# The Graph Abstract Data Type 13

## CHAPTER OUTLINE



As suggested in Chapter 10 when we introduced the Tree ADT, a graph is a more general instance of a tree. Indeed, we can say that a graph is the most general of all the collections we have seen because unlike the other collections presented, graphs allow arbitrary relationships to exist among their elements.

As you might expect, graphs have many applications: from determining the shortest flight time between a collection of cities, to identifying an activity graph for the construction of a building, to identifying connections on a circuit board or computer network.

This chapter provides an introduction to graphs, their representation as data structures, and some important graph algorithms. The implementation of the Graph ADT provides some new and interesting challenges.

# 13.1 Graph Terminology

A graph consists of a collection of vertices and edges that connect vertices

A **graph** G = (V, E) is an ordered pair of finite sets V and E, where V(G) is the set of vertices in G, and E(G) is the set of edges in G. The edge connecting vertices $v_i$ and $v_j$ is represented as $(v_i, v_j)$. The number of vertices in G is given as |V| and the number of edges in G is given as |E|.

Edges in an undirected graph are bidirectional

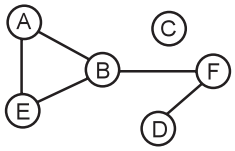
The edges in an **undirected graph** are bidirectional; therefore, they can be traversed in both directions. Graph $G_1$ shown in Figure 13.1 (a) is undirected; the edge connecting vertices B and E allows you to traverse from B to E and from E to B. The edges in a **directed graph** (also called a **digraph**) are unidirectional from the source vertex to the destination vertex. Graph $G_2$ shown in Figure 13.1 (b) allows traversal from T to Z (note the direction of the arrow on the edge), but not from Z to T. Similarly, you can travel from Z to R, but not from R to Z. Bidirectionality between adjacent vertices in a digraph is achieved by providing a pair of edges. Digraph $G_3$ shown in Figure 13.1 (c) allows the same connectivity as the undirected graph $G_1$. Note that this is easily accomplished by turning every undirected edge into a pair of directed edges traveling in opposite directions. We will disallow a **loop**, also called a **self-edge**, which is an edge from a vertex to itself.

Edges in a directed graph are directional

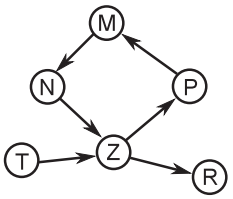
A weighted graph associates a non-negative value with each edge

A **weighted graph** associates a non-negative value with each edge. This weight could represent, for example, the cost in time or money to traverse the edge, or the distance between the adjacent vertices. Graph $G_4$ shown in Figure 13.1 (d) has weighted edges. The algorithms we will examine for weighted graphs assume that the weights are non-negative.

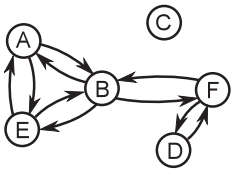
**Figure 13.1** Sample undirected, directed, and weighted graphs.



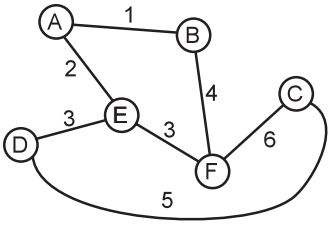

(a) Graph $G_1$—an undirected graph.




(b) Graph $G_2$—a directed graph.




(c) Graph $G_3$—a directed version of graph $G_1$.




(d) Graph $G_4$—an undirected graph with weighted edges.

Note that vertex C in $G_1$ is not connected to any other vertex and is therefore unreachable from another vertex in the graph.

Two vertices are **adjacent** if they are connected by an edge. An edge $(v_i, v_j)$ is said to be **incident on** vertices $v_i$ and $v_j$. The **in-degree** of a vertex is the number of edges coming into it; the **out-degree** of a vertex is the number of edges leaving it. In an undirected graph, a vertex's in-degree and out-degree are the same, so we usually refer to the degree of a vertex, without the "in/out." By contrast, the in- and out-degree of a vertex in a digraph may be different. For example, in $G_1$ the degree of B is 3 and the degree of C is 0. In the directed graph $G_2$, the in-degree of Z is 2 and the out-degree is also 2, while the in-degree of R is 1 and the out-degree is 0.

A **path** in a graph is a sequence of adjacent vertices. The length of the path is the number of edges it contains. In $G_1$ there is a path from A to D of length 3. There is no path from A to C. $G_2$ has a path of length 2 from N to R. If the edges are weighted, we talk about the **cost** of the path, which is the sum of the edge weights in the path. In $G_4$ there are two paths from D to B: {D, E, F, B} with a cost of 10, and {D, E, A, B} with a cost of 6. A **simple path** is a path in which each vertex appears only once. The path from A to D in $G_1$ is an example, as is the path from A to F.

A **cycle** is a path that begins and ends with the same vertex. The path {A, B, E, A} in $G_1$ is an example. There are no other cycles in $G_1$.

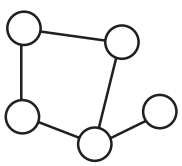
In $G_2$ {T, Z, P, M} and {T, Z, R} are simple paths. Vertices {N, Z, P, M, N} form a cycle. Note that there is no path from R to any other vertex in the graph, but R is reachable from every other vertex.

Two vertices $v_i$ and $v_j$ in graph G are said to be **connected** if there is a path in G from $v_i$ to $v_j$. We also say that $v_j$ is **reachable** from $v_i$. If G is undirected, this implies that there is also a path from $v_j$ to $v_i$ and that $v_i$ is reachable from $v_j$.

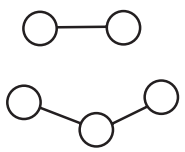
A **connected graph**, G, is one in which every vertex in G is reachable from (connected to) every other vertex in G. A **complete graph** is a connected graph in which each pair of vertices is connected by an edge. There are $n\ (n-1)\ /\ 2$ edges in a complete undirected graph and at most $n\ (n-1)$ edges in a complete directed graph. Figure 13.2 shows some examples.

A **subgraph** of G is a graph containing some or all of the vertices and edges from G. More formally, G′ is a subgraph of G such that $V(G') \subseteq V(G)$ and $E(G') \subseteq E(G)$. Figure 13.3 shows some examples.

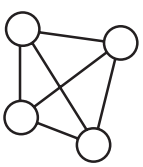
**Figure 13.2** Connected, disconnected, and complete graphs.



(a) A connected graph.

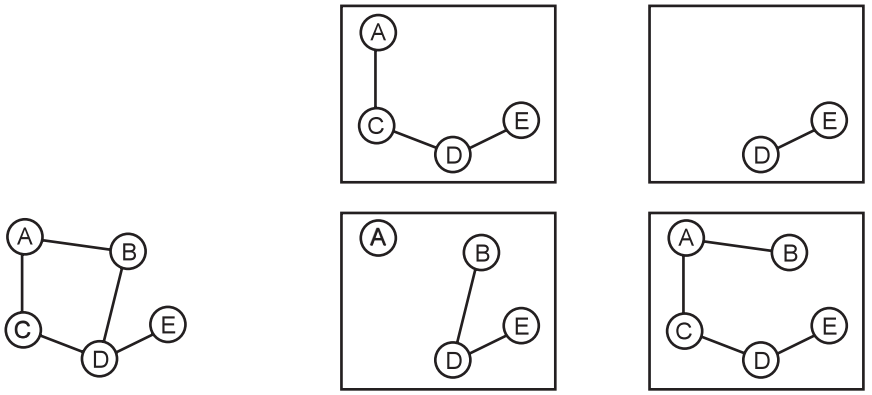


(b) A disconnected graph.



(c) A complete graph.

**Figure 13.3** A graph $G_5$ and some of its subgraphs.




(a) Graph $G_5$.

(b) Some subgraphs of $G_5$.

## SUMMING UP

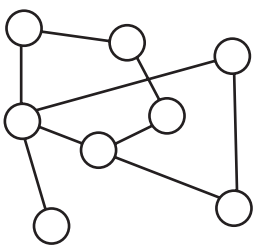
A graph is a collection of vertices and edges connecting vertices. In an undirected graph the edges are bidirectional, and in a directed graph the edges are directional from source to destination vertex. The edges in a weighted graph have an associated weight (cost).

## CHECKPOINT

**13.1** Give an example of the following:

a. A connected undirected graph containing five vertices
b. An unconnected digraph containing five vertices
c. A directed graph of five vertices containing a cycle
d. An undirected acyclic graph

**13.2** Give two subgraphs for the following graph:



**13.3** Provide the equivalent directed graph for graph $G_5$.

# 13.2 Representations of Graphs

There are two common representations of graphs: adjacency matrices and adjacency lists. They have different space complexities, which has implications for the time complexities of graph algorithms, as we will see later.

## 13.2.1 Adjacency Matrix Representation

**Adjacency matrix: a $|V| \times |V|$ array of integer**

The idea behind the adjacency matrix representation is very simple. An adjacency matrix is a $|V| \times |V|$ array of integer. A matrix entry [i, j] is 1 if the graph contains an edge (i, j), and is 0 otherwise. If the graph is undirected, the entry [j, i] will also be 1 (recall that edges in an undirected graph are bidirectional). If the edges are weighted, the [i, j] entry for an edge connecting $v_i$ to $v_j$ can be the weight of that edge (in this case we might want to make the matrix an array of double). A special value indicating that no edge exists must be selected for weighted graphs.

Figure 13.4 (a) shows an adjacency matrix of an undirected graph. Note that it is symmetric around the diagonal. Figure 13.4 (b) shows the adjacency matrix of a directed graph. Note that the adjacency matrix of a directed graph does not need to be symmetric around the diagonal. Figure 13.4 (c) gives the adjacency matrix of a weighted graph in which a value of 0 indicates no connection.

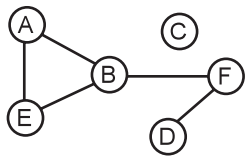
**Adjacency matrix space complexity: $\Theta(|V|^2)$**

The space complexity of an adjacency matrix is always $\Theta(|V|^2)$ because it maintains an entry for each *possible* edge.

## 13.2.2 Adjacency List Representation

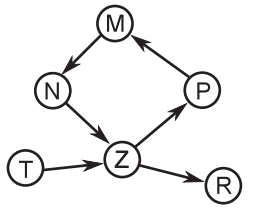
If the graph is large and sparse (the number of actual edges in the graph is much smaller than the number of possible edges), then the adjacency matrix representation is inefficient in its space usage (most of the entries are 0).

**Figure 13.4** Adjacency matrices of undirected and directed graphs.



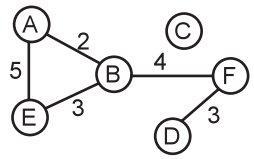

| | A | B | C | D | E | F |
|---|---|---|---|---|---|---|
| A | 0 | 1 | 0 | 0 | 1 | 0 |
| B | 1 | 0 | 0 | 0 | 1 | 1 |
| C | 0 | 0 | 0 | 0 | 0 | 0 |
| D | 0 | 0 | 0 | 0 | 0 | 1 |
| E | 1 | 1 | 0 | 0 | 0 | 0 |
| F | 0 | 1 | 0 | 1 | 0 | 0 |

(a) Adjacency matrix of an undirected graph.




| | M | N | P | R | T | Z |
|---|---|---|---|---|---|---|
| M | 0 | 1 | 0 | 0 | 0 | 0 |
| N | 0 | 0 | 0 | 0 | 0 | 1 |
| P | 1 | 0 | 0 | 0 | 0 | 0 |
| R | 0 | 0 | 0 | 0 | 0 | 0 |
| T | 0 | 0 | 0 | 0 | 0 | 1 |
| Z | 0 | 0 | 1 | 1 | 0 | 0 |

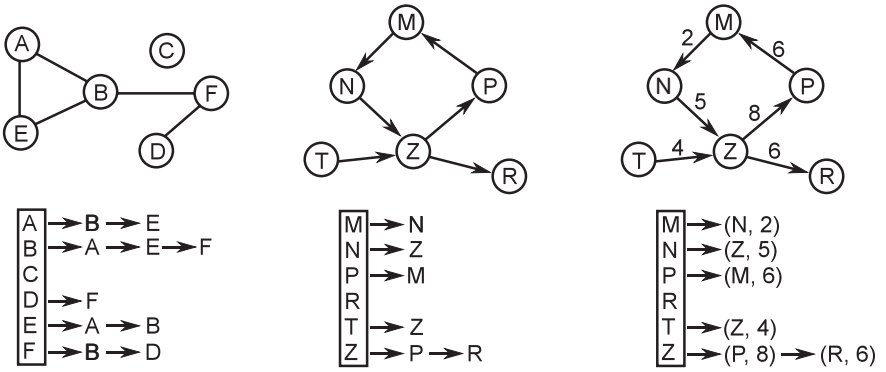
(b) Adjacency matrix of a directed graph.




| | A | B | C | D | E | F |
|---|---|---|---|---|---|---|
| A | 0 | 2 | 0 | 0 | 5 | 0 |
| B | 2 | 0 | 0 | 0 | 3 | 4 |
| C | 0 | 0 | 0 | 0 | 0 | 0 |
| D | 0 | 0 | 0 | 0 | 0 | 3 |
| E | 5 | 3 | 0 | 0 | 0 | 0 |
| F | 0 | 4 | 0 | 3 | 0 | 0 |

(c) Adjacency matrix of an undirected weighted graph.

**Figure 13.5** Adjacency lists of undirected and directed graphs.




(a) Adjacency list of an undirected graph.

(b) Adjacency list of a directed graph.

(c) Adjacency list of a directed weighted graph.

In this situation, we would prefer to store information about the edges that are actually in the graph. Unlike an adjacency matrix, this is precisely what the **adjacency list** representation allows us to do. As shown in Figure 13.5, each vertex has an associated linked list of nodes, each of which stores information about an adjacent vertex. If the edges are weighted, the nodes must store ⟨vertex, weight⟩ pairs.

**Adjacency list: a |V| array of linked lists**

We need a container of size $\Theta(|V|)$ to store the vertices, and there will be $\Theta(|E|)$ elements in the adjacency lists for a total cost of $\max(\Theta(|V|), \Theta(|E|))$. Of course, what we save in space we give up in the time needed to traverse the adjacency lists searching for a vertex—a lesson we learned in Chapter 3.

**Adjacency list space complexity: $\max(\Theta(|V|), \Theta(|E|))$**

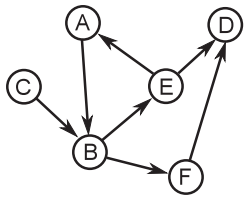
## SUMMING UP

There are two common representations of graphs: adjacency matrices and adjacency lists. The adjacency matrix representation stores an entry for every possible edge in a graph; thus, it has a space complexity of $\Theta(|V|^2)$. The adjacency list stores entries for edges actually in the graph; thus, its space complexity is $\max(\Theta(|V|), \Theta(|E|))$. The representation used has implications for some graph algorithms.

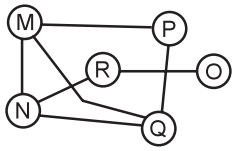
## CHECKPOINT

**13.4** Provide the adjacency list for graph $G_6$.

**13.5** Provide the adjacency matrix for graph $G_7$.




Graph $G_6$.




Graph $G_7$.

**13.6** Explain why the space complexity for the adjacency matrix representation is $\Theta(|V|^2)$.

**13.7** Explain why the space complexity for the adjacency list representation is $\Theta \max(\Theta(|V|), \Theta(|E|))$.

## ■ 13.3 Graph Algorithms: Traversals

There are two general strategies for traversing (searching) a graph: depth first search (DFS) and breadth first search (BFS).

### 13.3.1 Depth First Search

**Depth first search: visits descendants before visiting siblings**

DFS behaves similarly to a preorder traversal of a binary tree. In the language of tree traversals, descendants of a node are visited before the node's siblings. It is the same with graphs, except that in a graph we need to deal with the possibility of cycles.

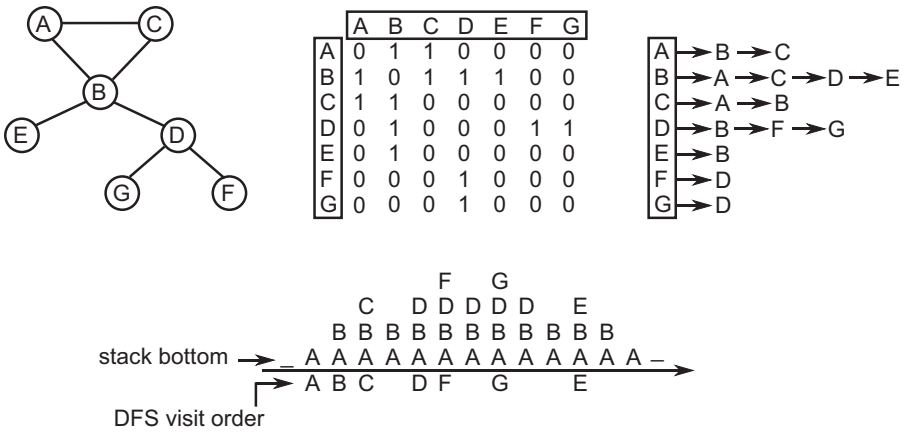
As shown in the pseudocode for DFS, the algorithm is naturally recursive (as with the preorder traversal for binary trees). The runtime stack stores vertices that still need to be explored and guarantees that we go as deep in the graph as we can (visiting a node's descendants) before spreading out (visiting a node's siblings).

```
// Perform a depth first search of a Graph g
// originating from Vertex v

Pseudocode: DFS ( Graph g, Vertex v )
mark v as visited     // don't need to visit a vertex more than once!
for each Vertex w adjacent to v
   if w has not been visited
      DFS( w )
```

Next we discuss an aspect of DFS that we will see repeated in our coverage of graph algorithms. An important difference between a tree and a graph is that a graph can contain cycles and a tree cannot. This is important for graph algorithms because we don't want to visit a vertex more than once, and of course, we want to avoid the embarrassment of getting caught in an infinite loop going round and round the vertices in a cycle. The solution is somehow to mark a vertex as *visited* as done in the first step of the pseudocode. A common solution is to use colors: white

**Figure 13.6** Contents of the queue during a depth first search.




indicates not yet visited and black indicates visited. It's useful to have a third color (gray) to indicate a state somewhere between visited and not visited. Figure 13.6 shows the order in which the vertices are visited and the vertices on the method call stack as the graph is searched, starting with vertex A.

What is DFS's time complexity? Since DFS follows available edges, intuition tells us that the cost is tied to the number of edges in the graph. However, the underlying graph representation plays an important role here. If the graph is represented as an adjacency matrix, step 2 of the pseudocode (*for each Vertex w adjacent to v*) has a cost of $\Theta(|V|)$ since we need to examine every column in a row to see if the entry corresponds to an adjacent vertex. We must do this for all $|V|$ rows in the matrix, giving a cost of $\Theta(|V|^2)$ for DFS for the adjacency matrix representation. Significantly, this is independent of the actual number of edges in the graph.

**DFS time complexity—adjacency matrix: $\Theta(|V|^2)$ adjacency list: $O(|V|^2)$**

This highlights the advantage of the adjacency list representation for some algorithms because an adjacency list contains only edges that are actually in the graph. To get a feel for the difference, consider the adjacency entries for vertex F shown in Figure 13.6. In the adjacency matrix representation you need to look at all seven columns to determine which vertices are adjacent to F and which are not, but in the adjacency list representation you need to examine only the one vertex to which F is adjacent.

In the best case there are no edges and the cost is $O(1)$ since the starting node will have no adjacent vertices and the algorithm will quit. In the worst case, the graph is fully connected and the cost is $\Theta(|V|^2)$. In the absence of information about the connectivity of a graph, the cost of DFS for a graph when stored as an adjacency list is $O(|V|^2)$. (Note the use of big O here.)

## 13.3.2 Breadth First Search

**Breadth first search: visits children before visiting grandchildren**

BFS is similar to a level order traversal of a tree: it visits a node's children before visiting any of their children. In graph terms this translates to a visit pattern that

spreads out in waves from the start vertex; the first wave is one edge away from the start vertex; the second wave is two edges away from the start vertex, and so on, as shown in the top left of Figure 13.7.

As shown in the following pseudocode, BFS uses a queue to store vertices still to be explored. This guarantees that those vertices that are enqueued first (those nearest to the start vertex) will be visited before those enqueued later (those farther from the start vertex). Figure 13.7 shows the contents of the queue as the graph is searched starting from vertex A.

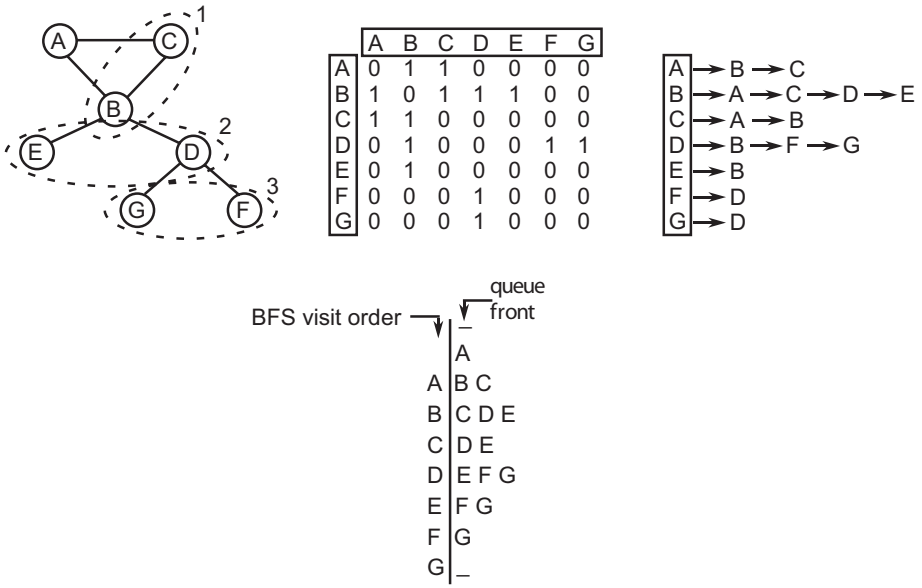
**The time complexity of BFS is the same as DFS**

The cost of BFS is the same as DFS and for similar reasons.

```
// Perform a breadth first search of a Graph g
// originating from Vertex v

Pseudocode: BFS ( Graph g, Vertex v ) vertexQueue—a Queue of Vertices
mark v as visited
enqueue v in vertexQueue
while vertexQueue is not empty
   let v be the element removed from the front of vertexQueue
      for all Vertices w adjacent to v
         if w has not been visited
            enqueue w in vertexQueue
            mark w as visited
```

**Figure 13.7** Contents of the queue during a breadth first search. The vertices grouped by the dotted ovals represent their distance from the start vertex.



## SUMMING UP

Depth first search (DFS) and breadth first search (BFS) are common graph traversal algorithms that are similar to some tree traversal algorithms. The important difference is that DFS and BFS must deal with the possibility of a cycle in a graph. DFS is similar to a preorder traversal of a tree: a vertex is visited, and then its descendants are visited before its siblings are visited. DFS naturally uses a stack to store vertices yet to be visited. BFS is similar to a level order traversal, visiting a node and all its children before extending to grandchildren. BFS naturally uses a queue to store vertices yet to be visited.

## CHECKPOINT

**13.8** Show the order in which the vertices in graph $G_8$ would be visited using the following:

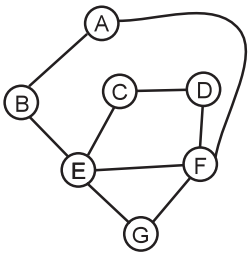
a. DFS starting at vertex A

b. BFS starting at vertex A

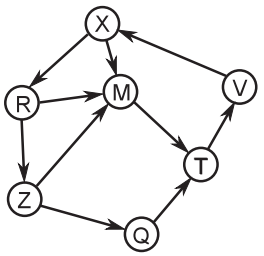
**13.9** Show the order in which the vertices in graph $G_9$ would be visited using the following:

a. DFS starting at vertex X

b. BFS starting at vertex X




Graph $G_8$.




Graph $G_9$.

## INVESTIGATE Applying BFS and DFS—Shortest Path and Spanning Tree Algorithms

BFS and DFS are the basis for a number of important graph algorithms. In this *Investigate* you develop two of those algorithms, shortest path and spanning tree, while further investigating the similarities and differences between BFS and DFS, and directed and undirected graphs.

### Shortest Path Algorithm

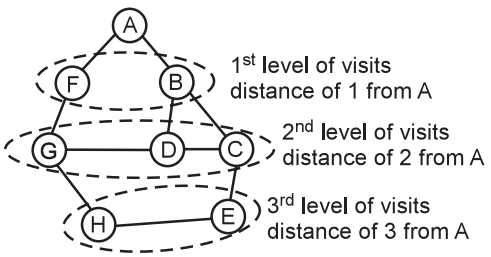
As its name suggests, the shortest path algorithm finds the shortest path in a graph between a source vertex and a destination vertex, as measured by the number of

edges in the path. A little thought about the behavior of BFS will tell you that it can find the shortest path rather easily.

1. Enter the pseudocode for BFS into a text file or as comments in a file opened in your IDE. You will gradually modify the pseudocode to provide the following operation:

   **List shortestPath( Graph g, Vertex src, Vertex dest )**
   *// Find a shortest path from src to dest in Graph g. Return an empty List if no path*
   *// exists, or the vertices in the path from src to dest otherwise.*
2. Produce the adjacency matrix or adjacency list for graph $G_{10}$ in Figure 13.8.
3. Figure 13.8 shows the levels visited by BFS starting at vertex A. Provide subscripts for the vertices showing the order in which they are visited by BFS, assuming that adjacent vertices are visited in alphabetical order.
4. As you can see, if the destination is reachable from the source vertex, the algorithm eventually finds the destination vertex. When should the modified BFS algorithm stop searching? Make this change to the pseudocode.
5. Clearly, if successful, we will end up with a path from *src* to *dest*. As vertices are visited, a path is built *from src toward dest*. We need to keep track of each vertex's predecessor in the path. This can easily be done by storing a pair $\langle predecessor, v \rangle$, where $v$ is a new vertex found in some possible path to the destination, and *predecessor* is $v$'s predecessor in that path. For example, we see A has two adjacent vertices that could be on a path to E, so we would add them to our collection as $\langle A, F \rangle$ and $\langle A, B \rangle$.
   a. What kind of collection is appropriate to store these pairs? On what basis do you decide what is appropriate?
   b. Modify the algorithm to collect these pairs.
6. Apply your algorithm to the graph shown in Figure 13.8 looking for a path from A to E. Be sure to show all vertex pairs collected.
7. In step 4 you determined when the search part of the algorithm should stop. Assuming the destination vertex was found, we now need to reconstruct the path from the source to the destination. The algorithm cannot know in advance

**Figure 13.8** The levels of vertices visited by BFS starting at vertex A.




Graph $G_{10}$.

which pairs of adjacent vertices are in the path from the source to the destination, and the code trace you completed in step 6 will have revealed that some vertex pairs may have been collected that are not in the path. How do we pull out just the pairs on the path we want? The answer is pretty straightforward. We will have arrived at the *destination* vertex from some other vertex, *v*, in the graph, giving us a pair ⟨*v, destination*⟩ in our collection. For example, we might have arrived at E from C, giving us a pair ⟨*C, E*⟩. What we need to do is to find a pair in our collection that has *v* as the successor to some vertex *w* (that is, a pair ⟨*w, v*⟩). We keep doing this until we get the pair ⟨*source, v*⟩. Really what we are doing is tracing the path *backward* from the destination to the source.

a. Make this addition to the pseudocode so that a list is returned containing the vertices in the path from the source to the destination vertex. For the graph shown in Figure 13.8, this would be {A, B, C, E}.

b. How can we determine the length of this path?

c. Provide an argument for why your algorithm is correct. How is it guaranteed to find the shortest path?

**8.** Create another graph and apply your pseudocode algorithm to it. If the result isn't correct, modify the pseudocode and reapply it to the graph. If any of your pseudocoded instructions are not clear, rephrase them to better describe their purpose.

**9.** What is the time complexity of this algorithm assuming the graph is represented using an adjacency matrix? An adjacency list?

**10.** Will this algorithm work for a directed graph? If so, explain why. If not, provide a counter example.

**11.** Will this algorithm work for a weighted graph? If so, explain why. If not, provide a counter example.

**12.** Could you modify DFS to give a shortest path? If so, outline the changes. If not, explain why not.

## Spanning Tree Algorithm

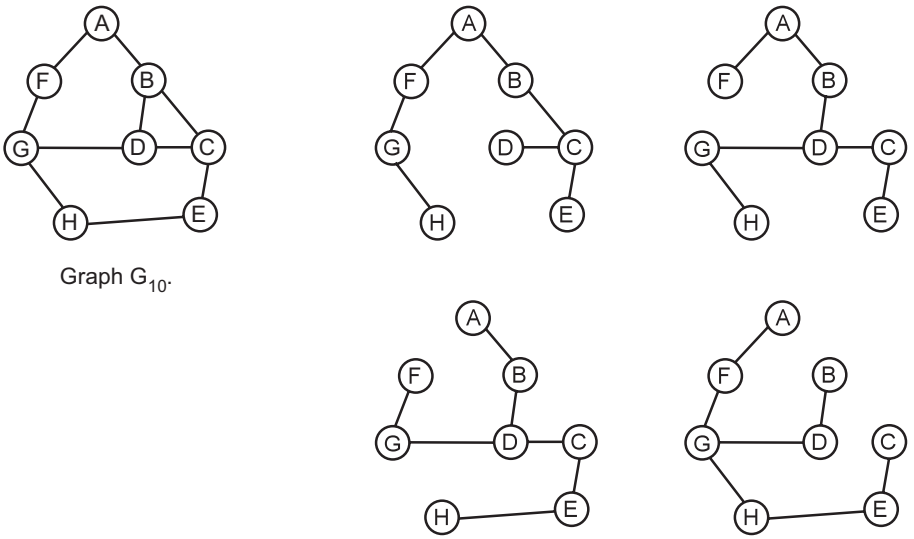
A spanning tree of a graph G is a connected acyclic subgraph, G′, containing all the vertices of G; that is, $V(G') = V(G)$. Since G′ is connected and acyclic, it must contain the minimum number of edges possible ($|V| - 1$). Figure 13.9 shows graph $G_{10}$ from Figure 13.8 and four possible spanning trees for it.

As we have seen, DFS will visit all the vertices of a connected graph just once. If we modify DFS to keep track of connected pairs of vertices, it will produce a spanning tree.

**1.** Enter the pseudocode for DFS into a text file or as comments in a file opened in your IDE. You will gradually modify the pseudocode to provide the following operation:

**Graph spanningTree( Graph g, Vertex src )**
*// Return a spanning tree of g using src as the root of the tree. If the returned*
*// graph is empty, g is not connected.*

**Figure 13.9** Four possible spanning trees for graph $G_{10}$ from Figure 13.8.




Graph $G_{10}$.

2. Apply the DFS algorithm to graph $G_{10}$ and provide subscripts for the vertices showing the order in which they are visited, assuming that adjacent vertices are visited in alphabetical order.
   a. Given that we want to identify a spanning tree for the graph, when should the DFS algorithm stop searching? Make this change to the pseudocode.
3. If successful, we will visit all the vertices in the graph once. As vertices are visited, we need to keep track of each vertex's predecessor. This can easily be done with a pair ⟨*predecessor*, *v*⟩, where *v* is a newly visited vertex and *predecessor* is *v*'s predecessor in the graph.
   a. What kind of collection is appropriate to store these pairs? On what basis do you decide what is appropriate?
   b. Modify the algorithm to collect these pairs.
4. In step 2 we determined when the search part of the algorithm should stop. Now we need to reconstruct the spanning tree. This can be done using the pairs collected in step 3. Make this change to the algorithm. The graph created is returned as the result of the method.
5. Create another graph and apply your pseudocode algorithm to it. If the result isn't correct, modify the pseudocode and reapply it to the graph. If any of your pseudocoded instructions are not clear, rephrase them to better describe their purpose.
6. What is the time complexity of this algorithm assuming the graph is represented using an adjacency matrix? An adjacency list?
7. Will this algorithm work for a directed graph? If so, explain why. If not, provide a counter example.

8. Will this algorithm work for a weighted graph? If so, explain why. If not, provide a counter example.
9. Could you modify BFS to provide a spanning tree? If so, outline the changes. If not, explain why not.

## 13.4 Graph Algorithms: Minimum Cost Paths and Minimal Spanning Trees

The shortest path and spanning tree algorithms you developed in the *Investigate* are insufficient if the edges have different weights (costs) associated with them. In this section we present a minimum cost path algorithm based on a solution developed by E.W. Dijkstra[1] and an algorithm for finding a minimal spanning tree based on a solution credited to R.C. Prim.[2] These algorithms use a greedy strategy to find their solutions; that is, at each step, a **greedy algorithm** makes a locally optimal (greedy) selection to produce a globally optimal solution.

**Greedy algorithms produce optimal solutions by making locally optimal selections**

### 13.4.1 Minimum Cost Paths

We want to find a minimum cost path in a weighted graph from a source vertex to a destination vertex. As we did with our modified BFS algorithm for shortest path in the *Investigate*, our approach here is to build a path gradually from the source toward the destination one edge at a time. The important difference is that at each step in extending the path we are building, we will always select the vertex whose path length from the source vertex is the *minimum* of *all* paths seen so far (this is the greedy selection part). We will store this information in a pair ⟨*Vertex, minCostToVertex*⟩ where *minCostToVertex* is the total cost of a path from the source vertex to *Vertex*. Since we always want to work from the minimum path seen so far, we'll keep our pairs in a priority queue using the *minCostToVertex* field to determine priority. As shown in the pseudocode, we start with the *src* vertex, which has a *minCostToVertex* of 0 (that is, the cost to get from the *src* vertex to *src* is 0).

*// Return the cost of a minimal cost path in*
*// Graph g from Vertex src to dest*

**Pseudocode: Minimal Path( Graph g, Vertex src, Vertex dest )**

priorityQueue—a priority queue holding ⟨Vertex, minCostToVertex⟩ pairs
verticesInSomePathToDest—a collection of vertices in some possible path from src to dest

*place ⟨src, 0⟩ in priorityQueue*
*while priorityQueue is not empty*
  *// get the least expensive path seen so far*

[1]E.W. Dijkstra, "A Note on Two Problems in Connection with Graphs," *Numerische Mathematik*, 1(1959), pp. 269–271.

[2]R.C. Prim, "Shortest Connection Networks and Some Generalizations," *Bell System Technical Journal*, 36(1957), pp. 1389–1401.

```
pair = the ⟨vertex, minCostToVertex⟩ pair removed from priorityQueue
v  = pair.vertex                                  // extract the vertex field from pair
minCostToV = pair. minCostToVertex                // extract the cost field from pair
if v == dest
   return minCostToV                              // success; return cost from src to v

// haven't found the target yet, so continue the search
add v to verticesInSomePathToDest
for each Vertex w adjacent to v
   if w is not in verticesInSomePathToDest        // avoid revisiting a visited vertex
      let minCostToW be minCostToV + weight(v, w) // get total cost from src to w
      add the pair ⟨w, minCostToW⟩ to priorityQueue
return failure                                    // there is no path from src to dest
```

The annotated example shown in Figure 13.10 (next page) illustrates the algorithm's behavior and provides an intuitive sense for why it works. As you can see from the example, the algorithm is actually building several *possible* paths. After four steps, for example, we have {A, F} and {A, B} (look at the shaded vertices in the graphs); after five steps we have {A, F}, and {A, B, C}; and after six steps there are three possibilities under consideration: {A, F}, {A, B, D}, and {A, B, C}.

Since a path is always extended to the least cost adjacent vertex, every ⟨*Vertex, minCostToVertex*⟩ pair pulled from the priority queue represents the minimal cost of a path from *src* to Vertex. Consequently, when a pair is pulled from the priority queue that has *dest* as its Vertex, the minimal cost of a path from *src* to *dest* has been found and the algorithm can terminate.

### 13.4.2 Minimal Spanning Tree

From the *Investigate* you learned that a spanning tree of a graph G is a connected acyclic subgraph containing all the vertices of G and the minimum number of edges possible ($|V| - 1$). A **minimal spanning tree (MST)** of a weighted graph G is a spanning tree with a minimum total path cost over all possible spanning trees of G.

The DFS-based approach taken in the *Investigate* to produce a spanning tree will not produce a *minimal* spanning tree for a graph whose edges have different weights (costs). However, just as we were able to modify the BFS algorithm to produce a spanning tree, we can base a minimal spanning tree algorithm on the ideas behind the minimal path algorithm. The algorithm presented here is called Prim's algorithm. Like the minimal path algorithm, Prim's algorithm follows a greedy strategy.

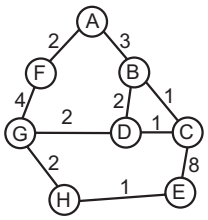
The main idea behind Prim's algorithm is quite simple and will look familiar to you from the minimal path algorithm. As with the minimal path, we build our MST incrementally using a greedy strategy to pick a least cost edge from a set of *candidate edges*. Here is how it works. To find an MST for a weighted graph G, we start by adding the source vertex, *v*, to the MST (this will be the root of the MST). The candidate edges are those edges *(v, w)* such that *v* is in our MST and *w* is not. So the first edges to add to the candidate edges are those incident on the source vertex. At each iteration of the main loop we choose the least cost edge *(v, w)* from the candidate edges. By adding *(v, w)* to our MST, we are extending it by one more least cost

**Figure 13.10** Annotated example of the minimal path algorithm searching for a path from A to E.

v A vertex in a possible path from src to dest.

Vertex represents the vertex extracted from Priority Queue. The associated path cost is the minimum path cost from src to this Vertex.

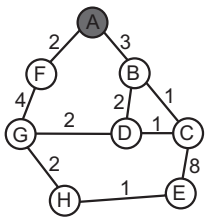
Priority Queue holds (vertex, path cost) pairs and is ordered such that the pair with the lowest path cost has the highest priority (shown as boldface and underlined). It is assumed that duplicate pairs are not allowed, so (D, 5) and (D, 5) would not both appear in Priority Queue, but (D, 5) and (D, 8) could.




Vertex: none

Priority Queue:
**(A, 0)**

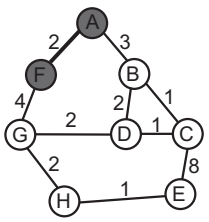
The search for a minimal cost path begins with the source vertex, A, which has a distance of 0 from itself, so (A, 0) is placed in Priority Queue.




Vertex: A

Priority Queue:
**(F, 2)**, (B, 3)

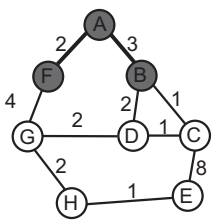
The path is extended by selecting the smallest path seen so far from Priority Queue. This is (A, 0). The vertex removed from Priority Queue is marked as visited (it becomes shaded). If they haven't already been visited, the vertices adjacent to A are added to Priority Queue along with their distance from the source vertex (A). From A we can get to F with a cost of 2, and to B with a cost of 3.




Vertex: F

Priority Queue:
**(B, 3)**, (G, 6)

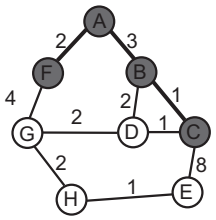
The path from A to F with a cost of 2 is the cheapest path seen so far, so (F, 2) is removed from Priority Queue. F is marked as visited. G is adjacent to F and hasn't been visited yet, so (G, 6) is added to Priority Queue. Note that the cost of 6 represents the *total* cost from A to G:
(A, F, 2) + (F, G, 4) => (A, G, 6)




Vertex: B

Priority Queue:
**(C, 4)**, (D, 5)
(G, 6)

The cheapest path in Priority Queue is (B, 3), which is removed from Priority Queue. B is marked as visited. C and D are adjacent to B and have not been visited, so we add paths from A to D (D, 5) and A to C (C, 4) to Priority Queue.

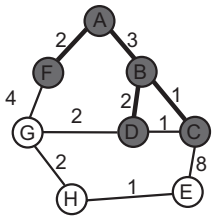



Vertex: C

Priority Queue:
**(D, 5)**, (G, 6)
(E, 12)

(C, 4) was at the front of Priority Queue, so is removed and C is marked as visited. Unvisited vertices D and E are adjacent to C, however, since (D, 5) is already in Priority Queue, it does not get added again. (E, 12) is added. Note that B is also adjacent to C, but since B has already been visited, it is ignored.

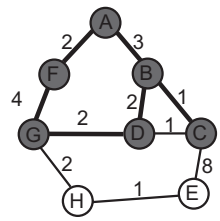
**Figure 13.10 (continued)** Annotated example of the minimal path algorithm searching for a path from A to E.




Vertex: D

Priority Queue:
**(G, 6)**, (E, 12)
(G, 7)

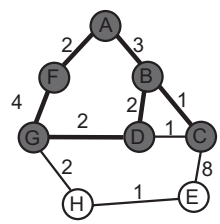
(D, 5) was at the front of Priority Queue, so is removed and D is marked as visited. Adjacent vertices B and C have already been visited, so are ignored. Unvisited vertex G is added to Priority Queue along with its total cost from the source vertex. Note that this produces another path from A to G. However, we will always select the *least cost* path from the Priority Queue, so the extra path does not affect the algorithm's correctness.




Vertex: G

Priority Queue:
**(G, 7)**, (E, 12)
(H, 8)

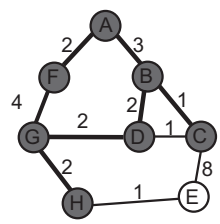
(G, 6) is removed from Priority Queue and G is marked as visited. Unvisited vertex H is added to Priority Queue with its total cost from the source vertex. (H, 8) represents the path {A, F, G, H}. Vertices F and D are also adjacent to G, but have been visited already, so are ignored.




Vertex: G

Priority Queue:
**(H, 8)**, (E, 12)
(H, 9)

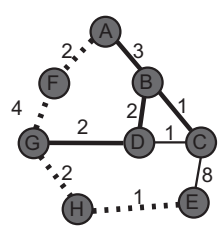
(G, 7) is removed from Priority Queue. Again, since adjacent vertex H is unvisited, it is added to Priority Queue with its total cost from the source vertex following this *different* path: (H, 9) = {A, B, D, G, H}.




Vertex: H

Priority Queue:
**(E, 9)**, (E, 12)
(H, 9)

(H, 8) is removed from Priority Queue. Unvisited adjacent vertex E is added to Priority Queue (E, 9).




Vertex: E

Priority Queue:
**(H, 9)**, (E, 12)

(E, 9) is removed from Priority Queue. Since E is the destination vertex, the search is over and since (E, 9) was the minimal path from Priority Queue, this must be the minimal path to the destination vertex.

The minimal path is {A, F, G, H, E} with a cost of 9.

edge. With *w* added to our MST, we then add to the collection of candidate edges all the edges *(w, x)* for which *x* is not in our MST. This process continues until either there are no more candidate edges to examine or the number of edges in MST is $|V| - 1$, which you'll recall is the minimum number of edges for a connected graph

with $|V|$ vertices. If the main loop terminates and the number of edges in the MST is less than $|V| - 1$, the graph is not connected and failure is returned.

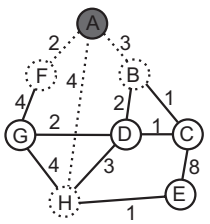
Let's look at a high level algorithm (appears after Figure 13.11) that succinctly captures what has to be done. An annotated example is shown in Figure 13.11.

**Figure 13.11** Annotated example of Prim's algorithm to find a minimal spanning tree.


A vertex / edge that is part of the MST.

A vertex / edge in the collection of candidate edges.

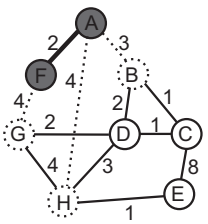
A vertex / edge in the graph that hasn't been seen yet.




MST: A

Candidate Edges: **(A, F, 2)**, (A, H, 4), (A, B, 3)

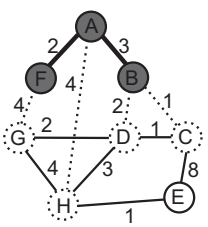
The source vertex, A, is added to the MST. Edges incident on A become candidate edges on the fringe of the MST being built.




MST: A, F

Candidate Edges: **(A, B, 3)**, (A, H, 4), (F, G, 4)

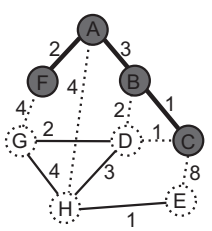
The least cost candidate edge (A, F, 2) is selected. By definition it has one vertex (A) in the MST and the other, F, not in the MST. F is added to the MST and edges incident on F are added to the candidate edges. Note this does not include edge (F, A, 2) since F and A are already in the MST.




MST: A, F, B

Candidate Edges: **(B, C, 1)**, (A, H, 4), (F, G, 4), (B, D, 2)

The least cost candidate edge, (A, B, 3) is selected and is added to the MST causing edges (B, D, 2) and (B, C, 1) to be added to the candidate edges.

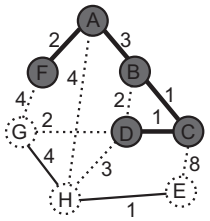



MST: A, F, B, C

Candidate Edges: **(C, D, 1)**, (A, H, 4), (F, G, 4), (B, D, 2), (C, E, 8)

Edge (B, C, 1) is selected from the pool of candidate edges and is added to the MST. This provides additional candidate edges (C, D, 1) and (C, E, 8).

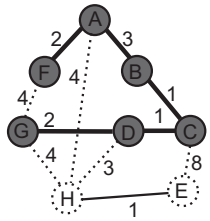
**Figure 13.11 (continued)** Annotated example of Prim's algorithm to find a minimal spanning tree




MST: A, F, B, C, D

Candidate Edges: (*B, D, 2*), (A, H, 4) (F, G, 4), (C, E, 8), **(D, G, 2)**, (A, H, 4)

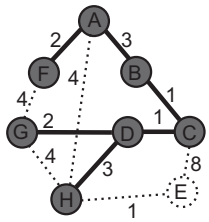
Edge (C, D, 1) is selected from the pool of candidate edges and added to the MST. This provides additional candidate edges (D, G, 2) and (D, H, 3). Edge (B, D, 2) is the least cost edge among the candidate edges, but since both of its vertices are already in the MST, it will be removed and ignored. The least cost edge with one vertex *not* in the MST is (D, G, 2).




MST: A, F, B, C, D, G

Candidate Edges: **(D, H, 3)**, (A, H, 4) (F, G, 4), (C, E, 8), (G, H, 4)

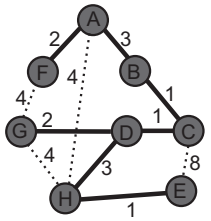
(D, G, 2) is removed from the candidate edges and is added to the MST. From vertex G, we add (G, H, 4) to the candidate edges.




MST: A, F, B, C, D, G, H

Candidate Edges: **(H, E, 1)**, (A, H, 4) (F, G, 4), (C, E, 8), (G, H, 4)

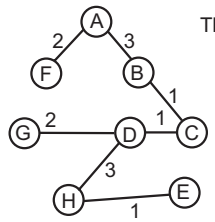
Edge (D, H, 3) is the least cost edge among the candidate edges and is added to the MST. From vertex H we add (H, E, 1) to the candidate edges.




MST: A, F, B, C, D, G, H, E

Candidate Edges: **(A, H, 4)**, (F, G, 4) (C, E, 8), (G, H, 4)

Finally, edge (H, E, 1) is added to the MST. There are now |V| vertices and |V| –1 edges in the MST, so the construction is complete.




The MST with vertex A as its root.

*// Identify a Minimal Spanning Tree for a weighted Graph g with Vertex v as its root.*

**Pseudocode: minimalSpanningTree ( Graph g, Vertex v )**
MST—a collection of edges (v, w) in the Minimal Spanning Tree; initially this collection is empty
mstCost—the total cost of all edges in MST; initially 0
visitedVertices—a collection of all vertices that are in MST; initially this collection stores a vertex from G that will be the root of the MST

```
allVerticesVisited = false
put the starting vertex v in visitedVertices
// while the MST is not complete and there are vertices to visit
while MST.size() < |V| – 1 and not allVerticesVisited
   // select the least cost candidate edge
   get the least cost edge (v, w) such that v is in visitedVertices
   and w is not in visitedVertices
if no such edge exists then
   allVerticesVisited is true
else
   add (v, w) to MST
   add the cost of edge (v, w) to mstCost
   add w to visitedVertices
   // loop invariant: MST contains the edges of a minimal spanning
   //                 tree for all vertices from G that are in visitedVertices
if MST.size() != |V| – 1 then
   return failure
return MST              // success
```

## SUMMING UP

A minimal cost path between two vertices in a weighted graph is the path with the least total path cost over all possible paths between the vertices. The minimal path algorithm uses a greedy strategy to find the minimal cost path. At each step, a greedy algorithm makes locally optimal (greedy) selections to produce a globally optimal solution. The minimal path algorithm builds a collection of *possible* paths to the destination, extending these paths with the least cost edge available. The algorithm terminates when a least cost edge to the destination vertex is added to a possible path.

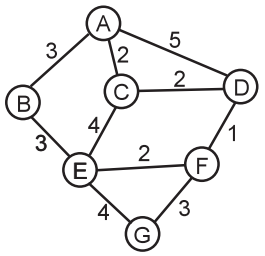
A minimal spanning tree (MST) is a spanning tree of a weighted graph that has the least total cost over all possible spanning trees for the graph. Prim's algorithm finds a minimal spanning tree using a greedy strategy. Starting from a single vertex in the graph, it keeps track of candidate edges *(v, w)* such that *v* is in the MST and *w* is not. In adding *w* to the MST, the pool of candidate edges is extended by those edges *(w, x)* incident on *w* for which *x* is not in the MST. The algorithm terminates with success when there are $|V|$ vertices and $|V| - 1$ edges in the MST.

## CHECKPOINT

**13.10** What makes a greedy algorithm greedy?

**13.11** The minimal path algorithm works when there are multiple paths with different costs represented in *PriorityQueue* (⟨*H, 8*⟩ and ⟨*H, 9*⟩, for example). Why does this not pose a problem for the algorithm?

**13.12** Will you get the same MST for a graph no matter which vertex is the MST's starting point?




Graph $G_{11}$.

**13.13** Apply the minimal path algorithm to graph $G_{11}$ to find the cost of the minimal path from A to G.

**13.14** Apply the MST algorithms to graph $G_{11}$ starting at vertex A.

# 13.5 Graph ADTs

In this section we specify three related ADTs: `Graph`, `WeightedGraph`, and `BFSSearcher`. In the next section we will look at their implementation.

## 13.5.1 The Graph ADT

When designing a graph we must decide what needs to be represented explicitly. For example, do we need a type to represent vertices? What about edges?

The approach taken here is to represent vertices *explicitly* because we want to be able to identify them via a label, and to represent edges *implicitly* via the vertices they connect. This design decision is reflected in the Graph ADT, which specifies an unweighted graph that could be directed or undirected. The ADT supports adding and deleting vertices and edges, getting the number of vertices and edges in the graph, and getting a list of the neighbors of a vertex.

It is in the addition and deletion of edges that directionality matters. As you would expect, the operation description specifies that if the implementation is for a directed graph, then `addEdge(v1, v2)` creates an edge *from* `v1` *to* `v2`. If the graph is undirected, there is *also* an edge from `v2` to `v1`. This piece of otherwise obvious information is offered because it will play a key role in making some implementation decisions in the next section.

## ADT Name: Graph

**Description:**
A Graph is a non-linear collection containing vertices and edges connecting vertices. This ADT does not specify if the edges are directed, leaving that to an implementation. The edges have no weights.

**Invariants:**
**1.** Empty graph: *number of vertices* is 0; *number of edges* is 0.
**2.** Self-loops are not allowed.

**Attributes:**

| | |
|---|---|
| *number of vertic*es: | The number of vertices in the graph; number of vertices ≥ 0. |
| *number of edges*: | The number of edges in the graph; 0 ≤ number of edges ≤ $n$ ($n$ – 1), where $n$ is the number of vertices in the graph. |

**Operations:**

Graph ()

| | |
|---|---|
| pre-condition: | none |
| responsibilities: | initializes the graph attributes |
| post-condition: | *number of vertices* is 0<br>*number of edges* is 0 |
| returns: | nothing |

addVertex( Vertex v )

| | |
|---|---|
| pre-condition: | v is not already in the graph |
| responsibilities: | insert a Vertex into this graph |
| post-condition: | a Vertex is added to this graph<br>*number of vertices* is incremented by 1 |
| exception: | if Vertex v is already in this graph |
| returns: | nothing |

addEdge( Vertex v1 , Vertex v2 )

| | |
|---|---|
| pre-condition: | v1 and v2 are Vertices in this graph and aren't already connected by an edge |
| responsibilities: | connect Vertices v1 to v2; if this is an undirected graph, this edge also connects v2 to v1 |
| post-condition: | an edge connecting v1 and v2 is added to this graph<br>*number of edges* is incremented by 1 |
| exception: | if v1 or v2 are not in the graph or are already connected by an edge |
| returns: | nothing |

removeVertex ( Vertex v )

| | |
|---|---|
| pre-condition: | v is a Vertex in this graph |
| responsibilities: | remove Vertex v from this graph |

post-condition: Vertex v is removed from this graph
All edges incident on v are removed
*number of vertices* is decremented by 1
*number of edges* is decremented by degree( v )
exception: if v is not in this graph
returns: nothing

removeEdge (Vertex v1 , Vertex v2 )
pre-condition: v1 and v2 are vertices in this graph and an edge exists from v1 to v2
responsibilities: remove from this graph the edge connecting v1 to v2; if this is an undirected graph, there is no longer an edge from v2 to v1
post-condition: the edge connecting v1 and v2 is removed from this graph
*number of edges* is decremented by 1
exception: if v1 or v2 are not in this graph, or if no edge from v1 to v2 exists
returns: nothing

getNeighbors ( Vertex v )
pre-condition: v is a Vertex in this graph
responsibilities: get the neighbors of Vertex v from this graph
post-condition: the graph is unchanged
exception: if v is not in this graph
returns: a collection containing the Vertices incident on v

getNumberOfVertices()
pre-condition: none
responsibilities: get the number of vertices in this graph
post-condition: the graph is unchanged
returns: the number of vertices in this graph

getNumberOfEdges()
pre-condition: none
responsibilities: get the number of edges in this graph
post-condition: the graph is unchanged
returns: the number of edges in this graph

### 13.5.2 The WeightedGraph ADT

As described at the beginning of the chapter, a weighted graph is simply a graph whose edges have weights. Consequently, the weighted graph specified in the WeightedGraph ADT is shown as an extension of the Graph ADT. The only significant difference is the addition of operations to support edge weights. The presentation of a weighted graph as an extension of a graph will be reflected in the implementation described in the next section.

**ADT Name: WeightedGraph extends Graph**

**Description:**
A WeightedGraph is a graph whose edges have non-negative weights.

**Invariants:**
**1.** Edge weights must be >= 0; the default edge weight is 1.0.

**Operations:**

addEdge( Vertex v1 , double w, Vertex v2 )

| | |
|---|---|
| pre-condition: | v1 and v2 are Vertices in this graph and aren't already connected by an edge; w is >= 0 |
| responsibilities: | connect Vertices v1 to v2 with weight w; if this is an undirected graph, this edge also connects v2 to v1 |
| post-condition: | an edge connecting v1 and v2 with weight w is added to this graph<br>*number of edges* is incremented by 1 |
| exception: | if v1 or v2 are not in the graph, are already connected by an edge, or w < 0 |
| returns: | nothing |

getEdgeWeight( Vertex v1 , Vertex v2 )

| | |
|---|---|
| pre-condition: | v1 and v2 are Vertices in this graph and are connected by an edge |
| responsibilities: | get the weight of the edge connecting Vertices v1 to v2 |
| post-condition: | the graph is unchanged |
| exception: | if v1 or v2 are not in the graph or are not connected by an edge |
| returns: | the weight of the edge connecting v1 to v2 |

setEdgeWeight( Vertex v1 , double newWeight, Vertex v2 )

| | |
|---|---|
| pre-condition: | v1 and v2 are Vertices in this graph and are connected by an edge; newWeight is >= 0 |
| responsibilities: | set the weight of the edge connecting Vertices v1 to v2 to newWeight |
| post-condition: | the graph is unchanged |
| exception: | if v1 or v2 are not in the graph, are not connected by an edge, or newWeight < 0 |
| returns: | nothing |

### 13.5.3 The BFSSearcher ADT

Finally, the BFSSearcher ADT specifies a breadth first search utility tool. Its constructor takes a graph as an argument and supports operations to determine if a path exists between two vertices in the graph, the length of that path (in edges), and the vertices in the path. We use this tool in the test plans developed next.

**ADT Name: BFSSearcher**

**Description:**
Performs searches on a given Graph using breadth first search.

**Attributes:**

| | |
|---|---|
| *graph*: | The graph to search. |

**Operations:**

BFSSearcher ( Graph g )

| | |
|---|---|
| pre-condition: | g is not null |
| responsibilities: | constructor initializes the graph attributes |
| post-condition: | set *graph* to g |
| exception: | if g is null |
| returns: | nothing |

containsPath ( Vertex v1, Vertex v2 )

| | |
|---|---|
| pre-condition: | v1 and v2 are Vertices in *graph* |
| responsibilities: | determine if there is a path from v1 to v2 in *graph* |
| post-condition: | *graph* is unchanged |
| exception: | if v1 or v2 are not in *graph* |
| returns: | true if there is a path from v1 to v2 in *graph*; false otherwise |

getPathLength ( Vertex v1, Vertex v2 )

| | |
|---|---|
| pre-condition: | v1 and v2 are Vertices in *graph* |
| responsibilities: | determine the length of the path (in number of edges) from v1 to v2 in *graph* |
| post-condition: | *graph* is unchanged |
| exception: | if v1 or v2 are not in *graph* |
| returns: | the length of the path from v1 to v2, 0 if there is no path |

getPath ( Vertex v1, Vertex v2 )

| | |
|---|---|
| pre-condition: | v1 and v2 are Vertices in *graph* |
| responsibilities: | get the vertices on the path from v1 to v2 in *graph* |
| post-condition: | *graph* is unchanged |
| exception: | if v1 or v2 are not in *graph* |
| returns: | a collection containing the vertices on the path from v1 to v2, null if there is no path |

## 13.6 Implementation of Graph ADTs

**How should graph types be related?**

The `Graph` interface shown in Listing 13.1 reflects the graph specification in the Graph ADT. Like the ADT, the interface is silent on whether or not an implementing class is a directed or undirected graph. Directed, undirected, and weighted

**Listing 13.1** The `Graph` interface based on the Graph ADT.

```
1   package gray.adts.graph;
2
3   import java.util.List;
4
5   /**
6    * A graph G consists of a finite set of vertices, V, and edges,
7    * E. An edge in E connects two vertices in V.
8    *
9    * This interface makes no assumption about whether the graph
10   * is directed or undirected; implementing classes make that
11   * specialization. Edges are unweighted.
12   */
13  public interface Graph<T> {
14
15      /**
16       * Insert <tt>Vertex v</tt> into this graph.
17       * @param v The <tt>Vertex</tt> to add to the graph;
18       * can't already be in the graph.
19       * @throws IllegalArgumentException if <tt>v</tt> is
20       * already in this graph.
21       */
22      void addVertex( Vertex<T> v );
23
24      /**
25       * Add an edge connecting vertex <tt>v1</tt> to
26       * <tt>v2</tt>. The edge must not already be in the
27       * graph. In an undirected graph, this edge is bidirectional.
28       * @param v1 The source vertex; must not be
29       * <tt>null</tt> and must be a vertex in this graph.
30       * param v2 The destination vertex; must not be
31       * <tt>null</tt> and must be a vertex in this graph.
32       * @throws IllegalArgumentException if <tt>v1</tt> or
33       * <tt>v2</tt> are <tt>null</tt>, are not in this
34       * graph, or if the edge already exists.
35       */
36      void addEdge( Vertex<T> v1 , Vertex<T> v2 );
37
38      /**
39       * Remove vertex <tt>v</tt> and all edges incident on
40       * <tt>v</tt> from this graph.
41       * @param v The vertex to remove; must not be
42       * <tt>null</code and must be a vertex in this graph.
43       * @throws IllegalArgumentException if <tt>v1</tt> is
```

```
44         * <tt>null</tt> or is not in this graph.
45         */
46        void removeVertex( Vertex<T>  v );
47
48        /**
49         * Remove the edge from <tt>v1</tt> to <tt>v2</tt>
50         * from this graph.
51         *
52         * @parm v1 The source vertex for the edge to remove; must
53         * not be <tt>null</tt> and must be a vertex in this
54         * graph.
55         * @param v2 The destination vertex for the edge to remove;
56         * must not be <tt>null</tt> and must be a vertex in this
57         * graph.
58         * @throws IllegalArgumentException if <tt>v1</tt> or
59         * <tt>v2</tt> are <tt>null</tt>, are not in this
60         * graph, or the edge doesn't exist.
61         */
62        void removeEdge( Vertex<T> v1, Vertex<T> v2 );
63
64
65        /**
66         * Get the neighbors of Vertex <tt>v</tt> in this graph.
67         * @param v Vertex The vertex whose neighbors we want; must
68         * not be <tt>null</tt> and must be a vertex in this
69         * graph.
70         * @return List The vertices incident on <tt>v</tt>.
71         * @throws IllegalArgumentException if <tt>v1</tt> is
72         * <tt>null</tt> or is not in this graph.
73         */
74        List< Vertex<T> > getNeighbors ( Vertex<T> v );
75
76        /**
77         * Get the number of vertices in this graph.
78         * @return int The number of vertices in this graph.
79         */
80        int getNumberOfVertices();
81
82        /**
83         * Get the number of edges in this graph.
84         * @return int The number of edges in this graph.
85         */
86        int getNumberOfEdges();
87    }
```

graphs are clearly related so we would like to create an inheritance hierarchy of classes that promotes code reuse through inheritance. But can a directed graph class and an undirected graph class be directly related? Should a digraph be a subclass of an undirected graph class, or vice versa? How do weighted graphs affect this design?

The discussion of graph ADTs in the last section gave a hint about how to relate our graph classes. The deciding issue is how to handle the addition and deletion of edges. The `addEdge(v1, v2)` operation will produce an edge from `v1` to `v2` in a directed graph, *and* an edge from `v2` to `v1` in an undirected graph. With this in mind, we will have `AdjMatrixDiGraph` implement the `Graph` interface for directed graphs using an adjacency matrix to store the graph. To represent undirected graphs we'll have `AdjMatrixGraph` be a subclass of `AdjMatrixDiGraph`. When dealing with edges, its methods will invoke their superclass (directed graph) counterparts, and then will handle the bidirectionality of an undirected graph locally. We'll do the implementation shortly and you will see that this design makes the overridden methods quite short, but we need to do some planning to make this work. `WeightedAdjMatrixGraph` is a subclass of `AdjMatrixGraph` and will also implement the `WeightedGraph` interface.

The UML class diagram shown in Figure 13.12 shows our hierarchy for the adjacency matrix representations. The methods listed in a class are implemented in that class or are overridden from an inherited class. Portions of these classes will be presented here with emphasis on code reuse through inheritance. We'll also look at an implementation of the minimal path algorithm.

### 13.6.1 The `AdjMatrixGraph` Class: A Directed Graph Implementation of the `Graph` Interface

The Graph ADT attributes are declared on lines 12 and 13 of Listing 13.2, and the adjacency matrix, `adjMatrix` is declared on line 19. An array, `vertices`, of `Vertex` references is declared on line 26. The ith entry in `vertices` corresponds to the ith row and column of `adjMatrix`. These fields are declared `protected` to allow subclasses to access them directly.

Thinking ahead to our needs in the subclasses, `int` variables `v1Pos` and `v2Pos` (line 32) are also `protected`. These integers will represent an entry in the adjacency matrix and will be used by all methods (in this class and in subclasses) that access an edge.

**`addEdge()`**

Method `addEdge(v1, v2)` (lines 46–72) creates a directed edge from `v1` to `v2`. It uses a utility method, `getVertexIndexFor()`, to get the index into the `vertices` array for some `Vertex`. If either of the vertices is not found (lines 62–64) or if the edge already exists (line 66), an exception is thrown. Otherwise, an entry is made indicating a new edge (line 67). Remember, this implementation is for a *directed* graph, so the method only establishes an edge *from* `v1` *to* `v2`.

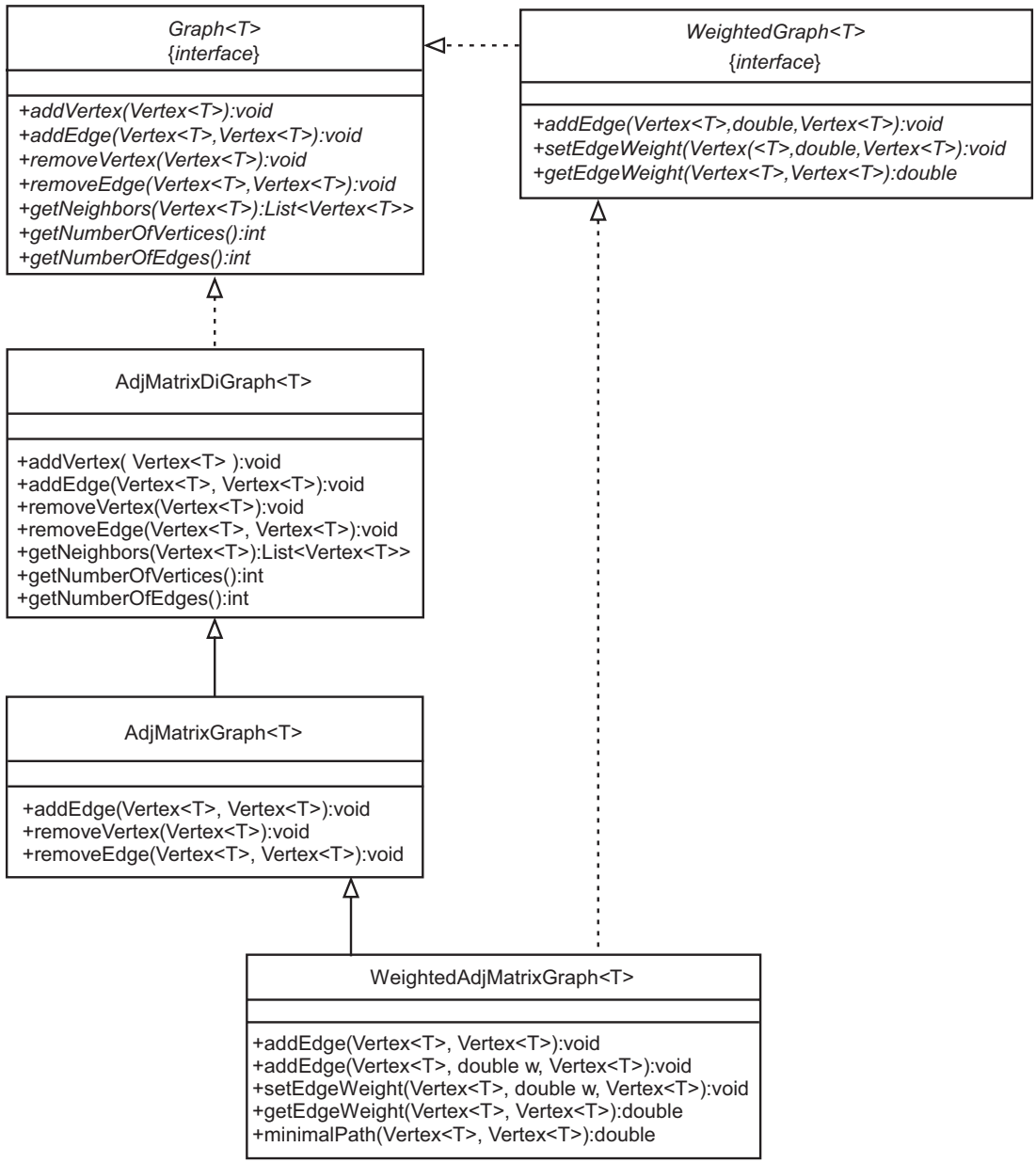
**`removeEdge()`**

Method `removeEdge(v1, v2)` (lines 74–97) removes the edge from `v1` to `v2` and has the same structure as `addEdge()`. The only difference is that the `adjMatrix` entry is cleared (line 93).

**`removeVertex()`**

Method `removeVertex(v)` (lines 99–129) must not only remove the vertex from the graph (lines 113–114), it must also remove all edges coming into or going out of the vertex. This is accomplished by walking along the vertex's row in the adjacency

**Figure 13.12** The graph hierarchy. The methods shown in a concrete class are either implemented in that class or are overridden there to provide a specialization of the method's behavior (UML class diagram).




matrix looking for an edge from $v$ to some other vertex. When such an edge is found, it is cleared and `numberOfEdges` is decremented (lines 116–123). Similarly, we then walk down the column for $v$ (lines 124–127) looking for edges *coming into* $v$.

**Listing 13.2** Portion of the `AdjMatrixDiGraph` class.

```
 1  package gray.adts.graph;
 2
 3  import java.util.*;
 4
 5  /**
 6   * An implementation of the <tt>Graph</tt> interface for a
 7   * directed graph using an adjacency matrix to indicate the
 8   * presence/absence of edges connecting vertices in the graph.
 9   */
10  public class AdjMatrixDiGraph<T> implements Graph<T> {
11
12      protected int numberOfVertices;
13      protected int numberOfEdges;
14
15      /**
16       * adjMatrix[i][j] = 1; an edge exists FROM vertex i TO vertex j
17       * adjMatrix[i][j] = 0; NO edge exists from vertex i to vertex j
18       */
19      protected int[][] adjMatrix;
20
21      /**
22       * Stores the vertices that are part of this graph. There is no
23       * requirement that the vertices be in adjacent cells of the
24       * array; as vertices are deleted, some gaps may appear.
25       */
26      protected Vertex<T>[] vertices;
27
28      /**
29       * v1Pos and v2Pos represent a position in adjMatrix. This
30       * class and subclasses use it to access an edge.
31       */
32      protected int v1Pos, v2Pos;
33
34      protected static int SIZE = 10;
35
36      /**
37       * Constructor. Create an empty instance of a directed graph.
38       */
39      public AdjMatrixDiGraph() {
40          this.numberOfVertices = 0;
41          this.numberOfEdges = 0;
42          this.adjMatrix = new int[this.SIZE][this.SIZE];
43          this.vertices = new Vertex[this.SIZE];
44      }
45
```

```
46        /**
47         * Add an edge connecting vertex <tt>v1</tt> to
48         * <tt>v2</tt>. This edge must not already be in the graph.
49         * In an undirected graph, this edge is bidirectional.
50         * @param v1 the source vertex; must not be <tt>null</tt>
51         * and must be a vertex in this graph.
52         * @param v2 the destination vertex; must not be
53         * <tt>null</tt> and must be a vertex in this graph.
54         * @throws IllegalArgumentException if <tt>v1</tt> or
55         * <tt>v2</tt> are <tt>null</tt>, are not in this
56         * graph or if the edge already exists in the graph.
57         */
58        public void addEdge( Vertex<T> v1, Vertex<T> v2 ) {
59            v1Pos = getVerticesIndexFor( v1 );
60            v2Pos = getVerticesIndexFor( v2 );
61
62            if ( v1Pos == -1 || v2Pos == -1 ) {
63                throw new IllegalArgumentException("vertex not found");
64            }
65            // avoid adding duplicate edges
66            if ( this.adjMatrix[v1Pos][v2Pos] == 0 ) {
67                this.adjMatrix[v1Pos][v2Pos] = 1;
68                this.numberOfEdges++;
69            }
70            else throw new IllegalArgumentException( "duplicate edge "
71                                                     + v1 + " " + v2 );
72        }
73
74        /**
75         * Remove the edge from <tt>v1</tt> to <tt>v2</tt> from
76         * this graph.
77         * @param v1 the source vertex of the edge to remove; must not
78         * be <tt>null</tt> and must be a vertex in this graph.
79         * @param v2 the destination vertex of the edge to remove; must
80         * not be <tt>null</tt> and must be a vertex in this graph.
81         * @throws IllegalArgumentException if <tt>v1</tt> or
82         * <tt>v2</tt> are <tt>null</tt>, are not in this
83         * graph, or the edge doesn't exist.
84         */
85        public void removeEdge( Vertex<T> v1, Vertex<T> v2 ) {
86            v1Pos = getVerticesIndexFor( v1 );
87            v2Pos = getVerticesIndexFor( v2 );
88
89            if ( v1Pos == -1 || v2Pos == -1 ) {
90                throw new IllegalArgumentException("vertex not found");
91            }
92            if ( this.adjMatrix[v1Pos][v2Pos] == 1 ) {
```

```
 93                 this.adjMatrix[v1Pos][v2Pos] = 0;
 94                 this.numberOfEdges--;
 95             }
 96             else throw new IllegalArgumentException("edge not found");
 97         }
 98
 99         /**
100          * Remove vertex <tt>v</tt> and all edges incident
101          * on <tt>v</tt> from this graph.
102          * @param v the vertex to remove; must not be
103          * <tt>null</tt> and must be a vertex in this graph.
104          * @throws IllegalArgumentException if <tt>v1</tt>
105          * is <tt>null</tt> or is not in this graph.
106          */
107         public void removeVertex( Vertex<T> v ) {
108             int pos = getVerticesIndexFor( v );
109             if ( pos == -1 ) {
110                 throw new IllegalArgumentException("vertex not found");
111             }
112
113             this.numberOfVertices--;
114             this.vertices[pos] = null;
115
116             // now we need to go through the adjacency matrix and
117             // remove all edges incident on v. We do this by walking
118             // along the row and column for v in the adjacency matrix
119             for ( int i = 0; i < vertices.length; i++ ) {
120                 if ( this.adjMatrix[pos][i] == 1 ) {  // row check
121                     this.adjMatrix[pos][i] = 0;
122                     this.numberOfEdges--;
123                 }
124                 if ( this.adjMatrix[i][pos] == 1 ) {  // column check
125                     this.adjMatrix[i][pos] = 0;
126                     this.numberOfEdges--;
127                 }
128             }
129         }
. . .
```

Because we'll revisit this issue shortly it is important to note that for each such edge found, the edge is cleared and `numberOfEdges` is decremented.

The other methods in `AdjMatrixDiGraph` are not shown and can be found in the source code on the book's companion Web site.

### 13.6.2 The `AdjMatrixGraph` Class: A Subclass of `AdjMatrixDigraph`

`AdjMatrixGraph` is the adjacency matrix implementation of an *un*directed graph. As discussed earlier and shown in Figure 13.12, it is a subclass of `AdjMatrixDigraph`. The complete class is shown in Listing 13.3. Here we continue our focus on the use of inheritance and overridden methods.

**`addEdge()`**

Method `addEdge(v1, v2)` (lines 12–29) invokes its superclass counterpart to make an entry for the edge from `v1` to `v2`. If that operation is successful, variables `v1Pos` and `v2Pos` hold the information needed to make the edge bidirectional (line 28). Only two lines of code—not bad! Note the comment on lines 26–27 explaining our expectation about what executing the superclass method means.

**`removeEdge()`**

Not surprisingly, method `removeEdge(v1, v2)` (lines 31–49) has the same structure as `addEdge()`. Again, code reuse through inheritance (and some careful design) has led to a compact implementation.

**`removeVertex()`**

Finally, we look at `removeVertex(v)` (lines 51–69). When removing a vertex we must remove all edges going into and coming out of the vertex. Method `removeVertex()` in the *directed* graph `AdjMatrixDiGraph` does this by walking along the row and down the column for the vertex's entry in the adjacency matrix zeroing out any entries storing 1. Unfortunately, in a *directed* graph, each of these entries is treated as a *separate* edge, so attribute `numberOfEdges` is decremented twice for each deleted edge in an *un*directed graph. The somewhat complicated code in lines 67–68 corrects this problem by figuring that the actual number of edges that should have been deleted is half of what `AdjMatrixDiGraph` thinks should be deleted.[3]

**Listing 13.3** The `AdjMatrixGraph` class (a subclass of `AdjMatrixDiGraph`).

```
 1  package gray.adts.graph;
 2
 3  /**
 4   * Adjacency Matrix implementation of an undirected Graph.
 5   */
 6  public class AdjMatrixGraph<T> extends AdjMatrixDiGraph<T> {
 7
 8      public AdjMatrixGraph() {
 9          super();
10      }
11
12      /**
13       * Add an edge connecting vertex <tt>v1</tt> to
14       *  <tt>v2</tt>. This edge is bidirectional.
15       * @param v1 the source vertex; must not be <tt>null</tt>
```

[3]The Exercises have you consider a much simpler solution.

```
16          * and must be a vertex in this graph.
17          * @param v2 the destination vertex; must not be
18          * <tt>null</tt> and must be a vertex in this graph.
19          * @throws IllegalArgumentException if <tt>v1</tt> or
20          * <tt>v2</tt> are <tt>null</tt> or are not in this
21          * graph.
22          */
23         public void addEdge( Vertex<T> v1, Vertex<T> v2 ) {
24            super.addEdge(v1, v2);
25
26            // if we get here, the superclass method completed
27            // successfully and we can set edge from v2 to v1
28            this.adjMatrix[v2Pos][v1Pos] = 1;
29         }
30
31         /**
32          * Remove the edge from <tt>v1</tt> to <tt>v2</tt>
33          * from this graph.
34          * @param v1 the source vertex for the edge to remove; must
35          * not be <tt>null</tt> and must be a vertex in this graph.
36          * @param v2 the destination vertex for the edge to remove;
37          * must not be <tt>null</tt> and must be a vertex in this
38          * graph.
39          * @throws IllegalArgumentException if <tt>v1</tt> or
40          * <tt>v2</tt> are <tt>null</tt> or are not in this
41          * graph.
42          */
43         public void removeEdge( Vertex<T> v1, Vertex<T> v2 ) {
44            super.removeEdge( v1, v2 );
45
46            // if we get here, the superclass method completed
47            // successfully and we can clear edge from v2 to v1
48            this.adjMatrix[v2Pos][v1Pos] = 0;
49         }
50
51         /**
52          * Remove vertex <tt>v</tt> and all edges incident on
53          * <tt>v</tt> from this graph.
54          * @param v the vertex to remove; must not be <tt>null</tt>
55          * and must be a vertex in this graph.
56          * @throws IllegalArgumentException if <tt>v1</tt> is
57          * <tt>null</tt> or is not in this graph.
58          */
59        public void removeVertex( Vertex<T> v ) {
60            int numEdges = super.numberOfEdges;
61            super.removeVertex( v );
62
```

**See `addEdge()` in `adjMatrixDiGraph`, lines 46–72**

**See `removeEdge()` in `adjMatrixDiGraph`, lines 74–97**

```
63            // if we get here, the superclass method completed
64            // successfully and we can update the number of edges
65            // For edge (v1, v2), the superclass removeVertex() will
66            // decrement numberOfEdges twice.
67            super.numberOfEdges = numEdges - (numEdges -
68                                                  super.numberOfEdges) / 2;
69        }
70    }
```

### 13.6.3 The `WeightedAdjMatrixGraph` Class: A Subclass of `AdjMatrixGraph`

Listing 13.4 gives the implementation of `WeightedAdjMatrixGraph`. As shown in Figure 13.12 and in the class header (lines 13–14), the class extends class `AdjMatrixGraph` and implements interface `WeightedGraph`. A quick look at the code tells you that the bulk of the work is done in the superclass. Method `addEdge(v1, weight, v2)` (lines 31–55), for example, simply invokes its superclass counterpart, then if all goes well adding the edge, the method updates the weights matrix (declared on line 24). The overloaded `addEdge(v1, v2)` method (lines 104–119) simply uses `addEdge(v1, weight, v2)` to create an edge using a default edge weight.

**Listing 13.4** Part of the `WeightedAdjMatrixGraph` class.

```
1   package gray.adts.graph;
2
3   import java.util.*;
4
5   import gray.adts.priorityqueue.*;
6   import gray.adts.priorityqueue.PriorityQueue;
7   import gray.misc.*;
8
9   /**
10   * A weighted, undirected graph stored in an adjacency matrix.
11   *  The weights must be >= 0.
12   */
13  public class WeightedAdjMatrixGraph<T> extends AdjMatrixGraph<T>
14                                      implements WeightedGraph<T> {
15      /**
16       * The default weight for an edge in a weighted graph.
17       */
18      public float DEFAULT_WEIGHT = 1.0;
19
20      /**
```

```
21          * Store weight edges. The adjacency matrix storing
22          * edges is in an ancestor class.
23          */
24         protected double[][] weights;
25
26         public WeightedAdjMatrixGraph() {
27             super();
28             weights = new double[super.SIZE][super.SIZE];
29         }
30
31         /**
32          * Add an edge connecting vertex <tt>v1</tt> to
33          * <tt>v2</tt>. In an undirected graph, this edge is
34          * bidirectional.
35          * @param v1 the source vertex; must not be <tt>null</tt>
36          * and must be a vertex in this graph.
37          * @param weight the weight of this edge; must be >= 0.0.
38          * @param v2 the destination vertex; must not be
39          * <tt>null</tt> and must be a vertex in this graph.
40          * @throws IllegalArgumentException if <tt>v1</tt> or
41          * <tt>v2</tt> are <tt>null</tt> or are not in this
42          * graph, or if <tt>weight</tt> is < 0.
43          */
44         public void addEdge( Vertex<T> v1, double weight,
45                              Vertex<T> v2 ) {
46             if ( weight < 0.0 )
47                 throw new IllegalArgumentException ("Edge weight " +
48                                                     " must be >= 0.0");
49
50             super.addEdge( v1, v2 );
51
52             // if we get here, method in superclass didn't throw
53             // an exception and method preconditions are met
54             this.setEdgeWeight( v1, weight, v2 );
55         }
56
57         /**
58          * Get the weight of the edge from <tt>v1</tt> to
59          * <tt>v2</tt>.
60          * @param v1 the source vertex; must not be
61          * <tt>null</tt> and must be a vertex in this graph.
62          * @param v2 the destination vertex; must not be
63          * <tt>null</tt> and must be a vertex in this graph.
64          * @return double the weight of the edge from <tt>v1</tt>
65          * to <tt>v2</tt>.
66          * @throws IllegalArgumentException if <tt>v1</tt> or
67          * <tt>v2</tt> are <tt>null</tt> or are not in this
```

**See `addEdge()` in `adjMatrixDiGraph`, lines 46–72**

```
68          * graph.
69          */
70         public double getEdgeWeight( Vertex<T> v1, Vertex<T> v2 ) {
71             int v1Pos = super.getVerticesIndexFor( v1 );
72             int v2Pos = super.getVerticesIndexFor( v2 );
73             // if we get here, method in superclass didn't throw
74             // an exception and method preconditions are met
75             return weights[v1Pos][v2Pos];
76         }
77
78         /**
79          * Reset the weight for the edge connecting vertex
80          * <tt>v1</tt> to <tt>v2</tt>.
81          * @param v1 the source vertex; must not be <tt>null</tt>
82          * and must be a vertex in this graph.
83          * @param newWeight the weight of this edge; must be >= 0.0.
84          * @param v2 the destination vertex; must not be
85          * <tt>null</tt> and must be a vertex in this graph.
86          * @throws IllegalArgumentException if <tt>v1</tt> or
87          * <tt>v2</tt> are <tt>null</tt> or are not in this
88          * graph, or if <tt>weight</tt> is < 0.
89          */
90         public void setEdgeWeight( Vertex<T> v1, double newWeight,
91                                    Vertex<T> v2 ) {
92             if ( newWeight < 0.0 )
93               throw new IllegalArgumentException ("Edge weight "
94                                                   + "must be >= 0.0");
95             int v1Pos = super.getVerticesIndexFor( v1 );
96             int v2Pos = super.getVerticesIndexFor( v2 );
97             // if we get here, method in superclass didn't throw an
98             // exception and method preconditions are met
99             weights[v1Pos][v2Pos] = newWeight;
100            weights[v2Pos][v1Pos] = newWeight;
101        }
102
103        // overloaded methods from AdjMatrixGraph
104        /**
105         * Add an edge connecting vertex <tt>v1</tt> to
106         * <tt>v2</tt>. The edge is bidirectional in an
107         * undirected graph. The default weight for an edge
108         * is <tt>DEFAULT_WEIGHT</tt>.
109         * @param v1 the source vertex; must not be <tt>null</tt>
110         *  and must be a vertex in this graph.
111         * @param v2 the destination vertex; must not be
112         * <tt>null</tt> and must be a vertex in this graph.
113         * @throws IllegalArgumentException if <tt>v1</tt> or
114         * <tt>v2</tt> are <tt>null</tt> or are not in this
```

```
115         * graph.
116         */
117        public void addEdge( Vertex<T> v1, Vertex<T> v2 ) {
118            this.addEdge( v1, DEFAULT_WEIGHT, v2 );
119        }
120
121        /**
122         * Find a minimal cost path from <tt>src</tt> to
123         * <tt>dest</tt> in this graph. Assumes edge weights
124         * are positive.
125         * @param src Vertex the first vertex in the path.
126         * @param dest Vertex the last vertex in the path.
127         * @return double the cost of the path or -1 if none is found.
128         */
129        public double minimalPath( Vertex<T> src, Vertex<T> dest ) {
130            // keep track of which vertices have been visited already
131            ArrayList<Vertex<T>> visitedVertices =
132                                  new ArrayList<Vertex<T>> ();
133
134            // Comparator for the priority queue where the shortest
135            //  paths found so far are stored.
136            final Comparator pathCostComparator = new Comparator() {
137                public int compare( Object o1, Object o2 ) {
138                    Double i1 =
139                          ((Tuple<Vertex<T>, Double>)o1).getSecondElement();
140                    Double i2 =
141                          ((Tuple<Vertex<T>, Double>)o2).getSecondElement();
142                    return i1.compareTo( i2 );
143                }
144
145                public boolean equals( Object obj ) {
146                    return false;
147                }
148            };
149
150            // Stores the shortest paths from the source vertex
151            // found so far. These are stored as tuples.
152            // The first field of the tuple is the terminating
153            //  node in some shortest path starting from src
154            // The second field is the cost of that path
155            PriorityQueue<Tuple<Vertex<T>, Double>> pq
156                = new HeapPriorityQueue<Tuple<Vertex<T>,
157                                        Double>>( pathCostComparator );
158
159            Tuple<Vertex<T>, Double> pathTuple;
160
```

```
161             // start with the source, which has a cost of 0 to
162             //  get to itself
163             pq.enqueue( new Tuple( src, 0.0 ) );
164
165             while ( !pq.isEmpty() ) {
166                 // get cheapest path seen so far from src to some
167                 //  other vertex
168                 pathTuple = pq.dequeue();
169
170                 // extract the fields of the tuple so we can
171                 //  work with them
172                 Vertex<T> v = pathTuple.getFirstElement();
173                 double minCostToV = pathTuple.getSecondElement();
174
175                 visitedVertices.add( v ); // visit vertex v
176
177                 // if v is the destination vertex, we are done
178                 if ( v.equals( dest ) ) {
179                     return minCostToV;
180                 }
181
182                 // okay, not done yet; look at the vertices
183                 // adjacent to v
184                 ArrayList<Vertex<T>>
185                     neighbors = (ArrayList<Vertex<T>>)getNeighbors(v);
186                 while ( !neighbors.isEmpty() ) {
187                     Vertex<T> w = neighbors.remove( 0 ); // next neighbor
188
189                     // if w hasn't been visited already, add it to
190                     // the priority queue
191                     if ( !visitedVertices.contains( w ) ) {
192                         // get the total path cost from src to v
193                         double minCostToW = minCostToV
194                                             + getEdgeWeight( v, w );
195                         pathTuple =
196                           new Tuple<Vertex<T>, Double>(w, minCostToW);
197                         pq.enqueue( pathTuple );
198                     }
199                 }
200             }
201             // if the loop terminates naturally, we never found
202             // the destination vertex, so return failure
203             return -1;
204         }
205     }
```

An implementation for the minimal path algorithm pseudocoded in Section 13.4.1 is given in `AdjMatrixWeightedGraph` on lines 121–204). The implementation uses a new class, `Tuple`, which is like the `Pair` class developed in Chapter 2, except `Tuple` supports storing a pair of elements of *different* types. The `PriorityQueue` (lines 150–157) stores tuples consisting of a `Vertex` and the cost to get to that vertex from the `src` vertex. This tuple corresponds to the ⟨*Vertex, minCostToVertex*⟩ from the pseudocoded algorithm. The `Comparator` defined on lines 134–148 is used by the `PriorityQueue` to compare the path cost field of `Tuple` instances.

A `Vertex` is marked as visited by placing it in an array of visited vertices (see lines 131–132, 175, and 191). The main work of the algorithm (lines 165–200) follows closely from the pseudocode.

## SUMMING UP

The implementation of a hierarchy of graph classes makes careful use of inheritance and the `protected` access modifier to maximize code reuse. The `Graph` interface supports directed and undirected *un*weighted graphs and makes no assumptions about the underlying representation of the graph. The `AdjMatrixDiGraph` class provides a directed graph implementation of `Graph`, using an adjacency matrix to represent the graph. Class `AdjMatrixGraph` extends `AdjMatrixDiGraph` to provide an undirected graph. The methods in `AdjMatrixGraph` to create and remove edges extend and use their superclass counterparts to create/remove the unidirectional edge, and then finish the operation by creating/removing the reverse direction edge. Class `WeightedAdjMatrixGraph` extends `AdjMatrixGraph` and implements interface `WeightedGraph` to provide a weighted, undirected graph. `WeightedAdjMatrixGraph` must override the `addEdge()` method it inherits to provide the edge a weight. All other inherited methods can be used as is. `WeightedAdjMatrixGraph` adds several methods that deal with edge weights.

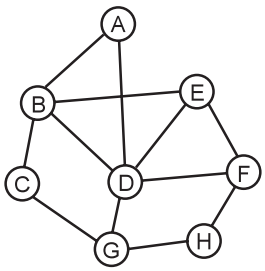
## EXERCISES

**Review**

1. Describe the similarities and differences between a graph and a tree.
2. How is a directed graph different from an undirected graph? Draw an example of each.
3. What does it mean for two vertices to be adjacent? Does it matter if the graph is directed or not?
4. What might the weights in a weighted graph represent?
5. What is a simple path? How do you determine its length? How is a simple path different from a cycle?

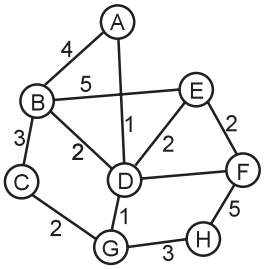
6. What does it mean to say a graph is connected? Is it possible for a graph to have more than one connected component? If so, draw such a graph.
7. How many edges are there in a complete graph?
8. Briefly describe the similarities and differences between the adjacency matrix and adjacency list representation of a graph. What are their space complexities?
9. How are the adjacency matrix and adjacency list representations different for directed and undirected graphs?
10. If the graph is weighted, how does that affect an adjacency matrix representation? An adjacency list representation?
11. Briefly describe the difference between depth first and breadth first search of a graph.
12. What does it mean to mark a vertex? Why would you need to do this?
13. Why is the time complexity of DFS guaranteed to be $\theta(|V|^2)$ if the graph is represented as an adjacency matrix? How does this change if an adjacency list is used?
14. Why is a stack a good collection to use for DFS? Why is a queue a good collection to use for BFS?
15. What is a spanning tree of a graph?
16. How can the DFS and BFS algorithms be modified to produce a spanning tree?
17. What is a greedy algorithm?
18. In what way is the minimal cost path algorithm greedy?
19. What is a minimal spanning tree?
20. How is the MST algorithm greedy?
21. The implementation presented in this chapter had an undirected graph as a subclass of a directed graph. What are the implications of this when adding an edge to a graph? If the inheritance were reversed so that a directed graph was a subclass of an undirected class, what would the implications be for adding an edge to a graph?
22. How is the `Tuple` class different from the `Pair` class?

**Paper and Pencil**




Graph $G_{12}$.

1. Show the order in which DFS and BFS would visit graph $G_{12}$.
2. Apply the shortest path algorithm to find the shortest path from A to G.
3. Apply the spanning tree algorithm to graph $G_{12}$.




Graph $G_{13}$.

4. Apply the minimal path algorithm to get the minimal cost path from B to H in graph $G_{13}$.
5. Show the minimal spanning tree found by the MST algorithm for graph $G_{13}$ starting with vertex C.

**Modifying the Code**

1. How might the idea about the representation for graphs be used to represent dense and sparse matrices?
2. Instead of decorating an edge with a weight (a `double`), David wants to use a `String`. What changes have to be made to the interface to support this? What changes need to be made to the internal representation of vertices and edges?
3. Instead of decorating an edge with just a weight (a `double`), Amir needs a weight (a `double`) *and* a name (a `String`). What changes have to be made to the interface to support this? What changes need to be made to the internal representation of vertices and edges?
4. Instead of decorating an edge with just a weight (a `double`), Felice needs a weight (a `double` representing distance), a route number (an `int`), and a name (a `String`). What changes have to be made to the interface to support this? What changes need to be made to the internal representation of vertices and edges?
5. How would you redesign the graph interface so that you can accommodate Gary, Amir, *and* Felice's needs? (See the last three problems.) Provide the following:
   a. A modified `Graph` interface
   b. A test plan for the new feature
   c. An implementation of one of the concrete classes
   d. A short paper describing the changes that had to be made, a rationale for your approach, and an evaluation of the difficulty of changing an existing piece of code (consider such things as reading and understanding the existing code, identifying what needed to be changed, and so on)

6. Provide a non-recursive version of `depthFirstSearch()`.
7. In her implementation, Yvonne mistakenly used a priority queue that allows duplicates. That is, there could be two entries ⟨D, 5⟩. Will the minimal path algorithm still work?
8. Method `AdjMatrixDiGraph.removeVertex()` tests to see if an entry represents an edge (is equal to 1) and if so, clears it (sets it to 0). Would the method be more efficient if it just walked through and cleared the appropriate row and column of the adjacency matrix? Would this work?
9. The implementation of `AdjMatrixDiGraph` has a few "opportunities for improvement." Make the following changes to the class:
   a. Currently the maximum number of vertices is fixed at 10. There is no mechanism for the client to specify how many vertices to use, nor is it possible to expand the size of the graph dynamically. Provide a constructor that takes a positive integer as an argument (you may want to specify an upper bound on the size, but be sure to document it) and creates a `vertices` array and an `adjMatrix` two-dimensioned array for this number of vertices.
   b. Provide support that will allow the dimensions of the graph to increase dynamically if an attempt to add another vertex is made when the `vertices` array is full. How much more difficult would it be to shrink the arrays if the graph goes from being full and large to being much smaller?
   c. The `getFreeVertexPosition()` and `getVertexNumber()` utility methods return −1 to indicate failure. Provide meaningful symbolic constants instead.
10. Traian is tired of getting exceptions when he tries to delete a vertex or edge that isn't in a graph or when he tries to add an edge to a graph and one or the other of the vertices isn't in it. He argues that if the class is going to throw an exception for these things, it ought to provide the programmer a way to check that a vertex or edge is in a graph. That is, there ought to be something like `hasVertex( Vertex v )` and `hasEdge( Vertex v1, Vertex v2 )` predicate methods. Add these methods to the ADT, provide test cases for them, and then add them to the implementation.
11. The minimal path algorithm finds the cost of the least expensive path from the source to the destination vertex (if such a path exists), but it doesn't determine which vertices are in that path. Modify the pseudocode to return an object that consists of the vertices in the path from the source to the destination (in the order they would be traversed while following the path), and the cost of the path. *Hint*: Instead of storing vertices in `verticesInSomePathToDest`, store vertex pairs where each pair represents an edge in a possible path to the destination.
    a. What is the cost of your algorithm?
    b. Modify the book's implementation to include your additions.
12. Provide a test plan and implementation for the minimal spanning tree algorithm.