Chapter 10

Graphs

Chapter Objectives

- ◆ To become familiar with graph terminology and the different types of graphs
- ◆ To study a Graph ADT (abstract data type) and different implementations of the Graph ADT
- ◆ To learn the breadth-first and depth-first search traversal algorithms
- ◆ To learn some algorithms involving weighted graphs
- ◆ To study some applications of graphs and graph algorithms

One of the limitations of trees is that they cannot represent information structures in which a data item has more than one parent. In this chapter, we introduce a data structure known as a *graph* that will allow us to overcome this limitation.

Graphs and graph algorithms were being studied long before computers were invented. The advent of the computer made the application of graph algorithms to real-world problems possible. Graphs are especially useful in analyzing networks. Thus, it is not surprising that much of modern graph theory and application was developed at Bell Laboratories, which needed to analyze the very large communications network that is the telephone system. Graph algorithms are also incorporated into the software that makes the Internet function. You can also use graphs to describe a road map, airline routes, or course prerequisites. Computer chip designers use graph algorithms to determine the optimal placement of components on a silicon chip.

You will learn how to represent a graph, determine the shortest path through a graph, and find the minimum subset of a graph.

Graphs

10.1 Graph Terminology

A graph is a data structure that consists of a set of *vertices* (or nodes) and a set of *edges* (relations) between the pairs of vertices. The edges represent paths or connections between the vertices. Both the set of vertices and the set of edges must be finite, and either set may be empty. If the set of vertices is empty, naturally the set of edges must also be empty. We restrict our discussion to simple graphs in which there is at most one edge from a given vertex to another vertex.

EXAMPLE 10.1 The following set of vertices, V, and set of edges, E, define a graph that has five vertices, with labels A through E, and four edges.

$V = \{A, B, C, D, E\}$

$E = \{\{A, B\}, \{A, D\}, \{C, E\}, \{D, E\}\}$

Each edge is a set of two vertices. There is an edge between A and B (the edge {A, B}), between A and D, between C and E, and between D and E. If there is an edge between any pair of vertices x, y, this means there is a path from vertex x to vertex y and vice versa. We discuss the significance of this shortly.

Visual Representation of Graphs

Visually we represent vertices as points or labeled circles and the edges as lines joining the vertices. Figure 10.1 shows the graph from Example 10.1.

There are many ways to draw any given graph. The physical layout of the vertices, and even their labeling, are not relevant. Figure 10.2 shows two ways to draw the same graph.

FIGURE 10.1 Graph Given in Example 10.1

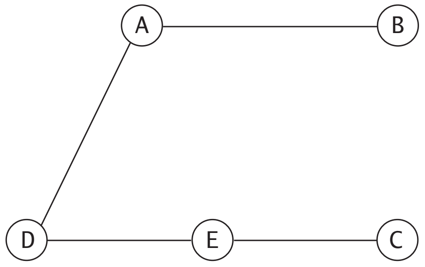
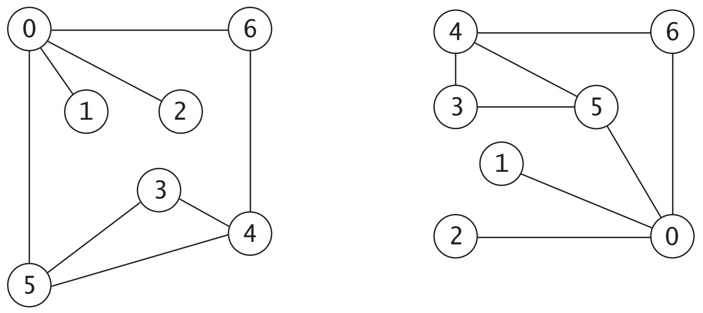


FIGURE 10.2
Two Representations of the Same Graph



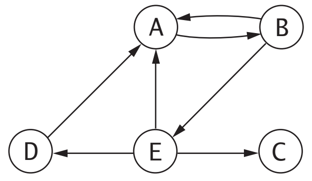
Directed and Undirected Graphs

The edges of a graph are *directed* if the existence of an edge from A to B does not necessarily guarantee that there is a path in both directions. A graph that contains directed edges is known as a *directed graph* or *digraph*, and a graph that contains undirected edges is known as an *undirected graph* or simply a graph. A directed edge is like a one-way street; you can travel on it in only one direction. Directed edges are represented as lines with an arrow on one end, whereas undirected edges are represented as single lines. The graph in Figure 10.1 is undirected; Figure 10.3 shows a directed graph. The set of edges for the directed graph follows:

$E = \{(\text{A, B}), (\text{B, A}), (\text{B, E}), (\text{D, A}), (\text{E, A}), (\text{E, C}), (\text{E, D})\}$

Each edge above is an ordered pair of vertices instead of a set as in an undirected graph. The edge (A, B) means there is a path from A to B. Observe that there is a path from both A to B and from B to A, but these are the only two vertices in which there is an edge in both directions. Our convention will be to denote an edge for a directed graph as an ordered pair (u, v) where this notation means that v (the destination) is adjacent to u (the source). We denote an edge in an undirected graph as the set $\{u, v\}$, which means that u is adjacent to v and v is adjacent to u. Therefore, you can create a directed graph that is equivalent to an undirected graph by substituting for each edge $\{u, v\}$ the ordered pairs (u, v) and (v, u). In general, when we describe graph algorithms in this chapter, we will use the ordered pair notation (u, v) for an edge.

FIGURE 10.3
Example of a Directed Graph



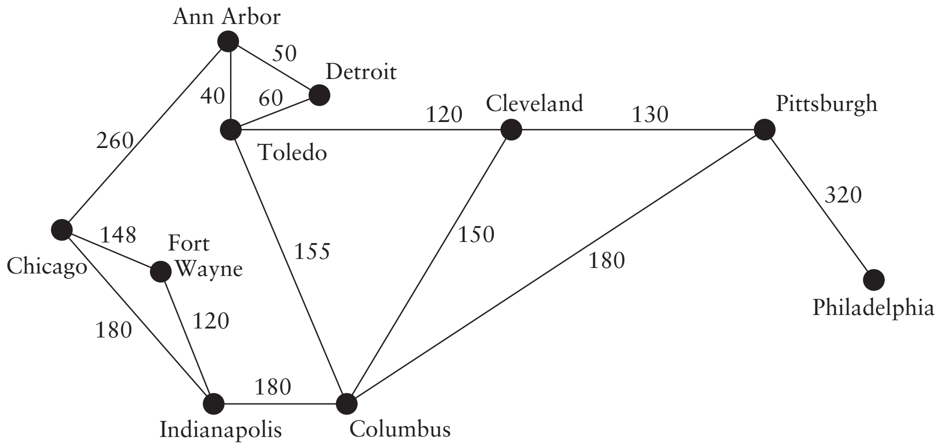
The edges in a graph may have values associated with them known as their *weights*. A graph with weighted edges is known as a *weighted graph*. In an illustration of a weighted graph, the weights are shown next to the edges. Figure 10.4 shows an example of a weighted graph. Each weight is the distance between the two cities (vertices) connected by the edge. Generally, the weights are nonnegative, but there are graph problems and graph algorithms that deal with negative weighted edges.

Paths and Cycles

One reason we study graphs is to find pathways between vertices. We use the following definitions to describe pathways between vertices.

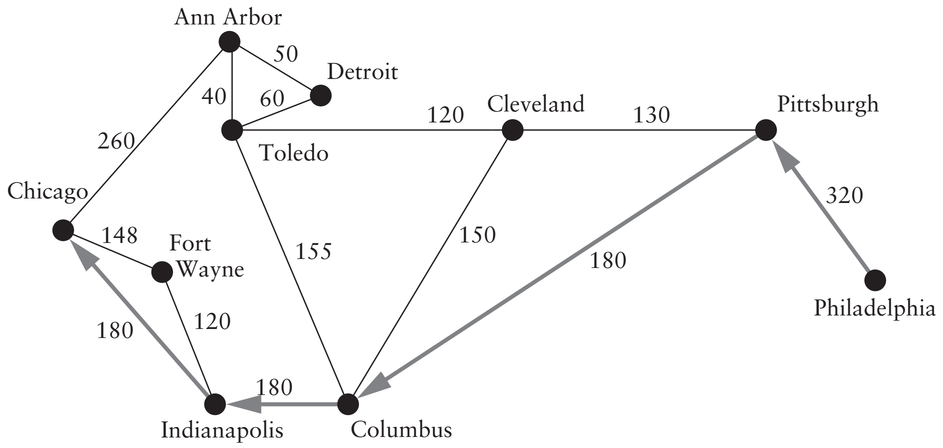
- A vertex is *adjacent* to another vertex if there is an edge to it from that other vertex. In Figure 10.4, Philadelphia is adjacent to Pittsburgh. In Figure 10.3, A is adjacent to D, but since this is a directed graph, D is not adjacent to A.

FIGURE 10.4
Example of a Weighted Graph



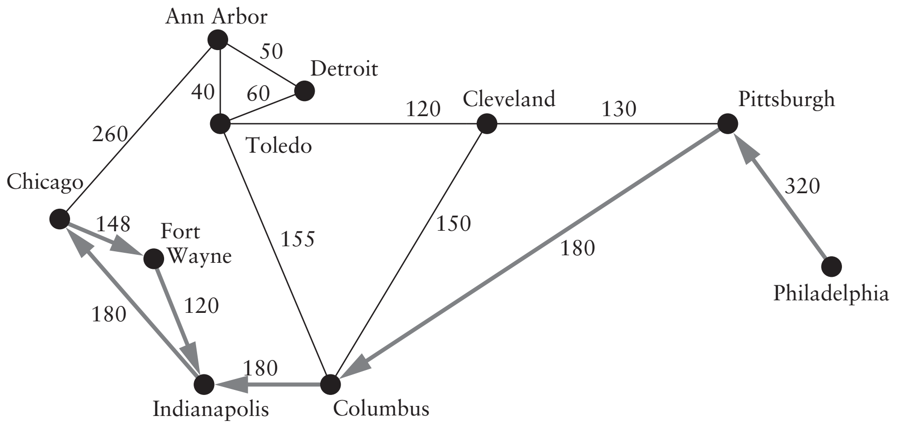
- A *path* is a sequence of vertices in which each successive vertex is adjacent to its predecessor. In Figure 10.5, the following sequence of vertices is a path: Philadelphia → Pittsburgh → Columbus → Indianapolis → Chicago.

FIGURE 10.5
A Simple Path



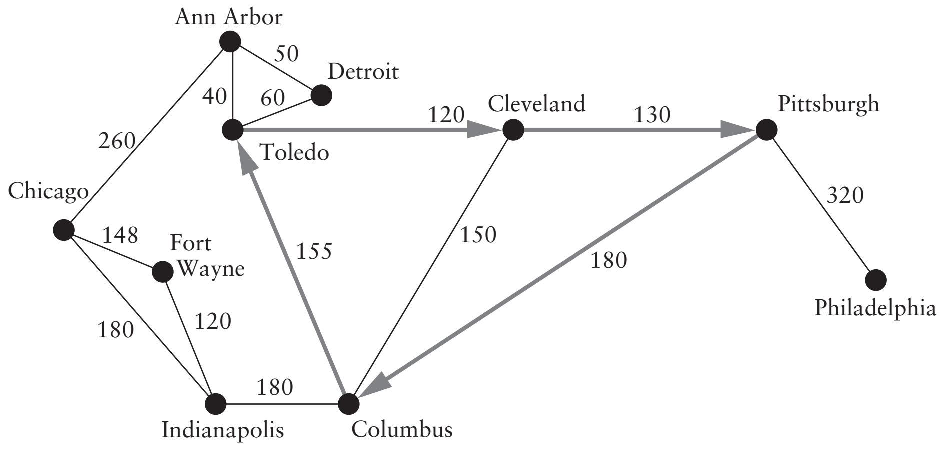
- In a *simple path*, the vertices and edges are distinct, except that the first and last vertices may be the same. In Figure 10.5, the path Philadelphia → Pittsburgh → Columbus → Indianapolis → Chicago is a simple path. The path Philadelphia → Pittsburgh → Columbus → Indianapolis → Chicago → Fort Wayne → Indianapolis is a path but not a simple path (see Figure 10.6).

FIGURE 10.6
Not a Simple Path



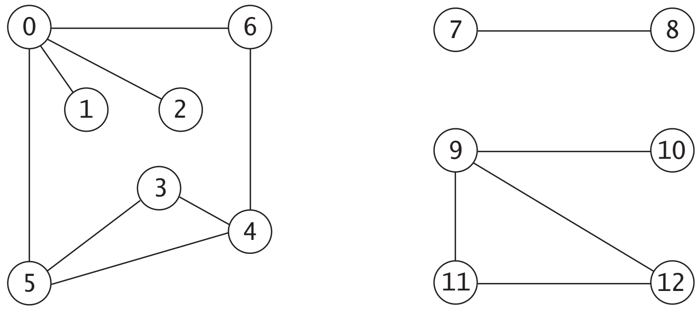
- A *cycle* is a simple path in which only the first and final vertices are the same. In Figure 10.7, the path Pittsburgh → Columbus → Toledo → Cleveland → Pittsburgh is a cycle. For an undirected graph, a cycle must contain at least three distinct vertices. Thus, Pittsburgh → Columbus → Pittsburgh is not considered a cycle.

FIGURE 10.7
A Cycle



- An undirected graph is called a *connected graph* if there is a path from every vertex to every other vertex. Figure 10.7 is a connected graph, whereas Figure 10.8 is not.

FIGURE 10.8
Example of an Unconnected Graph



- If a graph is not connected, it is considered *unconnected*, but it will still consist of *connected components*. A connected component is a subset of the vertices and the edges connected to those vertices in which there is a path between every pair of vertices in the component. A single vertex with no edges is also considered a connected component. Figure 10.8 consists of the connected components {0, 1, 2, 3, 4, 5, 6}, {7, 8}, and {9, 10, 11, 12}.

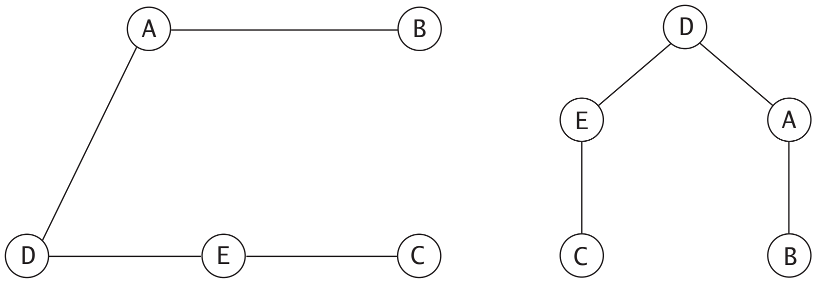
Relationship between Graphs and Trees

The graph is the most general of the data structures we have studied. It allows for any conceivable relationship among the data elements (the vertices). A tree is actually a special case of a graph. Any graph that is connected and contains no cycles can be viewed as a tree by picking one of its vertices (nodes) as the root. For example, the graph shown in Figure 10.1 can be viewed as a tree if we consider the node labeled D to be the root (see Figure 10.9).

Graph Applications

We can use graphs to help solve a number of different kinds of problems. For example, we might want to know whether there is a connection from one node in a network to all others. If we can show that the graph is connected, then a path must exist from one node to every other node.

FIGURE 10.9
A Graph Viewed as a Tree



In college you must take some courses before you take others. These are called prerequisites. Some courses have multiple prerequisites, and some prerequisites have prerequisites of their own. It can be quite confusing. You may even feel that there is a loop in the maze of prerequisites and that it is impossible to schedule your classes to meet the prerequisites. We can represent the set of prerequisites by a directed graph. If the graph has no cycles, then we can find a solution. We can also find the cycles.

Another application would be finding the least-cost path or shortest path from each vertex to all other vertices in a weighted graph. For example, in Figure 10.4, we might want to find the shortest path from Philadelphia to Chicago. Or we might want to create a table showing the distance (miles in the shortest route) between each pair of cities.

EXERCISES FOR SECTION 10.1

SELF-CHECK

1. In the graph shown in Figure 10.1, what vertices are adjacent to D? Also check in Figure 10.3.
2. In Figure 10.3, is it possible to get from A to all other vertices? How about from C?
3. In Figure 10.4, what is the shortest path from Philadelphia to Chicago?

10.2 The Graph ADT and Edge Class

Java does not provide a Graph ADT, so we have the freedom to design our own. To write programs for the applications mentioned at the end of the previous section, we need to be able to navigate through a graph or traverse it (visit all its vertices). To accomplish this, we need to be able to advance from one vertex in a graph to all its adjacent vertices. Therefore, we need to be able to do the following:

1. Create a graph with the specified number of vertices.
2. Iterate through all of the vertices in the graph.
3. Iterate through the vertices that are adjacent to a specified vertex.
4. Determine whether an edge exists between two vertices.
5. Determine the weight of an edge between two vertices.
6. Insert an edge into the graph.

With the exception of item 1, we can specify these requirements in a Java interface. Since a Java interface cannot include a constructor, the requirements for item 1 can only be specified in the comment at the beginning of the interface.

Listing 10.1 gives the declaration of the `Graph` interface.