19 Graphs

Overview

This chapter covers the concepts of graphs and networks. It uses object classes developed in earlier chapters: stacks, queues, priority queues, and collections.

- Section 19.1 describes the general abstraction of a graph and an application of graphs.
- Sections 19.2-19.4 present two different standard implementations of graphs, as well as a basic algorithm for a topological sort of the vertices of a graph.
- Section 19.5 discusses traversals of graphs, both depth-first and breadth-first. It solves the classic Traveling Salesman problem in graph theory.
- Sections 19.6-19.8 introduce the concept of a weighted graph (a network) and develop several standard algorithms for them: Kruskal's, Prim's, and Dijkstra's. Kruskal's algorithm requires UnionFind algorithms, so they are included here.
- Section 19.9 is just a bit of dynamic programming to finish the chapter up nicely.

19.1 The Hamiltonian Software

You have been hired to develop a game named Hamiltonian, a thinking game for children. It displays 20 to 25 small graphic images on the screen (we call these the **vertices**) and a large number of lines, each line connecting two of the vertices (we call these the **edges**). Each time the game is played, the number of vertices and the particular edges drawn change. The game is most interesting when each vertex has an average of 3 to 4 edges drawn to it.

One of the vertices is labeled END. The player clicks on a vertex connected (by an edge) to the END vertex, then clicks on a second vertex connected to the vertex chosen first, then clicks on a third vertex connected to the vertex chosen second, etc., until the player chooses to end play by clicking on END. Each vertex is green originally, but becomes red when clicked. A "bad click" is a vertex previously clicked or a vertex (other than END) not connected to the one previously clicked (or to END if it is the first vertex clicked). The bad clicks are ignored in the play. The score is 3 times the number of vertices clicked minus the number of elapsed seconds minus the number of bad clicks, plus a 20 point bonus if all of the vertices are clicked (forming what is called a **Hamiltonian path**), plus another 10 point bonus if the vertex clicked just before clicking END is connected to END (forming what is called a **Hamiltonian circuit**).

To plan this game software, we concentrate here on the graph-handling methods needed. A **graph** is a set of vertices and edges. An edge is a connection from one vertex to another. A **path** in the graph is a sequence of edges where the vertex at the end of each edge is the vertex at the beginning of the next edge. A graph is **connected** if you can get from any one vertex to any other vertex by following edges in the graph, i.e., there is a path from any one vertex to any other vertex.

The client wants the game software to first create a graph by choosing edges at random until you have enough for the game, eliminating choices that make the graph connected (so the game will not be too easy). Once you have enough edges, you then add one edge at a time until the graph becomes connected. So you need a way of testing the graph to see if it is connected yet, and you need to be able to tell whether two given vertices are connected to each other. The methods in the **HamGraph** class of Listing 19.1 (see next page) seem appropriate for a start. Figure 19.1 shows a possible game layout, with 20 vertices and 30 edges.

Listing 19.1 The HamGraph class of objects, stubbed partial documentation

```
public class HamGraph  // extends some Graph implementation
{
   /** Create a graph on n vertices with at least 1.5 * n
    *  randomly chosen edges, but the graph cannot be connected
    *  (it must have at least two components). */

   public void createUnConnectedGraph (int n)
   {
   } //===========================================

   /** Add 1 randomly chosen edge at a time until this graph
    *  becomes connected.  */

   public void makeTheGraphConnected()
   {
   } //===========================================

   /** Tell whether this graph is connected. */

   public boolean isConnected()
   {  return false;
   } //===========================================
}
```

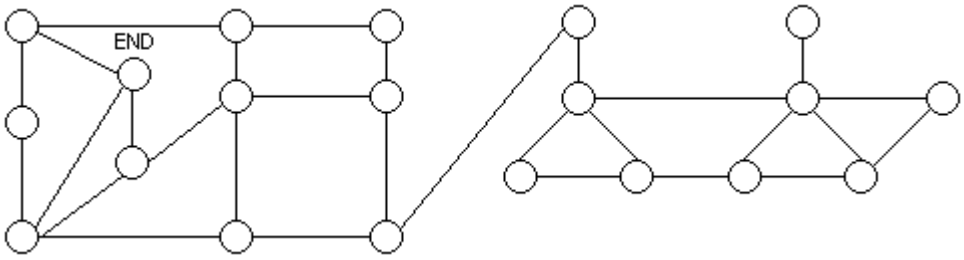


Figure 19.1 Possible game layout for the Hamiltonian software

Generic Graphs

The methods in the HamGraph class are specific to this game problem. We will be using graphs to solve various problems in this chapter, so we need to develop a more general definition of graphs, analogous to the Sun standard library Collection and Map types. Since Sun does not supply one, we develop it ourselves in this section. We will then use this Graph class in the HamGraph class.

A graph has a specific positive number of vertices, numbered consecutively with positive integer id numbers from 1 on up. We need to be able to create a graph, initially with no vertices, and ask a graph how many vertices it has, so the **Graph** class should have the following methods:

```
   public Graph()  // no edges, no vertices
   public final int getNumVertices() // tells number of vertices
```

We also need a class of Vertex objects to go with these Graph objects. The Graph should allow us to add a Vertex object with a given id number if no Vertex already has that number, to get access to the Vertex object with a given id number, and to create an Iterator object that goes through the collection of Vertex objects.

The Graph class with the methods described so far is in Listing 19.2. Note that three of these Graph methods are final methods, which means that subclasses cannot override them. So calls of these methods execute faster due to early binding. By contrast, some other Graph methods will be abstract, which means that subclasses must override them. Overriding `addVertex` is optional in subclasses of Graph.

Listing 19.2 The Graph class of objects with edge-related methods postponed

```
import java.util.ArrayList;
import java.util.Iterator;

public abstract class Graph
{
   private ArrayList itsVertices = new ArrayList();


   /** Create a graph with zero vertices and zero edges.  The
    *  vertices always have ids ranging 1...getNumVertices(). */

   public Graph()
   {  super();   // to remind you of the default constructor
   } //===========================================


   /** Return the current number of vertices. */

   public final int getNumVertices()
   {  return itsVertices.size();
   } //===========================================


   /** Return the Vertex object for the given id.  Throw a
    *  RuntimeException if the id is not 1...getNumVertices(). */

   public final Vertex getVertex (int id)
   {  return (Vertex) itsVertices.get (id - 1);
   } //===========================================


   /** Return an iteration over all Vertex objects. */

   public final Iterator vertices()
   {  return itsVertices.iterator();
   } //===========================================


   /** Add the given vertex to the graph if it has no vertex with
    *  the same id number; ignore non-positive ids.  Add vertices
    *  as needed for all positive ints below id, with null data.
    *  Throw a RuntimeException if the given vertex is null. */

   public void addVertex (Vertex v)
   {  if (v.ID > itsVertices.size())
      {  for (int k = itsVertices.size() + 1;  k < v.ID;  k++)
            itsVertices.add (new Vertex (k, null));
         itsVertices.add (v);  // added at the end of the list
      }
   } //===========================================
}
```

Reminder An **ArrayList**, discussed in some detail in Section 7.11, is a Sun standard library implementation of Collection, so it has the `size`, `add`, `contains`, `remove`, and `iterator` methods that any Collection has, along with the following additional array-like methods (this is all you need to know about ArrayLists to understand this chapter):

```
public set (int n, Object ob)  // replace at component n
public get (int n)   // return the object at component n
```

Edges in the Graph class

If a graph has only vertices and no edges, it is not of much use. We need to have some Edge objects as well. A graph should be able to add an Edge running from one Vertex to another Vertex, to tell whether there already exists an Edge running from one Vertex to another, to remove a given Edge, and to clear out all existing Edges. With these methods, you can easily implement HamGraph as an extension of some concrete implementation of the Graph class. For instance, add the following private method to HamGraph to make coding `createUnConnectedGraph()` easy. Assume that you will have an Edge constructor with two Vertex parameters:

```
/** Add a randomly chosen edge not already in the graph. */

private void addEdgeChosenRandomly()  // in HamGraph
{  int one = 1 + (int) (Math.random() * getNumVertices());
   int two = 1 + (int) (Math.random() * getNumVertices());
   Edge e = new Edge (getVertex (one), getVertex (two));
   if (this.contains (e))
      addEdgeChosenRandomly();
   else
   {  this.add (e);
      this.add (new Edge (getVertex (two), getVertex (one)));
      // in the opposite direction, for an undirected graph
   }
} //===============================================
```

Note that the `addEdgeChosenRandomly` method always adds a pair of Edges, one in each direction between `one` and `two`. That is because the game software calls for an **undirected graph**, i.e., a graph where there is an Edge from x to y if and only if there is an Edge from y to x. If a graph does not have this property, it is a **directed graph**. If a graph is undirected, then the **out-degree** of each vertex (defined to be the number of Edges exiting the vertex) always equals the **in-degree** of that vertex (defined to be the number of Edges coming in to that vertex).

The following method is a horribly inefficient algorithm for the HamGraph method, but it works. Let it go at that for now; once you have the beta version of the game ready, you can come back and improve this. Assume you have an appropriate Vertex constructor:

```
/** Create a graph on n vertices with at least 1.5 * n
 *  randomly chosen edges but not a connected graph. */

public void createUnConnectedGraph (int n)  // in HamGraph
{  this.clear();
   this.addVertex (new Vertex (n, null));
   for (int k = 1;  k < 1.5 * this.getNumVertices();  k++)
      this.addEdgeChosenRandomly();
   if (this.isConnected())
      this.createUnConnectedGraph (n);
} //===============================================
```

The Graph class also needs an Iterator that goes through the sequence of Edges that exit a given Vertex and a method to clear all edges from the graph. Listing 19.3 describes the additional Graph methods more precisely. All but one are abstract, to be overridden in any concrete implementation of Graph.

Listing 19.3 The rest of the Graph methods, six to override

```
// public abstract class Graph continued

   /** Remove all edges from the Graph.  Leave the vertices. */
   public abstract void clear();

   /** The following methods throw a RuntimeException if the
    *  parameter is null or its vertex ids are not in the range
    *  1...getNumVertices(), except they may simply ignore 0.
    *  Do not use a Graph iterator to remove anything. */

   /** Tell whether this graph contains the given Edge. */
   public abstract boolean contains (Edge given);

   /** Remove the given Edge if it is in there and return true.
    *  But if the Edge is not in there, simply return false. */
   public abstract boolean remove (Edge given);

   /** Add the given Edge if it is not in there and return true.
    *  But if the Edge is in there, simply return false. */
   public abstract void add (Edge given);

   /** Return an iteration over all edges that exit Vertex v. */
   public abstract Iterator edgesFrom (Vertex v);

   /** Return the number of Edges that end at that Vertex. */
   public abstract int inDegree (Vertex given);

   /** Return the number of Edges that start at that Vertex. */

   public int outDegree (Vertex given)
   {  Iterator it = edgesFrom (given);
      int count = 0;
      while (it.hasNext())
      {  it.next();
         count++;
      }
      return count;
   } //============================================
```

Exercise 19.1 Write the `makeTheGraphConnected()` HamGraph method.
Exercise 19.2 (harder) Write a generic method that tells the in-degree of a given vertex.
Exercise 19.3* Write a Graph method that removes all the Edges exiting a given vertex.
Exercise 19.4* Explain how both `getVertex` and `addVertex` automatically throw the Exceptions that their comment descriptions say they do.
Exercise 19.5* Assume that MatrixGraph is a concrete implementation of Graph. Write an independent method that creates a connected MatrixGraph with a given number n of Vertices (the parameter) and 1 less Edge than it has Vertices.
Exercise 19.6* Write a Graph method that tells the total number of edges in the Graph.
Exercise 19.7* Write a Graph method that adds enough edges so that the vertex with ID 1 connects to every other vertex. Return the number of new edges that were added.
Exercise 19.8** Write a generic method that tells whether the graph is directed. Assume each Edge has public final instance variables TAIL and HEAD of Vertex type.

19.2 The Adjacency Matrix Implementation Of Graphs

A Vertex knows its id number and some associated data such as its name or color or graphical icon. In addition, we will find it useful to be able to assign a marker number to a Vertex from time to time. This marker number is always initialized to zero. A suitable definition is in the upper part of Listing 19.4. Note that we can make `ID` and `DATA` available for public use without violating the encapsulation principle, since they are unchangeable ("final") once the object has been constructed.

Listing 19.4 The Vertex and Edge classes of objects

```
public class Vertex extends GraphPart
{
   public final int ID;
   public final Object DATA;   // identifying info, e.g., a name


   public Vertex (int id, Object data)
   {  ID = id;
      DATA = data;
   } //===============================================
}
//#####################################################


public class GraphPart
{
   private int itsMark = 0;


   public final int getMark()
   {  return itsMark;
   } //===============================================

   public final void setMark (int given)
   {  itsMark = given;
   } //===============================================
}
//#####################################################


public class Edge extends GraphPart
{
   public final Vertex TAIL;  // vertex where it begins
   public final Vertex HEAD;  // vertex where it ends


   public Edge (Vertex from, Vertex to)
   {  TAIL = from;
      HEAD = to;
   } //===============================================

   public boolean equals (Object ob)
   {  return ob instanceof Edge && this.HEAD == ((Edge) ob).HEAD
                                && this.TAIL == ((Edge) ob).TAIL;
   } //===============================================
}
```

The marker part of the Vertex class is separated out so it can be used by the Edge class too. That is, we have both the **Vertex** class and the **Edge** class inherit from the **GraphPart** class, which provides services to set and inspect the current value of the int marker value on the Vertex or Edge, whichever the case may be. In practice, we mostly mark a Vertex or Edge with the number 1 to indicate something special about it and later change the mark back to 0.

Edge objects are also straightforward. An Edge runs from one Vertex, its **tail**, to another Vertex, its **head**. An `equals` method is occasionally needed; two Edges are equal if they have the same head and the same tail, even if the marking numbers are different. The lower part of Listing 19.4 defines the Edge class.

The adjacency-matrix design

One popular way of implementing a graph is to use a two-dimensional matrix: We make `edgeAt[x][y]` be the Edge going from `x` to `y` if the graph has that edge, otherwise `edgeAt[x][y]` is null.

One restriction is that it is quite time-consuming to add more vertices beyond the original amount, since that would require replacing the existing two-dimensional array by a larger one and copying all the Edge values into the new one. So this MatrixGraph implementation overrides `addVertex` to keep it from accepting vertex id numbers larger than the original number of vertices. As the upper part of Listing 19.5 shows (see next page), the constructor requires that you specify the maximum number of vertices.

For this book's particular design of the Graph concept, we never remove any vertices once we add them to the graph, so the `clear` method only clears out Edges. It does this by assigning null to each component of the matrix. Since an edge of a graph tells which two vertices are adjacent, and since we use a two-dimensional matrix to store the adjacency information, MatrixGraph is called the **adjacency-matrix implementation** of Graph. The next section discusses the use of a list of these adjacencies for each vertex; that implementation of Graph is called the **adjacency-list implementation** of Graph.

The real problem for MatrixGraph is to get an Iterator object that goes through the Edges exiting a given vertex id number. This method is left as an exercise; two alternatives are proposed. Whatever you do, iteration will execute in at least big-oh of N time for the MatrixGraph implementation. Consider that most graphs you run across in real life tend to have 3 to 6 Edges per vertex on average. Big-oh of N is a lot worse than big-oh of 6, which is what you get with the adjacency-list implementation discussed shortly.

The `contains`, `remove`, and `add` methods all have straightforward coding and execute in big-oh of 1 time. The `addVertex` method executes in big-oh of 1 time averaged over all vertices. Figure 19.2 illustrates this matrix structure. A picture of a graph is on the left and the corresponding matrix is on the right.

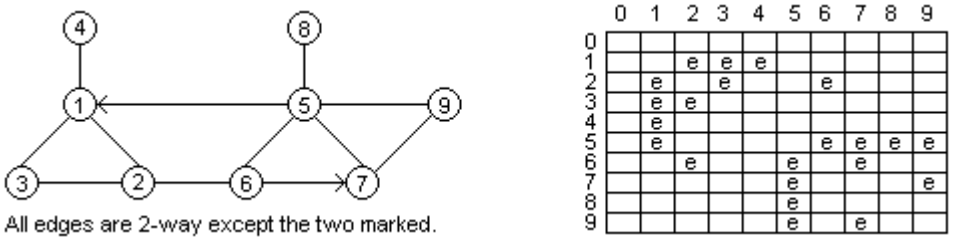


Figure 19.2 The adjacency-matrix representation of a Graph

Listing 19.5 The MatrixGraph class of objects

```
public class MatrixGraph extends Graph
{
   private Edge[][] edgeAt;


   /** Specify the maximum number of vertices the graph can have.
    *  This number cannot be changed by later method calls. */

   public MatrixGraph (int maxVerts)
   {  super();
      edgeAt = new Edge[maxVerts + 1][maxVerts + 1]; // all null
   } //===========================================


   public final void addVertex (Vertex v)
   {  if (v.ID >= edgeAt.length)
         throw new IllegalArgumentException ("not with matrix");
      super.addVertex (v);
   } //===========================================


   public final void clear()
   {  int maxVerts = edgeAt.length;
      edgeAt = new Edge[maxVerts + 1][maxVerts + 1]; // all null
   } //===========================================


   public final boolean contains (Edge given)
   {  return edgeAt[given.TAIL.ID][given.HEAD.ID] != null;
   } //===========================================


   public final boolean remove (Edge given)
   {  Edge e = edgeAt[given.TAIL.ID][given.HEAD.ID];
      edgeAt[given.TAIL.ID][given.HEAD.ID] = null;
      return e != null;
   } //===========================================


   public final void add (Edge given)
   {  edgeAt[given.TAIL.ID][given.HEAD.ID] = given;
   } //===========================================


   // the following two are left as exercises
   public final java.util.Iterator edgesFrom (Vertex v)
   public final int inDegree (Vertex given)
}
```

As an example of additional methods that could be in the MatrixGraph class, you could have a method that tells whether a given vertex has a connection both to and from any other vertex, using the standard some-A-are-B logic:

```
public boolean hasTwoWayEdge (Vertex given) // in MatrixGraph
{  for (int k = 1;  k <= getNumVertices();  k++)
   {  if (edgeAt[given.ID][k] != null
              && edgeAt[k][given.ID] != null)
         return true;
   }
   return false;
}  //=======================================
```

Exercise 19.9 Write a MatrixGraph method that removes all the Edges exiting a given vertex, by directly accessing the two-dimensional array.
Exercise 19.10 Write a MatrixGraph method that tells the out-degree of a given vertex id by directly accessing the two-dimensional array.
Exercise 19.11 Write a MatrixGraph method that tells the in-degree of a given vertex id by directly accessing the two-dimensional array.
Exercise 19.12 (harder) Implement the `edgesFrom` method by creating an ArrayList object, adding to it all the non-null Edge values in the given row, then returning its Iterator object.
Exercise 19.13 (harder) Add an instance variable to the MatrixGraph class of objects and coding so that one can find the in-degree of a vertex simply by calling the following: `public int inDegree (Vertex given) { return itsIns[given.ID]; }`
Exercise 19.14* Write a Graph method that resets the mark on every Vertex to zero.
Exercise 19.15* Write a Graph method that resets the mark on every Edge to zero.
Exercise 19.16* Write a Graph method that tells how many edges have a tail whose ID is a smaller number than its head's ID.
Exercise 19.17* Write a MatrixGraph method `public Graph copy()`: The executor creates and returns an exact duplicate of itself.
Exercise 19.18* Write a MatrixGraph method `public Graph transpose()`: The executor returns a new MatrixGraph object that has the same vertices but the reversed edges, i.e., an edge from v to w is in one if and only if an edge from w to v is in the other.
Exercise 19.19** Implement the `edgesFrom` method by using a private Iterator class nested in the MatrixGraph class. This executes faster than the way described in the earlier exercise, if each such Iterator has an instance variable `itsPos` that keeps track of the index of the next available Edge object.

19.3 The Adjacency List Implementation Of Graphs

Another popular way of implementing a graph is to keep, for each vertex id number, a list of all the Edges exiting the Vertex of that id number. The list for one vertex is only accessed sequentially, not random-access like an array. However, the list of all these Edge lists is random-access, i.e., it can be directly accessed by the index number of a Vertex. This makes some graph operations execute faster, but other graph operations execute slower, than in the adjacency matrix implementation.

We use an ArrayList named `itsList` to random-access the Edge lists, one list of Edges for each Vertex id number. We do not use the zeroth component of this main list. Any Collection type would be fine for each individual Edge list, but for convenience we also use ArrayLists for them. Listing 19.6 (see next page) gives this concrete ListGraph subclass of Graph.

The ListGraph constructor puts a dummy value in for the zeroth component of `itsList`, so that a Vertex with id `k` will have its edge list at index `k`. We need to override the `addVertex` method from the basic Graph class to add coding that allows for the list of edges. First we execute the basic `addVertex` method from the Graph class that we override (the statement `super.addVertex(v)`). Then we go on to add one new empty list of Edges for each of the new vertices.

Listing 19.6 The ListGraph class of objects

```
import java.util.*;

public class ListGraph extends Graph
{
   private ArrayList itsList = new ArrayList();

   public ListGraph()
   {  super();
      itsList.add (null);  // we do not use the value at index 0
   } //=============================================

   public final void addVertex (Vertex v)
   {  super.addVertex (v);
      if (v.ID >= itsList.size())
      {  for (int k = itsList.size();  k <= v.ID;  k++)
            itsList.add (new ArrayList());
      }
   } //=============================================

   /** Remove all edges from the graph, but not the vertices. */

   public final void clear()
   {  for (int k = 1;  k <= getNumVertices();  k++)
         itsList.set (k, new ArrayList());
   } //=============================================

   public final boolean contains (Edge given)
   {  return ((Collection) itsList.get (given.TAIL.ID))
               .contains (given);
   } //=============================================

   public final boolean remove (Edge given)
   {  return ((Collection) itsList.get (given.TAIL.ID))
               .remove (given);
   } //=============================================

   public final void add (Edge given)
   {  if (given.HEAD.ID < 1 || given.HEAD.ID > getNumVertices())
         throw new IllegalArgumentException ("bad vertex");
      Collection z = (Collection) itsList.get (given.TAIL.ID);
      if (z.contains (given))
         z.remove (given);
      z.add (given);
   } //=============================================

   public final Iterator edgesFrom (Vertex v)
   {  return ((Collection) itsList.get (v.ID)).iterator();
   } //=============================================
}
```

Remember that, if e.g. we add a Vertex with id number 10 at a time when `getNumVertices()` is 6, we have to add four more vertices (numbered 7, 8, 9, 10), not just one. That is, we always have one vertex for every id number from 1 up to `getNumVertices()` inclusive.

For a given Vertex v, `(Collection)itsList.get(v.ID)` is the Collection object (specifically an ArrayList object) that contains all the edges exiting v. So you can use its iterator to go through the list of all edges that exit from that vertex. The `edgesFrom` method simply returns that Iterator object. Figure 19.3 shows a graph and the corresponding ArrayLists that the ListGraph implementation would have.

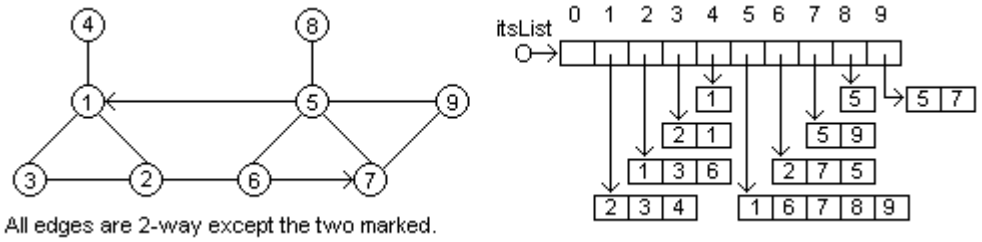


Figure 19.3 The adjacency-list representation of a Graph

A given Edge exits from the vertex `given.TAIL`, so `itsList.get(given.TAIL.ID)` is the Collection object that contains all the edges exiting `given.TAIL`. So ListGraph's `contains(Edge)` method simply checks whether that Collection object contains an Edge that equals the given Edge, and its `remove(Edge)` method simply has that Collection object remove the given Edge. Both should throw an Exception if their vertices have ID numbers outside the range `1...numVertices()`; this is left as an exercise.

The `add(Edge)` method is trickier because an ArrayList always adds the value it is given regardless of whether it is already in there. So you have to first check whether the Edge is already there before you add it. If so, you should delete the current one first, which may have a different mark on it. Also, you have to throw a RuntimeException if the vertex ids in the given Edge are not from 1 to `getNumVertices()`. That happens "automatically" if the list of Edges for the given Edge's tail does not exist, but you have to check separately that the given Edge's head is also in the allowable range.

Note that the coding in Listing 19.6 follows the general coding principle that you create a variable to store a value only if that value will be used later more than once. In the `add` method, the value of `z` is used several times in the statement after it is assigned. In the `contains` and `remove` method, the value returned by `itsList.get` is only used once, as the executor of a Collection method, so we do not assign the value to a variable. The generic `indegree` method of Exercise 19.2 is a reasonable coding for ListGraph.

Exercise 19.20 Revise the `contains` and `remove` methods in Listing 19.6 to throw an Exception when the vertex indexes are outside the range `1...numVertices()`.
Exercise 19.21 Write a ListGraph method that tells the out-degree of a given vertex, by directly accessing the ArrayList corresponding to the given vertex id.
Exercise 19.22 (harder) Write a ListGraph method that removes all the Edges exiting a given vertex v whose head has a larger ID than v, by directly accessing the ArrayLists.
Exercise 19.23* Rewrite the `clear` method in Listing 19.6 to clear out all the values in the ArrayList objects that are already in the list, rather than replacing them by new empty ArrayList objects.
Exercise 19.24* Rewrite the `add` method in Listing 19.6 to execute much faster, using indexes in the ArrayList.
Exercise 19.25** Write a ListGraph method `public Graph copy()`: The executor creates and returns an exact duplicate of itself.
Exercise 19.26** Write a ListGraph method `public Graph transpose()`: The executor returns a new ListGraph object with the same vertices but with the reversed edges, i.e., an edge from v to w is in one if and only if an edge from w to v is in the other.

19.4 Topological Sorting; Big-Oh For Graph Algorithms

There are many applications where a directed graph tells which activities must be performed before which other activities (an edge from v to w indicates that v must be performed before w). **Critical-path analysis** tries to find optimal scheduling of tasks in such situations. Scheduling college courses to satisfy prerequisites is an example.

Listing the vertices of a graph in a sequence so that, for each edge from some vertex v to some other vertex w, v comes before w in the sequence, is a **topological sort** of the vertices. Then the activities to be scheduled can be performed in the order that their vertices appear on that list.

First we obtain a list of those vertices that have no edges coming in to them. If none of these "starter vertices" exist, then the vertices cannot be sorted. The method in the upper part of Listing 19.7 has a queue as the parameter, initially empty; the method fills the queue with all of the "starter vertices" available. It uses the `inDegree` method. It also marks each non-starter vertex with the number of edges that come into it.
Reminder: The three basic QueueADT methods for queues are as follows:

- `q.enqueue(v)` to add a value v;
- `q.dequeue()` to remove a value and return it (the return type is Object); and
- `q.isEmpty()` to see whether any values are left.

Listing 19.7 Methods used in topological sorting

```
private void getStarters (QueueADT toCheck)
{  Iterator verts = vertices();
   while (verts.hasNext())
   {  Vertex v = (Vertex) verts.next();
      int numPredecessors = inDegree (v);
      if (numPredecessors == 0)
         toCheck.enqueue (v);
      else
         v.setMark (numPredecessors);
   }
}  //=========================================


private void topoSort (QueueADT toCheck, QueueADT sorted)
{  getStarters (toCheck);
   while ( ! toCheck.isEmpty())   // once per Vertex
   {  Vertex ok = (Vertex) toCheck.dequeue();
      sorted.enqueue (ok);
      Iterator edges = edgesFrom (ok);
      while (edges.hasNext())
      {  Vertex v = ((Edge) edges.next()).HEAD;
         v.setMark (v.getMark() - 1);
         if (v.getMark() == 0)
            toCheck.enqueue (v);
      }
   }
}  //=========================================
```

To illustrate, the `getStarters` method applied to the directed graph in Figure 19.4 would put the vertices numbered 4 and 8 on the `toCheck` queue and set the mark on each other vertex to its in-degree. So the number at the left of each vertex is the mark for that vertex.

The directed graph at right has vertices numbered 1 to 9. To the left of each vertex is its in-degree -- the number of edges that have that vertex as the head.

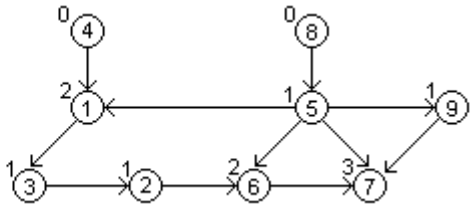


Figure 19.4 A directed graph with in-degrees specified

The topoSort method

Once we have the starter vertices put on a queue, we can go through the elements of that queue and do two things with each vertex we take from the queue: shift that vertex to the rear of a different queue of sorted vertices, and reduce the mark on every other vertex to what its in-degree would be if the shifted vertex were not in the graph.

For the graph in Figure 19.4, we would move vertex 4 from the `toCheck` queue to the sorted queue and reduce the mark on vertex 1 from 2 to 1. We would then move vertex 8 from the `toCheck` queue to the sorted queue and reduce the mark on vertex 5 from 1 to 0, which entitles vertex 5 to go on the `toCheck` queue.

In other words, we have the mark on any remaining vertex v be the number of edges coming into v from vertices that are not on the sorted queue. So when the amount marked on v is reduced to zero, that means that every vertex that comes before v is already on the sorted queue, so v is also a candidate for the sorted queue. This logic is coded in the lower part of Listing 19.7.

Continuing with the example of Figure 19.4, the only vertex now on the `toCheck` queue is vertex 5. We move it from the `toCheck` queue to the sorted queue and go through the edges exiting vertex 5, which reduces the marks on vertices 1 and 9 from 1 to 0, and reduces the marks on vertices 6 and 7 to 1 and 2, respectively. That puts vertices 1 and 9 on the `toCheck` queue. The rest of the process is left as an exercise.

These two methods are private because a client of the class would call some public method that would create the two queues, call the `topoSort` method, then print out the information stored in the sorted queue. The sorted queue will contain all the vertices if the Graph can be topologically sorted, otherwise it will be missing some vertices.

Big-oh for basic Graph algorithms

For this section and later in the chapter, let **NV** stand for the number of vertices in the graph under discussion and let **NE** stand for the number of edges in that same graph. The execution time for graph operations is generally a function of one or both of these numbers.

The `clear` method for ListGraph is a big-oh of NV operation, since it executes one operation for each vertex in `itsList`. The `clear` method for MatrixGraph is a big-oh of 1 operation, since it executes a single assignment. Check the coding in Listings 19.4 and 19.5 to verify this.

The `contains(Edge)` method for MatrixGraph is a big-oh of 1 operation, since it only executes three actions independently of NE or NV (okay, you could call it big-oh of 3, but we do not do that -- review the definition of big-oh to see why). The `contains(Edge)` method for ListGraph is a big-oh of NE/NV operation, since it directly accesses the ArrayList of the given edge's tail vertex but then searches sequentially through that ArrayList to find the specified edge. Thus the execution time is proportional to the number of edges that particular vertex has. On average, that will be NE/NV, because if you were to add up the sizes of each of the ArrayList adjacency lists, one per vertex, you would have a total of NE. The NE/NV value is called an **amortized average**.

The `remove(Edge)` and `add(Edge)` methods are also big-oh of 1 for MatrixGraph and big-oh of NE/NV for ListGraph, and for the same reasons, but we restrict our discussion in the rest of this section to graphs with a fixed number of edges and vertices.

Big-oh for the topological sort

The `getStarters` method in Listing 19.7 clearly takes big-oh of NV time plus whatever time calculation of the in-degrees takes. If these in-degree values are stored as the graph is built, it takes time proportional to NE (one or two extra operations each time a new edge is added). So big-oh for `getStarters` is NV+NE.

The `topoSort` method has a while-loop that executes once for each vertex. That while-loop contains an inner while-loop that executes once for each edge out of the current vertex. So the statements in the body of the inner while-loop are executed once for each edge in the graph. All of the other methods called in that coding can be done in big-oh of 1 time. So the overall execution time for the sorting process is NV+NE for `getStarters` followed by NV+NE for the actual sorting. The big-oh execution time is therefore big-oh of NV+NE.

Oops! That is not quite right. One pass through an `edgesFrom(v)` iterator requires NE/NV operations for ListGraph (i.e., the average number of edges per vertex), but it requires NV operations for MatrixGraph (the number of components in one row of the matrix). So for the adjacency-matrix implementation, a topological sort requires NV+NE followed by NV*NV, which is big-oh of NV-squared.

To see the difference, consider that many situations have a low average number of edges per vertex regardless of the number of vertices. For instance, it is common that NE ranges from 3 to 6 times the number of vertices, whether you have tens or hundreds or thousands of vertices. So a topological sort for ListGraph is big-oh of 6*NV, which is equivalent to big-oh of NV, whereas a topological sort for MatrixGraph is big-oh of NV^2.

Exercise 19.27* Complete the description of the topological sort process for Figure 19.4. You already have vertices 4, 8, and 5 on the sorted queue and vertices 1 and 9 on the `toCheck` queue.

Exercise 19.28* Write the public sort method that calls the `topoSort` method in Listing 19.7 and prints the information in the queue.

19.5 Graph Traversals

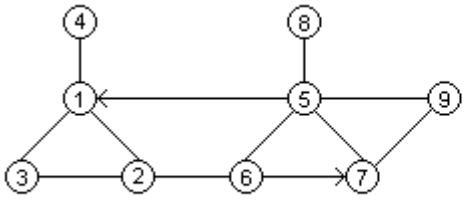
A **subgraph** of a graph G is a graph all of whose edges and vertices are in G, though not necessarily vice versa. A **subtree** of a graph G is a directed subgraph of G for which you can get from a specific vertex (the subtree's **root**) to any other vertex in the subtree by one and only one path in the subtree. The subtree is a **spanning tree** of G if it contains all the vertices of G.

A **depth-first traversal** of a graph is a processing of the vertices of some subtree of the graph starting with the root of the subtree, such that (a) no vertex is processed before its parent is processed, and (b) once a vertex v is processed, then all the vertices you can reach from v by following a path in the subtree are processed before any other vertices are processed. If this definition seems too abstract, a simple definition is that depth-first traversal is what you get from an appropriate recursive method such as the one to be described next.

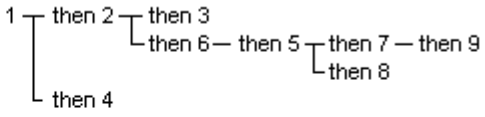
An example of a depth-first traversal is the `markReachables` method in the upper part of Listing 19.8 (see next page). It marks with 1 every vertex that can be reached from a given vertex. The 1 indicates that vertex has been "visited" by the algorithm. We leave the mark on the vertex until after the traversal is finished, to avoid visiting it twice.

All of the vertices marked 1 are in the traversal subtree, assuming that the initial call of the method was when all vertices of the graph were marked zero. The root is the first vertex marked 1. Also, any edge obtained in line 4 of that method whose head v satisfies `v.getMark() == 0` is to be considered part of the traversal subtree. So if you reach all the vertices of the graph with this method, you also get a spanning tree of the graph by recording those particular edges as you go.

An obvious problem with this recursive logic is that it leaves marks on some of the vertices, which should be removed before the method returns its answer. So that method should only be called by something like the public `isConnected` method in the middle part of Listing 19.8, which is the method needed by the Hamiltonian software of Section 19.1: It tells whether the graph is connected. Figure 19.5 gives the order in which vertices are visited for one particular graph using `markReachables`.



All edges are 2-way except the two marked.



So 1, 2, 3, 6, 5, 7, 9, 8, 4 in that order.
This order of depth-first traversal assumes that edgesFrom produces vertices in increasing order of id number.

Figure 19.5 Depth-first traversal of a graph starting from vertex #1

This basic depth-first traversal logic can be modified to solve several different problems. If you want a list of all edges in a spanning tree, assuming that the graph is connected, you could insert at line 6 (subordinate to the if-condition) a statement that puts that edge on a list of edges. Later you return that list as the answer. If you want to find out whether the graph has a **cycle** (a path of two or more vertices that starts and ends at the same vertex) involving the starting vertex, it will be when line 4 detects the starting vertex.

Listing 19.8 Traversal methods for the Graph class of objects

```
   /** Mark 1 on each vertex reachable from the given Vertex. */

   private void markReachables (Vertex given)
   {  given.setMark (1);                                    //1
      Iterator it = edgesFrom (given);                      //2
      while (it.hasNext())                                  //3
      {  Vertex v = ((Edge) it.next()).HEAD;                //4
         if (v.getMark() == 0)                              //5
            markReachables (v);                             //6
      }                                                     //7
   } //=============================================

   /** Tell whether you can reach each vertex from vertex #1. */

   public boolean isConnected()
   {  markReachables (getVertex (1));                       //8
      Iterator it = vertices();                             //9
      int count = 0;                                        //10
      while (it.hasNext())                                  //11
      {  Vertex v = (Vertex) it.next();                     //12
         if (v.getMark() == 1)                              //13
         {  count++;                                        //14
            v.setMark (0);                                  //15
         }                                                  //16
      }                                                     //17
      return count == getNumVertices();                     //18
   } //=============================================

   /** Find a path from given through all unmarked vertices and
    *  mark each vertex with its sequence number in that path. */

   private boolean solveSalesman (Vertex given, int seqNum)
   {  given.setMark (seqNum);                               //19
      if (seqNum == getNumVertices())                       //20
         return true;                                       //21
      Iterator it = edgesFrom (given);                      //22
      while (it.hasNext())                                  //23
      {  Vertex v = ((Edge) it.next()).HEAD;                //24
         if (v.getMark() == 0)                              //25
         {  if (solveSalesman (v, seqNum + 1))              //26
               return true;                                 //27
         }                                                  //28
      }                                                     //29
      given.setMark (0);                                    //30
      return false;                                         //31
   }  //=========================================
```

You could also use depth-first traversal to do a topological sort when you are certain the graph does not contain cycles: Recursively set the mark of each vertex to 1 plus the sum of all marks of vertices that it connects to. Then for any edge connecting v to w, v will have a larger mark than w and thus comes earlier in the "sorted" sequence.

Note that the following problems can be solved nicely with depth-first traversals:

1. You have a two-dimensional array of pixels and you have to start from a given white pixel and color it and all adjacent white pixels a given different color.
2. You have a two-dimensional representation of a maze and you have to find your way out of it from a given point.

The Traveling Salesman Problem

A classic problem in graph theory is to find a path from a given vertex that goes through every other vertex in the graph exactly one time. That is, you cannot pass through a vertex you have previously passed through. An application is to find the sequence of cities a traveling salesman should visit so that he or she does not go through the same city twice. So this is called the **Traveling Salesman Problem**.

The basic depth-first traversal logic can be adapted to this problem, but we have to actually say which vertices are visited in which order. The `solveSalesman` method in the lower part of Listing 19.8 is given a vertex that we can visit next and the number of that vertex in the path we are constructing. For instance, the second parameter is 5 if the given vertex is the fifth one in the sequence from the starting vertex. We would call it initially as `solveSalesman(startingVertex,1)`.

As an illustration, the `solveSalesman` method applied with the starting vertex numbered 4 in the earlier Figure 19.5 would set the mark on vertex 4 to 1, then set the mark on vertex 1 to 2, then set the mark on vertex 2 to 3 but fail to complete the path. So it would change the mark on vertex 2 back to 0 and set the mark on vertex 3 to 3. Next it sets the mark on vertex 2 to 4, then the mark on vertex 6 to 5. The rest of the process is left as an exercise.

Compare this logic step for step with the logic of `markReachables`. We of course assign consecutive numbers to the vertices as they are visited instead of 1 to each of them. This is why we pass `seqNum + 1` on the recursive call in line 26. Other than that, there are only a few differences:

- Lines 26-27 return `true` if a path is found in later calls, without re-setting the markers.
- Lines 20-21 return `true` if a path is found in this call, without re-setting the marker.
- Lines 30 and 31 are executed when we fail to complete the path from the given vertex. They set the mark back to zero and return `false` (since we cannot put the given vertex on the current path and manage to complete it).

Using a stack to store the sequence of visits

If a call of `solveSalesman` with an initial count of 1 returns `true`, we can easily find the sequence number for each vertex, but we do not have the vertices in the order they occur in that sequence. Perhaps a better solution would be to also pass a stack object as a parameter. We would then push a vertex on the stack when we mark it, and pop it from the stack when we set its mark back to zero.

Then the public method that calls the `solveSalesman` method tests whether the stack is empty when the method returns and, if not, prints the values in the order in which they come off the stack.

Breadth-first traversal

Suppose you would like to know the length of the shortest path from a given vertex v to each of the vertices you can reach from v. You would mark all the vertices that you can reach directly from v with a 1. Then you would mark with a 2 all the vertices you can reach in one more step, i.e., directly from a vertex marked 1. This keeps up until you have marked all the vertices possible.

This is easiest to do with a queue. The following method would be passed an initially empty QueueADT object. Compare it with the `markReachables` method in Listing 19.8 to see how it differs:

```
private void markDistance (Vertex given, QueueADT queue)
{  given.setMark (1);
   queue.enqueue (given);
   while (! queue.isEmpty())
   {  Vertex current = (Vertex) queue.dequeue();
      Iterator it = this.edgesFrom (current);
      while (it.hasNext())
      {  Vertex v = ((Edge) it.next()).HEAD;
         if (v.getMark() == 0)
         {  v.setMark (current.getMark() + 1);
            queue.enqueue (v);
         }
      }
   }
}  //=======================================
```

For the graph shown in the earlier Figure 19.5, the order in which this method visits the vertices is as follows, assuming the initial call is with vertex 1: visit 1, then put 2, 3, and 4 on the queue. So visit 2 next, which puts 6 on the queue after 3 and 4. Visit 3 next, which adds nothing to the queue. Visit 4 next, which also adds nothing to the queue, so the queue now just has 6 on it. Visit 6 next, which adds 5 and 7 to the queue. Visit 5 next, which adds 8 and 9 to the queue after 7. Finally, visit 7, then 8, then 9. So the overall sequence is 1,2,3,4,6,5,7,8,9.

All of these traversal algorithms mark the vertices they visit and, if they see a previously visited vertex, ignore it. This causes the algorithm to traverse only a subtree of the graph, not the whole graph. The advantage is that you avoid going around in circles, perhaps never terminating the algorithm.

Big-oh for depth-first or breadth-first traversal of a graph

The `markReachables` method in the upper part of Listing 19.8 is a prototypical depth-first traversal. Since a call of the method marks its parameter with a 1, and the method is never called when the parameter is already marked 1, it follows that the method is called at most once for each vertex. So the execution time is NV multiplied by the average execution time for one pass through the `edgesFrom` iterator.

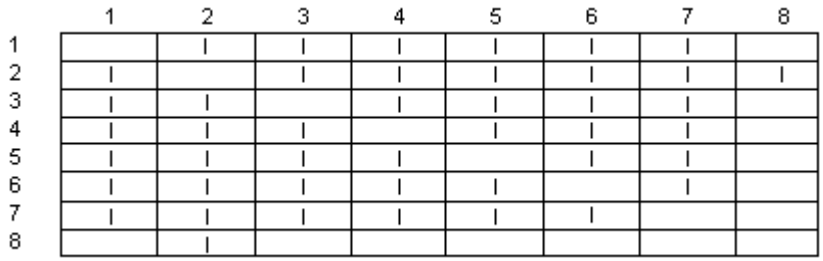
In general, traversal of a graph goes through each vertex one time and, at each vertex, goes through each edge out of that vertex one time. That is again big-oh of NV+NE for the adjacency-list implementation and big-oh of NV^2 for the adjacency-matrix implementation, just as for the topological sort.

Since hardly anyone deals with graphs that have much fewer edges than vertices, NE will generally be larger than NV. In such cases, the topological sort and depth-first traversal for adjacency lists are simply big-oh of NE, though they are NV^2 for an adjacency matrix.

Big-oh for the traveling salesman

The above analysis applies for true traversal of the vertices, where each vertex is visited but once, whether depth-first or breadth-first. But when you call the `solveSalesman` method in Listing 19.8 recursively, you mark the vertex, amble around for a while, then unmark it. That means that it can be visited again, perhaps many times. That can make the execution time exponential in the number of vertices.

For a specific example: Consider a graph of 8 vertices in which vertex #2 is the only vertex that connects to vertex #8, but every vertex numbered 1 through 7 connects to every other vertex numbered 1 through 7. Figure 19.6 (see next page) shows the adjacency matrix with a mark where there is an edge (the empty boxes are nulls; the components with indexes of zero are left out, since they are ignored).



	1	2	3	4	5	6	7	8
1		I	I	I	I	I	I	
2	I		I	I	I	I	I	I
3	I	I		I	I	I	I	
4	I	I	I		I	I	I	
5	I	I	I	I		I	I	
6	I	I	I	I	I		I	
7	I	I	I	I	I	I		
8		I						

Figure 19.6 Edges between vertices (only #2 connects to #8)

Then `solveSalesman (getVertex(1), 1)` goes through more than 5! (that is 5-factorial, i.e., 120) different paths that start with the edge from 1 to 2 before it finds the path 1 -> 3 -> 4 -> 5 -> 6 -> 7 -> 2 -> 8. And if we add one more vertex, so that #2 is the only one that connects to #9 but all other possible connections are there, it will look at more than 6! = 720 different paths. If there are 20 vertices for which #2 is the only one that connects to #20 but all other possible connections are there, the method will look at more than 17! > trillions of different paths before it finds the right one. In general, the execution time is greater than (NV-3)!, which is greater than 2^{NV} when NV is at least 9. That is, this method has at least an exponential big-oh execution time.

Inductive proof that (NV-3)! > 2^{NV} when NV is at least 9: When NV is 9, (9-3)! is 720 and 2^9 is only 512, so the inequality clearly holds when NV is 9. Once we have proven the inequality true when NV is any k >= 9, then the inequality for k+1 has k-2 times as much on the left (thus at least 7 times as much), and the inequality has only twice as much on the right, so the left-hand side remains larger than the right-hand side.

Exercise 19.29 What is the depth-first traversal sequence for the graph in Figure 19.5 starting from vertex #6 instead of from vertex #1? Assume that `edgesFrom` returns vertices in increasing order of id numbers.
Exercise 19.30 Same question as the previous exercise, except for breadth-first traversal rather than depth-first traversal.
Exercise 19.31 Complete the given description of the traveling salesman process for Figure 19.5. You already have visited vertices 4, 1, 3, 2, 6 in that order. Include all vertices mentioned, even those that led to a failure. Can you find a traversal starting from any other vertex besides vertex 4?
Exercise 19.32 Revise the coding in Listing 19.8 to obtain a method `public boolean CanGo(Vertex one, Vertex two)` to tell whether there is a path from `one` to `two`.
Exercise 19.33 Revise the coding of the `markDistance` method to have an extra int parameter `n` and to return the distance to the vertex with ID `n` as soon as it is found.
Exercise 19.34* For each value of N from 2 to 5, draw a graph with N vertices and no cycles that has the maximum possible number of edges. Through trial and error, find the formula for the number of edges in terms of N.
Exercise 19.35* Write a method to change all vertices' marks back to zero.
Exercise 19.36* Write a method that calls `solveSalesman` in Listing 19.8 and returns a String value with pairs such as (3,5) indicating that vertex 3 is the fifth vertex in the traveling salesman path. It should return the empty String if there is no solution.
Exercise 19.37* Revise `solveSalesman` as described for a stack parameter and write the method that contains the method call `solveSalesman (getVertex(1), 1, stack)`. It should print the vertices in the full traveling salesman path in the order they are to be visited.
Exercise 19.38* Write a method that prints the edges in a spanning tree of a graph if it is a connected graph. Revise `markReachables` in Listing 19.8 slightly and use it to store the edges on a queue.
Exercise 19.39* Rewrite `isConnected` to do the same thing but without using a counter. Hint: Use a boolean local variable instead.

Part B Enrichment And Reinforcement

19.6 Union-Find Structures

The algorithms for weighted graphs in the next section require the use of a **UnionFind** structure. This, in its simplest form, is a data structure that stores information about the non-negative integers up to but not including some fixed number `N`. Specifically, it treats those integers as divided into a number of disjoint groups. This partitioning of the integers is initially done (when the UnionFind object is constructed) with each integer in its own group of that one number. The groups can then be combined.

The only change you can make in a UnionFind object is to tell it to combine two separate groups into one larger group; the `unite` method does this. The only query you can make of a UnionFind object is to ask it whether two integers you specify are in the same group; the `equates` method does this. Example: The following coding divides the int values 0 through 99 into two groups, the odd numbers and the even numbers. Then the expression `group.equates(x, 1)` tells whether x is odd:

```
UnionFind group = new UnionFind (100);
for (int k = 0;  k < 98;  k++)
   group.unite (k, k+2);
```

The coding in Listing 19.9 (see next page) gives one of the simplest implementations of this data structure: We use an array of `N` different int values to name the groups. So initially 0 is in group #0, 1 is in group #1, 2 is in group #2, etc. When we unite the groups for two values `one` and `two`, we simply go through the array of group names and change the group name of each element of `two`'s group to be `one`'s group name. Naturally, we first make sure that the two given ints are in the range `0...N-1`.

The execution time for `equates` is big-oh of 1, but the execution time for `unite` is big-oh of `N` time. This is simply not good enough for our algorithms. We need something that executes faster.

Using circular linked lists of nodes

An interesting idea is to store in each array component a reference to one node in a circular linked list of nodes, where each node contains an int value that is in the group. Circular means that the last node links back to the first. For instance, if 2, 7, and 4 are all in the same group, then we could have a linked list of three nodes with 2 in the first node, which links to a node containing 7 as its data, which links to a node containing 4 as its data, which links back to the node containing 2 as its data.

Initially, each component `itsItem[k]` references a node `p` that contains `k` as its data and links back to itself (`p.itsData` is `k` and `p.itsNext` is `p`). The really interesting part is that we never need to change the values stored in the array; we only need to change the links. For instance, to unite the group containing 2 (and possibly others) with the group containing 5 (and possibly others), execute this sequence of statements:

```
Node saved = itsItem[2].itsNext;
itsItem[2].itsNext = itsItem[5].itsNext;
itsItem[5].itsNext = saved;
```

So execution time for `unite` is now reduced to big-oh of 1. However, answering the true-false question `equates(3,7)` requires starting from the node in `itsItem[3]` (which will of course contain 3) and checking each node on that linked list to see if it contains 7. So `equates` now executes in big-oh of `N` time. That is still not acceptable.

Listing 19.9 The UnionFind class of objects

```
public class UnionFind    // simple array implementation
{
   private final int itsLength;
   private int[] itsItem;


   /** Create a partition of the int values 0...numValues-1,
    *  initially with each int in its own 1-element group.  Use
    *  1 in place of numValues if numValues is not positive. */

   public UnionFind (int numValues)
   {  itsLength = numValues > 0  ?  numValues  :  1;
      itsItem = new int[itsLength];
      for (int k = 0;  k < itsLength;  k++)
         itsItem[k] = k;
   } //=============================================


   /** Tell whether one and two are in the same group. */

   public boolean equates (int one, int two)
   {  return one >= 0 && one < itsLength && two >= 0
             && two < itsLength && itsItem[one] == itsItem[two];
   } //=============================================


   /** Assure that one and two are in the same group.
    *  No effect if one or two is not in 0...numValues-1. */

   public void unite (int one, int two)
   {  if (one >= 0 && one < itsLength && two >= 0
             && two < itsLength && itsItem[one] != itsItem[two])
      {  for (int k = 0;  k < itsLength;  k++)
         {  if (itsItem[k] == itsItem[two])
               itsItem[k] = itsItem[one];
         }
      }
   }  //=========================================
}
```

Using queues of nodes

We try a third implementation: The component for each element stores a non-empty queue object using a linked list of nodes, each node containing one of the elements of its group (rather like NodeQueue in Section 14.4). Now `equates` is back to being big-oh of 1: simply check that `itsItem[one]` and `itsItem[two]` are the same queue. The `unite` method for two groups of size x and y changes one of the two queues to contain a linked list of x+y nodes; this can be done in big-oh of 1 time. Then it changes the component for each element of the other group to be that revised queue value.

As long as we can be sure to always change the components for the smaller group, this will take at most `(N/2)*log(N)` operations by the time we have combined all `N` elements into one group. Since that requires that `unite` is called `N-1` times, average execution time is about `log(N)/2` per `unite` operation. This `unite` method is in Listing 19.10. The rest of this improved UnionFind class is left as an exercise. An alternate implementation in which `unite` is big-oh of 1 and `equates` is big-oh of log(N) is left as a major programming project.

Listing 19.10 The UnionFind class of objects, improved

```
public class UnionFind    // Queue of Nodes implementation
{
   private final int itsLength;
   private Queue[] itsItem;     // Queue is a nested class
   // equates is unchanged.  The constructor is an exercise.


   /** Assure that one and two are in the same group.
    *  No effect if one or two is not in 0...numValues-1. */

   public void unite (int one, int two)
   {  if (one >= 0 && one < itsLength && two >= 0
              && two < itsLength && itsItem[one] != itsItem[two])
      {  if (itsItem[one].itsSize >= itsItem[two].itsSize)
            combineQueues (itsItem[one], itsItem[two]);
         else  // swap the order so the larger is first
            combineQueues (itsItem[two], itsItem[one]);
      }
   }  //=========================================

   private void combineQueues (Queue first, Queue second)
   {  first.itsRear.itsNext = second.itsFront;
      first.itsRear = second.itsRear;
      first.itsSize += second.itsSize;
      for (Node p = second.itsFront; p != null; p = p.itsNext)
         itsItem[p.itsData] = first;
   }  //=========================================


   private static class Queue
   {
      public Node itsFront;
      public Node itsRear;
      public int itsSize;

      public Queue (int given)
      {  itsFront = new Node (given, null);
         itsRear = itsFront;
         itsSize = 1;
      }
   }  //=========================================

   private static class Node { /* left as an exercise */ }
}
```

It is hard to prove the total number of operations is at most `(N/2)*log(N)`, but you can easily prove a limit of `N*log(N)`: Each time the value of a component of `itsItem` changes, the new queue is at least twice the size of the old queue, so it does not change more than `log(N)` times. Example for a power of 2: Let `N` be 16, and make 8 groups of 2 (8 operations), then 4 groups of 4 (8 more operations), then 2 groups of 8 (8 more), then 1 group of 16 (8 more), for a total of 32 operations, which is `(16/2) * log(16)`.

Exercise 19.40 Write the Node class for Listing 19.10.
Exercise 19.41 Write the constructor for Listing 19.10.
Exercise 19.42* Write the complete UnionFind implementation using a circular linked list, as described in this section.

19.7 Weighted Graphs; Algorithms For Minimum Spanning Trees

In some applications, each Edge of a graph has a positive number assigned to it called its **weight**. The graph is then called a **weighted graph** or a **network**. For example, if each Vertex represents a city and each Edge represents a section of train track that could be built between two cities, then the problem of finding the cheapest way to connect all cities depends on the cost of each segment of track. In that case, the cost of the track is the weight of the Edge. The problem of finding a way to connect cities that minimizes travel time depends on the mileage of each segment of track. In that case, the mileage of the track is the weight of the Edge.

It is easy to modify the declarations in the Edge class to allow for Edges having weights. Simply add a constructor that has the weight as a third parameter and provide a way of retrieving the weight of the Edge later. Also amend the existing 2-parameter Edge constructor to assign WEIGHT = 1. We only allow positive weights:

```
// modifications to the Edge class
public final double WEIGHT;

public Edge (Vertex from, Vertex to, double weight)
{  TAIL = from;
   HEAD = to;
   WEIGHT = weight > 0.0 ? weight : 1.0;
} //================================================
```

A method to find the average weight of the edges leaving a given vertex is as follows, assuming that the vertex does in fact have any edges leaving it:

```
public double averageWeight (Vertex given)
{  Iterator it = edgesFrom (given);
   double total = ((Edge) it.next()).WEIGHT;
   int count = 1;
   while (it.hasNext())
   {  total += ((Edge) it.next()).WEIGHT;
      count++;
   }
   return total / count;
} //================================================
```

For the rest of this section, we will assume that our graphs are undirected, which means that for every edge from vertex v to vertex w there is an edge from w to v of the same weight. From an undirected point of view, we consider these to be the same edge. When we work with a subtree of the undirected graph, we have a particular root vertex in mind, and paths in the tree structure go away from that root vertex. But we also talk about a path in the tree from one vertex to another, in which case we allow going in whatever direction is necessary to get from the first vertex to the second.

Minimum spanning trees

A **minimum spanning tree (MST)** for a weighted graph is a spanning tree for which the total of the weights of its edges is no more than the total for any other spanning tree. That is, out of all spanning trees that the graph has, none is "lighter" in total weight.

Reminder A spanning tree is a subgraph which includes all NV vertices but has no cycles. The number of edges in a tree with NV vertices is NV-1 (since each vertex except the root is at the head of exactly one edge in the tree).

You could find a minimum spanning tree for a weighted graph by considering all possible collections of NV-1 edges, checking each collection to see if it forms a tree and, if so, calculating its total weight. Then take one with the smallest total (there may be several). The execution time for this algorithm is exponential in the number of edges.

But a simple **greedy algorithm** can find the answer much faster. What we do is make a sequence of NV-1 choices, adding one edge at each choice to a subgraph that is initially empty but gradually grows to be a MST. A greedy algorithm chooses the edge each time which saves as much weight as possible. Computer scientists named Kruskal and Prim have found two moderately different ways to do this. The underlined parts of the following two descriptions are the only significant differences between the two algorithms.

Kruskal's algorithm: For the first choice, choose the lightest edge of all and add it (plus its two end points) to the subgraph. For each additional choice, choose the lightest edge that connects two vertices for which no path yet exists between them in the subgraph. If at any point along the way this gives you several different edges of the same weight, it makes no difference which one you pick.

Prim's algorithm: For the first choice, choose the lightest edge exiting any chosen vertex and add it (plus its two end points) to the subgraph. For each additional choice, choose the lightest edge that connects two vertices for which you already have exactly one in the subgraph so far. If at any point along the way this gives you several different edges of the same weight, it makes no difference which one you pick.

In Figure 19.7, the same graph is shown twice, with the weight of each edge marked above or to the side of the edge. On the left, Kruskal's algorithm would first choose the edge between 1 and 4, since it has the smallest weight of all. Second, Kruskal's would choose the edge between 6 and 7, since it is the second lightest. On the right, Prim's algorithm starting from vertex 3 would choose the edge between 3 and 2, since it has the smallest weight of all edges out of vertex 3. Second, Prim's would choose the edge between 2 and 6, since it has the smallest weight of all edges out of either 2 or 3.

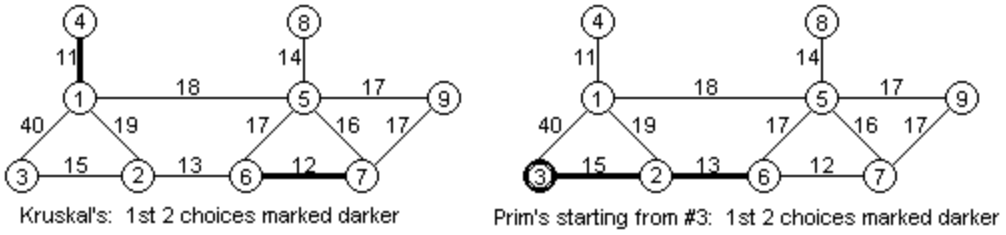


Figure 19.7 The first two edges chosen in the two algorithms

Kruskal's next choice is the edge between 2 and 6, then the edge between 5 and 8, because those are the two lightest still left. Prim's next choice is the edge between 6 and 7, then the edge between 7 and 5, because those are the lightest that connect to vertices already in the subgraph. Neither algorithm will ever choose the edge between 5 and 6.

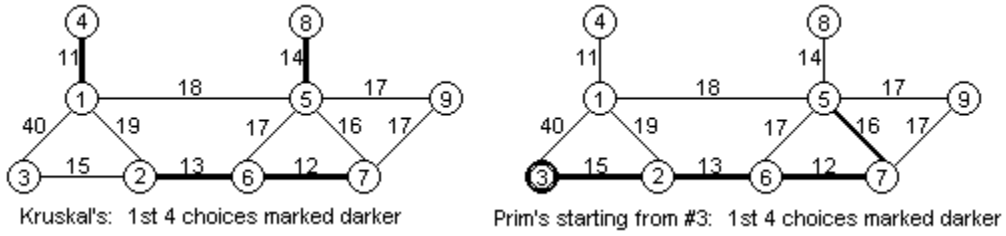


Figure 19.8 The first four edges chosen in the two algorithms

The correctness of the two algorithms

Any time you have an algorithm proposed as a solution to a problem, you have to find out at least three things about it: (a) Does it always give the right answer? (b) How do you code it? (c) How fast does it execute? We begin with the first question.

Both algorithms can be expressed with the following structured design:

1. Let X be an empty set of edges.
2. Do the following NV-1 times...
   2a. Add to X the lightest edge that satisfies the required property.

We define X[k] to be the set X after k iterations of this loop. So X[0] is the empty set of edges, X[1] contains just one edge, X[2] contains two edges, etc. X[NV-1] is the final result.

Proof of the correctness of these two algorithms

Theorem: Prim's algorithm and Kruskal's algorithm each produces an MST.

Proof part 1, that X[NV-1] is an MST: Any MST contains NV-1 edges. X[NV-1] contains NV-1 edges (obviously) and X[NV-1] is contained in some MST (this requires further proof; see part 2). Therefore, X[NV-1] must be the whole MST.

Proof part 2, that X[NV-1] is contained in some MST: X[0] is contained in some MST (obviously, since it is empty), and X[k+1] is contained in some MST whenever X[k] is (this requires further proof; see part 3). So by the induction principle, X[NV-1] is contained in some MST.

Proof part 3, that X[k+1] is contained in some MST if X[k] is contained in some MST:

1. Define <v,w> to be the edge added to X[k] to obtain X[k+1]. Define `ralph` to be the MST known to contain X[k]. If <v,w> is in `ralph`, we are done. Otherwise continue through steps 2-10 below.
2. There is some path from v to w in `ralph` (because `ralph` spans the graph).
3. For Prim's, v is in an edge of X[k] but w is not. So there is a first edge <x,y> along the path of step 2 for which x is in an edge of X[k] but y is not.
4. For Kruskal's, we cannot get from v to w using only edges in X[k]. So there is a first edge <x,y> along the path of step 2 for which you cannot get from x to y using only edges in X[k].
5. The edge <v,w> is at least as light as the edge <x,y> (since otherwise the algorithms would have chosen <x,y> instead of <v,w> to be added to the subgraph).
6. Define M to be the subgraph built by having <v,w> replace <x,y> in `ralph`.
7. M contains X[k+1] (since `ralph` contains X[k], M contains all of the edges of X[k] plus the edge <v,w> that was added to X[k] to get X[k+1]).
8. M is a spanning tree (since you can still get from x to y within M, by following the path of step 2 from x back to v, then taking the edge <v,w>, then following the path of step 2 from w back to y).
9. M has no more total weight than `ralph` (by step 5, since M is `ralph` except that <v,w> replaces <x,y>).
10. So M is a minimal spanning tree containing X[k+1] (from steps 7, 8, and 9).

It may help you understand Kruskal's algorithm better if you contrast it with a different algorithm for finding a minimum spanning tree, to wit, Jones's algorithm:

1. Let X be the set of all the NE edges.
2. Do the following until X has only NV-1 edges...
   2a. Delete the heaviest edge that connects two vertices for which there is another path in X between them.

This algorithm can be proven logically to give the correct answer. However, it has the longest execution time of the three, which is why it is not written up in more books. For one thing, the basic loop of step 2 executes NE-NV times, which is substantially larger than the basic loop for Kruskal's algorithm except for very "sparse" graphs (less than two edges per vertex on average). For another, the time required inside the basic loop to see if there is another path between two vertices is around big-oh of NE time.

Coding and the big-oh

Listing 19.11 (see next page) gives the coding for Kruskal's algorithm. Lines 2-3 put all the edges on a priority queue in increasing order of weight; this along with the taking off the priority queue can be done in NE*log(NE) time using a heap (which was described in Section 18.7, but you do not need to have read it for this section). We make sure that a two-way edge only goes on the list once, when its tail is smaller than its head.

Reminder The three basic PriQue methods for priority queues are as follows:

- `q.add(x)` to add a value x;
- `q.removeMin()` to remove a value and return it; and
- `q.isEmpty()` to see whether any values are left.

The HeapPriQue constructor has a parameter of type Comparator, which is an object that supplies a function for deciding the priority of elements on the queue. This function is the `compare` method in the ByWeight class at the bottom of Listing 19.11.

Line 3 calls a method to construct the priority queue of edges in the MST, which executes in big-oh of NE*log(NE) time. Line 4 calls a method that creates a UnionFind object that partitions the vertices so that each is in a separate group by itself. Then it uses a while-loop that executes NE times and contains a call of `unite` whose average execution time is big-oh of log(NV)/2. So the overall execution time of `constructMST` is big-oh of NE*log(NV)/2. Since the big-oh of the `Kruskals` method is the largest of these values, it has an execution time that is big-oh of NE*log(NE).

By contrast, Prim's algorithm maintains a priority queue only of the edges that run from a vertex in the subgraph so far to a vertex not in the subgraph so far, in increasing order of weight. Instead of keeping the vertices in many groups according to which vertices are connected to each other, it only has two groups: those vertices that are on an edge in the subgraph so far, and those that are not. If the priority queue of edges is implemented using a heap, then the average time to put an edge on the queue and take it off again is big-oh of log(NE) time. So Prim's algorithm also has an overall execution time of big-oh of NE*log(NE).

Both of these algorithms actually execute in big-oh of NE*log(NV) time, because that is the same as big-oh of NE*log(NE) time. The reason is that the number of edges must be less than the square of the number of vertices, so $\log(NE) < \log(NV^2) = 2*\log(NV)$.

Exercise 19.43 Complete the discussion of the process for Prim's algorithm, starting from the situation on the right side of Figure 19.8.
Exercise 19.44 Tell which edges are removed in what order, and why, in the application of Jones's algorithm to the graph in Figure 19.8.
Exercise 19.45* Complete the discussion of the process for Kruskal's algorithm, starting from the situation on the left side of Figure 19.8.
Exercise 19.46** Essay: Explain why, for a graph where all edges have different weights, every MST contains the lightest edge exiting every vertex. Note that this is a stronger assertion than what Kruskal's algorithm indicates, namely, that the lightest edge exiting any particular vertex is in some MST.

Listing 19.11 Kruskal's algorithm in the Graph class for finding an MST

```
/** Return a stack containing the edges on an MST for an
 *  undirected graph.  If the graph is not connected, the
 *  edges should form one MST for each connected part. */

public StackADT Kruskals()
{  StackADT spanner = new NodeStack();                    //1
   PriQue queue = new HeapPriQue (new ByWeight());        //2
   putEdgesOnPriorityQueueByWeight (queue);               //3
   constructMST (spanner, queue, getNumVertices());       //4
   return spanner;                                        //5
} //============================================


/** Put all Edges of the Graph on the priority queue with
 *  lighter Edges having higher priority.
 *  Precondition:  the queue is not null.  */

private void putEdgesOnPriorityQueueByWeight (PriQue queue)
{  Iterator it = vertices();                              //6
   while (it.hasNext())                                   //7
   {  Iterator edges = edgesFrom ((Vertex) it.next());    //8
      while (edges.hasNext())                             //9
      {  Edge e = (Edge) edges.next();                    //10
         if (e.TAIL.ID < e.HEAD.ID)                       //11
            queue.add (e);                                //12
      }                                                   //13
   }                                                      //14
}   //============================================


/** Check each Edge in order from lightest to heaviest.  Put
 *  the Edge in spanner whenever its two endpoints do not
 *  have a path between them using Edges already in spanner.
 *  Precondition:  spanner and queue are not null.  */

private static void constructMST (StackADT spanner,
                PriQue queue, int numLeft)
{  UnionFind group = new UnionFind (numLeft + 1);         //15
   while (! queue.isEmpty() && numLeft > 1)               //16
   {  Edge e = (Edge) queue.removeMin();                  //17
      if (! group.equates (e.TAIL.ID, e.HEAD.ID))         //18
      {  spanner.push (e);                                //19
         group.unite (e.TAIL.ID, e.HEAD.ID);              //20
         numLeft--;                                       //21
      }                                                   //22
   }                                                      //23
} //============================================


private static class ByWeight implements java.util.Comparator
{
   public in compare (Object one, Object two)
   {  double diff = ((Edge) one).WEIGHT - ((Edge) two).WEIGHT;
      return diff > 0 ? 1 : diff < 0 ? -1 : 0;            //25
   }
} //============================================
```

19.8 The Shortest Path Problem: Dijkstra's Algorithm

In a weighted graph, it is often very useful to know the cheapest path from a given vertex to every other vertex. For instance, you may live in a town situated at the source vertex and want to know how little you can get by with to travel to any other town. You would have to know what each individual part of the trip costs, i.e., the weight of the edge from each vertex to each other vertex. Or if a company were to send messages over a network to various other points in the network, it would want to know the shortest distance from its position to each other vertex in the weighted graph.

Dijkstra's algorithm

Dijkstra's algorithm is a greedy algorithm. It fills in and returns an array of double values, one per vertex, such that `dist[k]` is the shortest distance from the given starting vertex `start` to the vertex with id `k`. It leaves `dist[k]` zero if you cannot reach vertex `k` from `start`.

You start by putting all the vertices you can reach in one step from `start` on a list of vertices conventionally called the "fringe". You also set `dist[k]` equal to the length of the edge from `start` to the vertex with ID `k`, for each of those vertices on the fringe list, since you know you can reach them in that distance (or possibly less). It turns out later to be convenient to set `dist[start.ID]` to -1.

Next you find the vertex `best` on the fringe list with the shortest `dist[k]` value. That will be the shortest distance to that vertex. Remove `best` from the list. Then for each edge `e` whose tail is `best`, put `e.HEAD` on the fringe list, setting `dist[e.HEAD.ID]` to `newDist`, which is the distance from `start` to `best` plus the weight of the edge from `best` to the vertex added.

Exception: If `e.HEAD` already has a non-zero value for its distance, that means you have previously found a path to it from `start` and it is already on the fringe list. So do not add it (again) to the fringe list, and replace its `dist[e.HEAD.ID]` by `newDist` only if that is less than `dist[e.HEAD.ID]`. In other words, you make a change only when you find a shorter path than any path you had previously found to `e.HEAD`.

Repeat the process of finding the best (closest) vertex on the fringe list, removing it, and putting on the fringe list all vertices you can reach from best that you could not reach before, as well as updating the `dist` array. When the fringe list is empty, `dist` tells the shortest distance from `start` to each other vertex that is reachable from `start`. Example: For Figure 19.8 with `start` being vertex 3, the first distance found is the 15 from 3 to 2, then the 28 from 3 to 6 by way of 2, then the 34 from 3 to 1 by way of 2.

The coding for this algorithm is the `Dijkstras` method in the upper part of Listing 19.12. It uses an ArrayList for the fringe list, since it is reasonably efficient to do so. The `updateFringe` method (line 3) iterates through all the edges `e` exiting `start` and adds `e.HEAD` to the fringe list. It also assigns to each vertex the shortest distance known so far from `start` to that vertex. Then the method loops as long as the fringe list is not empty, finding the vertex `best` on the fringe list with the shortest distance (line 6) and readjusting the fringe list and `dist` array accordingly.

Execution time for Dijkstra's algorithm is as follows: The while-statement in line 5 executes NV times, each time calling `findClosest`, which iterates NV times, and then calling `updateFringe`, which iterates less than NV times. So the overall execution time is big-oh of NV^2. However, use of a priority queue can improve this (see the exercises).

Listing 19.12 Dijkstra's algorithm in Graph class for finding shortest paths from a vertex

```
/** Return an array of doubles in which the kth component is
 *  the minimum distance from the given vertex to vertex #k.
 *  A component value of 0.0 means it is not reachable. */

public double[] Dijkstras (Vertex start)
{  ArrayList fringe = new ArrayList();                    //1
   double[] dist = new double [getNumVertices() + 1];     //2
   updateFringe (start, fringe, dist);                    //3
   dist[start.ID] = -1.0;                                 //4
   while (! fringe.isEmpty())                             //5
   {  Vertex best = findClosest (fringe, dist);           //6
      updateFringe (best, fringe, dist);                  //7
   }                                                      //8
   return dist;                                           //9
}  //=============================================


/** Add to the fringe each vertex reachable from vert but
 *  not previously processed. Update the shortest distance
 *  array for any vertex that is reachable from vert. */

private void updateFringe (Vertex vert, ArrayList fringe,
                           double[] dist)
{  Iterator it = edgesFrom (vert);                        //10
   while (it.hasNext())                                   //11
   {  Edge e = (Edge) it.next();                          //12
      double newDist = dist[vert.ID] + e.WEIGHT;          //13
      if (dist[e.HEAD.ID] == 0.00) //not on fringe list   //14
      {  dist[e.HEAD.ID] = newDist;                       //15
         fringe.add (e.HEAD);     // add to list          //16
      }                                                   //17
      else if (newDist < dist[e.HEAD.ID])                 //18
         dist[e.HEAD.ID] = newDist;                       //19
   }                                                      //20
} //============================================


/** Find and return the vertex in the fringe that has the
 *  smallest dist value, after removing it from the fringe. */

private Vertex findClosest (ArrayList fringe, double[] dist)
{  Vertex best = (Vertex) fringe.get (0);                 //21
   int indexBest = 0;                                     //22
   for (int k = 1;  k < fringe.size();  k++)              //23
   {  Vertex v = (Vertex) fringe.get (k);                 //24
      if (dist[v.ID] < dist[best.ID])                     //25
      {  best = v;                                        //26
         indexBest = k;                                   //27
      }                                                   //28
   }                                                      //29
   fringe.remove (indexBest);                             //30
   return best;                                           //31
} //============================================
```

Correctness of Dijkstra's algorithm

How do we know that Dijkstra's algorithm always gives the shortest path from `start` to each other vertex? At any point when the loop tests `! fringe.isEmpty()` in line 5, let us say the "cloud" is `start` plus all of the vertices found to be best up to that point (and thus removed from the fringe). For purposes of this discussion, treat `start` as being at a distance of -1 from itself. We claim that each vertex in the cloud does in fact have recorded in the `dist` array the shortest distance possible from `start`.

This is trivially true the first time line 5 is evaluated (when `start` is the only vertex in the cloud). Suppose it is true the Nth time line 5 is evaluated (N is 1 or more). Then line 6 finds a `best` value. Let w be the last cloud vertex on the shortest path from `start` to `best`. Then w must come immediately before `best` on that path (otherwise the vertex after w must be in the fringe and have a smaller `dist` value than `best`'s, so it would have been chosen instead of `best`). Since w was in the cloud the Nth time line 5 was evaluated, `dist[w]` is the shortest distance possible from `start` to w, so `dist[best]` is the length of that shortest path from `start` through w. Thus we have an inductive proof of the correctness of Dijkstra's algorithm.

Directed graphs

Dijkstra's algorithm finds the shortest paths from a given vertex in a directed graph as well as in an undirected graph, in as little as big-oh of NE*log(NV). In a **directed acyclic graph** (DAG for short; acyclic means there are no cycles), topological sorting can find either the shortest path or the longest path in big-oh of NE time, even with some negative weights on the edges.

The **transitive closure** of a graph is the graph in which there is an edge from vertex v to vertex w when (and only when) there is a path from v to w in the original graph. You can add edges to an undirected graph that turn it into its transitive closure in big-oh of NE*log(NV) time if you make good use of the stack of edges produced by Kruskal's algorithm. This is left as an exercise. But computing the transitive closure of a directed graph is trickier; this is left as a major programming project.

Exercise 19.47 For Figure 19.8, after Dijkstra's algorithm finds the shortest paths from vertex 3 to vertices 2, 6, and 1, what are the next two shortest paths it finds?
Exercise 19.48 Explain why it helps to set `dist[start.ID]` to -1 instead of leaving it 0. Why is the assignment made in line 4 instead of immediately, after line 2?
Exercise 19.49 Add a QueueADT parameter to the `Dijkstras` method and put on it all the vertices you can reach from `start` in increasing order of the shortest distances from `start`, with the closest one at the front of the queue.
Exercise 19.50 Improve execution of the `findClosest` method by replacing the value at index `closest` by the last value in the ArrayList and then eliminating the last value.
Exercise 19.51 (harder) Essay: Explain why using a priority queue of vertices is problematic in Dijkstra's algorithm, even though finding the best would execute faster.
Exercise 19.52* Essay: Explain how best to modify Listing 19.12 to use a priority queue of <Vertex,double> objects. Use marks to correct for multiple appearances of a Vertex.
Exercise 19.53* Add an array parameter named `lastEdge` to `Dijkstras` method and assign `lastEdge[k]` to be the last edge in the shortest path from `start` to the vertex with ID `k`. This is used in the following exercise.
Exercise 19.54* Write a method that accepts two vertices as parameters, calls `Dijkstras` method with the `lastEdge` array parameter described in the preceding exercise, and then prints out the shortest path from the first vertex to the second vertex.
Exercise 19.55* Essay: Explain clearly how you would use the stack produced by Kruskal's algorithm to compute the transitive closure of an undirected graph.

*19.9 Dynamic Programming (*Enrichment)*

The Fibonacci sequence of numbers is 1, 1, 2, 3, 5, 8, 13, 21,..., where each number is the sum of the two numbers before it: `fib(n) = fib(n-2) + fib(n-1)` except the first two numbers `fib(0)` and `fib(1)` are 1. This sequence appears in several different situations.

For example, if a pair of rabbits requires 1 month to mature and then gives birth to a pair of rabbits each month thereafter, and you start with one new-born pair of rabbits, you have 1 pair at the start (`fib(0)`) and at the end of 1 month (`fib(1)`). A birth gives you 2 pairs at the end of 2 months (`fib(2)`) and another birth gives you 3 pairs at the end of 3 months (`fib(3)`). Then the pair born at the end of month 2 starts producing, which gives you 5 pairs at the end of 4 months (`fib(4)`), 3 of which are mature. So at the end of 5 months, you have your 5 pairs plus 3 more pairs produced by the mature ones, which gives you 8 pairs altogether, of which 5 pairs are now mature, etc.

An obvious but horribly inefficient method of calculating Fibonacci numbers is the following:

```
public static int fib (int n)
{  return n <= 1  ?  1  :  fib (n - 2) + fib (n - 1);
} //===============================================
```

The problem with this coding is that you calculate the answer to the same problem many times over. For instance, `fib(10)` calculates `fib(8)` twice, once directly and once as part of the calculation of `fib(9)`. This leads to the calculation of `fib(7)` 3 times, the calculation of `fib(6)` 5 times, the calculation of `fib(5)` 8 times, etc. Those numbers of times are themselves Fibonacci numbers.

The basic kind of **dynamic programming** stores the solutions to subproblems in an array of values so that, the next time it is to calculate the same result, it looks it up in the array instead of recalculating. For the Fibonacci sequence, you could have an array named `answer` for which (1) `answer[k]` initially stores 0; (2) when `fib(k)` is first calculated, `answer[k]` stores that value in place of 0; (3) any later time `fib(k)` is needed, the value is taken from `answer[k]`. This leads to the coding in Listing 19.13. We make the restriction that the method is not called for any value greater than 10000.

Listing 19.13 The Fibonacci function using dynamic programming

```
public class Fibonacci
{
   public static final int MAX = 10000;
   private static int[] answer = new int[MAX];


   /** Return the nth Fibonacci number, where fib(0) = 1 and
    *  fib(1) = 1 and otherwise fib(n) = fib(n-2) + fib(n-1).
    *  But return 1 if n >= MAX. */

   public static int fib (int n)
   {  if (n <= 1 || n >= MAX)
         return 1;
      if (answer[n] == 0)  // the first request for this value
         answer[n] = fib (n - 2) + fib (n - 1);
      return answer[n];
   }  //===============================================
}
```

Look what happens when this function is first called for n == 5: It calculates `fib(n-2)` which is `fib(3)`. That fills in `answer[3]` with 3 and `answer[2]` with 2. Then it calculates `fib(n-1)`, which is `fib(4)`. That calls `fib(3)` and `fib(2)`, which simply return 3 and 2, storing the 5 in `answer[4]`. Then 3+5 is stored in `answer[5]` and returned for the original call. Any future call of the method will only go as deep as `fib(5)` and `fib(4)` before terminating recursion and returning the answer previously calculated. So execution time is reduced from exponential big-oh time to big-oh of n.

Counting comparisons for the merge sort

The merge sort logic divides a set of values to be sorted into two equal halves, except one half is 1 larger than the other half if you are sorting an odd number of values. Then it sorts each half using the merge sort logic. Finally, it merges the two halves together into one long sorted list, in a process that requires at most `n-1` comparisons if the original list has n values. The question is, what is `compsMS(n)`, the number of comparisons required to sort n values in the worst case?

An obvious but very inefficient method of calculating `compsMS` is the following:

```
   public static int compsMS (int n)
   {  return n <= 1  ?  0
             :  compsMS (n / 2) + compsMS (n - n / 2) + n - 1;
   } //=============================================
```

The problem with this coding is that you calculate the answer to the same problem many times over. For instance, `compsMS(16)` calculates `compsMS(8)` twice, and each of those calculates `compsMS(4)` twice for a total of 4 times.

A dynamic programming solution reduces the execution time from big-oh of n to big-oh of log(n). The straightforward solution is in Listing 19.14. Applying it several times gives the following table of values:

```
n =            2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17
compsMS(n) =   1  3  5  8 11 14 17 21  25 29 33 37 41 45 49 54
```

Listing 19.14 The compsMS function using dynamic programming

```
public class MergeCount
{
   public static final int MAX = 10000;
   private static int[] answer = new int[MAX];


   /** Return the nth compsMS number, which is the maximum
    *  number of comparisons required to sort n values using
    *  the merge sort logic.  But return 0 if n >= MAX. */

   public static int compsMS (int n)
   {  if (n <= 1 || n >= MAX)
         return 0;
      if (answer[n] == 0)  // the first request for this value
         answer[n] = compsMS (n / 2) + compsMS (n - n / 2) + n-1;
      return answer[n];
   }  //=============================================
}
```

You might have noticed a pattern in this table: Add 2 each time up to n==4, then add 3 each time up to n==8, then add 4 each time up to n==16, then start adding 5 each time. For instance, `compsMS(32)` is `2 * compsMS(16) + 31` which is 129, which is 80 more than `compsMS(16)`, which means you add 5 every time as you go from 16 up to 32.

The number of binary trees

How would you count the number of different shapes of binary search trees with 6 nodes? You could categorize them according to which data value the root node holds. The root could contain the first data value, thus it would have a 5-node tree as its right subtree. Or it could contain the second data value, thus it would have a 4-node tree on its right and a 1-node tree on its left. If `trees(n)` is the number of different binary search trees with n nodes, then there are `trees(5)` cases with the root node the first data value and `trees(4)` cases with the root node the second data value.

If the root contains the third data value, it has a 3-node tree on its right and a 2-node tree on its left. Since there are 2 different 2-node trees and 5 different 3-node trees (draw the pictures and see), you multiply to see there are `2 * 5` possible combinations. That is, there are `trees(2) * trees(3)` cases with the root node the third data value.

Since the other alternatives are symmetric to those three, you have to double the total, so you have the following formula:

```
trees(6) = 2*trees(5) + 2*trees(4) + 2*trees(3)*trees(2)
```

This process can be generalized to calculate `trees(n)` for any positive value n. You must make a small adjustment for odd values of n, however; the number of trees with the root in the exact center is not doubled in its calculation.

There is only 1 tree of size 1 and there are only 2 trees of size 2, so a not-so-obvious yet horribly inefficient method of calculating `trees` is as follows:

```
public static int trees (int n)
{  if (n <= 2)
      return n;
   int total = trees (n - 1);  // root at far left
   for (int k = 2;  k <= n / 2;  k++)
      total += trees (k - 1) * trees (n - k);
   return (n % 2 == 0)  ?  2 * total
          :  2 * total + trees (n / 2) * trees (n / 2);
} //================================================
```

A dynamic programming solution reduces the execution time greatly, from exponential big-oh time to only $n^2/2$ additions and multiplications performed. The coding is in Listing 19.15 (see next page). There are 14 of the 4-node trees, 42 of the 5-node trees, and 132 of the 6-node trees.

The basic approach in all three of these situations is the same: After you write an algorithm recursively, check whether the same subproblems are being solved many times over. If so, store the result of the first calculation in an array, generally indexed by the parameter(s), and retrieve that result when the problem is to be re-solved.

Listing 19.15 The trees function using dynamic programming

```
public class TreeCount
{
   public static final int MAX = 10000;
   private static int[] answer = new int[MAX];

   /** Return the nth trees number, which is the number of
    *  different binary trees.  But return n if n >= MAX. */

   public static int trees (int n)
   {  if (n <= 2 || n >= MAX)
         return n;
      if (answer[n] == 0)  // the first request for this value
      {  int total = trees (n - 1);
         for (int k = 2;  k <= n / 2;  k++)
            total += trees (k - 1) * trees (n - k);
         answer[n] = (n % 2 == 0)  ?  2 * total
                   : 2 * total + trees (n / 2) * trees (n / 2);
      }
      return answer[n];
   }  //=============================================
}
```

The longest common subsequence problem

To test whether two individuals have a genetic relationship, scientists look at their DNA strands and see if they have the same long subsequence. Dynamic programming can help find it. A **subsequence** `sub` of a string s is a sequence of characters that appear in s in the same order they appear in `sub`, though not necessarily right next to each other in s. For instance, "CAT" has seven subsequences of positive length: C, A, T, CAT, CA, AT, and CT. Notice that the last one is not a substring of "CAT".

The general problem is, given two strings s and t, to find the longest subsequence of characters that s and t have in common. We could list all possible subsequence of s and check each one against t, but there are $2^{s.length()}$ subsequences of s, which makes this an exponential solution.

Let LCS stand for the phrase "longest common subsequence". Let `longest(s,t)` denote the length of the longest subsequence of the strings s and t. For convenience, let `s'` denote the result of removing the last character of s. We begin by noticing that, if the last characters of s and t are the same, then their longest substring includes that last character, so the following is true:

```
longest(s, t) = longest(s', t') + 1
```

On the other hand, if the last characters of s and t are different, then their LCS must be either the LCS of s and t' or else the LCS of s' and t, whichever is longer:

```
longest(s, t) = Math.max (longest(s', t), longest(s, t'))
```

The following method is a straightforward but quite inefficient method of calculating the longest value, assuming it can directly access the Strings s and t and is passed as parameters the number of initial characters of each String that it is to find the LCS of:

```
public static int longest (int len1, int len2)
{  if (len1 == 0 || len2 == 0)
      return 0;
   else if (s.charAt (len1 - 1) == t.charAt (len2 - 1))
      return longest (len1 - 1, len2 - 1) + 1;
   else
      return Math.max (longest (len1 - 1, len2),
                        longest (len1, len2 - 1));
} //============================================
```

A dynamic programming solution reduces the execution time greatly. The coding is in Listing 19.16 (see next page). There are some obvious differences from the preceding three listings: The public method is originally passed the two strings rather than two int values. So it checks the two strings to make sure they are non-null and not too long to handle. Then it sets up a two-dimensional matrix `answer[i][j]` that tells the length of the LCS that can be formed with the first i characters of the first string and the first j characters of the second string. This matrix cannot be filled with zeros as placeholders, since zero is a legitimate answer to the length of a common subsequence, so the method fills the array with -1. For speed, it also fills in the length 0 of the LCS for the cases where one of the substrings has no characters. Then it calls a private method to calculate the answers.

After the answer is calculated, the client of this class will find it useful to see what values were calculated for various initial substrings of the given strings. A public method named `lcs` is provided for that purpose. The client can easily find out the exact subsequence of characters that is longest by using the public `lcs` method. This is left as an exercise. Execution time is big-oh of `s.length() * t.length()` for this dynamic programming method to fill in the two-dimensional array. The execution time for finding the actual longest common subsequence of characters is `s.length() + t.length()`.

Once you have this straightforward solution, further analysis shows that you can increase the efficiency somewhat by replacing the body of the private `longest(int,int)` method down to the return statement by the following non-recursive equivalent:

```
for (int row = 1;  row <= len1;  row++)
   for (int col = 1;  col <= len2;  col++)
   {  if (one.charAt (row - 1) == two.charAt (col - 1))
         answer[row][col] = answer[row - 1][col - 1] + 1;
      else if (answer[row - 1][col] > answer[row][col - 1])
         answer[row][col] = answer[row - 1][col];
      else
         answer[row][col] = answer[row][col - 1];
   }
```

Listing 19.16 The LCS function using dynamic programming

```
public class LongestCommonSubsequence
{
   public static final int MAX = 1000;
   private static int[][] answer = new int[MAX][MAX];
   private static String one;
   private static String two;

   /** Return the length of the longest common subsequence of
    *  the two given Strings. Return 0 if either is null or
    *  has MAX or more characters.  */

   public static int longest (String first, String second)
   {  if (first == null || second == null
            || first.length() >= MAX || second.length() >= MAX)
         return 0;
      for (int row = 0;  row <= first.length();  row++)
      {  for (int col = 0;  col <= second.length();  col++)
            answer[row][col] = row * col == 0  ?  0  :  -1;
      }
      one = first;
      two = second;
      return longest (one.length(), two.length());
   }  //=============================================

   private static int longest (int len1, int len2)
   {  if (answer[len1][len2] >= 0)
         return answer[len1][len2];
      if (one.charAt (len1 - 1) == two.charAt (len2 - 1))
         answer[len1][len2] = longest (len1 - 1, len2 - 1) + 1;
      else
         answer[len1][len2] = Math.max (longest (len1 - 1, len2),
                                        longest (len1, len2 - 1));
      return answer[len1][len2];
   }  //=============================================

   /** Return the length of the LCS of the first len1 characters
    *  of the first String and the first len2 characters of the
    *  second String.  Throw an Exception for bad indexes. */

   public static int lcs (int len1, int len2)
   {  return answer[len1][len2];
   }  //=============================================
}
```

Exercise 19.56 List the next six Fibonacci numbers after 13 and 21.
Exercise 19.57 The double for-loop in the public `longest` method of Listing 19.16 will execute faster if you replace it by coding that fills in zeros separately from the -1s. Do so.
Exercise 19.58* Write a method that prints the longest common subsequence of s and t after calling `LongestCommonSubsequence.longest(s, t)`.

19.10 Review Of Chapter Nineteen

- A graph consists of a number of **Vertex** objects and **Edge** objects. Each Edge object runs from one Vertex object, its **tail**, to another Vertex object, its **head**. The graph is **undirected** if every edge from vertex v to vertex w is matched by an edge from vertex w to vertex v. In that case we generally treat both edges as one object.
- The **out-degree** of a vertex is the number of edges exiting the vertex, and the **in-degree** of a vertex is the number of edges entering the vertex.
- A **path** in a graph is a sequence of edges where the head of each edge is the tail of the next. A graph is **connected** if you can get from any one vertex to any other vertex by following edges in the graph, i.e., if there is a path from any one vertex to any other vertex.
- The two standard ways to implement a graph are the **adjacency list** (keeping a list of the edges exiting each vertex) and the **adjacency matrix** (keeping a matrix whose <v,w> component is the edge from vertex v to vertex w). The former is faster for depth-first traversals, but the latter is faster for seeing whether an edge exists.
- A **topological sort** of a graph is an ordering of the vertices so that, for each edge from some vertex v to some other vertex w, v comes before w in the sequence.
- A **subgraph** of a graph G is a subset of the edges of G plus the heads and tails of those edges. A **spanning tree** of a graph is a subgraph containing all the vertices of the graph and having exactly one path from the root vertex to any other vertex. A **minimum spanning tree** for a **weighted graph** (where edges have a positive weight value) is a spanning tree for which the total weight of the edges is the least possible.
- **Kruskal's algorithm** and **Prim's algorithm** find a minimum spanning tree for a connected graph. **Dijkstra's algorithm** finds the shortest paths from a given vertex to all other vertices.

Answers to Selected Exercises

19.1
```
public void makeTheGraphConnected()   // in HamGraph
{     while ( ! isConnected())
            addEdgeChosenRandomly();
}
```
19.2
```
public int inDegree (Vertex given)   // could be in any subclass of Graph
{     int count = 0;
      for (int k = 1;  k <= getNumVertices();  k++)
      {     if (contains (new Edge (getVertex (k), given)))
                  count++;
      }
      return count;
}
```
19.9
```
public void removeAllEdgesFrom (Vertex v)  // for MatrixGraph
{     for (int k = 1;  k <= getNumVertices();  k++)
            edgeAt[v.ID][k] = null;
}
```
19.10
```
public int outDegree (Vertex given) // for MatrixGraph
{     int count = 0;
      for (int k = 1;  k <= getNumVertices();  k++)
      {     if (edgeAt[given.ID][k] != null)
                  count++;
      }
      return count;
}
```
19.11 Same as for the preceding exercise except reverse the order of indexes: edgeAt[k][given.ID].

19.12
```
public Iterator edgesFrom (Vertex v)  // for MatrixGraph
{     ArrayList list = new ArrayList();
      for (int k = 1;  k <= getNumVertices();  k++)
      {     if (edgeAt[v.ID][k] != null)
                  list.add (edgeAt[v.ID][k]);
      }
      return list.iterator();
}
```

19.13 Declare a new instance variable: private int[] itsIns;
In the constructor, add: itsIns = new int [maxVerts + 1]; // initially all components are zero.
In remove(Edge), add: if (e != null) itsIns[given.HEAD.ID]--;
In add(Edge), add: if (edgeAt[given.TAIL.ID][given.HEAD.ID] == null) itsIns[given.HEAD.ID]++;

19.20 Copy the first two lines of the add method (the if statement) into both contains and remove.

19.21
```
public int outDegree (Vertex given)  // for ListGraph
{     return ((Collection) itsList.get (given.ID)).size();
}
```

19.22
```
public void removeSomeEdges (Vertex given)  // for ListGraph
{     ArrayList z = (ArrayList) itsList.get (given.ID);
      for (int k = z.size() - 1;  k >= 0;  k--)  // question: why doesn't it work in the other direction?
      {     if (((Edge) z.get (k)).HEAD.ID > given.ID)
                  z.remove (k);
      }
}
```

19.29 From 6 you visit 2, from 2 you go to 1, from 1 you go to 3. Backing up, from 1 you go to 4. Backing up, from 6 you go to 5, from 5 you go to 7, from 7 you go to 8. Finally, from 5 you go to 9.

19.30 From 6 you visit 2, 5, 7 in that order. Then from 2 you visit 1 and 3. From 5 you visit 8 and 9. From 8 and 9 there are no more visits. From 1 you visit 4, and you are done.

19.31 Now that you have visited 4, 1, 3, 2, 6 in that order, the next vertex to visit is 5 (since we assume that the edgesFrom iterator produces vertices in numeric order). From 5 you go to 7 and then 9 but you get stuck. You try again: from 5 you go to 8 but get stuck. Again: from 5 you go to 9 and then 7 but you get stuck. So you back up to 6 and go to 7, marking it 6. From 7 you go to 5 and then 8 but you get stuck. You try again: from 7 you go to 5 and then 9 but you get stuck. Again: from 7 you go to 9 (marking it 7), and then to 5 (marking it 8), and finally 8 (marking it 9). That finishes the traversal. No other traveling salesman traversal is possible with that graph.

19.32 Use the body of isConnected with these changes: Replace line 8 by: markReachables (one);
Insert after line 8: boolean valueToReturn = two.getMark() > 0;
Replace the last line of the method by: return valueToReturn;

19.33 Change the heading to: private int markDistance (Vertex given, QueueADT queue, int n).
Just before the if statement, insert: if (v.ID == n) return current.getMark() + 1;
At the end of the method, put return -1; (to signal failure).

19.40
```
private static class Node
{     public int itsData;
      public Node itsNext;
      public Node (int data, Node next)
      {     itsData = data;
            itsNext = next;
      }
}
```

19.41
```
public UnionFind (int numValues)
{     itsLength = numValues > 0  ?  numValues  :  1;
      itsItem = new Queue[itsLength];
      for (int k = 0;  k < itsLength;  k++)
            itsItem[k] = new Queue (k);
}
```

19.43 Prim's fifth choice is the edge of weight 14 between 5 and 8. Prim's sixth choice has to be an edge of weight 17. It cannot be the one between 5 and 6, because 5 is already reachable from 6. You may choose either of the two edges of weight 17 leaving vertex 9. Prim's seventh choice is the edge of weight 18 between 5 and 1, and finally the edge of weight 11 between 1 and 4.

19.44 Removing edges starting from the heaviest, you take the edge (1,3) of weight 40, then the edge (1,2) of weight 19, but skip the edge (1,5) of weight 18 (since that disconnects the graph). Then take one of the edges of weight 17 from 9 as well as the edge (5,6) of weight 17. The 8 edges left are an MST.

19.47 The path of length 40 from 3 to 7, then the path of length 45 from 3 to either 4 or 5.

19.48 Otherwise the test in line 14 of Listing 19.12 would be dist[e.HEAD.ID] == 0.0 && e.HEAD != start, which would execute slower and be harder to understand. If the assignment were before line 3, the updateFringe method would calculate the newDist as 1 less than it should be.

19.49 The method heading should be: public double[] Dijkstras (Vertex start, QueueADT queue)
Insert this statement after line 6: queue.enqueue (best);

19.50 Replace line 30 of Listing 19.12 by the following two statements:
fringe.set (closest, fringe.get (fringe.size() - 1)); fringe.remove (fringe.size() - 1);

19.51 The problem is in line 19. You have a vertex e.HEAD that has previously been added to the fringe with a certain dist value, which is what the Comparator of the priority queue would use. Line 19 reduces the dist value, so that vertex would have to be shifted in most implementations of PriQue, such as HeapPriQue or TreePriQue. Basic priority queues do not allow for such changes in priority.

19.56 13,21,34,55,89,144,233,377.

19.57
```
for (int row = 0;  row <= first.length();  row++)
      answer[row][0] = 0;
for (int col = 1;  col <= second.length();  col++)
      answer[0][col] = 0;
```
Then have the same double loop but start each index from 1 and simply assign -1 to each.