CHAPTER 15

Applications

In this chapter, we discuss briefly the most important applications of network flow problems.

1. The Transportation Problem

The network flow problem, when thought of as representing the shipment of goods along a transportation network, is called the *transshipment problem*. An important special case is when the set of nodes $\mathcal{N}$ can be partitioned into two sets $\mathcal{S}$ and $\mathcal{D}$,

$$\mathcal{N} = \mathcal{S} \cup \mathcal{D}, \qquad \mathcal{S} \cap \mathcal{D} = \emptyset,$$

such that every arc in $\mathcal{A}$ has its tail in $\mathcal{S}$ and its head in $\mathcal{D}$. The nodes in $\mathcal{S}$ are called *source (or supply) nodes*, while those in $\mathcal{D}$ are called *destination (or demand) nodes*. Such graphs are called *bipartite graphs* (see Figure 15.1). A network flow problem on such a bipartite graph is called a *transportation problem*.

In order for a transportation problem to be feasible, the supply must be nonnegative at every supply node, and the demand must be nonnegative at every demand node. That is,

$$b_i \geq 0 \quad \text{for } i \in \mathcal{S},$$
$$b_i \leq 0 \quad \text{for } i \in \mathcal{D}.$$

When put on paper, a bipartite graph has the annoying property that the arcs tend to cross each other many times. This makes such a representation inconvenient for carrying out the steps of the network simplex method. But there is a nice, uncluttered, tabular representation of a bipartite graph that one can use when applying the simplex method. To discover this tabular representation, first suppose that the graph is laid out as shown in Figure 15.2. Now if we place the supplies and demands on the nodes and the costs at the kinks in the arcs, then we get, for example, the following simple tabular representation of a transportation problem:

(15.1)

	-10	-23	-15
7	5	6	$*$
11	8	4	3
18	$*$	9	$*$
12	$*$	3	6

(the asterisks represent nonexistent arcs). The iterations of the simplex method can be written in this tabular format by simply placing the dual variables where the

R.J. Vanderbei, *Linear Programming*, International Series in Operations Research & Management Science 196, DOI 10.1007/978-1-4614-7630-6_15,

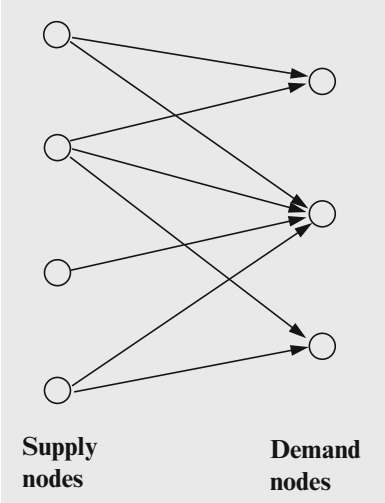

FIGURE 15.1. A bipartite graph—the network for a transportation problem.

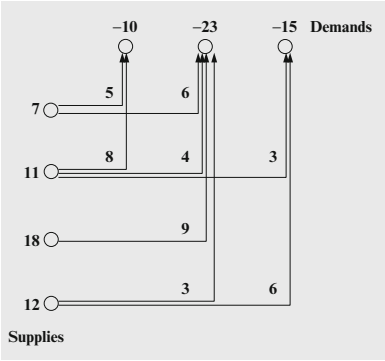

FIGURE 15.2. The bipartite graph from Figure 15.1 laid out in a rectangular fashion, with supplies and demands given at the nodes, and with costs given on the arcs.

supplies and demands are and by placing the primal flows and dual slacks where the arc costs are. Of course, some notation needs to be introduced to indicate which cells are part of the current spanning tree. For example, the tree could be indicated by putting a box around the primal flow values. Here is a (nonoptimal) tree solution for the data given above:

(15.2)

	5	1	4
0	[7]	5	*
−3	[3]	[8]	−4
−8	*	[18]	*
−2	*	[−3]	[15]

(the solution to this problem is left as an exercise).

In the case where every supply node is connected to every demand node, the problem is called the *Hitchcock Transportation Problem*. In this case, the equations defining the problem are especially simple. Indeed, if we denote the supplies at the supply nodes by r_i, $i \in \mathcal{S}$, and if we denote the demands at the demand nodes by s_j, $j \in \mathcal{D}$, then we can write the problem as

$$\begin{aligned} \text{minimize} \quad & \sum_{i\in\mathcal{S}}\sum_{j\in\mathcal{D}} c_{ij}x_{ij} \\ \text{subject to} \quad & \sum_{j\in\mathcal{D}} x_{ij} = r_i \qquad i \in \mathcal{S} \\ & \sum_{i\in\mathcal{S}} x_{ij} = s_j \qquad j \in \mathcal{D} \\ & x_{ij} \geq 0 \qquad i \in \mathcal{S}, j \in \mathcal{D}. \end{aligned}$$

2. The Assignment Problem

Given a set $\mathcal{S}$ of m people, a set $\mathcal{D}$ of m tasks, and for each $i \in \mathcal{S}$, $j \in \mathcal{D}$ a cost c_{ij} associated with assigning person i to task j, the *assignment problem* is to assign each person to one and only one task in such a manner that each task gets covered by someone and the total cost of the assignments is minimized. If we let

$$x_{ij} = \begin{cases} 1 & \text{if person } i \text{ is assigned task } j, \\ 0 & \text{otherwise,} \end{cases}$$

then the objective function can be written as

$$\text{minimize} \sum_{i\in\mathcal{S}}\sum_{j\in\mathcal{D}} c_{ij}x_{ij}.$$

The constraint that each person is assigned exactly one task can be expressed simply as

$$\sum_{j\in\mathcal{D}} x_{ij} = 1, \qquad \text{for all } i \in \mathcal{S}.$$

Also, the constraint that every task gets covered by someone is just

$$\sum_{i\in\mathcal{S}} x_{ij} = 1, \qquad \text{for all } j \in \mathcal{D}.$$

Except for the assumed integrality of the decision variables, x_{ij}, the assignment problem is just a Hitchcock transportation problem in which the supply at every supply node (person) is one and the demand at every demand node (task) is also one. This Hitchcock transportation problem therefore is called the *LP-relaxation* of the assignment problem. It is easy to see that every feasible solution to the assignment problem is a feasible solution for its LP-relaxation. Furthermore, every integral feasible solution to the LP-relaxation is a feasible solution to the assignment problem. Since the network simplex method applied to the LP-relaxation produces an integral solution, it therefore follows that the method solves not only the LP-relaxation but also the assignment problem itself. We should note that this is a very special and important feature of the network simplex method. For example, had we used

the primal–dual interior-point method to solve the LP-relaxation, there would be no guarantee that the solution obtained would be integral (unless the problem has a unique optimal solution, in which case any LP solver would find the same, integral answer—but typical assignment problems have alternate optimal solutions, and an interior-point method will report a convex combination of all of them).

3. The Shortest-Path Problem

Roughly speaking, the shortest-path problem is to find, well, the shortest path from one specific node to another in a network $(\mathcal{N}, \mathcal{A})$. In contrast to earlier usage, the arcs connecting successive nodes on a path must point in the direction of travel. Such paths are sometimes referred to as *directed paths*. To determine a shortest path, we assume that we are given the length of each arc. To be consistent with earlier notations, let us assume that the length of arc (i, j) is denoted by c_{ij}. Naturally, we assume that these lengths are nonnegative.

To find the shortest path from one node (say, s) to another (say, r), we will see that it is necessary to compute the shortest path from many, perhaps all, other nodes to r. Hence, we define the *shortest-path problem* as the problem of finding the shortest path from every node in $\mathcal{N}$ to a specific node $r \in \mathcal{N}$. The destination node r is called the *root node*.

3.1. Network Flow Formulation. The shortest-path problem can be formulated as a network flow problem. Indeed, put a supply of one unit at each nonroot node, and put the appropriate amount of demand at the root (to meet the total supply). The cost on each arc is just the length of the arc. Suppose that we've solved this network flow problem. Then the shortest path from a node i to r can be found by simply following the arcs from i to r on the optimal spanning tree. Also, the length of the shortest path is $y_r^* - y_i^*$.

While the network simplex method can be used to solve the shortest-path problem, there are faster algorithms designed especially for it. To describe these algorithms, let us denote the distance from i to r by v_i. These distances (or approximations thereof) are called *labels* in the networks literature. Some algorithms compute these distances systematically in a certain order. These algorithms are called *label-setting algorithms*. Other algorithms start with an estimate for these labels and then iteratively correct the estimates until the optimal values are found. Such algorithms are called *label-correcting algorithms*.

Note that if we set y_r^* to zero in the network flow solution, then the labels are simply the negative of the optimal dual variables. In the following subsections, we shall describe simple examples of label-setting and label-correcting algorithms.

3.2. A Label-Correcting Algorithm. To describe a label-correcting algorithm, we need to identify a system of equations that characterize the shortest-path distances. First of all, clearly

$$v_r = 0.$$

What can we say about the labels at other nodes, say, node i? Suppose that we select an arc (i, j) that leaves node i. If we were to travel along this arc and then, from node j, travel along the shortest path to r, then the distance to the root would be $c_{ij} + v_j$. So, from node i, we should select the arc that minimizes these distances. This selection will then give the shortest distance from i to r. That is,

$$v_i = \min\{c_{ij} + v_j : (i, j) \in \mathcal{A}\}, \qquad i \neq r. \tag{15.3}$$

The argument we have just made is called the *principle of dynamic programming*, and equation (15.3) is called *Bellman's equation*. Dynamic programming is a whole subject of its own—we shall only illustrate some of its basic ideas by our study of the shortest-path problem. In the dynamic programming literature, the set of v_i's viewed as a function defined on the nodes is called the *value function* (hence the notation).

From Bellman's equation, it is easy to identify the arcs one would travel on in a shortest-path route to the root. Indeed, these arcs are given by

$$\mathcal{T} = \{(i, j) \in \mathcal{A} : v_i = c_{ij} + v_j\}.$$

This set of arcs may contain alternate shortest paths to the root, and so the set is not necessarily a tree. Nonetheless, any path that follows these arcs will get to the root on a shortest-path route.

3.2.1. *Method of Successive Approximation.* Bellman's equation is an implicit system of equations for the values v_i, $i \in \mathcal{N}$. Implicit equations such as this arise frequently and beg to be solved by starting with a guess at the solution, using this guess in the right-hand side, and computing a new guess by evaluating the right-hand side. This approach is called the *method of successive approximations*. To apply it to the shortest-path problem, we initialize the labels as follows:

$$v_i^{(0)} = \begin{cases} 0 & i = r \\ \infty & i \neq r. \end{cases}$$

Then the updates are computed using Bellman's equation:

$$v_i^{(k+1)} = \begin{cases} 0 & i = r \\ \min\{c_{ij} + v_j^{(k)} : (i, j) \in \mathcal{A}\} & i \neq r. \end{cases}$$

3.2.2. *Efficiency.* The algorithm stops when an update leaves all the v_i's unchanged. It turns out that the algorithm is guaranteed to stop in no more than m iterations. To see why, it suffices to note that $v_i^{(k)}$ has a very simple description: it is the length of the shortest path from i to r that has k or fewer arcs in the path. (It is not hard to convince yourself with an induction on k that this is correct, but a pedantic proof requires introducing a significant amount of added notation that we wish to avoid.) Hence, the label-correcting algorithm cannot take more than m iterations, since every shortest path can visit each node at most once. Since each iteration involves looking at every arc of the network, it follows that the number of additions/comparisons needed to solve a shortest-path problem using the label-correcting algorithm is about nm.

Initialize:
$\mathcal{F} = \emptyset$
$v_j = \begin{cases} 0 & j = r, \\ \infty & j \neq r. \end{cases}$
while $(|\mathcal{F}^c| > 0)\{$
$j = \text{argmin}\{v_k : k \notin \mathcal{F}\}$
$\mathcal{F} \leftarrow \mathcal{F} \cup \{j\}$
for each i for which $(i, j) \in \mathcal{A}$ and $i \notin \mathcal{F}$ {
if $(c_{ij} + v_j < v_i)$ {
$v_i = c_{ij} + v_j$
$h_i = j$
}
}
}

FIGURE 15.3. Dijkstra's shortest-path algorithm.

3.3. A Label-Setting Algorithm. In this section, we describe *Dijkstra's algorithm* for solving shortest-path problems. The data structures that are carried from one iteration to the next are a set $\mathcal{F}$ of *finished* nodes and two arrays indexed by the nodes of the graph. The first array, v_j, $j \in \mathcal{N}$, is just the array of labels. The second array, h_i, $i \in \mathcal{N}$, indicates the next node to visit from node i in a shortest path. As the algorithm proceeds, the set $\mathcal{F}$ contains those nodes for which the shortest path has already been found. This set starts out empty. Each iteration of the algorithm adds one node to it. This is why the algorithm is called a label-setting algorithm, since each iteration sets one label to its optimal value. For finished nodes, the labels are fixed at their optimal values. For each unfinished node, the label has a temporary value, which represents the length of the shortest path from that node to the root, subject to the condition that all intermediate nodes on the path must be finished nodes. At those nodes for which no such path exists, the temporary label is set to infinity (or, in practice, a large positive number).

The algorithm is initialized by setting all the labels to infinity except for the root node, whose label is set to 0. Also, the set of finished nodes is initialized to the empty set. Then, as long as there remain unfinished nodes, the algorithm selects an unfinished node j having the smallest temporary label, adds it to the set of finished nodes, and then updates each unfinished "upstream" neighbor i by setting its label to $c_{ij} + v_j$ if this value is smaller than the current value v_i. For each neighbor i whose label gets changed, h_i is set to j. The algorithm is summarized in Figure 15.3.

4. Upper-Bounded Network Flow Problems

Some real-world network flow problems involve upper bounds on the amount of flow that an arc can handle. There are modified versions of the network simplex method that allow one to handle such upper bounds implicitly, but we shall simply show how to reduce an upper-bounded network flow problem to one without upper bounds.

Let us consider just one arc, (i, j), in a network flow problem. Suppose that there is an upper bound of u_{ij} on the amount of flow that this arc can handle. We can express this bound as an extra constraint:

$$0 \leq x_{ij} \leq u_{ij}.$$

Introducing a slack variable, t_{ij}, we can rewrite these bound constraints as

$$\begin{aligned} x_{ij} + t_{ij} &= u_{ij} \\ x_{ij},\ t_{ij} &\geq 0. \end{aligned}$$

If we look at the flow balance constraints and focus our attention on the variables x_{ij} and t_{ij}, we see that they appear in only three constraints: the flow balance constraints for nodes i and j and the upper bound constraint,

$$\begin{array}{cccc} \cdots & -x_{ij} & \cdots & = -b_i \\ \cdots & x_{ij} & \cdots & = -b_j \\ & x_{ij} & +t_{ij} & = u_{ij}. \end{array}$$

If we subtract the last constraint from the second one, we get

$$\begin{array}{cccc} \cdots & -x_{ij} & \cdots & = -b_i \\ \cdots & & \cdots -t_{ij} & = -b_j - u_{ij} \\ & x_{ij} & +t_{ij} & = u_{ij}. \end{array}$$

Note that we have restored a network structure in the sense that each column again has one $+1$ and one -1 coefficient. To make a network picture, we need to create a new node (corresponding to the third row). Let us call this node k. The network transformation is shown in Figure 15.4.

We can use the above transformation to derive optimality conditions for upper-bounded network flow problems. Indeed, let us consider an optimal solution to the transformed problem. Clearly, if x_{ik} is zero, then the corresponding dual slack $z_{ik} = y_i + c_{ij} - y_k$ is nonnegative:

$$y_i + c_{ij} - y_k \geq 0. \tag{15.4}$$

Furthermore, the back-flow x_{jk} must be at the upper bound rate:

$$x_{jk} = u_{ij}.$$

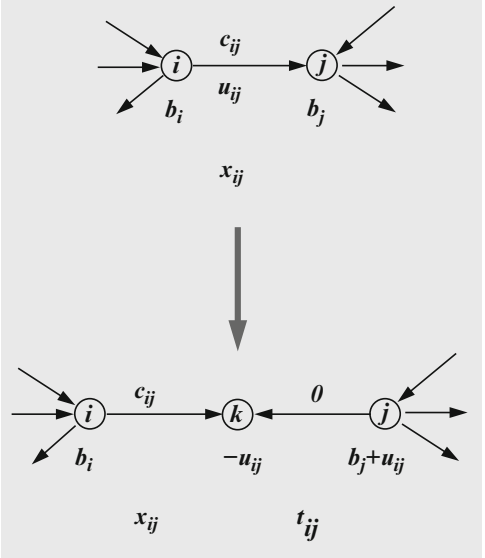

FIGURE 15.4. Adding a new node, k, to accommodate an arc (i, j) having an upper bound u_{ij} on its flow capacity.

Hence, by complementarity, the corresponding dual slack must vanish:

$$z_{jk} = y_j - y_k = 0. \tag{15.5}$$

Combining (15.4) with (15.5), we see that

$$y_i + c_{ij} \geq y_j.$$

On the other hand, if the flow on arc (i, k) is at the capacity value, then the back-flow on arc (j, k) must vanish. The complementarity conditions then say that

$$\begin{aligned} z_{ik} &= y_i + c_{ij} - y_k = 0 \\ z_{jk} &= y_j - y_k \geq 0. \end{aligned}$$

Combining these two statements, we get

$$y_i + c_{ij} \leq y_j.$$

Finally, if $0 < x_{ij} < u_{ij}$, then both slack variables vanish, and this implies that

$$y_i + c_{ij} = y_j.$$

These properties can then be summarized as follows:

$$\begin{aligned} x_{ij} = 0 &\Longrightarrow y_i + c_{ij} \geq y_j \\ x_{ij} = u_{ij} &\Longrightarrow y_i + c_{ij} \leq y_j \\ 0 < x_{ij} < u_{ij} &\Longrightarrow y_i + c_{ij} = y_j. \end{aligned} \tag{15.6}$$

While upper-bounded network flow problems have important applications, we admit that our main interest in them is more narrowly focused. It stems from their

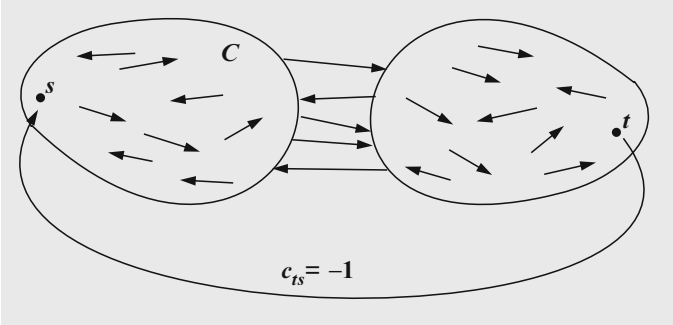

FIGURE 15.5. A cut set C for a maximum flow problem.

relation to an important theorem called the Max-Flow Min-Cut Theorem. We shall state and prove this theorem in the next section. The only tool we need to prove this theorem is the above result giving the complementarity conditions when there are upper bounds on arcs. So on with the show.

5. The Maximum-Flow Problem

The subject of this section is the class of problems called *maximum-flow problems*. These problems form an important topic in the theory of network flows. There are very efficient algorithms for solving them, and they appear as subproblems in many algorithms for the general network flow problem. However, our aim is rather modest. We wish only to expose the reader to one important theorem in this subject, which is called the Max-Flow Min-Cut Theorem.

Before we can state this theorem we need to set up the situation. Suppose that we are given a network $(\mathcal{N}, \mathcal{A})$, a distinguished node $s \in \mathcal{N}$ called the *source node*, a distinguished node $t \in \mathcal{N}$ called the *sink node*, and upper bounds on the arcs of the network u_{ij}, $(i, j) \in \mathcal{A}$. For simplicity, we shall assume that the upper bounds are all finite (although this is not really necessary). The objective is to "push" as much flow from s to t as possible.

To solve this problem, we can convert it to an upper-bounded network flow problem as follows. First, let $c_{ij} = 0$ for all arcs $(i, j) \in \mathcal{A}$, and let $b_i = 0$ for every node $i \in \mathcal{N}$. Then add one extra arc (t, s) connecting the sink node t back to the source node s, put a negative cost on this arc (say, $c_{ts} = -1$), and let it have infinite capacity $u_{ts} = \infty$. Since the only nonzero cost is actually negative, it follows that we shall actually make a profit by letting more and more flow circulate through the network. But the upper bound on the arc capacities limits the amount of flow that it is possible to push through.

In order to state the Max-Flow Min-Cut Theorem, we must define what we mean by a cut. A *cut*, C, is a set of nodes that contains the source node but does not contain the sink node (see Figure 15.5). The *capacity* of a cut is defined as

$$\kappa(C) = \sum_{\substack{i \in C \\ j \notin C}} u_{ij}.$$

Note that here and elsewhere in this section, the summations are over "original" arcs that satisfy the indicated set membership conditions. That is, they don't include the arc that we added connecting from t back to s. (If it did, the capacity of every cut would be infinite—which is clearly not our intention.)

Flow balance tells us that the total flow along original arcs connecting the cut set C to its complement minus the total flow along original arcs that span these two sets in the opposite direction must equal the amount of flow on the artificial arc (t, s). That is,

$$x_{ts} = \sum_{\substack{i \in C \\ j \notin C}} x_{ij} - \sum_{\substack{i \notin C \\ j \in C}} x_{ij}. \tag{15.7}$$

We are now ready to state the Max-Flow Min-Cut Theorem.

THEOREM 15.1. *The maximum value of x_{ts} equals the minimum value of $\kappa(C)$.*

PROOF. The proof follows the usual sort of pattern common in subjects where there is a sort of duality theory. First of all, we note that it follows from (15.7) that

$$x_{ts} \leq \kappa(C) \tag{15.8}$$

for every feasible flow and every cut set C. Then all that is required is to exhibit a feasible flow and a cut set for which this inequality is an equality.

Let x^*_{ij}, $(i, j) \in \mathcal{A}$, denote the optimal values of the primal variables, and let y^*_i, $i \in \mathcal{N}$, denote the optimal values of the dual variables. Then the complementarity conditions (15.6) imply that

$$x^*_{ij} = 0 \quad \text{whenever} \quad y^*_i + c_{ij} > y^*_j \tag{15.9}$$

$$x^*_{ij} = u_{ij} \quad \text{whenever} \quad y^*_i + c_{ij} < y^*_j. \tag{15.10}$$

In particular,

$$y^*_t - 1 \geq y^*_s$$

(since $u_{ts} = \infty$). Put $C^* = \{k : y^*_k \leq y^*_s\}$. Clearly, C^* is a cut.

Consider an arc having its tail in C^* and its head in the complement of C^*. It follows from the definition of C^* that $y^*_i \leq y^*_s < y^*_j$. Since c_{ij} is zero, we see from (15.10) that $x^*_{ij} = u_{ij}$.

Now consider an original arc having its tail in the complement of C^* and its head in C^* (i.e., bridging the two sets in the opposite direction). It follows then that $y^*_j \leq y^*_s < y^*_i$. Hence, we see from (15.9) that $x^*_{ij} = 0$.

Combining the observations of the last two paragraphs with (15.7), we see that

$$x^*_{ts} = \sum_{\substack{i \in C \\ j \notin C}} u_{ij} = \kappa(C^*).$$

In light of (15.8), the proof is complete. □

Exercises

15.1 Solve the transportation problem given in (15.1), using (15.2) for the starting tree solution.

15.2 Solve the following linear programming problem:

$$\begin{array}{lrcl} \text{maximize} & 7x_1 - 3x_2 + 9x_3 + 2x_4 & & \\ \text{subject to} & x_1 + x_2 & \leq & 1 \\ & x_3 + x_4 & \leq & 1 \\ & x_1 + x_3 & \geq & 1 \\ & x_2 + x_4 & \geq & 1 \\ & x_1, x_2, x_3, x_4 & \geq & 0. \end{array}$$

(Note: there are two greater-than-or-equal-to constraints.)

15.3 Bob, Carol, David, and Alice are stranded on a desert island. Bob and David each would like to give their affection to Carol or to Alice. Food is the currency of trade for this starving foursome. Bob is willing to pay Carol 7 clams if she will accept his affection. David is even more keen and is willing to give Carol 9 clams if she will accept it. Both Bob and David prefer Carol to Alice (sorry Alice). To quantify this preference, David is willing to pay Alice only 2 clams for his affection. Bob is even more averse: he says that Alice would have to pay him for it. In fact, she'd have to pay him 3 clams for his affection. Carol and Alice, being proper young women, will accept affection from one and only one of the two guys. Between the two of them they have decided to share the clams equally between them and hence their objective is simply to maximize the total number of clams they will receive. Formulate this problem as a transportation problem. Solve it.

15.4 *Project Scheduling*.This problem deals with the creation of a project schedule; specifically, the project of building a house. The project has been divided into a set of jobs. The problem is to schedule the time at which each of these jobs should start and also to predict how long the project will take. Naturally, the objective is to complete the project as quickly as possible (time is money!). Over the duration of the project, some of the jobs can be done concurrently. But, as the following table shows, certain jobs definitely can't start until others are completed.

Job	Duration (weeks)	Must be preceded by
0. Sign contract with buyer	0	–
1. Framing	2	0
2. Roofing	1	1
3. Siding	3	1
4. Windows	2.5	3
5. Plumbing	1.5	3
6. Electrical	2	2,4
7. Inside finishing	4	5,6
8. Outside painting	3	2,4
9. Complete the sale to buyer	0	7,8

One possible schedule is the following:

Job	Start time
0. Sign contract with buyer	0
1. Framing	1
2. Roofing	4
3. Siding	6
4. Windows	10
5. Plumbing	9
6. Electrical	13
7. Inside finishing	16
8. Outside painting	14
9. Complete the sale to buyer	21

With this schedule, the project duration is 21 weeks (*the difference between the start times of jobs 9 and 0*).

To model the problem as a linear program, introduce the following decision variables:

$$t_j = \text{the start time of job } j.$$

(a) Write an expression for the objective function, which is to minimize the project duration.

(b) For each job j, write a constraint for each job i that must precede j; the constraint should ensure that job j doesn't start until job i is finished. These are called *precedence constraints*.

15.5 *Continuation.* This problem generalizes the specific example of the previous problem. A project consists of a set of jobs $\mathcal{J}$. For each job $j \in \mathcal{J}$ there is a certain set $\mathcal{P}_j$ of other jobs that must be completed before job j can be started. (This is called the set of *predecessors* of job j.) One of the jobs, say s, is the starting job; it has no predecessors. Another job, say t, is the final (or terminal) job; it is not the predecessor of any other job. The time it will take to do job j is denoted d_j (the *duration* of the job).

The problem is to decide what time each job should begin so that no job begins before its predecessors are finished, and the duration of the entire project is minimized. Using the notations introduced above, write out a complete description of this linear programming problem.

15.6 *Continuation*. Let x_{ij} denote the dual variable corresponding to the precedence constraint that ensures job j doesn't start until job i finishes.
(a) Write out the dual to the specific linear program in Problem 15.4.
(b) Write out the dual to the general linear program in Problem 15.5.
(c) Describe how the optimal value of the dual variable x_{ij} can be interpreted.

15.7 *Continuation*. The project scheduling problem can be represented on a directed graph with arc weights as follows. The nodes of the graph correspond to the jobs. The arcs correspond to the precedence relations. That is, if job i must be completed before job j, then there is an arc pointing from node i to node j. The weight on this arc is d_i.
(a) Draw the directed graph associated with the example in Problem 15.4, being sure to label the nodes and write the weights beside the arcs.
(b) Return to the formulation of the dual from Problem 15.6(a). Give an interpretation of that dual problem in terms of the directed graph drawn in Part (a).
(c) Explain why there is always an optimal solution to the dual problem in which each variable x_{ij} is either 0 or 1.
(d) Write out the complementary slackness condition corresponding to dual variable x_{26}.
(e) Describe the dual problem in the language of the original project scheduling model.

15.8 *Continuation*. Here is an algorithm for computing optimal start times t_j:
1. List the jobs so that the predecessors of each job come before it in the list.
2. Put $t_0 = 0$.
3. Go down the list of jobs and for job j put $t_j = \max\{t_i + d_i : i \text{ is a predecessor of } j\}$.

(a) Apply this algorithm to the specific instance from Problem 15.4. What are the start times of each of the jobs? What is the project duration?
(b) Prove that the solution found in Part (a) is optimal by exhibiting a corresponding dual solution and checking the usual conditions for optimality (*Hint: The complementary slackness conditions may help you find a dual solution.*).

15.9 *Currency Arbitrage*. Consider the world's currency market. Given two currencies, say the Japanese Yen and the US Dollar, there is an exchange rate between them (currently about 110 Yen to the Dollar). It is always

true that, if you convert money from one currency to another and then back, you will end up with less than you started with. That is, the product of the exchange rates between any pair of countries is always less than one. However, it sometimes happens that a longer chain of conversions results in a gain. Such a lucky situation is called an *arbitrage*. One can use a linear programming model to find such situations when they exist.

Consider the following table of exchange rates (which is actual data from the Wall Street Journal on Nov 10, 1996):

```
param rate:
      USD      Yen     Mark     Franc    :=
USD   .        111.52  1.4987   5.0852
Yen   .008966  .       .013493  .045593
Mark  .6659    73.964  .        3.3823
Franc .1966    21.933  .29507   .
;
```

It is not obvious, but the USD→Yen→Mark→USD conversion actually makes $0.002 on each initial dollar.

To look for arbitrage possibilities, one can make a *generalized network model*, which is a network flow model with the unusual twist that a unit of flow that leaves one node arrives at the next node multiplied by a scale factor—in our example, the currency conversion rate. For us, each currency is represented by a node. There is an arc from each node to every other node. A flow of one unit out of one node becomes a flow of a different magnitude at the head node. For example, one dollar flowing out of the USD node arrives at the Franc node as 5.0852 Francs.

Let x_{ij} denote the flow from node (i.e. currency) i to node j. This flow is measured in the currency of node i.

One node is special; it is the *home* node, say the US Dollars (USD) node. At all other nodes, there must be flow balance.

(a) Write down the flow balance constraints at the 3 non-home nodes (Franc, Yen, and Mark).

At the home node, we assume that there is a supply of one unit (to get things started). Furthermore, at this node, flow balance will not be satisfied. Instead one expects a net inflow. If it is possible to make this inflow greater than one, then an arbitrage has been found. Let f be a variable that represents this inflow.

(b) Using variable f to represent net inflow to the home node, write a flow balance equation for the home node.

Of course, the primal objective is to maximize f.

(c) Using y_i to represent the dual variable associated with the primal constraint for currency i, write down the dual linear program. (Regard the primal variable f as a free variable.)

Now consider the general case, which might involve hundreds of currencies worldwide.

(d) Write down the model mathematically using x_{ij} for the flow leaving node i heading for node j (measured in the currency of node i), r_{ij} for the exchange rate when converting from currency i to currency j, and f for the net inflow at the home node i^*.
(e) Write down the dual problem.
(f) Can you give an interpretation for the dual variables? Hint: It might be helpful to think about the case where $r_{ji} = 1/r_{ij}$ for all i, j.
(g) Comment on the conditions under which your model will be unbounded and/or infeasible.

Notes

The Hitchcock problem was introduced by Hitchcock (1941). Dijkstra's algorithm was discovered by Dijkstra (1959).

The Max-Flow Min-Cut Theorem was proved independently by Elias et al. (1956), by Ford and Fulkerson (1956) and, in the restricted case where the upper bounds are all integers, by Kotzig (1956). Fulkerson and Dantzig (1955) also proved the Max-Flow Min-Cut Theorem. Their proof uses duality, which is particularly relevant to this chapter.

The classic references for dynamic programming are the books by Bellman (1957) and Howard (1960). Further discussion of label-setting and label-correcting algorithms can be found in the book by Ahuja et al. (1993).