1

Introduction

1.1 PROBLEM FORMULATION

This book deals with a single type of network optimization problem with linear cost, known as the *transshipment* or *minimum cost flow* problem. In this section, we formulate this problem together with several special cases. One of the most important special cases is the *assignment problem*, which we will discuss in detail because it is simple and yet captures most of the important algorithmic aspects of the general problem.

Example 1.1. The Assignment Problem

Suppose that there are n persons and n objects that we have to match on a one-to-one basis. There is a benefit or value a_{ij} for matching person i with object j, and we want to assign persons to objects so as to maximize the total benefit. There is also a restriction that person i can be assigned to object j only if (i,j) belongs to a set of given pairs $\mathcal{A}$. Mathematically, we want to find a set of person-object pairs $(1,j_1),\ldots,(n,j_n)$ from $\mathcal{A}$ such that the objects $j_1,\ldots,j_n$ are all distinct, and the total benefit $\sum_{i=1}^{n} a_{ij_i}$ is maximized.

The assignment problem is important in many practical contexts. The most obvious ones are resource allocation problems, such as assigning employees to jobs, machines to tasks, etc. There are also situations where the assignment problem appears as a subproblem in various methods for solving more complex problems.

We may associate any assignment with the set of variables $\{x_{ij} \mid (i,j) \in \mathcal{A}\}$, where $x_{ij} = 1$ if person i is assigned to object j and $x_{ij} = 0$ otherwise. We may then formulate the assignment problem as the linear program

$$\begin{aligned} &\text{maximize} \quad \sum_{(i,j)\in\mathcal{A}} a_{ij}x_{ij} \\ &\text{subject to} \\ &\sum_{\{j \mid (i,j)\in\mathcal{A}\}} x_{ij} = 1, \qquad \forall\ i = 1,\ldots,n, \\ &\sum_{\{i \mid (i,j)\in\mathcal{A}\}} x_{ij} = 1, \qquad \forall\ j = 1,\ldots,n, \\ &0 \le x_{ij} \le 1, \qquad \forall\ (i,j) \in \mathcal{A}. \end{aligned} \tag{1.1}$$

Actually we should further restrict x_{ij} to be either 0 or 1; however, as we will show in the next chapter, the above linear program has a remarkable property: if it has a feasible solution at all, then it has an optimal solution where all x_{ij} are either 0 or 1. In fact, the set of its optimal solutions includes all the optimal assignments.

Another important property of the assignment problem is that it can be represented by a graph as shown in Fig. 1.1. Here, there are $2n$ nodes divided into two groups: n corresponding to persons and n corresponding to objects. Also, for every $(i,j) \in \mathcal{A}$, there is an arc connecting person i with object j. In the terminology of network problems, the variable x_{ij} is referred to as the *flow* of arc (i,j). The constraint $\sum_{\{j \mid (i,j)\in\mathcal{A}\}} x_{ij} = 1$ indicates that the total outgoing flow from node i should be equal to 1, which may be viewed as the (exogenous) *supply* of the node. Similarly, the constraint $\sum_{\{i \mid (i,j)\in\mathcal{A}\}} x_{ij} = 1$ indicates that the total incoming flow to node j should be equal to 1, which may be viewed as the (exogenous) *demand* of the node.

Before we can proceed with a formulation of more general network flow problems we must introduce some notation and terminology.

1.1.1 Graphs and Flows

We define a *directed graph*, $\mathcal{G} = (\mathcal{N}, \mathcal{A})$, to be a set $\mathcal{N}$ of *nodes* and a set $\mathcal{A}$ of pairs of distinct nodes from $\mathcal{N}$ called *arcs*. The numbers of nodes and arcs of $\mathcal{G}$ are denoted by N and A, respectively, and we assume throughout that $1 \le N < \infty$ and $0 \le A < \infty$. An arc (i,j) is viewed as an ordered pair, and is to be distinguished from the pair (j,i). If (i,j) is an arc, we say that (i,j) is *outgoing* from node i and *incoming* to node j; we also say that j is an *outward neighbor* of i and that i is an *inward neighbor* of j. We say that arc (i,j) is *incident* to i and to j, and that i is the *start* node and j is the

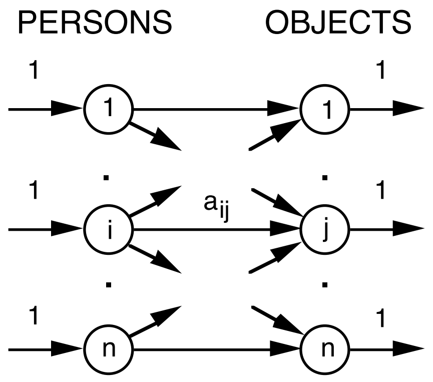


Figure 1.1 The graph representation of an assignment problem.

end node of the arc. The *degree* of a node i is the number of arcs that are incident to i.

A graph is said to be *bipartite* if its nodes can be partitioned into two sets $\mathcal{S}$ and $\mathcal{T}$ such that every arc has its start in $\mathcal{S}$ and its end in $\mathcal{T}$. The assignment graph of Fig. 1.1 is an example of a bipartite graph, with $\mathcal{S}$ and $\mathcal{T}$ being the sets of persons and objects, respectively.

We do not exclude the possibility that there is a separate arc connecting a pair of nodes in each of the two directions. However, we do not allow more than one arc between a pair of nodes in the same direction, so that we can refer unambiguously to the arc with start i and end j as arc (i, j). This was done for notational convenience. Our analysis can be simply extended to handle multiple arcs with start i and end j; the extension is based on modifying the graph by introducing for each such arc, an additional node, call it n, together with the two arcs (i, n) and (n, j). The codes in the appendixes can handle graphs that have multiple arcs between any pair of nodes in the same direction, without the above modification.

Paths and Cycles

A *path* P in a directed graph is a sequence of nodes $(n_1, n_2, \ldots, n_k)$ with $k \geq 2$ and a corresponding sequence of $k - 1$ arcs such that the ith arc in the sequence is either (n_i, n_{i+1}) (in which case it is called a *forward* arc of the path) or (n_{i+1}, n_i) (in which case it is called a *backward* arc of the path). A path is said to be *forward* (or *backward*) if all of its arcs are forward (respectively, backward) arcs. We denote by P^+ and P^- the sets of forward and backward arcs of P, respectively. Nodes n_1 and n_k are called the *start node* (or *origin*) and the *end node* (or *destination*) of P, respectively.

A *cycle* is a path for which the start and end nodes are the same. A path is said to be *simple* if it contains no repeated arcs and no repeated nodes, except that the start and end nodes could be the same (in which case the path is called a *simple cycle*). These definitions are illustrated in Fig. 1.2.

Note that the sequence of nodes $(n_1, n_2, \ldots, n_k)$ is not sufficient to specify a path; the sequence of arcs is also important, as Fig. 1.2(c) shows. The difficulty arises when for two successive nodes n_i and n_{i+1} of the path, both (n_i, n_{i+1}) and (n_{i+1}, n_i) are arcs, so there is ambiguity as to which of the two is the corresponding arc of the path. However, when the path is known to be forward or is known to be backward, it is uniquely specified by the sequence of its nodes. Throughout the book, we will make sure that the intended sequence of arcs is explicitly defined in ambiguous situations.

A graph that contains no simple cycles is said to be *acyclic*. A graph is said to be *connected* if for each pair of nodes i and j, there is a path starting at i and ending at j; it is said to be *strongly connected* if for each pair of nodes i and j, there is a forward path starting at i and ending at j. For example, the assignment graph of Fig. 1.1 may be connected but cannot be strongly connected.

We say that $\mathcal{G}' = (\mathcal{N}', \mathcal{A}')$ is a *subgraph* of $\mathcal{G} = (\mathcal{N}, \mathcal{A})$ if $\mathcal{G}'$ is a graph, $\mathcal{N}' \subset \mathcal{N}$, and $\mathcal{A}' \subset \mathcal{A}$. A *tree* is a connected acyclic graph. A *spanning tree* of a graph $\mathcal{G}$ is a subgraph of $\mathcal{G}$ that is a tree and that includes all the nodes of $\mathcal{G}$.

Flow and Divergence

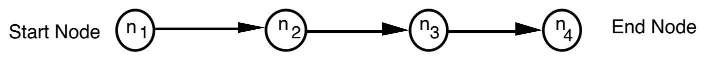
A *flow vector* x in a graph $(\mathcal{N}, \mathcal{A})$ is a set of scalars $\big\{x_{ij} \mid (i,j) \in \mathcal{A}\big\}$. We refer to x_{ij} as the flow of the arc (i,j), and we place no restriction (such as nonnegativity) on its value. The *divergence vector* y associated with a flow vector x is the N-dimensional vector with coordinates

$$y_i = \sum_{\{j \mid (i,j) \in \mathcal{A}\}} x_{ij} - \sum_{\{j \mid (j,i) \in \mathcal{A}\}} x_{ji}, \qquad \forall\ i \in \mathcal{N}. \tag{1.2}$$

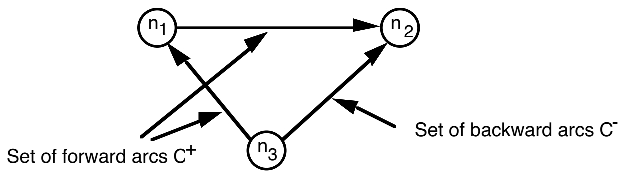
Thus, y_i is the total flow departing from node i less the total flow arriving at i; it is referred to as the divergence of i. For example, an assignment corresponds to a flow vector x with $x_{ij} = 1$ if person i is assigned to object j and $x_{ij} = 0$ otherwise (see Fig. 1.1); the assigned pairs involve each person exactly once and each object exactly once, if the divergence of each person node i is $y_i = 1$, and the divergence of each object node j is $y_j = -1$.

We say that node i is a *source* (respectively, *sink*) for the flow vector x if $y_i > 0$ (respectively, $y_i < 0$). If $y_i = 0$ for all $i \in \mathcal{N}$, then x is called a *circulation*. These definitions are illustrated in Fig. 1.3. Note that by adding Eq. (1.2) over all $i \in \mathcal{N}$, we obtain

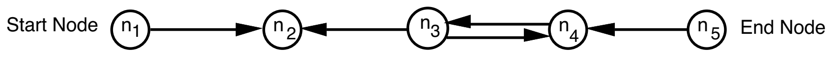
$$\sum_{i \in \mathcal{N}} y_i = 0$$



(a) A simple forward path P = (n_1, n_2, n_3, n_4).
The path P = (n_1, n_2, n_3, n_4, n_3, n_2, n_3) is also legitimate;
it is not simple, and it is neither forward nor backward.



(b) A simple cycle C = (n_1, n_2, n_3, n_1) which is neither forward nor backward.



(c) Path P = (n_1, n_2, n_3, n_4, n_5) with corresponding sequence of arcs
{ (n_1, n_2), (n_3, n_2), (n_3, n_4), (n_5, n_4) }.

Figure 1.2 Illustration of various types of paths. Note that for the path (c) it is necessary to specify the sequence of arcs of the path (rather than just the sequence of nodes) because both (n_3, n_4) and (n_4, n_3) are arcs. For a somewhat degenerate example that illustrates the fine points of the definitions, note that for the graph of (c), the node sequence

$$C = (n_3, n_4, n_3)$$

is associated with four cycles:

(1) The simple forward cycle with

$$C^+ = \{(n_3, n_4), (n_4, n_3)\}, \qquad C^- : \text{ empty.}$$

(2) The simple backward cycle with

$$C^- = \{(n_4, n_3), (n_3, n_4)\}, \qquad C^+ : \text{ empty.}$$

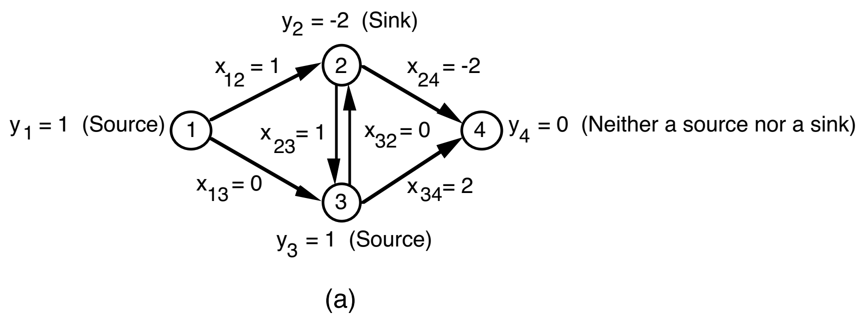
(3) The (nonsimple) cycle with

$$C^+ = \{(n_3, n_4)\}, \qquad C^- = \{(n_3, n_4)\}.$$

(4) The (nonsimple) cycle with

$$C^+ = \{(n_4, n_3)\}, \qquad C^- = \{(n_4, n_3)\}.$$

Note that the node sequence (n_3, n_4, n_3) determines the cycle uniquely if it is specified that the cycle is either forward or is backward.



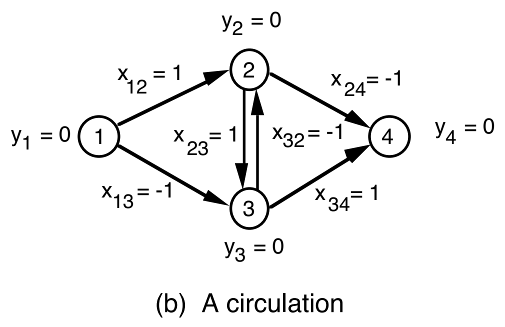


Figure 1.3 Illustration of various types of flows. The flow in (b) is a circulation because $y_i = 0$ for all i.

for any divergence vector y.

In applications, a negative arc flow indicates that whatever flow represents (material, electric current, etc.), moves in a direction opposite to the direction of the arc. We can always change the sign of the arc flow to positive as long as we change the arc direction, so in many situations we can assume without loss of generality that all arc flows are nonnegative. For the development of a general methodology, however, this device is often cumbersome, which is why we prefer to simply accept the possibility of negative arc flows.

Conformal Decomposition

It is often convenient to break down a flow vector into the sum of simpler components. A particularly useful decomposition arises when the components involve simple paths and cycles with orientation which is consistent to that of the original flow vector. This leads to the notion of a conformal realization, which we proceed to discuss.

We say that a path P *conforms* to a flow vector x if $x_{ij} > 0$ for all forward arcs (i,j) of P and $x_{ij} < 0$ for all backward arcs (i,j) of P, and furthermore either P is a cycle or else the start and end nodes of P are a source and a sink of x, respectively. Roughly, a path conforms to a flow vector if it "carries flow in the forward direction" – that is, in the direction from the start node to the end node. In particular, for a forward cycle to conform to a flow vector, all its arcs must have positive flow; for a forward path which is not a cycle to conform to a flow vector, its arcs must have positive flow, and in addition the start and end nodes must be a source and a sink, respectively.

A *simple path flow* is a flow vector that corresponds to sending a positive amount of flow along a simple path; more precisely, it is a flow vector x of the form

$$x_{ij} = \begin{cases} a & \text{if } (i,j) \in P^+ \\ -a & \text{if } (i,j) \in P^- \\ 0 & \text{otherwise,} \end{cases} \tag{1.3}$$

where a is a positive scalar, and P^+ and P^- are the sets of forward and backward arcs, respectively, of some simple path P.

We say that a simple path flow x^s *conforms* to a flow vector x if the path P corresponding to x^s via Eq. (1.3) conforms to x. This is equivalent to requiring that

$$0 < x_{ij} \qquad \text{for all arcs } (i,j) \text{ with } 0 < x^s_{ij},$$

$$x_{ij} < 0 \qquad \text{for all arcs } (i,j) \text{ with } x^s_{ij} < 0,$$

and that either P is a cycle or else the divergence (with respect to x) of the start node of P is positive and the divergence (with respect to x) of the end node of P is negative.

We now show that any flow vector can be decomposed into a set of conforming simple path flows. This result, illustrated in Fig. 1.4, turns out to be fundamental for our purposes. The proof is based on an algorithm that can be used to construct the constituent conforming components one by one. Such constructive proofs are often used in network optimization.

Proposition 1.1: (*Conformal Realization Theorem*) A nonzero flow vector x can be decomposed into the sum of t simple path flow vectors $x^1, x^2, \ldots, x^t$ that conform to x, with t being at most equal to the sum of the numbers of arcs and nodes $A+N$. If x is integer, then $x^1, x^2, \ldots, x^t$ can also be chosen to be integer. If x is a circulation, then $x^1, x^2, \ldots, x^t$ can be chosen to be simple circulations, and $t \le A$.

Proof: We first assume that x is a circulation. Our proof consists of showing how to obtain from x a simple circulation x' conforming to x and such that

$$0 \le x'_{ij} \le x_{ij} \qquad \text{for all arcs } (i,j) \text{ with } 0 \le x_{ij}, \tag{1.4a}$$

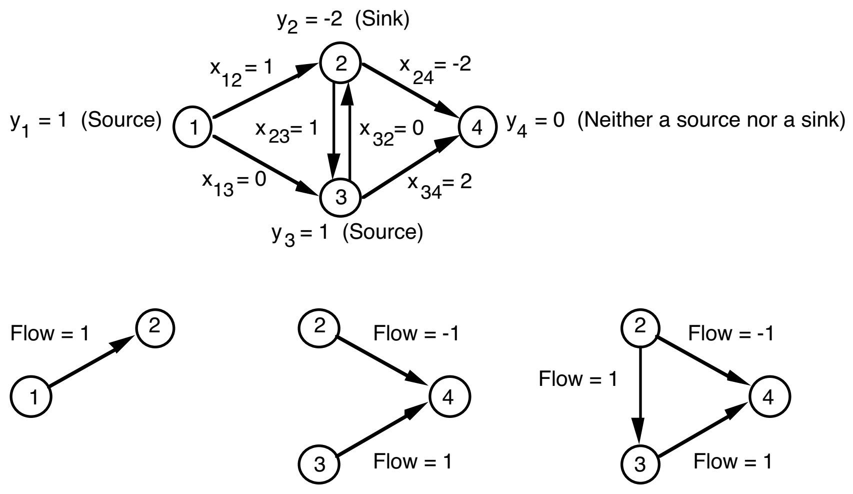


Figure 1.4 Decomposition of a flow vector x into three simple path flows conforming to x. The corresponding simple paths are $(1,2)$, $(3,4,2)$, and $(2,3,4,2)$. The first two are not cycles; they start at a source and end at a sink. Consistent with the definition of conformance of a path flow, each arc (i,j) of these paths carries positive (or negative) flow only if $x_{ij} > 0$ (or $x_{ij} < 0$, respectively). Arcs $(1,3)$ and $(3,2)$ do not belong to any of these paths because they carry zero flow. In this example, the decomposition is unique, but in general this need not be the case.

$$x_{ij} \le x'_{ij} \le 0 \qquad \text{for all arcs } (i,j) \text{ with } x_{ij} \le 0, \tag{1.4b}$$

$$x_{ij} = x'_{ij} \qquad \text{for at least one arc } (i,j) \text{ with } x_{ij} \ne 0. \tag{1.4c}$$

Once this is done, we subtract x' from x. We have $x_{ij} - x'_{ij} > 0$ only for arcs (i,j) with $x_{ij} > 0$, $x_{ij} - x'_{ij} < 0$ only for arcs (i,j) with $x_{ij} < 0$, and $x_{ij} - x'_{ij} = 0$ for at least one arc (i,j) with $x_{ij} \ne 0$. If x is integer, then x' and $x - x'$ will also be integer. We then repeat the process (for at most A times) with the circulation x replaced by the circulation $x - x'$ and so on, until the zero flow is obtained. This is guaranteed to happen eventually because $x - x'$ has at least one more arc with zero flow than x.

We now describe the procedure by which x' with the properties (1.4) is obtained; see Fig. 1.5. Choose an arc (i,j) with $x_{ij} \ne 0$. Assume that $x_{ij} > 0$. (A similar procedure can be used when $x_{ij} < 0$.) Construct a sequence of node subsets $T_0, T_1, \ldots$, as follows: Take $T_0 = \{j\}$. For $k = 0, 1, \ldots$, given T_k, let

$$T_{k+1} = \big\{n \notin \cup_{p=0}^{k} T_p \mid \text{there is a node } m \in T_k, \text{ and either an arc } (m,n) \\ \text{such that } x_{mn} > 0 \text{ or an arc } (n,m) \text{ such that } x_{nm} < 0\big\},$$

and mark each node $n \in T_{k+1}$ with the label "(m,n)" or "(n,m)," where m is a node of T_k such that $x_{mn} > 0$ or $x_{nm} < 0$, respectively. The procedure terminates when T_{k+1} is empty. We may view T_k as the set of nodes n that can be reached from j with a path of k arcs carrying "positive flow" in the direction from j to n.

We claim that one of the sets T_k contains node i. To see this, consider the set $\cup_k T_k$ of all nodes that belong to one of the sets T_k. By construction, there is no outgoing arc from $\cup_k T_k$ with positive flow and no incoming arc into $\cup_k T_k$ with negative flow. If i did not belong to $\cup_k T_k$, there would exist at least one incoming arc into $\cup_k T_k$ with positive flow, namely the arc (i,j). Thus, the total flow of arcs incoming to $\cup_k T_k$ must be positive, while the total flow of arcs outgoing from $\cup_k T_k$ is negative or zero. On the other hand, these two flows must be equal, since x is a circulation; this can be seen by adding the equation

$$\sum_{\{n \mid (m,n) \in \mathcal{A}\}} x_{mn} = \sum_{\{n \mid (n,m) \in \mathcal{A}\}} x_{nm}$$

over all nodes $m \in \cup_k T_k$. Therefore, we obtain a contradiction, and it follows that one of the sets T_k contains node i.

We now trace labels backward from i until node j is reached. [This will happen eventually because if "(m,n)" or "(n,m)" is the label of node n and $n \in T_{k+1}$, then $m \in T_k$, so a "cycle" of labels cannot be formed before reaching j.] In particular, let "(i_1,i)" or "(i,i_1)" be the label of i, let "(i_2,i_1)" or "(i_1,i_2)" be the label of i_1, etc., until a node i_k with label "(i_k,j)" or "(j,i_k)" is found. The cycle $C = (j, i_k, i_{k-1}, \ldots, i_1, i, j)$ is simple, it contains (i,j) as a forward arc, and is such that all its forward arcs have positive flow and all its backward arcs have negative flow (see Fig. 1.2). Let $a = \min_{(m,n)\in C} |x_{mn}| > 0$. Then the circulation x', where

$$x'_{ij} = \begin{cases} a & \text{if } (i,j) \in C^+ \\ -a, & \text{if } (i,j) \in C^- \\ 0 & \text{otherwise,} \end{cases}$$

has the required properties (1.4).

Consider now the case where x is not a circulation. We form an enlarged graph by introducing a new node s and by introducing for each node $i \in \mathcal{N}$ an arc (s,i) with flow x_{si} equal to the divergence y_i of Eq. (1.2). Then (by using also the fact $\sum_{i\in\mathcal{N}} y_i = 0$) the resulting flow vector is seen to be a circulation in the enlarged graph. This circulation, by the result just shown, can be decomposed into at most $A + N$ simple circulations of the enlarged graph, conforming to the flow vector. Out of these circulations, we consider those containing node s, and we remove s and its two incident arcs while leaving the other circulations unchanged. As a result we obtain a set of at most $A + N$ path flows of the original graph, which add up to x. These path flows also conform to x, as is required in order to prove the proposition. **Q.E.D.**

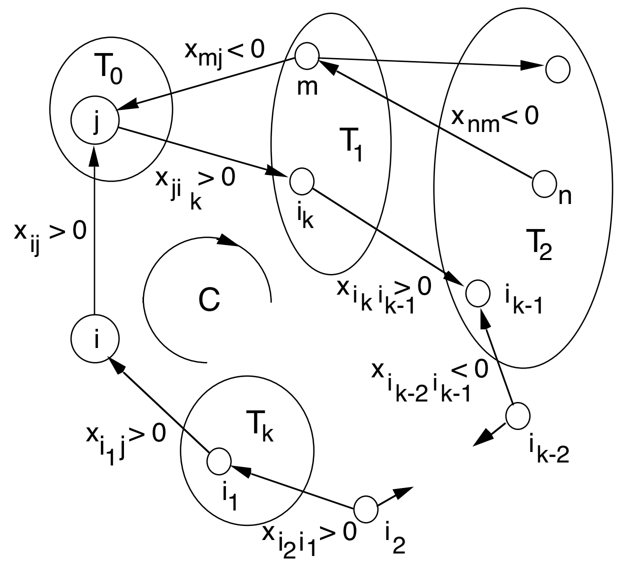


Figure 1.5 Construction of a cycle of nonzero flow arcs used in the proof of the Conformal Realization Theorem.

1.1.2 The Minimum Cost Flow Problem

The minimum cost flow problem is to find a set of arc flows that minimize a linear cost function subject to the constraints that they produce a given divergence vector and lie within some bounds; that is,

$$\text{minimize} \quad \sum_{(i,j)\in\mathcal{A}} a_{ij}x_{ij} \tag{MCF}$$

subject to

$$\sum_{\{j|(i,j)\in\mathcal{A}\}} x_{ij} - \sum_{\{j|(j,i)\in\mathcal{A}\}} x_{ji} = s_i, \qquad \forall\ i \in \mathcal{N}, \tag{1.5}$$

$$b_{ij} \le x_{ij} \le c_{ij}, \qquad \forall\ (i,j) \in \mathcal{A}, \tag{1.6}$$

where a_{ij}, b_{ij}, c_{ij}, and s_i are given scalars.

We use the following terminology.

a_{ij}: the *cost coefficient* (or simply *cost*) of (i,j).

b_{ij} and c_{ij}: the *flow bounds* of (i,j).

$[b_{ij}, c_{ij}]$: the *feasible flow range* of (i,j).

s_i: the *supply* of node i.

We also refer to the constraints (1.5) and (1.6) as the *conservation of flow constraints*, and the *capacity constraints*, respectively. A flow vector satisfying both of these constraints is called *feasible*, and if it satisfies just the capacity constraints, it is called *capacity-feasible*. If there exists at least one feasible flow vector, problem (MCF) is called *feasible*; otherwise it is called *infeasible*. Note that for feasibility we must have

$$\sum_{i \in \mathcal{N}} s_i = 0, \tag{1.7}$$

since by Eq. (1.2), for any flow vector, the sum of all the corresponding node divergences must be zero.

For a typical application of the minimum cost flow problem, think of the nodes as locations (cities, warehouses, or factories) where a certain product is produced or consumed. Think of the arcs as transportation links between the locations, each with transportation cost a_{ij} per unit transported. The problem then is to move the product from the production points to the consumption points at minimum cost while observing the capacity constraints of the transportation links.

On occasion, we will consider the variation of the minimum cost flow problem where the lower or the upper flow bound of some of the arcs is either $-\infty$ or ∞, respectively. In these cases, we will explicitly state so; thus, *in the absence of a contrary statement, we implicitly assume that every arc has real lower and upper flow bounds*.

The minimum cost flow problem is a special case of a linear programming problem, but it has a much more favorable structure than a general linear program. It has certain special properties that strongly affect the performance of algorithms. For example, the minimum cost flow problem with integer data can be solved using integer calculations exclusively. Furthermore, some methods (relaxation, auction) are very efficient for some minimum cost flow problems but are less efficient or inapplicable for general linear programs. In practice, minimum cost flow problems can often be solved hundreds and even thousands of times faster than general linear programs of comparable dimension.

The assignment problem is a special case of the minimum cost flow problem [see Eq. (1.1); by reversing the sign of the cost function, maximization can be turned into minimization]. Two other important special cases are described below.

Example 1.2. The Max-Flow Problem

In the max-flow problem there are two special nodes: the *source* (s) and the *sink* (t). Roughly, the objective is to push as much flow as possible from s into t while observing the capacity constraints. More precisely, we want to make the divergence of all nodes other than s and t equal to zero while maximizing the divergence of s (or, equivalently, minimizing the divergence of t).

The max-flow problem arises in many practical contexts, such as calculating the throughput of a highway system or a communication network. It also arises often as a subproblem in more complicated problems or algorithms.

We formulate this problem as a special case of the minimum cost flow problem by assigning cost zero to all arcs and by introducing an arc (t, s) with cost -1 and with an appropriately large upper flow bound and small lower flow bound, as shown in Fig. 1.6. Mathematically, the problem is as follows:

$$\begin{aligned}
&\text{maximize} \quad x_{ts} \\
&\text{subject to} \\
&\sum_{\{j|(i,j)\in\mathcal{A}\}} x_{ij} - \sum_{\{j|(j,i)\in\mathcal{A}\}} x_{ji} = 0, \qquad \forall\ i \in \mathcal{N} \text{ with } i \neq s \text{ and } i \neq t, \\
&\sum_{\{j|(s,j)\in\mathcal{A}\}} x_{sj} = \sum_{\{i|(i,t)\in\mathcal{A}\}} x_{it} = x_{ts}, \\
&b_{ij} \leq x_{ij} \leq c_{ij}, \qquad \forall\ (i,j) \in \mathcal{A} \text{ with } (i,j) \neq (t,s), \\
&\sum_{\{i|(i,t)\in\mathcal{A}\}} b_{it} \leq x_{ts} \leq \sum_{\{i|(i,t)\in\mathcal{A}\}} c_{it}.
\end{aligned} \tag{1.8}$$

The upper and lower bounds on x_{ts} are introduced in order to place the problem in the minimum cost flow format; they are actually redundant since they are implied by the upper and lower bounds on the flows of the arcs of $\mathcal{A}$. Also, viewing the problem as a maximization is consistent with its intuitive interpretation. Alternatively, we could write the problem as a minimization of $-x_{ts}$ subject to the same constraints.

In an alternative formulation the flow bounds on x_{ts} could be discarded, since they are implied by other bounds, namely $b_{it} \leq x_{it} \leq c_{it}$ for all $(i,t) \in \mathcal{A}$. We would then be dealing with a special case of the version of the minimum cost flow problem in which some of the flow bounds are $-\infty$ and/or ∞.

Example 1.3. The Transportation Problem

This problem is the same as the assignment problem except that the node supplies need not be 1 or -1 and maximization is replaced by minimization.

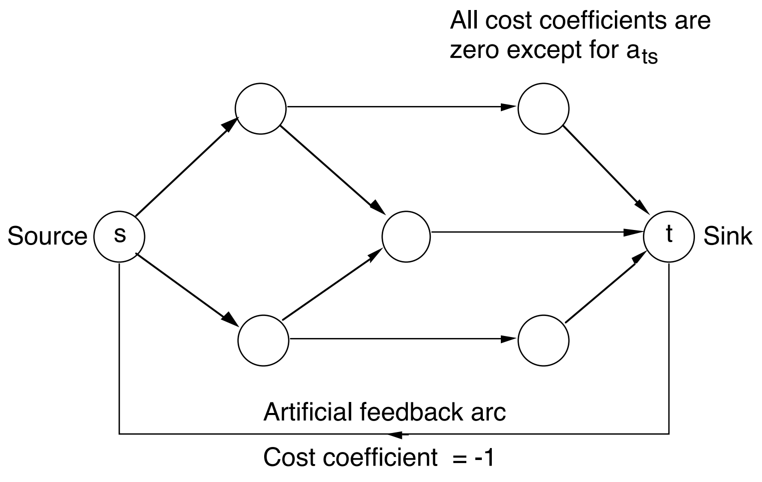


Figure 1.6 The minimum cost flow representation of a max-flow problem. At the optimum, the flow x_{ts} equals the maximum flow that can be sent from s to t through the subgraph obtained by deleting arc (t, s).

It has the form

$$\begin{aligned}
&\text{minimize} \quad \sum_{(i,j)\in\mathcal{A}} a_{ij}x_{ij} \\
&\text{subject to} \\
&\sum_{\{j|(i,j)\in\mathcal{A}\}} x_{ij} = \alpha_i, \qquad \forall\ i = 1,\ldots,m, \\
&\sum_{\{i|(i,j)\in\mathcal{A}\}} x_{ij} = \beta_j, \qquad \forall\ j = 1,\ldots,n, \\
&\quad 0 \le x_{ij} \le \min\{\alpha_i, \beta_j\}, \qquad \forall\ (i,j)\in\mathcal{A}.
\end{aligned} \tag{1.9}$$

Here α_i and β_j are positive scalars, which for feasibility must satisfy

$$\sum_{i=1}^{m} \alpha_i = \sum_{j=1}^{n} \beta_j,$$

[see Eq. (1.7)]. In an alternative formulation, the upper bound constraint $x_{ij} \le \min\{\alpha_i, \beta_j\}$ could be discarded, since it is implied by the conservation of flow and the nonnegativity constraints.

1.1.3 Transformations and Equivalences

The minimum cost flow problem can be represented in several equivalent forms, which we describe below.

Setting the Lower Flow Bounds to Zero

The lower flow bounds b_{ij} can be changed to zero by a translation of variables, that is, by replacing x_{ij} by $x_{ij} - b_{ij}$ and by adjusting the upper flow bounds and the supplies according to

$$c_{ij} := c_{ij} - b_{ij},$$

$$s_i := s_i - \sum_{\{j \mid (i,j) \in \mathcal{A}\}} b_{ij} + \sum_{\{j \mid (j,i) \in \mathcal{A}\}} b_{ji}.$$

Optimal flows and the optimal value of the original problem are obtained by adding b_{ij} to the optimal flow of each arc (i,j) and adding $\sum_{(i,j) \in \mathcal{A}} a_{ij} b_{ij}$ to the optimal value of the transformed problem, respectively. Working with the transformed problem saves computation time and storage, and for this reason most network flow codes assume that all lower flow bounds are zero.

Eliminating the Upper Flow Bounds

Once the lower flow bounds have been changed to zero, it is possible to eliminate the upper flow bounds, obtaining a problem with just a nonnegativity constraint on all the flows. This can be done by introducing an additional nonnegative variable z_{ij} that must satisfy the constraint

$$x_{ij} + z_{ij} = c_{ij}.$$

(In linear programming terminology, z_{ij} is known as a *slack variable*.) The resulting problem is a minimum cost flow problem involving for each arc (i,j), an extra node with supply c_{ij}, and two outgoing arcs, corresponding to the flows x_{ij} and z_{ij}; see Fig. 1.7.

Reduction to a Circulation Format

The problem can be put into *circulation format*, in which all node supplies are zero. One way of doing this is to introduce a new node t and an arc (t,i) for each node i with nonzero supply s_i. We may then introduce the constraint $s_i \leq x_{ti} \leq s_i$ and an arbitrary cost for the flow x_{ti}. Alternatively, we may introduce an arc (t,i) and a constraint $0 \leq x_{ti} \leq s_i$ for all i with $s_i > 0$, and an arc (i,t) and a constraint $0 \leq x_{it} \leq -s_i$ for all i with $s_i < 0$. The cost of these arcs should be very small (i.e., large negative) to force the corresponding flows to be at their upper bound at the optimum; see Fig. 1.8.

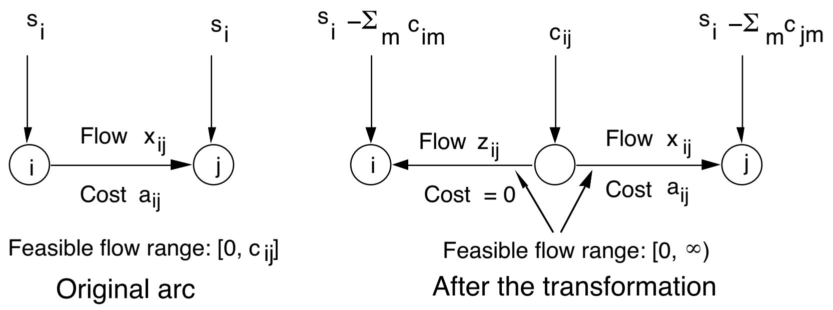


Figure 1.7 Eliminating the upper capacity bound by replacing each arc with a node and two outgoing arcs. Since for feasibility we must have $z_{ij} = c_{ij} - x_{ij}$, the upper bound constraint $x_{ij} \le c_{ij}$ is equivalent to the lower bound constraint $0 \le z_{ij}$. Furthermore, in view again of the equation $x_{ij} = c_{ij} - z_{ij}$, the conservation of flow equation

$$-\sum_j z_{ij} - \sum_j x_{ji} = s_i - \sum_j c_{ij}$$

for the modified problem is equivalent to the conservation of flow equation

$$\sum_j x_{ij} - \sum_j x_{ji} = s_i$$

for the original problem. Using these facts, it can be seen that the feasible flow vectors (x, z) of the modified problem can be paired on a one-to-one basis with the feasible flow vectors x of the original problem, and that the corresponding costs are equal. Thus, the modified problem is equivalent to the original problem.

Reduction to a Transportation or an Assignment Problem

Finally, the minimum cost flow problem may be transformed into a transportation problem of the form (1.9); see Fig. 1.9. The transportation problem (1.9) can itself be converted into an assignment problem by creating α_i unit supply sources (β_j unit demand sinks) for each transportation problem source i (sink j, respectively). For this reason, any algorithm that solves the assignment problem can be extended into an algorithm for the minimum cost flow problem. This motivates a useful way to develop and analyze new algorithmic ideas; apply them to the simpler assignment problem and generalize them using the construction just given to the minimum cost flow problem.

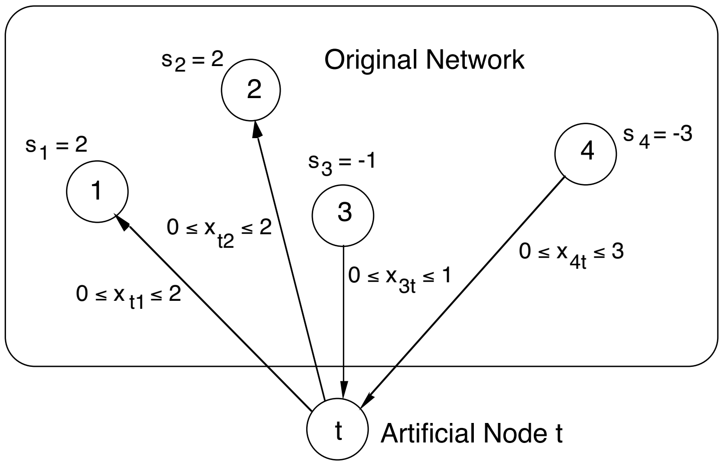


Figure 1.8 A transformation of the minimum cost flow problem into a circulation format. All artificial arcs have very large negative cost, to force the corresponding flows to their upper bounds at the optimum.

E X E R C I S E S

Exercise 1.1

Use the algorithm of the proof of the Conformal Realization Theorem to decompose the flow vector of Fig. 1.10 into simple path flows.

Exercise 1.2

Convert the minimum cost flow problem of Fig. 1.11 into a linear network flow problem involving only nonnegativity constraints on the variables.

Exercise 1.3

Consider the minimum cost flow problem and let p_i be a scalar for each node i. Change the cost of each arc (i, j) from a_{ij} to $a_{ij} + p_j - p_i$. Show that the optimal flow vectors are unaffected. *Note:* This transformation is often useful; for example to make all arc costs nonnegative – see Section 1.3.5.

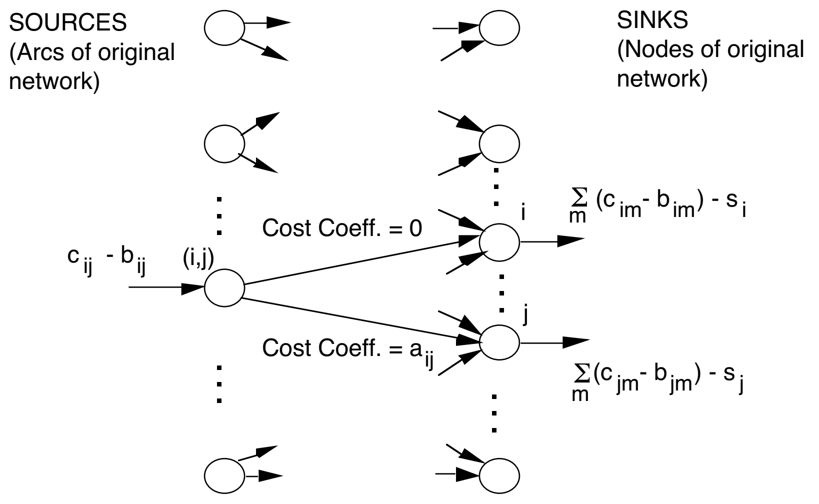

SOURCES
(Arcs of original network)
SINKS
(Nodes of original network)
Cost Coeff. = 0
Cost Coeff. = a_{ij}
$c_{ij} - b_{ij}$
(i,j)
i
j
$\sum_m (c_{im} - b_{im}) - s_i$
$\sum_m (c_{jm} - b_{jm}) - s_j$

Figure 1.9 Transformation of a minimum cost flow problem into a transportation problem of the form (1.9). The idea is to introduce a new node for each arc and introduce a slack variable for every arc flow; see Fig. 1.7. This not only eliminates the upper bound constraint on the arc flows, as in Fig. 1.7, but also creates a bipartite graph structure. In particular, we take as sources of the transportation problem the arcs of the original network, and as sinks of the transportation problem the nodes of the original network. Each transportation problem source has two outgoing arcs with cost coefficients as shown. The supply of each transportation problem source is the feasible flow range length of the corresponding original network arc. The demand of each transportation problem sink is the sum of the feasible flow range lengths of the outgoing arcs from the corresponding original network node minus the supply of that node, as shown. An arc flow x_{ij} in (MCF) corresponds to flows equal to x_{ij} and $c_{ij} - b_{ij} - x_{ij}$ on the transportation problem arcs $\big((i,j),j\big)$ and $\big((i,j),i\big)$, respectively.

Exercise 1.4 (Breadth-First Search)

Let i and j be two nodes of a directed graph $(\mathcal{N}, \mathcal{A})$.

(a) Consider the following algorithm, known as *breadth-first search*, for finding a path from i to j. Let $T_0 = \{i\}$. For $k = 0, 1, \ldots$, let

$$T_{k+1} = \{n \notin \cup_{p=0}^{k} T_p \mid \text{for some node } m \in T_k,\ (m,n) \text{ or } (n,m) \text{ is an arc}\},$$

and mark each node $n \in T_{k+1}$ with the label "(m,n)" or "(n,m)," where m is a node of T_k such that (m,n) or (n,m) is an arc, respectively. The algorithm terminates if either (1) T_{k+1} is empty or (2) $j \in T_{k+1}$. Show

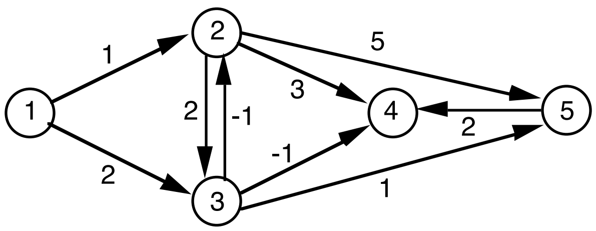


Figure 1.10 Flow vector for Exercise 1.1. The arc flows are the numbers shown next to the arcs.

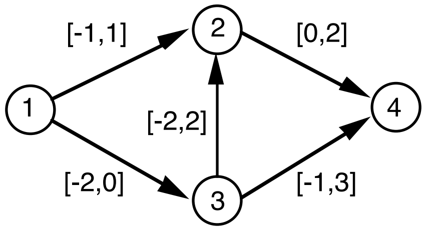


Figure 1.11 Minimum cost flow problem for Exercise 1.2. All arc costs are equal to 1, and all node supplies are equal to zero. The feasible flow ranges of the arcs are shown next to the arcs.

that case (1) occurs if and only if there is no path from i to j. If case (2) occurs, how would you use the labels to construct a path from i to j?

(b) Show that a path found by breadth-first search has a minimum number of arcs over all paths from i to j.

(c) Modify the algorithm of part (a) so that it finds a *forward* path from i to j.

Exercise 1.5 (Path Decomposition Theorem)

Use the Conformal Realization Theorem to show that a forward path P can be decomposed into a (possibly empty) collection of simple forward cycles, together with a simple forward path that has the same start node and end node as P. (Here "decomposition" means that the union of the arcs of the component paths is equal to the set of arcs of P with the multiplicity of repeated arcs properly accounted for.)

Exercise 1.6 (Inequality Constrained Minimum Cost Flows)

Consider the following variation of the minimum cost flow problem:

$$\text{minimize} \quad \sum_{(i,j)\in\mathcal{A}} a_{ij}x_{ij}$$

subject to

$$\underline{s}_i \le \sum_{\{j|(i,j)\in\mathcal{A}\}} x_{ij} - \sum_{\{j|(j,i)\in\mathcal{A}\}} x_{ji} \le \overline{s}_i, \qquad \forall\ i \in \mathcal{N},$$

$$b_{ij} \le x_{ij} \le c_{ij}, \qquad \forall\ (i,j) \in \mathcal{A},$$

where the bounds $\underline{s}_i$ and $\overline{s}_i$ on the divergence of node i are given. Convert this problem into the standard form of the minimum cost flow problem by adding an extra node and an arc from this node to every other node.

Exercise 1.7 (Node Throughput Constraints)

Consider the minimum cost flow problem with the additional constraints that the total flow of the outgoing arcs from each node i must lie within a given range $[\underline{t}_i, \overline{t}_i]$, that is,

$$\underline{t}_i \le \sum_{\{j|(i,j)\in\mathcal{A}\}} x_{ij} \le \overline{t}_i.$$

Convert this problem into the standard form of the minimum cost flow problem by adding an extra node and an extra arc for each existing node.

Exercise 1.8 (Piecewise Linear Arc Costs)

Consider the minimum cost flow problem with the difference that, instead of the linear form $a_{ij}x_{ij}$, each arc's cost function has the piecewise linear form

$$f_{ij}(x_{ij}) = \begin{cases} a_{ij}^1 x_{ij} & \text{if } b_{ij} \le x_{ij} \le m_{ij} \\ a_{ij}^1 m_{ij} + a_{ij}^2 (x_{ij} - m_{ij}) & \text{if } m_{ij} \le x_{ij} \le c_{ij}, \end{cases}$$

where m_{ij}, a_{ij}^1, and a_{ij}^2 are given scalars satisfying $b_{ij} \le m_{ij} \le c_{ij}$ and $a_{ij}^1 \le a_{ij}^2$.

(a) Show that the problem can be converted to a linear minimum cost flow problem where each arc (i,j) is replaced by two arcs with arc cost coefficients a_{ij}^1 and a_{ij}^2, and arc flow ranges $[b_{ij}, m_{ij}]$ and $[0, c_{ij} - m_{ij}]$, respectively.

(b) Generalize to the case of piecewise linear cost functions with more than two pieces.

1.2 THREE BASIC ALGORITHMIC IDEAS

In this section we will explain three main ideas underlying minimum cost flow algorithms:

(a) Primal cost improvement. Here we try to iteratively improve the cost to its optimal value by constructing a corresponding sequence of feasible flows.

(b) Dual cost improvement. Here we define a problem related to the minimum cost flow problem, called *dual problem*, whose variables are called *prices*. We then try to iteratively improve the dual cost to its optimal value by constructing a corresponding sequence of prices. Dual cost improvement algorithms also iterate on flows, which are related to the prices through a property called *complementary slackness*.

(c) Auction. This is a process that generates a sequence of prices in a way that is reminiscent of real-life auctions. Strictly speaking, there is no primal or dual cost improvement here, although one may view the auction process as trying to iteratively improve the dual cost in an approximate sense. In addition to prices, auction algorithms also iterate on flows, which are related to prices through a property called *ϵ-complementary slackness*; this is an approximate form of the complementary slackness property mentioned above.

For simplicity, in this chapter we will explain these ideas primarily through the assignment problem and the max-flow problem, deferring a more detailed development to subsequent chapters. Our illustrations, however, are relevant to the general minimum cost flow problem, since this problem can be reduced to the assignment problem (as was shown in the preceding section). Except for the max-flow analysis and the duality theory, the explanations in this section are somewhat informal. Precise statements of algorithms and results will be given in subsequent chapters.

1.2.1 Primal Cost Improvement

An important algorithmic idea for the minimum cost flow problem is to start from an initial feasible flow vector and then generate a sequence of feasible flow vectors, each having a better cost than the preceding one. The difference of any two successive flow vectors must be a circulation (since both are feasible), and for many interesting algorithms, including the simplex method, this circulation involves only a simple cycle. This idea will be first illustrated in terms of the assignment problem.

Multi-Person Swaps in the Assignment Problem

Consider the $n \times n$ assignment problem and suppose that we have a feasible assignment, that is, a set of n pairs (i, j) involving each person i exactly once and each object j exactly once. Consider now what happens if we do a *two-person swap*, that is, we replace two pairs (i_1, j_1) and (i_2, j_2) from the assignment with the pairs (i_1, j_2) and (i_2, j_1). The resulting assignment will still be feasible, and it will have a higher value if and only if

$$a_{i_1 j_2} + a_{i_2 j_1} > a_{i_1 j_1} + a_{i_2 j_2}.$$

Unfortunately, it may be impossible to improve the current assignment by a two-person swap, even if the assignment is not optimal; see Fig. 2.1. It turns out, however, that an improvement is possible by means of a *k-person swap*, for some $k \geq 2$, where a set of pairs $(i_1, j_1), \ldots, (i_k, j_k)$ from the current assignment is replaced by the pairs $(i_1, j_2), \ldots, (i_{k-1}, j_k), (i_k, j_1)$. This can be shown in the context of the minimum cost flow representation of the assignment problem:

$$\begin{aligned} &\text{maximize} \quad \sum_{(i,j)\in\mathcal{A}} a_{ij} x_{ij} \\ &\text{subject to} \\ &\sum_{\{j \mid (i,j)\in\mathcal{A}\}} x_{ij} = 1, \qquad \forall\, i = 1, \ldots, n, \\ &\sum_{\{i \mid (i,j)\in\mathcal{A}\}} x_{ij} = 1, \qquad \forall\, j = 1, \ldots, n, \\ &0 \leq x_{ij} \leq 1, \qquad \forall\, (i,j) \in \mathcal{A}. \end{aligned} \tag{2.1}$$

Feasible assignments correspond to feasible flow vectors $\{x_{ij} \mid (i, j) \in \mathcal{A}\}$ such that x_{ij} is either 0 or 1, and a k-person swap corresponds to a simple cycle with k forward arcs (corresponding to the new assignment pairs) and k backward arcs (corresponding to the current assignment pairs that are being replaced); see Fig. 2.2. Thus, *performing a k-person swap is equivalent to pushing one unit of flow along the corresponding simple cycle*. The k-person swap improves the assignment if and only if the *value* of the k-person swap, defined by

$$a_{i_k j_1} + \sum_{m=1}^{k-1} a_{i_m j_{m+1}} - \sum_{m=1}^{k} a_{i_m j_m}, \tag{2.2}$$

is positive.

By associating k-person swaps with simple cycle flows, we can show that a value-improving k-person swap exists if the current assignment is not optimal. For a detailed proof, see the subsequent Prop. 2.1. The main argument is

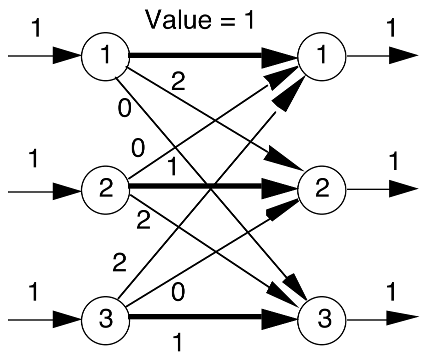


Figure 2.1 An example of a nonoptimal feasible assignment that cannot be improved by a two-person swap. The value of each pair is shown next to the corresponding arc. Here, the value of the assignment $\{(1,1),(2,2),(3,3)\}$ is left unchanged at 3 by any two-person swap. Through a three-person swap, however, we obtain the optimal assignment, $\{(1,2),(2,3),(3,1)\}$, which has value 6.

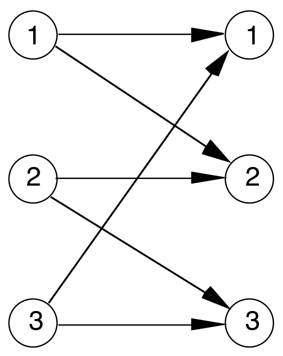


Figure 2.2 Correspondence of a k-person swap to a simple cycle. This is the same example as in the preceding figure. The backward arcs of the cycle are $(1,1)$, $(2,2)$, and $(3,3)$, and correspond to the current assignment pairs. The forward arcs of the cycle are $(1,2)$, $(2,3)$, and $(3,1)$, and correspond to the new assignment pairs. The k-person swap is value-improving because the sum of the values of the forward arcs $(2+2+2)$ is greater than the sum of the values of the backward arcs $(1+1+1)$.

based on the Conformal Realization Theorem (Prop. 1.1). Briefly, the difference between the flow vector corresponding to an optimal assignment and the vector corresponding to the current assignment is a circulation with arc flows equal to 0, 1, or -1, which can be decomposed into several conforming simple cycles (that is, k-person swaps). Thus, the value of the optimal assignment is equal to the value of the current assignment plus the sum of the values of the k-person swaps. It follows that if the current assignment is not optimal, then the value of at least one of the k-person swaps must be positive.

Primal cost improvement algorithms for the assignment problem are based on successive k-person swaps, each having positive or at least nonnegative value. There are several different algorithms of this type, including various forms of the simplex method, which will be discussed in detail in the next chapter.

Extension to the Minimum Cost Flow Problem

The algorithmic ideas just described for the assignment problem can be extended to the minimum cost flow problem

$$\text{minimize} \quad \sum_{(i,j)\in\mathcal{A}} a_{ij}x_{ij} \tag{MCF}$$

$$\text{subject to}$$

$$\sum_{\{j|(i,j)\in\mathcal{A}\}} x_{ij} - \sum_{\{j|(j,i)\in\mathcal{A}\}} x_{ji} = s_i, \qquad \forall\, i \in \mathcal{N},$$

$$b_{ij} \le x_{ij} \le c_{ij}, \qquad \forall\, (i,j) \in \mathcal{A}.$$

The role of k-person swaps is played by simple cycles with special properties. In particular, let x be a nonoptimal feasible flow vector, and let x^* be another feasible flow vector with smaller cost than x (for example, x^* could be an optimal flow vector). The difference $w = x^* - x$ is a circulation satisfying, for all arcs (i,j),

$$b_{ij} \le x^*_{ij} < x_{ij} \qquad \text{for all arcs } (i,j) \text{ with } w_{ij} < 0, \tag{2.3a}$$

$$x_{ij} < x^*_{ij} \le c_{ij} \qquad \text{for all arcs } (i,j) \text{ with } 0 < w_{ij}. \tag{2.3b}$$

According to the Conformal Realization Theorem (Prop. 1.1), w can be decomposed into the sum of several simple cycle flows x^s, $s = 1,\ldots,t$, which are conforming in the sense that, for all arcs (i,j),

$$w_{ij} < 0 \qquad \text{for all arcs } (i,j) \text{ with } x^s_{ij} < 0, \tag{2.4a}$$

$$0 < w_{ij} \qquad \text{for all arcs } (i,j) \text{ with } 0 < x^s_{ij}. \tag{2.4b}$$

Let us define a path P to be *unblocked with respect to* x if $x_{ij} < c_{ij}$ for all forward arcs $(i,j) \in P^+$ and $b_{ij} < x_{ij}$ for all backward arcs $(i,j) \in P^-$. From Eqs. (2.3) and (2.4), we see that each of the simple cycle flows x^s involves a cycle that is unblocked with respect to x. Let us define also the *cost of a* *simple cycle* C as the sum of the costs of the forward arcs minus the sum of the costs of the backward arcs of C, that is,

$$\sum_{(i,j)\in C^+} a_{ij} - \sum_{(i,j)\in C^-} a_{ij}.$$

Since $w = x^* - x$, the cost of w (that is, $\sum_{(i,j)\in\mathcal{A}} a_{ij}w_{ij}$) is equal to the cost of x^* minus the cost of x, so the cost of w must be negative. On the other hand, w is the sum of the simple cycle flows x^s, so the cost of w is equal to the sum of the costs of the corresponding simple cycles multiplied by positive constants (the flow values of the corresponding simple cycle flows). Therefore, the cost of at least one of these simple cycles must be negative. We have thus proved the following proposition.

Proposition 2.1: Consider the minimum cost flow problem and let x be a feasible flow vector which is not optimal. Then there exists a simple cycle flow that when added to x, produces a feasible flow vector with smaller cost that x; the corresponding cycle is unblocked with respect to x and has negative cost.

The major primal cost improvement algorithm for the minimum cost flow problem, the simplex method, uses simple cycle flows to produce improved feasible flow vectors, as will be discussed in the next chapter.

1.2.2 Application to the Max-Flow Problem – The Max-Flow/Min-Cut Theorem

We will now illustrate the preceding primal cost improvement approach in terms of the max-flow problem. In the process we will derive one of the most celebrated theorems of network optimization. To get a sense of the main ideas, consider the minimum cost flow formulation of the max-flow problem, given in Example 1.2, which involves the artificial feedback arc (t, s). Then, a negative cost cycle must necessarily include the arc (t, s), since this is the only arc with nonzero cost. By Prop. 2.1, if a feasible flow vector x is not optimal, there must exist a simple cycle with negative cost that is unblocked with respect to x; this cycle must consist of the arc (t, s) and a path from s to t, which is unblocked with respect to x. Thus, by adding to x the corresponding path flow, we obtain an improved flow vector. By similar reasoning, it follows that if there is no path from s to t that is unblocked with respect to a given flow vector x, then x must be optimal.

The max-flow/min-cut theorem and the Ford-Fulkerson algorithm, to be described shortly, are based on the above ideas. However, in view of the simplicity of the max-flow problem, the subsequent analysis will be couched in first principles; it will also develop some concepts that will be useful later. First some definitions are needed.

Cuts in a Graph

A *cut* Q in a graph $(\mathcal{N}, \mathcal{A})$ is a partition of the node set $\mathcal{N}$ into two nonempty subsets, a set $\mathcal{S}$ and its complement $\mathcal{N} - \mathcal{S}$; we will use the notation $Q = [\mathcal{S}, \mathcal{N} - \mathcal{S}]$. Note that the partition is ordered in the sense that the cut $[\mathcal{S}, \mathcal{N} - \mathcal{S}]$ is distinct from the cut $[\mathcal{N} - \mathcal{S}, \mathcal{S}]$. For a cut $Q = [\mathcal{S}, \mathcal{N} - \mathcal{S}]$, we will use the notation

$$Q^+ = \{(i,j) \in \mathcal{A} \mid i \in \mathcal{S}, j \notin \mathcal{S}\},$$

$$Q^- = \{(i,j) \in \mathcal{A} \mid i \notin \mathcal{S}, j \in \mathcal{S}\},$$

and we will say that Q^+ and Q^- are the *sets of forward and backward arcs of the cut*, respectively. We will say that the cut Q is *nonempty* if $Q^+ \cup Q^- \neq \emptyset$; otherwise we will say that Q is *empty*. We will say that the cut $[\mathcal{S}, \mathcal{N} - \mathcal{S}]$ *separates node s from node t* if $s \in \mathcal{S}$ and $t \notin \mathcal{S}$. These definitions are illustrated in Fig. 2.3.

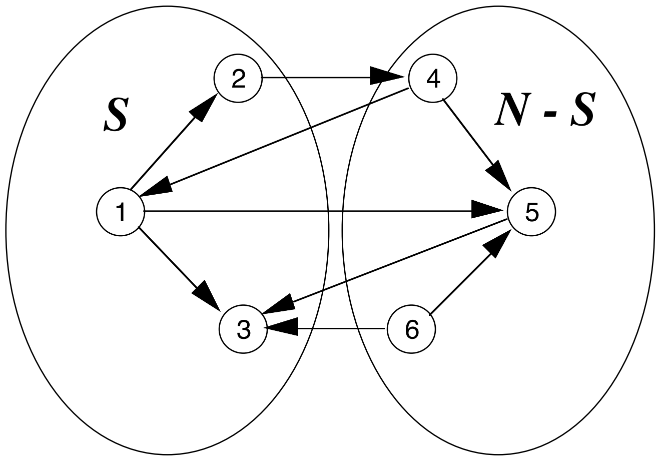


Figure 2.3 Illustration of a cut $Q = [\mathcal{S}, \mathcal{N} - \mathcal{S}]$, where $\mathcal{S} = \{1, 2, 3\}$. We have

$$Q^+ = \{(2,4), (1,5)\}, \qquad Q^- = \{(4,1), (5,3), (6,3)\}.$$

Given a flow vector x, the *flux across a nonempty cut* $Q = [\mathcal{S}, \mathcal{N} - \mathcal{S}]$ is defined to be the total net flow coming out of $\mathcal{S}$, that is, the scalar

$$F(Q) = \sum_{(i,j) \in Q^+} x_{ij} - \sum_{(i,j) \in Q^-} x_{ij}.$$

Using the definition of the divergence of a node [see Eq. (1.2)] and the following calculation, it can be seen that $F(Q)$ is also equal to the sum of the divergences y_i of the nodes in $\mathcal{S}$:

$$\begin{aligned} F(Q) &= \sum_{\{(i,j)\in\mathcal{A}\mid i\in\mathcal{S},j\notin\mathcal{S}\}} x_{ij} - \sum_{\{(i,j)\in\mathcal{A}\mid i\notin\mathcal{S},j\in\mathcal{S}\}} x_{ij} \\ &= \sum_{i\in\mathcal{S}}\left(\sum_{\{j\mid(i,j)\in\mathcal{A}\}} x_{ij} - \sum_{\{j\mid(j,i)\in\mathcal{A}\}} x_{ji}\right) = \sum_{i\in\mathcal{S}} y_i. \end{aligned} \tag{2.5}$$

(The second equality holds because the flow of an arc with both end nodes in $\mathcal{S}$ cancels out within the parentheses; it appears twice, once with a positive and once with a negative sign.)

Given flow bounds b_{ij} and c_{ij} for each arc (i,j), the *capacity of a nonempty cut* Q is

$$C(Q) = \sum_{(i,j)\in Q^+} c_{ij} - \sum_{(i,j)\in Q^-} b_{ij}. \tag{2.6}$$

Clearly, for any capacity-feasible flow vector x, the flux $F(Q)$ across Q is no larger than the cut capacity $C(Q)$. If $F(Q) = C(Q)$, then Q is said to be a *saturated cut with respect to* x; the flow of each forward (backward) arc of such a cut must be at its upper (lower) bound. By convention, every empty cut is also said to be saturated. The following is a simple but very useful result.

Proposition 2.2: Let x be a capacity-feasible flow vector, and let s and t be two nodes. Then exactly one of the following two alternatives holds:

(1) There exists a path from s to t that is unblocked with respect to x.

(2) There exists a saturated cut Q that separates s from t.

Proof: The proof is obtained by constructing an algorithm that terminates with either a path as in (1) or a cut as in (2). Consider the following algorithm, which is similar to the breadth-first search algorithm of Exercise 1.4; see Fig. 2.4. It generates a sequence of node sets $\{T_k\}$, starting with $T_0 = \{s\}$; each set T_k represents the set of nodes that can be reached from s with an unblocked path of k arcs.

Unblocked Path Search Algorithm

For $k = 0, 1, \ldots$, given T_k, terminate if either T_k is empty or $t \in T_k$; otherwise, set

$$\begin{aligned} T_{k+1} = \big\{n \notin \cup_{i=0}^{k} T_i \mid &\text{ there is a node } m \in T_k, \text{ and either an arc } (m,n) \\ &\text{such that } x_{mn} < c_{mn}, \text{ or an arc } (n,m) \text{ such that } b_{nm} < x_{nm}\big\}, \end{aligned}$$

and mark each node $n \in T_{k+1}$ with the label "(m, n)" or "(n, m)," where m is a node of T_k and (m, n) or (n, m) is an arc with the property stated in the above equation, respectively.

Since the algorithm terminates if T_k is empty, and T_k must consist of nodes not previously included in $\cup_{i=0}^{k-1} T_i$, the algorithm must eventually terminate. Let $\mathcal{S}$ be the union of the sets T_i upon termination. There are two possibilities:

(a) The final set T_k contains t, in which case, by tracing labels backward from t, an unblocked path P from s to t can be constructed. The forward arcs of P are of the form (m, n) with $x_{mn} < c_{mn}$ and the label of n being "(m, n)"; the backward arcs of P are of the form (n, m) with $b_{nm} < x_{nm}$ and the label of n being "(n, m)." Any cut separating s from t must contain a forward arc (m, n) of P with $x_{mn} < c_{mn}$ or a backward arc (n, m) of P with $b_{nm} < x_{nm}$, and therefore cannot be saturated. Thus, the result is proved in this case.

(b) The final set T_k is empty, in which case from the equation defining T_k, it can be seen that the cut $Q = [\mathcal{S}, \mathcal{N} - \mathcal{S}]$ is saturated and separates s from t. To show that there is no unblocked path from s to t, note that for any such path, we must have either an arc $(m, n) \in Q^+$ with $x_{mn} < c_{mn}$ or an arc $(n, m) \in Q^-$ with $b_{nm} < x_{nm}$, which is impossible, since Q is saturated.

Q.E.D.

A generalization of Prop. 2.2 that involves two disjoint subsets of nodes $\mathcal{N}^+$ and $\mathcal{N}^-$ in place of s and t is given in Exercise 2.14.

The Max-Flow/Min-Cut Theorem

Consider now the max-flow problem. We have a graph $(\mathcal{N}, \mathcal{A})$ with flow bounds b_{ij} and c_{ij} for the arcs, and two special nodes s and t. We want to maximize the divergence out of s over all capacity-feasible flow vectors having zero divergence for all nodes other than s and t. Given any such flow vector and any cut Q separating s from t, the divergence out of s is equal to the flux across Q [cf. Eq. (2.5)], which in turn is no larger than the capacity of Q. Thus, if the max-flow problem is feasible, we have

$$\text{Maximum Flow} \leq \text{Capacity of } Q. \tag{2.7}$$

The following theorem asserts that equality is attained for some Q. Part (a) of the theorem will assume the existence of an optimal solution to the max-flow problem. This assumption need not be satisfied; indeed it is possible that the max-flow problem has no feasible solution at all (consider a graph consisting

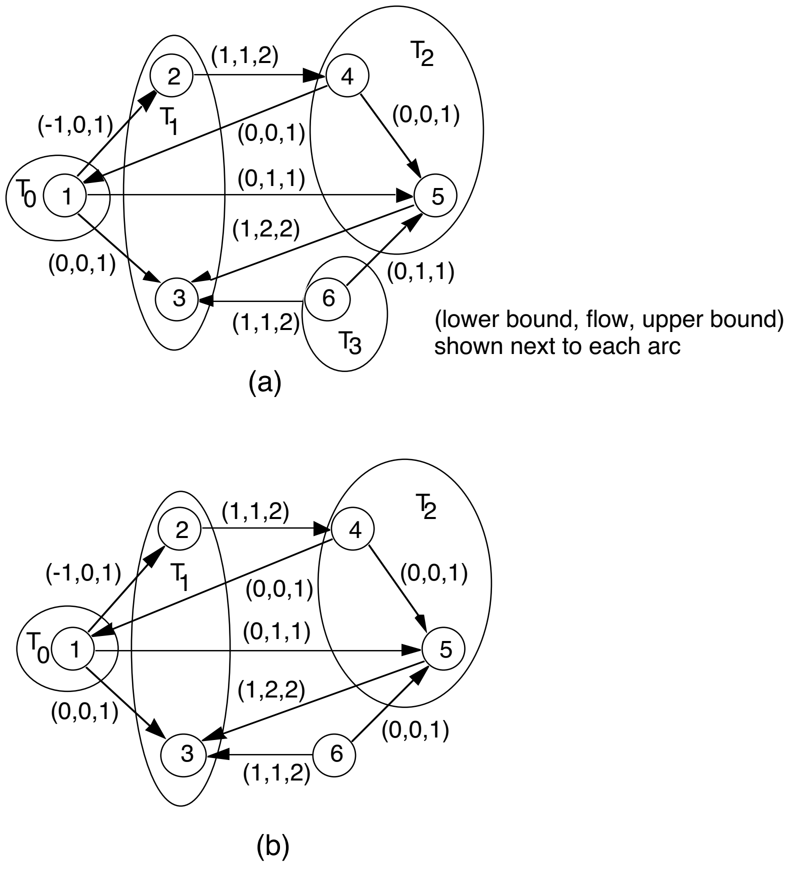


Figure 2.4 Illustration of the unblocked path search algorithm for finding an unblocked path from node 1 to node 6, or a saturated cut separating 1 from 6. The triplet (lower bound, flow, upper bound) is shown next to each arc. The figure shows the successive sets T_k generated by the algorithm. In case (a) there exists a unblocked path from 1 to 6, namely the path $(1, 3, 5, 6)$. In case (b), where the flow of arc $(6, 5)$ is at the lower bound rather than the upper bound, there is a saturated cut $[\mathcal{S}, \mathcal{N} - \mathcal{S}]$ separating 1 from 6, where $\mathcal{S} = \{1, 2, 3, 4, 5\}$ is the union of the sets T_k.

of a path from s to t the arcs of which have disjoint feasible flow ranges). In Chapter 2, however, we will show using the theory of the simplex method (see Prop. 3.1 in Section 2.3), that the max-flow problem (and indeed every minimum cost flow problem) has an optimal solution if it has at least one feasible solution. [This can also be easily shown using a fundamental result of mathematical analysis, the Weierstrass Theorem (see e.g. [Lue69], [Rud76]), which states that a continuous function attains a maximum over a nonempty and compact set.] If the lower flow bound is zero for every arc, the max-flow problem has at least one feasible solution, namely the zero flow vector. Thus the theory of Chapter 2 (or the Weierstrass Theorem) guarantees that the max-flow problem has an optimal solution in this case. This is stated as part (b) of the following theorem, even though its complete proof must await the developments of Chapter 2.

Proposition 2.3: (Max-Flow/Min-Cut Theorem)

(a) If x^* is an optimal solution of the max-flow problem, then the divergence out of s corresponding to x^* is equal to the minimum cut capacity over all cuts separating s from t.

(b) If all lower arc flow bounds are zero, the max-flow problem has an optimal solution, and the maximal divergence out of s is equal to the minimum cut capacity over all cuts separating s from t.

Proof: (a) Let F^* be the value of the maximum flow, that is, the divergence out of s corresponding to x^*. There cannot exist an unblocked path P from s to t with respect to x^*, since by increasing the flow of the forward arcs of P and by decreasing the flow of the backward arcs of P by a common positive increment, we would obtain a flow vector with divergence out of s larger than F^*. Therefore, by Prop. 2.2, there must exist a cut Q, that is saturated with respect to x^* and separates s from t. The flux across Q is equal to F^* and is also equal to the capacity of Q [since Q is saturated; see Eqs. (2.5) and (2.6)]. Since we know that F^* is less or equal to the minimum cut capacity [cf. Eq. (2.7)], the result follows.

(b) See the discussion preceding the proposition. **Q.E.D.**

The Ford-Fulkerson Algorithm

We now turn to an algorithm for solving the max-flow problem. This algorithm is of the primal cost improvement type, because it improves the primal cost (the divergence out of s) at every iteration. The idea is that, given a feasible flow vector x (i.e., one that is capacity-feasible and has zero divergence out of every node other than s and t), and a path P from s to t, which is unblocked with respect to x, we can increase the flow of all forward arcs $(i,j) \in P^+$ and decrease the flow of all backward arcs $(i,j) \in P^-$ by the positive amount

$$\delta = \min\big\{\{c_{ij} - x_{ij} \mid (i,j) \in P^+\}, \{x_{ij} - b_{ij} \mid (i,j) \in P^-\}\big\}.$$

The resulting flow vector $\overline{x}$, given by

$$\overline{x}_{ij} = \begin{cases} x_{ij} + \delta & \text{if } (i,j) \in P^+ \\ x_{ij} - \delta & \text{if } (i,j) \in P^- \\ x_{ij} & \text{otherwise,} \end{cases}$$

is feasible, and it has a divergence out of s that is larger by δ than the divergence out of s corresponding to x. We refer to P as an *augmenting path*, and we refer to the operation of replacing x by $\overline{x}$ as a *flow augmentation* along P. Such an operation may also be viewed as a modification of x along the negative cost cycle consisting of P and an artificial arc (t, s) that has cost -1; see the formulation of the max-flow problem as a minimum cost flow problem in Example 1.2 and Fig. 1.6, and the discussion at the beginning of the present subsection.

The algorithm starts with a feasible flow vector x. If the lower flow bound is zero for all arcs, the zero flow vector can be used as a starting vector; otherwise, a feasible starting flow vector can be obtained by solving an auxiliary max-flow problem with zero lower flow bounds – see Exercise 2.5. At each iteration the algorithm has a feasible flow vector x and uses the unblocked path search method, given in the proof of Prop. 2.2, to either generate a new feasible flow vector with larger divergence out of s or terminate with a maximum flow and a minimum capacity cut.

Typical Iteration of Ford-Fulkerson Algorithm

Use the unblocked path search method to either (1) find a saturated cut separating s from t or (2) find an unblocked path P with respect to x starting from s and ending at t. In case (1), terminate the algorithm; the current flow vector solves the max-flow problem. In case (2), perform an augmentation along P and go to the next iteration.

Figure 2.5 illustrates the Ford-Fulkerson algorithm.

Based on the preceding discussion, we see that with each augmentation the Ford-Fulkerson algorithm will improve the primal cost (the divergence out of s) by the augmentation increment δ. Thus, if δ is bounded below by some positive number, the algorithm can execute only a finite number of iterations and must terminate with an optimal solution. In particular, if the arc flow bounds are integer and the initial flow vector is also integer, δ will be a positive integer at each iteration, and the algorithm will terminate. The same is true even if the arc flow bounds and the initial flow vector are rational; by multiplication with a suitably large integer, one can scale these numbers up to integer while leaving the problem essentially unaffected.

On the other hand, if the problem data are irrational, proving termination of the Ford-Fulkerson algorithm is nontrivial. The proof (outlined in Exercise 2.10) depends on the use of the specific unblocked path search

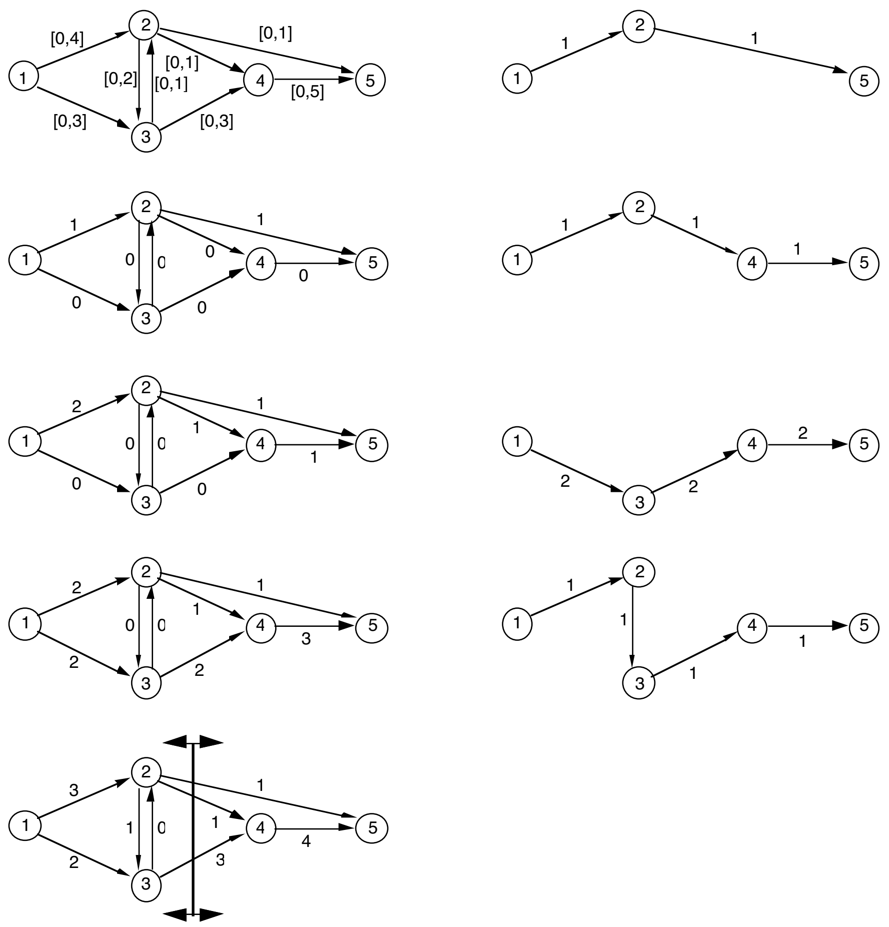


Figure 2.5 Illustration of the Ford-Fulkerson algorithm for finding a maximum flow from node $s = 1$ to node $t = 5$. The arc flow bounds are shown next to the arcs in the top left figure, and the starting flow is zero. The sequence of successive flow vectors is shown on the left, and the corresponding sequence of augmentations is shown on the right. The saturated cut obtained is $[\{1,2,3\},\{4,5\}]$. The capacity of this cut as well as the maximum flow is 5.

method of Prop. 2.2; this method yields augmenting paths with as few arcs as possible (Exercise 2.10). If unblocked paths are constructed using a different method, then, surprisingly, the Ford-Fulkerson algorithm need not terminate, and the generated sequence of divergences out of s may converge to a value strictly smaller than the maximum flow (for an example, see Exercise 2.9, and for a different example, see [FoF62], or [PaS82], p. 126, or [Roc84], p. 92).

Even with integer problem data, if the augmenting paths are constructed using a different unblocked path search method the Ford-Fulkerson algorithm may terminate in a very large number of iterations; see Fig. 2.6.

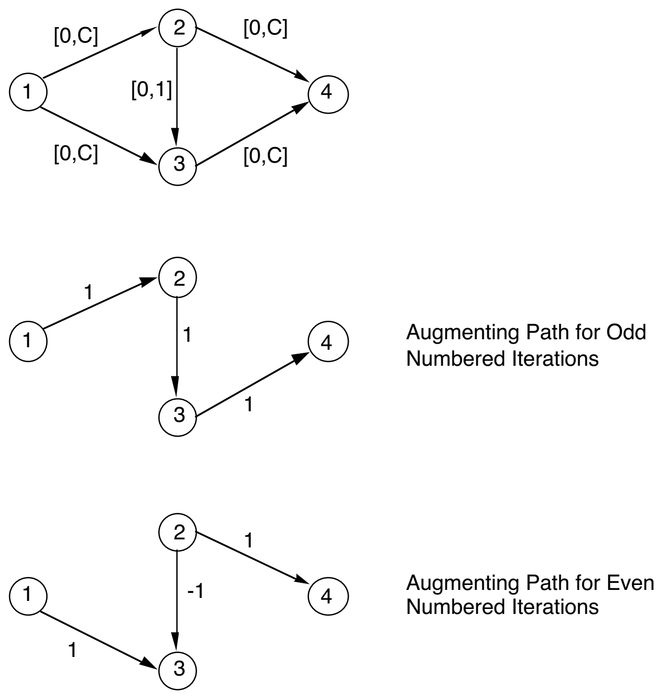

[0,C]
[0,C]
[0,1]
[0,C]
[0,C]
1
1
1
Augmenting Path for Odd Numbered Iterations
1
-1
1
Augmenting Path for Even Numbered Iterations

Figure 2.6 An example showing that if the augmenting paths used in the Ford-Fulkerson algorithm do not have a number of arcs that is as small as possible, the number of iterations may be very large. Here, C is a large integer. The maximum flow is $2C$, and can be produced after a sequence of $2C$ augmentations using the three-arc augmenting paths shown in the figure. If on the other hand the two-arc augmenting paths $(1, 2, 4)$ and $(1, 3, 4)$ are used, only two augmentations are needed.

The number of augmentations of the Ford-Fulkerson algorithm, with the unblocked path search method given, can be estimated as $O(NA)$ for an $O(NA^2)$ running time [since each augmentation requires $O(A)$ operations]; see Exercise 2.10. Several max-flow algorithms with more favorable worst case complexity estimates are available; see the references and Chapter 4.

1.2.3 Duality and Dual Cost Improvement

Linear programming duality theory deals with the relation between the original linear program and another linear program called *dual*. To develop an intuitive understanding of duality, we will focus on the assignment problem and consider a closely related economic equilibrium problem. Consider matching the n objects with the n persons through a market mechanism, viewing each person as an economic agent acting in his or her own best interest. Suppose that object j has a price p_j and that the person who receives the object must pay the price p_j. Then the net value of object j for person i is $a_{ij} - p_j$, and each person i will logically want to be assigned to an object j_i with maximal value, that is, with

$$a_{ij_i} - p_{j_i} = \max_{j \in A(i)} \{a_{ij} - p_j\}, \tag{2.8}$$

where

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

is the set of objects that can be assigned to person i. When this condition holds for all persons i, we say that the assignment and the set of prices satisfy *complementary slackness* (CS for short); the name comes from standard linear programming terminology. The economic system is then at equilibrium, in the sense that no person would have an incentive to unilaterally seek another object. Such equilibrium conditions are naturally of great interest to economists, but there is also a fundamental relation with the assignment problem. We have the following proposition.

Proposition 2.4: If a feasible assignment and a set of prices satisfy the complementary slackness conditions (2.8) for all persons i, then the assignment is optimal and the prices are an optimal solution of the following problem

$$\min_{\substack{p_j \\ j=1,\ldots,n}} \left\{ \sum_{i=1}^{n} \max_{j \in A(i)} \{a_{ij} - p_j\} + \sum_{j=1}^{n} p_j \right\}, \tag{2.9}$$

called the *dual problem*. Furthermore, the benefit of the optimal assignment and the optimal cost of the dual problem are equal.

Proof: The total cost of any feasible assignment $\{(i,k_i) \mid i = 1,\ldots,n\}$ satisfies

$$\sum_{i=1}^{n} a_{ik_i} \le \sum_{i=1}^{n} \max_{j \in A(i)} \{a_{ij} - p_j\} + \sum_{j=1}^{n} p_j, \tag{2.10}$$

for any set of prices $\{p_j \mid j = 1,\ldots,n\}$, since the first term of the right-hand side is no less than

$$\sum_{i=1}^{n} \left(a_{ik_i} - p_{k_i}\right),$$

while the second term is equal to $\sum_{i=1}^n p_{k_i}$. On the other hand, the given assignment and set of prices, denoted by $\{(i, j_i) \mid i = 1, \ldots, n\}$ and $\{\overline{p}_j \mid j = 1, \ldots, n\}$, respectively, satisfy the CS conditions, so we have

$$a_{ij_i} - \overline{p}_{j_i} = \max_{j \in A(i)} \{a_{ij} - \overline{p}_j\}, \qquad i = 1, \ldots, n.$$

By adding this relation over all i, we see that

$$\sum_{i=1}^n a_{ij_i} = \sum_{i=1}^n \left(\max_{j \in A(i)} \{a_{ij} - \overline{p}_j\} + \overline{p}_{j_i} \right).$$

Therefore, the assignment $\{(i, j_i) \mid i = 1, \ldots, n\}$ attains the maximum of the left-hand side of Eq. (2.10) and is optimal for the primal problem, while $\{\overline{p}_j \mid j = 1, \ldots, n\}$ attains the minimum of the right-hand side of Eq. (2.10) and is optimal for the dual problem. Furthermore, the two optimal values are equal. **Q.E.D.**

Duality for the Minimum Cost Flow Problem

Consider now the minimum cost flow problem, which in a duality context will also be referred to as the *primal problem*. To develop duality theory for this problem, we introduce a price vector $p = \{p_j \mid j \in \mathcal{N}\}$, and we say that a flow-price vector pair (x, p) satisfies *complementary slackness (or CS for short)* if x is capacity-feasible and

$$p_i - p_j \leq a_{ij} \qquad \text{for all } (i,j) \in \mathcal{A} \text{ with } x_{ij} < c_{ij}, \tag{2.11a}$$

$$p_i - p_j \geq a_{ij} \qquad \text{for all } (i,j) \in \mathcal{A} \text{ with } b_{ij} < x_{ij}. \tag{2.11b}$$

The above conditions also imply that we must have

$$p_i = a_{ij} + p_j \qquad \text{for all } (i,j) \in \mathcal{A} \text{ with } b_{ij} < x_{ij} < c_{ij}.$$

An equivalent way to write the CS conditions is that, for all arcs (i, j), we have $b_{ij} \leq x_{ij} \leq c_{ij}$ and

$$x_{ij} = \begin{cases} c_{ij} & \text{if } p_i > a_{ij} + p_j \\ b_{ij} & \text{if } p_i < a_{ij} + p_j. \end{cases}$$

The above definition of CS and the subsequent proposition are also valid for the variations of the minimum cost flow problem where $b_{ij} = -\infty$ and/or $c_{ij} = \infty$ for some arcs (i, j). In particular, in the case where in place of the capacity constraints $b_{ij} \le x_{ij} \le c_{ij}$ there are only nonnegativity constraints $0 \le x_{ij}$, the CS conditions take the form

$$p_i - p_j \le a_{ij}, \qquad \forall\ (i,j) \in \mathcal{A}, \tag{2.11c}$$

$$p_i - p_j = a_{ij} \qquad \text{for all } (i,j) \in \mathcal{A} \text{ with } 0 < x_{ij}. \tag{2.11d}$$

The dual problem is obtained by a procedure which is standard in duality theory. We view p_i as a Lagrange multiplier associated with the conservation of flow constraint for node i and we form the corresponding Lagrangian function

$$\begin{aligned} L(x,p) &= \sum_{(i,j)\in\mathcal{A}} a_{ij}x_{ij} + \sum_{i\in\mathcal{N}} \left(s_i - \sum_{\{j|(i,j)\in\mathcal{A}\}} x_{ij} + \sum_{\{j|(j,i)\in\mathcal{A}\}} x_{ji} \right) p_i \\ &= \sum_{(i,j)\in\mathcal{A}} (a_{ij} + p_j - p_i)x_{ij} + \sum_{i\in\mathcal{N}} s_i p_i. \end{aligned} \tag{2.12}$$

Then the dual function value $q(p)$ at a vector p is obtained by minimizing $L(x,p)$ over all capacity-feasible flows x,

$$q(p) = \min_x \big\{ L(x,p) \mid b_{ij} \le x_{ij} \le c_{ij}, (i,j) \in \mathcal{A} \big\}. \tag{2.13}$$

Because the Lagrangian function $L(x,p)$ is separable in the arc flows x_{ij}, its minimization decomposes into A separate minimizations, one for each arc (i,j). Each of these minimizations can be carried out in closed form, yielding

$$q(p) = \sum_{(i,j)\in\mathcal{A}} q_{ij}(p_i - p_j) + \sum_{i\in\mathcal{N}} s_i p_i, \tag{2.14a}$$

where

$$\begin{aligned} q_{ij}(p_i - p_j) &= \min_{x_{ij}} \big\{ (a_{ij} + p_j - p_i)x_{ij} \mid b_{ij} \le x_{ij} \le c_{ij} \big\} \\ &= \begin{cases} (a_{ij} + p_j - p_i)b_{ij} & \text{if } p_i \le a_{ij} + p_j \\ (a_{ij} + p_j - p_i)c_{ij} & \text{if } p_i > a_{ij} + p_j. \end{cases} \end{aligned} \tag{2.14b}$$

The dual problem is

$$\begin{aligned} &\text{maximize } q(p) \\ &\text{subject to no constraint on } p, \end{aligned} \tag{2.15}$$

with the dual functional q given by Eq. (2.14).

Figure 2.7 illustrates the form of the functions q_{ij}. Since each of these functions is piecewise linear, the dual function q is also piecewise linear. The dual function also has some additional interesting structure. In particular, suppose that all node prices are changed by the same amount. Then the values of the functions q_{ij} do not change, since these functions depend on the price differences $p_i - p_j$. If in addition we have $\sum_{i\in\mathcal{N}} s_i = 0$, as we must if the problem is feasible, we see that the term $\sum_{i\in\mathcal{N}} s_i p_i$ also does not change. Thus, the dual function value does not change when all node prices are changed by the same amount, implying that the equal cost surfaces of the dual cost function are unbounded. Figure 2.8 illustrates the dual function for a simple example.

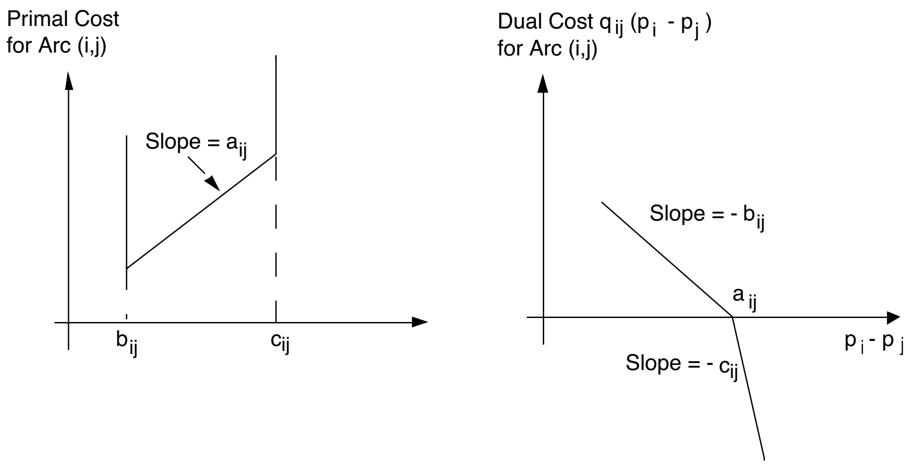


Figure 2.7 Form of the dual cost function q_{ij} for arc (i, j).

The following proposition is basic.

Proposition 2.5: If a feasible flow vector x^* and a price vector p^* satisfy the complementary slackness conditions (2.11a) and (2.11b), then x^* is an optimal primal solution and p^* is an optimal dual solution. Furthermore, the optimal primal cost and the optimal dual cost are equal.

Proof: We first show that for any feasible flow vector x and any price vector p, the primal cost of x is no less than the dual cost of p.

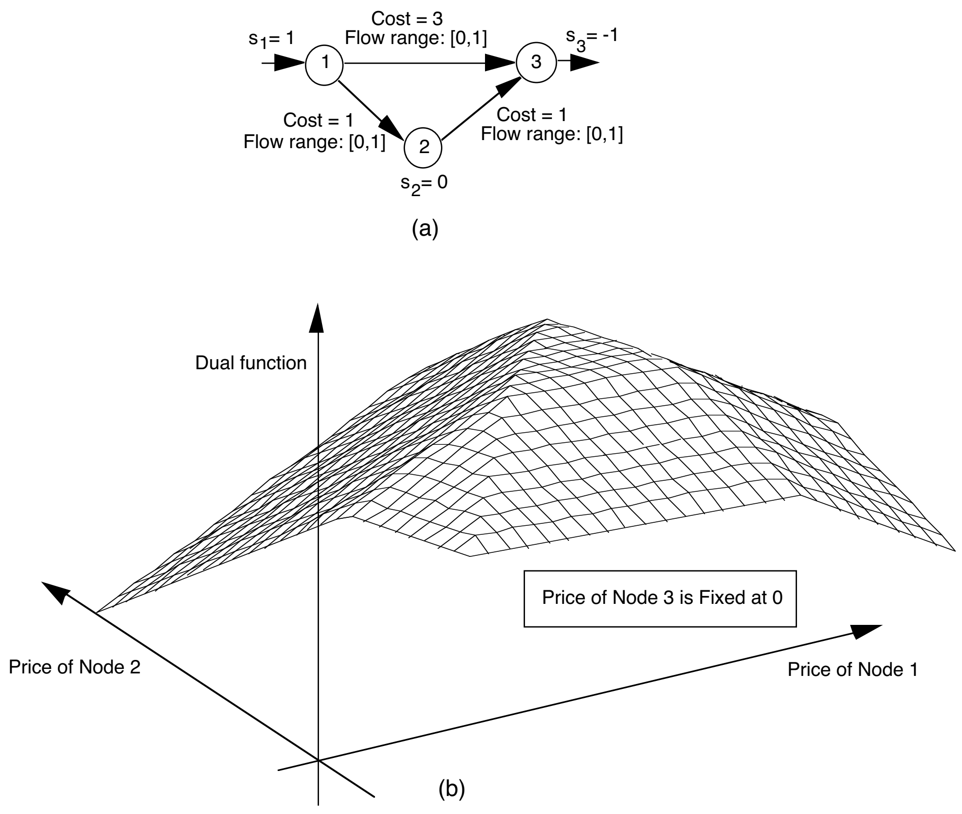


Figure 2.8 Form of the dual cost function q for the 3-node problem in (a). The optimal flow is $x_{12} = 1$, $x_{23} = 1$, $x_{13} = 0$. The dual function is

$$q(p_1, p_2, p_3) = \min\{0, 1 + p_2 - p_1\} + \min\{0, 1 + p_3 - p_2\}$$
$$+ \min\{0, 3 + p_3 - p_1\} + p_1 - p_3.$$

Diagram (b) shows the graph of the dual function in the space of p_1 and p_2, with p_3 fixed at 0. For a different value of p_3, say γ, the graph is "translated" by the vector (γ, γ); that is, we have $q(p_1, p_2, 0) = q(p_1 + \gamma, p_2 + \gamma, \gamma)$ for all (p_1, p_2). The dual function is maximized at the vectors p that satisfy CS together with the optimal x. These are the vectors of the form $(p_1 + \gamma, p_2 + \gamma, \gamma)$, where

$$1 \le p_1 - p_2, \qquad p_1 \le 3, \qquad 1 \le p_2.$$

Indeed, we have

$$\begin{aligned} q(p) &\leq L(x,p) \\ &= \sum_{(i,j)\in\mathcal{A}} a_{ij}x_{ij} + \sum_{i\in\mathcal{N}} \left(s_i - \sum_{\{j\mid(i,j)\in\mathcal{A}\}} x_{ij} + \sum_{\{j\mid(j,i)\in\mathcal{A}\}} x_{ji} \right) p_i \\ &= \sum_{(i,j)\in\mathcal{A}} a_{ij}x_{ij}, \end{aligned} \tag{2.16}$$

where the last equality follows from the feasibility of x. On the other hand, we have by the definition (2.13) of q

$$q(p^*) = \min_x \bigl\{ L(x,p^*) \mid b_{ij} \leq x_{ij} \leq c_{ij}, (i,j) \in \mathcal{A} \bigr\} = L(x^*,p^*) = \sum_{(i,j)\in\mathcal{A}} a_{ij}x_{ij}^*,$$

where the second equality is true because

(x^*, p^*) satisfies CS if and only if

$$x_{ij}^* \text{ minimizes } (a_{ij} + p_j^* - p_i^*)x_{ij} \text{ over all } x_{ij} \in [b_{ij}, c_{ij}], \ \forall\ (i,j) \in \mathcal{A},$$

and the last equality follows from the Lagrangian expression (2.12) and the feasibility of x^*. Therefore, x^* attains the minimum of the primal cost on the right-hand side of Eq. (2.16), and p^* attains the maximum of $q(p)$ on the left-hand side of Eq. (2.16), while the optimal primal and dual values are equal. **Q.E.D.**

There are also several other important duality results. In particular:

(a) The converse of the preceding proposition can be shown. That is, if x^* and p^* are optimal flow and price vectors for the minimum cost flow problem, and its dual problem, respectively, then x^* must be feasible and together with p^* it must satisfy CS.

(b) If the minimum cost flow problem (with upper and lower bounds on the arc flows) is feasible, then it can be shown that optimal primal and dual solutions x^* and p^* with equal cost exist. If the problem data (a_{ij}, b_{ij}, c_{ij}, and s_i) are integer, then these optimal solutions can be taken to be integer. [If some of the arc flows have no upper bound constraints the situation is somewhat more complicated, because it is possible that there exist feasible flow vectors of arbitrarily small (i.e., large negative) cost; such a problem will be called *unbounded* in Chapter 2. Barring this possibility, the existence of primal and dual optimal solutions with equal cost will be shown in Section 2.2.]

We will prove these results constructively in Chapter 2 (see Prop. 2.3 in Section 2.2 and Prop. 3.2 in Section 2.3) by deriving algorithms that obtain primal and dual optimal solutions, which are integer if the problem data are integer.

Interpretation of Complementary Slackness and the Dual Problem

As in the case of the assignment problem, the CS conditions have an economic interpretation. In particular, think of each node i as choosing the flow x_{ij} of each of its outgoing arcs (i,j) from the range $[b_{ij}, c_{ij}]$, on the basis of the following economic considerations: For each unit of the flow x_{ij} that node i sends to node j along arc (i,j), node i must pay a transportation cost a_{ij} plus a storage cost p_j at node j; for each unit of the residual flow $c_{ij} - x_{ij}$ that node i does not send to j, node i must pay a storage cost p_i. Thus, the total cost to node j is

$$(a_{ij} + p_j)x_{ij} + (c_{ij} - x_{ij})p_i.$$

It can be seen that the CS conditions (2.11) are equivalent to requiring that node i act in its own best interest by selecting the flow that minimizes the corresponding costs for each of its outgoing arcs (i,j); that is,

$$\begin{aligned}&(x,p) \text{ satisfies CS if and only if}\\ &\qquad x_{ij} \text{ minimizes } (a_{ij} + p_j - p_i)z_{ij} \text{ over all } z_{ij} \in [b_{ij}, c_{ij}], \ \forall\ (i,j) \in \mathcal{A}.\end{aligned}$$

To interpret the dual function $q(p)$, we continue to view a_{ij} and p_i as transportation and storage costs, respectively. Then, for a given price vector p and supply vector s, the dual function

$$q(p) = \min_{\substack{b_{ij} \le x_{ij} \le c_{ij},\\ (i,j)\in\mathcal{A}}} \left\{ \sum_{(i,j)\in\mathcal{A}} a_{ij}x_{ij} + \sum_{i\in\mathcal{N}} \left(s_i - \sum_{\{j \mid (i,j)\in\mathcal{A}\}} x_{ij} + \sum_{\{j \mid (j,i)\in\mathcal{A}\}} x_{ji} \right) p_i \right\}$$

is the minimum total transportation and storage cost to be incurred by the nodes, by choosing flows that satisfy the capacity constraints.

Suppose now that we introduce an organization that sets the node prices and collects the transportation and storage costs from the nodes. We see that if the organization wants to maximize its total revenue (given that the nodes will act in their own best interest), it must choose prices that solve the dual problem optimally.

Finally, we provide in Fig. 2.9, a geometric view of the relation between the primal and the dual problem. This geometric interpretation is directed toward the advanced reader and will not be needed in what follows. It demonstrates why the cost of any feasible flow vector is no less than the dual cost of any price vector, and why the optimal primal and dual costs are equal.

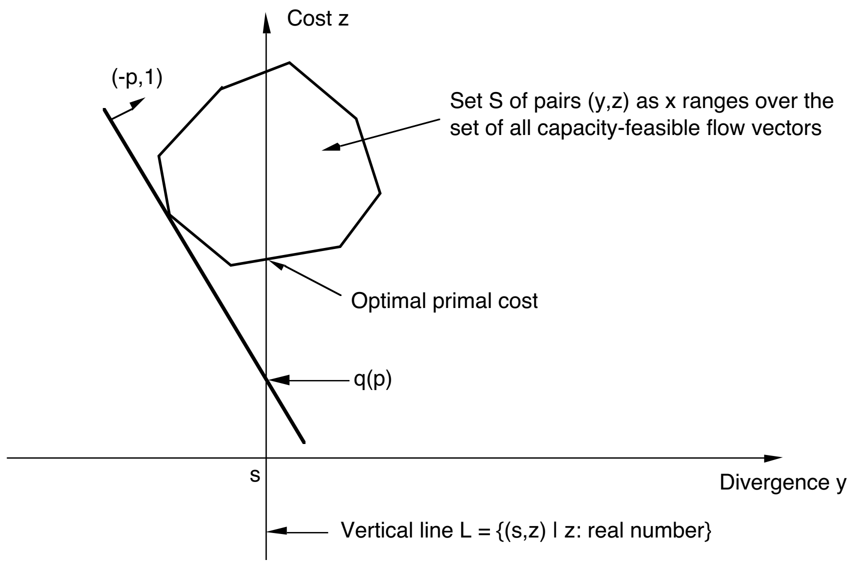

Cost z
(-p,1)
Set S of pairs (y,z) as x ranges over the set of all capacity-feasible flow vectors
Optimal primal cost
q(p)
s
Divergence y
Vertical line L = {(s,z) I z: real number}

Figure 2.9 Geometric interpretation of duality for the reader who is familiar with the notion and the properties of hyperplanes in a vector space. Consider the (polyhedral) set S consisting of all pairs (y, z), where y is the divergence vector corresponding to x and z is the cost of x, as x ranges over all capacity-feasible flow vectors. Then feasible flow vectors correspond to common points of S and the vertical line

$$L = \{(s, z) \mid z : \text{ real number}\}.$$

The optimal primal cost corresponds to the lowest common point.

On the other hand, for a given price vector p, the dual cost $q(p)$ can be expressed as [cf. Eq. (2.13)]

$$q(p) = \min_{x:\ \text{capacity feasible}} L(x, p) = \min_{(y,z)\in S} \left\{ z - \sum_{i\in\mathcal{N}} y_i p_i \right\} + \sum_{i\in\mathcal{N}} s_i p_i.$$

Based on this expression, it can be seen that $q(p)$ corresponds to the intersection point of the vertical line L with the hyperplane

$$\left\{ (y, z) \,\middle|\, z - \sum_{i\in\mathcal{N}} y_i p_i = q(p) - \sum_{i\in\mathcal{N}} s_i p_i \right\},$$

which supports from below the set S, and is normal to the vector $(-p, 1)$. The dual problem is to find a price vector p for which the intersection point is as high as possible. The figure illustrates the equality of the lowest common point of S and L (optimal primal cost), and the highest point of intersection of L by a hyperplane that supports S from below (optimal dual cost).

Dual Cost Improvement Algorithms

In analogy with primal cost improvement algorithms, one may start with a price vector and try to successively obtain new price vectors with improved dual cost. The major algorithms of this type involve price changes along a particular type of directions, known as *elementary*. Such directions are of the form $d = (d_1, \ldots, d_N)$, where

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

where $\mathcal{S}$ is a connected subset of nodes. Different algorithms correspond to different methods for determining the node set $\mathcal{S}$. Given an elementary direction of cost improvement and a corresponding set $\mathcal{S}$, the prices are iterated according to

$$p_i := \begin{cases} p_i + \gamma & \text{if } i \in \mathcal{S} \\ p_i & \text{if } i \notin \mathcal{S}, \end{cases}$$

where γ is some positive scalar that is small enough to ensure that the new price vector has an improved dual cost.

The existence of at least one elementary direction of improvement at a nonoptimal price vector will be shown in Chapter 3. This is an important and remarkable result, which may be viewed as a dual version of the result of Prop. 2.1 (at a nonoptimal flow vector, there exists at least one unblocked simple cycle with negative cost). In fact both results are special cases of a more general theorem concerning elementary vectors of subspaces, which is central in the theory of *monotropic programming*; see [Roc70], [Roc84].

Most dual cost improvement methods, simultaneously with changing p along a direction of dual cost improvement, also iterate on a flow vector x satisfying CS together with p. They terminate when x becomes feasible, at which time, by Prop. 2.5, the pair (x, p) must consist of a primal and a dual optimal solution.

In Chapter 3 we will discuss two main methods that select elementary directions of dual cost improvement in different ways:

(a) In the *primal-dual method*, the elementary direction has a *steepest ascent property*, that is, it provides the maximal rate of improvement of the dual cost per unit change in the price vector.

(b) In the *relaxation (or coordinate ascent) method*, the elementary direction is computed so that it has a small number of nonzero elements (i.e., the set $\mathcal{S}$ has few nodes). Such a direction may not be optimal in terms of rate of dual cost improvement, but can typically be computed much faster than the steepest ascent direction. Often the elementary direction has only one nonzero element, in which case only one node price coordinate is changed; this motivates the name "coordinate ascent." Note, however, that coordinate ascent directions cannot be used exclusively to improve the dual cost, as is shown in Fig. 2.10.

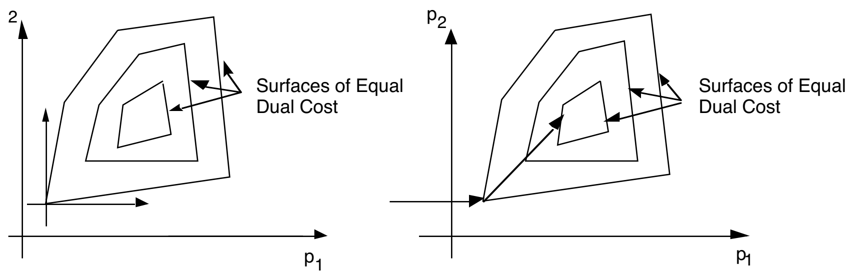


Figure 2.10 (a) The difficulty with using coordinate ascent iterations exclusively. The dual cost is piecewise linear, so at some corner points it may be impossible to improve the dual cost by changing any *single* price coordinate. (b) A dual cost improvement is possible by changing several price coordinates by equal amounts, which corresponds to an elementary direction.

As will be shown in Chapter 3, both the primal-dual method and the relaxation method terminate if the problem data are integer. Furthermore, simultaneously with an optimal price vector, they provide an optimal flow vector.

1.2.4 Auction

Our third type of algorithm represents a significant departure from the cost improvement idea; at any one iteration, it may deteriorate both the primal and the dual cost, although in the end it does find an optimal primal solution. It is based on an approximate version of complementary slackness, called ϵ-*complementary slackness*, and while it implicitly tries to solve a dual problem, it actually attains a dual solution that is not quite optimal. This subsection introduces the main ideas underlying auction algorithms. Chapter 4 provides a more complete discussion.

Naive Auction

Let us return to the assignment problem and consider a natural process for finding an equilibrium assignment and price vector. We will call this process the *naive auction algorithm*, because it has a serious flaw, as will be seen shortly. Nonetheless, this flaw will help motivate a more sophisticated and correct algorithm.

The naive auction algorithm proceeds in iterations and generates a sequence of price vectors and partial assignments. By a *partial assignment* we mean an assignment where only a subset of the persons have been matched with objects. A partial assignment should be contrasted with a feasible or complete assignment where all the persons have been matched with objects on a one-to-one basis. At the beginning of each iteration, the CS condition [cf. Eq. (2.8)]

$$a_{ij_i} - p_{j_i} = \max_{j \in A(i)} \{a_{ij} - p_j\}$$

is satisfied for all pairs (i, j_i) of the partial assignment. If all persons are assigned, the algorithm terminates. Otherwise some person who is unassigned, say i, is selected. This person finds an object j_i which offers maximal value, that is,

$$j_i = \arg \max_{j \in A(i)} \{a_{ij} - p_j\}, \tag{2.17}$$

and then:

(a) Gets assigned to the best object j_i; the person who was assigned to j_i at the beginning of the iteration (if any) becomes unassigned.

(b) Sets the price of j_i to the level at which he or she is indifferent between j_i and the second best object, that is, he or she sets p_{j_i} to

$$p_{j_i} + \gamma_i, \tag{2.18}$$

where

$$\gamma_i = v_i - w_i, \tag{2.19}$$

v_i is the best object value,

$$v_i = \max_{j \in A(i)} \{a_{ij} - p_j\}, \tag{2.20}$$

and w_i is the second best object value,

$$w_i = \max_{j \in A(i),\, j \neq j_i} \{a_{ij} - p_j\}. \tag{2.21}$$

(Note that as p_{j_i} is increased, the value $a_{ij_i} - p_{j_i}$ offered by object j_i to person i is decreased. γ_i is the largest increment by which p_{j_i} can be increased, while maintaining the property that j_i offers maximal value to i.)

This process is repeated in a sequence of iterations until each person has an assigned object.

We may view this process as an auction where at each iteration the bidder i raises the price of a preferred object by the *bidding increment* γ_i. Note that γ_i cannot be negative, since $v_i \geq w_i$ [compare Eqs. (2.20) and (2.21)], so the object prices tend to increase. The choice γ_i is illustrated in Fig. 2.11. Just as in a real auction, bidding increments and price increases spur competition by making the bidder's own preferred object less attractive to other potential bidders.

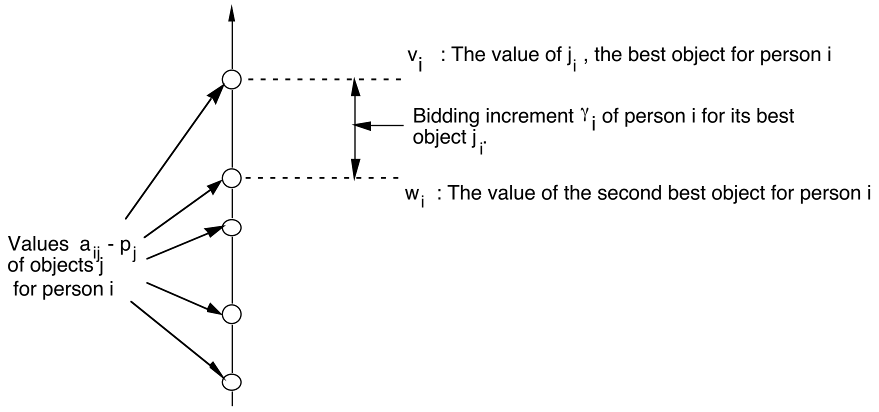

v_i : The value of j_i , the best object for person i
Bidding increment γ_i of person i for its best object j_i.
w_i : The value of the second best object for person i
Values a_{ij} - p_j of objects j for person i

Figure 2.11 In the naive auction algorithm, even after the price of the best object j_i is increased by the bidding increment γ_i, j_i continues to be the best object for the bidder i, so CS is satisfied at the end of the iteration. However, $\gamma_i = 0$ if there is a tie between two or more objects that are most preferred by i.

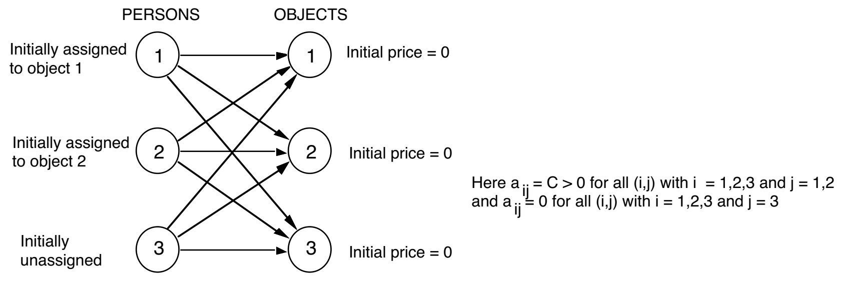
ϵ-Complementary Slackness

Unfortunately, the naive auction algorithm does not always work (although it is an excellent initialization procedure for other methods, such as primal-dual or relaxation, and it is useful in other specialized contexts; see Section 4.3). The difficulty is that the bidding increment γ_i is zero when two or more objects offer maximum value for the bidder i. As a result, a situation may be created where several persons contest a smaller number of equally desirable objects without raising their prices, thereby creating a never ending cycle; see Fig. 2.12.

To break such cycles, we introduce a perturbation mechanism, motivated by real auctions where each bid for an object must raise its price by a minimum positive increment, and bidders must on occasion take risks to win their preferred objects. In particular, let us fix a positive scalar ϵ, and say that a partial assignment and a price vector p satisfy *ϵ-complementary slackness (ϵ-CS for short)* if

$$a_{ij} - p_j \geq \max_{k\in A(i)} \{a_{ik} - p_k\} - \epsilon \tag{2.22}$$

for all assigned pairs (i, j). In words, to satisfy ϵ-CS, all assigned persons of the partial assignment must be assigned to objects that are within ϵ of being best.


PERSONS
OBJECTS
Initially assigned to object 1
Initially assigned to object 2
Initially unassigned
Initial price = 0
Initial price = 0
Initial price = 0
Here a_{ij} = C > 0 for all (i,j) with i = 1,2,3 and j = 1,2
and a_{ij} = 0 for all (i,j) with i = 1,2,3 and j = 3

At Start of Iteration #	Object Prices	Assigned Pairs	Bidder	Preferred Object	Bidding Increment
1	0,0,0	(1,1), (2,2)	3	2	0
2	0,0,0	(1,1), (3,2)	2	2	0
3	0,0,0	(1,1), (2,2)	3	2	0

Figure 2.12 Illustration of how the naive auction algorithm may never terminate for a problem involving three persons and three objects. Here objects 1 and 2 offer benefit $C > 0$ to all persons, and object 3 offers benefit 0 to all persons. The algorithm cycles as persons 2 and 3 alternately bid for object 2 without changing its price because they prefer equally object 1 and object 2 ($\gamma_i = 0$; compare Fig. 2.11).

The Auction Algorithm

We now reformulate the previous auction process so that the bidding increment is always at least equal to ϵ. The resulting method, the *auction algorithm*, is the same as the naive auction algorithm, except that the bidding increment γ_i is

$$\gamma_i = v_i - w_i + \epsilon \tag{2.23}$$

rather than $\gamma_i = v_i - w_i$ as in Eq. (2.19). With this choice, the ϵ-CS condition is satisfied, as illustrated in Fig. 2.13. The particular increment $\gamma_i = v_i - w_i + \epsilon$ used in the auction algorithm is the maximum amount with this property. Smaller increments γ_i would also work as long as $\gamma_i \geq \epsilon$, but using the largest possible increment accelerates the algorithm. This is consistent with experi-

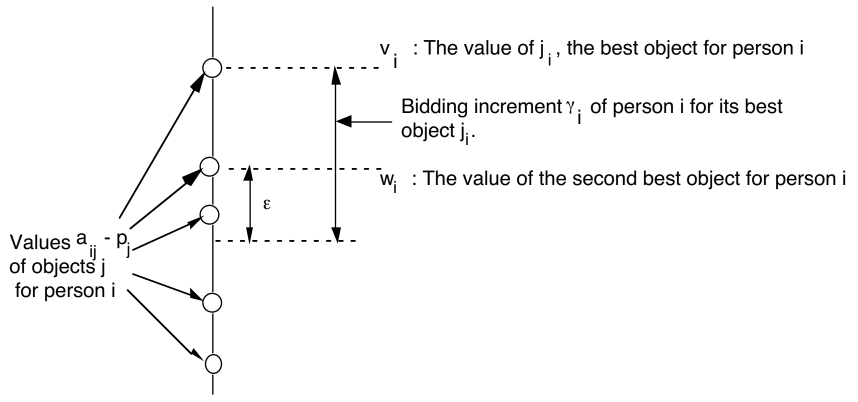
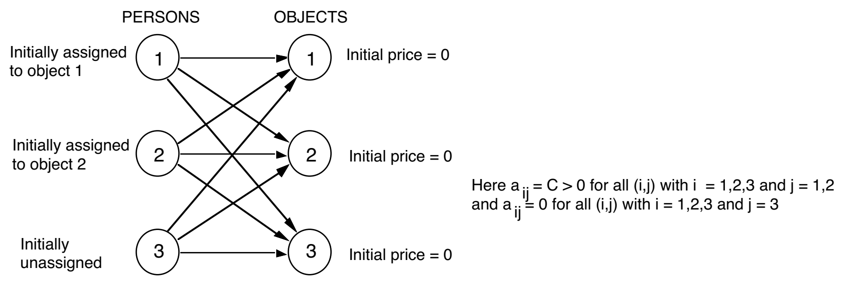

v_i : The value of j_i, the best object for person i
Bidding increment γ_i of person i for its best object j_i.
w_i : The value of the second best object for person i
ε
Values $a_{ij} - p_j$ of objects j for person i

Figure 2.13 In the auction algorithm, even after the price of the preferred object j_i is increased by the bidding increment γ_i , j_i will be within ϵ of being most preferred, so the ϵ-CS condition holds at the end of the iteration.

ence from real auctions, which tend to terminate faster when the bidding is aggressive.

It can be shown that this reformulated auction process terminates, necessarily with a feasible assignment and a set of prices that satisfy ϵ-CS. To get a sense of this, note that if an object receives a bid at m iterations, its price must exceed its initial price by at least $m\epsilon$. Thus, for sufficiently large m, the object will become "expensive" enough to be judged "inferior" to some object that has not received a bid so far. It follows that only for a limited number of iterations can an object receive a bid while some other object still has not yet received any bid. On the other hand, once every object has received at least one bid, the auction terminates. (This argument assumes that any person can bid for any object, but it can be generalized to the case where the set of feasible person-object pairs is limited, as long as at least one feasible assignment exists; see Prop. 1.2 in Section 4.1.) Figure 2.14 shows how the auction algorithm, based on the bidding increment $\gamma_i = v_i - w_i + \epsilon$ [see Eq. (2.23)], overcomes the cycling problem of the example of Fig. 2.12.

When the auction algorithm terminates, we have an assignment satisfying ϵ-CS, but is this assignment optimal? The answer depends strongly on the size of ϵ. In a real auction, a prudent bidder would not place an excessively high bid for fear of winning the object at an unnecessarily high price. Consistent with this intuition, we can show that if ϵ is small, then the final assignment will be "almost optimal." In particular, we will show that *the total benefit of the final assignment is within $n\epsilon$ of being optimal.* The idea is that a feasible assignment and a set of prices satisfying ϵ-CS may be viewed as satisfying CS for a *slightly different* problem, where all benefits a_{ij} are the


PERSONS
OBJECTS
Initially assigned to object 1
Initially assigned to object 2
Initially unassigned
Initial price = 0
Initial price = 0
Initial price = 0
Here $a_{ij} = C > 0$ for all (i,j) with i = 1,2,3 and j = 1,2
and $a_{ij} = 0$ for all (i,j) with i = 1,2,3 and j = 3

At Start of Iteration #	Object Prices	Assigned Pairs	Bidder	Preferred Object	Bidding Increment
1	0,0,0	(1,1), (2,2)	3	2	ϵ
2	0,ϵ,0	(1,1), (3,2)	2	1	2ϵ
3	2ϵ,ϵ,0	(2,1), (3,2)	1	2	2ϵ
4	2ϵ,3ϵ,0	(1,2), (2,1)	3	1	2ϵ
5	4ϵ,3ϵ,0	(1,2), (3,1)	2	2	2ϵ
6	...	...	...	...	...

Figure 2.14 Illustration of how the auction algorithm overcomes the cycling problem for the example of Fig. 2.12 by making the bidding increment at least ϵ. The table shows one possible sequence of bids and assignments generated by the auction algorithm, starting with all prices equal to 0 and with the partial assignment $\{(1,1),(2,2)\}$. At each iteration except the last, the person assigned to object 3 bids for either object 1 or 2, increasing its price by ϵ in the first iteration and by 2ϵ in each subsequent iteration. In the last iteration, after the prices of 1 and 2 reach or exceed C, object 3 receives a bid and the auction terminates.

same as before except the benefits of the n assigned pairs, which are modified by no more than ϵ.

Proposition 2.6: A feasible assignment, which satisfies ϵ-complementary slackness together with some price vector, is within $n\epsilon$ of being optimal. Furthermore, the price vector is within $n\epsilon$ of being an optimal solution of the dual problem.

Proof: Let A^* be the optimal total assignment benefit

$$A^* = \max_{\substack{k_i,\, i=1,\ldots,n \\ k_i \neq k_m \text{ if } i \neq m}} \sum_{i=1}^{n} a_{ik_i}$$

and let D^* be the optimal dual cost

$$D^* = \min_{\substack{p_j \\ j=1,\ldots,n}} \left\{ \sum_{i=1}^{n} \max_{j \in A(i)} \{a_{ij} - p_j\} + \sum_{j=1}^{n} p_j \right\}.$$

If $\{(i, j_i) \mid i = 1, \ldots, n\}$ is the given assignment satisfying the ϵ-CS condition together with a price vector $\overline{p}$, we have

$$\max_{j \in A(i)} \{a_{ij} - \overline{p}_j\} - \epsilon \leq a_{ij_i} - \overline{p}_{j_i}.$$

By adding this relation over all i, we see that

$$D^* \leq \sum_{i=1}^{n} \left(\max_{j \in A(i)} \{a_{ij} - \overline{p}_j\} + \overline{p}_{j_i} \right) \leq \sum_{i=1}^{n} a_{ij_i} + n\epsilon \leq A^* + n\epsilon.$$

Since we showed in Prop. 2.4 that $A^* = D^*$, it follows that the total assignment benefit $\sum_{i=1}^{n} a_{ij_i}$ is within $n\epsilon$ of the optimal value A^*, while the dual cost of $\overline{p}$ is within $n\epsilon$ of the optimal dual cost. **Q.E.D.**

Suppose now that the benefits a_{ij} are all integer, which is the typical practical case. (If a_{ij} are rational numbers, they can be scaled up to integer by multiplication with a suitable common number.) Then the total benefit of any assignment is integer, so if $n\epsilon < 1$, any complete assignment that is within $n\epsilon$ of being optimal must be optimal. It follows that *if*

$$\epsilon < \frac{1}{n}$$

and the benefits a_{ij} are all integer, then the assignment obtained upon termination of the auction algorithm is optimal.

Figure 2.15 shows the sequence of generated object prices for the example of Fig. 2.14 in relation to the contours of the dual cost function. It can be seen from this figure that each bid has the effect of setting the price of the object receiving the bid nearly equal (within ϵ) to the price that minimizes the dual cost with respect to that price, with all other prices held fixed (this will be shown rigorously in Secton 4.1). Successive minimization of a cost function along single coordinates is a central feature of coordinate descent and relaxation methods, which are popular for unconstrained minimization of smooth functions and for solving systems of smooth equations. Thus, the auction algorithm can be interpreted as an approximate coordinate descent method; as such, it is related to the relaxation method discussed in the previous subsection.

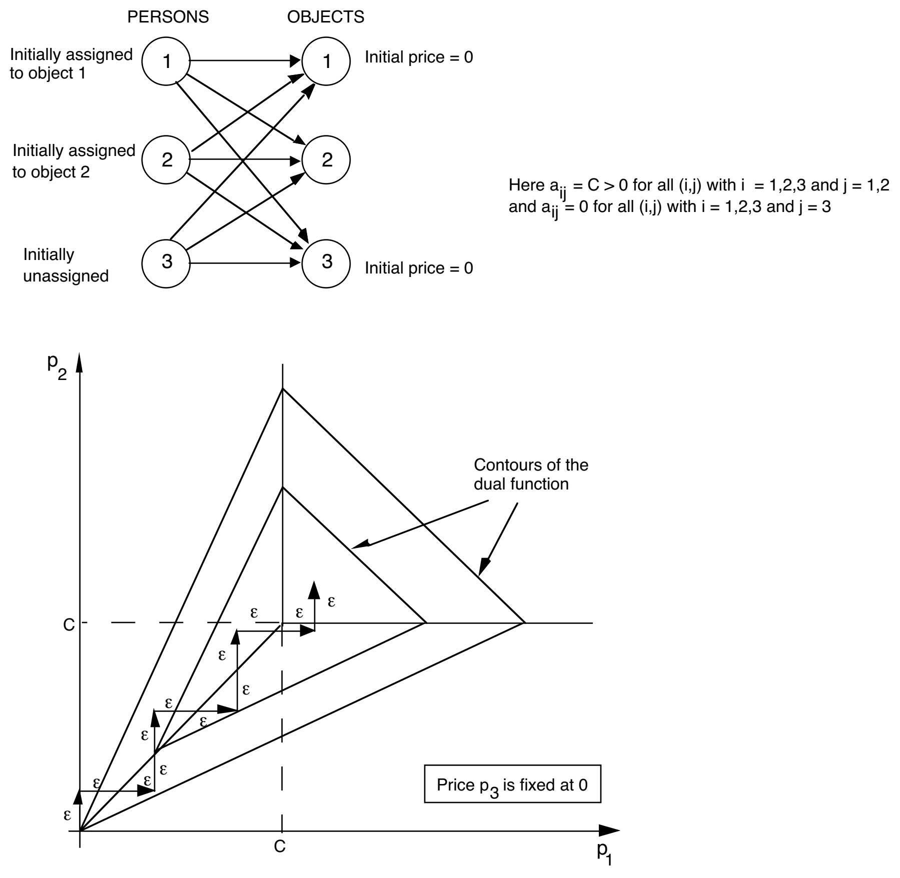

PERSONS
OBJECTS
Initially assigned to object 1
Initially assigned to object 2
Initially unassigned
1
2
3
1
2
3
Initial price = 0
Initial price = 0
Here a_{ij} = C > 0 for all (i,j) with i = 1,2,3 and j = 1,2
and a_{ij} = 0 for all (i,j) with i = 1,2,3 and j = 3
p_2
p_1
C
C
Contours of the dual function
ε
Price p_3 is fixed at 0

Figure 2.15 A sequence of prices p_1 and p_2 generated by the auction algorithm for the example of Figs. 2.12 and 2.14. The figure shows the equal dual cost surfaces in the space of p_1 and p_2 with p_3 fixed at 0.

Scaling

Figure 2.15 also illustrates a generic feature of auction algorithms. The amount of work needed to solve the problem can depend strongly on the value of ϵ and on the maximum absolute object benefit

$$C = \max_{(i,j)\in\mathcal{A}} |a_{ij}|.$$

Basically, for many types of problems, the number of iterations up to termination tends to be proportional to C/ϵ. This can be seen from the figure, where the total number of iterations is roughly C/ϵ, starting from zero initial prices.

Note also that there is a dependence on the initial prices; if these prices are “near optimal,” we expect that the number of iterations needed to solve the problem will be relatively small. This can be seen from the figure; if the initial prices satisfy $p_1 \approx p_3 + C$ and $p_2 \approx p_3 + C$, the number of iterations up to termination is quite small.

The preceding observations suggest the idea of *ϵ-scaling*, which consists of applying the algorithm several times, starting with a large value of ϵ and successively reducing ϵ until it is less than some critical value (for example, $1/n$, when a_{ij} are integer). Each application of the algorithm provides good initial prices for the next application. This is a common idea in nonlinear programming; it is encountered, for example, in barrier and penalty function methods; see e.g. [Ber82a], [Lue84]. An alternative form of scaling, called *cost scaling*, is based on successively representing a_{ij} with an increasing number of bits while keeping ϵ at a constant value.

In practice, scaling is typically beneficial, particularly for sparse assignment problems, that is, problems where the set of feasible assignment pairs is severely restricted.

Extension to the Minimum Cost Flow Problem

The ϵ-CS condition (2.22) can be generalized for the minimum cost flow problem. For a capacity-feasible flow vector x and a price vector p it takes the form

$$p_i - p_j \le a_{ij} + \epsilon \qquad \text{for all } (i,j) \in \mathcal{A} \text{ with } x_{ij} < c_{ij}, \tag{2.24a}$$

$$p_i - p_j \ge a_{ij} - \epsilon \qquad \text{for all } (i,j) \in \mathcal{A} \text{ with } b_{ij} < x_{ij}, \tag{2.24b}$$

[cf. Eq. (2.11)]. It will be shown in Section 4.1 (Prop. 4.1) that if the problem data are integer, if $\epsilon < 1/N$, where N is the number of nodes, and if x is feasible and satisfies the ϵ-CS condition (2.24) together with some p, then x is optimal.

The auction algorithm can also be generalized for the minimum cost flow problem; see Chapter 4. A broad generalization, called *generic auction algorithm*, is given in Section 4.4. It involves price increases and flow changes that preserve ϵ-CS. An interesting special case of the generic algorithm, called *ϵ-relaxation*, is discussed in Section 4.5. This algorithm may also be obtained by using the transformation of Section 1.1.3 to convert the minimum cost flow problem into an assignment problem and by applying the auction algorithm to this problem. We may view ϵ-relaxation as an approximate coordinate ascent method for maximizing the piecewise linear dual cost function (2.14) introduced in the preceding subsection; see Section 4.5.

1.2.5 Good, Bad, and Polynomial Algorithms

We have already discussed several types of methods, so the natural question arises: is there a best method and what criterion should we use to rank methods?

A practitioner who has a specific type of problem to solve, perhaps repeatedly, with the data and size of the problem within some limited range, will usually be interested in one or more of the following:

(a) Fast solution time.

(b) Flexibility to use good starting solutions (which the practitioner can usually provide, on the basis of his or her knowledge of the problem).

(c) The ability to perform sensitivity analysis (resolve the problem with slightly different problem data) quickly.

(d) The ability to take advantage of parallel computing hardware.

(e) Small memory requirements (this seems to be a diminishing concern nowadays).

Given the diversity of these considerations, it is not surprising that there is no algorithm that will dominate the others in all or even most practical situations. Otherwise expressed, every type of algorithm that we will discuss is best given the right type of practical problem. Thus, to make intelligent choices, the practitioner needs to understand the properties of different algorithms relating to speed of convergence, flexibility, parallelization, and suitability for specific problem structures. For challenging problems, the choice of algorithm is usually settled by experimentation with several candidates.

A theoretical analyst may also have difficulty ranking different algorithms for specific types of problems. The most common approach for this purpose is worst-case computational complexity analysis. Here one tries to bound the number of elementary numerical operations needed by a given algorithm with some measure of the "problem size," that is, with some expression of the form

$$Kf(N, A, C, U, S), \tag{2.25}$$

where

N is the number of nodes.

A is the number of arcs.

C is the arc cost range $\max_{(i,j)\in\mathcal{A}} |a_{ij}|$.

U is the maximum arc flow range $\max_{(i,j)\in\mathcal{A}}(c_{ij} - b_{ij})$.

S is the supply range $\max_{i\in\mathcal{N}} |s_i|$.

f is some known function.

K is a (usually unknown) constant.

If a bound of this form can be found, we say that the *running time* or *operation count of the algorithm is* $O\big(f(N,A,C,U,S)\big)$. If $f(N,A,C,U,S)$ can be written as a polynomial function of the number of bits needed to express the problem data, the algorithm is said to be *polynomial*. Examples of polynomial complexity bounds are $O\big(N^\alpha A^\beta\big)$ and $O\big(N^\alpha A^\beta \log C\big)$, where α and β are positive integers. The bound $O\big(N^\alpha A^\beta\big)$ is sometimes said to be *strongly polynomial* because it involves only the graph size parameters. A bound of the form $O\big(N^\alpha A^\beta C\big)$ is not polynomial because C is not a polynomial expression of $\log C$, the number of bits needed to express a single number of value C. Bounds like $O\big(N^\alpha A^\beta C\big)$, which are polynomial in the problem data rather than in the number of bits needed to express the data, are called *pseudopolynomial*.

A common assumption in theoretical computer science is that polynomial algorithms are "better" than pseudopolynomial, and pseudopolynomial algorithms are "better" than exponential (for example, those with a bound of the form $K2^{g(N,A)}$, where g is a polynomial in N and A). Furthermore, it is thought that two polynomial algorithms can be compared in terms of the degree of the polynomial bound; e.g., an $O(N^2)$ algorithm is "better" than an $O(N^3)$ algorithm. Unfortunately, quite often this assumption is not supported by computational practice in linear programming and network optimization. Pseudopolynomial and even exponential algorithms are often faster in practice than polynomial ones. In fact, the simplex method for general linear programs is an exponential algorithm [KlM72], [Chv83], and yet it is still used widely, because it performs very well in practice.

There are two main reasons why worst-case complexity estimates may fail to predict the practical performance of network flow algorithms. First, the upper bounds they provide may be very pessimistic as they may correspond to possible but highly unlikely problem instances. (Average complexity estimates would be more appropriate for such situations. However, obtaining these is usually hard, and the statistical assumptions underlying them may be inappropriate for many types of practical problems.) Second, worst-case complexity estimates involve the (usually unknown) constant K, which may dominate the estimate for all except for unrealistically large problem sizes. Thus, a comparison between two algorithms that is based on the size-dependent terms of running time estimates, and does not take into account the corresponding constants may be far from the mark.

This book is guided more by insights obtained through computational practice than by insights gained by estimating computational complexity. However, this is not to suggest that worst-case complexity analysis is useless; for all its unreliability, it has repeatedly proved its value by illuminating the computational bottlenecks of many algorithms and by stimulating the use of efficient data structures. For this reason, throughout the book, we will comment on available complexity estimates, and we will try to relate these estimates to computational practice. However, the treatment of complexity bounds is brief, and most of the corresponding proofs are omitted.

E X E R C I S E S

Exercise 2.1

Solve the max-flow problem of Fig. 2.16 using the Ford-Fulkerson method, where $s = 1$ and $t = 5$.

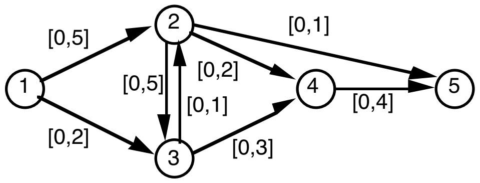


Figure 2.16 Max-flow problem for Exercise 2.1. The arc capacities are shown next to the arcs.

Exercise 2.2

Use ϵ-CS to verify that the assignment of Fig. 2.17 is optimal and obtain a bound on how far from optimal the given price vector is. State the dual problem and verify the correctness of the bound by comparing the dual value of the price vector with the optimal dual value.

Exercise 2.3

Consider the assignment problem.

(a) Show that every k-person swap can be accomplished with a sequence of $k - 1$ successive two-person swaps.

(b) In light of the result of part (a), how do you explain that a nonoptimal assignment may not be improvable by any two-person swap?

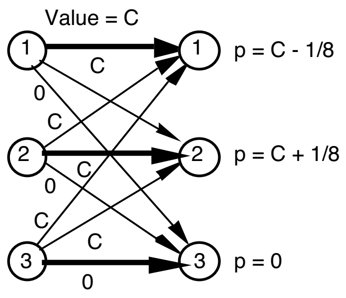


Figure 2.17 Assignment problem for Exercise 2.2. Objects 1 and 2 have value C for all persons. Object 3 has value 0 for all persons. Object prices are as shown. The thick lines indicate the given assignment.

Exercise 2.4 (Feasible Distribution Theorem)

Show that the minimum cost flow problem has a feasible solution if and only if $\sum_{i\in\mathcal{N}} s_i = 0$ and for every cut $Q = [\mathcal{S}, \mathcal{N} - \mathcal{S}]$ we have

$$\text{Capacity of } Q \ \geq \sum_{i\in\mathcal{S}} s_i.$$

Show also that feasibility of the problem can be determined by solving a max-flow problem with zero lower flow bounds. *Hint:* Assume first that all lower flow bounds b_{ij} are zero. Introduce two nodes s and t. For each node $i \in \mathcal{N}$ with $s_i > 0$ introduce an arc (s, i) with feasible flow range $[0, s_i]$, and for each node $i \in \mathcal{N}$ with $s_i < 0$ introduce an arc (i, t) with feasible flow range $[0, -s_i]$. Apply the max-flow/min-cut theorem. In the general case, transform the problem to one with zero lower flow bounds.

Exercise 2.5 (Finding a Feasible Flow Vector)

Show that one may find a feasible solution of a feasible minimum cost flow problem by solving a max-flow problem with zero lower flow bounds. Furthermore, if the supplies s_i and the arc flow bounds b_{ij} and c_{ij} are integer, show that the feasible solution found will be integer. *Hint:* Use the hint of Exercise 2.4.

Exercise 2.6 (Integer Approximations of Feasible Solutions)

Given a graph $(\mathcal{N}, \mathcal{A})$ and a flow vector x, show that there exists an integer flow vector $\overline{x}$ having the same divergence vector as x and satisfying

$$|x_{ij} - \overline{x}_{ij}| < 1, \qquad \forall\ (i, j) \in \mathcal{A}.$$

Hint: For each arc (i, j), define the integer flow bounds

$$b_{ij} = \lfloor x_{ij} \rfloor, \qquad c_{ij} = \lceil x_{ij} \rceil.$$

Use the result of Exercise 2.5.

Exercise 2.7 (Maximal Matching/Minimal Cover Theorem)

Consider a bipartite graph consisting of two sets of nodes $\mathcal{S}$ and $\mathcal{T}$ such that every arc has its start node in $\mathcal{S}$ and its end node in $\mathcal{T}$. A *matching* is a subset of arcs such that all the start nodes of the arcs are distinct and all the end nodes of the arcs are distinct. A maximal matching is a matching with a maximal number of arcs.

(a) Show that the problem of finding a maximal matching can be formulated as a max-flow problem.

(b) Define a *cover* $\mathcal{C}$ to be a subset of $\mathcal{S} \cup \mathcal{T}$ such that for each arc (i, j), either $i \in \mathcal{C}$ or $j \in \mathcal{C}$ (or both). A minimal cover is a cover with a minimal number of nodes. Show that the number of arcs in a maximal matching and the number of nodes in a minimal cover are equal. *Hint:* Use the max-flow/min-cut theorem.

Exercise 2.8 (Feasibility of an Assignment Problem)

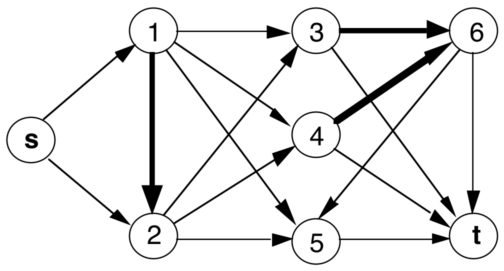
Show that an assignment problem is infeasible if and only if there exists a subset of person nodes I and a subset of object nodes J such that I has more nodes than J, and every arc with start node in I has an end node in J. *Hint:* Use the maximal matching/minimal cover theorem of the preceding exercise.

Exercise 2.9 (Ford-Fulkerson Method – Counterexample [Chv83])

This exercise illustrates how the version of the Ford-Fulkerson method where augmenting paths need not have as few arcs as possible may not terminate for a problem with irrational arc flow bounds. Consider the max-flow problem shown in Fig. 2.18.

(a) Verify that an infinite sequence of augmenting paths is characterized by the table of Fig. 2.18; each augmentation increases the divergence out of the source s but the sequence of divergences converges to a value which can be arbitrarily smaller than the maximum flow.

(b) Solve the problem with the Ford-Fulkerson method as given in Section 1.2.



After Iter. #	Augm. Path	x_{12}	x_{36}	x_{46}	x_{65}
$6k+1$	$(s,1,2,3,6,t)$	σ	$1-\sigma^{3k+2}$	$\sigma-\sigma^{3k+1}$	0
$6k+2$	$(s,2,1,3,6,5,t)$	$\sigma-\sigma^{3k+2}$	1	$\sigma-\sigma^{3k+1}$	σ^{3k+2}
$6k+3$	$(s,1,2,4,6,t)$	σ	1	$\sigma-\sigma^{3k+3}$	σ^{3k+2}
$6k+4$	$(s,2,1,4,6,3,t)$	$\sigma-\sigma^{3k+3}$	$1-\sigma^{3k+3}$	σ	σ^{3k+2}
$6k+5$	$(s,1,2,5,6,t)$	σ	$1-\sigma^{3k+3}$	σ	σ^{3k+4}
$6k+6$	$(s,2,1,5,6,4,t)$	$\sigma-\sigma^{3k+4}$	$1-\sigma^{3k+3}$	$\sigma-\sigma^{3k+4}$	0
$6(k+1)+1$	$(s,1,2,3,6,t)$	σ	$1-\sigma^{3(k+1)+2}$	$\sigma-\sigma^{3(k+1)+1}$	0

Figure 2.18 Max-flow problem illustrating that if the augmenting paths in the Ford-Fulkerson method do not have a minimum number of arcs, then the method may not terminate. All lower arc flow bounds are zero. The upper flow bounds are larger than one, with the exception of the thick-line arcs; these are arc $(3,6)$ which has upper flow bound equal to one, and arcs $(1,2)$ and $(4,6)$ which have upper flow bound equal to $\sigma = \big(-1+\sqrt{5}\big)/2$. (Note a crucial property of σ; it satisfies $\sigma^{k+2} = \sigma^k - \sigma^{k+1}$ for all integer $k \geq 0$.) The table gives a sequence of augmentations.

Exercise 2.10 (Termination of the Ford-Fulkerson Algorithm)

Consider the Ford-Fulkerson algorithm as described in Section 1.2.2. This exercice addresses the termination issue when the problem data are noninteger. Let x^0 be the initial feasible flow vector; let x^k, $k = 1, 2, \ldots$, be the flow vector after the kth augmentation; and let P_k be the corresponding augmenting path. An arc (i,j) is said to be a k^+*-bottleneck* if $(i,j) \in P_k^+$ and $x_{ij}^k = c_{ij}$, and it is said to be a k^-*-bottleneck* if $(i,j) \in P_k^-$ and $x_{ij}^k = b_{ij}$.

(a) Show that if $k < \overline{k}$ and an arc (i,j) is a k^+-bottleneck and a $\overline{k}^+$-

bottleneck, then for some m with $k < m < \overline{k}$ we must have $(i,j) \in P_m^-$. Similarly, if an arc (i,j) is a k^--bottleneck and a $\overline{k}^-$-bottleneck, then for some m with $k < m < \overline{k}$ we must have $(i,j) \in P_m^+$.

(b) Show that P_k is a path with a minimal number of arcs over all augmenting paths with respect to x^{k-1}. (This property depends on the implementation of the unblocked path search as a breadth-first search.)

(c) For any path P that is unblocked with respect to x^k, let $n_k(P)$ be the number of arcs of P, let $a_k^+(i)$ be the minimum of $n_k(P)$ over all unblocked P from s to i, and let $a_k^-(i)$ be the minimum of $n_k(P)$ over all unblocked P from i to t. Show that for all i and k we have

$$a_k^+(i) \leq a_{k+1}^+(i), \qquad a_k^-(i) \leq a_{k+1}^-(i).$$

(d) Show that if $k < \overline{k}$ and arc (i,j) is both a k^+-bottleneck and a $\overline{k}^+$-bottleneck, or is both a k^--bottleneck and a $\overline{k}^-$-bottleneck, then $a_k^+(t) < a_{\overline{k}}^+(t)$.

(e) Show that the algorithm terminates after $O(NA)$ augmentations, for an $O(NA^2)$ running time.

Exercise 2.11 (Duality for Nonnegativity Constraints)

Consider the version of the minimum cost flow problem where there are nonnegativity constraints

$$\text{minimize} \quad \sum_{(i,j)\in\mathcal{A}} a_{ij}x_{ij}$$

subject to

$$\sum_{\{j|(i,j)\in\mathcal{A}\}} x_{ij} - \sum_{\{j|(j,i)\in\mathcal{A}\}} x_{ji} = s_i, \qquad \forall\ i \in \mathcal{N},$$

$$0 \leq x_{ij}, \qquad \forall\ (i,j) \in \mathcal{A}.$$

Show that if a feasible flow vector x^* and a price vector p^* satisfy the following CS conditions

$$p_i^* - p_j^* \leq a_{ij}, \qquad \text{for all } (i,j) \in \mathcal{A},$$

$$p_i^* - p_j^* = a_{ij} \qquad \text{for all } (i,j) \in \mathcal{A} \text{ with } 0 < x_{ij}^*,$$

then x^* is optimal. Furthermore, p^* is an optimal solution of the following dual problem:

$$\text{maximize} \sum_{i\in\mathcal{N}} s_i p_i$$

$$\text{subject to} \quad p_i - p_j \leq a_{ij}, \qquad \forall\ (i,j) \in \mathcal{A}.$$

Hint: Complete the details of the following argument. Define

$$q(p) = \begin{cases} \sum_{i\in\mathcal{N}} s_i p_i & \text{if } p_i - p_j \le a_{ij},\ \forall\ (i,j)\in\mathcal{A} \\ -\infty & \text{otherwise} \end{cases}$$

and note that

$$\begin{aligned} q(p) &= \sum_{(i,j)\in\mathcal{A}} \min_{0\le x_{ij}} \big(a_{ij} + p_j - p_i\big) x_{ij} + \sum_{i\in\mathcal{N}} s_i p_i \\ &= \min_{0\le x} \left\{ \sum_{(i,j)\in\mathcal{A}} a_{ij} x_{ij} + \sum_{i\in\mathcal{N}} \left(s_i - \sum_{\{j\mid (i,j)\in\mathcal{A}\}} x_{ij} + \sum_{\{j\mid (j,i)\in\mathcal{A}\}} x_{ji} \right) p_i \right\}. \end{aligned}$$

Thus, for any feasible x and any p, we have

$$\begin{aligned} q(p) &\le \sum_{(i,j)\in\mathcal{A}} a_{ij} x_{ij} + \sum_{i\in\mathcal{N}} \left(s_i - \sum_{\{j\mid (i,j)\in\mathcal{A}\}} x_{ij} + \sum_{\{j\mid (j,i)\in\mathcal{A}\}} x_{ji} \right) p_i \\ &= \sum_{(i,j)\in\mathcal{A}} a_{ij} x_{ij}. \end{aligned} \tag{2.26}$$

On the other hand, we have

$$q(p^*) = \sum_{i\in\mathcal{N}} s_i p_i^* = \sum_{(i,j)\in\mathcal{A}} \big(a_{ij} + p_j^* - p_i^*\big) x_{ij}^* + \sum_{i\in\mathcal{N}} s_i p_i^* = \sum_{(i,j)\in\mathcal{A}} a_{ij} x_{ij}^*,$$

where the second equality is true because the CS conditions imply that $(a_{ij} + p_j^* - p_i^*) x_{ij}^* = 0$ for all $(i,j)\in\mathcal{A}$, and the last equality follows from the feasibility of x^*. Therefore, x^* attains the minimum of the primal cost on the right-hand side of Eq. (2.26). Furthermore, p^* attains the maximum of $q(p)$ on the left side of Eq. (2.26), which means that p^* is an optimal solution of the dual problem.

Exercise 2.12 (Node-Disjoint Paths)

Given two nodes i and j in a graph, consider the problem of finding the maximum number of paths starting at i and ending at j that are node-disjoint in the sense that any two of them share no nodes other than i and j. Formulate this problem as a max-flow problem.

Exercise 2.13 (Hall's Theorem of Distinct Representatives)

Given finite sets $S_1, S_2, \ldots, S_k$, we say that the collection $\{s_1, s_2, \ldots, s_k\}$ is a system of distinct representatives if $s_i \in S_i$ for all i and $s_i \neq s_j$ for $i \neq j$. (For example, if $S_1 = \{a, b, c\}$, $S_2 = \{a, b\}$, $S_1 = \{a\}$, then $s_1 = c$, $s_2 = b$, $s_3 = a$ is a system of distinct representatives). Show that there exists no system of distinct representatives if and only if there exists an index set $I \subset \{1, 2, \ldots, k\}$ such that the number of elements in $\cup_{i \in I} S_i$ is less than the number of elements in I. *Hint:* Consider a bipartite graph with each of the right side nodes representing an element of $\cup_{i \in I} S_i$, with each of the left side nodes representing one of the sets $S_1, S_2, \ldots S_k$, and with an arc from a left node S to a right node s if $s \in S$. Use the maximal matching/minimal cover theorem of Exercise 2.7.

Exercise 2.14

Prove the following generalization of Prop. 2.2. Let x be a capacity-feasible flow vector, and let $\mathcal{N}^+$ and $\mathcal{N}^-$ be two disjoint subsets of nodes. Then exactly one of the following two alternatives holds:

(1) There exists a path that starts at some node of $\mathcal{N}^+$, ends at some node of $\mathcal{N}^-$, and is unblocked with respect to x.

(2) There exists a saturated cut $Q = [\mathcal{S}, \mathcal{N} - \mathcal{S}]$ such that $\mathcal{N}^+ \subset \mathcal{S}$ and $\mathcal{N}^- \subset \mathcal{N} - \mathcal{S}$.

Exercise 2.15 (Duality and the Max-Flow/Min-Cut Theorem)

Consider a feasible max-flow problem and let $Q = [\mathcal{S}, \mathcal{N} - \mathcal{S}]$ be a minimum capacity cut separating s and t. Consider also the minimum cost flow problem formulation (1.8) for the max-flow problem (see Example 1.2). Show that the price vector

$$p_i = \begin{cases} 1 & \text{if } i \in \mathcal{S} \\ 0 & \text{if } i \notin \mathcal{S} \end{cases}$$

is an optimal solution of the dual problem. Furthermore, show that the max-flow/min-cut theorem expresses the equality of the primal and the dual optimal values. *Hint:* Relate the capacity of Q with the dual function value corresponding to p.

Exercise 2.16

Consider a feasible max-flow problem. Show that if the upper flow bound of each arc is increased by $\alpha > 0$, then the value of the maximum flow is increased by no more than αA, where A is the number of arcs.

Exercise 2.17 (Dual Cost Improvement Directions)

Consider the assignment problem. Let p_j be the price of object j, let T be a subset of objects, and let

$$S = \Big\{ i \mid \text{the maximum of } a_{ij} - p_j \text{ over } j \in A(i) \text{ is attained by some element of } T \Big\}.$$

Suppose that

(1) For each $i \in S$, the maximum of $a_{ij} - p_j$ over $j \in A(i)$ is attained only by elements of T.

(2) S has more elements than T.

Show that the direction $d = (d_1, \ldots, d_n)$, where $d_j = 1$ if $j \in T$ and $d_j = 0$ if $j \notin T$, is a direction of dual cost improvement. *Note*: Directions of this type are used by the most common dual cost improvement algorithms for the assignment problem.

1.3 THE SHORTEST PATH PROBLEM

The shortest path problem is a classical and important combinatorial problem that arises in many contexts. We are given a directed graph $(\mathcal{N}, \mathcal{A})$ with nodes numbered $1, \ldots, N$. Each arc $(i, j) \in \mathcal{A}$ has a cost or "length" a_{ij} associated with it. The length of a path $(i_1, i_2, \ldots, i_k)$, which consists exclusively of forward arcs, is equal to the length of its arcs

$$\sum_{n=1}^{k-1} a_{i_n i_{n+1}}.$$

This path is said to be *shortest* if it has minimum length over all paths with the same origin and destination nodes. The length of a shortest path is also called the *shortest distance*. The shortest distance from a node to itself is taken to be zero by convention. The shortest path problem deals with finding shortest distances between selected pairs of nodes. [Note that here we are optimizing over forward paths, that is, paths consisting of forward arcs; when we refer to a path (or a cycle) in connection with the shortest path problem, we implicitly assume that the path (or the cycle) is forward.]

All the major shortest path algorithms are based on the following simple proposition.

Proposition 3.1: Let $d = (d_1, d_2, \ldots, d_N)$ be a vector satisfying

$$d_j \le d_i + a_{ij}, \qquad \forall\ (i, j) \in \mathcal{A} \tag{3.1}$$

and let P be a path starting at a node i_1 and ending at a node i_k. If

$$d_j = d_i + a_{ij}, \qquad \text{for all arcs } (i,j) \text{ of } P \tag{3.2}$$

then P is a shortest path from i_1 to i_k.

Proof: By adding Eq. (3.2) over the arcs of P, we see that the length of P is $d_{i_k} - d_{i_1}$. By adding Eq. (3.1) over the arcs of any other path P' starting at i_1 and ending at i_k, we see that the length of P' must be at least equal to $d_{i_k} - d_{i_1}$. Therefore, P is a shortest path. **Q.E.D.**

The conditions (3.1) and (3.2) will be called the *complementary slackness (CS) conditions for the shortest path problem*. This terminology is motivated by the connection of the problem of finding a shortest path from i_1 to i_k with the following minimum cost flow problem

$$\text{minimize} \sum_{(i,j)\in\mathcal{A}} a_{ij}x_{ij} \tag{3.3}$$

subject to

$$\sum_{\{j|(i,j)\in\mathcal{A}\}} x_{ij} - \sum_{\{j|(j,i)\in\mathcal{A}\}} x_{ji} = s_i, \qquad \forall\, i \in \mathcal{N},$$

$$0 \le x_{ij}, \qquad \forall\, (i,j) \in \mathcal{A},$$

where

$$s_{i_1} = 1, \qquad s_{i_k} = -1, \qquad s_i = 0, \qquad \forall\, i \ne i_1, i_k.$$

It can be seen that a path P from i_1 to i_k is shortest if and only if the path flow x defined by

$$x_{ij} = \begin{cases} 1 & \text{if } (i,j) \text{ belongs to } P \\ 0 & \text{otherwise} \end{cases} \tag{3.4}$$

is an optimal solution of the minimum cost flow problem (3.3).

The CS conditions (3.1) and (3.2) of Prop. 3.1 are in effect the CS conditions for the equivalent minimum cost flow problem (3.3), which take the form

$$p_i \le a_{ij} + p_j, \qquad \forall\, (i,j) \in \mathcal{A}, \tag{3.5}$$

$$p_i = a_{ij} + p_j, \qquad \text{for all arcs } (i,j) \text{ with } 0 < x_{ij} \tag{3.6}$$

[cf. Eqs. (2.11c) and (2.11d)]. Indeed, if we associate the given path P in Prop. 3.1 with the flow vector of Eq. (3.4), and we identify p_i with $-d_i$, we see that the conditions (3.1) and (3.2) are identical to the CS conditions (3.5) and (3.6). Thus, the optimality of the path P under the conditions of Prop. 3.1 can also be inferred from the general result of Prop. 2.5, which asserts optimality of feasible pairs (x, p) satisfying CS. It also follows from the same general result that if a vector d satisfies the conditions of Prop. 3.1, then $p = -d$ is an optimal solution of the dual problem corresponding to the minimum cost flow problem (3.3).

Most shortest path algorithms can be viewed as primal cost or dual cost improvement algorithms for an appropriate variation of the minimum cost flow problem (3.3), as we will see later. However, the shortest path problem is simple, so we will discuss it first without much reference to cost improvement. This choice serves a dual purpose. First, it provides an opportunity to illustrate some basic concepts in the context of a simple problem, which is rich in intuition. Second, it allows the early development of some ideas and results that will be used later in a variety of other algorithmic contexts.

1.3.1 A General Single Origin/Many Destinations Shortest Path Method

The shortest path problem can be posed in a number of ways; for example, finding a shortest path from a single origin to a single destination, or finding a shortest path from each of several origins to each of several destinations. We will focus initially on the single origin/many destinations problem. For concreteness, we take the origin node to be node 1.

Let us now describe a prototype shortest path method that contains several interesting algorithms as special cases. In this method, we start with some vector $(d_1, d_2, \ldots, d_N)$, we successively select arcs (i, j) that violate the CS condition (3.1), that is, $d_j > d_i + a_{ij}$, and we set

$$d_j := d_i + a_{ij}.$$

This is continued until the CS condition $d_j \leq d_i + a_{ij}$ is satisfied for all arcs (i, j).

A key idea is that, in the course of the algorithm, d_i can be interpreted for all i as the length of some path P_i from 1 to i. Therefore, if $d_j > d_i + a_{ij}$ for some arc (i, j), the path obtained by extending path P_i by arc (i, j), which has length $d_i + a_{ij}$, is a better path than the current path P_j, which has length d_j. Thus, the algorithm finds successively better paths from the origin to various destinations.

It should be noted that replacing the current path P_j with the shorter path consisting of P_i followed by the arc (i, j), as discussed above, is essentially a *primal cost improvement operation*; in the context of a minimum cost flow formulation of the many destinations shortest path problem [cf. Eq. (3.3)], it can be interpreted as pushing one unit of flow along the cycle that starts at 1, traverses P_i and (i, j) in the forward direction, and then traverses P_j in the backward direction. The cost of this cycle, as defined earlier in Section 1.2.1, is equal to the length of P_i, plus the length of (i, j), minus the length of P_j, and is therefore negative. Thus the general algorithm of this section can be viewed as a primal cost improvement algorithm. It will be seen in Chapter 3 (Exercise 2.3) that an important special case, Dijkstra's method to be discussed shortly, can also be viewed as a dual cost improvement algorithm. Another algorithm, the auction/shortest path algorithm to be presented in Section 4.3, does not fit the framework of the present section (even though it crucially depends on the CS conditions of Prop. 3.1); it will be shown to be a dual cost improvement algorithm.

It is usually most convenient to implement the prototype shortest path method by examining the outgoing arcs of a given node i consecutively. The corresponding algorithm, referred to as *generic*, maintains a list of nodes V, called the *candidate list*, and a vector $d = (d_1, d_2, \ldots, d_N)$, where each d_j, called the *label of node* j, is either a real number or ∞. Initially,

$$V = \{1\}, \tag{3.7}$$

$$d_1 = 0, \qquad d_i = \infty, \qquad \forall\ i \neq 1. \tag{3.8}$$

The algorithm proceeds in iterations and terminates when V is empty. The typical iteration (assuming V is nonempty) is as follows:

Typical Iteration of the Generic Shortest Path Algorithm

Remove a node i from the candidate list V. For each outgoing arc $(i, j) \in \mathcal{A}$, with $j \neq 1$, if $d_j > d_i + a_{ij}$, set

$$d_j := d_i + a_{ij} \tag{3.9}$$

and add j to V if it does not already belong to V.

It can be seen that, in the course of the algorithm, the labels are monotonically nonincreasing. Furthermore, we have

$$d_i < \infty \qquad \Longleftrightarrow \qquad i \text{ has entered the candidate list } V \text{ at least once.}$$

Figure 3.1 illustrates the algorithm. The following proposition gives its main properties.

Proposition 3.2: Consider the generic shortest path algorithm.

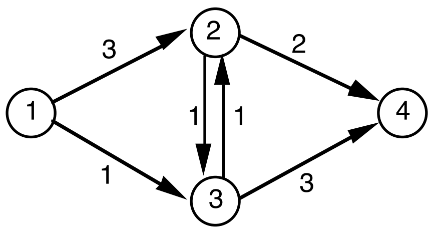
(a) At the end of each iteration, the following conditions hold:

   (i) $d_1 = 0$.

   (ii) If $d_j < \infty$ and $j \neq 1$, then d_j is the length of some path that starts at 1, never returns to 1, and ends at j.

   (iii) If $i \notin V$, then either $d_i = \infty$ or else

$$d_j \leq d_i + a_{ij}, \qquad \forall\ j \text{ such that } (i, j) \in \mathcal{A}.$$



Iteration #	**Candidate List V**	**Node Labels**	**Node out of V**
1	$\{1\}$	$(0, \infty, \infty, \infty)$	1
2	$\{2, 3\}$	$(0, 3, 1, \infty)$	2
3	$\{3, 4\}$	$(0, 3, 1, 5)$	3
4	$\{4, 2\}$	$(0, 2, 1, 4)$	4
5	$\{2\}$	$(0, 2, 1, 4)$	2
	$\emptyset$	$(0, 2, 1, 4)$	

Figure 3.1 Illustration of the generic shortest path algorithm. The numbers next to the arcs are the arc lengths. Note that node 2 enters the candidate list twice. If in iteration 2 node 3 was removed from V instead of node 2, each node would enter V only once. Thus, the order in which nodes are removed from V is significant.

(b) If the algorithm terminates, then upon termination, for all $j \neq 1$ such that $d_j < \infty$, d_j is the shortest distance from 1 to j and

$$d_j = \min_{(i,j) \in \mathcal{A}} \{d_i + a_{ij}\}; \tag{3.10}$$

furthermore, $d_j = \infty$ if and only if there is no path from 1 to j.

(c) If the algorithm does not terminate, then there exist paths of arbitrarily small (i.e., large negative) length that start at 1 and never return to 1.

Proof: (a) Condition (i) holds because initially $d_1 = 0$, and by the rules of the algorithm, d_1 cannot change.

We prove (ii) by induction on the iteration count. Indeed, initially (ii) holds, since node 1 is the only node j with $d_j < \infty$. Suppose that (ii) holds at the start of some iteration at which a node i is removed from V. If $i = 1$, which happens only at the first iteration, then at the end of the iteration we have $d_j = a_{1j}$ for all outward neighbors j of 1, and $d_j = \infty$ for all other $j \neq 1$, so d_j has the required property. If $i \neq 1$, then $d_i < \infty$ (which is true for all nodes of V by the rules of the algorithm), and (by the induction hypothesis) d_i is the length of some path P_i starting at 1, never returning to 1, and ending at i. When a label d_j changes as a result of the iteration, d_j is set to $d_i + a_{ij}$, which is the length of the path P_j consisting of P_i followed by arc (i, j). Since $j \neq 1$, P_j never returns to 1. This completes the induction proof of (ii).

To prove (iii), note that for any i, each time i is removed from V, the condition $d_j \leq d_i + a_{ij}$ is satisfied for all $(i, j) \in \mathcal{A}$ by the rules of the algorithm. Up to the next entrance of i into V, d_i stays constant, while the labels d_j for all j with $(i, j) \in \mathcal{A}$ cannot increase, thereby preserving the condition $d_j \leq d_i + a_{ij}$.

(b) We first introduce the sets

$$I = \{i \mid d_i < \infty \text{ upon termination}\},$$
$$\overline{I} = \{i \mid d_i = \infty \text{ upon termination}\},$$

and we show that we have $d_i \in \overline{I}$ if and only if there is no path from 1 to j. Indeed, if $i \in I$, then, since $i \notin V$ upon termination, it follows from condition (iii) of part (a) that $j \in I$ for all $(i, j) \in \mathcal{A}$. Therefore, if $j \in \overline{I}$, there is no path from any node of I (and in particular, node 1) to node j. Conversely, if there is no path from 1 to j, it follows from condition (ii) of part (a) that we cannot have $d_j < \infty$ upon termination, so $j \in \overline{I}$.

We show now that for all $i \in I$, we have $d_j = \min_{(i,j)\in\mathcal{A}}\{d_i + a_{ij}\}$ upon termination. Indeed, conditions (ii) and (iii) of part (a) imply that upon termination we have, for all $i \in I$,

$$d_j \leq d_i + a_{ij}, \qquad \forall\ j \text{ such that } (i, j) \in \mathcal{A}$$

while d_i is the length of some path P_i from 1 to i. Fix a node $m \in I$. By adding this condition over the arcs (i, j) of any path P from 1 to m, we see that the length of P is no less than d_m. Hence P_m is a shortest path from 1 to m. Furthermore, the equality $d_j = d_i + a_{ij}$ must hold for all arcs (i, j) on the shortest paths P_m, $m \in I$, implying that $d_j = \min_{(i,j)\in\mathcal{A}}\{d_i + a_{ij}\}$.

(c) If the algorithm never terminates, some label d_j must decrease strictly an infinite number of times, generating a corresponding sequence of distinct paths P_j as per condition (ii) of part (b). Each of these paths can be decomposed into a simple path from 1 to j plus a collection of simple cycles, as in Exercise 1.5. Since the number of simple paths from 1 to j is finite, and the length of the path P_j is monotonically decreasing, it follows that P_j eventually must involve a cycle with negative length. By replicating this cycle a sufficiently large number of times, one can obtain paths from 1 to j with arbitrarily small length. **Q.E.D.**

Termination and the Existence of Negative Length Cycles

So far we have imposed no assumptions on the structure of the graph of the problem or the lengths of the arcs. Thus, Prop. 3.2 does not guarantee that the algorihm will terminate. On the other hand, Prop. 3.2 shows that the generic algorithm will terminate if and only if there is a lower bound on the length of all paths that start at node 1 and never return to node 1. Thus, the algorithm will terminate if and only if there is no path starting at node 1, never returning to 1, and containing a cycle with negative length. One can detect the presence of such a cycle (and stop the algorithm) once some label d_j becomes less than $(N-1)\min_{(i,j)\in\mathcal{A}} a_{ij}$, which is a lower bound to the length of all simple paths.

Bellman's Equation and Shortest Path Construction

When all cycles have nonnegative length and there exists a path from 1 to every node j, then Prop. 3.2 shows that the generic algorithm terminates and that, upon termination, all labels are finite and satisfy

$$d_j = \min_{(i,j)\in\mathcal{A}}\{d_i + a_{ij}\}, \qquad \forall\ j \neq 1, \tag{3.11a}$$

$$d_1 = 0. \tag{3.11b}$$

This equation, which is in effect the CS conditions of Prop. 3.1, is called *Bellman's equation*. It expresses that the shortest distance from 1 to j is the sum of the shortest distance from 1 to the node preceding j on the shortest path, plus the length of the arc connecting that node to j.

From Bellman's equation, we can obtain the shortest paths (as opposed to the shortest path lengths) if all cycles not including node 1 have strictly positive length. To do this, select for each $j \neq 1$ one arc (i,j) that attains the minimum in $d_j = \min_{(i,j)\in\mathcal{A}}\{d_i + a_{ij}\}$ and consider the subgraph consisting of these $N-1$ arcs; see Fig. 3.2. To find the shortest path to any node j, start from j and follow the corresponding arcs of the subgraph backward until node 1 is reached. Note that the same node cannot be reached twice before node 1 is reached, since a cycle would be formed that [on the basis of Eq. (3.11)] would have zero length. [Let $(i_1, i_2, \ldots, i_k, i_1)$ be the cycle and add the equations

$$d_{i_1} = d_{i_2} + a_{i_2 i_1}$$

$$\ldots$$

$$d_{i_{k-1}} = d_{i_k} + a_{i_k i_{k-1}}$$

$$d_{i_k} = d_{i_1} + a_{i_1 i_k},$$

obtaining $a_{i_2 i_1} + \cdots + a_{i_k i_{k-1}} + a_{i_1 i_k} = 0$.] Since the subgraph is connected and has $N-1$ arcs, it must be a spanning tree. We call this subgraph a *shortest path spanning tree*, and we note that it has the special structure of having a root (node 1), with every arc of the tree directed away from the root. The preceding argument can also be used to show that Bellman's equation has no solution other than the shortest distances; see Exercise 3.12.

A shortest path spanning tree can also be constructed in the process of executing the generic shortest path algorithm by recording the arc (i,j) every time d_j is decreased to $d_i + a_{ij}$; see Exercise 3.3.

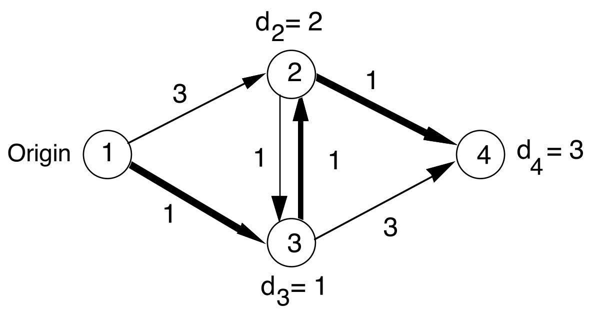


Figure 3.2 Example of construction of shortest path spanning tree. The arc lengths are shown next to the arcs, and the shortest distances are shown next to the nodes. For each $j \neq 1$, we select an arc (i,j) such that

$$d_j = d_i + a_{ij}$$

and we form the shortest path spanning tree. The arcs selected in this example are $(1,3)$, $(3,2)$, and $(2,4)$.

Implementations of the Generic Algorithm

There are many implementations of the generic algorithm; they differ in how they select the node to be removed from the candidate list V. They are broadly divided into two categories:

(a) *Label setting methods.* In these methods, the node i removed from V is a node with minimum label. Under the assumption that *all arc lengths are nonnegative*, these methods have a remarkable property: each node will enter V at most *once*; its label has its permanent or final value the first time it is removed from V. The most time consuming part of these methods is calculating the minimum label node from V at each iteration;

there are several implementations, that use a variety of creative methods to calculate this minimum.

(b) *Label correcting methods.* In these methods the choice of the node i removed from V is less sophisticated than in label setting methods, and requires less calculation. However, a node may enter V multiple times.

Generally in practice, when the arc lengths are nonnegative, the best label setting methods and the best label correcting methods are competitive. There are also several worst case complexity bounds for label setting and label correcting methods. The best bounds correspond to label setting methods. The best practical methods, however, are not necessarily the ones with the best complexity bounds, as will be discussed shortly.

1.3.2 Label Setting (Dijkstra) Methods

The basic label setting method, first published by Dijkstra [Dij59] but also discovered independently by several other researchers, is the special case of the generic algorithm where the node j removed from the candidate list V at each iteration has minimum label, that is,

$$d_j = \min_{i \in V} d_i.$$

For convenient reference, let us state this method explicitly.

Initially, we have

$$V = \{1\}, \tag{3.12}$$

$$d_1 = 0, \qquad d_i = \infty, \qquad \forall\, i \neq 1. \tag{3.13}$$

The method proceeds in iterations and terminates when V is empty. The typical iteration (assuming V is nonempty) is as follows:

Typical Iteration of the Label Setting Method

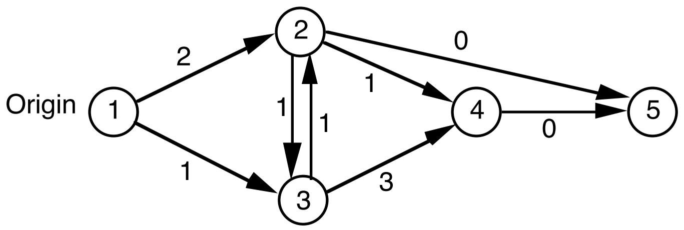
Remove from the candidate list V a node i such that

$$d_i = \min_{j \in V} d_j.$$

For each outgoing arc $(i, j) \in \mathcal{A}$, with $j \neq 1$, if $d_j > d_i + a_{ij}$, set

$$d_j := d_i + a_{ij} \tag{3.14}$$

and add j to V if it does not already belong to V.



Iteration #	**Candidate List V**	**Node Labels**	**Node out of V**
1	$\{1\}$	$(0,\infty,\infty,\infty,\infty)$	1
2	$\{2,3\}$	$(0,2,1,\infty,\infty)$	3
3	$\{2,4\}$	$(0,2,1,4,\infty)$	2
4	$\{4,5\}$	$(0,2,1,3,2)$	5
5	$\{4\}$	$(0,2,1,3,2)$	4
	$\emptyset$	$(0,2,1,3,2)$	

Figure 3.3 Example illustrating the label setting method. At each iteration, the node with the minimum label is removed from V. Each node enters V only once.

Figure 3.3 illustrates the label setting method.

Some insight into the label setting method can be gained by considering the set W of nodes that have already been in V but are not currently in V,

$$W = \{i \mid d_i < \infty, i \notin V\}. \tag{3.15}$$

We will see that as a consequence of the policy of removing from V a minimum label node, W contains nodes with "small" labels throughout the algorithm, in the sense that

$$d_j \leq d_i, \qquad \text{if } j \in W \text{ and } i \notin W. \tag{3.16}$$

On the basis of this property and the assumption $a_{ij} \geq 0$, it can be seen that when a node i is removed from V, we have, for all $j \in W$ for which (i,j) is an arc,

$$d_j \leq d_i + a_{ij}.$$

Hence, once a node enters W, it stays in W and its label does not change further. Thus, W can be viewed as the set of *permanently labeled nodes*, that is, the nodes that have acquired a final label, which by Prop. 3.2, must be equal to their shortest distance from the origin.

To understand why the property (3.16) is preserved, consider an iteration in which node i is removed from V, and assume that Eq. (3.16) holds at the start of the iteration. Then, any label d_j that changes during the iteration must correspond to a node $j \notin W$ (as was argued below), and at the end of the iteration it must satisfy $d_j = d_i + a_{ij} \geq d_i \geq d_k$ for all $k \in W$, thereby maintaining Eq. (3.16).

The following proposition makes the preceding arguments more precise and proves some additional facts.

Proposition 3.3: Assume that all arc lengths are nonnegative and that there exists at least one path from node 1 to each other node.

(a) For any iteration of the label setting method, the following hold for the set
$$W = \{i \mid d_i < \infty, i \notin V\}.$$

  (i) No node belonging to W at the start of the iteration will enter the candidate list V during the iteration.

  (ii) At the end of the iteration, we have $d_i \leq d_j$ for all $i \in W$ and $j \notin W$.

  (iii) For each node i, consider paths that start at 1, end at i, and have all their other nodes in W at the end of the iteration. Then the label d_i at the end of the iteration is equal to the length of the shortest of these paths ($d_i = \infty$ if no such path exists).

(b) In the label setting method, all nodes will be removed from the candidate list V exactly once in order of increasing shortest distance from node 1; that is, i will be removed before j if the final labels satisfy $d_i < d_j$.

Proof: (a) Properties (i) and (ii) will be proved simultaneously by induction on the iteration count. Clearly (i) and (ii) hold for the initial iteration at which node 1 exits V and enters W.

Suppose that (i) and (ii) hold for iteration $k-1$, and suppose that during iteration k, node i satisfies $d_i = \min_{j \in V} d_j$ and exits V. Let W and $\overline{W}$ be the set of Eq. (3.15) at the start and at the end of iteration k, respectively. Let d_j and $\overline{d}_j$ be the label of each node j at the start and at the end of iteration k, respectively. Since by the induction hypothesis we have $d_j \leq d_i$ for all $j \in W$, and $a_{ij} \geq 0$ for all arcs (i,j), it follows that $d_j \leq d_i + a_{ij}$ for all arcs (i,j) with $j \in W$. Hence, a node $j \in W$ cannot enter V at iteration k. This completes the induction proof of property (i), and shows that
$$\overline{W} = W \cup \{i\}.$$

Thus, at iteration k, the only labels that may change are the labels d_j of nodes $j \notin \overline{W}$ such that (i, j) is an arc; the label $\overline{d}_j$ at the end of the iteration will be $\min\{d_j, d_i + a_{ij}\}$. Since $a_{ij} \geq 0$, $d_i \leq d_j$ for all $j \notin W$, and $d_i = \overline{d}_i$, we must have $\overline{d}_i \leq \overline{d}_j$ for all $j \notin \overline{W}$. Since by the induction hypothesis we have $d_m \leq d_i$ and $d_m = \overline{d}_m$ for all $m \in \overline{W}$, it follows that $\overline{d}_m \leq \overline{d}_j$ for all $m \in \overline{W}$ and $j \notin \overline{W}$. This completes the induction proof of property (ii).

To prove property (iii), choose any node i and consider the subgraph consisting of the nodes $W \cup \{i\}$ together with the arcs that have both end nodes in $W \cup \{i\}$. Consider also a modified shortest path problem involving this subgraph and the same origin and arc lengths as in the original shortest path problem. In view of properties (i) and (ii), the label setting method applied to the modified shortest path problem yields the same sequence of nodes exiting V and the same sequence of labels as when applied to the original problem up to the current iteration. By Prop. 3.2, the label setting method for the modified problem terminates with the labels equal to the shortest distances of the modified problem at the current iteration. This means that the labels at the end of the iteration have the property stated in the proposition.

(b) By Prop. 3.2, we see that, under our assumptions, the label setting method will terminate with all labels finite. Therefore, each node will enter V at least once. At each iteration the node removed from V is added to W, and according to property (i) (proved above), no node from W is ever returned to V. Therefore, each node will be removed from V and simultaneously entered in W exactly once, and, by the rules of the algorithm, its label cannot change after its entrance in W. Property (ii) then shows that each new node added to W has a label at least as large as the labels of the nodes already in W. Therefore, the nodes are removed from V in the order stated in the proposition. **Q.E.D.**

Performance and Implementations of the Label Setting Method

In label setting methods, the candidate list V is typically maintained with the help of some data structure that facilitates the removal and the addition of nodes, and also facilitates finding the minimum label node from the list. The choice of data structure is crucial for good practical performance as well as for good theoretical worst case performance.

To gain some insight into this, we first consider a naive implementation that will serve as a yardstick for comparison. By Prop. 3.3, there will be exactly N iterations, and in each of these the candidate list V will be searched for a minimum label node. Suppose this is done by examining all nodes in sequence, checking whether they belong to V, and finding one with minimum label among those who do. Searching V in this way requires $O(N)$ operations per iteration, for a total of $O(N^2)$ operations. Also during the algorithm, we must examine each arc (i,j) exactly once to check whether $j \neq 1$ or whether the condition $d_j > d_i + a_{ij}$ holds, and to set $d_j := d_i + a_{ij}$ if it does. This requires $O(A)$ operations, which is dominated by the preceding $O(N^2)$ estimate.

The $O(A)$ operation count for arc examination is unavoidable and cannot be reduced. However, the $O(N^2)$ operation count for minimum label searching can be reduced considerably by using appropriate data structures. The best estimates of the worst case running time that have been thus obtained are $O(A + N \log N)$ and $O(A + N\sqrt{\log C})$, where C is the arc length range $C = \max_{(i,j)\in\mathcal{A}} a_{ij}$; see [FrT84], [AMO88]. On the basis of present experience, however, the methods that perform best in practice have far worse running time estimates. We will discuss two of these methods.

Binary Heap Method

Here the nodes are organized as a binary heap on the basis of label values and membership in V; see Fig. 3.4. The node at the top of the heap is the node of V that has minimum label, and the label of every node in V is no larger than the labels of all the nodes that are in V and are its descendants in the heap. Nodes that are not in V may be in the heap but may have no descendants that are in V.

At each iteration, the top node of the heap is removed from V. Furthermore, the labels of some nodes already in V may decrease, so these may have to be repositioned in the heap; also, some other nodes may enter V for the first time and have to be inserted in the heap at the right place. It can be seen that each of these removals, repositionings, and insertions can be done in $O(\log N)$ time. Since there is one removal per iteration, and at most one repositioning or node insertion per arc (each arc is examined at most once), the total operation count for maintaining the heap is $O(A \log N)$. This dominates the $O(A)$ operation count to examine all arcs, so the worst case running time of the method is $O(A \log N)$. For sparse graphs, where $A << N^2$, the binary heap method performs very well in practice.

Dial's Algorithm [Dia69]

This algorithm requires that all arc lengths be nonnegative integers. It uses a naive yet often surprisingly effective method for finding the minimum label node in V. We first note that, since every finite label is equal to the length of some path with no cycles [Prop. 3.3(a), part (iii)], the possible label values range from 0 to $(N-1)C$, where

$$C = \max_{(i,j)\in\mathcal{A}} a_{ij}.$$

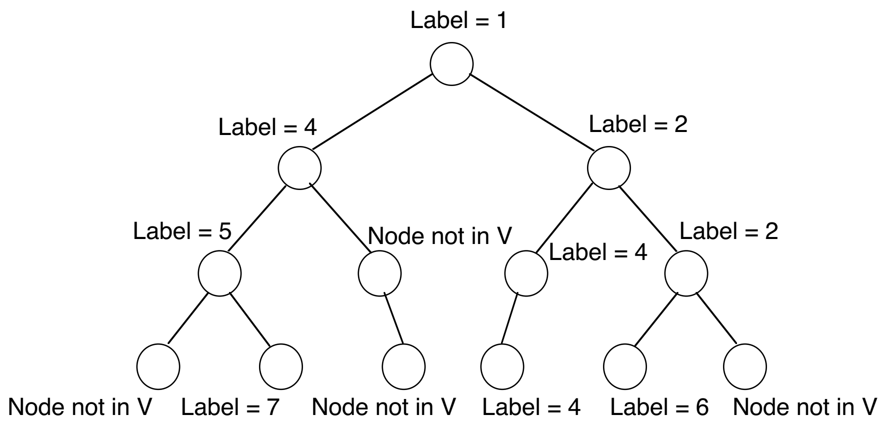


Figure 3.4 A binary heap organized on the basis of node labels is a binary balanced tree such that the label of each node of V is no larger than the labels of all its descendants that are in V. Nodes that are not in V may have no descendants that are in V. The topmost node, called the *root*, has the minimum label. The tree is balanced in that the numbers of arcs in the paths from the root to any nodes with no descendants differ by at most 1. If the label of some node decreases, the node must be moved upward toward the root, requiring $O(\log N)$ operations. [It takes $O(1)$ operations to compare the label of a node i with the label of one of its descendants j, and to interchange the positions of i and j if the label of j is smaller. Since there are $\log N$ levels in the tree, it takes at most $\log N$ such comparisons and interchanges to move a node upward to the appropriate position once its label is decreased.] Similarly, when the topmost node is removed from V, moving the node downward to the appropriate level in the heap requires at most $\log N$ steps and $O(\log N)$ operations. (Each step requires the interchange of the position of the node and the position of one of its descendants. The descendant must be in V for the step to be executed; if both descendants are in V, the one with smaller label is selected.)

Suppose that for each possible label value, we keep a list of the nodes that have this label value. Then we may scan the $(N-1)C+1$ possible label values (in ascending order) looking for a label value with nonempty list, instead of scanning the candidate list V. As will be seen shortly, this leads to a worst case operation count of $O(NC)$ for minimum label node searching, and to an $O(A+NC)$ operation count overall. The algorithm is pseudopolynomial, but for small values of C (much smaller than N) it performs very well in practice.

To visualize the algorithm, it is useful to think of each integer in the range $[0,(N-1)C]$ as some kind of container, referred to as a *bucket*. Each bucket b holds the nodes with label equal to b. A data structure such as

Bucket b	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Contents	**3**	–	**1,4,5**	**2,7**	–	**6**	–	–	–
FIRST(b)	**3**	**0**	**1**	**2**	**0**	**6**	**0**	**0**	**0**

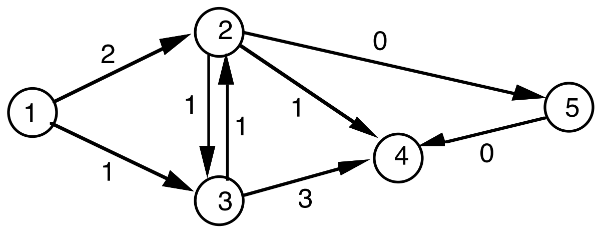
Node i	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Label d_i	**2**	**3**	**0**	**2**	**2**	**5**	**3**
NEXT(i)	**4**	**7**	**0**	**5**	**0**	**0**	**0**
PREVIOUS(i)	**0**	**0**	**0**	**1**	**4**	**0**	**2**

Figure 3.5 Organization of the candidate list V in buckets using a doubly linked list. For each bucket b we maintain the first node of the bucket in an array element *FIRST*(b), where *FIRST*$(b) = 0$ if bucket b is empty. For every node i we maintain two array elements, *NEXT*(i) and *PREVIOUS*(i), giving the next node and the preceding node, respectively, of node i in the bucket where i is curently residing [*NEXT*$(i) = 0$ or *PREVIOUS*$(i) = 0$ if i is the last node or the first node in its bucket, respectively]. In this example, there are 7 nodes and 8 buckets.

a doubly linked list (see Fig. 3.5) can be used to maintain the set of nodes belonging to a given bucket, so that checking the emptiness of a bucket and inserting or removing a node from a bucket are easy, requiring $O(1)$ operations.

Figure 3.6 illustrates the method with an example. Tracing steps, we see that the method starts with the origin node 1 in bucket 0 and all other buckets empty. At the first iteration, each node j with $(1, j) \in \mathcal{A}$ enters the candidate list V and is inserted in bucket $d_j = a_{1j}$. If for some j we have $d_j = 0$, then node j is inserted in bucket 0, and is removed next from V. After we are done with bucket 0, we proceed to check bucket 1. If it is nonempty, we repeat the process, removing from V all nodes with label 1 and moving other nodes to smaller numbered buckets as required; if not, we check bucket 2, and so on.

We note that it is sufficient to maintain only $C + 1$ buckets, rather than $(N - 1)C + 1$, thereby significantly saving in memory. The reason is that if



Iter. #	Cand. List V	Node Labels	Buck. 0	Buck. 1	Buck. 2	Buck. 3	Buck. 4	Out of V
1	$\{1\}$	$(0,\infty,\infty,\infty,\infty)$	1	–	–	–	–	1
2	$\{2,3\}$	$(0,2,1,\infty,\infty)$	1	3	2	–	–	3
3	$\{2,4\}$	$(0,2,1,4,\infty)$	1	3	2	–	4	2
4	$\{4,5\}$	$(0,2,1,3,2)$	1	3	2,5	4	–	5
5	$\{4\}$	$(0,2,1,2,2)$	1	3	2,4,5	–	–	4
	$\emptyset$	$(0,2,1,2,2)$	1	3	2,4,5	–	–	

Figure 3.6 An example illustrating Dial's method.

we are currently searching bucket b, then all buckets beyond $b+C$ are known to be empty. To see this, note that the label d_j of any node j must be of the form $d_i + a_{ij}$, where i is a node that has already been removed from the candidate list. Since $d_i \leq b$ and $a_{ij} \leq C$, it follows that $d_j \leq b + C$.

The idea of using buckets to maintain the nodes of the candidate list can be generalized considerably. In particular, buckets of width larger than 1 may be used. This results in fewer buckets to search over, thereby alleviating the $O(NC)$ bottleneck of the operation count of the algorithm. There is a price for this, namely the need to search for a minimum label node within the current bucket. This search can be speeded up by using buckets with nonuniform widths, and by breaking down buckets of large width into buckets of smaller width at the right moment. With intelligent strategies of this type, one may obtain label setting methods with very good polynomial complexity bounds; see [Joh77], [DeF79], [AMO88].

1.3.3 Label Correcting Methods

In these methods, the selection of the node to be removed from the candidate list V is faster than in label setting methods, at the expense of multiple entrances of nodes in V.

All of these methods use some type of queue to maintain the candidate list V. They differ in the way the queue is structured, and in the choice of the queue position into which nodes are entered.

The simplest of these methods, operates in cycles of iterations. In each cycle the nodes are scanned in some order; when a node i is found to belong to V, an iteration removing i from V is performed. This is a variant of one of the first methods proposed for the shortest path problem, known as the *Bellman-Ford method*. It is possible to show that if all cycles have nonnegative length this method requires at most N cycles; see Exercise 3.4. Each cycle consists of at most N iterations, requiring a total of $O(A)$ operations (each arc is examined at most once in each cycle). Thus, the total operation count for the method is $O(NA)$.

The best practical implementations of label correcting methods are more sophisticated than the one just described. Their worst case complexity bound is no better than the $O(NA)$ bound for the simple implementation derived above, and in some cases it is far worse. Yet their practical performance is far better.

The D'Esopo-Pape Algorithm

In this method, a node is always removed from the top of the queue used to store the candidate list V. A node, upon entrance in the queue, is placed at the bottom of the queue if it has never been in the queue before; otherwise it is placed at the top. The idea here is that when a node i is removed from the queue, its label affects the labels of a subset B_i of the neighbor nodes j with $(i, j) \in \mathcal{A}$. When the label of i changes again, it is likely that the labels of the nodes in B_i will require updating also. It is thus intuitively sensible to place the node at the top of the queue so that the labels of the nodes in B_i get a chance to be updated as quickly as possible.

The D'Esopo-Pape algorithm is very simple to implement and performs very well in practice for a broad variety of problems. Despite this fact, examples have been constructed [Ker81], [ShW81], where it performs very poorly. In particular, in these examples, the number of times some nodes enter the candidate list V is not polynomial. References [Pal84] and [GaP88] give a polynomial variation of the algorithm, which is the basis for the code of Appendix A.2.

The Threshold Algorithm

The premise of this algorithm is that *it is generally a good policy to remove from the candidate list a node with relatively small label*. When the arc lengths are nonnegative, this policy tends to reduce the number of times a node reenters the candidate list. In particular, when the node with smallest label is removed from the candidate list, as in Dijkstra's algorithm, this node never reenters the list; see also the discussion preceding Prop. 3.3 and Exercise 3.7.

The threshold algorithm attempts to emulate approximately the minimum label selection policy of Dijkstra's algorithm with a much smaller computational effort. The candidate list V is organized into two distinct queues Q' and Q'' using a *threshold* parameter s. The queue Q' contains nodes with "small" labels; that is, it contains only nodes whose labels are no larger than s. At each iteration, a node is removed from Q', and any node j to be added to the candidate list is inserted in Q''. When the queue Q' is exhausted, the entire candidate list is repartitioned. The threshold is adjusted and the queues Q' and Q'' are recalculated, so that Q' consists of the nodes with labels that are no larger than the new threshold.

The performance of this method is quite sensitive to the method used to adjust the thresholds. For example, if s is taken to be equal to the current minimum label, the method is identical to Dijkstra's algorithm; if s is larger than all node labels, Q'' is empty and the algorithm reduces to the generic label correcting method. With an effective choice of threshold, the practical performance of the algorithm is very good. A number of heuristic approaches have been developed for selecting the threshold (see [GKP85a], [GKP85b], and [GaP88]). If all arc lengths are nonnegative, a bound $O(NA)$ on the operation count of the algorithm can be shown; see Exercise 3.7.

1.3.4 Single Origin/Single Destination Methods

Suppose that there is only one destination, call it t, and we want to find the shortest distance from the origin node 1 to t. We could use our earlier single origin/all destinations algorithms, but some improvements are possible.

Label Setting

Suppose first that we use the label setting method. Then we can stop the method when the destination t becomes permanently labeled; further computation will not improve the label d_t. If t is closer to the origin than many other nodes, the saving in computation time will be significant. Note that this approach can also be used when there are several destinations. The method is stopped when all destinations have been permanently labeled.

Another interesting possibility is to use a *two-sided label setting method*; that is, a method that simultaneously proceeds from the origin to the destination *and* from the destination to the origin. In this method, we successively label permanently the closest nodes to the origin (with their shortest distance *from* the origin) and the closest nodes to the destination (with their shortest distance *to* the destination). When some node gets permanently labeled from both sides, the labeling can stop; by combining the forward and backward paths of each labeled node and by comparing the resulting origin-to-destination paths, one can obtain a shortest path (see Exercise 3.8). For many problems, this approach can lead to a dramatic reduction in the total number of iterations. However, this two-sided labeling approach does not work when there are multiple destinations.

Label Correcting

Unfortunately, when label correcting methods are used, it may not be easy to realize the savings just discussed in connection with label setting. The difficulty is that even after we discover several paths to the destination t (each marked by an entrance of t into V), we cannot be sure that better paths will not be discovered later. In the presence of additional problem structure, however, the number of times various nodes will enter V can be reduced considerably.

Suppose that at the start of the algorithm we have, for each node i, an *underestimate* u_i of the shortest distance from i to t (we require $u_t = 0$). For example, if all arc lengths are nonnegative we may take $u_i = 0$ for all i. (We do not exclude the possibility that $u_i = -\infty$ for some i, which corresponds to the case where no underestimate is available for the shortest distance of i.) The following algorithm is a modified version of the generic shortest path algorithm.

Initially

$$V = \{1\},$$

$$d_1 = 0, \qquad d_i = \infty, \qquad \forall\ i \neq 1.$$

The algorithm proceeds in iterations and terminates when V is empty. The typical iteration (if V is assumed nonempty) is as follows.

Typical Iteration of the Generic Single Origin/Single Destination Algorithm

Remove a node i from V. For each outgoing arc $(i,j) \in \mathcal{A}$, with $j \neq 1$, if

$$d_i + a_{ij} < \min\{d_j, d_t - u_j\}$$

set

$$d_j := d_i + a_{ij}$$

and add j to V if it does not already belong to V.

The preceding iteration is the same as that of the generic algorithm, except that the test $d_i + a_{ij} < d_j$ for entering a node j into V is replaced by the more stringent test $d_i + a_{ij} < \min\{d_j, d_t - u_j\}$. (In fact, when the trivial underestimate $u_j = -\infty$ is used for all $j \neq t$ the two iterations coincide.) The idea is as follows: The label d_j corresponds at all times to the best path found thus far from 1 to j (cf. Prop. 3.2). Intuitively, the purpose of entering node j in V when its label is reduced is to generate shorter paths to the destination that pass through node j. If P_j is the path from 1 to j corresponding to $d_i + a_{ij}$, then $d_i + a_{ij} + u_j$ is an underestimate of the shortest path length among the set of paths $\mathcal{P}_j$ that first follow path P_j to node j and then follow some other path from j to t. If

$$d_i + a_{ij} + u_j \geq d_t,$$

then the current best path to t, which corresponds to d_t, is at least as short as any of the paths in $\mathcal{P}_j$, which have P_j as their first component. It is unnecessary to consider such paths, and for this reason node j need not be entered in V. In this way, the number of node entrances in V may be sharply reduced.

Figure 3.7 illustrates the algorithm. The following proposition proves its validity.

Proposition 3.4: Consider the generic single origin/single destination algorithm.

(a) At the end of each iteration, the following conditions hold:

  (i) $d_1 = 0$.

  (ii) If $d_j < \infty$ and $j \neq 1$, then d_j is the length of some path that starts at 1, never returns to 1, and ends at j.

(b) If the algorithm terminates, then upon termination, either $d_t < \infty$, in which case d_t is the shortest distance from 1 to t, or else there is no path from 1 to t.

(c) If the algorithm does not terminate, there exist paths of arbitrarily small length that start at 1 and never return to 1.

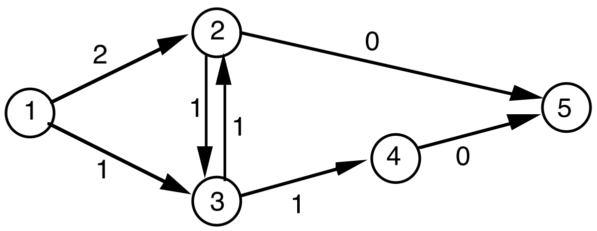
Proof: (a) The proof is identical to the corresponding parts of Prop. 3.2.

(b) If upon termination we have $d_t = \infty$, then the extra test $d_i + a_{ij} + u_j < d_t$ for entering V is always passed, so the algorithm generates the same label sequences as the generic (many destinations) shortest path algorithm. Therefore, Prop. 3.2(b) applies and shows that there is no path from 1 to t.

Let $\overline{d}_j$ be the final values of the labels d_j obtained upon termination and suppose that $\overline{d}_t < \infty$. Assume, to arrive at a contradiction, that there is a path $P_t = (1, j_1, j_2, \ldots, j_k, t)$ that has length L_t with $L_t < \overline{d}_t$. For $m = 1, \ldots, k$, let L_{j_m} be the length of the path $P_m = (1, j_1, j_2, \ldots, j_m)$.

Let us focus on the node j_k preceding t on the path P_t. We claim that $L_{j_k} < \overline{d}_{j_k}$. Indeed, if this were not so, then j_k must have been removed at some iteration from V with a label d_{j_k} satisfying $d_{j_k} \leq L_{j_k}$. If d_t is the label of t at the start of that iteration, we would then have

$$d_{j_k} + a_{j_k t} \leq L_{j_k} + a_{j_k t} = L_t < \overline{d}_t \leq d_t,$$



Iter. #	**Candidate List V**	**Node Labels**	**Node out of V**
1	$\{1\}$	$(0,\infty,\infty,\infty,\infty)$	1
2	$\{2,3\}$	$(0,2,1,\infty,\infty)$	2
3	$\{3,5\}$	$(0,2,1,\infty,2)$	3
4	$\{5\}$	$(0,2,1,\infty,2)$	5
	$\emptyset$	$(0,2,1,\infty,2)$	

Figure 3.7 Illustration of the generic single origin/single destination algorithm. Here the destination is $t = 5$ and the underestimates of shortest distances to t are $u_i = 0$ for all i. Note that at iteration 3, when node 3 is removed from V, the label of node 4 is not improved to $d_4 = 2$ and node 4 is not entered in V. The reason is that $d_3 + a_{34}$ (which is equal to 2) is not smaller than $d_5 - u_4$ (which is also equal to 2). Note also that upon termination the label of a node other than t may not be equal to its shortest distance (e.g. d_4).

implying that the label of t would be reduced at that iteration from d_t to $d_{j_k} + a_{j_k t}$, which is less than the final label $\overline{d}_t$ – a contradiction.

Next we focus on the node j_{k-1} preceding j_k and t on the path P_t. We use a similar (though not identical) argument to show that $L_{j_{k-1}} < \overline{d}_{j_{k-1}}$. Indeed, if this were not so, then j_{k-1} must have been removed at some iteration from V with a label $d_{j_{k-1}}$ satisfying $d_{j_{k-1}} \le L_{j_{k-1}}$. If d_{j_k} and d_t are the labels of j_k and t at the start of that iteration, we would then have

$$d_{j_{k-1}} + a_{j_{k-1}j_k} \le L_{j_{k-1}} + a_{j_{k-1}j_k} = L_{j_k} < \overline{d}_{j_k} \le d_{j_k}, \tag{3.17}$$

and since $L_{j_k} + u_{j_k} \le L_t < \overline{d}_t \le d_t$, we would also have

$$d_{j_{k-1}} + a_{j_{k-1}j_k} < d_t - u_{j_k}. \tag{3.18}$$

From Eqs. (3.17) and (3.18), it follows that the label of j_k would be reduced at that iteration from d_{j_k} to $d_{j_k} + a_{j_k t}$, which is less than the final label $\overline{d}_{j_k}$ – a contradiction.

Proceeding similarly, we obtain $L_{j_m} < \overline{d}_{j_m}$ for all $m = 1, \ldots, k$, and in particular $a_{1j_1} = L_{j_1} < \overline{d}_{j_1}$. Since

$$a_{1j_1} + u_{j_1} \leq L_t < \overline{d}_t,$$

and d_t is monotonically nonincreasing throughout the algorithm, we see that at the first iteration, j_1 will enter V with the label a_{1j_1}, which cannot be less than the final label $\overline{d}_{j_1}$. This is a contradiction; the proof of part (b) is complete.

(c) The proof is identical to the proof of Prop. 3.2(c). **Q.E.D.**

There are a number of possible implementations of the algorithm of this subsection, which parallel the ones given earlier for the many destinations problem. An interesting possibility to speed up the algorithm arises when an *overestimate* v_j of the shortest distance from j to t is known *a priori*. (We require $v_t = 0$; also $v_j = \infty$ implies that no overestimate is known for j.) The idea is that the method still works if the test $d_i + a_{ij} < d_t - u_j$ is replaced by the possibly sharper test $d_i + a_{ij} < D - u_j$, where D is any overestimate of the shortest distance from 1 to t with $D \leq d_t$ (check the proof of Prop. 3.4). We can obtain estimates D that may be strictly smaller than d_t by using the scalars v_j as follows: each time the label of a node j is reduced, we check whether $d_j + v_j < D$; if this is so, we replace D by $d_j + v_j$. In this way, we make the test for future admissibility into the candidate list V more stringent and save some unnecessary node entrances in V. This idea is used in some versions of the branch-and-bound method for integer programming; see Section 1.4 of [Ber87].

1.3.5 Multiple Origin/Multiple Destination Methods

Consider now the all-pairs shortest path problem where we want to find a shortest path from each node to each other node. The *Floyd-Warshall algorithm* is specifically designed for this problem, and it is not any faster when applied to the single destination problem. It starts with the initial condition

$$D_{ij}^0 = \begin{cases} a_{ij}, & \text{if } (i,j) \in \mathcal{A} \\ \infty, & \text{otherwise} \end{cases}$$

and generates sequentially for all $k = 0, 1, \ldots, N-1$, and all nodes i and j,

$$D_{ij}^{k+1} = \begin{cases} \min\left\{ D_{ij}^k,\ D_{i(k+1)}^k + D_{(k+1)j}^k \right\}, & \text{if } j \neq i \\ \infty, & \text{otherwise.} \end{cases}$$

An induction argument shows that D_{ij}^k gives the shortest distance from node i to node j using only nodes from 1 to k as intermediate nodes. Thus, D_{ij}^N gives the shortest distance from i to j (with no restriction on the intermediate nodes). There are N iterations, each requiring $O(N^2)$ operations, for a total of $O(N^3)$ operations.

Unfortunately, the Floyd-Warshall algorithm cannot take advantage of sparsity of the graph. It appears that for sparse problems it is typically better to apply a single origin/all destinations algorithm separately for each origin. If all the arc lengths are nonnegative, a label setting method can be used separately for each origin. If there are negative arc lengths (but no negative length cycles), one can of course apply a label correcting method separately for each origin, but there is another alternative that results in a superior worst-case complexity. It is possible to apply a label correcting method only *once* to a single origin/all destinations problem and obtain an equivalent all-pairs shortest path problem with nonnegative arc lengths; the latter problem can be solved using N separate applications of a label setting method. This alternative is based on the following proposition, which applies to the general minimum cost flow problem.

Proposition 3.5: Every minimum cost flow problem with arc costs a_{ij} such that all simple forward cycles have nonnegative cost is equivalent to another minimum cost flow problem involving the same graph and nonnegative arc costs $\hat{a}_{ij}$ of the form

$$\hat{a}_{ij} = a_{ij} + d_i - d_j, \qquad \forall\ (i,j) \in \mathcal{A},$$

where the scalars d_i can be found by solving a single origin/all destinations shortest path problem. The two problems are equivalent in the sense that they have the same constraints, and the cost function of one is the same as the cost function of the other plus a constant.

Proof: Let $(\mathcal{N}, \mathcal{A})$ be the graph of the given problem. Introduce a new node 0 and an arc $(0,i)$ for each $i \in \mathcal{N}$, thereby obtaining a new graph $(\mathcal{N}', \mathcal{A}')$. Consider the shortest path problem involving this graph, with arc lengths a_{ij} for the arcs $(i,j) \in \mathcal{A}$ and 0 for the arcs $(0,i)$. Since all incident arcs of node 0 are outgoing, all simple forward cycles of $(\mathcal{N}', \mathcal{A}')$ are also simple forward cycles of $(\mathcal{N}, \mathcal{A})$ and, by assumption, have nonnegative length. Since any forward cycle can be decomposed into a collection of simple forward cycles (cf. Exercise 1.5), all forward cycles (not necessarily simple) of $(\mathcal{N}', \mathcal{A}')$ have nonnegative length. Furthermore, there is at least one path from node 0 to every other node i, namely the path consisting of arc $(0,i)$. Therefore, the shortest distances d_i from node 0 to all other nodes i can be found by a label correcting method, and by Prop. 3.2, we have

$$\hat{a}_{ij} = a_{ij} + d_i - d_j \geq 0, \qquad \forall\ (i,j) \in \mathcal{A}.$$

Let us now view $\sum_{(i,j)\in\mathcal{A}} \hat{a}_{ij}x_{ij}$ as the cost function of a minimum cost flow problem involving the graph $(\mathcal{N},\mathcal{A})$ and the constraints of the original problem. We have

$$\begin{aligned}\sum_{(i,j)\in\mathcal{A}} \hat{a}_{ij}x_{ij} &= \sum_{(i,j)\in\mathcal{A}} \big(a_{ij}+d_i-d_j\big)x_{ij}\\ &= \sum_{(i,j)\in\mathcal{A}} a_{ij}x_{ij} + \sum_{i\in\mathcal{N}} d_i\left(\sum_{\{j\mid(i,j)\in\mathcal{A}\}} x_{ij} - \sum_{\{j\mid(j,i)\in\mathcal{A}\}} x_{ji}\right)\\ &= \sum_{(i,j)\in\mathcal{A}} a_{ij}x_{ij} + \sum_{i\in\mathcal{N}} d_i s_i,\end{aligned}$$

where s_i is the given supply of node i. Thus, the two cost functions $\sum_{(i,j)\in\mathcal{A}} \hat{a}_{ij}x_{ij}$ and $\sum_{(i,j)\in\mathcal{A}} a_{ij}x_{ij}$ differ by the constant $\sum_{i\in\mathcal{N}} d_i s_i$. **Q.E.D.**

It can be seen now that the all-pairs shortest path problem can be solved by using a label correcting method to solve the single origin/all destinations problem described in the above proof, thereby obtaining the scalars d_i and $\hat{a}_{ij}$, and by then applying a label setting method N times to solve the all-pairs shortest path problem involving the nonnegative arc lengths $\hat{a}_{ij}$. The shortest distance D_{ij} from i to j is obtained by adding $d_i - d_j$ to the shortest distance from i to j found by the label setting method.

Still another possibility for solving the all-pairs shortest path problem is to solve N separate single origin/all destinations problems but to also use the results of the computation for one origin to start the computation for the next origin. This can be done efficiently in the context of the simplex method presented in the next chapter; see also [GaP86], [GaP88].

E X E R C I S E S

Exercise 3.1

Consider the graph of Fig. 3.8. Find a shortest path from 1 to all nodes using the binary heap method, Dial's algorithm, and the D'Esopo-Pape algorithm.

Exercise 3.2

Consider the graph of Fig. 3.8. Find a shortest path from node 1 to node 6 using the generic single origin/single destination method of Section 1.3.4 with all distance underestimates equal to zero.

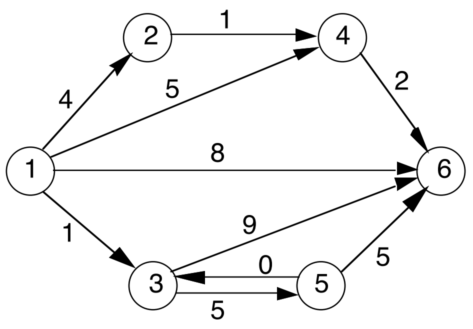


Figure 3.8 Graph for Exercises 3.1 and 3.2. The arc lengths are the numbers shown next to the arcs.

Exercise 3.3 (Shortest Path Tree Construction)

Consider the single origin/all destinations shortest path problem and assume that all cycles have nonnegative length. Consider the generic algorithm of Section 1.3.1, and assume that each time a label d_j is decreased to $d_i + a_{ij}$ the arc (i, j) is recorded in an array *PRED*(j). Consider the subgraph of the arcs *PRED*(j), $j \in \mathcal{N}$, $j \neq 1$. Show that after the first iteration this subgraph is a tree rooted at the origin, and that upon termination it is a shortest path tree.

Exercise 3.4 (The Bellman-Ford Algorithm)

Consider the single origin/all destinations shortest path problem. Assume that there is a path from the origin to all destinations, and that all cycles have nonnegative length. The Bellman-Ford algorithm starts with the initial conditions

$$d_1^0 = 0, \qquad d_j^0 = \infty, \qquad \forall\, j \neq 1$$

and generates d_j^k, $k = 1, 2, \ldots$, according to

$$d_1^k = 0, \qquad d_j^k = \min_{(i,j)\in\mathcal{A}} \{d_i^{k-1} + a_{ij}\}, \qquad \forall\, j \neq 1.$$

(a) Show that for all k, d_j^k is the shortest distance from 1 to j using paths with k arcs or less.

(b) Show that the algorithm terminates after at most N iterations, in the sense that for some $k \leq N$ we have $d_j^k = d_j^{k-1}$ for all j. Conclude that the running time of the algorithm is $O(NA)$.

(c) Consider a label correcting method that operates in cycles of iterations. In each cycle the nodes are scanned in a fixed order, and when a node i is found to belong to V an iteration removing i from V is performed (thus, there are as many as N iterations in a single cycle). Show that if $\overline{d}_j^k$ is the label of node j at the end of the kth cycle then $\overline{d}_j^k \le d_j^k$, where d_j^k are the iterates of the Bellman-Ford algorithm. Conclude that this label correcting method has an $O(NA)$ running time.

Exercise 3.5 (Min-Path/Max-Tension Theorem)

For a price vector $p = (p_1, \ldots, p_N)$, define the *tension* of arc (i, j) as $t_{ij} = p_i - p_j$ and the tension of a forward path P as $T_P = \sum_{(i,j)\in P^+} t_{ij}$. Show that the shortest distance between two nodes i_1 and i_2 is equal to the maximal tension T_P over all forward paths P starting at i_1 and ending at i_2, and all price vectors p satisfying the constraint $t_{ij} \le a_{ij}$ for all arcs (i, j). Interpret this as a duality result. *Note*: An intuitive explanation of this result in terms of a mechanical model is given in Section 4.3; see Fig. 3.1 of that section.

Exercise 3.6 (Path Bottleneck Problem)

Consider the framework of the shortest path problem. For any path P, define the *bottleneck arc* of P as an arc that has maximum length over all arcs of P. Consider the problem of finding a path connecting two given nodes and having minimum length of bottleneck arc. Derive an analog of Prop. 3.1 for this problem. Consider also a single origin/all destinations version of this problem. Develop an analog of the generic algorithm of Section 1.3.1, and prove an analog of Prop. 3.2. *Hint:* Replace $d_i + a_{ij}$ with $\max\{d_i, a_{ij}\}$.

Exercise 3.7 (Complexity of the Generic Algorithm)

Consider the generic algorithm, and assume that all arc lengths are nonnegative.

(a) Consider a node j satisfying at some time

$$d_j \le d_i, \qquad \forall\ i \in V.$$

Show that this relation will be satisfied at all subsequent times and that j will never again enter V. Furthermore, d_j will remain unchanged.

(b) Suppose that the algorithm is structured so that it removes from V a node of minimum label at least once every k iterations (k is some integer). Show that the algorithm will terminate in at most kN iterations.

(c) Show that the running time of the threshold algorithm is $O(NA)$. *Hint:* Define a cycle to be a sequence of iterations between successive repartitionings of the candidate list V. In each cycle, the node of V with minimum label at the start of the cycle will be removed from V during the cycle.

Exercise 3.8 (Two-Sided Label Setting)

Consider the shortest path problem from an origin node 1 to a destination node t, and assume that all arc lengths are nonnegative. This exercise considers an algorithm where label setting is applied simultaneously and independently from the origin and from the destination. In particular, the algorithm maintains a subset of nodes W, which are permanently labeled from the origin, and a subset of nodes V, which are permanently labeled from the destination. When W and V have a node i in common the algorithm terminates. The idea is that a shortest path from 1 to t cannot contain a node $j \notin W \cup V$; any such path must be longer than a shortest path from 1 to i followed by a shortest path from i to t (unless j and i are equally close to both 1 and to t).

Consider two subsets of nodes W and V with the following properties:

(1) $1 \in W$ and $t \in V$.

(2) W and V have nonempty intersection.

(3) If $i \in W$ and $j \notin W$, then the shortest distance from 1 to i is less than or equal to the shortest distance from 1 to j.

(4) If $i \in V$ and $j \notin V$, then the shortest distance from i to t is less than or equal to the shortest distance from j to t.

Let d_i^1 be the shortest distance from 1 to i using paths all the nodes of which, with the possible exception of i, lie in W ($d_i^1 = \infty$ if no such path exists), and let d_i^t be the shortest distance from i to t using paths all the nodes of which, with the possible exception of i, lie in V ($d_i^t = \infty$ if no such path exists).

(a) Show that such W, V, d_i^1, and d_i^t can be found by applying a label setting method simultaneously for the single origin problem with origin node 1 and for the single destination problem with destination node t.

(b) Show that the shortest distance D_{1t} from 1 to t is given by

$$D_{1t} = \min_{i \in W} \left\{ d_i^1 + d_i^t \right\} = \min_{i \in W \cup V} \left\{ d_i^1 + d_i^t \right\} = \min_{i \in V} \left\{ d_i^1 + d_i^t \right\}.$$

(c) Show that the nonempty intersection condition (2) can be replaced by the condition $\min_{i \in W} \left\{ d_i^1 + d_i^t \right\} \leq \max_{i \in W} d_i^1 + \max_{i \in V} d_i^t$.

Exercise 3.9 (k Shortest Node-Disjoint Paths)

Consider a graph with an origin 1, a destination t, and a length for each arc. We want to find k paths from 1 to t which share no node other 1 and t and which are such that the sum of the k path lengths is minimum. Formulate this problem as a minimum cost flow problem. *Hint:* Replace each node i other than 1 and t with two nodes i and i' and a connecting arc (i, i') with flow bounds $0 \leq x_{ii'} \leq 1$.

Exercise 3.10 (The Doubling Algorithm)

The *doubling algorithm* for solving the all-pairs shortest path problem is given by

$$D_{ij}^1 = \begin{cases} a_{ij}, & \text{if } (i,j) \in \mathcal{A} \\ 0, & \text{if } i = j \\ \infty, & \text{otherwise} \end{cases}$$

$$D_{ij}^{2k} = \begin{cases} \min_m \bigl\{D_{im}^k + D_{mj}^k\bigr\}, & \text{if } i \neq j,\ k = 1, 2, \ldots, \lfloor \log(N-1) \rfloor \\ 0, & \text{if } i = j,\ k = 1, 2, \ldots, \lfloor \log(N-1) \rfloor. \end{cases}$$

Show that for $i \neq j$, D_{ij}^k gives the shortest distance from i to j using paths with 2^{k-1} arcs or fewer. Show also that the running time is $O\bigl(N^3 \log m^*\bigr)$, where m^* is the maximum number of arcs in a shortest path.

Exercise 3.11 (Nonstandard Initial Conditions)

It is sometimes useful to start the generic algorithm with initial conditions other than the standard $V = \{1\}$, $d_1 = 0$, $d_j = \infty$ for $j \neq 1$. Such a possibility arises, for example, when shortest paths with respect to slightly different arc lengths are known from an earlier optimization. This exercise characterizes initial conditions under which the algorithm maintains its validity.

(a) Suppose that the initial V and d in the generic algorithm satisfy conditions (i), (ii), and (iii) of part (a) of Prop. 3.2. Show that the algorithm is still valid in the sense that parts (b) and (c) of Prop. 3.2 hold.

(b) Use the result of part (a) to derive "promising" initial conditions for application of the generic algorithm using paths from 1 to all other nodes, which are shortest with respect to slightly different arc lengths.

Exercise 3.12 (Uniqueness of Solution of Bellman's Equation)

Assume that all cycles have positive length. Show that if a vector $d = (d_1, d_2, \ldots, d_N)$ satisfies

$$d_j = \min_{(i,j) \in \mathcal{A}} \{d_i + a_{ij}\}, \qquad \forall\ j \neq 1,$$

$$d_1 = 0,$$

then for all j, d_j is the shortest distance from 1 to j. Show by example that this need not be true if there is a cycle of length 0. *Hint:* Consider the arcs (i, j) attaining the minimum in the above equation and consider the paths formed by these arcs.

1.4 NOTES AND SOURCES

Network problems are discussed in many books ([BeG62], [Dan63], [BuS65], [Iri69], [Hu69], [FrF70], [Chr75], [Mur76], [Law76], [Zou76], [BaJ78], [Min78], [KeH80], [JeB80], [PaS82], [Chv83], [GoM84], [Lue84], [Roc84], [BJS90]). Several of these books discuss linear programming first and develop linear network optimization as a special case. An alternative approach that relies heavily on duality, is given in [Roc84]. Bibliographies on the subject are provided in [GoM77], [VoR82], and [VoR85].

1.1. The conformal realization theorem has been developed in different forms in several sources [FoF62], [BuS65]. In our presentation we follow [Roc84].

1.2. The primal cost improvement approach for network optimization was initiated by Dantzig [Dan51], who specialized the simplex method to the transportation problem. The extensive subsequent work using this approach is surveyed at the end of Chapter 2.

The max flow-min cut theorem was discovered independently in [DaF56], [EFS56], and [FoF56b]. The proof that the Ford-Fulkerson algorithm with breadth-first search has polynomial complexity $O(NA^2)$ (Exercise 2.10) is due to [EdK72]. With proper implementation, this bound was improved to $O(N^2A)$ in [Din70], and to $O(N^3)$ in [Kar74]. A number of algorithms based on augmentation ideas were subsequently proposed ([Che77], [MKM78], [Gal80], [GaN80]). A different approach, which bears a close connection to the auction and ϵ-relaxation ideas discussed in Chapter 4, was proposed in [Gol85b]; see also [GoT86], [AhO86].

The dual cost improvement approach was initiated by Kuhn [Kuh55] who proposed the *Hungarian method* for the assignment problem. (The name of the algorithm honors its connection with the research of the Hungarian mathematician Egervary [Ege31].) Work using this approach is surveyed in Chapter 3.

The auction approach was initiated by the author in [Ber79] for the assignment problem, and in [Ber86a], [Ber86b] for the minimum cost flow problem. Work using this approach is surveyed at the end of Chapter 4.

The feasible distribution theorem (Exercise 2.5) is due to [Gal57] and [Hof60]. The maximal matching/minimal cover theorem is due to [Kon31] and [Ege31]. The theory of distinct representatives (Exercise 2.13) originated with [Hal56]; see also [HoK56] and [MeD58].

1.3. Work on the shortest path problem is very extensive. Literature surveys are given in [Dre69], [GPR82], and [DeP84]. The generic algorithm was first explicitly suggested as a unifying framework of many of the existing shortest path algorithms in [Pal84] and [GaP86].

The first label setting method was suggested in [Dij59], and also independently in [Dan60] and [WhH60]. The binary heap and related implementations were suggested in [Joh77]. Dial's algorithm was proposed in [Dia69] and received considerable attention after the appearance of [DGK79]; see also [DeF79]. For related algorithms using variable size buckets, see [Joh77], [DeF79], and [AMO88].

Label correcting methods were proposed in [Bel57] and [For56]. The D'Esopo-Pape algorithm appeared in [Pap74] based on an earlier suggestion of D'Esopo. The threshold algorithm is developed in [GKP85a], [GKP85b], and [GGK86a].

Two-sided label setting methods for the single origin/single destination problem (Exercise 3.8) were proposed in [Nic66]; see also [HKS89], which contains extensive computational results. A new type of two-sided label setting method is described in Section 4.3 (Exercise 3.5).

The Floyd-Warshall algorithm was given in [Flo62] and uses a theorem due to [War62]. Alternative algorithms for the all-pairs problem are given in [Dan67] and [Tab73].