Shortest-Path Problems

Introduction

Many problems can be modeled using graphs with weights assigned to their edges. As an illustration, consider how an airline system can be modeled.

We set up the basic graph model by representing cities by vertices and flights by edges.

- Problems involving distances can be modeled by assigning distances between cities to the edges.
- Problems involving flight time can be modeled by assigning flight times to edges.
- Problems involving fares can be modeled by assigning fares to the edges. Figure 1 displays three different assignments of weights to the edges of a graph representing distances, flight times, and fares, respectively.

- Graphs that have a number assigned to each edge are called **weighted graphs**.
- Weighted graphs are used to model computer networks.
- Communications costs (such as the monthly cost of leasing a telephone line), the response times of the computers over these lines, or the distance between computers, can all be studied using weighted graphs.
- Figure 2 displays weighted graphs that represent three ways to assign weights to the edges of a graph of a computer network, corresponding to distance, response time, and cost.

Several types of problems involving weighted graphs arise frequently. Determining a path of least length between two vertices in a network is one such problem. To be more specific, let the **length** of a path in a weighted graph be the sum of the weights of the edges of this path. (The reader should note that this use of the term *length* is different from the use of *length* to denote the number of edges in a path in a graph without weights.) The question is: What is a shortest path, that is, a path of least length, between two given vertices? For instance, in the airline system represented by the weighted graph shown in Figure 1, what is a shortest path in air distance between Boston and Los Angeles? What combinations of flights has the smallest total flight time (that is, total time in the air, not including time between flights) between Boston and Los Angeles? What is the cheapest fare between these two cities? In the computer network shown in Figure 2, what is a least expensive set of telephone lines needed to connect the computers in San Francisco with those in New York? Which set of telephone lines gives a fastest response time for communications between San Francisco and New York? Which set of lines has a shortest overall distance?

Another important problem involving weighted graphs asks for a circuit of shortest total length that visits every vertex of a complete graph exactly once. This is the famous *traveling salesperson problem*, which asks for an order in which a salesperson should visit each of the cities on his route exactly once so that he travels the minimum total distance. We will discuss the traveling salesperson problem later in this section.

A Shortest-Path Algorithm

There are several different algorithms that find a shortest path between two vertices in a weighted graph. We will present a greedy algorithm discovered by the Dutch mathematician Edsger Dijkstra in 1959. The version we will describe solves this problem in undirected weighted graphs where all the weights are positive. It is easy to adapt it to solve shortest-path problems in directed graphs

- Before giving a formal presentation of the algorithm, we will give an illustrative example.

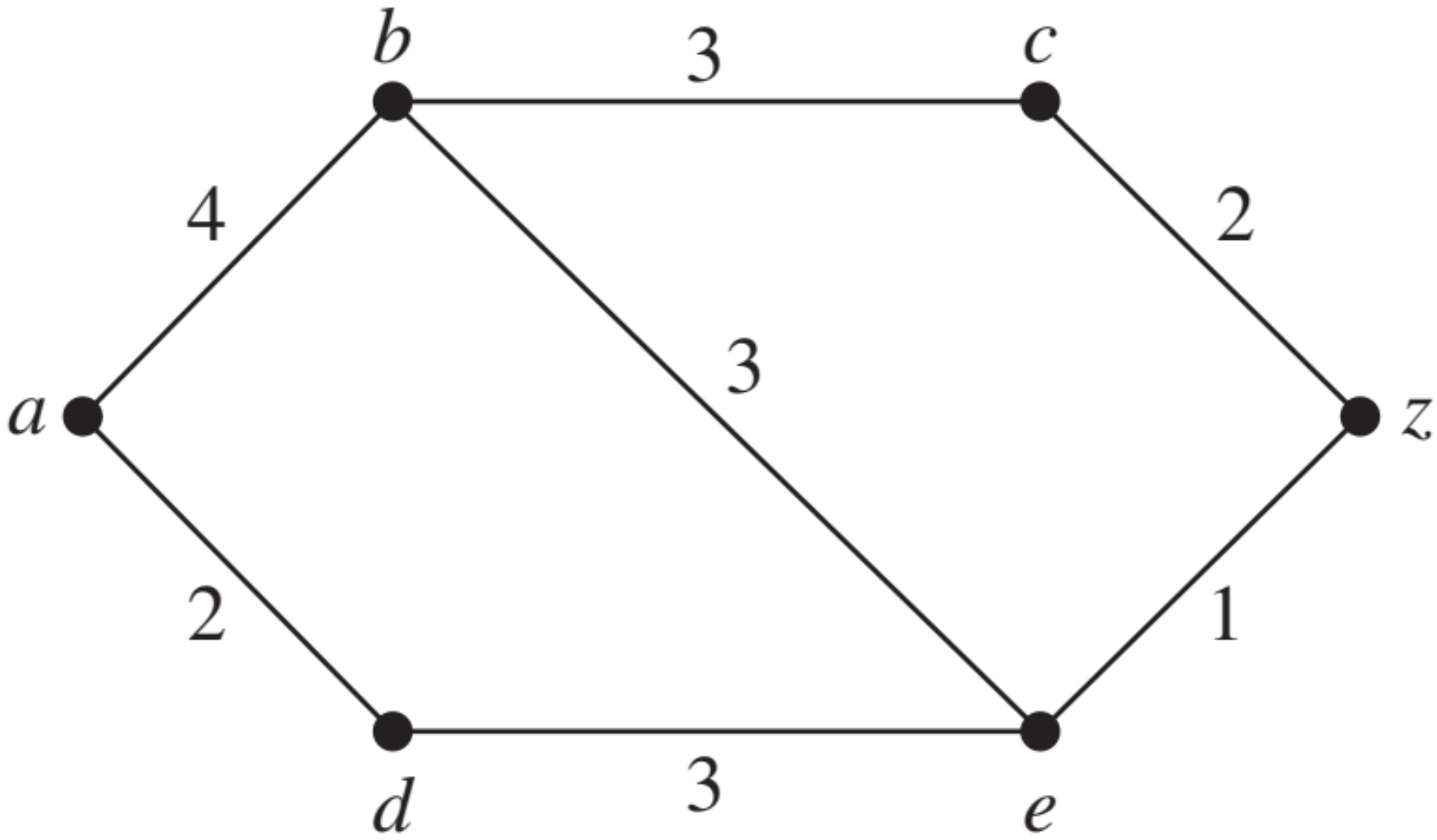


FIGURE 3 **A Weighted Simple Graph.**

EXAMPLE 1

What is the length of a shortest path between a and z in the weighted graph shown in Figure 3?

Solution:

Note: Although a shortest path is easily found by inspection, we will develop some ideas useful in understanding Dijkstra's algorithm. We will solve this problem by finding the length of a shortest path from a to successive vertices, until z is reached.

The only paths starting at a that contain no vertex other than a are formed by adding an edge that has a as one endpoint. These paths have only one edge. They are a, b of length 4 and a, d of length 2. It follows that d is the closest vertex to a, and the shortest path from a to d has length 2.

We can find the second closest vertex by examining all paths that begin with the shortest path from a to a vertex in the set $\{a, d\}$, followed by an edge that has one endpoint in $\{a, d\}$ and its other endpoint not in this set. There are two such paths to consider, a, d, e of length 5 and a, b of length 4. Hence, the second closest vertex to a is b and the shortest path from a to b has length 4.

To find the third closest vertex to a, we need examine only the paths that begin with the shortest path from a to a vertex in the set $\{a, d, b\}$, followed by an edge that has one endpoint in the set $\{a, d, b\}$ and its other endpoint not in this set. There are three such paths, a, b, c of length 7, a, b, e of length 7, and a, d, e of length 5. Because the shortest of these paths is a, d, e,the third closest vertex to a is e and the length of the shortest path from a to e is 5.

To find the fourth closest vertex to a, we need examine only the paths that begin with the shortest path from a to a vertex in the set $\{a, d, b, e\}$, followed by an edge that has one endpoint in the set $\{a, d, b, e\}$ and its other endpoint not in this set. There are two such paths, a, b, c of length 7 and a, d, e, z of length 6. Because the shorter of these paths is a, d, e, z, the fourth closest vertex to a is z and the length of the shortest path from a to z is 6.

Note:

Example 1 illustrates the general principles used in Dijkstra's algorithm. Note that a shortest path from a to z could have been found by a brute force approach by examining the length of every path from a to z. However, this brute force approach is impractical for humans and even for computers for graphs with a large number of edges.

We will now consider the general problem of finding the length of a shortest path between a and z in an undirected connected simple weighted graph. Dijkstra's algorithm proceeds by finding the length of a shortest path from a to a first vertex, the length of a shortest path from a to a second vertex, and so on, until the length of a shortest path from a to z is found. As a side benefit, this algorithm is easily extended to find the length of the shortest path from a to all other vertices of the graph, and not just to z.

The algorithm relies on a series of iterations. A distinguished set of vertices is constructed by adding one vertex at each iteration. A labeling procedure is carried out at each iteration. In this labeling procedure, a vertex w is labeled with the length of a shortest path from a to w that contains only vertices already in the distinguished set. The vertex added to the distinguished set is one with a minimal label among those vertices not already in the set.

We now give the details of Dijkstra's algorithm. It begins by labeling a with 0 and the other vertices with ∞. We use the notation $L_0(a) = 0$ and $L_0(v) = \infty$ for these labels before any iterations have taken place (the subscript 0 stands for the "0th" iteration). These labels are the lengths of shortest paths from a to the vertices, where the paths contain only the vertex a. (Because no path from a to a vertex different from a exists, ∞ is the length of a shortest path between a and this vertex.)

Dijkstra's algorithm proceeds by forming a distinguished set of vertices. Let S_k denote this set after k iterations of the labeling procedure. We begin with $S_0 = \emptyset$. The set S_k is formed from S_{k-1} by adding a vertex u not in S_{k-1} with the smallest label. Once u is added to S_k, we update the labels of all vertices not in S_k, so that $L_k(v)$, the label of the vertex v at the kth stage, is the length of a shortest path from a to v that contains vertices only in S_k (that is, vertices that were already in the distinguished set together with u). Note that the way we choose the vertex u to add to S_k at each step is an optimal choice at each step, making this a greedy algorithm. (We will prove shortly that this greedy algorithm always produces an optimal solution.)

Let v be a vertex not in S_k. To update the label of v, note that $L_k(v)$ is the length of a shortest path from a to v containing only vertices in S_k. The updating can be carried out efficiently when this observation is used: A shortest path from a to v containing only elements of S_k is either a shortest path from a to v that contains only elements of S_{k-1} (that is, the distinguished vertices not including u), or it is a shortest path from a to u at the $(k-1)$st stage with the edge $\{u, v\}$ added. In other words,

$$L_k(a, v) = \min\{L_{k-1}(a, v), L_{k-1}(a, u) + w(u, v)\},$$

where $w(u, v)$ is the length of the edge with u and v as endpoints. This procedure is iterated by successively adding vertices to the distinguished set until z is added. When z is added to the distinguished set, its label is the length of a shortest path from a to z.

ALGORITHM 1 Dijkstra's Algorithm.

procedure *Dijkstra*(G: weighted connected simple graph, with
all weights positive)
{G has vertices $a = v_0, v_1, \ldots, v_n = z$ and lengths $w(v_i, v_j)$
where $w(v_i, v_j) = \infty$ if $\{v_i, v_j\}$ is not an edge in G}
for $i := 1$ **to** n
$L(v_i) := \infty$
$L(a) := 0$
$S := \emptyset$
{the labels are now initialized so that the label of a is 0 and all
other labels are ∞, and S is the empty set}
while $z \notin S$
$u :=$ a vertex not in S with $L(u)$ minimal
$S := S \cup \{u\}$
for all vertices v not in S
if $L(u) + w(u, v) < L(v)$ **then** $L(v) := L(u) + w(u, v)$
{this adds a vertex to S with minimal label and updates the
labels of vertices not in S}
return $L(z)$ {$L(z) =$ length of a shortest path from a to z}

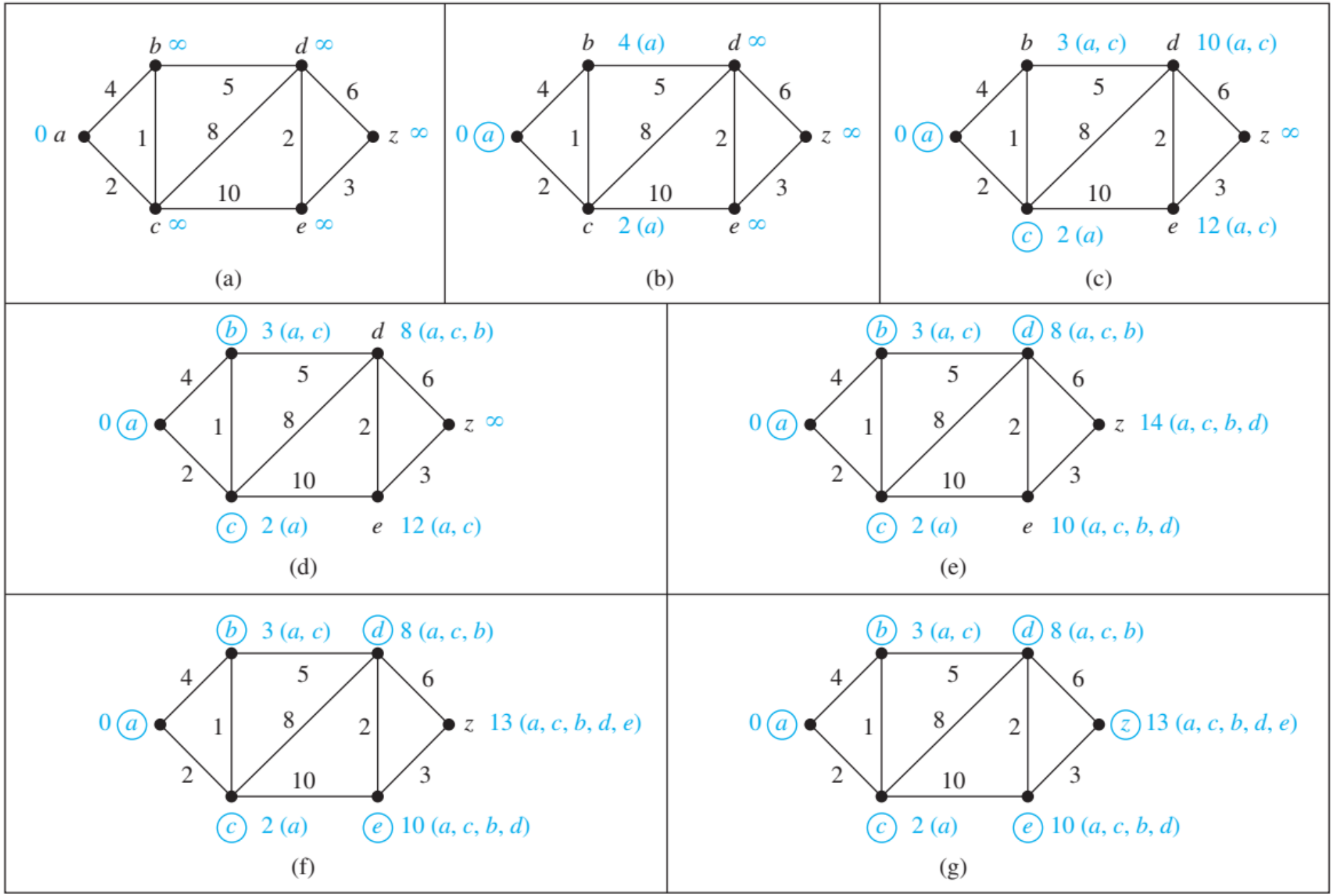


FIGURE 4 **Using Dijkstra's Algorithm to Find a Shortest Path from a to z.**

Example 2 illustrates how Dijkstra's algorithm works. Afterward, we will show that this algorithm always produces the length of a shortest path between two vertices in a weighted graph.

EXAMPLE 2 Use Dijkstra's algorithm to find the length of a shortest path between the vertices a and z in the weighted graph displayed in Figure 4(a).

Solution: The steps used by Dijkstra's algorithm to find a shortest path between a and z are shown in Figure 4. At each iteration of the algorithm the vertices of the set S_k are circled. A shortest path from a to each vertex containing only vertices in S_k is indicated for each iteration. The algorithm terminates when z is circled. We find that a shortest path from a to z is a, c, b, d, e, z, with length 13.

Remark: In performing Dijkstra's algorithm it is sometimes more convenient to keep track of labels of vertices in each step using a table instead of redrawing the graph for each step.

Next, we use an inductive argument to show that Dijkstra's algorithm produces the length of a shortest path between two vertices a and z in an undirected connected weighted graph. Take as the inductive hypothesis the following assertion: At the kth iteration

(i) the label of every vertex v in S is the length of a shortest path from a to this vertex, and

(ii) the label of every vertex not in S is the length of a shortest path from a to this vertex that contains only (besides the vertex itself) vertices in S.

When $k = 0$, before any iterations are carried out, $S = \emptyset$, so the length of a shortest path from a to a vertex other than a is ∞. Hence, the basis case is true.

Assume that the inductive hypothesis holds for the kth iteration. Let v be the vertex added to S at the $(k+1)$st iteration, so v is a vertex not in S at the end of the kth iteration with the smallest label (in the case of ties, any vertex with smallest label may be used).

From the inductive hypothesis we see that the vertices in S before the $(k+1)$st iteration are labeled with the length of a shortest path from a. Also, v must be labeled with the length of a shortest path to it from a. If this were not the case, at the end of the kth iteration there would be a path of length less than $L_k(v)$ containing a vertex not in S [because $L_k(v)$ is the length of a shortest path from a to v containing only vertices in S after the kth iteration]. Let u be the first vertex not in S in such a path. There is a path with length less than $L_k(v)$ from a to u containing only vertices of S. This contradicts the choice of v. Hence, (i) holds at the end of the $(k+1)$st iteration.

Let u be a vertex not in S after $k+1$ iterations. A shortest path from a to u containing only elements of S either contains v or it does not. If it does not contain v, then by the inductive hypothesis its length is $L_k(u)$. If it does contain v, then it must be made up of a path from a to v of shortest possible length containing elements of S other than v, followed by the edge from v to u. In this case, its length would be $L_k(v) + w(v, u)$. This shows that (ii) is true, because $L_{k+1}(u) = \min\{L_k(u), L_k(v) + w(v, u)\}$.

We now state the thereom that we have proved.

THEOREM 1

Dijkstra's algorithm finds the length of a shortest path between two vertices in a connected simple undirected weighted graph.

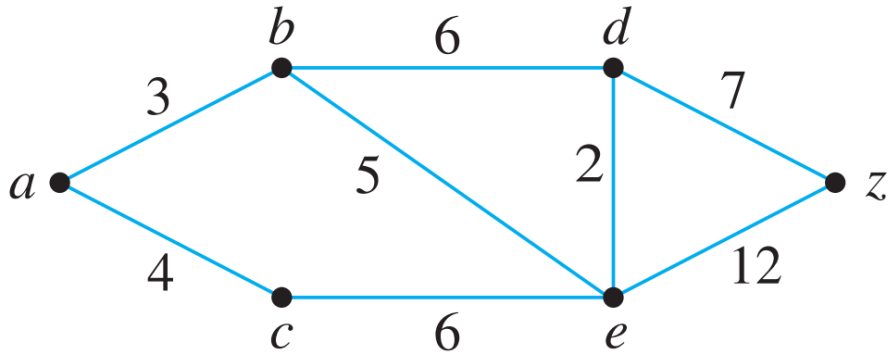
We can now estimate the computational complexity of Dijkstra's algorithm (in terms of additions and comparisons). The algorithm uses no more than $n - 1$ iterations where n is the number of vertices in the graph, because one vertex is added to the distinguished set at each iteration. We are done if we can estimate the number of operations used for each iteration. We can identify the vertex not in S_k with the smallest label using no more than $n - 1$ comparisons. Then we use an addition and a comparison to update the label of each vertex not in S_k. It follows that no more than $2(n - 1)$ operations are used at each iteration, because there are no more than $n - 1$ labels to update at each iteration. Because we use no more than $n - 1$ iterations, each using no more than $2(n - 1)$ operations, we have Theorem 2.

THEOREM 2

Dijkstra's algorithm uses $O(n^2)$ operations (additions and comparisons) to find the length of a shortest path between two vertices in a connected simple undirected weighted graph with n vertices.

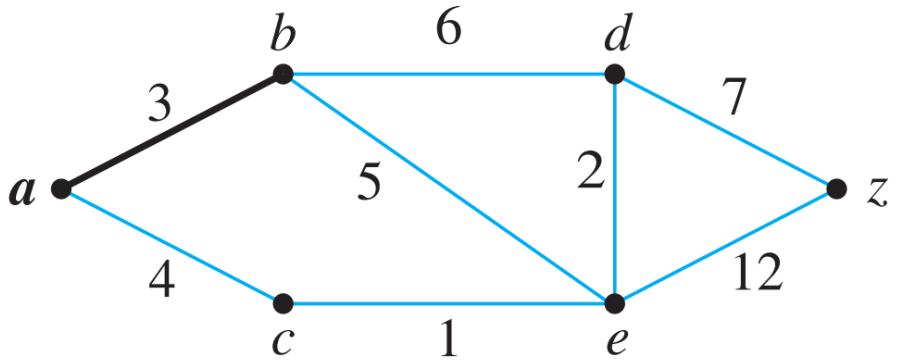
Example 2 using Dijkstra Algorithm

Show the steps in the execution of Dijkstra's shortest path algorithm for the graph shown below with starting vertex a and ending vertex z.



Solution

Step 1: Going into the **while** loop: $V(T) = \{a\}$, $E(T) = \emptyset$, and $F = \{a\}$

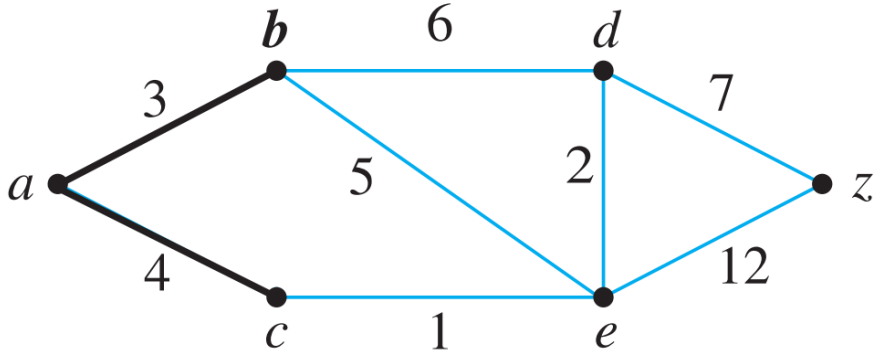


During iteration:

$F = \{b, c\}$, $L(b) = 3$, $L(c) = 4$.

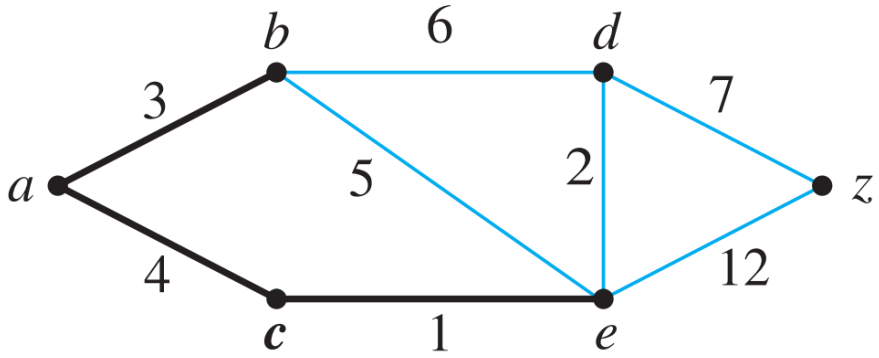
Since $L(b) < L(c)$, b is added to $V(T)$ and $\{a, b\}$ is added to $E(T)$.

Step 2: Going into the **while** loop: $V(T) = \{a, b\},\ E(T) = \{\{a, b\}\}$



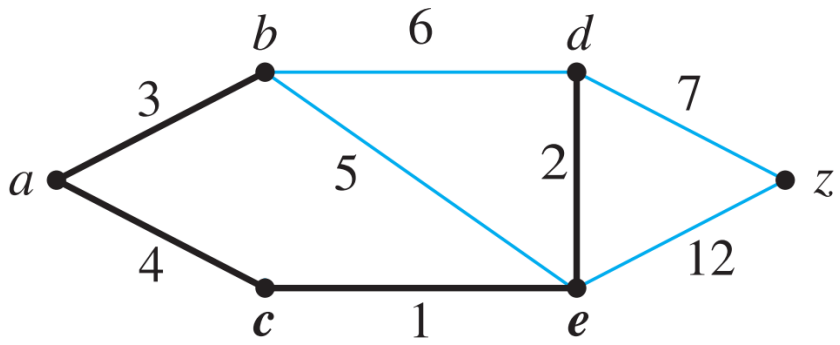
During iteration:
$F = \{c, d, e\},\ L(c) = 4,\ L(d) = 9,$
$L(e) = 8.$
Since $L(c) < L(d)$ and $L(c) < L(e)$, c is added to $V(T)$ and $\{a, c\}$ is added to $E(T)$.

Step 3: Going into the **while** loop: $V(T) = \{a, b, c\},\ E(T) = \{\{a, b\},\ \{a, c\}\}$



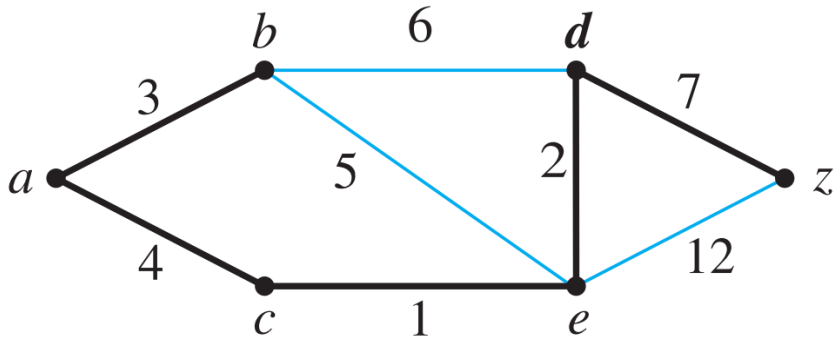
During iteration:
$F = \{d, e\},\ L(d) = 9,\ L(e) = 5$
$L(e)$ becomes 5 because *ace*, which has length 5, is a shorter path to e than *abe*, which has length 8.
Since $L(e) < L(d)$, e is added to $V(T)$ and $\{c,\ e\}$ is added to $E(T)$.

Step 4: Going into the **while** loop: $V(T) = \{a,\ b,\ c,\ e\}$,
$E(T) = \{\{a,\ b\},\ \{a,\ c\},\ \{c,\ e\}\}$



During iteration:
$F = \{d, z\},\ L(d) = 7,\ L(z) = 17$
$L(d)$ becomes 7 because *aced*, which has length 7, is a shorter path to d than *abd*, which has length 9.
Since $L(d) < L(z),\ d$ is added to $V(T)$ and $\{e, d\}$ is added to $E(T)$.

Step 5: Going into the **while** loop: $V(T) = \{a, b, c, e, d\}$,
$E(T) = \{\{a, b\},\{a, c\},\ \{c, e\},\ \{e, d\}\}$



During iteration: $F = \{z\},\ L(z) = 14$
$L(z)$ becomes 14 because *acedz*, which has length 14, is a shorter path to d than *abdz*, which has length 17.
Since z is the only vertex in F, its label is a minimum, and so z is added to $V(T)$ and $\{e, z\}$ is added to $E(T)$.

Execution of the algorithm terminates at this point because $z \in V(T)$. The shortest path from a to z has length $L(z) = 14$. ■

Keeping track of the steps in a table is a convenient way to show the action of Dijkstra's algorithm. Table 10.7.1 does this for the graph in Example 10.7.5.

Table 10.7.1

Step	$V(T)$	$E(T)$	F	$L(a)$	$L(b)$	$L(c)$	$L(d)$	$L(e)$	$L(z)$
0	$\{a\}$	$\emptyset$	$\{a\}$	**0**	∞	∞	∞	∞	∞
1	$\{a\}$	$\emptyset$	$\{b, c\}$	0	**3**	4	∞	∞	∞
2	$\{a, b\}$	$\{\{a, b\}\}$	$\{c, d, e\}$	0	3	**4**	9	8	∞
3	$\{a, b, c\}$	$\{\{a, b\}, \{a, c\}\}$	$\{d, e\}$	0	3	4	9	**5**	∞
4	$\{a, b, c, e\}$	$\{\{a, b\}, \{a, c\}, \{c, e\}\}$	$\{d, z\}$	0	3	4	**7**	5	17
5	$\{a, b, c, e, d\}$	$\{\{a, b\}, \{a, c\}, \{c, e\}, \{e, d\}\}$	$\{z\}$	0	3	4	7	5	**14**
6	$\{a, b, c, e, d, z\}$	$\{\{a, b\}, \{a, c\}, \{c, e\}, \{e, d\}, \{e, z\}\}$							

It is clear that Dijkstra's algorithm keeps adding vertices to I until it has added z. The proof of the following theorem shows that when the algorithm terminates, the label z goes the length of the shortest path to it from a.

The Traveling Salesperson Problem

We now discuss an important problem involving weighted graphs. Consider the following problem: A traveling salesperson wants to visit each of n cities exactly once and return to his starting point. For example, suppose that the salesperson wants to visit Detroit, Toledo, Saginaw, Grand Rapids, and Kalamazoo (see Figure 5). In which order should he visit these cities to travel the minimum total distance? To solve this problem we can assume the salesperson starts in Detroit (because this must be part of the circuit) and examine all possible ways for him to visit the other four cities and then return to Detroit (starting elsewhere will produce the same circuits). There are a total of 24 such circuits, but because we travel the same distance when we travel a circuit in reverse order, we need only consider 12 different circuits to find the minimum total distance he must travel. We list these 12 different circuits and the total distance traveled for each circuit. As can be seen from the list, the minimum total distance of 458 miles is traveled using the circuit Detroit–Toledo–Kalamazoo–Grand Rapids–Saginaw–Detroit (or its reverse)

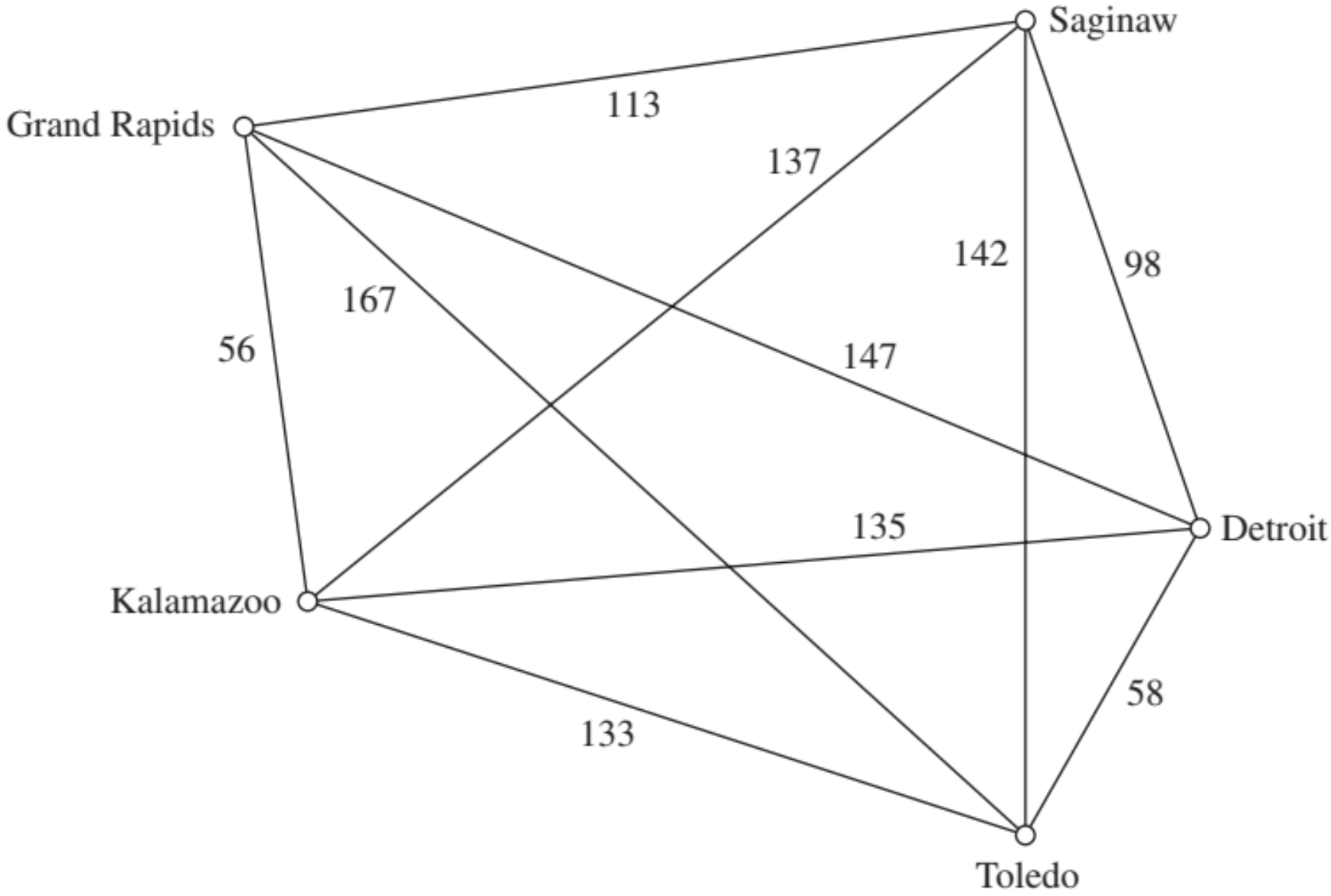


FIGURE 5 **The Graph Showing the Distances between Five Cities.**

We just described an instance of the **traveling salesperson problem**. The traveling salesperson problem asks for the circuit of minimum total weight in a weighted, complete, undirected graph that visits each vertex exactly once and returns to its starting point. This is equivalent to asking for a Hamilton circuit with minimum total weight in the complete graph, because each vertex is visited exactly once in the circuit.

The most straightforward way to solve an instance of the traveling salesperson problem is to examine all possible Hamilton circuits and select one of minimum total length. How many circuits do we have to examine to solve the problem if there are n vertices in the graph? Once a starting point is chosen, there are $(n-1)!$ different Hamilton circuits to examine, because there are $n-1$ choices for the second vertex, $n-2$ choices for the third vertex, and so on. Because a Hamilton circuit can be traveled in reverse order, we need only examine $(n-1)!/2$ circuits to find our answer. Note that $(n-1)!/2$ grows extremely rapidly. Trying to solve a traveling salesperson problem in this way when there are only a few dozen vertices is impractical. For example, with 25 vertices, a total of $24!/2$ (approximately 3.1×1023) different Hamilton circuits would have to be considered. If it took just one nanosecond (10-9 second) to examine each Hamilton circuit, a total of approximately ten million years would be required to find a minimum-length Hamilton circuit in this graph by exhaustive search techniques.

Because the traveling salesperson problem has both practical and theoretical importance, a great deal of effort has been devoted to devising efficient algorithms that solve it. However, no algorithm with polynomial worst-case time complexity is known for solving this problem. Furthermore, if a polynomial worst-case time complexity algorithm were discovered for the traveling salesperson problem, many other difficult problems would also be solvable using polynomial worst-case time complexity algorithms (such as determining whether a proposition in n variables is a tautology, discussed in Chapter 1).

A practical approach to the traveling salesperson problem when there are many vertices to visit is to use an **approximation algorithm**. These are algorithms that do not necessarily produce the exact solution to the problem but instead are guaranteed to produce a solution that is close to an exact solution. (Also, see the preamble to Exercise 46 in the Supplmentary Exercises of Chapter 3.) That is, they may produce a Hamilton circuit with total weight W' such that $W \leq W' \leq cW$, where W is the total length of an exact solution and c is a constant. For example, there is an algorithm with polynomial worst-case time complexity that works if the weighted graph satisfies the triangle inequality such that $c = 3/2$. For general weighted graphs for every positive real number k no algorithm is known that will always produce a solution at most k times a best solution. If such an algorithm existed, this would show that the class P would be the same as the class NP, perhaps the most famous open question about the complexity of algorithms (see Section 3.3).

In practice, algorithms have been developed that can solve traveling salesperson problems with as many as 1000 vertices within 2% of an exact solution using only a few minutes of computer time. For more information about the traveling salesperson problem, including history, applications, and algorithms, see the chapter on this topic in *Applications of Discrete Mathematics* [MiRo91] also available on the website for this book