graph***. Likewise, a ***directed graph***, also called a ***digraph***, is a graph whose edges are all directed. A graph that has both directed and undirected edges is often called a ***mixed graph***. Note that an undirected or mixed graph can be converted into a directed graph by replacing every undirected edge (u,v) by the pair of directed edges (u,v) and (v,u). It is often useful, however, to keep undirected and mixed graphs represented as they are, for such graphs have several applications, such as that of the following example.

Example 13.3: *A city map can be modeled by a graph whose vertices are intersections or dead ends, and whose edges are stretches of streets without intersections. This graph has both undirected edges, which correspond to stretches of two-way streets, and directed edges, which correspond to stretches of one-way streets. Thus, in this way, a graph modeling a city map is a mixed graph.*

Example 13.4: *Physical examples of graphs are present in the electrical wiring and plumbing networks of a building. Such networks can be modeled as graphs, where each connector, fixture, or outlet is viewed as a vertex, and each uninterrupted stretch of wire or pipe is viewed as an edge. Such graphs are actually components of much larger graphs, namely the local power and water distribution networks. Depending on the specific aspects of these graphs that we are interested in, we may consider their edges as undirected or directed, because, in principle, water can flow in a pipe and current can flow in a wire in either direction.*

The two vertices joined by an edge are called the ***end vertices*** (or ***endpoints***) of the edge. If an edge is directed, its first endpoint is its ***origin*** and the other is the ***destination*** of the edge. Two vertices u and v are said to be ***adjacent*** if there is an edge whose end vertices are u and v. An edge is said to be ***incident*** on a vertex if the vertex is one of the edge's endpoints. The ***outgoing edges*** of a vertex are the directed edges whose origin is that vertex. The ***incoming edges*** of a vertex are the directed edges whose destination is that vertex. The ***degree*** of a vertex v, denoted $\deg(v)$, is the number of incident edges of v. The ***in-degree*** and ***out-degree*** of a vertex v are the number of the incoming and outgoing edges of v, and are denoted $\text{indeg}(v)$ and $\text{outdeg}(v)$, respectively.

Example 13.5: *We can study air transportation by constructing a graph G, called* **a flight network** *, whose vertices are associated with airports, and whose edges are associated with flights. (See Figure 13.2.) In graph G, the edges are directed because a given flight has a specific travel direction (from the origin airport to the destination airport). The endpoints of an edge e in G correspond respectively to the origin and destination for the flight corresponding to e. Two airports are adjacent in G if there is a flight that flies between them, and an edge e is incident upon a vertex v in G if the flight for e flies to or from the airport for v. The outgoing edges of a vertex v correspond to the outbound flights from v's airport, and the incoming edges correspond to the inbound flights to v's airport. Finally, the in-degree of a vertex v of G corresponds to the number of inbound flights to v's airport, and the out-degree of a vertex v in G corresponds to the number of outbound flights.*

The definition of a graph refers to the group of edges as a ***collection***, not a ***set***, thus allowing for two undirected edges to have the same end vertices, and for two directed edges to have the same origin and the same destination. Such edges are called ***parallel edges*** or ***multiple edges***. Parallel edges can be in a flight network (Example 13.5), in which case multiple edges between the same pair of vertices could indicate different flights operating on the same route at different times of the day. Another special type of edge is one that connects a vertex to itself. Namely, we say that an edge (undirected or directed) is a ***self-loop*** if its two endpoints coincide. A self-loop may occur in a graph associated with a city map (Example 13.3), where it would correspond to a "circle" (a curving street that returns to its starting point).

With few exceptions, graphs do not have parallel edges or self-loops. Such graphs are said to be ***simple***. Thus, we can usually say that the edges of a simple graph are a ***set*** of vertex pairs (and not just a collection). Throughout this chapter, we assume that a graph is simple unless otherwise specified.

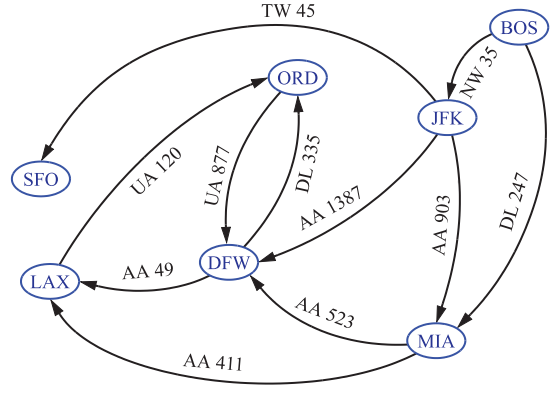


Figure 13.2: Example of a directed graph representing a flight network. The endpoints of edge UA 120 are LAX and ORD; hence, LAX and ORD are adjacent. The in-degree of DFW is 3, and the out-degree of DFW is 2.

In the propositions that follow, we explore a few important properties of graphs.

Proposition 13.6: *If G is a graph with m edges, then*

$$\sum_{v \text{ in } G} deg(v) = 2m.$$

Justification: An edge (u,v) is counted twice in the summation above; once by its endpoint u and once by its endpoint v. Thus, the total contribution of the edges to the degrees of the vertices is twice the number of edges. ■

Proposition 13.7: *If G is a directed graph with m edges, then*

$$\sum_{v \text{ in } G} indeg(v) = \sum_{v \text{ in } G} outdeg(v) = m.$$

Justification: In a directed graph, an edge (u,v) contributes one unit to the out-degree of its origin u and one unit to the in-degree of its destination v. Thus, the total contribution of the edges to the out-degrees of the vertices is equal to the number of edges, and similarly for the out-degrees. ■

We next show that a simple graph with n vertices has $O(n^2)$ edges.

Proposition 13.8: *Let G be a simple graph with n vertices and m edges. If G is undirected, then $m \leq n(n-1)/2$, and if G is directed, then $m \leq n(n-1)$.*

Justification: Suppose that G is undirected. Since no two edges can have the same endpoints and there are no self-loops, the maximum degree of a vertex in G is $n-1$ in this case. Thus, by Proposition 13.6, $2m \leq n(n-1)$. Now suppose that G is directed. Since no two edges can have the same origin and destination, and there are no self-loops, the maximum in-degree of a vertex in G is $n-1$ in this case. Thus, by Proposition 13.7, $m \leq n(n-1)$. ■

A ***path*** is a sequence of alternating vertices and edges that starts at a vertex and ends at a vertex such that each edge is incident to its predecessor and successor vertex. A ***cycle*** is a path with at least one edge that has the same start and end vertices. We say that a path is ***simple*** if each vertex in the path is distinct, and we say that a cycle is ***simple*** if each vertex in the cycle is distinct, except for the first and last one. A ***directed path*** is a path such that all edges are directed and are traversed along their direction. A ***directed cycle*** is similarly defined. For example, in Figure 13.2, (BOS, NW 35, JFK, AA 1387, DFW) is in a directed simple path, and (LAX, UA 120, ORD, UA 877, DFW, AA 49, LAX) is a directed simple cycle. If a path P or cycle C is a simple graph, we may omit the edges in P or C, as these are well defined, in which case P is a list of adjacent vertices and C is a cycle of adjacent vertices.

Example 13.9: *Given a graph G representing a city map (see Example 13.3), we can model a couple driving to dinner at a recommended restaurant as traversing a path though G. If they know the way, and don't accidentally go through the same intersection twice, then they traverse a simple path in G. Likewise, we can model the entire trip the couple takes, from their home to the restaurant and back, as a cycle. If they go home from the restaurant in a completely different way than how they went, not even going through the same intersection twice, then their entire round trip is a simple cycle. Finally, if they travel along one-way streets for their entire trip, we can model their night out as a directed cycle.*

A ***subgraph*** of a graph G is a graph H whose vertices and edges are subsets of the vertices and edges of G, respectively. For example, in the flight network of Figure 13.2, vertices BOS, JFK, and MIA, and edges AA 903 and DL 247 form a subgraph. A ***spanning subgraph*** of G is a subgraph of G that contains all the vertices of the graph G. A graph is ***connected*** if, for any two vertices, there is a path between them. If a graph G is not connected, its maximal connected subgraphs are called the ***connected components*** of G. A ***forest*** is a graph without cycles. A ***tree*** is a connected forest, that is, a connected graph without cycles. Note that this definition of a tree is somewhat different from the one given in Chapter 7. Namely, in the context of graphs, a tree has no root. Whenever there is ambiguity, the trees of Chapter 7 should be referred to as ***rooted trees***, while the trees of this chapter should be referred to as ***free trees***. The connected components of a forest are (free) trees. A ***spanning tree*** of a graph is a spanning subgraph that is a (free) tree.

Example 13.10: *Perhaps the most talked about graph today is the Internet, which can be viewed as a graph whose vertices are computers and whose (undirected) edges are communication connections between pairs of computers on the Internet. The computers and the connections between them in a single domain, like* wiley.com*, form a subgraph of the Internet. If this subgraph is connected, then two users on computers in this domain can send e-mail to one another without having their information packets ever leave their domain. Suppose the edges of this subgraph form a spanning tree. This implies that, if even a single connection goes down (for example, because someone pulls a communication cable out of the back of a computer in this domain), then this subgraph will no longer be connected.*

There are a number of simple properties of trees, forests, and connected graphs.

Proposition 13.11: *Let G be an undirected graph with n vertices and m edges.*

- *If G is connected, then $m \geq n - 1$*
- *If G is a tree, then $m = n - 1$*
- *If G is a forest, then $m \leq n - 1$*

13.1.1 The Graph ADT

In this section, we introduce a simplified graph abstract data type (ADT), which is suitable for undirected graphs, that is, graphs whose edges are all undirected. Additional functions for dealing with directed edges are discussed in Section 13.4.

As an abstract data type, a graph is a collection of elements that are stored at the graph's ***positions***—its vertices and edges. Hence, we can store elements in a graph at either its edges or its vertices (or both). The graph ADT defines two types, Vertex and Edge. It also provides two list types for storing lists of vertices and edges, called VertexList and EdgeList, respectively.

Each Vertex object u supports the following operations, which provide access to the vertex's element and information regarding incident edges and adjacent vertices.

operator*(): Return the element associated with u.

incidentEdges(): Return an edge list of the edges incident on u.

isAdjacentTo(v): Test whether vertices u and v are adjacent.

Each Edge object e supports the following operations, which provide access to the edge's end vertices and information regarding the edge's incidence relationships.

operator*(): Return the element associated with e.

endVertices(): Return a vertex list containing e's end vertices.

opposite(v): Return the end vertex of edge e distinct from vertex v; an error occurs if e is not incident on v.

isAdjacentTo(f): Test whether edges e and f are adjacent.

isIncidentOn(v): Test whether e is incident on v.

Finally, the full graph ADT consists of the following operations, which provide access to the lists of vertices and edges, and provide functions for modifying the graph.

vertices(): Return a vertex list of all the vertices of the graph.

edges(): Return an edge list of all the edges of the graph.

insertVertex(x): Insert and return a new vertex storing element x.

insertEdge(v, w, x): Insert and return a new undirected edge with end vertices v and w and storing element x.

eraseVertex(v): Remove vertex v and all its incident edges.

eraseEdge(e): Remove edge e.

The VertexList and EdgeList classes support the standard list operations, as described in Chapter 6. In particular, we assume that each provides an iterator (Section 6.2.1), which we call VertexItor and EdgeItor, respectively. They also provide functions begin and end, which return iterators to the beginning and end of their respective lists.

13.2 Data Structures for Graphs

In this section, we discuss three popular ways of representing graphs, which are usually referred to as the ***edge list*** structure, the ***adjacency list*** structure, and the ***adjacency matrix***. In all three representations, we use a collection to store the vertices of the graph. Regarding the edges, there is a fundamental difference between the first two structures and the latter. The edge list structure and the adjacency list structure only store the edges actually present in the graph, while the adjacency matrix stores a placeholder for every pair of vertices (whether there is an edge between them or not). As we will explain in this section, this difference implies that, for a graph G with n vertices and m edges, an edge list or adjacency list representation uses $O(n+m)$ space, whereas an adjacency matrix representation uses $O(n^2)$ space.

13.2.1 The Edge List Structure

The ***edge list*** structure is possibly the simplest, though not the most efficient, representation of a graph G. In this representation, a vertex v of G storing an element x is explicitly represented by a vertex object. All such vertex objects are stored in a collection V, such as a vector or node list. If V is a vector, for example, then we naturally think of the vertices as being numbered.

Vertex Objects

The vertex object for a vertex v storing element x has member variables for:

- A copy of x
- The position (or entry) of the vertex-object in collection V

The distinguishing feature of the edge list structure is not how it represents vertices, but the way in which it represents edges. In this structure, an edge e of G storing an element x is explicitly represented by an edge object. The edge objects are stored in a collection E, which would typically be a vector or node list.

Edge Objects

The edge object for an edge e storing element x has member variables for:

- A copy of x
- The vertex positions associated with the endpoint vertices of e
- The position (or entry) of the edge-object in collection E

Visualizing the Edge List Structure

We illustrate an example of the edge list structure for a graph G in Figure 13.3.

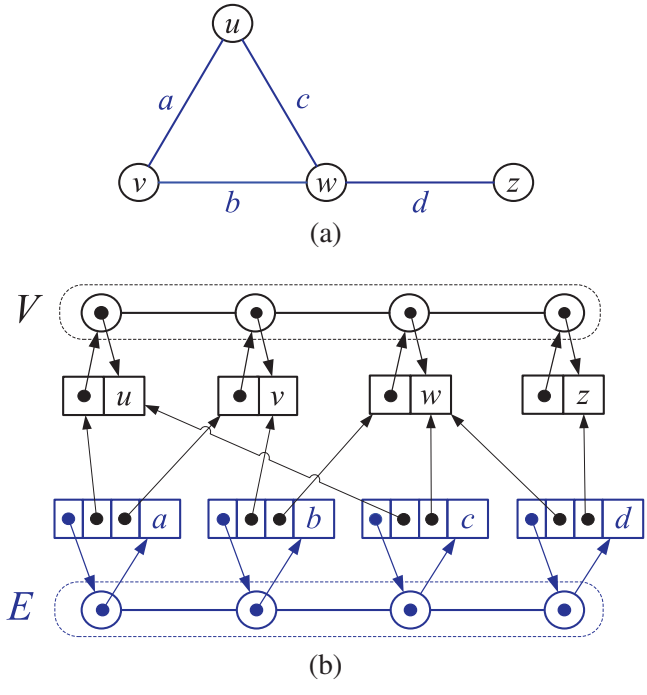


Figure 13.3: (a) A graph G. (b) Schematic representation of the edge list structure for G. We visualize the elements stored in the vertex and edge objects with the element names, instead of with actual references to the element objects.

The reason this structure is called the ***edge list structure*** is that the simplest and most common implementation of the edge collection E is by using a list. Even so, in order to be able to conveniently search for specific objects associated with edges, we may wish to implement E with a dictionary (whose entries store the element as the key and the edge as the value) in spite of our calling this the "edge list." We may also want to implement the collection V by using a dictionary for the same reason. Still, in keeping with tradition, we call this structure the edge list structure.

The main feature of the edge list structure is that it provides direct access from edges to the vertices they are incident upon. This allows us to define simple algorithms for functions e.endVertices() and e.opposite(v).

Performance of the Edge List Structure

One method that is inefficient for the edge list structure is that of accessing the edges that are incident upon a vertex. Determining this set of vertices requires an exhaustive inspection of all the edge objects in the collection E. That is, in order to determine which edges are incident to a vertex v, we must examine all the edges in the edge list and check, for each one, if it happens to be incident to v. Thus, function v.incidentEdges() runs in time proportional to the number of edges in the graph, not in time proportional to the degree of vertex v. In fact, even to check if two vertices v and w are adjacent by the v.isAdjacentTo(w) function, requires that we search the entire edge collection looking for an edge with end vertices v and w. Moreover, since removing a vertex involves removing all of its incident edges, function eraseVertex also requires a complete search of the edge collection E.

Table 13.1 summarizes the performance of the edge list structure implementation of a graph under the assumption that collections V and E are realized with doubly linked lists (Section 3.3).

Operation	*Time*
vertices	$O(n)$
edges	$O(m)$
endVertices, opposite	$O(1)$
incidentEdges, isAdjacentTo	$O(m)$
isIncidentOn	$O(1)$
insertVertex, insertEdge, eraseEdge,	$O(1)$
eraseVertex	$O(m)$

Table 13.1: Running times of the functions of a graph implemented with the edge list structure. The space used is $O(n+m)$, where n is the number of vertices and m is the number of edges.

Details for selected functions of the graph ADT are as follows:

- Methods vertices() and edges() are implemented by using the iterators for V and E, respectively, to enumerate the elements of the lists.
- Methods incidentEdges and isAdjacentTo all take $O(m)$ time, since to determine which edges are incident upon a vertex v we must inspect all edges.
- Since the collections V and E are lists implemented with a doubly linked list, we can insert vertices, and insert and remove edges, in $O(1)$ time.
- The update function eraseVertex(v) takes $O(m)$ time, since it requires that we inspect all the edges to find and remove those incident upon v.

Thus, the edge list representation is simple but has significant limitations.

13.2.2 The Adjacency List Structure

The ***adjacency list*** structure for a graph G adds extra information to the edge list structure that supports direct access to the incident edges (and thus to the adjacent vertices) of each vertex. This approach allows us to use the adjacency list structure to implement several functions of the graph ADT much faster than what is possible with the edge list structure, even though both of these two representations use an amount of space proportional to the number of vertices and edges in the graph. The adjacency list structure includes all the structural components of the edge list structure plus the following:

- A vertex object v holds a reference to a collection $I(v)$, called the ***incidence collection*** of v, whose elements store references to the edges incident on v.
- The edge object for an edge e with end vertices v and w holds references to the positions (or entries) associated with edge e in the incidence collections $I(v)$ and $I(w)$.

Traditionally, the incidence collection $I(v)$ for a vertex v is a list, which is why we call this way of representing a graph the ***adjacency list*** structure. The adjacency list structure provides direct access both from the edges to the vertices and from the vertices to their incident edges. We illustrate the adjacency list structure of a graph in Figure 13.4.

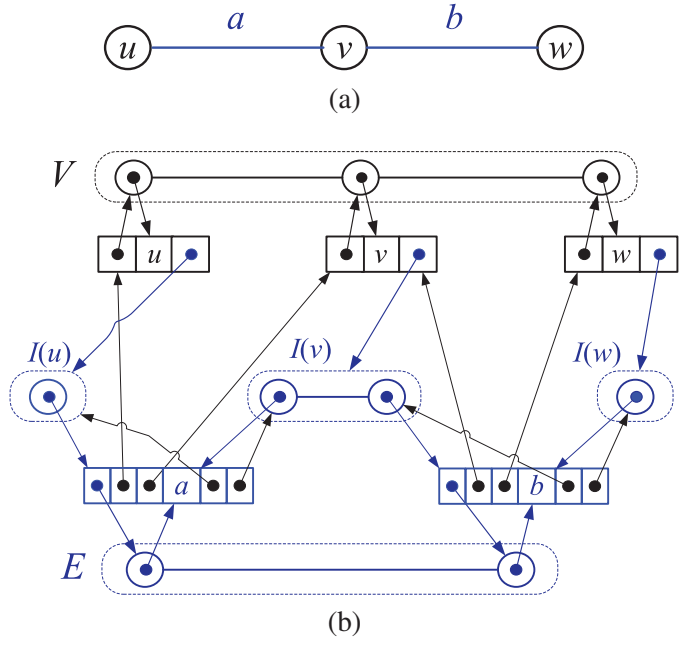


Figure 13.4: (a) A graph G. (b) Schematic representation of the adjacency list structure of G. As in Figure 13.3, we visualize the elements of collections with names.

Performance of the Adjacency List Structure

All of the functions of the graph ADT that can be implemented with the edge list structure in $O(1)$ time can also be implemented in $O(1)$ time with the adjacency list structure, using essentially the same algorithms. In addition, being able to provide access between vertices and edges in both directions allows us to speed up the performance of a number of graph functions by using an adjacency list structure instead of an edge list structure. Table 13.2 summarizes the performance of the adjacency list structure implementation of a graph, assuming that collections V and E and the incidence collections of the vertices are all implemented with doubly linked lists. For a vertex v, the space used by the incidence collection of v is proportional to the degree of v, that is, it is $O(\deg(v))$. Thus, by Proposition 13.6, the space used by the adjacency list structure is $O(n+m)$.

Operation	*Time*
vertices	$O(n)$
edges	$O(m)$
endVertices, opposite	$O(1)$
v.incidentEdges()	$O(\deg(v))$
v.isAdjacentTo(w)	$O(\min(\deg(v),\deg(w))$
isIncidentOn	$O(1)$
insertVertex, insertEdge, eraseEdge,	$O(1)$
eraseVertex(v)	$O(\deg(v))$

Table 13.2: Running times of the functions of a graph implemented with the adjacency list structure. The space used is $O(n+m)$, where n is the number of vertices and m is the number of edges.

In contrast to the edge-list way of doing things, the adjacency list structure provides improved running times for the following functions:

- Methods vertices() and edges() are implemented by using the iterators for V and E, respectively, to enumerate the elements of the lists.
- Method v.incidentEdges() takes time proportional to the number of incident vertices of v, that is, $O(\deg(v))$ time.
- Method v.isAdjacentTo(w) can be performed by inspecting either the incidence collection of v or that of w. By choosing the smaller of the two, we get $O(\min(\deg(v),\deg(w)))$ running time.
- Method eraseVertex(v) takes $O(\deg(v))$ time.

13.2.3 The Adjacency Matrix Structure

Like the adjacency list structure, the ***adjacency matrix*** structure of a graph also extends the edge list structure with an additional component. In this case, we augment the edge list with a matrix (a two-dimensional array) A that allows us to determine adjacencies between pairs of vertices in constant time. In the adjacency matrix representation, we think of the vertices as being the integers in the set $\{0,1,\ldots,n-1\}$ and the edges as being pairs of such integers. This allows us to store references to edges in the cells of a two-dimensional $n \times n$ array A. Specifically, the adjacency matrix representation extends the edge list structure as follows (see Figure 13.5):

- A vertex object v stores a distinct integer i in the range $0,1,\ldots,n-1$, called the ***index*** of v.
- We keep a two-dimensional $n \times n$ array A such that the cell $A[i,j]$ holds a reference to the edge (v,w), if it exists, where v is the vertex with index i and w is the vertex with index j. If there is no such edge, then $A[i,j] = null$.

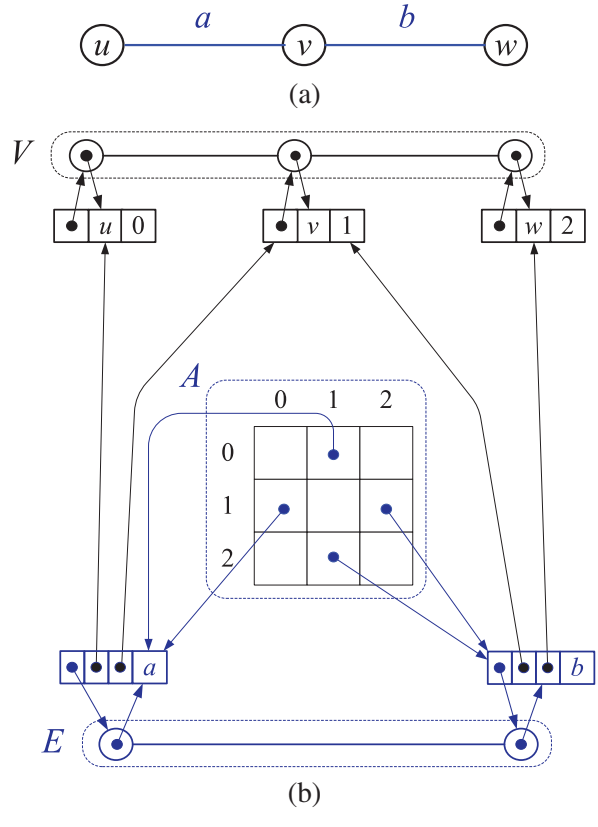


Figure 13.5: (a) A graph G without parallel edges. (b) Schematic representation of the simplified adjacency matrix structure for G.

Performance of the Adjacency Matrix Structure

For graphs with parallel edges, the adjacency matrix representation must be extended so that, instead of having $A[i,j]$ storing a pointer to an associated edge (v,w), it must store a pointer to an incidence collection $I(v,w)$, which stores all the edges from v to w. Since most of the graphs we consider are simple, we do not consider this complication here.

The adjacency matrix A allows us to perform v.isAdjacentTo(w) in $O(1)$ time. This is done by accessing vertices v and w to determine their respective indices i and j, and then testing whether $A[i,j]$ is null. The efficiency of isAdjacentTo is counteracted by an increase in space usage, however, which is now $O(n^2)$, and in the running time of other functions. For example, function v.incidentEdges() now requires that we examine an entire row or column of array A and thus runs in $O(n)$ time. Moreover, any vertex insertions or deletions now require creating a whole new array A, of larger or smaller size, respectively, which takes $O(n^2)$ time.

Table 13.3 summarizes the performance of the adjacency matrix structure implementation of a graph. From this table, we observe that the adjacency list structure is superior to the adjacency matrix in space, and is superior in time for all functions except for the isAdjacentTo function.

Operation	*Time*
vertices	$O(n)$
edges	$O(n^2)$
endVertices, opposite	$O(1)$
isAdjacentTo, isIncidentOn	$O(1)$
incidentEdges	$O(n)$
insertEdge, eraseEdge,	$O(1)$
insertVertex, eraseVertex	$O(n^2)$

Table 13.3: Running times for a graph implemented with an adjacency matrix.

Historically, Boolean adjacency matrices were the first representations used for graphs (so that $A[i,j] =$ **true** if and only if (i,j) is an edge). We should not find this fact surprising, however, for the adjacency matrix has a natural appeal as a mathematical structure (for example, an undirected graph has a symmetric adjacency matrix). The adjacency list structure came later, with its natural appeal in computing due to its faster methods for most algorithms (many algorithms do not use function isAdjacentTo) and its space efficiency.

Most of the graph algorithms we examine run efficiently when acting upon a graph stored using the adjacency list representation. In some cases, however, a trade-off occurs, where graphs with few edges are most efficiently processed with an adjacency list structure, and graphs with many edges are most efficiently processed with an adjacency matrix structure.

13.3 Graph Traversals

Greek mythology tells of an elaborate labyrinth that was built to house the monstrous Minotaur, which was part bull and part man. This labyrinth was so complex that neither beast nor human could escape it. No human, that is, until the Greek hero, Theseus, with the help of the king's daughter, Ariadne, decided to implement a ***graph-traversal*** algorithm. Theseus fastened a ball of thread to the door of the labyrinth and unwound it as he traversed the twisting passages in search of the monster. Theseus obviously knew about good algorithm design, because, after finding and defeating the beast, Theseus easily followed the string back out of the labyrinth to the loving arms of Ariadne. Formally, a ***traversal*** is a systematic procedure for exploring a graph by examining all of its vertices and edges.

13.3.1 Depth-First Search

The first traversal algorithm we consider in this section is ***depth-first search*** (DFS) in an undirected graph. Depth-first search is useful for testing a number of properties of graphs, including whether there is a path from one vertex to another and whether or not a graph is connected.

Depth-first search in an undirected graph G is analogous to wandering in a labyrinth with a string and a can of paint without getting lost. We begin at a specific starting vertex s in G, which we initialize by fixing one end of our string to s and painting s as "visited." The vertex s is now our "current" vertex—call our current vertex u. We then traverse G by considering an (arbitrary) edge (u,v) incident to the current vertex u. If the edge (u,v) leads us to an already visited (that is, painted) vertex v, we immediately return to vertex u. If, on the other hand, (u,v) leads to an unvisited vertex v, then we unroll our string, and go to v. We then paint v as "visited," and make it the current vertex, repeating the computation above. Eventually, we get to a "dead end," that is, a current vertex u such that all the edges incident on u lead to vertices already visited. Thus, taking any edge incident on u causes us to return to u. To get out of this impasse, we roll our string back up, backtracking along the edge that brought us to u, going back to a previously visited vertex v. We then make v our current vertex and repeat the computation above for any edges incident upon v that we have not looked at before. If all of v's incident edges lead to visited vertices, then we again roll up our string and backtrack to the vertex we came from to get to v, and repeat the procedure at that vertex. Thus, we continue to backtrack along the path that we have traced so far until we find a vertex that has yet unexplored edges, take one such edge, and continue the traversal. The process terminates when our backtracking leads us back to the start vertex s, and there are no more unexplored edges incident on s.

This simple process traverses all the edges of G. (See Figure 13.6.)

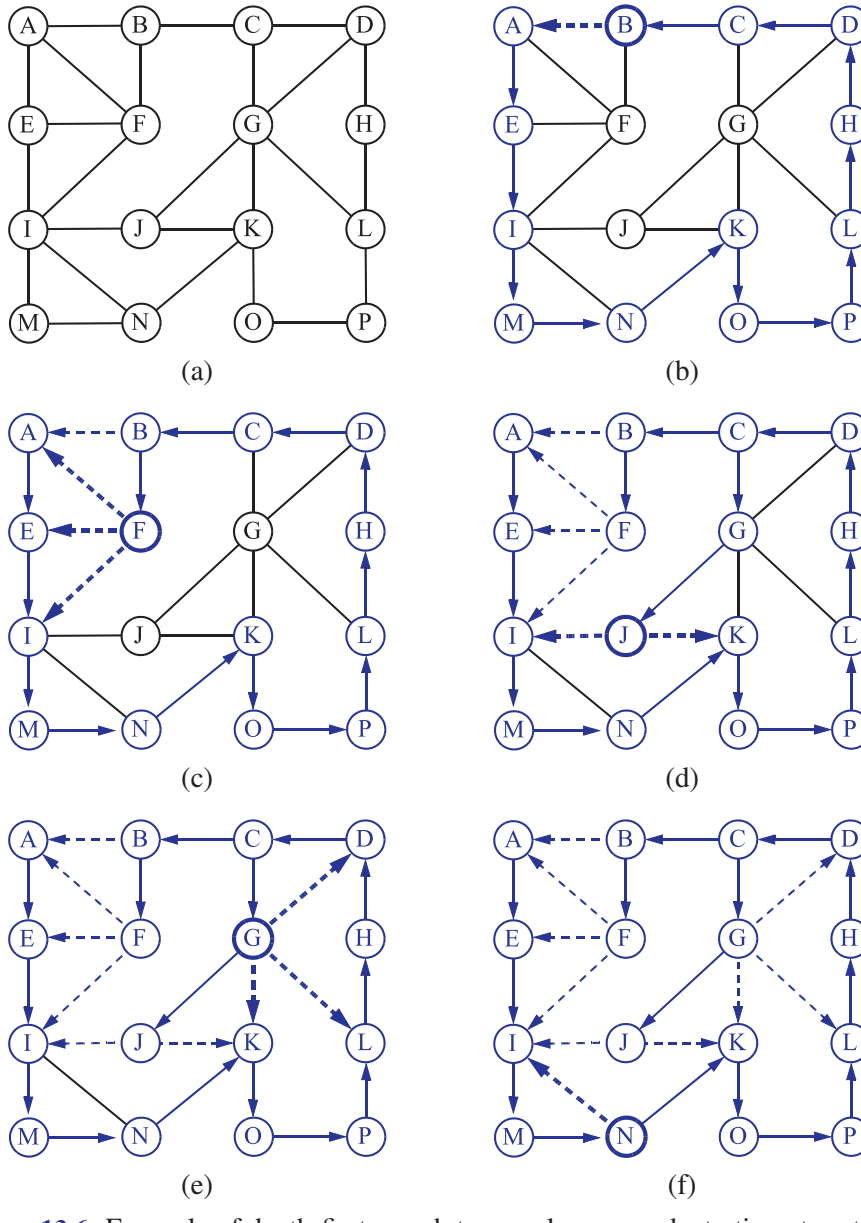


Figure 13.6: Example of depth-first search traversal on a graph starting at vertex A. Discovery edges are shown with solid lines and back edges are shown with dashed lines: (a) input graph; (b) path of discovery edges traced from A until back edge (B,A) is hit; (c) reaching F, which is a dead end; (d) after backtracking to C, resuming with edge (C,G), and hitting another dead end, J; (e) after backtracking to G; (f) after backtracking to N.

Discovery Edges and Back Edges

We can visualize a DFS traversal by orienting the edges along the direction in which they are explored during the traversal, distinguishing the edges used to discover new vertices, called ***discovery edges***, or ***tree edges***, from those that lead to already visited vertices, called ***back edges***. (See Figure 13.6(f).) In the analogy above, discovery edges are the edges where we unroll our string when we traverse them, and back edges are the edges where we immediately return without unrolling any string. As we will see, the discovery edges form a spanning tree of the connected component of the starting vertex s. We call the edges not in this tree "back edges" because, assuming that the tree is rooted at the start vertex, each such edge leads back from a vertex in this tree to one of its ancestors in the tree.

The pseudo-code for a DFS traversal starting at a vertex v follows our analogy with string and paint. We use recursion to implement the string analogy, and we assume that we have a mechanism (the paint analogy) to determine if a vertex or edge has been explored or not, and to label the edges as discovery edges or back edges. This mechanism will require additional space and may affect the running time of the algorithm. A pseudo-code description of the recursive DFS algorithm is given in Code Fragment 13.1.

```
Algorithm DFS(G, v):
    Input: A graph G and a vertex v of G
    Output: A labeling of the edges in the connected component of v as discovery
      edges and back edges
  label v as visited
  for all edges e in v.incidentEdges() do
    if edge e is unvisited then
      w ← e.opposite(v)
      if vertex w is unexplored then
        label e as a discovery edge
        recursively call DFS(G, w)
      else
        label e as a back edge
```

Code Fragment 13.1: The DFS algorithm.

There are a number of observations that we can make about the depth-first search algorithm, many of which derive from the way the DFS algorithm partitions the edges of the undirected graph G into two groups, the discovery edges and the back edges. For example, since back edges always connect a vertex v to a previously visited vertex u, each back edge implies a cycle in G, consisting of the discovery edges from u to v plus the back edge (u,v).

Proposition 13.12: *Let G be an undirected graph on which a DFS traversal starting at a vertex s has been performed. Then the traversal visits all vertices in the connected component of s, and the discovery edges form a spanning tree of the connected component of s.*

Justification: Suppose there is at least one vertex v in s's connected component not visited, and let w be the first unvisited vertex on some path from s to v (we may have $v = w$). Since w is the first unvisited vertex on this path, it has a neighbor u that was visited. But when we visited u, we must have considered the edge (u,w); hence, it cannot be correct that w is unvisited. Therefore, there are no unvisited vertices in s's connected component.

Since we only mark edges when we go to unvisited vertices, we will never form a cycle with discovery edges, that is, discovery edges form a tree. Moreover, this is a spanning tree because, as we have just seen, the depth-first search visits each vertex in the connected component of s. ■

In terms of its running time, depth-first search is an efficient method for traversing a graph. Note that DFS is called exactly once on each vertex, and that every edge is examined exactly twice, once from each of its end vertices. Thus, if n_s vertices and m_s edges are in the connected component of vertex s, a DFS starting at s runs in $O(n_s + m_s)$ time, provided the following conditions are satisfied:

- The graph is represented by a data structure such that creating and iterating through the list generated by v.incidentEdges() takes $O(\mathsf{degree}(v))$ time, and e.opposite(v) takes $O(1)$ time. The adjacency list structure is one such structure, but the adjacency matrix structure is not.
- We have a way to "mark" a vertex or edge as explored, and to test if a vertex or edge has been explored in $O(1)$ time. We discuss ways of implementing DFS to achieve this goal in the next section.

Given the assumptions above, we can solve a number of interesting problems.

Proposition 13.13: *Let G be a graph with n vertices and m edges represented with an adjacency list. A DFS traversal of G can be performed in $O(n+m)$ time, and can be used to solve the following problems in $O(n+m)$ time:*

- *Testing whether G is connected*
- *Computing a spanning tree of G, if G is connected*
- *Computing the connected components of G*
- *Computing a path between two given vertices of G, if it exists*
- *Computing a cycle in G, or reporting that G has no cycles*

The justification of Proposition 13.13 is based on algorithms that use slightly modified versions of the DFS algorithm as subroutines.

13.3.2 Implementing Depth-First Search

As we have mentioned above, the data structure we use to represent a graph impacts the performance of the DFS algorithm. For example, an adjacency list can be used to yield a running time of $O(n+m)$ for traversing a graph with n vertices and m edges. Using an adjacency matrix, on the other hand, would result in a running time of $O(n^2)$, since each of the n calls to the incidentEdges function would take $O(n)$ time. If the graph is ***dense***, that is, it has close to $O(n^2)$ edges, then the difference between these two choices is minor, as they both would run in $O(n^2)$ time. But if the graph is ***sparse***, that is, it has close to $O(n)$ edges, then the adjacency matrix approach would be much slower than the adjacency list approach.

Another important implementation detail deals with the way vertices and edges are represented. In particular, we need to have a way of marking vertices and edges as visited or not. There are two simple solutions, but each has drawbacks.

- We can build our vertex and edge objects to contain a *visited* field, which can be used by the DFS algorithm for marking. This approach is quite simple, and supports constant-time marking and unmarking, but it assumes that we are designing our graph with DFS in mind, which will not always be valid. Furthermore, this approach needlessly restricts DFS to graphs with vertices having a *visited* field. Thus, if we want a generic DFS algorithm that can take any graph as input, this approach has limitations.
- We can use an auxiliary hash table to store all the explored vertices and edges during the DFS algorithm. This scheme is general, in that it does not require any special fields in the positions of the graph. But this approach does not achieve worst-case constant time for marking and unmarking of vertices edges. Instead, such a hash table only supports the mark (insert) and test (find) operations in constant ***expected*** time (see Section 9.2).

Fortunately, there is a middle ground between these two extremes.

The Decorator Pattern

Marking the explored vertices in a DFS traversal is an example of the ***decorator*** software engineering design pattern. This pattern is used to add ***decorations*** (also called ***attributes***) to existing objects. Each decoration is identified by a ***key*** identifying this decoration and by a ***value*** associated with the key. The use of decorations is motivated by the need of some algorithms and data structures to add extra variables, or temporary scratch data, to objects that do not normally have such variables. Hence, a decoration is a key-value pair that can be dynamically attached to an object. In our DFS example, we would like to have "decorable" vertices and edges with a *visited* decoration and a Boolean value.

Making Graph Vertices Decorable

We can realize the decorator pattern for any position by allowing it to be decorated. This allows us to add labels to vertices and edges, without requiring that we know in advance the kinds of labels that we will need. We say that an object is ***decorable*** if it supports the following functions:

set(a,x): Set the value of attribute a to x.

get(a): Return the value of attribute a.

We assume that Vertex and Edge objects of our graph ADT are decorable, where attribute keys are strings and attribute values are pointers to a generic object class, called Object.

As an example of how this works, suppose that we want to mark vertices as being either visited or not visited by a search procedure. To implement this, we could create two new instances of the Object class, and store pointers to these objects in two variables, say *yes* and *no*. The values of these objects are unimportant to us—all we require is the ability to distinguish between them. Let v be an object of type Decorator. To indicate that v is visited we invoke v.set("visited",*yes*) and to indicate that it was not visited we invoke v.set("visited",*no*). In order to test the value of this decorator, we invoke v.get("visited") and test to see whether the result is *yes* or *no*. This is shown in the following code fragment.

```
Object* yes = new Object;              // decorator values
Object* no = new Object;
Decorator v;                           // a decorable object
// ...
v.set("visited", yes);                 // set "visited" attribute
// ...
if (v.get("visited") == yes) cout << "v was visited";
else cout << "v was not visited";
```

In Code Fragment 13.2, we present a C++ implementation of class Decorator. It works by creating an STL map (Section 9.1.3), whose keys are strings and whose values are of type Object*.

```
class Decorator {
private:                                   // member data
  std::map<string,Object*> map;            // the map
public:
  Object* get(const string& a)             // get value of attribute
    { return map[a]; }
  void set(const string& a, Object* d)     // set value
    { map[a] = d; }
};
```

Code Fragment 13.2: A C++ implementation of class Decorator.

DFS Traversal using Decorable Positions

Using decorable positions, the complete DFS traversal algorithm can be described in more detail, as shown in Code Fragment 13.3. We create an attribute named "status" in which to record the status information about vertices and edges. This attribute may take on one of four possible values: *unvisited*, *visited*, *discovery*, and *back*. Initially, all attribute values are assumed to have been set to *unvisited*. On termination, edges will be labeled as *discovery* or *back*, depending on whether they are discovery edges or back edges.

Algorithm DFS(G, v):
  Input: A graph G with decorable vertices and edges, a vertex v of G, such that all vertices and edges have been decorated with the status value of *unvisited*
  Output: A decoration of the vertices of the connected component of v with the value *visited* and of the edges in the connected component of v with values *discovery* and *back*, according to a depth-first search traversal of G

```
v.set("status", visited)
for all edges e in v.incidentEdges() do
  if e.get("status") = unvisited then
    w ← e.opposite(v)
    if w.get("status") = unvisited then
      e.set("status", discovered)
      DFS(G, w)
    else
      e.set("status", back)
```

Code Fragment 13.3: DFS on a graph with decorable edges and vertices.

13.3.3 A Generic DFS Implementation in C++

In this section, we present a C++ implementation of a generic depth-first search traversal by means of a class, called DFS. This class defines a recursive member function, dfsTraversal(v), which performs a DFS traversal of the graph starting at vertex v. The behavior of the traversal function can be specialized for a particular application by redefining a number of functions, which are invoked in response to various events that arise in the traversal.

We assume that the vertices and edges are decorable positions and use decorations to determine whether vertices and edges have been visited. The generic DFS class contains the following virtual functions, which may be overridden by concrete subclasses to affect a desired behavior:

- startVisit(v): called at the start of the visit of v

- traverseDiscovery(e,v): called when a discovery edge e out of v is traversed
- traverseBack(e,v): called when a back edge e out of v is traversed
- isDone(): called to determine whether to end the traversal early
- finishVisit(v): called when we are finished exploring from v

The class DFS is presented in Code Fragment 13.4. The class is templated with the graph type G. It begins with a number of convenience type definitions to allow us to access elements of the underlying graph type more succinctly. We have omitted some of the type definitions, such as VertexList, EdgeList, VertexItor, and EdgeItor. This is followed by the member data of the class, which consists of a reference to the graph, the vertex at which the depth-first traversal begins, and two decorator objects *yes* and *no*, which will be used in decorating vertices and edges. Their actual values are irrelevant, as long as we can distinguish one from the other.

```
template <typename G>
class DFS {                                          // generic DFS
protected:                                           // local types
  typedef typename G::Vertex Vertex;                 // vertex position
  typedef typename G::Edge Edge;                     // edge position
  // ...insert other typename shortcuts here
protected:                                           // member data
  const G& graph;                                    // the graph
  Vertex start;                                      // start vertex
  Object *yes, *no;                                  // decorator values
protected:                                           // member functions
  DFS(const G& g);                                   // constructor
  void initialize();                                 // initialize a new DFS
  void dfsTraversal(const Vertex& v);                // recursive DFS utility
                                                     // overridden functions
  virtual void startVisit(const Vertex& v) {}        // arrived at v
                                                     // discovery edge e
  virtual void traverseDiscovery(const Edge& e, const Vertex& from) {}
                                                     // back edge e
  virtual void traverseBack(const Edge& e, const Vertex& from) {}
  virtual void finishVisit(const Vertex& v) {}       // finished with v
  virtual bool isDone() const { return false; }      // finished?
  // ...insert marking utilities here
};
```

Code Fragment 13.4: A generic implementation of depth-first search.

The class's member functions are all protected. They are invoked only by public members of the derived subclasses. These member functions include a constructor, an initialization function, and the generic DFS traversal function. There are a number of virtual functions corresponding to each of the above operations, which are overridden by subclasses of class DFS.

We specify whether vertices and edges have been visited during the traversal through calls to the marking utility functions visit, unvisit, and isVisited, which are shown in Code Fragment 13.5.

```
protected:                                          // marking utilities
  void visit(const Vertex& v)          { v.set("visited", yes); }
  void visit(const Edge& e)            { e.set("visited", yes); }
  void unvisit(const Vertex& v)        { v.set("visited", no); }
  void unvisit(const Edge& e)          { e.set("visited", no); }
  bool isVisited(const Vertex& v)      { return v.get("visited") == yes; }
  bool isVisited(const Edge& e)        { return e.get("visited") == yes; }
```

Code Fragment 13.5: Vertex and edge marking utilities, which are part of DFS.

In Code Fragment 13.6, we present the class constructor and a function that performs initializations before the DFS traversal is performed. (We present the external member functions using the condensed notation, which we introduced in Section 9.2.7.) The constructor initializes the graph reference and allocates the generic objects *yes* and *no*, which are used by the marking utilities. (Eventually, these are deallocated by the class destructor, which we have omitted.) The initialization function marks all vertices and edges as "unvisited."

```
/* DFS⟨G⟩ :: */                                     // constructor
  DFS(const G& g)
    : graph(g), yes(new Object), no(new Object) {}

/* DFS⟨G⟩ :: */                                     // initialize a new DFS
  void initialize() {
    VertexList verts = graph.vertices();
    for (VertexItor pv = verts.begin(); pv != verts.end(); ++pv)
      unvisit(*pv);                                 // mark vertices unvisited
    EdgeList edges = graph.edges();
    for (EdgeItor pe = edges.begin(); pe != edges.end(); ++pe)
      unvisit(*pe);                                 // mark edges unvisited
  }
```

Code Fragment 13.6: The class constructor for DFS and the initialization function.

The recursive DFS traversal function is presented in Code Fragment 13.7. The function follows the same structure as presented in Code Fragment 13.3. Note, however, the introduction of calls to the virtual functions startVisit, isDone, traverseDiscovery, traverseBack, and finishVisit. These have not yet been specified, but their definitions determine the concrete behavior of the traversal process.

```
/* DFS⟨G⟩ :: */                                    // recursive traversal
void dfsTraversal(const Vertex& v) {
  startVisit(v);    visit(v);                      // visit v and mark visited
  EdgeList incident = v.incidentEdges();
  EdgeItor pe = incident.begin();
  while (!isDone() && pe != incident.end()) {      // visit v's incident edges
    Edge e = *pe++;
    if (!isVisited(e)) {                           // discovery edge?
      visit(e);                                    // mark it visited
      Vertex w = e.opposite(v);                    // get opposing vertex
      if (!isVisited(w)) {                         // unexplored?
        traverseDiscovery(e, v);                   // let's discover it
        if (!isDone()) dfsTraversal(w);            // continue traversal
      }
      else traverseBack(e, v);                     // process back edge
    }
  }
  if (!isDone()) finishVisit(v);                   // finished with v
}
```

Code Fragment 13.7: The function dfsTraversal, which implements a generic DFS traversal.

Using the Template Method Pattern for DFS

In the remainder of this section, we illustrate a number of concrete applications of our generic DFS traversal. In order to do anything interesting, we must extend DFS and redefine some of its auxiliary functions. This is an example of the template method pattern (see Section 7.3.7), in which a generic computation mechanism is specialized by providing concrete implementations of certain generic steps.

Our first example is the class Components, which counts the number of connected components of a graph. The class is presented in Code Fragment 13.8. Observe that this class is derived from DFS, and so inherits its members. The Components class contains a single data member, which is a counter *nComponents* of the number of connected components it has encountered.

```
template <typename G>
class Components : public DFS<G> {                 // count components
private:
  int nComponents;                                 // num of components
public:
  Components(const G& g): DFS<G>(g) {}             // constructor
  int operator()();                                // count components
};
```

Code Fragment 13.8: The class Components, which extends the DFS class in order to count the number of components of a graph by overloading the "()" operator.

The class provides a simple constructor, which simply invokes the constructor for the base class by passing it the graph. In order to define the component counting function, we have overloaded the "()" operator. This overloaded function returns the number of components. This means that, given a graph G, we can declare a Components objects and invoke it as follows:

```
Components components(G);      // declare a Components object
int nc = components();         // compute the number of components
```

The function that computes the number of connected components is shown in Code Fragment 13.9. After initializing (by invoking the DFS member function initialize and setting the component counter to zero), it iterates through the vertices of the graph. Whenever it finds an unvisited vertex, it invokes the DFS traversal on this vertex and increments the component counter. The DFS traversal returns only after every vertex of this connected component has been visited (Proposition 13.12). Therefore, any unvisited vertex must lie in a different component. By repeating this on every unvisited vertex, eventually every connected component will be found and counted.

```
/* Components⟨G⟩ :: */                                    // count components
  int operator()() {
    initialize();                                          // initialize DFS
    nComponents = 0;                                       // init vertex count
    VertexList verts = graph.vertices();
    for (VertexItor pv = verts.begin(); pv != verts.end(); ++pv) {
      if (!isVisited(*pv)) {                               // not yet visited?
        dfsTraversal(*pv);                                 // visit
        nComponents++;                                     // one more component
      }
    }
    return nComponents;
  }
```

Code Fragment 13.9: The overloaded () operator for class Components, which computes the number of connected components of a graph.

Our next example is class FindPath, which finds a path between a given source vertex and target vertex. The class is presented in Code Fragment 13.10. The class's member data consists of the vertex list forming the path (*path*), the target vertex (*target*), and a boolean variable indicating whether the search is complete (*done*). Like before, the principal path finding function has been defined by overloading the "()" operator. This function is given the source vertex s and target vertex t, and returns the path as a list of vertices from s to t. If there is no such path, an empty list is returned. Also like before, to use this class, we first create a new FindPath object, say *findPath*, and then we invoke *findPath*(s,t), for the desired source vertex s and target vertex t.

```
template <typename G>
class FindPath : public DFS<G> {                    // find a path by DFS
private:                                            // local data
  VertexList path;                                  // the path
  Vertex target;                                    // the target vertex
  bool done;                                        // is target found?
protected:                                          // overridden functions
  void startVisit(const Vertex& v);                 // visit vertex
  void finishVisit(const Vertex& v);                // finished with vertex
  bool isDone() const;                              // done yet?
public:
  FindPath(const G& g) : DFS<G>(g) {}               // constructor
                                                    // find path from s to t
  VertexList operator()(const Vertex& s, const Vertex& t);
};
```

Code Fragment 13.10: The class FindPath, which extends the DFS class in order to compute a path from source vertex s to target vertex t.

The path function is presented in Code Fragment 13.11. After initializing search and the member data, it invokes the recursive DFS traversal. On termination, the vertex list containing the path is returned.

```
/* FindPath⟨G⟩ :: */                                // find path from s to t
  VertexList operator()(const Vertex& s, const Vertex& t) {
    initialize();                                   // initialize DFS
    path.clear();                                   // clear the path
    target = t;                                     // save the target
    done = false;
    dfsTraversal(s);                                // traverse starting at s
    return path;                                    // return the path
  }
```

Code Fragment 13.11: The overloaded () operator for class FindPath, which returns a path from s to t.

The approach is to perform a depth-first search traversal beginning at the source vertex. We maintain the path of discovery edges from the source to the current vertex. When we encounter an unexplored vertex, we add it to the end of the path. This is processed by overriding the function startVisit. When we finish processing a vertex, we remove it from the path. This is done by overriding function finishVisit. Thus, at any point, the vertex list consists of vertices along the path of the DFS tree from the source to the current vertex. The traversal is terminated when the target vertex is encountered. This is done by setting the boolean flag *done*. We override the function isDone to check for this event. These overridden functions are shown in Code Fragment 13.12.

```
/* FindPath⟨G⟩ :: */                          // visit vertex
  void startVisit(const Vertex& v) {
    path.push_back(v);                        // insert into path
    if (v == target) done = true;             // reached target vertex
  }
/* FindPath⟨G⟩ :: */                          // finished with vertex
  void finishVisit(const Vertex& v)
    { if (!done) path.pop_back(); }           // remove last vertex
/* FindPath⟨G⟩ :: */                          // done yet?
  bool isDone() const
    { return done; }
```

Code Fragment 13.12: The overridden functions used by class FindPath.

Our final example is class FindCycle, which finds a cycle in the connected component of a given start vertex s. (The cycle need not contain s.) The class is given in Code Fragment 13.13. Its member data consists of the edge list containing the cycle (*cycle*), the first vertex of the cycle (*cycleStart*), and a boolean variable that indicates whether the search is complete (*done*). The cycle function is given by overloading the "()" operator and passing in the start vertex s. It returns the cycle as a list of edges. If there is no such cycle, an empty list is returned.

```
template <typename G>
class FindCycle : public DFS<G> {
private:                                      // local data
  EdgeList cycle;                             // cycle storage
  Vertex cycleStart;                          // start of cycle
  bool done;                                  // cycle detected?
protected:                                    // overridden functions
  void traverseDiscovery(const Edge& e, const Vertex& from);
  void traverseBack(const Edge& e, const Vertex& from);
  void finishVisit(const Vertex& v);          // finished with vertex
  bool isDone() const;                        // done yet?
public:
  FindCycle(const G& g) : DFS<G>(g) {}        // constructor
  EdgeList operator()(const Vertex& s);       // find a cycle
};
```

Code Fragment 13.13: The class FindCycle, which extends the DFS class in order to compute a cycle in the connected component of a given start vertex s.

The cycle finding function is presented in Code Fragment 13.14. After initializing search and the member data, it invokes the recursive DFS traversal. As we show below, upon termination, the edge list consists of an initial prefix of edges from s to the vertex *cycleStart*, followed by the edges of the cycle. We traverse the edges of the cycle and remove each until reaching the first edge that is incident to *cycleStart*.

```
/* FindCycle⟨G⟩ :: */                                    // find a cycle
  EdgeList operator()(const Vertex& s) {
    initialize();                                        // initialize DFS
    cycle = EdgeList(); done = false;                    // initialize members
    dfsTraversal(s);                                     // do the search
    if (!cycle.empty() && s != cycleStart) {             // found a cycle?
      EdgeItor pe = cycle.begin();
      while (pe != cycle.end()) {                        // search for prefix
        if ((pe++)->isIncidentOn(cycleStart)) break;     // last edge of prefix?
      }
      cycle.erase(cycle.begin(), pe);                    // remove prefix
    }
    return cycle;                                        // return the cycle
  }
```

Code Fragment 13.14: The function of class FindCycle, which computes the cycle.

Our approach is based on performing a DFS traversal and storing the discovery edges in the edge list. As shown in Code Fragment 13.15, in traverseDiscovery, we push the current edge onto the edge list. In the function traverseBack, when we encounter a back edge we complete a cycle and set the boolean flag *done* to true to indicate that the cycle has been detected. We also set the variable *cycleStart* to the vertex on the opposite side of the back edge. When backing out of a vertex, as shown in the function finishVisit, we pop the most recent edge off the edge list, unless the cycle has been found.

```
/* FindCycle⟨G⟩ :: */                                    // discovery edge e
  void traverseDiscovery(const Edge& e, const Vertex& from)
    { if (!done) cycle.push_back(e); }                   // add edge to list
/* FindCycle⟨G⟩ :: */                                    // back edge e
  void traverseBack(const Edge& e, const Vertex& from) {
    if (!done) {                                         // no cycle yet?
      done = true;                                       // cycle now detected
      cycle.push_back(e);                                // insert final edge
      cycleStart = e.opposite(from);                     // save starting vertex
    }
  }
/* FindCycle⟨G⟩ :: */                                    // finished with vertex
  void finishVisit(const Vertex& v) {
    if (!cycle.empty() && !done)                         // not building a cycle?
      cycle.pop_back();                                  // remove this edge
  }
/* FindCycle⟨G⟩ :: */                                    // done yet?
  bool isDone() const
    { return done; }
```

Code Fragment 13.15: The overridden functions used by class FindCycle.

13.3.4 Polymorphic Objects and Decorator Values ⋆

Programming decorations that support multiple value types poses an interesting problem in C++. To illustrate the problem, let us suppose that we wish to implement an algorithm that associates a string with a persons name and an integer containing the persons current age with each vertex of a social network. Given a vertex v, we would like to associate it with two decorators, one for name and one for age.

```
v.set("name", "Bob");
v.set("age", 23);
```

C++'s strong type checking does not allow this, however, since we must specify the type of the attribute value, either string or int, but not both.

To solve this problem, we create a ***polymorphic*** value type (Section 2.2.2). We define a generic class, called Object, and we derive subclasses from this. Each subclass is specialized to store a single value of a particular type, for example, bool, char, int, or string. To make this more concrete, let us consider a simple example for just two types, int and string. It is straightforward to generalize this to other types, even user-defined types.

Our generic Object class is shown in Code Fragment 13.16. It has no data members, but it supports two member functions, intValue and stringValue. The first returns the value from an integer subclass and the second returns the value from a string subclass. An attempt to extract a string value from an integer object or an integer value from a string argument results in an exception being thrown.

```
class Object {                                          // generic object
public:
  int      intValue()    const throw(BadCast);
  string   stringValue() const throw(BadCast);
};
```

Code Fragment 13.16: A generic class, called Object, for storing a polymorphic object of type int or string.

Next, we derive two concrete subclasses from Object. The first, called Integer, stores a single integer, and the second, called String, stores a single STL string. These are shown in Code Fragment 13.17. In addition to a simple constructor, they each provide a member function getValue, which returns the stored value.

Finally, we define the member functions intValue and stringValue of class Object. We show only intValue in Code Fragment 13.18 (stringValue is analogous). This function assumes that the underlying object is a pointer to an Integer. It attempts to dynamically cast itself to an Integer pointer. If successful, it returns the resulting integer value. If not, an exception is thrown.

```
class Integer : public Object {
private:
  int value;
public:
  Integer(int v = 0) : value(v) { }
  int getValue() const
        { return value; }
};
```

```
class String : public Object {
private:
  string value;
public:
  String(string v = "") : value(v) { }
  string getValue() const
        { return value; }
};
```

Code Fragment 13.17: Concrete subclasses, Integer and String, for storing a single integer and a single string, respectively.

```
int Object::intValue() const throw(BadCast) {       // cast to Integer
  const Integer* p = dynamic_cast<const Integer*>(this);
  if (p == NULL) throw BadCast("Illegal attempt to cast to Integer");
  return p->getValue();
}
```

Code Fragment 13.18: The member function intValue of class Object, which returns the underlying integer value.

To show how to apply this useful polymorphic object, let us return to our earlier example. Recall that v is a vertex to which we want to assign two attributes, a name and an age. We create new entities, the first of type String and the second of type Integer. We initialize each with the desired value. Because these are subclasses of Object, we may store these entities as decorators as shown in Code Fragment 13.19.

```
Decorator v;                                  // a decorable object
v.set("name", new String("Bob"));             // store name as "Bob"
v.set("age", new Integer(23));                // store age as 23
// ...
string n = v.get("name")->stringValue();      // n = "Bob"
int a = v.get("age")->intValue();             // a = 23
```

Code Fragment 13.19: Example use of Object with a polymorphic dictionary.

When we extract the values of these decorators, we make use of the fact that we know that the name is a string, and the age is an integer. Thus, we may apply the appropriate function, stringValue or intValue, to extract the desired attribute value. This example shows the usefulness of polymorphic behavior of objects in C++.

13.3.5 Breadth-First Search

In this section, we consider a different traversal algorithm, called ***breadth-first search*** (BFS). Like DFS, BFS traverses a connected component of a graph, and in so doing it defines a useful spanning tree. BFS is less "adventurous" than DFS, however. Instead of wandering the graph, BFS proceeds in rounds and subdivides the vertices into ***levels***. BFS can also be thought of as a traversal using a string and paint, with BFS unrolling the string in a more conservative manner.

BFS starts at vertex s, which is at level 0 and defines the "anchor" for our string. In the first round, we let out the string the length of one edge and we visit all the vertices we can reach without unrolling the string any farther. In this case, we visit, and paint as "visited," the vertices adjacent to the start vertex s—these vertices are placed into level 1. In the second round, we unroll the string the length of two edges and we visit all the new vertices we can reach without unrolling our string any farther. These new vertices, which are adjacent to level 1 vertices and not previously assigned to a level, are placed into level 2, and so on. The BFS traversal terminates when every vertex has been visited.

Pseudo-code for a BFS starting at a vertex s is shown in Code Fragment 13.20. We use auxiliary space to label edges, mark visited vertices, and store collections associated with levels. That is, the collections L_0, L_1, L_2, and so on, store the vertices that are in level 0, level 1, level 2, and so on. These collections could, for example, be implemented as queues. They also allow BFS to be nonrecursive.

Algorithm BFS(s):
  initialize collection L_0 to contain vertex s
  $i \leftarrow 0$
  while L_i is not empty **do**
    create collection L_{i+1} to initially be empty
    for all vertices v in L_i **do**
      for all edges e in v.incidentEdges() **do**
        if edge e is unexplored **then**
          $w \leftarrow e$.opposite(v)
          if vertex w is unexplored **then**
            label e as a discovery edge
            insert w into L_{i+1}
          else
            label e as a cross edge
    $i \leftarrow i + 1$

Code Fragment 13.20: The BFS algorithm.

We illustrate a BFS traversal in Figure 13.7.

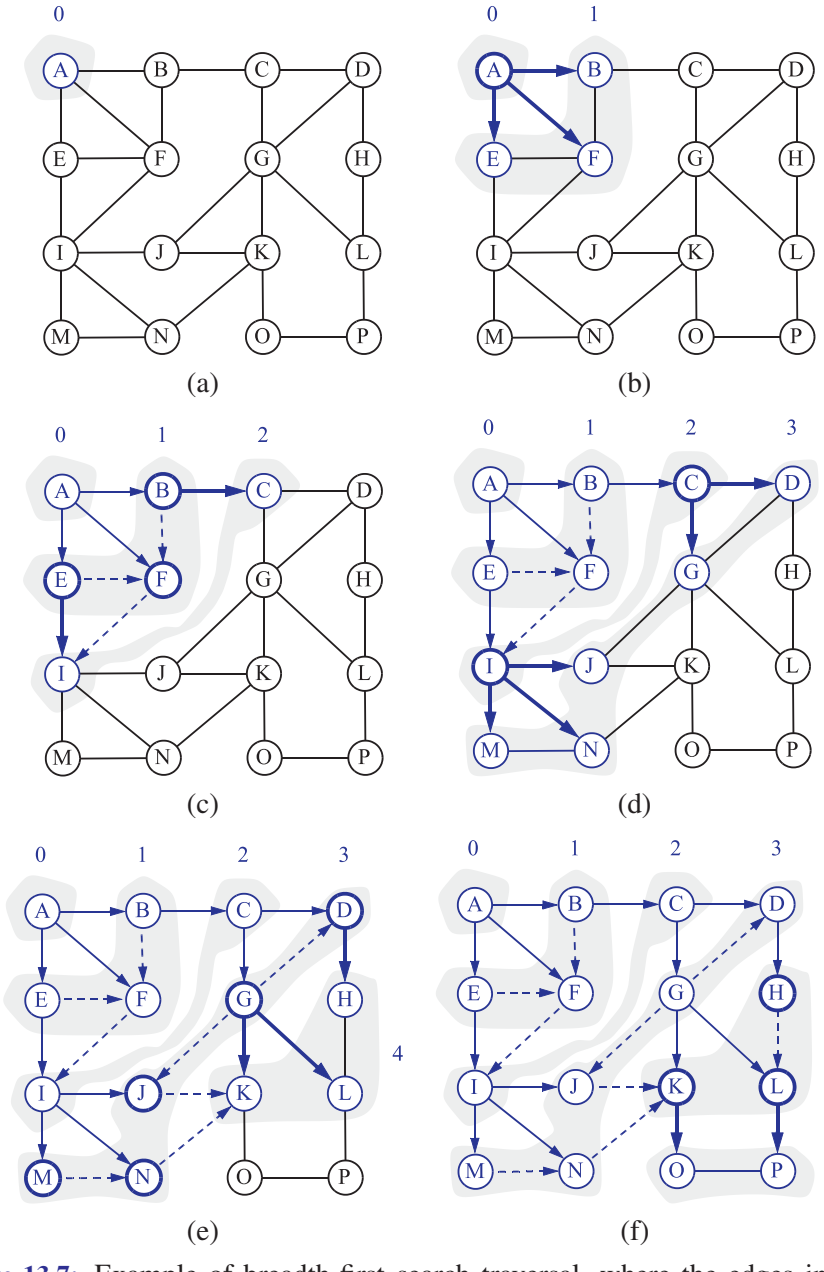


Figure 13.7: Example of breadth-first search traversal, where the edges incident on a vertex are explored by the alphabetical order of the adjacent vertices. The discovery edges are shown with solid lines and the cross edges are shown with dashed lines: (a) graph before the traversal; (b) discovery of level 1; (c) discovery of level 2; (d) discovery of level 3; (e) discovery of level 4; (f) discovery of level 5.

One of the nice properties of the BFS approach is that, in performing the BFS traversal, we can label each vertex by the length of a shortest path (in terms of the number of edges) from the start vertex s. In particular, if vertex v is placed into level i by a BFS starting at vertex s, then the length of a shortest path from s to v is i.

As with DFS, we can visualize the BFS traversal by orienting the edges along the direction in which they are explored during the traversal, and by distinguishing the edges used to discover new vertices, called ***discovery edges***, from those that lead to already visited vertices, called ***cross edges***. (See Figure 13.7(f).) As with the DFS, the discovery edges form a spanning tree, which in this case we call the BFS tree. We do not call the nontree edges "back edges" in this case, however, because none of them connects a vertex to one of its ancestors. Every nontree edge connects a vertex v to another vertex that is neither v's ancestor nor its descendent.

The BFS traversal algorithm has a number of interesting properties, some of which we explore in the proposition that follows.

Proposition 13.14: *Let G be an undirected graph on which a BFS traversal starting at vertex s has been performed. Then*

- *The traversal visits all vertices in the connected component of s*
- *The discovery-edges form a spanning tree T, which we call the BFS tree, of the connected component of s*
- *For each vertex v at level i, the path of the BFS tree T between s and v has i edges, and any other path of G between s and v has at least i edges*
- *If (u,v) is an edge that is not in the BFS tree, then the level numbers of u and v differ by at most 1*

We leave the justification of this proposition as an exercise (Exercise C-13.13). The analysis of the running time of BFS is similar to the one of DFS, which implies the following.

Proposition 13.15: *Let G be a graph with n vertices and m edges represented with the adjacency list structure. A BFS traversal of G takes $O(n+m)$ time. Also, there exist $O(n+m)$-time algorithms based on BFS for the following problems:*

- *Testing whether G is connected*
- *Computing a spanning tree of G, if G is connected*
- *Computing the connected components of G*
- *Given a start vertex s of G, computing, for every vertex v of G, a path with the minimum number of edges between s and v, or reporting that no such path exists*
- *Computing a cycle in G, or reporting that G has no cycles*

13.4 Directed Graphs

In this section, we consider issues that are specific to directed graphs. Recall that a directed graph (***digraph***), is a graph whose edges are all directed.

Methods Dealing with Directed Edges

In order to allow some or all the edges in a graph to be directed, we add the following functions to the graph ADT.

e.isDirected(): Test whether edge e is directed.

e.origin(): Return the origin vertex of edge e.

e.dest(): Return the destination vertex of edge e.

insertDirectedEdge(v,w,x): Insert and return a new directed edge with origin v and destination w and storing element x.

Also, if an edge e is directed, the function e.endVertices() should return a vertex list whose first element is the origin of e, and whose second element is the destination of e. The running time for the functions e.isDirected(), e.origin(), and e.dest() should be $O(1)$, and the running time of the function insertDirectedEdge(v,w,x) should match that of undirected edge insertion.

Reachability

One of the most fundamental issues with directed graphs is the notion of ***reachability***, which deals with determining which vertices can be reached by a path in a directed graph. A traversal in a directed graph always goes along directed paths, that is, paths where all the edges are traversed according to their respective directions. Given vertices u and v of a digraph G, we say that u ***reaches*** v (and v is ***reachable*** from u) if G has a directed path from u to v. We also say that a vertex v reaches an edge (w,z) if v reaches the origin vertex w of the edge.

A digraph G is ***strongly connected*** if for any two vertices u and v of G, u reaches v and v reaches u. A ***directed cycle*** of G is a cycle where all the edges are traversed according to their respective directions. (Note that G may have a cycle consisting of two edges with opposite direction between the same pair of vertices.) A digraph G is ***acyclic*** if it has no directed cycles. (See Figure 13.8 for some examples.)

The ***transitive closure*** of a digraph G is the digraph G^* such that the vertices of G^* are the same as the vertices of G, and G^* has an edge (u,v), whenever G has a directed path from u to v. That is, we define G^* by starting with the digraph G and adding in an extra edge (u,v) for each u and v such that v is reachable from u (and there isn't already an edge (u,v) in G).

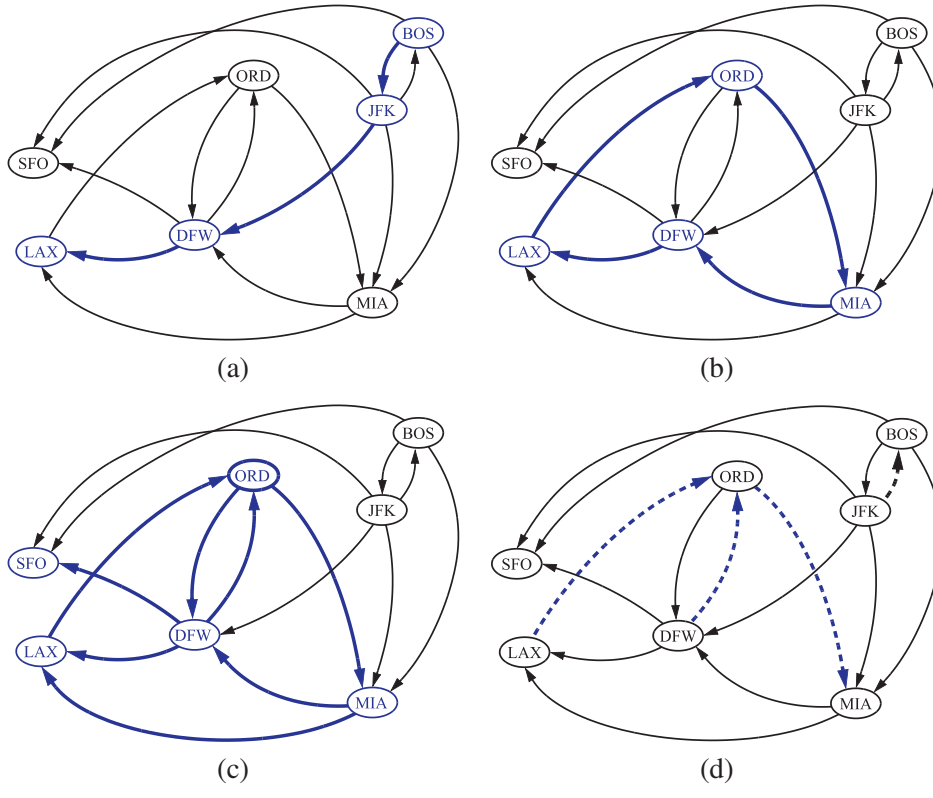


Figure 13.8: Examples of reachability in a digraph: (a) a directed path from BOS to LAX is drawn in blue; (b) a directed cycle (ORD, MIA, DFW, LAX, ORD) is shown in blue; its vertices induce a strongly connected subgraph; (c) the subgraph of the vertices and edges reachable from ORD is shown in blue; (d) removing the dashed blue edges gives an acyclic digraph.

Interesting problems that deal with reachability in a digraph G include the following:

- Given vertices u and v, determine whether u reaches v
- Find all the vertices of G that are reachable from a given vertex s
- Determine whether G is strongly connected
- Determine whether G is acyclic
- Compute the transitive closure G^* of G

In the remainder of this section, we explore some efficient algorithms for solving these problems.

13.4.1 Traversing a Digraph

As with undirected graphs, we can explore a digraph in a systematic way with methods akin to the depth-first search (DFS) and breadth-first search (BFS) algorithms defined previously for undirected graphs (Sections 13.3.1 and 13.3.5). Such explorations can be used, for example, to answer reachability questions. The directed depth-first search and breadth-first search methods we develop in this section for performing such explorations are very similar to their undirected counterparts. In fact, the only real difference is that the directed depth-first search and breadth-first search methods only traverse edges according to their respective directions.

The directed version of DFS starting at a vertex v can be described by the recursive algorithm in Code Fragment 13.21. (See Figure 13.9.)

Algorithm DirectedDFS(v):
  Mark vertex v as visited.
  for each outgoing edge (v, w) of v **do**
    if vertex w has not been visited **then**
      Recursively call DirectedDFS(w).

Code Fragment 13.21: The DirectedDFS algorithm.

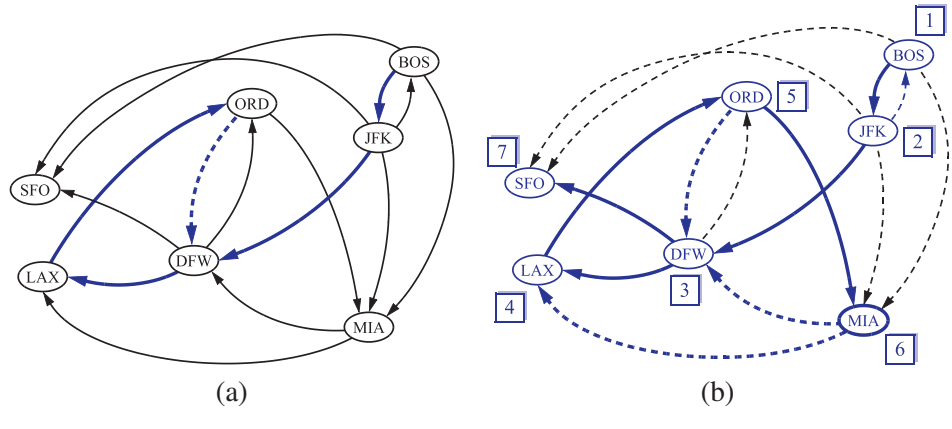


Figure 13.9: DFS in a digraph starting at vertex BOS: (a) intermediate step, where, for the first time, an already visited vertex (DFW) is reached; (b) the completed DFS. The tree edges are shown with solid blue lines, the back edges are shown with dashed blue lines, and the forward and cross edges are shown with dashed black lines. The order in which the vertices are visited is indicated by a label next to each vertex. The edge (ORD, DFW) is a back edge, but (DFW, ORD) is a forward edge. Edge (BOS, SFO) is a forward edge, and (SFO, LAX) is a cross edge.

A DFS on a digraph G partitions the edges of G reachable from the starting vertex into ***tree edges*** or ***discovery edges***, which lead us to discover a new vertex, and ***nontree edges***, which take us to a previously visited vertex. The tree edges form a tree rooted at the starting vertex, called the ***depth-first search*** tree. There are three kinds of nontree edges:

- ***Back edges***, which connect a vertex to an ancestor in the DFS tree
- ***Forward edges***, which connect a vertex to a descendent in the DFS tree
- ***Cross edges***, which connect a vertex to a vertex that is neither its ancestor nor its descendent

Refer back to Figure 13.9(b) to see an example of each type of nontree edge.

Proposition 13.16: *Let G be a digraph. Depth-first search on G starting at a vertex s visits all the vertices of G that are reachable from s. Also, the DFS tree contains directed paths from s to every vertex reachable from s.*

Justification: Let V_s be the subset of vertices of G visited by DFS starting at vertex s. We want to show that V_s contains s and every vertex reachable from s belongs to V_s. Suppose now, for the sake of a contradiction, that there is a vertex w reachable from s that is not in V_s. Consider a directed path from s to w, and let (u,v) be the first edge on such a path taking us out of V_s, that is, u is in V_s but v is not in V_s. When DFS reaches u, it explores all the outgoing edges of u, and thus must also reach vertex v via edge (u,v). Hence, v should be in V_s, and we have obtained a contradiction. Therefore, V_s must contain every vertex reachable from s. ■

Analyzing the running time of the directed DFS method is analogous to that for its undirected counterpart. In particular, a recursive call is made for each vertex exactly once, and each edge is traversed exactly once (from its origin). Hence, if n_s vertices and m_s edges are reachable from vertex s, a directed DFS starting at s runs in $O(n_s + m_s)$ time, provided the digraph is represented with a data structure that supports constant-time vertex and edge methods. The adjacency list structure satisfies this requirement, for example.

By Proposition 13.16, we can use DFS to find all the vertices reachable from a given vertex, and hence to find the transitive closure of G. That is, we can perform a DFS, starting from each vertex v of G, to see which vertices w are reachable from v, adding an edge (v,w) to the transitive closure for each such w. Likewise, by repeatedly traversing digraph G with a DFS, starting in turn at each vertex, we can easily test whether G is strongly connected. That is, G is strongly connected if each DFS visits all the vertices of G.

Thus, we may immediately derive the proposition that follows.

Proposition 13.17: *Let G be a digraph with n vertices and m edges. The following problems can be solved by an algorithm that traverses G n times using DFS, runs in $O(n(n+m))$ time, and uses $O(n)$ auxiliary space:*

- *Computing, for each vertex v of G, the subgraph reachable from v*
- *Testing whether G is strongly connected*
- *Computing the transitive closure G^* of G*

Testing for Strong Connectivity

Actually, we can determine if a directed graph G is strongly connected much faster than this, just by using two depth-first searches. We begin by performing a DFS of our directed graph G starting at an arbitrary vertex s. If there is any vertex of G that is not visited by this DFS and is not reachable from s, then the graph is not strongly connected. So, if this first DFS visits each vertex of G, then we reverse all the edges of G (using the reverseDirection function) and perform another DFS starting at s in this "reverse" graph. If every vertex of G is visited by this second DFS, then the graph is strongly connected because each of the vertices visited in this DFS can reach s. Since this algorithm makes just two DFS traversals of G, it runs in $O(n+m)$ time.

Directed Breadth-First Search

As with DFS, we can extend breadth-first search (BFS) to work for directed graphs. The algorithm still visits vertices level by level and partitions the set of edges into ***tree edges*** (or ***discovery edges***). Together these form a directed ***breadth-first search*** tree rooted at the start vertex and ***nontree edges***. Unlike the directed DFS method, however, the directed BFS method only leaves two kinds of nontree edges: ***back edges***, which connect a vertex to one of its ancestors, and ***cross edges***, which connect a vertex to another vertex that is neither its ancestor nor its descendent. There are no forward edges, which is a fact we explore in an exercise (Exercise C-13.9).

13.4.2 Transitive Closure

In this section, we explore an alternative technique for computing the transitive closure of a digraph. Let G be a digraph with n vertices and m edges. We compute the transitive closure of G in a series of rounds. We initialize $G_0 = G$. We also arbitrarily number the vertices of G as $v_1, v_2, \ldots, v_n$. We then begin the computation of the rounds, beginning with round 1. In a generic round k, we construct digraph G_k starting with $G_k = G_{k-1}$ and add to G_k the directed edge (v_i, v_j) if digraph G_{k-1} contains both the edges (v_i, v_k) and (v_k, v_j). In this way, we enforce a simple rule embodied in the proposition that follows.

Proposition 13.18: *For $i = 1, \ldots, n$, digraph G_k has an edge (v_i, v_j) if and only if digraph G has a directed path from v_i to v_j, whose intermediate vertices (if any) are in the set $\{v_1, \ldots, v_k\}$. In particular, G_n is equal to G^*, the transitive closure of G.*

Proposition 13.18 suggests a simple algorithm for computing the transitive closure of G that is based on the series of rounds we described above. This algorithm is known as the ***Floyd-Warshall algorithm***, and its pseudo-code is given in Code Fragment 13.22. From this pseudo-code, we can easily analyze the running time of the Floyd-Warshall algorithm assuming that the data structure representing G supports functions isAdjacentTo and insertDirectedEdge in $O(1)$ time. The main loop is executed n times and the inner loop considers each of $O(n^2)$ pairs of vertices, performing a constant-time computation for each one. Thus, the total running time of the Floyd-Warshall algorithm is $O(n^3)$.

Algorithm FloydWarshall(G):
Input: A digraph G with n vertices
Output: The transitive closure G^* of G
let $v_1, v_2, \ldots, v_n$ be an arbitrary numbering of the vertices of G
$G_0 \leftarrow G$
for $k \leftarrow 1$ to n **do**
  $G_k \leftarrow G_{k-1}$
  for all i, j in $\{1, \ldots, n\}$ with $i \neq j$ and $i, j \neq k$ **do**
    if both edges (v_i, v_k) and (v_k, v_j) are in G_{k-1} **then**
      add edge (v_i, v_j) to G_k (if it is not already present)
return G_n

Code Fragment 13.22: Pseudo-code for the Floyd-Warshall algorithm. This algorithm computes the transitive closure G^* of G by incrementally computing a series of digraphs $G_0, G_1, \ldots, G_n$, where $k = 1, \ldots, n$.

This description is actually an example of an algorithmic design pattern known as dynamic programming, which is discussed in more detail in Section 12.2. From the description and analysis above, we may immediately derive the following proposition.

Proposition 13.19: *Let G be a digraph with n vertices, and let G be represented by a data structure that supports lookup and update of adjacency information in $O(1)$ time. Then the Floyd-Warshall algorithm computes the transitive closure G^* of G in $O(n^3)$ time.*

We illustrate an example run of the Floyd-Warshall algorithm in Figure 13.10.

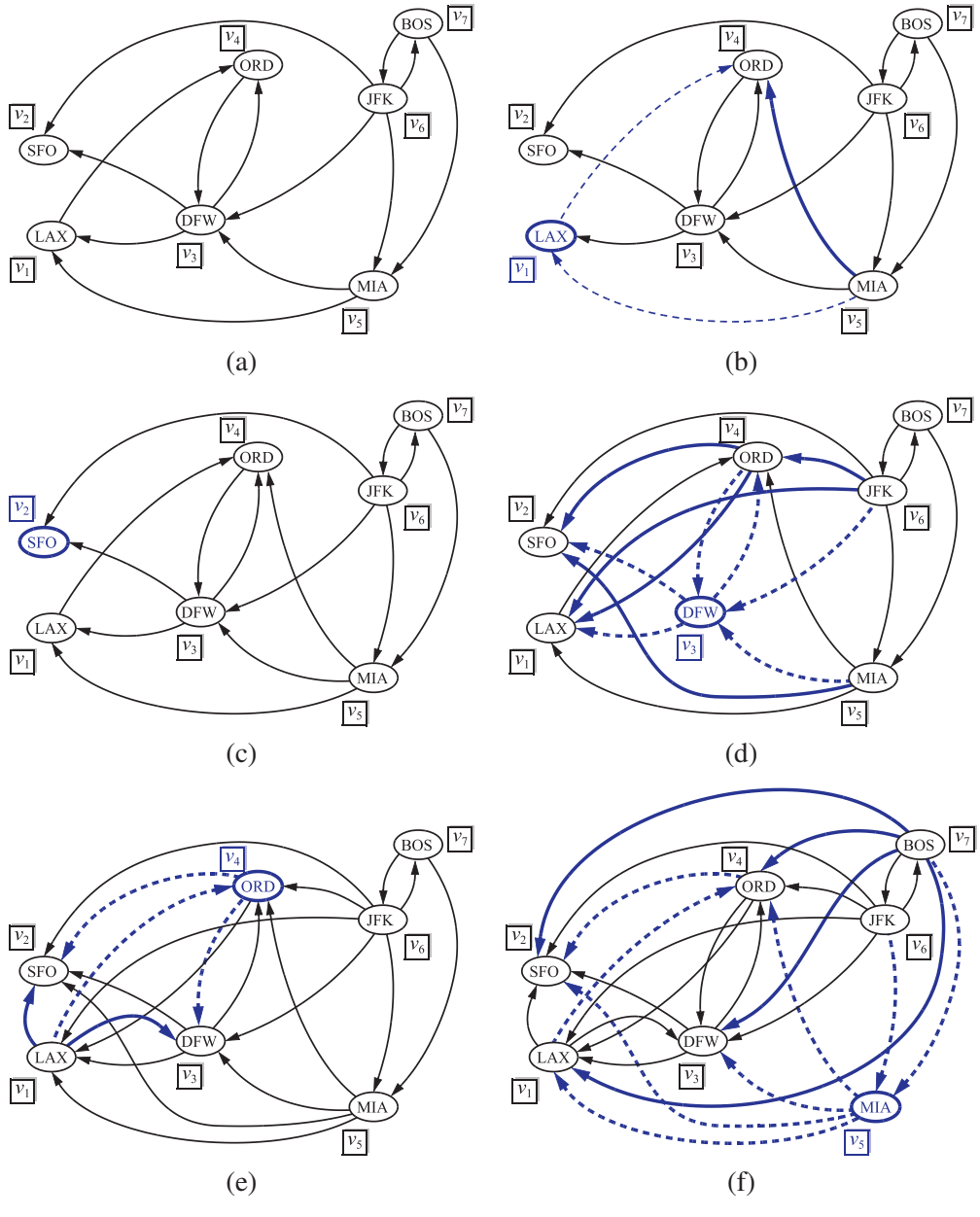


Figure 13.10: Sequence of digraphs computed by the Floyd-Warshall algorithm: (a) initial digraph $G = G_0$ and numbering of the vertices; (b) digraph G_1; (c) G_2; (d) G_3; (e) G_4; (f) G_5. (Note that $G_5 = G_6 = G_7$.) If digraph G_{k-1} has the edges (v_i, v_k) and (v_k, v_j), but not the edge (v_i, v_j). In the drawing of digraph G_k, we show edges (v_i, v_k) and (v_k, v_j) with dashed blue lines, and edge (v_i, v_j) with a thick blue line.

Performance of the Floyd-Warshall Algorithm

The running time of the Floyd-Warshall algorithm might appear to be slower than performing a DFS of a directed graph from each of its vertices, but this depends upon the representation of the graph. If a graph is represented using an adjacency matrix, then running the DFS method once on a directed graph G takes $O(n^2)$ time (we explore the reason for this in Exercise R-13.9). Thus, running DFS n times takes $O(n^3)$ time, which is no better than a single execution of the Floyd-Warshall algorithm, but the Floyd-Warshall algorithm would be much simpler to implement. Nevertheless, if the graph is represented using an adjacency list structure, then running the DFS algorithm n times would take $O(n(n+m))$ time to compute the transitive closure. Even so, if the graph is ***dense***, that is, if it has $\Omega(n^2)$ edges, then this approach still runs in $O(n^3)$ time and is more complicated than a single instance of the Floyd-Warshall algorithm. The only case where repeatedly calling the DFS method is better is when the graph is not dense and is represented using an adjacency list structure.

13.4.3 Directed Acyclic Graphs

Directed graphs without directed cycles are encountered in many applications. Such a digraph is often referred to as a ***directed acyclic graph***, or ***DAG***, for short. Applications of such graphs include the following:

- Inheritance between classes of a C++ program
- Prerequisites between courses of a degree program
- Scheduling constraints between the tasks of a project

Example 13.20: *In order to manage a large project, it is convenient to break it up into a collection of smaller tasks. The tasks, however, are rarely independent, because scheduling constraints exist between them. (For example, in a house building project, the task of ordering nails obviously precedes the task of nailing shingles to the roof deck.) Clearly, scheduling constraints cannot have circularities, because they would make the project impossible. (For example, in order to get a job you need to have work experience, but in order to get work experience you need to have a job.) The scheduling constraints impose restrictions on the order in which the tasks can be executed. Namely, if a constraint says that task a must be completed before task b is started, then a must precede b in the order of execution of the tasks. Thus, if we model a feasible set of tasks as vertices of a directed graph, and we place a directed edge from v to w whenever the task for v must be executed before the task for w, then we define a directed acyclic graph.*

The example above motivates the following definition. Let G be a digraph with n vertices. A ***topological ordering*** of G is an ordering $v_1, \ldots, v_n$ of the vertices of G such that for every edge (v_i, v_j) of G, $i < j$. That is, a topological ordering is an ordering such that any directed path in G traverses vertices in increasing order. (See Figure 13.11.) Note that a digraph may have more than one topological ordering.

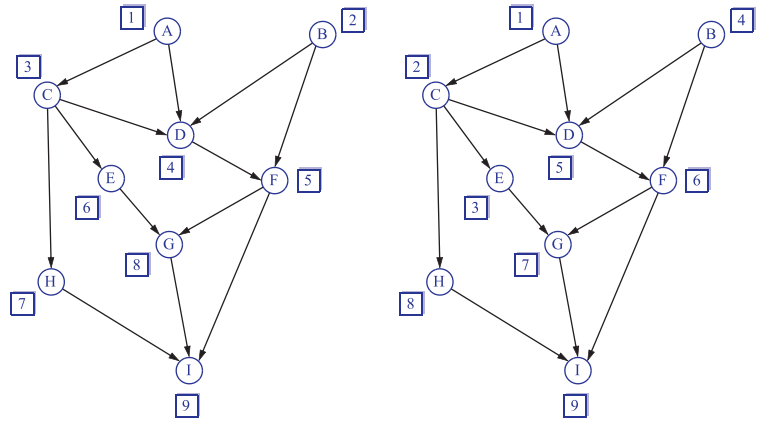


Figure 13.11: Two topological orderings of the same acyclic digraph.

Proposition 13.21: *G has a topological ordering if and only if it is acyclic.*

Justification: The necessity (the "only if" part of the statement) is easy to demonstrate. Suppose G is topologically ordered. Assume, for the sake of a contradiction, that G has a cycle consisting of edges $(v_{i_0}, v_{i_1}), (v_{i_1}, v_{i_2}), \ldots, (v_{i_{k-1}}, v_{i_0})$. Because of the topological ordering, we must have $i_0 < i_1 < \cdots < i_{k-1} < i_0$, which is clearly impossible. Thus, G must be acyclic.

We now argue the sufficiency of the condition (the "if" part). Suppose G is acyclic. We give an algorithmic description of how to build a topological ordering for G. Since G is acyclic, G must have a vertex with no incoming edges (that is, with in-degree 0). Let v_1 be such a vertex. Indeed, if v_1 did not exist, then in tracing a directed path from an arbitrary start vertex we would eventually encounter a previously visited vertex, thus contradicting the acyclicity of G. If we remove v_1 from G, together with its outgoing edges, the resulting digraph is still acyclic. Hence, the resulting digraph also has a vertex with no incoming edges, and we let v_2 be such a vertex. By repeating this process until the digraph becomes empty, we obtain an ordering $v_1, \ldots, v_n$ of the vertices of G. Because of the construction above, if (v_i, v_j) is an edge of G, then v_i must be deleted before v_j can be deleted, and thus $i < j$. Thus, $v_1, \ldots, v_n$ is a topological ordering. ■

Proposition 13.21's justification suggests an algorithm (Code Fragment 13.23), called ***topological sorting***, for computing a topological ordering of a digraph.

```
Algorithm TopologicalSort(G):
    Input: A digraph G with n vertices
    Output: A topological ordering v_1, ..., v_n of G
    S ← an initially empty stack.
    for all u in G.vertices() do
        Let incounter(u) be the in-degree of u.
        if incounter(u) = 0 then
            S.push(u)
    i ← 1
    while !S.empty() do
        u ← S.pop()
        Let u be vertex number i in the topological ordering.
        i ← i + 1
        for all outgoing edges (u, w) of u do
            incounter(w) ← incounter(w) − 1
            if incounter(w) = 0 then
                S.push(w)
```

Code Fragment 13.23: Pseudo-code for the topological sorting algorithm. (We show an example application of this algorithm in Figure 13.12.)

Proposition 13.22: *Let G be a digraph with n vertices and m edges. The topological sorting algorithm runs in $O(n+m)$ time using $O(n)$ auxiliary space, and either computes a topological ordering of G or fails to number some vertices, which indicates that G has a directed cycle.*

Justification: The initial computation of in-degrees and setup of the incounter variables can be done with a simple traversal of the graph, which takes $O(n+m)$ time. We use the decorator pattern to associate counter attributes with the vertices. Say that a vertex u is ***visited*** by the topological sorting algorithm when u is removed from the stack S. A vertex u can be visited only when incounter$(u) = 0$, which implies that all its predecessors (vertices with outgoing edges into u) were previously visited. As a consequence, any vertex that is on a directed cycle will never be visited, and any other vertex will be visited exactly once. The algorithm traverses all the outgoing edges of each visited vertex once, so its running time is proportional to the number of outgoing edges of the visited vertices. Therefore, the algorithm runs in $O(n+m)$ time. Regarding the space usage, observe that the stack S and the incounter variables attached to the vertices use $O(n)$ space. ■

As a side effect, the topological sorting algorithm of Code Fragment 13.23 also tests whether the input digraph G is acyclic. Indeed, if the algorithm terminates without ordering all the vertices, then the subgraph of the vertices that have not been ordered must contain a directed cycle.

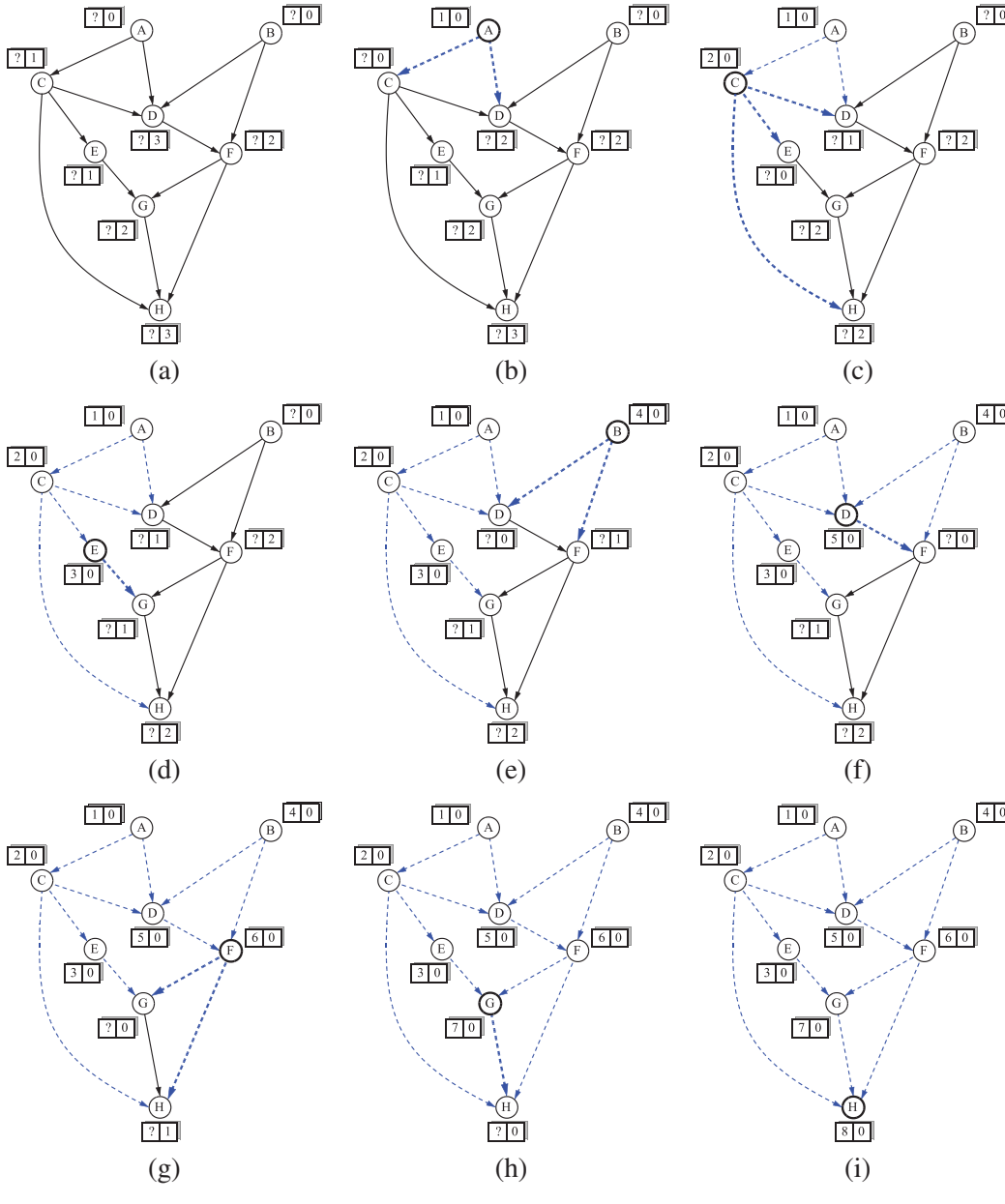


Figure 13.12: Example of a run of algorithm TopologicalSort (Code Fragment 13.23): (a) initial configuration; (b–i) after each while-loop iteration. The vertex labels show the vertex number and the current incounter value. The edges traversed are shown with dashed blue arrows. Thick lines denote the vertex and edges examined in the current iteration.

13.5 Shortest Paths

As we saw in Section 13.3.5, the breadth-first search strategy can be used to find a shortest path from some starting vertex to every other vertex in a connected graph. This approach makes sense in cases where each edge is as good as any other, but there are many situations where this approach is not appropriate. For example, we might be using a graph to represent a computer network (such as the Internet), and we might be interested in finding the fastest way to route a data packet between two computers. In this case, it is probably not appropriate for all the edges to be equal to each other, since some connections in a computer network are typically much faster than others (for example, some edges might represent slow phone-line connections while others might represent high-speed, fiber-optic connections). Likewise, we might want to use a graph to represent the roads between cities, and we might be interested in finding the fastest way to travel cross country. In this case, it is again probably not appropriate for all the edges to be equal to each other, because some inter-city distances will likely be much larger than others. Thus, it is natural to consider graphs whose edges are not weighted equally.

13.5.1 Weighted Graphs

A ***weighted graph*** is a graph that has a numeric (for example, integer) label $w(e)$ associated with each edge e, called the ***weight*** of edge e. We show an example of a weighted graph in Figure 13.13.

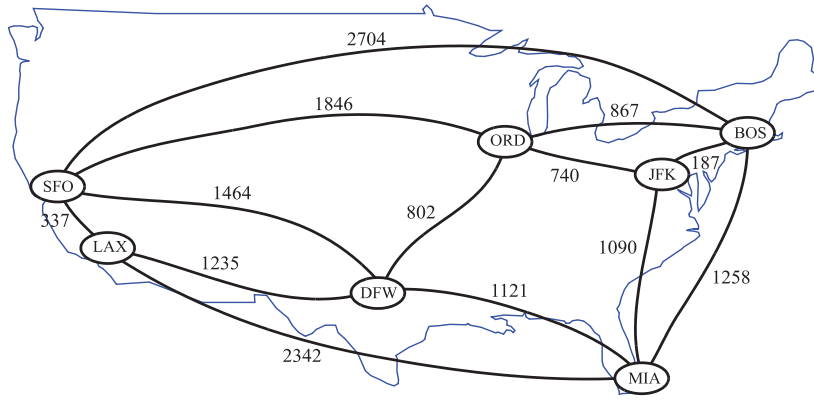


Figure 13.13: A weighted graph whose vertices represent major U.S. airports and whose edge weights represent distances in miles. This graph has a path from JFK to LAX of total weight 2,777 (going through ORD and DFW). This is the minimum weight path in the graph from JFK to LAX.

Defining Shortest Paths in a Weighted Graph

Let G be a weighted graph. The ***length*** (or weight) of a path is the sum of the weights of the edges of P. That is, if $P = ((v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k))$, then the length of P, denoted $w(P)$, is defined as

$$w(P) = \sum_{i=0}^{k-1} w((v_i, v_{i+1})).$$

The ***distance*** from a vertex v to a vertex u in G, denoted $d(v,u)$, is the length of a minimum length path (also called ***shortest path***) from v to u, if such a path exists.

People often use the convention that $d(v,u) = +\infty$ if there is no path at all from v to u in G. Even if there is a path from v to u in G, the distance from v to u may not be defined, however, if there is a cycle in G whose total weight is negative. For example, suppose vertices in G represent cities, and the weights of edges in G represent how much money it costs to go from one city to another. If someone were willing to actually pay us to go from say JFK to ORD, then the "cost" of the edge (JFK, ORD) would be negative. If someone else were willing to pay us to go from ORD to JFK, then there would be a negative-weight cycle in G and distances would no longer be defined. That is, anyone could now build a path (with cycles) in G from any city A to another city B that first goes to JFK and then cycles as many times as he or she likes from JFK to ORD and back, before going on to B. The existence of such paths would allow us to build arbitrarily low negative-cost paths (and, in this case, make a fortune in the process). But distances cannot be arbitrarily low negative numbers. Thus, any time we use edge weights to represent distances, we must be careful not to introduce any negative-weight cycles.

Suppose we are given a weighted graph G, and we are asked to find a shortest path from some vertex v to each other vertex in G, viewing the weights on the edges as distances. In this section, we explore efficient ways of finding all such shortest paths, if they exist. The first algorithm we discuss is for the simple, yet common, case when all the edge weights in G are nonnegative (that is, $w(e) \geq 0$ for each edge e of G); hence, we know in advance that there are no negative-weight cycles in G. Recall that the special case of computing a shortest path when all weights are equal to one was solved with the BFS traversal algorithm presented in Section 13.3.5.

There is an interesting approach for solving this ***single-source*** problem based on the ***greedy method*** design pattern (Section 12.4.2). Recall that in this pattern we solve the problem at hand by repeatedly selecting the best choice from among those available in each iteration. This paradigm can often be used in situations where we are trying to optimize some cost function over a collection of objects. We can add objects to our collection, one at a time, always picking the next one that optimizes the function from among those yet to be chosen.

13.5.2 Dijkstra's Algorithm

The main idea behind applying the greedy method pattern to the single-source, shortest-path problem is to perform a "weighted" breadth-first search starting at v. In particular, we can use the greedy method to develop an algorithm that iteratively grows a "cloud" of vertices out of v, with the vertices entering the cloud in order of their distances from v. Thus, in each iteration, the next vertex chosen is the vertex outside the cloud that is closest to v. The algorithm terminates when no more vertices are outside the cloud, at which point we have a shortest path from v to every other vertex of G. This approach is a simple, but nevertheless powerful, example of the greedy method design pattern.

A Greedy Method for Finding Shortest Paths

Applying the greedy method to the single-source, shortest-path problem, results in an algorithm known as ***Dijkstra's algorithm***. When applied to other graph problems, however, the greedy method may not necessarily find the best solution (such as in the so-called ***traveling salesman problem***, in which we wish to find the shortest path that visits all the vertices in a graph exactly once). Nevertheless, there are a number of situations in which the greedy method allows us to compute the best solution. In this chapter, we discuss two such situations: computing shortest paths and constructing a minimum spanning tree.

In order to simplify the description of Dijkstra's algorithm, we assume, in the following, that the input graph G is undirected (that is, all its edges are undirected) and simple (that is, it has no self-loops and no parallel edges). Hence, we denote the edges of G as unordered vertex pairs (u,z).

In Dijkstra's algorithm for finding shortest paths, the cost function we are trying to optimize in our application of the greedy method is also the function that we are trying to compute—the shortest path distance. This may at first seem like circular reasoning until we realize that we can actually implement this approach by using a "bootstrapping" trick, consisting of using an approximation to the distance function we are trying to compute, which in the end is equal to the true distance.

Edge Relaxation

Let us define a label $D[u]$ for each vertex u in V, which we use to approximate the distance in G from v to u. The meaning of these labels is that $D[u]$ always stores the length of the best path we have found ***so far*** from v to u. Initially, $D[v] = 0$ and $D[u] = +\infty$ for each $u \neq v$, and we define the set C (which is our "***cloud***" of vertices) to initially be the empty set $\emptyset$. At each iteration of the algorithm, we select a vertex u not in C with smallest $D[u]$ label, and we pull u into C. In the very first

iteration we will, of course, pull v into C. Once a new vertex u is pulled into C, we then update the label $D[z]$ of each vertex z that is adjacent to u and is outside of C, to reflect the fact that there may be a new and better way to get to z via u.

This update operation is known as a ***relaxation*** procedure, because it takes an old estimate and checks if it can be improved to get closer to its true value. (A metaphor for why we call this a relaxation comes from a spring that is stretched out and then "relaxed" back to its true resting shape.) In the case of Dijkstra's algorithm, the relaxation is performed for an edge (u,z) such that we have computed a new value of $D[u]$ and wish to see if there is a better value for $D[z]$ using the edge (u,z). The specific edge relaxation operation is as follows:

Edge Relaxation:

if $D[u] + w((u,z)) < D[z]$ **then**
$D[z] \leftarrow D[u] + w((u,z))$

We give the pseudo-code for Dijkstra's algorithm in Code Fragment 13.24. Note that we use a priority queue Q to store the vertices outside of the cloud C.

Algorithm ShortestPath(G,v):
Input: A simple undirected weighted graph G with nonnegative edge weights and a distinguished vertex v of G
Output: A label $D[u]$, for each vertex u of G, such that $D[u]$ is the length of a shortest path from v to u in G

```
Initialize D[v] ← 0 and D[u] ← +∞ for each vertex u ≠ v.
Let a priority queue Q contain all the vertices of G using the D labels as keys.
while Q is not empty do
  {pull a new vertex u into the cloud}
  u ← Q.removeMin()
  for each vertex z adjacent to u such that z is in Q do
    {perform the relaxation procedure on edge (u,z)}
    if D[u] + w((u,z)) < D[z] then
      D[z] ← D[u] + w((u,z))
      Change to D[z] the key of vertex z in Q.
return the label D[u] of each vertex u
```

Code Fragment 13.24: Dijkstra's algorithm for the single-source, shortest-path problem.

We illustrate several iterations of Dijkstra's algorithm in Figures 13.14 and 13.15.

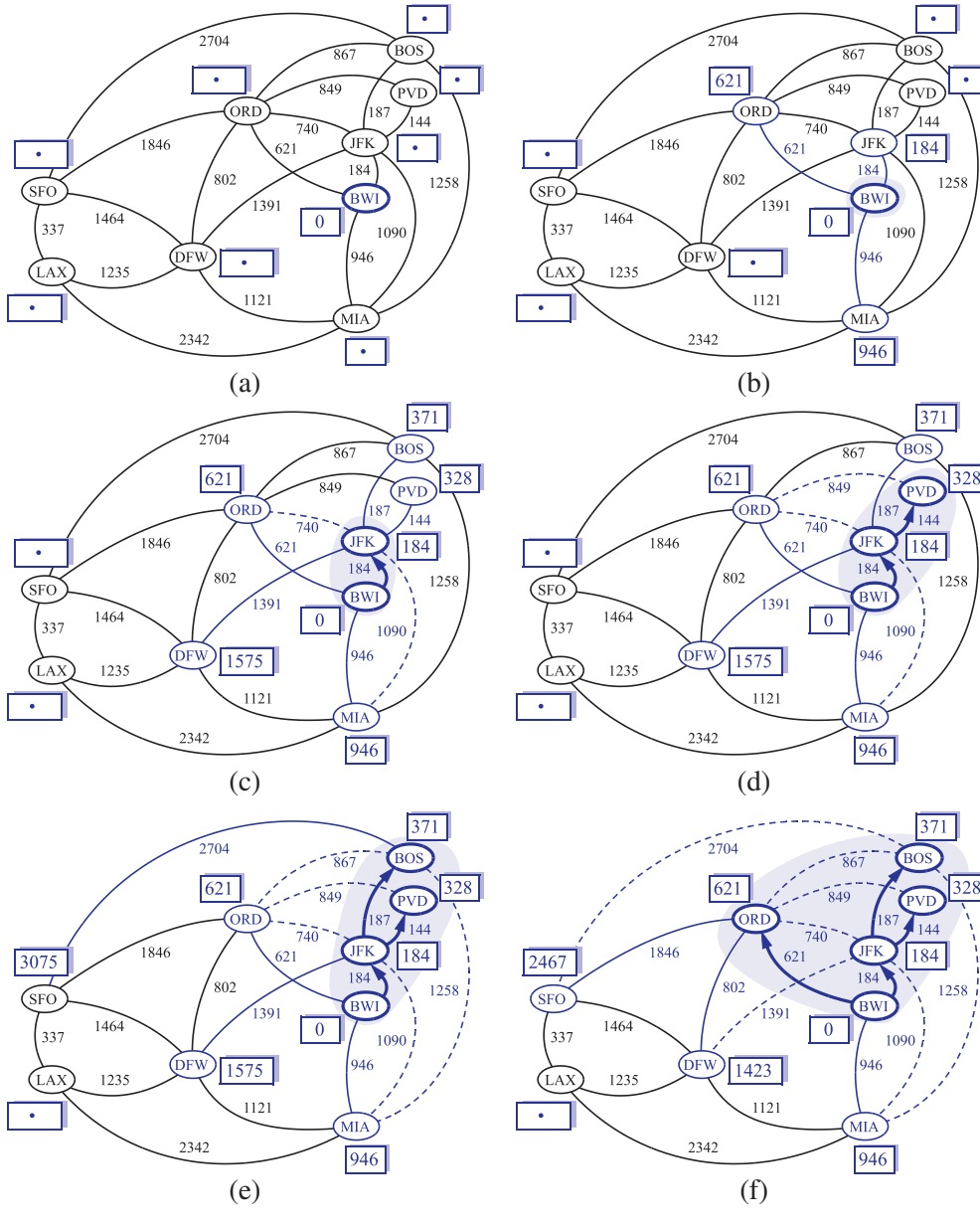


Figure 13.14: An execution of Dijkstra's algorithm on a weighted graph. The start vertex is BWI. A box next to each vertex v stores the label $D[v]$. The symbol • is used instead of $+\infty$. The edges of the shortest-path tree are drawn as thick blue arrows and, for each vertex u outside the "cloud," we show the current best edge for pulling in u with a solid blue line. (Continues in Figure 13.15.)

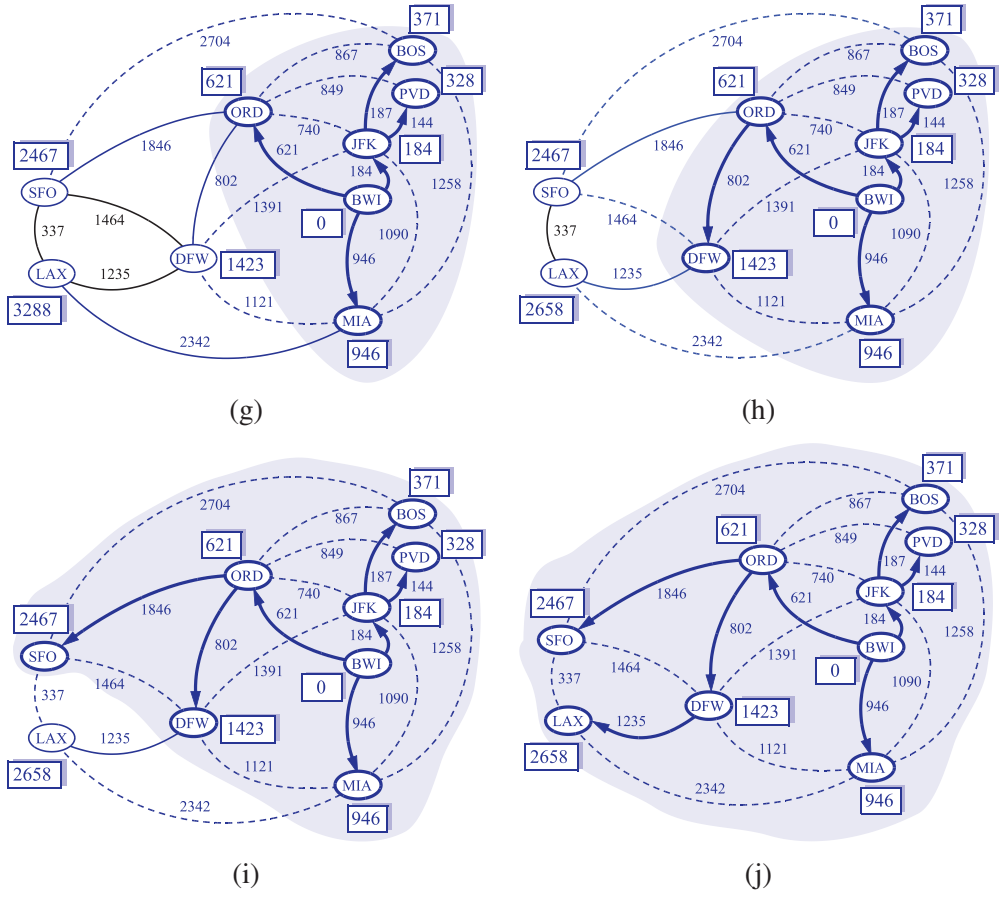


Figure 13.15: An example execution of Dijkstra's algorithm. (Continued from Figure 13.14.)

Why It Works

The interesting, and possibly even a little surprising, aspect of the Dijkstra algorithm is that, at the moment a vertex u is pulled into C, its label $D[u]$ stores the correct length of a shortest path from v to u. Thus, when the algorithm terminates, it will have computed the shortest-path distance from v to every vertex of G. That is, it will have solved the single-source, shortest-path problem.

It is probably not immediately clear why Dijkstra's algorithm correctly finds the shortest path from the start vertex v to each other vertex u in the graph. Why is it that the distance from v to u is equal to the value of the label $D[u]$ at the time vertex u is pulled into the cloud C (which is also the time u is removed from the priority queue Q)? The answer to this question depends on there being no negative-weight edges in the graph, since that allows the greedy method to work correctly, as we show in the proposition that follows.

Proposition 13.23: *In Dijkstra's algorithm, whenever a vertex u is pulled into the cloud, the label $D[u]$ is equal to $d(v,u)$, the length of a shortest path from v to u.*

Justification: Suppose that $D[t] > d(v,t)$ for some vertex t in V, and let u be the ***first*** vertex the algorithm pulled into the cloud C (that is, removed from Q) such that $D[u] > d(v,u)$. There is a shortest path P from v to u (for otherwise $d(v,u) = +\infty = D[u]$). Let us therefore consider the moment when u is pulled into C, and let z be the first vertex of P (when going from v to u) that is not in C at this moment. Let y be the predecessor of z in path P (note that we could have $y = v$). (See Figure 13.16.) We know, by our choice of z, that y is already in C at this point. Moreover, $D[y] = d(v,y)$, since u is the ***first*** incorrect vertex. When y was pulled into C, we tested (and possibly updated) $D[z]$ so that we had at that point

$$D[z] \leq D[y] + w((y,z)) = d(v,y) + w((y,z)).$$

But since z is the next vertex on the shortest path from v to u, this implies that

$$D[z] = d(v,z).$$

But we are now at the moment when we pick u, not z, to join C; hence

$$D[u] \leq D[z].$$

It should be clear that a subpath of a shortest path is itself a shortest path. Hence, since z is on the shortest path from v to u

$$d(v,z) + d(z,u) = d(v,u).$$

Moreover, $d(z,u) \geq 0$ because there are no negative-weight edges. Therefore

$$D[u] \leq D[z] = d(v,z) \leq d(v,z) + d(z,u) = d(v,u).$$

But this contradicts the definition of u; hence, there can be no such vertex u. ■

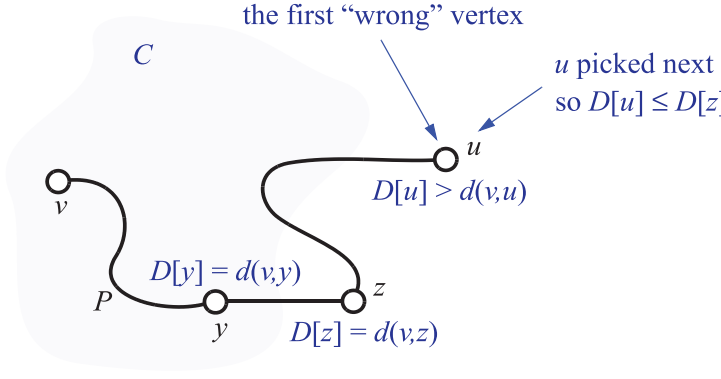


Figure 13.16: A schematic for the justification of Proposition 13.23.

The Running Time of Dijkstra's Algorithm

In this section, we analyze the time complexity of Dijkstra's algorithm. We denote the number of vertices and edges of the input graph G with n and m, respectively. We assume that the edge weights can be added and compared in constant time. Because of the high level of the description we gave for Dijkstra's algorithm in Code Fragment 13.24, analyzing its running time requires that we give more details on its implementation. Specifically, we should indicate the data structures used and how they are implemented.

Let us first assume that we are representing the graph G using an adjacency list structure. This data structure allows us to step through the vertices adjacent to u during the relaxation step in time proportional to their number. It still does not settle all the details for the algorithm, however, as we must say more about how to implement the other principle data structure in the algorithm—the priority queue Q.

An efficient implementation of the priority queue Q uses a heap (Section 8.3). This allows us to extract the vertex u with smallest D label (call to the removeMin function) in $O(\log n)$ time. As noted in the pseudo-code, each time we update a $D[z]$ label we need to update the key of z in the priority queue. Thus, we actually need a heap implementation of an adaptable priority queue (Section 8.4). If Q is an adaptable priority queue implemented as a heap, then this key update can, for example, be done using the replace(e,k), where e is the entry storing the key for the vertex z. If e is location aware, then we can easily implement such key updates in $O(\log n)$ time, since a location-aware entry for vertex z would allow Q to have immediate access to the entry e storing z in the heap (see Section 8.4.2). Assuming this implementation of Q, Dijkstra's algorithm runs in $O((n+m)\log n)$ time.

Referring back to Code Fragment 13.24, the details of the running-time analysis are as follows:

- Inserting all the vertices in Q with their initial key value can be done in $O(n \log n)$ time by repeated insertions, or in $O(n)$ time using bottom-up heap construction (see Section 8.3.6).
- At each iteration of the **while** loop, we spend $O(\log n)$ time to remove vertex u from Q, and $O(\text{degree}(v)\log n)$ time to perform the relaxation procedure on the edges incident on u.
- The overall running time of the **while** loop is

$$\sum_{v \text{ in } G} (1+\text{degree}(v))\log n,$$

  which is $O((n+m)\log n)$ by Proposition 13.6.

Note that if we wish to express the running time as a function of n only, then it is $O(n^2 \log n)$ in the worst case.

13.6 Minimum Spanning Trees

Suppose we wish to connect all the computers in a new office building using the least amount of cable. We can model this problem using a weighted graph G whose vertices represent the computers, and whose edges represent all the possible pairs (u,v) of computers, where the weight $w((v,u))$ of edge (v,u) is equal to the amount of cable needed to connect computer v to computer u. Rather than computing a shortest-path tree from some particular vertex v, we are interested instead in finding a (free) tree T that contains all the vertices of G and has the minimum total weight over all such trees. Methods for finding such a tree are the focus of this section.

Problem Definition

Given a weighted undirected graph G, we are interested in finding a tree T that contains all the vertices in G and minimizes the sum

$$w(T) = \sum_{(v,u) \text{ in } T} w((v,u)).$$

A tree, such as this, that contains every vertex of a connected graph G is said to be a ***spanning tree***, and the problem of computing a spanning tree T with smallest total weight is known as the ***minimum spanning tree*** (or ***MST***) problem.

The development of efficient algorithms for the minimum spanning tree problem predates the modern notion of computer science itself. In this section, we discuss two classic algorithms for solving the MST problem. These algorithms are both applications of the ***greedy method***, which, as was discussed briefly in the previous section, is based on choosing objects to join a growing collection by iteratively picking an object that minimizes some cost function. The first algorithm we discuss is Kruskal's algorithm, which "grows" the MST in clusters by considering edges in order of their weights. The second algorithm we discuss is the Prim-Jarník algorithm, which grows the MST from a single root vertex, much in the same way as Dijkstra's shortest-path algorithm.

As in Section 13.5.2, in order to simplify the description of the algorithms, we assume, in the following, that the input graph G is undirected (that is, all its edges are undirected) and simple (that is, it has no self-loops and no parallel edges). Hence, we denote the edges of G as unordered vertex pairs (u,z).

Before we discuss the details of these algorithms, however, let us give a crucial fact about minimum spanning trees that forms the basis of the algorithms.

A Crucial Fact About Minimum Spanning Trees

The two MST algorithms we discuss are based on the greedy method, which in this case depends crucially on the following fact. (See Figure 13.17.)

e Belongs to a Minimum Spanning Tree

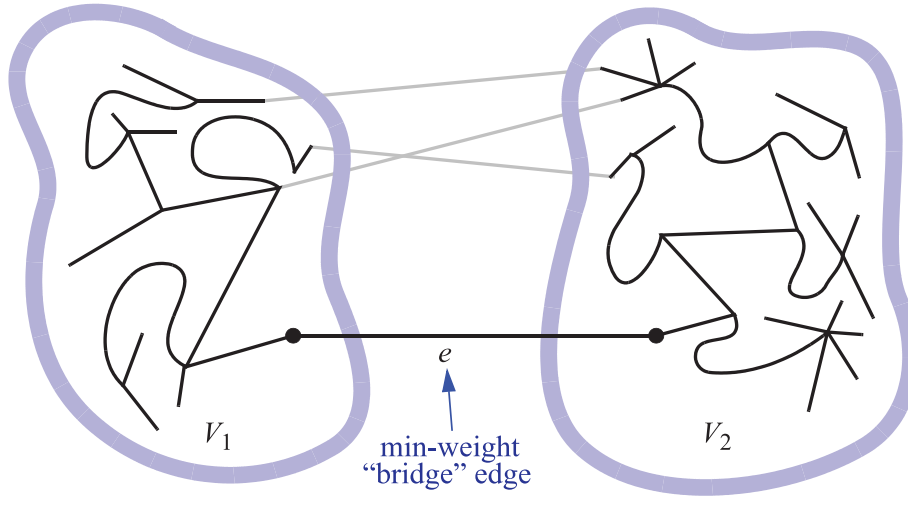


Figure 13.17: The crucial fact about minimum spanning trees.

Proposition 13.24: *Let G be a weighted connected graph, and let V_1 and V_2 be a partition of the vertices of G into two disjoint nonempty sets. Furthermore, let e be an edge in G with minimum weight from among those with one endpoint in V_1 and the other in V_2. There is a minimum spanning tree T that has e as one of its edges.*

Justification: Let T be a minimum spanning tree of G. If T does not contain edge e, the addition of e to T must create a cycle. Therefore, there is some edge f of this cycle that has one endpoint in V_1 and the other in V_2. Moreover, by the choice of e, $w(e) \leq w(f)$. If we remove f from $T \cup \{e\}$, we obtain a spanning tree whose total weight is no more than before. Since T is a minimum spanning tree, this new tree must also be a minimum spanning tree. ∎

In fact, if the weights in G are distinct, then the minimum spanning tree is unique. We leave the justification of this less crucial fact as an exercise (Exercise C-13.17). In addition, note that Proposition 13.24 remains valid even if the graph G contains negative-weight edges or negative-weight cycles, unlike the algorithms we presented for shortest paths.

13.6.1 Kruskal's Algorithm

The reason Proposition 13.24 is so important is that it can be used as the basis for building a minimum spanning tree. In Kruskal's algorithm, it is used to build the minimum spanning tree in clusters. Initially, each vertex is in its own cluster all by itself. The algorithm then considers each edge in turn, ordered by increasing weight. If an edge e connects two different clusters, then e is added to the set of edges of the minimum spanning tree, and the two clusters connected by e are merged into a single cluster. If, on the other hand, e connects two vertices that are already in the same cluster, then e is discarded. Once the algorithm has added enough edges to form a spanning tree, it terminates and outputs this tree as the minimum spanning tree.

We give pseudo-code for Kruskal's MST algorithm in Code Fragment 13.25 and we show the working of this algorithm in Figures 13.18, 13.19, and 13.20.

Algorithm Kruskal(G):
  Input: A simple connected weighted graph G with n vertices and m edges
  Output: A minimum spanning tree T for G
for each vertex v in G **do**
  Define an elementary cluster $C(v) \leftarrow \{v\}$.
Initialize a priority queue Q to contain all edges in G, using the weights as keys.
$T \leftarrow \emptyset$ {T will ultimately contain the edges of the MST}
while T has fewer than $n-1$ edges **do**
  $(u,v) \leftarrow Q$.removeMin()
  Let $C(v)$ be the cluster containing v, and let $C(u)$ be the cluster containing u.
  if $C(v) \neq C(u)$ **then**
    Add edge (v,u) to T.
    Merge $C(v)$ and $C(u)$ into one cluster, that is, union $C(v)$ and $C(u)$.
return tree T

Code Fragment 13.25: Kruskal's algorithm for the MST problem.

As mentioned before, the correctness of Kruskal's algorithm follows from the crucial fact about minimum spanning trees, Proposition 13.24. Each time Kruskal's algorithm adds an edge (v,u) to the minimum spanning tree T, we can define a partitioning of the set of vertices V (as in the proposition) by letting V_1 be the cluster containing v and letting V_2 contain the rest of the vertices in V. This clearly defines a disjoint partitioning of the vertices of V and, more importantly, since we are extracting edges from Q in order by their weights, e must be a minimum-weight edge with one vertex in V_1 and the other in V_2. Thus, Kruskal's algorithm always adds a valid minimum spanning tree edge.

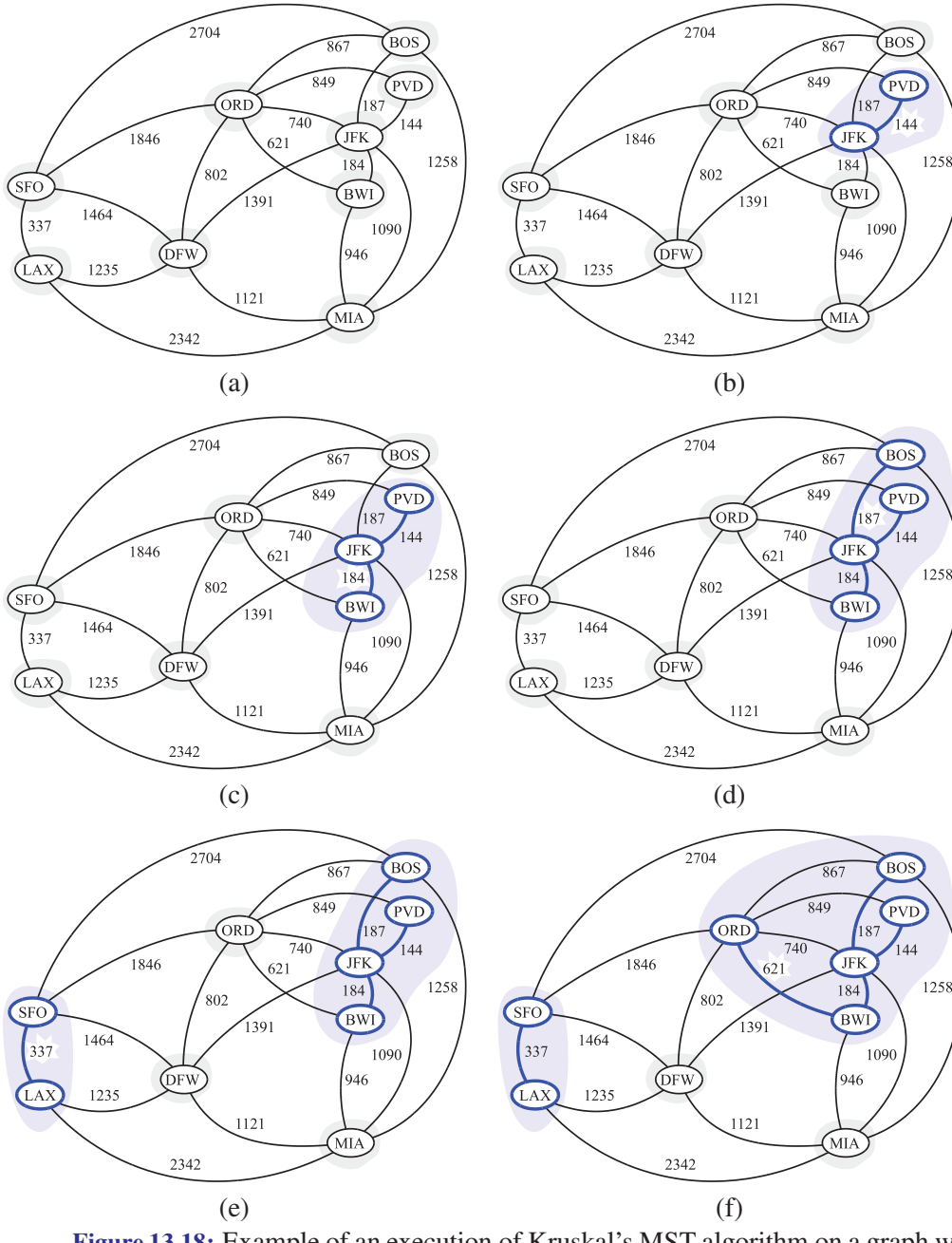


Figure 13.18: Example of an execution of Kruskal's MST algorithm on a graph with integer weights. We show the clusters as shaded regions and we highlight the edge being considered in each iteration. (Continues in Figure 13.19.)

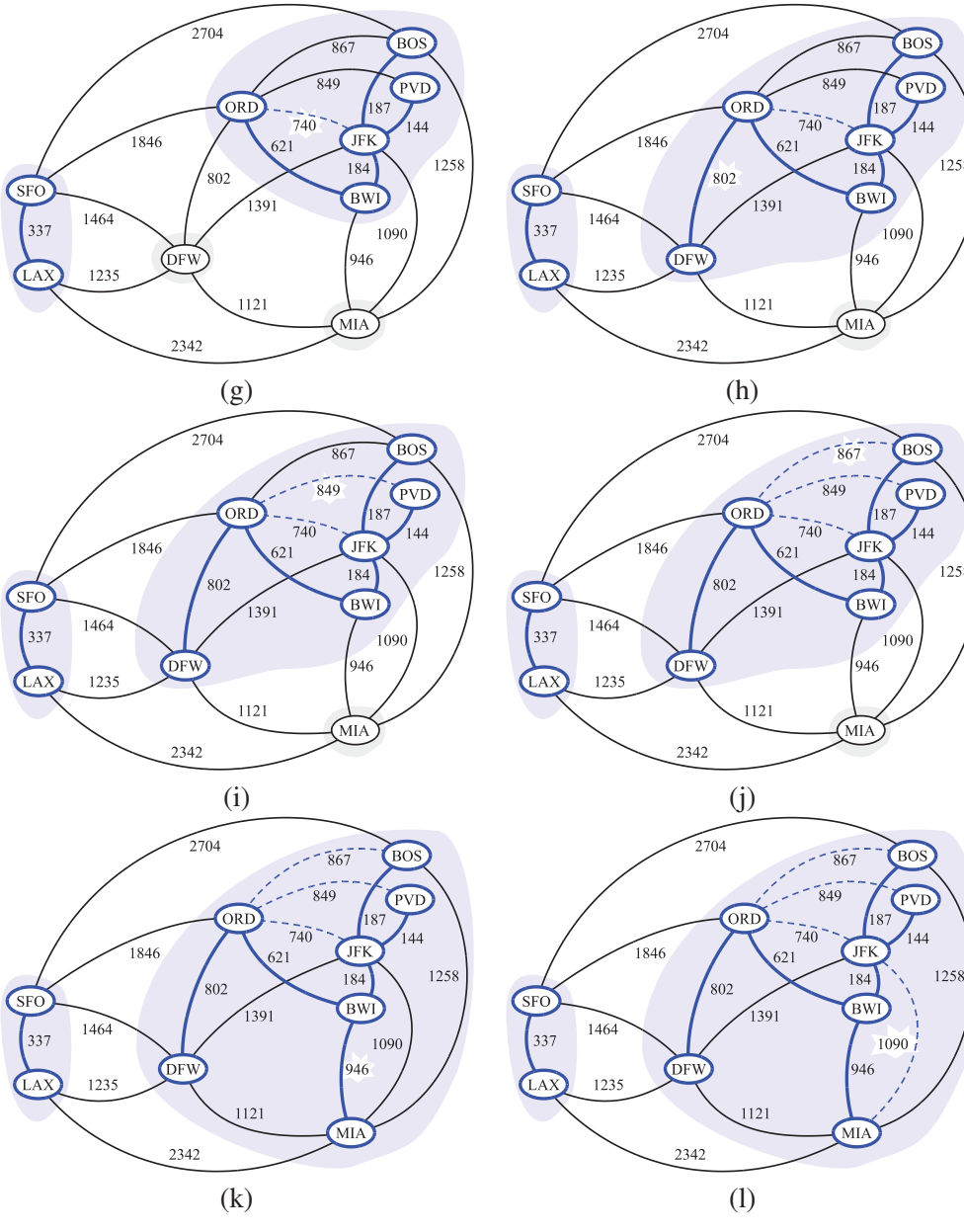


Figure 13.19: Example of an execution of Kruskal's MST algorithm. Rejected edges are shown dashed. (Continues in Figure 13.20.)

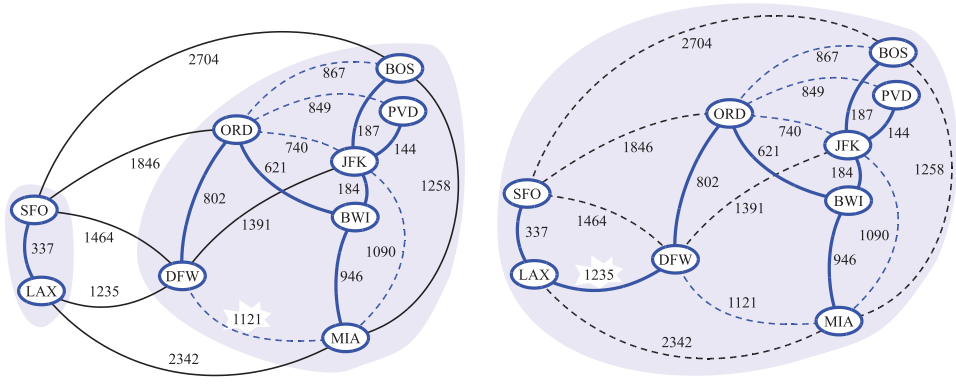


Figure 13.20: Example of an execution of Kruskal's MST algorithm. The edge considered in (n) merges the last two clusters, which concludes this execution of Kruskal's algorithm. (Continued from Figure 13.19.)

The Running Time of Kruskal's Algorithm

We denote the number of vertices and edges of the input graph G with n and m, respectively. Because of the high level of the description we gave for Kruskal's algorithm in Code Fragment 13.25, analyzing its running time requires that we give more details on its implementation. Specifically, we should indicate the data structures used and how they are implemented.

We can implement the priority queue Q using a heap. Thus, we can initialize Q in $O(m \log m)$ time by repeated insertions, or in $O(m)$ time using bottom-up heap construction (see Section 8.3.6). In addition, at each iteration of the **while** loop, we can remove a minimum-weight edge in $O(\log m)$ time, which actually is $O(\log n)$, since G is simple. Thus, the total time spent performing priority queue operations is no more than $O(m \log n)$.

We can represent each cluster C using one of the union-find partition data structures discussed in Section 11.4.3. Recall that the sequence-based union-find structure allows us to perform a series of N union and find operations in $O(N \log N)$ time, and the tree-based version can implement such a series of operations in $O(N \log^* N)$ time. Thus, since we perform $n-1$ calls to function union and at most m calls to find, the total time spent on merging clusters and determining the clusters that vertices belong to is no more than $O(m \log n)$ using the sequence-based approach or $O(m \log^* n)$ using the tree-based approach.

Therefore, using arguments similar to those used for Dijkstra's algorithm, we conclude that the running time of Kruskal's algorithm is $O((n+m) \log n)$, which can be simplified as $O(m \log n)$, since G is simple and connected.

13.6.2 The Prim-Jarník Algorithm

In the Prim-Jarník algorithm, we grow a minimum spanning tree from a single cluster starting from some "root" vertex v. The main idea is similar to that of Dijkstra's algorithm. We begin with some vertex v, defining the initial "cloud" of vertices C. Then, in each iteration, we choose a minimum-weight edge $e = (v,u)$, connecting a vertex v in the cloud C to a vertex u outside of C. The vertex u is then brought into the cloud C and the process is repeated until a spanning tree is formed. Again, the crucial fact about minimum spanning trees comes to play, because by always choosing the smallest-weight edge joining a vertex inside C to one outside C, we are assured of always adding a valid edge to the MST.

To efficiently implement this approach, we can take another cue from Dijkstra's algorithm. We maintain a label $D[u]$ for each vertex u outside the cloud C, so that $D[u]$ stores the weight of the best current edge for joining u to the cloud C. These labels allow us to reduce the number of edges that we must consider in deciding which vertex is next to join the cloud. We give the pseudo-code in Code Fragment 13.26.

Algorithm PrimJarnik(G):
Input: A weighted connected graph G with n vertices and m edges
Output: A minimum spanning tree T for G

Pick any vertex v of G
$D[v] \leftarrow 0$
for each vertex $u \neq v$ **do**
  $D[u] \leftarrow +\infty$
Initialize $T \leftarrow \emptyset$.
Initialize a priority queue Q with an entry $((u,null),D[u])$ for each vertex u, where $(u,null)$ is the element and $D[u]$ is the key.
while Q is not empty **do**
  $(u,e) \leftarrow Q$.removeMin()
  Add vertex u and edge e to T.
  for each vertex z adjacent to u such that z is in Q **do**
    {perform the relaxation procedure on edge (u,z)}
    if $w((u,z)) < D[z]$ **then**
      $D[z] \leftarrow w((u,z))$
      Change to $(z,(u,z))$ the element of vertex z in Q.
      Change to $D[z]$ the key of vertex z in Q.
return the tree T

Code Fragment 13.26: The Prim-Jarník algorithm for the MST problem.

Analyzing the Prim-Jarník Algorithm

Let n and m denote the number of vertices and edges of the input graph G, respectively. The implementation issues for the Prim-Jarník algorithm are similar to those for Dijkstra's algorithm. If we implement the adaptable priority queue Q as a heap that supports location-aware entries (Section 8.4.2), then we can extract the vertex u in each iteration in $O(\log n)$ time. In addition, we can update each $D[z]$ value in $O(\log n)$ time, as well, which is a computation considered at most once for each edge (u,z). The other steps in each iteration can be implemented in constant time. Thus, the total running time is $O((n+m)\log n)$, which is $O(m \log n)$.

Illustrating the Prim-Jarník Algorithm

We illustrate the Prim-Jarník algorithm in Figures 13.21 through 13.22.

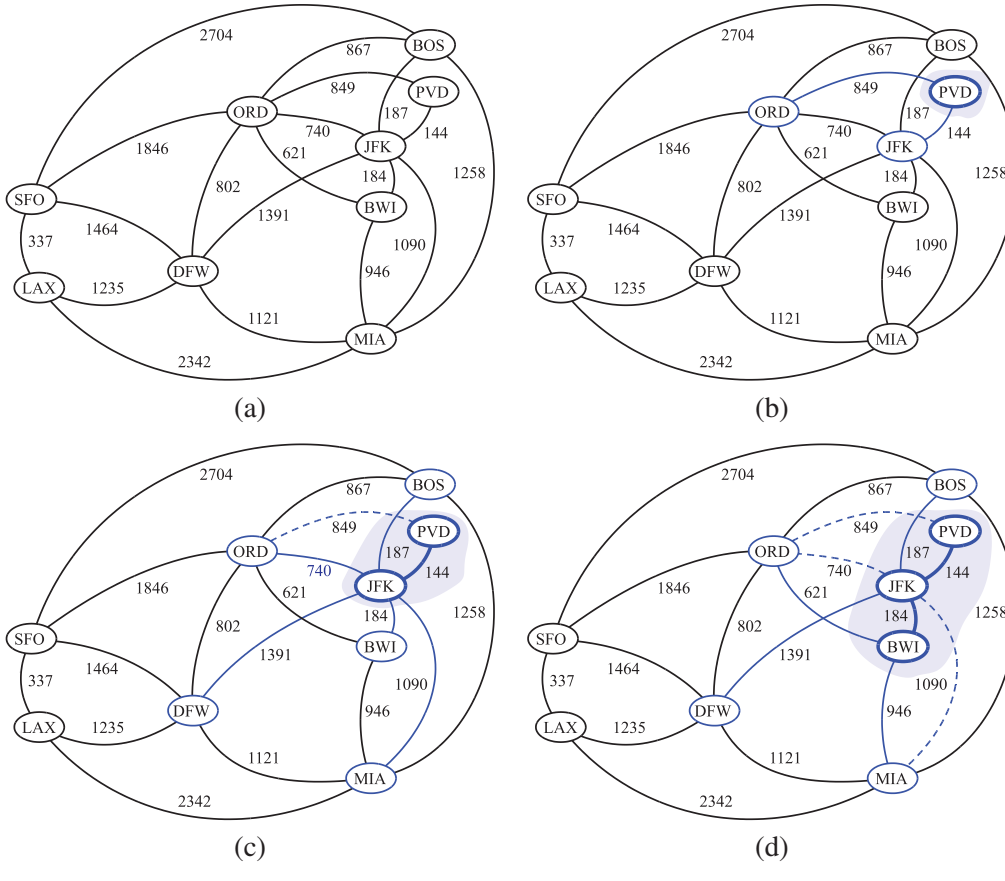


Figure 13.21: The Prim-Jarník MST algorithm. (Continues in Figure 13.22.)

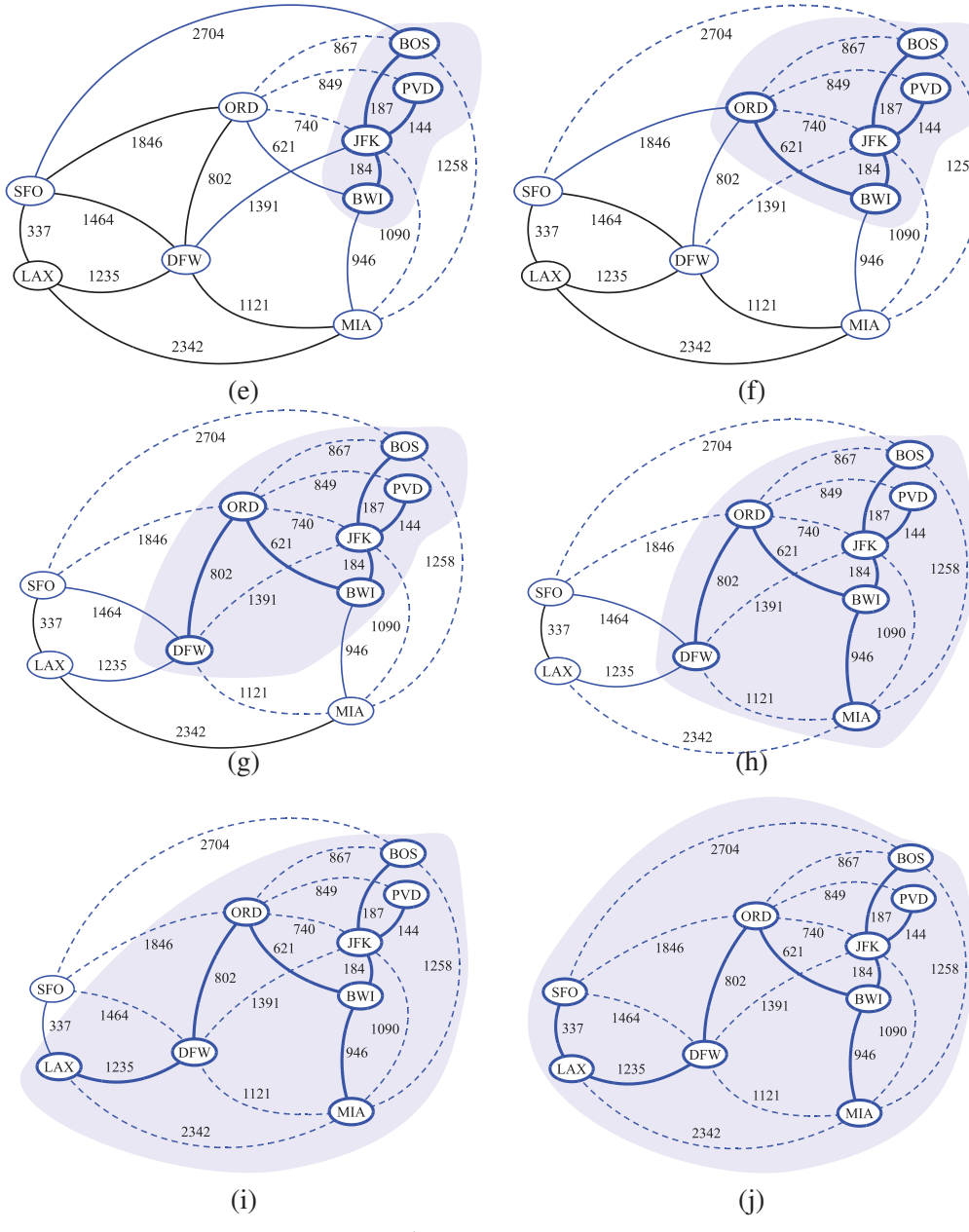


Figure 13.22: The Prim-Jarník MST algorithm. (Continued from Figure 13.21.)

13.7 Exercises

For help with exercises, please visit the web site, www.wiley.com/college/goodrich.

Reinforcement

R-13.1 Draw a simple undirected graph G that has 12 vertices, 18 edges, and 3 connected components. Why would it be impossible to draw G with 3 connected components if G had 66 edges?

R-13.2 Draw an adjacency list and adjacency matrix representation of the undirected graph shown in Figure 13.1.

R-13.3 Draw a simple connected directed graph with 8 vertices and 16 edges such that the in-degree and out-degree of each vertex is 2. Show that there is a single (nonsimple) cycle that includes all the edges of your graph, that is, you can trace all the edges in their respective directions without ever lifting your pencil. (Such a cycle is called an ***Euler tour***.)

R-13.4 Repeat the previous problem and then remove one edge from the graph. Show that now there is a single (nonsimple) path that includes all the edges of your graph. (Such a path is called an ***Euler path***.)

R-13.5 Bob loves foreign languages and wants to plan his course schedule for the following years. He is interested in the following nine language courses: LA15, LA16, LA22, LA31, LA32, LA126, LA127, LA141, and LA169. The course prerequisites are:

- LA15: (none)
- LA16: LA15
- LA22: (none)
- LA31: LA15
- LA32: LA16, LA31
- LA126: LA22, LA32
- LA127: LA16
- LA141: LA22, LA16
- LA169: LA32

Find the sequence of courses that allows Bob to satisfy all the prerequisites.

R-13.6 Suppose we represent a graph G having n vertices and m edges with the edge list structure. Why, in this case, does the insertVertex function run in $O(1)$ time while the eraseVertex function runs in $O(m)$ time?

R-13.7 Let G be a graph whose vertices are the integers 1 through 8, and let the adjacent vertices of each vertex be given by the table below:

Vertex	*Adjacent Vertices*
1	(2, 3, 4)
2	(1, 3, 4)
3	(1, 2, 4)
4	(1, 2, 3, 6)
5	(6, 7, 8)
6	(4, 5, 7)
7	(5, 6, 8)
8	(5, 7)

Assume that, in a traversal of G, the adjacent vertices of a given vertex are returned in the same order as they are listed in the table above.

a. Draw G.
b. Give the sequence of vertices of G visited using a DFS traversal starting at vertex 1.
c. Give the sequence of vertices visited using a BFS traversal starting at vertex 1.

R-13.8 Would you use the adjacency list structure or the adjacency matrix structure in each of the following cases? Justify your choice.

a. The graph has 10,000 vertices and 20,000 edges, and it is important to use as little space as possible.
b. The graph has 10,000 vertices and 20,000,000 edges, and it is important to use as little space as possible.
c. You need to answer the query isAdjacentTo as fast as possible, no matter how much space you use.

R-13.9 Explain why the DFS traversal runs in $O(n^2)$ time on an n-vertex simple graph that is represented with the adjacency matrix structure.

R-13.10 Draw the transitive closure of the directed graph shown in Figure 13.2.

R-13.11 Compute a topological ordering for the directed graph drawn with solid edges in Figure 13.8(d).

R-13.12 Can we use a queue instead of a stack as an auxiliary data structure in the topological sorting algorithm shown in Code Fragment 13.23? Why or why not?

R-13.13 Draw a simple, connected, weighted graph with 8 vertices and 16 edges, each with unique edge weights. Identify one vertex as a "start" vertex and illustrate a running of Dijkstra's algorithm on this graph.

R-13.14 Show how to modify the pseudo-code for Dijkstra's algorithm for the case when the graph may contain parallel edges and self-loops.

R-13.15 Show how to modify the pseudo-code for Dijkstra's algorithm for the case when the graph is directed and we want to compute shortest directed paths from the source vertex to all the other vertices.

R-13.16 Show how to modify Dijkstra's algorithm to not only output the distance from v to each vertex in G, but also to output a tree T rooted at v such that the path in T from v to a vertex u is a shortest path in G from v to u.

R-13.17 There are eight small islands in a lake, and the state wants to build seven bridges to connect them so that each island can be reached from any other one via one or more bridges. The cost of constructing a bridge is proportional to its length. The distances between pairs of islands are given in the following table.

	1	2	3	4	5	6	7	8
1	-	240	210	340	280	200	345	120
2	-	-	265	175	215	180	185	155
3	-	-	-	260	115	350	435	195
4	-	-	-	-	160	330	295	230
5	-	-	-	-	-	360	400	170
6	-	-	-	-	-	-	175	205
7	-	-	-	-	-	-	-	305
8	-	-	-	-	-	-	-	-

Find which bridges to build to minimize the total construction cost.

R-13.18 Draw a simple, connected, undirected, weighted graph with 8 vertices and 16 edges, each with unique edge weights. Illustrate the execution of Kruskal's algorithm on this graph. (Note that there is only one minimum spanning tree for this graph.)

R-13.19 Repeat the previous problem for the Prim-Jarník algorithm.

R-13.20 Consider the unsorted sequence implementation of the priority queue Q used in Dijkstra's algorithm. In this case, why is this the best-case running time of Dijkstra's algorithm $O(n^2)$ on an n-vertex graph?

R-13.21 Describe the meaning of the graphical conventions used in Figure 13.6 illustrating a DFS traversal. What do the colors blue and black refer to? What do the arrows signify? How about thick lines and dashed lines?

R-13.22 Repeat Exercise R-13.21 for Figure 13.7 illustrating a BFS traversal.

R-13.23 Repeat Exercise R-13.21 for Figure 13.9 illustrating a directed DFS traversal.

R-13.24 Repeat Exercise R-13.21 for Figure 13.10 illustrating the Floyd-Warshall algorithm.

R-13.25 Repeat Exercise R-13.21 for Figure 13.12 illustrating the topological sorting algorithm.

R-13.26 Repeat Exercise R-13.21 for Figures 13.14 and 13.15 illustrating Dijkstra's algorithm.

R-13.27 Repeat Exercise R-13.21 for Figures 13.18 and 13.20 illustrating Kruskal's algorithm.

R-13.28 Repeat Exercise R-13.21 for Figures 13.21 and 13.22 illustrating the Prim-Jarník algorithm.

R-13.29 How many edges are in the transitive closure of a graph that consists of a simple directed path of n vertices?

R-13.30 Given a complete binary tree T with n nodes, consider a directed graph G having the nodes of T as its vertices. For each parent-child pair in T, create a directed edge in G from the parent to the child. Show that the transitive closure of G has $O(n \log n)$ edges.

R-13.31 A simple undirected graph is ***complete*** if it contains an edge between every pair of distinct vertices. What does a depth-first search tree of a complete graph look like?

R-13.32 Recalling the definition of a complete graph from Exercise R-13.31, what does a breadth-first search tree of a complete graph look like?

R-13.33 Say that a maze is ***constructed correctly*** if there is one path from the start to the finish, the entire maze is reachable from the start, and there are no loops around any portions of the maze. Given a maze drawn in an $n \times n$ grid, how can we determine if it is constructed correctly? What is the running time of this algorithm?

Creativity

C-13.1 Say that an n-vertex directed acyclic graph G is ***compact*** if there is some way of numbering the vertices of G with the integers from 0 to $n-1$ such that G contains the edge (i, j) if and only if $i < j$, for all i, j in $[0, n-1]$. Give an $O(n^2)$-time algorithm for detecting if G is compact.

C-13.2 Describe, in pseudo-code, an $O(n+m)$-time algorithm for computing ***all*** the connected components of an undirected graph G with n vertices and m edges.

C-13.3 Let T be the spanning tree rooted at the start vertex produced by the depth-first search of a connected, undirected graph G. Argue why every edge of G not in T goes from a vertex in T to one of its ancestors, that is, it is a ***back edge***.

C-13.4 Suppose we wish to represent an n-vertex graph G using the edge list structure, assuming that we identify the vertices with the integers in the set $\{0, 1, \ldots, n-1\}$. Describe how to implement the collection E to support $O(\log n)$-time performance for the areAdjacent function. How are you implementing the function in this case?

C-13.5 Tamarindo University and many other schools worldwide are doing a joint project on multimedia. A computer network is built to connect these schools using communication links that form a free tree. The schools decide to install a file server at one of the schools to share data among all the schools. Since the transmission time on a link is dominated by the link setup and synchronization, the cost of a data transfer is proportional to the number of links used. Hence, it is desirable to choose a "central" location for the file server. Given a free tree T and a node v of T, the *eccentricity* of v is the length of a longest path from v to any other node of T. A node of T with minimum eccentricity is called a *center* of T.

a. Design an efficient algorithm that, given an n-node free tree T, computes a center of T.

b. Is the center unique? If not, how many distinct centers can a free tree have?

C-13.6 Show that, if T is a BFS tree produced for a connected graph G, then, for each vertex v at level i, the path of T between s and v has i edges, and any other path of G between s and v has at least i edges.

C-13.7 The time delay of a long-distance call can be determined by multiplying a small fixed constant by the number of communication links on the telephone network between the caller and callee. Suppose the telephone network of a company named RT&T is a free tree. The engineers of RT&T want to compute the maximum possible time delay that may be experienced in a long-distance call. Given a free tree T, the *diameter* of T is the length of a longest path between two nodes of T. Give an efficient algorithm for computing the diameter of T.

C-13.8 A company named RT&T has a network of n switching stations connected by m high-speed communication links. Each customer's phone is directly connected to one station in his or her area. The engineers of RT&T have developed a prototype video-phone system that allows two customers to see each other during a phone call. In order to have acceptable image quality, however, the number of links used to transmit video signals between the two parties cannot exceed four. Suppose that RT&T's network is represented by a graph. Design an efficient algorithm that computes, for each station, the set of stations reachable using no more than four links.

C-13.9 Explain why there are no forward nontree edges with respect to a BFS tree constructed for a directed graph.

C-13.10 An ***Euler tour*** of a directed graph G with n vertices and m edges is a cycle that traverses each edge of G exactly once according to its direction. Such a tour always exists if G is connected and the in-degree equals the out-degree of each vertex in G. Describe an $O(n+m)$-time algorithm for finding an Euler tour of such a digraph G.

C-13.11 An independent set of an undirected graph $G = (V,E)$ is a subset I of V such that no two vertices in I are adjacent. That is, if u and v are in I, then (u,v) is not in E. A ***maximal independent set*** M is an independent set such that, if we were to add any additional vertex to M, then it would not be independent any more. Every graph has a maximal independent set. (Can you see this? This question is not part of the exercise, but it is worth thinking about.) Give an efficient algorithm that computes a maximal independent set for a graph G. What is this algorithm's running time?

C-13.12 Let G be an undirected graph G with n vertices and m edges. Describe an $O(n+m)$-time algorithm for traversing each edge of G exactly once in each direction.

C-13.13 Justify Proposition 13.14.

C-13.14 Give an example of an n-vertex simple graph G that causes Dijkstra's algorithm to run in $\Omega(n^2 \log n)$ time when its implemented with a heap.

C-13.15 Give an example of a weighted directed graph G with negative-weight edges but no negative-weight cycle, such that Dijkstra's algorithm incorrectly computes the shortest-path distances from some start vertex v.

C-13.16 Consider the following greedy strategy for finding a shortest path from vertex *start* to vertex *goal* in a given connected graph.

1. Initialize *path* to *start*.
2. Initialize *VisitedVertices* to $\{start\}$.
3. If *start*=*goal*, return *path* and exit. Otherwise, continue.
4. Find the edge (*start*,*v*) of minimum weight such that *v* is adjacent to *start* and *v* is not in *VisitedVertices*.
5. Add *v* to *path*.
6. Add *v* to *VisitedVertices*.
7. Set *start* equal to *v* and go to step 3.

Does this greedy strategy always find a shortest path from *start* to *goal*? Either explain intuitively why it works, or give a counter example.

C-13.17 Show that if all the weights in a connected weighted graph G are distinct, then there is exactly one minimum spanning tree for G.

C-13.18 Design an efficient algorithm for finding a ***longest*** directed path from a vertex s to a vertex t of an acyclic weighted digraph G. Specify the graph representation used and any auxiliary data structures used. Also, analyze the time complexity of your algorithm.

C-13.19 Consider a diagram of a telephone network, which is a graph G whose vertices represent switching centers and whose edges represent communication lines joining pairs of centers. Edges are marked by their bandwidth, and the bandwidth of a path is the bandwidth of its lowest bandwidth edge. Give an algorithm that, given a diagram and two switching centers a and b, outputs the maximum bandwidth of a path between a and b.

C-13.20 Computer networks should avoid single points of failure, that is, network nodes that can disconnect the network if they fail. We say a connected graph G is ***biconnected*** if it contains no vertex whose removal would divide G into two or more connected components. Give an $O(n+m)$-time algorithm for adding at most n edges to a connected graph G, with $n \geq 3$ vertices and $m \geq n-1$ edges, to guarantee that G is biconnected.

C-13.21 NASA wants to link n stations spread over the country using communication channels. Each pair of stations has a different bandwidth available, which is known a priori. NASA wants to select $n-1$ channels (the minimum possible) in such a way that all the stations are linked by the channels and the total bandwidth (defined as the sum of the individual bandwidths of the channels) is maximum. Give an efficient algorithm for this problem and determine its worst-case time complexity. Consider the weighted graph $G = (V,E)$, where V is the set of stations and E is the set of channels between the stations. Define the weight $w(e)$ of an edge e in E as the bandwidth of the corresponding channel.

C-13.22 Suppose you are given a *timetable*, which consists of:

- A set $\mathcal{A}$ of n airports, and for each airport a in $\mathcal{A}$, a minimum connecting time $c(a)$.
- A set $\mathcal{F}$ of m flights, and the following, for each flight f in $\mathcal{F}$:
  - Origin airport $a_1(f)$ in $\mathcal{A}$
  - Destination airport $a_2(f)$ in $\mathcal{A}$
  - Departure time $t_1(f)$
  - Arrival time $t_2(f)$

Describe an efficient algorithm for the flight scheduling problem. In this problem, we are given airports a and b, and a time t, and we wish to compute a sequence of flights that allows one to arrive at the earliest possible time in b when departing from a at or after time t. Minimum connecting times at intermediate airports should be observed. What is the running time of your algorithm as a function of n and m?

C-13.23 Inside the Castle of Asymptopia there is a maze, and along each corridor of the maze there is a bag of gold coins. The amount of gold in each bag varies. A noble knight, named Sir Paul, will be given the opportunity to walk through the maze, picking up bags of gold. He may enter the maze only through a door marked "ENTER" and exit through another door marked "EXIT." While in the maze, he may not retrace his steps. Each corridor of the maze has an arrow painted on the wall. Sir Paul may only go down the corridor in the direction of the arrow. There is no way to traverse a "loop" in the maze. Given a map of the maze, including the amount of gold in and the direction of each corridor, describe an algorithm to help Sir Paul pick up the most gold.

C-13.24 Let G be a weighted digraph with n vertices. Design a variation of Floyd-Warshall's algorithm for computing the lengths of the shortest paths from each vertex to every other vertex in $O(n^3)$ time.

C-13.25 Suppose we are given a directed graph G with n vertices, and let M be the $n \times n$ adjacency matrix corresponding to G.

a. Let the product of M with itself (M^2) be defined, for $1 \leq i, j \leq n$, as follows

$$M^2(i,j) = M(i,1) \odot M(1,j) \oplus \cdots \oplus M(i,n) \odot M(n,j),$$

where "$\oplus$" is the Boolean **or** operator and "$\odot$" is Boolean **and**. Given this definition, what does $M^2(i,j) = 1$ imply about the vertices i and j? What if $M^2(i,j) = 0$?

b. Suppose M^4 is the product of M^2 with itself. What do the entries of M^4 signify? How about the entries of $M^5 = (M^4)(M)$? In general, what information is contained in the matrix M^p?

c. Now suppose that G is weighted and assume the following:

(a) [1.] For $1 \leq i \leq n$, $M(i,i) = 0$.
(b) [2.] For $1 \leq i, j \leq n$, $M(i,j) = weight(i,j)$ if (i,j) is in E.
(c) [3.] For for $1 \leq i, j \leq n$, $M(i,j) = \infty$ if (i,j) is not in E.

Also, let M^2 be defined, for $1 \leq i, j \leq n$, as follows

$$M^2(i,j) = \min\{M(i,1) + M(1,j), \ldots, M(i,n) + M(n,j)\}.$$

If $M^2(i,j) = k$, what may we conclude about the relationship between vertices i and j?

C-13.26 A graph G is ***bipartite*** if its vertices can be partitioned into two sets X and Y such that every edge in G has one end vertex in X and the other in Y. Design and analyze an efficient algorithm for determining if an undirected graph G is bipartite (without knowing the sets X and Y in advance).

C-13.27 An old MST method, called ***Barůvka's algorithm***, works as follows on a graph G having n vertices and m edges with distinct weights.

Let T be a subgraph of G initially containing just the vertices in V.
while T has fewer than $n-1$ edges **do**
  for each connected component C_i of T **do**
    Find the lowest-weight edge (v,u) in E with v in C_i and u not in C_i.
    Add (v,u) to T (unless it is already in T).
return T

Argue why this algorithm is correct and why it runs in $O(m \log n)$ time.

C-13.28 Let G be a graph with n vertices and m edges such that all the edge weights in G are integers in the range $[1,n]$. Give an algorithm for finding a minimum spanning tree for G in $O(m \log^* n)$ time.

Projects

P-13.1 Write a class implementing a simplified graph ADT that has only functions relevant to undirected graphs and does not include update functions using the adjacency matrix structure. Your class should include a constructor that takes two collections (for example, sequences)—a collection V of vertex elements and a collection E of pairs of vertex elements—and produces the graph G that these two collections represent.

P-13.2 Implement the simplified graph ADT described in Project P-13.1 using the adjacency list structure.

P-13.3 Implement the simplified graph ADT described in Project P-13.1 using the edge list structure.

P-13.4 Extend the class of Project P-13.2 to support all the functions of the graph ADT (including functions for directed edges).

P-13.5 Implement a generic BFS traversal using the template method pattern.

P-13.6 Implement the topological sorting algorithm.

P-13.7 Implement the Floyd-Warshall transitive closure algorithm.

P-13.8 Design an experimental comparison of repeated DFS traversals versus the Floyd-Warshall algorithm for computing the transitive closure of a digraph.

P-13.9 Implement Dijkstra's algorithm assuming that the edge weights are integers.

P-13.10 Implement Kruskal's algorithm assuming that the edge weights are integers.

P-13.11 Implement the Prim-Jarník algorithm assuming that the edge weights are integers.

P-13.12 Perform an experimental comparison of two of the minimum spanning tree algorithms discussed in this chapter (Kruskal and Prim-Jarník). Develop an extensive set of experiments to test the running times of these algorithms using randomly generated graphs.

P-13.13 One way to construct a ***maze*** starts with an $n \times n$ grid such that each grid cell is bounded by four unit-length walls. We then remove two boundary unit-length walls, to represent the start and finish. For each remaining unit-length wall not on the boundary, we assign a random value and create a graph G, called the ***dual***, such that each grid cell is a vertex in G and there is an edge joining the vertices for two cells if and only if the cells share a common wall. The weight of each edge is the weight of the corresponding wall. We construct the maze by finding a minimum spanning tree T for G and removing all the walls corresponding to edges in T. Write a program that uses this algorithm to generate mazes and then

solves them. Minimally, your program should draw the maze and, ideally, it should visualize the solution as well.

P-13.14 Write a program that builds the routing tables for the nodes in a computer network, based on shortest-path routing, where path distance is measured by hop count, that is, the number of edges in a path. The input for this problem is the connectivity information for all the nodes in the network, as in the following example:

241.12.31.14: 241.12.31.15 241.12.31.18 241.12.31.19

which indicates three network nodes that are connected to 241.12.31.14, that is, three nodes that are one hop away. The routing table for the node at address A is a set of pairs (B,C), which indicates that, to route a message from A to B, the next node to send to (on the shortest path from A to B) is C. Your program should output the routing table for each node in the network, given an input list of node connectivity lists, each of which is input in the syntax as shown above, one per line.

Chapter Notes

The depth-first search method is a part of the "folklore" of computer science, but Hopcroft and Tarjan [46, 94] are the ones who showed how useful this algorithm is for solving several different graph problems. Knuth [59] discusses the topological sorting problem. The simple linear-time algorithm that we describe for determining if a directed graph is strongly connected is due to Kosaraju. The Floyd-Warshall algorithm appears in a paper by Floyd [32] and is based upon a theorem of Warshall [102]. To learn about different algorithms for drawing graphs, please see the book chapter by Tamassia and Liotta [92] and the book by Di Battista, Eades, Tamassia and Tollis [28]. The first known minimum spanning tree algorithm is due to Barůvka [8], and was published in 1926. The Prim-Jarník algorithm was first published in Czech by Jarník [50] in 1930 and in English in 1957 by Prim [85]. Kruskal published his minimum spanning tree algorithm in 1956 [62]. The reader interested in further study of the history of the minimum spanning tree problem is referred to the paper by Graham and Hell [41]. The current asymptotically fastest minimum spanning tree algorithm is a randomized algorithm of Karger, Klein, and Tarjan [52] that runs in $O(m)$ expected time.

Dijkstra [29] published his single-source, shortest-path algorithm in 1959. The reader interested in further study of graph algorithms is referred to the books by Ahuja, Magnanti, and Orlin [6], Cormen, Leiserson, and Rivest [24], Even [31], Gibbons [36], Mehlhorn [74], and Tarjan [95], and the book chapter by van Leeuwen [98]. Incidentally, the running time for the Prim-Jarník algorithm, and also that of Dijkstra's algorithm, can actually be improved to be $O(n \log n + m)$ by implementing the queue Q with either of two more sophisticated data structures, the "Fibonacci Heap" [34] or the "Relaxed Heap" [30].

This page intentionally left blank