CHAPTER 14

Graphs

LEARNING OBJECTIVES

When you complete Chapter 14, you will be able to...

- follow and explain graph-based algorithms using the usual computer science terminology.
- design and implement classes for labeled or unlabeled graphs.
- list the order in which nodes are visited for the two common graph traversals (breadth-first and depth-first) and implement these algorithms.
- simulate the steps of simple path algorithms (such as determining whether a path exists) and be able to design and implement such algorithms.
- simulate the steps of Dijkstra's shortest-path algorithm and be able to implement it.

CHAPTER CONTENTS

14.1 Graph Definitions
14.2 Graph Implementations
14.3 Graph Traversals
14.4 Path Algorithms
Chapter Summary
Solutions to Self-Test Exercises
Programming Projects

So many gods, so many creeds,
So many paths that wind and wind,
When just the art of being kind
Is all this sad world needs.

ELLA WHEELER WILCOX
"The World's Need"

Graphs are the most general data structure in this text. In fact, it's fair to say that graphs are the ultimate commonly used data structure. Many of the data structures you will study in the future can be expressed in terms of graphs. This chapter provides an introduction to graphs and their algorithms, including the implementation of a graph class in Java.

14.1 GRAPH DEFINITIONS

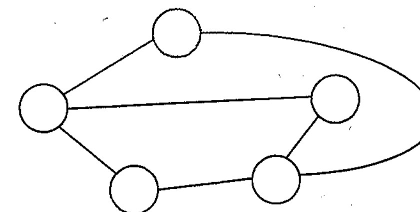
A graph, like a tree, is a nonlinear data structure consisting of nodes and links between the nodes. In the trees we have already seen, the nodes are somewhat orderly: The root is linked to its children, which are linked to their children, and so on to the leaves. But in a graph, even this modicum of order is gone. Graph nodes can be linked in any pattern—or lack of pattern—depending only on the needs of an application.

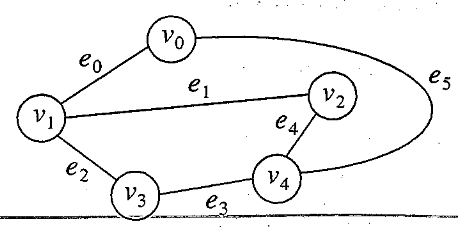
Graphs occur in several varieties. We'll start with the simplest form: *undirected graphs.*

graphs occur in several varieties, the simplest of which is an undirected graph

Undirected Graphs

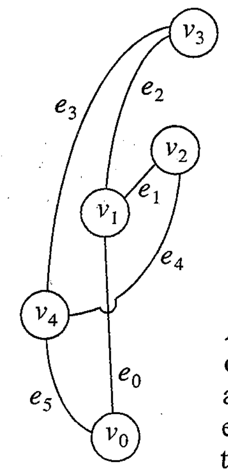
An undirected graph is a set of nodes and a set of links between the nodes. Each node is called a **vertex**, each link is called an **edge**, and each edge connects two vertices. Undirected graphs are drawn by putting a circle for each vertex and a line for each edge. For example, here is a drawing of an undirected graph with five vertices and six edges:





Often we'll need to refer to the vertices and edges of a graph, and we can do this by writing names next to each vertex and edge. For example, the graphs in the margin both have vertices named v_0, v_1, v_2, v_3, and v_4, whereas its edges are named e_0, e_1, e_2, e_3, e_4, and e_5. In drawings such as these, the actual placement of the vertices and edges is unimportant. The only important points are which vertices are connected and which edges are used to connect them.

in a drawing, the placement of the vertices and edges is unimportant



Here's a formal definition of these graphs:

> **Undirected Graphs**
>
> An **undirected graph** is a finite set of vertices together with a finite set of edges. Both sets might be empty (no vertices and no edges), which is called the **empty graph**.
>
> Each edge is associated with two vertices. We sometimes say that the edge **connects** its two vertices. The order of the two connected vertices is unimportant, so it does not matter whether we say "This edge connects vertices *u* and *v*," or "This edge connects vertices *v* and *u*."

An edge is even allowed to connect a single vertex to itself—we'll see examples of this later, and we'll also see several different variants of graphs. Many applications require additional data to be attached to each vertex or to each edge, but even without these extras, we can give the flavor of a graph application with an example.

PROGRAMMING EXAMPLE: Undirected State Graphs

As an example of a problem where graphs are useful, we'll look at a little game. To start the game, you place three coins on the table in a line, as shown here:

The start of the game

At the start of the game, the middle coin is "tails" and the other two are "heads." The goal is to change the configuration of coins so that the middle coin is heads and the other two are tails, like this:

The goal of the game

Now, this wouldn't be much of a game without a few good rules. Here are the rules:

1. You may flip the middle coin (from heads to tails or vice versa) whenever you want to.
2. You may flip one of the end coins (from heads to tails or vice versa) only if the other two coins are the same as each other (both heads or both tails).

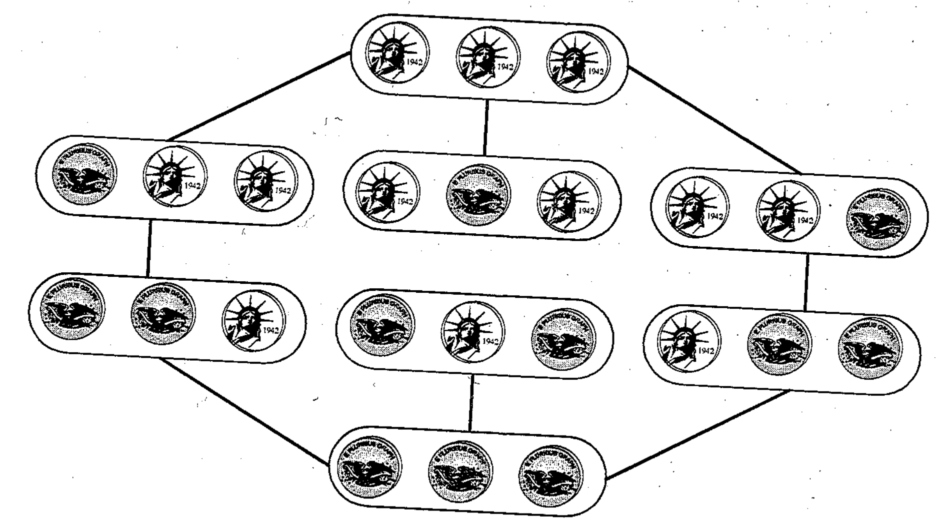
You are not allowed to change the coins in any other way, such as shuffling them around. But within these rules you may flip coins. For example, if you start with the position *head-tail-head*, then the first rule allows you to flip the middle coin, resulting in three heads:

From the start position, flip the middle coin.

If you play the game for a while, you'll soon figure out how to get from the starting position (*head-tail-head*) to the goal position (*tail-head-tail*) within the limits of the rules. But our *real* goal is to figure out how a graph can aid in solving this kind of problem—even if the rules were beyond human manageability. The graph we'll use is called an undirected state graph, which is a graph in which each of the vertices represents one of the possible configurations of the game. These configurations are called "states," and the coin game has eight states ranging from *head-head-head* to *tail-tail-tail*. Therefore, these eight states are the vertices of the state graph for the coin game. Figure 14.1 shows each vertex as a large oval so that we have room inside the oval to draw the state of the coins. Two vertices of the undirected state graph are connected by an edge

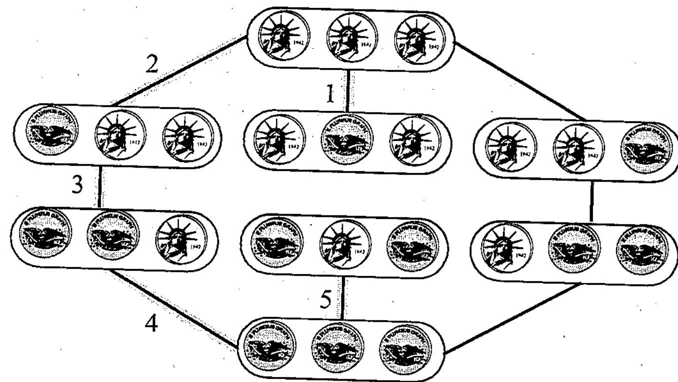
a graph ma represent legal move a game

FIGURE 14.1 Undirected State Graph for the Coin Game



whenever it is possible to move back and forth between the two states using one of the rules. For example, Rule 1 allows us to move between *head-head-head* and *head-tail-head*, so one of the edges in the graph goes from the topmost state in Figure 14.1 (*head-head-head*) to the state directly below it (*head-tail-head*). Figure 14.1 shows a total of eight edges: The four vertical edges come from Rule 1, and the four diagonal edges come from Rule 2. You may have noticed a curious fact about our rules: Whenever it is possible to move from one state to another (such as moving *head-head-head* to *head-tail-head*), it is also possible to move in the other direction (such as *head-tail-head* to *head-head-head*). If you study the rules, you will see that this is true. The way we have drawn the edges reflects this symmetry. The edges are drawn as line segments connecting two vertices, with no indication as to which direction a movement must proceed. In this game, if an edge connects two vertices v_0 and v_1, then a movement is permitted in both directions, from v_0 to v_1 or from v_1 to v_0. This property of the coin game is the reason why we can use an undirected state graph. Later we will see more complex games in which movements might be permitted in only one direction, and hence we will need more complex graphs.

Once we know the undirected state graph, the game becomes a problem of finding a path from one vertex to another, where the path is allowed only to follow edges. According to our rules, we need to find a path from the vertex *head-tail-head* to the vertex *tail-head-tail*, and one such path consists of edges 1 through 5, highlighted here:



The coin game is a small problem, and the state graph isn't vital to the solution. But the important idea of the example goes beyond this small game.

> **Graphs in Problem Solving**
>
> Often a problem can be represented as a graph, and the solution to the problem is obtained by solving a problem on the corresponding graph.

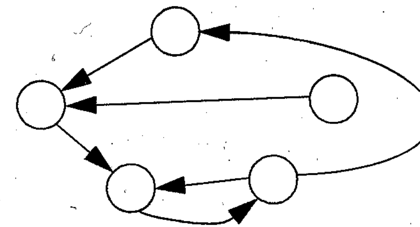
Because of this wide applicability of graphs, we will study many different kinds of graphs, exploring how to implement the graphs and how to solve problems such as "Is there a path from here to there?" The rest of this section shows the kinds of graphs we'll study and some of the problems we'll solve.

Directed Graphs

The graphs we have seen so far are all *undirected*, which means that each edge connects two vertices with no particular orientation or direction. An edge just connects two vertices—there is no "first vertex" or "second vertex." But there is another kind of graph called a *directed graph* in which each edge has an orientation connecting its first vertex (called the edge's **source**) to its second vertex (the edge's **target**). Here is the formal definition of a directed graph:

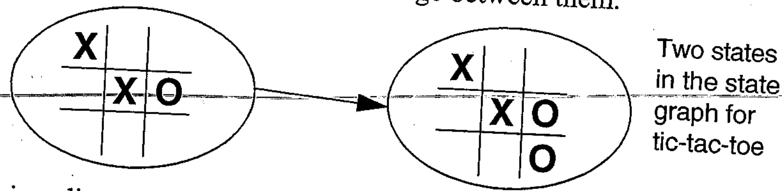
> **Directed Graphs**
>
> A **directed graph** is a finite set of vertices together with a finite set of edges. Both sets might be empty (no vertices and no edges), which is called the **empty graph**.
>
> Each edge is associated with two vertices, called its **source** and **target** vertices. We sometimes say that the edge **connects** its source to its target. The order of the two connected vertices *is* important, so it *does* matter whether we say "This edge connects vertex *u* to vertex *v*," or "This edge connects vertex *v* to vertex *u*."

Directed graphs are drawn as diagrams with circles representing the vertices and *arrows* representing the edges. Each arrow starts at an edge's source and has the arrowhead at the edge's target. For example:



One application of directed graphs is a state graph for a game where reversing a move is sometimes forbidden. For such a game, the state graph might have an edge from a source v_1 to a target v_2 but *not* include the reverse edge from v_2 to v_1. This would indicate that the game's rules permit a move from state v_1 to state

v_2 but not the other way around. For example, we could use a large state graph to represent the different states in a game of tic-tac-toe. Two of the many possible states are shown next, with a directed edge between them.

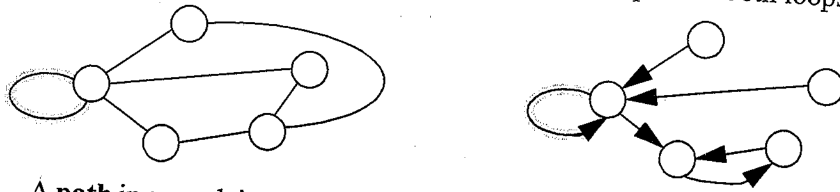


Two states in the state graph for tic-tac-toe

There is a directed arrow between these two states since it is possible to move from the first state to the second state (by placing an O), but it is not possible to move in the other direction (since once an O is placed, it may not be removed).

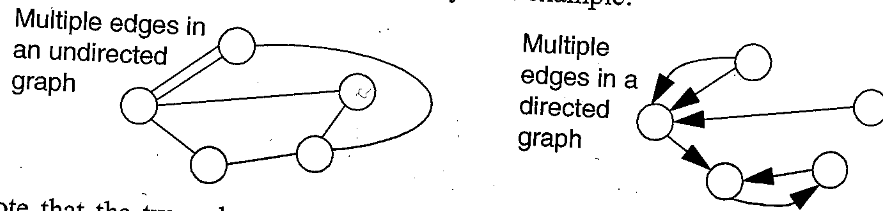
More Graph Terminology

Loop. A **loop** is an edge that connects a vertex to itself. In the diagrams that we've been using, this is drawn as a line (or arrow) with both ends at the same location. The highlighted edges at the left of these two graphs are both loops:



Path. A **path** in a graph is a sequence of vertices, $p_0, p_1, \ldots p_m$, such that each adjacent pair of vertices p_i and p_{i+1} are connected by an edge. In a directed graph, the connection must go from the source p_i to the target p_{i+1}.

Multiple Edges. In principle, a graph can have two or more edges connecting the same two vertices in the same direction. These are called **multiple edges**. In a diagram, each edge is drawn separately. For example:



Note that the two edges at the bottom of the directed graph are not multiple edges because they connect the two vertices in different directions. Many applications do not require multiple edges. In fact, many implementations of graphs do not permit multiple edges. Throughout the graph implementations of this chapter, we will specify which implementations permit multiple edges and which implementations forbid them.

Simple Graphs. The simplest of graphs have no loops and no multiple edges. Appropriately enough, these graphs are called **simple** graphs. Many applications require only simple directed graphs or even simple undirected graphs.

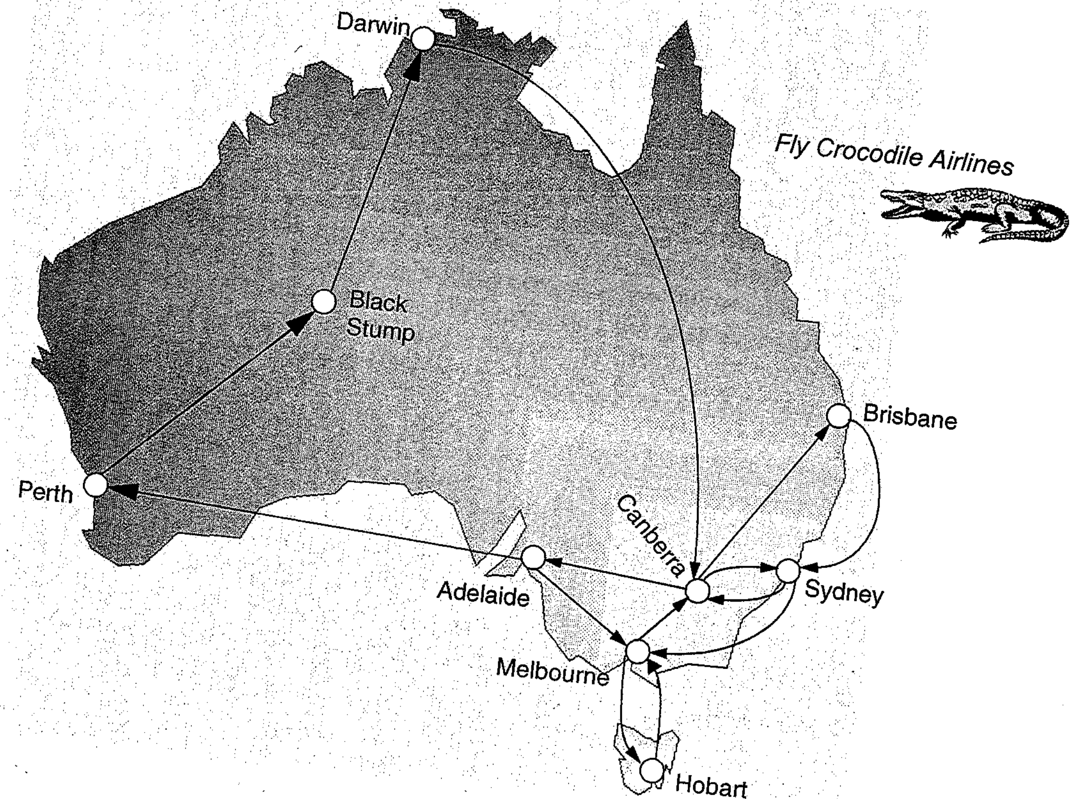
Airline Routes Example

airline routes form a direct graph

Let's examine a directed graph that represents the flights of a small airline called Crocodile Airlines. Each vertex in the graph is a city, and each edge represents a regularly scheduled flight from one city to another. Crocodile's complete collection of flights is shown in Figure 14.2. Notice that the graph is directed; for example, it's possible to fly from Darwin to Canberra on a nonstop flight but not the other way around.

The point of expressing the flights as a graph is that questions about the airline can be answered by carrying out common algorithms on the graph. For example, a sheep farmer might wonder what is the fewest number of flights required to fly from Black Stump to Melbourne. This is an example of a *shortest path* problem that we will solve in Section 14.4. With a small graph such as Figure 14.2, you can probably see that the shortest path from Black Stump to Melbourne consists of four edges, but a manual examination might not suffice for a larger graph.

FIGURE 14.2 Crocodile Airlines Routes



Self-Test Exercises for Section 14.1

1. How many vertices and edges does the graph in Figure 14.2 have? How many loops? How would you interpret a loop in this graph?
2. Is Figure 14.2 a simple graph? Why or why not?
3. Suppose we have four coins in the coin game instead of just three. At the start of the game, the coins are in a line, with the two end coins heads and the other two coins tails. The goal is to change the configuration so that the two end coins are tails and the other two coins are heads. There are three rules for this game: (1) Either of the end coins may be flipped whenever you want to; (2) A middle coin may be flipped from heads to tails only if the coin to its immediate right is already heads; (3) A middle coin may be flipped from tails to heads only if the coin to its immediate left is already tails. Your mission: Draw the directed state graph for this game and determine whether it is possible to go from the start configuration to the goal. Why does the graph need to be directed?

14.2 GRAPH IMPLEMENTATIONS

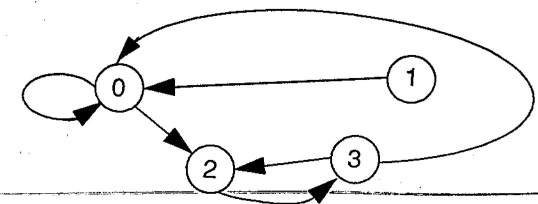
"I could spin a web if I tried," said Wilbur, boasting. "I've just never tried."

E. B. WHITE
Charlotte's Web

Different kinds of graphs require different kinds of implementations, but the fundamental concepts of all graph implementations are similar. We'll look at several representations for one particular kind of graph: directed graphs in which loops are allowed. Some of the representations allow multiple edges, and some do not. In each of the representations, the vertices of the graph are named with numbers 0, 1, 2,..., so that we can refer to each vertex individually by saying "vertex 0," "vertex 1," and so on.

Representing Graphs with an Adjacency Matrix

Let's start with a directed graph with no multiple edges. The graph's edges can be represented in a square grid of boolean (`true`/`false`) values, called the graph's *adjacency matrix*. The top of the next page shows an example of a graph with four vertices and its adjacency matrix.



	0	1	2	3
0	*true*	*false*	*true*	*false*
1	*true*	*false*	*false*	*false*
2	*false*	*false*	*false*	*true*
3	*true*	*false*	*true*	*false*

Each component of the adjacency matrix indicates whether a certain edge is present. For example, is there an edge from vertex 0 to vertex 2? Yes, because `true` appears at row 0 of column 2. But is there also an edge from vertex 2 to vertex 0? No, because `false` appears at row 2 of column 0. The general rule for using an adjacency matrix is given here:

a true component at row i and column j indicates an edge from vertex i to vertex j

Adjacency Matrix

An **adjacency matrix** is a square grid of `true`/`false` values that represent the edges of a graph. If the graph contains *n* vertices, then the grid contains *n* rows and *n* columns. For two vertex numbers *i* and *j*, the component at row *i* and column *j* is `true` if there is an edge from vertex *i* to vertex *j*; otherwise, the component is `false`.

Using a Two-Dimensional Array to Store an Adjacency Matrix

In Java, an adjacency matrix can be stored in a **two-dimensional array** in which every component has two indexes rather than the usual one index. Programmers usually view a two-dimensional array as a grid of elements, where the first index provides the row number of a component and the second index provides the column number of a component. In a declaration of a two-dimensional array, the data type of the components is followed by two pairs of brackets, as shown here:

in a two-dimensional array, every component has two indexes

```
double[ ][ ] budget;
```

As with any array, the declaration just provides a reference variable that is capable of refering to a two-dimensional array. The actual array must be allocated with the new operator. For example, this allocates a two-dimensional array of double numbers with 12 rows and 8 columns, as declared here:

```
budget = new double [12][8];
```

allocating a two-dimensional array with 12 rows and 8 columns

Within the program, the individual components of a two-dimensional array can be accessed by providing both indexes. For example, we can assign the value 3.14 to row number 2 and column number 6 with this assignment statement:

```
budget[2][6] = 3.14;
```

As with an ordinary array, the index numbers begin with zero, so our budget has rows numbered from 0 to 11 and columns numbered from 0 to 7.

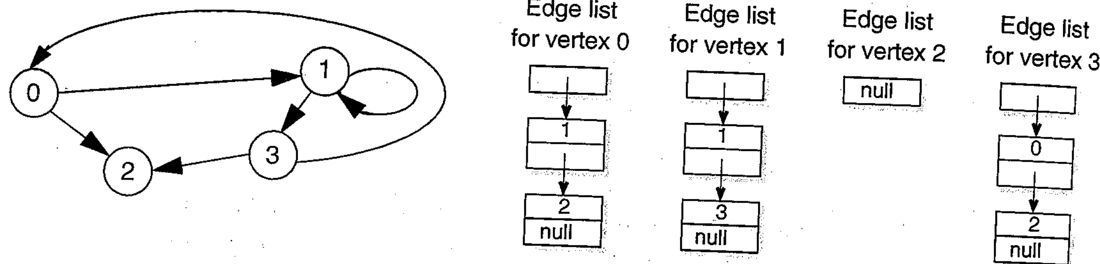
For an adjacency matrix, we use a two-dimensional array with one row and one column for each vertex of the graph. The components of the array are boolean true/false values. For example, the adjacency matrix for a graph with four vertices can be stored using this declaration of a two-dimensional array:

```
boolean[ ][ ] adjacent = new boolean[4][4];
```

The location adjacent[i][j] contains true if there is an edge from vertex i to vertex j. A false value indicates no edge. Edges of a graph can be added (by placing true in a location of the matrix) or removed (by placing false). Once the adjacency matrix has been set, an application can examine locations of the matrix to determine which edges are present and which are missing.

Representing Graphs with Edge Lists

Again, suppose we have a directed graph with no multiple edges. Such a graph can be represented by creating a linked list for each vertex. Here's an example:



To understand the example, look at the linked list for vertex number 0. This list contains 1 and 2, which means that there is an edge from vertex 0 to vertex 1 and a second edge from vertex 0 to vertex 2. In general, the linked list for vertex number i is a list of vertex numbers following this rule:

> **Linked List Representation of Graphs**
>
> A directed graph with n vertices can be represented by n different linked lists. List number i provides the connections for vertex i. To be specific: For each entry j in list number i, there is an edge from i to j.

Loops are allowed in this representation; for example, look at the list for vertex 1 in the preceding edge lists. Number 1 appears on this list itself, so there is an edge from vertex 1 to vertex 1 in the graph. We may also have vertices that are not the source of any edge—for example, vertex 2 in the preceding graph. Since vertex 2 is not the source of any edge, the edge list for vertex 2 is empty. If we allow the same element to appear more than once on the list, then multiple edges could also be allowed. But often multiple edges are not allowed.

In a graph with n vertices, there are n edge lists. References to the heads of the n edge lists can be stored in an array of n node variables. When we need to determine whether an edge exists from vertex i to vertex j, we check to see whether j appears on list number i.

Representing Graphs with Edge Sets

Another implementation of graphs requires a previously programmed implementation of a set ADT. For example, suppose we have declared IntSet as a class capable of holding a set of integers. Presumably, the set class has operations to add an integer, check whether an integer is in the set, remove an integer, and so forth. To represent a graph with 10 vertices, we can declare an array of 10 sets of integers, as shown here:

```
IntSet[ ] connections = new IntSet[10];
```

Using this representation, a set such as connections[i] contains the vertex numbers of all of the vertices to which vertex i is connected. For example, suppose connections[3] contains the numbers 1 and 2. In this case, there is an edge from vertex 3 to vertex 1 and another edge from vertex 3 to vertex 2.

Which Representation Is Best?

If the space is available, then an adjacency matrix is easier to implement and is generally easier to use than edge lists or edge sets. But sometimes there are other considerations. For example, how often will you be doing each of the following operations?

1. Adding or removing edges
2. Checking whether a particular edge is present
3. Iterating a loop that executes one time for each edge with a particular source vertex

The first two operations require only a small constant amount of time with the adjacency matrices. But in the worst case, both (1) and (2) require $O(n)$ operations with the edge list representation (where n is the number of vertices). This worst case occurs when the operation must traverse an entire edge list, and that edge list might contain as many as n edges. With edge sets, both (1) and (2) might also require $O(n)$ operations—but this time could be cut to $O(\log n)$ by using a fast set representation such as B-trees from Section 10.2.

On the other hand, the edge lists are efficient for the third operation. With an edge list, the third operation can be carried out by ...

list one element at a time. The time required is $O(e)$, where e is the number of edges that have vertex i as their source. A good set ADT should also provide the operations to step through the elements of the set one at a time. In this way, both edge lists and edge sets are likely to require just $O(e)$ operations to step through the edges with a particular source vertex. But with an adjacency matrix, the act of stepping through the edges with source vertex i requires that each component of row i be examined. This traversal of the entire row is necessary just to see whether each entry is `true` or `false`. This always requires $O(n)$ time, where n is the number of vertices.

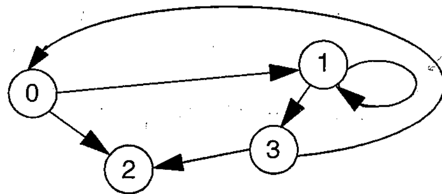
In general, your choice of representations should be based on your expectations as to which operations are most frequent. The availability of a previously programmed set ADT may also affect your choice. One last consideration is the average number of edges originating at a vertex. If each vertex has only a few edges (a so-called **sparse** graph), then an adjacency matrix is mostly wasted space filled with the value `false`.

Programming Example: Labeled Graph ADT

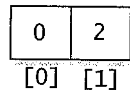
Now we'll implement a class called `Graph` for directed graphs with no multiple edges. This first ADT has only a few methods; for example, the number of vertices is set by the constructor, and there is no way to add or remove vertices later. An n-vertex graph always has its vertices numbered from zero to $n-1$. This simple form of the first `Graph` class provides a sharp focus on graphs and their algorithms.

With our `Graph` class, there are methods to add or remove edges between the vertices. The edges are stored in an adjacency matrix, which is implemented as the two-dimensional boolean array called `edges`. The `edges` array is allocated within the constructor, based on the number of vertices required for the graph.

Even though we're implementing our graph using an adjacency matrix, the class will also have a method, called `neighbors`, that computes an edge list for a specified vertex. This edge list will be provided as an array of integers that contains all of the vertex numbers for the targets of edges that start at a specified source. Here's an example to show the integer array that's computed and returned by `g.neighbors(3)` for a particular graph:



If g is the graph shown here,
then `g.neighbors(3)` returns the array:



This means that vertex 3 is the source of two edges, going to vertices 0 and 2.

The `neighbors` method may take $O(n)$ time to compute one of its arrays, but once an array is available, it can be used to quickly traverse all the neighbors of a vertex. Notice that the length of the array computed by `g.neighbors(i)` is equal to the number of edges that have vertex `i` as the source, and this could even be zero (resulting in the unusual occurrence of an array with zero components).

Our `Graph` class will have one extra feature: The vertices will have information attached to them in the same way that we attached information to tree nodes. For example, in the airline route graph of Figure 14.2 on page 707, each vertex is associated with a city name. The addition of information at each vertex makes the graph a **labeled graph**, and the information itself is called a vertex's **label**. Of course, the information might be any data type—integers, doubles, strings, you name it—which suggests that the labeled graph should be a generic class with the type of the label determined by the generic type parameter. This does not allow us to use the eight primitive types for labels, though if we do want something like an integer label, we could use the `Integer` wrapper class described on page 247.

The labels of the vertices will be stored in a private instance variable named `labels`, which is an ordinary one-dimensional array. The label for vertex number `i` will be stored in `labels[i]`. Thus, our graph implementation has a total of two instance variables, as shown here:

```
public class Graph<E>
{
   private boolean[ ][ ] edges;
   private E[ ] labels;

   // We'll discuss the methods in a moment.
}
```

The Graph Constructor and size Method

When a graph is initialized with the constructor, it has a given number of vertices and no edges, as shown in this specification of the constructor:

◆ **Constructor for the Graph<E>**
`public Graph(int n)`
Initialize a `Graph` with `n` vertices, no edges, and null labels.
Parameters:
`n` – the number of vertices for this `Graph`
Precondition:
`n >= 0`.
Postcondition:
This `Graph` has `n` vertices, numbered from `0` to `n-1`. It has no edges and all vertex labels are null.
Throws: `OutOfMemoryError`
Indicates insufficient memory to create this `Graph`.
Throws: `NegativeArraySizeException`
Indicates that `n` is negative.

The only work in the implementation is the allocation of the two instance variables, which are both arrays, as shown here:

```
public Graph(int n)
{
    edges = new boolean[n][n];      // All values initially false
    labels = (E[]) new Object[n]; // All values initially null
}
```

After construction, the current number of vertices can be obtained from an accessor method named `size`.

Methods for Manipulating Edges

Two methods, `addEdge` and `removeEdge`, allow us to add and remove edges. One other method, called `isEdge`, determines whether a specified edge is present. The methods have these specifications:

◆ **addEdge —and— isEdge —and— removeEdge**

`public void addEdge(int source, int target)`
`public boolean isEdge(int source, int target)`
`public void removeEdge(int source, int target)`

Add an edge, test whether an edge exists, or remove an edge of this `Graph`.

Parameters:
`source` – the vertex number of the source of the edge
`target` – the vertex number of the target of the edge

Precondition:
Both `source` and `target` are non-negative and less than `size()`.

Postcondition:
For `addEdge`, the specified edge is added to this `Graph` (unless it was already present); for `isEdge`, the return value is `true` if the specified edge exists and is `false` otherwise; for `removeEdge`, the specified edge is removed from this `Graph` (unless it was already not present).

Throws: `ArrayIndexOutOfBoundsException`
Indicates that the `source` or `target` was not a valid vertex number.

The implementations of these methods will be short. The `addEdge` method sets `edges[source][target]` to `true`, and the `removeEdge` method sets it to `false`. The `isEdge` method just returns the current value of `edges[source][target]`. In all cases, an illegal parameter will try accessing `edges[source][target]` and will result in an `ArrayIndexOutOfBoundsException`.

As we have already discussed, there is also a method called `neighbors` to generate an array that contains an edge list for a specified vertex.

Methods for Manipulating Vertex Labels

There are two methods for setting and retrieving the label of a vertex:

◆ **getLabel**

`public E getLabel(int vertex)`

Accessor method to get the label of a vertex of this `Graph`.

Parameters:
`vertex` – a vertex number

Precondition:
`vertex` is non-negative and less than `size()`.

Returns:
the label of the specified vertex in this `Graph`

Throws: `ArrayIndexOutOfBoundsException`
Indicates that the `vertex` was not a valid vertex number.

◆ **setLabel**

`public void setLabel(int vertex, E newLabel)`

Change the label of a vertex of this `Graph`.

Parameters:
`vertex` – a vertex number
`newLabel` – a vertex number

Precondition:
`vertex` is non-negative and less than `size()`.

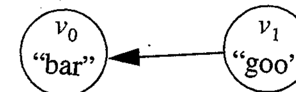
Postcondition:
The label of the specified `vertex` in this `Graph` has been changed to `newLabel`.

Throws: `ArrayIndexOutOfBoundsException`
Indicates that the `vertex` was not a valid vertex number.

For example, the following statements create the two-vertex graph shown in the picture. In this graph, vertex number zero (v_0) has the string "bar" as its label, and vertex number one (v_1) has the string "goo" as its label:

```
Graph<String> t = new Graph<String>(2);

t.setLabel(0, "bar"); // Provides vertex number 0 with label of "bar"
t.setLabel(1, "goo"); // Provides vertex number 1 with label of "goo"
t.addEdge(1, 0);      // Adds an edge from vertex 1 to vertex 0
```



Labeled Graph ADT—Implementation

The complete specification and implementation for the graph ADT are given in Figure 14.3. In addition to the methods we have already mentioned, the `Graph` class also implements the `Cloneable` interface.

FIGURE 14.3 Specification and Implementation of the Graph Class

Generic Class Graph

❖ **public class Graph<E> from the package edu.colorado.graphs**
A `Graph<E>` is a labeled graph with a fixed number of vertices and labels of type E.

Specification

♦ **Constructor for the Graph<E>**
`public Graph(int n)`
Initialize a `Graph` with `n` vertices, no edges, and null labels.
Parameters:
`n` – the number of vertices for this `Graph`
Precondition:
`n >= 0.`
Postcondition:
This `Graph` has `n` vertices, numbered from `0` to `n-1`. It has no edges and all vertex labels are null.
Throws: `OutOfMemoryError`
Indicates insufficient memory to create this `Graph`.
Throws: `NegativeArraySizeException`
Indicates that `n` is negative.

♦ **addEdge —and— isEdge —and— removeEdge**
```
public void addEdge(int source, int target)
public boolean isEdge(int source, int target)
public void removeEdge(int source, int target)
```
Add an edge, test whether an edge exists, or remove an edge of this `Graph`.
Parameters:
`source` – the vertex number of the source of the edge
`target` – the vertex number of the target of the edge
Precondition:
Both `source` and `target` are non-negative and less than `size()`.
Postcondition:
For `addEdge`, the specified edge is added to this `Graph` (unless it was already present); for `isEdge`, the return value is `true` if the specified edge exists and is `false` otherwise; for `removeEdge`, the specified edge is removed from this `Graph` (unless it was already not present).
Throws: `ArrayIndexOutOfBoundsException`
Indicates that the `source` or `target` was not a valid vertex number.

(continued)

(FIGURE 14.3 continued)

♦ **clone**
`public Graph<E> clone( )`
Generate a copy of this `Graph`.
Returns:
The return value is a copy of this `Graph`. Subsequent changes to the copy will not affect the original, nor vice versa. The return value must be typecast to a `Graph` before it is used.
Throws: `OutOfMemoryError`
Indicates insufficient memory for creating the clone.

♦ **getLabel**
`public E getLabel(int vertex)`
Accessor method to get the label of a vertex of this `Graph`.
Parameters:
`vertex` – a vertex number
Precondition:
`vertex` is non-negative and less than `size()`.
Returns:
the label of the specified vertex in this `Graph`
Throws: `ArrayIndexOutOfBoundsException`
Indicates that the `vertex` was not a valid vertex number.

♦ **neighbors**
`public int[ ] neighbors(int vertex)`
Accessor method to obtain a list of neighbors of a specified vertex of this `Graph`.
Parameters:
`vertex` – a vertex number
Precondition:
`vertex` is non-negative and less than `size()`.
Returns:
The return value is an array that contains all the vertex numbers of vertices that are targets of edges with a source at the specified vertex.
Throws: `ArrayIndexOutOfBoundsException`
Indicates that the `vertex` was not a valid vertex number.

(continued)

```
public void setLabel(int vertex, E newLabel)
```
Change the label of a vertex of this Graph.

Parameters:
vertex – a vertex number
newLabel – a vertex number

Precondition:
vertex is non-negative and less than size().

Postcondition:
The label of the specified vertex in this Graph has been changed to newLabel.

Throws: ArrayIndexOutOfBoundsException
Indicates that the vertex was not a valid vertex number.

♦ **size**

```
public int size( )
```
Accessor method to determine the number of vertices in this Graph.

Returns:
the number of vertices in this Graph

Implementation

```
// File: Graph.java from the package edu.colorado.graphs
// Complete documentation is on pages 716—717 or from the Graph link in
//    http://www.cs.colorado.edu/~main/docs/

package edu.colorado.graphs;

public class Graph<E> implements Cloneable
{
   // Invariant of the Graph<E> class:
   //    1. The vertex numbers range from 0 to labels.length-1.
   //    2. For each vertex number i, labels[i] contains the label for vertex i.
   //    3. For any two vertices i and j, edges[i][j] is true if there is a vertex from i to j;
   //       otherwise edges[i][j] is false.
   private boolean[ ][ ] edges;
   private E[ ] labels;

   public Graph(int n)
   {
      edges = new boolean[n][n];        // All values initially false
      labels = (E[]) new Object[n]; // All values initially null
   }
```

(continued)


```
   public void addEdge(int source, int target)
   {
      edges[source][target] = true;
   }

   public Graph<E> clone( )
   {  // Clone a Graph<E> object.
      Graph<E> answer;

      try
      {
         answer = (Graph<E>) super.clone( );
      }
      catch (CloneNotSupportedException e)
      {
         // This exception should not occur. But if it does, it would probably indicate a
         // programming error that made super.clone unavailable. The most common
         // error would be forgetting the "Implements Cloneable"
         // clause at the start of this class.
         throw new RuntimeException
         ("This class does not implement Cloneable");
      }

      answer.edges = edges.clone( );
      answer.labels = labels.clone( );

      return answer;
   }

   public E getLabel(int vertex)
   {
      return labels[vertex];
   }

   public boolean isEdge(int source, int target)
   {
      return edges[source][target];
   }
```

(continued)

(FIGURE 14.3 continued)

```
    public int[ ] neighbors(int vertex)
    {
        int i;
        int count;
        int[ ] answer;

        // First count how many edges have the vertex as their source
        count = 0;
        for (i = 0; i < labels.length; i++)
        {
            if (edges[vertex][i])
                count++;
        }

        // Allocate the array for the answer
        answer = new int[count];

        // Fill the array for the answer
        count = 0;
        for (i = 0; i < labels.length; i++)
        {
            if (edges[vertex][i])
                answer[count++] = i;
        }

        return answer;
    }

    public void removeEdge(int source, int target)
    {
        edges[source][target] = false;
    }

    public void setLabel(int vertex, E newLabel)
    {
        labels[vertex] = newLabel;
    }

    public int size( )
    {
        return labels.length;
    }
}
```


Self-Test Exercises for Section 14.2

4. Write a new Graph method to remove a specified number of vertices from a graph. The removed vertices should be the k vertices with the highest numbers. Write another method to add a specified number of new vertices.
5. Write a new Graph method that interchanges two specified vertices. For example, after interchanging vertices i and j, the original neighbors of vertex i will now be neighbors of vertex j and vice versa.
6. We have assumed that there are no multiple edges in a graph, and therefore we could store simple true/false in the adjacency matrix. Describe how adjacency matrixes might be used even if the graphs do have multiple edges.
7. Compare and contrast adjacency lists and edge lists.
8. Write a generic static method with one Graph<E> parameter g. The precondition requires g to have at least one vertex. The method's return value is the number of the vertex that is the source of the most edges. If there are several vertices with an equally high number of edges, then you may return whichever vertex number you prefer.

14.3 GRAPH TRAVERSALS

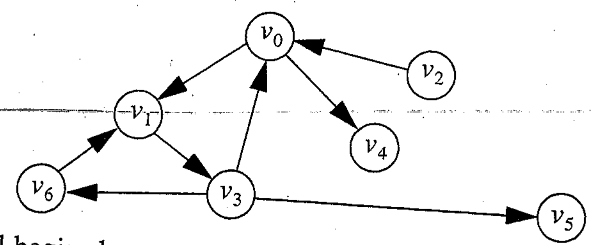
In the chapter on trees, we saw three different binary tree traversals. Each traversal visits all of a binary tree's nodes and does some processing at each node. The three traversals had similar recursive implementations, with the distinguishing factor being whether a node was visited before, after, or in between its two children. A graph vertex doesn't have children like a tree node, so the tree traversal algorithms are not immediately applicable to graphs. But there are two common ways of traversing a graph. One of the methods (breadth-first search) uses a queue to keep track of vertices that still need to be visited, and the other method (depth-first search) uses a stack. The depth-first search can also be implemented recursively in a way that does not explicitly use a stack of vertices.

This section discusses the two traversal algorithms in a general way and then provides implementations of the algorithms to traverse a Graph object (using the Graph class; see Figure 14.3 on page 716). Both of the traversal algorithms have the same underlying purpose: to start at one vertex of a graph (the "start" vertex), process the information contained at that vertex, and then move along an edge to process a neighbor. When the traversal finishes, all of the vertices that can be reached from the start vertex (via some directed path) have been processed.

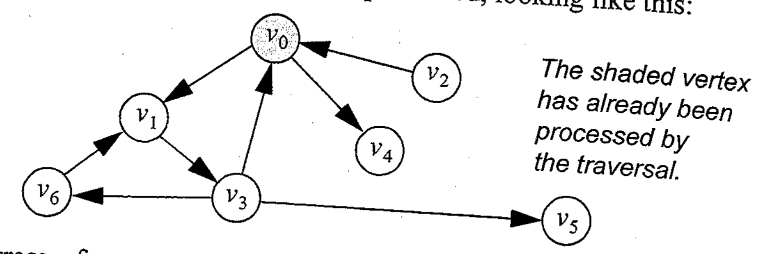
when the traversal finishes, all of the vertices that can be reached from the start vertex have been processed

A traversal algorithm must be careful that it doesn't enter a repetitive cycle—for example, moving from the start vertex to a neighbor, from there to the neighbor's neighbor, from there back to the starting point and then back to the same neighbor, and so on. To prevent this potential "spinning your wheels," we will

...the ability to *mark* each vertex as it is processed. If a traversal ever returns to a vertex that is already marked, then reprocessing is not done. If we are drawing the graphs, then we might indicate a marked vertex by shading it. For example, suppose a traversal starts at vertex 0 of this graph:



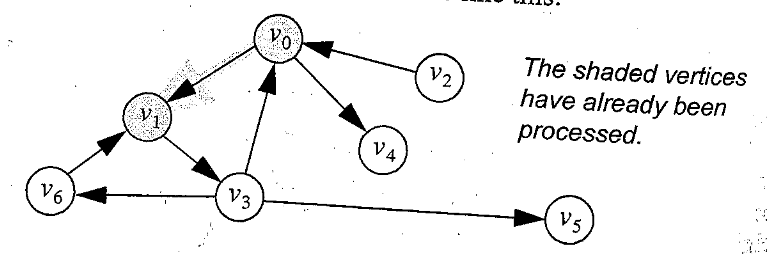
The traversal begins by processing vertex number 0. We don't really care what kind of processing occurs—maybe the labels are printed out, or perhaps there is more complicated processing. In any case, there will be some processing of vertex number 0, and then we will mark it as processed, looking like this:



The progress of a traversal after the start vertex depends on the traversal method being used. We'll start by looking at how a depth-first search proceeds.

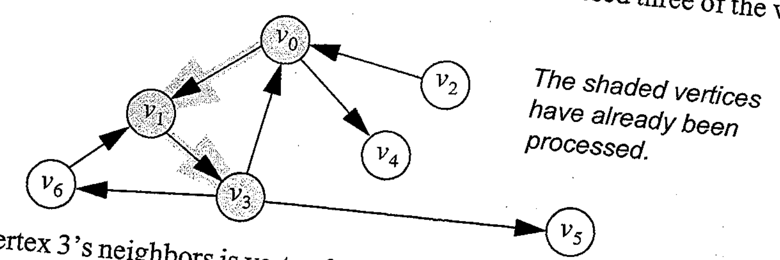
Depth-First Search

After processing vertex 0, a depth-first search moves along a directed edge to one of vertex 0's neighbors. In our example, there are two possibilities: moving to vertex 1 (along the edge from 0 to 1) or to vertex 4 (along the edge from 0 to 4). In our example, it is not possible to move from vertex 0 to 3 or from vertex 0 to 2 because the edges go in the wrong direction. So the traversal has a choice: Move to vertex 1 or move to vertex 4. Right now, we won't worry about exactly how the choice is made—let's just assume that the next vertex processed is vertex 1. After processing vertex 1, the picture looks like this:



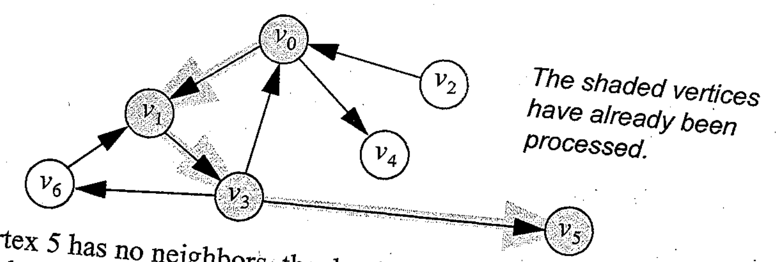
In the drawing, we have highlighted the edge from vertex 0 to vertex 1 to indicate the intuitive notion of "moving from one vertex to another."

From here, the traversal moves to one of vertex 1's neighbors. You may think of vertex 1 as being at the "leading edge" of the depth-first traversal. It is the vertex that has most recently been processed, so we will continue pushing forward from vertex 1, moving to one of vertex 1's unprocessed neighbors. In this example, there is only one unprocessed neighbor to consider, vertex 3. So we will move to vertex 3 and process it. At this point, we have processed three of the vertices, as shown here:

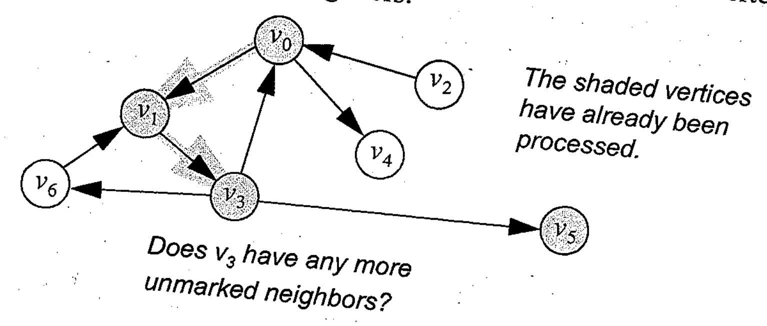


One of vertex 3's neighbors is vertex 0—but vertex 0 has already been marked as previously processed. So, to prevent the traversal from going around in circles, we will not move from 3 back to 0. In general, *we will never move forward to a marked vertex (since it has already been processed)*. But we will move to vertex 3's other neighbors (vertices 5 and 6). Suppose we move to vertex 5 first. After processing vertex 5, the picture looks like this:

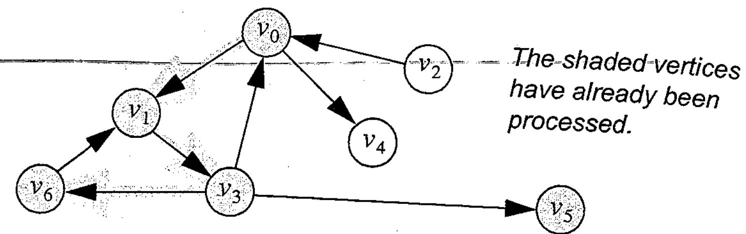
to prevent going around in circles, never move forward to a marked vertex



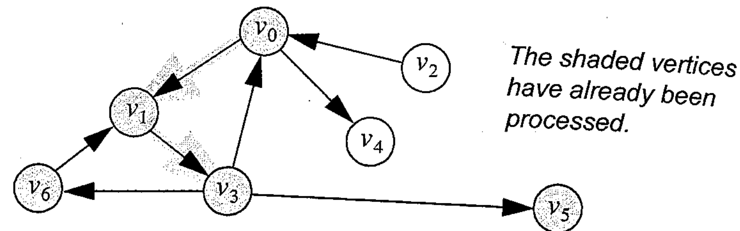
Since vertex 5 has no neighbors, the depth-first traversal cannot proceed forward any farther. Instead, the traversal comes back to see if the previous vertex—vertex 3—has any more unmarked neighbors:



In this drawing, you can see that four vertices have been processed, and the "leading edge" of the search has pulled back to vertex v_3. Does v_3 have any more unmarked neighbors where the search can proceed? Yes—v_6 is an unmarked neighbor of v_3, so again the search plunges forward, along the edge from v_3 to v_6. Vertex 6 is processed, giving the following picture:

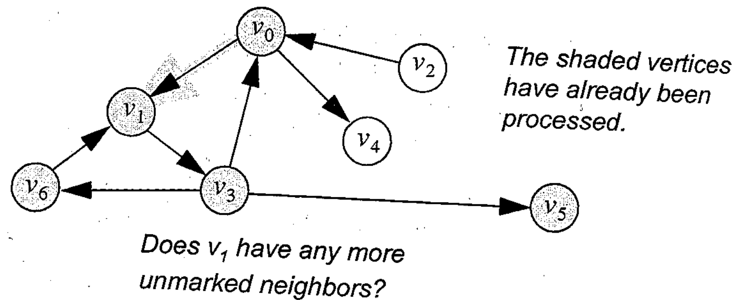


What now? Vertex 6 does have a neighbor—vertex 1—but v_1 has already been marked. So, since vertex 6 has no *unmarked* neighbors, the traversal again backs up to see if v_3 has any more unmarked neighbors. After the backup, we have this situation:



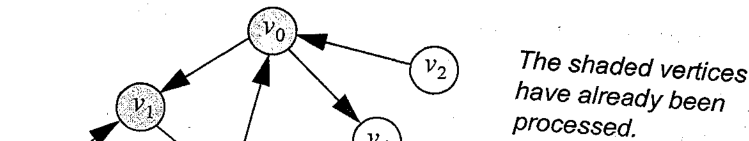
Does v_3 have any more unmarked neighbors?

Vertex 3 has no more unmarked neighbors (thank goodness!). So back we go to the previous vertex—vertex 1—to check whether it has any unmarked neighbors:



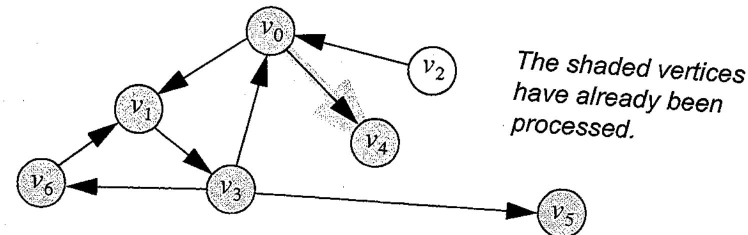
Does v_1 have any more unmarked neighbors?

You can see that the leading edge of the search is now at v_1, and that v_1 has no more unmarked neighbors, so back we go to vertex 0 to see if it has any unfinished business. This situation is drawn at the top of the next page.

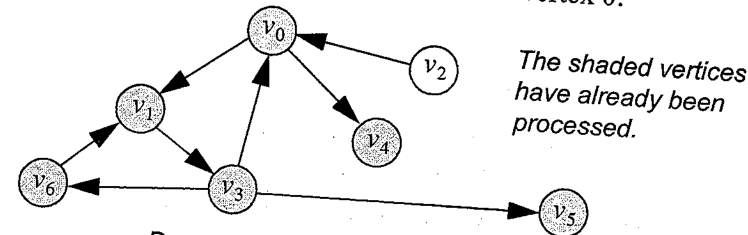


Does v_0 have any more unmarked neighbors?

From vertex 0, we can still travel to the unmarked vertex 4 and process it, as shown here:



Vertex 4 has no neighbors, so we back up once more to vertex 0:



Does v_0 have any more unmarked neighbors?

Vertex 0 has no more unmarked neighbors—and since this was the starting point, there is no place left to back up to. That's the end of the traversal.

In this example, vertex 2 was never processed because there was no path from the start vertex (vertex 0) to vertex 2. A traversal processes only those vertices that can be reached from the start vertex.

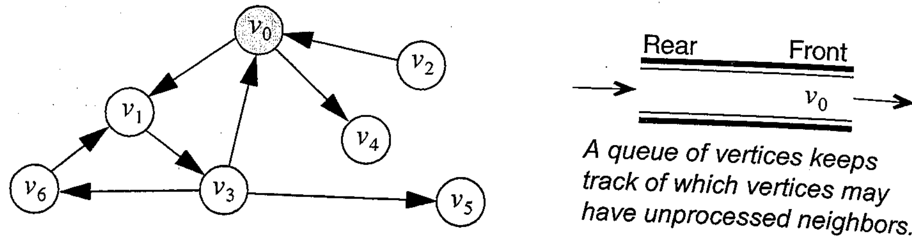
some vertices are not processed because they can't be reached from the start vertex

There's one other important point about a depth-first search: From the start vertex, the traversal proceeds to a neighbor and from there to another neighbor and so on—always going as far as possible before it ever backs up. In describing this behavior, it seems as if there's a lot to keep track of: where we start, where we go from there, and where we go from there. But the actual implementation can use recursion to keep track of most of these details in a simple way. We'll tackle that recursive implementation after looking at an alternative method: breadth-first search.

By the way, did you notice that the maze problem from Section 8.2 was solved by a recursive implementation of a depth-first search?

Breadth-First Search

A breadth-first search uses a queue to keep track of which vertices might still have unprocessed neighbors. The search begins with a starting vertex, which is processed, marked, and placed in the queue. For example, suppose we are processing this graph with vertex 0 as our starting point so that vertex 0 is the first vertex to be processed, marked, and placed into the queue:

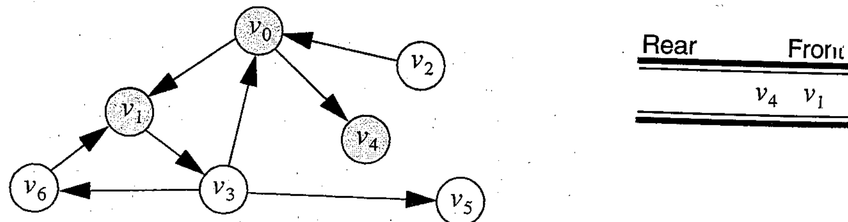


A queue of vertices keeps track of which vertices may have unprocessed neighbors.

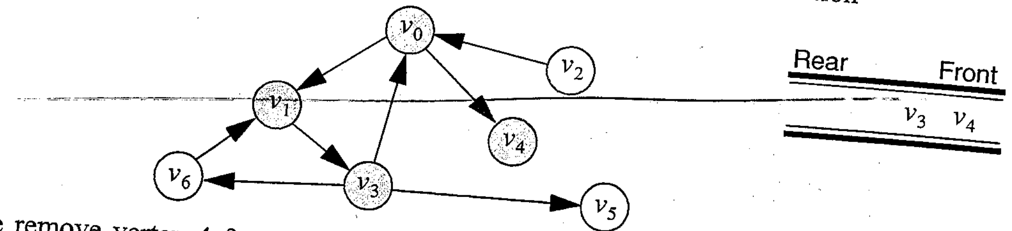
Once the starting vertex has been processed, marked, and placed in the queue, the main part of the breadth-first search begins. This consists of repeatedly carrying out the following steps:

1. Remove a vertex, v, from the front of the queue.
2. For each unmarked neighbor u of v: Process u, mark u, and then place u in the queue (since u may have further unprocessed neighbors).

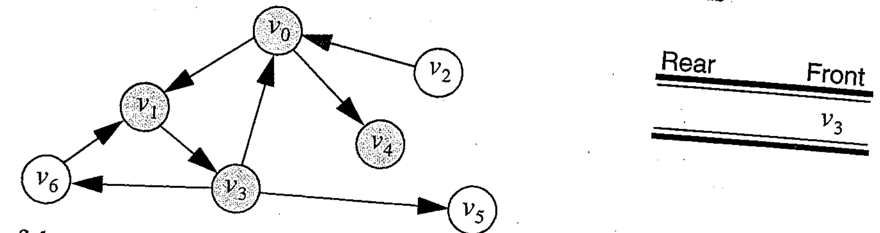
These two steps are repeated until the queue becomes empty. Let's look at our example to see how these two steps are carried out when vertex 0 is at the head of the queue. The vertex 0 is removed from the queue, and we note that it has two unprocessed neighbors: vertices 1 and 4. Vertices 1 and 4 will each be processed, marked, and placed in the queue. Let's assume that vertex 1 is placed in the queue first and then vertex 4. (The queuing could also occur the other way, with vertex 4 placed first; the algorithm is correct either way.) After 1 and 4 are in the queue, the situation looks like this:



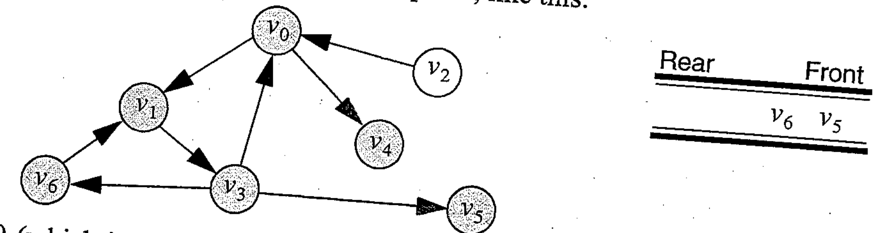
Since the queue still has entries, we repeat the two steps again: Remove the front entry (vertex 1), process and mark any unmarked neighbors of vertex 1, and enter these neighbors into the queue. The only unmarked neighbor of vertex 1 is vertex 3, so after processing, marking, and entering vertex 3, the situation looks like this:



Next we remove vertex 4 from the front of the queue. Since vertex 4 has no neighbors, no new entries are processed or placed in the queue. The situation is now:



Vertex 3 comes out of the queue next. It has two unmarked neighbors (vertices 5 and 6) that are processed, marked, and placed in the queue, like this:



Notice that vertex 0 (which is also a neighbor of vertex 3) does not get reprocessed because it is already marked.

At this point, we remove vertex 5 from the front of the queue, but it has no unmarked neighbors to worry about. We then remove vertex 6 from the queue. It also has no unmarked neighbors. The queue is finally empty, which ends the breadth-first search.

the effects of breadth-first and depth-first searches are similar—only the order of processing vertices differs

The effect of the breadth-first search is similar to the depth-first search: Vertices 0, 1, 3, 4, 5, and 6 have all been processed since they could all be reached from the starting point (vertex 0). Vertex 2 was not processed since there is no directed path from vertex 0 to vertex 2. However, the breadth-first search processes the vertices in a different order than the depth-first search. The breadth-first search first processed vertex 0, then processed the neighbors of the start point (1 and 4), then processed *their* neighbors (vertex 3), and so on. This contrasts with a depth-first search that processed the vertices 0, 1, 3, and 5 to start with (in that order).

You should now be able to carry out depth-first and breadth-first searches by hand on a directed graph. Next we will specify and implement the two searches as static methods that can print the labels of the vertices of a Graph.

Depth-First Search—Implementation

One way to print the labels of graph is with a depth-first search as specified in this static method:

♦ **depthFirstPrint**

`public static <E> void depthFirstPrint(Graph<E> g, int start)`

Static method to print the labels of a graph with a depth-first search.

Parameters:

g – a non-null Graph

start – a vertex number from the Graph g

Precondition:

start is non-negative and less than g.size( ).

Postcondition:

A depth-first search of g has been conducted, starting at the specified start vertex. Each vertex visited has its label printed using System.out.println. Note that vertices that are not connected to the start will not be visited.

Throws: NullPointerException

Indicates that g is null.

Throws: ArrayIndexOutOfBoundsException

Indicates that the start was not a valid vertex number.

Throws: OutOfMemoryError

Indicates that there is insufficient memory for an array of boolean values used by this method.

In this specification, the parameter g may be any Graph with labels that can be printed by System.out.println. During a search, each time a vertex v is reached, that vertex is "processed" by activating System.out.println with the label of v as the argument.

The implementation of depthFirstPrint uses an array of boolean values declared as a local variable of the method, shown here:

```
boolean[ ] marked = new boolean[g.size( )];
```

This array has one component for each possible vertex of the graph g, and its purpose is to keep track of which vertices have been marked as visited by the search. In general, for a vertex number v, the component marked[v] is true if v has already been visited by the search and is false otherwise. The complete pseudocode for depthFirstPrint is short, as shown next.

1. Allocate the marked array (which automatically has all of its components set to false).

2. Activate a separate static method to actually carry out the search.

pseudocode depth-first search

You may be wondering, "Why use a separate method in Step 2? Why can't we just carry out the search in the body of the depthFirstPrint method itself?" Good question. The answer is that we plan to use recursion: The start vertex is processed, and then recursive calls are made to process each of the start vertex's neighbors. Further recursive calls are made to process the neighbors' neighbors, and so on. So, if the work were carried out in the body of the depthFirstPrint method, each time a recursive call is made, we would do Step 1—clearing the marked array—and . . . oops! Clearing the marked array at every recursive call will definitely lead to trouble.

The new method, executed in Step 2, will be called depthFirstRecurse with this specification:

♦ **depthFirstRecurse**

```
public static <E> void depthFirstRecurse
   (Graph<E> g, int v, boolean[ ] marked)
```

Recursive method to carry out the work of depthFirstPrint.

Parameters:

g – a non-null Graph

v – a vertex number from the Graph g

marked – an array to indicate which vertices of g have already been visited

Precondition:

v is non-negative and less than g.size( ). marked.length is equal to g.size( ); for each vertex x of g, marked[x] is true if x has already been visited by this search; otherwise, marked[x] is false. The vertex v is an unmarked vertex at which the search has just arrived.

Postcondition:

The depth-first search of g has been continued through vertex v and beyond to all vertices that can be reached from v via a path of unmarked vertices. Each vertex visited has its label printed using System.out.println.

Throws: NullPointerException

Indicates that g is null.

Throws: ArrayIndexOutOfBoundsException

Indicates that v was not valid or marked was the wrong size.

Now let's examine the implementation of the depthFirstRecurse method. The first task is to mark and process the vertex v. After this, we will examine each of v's neighbors. Each time we find an unmarked neighbor, we will make a recursive call to continue the search through that neighbor and beyond.

unmarked neighbor, then there will be another recursive call at that level and so on until we reach a vertex with no unmarked neighbors. This description of `depthFirstRecurse` is implemented at the top of Figure 14.4, and the actual `depthFirstPrint` method is implemented at the bottom of the figure.

Breadth-First Search—Implementation

The breadth-first search is implemented with a queue of vertex numbers. The start vertex is processed, marked, and placed in the queue. Then the following steps are repeated until the queue is empty: (1) Remove a vertex, v, from the front of the queue, and (2) For each unmarked neighbor u of v: process u, mark u, and then place u in the queue (since u may have further unprocessed neighbors).

The implementation of this breadth-first search is Programming Project 1.

FIGURE 14.4 Depth-First Search

Implementations

```
public static <E> void depthFirstRecurse(Graph<E> g, int v, boolean[ ] marked)
{
   int[ ] connections = g.neighbors(v);
   int i;

   marked[v] = true;
   System.out.println(g.getLabel(v));

   // Traverse all the neighbors, looking for unmarked vertices:
   for (int nextNeighbor : connections)
   {
      if (!marked[nextNeighbor])
         depthFirstRecurse(g, nextNeighbor, marked);
   }
}

public static <E> void depthFirstPrint(Graph<E> g, int start)
{
   boolean[ ] marked = new boolean[g.size( )];

   depthFirstRecurse(g, start, marked);
}
```

Self-Test Exercises for Section 14.3

9. Do a depth-first search of the Australia graph (Figure 14.2 on page 707), starting at Sydney. List the order in which the cities are processed. Do the same for a breadth-first search.
10. Suppose you are doing a breadth-first search of a graph with n vertices. How large can the queue get?
11. What kind of search occurs if you replace breadth-first search's queue with a stack?

14.4 PATH ALGORITHMS

Determining Whether a Path Exists

Frequently, a problem is represented by a graph, and the answer to the problem can be found by answering some question about paths in the graph. For example, a network of computers can be represented by a graph, with each vertex representing one of the machines in the network and each edge representing a communication wire between two machines. The question of whether one machine can send a message to another machine boils down to whether the corresponding vertices are connected by a path.

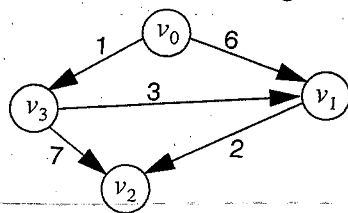
Either a breadth-first search or a depth-first search can be used to determine whether a path exists between two vertices, u and v. The idea is to use u as the start vertex of the search and proceed with a breadth-first or depth-first search. If the vertex v is ever visited, then the search can stop and announce that there is a path from u to v. On the other hand, if v is never visited, then there is no path from u to v.

Graphs with Weighted Edges

Often we need to know more than just *"Does a path exist?"* In the network example, each edge represents a communication wire between machines. Such a wire might have a "cost" associated with using the wire. The cost could be the amount of energy required to use the path, or perhaps the amount of time required for the wire to transmit a message, or even a dollars-and-cents cost required to use the wire to send one message. In any case, there could be many paths from one vertex to another, and we might want to find the path with the lowest total cost (that is, the path with the lowest possible sum of its edge costs).

This kind of question can be solved by using a graph in which each edge has a non-negative integer value attached to it, called the **weight** or **cost** of the edge.

Here's an example of a graph with edge weights:



the shortest path between two vertices is the path with lowest total cost

In this example, there are several paths from vertex v_0 to vertex v_2. The path with the lowest total cost traverses the edge from v_0 to v_3 (with a cost of 1), and then from v_3 to v_1 (with a cost of 3) and finally from v_1 to v_2 (with a cost of 2). The total cost of this path is $1 + 3 + 2$, which is 6. There is a path with fewer edges (such as the path from v_0 to v_1 to v_2), but no other path has a lower total cost. The path with the lowest total cost is called the **shortest path**. (If you think of the weights as distances, then the term *shortest path* will sound like a sensible term.) The problem of finding the shortest path between two vertices of a graph occurs often in computer science (such as the network example) and in applications (such as finding the shortest driving distance between two points on a road map).

Here is a summary of the concepts we have introduced:

> **Weighted Edges and Shortest Paths**
>
> A **weighted edge** is an edge together with a non-negative integer called the edge's **weight**.
>
> The **weight of a path** is the total sum of the weights of all the edges in the path. (Note: The weight of the empty path, with no edges, is always zero.)
>
> If two vertices are connected by at least one path, then we can define the **shortest path** between two vertices. This is the path that has the smallest weight. (There may be several paths with equally small weights, in which case each of the paths is called "smallest.")

Shortest Distance Algorithm

In this section, we will present an efficient algorithm called *Dijkstra's algorithm* (named for computer scientist Edsgar Dijkstra, who proposed the algorithm) for finding the shortest path between two vertices. Throughout the section, we will use graphs with weighted edges where each weight is a non-negative integer. We will use pseudocode rather than a particular language such as Java.

We start with a problem that's simpler than actually finding the shortest path from one vertex to another. We'll concentrate on simply finding the *weight* of the shortest path—in other words the smallest possible sum of edge weights along a path from one vertex to another. This weight is called the *shortest distance*.

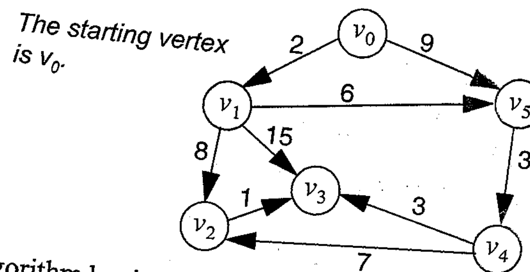
we'll actua shortest distances than short paths

Dijkstra's algorithm actually provides more information than just the shortest distance from one vertex to another. In fact, the algorithm provides the shortest distance from a starting vertex (which we call `start`) to *every* vertex in the graph. The algorithm uses an integer array called `distance` with one component for each vertex in the graph. Here is the algorithm's goal:

> **Goal of the Shortest Distance Algorithm**
>
> The goal is to completely fill the distance array so that for each vertex v, the value of `distance[v]` is the weight of the shortest path from `start` to v.

We'll illustrate how the algorithm works with this small graph:

The starting vertex is v_0.



The algorithm begins by filling in one value in the distance array. We fill in zero for the component that is indexed by the `start` vertex itself, indicating that the weight of the shortest path from the `start` vertex to the `start` vertex itself is zero. This is correct since the empty path exists from the `start` vertex to itself. At this point, the distance array has one known value:

distance	[0]	[1]	[2]	[3]	[4]	[5]
	0	?	?	?	?	?

At this point, we have one correct value, `distance[0]`. In the other locations we will write a value based on what we know so far. Because we don't know too much, our initial values won't be too accurate, but that's okay. In fact, the values we fill in will all be *infinity*, represented by the symbol ∞. The distance array with mostly ∞ is shown at the top of the next page.

distance	0	∞	∞	∞	∞	∞
	[0]	[1]	[2]	[3]	[4]	[5]

In an actual implementation, we would use some special integer value for ∞. For example, we could use –1 and make sure all the rest of our programming always treats an occurrence of –1 as if it were infinity.

Now we are ready to do some processing that will steadily improve the values in the distance array. We begin with an observation about the initial values that we've placed in the distance array: These initial values are actually correct, *if we are permitting only the empty path and ignoring all other paths*. In other words, if the empty path is the only path we are permitting, then there is a path from the start vertex to itself (namely the empty path with weight zero). But there is no way to get from the start vertex to any other vertex, and the fact that there is no path is represented by ∞ in the distance array.

the algorithm gradually improves the values in the distance array

Of course, permitting *only* the empty path is an overwhelming restriction. The key to the algorithm is in gradually relaxing this restriction, allowing more vertices to appear in permitted paths. As more and more vertices are allowed, we will continually revise the distance array so its values are correct *for paths that pass through only allowed vertices*. By the end of the algorithm, all vertices are allowed and the distance array has entirely correct values.

The idea we have described needs some refinement. How are newly allowed vertices selected? How do we keep track of which vertices are currently allowed? How is the distance array revised to permit the newly allowed vertices to appear in a path? These questions are addressed in the following list of three steps for the complete algorithm.

Step 1. Fill in the distance array with ∞ at every location with the exception of `distance[Start]`, which is assigned the value zero.

Step 2. Initialize a *set* of vertices, called `allowedVertices`, to be the empty set. Throughout the algorithm, a *permitted path* is a path that starts at the start vertex and in which each vertex on the path (except perhaps the final vertex) is in the set of allowed vertices. The final vertex on a permitted path does not need to be in the `allowedVertices` set. At this point, `allowedVertices` is the empty set, so the only permitted path is the empty path without any edges. (This empty path does contain one vertex, the start vertex. But since this vertex is the *final* vertex on the path, the vertex is not required to be in the allowed vertices set.)

Step 3. The third step is a loop. Each time through the loop we will add one more vertex to `allowedVertices` and then update the distance array so all the allowed vertices may appear on paths. Here's a brief summary of the loop:


```
// Loop in Step 3 of the shortest distance algorithm:
// n is the number of vertices in the graph

for (allowedSize = 1; allowedSize <= n; allowedSize++)
{
   // At this point, allowedVertices contains allowedSize - 1 vertices,
   // which are the allowedSize - 1 closest vertices to the start vertex.
   // Also, for each vertex v, distance[v] is the shortest distance from
   // the start vertex to vertex v, provided we are
   // considering only permitted paths (i.e., paths where
   // each vertex except the final vertex must be in
   // allowedVertices).

   Step 3a. Let next be the closest vertex to the start vertex that is not
   yet in the set of allowed vertices. (If several vertices are equally close,
   then you may choose next to be any of them.)

   Step 3b. Add the vertex next to the set allowedVertices.

   Step 3c. Revise the distance array so the new vertex (next) may
   appear on permitted paths.
}
```

The loop's computation hinges on the condition written just before Step 3a. The condition indicates that the `allowedVertices` set actually contains the `allowedSize-1` vertices that are *closest* to the start vertex. The condition also indicates that `distance[v]` is always the shortest distance from the start vertex to vertex `v`, provided we are considering only permitted paths (that is, paths where all vertices except the final vertex must be in the set of allowed vertices). This condition is `true` the first time the loop is entered, and it is also `true` at the start of each subsequent iteration. The responsibility of the three steps—3a, 3b, and 3c—is to ensure the condition remains valid at the start of each iteration. Let's examine the three steps in some detail.

Step 3a. This step must determine which of the *unallowed* vertices is closest to the start vertex. There is a simple rule for choosing this vertex:

> **How to Choose the Next Vertex in Step 3a**
>
> In Step 3a, we will always choose the unallowed vertex that has the *smallest* current value in the distance array. (If several vertices have equally small distances, then we may choose any of them.)

For example, suppose we reach Step 3a and have this situation:

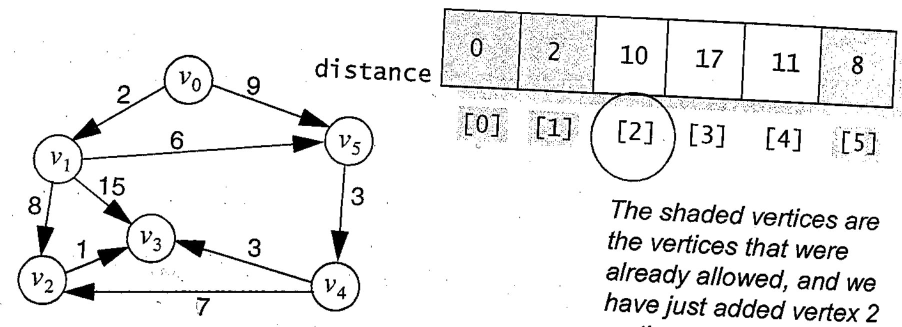
	[0]	[1]	[2]	[3]	[4]	[5]
distance	0	2	10	17	∞	8

In this example, the two shaded vertices, 0 and 1, are already in the set of allowed vertices.

Since vertices 0 and 1 are already in the allowed set, we cannot choose them as the next vertex. Among the other vertices, vertex 5 has the smallest current value in the distance array (the value is 8). So we would choose vertex 5 as the next vertex. In the answer to Self-Test Exercise 12, we will explain precisely *why* this rule works, but for now it is sufficient to know this is the correct way to select the next vertex.

Step 3b. In this step, we "add the vertex `next` to the set `allowedVertices`." The implementation of this step depends on how the set of allowed vertices is represented. One possibility is to implement `allowedVertices` as a set of vertex numbers, using an ADT for a set of integers. In this case, Step 3b merely inserts `next` into the set.

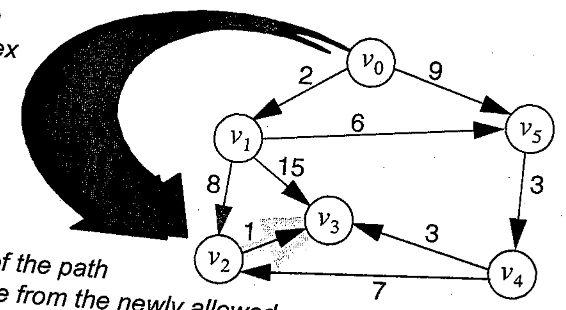
Step 3c. Finally, we must revise the distance array so the newly allowed vertex, `next`, is permitted on a path from the start vertex to another vertex. An example will explain the necessary revisions. Suppose vertices 0, 1, and 5 are already in the `allowedVertices` set, and we have just added vertex 2 as our next vertex, as shown here:



The shaded vertices are the vertices that were already allowed, and we have just added vertex 2 as the next allowed vertex.

Since we have just added vertex 2 as a newly allowed vertex, we must now update the distance array to reflect the fact that vertex 2 may now appear on paths.

For example, `distance[3]` is currently 17. This means that there is a path from the start vertex to vertex 3 that uses only vertices 0, 1, and 5 and that has a length of 17. Here's the key question: *If we also allow vertex 2 to appear on a path from the start vertex to vertex 3, can we obtain a distance that is smaller than 17?* A smaller distance might be possible by taking a two-part path, shown here:



The first part of the path goes from the start vertex v_0 to the newly allowed vertex v_2 . . .

. . . and the second part of the path consists of one more edge from the newly allowed vertex v_2 to vertex v_3.

This path has two parts: the part from the start vertex to vertex 2 and the part that has the single edge from vertex 2 to vertex 3. The total weight of this path is:

`distance[2]` + (weight of the edge from vertex 2 to vertex 3)

In our example, this sum is 11, which is smaller than the current "best distance to vertex 3." Therefore, we should replace `distance[3]` with this smaller sum. We must also create similar two-part paths for each of the other unallowed vertices, and if the two-part path is smaller than the current distance, then we modify the distance array. This provides the following refined pseudocode for Step 3c:

Step 3c (revised). Revise the distance array so that the new vertex (`next`) may appear on permitted paths. The integer `n` is the number of vertices; `v` and `sum` are local integer variables, as shown in this code:

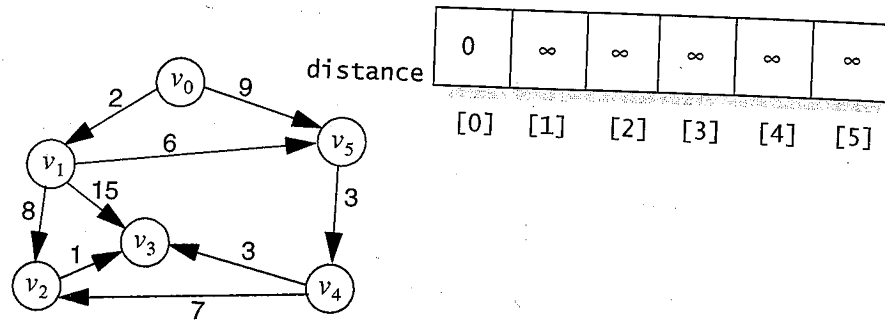
```
for (v = 0; v < n; v++)
    if ((v is not an allowed vertex) and (there is an edge from next to v))
    {
        sum = distance[next] + (weight of the edge from next to v);
        if (sum < distance[v])
            distance[v] = sum;
    }
```

Notice that we do not consider a possible new, smaller distance to vertices that are already allowed vertices. That is because these vertices are all closer to the start vertex than `next` is, so `distance[next]` is going to be larger than the shortest distance to any of these vertices.

... that a longer path from next to the end vertex would have to pass through other allowed vertices, meaning that we have a path that goes from the start vertex, through `next`, through *another* allowed vertex, and finally to the end vertex. But such a path will always be shorter by avoiding the `next` vertex altogether—just go from the start vertex to that other allowed vertex (using the shortest path) and then to the end vertex. Such a path, which does not go through the `next` vertex, is already permitted. We don't need to consider such a path again—we need only consider new paths that pass through the `next` newly allowed vertex.

The complete pseudocode for the algorithm is shown in Figure 14.5. The main loop actually stops at the number of vertices minus 1 because at that point, there is only one unallowed vertex, and this must be the farthest vertex from the start vertex so that no shortest paths can go through this farthest vertex.

We'll execute the algorithm on our example graph. Here's the situation after initializing the distance array:



At this point, the set of allowed vertices is empty, and we will enter the main loop for the first time. In Step 3a, the value of `next` is set to vertex 0 (since `distance[0]` is the smallest value in the distance array). We then look at each unallowed vertex `v` with an edge from vertex 0 to vertex `v`. These are vertices 1 and 5, so we check to see whether we need to revise `distance[1]` and `distance[5]`:

- `distance[0]` + (the weight of the edge from 0 to 1) is 2. Since this is smaller than the current value of `distance[1]`, we replace `distance[1]` with 2.

- `distance[0]` + (the weight of the edge from 0 to 5) is 9. Since this is smaller than the current value of `distance[5]`, we replace `distance[5]` with 9.

Pseudocode

Input. A directed graph with positive, integer edge weights and `n` vertices. One of the ve... called `start`, is specified as the start vertex.

Output. A list of the shortest distances from the start vertex to every other vertex in the gr...

The algorithm uses an array of `n` integers (called `distance`) and a set of vertices (... `allowedVertices`). The variables `v`, `allowedSize`, and `sum` are local integer variables. Th... some special value (∞) that we can place in the distance array to indicate an infinite dis... (which means there is no path).

Step 1. Initialize the distance array to contain all ∞ except `distance[start]`, which is set to ...

Step 2. Initialize the set of allowed vertices to be the empty set.

Step 3. Compute the complete distance array:

```
for (allowedSize = 1; allowedSize < n; allowedSize++)
{
    // At this point, allowedVertices contains allowedSize - 1 vertices, which are the
    // allowedSize - 1 closest vertices to the start vertex. Also, for each vertex v, distance
    // is the shortest distance from the start vertex to vertex v, provided that we are
    // considering only permitted paths (i.e., paths where each vertex except the final ver
    // must be in allowedVertices).
```

Step 3a. Let `next` be the closest vertex to the start vertex, which is not yet in the set of allowed vertices. (If several vertices are equally close, then you may choose `next` to be a... of them.)

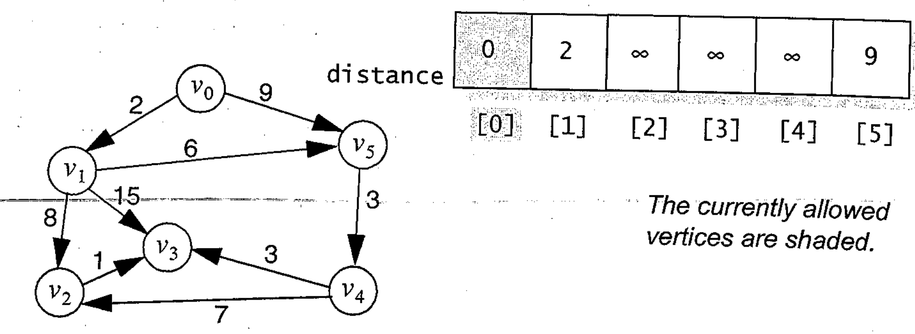
Step 3b. Add the vertex `next` to the set `allowedVertices`.

Step 3c. Revise the distance array so that the new vertex (`next`) may appear on permitte... paths:

```
for (v = 0; v < n; v++)
    if ((v is not an allowed vertex) and (there is an edge from next to v))
    {
        sum = distance[next] + (weight of the edge from next to v);
        if (sum < distance[v])
            distance[v] = sum;
    }
}
```

Step 4. Output the values in the distance array. (Each `distance[v]` is the shortest distance fro... the start vertex to vertex `v`.)

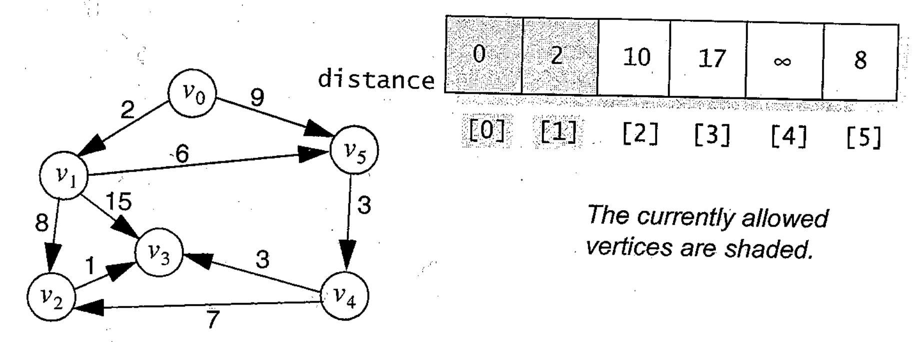
shaded):



The second time we enter the main loop, the value of `next` is set to vertex 1 (since `distance[1]` is the smallest value of the unallowed vertices). We then look at each unallowed vertex `v` with an edge from vertex 1 to vertex `v`. These are vertices 2, 3, and 5, so we check to see whether we need to revise `distance[2]`, `distance[3]`, and `distance[5]`:

- `distance[1]` + (the weight of the edge from 1 to 2) is 10. Since this is smaller than the current value of `distance[2]`, we replace `distance[2]` with 10.
- `distance[1]` + (the weight of the edge from 1 to 3) is 17. Since this is smaller than the current value of `distance[3]`, we replace `distance[3]` with 17.
- `distance[1]` + (the weight of the edge from 0 to 5) is 8. Since this is smaller than the current value of `distance[5]`, we replace `distance[5]` with 8.

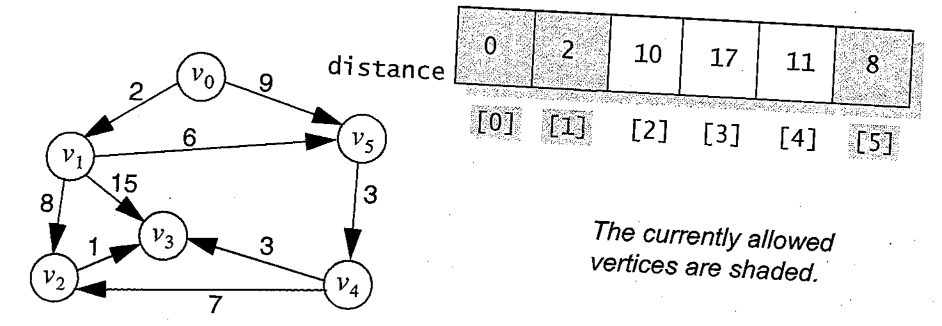
At this point, the distance array is:



The third time we enter the main loop, the value of `next` will be set to vertex 5 (since `distance[5]` is the smallest value of the unallowed vertices). We then look at each unallowed vertex `v` with an edge from vertex 5 to vertex `v`. Vertex 4 is the only vertex that is the target of an edge from vertex 5, so we check to see whether we need to revise `distance[4]`:

- `distance[5]` + (the weight of the edge from 5 to 4) is 11. Since this is smaller than the current value of `distance[4]`, we replace `distance[4]` with 11.

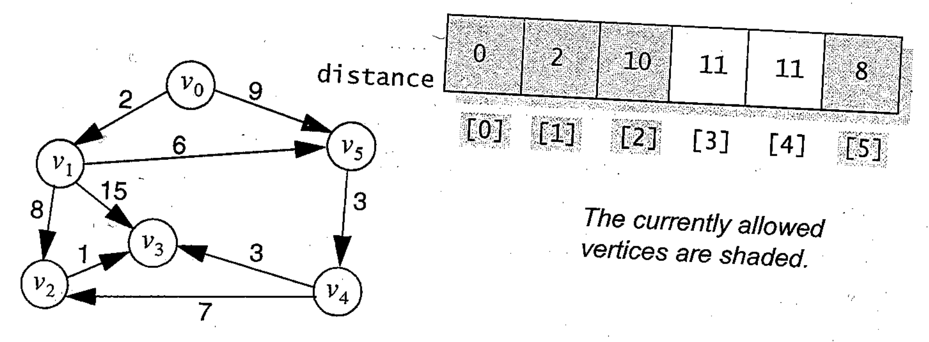
At this point, we have this situation:



The fourth time we enter the main loop, the value of `next` will be set to vertex 2 (since `distance[2]` is the smallest value of the unallowed vertices). We then look at each unallowed vertex `v` with an edge from vertex 2 to vertex `v`. Vertex 3 is the only such vertex, so we check to see whether we need to revise `distance[3]`:

- `distance[2]` + (the weight of the edge from 2 to 3) is 11. Since this is smaller than the current value of `distance[3]`, we replace `distance[3]` with 11.

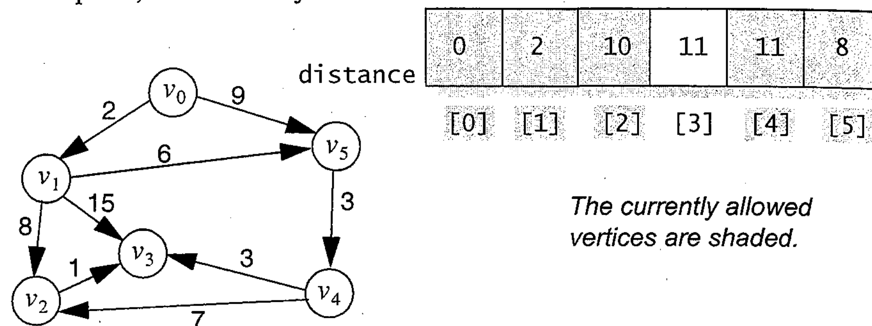
At this point, the situation is:



The fifth time we enter the main loop, the value of next will be set to either vertex 3 or vertex 4 (since both distance[3] and distance[4] are 11). It doesn't matter which one we choose, so let's choose vertex 4. We then look at each unallowed vertex v with an edge from vertex 4 to vertex v. This is only vertex 3, so we check to see whether we need to revise distance[4]:

- distance[4] + (the weight of the edge from 4 to 3) is 14. Since this is *larger* than the current value of distance[3], we do not replace distance[3].

At this point, we are nearly done:



The currently allowed vertices are shaded.

The main loop of the algorithm now stops. We don't need to process that last unallowed vertex (3) since it is the farthest vertex from the start vertex. For each vertex v, the value of distance[v] is the shortest distance from the start vertex (vertex 0) to v.

Shortest Path Algorithm

We have shown how to compute the *weight* of the shortest path from a start vertex to each other vertex in a graph. But how can we compute the actual sequence of vertices that occurs along the shortest path? It turns out that Dijkstra's shortest distance algorithm contains enough information to actually print the shortest path from the start vertex to any other vertex, provided that we keep track of one extra piece of information:

> **Predecessor Information for Shortest Paths**
>
> For each vertex v, we will keep track of which vertex was the next vertex when distance[v] was given a new, smaller value. We will keep track of these values in an array called predecessor so that for each vertex v, predecessor[v] is the value of next at the time when distance[v] was given a new, smaller value. (Note: predecessor[start] does not need to have a value since distance[start] is never updated.)

The predecessor information is easy to keep track of. Each time we update distance[v] with the assignment distance[v] = sum, we must also update predecessor[v] with this assignment:

```
predecessor[v] = next;
```

When the algorithm finishes, the value of distance[v] is the weight of the shortest path from the start vertex to vertex v. Or distance[v] might be infinity, indicating that there is no path from the start vertex to vertex v. But when distance[v] is not infinity, we can actually print out the vertices on the shortest path from the start vertex to vertex v, using the following code:

```
// Printing the vertices on the shortest path from the start vertex to v:
// Vertices are printed in reverse order, starting at v, and going to start.

vertexOnPath = v; // The last vertex on the path.
System.out.println(vertexOnPath); // Print the final vertex.
while (vertexOnPath != start)
{
    vertexOnPath = predecessor[vertexOnPath];
    System.out.println(vertexOnPath);
}
```

In other words, the last vertex on the path to v is vertex v itself. The next-to-last vertex is predecessor[v], and the vertex before that is obtained by applying predecessor to the next-to-last vertex—and so on, right back to the start vertex. As indicated, this algorithm manages to print the vertices in reverse order, from vertex v back to the start vertex.

Self-Test Exercises for Section 14.4

12. When we select the next vertex in the shortest distance algorithm, we need to select the unallowed vertex that is closest to the start vertex. How is this selection done?
13. Consider the graph we have been using throughout this section, except change the weight of the edge that goes from 0 to 5. Its new weight is 4. Go through the entire algorithm to compute the new distance array.
14. Compute the complete predecessor array for the previous exercise and use it to find the actual shortest path from vertex 0 to vertex 3.

CHAPTER SUMMARY

- Graphs are a flexible data structure with many occurrences in computer science and in applications. Many problems can be solved by asking an appropriate question about paths in a graph.
- There are several different kinds of graphs: undirected graphs (in which edges have no particular orientation), directed graphs (in which each edge goes from a source vertex to a target vertex), graphs with loops (i.e., an edge connecting a vertex to itself), graphs with multiple edges (i.e., more than one edge can connect the same pair of vertices), labeled graphs (in which each vertex has an associated label), and graphs with weighted edges (in which each edge has an associated number, called its weight).
- There are two common ways to implement a graph: an adjacency matrix or edge lists. The different implementations have different time performance for common operations such as determining whether two vertices are connected.
- There are two common ways to traverse a graph: depth-first search and breadth-first search.
- Dijkstra's algorithm provides an efficient way to determine the shortest path from a given start vertex to every other vertex in a graph with weighted edges.

Solutions to Self-Test Exercises

1. The airline route graph has 9 vertices, 14 edges, and no loops. A loop would be an excursion flight that takes off and lands at the same city.

2. It is simple (no loops, no multiple edges).

3. Your state graph should have 16 vertices and 48 directed edges. (There are 32 edges from Rule 1, and 8 edges each from Rules 2 and 3.) It is possible to go from the start state to the goal state in four moves.

4. For the removal method, suppose that k is the number of vertices that are being removed and that this will leave `m` vertices in the new, smaller graph. The method should allocate new, smaller arrays for the edges and labels (and these new arrays can be referred to by local variables of the method). Then copy the information for the first `m` vertices into these smaller arrays.
   The method that adds new vertices must allocate new, larger arrays and copy all the information from the old arrays to the new arrays.
   In both methods, the final step is to make the instance variables (`edges` and `labels`) refer to the new arrays.

5. Suppose the vertex numbers of the two vertices are `i` and `j`. The method must interchange their labels and also interchange the rows and columns of the edges matrix. If you aren't careful, you might accidentally interchange `edges[i][i]` with `edges[j][j]` two times (once when you are interchanging the rows and once when you are interchanging the columns.)

6. The adjacency matrix, `a`, could be a two-dimensional array of integers with `a[i][j]` storing the number of edges from vertex `i` to vertex `j`.

7. This is an open-ended question, but your answer should consider space requirements for each representation and time requirements for common operations.

8. Here is one solution:

```
public static <E> int maxEdges
(Graph<E> g)
{
   int i;
   int answer;
   int manyEdges;
   int maxEdges = 0;
   int[ ] ni;

   if (g.size( ) == 0)
      throw new
      IllegalArgumentException
      ("Graph is empty");

   for (i = 0; i < g.size( ); i++)
   {
      ni = g.neighbors(i);
      manyEdges = ni.length;
      if (manyEdges >= maxEdges)
      {
         answer = i;
         maxEdges = manyEdges;
      }
   }

   return answer;
}
```

9. Traversals sometimes make choices a which vertex to visit next. When we have a choice, we will visit the vertex that is al betically first, giving these two orders: *De first:* Sydney, Canberra, Adelaide, Melbou Hobart, Perth, Black Stump, Darwin, B bane. *Breadth-first:* Sydney, Canberra, M bourne, Adelaide, Brisbane, Hobart, Pe Black Stump, Darwin.

10. If n is 1, then the queue needs room for o one vertex. If n is more than 1, then the qu will never have more than $n-1$ entries. Her why: The start vertex is only in the queue o (by itself), and then we remove the start ve from the queue. Each other vertex is place the queue at most once, so the largest qu needed would occur if all $n-1$ vertices neighbors of the start vertex.

11. A depth-first search.

12. From among the unallowed vertices, we sel the vertex with the smallest current value the distance array. This works because the c rent distance array contains the correct d tance values if we are only permitting allow vertices, and the currently allowed vertices a the `n-1` closest vertices. Therefore, the sho est path to the nth closest vertex must pa through only currently allowed vertices, a therefore the distance array contains the co rect value for that n^{th} closest vertex.

13. The new final distance array: (0, 2, 10, 10, 4).

14. The new shortest path to vertex 3: vertex 0 vertex 5, vertex 5 to vertex 4, vertex 4 vertex 3.