2

PATHS, TREES, AND CYCLES

I hate definitions.
—Benjamin Disraeli

Chapter Outline

2.1 INTRODUCTION

Because graphs and networks arise everywhere and in a variety of alternative forms, several professional disciplines have contributed important ideas to the evolution of network flows. This diversity has yielded numerous benefits, including the infusion of many rich and varied perspectives. It has also, however, imposed costs: For example, the literature on networks and graph theory lacks unity and authors have adopted a wide variety of conventions, customs, and notation. If we so desired, we could formulate network flow problems in several different standard forms and could use many alternative sets of definitions and terminology. We have chosen to adopt a set of common, but not uniformly accepted, definitions: for example, arcs and nodes instead of edges and vertices (or points). We have also chosen to use models with capacitated arcs and with exogenous supplies and demands at the nodes. The circulation problem we introduced in Chapter 1, without exogenous supplies and demands, is an alternative model and so is the capacitated transportation problem. Another special case is the uncapacitated network flow problem. In Chapter 1 we viewed each of these models as special cases of the minimum cost network flow problem. Perhaps somewhat surprisingly, we could have started with any of these models and shown that all the others were special cases. In this sense, each of these models offers another way to capture the mathematical essence of network flows.

In this chapter we have three objectives. First, we bring together many basic definitions of network flows and graph theory, and in doing so, we set the notation that we will be using throughout this book. Second, we introduce several different data structures used to represent networks within a computer and discuss the relative advantages and disadvantages of each of these structures. In a very real sense, data structures are the life blood of most network flow algorithms, and choosing among alternative data structures can greatly influence the efficiency of an algorithm, both

in practice and in theory. Consequently, it is important to have a good understanding of the various available data structures and an idea of how and when to use them. Third, we discuss a number of different ways to transform a network flow problem and obtain an equivalent model. For example, we show how to eliminate flow bounds and formulate any model as an uncapacitated problem. As another example, we show how to formulate the minimum cost flow problem as a transportation problem (i.e., how to define it over a bipartite graph). This discussion is of theoretical interest, because it establishes the equivalence between several alternative models and therefore shows that by developing algorithms and theory for any particular model, we will have at hand algorithms and theory for several other models. That is, our results enjoy a certain universality. This development is also of practical value since on various occasions throughout our discussion in this book we will find it more convenient to work with one modeling assumption rather than another—our discussion of network transformations shows that there is no loss in generality in doing so. Moreover, since algorithms developed for one set of modeling assumptions also apply to models formulated in other ways, this discussion provides us with one very reassuring fact: We need not develop separate computer implementations for every alternative formulation, since by using the transformations, we can use an algorithm developed for any one model to solve any problem formulated as one of the alternative models.

We might note that many of the definitions we introduce in this chapter are quite intuitive, and much of our subsequent discussion does not require a complete understanding of all the material in this chapter. Therefore, the reader might simply wish to skim this chapter on first reading to develop a general overview of its content and then return to the chapter on an "as needed" basis later as we draw on the concepts introduced at this point.

2.2 NOTATION AND DEFINITIONS

In this section we give several basic definitions from graph theory and present some basic notation. We also state some elementary properties of graphs. We begin by defining directed and undirected graphs.

Directed Graphs and Networks: A *directed graph* $G = (N, A)$ consists of a set N of nodes and a set A of arcs whose elements are ordered pairs of distinct nodes. Figure 2.1 gives an example of a directed graph. For this graph, $N = \{1, 2, 3, 4, 5, 6, 7\}$ and $A = \{(1, 2), (1, 3), (2, 3), (2, 4), (3, 6), (4, 5), (4, 7), (5, 2), (5, 3), (5, 7), (6, 7)\}$. A *directed network* is a directed graph whose nodes and/or arcs have associated numerical values (typically,

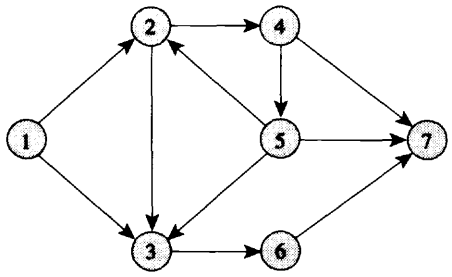


Figure 2.1 Directed graph.

costs, capacities, and/or supplies and demands). In this book we often make no distinction between graphs and networks, so we use the terms "graph" and "network" synonymously. As before, we let n denote the number of nodes and m denote the number of arcs in G.

Undirected Graphs and Networks: We define an undirected graph in the same manner as we define a directed graph except that arcs are unordered pairs of distinct nodes. Figure 2.2 gives an example of an undirected graph. In an undirected graph, we can refer to an arc joining the node pair i and j as either (i, j) or (j, i). An undirected arc (i, j) can be regarded as a two-way street with flow permitted in both directions: either from node i to node j or from node j to node i. On the other hand, a directed arc (i, j) behaves like a one-way street and permits flow only from node i to node j.

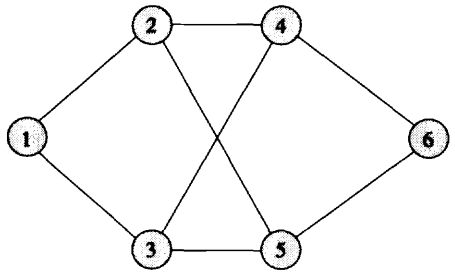


Figure 2.2 Undirected graph.

In most of the material in this book, we assume that the underlying network is directed. Therefore, we present our subsequent notation and definitions for directed networks. The corresponding definitions for undirected networks should be transparent to the reader; nevertheless, we comment briefly on some definitions for undirected networks at the end of this section.

Tails and Heads: A directed arc (i, j) has two *endpoints* i and j. We refer to node i as the *tail* of arc (i, j) and node j as its *head*. We say that the arc (i, j) *emanates* from node i and *terminates* at node j. An arc (i, j) is *incident to* nodes i and j. The arc (i, j) is an *outgoing arc* of node i and an *incoming arc* of node j. Whenever an arc $(i, j) \in A$, we say that node j is *adjacent* to node i.

Degrees: The *indegree* of a node is the number of incoming arcs of that node and its *outdegree* is the number of its outgoing arcs. The *degree* of a node is the sum of its indegree and outdegree. For example, in Figure 2.1, node 3 has an indegree of 3, an outdegree of 1, and a degree of 4. It is easy to see that the sum of indegrees of all nodes equals the sum of outdegrees of all nodes and both are equal to the number of arcs m in the network.

Adjacency List: The *arc adjacency list* $A(i)$ of a node i is the set of arcs emanating from that node, that is, $A(i) = \{(i, j) \in A : j \in N\}$. The *node adjacency list* $A(i)$ is the set of nodes adjacent to that node; in this case, $A(i) = \{j \in N : (i, j) \in A\}$. Often, we shall omit the terms "arc" and "node" and simply refer to the adjacency list; in all cases it will be clear from context whether we mean arc adjacency list or node adjacency list. We assume that arcs in the adjacency list $A(i)$ are arranged so that the head nodes of arcs are in increasing order. Notice that $| A(i) |$ equals the outdegree of node i. Since the sum of all node outdegrees equals m, we immediately obtain the following property:

Property 2.1. $\sum_{i \in N} | A(i) | = m.$

Multiarcs and Loops: *Multiarcs* are two or more arcs with the same tail and head nodes. A *loop* is an arc whose tail node is the same as its head node. In most of the chapters in this book, we assume that graphs contain no multiarcs or loops.

Subgraph: A graph $G' = (N', A')$ is a *subgraph* of $G = (N, A)$ if $N' \subseteq N$ and $A' \subseteq A$. We say that $G' = (N', A')$ is the subgraph of G *induced* by N' if A' contains each arc of A with both endpoints in N'. A graph $G' = (N', A')$ is a *spanning subgraph* of $G = (N, A)$ if $N' = N$ and $A' \subseteq A$.

Walk: A *walk* in a directed graph $G = (N, A)$ is a subgraph of G consisting of a sequence of nodes and arcs $i_1 - a_1 - i_2 - a_2 - \cdots - i_{r-1} - a_{r-1} - i_r$ satisfying the property that for all $1 \le k \le r - 1$, either $a_k = (i_k, i_{k+1}) \in A$ or $a_k = (i_{k+1}, i_k) \in A$. Alternatively, we shall sometimes refer to a walk as a set of (sequence of) arcs (or of nodes) without any explicit mention of the nodes (without explicit mention of arcs). We illustrate this definition using the graph shown in Figure 2.1. Figure 2.3(a) and (b) illustrates two walks in this graph: 1–2–5–7 and 1–2–4–5–2–3.

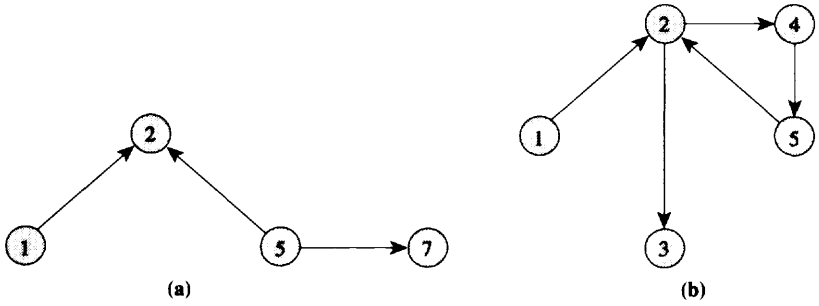


Figure 2.3 Examples of walks.

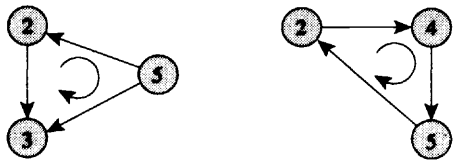
Directed Walk: A *directed walk* is an "oriented" version of a walk in the sense that for any two consecutive nodes i_k and i_{k+1} on the walk, $(i_k, i_{k+1}) \in A$. The walk shown in Figure 2.3(a) is not directed; the walk shown in Figure 2.3(b) is directed.

Path: A *path* is a walk without any repetition of nodes. The walk shown in Figure 2.3(a) is also a path, but the walk shown in Figure 2.3(b) is not because it repeats node 2 twice. We can partition the arcs of a path into two groups: forward arcs and backward arcs. An arc (i, j) in the path is a *forward arc* if the path visits node i prior to visiting node j, and is a *backward arc* otherwise. For example, in the path shown in Figure 2.3(a), the arcs (1, 2) and (5, 7) are forward arcs and the arc (5, 2) is a backward arc.

Directed Path: A *directed path* is a directed walk without any repetition of nodes. In other words, a directed path has no backward arcs. We can store a path (or a directed path) easily within a computer by defining a *predecessor* index *pred(j)* for every node j in the path. If i and j are two consecutive nodes on the path (along its orientation), pred(j) = i. For the path 1–2–5–7 shown in Figure 2.3(a), pred(7) = 5, pred(5) = 2, pred(2) = 1, and pred(1) = 0. (Frequently, we shall use the convention of setting the predecessor index of the initial node of a path equal to zero to indicate the beginning of the path.) Notice that we cannot use predecessor indices to store a walk since a walk may visit a node more than once, and a single predecessor index of a node cannot store the multiple predecessors of any node that a walk visits more than once.

Cycle: A *cycle* is a path $i_1 - i_2 - \cdots - i_r$ together with the arc (i_r, i_1) or (i_1, i_r). We shall often refer to a cycle using the notation $i_1 - i_2 - \cdots - i_r - i_1$. Just as we did for paths, we can define forward and backward arcs in a cycle. In Figure 2.4(a) the arcs (5, 3) and (3, 2) are forward arcs and the arc (5, 2) is a backward arc of the cycle 2–5–3.

Directed Cycle: A *directed cycle* is a directed path $i_1 - i_2 - \cdots - i_r$ together with the arc (i_r, i_1). The graph shown in Figure 2.4(a) is a cycle, but not a directed cycle; the graph shown in Figure 2.4(b) is a directed cycle.

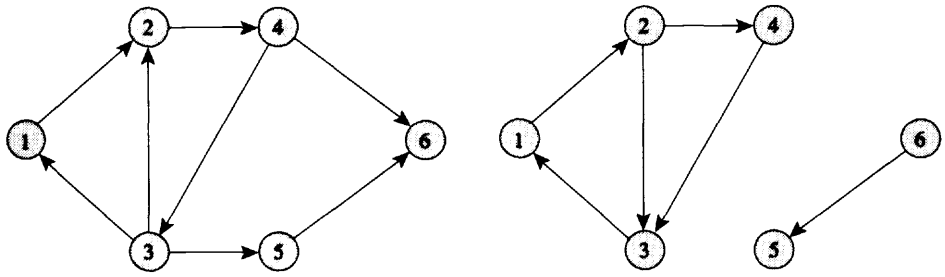


(a) (b)

Figure 2.4 Examples of cycles.

Acyclic Graph: A graph is a *acyclic* if it contains no directed cycle.

Connectivity: We will say that two nodes i and j are *connected* if the graph contains at least one path from node i to node j. A graph is *connected* if every pair of its nodes is connected; otherwise, the graph is *disconnected*. We refer to the maximal connected subgraphs of a disconnected network as its *components*. For instance, the graph shown in Figure 2.5(a) is connected, and the graph shown in Figure 2.5(b) is disconnected. The latter graph has two components consisting of the node sets {1, 2, 3, 4} and {5, 6}. In Section 3.4 we describe a method for determining whether a graph is connected or not, and in Exercise 3.41 we discuss a method for identifying all components of a graph.



(a) (b)

Figure 2.5 (a) Connected and (b) disconnected graphs.

Strong Connectivity: A connected graph is *strongly connected* if it contains at least one *directed* path from every node to every other node. In Figure 2.5(a) the component [see Figure 2.5(b)] defined on the node set {1, 2, 3, 4} is strongly connected; the component defined by the node set {5, 6} is not strongly connected because it contains no directed path from node 5 to node 6. In Section 3.4 we describe a method for determining whether or not a graph is strongly connected.

Cut: A *cut* is a partition of the node set N into two parts, S and $\bar{S} = N - S$. Each cut defines a set of arcs consisting of those arcs that have one endpoint in S and another endpoint in $\bar{S}$. Therefore, we refer to this set of arcs as a cut and represent it by the notation $[S, \bar{S}]$. Figure 2.6 illustrates a cut with $S = \{1, 2, 3\}$ and $\bar{S} = \{4, 5, 6, 7\}$. The set of arcs in this cut are $\{(2, 4), (5, 2), (5, 3), (3, 6)\}$.

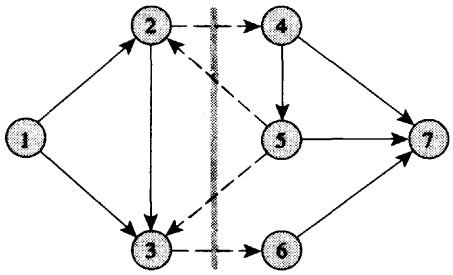


Figure 2.6 Cut.

s–t Cut: An *s–t cut* is defined with respect to two distinguished nodes s and t, and is a cut $[S, \bar{S}]$ satisfying the property that $s \in S$ and $t \in \bar{S}$. For instance, if $s = 1$ and $t = 6$, the cut depicted in Figure 2.6 is an s–t cut; but if $s = 1$ and $t = 3$, this cut is not an s–t cut.

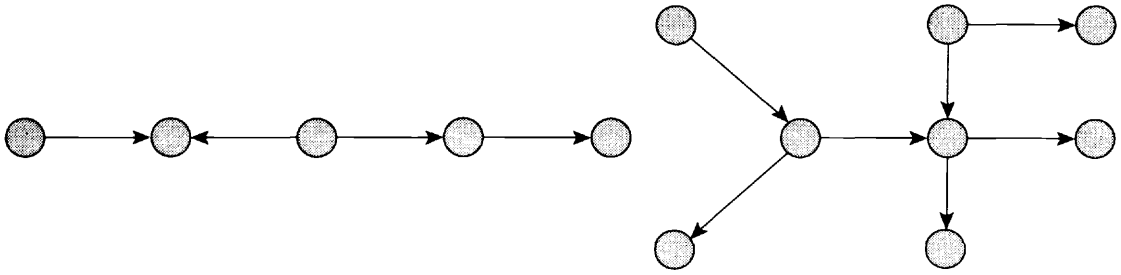


Figure 2.7 Example of two trees.

Tree. A *tree* is a connected graph that contains no cycle. Figure 2.7 shows two examples of trees.

A tree is a very important graph theoretic concept that arises in a variety of network flow algorithms studied in this book. In our subsequent discussion in later chapters, we use some of the following elementary properties of trees.

Property 2.2

(a) *A tree on n nodes contains exactly $n - 1$ arcs.*

(b) *A tree has at least two leaf nodes (i.e., nodes with degree 1).*

(c) *Every two nodes of a tree are connected by a unique path.*

Proof. See Exercise 2.13.

Forest: A graph that contains no cycle is a *forest*. Alternatively, a forest is a collection of trees. Figure 2.8 gives an example of a forest.

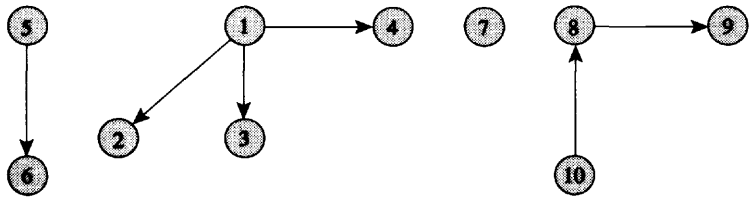


Figure 2.8 Forest.

Subtree: A connected subgraph of a tree is a *subtree.*

Rooted Tree: A rooted tree is a tree with a specially designated node, called its *root*; we regard a rooted tree as though it were hanging from its root. Figure 2.9 gives an instance of a rooted tree; in this instance, node 1 is the root node.

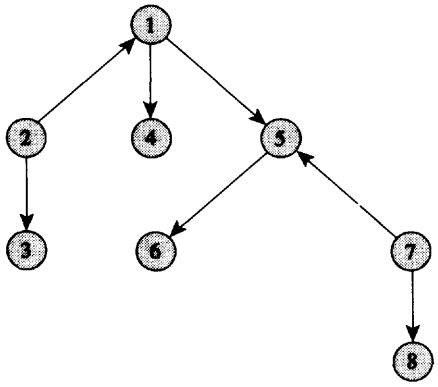


Figure 2.9 Rooted tree.

We often view the arcs in a rooted tree as defining predecessor–successor (or parent–child) relationships. For example, in Figure 2.9, node 5 is the predecessor of nodes 6 and 7, and node 1 is the predecessor of nodes 2, 4, and 5. Each node i (except the root node) has a unique predecessor, which is the next node on the unique path in the tree from that node to the root; we store the predecessor of node i using a predecessor index *pred(i)*. If j = pred(i), we say that node j is the predecessor of node i and node i is a successor of node j. These predecessor indices uniquely define a rooted tree and also allow us to trace out the unique path from any node back to the root. The *descendants* of a node i consist of the node itself, its successors, successors of its successors, and so on. For example, in Figure 2.9 the node set {5, 6, 7, 8} is the set of descendants of node 5. We say that a node is an *ancestor* of all of its descendants. For example, in the same figure, node 2 is an ancestor of itself and node 3.

In this book we occasionally use two special type of rooted trees, called a *directed in-tree* and a *directed out-tree.*

Directed-Out-Tree: A tree is a *directed out-tree routed at node s* if the unique path in the tree from node s to every other node is a directed path. Figure 2.10(a) shows an instance of a directed out-tree rooted at node 1. Observe that every node in the directed out-tree (except node 1) has indegree 1.

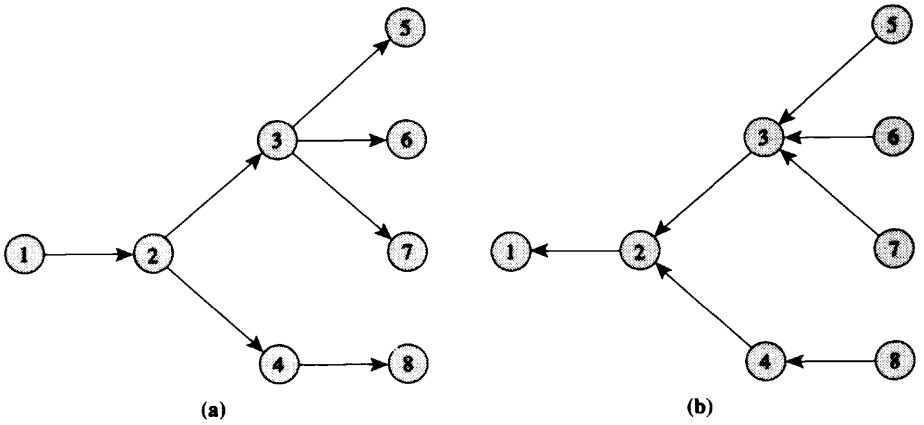


Figure 2.10 Instances of directed out-tree and directed in-tree.

Directed-In-Tree: A tree is a *directed in-tree routed at node s* if the unique path in the tree from any node to node s is a directed path. Figure 2.10(b) shows an instance of a directed in-tree rooted at node 1. Observe that every node in the directed in-tree (except node 1) has outdegree 1.

Spanning Tree: A tree T is a spanning tree of G if T is a spanning subgraph of G. Figure 2.11 shows two spanning trees of the graph shown in Figure 2.1. Every spanning tree of a connected n-node graph G has $(n - 1)$ arcs. We refer to the arcs belonging to a spanning tree T as *tree arcs* and arcs not belonging to T as *nontree arcs*.

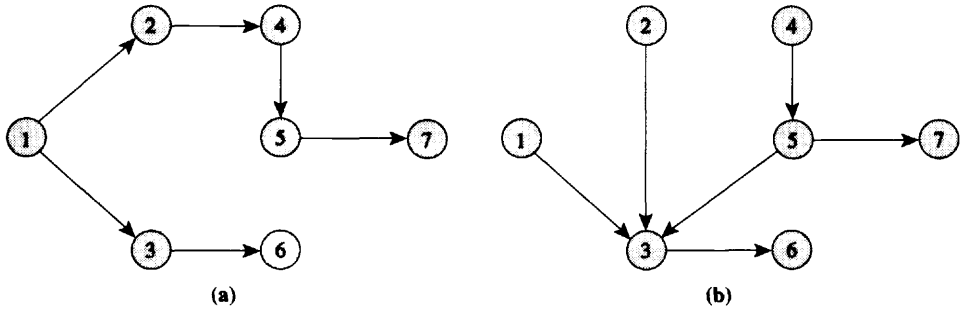


Figure 2.11 Two spanning trees of the network in Figure 2.1.

Fundamental Cycles: Let T be a spanning tree of the graph G. The addition of any nontree arc to the spanning tree T creates exactly one cycle. We refer to any such cycle as a *fundamental cycle* of G with respect to the tree T. Since the network contains $m - n + 1$ nontree arcs, it has $m - n + 1$ fundamental cycles. Observe that if we delete any arc in a fundamental cycle, we again obtain a spanning tree.

Fundamental Cuts: Let T be a spanning tree of the graph G. The deletion of any tree arc of the spanning tree T produces a disconnected graph containing two subtrees T_1 and T_2. Arcs whose endpoints belong to the different subtrees constitute a cut. We refer to any such cut as a *fundamental cut* of G with respect to the tree T. Since a spanning tree contains $n - 1$ arcs, the network has $n - 1$ fundamental cuts with respect to any tree. Observe that when we add any arc in the fundamental cut to the two subtrees T_1 and T_2, we again obtain a spanning tree.

Bipartite Graph: A graph $G = (N, A)$ is a *bipartite graph* if we can partition its node set into two subsets N_1 and N_2 so that for each arc (i, j) in A either (i) $i \in N_1$ and $j \in N_2$, or (ii) $i \in N_2$ and $j \in N_1$. Figure 2.12 gives two examples of bipartite graphs. Although it might not be immediately evident whether or not the graph in Figure 2.12(b) is bipartite, if we define $N_1 = \{1, 2, 3, 4\}$ and $N_2 = \{5, 6, 7, 8\}$, we see that it is.

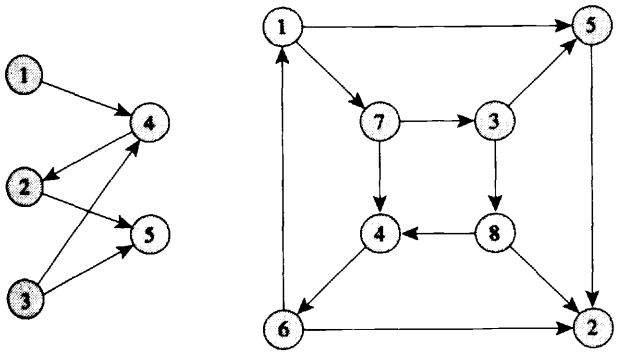


Figure 2.12 Examples of bipartite graphs.

Frequently, we wish to discover whether or not a given graph is bipartite. Fortunately, there is a very simple method for resolving this issue. We discuss this method in Exercise 3.42, which is based on the following well-known characterization of bipartite graphs.

Property 2.3. *A graph G is a bipartite graph if and only if every cycle in G contains an even number of arcs.*

Proof. See Exercise 2.21.

Definitions for undirected networks. The definitions for directed networks easily translate into those for undirected networks. An undirected arc (i, j) has two endpoints, i and j, but its tail and head nodes are undefined. If the network contains the arc (i, j), node i is adjacent to node j, and node j is adjacent to node i. The arc adjacency list (as well as the node adjacency list) is defined similarly except that arc (i, j) appears in $A(i)$ as well as $A(j)$. Consequently, $\sum_{i \in N} | A(i) | = 2m$. The degree of a node is the number of nodes adjacent to node i. Each of the graph theoretic concepts we have defined so far—walks, paths, cycles, cuts and trees—has essentially the same definition for undirected networks except that we do not distinguish between a path and a directed path, a cycle and a directed cycle, and so on.

2.3 NETWORK REPRESENTATIONS

The performance of a network algorithm depends not only on the algorithm, but also on the manner used to represent the network within a computer and the storage scheme used for maintaining and updating the intermediate results. By representing

a network more cleverly and by using improved data structures, we can often improve the running time of an algorithm. In this section we discuss some popular ways of representing a network. In representing a network, we typically need to store two types of information: (1) the network topology, that is, the network's node and arc structure; and (2) data such as costs, capacities, and supplies/demands associated with the network's nodes and arcs. As we will see, usually the scheme we use to store the network's topology will suggest a natural way for storing the associated node and arc information. In this section we describe in detail representations for directed graphs. The corresponding representations for undirected networks should be apparent to the reader. At the end of the section, however, we briefly discuss representations for undirected networks.

Node–Arc Incidence Matrix

The *node–arc incidence matrix* representation, or simply the *incidence matrix* representation, represents a network as the constraint matrix of the minimum cost flow problem that we discussed in Section 1.2. This representation stores the network as an $n \times m$ matrix $\mathcal{N}$ which contains one row for each node of the network and one column for each arc. The column corresponding to arc (i, j) has only two nonzero elements: It has a $+1$ in the row corresponding to node i and a -1 in the row corresponding to node j. Figure 2.14 gives this representation for the network shown in Figure 2.13.

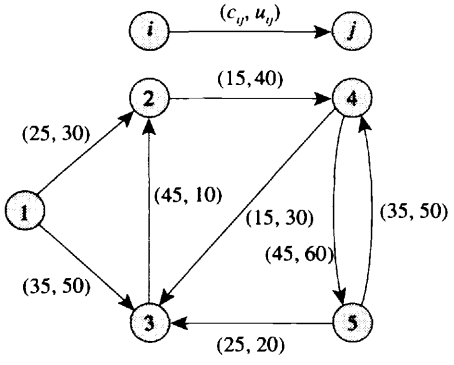


Figure 2.13 Network example.

	(1, 2)	(1, 3)	(2, 4)	(3, 2)	(4, 3)	(4, 5)	(5, 3)	(5, 4)
1	1	1	0	0	0	0	0	0
2	−1	0	1	−1	0	0	0	0
3	0	−1	0	1	−1	0	−1	0
4	0	0	−1	0	1	1	0	−1
5	0	0	0	0	0	−1	1	1

Figure 2.14 Node–arc incidence matrix of the network example.

The node–arc incidence matrix has a very special structure: Only $2m$ out of its nm entries are nonzero, all of its nonzero entries are $+1$ or -1, and each column has exactly one $+1$ and one -1. Furthermore, the number of $+1$'s in a row equals the outdegree of the corresponding node and the number of -1's in the row equals the indegree of the node.

Because the node–arc incidence matrix $\mathcal{N}$ contains so few nonzero coefficients, the incidence matrix representation of a network is not space efficient. More efficient schemes, such as those that we consider later in this section would merely keep track of the nonzero entries in the matrix. Because of its inefficiency in storing the underlying network topology, use of the node–arc incidence matrix rarely produces efficient algorithms. This representation is important, however, because it represents the constraint matrix of the minimum cost flow problem and because the node–arc incidence matrix possesses several interesting theoretical properties. We study some of these properties in Sections 11.11 and 11.12.

Node–Node Adjacency Matrix

The node–node adjacency matrix representation, or simply the adjacency matrix representation, stores the network as an $n \times n$ matrix $\mathcal{H} = \{h_{ij}\}$. The matrix has a row and a column corresponding to every node, and its ijth entry h_{ij} equals 1 if $(i, j) \in A$ and equals 0 otherwise. Figure 2.15 specifies this representation for the network shown in Figure 2.13. If we wish to store arc costs and capacities as well as the network topology, we can store this information in two additional $n \times n$ matrices, $\mathcal{C}$ and $\mathcal{U}$.

The adjacency matrix has n^2 elements, only m of which are nonzero. Consequently, this representation is space efficient only if the network is sufficiently dense; for sparse networks this representation wastes considerable space. Nevertheless, the simplicity of the adjacency representation permits us to use it to implement most network algorithms rather easily. We can determine the cost or capacity of any arc (i, j) simply by looking up the ijth element in the matrix $\mathcal{C}$ or $\mathcal{U}$. We can obtain the arcs emanating from node i by scanning row i: If the jth element in this row has a nonzero entry, (i, j) is an arc of the network. Similarly, we can obtain the arcs entering node j by scanning column j: If the ith element of this column has a nonzero entry, (i, j) is an arc of the network. These steps permit us to identify all the outgoing or incoming arcs of a node in time proportional to n. For dense networks we can usually afford to spend this time to identify the incoming or outgoing arcs, but for

	1	2	3	4	5
1	0	1	1	0	0
2	0	0	0	1	0
3	0	1	0	0	0
4	0	0	1	0	1
5	0	0	1	1	0

Figure 2.15 Node–node adjacency matrix of the network example.

sparse networks these steps might be the bottleneck operations for an algorithm. The two representations we discuss next permit us to identify the set of outgoing arcs $A(i)$ of any node in time proportional to $|A(i)|$.

Adjacency Lists

Earlier we defined the *arc adjacency list* $A(i)$ of a node i as the set of arcs emanating from that node, that is, the set of arcs $(i, j) \in A$ obtained as j ranges over the nodes of the network. Similarly, we defined the *node adjacency list* of a node i as the set of nodes j for which $(i, j) \in A$. The *adjacency list representation* stores the node adjacency list of each node as a singly linked list (we refer the reader to Appendix A for a description of singly linked lists). A linked list is a collection of cells each containing one or more fields. The node adjacency list for node i will be a linked list having $|A(i)|$ cells and each cell will correspond to an arc $(i, j) \in A$. The cell corresponding to the arc (i, j) will have as many fields as the amount of information we wish to store. One data field will store node j. We might use two other data fields to store the arc cost c_{ij} and the arc capacity u_{ij}. Each cell will contain one additional field, called the *link*, which stores a pointer to the next cell in the adjacency list. If a cell happens to be the last cell in the adjacency list, by convention we set its link to value zero.

Since we need to be able to store and access n linked lists, one for each node, we also need an array of pointers that point to the first cell in each linked list. We accomplish this objective by defining an n-dimensional array, *first*, whose element *first(i)* stores a pointer to the first cell in the adjacency list of node i. If the adjacency list of node i is empty, we set *first(i)* = 0. Figure 2.16 specifies the adjacency list representation of the network shown in Figure 2.13.

In this book we sometimes assume that whenever arc (i, j) belongs to a network, so does the reverse arc (j, i). In these situations, while updating some information about arc (i, j), we typically will also need to update information about arc (j, i). Since we will store arc (i, j) in the adjacency list of node i and arc (j, i) in the adjacency list of node j, we can carry out any operation on both arcs efficiently if we know where to find the reversal (j, i) of each arc (i, j). We can access both arcs

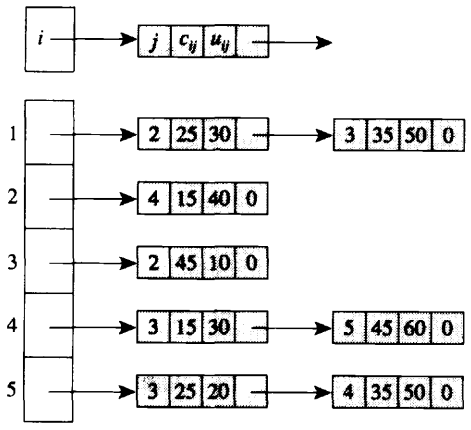


Figure 2.16 Adjacency list representation of the network example.

easily if we define an additional field, *mate*, that contains a pointer to the cell containing data for the reversal of each arc. The mate of arc (i, j) points to the cell of arc (j, i) and the mate of arc (j, i) points to the cell of arc (i, j).

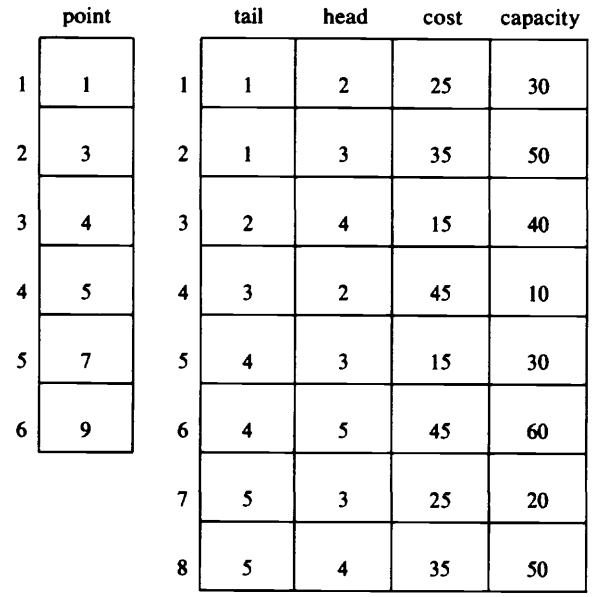
Forward and Reverse Star Representations

The *forward star representation* of a network is similar to the adjacency list representation in the sense that it also stores the node adjacency list of each node. But instead of maintaining these lists as linked lists, it stores them in a single array. To develop this representation, we first associate a unique sequence number with each arc, thus defining an ordering of the arc list. We number the arcs in a specific order: first those emanating from node 1, then those emanating from node 2, and so on. We number the arcs emanating from the same node in an arbitrary fashion. We then sequentially store information about each arc in the arc list. We store the tails, heads, costs, and capacities of the arcs in four arrays: *tail*, *head*, *cost*, and *capacity*. So if arc (i, j) is arc number 20, we store the tail, head, cost, and capacity data for this arc in the array positions tail(20), head(20), cost(20), and capacity(20). We also maintain a pointer with each node i, denoted by *point(i)*, that indicates the smallest-numbered arc in the arc list that emanates from node i. [If node i has no outgoing arcs, we set point(i) equal to point($i + 1$).] Therefore, the forward star representation will store the outgoing arcs of node i at positions point(i) to (point($i + 1$) $- 1$) in the arc list. If point(i) $>$ point($i + 1$) $- 1$, node i has no outgoing arc. For consistency, we set point(1) $= 1$ and point($n + 1$) $= m + 1$. Figure 2.17(a) specifies the forward star representation of the network given in Figure 2.13.

The forward star representation provides us with an efficient means for determining the set of outgoing arcs of any node. To determine, simultaneously, the set of incoming arcs of any node efficiently, we need an additional data structure known as the *reverse star representation*. Starting from a forward star representation, we can create a reverse star representation as follows. We examine the nodes $i = 1$ to n in order and sequentially store the heads, tails, costs, and capacities of the incoming arcs at node i. We also maintain a reverse pointer with each node i, denoted by *rpoint(i)*, which denotes the first position in these arrays that contains information about an incoming arc at node i. [If node i has no incoming arc, we set rpoint(i) equal to rpoint($i + 1$).] For sake of consistency, we set rpoint(1) $= 1$ and rpoint($n + 1$) $= m + 1$. As before, we store the incoming arcs at node i at positions rpoint(i) to (rpoint($i + 1$) $- 1$). This data structure gives us the representation shown in Figure 2.17(b).

Observe that by storing both the forward and reverse star representations, we will maintain a significant amount of duplicate information. We can avoid this duplication by storing arc numbers in the reverse star instead of the tails, heads, costs, and capacities of the arcs. As an illustration, for our example, arc (3, 2) has arc number 4 in the forward star representation and arc (1, 2) has an arc number 1. So instead of storing the tails, costs, and capacities of the arcs, we simply store arc numbers; and once we know the arc numbers, we can always retrieve the associated information from the forward star representation. We store arc numbers in an array *trace* of size m. Figure 2.18 gives the complete *trace* array of our example.

In our discussion of the adjacency list representation, we noted that sometimes



	point
1	1
2	3
3	4
4	5
5	7
6	9

	tail	head	cost	capacity
1	1	2	25	30
2	1	3	35	50
3	2	4	15	40
4	3	2	45	10
5	4	3	15	30
6	4	5	45	60
7	5	3	25	20
8	5	4	35	50

(a)

	cost	capacity	tail	head
1	45	10	3	2
2	25	30	1	2
3	35	50	1	3
4	15	30	4	3
5	25	20	5	3
6	35	50	5	4
7	15	40	2	4
8	45	60	4	5

	rpoint
1	1
2	1
3	3
4	6
5	8
6	9

(b)

Figure 2.17 (a) Forward star and (b) reverse star representations of the network example.

while updating data for an arc (i, j), we also need to update data for its reversal (j, i). Just as we did in the adjacency list representation, we can accomplish this task by defining an array *mate* of size m, which stores the arc number of the reversal of an arc. For example, the forward star representation shown in Figure 2.17(a) assigns the arc number 6 to arc (4, 5) and assigns the arc number 8 to arc (5, 4).

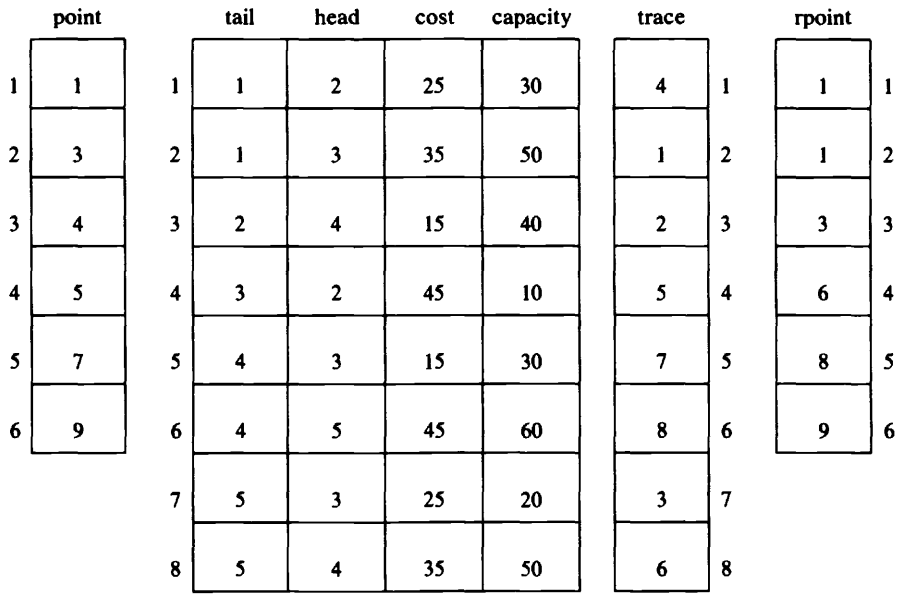


Figure 2.18 Compact forward and reverse star representation of the network example.

Therefore, if we were using the *mate* array, we would set *mate(6)* = 8 and *mate(8)* = 6.

Comparison of Forward Star and Adjacency List Representations

The major advantage of the forward star representation is its space efficiency. It requires less storage than does the adjacency list representation. In addition, it is much easier to implement in languages such as FORTRAN that have no natural provisions for using linked lists. The major advantage of adjacency list representation is its ease of implementation in languages such as Pascal or C that are able to manipulate linked lists efficiently. Further, using an adjacency list representation, we can add or delete arcs (as well as nodes) in constant time. On the other hand, in the forward star representation these steps require time proportional to m, which can be too time consuming.

Storing Parallel Arcs

In this book we assume that the network does not contain parallel arcs; that is, no two arcs have the same tail and head nodes. By allowing parallel arcs, we encounter some notational difficulties, since (i, j) will not specify the arc uniquely. For networks with parallel arcs, we need more complex notation to specify arcs, arc costs, and capacities. This difficulty is merely notational, however, and poses no problems computationally: both the adjacency list representation and the forward star representation data structures are capable of handling parallel arcs. If a node i has two

outgoing arcs with the same head node but (possibly) different costs and capacities, the linked list of node i will contain two cells corresponding to these two arcs. Similarly, the forward star representation allows several entries with the same tail and head nodes but different costs and capacities.

Representing Undirected Networks

We can represent undirected networks using the same representations we have just described for directed networks. However, we must remember one fact: Whenever arc (i, j) belongs to an undirected network, we need to include both of the pairs (i, j) and (j, i) in the representations we have discussed. Consequently, we will store each arc (i, j) of an undirected network twice in the adjacency lists, once in the list for node i and once in the list for node j. Some other obvious modifications are needed. For example, in the node–arc incidence matrix representation, the column corresponding to arc (i, j) will have $+1$ in both rows i and j. The node–node adjacency matrix will have $+1$ in position h_{ij} and h_{ji} for every arc $(i, j) \in A$. Since this matrix will be symmetric, we might as well store half of the matrix. In the adjacency list representation, the arc (i, j) will be present in the linked lists of both nodes i and j. Consequently, whenever we update information for one arc, we must update it for the other arc as well. We can accomplish this task by storing for each arc the address of its other occurrence in an additional mate array. The forward star representation requires this additional storage as well. Finally, observe that undirected networks do not require the reverse star representation.

2.4 NETWORK TRANSFORMATIONS

Frequently, we require network transformations to simplify a network, to show equivalences between different network problems, or to state a network problem in a standard form required by a computer code. In this section, we describe some of these important transformations. In describing these transformations, we assume that the network problem is a minimum cost flow problem as formulated in Section 1.2. Needless to say, these transformations also apply to special cases of the minimum cost flow problem, such as the shortest path, maximum flow, and assignment problems, wherever the transformations are appropriate. We first recall the formulation of the minimum cost flow problem for convenience in discussing the network transformations.

$$\text{Minimize} \sum_{(i,j)\in A} c_{ij}x_{ij} \tag{2.1a}$$

subject to

$$\sum_{\{j:(i,j)\in A\}} x_{ij} - \sum_{\{j:(j,i)\in A\}} x_{ji} = b(i) \qquad \text{for all } i \in N, \tag{2.1b}$$

$$l_{ij} \le x_{ij} \le u_{ij} \qquad \text{for all } (i, j) \in A. \tag{2.1c}$$

Undirected Arcs to Directed Arcs

Sometimes minimum cost flow problems contain undirected arcs. An undirected arc (i, j) with cost $c_{ij} \geq 0$ and capacity u_{ij} permits flow from node i to node j and also from node j to node i; a unit of flow in either direction costs c_{ij}, and the total flow (i.e., from node i to node j plus from node j to node i) has an upper bound u_{ij}. That is, the undirected model has the constraint $x_{ij} + x_{ji} \leq u_{ij}$ and the term $c_{ij}x_{ij} + c_{ij}x_{ji}$ in the objective function. Since the cost $c_{ij} \geq 0$, in some optimal solution one of x_{ij} and x_{ji} will be zero. We refer to any such solution as non-overlapping.

For notational convenience, in this discussion we refer to the undirected arc (i, j) as $\{i, j\}$. We assume (with some loss of generality) that the arc flow in either direction on arc $\{i, j\}$ has a lower bound of value 0; our transformation is not valid if the arc flow has a nonzero lower bound or the arc cost c_{ij} is negative (why?). To transform the undirected case to the directed case, we replace each undirected arc $\{i, j\}$ by two directed arcs, (i, j) and (j, i), both with cost c_{ij} and capacity u_{ij}. To establish the correctness of this transformation, we show that every non-overlapping flow in the original network has an associated flow in the transformed network with the same cost, and vice versa. If the undirected arc $\{i, j\}$ carries α units of flow from node i to node j, in the transformed network $x_{ij} = \alpha$ and $x_{ji} = 0$. If the undirected arc $\{i, j\}$ carries α units of flow from node j to node i, in the transformed network $x_{ij} = 0$ and $x_{ji} = \alpha$. Conversely, if x_{ij} and x_{ji} are the flows on arcs (i, j) and (j, i) in the directed network, $x_{ij} - x_{ji}$ or $x_{ji} - x_{ij}$ is the associated flow on arc $\{i, j\}$ in the undirected network, whichever is positive. If $x_{ij} - x_{ji}$ is positive, the flow from node i to node j on arc $\{i, j\}$ equals this amount. If $x_{ji} - x_{ij}$ is positive, the flow from node j to node i on arc $\{i, j\}$ equals $x_{ji} - x_{ij}$. In either case, the flow in the opposite direction is zero. If $x_{ji} - x_{ij}$ is zero, the flow on arc $\{i, j\}$ is 0.

Removing Nonzero Lower Bounds

If an arc (i, j) has a nonzero lower bound l_{ij} on the arc flow x_{ij}, we replace x_{ij} by $x'_{ij} + l_{ij}$ in the problem formulation. The flow bound constraint then becomes $l_{ij} \leq x'_{ij} + l_{ij} \leq u_{ij}$, or $0 \leq x'_{ij} \leq (u_{ij} - l_{ij})$. Making this substitution in the mass balance constraints decreases $b(i)$ by l_{ij} units and increases $b(j)$ by l_{ij} units [recall from Section 1.2 that the flow variable x_{ij} appears in the mass balance constraint (2.1b) of only nodes i and j]. This substitution changes the objective function value by a constant that we can record separately and then ignore when solving the problem. Figure 2.19 illustrates this transformation graphically. We can view this transformation as a two-step flow process: We begin by sending l_{ij} units of flow on arc (i, j), which decreases $b(i)$ by l_{ij} units and increases $b(j)$ by l_{ij} units, and then we measure (by the variable x'_{ij}) the incremental flow on the arc beyond the flow value l_{ij}.

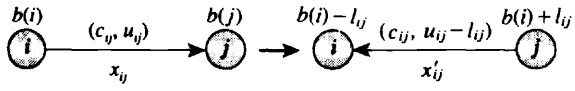


Figure 2.19 Removing nonzero lower bounds.

Arc Reversal

The arc reversal transformation is typically used to remove arcs with negative costs. Let u_{ij} denote the capacity of the arc (i, j) or an upper bound on the arc's flow if the arc is uncapacitated. In this transformation we replace the variable x_{ij} by $u_{ij} - x_{ji}$. Doing so replaces the arc (i, j), which has an associated cost c_{ij}, by the arc (j, i) with an associated cost $-c_{ij}$. As shown in Figure 2.20, the transformation has the following network interpretation. We first send u_{ij} units of flow on the arc (which decreases $b(i)$ by u_{ij} units and increases $b(j)$ by u_{ij} units) and then we replace arc (i, j) by arc (j, i) with cost $-c_{ij}$. The new flow x_{ji} measures the amount of flow we "remove" from the "full capacity" flow of u_{ij}.

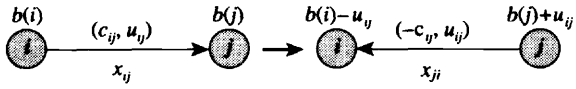


Figure 2.20 Arc reversal transformation.

Removing Arc Capacities

If an arc (i, j) has a positive capacity u_{ij}, we can remove the capacity, making the arc *uncapacitated*, by using the following idea: We introduce an additional node so that the capacity constraint on arc (i, j) becomes the mass balance constraint of the new node. Suppose that we introduce a slack variable $s_{ij} \geq 0$, and write the capacity constraint $x_{ij} \leq u_{ij}$ in an equality form as $x_{ij} + s_{ij} = u_{ij}$. Multiplying both sides of the equality by -1, we obtain

$$-x_{ij} - s_{ij} = -u_{ij} \tag{2.2}$$

We now treat constraint (2.2) as the mass balance constraint of an additional node k. Observe that the flow variable x_{ij} now appears in three mass balance constraints and s_{ij} in only one. By subtracting (2.2) from the mass balance constraint of node j (which contains the flow variable x_{ij} with a negative sign), we assure that each of x_{ij} and s_{ij} appears in exactly two constraints—in one with a positive sign and in the other with a negative sign. These algebraic manipulations correspond to the network transformation shown in Figure 2.21.

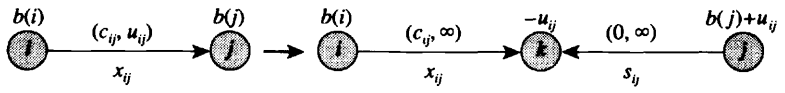


Figure 2.21 Transformation for removing an arc capacity.

To see the relationships between the flows in the original and transformed networks, we make the following observations. If x_{ij} is the flow on arc (i, j) in the original network, the corresponding flow in the transformed network is $x'_{ik} = x_{ij}$ and $x'_{jk} = u_{ij} - x_{ij}$. Notice that both the flows x and x' have the same cost. Similarly, a flow x'_{ik}, x'_{jk} in the transformed network yields a flow $x_{ij} = x'_{ik}$ of the same cost in the original network. Furthermore, since $x'_{ik} + x'_{jk} = u_{ij}$ and x'_{ik} and x'_{jk} are both nonnegative, $x_{ij} = x'_{ik} \leq u_{ij}$. Therefore, the flow x_{ij} satisfies the arc capacity, and the transformation does correctly model arc capacities.

Suppose that every arc in a given network $G = (N, A)$ is capacitated. If we apply the preceding transformation to every arc, we obtain a bipartite uncapacitated network G' (see Figure 2.22 for an illustration). In this network (1) each node i on the left corresponds to a node $i \in N$ of the original network and has a supply equal to $b(i) + \sum_{\{k:(k,i)\in A\}} u_{ki}$, and (2) each node i–j on the right corresponds to an arc $(i, j) \in A$ in the original network and has a demand equal to u_{ij}; this node has exactly two incoming arcs, originating at nodes i and j from the left. Consequently, the transformed network has $(n + m)$ nodes and $2m$ arcs.

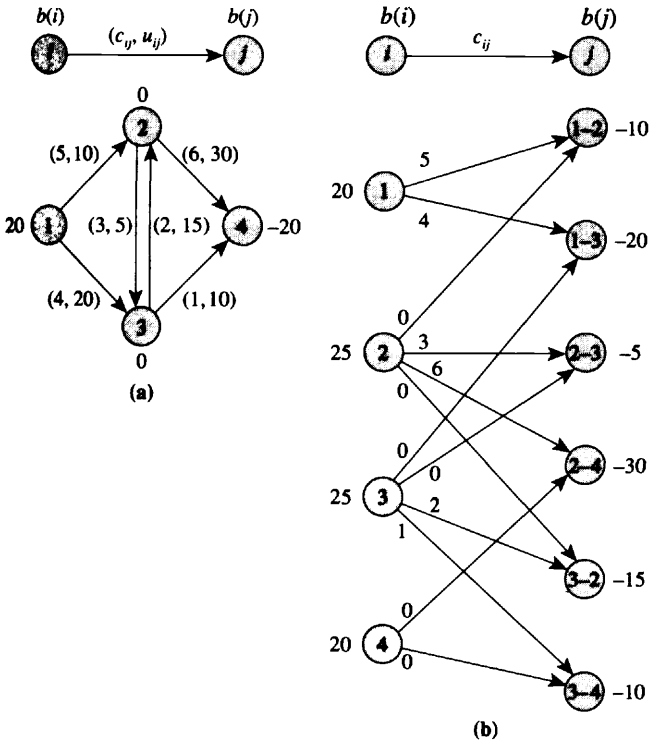


Figure 2.22 Transformation for removing arc capacities: (a) original network; (b) transformed network with uncapacitated arcs.

At first glance we might be tempted to believe that this technique for removing arc capacities would be unattractive computationally since the transformation substantially increases the number of nodes in the network. However, on most occasions the original and transformed networks have algorithms with the same complexity, because the transformed network possesses a special structure that permits us to design more efficient algorithms.

Node Splitting

The node splitting transformation splits each node i into two nodes i' and i'' corresponding to the node's *output* and *input* functions. This transformation replaces each original arc (i, j) by an arc (i', j'') of the same cost and capacity. It also adds an arc (i'', i') of zero cost and with infinite capacity for each i. The input side of

node i (i.e., node i'') receives all the node's inflow, the output side (i.e., node i') sends all the node's outflow, and the additional arc (i'', i') carries flow from the input side to the output side. Figure 2.23 illustrates the resulting network when we carry out the node splitting transformation for all the nodes of a network. We define the supplies/demands of nodes in the transformed network in accordance with the following three cases:

1. If $b(i) > 0$, then $b(i'') = b(i)$ and $b(i') = 0$.
2. If $b(i) < 0$, then $b(i'') = 0$ and $b(i') = b(i)$.
3. If $b(i) = 0$, then $b(i') = b(i'') = 0$.

It is easy to show a one-to-one correspondence between a flow in the original network and the corresponding flow in the transformed network; moreover, the flows in both networks have the same cost.

The node splitting transformation permits us to model numerous applications in a variety of practical problem domains, yet maintain the form of the network flow model that we introduced in Section 1.2. For example, we can use the transformation to handle situations in which nodes as well as arcs have associated capacities and costs. In these situatio , each flow unit passing through a node i incurs a cost c_i and the maximum flow that can pass through the node is u_i. We can reduce this problem to the standard "arc flow" form of the network flow problem by performing the node splitting transformation and letting c_i and u_i be the cost and capacity of arc

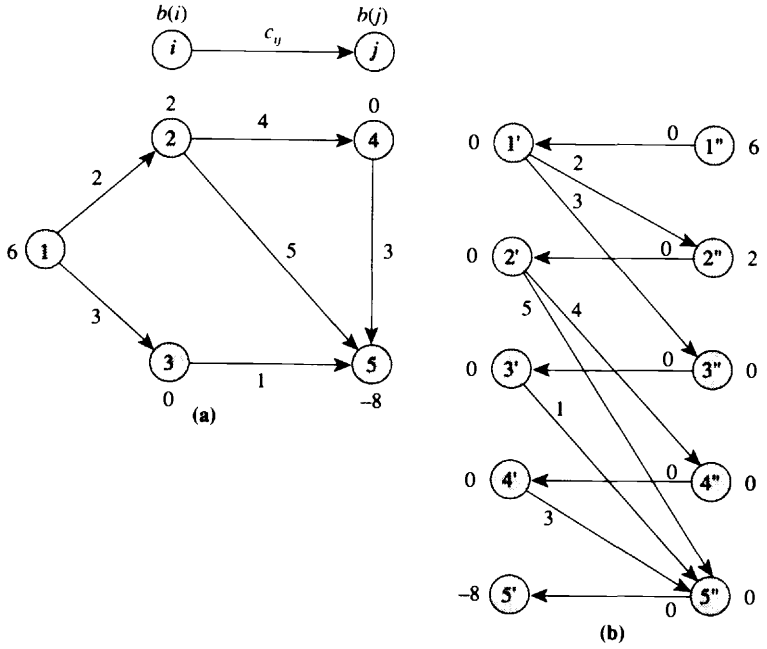


Figure 2.23 Node splitting transformation: (a) original network; (b) transformed network.

(i'', i'). We shall study more applications of the node splitting transformation in Sections 6.6 and 12.7 and in several exercises.

Working with Reduced Costs

In many of the network flow algorithms discussed in this book, we measure the cost of an arc relative to "imputed" costs associated with its incident nodes. These imputed costs typically are intermediate data that we compute within the context of an algorithm. Suppose that we associate with each node $i \in N$ a number $\pi(i)$, which we refer to as the *potential* of that node. With respect to the node potentials $\pi = (\pi(1), \pi(2), \ldots, \pi(n))$, we define the *reduced cost* c_{ij}^{π} of an arc (i, j) as

$$c_{ij}^{\pi} = c_{ij} - \pi(i) + \pi(j). \tag{2.3}$$

In many algorithms discussed later, we often work with reduced costs c_{ij}^{π} instead of the actual costs c_{ij}. Consequently, it is important to understand the relationship between the objective functions $z(\pi) = \sum_{(i,j)\in A} c_{ij}^{\pi} x_{ij}$ and $z(0) = \sum_{(i,j)\in A} c_{ij} x_{ij}$. Suppose, initially, that $\pi = 0$ and we then increase the node potential of node k to $\pi(k)$. The definition (2.3) of reduced costs implies that this change reduces the reduced cost of each unit of flow leaving node k by $\pi(k)$ and increases the reduced cost of each flow unit entering node k by $\pi(k)$. Thus the total decrease in the objective function equals $\pi(k)$ times the outflow of node k minus the inflow of node k. By definition (see Section 1.2), the outflow minus inflow equals the supply/demand of the node. Consequently, increasing the potential of node k by $\pi(k)$ decreases the objective function value by $\pi(k)b(k)$ units. Repeating this argument iteratively for each node establishes that

$$z(0) - z(\pi) = \sum_{i\in N} \pi(i)b(i) = \pi b.$$

For a given node potential π, πb is a constant. Therefore, a flow that minimizes $z(\pi)$ also minimizes $z(0)$. We formalize this result for easy future reference.

Property 2.4. *Minimum cost flow problems with arc costs c_{ij} or c_{ij}^{π} have the same optimal solutions. Moreover, $z(\pi) = z(0) - \pi b$.*

We next study the effect of working with reduced costs on the cost of cycles and paths. Let W be a directed cycle in G. Then

$$\begin{aligned}
\sum_{(i,j)\in W} c_{ij}^{\pi} &= \sum_{(i,j)\in W} (c_{ij} - \pi(i) + \pi(j)), \\
&= \sum_{(i,j)\in W} c_{ij} + \sum_{(i,j)\in W} (\pi(j) - \pi(i)), \\
&= \sum_{(i,j)\in W} c_{ij}.
\end{aligned}$$

The last equality follows from the fact that for any directed cycle W, the expression $\sum_{(i,j)\in W} (\pi(j) - \pi(i))$ sums to zero because for each node i in the cycle W, $\pi(i)$ occurs once with a positive sign and once with a negative sign. Similarly, if P

is a directed path from node k to node l, then

$$\begin{aligned}\sum_{(i,j)\in P} c_{ij}^{\pi} &= \sum_{(i,j)\in P} (c_{ij} - \pi(i) + \pi(j)),\\ &= \sum_{(i,j)\in P} c_{ij} - \sum_{(i,j)\in P} (\pi(i) - \pi(j)),\\ &= \sum_{(i,j)\in P} c_{ij} - \pi(k) + \pi(l),\end{aligned}$$

because all $\pi(\cdot)$ corresponding to the nodes in the path, other than the terminal nodes k and l, cancel each other in the expression $\sum_{(i,j)\in P} (\pi(i) - \pi(j))$. We record these results for future reference.

Property 2.5

(a) *For any directed cycle W and for any node potentials π, $\sum_{(i,j)\in W} c_{ij}^{\pi} = \sum_{(i,j)\in W} c_{ij}$.*

(b) *For any directed path P from node k to node l and for any node potentials π, $\sum_{(i,j)\in P} c_{ij}^{\pi} = \sum_{(i,j)\in P} c_{ij} - \pi(k) + \pi(l)$.*

Working with Residual Networks

In designing, developing, and implementing network flow algorithms, it is often convenient to measure flow not in absolute terms, but rather in terms of incremental flow about some given feasible solution—typically, the solution at some intermediate point in an algorithm. Doing so leads us to define a new, ancillary network, known as the *residual network*, that functions as a "remaining flow network" for carrying the incremental flow. We show that formulations of the problem in the original network and in the residual network are equivalent in the sense that they give a one-to-one correspondence between feasible solutions to the two problems that preserves the value of the cost of solutions.

The concept of residual network is based on the following intuitive idea. Suppose that arc (i, j) carries x_{ij}° units of flow. Then we can send an additional $u_{ij} - x_{ij}^{\circ}$ units of flow from node i to node j along arc (i, j). Also notice that we can send up to x_{ij}° units of flow from node j to node i over the arc (i, j), which amounts to canceling the existing flow on the arc. Whereas sending a unit flow from node i to node j on arc (i, j) increases the flow cost by c_{ij} units, sending flow from node j to node i on the same arc decreases the flow cost by c_{ij} units (since we are saving the cost that we used to incur in sending the flow from node i to node j).

Using these ideas, we define the residual network with respect to a given flow x° as follows. We replace each arç (i, j) in the original network by two arcs, (i, j) and (j, i): the arc (i, j) has cost c_{ij} and *residual capacity* $r_{ij} = u_{ij} - x_{ij}^{\circ}$, and the arc (j, i) has cost $-c_{ij}$ and *residual capacity* $r_{ji} = x_{ij}^{\circ}$ (see Figure 2.24). The residual network consists of only the arcs with a positive residual capacity. We use the notation $G(x^{\circ})$ to represent the residual network corresponding to the flow x°.

In general, the concept of residual network poses some notational difficulties. If for some pair i and j of nodes, the network G contains both the arcs (i, j) and

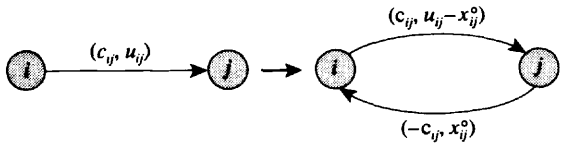


Figure 2.24 Constructing the residual network $G(x^0)$.

(j, i), the residual network may contain two (parallel) arcs from node i to node j with different costs and residual capacities, and/or two (parallel) arcs from node j to node i with different costs and residual capacities. In these instances, any reference to arc (i, j) will be ambiguous and will not define a unique arc cost and residual capacity. We can overcome this difficulty by assuming that for any pair of nodes i and j, the graph G does not contain both arc (i, j) and arc (j, i); then the residual network will contain no parallel arcs. We might note that this assumption is merely a notational convenience; it does not impose any loss of generality, because by suitable transformations we can always define a network that is equivalent to any given network and that will satisfy this assumption (see Exercise 2.47). However, we need not actually make this transformation in practice, since the network representations described in Section 2.3 are capable of handling parallel arcs.

We note further that although the construction and use of the residual network poses some notational difficulties for the general minimum cost flow problem, the difficulties might not arise for some special cases. In particular, for the maximum flow problem, the parallel arcs have the same cost (of zero), so we can merge both of the parallel arcs into a single arc and set its residual capacity equal to the sum of the residual capacities of the two arcs. For this reason, in our discussion of the maximum flow problem, we will permit the underlying network to contain arcs joining any two nodes in both directions.

We now show that every flow x in the network G corresponds to a flow x' in the residual network $G(x^\circ)$. We define the flow $x' \geq 0$ as follows:

$$x'_{ij} - x'_{ji} = x_{ij} - x^\circ_{ij}, \tag{2.4}$$

and

$$x'_{ij}x'_{ji} = 0. \tag{2.5}$$

The condition (2.5) implies that x'_{ij} and x'_{ji} cannot both be positive at the same time. If $x_{ij} \geq x^\circ_{ij}$, we set $x'_{ij} = (x_{ij} - x^\circ_{ij})$ and $x'_{ji} = 0$. Notice that if $x_{ij} \leq u_{ij}$, then $x'_{ij} \leq u_{ij} - x^\circ_{ij} = r_{ij}$. Therefore, the flow x'_{ij} satisfies the flow bound constraints. Similarly, if $x_{ij} < x^\circ_{ij}$, we set $x'_{ij} = 0$ and $x'_{ji} = x^\circ_{ij} - x_{ij}$. Observe that $0 \leq x'_{ji} \leq x^\circ_{ij} = r_{ji}$, so the flow x'_{ji} also satisfies the flow bound constraints. These observations show that if x is a feasible flow in G, its corresponding flow x' is a feasible flow in $G(x^\circ)$.

We next establish a relationship between the cost of a flow x in G and the cost of the corresponding flow x' in $G(x^\circ)$. Let c' denote the arc costs in the residual network. Then for every arc $(i, j) \in A$, $c'_{ij} = c_{ij}$ and $c'_{ji} = -c_{ij}$. For a flow x_{ij} on arc (i, j) in the original network G, the cost of flow on the pair of arcs (i, j) and (j, i) in the residual network $G(x^\circ)$ is $c'_{ij}x'_{ij} + c'_{ji}x'_{ji} = c'_{ij}(x'_{ij} - x'_{ji}) = c_{ij}x_{ij} - c_{ij}x^\circ_{ij}$; the last equality follows from (2.4). We have thus shown that

$$c'x' = cx - cx^\circ.$$

Similarly, we can show the converse result that if x' is a feasible flow in the residual network $G(x^\circ)$, the solution given by $x_{ij} = (x'_{ij} - x'_{ji}) + x^\circ_{ij}$ is a feasible flow in G. Moreover, the costs of these two flows is related by the equality $cx = c'x' + cx^\circ$. We ask the reader to prove these results in Exercise 2.48. We summarize the preceding discussion as the following property.

Property 2.6. *A flow x is a feasible flow in the network G if and only if its corresponding flow x', defined by $x'_{ij} - x'_{ji} = x_{ij} - x^\circ_{ij}$ and $x'_{ij}x'_{ji} = 0$, is feasible in the residual network $G(x^\circ)$. Furthermore, $cx = c'x' + cx^\circ$.*

One important consequence of Property 2.6 is the flexibility it provides us. Instead of working with the original network G, we can work with the residual network $G(x^\circ)$ for some x°: Once we have determined an optimal solution in the residual network, we can immediately convert it into an optimal solution in the original network. Many of the maximum flow and minimum cost flow algorithms discussed in the subsequent chapters use this result.

2.5 SUMMARY

In this chapter we brought together many basic definitions of network flows and graph theory and presented basic notation that we will use throughout this book. We defined several common graph theoretic terms, including adjacency lists, walks, paths, cycles, cuts, and trees. We also defined acyclic and bipartite networks.

Although networks are often geometric entities, optimization algorithms require computer representations of them. The following four representations are the most common: (1) the node–arc incidence matrix, (2) the node–node adjacency matrix, (3) adjacency lists, and (4) forward and reverse star representations. Figure 2.25 summarizes the basic features of these representations.

Network representations	Storage space	Features
Node–arc incidence matrix	nm	1. Space inefficient 2. Too expensive to manipulate 3. Important because it represents the constraint matrix of the minimum cost flow problem
Node–node adjacency matrix	kn^2 for some constant k	1. Suited for dense networks 2. Easy to implement
Adjacency list	$k_1n + k_2m$ for some constants k_1 and k_2	1. Space efficient 2. Efficient to manipulate 3. Suited for dense as well as sparse networks
Forward and reverse star	$k_3n + k_4m$ for some constants k_3 and k_4	1. Space efficient 2. Efficient to manipulate 3. Suited for dense as well as sparse networks

Figure 2.25 Comparison of various network representations.

The field of network flows is replete with transformations that allow us to transform one problem to another, often transforming a problem that appears to include new complexities into a simplified "standard" format. In this chapter we described some of the most common transformations: (1) transforming undirected networks to directed networks, (2) removing nonzero lower flow bounds (which permits us to assume, without any loss of generality, that flow problems have zero lower bounds on arc flows), (3) performing arc reversals (which often permits us to assume, without any loss of generality, that arcs have nonnegative arc costs), (4) removing arc capacities (which allows us to transform capacitated networks to uncapacitated networks), (5) splitting nodes (which permits us to transform networks with constraints and/or cost associated with "node flows" into our formulation with all data and constraints imposed upon arc flows), and (6) replacing costs with reduced costs (which permits us to alter the cost coefficients, yet retain the same optimal solutions).

The last transformation we studied in this chapter permits us to work with residual networks, which is a concept of critical importance in the development of maximum flow and minimum cost flow algorithms. With respect to an existing flow x, the residual network $G(x)$ represents the capacity and cost information in the network for carrying incremental flows on the arcs. As our discussion has shown, working with residual networks is equivalent to working with the original network.

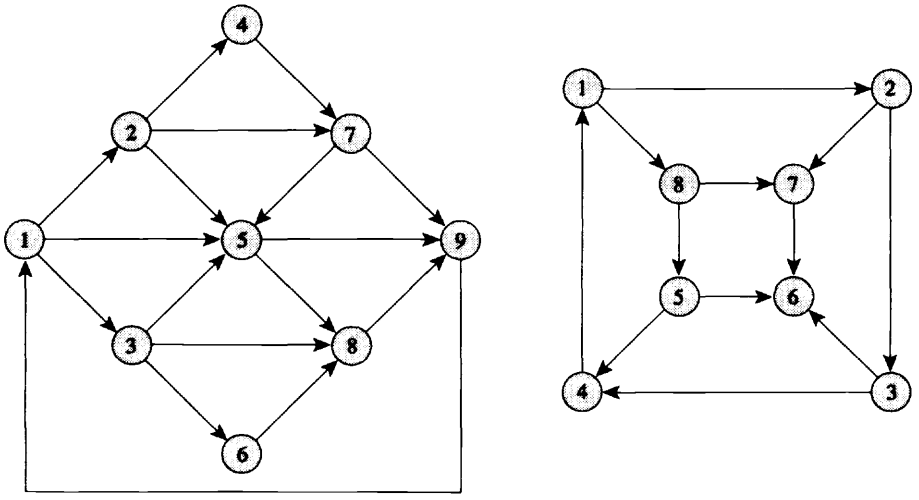
REFERENCE NOTES

The applied mathematics, computer science, engineering, and operations research communities have developed no standard notation of graph concepts; different researchers and authors use different names to denote the same object (e.g., some authors refer to nodes as vertices or points). The notation and definitions we have discussed in Section 2.2 and adopted throughout this book are among the most popular in the literature. The network representations and transformation that we described in Sections 2.3 and 2.4 are part of the folklore; it is difficult to pinpoint their origins. The books by Aho, Hopcroft, and Ullman [1974], Gondran and Minoux [1984], and Cormen, Leiserson, and Rivest [1990] contain additional information on network representations. The classic book by Ford and Fulkerson [1962] discusses many transformations of network flow problems.

EXERCISES

Note: If any of the following exercises does not state whether a graph is undirected or directed, assume either option, whichever is more convenient.

2.1 Consider the two graphs shown in Figure 2.26.
- **(a)** List the indegree and outdegree of every node.
- **(b)** Give the node adjacency list of each node. (Arrange each list in the increasing order of node numbers.)
- **(c)** Specify a directed walk containing six arcs. Also, specify a walk containing eight arcs.
- **(d)** Specify a cycle containing nine arcs and a directed cycle containing seven arcs.



(a) (b)

Figure 2.26 Example networks for Exercises 2.1 to 2.4.

2.2. Specify a spanning tree of the graph in Figure 2.26(a) with six leaves. Specify a cut of the graph in Figure 2.26(a) containing six arcs.

2.3. For the graphs shown in Figure 2.26, answer the following questions.
- **(a)** Are the graphs acyclic?
- **(b)** Are the graphs bipartite?
- **(c)** Are the graphs strongly connected?

2.4. Consider the graphs shown in Figure 2.26.
- **(a)** Do the graphs contain a directed in-tree for some root node?
- **(b)** Do the graphs contain a directed out-tree for some root node?
- **(c)** In Figure 2.26(a), list all fundamental cycles with respect to the following spanning tree $T = \{(1, 5), (1, 3), (2, 5), (4, 7), (7, 5), (7, 9), (5, 8), (6, 8)\}$.
- **(d)** For the spanning tree given in part (c), list all fundamental cuts. Which of these are the s–t cuts when $s = 1$ and $t = 9$?

2.5.
- **(a)** Construct a directed strongly connected graph with five nodes and five arcs.
- **(b)** Construct a directed bipartite graph with six nodes and nine arcs.
- **(c)** Construct an acyclic directed graph with five nodes and ten arcs.

2.6. **Bridges of Königsberg.** The first paper on graph theory was written by Leonhard Euler in 1736. In this paper, he started with the following mathematical puzzle: The city of Königsburg has seven bridges, arranged as shown in Figure 2.27. Is it possible to start at some place in the city, cross every bridge exactly once, and return to the starting place? Either specify such a tour or prove that it is impossible to do so.

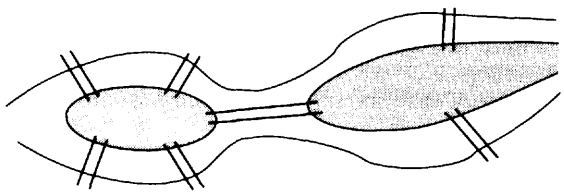


Figure 2.27 Bridges of Königsberg.

2.7. At the beginning of a dinner party, several participants shake hands with each other. Show that the participants that shook hands an odd number of times must be even in number.

2.8. Show that in a directed strongly connected graph containing more than one node, no node can have a zero indegree or a zero outdegree.

2.9. Suppose that every node in a directed graph has a positive indegree. Show that the graph must contain a directed cycle.

2.10. Show that a graph G remains connected even after deleting an arc (i, j) if and only if arc (i, j) belongs to some cycle in G.

2.11. Show that an undirected graph $G = (N, A)$ is connected if and only if for every partition of N into subsets N_1 and N_2, some arc has one endpoint in N_1 and the other endpoint in N_2.

2.12. Let $d_{\min}$ denote the minimum degree of a node in an undirected graph. Show that the graph contains a path containing at least $d_{\min}$ arcs.

2.13. Prove the following properties of trees.
(a) A tree on n nodes contains exactly $(n - 1)$ arcs.
(b) A tree has at least two leaf nodes (i.e., nodes with degree 1).
(c) Every two nodes of a tree are connected by a unique path.

2.14. Show that every tree is a bipartite graph.

2.15. Show that a forest consisting of k components has $m = n - k$ arcs.

2.16. Let $d_{\max}$ denote the maximum degree of a node in a tree. Show that the tree contains at least $d_{\max}$ nodes of degree 1. (*Hint*: Use the fact that the sum of the degrees of all nodes in a tree is $2m = 2n - 2$.)

2.17. Let Q be any cut of a connected graph and T be any spanning tree. Show that $Q \cap T$ is nonempty.

2.18. Show that a closed directed walk containing an odd number of arcs contains a directed cycle having an odd number of arcs. Is it true that a closed directed walk containing an even number of arcs also contains a directed cycle having an even number of arcs?

2.19. Show that any cycle of a graph G contains an even number of arcs (possibly zero) in common with any cut of G.

2.20. Let $d_{\min}$ denote the minimum degree of a node in an undirected graph G. Show that if $d_{\min} \geq 2$, then G must contain a cycle.

2.21. **(a)** Show that in a bipartite graph every cycle contains an even number of arcs.
(b) Show that a (connected) graph, in which every cycle contains an even number of arcs, must be bipartite. Conclude that a graph is bipartite if and only if every cycle has an even number of arcs.

2.22. The *k-color problem* on an undirected graph $G = (N, A)$ is defined as follows: Color all the nodes in N using at most k colors so that for every arc $(i, j) \in A$, nodes i and j have a different color.
(a) Given a world map, we want to color countries using at most k colors so that the countries having common boundaries have a different color. Show how to formulate this problem as a k-color problem.
(b) Show that a graph is bipartite if and only if it is *2-colorable* (i.e., can be colored using at most two colors).

2.23. Two undirected graphs $G = (N, A)$ and $G' = (N', A')$ are said to be *isomorphic* if we can number the nodes of the graph G so that G becomes identical to G'. Equivalently, G is isomorphic to G' if some one-to-one function f maps N onto N' so that (i, j) is an arc in A if and only if $(f(i), f(j))$ is an arc in A'. Give several necessary conditions for two undirected graphs to be isomorphic. (*Hint*: For example, they must have the same number of nodes and arcs.)

2.24. **(a)** List all nonisomorphic trees having four nodes.
(b) List all nonisomorphic trees having five nodes. (*Hint:* There are three such trees.)

2.25. For any undirected graph $G = (N, A)$, we define its *complement* $G^c = (N, A^c)$ as follows: If $(i, j) \in A$, then $(i, j) \notin A^c$, and if $(i, j) \notin A$, then $(i, j) \in A^c$. Show that if the graph G is disconnected, its complement G^c is connected.

2.26. Let $G = (N, A)$ be an undirected graph. We refer to a subset $N_1 \subseteq N$ as *independent* if no two nodes in N_1 are adjacent. Let $\beta(G)$ denote the maximum cardinality of any independent set of G. We refer to a subset $N_2 \subseteq N$ as a *node cover* if each arc in A has at least one of its endpoints in N_2. Let $\eta(G)$ denote the minimum cardinality of any node cover G. Show that $\beta(G) + \eta(G) = n$. (*Hint:* Show that the complement of an independent set is a node cover.)

2.27. **Problem of queens.** Consider the problem of determining the maximum number of queens that can be placed on a chessboard so that none of the queens can be taken by another. Show how to transform this problem into an independent set problem defined in Exercise 2.26.

2.28. Consider a directed graph $G = (N, A)$. For any subset $S \subseteq N$, let *neighbor(S)* denote the set of neighbors of S [i.e., neighbor$(S) = \{j \in N$: for some $i \in S$, $(i, j) \in A$ and $j \notin S\}$]. Show that G is strongly connected if and only if for every proper nonempty subset $S \subset N$, neighbor$(S) \neq \emptyset$.

2.29. A subset $N_1 \subseteq N$ of nodes in an undirected graph $G = (N, A)$ is said to be a *clique* if every pair of nodes in N_1 is connected by an arc. Show that the set N_1 is a clique in G if and only if N_1 is independent in its complement G^c.

2.30. Specify the node–arc incidence matrix and the node–node adjacency matrix for the graph shown in Figure 2.28.

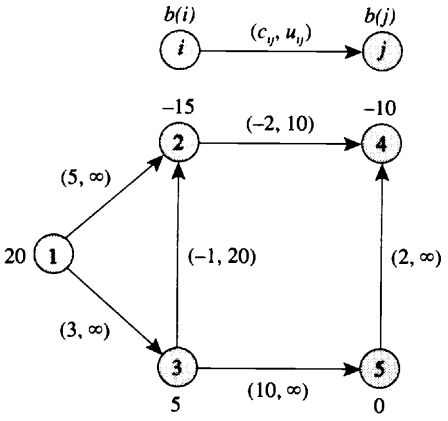


Figure 2.28 Network example.

2.31. (a) Specify the forward star representation of the graph shown in Figure 2.28.
(b) Specify the forward and reverse star representations of the graph shown in Figure 2.28.

2.32. Let $\mathcal{N}$ denote the node–arc incidence matrix of an undirected graph and let $\mathcal{N}^T$ denote its transpose. Let "$\cdot$" denote the operation of taking a product of two matrices. Show how to interpret the diagonal elements of $\mathcal{N} \cdot \mathcal{N}^T$?

2.33. Let $\mathcal{H}$ denote the node–node adjacency matrix of a directed network, and let $\mathcal{N}$ denote the node–arc incidence matrix of this network. Can $\mathcal{H} = \mathcal{N} \cdot \mathcal{N}^T$?

2.34. Let $\mathcal{H}$ be the node–node adjacency matrix of a directed graph $G = (N, A)$. Let $\mathcal{H}^T$ be the transpose of $\mathcal{H}$, and let G^T be the graph corresponding to $\mathcal{H}^T$. How is the graph G^T related to G?

2.35. Let G be a bipartite graph. Show that we can always renumber the nodes of G so that the node–node adjacency matrix $\mathcal{H}$ of G has the following form:

0	F
E	0

2.36. Show that a directed graph G is acyclic if and only if we can renumber its nodes so that its node–node adjacency matrix is a lower triangular matrix.

2.37. Let $\mathcal{H}$ denote the node–node adjacency matrix of a network G. Define $\mathcal{H}^k = \mathcal{H} \cdot \mathcal{H}^{k-1}$ for each $k = 2, 3, \ldots, n$. Show that the ijth entry of the matrix $\mathcal{H}^2$ is the number of directed paths consisting of two arcs from node i to node j. Then using induction, show that the ijth entry of matrix $\mathcal{H}^k$ is the number of distinct walks from node i to node j containing exactly k arcs. In making this assessment, assume that two walks are *distinct* if their sequences of arcs are different (even if the unordered set of arcs are the same).

2.38. Let $\mathcal{H}$ denote the node–node adjacency matrix of a network G. Show that G is strongly connected if and only if the matrix $\mathcal{R}$ defined by $\mathcal{R} = \mathcal{H} + \mathcal{H}^2 + \mathcal{H}^3 + \cdots + \mathcal{H}^n$ has no zero entry.

2.39. Write a pseudocode that takes as an input the node–node adjacency matrix representation of a network and produces as an output the forward and reverse star representations of the network. Your pseudocode should run in $O(n^2)$ time.

2.40. Write a pseudocode that accepts as an input the forward star representation of a network and produces as an output the network's node–node adjacency matrix representation.

2.41. Write a pseudocode that takes as an input the forward star representation of a network and produces the reverse star representation. Your pseudocode should run in $O(m)$ time.

2.42. Consider the minimum cost flow problem shown in Figure 2.28. Suppose that arcs (1, 2) and (3, 5) have lower bounds equal to $l_{12} = l_{35} = 5$. Transform this problem to one where all arcs have zero lower bounds.

2.43. In the network shown in Figure 2.28, some arcs have finite capacities. Transform this problem to one where all arcs are uncapacitated.

2.44. Consider the minimum cost flow problem shown in Figure 2.28 (note that some arcs have negative arc costs). Modify the problem so that all arcs have nonnegative arc costs.

2.45. Construct the residual network for the minimum cost flow problem shown in Figure 2.28 with respect to the following flow: $x_{12} = x_{13} = x_{32} = 10$ and $x_{24} = x_{35} = x_{54} = 5$.

2.46. For the minimum cost flow problem shown in Figure 2.28, specify a vector π of node potentials so that $c_{ij}^{\pi} \geq 0$ for every arc $(i, j) \in A$. Compute cx, $c^{\pi}x$, and πb for the flow given in Exercise 2.45 and verify that $cx = c^{\pi}x + \pi b$.

2.47. Suppose that a minimum cost flow problem contains both arcs (i, j) and (j, i) for some pair of nodes. Transform this problem to one in which the network contains either arc (i, j) or arc (j, i), but not both.

2.48. Show that if x' is a feasible flow in the residual network $G(x^\circ)$, the solution given by $x_{ij} = (x'_{ij} - x'_{ji}) + x^\circ_{ij}$ is a feasible flow in G and satisfies $cx = c'x' + cx^\circ$.

2.49. Suppose that you are given a minimum cost flow code that requires that its input data be specified so that $l_{ij} = u_{ij}$ for no arc (i, j). How would you eliminate such arcs?

2.50. Show how to transform a minimum cost flow problem stated in (2.1) into a circulation problem. Establish a one-to-one correspondence between the feasible solutions of these two problems. (*Hint:* Introduce two new nodes and some arcs.)

2.51. Show that by adding an extra node and appropriate arcs, we can formulate any minimum cost flow problem with one or more inequalities for supplies and demands (i.e., the mass balance constraints are stated as "$\leq b(i)$" for a supply node i, and/or "$\geq b(j)$" for a demand node j) into an equivalent problem with all equality constriants (i.e., "$= b(k)$" for all nodes k).