8 Modeling Using Graph Theory

Introduction

The manager of a recreational softball team has 15 players on her roster: Al, Bo, Che, Doug, Ella, Fay, Gene, Hal, Ian, John, Kit, Leo, Moe, Ned, and Paul. She has to pick a starting team, which consists of 11 players to fill 11 positions: pitcher (1), catcher (2), first base (3), second base (4), third base (5), shortstop (6), left field (7), left center (8), right center (9), right field (10) and additional hitter (11). Table 8.1 summarizes the positions each player can play.

Table 8.1 Positions players can play

Al	Bo	Che	Doug	Ella	Fay	Gene	Hal	Ian	John	Kit	Leo	Moe	Ned	Paul
2, 8	1, 5, 7	2, 3	1, 4, 5, 6, 7	3, 8	10, 11	3, 8, 11	2, 4, 9	8, 9, 10	1, 5, 6, 7	8, 9	3, 9, 11	1, 4, 6, 7		9, 10

© Cengage Learning

Can you find an assignment where all of the 11 starters are in a position they can play? If so, is it the only possible assignment? Can you determine the ‘‘best" assignment? Suppose players' talents were as summarized in Table 8.2 instead of Table 8.1. Table 8.2 is the same as Table 8.1, except that now Hal can't play second base (position 4). Now can you find a feasible assignment?

Table 8.2 Positions players can play (updated)

Al	Bo	Che	Doug	Ella	Fay	Gene	Hal	Ian	John	Kit	Leo	Moe	Ned	Paul
2, 8	1, 5, 7	2, 3	1, 4, 5, 6, 7	3, 8	10, 11	3, 8, 11	2, 9	8, 9, 10	1, 5, 6, 7	8, 9	3, 9, 11	1, 4, 6, 7		9, 10

© Cengage Learning

This is just one of an almost unlimited number of real-world situations that can be modeled using a graph. A graph is a mathematical object that we will learn how to leverage in this chapter in order to solve relevant problems.

We won't attempt to be comprehensive in our coverage of graphs. In fact, we're going to consider only a few ideas from the branch of mathematics known as *graph theory*.

8.1 Graphs as Models

So far in this book we have seen a variety of mathematical models. Graphs are mathematical models too. In this section, we will look at two examples.

The Seven Bridges of Königsberg

In his 1736 paper *Solutio problematic ad geometriam situs pertinenis* (“The solution to a problem pertinent to the geometry of places"), the famous Swiss mathematician Leonhard Euler (pronounced “Oiler") addressed a problem of interest to the citizens of Königsberg, Prussia. At the time, there were seven bridges crossing various branches of the river Pregel, which runs through the city. Figure 8.1 is based on the diagram that appeared in Euler’s paper. The citizens of Königsberg enjoyed walking through the city—and specifically across the bridges. They wanted to start at some location in the city, cross each bridge *exactly* once, and end at the same (starting) location. Do you think this can be done? Euler developed a representation, or mathematical model, of the problem by using a graph. With the help of this model, he was able to answer the walkers’ question.

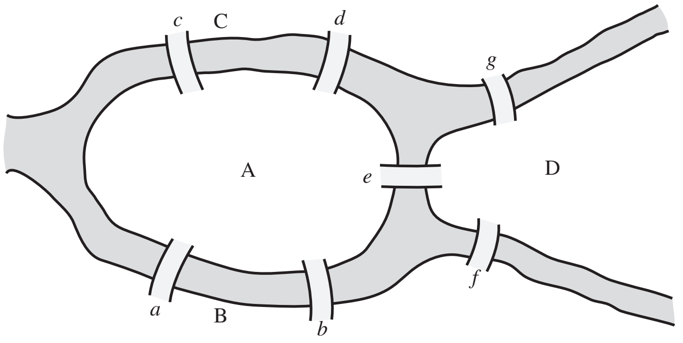


Figure 8.1 The seven bridges of Königsberg

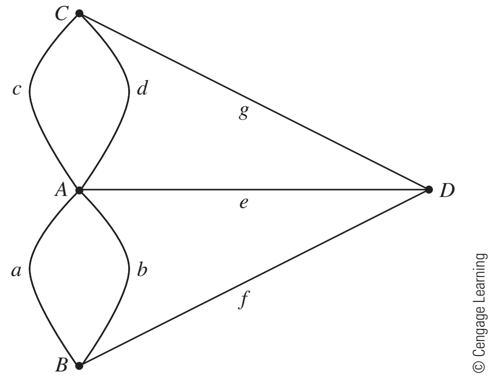
Euler’s Problem *Can the seven bridges be traversed exactly once starting and ending at the same place?* Try to answer this question yourself before reading on.

There are really two problems to solve here. One is transforming the map in Figure 8.1 into a graph. In this sense of the word, a **graph** is a mathematical way of describing relationships between things. In this case, we have a set of bridges and a set of land masses. We also have a relationship between these things; that is, each bridge joins exactly two specific land masses. The graph in Figure 8.2 is a mathematical model of the situation in Königsberg in 1736 with respect to what we can call the bridge-walking problem.

But Figure 8.2 doesn’t answer our original question directly. That brings us to the second problem. Given a graph that *models* the bridges and land masses, how can you tell whether it is possible to start on some land mass, cross every bridge exactly once, and end up where you started? Does it matter where you started? Euler answered these questions in 1736, and if you think about it for a while, you might be able to answer them too.

Euler actually showed that it was *impossible* to walk through Königsberg in the stated manner, regardless of where the walker started. Euler’s solution answered the bridge-walking question not only for Königsberg but also for every other city on Earth! In fact, in a

Figure 8.2
Graph model of Königsberg



sense he answered it for every possible city that might ever be built that has land masses and bridges that connect them. That's because he chose to solve the following problem instead.

Euler's Problem (restated) *Given a graph, under what conditions is it possible to find a closed walk that traverses every edge exactly once?* We will call graphs for which this kind of walk is possible **Eulerian**.

Which graphs are Eulerian? If you think for a moment, it is pretty clear that the graph has to be *connected*—that is, there must be a path between every pair of vertices. Another observation you might make is that whenever you walk over bridges between land masses and return to the starting point, the number of times you enter a land mass is the same as the number of times you leave it. If you add the number of times you enter a specific land mass to the number of times you leave it, you therefore get an even number. This means, in terms of the graph that represents the bridges and land masses, that an even number of edges *incident* with each vertex is needed. In the language of graph theory, we say that every vertex has *even degree* or that the graph has *even degree*.

Thus we have reasoned that Eulerian graphs must be connected and must have even degree. In other words, for a graph to be Eulerian, it is *necessary* that it both be connected and have even degree. But it is also true that for a graph to be Eulerian, it is *sufficient* that it be connected with even degree. Establishing necessary and sufficient conditions between two concepts—in this case, ''all Eulerian graphs'' and ''all connected graphs with even degree''—is an important idea in mathematics with practical consequences. Once we establish that being connected with even degree is necessary and sufficient for a graph to be Eulerian, we need only model a situation with a graph, and then check to see whether the graph is connected and each vertex of the graph has even degree. Almost any textbook on graph theory will contain a proof of Euler's result; see 8.2 Further Reading.

Graph Coloring

Our second example is also easy to describe.

Four-Color Problem *Given a geographic map, is it possible to color it with four colors so that any two regions that share a common border (of length greater than 0) are assigned different colors?* Figure 8.3 illustrates this problem.

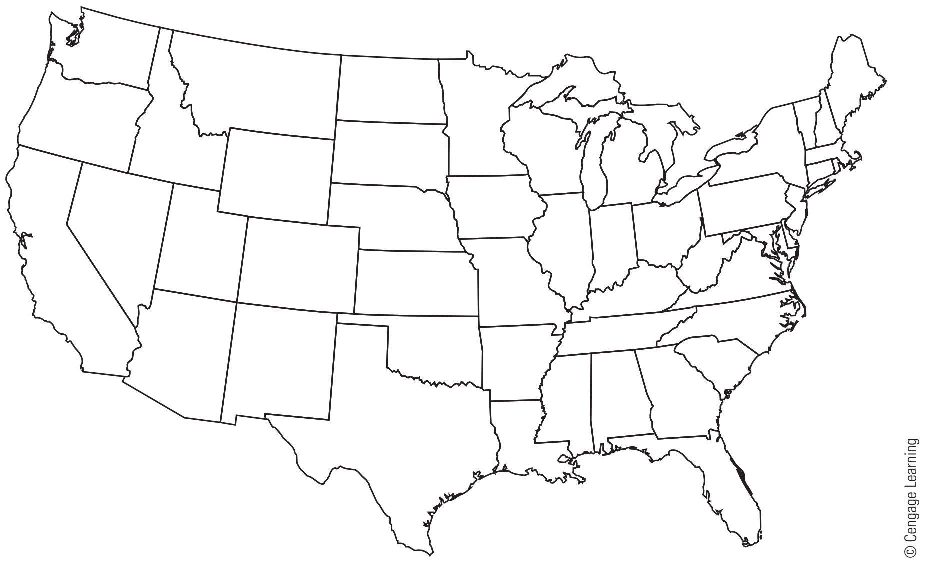


Figure 8.3
United States map

The four-color problem can be modeled with a graph. Figure 8.4 shows a map with a vertex representing each state in the continental United States and an edge between every pair of vertices corresponding to states that share a common (land) border. Note that Utah and New Mexico, for example, are not considered adjacent because their common border is only a point.

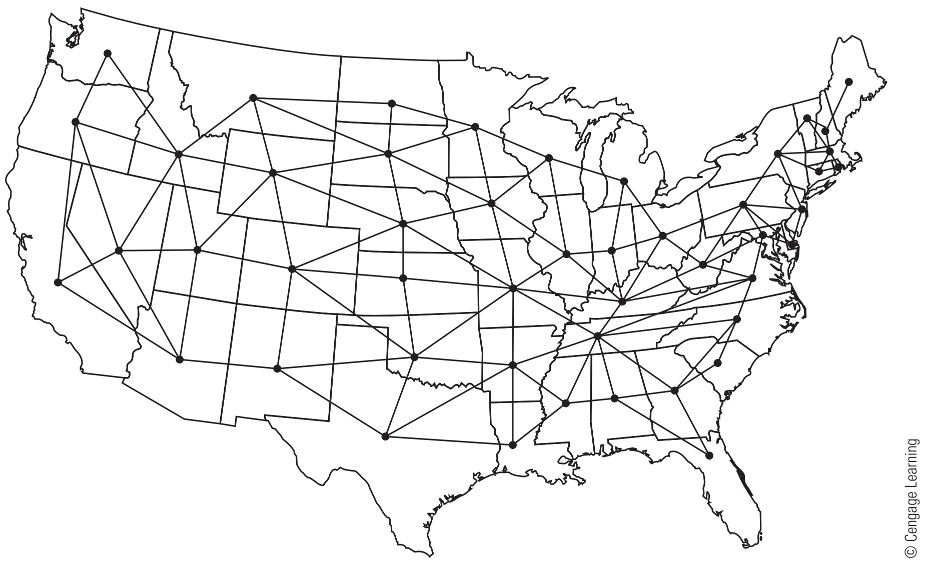


Figure 8.4
United States map with graph superimposed

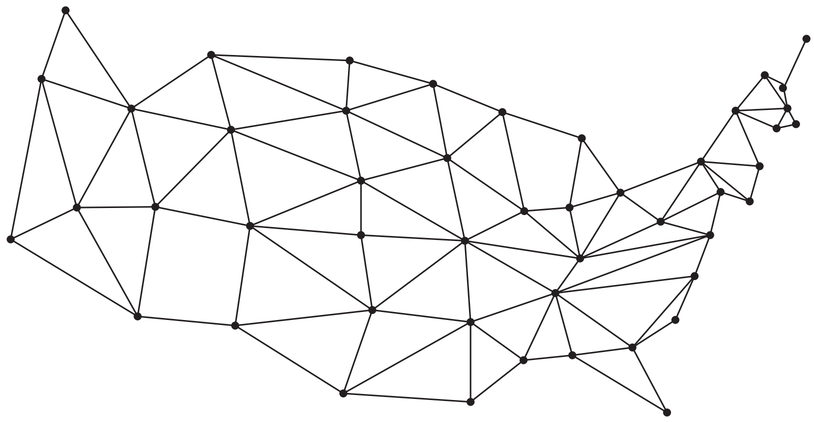


Figure 8.5
Graph-coloring problem

Next, Figure 8.5 shows the graph only. Now the original question we posed about the map of the United States has been transformed into a question about a graph.

Four-Color Problem (restated) *Using only four colors, can you color the vertices of a graph drawn without edges crossing so that no vertex gets the same color as an adjacent vertex?* Try it on the graph in Figure 8.5. We'll call a coloring **proper** if no two adjacent vertices share the same color. Assuming you can find a proper four coloring of the graph, it is very easy to color the map accordingly.

It is easy to see that a graph drawn without edges crossing (we'll call such a graph a *plane* graph) can be properly vertex colored with at most *six* colors. To see this, we will suppose instead that there is a plane graph that requires more than six colors to properly color its vertices. Now consider a smallest (in terms of number of vertices) of all such graphs; we'll call it G. We need to make use of the fact that every plane graph has a vertex with five or fewer incident edges (you can read about this fact in almost any graph theory textbook).

Let's let x be a vertex with five or fewer incident edges in G. Consider a new graph H that is formed by deleting x and all of its (at most five) incident edges. The new graph H is strictly smaller than G, so by hypothesis it must be possible to properly color its vertices with six colors. But now observe what this means for G. Apply the coloring used in H to color all of the vertices of G except for x. Since x has at most five neighbors in G, there are at most five colors used by the neighbors of x. This leaves the sixth color free to be used on x itself. This means we have properly colored the vertices of G using six colors. But in the previous paragraph we said G was a graph that required more than six colors. This contradiction means that our hypothesized plane graph G that requires more than six colors to properly color its vertices **cannot exist**. This establishes the fact that **all** plane graphs can be properly vertex colored using six (or fewer) colors.

The result of the previous paragraph has been known for over 100 years. Cartographers and others knew that maps that arose in practice (i.e., maps of actual places on the earth) could all be properly colored with only **four** colors, but there was no mathematical proof that every plane graph could be colored in this way. In 1879, the English mathematician Alfred Kempe published a proof of the so-called Four Color Theorem, but an error in his proof was discovered in 1890 that invalidated Kempe's theorem.

It wasn't until the dawn of the information age that a valid proof of the Four Color Theorem was published by Appel, Haken, and Koch in 1977. There was a good reason for the timing—the proof actually uses a computer! It relies on computer analysis of a very large number of possible cases called "configurations." So from 1890 until the 1970s Four Color Theorem was known as the Four Color *Conjecture*; in fact it was one of the most important and well-known mathematical conjectures of the 20th century.

Figure 8.6 shows a solution to the four-color problem on the graph in Figure 8.5.

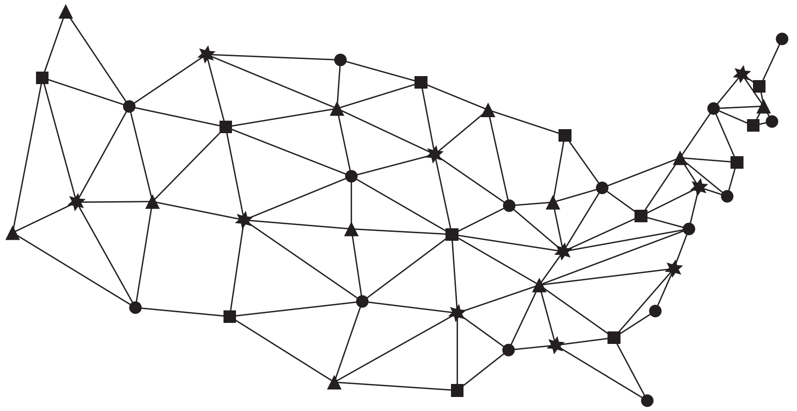


Figure 8.6
Graph-coloring solution (colors represented by shapes)

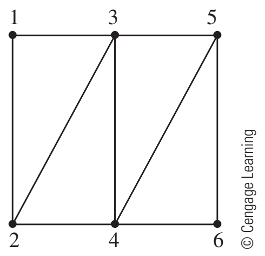
The four-color problem is really a special case of a more general graph-coloring problem: Given a graph, find the smallest number of colors needed to color the vertices properly. Graph coloring turns out to be a good model for a large variety of real-world problems. We explore a few of these below.

Applications of Graph Coloring One problem typically modeled with graph coloring is final exam scheduling. Suppose that a university has n courses in which a final exam will be given and that it desires to minimize the number of exam periods used, while avoiding "conflicts." A conflict happens when a student is scheduled for two exams at the same time. We can model this problem using a graph as follows. We start by creating a vertex for each course. Then we draw an edge between two vertices whenever there is a student enrolled in both courses corresponding to those vertices. Now we solve the graph-coloring problem; that is, we properly color the vertices of the resulting graph in a way that minimizes the number of colors used. The color classes are the time periods. If some vertices are colored blue in a proper coloring, then there can be no edges between any pair of them. This means that no student is enrolled in more than one of the classes corresponding to these vertices. Then each color class can be given its own time slot.

8.1 PROBLEMS

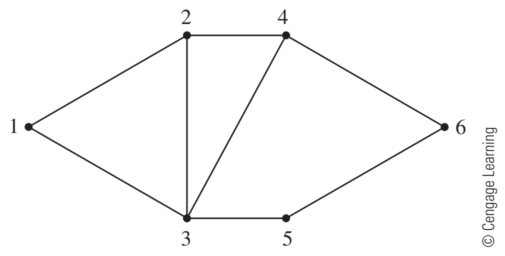
1. Solve the softball manager's problem (both versions) from the Introduction to this chapter.
2. The bridges and land masses of a certain city can be modeled with graph G in Figure 8.7.

Figure 8.7
Graph G



   a. Is G Eulerian? Why or why not?
   b. Suppose we relax the requirement of the walk so that the walker need not start and end at the same land mass but still must traverse every bridge exactly once. Is this type of walk possible in a city modeled by the graph in Figure 8.7? If so, how? If not, why not?
3. Find a political map of Australia. Create a graph model where there is a vertex for each of the six mainland states (Victoria, South Australia, Western Australia, Northern Territory, Queensland, and New South Wales) and an edge between two vertices if the corresponding states have a common border. Is the resulting graph Eulerian? Now suppose you add a seventh state (Tasmania) that is deemed to be adjacent (by boat) to South Australia, Northern Territory, Queensland, and New South Wales. Is the new graph Eulerian? If so, find a ‘‘walkabout" (a list of states) that shows this.
4. Can you think of other real-world problems that can be solved using techniques from the section about the bridges of Königsberg?
5. Consider the two political maps of Australia described in Problem 3. What is the smallest number of colors needed to color these maps?
6. Consider the graph of Figure 8.8.

Figure 8.8
Graph for Problem 6



   a. Color the graph with three colors.
   b. Now suppose that vertices 1 and 6 must be colored red. Can you still color the graph with three colors (including red)?

7. The Mathematics Department at a small college plans to schedule final exams. The class rosters for all the upper-class math courses are listed in Table 8.3. Find an exam schedule that minimizes the number of time periods used.

Table 8.3 Mathematics course rosters at Sunnyvale State

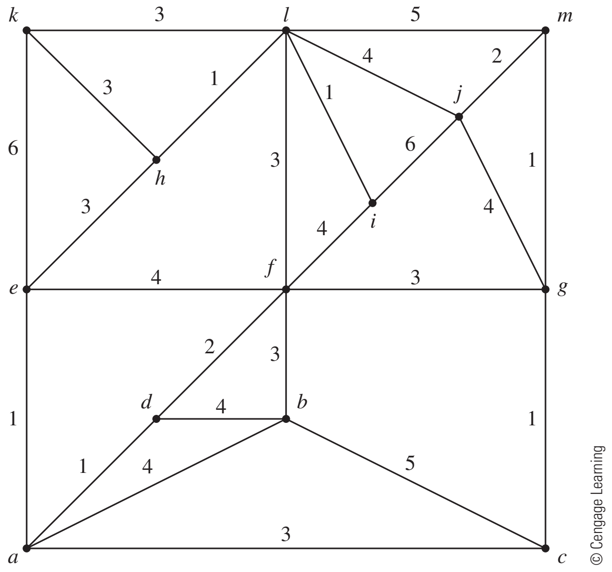
Course	Students			
math 350	Jimi	B. B.	Eric	
math 365	Ry	Jimmy P.	Carlos	
math 385	Jimi	Chrissie	Bonnie	Brian
math 420	Bonnie	Robin	Carlos	
math 430	Ry	B. B.	Buddy	Robin
math 445	Brian	Buddy		
math 460	Jimi	Ry	Brian	Mark

© Cengage Learning

8.1 PROJECT

1. Following a major storm, an inspector must walk down every street in a region to check for damaged power lines. Suppose the inspector's region can be modeled with the following graph. Vertices represent intersections, and edges represent streets. The numbers on the edges are called *edge weights*; they represent the distance the inspector must travel to inspect the corresponding street. How can the inspector check all the streets in a way that minimizes the total distance traveled? Hint: This has something to do with the Seven Bridges of Königsberg.

Figure 8.9
Graph for Project 1



8.1 Further Reading

Appel, K., W. Haken, & J. Koch. "Every planar map is four-colorable." *Illinois J. Math.,* 21(1977); 429–567.
Chartrand, G., & P. Zhang. *Introduction to Graph Theory*. New York: McGraw-Hill, 2005.
The Four Color Theorem (website). http://people.math.gatech.edu/~thomas/FC/fourcolor.html.
Robertson, N., D. Sanders, P. Seymour, & R. Thomas. "The four colour theorem." *J. Combin. Theory Ser. B.,* 70 (1997): 2–44.

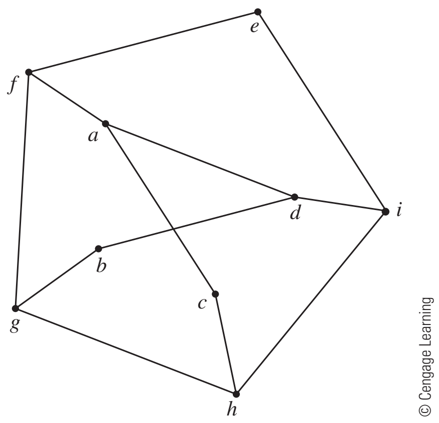
8.2 Describing Graphs

Before we proceed, we need to develop some basic notation and terminolgy that we can use to describe graphs. Our intent is to present just enough information to begin our discussion about modeling with graphs.

As we have noted, a **graph** is a mathematical way of describing relationships between things. A graph G consists of two sets: a **vertex set** $V(G)$ and an **edge set** $E(G)$. Each element of $E(G)$ is a pair of elements of $V(G)$. Figure 8.10 shows an example. When we refer to the vertices of a graph, we often write them using set notation. In our example we would write $V(G) = \{a, b, c, d, e, f, g, h, i\}$. The edge set is often described as a set of pairs of vertices; for our example we could write $E(G) = \{ac, ad, af, bd, bg, ch, di, ef, ei, fg, gh, hi\}$. Vertices don't have to be labeled with letters. They can instead be labeled with numbers, or the names of things they represent, and so forth. Note that when we draw a graph, we might decide to draw it so that edges cross (or we might be forced to do so by the nature of the graph). Places on the paper where edges cross are not necessarily vertices; the example of Figure 8.10 shows a crossing point where there is no vertex.

When an edge ij has a vertex j as one of its endpoints, we say edge ij is **incident** with vertex j. For example, in the graph of Figure 8.10, edge bd is incident with vertex b but not with vertex a. When there is an edge ij between two vertices, we say vertices i and j are **adjacent**. In our example, vertices c and h are adjacent but a and b are not. The **degree** of a vertex j, deg(j), is the number of incidences between j and an edge. In our example, deg$(b) = 2$ and deg$(a) = 3$. As we saw in the previous section, a vertex v is said to have

Figure 8.10
Example graph



even degree if $\deg(v)$ is an even number (a number divisible by 2). Similarly, a graph is said to have even degree if every vertex in the graph has even degree.

Because graphs are often described using sets, we need to introduce some set notation. If S is a set, then $|S|$ denotes the number of elements in S. In our example, $|V(G)| = 9$ and $|E(G)| = 12$. We use the symbol $\in$ as shorthand for "is an element of" and $\notin$ for "is not an element of." In our example, $c \in V(G)$ and $bd \in E(G)$, but $m \notin V(G)$ and $b \notin E(G)$ (because b is a vertex and not an edge).

We will also use *summation* notation. The Greek letter $\sum$ (that's a capital letter sigma) is used to represent the idea of adding things up. For example, suppose we have a set $Q = \{q_1, q_2, q_3, q_4\} = \{1, 3, 5, 7\}$. We can succinctly express the idea of adding up the elements of Q using the summation symbol like this:

$$\sum_{q_i \in Q} q_i = 1 + 3 + 5 + 7 = 16$$

If we were reading the previous line aloud, we would say, "The sum of q sub i, for all q sub i in the set Q, equals 1 plus 3 plus 5 plus 7, which equals 16." Another way to express the same idea is

$$\sum_{i=1}^{4} q_i = 1 + 3 + 5 + 7 = 16$$

In this case, we say, "The sum, for i equals 1 to 4, of q sub i, equals 1 plus 3 plus 5 plus 7, which equals 16." We can also perform the summation operation on other functions of q. For example,

$$\sum_{i=1}^{4} \left(q_i^2 + 4\right) = (1^2 + 4) + (3^2 + 4) + (5^2 + 4) + (7^2 + 4) = 100$$

8.2 PROBLEMS

1. Consider the graph in Figure 8.11.

a. Write down the set of edges $E(G)$.

b. Which edges are incident with vertex b?

c. Which vertices are adjacent to vertex c?

d. Compute $\deg(a)$.

e. Compute $|E(G)|$.

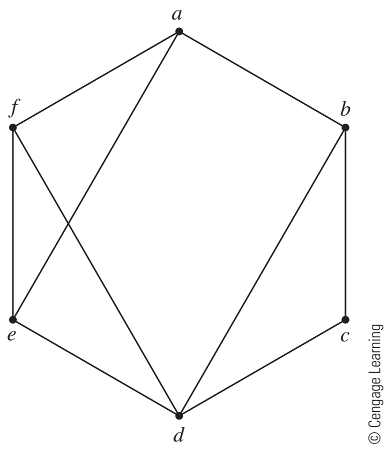
2. Suppose $r_1 = 4$, $r_2 = 3$, and $r_3 = 7$.

a. Compute $\sum_{i=1}^{3} r_i$.

b. Compute $\sum_{i=1}^{3} r_i^2$.

c. Compute $\sum_{i=1}^{3} i r_i$.

Figure 8.11
Graph for Problem 1



3. At a large meeting of business executives, lots of people shake hands. Everyone at the meeting is asked to keep track of the number of times she or he shook hands, and as the meeting ends, these data are collected. Explain why you will obtain an even number if you add up all the individual handshake numbers collected. What does this have to do with graphs? Express this idea using the notation introduced in this section.

8.2 Further Reading

Buckley, F., & M. Lewinter. *A Friendly Introduction to Graph Theory*. Upper Saddle River, NJ: Pearson Education, 2003.
Chartrand, G., & P. Zhang. *Introduction to Graph Theory*. New York: McGraw Hill, 2005.
Chung, F., & R. Graham. *Erdös on Graphs*. Wellesley, MA: A. K. Peters, 1998.
Diestel, R. *Graph Theory*. Springer Verlag, 1992.
Goodaire, E., & M. Parmenter. *Discrete Mathematics with Graph Theory*, 2nd ed. Upper Saddle River, NJ: Prentice Hall, 2002.
West, D. *Introduction to Graph Theory*, 2nd ed. Upper Saddle River, NJ: Prentice Hall, 2001.

8.3 Graph Models

Bacon Numbers

You may have heard of the popular trivia game "Six Degrees of Kevin Bacon." In this game, players attempt to connect an actor to Kevin Bacon quickly by using a small number of connections. A *connection* in this context is a movie in which two actors you are connecting both appeared. The *Bacon number* of an actor is the smallest number of connections needed to connect him or her with Kevin Bacon. The *Bacon number problem* is the problem of computing Bacon numbers. An *instance* of the Bacon number problem is computing the

Bacon number for a specific actor. For example, consider Elvis Presley. Elvis was in the movie *Change of Habit* with Ed Asner in 1969, and Ed Asner was in *JFK* with Kevin Bacon in 1991. This means that Elvis Presley's Bacon number is at most 2. Because Elvis never appeared in a movie with Kevin Bacon, his Bacon number can't be 1, so we know Elvis Presley's Bacon number *is* 2.

For another example, consider Babe Ruth, the New York Yankees baseball star of the 1930s. "The Babe" actually had a few acting parts, and he has a Bacon number of 3. He was in *The Pride of the Yankees* in 1942 with Teresa Wright; Teresa Wright was in *Somewhere in Time* (1980) with JoBe Cerny; and JoBe Cerny was in *Novocaine* (2001) with Kevin Bacon. How can we determine the Bacon numbers of other actors? By now, you won't be surprised to hear that the problem can be modeled using a graph.

To model the problem of determining an actor's Bacon number, let $G = (V(G), E(G))$ be a graph with one vertex for each actor who ever played in a movie (here we use the word *actor* in the modern sense to include both sexes). There is an edge between vertices representing two actors if they appeared together in a movie.

We should pause for a moment to consider some practical aspects of the resulting graph. For one thing, it is enormous! The Internet Movie Database (www.imdb.com) lists over a million actors—meaning there are over a million vertices in the graph—and over 300,000 movies. There is a potential new edge for every distinct pair of actors in a movie. For example, consider a small movie with 10 actors. Assuming these actors have not worked together on another movie, the first actor's vertex has 9 new edges going to the other 9 actors' vertices. The second actor's vertex will have 8 new edges, since we don't want to double-count the connection between the first and second actors. The pattern continues, so this single movie creates $9 + 8 + 7 + \cdots + 1 = 45$ new edges. In general, a movie with n actors has the potential to add $\frac{n(n-1)}{2}$ edges to the graph. We say "potential" to acknowledge the possibility that some of the n actors have appeared together in another movie. Of course, the total number of vertices in the graph grows too, because the list of actors who have appeared in a movie grows over time.

Despite the size of the associated graph, solving the Bacon number problem for a given actor is simple in concept. You just need to find the length of a shortest path in the graph from the vertex corresponding to your actor to the vertex corresponding to Kevin Bacon. Note that we said *a* shortest path, not *the* shortest path. The graph might have several paths that are "tied" for being the shortest. We consider the question of finding a shortest path between two vertices in Section 8.4.

The Bacon number graph is just one of a broader class of models called social networks. A **social network** consists of a set of individuals, groups, or organizations and certain social relationships between them. These networks can be modeled with a graph. For example, a **friendship network** is a graph where the vertices are people and there is an edge between two people if they are friends. Once a social network model is constructed, a variety of mathematical techniques can be applied to gain insight into the situation under study. For example, a recent Harvard Medical School study used friendship networks to investigate the prevelance of obesity in people. The investigators found that having fat friends is a strong predictor of obesity. That is, people with obese friends are more likely to be obese themselves than are people without obese friends. There are many other social relationships that could be modeled and then analyzed in this way. Powerful computational tools have recently been developed to keep track of and analyze huge social networks. This sort of analysis is a part of an emerging field called **network science** that may shed new light on many old problems.

Fitting a Piecewise Linear Function to Data

Suppose you have a collection of data $p_1, p_2, \ldots, p_n$, where each p_i is an ordered pair (x_i, y_i). Further suppose that the data are ordered so that $x_1 \le x_2 \le \cdots \le x_n$. You could plot these data by thinking of each p_i as a point in the xy-plane. A small example with the data set $S = \{(0,0), (2,5), (3,1), (6,4), (8,10), (10,13), (13,11)\}$ is given in Figure 8.12.

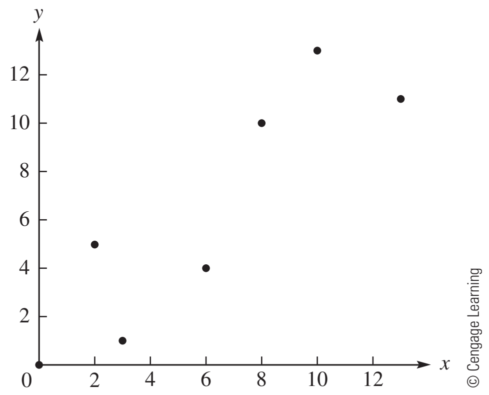


Figure 8.12
Data example

There are many applications where a piecewise linear function that goes through *some* of the points is desired. On the one hand, we could build our model to go through just the first and last points. To do this, we simply draw a line from the first point $p_1 = (x_1, y_1)$ to the last point $p_n = (x_n, y_n)$, obtaining the model

$$y = \frac{y_n - y_1}{x_n - x_1}(x - x_1) + y_1 \tag{8.1}$$

Figure 8.13 shows Model (8.1) displayed on the data.

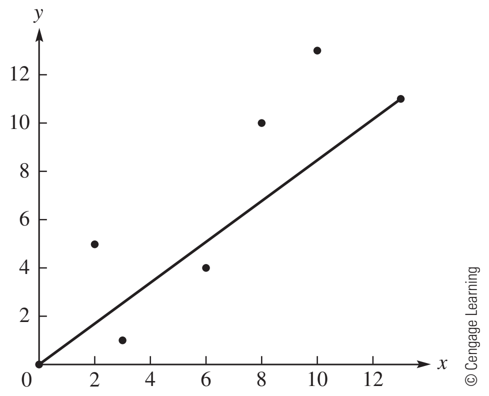


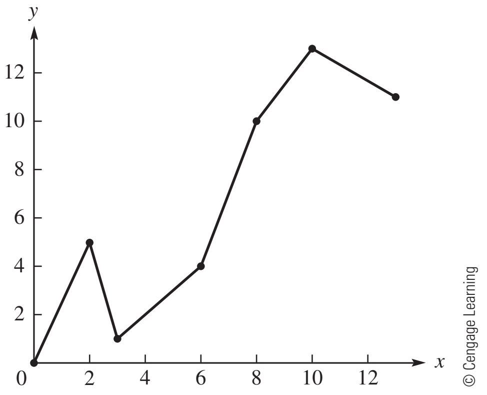
Figure 8.13
Data with Model (8.1)

On the other hand, we could draw a line segment from (x_1, y_1) to (x_2, y_2), another from (x_2, y_2) to (x_3, y_3), and so forth, all the way to (x_n, y_n). There are $n - 1$ line segments in all, and they can be described as follows:

$$y = \frac{y_{i+1} - y_i}{x_{i+1} - x_i}(x - x_i) + y_i \quad \text{for } x_i \le x \le x_{i+1} \tag{8.2}$$

where $i = 1, 2, \dots, n-1$. Figure 8.14 shows the data with Model (8.2).

Figure 8.14
Data with Model (8.2)



There is a trade-off between these two extreme choices. Model (8.1) is simple to obtain and simple to use, but it might miss some of the points by a great amount. Model (8.2) doesn't miss any points, but it is complicated, especially if the data set contains many points—that is, when n is large.

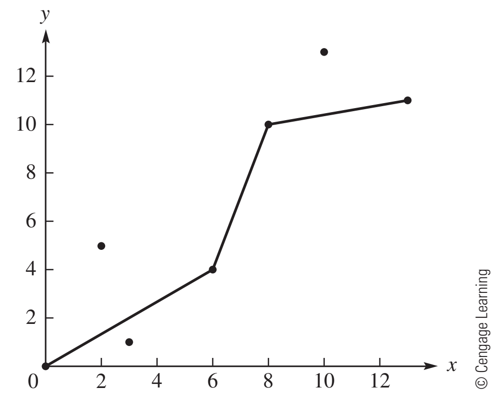
One option is to build a model that uses more than the one line segment present in Model (8.1), but fewer than the $n-1$ line segments used in Model (8.2). Suppose we assume that the model at least goes through the first point and the last point (in this case p_1 and p_7, respectively). One possible model for the specific data set given is to choose to go through points $p_1 = (0,0)$, $p_4 = (6,4)$, $p_5 = (8,10)$, and $p_7 = (13,11)$. Here is an algebraic description of this idea:

$$y = \begin{cases} \frac{2}{3}x & \text{for } 0 \le x < 6 \\ 3(x-6) + 4 & \text{for } 6 \le x < 8 \\ \frac{1}{5}(x-8) + 10 & \text{for } 8 \le x \le 13 \end{cases} \tag{8.3}$$

A graphical interpretation of the same idea appears in Figure 8.15.

Now which model is *best*? There isn't a universal answer to that question; the situation and context must dictate which choice is best. We can, however, develop a framework for analyzing how good a given model is. Suppose there is a cost associated with having the model not go through a given point. In this case, we say the model "misses" that point. There are many ways to formalize this idea, but one natural way is to use the least-squares criteria discussed in Chapter 3. To capture the idea of a trade-off, there also needs to be a cost associated with each line segment that we use in our model. Thus Model (8.1) has a relatively high cost for missing points, but a minimum cost for using line segments. Model (8.2) has a minimum cost for missing points (in fact that cost is 0!) but has the largest reasonable cost for line segments.

Figure 8.15
Data with Model (8.3)



Let's be a little more specific and see how we can find the best possible model. First recall that we always go through p_1 and p_n. So we must decide which of the other points $p_2, p_3, \ldots, p_{n-1}$ we will visit. Suppose we are considering going directly from p_1 to p_4, as we did in Model (8.3). What is the cost of this decision? We have to pay the fixed cost of adding a line segment to our model. Let's use α to represent that cost parameter. We also have to pay for all the points we miss along the way. We will use the notation $f_{i,j}$ to represent the line segment from p_i to p_j. In Model (8.3), $f_{1,4}(x) = \frac{2}{3}x$ and $f_{4,5}(x) = 3(x-6)+4$, because those are the algebraic descriptions of the line segments going from p_1 to p_4 and from p_4 to p_5, respectively. Accordingly, we can use f to compute the y value associated with a specific part of the model. For example, $f_{1,4}(x_4) = \frac{2}{3}x_4 = \frac{2}{3}(6) = 4$. This isn't surprising. Recall that $p_4 = (x_4, y_4) = (6, 4)$, so we expect $f_{1,4}(x_4)$ to equal 4 because we have chosen a model that goes through p_4. On the other hand, $f_{1,4}(x_3) = \frac{2}{3}x_3 = \frac{2}{3}(3) = 2$, but $y_3 = 1$, so our line drawn from p_1 to p_4 misses p_3. In fact, the amount it misses by, or the **error**, is the absolute difference between $f_{1,4}(x_3)$ and y_3, which is $|2-1| = 1$. To avoid some of the complexities of dealing with the absolute value function, we will square these errors, as we did in Chapter 3. When we chose to skip points p_2 and p_3 in Model (8.3), we did so believing that it was reasonable to pay the price of missing these points in exchange for using only one line segment on the interval from x_1 to x_4. The standard least-squares measure of the extent to which we missed points on this interval is

$$\sum_{k=1}^{4} (f_{1,4}(x_k) - y_k)^2 \tag{8.4}$$

The expression 8.4 simply adds up the squared vertical distances, which are the squared errors, between the line segment $f_{1,4}$ and the original data points. It does this for each point p_1, p_2, p_3, and p_4. Table 8.4 shows how the sum of squared errors is used to compute the value of (8.4).

Thus

$$\sum_{k=1}^{4} (f_{1,4}(x_k) - y_k)^2 = 0 + \frac{121}{9} + 1 + 0 = \frac{130}{9} \approx 14.4444$$

Table 8.4 Computing the sum of squared errors for (8.4)

k	x_k	$f_{1,4}(x_k)$	y_k	$(f_{1,4}(x_k) - y_k)^2$
1	0	0	0	0
2	2	$\frac{4}{3}$	5	$\frac{121}{9}$
3	3	2	1	1
4	6	4	4	0

© Cengage Learning

Verbally, it means that the line that goes from $p_1 = (0, 0)$ to $p_4 = (6, 4)$ (we call that line $f_{1,4}$) doesn't miss p_1 at all, misses p_2 by $\frac{11}{3}$ units (note that $\left(\frac{11}{3}\right)^2 = \frac{121}{9}$), misses p_3 by 1 unit (as we saw in the previous paragraph), and doesn't miss p_4. The total of $\frac{130}{9} \approx 14.4444$ is a measure of the extent to which $f_{1,4}$ fails to accurately model the behavior of the data we have between the first and fourth points.

Although the sum of the squared errors may not describe the "cost" of missing the points along the way, we can scale the sum so that it does. We will use β to represent the cost per unit of summed squared errors. The modeler can choose a value of β that reflects his or her aversion to missing points, also considering the extent to which they are missed. Similarly, we can let α be the fixed cost associated with each line segment. A model with only one line segment will pay the cost α once, and a model with, say, five line segments will pay α five times. Then the total cost of the portion of the model going from p_1 to p_4 is

$$\alpha + \beta \sum_{k=1}^{4} (f_{1,4}(x_k) - y_k)^2 \tag{8.5}$$

If we choose parameter values $\alpha = 10$ and $\beta = 1$, then the total cost of the model over the interval from p_1 to p_4 is

$$\alpha + \beta \sum_{k=1}^{4} (f_{1,4}(x_k) - y_k)^2 = 10 + 1\frac{130}{9} \approx 24.4444 \tag{8.6}$$

It is easy, though somewhat tedious, to employ this same procedure for the other possible choices for our model. Building on what we have already done, we could easily compute the cost associated with the portion of our model that goes from p_4 to p_5. It is 10 because all we are paying for is the line segment cost (no points are missed in this case, on this interval). Finally, we could compute the cost of the remaining portion of the model. So we just compute

$$\alpha + \beta \sum_{k=5}^{7} (f_{5,7}(x_k) - y_k)^2 = 10 + 1\frac{169}{25} = 16.76 \tag{8.7}$$

The details of this computation are left to the reader. Thus the total cost of Model (8.3) is about $24.4444 + 10 + 16.76 = 51.2044$.

We could also compute the cost of choices we did *not* make in Model (8.3). In fact, if we compute the cost of *each possible choice* we can make in creating a similar model, we can compare options to see which model is best, given the data and our selected values of α and β. Table 8.5 shows the computed costs of all possible segments of a piecewise linear model for our data and parameter values.

Table 8.5 Cost for each line segment

	2	3	4	5	6	7
1	10	28.7778	24.4444	36.0625	38.77	55.503
2		10	24.0625	52.1389	61	56.9917
3			10	15.76	14.7755	51
4				10	12.25	51
5					10	16.76
6						10

© Cengage Learning

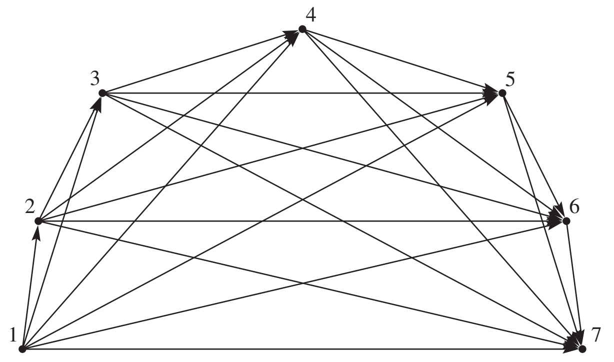
Now we can use Table 8.5 to compute the total cost of any peicewise linear model we want to consider in the following way. Let $c_{i,j}$ be the cost of a line segment from point p_i to point p_j. In other words, $c_{i,j}$ is the entry in row i and column j in Table 8.5. Recall our first model, (8.1). Because this model had one line segment going from p_1 to p_7, we can compute its cost by looking at the entry in row 1 and column 7 of Table 8.5, which is $c_{1,7} = 55.503$. Our next model, (8.2), included six line segments; from p_1 to p_2, from p_2 to p_3, and so on, up to p_6 to p_7. These correspond to the lowest entry in each column of Table 8.5, so the total cost of this model is $c_{1,2} + c_{2,3} + \cdots + c_{6,7} = 10 + 10 + 10 + 10 + 10 + 10 = 60$. Other models can be considered as well. For example, suppose we decided to go from p_1 to p_3 to p_6 to p_7. This model costs $c_{1,3}+c_{3,6}+c_{6,7} = 28.7778+14.7755+10 = 53.5533$.

It is natural at this point to ask which model is best of all the possible models. We are looking for a piecewise linear model that starts at p_1 and ends at p_7 and has the smallest possible cost. We just have to decide which points we will visit. We can visit or not visit each of p_2, p_3, p_4, p_5, and p_6, because the conditions of the problem force us to visit p_1 and p_7. You can think of this as looking for a **path** that goes from p_1 to p_7, visiting any number of the five intermediate points. Of all those paths we want to find the best (cheapest) one.

One way to find the best model is to look at every possible model and pick the best one. For our example, this isn't a bad idea. Because each of the five points p_2, p_3, p_4, p_5, and p_6 is either visited or not visited, there are $2^5 = 32$ possible paths from p_1 to p_7. It would not be too hard to check all 32 of these and pick the best one. But think about what we would be up against if the original data set contained, say, 100 points instead of 7. In this case there would be $2^{98} = 316{,}912{,}650{,}057{,}350{,}374{,}175{,}801{,}244$ possible models. Even if we could test a million models every *second*, it would take more than 10,000,000,000,000,000 *years* to check them all. This is almost a million times longer than the time since the big bang! Fortunately, there is a better way to solve this kind of problem.

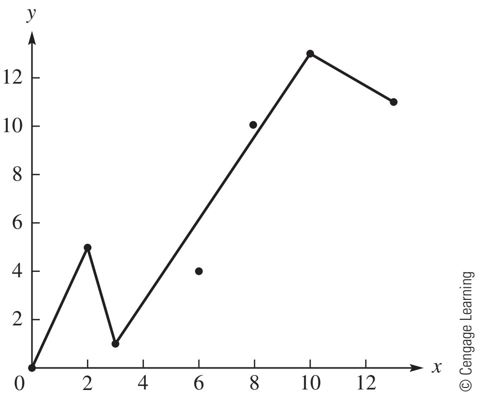
It turns out that we can also find the best model by solving a simple graph problem. Consider Figure 8.16. This represents the paths from p_1 to p_7 as a directed graph. Each path described in the paragraphs above is represented by a path from vertex 1 to vertex 7 in Figure 8.16. Look again at Table 8.5. If we use the data from this table as edge weights for the graph in Figure 8.16, we can solve our data-fitting problem of finding the best-fitting piecewise linear function by finding a shortest path in the graph.

Figure 8.16
Graph model for data fitting



We will learn how to solve shortest-path problems later in this chapter. Once the shortest-path problem on the graph in Figure 8.16 using the cost data in Table 8.5 is solved, it will be clear that the best solution for this particular instance is the path from p_1 to p_2 to p_3 to p_6 to p_7. The model has a total cost of $c_{1,2}+c_{2,3}+c_{3,6}+c_{6,7} = 10 + 10 + 14.7755 + 10 = 44.7755$. It turns out that this is the best possible model for our chosen parameter values $\alpha = 10$ and $\beta = 1$. This *optimal* model appears in Figure 8.17.

Figure 8.17
Optimal piecewise linear model



Recall that in Section 8.3 we found that the Bacon number problem can be solved by finding the distance between two vertices in a graph. Finding the distance between two vertices is really a great deal like finding the shortest path between two vertices—we'll state this fact with more precision at the beginning of Section 8.4. One remarkable aspect of mathematics is that technical procedures you can learn in the classroom (such as solving shortest-path problems) can be directly applied in many settings (such as finding Bacon numbers and fitting linear models to data). It is also interesting to observe that mathematics can link two situations that seem vastly different. The data-fitting problem and the Bacon number problem are very similar; they both reduce to a shortest-path problem. In Section 8.4 we will learn about a procedure to solve shortest-path problems on graphs.

The Softball Example Revisited

Recall the softball example from the Introduction to this chapter. Each player can play only in the positions specified in Table 8.6.

Table 8.6 Positions players can play

Al	Bo	Che	Doug	Ella	Fay	Gene	Hal	Ian	John	Kit	Leo	Moe	Ned	Paul
2,8	1,5,7	2,3	1,4,5,6,7	3,8	10,11	3,8,11	2,4,9	8,9,10	1,5,6,7	8,9	3,9,11	1,4,6,7		9,10

This problem can be modeled using a graph. We'll construct a graph from Table 8.6 as follows. For each player, we'll create a vertex. We'll call these vertices *player vertices*. We will also create a vertex for each position and call these *position vertices*. If we let A represent the player vertices and B represent the position vertices, we can write $V(G) = \langle A, B \rangle$ to represent the idea that the vertex set for our graph has two distinct types of vertices. Now we can create an edge in our graph for every entry in Table 8.6. That is, there is an edge between a specific player vertex and a specific position vertex whenever that player can play that position according to our table. For example, our graph has an edge from the vertex representing Che to the vertex representing position 3 (first base), but there is no edge from Che to, say, position 6.

The graph for the softball problem is shown in Figure 8.18. Note that the A vertices are on the left and the B vertices are on the right. Also note that all of the edges in the graph have one end in A and the other end in B. Graphs with this special property are called **bipartite**. The name comes from the fact that the vertex set can be partitioned (*-partite*) into two (*bi-*) sets (A and B) so that all of the edges have one end in A and one end in B. As we will see in Section 8.4, we can solve some kinds of problems particularly easily on bipartite graphs.

Let's return to the problem of solving the softball manager's problem. We'd like an assignment of players to positions (or positions to players) that covers every position. Think

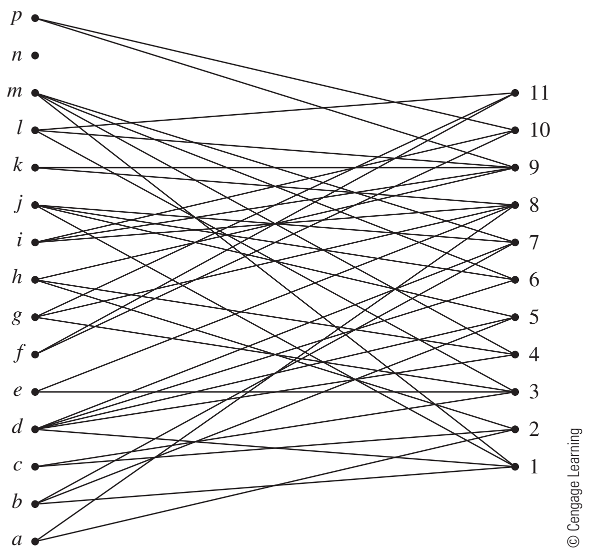


Figure 8.18
Graph for the softball problem

about this in terms of Figure 8.18. You can think of the edge from, say, Che to position 3 as having the option of assigning Che to position 3. Note that once you decide to put Che at position 3, Che cannot be assigned a different position, and no other player can play in position 3. In terms of the graph, we can think of choosing some of the edges so that no two of the chosen edges are incident with the same vertex. If we can choose enough of the edges (in this case, 11 of them), we have solved the softball manager's problem.

Given any graph $G = (V(G), E(G))$, a subset $M \subseteq E(G)$ is called a **matching** if no two members of M are incident with the same vertex. A set of edges with this property are said to be **independent**. A **maximum matching** is a matching of the largest possible size. When G is bipartite with bipartition $\langle A, B\rangle$, it is clear that no matching can be bigger than $|A|$, and no matching can be bigger than $|B|$. Therefore, if we can find a matching in the graph for the softball problem that has 11 edges, then there is a feasible solution to the softball manager's problem. Furthermore, if the largest matching in the graph has fewer than 11 edges, *no feasible solution to the softball manager's problem is possible*. That is, the graph has a matching of size eleven **if and only if** there is a feasible solution to the softball manager's problem.

We have shown the relationship between the softball manager's problem and a matching problem in a bipartite graph. In Section 8.4, we will see how to find maximum matchings in bipartite graphs.

A 0–1 Matrix Problem

Informally, a **matrix** is an array or table with rows and columns. A matrix usually has an entry for each possible row and column pair. The entries themselves are ordinarily numbers. An "m by n 0–1 matrix" is a matrix that has m rows and n columns, where each entry is either 0 or 1. Using 0's and 1's in a mathematical model is a common way of expressing ideas such as "yes" or "no," "true" or "false," "on" or "off," and so forth. In this section, we analyze a problem related to 0–1 matrices.

Consider an m by n 0–1 matrix. Let r_i for $i \in \{1, 2, \ldots, m\}$ be the row sum of the ith row, or, equivalently, the number of 1's in the ith row. Similarly, let s_i for $i \in \{1, 2, \ldots, n\}$ be the column sum of the ith column, or, equivalently, the number of 1's in the ith column. Below is an example with $m = 4$, $n = 6$, $r_1 = 3$, $r_2 = 2$, $r_3 = 3$, $r_4 = 4$, $s_1 = 3$, $s_2 = 2$, $s_3 = 2$, $s_4 = 3$, $s_5 = 1$, $s_6 = 1$.

$$\begin{pmatrix} 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 \end{pmatrix}$$

Now consider the following problem.

The 0–1 Matrix Problem *Given values for m and n along with $r_1, r_2, \ldots, r_m$ and $s_1, s_2, \ldots, s_n$, does there exist a 0–1 matrix with those properties?* If you were given

$$\begin{gathered} m = 4, n = 6 \\ r_1 = 3, r_2 = 2, r_3 = 3, r_4 = 4 \\ s_1 = 3, s_2 = 2, s_3 = 2, s_4 = 3, s_5 = 1, s_6 = 1 \end{gathered} \tag{8.8}$$

the answer is clearly "yes," as the matrix above demonstrates. It's easy to say "no" if $r_1 + r_2 + \cdots + r_m \neq s_1 + s_2 + \cdots + s_n$ because the sum of the row sum values must equal the sum of the column sum values (why?). But if you were given values of m and n and $r_1, r_2, \ldots, r_m$ and $s_1, s_2, \ldots, s_n$, where $r_1 + r_2 + \cdots + r_m = s_1 + s_2 + \cdots + s_n$, it might not be so easy to decide whether such a matrix is possible.

Let's pause for a moment to consider why we might *care* about this matrix problem. Some real-world problems can be modeled this way. For example, suppose a supply network has suppliers who supply widgets and has demanders who demand them. Each supplier can send at most one widget to each demander.

The 0–1 Matrix Problem (restated) *Is there a way to determine which demander each supplier sends widgets to in a way that satisfies constraints on how many widgets each supplier produces and on how many widgets each demander demands?*

We can model this situation with some kind of graph. First draw m "row" vertices and n "column" vertices, along with two additional vertices: one called s and one called t. From s, draw an arc with capacity r_i to the ith row vertex. From the ith column vertex, draw an arc with capacity s_i to t. Then, from every row vertex, draw an arc with capacity 1 to every column vertex.

To decide whether there is a 0–1 matrix with the specified properties, we find the maximum flow from s to t in the directed graph, see Figure 8.19. If there's a flow of magnitude $r_1 + r_2 + \cdots + r_m$, then the answer to the original matrix question is "yes." Furthermore, the arcs that go from row vertices to column vertices that have a flow of 1 on them in the solution to the maximum-flow problem correspond to the places to put the 1's in the matrix to achieve the desired property. If the maximum flow you find is less than $r_1 + r_2 + \cdots + r_m$, we can say "no" to the original matrix question. Figure 8.19 shows the graph constructed in accordance with this transformation with the data specified in (8.8).

Policing a City

Suppose you are the commander of a company of military police soldiers and you are charged with policing all the roads in a section of a city. The section of the city in question

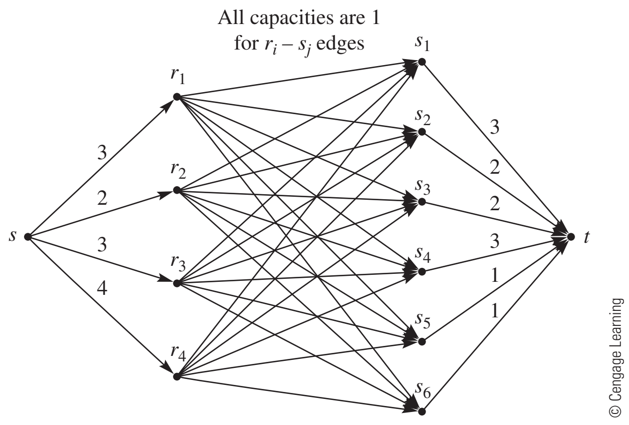


Figure 8.19
Directed graph with arc capacities

can be modeled with the graph in Figure 8.20, where the road intersections in the city are vertices and the roads between the intersections are edges. Suppose the roads in the city are straight enough and short enough so a soldier stationed at an intersection can effectively police all roads immediately incident with that intersection. For example, a soldier stationed at vertex (intersection) 7 can police the road between 4 and 7, the road between 6 and 7, and the road between 7 and 8.

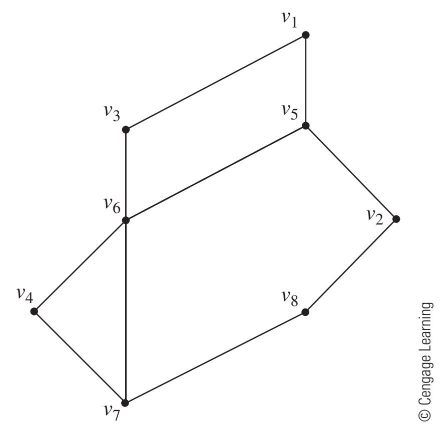


Figure 8.20
Vertex cover example graph

Vertex Cover Problem *What is the smallest number of military police soldiers needed to accomplish the mission of policing all the roads?*

It is easy to see that all roads can be policed by placing a soldier at every vertex. This requires 8 soldiers but this may not be the smallest number of soldiers required. A better solution places soldiers at intersections 3, 4, 5, 6, and 8—using only 5 soldiers. Note that you can't improve on this solution simply by removing a soldier (and not moving another one). An even better solution is to locate soldiers at intersections 1, 2, 6, and 7. It turns out that using only 4 soldiers is the best you can do for this graph.

Let's describe this problem in mathematical terms.

Vertex Cover Problem (restated) *Given a graph $G = (V(G), E(G))$, the vertex cover problem seeks a subset $S \subseteq V(G)$ of smallest number of elements such that every edge in the graph is incident with at least one element in S.*

An edge e is said to be **incident** with a vertex v if one of the endpoints of e is v. If G is a graph, we denote the minimum size of a vertex cover as $\beta(G)$. Another way of expressing this is $\beta(G) = \min\{|S| : S \text{ is a vertex cover}\}$.

8.3 PROBLEMS

1. Suppose you are an actor and your Bacon number is 3. Can future events ever cause your Bacon number to rise above 3? What, in general, can you say about an actor's Bacon number in terms of how it can change over time?

2. Just as actors have their Bacon numbers, there is a relation defined between authors of scholarly papers and the prolific Hungarian mathematician Paul Erdös. Use the Internet

to find out what you can about Paul Erdös and Erdös numbers. Consider your favorite mathematics professor (no jokes here, please!). Can you determine his or her Erdös number?

3. Can you think of other relations that one could consider?

4. For the example in the text above, explain why, in Table 8.5, the entries in row 3, column 7, and row 4, column 7 are the same. *Hint:* Plot the data and draw a line segment from point 3 to point 7, and another from point 4 to point 7.

5. Using the same data set from the example in the text,

$$S = \{(0,0),(2,5),(3,1),(6,4),(8,10),(10,13),(13,11)\}$$

recompute Table 8.5 with $\alpha = 2$ and $\beta = 1$.

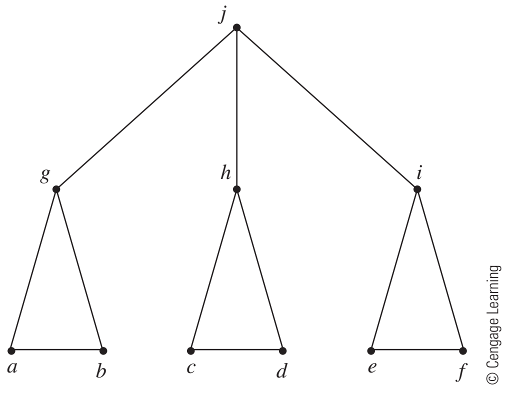
6. Consider the data set $S = \{(0,0),(2,9),(4,7),(6,10),(8,20)\}$. Using $\alpha = 5$ and $\beta = 1$, determine the best piecewise linear function for S.

7. Write computer software that finds the best piecewise linear function given a data set S along with α and β.

8. In the text for this section, there is the sentence "When G is bipartite with bipartition $\langle A, B\rangle$, it is clear that no matching can be bigger than $|A|$, and no matching can be bigger than $|B|$." Explain why this is true.

9. Find a maximum matching in the graph in Figure 8.21. How many edges are in the maximum matching? Now suppose we add the edge bh to the graph. Can you find a larger matching?

Figure 8.21
Graph for Problem 9



10. A basketball coach needs to find a starting lineup for her team. There are five positions that must be filled: point guard (1), shooting guard (2), swing (3), power forward (4), and center (5). Given the data in Table 8.7, create a graph model and use it to find a feasible starting lineup. What changes if the coach decides she can't play Hermione in position 3?

11. Will graphs formed with the procedure used to make the one in Figure 8.18 always be bipartite, regardless of the data? Why or why not?

Table 8.7 Positions players can play

Alice	Bonnie	Courtney	Deb	Ellen	Fay	Gladys	Hermione
1, 2	1	1, 2	3, 4, 5	2	1	3, 4	2, 3

© Cengage Learning

12. Considering the values below, determine whether there is a 0–1 matrix with m rows and n columns, with row sums r_i and column sums s_j. If there is such a matrix, write it down.

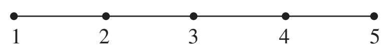
$$\begin{gathered} m = 3, n = 6 \\ r_1 = 4, r_2 = 2, r_3 = 3 \\ s_1 = 2, s_2 = 2, s_3 = 1, s_4 = 0, s_5 = 3, s_6 = 1 \end{gathered} \tag{8.9}$$

13. Considering the following values, determine whether there is a 0–1 matrix with m rows and n columns, with row sums r_i and column sums s_j. If there is such a matrix, write it down.

$$\begin{gathered} m = 3, n = 5 \\ r_1 = 4, r_2 = 2, r_3 = 3 \\ s_1 = 3, s_2 = 0, s_3 = 3, s_4 = 0, s_5 = 3 \end{gathered} \tag{8.10}$$

14. Explain, in your own words, why a maximum-flow algorithm can solve the matrix problem from this section.

15. A path on n vertices, P_n, is a graph with vertices that can be labeled $v_1, v_2, v_3, \ldots, v_n$, so that there is an edge between v_1 and v_2, between v_2 and v_3, between v_3 and v_4, …, and between v_{n-1} and v_n. For example, the graph P_5 appears in Figure 8.22. Compute $\beta(P_5)$. Compute $\beta(P_6)$. Compute $\beta(P_n)$ (your answer should be a function of n).

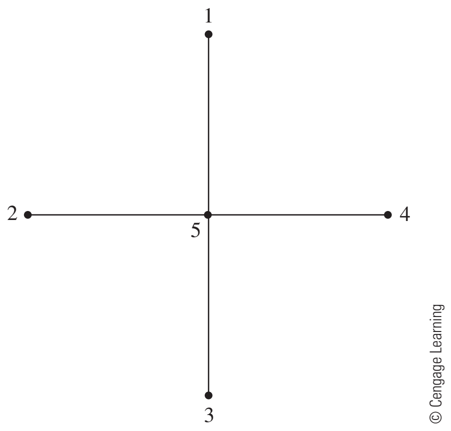


© Cengage Learning

Figure 8.22
P_5

16. Here we consider the *weighted* vertex cover problem. Suppose the graph in Figure 8.23 represents an instance of vertex cover in which the cost of having vertex i in S is $w(i) = (i - 2)^2 + 1$ for $i = 1, 2, 3, 4, 5$. For example, if v_4 is in S, we must use $w(4) = (4 - 2)^2 + 1 = 5$ units of our resource. Now, rather than minimizing the *number* of vertices in S, we seek a solution that minimizes the total amount of resource used $\sum_{i \in S} w(i)$. Using our analogy of soldiers guarding intersections, you can think of $w(i)$ as a description of the number of soldiers needed to guard intersection i. Given the graph in Figure 8.23 and the weighting function $w(i) = (i - 2)^2 + 1$, find a minimum-cost weighted vertex cover.

Figure 8.23
Graph for Problem 16



8.3 PROJECTS

1. Investigate a social network that is of interest to you. Carefully define what the vertices represent and what the edges represent. Are there any new modeling techniques that you had to employ?

2. Write a computer program that takes integers m, n, r_i for $1 \leq i \leq m$ and s_j for $1 \leq j \leq n$ as input and that either outputs a 0–1 matrix with m rows and n columns with row sums r_i and column sums s_j, or says that no such matrix can exist (some programming experience required).

3. Given a graph $G = (V(G), E(G))$, consider the following strategy for finding a minimum vertex cover in a graph.

   Step 0: Start with $S = \emptyset$.

   Step 1: Find a vertex v of maximum degree (one that has a greatest number of incident edges). Add this vertex to S.

   Step 2: Delete v and all its incident edges from G. If G is now edgeless, stop. Otherwise, repeat Steps 1 and 2.

   Either show that this strategy always finds a minimum vertex cover for G or find a graph for which it fails to do so.

4. Consider a modification of the softball manager's problem where we are interested in the *best* starting lineup. How can our mathematical model be modified to solve this problem? What new techniques are needed to solve models of this type?

8.3 Further Reading

Ahuja, R., T. Magnanti, & J. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Englewood Cliffs, NJ: Prentice Hall, 1993.

Huber, M., & S. Horton. "How Ken Griffey, Jr. is like Kevin Bacon or, degrees of separation in baseball." *Journal of Recreational Mathematics*, 33 (2006): 194–203.

Wilf, H. *Algorithms and Complexity*, 2nd ed. Natick, MA: A. K. Peters, 2002.

8.4 Using Graph Models to Solve Problems

In Section 8.3 we saw several real-world problems that can be expressed as problems on graphs. For example, we can solve instances of the Bacon number problem by finding the distance between two vertices in a graph. We also saw that the data-fitting problem could be transformed into a shortest-path problem. We learned that the softball manager's problem and a 0–1 matrix problem could be solved using maximum flows. Finally, we established a relationship between a problem about stationing police on street corners and finding vertex covers in graphs.

In this section we will learn about some simple ways to solve instances of some of these graph problems. This is part of an approach to problem solving similar to our approach in Chapter 2. We start with a real problem, often expressed in nonmathematical terms. Using the insight that comes from practice, experience, and creativity, we recognize that the problem can be expressed in mathematical terms—in these cases, using graphs. Often the resulting graph problem is familiar to us. If so, we solve the problem and then translate the solution back to the language and setting of our original problem.

Solving Shortest-Path Problems

Given a graph $G = (V(G), E(G))$ and a pair of vertices u and v in $V(G)$, we can define the *distance* from u to v as the smallest number of edges in a path from u to v. We use the notation $d(u, v)$ to denote this distance. The *distance problem*, then, is to compute $d(u, v)$ given a graph G and two specific vertices u and v.

Rather than considering methods to solve the distance problem, we will consider a generalization of the problem. Let c_{ij} denote the *length* of the edge ij. Now we can define the length of a shortest path from u to v in a graph to be the minimum (over all possible paths) of the sum of the lengths of the edges in a path from u to v. Given a graph G, edge lengths c_{ij} for each edge $ij \in E(G)$, and two specific vertices u and v, the *shortest-path problem* is to compute the length of a shortest path from u to v in G. It is easy to observe that the distance problem from the previous paragraph is a special case of the shortest-path problem—namely, the case where each $c_{ij} = 1$. Accordingly, we will focus on a technique to solve shortest-path problems, because such a technique will also solve distance problems.

For example, consider the graph in Figure 8.24. Ignoring the edge lengths printed on the figure, it is easy to see that $d(u, y) = 2$ because there is a path of two edges from vertex u to vertex y. However, when the edge lengths are considered, the shortest path from u to y in G goes through vertices v and w and has total length $1 + 2 + 3 = 6$. Thus $d(i, j)$

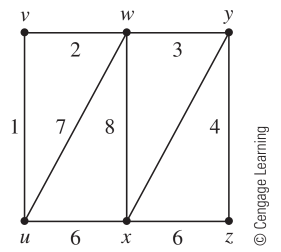


Figure 8.24
Example graph for shortest path

considers only the number of edges between i and j, whereas shortest paths consider edge lengths.

Shortest-path problems have a simple structure that enables us to solve them with very intuitive methods. Consider the following physical analogy. We are given a graph with two specific vertices u and v, as well as edge lengths for each edge. We seek a shortest path from vertex u to vertex v. We'll make our model entirely out of string. We tie the string together with knots that don't slip. The knots correspond to the vertices, and the strings are the edges. Each string is equal in length to the weight (distance) on the edge it represents (plus enough extra length to tie the knots). Of course, if the distances represented in the graph are in miles, we might build our string model scaling, say, miles to inches. Now, to find the length of a shortest path, we hold the two knots representing u and v and pull them apart until some of the strings are tight. Now the length of a shortest path in our original graph corresponds to the distance between the two knots. Furthermore, this model shows which path(s) are shortest (those whose strings are tight). The loose strings are said to have "slack," which means that even if they were slightly shorter, the length of a shortest path would not change.

The intuition of this analogy underlies the algorithm we will use to solve shortest-path problems. Note that the procedure we will present here works to find a shortest path in any graph satisfying $c_{ij} \geq 0$ for all $ij \in E(G)$. The following procedure will find the length of a shortest path in a graph G from vertex s to vertex t, given nonnegative edge lengths c_{ij}.

Dijkstra's Shortest-Path Algorithm

Input A graph $G = (V(G), E(G))$ with a source vertex s and a sink vertex t and nonnegative edge lengths c_{ij} for each edge $ij \in E(G)$.

Output The length of a shortest path from s to t in G.

Step 0 Start with temporary labels L on each vertex as follows: $L(s) = 0$ and $L(i) = \infty$ for all vertices except s.

Step 1 Find the vertex with the smallest temporary label (if there's a tie, pick one at random). Make that label *permanent* meaning it will never change.

Step 2 For every vertex j without a permanent label that is adjacent to a vertex with a permanent label, compute a new temporary label as follows: $L(j) = \min\{L(i) + c_{ij}\}$, where we minimize over all vertices i with a *permanent* label. Repeat Steps 1 and 2 until all vertices have permanent labels.

This algorithm was formulated by the Dutch computer scientist Edsger Dijkstra (1930–2002). When Dijkstra's Algorithm stops, all vertices have permament labels, and each label $L(j)$ is the length of a shortest path from s to j. We now demonstrate Dijkstra's Algorithm on the graph in Figure 8.24.

Let's start with some notation. Let $L(V) = (L(u), L(v), L(w), L(x), L(y), L(z))$ be the current labels on the vertices of the graph in Figure 8.24, listed in the specified order. We will also add an asterisk to any label we have made permanent. Thus $L(V) = (0^*, 1^*, 3^*, 6, 6, \infty)$ would mean the labels on vertices u, v, w, x, y, and z are $0, 1, 3, 6, 6$, and ∞, respectively, and that the labels on u, v, and w are permanent.

Now we are ready to execute Dijkstra's Algorithm to find the shortest distances in G from u to every other vertex. First, in Step 0 we initialize the labels $L(V) = (0, \infty, \infty, \infty, \infty, \infty)$. Next, in Step 1 we make the smallest label permanent, so

$L(V) = (0^*, \infty, \infty, \infty, \infty, \infty)$. In Step 2 we compute a new temporary label for each vertex without a permanent label that is adjacent to one with a permanent label. The only vertex with a permanent label is u, and it is adjacent to v, w, and x. Now we can compute $L(v) = \min\{L(i) + c_{iv}\} = L(u) + c_{uv} = 0 + 1 = 1$. Similarly, $L(w) = \min\{L(i) + c_{iw}\} = L(u) + c_{uw} = 0+7 = 7$ and $L(x) = \min\{L(i)+c_{ix}\} = L(u)+c_{ux} = 0+6 = 6$. Accordingly, our new label list is $L(V) = (0^*, 1, 7, 6, \infty, \infty)$.

Now we repeat Steps 1 and 2. First we examine the current $L(V)$ and note that the smallest temporary label is $L(v) = 1$. Therefore, we make label $L(v)$ permanent and update our label list accordingly: $L(V) = (0^*, 1^*, 7, 6, \infty, \infty)$. In Step 2, there are two vertices with permanent labels to consider: u and v. The situation at x will not change; $L(x)$ will remain at 6. However, there are now two cases to consider as we recompute the label at w: $L(w) = \min\{L(i) + c_{iw}\} = \min\{L(u) + c_{uw}, L(v) + c_{vw}\} = \min\{0 + 7, 1 + 2\} = \min\{7, 3\} = 3$. Our new label list is $L(V) = (0^*, 1^*, 3, 6, \infty, \infty)$.

We continue repeating Steps 1 and 2. A computer scientist would say that we are now on "iteration three" of the main step of the algorithm. First, we add a permanent label to our list, obtaining $L(V) = (0^*, 1^*, 3^*, 6, \infty, \infty)$. After Step 2 we arrive at $L(V) = (0^*, 1^*, 3^*, 6, 6, \infty)$. In iteration four, we pick one of the vertices with $L(i) = 6$; our algorithm says to break ties at random, so say we pick y to get the permanent label. This results in the label list $L(V) = (0^*, 1^*, 3^*, 6, 6^*, \infty)$. After the next step, we obtain $L(V) = (0^*, 1^*, 3^*, 6, 6^*, 10)$, followed (in iteration five) by $L(V) = (0^*, 1^*, 3^*, 6^*, 6^*, 10)$ and then $L(V) = (0^*, 1^*, 3^*, 6^*, 6^*, 10)$. Note that Step 2 of iteration five resulted in no change to $L(V)$ because the path uxz doesn't improve on the other path $uvwyz$ from u to z that we identified in the previous iteration. Finally, in iteration six, we make the last temporary label permanent. After that, there's nothing left to do, and the algorithm stops with $L(V) = (0^*, 1^*, 3^*, 6^*, 6^*, 10^*)$. This list gives the lengths of shortest paths from u to every vertex in the graph.

Solving Maximum-Flow Problems

We saw in Section 8.3 that both the data-fitting problem and the 0–1 matrix problem can be solved by finding a maximum flow in a related graph. We will see in Section 8.4 that the softball manager's problem can also be solved this way. In fact, many practical problems can be modified as maximum flows in graphs. In this section we present a simple technique for finding a maximum flow in a graph.

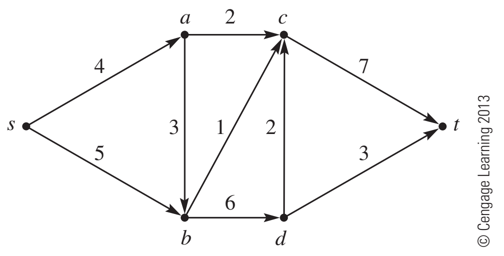
So far, all of the graphs we have considered have vertices and edges, but the edges don't have any notion of direction. That is, an edge goes between some pair of vertices rather than specifically from one vertex *to* another. In this section, we will consider directed graphs instead. Directed graphs are just like their undirected siblings, except that each edge has a specific direction associated with it. We will use the term **arc** to refer to a directed edge. More formally, we can define a **directed graph** $G = (V(G), A(G))$ as two sets: a **vertex set** $V(G)$ and an **arc set** $A(G)$. Each element of $A(G)$ is an **ordered pair** of elements of $V(G)$. We will use the notation (i, j) to represent an arc oriented from i to j. Note that undirected graphs can be transformed into directed graphs in many ways. Given an undirected graph, an orientation for each edge in $E(G)$ can be selected. Alternatively, each edge $ij \in E(G)$ can be transformed into two arcs; one from i to j and one from j to i.

Now we return to the problem of solving maximum-flow problems in (directed) graphs. Just as in shortest-path problems, there are many algorithms for finding maximum flows in graphs. These procedures go from very simple and intuitive algorithms to much more detailed methods that can have theoretical or computational advantages over other approaches. Here we will focus on the former. This chapter concludes with references where the interested reader can explore more sophisticated algorithms.

Given a directed graph $G = (V(G), A(G))$, two specific vertices s and t, and a finite flow capacity u_{ij} for each arc (i, j), we can use a very simple technique for finding a maximum flow from s to t in G. We'll demonstrate the technique with an example first, and then we'll describe in more general terms what we did.

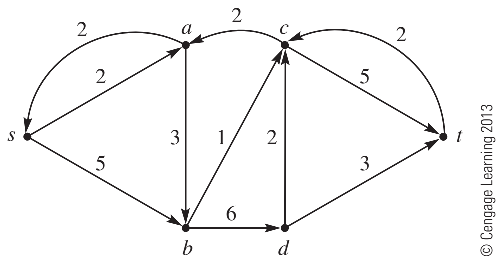
Before we start, we need to define the concept of a directed path. A **directed path** is a path that respects the orientation of arcs in the path. For example, consider the graph in Figure 8.25. This directed graph has vertex set $V(G) = \{s, a, b, c, d, t\}$ and arc set $A(G) = \{sa, sb, ab, ac, bc, bd, ct, dc, dt\}$. Note that $ab \in A(G)$ but $ba \notin A(G)$. Accordingly, $s - a - b - d - t$ is a directed path from s to t, but $s - b - a - c - t$ is not (because $ba \notin A(G)$). The graph also has arc flow capacities annotated by each arc.

Figure 8.25
Example graph for maximum flow



Now we are ready to begin our example. Consider again the graph in Figure 8.25. We start by finding a directed path from s to t. There are several choices, but let's say we pick the path $s - a - c - t$. We observe that the *lowest-capacity* arc on this path is ac, which has capacity $u_{ac} = 2$. Accordingly, we create a new graph that represents the remaining capacity, or *residual capacity* of the network after 2 units of flow are pushed along the path $s - a - c - t$. All we do is account for the flow by reducing the remaining capacity by 2 for each arc in the path. Note that this reduces the residual flow on arc ac to zero, so we delete that arc from the next graph. Also note that three new arcs appear; these represent our ability at some future iteration to "back up" or "reverse" flow. The result of these changes appears in Figure 8.26. At this point we have managed to get 2 units of flow from s to t. This completes the first iteration of our maximum-flow algorithm.

Figure 8.26
Example graph for maximum flow, after Iteration 1



Now we are ready to start the second iteration. Again, we look for a directed path from s to t in the residual graph from the previous iteration (Figure 8.26). The directed path $s - b - d - t$ is one possiblility. The smallest arc capacity in the residual graph is $u_{dt} = 3$, so we create another residual graph by reducing all of the capacities along the path. In this iteration we pushed 3 more units of flow from s to t, which brings our total so far to 5 units. The result appears in Figure 8.27, and we have finished iteration 2.

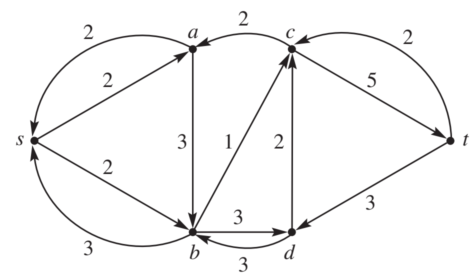


Figure 8.27 Example graph for maximum flow, after Iteration 2

Iteration 3 proceeds like the others. The residual graph in Figure 8.27 has the directed path $s - b - d - c - t$ with a minimum capacity of 2 units of flow. We have now delivered 7 units of flow from s to t. The appropriate modification leads us to Figure 8.28.

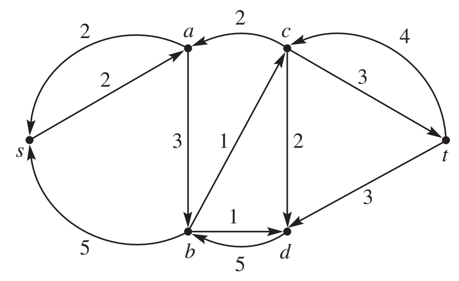


Figure 8.28 Example graph for maximum flow, after Iteration 3

At this point there is only one directed path from s to t; it is $s - a - b - c - t$. Because $u_{bc} = 1$ in Figure 8.28, we add the 1 unit of flow to our previous total, bringing us to 8. After reducing the capacity on each arc in the directed path by 1, we obtain the graph in Figure 8.29.

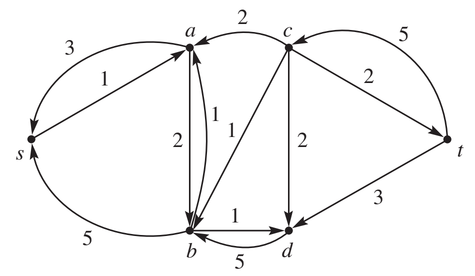


Figure 8.29 Example graph for maximum flow, final residual graph

We continue by looking for a directed path from s to t in Figure 8.29, but there is no such path. Accordingly, we stop and claim that the maximum flow from s to t in our original graph of Figure 8.25 is 8.

Finally, we can generalize our maximum-flow algorithm. But first, we need to be a little more precise in describing our problem.

Maximum-Flow Problem *Given a directed graph $G = (V(G), A(G))$ with a source vertex s and a sink vertex t and a finite flow capacity u_{ij} for each arc $ij \in A(G)$, find the maximum flow in the graph from vertex s to vertex t.*

The algorithm we used above is more formally and more generally described below.

Maximum-Flow Algorithm

Input A directed graph $G = (V(G), A(G))$ with a source vertex s and a sink vertex t and a finite flow capacity u_{ij} for each arc $ij \in A(G)$. Let $u_{ij} = 0$ for all $ij \notin A(G)$.

Output The maximum flow from s to t in G.

Step 0 Set the current flow to zero: $f_c \leftarrow 0$.

Step 1 Find a directed path from s to t in the current graph. If there is no such path, stop. The maximum flow from s to t in G is f_c.

Step 2 Compute $u_{\min}$, the minimum capacity in the current graph of all the arcs in the directed path.

Step 3 For each arc ij in the directed path, update the residual capacity in the current graph: $u_{ij} \leftarrow u_{ij} - u_{\min}$. Also update "reverse" arc capacities: $u_{ji} \leftarrow u_{ji} + u_{\min}$.

Step 4 Set $f_c \leftarrow f_c + u_{\min}$ and return to Step 1.

Now we turn our attention to using the Maximum-Flow Algorithm to solve other problems.

Solving Bipartite Matching Problems Using Maximum Flows In Section 8.3 we learned that the softball manager's problem can be solved by finding a maximum matching in a bipartite graph derived from the problem instance. In this section, we will find that the Maximum-Flow Algorithm from the previous section can be used to find maximum matchings in bipartite graphs. By combining these two, we will be able to solve the softball manager's problem using our maximum-flow algorithm. This is an exciting aspect of mathematical modeling: by looking at problems the right way, we can often solve them using techniques that at first glance seem unrelated.

Recall that a *matching* in a graph G is a subset $S \subseteq E(G)$ such that no two edges in S end at a common vertex. In other words, the edges in S are *independent*. A maximum matching is a matching of largest size. If the graph G is *bipartite*, the maximum-matching problem can be solved using a maximum-flow algorithm. Recall that a graph is bipartite if the vertex set can be partitioned into two sets X and Y such that all of the edges have one end in X and the other end in Y. Figure 8.30 is an example of a bipartite graph. Which edges should we put in S to make $|S|$ as large as possible?

We might start by including x_1y_1 in S. This choice eliminates all other edges incident with x_1 or y_1 from further consideration. Accordingly, we can also include x_2y_3 in S. Now we have two edges in S. Unfortunately, we cannot add more edges to S, at least not without

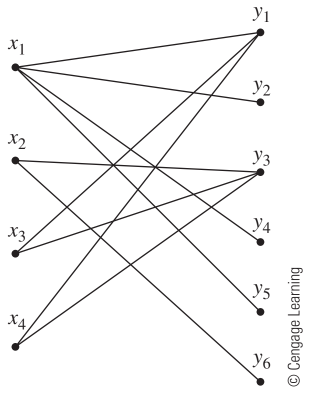


Figure 8.30
Maximum-matching problem

removing some edges that are already in. Can you find a matching larger than size 2 in G? It seems that the initial choices we made were poor in the sense that they blocked too many other options. For this small problem, it was probably easy for you to see how to get a larger matching. In fact you can probably find a maximum matching just by looking at the problem; recall from Problem 8 in Section 8.3 that no matching in this graph can have more than $\min\{|X|, |Y|\} = 4$ edges.

A larger instance of this problem (that is, a larger graph) might be more difficult to solve by inspection. As the title of this section suggests, there is a way to solve bipartite matching problems using a maximum-flow algorithm. Here's how. From the (undirected) instance bipartite graph with bipartition $V(G) = \langle X, Y \rangle$, we orient each edge so that it goes from X to Y. We allow each of these arcs to have an unlimited capacity. Then we create two new vertices s and t. Finally, we create arcs from s to every vertex in X, and arcs from every vertex in Y to t. Each of these arcs has a capacity of 1. Figure 8.31 demonstrates the idea. Now we find the maximum flow in the resulting directed graph from s to t, and that maximum flow will equal the size of a maximum matching in the original graph.

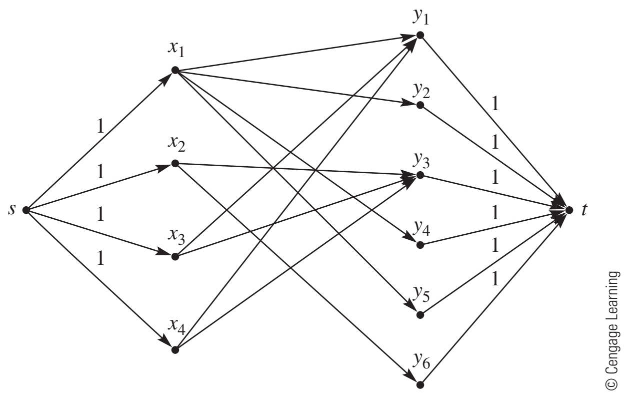


Figure 8.31
Matching problem expressed as a maximum-flow problem

One important feature that shortest paths and maximum flows have in common is that they can be found *efficiently* by a computer. Unfortunately, not all graph problems have this property. We now consider some problems that can be described just as easily but seem to be much harder to solve by algorithmic means. In Section 8.3, we learned about one such problem: vertex cover. Here we introduce one more, and then we make a surprising observation about these two problems.

Consider a department store that delivers goods to customers. Every day, a delivery crew loads a truck at the company's warehouse with the day's orders. Then the truck is driven to each customer and returned to the warehouse at the end of the day. The delivery crew supervisor needs to decide on a route that visits each customer once and then returns to the starting point. The set of customers who require a delivery changes each day, so the supervisor has a new instance to solve every day. We can state this problem as follows.

Traveling Salesman Problem *A salesman must visit every location in a list and return to the starting location. In what order should the salesman visit the locations in the list to minimize the distance traveled?*

It is easy to see that this problem can be modeled with a graph. First we note that this problem is usually modeled with a graph that is *complete*. A complete graph has an edge between every pair of vertices. We also need a cost c_{ij} for every edge. This number represents the cost of traveling on the edge between vertex i and vertex j. You can think of this cost as distance, or as time, or as an actual monetary cost. A list of all of the locations, in any order, defines a **tour**. The **cost** of a tour is the sum of the edge costs on that tour. For example, consider the graph in Figure 8.32. Let's say we start at vertex a (it turns out not to matter where we start). It seems reasonable to go first to vertex b, because the cost of doing so is small; only 1 unit. From there, the edge to c looks attractive. So far we have paid $1 + 2 = 3$ units to get this far. From c, we can go to d, and then to e (the only vertex we have not yet visited). Now our cumulative cost is $1 + 2 + 4 + 8 = 15$. Finally, we have to return to the starting point, which adds 3 to our cost, bringing the total to 18.

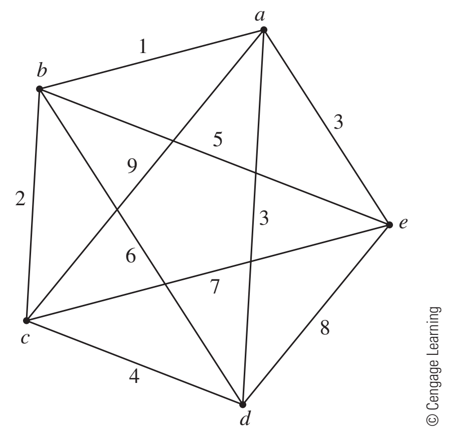


Figure 8.32
Example graph for traveling salesman problem

It turns out that this is not the optimal tour. The tour $a - d - c - b - e - a$ has a cost of 17 (the reader should verify this). Notice that this improved tour does not use the edge from a to b that seemed so attractive. This example shows that the greedy choice of picking the cheapest edge ($a - b$ in this case) isn't always the best thing when looking for an optimal solution. Problems (like the Traveling Salesman Problem) that exhibit this "greedy choice might be bad" property are often problems for which good algorithms are hard to find or unknown. Now we can restate the problem in graph theoretic terms.

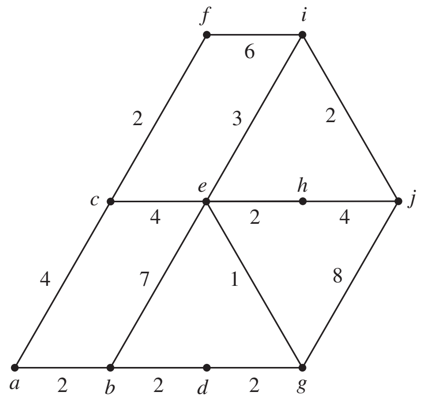
Traveling Salesman Problem (restated) *Given a complete graph $G = (V(G), E(G))$ and a cost c_{ij} for each edge in $E(G)$, find a tour of minimum cost.*

Finally, it is important to understand that this problem is different from the shortest-path problems that we have considered. It may seem as though some of the same techniques can be used, but it turns out that this isn't the case. Instances of the shortest-path problem can always be solved efficiently, whereas the Traveling Salesman Problem (and the vertex cover problem) are not so easy to deal with. In fact, this section is about describing *algorithms* to solve various graph problems. It might come as a suprise to learn that there is *no* known efficient algorithm for solving either the Traveling Salesman Problem or the vertex cover problem. For smaller instances, it is often possible to enumerate—that is, to check all of the possible solutions. Unfortunately, that strategy is doomed to take too long for large instances. We will elaborate on this idea in the next section.

8.4 PROBLEMS

1. Find a shortest path from node a to node j in the graph in Figure 8.33 with edge weights shown on the graph.

Figure 8.33
Graph for Problem 1



2. A small suburban city is experimenting with a new way to keep its main park clean. From May through October, a crew is needed every day to pick up and remove trash. Rather than contracting with one company for the entire period, the city manager takes bids from firms on its website. Firms submit bids with dates of work and the fee for which the firm is willing to do the clean-up work. On some published date, the city manager reviews all of the bids submitted and decides which ones to accept. For example, a firm might make a bid to perform the work June 7–20 for \$1200. Another firm might bid to work June 5–15 for \$1000. Because the bids overlap, they cannot both be accepted. How can the city manager use Dijkstra's Algorithm to choose what bids to accept to minimize costs? What assumptions do you need to make?

3. Use our maximum-flow algorithm to find the maximum flow from s to t in the graph of Figure 8.31.

4. Explain why the procedure for using a maximum-flow algorithm to find the size of a maximum matching in a bipartite graph works. Can it be used to find a maximum matching (as opposed to the size of one)? Can it be used for graphs that are not bipartite? Why or why not?

5. In the sport of orienteering, contestants (''orienteers'') are given a list of locations on a map (''points'') that they need to visit. Orienteering courses are typically set up in natural areas such as forests or parks. The goal is to visit each point and return to the starting location as quickly as possible. Suppose that an orienteer's estimates of the time it will take her between each pair of points are given in Table 8.8. Find the tour that minimizes the orienteer's time to negotiate the course.

Table 8.8 Orienteer's time estimates between points (in minutes)

	start	*a*	*b*	*c*	*d*
start	—	15	17	21	31
a	14	—	19	14	17
b	27	22	—	18	19
c	16	19	26	—	15
d	18	22	23	29	—

© Cengage Learning

6. Does Dijkstra's Algorithm work when there might be arcs with *negative* weights?

8.4 Further Reading

Ahuja, R., T. Magnanti, & J. Orlin. *Network Flows: Theory, Algorithms, and Applications.* Englewood Cliffs, NJ: Prentice Hall, 1993.

Buckley, F., & F. Harary. *Distance in Graphs.* NY: Addison-Wesley, 1990.

Lawler, E., J. Lenstra, A. R. Kan, & D. Shmoys, eds. *The Traveling Salesman Problem.* NY: Wiley, 1985.

Skiena, S., & S. Pemmaraju. *Computational Discrete Mathematics: Combinatorics and Graph Theory with Mathematica.* Cambridge: Cambridge University Press, 2003.

Winston, W., & M. Venkataramanan. *Introduction to Mathematical Programming: Applications and Algorithms, Volume 1,* 4th ed. Belmont, CA: Brooks-Cole, 2003.

8.5 Connections to Mathematical Programming

In the previous chapter, we learned about modeling decision problems with linear programs and then using the Simplex Method to solve the associated linear program. In this section, we will consider how linear programming and integer programming can be used to model some of the problems presented in the previous section.

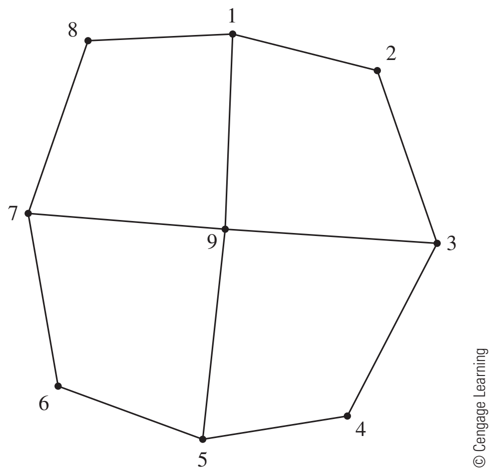


Figure 8.34
Example graph

Vertex Cover

Recall the vertex cover problem discussed in Section 8.3. In words, we are looking for a set S that is a subset of $V(G)$ such that every edge in the graph is incident with a member of S, and we want $|S|$ to be as small as possible. We have learned that this can be a hard problem to solve, but sometimes integer programming can help.

Consider the graph in Figure 8.34. Let's try to find a minimum vertex cover. Each vertex we put in S in a sense reduces the number of edges that are still uncovered. Perhaps we should start by being *greedy*—that is, by looking for a vertex of highest degree (the degree of a vertex is the number of edges that are incident with it). This is a greedy choice because it gives us the biggest possible return for our "investment" of adding one vertex to S. Vertex 9 is the only vertex that allows us to cover four edges with one vertex, so this seems like a great start. Unfortunately it turns out that this choice isn't optimal; the only minimum vertex cover for this graph is $S = \{1, 3, 5, 7\}$, and our initial choice, vertex 9, is conspicuously absent.

Our small example could certainly be solved by trial and error, but we will see shortly that trial and error can quickly become a completely useless solution strategy for larger graphs: there are simply too many possible solutions to check. Integer programming can sometimes provide an alternative.

An Integer Programming Model of the Vertex Cover Problem The key to modeling with integer programs is the same as the key to other kinds of modeling. One of the most important steps in any modeling problem is to define your variables. Let's do that now. We can let

$$x_i = \begin{cases} 1 & \text{if } i \in S \\ 0 & \text{otherwise} \end{cases} \tag{8.11}$$

The variable x_i is sometimes called a **decision variable** because each feasible value you choose to assign to x_i represents a decision you might make regarding the vertices of the graph in Figure 8.34 *vis-à-vis* their membership in S. For example, setting $x_5 = 1$ represents including vertex 5 in S, and $x_7 = 0$ represents excluding vertex 7 from S. If we define the vector $x = \langle x_1, x_2, \ldots, x_9 \rangle$, we see that any binary string of length 9 can be thought of as a choice of vertices to include in a vertex cover S.

As an aside, you might think of how many possible decisions there are at this point. Each x_i must take on one of two possible values. Thus there are $2^9 = 512$ possibilities. This isn't too bad, but if we try this problem on a somewhat larger graph of 100 verticies, we would have to consider $2^{100} = 1{,}267{,}650{,}600{,}228{,}229{,}401{,}496{,}703{,}205{,}376$ different possible choices for which vertices go in S! Clearly, any solution strategy that checks all of these possibilities individually, *even with a very fast computer*, is doomed to take too much time to be useful. (You might enjoy doing a little computation to check this out. See Problem 3 at the end of this section.)

Of course, some choices are better than others! In fact, there are two big issues to consider when we make our choice. First, our choice must be *feasible*. That is, it must satisfy the requirement that every edge be incident with a vertex that is in S. We really have to check all of the edges to make sure this requirement, which is also called a *constraint*, is satisfied. Just saying it this way helps us understand how to write the constraint in the language of integer programming. Consider the edge incident with vertex 4 and vertex 5. The presence of this edge requires that either x_4 or x_5 (or both) equal 1. One way to write this algebraically is

$$(1 - x_4)(1 - x_5) = 0 \tag{8.12}$$

You should take a moment to convince yourself that this equality is exactly equivalent to the previous sentence. In other words, verify that (8.12) is satisfied if and only if $x_4 = 1$ or $x_5 = 1$ (or both).

Unfortunately, it turns out that (8.12) isn't the best way to write this idea down. In general, software that solves integer programs will perform better if constraints are written as *linear* inequalities (equalities are okay, too). The expression (8.12) isn't linear because we end up multiplying two decision variables. A much better formulation is to express (8.12) this way:

$$x_4 + x_5 \geq 1 \tag{8.13}$$

Again you can verify that (8.13) is satisfied exactly when $x_4 = 1$ or $x_5 = 1$ (or both).

Certainly the same line of reasoning applies to all 12 edges in the graph. The following is a complete list of these constraints

$$\begin{aligned}
x_1 + x_2 &\geq 1 \\
x_2 + x_3 &\geq 1 \\
x_3 + x_4 &\geq 1 \\
x_4 + x_5 &\geq 1 \\
x_5 + x_6 &\geq 1 \\
x_6 + x_7 &\geq 1 \\
x_7 + x_8 &\geq 1 \\
x_8 + x_1 &\geq 1 \\
x_1 + x_9 &\geq 1
\end{aligned} \tag{8.14}$$

$$x_3 + x_9 \geq 1$$
$$x_5 + x_9 \geq 1$$
$$x_7 + x_9 \geq 1$$

Rather than write down this long list, we can write a single, more general expression that means the same thing:

$$x_i + x_j \geq 1 \quad \forall ij \in E(G) \tag{8.15}$$

The inequality (8.15) captures exactly the 12 inequalities in (8.14).

Because we defined the decision variables so that they always take values of 0 or 1, we need a constraint in the integer program to account for that. We can write it this way:

$$x_i \in \{0, 1\} \quad \forall i \in V(G) \tag{8.16}$$

This forces each x_i to be either 0 or 1, as specified by our model.

Now we have taken care of all of the constraints. A choice of $x = \langle x_1, x_2, \ldots, x_9 \rangle$ that satisfies all of the constraints in (8.15) and (8.16) is called feasible or a **feasible point**. Now we simply seek, from all of the feasible points, the *best* one (or perhaps *a* best one). We need some measure of "goodness" to decide which choice for x is best. In this case, as we already said, we want the solution that minimizes $|S|$. In terms of our integer programming formulation, we want to minimize $\sum_{i \in V(G)} x_i$. This is called the **objective function**.

Finally, we can collect our objective function and our constraints (8.15) and (8.16) in one place to obtain a general way to write down *any* vertex cover problem as an integer program

$$\text{Minimize } z = \sum_{i \in V(G)} x_i$$

subject to (8.17)

$$x_i + x_j \geq 1 \quad \forall ij \in E(G)$$
$$x_i \in \{0, 1\} \quad \forall i \in V(G)$$

Note that (8.17), without the last line $x_i \in \{0, 1\} \;\; \forall i \in V(G)$, is a *linear* program instead of an *integer* program. It turns out that solving integer programs often involves first solving the linear program obtained by dropping the integrality requirement. There is an extensive body of knowledge about how to formulate and solve linear and integer programming problems, and about how to interpret solution results. The list of readings at the end of the chapter can provide the interested reader with a few places to start.

There is one more point to consider about using integer programming to solve large instances of the vertex cover problem. The formulation 8.17 shows that we can at least formulate any vertex cover problem as an integer program. Unfortunately, it is not known whether there is a fast procedure for solving all integer programs. *Computational complexity* is the branch of theoretical computer science that considers issues such as this. The references at the end of the chapter provide additional information.

Maximum Flows

Now we reconsider the maximum-flow problem on directed graphs. This problem was defined in the previous section. We are given a directed graph $G = (V(G), A(G))$, a source vertex s, a sink vertex t, and a flow capacity u_{ij} for every arc $ij \in A(G)$. We start by defining variables to represent flow. We let x_{ij} represent the flow from vertex i to vertex j.

There are several types of constraints to consider. First, we will allow only nonnegative flows, so

$$x_{ij} \geq 0 \qquad \forall ij \in A(G) \tag{8.18}$$

Recall that the symbol $\forall$ means "for all"; thus, the constraint $x_{ij} \geq 0$ applies for every arc ij in the graph's arc set $A(G)$.

We also know that flow on each arc is limited to the capacity, or upper bound, on that arc. The following constraint captures that idea

$$x_{ij} \leq u_{ij} \qquad \forall ij \in A(G) \tag{8.19}$$

Before we consider the last type of constraint, we need to make a key observation. At every vertex in the graph except for s and t, flow is *conserved*. That is, the flow *in* is equal to the flow *out* at every vertex (except s and t). Now we are ready to write down the flow balance constraint

$$\sum_i x_{ij} = \sum_k x_{jk} \qquad \forall j \in V(G) - \{s,t\} \tag{8.20}$$

The set $V(G) - \{s,t\}$ is just $V(G)$ with s and t removed; in this case, $V(G) - \{s,t\} = \{a, b, c, d\}$.

Now we turn our attention to the objective function. We seek to maximize flow from s to t. Because flow is conserved everywhere except at s and t, we observe that any flow going out of s must eventually find its way to t. That is, the quantity we want to maximize is the sum of all the flow out of s (we could instead maximize the sum of all the flow to t—the result would be the same). Therefore, our objective function is

$$\text{Maximize} \quad \sum_j x_{sj} \tag{8.21}$$

Combining (8.18), (8.19), (8.20), and (8.21), we can write any maximum-flow problem as a linear program as follows:

$$\text{Maximize } z = \sum_j x_{sj}$$

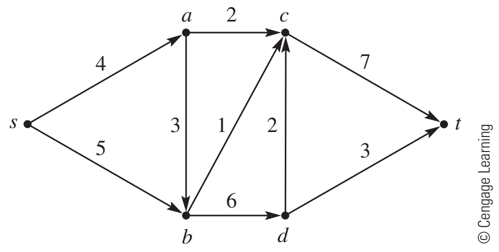
subject to

$$\begin{aligned} \sum_i x_{ij} = \sum_k x_{jk} &\qquad \forall j \in V(G) - \{s,t\} \\ x_{ij} \leq u_{ij} &\qquad \forall ij \in A(G) \\ x_{ij} \geq 0 &\qquad \forall ij \in A(G) \end{aligned} \tag{8.22}$$

When the linear program (8.22) is solved, the resulting flow x is a maximum flow in G, and the largest amount of flow that can go from s to t is $\sum_j x_{sj}$.

Recall our maximum-flow example problem from the previous section (repeated in Figure 8.35 below for convenience).

Figure 8.35
Example graph for maximum flow

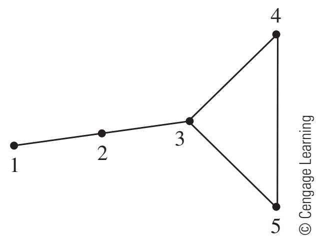


First note that constraints of type (8.18) require flow to be nonnegative on each arc. Thus, $x_{ac} \geq 0$ is one (of nine) such constraints for this instance. Constraints of type (8.19) require that flow satisfy the upper bound given. The nine constraints of this type for this example include $x_{ac} \leq 2$ and $x_{ct} \leq 7$. Constraints of type (8.20) are for flow balance. There are four such constraints because there are four vertices, not including s and t. At vertex c, the flow balance constraint is $x_{ac} + x_{bc} + x_{dc} = x_{ct}$. The objective function for this instance is to maximize $x_{sa} + x_{sb}$.

8.5 PROBLEMS

1. Consider the graph shown in Figure 8.36.

Figure 8.36
Graph *G*



a. Find a minimum vertex cover in the graph in Figure 8.36.

b. Formulate an integer program to find a minimum vertex cover in the graph in Figure 8.36.

c. Solve the integer program in part (b) with computer software.

2. Consider again the graph in Figure 8.36. Now suppose that the cost of placing a vertex in S varies. Suppose the cost of placing vertex i in S is $g(i) = (-i^2 + 6i - 5)^3$ for $i \in \{1, 2, 3, 4, 5\}$. Repeat parts (a), (b), and (c) of the previous problem for this new version of the problem. This is an instance of *weighted* vertex cover.

3. Suppose a computer procedure needs to check all of 2^{100} possibilities to solve a problem. Assume the computer can check 1,000,000 possibilities each second.

a. How long will it take the computer to solve this problem this way?

b. Suppose that the computer company comes out with a new computer that operates

1000 times faster than the old model. How does that change the answer to question a? What is the *practical* impact of the new computer on solving the problem this way?

4. Write down the linear program associated with solving maximum flow from s to t in the graph in Figure 8.37.

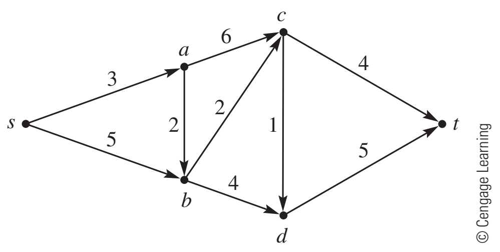


Figure 8.37
Graph for Problem 1

8.5 PROJECT

1. Express the softball manager's problem as a linear or integer program, and solve it with computer software.

8.5 Further Reading

Garey, M., & D. Johnson. *Computers and Intractability.* W. H. Freeman, 1978.
Nemhauser, G., & L. Wolsey. *Interger and Combinatorial Optimization.* New York: Wiley, 1988.

9 Modeling with Decision Theory

Introduction

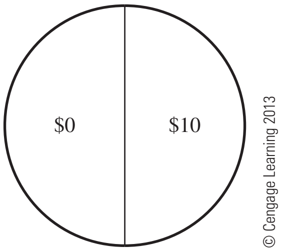
In the mathematical modeling process in Chapter 2, we discussed the importance of assumptions to the models and methods that we might develop. Decision theory, which is also called decision analysis, is a collection of mathematical models and tools developed to assist people in choosing among alternative actions in complex situations involving chance and risk. In many decisions, the options we have are clearly defined, and the information is determined. For example, in Section 7.3 we analyzed the decision of a carpenter who wants to maximize his profit by producing tables and bookcases given constraints on lumber and labor. We assumed that tables and bookcases always took the same amount of resources and always returned the same net profit. Such situations and assumptions are called **deterministic**.

However, in many situations **chance** and **risk** are involved, and we may want to develop models to help us make decisions considering chance, risk, or both. Many of these involve a random factor in addition to the presence of well-defined elements. The presence of the random factor makes these situations inherently nondeterministic. Such situations are called **stochastic** because their subsequent states are determined by both predictable factors and random factors. For example, suppose you are at the fairgrounds and encounter the wheel in Figure 9.1. It costs \$4 to spin. You can play as many games as you like. Assuming the wheel is ''fair,'' should you play the game? If so, how much do you expect to win or lose over the course of the evening?

If the wheel is fair, we would expect to spin a \$0 half the time and \$10 the other half and average \$5 per spin. Since it costs \$4 to play the game and the outcome is either \$0 or \$10, we either lose \$4 or win \$6. That is, our average net value, or expectation E, is

$$E = (0 - \$4)0.5 + (\$10 - \$4)0.5 = \$1.$$

Figure 9.1
Wheel with two outcomes


$0
$10
© Cengage Learning 2013