6

Applications of Network Models

So far in this book we have concentrated on models that view urban services as spatially distributed systems operating in planar areas. We have thus disregarded the fact that the two-dimensional urban space is not really a continuum but rather a largely "discretized" space in which travel is restricted to take place along the highways, avenues, streets, and other transportation arteries (rail links, waterways, etc.) that compose a typical urban transportation grid. Our strategy has actually been that of considering such departures from the continuum model as "perturbations" (cf. Section 3.2) and thus accounting for their effects by computing the perturbation terms that arise from their presence.

This approach is an entirely justifiable one for the types of questions that we have examined—questions which for the most part explored the spatial relationships within urban areas and the resultant characteristics of service requirements in these areas. There are, however, many other types of questions for which it is more appropriate or more convenient to recognize explicitly the presence of a transportation network and, in fact, to use that network as the "space" in which urban services operate. In these cases, we should use *network-based* models and analysis. The purpose of this chapter is to present such network-based models and their applications.

We shall again be concerned mostly with questions that arise from the spatial distribution of demands for urban services. Specifically, we shall consider:

1. Urban travel distances and travel times when travel is restricted to take place along the links of a transportation network.

2. Vehicle-routing and collection and distribution problems such as those that arise in mail delivery, street sweeping, solid waste collection, demand-responsive bus services, snowplowing, parcel delivery, or telephone-booth repairing.
3. Problems of site selection for the location of facilities for emergency and nonemergency services such as fire houses and transportation terminals.

Throughout the discussion of these topics our representation of the urban environment will remain basically unchanged: the transportation grid in the area of interest will be our network; intersections of transportation arteries will be the network's nodes and artery segments between intersections its links. While at times it will be assumed that demand for urban services is distributed (in some way) along the whole length of the network, at other times it will be assumed that demand is concentrated at specific points on the network. The latter points will also be denoted as nodes of the network. Which of the two assumptions will be used will depend on which one is more appropriate for the application at hand.

The methodology that will be developed and used in the following sections is that of *graph theory*, the study of graphs and their properties.[1] This is still a rapidly developing area of applied mathematics that has attracted a great deal of attention during the last 15 years. The stimulus for this attention has been provided by the opportunities that the digital computer has opened for solving computationally difficult problems through the use of powerful algorithms. Indeed, graph theory is very much oriented toward the development of algorithmic approaches, and thus major parts of this chapter will be devoted to the presentation of several such algorithms.

We feel it is important, at the outset, to justify briefly for the reader the adoption of a graph-theoretic approach for presentation of this material. For it is true that most of the models that will be developed and solved here can also be formulated as linear programming (LP) or integer programming (IP) problems (and, indeed, this is how, in historical terms, many of these problems were initially formulated and investigated). However, the graph-theoretic approach has three advantages:

1. Algorithms motivated by graph-theoretic considerations are generally much more efficient than the more traditional mathematical programming algorithms (even in cases where simplex-type LP algorithms can be applied). Thus, problems of much larger size than can be handled by standard mathematical programming algorithms can often be investigated using graph-theoretic techniques.

[1]Graphs (and other related terms) will be defined in the next section.

2. Perhaps most important, the graph-theoretic approach, based as it is on the pictorial representation of problems in a network context, lends itself in an excellent way to an intuition-based presentation of algorithms and to the discovery of simple, ad hoc procedures for obtaining good approximate solutions to difficult and mathematically complex problems. We shall present several such ad hoc procedures, known as *heuristic algorithms*, later in this chapter.
3. For applications in urban service systems, the graph-theoretic approach is a most natural one since an urban transportation grid can readily be translated into a network (graph) model in the manner that was outlined above. Indeed, any good map of an urban area can serve as a handy workbench for studying network-based models of urban services.

In concluding this introduction, we note that some of the algorithms presented below are not necessarily the most computationally efficient among those available for the problem they solve or are not as readily translatable into computer code as other algorithms that can be used for the same purpose. The criteria that have been used here for algorithm selection have been:

1. That the algorithm make good intuitive sense with reference to the physical problem at hand.
2. That the most efficient algorithms currently in use be but simple variations or extensions of the algorithms described here.

6.1 DEFINITION OF TERMS AND NOTATION

The terms *network* and *graph* will be used interchangeably to refer to an entity $G(N, A)$ consisting of a finite set, N, of *nodes* (or *vertices*) and a finite set, A, of *edges* (or *arcs* or *links* or *branches*) which connect pairs of nodes. An edge connecting nodes $i \in N$ and $j \in N$ will be denoted as (i, j). If every edge has a specified orientation, the graph is called *directed* (or *oriented*); if no edge has an orientation, the graph is *undirected* (or *nonoriented*); and if some edges are directed and some are not, the graph is *mixed*. A directed edge (i, j) leads *from* i *to* j (see also Figures 6.1 and 6.2).

An edge is *incident* on the two nodes it connects. Any two nodes connected by an edge or any two edges connected by a node are said to be *adjacent*. The *degree* of a node in an undirected graph is the number of edges incident on it; for directed graphs the *indegree* of a node is the number of edges leading into that node and its *outdegree*, the number of edges leading away from it (see also Figures 6.1 and 6.2).

A *path* (or *chain*) on an undirected graph is a sequence of adjacent edges

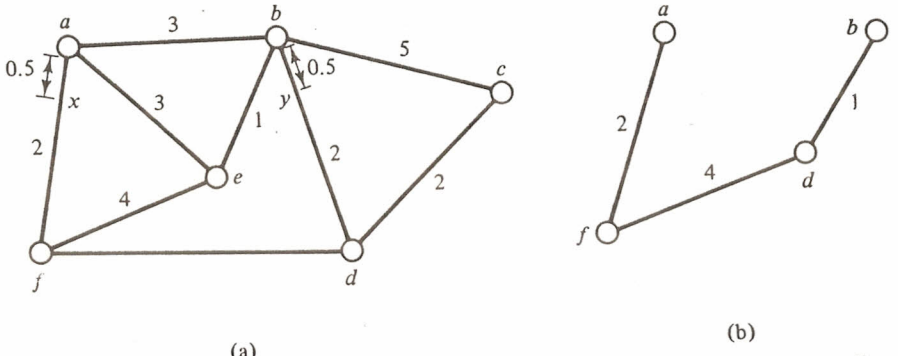


FIGURE 6.1 (a) An undirected graph G with six nodes and nine edges. Nodes a and b are adjacent; a and c are not. The degree of node e is 3, since edges (b, e) (a, e), and (f, e) are incident on it. A path from node a to node d is the path $\{a, e, f, d\} = \{(a, e), (e, f), (f, d)\}$; this path is both simple and elementary. The path $\{a, b, e, f, d, b, c\}$ contains a cycle. G is connected. (b) A subgraph of G which is also a tree but not a spanning tree. The numbers on the links of G (Fig. 6.1a) are the link lengths. The shortest distance between nodes a and c, $d(a, c)$ is equal to 7 units. The shortest distance between the points $x \in (a, f)$ and $y \in (b, d)$ is 4 units.

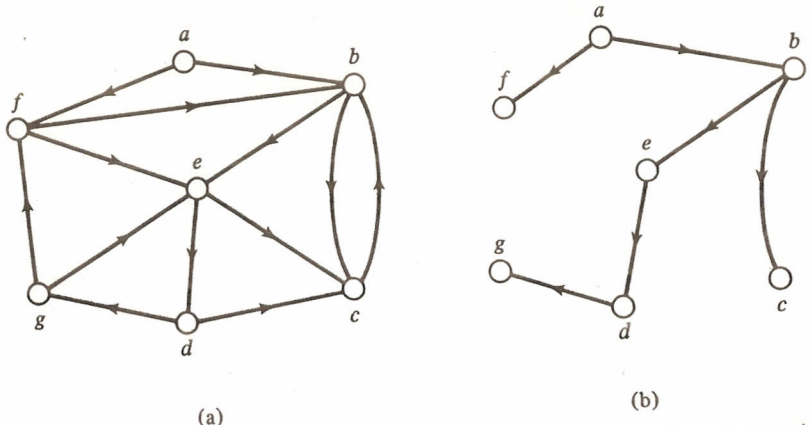


FIGURE 6.2 (a) A directed graph G. The nodes f and e are adjacent; the nodes a and d are not. The path $\{a, f, e, d, g, f, b\}$ is simple; it is not elementary, since it contains a cycle. G is connected but not strongly connected since it is impossible to go, for instance, from node g to node a. (b) A subgraph of G which is also a directed tree rooted at node a. Moreover, this tree is an arborescence, since it contains all nodes of G. The indegree of node e on G (Figure 5.2a) is 3 and its outdegree is 2.

and nodes. In a directed network, paths are directed as well, with adjacent edges leading into and away from successive nodes. Paths can be indicated as sequences of adjacent nodes (e.g., $S = \{a, b, c, \ldots, i, j, k\}$) or of adjacent edges [e.g., $S = \{(a, b), (b, c), \ldots, (i, j), (j, k)\}$]. A path is *simple* if each edge appears only once in the sequence and it is *elementary* if each node appears

only once in the sequence. A *cycle* (or *circuit*) is a path whose initial and final nodes coincide (see also Figures 6.1 and 6.2).

Related to the concept of a path is the concept of network connectivity. A node i is connected to a node j if there exists a path leading from i to j. A *connected undirected graph* is one for which a path exists between every pair of nodes $i, j \in N$. Similarly, a *directed graph* is connected if its associated undirected graph (i.e., the graph that results if orientation is removed from its edges) is connected. Note that in a connected directed graph no path may exist leading from some node i to some other node j (i.e., i is not connected to j). However, in the case of a *strongly connected directed* graph, a path exists between all ordered pairs of nodes (i.e., from every node $i \in N$ to every other node $j \in N$, and vice versa) (see also Figures 6.1 and 6.2).

A *subgraph* $G'(N', A')$ of a graph $G(N, A)$ is a graph such that[2] $N' \subset N$ and $A' \subset A$. A' can only contain arcs whose end points are in N'. A *tree* of an undirected network is a connected subgraph that has no cycles. Thus, a connected tree with t nodes has exactly $t - 1$ edges and there exists a single path between any two nodes on the tree. A *spanning tree* of a graph $G(N, A)$ contains the complete set N of the nodes of G. For directed graphs, a *directed tree* has a *root node* and a unique path from that node to every other node in the tree. An *arborescence* of a directed graph $G(N, A)$ is a directed tree that contains the complete set N of the nodes of G (see also Figures 6.1 and 6.2).

In a network problem, a number of node or link characteristics are usually specified. Most commonly, a *length* is associated with every link of G, where "length" can be in units of distance, time, money, and so on. Link lengths will be indicated as $\ell(i, j)$, where $(i, j) \in A$. The length of a path, S, between two nodes on G is then given by $L(S) = \sum_{(i,j)\in S} \ell(i, j)$. We shall use the notation $d(x, y)$ to indicate the *shortest distance* between two points $x, y \in G$. Note that x and y are not restricted to be members of the node set of G but can be any two points on G. We shall often use the notation $d(i, j)$ to indicate the shortest distance between points which belong to the node set N of G (see also Figures 6.1 and 6.2).

Some further notational conventions will be introduced as they become necessary later in this chapter.

6.2 TRAVEL DISTANCES ON NETWORKS

The most common question facing a motorist who sets out to drive from some location in a city to another concerns the choice of a route. Among the alternative routes available, one is chosen on the basis of such criteria as perceived travel distance, travel time, route "reliability," and so on. Similar

[2]The notation $A \subset B$ indicates that a set A is contained in a set B. This includes the possibilities that $A = \emptyset$, the null set, and that $A = B$.

choices must be made on a continuous basis by the dispatchers and/or the drivers of vehicles providing urban emergency services. The dispatchers, in particular, must mentally compare on a continuous basis the characteristics of alternative routes to an incident for a number of different, spatially distributed vehicles which are candidates for dispatching at any given time.

In this section we shall examine efficient methods for determining the paths that minimize travel time (or distance or cost) between any two specific points or between sets of pairs of points in a transportation network.

While the applicability of these techniques to the problem of vehicle dispatching and routing is obvious, there is an even more important motivation for beginning our presentation with a discussion of this topic: the determination of shortest paths appears consistently as a subproblem in virtually every network problem, and thus shortest-path algorithms are used as building blocks in algorithms that solve more complex problems.

A large number of algorithms have been proposed over the years for solving shortest-path problems. However, they can all be viewed as variations on a few basic themes and the two algorithms that will be described here illustrate all these themes. The first of the algorithms will determine the shortest paths between a given node and all other nodes in a network, while the second obtains simultaneously the shortest paths between all pairs of nodes. Both algorithms can be used with directed, undirected, or mixed graphs, and both assume that *all link lengths are nonnegative*. While this latter assumption is a reasonable one in the context of urban service systems—where link lengths usually represent travel distances or travel times—this may not be the case in other applications where link lengths can, for instance, represent costs (and thus a negative link length would simply imply a "profit" for traversing the corresponding link).

Algorithms for determining shortest paths in networks with some negative link lengths also exist and can be viewed as relatively simple extensions of the two algorithms that will be described below especially with regard to Algorithm 6.2. Such algorithms are developed in Problem 6.1.

6.2.1 Shortest Paths From a Given Node to All Other Nodes [$\ell(i, j) \geq 0$]

The first problem that we shall examine is that of finding the shortest path between a given node and all other nodes on a given graph $G(N, A)$. It is assumed that all link lengths, $\ell(i, j)$ are nonnegative.

The algorithm basically consists of beginning at the specified node s (the "source" node) and then successively finding its closest, second closest, third closest, and so on, node, one at a time, until all nodes in the network have been exhausted. This procedure is aided by the use of a two-entry label, $(d(j), p(j))$, for each node j.

In the evolution of the algorithm, each node can be in one of two states:

1. In the *open* state, when its label is still tentative; or
2. In the *closed* state, when its label is permanent.

At any stage in the algorithm, the label entries for node j contain the following:

$d(j)$ = length of the shortest path from s to j discovered so far

$p(j)$ = immediate predecessor node to j in the shortest path from s to j discovered so far

Finally, we shall use the symbol k to indicate the most recent (i.e., the last) node to be closed and the dummy symbol "*" to indicate the predecessor of the source node s.

The algorithm can now be described as follows:

First Shortest-Path Algorithm (Algorithm 6.1)

STEP 1: To initialize the process set $d(s) = 0, p(s) = *$; set $d(j) = \infty, p(j) = -$ for all other nodes $j \neq s$; consider node s as *closed* and all other nodes as *open*; set $k = s$ (i.e., s is the last closed node).

STEP 2: To update the labels, examine *all* edges (k, j) out of the last closed node; if node j is closed, go to the next edge; if node j is open, set the first entry of its label to

$$d(j) = \text{Min}\,[d(j), d(k) + \ell(k, j)] \tag{6.1}$$

STEP 3: To choose the next node to close, compare the $d(j)$ parts of the labels for all nodes that are in the open state. Choose the node with the smallest $d(j)$ as the next node to be closed. Suppose that this is node i.

STEP 4: To find the predecessor node of the next node to be closed, i, consider, one at a time, the edges (j, i) leading from *closed* nodes to i until one is found such that

$$d(i) - \ell(j, i) = d(j)$$

Let this predecessor node be j^*. Then set $p(i) = j^*$.

STEP 5: Now consider node i as a closed node. If all nodes in the graph are closed, then *stop*; the procedure is finished. If there are still some open nodes in the graph, set $k = i$ and return to Step 2.

Upon termination of the algorithm, the $d(j)$ part of the label of node j indicates the length of the shortest path from s to j, while the $p(j)$ part indicates the predecessor node to j on this shortest path. By tracing back the $p(j)$ parts, it is easy to identify the shortest paths between s and each of the nodes of G.

Notes

1. Ties in Step 3 can be broken arbitrarily [i.e., when two or more nodes have equal $d(j)$ and thus qualify as the next nodes to be closed, choose one among them arbitrarily]. The same applies for ties for the predecessor node in Step 4.
2. The algorithm is easy to carry out in tableau form and especially simple for computer implementation. When using the algorithm for manual solutions, the procedure can be speeded up considerably by carrying out Steps 2–4 almost simultaneously and virtually by inspection.
3. The algorithm can also be used for finding the shortest path between a source node and a specified terminal node. All that need be changed is Step 5: termination now occurs as soon as the terminal node becomes closed.
4. The algorithm can be used with directed, undirected, or mixed graphs, as long as $\ell(i, j) \geq 0$ for all $(i, j) \in A$.
5. This algorithm—or, more exactly, a minor variation of it—is generally attributed to Dijkstra [DIJK 59], but many very similar algorithms have been proposed by a number of researchers over the years.

Example 1

Consider the mixed graph shown in Figure 6.3 and suppose that the source node is a. We wish to find the shortest paths from a to all the other nodes.

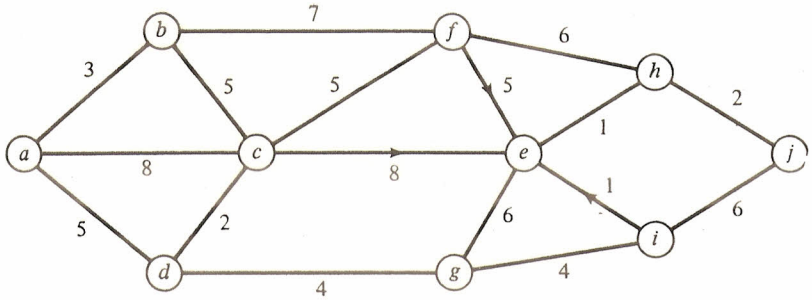


FIGURE 6.3 A mixed graph. Directed edges are denoted by directional arrows. Numbers indicate the "lengths" of edges.

Solution

Figure 6.4a shows the labels on the nodes on completion of the first pass over the five steps of the algorithm. Node b is the next node to be closed; hence after this pass $k = b$. The two closed nodes up to this point, a and b are indicated by a "+" next to their labels. These two labels are thus permanent. [It is advisable in manual solutions to use a special symbol (such as the "+") to keep track of the closed nodes.]

Figure 6.4b shows the status of the network's nodes after completion of the second pass over the algorithm. The label of f has changed, the label of c remained unchanged (why?), d is the next closed node, and $k = d$ at the end of the pass.

Let us now trace in some detail what happens during the third pass. The edges out of d to open nodes are (d, c) and (d, g). For c we have

$$d(c) = \text{Min}\,[d(c), d(d) + \ell(d, c)] = \text{Min}\,[8, 5 + 2] = 7$$

Thus, $d(c)$ is changed from 8 to 7. We also change $d(g)$ to 9 (from ∞). We now find that

$$\begin{aligned}\text{Min}\,[d(c), d(e), d(f), d(g), d(h), d(i), d(j)] &= \text{Min}\,[7, \infty, 10, 9, \infty, \infty, \infty] \\ &= 7 \\ &= d(c)\end{aligned}$$

Thus, c will be closed next. *The edges to* c from closed nodes are (b, c), (a, c), and (d, c). Checking, we find that $d(c) - \ell(d, c) = 7 - 2 = 5 = d(d)$. Therefore, we set $p(c) = d$ (i.e., the predecessor of c is d). Since there are still some open nodes, we set $k = c$ to complete this pass through the algorithm. The status of the graph at this point is shown in Figure 6.4c.

Proceeding in the same way, and after six more passes, we finally complete this work and reach the status shown in Figure 6.4d. The labels now contain all necessary information about the shortest path from a to every other node on the graph. For instance, the shortest path from a to h is of length 15 and [working backward from $p(h)$ to $p(e)$ to $p(i)$, etc.] that shortest path is $\{a, d, g, i, e, h\}$. The tree that contains all the shortest paths out of node a is now shown in Figure 6.5.

Trees like the one shown in Figure 6.5 are often referred to as *shortest-path trees.*

6.2.2 Shortest Paths between All Pairs of Nodes [$\ell(i, j) \geq 0$]

It is very often the case that the shortest paths between *all* pairs of nodes in a network are required. An obvious example is the preparation of tables indicating distances between all pairs of major cities and towns in road maps of states or regions, which often accompany such maps. Another example is

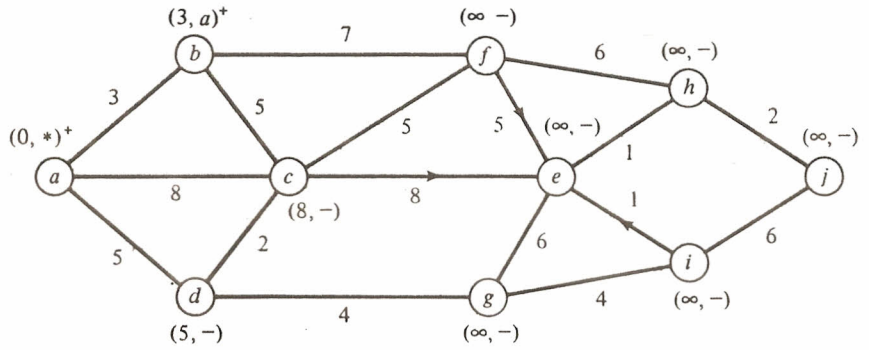


FIGURE 6.4(a) The graph of Figure 6.3 after completion of the first iteration of Algorithm 6.1 (b)–(d) the network of Figure 6.3 on completion of the second, third, and ninth (and final) iterations of Algorithm 6.1.

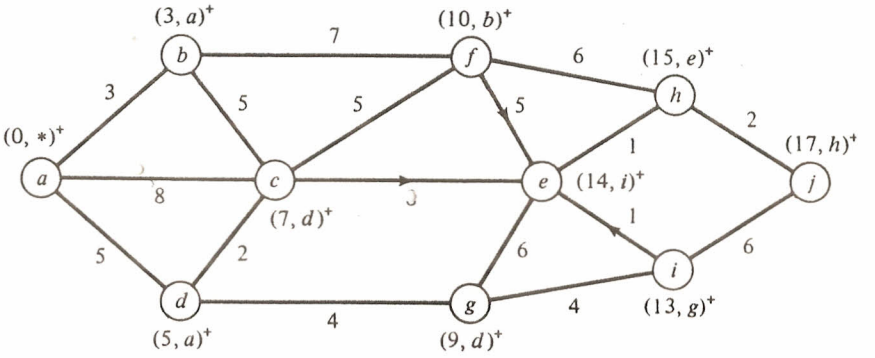


FIGURE 6.4 (cont.)

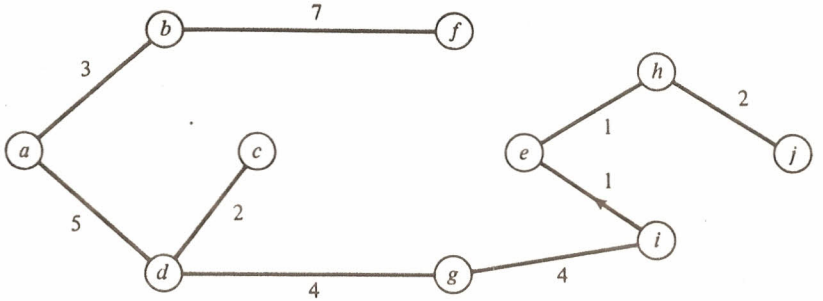


FIGURE 6.5 The shortest-path tree out of node *a* for the network of Figure 6.3.

the calculation of the distances between all pairs of "atoms" in connection with the use of the hypercube model of Section 5.4. This type of computation, as we shall see later in this chapter, is also virtually invariably required in all urban service system problems related to the location of urban facilities or to the distribution or delivery of goods. It is therefore important to have available a highly efficient method for obtaining these shortest paths.

One obvious approach is the repeated application of Algorithm 6.1 of the previous section, setting each node of the graph, in succession, as the source node and thus finding the shortest-path tree associated with each and every one of the graph's nodes. Another elegant and simple-to-program approach is generally attributed to Floyd [FLOY 62].

Floyd's algorithm is simple to describe, but its logic is not particularly easy to grasp. We shall list the algorithm below, in parallel with a convenient procedure for maintaining a record of the shortest paths. This procedure was not originally a part of Floyd's algorithm—which was only concerned with obtaining the lengths of all the shortest paths.

To begin, we number the n nodes of the graph $G(N, A)$ with the positive

integers 1, 2, . . . , n. Then, two matrices, a distance matrix, $D^{(0)}$, and a predecessor matrix, $P^{(0)}$, are set up with elements

$$d_0(i,j) = \begin{cases} \ell(i,j) & \text{if arc } (i,j) \text{ exists}^3 \\ 0 & \text{if } i = j \\ \infty & \text{if arc } (i,j) \text{ does not exist} \end{cases}$$

and

$$p_0(i,j) = \begin{cases} i & \text{for } i \neq j \\ -(\text{blank}) & \text{for } i = j \end{cases}$$

The algorithm then proceeds as follows:

Shortest-Path Algorithm 2 (Algorithm 6.2)

STEP 1: Set $k = 1$.

STEP 2: Obtain all the elements of the updated distance matrix $D^{(k)}$ from the relation

$$d_k(i,j) = \text{Min } [d_{k-1}(i,j), d_{k-1}(i,k) + d_{k-1}(k,j)] \qquad (6.2)$$

STEP 3: Obtain all the elements of the updated predecessor matrix $P^{(k)}$ by using

$$p_k(i,j) = \begin{cases} p_{k-1}(k,j) & \text{if } d_k(i,j) \neq d_{k-1}(i,j) \\ p_{k-1}(i,j) & \text{otherwise} \end{cases} \qquad (6.3)$$

STEP 4: If $k = n$, *stop*; if $k < n$, set $k = k + 1$ and return to Step 2.

On termination, the length, $d(i,j)$, of the shortest path from node i to node j is given by element $d_n(i,j)$ of the final matrix $D^{(n)}$, while the final predecessor matrix, $P^{(n)}$, makes possible the tracing back of each one of the shortest paths.

While the detailed example that follows should help clarify the algorithm for the reader, its basic rationale can be quickly illustrated. Say that the shortest path from node 4 to node 6 is the path {4, 5, 1, 7, 3, 6}. Then the first pass ($k = 1$) over the algorithm will replace $d(5, 7)$ ($= \ell(5, 7)$) by $d(5, 1) + d(1, 7)$ ($= \ell(5, 1) + \ell(1, 7)$). Then, on the third pass ($k = 3$), $d(7, 6)$ [which may already have been changed from its original value $\ell(7, 6)$] will be replaced by $d(7, 3) + d(3, 6)$ [$= \ell(7, 3) + \ell(3, 6)$]. On the fifth pass ($k = 5$), the current $d(4, 7)$ will be replaced by the sum of $d(4, 5)$ [$= \ell(4, 5)$] and $d(5, 7)$ (with the latter as modified during pass 1). Finally, on pass seven ($k = 7$), $d(4, 6)$ will be replaced by $d(4, 7) + d(7, 6)$. This sum (from the earlier passes) is equal to $\ell(4, 5) + \ell(5, 1) + \ell(1, 7) + \ell(7, 3) + \ell(3, 6)$.

[3] If $m > 1$ arcs exist from i to j, set $d_0(i,j) = \text{Min } [\ell_1(i,j), \ell_2(i,j), \ldots, \ell_m(i,j)]$.

Example 2

We wish to find the shortest paths between all pairs of nodes for the mixed graph of Figure 6.6. Since there are five nodes in the graph, five passes over the algorithm will be required. The initial matrices $D^{(0)}$ and $P^{(0)}$ and their five updated versions are listed below. To facilitate comprehension of the example, we indicate with a "+" all the matrix elements that have been

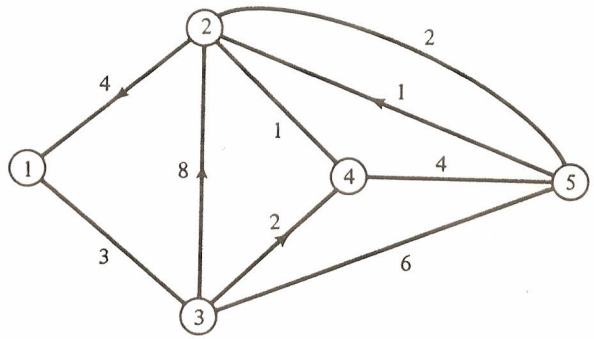


FIGURE 6.6 Mixed graph used for Example 2.

updated from the previous pass. At each pass we also indicate with a "*" that row and that column (row and column k in each case) that is known a priori not to require any changes and thus can be copied directly in the updated matrix (see Step 2 of Algorithm 6.2).

Initialization:

$D^{(0)} =$

	1	2	3	4	5
1	0	∞	3	∞	∞
2	4	0	∞	1	2
3	3	8	0	2	6
4	∞	1	∞	0	4
5	∞	1	6	4	0

$P^{(0)} =$

	1	2	3	4	5
1	—	1	1	1	1
2	2	—	2	2	2
3	3	3	—	3	3
4	4	4	4	—	4
5	5	5	5	5	—

Through Node 1:

$D^{(1)} =$

	1*	2	3	4	5
*1	0	∞	3	∞	∞
2	4	0	7+	1	2
3	3	8	0	2	6
4	∞	1	∞	0	4
5	∞	1	6	4	0

$P^{(1)} =$

	1*	2	3	4	5
*1	—	1	1	1	1
2	2	—	1+	2	2
3	3	3	—	3	3
4	4	4	4	—	4
5	5	5	5	5	—

Through Node 2:

$D^{(2)}$ =

	1	2*	3	4	5
1	0	∞	3	∞	∞
*2	4	0	7	1	2
3	3	8	0	2	6
4	5+	1	8+	0	3+
5	5+	1	6	2+	0

$P^{(2)}$ =

	1	2*	3	4	5
1	—	1	1	1	1
*2	2	—	1	2	2
3	3	3	—	3	3
4	2+	4	1+	—	2+
5	2+	5	5	2+	—

Through Node 3:

$D^{(3)}$ =

	1	2	3*	4	5
1	1	11+	3	5+	9+
2	4	0	7	1	2
*3	3	8	0	2	6
4	5	1	8	0	3
5	5	1	6	2	0

$P^{(3)}$ =

	1	2	3*	4	5
1	—	3+	1	3+	3+
2	2	—	1	2	2
*3	3	3	—	3	3
4	2	4	1	—	2
5	2	5	5	2	—

Through Node 4:

$D^{(4)}$ =

	1	2	3	4*	5
1	0	6+	3	5	8+
2	4	0	7	1	2
3	3	3+	0	2	5+
*4	5	1	8	0	3
5	5	1	6	2	0

$P^{(4)}$ =

	1	2	3	4*	5
1	—	4+	1	3	2+
2	2	—	1	2	2
3	3	4+	—	3	2+
*4	2	4	1	—	2
5	2	5	5	3	—

Through Node 5:

$D^{(5)}$ =

	1	2	3	4	5*
1	0	6	3	5	8
2	4	0	7	1	2
3	3	3	0	2	5
4	5	1	8	0	3
*5	5	1	6	2	0

$P^{(5)}$ =

	1	2	3	4	5*
1	—	4	1	3	2
2	2	—	1	2	2
3	3	4	—	3	2
4	2	4	1	—	2
*5	2	5	5	2	—

Note that on the last pass no improvements could be found for $D^{(5)}$ over $D^{(4)}$. The final matrices $D^{(5)}$ and $P^{(5)}$ indicate, for instance, that the shortest path from node 1 to node 5 has length $d(1, 5) = 8$ units and that this shortest path is the path {1, 3, 4, 2, 5}. To identify that shortest path, we examined row 1 of the $P^{(5)}$ matrix. Entry $p_5(1, 5)$ says that the predecessor node to 5 in the path from 1 to 5 is node 2; then, entry $p_5(1, 2)$ says that the predecessor node to 2 in the path from 1 to 2 is node 4; similarly, we backtrace the rest of the path by examining $p_5(1, 4)$ $(= 3)$ and $p_5(1, 3) = 1$. In general, backtracing stops when the predecessor node is the same as the initial node of the required path.

For another illustration, the shortest path from node 4 to node 3 is $d(4, 3) = 8$ units long and the path is {4, 2, 1, 3}. The predecessor entries that must be read are, in order, $p_5(4, 3) = 1$, $p_5(4, 1) = 2$, and finally $p_5(4, 2) = 4$—at which point we have "returned" to the initial node.

6.2.3 Traffic Assignment Problem

A simple application of shortest-path techniques occurs in the preliminary determination of the traffic loads to be expected on different segments of a transportation grid. To make this determination, an origin-destination traffic matrix is required for the network of interest. Passengers are then assumed to follow the least distance (or time or cost) path from origin to destination through the network. The levels of flow on all links of the network are then computed by:

1. Using some algorithm to find the shortest paths between all pairs of nodes.
2. Adding, for each origin-destination pair i–j, the traffic flow $t(i, j)$ to the flow on every link in the shortest path from i to j.

Example 3

Consider again the network of Figure 6.6. Assuming that the origin-destination data are as shown below, find the traffic flow on the links of the graph.

Origin-destination matrix (in tens of vehicles per hour):

$[t(i, j)]$ =

From\To → / ↓	1	2	3	4	5
1	—	30	35	40	15
2	10	—	15	12	10
3	50	40	—	35	20
4	25	30	35	—	40
5	45	30	35	40	—

Solution

We have already obtained all shortest paths for this graph and recorded them in the matrix $P^{(5)}$ (see Section 6.2.2). We can then determine the traffic flow on each link of the graph, as shown in Figure 6.7. To better understand the computation of the various flows, consider the flow of 52 units on the undirected link (2, 4) and in the direction from 2 to 4. The link (2, 4) in the 2 → 4

Through Node 2:

$D^{(2)}$

	1	2*	3	4	5
1	0	∞	3	∞	∞
*2	4	0	7	1	2
3	3	8	0	2	6
4	5⁺	1	8⁺	0	3⁺
5	5⁺	1	6	2⁺	0

$P^{(2)}$

	1	2*	3	4	5
1	—	1	1	1	1
*2	2	—	1	2	2
3	3	3	—	3	3
4	2⁺	4	1⁺	—	2⁺
5	2⁺	5	5	2⁺	—

Through Node 3:

$D^{(3)}$

	1	2	3*	4	5
1	1	11⁺	3	5⁺	9⁺
2	4	0	7	1	2
*3	3	8	0	2	6
4	5	1	8	0	3
5	5	1	6	2	0

$P^{(3)}$

	1	2	3*	4	5
1	—	3⁺	1	3⁺	3⁺
2	2	—	1	2	2
*3	3	3	—	3	3
4	2	4	1	—	2
5	2	5	5	2	—

Through Node 4:

$D^{(4)}$

	1	2	3	4*	5
1	0	6⁺	3	5	8⁺
2	4	0	7	1	2
3	3	3⁺	0	2	5⁺
*4	5	1	8	0	3
5	5	1	6	2	0

$P^{(4)}$

	1	2	3	4*	5
1	—	4⁺	1	3	2⁺
2	2	—	1	2	2
3	3	4⁺	—	3	2⁺
*4	2	4	1	—	2
5	2	5	5	3	—

Through Node 5:

$D^{(5)}$

	1	2	3	4	5*
1	0	6	3	5	8
2	4	0	7	1	2
3	3	3	0	2	5
4	5	1	8	0	3
*5	5	1	6	2	0

$P^{(5)}$

	1	2	3	4	5*
1	—	4	1	3	2
2	2	—	1	2	2
3	3	4	—	3	2
4	2	4	1	—	2
*5	2	5	5	2	—

Note that on the last pass no improvements could be found for $D^{(5)}$ over $D^{(4)}$. The final matrices $D^{(5)}$ and $P^{(5)}$ indicate, for instance, that the shortest path from node 1 to node 5 has length $d(1, 5) = 8$ units and that this shortest path is the path {1, 3, 4, 2, 5}. To identify that shortest path, we examined row 1 of the $P^{(5)}$ matrix. Entry $p_5(1, 5)$ says that the predecessor node to 5 in the path from 1 to 5 is node 2; then, entry $p_5(1, 2)$ says that the predecessor node to 2 in the path from 1 to 2 is node 4; similarly, we backtrace the rest of the path by examining $p_5(1, 4)$ (= 3) and $p_5(1, 3) = 1$. In general, backtracing stops when the predecessor node is the same as the initial node of the required path.

For another illustration, the shortest path from node 4 to node 3 is $d(4, 3) = 8$ units long and the path is {4, 2, 1, 3}. The predecessor entries that must be read are, in order, $p_5(4, 3) = 1$, $p_5(4, 1) = 2$, and finally $p_5(4, 2) = 4$—at which point we have "returned" to the initial node.

6.2.3 Traffic Assignment Problem

A simple application of shortest-path techniques occurs in the preliminary determination of the traffic loads to be expected on different segments of a transportation grid. To make this determination, an origin-destination traffic matrix is required for the network of interest. Passengers are then assumed to follow the least distance (or time or cost) path from origin to destination through the network. The levels of flow on all links of the network are then computed by:

1. Using some algorithm to find the shortest paths between all pairs of nodes.
2. Adding, for each origin-destination pair i–j, the traffic flow $t(i, j)$ to the flow on every link in the shortest path from i to j.

Example 3

Consider again the network of Figure 6.6. Assuming that the origin-destination data are as shown below, find the traffic flow on the links of the graph.

Origin-destination matrix (in tens of vehicles per hour):

$[t(i, j)]$

From\To →	1	2	3	4	5
1	—	30	35	40	15
2	10	—	15	12	10
3	50	40	—	35	20
4	25	30	35	—	40
5	45	30	35	40	—

Solution

We have already obtained all shortest paths for this graph and recorded them in the matrix $P^{(5)}$ (see Section 6.2.2). We can then determine the traffic flow on each link of the graph, as shown in Figure 6.7. To better understand the computation of the various flows, consider the flow of 52 units on the undirected link (2, 4) and in the direction from 2 to 4. The link (2, 4) in the 2 → 4

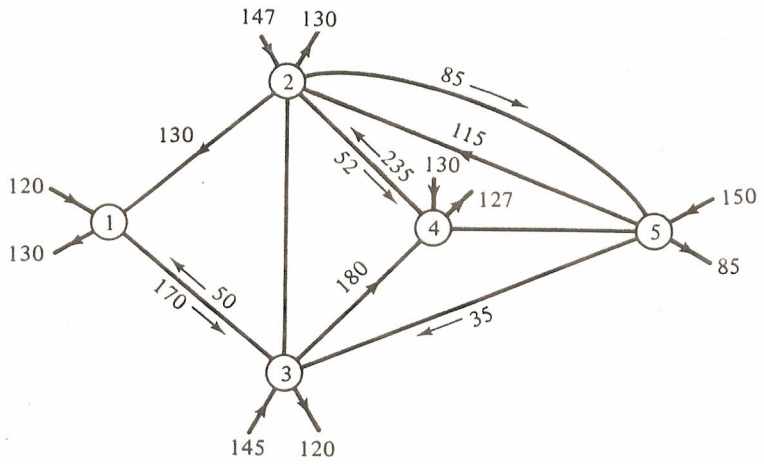


FIGURE 6.7 Final traffic assignment of Example 3.

direction is part of the shortest path from node 2 to node 4 and from node 5 to node 4. Therefore, the traffic for these two destination pairs (i.e., 12 and 40 units, respectively) will use the (2, 4) link in the 2 ⟶ 4 direction, hence the flow of 52 units.

Note that there are traffic flows in both directions through some undirected links and that other (directed or undirected) links remain completely unused as indicated by the absence of any traffic flow numbers next to them. Finally, note the conservation of traffic at each node of the network.

This type of traffic flow determination can be useful in determining the capacity requirements for an urban transportation network at the initial stages of planning or as long as the network is relatively free of congestion. It is much less useful and meaningful in predicting traffic flows for a transportation network that has already been built and is heavily utilized. This is so because the determination of traffic flows, as performed above, ignores the facts that:

1. Shortest distance—or, shortest travel time—is not the sole criterion for route selection.
2. The capacity of any link on a transportation system is finite.
3. Because of 2, travel speeds are a function of the amount of congestion on a transportation link and, as a result, the determination of the shortest paths cannot be made independently of traffic-flow determination.

In particular, the effects of congestion call for more sophisticated approaches than the above in determining traffic flows. These approaches are

generally referred to as "traffic assignment" techniques. Good general references about these techniques can be found in [FLOR 76]. (See also Problem 4.10 for the differences between "user-optimum" and "system-optimum" traffic assignment.)

6.2.4 Some Complications for Urban Travel

Certain complications arise when shortest-path algorithms, such as the two we have already discussed, are applied in the urban environment.

For instance, a motorist often suffers a penalty, in terms of increased travel time for right and, especially, for left turns when traveling on urban streets. If one wishes to take this into account in a network model and is careless in doing so, serious difficulties can result.

Example 4: Turn Penalties and Constraints

Consider the network of Figure 6.8 representing a small part of a downtown urban street grid. The numbers next to the arcs indicate average travel *times* on each street segment and nodes represent street corners. Suppose that each turn (right or left) carries a penalty of two time units. Then the shortest path from node A to node D is the path $\{A, B, D\}$ of length equal to 12 time units

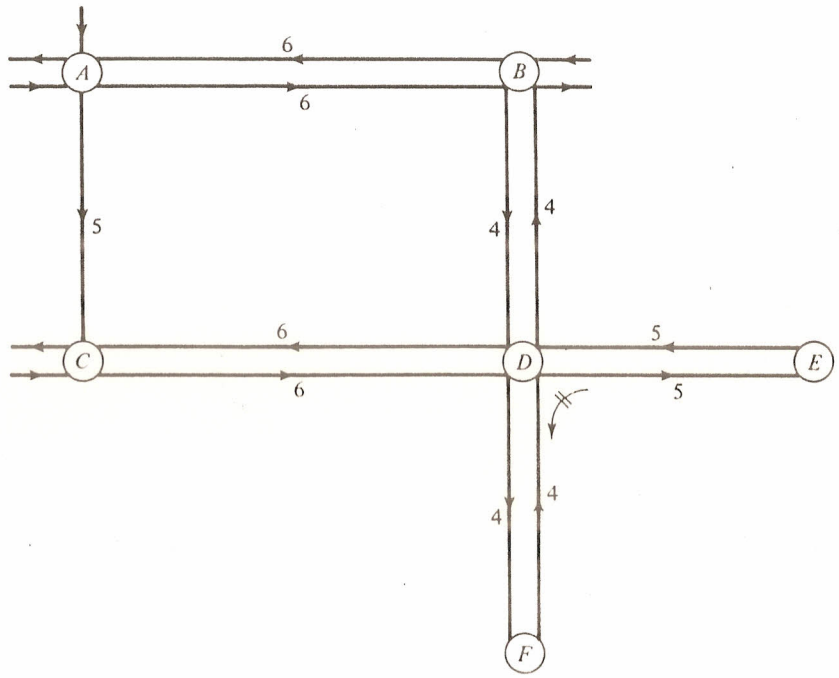


FIGURE 6.8 Section of a downtown urban grid.

(including the 2-unit penalty for the turn at D), whereas the shortest path from A to E is $\{A, C, D, E\}$ of length 20 units. Thus, the path that one would follow in going from A to D along the shortest path from A to E does not coincide with (and is of different length from) the shortest path from A to D! This, of course, is inconsistent with the fundamental premise of all shortest-path algorithms (why?).

As Example 4 suggests, "naive" network models (such as that of Figure 6.8) must be modified somewhat to make them appropriate for application of our two shortest-path algorithms when turn penalties are to be taken into consideration. Along the same lines, shortest paths on a downtown urban grid may also include cycles (in the sense of passing through the same intersection twice). For instance, if turns to the left are prohibited at intersection D of Figure 6.8, the shortest path from E to F might be something like $\{E, D, B, A, C, D, F\}$. This again is not provided for by shortest-path algorithms thus requiring modification of the straightforward network model.

Similarly, in modeling the possible routes of a commuter—who may have a choice between, say, driving a car all the way to work or driving that car to the nearest subway station, parking the car there, and riding to the job on a train—care must be taken to account for any time losses that the commuter may face at any mode-changing points (or, for that matter, at stations where the commuter may have to change trains).

In conclusion, one should be aware of such complications. The level of detail of a model of an urban network must be consistent with the level of detail sought in the analysis, if one wishes to apply the shortest-path algorithms that we have already discussed (as well as the other algorithms to be presented later) to these models. Increasing the level of detail usually means creating "dummy" (or "artificial") nodes and arcs. These ideas are illustrated in Problems 6.2 and 6.3.

6.2.5 Complexity of Algorithms

A natural question to ask at this point is the following: Which one of the two shortest-path algorithms that we have described is better for computing the shortest paths between all possible pairs[4] of nodes on a network? To answer this question, a brief discussion of the subject of algorithmic complexity is first in order. The "goodness" of an algorithm is described in good part by its complexity.

The complexity of an algorithm is most often measured through the number of *elementary operations* that the algorithm requires to arrive at an

[4]For Algorithm 6.1, as we have already noted, this implies its repeated application, making each node of the graph, in succession, the source node.

answer under "worst-case conditions." An elementary operation is any one of the arithmetic operations (addition, subtraction, multiplication, division) or a comparison between two numbers or the execution of a branching instruction. We shall discuss the meaning of "worst-case conditions" in a short while.

Consider, for instance, the application of Algorithm 6.2 to the case of a network with n nodes. Each pass through Step 2 of the algorithm requires the checking and possible updating of $(n-1)^2$ matrix elements. (Remember that there is no need to check and update the elements of the kth row and of the kth column—see also Example 2—since we know that these elements will remain unchanged.) Thus, each pass through Step 2 requires $(n-1)^2$ additions [to compute the quantities $d_{k-1}(i,k) + d_{k-1}(k,j)$] as well as $(n-1)^2$ comparisons between two numbers. Similarly, Step 3 of the algorithm requires $(n-1)^2$ comparisons on each pass, while Step 4 requires a single comparison (k with n), an addition (incrementing k by 1), and a branching[5] (return to Step 2). Thus, a total of $3(n-1)^2 + 3$ elementary operations are needed for each pass of the algorithm. Since there are a total of n passes, we conclude that the number of elementary operations, T, required for Algorithm 6.2 to provide the desired answers is

$$T = 3n(n-1)^2 + 3n = 3n^3 - 6n^2 + 6n$$

Now, as n increases, the value of T is largely determined by the $3n^3$ term. Furthermore, what really counts when we compare the performance of this algorithm with the performance of other algorithms for large values of n is the term n^3—after all, the factor of 3 is insignificant in view of the tenfold advances in computer speeds that occur every few years and in view of the differences in speeds among different types of computers. Thus, we say that "the complexity of Algorithm 6.2 is proportional to n^3" or, more simply, that "Algorithm 6.2 is[6] $O(n^3)$." Note that the complexity of the algorithm is measured with reference to the size of the input to the algorithm (i.e., the quantity n). Finally, under the assumption that each elementary operation takes approximately one unit of time (be that 10^{-9} second or 10^{-3} second depending on the speed of the computer), we also say that "Algorithm 6.2 requires $O(n^3)$ time."

Consider now a case in which a network with n nodes is *completely connected* (i.e., it is possible to go from any node to any other node directly, without passing through any intermediate nodes).

[5]On the last pass only a comparison and a branching to STOP are needed.

[6]Do not confuse this with the notation $o(\cdot)$ which was used in Chapter 2 in connection with the definition of infinitesimal terms related to the Poisson process.

Exercise 6.1 Show that if Algorithm 6.1 is used with such a network to find the shortest path from any *given* source node to all other nodes (as in Example 1), the algorithm requires $O(n^2)$ time.

It follows from this exercise that, since Algorithm 6.1 must be repeated n times (once for each possible source node) in order to find the shortest paths between all possible pairs of nodes, time $O(n^3)$ is required to find all shortest paths through repeated uses of Algorithm 6.1.

Note now that the case of a completely connected network is actually a *worst case* as far as Algorithm 6.1 is concerned. For it is possible to find other instances of networks for which it may take less than $O(n^3)$ time to find all shortest paths through the algorithm. For example, if the network is, say, an undirected tree—so that only a single path exists between each pair of nodes and the distance matrix $[d(i, j)]$, is symmetric—Algorithm 6.1 could be adjusted so that it requires far less than $O(n^3)$ time.[7] This example should clarify what we mean by worst-case conditions.

We conclude that the complexity of Algorithms 6.1 and 6.2 is the same, $O(n^3)$, under worst-case conditions. Thus, according to our definition, *Algorithms 6.1 and 6.2 are equally "good,"* although it is true that Algorithm 6.1 is more flexible and can thus better take advantage of special network structures.[8] (On the other hand, Algorithm 6.2 is easier to program on a computer.)

Algorithms are generally classified by computer scientists into two broad categories. *Polynomial algorithms* are those whose complexity is proportional to or bounded by a polynomial function of the size of the input. *Exponential* (or *non-polynomial*) *algorithms* are those that, for sufficiently large sizes of the input, violate all polynomial bounds. For instance, algorithms whose complexity is $O(n^3)$ or $O(n^5)$ are obviously polynomial ones; so is an algorithm with complexity, say, $O(n \cdot \log_2 n)$, since the quantity $n \cdot \log_2 n$ is bounded from above by n^2. On the other hand, algorithms that are $O(2^n)$ or $O(k^n)$—where k is a constant—or $O(n!)$—remember Stirling's approximation for factorials—or $O(n^{\log_2 n})$ are exponential.

Polynomial algorithms are considered "good" algorithms while exponential algorithms are "poor" ones. The terms "efficient" and "inefficient" are most often used for "good" and "poor," respectively. While an algorithm that is $O(2^n)$ may terminate faster than an algorithm that is $O(n^5)$ for small values of n, it is possible to find an input size n_0, such that if $n \geq n_0$, the former algorithm will terminate after the latter one.

[7] Note that the same is not true for Algorithm 6.2 as described here. It always requires $O(n^3)$ time, irrespective of the structure of the network.

[8] When two algorithms are both $O(n^k)$ for some k, then other factors such as the respective coefficients of n^k, ease of programming, computer memory requirements and *average* performance should be considered.

Many computer scientists have argued that it would perhaps be better to measure complexity in a probabilistic sense (i.e., in terms of the number of operations required by the algorithm in a "typical" application). The problem with this is defining what "typical" means. While we shall continue to use worst-case performance as a measure of algorithmic complexity, it is important to appreciate at this point that this is only *one of several* choices (and still the most common one) that could be made.

Finally, algorithms are also described as *exact* or *heuristic*. The former are guaranteed to terminate, given sufficient (computer) time, with an optimal solution to the problem which they are designed to solve. Heuristic algorithms do not offer such a guarantee; they usually trade off "guaranteed optimality" for speed in providing an answer.

Later in this chapter we shall have occasion to use all the terms introduced in this section.

6.3 MINIMUM-SPANNING-TREE PROBLEM

A spanning tree of an undirected graph $G(N, A)$ has already been defined as a tree of the graph G that contains the complete set of nodes, N, of G (see also Figure 6.9). The minimum-spanning-tree problem is then concerned with finding the one among all possible spanning trees of a graph $G(N, A)$ with the minimum total link length. If the number of nodes in the set N is n, then all spanning trees of G obviously contain $n - 1$ links.

Just as in the case of shortest-path problems, the minimum-spanning-tree (MST) problem has direct applications of its own, primarily in problems of transportation network design, and also constitutes an important building block in some solution approaches to other more complex problems such as node-covering problems (see Section 6.4). For a brief example of an application of the MST problem, consider the case in which map locations of n rural towns are given along with a matrix listing the Euclidean distances (or

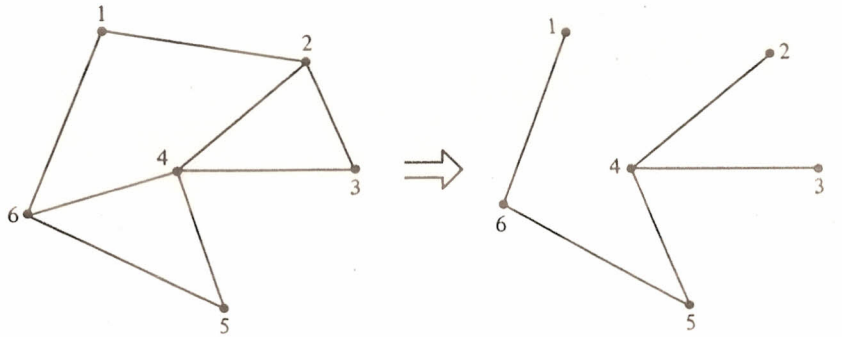


FIGURE 6.9 A graph and one of its spanning trees.

some other measure of distance, time, or cost) between all possible pairs of towns. The MST can be interpreted in this case as the minimum length of roads needed to connect, directly or indirectly, all pairs of towns, *assuming that all road segments (links) begin and end in some pair of towns.*

The last sentence emphasizes the distinction between the MST problem and the so-called "Steiner problem." The latter has the same objective as the former but allows the introduction of artificial "nodes" (i.e., links can meet anywhere on the plane instead of only at the specified points that are to be spanned). To illustrate this further, consider four towns that lie at the corners of a unit square. Obviously, one solution to the MST problem in this case is the one shown in Figure 6.10a; the total length of the minimum spanning tree is 3 units. On the other hand, it can be shown that if creation of artificial nodes is permitted (Steiner's problem), the total line length needed to connect the four points can be further reduced. The solution to Steiner's problem in this case is shown in Figure 6.10b and involves creation of two artificial nodes (A and B in Figure 6.10b); the total length of the Steiner tree is $1 + \sqrt{3}$ in this case.

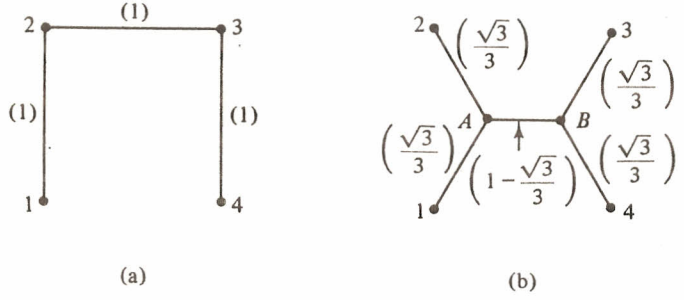


FIGURE 6.10 Solution of the MST problem (a) and of the Steiner problem (b) for four points lying at the corners of a unit square.

6.3.1 Solving the MST Problem

The solution of MST problems turns out to be very simple. Of the several efficient algorithms that have been proposed for it, some are more useful for manual computations and others more suitable for computer applications. All the algorithms derive their validity from the following property of minimum spanning trees.

Fundamental property of MST's. Suppose that a procedure for finding the MST of any graph G has been discovered and that in the course of the construction of the MST, by using this procedure, the set N of nodes of G has been partitioned into k distinct trees $T_1(N_1, A_1), T_2(N_2, A_2), \ldots, T_k(N_k, A_k)$ with $N_1 \cup N_2 \cup \ldots \cup N_k = N$ and $N_i \cap N_j = \varnothing$ for $i, j = 1, 2, \ldots, k$ $(i \neq j)$. By the construction assumption $T_1, T_2, \ldots, T_k$ will eventually become constituent parts of the *MST*. Note that a "tree" may also consist of a single node of G. Now let T_s be one of these trees and let (i^*, j^*) be the *shortest link* with the property that one of its roots, node i^*, is in the tree T_s and the other root, node j^*, is not in the tree T_s. Let T_t be the tree to which j^* belongs. It is then true that the link (i^*, j^*) must be a link in the final MST and that, therefore, trees T_s and T_t can be connected next through link (i^*, j^*).

Proof: (By contradiction) Assume that (i^*, j^*) is not part of the final MST, which, however, must, by assumption, contain both the tree T_s and the tree T_t. By definition of the MST, there is exactly one path leading from T_s to T_t. Let then (i, j) be that link on the MST which has one of its roots, i, in T_s and the other, j, not in T_s and which is the first link on the path from T_s to T_t. (It is possible that $i = i^*$ or that $j = j^*$.) By definition of (i^*, j^*), $\ell(i, j) > \ell(i^*, j^*)$. If we then replace (i, j) by (i^*, j^*), a new spanning tree will result which will have less total length than the MST—a contradiction.

Note that in stating the property above we did not make any specific assumptions about the procedure for constructing the MST. We can then begin the procedure with n trees (where n is the number of nodes in the set N), with each tree consisting of a *single isolated node* and no links. We can then, according to the property that we proved, connect any arbitrarily chosen tree (node) to its nearest tree (node) and continue this procedure until all nodes are finally connected. Before proceeding to describe the details of this algorithm, we state the following corollary to the fundamental property of the MST's that we have proven:

Corollary: In an undirected network, G, the link of shortest length out of any node is part of the MST.

Proof: Follows directly from the fundamental property of MST's.

This corollary can be used to speed up the algorithmic procedure outlined below. However, it will not be explicitly included here.

MST Algorithm (Algorithm 6.3)

STEP 1: Begin the construction of the minimum spanning tree at some arbitrary node, say node i. Find the node closest to i, say j,[9] and connect it to i (i.e., find the shortest link out of i and include it in

[9]The algorithm assumes $n \geq 2$, the case $n = 1$ is trivial.

the tree together with the node at its other end). Break ties, if any, arbitrarily.

STEP 2: If all nodes have been connected, *stop*; the minimum spanning tree has been obtained. If there are still isolated nodes, go to Step 3.

STEP 3: Find the one among the still isolated nodes which is closest to the already connected nodes and connect it with the already connected nodes (i.e., find the shortest link from the connected nodes to isolated nodes and include it in the tree together with the node at its other end). Break ties, if any, arbitrarily. Return to Step 2.

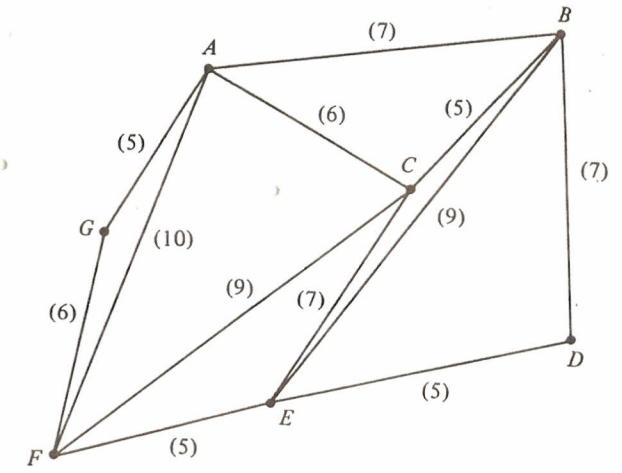
Example 5: MST

For the graph shown in Figure 6.11a, the algorithm above works as follows. Suppose that we choose arbitrarily to begin construction of the MST at node A. Since node G is closest to A, G and the link (A, G) first become part of the MST. There is next a tie between C and F as the next node to be connected to the MST (C is 6 units away from A and F 6 units away from G). We break the tie arbitrarily in favor of F, and link (G, F) and the node F are thus included in the MST. Proceeding in this way, nodes E, D, C, and B in this order, are subsequently included in the MST, resulting in the MST shown in Figure 6.11b with a total length of 32 units.

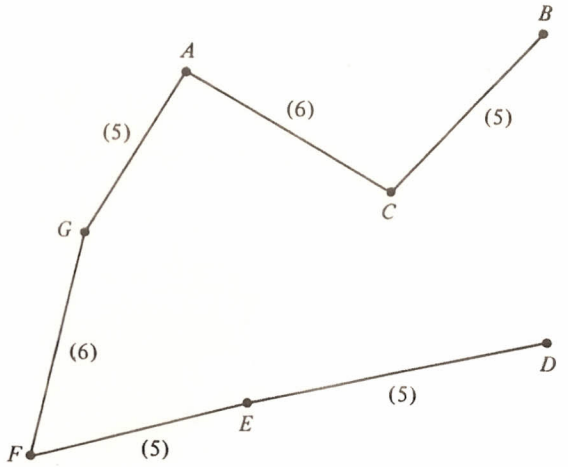
Obviously, the most time-consuming of our MST algorithm's steps is the one that involves finding the next node to be included in the MST. This, in effect, means comparing the lengths of all links leading from nodes already included in the MST to nodes not included in the MST to find the shortest one. Repeated application of this step is computationally expensive. To overcome this problem, an alternative algorithm calls for the ranking of all links in G according to length and then the addition to the MST at each stage of the shortest link such that no cycle is formed with previously chosen links. Unfortunately, this approach also suffers from a computational disadvantage: checking for cycles—a trivial task in the case of manual solutions—is nontrivial in the case of computer applications and becomes "expensive" for large problems.

Finally, we note that whenever, in the course of applying Algorithm 6.3, a tie arises (and is broken arbitrarily), this raises the possibility that two or more minimum spanning trees of the graph G exist. Although the total length of all of these tied MST's will be the same, they may differ markedly from each other in terms of which links each of them includes.

Exercise 6.2 Show that Algorithm 6.3 can be executed in $O(n^2)$ time.



(a)



(b)

FIGURE 6.11 A graph and its minimum spanning tree.

6.4 ROUTING PROBLEMS

One of the most common problems in urban service systems is the design of routes for vehicles or people. In some instances, these routes must be designed so that they traverse in an exhaustive way the streets in a neighborhood or in a specific part of a city or, occasionally, in a whole city. Alternatively, the objective may be to visit a set of given geographical points in a city in order to provide some service there or to deliver or collect goods.

Examples in which the first type of routing problem arises include the cleaning and sweeping of streets, the plowing of snow after a snowstorm, the delivery of mail to residences, and the collection of refuse from houses. On the other hand, the daily routing of school buses, the distribution of newspapers to newsstands and kiosks, the routine inspection of and coin collection from public telephone booths, and the delivery of mail packages to addressees are all illustrations of the point-visiting type of routing problem.

For obvious reasons, these two classes of problems are referred to as *edge-covering* and *node-covering* problems, respectively. Certain specific versions of these problems have, over the years, received extensive treatment in the mathematics and operations research literature. For instance, the famous "traveling salesman problem," the best-known and most straightforward (in terms of problem description) version of the node-covering class of problems, has been the subject of literally hundreds of scientific reports and papers.

Our main aim here is not so much to review such theoretical work as it is to provide some insights into the ways these problems can be solved, exactly or approximately. In the process we illustrate, as well, both the applicability and the limitations of the techniques that are being described.

6.4.1 Edge Covering: The Chinese Postman's Problem

Consider the case of a mailman who is responsible for the delivery of mail in the city area shown in graph form on Figure 6.12. The mailman must always begin his delivery route at node A where the post office is located; must traverse every single street in his area; and, eventually, must return to node A (while using only the street grid shown). The lengths of the edges of the graph (where each edge represents a street segment between street intersections indicated as nodes) are given on Figure 6.12. The graph is undirected.

The most natural question to ask in this case is: How should the mailman's route be designed to minimize the total distance he walks, while at the same time traversing every street segment at least once.[10]

This type of edge-covering problem is known as the *Chinese postman's problem.*[11] We shall discuss the problem only for undirected graphs and we use Problem 6.6 to extend our results to directed graphs.

The Chinese postman's problem (CPP) has an interesting history. It was examined in detail for the first time by the great Swiss mathematician and

[10] On the other hand, the mailman might instead wish to minimize the number of "pound-miles" per day. This may result in a distinctly different route.

[11] The name derives from the fact that an early paper discussing this problem appeared in the journal *Chinese Mathematics* [MEI 52].

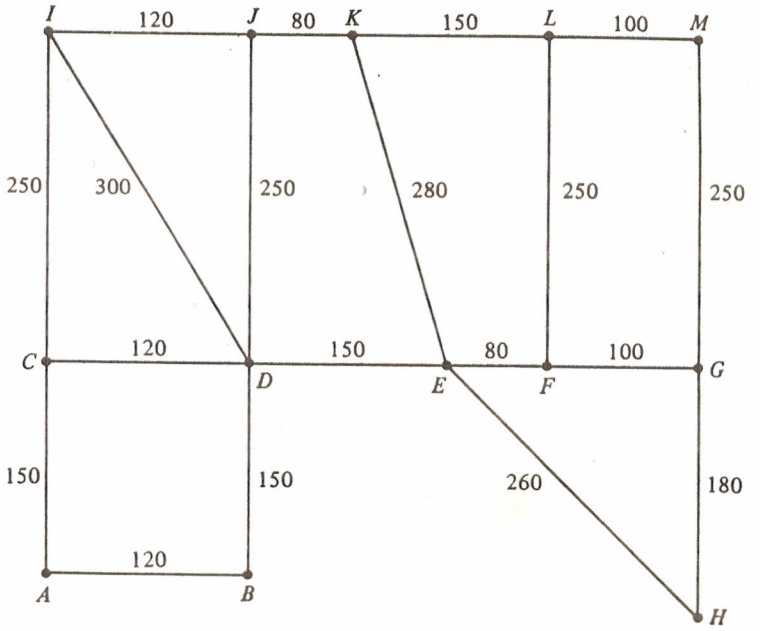


FIGURE 6.12 City area that must be traversed by a mailman.

physicist Leonhard Euler in the eighteenth century. Euler tried to find a way in which a parade procession could cross all seven bridges shown in Figure 6.13 *exactly once*. These seven bridges were at the Russian city of Königsberg (now Kaliningrad) on the river Pregel.

Euler proved in 1736 that no solution to the Königsberg routing problem exists. He also derived some general results that provide the motivation for the solution approaches to the CPP that have been devised recently.

At this time, efficient (i.e., polynomial time) algorithms exist for solving the CPP on *undirected* graphs and on *directed* graphs. Interestingly, a solution approach to the CPP on directed graphs (which, incidentally differs in important respects from the corresponding approach in the case of undirected

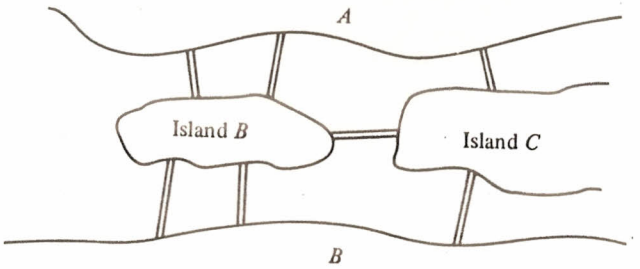


FIGURE 6.13 Seven-bridge problem at Königsberg.

graphs) was developed in the course of a project aimed at designing routes for street-sweeping vehicles in New York City [BELT 74]. After several researchers had spent a good amount of time trying to develop a similarly efficient procedure for solving the CPP on a *mixed* graph, it was finally shown [PAPA 76] that this last problem belongs to a class of very hard ("NP-complete") problems for which it is unlikely that polynomial algorithms will ever be found (see also Section 6.4.6)!

6.4.2 Chinese Postman's Problem on an Undirected Graph

Assume that an undirected graph $G(N, A)$ is given with known edge lengths $\ell(i, j)$ (> 0) for all edges $(i, j) \in A$. [As usual, $\ell(i, j)$ could also represent cost, travel time, and so on, associated with edge (i, j), depending on the context of the problem.] Then the CPP can be formally stated as follows:

> Find a circuit that will traverse every edge of G at least once and for which the sum
>
> $$\sum_{\text{all } (i,j) \in A} n(i, j)\ell(i, j) \text{ is minimum}$$
>
> where $n(i, j)$ is the number of times edge (i, j) is traversed.

The following definitions are now necessary for our subsequent discussion:

> *Definitions:* An Euler tour is a circuit which traverses every edge on a graph exactly once (beginning and terminating at the same node). An Euler path is a path which traverses every edge on a graph exactly once.

We then have the following fundamental result:

Euler's theorem: A connected graph G possesses an Euler tour (Euler path) if and only if G contains exactly zero (exactly two) nodes of odd degree.[12]

Figure 6.14 illustrates Euler's theorem. Figure 6.14a and b have no nodes of odd degree and must, therefore, possess an Euler tour. Such a tour, for instance, is the circuit $\{B, C, D, A, C, A, B\}$ for the graph of Figure 6.14b (note that the A–C–A portion of the tour can be traversed in either of two ways). Graphs 6.14c and d have exactly two nodes of odd degree. Thus, they must possess an Euler path according to the theorem. For instance, $\{A, B,$

[12] In the case of directed graphs, Euler's theorem requires that the indegree and outdegree of each node must be equal for an Euler tour to exist (see Problem 6.6).

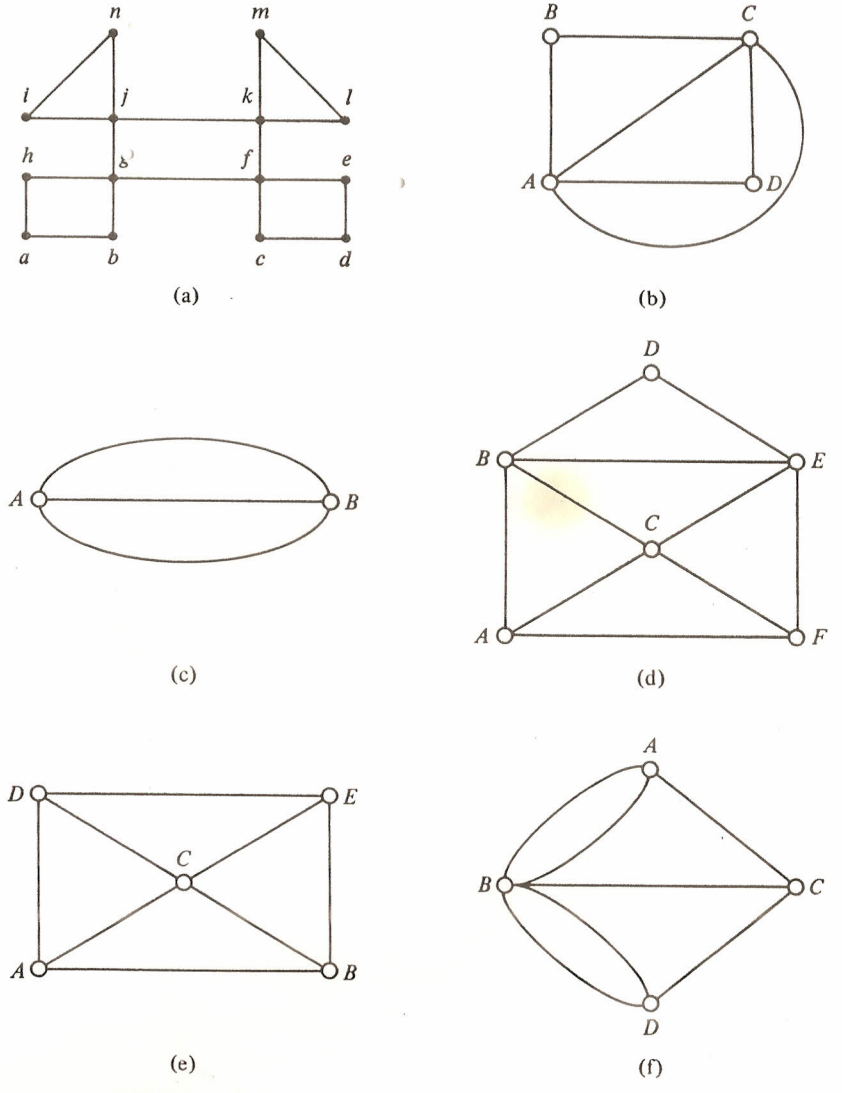


FIGURE 6.14 Some sample graphs that illustrate Euler's theorem.

$D, E, C, A, F, C, B, E, F\}$ is an Euler path for graph 6.14d. (By the way, Euler paths are simple but not always elementary.) However, it is impossible to find an Euler path that does not begin at A and terminate at F, or begin at F and terminate at A. Finally, graphs 6.14e and f have four nodes of odd degree, and thus neither an Euler tour nor an Euler path can be drawn on either. Note that graph 6.14f is a network model of the Königsberg problem with the seven bridges.

Proof of Euler's theorem: Assume that G has zero nodes of odd degree. It can then be shown that this is a necessary and a sufficient condition for an Euler tour to exist. It is necessary because any Euler tour drawn on the graph must always enter a node through some edge and leave through another and all edges on the graph must be used exactly once. Thus, an even number of incident edges is required for every node on the graph.

Sufficiency, on the other hand, can be shown through the following tour-construction argument. We begin at some initial node $j_0 \in N$ and draw a circuit through G (thus eventually returning to j_0). Let this circuit be denoted C_0. If C_0 happens to be an Euler tour, this is fine; we stop. If C_0 is not an Euler tour, then if we remove from G all edges used by circuit C_0, there must be some edges left over (not traversed by C_0). Moreover, at least two of these edges must be incident on some node j_1 through which circuit C_0 has passed. This must be so since, by assumption, G is, first, connected and, second, all its nodes are of even degree (and C_0 has only used up an even number of edges which are incident on j_1). Thus, it is possible to draw another circuit C_1, originating and terminating at j_1, which uses only edges of G', the graph left after we eliminate the edges of C_0 from G. If all edges in G' are used up by C_1, then we are finished: the Euler tour is the circuit that results from using the initial portion of C_0 to go from j_0 to j_1, then C_1 to return back to j_1, and finally the second portion of C_0 to go from j_1 to j_0. If, however, after eliminating the edges on C_0 and C_1 from G, there are still some edges left uncovered, then at least two of these edges must be incident on a node j_2 which is either on C_0 or on C_1 (or both if $j_2 = j_1$). Then, it should be possible to find a tour C_2, originating and terminating at j_2 and using only edges on G'', where G'' is the graph that results when the edges covered by C_0 and C_1 are eliminated from G.

This procedure may now be continued until eventually, say after the kth step, there will be no edges left uncovered. At that time, an Euler tour will also have been obtained which will be a combination of circuits $C_0, C_1, C_2, \ldots, C_k$. The tour is constructed in the manner explained above for C_0 and C_1.

It is interesting to note that the tour-construction approach that we have just outlined in proving sufficiency happens, in fact, to be one of the possible algorithms that can be used in practice to construct Euler tours on graphs.

> **Exercise 6.3** Prove the part of Euler's theorem which claims that an Euler path can be drawn on graphs with exactly two nodes of odd degree.

What are now the implications of Euler's theorem for the Chinese postman's problem? First, and obviously, if a graph possesses an Euler tour, then this tour is clearly the solution to the Chinese postman's problem. More important, however, as we shall now see, Euler's theorem provides the guidelines for solving the Chinese postman's problem, even for graphs on which an Euler tour does not exist. The solution approach basically consists of *augmenting these graphs in such a way that all odd-degree nodes are transformed to even-degree nodes*, which in turn means that an Euler tour can be drawn on the augmented graph.

Before discussing the details of this solution approach, we shall present an algorithm for drawing Euler tours on graphs where such tours are known to exist.

6.4.3 Obtaining an Euler Tour

Let $G(N, A)$ be a connected graph with no nodes of odd degree. Then an Euler tour can be drawn on G in the following manner.

Euler-Tour Algorithm (Algorithm 6.4)

Begin at any desired starting node $n_0 \in N$ and traverse successively the edges of G, keeping track of the route followed, and deleting from G each edge of G as it is traversed. Do not, at any particular point in this procedure, traverse an edge that is an *isthmus*. (An isthmus is an edge whose deletion at that particular point in the procedure would divide the yet undeleted part of G into two separate nonempty components.) Continue this procedure until all edges of G have been deleted, at which point the traversed route is an Euler tour and you have returned to the starting node, n_0.

> ***Example 6: Drawing an Euler Tour***
>
> We refer to the graph of Figure 6.14a. Assume that the starting node is node a. Suppose that we follow the route "a to b to g." At this point, edge (g, h) should not be traversed because it is an isthmus: if (g, h) is deleted, then the resulting undeleted part of G will consist of two separate nonempty components [edge (h, a) is separated from the rest of the undeleted part of G]. Thus, the next edge to be traversed should be either (g, j) or (g, f). The reader may wish to continue from here.

Algorithm 6.4 is very simple and convenient for manual applications. However, it is not particularly well suited for computer solutions. Whereas it is a trivial task for a human being to decide at each point whether a particular edge is or is not an isthmus, this is a time-consuming procedure for a computer. For that reason, alternative algorithms for obtaining Euler tours have been suggested for computer implementation. One of these algorithms, as we have already mentioned, is based on the procedure outlined in our earlier proof of Euler's theorem.

Exercise 6.4 Consider the procedure outlined in the proof of Euler's theorem and describe it in the form of a computer-programmable algorithm (possibly by drawing a flowchart).

From the description of Algorithm 6.4 it is clear that, in most cases, there exist many different Euler tours for any connected $G(N, A)$ with no nodes of odd degree, in the sense that the routes followed in traversing all the edges of G are different. However, all Euler tours of any given graph naturally have the same total length, equal to

$$\sum_{\text{all}(i,j)\in A} \ell(i,j)$$

The application of Algorithm 6.4 to finding an Euler path for graphs with two odd-degree nodes is also straightforward (one simply starts from one of the odd-degree nodes and ends up at the other).

6.4.4 Solving the Chinese Postman's Problem

We now return to the Chinese postman's problem and solve the problem of finding the minimum length, edge-covering tour of a graph $G(N, A)$ with no restrictions placed on G other than that it be connected and undirected.

The solution procedure consists of adding, in a judicious manner, artificial edges, parallel to the existing ones, to the original graph G to obtain a new graph $G^1(N, A^1)$ on which an Euler tour can be drawn. This means that the addition of the artificial edges should be such as to turn all odd-degree nodes on G into even-degree nodes on G^1. Moreover, this should be accomplished by selecting that combination of artificial edges which results in the minimum-length tour at the end. The addition of an artificial edge in parallel to an existing one in the original network simply means that the existing edge will have to be traversed twice in the final Chinese postman's tour.

An important observation is helpful at this point.

Lemma: The number of nodes of odd degree in an undirected graph G is always even.

Exercise 6.5 Prove the lemma.

Hint: The sum of the degrees of all the nodes in G is an even number since each edge is incident on two nodes.

We now describe the solution approach in general terms:

Chinese Postman Algorithm—Undirected Graph (Algorithm 6.5)

STEP 1: Identify all nodes of odd degree in $G(N, A)$. Say there are m of them, where m is an even number according to the lemma.

STEP 2: Find a minimum-length pairwise matching (see below) of the m odd-degree nodes and identify the $m/2$ shortest paths between the two nodes composing each of the $m/2$ pairs.

STEP 3: For each of the pairs of odd-degree nodes in the minimum-length pairwise matching found in Step 2, add to the graph $G(N, A)$ the edges of the shortest path between the two nodes in the pair. The graph $G^1(N, A^1)$ thus obtained contains no nodes of odd degree.

STEP 4: Find an Euler tour on $G^1(N, A^1)$. This Euler tour is an optimal solution to the Chinese postman's problem on the original graph $G(N, A)$. The length of the optimal tour is equal to the total length of the edges in $G(N, A)$ plus the total length of the edges in the minimum-length matching.

A pairwise matching of a subset $N' \subset N$ of nodes of a graph G is a pairing of all the nodes in N' (assuming that the number of nodes in N' is even). A minimum-length pairwise matching of the nodes in N' is then a matching such that the total length of the shortest paths between the paired nodes is minimum.

Example 7: Application of Algorithm 6.5

Consider the graph of Figure 6.15a. The numbers on the edges indicate the lengths of the edges. The graph contains four nodes of odd degree (a, b, d, and e). Thus, there are three possible matchings of the odd-degree nodes: a–b and d–e; a–d and b–e; and a–e and b–d. The augmented networks that would result from the addition of the artificial edges corresponding to each of these three matchings are shown in Figure 6.15b–d, respectively. Of the three matchings, the optimal is obviously the one shown in Figure 6.15c. It adds only 12 units of length to the tour as opposed to 16 for Figure 6.15b and 20 for 6.15c. Thus, the graph shown on Figure 6.15c should be the result of the procedure outlined in Steps 2 and 3 of Algorithm 6.5. Supposing that the required tour must begin at node a, a solution to the Chinese postman's problem for the graph of Figure 6.15a is the tour $\{a, d, a, c, d, e, c, b, e, b, a\}$. Its total length is 60 units, of which 48 is the total length of the graph 6.15a, while 12 units are due to the artificial edges. In other words, edges (a, d) and (b, e) will be traversed twice.

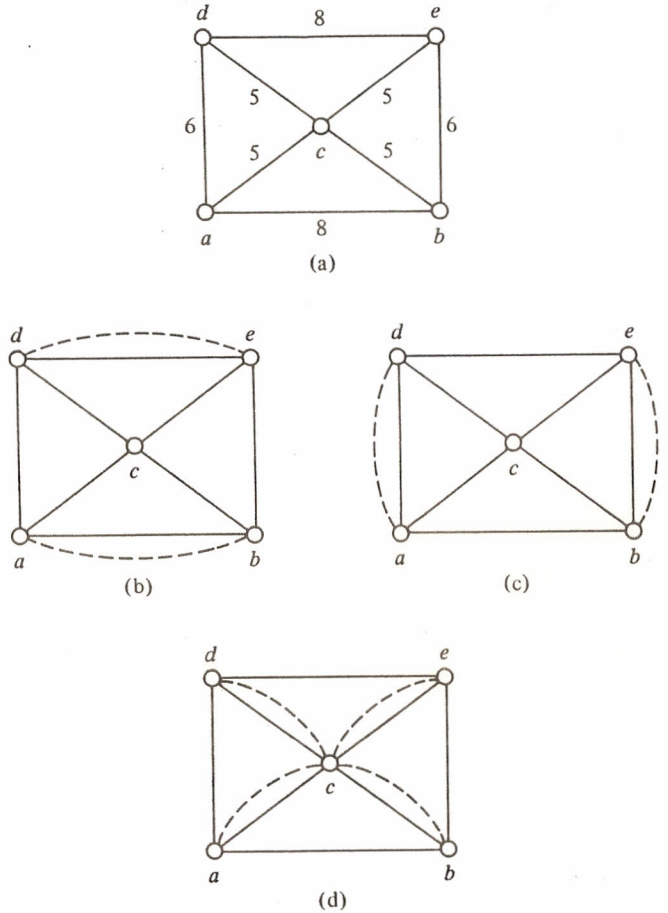


FIGURE 6.15 Illustration of the application of Algorithm 6.5.

In Algorithm 6.5 we have concentrated exclusively on the *shortest paths* between pairs of odd-degree nodes. That this should be so can be seen as follows. Were a supposedly optimal pairwise matching between odd-degree nodes to contain a path, P^1, between any given pair of nodes that is not the shortest path, P, between these two nodes, the solution could be improved by simply substituting P for P^1 in the drawing of the artificial edges, since the length of P is, by definition, smaller than the length of P^1.

In reviewing Algorithm 6.5, we can see that its only difficulty lies in Step 2, since Steps 1, 3, and 4 are all very simple. For Step 2, however, the number of possible pairwise matching combinations increases very quickly with the number of odd-degree nodes. Some thought will show that with m odd-

degree nodes in G (remember that m is always even), there are

$$S = \prod_{i=1}^{m/2} (2i - 1) \tag{6.4}$$

possible distinct pairwise matching combinations. Thus, for $m = 4$, there are $1 \cdot 3 = 3$ possible combinations (as we have already seen in our last example), for $m = 10$, 945 combinations, and for $m = 20$, about 655 million combinations.

Fortunately, an efficient, but quite complicated algorithm for minimum-length matchings on undirected graphs is now available. This algorithm is based on the theory of matching on graphs that has been developed in recent years primarily by J. Edmonds. The interested reader is referred to [EDMO 73] or to [CHRI 75] for its details. Here we only note that the algorithm is an exact one (i.e., finds the optimum matchings) and efficient computer-implemented versions solve the problem in time proportional to n^3, where n is the number of nodes in the graph under consideration. The algorithm not only finds the minimum-length matchings but also identifies the shortest path associated with each pair. It follows that in the worst case Algorithm 6.5 is also $O(n^3)$.

Manual approach to matching. When Edmonds' minimum-length matching algorithm is used in Step 2 of Algorithm 6.5, an optimum solution to the Chinese postman's problem is obtained. Experience has shown, however, that whenever the Chinese postman problem is posed in a geographical context (and this of course is the case in urban service systems applications), then excellent—in the sense of being very close to the optimum—manual solutions can be obtained practically by inspection with the aid of a good map. The map can be used to perform all four steps of Algorithm 6.5, and the only step where one risks deviating from the optimum solution is in finding the minimum-length matching in Step 2.

The manual search for an approximately minimum-length matching of odd-degree nodes in Step 2 is greatly aided by the following key observation:

> *In a minimum-length matching of the odd-degree nodes, no two shortest paths in the matching can have any edges in common.*

To see this, consider Figure 6.16, assuming that nodes a, b, c, and d are all of odd degree and that the shortest paths between, say, the pairs of nodes a–c and b–d have edge (p, q) in common. Then it is impossible that the minimum-length pairwise matching contain the pairs a–c and b–d, since it would then be possible to obtain a better (shorter) matching by coupling

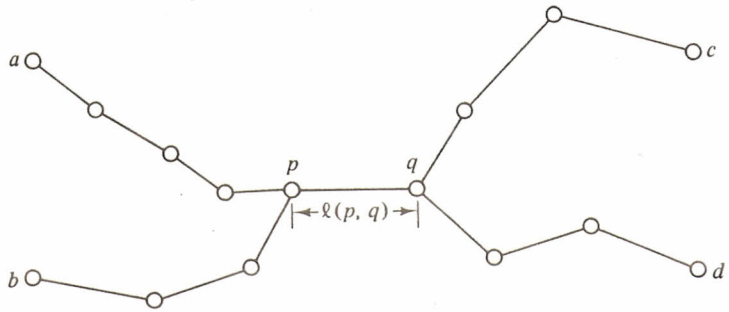


FIGURE 6.16 In an optimum pairwise matching, node *a* can only be matched with node *b* and node *c* with node *d*.

together nodes a and b into one pair and nodes c and d into another. This latter matching will reduce the sum of the shortest path lengths by at least $2\ell(p, q)$ units.

From the practical point of view, this observation has two effects:

1. It eliminates from consideration a very large percentage of the possible sets of matchings that might be considered. (This percentage might be expected to increase as the number of odd-degree nodes increases.)
2. It indicates that in a minimum-length matching odd-degree nodes will be matched to other odd-degree nodes in their immediate neighborhood.

Other observations (of a somewhat more complicated nature) have also been made (see [STRI 70]) with regard to finding good approximate solutions to minimum-length matching problems without using a matching algorithm. It has been the authors' experience that, whenever good maps of urban areas are available, it takes only a limited amount of practice with some examples before one develops the ability to find *virtually by inspection* excellent solutions to matching problems (by taking advantage of the foregoing observations).

Example 8

Consider again our initial Chinese Postman problem shown in Figure 6.12. The odd-degree nodes on Figure 6.12 are C, D, F, G, I, J, K, and L. They are shown on Figure 6.17, with that part of the network model of the district that contains all the shortest paths between the odd-degree nodes.

Inspection of Figure 6.17 leads to the conclusion that (although theoretically there are $7 \cdot 5 \cdot 3 \cdot 1 = 105$ possible sets of matchings among the eight

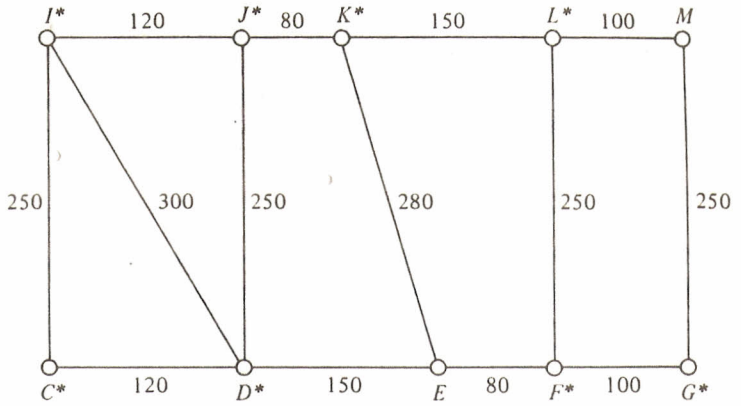


FIGURE 6.17 Odd-degree nodes from the graph of Figure 6.12 are indicated with a *.

odd-degree nodes) very few sets of matchings would even qualify for consideration. In fact, a couple of trials should be sufficient to convince the reader that the minimum-cost matching can only be the matching I–J, K–L, C–D, and F–G for a total cost (i.e., duplicated street length) of 490 units of length. The final result for the Chinese postman's problem in this case is then shown on Figure 6.18, where edges to be traversed twice have been substituted by two edges (or pseudo-edges) each of equal length to the original one. The graph of Figure 6.18 now contains no nodes of odd degree, and thus an Euler tour can be drawn on it, beginning at any desired node and ending at the same node.

The length of all Euler tours on this graph will be equal to the total length of the original graph G (3,340 units) plus the distance to be traversed twice (490 units), for a total length of 3,830 units, or about 15 percent more than the total street length of the district that the mailman is responsible for. (This turns out to be the optimum solution.)

6.4.5 Node Covering: The Traveling Salesman Problem

When deliveries, collections, or visits must be made to (or from) a number of specific (and, often, widely separated) points, the routing problem that must be solved becomes a node-covering one. The demand (or supply) points can then be represented as the nodes on the network model of the urban transportation grid and the question of the order in which to visit these nodes so as to achieve some objective is then addressed.

The most fundamental and best known of all node-covering problems is the *traveling salesman problem* (TSP):

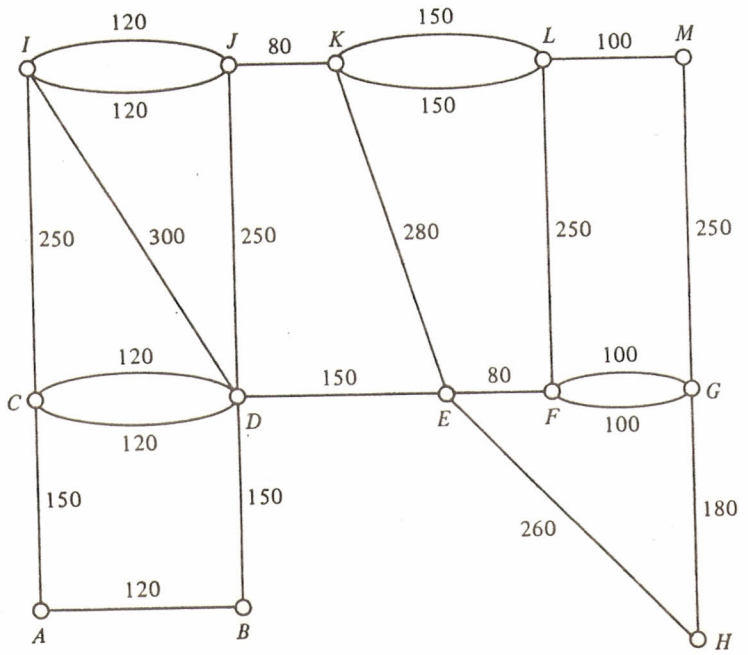


FIGURE 6.18 Optimal solution to the CPP of Figure 6.12. Edges (C, D), (F, G), (K, L), and (I, J) must be transversed twice, as indicated.

Find the minimum distance route that begins at a given node of a network, visits all the members of a specified set of nodes on the network at least once, and returns eventually to the initial node.

This statement of the TSP is more general than the one most commonly used. The latter specifies that each node must be visited *exactly* once. However, this implicitly assumes that it is possible to find a tour with such a property. This is not, for instance, the case in the network of Figure 6.19, where in order to visit node A, one must visit node B twice.

We shall be particularly interested in solving the TSP in the following context: $n - 1$ points must be visited by a vehicle which must begin and finish its tour at another specific point (depot). The transportation network that interconnects these n (in total) points is *completely connected*, that is it is possible to go directly from any point to any other point without passing through any of the other points in the set. The shortest distances between all pairs of points are equal to the lengths of the direct links between the two points, i.e. if i and j are any two of the n points then $d(i,j) = \ell(i,j)$. This implies that our network (or travel metric) satisfies the *triangular inequality* $\ell(i,j) \leq \ell(i,k) + \ell(k,j)$ for any three points i, j, and k. Finally, the shortest

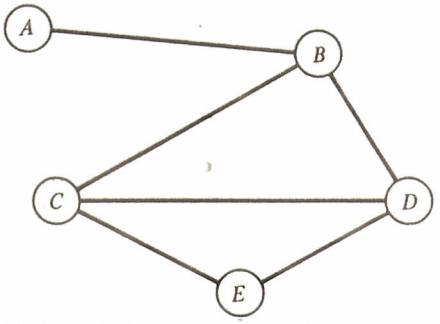


FIGURE 6.19 Network on which a traveling salesman tour must visit at least one node (node B) twice.

distance matrix is assumed to be symmetric. The applicability of this version of the TSP to continuum models and to network models of the urban environment is clear.[13] The complete connectivity of the network and the triangular inequality assure us that a shortest length traveling salesman tour through the n points can be found such that each point is visited exactly once.

That the TSP may be a very difficult problem to solve can now be seen by the fact that in our (simplified) version of the TSP there are $(n - 1)!$ different orderings of the points to be visited, that is, $(n - 1)!/2$ different solutions of the TSP (since each tour can be run in either of two directions). Thus, a TSP with only 10 nodes has 1,814,400 possible solutions, and one with 20 nodes about 1.2×10^{18}.

The TSP has indeed turned out to be an extremely difficult problem. While several exact algorithms (i.e., algorithms that are guaranteed to terminate with an optimal solution) have been devised for the TSP over the years, none of them is efficient, in the sense of being a polynomial-time algorithms. In fact, the TSP is considered by mathematicians and computer scientists as a problem which is so difficult intrinsically that no efficient algorithm will probably ever be found to solve it.[14] (On the other hand, no one has yet proven that no efficient algorithm for the TSP exists, so there is still hope!)

Of the existing exact algorithms, an early one [HELD 62] based on the idea of dynamic programming is $O(n^2 2^n)$, where n is the number of points in the tour. As if this were not bad enough, the algorithm also requires computer storage proportional to $n2^n$. Thus, even the largest available computers

[13] Note that this also includes directed networks (i.e., cases with one-way streets) as long as, for the n points of interest, $d(i,j) = d(j,i)$.

[14] The TSP is the prototypical one of the special class of difficult combinatorial problems called NP-complete problems.