in Section 11.1. The same idea can also be applied to more gener convex optimization problems; see, e.g., Bertsekas (1995b).

6.4. Dantzig-Wolfe decomposition was developed by Dantzig and Wo (1960). Example 6.2 is adapted from Bradley, Hax, and Magna (1977).

6.5. Stochastic programming began with work by Dantzig in the 1950 and has been extensively studied since then. Some books on this su ject are Kall and Wallace (1994), and Infanger (1993); Example 6 is adapted from the latter reference. The Benders decompositio method was developed by Benders (1962). It finds applications i other contexts as well, such as discrete optimization; see, e.g., Sch jver (1986), and Nemhauser and Wosley (1988).

Chapter 7

Network flow problems

Contents

Network flow problems (also known as *transshipment* problems) are the most frequently solved linear programming problems. They include as special cases, the assignment, transportation, maximum flow, and shortest path problems, and they arise naturally in the analysis and design of communication, transportation, and logistics networks, as well as in many other contexts.

The network flow problem is a special case of linear programming and any algorithm for linear programming can be directly applied. On the other hand, network flow problems have a special structure which results in substantial simplification of general methods (e.g., of the simplex method), as well as in new, special purpose, methods.

From a high level point of view, most of the available algorithms for network flow problems fall into one of three categories:

(a) **Primal methods.** These methods maintain and keep improving a primal feasible solution. The primal *simplex method*, presented in Section 7.3, is an important representative. An alternative algorithm is derived from first principles in Section 7.4.

(b) **Dual ascent methods.** These methods, which are discussed in Section 7.7, maintain a dual feasible solution and an auxiliary primal (usually infeasible) solution that satisfy complementary slackness. The dual variables are updated so as to increase the value of the dual objective and reduce the infeasibility of the complementary primal solution. The *Hungarian*, *primal-dual*, *relaxation*, and *dual simplex* methods fall in this general category.

(c) **Approximate dual ascent methods.** These methods are similar in spirit to the dual ascent methods, except that small decreases in the dual objective are allowed to occur and the complementary slackness conditions are only approximately enforced. The *auction* algorithm, which is discussed in Section 7.8, as well as the *ϵ-relaxation* and *preflow-push* methods, are of this type.

In this chapter, all three of the above mentioned algorithm types will be encountered. The chapter begins with a brief introduction to graphs (Section 7.1), that provides us with the language for studying network flow problems, and with a problem formulation (Section 7.2). We develop a number of general methods, but we also pay attention to special cases whose structure can be further exploited, such as the maximum flow problem (Section 7.5), the assignment problem (Section 7.8), and the shortest path problem (Section 7.9). We also discuss the minimum spanning tree problem (Section 7.10), which is not a network flow problem, but has a similar underlying graph structure. Throughout this chapter, our focus is on major algorithmic ideas, rather than on the refinements that can lead to better complexity estimates.

7.1 Graphs

Network flow problems are defined on graphs. In this section, we introduce graphs formally and provide a number of elementary definitions and properties.

Undirected graphs

An *undirected graph* $G = (\mathcal{N}, \mathcal{E})$ consists of a set $\mathcal{N}$ of *nodes* and a set $\mathcal{E}$ of *(undirected) arcs* or *edges*, where an edge e is an *unordered pair* of distinct nodes, that is, a two-element subset $\{i, j\}$ of $\mathcal{N}$; see Figure 7.1. Note that

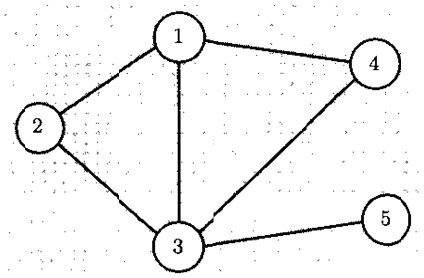


Figure 7.1: An undirected graph $G = (\mathcal{N}, \mathcal{E})$ with $\mathcal{N} = \{1,2,3,4,5\}$ and $\mathcal{E} = \big\{\{1,2\}, \{1,3\}, \{2,3\}, \{1,4\}, \{3,4\}, \{3,5\}\big\}$.

an undirected arc $\{i, j\}$ is one and the same object as the undirected arc $\{j, i\}$. Furthermore, "self-arcs" like $\{i, i\}$ are not allowed. We say that the arc $\{i, j\}$ is *incident* to nodes i and j, and these nodes are called the *endpoints* of the arc.

The *degree* of a node in an undirected graph is the number of arcs incident to that node. The degree of an undirected graph is defined as the maximum of the degrees of its nodes.

A *walk* from node i_1 to node i_t in an undirected graph is defined as a finite sequence of nodes $i_1, i_2, \ldots, i_t$ such that $\{i_k, i_{k+1}\} \in \mathcal{E}$, $k = 1, 2, \ldots, t-1$. A walk is called a *path* if it has no repeated nodes. A *cycle* is defined as a walk $i_1, i_2, \ldots, i_t$ such that the nodes $i_1, \ldots, i_{t-1}$ are distinct (and hence form a path) and $i_t = i_1$. In addition, we require the number $t-1$ of distinct nodes to be at least 3. This is in order to exclude a walk of the form i, j, i, where the same arc $\{i, j\}$ is traversed back and forth. An undirected graph is said to be *connected* if for every two distinct nodes $i, j \in \mathcal{N}$, there exists a path from i to j.

As an example, the graph in Figure 7.1 is connected. The sequence 1,2,3,1,4 is a walk but not a path. The sequence 1,2,3,1 is a cycle, and the sequence 1,3,5 is a path.

For undirected graphs, we will often denote the number of nodes b $|\mathcal{N}|$ or n, and the number of edges by $|\mathcal{E}|$ or m.

Directed graphs

A *directed graph* $G = (\mathcal{N},\mathcal{A})$ consists of a set $\mathcal{N}$ of *nodes* and a set of *(directed) arcs*, where a directed arc is an *ordered pair* (i,j) of distin nodes; see Figure 7.2. Our definition allows for both (i,j) and (j,i) to b

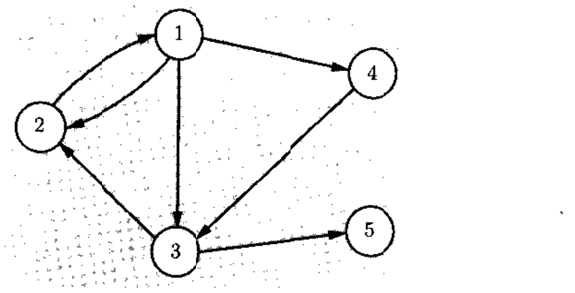


Figure 7.2: A directed graph $G = (\mathcal{N},\mathcal{A})$ with $\mathcal{N} = \{1,2,3,4,5\}$ and $\mathcal{A} = \{(1,2),\ (2,1),\ (1,3),\ (3,2),\ (1,4),\ (4,3),\ (3,5)\}$.

elements of the arc set $\mathcal{A}$, but self-arcs like (i,i) are not allowed.

For any arc (i,j), we say that i is the *start node* and j is the *en node*. The arc (i,j) is said to be *outgoing* from node i, *incoming* to nod j, and *incident* to both i and j. We define $I(i)$ and $O(i)$ as the set of star nodes (respectively, end nodes) of arcs that are incoming to (respectively outgoing from) node i. That is,

$$I(i) = \{j \in \mathcal{N} \mid (j,i) \in \mathcal{A}\},$$

and

$$O(i) = \{j \in \mathcal{N} \mid (i,j) \in \mathcal{A}\}.$$

Starting from a directed graph, we can construct a correspondin undirected graph by ignoring the direction of the arcs and by deleting re- peated arcs; for example, the directed graph in Figure 7.2 leads to the undirected graph in Figure 7.1. Under one possible interpretation, flow or movement in a directed arc is permitted only from the start node to the end node, whereas in an undirected arc, flow or movement is permitted in both directions. We say that a directed graph is *connected* if the resulting undirected graph is connected.

We now present a definition of walks in directed graphs; it is im- portant to note that this definition allows us to traverse an arc in either direction, irrespective of the arc's direction. More specifically, a *walk* is

defined as a sequence $i_1,\ldots,i_t$ of nodes, together with an associated sequence $a_1,\ldots,a_{t-1}$ of arcs such that for $k = 1,\ldots,t-1$, we have either $a_k = (i_k, i_{k+1})$ (in which case we say that a_k is a *forward* arc) or $a_k = (i_{k+1}, i_k)$ (in which case we say that a_k is a *backward* arc). Note that if i_k and i_{k+1} are consecutive nodes in a walk and if (i_k,i_{k+1}) and (i_{k+1},i_k) are both arcs of the underlying directed graph, then either arc can be used in the walk. The reason for including the arcs a_k in the definition of a walk is precisely to avoid such ambiguities.

A walk is said to be a *path* if all of its nodes $i_1,\ldots,i_t$ are distinct, and a *cycle* if the nodes $i_1,\ldots,i_{t-1}$ are distinct and $i_t = i_1$. Note that we allow a cycle to consist of only two distinct nodes (in contrast to our definition for the case of undirected graphs). Thus, a sequence $i,(i,j),j,(j,i),i$ is a bona fide cycle. Finally, a walk, path, or cycle is said to be *directed* if it only contains forward arcs.

For the graph shown in Figure 7.2, the sequence $1,(1,3),3,(3,2),2,(1,2),1,(1,4),4$ is a walk, but not a directed walk, because $(1,2)$ is a backward arc. The sequence $1,(1,3),3,(3,2),2,(2,1),1$ is a directed cycle. The sequence $1,(1,2),2,(2,1),1$ is also a directed cycle. The sequence $4,(4,3),3,(1,3),1,(1,2),2$ is a path, but not a directed path, because $(1,3)$ is a backward arc.

For directed graphs, we will often denote the number of nodes by $|\mathcal{N}|$ or n, and the number of arcs by $|\mathcal{A}|$ or m.

Trees

An undirected graph $G = (\mathcal{N},\mathcal{E})$ is called a *tree* if it is connected and has no cycles. If a node of a tree has degree equal to 1, it is called a *leaf*. See Figure 7.3 for an illustration.

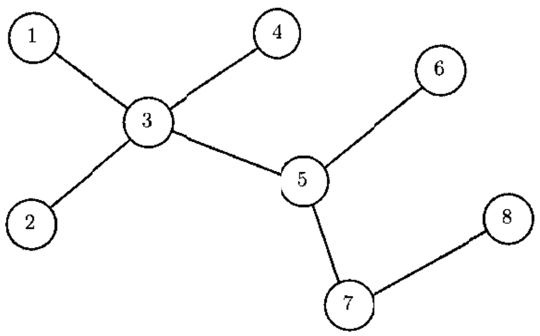


Figure 7.3: A tree with 8 nodes, 7 arcs, and 5 leaves. Note that if we were to add the arc $\{2,7\}$, a single cycle would be created, namely, 2,3,5,7,2.

We now present some important properties of trees that will be of u
later on (e.g., in the development of the simplex method, in Section 7.3

Theorem 7.1

(a) *Every tree with more than one node has at least one leaf.*

(b) *An undirected graph is a tree if and only if it is connected*
has $|\mathcal{N}| - 1$ arcs.

(c) *For any two distinct nodes i and j in a tree, there exists a uni*
path from i to j.

(d) *If we start with a tree and add a new arc, the resulting graph c*
tains exactly one cycle (as long as we do not distinguish betw
cycles involving the same set of nodes).

Proof.

(a) Consider a tree with more than one node and suppose that there
no leaves. Then, every node has degree greater than 1. (If the deg
of a node were 1, that node would be a leaf, and if it were 0, t
graph would not be connected.) Therefore, given a node and an
through which we enter the node, we can find a different arc throu
which we can exit. By repeating such a process, we must eventua
visit the same node twice, which implies that there exists a cyc
contradicting the definition of a tree.

(b) We first prove that every tree has $|\mathcal{N}| - 1$ arcs. This is trivially tr
if the tree has a single node. Consider now a tree that has more th
one node. Such a tree must have at least one leaf, by part (a). W
delete that leaf together with the single arc incident to that node. T
resulting graph is again a tree, because the deletion of a leaf cann
create a cycle or cause a graph to become disconnected. This proce
can be carried out $|\mathcal{N}| - 1$ times, until we are left with a single no
and, therefore, no arcs. Since at each stage there was exactly one a
deletion, we conclude that the original tree had $|\mathcal{N}| - 1$ arcs.

In order to prove the converse statement, let us consider a connecte
graph with $|\mathcal{N}| - 1$ arcs. If this graph contains a cycle, we can dele
one of the arcs in the cycle and still maintain connectivity. We r
peat this process as many times as needed, until we are left with
connected graph without any cycles, that is, a tree. We have alread
proved that a tree with $|\mathcal{N}|$ nodes must have $|\mathcal{N}| - 1$ arcs, and th
shows that the final tree has as many arcs as the original graph.
follows that no arc was deleted and the original graph was a tree t
start with.

(c) Suppose that there exist two different paths joining the same nodes
and j. By joining these two paths and by deleting any arcs that a

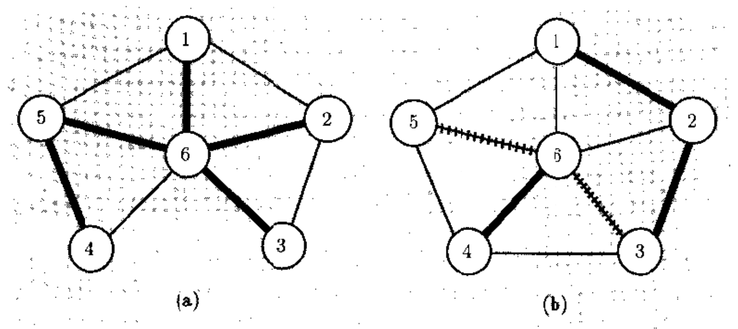


Figure 7.4: (a) An undirected graph. The thicker arcs form a spanning tree. (b) Another undirected graph. The arcs $\{1,2\}$, $\{2,3\}$, $\{4,6\}$ do not form any cycle. They can be augmented to form a spanning tree, e.g., by adding arcs $\{3,6\}$ and $\{5,6\}$.

common to both, we are left with one or more cycles, contradicting the definition of a tree.

(d) Consider a tree, and let us add an undirected arc $\{i,j\}$. Using part (b), the resulting graph must have $|\mathcal{N}|$ arcs. Therefore, it cannot be a tree, and must have a cycle. Any cycle created by this addition consists of the arc $\{i,j\}$ and a path from i to j. Since there exists a unique path from i to j [part (c)], it follows that a unique cycle has been created. □

Spanning trees

Given a connected undirected graph $G = (\mathcal{N}, \mathcal{E})$, let $\mathcal{E}_1$ be a subset of $\mathcal{E}$ such that $T = (\mathcal{N}, \mathcal{E}_1)$ is a tree. Such a tree is called a *spanning tree*. The following result will be used later on (in Sections 7.3 and 7.10) and is illustrated in Figure 7.4.

Theorem 7.2 *Let $G = (\mathcal{N}, \mathcal{E})$ be a connected undirected graph and let $\mathcal{E}_0$ be some subset of the set $\mathcal{E}$ of arcs. Suppose that the arcs in $\mathcal{E}_0$ do not form any cycles. Then, the set $\mathcal{E}_0$ can be augmented to a set $\mathcal{E}_1 \supset \mathcal{E}_0$ so that $(\mathcal{N}, \mathcal{E}_1)$ is a spanning tree.*

Proof. Let $G = (\mathcal{N}, \mathcal{E})$ be a connected undirected graph. Suppose that $\mathcal{E}_0 \subset \mathcal{E}$, and that the arcs in $\mathcal{E}_0$ do not form any cycles. If G is a tree, we may let $\mathcal{E}_1 = \mathcal{E}$ and we are done. Otherwise, G contains at least one cycle. A cycle cannot consist exclusively of arcs in $\mathcal{E}_0$, because of our assumption on $\mathcal{E}_0$. Let us choose and delete an arc that lies on a cycle and that does

We now present some important properties of trees that will be of use later on (e.g., in the development of the simplex method, in Section 7.3).

Theorem 7.1

(a) *Every tree with more than one node has at least one leaf.*

(b) *An undirected graph is a tree if and only if it is connected and has $|\mathcal{N}| - 1$ arcs.*

(c) *For any two distinct nodes i and j in a tree, there exists a unique path from i to j.*

(d) *If we start with a tree and add a new arc, the resulting graph contains exactly one cycle (as long as we do not distinguish between cycles involving the same set of nodes).*

Proof.

(a) Consider a tree with more than one node and suppose that there are no leaves. Then, every node has degree greater than 1. (If the degree of a node were 1, that node would be a leaf, and if it were 0, the graph would not be connected.) Therefore, given a node and an arc through which we enter the node, we can find a different arc through which we can exit. By repeating such a process, we must eventually visit the same node twice, which implies that there exists a cycle, contradicting the definition of a tree.

(b) We first prove that every tree has $|\mathcal{N}| - 1$ arcs. This is trivially true if the tree has a single node. Consider now a tree that has more than one node. Such a tree must have at least one leaf, by part (a). We delete that leaf together with the single arc incident to that node. The resulting graph is again a tree, because the deletion of a leaf cannot create a cycle or cause a graph to become disconnected. This process can be carried out $|\mathcal{N}| - 1$ times, until we are left with a single node and, therefore, no arcs. Since at each stage there was exactly one arc deletion, we conclude that the original tree had $|\mathcal{N}| - 1$ arcs.

In order to prove the converse statement, let us consider a connected graph with $|\mathcal{N}| - 1$ arcs. If this graph contains a cycle, we can delete one of the arcs in the cycle and still maintain connectivity. We repeat this process as many times as needed, until we are left with a connected graph without any cycles, that is, a tree. We have already proved that a tree with $|\mathcal{N}|$ nodes must have $|\mathcal{N}| - 1$ arcs, and this shows that the final tree has as many arcs as the original graph. It follows that no arc was deleted and the original graph was a tree to start with.

(c) Suppose that there exist two different paths joining the same nodes i and j. By joining these two paths and by deleting any arcs that are common to both, we are left with one or more cycles, contradicting the definition of a tree.

(d) Consider a tree, and let us add an undirected arc $\{i, j\}$. Using part (b), the resulting graph must have $|\mathcal{N}|$ arcs. Therefore, it cannot be a tree, and must have a cycle. Any cycle created by this addition consists of the arc $\{i, j\}$ and a path from i to j. Since there exists a unique path from i to j [part (c)], it follows that a unique cycle has been created. □

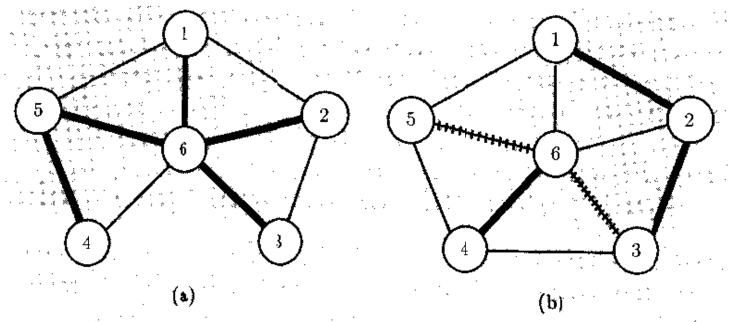


Figure 7.4: (a) An undirected graph. The thicker arcs form a spanning tree. (b) Another undirected graph. The arcs $\{1, 2\}$, $\{2, 3\}$, $\{4, 6\}$ do not form any cycle. They can be augmented to form a spanning tree, e.g., by adding arcs $\{3, 6\}$ and $\{5, 6\}$.

Spanning trees

Given a connected undirected graph $G = (\mathcal{N}, \mathcal{E})$, let $\mathcal{E}_1$ be a subset of $\mathcal{E}$ such that $T = (\mathcal{N}, \mathcal{E}_1)$ is a tree. Such a tree is called a *spanning tree*. The following result will be used later on (in Sections 7.3 and 7.10) and is illustrated in Figure 7.4.

Theorem 7.2 *Let $G = (\mathcal{N}, \mathcal{E})$ be a connected undirected graph and let $\mathcal{E}_0$ be some subset of the set $\mathcal{E}$ of arcs. Suppose that the arcs in $\mathcal{E}_0$ do not form any cycles. Then, the set $\mathcal{E}_0$ can be augmented to a set $\mathcal{E}_1 \supset \mathcal{E}_0$ so that $(\mathcal{N}, \mathcal{E}_1)$ is a spanning tree.*

Proof. Let $G = (\mathcal{N}, \mathcal{E})$ be a connected undirected graph. Suppose that $\mathcal{E}_0 \subset \mathcal{E}$, and that the arcs in $\mathcal{E}_0$ do not form any cycles. If G is a tree, we may let $\mathcal{E}_1 = \mathcal{E}$ and we are done. Otherwise, G contains at least one cycle. A cycle cannot consist exclusively of arcs in $\mathcal{E}_0$, because of our assumption on $\mathcal{E}_0$. Let us choose and delete an arc that lies on a cycle and that does not belong to $\mathcal{E}_0$. The resulting graph is still connected. By repeating this process as many times as needed, we end up with a connected graph $(\mathcal{N}, \mathcal{E}_1)$ without any cycles, hence a tree. In addition, since the arcs in $\mathcal{E}_0$ are never deleted, we have $\mathcal{E}_0 \subset \mathcal{E}_1$. □

7.2 Formulation of the network flow problem

A *network* is a directed graph $G = (\mathcal{N}, \mathcal{A})$ together with some additional numerical information, such as numbers b_i representing the external *supply* to each node $i \in \mathcal{N}$, nonnegative (possibly infinite) numbers u_{ij} representing the *capacity* of each arc $(i,j) \in \mathcal{A}$, and numbers c_{ij} representing the cost per unit of flow along arc (i,j).

We visualize a network by thinking of some material that flows on each arc. We use f_{ij} to denote the amount of flow through arc (i,j). The supply b_i is interpreted as the amount of flow that enters the network from the outside, at node i. In particular, node i is called a *source* if $b_i > 0$, and a *sink* if $b_i < 0$. If node i is a sink, the quantity $|b_i|$ is sometimes called the *demand* at node i. We impose the following conditions on the flow variables f_{ij}, $(i,j) \in \mathcal{A}$:

$$b_i + \sum_{j \in I(i)} f_{ji} = \sum_{j \in O(i)} f_{ij}, \qquad \forall\, i \in \mathcal{N}, \tag{7.1}$$

$$0 \le f_{ij} \le u_{ij}, \qquad \forall\, (i,j) \in \mathcal{A}. \tag{7.2}$$

Equation (7.1) is a flow conservation law: it states that the amount of flow into a node i must be equal to the total flow out of that node. Equation (7.2) simply requires that the flow through an arc must be nonnegative and cannot exceed the capacity of the arc. Any vector with components f_{ij}, $(i,j) \in \mathcal{A}$, will be called a *flow*. If it also satisfies the constraints (7.1)-(7.2), it will be called a *feasible flow*.

By summing both sides of Eq. (7.1) over all $i \in \mathcal{N}$, we obtain

$$\sum_{i \in \mathcal{N}} b_i = 0,$$

which means that the total flow from the environment into the network (at the sources) must be equal to the total flow from the network (at the sinks) to the environment. From now on, we will always assume that the condition $\sum_{i \in \mathcal{N}} b_i = 0$ holds, because otherwise no flow vector could satisfy the flow conservation constraints, and we would have an infeasible problem.

The general minimum cost network flow problem deals with the minimization of a linear cost function of the form

$$\sum_{(i,j) \in \mathcal{A}} c_{ij} f_{ij},$$

over all feasible flows. We observe that this is a linear programming problem. If $u_{ij} = \infty$ for all $(i,j) \in \mathcal{A}$, we say that the problem is *uncapacitated*; otherwise, we say that it is *capacitated*. Note that in the uncapacitated case, we only have equality and nonnegativity constraints, and the problem is in standard form.

We now provide an overview of important special cases of the network flow problem; most of them will be studied later in this chapter.

The shortest path problem

For any directed path in a network, we define its *length* as the sum of the costs of all arcs on the path. We wish to find a *shortest path*, that is, a directed path from a given origin node to a given destination node whose length is smallest. This problem is studied in Section 7.9, where we show that it can be formulated as a network flow problem, under a certain assumption on the arc lengths.

The maximum flow problem

In the maximum flow problem, we wish to determine the largest possible amount of flow that can be sent from a given source node to a given sink node, without exceeding the arc capacities. This problem is studied in Section 7.5.

The transportation problem

Let there be m suppliers and n consumers. The ith supplier can provide s_i units of a certain good and the jth consumer has a demand for d_j units. We assume that the total supply $\sum_{i=1}^m s_i$ is equal to the total demand $\sum_{j=1}^n d_j$. Finally, we assume that the transportation of goods from the ith supplier to the jth consumer carries a cost of c_{ij} per unit of goods transported. The problem is to transport the goods from the suppliers to the consumers at minimum cost. Let f_{ij} be the amount of goods transported from the ith supplier to the jth consumer. We then have the following problem:

$$\begin{array}{lll}
\text{minimize} & \displaystyle\sum_{i=1}^m \sum_{j=1}^n c_{ij} f_{ij} & \\
\text{subject to} & \displaystyle\sum_{i=1}^m f_{ij} = d_j, & j = 1, \ldots, n, \\
& \displaystyle\sum_{j=1}^n f_{ij} = s_i, & i = 1, \ldots, m, \\
& f_{ij} \ge 0, & \forall\, i, j.
\end{array}$$

The first equality constraint specifies that the demand d_j of each consum
must be met; the second equality constraint requires that the entire supp
s_i of each supplier must be shipped. This is a special case of the unc
pacitated network flow problem, where the underlying graph has a speci
structure; see Figure 7.5. It turns out that every network flow problem ca

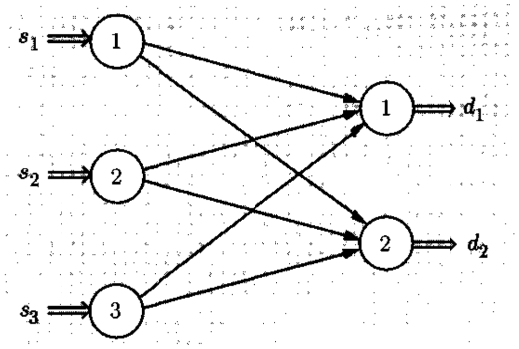


Figure 7.5: A network corresponding to a transportation problem with three suppliers and two consumers.

be transformed into an equivalent transportation problem (Exercises 7.
and 7.6). Consequently, any algorithm for the transportation problem ca
be adapted and can be used to solve general network flow problems. Fo
this reason, the initial development and testing of new algorithms is ofte
carried out for the special case of transportation problems.

The assignment problem

The assignment problem is a special case of the transportation problem, where the number of suppliers is equal to the number of consumers, each supplier has unit supply, and each consumer has unit demand. As will be proved later in this chapter, one can always find an optimal solution in which every f_{ij} is either 0 or 1. This means that for each i there will be a unique and distinct j for which $f_{ij} = 1$, and we can say that the ith supplier is *assigned* to the jth consumer; this justifies the name of this problem.

Variants of the network flow problem

There are several variants of the network flow problem all of which can be shown to be equivalent to each other. For example, we have already mentioned that every network flow problem is equivalent to a transportation problem. We now discuss some more examples.

(a) *Every network flow problem can be reduced to one with exactly one source and exactly one sink node.* This is illustrated in Figure 7.6.

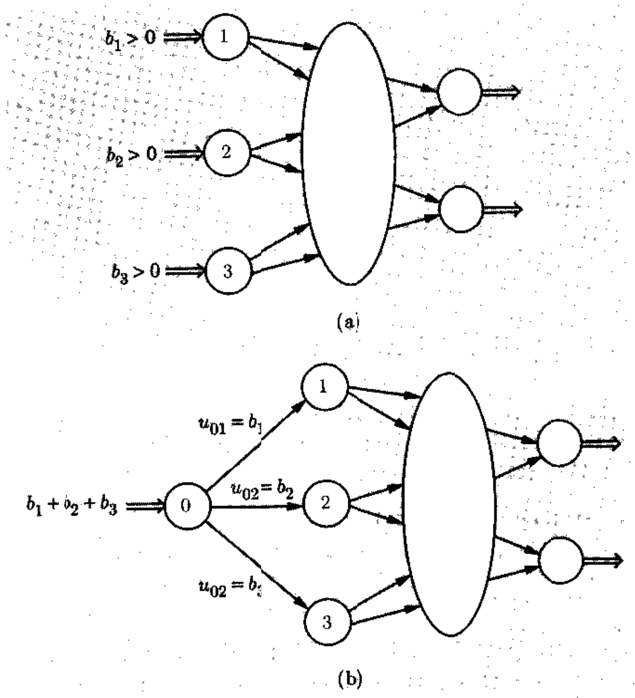


Figure 7.6: (a) A network with three source nodes. (b) A network with only one source node. The costs of the new arcs are zero. Because of the way that the arc capacities u_{0i} are chosen ($u_{0i} = b_i$, $i = 1, 2, 3$), exactly b_i units must flow on each arc $(0, i)$, $i = 1, 2, 3$. The reduction to a network with a single sink node is similar.

(b) *Every network flow problem can be reduced to one without sources or sinks.* (Problems in which all of the supplies are zero are called *circulation* problems.) Consider, without loss of generality, a network with a single source s and a single sink t. We introduce a new arc (t, s) whose capacity u_{ts} is equal to b_s and whose unit cost is $c_{ts} = -M$, where M is a large number; see Figure 7.7. Since M is large, an optimal solution to the circulation problem will try to set f_{ts} to b_s, which has the same effect as having a supply of b_s at node s. If an optimal solution to the circulation problem does not succeed in setting f_{ts} to b_s, this means that there is no way of shipping b_s units of flow from s to t, and the original problem is infeasible.

(c) *Node capacities.* Suppose that we have an upper bound of g_i on the total flow that can enter a given node i; for example, if i is a source

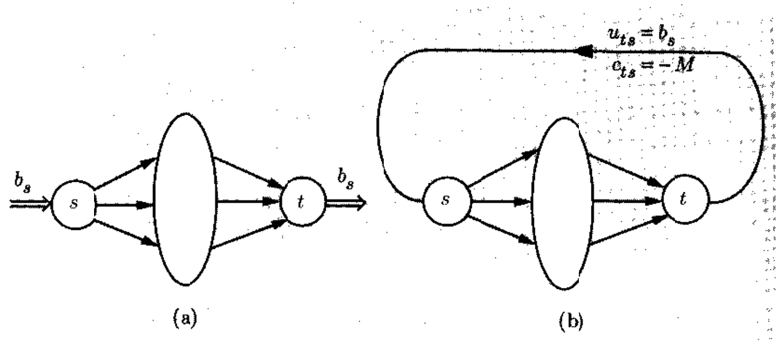


Figure 7.7: (a) A network. (b) An equivalent circulation problem.

node, we may have a constraint

$$b_i + \sum_{j \in I(i)} f_{ji} \le g_i.$$

By splitting node i into two nodes i and i', and by letting g_i be the capacity of arc (i, i'), we are back to the case where we only have arc capacities; see Figure 7.8.

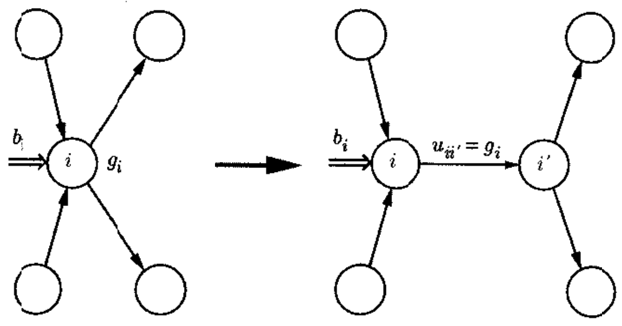


Figure 7.8: Transformation of a node capacity into an arc capacity.

(d) *Lower bounds on the arc flows.* Suppose that we add constraints of the form $f_{ij} \ge d_{ij}$, where d_{ij} are given scalars. The resulting problem can be reduced to an equivalent problem in which every d_{ij} is equal to zero. Exercise 7.7 provides some guidance as to how this can be accomplished.

A concise formulation

We now discuss how to rewrite the network flow problem, and especially the flow conservation constraint, in more economical matrix-vector notation. We assume that $\mathcal{N} = \{1, \ldots, n\}$ and we let m be the number of arcs. Let us fix a particular ordering of the arcs, and let $\mathbf{f}$ be the vector of flows that results when the components f_{ij} are ordered accordingly. We define the *node-arc incidence matrix* $\mathbf{A}$ as follows: its dimensions are $n \times m$ (each row corresponds to a node and each column to an arc) and its (i, k)th entry a_{ik} is associated with the ith node and the kth arc. We let

$$a_{ik} = \begin{cases} 1, & \text{if } i \text{ is the start node of the } k\text{th arc}, \\ -1, & \text{if } i \text{ is the end node of the } k\text{th arc}, \\ 0, & \text{otherwise.} \end{cases}$$

Thus, every column of $\mathbf{A}$ has exactly two nonzero entries, one equal to $+1$, and one equal to -1, indicating the start and the end node of the corresponding arc.

Example 7.1 Consider the directed graph of Figure 7.2 and let us use the following ordering of the arcs: $(1,2), (2,1), (3,2), (4,3), (1,4), (1,3), (3,5)$. The corresponding node-arc incidence matrix is

$$\mathbf{A} = \begin{bmatrix} 1 & -1 & 0 & 0 & 1 & 1 & 0 \\ -1 & 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & -1 & 1 \\ 0 & 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 \end{bmatrix}.$$

Let us now focus on the ith row of $\mathbf{A}$, denoted by $\mathbf{a}_i'$ (this is the row associated with node i). Nonzero entries indicate the arcs that are incident to node i; such entries are $+1$ or -1 depending on whether the arc is outgoing or incoming, respectively. Thus,

$$\mathbf{a}_i'\mathbf{f} = \sum_{j \in O(i)} f_{ij} - \sum_{j \in I(i)} f_{ji},$$

and the flow conservation constraint at node i [cf. Eq. (7.1)] can be written as

$$\mathbf{a}_i'\mathbf{f} = b_i,$$

or, in matrix notation,

$$\mathbf{A}\mathbf{f} = \mathbf{b},$$

where $\mathbf{b}$ is the vector $(b_1, \ldots, b_n)$.

We observe that the sum of the rows of $\mathbf{A}$ is equal to the zero vector; in particular, the rows of $\mathbf{A}$ are linearly dependent. Thus, the matrix $\mathbf{A}$ violates one of the basic assumptions underlying our development of the simplex method. As discussed in Chapter 2 (cf. Theorem 2.5 in Section 2.3), either the problem is infeasible or we can remove some of the equality constraints, without affecting the feasible set, so that the remaining constraints are linearly independent. We revisit this issue in the next section.

Circulations

We close by introducing some elementary concepts that are central to many network flow algorithms.

Any flow vector $\mathbf{f}$ (feasible or infeasible) that satisfies

$$\mathbf{Af} = \mathbf{0},$$

is called a *circulation*. Intuitively, we have flow conservation within the network and zero external supply or demand, which means that the flow "circulates" inside the network.

Let us now consider a cycle C. We let F and B be the set of forward and backward arcs of the cycle, respectively. The flow vector $\mathbf{h}^C$ with components

$$h_{ij}^C = \begin{cases} 1, & \text{if } (i,j) \in F, \\ -1, & \text{if } (i,j) \in B, \\ 0, & \text{otherwise.} \end{cases}$$

is called the *simple circulation* associated with the cycle C. It is easily seen that $\mathbf{h}^C$ satisfies

$$\mathbf{Ah}^C = \mathbf{0}, \tag{7.3}$$

and is indeed a circulation. The reason is that any two consecutive arcs on the cycle are either similarly oriented and carry the same amount of flow, or they have the opposite orientation and the sum of the flows that they carry is equal to 0; in either case, the net inflow to any node is zero; see Figure 7.9 We finally define the *cost of a cycle* C to be equal to

$$\mathbf{c}'\mathbf{h}^C = \sum_{(i,j)\in F} c_{ij} - \sum_{(i,j)\in B} c_{ij}.$$

If $\mathbf{f}$ is a flow vector, C is a cycle, and θ is a scalar, we say that the flow vector $\mathbf{f} + \theta\mathbf{h}^C$ is obtained from $\mathbf{f}$ by *pushing* θ units of flow around the cycle C. Note that the resulting cost change is θ times the cost $\mathbf{c}'\mathbf{h}^C$ of the cycle C.

7.3 The network simplex algorithm

In this section, we develop the details of the simplex method, as applied to the uncapacitated network flow problem

$$\begin{aligned} \text{minimize} \quad & \mathbf{c}'\mathbf{f} \\ \text{subject to} \quad & \mathbf{Af} = \mathbf{b} \\ & \mathbf{f} \geq \mathbf{0}, \end{aligned}$$

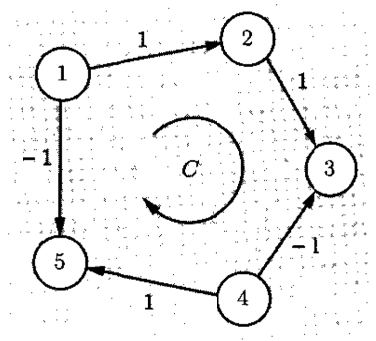


Figure 7.9: A cycle and the corresponding simple circulation. Arcs (4, 3) and (1, 5) are backward arcs and carry a flow of -1. Note that flow is conserved at each node.

where $\mathbf{A}$ is the node-arc incidence matrix of a directed graph $G = (\mathcal{N}, \mathcal{A})$. (Capacitated problems are briefly discussed at the end of this section.) The network simplex algorithm is widely used in practice, and is included in many commercial optimization codes, due to its simplicity and efficiency. In particular, it tends to run an order of magnitude faster than a general purpose simplex code applied to a network flow problem.

Due to our restriction to uncapacitated problems, we are dealing with a linear programming problem in standard form. We let m and n be the number of arcs and nodes, respectively. We therefore have m flow variables and n equality constraints which, unfortunately, is the exact opposite of the notational conventions used in earlier chapters.

There are two different ways of developing the network simplex method. The first is to go through the mechanics of the general simplex method and specialize each step to the present context. The second is to develop the algorithm from first principles and then to point out that it is a special case of the simplex method. We take a middle ground that proceeds along two parallel tracks; each step is justified from first principles, but its relation to the simplex method is also explained. The end result is an algorithm with a fairly intuitive structure.

Throughout this section, the following assumption will be in effect.

Assumption 7.1
(a) *We have* $\sum_{i\in\mathcal{N}} b_i = 0$.
(b) *The graph G is connected.*

Part (a) of this assumption is natural, because otherwise the problem is infeasible. Part (b) is also natural, because if the graph is not connected, then the problem can be decomposed into subproblems that can be treated independently.

As noted in Section 7.2, the rows of the matrix $\mathbf{A}$ sum to the zero vector and are therefore linearly dependent. In fact, the last constraint (flow conservation at node n) is a consequence of the flow conservation constraints at the other nodes, and can be omitted without affecting the feasible set. Let us define the *truncated node-arc incidence matrix* $\tilde{\mathbf{A}}$ to be the matrix of dimensions $(n-1) \times m$, which consists of the first $n-1$ rows of the matrix $\mathbf{A}$. Any column of $\tilde{\mathbf{A}}$ that corresponds to an arc of the form (i,n) has a single nonzero entry, equal to 1, at the ith row. Similarly, any column of $\tilde{\mathbf{A}}$ that corresponds to an arc of the form (n,i) has a single nonzero entry, equal to -1, at the ith row. All other columns of $\tilde{\mathbf{A}}$ have two nonzero entries. Let $\tilde{\mathbf{b}} = (b_1, \ldots, b_{n-1})$. We replace the original equality constraint $\mathbf{Af} = \mathbf{b}$ by the constraint $\tilde{\mathbf{A}}\mathbf{f} = \tilde{\mathbf{b}}$. We will see shortly that under Assumption 7.1, the matrix $\tilde{\mathbf{A}}$ has linearly independent rows.

Example 7.2 Consider the node-arc incidence matrix $\mathbf{A}$ in Example 7.1. The associated matrix $\tilde{\mathbf{A}}$ is given by

$$\tilde{\mathbf{A}} = \begin{bmatrix} 1 & -1 & 0 & 0 & 1 & 1 & 0 \\ -1 & 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & -1 & 1 \\ 0 & 0 & 0 & 1 & -1 & 0 & 0 \end{bmatrix}.$$

It can be verified that the matrix $\tilde{\mathbf{A}}$ has full rank. For example, the third, fourth, sixth, and seventh columns are linearly independent.

Trees and basic feasible solutions

We now introduce an important definition.

Definition 7.1 *A flow vector* $\mathbf{f}$ *is called a* **tree solution** *if it can be constructed by the following procedure.*

(a) *Pick a set $T \subset \mathcal{A}$ of $n-1$ arcs that form a tree when their direction is ignored.*

(b) *Let $f_{ij} = 0$ for every $(i,j) \notin T$.*

(c) *Use the flow conservation equation $\tilde{\mathbf{A}}\mathbf{f} = \tilde{\mathbf{b}}$ to determine the flow variables f_{ij}, for $(i,j) \in T$.*

A tree solution that also satisfies $\mathbf{f} \geq \mathbf{0}$, *is called a* **feasible tree solution.**

Step (c) in the above definition can be carried out using the following systematic procedure, illustrated in Figure 7.10:

(a) Call node n the *root* of the tree.

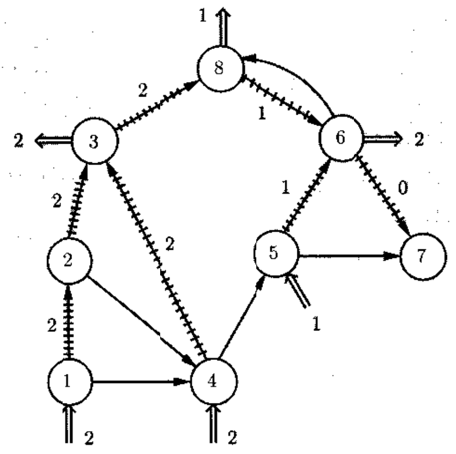


Figure 7.10: A network and a set of $n-1$ arcs (indicated by thatched lines) that form a tree. By setting the arc flows outside the tree to zero, we obtain $f_{12} = 2$, $f_{23} = 2$ and $f_{43} = 2$. We then use conservation of flow at node 3, to obtain $f_{38} = 2$. We also have $f_{56} = 1$ and $f_{67} = 0$. Using conservation of flow at node 6, we obtain $f_{86} = 1$. Note that this is a feasible tree solution.

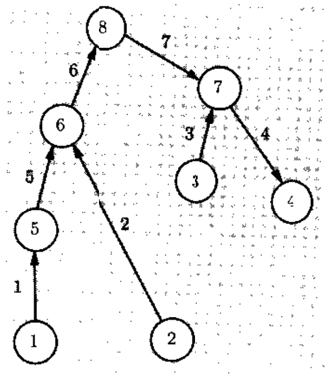
(b) Use the flow conservation equations to determine the flows on the arcs incident to the leaves, and continue by proceeding from the leaves towards the root.

It should be pretty obvious from Figure 7.10 that once a tree is fixed, a corresponding tree solution is uniquely determined. Nevertheless, we provide a rigorous proof.

Theorem 7.3 *Let $T \subset \mathcal{A}$ be a set of $n-1$ arcs that form a tree when their direction is ignored. Then, the system of linear equations $\tilde{\mathbf{A}}\mathbf{f} = \tilde{\mathbf{b}}$, and $f_{ij} = 0$ for all $(i,j) \notin T$, has a unique solution.*

Proof. Let $\mathbf{B}$ be the $(n-1) \times (n-1)$ matrix that results if we only keep those $n-1$ columns of $\tilde{\mathbf{A}}$ that correspond to the arcs in T. Let $\mathbf{f}_T$ be the subvector of $\mathbf{f}$, of dimension $n-1$, whose entries are the flow variables f_{ij}, $(i,j) \in T$. We need to show that the linear system $\mathbf{B}\mathbf{f}_T = \tilde{\mathbf{b}}$ has a unique solution. For this, it suffices to show that the matrix $\mathbf{B}$ is nonsingular.

Let us assume that the nodes have been renumbered so that numbers increase along any path from a leaf to the root node n. Let us also assign



$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & -1 & 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & -1 & 1 & 0 & 0 & -1 \end{bmatrix}$$

Figure 7.11: A numbering of the nodes and arcs of a tree, and the corresponding **B** matrix.

to every arc $(i,j) \in T$, the number $\min\{i,j\}$; see Figure 7.11. Such a renumbering of nodes and arcs amounts to a reordering of the rows and columns of **B** but does not affect whether **B** is singular or not.

With the above numbering, the ith column of **B** corresponds to the ith arc, which is an arc of the form (i,j) or (j,i), with $j > i$. Thus, any nonzero entries in the ith column will be in row i or j. Since $j > i$, no nonzero entry can be found above the diagonal. We conclude that **B** is lower triangular and has nonzero diagonal entries. This implies that **B** has nonzero determinant and is nonsingular, which completes the proof. □

We note an important corollary of the proof of the previous theorem.

Corollary 7.1 *If the graph G is connected, then the matrix $\tilde{\mathbf{A}}$ has linearly independent rows.*

Proof. If the graph G is connected, then there exists a set of arcs $T \subset \mathcal{A}$ that form a tree, when their orientation is ignored (cf Theorem 7.2). Let us pick such a set T and form the corresponding matrix **B**, as in the proof of Theorem 7.3. Since the $(n-1) \times (n-1)$ matrix **B** is nonsingular, it has linearly independent columns. Hence, the matrix $\tilde{\mathbf{A}}$ has $n-1$ linearly independent columns and, therefore, has $n-1$ linearly independent rows. □

With our construction of a tree solution, the columns of **B** are the columns of $\tilde{\mathbf{A}}$ corresponding to the variables f_{ij}, for $(i,j) \in T$, and are linearly independent. In general linear programming terminology, **B** is a basis matrix. Since the remaining variables f_{ij}, $(i,j) \notin T$, are set to zero, the resulting flow vector **f** is the basic solution corresponding to this basis. Thus, a tree solution is a basic solution, and a feasible tree solution is a basic feasible solution. In fact, the converse is also true.

Theorem 7.4 *A flow vector is a basic solution if and only if it is a tree solution.*

Proof. We have already argued that a tree solution is a basic solution. Suppose now that a flow vector **f** is not a tree solution. We will show that it is not a basic solution. Note that if $\mathbf{Af} \neq \mathbf{b}$, then **f** is not a basic solution, by definition. Thus, we only need to consider the case where $\mathbf{Af} = \mathbf{b}$.

Let $S = \{(i,j) \in \mathcal{A} \mid f_{ij} \neq 0\}$. If the arcs in the set S do not form a cycle, then there exists a set T of $n-1$ arcs such that $S \subset T$, and such that the arcs in T form a tree [cf. Assumption 7.1(b) and Theorem 7.2]. Since $f_{ij} = 0$ for all $(i,j) \notin T$, the flow vector **f** is the tree solution associated with T, which is a contradiction.

Let us now assume that the set S contains a cycle C and let $\mathbf{h}^C$ be the simple circulation associated with C. Consider the flow vector $\mathbf{f} + \mathbf{h}^C$. We have $\mathbf{Af} = \mathbf{b}$ and $\mathbf{Ah}^C = \mathbf{0}$, which implies that $\mathbf{A}(\mathbf{f} + \mathbf{h}^C) = \mathbf{b}$. Furthermore, whenever $f_{ij} = 0$ the arc (i,j) does not belong to the cycle C, and we have $h_{ij}^C = 0$. We see that all constraints that are active at the vector **f** are also active at the vector $\mathbf{f} + \mathbf{h}^C$. Thus, the constraints that are active at **f** do not have a unique solution, and **f** is not a basic solution (cf. Theorem 2.2 and Definition 2.9 in Section 2.2). See Figure 7.12 for an illustration. □

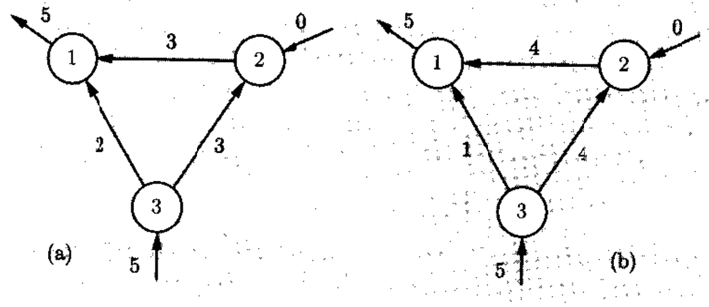


Figure 7.12: (a) Part of a flow vector that satisfies $\mathbf{Af} = \mathbf{b}$. This flow vector is not a tree solution because the arcs (2, 1), (3, 1), and (3, 2) form a cycle C and carry nonzero flow. (b) The flow vector $\mathbf{f} + \mathbf{h}^C$. Active constraints (arcs that carry zero flow) under **f** remain active under $\mathbf{f} + \mathbf{h}^C$.

To summarize our conclusions so far, we have established the following:

(a) Basic (feasible) solutions are (feasible) tree solutions and vice versa.

(b) Every basis matrix is triangular when its rows and columns are suitably reordered.

(c) Given a basis matrix $\mathbf{B}$, the vector of basic variables $\mathbf{B}^{-1}\tilde{\mathbf{b}}$ can be easily computed, without the need to maintain $\mathbf{B}^{-1}$ in a tableau.

As in the case of general linear programming problems, a basic feasible solution can be degenerate. This happens if the flow on some arc $(i,j) \in T$ turns out to be 0. In this case, the same basic feasible solution may correspond to several trees. For example, the tree shown in Figure 7.10 leads to a degenerate basic feasible solution, because $f_{67} = 0$. A different tree that would yield the same basic feasible solution is obtained by replacing arc (6,7) by arc $(5,7)$.

Change of basis

We will now develop the mechanics of a change of basis. Recall that in a general linear programming problem, we first choose a nonbasic variable that enters the basis, find how to adjust the basic variables in order to maintain the equality constraints, and increase the value of the entering variable until one of the old basic variables is about to become negative. We specialize this procedure to the network case. Picking a nonbasic variable is the same as choosing an arc (i,j) that does not belong to T. Then, the arc (i,j) together with some of the arcs in T form a cycle. Let us choose the orientation of the cycle so that (i,j) is a forward arc. Let F and B be the sets of forward and backward arcs in the cycle, respectively. If we are to increase the value of the nonbasic variable f_{ij} to some θ, the old basic variables need to be adjusted in order not to violate the flow conservation constraints. This can be accomplished by pushing θ units of flow around the cycle. More precisely, $f_{k\ell}$ is increased (decreased) by θ for all forward (backward) arcs of the cycle. The new flow variables $\hat{f}_{k\ell}$ are given by

$$\hat{f}_{k\ell} = \begin{cases} f_{k\ell} + \theta, & \text{if } (k,\ell) \in F, \\ f_{k\ell} - \theta, & \text{if } (k,\ell) \in B, \\ f_{k\ell}, & \text{otherwise}. \end{cases} \tag{7.4}$$

We set θ as large as possible, provided that all arc flows remain nonnegative. It is clear that the largest possible value of θ is equal to

$$\theta^* = \min_{(k,\ell)\in B} f_{k\ell}, \tag{7.5}$$

except if B is empty, in which case we let $\theta^* = \infty$. A variable $f_{k\ell}$ that attains the minimum in Eq. (7.5) is set to zero and exits the basis. If $f_{k\ell} = 0$ for some arc $(k,\ell) \in B$ (which can happen if we start with a degenerate basic feasible solution), then the change of basis occurs without any change of the arc flows. (For the example shown in Figure 7.10, if f_{57} enters the basis, f_{67} exits the basis and $\theta^* = 0$.)

Calculation of the cost change

The cost change resulting from the above described change of basis, is equal to

$$\theta^* \cdot \left(\sum_{(k,\ell)\in F} c_{k\ell} - \sum_{(k,\ell)\in B} c_{k\ell} \right). \tag{7.6}$$

Naturally, the variable f_{ij} should enter the basis only if the value of the expression (7.6) is negative.

From the development of the simplex method for general linear programming problems, we know that if the variable that enters the basis takes the value θ^*, then the cost changes by θ^* times the reduced cost of the entering variable. Comparing with Eq. (7.6), we see that the reduced cost $\bar{c}_{ij}$ of a nonbasic variable f_{ij} is given by

$$\bar{c}_{ij} = \sum_{(k,\ell)\in F} c_{k\ell} - \sum_{(k,\ell)\in B} c_{k\ell}, \tag{7.7}$$

which is simply the cost of the cycle around which flow is being pushed.

We will now derive an alternative formula for the reduced costs that allows for more efficient computation. Recall the general formula $\bar{\mathbf{c}}' = \mathbf{c}' - \mathbf{p}'\tilde{\mathbf{A}}$ for determining the reduced costs, where $\mathbf{p}$ is the dual vector given by $\mathbf{p}' = \mathbf{c}'_B\mathbf{B}^{-1}$, $\mathbf{B}$ is the current basis matrix, and $\mathbf{c}_B$ is the vector with the costs of the basic variables. The dimension of $\mathbf{p}$ is equal to the number of rows of $\tilde{\mathbf{A}}$, which is $n-1$, and we have one dual variable p_i associated with each node $i \neq n$. Suppose that (i,j) is the kth arc of the graph. Then, the kth entry of the vectors $\bar{\mathbf{c}}$ and $\mathbf{c}$ is equal to $\bar{c}_{ij}$ and c_{ij}, respectively. The kth entry of $\mathbf{p}'\tilde{\mathbf{A}}$ is equal to the inner product of $\mathbf{p}$ with the kth column of $\tilde{\mathbf{A}}$. From the definition of the node-arc incidence matrix, the kth column of $\tilde{\mathbf{A}}$ has an entry equal to 1 at the ith row (if $i < n$), and an entry equal to -1 at the jth row (if $j < n$). We conclude that

$$\bar{c}_{ij} = \begin{cases} c_{ij} - (p_i - p_j), & \text{if } i,j \neq n, \\ c_{ij} - p_i, & \text{if } j = n, \\ c_{ij} + p_j, & \text{if } i = n. \end{cases} \tag{7.8}$$

Equation (7.8) can be written more concisely if we define $p_n = 0$, in which case we have

$$\bar{c}_{ij} = c_{ij} - (p_i - p_j), \qquad \forall\, (i,j) \in \mathcal{A}. \tag{7.9}$$

It remains to compute the dual vector $\mathbf{p}' = \mathbf{c}'_B\mathbf{B}^{-1}$ associated with the current basis. Since the reduced cost of every basic variable must be equal to zero, Eq. (7.9) yields

$$\begin{aligned} p_i - p_j &= c_{ij}, \qquad \forall\, (i,j) \in T, \\ p_n &= 0. \end{aligned} \tag{7.10}$$

The system of equations (7.10) is easily solved using the following procedure. We view node n as the root of the tree and set $p_n = 0$. We then go down the tree, proceeding from the root towards the leaves, with a new component of $\mathbf{p}$ being evaluated at each step; see Figure 7.13.

Overview of the algorithm

We start with a summary of the network simplex algorithm and then proceed to discuss some issues related to initialization and termination.

The simplex method for uncapacitated network flow problem

1. A typical iteration starts with a basic feasible solution $\mathbf{f}$ associated with a tree T.
2. To compute the dual vector $\mathbf{p}$, solve the system of equation (7.10), by proceeding from the root towards the leaves.
3. Compute the reduced costs $\bar{c}_{ij} = c_{ij} - (p_i - p_j)$ of all arcs (i,j) T. If they are all nonnegative, the current basic feasible solution is optimal and the algorithm terminates; else, choose some (i,j) with $\bar{c}_{ij} < 0$ to be brought into the basis.
4. The entering arc (i,j) and the arcs in T form a unique cycle. If all arcs in the cycle are oriented the same way as (i,j), then the optimal cost is $-\infty$ and the algorithm terminates.
5. Let B be the set of arcs in the cycle that are oriented in the opposite direction from (i,j). Let $\theta^* = \min_{(k,\ell)\in B} f_{k\ell}$, and push θ^* units of flow around the cycle. A new flow vector is determined according to Eq. (7.4). Remove from the basis one of the old basic variables whose new value is equal to zero.

In the case where finding an initial basic feasible solution is difficult, we may need to form and solve an auxiliary problem For example, for each pair of source and sink nodes, we may introduce an auxiliary arc; finding a basic feasible solution in the presence of these arcs is straightforward. Furthermore, if the unit costs c_{ij} of the auxiliary arcs are chosen large enough solving the auxiliary problem is equivalent to solving the original problem.

The network simplex algorithm is similar to the naive implementation described in Section 3.3. Because of the special structure of the basis matrix $\mathbf{B}$, the system $\mathbf{c}'_B = \mathbf{p}'\mathbf{B}$ can be solved on the fly, without the need to maintain a simplex tableau or the inverse basis matrix $\mathbf{B}^{-1}$. For a rough

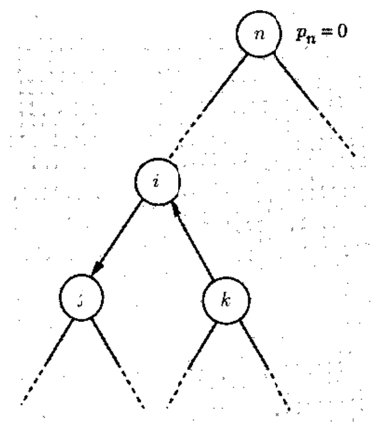


Figure 7.13: Once p_i is computed, p_j and p_k can also be computed, because we have $p_i - p_j = c_{ij}$ and $p_k - p_i = c_{ki}$. Starting from the root and continuing in this fashion, all dual variables can be computed.

count of the computational requirements of each iteration, we need $O(n)$ computations to evaluate the dual vector $\mathbf{p}$, $O(m)$ computations to evaluate all of the reduced costs, and another $O(n)$ computations to effect the change of basis. Given that $m \geq n-1$, the total is $O(m)$, which compares favorably with the $O(mn)$ computational requirements of an iteration of the simplex method for general linear programming problems. In practice, the running time of the network simplex algorithm is improved further by using a somewhat more clever way of updating the dual variables, and by using suitable data structures to organize the computation.

All of the theory in Chapters 3 and 4 applies to the network simplex method. In particular, in the absence of degeneracy, the algorithm is guaranteed to terminate after a finite number of steps. In the presence of degeneracy, the algorithm may cycle. Cycling can be avoided by using either a general purpose anticycling rule or special methods. If the optimal cost is $-\infty$, the algorithm terminates with a negative cost directed cycle. (The simple circulation $\mathbf{h}^C$ associated with that cycle is an extreme ray of the feasible set, and $\mathbf{c}'\mathbf{h}^C < 0$.) If the optimal cost is finite, the algorithm terminates with an optimal flow vector $\mathbf{f}$ and an optimal dual vector $\mathbf{p}$. In practice, the number of iterations is often $O(m)$, but there exist examples involving an exponential number of basis changes.

Example 7.3 Consider the uncapacitated network problem shown in Figure 7.14(a); the numbers next to each arc are the corresponding costs. Figure 7.14(b) shows a tree and a corresponding feasible tree solution. Arc $(4,3)$ forms a cycle consisting of nodes 4, 3, and 5. The reduced cost $\bar{c}_{43}$ of f_{43} is equal to the cost of

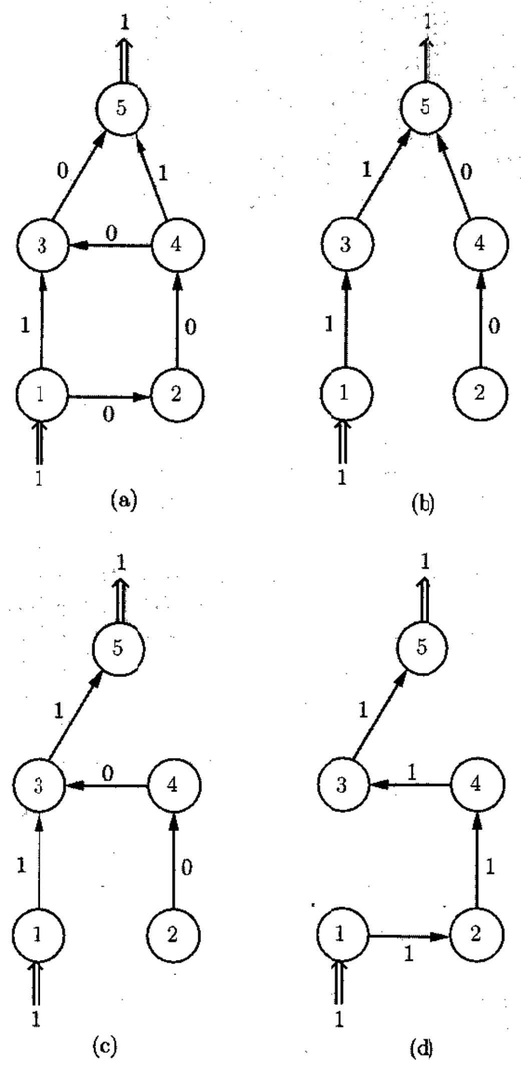


Figure 7.14: (a) An uncapacitated network flow problem. Arc costs are indicated next to each arc. (b) An initial feasible tree solution. The arc flows are indicated next to each arc. (c)-(d) Feasible tree solutions obtained after the first and the second change of basis, respectively.

that cycle, which is $c_{43}+c_{35}-c_{45}=-1$. We let arc $(4,3)$ enter the tree. Pushing flow along the cycle attempts to reduce the flow along the arc $(4,5)$. Since this was zero to start with (degeneracy), we have $\theta^*=0$; the arc $(4,5)$ leaves the tree and we obtain the feasible tree solution indicated in Figure 7.14(c). The reduced cost associated with arc $(1,2)$ is $c_{12}+c_{24}+c_{43}-c_{13}=-1$, and we let that arc enter the tree. We can push up to one unit of flow along the cycle 1,2,4,3,1, that is, until the flow along arc $(1,3)$ is set to zero. Thus, $\theta^*=1$, the arc $(1,3)$ leaves the tree, and we obtain the feasible tree solution indicated in Figure 7.14(d). It is not hard to verify that all reduced costs are nonnegative and we have an optimal solution.

Integrality of optimal solutions

An important feature of network flow problems is that when the problem data are integer, most quantities of interest are also integer and the simplex method can be implemented using integer (as opposed to floating point) arithmetic. This allows for faster computation and, equally important, the issues of finite precision and truncation error disappear. The theorem that follows provides a summary of integrality properties.

Theorem 7.5 *Consider an uncapacitated network flow problem and assume that the underlying graph is connected.*

(a) *For every basis matrix $\mathbf{B}$, the matrix $\mathbf{B}^{-1}$ has integer entries.*

(b) *If the supplies b_i are integer, then every basic solution has integer coordinates.*

(c) *If the cost coefficients c_{ij} are integer, then every dual basic solution has integer coordinates.*

Proof.

(a) As shown in the proof of Theorem 7.3, we can reorder the rows and columns of a basis matrix $\mathbf{B}$ so that it becomes lower triangular and its diagonal entries are either 1 or -1. Therefore, the determinant of $\mathbf{B}$ is equal to 1 or -1. By Cramer's rule, $\mathbf{B}^{-1}$ has integer entries.

(b) This follows by inspecting the nature of the algorithm that determines the values of the basic variables (see the proof of Theorem 7.3), or from the formula $\mathbf{f}_T=\mathbf{B}^{-1}\tilde{\mathbf{b}}$.

(c) This follows by inspecting the nature of the algorithm that determines the values of the dual variables, or from the formula $\mathbf{p}'=\mathbf{c}_B'\mathbf{B}^{-1}$. □

We now have the following important corollary of Theorem 7.5.

Corollary 7.2 *Consider an uncapacitated network flow problem, and assume that the optimal cost is finite.*

(a) *If all supplies b_i are integer, there exists an integer optimal flow vector.*

(b) *If all cost coefficients c_{ij} are integer, there exists an integer optimal solution to the dual problem.*

The simplex method for capacitated problems

We will now generalize the simplex method to the case where some of the arc capacities are finite and we have constraints of the form

$$d_{ij} \le f_{ij} \le u_{ij}, \qquad (i,j) \in \mathcal{A}.$$

There are only some minor differences from the discussion earlier in this section. For this reason, our development will be less formal.

Consider a set $T \subset \mathcal{A}$ of $n-1$ arcs that form a tree when their direction is ignored. We partition the remaining arcs into two disjoint subsets D and U. We let $f_{ij} = d_{ij}$ for every $(i,j) \in D$, $f_{ij} = u_{ij}$ for every $(i,j) \in U$, and then solve the flow conservation equations for the remaining variables f_{ij}, $(i,j) \in T$. The resulting flow vector is easily shown to be a basic solution, and all basic solutions can be obtained in this manner; the argument is similar to the proofs of Theorems 7.3 and 7.4.

Given a basic feasible solution associated with the sets T, D, and U, we evaluate the vector of reduced costs using the same formulae as before, and then examine the arcs outside T. If we find an arc $(i,j) \in D$ whose reduced cost is negative, we push as much flow as possible around the cycle created by that arc. (This is the same as in our previous development.) Alternatively, if we can find an arc $(i,j) \in U$ with positive reduced cost, we push as much flow as possible around the cycle created by that arc, but in the opposite direction. In either case, we are dealing with a direction of cost decrease. Determining how much flow can be pushed is done as follows. Let F be the set of arcs whose flow is to increase due to the contemplated flow push; let B be the set of arcs whose flow is to decrease. Then, the flow increment is limited by θ^*, defined as follows:

$$\theta^* = \min\left\{ \min_{(k,\ell)\in B}\{f_{k\ell} - d_{k\ell}\},\ \min_{(k,\ell)\in F}\{u_{k\ell} - f_{k\ell}\} \right\}. \qquad (7.11)$$

By pushing θ^* units of flow around the cycle, there will be at least one arc (k,ℓ) whose flow is set to either $d_{k\ell}$ or $u_{k\ell}$. If the arc (k,ℓ) belongs to T, it is removed from the tree and is replaced by (i,j). The other possibility is that $(k,\ell) = (i,j)$. (For example, pushing flow around the cycle may result in f_{ij} being reduced from u_{ij} to d_{ij}.) In that case, the set T remains the same, but (i,j) is moved from U to D, or vice versa. In any case, we obtain a new basic feasible solution. (In the presence of degeneracy, it is possible that the new basic feasible solution coincides with the old one, and only the sets T, D, or U change.) To summarize, the network simplex algorithm for capacitated problems is as follows.

The simplex method for capacitated network flow problems

1. A typical iteration starts with a basic feasible solution $\mathbf{f}$ associated with a tree T, and a partition of the remaining arcs into two sets D, U, such that $f_{ij} = d_{ij}$ for $(i,j) \in D$, and $f_{ij} = u_{ij}$ for $(i,j) \in U$.
2. Solve the system of equations (7.10) for $p_1, \ldots, p_n$, by proceeding from the root towards the leaves.
3. Compute the reduced costs $\bar{c}_{ij} = c_{ij} - (p_i - p_j)$ of all arcs $(i,j) \notin T$. If $\bar{c}_{ij} \ge 0$ for all $(i,j) \in D$, and $\bar{c}_{ij} \le 0$ for all $(i,j) \in U$, the current basic feasible solution is optimal and the algorithm terminates.
4. Let (i,j) be an arc such that $\bar{c}_{ij} < 0$ and $(i,j) \in D$, or such that $\bar{c}_{ij} > 0$ and $(i,j) \in U$. This arc (i,j) together with the tree T forms a unique cycle. Choose the orientation of the cycle as follows. If $(i,j) \in D$, then (i,j) should be a forward arc. If $(i,j) \in U$, then (i,j) should be a backward arc.
5. Let F and B be the forward and backward arcs, respectively, in the cycle. Determine θ^* according to Eq. (7.11). Compute a new flow vector, with components $\hat{f}_{k\ell}$, by letting

$$\hat{f}_{k\ell} = \begin{cases} f_{k\ell} + \theta^*, & \text{if } (k,\ell) \in F, \\ f_{k\ell} - \theta^*, & \text{if } (k,\ell) \in B, \\ f_{k\ell}, & \text{otherwise.} \end{cases}$$

Finally, update the sets T, D, U.

7.4 The negative cost cycle algorithm

The network simplex algorithm incorporates a basic idea, which is present in practically every primal method for network flow problems: given a current primal feasible solution, find an improved one by identifying a negative cost cycle along which flow can be pushed. One advantage of the simplex method is that it searches for negative cost cycles using a streamlined and efficient mechanism. A potential disadvantage is that a change of basis can be degenerate, with no flow being pushed, and without any cost improvement.

In this section, we present a related, but different, algorithm, where every iteration aims at a nonzero cost improvement. In particular, at every

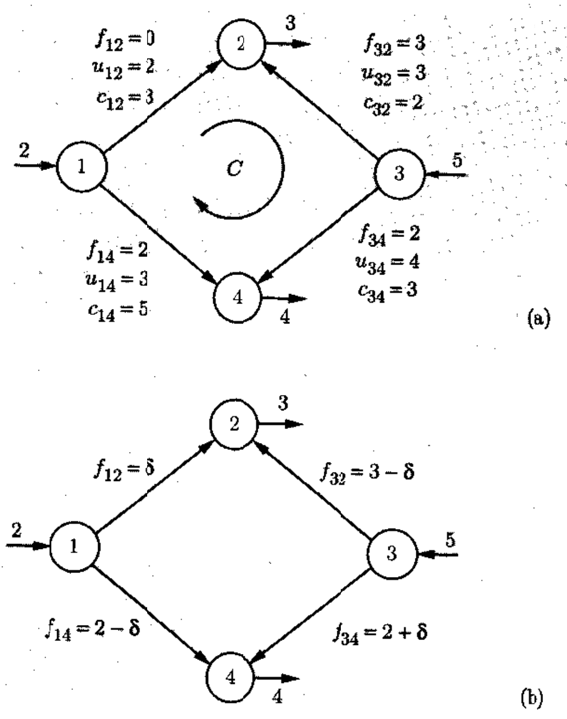


Figure 7.15: (a) A portion of a network, together with the values of some of the flow variables. (b) The new arc flows after pushing δ units of flow around the cycle C.

iteration we push some flow around a negative cost cycle. The algorithm terminates when no profitable cycle can be identified. The method is justified by a key result that relates the absence of profitable cycles with optimality.

Motivation

Consider the portion of a network shown in Figure 7.15(a). Could the flow vector **f** given in the figure be optimal? The answer is no, for the following reason. Suppose that we push δ units of flow along the indicated cycle, where δ is a positive scalar. Taking into account the direction of the arcs, the new flow variables take the values indicated in Figure 7.15(b). In particular, the flow on every forward arc is increased by δ and the flow on every backward arc is reduced by δ. Flow conservation is preserved, and as long as $\delta \leq 2$, the constraints $0 \leq f_{ij} \leq u_{ij}$ are respected, and the new

flow is feasible. The change in costs is

$$c_{12}\delta + c_{32}(-\delta) + c_{34}\delta + c_{14}(-\delta) = -\delta,$$

which is negative, and **f** cannot be optimal. As this example illustrates, a flow **f** can be improved if we can identify a cycle along which flow can be profitably pushed.

Description of the algorithm

In this subsection, we present the algorithm of interest after developing some of its elements. We assume that we have a network described by a directed graph $G = (\mathcal{N}, \mathcal{A})$, supplies b_i, arc capacities u_{ij}, and cost coefficients c_{ij}. Let C be a cycle, and let F and B be the sets of forward and backward arcs of the cycle, respectively. Let $\mathbf{h}^C$ be the simple circulation associated with this cycle, that is,

$$h_{ij}^C = \begin{cases} 1, & \text{if } (i,j) \in F, \\ -1, & \text{if } (i,j) \in B, \\ 0, & \text{otherwise.} \end{cases}$$

Let **f** be a feasible flow vector and let δ be a nonnegative scalar. If we change **f** to $\mathbf{f} + \delta\mathbf{h}^C$, we say that we are *pushing δ units of flow along the cycle C*. Since **f** is feasible, we have $\mathbf{Af} = \mathbf{b}$; since $\mathbf{Ah}^C = \mathbf{0}$, we obtain $\mathbf{A}(\mathbf{f} + \delta\mathbf{h}^C) = \mathbf{b}$, and the flow conservation constraint is still satisfied. In order to maintain feasibility, we also need

$$0 \leq f_{ij} + \delta h_{ij}^C \leq u_{ij},$$

that is,

$$\begin{aligned} 0 &\leq f_{ij} + \delta \leq u_{ij}, && \text{if } (i,j) \in F, \\ 0 &\leq f_{ij} - \delta \leq u_{ij}, && \text{if } (i,j) \in B. \end{aligned}$$

Since $\delta \geq 0$ and $0 \leq f_{ij} \leq u_{ij}$, this is equivalent to

$$\begin{aligned} \delta &\leq u_{ij} - f_{ij}, && \text{if } (i,j) \in F, \\ \delta &\leq f_{ij}, && \text{if } (i,j) \in B. \end{aligned}$$

Thus, the maximum amount of flow that can be pushed along the cycle, which we denote by $\delta(C)$, is given by

$$\delta(C) = \min\left\{ \min_{(i,j)\in F} (u_{ij} - f_{ij}),\ \min_{(i,j)\in B} f_{ij} \right\}. \qquad (7.12)$$

If the set B is empty and if $u_{ij} = \infty$ for every arc in the cycle, then there are no restrictions on δ, and we set $\delta(C) = \infty$. If $f_{ij} < u_{ij}$ for all forward arcs and $f_{ij} > 0$ for all backward arcs, then $\delta(C) > 0$, and we say that the cycle is *unsaturated*. For the cycle considered in Figure 7.15(a), we have $\delta(C) = 2$.

We now calculate the cost change when we push a unit of flow along a cycle C. Using the definition of $\mathbf{h}^C$, the cost change is

$$\mathbf{c}'\mathbf{h}^C = \sum_{(i,j)\in F} c_{ij} - \sum_{(i,j)\in B} c_{ij},$$

the *cost of cycle C*.

We can now propose an algorithm which at each iteration looks for a negative cost unsaturated cycle and pushes as much flow as possible along that cycle.

Negative cost cycle algorithm

```
1. Start with a feasible flow $\mathbf{f}$.
2. Search for an unsaturated cycle with negative cost.
3. If no unsaturated cycle with negative cost can be found, the
   algorithm terminates.
4. If a negative cost unsaturated cycle $C$ is found, then:
   (a) If $\delta(C) < \infty$, construct the new feasible flow $\mathbf{f} + \delta(C)\mathbf{h}^C$,
       and go to Step 2.
   (b) If $\delta(C) = \infty$, the algorithm terminates and the optimal
       cost is $-\infty$.
```

There are a few different issues that need to be discussed:

(a) How do we start the algorithm?

(b) How do we search for an unsaturated cycle with negative cost?

(c) If the algorithm terminates, does it provide us with an optimal solution?

(d) Is the algorithm guaranteed to terminate?

These issues are addressed, one at a time, in the subsections that follow.

Starting the algorithm

As discussed in Section 7.2, every network flow problem can be converted into an equivalent problem with no sources or sinks. For the latter problem, the zero flow is a feasible solution that provides a starting point. As an alternative, a feasible flow (if one exists) can be constructed by solving a suitable maximum flow problem (Exercise 7.21).

The residual network

Suppose that we have a network $G = (\mathcal{N}, \mathcal{A})$ and a feasible flow $\mathbf{f}$. The *residual network* is an auxiliary network $\tilde{G} = (\mathcal{N}, \tilde{\mathcal{A}})$ with the same set of nodes, but with different arcs and arc capacities. It is a convenient device to keep track of the amount of flow that can be pushed along the arcs of the original network.

Consider an arc (i,j) with capacity u_{ij}, and let f_{ij} be the current flow through that arc. Then, f_{ij} can be increased by up to $u_{ij} - f_{ij}$, or can be decreased by up to f_{ij}. We represent these options in the residual network by introducing an arc (i,j) with capacity $u_{ij} - f_{ij}$, and an arc (j,i), with capacity f_{ij}. Any flow on the arc (j,i) in the residual network is to be interpreted as a corresponding reduction of the flow on the arc (i,j) of the original network.

We assign costs to the arcs of the residual network in a way that reflects the cost changes in the original network. In particular, we associate a cost of c_{ij} with the arc (i,j) of the residual network, and a cost of $-c_{ij}$ with the arc (j,i) of the residual network. [This is because a unit of flow on the arc (j,i) corresponds to a unit reduction of the flow on the arc (i,j) of the original network, and a cost change of $-c_{ij}$.] All supplies in the residual network are set to zero, which implies that every feasible flow is a circulation. Finally, we delete those arcs of the residual network that have zero capacity.

The construction of the residual network is shown in Figure 7.16. As seen in the figure, the residual network may contain two arcs with the same start node and the same end node. In particular, the presence of two arcs from i to j indicates that we can push flow from i to j either by increasing the value of f_{ij} or by decreasing the value of f_{ji}. Strictly speaking, this violates our original definition of a graph, but this turns out not to be a problem.

Let $\mathbf{f}$ be a feasible flow in the original network and let $\mathbf{f} + \overline{\mathbf{f}}$ be another feasible flow in the original network. The flow increment $\overline{\mathbf{f}}$ can be associated with a flow vector $\tilde{\mathbf{f}}$ in the residual network as follows.

(a) If $\overline{f}_{ij} > 0$, we let the flow $\tilde{f}_{ij}$ on the corresponding arc (i,j) in the residual network be equal to $\overline{f}_{ij}$. Feasibility in the original network implies that $\overline{f}_{ij} \le u_{ij} - f_{ij}$, and $\tilde{f}_{ij}$ satisfies the capacity constraint in the residual network.

(b) If $\overline{f}_{ij} < 0$, we let the flow $\tilde{f}_{ji}$ on the corresponding arc (j,i) in the residual network be equal to $-\overline{f}_{ij}$. Feasibility in the original network implies that $-\overline{f}_{ij} \le f_{ij}$ and therefore $\tilde{f}_{ji}$ satisfies the capacity constraint in the residual network.

All variables $\tilde{f}_{ij}$ that are not set by either (a) or (b) above are left at zero value. See Figure 7.17 for an illustration.

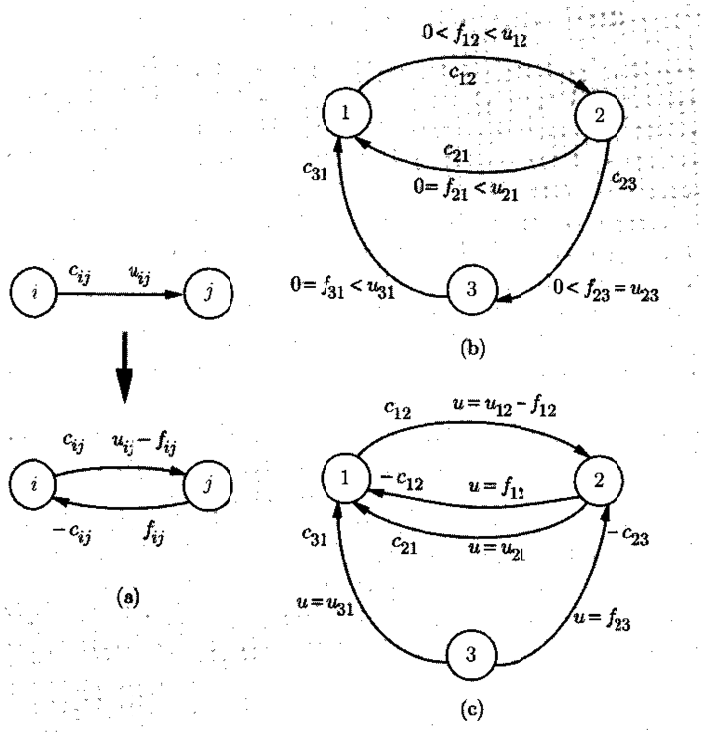


Figure 7.16: (a) Each arc of the original network leads to two arcs in the residual network. (b) A network and an associated feasible flow. (c) The corresponding residual network. Note that zero capacity arcs have been deleted.

We make the following observations:

(a) We have $\tilde{f}_{ij} \geq 0$ for all arcs in the residual network.

(b) The flow $\tilde{\mathbf{f}}$ in the residual network is a circulation. This is because in the original network, we have $\mathbf{Af} = \mathbf{b} = \mathbf{A}(\mathbf{f} + \overline{\mathbf{f}})$. Hence, $\mathbf{A}\overline{\mathbf{f}} = \mathbf{0}$, which means that with the flow vector $\overline{\mathbf{f}}$, the net flow into any node i is zero. Because of the way $\tilde{\mathbf{f}}$ was constructed, the net flow into any node of the residual network must also be zero.

(c) The cost of $\tilde{\mathbf{f}}$ in the residual network is equal to $\sum_{(i,j)} c_{ij}\overline{f}_{ij}$, which is the cost of $\overline{\mathbf{f}}$ in the original network. This is because for each arc with $\overline{f}_{ij} > 0$, we have an equal flow $\tilde{f}_{ij}$ in a corresponding arc (i,j) in the residual network, and the latter arc has unit cost c_{ij}. Furthermore, for each arc with $\overline{f}_{ij} < 0$ in the original network, we have a flow $\tilde{f}_{ji} = -\overline{f}_{ij}$ in a corresponding arc (j,i) in the residual network, and the latter arc has unit cost $-c_{ij}$. Since $(-c_{ij})\tilde{f}_{ji} = c_{ij}\overline{f}_{ij}$, we see that $\overline{f}_{ij}$ and $\tilde{f}_{ji}$ incur the same cost.

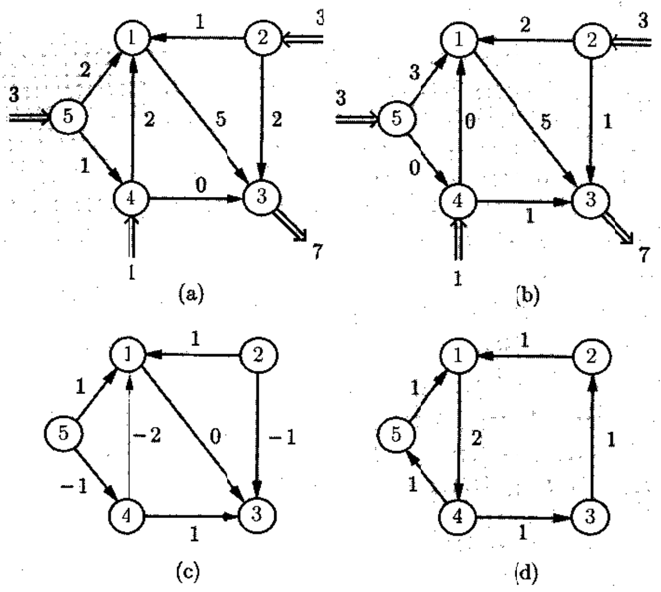


Figure 7.17: In this figure, the numbers next to each arc indicate arc flows. (a) A feasible flow $\mathbf{f}$ in a network. (b) Another feasible flow $\mathbf{f} + \overline{\mathbf{f}}$. (c) The flow increment $\overline{\mathbf{f}}$. Note that it is a circulation. (d) The flow $\tilde{\mathbf{f}}$ in the residual network (only arcs with nonzero flows are shown).

The preceding arguments can be reversed. That is, if we start with a feasible circulation $\tilde{\mathbf{f}}$ in the residual network, we can construct a circulation $\overline{\mathbf{f}}$ in the original network such that $\mathbf{f}+\overline{\mathbf{f}}$ is feasible and such that $\mathbf{c}'\overline{\mathbf{f}}$ is equal to the cost of $\tilde{\mathbf{f}}$ in the residual network.

We finally note that every unsaturated cycle in the original network corresponds to a directed cycle in the residual network in which all arcs have positive capacity and vice versa. Furthermore, the costs of these cycles in their respective networks are equal. We conclude that the search for negative cost unsaturated cycles in the original network can be accomplished by searching for a negative cost directed cycle in the residual network. In Section 7.9, we show that the problem of finding negative cost directed cycles in a graph can be solved in time $O(n^3)$; hence, the computational requirements of each iteration of the negative cost cycle algorithm are also $O(n^3)$.

Optimality conditions

We now investigate what happens at termination of the negative cost cycle algorithm If the algorithm terminates because it discovered a negative cost cycle with $\delta(C) = \infty$, then the optimal cost is $-\infty$. In particular, the flow $\mathbf{f}+\delta\mathbf{h}^C$ is feasible for every $\delta > 0$, and by letting δ become arbitrarily large, the cost of such feasible solutions is unbounded below.

The algorithm may also terminate because no unsaturated negative cost cycle can be found. In that case, we have an optimal solution, as shown by the next result.

Theorem 7.6 *A feasible flow* $\mathbf{f}$ *is optimal if and only if there is no unsaturated cycle with negative cost.*

Proof. One direction is easy. If C is an unsaturated cycle with negative cost, then $\mathbf{f} + \delta(C)\mathbf{h}^C$ is a feasible flow whose cost is less than the cost of $\mathbf{f}$, and so $\mathbf{f}$ is not optimal.

For the converse, we argue by contradiction. Suppose that $\mathbf{f}$ is a feasible flow that is not optimal. Then, there exists another feasible flow $\mathbf{f} + \overline{\mathbf{f}}$ whose cost is less, and in particular, $\mathbf{c}'\overline{\mathbf{f}} < 0$. As discussed in the preceding subsection, it follows that there exists a feasible (in particular, nonnegative) circulation $\overline{\mathbf{f}}$ in the residual network whose cost is negative. To prove that this circulation implies the existence of a negative cost directed cycle in the residual network, we need the following important result.

Lemma 7.1 (Flow decomposition theorem) *Let* $\mathbf{f} \geq \mathbf{0}$ *be a nonzero circulation. Then, there exist simple circulations* $\mathbf{f}^1, \ldots, \mathbf{f}^k$, *involving only forward arcs, and positive scalars* $a_1, \ldots, a_k$, *such that*

$$\mathbf{f} = \sum_{i=1}^{k} a_i \mathbf{f}^i.$$

Furthermore, if $\mathbf{f}$ *is an integer vector, then each* a_i *can be chosen to be an integer.*

Proof. (See Figure 7.18 for an illustration.) If $\mathbf{f}$ is the zero vector, the result is trivially true, with $k = 0$. Suppose that $\mathbf{f}$ is nonzero. Then, there exists some arc (i, j) for which $f_{ij} > 0$. Let us traverse arc (i, j). Because of flow conservation at node j, there exists some arc (j, k) for which $f_{jk} > 0$. We then traverse arc (j, k) and keep repeating the same process. Since there are finitely many nodes, some node will be eventually visited for a second time. At that point, we have found a directed cycle with each arc in the cycle carrying a positive amount of flow. Let $\mathbf{f}^1$ be the simple circulation

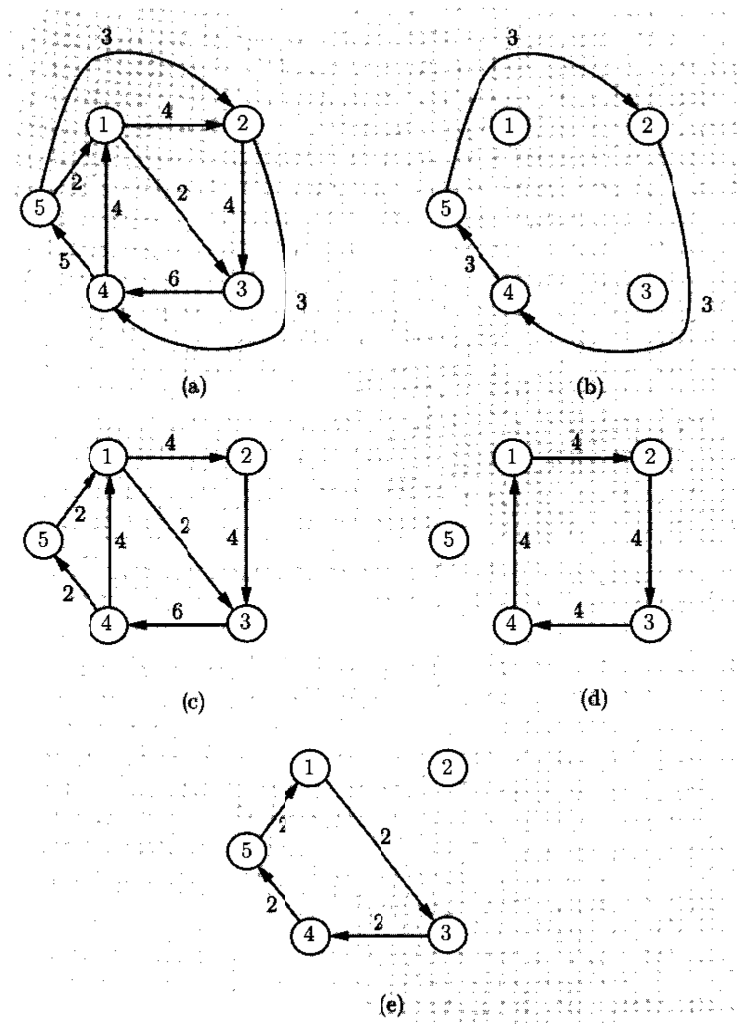


Figure 7.18: Illustration of the flow decomposition theorem. The numbers next to each arc indicate the value of the corresponding arc flows. Arcs with zero flow are not shown. (a) A nonnegative circulation $\mathbf{f}$. (b) The circulation $a_1\mathbf{f}^1$. (c) The remaining flow $\mathbf{f} - a_1\mathbf{f}^1$. (d) The circulation $a_2\mathbf{f}^2$. (e) The remaining flow $\mathbf{f} - a_1\mathbf{f}^1 - a_2\mathbf{f}^2$ is a simple circulation and we let $a_3\mathbf{f}^3$ be equal to it.

corresponding to that cycle. Let a_1 be the minimum value of f_{ij}, where the minimum is taken over all arcs in the cycle, and consider the vector $\hat{\mathbf{f}} = \mathbf{f} - a_1\mathbf{f}^1$. This vector is nonnegative because of the way that a_1 was chosen. In addition, we have $\mathbf{Af} = \mathbf{0}$ and $\mathbf{Af}^1 = \mathbf{0}$, which implies that $\mathbf{A}\hat{\mathbf{f}} = \mathbf{0}$ and $\hat{\mathbf{f}}$ is a circulation. By the definition of a_1, there exists some arc (k, ℓ) on the cycle for which $f_{k\ell} = a_1$ and $\hat{f}_{k\ell} = 0$. Therefore, the number of positive components of $\hat{\mathbf{f}}$ is smaller than the number of positive components of $\mathbf{f}$. We can now apply the same procedure to $\hat{\mathbf{f}}$, to obtain a new simple circulation $\mathbf{f}^2$, and continue similarly. Each time, the number of arcs that carry positive flow is reduced by at least one. Thus, after repeating this procedure a finite number of times, we end up with the zero flow. When this happens, we have succeeded in decomposing $\mathbf{f}$ as a nonnegative linear combination of simple circulations. Furthermore, since all of the cycles constructed were directed, these simple circulations involve only forward arcs.

If $\mathbf{f}$ is integer, then a_1 is integer, and $\hat{\mathbf{f}}$ is also an integer vector. It follows, by induction, that if we start with an integer flow vector $\mathbf{f}$, all flows produced in the course of the above procedure are integer, and all coefficients a_i are also integer. This concludes the proof of Lemma 7.1. □

We now apply Lemma 7.1 to the residual network. The circulation $\tilde{\mathbf{f}}$ can be decomposed in the form

$$\tilde{\mathbf{f}} = \sum_i a_i \tilde{\mathbf{f}}^i,$$

where each $\tilde{\mathbf{f}}^i$ is a simple circulation involving only forward arcs, and each a_i is positive. Since $\tilde{\mathbf{f}}$ has negative cost, at least one of the circulations $\tilde{\mathbf{f}}^i$ must also have negative cost; hence, the residual network has a negative cost directed cycle. As discussed in the preceding subsection, this implies that the original network contains a negative cost unsaturated cycle, and the proof of Theorem 7.6 is now complete. □

Termination of the algorithm

Before concluding that the algorithm is correct, we need a guarantee that it will eventually terminate. This is the subject of our next theorem.

Theorem 7.7 *Suppose that all arc capacities u_{ij} are integer or infinite, and that the negative cost cycle algorithm is initialized with an integer feasible flow. Then, the arc flow variables remain integer throughout the algorithm and, if the optimal cost is finite, the algorithm terminates with an integer optimal solution.*

Proof. If the current flow $\mathbf{f}$ is integer, then $\delta(C)$ is integer or infinite, for every cycle C. Hence, the flow obtained after one iteration of the algorithm must also be integer, and integrality is preserved.

At each iteration, before the algorithm terminates, we have a cost reduction of $\delta(C)|\mathbf{c}'\mathbf{h}^C|$, where C is the negative cost cycle along which flow is pushed. Since $\delta(C) \geq 1$, this is no smaller than $v = \min_D |\mathbf{c}'\mathbf{h}^D|$, where the minimum is taken over all negative cost cycles D. Thus, each iteration of the algorithm reduces the cost by at least v, which is positive. It follows that if the optimal cost is finite, the algorithm must terminate after a finite number of iterations. □

Note that Theorem 7.7 establishes an integrality property of optimal solutions. This is the same conclusion that was reached in Corollary 7.2(a), for standard form problems.

Surprisingly, and unlike the simplex method, if the arc capacities are not integer, the algorithm is not guaranteed to terminate, even if the optimal cost is finite. One possibility is that the algorithm makes an infinite number of steps, each step results in lower costs, but the cost reductions become smaller and smaller, and the cost of the current flow does not converge to the optimal cost. It turns out that finite termination can be guaranteed under special rules for choosing between negative cost cycles. Two possible rules that are known to lead to finite termination are the following:

(a) **Largest improvement rule:** Choose a negative cost cycle for which the cost improvement $\delta(C)\,|\mathbf{c}'\mathbf{h}^C|$ is largest. Unfortunately, finding such a cycle is difficult. See Exercise 7.16 for an upper bound on the number of iterations.

(b) **Mean cost rule:** Choose a negative cost cycle for which $|\mathbf{c}'\mathbf{h}^C|/k(C)$ is largest, where $k(C)$ is the number of arcs in cycle C. It turns out that the search for such a cycle is not too difficult (Exercise 7.37).

When the optimal cost is $-\infty$, the algorithm may fail to terminate after a finite number of iterations, even if the arc capacities are integer. For this reason, one should verify that the optimal cost is finite before starting the algorithm; a simple criterion is developed in Exercise 7.17.

7.5 The maximum flow problem

In the maximum flow problem, we are given a directed graph $G = (\mathcal{N}, \mathcal{A})$ and an arc capacity bound $u_{ij} \in [0, \infty]$ for each arc $(i, j) \in \mathcal{A}$. Let s and t be two special nodes, called the source and sink node, respectively. The problem is to find the largest possible amount of flow that can be sent through the network, from s to t. We will see shortly that this is a special case of the general network flow problem. On the other hand, special purpose algorithms are possible, because of the simple structure of the problem. The maximum flow problem arises in a variety of applications. Some are rather obvious (e.g., maximizing throughput in a logistics network), while others are less expected; see the example that follows.

Example 7.4 (Preemptive scheduling) We are given m identical machines and n jobs. Each job j must be processed for a total of p_j periods. (We assume that each p_j is an integer.) However, we allow *preemption*. That is, the processing of a job can be broken down and can be carried out by different machines in different periods. Each machine can only process one job at a time, and a job can only be processed by a single machine at a time. In addition, each job j is associated with a release time r_j and a deadline d_j processing cannot start before period r_j, and must be completed before period d_j. Naturally, we assume that $r_j+p_j \leq d_j$ for all jobs j. We wish to determine a schedule whereby all jobs are processed, without violating the release times and deadlines, or show that no such schedule exists.

We will now construct a maximum flow formulation of the problem. The first step is to rank all the release times and deadlines in ascending order. The resulting ordered list of numbers divides the time horizon into a number of nonoverlapping intervals. Let T_{kl} be the interval that starts in the beginning of period k and ends in the beginning of period l. Note that during each interval T_{kl}, the set of jobs that can be processed does not change. In particular, we can process any job j that has been released ($r_j \leq k$) and its deadline has not yet been reached ($l \leq d_j$). For a concrete example, suppose that we have four jobs with release times 3, 1, 3, 5, and deadlines 5, 4, 7, 9. The ascending list of release times and deadlines is $1, 3, 4, 5, 7, 9$. We then obtain five intervals, namely, T_{13}, T_{34}, T_{45}, T_{57}, and T_{79}.

We construct a network involving a source node s, a sink node t, a node corresponding to each job j, and a node corresponding to each interval T_{kl}. The arcs and their capacities are as follows. For every job j, we have an arc (s, j), with capacity p_j. We interpret the flow along this arc as the number of periods of processing that job j receives. For every node T_{kl}, we introduce an arc (T_{kl}, t), with capacity $m(l-k)$. The flow along this arc represents the total number of machine-periods of processing during the interval T_{kl}. Finally, if a job j is available for processing during the interval T_{kl}, that is, if $r_j \leq k \leq l \leq d_j$, we introduce an arc (j, T_{kl}), with capacity $l-k$. The flow along this arc represents the number of periods that job j is processed during this interval. See Figure 7.19 for an illustration. It is not hard to show that every feasible schedule corresponds to a flow through this network, with value $\sum_{j=1}^{n} p_j$, and conversely. Therefore, the scheduling problem can be solved by solving a maximum flow problem, and checking whether the resulting maximum flow value is equal to $\sum_{j=1}^{n} p_j$.

Mathematically, the maximum flow problem can be formulated as follows

$$\begin{array}{lrcll} \text{maximize} & b_s & & & \\ \text{subject to} & \mathbf{Af} & = & \mathbf{b} & \\ & b_t & = & -b_s & \\ & b_i & = & 0, & \forall\, i \neq s, t, \\ & 0 & \leq & f_{ij} \leq u_{ij}, & \forall\, (i,j) \in \mathcal{A}. \end{array}$$

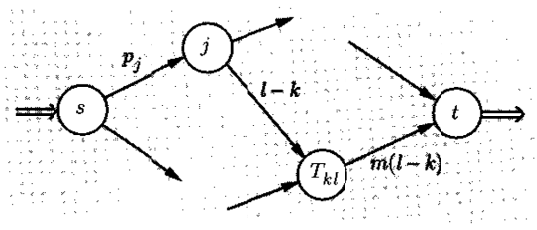


Figure 7.19: The structure of the network associated with the preemptive scheduling problem. The number next to each arc indicates its capacity. The arc from node j to node (k, l) is present only if $r_j \leq k \leq l \leq d_j$.

Note that, in contrast to the network flow problems considered earlier, b_s is a variable to be optimized. Any flow vector $\mathbf{f}$ satisfying the above constraints is called a feasible flow and the corresponding value of b_s is called the *value* of that flow.

The maximum flow problem can be reformulated as a network flow problem, as follows (see Figure 7.20 for an illustration). We let the cost of every arc be equal to zero and we introduce a new infinite capacity arc (t, s), with cost $c_{ts} = -1$. Minimizing $\sum_{(i,j)} c_{ij} f_{ij}$ in the new network is the same as maximizing the flow f_{ts} on the new arc. Since the flow on the arc (t, s) must return from s to t through the original network, maximizing f_{ts} is the same as solving the original maximum flow problem.

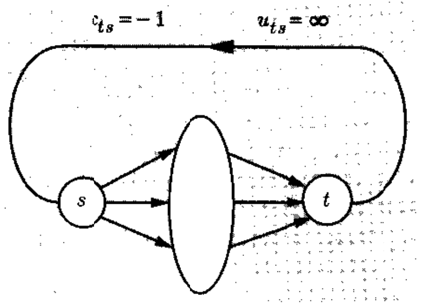


Figure 7.20: Reformulation of the maximum flow problem as a network flow problem

Once the maximum flow problem is formulated as a network flow problem, the negative cost cycle algorithm of Section 7.4 can be applied, and this is one way of deriving the main algorithm in this section (Exercise 7.18). However, our derivation will be self-contained.

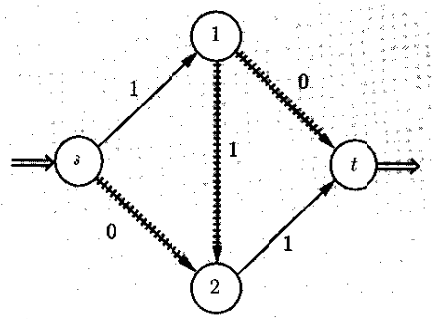


Figure 7.21: Let all arc capacities be equal to 1. The numbers next to each arc indicate the values of the arc flows. Note that up to one unit of additional flow can be pushed along the path indicated by thatched arcs.

Consider the flow illustrated in Figure 7.21. Its value can be increased by pushing additional flow along the path consisting of the arcs $(s,2)$, $(1,2)$, $(1,t)$. Note that arc $(1,2)$ is a backward arc of that path; pushing δ units of flow along the path, reduces the flow along arc $(1,2)$ by δ. The definition that follows deals with paths of this type, through which additional flow can be pushed.

Definition 7.2 *Let* **f** *be a feasible flow vector. An* **augmenting path** *is a path from s to t such that $f_{ij} < u_{ij}$ for all forward arcs, and $f_{ij} > 0$ for all backward arcs on the path.*

Suppose that we have a feasible flow and that we have found an augmenting path P. We can then increase the flow along every forward arc, decrease the flow along every backward arc by the same amount, and still satisfy all of the problem constraints; we then say that we are *pushing flow along the path P*, or that we have a *flow augmentation*. The amount of flow pushed along P can be no more than $\delta(P)$, defined by

$$\delta(P) = \min\left\{ \min_{(i,j)\in F} (u_{ij} - f_{ij}),\ \min_{(i,j)\in B} f_{ij} \right\}, \tag{7.13}$$

where F and B are the sets of forward and backward arcs, respectively, in the augmenting path. If the augmenting path consists exclusively of forward arcs, and if all arcs on the path have infinite capacity, then there is no limit on the amount of flow that can be pushed, and we have $\delta(P) = \infty$. For the example in Figure 7.21, we have $\delta(P) = 1$.

We now introduce a natural algorithm for the maximum flow problem.

The Ford-Fulkerson algorithm

1. Start with a feasible flow **f**.
2. Search for an augmenting path.
3. If no augmenting path can be found, the algorithm terminates.
4. If an augmenting path P is found, then:
   (a) If $\delta(P) < \infty$, push $\delta(P)$ units of flow along P, and go to Step 2.
   (b) If $\delta(P) = \infty$, the algorithm terminates.

If the algorithm terminates because $\delta(P) = \infty$, we have found an augmenting path without capacity limitations and, using that path, an arbitrarily large amount of flow can be sent to the sink.

We now address the termination properties of the algorithm.

Theorem 7.8 *Suppose that all arc capacities u_{ij} are integer or infinite, and that the Ford-Fulkerson algorithm is initialized with an integer flow vector. Then, the arc flow variables remain integer throughout the algorithm and, if the optimal value is finite, the algorithm terminates after a finite number of steps.*

Proof. This result can be derived as a corollary of Theorem 7.7 in Section 7.4. For a self-contained proof, note that if we have an integer feasible flow, and if all arc capacities are integer or infinite, then $\delta(P)$ is integer or infinite. Thus, integrality of flows is maintained throughout the algorithm. Every iteration of the algorithm increases the value of the flow by at least 1 [since $\delta(P)$ is integer]. Hence, either the value of the flow increases to infinity, or the algorithm must terminate. □

Example 7.5 Consider the network shown in Figure 7.22(a) and let us start with the zero flow. The path consisting of the thatched arcs in Figure 7.22(b) is an augmenting path, with $\delta(P) = 1$. After a flow augmentation, we obtain the flow indicated. The path consisting of the thatched arcs in Figure 7.22(c) is an augmenting path, with $\delta(P) = 1$. By continuing similarly, and after a total of four flow augmentations, we obtain the flow shown in Figure 7.22(e), whose value is equal to 6. At this point, no augmenting path can be found. In fact, this flow must be optimal because the total capacity of the arcs leaving node s is equal to 6, and this is a bottleneck that cannot be overcome.

If the arc capacities are rational numbers, the algorithm is again guaranteed to terminate after a finite number of iterations. This is because we can multiply all arc capacities by their least common denominator, and

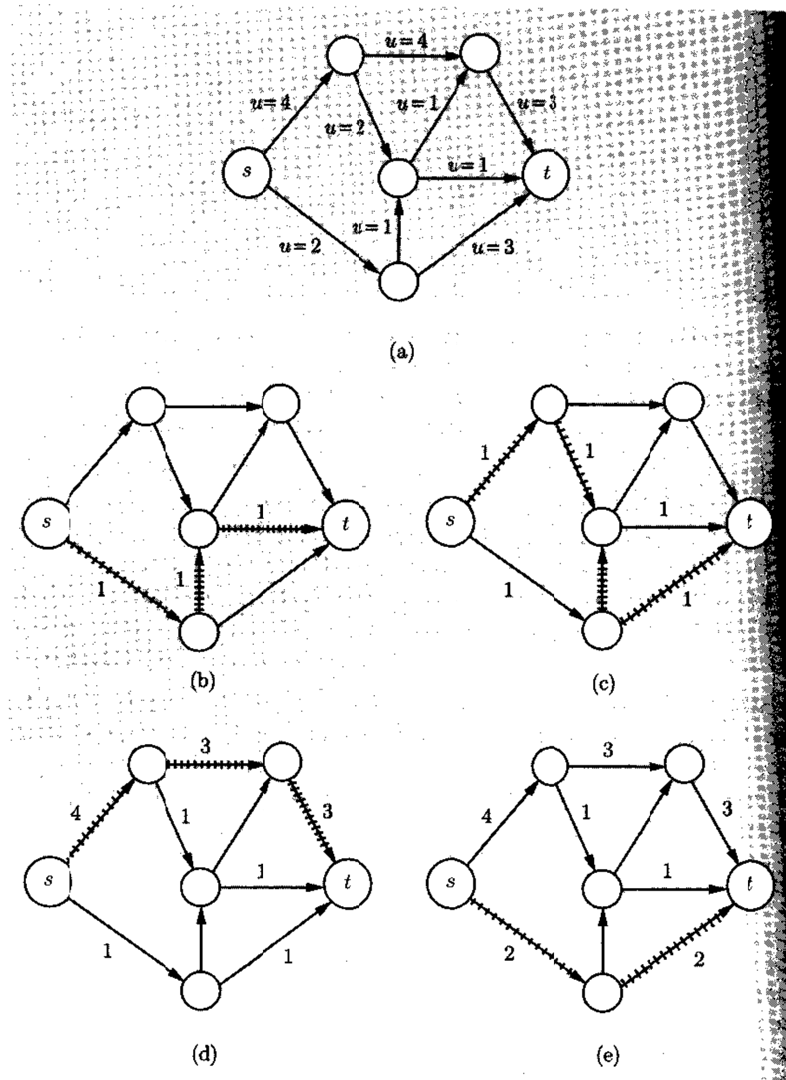


Figure 7.22: Illustration of the Ford-Fulkerson algorithm. The numbers next to the arcs in part (a) are arc capacities. We start with the zero flow. (b)-(e) In each case, we identify the augmenting path indicated in the figure, and push as much flow as possible. The numbers next to the arcs correspond to the arc flows after the flow augmentation. The flow indicated in part (e) is optimal.

obtain an equivalent problem with integer arc capacities. However, if the arc capacities are not rational, there exist examples for which the algorithm never terminates. In particular, even though the value of the flow is monotonically increasing, its limit can be strictly less than the optimal.

For the non-rational case, the Ford-Fulkerson algorithm can be made to terminate after a finite number of iterations, if one uses special methods for choosing an augmenting path. For example, if one looks for an augmenting path with the least possible number of arcs, then the algorithm can be shown to terminate after $O(|\mathcal{A}| \cdot |\mathcal{N}|)$ iterations.

If the algorithm does terminate, it provides us with an optimal solution. This fact can be obtained as a corollary of the optimality conditions in Section 7.4. A self-contained proof using different ideas will be provided shortly. However, we will first discuss some issues related to the search for an augmenting path.

Searching for an augmenting path

The search for an augmenting path can be carried out in a fairly simple manner, using a method known as the *labeling algorithm.*

Suppose that we have a feasible flow **f**. Consider a path from the source s to some node k, such that $f_{ij} < u_{ij}$ for all forward arcs on the path, and $f_{ij} > 0$ for all backward arcs on the path; we say that this is an *unsaturated* path from s to k. Such a path can be used to push additional flow from node s to node k, without violating the capacity constraints. Note that an unsaturated path from s to t is the same as an augmenting path.

Let us say that a node i is *labeled* if we have determined that there exists an unsaturated path from s to i.

(a) Suppose that node i is labeled, that we have an unsaturated path P from s to i, and that (i,j) is an arc for which $f_{ij} < u_{ij}$. We may then append arc (i,j) to the path P, and obtain an unsaturated path from s to j. Thus, node j can also be labeled.

(b) Similarly, if we have an unsaturated path P from s to i, and if (j,i) is an arc for which $f_{ji} > 0$, we may append arc (j,i) to P (as a backward arc), and obtain an unsaturated path from s to j. Then, node j can be labeled.

The process of examining all nodes j neighboring a given labeled node i, to determine whether they can also be labeled, is called *scanning* node i. We now have the following algorithm, where I is the set of nodes that have been labeled but not yet scanned.

The labeling algorithm

1. The algorithm is initialized with $I = \{s\}$, and with s being the only labeled node.
2. A typical iteration starts with a set I of labeled, but not yet scanned nodes. If $t \in I$ or if $I = \emptyset$, the algorithm terminates. Otherwise, choose a node $i \in I$ to be scanned, and remove it from the set I. Examine all arcs of the form (i, j) or (j, i).
3. If $(i, j) \in \mathcal{A}$, $f_{ij} < u_{ij}$, and j is unlabeled, then label j, and add j to the set I.
4. If $(j, i) \in \mathcal{A}$, $f_{ji} > 0$, and j is unlabeled, then label j, and add j to the set I.

Note that a node enters the set I only if it changes from unlabeled to labeled. Therefore, a node can enter the set I at most once. Since each iteration removes a node from the set I, the algorithm must eventually terminate. We distinguish between two different possibilities.

(a) Suppose that the algorithm terminates because node t has been labeled. Then, there exists an unsaturated path from s to t, that is, an augmenting path. That path can be easily recovered if we do some extra bookkeeping in the course of the labeling algorithm, as follows. Whenever a node j is labeled while scanning a previously labeled node i, we record node i as the *parent* of j. At the end of the algorithm, we may start at node t, go to its parent, then to its parent's parent, etc., until we reach node s; the resulting path is an augmenting path from s to t.

(b) The second possibility is that the algorithm terminates because the set I is empty. We will now argue that this implies that there exists no augmenting path. Let S be the set of labeled nodes at termination, and suppose that there exists an augmenting path. Since $s \in S$ and $t \notin S$, it follows that there exist two consecutive nodes i and j on the augmenting path, such that $i \in S$ and $j \notin S$. Since i and j are consecutive nodes of an augmenting path, we have either $(i, j) \in \mathcal{A}$ and $f_{ij} < u_{ij}$, or $(j, i) \in \mathcal{A}$ and $f_{ji} > 0$. In either case, we see that node j should have been labeled at the time that node i was scanned. This is a contradiction and shows that no augmenting path exists.

Example 7.6 Consider the network shown in Figure 7.23. The labeling algorithm operates as follows:

1. $I = \{s\}$. Node s is scanned. Nodes 1, 2 are labeled.
2. $I = \{1, 2\}$. Node 1 is scanned. Node 4 is labeled.
3. $I = \{2, 4\}$. Node 4 is scanned. No node is labeled.
4. $I = \{2\}$. Node 2 is scanned. Node 3 is labeled.

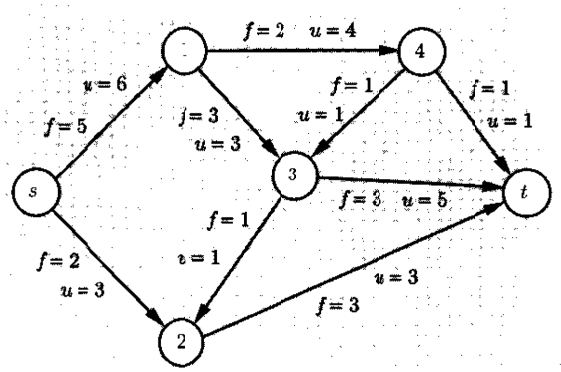


Figure 7.23: The network in Example 7.6 together, with a feasible flow.

5. $I = \{3\}$. Node 3 is scanned. Node t is labeled.

Since node t is labeled, we conclude that there exists an augmenting path, which can be obtained by backtracking, as follows. Node t was labeled while scanning node 3. Node 3 was labeled while scanning node 2. Node 2 was labeled while scanning node s. This leads us to the augmenting path $s, 2, 3, t$.

We conclude our analysis of the labeling algorithm with a brief discussion of its complexity. Every node is scanned at most once, and every arc is examined only when one of its end nodes is scanned. Thus, each arc is examined at most twice. Examining an arc entails only a constant (and small) number of arithmetic operations. We conclude that the computational complexity of the algorithm is proportional to the number of arcs.

We now formally record our conclusions so far.

Theorem 7.9 *The labeling algorithm runs in time $O(|\mathcal{A}|)$. At termination, the node t is labeled if and only if there exists an augmenting path.*

Cuts

We define an *s-t cut* as a subset S of the set of nodes $\mathcal{N}$, such that $s \in S$ and $t \notin S$. In our context, the nodes s and t are fixed, and we refer to S as simply a *cut*. We define the *capacity* $C(S)$ of a cut S as the sum of the capacities of the arcs that cross from S to its complement, that is,

$$C(S) = \sum_{\{(i,j) \in \mathcal{A} \mid i \in S,\ j \notin S\}} u_{ij}$$

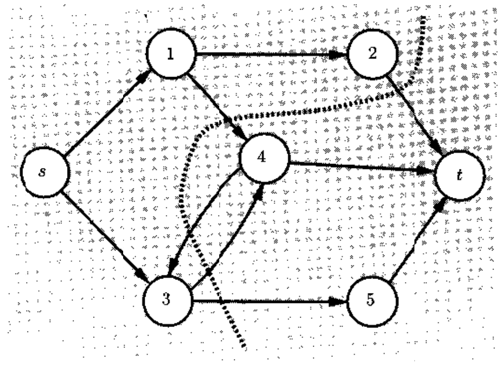


Figure 7.24: The set $S = \{s, 1, 2, 3\}$ is a cut. The capacity of this cut is $u_{2t} + u_{14} + u_{34} + u_{35}$.

(see Figure 7.24). Any flow from s to t must at some point cross an arc (i, j) with $i \in S$ and $j \notin S$. For this reason, the value v of any feasible flow satisfies

$$v \leq C(S), \tag{7.14}$$

for every cut. In essence, each cut provides a potential bottleneck for the maximum flow. Our next result shows that the value of the maximum flow is equal to the tightest of these bottlenecks.

Theorem 7.10

(a) *If the Ford-Fulkerson algorithm terminates because no augmenting path can be found, then the current flow is optimal.*

(b) **(Max-flow min-cut theorem)** *The value of the maximum flow is equal to the minimum cut capacity.*

Proof. (a) Suppose that the Ford-Fulkerson algorithm has terminated because it failed to find an augmenting path. Let S be the set of labeled nodes at termination. These are the nodes i for which there exists an unsaturated path from s to i. Since the search for an augmenting path starts by labeling node s, we have $s \in S$. On the other hand, since no augmenting path was found, node t is not labeled. Therefore, the set S is a cut. For every arc $(i, j) \in \mathcal{A}$, with $i \in S$ and $j \notin S$, we must have $f_{ij} = u_{ij}$. (Otherwise, node j would have been labeled by the labeling algorithm.) Thus, the total amount of flow that exits the set S is equal to $C(S)$. In addition, if $(i, j) \in \mathcal{A}$, with $i \notin S$ and $j \in S$, then $f_{ij} = 0$. (Otherwise, node i would have been labeled by the labeling algorithm.) Thus, the flow crossing from S to its complement cannot return to S, and must exit at the sink node t; see Figure 7.25. This establishes that the value of the

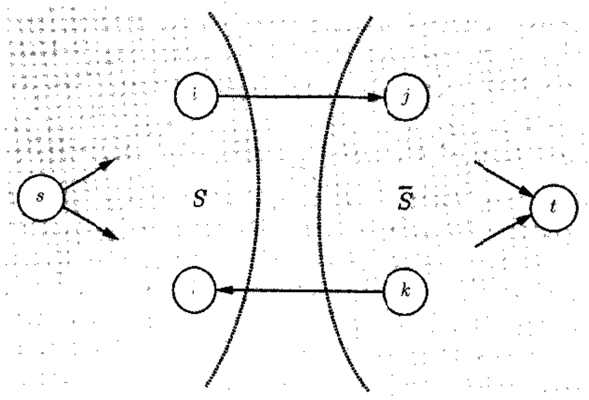


Figure 7.25: Let S and $\overline{S}$ be the sets of labeled and unlabeled nodes, respectively, at termination of the Ford-Fulkerson algorithm. Since j is not labeled, we must have $f_{ij} = u_{ij}$. Since k is not labeled, we must have $f_{kl} = 0$. In particular, all flow moves from s to the rest of S, then to nodes in $\overline{S}$, and finally exits at t.

flow from s to t, when the Ford-Fulkerson algorithm terminates, is equal to $C(S)$. Since the value of the maximum flow can be no higher than $C(S)$ [cf. Eq. (7.14)], we conclude that at termination of the Ford-Fulkerson algorithm, an optimal flow is obtained.

(b) If the optimal value of the flow is infinite, it is not hard to see that there must exist a directed path P from s to t (consisting only of forward arcs), such that every arc in P has infinite capacity. For every cut S, there is an arc (i, j) on the path P such that $i \in S$ and $j \notin S$. Since that arc has infinite capacity, we conclude that $C(S) = \infty$. Since this is true for every cut, we conclude that the minimum cut capacity is infinite and equal to the maximum flow value.

Suppose now that the optimal value, denoted by v^*, is finite. This implies that there exists an optimal solution, that is, a flow whose value is v^*. Let us apply the Ford-Fulkerson algorithm, starting with an optimal flow. Due to optimality of the initial flow, no flow augmentation is possible, and the algorithm terminates with the first iteration. Let S be the set of labeled nodes at termination, as in part (a). From the argument in the proof of part (a), it follows that $C(S) = v^*$. On the other hand, we have $v^* \leq C(S')$ for every cut S'. It follows that $C(S)$ is the minimum cut capacity and is equal to the value of a maximum flow. □

The proof of the max-flow min-cut theorem did rely on the details of the Ford-Fulkerson algorithm. On the other hand, since this theorem relates the optimal values of two optimization problems, one being a minimization and the other being a maximization problem, it is reminiscent of the duality theorem. Indeed, the max-flow min-cut theorem can be proved by constructing a suitable pair of linear programming problems, dual to each other, and then appealing to the duality theorem (Exercise 7.20).

The complexity of the Ford-Fulkerson algorithm

We close with a discussion of the computational complexity of the Ford-Fulkerson algorithm. We assume that every arc capacity is either integer or infinite, and that the maximum flow value is finite. Let U be the largest of those arc capacities that are finite. The capacity of any cut is either infinite or bounded above by $|\mathcal{A}| \cdot U$. If the maximum flow value is finite, there exists at least one cut with finite capacity, and the value is bounded above by $|\mathcal{A}| \cdot U$. Therefore, there can be at most $|\mathcal{A}| \cdot U$ flow augmentations. Since each flow augmentation involves $O(|\mathcal{A}|)$ computations (to run the labeling algorithm), the overall complexity of the algorithm is $O(|\mathcal{A}|^2 \cdot U)$. Under the stronger assumption that all arcs outgoing from node s have finite capacity, the maximum flow value can be bounded above by $|\mathcal{N}| \cdot U$, by focusing on these arcs. The complexity bound then becomes $O(|\mathcal{A}| \cdot |\mathcal{N}| \cdot U)$.

The linear dependence of our complexity estimate on U is unappealing, especially if U is a large number. Exercise 7.25 develops a related algorithm whose complexity is proportional to the *logarithm* of U. The key idea is to *scale* the arc capacities, leading to a new problem with smaller arc capacities, which is easier to solve, and whose optimal solution provides a near-optimal solution to the original problem.

There is an alternative method that eliminates the dependence on U altogether. As mentioned earlier, if we always choose an augmenting path with the least possible number of arcs, the number of iterations is $O(|\mathcal{A}| \cdot |\mathcal{N}|)$, which implies that the complexity is $O(|\mathcal{A}|^2 \cdot |\mathcal{N}|)$. With proper implementation, this complexity estimate can be further reduced.

7.6 Duality in network flow problems

In this section, we examine the structure of the dual of the network flow problem. For simplicity, we restrict ourselves to the uncapacitated case. We provide interpretations of the dual variables, of complementary slackness, and of the duality theorem. Throughout this section, we let Assumption 7.1 be in effect; that is, the network is assumed to be connected and the supplies satisfy $\sum_{i \in \mathcal{N}} b_i = 0$.

The dual problem

The dual of the uncapacitated network flow problem is

$$\begin{array}{ll} \text{maximize} & \mathbf{p}'\mathbf{b} \\ \text{subject to} & \mathbf{p}'\mathbf{A} \le \mathbf{c}'. \end{array}$$

Due to the structure of $\mathbf{A}$, the dual constraints are of the form

$$p_i - p_j \le c_{ij}, \qquad (i,j) \in \mathcal{A}.$$

Suppose that $(p_1, \ldots, p_n)$ is a dual feasible solution. Let θ be some scalar and consider the vector $(p_1 + \theta, \ldots, p_n + \theta)$. It is clear that this is also a dual feasible solution. Furthermore, using the equality $\sum_{i \in \mathcal{N}} b_i = 0$, we have

$$\sum_{i=1}^{n} (p_i + \theta) b_i = \sum_{i=1}^{n} p_i b_i + \theta \sum_{i=1}^{n} b_i = \sum_{i=1}^{n} p_i b_i.$$

Thus, adding a constant to all components of a dual vector is of no consequence as far as dual feasibility or the dual objective is concerned. For this reason, we can and will assume throughout this section that p_n has been set to zero. Note that this is equivalent to eliminating the (redundant) flow conservation constraint for node n.

According to the duality theorem in Chapter 4, if the original problem has an optimal solution, so does the dual, and the optimal value of the objective function is the same for both problems. The example that follows provides an interpretation of the duality theorem in the network context.

Example 7.7 Suppose that we are running a business and that we need to transport a quantity $b_i > 0$ of goods from each node $i = 1, \ldots, n-1$, to node n through our private network. The solution to the corresponding network flow problem provides us with the best way of transporting these goods.

Consider now a transportation services company that offers to transport goods from any node i to node n, at a unit price of p_i. If (i,j) is an arc in our private network, we can always transport some goods from i to j, at a cost of c_{ij} and then give them to the transportation services company to transport them to node n. This would cost us $c_{ij} + p_j$ per unit of goods. The transportation services company knows $\mathbf{b}$ and $\mathbf{c}$. It wants to take over all of our transportation business, and it sets its prices so that we have no incentive of using arc (i,j). In particular, prices are set so that $p_i \le c_{ij} + p_j$, and $p_n = 0$. Having ensured that its prices are competitive, it now tries to maximize its total revenue $\sum_{i=1}^{n-1} p_i b_i$. The duality theorem asserts that its optimal revenue is the same as our optimal cost if we were to use our private network. In other words, when the prices are set right, the new options opened up by the transportation services company will not result in any savings on our part.

Sensitivity

We now provide a sensitivity interpretation of the dual variables. In order to establish a connection with the theory of Chapter 5, we assume that we have eliminated the flow conservation constraint associated with node n, and that the remaining equality constraints are linearly independent.

Suppose that $\mathbf{f}$ is a nondegenerate optimal basic feasible solution, associated with a certain tree, and let $\mathbf{p}$ be the optimal solution to the dual.

Consider some node $i \neq n$ and let p_i be the associated dual variable. Let us change the supply b_i to $b_i + \epsilon$, where ϵ is a small positive number, while keeping the supplies $b_2, \ldots, b_{n-1}$ unchanged. The condition $\sum_{i=1}^{n} b_i = 0$ then requires that b_n be changed to $b_n - \epsilon$, but this only affects the nth equality constraint which has already been omitted. As long as we insist on keeping the same basis, the only available option is to route the supply increment ϵ from node i to the root node n, along the unique path determined by the tree. Because of the way that dual variables are calculated [cf. Eq. (7.10) in Section 7.3], the resulting cost change is precisely ϵp_i. This is in agreement with the discussion in Chapters 4 and 5, where we saw that a dual variable is the sensitivity of the cost with respect to changes in the right-hand side of the equality constraints.

By following a similar reasoning, we see that if we increase b_i by ϵ, decrease b_j by ϵ, keep all other supplies unchanged, and use the same basis, the resulting cost change is exactly $\epsilon(p_i - p_j)$, in the absence of degeneracy and for small ϵ. We conclude that, in the absence of degeneracy, $p_i - p_j$ is the marginal cost of shipping an additional unit of flow from node i to node j.

Complementary slackness

The complementary slackness conditions for the minimum cost network flow problem are the following:

(a) If $p_i \neq 0$, then $[\mathbf{Af}]_i = b_i$. This condition is automatically satisfied by any feasible flow $\mathbf{f}$.

(b) If $f_{ij} > 0$, then $p_i - p_j = c_{ij}$. This condition is interpreted as follows. We have $p_i - p_j \leq c_{ij}$, by dual feasibility. If $p_i - p_j < c_{ij}$, then there is a way of sending flow from i to j, which is less expensive than using arc (i,j). Hence, that arc should not carry any flow.

From Theorem 4.5 in Section 4.3, we know that $\mathbf{f}$ is primal optimal and $\mathbf{p}$ is dual optimal if and only if $\mathbf{f}$ is primal feasible, $\mathbf{p}$ is dual feasible, and complementary slackness holds. Consider now Figure 7.26, which captures the dual feasibility constraint $p_i - p_j \leq c_{ij}$, the nonnegativity constraint $f_{ij} \geq 0$, and the second complementary slackness condition. We then obtain the following result.

Theorem 7.11 *For any uncapacitated network flow problem, the following are equivalent.*

(a) *The vectors $\mathbf{f}$ and $\mathbf{p}$ are optimal solutions to the primal and the dual problem, respectively.*

(b) *The vector $\mathbf{f}$ satisfies the flow conservation equation $\mathbf{Af} = \mathbf{b}$, and for every arc (i,j), the pair $(p_i - p_j, f_{ij})$ satisfies the relations indicated by Figure 7.26.*

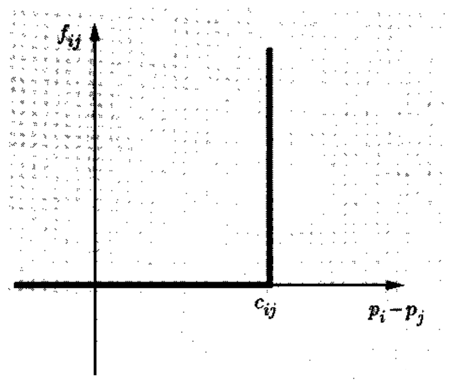


Figure 7.26: Illustration of the complementary slackness conditions. For any arc (i,j), the pair $(p_i - p_j, f_{ij})$ must lie on the heavy line.

A circuit analogy

We now draw an analogy between networks, as defined in this chapter, and electrical circuits. We visualize each node in the network as a place where several "wires" meet, and each arc as a two-terminal circuit element through which current may flow. Let us think of f_{ij} as the current on arc (i,j), and let b_i be the current pumped into the circuit at node i, by means of a current source. Then, the flow conservation equation $\mathbf{Af} = \mathbf{b}$ amounts to Kirchoff's current law. Let us view p_i as an electric potential. In these terms, Figure 7.26 specifies a relation between the "potential difference" $p_i - p_j$ across arc (i,j) and the current through that same arc. Such a relation is very much in the spirit of Ohm's law (potential difference equals current times resistance) except that here the relation between the potential difference and the current is a bit more complicated.

In circuit terms, Theorem 7.11 can be restated as follows. The vectors $\mathbf{f}$ and $\mathbf{p}$ are optimal solutions to the primal and dual problem, respectively, if and only if they are equal to an equilibrium state of an electrical circuit, where each circuit element is described by the relation specified by Figure 7.26. If circuit elements with the properties indicated by Figure 7.26 were easy to assemble and calibrate, we could build a circuit, drive it with current sources, and let it come to equilibrium. This would be an analog device that solves the network flow problem. While such devices do not seem promising at present, the conceptual connections with circuit theory are quite deep, and are valid in greater generality (e.g., in network flow problems with a convex nonlinear cost function).

7.7 Dual ascent methods*

In this section, we introduce a second major class of algorithms for the network flow problem, based on dual ascent. These algorithms maintain at all times a dual feasible solution which, at each iteration, is updated in a direction of increase of the dual objective (such a direction is called a *dual ascent* direction), until the algorithm terminates with a dual optimal solution. Algorithms of this type seem to be among the fastest available. In this section, we only consider the special case where all arc capacities are infinite; the reader is referred to the literature for extensions to the general case.

Recall that the dual of the network flow problem takes the form

$$\begin{array}{ll} \text{maximize} & \displaystyle\sum_{i=1}^{n} p_i b_i \\ \text{subject to} & p_i \le c_{ij} + p_j, \qquad (i,j) \in \mathcal{A}. \end{array}$$

Given a dual feasible vector $\mathbf{p}$, we are interested in changing $\mathbf{p}$ to a new feasible vector $\mathbf{p} + \theta \mathbf{d}$, where θ is a positive scalar, and where $\mathbf{d}$ satisfies $\mathbf{d}'\mathbf{b} > 0$ (which makes $\mathbf{d}$ a dual ascent direction).

Let S be some subset of the set $\mathcal{N} = \{1, \ldots, n\}$ of nodes. The *elementary direction* $\mathbf{d}^S$ associated with S is defined as the vector with components

$$d_i^S = \begin{cases} 1, & \text{if } i \in S, \\ 0, & \text{if } i \notin S. \end{cases}$$

Moving along an elementary direction is the same as picking a set S of nodes and raising the "price" p_i of each one of these nodes by the same amount. A remarkable property of network flow problems is that the search for a feasible ascent direction can be confined to the set of elementary directions, as we now show.

Theorem 7.12 *Let $\mathbf{p}$ be a feasible solution to the dual problem. Then, either $\mathbf{p}$ is dual optimal or there exists some $S \subset \mathcal{N}$ and some $\theta > 0$, such that $\mathbf{p} + \theta\mathbf{d}^S$ is dual feasible and $(\mathbf{d}^S)'\mathbf{b} > 0$.*

Proof Let $S \subset \mathcal{N}$ and consider the vector $\mathbf{d}^S$. We start by deriving conditions under which $\mathbf{p} + \theta\mathbf{d}^S$ is feasible for some $\theta > 0$. We only need to check whether any active dual constraints are violated by moving along $\mathbf{d}^S$. Note that the dual constraint corresponding to an arc $(i,j) \in \mathcal{A}$ is active if and only if $p_i = c_{ij} + p_j$, in which case we say that the arc is *balanced*. Clearly, if (i,j) is a balanced arc and if $i \in S$, raising the value of p_i will violate the constraint $p_i \le c_{ij} + p_j$, unless the value of p_j is also raised. We conclude that dual feasibility of $\mathbf{p} + \theta\mathbf{d}^S$, for some $\theta > 0$, amounts to the

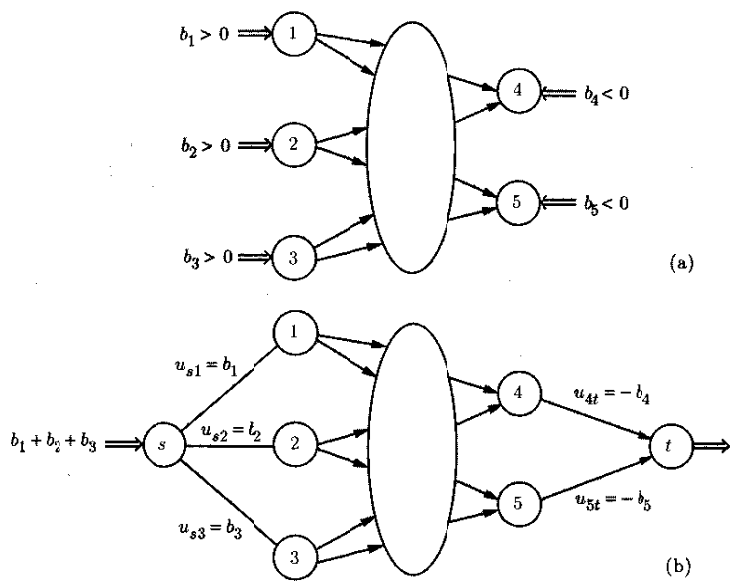


Figure 7.27: (a) A network with some source nodes and some sink nodes, and in which we have only kept the balanced arcs. (b) A corresponding maximum flow problem; all arcs have infinite capacity with the exception of the arcs (s,i) and (j,t), where i is a source and j is a sink in the original network. There is a feasible solution to the problem in (a) if and only if the optimal value in the maximum flow problem in (b) is equal to $V = b_1 + b_2 + b_3$.

following requirement:

$$\text{if } i \in S \text{ and } (i,j) \text{ is balanced, then } j \in S. \tag{7.15}$$

Let $Q_+ = \{i \in \mathcal{N} \mid b_i > 0\}$ be the set of source nodes and let $Q_- = \{i \in \mathcal{N} \mid b_i < 0\}$ be the set of sink nodes. Let $V = \sum_{i \in Q_+} b_i$ be the total amount of flow that has to be routed from the sources to the sinks. Our first step is to determine whether the entire supply can be routed to the sinks using *only balanced arcs*. This is accomplished by solving a maximum flow problem of the type shown in Figure 7.27.

Let us run the labeling algorithm, starting from a maximum flow $\mathbf{f}$. Since we already have a maximum flow, no augmenting path is found and node t remains unlabeled. We partition the set $\{1, \ldots, n\}$ of original nodes into sets S and $\overline{S}$ of labeled and unlabeled nodes, respectively. Then, the situation is as shown in Figure 7.28(a).

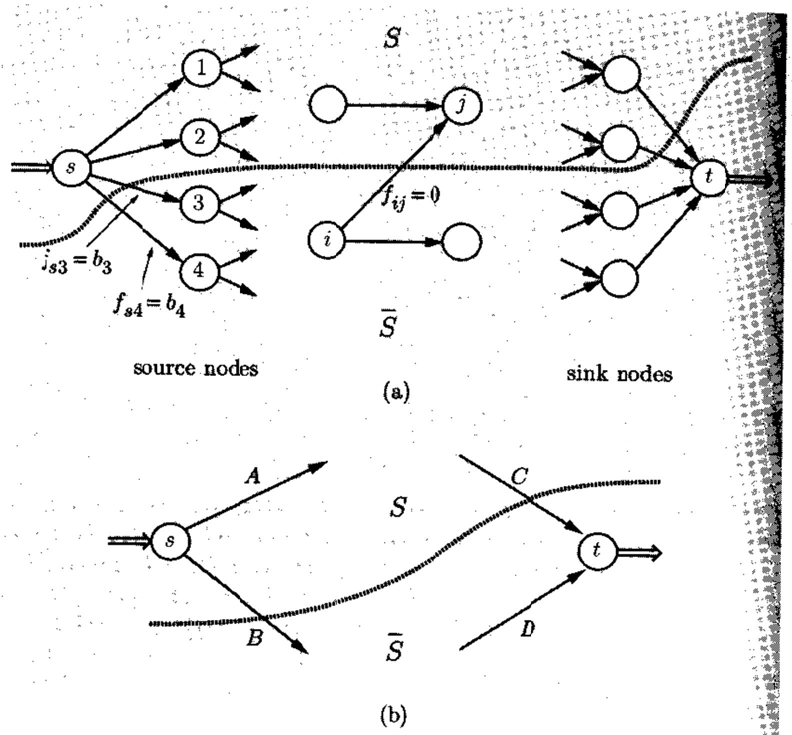


Figure 7.28: The cut obtained at termination of the labeling algorithm, for the network involving only balanced arcs. (a) If a source node i is not labeled, we must have $f_{si} = b_i$ and arc (s, i) is saturated. If a sink node j is labeled, we must have $f_{jt} = |b_j|$ and arc (j, t) is saturated, because otherwise node t would also be labeled. If (i, j) is a balanced arc and if $i \in S$, then we must also have $j \in S$, because otherwise node j would have been labeled (recall that arc capacities are infinite). If (i, j) is a balanced arc and i is not in S, j can be either in S or outside S; if $j \in S$, we must have $f_{ij} = 0$, because otherwise node i would have been labeled. (b) Interpretation of the variables A, B, C, D.

Let

$$A = \sum_{i \in Q_+ \cap S} f_{si}, \qquad C = \sum_{j \in Q_- \cap S} f_{jt},$$

$$B = \sum_{i \in Q_+ \cap \bar{S}} f_{si}, \qquad D = \sum_{j \in Q_- \cap \bar{S}} f_{jt};$$

see Figure 7.28(b) for an interpretation. The total flow F that leaves node s is equal to $A + B$. On the other hand, all of the flow must at some point traverse an arc that starts in $\{s\} \cup S$ and ends in $\{t\} \cup \bar{S}$. By adding the flow of all such arcs, we obtain $F = B + C$. We conclude that $A = C$, or

$$\sum_{i \in Q_+ \cap S} f_{si} = \sum_{j \in Q_- \cap S} f_{jt}.$$

For every labeled sink node $j \in Q_- \cap S$, we have $f_{jt} = |b_j| = -b_j$, because otherwise node t would have been labeled, which shows that

$$\sum_{i \in Q_+ \cap S} f_{si} = \sum_{j \in Q_- \cap S} |b_j|.$$

We finally note that

$$(\mathbf{d}^S)'\mathbf{b} = \sum_{i \in S} b_i = \sum_{i \in Q_+ \cap S} b_i - \sum_{j \in Q_- \cap S} |b_j| \geq \sum_{i \in Q_+ \cap S} f_{si} - \sum_{j \in Q_- \cap S} |b_j| = 0.$$

We distinguish between two cases. If $(\mathbf{d}^S)'\mathbf{b} > 0$, we have a dual ascent direction, as desired. On the other hand, if $(\mathbf{d}^S)'\mathbf{b} = 0$, we must have $f_{si} = b_i$ for every $i \in Q_+ \cap S$. Since we also have $f_{si} = b_i$ for every $i \in Q_+ \cap \bar{S}$, it follows that the value of the maximum flow is equal to $V = \sum_{i \in Q_+} b_i$, and we have a feasible solution to the original (primal) network flow problem. In addition, since positive flow is only carried by the balanced arcs, complementary slackness holds, and we have an optimal solution to the primal and the dual problem. $\square$

Theorem 7.12 leads to a general class of algorithms for the network flow problem.

Dual ascent algorithm

1. A typical iteration starts with a dual feasible solution $\mathbf{p}$.
2. Search for a set $S \subset \mathcal{N}$ with the property

$$\text{if } i \in S \text{ and } (i,j) \text{ is balanced, then } j \in S,$$

and such that $\sum_{i \in S} b_i > 0$. If no such set S exists, $\mathbf{p}$ is du
optimal and the algorithm terminates.

3. Update $\mathbf{p}$ to $\mathbf{p} + \theta^* \mathbf{d}^S$, where θ^* is the largest value for whi
$\mathbf{p} + \theta \mathbf{d}^S$ is dual feasible. If $\theta^* = \infty$, the algorithm terminat
otherwise, go back to Step 2.

The value of θ^* in the dual ascent algorithm is easily determined, a
follows. We consider each constraint $p_i \le c_{ij} + p_j$. The possibility tha
$\mathbf{p} + \theta \mathbf{d}^S$ may violate this constraint arises only if $i \in S$ and $j \notin S$. For such
pairs (i,j), we need $p_i + \theta \le c_{ij} + p_j$, and we obtain

$$\theta^* = \min_{\{(i,j) \in \mathcal{A} \mid i \in S,\ j \notin S\}} (c_{ij} + p_j - p_i). \qquad (7.16)$$

If there is no (i,j) for which $i \in S$ and $j \notin S$, then θ can be taken arbitrarily
large, and we let $\theta^* = \infty$. Since we are moving in a direction of dual cost
increase, this implies that the optimal dual cost is $+\infty$ and, in particular,
the primal problem is infeasible.

Our next results deals with the finite termination of the algorithm.

Theorem 7.13 *Suppose that the optimal cost is finite. If the cost coefficients c_{ij} are all integer, and if the dual ascent algorithm is ini-tialized with an integer dual feasible vector, it terminates in a finite number of steps with a dual optimal solution.*

Proof. Suppose that the algorithm is initialized with an integer vector $\mathbf{p}$. Then, the value of θ^* is integer. (It cannot be infinite, because the dual optimal cost would also be infinite, which we assumed not to be the case.) Let $v = \min_S (\mathbf{d}^S)'\mathbf{b}$, where the minimum is taken over all S for which $(\mathbf{d}^S)'\mathbf{b} > 0$. Clearly, v is positive. Since θ^* is integer, every iteration increases the dual objective by at least v. It follows that the algorithm must terminate after a finite number of steps. □

There are several variations of the dual ascent algorithm which differ primarily in the method that they use to search for an elementary dual ascent direction. If the set S is chosen as in the proof of Theorem 7.12, we have the so-called *primal-dual* method. (When specialized to the assignment problem, it is also known as the *Hungarian* method.) It can be verified

that the primal-dual method uses a "steepest" ascent direction, that is, an elementary ascent direction that maximizes $(\mathbf{d}^S)'\mathbf{b}$ (Exercise 7.30). On the other hand, the so-called *relaxation* method tries to discover an elementary ascent direction $\mathbf{d}^S$ as quickly as possible. In one implementation, a one-element set S is tried first. If it cannot provide a direction of ascent, the set is progressively enlarged until an ascent direction is found. In practice, a greedy search of this type pays off and the relaxation method is one of the fastest available methods for linear network flow problems.

In all of the available dual ascent methods, the search for an elementary ascent direction is streamlined and organized by maintaining a nonnegative vector $\mathbf{f}$ of primal flow variables. Throughout the algorithm, the vectors $\mathbf{f}$ and $\mathbf{p}$ are such that the complementary slackness condition $(c_{ij} + p_j - p_i) f_{ij} = 0$ is enforced. (That is, flow is only carried by balanced arcs.) If such a complementary vector $\mathbf{f}$ is primal feasible, we have an optimal solution to both the primal and the dual. For this reason, dual ascent algorithms can be alternatively described by focusing on the vector $\mathbf{f}$, and by interpreting the different steps as an effort to attain primal feasibility. (This is also the historical reason for the term "primal-dual.")

The primal-dual method

In this subsection, we consider in greater depth the primal-dual method. We do that in order to develop a complexity estimate, and also to illustrate how a network algorithm can be made more efficient by suitable refinements.

The primal-dual method is the special case of the dual ascent algorithm, where the set S is chosen exactly as in the proof of Theorem 7.12. In particular, given a dual feasible vector $\mathbf{p}$, we form a maximum flow problem, in which only the balanced arcs are retained, and we let S be the set of nodes in $\{1, \ldots, n\}$ that are labeled at termination of the maximum flow algorithm. We then update the price vector from $\mathbf{p}$ to $\mathbf{p} + \theta^* \mathbf{d}^S$, form a new maximum flow problem, and continue similarly.

We provide some observations that form the basis of efficient implementations of the algorithm.

(a) *The maximum flow and the current dual vector satisfy the complementary slackness condition* $(c_{ij} + p_j - p_i) f_{ij} = 0$. This is because in the maximum flow problem, we only allow flow on balanced arcs.

(b) *If we determine a maximum flow and then perform a dual update, the complementary slackness condition $(c_{ij} + p_j - p_i) f_{ij} = 0$ is preserved.* Suppose that an arc (i,j) carries positive flow in the solution to the maximum flow problem under the old prices. In particular, (i,j) must have been a balanced arc before the dual update. Note that $i \in S$ if and only if $j \in S$. (If $i \in S$, then j gets labeled because the arc capacity is infinite; if $j \in S$, then i gets labeled because $f_{ij} > 0$.) This implies that p_i and p_j are changed by the same amount, the arc (i,j) remains balanced, and complementary slackness is preserved.

(c) An important consequence of observation (b) is that subsequent to a dual update, we do not need to solve a new maximum flow problem from scratch. Instead, we use the maximum flow under the old prices as an initial feasible solution to the maximum flow problem under the new prices. Furthermore, the nodes that were labeled at termination of the maximum flow algorithm under the old prices, will be labeled at the first pass of the labeling algorithm under the new prices. [This is because if node j got its label from a node i through a balanced arc (i,j) or (j,i), then p_i and p_j get raised by the same amount, the arc (i,j) or (j,i) remains balanced, and that arc can be used to label j under the new prices.] Our conclusion is that subsequent to a dual update and given the current flow, we do not need to start the labeling algorithm from scratch, but we can readily assign a label to all nodes that were previously labeled.

(d) *A dual update (with $\theta^* < \infty$) results in at least one unlabeled node becoming labeled.* Consider an arc (i,j) with $i \in S$, $j \notin S$, and such that $\theta^* = c_{ij} + p_j - p_i$. Such an arc exists by the definition of θ^*. Subsequent to the dual update, this arc becomes balanced. At the first pass of the labeling algorithm, node j will inherit a label from node i.

The preceding observations lead to a new perspective of the primal-dual method. Instead of viewing the algorithm as a sequence of dual updates, with maximum flow problems solved in between, we can view it as a sequence of applications of the labeling algorithm, resulting in flow augmentations, interrupted by dual updates that create new labeled nodes.

At the beginning of a typical iteration, we have a price vector, a flow vector that only uses balanced arcs, and a set of labeled nodes; these are nodes to which additional flow can be sent, using only balanced arcs. We distinguish two cases:

(a) If node t is labeled, we have discovered an augmenting path and we are not yet at an optimal solution to the maximum flow problem. We push as much flow as possible along the augmenting path. At this point, we delete all labels and start another round of the labeling algorithm, to see whether further flow augmentation is possible.

(b) If node t is not labeled, we have a maximum flow and we perform a dual update. Right after the dual update, we resume with the labeling algorithm, but without erasing the old labels. Recall that a dual update results in at least one new balanced arc (i,j), with node i previously labeled and node j previously unlabeled. Node i remains labeled and j will now become labeled. Since every dual update results in an additional node being labeled, node t will become labeled after at most n dual updates, and a flow augmentation will take place.

We can now get an upper bound on the running time of the algorithm.

Let, as before, V be the sum of the supplies at the source nodes. Assuming that all supplies are integer, there can be at most V flow augmentations. Since there can be at most n dual updates between any two successive flow augmentations, the algorithm terminates after at most nV dual updates. If at each dual update we determine θ^* using Eq. (7.16), we need $O(m)$ arithmetic operations per dual update, and the running time of the algorithm is $O(mnV) = O(n^4B)$, where $B = \max_i |b_i|$. With a more clever way of computing θ^*, the running time can be brought down to $O(n^3B)$ (Exercise 7.28). For the assignment problem, we have $B = 1$, and we obtain the so-called Hungarian method, which runs in time $O(n^3)$.

Example 7.8 We go through an example of the primal-dual method. Consider the network shown in Figure 7.29(a), and let us start with the dual vector $\mathbf{p} = (1,1,1,1,0)$. It is easily checked that we have $p_i \le c_{ij} + p_j$ for all arcs (i,j), and we therefore have a dual feasible solution. The balanced arcs are $(1,4)$, $(2,4)$, $(3,5)$. In Figure 7.29(b), we form a maximum flow problem involving only the balanced arcs. We solve this problem using the Ford-Fulkerson algorithm. At termination, we obtain the labels and the flows shown in Figure 7.29(c). (Node 2 inherits a label from node 4.) The set of labeled nodes is $S = \{1,2,4\}$ and the corresponding elementary direction is $\mathbf{d}^S = (1,1,0,1,0)$. The only arc (i,j) with $i \in S$, $j \notin S$, is the arc $(2,5)$, and Eq. (7.16) yields $\theta^* = 2$. The new dual vector is $\mathbf{p} + \theta^*\mathbf{d}^S = (3,3,1,3,0)$. The arc (2,5) has now become balanced and all nodes that were labeled remain labeled. Since node 2 is labeled, and arc $(2,5)$ has become balanced, node 5 gets labeled. Finally, because arc $(5,t)$ is unsaturated ($f_{5t} = 2 < 3 = |b_5|$), node t also gets labeled. At this point, we have identified a path through which additional flow can be shipped, namely the path $s,1,4,2,5,t$. By shipping one unit of flow along this path, the value of the flow becomes 8. We now have a feasible solution to the original primal problem, which satisfies complementary slackness, and is therefore optimal. If primal feasibility had not been attained, we would erase all labels and rerun the labeling algorithm, in an attempt to discover a new augmenting path.

Comparison with the dual simplex method

Network flow problems (like all linear programming problems) can be solved by the dual simplex method. This is also a dual ascent method, in the sense that it maintains a dual feasible solution and keeps increasing the dual objective. Furthermore, it can be verified that dual updates in the dual simplex method only take place along elementary directions (Exercise 7.31). On the other hand, the dual simplex method can only visit basic feasible solutions in the dual feasible set. In contrast, the methods considered in this section, have more directions to choose from and do not always move along the edges of the dual feasible set.

A key difference between the dual simplex method and the dual ascent methods of this section is in the nature of the auxiliary flow information

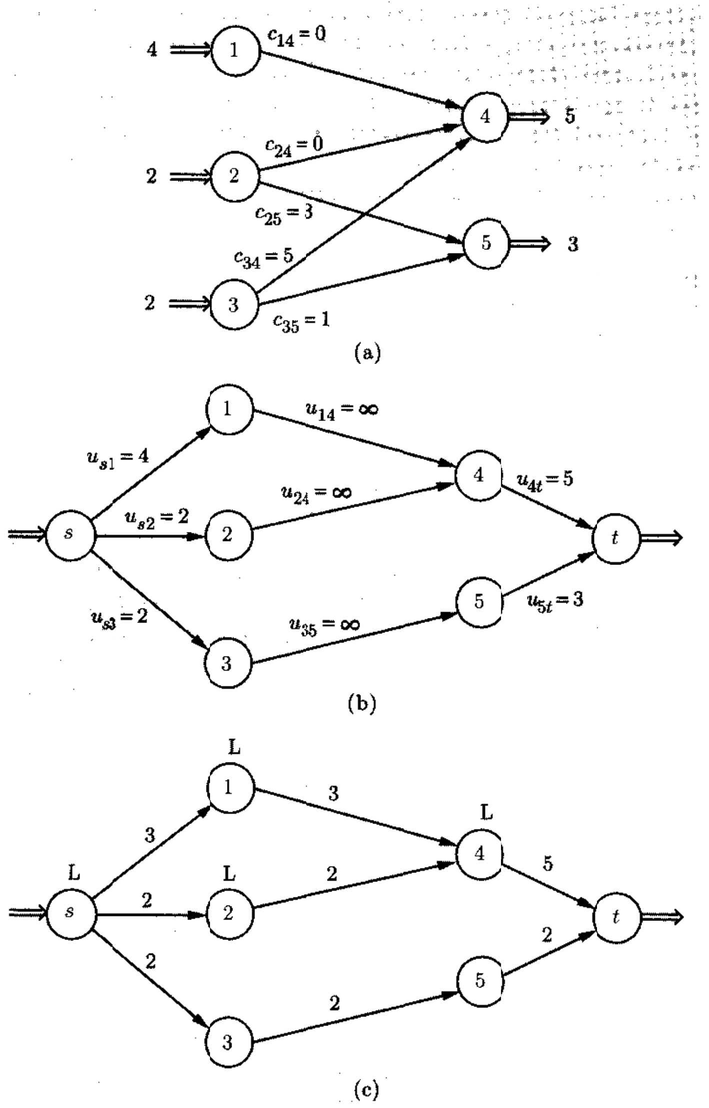


Figure 7.29: Illustration of the primal-dual method in Example 7.8.

that they employ. In the dual simplex method, we maintain a basic solution to the primal; in particular, the flow conservation constraints are always satisfied. If the basic solution is infeasible, it is only because some of the nonnegativity constraints are violated. In contrast, with the dual ascent methods of this section, auxiliary flow variables are always nonnegative, but we allow the flow conservation equations to be violated.

7.8 The assignment problem and the auction algorithm

The auction algorithm, which is the subject of this section, is a method that can be used to solve general network flow problems. We restrict ourselves to a special case, the assignment problem, because it results in a simpler and more intuitive form of the algorithm. The auction algorithm resembles dual ascent methods, except that it only changes the price of a single node at a time. Given a nonoptimal feasible solution to the dual, it is sometimes impossible to find a dual ascent direction involving a single node. For this reason, a typical iteration may result in a temporary deterioration (i.e., decrease) of the dual objective. As long as this deterioration is kept small, the algorithm is guaranteed to make progress in the long run, and can be viewed as an approximate dual ascent method. Our presentation bypasses this approximate dual ascent interpretation, for which the reader is referred to the literature.

The problem

$$\begin{array}{lll}
\text{minimize} & \displaystyle\sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij} f_{ij} & \\
\text{subject to} & \displaystyle\sum_{i=1}^{n} f_{ij} = 1, & j = 1,\ldots,n, \\
& \displaystyle\sum_{j=1}^{n} f_{ij} = 1, & i = 1,\ldots,n, \\
& f_{ij} \geq 0, & \forall\, i,j.
\end{array}$$

is known as the *assignment* problem. One interpretation is that there are n persons and n projects and that we wish to assign a different person to each project while minimizing a linear cost function of the form $\sum_{(i,j)} c_{ij} f_{ij}$, where $f_{ij}=1$ if the ith person is assigned to the jth project, and $f_{ij}=0$ otherwise. With this interpretation, it would be natural to introduce the additional constraint $f_{ij} \in \{0,1\}$. However, this is unnecessary for the following reasons. First, the constraint $f_{ij} \leq 1$ is implied by the constraints that we already have. Second, Corollary 7.2 implies that the assignment problem always has an integer optimal solution. In particular, if we solve the assignment problem using the simplex method or the negative cost cycle algorithm, the optimal value obtained for each variable f_{ij} will be zero or one.

Let us now digress to mention an interesting special case of the assignment problem. Suppose that the cost coefficients c_{ij} are either zero or one. The resulting problem is called the *bipartite matching* problem and has the following interpretation. We have $c_{ij} = 0$ if and only if person i is compatible with project j and we are interested in finding as many compatible person-project pairs as possible. If the optimal value turns out to be 0, we say that there exists a *perfect matching*. Besides being an assignment problem, the bipartite matching problem is also a special case of the max-flow problem (send as much flow as possible from persons to projects using only zero cost arcs) and as such it can be also solved using maximum flow algorithms. There are even better special purpose algorithms, which can be found in the literature.

Duality and complementary slackness

We form the dual of the assignment problem. We associate a dual variable r_i with each constraint $\sum_{j=1}^n f_{ij} = 1$, and a dual variable p_j to each constraint $\sum_{i=1}^n f_{ij} = 1$. Then, the dual problem takes the form

$$\begin{aligned} \text{maximize} \quad & \sum_{i=1}^n r_i + \sum_{j=1}^n p_j \\ \text{subject to} \quad & r_i + p_j \le c_{ij}, \qquad \forall\, i, j. \end{aligned}$$

It is clear from the form of the dual constraints that once the values of $p_1, \ldots, p_n$ are determined, $\sum_{i=1}^n r_i$ is maximized if we set each r_i to the largest value allowed by the constraints $r_i + p_j \le c_{ij}$, which is

$$r_i = \min_{j=1,\ldots,n} \{c_{ij} - p_j\}. \tag{7.17}$$

This leads to the following equivalent dual problem:

$$\text{maximize} \quad \sum_{j=1}^n p_j + \sum_{i=1}^n \min_j \{c_{ij} - p_j\}. \tag{7.18}$$

Note that this is an unconstrained problem with a piecewise linear concave objective function.

We now consider the complementary slackness conditions for the assignment problem, which are the following:

(a) flow must be conserved;

(b) if $f_{ij} > 0$, then $r_i + p_j = c_{ij}$.

Using Eq. (7.17) to eliminate r_i, the second complementary slackness condition is equivalent to

$$\text{if} \quad f_{ij} > 0, \quad \text{then} \quad p_j - c_{ij} = -r_i = \max_k \{p_k - c_{ik}\}. \tag{7.19}$$

Condition (7.19) admits the following interpretation: each project k carries a reward p_k and if person i is assigned to it, there is a cost c_{ik}. The difference $p_k - c_{ik}$ is viewed as the profit to person i derived from carrying out project k. Condition (7.19) then states that each person should be assigned to a most profitable project.

Auction mechanisms

We recall that a pair of primal and dual solutions is optimal if and only if we have primal and dual feasibility, and complementary slackness. Having defined r_i according to Eq. (7.17), dual feasibility holds automatically. Thus, the problem boils down to finding a set of prices p_j and a feasible assignment, for which the condition (7.19) holds. This motivates a bidding mechanism whereby persons bid for the most profitable projects. It can be visualized by thinking about a set of contractors who compete for the same projects and therefore keep lowering the price (or reward) they are willing to accept for any given project.

Naive auction algorithm

1. *Bidding phase.* Given a set of prices $p_1, \ldots, p_n$ for the different projects, and a partial assignment of persons to projects, each unassigned person finds a best project j, that maximizes the profit $p_j - c_{ij}$, and "bids" for it, by accepting a lower price. In particular, the price is lowered by

   (profit of the best project) − (profit of the second best project).

   This is the maximum amount by which the price could be lowered before the best project ceases to be the best one.
2. Following the bidding phase, there is an assignment phase during which every project is assigned to the lowest bidder (if any). The new price of each project is set to the value of the lowest bid. The old holder of the project (if any) now becomes unassigned.

Example 7.9 Consider an assignment problem involving three persons and three objects; see Figure 7.30. Suppose that all p_i are equal to one, that person 1 is assigned to project 1, person 2 is assigned to object 2, and person 3 is unassigned.

Person 3 computes the profits of the different projects; they are $1 - 0 = 1$ for the first and second project, and $1 - 1 = 0$ for the third project. Person 3 bids for the second object. The bid for project 2 cannot be lower than one, because that would make project 2 less profitable than project 1. Hence, the bid is equal to one. Person 3, as the sole bidder, is assigned project 2, and person 2 becomes unassigned. However, there is no price change. In the next iteration, person 2 who is unassigned goes through a similar process, and bids for project 2. The price is again unchanged, and we end up in exactly the same situation as when the algorithm was started.

As Example 7.9 shows, the naive auction algorithm does not always work. The reason is that if there are two equally profitable projects, a bidder cannot lower the price of either, and the algorithm gets deadlocked. However, the algorithm works properly after a simple modification. Let us fix a positive number ϵ. The bid placed for a project is lower by ϵ than what it would have been if we wished that project to remain the best one; as a result, the project comes short, by ϵ, of being the most profitable one. A complete description of the algorithm is given below.

The auction algorithm

1. A typical iteration starts with a set of prices $p_1, \ldots, p_n$ for the different projects, a set S of assigned persons, and a project j_i assigned to each person $i \in S$ (that is, $f_{ij_i} = 1$, $i \in S$). (At the beginning of the algorithm, the set S is empty.)
2. Each unassigned person $i \notin S$ finds a best project k_i by maximizing the profit $p_k - c_{ik}$ over all k. Let k_i' be a second best project, that is,
$$p_{k_i'} - c_{ik_i'} \geq p_k - c_{ik}, \qquad \text{for all } k \neq k_i.$$
Let
$$\Delta_{k_i} = (p_{k_i} - c_{ik_i}) - (p_{k_i'} - c_{ik_i'}).$$
Person i "bids" $p_{k_i} - \Delta_{k_i} - \epsilon$ for project i.
3. Every project for which there is at least one bid is assigned to a lowest bidder; the old holder of the project (if any) becomes unassigned. The new price p_i of each project that has received at least one bid is set to the value of the lowest bid.

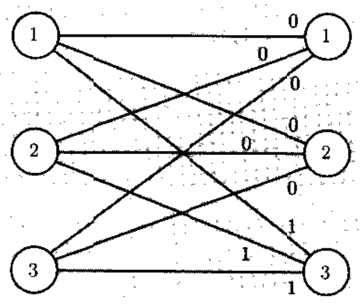


Figure 7.30: An assignment problem. The costs c_{i1} and c_{i2} for the first two projects are zero, for every i. The costs c_{i3} for the third project are equal to one, for every i.

Example 7.10 We apply the auction algorithm to the problem considered in Example 7.9. Once more, we assume that persons 1 and 2 are assigned to projects 1 and 2, respectively, and the initial prices are all equal to 1. Person 3 chooses to bid for project 2 and decreases its price to $1-\epsilon$. Person 2 becomes unassigned and computes the profits of the different projects; they are: $1-0=1$, $(1-\epsilon)-0 = 1-\epsilon$, and $1-1=0$, respectively. Project 1 is the most profitable. Its price is to be brought down so that its profit becomes equal to the profit of the second best project, minus ϵ. Therefore, the bid is equal to $1-2\epsilon$.

At the next iteration, person 1, who is unassigned, bids for project 2 and brings its price down to $1-3\epsilon$. The same process is then repeated. At each iteration, projects 1 and 2 have prices that are within ϵ of each other. An unassigned person always bids for the one that has the larger price, and brings its price down by 2ϵ. After a certain number of iterations, the prices of projects 1 and 2 become negative. At that point, project 3 finally becomes profitable, receives a bid, becomes assigned, and the algorithm terminates.

Note that a bid pushes the price of a project below the level at which that project would be the most profitable. For this reason, persons will not, in general, be assigned to their most profitable project, and the complementary slackness conditions fail to hold. On the other hand, since persons may underbid only by ϵ, the complementary slackness conditions are close to being satisfied. This motivates our next definition.

Definition 7.3 *Consider a set of prices p_j and a partial assignment where each assigned person $i \in S$ is assigned a project j_i. We say that the* **ϵ-complementary slackness** *condition holds if we have*
$$p_{j_i} - c_{ij_i} \geq \max_k \{p_k - c_{ik}\} - \epsilon, \qquad \forall\, i \in S.$$

The following result deals with a key property of the auction algorithm.

Theorem 7.14 *Throughout the auction algorithm, the ϵ-complementary slackness condition is satisfied.*

Proof. The condition is satisfied initially, before any person is assigned. Whenever a person i is assigned a project j_i, the price is chosen so that the profit $p_{j_i} - c_{ij_i}$ cannot be smaller than the profit of any other project by more than ϵ, assuming the other prices do not change. If the prices of some other projects do change, they can only go down, and project j_i is again guaranteed to be within ϵ of being most profitable. As long as a person holds the same project, the price of that project cannot change, and its profit stays constant. In the meantime, the prices of any other projects can only go down, thus reducing their profits, which means that the person still holds a project whose profit is within ϵ of the maximum profit. □

We also have the following result that ensures the finite termination of the algorithm.

Theorem 7.15 *The auction algorithm terminates after a finite number of stages with a feasible assignment.*

Proof. The proof rests on the following observations:

(a) Once a project receives a bid, it gets assigned to some person. Once a project is assigned, it may be later reassigned to another person, but it will never become unassigned. Thus, if all projects have received at least one bid, then all projects are assigned and, consequently, all persons are also assigned.

(b) If all persons are assigned, no person bids and the algorithm terminates.

(c) If the algorithm does not terminate, then some project never gets assigned. Such a project has never received a bid and its price is fixed at its initial value.

(d) If the algorithm does not terminate, some project receives an infinite number of bids. Since every successive bid lowers its price by at least ϵ, the price of such a project decreases to $-\infty$.

Using observations (c) and (d), a project that has never received a bid must eventually become more profitable than any project that receives an infinite number of bids. On the other hand, for a project to receive an infinite number of bids, it must remain more profitable than any project that has not received any bids. This is a contradiction, which establishes that every project will eventually receive a bid. Using observations (a) and (b), the algorithm must eventually terminate with all persons assigned to projects. □

The preceding proof generalizes to the case where some assignments are not allowed, which is the same as setting some of the coefficients c_{ij} to infinity. However, a slightly more involved argument is needed (see Exercise 7.32).

At termination of the auction algorithm, we have:

(a) primal feasibility (all persons are assigned a project);

(b) dual feasibility (if we define $r_i = \max_k\{p_k - c_{ik}\}$, we have a dual feasible solution);

(c) ϵ-complementary slackness (Theorem 7.14).

If we had complementary slackness instead of ϵ-complementary slackness, linear programming theory would imply that we have an optimal solution. As it turns out, because of the special structure of the problem, ϵ-complementary slackness is enough, when ϵ is sufficiently small.

Theorem 7.16 *If the cost coefficients c_{ij} are integer and if*

$$0 < \epsilon < 1/n,$$

the auction algorithm terminates with an optimal solution.

Proof. Let j_i be the project assigned to person i when the algorithm terminates. Using ϵ-complementary slackness, we have

$$p_{j_i} - c_{ij_i} \geq \max_j\{p_j - c_{ij}\} - \epsilon, \qquad \forall\ i.$$

By adding these inequalities over all i, and rearranging, we obtain

$$\begin{aligned}\sum_{i=1}^n c_{ij_i} &\leq \sum_{i=1}^n \Big(p_{j_i} - \max_j\{p_j - c_{ij}\}\Big) + n\epsilon \\ &= \sum_{i=1}^n \Big(p_{j_i} + \min_j\{c_{ij} - p_j\}\Big) + n\epsilon.\end{aligned}$$

Let z be the cost of an optimal assignment. The sum in the right-hand side of the above equation is the same as the dual objective function [cf. Eq. (7.18)] and by weak duality, it is bounded above by the optimal cost z. This implies that

$$\sum_{i=1}^n c_{ij_i} \leq z + n\epsilon < z + 1.$$

On the other hand

$$\sum_{i=1}^{n} c_{ij_i} \geq z,$$

by the definition of z. Since z and all c_{ij_i} are integer, we conclude that

$$\sum_{i=1}^{n} c_{ij_i} = z,$$

and optimality has been established. □

Discussion

Let us assume, for simplicity, that $c_{ij} \geq 0$ for all i, j, and let $C = \max_{i,j} c_{ij}$. Suppose that the algorithm is initialized with all projects having the same prices. If some project has received C/ϵ or more bids, then its price is lower than the price of any project that has not received any bids, by at least C. (This is because each bid lowers the price by at least ϵ.) At that point, a project that has not received any bids would become more profitable. We conclude that every project receives at most C/ϵ bids. The total number of bids is at most nC/ϵ. Since there is at least one bid at each iteration, this is also a bound on the number of iterations. Finally, the computational effort per iteration is easily seen to be $O(n^2)$. If we let ϵ be slightly smaller than $1/n$, the version of the auction algorithm that we have described here runs in time $O(n^4 C)$.

The auction algorithm can be sped up using the idea of *ϵ-scaling*. One first uses a relatively large value of ϵ, and obtains a solution which is optimal within $n\epsilon$. (The proof is the same as the proof of Theorem 7.16.) Then, the obtained prices are used to start another solution phase, with a smaller value of ϵ, etc. This device leads to better theoretical running time estimates and also to improved performance in practice.

7.9 The shortest path problem

The shortest path problem is an important problem that arises in a multitude of applications in transportation networks, communication networks, optimal control, as well as a subproblem of more complex problems. As will be seen shortly, it can be posed as a network flow problem. However, practical methods for solving the shortest path problem do not rely on the network flow formulation. Instead, they are centered around a set of optimality conditions, known as Bellman's equation, which are intimately related to the subject of dynamic programming (see Section 11.3). We will use duality to derive Bellman's equation, and we will then proceed to develop a suite of algorithms. Some of these algorithms are of a somewhat ad hoc nature, but they are quite efficient in practice.

Throughout this section, the words walk, path, and cycle will always mean *directed* walk, path, and cycle, respectively; that is, all arcs are traversed in the forward direction. This should not lead to any confusion, because in this section we never need to consider walks, paths, or cycles that are not directed.

Formulation

We are given a directed graph $G = (\mathcal{N}, \mathcal{A})$ with n nodes and m arcs. For each arc $(i,j) \in \mathcal{A}$, we are also given a cost or *length* c_{ij}; in general, the numbers c_{ij} are allowed to be negative. The *length* of a walk, path, or cycle is defined as the sum of the lengths of its arcs. A path from a certain node to another is said to be *shortest* if it has minimum length among all possible paths with the same origin and destination. A *shortest walk* from a node to another is defined similarly. A shortest walk and a shortest path from one node to another are not necessarily the same. In particular, if there exists a cycle with negative length, we can construct walks whose length converges to $-\infty$ (we can traverse the cycle several times before reaching the destination). On the other hand, the length of any path is bounded below by $-nC$, where $C = \max_{(i,j)\in\mathcal{A}} |c_{ij}|$. If all cycles have nonnegative length, there is no incentive to go around a cycle and, for this reason, a shortest path is also a shortest walk. Conversely, any cycles contained in a shortest walk must have zero length; by removing such cycles, we obtain a shortest path.

The shortest path problem can be posed in a few different ways; for example, we might be interested in a shortest path from a given origin to a given destination, or we might be interested in shortest paths from each of a number of selected origins to each of several destinations. We will focus on the problem of finding a shortest path from all possible origins to a particular destination node, which is called the *all-to-one shortest path* problem, as well as on the problem of finding shortest paths for all possible origin destination pairs, which is called the *all-pairs shortest path* problem.

Before continuing, we introduce two more concepts that will prove useful. Consider a tree, and suppose that all arcs are assigned directions so that we have a (directed) path from every node $i \neq n$ to node n. Such a directed graph will be called an *intree rooted at node n*; see Figure 7.31. If it happens that for every $i \neq n$, the path from i to n along the tree is a shortest path, we say that we have a *tree of shortest paths.*

Relation to the network flow problem

We consider here the all-to-one shortest path problem. For concreteness, we assume that node n is the destination node. We also assume that there exists at least one path from every node $i \neq n$ to node n, which means that the all-to-one shortest path problem is feasible. Finally, and without loss of

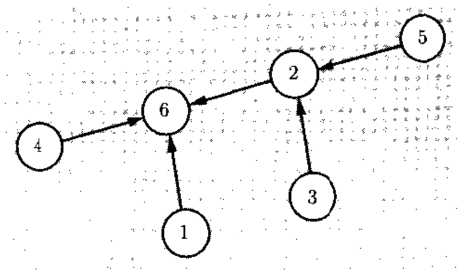


Figure 7.31: An intree rooted at node 6.

generality, we assume that there are no outgoing arcs from node n. These assumptions will remain in effect throughout this section.

We view the graph G as a network of infinite capacity arcs. Suppose that each one of the nodes $1, \ldots, n-1$ is a source node, with unit supply, and that node n is the only sink node, with a demand of $n-1$. If we pose the problem of minimizing $\sum_{(i,j)\in\mathcal{A}} c_{ij} f_{ij}$ over all feasible flow vectors, it should be clear that for every node i other than n, one unit of flow should be shipped from node i to node n, at least cost. As long as there are no negative length cycles, this should be done along a shortest path. If on the other hand, there are negative length cycles, the optimal cost in the network flow problem is $-\infty$, because we could "push" an arbitrarily large amount of flow around such a cycle. This discussion is refined in the following theorem.

Theorem 7.17 *Consider the shortest path problem in a directed graph with n nodes and the associated network flow problem. We assume that there is a path to node n from every other node, and that node n has no outgoing arcs.*

(a) *If there exists a negative length cycle, the optimal cost in the network flow problem is $-\infty$.*

(b) *Suppose that all cycles have nonnegative length. If a feasible tree solution is optimal, then the corresponding tree is a tree of shortest paths.*

(c) *Suppose that all cycles have nonnegative length. If we fix p_n to zero, the dual problem has a unique solution $\mathbf{p}^*$, and p_i^* is the shortest path length from node i.*

Proof. Part (a) is trivial. For part (b), we first note that in a feasible tree solution, all arcs in the tree must be oriented from the leaves towards the root and therefore form an intree rooted at node n. This is because if some

arc (i,j) in the tree is pointing away from the root, the flow on that arc must be negative, contradicting feasibility. If for some node i there exists a path from i to n whose length is smaller than that of the path on the tree, the feasible tree solution is not optimal, because some flow could be redirected to that path. Thus, an optimal feasible tree solution provides us with a tree of shortest paths and this proves part (b).

Let $p_n^* = 0$ and let $(p_1^*, \ldots, p_{n-1}^*)$ be the vector of dual variables associated with an optimal feasible tree solution. For each arc on the tree, we have $p_i^* = c_{ij} + p_j^*$. Since all arcs are oriented towards the root, we can add the equalities $p_i^* = c_{ij} + p_j^*$ along a path contained in the tree, and conclude that p_i^* is the length of the path from node i to node n. Note that this is a shortest path, since we are dealing with an optimal feasible tree solution. Thus, p_i^* is the shortest path length.

We finally note that every feasible tree solution is nondegenerate. This is because any arc (i,j) on the tree must carry the supply at node i. It follows that the dual problem has a unique solution. □

The connection between shortest paths, network flows, and linear programming duality is illustrated by our next example, which arises in practical context.

Example 7.11 (Project management) A project consists of a set of jobs and a set of *precedence relations* In particular, we are given a set $\mathcal{A}$ of job pairs (i,j) indicating that job i cannot start before job j is completed. Let c_i be the duration of job i. We wish to identify the least possible duration of the project. We will show that this can be accomplished by solving a shortest path problem.

In addition to the original jobs, we introduce two artificial jobs s and t, of zero duration, that signify the beginning and the completion of the project. We augment the set $\mathcal{A}$ by introducing the additional precedence relations (s,i) and (i,t) for all jobs i. Let p_i be the time that job i begins. A precedence relation $(i,j) \in \mathcal{A}$ leads to a constraint $p_j \geq p_i + c_i$, that is, project j cannot begin before the completion time $p_i + c_i$ of project i. The project duration is $p_t - p_s$ and the minimal project duration is obtained by solving the following problem:

$$\begin{array}{lll} \text{minimize} & p_t - p_s & \\ \text{subject to} & p_j - p_i \geq c_i, & \forall\ (i,j) \in \mathcal{A}. \end{array}$$

The dual of this problem is

$$\begin{array}{llr} \text{maximize} & \displaystyle\sum_{(i,j)\in\mathcal{A}} c_i f_{ij} & \\ \text{subject to} & \displaystyle\sum_{\{j\,|\,(j,i)\in\mathcal{A}\}} f_{ji} - \sum_{\{j\,|\,(i,j)\in\mathcal{A}\}} f_{ij} = b_i, & \forall\ i, \\ & f_{ij} \geq 0, & \forall\ (i,j) \in \mathcal{A}. \end{array}$$

Here, $b_s = -1$, $b_t = 1$, and $b_i = 0$ for $i \neq s,t$. This is a shortest path problem, where each precedence relation $(i,j) \in \mathcal{A}$ corresponds to an arc with cost of $-c_i$. It is natural to assume that the set of arcs $\mathcal{A}$ does not contain any cycles, because otherwise the project cannot be completed. In that case, the network is guaranteed to have no negative cost cycles.

Bellman's equation

Recall that $b_1 = \cdots = b_{n-1} = 1$. Under the convention $p_n = 0$, the dual problem is of the form

$$\begin{array}{ll} \text{maximize} & \sum_{i=1}^{n-1} p_i \\ \text{subject to} & p_i \le c_{ij} + p_j, \qquad \forall\ (i,j) \in \mathcal{A}. \end{array}$$

It is evident that if all components of $\mathbf{p}$, except for p_i, are fixed to some values, the remaining component p_i should be set to the largest value allowed by the constraints, that is, $\min_{k\in O(i)}\{c_{ik} + p_k\}$. [Recall that $O(i)$ is the set of endpoints of arcs that are outgoing from node i.] We conclude that the optimal solution $\mathbf{p}^*$ to the dual problem, which is the same as the vector of shortest path lengths, satisfies

$$p_i^* = \min_{k\in O(i)} \{c_{ik} + p_k^*\}, \qquad i = 1,\ldots,n-1, \tag{7.20}$$

where $p_n^* = 0$. This is a system of $n-1$ nonlinear equations in $n-1$ unknowns, and is known as *Bellman's equation*. It has a rather intuitive interpretation: suppose that we are interested in paths that start at node i, but that we also impose the additional constraint that the path must start with the arc (i,k). Then, the best we can do is to find a shortest path from node k to n, for a total length of $c_{ik} + p_k^*$. However, since the first node k is of our own choosing, we should make an optimal choice of k, and therefore the length of a shortest path is $\min_{k\in O(i)}\{c_{ik} + p_k^*\}$. The key idea behind Bellman's equation is the so-called *principle of optimality*: if a shortest path from i to n goes through an intermediate node k, then the portion of the path from k to n is also a shortest path.

We have argued that the shortest path lengths satisfy Bellman's equation. Thus, one possible method of computing shortest path distances is by trying to solve Bellman's equation directly. However, some care is needed, because Bellman's equation may have several solutions, and only one of them will give us the correct shortest path lengths; an example is given in Figure 7.32. It turns out that the shortest path lengths are the unique solution to Bellman's equation if all cycles have positive lengths. If all cycles have nonnegative length, we can only assert that the shortest path lengths are the *largest* solution to Bellman's equation (Exercise 7.33).

The Bellman-Ford algorithm

A common method for solving a system of equations of the form $\mathbf{x} = \mathbf{F}(\mathbf{x})$ is to use the iteration $\mathbf{x} := \mathbf{F}(\mathbf{x})$. If we attempt to solve Bellman's equation

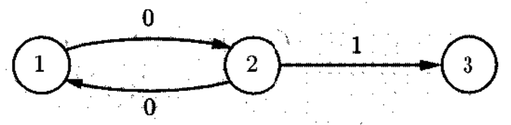


Figure 7.32: Consider a graph with three nodes and let the arc lengths be as indicated. The shortest path lengths to node 3 are $p_1^* = p_2^* = 1$. Bellman's equation is of the form $p_1 = p_2$ and $p_2 = \min\{p_1, 1\}$. It is easily seen that $p_1 = p_2 = \beta$ is a solution to Bellman's equation for every $\beta \le 1$. Note that the shortest path lengths are the largest solution to Bellman's equation.

in this fashion, we obtain the Bellman-Ford algorithm. In the discussion that follows, we again assume that node n has no outgoing arcs.

Let $p_i(t)$ be the length of a shortest walk from node i to node n that uses at most t arcs; we let $p_i(t) = \infty$ if no such walk exists. We use the convention $p_n(t) = 0$ for all t, and $p_i(0) = \infty$ for all $i \ne n$. Note that $p_i(t+1) \le p_i(t)$ for all i and t, because as t increases, there are more walks to choose from. A shortest walk from node i to node n that uses at most $t+1$ arcs, consists of an initial arc (i,k) and a walk from node k to node n that consists of at most t arcs. Of course, the latter walk should be chosen as short as possible and its length is therefore $p_k(t)$. Since node k should also be chosen in the most profitable fashion, we have

$$p_i(t+1) = \min_{k\in O(i)} \{c_{ik} + p_k(t)\}, \qquad i = 1,\ldots,n-1,$$

and this equation defines the Bellman-Ford algorithm. We now discuss the termination properties of the algorithm.

(a) Suppose that there are no negative length cycles. Then, there exists a shortest walk, which is also a shortest path, and has at most $n-1$ arcs. In particular, $p_i(n-1) = p_i^*$. Allowing for a walk with n or more arcs cannot reduce the total length, and we have $p_i(n) = p_i(n-1)$ for all nodes.

(b) Suppose that there exists a negative length cycle. Suppose for a moment, that we also have $\mathbf{p}(n) = \mathbf{p}(n-1)$. This implies that $\mathbf{p}(t) = \mathbf{p}(n)$ for all $t \ge n$ and the length of any walk is bounded below. However, in the presence of negative length cycles, there exist walks whose length tends to $-\infty$. This is a contradiction and proves that $\mathbf{p}(n) \ne \mathbf{p}(n-1)$.

By comparing the two cases just discussed, we see that no more than n iterations are needed. If $\mathbf{p}(n) = \mathbf{p}(n-1)$, then $\mathbf{p}(n)$ is the vector of shortest path lengths. (An example is given in Figure 7.33.) If on the other hand $\mathbf{p}(n) \ne \mathbf{p}(n-1)$, we conclude that there exists a negative length cycle.

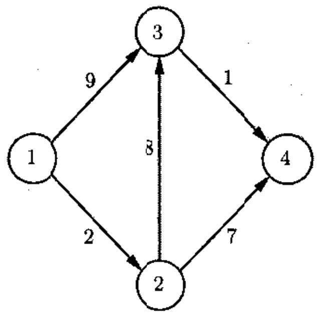


Figure 7.33: We apply the Bellman-Ford algorithm to the graph shown. Node 4 is the destination node. We have $p_4(t) = 0$ for all t, and

$$\begin{array}{llll} p_1(0) = \infty, & p_1(1) = \infty, & p_1(2) = 9, & p_1(3) = 9, \\ p_2(0) = \infty, & p_2(1) = 7, & p_2(2) = 7, & p_2(3) = 7, \\ p_3(0) = \infty, & p_3(1) = 1, & p_3(2) = 1, & p_3(3) = 1. \end{array}$$

We observe that $\mathbf{p}(3) = \mathbf{p}(2)$ and, therefore, $\mathbf{p}(2)$ is equal to the shortest path length vector $\mathbf{p}^*$.

The computational complexity of the algorithm is $O(mn)$ becau
there are at most n iterations and at each iteration, each arc is only exa
ined once.

We have focused so far on the computation of the shortest pat
lengths rather than the shortest paths. The reason is that once the shorte
path lengths are available, shortest paths can be determined fairly easil
(Exercise 7.34). The task of finding shortest paths is made even easier if i
the course of the algorithm, we maintain some information that allows u
to backtrack and recover a shortest path. This is done as follows. For ever
node i, we keep a record of a *successor* node $s(i)$, chosen as the first nod
in a path whose total length is equal to the current estimate $p_i(t)$ availab
at node i. Determining a successor node with such a property is simpl
whenever we have $p_i(t+1) < p_i(t)$, we delete the old successor of i, if an
and let $s(i)$ be such that $p_i(t+1) = c_{is(i)} + p_{s(i)}(t)$.

As noted earlier, the Bellman-Ford algorithm provides us with
method for checking whether there are any negative length cycles. Be
sides detecting the existence of a negative length cycle, some applicatio
such as the negative cost cycle algorithm of Section 7.4, require the co
struction of a negative length cycle. This can be accomplished as follow
Consider a node i for which $p_i(n) < p_i(n-1)$. By starting at node i an
going from each node to its successor, we obtain a walk with n arcs who
length is $p_i(n)$. Since there are only n nodes in the graph, this walk mu
contain a cycle. Suppose that this cycle has nonnegative length. Let u

delete the arcs on the cycle and we are left with a walk with fewer than n arcs whose length is no greater than $p_i(n)$. This contradicts the inequality $p_i(n) < p_i(n-1)$. We conclude that by tracing the successors of node i, we will discover a negative length cycle.

Label correcting methods

Label correcting methods are a general class of shortest path algorithms, that have proved to be very efficient in practice. They are similar in spirit to the Bellman-Ford algorithm, but they are more flexible, hence the potential for improved performance.

The key idea is to maintain at each node j, a label p_j equal to the length of the shortest walk from j to n discovered thus far. Given a walk from j to n, of length p_j, there exists a walk from i to n of length $c_{ij} + p_j$. Thus, each time that p_j is revised downwards ("corrected"), we also have an opportunity to revise downwards the labels of all nodes i that have an outgoing arc to node j (the *predecessors* of j). The algorithm maintains a *list* S of all nodes whose labels have been revised downwards, and such that the revision has not yet been propagated to their predecessors. (The list S plays a role similar to the list of labeled but not yet scanned nodes in the labeling algorithm of Section 7.5.)

Label correcting algorithm

The algorithm is initialized with $S = \{n\}$, $p_n = 0$, and $p_i = \infty$ for every $i \neq n$. A typical iteration is as follows.

1. Remove a node j from S.
2. For every node $i \neq n$ such that (i,j) is an arc, do the following. Let $p_i := \min\{p_i, c_{ij} + p_j\}$. If the new value of p_i is smaller, add node i to the set S.
3. If S is empty, the algorithm terminates. Otherwise, go back to Step 1.

The label of a node is always equal to the length of some walk to node n. (Except when the label is infinite, indicating that a path has not yet been discovered.) This is easily shown by induction. Indeed, assuming this to be true before an update, the new label $\min\{p_i, c_{ij} + p_j\}$ is either equal to the length p_i of a previously identified walk, or is equal to the length $c_{ij} + p_j$ of a walk that starts with arc (i,j) and follows a previously identified walk from j to n.

We now establish the finite termination of the algorithm. We assume that all cycles have nonnegative length. Let p_i^0 be the first finite label assigned to node i. Any walk from i to n whose length is less than p_i^0, consists of a path from i to n, an arbitrary number of zero length cycles, and a bounded number of positive length cycles. Since zero length cycles have no effect on the length of the walk, the possible values of p_i that are smaller than p_i^0, are finitely many. This implies that there can only be finitely many downward revisions of each label. After some point, there will be no more revisions, and each iteration will only result in the removal of some node from S. It follows that S eventually becomes empty and the algorithm terminates.

We conclude our analysis, by analyzing the correctness of the algorithm.

Theorem 7.18 *Suppose that there exists a path from every node to node n, and that all cycles have nonnegative length. Then, the label correcting algorithm eventually terminates with the label p_i of each node equal to the shortest path length p_i^*.*

Proof. Consider a shortest path $i_1, i_2, \ldots, i_t = n$ from some node i_1 to n. By the definition of the algorithm, we have $p_n = 0 = p_n^*$, at all times. At the first iteration of the algorithm, we have $S = \{n\}$, the predecessors of n are examined, and we set $p_{i_{t-1}} = c_{i_{t-1}n}$, which is equal to $p_{i_{t-1}}^*$. (This is because the last arc of a shortest path is itself a shortest path.)

Consider now an intermediate node i_k in the path, and suppose that the final label p_{i_k} is equal to $p_{i_k}^*$. Since p_{i_k} was initially infinite, its label has changed at least once. The last time that p_{i_k} was changed, and was set to $p_{i_k}^*$, node i_k entered the set S. When at some later iteration, i_k exited S, $p_{i_{k-1}}$ was set to $\min\{p_{i_{k-1}}, c_{i_{k-1}i_k} + p_{i_k}^*\}$. This is less than or equal to $c_{i_{k-1}i_k} + p_{i_k}^* = p_{i_{k-1}}^*$. On the other hand, $p_{i_{k-1}}$ is the length of some walk, and can be no smaller than $p_{i_{k-1}}^*$. We have therefore completed an inductive proof that $p_{i_k} = p_{i_k}^*$ for all k. □

The practical efficiency of label correcting methods is highly dependent on the rule used to select a node from the list S. It is interesting to note that for certain rules, including some that have been very successful in practice, the worst-case complexity is exponential in n. The reader is referred to the literature for a more detailed discussion.

Dijkstra's algorithm

Dijkstra's algorithm is an alternative to the Bellman-Ford algorithm and label correcting methods. We will see shortly that Dijkstra's algorithm is more efficient, but can only be applied if all arc lengths are nonnegative, which will be assumed throughout this section. The key idea in Dijkstra's algorithm is to identify the nodes in the order of the corresponding shortest path lengths, starting with a node for which the shortest path length is smallest. In order to simplify the presentation, we assume that c_{ij} is defined

for every pair (i, j) of distinct nodes (with $i \neq n$), but may be equal to infinity for some pairs.

Our first step is to show that a node ℓ with a smallest shortest path length is easy to find. Nonnegativity of the arc lengths is crucial here.

Theorem 7.19 *Suppose that $c_{ij} \geq 0$ for all i, j. Let $\ell \neq n$ be such that*

$$c_{\ell n} = \min_{i \neq n} c_{in}.$$

Then, $p_\ell^* = c_{\ell n}$ and $p_\ell^* \leq p_k^*$ for all $k \neq n$.

Proof. Any path to node n has a last arc (i, n) whose length c_{in} is at least $c_{\ell n}$. Thus, $p_k^* \geq c_{\ell n}$ for all $k \neq n$. For node ℓ, we also have $p_\ell^* \leq c_{\ell n}$. We conclude that $p_\ell^* = c_{\ell n} \leq p_k^*$ for all $k \neq n$. □

Suppose that ℓ and p_ℓ^* have been determined as in Theorem 7.19, and consider an arbitrary node i. One of the options available at that node is to traverse the arc (i, ℓ) and visit node ℓ. Once at node ℓ, we should traverse arc (ℓ, n), because this is a shortest path from ℓ to n. Thus, once an arc (i, ℓ) is traversed, the traversal of arc (ℓ, n) can be assumed to be automatic. We can therefore replace the two arcs (i, ℓ) and (ℓ, n) by a single arc $(i, n)'$ of length $c_{i\ell} + c_{\ell n}$; once we do that for every $i \neq \ell, n$, node ℓ can be taken out of the picture. Note that a node i may now have two direct arcs to node n, the original arc (i, n) as well as the new artificial arc $(i, n)'$. Naturally, any shortest path would only use the least expensive of the two. We therefore remove $(i, n)'$ and replace c_{in} by

$$\min\{c_{in}, c_{i\ell} + c_{\ell n}\}.$$

We are left with a new shortest path problem with one node less. We apply the same process to the new shortest path problem. Each iteration evaluates the shortest path length for one more node and, therefore, after $n - 1$ iterations, the algorithm terminates.

The resulting algorithm is summarized next.

Dijkstra's algorithm

1. Find a node $\ell \neq n$ such that $c_{\ell n} \leq c_{in}$ for all $i \neq n$. Set $p_\ell^* = c_{\ell n}$.
2. For every node $i \neq \ell, n$, set
$$c_{in} := \min\{c_{in}, c_{i\ell} + c_{\ell n}\}.$$
3. Remove node ℓ from the graph and apply the same steps to the new graph.

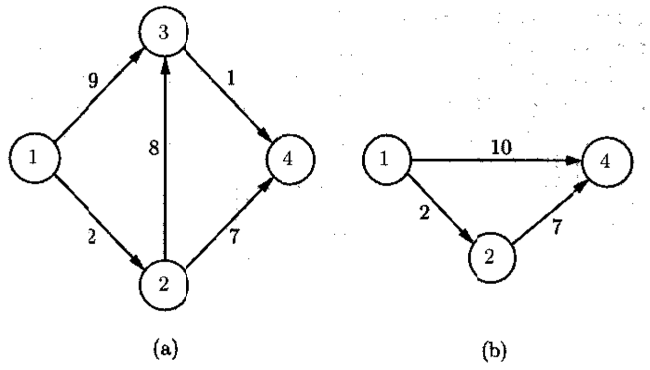


Figure 7.34: (a) A graph with arc lengths. The arcs that are not shown have infinite length. (b) The graph obtained after one iteration of Dijkstra's algorithm.

Example 7.12 We apply Dijkstra's algorithm to graph shown in Figure 7.34(a), with node $n = 4$ being the destination node. We have $\ell = 3$ and $p_3^* = 1$. The following arc lengths are modified: $c_{14} := \min\{\infty\ 9 + 1\} = 10$ and $c_{24} := \min\{7, 8 + 1\} = 7$. We now eliminate node 3 and obtain the graph shown in Figure 7.34(b). We obtain $\ell = 2$ and $p_2^* = 7$. The arc length c_{14} is modified by $c_{14} = \min\{10, 2 + 7\} = 9$. Node 2 is eliminated. Since node 1 is the only nonterminal node left, p_1^* is equal to the current value of c_{14}, which is 9.

We now estimate the computational complexity of the Dijkstra algorithm. A typical iteration starts by comparing the coefficients c_{in} and this takes $O(n)$ time. Having determined ℓ, we need to update c_{in} for each node i. We conclude that there are only $O(n)$ arithmetic operations per iteration. The overall complexity is $O(n^2)$, which is one order of magnitude better than the Bellman-Ford algorithm. For a dense graph with $\Omega(n^2)$ arcs, any shortest path algorithm needs $\Omega(n^2)$ arithmetic operations because, in general, every arc has to be examined at least once. Thus, for dense graphs, Dijkstra's algorithm is the best possible.

For sparse graphs, that is, when m is much smaller than n, the computational complexity of Dijkstra's algorithm can be brought down to $O(m \log n)$. Doing so requires keeping the coefficients c_{in} in a suitable data structure that allows us to obtain the smallest such coefficient with minimal work.

Reduction to the case of nonnegative arc lengths and the all-pairs problem

Suppose that some of the arc lengths are negative, but that all cycles have nonnegative length. Let p_i^* be the shortest path length from node i to node

n. From Bellman's equation, we have

$$p_i^* \leq c_{ij} + p_j^*, \tag{7.21}$$

for all arcs (i, j). Let us now construct a new shortest path problem in which the arc lengths c_{ij} are replaced by new arc lengths $\overline{c}_{ij}$, defined by

$$\overline{c}_{ij} = c_{ij} + p_j^* - p_i^*.$$

Using Eq. (7.21), we have $\overline{c}_{ij} \geq 0$ for all $(i, j) \in \mathcal{A}$. Under the new arc lengths, the length of any path $i_1, \ldots, i_t$ from some node i_1 to some other node i_t is given by

$$\sum_{\tau=1}^{t-1} \overline{c}_{i_\tau i_{\tau+1}} = \sum_{\tau=1}^{t-1} (c_{i_\tau i_{\tau+1}} + p_{i_{\tau+1}}^* - p_{i_\tau}^*) = p_{i_t}^* - p_{i_1}^* + \sum_{\tau=1}^{t-1} c_{i_\tau i_{\tau+1}}.$$

In particular, for any given pair of nodes, a shortest path under the new arc lengths is a shortest path under the old arc lengths, and conversely. Since the new arc lengths are nonnegative, we are in a position to apply Dijkstra's algorithm.

If we are only interested in a single destination, the transformation that we have just described is of no particular use. On the other hand, if we are interested in the all-pairs problem, we can solve a single all-to-one problem, using the Bellman-Ford algorithm, transform the arc lengths, and finally solve $n - 1$ additional all-to-one problems (one problem for every possible destination) using Dijkstra's algorithm. The overall complexity is $O(n^3) + (n - 1) \cdot O(n^2) = O(n^3)$. This is much better than applying the Bellman-Ford algorithm n times, which would require $O(n^4)$ time. For sparse graphs, the running time can be brought down to $O(nm \log n)$ by using an efficient implementation of Dijkstra's algorithm. An alternative $O(n^3)$ algorithm for the all-pairs problem is developed in Exercise 7.38.

7.10 The minimum spanning tree problem

We are given a connected *undirected* graph $G = (\mathcal{N}, \mathcal{E})$, with n nodes. For each edge $e \in \mathcal{E}$, we are also given a cost coefficient c_e. (Recall that an edge in an undirected graph is an unordered pair $e = \{i, j\}$ of distinct nodes in $\mathcal{N}$.) A *minimum spanning tree* (MST) is defined as a spanning tree such that the sum of the costs of its edges is as small as possible.

The minimum spanning tree problem arises naturally in many applications. For example, if edges correspond to communication links, a spanning tree is a set of links that allows every node to communicate (possibly, indirectly) to every other node. Then, a minimum spanning tree is a communication network that provides this type of connectivity, and whose cost is the smallest possible. The minimum spanning tree problem

also arises as a subproblem of more complex, seemingly unrelated, pro
lems. An example will be seen in Section 11.5, where it forms a basis fo
heuristic for the traveling salesman problem.

Even though the MST problem is not a network flow problem,
include it in this chapter, because of its graph-theoretic structure. We w
see that it can be solved by means of a simple *greedy* algorithm. A gree
algorithm is one consisting of a sequence of choices that appear to be b
in the short run. For certain problems, like the MST, short run optim
decisions turn out to be optimal in the long run as well. The algorit
that we describe builds an MST by progressively adding edges to a curre
tree. At any stage, we have a tree and we add a least expensive edge th
connects a node in the tree with a node outside the tree.

Greedy algorithm for the minimum spanning tree problem

1. The input to the algorithm is a connected undirected graph G
$(\mathcal{N},\mathcal{E})$ and a coefficient c_e for each edge $e \in \mathcal{E}$. The algorith
is initialized with a tree $(\mathcal{N}_1,\mathcal{E}_1)$ that has a single node and
edges ($\mathcal{E}_1$ is empty).
2. Once $(\mathcal{N}_k,\mathcal{E}_k)$ is available, and if $k < n$, we consider all edg
$\{i,j\} \in \mathcal{E}$ such that $i \in \mathcal{N}_k$ and $j \notin \mathcal{N}_k$. Choose an edge e^*
$\{i,j\}$ of this type whose cost is smallest. Let
$$\mathcal{N}_{k+1} = \mathcal{N}_k \cup \{j\}, \qquad \mathcal{E}_{k+1} = \mathcal{E}_k \cup \{e^*\}.$$

Since at each stage we connect a node in the current tree with a nod
outside the tree, no cycles are ever formed, and we always have a tre
The set $\mathcal{N}_n$ has n elements and, therefore, $(\mathcal{N}_n,\mathcal{E}_n)$ is a spanning tree.
remains to show that it is a minimum spanning tree. This is accomplishe
by showing a somewhat stronger property.

Theorem 7.20 *For $k = 1,\ldots,n$, the tree $(\mathcal{N}_k,\mathcal{E}_k)$ is part of some MST. That is, there exists an MST $(\mathcal{N},\overline{\mathcal{E}}_k)$ such that $\mathcal{E}_k \subset \overline{\mathcal{E}}_k$.*

Proof. The proof uses induction on k. The result is trivially true for
$k=1$, because the empty set $\mathcal{E}_1$ is a subset of the edge set of any spanning
tree.

Suppose now that $k < n$, and that $\mathcal{E}_k$ is a subset of some MST $\overline{\mathcal{E}}_k$.
[We are slightly abusing terminology by referring to $\overline{\mathcal{E}}_k$ instead of $(\mathcal{N},\overline{\mathcal{E}}_k)$
as a spanning tree.] Let $e^* = \{i,j\}$ be the edge added to $\mathcal{E}_k$; that is, $i \in \mathcal{N}_k$
$j \notin \mathcal{N}_k$, and $\mathcal{E}_{k+1} = \mathcal{E}_k \cup \{e^*\}$. If $e^* \in \overline{\mathcal{E}}_k$, then $\mathcal{E}_{k+1}$ is also a subset o
$\overline{\mathcal{E}}_k$, and the induction hypothesis is verified for $k+1$, with $\overline{\mathcal{E}}_{k+1} = \overline{\mathcal{E}}_k$.

Suppose now that $e^* \notin \overline{\mathcal{E}}_k$. Then, e^*, together with $\overline{\mathcal{E}}_k$, forms a unique cycle [Theorem 7.1(d)]. This cycle must contain a second edge (call it $\overline{e}$) with one endpoint in $\mathcal{N}_k$ and another outside $\mathcal{N}_k$; see Figure 7.35. Since

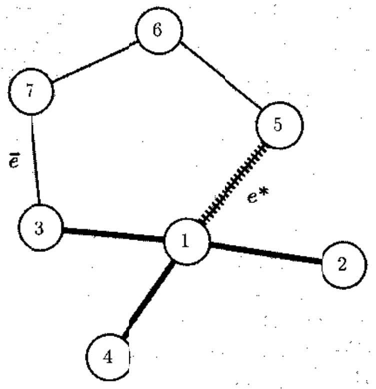


Figure 7.35: The thicker edges correspond to a tree $(\mathcal{N}_4,\mathcal{E}_4)$ involving 4 nodes. This is assumed to be part of an MST $\overline{\mathcal{E}}_4$, which consists of all edges shown, with the exception of e^*. If the algorithm selects e^*, its cost can be no greater than the cost of $\overline{e}$, and $\mathcal{E}_4 \cup \{e^*\}$ is part of an alternative MST, in which $\overline{e}$ is replaced by e^*.

the algorithm selected e^* rather than $\overline{e}$ to be added to $\mathcal{E}_k$, we must have $c_{e^*} \le c_{\overline{e}}$. Let us now take the MST $\overline{\mathcal{E}}_k$, delete edge $\overline{e}$, and replace it by e^*. We obtain a new spanning tree, call it $\overline{\mathcal{E}}_{k+1}$, and the cost change is $c_{e^*} - c_{\overline{e}} \le 0$. By the optimality of $\overline{\mathcal{E}}_k$, we must have $c_{e^*} = c_{\overline{e}}$, and both spanning trees are optimal. We now note that $\mathcal{E}_{k-1}$ is a subset of the MST $\overline{\mathcal{E}}_{k+1}$, and the induction is complete. □

Having proved the correctness of the algorithm, we now discuss its computational complexity. We have $n-1$ iterations. At each iteration, we need to examine each edge to see whether it is eligible for becoming part of the tree, and we then need to find the least expensive one, which can all be done in time $O(n^2)$. Thus, the overall complexity is $O(n^3)$. With a more clever implementation, it can be brought down to $O(n^2)$; see Exercise 7.39.

7.11 Summary

In this chapter, we provided an overview of a broad range of topics related to network flow problems, and we have covered most of the major available methodologies.

Network flow problems are special cases of linear programming problems, and can be solved by applying general purpose methods, suitably tuned to exploit the network structure. For example, the primal or the dual simplex method can be used. As we have pointed out, the underlying network structure allows for simple and efficient rules for updating the basic variables and the reduced costs. In addition, when the problem data are integer, integer arithmetic can also be employed.

An important property of network flow problems that we discovered in the course of our development, relates to integrality of basic solutions. Assuming that problem data are integer, we have shown that basic solutions to the primal and the dual have integer coordinates. The key reason behind this property is that the determinant of any basis matrix **B** has unit magnitude. Unfortunately, there are only precious few classes of linear programming problems that have such remarkable properties.

Besides fine tuning the simplex method, we also developed some algorithms that are specially tailored to network flow problems. These include the negative cost cycle algorithm of Section 7.4 and the dual ascent methods of Section 7.7. These two methods are dual to each other in many ways that can be made mathematically precise, but which are beyond our scope. Nevertheless, it is important to point out a common feature. In both methods, a direction of improvement is identified by examining only a finite number of possible directions, which are independent of the numerical values of the input data. (In the negative cost cycle algorithm, the directions considered correspond to simple circulations. In dual ascent methods, the directions considered correspond to subsets of the set of nodes.)

Both the negative cost cycle algorithm and the dual ascent methods can be described at a high level of generality, while leaving a lot of freedom on how to choose a cycle or a dual ascent direction. By making some more specific choices, the worst-case number of iterations can be reduced. Furthermore, the search for a direction of cost improvement, carried out in the course of each iteration, usually has a lot of room for increased efficiency. (An example of this is our development of the primal-dual method, where the search for an ascent direction is implemented by means of an auxiliary maximum flow problem and the labeling algorithm.) Such refinements lead to improved worst-case complexity bounds. It should be kept in mind, however, that worst-case complexity bounds may not accurately reflect the performance of an algorithm in practice.

The network flow problem contains some important special cases that can be solved by suitable special purpose algorithms. We saw the Ford-Fulkerson algorithm for the maximum flow problem, the auction algorithm for the assignment problem, and a number of (somewhat ad hoc) methods for the shortest path problem. Auction algorithms can also be developed for the general network flow problem, but this is a direction that we did not pursue.

The minimum spanning tree problem is somewhat disjoint from the rest of the chapter. It was included because of its importance, and also because it shares an underlying graph-theoretic structure.

7.12 Exercises

Exercise 7.1 (The caterer problem) A catering company must provide to a client r_i tablecloths on each of N consecutive days. The catering company can buy new tablecloths at a price of p dollars each, or launder the used ones. Laundering can be done at a fast service facility that makes the tablecloths unavailable for the next n days and costs f dollars per tablecloth, or at a slower facility that makes tablecloths unavailable for the next m days (with $m > n$) at a cost of g dollars per tablecloth ($g < f$). The caterer's problem is to decide how to meet the client's demand at minimum cost, starting with no tablecloths and under the assumption that any leftover tablecloths have no value.

(a) Show that the problem can be formulated as a network flow problem. *Hint:* Use a node corresponding to clean tablecloths and a node corresponding to dirty tablecloths for each day; more nodes may also be needed.

(b) Show explicitly the form of the network if $N = 5$, $n = 1$, $m = 3$.

Exercise 7.2 Consider a wood product company that owns M forest units and wants to find an optimal cutting schedule over a period of K years. Forest unit i is predicted to have a_{ij} tons of wood available for harvesting during period j. The company wants to meet a demand of d_j tons during year j. However, due to capacity limitations, it can only harvest up to u_j tons during that year. Wood harvested in past years can be stored and used to meet demand in subsequent years, but there is a cost of c_j for storing one ton of wood between year $j-1$ and j. We also assume that wood that is available but not harvested during a year remains available for harvesting in later years. Formulate the problem of determining a minimum cost harvesting schedule that meets the demand as a network flow problem.

Exercise 7.3 (The tournament problem) Each of n teams plays against every other team a total of k games. Assume that every game ends in a win or a loss (no draws) and let x_i be the number of wins of team i. Let X be the set of all possible outcome vectors $(x_1, \ldots, x_n)$. Given an arbitrary vector $(x_1, \ldots, x_n)$, we would like to determine whether it belongs to X, that is, whether it is a possible tournament outcome vector. Provide a network flow formulation of this problem.

Exercise 7.4 (Piecewise linear convex costs)

(a) Consider the capacitated network flow problem except that the cost at each arc is a piecewise linear convex function of the flow on that arc. Show that the problem can be reduced to one with linear costs, but in which we allow multiple arcs with the same start node and end node.

(b) Show that a capacitated problem in which we have multiple arcs with the same start node and end node can be reduced to a problem without any such multiple arcs.

Exercise 7.5 (Equivalence of uncapacitated network flow and transportation problems) Consider an uncapacitated network flow problem and assume that $c_{ij} \geq 0$ for all arcs. Let S_+ and S_- be the sets of source and sink nodes, respectively. Let d_{ij} be the length of a shortest directed path from node $i \in S_+$ to node $j \in S_-$. We construct a transportation problem with the same source and sink nodes, and the same values for the supplies and the demands. For every source node i and every sink node j, we introduce a direct link with cost d_{ij}. Show that the two problems have the same optimal cost.

Exercise 7.6 (Equivalence of capacitated network flow and transporta tion problems) Consider a capacitated network flow problem defined by a grap $G = (\mathcal{N}, \mathcal{A})$ and the data u_{ij}, c_{ij}, b_i. Assume that the capacity u_{ij} of every ar is finite. We construct a related transportation problem as follows. For every ar $(i, j) \in \mathcal{A}$, we form a source node in the transportation problem with supply u_{ij} For every node $i \in \mathcal{N}$, we construct a sink node with demand $\sum_{\{k|(i,k)\in\mathcal{A}\}} u_{ik} - b_i$ At every supply node (i, j) there are two outgoing infinite capacity arcs: one go to demand node i, and its cost coefficient is 0; the other goes to demand node and its cost coefficient is c_{ij}. See Figure 7.36 for an illustration.

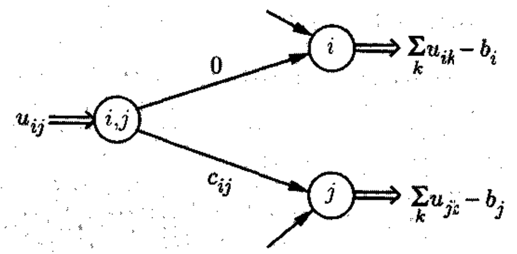


Figure 7.36: The transportation problem in Exercise 7.6.

Show that that there is a one-to-one correspondence between feasible flo in the two problems and that the cost of the two corresponding flows is the sam

Exercise 7.7 (Lower bounds on arc flows) Consider a network flow pro lem in which we impose an additional constraint $f_{ij} \geq d_{ij}$ for every arc $(i, j$ Construct an equivalent network flow problem in which there are no nonze lower bounds on the arc costs. *Hint:* Let $\overline{f}_{ij} = f_{ij} - d_{ij}$ and construct a ne network for the arc flows $\overline{f}_{ij}$. How should b_i be changed?

Exercise 7.8 Consider a transportation problem in which all cost coefficien c_{ij} are positive. Suppose that we increase the supply at some source nodes an the demand at some sink nodes. (In order to maintain feasibility, we assum that the increases are such that total demand is equal to total supply.) Is true that the value of the optimal cost will also increase? Prove or provide counterexample.

Exercise 7.9 Consider the uncapacitated network flow problem shown in Fi ure 7.37. The label next to each arc is its cost.

(a) What is the matrix $\mathbf{A}$ corresponding to this problem?

(b) Solve the problem using the network simplex algorithm. Start with t tree indicated by the dashed arcs in the figure.

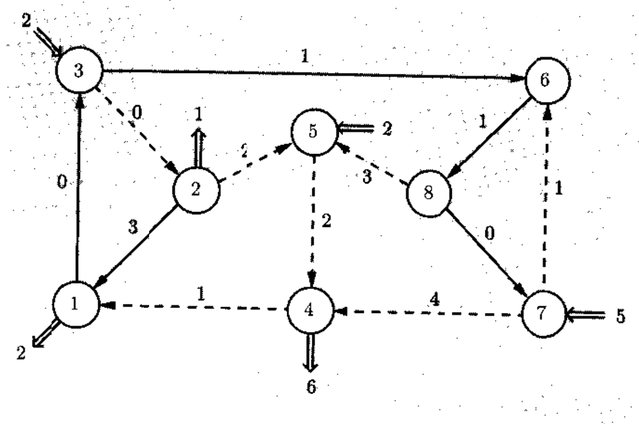


Figure 7.37: The network flow problem in Exercise 7.9.

Exercise 7.10 Consider the uncapacitated network flow problem shown in Figure 7.38. The label next to each arc is its cost. Consider the spanning tree indicated by the dashed arcs in the figure and the associated basic solution.

(a) What are the values of the arc flows corresponding to this basic solution? Is this a basic feasible solution?

(b) For this basic solution, find the reduced cost of each arc in the network.

(c) Is this basic solution optimal?

(d) Does there exist a nondegenerate basic feasible solution?

(e) Find an optimal dual solution.

(f) By how much can we increase c_{56} [the cost of arc (5,6)] and still have the same optimal basic feasible solution?

(g) If we increase the supply at node 1 and the demand at node 9 by a small positive amount δ, what is the change in the value of the optimal cost?

(h) Does this problem have a special structure that makes it simpler than the general uncapacitated network flow problem?

Exercise 7.11 (Degeneracy in a transportation problem) Consider a transportation problem with two source nodes s_1, s_2, and n demand nodes $1, \ldots, n$. All arcs (s_i, j) are assumed to be present and to have infinite capacity. Let $D = \sum_{i=1}^{n} d_i$ be the total demand. Let the supply at each source node be equal to $D/2$.

(a) How many basic variables are there in a basic feasible solution?

(b) Show that there exists a degenerate basic feasible solution if and only if there exists some set $S \subset \{1, \ldots, n\}$ such that $\sum_{i\in S} d_i = D/2$.

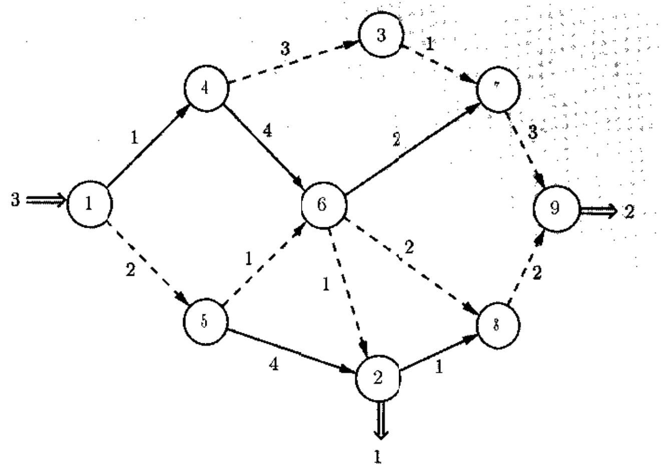


Figure 7.38: The network flow problem in Exercise 7.10.

Exercise 7.12 * (Degeneracy in the assignment problem) Consider the polyhedron $P \subset \Re^{k^2}$ defined by the constraints

$$\begin{aligned} \sum_{i=1}^{k} f_{ij} &= 1, & j &= 1,\dots,k, \\ \sum_{j=1}^{k} f_{ij} &= 1, & i &= 1,\dots,k, \\ f_{ij} &\geq 0, & i,j &= 1,\dots,k. \end{aligned}$$

(a) Show that P has $k!$ basic feasible solutions and that if $k > 1$, every basic feasible solution is degenerate.

(b) Show that there are $2^{k-1}k^{k-2}$ different bases that lead to any given basic feasible solution.

Exercise 7.13 Suppose that we are given a noninteger optimal solution to an uncapacitated network flow problem with integer data.

(a) Show that there exists a cycle with every arc on the cycle carrying a positive flow. What can you say about the cost of such a cycle?

(b) Suggest a method for constructing an integer optimal solution, without solving the problem from scratch. *Hint:* Remove cycles.

Exercise 7.14 (Decomposition of circulations) Let $\mathbf{A}$ be the node-arc incidence matrix associated with a directed graph with m arcs. Suppose that a vector $\mathbf{f}$ satisfies $\mathbf{Af} = \mathbf{0}$. Show that there exists a nonnegative integer k (with $k \leq m$), cycles $C_1, \dots, C_k$, and nonnegative scalars $a_1, \dots, a_k$, such that:

(i) $\mathbf{f} = \sum_{i=1}^{k} a_i \mathbf{h}^{C_i}$,

(ii) for every arc (k, ℓ) on a cycle C_i, $h_{k\ell}^{C_i}$ and $f_{k\ell}$ have the same sign.

Furthermore, show that if $\mathbf{f}$ is an integer vector, then the coefficients $a_1, \dots, a_k$ can be chosen to be integer. *Hint:* Reverse the arcs that carry negative flow and apply Lemma 7.1.

Exercise 7.15 (Flow decomposition theorem) State and prove a result analogous to the flow decomposition theorem in Exercise 7.14, for the case of a flow vector $\mathbf{f}$ that satisfies $\mathbf{Af} = \mathbf{b}$. *Hint:* Besides cycles, use paths as well.

Exercise 7.16 * (Negative cost cycle algorithm under the largest improvement rule) Consider the variant of the negative cost cycle algorithm in which we always choose a cycle C with the largest value of $\delta(C)|\mathbf{c}'\mathbf{h}^C|$. Let $\mathbf{f}$ be the current flow and let $\mathbf{f}^*$ be an optimal flow.

(a) Show that $\mathbf{f}^* - \mathbf{f}$ is equal to a nonnegative linear combination of at most m simple circulations, where m is the number of arcs. Furthermore, each such simple circulation is associated with an unsaturated cycle. *Hint:* Use the result in Exercise 7.14.

(b) Show that under the largest improvement rule, the cost improvement at each iteration is at least $(\mathbf{c}'\mathbf{f} - \mathbf{c}'\mathbf{f}^*)/m$.

(c) Assuming that all problem data are integer, show that the algorithm terminates after $O\big(m \log(mCU)\big)$ iterations, where C and U are upper bounds for $|c_{ij}|$ and u_{ij}, respectively.

Exercise 7.17 Consider a network flow problem and assume that there exists at least one feasible solution. We wish to show that the optimal cost is $-\infty$ if and only if there exists a negative cost directed cycle such that every arc on the cycle has infinite capacity.

(a) Provide a proof based on the flow decomposition theorem.

(b) For uncapacitated problems, provide a proof based on the network simplex method.

Exercise 7.18 Show that there is a one-to-one correspondence between augmenting paths in the maximum flow algorithm and negative cost unsaturated cycles in the network flow formulation of the maximum flow problem.

Exercise 7.19 Consider the maximum flow problem. Describe an algorithm with $O(|\mathcal{A}|)$ running time that determines whether the value of the maximum flow is infinite.

Exercise 7.20 (Duality and the max-flow min-cut theorem) Consider the maximum flow problem.

(a) Let p_i be a price variable associated with the flow conservation constraint at node i. Let q_{ij} be a price variable associated with the capacity constraint at arc (i, j). Write down a minimization problem, with variables p_i and q_{ij}, whose dual is the maximum flow problem.

(b) Show that the optimal value in the minimization problem is equal to the minimum cut capacity, and prove the max-flow min-cut theorem.

Exercise 7.21 (Finding a feasible solution) Show that a feasible solution to a capacitated network problem (if one exists) can be found by solving a maximum flow problem.

Exercise 7.22 (Connectivity and vulnerability) Consider a directed graph, and let us fix an origin node s and a destination node t. We define the *connectivity* of the graph as the maximum number of directed paths from s to t that do not share any nodes. We define the *vulnerability* of the graph as the minimum number of nodes (besides s and t) that need to be removed so that there exists no directed path from s to t. Prove that connectivity is equal to vulnerability. *Hint:* Convert the connectivity problem to a maximum flow problem.

Exercise 7.23 (The marriage problem) A small village has n unmarried men, n unmarried women, and m marriage brokers. Each broker knows a subset of the men and women and can arrange up to b_i marriages between any pair of men and women that she knows. Assuming that marriages are heterosexual and that each person can get married at most once, we are interested in determining the maximum number of marriages that are possible. Show that the answer can be found by solving a maximum flow problem.

Exercise 7.24 * (König-Egerváry theorem) Consider an $m \times n$ matrix whose entries are zero or one. We refer to a row or a column as a *line*. We say that a set of lines is a *cover* if every unit entry lies on one of the lines in the set. A set of unit entries are called *independent* if no two of them lie on the same line. Prove that the maximum cardinality of an independent set is equal to the smallest cardinality of a cover. *Hint:* Formulate an appropriate maximum flow problem.

Exercise 7.25 (The scaling method for the maximum flow problem) This exercise illustrates the *scaling method*, a common technique for reducing the complexity of network flow algorithms.

Consider a maximum flow problem Π. Let n be the number of nodes, let u_{ij} be the capacity of arc (i,j), assumed integer, and let v be the value of a maximum flow. We construct a scaled problem Π_s in which the capacity of each arc (i,j) is $\lfloor u_{ij}/2 \rfloor$, and we let v_s be the corresponding optimal value. (The notation $\lfloor a \rfloor$ stands for the largest integer k that satisfies $k \leq a$.)

(a) Consider an optimal flow for the problem Π_s, and multiply it by 2. Show that the result is a feasible flow for the original problem Π.

(b) Show that $2v_s \leq v \leq 2v_s + n^2$.

(c) Consider running the Ford-Fulkerson algorithm on problem Π, starting with the feasible flow described in (a). How many flow augmentations will be needed, and what is the total computational effort?

(d) Show how to solve the maximum flow problem with a total of $O(n^4 \log U)$ arithmetic operations, where U is an upper bound on the capacities u_{ij}.

Exercise 7.26 * (Birkhoff-von Neumann theorem) A square matrix $\mathbf{A}$ is called *doubly stochastic* if $\sum_{i=1}^{n} a_{ij} = 1$ for all j, $\sum_{j=1}^{n} a_{ij} = 1$ for all i, and all entries are nonnegative. A matrix $\mathbf{P}$ is called a *permutation matrix* if each row and each column has exactly one nonzero entry, which is equal to 1.

(a) Let $\mathbf{P}_1, \ldots, \mathbf{P}_k$ be permutation matrices, and let $\lambda_1, \ldots, \lambda_k$ be nonnegative scalars that sum to 1. Show that $\sum_{i=1}^{k} \lambda_i \mathbf{P}_i$ is doubly stochastic.

(b) Let $\mathbf{A}$ be a doubly stochastic matrix. Show that there exist permutation matrices $\mathbf{P}_1, \ldots, \mathbf{P}_k$, and nonnegative scalars $\lambda_1, \ldots, \lambda_k$ that sum to 1, such that $\mathbf{A} = \sum_{i=1}^{k} \lambda_i \mathbf{P}_i$. *Hint:* Consider the assignment problem.

Exercise 7.27 Consider the transportation problem shown in Figure 7.39, and solve it using the primal-dual method. Use $\mathbf{p} = (1, 1, 0, 0)$ to start the algorithm.

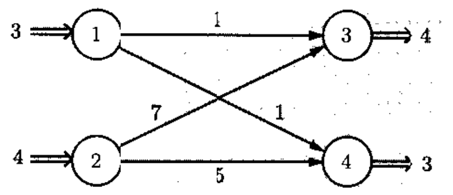


Figure 7.39: The transportation problem in Exercise 7.27. Arc costs are shown next to each arc.

Exercise 7.28 This exercise develops a more efficient method for computing θ^* in the primal-dual method. Let S be the set of nodes whose prices are to increase, as in the description of the general dual ascent algorithm. For every $j \notin S$, let

$$\theta_j^* = \min_{\{i \in S \mid (i,j) \in \mathcal{A}\}} (c_{ij} + p_j - p_i).$$

(a) Show that $\theta^* = \min_{j \notin S} \theta_j^*$.

(b) Suppose that some node $k \notin S$ satisfies $\theta_k^* = \theta^*$, so that node k enters the set S subsequent to the price increase. Let

$$\overline{\theta}_j = \min_{\{i \in S \cup \{k\} \mid (i,j) \in \mathcal{A}\}} (c_{ij} + p_j - p_i), \qquad j \notin S \cup \{k\}.$$

Show that $\overline{\theta}_j = \min\{\theta_j^*, c_{kj} + p_j - p_k\}$.

(c) Explain how to carry out each dual update in time proportional to n times the number of previously unlabeled nodes that become labeled.

(d) Show that the primal-dual method can be implemented so that it runs in time $O(n^3 B)$, where $B = \max_i |b_i|$.

Exercise 7.29 Consider a bipartite matching problem and suppose that every node has the same degree d. Show that there exists a perfect matching. *Hint:* Convert to a maximum flow problem and use the max-flow min-cut theorem.

Exercise 7.30* **(The primal-dual method as steepest dual ascent)** Consider the dual ascent algorithm. Show that the choice of the set S in the primal-dual method maximizes $(\mathbf{d}^S)'\mathbf{b}$ over all sets S for which $\mathbf{d}^S$ is a feasible direction.

Exercise 7.31 **(Dual simplex method for network flow problems)** Consider the uncapacitated network flow problem.

(a) Show that every spanning tree determines a basic solution to the dual problem.

(b) Given a basic feasible solution to the dual problem, associated with a certain tree, show that it is optimal if and only if the corresponding tree solution to the primal is feasible.

(c) If the tree solution in part (b) is infeasible, remove an arc that carries negative flow. Given that we wish to maintain dual feasibility, how should an arc be chosen to enter the tree?

(d) Note that the entering arc divides the tree into two parts. Consider the dual variables following a dual simplex update. Show that the dual variables in one part of the tree remain unchanged and in the other part of the tree, they are all changed by the same amount.

Exercise 7.32 **(Termination of the auction algorithm)** Consider a variation of the assignment problem in which we are given a subset $\mathcal{A}$ of the set of person-project pairs, and we allow f_{ij} to be nonzero only if $(i,j) \in \mathcal{A}$. We modify the bidding phase of the auction algorithm as follows. A person i takes into consideration only the profits $p_k - c_{ik}$ of those projects k for which $(i,k) \in \mathcal{A}$. Suppose that this form of the auction algorithm fails to terminate. Let I be the set of persons that bid an infinite number of times. Let J be the set of projects that receive an infinite number of bids.

(a) Show that if $i \in I$ and $(i,j) \in \mathcal{A}$, then $j \in J$.

(b) Show that the cardinality of I is strictly larger than the cardinality of J.

(c) Show the problem must be infeasible.

Exercise 7.33 **(Shortest path lengths and Bellman's equation)** Consider the all-to-one shortest path problem, and let $\mathbf{p}^*$ be the vector of shortest path lengths.

(a) Show that if every (directed) cycle has positive length, then Bellman's equation has a unique solution, equal to the shortest path lengths.

(b) Show that if every (directed) cycle has nonnegative length, and if $\mathbf{p}$ is a solution to Bellman's equation, then $\mathbf{p} \leq \mathbf{p}^*$. *Hint:* Consider $\max\{p_i, p_i^*\}$.

Exercise 7.34 **(From shortest path lengths to shortest paths)** Suppose that all directed cycles in a directed graph have nonnegative costs. Furthermore, suppose that the shortest path length p_i^* from any node to node n is known. Provide an algorithm that uses this information to determine a shortest path from node 1 to node n.

Exercise 7.35 **(Convergence of the Bellman-Ford algorithm)** This exercise develops an alternative proof of the convergence of the Bellman-Ford algorithm. Assume that the length of every cycle is nonnegative.

(a) Prove that $\mathbf{p}(t+1) \leq \mathbf{p}(t)$ for all t

(b) Prove that $\mathbf{p}(t) \geq \mathbf{p}^*$ for all t, and conclude that $\mathbf{p}(t)$ has a limit.

(c) Prove that $\mathbf{p}(t)$ can take only a finite number of values and therefore converges.

(d) Prove that the limit satisfies Bellman's equation.

(e) Prove that the algorithm converges to $\mathbf{p}^*$.

Exercise 7.36 **(Minimization of the mean cost of a cycle using linear programming)** Consider a directed graph in which each arc is associated with a cost c_{ij}. For any directed cycle, we define its mean cost as the sum of the costs of its arcs, divided by the number of arcs. We are interested in a directed cycle whose mean cost is minimal. We assume that there exists at least one directed cycle.

Consider the linear programming problem

$$\begin{array}{ll} \text{maximize} & \lambda \\ \text{subject to} & p_i + \lambda \leq p_j + c_{ij}, \qquad \text{for all arcs } (i,j). \end{array}$$

(a) Show that this maximization problem is feasible.

(b) Show that if $(\lambda, \mathbf{p})$ is a feasible solution to the maximization problem, then the mean cost of every directed cycle is at least λ.

(c) Show that the maximization problem has an optimal solution.

(d) Show how an optimal solution to the maximization problem can be used to construct a directed cycle with minimal mean cost.

Exercise 7.37 **(Minimization of the mean cost of a cycle using the Bellman-Ford algorithm)** Consider a directed graph in which each arc is associated with a cost c_{ij}. For any directed cycle, we define its mean cost as the sum of the costs of its arcs, divided by the number of arcs. We are interested in a directed cycle whose mean cost is minimal. We assume that there exists at least one directed cycle.

(a) Consider the algorithm

$$p_i(t+1) = \min_{j \in O(i)} \{c_{ij} + p_j(t)\}, \qquad \text{for all } i,$$

initialized with $p_i(0) = 0$ for all i. Show that $p_i(t)$ is equal to the length of a shortest walk that starts at i and and traverses t arcs.

(b) Prove that the optimal mean cycle cost λ satisfies

$$\lambda = \min_{i=1,\ldots,n} \max_{0 \leq k \leq n-1} \left(\frac{p_i(n) - p_i(k)}{n-k} \right),$$

where n is the number of nodes.

Exercise 7.38 **(Floyd-Warshall all-pairs shortest path algorithm)** Consider the all-pairs shortest path problem and assume that there are no negative