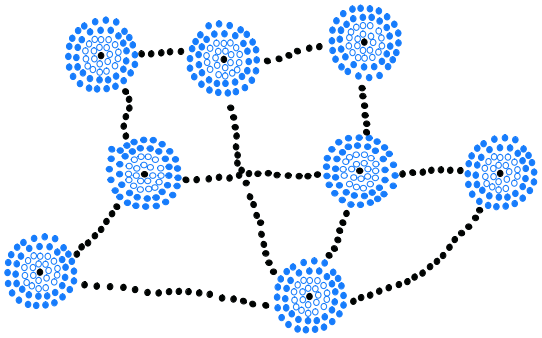
Chapter 14 Graph Algorithms



Contents

14.1 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, called edges. Graphs have applications in modeling many domains, including mapping, transportation, computer networks, and electrical engineering. By the way, 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.

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. Incidentally, 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 14.1: *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 14.1.) 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.*

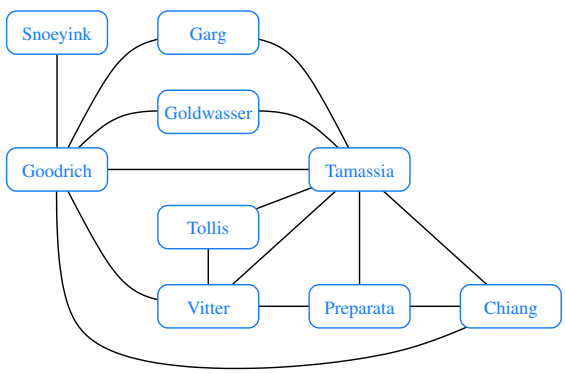


Figure 14.1: Graph of coauthorship among some authors.

Example 14.2: *We can associate with an object-oriented program 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 inherits from 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, as in the following example.

Example 14.3: *A city map can be modeled as 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 14.4: *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, for, 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*** to 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 14.5: *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 14.2.) In graph G, the edges are directed because a given flight has a specific travel direction. The endpoints of an edge e in G correspond respectively to the origin and destination of 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 to 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.*

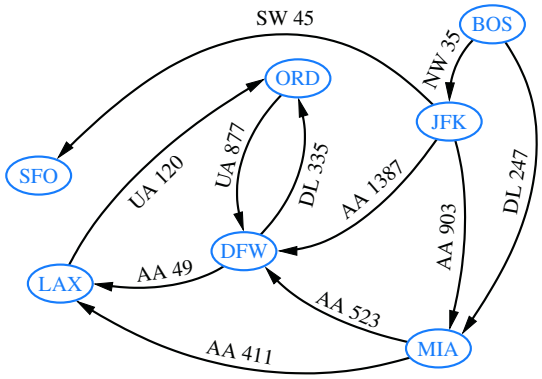


Figure 14.2: Example of a directed graph representing a flight network. The endpoints of edge UA 120 are LAX and ORD; hence, LAX and ORD are adjacent. The in-degree of DFW is 3, and the out-degree of DFW is 2.

The definition of a graph refers to the group of edges as a ***collection***, not a ***set***, thus allowing 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***. A flight network can contain parallel edges (Example 14.5), such that 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 14.3), 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 will assume that a graph is simple unless otherwise specified.

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 that starts and ends at the same vertex, and that includes at least one edge. 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 14.2, (BOS, NW 35, JFK, AA 1387, DFW) is a directed simple path, and (LAX, UA 120, ORD, UA 877, DFW, AA 49, LAX) is a directed simple cycle. Note that a directed graph may have a cycle consisting of two edges with opposite direction between the same pair of vertices, for example (ORD, UA 877, DFW, DL 335, ORD) in Figure 14.2. A directed graph is ***acyclic*** if it has no directed cycles. For example, if we were to remove the edge UA 877 from the graph in Figure 14.2, the remaining graph is acyclic. If a graph is simple, we may omit the edges when describing path P or cycle 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 14.6: *Given a graph G representing a city map (see Example 14.3), we can model a couple driving to dinner at a recommended restaurant as traversing a path though G. If they know the way, and do not 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.*

Given vertices u and v of a (directed) graph G, we say that u ***reaches*** v, and that v is ***reachable*** from u, if G has a (directed) path from u to v. In an undirected graph, the notion of ***reachability*** is symmetric, that is to say, u reaches v if an only if v reaches u. However, in a directed graph, it is possible that u reaches v but v does not reach u, because a directed path must be traversed according to the respective directions of the edges. A graph is ***connected*** if, for any two vertices, there is a path between them. A directed graph $\vec{G}$ is ***strongly connected*** if for any two vertices u and v of $\vec{G}$, u reaches v and v reaches u. (See Figure 14.3 for some examples.)

A ***subgraph*** of a graph G is a graph H whose vertices and edges are subsets of the vertices and edges of G, respectively. A ***spanning subgraph*** of G is a subgraph of G that contains all the vertices of the graph G. 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. A ***spanning tree*** of a graph is a spanning subgraph that is a tree. (Note that this definition of a tree is somewhat different from the one given in Chapter 8, as there is not necessarily a designated root.)

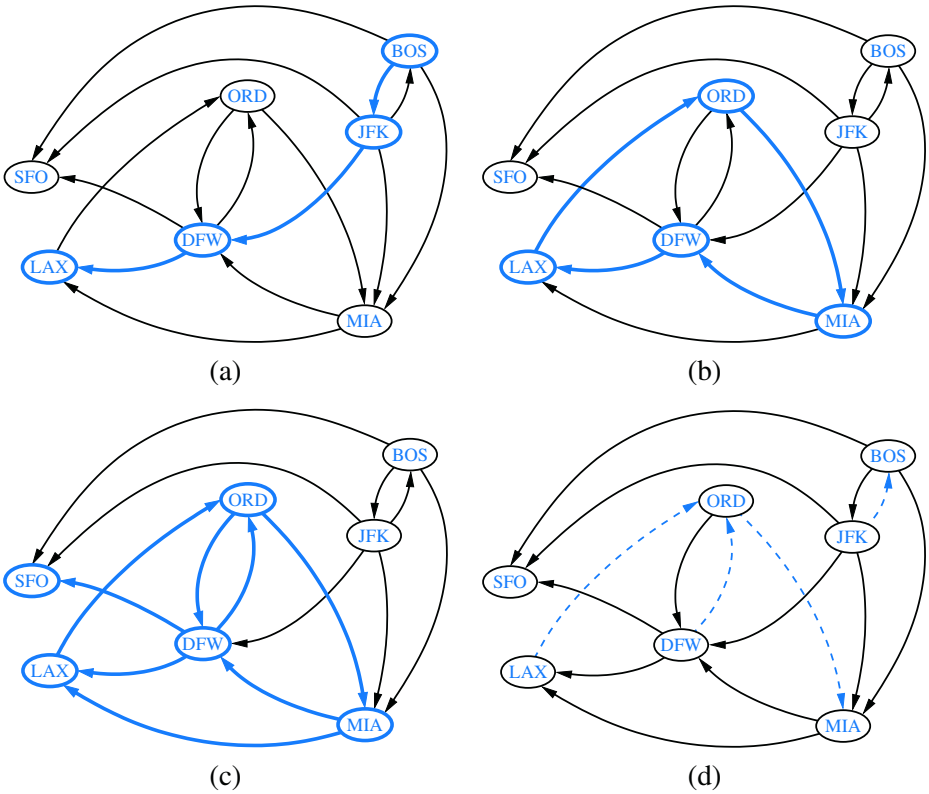


Figure 14.3: Examples of reachability in a directed graph: (a) a directed path from BOS to LAX is highlighted; (b) a directed cycle (ORD, MIA, DFW, LAX, ORD) is highlighted; its vertices induce a strongly connected subgraph; (c) the subgraph of the vertices and edges reachable from ORD is highlighted; (d) the removal of the dashed edges results in a directed acyclic graph.

Example 14.7: *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 email 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.*

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

Proposition 14.8: *If G is a graph with m edges and vertex set V, then*

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

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 14.9: *If G is a directed graph with m edges and vertex set V, then*

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

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 in-degrees. ■

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

Proposition 14.10: *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 14.8, $2m \leq n(n-1)$. 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 14.9, $m \leq n(n-1)$. ■

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

Proposition 14.11: *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 $m \leq n-1$.*

14.1.1 The Graph ADT

A graph is a collection of vertices and edges. We model the abstraction as a combination of three data types: Vertex, Edge, and Graph. A Vertex is a lightweight object that stores an arbitrary element provided by the user (e.g., an airport code); we assume the element can be retrieved with the getElement( ) method. An Edge also stores an associated object (e.g., a flight number, travel distance, cost), which is returned by its getElement( ) method.

The primary abstraction for a graph is the Graph ADT. We presume that a graph can be either ***undirected*** or ***directed***, with the designation declared upon construction; recall that a mixed graph can be represented as a directed graph, modeling edge $\{u,v\}$ as a pair of directed edges (u,v) and (v,u). The Graph ADT includes the following methods:

numVertices( ): Returns the number of vertices of the graph.

vertices( ): Returns an iteration of all the vertices of the graph.

numEdges( ): Returns the number of edges of the graph.

edges( ): Returns an iteration of all the edges of the graph.

getEdge(u, v): Returns the edge from vertex u to vertex v, if one exists; otherwise return null. For an undirected graph, there is no difference between getEdge(u, v) and getEdge(v, u).

endVertices(e): Returns an array containing the two endpoint vertices of edge e. If the graph is directed, the first vertex is the origin and the second is the destination.

opposite(v, e): For edge e incident to vertex v, returns the other vertex of the edge; an error occurs if e is not incident to v.

outDegree(v): Returns the number of outgoing edges from vertex v.

inDegree(v): Returns the number of incoming edges to vertex v. For an undirected graph, this returns the same value as does outDegree(v).

outgoingEdges(v): Returns an iteration of all outgoing edges from vertex v.

incomingEdges(v): Returns an iteration of all incoming edges to vertex v. For an undirected graph, this returns the same collection as does outgoingEdges(v).

insertVertex(x): Creates and returns a new Vertex storing element x.

insertEdge(u, v, x): Creates and returns a new Edge from vertex u to vertex v, storing element x; an error occurs if there already exists an edge from u to v.

removeVertex(v): Removes vertex v and all its incident edges from the graph.

removeEdge(e): Removes edge e from the graph.

14.2 Data Structures for Graphs

In this section, we introduce four data structures for representing a graph. In each representation, we maintain a collection to store the vertices of a graph. However, the four representations differ greatly in the way they organize the edges.

- In an ***edge list***, we maintain an unordered list of all edges. This minimally suffices, but there is no efficient way to locate a particular edge (u,v), or the set of all edges incident to a vertex v.
- In an ***adjacency list***, we additionally maintain, for each vertex, a separate list containing those edges that are incident to the vertex. This organization allows us to more efficiently find all edges incident to a given vertex.
- An ***adjacency map*** is similar to an adjacency list, but the secondary container of all edges incident to a vertex is organized as a map, rather than as a list, with the adjacent vertex serving as a key. This allows more efficient access to a specific edge (u,v), for example, in $O(1)$ expected time with hashing.
- An ***adjacency matrix*** provides worst-case $O(1)$ access to a specific edge (u,v) by maintaining an $n \times n$ matrix, for a graph with n vertices. Each slot is dedicated to storing a reference to the edge (u,v) for a particular pair of vertices u and v; if no such edge exists, the slot will store null.

A summary of the performance of these structures is given in Table 14.1.

Method	Edge List	Adj. List	Adj. Map	Adj. Matrix
numVertices( )	$O(1)$	$O(1)$	$O(1)$	$O(1)$
numEdges( )	$O(1)$	$O(1)$	$O(1)$	$O(1)$
vertices( )	$O(n)$	$O(n)$	$O(n)$	$O(n)$
edges( )	$O(m)$	$O(m)$	$O(m)$	$O(m)$
getEdge(u, v)	$O(m)$	$O(\min(d_u, d_v))$	$O(1)$ exp.	$O(1)$
outDegree(v) inDegree(v)	$O(m)$	$O(1)$	$O(1)$	$O(n)$
outgoingEdges(v) incomingEdges(v)	$O(m)$	$O(d_v)$	$O(d_v)$	$O(n)$
insertVertex(x)	$O(1)$	$O(1)$	$O(1)$	$O(n^2)$
removeVertex(v)	$O(m)$	$O(d_v)$	$O(d_v)$	$O(n^2)$
insertEdge(u, v, x)	$O(1)$	$O(1)$	$O(1)$ exp.	$O(1)$
removeEdge(e)	$O(1)$	$O(1)$	$O(1)$ exp.	$O(1)$

Table 14.1: A summary of the running times for the methods of the graph ADT, using the graph representations discussed in this section. We let n denote the number of vertices, m the number of edges, and d_v the degree of vertex v. Note that the adjacency matrix uses $O(n^2)$ space, while all other structures use $O(n+m)$ space.

14.2.1 Edge List Structure

The ***edge list*** structure is possibly the simplest, though not the most efficient, representation of a graph G. All vertex objects are stored in an unordered list V, and all edge objects are stored in an unordered list E. We illustrate an example of the edge list structure for a graph G in Figure 14.4.

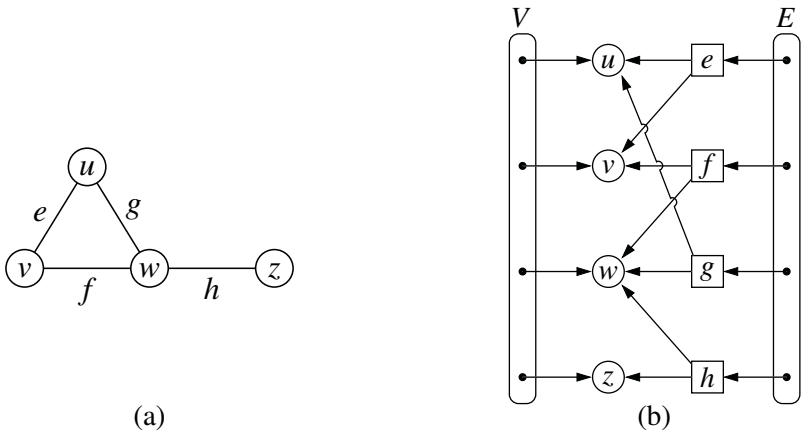


Figure 14.4: (a) A graph G; (b) schematic representation of the edge list structure for G. Notice that an edge object refers to the two vertex objects that correspond to its endpoints, but that vertices do not refer to incident edges.

To support the many methods of the Graph ADT (Section 14.1), we assume the following additional features of an edge list representation. Collections V and E are represented with doubly linked lists using our LinkedPositionalList class from Chapter 7.

Vertex Objects

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

- A reference to element x, to support the getElement( ) method.
- A reference to the position of the vertex instance in the list V, thereby allowing v to be efficiently removed from V if it were removed from the graph.

Edge Objects

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

- A reference to element x, to support the getElement( ) method.
- References to the vertex objects associated with the endpoint vertices of e. These will allow for constant-time support for methods endVertices(e) and opposite(v, e).
- A reference to the position of the edge instance in list E, thereby allowing e to be efficiently removed from E if it were removed from the graph.

Performance of the Edge List Structure

The performance of an edge list structure in fulfilling the graph ADT is summarized in Table 14.2. We begin by discussing the space usage, which is $O(n+m)$ for representing a graph with n vertices and m edges. Each individual vertex or edge instance uses $O(1)$ space, and the additional lists V and E use space proportional to their number of entries.

In terms of running time, the edge list structure does as well as one could hope in terms of reporting the number of vertices or edges, or in producing an iteration of those vertices or edges. By querying the respective list V or E, the numVertices and numEdges methods run in $O(1)$ time, and by iterating through the appropriate list, the methods vertices and edges run respectively in $O(n)$ and $O(m)$ time.

The most significant limitations of an edge list structure, especially when compared to the other graph representations, are the $O(m)$ running times of methods getEdge(u, v), outDegree(v), and outgoingEdges(v) (and corresponding methods inDegree and incomingEdges). The problem is that with all edges of the graph in an unordered list E, the only way to answer those queries is through an exhaustive inspection of all edges.

Finally, we consider the methods that update the graph. It is easy to add a new vertex or a new edge to the graph in $O(1)$ time. For example, a new edge can be added to the graph by creating an Edge instance storing the given element as data, adding that instance to the positional list E, and recording its resulting Position within E as an attribute of the edge. That stored position can later be used to locate and remove this edge from E in $O(1)$ time, and thus implement the method removeEdge(e).

It is worth discussing why the removeVertex(v) method has a running time of $O(m)$. As stated in the graph ADT, when a vertex v is removed from the graph, all edges incident to v must also be removed (otherwise, we would have a contradiction of edges that refer to vertices that are not part of the graph). To locate the incident edges to the vertex, we must examine all edges of E.

Method	Running Time
numVertices( ), numEdges( )	$O(1)$
vertices( )	$O(n)$
edges( )	$O(m)$
getEdge(u, v), outDegree(v), outgoingEdges(v)	$O(m)$
insertVertex(x), insertEdge(u, v, x), removeEdge(e)	$O(1)$
removeVertex(v)	$O(m)$

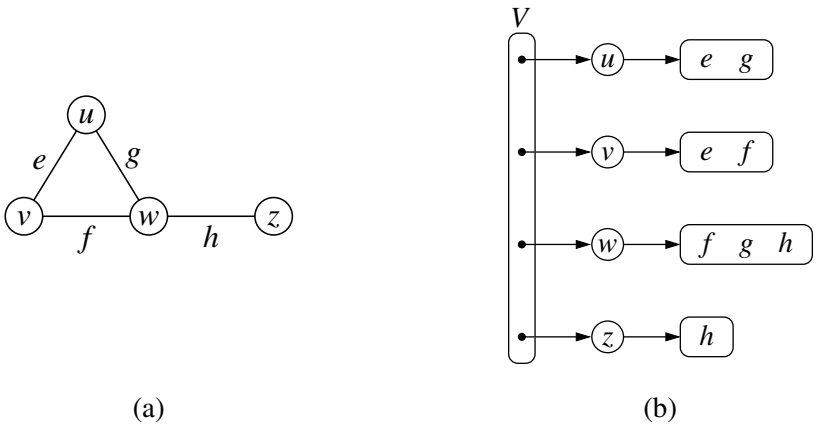
Table 14.2: Running times of the methods of a graph implemented with the edge list structure. The space used is $O(n+m)$, where n is the number of vertices and m is the number of edges.

14.2.2 Adjacency List Structure

The adjacency list structure for a graph 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. Specifically, for each vertex v, we maintain a collection $I(v)$, called the ***incidence collection*** of v, whose entries are edges incident to v. In the case of a directed graph, outgoing and incoming edges can be respectively stored in two separate collections, $I_{\text{out}}(v)$ and $I_{\text{in}}(v)$. 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.

We require that the primary structure for an adjacency list maintain the collection V of vertices in a way so that we can locate the secondary structure $I(v)$ for a given vertex v in $O(1)$ time. This could be done by using a positional list to represent V, with each Vertex instance maintaining a direct reference to its $I(v)$ incidence collection; we illustrate such an adjacency list structure of a graph in Figure 14.5. If vertices can be uniquely numbered from 0 to $n-1$, we could instead use a primary array-based structure to access the appropriate secondary lists.

The primary benefit of an adjacency list is that the collection $I(v)$ (or more specifically, $I_{\text{out}}(v)$) contains exactly those edges that should be reported by the method outgoingEdges(v). Therefore, we can implement this method by iterating the edges of $I(v)$ in $O(\deg(v))$ time, where $\deg(v)$ is the degree of vertex v. This is the best possible outcome for any graph representation, because there are $\deg(v)$ edges to be reported.



(a) (b)

Figure 14.5: (a) An undirected graph G; (b) a schematic representation of the adjacency list structure for G. Collection V is the primary list of vertices, and each vertex has an associated list of incident edges. Although not diagrammed as such, we presume that each edge of the graph is represented with a unique Edge instance that maintains references to its endpoint vertices, and that E is a list of all edges.

Performance of the Adjacency List Structure

Table 14.3 summarizes the performance of the adjacency list structure implementation of a graph, assuming that the primary collection V and E, and all secondary collections $I(v)$ are implemented with doubly linked lists.

Asymptotically, the space requirements for an adjacency list are the same as an edge list structure, using $O(n+m)$ space for a graph with n vertices and m edges. It is clear that the primary lists of vertices and edges use $O(n+m)$ space. In addition, the sum of the lengths of all secondary lists is $O(m)$, for reasons that were formalized in Propositions 14.8 and 14.9. In short, an undirected edge (u,v) is referenced in both $I(u)$ and $I(v)$, but its presence in the graph results in only a constant amount of additional space.

We have already noted that the outgoingEdges(v) method can be achieved in $O(\deg(v))$ time based on use of $I(v)$. For a directed graph, this is more specifically $O(\text{outdeg}(v))$ based on use of $I_{\text{out}}(v)$. The outDegree(v) method of the graph ADT can run in $O(1)$ time, assuming collection $I(v)$ can report its size in similar time. To locate a specific edge for implementing getEdge(u, v), we can search through either $I(u)$ and $I(v)$ (or for a directed graph, either $I_{\text{out}}(u)$ or $I_{\text{in}}(v)$). By choosing the smaller of the two, we get $O(\min(\deg(u),\deg(v)))$ running time.

The rest of the bounds in Table 14.3 can be achieved with additional care. To efficiently support deletions of edges, an edge (u,v) would need to maintain a reference to its positions within both $I(u)$ and $I(v)$, so that it could be deleted from those collections in $O(1)$ time. To remove a vertex v, we must also remove any incident edges, but at least we can locate those edges in $O(\deg(v))$ time.

Method	Running Time
numVertices( ), numEdges( )	$O(1)$
vertices( )	$O(n)$
edges( )	$O(m)$
getEdge(u, v)	$O(\min(\deg(u),\deg(v)))$
outDegree(v), inDegree(v)	$O(1)$
outgoingEdges(v), incomingEdges(v)	$O(\deg(v))$
insertVertex(x), insertEdge(u, v, x)	$O(1)$
removeEdge(e)	$O(1)$
removeVertex(v)	$O(\deg(v))$

Table 14.3: Running times of the methods 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.

14.2.3 Adjacency Map Structure

In the adjacency list structure, we assume that the secondary incidence collections are implemented as unordered linked lists. Such a collection $I(v)$ uses space proportional to $O(\deg(v))$, allows an edge to be added or removed in $O(1)$ time, and allows an iteration of all edges incident to vertex v in $O(\deg(v))$ time. However, the best implementation of getEdge(u, v) requires $O(\min(\deg(u), \deg(v)))$ time, because we must search through either $I(u)$ or $I(v)$.

We can improve the performance by using a hash-based map to implement $I(v)$ for each vertex v. Specifically, we let the opposite endpoint of each incident edge serve as a key in the map, with the edge structure serving as the value. We call such a graph representation an ***adjacency map***. (See Figure 14.6.) The space usage for an adjacency map remains $O(n+m)$, because $I(v)$ uses $O(\deg(v))$ space for each vertex v, as with the adjacency list.

The advantage of the adjacency map, relative to an adjacency list, is that the getEdge(u, v) method can be implemented in ***expected*** $O(1)$ time by searching for vertex u as a key in $I(v)$, or vice versa. This provides a likely improvement over the adjacency list, while retaining the worst-case bound of $O(\min(\deg(u), \deg(v)))$.

In comparing the performance of adjacency map to other representations (see Table 14.1), we find that it essentially achieves optimal running times for all methods, making it an excellent all-purpose choice as a graph representation.

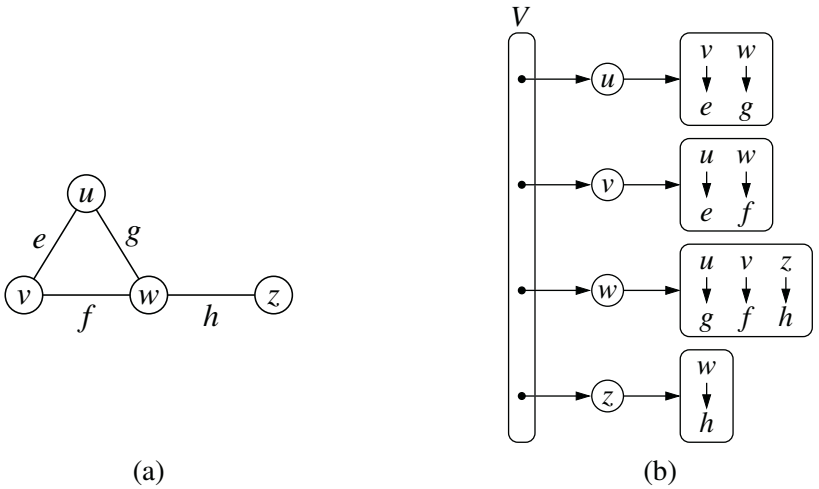


Figure 14.6: (a) An undirected graph G; (b) a schematic representation of the adjacency map structure for G. Each vertex maintains a secondary map in which neighboring vertices serve as keys, with the connecting edges as associated values. As with the adjacency list, we presume that there is also an overall list E of all Edge instances.

14.2.4 Adjacency Matrix Structure

The ***adjacency matrix*** structure for a graph G augments the edge list structure with a matrix A (that is, a two-dimensional array, as in Section 3.1.5), which allows us to locate an edge between a given pair of vertices in *worst-case* 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 cell $A[i][j]$ holds a reference to the edge (u,v), if it exists, where u is the vertex with index i and v is the vertex with index j. If there is no such edge, then $A[i][j] = \text{null}$. We note that array A is symmetric if graph G is undirected, as $A[i][j] = A[j][i]$ for all pairs i and j. (See Figure 14.7.)

The most significant advantage of an adjacency matrix is that any edge (u,v) can be accessed in worst-case $O(1)$ time; recall that the adjacency map supports that operation in $O(1)$ *expected* time. However, several operation are less efficient with an adjacency matrix. For example, to find the edges incident to vertex v, we must presumably examine all n entries in the row associated with v; recall that an adjacency list or map can locate those edges in optimal $O(\deg(v))$ time. Adding or removing vertices from a graph is problematic, as the matrix must be resized.

Furthermore, the $O(n^2)$ space usage of an adjacency matrix is typically far worse than the $O(n+m)$ space required of the other representations. Although, in the worst case, the number of edges in a ***dense*** graph will be proportional to n^2, most real-world graphs are ***sparse***. In such cases, use of an adjacency matrix is inefficient. However, if a graph is dense, the constants of proportionality of an adjacency matrix can be smaller than that of an adjacency list or map. In fact, if edges do not have auxiliary data, a boolean adjacency matrix can use one bit per edge slot, such that $A[i][j] = \text{true}$ if and only if associated (u,v) is an edge.

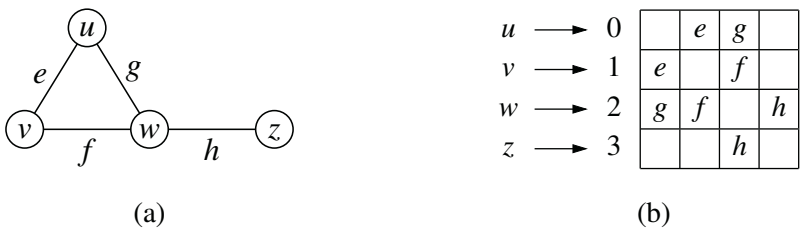


Figure 14.7: (a) An undirected graph G; (b) a schematic representation of the auxiliary adjacency matrix structure for G, in which n vertices are mapped to indices 0 to $n-1$. Although not diagrammed as such, we presume that there is a unique Edge instance for each edge, and that it maintains references to its endpoint vertices. We also assume that there is a secondary edge list (not pictured), to allow the edges( ) method to run in $O(m)$ time, for a graph with m edges.

14.2.5 Java Implementation

In this section, we provide an implementation of the Graph ADT, based on the ***adjacency map*** representation, as described in Section 14.2.3. We use positional lists to represent each of the primary lists V and E, as originally described in the edge list representation. Additionally, for each vertex v, we use a hash-based map to represent the secondary incidence map $I(v)$.

To gracefully support both undirected and directed graphs, each vertex maintains two different map references: outgoing and incoming. In the directed case, these are initialized to two distinct map instances, representing $I_{\text{out}}(v)$ and $I_{\text{in}}(v)$, respectively. In the case of an undirected graph, we assign both outgoing and incoming as aliases to a single map instance.

Our implementation is organized as follows. We assume definitions for Vertex, Edge, and Graph interfaces, although for the sake of brevity, we do not include those definitions in the book (they are available online). We then define a concrete AdjacencyMapGraph class, with nested classes InnerVertex and InnerEdge to implement the vertex and edge abstractions. These classes use generic parameters V and E to designate the element type stored respectively at vertices and edges.

We begin in Code Fragment 14.1, with the definitions of the InnerVertex and InnerEdge classes (although in reality, those definitions should be nested within the following AdjacencyMapGraph class). Note well how the InnerVertex constructor initializes the outgoing and incoming instance variables depending on whether the overall graph is undirected or directed.

Code Fragments 14.2 and 14.3 contain the core implementation of the class AdjacencyMapGraph. A graph instance maintains a boolean variable that designates whether the graph is directed, and it maintains the vertex list and edge list. Although not shown in these code fragments, our implementation includes private validate methods that perform type conversions between the public Vertex and Edge interface types to the concrete InnerVertex and InnerEdge classes, while also performing some error checking. This design is similar to the validate method of the LinkedPositionalList class (see Code Fragment 7.10 of Section 7.3.3), which converts an outward Position to the underlying Node type for that class.

The most complex methods are those that modify the graph. When insertVertex is called, we must create a new InnerVertex instance, add that vertex to the list of vertices, and record its position within that list (so that we can efficiently delete it from the list if the vertex is removed from the graph). When inserting an edge (u,v), we must also create a new instance, add it to the edge list, and record its position, yet we must also add the new edge to the outgoing adjacency map for vertex u, and the incoming map for vertex v. Code Fragment 14.3 contains code for removeVertex as well; the implementation of removeEdge is not included, but is available in the online version of the code.

```
1  /** A vertex of an adjacency map graph representation. */
2  private class InnerVertex<V> implements Vertex<V> {
3    private V element;
4    private Position<Vertex<V>> pos;
5    private Map<Vertex<V>, Edge<E>> outgoing, incoming;
6    /** Constructs a new InnerVertex instance storing the given element. */
7    public InnerVertex(V elem, boolean graphIsDirected) {
8      element = elem;
9      outgoing = new ProbeHashMap<>();
10     if (graphIsDirected)
11       incoming = new ProbeHashMap<>();
12     else
13       incoming = outgoing;                    // if undirected, alias outgoing map
14   }
15   /** Returns the element associated with the vertex. */
16   public V getElement() { return element; }
17   /** Stores the position of this vertex within the graph's vertex list. */
18   public void setPosition(Position<Vertex<V>> p) { pos = p; }
19   /** Returns the position of this vertex within the graph's vertex list. */
20   public Position<Vertex<V>> getPosition() { return pos; }
21   /** Returns reference to the underlying map of outgoing edges. */
22   public Map<Vertex<V>, Edge<E>> getOutgoing() { return outgoing; }
23   /** Returns reference to the underlying map of incoming edges. */
24   public Map<Vertex<V>, Edge<E>> getIncoming() { return incoming; }
25 } //---------- end of InnerVertex class ----------
26
27 /** An edge between two vertices. */
28 private class InnerEdge<E> implements Edge<E> {
29   private E element;
30   private Position<Edge<E>> pos;
31   private Vertex<V>[ ] endpoints;
32   /** Constructs InnerEdge instance from u to v, storing the given element. */
33   public InnerEdge(Vertex<V> u, Vertex<V> v, E elem) {
34     element = elem;
35     endpoints = (Vertex<V>[ ]) new Vertex[ ]{u,v};  // array of length 2
36   }
37   /** Returns the element associated with the edge. */
38   public E getElement() { return element; }
39   /** Returns reference to the endpoint array. */
40   public Vertex<V>[ ] getEndpoints() { return endpoints; }
41   /** Stores the position of this edge within the graph's vertex list. */
42   public void setPosition(Position<Edge<E>> p) { pos = p; }
43   /** Returns the position of this edge within the graph's vertex list. */
44   public Position<Edge<E>> getPosition() { return pos; }
45 } //---------- end of InnerEdge class ----------
```

Code Fragment 14.1: InnerVertex and InnerEdge classes (to be nested within the AdjacencyMapGraph class). Interfaces Vertex<V> and Edge<E> are not shown.

```
1  public class AdjacencyMapGraph<V,E> implements Graph<V,E> {
2    // nested InnerVertex and InnerEdge classes defined here...
3    private boolean isDirected;
4    private PositionalList<Vertex<V>> vertices = new LinkedPositionalList<>();
5    private PositionalList<Edge<E>> edges = new LinkedPositionalList<>();
6    /** Constructs an empty graph (either undirected or directed). */
7    public AdjacencyMapGraph(boolean directed) { isDirected = directed; }
8    /** Returns the number of vertices of the graph */
9    public int numVertices( ) { return vertices.size( ); }
10   /** Returns the vertices of the graph as an iterable collection */
11   public Iterable<Vertex<V>> vertices( ) { return vertices; }
12   /** Returns the number of edges of the graph */
13   public int numEdges( ) { return edges.size( ); }
14   /** Returns the edges of the graph as an iterable collection */
15   public Iterable<Edge<E>> edges( ) { return edges; }
16   /** Returns the number of edges for which vertex v is the origin. */
17   public int outDegree(Vertex<V> v) {
18     InnerVertex<V> vert = validate(v);
19     return vert.getOutgoing( ).size( );
20   }
21   /** Returns an iterable collection of edges for which vertex v is the origin. */
22   public Iterable<Edge<E>> outgoingEdges(Vertex<V> v) {
23     InnerVertex<V> vert = validate(v);
24     return vert.getOutgoing( ).values( );  // edges are the values in the adjacency map
25   }
26   /** Returns the number of edges for which vertex v is the destination. */
27   public int inDegree(Vertex<V> v) {
28     InnerVertex<V> vert = validate(v);
29     return vert.getIncoming( ).size( );
30   }
31   /** Returns an iterable collection of edges for which vertex v is the destination. */
32   public Iterable<Edge<E>> incomingEdges(Vertex<V> v) {
33     InnerVertex<V> vert = validate(v);
34     return vert.getIncoming( ).values( );  // edges are the values in the adjacency map
35   }
36   public Edge<E> getEdge(Vertex<V> u, Vertex<V> v) {
37   /** Returns the edge from u to v, or null if they are not adjacent. */
38     InnerVertex<V> origin = validate(u);
39     return origin.getOutgoing( ).get(v);  // will be null if no edge from u to v
40   }
41   /** Returns the vertices of edge e as an array of length two. */
42   public Vertex<V>[ ] endVertices(Edge<E> e) {
43     InnerEdge<E> edge = validate(e);
44     return edge.getEndpoints( );
45   }
```

Code Fragment 14.2: AdjacencyMapGraph class definition. (Continues in Code Fragment 14.3.) The validate(v) and validate(e) methods are available online.

```
46    /** Returns the vertex that is opposite vertex v on edge e. */
47    public Vertex<V> opposite(Vertex<V> v, Edge<E> e)
48                                                throws IllegalArgumentException {
49      InnerEdge<E> edge = validate(e);
50      Vertex<V>[ ] endpoints = edge.getEndpoints( );
51      if (endpoints[0] == v)
52        return endpoints[1];
53      else if (endpoints[1] == v)
54        return endpoints[0];
55      else
56        throw new IllegalArgumentException("v is not incident to this edge");
57    }
58    /** Inserts and returns a new vertex with the given element. */
59    public Vertex<V> insertVertex(V element) {
60      InnerVertex<V> v = new InnerVertex<>(element, isDirected);
61      v.setPosition(vertices.addLast(v));
62      return v;
63    }
64    /** Inserts and returns a new edge between u and v, storing given element. */
65    public Edge<E> insertEdge(Vertex<V> u, Vertex<V> v, E element)
66                                                throws IllegalArgumentException {
67      if (getEdge(u,v) == null) {
68        InnerEdge<E> e = new InnerEdge<>(u, v, element);
69        e.setPosition(edges.addLast(e));
70        InnerVertex<V> origin = validate(u);
71        InnerVertex<V> dest = validate(v);
72        origin.getOutgoing( ).put(v, e);
73        dest.getIncoming( ).put(u, e);
74        return e;
75      } else
76        throw new IllegalArgumentException("Edge from u to v exists");
77    }
78    /** Removes a vertex and all its incident edges from the graph. */
79    public void removeVertex(Vertex<V> v) {
80      InnerVertex<V> vert = validate(v);
81      // remove all incident edges from the graph
82      for (Edge<E> e : vert.getOutgoing( ).values( ))
83        removeEdge(e);
84      for (Edge<E> e : vert.getIncoming( ).values( ))
85        removeEdge(e);
86      // remove this vertex from the list of vertices
87      vertices.remove(vert.getPosition( ));
88    }
89  }
```

Code Fragment 14.3: AdjacencyMapGraph class definition (continued from Code Fragment 14.2). We omit the removeEdge method, for brevity.

14.3 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, for, 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. A traversal is efficient if it visits all the vertices and edges in time proportional to their number, that is, in linear time.

Graph traversal algorithms are key to answering many fundamental questions about graphs involving the notion of ***reachability***, that is, in determining how to travel from one vertex to another while following paths of a graph. Interesting problems that deal with reachability in an undirected graph G include the following:

- Computing a path from vertex u to vertex v, or reporting that no such path exists.
- 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.
- Testing whether G is connected.
- Computing a spanning tree of G, if G is connected.
- Computing the connected components of G.
- Identifying a cycle in G, or reporting that G has no cycles.

Interesting problems that deal with reachability in a directed graph $\vec{G}$ include the following:

- Computing a directed path from vertex u to vertex v, or reporting that no such path exists.
- Finding all the vertices of $\vec{G}$ that are reachable from a given vertex s.
- Determine whether $\vec{G}$ is acyclic.
- Determine whether $\vec{G}$ is strongly connected.

In the remainder of this section, we will present two efficient graph traversal algorithms, called ***depth-first search*** and ***breadth-first search***, respectively.

14.3.1 Depth-First Search

The first traversal algorithm we consider in this section is ***depth-first search*** (DFS). 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 a 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. In general, if we call our current vertex u, we traverse G by considering an arbitrary edge (u,v) incident to the current vertex u. If the edge (u,v) leads us to a vertex v that is already visited (that is, painted), we ignore that edge. 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 will get to a "dead end," that is, a current vertex v such that all the edges incident to v lead to vertices already visited. To get out of this impasse, we roll our string back up, backtracking along the edge that brought us to v, going back to a previously visited vertex u. We then make u our current vertex and repeat the computation above for any edges incident to u that we have not yet considered. If all of u'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 u, 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 to s.

The pseudocode for a depth-first search traversal starting at a vertex u (see Code Fragment 14.4) 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 whether a vertex or edge has been previously explored.

Algorithm DFS(G, u):
Input: A graph G and a vertex u of G
Output: A collection of vertices reachable from u, with their discovery edges
Mark vertex u as visited.
for each of u's outgoing edges, $e = (u,v)$ **do**
  if vertex v has not been visited **then**
    Record edge e as the discovery edge for vertex v.
    Recursively call DFS(G, v).

Code Fragment 14.4: The DFS algorithm.

Classifying Graph Edges with DFS

An execution of depth-first search can be used to analyze the structure of a graph, based upon the way in which edges are explored during the traversal. The DFS process naturally identifies what is known as the ***depth-first search tree*** rooted at a starting vertex s. Whenever an edge $e = (u,v)$ is used to discover a new vertex v during the DFS algorithm of Code Fragment 14.4, that edge is known as a ***discovery edge*** or ***tree edge***, as oriented from u to v. All other edges that are considered during the execution of DFS are known as ***nontree edges***, which take us to a previously visited vertex. In the case of an undirected graph, we will find that all nontree edges that are explored connect the current vertex to one that is an ancestor of it in the DFS tree. We will call such an edge a ***back edge***. When performing a DFS on a directed graph, there are three possible 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 descendant in the DFS tree
- ***cross edges***, which connect a vertex to a vertex that is neither its ancestor nor its descendant

An example application of the DFS algorithm on a directed graph is shown in Figure 14.8, demonstrating each type of nontree edge. An example application of the DFS algorithm on an undirected graph is shown in Figure 14.9.

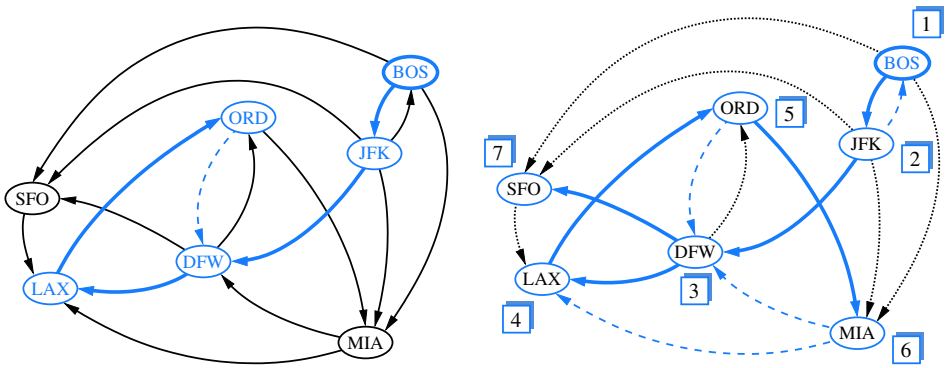


Figure 14.8: An example of a DFS in a directed graph, starting at vertex (BOS): (a) intermediate step, where, for the first time, a considered edge leads to an already visited vertex (DFW); (b) the completed DFS. The tree edges are shown with thick blue lines, the back edges are shown with dashed blue lines, and the forward and cross edges are shown with dotted 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.

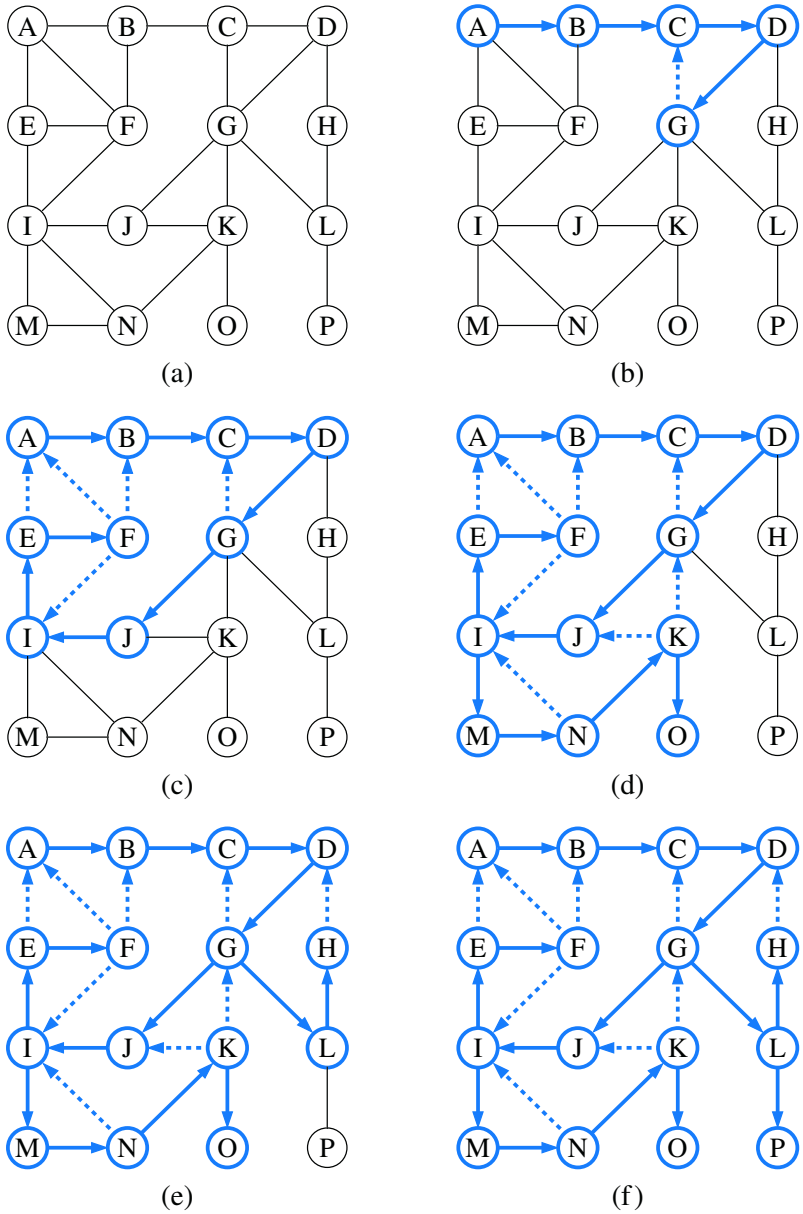


Figure 14.9: Example of depth-first search traversal on an undirected graph starting at vertex *A*. We assume that a vertex's adjacencies are considered in alphabetical order. Visited vertices and explored edges are highlighted, with discovery edges drawn as solid lines and nontree (back) edges as dashed lines: (a) input graph; (b) path of tree edges, traced from A until back edge (G,C) is examined; (c) reaching F, which is a dead end; (d) after backtracking to I, resuming with edge (I,M), and hitting another dead end at O; (e) after backtracking to G, continuing with edge (G,L), and hitting another dead end at H; (f) final result.

Properties of a Depth-First Search

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 a graph G into groups. We will begin with the most significant property.

Proposition 14.12: *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 w in s's connected component not visited, and let v be the first unvisited vertex on some path from s to w (we may have $v = w$). Since v 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,v); hence, it cannot be correct that v is unvisited. Therefore, there are no unvisited vertices in s's connected component.

Since we only follow a discovery edge when we go to an unvisited vertex, we will never form a cycle with such edges. Therefore, the discovery edges form a connected subgraph without cycles, hence 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. ■

Proposition 14.13: *Let $\vec{G}$ be a directed graph. Depth-first search on $\vec{G}$ starting at a vertex s visits all the vertices of $\vec{G}$ 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 $\vec{G}$ 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.

We prove the second fact by induction on the steps of the algorithm. We claim that each time a discovery edge (u,v) is identified, there exists a directed path from s to v in the DFS tree. Since u must have previously been discovered, there exists a path from s to u, so by appending the edge (u,v) to that path, we have a directed path from s to v. ■

Note that 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).

Running Time of Depth-First Search

In terms of its running time, depth-first search is an efficient method for traversing a graph. Note that DFS is called at most once on each vertex (since it gets marked as visited), and therefore every edge is examined at most twice for an undirected graph, once from each of its end vertices, and at most once in a directed graph, from its origin vertex. If we let $n_s \leq n$ be the number of vertices reachable from a vertex s, and $m_s \leq m$ be the number of incident edges to those vertices, 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 outgoingEdges(v) takes $O(\deg(v))$ time, and the opposite(v, e) method 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 14.14: *Let G be an undirected graph with n vertices and m edges. 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:*

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

Proposition 14.15: *Let $\vec{G}$ be a directed graph with n vertices and m edges. A DFS traversal of $\vec{G}$ can be performed in $O(n+m)$ time, and can be used to solve the following problems in $O(n+m)$ time:*

- *Computing a directed path between two given vertices of $\vec{G}$, if one exists.*
- *Computing the set of vertices of $\vec{G}$ that are reachable from a given vertex s.*
- *Testing whether $\vec{G}$ is strongly connected.*
- *Computing a directed cycle in $\vec{G}$, or reporting that $\vec{G}$ is acyclic.*

The justification of Propositions 14.14 and 14.15 is based on algorithms that use slightly modified versions of the DFS algorithm as subroutines. We will explore some of those extensions in the remainder of this section.

14.3.2 DFS Implementation and Extensions

We will begin by providing a Java implementation of the depth-first search algorithm. We originally described the algorithm with pseudocode in Code Fragment 14.4. In order to implement it, we must have a mechanism for keeping track of which vertices have been visited, and for recording the resulting DFS tree edges. For this bookkeeping, we use two auxiliary data structures. First, we maintain a set, named known, containing vertices that have already been visited. Second, we keep a map, named forest, that associates, with a vertex v, the edge e of the graph that is used to discover v (if any). Our DFS method is presented in Code Fragment 14.5.

```
1  /** Performs depth-first search of Graph g starting at Vertex u. */
2  public static <V,E> void DFS(Graph<V,E> g, Vertex<V> u,
3                      Set<Vertex<V>> known, Map<Vertex<V>,Edge<E>> forest) {
4    known.add(u);                              // u has been discovered
5    for (Edge<E> e : g.outgoingEdges(u)) {     // for every outgoing edge from u
6      Vertex<V> v = g.opposite(u, e);
7      if (!known.contains(v)) {
8        forest.put(v, e);                      // e is the tree edge that discovered v
9        DFS(g, v, known, forest);              // recursively explore from v
10     }
11   }
12 }
```

Code Fragment 14.5: Recursive implementation of depth-first search on a graph, starting at a designated vertex u. As an outcome of a call, visited vertices are added to the known set, and discovery edges are added to the forest.

Our DFS method does not make any assumption about how the Set or Map instances are implemented; however, the $O(n+m)$ running-time analysis of the previous section does presume that we can "mark" a vertex as explored or test the status of a vertex in $O(1)$ time. If we use hash-based implementations of the set and map structure, then all of their operations run in $O(1)$ *expected* time, and the overall algorithm runs in $O(n+m)$ time with very high probability. In practice, this is a compromise we are willing to accept.

If vertices can be numbered from $0,\ldots,n-1$ (a common assumption for graph algorithms), then the set and map can be implemented more directly as a lookup table, with a vertex label used as an index into an array of size n. In that case, the necessary set and map operations run in worst-case $O(1)$ time. Alternatively, we can "decorate" each vertex with the auxiliary information, either by leveraging the generic type of the element that is stored with each vertex, or by redesigning the Vertex type to store additional fields. That would allow marking operations to be performed in $O(1)$-time, without any assumption about vertices being numbered.

Reconstructing a Path from u to v

We can use the basic DFS method as a tool to identify the (directed) path leading from vertex u to v, if v is reachable from u. This path can easily be reconstructed from the information that was recorded in the forest of discovery edges during the traversal. Code Fragment 14.6 provides an implementation of a secondary method that produces an ordered list of vertices on the path from u to v, if given the map of discovery edges that was computed by the original DFS method.

To reconstruct the path, we begin at the *end* of the path, examining the forest of discovery edges to determine what edge was used to reach vertex v. We then determine the opposite vertex of that edge and repeat the process to determine what edge was used to discover it. By continuing this process until reaching u, we can construct the entire path. Assuming constant-time lookup in the forest map, the path reconstruction takes time proportional to the length of the path, and therefore, it runs in $O(n)$ time (in addition to the time originally spent calling DFS).

```
1  /** Returns an ordered list of edges comprising the directed path from u to v. */
2  public static <V,E> PositionalList<Edge<E>>
3  constructPath(Graph<V,E> g, Vertex<V> u, Vertex<V> v,
4                Map<Vertex<V>,Edge<E>> forest) {
5    PositionalList<Edge<E>> path = new LinkedPositionalList<>();
6    if (forest.get(v) != null) {          // v was discovered during the search
7      Vertex<V> walk = v;                 // we construct the path from back to front
8      while (walk != u) {
9        Edge<E> edge = forest.get(walk);
10       path.addFirst(edge);              // add edge to *front* of path
11       walk = g.opposite(walk, edge);    // repeat with opposite endpoint
12     }
13   }
14   return path;
15 }
```

Code Fragment 14.6: Method to reconstruct a directed path from u to v, given the trace of discovery from a DFS started at u. The method returns an ordered list of vertices on the path.

Testing for Connectivity

We can use the basic DFS method to determine whether a graph is connected. In the case of an undirected graph, we simply start a depth-first search at an arbitrary vertex and then test whether known.size( ) equals n at the conclusion. If the graph is connected, then by Proposition 14.12, all vertices will have been discovered; conversely, if the graph is not connected, there must be at least one vertex v that is not reachable from u, and that will not be discovered.

For directed graph, $\vec{G}$, we may wish to test whether it is ***strongly connected***, that is, whether for every pair of vertices u and v, both u reaches v and v reaches u. If we start an independent call to DFS from each vertex, we could determine whether this was the case, but those n calls when combined would run in $O(n(n+m))$. However, we can determine if $\vec{G}$ is strongly connected much faster than this, requiring only two depth-first searches.

We begin by performing a depth-first search of our directed graph $\vec{G}$ starting at an arbitrary vertex s. If there is any vertex of $\vec{G}$ that is not visited by this traversal, and is not reachable from s, then the graph is not strongly connected. If this first depth-first search visits each vertex of $\vec{G}$, we need to then check whether s is reachable from all other vertices. Conceptually, we can accomplish this by making a copy of graph $\vec{G}$, but with the orientation of all edges reversed. A depth-first search starting at s in the reversed graph will reach every vertex that could reach s in the original. In practice, a better approach than making a new graph is to reimplement a version of the DFS method that loops through all ***incoming*** edges to the current vertex, rather than all ***outgoing*** edges. Since this algorithm makes just two DFS traversals of $\vec{G}$, it runs in $O(n+m)$ time.

Computing All Connected Components

When a graph is not connected, the next goal we may have is to identify all of the ***connected components*** of an undirected graph, or the ***strongly connected components*** of a directed graph. We will begin by discussing the undirected case.

If an initial call to DFS fails to reach all vertices of a graph, we can restart a new call to DFS at one of those unvisited vertices. An implementation of such a comprehensive DFSComplete method is given in Code Fragment 14.7. It returns a map that represents a ***DFS forest*** for the entire graph. We say this is a forest rather than a tree, because the graph may not be connected.

Vertices that serve as roots of DFS trees within this forest will not have discovery edges and will not appear as keys in the returned map. Therefore, the number of connected components of the graph g is equal to g.numVertices( ) − forest.size( ).

```
1  /** Performs DFS for the entire graph and returns the DFS forest as a map. */
2  public static <V,E> Map<Vertex<V>,Edge<E>> DFSComplete(Graph<V,E> g) {
3    Set<Vertex<V>> known = new HashSet<>();
4    Map<Vertex<V>,Edge<E>> forest = new ProbeHashMap<>();
5    for (Vertex<V> u : g.vertices())
6      if (!known.contains(u))
7        DFS(g, u, known, forest);              // (re)start the DFS process at u
8    return forest;
9  }
```

Code Fragment 14.7: Top-level method that returns a DFS forest for an entire graph.

We can further determine which vertices are in which component, either by examining the structure of the forest that is returned, or by making a minor modification to the core DFS method to tag each vertex with a component number when it is first discovered. (See Exercise C-14.43.)

Although the DFSComplete method makes multiple calls to the original DFS method, the total time spent by a call to DFSComplete is $O(n+m)$. For an undirected graph, recall from our original analysis on page 635 that a single call to DFS starting at vertex s runs in time $O(n_s+m_s)$ where n_s is the number of vertices reachable from s, and m_s is the number of incident edges to those vertices. Because each call to DFS explores a different component, the sum of n_s+m_s terms is $n+m$.

The situation is more complex for finding strongly connected components of a directed graph. The $O(n+m)$ total bound for a call to DFSComplete applies to the directed case as well, because when restarting the process, we proceed with the existing set of known vertices. This ensures that the DFS subroutine is called once on each vertex, and therefore that each outgoing edge is explored only once during the entire process.

As an example, consider again the graph of Figure 14.8. If we were to start the original DFS method at vertex ORD, the known set of vertices would become { ORD, DFW, SFO, LAX, MIA }. If restarting the DFS method at vertex BOS, the outgoing edges to vertices SFO and MIA would not result in further recursion, because those vertices are marked as known.

However, the forest returned by a single call to DFSComplete does not represent the strongly connected components of the graph. There exists an approach for computing those components in $O(n+m)$ time, making use of two calls to DFSComplete, but the details are beyond the scope of this book.

Detecting Cycles with DFS

For both undirected and directed graphs, a cycle exists if and only if a ***back edge*** exists relative to the DFS traversal of that graph. It is easy to see that if a back edge exists, a cycle exists by taking the back edge from the descendant to its ancestor and then following the tree edges back to the descendant. Conversely, if a cycle exists in the graph, there must be a back edge relative to a DFS (although we do not prove this fact here).

Algorithmically, detecting a back edge in the undirected case is easy, because all edges are either tree edges or back edges. In the case of a directed graph, additional modifications to the core DFS implementation are needed to properly categorize a nontree edge as a back edge. When a directed edge is explored leading to a previously visited vertex, we must recognize whether that vertex is an ancestor of the current vertex. This can be accomplished, for example, by maintaining another set, with all vertices upon which a recursive call to DFS is currently active. We leave details as an exercise (C-14.42).

14.3.3 Breadth-First Search

The advancing and backtracking of a depth-first search, as described in the previous section, defines a traversal that could be physically traced by a single person exploring a graph. In this section, we will consider another algorithm for traversing a connected component of a graph, known as a ***breadth-first search*** (BFS). The BFS algorithm is more akin to sending out, in all directions, many explorers who collectively traverse a graph in coordinated fashion.

A BFS proceeds in rounds and subdivides the vertices into ***levels***. BFS starts at vertex s, which is at level 0. In the first round, we paint as "visited," all vertices adjacent to the start vertex s; these vertices are one step away from the beginning and are placed into level 1. In the second round, we allow all explorers to go two steps (i.e., edges) away from the starting vertex. These new vertices, which are adjacent to level 1 vertices and not previously assigned to a level, are placed into level 2 and marked as "visited." This process continues in similar fashion, terminating when no new vertices are found in a level.

A Java implementation of BFS is given in Code Fragment 14.8. We follow a convention similar to that of DFS (Code Fragment 14.5), maintaining a known set of vertices, and storing the BFS tree edges in a map. We illustrate a BFS traversal in Figure 14.10.

```
1  /** Performs breadth-first search of Graph g starting at Vertex u. */
2  public static <V,E> void BFS(Graph<V,E> g, Vertex<V> s,
3                               Set<Vertex<V>> known, Map<Vertex<V>,Edge<E>> forest) {
4    PositionalList<Vertex<V>> level = new LinkedPositionalList<>();
5    known.add(s);
6    level.addLast(s);                          // first level includes only s
7    while (!level.isEmpty()) {
8      PositionalList<Vertex<V>> nextLevel = new LinkedPositionalList<>();
9      for (Vertex<V> u : level)
10       for (Edge<E> e : g.outgoingEdges(u)) {
11         Vertex<V> v = g.opposite(u, e);
12         if (!known.contains(v)) {
13           known.add(v);
14           forest.put(v, e);                    // e is the tree edge that discovered v
15           nextLevel.addLast(v);                // v will be further considered in next pass
16         }
17       }
18     level = nextLevel;                         // relabel 'next' level to become the current
19   }
20 }
```

Code Fragment 14.8: Implementation of breadth-first search on a graph, starting at a designated vertex s.

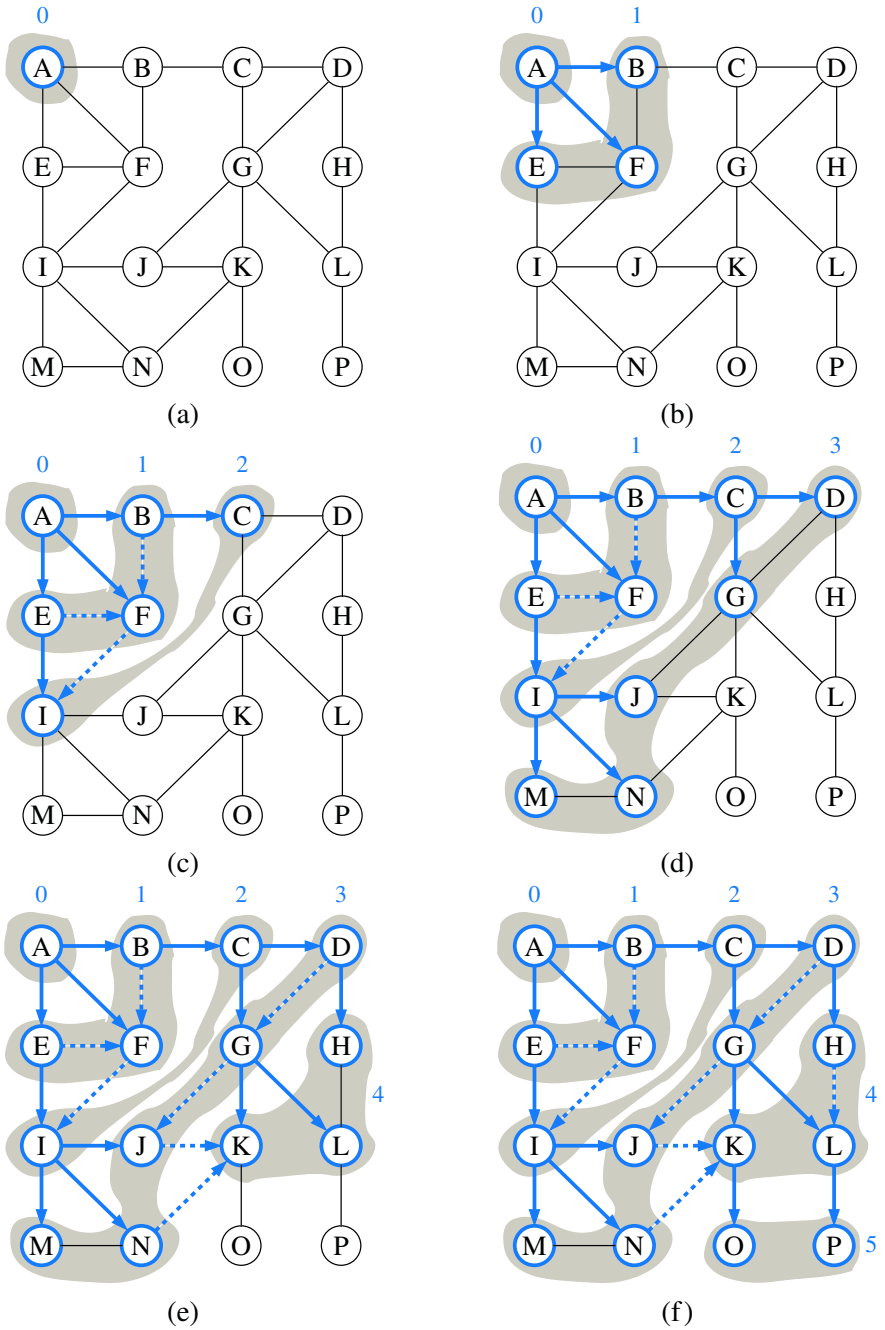


Figure 14.10: Example of breadth-first search traversal, where the edges incident to a vertex are considered in alphabetical order of the adjacent vertices. The discovery edges are shown with solid lines and the nontree (cross) edges are shown with dashed lines: (a) starting the search at A; (b) discovery of level 1; (c) discovery of level 2; (d) discovery of level 3; (e) discovery of level 4; (f) discovery of level 5.

When discussing DFS, we described a classification of nontree edges being either ***back edges***, which connect a vertex to one of its ancestors, ***forward edges***, which connect a vertex to one of its descendants, or ***cross edges***, which connect a vertex to another vertex that is neither its ancestor nor its descendant. For BFS on an undirected graph, all nontree edges are cross edges (see Exercise C-14.46), and for BFS on a directed graph, all nontree edges are either back edges or cross edges (see Exercise C-14.47).

The BFS traversal algorithm has a number of interesting properties, some of which we explore in the proposition that follows. Most notably, a path in a breadth-first search tree rooted at vertex s to any other vertex v is guaranteed to be the shortest such path from s to v in terms of the number of edges.

Proposition 14.16: *Let G be an undirected or directed graph on which a BFS traversal starting at vertex s has been performed. Then*

- *The traversal visits all vertices of G that are reachable from 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 from s to v has at least i edges.*
- *If (u,v) is an edge that is not in the BFS tree, then the level number of v can be at most 1 greater than the level number of u.*

We leave the justification of this proposition as Exercise C-14.49.

The analysis of the running time of BFS is similar to the one of DFS, with the algorithm running in $O(n+m)$ time, or more specifically, in $O(n_s+m_s)$ time if n_s is the number of vertices reachable from vertex s, and $m_s \leq m$ is the number of incident edges to those vertices. To explore the entire graph, the process can be restarted at another vertex, akin to the DFSComplete method of Code Fragment 14.7. The actual path from vertex s to vertex v can be reconstructed using the constructPath method of Code Fragment 14.6

Proposition 14.17: *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.*

Although our implementation of BFS in Code Fragment 14.8 progresses level by level, the BFS algorithm can also be implemented using a single FIFO queue to represent the current fringe of the search. Starting with the source vertex in the queue, we repeatedly remove the vertex from the front of the queue and insert any of its unvisited neighbors to the back of the queue. (See Exercise C-14.50.)

In comparing the capabilities of DFS and BFS, both can be used to efficiently find the set of vertices that are reachable from a given source, and to determine paths to those vertices. However, BFS guarantees that those paths use as few edges as possible. For an undirected graph, both algorithms can be used to test connectivity, to identify connected components, or to locate a cycle. For directed graphs, the DFS algorithm is better suited for certain tasks, such as finding a directed cycle in the graph, or in identifying the strongly connected components.

14.4 Transitive Closure

We have seen that graph traversals can be used to answer basic questions of reachability in a directed graph. In particular, if we are interested in knowing whether there is a path from vertex u to vertex v in a graph, we can perform a DFS or BFS traversal starting at u and observe whether v is discovered. If representing a graph with an adjacency list or adjacency map, we can answer the question of reachability for u and v in $O(n+m)$ time (see Propositions 14.15 and 14.17).

In certain applications, we may wish to answer many reachability queries more efficiently, in which case it may be worthwhile to precompute a more convenient representation of a graph. For example, the first step for a service that computes driving directions from an origin to a destination might be to assess whether the destination is reachable. Similarly, in an electricity network, we may wish to be able to quickly determine whether current flows from one particular vertex to another. Motivated by such applications, we introduce the following definition. The ***transitive closure*** of a directed graph $\vec{G}$ is itself a directed graph $\vec{G}^*$ such that the vertices of $\vec{G}^*$ are the same as the vertices of $\vec{G}$, and $\vec{G}^*$ has an edge (u,v), whenever $\vec{G}$ has a directed path from u to v (including the case where (u,v) is an edge of the original $\vec{G}$).

If a graph is represented as an adjacency list or adjacency map, we can compute its transitive closure in $O(n(n+m))$ time by making use of n graph traversals, one from each starting vertex. For example, a DFS starting at vertex u can be used to determine all vertices reachable from u, and thus a collection of edges originating with u in the transitive closure.

In the remainder of this section, we explore an alternative technique for computing the transitive closure of a directed graph that is particularly well suited for when a directed graph is represented by a data structure that supports $O(1)$-time lookup for the getEdge(u, v) method (for example, the adjacency-matrix structure). Let $\vec{G}$ be a directed graph with n vertices and m edges. We compute the transitive closure of $\vec{G}$ in a series of rounds. We initialize $\vec{G}_0 = \vec{G}$. We also arbitrarily number the vertices of $\vec{G}$ 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 directed graph $\vec{G}_k$ starting with $\vec{G}_k = \vec{G}_{k-1}$ and adding to $\vec{G}_k$ the directed edge (v_i, v_j) if directed graph $\vec{G}_{k-1}$ contains both the edges (v_i, v_k) and (v_k, v_j). In this way, we will enforce a simple rule embodied in the proposition that follows.

Proposition 14.18: *For $i = 1, \ldots, n$, directed graph $\vec{G}_k$ has an edge (v_i, v_j) if and only if directed graph $\vec{G}$ 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, $\vec{G}_n$ is equal to $\vec{G}^*$, the transitive closure of $\vec{G}$.*

Proposition 14.18 suggests a simple algorithm for computing the transitive closure of $\vec{G}$ that is based on the series of rounds to compute each $\vec{G}_k$. This algorithm is known as the ***Floyd-Warshall algorithm***, and its pseudocode is given in Code Fragment 14.9. We illustrate an example run of the Floyd-Warshall algorithm in Figure 14.11.

Algorithm FloydWarshall($\vec{G}$):
Input: A directed graph $\vec{G}$ with n vertices
Output: The transitive closure $\vec{G}^*$ of $\vec{G}$
let $v_1, v_2, \ldots, v_n$ be an arbitrary numbering of the vertices of $\vec{G}$
$\vec{G}_0 = \vec{G}$
for $k = 1$ to n **do**
  $\vec{G}_k = \vec{G}_{k-1}$
  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 $\vec{G}_{k-1}$ **then**
      add edge (v_i, v_j) to $\vec{G}_k$ (if it is not already present)
return $\vec{G}_n$

Code Fragment 14.9: Pseudocode for the Floyd-Warshall algorithm. This algorithm computes the transitive closure $\vec{G}^*$ of G by incrementally computing a series of directed graphs $\vec{G}_0, \vec{G}_1, \ldots, \vec{G}_n$, for $k = 1, \ldots, n$.

From this pseudocode, we can easily analyze the running time of the Floyd-Warshall algorithm assuming that the data structure representing G supports methods getEdge and insertEdge 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 $O(n^3)$. From the description and analysis above we may immediately derive the following proposition.

Proposition 14.19: *Let $\vec{G}$ be a directed graph with n vertices, and let $\vec{G}$ 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 $\vec{G}^*$ of $\vec{G}$ in $O(n^3)$ time.*

Performance of the Floyd-Warshall Algorithm

Asymptotically, the $O(n^3)$ running time of the Floyd-Warshall algorithm is no better than that achieved by repeatedly running DFS, once from each vertex, to compute the reachability. However, the Floyd-Warshall algorithm matches the asymptotic bounds of the repeated DFS when a graph is dense, or when a graph is sparse but represented as an adjacency matrix. (See Exercise R-14.13.)

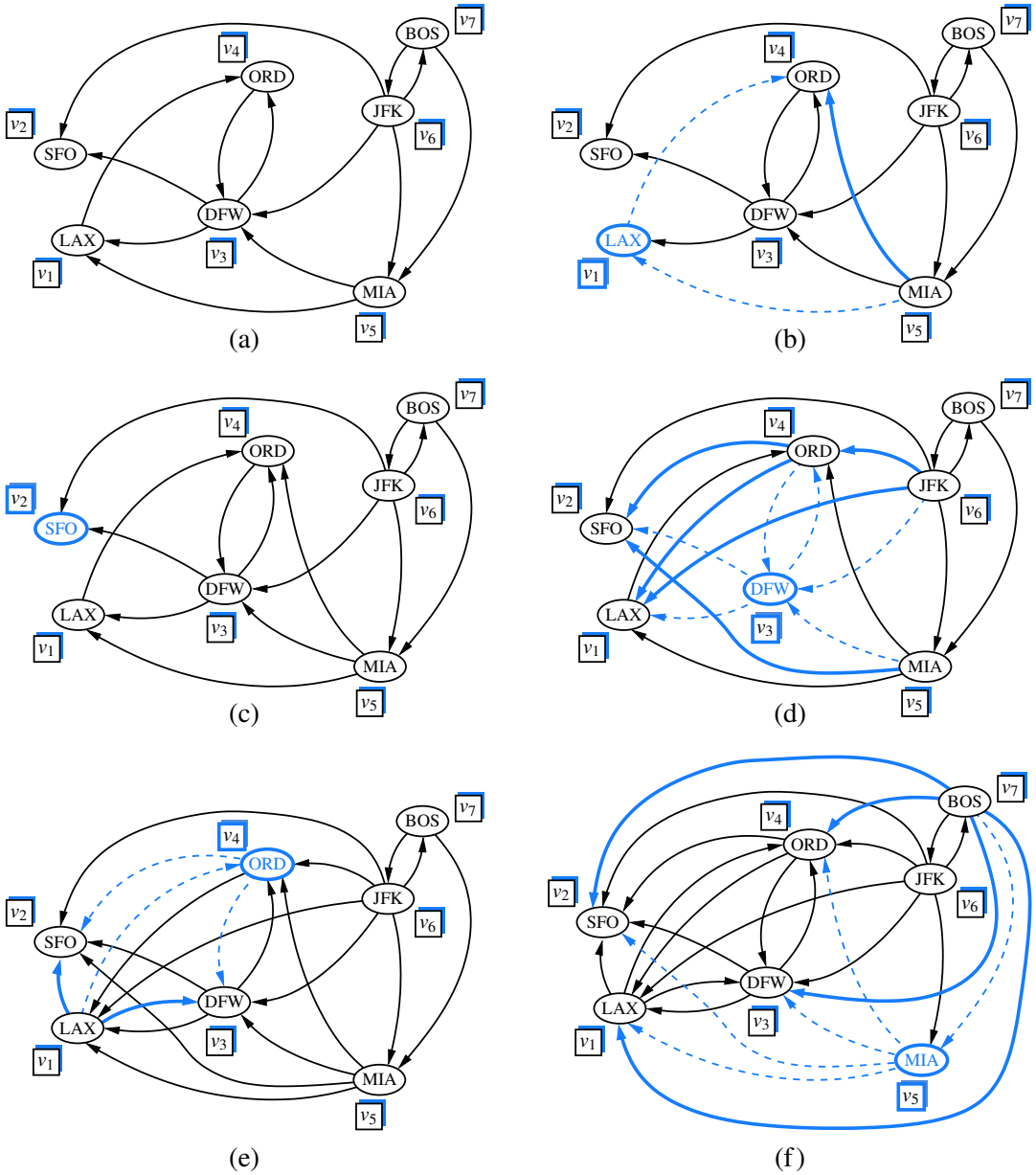


Figure 14.11: Sequence of directed graphs computed by the Floyd-Warshall algorithm: (a) initial directed graph $\vec{G} = \vec{G}_0$ and numbering of the vertices; (b) directed graph $\vec{G}_1$; (c) $\vec{G}_2$; (d) $\vec{G}_3$; (e) $\vec{G}_4$; (f) $\vec{G}_5$. Note that $\vec{G}_5 = \vec{G}_6 = \vec{G}_7$. If directed graph $\vec{G}_{k-1}$ has the edges (v_i, v_k) and (v_k, v_j), but not the edge (v_i, v_j), in the drawing of directed graph $\vec{G}_k$, we show edges (v_i, v_k) and (v_k, v_j) with dashed lines, and edge (v_i, v_j) with a thick line. For example, in (b) existing edges (MIA,LAX) and (LAX,ORD) result in new edge (MIA,ORD).

The importance of the Floyd-Warshall algorithm is that it is much easier to implement than repeated DFS, and much faster in practice because there are relatively few low-level operations hidden within the asymptotic notation. The algorithm is particularly well suited for the use of an adjacency matrix, as a single bit can be used to designate the reachability modeled as an edge (u,v) in the transitive closure.

However, note that repeated calls to DFS results in better asymptotic performance when the graph is sparse and represented using an adjacency list or adjacency map. In that case, a single DFS runs in $O(n+m)$ time, and so the transitive closure can be computed in $O(n^2+nm)$ time, which is preferable to $O(n^3)$.

Java Implementation

We will conclude with a Java implementation of the Floyd-Warshall algorithm, as presented in Code Fragment 14.10. Although the pseudocode for the algorithm describes a series of directed graphs $\vec{G}_0, \vec{G}_1, \ldots, \vec{G}_n$, we directly modify the original graph, repeatedly adding new edges to the closure as we progress through rounds of the Floyd-Warshall algorithm.

Also, the pseudocode for the algorithm describes the loops based on vertices being indexed from 0 to $n-1$. With our graph ADT, we prefer to use Java's for-each loop syntax directly on the vertices of the graph. Therefore, in Code Fragment 14.10, variables i, j, and k are references to vertices, not integer indices into the sequence of vertices.

Finally, we make one additional optimization in the Java implementation, relative to the pseudocode, by not bothering to iterate through values of j unless we have verified that edge (i,k) exists in the current version of the closure.

```
1  /** Converts graph g into its transitive closure. */
2  public static <V,E> void transitiveClosure(Graph<V,E> g) {
3    for (Vertex<V> k : g.vertices())
4      for (Vertex<V> i : g.vertices())
5        // verify that edge (i,k) exists in the partial closure
6        if (i != k && g.getEdge(i,k) != null)
7          for (Vertex<V> j : g.vertices())
8            // verify that edge (k,j) exists in the partial closure
9            if (i != j && j != k && g.getEdge(k,j) != null)
10             // if (i,j) not yet included, add it to the closure
11             if (g.getEdge(i,j) == null)
12               g.insertEdge(i, j, null);
13 }
```

Code Fragment 14.10: Java implementation of the Floyd-Warshall algorithm.

14.5 Directed Acyclic Graphs

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

- Prerequisites between courses of an academic program.
- Inheritance between classes of an object-oriented program.
- Scheduling constraints between the tasks of a project.

We will explore this latter application further in the following example:

Example 14.20: *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 u to v whenever the task for u must be executed before the task for v, then we define a directed acyclic graph.*

14.5.1 Topological Ordering

The example above motivates the following definition. Let $\vec{G}$ be a directed graph with n vertices. A ***topological ordering*** of $\vec{G}$ is an ordering $v_1, \ldots, v_n$ of the vertices of $\vec{G}$ such that for every edge (v_i, v_j) of $\vec{G}$, it is the case that $i < j$. That is, a topological ordering is an ordering such that any directed path in $\vec{G}$ traverses vertices in increasing order. Note that a directed graph may have more than one topological ordering. (See Figure 14.12.)

Proposition 14.21: *$\vec{G}$ 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 $\vec{G}$ is topologically ordered. Assume, for the sake of a contradiction, that $\vec{G}$ 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, $\vec{G}$ must be acyclic.

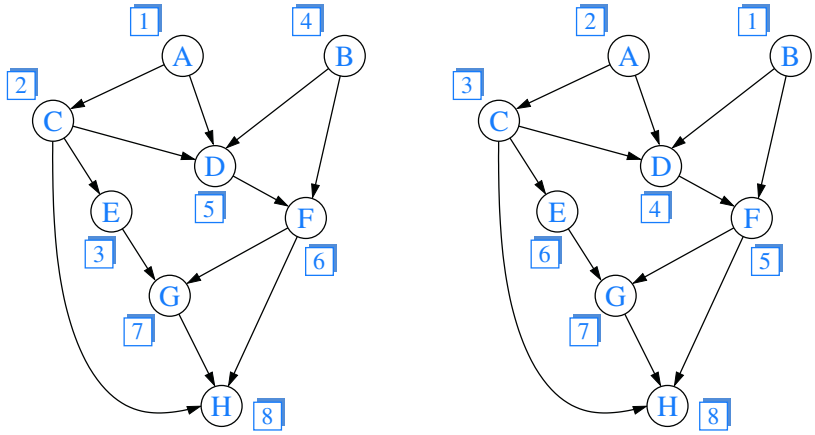


Figure 14.12: Two topological orderings of the same acyclic directed graph.

We now argue the sufficiency of the condition (the "if" part). Suppose $\vec{G}$ is acyclic. We will give an algorithmic description of how to build a topological ordering for $\vec{G}$. Since $\vec{G}$ is acyclic, $\vec{G}$ 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 $\vec{G}$. If we remove v_1 from $\vec{G}$, together with its outgoing edges, the resulting directed graph is still acyclic. Hence, the resulting directed graph also has a vertex with no incoming edges, and we let v_2 be such a vertex. By repeating this process until the directed graph becomes empty, we obtain an ordering $v_1, \ldots, v_n$ of the vertices of $\vec{G}$. Because of the construction above, if (v_i, v_j) is an edge of $\vec{G}$, then v_i must be deleted before v_j can be deleted, and thus, $i < j$. Therefore, $v_1, \ldots, v_n$ is a topological ordering. ■

Proposition 14.21's justification suggests an algorithm for computing a topological ordering of a directed graph, which we call ***topological sorting***. We present a Java implementation of the technique in Code Fragment 14.11, and an example execution of the algorithm in Figure 14.13. Our implementation uses a map, named inCount, to map each vertex v to a counter that represents the current number of incoming edges to v, excluding those coming from vertices that have previously been added to the topological order. As was the case with our graph traversals, a hash-based map only provides $O(1)$ expected time access to its entries, rather than worst-case time. This could easily be converted to worst-case time if vertices could be indexed from 0 to $n-1$, or if we store the count as a field of the vertex instance.

As a side effect, the topological sorting algorithm of Code Fragment 14.11 also tests whether the given directed graph $\vec{G}$ 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.

```
1  /** Returns a list of verticies of directed acyclic graph g in topological order. */
2  public static <V,E> PositionalList<Vertex<V>> topologicalSort(Graph<V,E> g) {
3    // list of vertices placed in topological order
4    PositionalList<Vertex<V>> topo = new LinkedPositionalList<>();
5    // container of vertices that have no remaining constraints
6    Stack<Vertex<V>> ready = new LinkedStack<>();
7    // map keeping track of remaining in-degree for each vertex
8    Map<Vertex<V>, Integer> inCount = new ProbeHashMap<>();
9    for (Vertex<V> u : g.vertices()) {
10     inCount.put(u, g.inDegree(u));          // initialize with actual in-degree
11     if (inCount.get(u) == 0)                // if u has no incoming edges,
12       ready.push(u);                        // it is free of constraints
13   }
14   while (!ready.isEmpty()) {
15     Vertex<V> u = ready.pop();
16     topo.addLast(u);
17     for (Edge<E> e : g.outgoingEdges(u)) {  // consider all outgoing neighbors of u
18       Vertex<V> v = g.opposite(u, e);
19       inCount.put(v, inCount.get(v) - 1);   // v has one less constraint without u
20       if (inCount.get(v) == 0)
21         ready.push(v);
22     }
23   }
24   return topo;
25 }
```

Code Fragment 14.11: Java implementation for the topological sorting algorithm. (We show an example execution of this algorithm in Figure 14.13.)

Proposition 14.22: *Let $\vec{G}$ be a directed graph with n vertices and m edges, using an adjacency list representation. The topological sorting algorithm runs in $O(n+m)$ time using $O(n)$ auxiliary space, and either computes a topological ordering of $\vec{G}$ or fails to include some vertices, which indicates that $\vec{G}$ has a directed cycle.*

Justification: The initial recording of the n in-degrees uses $O(n)$ time based on the inDegree method. Say that a vertex u is ***visited*** by the topological sorting algorithm when u is removed from the ready list. A vertex u can be visited only when inCount.get(u) is 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. In accordance with Proposition 14.9, the running time is $(n+m)$. Regarding the space usage, observe that containers topo, ready, and inCount have at most one entry per vertex, and therefore use $O(n)$ space. ■

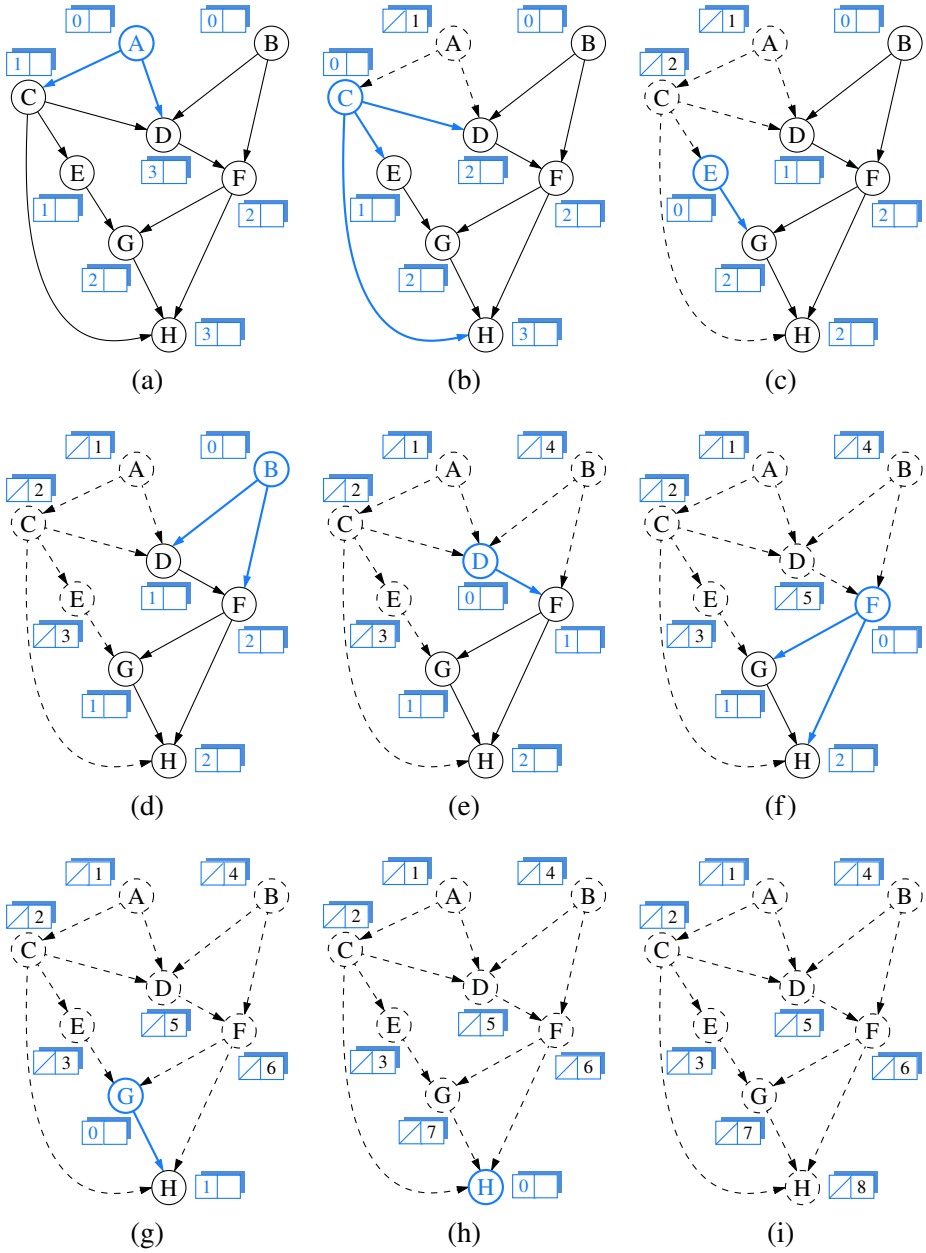


Figure 14.13: Example of a run of algorithm topologicalSort (Code Fragment 14.11). The label near a vertex shows its current inCount value, and its eventual rank in the resulting topological order. The highlighted vertex is one with inCount equal to zero that will become the next vertex in the topological order. Dashed lines denote edges that have already been examined, which are no longer reflected in the inCount values.

14.6 Shortest Paths

As we saw in Section 14.3.3, the breadth-first search strategy can be used to find a path with as few edges as possible 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 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 probably not appropriate for all the edges to be equal to each other, for some inter-city distances will likely be much larger than others. Likewise, 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 again may not be appropriate for all the edges to be equal to each other, for some connections in a computer network are typically much faster than others (for example, some edges might represent low-bandwidth connections, while others might represent high-speed, fiber-optic connections). It is natural, therefore, to consider graphs whose edges are not weighted equally.

14.6.1 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. For $e = (u,v)$, we let notation $w(u,v) = w(e)$. We show an example of a weighted graph in Figure 14.14.

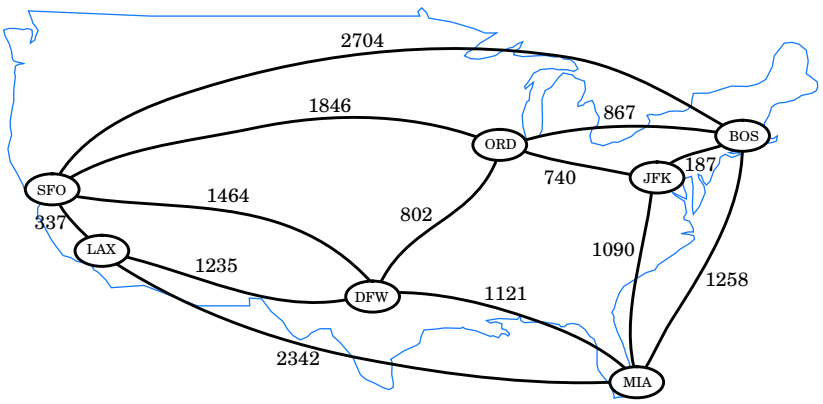


Figure 14.14: A weighted graph whose vertices represent major U.S. airports and whose edge weights represent distances in miles. This graph has a path from JFK to LAX of total weight 2,777 (going through ORD and DFW). This is the minimum-weight path in the graph from JFK to LAX.

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

People often use the convention that $d(u,v) = \infty$ if there is no path at all from u to v in G. Even if there is a path from u to v in G, however, if there is a cycle in G whose total weight is negative, the distance from u to v may not be defined. 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 s 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 14.3.3.

There is an interesting approach for solving this ***single-source*** problem based on the ***greedy-method*** design pattern (Section 13.4.2). 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.

14.6.2 Dijkstra's Algorithm

The main idea in applying the greedy-method pattern to the single-source shortest-path problem is to perform a "weighted" breadth-first search starting at the source vertex s. In particular, we can use the greedy method to develop an algorithm that iteratively grows a "cloud" of vertices out of s, with the vertices entering the cloud in order of their distances from s. Thus, in each iteration, the next vertex chosen is the vertex outside the cloud that is closest to s. The algorithm terminates when no more vertices are outside the cloud (or when those outside the cloud are not connected to those within the cloud), at which point we have a shortest path from s to every vertex of G that is reachable from s. This approach is a simple, but nevertheless powerful, example of the greedy-method design pattern. Applying the greedy method to the single-source, shortest-path problem, results in an algorithm known as ***Dijkstra's algorithm***.

Edge Relaxation

Let us define a label $D[v]$ for each vertex v in V, which we use to approximate the distance in G from s to v. The meaning of these labels is that $D[v]$ will always store the length of the best path we have found ***so far*** from s to v. Initially, $D[s] = 0$ and $D[v] = \infty$ for each $v \neq s$, and we define the set C, which is our "***cloud***" of vertices, to initially be the empty set. 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 general, we will use a priority queue to select among the vertices outside the cloud.) In the very first iteration we will, of course, pull s into C. Once a new vertex u is pulled into C, we update the label $D[v]$ of each vertex v 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 v via u. This update operation is known as a ***relaxation*** procedure, for it takes an old estimate and checks if it can be improved to get closer to its true value. The specific edge relaxation operation is as follows:

Edge Relaxation:

if $D[u] + w(u,v) < D[v]$ **then**
$D[v] = D[u] + w(u,v)$

Algorithm Description and Example

We give the pseudocode for Dijkstra's algorithm in Code Fragment 14.12, and illustrate several iterations of Dijkstra's algorithm in Figures 14.15 through 14.17.

Algorithm ShortestPath(G,s):
Input: A directed or undirected graph G with nonnegative edge weights, and a distinguished vertex s of G.
Output: The length of a shortest path from s to v for each vertex v of G.

Initialize $D[s] = 0$ and $D[v] = \infty$ for each vertex $v \neq s$.
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}
  $u =$ value returned by Q.removeMin()
  for each edge (u,v) such that v is in Q **do**
    {perform the ***relaxation*** procedure on edge (u,v)}
    if $D[u] + w(u,v) < D[v]$ **then**
      $D[v] = D[u] + w(u,v)$
      Change the key of vertex v in Q to $D[v]$.
return the label $D[v]$ of each vertex v

Code Fragment 14.12: Pseudocode for Dijkstra's algorithm, solving the single-source shortest-path problem for an undirected or directed graph.

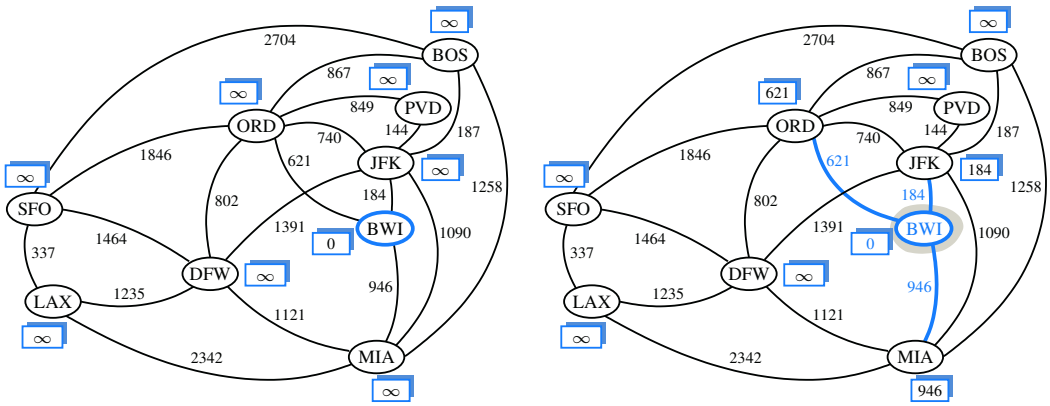


Figure 14.15: An example execution of Dijkstra's shortest-path algorithm on a weighted graph. The start vertex is BWI. A box next to each vertex v stores the label $D[v]$. The edges of the shortest-path tree are drawn as thick arrows, and for each vertex u outside the "cloud" we show the current best edge for pulling in u with a thick line. (Continues in Figure 14.16.)

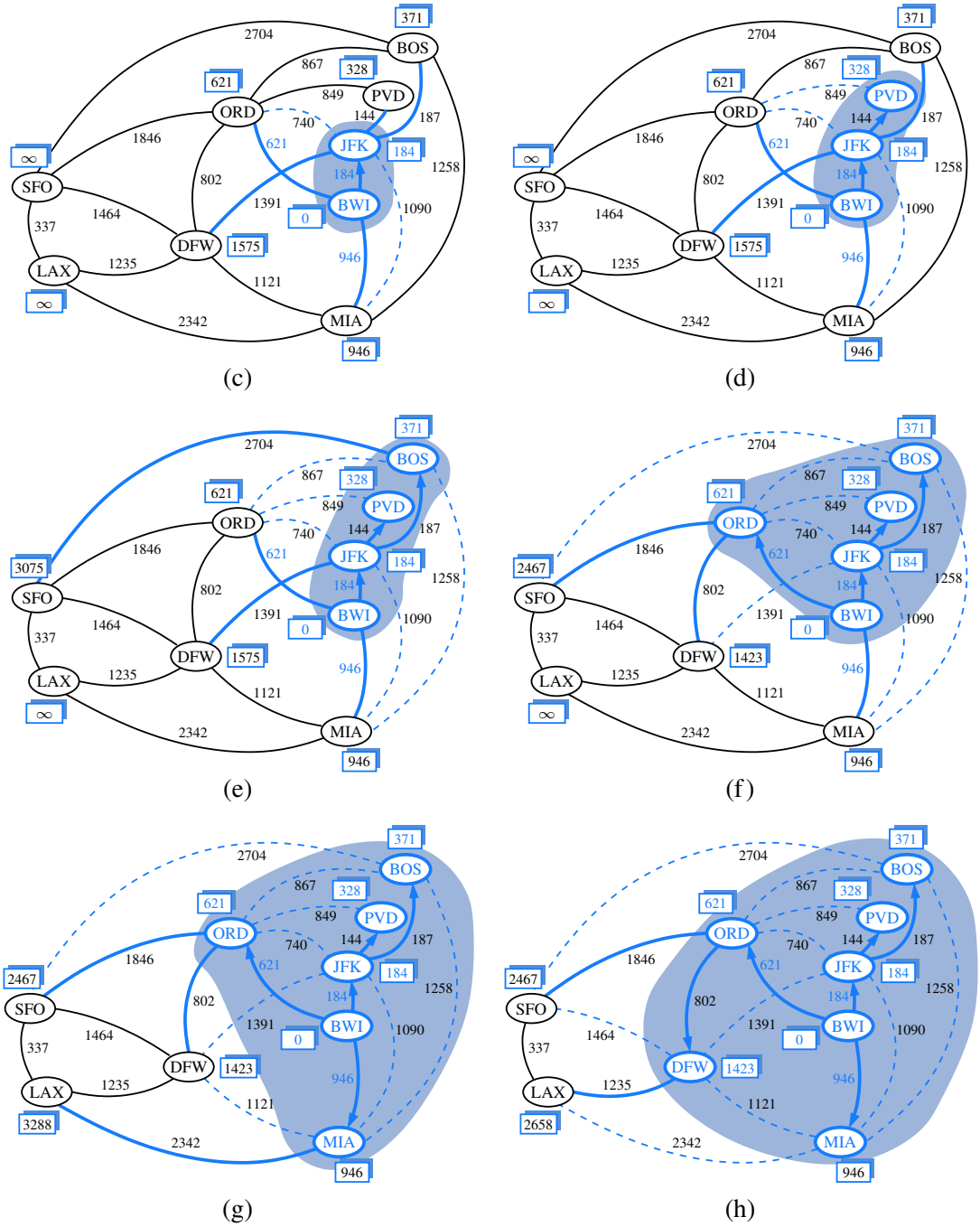


Figure 14.16: An example execution of Dijkstra's shortest-path algorithm on a weighted graph. (Continued from Figure 14.15; continues in Figure 14.17.)

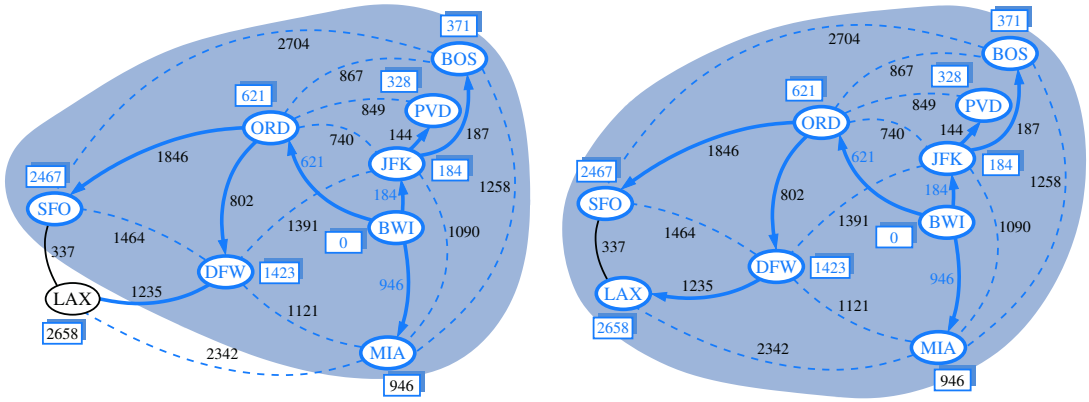


Figure 14.17: An example execution of Dijkstra's shortest-path algorithm on a weighted graph. (Continued from Figure 14.16.)

Why It Works

The interesting 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 s 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 s to each other vertex u in the graph. Why is it that the distance from s to u is equal to the value of the label $D[u]$ at the time vertex u is removed from the priority queue Q and added to the cloud C? The answer to this question depends on there being no negative-weight edges in the graph, for it allows the greedy method to work correctly, as we show in the proposition that follows.

Proposition 14.23: *In Dijkstra's algorithm, whenever a vertex v is pulled into the cloud, the label $D[v]$ is equal to $d(s,v)$, the length of a shortest path from s to v.*

Justification: Suppose that $D[v] > d(s,v)$ for some vertex v in V, and let z be the ***first*** vertex the algorithm pulled into the cloud C (that is, removed from Q) such that $D[z] > d(s,z)$. There is a shortest path P from s to z (for otherwise $d(s,z) = \infty = D[z]$). Let us therefore consider the moment when z is pulled into C, and let y be the first vertex of P (when going from s to z) that is not in C at this moment. Let x be the predecessor of y in path P (note that we could have $x = s$). (See Figure 14.18.) We know, by our choice of y, that x is already in C at this point.

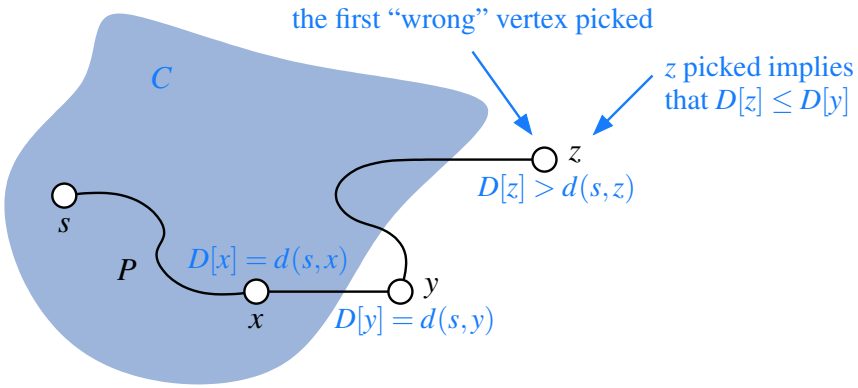


Figure 14.18: A schematic illustration for the justification of Proposition 14.23.

Moreover, $D[x] = d(s,x)$, since z is the ***first*** incorrect vertex. When x was pulled into C, we tested (and possibly updated) $D[y]$ so that we had at that point

$$D[y] \leq D[x] + w(x,y) = d(s,x) + w(x,y).$$

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

$$D[y] = d(s,y).$$

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

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

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

$$d(s,y) + d(y,z) = d(s,z).$$

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

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

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

The Running Time of Dijkstra's Algorithm

In this section, we analyze the time complexity of Dijkstra's algorithm. We denote with n and m the number of vertices and edges of the input graph G, 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 14.12, 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 or adjacency map structure. This data structure allows us to step through the vertices adjacent to u during the relaxation step in time proportional to their number. Therefore, the time spent in the management of the nested **for** loop, and the number of iterations of that loop, is

$$\sum_{u \text{ in } V_G} \text{outdeg}(u),$$

which is $O(m)$ by Proposition 14.9. The outer **while** loop executes $O(n)$ times, since a new vertex is added to the cloud during each iteration. This still does not settle all the details for the algorithm analysis, however, for we must say more about how to implement the other principal data structure in the algorithm—the priority queue Q.

Referring back to Code Fragment 14.12 in search of priority queue operations, we find that n vertices are originally inserted into the priority queue; since these are the only insertions, the maximum size of the queue is n. In each of n iterations of the **while** loop, a call to removeMin is made to extract the vertex u with smallest D label from Q. Then, for each neighbor v of u, we perform an edge relaxation, and may potentially update the key of v in the queue. Thus, we actually need an implementation of an ***adaptable priority queue*** (Section 9.5), in which case the key of a vertex v is changed using the method replaceKey(e, k), where e is the priority queue entry associated with vertex v. In the worst case, there could be one such update for each edge of the graph. Overall, the running time of Dijkstra's algorithm is bounded by the sum of the following:

- n insertions into Q.
- n calls to the removeMin method on Q.
- m calls to the replaceKey method on Q.

If Q is an adaptable priority queue implemented as a heap, then each of the above operations run in $O(\log n)$, and so the overall running time for Dijkstra's algorithm is $O((n+m)\log n)$. 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.

Let us now consider an alternative implementation for the adaptable priority queue Q using an unsorted sequence. (See Exercise P-9.52.) This, of course, requires that we spend $O(n)$ time to extract the minimum element, but it affords very fast key updates, provided Q supports location-aware entries (Section 9.5.1). Specifically, we can implement each key update done in a relaxation step in $O(1)$ time—we simply change the key value once we locate the entry in Q to update. Hence, this implementation results in a running time that is $O(n^2+m)$, which can be simplified to $O(n^2)$ since G is simple.

Comparing the Two Implementations

We have two choices for implementing the adaptable priority queue with location-aware entries in Dijkstra's algorithm: a heap implementation, which yields a running time of $O((n+m)\log n)$, and an unsorted sequence implementation, which yields a running time of $O(n^2)$. Since both implementations would be fairly simple to code, they are about equal in terms of the programming sophistication needed. These two implementations are also about equal in terms of the constant factors in their worst-case running times. Looking only at these worst-case times, we prefer the heap implementation when the number of edges in the graph is small (that is, when $m < n^2/\log n$), and we prefer the sequence implementation when the number of edges is large (that is, when $m > n^2/\log n$).

Proposition 14.24: *Given a weighted graph G with n vertices and m edges, such that the weight of each edge is nonnegative, and a vertex s of G, Dijkstra's algorithm can compute the distance from s to all other vertices of G in the better of $O(n^2)$ or $O((n+m)\log n)$ time.*

We note that an advanced priority queue implementation, known as a ***Fibonacci heap***, can be used to implement Dijkstra's algorithm in $O(m+n\log n)$ time.

Programming Dijkstra's Algorithm in Java

Having given a pseudocode description of Dijkstra's algorithm, let us now present Java code for performing Dijkstra's algorithm, assuming we are given a graph whose edge elements are nonnegative integer weights. Our implementation of the algorithm is in the form of a method, shortestPathLengths, that takes a graph and a designated source vertex as parameters. (See Code Fragment 14.13.) It returns a map, named cloud, storing the shortest-path distance $d(s,v)$ for each vertex v that is reachable from the source. We rely on our HeapAdaptablePriorityQueue developed in Section 9.5.2 as an adaptable priority queue.

As we have done with other algorithms in this chapter, we rely on hash-based maps to store auxiliary data (in this case, mapping v to its distance bound $D[v]$ and its adaptable priority queue entry). The expected $O(1)$-time access to elements of these dictionaries could be converted to worst-case bounds, either by numbering vertices from 0 to $n-1$ to use as indices into an array, or by storing the information within each vertex's element.

The pseudocode for Dijkstra's algorithm begins by assigning $D[v] = \infty$ for each v other than the source; we rely on the special value Integer.MAX_VALUE in Java to provide a sufficient numeric value to model infinity. However, we avoid including vertices with this "infinite" distance in the resulting cloud that is returned by the method. The use of this numeric limit could be avoided altogether by waiting to add a vertex to the priority queue until after an edge that reaches it is relaxed. (See Exercise C-14.62.)

```
1  /** Computes shortest-path distances from src vertex to all reachable vertices of g. */
2  public static <V> Map<Vertex<V>, Integer>
3  shortestPathLengths(Graph<V,Integer> g, Vertex<V> src) {
4    // d.get(v) is upper bound on distance from src to v
5    Map<Vertex<V>, Integer> d = new ProbeHashMap<>();
6    // map reachable v to its d value
7    Map<Vertex<V>, Integer> cloud = new ProbeHashMap<>();
8    // pq will have vertices as elements, with d.get(v) as key
9    AdaptablePriorityQueue<Integer, Vertex<V>> pq;
10   pq = new HeapAdaptablePriorityQueue<>();
11   // maps from vertex to its pq locator
12   Map<Vertex<V>, Entry<Integer,Vertex<V>>> pqTokens;
13   pqTokens = new ProbeHashMap<>();
14
15   // for each vertex v of the graph, add an entry to the priority queue, with
16   // the source having distance 0 and all others having infinite distance
17   for (Vertex<V> v : g.vertices()) {
18     if (v == src)
19       d.put(v,0);
20     else
21       d.put(v, Integer.MAX_VALUE);
22     pqTokens.put(v, pq.insert(d.get(v), v));          // save entry for future updates
23   }
24   // now begin adding reachable vertices to the cloud
25   while (!pq.isEmpty()) {
26     Entry<Integer, Vertex<V>> entry = pq.removeMin();
27     int key = entry.getKey();
28     Vertex<V> u = entry.getValue();
29     cloud.put(u, key);                                // this is actual distance to u
30     pqTokens.remove(u);                               // u is no longer in pq
31     for (Edge<Integer> e : g.outgoingEdges(u)) {
32       Vertex<V> v = g.opposite(u,e);
33       if (cloud.get(v) == null) {
34         // perform relaxation step on edge (u,v)
35         int wgt = e.getElement();
36         if (d.get(u) + wgt < d.get(v)) {              // better path to v?
37           d.put(v, d.get(u) + wgt);                   // update the distance
38           pq.replaceKey(pqTokens.get(v), d.get(v));   // update the pq entry
39         }
40       }
41     }
42   }
43   return cloud;          // this only includes reachable vertices
44 }
```

Code Fragment 14.13: Java implementation of Dijkstra's algorithm for computing the shortest-path distances from a single source. We assume that *e*.getElement( ) for edge *e* represents the weight of that edge.

Reconstructing a Shortest-Path Tree

Our pseudocode description of Dijkstra's algorithm in Code Fragment 14.12 and our implementation in Code Fragment 14.13 compute the value $D[v]$, for each vertex v, that is the length of a shortest path from the source vertex s to v. However, those forms of the algorithm do not explicitly compute the actual paths that achieve those distances. Fortunately, it is possible to represent shortest paths from source s to every reachable vertex in a graph using a compact data structure known as a ***shortest-path tree***. This is possible because if a shortest path from s to v passes through an intermediate vertex u, it must begin with a shortest path from s to u.

We next demonstrate that a shortest-path tree rooted at source s can be reconstructed in $O(n+m)$ time, given the $D[v]$ values produced by Dijkstra's algorithm using s as the source. As we did when representing the DFS and BFS trees, we will map each vertex $v \neq s$ to a parent u (possibly, $u = s$), such that u is the vertex immediately before v on a shortest path from s to v. If u is the vertex just before v on a shortest path from s to v, it must be that

$$D[u] + w(u,v) = D[v].$$

Conversely, if the above equation is satisfied, then a shortest path from s to u followed by the edge (u,v) is a shortest path to v.

Our implementation in Code Fragment 14.14 reconstructs a tree based on this logic, testing all *incoming* edges to each vertex v, looking for a (u,v) that satisfies the key equation. The running time is $O(n+m)$, as we consider each vertex and all incoming edges to those vertices. (See Proposition 14.9.)

```
1  /**
2   * Reconstructs a shortest-path tree rooted at vertex s, given distance map d.
3   * The tree is represented as a map from each reachable vertex v (other than s)
4   * to the edge e = (u,v) that is used to reach v from its parent u in the tree.
5   */
6  public static <V> Map<Vertex<V>,Edge<Integer>>
7  spTree(Graph<V,Integer> g, Vertex<V> s, Map<Vertex<V>,Integer> d) {
8    Map<Vertex<V>, Edge<Integer>> tree = new ProbeHashMap<>();
9    for (Vertex<V> v : d.keySet())
10     if (v != s)
11       for (Edge<Integer> e : g.incomingEdges(v)) { // consider INCOMING edges
12         Vertex<V> u = g.opposite(v, e);
13         int wgt = e.getElement();
14         if (d.get(v) == d.get(u) + wgt)
15           tree.put(v, e);                          // edge is is used to reach v
16       }
17   return tree;
18 }
```

Code Fragment 14.14: Java method that reconstructs a single-source shortest-path tree, based on knowledge of the shortest-path distances.

14.7 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 an undirected, weighted graph G whose vertices represent the computers, and whose edges represent all the possible pairs (u,v) of computers, where the weight $w(u,v)$ of edge (u,v) is equal to the amount of cable needed to connect computer u to computer v. Rather than computing a shortest-path tree from some particular vertex v, we are interested instead in finding a tree T that contains all the vertices of G and has the minimum total weight over all such trees. Algorithms for finding such a tree are the focus of this section.

Problem Definition

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

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

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 the Prim-Jarník algorithm, which grows the MST from a single root vertex, much in the same way as Dijkstra's shortest-path algorithm. The second algorithm we discuss is Kruskal's algorithm, which "grows" the MST in clusters by considering edges in nondecreasing order of their weights.

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,v).

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 14.19.)

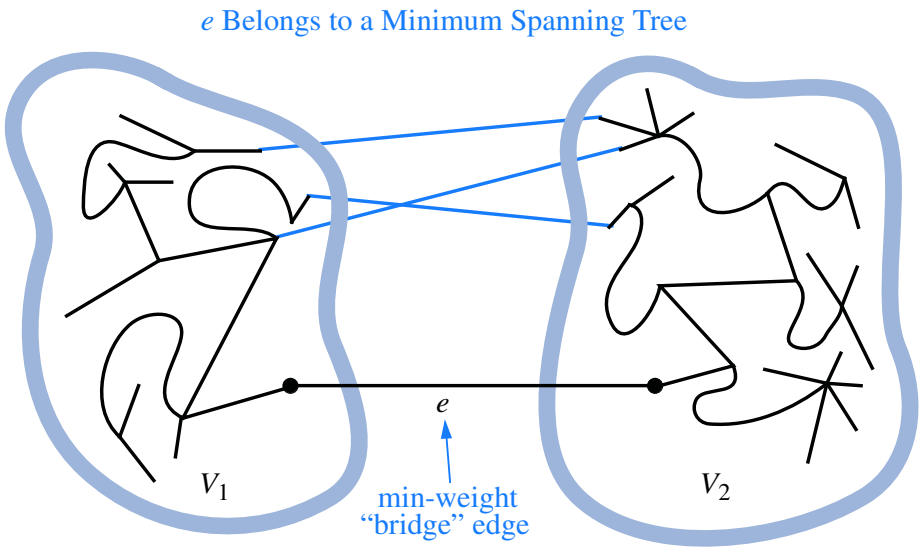


Figure 14.19: An illustration of the crucial fact about minimum spanning trees.

Proposition 14.25: *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 $f \neq e$ 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 was 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 (C-14.64). In addition, note that Proposition 14.25 remains valid even if the graph G contains negative-weight edges or negative-weight cycles, unlike the algorithms we presented for shortest paths.

14.7.1 Prim-Jarník Algorithm

In the Prim-Jarník algorithm, we grow a minimum spanning tree from a single cluster starting from some "root" vertex s. The main idea is similar to that of Dijkstra's algorithm. We will begin with some vertex s, defining the initial "cloud" of vertices C. Then, in each iteration, we choose a minimum-weight edge $e = (u,v)$, connecting a vertex u in the cloud C to a vertex v outside of C. The vertex v 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 into play, for 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 $D[v]$ for each vertex v outside the cloud C, so that $D[v]$ stores the weight of the minimum observed edge for joining v to the cloud C. (In Dijkstra's algorithm, this label measured the full path length from starting vertex s to v, including an edge (u,v).) These labels serve as keys in a priority queue used to decide which vertex is next in line to join the cloud. We give the pseudocode in Code Fragment 14.15.

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

```
Pick any vertex s of G
D[s] = 0
for each vertex v ≠ s do
    D[v] = ∞
Initialize T = ∅.
Initialize a priority queue Q with an entry (D[v], v) for each vertex v.
For each vertex v, maintain connect(v) as the edge achieving D[v] (if any).
while Q is not empty do
    Let u be the value of the entry returned by Q.removeMin().
    Connect vertex u to T using edge connect(e).
    for each edge e' = (u,v) such that v is in Q do
        {check if edge (u,v) better connects v to T}
        if w(u,v) < D[v] then
            D[v] = w(u,v)
            connect(v) = e'.
            Change the key of vertex v in Q to D[v].
return the tree T
```

Code Fragment 14.15: The Prim-Jarník algorithm for the MST problem.

Analyzing the Prim-Jarník Algorithm

The implementation issues for the Prim-Jarník algorithm are similar to those for Dijkstra's algorithm, relying on an adaptable priority queue Q (Section 9.5.1). We initially perform n insertions into Q, later perform n extract-min operations, and may update a total of m priorities as part of the algorithm. Those steps are the primary contributions to the overall running time. With a heap-based priority queue, each operation runs in $O(\log n)$ time, and the overall time for the algorithm is $O((n+m)\log n)$, which is $O(m \log n)$ for a connected graph. Alternatively, we can achieve $O(n^2)$ running time by using an unsorted list as a priority queue.

Illustrating the Prim-Jarník Algorithm

We illustrate the Prim-Jarník algorithm in Figures 14.20 and 14.21.

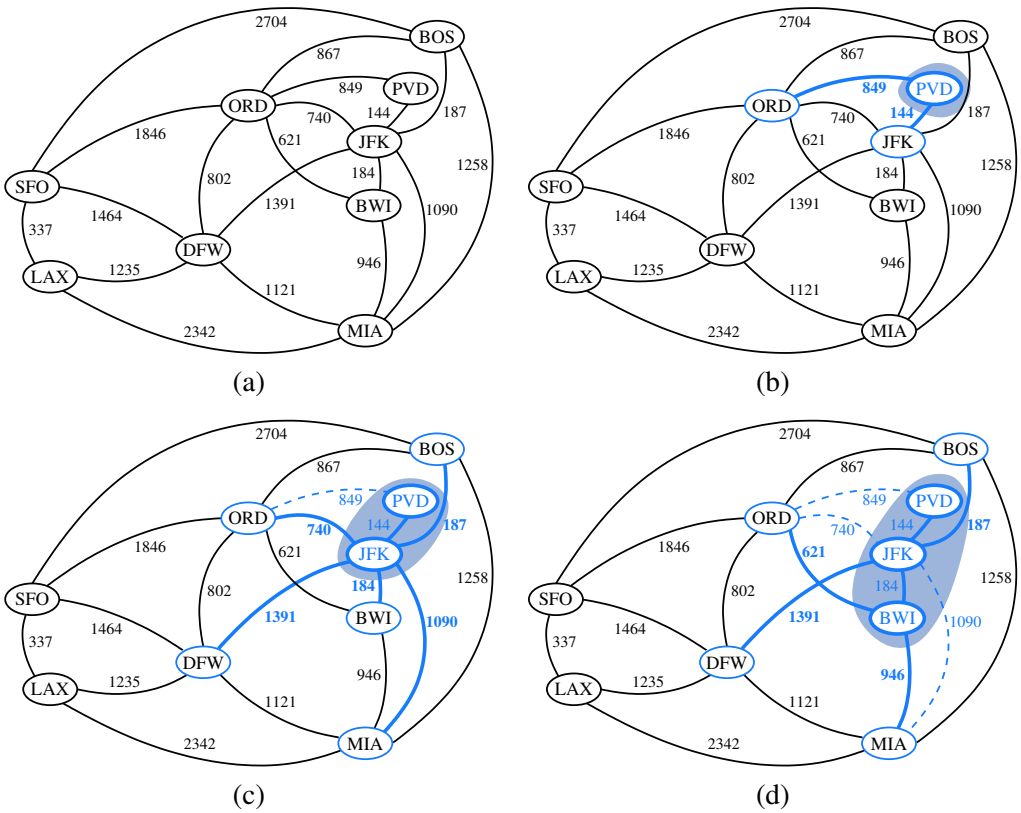


Figure 14.20: An illustration of the Prim-Jarník MST algorithm, starting with vertex PVD. (Continues in Figure 14.21.)

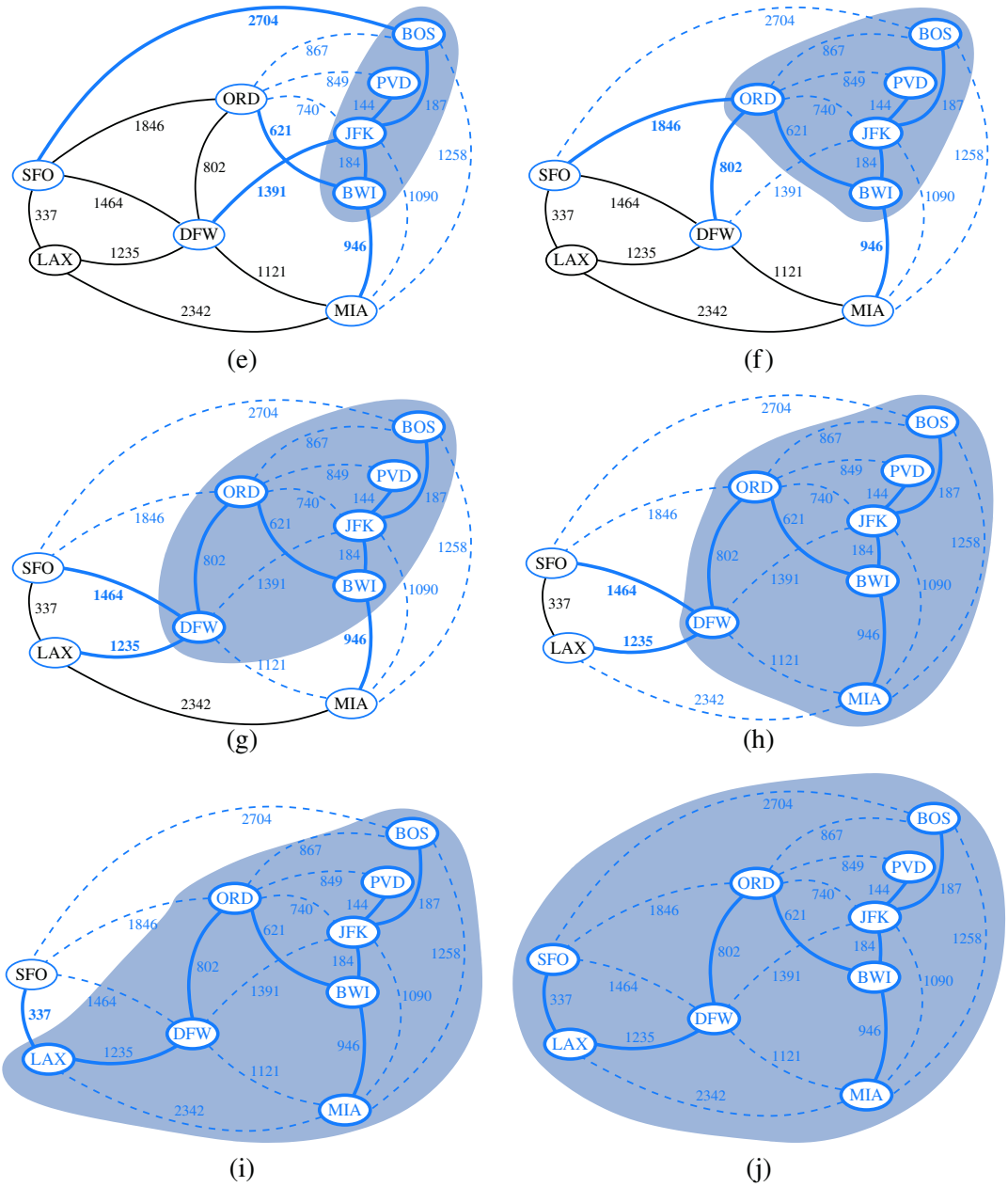


Figure 14.21: An illustration of the Prim-Jarník MST algorithm. (Continued from Figure 14.20.)

14.7.2 Kruskal's Algorithm

In this section, we will introduce ***Kruskal's algorithm*** for constructing a minimum spanning tree. While the Prim-Jarník algorithm builds the MST by growing a single tree until it spans the graph, Kruskal's algorithm maintains many smaller trees in a ***forest***, repeatedly merging pairs of trees until a single tree spans the graph.

Initially, each vertex is in its own cluster. The algorithm then considers each edge in turn, ordered by increasing weight. If an edge e connects vertices in two different clusters, then e is added to the set of edges of the minimum spanning tree, and the two trees are merged with the addition of e. If, on the other hand, e connects two vertices 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 pseudocode for Kruskal's MST algorithm in Code Fragment 14.16 and we show an example of this algorithm in Figures 14.22, 14.23, and 14.24.

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 $C(v) = \{v\}$.
Initialize a priority queue Q to contain all edges in G, using the weights as keys.
$T = \emptyset$ {T will ultimately contain the edges of an MST}
while T has fewer than $n-1$ edges **do**
  $(u,v) =$ value returned by Q.removeMin()
  Let $C(u)$ be the cluster containing u, and let $C(v)$ be the cluster containing v.
  if $C(u) \neq C(v)$ **then**
    Add edge (u,v) to T.
    Merge $C(u)$ and $C(v)$ into one cluster.
return tree T

Code Fragment 14.16: Kruskal's algorithm for the MST problem.

As was the case with the Prim-Jarník algorithm, the correctness of Kruskal's algorithm is based upon the crucial fact about minimum spanning trees from Proposition 14.25. Each time Kruskal's algorithm adds an edge (u,v) 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.

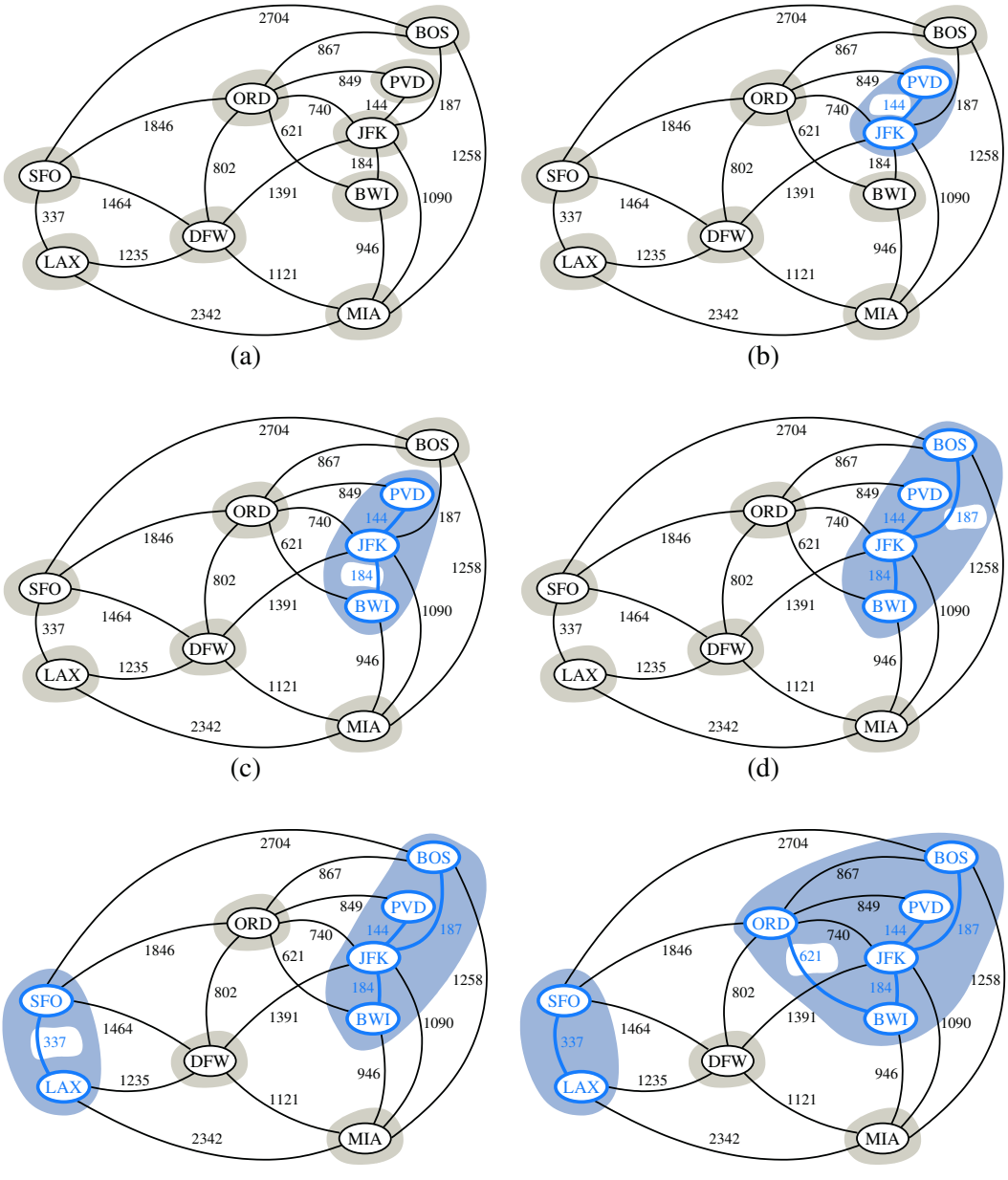


Figure 14.22: Example of an execution of Kruskal's MST algorithm on a graph with integer weights. We show the clusters as shaded regions and we highlight the edge being considered in each iteration. (Continues in Figure 14.23.)

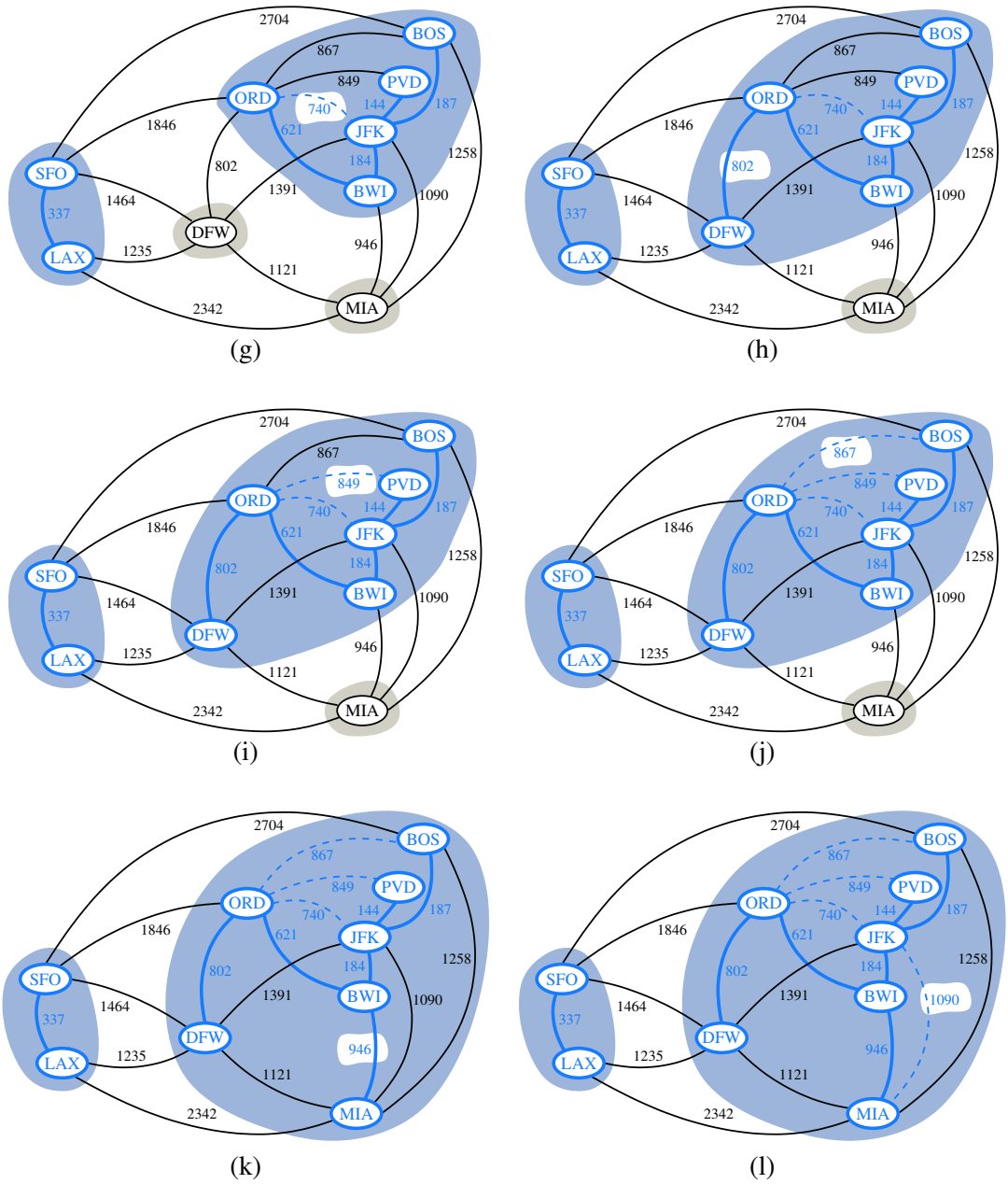


Figure 14.23: An example of an execution of Kruskal's MST algorithm. Rejected edges are shown dashed. (Continues in Figure 14.24.)

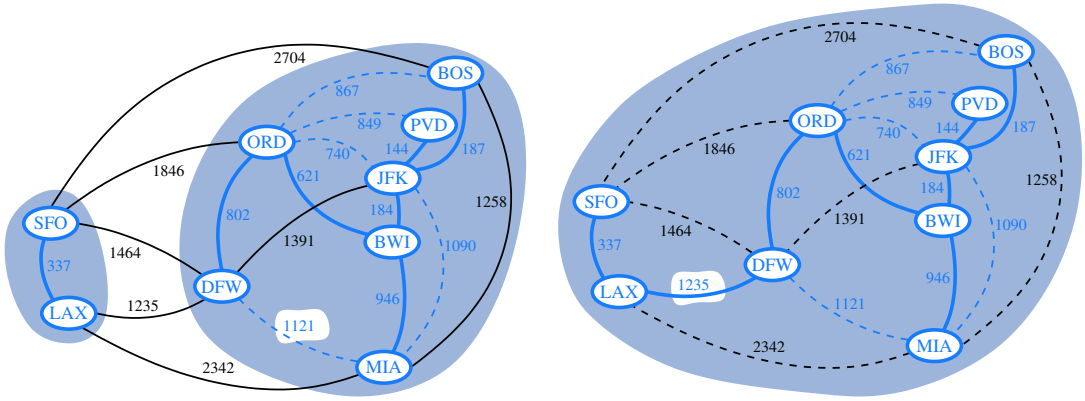


Figure 14.24: Example of an execution of Kruskal's MST algorithm (continued). The edge considered in (n) merges the last two clusters, which concludes this execution of Kruskal's algorithm. (Continued from Figure 14.23.)

The Running Time of Kruskal's Algorithm

There are two primary contributions to the running time of Kruskal's algorithm. The first is the need to consider the edges in nondecreasing order of their weights, and the second is the management of the cluster partition. Analyzing its running time requires that we give more details on its implementation.

The ordering of edges by weight can be implemented in $O(m \log m)$, either by use of a sorting algorithm or a priority queue Q. If that queue is implemented with a heap, 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 9.3.4), and the subsequent calls to removeMin each run in $O(\log m)$ time, since the queue has size $O(m)$. We note that since m is $O(n^2)$ for a simple graph, $O(\log m)$ is the same as $O(\log n)$. Therefore, the running time due to the ordering of edges is $O(m \log n)$.

The remaining task is the management of clusters. To implement Kruskal's algorithm, we must be able to find the clusters for vertices u and v that are endpoints of an edge e, to test whether those two clusters are distinct, and if so, to merge those two clusters into one. None of the data structures we have studied thus far are well suited for this task. However, we conclude this chapter by formalizing the problem of managing ***disjoint partitions***, and introducing efficient ***union-find*** data structures. In the context of Kruskal's algorithm, we perform at most $2m$ "find" operations and $n-1$ "union" operations. We will see that a simple union-find structure can perform that combination of operations in $O(m + n \log n)$ time (see Proposition 14.26), and a more advanced structure can support an even faster time.

For a connected graph, $m \geq n-1$; therefore, the bound of $O(m \log n)$ time for ordering the edges dominates the time for managing the clusters. We conclude that the running time of Kruskal's algorithm is $O(m \log n)$.

Java Implementation

Code Fragment 14.17 presents a Java implementation of Kruskal's algorithm. The minimum spanning tree is returned in the form of a list of edges. As a consequence of Kruskal's algorithm, those edges will be reported in nondecreasing order of their weights.

Our implementation assumes use of a Partition class for managing the cluster partition. An implementation of the Partition class is presented in Section 14.7.3.

```
1  /** Computes a minimum spanning tree of graph g using Kruskal's algorithm. */
2  public static <V> PositionalList<Edge<Integer>> MST(Graph<V,Integer> g) {
3    // tree is where we will store result as it is computed
4    PositionalList<Edge<Integer>> tree = new LinkedPositionalList<>();
5    // pq entries are edges of graph, with weights as keys
6    PriorityQueue<Integer, Edge<Integer>> pq = new HeapPriorityQueue<>();
7    // union-find forest of components of the graph
8    Partition<Vertex<V>> forest = new Partition<>();
9    // map each vertex to the forest position
10   Map<Vertex<V>,Position<Vertex<V>>> positions = new ProbeHashMap<>();
11
12   for (Vertex<V> v : g.vertices())
13     positions.put(v, forest.makeGroup(v));
14
15   for (Edge<Integer> e : g.edges())
16     pq.insert(e.getElement(), e);
17
18   int size = g.numVertices();
19   // while tree not spanning and unprocessed edges remain...
20   while (tree.size() != size - 1 && !pq.isEmpty()) {
21     Entry<Integer, Edge<Integer>> entry = pq.removeMin();
22     Edge<Integer> edge = entry.getValue();
23     Vertex<V>[ ] endpoints = g.endVertices(edge);
24     Position<Vertex<V>> a = forest.find(positions.get(endpoints[0]));
25     Position<Vertex<V>> b = forest.find(positions.get(endpoints[1]));
26     if (a != b) {
27       tree.addLast(edge);
28       forest.union(a,b);
29     }
30   }
31
32   return tree;
33 }
```

Code Fragment 14.17: Java implementation of Kruskal's algorithm for the minimum spanning tree problem. The Partition class is discussed in Section 14.7.3.

14.7.3 Disjoint Partitions and Union-Find Structures

In this section, we consider a data structure for managing a ***partition*** of elements into a collection of disjoint sets. Our initial motivation is in support of Kruskal's minimum spanning tree algorithm, in which a forest of disjoint trees is maintained, with occasional merging of neighboring trees. More generally, the disjoint partition problem can be applied to various models of discrete growth.

We formalize the problem with the following model. A partition data structure manages a universe of elements that are organized into disjoint sets (that is, an element belongs to one and only one of these sets). Unlike with the Set ADT, we do not expect to be able to iterate through the contents of a set, nor to efficiently test whether a given set includes a given element. To avoid confusion with such notions of a set, we will refer to the sets of our partition as ***clusters***. However, we will not require an explicit structure for each cluster, instead allowing the organization of clusters to be implicit. To differentiate between one cluster and another, we assume that at any point in time, each cluster has a designated element that we refer to as the ***leader*** of the cluster.

Formally, we define the methods of a ***partition ADT*** using positions, each of which stores an element x. The partition ADT supports the following methods.

makeCluster(x): Creates a singleton cluster containing new element x and returns its position.

union(p, q): Merges the clusters containing positions p and q.

find(p): Returns the position of the leader of the cluster containing position p.

Sequence Implementation

A simple implementation of a partition with a total of n elements uses a collection of sequences, one for each cluster, where the sequence for a cluster A stores element positions. Each position object stores a reference to its associated element x, and a reference to the sequence storing p, since this sequence is representing the cluster containing p's element. (See Figure 14.25.)

With this representation, we can easily perform the makeCluster(x) and find(p) operations in $O(1)$ time, allowing the first position in a sequence to serve as the "leader." Operation union(p, q) requires that we join two sequences into one and update the cluster references of the positions in one of the two. We choose to implement this operation by removing all the positions from the sequence with smaller size, and inserting them in the sequence with larger size. Each time we take a position from the smaller cluster A and insert it into the larger cluster B, we update the cluster reference for that position to now point to B. Hence, the operation union(p, q) takes time $O(\min(n_p, n_q))$, where n_p (resp. n_q) is the cardinality of the

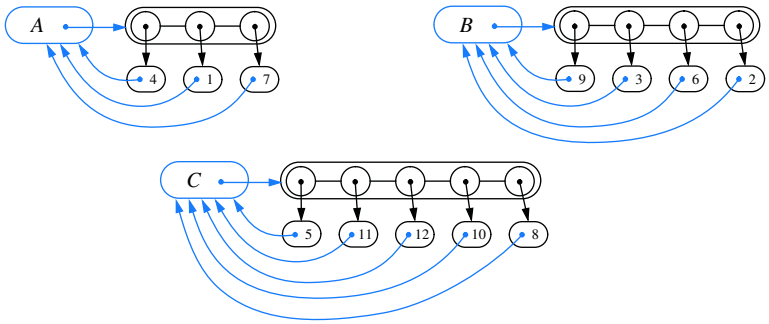


Figure 14.25: Sequence-based implementation of a partition consisting of three clusters: $A = \{1,4,7\}$, $B = \{2,3,6,9\}$, and $C = \{5,8,10,11,12\}$.

cluster containing position p (resp. q). Clearly, this time is $O(n)$ if there are n elements in the partition universe. However, we next present an amortized analysis that shows this implementation to be much better than appears from this worst-case analysis.

Proposition 14.26: *When using the sequence-based partition implementation, performing a series of k* makeCluster, union, *and* find *operations on an initially empty partition involving at most n elements takes $O(k + n\log n)$ time.*

Justification: We use the accounting method and assume that one cyber-dollar can pay for the time to perform a find operation, a makeCluster operation, or the movement of a position object from one sequence to another in a union operation. In the case of a find or makeCluster operation, we charge the operation itself 1 cyber-dollar. In the case of a union operation, we assume that 1 cyber-dollar pays for the constant-time work in comparing the sizes of the two sequences, and that we charge 1 cyber-dollar to each position that we move from the smaller cluster to the larger cluster. Clearly, the 1 cyber-dollar charged for each find and makeCluster operation, together with the first cyber-dollar collected for each union operation, accounts for a total of k cyber-dollars.

Consider, then, the number of charges made to positions on behalf of union operations. The important observation is that each time we move a position from one cluster to another, the size of that position's cluster at least doubles. Thus, each position is moved from one cluster to another at most $\log n$ times; hence, each position can be charged at most $O(\log n)$ times. Since we assume that the partition is initially empty, there are $O(n)$ different elements referenced in the given series of operations, which implies that the total time for moving elements during the union operations is $O(n\log n)$. ■

A Tree-Based Partition Implementation ★

An alternative data structure for representing a partition uses a collection of trees to store the n elements, where each tree is associated with a different cluster. In particular, we implement each tree with a linked data structure whose nodes serve as the position objects. (See Figure 14.26.) We view each position p as being a node having an instance variable, element, referring to its element x, and an instance variable, parent, referring to its parent node. By convention, if p is the ***root*** of its tree, we set p's parent reference to itself.

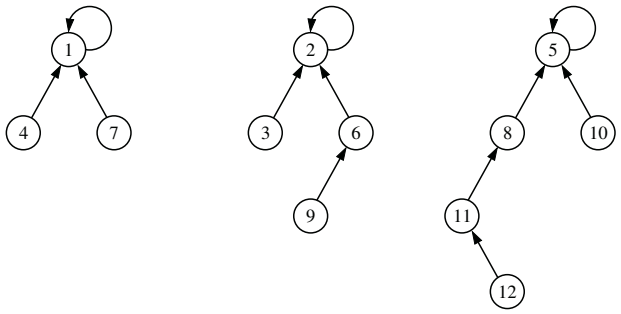


Figure 14.26: Tree-based implementation of a partition consisting of three clusters: $A = \{1,4,7\}$, $B = \{2,3,6,9\}$, and $C = \{5,8,10,11,12\}$.

With this partition data structure, operation find(p) is performed by walking up from position p to the root of its tree, which takes $O(n)$ time in the worst case. Operation union(p, q) can be implemented by making one of the trees a subtree of the other. This can be done by first locating the two roots, and then in $O(1)$ additional time by setting the parent reference of one root to point to the other root. See Figure 14.27 for an example of both operations.

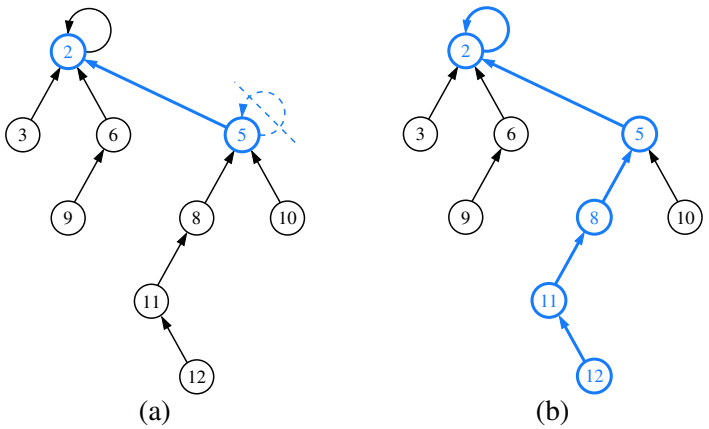


Figure 14.27: Tree-based implementation of a partition: (a) operation union(p, q); (b) operation find(p), where p denotes the position object for element 12.

At first, this implementation may seem to be no better than the sequence-based data structure, but we add the following two simple heuristics to make it run faster.

Union-by-Size: With each position p, store the number of elements in the subtree rooted at p. In a union operation, make the root of the smaller cluster become a child of the other root, and update the size field of the larger root.

Path Compression: In a find operation, for each position q that the find visits, reset the parent of q to the root. (See Figure 14.28.)

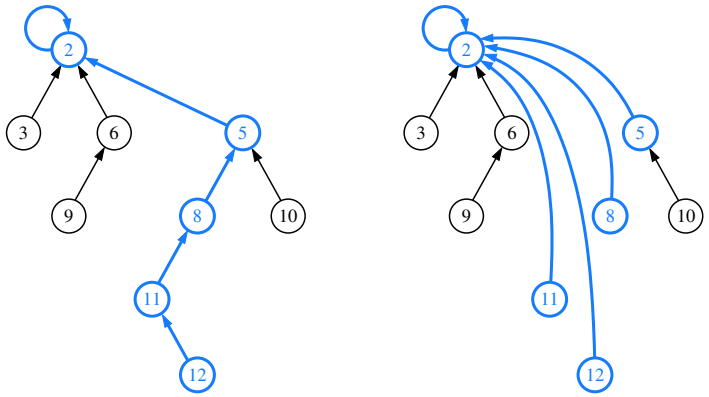


Figure 14.28: Path-compression heuristic: (a) path traversed by operation find on element 12; (b) restructured tree.

A surprising property of this data structure, when implemented using the union-by-size and path-compression heuristics, is that performing a series of k operations involving n elements takes $O(k \log^* n)$ time, where $\log^* n$ is the ***log-star*** function, which is the inverse of the ***tower-of-twos*** function. Intuitively, $\log^* n$ is the number of times that one can iteratively take the logarithm (base 2) of a number before getting a number smaller than 2. Table 14.4 shows a few sample values.

minimum n	2	$2^2 = 4$	$2^{2^2} = 16$	$2^{2^{2^2}} = 65,536$	$2^{2^{2^{2^2}}} = 2^{65,536}$
$\log^* n$	1	2	3	4	5

Table 14.4: Some values of $\log^* n$ and critical values for its inverse.

Proposition 14.27: *When using the tree-based partition representation with both union-by-size and path compression, performing a series of k* makeCluster, union, *and* find *operations on an initially empty partition involving at most n elements takes $O(k \log^* n)$ time.*

Although the analysis for this data structure is rather complex, its implementation is quite straightforward. We conclude with a Java implementation of the structure, given in Code Fragment 14.18.

```
1  /** A Union-Find structure for maintaining disjoint sets. */
2  public class Partition<E> {
3    //--------------- nested Locator class -------------
4    private class Locator<E> implements Position<E> {
5      public E element;
6      public int size;
7      public Locator<E> parent;
8      public Locator(E elem) {
9        element = elem;
10       size = 1;
11       parent = this;              // convention for a cluster leader
12     }
13     public E getElement( ) { return element; }
14   } //--------- end of nested Locator class ---------
15   /** Makes a new cluster containing element e and returns its position. */
16   public Position<E> makeCluster(E e) {
17     return new Locator<E>(e);
18   }
19   /**
20    * Finds the cluster containing the element identified by Position p
21    * and returns the Position of the cluster's leader.
22    */
23   public Position<E> find(Position<E> p) {
24     Locator<E> loc = validate(p);
25     if (loc.parent != loc)
26       loc.parent = (Locator<E>) find(loc.parent); // overwrite parent after recursion
27     return loc.parent;
28   }
29   /** Merges the clusters containing elements with positions p and q (if distinct). */
30   public void union(Position<E> p, Position<E> q) {
31     Locator<E> a = (Locator<E>) find(p);
32     Locator<E> b = (Locator<E>) find(q);
33     if (a != b)
34       if (a.size > b.size) {
35         b.parent = a;
36         a.size += b.size;
37       } else {
38         a.parent = b;
39         b.size += a.size;
40       }
41   }
42 }
```

Code Fragment 14.18: Java implementation of a Partition class using union-by-size and path compression. We omit the validate method due to space limitation.

14.8 Exercises

Reinforcement

R-14.1 Draw a simple undirected graph G that has 12 vertices, 18 edges, and 3 connected components.

R-14.2 If G is a simple undirected graph with 12 vertices and 3 connected components, what is the largest number of edges it might have?

R-14.3 Draw an adjacency matrix representation of the undirected graph shown in Figure 14.1.

R-14.4 Draw an adjacency list representation of the undirected graph shown in Figure 14.1.

R-14.5 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-14.6 Suppose we represent a graph G having n vertices and m edges with the edge list structure. Why, in this case, does the insertVertex method run in $O(1)$ time while the removeVertex method runs in $O(m)$ time?

R-14.7 Give pseudocode for performing the operation insertEdge(u, v, x) in $O(1)$ time using the adjacency matrix representation.

R-14.8 Repeat Exercise R-14.7 for the adjacency list representation, as described in the chapter.

R-14.9 Can edge list E be omitted from the adjacency matrix representation while still achieving the time bounds given in Table 14.1? Why or why not?

R-14.10 Can edge list E be omitted from the adjacency list representation while still achieving the time bounds given in Table 14.3? Why or why not?

R-14.11 Would you use the adjacency matrix structure or the adjacency list 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 getEdge(u, v) as fast as possible, no matter how much space you use.

R-14.12 In order to verify that all of its nontree edges are back edges, redraw the graph from Figure 14.8b so that the DFS tree edges are drawn with solid lines and oriented downward, as in a standard portrayal of a tree, and with all nontree edges drawn using dashed lines.

R-14.13 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-14.14 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-14.15 Recalling the definition of a complete graph from Exercise R-14.14, what does a breadth-first search tree of a complete graph look like?

R-14.16 Let G be an undirected 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-14.17 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

In what order can Bob take these courses, respecting the prerequisites?

R-14.18 Compute a topological ordering for the directed graph drawn with solid edges in Figure 14.3d.

R-14.19 Draw the transitive closure of the directed graph shown in Figure 14.2.

R-14.20 If the vertices of the graph from Figure 14.11 are ordered as (JFK, LAZ, MIA, BOS, ORD, SFO, DFW), in what order would edges be added to the transitive closure during the Floyd-Warshall algorithm?

R-14.21 How many edges are in the transitive closure of a graph that consists of a simple directed path of n vertices?

R-14.22 Given an n-node complete binary tree T, rooted at a given position, consider a directed graph $\vec{G}$ having the nodes of T as its vertices. For each parent-child pair in T, create a directed edge in $\vec{G}$ from the parent to the child. Show that the transitive closure of $\vec{G}$ has $O(n \log n)$ edges.

R-14.23 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-14.24 Show how to modify the pseudocode 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-14.25 Draw a simple, connected, undirected, weighted graph with 8 vertices and 16 edges, each with unique edge weights. Illustrate the execution of the Prim-Jarník algorithm for computing the minimum spanning tree of this graph.

R-14.26 Repeat the previous problem for Kruskal's algorithm.

R-14.27 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-14.28 Describe the meaning of the graphical conventions used in Figure 14.9 illustrating a DFS traversal. What do the line thicknesses signify? What do the arrows signify? How about dashed lines?

R-14.29 Repeat Exercise R-14.28 for Figure 14.8 that illustrates a directed DFS traversal.

R-14.30 Repeat Exercise R-14.28 for Figure 14.10 that illustrates a BFS traversal.

R-14.31 Repeat Exercise R-14.28 for Figure 14.11 illustrating the Floyd-Warshall algorithm.

R-14.32 Repeat Exercise R-14.28 for Figure 14.13 that illustrates the topological sorting algorithm.

R-14.33 Repeat Exercise R-14.28 for Figures 14.15 and 14.16 illustrating Dijkstra's algorithm.

R-14.34 Repeat Exercise R-14.28 for Figures 14.20 and 14.21 that illustrate the Prim-Jarník algorithm.

R-14.35 Repeat Exercise R-14.28 for Figures 14.22 through 14.24 that illustrate Kruskal's algorithm.

R-14.36 George claims he has a fast way to do path compression in a partition structure, starting at a position p. He puts p into a list L, and starts following parent pointers. Each time he encounters a new position, q, he adds q to L and updates the parent pointer of each node in L to point to q's parent. Show that George's algorithm runs in $\Omega(h^2)$ time on a path of length h.

Creativity

C-14.37 Give a Java implementation of the removeEdge(e) method for our adjacency map implementation of Section 14.2.5, making sure your implementation works for both directed and undirected graphs. Your method should run in $O(1)$ time.

C-14.38 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 getEdge(u, v) method. How are you implementing the method in this case?

C-14.39 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-14.40 Our solution to reporting a path from u to v in Code Fragment 14.6 could be made more efficient in practice if the DFS process ended as soon as v is discovered. Describe how to modify our code base to implement this optimization.

C-14.41 Let G be an undirected graph 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-14.42 Implement an algorithm that returns a cycle in a directed graph $\vec{G}$, if one exists.

C-14.43 Write a method, components(G), for undirected graph G, that returns a dictionary mapping each vertex to an integer that serves as an identifier for its connected component. That is, two vertices should be mapped to the same identifier if and only if they are in the same connected component.

C-14.44 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?

C-14.45 Computer networks should avoid single points of failure, that is, network vertices that can disconnect the network if they fail. We say an undirected, connected graph G is ***biconnected*** if it contains no vertex whose removal would divide G into two or more connected components. Give an 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. Your algorithm should run in $O(n+m)$ time.

C-14.46 Explain why all nontree edges are cross edges, with respect to a BFS tree constructed for an undirected graph.

C-14.47 Explain why there are no forward nontree edges with respect to a BFS tree constructed for a directed graph.

C-14.48 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-14.49 Justify Proposition 14.16.

C-14.50 Provide an implementation of the BFS algorithm that uses a FIFO queue, rather than a level-by-level formulation, to manage vertices that have been discovered until the time when their neighbors are considered.

C-14.51 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-14.52 An ***Euler tour*** of a directed graph $\vec{G}$ with n vertices and m edges is a cycle that traverses each edge of $\vec{G}$ exactly once according to its direction. Such a tour always exists if $\vec{G}$ is connected and the in-degree equals the out-degree of each vertex in $\vec{G}$. Describe an $O(n+m)$-time algorithm for finding an Euler tour of such a directed graph $\vec{G}$.

C-14.53 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 4. Suppose that RT&T's network is represented by a graph. Design an efficient algorithm that computes, for each station, the set of stations it can reach using no more than 4 links.

C-14.54 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 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 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-14.55 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 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 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 tree T, computes a center of T.
b. Is the center unique? If not, how many distinct centers can a tree have?

C-14.56 Say that an n-vertex directed acyclic graph $\vec{G}$ is ***compact*** if there is some way of numbering the vertices of $\vec{G}$ with the integers from 0 to $n-1$ such that $\vec{G}$ 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 $\vec{G}$ is compact.

C-14.57 Let $\vec{G}$ be a weighted directed graph 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-14.58 Design an efficient algorithm for finding a ***longest*** directed path from a vertex s to a vertex t of an acyclic weighted directed graph $\vec{G}$. Specify the graph representation used and any auxiliary data structures used. Also, analyze the time complexity of your algorithm.

C-14.59 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 method's running time?

C-14.60 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-14.61 Give an example of a weighted directed graph $\vec{G}$ with negative-weight edges, but no negative-weight cycle, such that Dijkstra's algorithm incorrectly computes the shortest-path distances from some start vertex s.

C-14.62 Our implementation of `shortestPathLengths` in Code Fragment 14.13 relies on use of "infinity" as a numeric value, to represent the distance bound for vertices that are not (yet) known to be reachable from the source. Reimplement that method without such a sentinel, so that vertices, other than the source, are not added to the priority queue until it is evident that they are reachable.

C-14.63 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 set *visited* 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 *visited*.
5: Add v to *path*.
6: Add v to *visited*.
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 counterexample.

C-14.64 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-14.65 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 (u,v) in E with u in C_i and v not in C_i.
    Add (u,v) to T (unless it is already in T).
return T

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

C-14.66 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.

C-14.67 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 equal to the lowest bandwidth among the path's edges. Give an algorithm that, given a network and two switching centers a and b, outputs the maximum bandwidth of a path between a and b.

C-14.68 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-14.69 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 each corridor, describe an algorithm to help Sir Paul pick up the most gold.

C-14.70 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 must be observed. What is the running time of your algorithm as a function of n and m?

C-14.71 Suppose we are given a directed graph $\vec{G}$ with n vertices, and let M be the $n \times n$ adjacency matrix corresponding to $\vec{G}$.

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 $\vec{G}$ is weighted and assume the following:

1: for $1 \leq i \leq n$, $M(i,i) = 0$.
2: for $1 \leq i, j \leq n$, $M(i,j) = \mathit{weight}(i,j)$ if (i,j) is in E.
3: 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-14.72 Karen has a new way to do path compression in a tree-based union/find partition data structure starting at a position p. She puts all the positions that are on the path from p to the root in a set S. Then she scans through S and sets the parent pointer of each position in S to its parent's parent pointer (recall that the parent pointer of the root points to itself). If this pass changed the value of any position's parent pointer, then she repeats this process, and goes on repeating this process until she makes a scan through S that does not change any position's parent value. Show that Karen's algorithm is correct and analyze its running time for a path of length h.

Projects

P-14.73 Use an adjacency matrix to implement a class supporting a simplified graph ADT that does not include update methods. Your class should include a constructor method that takes two collections—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-14.74 Implement the simplified graph ADT described in Exercise P-14.73, using the edge list structure.

P-14.75 Implement the simplified graph ADT described in Exercise P-14.73, using the adjacency list structure.

P-14.76 Extend the class of Exercise P-14.75 to support the update methods of the graph ADT.

P-14.77 Design an experimental comparison of repeated DFS traversals versus the Floyd-Warshall algorithm for computing the transitive closure of a directed graph.

P-14.78 Develop a Java implementation of the Prim-Jarník algorithm for computing the minimum spanning tree of a graph.

P-14.79 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-14.80 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-14.81 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, 87] are the ones who showed how useful this algorithm is for solving several different graph problems. Knuth [60] 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 [34] and is based upon a theorem of Warshall [94].

The first known minimum spanning tree algorithm is due to Barůvka [9], and was published in 1926. The Prim-Jarník algorithm was first published in Czech by Jarník [51] in 1930 and in English in 1957 by Prim [79]. Kruskal published his minimum spanning tree algorithm in 1956 [63]. 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 method of Karger, Klein, and Tarjan [53] that runs in $O(m)$ expected time. Dijkstra [30] published his single-source, shortest-path algorithm in 1959. 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" [36] or the "Relaxed Heap" [32].

To learn about different algorithms for drawing graphs, please see the book chapter by Tamassia and Liotta [85] and the book by Di Battista, Eades, Tamassia and Tollis [29]. The reader interested in further study of graph algorithms is referred to the books by Ahuja, Magnanti, and Orlin [7], Cormen, Leiserson, Rivest and Stein [25], Mehlhorn [72], and Tarjan [88], and the book chapter by van Leeuwen [90].