Constrained Shortest Paths

Consider the network shown in Figure 16.1(a) which has two attributes associated with each arc (i, j): a cost c_{ij} and a traversal time t_{ij}. Suppose that we wish to find the shortest path from the source node 1 to the sink node 6, but we wish to restrict our choice of paths to those that require no more than $T = 10$ time units to traverse. This type of constrained shortest path application arises frequently in practice since in many contexts a company (e.g., a package delivery firm) wants to provide its services at the lowest possible cost and yet ensure a certain level of service to its customers (as embodied in the time restriction). In general, the constrained shortest path problem from node 1 to node n can be stated as the following integer programming problem:

$$\text{Minimize} \quad \sum_{(i,j)\in A} c_{ij}x_{ij} \tag{16.1a}$$

subject to

$$\sum_{\{j:(i,j)\in A\}} x_{ij} - \sum_{\{j:(j,i)\in A\}} x_{ji} = \begin{cases} 1 & \text{for } i = 1 \\ 0 & \text{for } i \in N - \{1, n\}, \\ -1 & \text{for } i = n \end{cases} \tag{16.1b}$$

$$\sum_{(i,j)\in A} t_{ij}x_{ij} \leq T, \tag{16.1c}$$

$$x_{ij} = 0 \text{ or } 1 \qquad \text{for all } (i, j) \in A. \tag{16.1d}$$

The problem is not a shortest path problem because of the timing restriction. Rather, it is a shortest path problem with an additional side constraint (16.1c). Instead of solving this problem directly, suppose that we adopt an indirect approach by combining time and cost into a single *modified cost*; that is, we place a dollar equivalent on time. So instead of setting a limit on the total time we can take on the chosen path, we set a "toll charge" on each arc proportional to the time that it takes to

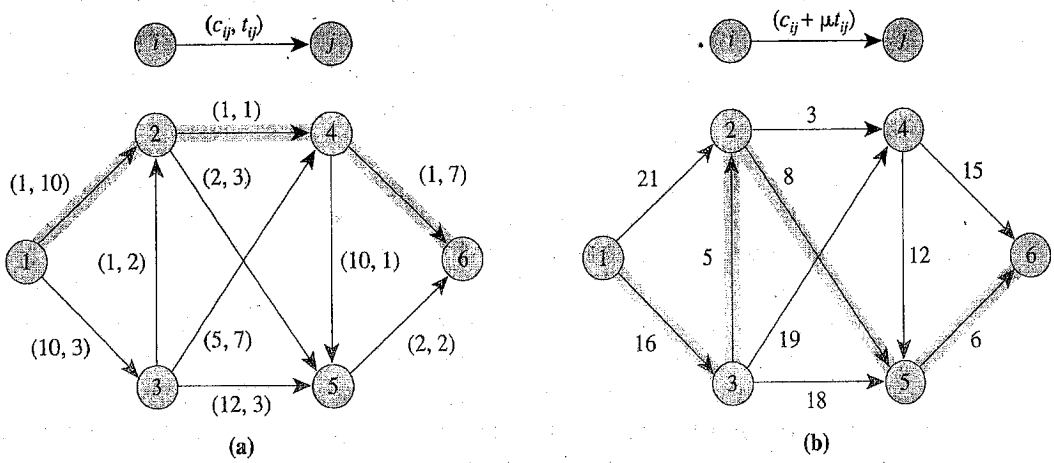


Figure 16.1 Time-constrained shortest path problem: (a) constrained shortest path problem (bold lines denote the shortest path for $\mu = 0$); (b) modified cost $c + \mu t$ with Lagrange multiplier $\mu = 2$ (bold lines denote the shortest path).

traverse that arc. For example, we might charge \$2 for each hour that it takes to traverse any arc. Note that if the toll charge is zero, we are ignoring time altogether and the problem becomes a usual shortest path problem with respect to the given costs. On the other hand, if the toll charge is very large, these charges become the dominant cost and we will be seeking the quickest path from the source to the sink. Can we find a toll charge somewhere in between these values so that by solving the shortest path problem with the combined costs (the toll charges and the original costs), we solve the constrained shortest path problem as a single shortest path problem?

For any choice μ of the toll charge, we solve a shortest path problem with respect to the modified costs $c_{ij} + \mu t_{ij}$. For the sample data shown in Figure 16.1(a), if $\mu = 0$, the modified problem becomes the shortest path problem with respect to the original costs c_{ij} and the shortest path 1–2–4–6 has length 3. This value is an obvious lower bound on the length of the constrained shortest path since it ignores the timing constraint. Now suppose that we set $\mu = 2$ and solve the modified problem. Figure 16.1(b) shows the modified costs $c_{ij} + 2t_{ij}$. The shortest path 1–3–2–5–6 has length 35. In this case, the path 1–3–2–5–6 that solves the modified problem happens to require 10 units to traverse, so it is a feasible constrained shortest path. Is it an optimal constrained shortest path?

To answer this question, let us make an important observation (which we will prove formally in the next section). Let P, with cost $c_P = \sum_{(i,j)\in P} c_{ij}$ and traversal time $t_P = \sum_{(i,j)\in P} t_{ij}$, be *any* feasible path to the constrained shortest path problem, and let $l(\mu)$ denote the optimal length of the shortest path with the modified costs when we impose a toll of μ units. Since the path P is feasible for the constrained shortest path problem, the time t_P required to traverse this path is at most $T = 10$ units. With respect to the modified costs $c_{ij} + \mu t_{ij}$, the cost $c_P + \mu t_P$ of the path P is the path's true cost c_P plus $\mu t_P \leq \mu T$ units. Therefore, if we subtract μT from the modified cost $c_P + \mu t_P$ of this path, we obtain a lower bound $c_P + \mu t_P - \mu T = c_P + \mu(t_P - T) \leq c_P$ on the cost c_P of this path. Since the shortest path with respect to the modified arc costs is less than or equal to the modified cost of any particular path, $l(\mu) \leq c_P + \mu t_P$ and so $l(\mu) - \mu T$ is a *common* lower bound on the length of *any* feasible path P and thus on the length of the constrained shortest path. Because this argument is completely general and applies to any value $\mu \geq 0$ of the toll charges, if we subtract μT from the optimal length of the shortest path of the modified problem, we obtain a lower bound on the optimal cost of the constrained shortest path problem.

Bounding Principle. *For any nonnegative value of the toll μ, the length $l(\mu)$ of the modified shortest path with costs $c_{ij} + \mu t_{ij}$ minus μT is a lower bound on the length of the constrained shortest path.*

Note that for our numerical example, for $\mu = 2$, the cost of the modified shortest path problem is 35 units and so $35 - 2(T) = 35 - 2(10) = 15$ is a lower bound on the length of the optimal constrained shortest path. But since the path 1–3–2–5–6 is a feasible solution to the constrained shortest path problem and its cost equals the lower bound of 15 units, we can be assured that it is an optimal constrained shortest path.

Observe that in this example we have been able to solve a difficult optimization

model (the constrained shortest path problem is an $\mathcal{NP}$-complete problem) by removing one or more problem constraints—in this case the single timing constraint—that makes the problem much more difficult to solve. Rather than solving the difficult optimization problem directly, we combined the complicating timing constraint with the original objective function, via the toll μ, so that we could then solve a resulting embedded shortest path problem. The motivation for adopting this approach was our observation that the original constrained shortest path problem had an attractive substructure, the shortest path problem, that we would like to exploit algorithmically. Whenever we can identify such attractive substructure, we could adopt a similar approach. For reasons that will become clearer in the next section, this general solution approach has become known as *Lagrangian relaxation*.

In our example we have been fortunate to find a constrained shortest path by solving the Lagrangian subproblem for a particular choice of the toll μ. We will not always be so lucky; nevertheless, as we will see, the lower bounding mechanism of Lagrangian relaxation frequently provides valuable information that we can exploit algorithmically.

Lagrangian relaxation is a general solution strategy for solving mathematical programs that permits us to decompose problems to exploit their special structure. As such, this solution approach is perfectly tailored for solving many models with embedded network structure. The Lagrangian solution strategy has a number of significant advantages:

1. Since it is often possible to decompose models in several ways and apply Lagrangian relaxation to each different decomposition, Lagrangian relaxation is a very flexible solution approach. Indeed, because of its flexibility, Lagrangian relaxation is more of a general problem solving strategy and solution framework than any single solution technique.
2. In decomposing problems, Lagrangian relaxation solves core subproblems as stand-alone models. Consequently, the solution approach permits us to exploit any known methodology or algorithm for solving the subproblems. In particular, when the subproblems are network models, the Lagrangian solution approach can take advantage of the various algorithms that we have developed previously in this book.
3. As we have already noted, Lagrangian relaxation permits us to develop bounds on the value of the optimal objective function and, frequently, to quickly generate good, though not necessarily optimal solutions with associated performance guarantees—that is, a bound on how far the solution could possibly be from optimality (in objective function value). In many instances in the context of integer programming, the bounds provided by Lagrangian relaxation methods are much better than those generated by solving the linear programming relaxation of the problems, and as a consequence, Lagrangian relaxation is often an attractive alternative to linear programming as a bounding mechanism in branch-and-bound methods for solving integer programs.
4. In many instances we can use Lagrangian relaxation methods to devise effective heuristic solution methods for solving complex combinatorial optimization problems and integer programs.

In the remainder of this chapter we describe the Lagrangian relaxation solution approach in more detail and demonstrate its use in solving several important network optimization models. Our purpose is not to present a comprehensive treatment of Lagrangian relaxation or of its applications to the field of network optimization, but rather to introduce this general solution strategy and to illustrate its applications in a way that would lay the essential foundations for applying the method in many other problem contexts. As a by-product of this discussion, in the text and in the exercises at the end of this chapter we introduce several noteworthy network optimization models that we do not treat elsewhere in the book.

Since one of the principal uses of Lagrangian relaxation is within implicit enumeration procedures for solving integer programs, before describing Lagrangian relaxation in more detail, we first discuss its use within classical branch-and-bound algorithms for solving integer programs. The reader can skip this section without loss of continuity.

16.2 PROBLEM RELAXATIONS AND BRANCH AND BOUND

In the last section we observed that Lagrangian relaxation permits us to develop a lower bound on the optimal length of a constrained shortest path. In Section 16.3 we develop a generalization of this result, showing that we can obtain a lower bound on the optimal objective function value of any minimization problem. These lower bounds can be of considerable value: for example, for our constrained shortest path example, we were able to use a lower bound to demonstrate that a particular solution that we generated by solving a shortest path subproblem, with modified costs, was optimal for the overall constrained problem. In general, we will not always be as fortunate in being able to use a lower bound to guarantee that the solution to a single subproblem solves the original problem. Nevertheless, as we show briefly in the section, we might still be able to use lower bounds as an algorithmic tool in reducing the number of computations required to solve combinatorial optimization problems formulated as integer programs.

Consider the following integer programming model:

Minimize cx

subject to

$x \in F$.

In this formulation, the set F represents the set of feasible solutions to an integer program, that is, the set of solutions $x = (x_1, x_2, \ldots, x_J)$ to the system

$$\mathcal{A}x = b,$$

$$x_j = 0 \text{ or } 1 \qquad \text{for } j = 1, 2, \ldots, J.$$

In a certain conceptual sense, this integer program is trivial to solve: We simply enumerate every combination of the decision variables, that is, all zero–one vectors $(x_1, x_2, \ldots, x_J)$ obtained by setting each variable x_j to value zero or 1; from among

all those vectors that satisfy the given equality constraint $\mathcal{A}x = b$, we choose the combination with the smallest value of the objective function cx. Of course, because of its combinatorial explosiveness, this total enumeration procedure is limited to very small problems; for a problem with 100 decision variables, even if we could compute one solution every nanosecond (10^{-9} second), enumerating all 2^{100} solutions would take us over a million million years—that is, a million different million years!

Can we avoid any of these computations? Suppose that $F = F^1 \cup F^2$. For example, we might obtain F^1 from F by adding the constraint $x_1 = 0$ and F^2 by adding the constraint $x_1 = 1$. Note that the optimal solution over the feasible set F is the best of the optimal solutions over F^1 and F^2. Suppose that we already have found an optimal solution $\bar{x}$ to $\min\{cx : x \in F^2\}$ and that its objective function value is $z(\bar{x}) = 100$. The number of potential integer solutions in F^1 is still 2^{J-1}, so it will be prohibitively expensive to enumerate all these possibilities, except when J is small.

Rather than attempt to solve the integer program over the feasible region F^1, suppose that we solve a *relaxed* version of the problem, possibly by relaxing the integrality constraints, and/or possibly by performing a Lagrangian relaxation of the problem. In general, we obtain a relaxation by removing some constraints from the model: for example, by replacing the restrictions $x_j \geq 0$ and integer, by the restriction $x_j \geq 0$, or by deleting one or more constraints of the form $\alpha x = \beta$. We could use many different types of relaxation—in Lagrangian relaxation, for example, we not only delete some problem constraints, but we also change the objective function of the problem. For the purpose of this discussion, we merely require that we relax some of the problem constraints and that the objective function value of the relaxation is a lower bound on the objective function value of the original problem.

Let x' denote an optimal solution to the relaxation, and let $z(x')$ denote the objective function value of this solution. We consider four possibilities:

1. The solution x' does not exist because the relaxed problem has no feasible solution.
2. The solution x' happens to lie in F^1 (even though we relaxed some of the constraints).
3. The solution x' does not lie in F^1 and its objective function value $z(x')$ satisfies the inequality $z(x') \geq z(\bar{x}) = 100$.
4. The solution x' does not lie in F^1 and its objective function value $z(x')$ of x' satisfies the inequality $z(x') < z(\bar{x}) = 100$.

Note that these four alternatives exhaust all possible outcomes and are mutually exclusive. Therefore, exactly one of them must occur.

We now make an important observation. In cases 1 to 3, we can terminate our computations: we have solved the original problem over the set F, even though we have not explicitly solved any integer program (assuming that we obtained the solution over the set F^2 without solving an integer program). In case 1, since the relaxation of the set F^1 is empty, the set F^1 is also empty, so the solution $\bar{x}$ solves the original (overall) integer program. In case 2, since we have found the optimal solution in the relaxation (and so a superset) of the set F^1, and this solution lies in

F^1, we have also found the best solution in F^1; therefore, either $\bar{x}$ or x' is the solution to the original problem (whichever solution has the smaller objective function value). Note that in this case we have implicitly considered (enumerated) all of the solutions in F^1 in the sense that we know that no solution in this set is better than $\bar{x}$. In case 3, the solution $\bar{x}$ has as good an objective function value as the best solution in a relaxation of F^1, so it has an objective function value that is as good as any solution in F^1. Therefore, $\bar{x}$ solves the original problem. Note that in case 3 we have used *bounding* information on the objective function value to eliminate the solutions in the set F^1 from further consideration.

In case 4, we have not yet solved the original problem. We can either try to solve the problem minimize $\{cx : x \in F^1\}$ by some direct method of integer programming or, we can partition F^1 into two sets F^3 and F^4. For example, we might obtain F^3 from F by constraining $x_1 = 0$ and $x_2 = 0$ and obtain F^4 by setting $x_1 = 0$ and $x_2 = 1$. We could then apply any relaxation or direct approach for the problems defined over the sets F^3 and F^4.

In a general branch-and-bound procedure, we would systematically partition the feasible region F into subregions $F^1, F^2, F^3, \ldots, F^K$. Let $\bar{x}$ denote the best feasible solution (in objective function value) we have obtained in prior computations. Suppose that for each $k = 1, 2, \ldots, K$, either F^k is empty or x^k is a solution of a relaxation of the set F^k and $c\bar{x} \leq cx^k$. Then no point in any of the regions F^1, $F^2, F^3, \ldots, F^k$ could have a better objective function value than $\bar{x}$, so $\bar{x}$ solves the original optimization problem. If $c\bar{x} > cx^k$, though, for any region F^k, we would need to subdivide this region by "branching" on some of the variables (i.e., dividing a subregion in two by setting $x_j = 0$ or $x_j = 1$ for some variable j to define two new subregions). Whenever we have satisfied the test $c\bar{x} \leq cx^k$ for all of the subregions (or we know they are empty), we have solved the original problem.

The intent of the branch-and-bound method is to find an optimal solution by solving only a small number of relaxations. To do so, we would need to obtain good solutions quickly and obtain good relaxations so that the objective function value $z(x^k)$ of the solution x^k to the relaxation of the set F^k is close in objective function value to the optimal solution over F^k itself.

In practice, in implementing the branch-and-bound procedure, we need to make many design decisions concerning the order for choosing the subregions, the variables to branch on for each subregion, and mechanisms (e.g., heuristic procedures) that we might use to find "good" feasible solutions. The literature contains many clever approaches for resolving these issues and for designing branch-and-bound procedures that are quite effective in practice. We also need to develop good relaxations that would permit us to obtain effective (tight) lower bounds: if the lower bounds are weak, cases 2 and 3 will rarely occur and the branch-and-bound procedure will degenerate into complete enumeration. On the other hand, if the bounds are very tight, the relaxations will permit us to eliminate much of the enumeration and develop very effective solution procedures. Since our purpose in this chapter is to introduce one relaxation procedure that has proven to be very effective in practice and discuss its applications, we will not consider the detailed design choices for implementing the branch-and-bound procedure.

We next summarize the basic underlying ideas of the Lagrangian relaxation technique.

16.3 LAGRANGIAN RELAXATION TECHNIQUE

To describe the general form of the Lagrangian relaxation procedure, suppose that we consider the following generic optimization model formulated in terms of a vector x of decision variables:

$$z^* = \min cx$$

subject to

$$\mathcal{A}x = b, \qquad \text{(P)}$$
$$x \in X.$$

This model (P) has a linear objective function cx and a set $\mathcal{A}x = b$ of explicit linear constraints. The decision variables x are also constrained to lie in a given constraint set X which, as we will see, often models embedded network flow structure. For example, the constraint set $X = \{x : \mathcal{N}x = q, 0 \leq x \leq u\}$ might be all the feasible solutions to a network flow problem with a supply/demand vector q. Or, the set X might contain the incidence vectors of all spanning trees or matchings of a given graph. Unless we state otherwise, we assume that the set X is finite (e.g., for network flow problems, we will let it be the finite set of spanning tree solutions).

As its name suggests, the Lagrangian relaxation procedure uses the idea of relaxing the explicit linear constraints by bringing them into the objective function with associated Lagrange multipliers μ (this old idea might be a familiar one from advanced calculus in the context of solving nonlinear optimization problems). We refer to the resulting problem

$$\text{Minimize} \quad cx + \mu(\mathcal{A}x - b)$$

subject to

$$x \in X,$$

as a *Lagrangian relaxation* or *Lagrangian subproblem* of the original problem, and refer to the function

$$L(\mu) = \min\{cx + \mu(\mathcal{A}x - b) : x \in X\},$$

as the *Lagrangian function*. Note that since in forming the Lagrangian relaxation, we have eliminated the constraints $\mathcal{A}x = b$ from the problem formulation, the solution of the Lagrangian subproblem need not be feasible for the original problem (P). Can we obtain any useful information about the original problem even when the solution to the Lagrangian subproblem is not feasible in the original problem (P)? The following elementary observation is a key result that helps to answer this question and that motivates the use of the Lagrangian relaxation technique in general.

Lemma 16.1 (Lagrangian Bounding Principle). *For any vector μ of the Lagrangian multipliers, the value $L(\mu)$ of the Lagrangian function is a lower bound on the optimal objective function value z^* of the original optimization problem* (P).

Proof. Since $\mathcal{A}x = b$ for every feasible solution to (P), for any vector μ of Lagrangian multipliers, $z^* = \min\{cx : \mathcal{A}x = b, x \in X\} = \min\{cx + \mu(\mathcal{A}x - b) : \mathcal{A}x = b, x \in X\}$. Since removing the constraints $\mathcal{A}x = b$ from the second formulation

cannot lead to an increase in the value of the objective function (the value might decrease), $z^* \geq \min\{cx + \mu(\mathcal{A}x - b) : x \in X\} = L(\mu)$. ◆

As we have seen, for any value of the Lagrangian multiplier μ, $L(\mu)$ is a lower bound on the optimal objective function value of the original problem. To obtain the sharpest possible lower bound, we would need to solve the following optimization problem

$$L^* = \max_{\mu} L(\mu)$$

which we refer to as the *Lagrangian multiplier problem* associated with the original optimization problem (P). The Lagrangian bounding principle has the following immediate implication.

Property 16.2 (Weak Duality). *The optimal objective function value L^* of the Lagrangian multiplier problem is always a lower bound on the optimal objective function value of the problem* (P) *(i.e., $L^* \leq z^*$).*

Our preceding discussion provides us with valid bounds for comparing objective function values of the Lagrange multiplier problem and optimization (P) for any choices of the Lagrange multipliers μ and any feasible solution x of (P):

$$L(\mu) \leq L^* \leq z^* \leq cx.$$

These inequalities furnish us with a guarantee when a Lagrange multiplier μ to the Lagrange multiplier problem or a feasible solution x to the original problem (P) are optimal.

Property 16.3 (Optimality Test)

(a) Suppose that μ is a vector of Lagrangian multipliers and x is a feasible solution to the optimization problem (P) *satisfying the condition $L(\mu) = cx$. Then $L(\mu)$ is an optimal solution of the Lagrangian multiplier problem [i.e., $L^* = L(\mu)$] and x is an optimal solution to the optimization problem* (P).

(b) If for some choice of the Lagrangian multiplier vector μ, the solution x^* of the Lagrangian relaxation is feasible in the optimization problem (P), *then x^* is an optimal solution to the optimization problem* (P) *and μ is an optimal solution to the Lagrangian multiplier problem.*

Note that by assumption in part (b) of this property, $L(\mu) = cx^* + \mu(\mathcal{A}x^* - b)$ and $\mathcal{A}x^* = b$. Therefore, $L(\mu) = cx^*$ and part (a) implies that x^* solves problem (P) and μ solves the Lagrangian multiplier problem.

As indicated by Property 16.3, the bounding principle immediately implies one advantage of the Lagrangian relaxation approach—the method can give us a *certificate* [in the form of the equality $L(\mu) = cx$ for some Lagrange multiplier μ] for guaranteeing that a given feasible solution x to the optimization problem (P) is an optimal solution. Even if $L(\mu) < cx$, having the lower bound permits us to state a bound on how far a given solution is from optimality: If $[cx - L(\mu)]/L(\mu) \leq 0.05$, for example, we know that the objective function value of the feasible solution x is no more than 5% from optimality. This type of bound is very useful in practice—it

permits us to assess the degree of suboptimality of given solutions and it permits us to terminate our search for an optimal solution when we have a solution that we know is close enough to optimality (in objective function value) for our purposes.

Lagrangian Relaxation and Inequality Constraints

In the optimization model (P), the constraints $\mathcal{A}x = b$ are all equality constraints. In practice, we often encounter models, such as the constrained shortest path problem, that are formulated more naturally in inequality form $\mathcal{A}x \leq b$. The Lagrangian multiplier problem for these problems is a slight variant of the one we have just introduced: The Lagrangian multiplier problem becomes

$$L^* = \max_{\mu \geq 0} L(\mu).$$

That is, the only change in the Lagrangian multiplier problem is that the Lagrangian multipliers now are restricted to be nonnegative. In Exercise 16.1, by introducing "slack variables" to formulate the inequality problem as an equivalent equality problem, we show how to obtain this optimal multiplier problem from the one we have considered for the equality problem. This development implies that the bounding property, the weak duality property, and the optimality test 16.3(a) are valid when we apply Lagrangian relaxation to any combination of equality and inequality constraints.

There is, however, one substantial difference between relaxing equality constraints and inequality constraints. When we relax inequality constraints $\mathcal{A}x \leq b$, if the solution x^* of the Lagrangian subproblem happens to satisfy these constraints, it need *not* be optimal (see Exercise 16.2). In addition to being feasible, this solution needs to satisfy the *complementary slackness condition* $\mu(\mathcal{A}x^* - b) = 0$, which is familiar to us from much of our previous discussion of network flows in section 9.4.

Property 16.4. *Suppose that we apply Lagrangian relaxation to the optimization problem* ($P^{\leq}$) *defined as minimize* $\{cx : \mathcal{A}x \leq b \text{ and } x \in X\}$ *by relaxing the inequalities* $\mathcal{A}x \leq b$. *Suppose, further, that for some choice of the Lagrangian multiplier vector* μ, *the solution* x^* *of the Lagrangian relaxation* (1) *is feasible in the optimization problem* ($P^{\leq}$), *and* (2) *satisfies the complementary slackness condition* $\mu(\mathcal{A}x^* - b) = 0$. *Then* x^* *is an optimal solution to the optimization problem* ($P^{\leq}$).

Proof. By assumption, $L(\mu) = cx^* + \mu(\mathcal{A}x^* - b)$. Since $\mu(\mathcal{A}x^* - b) = 0$, $L(\mu) = cx^*$. Moreover, since $\mathcal{A}x^* \leq b$, x^* is feasible, and so by Property 16.3(a) x^* solves problem ($P^{\leq}$). ◆

Are solutions to the Lagrangian subproblem of use in solving the original problem? Properties 16.3 and 16.4 show that certain solutions of the Lagrangian subproblem provably solve the original problem. We might distinguish two other cases: (1) when solutions obtained by relaxing inequality constraints are feasible but are not provably optimal for the original problem (since they do not satisfy the complementary slackness condition), and (2) when solutions to the Lagrangian relaxation are not feasible in the original problem.

In the first case, the solutions are candidate optimal solutions (possibly for use

in a branch-and-bound procedure). In the second case, for many applications, researchers have been able to devise methods to modify "modestly" infeasible solutions so that they become feasible with only a slightly degradation in the objective function value. These observations suggest that we might be able to use the solutions obtained from the Lagrangian subproblem as "approximate" solutions to the original problem, even when they are not provably optimal; in these instances, we can use Lagrangian relaxation as a heuristic method for generating provably good solutions in practice (the solutions might be provably good because of the Lagrangian lower bound information). The development of these heuristic methods depends heavily on the problem context we are studying, so we will not attempt to provide any further details.

Solving the Lagrangian Multiplier Problem

How might we solve the Lagrangian multiplier problem? To develop an understanding of possible solution techniques, let us consider the constrained shortest path problem that we defined in Section 16.1. Suppose that now we have a time limitation of $T = 14$ instead of $T = 10$. When we relax the time constraint, the Lagrangian multiplier function $L(\mu)$ for the constrained shortest path problem becomes

$$L(\mu) = \min\{c_P + \mu(t_P - T) : P \in \mathcal{P}\}.$$

In this formulation, $\mathcal{P}$ is the collection of all directed paths from the source node 1 to the sink node n. For convenience, we refer to the quantity $c_P + \mu(t_P - T)$ as the *composite cost* of the path P. For a specific value of the Lagrangian multiplier μ, we can solve $L(\mu)$ by enumerating all the directed paths in $\mathcal{P}$ and choosing the path with the smallest composite cost. Consequently, we can solve the Lagrangian multiplier problem by determining $L(\mu)$ for all nonnegative values of the Lagrangian multiplier μ and choosing the value that achieves $\max_{\mu \geq 0} L(\mu)$.

Let us illustrate this brute force approach geometrically. Figure 16.2 records the cost and time data for every path for our numerical example. Note that the composite cost $c_P + \mu(t_P - T)$ for any path P is a linear function of μ with an intercept of c_P and a slope of $(t_P - T)$. In Figure 16.3 we have plotted each of these path composite cost functions. Note that for any specific value of the Lagrange multiplier μ, we can find $L(\mu)$ by evaluating each composite cost function (line) and identifying the one with the least cost. This observation implies that the Lagrangian multiplier function $L(\mu)$ is the lower envelope of the composite cost lines and that the highest point on this envelope corresponds to the optimal solution of the Lagrangian multiplier problem.

In practice, we would never attempt to solve the problem in this way because the number of directed paths from the source node to the sink node typically grows exponentially in the number of nodes in the underlying network, so any such enumeration procedure would be prohibitively expensive. Nevertheless, this problem geometry helps us to understand the nature of the Lagrangian multiplier problem and suggests methods for solving the problem.

As we noted in the preceding paragraph, to find the optimal multiplier value μ^* of the Lagrangian multiplier problem, we need to find the highest point of the Lagrangian multiplier function $L(\mu)$. Suppose that we consider the polyhedron de-

Path P	Path cost c_P	Path time t_P	Composite cost $c_P + \mu (t_P - T)$
1–2–4–6	3	18	$3 + 4\mu$
1–2–5–6	5	15	$5 + \mu$
1–2–4–5–6	14	14	14
1–3–2–4–6	13	13	$13 - \mu$
1–3–2–5–6	15	10	$15 - 4\mu$
1–3–2–4–5–6	24	9	$24 - 5\mu$
1–3–4–6	16	17	$16 + 3\mu$
1–3–4–5–6	27	13	$27 - \mu$
1–3–5–6	24	8	$24 - 6\mu$

Figure 16.2 Path cost and time data for constrained shortest path example with $T = 14$.

fined by those points that lie on or below the function $L(\mu)$. These are the shaded points in Figure 16.3. Then geometrically, we are finding the highest point in a polyhedron defined by the function $L(\mu)$, which is a linear program.

Even though we have illustrated this property on a specific example, this situation is completely general. Consider the generic optimization model (P), defined

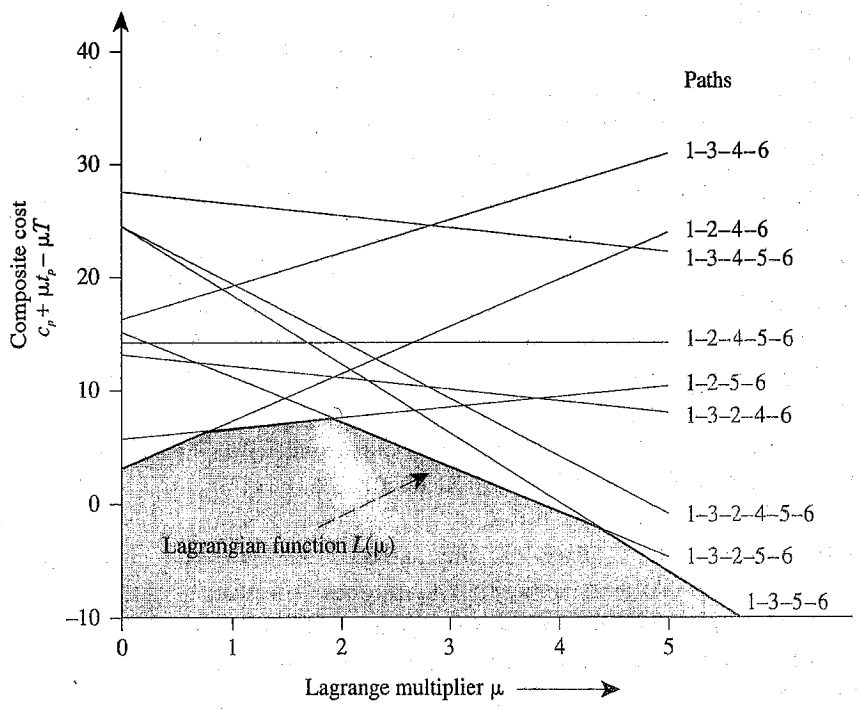


Figure 16.3 Lagrangian function for $T = 14$.

as $\min\{cx : \mathcal{A}x = b, x \in X\}$ and suppose that the set $X = \{x^1, x^2, \ldots, x^K\}$ is finite. By relaxing the constraints $\mathcal{A}x = b$, we obtain the Lagrangian multiplier function $L(\mu) = \min\{cx + \mu(\mathcal{A}x - b) : x \in X\}$. By definition,

$$L(\mu) \leq cx^k + \mu(\mathcal{A}x^k - b) \qquad \text{for all } k = 1, 2, \ldots, K.$$

In the space of composite costs and Lagrange multipliers μ (as in Figure 16.3), each function $cx^k + \mu(\mathcal{A}x^k - b)$ is a multidimensional "line" called a *hyperplane* (if μ is two-dimensional, it is a plane). The Lagrangian multiplier function $L(\mu)$ is the lower envelope of the hyperplanes $cx^k + \mu(\mathcal{A}x^k - b)$ for $k = 1, 2, \ldots, K$. In the Lagrangian multiplier problem, we wish to determine the highest point on this envelope: We can find this point by solving the optimization problem

$$\text{Maximize} \quad w$$

subject to

$$w \leq cx^k + \mu(\mathcal{A}x^k - b) \qquad \text{for all } k = 1, 2, \ldots, K,$$

$$\mu \text{ unrestricted},$$

which is clearly a linear program. We state this result as a theorem.

Theorem 16.5. *The Lagrangian multiplier problem* $L^* = \max_\mu L(\mu)$ *with* $L(\mu) = \min\{cx^k + \mu(\mathcal{A}x - b) : x \in X\}$ *is equivalent to the linear programming problem* $L^* = \max\{w : w \leq cx^k + \mu(\mathcal{A}x^k - b)$ *for* $k = 1, 2, \ldots, K\}$. ◆

Since, as shown by the preceding theorem, the Lagrangian multiplier problem is a linear program, we could solve this problem by applying the linear programming methodology. One resulting algorithm, which is known as *Dantzig–Wolfe decomposition* or *generalized linear programming*, is an important solution methodology that we discuss in some depth in Chapter 17 in the context of solving the multicommodity flow problem. One of the disadvantages of this approach is that it requires the solution of a series of linear programs that are rather expensive computationally. Another approach might be to apply some type of gradient method to the Lagrangian function $L(\mu)$. As shown by the constrained shortest path example, the added complication of this approach is that the Lagrangian function $L(\mu)$ is not differentiable. It is differentiable whenever the optimal solution of the Lagrangian subproblem is unique; but when the subproblem has two or more solutions, the Lagrangian function generally is not differentiable. For example, in Figure 16.4, at $\mu = 0$, the path 1–2–4–6 is the unique shortest path solution to the subproblem and the function $L(\mu)$ is differentiable. At this point, for the path $P = 1\text{–}2\text{–}4\text{–}6$, $L(\mu) = c_P + \mu(t_P - T)$; since $t_P = 18$ and $T = 14$, $L(\mu)$ has a slope $(t_P - T) = (18 - 14) = 4$. At the point $\mu = 2$, however, the paths 1–2–5–6 and 1–3–2–5–6 both solve the Lagrangian subproblem and the Lagrangian function is not differentiable. To accommodate these situations, we next describe a technique, known as the *subgradient optimization technique*, for solving the (nondifferentiable) Lagrangian multiplier problem.

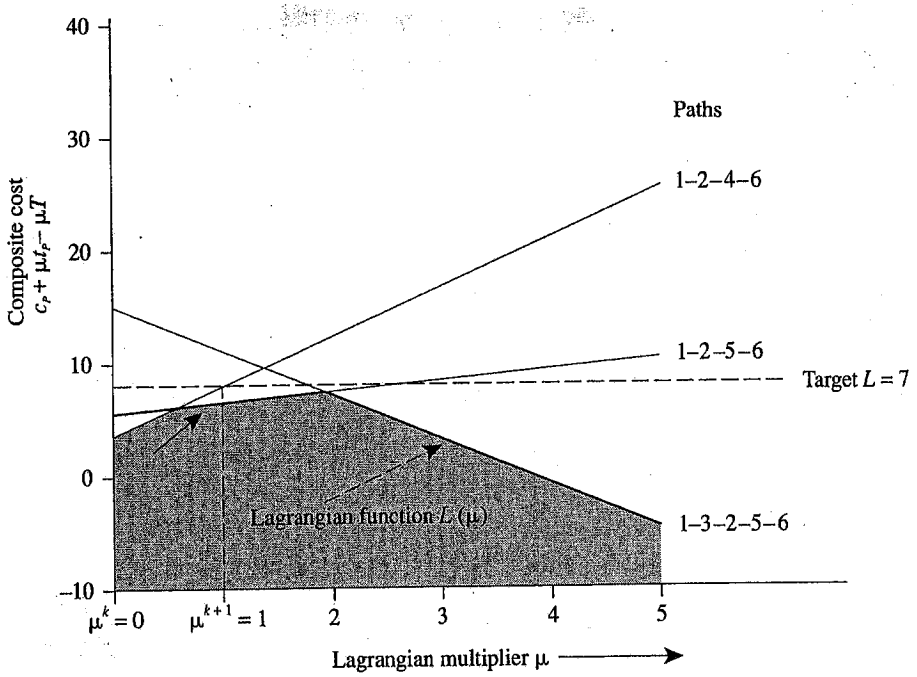


Figure 16.4 Steps of Newton's method for $T = 14$.

Subgradient Optimization Technique

In solving optimization problems with the nonlinear objective function $f(x)$ of an n-dimensional vector x, researchers and practitioners often use variations of the following classical idea: Form the gradient $\nabla f(x)$ of f defined as a row vector with components $(\partial f(x)/dx_1, \partial f(x)/dx_2, \ldots, \partial f(x)/dx_n)$. Recall from advanced calculus that the directional derivative of f in the direction d satisfies the equality

$$\lim_{\theta \to 0} \frac{f(x + \theta d) - f(x)}{\theta} = \nabla f(x) d.$$

So if we choose the direction d so that $\nabla f(x)d > 0$ and move in the direction d with a small enough "step length" θ—that is, change x to $x + \theta d$—we move uphill. This simple observation lies at the core of a considerable literature in nonlinear programming known as *gradient methods*.

Suppose that in solving the Lagrangian multiplier problem, we are at a point where the Lagrangian function $L(\mu) = \min\{cx + \mu(\mathcal{A}x - b) : x \in X\}$ has a unique solution $\bar{x}$, so is differentiable. Since $L(\mu) = c\bar{x} + \mu(\mathcal{A}\bar{x} - b)$ and the solution $\bar{x}$ remains optimal for small changes in the value of μ, the gradient at this point is $\mathcal{A}\bar{x} - b$, so a gradient method would change the value of μ as follows:

$$\mu \leftarrow \mu + \theta(\mathcal{A}\bar{x} - b).$$

In this expression, θ is a step size (a scalar) that specifies how far we move in the gradient direction. Note that this procedure has a nice intuitive interpretation. If $(\mathcal{A}\bar{x} - b)_i = 0$, the solution x uses up exactly the required units of the ith resource, and we hold the Lagrange multiplier (the toll) μ_i of that resource at its current value;

if $(\mathcal{A}\bar{x} - b)_i < 0$, the solution x uses up less than the available units of the ith resource and we decrease the Lagrange multiplier μ_i on that resource; and if $(\mathcal{A}\bar{x} - b)_i > 0$, the solution x uses up more than the available units of the ith resource and we increase the Lagrange multiplier μ_i on that resource.

To solve the Lagrangian multiplier problem, we adopt a rather natural extension of this solution approach. We let μ^0 be any initial choice of the Lagrange multiplier; we determine the subsequent values μ^k for $k = 1, 2, \ldots$, of the Lagrange multipliers as follows:

$$\mu^{k+1} = \mu^k + \theta_k(\mathcal{A}x^k - b).$$

In this expression, x^k is any solution to the Lagrangian subproblem when $\mu = \mu^k$ and θ_k is the step length at the kth iteration.

To ensure that this method solves the Lagrangian multiplier problem, we need to exercise some care in the choice of the step sizes. If we choose them too small, the algorithm would become stuck at the current point and not converge; if we choose the step sizes too large, the iterates μ^k might overshoot the optimal solution and perhaps even oscillate between two nonoptimal solutions (see Exercise 16.4 for an example). The following compromise ensures that the algorithm strikes an appropriate balance between these extremes and does converge:

$$\theta_k \to 0 \quad \text{and} \quad \sum_{j=1}^{k} \theta_j \to \infty.$$

For example, choosing $\theta_k = 1/k$ satisfies these conditions. These conditions ensure that the algorithm always converges to an optimal solution of the multiplier problem, but a proof of this convergence result is beyond the scope of our coverage in this book (the reference notes cite papers and books that examine the convergence of subgradient methods).

One important variant of the subgradient optimization procedure would be an adaptation of "Newton's method" for solving systems of nonlinear equations. Suppose, as before, that $L(\mu^k) = cx^k + \mu^k(\mathcal{A}x^k - b)$; that is, x^k solves the Lagrangian subproblem when $\mu = \mu^k$. Suppose that we assume that x^k continues to solve the Lagrangian subproblem as we vary μ; or, stated in another way, we make a linear approximation $r(\mu) = cx^k + \mu(\mathcal{A}x^k - b)$ to $L(\mu)$. Suppose further that we know the optimal value L^* of the Lagrangian multiplier problem (which we do not). Then we might move in the subgradient direction until the value of the linear approximation exactly equals L^*. Figure 16.4 shows an example of this procedure when applied to our constrained shortest path example, starting with $\mu^k = 0$. At this point, the path $P = 1$–2–4–6 solves the Lagrangian subproblem and $\mathcal{A}x^k - b$ equals $t_P - T = 18 - 4 = 4$. Since $L^* = 7$ and the path P has a cost $c_P = 3$, in accordance with this linear approximation, or Newton's method, we would approximate $L(\mu)$ by $r(\mu) = 3 + 4\mu$, set $3 + 4\mu = 7$, and define the new value of μ as $\mu^{k+1} = (7 - 3)/4 = 1$. In general, we set the step length θ_k so that

$$r(\mu^{k+1}) = cx^k + \mu^{k+1}(\mathcal{A}x^k - b) = L^*,$$

or since, $\mu^{k+1} = \mu^k + \theta_k(\mathcal{A}x^k - b)$,

$$r(\mu^{k+1}) = cx^k + [\mu^k + \theta_k(\mathcal{A}x^k - b)](\mathcal{A}x^k - b) = L^*.$$

Collecting terms, recalling that $L(\mu^k) = cx^k + \mu(\mathcal{A}x^k - b)$, and letting $\| y \| = (\sum_j y_j^2)^{1/2}$ denote the Euclidean norm of the vector y, we can solve for the step length and find that

$$\theta_k = \frac{L^* - L(\mu^k)}{\| \mathcal{A}x^k - b \|^2}.$$

Since we do not know the optimal objective function value L^* of the Lagrangian multiplier problem (after all, that's what we are trying to find), practitioners of Lagrangian relaxation often use the following popular heuristic for selecting the step length:

$$\theta_k = \frac{\lambda_k[\text{UB} - L(\mu^k)]}{\| \mathcal{A}x^k - b \|^2}.$$

In this expression, UB is an upper bound on the optimal objective function value z^* of the problem (P), and so an upper bound on L^* as well, and λ_k is a scalar chosen (strictly) between 0 and 2. Initially, the upper bound is the objective function value of any known feasible solution to the problem (P). As the algorithm proceeds, if it generates a better (i.e., lower cost) feasible solution, it uses the objective function value of this solution in place of the upper bound UB. Usually, practitioners choose the scalars λ_k by starting with $\lambda_k = 2$ and then reducing λ_k by a factor of 2 whenever the best Lagrangian objective function value found so far has failed to increase in a specified number of iterations. Since this version of the algorithm has no convenient stopping criteria, practitioners usually terminate it after it has performed a specified number of iterations.

The rationale for these choices of the step size and the convergence proof of the subgradient method would take us beyond the scope of our coverage. In passing, we might note that the subgradient optimization procedure is not the only way to solve the Lagrangian multiplier problem: practitioners have used a number of other heuristics, including methods known as *multiplier ascent methods* that are tailored for special problems. Since we merely wish to introduce some of the basic concepts of Lagrangian relaxation and to indicate some of the essential methods used to solve the Lagrangian multiplier problem, we will not discuss these alternative methods.

Subgradient Optimization and Inequality Constraints

As we noted earlier in this section, if we apply Lagrangian relaxation to a problem with constraints $\mathcal{A}x \leq b$ stated in inequality form instead of the equality constraints, the Lagrange multipliers μ are constrained to be nonnegative. The update formula $\mu^{k+1} = \mu^k + \theta_k(\mathcal{A}x^k - b)$ might cause one or more of the components μ_i of μ to become negative. To avoid this possibility, we modify the update formula as follows:

$$\mu^{k+1} = [\mu^k + \theta_k(\mathcal{A}x^k - b)]^+.$$

In this expression, the notation $[y]^+$ denotes the "positive part" of the vector y; that is, the ith component of $[y]^+$ equals the maximum of 0 and y_i. Stated in another way, if the update formula $\mu^{k+1} = \mu^k + \theta_k(\mathcal{A}x^k - b)$ would cause the ith component of μ_i to be negative, then we simply set the value of this component to be zero. We then implement all the other steps of the subgradient procedure (i.e., the choice of

the step size θ at each step and the solution of the Lagrangian subproblems) exactly the same as for problems with equality constraints. For problems with both equality and inequality constraints, we use a straightforward mixture of the equality and inequality versions of the algorithm: whenever the update formula for the Lagrange multipliers would cause any component μ_i of μ corresponding to an inequality constraint to become negative, we set the value of that multiplier to be zero.

Let us illustrate the subgradient method for inequality constraints on our constrained shortest path example. Suppose that we start to solve our constrained shortest path problem at $\mu^0 = 0$ with $\lambda^0 = 0.8$ and with UB = 24, the cost corresponding to the shortest path 1–3–5–6 joining nodes 1 and 6. Suppose that we choose to reduce the scalar λ_k by a factor of 2 whenever three successive iterations at a given value of λ_k have not improved on the best Lagrangian objective function value $L(\mu)$. As we have already noted, the solution x^0 to the Lagrangian subproblem with $\mu = 0$ corresponds to the path P = 1–2–4–6, the Lagrangian subproblem has an objective function value of $L(0) = 3$, and the subgradient $\mathcal{A}x^0 - b$ at $\mu = 0$ is $(t_P - 14) = 18 - 14 = 4$. So at the first step, we choose

$$\theta_0 = 0.8(24 - 3)/16 = 1.05,$$

$$\mu^1 = [0 + 1.05(4)]^+ = 4.2.$$

For this value of the Lagrange multiplier, from Figure 16.3, we see that the path P = 1–3–2–5–6 solves the Lagrangian subproblem; therefore, $L(4.2) = 15 + 4.2(10) - 4.2(14) = 15 - 16.8 = -1.8$, and $\mathcal{A}x^1 - b$ equals $(t_P - 14) = 10 - 14 = -4$. Since the path 1–3–2–5–6 is feasible, and its cost of 15 is less than UB, we change UB to value 15. Therefore,

$$\theta_1 = 0.8(15 + 1.8)/16 = 0.84,$$

$$\mu^2 = [4.2 + 0.84(-4)]^+ = 0.84.$$

From iterations 2 through 5, the shortest paths alternate between the paths 1–2–4–6 and 1–3–2–5–6. At the end of the fifth iteration, the algorithm has not improved upon (increased) the best Lagrangian objective function value of 6.36 for three iterations, so we reduce λ_k by a factor of 2. In the next 7 iterations the shortest paths are the paths 1–2–5–6, 1–3–5–6, 1–3–2–5–6, 1–3–2–5–6, 1–2–5–6, 1–3–5–6, and 1–3–2–5–6. Once again for three consecutive iterations, the algorithm has not improved the best Lagrangian objective function value, so we decrease λ_k by a factor of 2 to value 0.2. From this point on, the algorithm chooses either path 1–3–2–5–6 or path 1–2–5–6 as the shortest path at each step. Figure 16.5 shows the first 33 iterations of the subgradient algorithm. As we see, the Lagrangian objective function value is converging to the optimal value $L^* = 7$ and the Lagrange multiplier is converging to its optimal value of $\mu^* = 2$.

Note that for this example, the optimal multiplier objective function value of $L^* = 7$ is strictly less than the length of the shortest constrained path, which has value 13. In these instances, we say that the Lagrangian relaxation has a *duality (relaxation) gap*. To solve problems with a duality gap to completion (i.e., to find an optimal solution and a guarantee that it is optimal), we would apply some form of enumeration procedure, such as branch and bound, using the Lagrangian lower bound to help reduce the amount of concentration required.

k	μ^k	$t_p - T$	$L(\mu^k)$	λ_k	θ_k
0	0.0000	4	3.0000	0.80000	1.0500
1	4.2000	−4	−1.8000	0.80000	0.8400
2	0.8400	4	6.3600	0.80000	0.4320
3	2.5680	−4	4.7280	0.80000	0.5136
4	0.5136	4	5.0544	0.80000	0.4973
5	2.5027	−4	4.9891	0.40000	0.2503
6	1.5016	1	6.5016	0.40000	3.3993
7	4.9010	−6	−5.4059	0.40000	0.2267
8	3.5406	−4	0.8376	0.40000	0.3541
9	2.1244	−4	6.5026	0.40000	0.2124
10	1.2746	1	6.2746	0.40000	3.4902
11	4.7648	−6	−4.5886	0.40000	0.2177
12	3.4589	−4	1.1646	0.20000	0.1729
13	2.7671	−4	3.9316	0.20000	0.1384
14	2.2137	−4	6.1453	0.20000	0.1107
15	1.7709	1	6.7709	0.20000	1.6458
16	3.4167	−4	1.3330	0.20000	0.1708
17	2.7334	−4	4.0664	0.20000	0.1367
18	2.1867	−4	6.2531	0.10000	0.0547
19	1.9680	1	6.9680	0.10000	0.8032
20	2.7712	−4	3.9150	0.10000	0.0693
21	2.4941	−4	5.0235	0.10000	0.0624
22	2.2447	−4	6.0212	0.05000	0.0281
23	2.1325	−4	6.4701	0.05000	0.0267
24	2.0258	−4	6.8966	0.05000	0.0253
25	1.9246	1	6.9246	0.00250	0.0202
26	1.9447	1	6.9447	0.00250	0.0201
27	1.9649	1	6.9649	0.00250	0.0201
28	1.9850	1	6.9850	0.00250	0.0200
29	2.0050	−4	6.9800	0.00250	0.0013
30	2.0000	−4	7.0000	0.00250	0.0012
31	1.9950	1	6.9950	0.00250	0.0200
32	2.0150	−4	6.9400	0.00250	0.0013
33	2.0100	−4	6.9601	0.00125	0.0006

Figure 16.5 Subgradient optimization for a constrained shortest path problem.

16.4 *LAGRANGIAN RELAXATION AND LINEAR PROGRAMMING*

In this section we discuss several theoretical properties of the Lagrangian relaxation technique. As we have noted earlier in Section 16.2, the primary use of the Lagrangian relaxation technique is to obtain lower bounds on the objective function values of (discrete) optimization problems. By relaxing the integrality constraints in the integer programming formulation of a discrete optimization problem, thereby

creating a linear programming relaxation, we obtain an alternative method for generating a lower bound. Which of these lower bounds is sharper (i.e., larger in value)? In this section we answer this question by showing that the lower bound obtained by the Lagrangian relaxation technique is at least as sharp as that obtained by using a linear programming relaxation. As a result, and because the Lagrangian relaxation bound is often easier to obtain than the linear programming relaxation bound, Lagrangian relaxation has become a very useful lower bounding technique in practice.

The content in this section requires some background in linear algebra and linear programming. We refer the reader to Appendix C for a review of this material.

Our first result in this section concerns the application of Lagrangian relaxation to a linear programming problem.

Theorem 16.6. *Suppose that we apply the Lagrangian relaxation technique to a linear programming problem* (P′) *defined as* $\min\{cx : \mathcal{A}x = b, \mathcal{D}x \leq q, x \geq 0\}$ *by relaxing the constraints* $\mathcal{A}x = b$. *Then the optimal value* L^* *of the Lagrangian multiplier problem equals the optimal objective function value of* (P′).

Proof. We use linear programming optimality conditions to prove the theorem. Suppose that x^* is an optimal solution of the linear programming problem (P′) and that π^* and γ^* denote vectors of optimal dual variables associated with the constraints $\mathcal{A}x = b$ and $\mathcal{D}x \leq q$. By linear programming theory, x^*, π^*, and γ^* satisfy the following dual feasibility and complementary slackness conditions:

$$c + \pi^*\mathcal{A} + \gamma^*\mathcal{D} \geq 0, \quad [c + \pi^*\mathcal{A} + \gamma^*\mathcal{D}]x^* = 0, \quad \text{and} \quad \gamma^*[\mathcal{D}x - q] = 0.$$

Consider the Lagrangian subproblem $L(\mu)$ at $\mu = \pi^*$, which is $L(\pi^*) = \min\{cx + \pi^*(\mathcal{A}x - b) : \mathcal{D}x \leq q, x \geq 0\}$. Notice that x^* is feasible for this problem because it is feasible to (P′). Moreover, for the fixed value $\mu = \pi^*$, the previous dual feasibility and complementary slackness conditions are exactly those for the Lagrangian subproblem; therefore, x^* also solves the Lagrangian subproblem at $\mu = \pi^*$. But since $\pi^*(\mathcal{A}x^* - b) = 0$, $L(\pi^*) = cx^*$. Consequently, Property 16.3 implies that $L^* = L(\pi^*) = cx^*$, the optimal objective function value of (P′). ◆

The preceding theorem shows that the Lagrangian relaxation technique provides an alternative method for solving a linear programming problem. Instead of solving the linear programming problem directly using any linear programming algorithm, we can relax a subset of the constraints and solve the Lagrangian multiplier problem by using subgradient optimization and solving a sequence of relaxed problems. In some situations the relaxed problem is easy to solve, but the original problem is not; in these situations, a Lagrangian relaxation-based algorithm is an attractive solution approach.

Suppose next that we apply Lagrangian relaxation to a discrete optimization problem (P) defined as $\min\{cx : \mathcal{A}x = b, x \in X\}$. We assume that the discrete set X is specified as $X = \{x : \mathcal{D}x \leq q, x \geq 0 \text{ and integer}\}$ for an integer matrix $\mathcal{D}$ and an integer vector q. Consequently, the problem (P) becomes

$$z^* = \min\{cx : \mathcal{A}x = b, \mathcal{D}x \leq q, x \geq 0 \text{ and integer}\}. \qquad \text{(P)}$$

We incur essentially no loss of generality by specifying the set X in this manner because we can formulate almost all real-life discrete optimization problems as integer programming problems. Let (LP) denote the linear programming relaxation of the problem (P) and let z° denote its optimal objective function value. That is,

$$z^\circ = \min\{cx : \mathcal{A}x = b, \mathcal{D}x \leq q, x \geq 0\}. \qquad \text{(LP)}$$

Clearly, $z^\circ \leq z^*$ because the set of feasible solutions of (P) lies within the set of feasible solutions of (LP). Therefore, the linear programming relaxation provides a valid lower bound on the optimal objective function value of (P). We have earlier shown in Property 16.2 that the Lagrangian multiplier problem also gives a lower bound L^* on the optimal objective function value of (P). We now show that $z^\circ \leq L^*$; that is, Lagrangian relaxation yields a lower bound that is at least as good as that obtained from the linear programming relaxation. We establish this result by showing that the Lagrangian multiplier problem also solves a linear programming problem but that the solution space for this problem is contained within the solution space of the problem (LP). The linear programming problem that the Lagrangian multiplier problem solves uses "convexification" of the solution space $X = \{x : \mathcal{D}x \leq q, x \geq 0 \text{ and integer}\}$.

We assume that $X = \{x^1, x^2, \ldots, x^K\}$ is a finite set. We say that a solution x is a *convex combination* of the solutions $x^1, x^2, \ldots, x^K$ if $x = \sum_{k=1}^K \lambda_k x^k$ for some nonnegative weights $\lambda_1, \lambda_2, \ldots, \lambda_K$ satisfying the condition $\sum_{k=1}^K \lambda_k = 1$. Let $\mathcal{H}(X)$ denote the *convex hull* of X (i.e., the set of all convex combinations of X). In the subsequent discussion we use the following properties of $\mathcal{H}(X)$.

Property 16.7

(a) The set $\mathcal{H}(X)$ is a polyhedron, that is, it can be expressed as a solution space defined by a finite number of linear inequalities.

(b) Each extreme point solution of the polyhedron $\mathcal{H}(X)$ lies in X, and if we optimize a linear objective function over $\mathcal{H}(X)$, some solution in X will be an optimal solution.

(c) The set $\mathcal{H}(X)$ is contained in the set of solutions $\{x : \mathcal{D}x \leq q, x \geq 0\}$.

Proof. Part (a) is a well-known result in linear algebra which we do not prove. The first statement in part (b) follows from the fact that every point of $\mathcal{H}(X)$ not in X is a convex combination, with positive weights, of two or more points in X and so is not an extreme point (see Appendix C). The second statement in part (b) is a consequence of the fact that linear programs always have at least one extreme point solution (see Appendix C). Part (c) follows from the fact that every solution in X also belongs to the convex set $\{x : \mathcal{D}x \leq q, x \geq 0\}$, and consequently, every convex combination of solutions in X, which defines $\mathcal{H}(X)$, also belongs to the set $\{x : \mathcal{D}x \leq q, x \geq 0\}$. ◆

We now prove the main result of this section.

Theorem 16.8. *The optimal objective function value L^* of the Lagrangian multiplier problem equals the optimal objective function value of the linear program $\min\{cx : \mathcal{A}x = b, x \in \mathcal{H}(X)\}$.*

Proof. Consider the Lagrangian subproblem

$$L(\mu) = \min\{cx + \mu(\mathcal{A}x - b) : x \in X\},$$

for some choice μ of the Lagrange multipliers. This problem is equivalent to the problem

$$L(\mu) = \min\{cx + \mu(\mathcal{A}x - b) : x \in \mathcal{H}(X)\}, \tag{16.2}$$

because by Property 16.7(b), some extreme point solution of $\mathcal{H}(X)$ solves this problem and each extreme point solution of $\mathcal{H}(X)$ belongs to X. Now, notice that the Lagrangian subproblem defined by (16.2) is a linear programming problem because by Property 16.7(a), we can formulate the set $\mathcal{H}(X)$ as the set of solutions of a finite number of linear inequalities. Therefore, we can conceive of the Lagrangian subproblem (16.2) as a relaxation of the following linear programming problem:

$$\min\{cx : \mathcal{A}x = b, x \in \mathcal{H}(X)\}.$$

Finally, we use Theorem 16.6 to observe that the optimal value L^* of the Lagrangian multiplier problem equals the optimal objective function value of the linear program $\min\{cx : \mathcal{A}x = b, x \in \mathcal{H}(X)\}$. ◆

We subsequently refer to the problem $\min\{cx : \mathcal{A}x = b, x \in \mathcal{H}(X)\}$ as the *convexified version* of problem (P) and refer to it as (CP). The preceding theorem shows that L^* equals the optimal objective function value of the convexified problem. What is the relationship between the set of feasible solutions of the convexified problem (CP) and the linear programming relaxation (LP)? We illustrate this relationship using a numerical example.

For simplicity, in our example we assume that the relaxed constraints are of the form $\mathcal{A}x \leq b$ instead of $\mathcal{A}x = b$. We consider a two-variable problem with the constraints $\mathcal{A}x \leq b$ and $\mathcal{D}x \leq q$ as shown in Figure 16.6(a). This figure also specifies the set of solutions of the integer programming problem (P), denoted by the circled points. Figure 16.6(b) shows the solution space of the linear programming relaxation (LP) of the problem. Figure 16.6(c) shows the convex hull $\mathcal{H}(X)$ and Figure 16.6(d) depicts the solution space of the convexified problem (CP). Note that the solution space of (CP) is a subset of the solution space of (LP).

The preceding result is also easy to establish in general. Notice from Property 16.7(c) that since $\mathcal{H}(X)$ is contained in the set $\{x : \mathcal{D}x \leq q, x \geq 0\}$, the set of solutions of problem (CP) given by $\{x : \mathcal{A}x = b, x \in \mathcal{H}(X)\}$ is contained in the set of solutions of (LP) given by $\{x : \mathcal{A}x = b, \mathcal{D}x \leq q, x \geq 0\}$. Since optimizing the same objective function over a smaller solution space cannot improve the objective function value, we see that $z^\circ \leq L^*$. We state this important result as a theorem.

Theorem 16.9. *When applied to an integer program stated in minimization form, the lower bound obtained by the Lagrangian relaxation technique is always as large (or, sharp) as the bound obtained by the linear programming relaxation of the problem; that is,* $z^\circ \leq L^*$.

Under what situations will the Lagrangian bound equal the linear programming

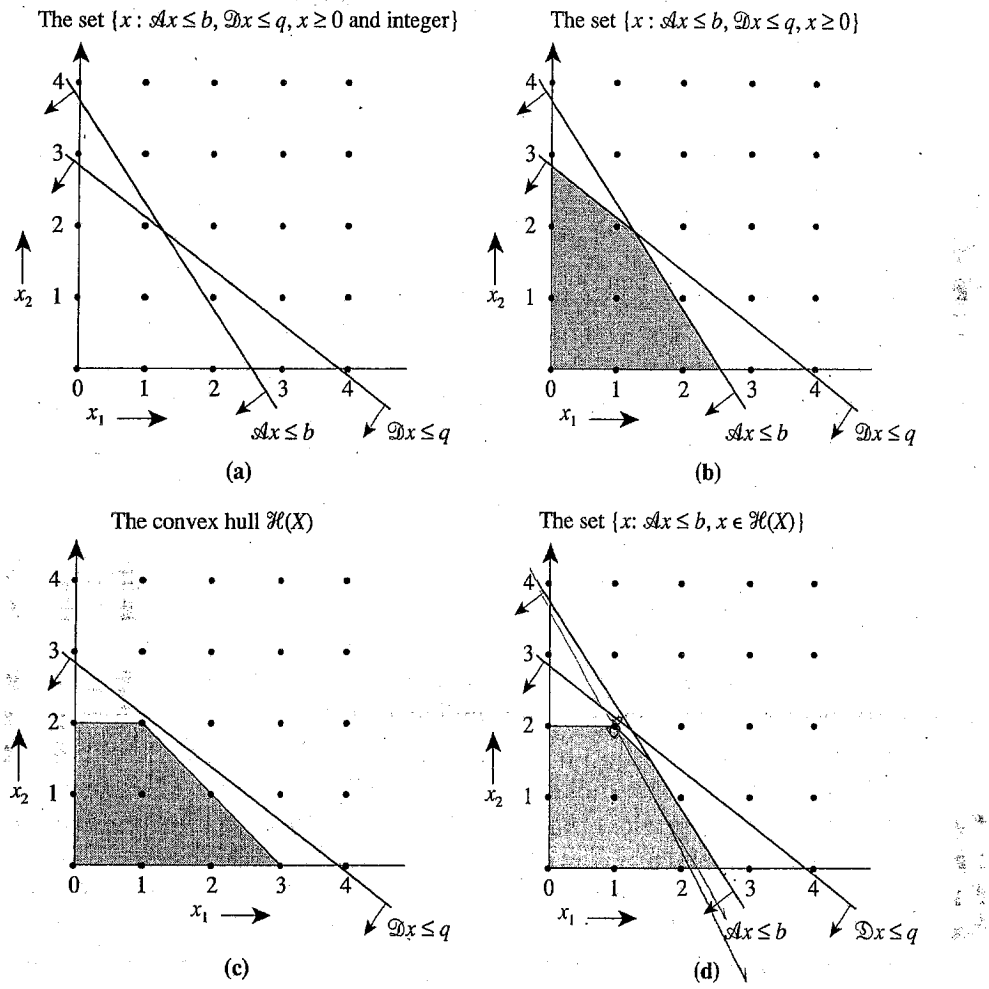


Figure 16.6 Illustrating the relationship between the problem (LP) and (CP): (a) solution space of the integer program (P); (b) solution space of the linear programming relaxation (LP); (c) convex hull $\mathcal{H}(x)$; (d) solution space of the convexified problem (CP).

bound? We show that if the Lagrangian subproblem satisfies a property, known as the *integrality property*, the Lagrangian bound will equal the linear programming bound. We say that the Lagrangian subproblem $\min\{dx : \mathcal{D}x \le q, x \ge 0 \text{ and integer}\}$ satisfies the integrality property if it has an integer optimal solution for every choice of objective function coefficients *even if we relax the integrality restrictions on the variables* x. Note that this condition implies that the problems $\min\{cx + \mu(\mathcal{A}x - b) : \mathcal{D}x \le q, x \ge 0 \text{ and integer}\}$ and $\min\{cx + \mu(\mathcal{A}x - b) : \mathcal{D}x \le q, x \ge 0\}$ have the same optimal objective function values for every choice of the Lagrange multiplier μ. For example, if the constraints $\mathcal{D}x \le q$ are the mass balance constraints of a minimum cost flow problem (or any of its special cases, such as the maximum flow, shortest path, and assignment problems), the problem $\min\{cx + \mu(\mathcal{A}x - b) : \mathcal{D}x \le q, x \ge 0\}$ will always have an integer optimal solution and imposing integrality constraints on the variables will not increase the optimal objective function value.

Theorem 16.10. *If the Lagrangian subproblem of the optimization problem* (P) *satisfies the integrality property, then* $z^\circ = L^*$.

Proof. Observe that the problem $\min\{dx : \mathcal{D}x \leq q, x \geq 0\}$ will have an integer optimal solution for every choice of d only if every extreme point solution of the constraints $\mathcal{D}x \leq q$, $x \geq 0$, is integer; for otherwise, we can select d so that a noninteger extreme point solution becomes an optimal solution. This observation implies that the set $\{x : \mathcal{D}x \leq q, x \geq 0\}$ equals the convex hull of $X = \{x : \mathcal{D}x \leq q, x \geq 0 \text{ and integer}\}$, which we have denoted by $\mathcal{H}(X)$. This result further implies that the sets $\{x : \mathcal{A}x = b, \mathcal{D}x \leq q, x \geq 0\}$ and $\{x : \mathcal{A}x = b, x \in \mathcal{H}(X)\}$ are the same. The first of these sets is the set of feasible solutions of the linear programming relaxation (LP) and the latter set is the set of feasible solutions of the convexified problem (CP). Since both the problems (LP) and (CP) have the same set of feasible solutions, they will have the same optimal objective function value, which is the desired conclusion of the theorem. ◆

This result shows that for problems satisfying the integrality property, solving the Lagrangian multiplier problem is equivalent to solving the linear programming relaxation of the problem. In these situations the Lagrangian relaxation technique provides no better a bound than the linear programming relaxation. Nevertheless, the Lagrangian relaxation technique might still be of considerable value, because solving the Lagrangian multiplier problem might be more efficient than solving the linear programming relaxation directly. Network optimization problems perhaps provide the most useful problem domain for exploiting this result because the Lagrangian subproblem in these cases often happens to be a minimum cost flow problem or one of its specializations.

As we have noted previously, in many (in fact, most) problem instances, the optimal objective function value L^* of the Lagrangian multiplier problem will be strictly less than the optimal objective function value z^* of problem (P); that is, the problem has a *duality gap*. As an example, consider the constrained shortest path example that we discussed in Section 16.3. For this example, $L^* = 7$ and $z^* = 13$. The duality gap occurs because the Lagrangian multiplier problem solves an optimization problem over a larger solution space (its convexification) than that of the original problem (P), and consequently, its optimal objective function value might be smaller.

16.5 APPLICATIONS OF LAGRANGIAN RELAXATION

As we noted earlier in the chapter, Lagrangian relaxation has many applications in network optimization. In this section we illustrate the breadth of these applications. The selected applications are both important in practice and illustrate how many of the network models we have considered in earlier chapters arise as Lagrangian subproblems. We consider the following models with embedded network structure.

Topic	Embedded network structure
Networks with side constraints	• Minimum cost flows • Shortest paths
Traveling salesman problem	• Assignment problem • Minimum cost flows
Vehicle routing	• Assignment problem • A variant of minimum spanning tree
Network design	• Shortest paths
Two-duty operator scheduling	• Shortest paths • Minimum cost flows
Degree-constrained minimum spanning trees	• Minimum spanning tree
Multi-item production planning	• Shortest paths • Minimum cost flows • Dynamic programs

Application 16.1 Networks with Side Constraints

The constrained shortest path problem is a special case of a broader set of optimization models known as network flow problems with side constraints. We can formulate a generic version of this problem as follows:

$$\text{Minimize} \quad cx$$

subject to

$$\mathcal{A}x \leq b,$$

$$\mathcal{N}x = q,$$

$$l \leq x \leq u, \text{ and } x_{ij} \text{ integer} \qquad \text{for all } (i, j) \in I.$$

In this formulation, as in the usual minimum cost flow problem, x is a vector of arc flows, $\mathcal{N}$ is a node–arc incidence matrix, q is a vector of node supplies and demands, and l and u are lower and upper bounds imposed on the arc flows. The set I is an index set of variables that must be integer. The flow vector x might be constrained to be integer or not, depending on the application being modeled. The added complication in this model are the side constraints $\mathcal{A}x \leq b$ that further restrict the arc flows.

For example, in the constrained shortest path problem, the network constraints model a shortest path problem [i.e., $q(s) = 1$ and $q(t) = -1$ for the source node s and destination node t, and $q(j) = 0$ for every other node j; also, every lower bound $l_{ij} = 0$ and every upper bound $u_{ij} = \infty$]. In this case the side constraint $\sum_{(i,j)\in A} t_{ij}x_{ij} \leq T$ is a single inequality constraint modeling the timing restriction.

The network flow model with side constraints arises in many application contexts in which the arc flows consume scarce resources (e.g., labor) or we wish to impose service constraints on the flows (e.g., maximum delay times in a communication and transportation network). The model also arises when the network flow model has multiple commodities, each governed by their own flow constraints, that

share common resources such as arc capacities. In Chapter 17 we consider one such model, the classical multicommodity flow problem, in some detail.

We might note that the network flow model with side constraints also arises in other, perhaps more surprising ways. As an illustration, consider a standard work force scheduling problem. Suppose that we wish to schedule employees (e.g., telephone operators, production workers, or nurses) in a way that ensures that $\alpha(j)$ employees are available for work on the jth day of the week; suppose, further, that we wish to schedule the employees so that each has two consecutive days off each week. That is, each of them works 5 consecutive days and then has 2 days off. We incur a cost c_j for each employee that is scheduled to work on day j. Figure 16.7 shows a network flow model with side constraints for this problem. The network contains three types of arcs.

1. A "work arc" for each day of the week: The flow on this arc is the number of employees scheduled to work on that day; the arc has an associated cost (e.g., weekends might have a pay premium and so a higher cost) and a lower flow bound equaling the number of employees required to work on that day.
2. A "total work force arc" that introduces the work force at the beginning of the planning cycle (which we arbitrarily take to be Sunday) and removes it at the end of the planning cycle (Saturday): the flow y on this arc is the total number of employees employed during the week.
3. "Days-off arcs" with the flows $x_{\text{sun}}, x_{\text{mon}}, \ldots, x_{\text{sat}}$, each representing a schedule with 2 days off beginning with the day indicated by the subscript: The flow on arc x_{sun}, for example, bypasses the Sunday and Monday work arcs, indicating that the employees working in this schedule are not available for work on Sunday and Monday.

A complicating feature of this network flow model is a single additional constraint indicating that every employee must be assigned to at least one schedule; that is,

$$y = x_{\text{sun}} + x_{\text{mon}} + \cdots + x_{\text{sat}}.$$

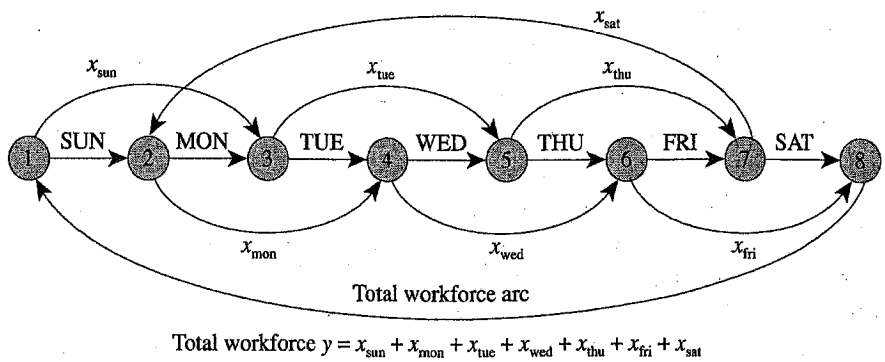


Figure 16.7 Network model of the cyclic scheduling problem. Lower bound on day arcs = demand for the day.

This side constraint specifies a flow relationship between several of the arcs in the network flow model. Relaxing this constraint and using Lagrangian relaxation provides us with one algorithmic approach for solving this problem. The algorithmic procedure for applying Lagrangian relaxation to the general network flow model with side constraints is essentially the same as the procedure we have discussed for the constrained shortest path problem: we associate nonnegative Lagrange multipliers μ with the side constraints $\mathcal{A}x \leq b$ and bring them into the objective function to produce the network flow subproblem

$$\text{minimize}\{cx + \mu(\mathcal{A}x - b) : \mathcal{N}x = q, l \leq x \leq u\},$$

and then solve a sequence of these problems with different values of the Lagrange multipliers μ which we update using the subgradient optimization technique. For each choice of the Lagrangian multiplier on this constraint, the Lagrangian subproblem is a network flow problem. In Exercise 9.9 we show that we can actually solve this special case of network flows with side constraints much more efficiently by solving a polynomial sequence of network flow problems.

Application 16.2 Traveling Salesman Problem

The traveling salesman problem is perhaps the most famous problem in all of network and combinatorial optimization: Its simplicity and yet its difficulty have made it an alluring problem that has attracted the attention of many noted researchers over a period of several decades. The problem is deceptively easy to state: Starting from his home base, node 1, a salesman wishes to visit each of several cities, represented by nodes 2, . . . , n, exactly once and return home, doing so at the lowest possible travel cost. We will refer to any feasible solution to this problem as a *tour* (of the cities).

The traveling salesman problem is a generic core model that captures the combinatorial essence of most routing problems and, indeed, most other routing problems are extensions of it. For example, in the classical vehicle routing problem, a set of vehicles, each with a fixed capacity, must visit a set of customers (e.g., grocery stores) to deliver (or pick up) a set of goods. We wish to determine the best possible set of delivery routes. Once we have assigned a set of customers to a vehicle, that vehicle should take the minimum cost tour through the set of customers assigned to it; that is, it should visit these customers along an optimal traveling salesman tour.

The traveling salesman problem also arises in problems that on the surface have no connection with routing. For example, suppose that we wish to find a sequence for loading jobs on a machine (e.g., items to be painted), and that whenever the machine processes job i after job j, we must reset the machine (e.g., clear the dies of the colors of the previous job), incurring a setup time c_{ij}. Then in order to find the processing sequence that minimizes the total setup time, we need to solve a traveling salesman problem—the machine, which functions as the "salesman," needs to "visit" the jobs in the most cost-effective manner.

There are many ways to formulate the traveling salesman problem as an optimization model. We present a model with an embedded (directed) network flow structure. Exercises 16.21 and 16.23 consider other modeling approaches. Let c_{ij}

denote the cost of traveling from city i to city j and let y_{ij} be a zero–one variable, indicating whether or not the salesman travels from city i to city j. Moreover, let us define flow variables x_{ij} on each arc (i, j) and assume that the salesman has $n - 1$ units available at node 1, which we arbitrarily select as a "source node," and that he must deliver 1 unit to each of the other nodes. Then the model is

$$\text{Minimize} \quad \sum_{(i,j)\in A} c_{ij}y_{ij} \tag{16.3a}$$

subject to

$$\sum_{1\le j\le n} y_{ij} = 1 \qquad \text{for all } i = 1, 2, \ldots, n, \tag{16.3b}$$

$$\sum_{1\le i\le n} y_{ij} = 1 \qquad \text{for all } j = 1, 2, \ldots, n, \tag{16.3c}$$

$$\mathcal{N}x = b, \tag{16.3d}$$

$$x_{ij} \le (n - 1)y_{ij} \qquad \text{for all } (i, j) \in A, \tag{16.3e}$$

$$x_{ij} \ge 0 \qquad \text{for all } (i, j) \in A, \tag{16.3f}$$

$$y_{ij} = 0 \text{ or } 1 \qquad \text{for all } (i, j) \in A. \tag{16.3g}$$

To interpret this formulation, let $A' = \{(i, j) : y_{ij} = 1\}$ and let $A'' = \{(i, j) : x_{ij} > 0\}$. The constraints (16.3b) and (16.3c) imply that exactly one arc of A' leaves and enters any node i; therefore, A' is the union of node disjoint cycles containing all of the nodes of N. In general, any integer solution satisfying (16.3b) and (16.3c) will be the union of disjoint cycles; if any such solution contains more than one cycle, we refer to each of the cycles as subtours, since they pass through only a subset of the nodes. Figure 16.8 gives an example of a subtour solution to the constraints (16.3b) and (16.3c).

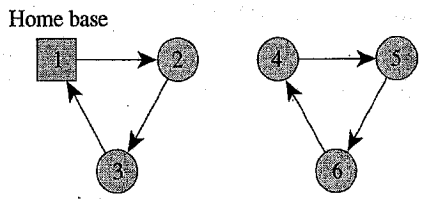


Figure 16.8 Infeasible solution for the traveling salesman problem containing subtours.

Constraint (16.3d) ensures that A'' is connected since we need to send 1 unit of flow from node 1 to every other node via arcs in A''. The "forcing" constraints (16.3e) imply that A'' is a subset of A'. [Notice that since no arc need ever carry more than $(n - 1)$ units of flow, the forcing constraint for arc (i, j) is redundant if $y_{ij} = 1$.] These conditions imply that the arc set A' is connected and so cannot contain any subtours. We conclude that the formulation (16.3) is a valid formulation for the traveling salesman problem.

One of the nice features of this formulation is that we can apply Lagrangian relaxation to it in several ways. For example, suppose that we attach Lagrange multipliers $\mu_{ij} \ge 0$ with the forcing constraints (16.3e) and bring them into the objective function, giving the Lagrangian objective function

$$\text{Minimize} \sum_{(i,j)\in A} [c_{ij} - (n-1)\mu_{ij}]y_{ij} + \sum_{(i,j)\in A} \mu_{ij}x_{ij},$$

and leaving (16.3b)–(16.3d), (16.3f), and (16.3g) as constraints in the Lagrangian subproblem. Note that nothing in this Lagrangian subproblem couples the variables y_{ij} and x_{ij}. Therefore, the subproblem decomposes into two separate subproblems: (1) an assignment problem in the variables y_{ij}, and (2) a minimum cost flow problem in the variables x_{ij}. So for any choice of the Lagrangian multipliers μ, we solve two network flow subproblems; by using subgradient optimization we can find the best lower bound and optimal values of the multipliers. By relaxing other constraints in this model, or by applying Lagrangian relaxation to other formulations of the traveling salesman problem, we could define other network flow subproblems (see Exercise 16.19).

Application 16.3 Vehicle Routing

The vehicle routing problem is a generic model that practitioners encounter in many problem settings including the delivery of consumer products to grocery stores, the collection of money from vending machines and telephone coin boxes, and the delivery of heating oil to households. As we have noted earlier in this section, the vehicle routing problem is a generalization of the traveling salesman problem.

The vehicle routing problem is easy to state: Given (1) a fleet of K capacitated vehicles domiciled at a common depot, say node 1, (2) a set of customer sites $j = 2, 3, \ldots, n$, each with a prescribed demand d_j, and (3) a cost c_{ij} of traveling from location i to location j, what is the minimum cost set of routes for delivering (picking up) the goods to the customer sites? We assume that the vehicle fleet is homogeneous and that each vehicle has a capacity of u units.

There are many different variants on this core vehicle routing problem. For example, the vehicle fleet might be nonhomogeneous, each vehicle route might have a total travel time restriction, or deliveries for each customer might have time window restrictions (earliest and latest delivery times). We illustrate the use of Lagrangian relaxation by considering only the basic model, which we formulate with decision variables x_{ij}^k indicating whether ($x_{ij}^k = 1$) or not ($x_{ij}^k = 0$) we dispatch vehicle k on arc (i, j) and y_{ij} indicating whether some vehicle travels on arc (i, j):

$$\text{Minimize} \sum_{1\le k\le K} \sum_{(i,j)\in A} c_{ij}x_{ij}^k \tag{16.4a}$$

subject to

$$\sum_{1\le k\le K} x_{ij}^k = y_{ij}, \tag{16.4b}$$

$$\sum_{1\le j\le n} y_{ij} = 1 \qquad \text{for } i = 2, 3, \ldots, n, \tag{16.4c}$$

$$\sum_{1\le i\le n} y_{ij} = 1 \qquad \text{for } j = 2, 3, \ldots, n, \tag{16.4d}$$

$$\sum_{1\le j\le n} y_{1j} = K, \tag{16.4e}$$

$$\sum_{1\le i\le n} y_{i1} = K, \tag{16.4f}$$

$$\sum_{2 \le i \le n} \sum_{1 \le j \le n} d_i x_{ij}^k \le u \qquad \text{for all } k = 1, 2, \ldots, K, \tag{16.4g}$$

$$\sum_{i \in Q} \sum_{j \in Q} y_{ij} \le |Q| - 1 \qquad \text{for all subsets } Q \text{ of } \{2, 3, \ldots, n\}, \tag{16.4h}$$

$$y_{ij} = 0 \text{ or } 1 \qquad \text{for all } (i,j) \in A, \tag{16.4i}$$

$$x_{ij}^k = 0 \text{ or } 1 \qquad \text{for all } (i,j) \in A \text{ and all } k = 1, 2, \ldots, K. \tag{16.4j}$$

Let $A' = \{(i, j) : y_{ij} = 1\}$. As in our discussion of the traveling salesman problem, constraints (16.4c) and (16.4d) ensure that A' is the union of node disjoint cycles containing all of the nodes in N. Constraint (16.4h) ensures that the solution must contain no cycle using the nodes 2, 3, . . . , n (i.e., not contain any subtours on these nodes); otherwise, the arcs A' would contain some cycle passing through a set Q of nodes and the solution would violate constraint (16.4h) since the left-hand side of the constraint (16.4h) would be at least $|Q|$. For this reason, we refer to the constraints (16.4h) as *subtour breaking constraints*.

We might note that if $K = 1$, and u is so large that the constraint (16.4g) is redundant, this model becomes an "assignment-based" formulation of the traveling salesman problem, which is an alternative formulation to the "flow-based" model that we introduced previously as (16.3). In Exercise 16.24 we study the relationships between these formulations as well as a third model, a multicommodity flow-based formulation.

Note that this formulation has several embedded structures that we might exploit in a Lagrangian relaxation solution approach. By relaxing some of the constraints, we are also able to decompose the problem into independent subproblems. For example, if we relax only constraints (16.4b), no constraint connects the x variables and y variables, so the problem decomposes into separate subproblems in each of these variables. By relaxing different combinations of the constraints, we create several different types of subproblems:

1. If we relax the constraints (16.4b), (16.4g), and (16.4h), the resulting formulation is an assignment problem.
2. If we relax the constraints (16.4b) to (16.4f), and (16.4h), the resulting problem decomposes into independent "knapsack problems," one for each vehicle k.
3. If we relax constraint (16.4b), the problem decomposes into separate subproblems, one in the y variables and one in the x^k variables for each vehicle k. The first of these problems is a so-called K-traveling salesman problem (see Exercise 16.25) and each problem in the variables x^k is a knapsack problem.
4. If we relax the assignment constraints (16.4c) to (16.4f), the constraint (16.4b) defining y, and the capacity constraint (16.4g), the resulting problem is a minimum forest problem on the nodes 2, 3, . . . , n. This problem is easy to solve by a simple variant of any minimum spanning tree algorithm. We could strengthen this approach by adding other (redundant) constraints to the problem formulation (see Exercise 16.28).
5. If we relax constraints (16.4b), (16.4c), and (16.4e) to (16.4g), the subproblem with the constraints (16.4d), (16.4h), and (16.4i) becomes a directed minimum spanning tree problem—any feasible solution will be a directed spanning tree

with exactly one arc directed into each node (except for the root node 1). Although we do not consider this problem in this book, it is polynomially solvable.

6. If we relax the constraint (16.4g), the problem becomes a variant of the K-traveling salesman problem.

These various possibilities illustrate the remarkable flexibility of the Lagrangian relaxation solution approach.

Application 16.4 Network Design

Suppose that we have the flexibility of designing a network as well as determining its optimal flow (routing). That is, we have a directed network $G = (N, A)$ and can introduce an arc or not into the design of the network: If we use (introduce) an arc (i, j), we incur a design (construction) cost f_{ij}. Our problem is to find the design that minimizes the total systems cost—that is, the sum of the design cost and the routing cost. This type of model arises in many application contexts, for example, the design of telecommunication or computer networks, load planning in the trucking industry (i.e., the design of a routing plan for trucks), and the design of production schedules.

Many alternative modeling assumptions arise in practice. We consider one version of the problem, the *uncapacitated network design problem*. In this model we need to route multiple commodities on the network; each commodity k has a single source node s^k and a single destination node d^k. Once we introduce an arc (i, j) into the network, we have sufficient capacity to route all of the flow by all commodities on this arc.

To formulate this problem as an optimization model, let x^k denote the vector of flows of commodity k on the network. Rather than letting x_{ij}^k model the total flow of commodity k on arc (i, j), however, we let x_{ij}^k denote the fraction of the required flow of commodity k to be routed from the source s^k to the destination d^k that flows on arc (i, j). Let c^k denote the cost vector for commodity k, which we scale to reflect the way that we have defined x_{ij}^k [i.e., c_{ij}^k is the per unit cost for commodity k on arc (i, j) times the flow requirement of that commodity]. Also, let y_{ij} be a zero–one vector indicating whether or not we select arc (i, j) as part of the network design. Using this notation, we can formulate the network design problem as follows:

$$\text{Minimize} \quad \sum_{1 \le k \le K} c^k x^k + fy \tag{16.5a}$$

subject to

$$\sum_{\{j:(i,j)\in A\}} x_{ij}^k - \sum_{\{j:(j,i)\in A\}} x_{ji}^k = \begin{cases} 1 & \text{if } i = s^k \\ -1 & \text{if } i = d^k \\ 0 & \text{otherwise} \end{cases} \quad \text{for all } i \in N,\ k = 1, 2, \ldots, K, \tag{16.5b}$$

$$x_{ij}^k \le y_{ij} \qquad \text{for all } (i, j) \in A,\ k = 1, 2, \ldots, K, \tag{16.5c}$$

$$x_{ij}^k \geq 0 \qquad \text{for all } (i, j) \in A \text{ and all } k = 1, 2, \ldots, K, \tag{16.5d}$$

$$y_{ij} = 0 \text{ or } 1 \qquad \text{for all } (i, j) \in A. \tag{16.5e}$$

In this formulation, the "forcing constraints" (16.5c) state that if we do not select arc (i, j) as part of the design, we cannot flow any fraction of commodity k's demand on this arc, and if we do select arc (i, j) as part of the design, we can flow as much of the demand of commodity k as we like on this arc.

Note that if we remove the forcing constraints from this model, the resulting model in the flow variables x^k decomposes into a set of independent shortest path problems, one for each commodity k. Consequently, the model is another attractive candidate for the application of Lagrangian relaxation. To see why this type of solution approach might be attractive, consider a typically sized problem with, say, 50 nodes and 500 candidate arcs. Suppose that we have a separate commodity for each pair of nodes (as is typical in communication settings in which each node is sending messages to every other node). Then we have 50(49) = 2450 commodities. Since each commodity can flow on each arc, the model has 2450(500) = 1,225,000 flow variables, and since (1) each flow variable defines a forcing constraint, and (2) each commodity has a flow balance constraint at each node, the model has 1,225,000 + 2450(50) = 1,347,500 constraints. In addition, it has 500 zero–one variables. So even as a linear program, this model far exceeds the capabilities of current state of the art software systems. By decomposing the problem, however, for each choice of the vector of Lagrange multipliers, we will solve 2450 small shortest path problems.

Application 16.5 Two-Duty Operator Scheduling

In many different problem contexts in work force planning, a private firm or public-sector organization must schedule its employees—for example, nurses, airline crews, telephone operators—to provide needed services. Typically, the problems are complicated by complex work rules, for example, airline crews have limits on the number of hours that they can fly in any week or month. Moreover, frequently, the demand for the services of these employees varies considerably by time of the day or week, or across geography (as in the case of airline crew scheduling). Consequently, finding a minimum cost schedule requires that we balance the prevailing work rules with the demand patterns. Figure 16.9 shows one example of a work force planning problem, which we will view as a driver schedule for a single bus line.

Every column in this table corresponds to a possible schedule. For example, in schedule 1, a driver operates the bus line in two shifts, from 8 to 11 and then from 1 to 3; in schedule 2 the driver works a single shift, from 11 to 1, and in schedule 3, he/she drives from 3 to 6. As indicated by the column entitled demand in the table, we wish to find a set of schedules satisfying the property that at least one driver is assigned to the bus at every hour of the day from 8 A.M. until 6 P.M. (if two drivers are assigned to the same bus at the same time, one drives and the other is a rider). One possibility is to choose schedules 1, 2, and 3; another is schedules 4 and 5; and still another is schedules 3, 5, and 6. Each schedule j has an associated

Time period	Schedule								Demand
	1	2	3	4	5	6	7	8	
8–9	1	0	0	1	0	1	0	1	≥ 1
9–10	1	0	0	1	0	1	0	1	≥ 1
10–11	1	0	0	0	1	0	0	1	≥ 1
11–12	0	1	0	0	1	0	0	1	≥ 1
12–1	0	1	0	0	1	0	0	1	≥ 1
1–2	1	0	0	1	0	1	1	0	≥ 1
2–3	1	0	0	1	0	1	1	0	≥ 1
3–4	0	0	1	1	0	0	1	0	≥ 1
4–5	0	0	1	1	0	0	1	0	≥ 1
5–6	0	0	1	1	0	0	1	0	≥ 1
Cost	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	

Figure 16.9 Two-duty operator schedule.

cost c_j and we wish to choose the set of schedules that meets the scheduling requirement at the lowest possible cost. To formulate this problem formally as an optimization model, let x_j be a binary (i.e., zero–one) variable indicating whether ($x_j = 1$) or not ($x_j = 0$), we choose schedule j, and let $\mathcal{A}$ denote the zero–one matrix of coefficients of the scheduling table (i.e., the ijth element is 1 if schedule j has a driver on duty during the ith hour of the day). Also, let e denote a column of 1's. Then the model is

$$\text{Minimize} \quad cx \tag{16.6a}$$

subject to

$$\mathcal{A}x \geq e, \tag{16.6b}$$

$$x_j = 0 \text{ or } 1 \qquad \text{for } j = 1, 2, \ldots, n. \tag{16.6c}$$

The choice of the available schedules in the problem depends on the governing work rules; as an illustration, in our example, no operator works in any shift of less than 2 hours. Moreover, note that the schedules permit split shifts, that is, time on, time off, and then time on again as in schedule 1. Note, however, that no schedule has more than two shifts. We refer to this special version of the general operator scheduling problem as the *two-duty operator scheduling problem.*

In Exercise 4.13 we showed how to solve the single-duty scheduling problem as a shortest path problem: The shortest path model contains a node for each time period $1, 2, \ldots, T$ to be covered, plus an artificial end node $T + 1$, and an arc from node i to node j whenever a schedule starts at the beginning of time period i and ends at the beginning of time period j. We interpret arc (i, j) as "covering" the time periods $i, i + 1, \ldots, j - 1$. The network also contains "backward" arcs of the form $(j + 1, j)$ that permits us to "back up" from time period $j + 1$ to time period j so that we can cover any node more than once and model the possibility that a schedule might assign more than one driver to any time period. Can we use

Lagrangian relaxation to exploit the fact that the single-duty problem is a shortest path problem? To do so, we will use an idea known as *variable splitting*.

Consider any column j of the matrix $\mathcal{A}$ that contains two sequences of 1's—that is, corresponds to a schedule with two shifts. Let us make two columns $\mathcal{A}'_j$ and $\mathcal{A}''_j$ out of this column; each of these columns contains one of the duties (sequence of 1's) from $\mathcal{A}_j$, so $\mathcal{A}_j = \mathcal{A}'_j + \mathcal{A}''_j$. Let us also replace the variable x_j in our model with two variables x'_j and x''_j. We form a new model with these variables as

$$\text{Minimize} \quad c'x' + c''x'' \tag{16.7a}$$

subject to

$$\mathcal{A}'x' + \mathcal{A}''x'' \geq e, \tag{16.7b}$$

$$x' - x'' = 0, \tag{16.7c}$$

$$x'_j \text{ and } x''_j = 0 \text{ or } 1 \qquad \text{for } j = 1, 2, \ldots, n. \tag{16.7d}$$

For convenience, in formulating this model we have assumed that we have split every column of the matrix $\mathcal{A}$. If not, we can simply assume that some columns of $\mathcal{A}''$ are columns of zeros. Moreover, we can split the cost of each variable x_j arbitrarily between c'_j and c''_j. For example, we could let each of these costs be half of c_j. This model and the original model (16.6) are clearly equivalent. Note, however, that the new model reveals embedded network structure; as shown in Figure 16.10, which is the network model associated with the data in Figure 16.9, the model is a shortest path problem with the complicating constraints that we need to choose arcs in pairs: We choose either both or none of the arcs corresponding to the variables x'_j and x''_j. If we eliminate the "complicating" constraint $x' - x'' = 0$, the problem becomes an easily solvable single duty scheduling problem, which as we have seen before, we can solve via a shortest path computation. This observation suggests that we adopt a Lagrangian relaxation approach, relaxing the constraints with a Lagrange multiplier μ so that the Lagrangian subproblem has the objective function

$$L(\mu) = \min(c' + \mu)x' + (c'' - \mu)x''. \tag{16.8}$$

Now, as usual to solve the Lagrangian multiplier problem, we apply subgradient

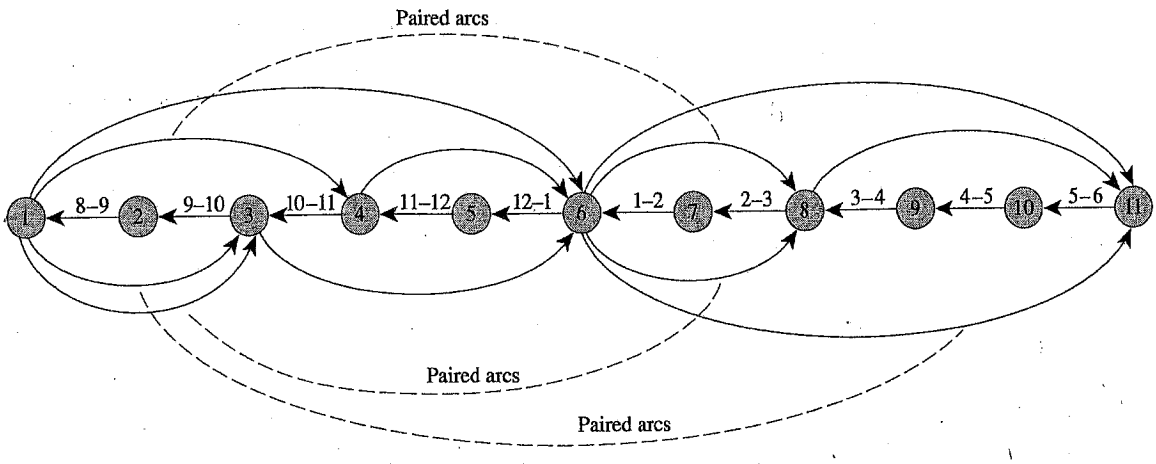


Figure 16.10 Shortest path subproblem for the two duty scheduling problem.

optimization, or some other solution technique, to maximize $L(\mu)$ over all possible choices of the Lagrange multipliers μ.

When we split a column A_j into two columns A_j' and A_j'', it does not matter how we split the cost c_j between c_j' and c_j'', as long as $c_j = c_j' + c_j''$. Since $x_j' = x_j''$ in any feasible solution, the cost of any feasible solution will be the same no matter how we allocate the cost. However, cost splitting does make a significant difference in the relaxed problem obtained by dropping the constraint $x_j' = x_j''$. If we were to make c_j' large and c_j'' small, then in the solution to the relaxed problem we would probably find that x_j' would be 0 and x_j'' would be 1. Similarly, if we made c_j' small and c_j'' large, in the solution to the relaxed problem we would likely find that x_j' would be 1 and x_j'' would be 0. Ideally, we should allocate the costs between c_j' and c_j'', so that either $x_j' = x_j'' = 0$ or $x_j' = x_j'' = 1$ in the relaxed problem.

As it turns out, we need not worry about the cost allocation at all if we use Lagrangian relaxation since the Lagrange multiplier μ_j for the constraint $x_j' - x_j'' = 0$ does the cost allocation. Suppose, for example, that $c_j' = c_j$ and $c_j'' = 0$. Then, since we are relaxing the constraint $x_j' - x_j'' = 0$, the coefficient of x_j' in the relaxed problem is $c_j + \mu_j$, and the coefficient of x_j'' is $-\mu_j$. As μ_j ranges over the real numbers, we obtain all possible ways of splitting the cost c_j between c_j' and c_j''.

The operator scheduling problem we have considered permits us to find an optimal schedule of drivers for a single bus line. If we wish to schedule several bus lines simultaneously, the right-hand-side coefficients in the constraints (16.6) will be arbitrary positive integers, indicating the number of required operators for each time period during the day. In this instance, the variable splitting device still permits us to use Lagrangian relaxation and network optimization to solve the problem. In this instance, the Lagrangian subproblems will be minimum cost flow problems rather than shortest path problems.

As this application shows, embedded network flow structure is not always so apparent and, consequently, the use of Lagrangian relaxation often requires considerable ingenuity in model formulation. Indeed, the application of Lagrangian relaxation typically requires considerable skill in modeling. Moreover, as several of our examples have shown, we often can formulate network optimization problems in several different ways, and by doing so we might be able to recognize and exploit different network substructures. The models we have proposed for the traveling salesman problem, both in the discussion of this problem and in the discussion of the vehicle routing problem, illustrate these possibilities. As a result, the design and implementation of Lagrangian relaxation algorithms often require careful choices concerning the "best" models to use and the "best" constraints to relax. The literature that we cite in the reference notes gives some guidance concerning these issues; successful prior applications, such as those that we have discussed in this section and in the exercises at the end of this chapter, provide additional guides.

Application 16.6 Degree-Constrained Minimum Spanning Trees

Suppose that we wish to find a minimum spanning tree of a network, but with the added provision that the tree contain exactly k arcs incident to a given root node, say node 1 (in some settings, the degree of the root node should be at most k). This

degree-constrained minimum spanning tree problem arises in several applications. For example, in computer networking, the root node might be a central processor with a fixed number of ports and the other nodes might be terminals that we need to connect to the processor. In the communication literature, this problem has become known as the *teleprocessing design problem* or as the *multidrop terminal layout problem*. The vehicle routing problem, described in Application 16.3, provides another application setting. If we are routing k vehicles and we delete the last arc from every route, every solution is a spanning tree with k arcs incident to the depot (the tree has additional structure: each subtree off the root is a single path). Therefore, the degree constrained minimum spanning tree problem is a relaxation of the vehicle routing problem. Note that this relaxation is stronger than the minimum spanning tree relaxation that we discussed in Application 16.3.

We might formulate the degree-constrained minimum spanning tree problem as follows.

$$\text{Minimize} \quad cx$$

subject to

$$\sum_{j=2}^{n} x_{1j} = k,$$

$$x \in X.$$

In this formulation, $x = (x_{ij})$ is a vector of decision variables and each x_{ij} is a zero–one variable indicating whether ($x_{ij} = 1$) or not ($x_{ij} = 0$), arc (i, j) belongs to the spanning tree. The number c_{ij} denotes the fixed cost of installing arc (i, j) and the set X denotes the set of incidence vectors of spanning trees. The additional constraint states that the degree of node 1 must be k. Let $C = \max\{c_{ij} : (i, j) \in A\}$.

To solve this problem, we might use Lagrangian relaxation. If we associate a Lagrange multiplier μ with the degree constraint and relax it, the objective function of the Lagrangian subproblem becomes $cx + \mu \sum_{j=2}^{n} x_{1j} - \mu k$ and the remaining (implicit) constraint, $x \in X$, states that the vector x defines a spanning tree. Note that if we ignore the last term, μk, which is a constant for any fixed value of μ, this problem is a parametric minimum spanning tree problem: for each j, the cost of arc $(1, j)$ is $c_{1j} + \mu$, and whenever $i \neq 1$ and $j \neq 1$, the cost of arc (i, j) is c_{ij}. We will use this observation to solve the degree constrained problem. That is, rather than using subgradient optimization, we will use a combinatorial algorithm to solve the Lagrangian multiplier problem.

We first solve the minimum spanning tree problem for $\mu = 0$. If the degree of node 1 in the optimal tree equals k, this tree is optimal for the degree-constrained minimum spanning tree problem. So suppose that the degree of node 1 is different than k. We first consider the case when the degree of node 1 is strictly less than k. Notice that since μ affects the lengths of only those arcs incident to node 1, changing the value μ affects the ranking of these arcs relative to the arcs not incident to node 1. Consequently, as we decrease the value of μ, the arcs incident to node 1 become more attractive relative to the other arcs, so we would insert these arcs into the spanning tree in place of the other arcs. The algorithm uses this observation: It starts with a minimum spanning tree T^1 for $\mu = 0$ and by decreasing the value of μ, it

generates a sequence of spanning trees $T^1, \ldots, T^{q-1}$, terminating with a minimum spanning tree T^q for $\mu = -C - 1$. Each tree $T^1, \ldots, T^{q-1}$ is a minimum spanning tree for some value of μ. The algorithm creates T^j from T^{j-1} by adding one arc $(1, i)$ to T^{j-1} and deleting one arc (p, q) with $p \neq 1$ and $q \neq 1$ from T^{j-1}. That is, at each step it increases the degree of node 1 by one. Finally, T^q includes all the arcs incident to node 1. (For a discussion of parametric minimum spanning trees, see Exercises 13.35 and 13.36.)

Let T^k denote the tree containing exactly k arcs incident to node 1 and let μ^k denote the value of μ for which T^k is a minimum spanning tree for the parametric problem. Further, let x^k denote the incidence vector associated with the spanning tree T^k. By definition, x^k solves the Lagrangian multiplier problem $L(\mu) = cx + \mu \sum_{j=2}^{n} x_{ij} - \mu k$, $x \in X$, for $\mu = \mu^k$ because μk is a constant. Now notice that $L(\mu^k) = cx^k + \mu^k \sum_{j=2}^{n} x_{1j} - \mu^k k = cx^k + \mu^k k - \mu^k k = cx^k$, which implies that for $\mu = \mu^k$, the optimal objective function value of the Lagrangian subproblem equals the value of a feasible solution x^k of the degree-constrained minimum spanning tree problem. Property 16.3 shows that x^k is an optimal solution of the degree-constrained minimum spanning tree problem.

When the optimal tree for $\mu = 0$ contains more than k arcs, we parametrically increase the value of μ until $\mu = C + 1$. As we increase the value of μ, the arcs incident on node 1 become less attractive, and they leave the optimal tree one by one. Eventually, node 1 will have degree exactly equal to k, and the tree at this point will be a minimum degree-constrained spanning tree.

Note that this application of Lagrangian relaxation is different than the others that we have considered in this chapter. In this case we have used Lagrangian relaxation to define a parametric problem that is related to the constrained model we are considering. We have then used a combinatorial algorithm rather than a general-purpose Lagrangian relaxation algorithm to solve the parametric problem. In this case the Lagrangian relaxation has proven to be valuable not only in formulating the parametric problem, but also in validating that the solution generated by the parametric problem is optimal for the constrained model.

Application 16.7 Multi-item Production Planning

In production planning we would like to find the best use of scarce resources (people, machinery, space) in order to meet customer demand at the least possible cost. As we show in Chapter 19, the research community has developed a number of different models for addressing various planning issues in this application domain. Some of these models are shortest path problems and some are minimum cost flow problems; still others are multicommodity flow problems or more general models with embedded network flow structure. In this section, to show how we might use Lagrangian relaxation to solve more general models, we consider two applications of production planning: multi-item production planning and production planning with changeover costs.

Suppose that we are producing K items over a planning horizon containing T periods (e.g., production shifts). Suppose, further, that we produce the items on the same machine and that we can produce at most one item in each period. We would

like to find the least cost production plan that will satisfy a demand d_{kt} for every item k in each period t.

Let x_{kt} denote the amount of item k that we produce in period t and let I_{kt} denote the amount of inventory of item k that we carry from period t to period $t + 1$. Let z_{kt} be a zero–one variable indicating whether or not we produce item k in period t. With this notation, we can model the multi-item production planning problem as follows:

$$\text{Minimize} \quad \sum_{k=1}^{K}\sum_{t=1}^{T} c_{kt}x_{kt} + \sum_{k=1}^{K}\sum_{t=1}^{T} h_{kt}I_{kt} + \sum_{k=1}^{K}\sum_{t=1}^{T} F_{kt}z_{kt} \tag{16.9a}$$

subject to

$$\sum_{k=1}^{K} z_{kt} \le 1 \qquad \text{for } t = 1, 2, \ldots, T, \tag{16.9b}$$

$$x_{kt} + I_{k,t-1} - I_{kt} = d_{kt} \qquad \text{for } k = 1, 2, \ldots, K \text{ and } t = 1, 2, \ldots, T, \tag{16.9c}$$

$$x_{kt} \le P_{kt}z_{kt} \qquad \text{for } k = 1, 2, \ldots, K \text{ and } t = 1, 2, \ldots, T, \tag{16.9d}$$

$$x_{kt} \ge 0, \quad I_{kt} \ge 0 \qquad \text{for } k = 1, 2, \ldots, K \text{ and } t = 1, 2, \ldots, T, \tag{16.9e}$$

$$z_{kt} = 0 \text{ or } 1 \qquad \text{for } k = 1, 2, \ldots, K \text{ and } t = 1, 2, \ldots, T. \tag{16.9f}$$

In this model c_{kt} is the per unit production cost and h_{kt} is the per unit inventory carrying cost for item k in period t. F_{kt} is a fixed cost that we incur if we produce item k in period t and P_{kt} is the production capacity for item k in period t. The constraint (16.9a) ensures that we produce at most one item in each period. Constraint (16.9c) states that we allocate the amount we have on hand of item k in period t (i.e., the production plus incoming inventory of that item) either to demand in that period or to inventory at the end of the period. The "forcing" constraint (16.9d) ensures that the quantity x_{kt} of item k produced in period t is zero if we do not select that item for production in that period, that is, if $z_{kt} = 0$; this constraint also ensures that the production of item k in period t never exceeds the production capacity of that item.

Note that constraints in (16.9b) are the only constraints in this model that link various items. Therefore, these constraints would be attractive candidates to relax via Lagrange multipliers λ_t. Doing so creates the following objective function

$$\text{Minimize} \quad \sum_{k=1}^{K}\sum_{t=1}^{T} c_{kt}x_{kt} + \sum_{k=1}^{K}\sum_{t=1}^{T} h_{kt}I_{kt} + \sum_{k=1}^{K}\sum_{t=1}^{T} [F_{kt} + \lambda_t]z_{kt} - \sum_{t=1}^{T} \lambda_t.$$

For a fixed value of the Lagrange multipliers, the last term is a constant, so the problem separates into a single-item production planning problem for each item k; the production and inventory carrying costs in the relaxation are the same as those in the original model, and in each period the fixed cost for each item in the relaxation is λ_t units more than in the original model.

The subproblems assume different forms, depending on the nature of the production capacities. In Chapter 19 we show that whenever each single-item subproblem is uncapacitated (i.e., P_{kt} is as large as the sum of the demands d_{kt} in periods $t + 1, t + 2, \ldots, T$), we can solve each single-item subproblem as a shortest path problem. If we impose production capacities, the subproblems are NP-complete. In

these instances, since the number of time periods is often very small, we might use a dynamic programming approach for solving the subproblems.

To conclude this discussion, we might note that we can enrich this basic multi-item production planning model in a variety of ways. For example, as shown in Chapter 19, we can model multiple stages of production or the backlogging of demand. As another example, we can model situations in which we incur a startup cost whenever we initiate the production of a new item. To model this situation, we let y_{kt} be a zero–one variable, indicating whether or not the production system switches from not producing item k in period $t - 1$ to producing the item in period t. We then add the following constraints to the basic model (16.9):

$$z_{kt} - z_{k,t-1} \leq y_{kt} \qquad \text{for } k = 1, 2, \ldots, K \text{ and } t = 1, 2, \ldots, T,$$

and for each "turn on" variable y_{kt}, we add a cost term $\alpha_{kt} y_{kt}$ to the objective function (α_{kt} is the cost for turning on the machine to produce item k in period t). By relaxing these constraints as well as the item choice constraints (16.9b), we again obtain separate production planning problems for each item. Or, by relaxing only the item choice constraints, we obtain a single-item production planning problem in which we incur three types of production costs: (1) a cost for turning the machine on, (2) a cost for setting up the machine in any period to produce any amount of the item, and (3) a per unit production cost.

This startup cost problem is important in many practical production settings. Moreover, this model is illustrative of the enhancements that we can make to the basic production planning problem and once again demonstrates the algorithmic flexibility of Lagrangian relaxation.

16.6 SUMMARY

Lagrangian relaxation is a flexible solution strategy that permits modelers to exploit the underlying structure in any optimization problem by relaxing (i.e., removing) complicating constraints. This approach permits us to "pull apart" models by removing constraints and instead place them in the objective function with associated Lagrange multipliers. In this chapter we have developed the core theory of Lagrangian relaxation, described popular solution approaches, and examined several application contexts in which Lagrangian relaxation effectively exploits network substructure.

The starting point for the application and theory of Lagrangian relaxation (as applied to a model specified as a minimization problem) is a key bounding principle stating that for any value of the Lagrange multiplier, the optimal value of the relaxed problem, called the Lagrangian subproblem, is always a lower bound on the objective function value of the problem. To obtain the best lower bound, we need to choose the Lagrangian multiplier so that the optimal value of the Lagrangian subproblem is as large as possible. We call this problem the Lagrangian multiplier problem. We can solve the Lagrangian multiplier problem in a variety of ways. The subgradient optimization technique is possibly the most popular technique for solving the Lagrangian multiplier problem and we have described this technique in some detail. The subgradient optimization technique solves a sequence of Lagrangian subproblems.

Usually, we choose the constraints to relax so that the Lagrangian subproblem is much easier to solve than the original problem. Consequently, when applying Lagrangian relaxation, we solve many "simple" problems instead of one single "complicated" problem. Frequently, the complicating constraints that we relax are the only constraints that couple otherwise independent subsystems (e.g., shortest path problems); in these instances, Lagrangian relaxation permits us to decompose a problem into smaller, more tractable subproblems. For this reason, the research community often refers to Lagrangian relaxation as a decomposition technique.

In discussing the theory of Lagrangian relaxation, we showed how to formulate the Lagrangian multiplier problem as an associated linear program with a large number of constraints; we also showed how to interpret the Lagrangian multiplier problem as a convexification of the original optimization model. That is, instead of restricting our choices to a discrete set of possible alternatives (e.g., spanning tree solutions), the multiplier problem produces the same objective function value that we would obtain if we solved the original problem, but permitted the use of convex combinations of the alternatives. We also showed that when applied to integer programs, the Lagrangian relaxation always gives at least as large a lower bound as does the linear programming relaxation of the problem. Finally, we showed that whenever the Lagrangian subproblem satisfies the integrality property (so it has an integer solution for all values of the Lagrange multiplier), solving the Lagrange multiplier problem is equivalent to solving the linear programming relaxation of the original optimization model. In these instances, even though the Lagrangian approach provides the same lower bound as the linear programming relaxation, it does have the ability to solve network (or other) subproblems quickly, which is often greatly preferred to solving the original problem by general-purpose linear programming codes.

Our discussion of applications has introduced several important network optimization models: networks with side constraints, the traveling salesman problem, vehicle routing, network design, personnel scheduling, degree-constrained minimum spanning trees, and production planning. As we have seen, these optimization models have applications in such diverse settings as machine scheduling, communication system design, delivery of consumer goods, telephone coin box collection, telephone operator scheduling, logistics, and production. Consequently, our discussion has illustrated the broad applicability of Lagrangian relaxation across many practical problem contexts. It has also illustrated the versatility of Lagrangian relaxation and its ability to exploit the core network substructures—shortest paths, minimum cost flows, the assignment problem, and minimum spanning tree problems—that we have studied in previous chapters. Our discussion of applications has also highlighted several other points:

1. *Need for creative modeling.* Formulating Lagrangian relaxations can require considerable ingenuity in modeling (as in the variable splitting device that we used to study the two-duty operator scheduling problem).
2. *Flexibility of Lagrangian relaxation.* In many models, such as the vehicle routing problem, we can obtain a variety of different Lagrangian subproblems by relaxing different constraints. This variety of potential subproblems permits us to develop different algorithms for solving the same problem.

3. *Use of Lagrangian relaxation as a conceptual as well as algorithmic tool.* On some occasions, as in our discussion of the degree-constrained minimum spanning tree problem, we can use the bounding information provided by Lagrangian relaxation as a stand-alone tool that is unrelated to any iterative method for solving the Lagrangian multiplier problem. For example, we can use the bounds to analyze the solutions generated by combinatorial or heuristic algorithms for solving a problem.

REFERENCE NOTES

The Lagrange multiplier technique of nonlinear optimization dates to the eighteenth century and was suggested by the famous mathematician Lagrange, for whom the technique is named. The use of this technique in integer programming and discrete optimization is much more recent, originating in the seminal papers by Held and Karp [1970, 1971], who studied the traveling salesman problem. Everett's [1963] development of Lagrangian multiplier methods for general mathematical programming problems was a precursor to this development. Held and Karp's application of the Lagrange multiplier method was not only an eye-opening successful application, but also set out many key ideas in applying the method to integer programming problems. Fisher [1981, 1985], Geoffrion [1974], and Shapiro [1979] provide insightful surveys of Lagrangian relaxation and its uses in integer programming. The papers by Fisher contain many citations to successful applications in a wide variety of problem settings. For a discussion of the branch-and-bound algorithm, see Winston [1991].

Most of the key results of Lagrangian relaxation (e.g., the bounding properties and optimality conditions) are special cases of more general results in mathematical programming duality theory. Rockafellar [1970] and Stoer and Witzgall [1970] provide comprehensive treatments of this subject. Magnanti, Shapiro, and Wagner [1976] establish the equivalence of the Lagrangian multiplier problem and generalized linear programming, whose development by Dantzig and Wolfe [1961] predates the formal development of Lagrangian relaxation in integer programming. The integrality property is due to Geoffrion [1974]. The subgradient method is an outgrowth of so-called relaxation methods for solving systems of linear inequalities. Bertsimas and Orlin [1991] have developed the most efficient algorithms (in the worst-case sense) for solving many classes of Lagrangian relaxation problems.

Several of the application contexts that we have discussed in Section 16.5 and in the exercises have very extensive literatures. The following books and survey articles, which contain many references to the literature, serve as good sources of information on these topics.

Traveling salesman problem: the book edited by Lawler, Lenstra, Rinnooy Kan, and Shmoys [1985]

Vehicle routing: surveys by Bodin, Golden, Assad, and Ball [1983], Laporte and Nobert [1987], and Magnanti [1981]

Network design: surveys by Magnanti and Wong [1984], Magnanti, Wolsey, and Wong [1992], and Minoux [1989]

Production planning: the survey paper by Shapiro [1992], the book by Hax and Candea [1984], and the paper by Graves [1982]

Several other of the applications discussed in this chapter are adapted from research papers from the literature. For a Lagrangian relaxation-based branch-and-bound approach to the constrained shortest path problem, see Handler and Zang [1980]. Shepardson and Marsten [1980] have used the variable splitting device and Lagrangian relaxation for solving the two-duty operator scheduling problem and applied this approach to bus operator scheduling. For an algorithmic approach to the network design problem, see Balakrishnan, Magnanti, and Wong [1989a]. Volgenant [1989] considers the degree-constrained minimum spanning tree problem.

EXERCISES

16.1. Lagrangian relaxation and inequality constraints. To develop the Lagrangian multiplier problem for an inequality constraint problem stated as $\min\{cx : \mathcal{A}x \leq b, x \in X\}$, suppose that we add nonnegative "slack" variables s to model the problem in the following equivalent equality form : $\min\{cx : \mathcal{A}x + s = b, x \in X \text{ and } s \geq 0\}$.

(a) State the Lagrangian multiplier problem for the equality formulation.

(b) Show that if some $\mu_i < 0$, then $L(\mu) = -\infty$. Further, show that if some $\mu_i > 0$, then in the optimal solution of the Lagrangian subproblem $L(\mu)$, the slack variable $s_i = 0$.

(c) Conclude from part (b) that the Lagrangian multiplier problem of the inequality constrained problem is $\max_{\mu \geq 0} L(\mu)$ with $L(\mu) = \min\{cx + \mu(\mathcal{A}x - b) : x \in X\}$.

16.2. Consider the problem

$$\text{Minimize} \quad -2x - 3y$$

subject to

$$x + 4y \leq 5,$$

$$x, y \in \{0, 1\},$$

and the corresponding relaxed problem

$$\text{Minimize} \quad -2x - 3y + (x + 4y - 5)$$

subject to

$$x, y \in \{0, 1\}.$$

Show that $x = 1, y = 0$ solves the relaxed problem, is feasible for the original problem, and yet does not solve the original problem. (Reconcile this example with Property 16.4.)

16.3. Lagrangian relaxation applied to linear programs. Suppose that we apply Lagrangian relaxation to the linear program $\mathcal{P}$ defined as $\min\{cx : \mathcal{A}x = b, x \geq 0\}$ by relaxing the equality constraints $\mathcal{A}x = b$. The Lagrangian function is $L(\mu) = \min_{x \geq 0} \{cx - \mu(\mathcal{A}x - b)\} = \min_{x \geq 0} \{(c - \mu\mathcal{A})x + \mu b\}$. (Since the constraints $\mathcal{A}x = b$ are equalities, the Lagrange multipliers μ are unconstrained in sign. For the purpose of this exercise, we have chosen a different sign convention than usual, that is, used $-\mu$ in place of μ.) Now, consider the Lagrangian multiplier problem $\max_{\mu} L(\mu)$.

(a) Suppose we choose a value of μ so that for some j, $(c - \mu\mathcal{A})_j < 0$. Show that $L(\mu) = -\infty$.

(b) Suppose we choose a value of μ so that for some j, $(c - \mu\mathcal{A})_j > 0$. Show that in the optimal solution of the Lagrangian subproblem, $x_j = 0$.

(c) Conclude from parts (a) and (b) that the Lagrangian multiplier problem is equiv-

alent to the linear programming dual of $\mathcal{P}$, that is, the problem $\max \mu b$, subject to $\mu \mathcal{A} \leq c$.

16.4. Oscillation in Lagrangian relaxation. Suppose that we apply Lagrangian relaxation to the constrained shortest path example shown in Figure 16.1 with the time constraint of $T = 14$, starting with value $\mu^0 = 0$ for the Lagrange multiplier μ. Show that if we choose the step size $\theta_k = 1$ at each iteration, the subgradient algorithm $\mu^{k+1} = \mu^k + \theta_k(\mathcal{A}x^k - b)$ oscillates between the values $\mu = 0$ and $\mu = 4$ and the Lagrangian subproblem solutions alternate between the paths 1–2–4–6 and 1–3–2–5–6.

16.5. In Section 16.4 we showed that when $T = 14$, our constrained shortest path example had an optimal objective function value $z^* = 13$ while the Lagrange multiplier problem had a value $L^* = 7$. Show that L^* equals the optimal objective function value of the linear programming relaxation of the problem. Interpret the solution of the linear program as the convex hull of shortest path solutions. That is, find a set of paths whose convex combination satisfies the timing constraint and whose weighted (i.e., convex combination) cost equals L^*.

16.6. Suppose that X is a finite set and that when we solve the Lagrangian multiplier problem corresponding to the optimization problem $\min\{cx : \mathcal{A}x = b, x \in X\}$ for *any* value of c, we find that the problem has no duality gap, that is, if x^* solves the given optimization problem and μ^* is an optimal solution to the Lagrangian multiplier problem, then $cx^* = L(\mu^*)$. Show that the polyhedron $\{x : \mathcal{A}x = b \text{ and } x \in \mathcal{H}(X)\}$ has integer extreme points. (*Hint*: Use the results given in the proofs of Theorems 16.9 and 16.10.)

16.7. Lagrangian relaxation interpretation of successive shortest paths. Recall from Section 9.7 that each intermediate stage of the successive shortest path algorithm for solving the minimum cost flow problem maintains a pseudoflow x satisfying the flow bound constraints and a vector π of node potentials satisfying the conditions $c_{ij}^{\pi} = c_{ij} - \pi(i) + \pi(j) \geq 0$ for all arcs $(i, j) \in G(x)$.

(a) Show that the pseudoflow x is optimal for the problem obtained by relaxing the mass balance constraints and replacing the objective function cx with the Lagrangian function

$$\text{Minimize} \quad \sum_{(i,j)\in A} (c_{ij} - \pi(i) + \pi(j))x_{ij}.$$

(b) Interpret the successive shortest path algorithm as a method that proceeds by adjusting the Lagrangian multipliers. At each stage the method adjusts the multipliers π so that (1) the current pseudoflow x is optimal for the Lagrangian subproblem, and (2) some alternate optimal pseudoflow x' for the Lagrangian relaxation is "less infeasible" than x. Finally, when the optimal pseudoflow becomes a flow, we obtain an optimal solution of the Lagrangian subproblem that is also feasible for the original problem; therefore, it must be an optimal solution of the original problem.

16.8. Generalized assignment problem (Ross and Soland [1975]). The generalized assignment problem is the optimization model

$$\text{Minimize} \quad \sum_{i\in I} \sum_{j\in J} c_{ij}x_{ij} \tag{16.10a}$$

subject to

$$\sum_{j\in J} x_{ij} = 1 \qquad \text{for all } i \in I, \tag{16.10b}$$

$$\sum_{i\in I} a_{ij}x_{ij} \leq d_j \qquad \text{for all } j \in J, \tag{16.10c}$$

$$x_{ij} \geq 0 \text{ and integer} \qquad \text{for all } (i, j) \in A. \tag{16.10d}$$

In this problem we wish to assign $|I|$ "objects" to $|J|$ "boxes." The variable $x_{ij} = 1$ if we assign object i to box j and $x_{ij} = 0$ otherwise. We wish to assign each

object to exactly 1 box; if assigned to box j, object i consumes a_{ij} units of a given "resource" in that box. The total amount of resource available in the jth box is d_j. This generic model arises in a variety of problem contexts. For example, in machine scheduling, the objects are jobs, the boxes are machines; a_{ij} is the processing time for job i on machine j and d_j is the total amount of time available on machine j.

(a) Outline the steps required for solving the Lagrangian subproblem obtained by (1) relaxing the constraint (16.9b), and (2) by relaxing the constraint (16.10c).

(b) Compare the lower bounds obtained by the two relaxations suggested in part (a). Which provides the sharper lower bound? Why? (*Hint*: Use Theorems 16.9 and 16.10.)

(c) Compare the optimal objective function value of the Lagrangian multiplier problem for each relaxation suggested in part (a) with the bound obtained by the linear programming relaxation of the generalized assignment model.

16.9. Facility location (Erlenkotter [1978]). Consider the following facility location model:

$$\text{Minimize} \quad \sum_{i \in I} \sum_{j \in J} c_{ij} x_{ij} + \sum_{j \in J} F_j y_j \tag{16.11a}$$

subject to

$$\sum_{j \in J} x_{ij} = 1 \qquad \text{for all } i = 1, 2, \ldots, I, \tag{16.11b}$$

$$\sum_{i \in I} d_i x_{ij} \leq K_j y_j \qquad \text{for all } j = 1, 2, \ldots, J, \tag{16.11c}$$

$$0 \leq x_{ij} \leq 1 \qquad \text{for all } i \in I \text{ and } j \in J, \tag{16.11d}$$

$$y_j = 0 \text{ or } 1 \qquad \text{for all } j \in J. \tag{16.11e}$$

In this model, I denotes a set of customers and J denotes a set of potential facility (e.g., warehouse) locations used to supply to the customers. The zero–one variable y_j indicates whether or not we choose to locate a facility at location j and x_{ij} is the fraction of the demand of customer i that we satisfy from facility j. The constant d_i is the demand of customer i. The cost coefficient c_{ij} is the cost (e.g., the transportation cost) of satisfying all of the ith customer's demand from facility j, and the cost coefficient F_j is the fixed cost of opening (e.g., leasing) a facility of size K_j at location j. The constraints (16.11b) state that we need to satisfy all of the demand for each customer, and the constraints (16.11c) state that (1) we cannot meet any of the demand of any customer if we do not locate a facility at location j (i.e., $x_{ij} = 0$ if $y_j = 0$), and (2) if we do locate a facility at location j (i.e., $y_j = 1$), the total demand met by the facility cannot exceed the facility's capacity K_j.

(a) Show how you would solve the Lagrangian subproblem obtained by relaxing the constraints (16.11b). (*Hint*: Note that the Lagrangian subproblem decomposes into a separate subproblem for each location.)

(b) Show next how you would solve the Lagrangian subproblem if we relax the constraints (16.11c). (*Hint*: Note that the Lagrangian subproblem decomposes into a separate subproblem for each customer.)

(c) Show that if $|I| = |J| = 1$, $K_1 = 10$, $d_1 = 5$, and $c_{11} = 0$, the relaxation suggested in part (a) gives a sharper lower bound than the relaxation in part (b). Next prove the general result that the relaxation in part (a) gives at least as good a bound as given by the relaxation in part (b).

16.10. Modified facility location. Suppose that in the model considered in Exercise 16.9, we impose the additional constraint that the demand for each customer should be "sole sourced"; that is, each variable x_{ij} has value zero or 1.

(a) Show how to use the solution of a single knapsack problem for each facility j to solve the Lagrangian relaxation obtained by relaxing the constraints (16.11b).

(b) Show that the bound obtained from the Lagrangian multiplier problem by relaxing

the constraints (16.11b) is always at least as strong as the bound obtained by relaxing the constraints (16.11c).

16.11 Tightening the facility location relaxation. Suppose that we add the redundant constraints $x_{ij} \leq \min\{y_j, K_j\}$ to the facility location model described in Exercise 16.9 and then we apply Lagrangian relaxation by relaxing the constraints (16.11b) or (16.11c).

(a) Show that the bound obtained from the Lagrangian multiplier problem is always as strong or stronger than the bound obtained by relaxing the corresponding constraints in the original model without the additional constraints $x_{ij} \leq \min\{y_j, K_j\}$.

(b) How would you solve the Lagrangian subproblem with the added constraints $x_{ij} \leq \min\{y_j, K_j\}$?

(c) How would your answers to parts (a) and (b) change if we considered the sole-sourcing-facility location model described in Exercise 16.10?

16.12. Local access capacity expansion (Balakrishnan, Magnanti, and Wong [1991]). The lowest level of national telephone networks are trees that connect individual customers to the rest of the national network through special nodes known as *switching centers*, which route telephone calls to their final destination. Each local access network (tree) T has its own switching center. As demand for service increases, telephone companies have two basic options for increasing the capacity of a local access network: (1) they can install more copper cables on the arcs of the networks; or (2) they can install devices, called *multiplexers* (or *concentrators*), at the nodes. The multiplexers compress calls so that they use less downstream cable capacity. We assume that once a call reaches a multiplexer, it requires negligible cable capacity to send it to the switching center. Every call must be routed through the tree T either to the switching center or to one of the multiplexers. Suppose that the existing capacity of arc (i, j) is u_{ij} and increasing the capacity by y_{ij} units incurs an arc-dependent cost $c_{ij}y_{ij}$. Let d_i denote the numbers of calls originating at node i that must be routed to the switching center or to a multiplexer. Each multiplexer has two associated costs: (1) a fixed cost F, and (2) a variable throughput cost α incurred for each unit of call compressed by that multiplexer. The optimization problem is to meet the demand for service by incurring minimum total cost.

(a) Let z_i be a zero–one variable indicating whether or not we place a multiplexer at node i. Further, let x_{ij} be a zero–one variable indicating whether or not we assign node i to the multiplexer j. In the local access network T, for any pair $[i, j]$ of nodes, we let P_{ij} denote the unique path between these two nodes, and for any arc (k, l) in T, we let Q_{kl} be the set of all node pairs $[i, j]$ from which P_{ij} contains the arc (k, l). We assume that node 1 is the switching center. Let node S denote the remaining nodes in the network. Using this notation, give an integer programming formulation of the local access network design problem.

(b) Suggest two relaxations of the formulation in part (a) that produce a relaxed problem with a structure that we have treated in this book.

16.13. Contiguous local access capacity expansion problem (Balakrishnan, Magnanti, and Wong [1991]). In most practical settings of the local access capacity expansion problems, the set of nodes assigned to the switching center or to any multiplexer must be contiguous. That is, if we assign node i to a multiplexer at node j and node k lies on the path in T from node i to node j, we must also assign node k to the multiplexer at node j. Therefore, the final configuration of the local access network will be a subdivision of the tree T into subtrees, with each subtree containing either the switching center or one multiplexer and the nodes it serves.

(a) Show that we can incorporate the contiguity condition in the formulation of Exercise 16.12 by adding the following constraints for every pair $[i, j]$ of nodes: if the path P_{ij} contains node k, then $x_{ik} \geq x_{ij}$.

(b) Consider the integer programming formulation of the contiguous local access capacity expansion problem from part (a). Suppose that we relax the capacity constraints imposed on the arcs. Show that we can solve the Lagrangian subproblem in polynomial time using a dynamic programming technique.

16.14. Design of telecommunication networks (Leung, Magnanti, and Singhal [1990] and Magnanti, Mirchandani, and Vachani [1991]). In designing telecommunications networks, we would like to install sufficient capacity to carry required traffic (telephone calls, data transmissions) simultaneously between various source–sink locations. Suppose that (s^k, t^k) for $1 \le k \le K$ denote K pairs of source–sink locations, and r^k denotes the number of messages sent from the source s^k to the sink t^k. We can install either of two different types of facilities on each link of the transmission network, so-called T0 lines and T1 lines. Each T0 line can carry 1 unit of message and each T1 line can carry 24 units of messages; installing a T0 line on arc (i, j) incurs a cost of a_{ij} and installing a T1 line on arc (i, j) incurs a cost of b_{ij}. Once we have installed the lines, we incur no additional costs in sending flow on them. This problem arises in practice because companies with large telecommunication requirements might be able to lease lines more cost-effectively than paying public tariffs. The same type of problem arises in trucking of freight; in this setting, the facilities to be "installed" on any arc are the trucks of a particular type (e.g., 36-foot trailers or 48-foot trailers) to be dispatched on that arc.

(a) Show how to formulate this telecommunication network design problem with two types of constraints: (1) a set of network flow constraints modeling the required flow between every pair of source–sink locations, and (2) capacity constraints restricting the total flow on each arc to be no more than the capacity that we install on that arc. (*Hint*: Use the following integer decision variables: (1) y_{ij} : the number of T0 lines for arc (i, j), (2) z_{ij} : the number of T1 lines for arc (i, j), and (3) the number of messages x_{ij}^k sent from the source s^k to the sink t^k that pass through the arc (i, j).)

(b) How would you solve the linear programming relaxation of this model? (*Hint*: Consider two cases: when $24\ a_{ij} < b_{ij}$ and when $24\ a_{ij} \ge b_{ij}$.)

(c) Show how to solve the Lagrangian subproblem obtained by relaxing constraints of type 1 in the model formulation. (*Hint*: Consider the two cases as in part (b).)

(d) Show how to solve the Lagrangian subproblem obtained by relaxing constraints of type 2 in the model formulation.

16.15. Steiner tree problem. The Steiner tree problem is an $\mathcal{NP}$-hard variant of the minimum spanning tree problem. In this problem we are given a subset $S \subseteq N$ of nodes, called *customer nodes*, and we wish to determine a minimum cost tree (not necessarily a spanning tree) that must contain all the nodes in S and, optionally, some nodes in $N - S$. This problem arises in many application settings, such as the design of rural road networks, pipeline networks, or communication networks. Formulate this problem as a special case of the network design problem discussed in Application 16.4 and show how to apply Lagrangian relaxation to the resulting formulation. (*Hint*: Designate any customer node as a source node and send 1 unit of flow to every other customer node.)

16.16. In this exercise we show how to formulate a directed traveling salesman problem as a network design problem. Consider a network design problem with the following data: (1) unit commodity flow requirements between every pair of nodes, (2) the cost of flow on every arc is zero, and (3) the fixed cost of the arc (i, j) is $c_{ij} + M$ for some sufficiently large number M. Show that the optimal network will be an optimal traveling salesman tour with c_{ij} as arc lengths. (*Hint*: The optimal network must be strongly connected and must contain the fewest possible number of arcs.)

16.17. Uncapacitated undirected network design problem. In the formulation of the directed uncapacitated network design problem in Application 16.4, the zero–one vector y_{ij} indicated whether we would include the directed arc (i, j) in the underlying network. Suppose, instead, that the arcs are undirected, so if we introduce arc (i, j) in the network, we can send flow in either direction on the arc.

(a) How would you formulate this problem and apply Lagrangian relaxation to obtain lower bounds?

(b) Show that in the uncapacitated undirected network design problem, if all flow costs c_{ij}^k are zero, all the fixed costs f_{ij} are nonnegative, and the problem has a commodity for each pair of nodes, the problem reduces to the minimum spanning tree problem.

16.18. (a) Show that if the uncapacitated network design problem has a single commodity (i.e., $K = 1$), we can solve the problem by solving a single shortest path problem.

(b) Show how to formulate the production planning problems that we described in Application 16.7 as capacitated or uncapacitated network design problems.

16.19. (a) Suppose that we relax the mass balance constraint $\mathcal{N}x = b$ in the formulation (16.3) of the traveling salesman problem described in Application 16.2. Show how to solve the Lagrangian subproblem as an assignment problem. (*Hint*: Show that some optimal solution of the Lagrangian subproblem satisfies the conditions $x_{ij} = 0$ or $x_{ij} = (n - 1)y_{ij}$ for each arc $(i, j) \in A$. Use this fact to eliminate the variables x_{ij} from the subproblem.)

(b) Suppose that we relax the assignment constraints (16.3b) and (16.3c) in the formulation (16.3) of the traveling salesman problem. Show how to solve the Lagrangian subproblem as an uncapacitated network design problem.

(c) What is the relationship between the optimal solutions of the Lagrangian multiplier problems obtained by the relaxations considered in parts (a) and (b), the relaxation described in the text (obtained by relaxing the forcing constraints $x_{ij} \leq (n - 1)y_{ij}$), and the optimal objective function value of the linear programming relaxation of the problem?

16.20. Assignment-based formulation of the traveling salesman problem

(a) Consider the integer program (16.3) with the constraints (16.3d) and (16.3e) replaced by the subtour breaking constraints (16.4h). Show that the resulting model is an integer programming formulation of the traveling salesman problem.

(b) Show that the solution of the Lagrangian subproblem formed by relaxing the subtour breaking constraints will be a set of directed cycles satisfying the property that each node is contained in exactly one cycle. Describe a heuristic method for modifying the Lagrangian subproblem solution so that it becomes a feasible traveling salesman tour.

16.21. Undirected traveling salesman problem. In the undirected (or symmetric) traveling salesman problem, we can traverse any arc (i, j) in either direction at the same cost c_{ij}. Let y_{ij} indicate whether or not we include arc (i, j) in a feasible tour.

(a) Give a formulation of this problem as an integer program containing three sets of constraints: (1) degree 2 constraints, indicating that each node should have degree at most 2 in any feasible tour; (2) subtour breaking constraints on the nodes $2, 3, \ldots, n$; and (3) a cardinality constraint indicating that the tour contains exactly n arcs. (*Hint*: Modify the assignment based formulation of the directed traveling salesman problem described in Exercise 16.21.)

(b) Show how to apply Lagrangian relaxation in two ways: (1) by relaxing the degree 2 constraints; and (2) by relaxing the subtour breaking constraints and the cardinality constraint. In case 1 show how to solve the Lagrangian subproblem as a 1-tree (see Exercise 13.37). In case 2 show how to solve the Lagrangian subproblem as a circulation problem. (*Hint*: In case 2, first show that any network with n arcs and degree at most 2 on each node, must have a degree of exactly 2 at each node.)

16.22. Multicommodity flow-based formulation of the traveling salesman problem. In Application 16.2 we examined a single-commodity flow-based formulation of the traveling salesman problem with $n - 1$ units available at a source node (which we arbitrarily took to be node 1) and 1 unit of demand required at each other node. Suppose that, instead, we formulated a multicommodity flow model with $2(n - 1)$ commodities, with two commodities k defined for each node $k \neq 1$, an "outgoing" commodity and an "incoming" commodity. The incoming commodity for node k has 1 unit of supply at node 1 and 1 unit of demand at node k, and the outgoing commodity for node k has

1 unit of supply at node k and 1 unit of demand at node 1 (i.e., we wish to send 1 unit from node 1 to node k and 1 unit from node k to node 1). We can state this formulation of the traveling salesman problem as the following integer program:

$$\text{Minimize} \quad \sum_{(i,j)\in A} c_{ij}y_{ij},$$

subject to

$$\sum_{1\le j\le n} y_{ij} = 1 \qquad \text{for all } i = 1, 2, \ldots, n,$$

$$\sum_{1\le i\le n} y_{ij} = 1 \qquad \text{for all } j = 1, 2, \ldots, n,$$

$$Nx^k = b^k \qquad \text{for all } k = 2, \ldots, n,$$

$$Nz^k = d^k \qquad \text{for all } k = 2, \ldots, n,$$

$$x_{ij}^k \le y_{ij} \text{ and } z_{ij}^k \le y_{ij} \qquad \text{for all } (i, j) \text{ and all } k,$$

$$y_{ij} \text{ and } x_{ij}^k = 0 \text{ or } 1 \qquad \text{for all } (i, j) \text{ and all } k.$$

Note that the supply/demand vectors b^k and d^k in this formulation have a special form: $d^k = -b^k$ and b_i^k is 1 if $i = 1$, is -1 if $i = k$, and is 0 if $i \ne 1$ and $i \ne k$.

(a) Suppose that we apply Lagrangian relaxation to the multicommodity flow-based model by relaxing the forcing constraints [i.e., the constraints $x_{ij}^k \le y_{ij}$ and $z_{ij}^k \le y_{ij}$, for all (i, j) and all k]. How would you solve the Lagrangian subproblem?

(b) Show that the lower bound L^* determined by the Lagrangian multiplier problem for the Lagrangian relaxation in part (a) is always as strong or stronger than the lower bound determined by relaxing the forcing constraints in the single-commodity flow-based formulation. (*Hint*: Compare the set of feasible solutions of both problems.)

(c) What is the relationship between the optimal objective function values of the linear programming relaxations of the single and multicommodity flow-based formulations? (*Hint*: Same as that in part (b).)

16.23. Alternate formulations of the traveling salesman problem (Wong [1980]). In this chapter we have considered three different formulations of the traveling salesman problem: (1) a single-commodity flow-based formulation in Application 16.2, (2) an assignment-based formulation discussed in Exercise 16.20, and (3) a multicommodity flow-based formulation in Exercise 16.22, where we showed that from the perspective of linear programming or Lagrangian relaxations, the multicommodity flow-based formulation is stronger than the single commodity flow-based formulation.

(a) Show that we can replace the subtour breaking constraints in the assignment-based formulation, or in its linear programming relaxation, by the constraints $\sum_{i\in S} \sum_{j\in N-S} y_{ij} \ge 1$ for all sets S of nodes satisfying the cardinality condition $1 \le |S| \le n - 1$, and in both cases obtain an equivalent model (i.e., one with the same feasible solutions).

(b) Using the max-flow min-cut theorem and part (a), show that the linear programming relaxation of the assignment-based formulation of the traveling salesman problem and the linear programming relaxation of the multicommodity flow-based formulation are equivalent in the sense that y is feasible in the linear programming relaxation of the assignment-based formulations if and only if for some flow vector x, (x, y) is feasible in the multicommodity flow-based formulation. (Note that the number of subtour breaking constraints in the assignment-based formulation is exponential in n. The number of constraints in the multicommodity flow-based formulation is polynomial in n, so this formulation is a so-called *compact* formulation.)

16.24. Consider the undirected traveling salesman problem shown in Figure 16.11.

(a) What is the optimal tour length for this problem?

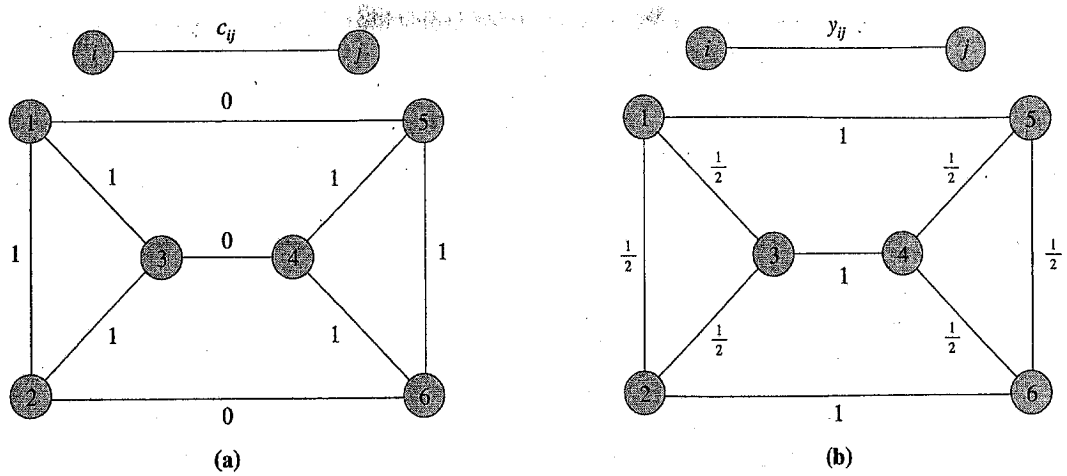


Figure 16.11 Traveling salesman problem: (a) network data; (b) solution to the linear programming relaxation.

(b) Show that the arc weights shown in Figure 16.11(b) solve the linear programming relaxation of the formulation developed in Exercise 16.21. Interpret this solution as the convex hull of 1-tree solutions to the Lagrangian subproblem that we obtain by relaxing the degree two constraints in this formulation. That is, show how to represent this solution as a convex combination of 1-tree solutions.

(c) Show the network corresponding to the equivalent directed traveling salesman problem and specify the optimal solution to the linear programming relaxation of the assignment-based formulation and both the single and multicommodity flow-based formulations.

(d) Interpret the solution to each linear programming relaxation as the convex hull of solutions to Lagrangian subproblems.

16.25. *K*-traveling salesman problem. Suppose that we wish to find a set of K arc-disjoint directed cycles in a directed graph satisfying the property that node 1 is contained in exactly K cycles and every other node is contained in exactly one cycle. In this model each arc has an associated cost c_{ij} and we wish to find a feasible solution with the smallest possible sum of arc costs. We refer to this problem as the K-traveling salesman problem since it corresponds to a situation in which K salesmen, all domiciled at the same node 1, need to visit all the other nodes of a graph.

(a) Formulate this problem as an optimization model and show how to apply Lagrangian relaxation to the formulation. (*Hint*: Modify the single-commodity flow-based formulation given in Application 16.2 or the assignment-based formulation given in Exercise 16.20.)

(b) By forming K copies of node 1 and assigning a large cost with all of the arcs joining the copies of node 1, show how to formulate the problem as an equivalent (single) traveling salesman problem.

16.26. Consider the K-traveling salesman problem described in Exercise 16.25. Show how to formulate this problem as a special case of the vehicle routing problem described in Application 16.3. The resulting formulation will be considerably simpler than the general vehicle routing problem. (*Hint*: First show that we can eliminate the constraints (16.4g). Next show how to use the constraints (16.4b) to eliminate the variables x_{ij}^k.)

16.27. Vehicle routing with nonhomogeneous fleets and with time restrictions. This exercise studies a generalization of the vehicle routing problem discussed in Application 16.3. Show how to formulate a vehicle routing problem with each of the following problem ingredients: (1) each vehicle k in a fleet of K vehicles can have different capacity u^k,

or (2) each vehicle must make its deliveries within T hours, given that it takes t_{ij} hours to traverse any arc (i, j).

16.28. Suppose that we add the redundant constraint $\sum_{(i,j)\in A} y_{ij} = n + K$ to the formulation (16.4) of the vehicle routing problem. Consider the additional set of constraints $\sum_{j\in S} y_{1j} + \sum_{i\in S} \sum_{j\in S} y_{ij} \leq |S|$ for all subsets S of $\{2, 3, \ldots, n\}$.

(a) Are these constraints valid?

(b) Are these constraints implied by the other constraints in the integer programming formulation of the problem? Are they implied by the other constraints in the linear programming relaxation of the problem?

(c) Suppose that we add the additional constraints to the formulation of the vehicle routing problem. Show that by relaxing the capacity constraints (16.4g) and the assignment constraints (16.2c) and (16.2d) for nodes 2, 3, . . . , n, the resulting Lagrangian subproblem decomposes into two subproblems: (i) a degree-constrained minimum spanning tree problem with degree K imposed on node 1, and (ii) a problem of choosing the K cheapest (with respect to the Lagrangian subproblem coefficients) arcs of the form y_{j1}. (*Hint*: First eliminate the variables x_{ij}^k and then note that the Lagrangian subproblem decomposes into two subproblems, one containing the variables y_{j1} and one containing all the other variables.)

16.29. Solve the degree-constrained minimum spanning tree problem shown in Figure 16.12 assuming that the degree of node 1 must be 8. Solve it if the degree of node 1 must be 5.

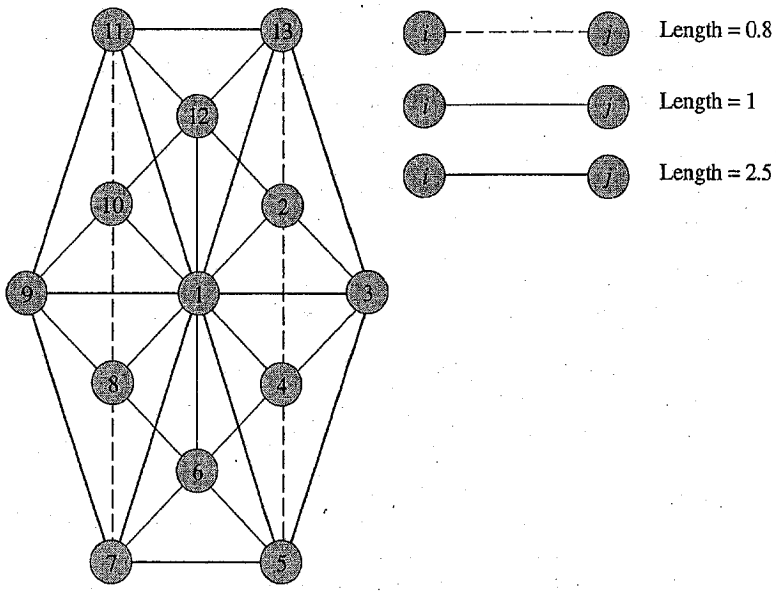


Figure 16.12 Constrained minimum spanning tree problem.

16.30. Suppose that X is a finite set and that when we solve the Lagrangian multiplier problem corresponding to the optimization problem $\min\{cx : \mathcal{A}x = b, x \in X\}$ for *any* value of c, we find that the problem has no relaxation gap; that is, if x^* solves the given optimization problem and μ^* is an optimal solution to the Lagrangian multiplier problem, then $cx^* = L(\mu^*)$. Show that the polyhedron $\{x \in \mathcal{H}(X) : \mathcal{A}x = b\}$ has integer extreme points. (*Hint*: Use the equivalence between convexification and Lagrangian relaxation (Theorem 16.10) and the fact that every extreme point to a polyhedron $\mathcal{P}$ is the *unique* optimal solution to the linear program $\min\{cx : x \in \mathcal{P}\}$ for *some* choice of the objective coefficients c.)

16.31. Let X denote the set of incidence vectors of spanning trees of a given network.

(a) Using Exercise 16.30 and the results in Section 16.4, show that for any value of k, the polyhedron $\{x : x \in \mathcal{H}(X) \text{ and } \sum_{j \neq 1} x_{ij} = k\}$ has integer extreme points. Note that if we view the set of solutions to $x \in \mathcal{H}(X)$ and $\sum_{j \neq 1} x_{1j} = k$ as we vary k as "parallel slices" through the polyhedron $x \in \mathcal{H}(X)$, this result says that extreme points of every slice are integer valued.

(b) For any subset S of nodes, let $A(S) = \{(i, j) \in A : i \in S \text{ and } j \in S\}$. Using the result of part (a) and the development in Section 13.8, show that for any value of k, the following polyhedron has integer extreme points:

$$\sum_{j \neq 1} x_{1j} = k,$$

$$\sum_{(i,j) \in A} x_{ij} = n - 1,$$

$$\sum_{(i,j) \in A(S)} x_{ij} \leq |S| - 1 \qquad \text{for any set } S \text{ of nodes,}$$

$$x_{ij} \geq 0.$$

(*Hint*: In Section 13.8 we showed that without the cardinality constraint $\sum_{j \neq 1} x_{1j} = k$, the extreme points in the polyhedron defined by the remaining constraints are incident vectors of spanning tree solutions (and so are integer valued).)

16.32. Suppose that we wish to find a minimum spanning tree of an undirected graph G satisfying the additional conditions that the degree of node 1 is k and the degree of node n is l. Suggest a Lagrangian relaxation bounding procedure for this problem. (*Hint*: Consider relaxing just one of the two degree constraints.)

16.33. Capacitated minimum spanning tree problem (Gavish [1985]). In some applications of the minimum spanning tree problem, we want to construct a capacitated tree T rooted at a specially designated node, say node 1. In this problem we wish to identify a minimum cost spanning tree subject to the additional condition that no subtree of T formed by eliminating all the arcs incident to node 1 contains more than a prescribed number u of nodes. This model arises, for example, in computer networking when node 1 is a central processor and for reasons of reliability we wish to limit the number of nodes (terminals) attached to this node through any of its ports (incident arcs). Let y_{ij} be a zero–one variable, indicating whether or not we include arc (i, j) in the optimal capacitated tree.

(a) Explain how the capacitated minimum spanning tree problem differs from the degree-constrained minimum spanning tree problem.

(b) By introducing additional constraints in the integer programming formulation (13.2) of the minimum spanning tree problem, obtain an integer programming formulation of the capacitated minimum spanning tree problem.

(c) Suggest a Lagrangian-based method for obtaining a lower bound on the optimal solution by solving a sequence of minimum spanning tree problems.

16.34. Identical customer vehicle routing problem. In the identical customer vehicle routing problem, each customer has the same demand. Formulate this problem as a capacitated minimum spanning tree problem with additional constraints. Show how to obtain bounds on the objective values by applying Lagrangian relaxation to this problem using the capacitated minimum spanning tree problem as a subproblem.

16.35. Note that every solution to a vehicle routing problem is a degree constrained minimum spanning tree (with degree K for node 1) together with K additional arcs incident to node 1 as well as another set of constraints modeling vehicle capacities. Use this observation to give a formulation of the vehicle routing problem and an associated Lagrangian relaxation that contains the degree-constrained minimum spanning tree problem as a subproblem.

16.36. Lagrangian decomposition (Guignard and Kim [1987a,b]). Consider the optimization problem P1 defined as $\min\{cx : \mathcal{A}x = b, \mathcal{D}x = d, x \geq 0 \text{ and integer}\}$. Suppose that by using a variable splitting technique described in Application 16.5, we restate this

problem in the following equivalent form P2: $\min\{\frac{1}{2}cx + \frac{1}{2}cy : \mathcal{A}x = b, \mathcal{D}y = d, x - y = 0, x, y \geq 0 \text{ and integer}\}$. We might form three different Lagrangian relaxations for this problem, one by relaxing the constraint $\mathcal{A}x = b$ in P2, one by relaxing the constraint $\mathcal{D}x = d$ in P2, and one by relaxing the constraint $x - y$ in P2. Let L^1, L^2, and L^3 denote the optimal values of the Lagrangian multiplier problems for each of these relaxations. The approach via problem L^3 is known as a *Lagrangian decomposition* since it permits us to decompose the problem into two separate subproblems, one corresponding to each set of equality constraints. Using Theorems 16.8 and Theorem 16.9, show that $L^3 \geq L^1$ and $L^3 \geq L^2$. (*Hint*: Let H^1 and H^2, respectively, denote the convex hulls of the sets $\{\mathcal{D}x = d, x \geq 0 \text{ and integer}\}$ and $\{\mathcal{A}x = b, x \geq 0 \text{ and integer}\}$. Consider the sets $H^1 \cap \{x : \mathcal{A}x = b\}$, $H^2 \cap \{x : \mathcal{D}x = d\}$, and $H^1 \cap H^2$, and consider the minimization of the objective function cx over each of these sets. Which of these problems has the smallest objective function value? What is the relationship between these optimal objective function values and the values L^1, L^2, and L^3?)

16.37. Example of Lagrangian decomposition. Suppose that we apply the Lagrangian decomposition procedure to the following integer programming example:

$$\text{Minimize} \quad -2x_1 - 3x_2$$

subject to

$$9x_1 + 10x_2 \leq 63,$$

$$4x_1 + 9x_2 \leq 36,$$

$$x_1, x_2 \geq 0 \text{ and integer.}$$

In this case, the reformulated problem is:

$$\text{Minimize} \quad -x_1 - \tfrac{3}{2}x_2 - y_1 - \tfrac{3}{2}y_2$$

subject to

$$9x_1 + 10x_2 \leq 63,$$

$$4y_1 + 9y_2 \leq 36,$$

$$x_1 - y_1 = 0,$$

$$x_2 - y_2 = 0,$$

$$x_1, x_2, y_1, y_2 \geq 0 \text{ and integer.}$$

Give a geometrical interpretation of the Lagrangian relaxation obtained by relaxing each of the following constraints: (1) $4x_1 + 9x_2 \leq 36$; (2) $4y_1 + 9y_2 \leq 36$; and (3) $x_1 - y_1 = 0$ and $x_2 - y_2 = 0$. From these geometrical considerations, interpret the fact that in the notation of Exercise 16.36, $L^3 \geq L^1$ and that $L^3 \geq L^2$.

17

MULTICOMMODITY FLOWS

You cannot conceive the many without the one.
—Plato

Chapter Outline

17.1 Introduction
17.2 Applications
17.3 Optimality Conditions
17.4 Lagrangian Relaxation
17.5 Column Generation Approach
17.6 Dantzig–Wolfe Decomposition
17.7 Resource-Directive Decomposition
17.8 Basis Partitioning
17.9 Summary

17.1 INTRODUCTION

Throughout most of our discussion to this point, we have considered network models composed of a single commodity—one that we wish to send from its source(s) to its sink(s) in some optimal fashion, for example, along a shortest path or via a minimum cost flow. In many application contexts, several physical commodities, vehicles, or messages, each governed by their own network flow constraints, share the same network. For example, in telecommunications applications, telephone calls between specific node pairs in an underlying telephone network each define a separate commodity. If the commodities do not interact in any way, then to solve problems with several commodities, we would solve each single-commodity problem separately using the techniques that we have developed in prior chapters. In other situations, however, because the commodities do share common facilities, the individual single commodity problems are not independent, so to find an optimal flow, we need to solve the problems in concert with each other. In this chapter we study one such model, known as the *multicommodity flow problem*, in which the individual commodities share common arc capacities. That is, each arc has a capacity u_{ij} that restricts the total flow of all commodities on that arc.

Let x_{ij}^k denote the flow of commodity k on arc (i, j), and let x^k and c^k denote the flow vector and per unit cost vector for commodity k. Using this notation we can formulate the multicommodity flow problem as follows:

$$\text{Minimize} \quad \sum_{1 \le k \le K} c^k x^k \tag{17.1a}$$

subject to

$$\sum_{1 \le k \le K} x_{ij}^k \le u_{ij} \qquad \text{for all } (i, j) \in A, \tag{17.1b}$$

$$\mathcal{N}x^k = b^k \qquad \text{for } k = 1, 2, \ldots, K, \tag{17.1c}$$

$$0 \le x_{ij}^k \le u_{ij}^k \qquad \text{for all } (i, j) \in A \text{ and all } k = 1, 2, \ldots, K. \tag{17.1d}$$

This formulation has a collection of K ordinary mass balance constraints (17.1c), modeling the flow of each commodity $k = 1, 2, \ldots, K$. The "bundle" constraints (17.1b) tie together the commodities by restricting the total flow $\sum_{1 \le k \le K} x_{ij}^k$ of all the commodities on each arc (i, j) to at most u_{ij}. Note that we also impose individual flow bounds u_{ij}^k on the flow of commodity k on arc (i, j). Many applications do not impose these bounds, so for these applications we set each bound to $+\infty$. Although we might formulate a variety of alternative multicommodity models with different assumptions, we will refer to this model as the multicommodity flow problem.

At times in our discussion, it will be more convenient to state the bundle constraints (17.1b) as equalities instead of inequalities. In these instances we introduce nonnegative "slack" variables s_{ij} and write the bundle constraints as

$$\sum_{1 \le k \le K} x_{ij}^k + s_{ij} = u_{ij} \qquad \text{for all } (i, j) \in A. \tag{17.1b'}$$

The slack variable s_{ij} for the arc (i, j) measures the unused bundle capacity on that arc.

Assumptions

Note that the model (17.1) imposes capacities on the arcs but not on the nodes. This modeling assumption imposes no loss of generality, since by using the node splitting techniques described in Section 2.4, we can use this formulation to model situations with node capacities as well. Three other features of the model are worth noting.

Homogeneous goods assumption. We are assuming that every unit flow of each commodity uses 1 unit of capacity of each arc. A more general model would permit the unit flow of each commodity k to consume a given amount ρ_{ij}^k of the capacity (or some other resource) associated with each arc (i, j), and replace the bundle constraint with a more general resource availability constraint $\sum_{1 \le k \le K} \rho_{ij}^k x_{ij}^k \le u_{ij}$. With minor modifications, the solution techniques that we will be discussing in this chapter apply to this more general model as well (see Exercise 17.10).

No congestion assumption. We are assuming that we have a hard (i.e., fixed) capacity on each arc and that the cost on each arc is linear in the flow on that arc. In some applications encountered in communication, transportation, and other problem domains, the commodities interact in a more complicated fashion in the sense that as the flow of any commodity increases on an arc, we incur an increasing and nonlinear cost on that arc. This type of model arises frequently, for example, in traffic networks where the objective function is to find the flow pattern of all the commodities that minimizes overall system delay. In this setting, because of queuing

effects, the greater the flow on an arc, the greater is the queuing delay on that arc. For example, a "congestion" model for multicommodity flows might contain the individual flow constraints (17.1b) and (17.1c), no bundle constraint, but a nonlinear objective function of the form

$$\text{Minimize} \quad \sum_{(i,j)\in A} \frac{x_{ij}}{u_{ij} - x_{ij}}.$$

In this model u_{ij} is the "nominal" capacity of the arc (i, j); as the total flow $x_{ij} = \sum_{(i,j)\in A} x_{ij}^k$ on any arc approaches the arc's nominal capacity, the delay approaches $+\infty$ (this form of the objective function is derived from basic results in queuing theory). Practitioners often use this type of model in the context of "performance modeling" to see how overall system delay, or performance, varies as a function of various system designs (e.g., in response to changes in the network topology). In Chapter 14 we show how to solve these nonlinear models for a single commodity as a generalization of the minimum cost flow problem. In Exercise 17.11 we show how to solve a class of nonlinear multicommodity flow problems.

Indivisible goods assumption. The model (17.1) assumes that the flow variables can be fractional. In some applications encountered in practice, this assumption is appropriate; in other contexts, however, the variables must be integer valued. In these instances the model that we are considering might still prove to be useful, since the linear programming model might either be a good approximation of the integer programming model, or we can use the linear programming model as a linear programming relaxation of the integer program and embed it within branch-and-bound or some other type of enumeration approach.

We note that the integrality of solutions is one very important distinguishing feature between single and multicommodity flow problems. As we have seen several times in our previous development, one very nice feature of single-commodity network flow problems is that they always have integer solutions whenever the supply/demand and capacity data are integer valued. Multicommodity flow problems do not satisfy this integrality property. For example, consider a multicommodity maximum flow problem with three nodes 1, 2, 3, and three commodities as shown in Figure 17.1. Each arc has a capacity of 1 unit. We wish to find a solution that maximizes the total flow between the source and sink nodes of all three commodities.

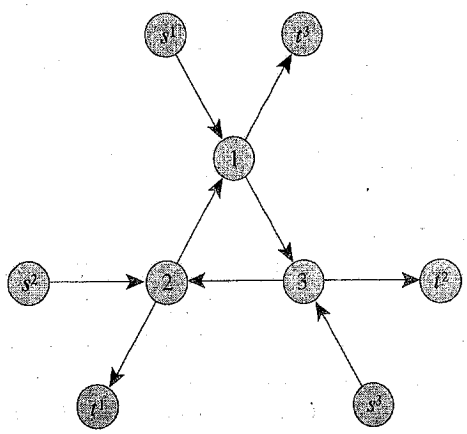


Figure 17.1 Multicommodity maximum flow problem with a fractional solution. Maximum total flow from all sources to all sinks is 1.5 units.

The optimal solution is to send 0.5 units between the source and sink of all three commodities for a total flow of 1.5 units. The optimal integral solution to this problem would send 1 unit between just one of the three commodities for a total flow of 1 unit. In Exercise 17.25 we ask the reader to formulate this multicommodity maximum flow problem as a special case of the multicommodity flow problem stated in (17.1) and so to show that this model does not always have integral solutions even when the problem data are integral.

Solution Approaches

Researchers have developed several approaches for solving the multicommodity flow problem, including:

1. Price-directive decomposition
2. Resource-directive decomposition
3. Partitioning methods

Price-directive decomposition methods place Lagrangian multipliers (or prices) on the bundle constraints and bring them into the objective function so that the resulting problem decomposes into a separate minimum cost flow problem for each commodity k. That is, these methods remove the capacity constraints and instead "charge" each commodity for the use of the capacity of each arc. These methods attempt to find appropriate prices so that some optimal solution to the resulting "pricing problem" or Lagrangian subproblem also solves the overall multicommodity flow problem. Several methods are available for finding appropriate prices. Building on our discussion of Lagrangian relaxation methods in Chapter 16, we describe the application of Lagrangian relaxation to find the correct prices in Section 17.4.

Dantzig–Wolfe decomposition is another approach for finding the correct prices; this method is a general-purpose approach for decomposing problems that have a set of "easy" constraints and also a set of "hard" constraints (that is, constraints that make the problem much more difficult to solve). For multicommodity flow problems, the network flow constraints are the easy constraints and the bundle constraints are the hard constraints. The approach begins, like Lagrangian relaxation, by ignoring or imposing prices on the bundle constraints and solving Lagrangian subproblems with only the single-commodity network flow constraints. The resulting solutions need not satisfy the bundle constraints, and the method uses linear programming to update the prices so that the solutions generated from the subproblems satisfy the bundle constraints. The method iteratively solves two different problems: a Lagrangian subproblem and a price-setting linear program. This method has played an important role in the field of optimization both because the algorithm itself has proven to be very useful, and also because it has stimulated many other approaches to problem decomposition. Moreover, the algorithm and its associated underlying theory have had a significant influence on the field of economics since this type of price decomposition formalizes ideas of transfer pricing and coordination that lie at the heart of planned economies. In Sections 17.5 and 17.6 we describe the use of

Dantzig–Wolfe decomposition, and the related technique of column generation, for solving the multicommodity flow problem.

An alternative way of viewing the multicommodity flow problem is as a capacity allocation problem. All the commodities are competing for the fixed capacity u_{ij} of every arc (i, j) of the network. Any optimal solution to the multicommodity flow problem will prescribe a specific flow on each arc (i, j) for each commodity which is the appropriate capacity to allocate to that commodity. If we started by allocating these capacities to the commodities and then solved the resulting (independent) single-commodity flow problems, we would be able to solve the problem quite easily as a set of independent single-commodity flow problems. Resource-directive methods provide a general solution approach for implementing this idea. They begin by allocating the capacities to the commodities and then use information gleaned from the solution to the resulting single-commodity problems to reallocate the capacities in a way that improves the overall system cost. In Section 17.7 we show how to solve the multicommodity flow problem using this resource-directive approach.

Partitioning methods exploit the fact that the multicommodity flow problem is a specially structured linear program with embedded network flow problems. As we have seen in Chapter 11, to solve any single-commodity flow problem, we can use the network simplex method, which works by generating a sequence of improving spanning tree solutions. In Section 11.11 we showed how to interpret the network simplex method as a special implementation of the simplex method for general linear programs and showed that spanning trees solutions correspond to basic feasible solutions of the minimum cost flow problem. This observation raises the following questions: (1) Can we adopt a similar approach for solving the multicommodity flow problem? (2) Can we somehow use spanning tree solutions for the embedded network flow constraints $\mathcal{N}x^k = b^k$? The partitioning method is a linear programming approach that permits us to answer both of these questions affirmatively. It maintains a linear programming basis that is composed of bases (spanning trees) of the individual single-commodity flow problems as well as additional arcs that are required to "tie" these solutions together to accommodate the bundle constraints. In Section 17.8 we describe the essential features of this approach.

The various solution techniques we describe in this chapter require linear programming background. We refer the reader to Appendix C for a review of this material. Before discussing the solution techniques, we describe several applications of the multicommodity flow problem.

17.2 APPLICATIONS

Multicommodity flow problems arise in a wide variety of application contexts. In this section we consider several instances of one very general type of application as well as a production planning and warehousing example and a vehicle fleet planning example.

Application 17.1 Routing of Multiple Commodities

In many applications of the multicommodity flow problem, we distinguish commodities because they are different physical goods, and/or because they have different points of origin and destination; that is, either (1) several physically distinct

commodities (e.g., different manufactured goods) share a common network, or (2) a single physical good (e.g., messages or products) flows on a network, but the good has multiple points of origin and destination defined by different pairs of nodes in the network that need to send the good to each other. This second type of application arises frequently in problem contexts such as communication systems or distribution/transportation systems. In this section we introduce several application domains of both types.

Communication networks. In a communication network, nodes represent origin and destination stations for messages, and arcs represent transmission lines. Messages between different pairs of nodes define distinct commodities; the supply and demand for each commodity is the number of messages to be sent between the origin and destination nodes of that commodity. Each transmission line has a fixed capacity (in some applications the capacity of each arc is fixed; in others, we might be able to increase the capacity at a certain cost per unit). In this network, the problem of determining the minimum cost routing of messages is a multicommodity flow problem.

Computer networks. In a computer communication network, the nodes represent storage devices, terminals, or computer systems. The supplies and demands correspond to the data transmission rates between the computer, terminals, and storage devices, and the transmission line capacities define the bundle constraints.

Railroad transportation networks. In a rail network, nodes represent yard and junction points, and arcs represent track sections between the yards. The demand is measured by the number of cars (or any other equivalent measure of tonnage) to be loaded on any train. Since the system incurs different costs for different goods, we divide traffic demand into different classes. Each commodity in this network corresponds to a particular class of demand between a particular origin–destination pair. The bundle capacity of each arc is the number of cars that we can load on the trains that are scheduled to be dispatched on that arc (over some period of time). The decision problem in this network is to meet the demands of cars at the minimum possible operating cost.

Distribution networks. In distribution systems planning, we wish to distribute multiple (nonhomogeneous) products from plants to retailers using a fleet of trucks or railcars and using a variety of railheads and warehouses. The products define the commodities of the multicommodity flow problem, and the joint capacities of the plants, warehouses, railyards, and the shipping lanes define the bundle constraints. Note that in this application the nodes (plants, warehouses) as well as the arcs have bundle constraints.

Foodgrain export–import network. The nodes in this network correspond to geographically dispersed locations in different countries, and the arcs correspond to shipments by rail, truck, and ocean freighter. Between these locations, the com-

modities are various foodgrains, such as corn, wheat, rice, and soybeans. The capacities at the ports define the bundle constraints.

Application 17.2 Warehousing of Seasonal Products

A company manufactures multiple products. The products are seasonal, with demands varying weekly, monthly, or quarterly. To use its work force and capital equipment efficiently, the company wishes to "smooth" production, storing preseason production to supplement peak-season production. The company has a warehouse with fixed capacity R that it uses to store all the products it produces. Its decision problem is to identify the production levels of all the products for every week, month, or quarter of the year that will permit it to satisfy the demands incurring the minimum possible production and storage costs.

We can view this warehousing problem as a multicommodity flow problem defined on an appropriate network. For simplicity, consider a situation in which the company makes two products and the company needs to schedule its production for each of the next four quarters of the year. Let d_j^1 and d_j^2 denote the demand for products 1 and 2 in quarter j. Suppose that the production capacity for the jth quarter is u_j^1 and u_j^2, and that the per unit cost of production for this quarter is c_j^1 and c_j^2. Let h_j^1 and h_j^2 denote the storage (holding) costs of the two products from quarter j to quarter $j + 1$.

Figure 17.2 shows the network corresponding to the warehousing problem. The network contains one node for each time period (quarter) as well as a source and sink node for each commodity. The supply and demand of the source and sink nodes is the total demand for the commodity over all four quarters. Each source node s^k has four outgoing arcs, one corresponding to each quarter. Only one commodity flows on each of these arcs. We associate a cost c_j^k and capacity u_j^k with arc (s^k, j). Similarly, the sink node t^k has four incoming arcs; sink arc (j, t^k) has a zero cost and capacity d_j^k. The remaining arcs are of the form $(j, j + 1)$ for $j = 1, 2, 3$; the flow on these arcs represents the units stored from period j to period $j + 1$. Each of these storage arcs has a capacity R, a per unit flow of h_j^1 for commodity 1 and a per unit flow cost of h_j^2 for commodity 2. The two commodities share the capacity of this arc.

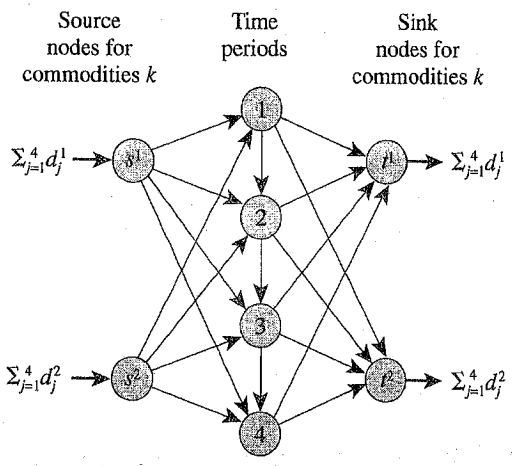


Figure 17.2 Optimal warehousing of seasonal products.

It is easy to see that each feasible multicommodity flow x in the network specifies a feasible production and inventory schedule for the two products with the same cost as the flow x. By optimizing the multicommodity flow, we find the optimal production and inventory plan.

The warehousing problem we have considered is a relatively simple model; we can augment it to include more realistic complexities: for example, transportation expenses incurred between plant–warehouse, warehouse–retailer, and plant–retailer combinations. It is relatively straightforward to incorporate these features in our model (see Exercise 17.2).

Application 17.3 Multivehicle Tanker Scheduling

Suppose that we wish to determine the optimal routing of fuel oil tankers required to achieve a prescribed schedule of deliveries: Each delivery is a shipment with a given delivery date of some commodity from a point of supply to a point of demand. In the simplest form this problem considers a single product (e.g., aviation gasoline or crude oil) to be delivered by a single type of tanker. We discussed this simple version of the problem in Application 6.6 and showed how to determine the minimum tanker fleet to meet the delivery schedule by solving a maximum flow problem. The multivehicle tanker scheduling problem, studied in this application, considers the scheduling and routing of a fixed fleet of nonhomogeneous tankers to meet a prespecified set of shipments of multiple products. The tankers differ in their speeds, carrying capabilities, and operating costs.

To formulate the multivehicle tanker scheduling problem as a multicommodity flow problem, we let the different commodities correspond to different tanker types. The network corresponding to the multivehicle tanker scheduling problem is similar to that of the single vehicle type, shown in Figure 6.9, except that each distinct type of tanker originates at a unique source node s^k. This network has four types of arcs (see Figure 17.3 for a partial example with two tanker types): in-service arcs, out-of-service arcs, delivery arcs, and return arcs. An in-service arc corresponds to the initial employment of a tanker type; the cost of this arc is the cost of deploying the tanker at the origin of the shipment. Similarly, an out-of-service arc corresponds to the removal of the tanker from service. A delivery arc (i, j) represents a shipment from origin i to destination j; the cost c_{ij}^k of this arc is the operating cost of carrying the shipment by a tanker of type k. A return arc (j, k) denotes the movement ("back-

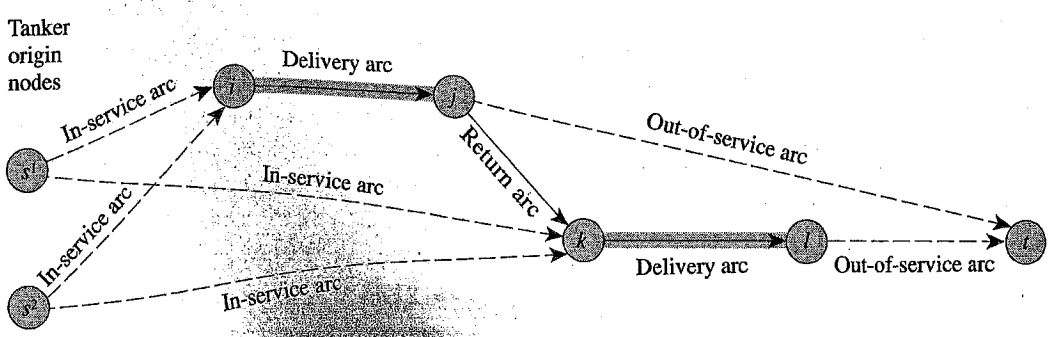


Figure 17.3 Multivehicle tanker scheduling problem as a multicommodity flow problem.

haul") of an empty tanker, with an appropriate cost, between two consecutive shipments (i, j) and (k, l).

Each arc in the network has a capacity of 1. The shipment arcs have a bundle capacity ensuring that at most one tanker type services that arc. Each shipment arc also has a lower flow bound of 1 unit, which ensures that the chosen schedule does indeed deliver the shipment. Some arcs might also have commodity-based capacities u_{ij}^k. For instance, if tanker type 2 is not capable of handling the shipment on arc (i, j), we set $u_{ij}^2 = 0$. Moreover, if tanker type 2 can use the return arc (j, k), but tanker type 1 cannot (because it is too slow to make the connection between shipments), we set $u_{jk}^1 = 0$.

Airline scheduling is another important application domain for this type of model. In this problem context, the vehicles are different types of airplanes in an airline's fleet (e.g., Boeing 727s or 747s or McDonald Douglas DC 10s). The delivery arcs in this problem context are the flight legs that the airline wishes to cover.

We might note that in this formulation of the multivehicle tanker scheduling problem, we are interested in integer solutions of the multicommodity flow problem. The solutions obtained by the multicommodity flow algorithms described in this chapter need not be integral. Nevertheless, the fractional solution might be useful in several ways. For example, we might be able to convert the nonintegral solution into a (possibly, suboptimal) integral solution by minor tinkering, or as we noted earlier, we might use the nonintegral solution as a bound in solving the integer-valued problem by a branch-and-bound enumeration procedure.

17.3 OPTIMALITY CONDITIONS

Throughout our previous development, as we introduced each new problem, we usually began by stating optimality conditions for characterizing when a given feasible solution was optimal. Doing so permitted us to assess whether or not we have found an optimal solution to the problem. It also permitted us to interpret various algorithms as particular methods for solving the optimality conditions, and in several instances even suggested novel algorithmic approaches for solving the problem we were studying. Optimality conditions for the multicommodity flow problem serve these same purposes; so before discussing algorithms for solving the problem, we describe these conditions. For this discussion we assume that the flow variables x_{ij}^k have no individual flow bounds; that is, each $u_{ij}^k = +\infty$ in the formulation (17.1).

Since the multicommodity flow problem is a linear program, we can use linear programming optimality conditions to characterize optimal solutions to the problem. These conditions assume a particularly simple, and familiar, form for the multicommodity flow problem. Since the linear programming formulation (17.1) of the problem has one bundle constraint for every arc (i, j) of the network and one mass balance constraint for each node–commodity combination, the dual linear program has two types of dual variables: a *price* w_{ij} on each arc (i, j) and a *node potential* $\pi^k(i)$ for each combination of commodity k and node i. Using these dual variables, we define the reduced cost $c_{ij}^{\pi,k}$ of arc (i, j) with respect to commodity k as follows:

$$c_{ij}^{\pi,k} = c_{ij}^k + w_{ij} - \pi^k(i) + \pi^k(j).$$

In matrix notation, this definition is $c^{\pi,k} = c^k + w - \pi^k \mathcal{N}$.

Note that if we consider a fixed commodity k, this reduced cost is similar to the reduced cost that we have used previously for the minimum cost flow problem; the difference is that we now add the arc price w_{ij} to the arc cost c_{ij}^k. Note that just as the bundle constraints provided a linkage between the otherwise independent commodity flow variables x_{ij}^k, the arc prices w_{ij} provide a linkage between the otherwise independent commodity reduced costs.

To use linear programming duality theory to characterize optimal solutions to the multicommodity flow problem, we first write the dual of the multicommodity flow problem (17.1):

$$\text{Maximize} \quad -\sum_{(i,j)\in A} u_{ij}w_{ij} + \sum_{k=1}^{K} b^k\pi^k$$

subject to

$$c_{ij}^{\pi,k} = c_{ij}^k + w_{ij} - \pi^k(i) + \pi^k(j) \geq 0 \qquad \text{for all } (i,j) \in A \text{ and all } k = 1, \ldots, K,$$

$$w_{ij} \geq 0 \qquad \text{for all } (i,j) \in A.$$

The optimality conditions for a linear programming, called the complementary slackness (optimality) conditions, state that a primal feasible solution x and a dual feasible solution (w, π^k) are optimal to the respective problems if and only if the product of each primal (dual) variable and the slack in the corresponding dual (primal) constraint is zero. The complementary slackness conditions for the primal–dual pair of the multicommodity flow problem assume the following special form. (In this statement, we use y_{ij}^k to denote a specific value of the flow variable x_{ij}^k.)

Multicommodity flow complementary slackness conditions. *The commodity flows y_{ij}^k are optimal in the multicommodity flow problem (17.1) with each $u_{ij}^k = +\infty$ if and only if they are feasible and for some choice of (nonnegative) arc prices w_{ij} and (unrestricted in sign) node potentials $\pi^k(i)$, the reduced costs and arc flows satisfy the following complementary slackness conditions:*

(a) $w_{ij}\left(\sum_{1\leq k\leq K} y_{ij}^k - u_{ij}\right) = 0$ *for all arcs* $(i,j) \in A$. (17.2a)

(b) $c_{ij}^{\pi,k} \geq 0$ *for all arcs* $(i,j) \in A$ *and all commodities* $k = 1, 2, \ldots, K$. (17.2b)

(c) $c_{ij}^{\pi,k} y_{ij}^k = 0$ *for all arcs* $(i,j) \in A$ *and all commodities* $k = 1, 2, \ldots, K$. (17.2c)

We refer to any set of arc prices and node potentials that satisfy the complementary slackness conditions as *optimal arc prices* and *optimal node potentials*. The following theorem shows the connection between the multicommodity and single-commodity flow problems.

Theorem 17.1 (Partial Dualization). *Let y_{ij}^k be optimal flows and let w_{ij} be optimal arc prices for the multicommodity flow problem (17.1). Then for each commodity k, the flow variables y_{ij}^k for $(i,j) \in A$ solve the following (uncapacitated) minimum cost flow problem:*

$$\min\{\sum_{(i,j)\in A} (c_{ij}^k + w_{ij})x_{ij}^k : \mathcal{N}x^k = b,\ x_{ij}^k \geq 0 \text{ for all } (i,j) \in A\}. \qquad (17.3)$$

Proof. Since y_{ij}^k are optimal flows and w_{ij} are optimal arc prices for the multicommodity flow problem (17.1), these variables together with some set of node potentials $\pi^k(i)$ satisfy the complementary slackness condition (17.2). Now notice that conditions (17.2b) and (17.2c) are the optimality conditions for the uncapacitated minimum cost flow problem for commodity k with arc costs $(c_{ij}^k + w_{ij})$ [see condition (9.8) in Section 9.3 with $u_{ij} = \infty$]. This observation implies that the flows y_{ij}^k solve the corresponding minimum cost flow problems. ◆

This property shows that we can use a sequential approach for obtaining optimal arc prices and node potentials: We first find optimal arc prices and then attempt to find the optimal node potentials and flows by solving the single-commodity minimum cost flow problems (17.3). In the next few sections we use this observation to develop and assess algorithms for solving the multicommodity flow problem.

To illustrate the partial dualization result, we consider a numerical example. The multicommodity flow problem shown in Figure 17.4 has two commodities and two arcs with bundle capacities: arc (s^1, t^1) with a capacity of 5 units and arc (1, 2)

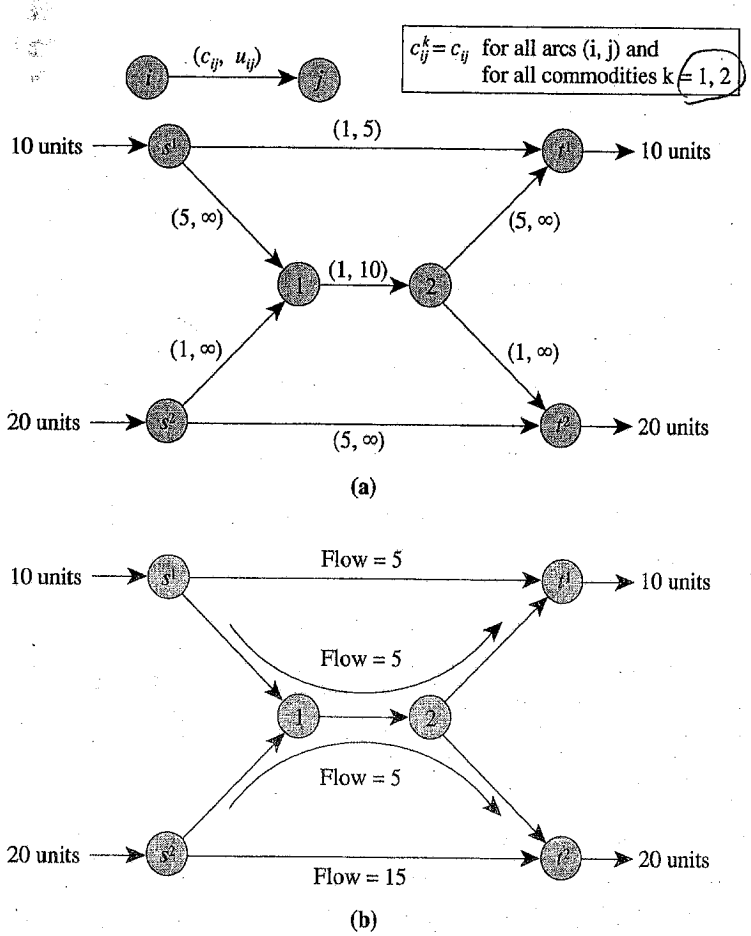


Figure 17.4 Multicommodity flow example: (a) problem data; (b) optimal solution.

with a capacity of 10 units. For this example, since each commodity has a single source and sink, for any choice of the arc prices w_{ij}, the network flow problem (17.3) is a shortest path problem. As shown in Figure 17.4(b), the optimal solution of this problem has $y^1_{s^1 t^1} = y^1_{s^1 1} = y^1_{12} = y^1_{2t^1} = y^2_{s^2 1} = y^2_{12} = y^2_{2t^2} = 5$, $y^2_{s^2 t^2} = 15$, and all other $y^k_{ij} = 0$. Note that the total flow on arc (s^1, t^1) and on arc (1, 2) equals its bundle capacity. Suppose that we set $w_{12} = 2$ and $w_{s^1 t^1} = 12$. Then with respect to the arc costs $c^k_{ij} + w_{ij}$, the shortest path distances for commodity 1 are $d^1(s^1) = 0$, $d^1(1) = 5$, $d^1(2) = 8$, $d^1(t^1) = 13$, $d^1(t^2) = 9$, $d^1(s^2) = +\infty$ and the shortest path distances for commodity 2 are $d^2(s^2) = 0$, $d^2(1) = 1$, $d^2(2) = 4$, $d^2(t^2) = 5$, $d^2(t^1) = 9$, $d^2(s^1) = +\infty$. If we set, $\pi^k(i) = -d^k(i)$ for all nodes i and for both commodities $k = 1, 2$, the node potentials $\pi^k(i)$, arc prices w_{ij}, and arc flows y^k_{ij} given earlier satisfy the optimality conditions (17.2). Therefore, we have verified that y is an optimal solution to the problem and that w, π^1, π^2 are optimal arc prices and node potentials.

17.4 LAGRANGIAN RELAXATION

To apply Lagrangian relaxation to the multicommodity flow problem, we associate nonnegative Lagrange multipliers w_{ij} with the bundle constraints (17.1b), creating the following Lagrangian subproblem:

$$L(w) = \min \sum_{1 \le k \le K} c^k x^k + \sum_{(i,j) \in A} w_{ij} \Big(\sum_{1 \le k \le K} x^k_{ij} - u_{ij} \Big) \tag{17.4a}$$

or, equivalently,

$$L(w) = \min \sum_{1 \le k \le K} \sum_{(i,j) \in A} (c^k_{ij} + w_{ij}) x^k_{ij} - \sum_{(i,j) \in A} w_{ij} u_{ij} \tag{17.4b}$$

subject to

$$\mathcal{N} x^k = b^k \qquad \text{for all } k = 1, \ldots, K, \tag{17.4c}$$

$$x^k_{ij} \ge 0 \qquad \text{for all } (i, j) \in A \text{ and all } k = 1, 2, \ldots, K. \tag{17.4d}$$

Note that since the term $-\sum_{(i,j) \in A} w_{ij} u_{ij}$ in the objective function of the Lagrangian subproblem is a constant for any given choice of the Lagrange multipliers, for any fixed value of these multipliers, this term is a constant and therefore we can ignore it. The resulting objective function for the Lagrangian subproblem has a cost of $c^k_{ij} + w_{ij}$ associated with every flow variable x^k_{ij}. Since none of the constraints in this problem contains the flow variables for more than one of the commodities, the problem decomposes into separate minimum cost flow problems, one for each commodity. Consequently, to apply the subgradient optimization procedure from Section 16.3 to this problem, we would alternately (1) solve a set of minimum cost flow problems (for a fixed value of the Lagrange multipliers w) with the cost coefficients $c^k_{ij} + w_{ij}$, and (2) update the multipliers by the algorithmic procedures described in Section 16.3. In this case, if y^k_{ij} denotes the optimal solution to the minimum cost flow subproblems when the Lagrange multipliers have the value w^q_{ij} at the qth iteration, the subgradient update formula becomes

$$w^{q+1}_{ij} = \Big[w^q_{ij} + \theta_q \Big(\sum_{1 \le k \le K} y^k_{ij} - u_{ij} \Big) \Big]^+.$$

In this expression, the notation $[\alpha]^+$ denotes the positive part of α, that is, $\max(\alpha, 0)$. The scalar θ_q is a step size specifying how far we move from the current solution w_{ij}^q. Note that this update formula increases the multiplier w_{ij}^q on arc (i, j) by the amount $(\sum_{1 \le k \le K} y_{ij}^k - u_{ij})$ if the subproblem solutions y_{ij}^k use more than the available capacity u_{ij} of that arc, or reduces the Lagrange multiplier of arc (i, j) by the amount $(u_{ij} - \sum_{1 \le k \le K} y_{ij}^k)$ if the subproblem solutions y_{ij}^k use less than the available capacity of that arc. If, however, the decrease would cause the multiplier w_{ij}^{q+1} to become negative, we reduce the multiplier to value zero. We choose the step sizes θ_q for iterations $q = 1, 2, \ldots$, in accordance with the procedures described in Chapter 16.

It is instructive to view the subgradient method for the multicommodity flow problem as a solution procedure for solving the linear program (17.1) that is able to exploit the special structure of the unrelaxed mass balance constraints. In Theorem 16.6 we noted that whenever we apply Lagrangian relaxation to any linear program, such as the multicommodity flow problem, the optimal value $L^* = \max_{w \ge 0} L(w)$ of the Lagrangian multiplier problem equals the optimal objective function value z^* of the linear program. We illustrate this technique with a numerical example.

Consider again the two-commodity example shown in Figure 17.4. As we noted in our discussion in Section 17.3, for any choice of the Lagrange multipliers w_{12} and $w_{s^1t^1}$ for the two capacitated arcs (s^1, t^1) and $(1, 2)$, the problem decomposes into two shortest path problems. If we start with the Lagrange multipliers $w_{12}^0 = w_{s^1t^1}^0 = 0$, then in the subproblem solutions, the shortest path s^1–t^1 carries 10 units of flow at a cost of 1(10) = 10 and the path s^2–1–2–t^2 carries 20 units of flow at a cost of 3(20) = 60. Therefore, $L(0) = 10 + 60 = 70$ is a lower bound on the optimal objective function value for the problem. Since $y_{s^1t^1}^1 + y_{s^1t^1}^2 - u_{s^1t^1} = 5$ and $y_{12}^1 + y_{12}^2 - u_{12} = 10$, the update formulas become

$$w_{s^1t^1}^1 = [0 + \theta_0 \cdot 5]^+ \quad \text{and} \quad w_{12}^1 = [0 + \theta_0 \cdot 10]^+.$$

If we choose $\theta_0 = 1$, then $w_{s^1t^1}^1 = 5$ and $w_{12}^1 = 10$. The new shortest path solutions send 10 units on the path s^1–t^1 at a cost of (1 + 5) (10) = 60 and 20 units on the path s^2–t^2 at a cost of 5(20) = 100. The new lower bound obtained through (17.4b) is $60 + 100 - w_{12}^1 u_{12} - w_{s^1t^1}^1 u_{s^1t^1} = 160 - 10(10) - 5(5) = 35$. (Note that the value of the lower bound has decreased.) At this point, $y_{s^1t^1}^1 + y_{s^1t^1}^2 - u_{s^1t^1} = 5$ and $y_{12}^1 + y_{12}^2 - u_{12} = 0$, so the update formulas become

$$w_{s^1t^1}^1 = [5 + \theta_0 \cdot 5]^+ \quad \text{and} \quad w_{12}^1 = [10 - \theta_0 \cdot 10]^+.$$

Choosing the step size $\theta_1 = 1$ again, we find that $w_{12}^1 = 0$ and $w_{s^1t^1}^1 = 10$ and that the new shortest path solutions send 10 units on the path s^1–t^1 at a cost of (1 + 10) (10) = 110 and 20 units on the path s^2–1–2–t^2 at a cost of 3(20) = 60. If we continue by choosing the step sizes for the kth iteration as $\theta_k = 1/k$, in accordance with the theory of subgradient optimization as discussed in Section 16.4, we obtain the set of iterates shown in Figure 17.5. From iteration 14 on, the values of the Lagrange multipliers oscillate about, and converge to, their optimal values $w_{12} = 2$ and $w_{s^1t^1} = 12$ and the optimal lower bound oscillates about its optimal value 150, which equals the optimal objective function value of the multicommodity flow problem.

Iteration number q	w_{12}^q	$w_s^{q1}{}_{t^1}$	Shortest paths	Shortest path costs	$10w_{12}^q + 5w_s^{q1}{}_{t^1}$	Lower bound $L(w^q)$	θ_q
0	0	0	s^1–t^1 s^2–1–2–t^2	10(1) 20(1 + 1 + 1)	0	70	1
1	10	5	s^1–t^1 s^2–t^2	10(1 + 5) 20(5)	125	35	1
2	0	10	s^1–t^1 s^2–1–2–t^2	10(1 + 10) 20(1 + 1 + 1)	50	120	0.5
3	5	12.5	s^1–t^1 s^2–t^2	10(1 + 12.5) 20(5)	112.5	122.5	0.333
4	1.67	14.17	s^1–1–2–t^1 s^2–1–2–t^2	10(5 + 2.67 + 5) 20(1 + 2.67 + 1)	87.55	132.5	0.25
5	6.67	12.92	s^1–t^1 s^2–t^2	10(1 + 12.92) 20(5)	131.3	107.9	0.2
6	4.67	13.92	s^1–t^1 s^2–t^2	10(1 + 13.92) 20(5)	116.3	132.9	0.167
7	3	14.75	s^1–1–2–t^1 s^2–t^2	10(5 + 4 + 5) 20(5)	103.8	136.3	0.143
8	3	14.04	s^1–1–2–t^1 s^2–t^2	10(5 + 4 + 5) 20(5)	100.2	139.8	0.125
9	3	13.41	s^1–1–2–t^1 s^2–t^2	10(5 + 4 + 5) 20(5)	97.05	143.0	0.111
10	3	12.86	s^1–t^1 s^2–t^2	10(1 + 12.86) 20(5)	94.3	144.3	0.1
11	2	13.36	s^1–1–2–t^1 s^2–t^2	10(5 + 3 + 5) 20(5)	86.8	143.2	0.091
12	2	12.9	s^1–1–2–t^1 s^2–t^2	10(5 + 3 + 5) 20(5)	84.5	145.5	0.083
13	2	12.48	s^1–1–2–t^1 s^2–t^2	10(5 + 3 + 5) 20(5)	82.2	147.6	0.077
14	2	12.09	s^1–1–2–t^1 s^2–t^2	10(5 + 3 + 5) 20(5)	80.25	149.5	0.071

Figure 17.5 Application of subgradient optimization to a multicommodity flow problem.

Figures 17.6 and 17.7 show how the values of the Lagrange multipliers and Lagrangian lower bounds vary during the execution of the algorithm. Note that neither the multiplier values nor the lower bounds converge monotonically to their optimal values. In this example, as in the application of Lagrangian relaxation in general, these values tend to oscillate, sometimes considerably, before settling down to their optimal values.

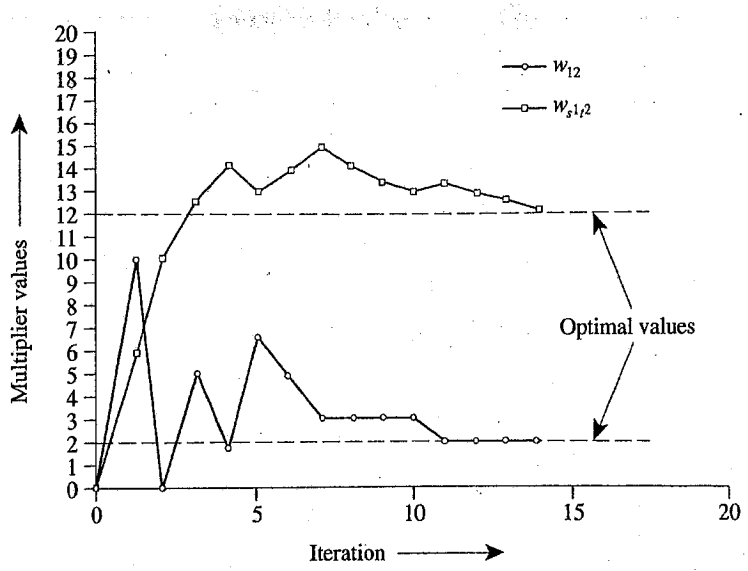


Figure 17.6 Multiplier values for the numerical example.

The use of subgradient optimization for solving the Lagrangian multiplier problem is attractive for several reasons. First, as we have just noted, this solution approach permits us to exploit the underlying network flow structure. Second, the formulas for updating the Lagrange multipliers w_{ij} are rather trivial computationally and very easy to encode in a computer program. This solution approach, however, also has some limitations. To ensure convergence we need to take small step sizes; as a result, the method does not converge very fast. Second, the method is dual-based, and so even though it is converging to the optimal dual variables w_{ij}, the optimal solutions y_{ij}^k of the subproblems need not converge to the optimal solution to the multicommodity flow problem. Indeed, even though we have shown that if we set the Lagrangian multipliers to their optimal values [i.e., as the optimal dual

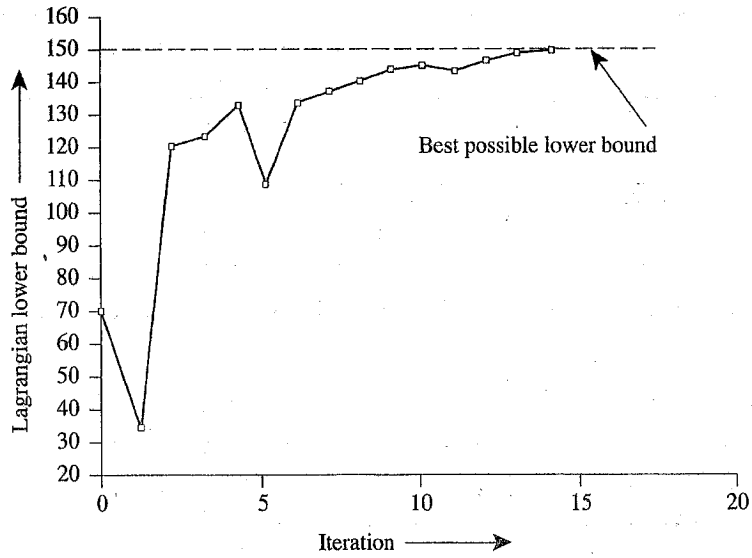


Figure 17.7 Lagrangian lower bound for the numerical example.

variables for the bundle constraints in the dual to the linear program (17.1)], the optimal flows y_{ij}^k solve the Lagrangian subproblem (17.4), these subproblems might also have other optimal solutions that do not satisfy the bundle constraints. For example, for the problem we have just solved, with the optimal Lagrange multipliers $w_{12} = 2$ and $w_{s^1t^1} = 12$, the shortest paths subproblems have solutions with 10 units on the path s^1–t^1 and 20 units on the path s^2–t^2. This solution violates the capacity of the arc (s^1, t^1). In general, to obtain optimal flows, even after we have solved the Lagrangian multiplier problem, requires additional work (see Exercise 17.18).

To conclude this discussion of Lagrangian relaxation, we might note that we can also combine Lagrangian relaxation with linear programming by using Lagrangian relaxation to develop an "advance basis" start for the linear programming formulation (17.1). Suppose that we solve the Lagrangian subproblem for *any* choice of the Lagrange multipliers by the network simplex algorithm (see Chapter 11). By doing so we obtain an optimal spanning tree solution for each commodity, which corresponds to a basis $\mathcal{B}^k$ of the network flow constraints $\mathcal{N}x^k = b^k$. We can extend these bases into an overall basis $\mathcal{B}$ of the linear program (17.1), as shown in Figure 17.8.

The identity matrix in the topmost row in this matrix corresponds to the slack variables s_{ij} in the equality formulation (17.1b′) of the bundle constraints. Note that in the solution corresponding to the basis $\mathcal{B}$, (1) each flow variable x^k for $k = 1, 2, \ldots, K$ equals the solution y^k to the Lagrangian subproblem, and (2) the values of the slack variables are given by $s_{ij} = u_{ij} - \sum_{1 \le k \le K} y_{ij}^k$. If all the slack variables are nonnegative, this basis is feasible. If any of them are negative, the basis is infeasible. In either case we can apply standard linear programming methods (either the primal simplex method or the dual simplex method) using this basis as a starting solution for solving the problem to completion.

This advanced basis defines an attractive starting solution for the simplex method because it permits us to exploit the underlying network structure and flow costs to generate a "smart" starting solution for the algorithm. For problems that are "modestly capacitated," the advanced basis can be a very good approximation

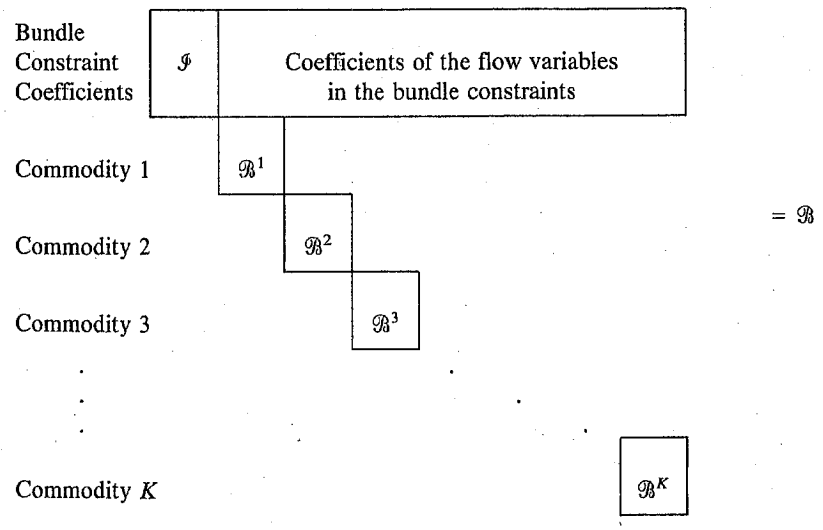


Figure 17.8 Initial basis formed from bases of individual commodity subproblems.

of the optimal solution of the problem and so can greatly improve the performance of the simplex method.

We next consider an alternative solution approach, known as *Dantzig–Wolfe decomposition*, for solving the Lagrangian multiplier problem. This approach requires considerably more work at each iteration for updating the Lagrange multipliers (the solution of a linear program) but has proved to converge faster than the subgradient optimization procedure for several classes of problems. Rather than describing the Dantzig–Wolfe decomposition procedure as a variation of Lagrangian relaxation, we will first develop it from an alternate large-scale linear programming viewpoint that provides a somewhat different perspective on the approach.

17.5 COLUMN GENERATION APPROACH

To simplify our discussion in this section, we consider a special case of the multicommodity flow problem: We assume that each commodity k has a single source node s^k and a single sink node t^k and a flow requirement of d^k units between these source and sink nodes. We also assume that we impose no flow bounds on the individual commodities other than the bundle constraints. Therefore, for each commodity k, the subproblem constraints $\mathcal{N}x^k = b^k$, $x^k \geq 0$ define a shortest path problem: For this model, for any choice w_{ij} of the Lagrange multipliers for the bundle constraints, the Lagrangian relaxation requires the solution of a series of shortest path problems, one for each commodity.

Reformulation with Path Flows

To begin our discussion in this section, let us first reformulate the multicommodity flow problem using path and cycle flows instead of arc flows. Recall from Section 3.5 that we can formulate any network flow problem using path and cycle flows. To simplify our discussion even further, let us assume that for every commodity the cost of every cycle W in the underlying network is nonnegative (in Exercise 17.8 we relax this assumption). The problem satisfies this condition, for example, if the arc flow costs are all nonnegative. If we impose this nonnegative cycle cost condition, then in some optimal solution to the problem, the flow on every cycle is zero, so we can eliminate the cycle flow variables. Therefore, throughout this section we assume that we can represent any potentially optimal solution as the sum of flows on directed paths. Let us recall our notation from Section 3.5 concerning path and cycle decompositions, tailored a bit for the multicommodity flow problem.

For each commodity k, let $\mathbf{P^k}$ denote the collection of all directed paths from the source node s^k to the sink node t^k in the underlying network $G = (N, A)$. In the path flow formulation, each decision variable $f(P)$ is the flow on some path P and for the kth commodity, we define this variable for every directed path P in $\mathbb{P}^k$.

As in Section 3.5, let $\delta_{ij}(P)$ be an arc-path indicator variable, that is, $\delta_{ij}(P)$ equals 1 if arc (i, j) is contained in the path P, and is 0 otherwise. The flow decomposition theorem of network flows states that we can always decompose some optimal arc flow x_{ij}^k into path flows $f(P)$ as follows:

$$f(P) = x_{ij}^k \sum_{P \in \mathbf{P^k}} \delta_{ij}(P) f(P).$$

Let $c^k(P) = \sum_{(i,j)\in A} c_{ij}^k \delta_{ij}(P) = \sum_{(i,j)\in P} c_{ij}^k$ denote the per unit cost of flow on the path $P \in \mathbf{P^k}$ with respect to the commodity k. Note that for each commodity k, if we substitute for the arc flow variables in the objective function, interchange the order of the summations, and collect terms, we find that

$$\sum_{(i,j)\in A} c_{ij}^k x_{ij}^k = \sum_{(i,j)\in A} c_{ij}^k \left[\sum_{P\in \mathbf{P^k}} \delta_{ij}(P) f(P)\right] = \sum_{P\in \mathbf{P^k}} c^k(P) f(P).$$

This observation shows that we can express the cost of any solution as either the cost of arc flows or the cost of path flows.

By substituting the path variables in the multicommodity flow formulation, we obtain the following equivalent path flow formulation of the problem:

$$\text{Minimize} \quad \sum_{1\le k\le K} \sum_{P\in \mathbf{P^k}} c^k(P) f(P) \tag{17.5a}$$

subject to

$$\sum_{1\le k\le K} \sum_{P\in \mathbf{P^k}} \delta_{ij}(P) f(P) \le u_{ij} \qquad \text{for all } (i, j) \in A, \tag{17.5b}$$

$$\sum_{P\in \mathbf{P^k}} f(P) = d^k \qquad \text{for all } k = 1, 2, \ldots, K, \tag{17.5c}$$

$$f(P) \ge 0 \qquad \text{for all } k = 1, 2, \ldots, K \text{ and all } P \in \mathbf{P^k}. \tag{17.5d}$$

In formulating this problem we have invoked the flow decomposition theorem stating that we can decompose any feasible arc flow of the system $\mathcal{N}x^k = b^k$ into a set of path and cycle flows in such a way that the path flows satisfy the mass balance condition (17.5c).

Note that the path flow formulation of the multicommodity flow problem has a very simple constraint structure. The problem has a single constraint for each arc (i, j) which states that the sum of the path flows passing through the arc is at most u_{ij}, the capacity of the arc. Moreover, the problem has a single constraint (17.5c) for each commodity k which states that the total flow on all the paths connecting the source node s^k and sink node t^k of commodity k must equal the demand d^k for this commodity. For a network with n nodes, m arcs, and K commodities, the path flow formulation contains $m + K$ constraints (in addition to the nonnegativity restrictions imposed on the path flow values). In contrast, the arc formulation (17.1) contains $m + nK$ constraints since it contains one mass balance constraint for every node and commodity combination. For example, a network with $n = 1000$ nodes and $m = 5000$ arcs and with a commodity between every pair of nodes has approximately $K \approx n^2 = 1{,}000{,}000$ commodities. Therefore, the path flow formulation contains about 1,005,000 constraints. In contrast, the arc flow formulation contains about 1,000,005,000 constraints. But the difference is even more pronounced: Because no path appears in more than one of the constraints (17.5c), we can apply a specialized version of the simplex method, known as the *generalized upper bounding simplex method*, to solve the path flow formulation very efficiently. Even though the linear programming basis for our example has size 1,005,000 by 1,005,000, the generalized upper bounding simplex method is able to perform all of its matrix computations on a much smaller basis of size 5000 by 5000. This method essentially solves the problem as though it contained only m bundle constraints, which, for this

sample data, means that we can essentially solve a linear program with only 5000 constraints instead of over 1 billion constraints in the arc formulation.

This savings in the number of constraints does come at a cost, however, since the path flow formulation has a variable for every path connecting a source and sink node for each of the commodities. The number of variables will typically be enormous, growing exponentially in the size of the network. All hope is not lost, though, since we might expect that only very few of the paths will carry flow in the optimal solution to the problem. In fact, linear programming theory permits us to show that at most $K + m$ paths carry positive flow in some optimal solution to the problem (see Exercise 17.6). Therefore, for a problem with 1,000,000 commodities and 5,000 arcs as in our previous example, we could, in principle, solve the path flow formulation using 1,005,000 paths. Since the problem contains 1,000,000 commodities, this solution would use two or more paths for at most 5,000 commodities and one path for at least the 995,000 remaining commodities. If we knew the optimal set of paths, or a very good set of paths, we could obtain an optimal solution (i.e., values for the path flows) by solving a linear program containing just the commodities with two or more sets of paths. The generalized upper bounding linear programming procedure for solving linear programs permits us to exploit this observation.

Optimality Conditions

Recall that the revised simplex method of linear programming maintains a basis at every step, and using this basis determines a vector of simplex multipliers for the constraints. Since the path flow formulation (17.5) contains one bundle constraint for each arc and one demand constraint (17.5c) for every commodity, the dual linear program has a dual variable w_{ij} for each arc (this is the same arc price that we have introduced before) and another dual variable σ^k for each commodity $k = 1, 2, \ldots, K$. With respect to these dual variables, the reduced cost $c_P^{\sigma,w}$ for each path flow variable $f(P)$ is

$$c_P^{\sigma,w} = c^k(P) + \sum_{(i,j)\in P} w_{ij} - \sigma^k.$$

and the complementary slackness conditions (17.2) for the arc formulation of the original problem assume the following form:

Path flow complementary slackness conditions. *The commodity path flows $f(P)$ are optimal in the path flow formulation* (17.5) *of the multicommodity flow problem if and only if for some arc prices w_{ij} and commodity prices σ^k, the reduced costs and arc flows satisfy the following complementary slackness conditions:*

$$\text{(a)} \quad w_{ij}\left[\sum_{1\le k\le K}\ \sum_{P\in\mathbb{P}^k} \delta_{ij}(P)f(P) - u_{ij}\right] = 0 \text{ for all } (i,j) \in A. \tag{17.6a}$$

$$\text{(b)} \quad c_P^{\sigma,w} \ge 0 \text{ for all } k = 1, 2, \ldots, K \text{ and all } P \in \mathbb{P}^k. \tag{17.6b}$$

$$\text{(c)} \quad c_P^{\sigma,w} f(P) = 0 \text{ for all } k = 1, 2, \ldots, K \text{ and all } P \in \mathbb{P}^k. \tag{17.6c}$$

We leave the proof of these conditions as an exercise for the reader. These optimality conditions have a very appealing and intuitive interpretation. Condition

(a) states that the price w_{ij} of arc (i, j) is zero if the optimal solution $f(P)$ does not use all of the capacity u_{ij} of the arc. That is, if the optimal solution does not fully use the capacity of that arc, we could ignore the constraint (place no price on it).

Since the cost $c^k(P)$ of path P is just the sum of the cost of the arcs contained in that path, that is, $c^k(P) = \sum_{(i,j)\in P} c_{ij}^k$, we can write the reduced cost of path P as

$$c_P^{\sigma,w} = \sum_{(i,j)\in P} (c_{ij}^k + w_{ij}) - \sigma^k.$$

That is, the reduced cost of path P is just the cost of that path with respect to the modified costs $c_{ij}^k + w_{ij}$ minus the commodity cost σ^k. The complementary slackness condition (17.6b) states that the modified path cost $\sum_{(i,j)\in P} (c_{ij}^k + w_{ij})$ for each path connecting the source node s^k and the sink node t^k of commodity k must be at least as large as the commodity cost σ^k. The condition (17.6c) implies that reduced cost $c_P^{\sigma,w}$ must be zero for any path P that carries flow in the optimal solution [i.e., for which the flow $f(P)$ is positive]; that is, the modified cost $\sum_{(i,j)\in P} (c_{ij}^k + w_{ij})$ of this path must equal the commodity cost σ^k. Therefore, conditions (17.6b) and (17.6c) imply that

σ^k is the shortest path distance from node s^k to node t^k with respect to the modified costs $c_{ij}^k + w_{ij}$ and in the optimal solution every path from node s^k to node t^k that carries a positive flow must be a shortest path with respect to the modified costs.

This result shows that the arc costs w_{ij} permit us to decompose the multicommodity flow problem into a set of independent "modified" cost shortest path problems.

Column Generation Solution Procedure

To this point we have restated the multicommodity flow problem as a large-scale linear program with an enormous number of columns—with one flow variable for each path connecting the source and sink of any commodity. We have also shown how to characterize any optimal solution to this formulation in terms of the linear programming dual variables w_{ij} and σ^k, interpreting these conditions as shortest path conditions with respect to the modified arc costs $c_{ij}^k + w_{ij}$. We next show how to solve the problem by using a solution procedure known as *column generation.*

The key idea in column generation is never to list explicitly all of the columns of the problem formulation, but rather to generate them only "as needed." The revised simplex method of linear programming is perfectly suited for carrying out this algorithmic strategy. Recall from Appendix C that the revised simplex method maintains a basis $\mathcal{B}$ at each iteration. It uses this basis to define a set of simplex multipliers π via the matrix computation $\pi\mathcal{B} = c_{\mathbf{B}}$ (in our application, the multipliers are w and σ). That is, the method defines the simplex multipliers so that the reduced costs $c_{\mathbf{B}}^{\pi}$ of the basic variables are zero; that is, $c_{\mathbf{B}}^{\pi} = c_{\mathcal{B}} - \pi\mathcal{B} = 0$. To find the simplex multipliers, the method requires no information about columns (variables) not in the basis. It then uses the multipliers to *price-out* the nonbasic columns, that is, compute their reduced costs. If any reduced cost is negative (assuming a min-

imization formulation), the method will introduce one nonbasic variable into the basis in place of one of the current basic variables, recompute the simplex multipliers π, and then repeat these computations. To use the column generation approach, the columns should have structural properties that permit us to perform the pricing out operations without explicitly examining every column.

When applied to the path flow formulation of the multicommodity flow problem, with respect to the current basis at any step (which is composed of a set of columns, or path variables, for the problem), the revised simplex method defines the simplex multipliers w_{ij} and σ^k so that the reduced cost of every variable in the basis is zero. Therefore, if a path P connecting the source s^k and sink t^k for commodity k is one of the basic variables, then $c_P^{\sigma,w} = 0$, or equivalently, $\sum_{(i,j)\in P} (c_{ij}^k + w_{ij}) = \sigma^k$. Therefore, the revised simplex method determines the simplex multipliers w_{ij} and σ^k so that they satisfy the following equations:

$$\sum_{(i,j)\in P} (c_{ij}^k + w_{ij}) = \sigma^k \text{ for every path } P \text{ in the basis.}$$

Notice that since each basis consists of $K + m$ paths, each basis gives rise to $K + m$ of these equations. Moreover, the equations contain $K + m$ variables (i.e., m arc prices w_{ij} and K shortest path distances σ^k). The revised simplex method uses matrix computations to solve the $K + m$ equations and determines the unique values of the simplex multipliers.

The complementary slackness condition (17.6c) dictates that $c_P^{\sigma,w} f(P) = 0$ for every path P in the network. Since each path P in the basis satisfies the condition $c_P^{\sigma,w} = 0$, we can send any amount of flow on it and still satisfy the condition (17.6c). To satisfy this condition for a path P not in the basis, we set $f(P) = 0$. Next consider the complementary slackness condition (17.6a). If the slack variable $s_{ij} = [\sum_{1\le k\le K} \sum_{(i,j)\in P^k} \delta_{ij}(P) f(P) - u_{ij}]$ is not in the basis, $s_{ij} = 0$, so the solution satisfies (17.6a). On the other hand, if the slack variable s_{ij} is in the basis, its reduced cost, which equals $0 - w_{ij}$, is zero, implying that $w_{ij} = 0$ and the solution satisfies (17.6a). We have thus shown that the solution defined by the current basis satisfies conditions (17.6a) and (17.6c); it is optimal if it satisfies condition (17.6b) (i.e., the reduced cost of every path flow variable is nonnegative). How could we check this condition? That is, how can we check to see if for each commodity k,

$$c_P^{\sigma,w} = \sum_{(i,j)\in P} (c_{ij}^k + w_{ij}) - \sigma^k \ge 0 \qquad \text{for all } P \in \mathbf{P^k},$$

or, equivalently,

$$\min_{P\in \mathbf{P^k}} \sum_{(i,j)\in P} (c_{ij}^k + w_{ij}) \ge \sigma^k.$$

As we have noted, the left-hand side of this inequality is just the length of the shortest path connecting the source and sink nodes, s^k and t^k, of commodity k with respect to the modified costs $c_{ij}^k + w_{ij}$. Thus, to see whether the arc prices w_{ij} together with current path distances σ^k satisfy the complementary slackness conditions, we solve a shortest path problem for each commodity k. If for all commodities k, the length of the shortest path for that commodity is at least as large as σ^k, we satisfy the complementary slackness condition (17.6b).

Otherwise, if for some commodity k, Q denotes the shortest path with respect

to the current modified costs $c_{ij}^k + w_{ij}$ and the reduced cost of path Q is less than the length σ^k of the minimum cost path from the set $\mathbf{P^k}$, then

$$c_Q^{\sigma,w} = \sum_{(i,j)\in Q} (c_{ij}^k + w_{ij}) - \sigma^k < 0.$$

In terms of the linear program (17.5), the path Q has a negative reduced cost, so we can profitably use it in the linear program in place of one of the paths P in the current basis $\mathcal{B}$. That is, using the usual steps of the simplex method, we would perform a basis change introducing the path Q into the current basis. Doing so would permit us to determine a new set of arc prices w_{ij} and a new modified shortest path distance σ^k between the source and sink nodes of commodity k. We choose the values of these variables so that the reduced cost of every basic variable is zero. That is, using matrix operations, we would once again solve the system $c_P^{\sigma,w} = \sum_{(i,j)\in P} (c_{ij}^k + w_{ij}) - \sigma^k = 0$ in the variables w_{ij} and σ^k. We would then, as before, solve a shortest path problem for each commodity k and see whether any path has a shorter length than σ^k. If so, we would introduce this path into the basis and continue by alternately (1) finding new values for the arc prices w_{ij} and for the path lengths σ^k, and (2) solving shortest path problems.

This discussion shows us how we would determine the variable to introduce into the basis at each step. The rest of the steps for implementing the simplex method (e.g., determining the variable to remove from the basis at each step) are the same as those of the usual implementation, so we do not specify any further details.

Determining Lower Bounds

Let z^* denote the optimal objective function value of the multicommodity flow problem (17.5) and let z^{lp} denote the optimal objective function value at any step in solving the path flow formulation of the problem (17.5) by the simplex method. Since z^{lp} corresponds to a feasible solution to the problem, $z^* \le z^{\text{lp}}$. As we have noted in our discussion of the Lagrangian relaxation technique in Section 17.4, for any choice of the arc prices w, the optimal value $L(w)$ of the Lagrangian subproblem is a lower bound on z^*. Therefore, suppose that at any point during the course of the algorithm, we solve the Lagrangian subproblem with respect to the current arc prices w_{ij}. That is, we solve for the shortest path lengths $l^k(w)$ for all the commodities k with respect to the modified costs $c_{ij}^k + w_{ij}$. (Notice that this is the same computation that we perform in pricing out columns for the simplex method.) Then from (17.4) the value $L(w)$ of the Lagrangian subproblem is

$$L(w) = \sum_{k=1}^{K} l^k(w) - \sum_{(i,j)\in A} w_{ij}u_{ij},$$

and by the theory of Lagrangian relaxation,

$$L(w) \le z^* \le z^{\text{lp}}.$$

Therefore, as a by-product of finding the shortest path distances $l^k(w)$ as we are pricing out columns in implementing the column generation procedure, we obtain a lower bound on the objective function value. This lower bound allows us to judge the quality of the current solution in the column generation technique and often

terminate the procedure without further computations if the difference between the solution value z^{lp} and lower bound $L(w)$ is sufficiently small. We might note that since at each step of the simplex method, the objective value z^{lp} of the problem stays the same or decreases, the upper bound is monotonically nonincreasing from step to step. On the other hand, the objective value $L(w)$ of the Lagrangian subproblem need not decrease from step to step, so at any point in the algorithm we would use the largest of the values $L(w)$ generated in all previous steps as the best lower bound.

17.6 DANTZIG–WOLFE DECOMPOSITION

In Section 17.5 we showed how to use a column generation procedure to solve the multicommodity flow problem formulated in the space of path flows. In this section we interpret this solution approach in another framework, known as *Dantzig–Wolfe decomposition.* Imagine that K different decision makers as well as one "coordinator" are solving the K-commodity flow problem. Each person plays a special role in solving the problem. The coordinator's job is to solve the path formulation (17.5) of the problem, which we refer to as the "master" or "coordinating" problem. In solving the problem the coordinator does not, however, generate the columns of the master problem; instead, the K decision makers, with guidance from the coordinator in the form of arc prices, generate these columns, with the kth decision maker generating the columns of the master problem corresponding to the kth commodity.

In general, the coordinator has on hand only a subset of the columns of the master problem. Since the coordinator can, at best, solve the linear program as restricted to this subset of columns, we refer to this smaller linear program as the *restricted master problem.*

The path formulation of the multicommodity flow problem has $m + K$ constraints: (1) one for each commodity k, specifying that the flow of commodity k is d^k; and (2) one for each arc (i, j), specifying that the total flow on that arc is at most u_{ij}. The coordinator solves the restricted master problem to optimality using any linear programming technique, such as the simplex algorithm, and then needs to determine whether the solution to the restricted master problem is optimal for the original problem or if some other column has a negative reduced cost. To this end the coordinator broadcasts the optimal set of simplex multipliers (or prices) of the restricted master problem, that is, broadcasts an arc price w_{ij} associated with arc (i, j) and a path length σ^k associated with each commodity k.

After the coordinator has broadcast the prices, the decision maker for commodity k determines the least cost way of shipping d^k units from the source node s^k to the sink t^k of commodity k, assuming that each arc (i, j) has an associated toll of w_{ij} in addition to its arc cost c_{ij}^k. If the cost of this shortest path is less than σ^k, the kth decision maker will report this solution to the coordinator as an improving solution. If the cost of this path equals σ^k, the kth decision maker need not report anything to the coordinator. (The cost will never be greater than σ^k because, as shown by our discussion of optimality conditions in Section 17.5, the coordinator is already using some path of cost σ^k for the kth commodity in his or her optimal solution.) To see that this interpretation is consistent with the preceding section,

note that to price out the columns for commodity k, we need to solve the following shortest path problem:

$$\text{Minimize} \quad \sum_{1 \le k \le K} \sum_{P \in \mathbf{P^k}} [c^k(P) + \sum_{(i,j) \in P} w_{ij}] f(P)$$

subject to

$$\sum_{P \in \mathbf{P^k}} f(P) = d^k \qquad \text{for all } k = 1, 2, \ldots, K,$$

$$f(P) \ge 0 \qquad \text{for all } k = 1, 2, \ldots, K \text{ and all } P \in \mathbf{P^k}.$$

which corresponds, since $c^k(P) = \sum_{(i,j) \in P} c_{ij}^k$, precisely to finding the best way of satisfying the demand for the kth commodity assuming a cost of $c_{ij}^k + w_{ij}$ associated with each arc (i, j). Moreover, the shortest path for commodity k will have a negative reduced cost if and only if the cost for the kth shortest path problem is less than σ^k. We refer to the problems solved by each of the K decision makers as *subproblems* since we are using them solely for the purpose of generating new columns for the restricted master problem or proving optimality of the current solution.

Let us now summarize our discussion. In Section 17.5 we interpreted this procedure as a column generation procedure that determines entering variables by solving K different shortest path problems. The kth shortest path problem corresponds to the shortest path problem for the kth commodity with the added cost of w_{ij} associated with arc (i, j). In the Dantzig–Wolfe interpretation of the procedure, a central coordinator is solving the path formulation of the multicommodity flow problem restricted to the columns that he or she has on hand. After obtaining an optimal solution for this restricted master problem, the coordinator asks each of the K decision makers to solve a shortest path problem using an additional arc cost of w_{ij} on each arc (i, j). Each decision maker (i.e., subproblem) then either provides the coordinator with a new path or says that it can generate no shorter path than the one(s) the coordinator is currently using. Figure 17.9 gives a schematic representation of the information flow in the algorithm.

This method is flexible in several respects. When implemented as the revised simplex method, the algorithm would maintain a linear programming basis and introduce one column at each iteration. We might interpret this approach as maintaining a restricted master problem at each step with only a single column that is

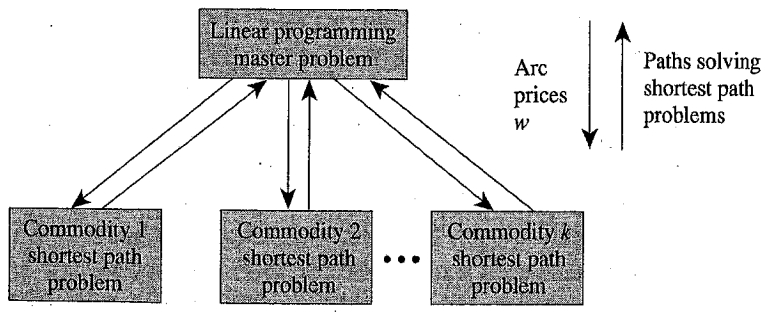


Figure 17.9 Information flow in Dantzig–Wolfe decomposition.

not in the basis; consequently, the algorithm discards any column that leaves the linear programming basis. However, generating new columns might be time consuming, so it might potentially be advantageous to save old columns since they might subsequently have a negative reduced cost. Therefore, in implementing the decomposition algorithm, we could retain some or all of the columns that we have generated previously. As a result, when we solve the restricted master problem, we would generally perform more than one basis change (to solve the restricted master problem to completion) before we broadcast new prices and once again solved the shortest path subproblems.

The column generation procedure and the decomposition procedure share one other potential advantage—the possibility of solving the problem using parallel processors. Once we have determined the arc prices w_{ij} in either procedure, the shortest path subproblems are independent of each other, so we could solve each of them simultaneously, using a separate processor for each shortest path problem.

Since each column of the master problem corresponds to one of a finite number of paths for each commodity, the decomposition solution technique will be finite as long as we never discard any columns of the restricted master problem. (Eventually, we would generate every column in the full master problem.) Alternatively, we could discard columns only when the solution to the restricted master problem strictly improves. Or, we could use an anticycling linear programming-based rule, such as lexicography, for selecting the variable to leave the basis at each iteration; any such rule would guarantee finiteness of the algorithm since it would ensure that we never repeat any of the finite number of bases to the linear program.

Finally, notice that the K subproblems correspond to the relaxed Lagrangian problem with a multiplier of w_{ij} imposed upon each arc (i, j). Consequently, we could view the coordinator as setting the Lagrange multipliers and solving the Lagrangian multiplier problem. In fact, Dantzig–Wolfe decomposition is an efficient method for solving the Lagrangian multiplier problem if we measure efficiency by the number of iterations an algorithm performs. (By Theorem 16.7, the Dantzig–Wolfe algorithm solves the Lagrangian multiplier problem since the multicommodity flow problem is a linear program, and for linear programs, the Lagrangian multiplier problem has the same objective function value as the linear program.) Unfortunately, in applying Dantzig–Wolfe decomposition, at each iteration the coordinator must solve a linear program with $m + K$ constraints, and this update step for the simplex multipliers is very expensive. It is far more time consuming to solve a linear program than to update the multipliers using subgradient optimization. Because each multiplier update for Dantzig–Wolfe decomposition is so expensive computationally, the Dantzig–Wolfe decomposition method has generally not proven to be an efficient method for solving the multicommodity flow problem; nevertheless, Dantzig–Wolfe decomposition has one important advantage that distinguishes it from other Lagrangian-based algorithms: The Dantzig–Wolfe decomposition algorithm always maintains a feasible solution of the problem. Since, as we have shown in Section 17.5, the solution to the subproblems provides us with a lower bound on the optimal value of the problem, at each step we also have a bound on how far the current feasible solution is from optimal. Therefore, we can terminate the algorithm at any step not only with a feasible solution, but also with a guarantee of how far, in objective function value, that solution is from optimality.

17.7 RESOURCE-DIRECTIVE DECOMPOSITION

Lagrangian relaxation and Dantzig–Wolfe decomposition are price-directive methods that decompose the multicommodity flow problem into single-commodity network flow problems (or shortest path problems) by placing tolls or prices on the complicating bundle constraints. The resource-directive method that we consider in this section takes a different algorithmic approach. Instead of using prices to decompose the problem, it allocates the joint bundle capacity of each arc to the individual commodities. When applied to the problem formulation (17.1), the resource-directive approach allocates $r_{ij}^k \leq u_{ij}^k$ units of the bundle capacity u_{ij} of arc (i, j) to commodity k, producing the following *resource-directive problem*:

$$z = \min \sum_{1 \leq k \leq K} c^k x^k \tag{17.7a}$$

subject to

$$\sum_{1 \leq k \leq K} r_{ij}^k \leq u_{ij} \quad \text{for all } (i, j) \in A, \tag{17.7b}$$

$$\mathcal{N} x^k = b^k \quad \text{for } k = 1, 2, \ldots, K, \tag{17.7c}$$

$$0 \leq x_{ij}^k \leq r_{ij}^k \quad \text{for all } (i, j) \in A \text{ and all } k = 1, 2, \ldots, K. \tag{17.7d}$$

Note that the constraint (17.7b) ensures that the total resource allocation for arc (i, j) does not exceed that arc's bundle capacity. Let $r = (r_{ij}^k)$ denote the vector of resource allocations.

We now make a few elementary observations about this problem.

Property 17.2. *The resource-directive problem* (17.7) *is equivalent to the original multicommodity flow problem* (17.1) *in the sense that* (1) *if* (x, r) *is feasible in the resource-directive problem, then* x *is feasible and has the same objective function value in the original problem, and* (2) *if* x *is feasible in the original problem and we set* $r = x$, *then* (x, r) *is feasible and has the same objective function value in the resource-directive problem.*

Now consider the following sequential approach for solving the resource-directive problem (17.7). Instead of solving the problem by choosing the vectors r and x simultaneously, let us choose them sequentially. We first fix the resource allocations r_{ij}^k and then choose the flow x_{ij}^k. That is, let $z(r)$ denote the optimal value of the resource-directive problem for a fixed value of the resource allocation r and consider the following derived *resource-allocation problem*:

$$\text{Minimize} \quad z(r) \tag{17.8a}$$

subject to

$$\sum_{1 \leq k \leq K} r_{ij}^k \leq u_{ij} \quad \text{for all } (i, j) \in A, \tag{17.8b}$$

$$0 \leq r_{ij}^k \leq u_{ij}^k \quad \text{for all } (i, j) \in A \text{ and all } k = 1, 2, \ldots, K. \tag{17.8c}$$

The objective function $z(r)$ for this problem is complicated. We know its value only implicitly as the solution of an optimization problem in the flow variables x_{ij}^k.

Moreover, note that for any fixed value of the resource variables r_{ij}^k, the resource-directive problem decomposes into a separate network flow subproblem for each commodity. That is, $z(r) = \sum_{k \in K} z^k(r^k)$ with the value $z^k(r^k)$ of the kth subproblem given by

$$z^k(r^k) = \min \sum_{1 \le k \le K} c^k x^k \tag{17.9a}$$

subject to

$$\mathcal{N} x^k = b^k \qquad \text{for all } k = 1, \ldots, K, \tag{17.9b}$$

$$0 \le x_{ij}^k \le r_{ij}^k \qquad \text{for all } (i, j) \in A \text{ and for all } k = 1, 2, \ldots, K. \tag{17.9c}$$

Property 17.3. *The resource-directive problem* (17.7) *is equivalent to the resource-allocation problem* (17.8) *in the sense that* (1) *if* (x, r) *is feasible in the resource-directive problem, then* r *is feasible in the resource-allocation problem and* $z(r) \le cx$, *and* (2) *if* r *is feasible in the resource-allocation problem, then for some vector* x, (x, r) *is feasible in the original problem and* $cx = z(r)$.

Proof. If $(x, r) = (x^1, x^2, \ldots, x^K, r^1, r^2, \ldots, r^K)$ is feasible in the resource-directive problem, then r is feasible in the resource-allocation problem. Moreover, x^k is feasible in the kth commodity subproblem (17.9), so $z^k(r^k) \le c^k x^k$. Therefore, $z(r) = \sum_{1 \le k \le K} z^k(r^k) \le \sum_{1 \le k \le K} c^k x^k = cx$. Conversely, if r is feasible in the resource-allocation problem, then by definition of $z^k(r^k)$, $z^k(r^k) = c^k x^k$ for some vector x^k, so $x = (x^1, x^2, \ldots, x^K)$ satisfies the condition $cx = z(r)$. ◆

Let us pause to consider the implication of Properties 17.2 and 17.3. They imply that rather than solving the multicommodity flow problem directly, we can decompose it into a resource-allocation problem with a very simple constraint structure with a single inequality constraint but with a complex objective function $z(r)$. Although the overall structure of the objective function is complicated, it is easy to evaluate: To find its value for any choice of the resource-allocation vector r, we need merely solve K single-commodity flow problems.

Another way to view the objective function $z(r)$ is as the cost of the linear program (17.7) as a function of the right-hand-side parameters r. That is, any value r for the allocation vector defines the values of right-hand-side parameters for this linear program. A well-known result in linear program shows us that the function has a special form. We state this result for a general linear programming problem that contains the multicommodity flow problem as a special case.

Property 17.4. *Let* R *denote the set of allocations for which the linear program minimize* $\{cx : \mathcal{A}x = b, 0 \le x \le r\}$ *is feasible. Let* $z(r)$ *denote the value of this linear program as a function of the right-hand-side parameter* r. *On the set* R, *the objective function* $z(r)$ *is a piecewise linear convex function of* r.

Proof. To establish convexity of $z(r)$, we need to show that if $\bar{r}$ and $\hat{r}$ are any two values of the parameter r for which the given linear program is feasible and θ is any scalar, $0 \le \theta \le 1$, then $z(\theta\bar{r} + (1 - \theta)\hat{r}) \le \theta z(\bar{r}) + (1 - \theta)z(\hat{r})$. Let $\bar{y}$ and $\hat{y}$ be optimal solutions to the linear program for the parameter choices $r = \bar{r}$ and $r =$

$\hat{r}$. Note that $\mathcal{A}\bar{y} = b$, $\mathcal{A}\hat{y} = b$, $\bar{y} \leq \bar{r}$, and $\hat{y} \leq \hat{r}$. But then $\mathcal{A}(\theta\bar{y} + (1 - \theta)\hat{y}) = b$ and $\theta\bar{y} + (1 - \theta)\hat{y} \leq \theta\bar{r} + (1 - \theta)\hat{r}$. Therefore, the vector $\theta\bar{y} + (1 - \theta)\hat{y}$ is feasible for the linear program with parameter vector $r = \theta\bar{r} + (1 - \theta)\hat{r}$, so the optimal objective function value for this problem is at most $c(\theta\bar{y} + (1 - \theta)\hat{y})$. Moreover, by our choice of $\bar{y}$ and $\hat{y}$, $z(\bar{r}) = c\bar{y}$ and $z(\hat{r}) = c\hat{y}$; therefore,

$$z(\theta\bar{r} + (1 - \theta)\hat{r}) \leq \theta c\bar{y} + (1 - \theta)c\hat{y} = \theta z(\bar{r}) + (1 - \theta)z(\hat{r}),$$

so $z(r)$ is a convex function.

The piecewise linearity of $z(r)$ follows from the optimal basis property of linear programs. That is, for any choice of the parameter r, the problem has a basic feasible optimal solution, and this basic feasible solution remains optimal for all values of r for which it remains feasible. Moreover, the objective function value of the linear program is linear in r for any given (optimal) basis. ◆

Solving the Resource-Directive Models

A number of algorithmic approaches are available for solving the resource-directive models that we have introduced in this section. In the discussion to follow, we outline a few basic approaches. The references cited in the reference notes contain further details about these methods.

Since the function $z(r)$ is nondifferentiable (because it is piecewise linear), we cannot use gradient methods from nonlinear programming to solve the resource-allocation problem. We could, instead, use several other approaches. For example, we could search for local improvement in $z(r)$ using a heuristic method. As one such possibility, we could use an "arc-at-a-time approach" by adding 1 to $r_{pq}^{k'}$ and subtracting 1 from $r_{pq}^{k''}$ for some arc (p, q) for two commodities k' and k'', choosing the arc and commodities at each step using some criterion (e.g., the choices that give the greatest decrease in the objective function value at each step). This approach is easy to implement but does not ensure convergence to an optimal solution. Note that we can view this approach as changing the resource allocation at each step using the formula

$$r \leftarrow r + \theta\gamma,$$

with a step length of $\theta = 1$ and a movement direction $\gamma = (\gamma_{ij}^k)$ given by $\gamma_{pq}^{k'} = 1$, $\gamma_{pq}^{k''} = -1$, and $\gamma_{ij}^k = 0$ for all other arc–commodity combinations. Borrowing ideas from subgradient optimization, however, we could use an optimization approach by choosing the movement direction γ as a subgradient corresponding to the resource allocation r (see Figure 17.10). A natural approach would be to search for a subgradient, or movement direction γ, and a step length θ that simultaneously maintain feasibility and ensure convergence to an optimal solution r of the resource-allocation problem (17.8). We will adopt a two-step approach. First we determine a subgradient direction and step length that would ensure convergence, provided that we had no constraints imposed on the allocations. If moving to $r + \theta\gamma$ gives us an infeasible solution, we transform r to a point r' that would maintain feasibility, yet ensure convergence. To apply this approach, we need to be able to answer two basic questions: (1) How can we find a subgradient γ of $z(r)$ at any given point r? (2) How could we transform any nonfeasible resource allocation $r + \theta\gamma$ so that it

becomes feasible for the resource-choice problem, that is, it satisfies the constraints $\sum_{1 \le k \le K} r_{ij}^k \le u_{ij}$ for all $(i, j) \in A$, and $0 \le r_{ij}^k \le u_{ij}^k$ for all $(i, j) \in A$ and all $k = 1, 2, \ldots, K$? (Notice that when we applied subgradient optimization to the Lagrangian multiplier problem in Chapter 15, the Lagrange multipliers were either unconstrained or constrained to be nonnegative, so we either used the update formula $r \leftarrow r + \theta\gamma$ directly or used a modest modification of it to ensure that the Lagrange multipliers remained nonnegative. Therefore, in the context of Lagrangian relaxation, feasibility was easy to ensure.)

To answer 1, recall that as shown in Figure 17.10, a subgradient γ of $z(r)$ at the point $r = \bar{r}$ is any vector satisfying the condition

$$z(r) \ge z(\bar{r}) + \gamma(r - \bar{r}) \qquad \text{for all } r = (r^1, r^2, \ldots, r^K) \text{ with } r^k \in R^k.$$

In this expression, R^k is the set of resource allocations for commodity k for which the subproblem (17.8) is feasible. The following property shows that to find any such subgradient, we can work with each of the constituent functions $z^k(r^k)$ independently.

Property 17.5. *Let γ^k for $k = 1, 2, \ldots, K$ be a subgradient of $z^k(r^k)$ at the point $\bar{r}^K$. Then $\gamma = (\gamma^1, \gamma^2, \ldots, \gamma^k)$ is a subgradient of $z(r)$ at $\bar{r} = (\bar{r}^1, \bar{r}^2, \ldots, \bar{r}^K)$.*

Proof. Since γ^k is a subgradient of $z^k(r^k)$,

$$z(r^k) \ge z(\bar{r}^k) + \gamma^k(r^k - \bar{r}^k) \qquad \text{for all } r^k \in R^k.$$

Adding these expressions for $k = 1, 2, \ldots, K$ and using the fact that $z(r) = \sum_{k \in K} z^k(r^k)$ and $\gamma(r - \bar{r}) = \sum_{k \in K} \gamma^k(r^k - \bar{r}^k)$ gives us the desired result. ◆

This result shows us that to obtain a subgradient of $z(r)$, we can find a subgradient of each function $z^k(r^k)$, which requires information about the sensitivity of the solution of a network flow problem (17.9) to changes in the upper bounds r_{ij}^k on the arc flows. Fortunately, as shown by the following result, this information will be a by-product of almost any solution procedure for solving the subproblem (17.9). Since this result is general and applies to broader application contexts, we state it for a general network flow problem (i.e., we subsume the index k).

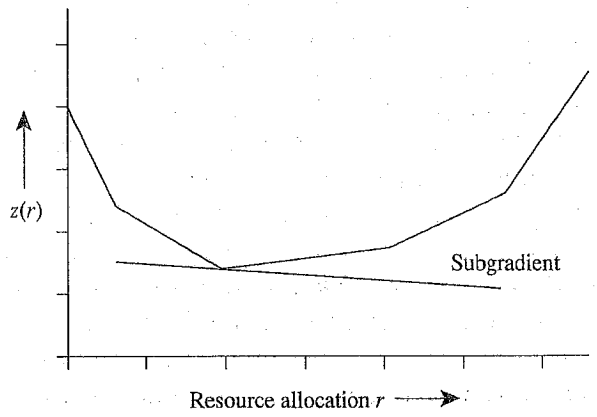


Figure 17.10 Subgradient for the function $z(r)$.

Property 17.6. *Consider the network flow problem* $z(q) = \min\{cx : \mathcal{N}x = b, 0 \le x \le q\}$, *with a parametric vector* q *of nonnegative upper bounds on the arc flows. Suppose that* x^* *is an optimal solution to this problem when* $q = q^*$, *and that* x^* *together with the reduced cost vector* c^π *satisfy the minimum cost flow optimality conditions* (9.8). *Define the vectors* $\mu = (\mu_{ij})$ *by setting* $\mu_{ij} = 0$ *if* $x^*_{ij} < q^*_{ij}$ *and* $\mu_{ij} = c^\pi_{ij}$ *if* $x^*_{ij} = q^*_{ij}$. *Then for any nonnegative vector* q' *for which the problem is feasible,* $z(q') \ge z(q^*) + \mu(q' - q^*)$. *That is,* μ *is a subgradient of the parametric objective function* $z(\mu)$ *at* $\mu = \mu^*$.

Proof. Recall that the definition of c^π implies that for any feasible flow vector x, $cx = c^\pi x - \pi b$. Therefore, $z(q^*) = cx^* = c^\pi x^* - \pi b = \mu q^* - \pi b$. The last equality in this expression follows from the optimality conditions (e.g., $c^\pi_{ij} = 0$ if $0 < x^*_{ij} < q^*_{ij}$) and the definition of μ. Let x' solve the parametric problem when $q = q'$ and note that $z(q') = cx' = c^\pi x' - \pi b \ge \mu q' - \pi b$. (The inequality in this expression follows from the fact that $c^\pi \le \mu$ and $0 \le x \le q'$. Further, $c^\pi_{ij} > \mu_{ij}$ is possible when $x^*_{ij} = 0$.) By combining the expressions for $z(q^*)$ and $z(q')$, we see that $z(q') \ge z(q^*) + \mu(q' - q^*)$. ◆

To apply Property 17.6, we solve each subproblem (17.9). After we solve the subproblem for the kth commodity with $q = r^k$, we set γ^k equal to the vector μ. We then use Property 17.6 and set $\gamma = (\gamma^1, \gamma^2, \ldots, \gamma^K)$. These results show us how to use any minimum cost flow algorithm that produces reduced costs c^π as well as an optimal flow vector to find a subgradient.

Once we have obtained a subgradient γ of $z(r)$, we could use the subgradient formula $r \leftarrow r + \theta\gamma$ to find the new resource allocation r. If the resulting resource-allocation vector r is feasible (i.e., satisfies the constraints $\sum_{1 \le k \le K} r^k_{ij} \le u_{ij}$), we move to this point. If the new resource allocation r is not feasible, however, we need to modify it to ensure feasibility, moving instead to some other point $\bar{r}$. One approach that ensures that the algorithm converges with the appropriate choice of step sizes (as discussed in Chapter 16) is to choose $\bar{r}$ as the feasible point that is as close as possible to r in the sense of minimizing the quantity $\sum_{1 \le k \le K} \sum_{(i,j) \in A} (r^k_{ij} - \bar{r}^k_{ij})^2$. The references cited at the end of this chapter show how to carry out this computation efficiently and show that with this choice of $\bar{r}$ the algorithm still converges.

17.8 BASIS PARTITIONING

Our development in Chapter 11 built on a principal advantage of the simplex method for the (single-commodity) minimum cost flow problem; namely, the algorithm can use the spanning tree structure of the linear programming basis to accelerate the required matrix computations. The multicommodity flow problem is a linear program that combines two sets of constraints: (1) independent mass conservation constraints $\mathcal{N}x^k = b^k$ for each commodity $k = 1, 2, \ldots, K$, and (2) bundle constraints that link the commodities through shared arc capacities. Because of this underlying network substructure, we might ask whether we could also use some form of spanning tree structure to implement the simplex method for the multicommodity flow problem. The basis partitioning algorithm that we consider in this section provides at least a partially affirmative answer to this question. As we will see, the method

combines network methodology together with more generic ideas from linear programming.

This discussion is intended as an introduction to the basis partitioning method, not as a detailed account of the method, which is rather involved; instead, we present just the essential underlying ideas with the aim of showing how we can exploit the underlying network structure. This discussion assumes familiarity with the network simplex method (described in Chapter 11) and with the basic concepts of linear programming (described in Appendix C).

Consider any basis $\mathcal{B}$ of the linear programming formulation (17.1) of the multicommodity flow problem. Since each column of $\mathcal{B}$ corresponds to the flow on an arc of some commodity, the columns in the basis define a set of subgraphs, one for each commodity. Let $\mathcal{M}^k$ denote the submatrix of $\mathcal{B}$ corresponding to commodity k. We note that each matrix $\mathcal{M}^k$ must contain a basis $\mathcal{B}^k$ of the node–arc incidence matrix $\mathcal{N}^k$ (see Exercise 17.19). Our results in Section 11.11 show that each such basis corresponds to a spanning tree T^k for the corresponding commodity k. Therefore, the subgraphs corresponding to any basis are spanning trees together with some additional arcs. Moreover, since the constraints of the multicommodity flow problem have a block angular form, so does the basis $\mathcal{B}$. Figure 17.11(a) shows the structure of the basis $\mathcal{B}$.

Rather than working directly with the basis matrix $\mathcal{B}$, through a change in variables, we will transform this matrix into another matrix $\mathcal{B}'$ so that we can perform the steps of the simplex method much more efficiently using the underlying network structure of the problem. Suppose that we state the bundle constraints as equality constraints [see (17.1b′)] by adding the nonnegative slack variable s_{ij} to the bundle constraint for arc (i, j). The variable s_{ij} denotes the amount of unused bundle capacity on arc (i, j). Consider the system of mass balance and bundle constraints for a basis of the multicommodity flow problem, which we write as $\mathcal{B}x_{\mathbf{B}} = b$. In this expression, $b = (u, b^1, b^2, \ldots, b^K)$ is the vector of right-hand-side coefficients for the equations and $x_{\mathbf{B}}$ is the set of variables corresponding to the basic variables. These variables include both flow variables x_{ij}^k and slack variables s_{ij} for the bundle constraints.

To simplify the system $\mathcal{B}x_{\mathbf{B}} = b$, suppose that we make a change in variables by letting $\mathcal{D}y_{\mathbf{B}} = x_{\mathbf{B}}$ for some appropriately selected nonsingular matrix $\mathcal{D}$. If we

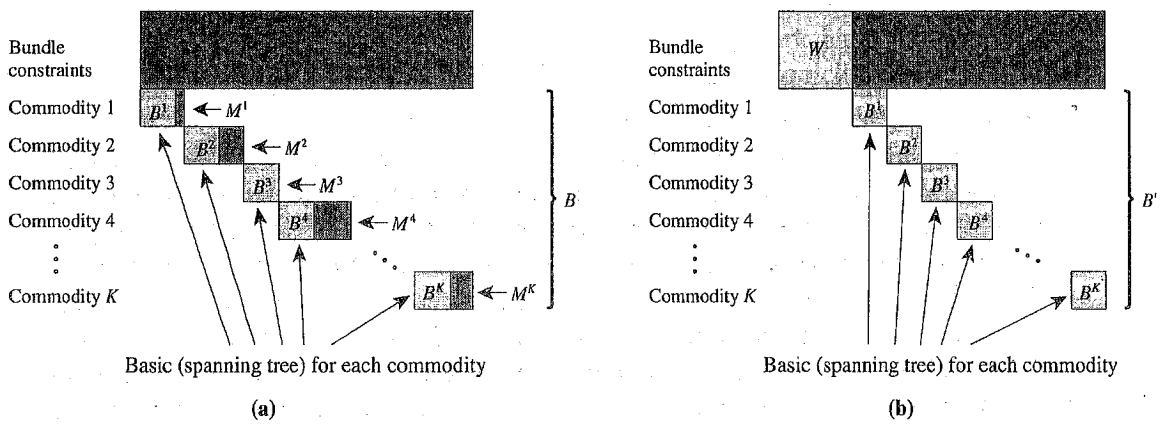


Figure 17.11 Partitioning a linear programming basis of the multicommodity flow problem: (a) original basis; (b) basis after change in variables and rearrangement of columns.

substitute $\mathcal{D}y_{\mathbf{B}}$ for $x_{\mathbf{B}}$ in the system $\mathcal{B}x_{\mathbf{B}} = b$, and let $\mathcal{B}' = \mathcal{B}\mathcal{D}$, the system becomes $\mathcal{B}(\mathcal{D}y_{\mathbf{B}}) = (\mathcal{B}\mathcal{D})y_{\mathbf{B}} = \mathcal{B}'y_{\mathbf{B}} = b$. Suppose that we can make this change in variables so that the matrix $\mathcal{B}'$ has the special structure shown in Figure 17.11(b); in this special structure, only the y variables that appear in the mass balance constraints for commodity k are those corresponding to the basis $\mathcal{B}^k$. Later in this section we show how to use the underlying network structure of the problem to make this change in variables. For the moment, so that we can obtain a broad view of the basic approach without the complicating details, let us assume that we can easily make this transformation of variables.

In the transformed system, let $\mathcal{W}$ denote the matrix from the bundle constraints that do not correspond to any of the columns in the bases $\mathcal{B}^k$ for $k = 1, 2, \ldots, K$. We refer to $\mathcal{W}$ as the *working basis*. In the pictorial representation of the basis $\mathcal{B}'$ in Figure 17.11(b), we have rearranged the columns of the basis so that those in the working basis appear first.

We now make two further observations:

1. Since a change in variables, which corresponds to column operations on the matrix $\mathcal{B}$, does not change the nonsingularity of this matrix, the matrix $\mathcal{B}'$ is also nonsingular. This fact implies that the working basis $\mathcal{W}$ is nonsingular.
2. Since column operations on the matrix $\mathcal{B}$ do not change the solution of the system $\pi\mathcal{B} = c_{\mathbf{B}}$, the vector of the simplex multipliers determined by the system $\pi\mathcal{B}' = c_{\mathbf{B}'}$ is the same as those determined by the original system $\pi\mathcal{B} = c_{\mathbf{B}}$. (When we perform the column operations, we change both $\mathcal{B}$ and $c_{\mathbf{B}}$; $c_{\mathbf{B}'}$ denotes the result of the column operations on the costs $c_{\mathbf{B}}$.)

These two observations and our change in variables gives us all the necessary ingredients for implementing the basis partitioning simplex method for the multicommodity flow problem. Recall that the simplex method requires that we make two types of matrix computations:

1. Solving a system of the form $\mathcal{B}x_{\mathbf{B}} = b$ or $\mathcal{B}x_{\mathbf{B}} = \mathcal{A}_j$ for some column $\mathcal{A}_j$ of the coefficient matrix $\mathcal{A}$ of the linear programming model (we make this computation at each step before we introduce column $\mathcal{A}_j$ into the basis).
2. Solving a system of the form $\pi\mathcal{B} = c_{\mathbf{B}}$ to find the vector π of simplex multipliers.

Rather than making these computations of the matrix $\mathcal{B}$ directly, we will use the matrix $\mathcal{B}'$. As we have already noted, the systems $\pi\mathcal{B} = c_{\mathbf{B}}$ and $\pi\mathcal{B}' = c_{\mathbf{B}'}$ have the same solutions. Suppose that we partition the vector of simplex multipliers as $\pi = (\pi^0, \pi^1, \ldots, \pi^K)$. The vector π^0 contains the simplex multipliers for the bundle constraints and the vector π^k contains the simplex multipliers corresponding to the mass balance constraints $\mathcal{N}x^k = b^k$ for commodity k. Let $c_{\mathbf{W}}$ denote the subvector of $c_{\mathbf{B}}$ corresponding to the columns in $\mathcal{W}$. We solve the system $\pi\mathcal{B}' = c_{\mathbf{B}'}$ efficiently by the following *forward substitution* procedure. We first solve the subsystem $\pi^0\mathcal{W} = c_{\mathbf{W}}$ by solving a system of linear equations. If we substitute these values into the system $\pi\mathcal{B}' = c_{\mathbf{B}'}$, the system decomposes into separate subsystems for each commodity k with modified coefficients $\hat{c}_{\mathbf{B}'}$ for $c_{\mathbf{B}'}$. Moreover, the computations for each subsystem are exactly those required to find the simplex multipliers for a single-commodity network flow problem. Therefore, we can use the very efficient

procedure for computing the simplex multipliers that we developed in Chapter 11. Recall that these computations are very efficient because they use the tree structure of the basis.

To solve a system of the form $\mathcal{B}x_{\mathbf{B}} = b$, we first solve the related system $\mathcal{B}'y_{\mathbf{B}} = b$ using forward substitution. That is, if we partition the vector $y_{\mathbf{B}}$ as $y_{\mathbf{B}} = (y^0, y^1, y^2, \ldots, y^K)$ corresponding to the columns $\mathcal{W}, \mathcal{B}^1, \mathcal{B}^2, \ldots, \mathcal{B}^K$ of $\mathcal{B}'$, we first solve the independent subsystems $\mathcal{B}^k y^k = b^k$. Notice that we can perform these computations by using the efficient procedure for computing the arc flows that we developed in Chapter 11. Recall that this procedure also used the spanning tree structure corresponding to the basis $\mathcal{B}^k$. We then substitute the values of the variables $y^1, y^2, \ldots, y^K$ into the system $\mathcal{B}'y = b$, leaving a reduced system of the form $\mathcal{W}y^0 = u'$. (Here u' denotes the modified arc capacities we obtain after substituting the values of the flow variables $y^1, y^2, \ldots, y^K$ in the transformed bundle constraints.) After solving this system of linear equations, we use the transformation between $x_{\mathbf{B}}$ and $y_{\mathbf{B}}$ (i.e., $x_{\mathbf{B}} = \mathcal{D}y_{\mathbf{B}}$) to find the values for the variables $x_{\mathbf{B}}$.

Notice what this approach has accomplished. Suppose that we apply the procedure to a problem with p bundle constraints (p might equal the number m of arcs if they all have bundle capacities, or it could be much less than m). Rather than solving systems of the form $\mathcal{B}x = b$ on the full matrix $\mathcal{B}$ which has dimension $p + K$ by $p + K$, we make K tree computations and solve one smaller $p \times p$ system $\mathcal{W}y^0 = u'$. Similarly, rather than solving a large $(p + K) \times (p + K)$ system of the form $\pi\mathcal{B} = c_{\mathbf{B}}$ solve a single $p \times p$ system $\pi^0\mathcal{W} = c_{\mathcal{W}}$ and make K tree computations. The resulting speed up in computations can be very substantial.

The rest of the steps of the simplex method (e.g., pricing out columns to find the entering variable at each step or performing the ratio test to find the outgoing variable) are exactly those of the method in general. Since our purpose in this discussion is not to describe the simplex method, but rather to show how we can exploit the underlying network structure to accelerate the computations, we do not describe these details. There is one other important feature of the algorithm that we do not discuss, namely how to recompute the matrix $\mathcal{B}'$ when we exchange one column in the basis for another. The references cited at the end of this chapter show how to perform these computations.

To complete this brief description of the basis partitioning method, let us show how to make the change in variables $x_{\mathbf{B}} = \mathcal{D}y_{\mathbf{B}}$ needed to eliminate any column $\mathcal{M}_{ij}^k$ that belongs to $\mathcal{M}^k$ but not to $\mathcal{B}^k$ from the mass balance equations for commodity k so that the only remaining variables in these equations will be those corresponding to the basis $\mathcal{B}^k$.

Consider the following example, which corresponds to the system $\mathcal{M}^k x^k = b^k$ (for simplicity, we subsume the commodity index k and use x_{ij} in place of x_{ij}^k):

$$
\begin{array}{c}
\textit{Node} \\
1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6
\end{array}
\begin{bmatrix}
+1 & +1 & & & & \\
-1 & & & & & \\
 & -1 & +1 & +1 & & \\
 & & -1 & & & +1 \\
 & & & -1 & -1 & \\
 & & & & +1 & -1
\end{bmatrix}
\begin{bmatrix} x_{12} \\ x_{13} \\ x_{34} \\ x_{35} \\ x_{65} \\ x_{46} \end{bmatrix}
=
\begin{bmatrix} 4 \\ 2 \\ 1 \\ 2 \\ 4 \\ 3 \end{bmatrix}
$$

Let $\mathcal{M}_{ij}$ denote the column of this system corresponding to the variable x_{ij}. The columns $\mathcal{M}_{12}$, $\mathcal{M}_{13}$, $\mathcal{M}_{34}$, $\mathcal{M}_{35}$, and $\mathcal{M}_{65}$ form a basis for this system and $\mathcal{M}_{46}$ is the nonbasic column that we would like to eliminate. Notice that $\mathcal{M}_{46} = -\mathcal{M}_{34} + \mathcal{M}_{35} - \mathcal{M}_{65}$, which in terms of the underlying network (see Figure 17.12) gives the representation of arc (4, 6) in terms of the spanning tree defined by the arcs (1, 2), (1, 3), (3, 4), (3, 5), and (6, 5). Now suppose that we make the change in variables $x_{34} = y_{34} - y_{46}$, $x_{35} = y_{35} + y_{46}$, $x_{65} = y_{65} - y_{46}$, $x_{12} = y_{12}$, $x_{13} = y_{13}$, and $x_{46} = y_{46}$. Then we find that

$$\begin{aligned}
&\mathcal{M}_{12}x_{12} + \mathcal{M}_{13}x_{13} + \mathcal{M}_{34}x_{34} + \mathcal{M}_{35}x_{35} + \mathcal{M}_{65}x_{65} + \mathcal{M}_{46}x_{46} \\
&\quad = \mathcal{M}_{12}y_{12} + \mathcal{M}_{13}y_{13} + \mathcal{M}_{34}(y_{34} - y_{46}) + \mathcal{M}_{35}(y_{35} + y_{46}) \\
&\qquad + \mathcal{M}_{65}(y_{65} - y_{46}) + \mathcal{M}_{46}y_{46} \\
&\quad = \mathcal{M}_{12}y_{12} + \mathcal{M}_{13}y_{13} + \mathcal{M}_{34}y_{34} + \mathcal{M}_{35}y_{35} + \mathcal{M}_{65}y_{65}.
\end{aligned}$$

That is, we have eliminated the column $\mathcal{M}_{46}$ from the system. Notice that the new variables y_{ij} have a natural interpretation. The new variables y_{34}, y_{35}, and y_{65} measure both flows x_{ij} on these arcs as well as the flow x_{46} "diverted" from the arc (4, 6) to these arcs on the path 4–3–5–6.

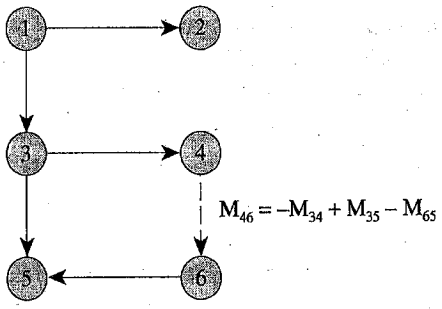


Figure 17.12 Representing nontree (nonbasic) arc for variable transformation.

In general, if we have several columns $\mathcal{M}_{ij}$ that we want to eliminate from the system, and in the underlying graph the arc (i, j) forms a unique cycle when added to the spanning tree T corresponding to the basis of the system, we let γ_{ij}^{pq} be either ± 1 or 0, depending on whether or not the arc (p, q) of the tree belongs to this cycle (the sign of each ± 1 depends upon the orientation of the tree arc (p, q) relative to the arc (i, j)). To transform the variables, we then set $x_{pq} = y_{pq} - \sum \gamma_{ij}^{pq} y_{pq}$ with the sum taken over the indices (i, j) for all the columns $\mathcal{M}_{ij}$ that we wish to eliminate. We also set $x_{ij} = y_{ij}$ for all arcs (i, j) not in the tree T.

To implement this procedure for the multicommodity flow problem, we apply these network computations and change of variables for each of the matrices $\mathcal{M}^k$. These computations give us the matrix $\mathcal{B}'$ and then we apply the operations discussed previously in this section.

17.9 SUMMARY

Multicommodity flow problems arise whenever several network flow problems share the same underlying network. This chapter has focused on a generic multicommodity flow problem with fixed bundle capacities imposed on the total flow on any arc by

all the commodities. As we have seen in this chapter, in Chapter 1, and as we show further in Chapter 19, this model arises in a wide variety of application settings in communications, logistics, manufacturing, and transportation as well as in such problem domains as urban housing and foodgrain export-import. The solution methods for solving the multicommodity flow problem generally attempt to exploit the network flow structure of the individual-constitutent single-commodity flow problems. Lagrangian relaxation removes the complicating bundle constraints by applying Lagrangian multipliers to them and bringing them into the objective function. The resulting Lagrangian subproblem separates into independent flow problems for each commodity. This method is price directive in that rather than considering the bundle constraints directly, it places tolls, or Lagrange multipliers on them. By the theory of Lagrangian relaxation, for any choice of the tolls, the Lagrangian subproblems give us a lower bound on the optimal objective function value of the problem. Moreover, because the multicommodity flow problem is a linear program, the best possible (i.e., largest) lower bound equals the optimal objective function value for the problem. Therefore, by solving the Lagrangian multiplier problem of finding the best Lagrangian lower bound, we find the optimal objective function value for the multicommodity flow problem.

The column generation procedure solves the multicommodity flow problem formulated in the space of path (and cycle) flows. This formulation is a large-scale linear program with an enormous number of variables (columns) but a rather simple constraint structure. The column generation procedure solves the problem by explicitly maintaining a relatively small number of these variables as part of a linear programming basis and then pricing-out the remaining columns to determine a column(s) that we could profitably add to the basis. The key to this procedure is the observation that to price-out the columns and to find the most profitable column to add to the basis (in terms of reduced cost for each commodity), we need not consider each column explicitly but can instead solve a shortest path problem for each commodity. Since this procedure is just a special (and efficient) implementation of the simplex method of linear programming, it inherits the finite convergence properties of the simplex method (with provisions for handling linear programming degeneracy). Moreover, since we can view the pricing-out operation as solving a Lagrangian relaxation of the problem, at each step we have not only a feasible solution, but also a lower bound providing a guarantee how far the solution is from optimality in the objective function value.

We might also interpret the column generation procedure as a price-setting algorithm with the basis for the linear program determining prices on the bundle capacities, and broadcasting these prices to shortest path subproblems, one for each commodity. When interpreted in this way, this so-called Dantzig–Wolfe decomposition procedure is an alternative solution procedure for solving the Lagrangian relaxation of the problem. In this approach we use a linear program, rather than simply move in a subgradient direction, to update the values of the Lagrangian multipliers at each step.

In our discussion of column generation and Dantzig–Wolfe decomposition, we have assumed that each commodity in the multicommodity flow problem has a single source and single sink. These algorithms apply to multicommodity flow problems with multiple sources and sinks for each commodity (see Exercise 17.14) and to more

general problems with otherwise independent subproblems coupled by joint resource constraints (see Exercises 17.37 and 17.38).

Resource-directive decomposition is an alternative conceptual approach for solving the multicommodity flow problem. Rather than removing the complicating bundle capacities by charging tolls for them, this approach decomposes the problem into a separate single commodity flow problem for each commodity by allocating the scarce bundle capacities to the various commodities. Finding the optimal allocation (i.e., the one that gives the overall lowest cost) is an optimization problem with a simple constraint structure and with a (complicated) convex cost objective function. Using sensitivity information about the single-commodity subproblems, however, we can generate subgradient information about the resource-allocation cost function and therefore we can solve the allocation problem by a version of the subgradient optimization technique.

One approach for solving the minimum cost flow problem is the simplex method implemented to exploit the underlying network flow structure, particularly the spanning tree interpretation of any linear programming basis. The basis partitioning method for solving the multicommodity flow problem is a generalization of this approach. It uses the fact that any linear programming basis for the multicommodity flow problem contains a basis for each commodity. Consequently, through a change of variables, we can transform any basis of the problem into a basis for each commodity as well as a small working basis for the bundle constraints that contains information about the interactions between the individual commodities. This approach permits us to solve the multicommodity flow problem by combining the special network simplex approach for each commodity together with more general linear programming matrix computations as applied to the working basis.

REFERENCE NOTES

Researchers have proposed a number of basic approaches for solving the multicommodity flow problem. The three basic approaches, all based on exploiting network flow substructure, that we have considered in this chapter and selected references are (1) price-directive decomposition algorithm (Cremeans, Smith, and Tyndall [1970], Swoveland [1971], Chen and Dewald [1974], and Assad [1980b]), (2) resource-directive decomposition algorithm (Geoffrion and Graves [1974], Kennington and Shalaby [1977], and Assad [1980a]), and (3) basis partitioning (Graves and McBride [1976] and Kennington and Helgason [1980]). Ford and Fulkerson [1958b] and Tomlin [1966] first suggested the column generation approach. The first of these papers was the forerunner of the general Dantzig–Wolfe [1960] decomposition procedure of mathematical programming. The excellent survey papers by Assad [1978] and by Kennington [1978] describe all of these algorithms and several standard properties of multicommodity flow problems. The book by Kennington and Helgason [1980] and the doctoral dissertation by Schneur [1991] are other valuable references on this topic.

Most of the material discussed in this chapter is classical and dates from the 1960s and 1970s. Many of the standard properties of multicommodity flows (e.g., nonintegrality of optimal flows), including several of the examples we have used in

the text and exercises to illustrate these properties, are due to Fulkerson [1963], Hu [1963], and Sakarovitch [1973].

The column generation and decomposition methods that we have considered in this chapter extend to other situations as long as we can represent any solution to a problem as a convex combination of other particularly "simple solutions"; in the text we have used shortest paths as the simple solutions. For some applications we might use solutions to knapsack problems as the simple solutions, and in other cases, such as the general multicommodity flow problem with multiple sources and destinations for each commodity, the simple solutions might be spanning tree solutions (see Exercise 17.14).

Researchers have developed and tested several codes for multicommodity flow problems. Kennington [1978] and Ali et al. [1984] have described the results of some of these computational experiments. These results have suggested that price-directive and partitioning algorithms are the fastest algorithms for solving multicommodity flow problems. The best multicommodity flow codes are two to five times faster than a general-purpose linear programming code. Recent computational experience by Bixby [1991] in solving large-scale network flow problems with side constraints has shown that the simplex method with an advanced starting basis technique can be very effective computationally. The approach that we have suggested at the end of our discussion of Lagrangian relaxation in Section 17.4 is a variant of this approach.

Interior point algorithms provide another approach for solving multicommodity flow problems. Although these algorithms yield the only known polynomial-time bounds for these problems, an efficient and practical implementation of these algorithms is the subject of future research. The best time bound for the multicommodity flow problem is due to Vaidya [1989]. Tardos's [1986] algorithm can solve the multicommodity flow problem in strongly polynomial time.

Several researchers have recently suggested new algorithms for the multicommodity flow problem: Gersht and Shulman [1987], Barnhart [1988], Pinar and Zenios [1990], and Farvolden and Powell [1990]. Schneur [1991] studied scaling techniques for the multicommodity flow problem. Matsumoto, Nishizeki, and Saito [1986] gave a polynomial-time combinatorial algorithm for solving a multicommodity flow problem in s–t planar networks.

In this chapter we have presented several applications of multicommodity flow problems, adapted from the following papers:

1. Routing of multiple commodities (Golden [1975] and Crainic, Ferland, and Rousseau [1984])
2. Warehousing of seasonal products (Jewell [1957])
3. Multivehicle tanker scheduling problem (Bellmore, Bennington, and Lubore [1971])

We have presented three additional applications elsewhere in this book: (1) racial balancing of schools in Application 1.10 (Clarke and Surkis [1968]), (2) optimal deployment of resources in Exercise 17.1 (Kaplan [1973]), and (3) multiproduct multistage production-inventory planning in Application 19.23 (Evans [1977]). Other applications of multicommodity flows arise in (1) multicommodity distribution sys-

tem design (Geoffrion and Graves [1974]), (2) rail freight planning (Bodin, Golden, Schuster, and Rowing [1980] and Assad [1980a]), and (3) VLSI chip design (Korte [1988]).

EXERCISES

17.1. **Optimal deployment of resources** (Kaplan [1973]). Suppose that an organization requires varying levels of resources at q geographical locations. Suppose, further, that changes in economic, political, social, or environmental conditions have brought about a sudden demand for the resources. For example, natural disasters such as floods might create a need for various types of rescue equipment at various flood locations. Or, the resources might be relief supplies such as food or medicines. This exercise studies a model that minimizes the combined cost of transportation and unfulfilled demands. Let d_j^k denote the demand of resource k (in tons) at location j. The resources are available in limited quantities at p different locations; let a_i^k denote the amount of resource k available at location i. The cost of transporting 1 unit of the resource k from location i to location j is c_{ij}^k. Due to technological constraints, at most u_i tons of all resources can be transported from location i. As a result, it might not be possible to completely satisfy the demands for the resource. Let α_j^k denote the cost of unfulfilled demand of resource k at location j. Our objective is to identify the shipment of resources that minimizes the total of the transportation costs and the costs arising due to unfulfilled demands. Formulate this problem as a multicommodity flow problem.

17.2. Show how to incorporate transportation expenses incurred between plant–warehouse, warehouse–retailer, and plant–retailer combinations in the warehousing of seasonal products model that we considered in Application 17.2.

17.3. Show that the optimal solution of the multicommodity flow problem in Figure 17.4(a), as shown in Figure 17.4(b), satisfies both the arc optimality conditions (17.2) and the path optimality conditions (17.6).

17.4. State a generalization of the arc optimality conditions (17.2) for the multicommodity flow problem with upper bounds u_{ij}^k imposed on arc flows x_{ij}^k.

17.5. Write the dual of the path flow formulation of the multicommodity flow problem given in (17.5). Show that (17.6) represents the complementary slackness conditions of this primal–dual pair for the multicommodity flow problem.

17.6. The path formulation (17.5) of a multicommodity flow problem with $p \leq m$ bundle constraints and K commodities contains $p + K$ constraints, and therefore any linear programming basis for the problem will contain $p + K$ basic variables.

(a) Show that at least one basic variable must appear in each of the K demand constraints $\sum_{P \in \mathbf{P}^k} f(P) = d^k$; that is, in any basis at least one path must carry a positive flow for each commodity.

(b) Show that in any feasible basis, at most p commodities will send flow on two or more paths, so at least $K - p$ commodities will send flow on a single path.

17.7. The numerical example shown in Figure 17.13 has four commodities: these commodities have nodes 1 and 4, 5 and 8, 9 and 12, and 13 and 16 as their source and sink nodes. We wish to send 10 units from the source node to the sink node of each commodity. The arcs (2, 3), (6, 7), (10, 11), and (14, 15) all have a bundle capacity of 15 units. All the other arcs are uncapacitated. The per unit flow costs shown next to each arc are the same for each commodity. Starting with the Lagrange multiplier values $\mu_{23} = \mu_{67} = \mu_{10,11} = \mu_{13,16} = 0$, and assuming step lengths of $\theta_0 = 1$ and $\theta_k = 1/k$ for $k \geq 1$, perform the first five steps of the Lagrangian relaxation algorithm for this problem.

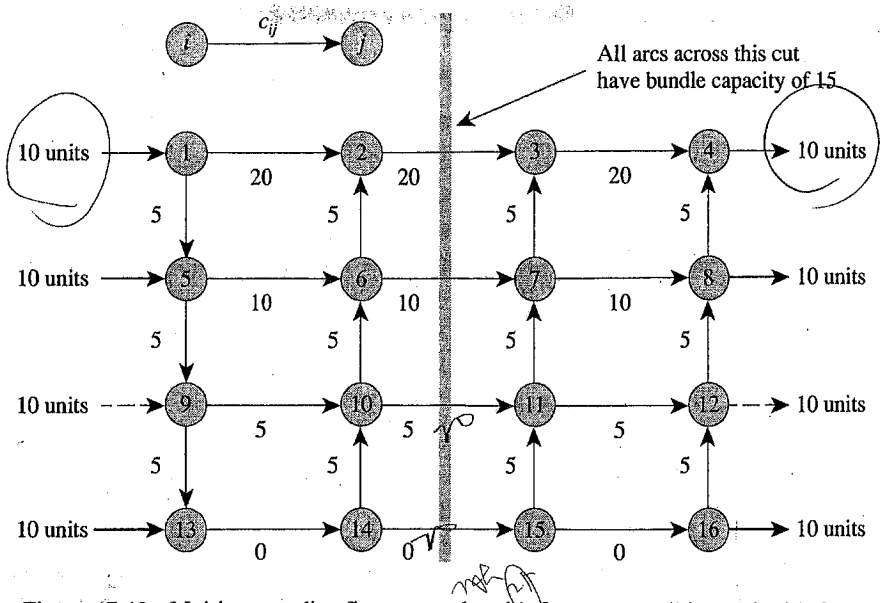


Figure 17.13 Multicommodity flow example with four commodities and with four arcs with bundle capacities.

17.8. Find an optimal solution of the problem introduced in Exercise 17.7 by any method, including visual inspection, and use the multicommodity flow optimality conditions to prove that your solution is optimal.

17.9. Since the multicommodity flow example shown in Figure 17.4(a) has two arcs with bundle capacities, the working basis in applying the basis partitioning algorithm to this problem will be a 2 × 2 matrix. Specify the optimal linear programming basis $\mathcal{B}$ corresponding to the optimal solution shown in Figure 17.4(b). Restricting your choice of variables to those in $\mathcal{B}$, specify a basis (spanning tree) for each commodity and the resulting working basis for $\mathcal{B}$. Show how to use the working basis and spanning tree to compute the simplex multipliers corresponding to the optimal linear programming basis.

17.10. Nonhomogeneous goods. Consider an extension of the multicommodity flow problem with nonhomogeneous goods. In this model, each unit x_{ij}^k of flow of commodity k on arc (i, j) consumes a given amount ρ_{ij}^k of the capacity (or some other resource) associated with the arc (i, j), and we replace the bundle constraint with a more general resource availability constraint $\sum_{1 \le k \le K} \rho_{ij}^k x_{ij}^k \le u_{ij}$.

(a) Show how to convert this model into the bundle constraint model (17.1) that we have considered in this chapter if each commodity consumes the same amount of resource on each arc, that is, for every arc (i, j), $\rho_{ij}^1 = \rho_{ij}^2 = \cdots = \rho_{ij}^K = \rho_{ij}$.

(b) Show how to solve the general version of this problem by using a modification of the following methods that we have considered in this chapter: (1) Lagrangian relaxation, and (2) column generation.

17.11. Modeling piecewise linear convex costs. Suppose that a multicommodity flow problem has no bundle constraints, but instead the cost of flow on each arc is a piecewise linear function of the total flow $x_{ij} = \sum_{1 \le k \le K} x_{ij}^k$ of all the commodities on that arc. Show how to model this problem as a (linear) multicommodity flow problem with bundle constraints. (*Hint*: See Section 14.3.)

17.12. Multicommodity circulation problem. Suppose that the flow of each commodity in a multicommodity flow problem must be a circulation (i.e., each supply/demand vector b^k is the zero vector). Also, assume that some of the arc costs might be negative. Show how to modify the column generation algorithm to solve this circulation variant

of the multicommodity flow problem. (*Hint*: It is always possible to express any circulation as the union of cycle flows. Also, recall that we can use the label correcting shortest path algorithm to detect a negative cost cycle.)

17.13. Show how to extend the column generation algorithm to handle situations with negative costs. In these situations, some potentially optimal feasible flows could be a weighted combination of both cycle and path flows, and the output of any shortest path subproblem might be a negative cost cycle.

17.14. Show how to apply column generation and Dantzig–Wolfe decomposition to multicommodity flow problems in which each commodity can have several sources and destinations. Assume that the cost vector for each commodity is nonnegative. For each commodity k, let $x^{k,q}$ for $q = 1, 2, \ldots, Q_k$ denote the flow vector corresponding to the qth of Q_k feasible spanning tree solution for the system $\mathcal{N}x^k = b^k$, $0 \leq x^k \leq u^k$. Use the fact that we can write any potentially optimal feasible solution x^k to this system as $x^k = \sum_{1 \leq k \leq K} \lambda^{k,q} x^{k,q}$ for some nonnegative weighting vectors $\lambda^{k,q}$ satisfying the condition $\sum_{1 \leq q \leq Q_k} \lambda^{k,q} = 1$. (Note that the nonnegativity of the costs implies that we need not consider cycles in the optimal solution.)

17.15. Multisink maximum flow problem (Rothfarb, Shein, and Frisch [1968])

(a) Consider a multicommodity maximum flow problem with a source node s and with two different sink nodes, t^1 and t^2. Suppose that every unit shipped from node s into node t^1 yields a profit of \$3, and that every unit shipped from node s into node t^2 yields a profit of \$2. Show how to find the maximum profit flow by solving two maximum flow problems. (*Hint*: First ship as much flow as possible from node s to node t^1, obtaining a maximum s–t^1 flow x'. Subsequently, ship as much flow as possible from s to t^2 in the residual network $G(x')$ while keeping the flow from node s to node t^2 constant. Show that the resulting flow x'' is optimal for the two-sink problem by showing that the residual network $G(x'')$ contains no augmenting path from node s to node t^1 and no augmenting path from node s to node t^2.)

(b) Generalize the results of part (a) to the following situation. A network $G = (N, A)$ has one source node s and K sink nodes $t^1, t^2, \ldots, t^K$, one for each of K commodities. For $k = 1$ to K, let c_k be the value of shipping each unit of commodity k to sink t^k. Assume that $c_1 \geq \cdots \geq c_K$. Show how to find the optimal-valued multicommodity flow. Justify your solution procedure. (*Hint*: Extending the results of part (a), solve the problem as a sequence of K maximum flow problems.)

17.16. Common-source or common-sink multicommodity flow problem

(a) The common-source multicommodity flow problem is a special case of the multicommodity flow problem (17.1) in which all commodities have a common source but (possibly) distinct sinks. Show how to solve the common-source multicommodity flow problem by solving one single-commodity minimum cost flow problem. (*Hint*: First solve a minimum cost flow problem and then use flow decomposition.)

(b) In the common-sink multicommodity flow problem, all commodities have a common sink but (possibly) distinct sources. Show how to solve the common-sink multicommodity flow problem by solving one single-commodity minimum cost flow problem.

17.17. Funnel problem. Let $G = (N, A)$ be a network with K source nodes $s^1, \ldots, s^K$ and J sink nodes $t^1, \ldots, t^J$. Suppose, further, that the network contains a *cut node* v whose deletion disconnects each source from each sink. That is, $G - v$ has two components, one containing all the source nodes and the other containing all the sink nodes. Each source node s^i and each sink node t^j defines a commodity and has an associated demand of d_{ij}; therefore, we must send d_{ij} units of flow from node s^i to node t^j. Suppose that each arc (p, q) has an associated per unit arc flow cost c_{pq} (which is the same for all the commodities) and an associated flow capacity u_{pq}. Show that by solving two single-commodity minimum cost flow problems, we can either

determine a minimum cost multicommodity flow in the network G or prove that none such flow exists. (*Hint*: Use ideas introduced in Exercise 17.16.)

17.18. Suppose that you had an algorithm that could find a feasible multicommodity flow for the problem (17.1). Further, suppose that you also had the optimal set of prices (tolls) for bundle constraints for the arcs. How might you use these prices and the solution algorithm to find an optimal flow for the multicommodity flow problem?

17.19. Let $\mathcal{M}^k$ denote the submatrix of a basis matrix $\mathcal{B}$ for the multicommodity flow model (17.1) corresponding to the flow equations $\mathcal{N}x^k = b^k$. Show that $\mathcal{M}^k$ must contain a basis $\mathcal{B}^k$ of the node–arc incident matrix $\mathcal{N}$. (*Hint*: Show that the arcs corresponding to the column in $\mathcal{M}^k$ must contain a spanning tree of the underlying network by recalling that if $\mathcal{B}$ is a basis matrix, we can solve the system of equations $\mathcal{B}x = d$ for any vector d.)

17.20. Cutting stock problem. The production of paper or cloth often uses the following process. We first produce the paper on long rolls which we then cut to meet specific demand requirements. The cutting stock problem is to meet the specified demand by using the minimum number of rolls. The cutting stock problem also arises in other guises as well. For example, it arises when we wish to store information with bit length L_i onto tracks in a computer disk, assuming that each track can hold L bits. Suppose that the rolls all have length L and that we have a demand of d_i for smaller rolls of size L_i. There might be many ways to cut any roll into smaller rolls. For example, if $L = 50$ and the L_i have lengths 10, 15, 25, 30, we could cut the rolls into several different patterns: for example, (1) 5 rolls of size 10; (2) 3 rolls of size 15; (3) 2 rolls of size 10 and 1 roll of size 25. Note that in the first pattern, we use the entire roll of size L, but in the other two patterns we incur a waste of 5. Let $\mathcal{P}$ denote the collection of all patterns, and for any pattern $P \in \mathcal{P}$, let $n_{i,P}$ denote the number of small rolls of size L_i in pattern P.

(a) Letting x_P be a nonnegative integer variable indicating how many rolls we cut into pattern P, show how to formulate the cutting stock problem of finding the fewest number of large rolls to meet the specified demands as an optimization model with a variable for each cutting pattern.

(b) Suppose that we wish to cut a single roll of size L into subrolls and that we receive a profit of π_i for every subroll of size L_i that we use in the solution. Show how to formulate this problem as an integer knapsack problem.

(c) Show how to use the column generation technique described in Section 17.5 to solve the linear programming relaxation of the cutting stock problem in part (a), using knapsack problems of the form stated in part (b) to price out columns.

17.21. Undirected multicommodity flow problem. In an undirected multicommodity flow problem, the bundle constraint of any arc (i, j) is $\sum_{k=1}^{K} | x_{ij}^k | \leq u_{ij}$. The flows x_{ij}^k can be either positive, zero, or negative. Assume that all flow costs c_{ij}^k are nonnegative.

(a) Show that using the transformation given in Figure 17.14 we can replace an undirected arc by a set of directed arcs.

(b) Explain how you would transform an undirected multicommodity flow problem into a directed multicommodity flow problem. What are the number of nodes and arcs in the transformed network?

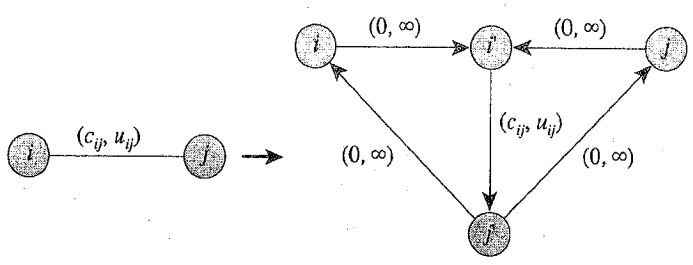


Figure 17.14 Converting an undirected arc with nonnegative cost into directed arcs.

17.22. Two-commodity undirected multicommodity flows (Hu [1963], Sakarovitch [1973]). Let $\mathcal{N}$ be the node–arc incidence matrix for a given undirected network and consider a two-commodity flow problem with the flow vectors x^1 and x^2:

$$\text{Minimize} \quad c^1x^1 + c^2x^2 \tag{17.10a}$$

subject to

$$\mathcal{N}x^1 = b^1, \tag{17.10b}$$

$$\mathcal{N}x^2 = b^2, \tag{17.10c}$$

$$|x^1| + |x^2| \leq u. \tag{17.10d}$$

In this formulation $u = (u_{ij})$ is a vector of upper bounds imposed on the total flow on the arcs. Assume that u is an integer vector. This model is a two-commodity version of the undirected multicommodity flow problem that we introduced in Exercise 17.21.

(a) Introducing vectors y^1 and y^2 and making the substitution $x^1 = y^1 + y^2 - u$ and $x^2 = y^1 - y^2$, show that the upper bound constraints (17.10d) are equivalent to the following constraints: $0 \leq y^1 \leq u$, $0 \leq y^2 \leq u$. Show that after we have made this substitution, the problem decomposes into two single commodity flow problems, one with flows y^1 and the other with flows y^2. (*Hint*: After substituting the y variables for the x variables, add and subtract (17.10b) and (17.10c).)

(b) Use the transformation introduced in part (a) to show that the two-commodity undirected multicommodity flow problem has an optimal solution in which each arc flow x_{ij}^1 is a multiple of $\frac{1}{2}$. In addition, show that if (1) the sum of the capacities of the arcs incident to each node is an even integer, and (2) all supplies and demands are even integers, the problem has an optimal solution with every x_{ij}^1 and x_{ij}^2 integer.

17.23. Multicommodity maximum flow problem. In the multiple commodity maximum flow problem, each commodity $k = 1, 2, \ldots, K$ has a source node s^k and a sink node t^k, and we wish to send the maximum total flow from the source nodes to the sink nodes while honoring a given bundle capacity u_{ij} on each arc (i, j). That is, we wish to maximize the sum of the flows over all the commodities. We refer to the largest possible sum of flows as the *maximum flow*. We say that a cut $[S, N - S]$ is a *source–sink cut* if the set S contains all of the source nodes s^k and the set $N - S$ contains all of the sink nodes t^k. The capacity of any source–sink cut is the sum of the bundle capacities across the cut. We can define the multiple commodity maximum flow problem for either directed or undirected problems. For undirected problems, we define the bundle constraints as in Exercise 17.21.

(a) Show that for either the directed or undirected versions of the problem, the maximum flow is always less than or equal to the capacity of any source–sink cut.

(b) Using the network shown in Figure 17.15, show that the maximum flow of the

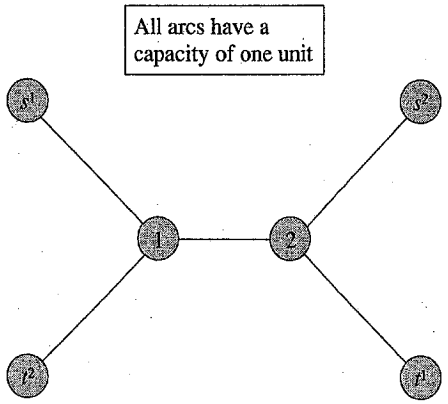


Figure 17.15 Two-commodity undirected maximum flow problem with value of the maximum flow strictly less than the value of the minimum cut.

undirected two-commodity flow problem can be strictly less than the minimum capacity of all source–sink cuts.

(c) Using the network shown in Figure 17.15 and the transformation of the undirected multicommodity flow problem into a directed problem in Exercise 17.21, show that the maximum flow of the directed two-commodity flow problem can be strictly less than the minimum capacity of all source–sink cuts.

17.24. Nonintegrality of solutions to the multicommodity maximum flow problem. Using the network shown in Figure 17.16, show that the maximum flow in an undirected two-commodity maximum flow problem can be noninteger. Give an example of a directed two-commodity flow problem with a noninteger optimal solution.

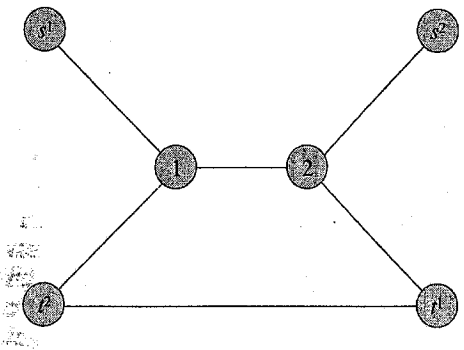


Figure 17.16 Two-commodity undirected maximum flow problem with a fractional solution.

17.25. Show how to formulate the multicommodity maximum flow problem as a version of the multicommodity flow problem (17.1).

17.26. Concurrent flow problem. Consider a multicommodity flow problem in which, for each commodity $k = 1, 2, \ldots, K$, we wish to send D_k units from the commodity's source node s^k to its destination node t^k. Suppose that the problem has no feasible solution; so instead of finding a feasible flow, we wish to find a flow that for some parameter $\theta \leq 1$ satisfies all the bundle constraints as well as sends θD_k units from the source to destination of each commodity k. We refer to the maximum value θ^* of the parameter θ for which the problem has a feasible solution as the *feasibility index* or the *optimal throughput* for the problem. We refer to the problem itself as the *concurrent flow problem* since its objeetive is to satisfy all demands by the same proportional amount θ.

(a) Formulate the concurrent flow problem as a multicommodity flow problem with one additional variable.

(b) *The feasibility index* for each commodity k is the maximum value θ_k^* of a parameter θ_k satisfying the property that the network has a feasible flow satisfying the demand $\theta_k D_k$ for commodity k (with no demand requirement for any other commodity). Show how to find θ_k^* for each commodity k.

(c) Let θ^* be the optimal throughput defined in part (a), let θ_k^* be as defined in part (b), and let $\hat{\theta} = \min_{1 \leq k \leq K} \theta_k^*$. Show that $\hat{\theta}/K \leq \theta^* \leq \hat{\theta}$.

(d) Suppose that you had an algorithm for finding a feasible flow, assuming that one exists, for a multicommodity flow problem. Show how to use this algorithm as a subroutine to determine the optimal throughput θ^* to within a factor of ϵ. That is, the objective is to determine a value θ' satisfying the inequalities $(1 - \epsilon)\theta^* \leq \theta' \leq (1 + \epsilon)\theta^*$. How many times do you need to call the subroutine for determining a feasible multicommodity flow? (*Hint*: Use the results of part (c) to limit the search.)

17.27. Show that for the concurrent flow problem described in Exercise 17.26, the optimal throughput θ^* satisfies the following inequalities: $\theta^* \leq \text{CAP}(S, N - S)/\text{DEMAND}(S)$ for every set S of nodes. In this expression, DEMAND(S) denotes the sum of the demands D_k of all the commodities whose origin nodes lie in S, and CAP(S, $N - S$) denotes the total bundle capacity of the arcs in the cut $[S, N - S]$.

17.28. Suppose that we associate a certain positive cost c_{ij} and a capacity u_{ij} with each arc (i, j) of the concurrent flow problem. For a given path P, let $c(P)$ denote the sum of the costs of the arcs in P. Also, let d_k denote the cost of the shortest path from the source node s^k to the sink node t^k of commodity k using the arc costs c_{ij}. Show that the maximum throughput θ^* satisfies the following inequality: $\theta^* \leq (\sum_{(i,j)\in A} c_{ij}u_{ij})/(\sum_{1\leq k\leq K} d_k D_k)$. (*Hint*: Show that $\theta^* \sum_{1\leq k\leq K} d_k D_k$ is a lower bound on the cost of any feasible flow that satisfies all the demands, and that $\sum_{(i,j)\in A} c_{ij}u_{ij}$ is an upper bound on the cost of any feasible flow.)

17.29. Dynamic throughput in computer networks. Suppose that in a computer system, for each pair $[i, j]$ of nodes, we would like to send d_{ij} units of information from node i to node j in the least amount of time. Sending any message on arc (i, j) requires t_{ij} units of time and each arc (i, j) has a bundle capacity of u_{ij} at each instant in time (i.e., it can carry at most u_{ij} messages at each instant in time). How would you approximate this dynamic throughput problem as a maximum concurrent flow problem?

17.30. Show that the multicommodity flow problem always has K redundant constraints by viewing it as follows. Each commodity flows on a separate, but identical network and the bundle constraints tie together the flows on the separate network. In this "layered representation" of the problem, let j^k denote the copy of node j on the network (layer) corresponding to commodity k. Next show that it is possible to replace the nodes 1^k for $k = 1$ to K by a single node 1^*, and that the resulting formulation corresponds to a connected network.

17.31. Single bundle constraint as a single-commodity flow problem. Suppose that a multicommodity flow problem has a bundle constraint imposed upon only one arc (i, j). Show that the resulting problem is a minimum cost flow problem. (*Hint*: Use the result of Exercise 17.30.)

17.32. Penalty approach for the multicommodity flow problem (Schneur [1991]). Consider the following penalty approach for solving the multicommodity flow problem:

$$\text{Minimize} \quad F(x) = \sum_{1\leq k\leq K} c^k x^k + \rho/2 \sum_{(i,j)\in A} (f_{ij})^2 \tag{17.11a}$$

subject to

$$\sum_{1\leq k\leq K} x_{ij}^k - f_{ij} \leq u_{ij} \qquad \text{for all arcs } (i, j) \in A, \tag{17.11b}$$

$$\mathcal{N}x^k = b^k \qquad \text{for all } k = 1, \ldots, K, \tag{17.11c}$$

$$f_{ij} \geq 0 \qquad \text{for all arcs } (i, j) \in A, \tag{17.11d}$$

$$0 \leq x_{ij}^k \leq u_{ij}^k \qquad \text{for all arcs } (i, j) \in A \text{ and all } k = 1, 2, \ldots, K. \tag{17.11e}$$

Note that in any optimal solution, f_{ij} is the excess flow in arc (i, j) [i.e., $f_{ij} = \max(0, \sum_{k=1,\ldots,K} x_{ij}^k - u_{ij})$]. Therefore, this model replaces the hard (i.e., fixed) bundle capacity u_{ij} on any arc (i, j) by a quadratic penalty for exceeding the arc's capacity. In this formulation ρ is a parameter specifying how much we penalize excess flows.

(a) Let z denote the optimal objective function value for the linear multicommodity flow problem and let $G(\rho)$ denote the optimal objective function value of the penalty problem for a fixed value of ρ. Show that $G(\rho) \leq z$ for all ρ. Show also that $\lim_{\rho\to\infty} G(\rho) = z$.

(b) Let y denote 1 unit flow of commodity k around the cycle W (i.e., a flow of $+1$ in each forward arc of W for commodity k and a flow of -1 in each backward arc of W for commodity k). We refer to the cycle W as a *negative cycle* with respect to a flow x' if $F(x' + \theta y) - F(x')$ is negative for some scalar $\theta > 0$. Show that a feasible flow x' of (17.11) is an optimal flow if and only if it contains no negative cost cycle. (*Hint*: The proof is similar to that of Theorem 3.8.)

17.33. Penalty method for the multicommodity flow problem. In this exercise we discuss an algorithm for the multicommodity flow problem that uses the penalty approach described in Exercise 17.32. Let W be a cycle in the network and $c(W)$ be its cost (i.e., the sum of the costs of the forward arcs in W minus the costs of the backward arcs in W). Moreover, let $f(W)$ denote the excess flow of W (i.e., the sum of the excess flows of the forward arcs minus the sum of the excess flows of the backward arcs). Finally, let y denote a unit flow of the commodity k around the cycle W (as defined in Exercise 17.32(b).

(a) Express $F(x' + \theta y) - F(x')$ in terms of θ, $| W |$, $c(W)$, and $f(W)$, and observe that $F(x' + \theta y) - F(x')$ is a convex function of θ. Use this observation to suggest a polynomial-time algorithm for finding a negative cycle in the network.

(b) Use your answer in part (a) to describe an algorithm for solving the penalty version of the multicommodity flow problem.

17.34. Suppose that we apply Lagrangian relaxation to the resource-allocation problem (17.7) by associating Lagrange multipliers w_{ij} with the resource-allocation constraints $\sum_{1 \le k \le K} r_{ij}^k \le u_{ij}$, to produce the following Lagrangian subproblem:

$$z = \min \sum_{1 \le k \le K} c^k x^k + \sum_{(i,j) \in A} w_{ij} \left(\sum_{1 \le k \le K} r_{ij}^k - u_{ij} \right)$$

subject to

$$\mathcal{N} x^k = b^k \qquad \text{for } k = 1, \ldots, K,$$

$$0 \le x_{ij}^k \le r_{ij}^k \qquad \text{for all } (i, j) \in A \text{ and all } k = 1, 2, \ldots, K.$$

Show that this problem is equivalent to the Lagrangian subproblem determined by applying Lagrangian relaxation to the original formulation (17.1) obtained by relaxing the bundle constraints $\sum_{1 \le k \le K} x_{ij}^k \le u_{ij}$.

17.35. Consider a linear program of the form

$$\text{Minimize} \quad \sum_{1 \le k \le K} c^k x^k \tag{17.12a}$$

subject to

$$\sum_{1 \le k \le K} \mathcal{A}^k x^k \le b, \tag{17.12b}$$

$$\mathcal{D}^k x^k = d^k \qquad \text{for all } k = 1, 2, \ldots, K, \tag{17.12c}$$

$$x^k \ge 0 \qquad \text{for all } k = 1, 2, \ldots, K. \tag{17.12d}$$

In this formulation, each x^k is a vector of decision variables. The constraints (17.12c) are separate constraints imposed on each vector x^k, and the constraint (17.12b) models the limited availability of joint resources shared by these decision vectors. Note that each $\mathcal{D}^k$ and $\mathcal{A}^k$ is a matrix.

(a) Suppose that we introduce new "resource-allocation" variables r^k and replace the constraint (17.12b) by the constraints $\sum_{1 \le k \le K} r^k \le b$ and $\mathcal{A}^k x^k \le r^k$ for each $k = 1, 2, \ldots, K$. Show that the formulation in the variables r^k is equivalent to the given formulation.

(b) Using the model with the resource-allocation variables r^k, show how to adapt the resource-directive decomposition technique described in Section 17.7 to solve (17.12).

17.36. Show how to apply the basis partitioning algorithm to solve the optimization model specified in the last exercise.

(a) First show that any basis $\mathcal{B}$ for the problem must contain a basis $\mathcal{B}^k$ for each matrix $\mathcal{D}^k$.

(b) Next show how to make a change in variables so that the only variables appearing

in the basis in the equations $\mathcal{D}^k x^k = d^k$ are those in the basis $\mathcal{B}^k$. (*Hint*: If γ is any column of $\mathcal{D}^k$, it is always possible to solve the equation $\mathcal{B}^k y = \gamma$).

(c) Define the working basis $\mathcal{W}$ as those columns of the constraints $\sum_{1 \le k \le K} \mathcal{A}^k x^k \le b$ that are in the basis $\mathcal{B}$ but not in any $\mathcal{B}^k$. Show that the columns of $\mathcal{W}$ are linearly independent.

(d) Show how to use the working basis $\mathcal{W}$ and the basis matrices $\mathcal{B}^1, \mathcal{B}^2, \ldots, \mathcal{B}^K$ to implement the simplex method for solving the problem.

17.37. Show how to apply Dantzig–Wolfe decomposition to the optimization model (17.12) with the constraints (17.12c) and (17.12d) defining the subproblems.

17.38. Consider the optimization model

$$\text{Minimize} \quad cx$$

subject to

$$Ax = b,$$

$$x \in X,$$

defined over a finite set X. Show how to apply column generation and Dantzig–Wolfe decomposition to solve the following convexified version of this problem:

$$\text{Minimize} \quad cx$$

subject to

$$Ax = b \tag{17.13}$$

$$x \in \mathcal{H}(x)$$

defined over the convex hull $\mathcal{H}(x)$ of x. Recall that any point $x \in \mathcal{H}(x)$ has the representation $x = \sum_{k=1}^{K} \lambda^k x^k$ for some set of nonnegative weights satisfying the weighting condition $\sum_{k=1}^{K} \lambda^k = 1$. (*Hint*: Make the replacement $x = \sum_{k=1}^{K} \lambda^k x^k$ to formulate problem (17.13) in terms the weighting variables $\lambda^1, \lambda^2, \ldots, \lambda^K$ and form a restricted version of the reformulated model by eliminating all but a small number of the λ^k's.)

18

COMPUTATIONAL TESTING OF ALGORITHMS

The purpose of mathematical programming is insight, not numbers.
—A. M Geoffrion

Chapter Outline

18.1 Introduction
18.2 Representative Operation Counts
18.3 Application to Network Simplex Algorithm
18.4 Summary

18.1 INTRODUCTION

In this book we have focused on developing the most "efficient" algorithms for solving network optimization problems. The notion of efficiency involves all the various computing resources needed for executing an algorithm. However, since time is often a dominant computing resource in practice, we have used computational time as the primary measure for assessing algorithmic efficiency. We have measured the computational time of an algorithm through its worst-case analysis. Worst-case analysis provides upper bounds on the number of steps that a given algorithm can take on *any* problem instance. As we have noted in Chapter 3, for a variety of reasons, the worst-case analysis of algorithms is a very popular criterion for judging algorithmic efficiency, and this approach has stimulated considerable research. However, worst-case analysis can be overly pessimistic since it permits "pathological" instances to determine the performance of an algorithm, even though they might be exceedingly rare in practice. Often, the empirical behavior of an algorithm is much better than suggested by its worst-case analysis. Consequently, the research community typically relies on the empirical testing of an algorithm to assess its performance in practice.

In the operations research literature, researchers have conducted a large number of empirical investigations of various network flow algorithms to determine the "best" algorithms in practice. A typical study tests more than one algorithm for a specific network flow problem and generally consists of the following steps: (1) write a computer program (often in FORTRAN) for each algorithm to be tested; (2) use pseudorandom network generators to generate random problem instances with selected combinations of input size parameters (e.g., nodes and arcs); (3) run computer

programs and note the CPU (central processing unit) times for the different algorithms on the data obtained by the network generators; and (4) declare the algorithm that takes the least amount of CPU time as the "winner" (if different algorithms are faster for different input size parameters, then report this fact as well).

The existing literature on computational testing has a tendency to overrely on CPU time as the primary measure of performance. CPU time depends greatly on subtle details of the computational environment and the test problems such as (1) the chosen programming language, compiler, and computer; (2) the implementation style and skill of the programmer; (3) network generators used to generate the random test problems; (4) combinations of input size parameters; and (5) the particular programming environment (e.g., the use of the computer system by other users). Because of the multiple sources of variabilities, CPU times are often difficult to replicate, which is contrary to the spirit of scientific investigation. Another drawback of the use of CPU time is that it is an aggregate measure of empirical performance and does not provide much insight about an algorithm's behavior. For example, an algorithm generally performs some fundamental operations repeatedly, and a typical CPU time analysis does not help us to identify these "bottleneck" operations. Identifying the bottleneck operations of an algorithm can provide useful guidelines for where to direct future efforts to understand and subsequently improve an algorithm.

The spirit of worst-case analysis is to identify theoretical bottlenecks in the performance of any algorithm and to provide upper bounds on the computation counts of these bottleneck operations as a measure of the algorithm's overall behavior. Borrowing this point of view for computational testing, we might attempt to measure the empirical performance of an algorithm (or its computer implementation) by counting the number of times the algorithm executes each of these bottleneck operations while solving each instance of the problem. That is, we would conduct computer experiments to obtain an actual count of bottleneck operations instead of providing a theoretical upper bound on this number. This approach suggests that in analyzing the empirical behavior of an algorithm, we need not count the number of times it executes each line (of possibly thousands of lines) of code, but instead can focus on a relatively small number of lines that are "summary measures" of the algorithm's empirical behavior. Even for the most complex algorithms described in this book, we need to keep track of the computation counts of at most three or four operations or lines of code.

For example, in the FIFO preflow-push algorithm for the maximum flow problem that we presented in Section 7.7, each push operation first selects an active node i, next selects an admissible arc (i, j), and then pushes $\min\{e(i), r_{ij}\}$ units of flow on the arc. If no admissible arc emanates from node i, we scan all of the arcs emanating from this node, and we relabel the distance label of node i, giving it the value $d(i) = 1 + \min\{d(j) : r_{ij} > 0\}$. We claim that for the generic preflow-push algorithm, we need to keep track of only two operations: (1) the number of pushes, and (2) the number of arcs scanned in the relabel operations.

These two operations dominate every other operation of the generic preflow-push algorithm. To establish this statement, note that to select a node requires $O(1)$ operations per push, so the algorithm spends O(number of pushes) operations in selecting nodes. In selecting admissible arcs, we check if the current arc is admissible, and if not, we modify the current arc. The algorithm modifies the current arc

for node i $O(|A(i)|)$ times, which is the number of arcs scanned in relabeling node i between successive relabels of node i. Thus the algorithm modifies the current arcs O(number of arcs scanned in relabels) times. We leave it to the reader to verify that these two operations also bound each of the other operations performed in the FIFO preflow-push algorithm. To summarize, even though an implementation of the FIFO preflow-push algorithm might contain hundreds of lines of code, we need to keep track of only two fundamental operations in order to identify the bottleneck operations, as well as to estimate the running time to within a constant factor.

We will soon formalize this notion of "representative operation counts" in computational testing; however, let us first summarize some of the advantages of this approach as compared to the more common approach of analyzing only CPU time.

1. Representative operation counts allow us to identify the asymptotic bottleneck operations of an algorithm—i.e., the operations that progressively consume a larger share of the computational time as the problem size increases. (For sufficiently large problem sizes, improving the asymptotic bottleneck operation has the maximum possible impact on the running time of the algorithm.)
2. Representative operation counts provide more guidance and insight about comparing two algorithms that are run on different computers and permits us even to compare algorithms implemented with different computer languages.
3. Representative operation counts permit us to determine lower and upper bounds on the asymptotic growth rate in computation time as a function of the problem size.
4. We can use statistical methodologies to estimate the CPU time on a computer as a linear function of the representative operation counts. We refer to this estimate of the CPU times as the *virtual CPU time*. Virtual CPU time permits researchers to carry out experiments on different computers, but estimate the running times as if all the experiments had been carried out on the same computer.

This type of asymptotic empirical analysis complements the worst-case analysis that we have examined in many other chapters. The empirical analysis using representative operation counts allows us to identify the actual empirical behavior of the algorithm for sufficiently large problem sizes. Just as the worst-case analysis ignores constant factors in the running time, empirical analysis using representative operation counts ignores constant factors in the running time, and instead focuses on the dominant term in the computations.

Before continuing we might note that the field of computational testing is very broad, and we cannot do justice to it in a short chapter. Rather than treat the wide range of topics of importance in computational testing (such as how to select test problems, how to conduct an experimental design, what type of statistical tests are most appropriate, and what data needs to be reported), we focus on the use of representative operation counts as an aid in the analysis of computational experiments. We believe that it is quite easy to count representative operations while conducting any computational testing and that generating this added information has considerable potential payback.

18.2 REPRESENTATIVE OPERATION COUNTS

In order to formalize our notion of counting operations performed by an algorithm, let us first stand back and consider what a computer does in executing a computer program. Suppose that $\mathcal{A}$ is a computer program for solving some problem and that I is an instance of the problem. The computer program consists of a finite number of lines of computer code, say $a_1, a_2, \ldots, a_K$. Each line of code gives either one or a small number of instructions to the computer. The instruction might tell the computer to carry out an arithmetic operation on a register or to move data from one memory location to another; or it might be a control instruction informing the computer which line of code it should execute next. For convenience, we assume that the program is written so that each line of the code gives $O(1)$ instructions to the computer and that each instruction requires $O(1)$ units of time. We assume that the fastest computer operation requires 1 time unit. (Although these assumptions are reasonable, they do imply some restrictions; for example, we do not permit lines of code that tell the computer to add two vectors, as would be allowed in some high-level languages such as APL. Rather, we would require that adding vectors be carried out as a loop that sums the two vectors one component at a time.) The assumption that each operation executed by the computer requires a comparable amount of time seems reasonable in practice with the notable exceptions of input–output, caching (moving data to and from storage), and paging (memory management of the secondary storage space).

The preceding discussion implies that executing any line of code requires $O(1)$ time units, and at least 1 time unit. Therefore, each line of code requires $\Theta(1)$ time units since its execution time is bounded from both above and below by a constant number of units. Suppose that the computer code we are investigating has K lines of code. For a given instance I of the problem, let $\alpha_k(I)$, for $k = 1$ to K, be the number of times that the computer executes line k of this computer program. Let CPU(I) denote the CPU time of the computer program on instance I. The preceding discussion implies the following lemma.

Lemma 18.1. $CPU(I) = \Theta(\sum_{k=1}^{K} \alpha_k(I))$. ◆

Lemma 18.1 states that we can estimate the running time of an algorithm to within a constant factor by counting the number of times it executes each line of code. However, counting each line of code is unnecessarily burdensome. As we shall soon see, it really suffices to count a relatively small number of lines of code. For example, consider the following fragment of code.

```
for i := 1 to 3 do
begin
  A(i) := A(i) + 1;
  B(i) := B(i) + 2;
  C(i) := C(i) + 3;
end;
```

We need not count the number of times the algorithm executes the statement "$B(i) := B(i) + 2$," since it executes this statement whenever it executes the

statement "$A(i) := A(i) + 1$." Similarly, we need not count the number of times the algorithm executes the statement "$C(i) := C(i) + 3$." Moreover, it appears that regardless of the size of the problem we are solving, the algorithm modifies nine elements of the vectors A, B, and C during the execution of the "for loop." So in this case it would suffice to keep track of just the number of times that the algorithm executes the "for loop."

Note that we have treated the number of iterations of the do loop as a constant. Suppose, instead, that the first line of the do loop were "**for** $i := 1$ **to** 10 **do**." Should we also treat the 10 as a constant? What if the first line of the do loop were "**for** $i := 1$ **to** 10,000 **do**." Should we continue to treat the 10,000 as a constant? In answering these questions, we might invoke two rules of thumb. First, we should determine the representative operation counts after expressing the algorithm as a pseudocode with all the problem parameters treated explicitly as parameters and not as constants. For example, although in many practical situations $\log U$ will be less than 16, in a pseudocode we should use the term "$\log U$" rather than the constant 16. We would not treat $\log U$ as a constant. Second, as a rule, we should not treat the number of iterations of a "do loop" or a "while loop" or any other loop as a constant since frequently the number of times that the program will call the loop depends on a problem parameter rather than a constant.

We now formalize the notion we have been suggesting; that is, keeping track of a small number of lines of code, which we call the *representative operation counts*. Let S denote a subset of $\{1, \ldots, K\}$, and let a_S denote the set $\{a_i : i \in S\}$. We say that a_S is a representative set of lines of code of a program if for some constant c,

$$\alpha_i(I) \leq c \left(\sum_{k \in S} \alpha_k(I)\right),$$

for every instance I of the problem and for every line a_i of code. In other words, the time spent in executing line a_i is dominated (up to a constant) by the time spent in executing the lines of code in a_S. With this definition, we have the following corollary to Lemma 18.1.

Property 18.2. *Let S be a representative set of lines of code. Then* $CPU(I) = \Theta(\sum_{k \in S} \alpha_k(I))$.

Proof. By Lemma 18.1, $CPU(I) = \Theta(\sum_{k=1}^{K} \alpha_k(I))$. Moreover, for each line α_i of code not in S, $\alpha_i(I) \leq c(\sum_{k \in S} \alpha_k(I))$, so $\sum_{k=1}^{K} \alpha_k(I) = \Theta(\sum_{k \in S} \alpha_k(I))$. ◆

This methodology identifies a set of representative lines of code, and during empirical investigations keeps track of the representative operation counts for each instance solved. Sometimes, the selected representatives will not refer specifically to one line of code. For example, we might keep track of the number of pushes in the preflow-push algorithms, and each push might be described over several lines of code. Rather than use the expression "the computation counts for a set of representative lines of code" we will refer to "the computation counts for a set of representative operations" or more briefly as *representative operation counts*.

In the next section we discuss various uses of representative operation counts. In addition to noting representative operation counts for various problem instances we have solved, we might record some other counts that might be helpful in assessing

an algorithm's behavior. For example, for each instance solved, we might note the number of major iterations that the algorithm performs, such as the number of pivots of the network simplex algorithm.

At first glance we might suspect that determining a representative set of operations could be difficult and that the set might be quite large. In fact, our experience suggests that it is generally quite easy to determine representative sets, and often we have several possible choices. In addition, the representative sets are typically quite small. For the algorithms presented in this book, the number of representative operation counts typically range from 1 to 4, even for quite complex algorithms.

We now give examples of representative sets for several network flow algorithms we have discussed in this book. The representative sets we indicate are not unique. In some cases we suggest additional operation counts that might be of value in empirical investigations.

Dial's implementation of Dijkstra's algorithm (see Section 4.6). A set of representative operations for this algorithm are (1) the number of buckets scanned while identifying the first nonempty bucket (as part of a findmin operation), and (2) the number of arcs scanned to update distance labels. Since we know that the implementation scans $\Theta(m)$ arcs, we need not actually count the number of arc scans. Thus we need to keep track of only one operation in our representative set. In addition to these representative operation counts, we might keep track of other operations. For example, we might record the number of decreases in the distance labels during the distance update operations since this set of operation counts would also allow us to bound the running time of the binary heap implementation of Dijkstra's algorithm.

Original implementation of Dijkstra's algorithm (see Section 4.5). The running time of this algorithm is $\Theta(n^2)$ since the number of nodes scanned in the findmin operation is $(n-1) + (n-2) + \cdots + 2 + 1 = n(n-1)/2 = \Omega(n^2)$ time. Therefore, we need not keep track of any counts in executing the algorithm.

FIFO label-correcting algorithm for the shortest path problem (see Section 5.4). The representative operation for this algorithm is the arcs scanned while examining nodes in the set LIST.

Labeling algorithm for the maximum flow problem (see Section 6.5). A representative operation for this algorithm is the arcs scanned while examining labeled nodes. This operation dominates all other operations that the labeling algorithm performs. In addition to this representative operation, we would probably want to count the number of augmentations. This additional information might provide insight for comparing the labeling algorithm to other maximum flow algorithms. For example, we could check whether the labeling algorithm requires more augmentations than other augmenting path algorithms.

Preflow-push algorithm for the maximum flow problem (see Section 7.7). A set of representative operations for this algorithm are (1) the number of nonsaturating pushes, and (2) the arcs scanned while updating distance labels of

nodes. (Operation 2 dominates the number of operations the algorithm performs in updating the current arc and in making saturating pushes.) We do not need to keep track of saturating pushes because the time spent in saturating pushes is bounded by the time spent in scanning arcs to determine a current arc, and this time, in turn, is bounded by the time spent updating the distance labels of nodes. Even though we know that saturating pushes will not be a bottleneck operation, we might want to keep track of them for a different reason. In practice, it appears that nonsaturating pushes are not a bottleneck operation, even though they are the theoretical bottleneck. An easy way to show that nonsaturating pushes are not a bottleneck operation is to show, for example, that the number of nonsaturating pushes are, in practice, at most a small constant times the number of saturating pushes. Keeping track of saturating pushes would permit us to make this assessment empirically. We might also want to record the number of saturating pushes because they might be a bottleneck operation if we use the dynamic trees data structure, and by keeping track of the two operations (1) and (2) we can estimate whether the use of dynamic trees might lead to an asymptotic improvement in the running time.

Successive shortest path algorithm for the minimum cost flow problem (see Section 9.7). The successive shortest path algorithm determines shortest paths from excess nodes to deficit nodes. Suppose that we use Dial's implementation to solve shortest path problems. Then the representative operations for this algorithm are: (1) the buckets scanned while the algorithm identifies nonempty buckets in Dial's algorithm, and (2) the number of arcs that the algorithm scans while it updates distance labels. In addition, we would want to keep track of the number of shortest path problems that the algorithm solves.

Network simplex algorithm (see Section 11.5). The network simplex algorithm has the following set of representative operations: (1) the number of arcs whose reduced cost the algorithm calculates while it is identifying the entering arc, (2) the number of arcs in the pivot cycles that the algorithm creates when it adds the entering arc to the current spanning tree, and (3) the number of nodes in the subtree T_2 (i.e., nodes whose potentials the algorithm changes during a pivot operation). The network simplex algorithm performs many other operations. We do not need to keep track of these operations because the three operation counts we have identified dominate them. For example, to update flows in a pivot cycle W requires $\Theta(|\ W\ |)$ time. (Even for degenerate pivots, the time spent by most algorithms is at least $|\ W\ |$, since most algorithms identify the pivot cycle before determining the flow to send around the cycle.) To update the multipliers in a subtree T_2 of nodes whose potential changes require $\Theta(|\ T_2\ |)$ time. Similarly, to update the tree indices requires $\Theta(|\ W\ | + |\ T_2\ |)$ time. This set of representative operations is strikingly compact considering the fact that implementations of the network simplex algorithm are typically quite intricate.

In addition to these representative operations, we might keep track of other operations. For example, we would most likely want to count the number of pivots. Moreover, if we were concerned about the effects of degeneracy, we would also record the number of degenerate pivots that the algorithm performs.

18.3 APPLICATION TO NETWORK SIMPLEX ALGORITHM

In this section we illustrate the use of representative operation counts using the network simplex algorithm for the minimum cost flow problem. We provide experiments based on the network simplex algorithm, as implemented with the first eligible arc pivot rule. As described in Section 11.5, this pivot rule selects the first arc with positive violation as the entering arc while scanning the arcs in the wraparound fashion.

The computational time that the network simplex algorithm requires to solve a minimum cost flow problem depends on a number of different parameters, including the number of nodes, the number of arcs, the network generator, the size of the cost and capacity data, and the number of supply nodes. To illustrate the approach discussed in this chapter, of these factors, we have chosen to focus on the number of nodes and the number of arcs. We conducted experiments on networks with 1000, 2000, 4000, 6000, and 8000 nodes. For each choice of n nodes, we created networks with $2n$, $4n$, $6n$, $8n$, and $10n$ arcs. Setting $d = m/n$, we have considered networks with $d = 2$, 4, 6, 8, or 10. Thus the largest network size we considered had 8000 nodes and 80,000 arcs.

We used the well-known network generator NETGEN to generate minimum cost flow problems with specified values of n and m. For each specific setting of n and m, we solved five different problems generated from NETGEN and computed the average of these five problem instances. (The averages have a lower variability of outcomes than the individual tests, so the resulting graphs and charts reveal patterns more clearly.) For each problem that we solved, we noted the following values:

α_E: number of arcs scanned in selecting an entering arc (summed over all pivots)

α_W: number of arcs in the pivot cycles (summed over all pivots)

α_P: number of node potentials modified (summed over all pivots)

p: number of pivots, further decomposed into the number of degenerate pivots and the number of nondegenerate pivots

τ: CPU time to execute the algorithm (times noted on a HP9000/850 computer under a multiprogramming and multisharing environment).

We computed the averages of these values over five problem instances for each parameter setting. Figure 18.1 gives these averages for the network simplex algorithm with the first eligible arc pivot rule.

Identifying Asymptotic Bottleneck Operations

We consider an operation to be a "bottleneck operation" for an algorithm if the operation consumes a significant percentage of the execution time on at least some fraction of the problems tested. We refer to an operation as an *asymptotic nonbottleneck operation* if its share in the computational time becomes smaller and approaches zero as the problem size increases. Otherwise, we refer to an operation as an *asymptotic bottleneck operation.* The asymptotic bottleneck operations are

No.	n	m	d	α_E	α_W	α_P	p	$\alpha_E + \alpha_W + \alpha_P$	CPU time
1	1,000	2,000	2	41,590	183,540	237,599	4,721	462,729	4.46
2	2,000	4,000	2	129,125	831,593	1,356,773	15,358	2,317,491	21.62
3	4,000	8,000	2	406,115	3,989,374	8,319,325	51,936	12,714,814	119.7
4	6,000	12,000	2	863,573	10,770,317	25,045,320	116,303	36,679,210	345.64
5	8,000	16,000	2	1,453,942	21,438,864	54,392,393	197,005	77,285,199	699.88
6	1,000	4,000	4	87,883	284,830	403,699	10,275	776,412	7.31
7	2,000	8,000	4	248,760	1,101,807	2,090,021	29,879	3,440,588	31.26
8	4,000	16,000	4	765,758	5,171,901	12,341,060	99,858	18,278,719	160.7
9	6,000	24,000	4	1,475,919	12,277,824	33,993,925	197,858	47,747,668	420.96
10	8,000	32,000	4	2,419,120	24,614,630	77,921,501	335,296	104,955,251	853.21
11	1,000	6,000	6	141,475	351,763	518,265	14,372	1,011,503	9.33
12	2,000	12,000	6	379,171	1,253,222	2,353,123	38,837	3,985,516	35.42
13	4,000	24,000	6	1,152,766	5,655,431	15,215,392	124,249	22,023,589	188.55
14	6,000	36,000	6	2,329,115	14,249,457	42,841,133	259,742	59,419,705	505.79
15	8,000	48,000	6	3,639,711	26,157,289	90,280,080	416,123	120,077,080	978.57
16	1,000	8,000	8	207,114	380,015	598,636	16,586	1,185,765	10.7
17	2,000	16,000	8	529,747	1,452,365	3,012,190	47,116	4,994,302	42.61
18	4,000	32,000	8	1,534,355	5,919,861	13,833,227	139,434	21,287,443	200.67
19	6,000	48,000	8	2,918,853	13,917,221	48,131,763	273,089	64,967,837	518.08
20	8,000	64,000	8	4,532,492	25,791,836	99,057,203	437,558	129,381,531	1,033.05
21	1,000	10,000	10	266,541	410,304	739,070	18,634	1,415,915	12.39
22	2,000	20,000	10	710,258	1,555,660	3,305,765	53,425	5,571,683	48.08
23	4,000	40,000	10	1,975,386	5,844,629	17,922,439	145,303	25,742,454	208.30
24	6,000	60,000	10	3,902,416	15,193,494	48,615,853	312,355	67,711,763	584.34
25	8,000	80,000	10	5,942,548	26,724,636	103,238,910	487,624	135,906,094	1,064.45

Figure 18.1 Table of computation counts for the network simplex algorithm with the first entering pivot rule.

important because they determine the running times of the algorithm for sufficiently large problem sizes.

We have earlier shown that the representative operation counts provide both upper and lower bounds on the number of operations an algorithm performs; therefore, the set of representative operations must contain at least one asymptotic bottleneck operation. There is no formal method for determining asymptotic bottleneck operations using computational testing unless we are willing to impose further assumptions on the behavior of an algorithm on large problems. After all, it is theoretically possible that an algorithm behaves one way for problems of sufficiently large size and it behaves totally differently for small problems. Nevertheless, some procedures seem quite effective in practice for determining asymptotic bottlenecks, even if we cannot completely justify their use.

We illustrate how to find an asymptotic bottleneck operation for the network simplex algorithm. Let $\alpha_S(I) = \alpha_E(I) + \alpha_W(I) + \alpha_P(I)$. Then we simply plot

$\alpha_E(I)/\alpha_S(I)$, $\alpha_W(I)/\alpha_S(I)$, $\alpha_P(I)/\alpha_S(I)$ for increasingly larger problem instances I and look for a trend. Figure 18.2 gives these plots for the network simplex algorithm with the first eligible arc pivot rule. In these plots we use the number of nodes as a surrogate for the problem size and provide a plot for each different density $d = m/n$. This choice helps us to visualize the effect of n and m on the growth in representative operation counts. These plots suggest that updating node potentials is an asymptotic bottleneck operation in the algorithm.

Estimating Growth Rates of Bottleneck Operations

How should we estimate the growth in the computational time (i.e., CPU time) as the problem size increases? Instead of directly estimating the growth in the computational time, we estimate the growth in the asymptotic bottleneck operations. By focusing only on the asymptotic bottleneck operations, we eliminate the contribution of the nonbottleneck operations, and therefore the estimates should be superior. In the notation of the preceding section, the asymptotic running time is proportional to $\lim_{|I|\to\infty} \max(\alpha_1(I), \alpha_2(I), \ldots, \alpha_K(I))$. In the manner that we have selected the representative operations, the asymptotic running time is also proportional to $\lim_{|I|\to\infty} \max(\alpha_i(I) : i \in S)$. We need not consider the nonbottleneck operations in estimating the asymptotic running time. The CPU time is a linear function of the terms $\alpha_i(I)$ for $i = 1$ to K, and thus includes information from many of the nonbottleneck operations. For this reason, the CPU time is a much "noisier" estimator of the asymptotic running time.

We describe a simple approach for estimating the growth rates of a bottleneck operation, which in our illustration is α_p, the number of potential updates. Our approach consists of the following steps:

1. Determine an appropriate functional form for estimating the counts for the operation α_p. In our example, we choose the functional form to be n^γ for some choice of a growth parameter γ. (A more common approach would be to choose a function of both n and m; however, we will consider each network density separately, and for each network density $d = m/n$, the ratio of m to n is fixed. Therefore, a function of m and n reduces to a function of n.)
2. Select a candidate lower bound n^l and a candidate upper bound n^u on the growth rate using one of the several possible methods.
3. Evaluate (using some methodology) whether n^l and n^u are really the lower and upper bounds on α_p. If not, return to step 2 and repeat the process.

For our illustration, suppose that we wished to consider a candidate lower bound of $n^{2.2}$ on the growth rate of the number of potential updates α_p and a candidate upper bound of $n^{2.8}$ (we might simply have guessed to determine these bounds). Figure 18.3 gives a plot of $\alpha_p/n^{2.2}$ and $\alpha_p/n^{2.8}$. In Figure 18.3(a) we find that the function $\alpha_p/n^{2.2}$ has an increasing trend with the problem size, suggesting that $n^{2.2}$ is indeed a lower bound on the number of potential updates. Further, we find in Figure 18.3(b) that the function $\alpha_p/n^{2.8}$ has a decreasing trend, suggesting that $n^{2.8}$ is an upper bound on the number of potential updates. If we want more refined lower and upper bounds, we carry out the technique further to see if $n^{2.4}$ is a valid lower

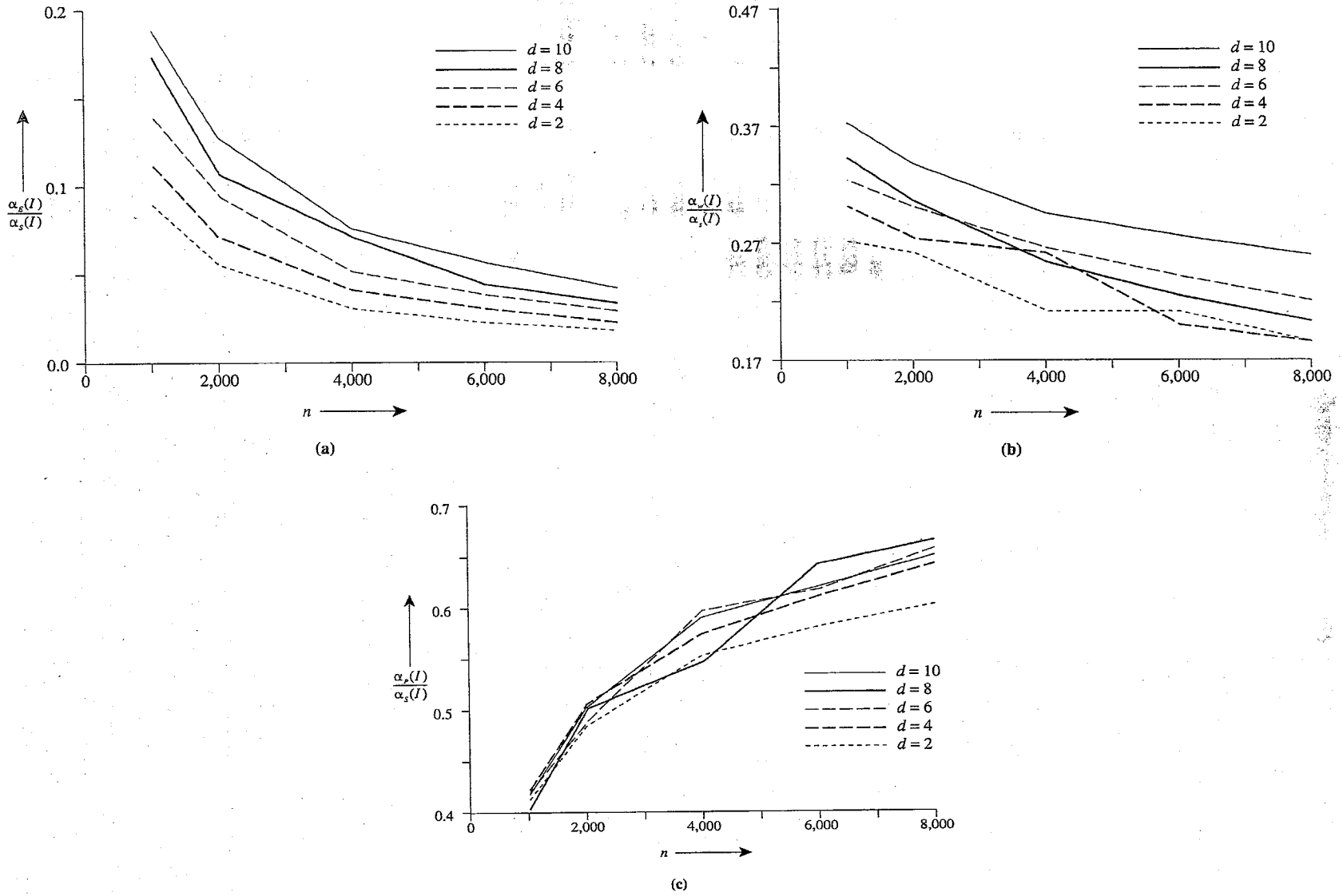


Figure 18.2 Identifying asymptotic bottleneck operations: (a) ratio of arc scan operations and total number of operations; (b) ratio of flow update operations and total number of operations; (c) ratio of potential update operations and the total number of operations.

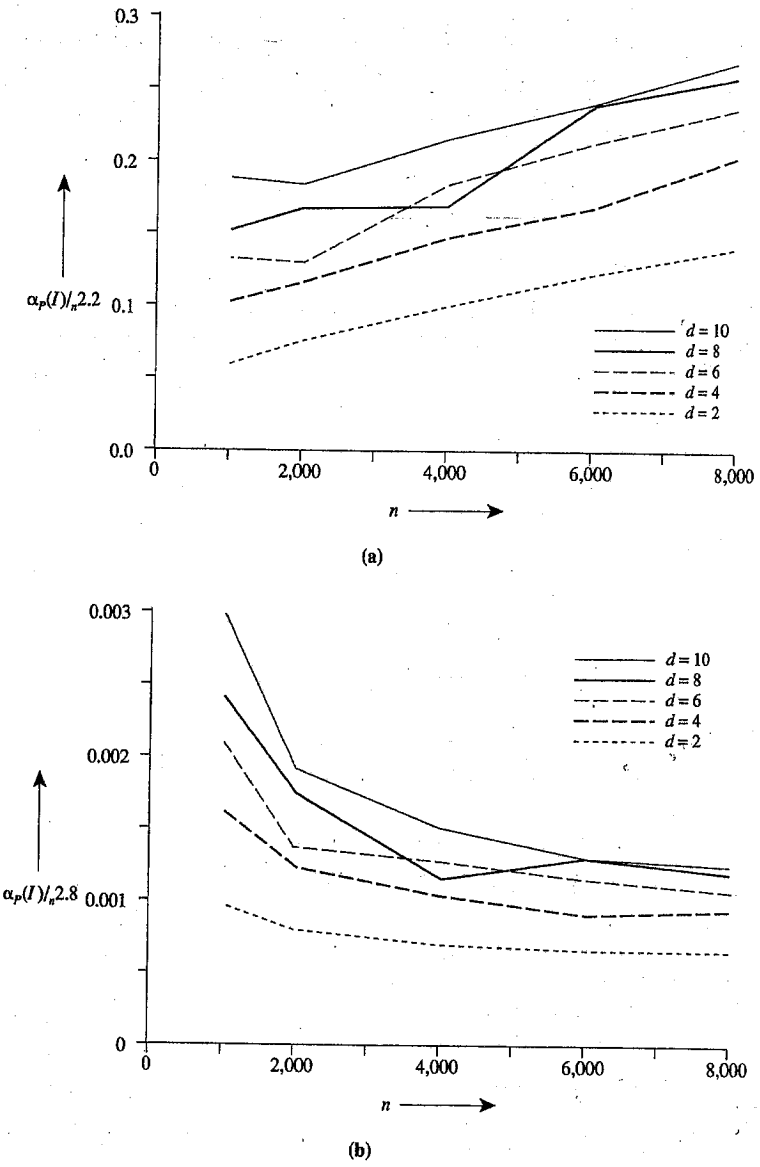


Figure 18.3 Determining asymptotic growth rates of the bottleneck step: (a) plot of $\alpha_P(I)/n^{2.2}$ has an increasing trend; (b) plot of $\alpha_P(I)/n^{2.8}$ has a decreasing trend.

bound or if $n^{2.6}$ is a valid upper bound. The polynomial estimates for the lower and upper bounds are significantly different because our method for estimating lower and upper bounds typically is quite conservative and underestimates the lower bound and overestimates the upper bound.

We might emphasize that our approach is intended primarily for gathering insight into the upper and lower bounds on function growth and is not rigorous. For example, to determine the upper bound on the growth, the methodology must recognize that a certain function has an increasing trend. However, we have not rigorously defined what we mean by "increasing trend"; it depends, to an extent, on a judgment call by the user.

In addition to its lack of rigor, the methodology, as we have used it, is perhaps too conservative, and can be strengthened through careful statistical analysis. To illustrate, in the previous example, we estimated the growth in the number of potential updates as being between $n^{2.2}$ and $n^{2.8}$. The gap between these bounds is very large, and careful statistical analysis should be able to narrow the gap considerably.

Comparing Two Algorithms

Suppose that we want to compare two different algorithms $\mathcal{AL}_1$ and $\mathcal{AL}_2$ for solving the same problem and are interested in knowing which algorithm performs better asymptotically. We can apply the same methodology for assessing asymptotic bottleneck operations to compare different algorithms. Let $\alpha_S^1(k)$ and $\alpha_S^2(k)$ be the total expected number of representative operations performed by the algorithms $\mathcal{AL}_1$ and $\mathcal{AL}_2$ on instances of size k. We say that algorithm $\mathcal{AL}_1$ is asymptotically superior to algorithm $\mathcal{AL}_2$ if

$$\lim_{k \to \infty} \frac{\alpha_S^1(k)}{\alpha_S^2(k)} = 0.$$

Virtual Running Times

In Section 18.1 we have already mentioned some difficulties that arise when we use CPU times as an empirical measure for computational investigations. We might note yet one more disadvantage: CPU times force unnecessary rigidity in the testing of an algorithm. To collect the CPU times for an algorithm, we should conduct all of our tests on the same machine using the same compiler. Moreover, if the machine is a time-sharing machine, we should ideally solve all the test problems when the machine has a similar work load. As an additional complication, for large instances, paging might dominate CPU times.

We can overcome some of these drawbacks by using virtual times instead of CPU times. (We point out that the virtual time is not directly related to "virtual memory.") The virtual running time of an algorithm is a linear estimate of its CPU time obtained by using its representative operation counts. For example, the virtual running time $V(I)$ of the network simplex algorithm to solve instance I is given by

$$V(I) = c_5\alpha_E(I) + c_6\alpha_W(I) + c_7\alpha_P(I),$$

for a set of constants c_5, c_6, and c_7 selected so that $V(I)$ is the best possible estimate of the algorithm's actual running time CPU(I) on the problem instance I. One plausible way to determine the constants c_5, c_6, and c_7 is to use (multiple) regression analysis. To do so, we consider the points (CPU(I), $\alpha_E(I)$, $\alpha_W(I)$, $\alpha_P(I)$) generated by solving various individual instances and use regression analysis to determine the constants c_5, c_6, and c_7 that minimizes the expression $\sum_I (\text{CPU}(I) - V(I))^2$.

In our regression analysis we generated each of the points (CPU(I), $\alpha_E(I)$, $\alpha_W(I)$, $\alpha_P(I)$) by taking an average of five problem instances; we used the averages reported in Figure 18.1, so we found the best linear fit to the 5-point averages. We found that for the network simplex algorithm with the first eligible pivot rule, we could estimate the virtual running time of an instance I as follows:

$$V(I) = (\alpha_E(I) + 2\alpha_W(I) + \alpha_P(I))/69{,}000.$$

To obtain an idea of the goodness of this fit, in Figure 18.4 we plot the ratio $V(I)/\text{CPU}(I)$ for all the data points. We find that in 15 of 25 cases, the error is less than 3%. In each case the error is at most 7%. So for this example, we can use the virtual running time instead of the CPU time with remarkably little loss of accuracy.

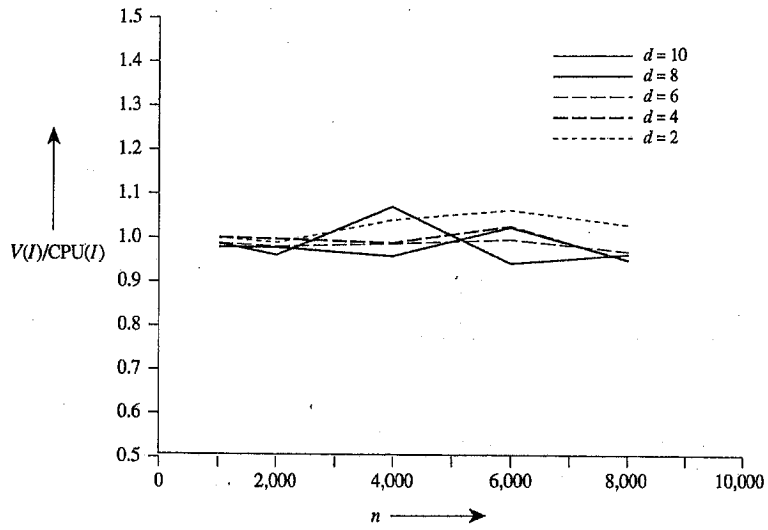


Figure 18.4 Determining how well the virtual running time estimates CPU time.

Using virtual running times has several advantages. First, the virtual running time helps us to assess the proportion of time that an algorithm spends on different representative operations. As an immediate consequence, it helps us to identify not only the asymptotic bottleneck operation, but also the bottleneck operations for different sizes of problem instances. For example, we estimated the virtual running time of the network simplex algorithm as $V(I) = (\alpha_E(I) + 2\alpha_W(I) + \alpha_P(I))/69{,}000$. To estimate the proportion of time spent on potential updates, in Figure 18.5 we plot $\alpha_P(I)/(\alpha_E(I) + 2\alpha_W(I) + \alpha_P(I))$. As we can see, for small problems, the percentage of time spent in updating node potential is less than 50 percent; however, as n increases the percentage of the time spent in updating node potential also increases, and it seems that as n approaches ∞, the percentage of time spent updating node potentials approaches 100 percent.

A second advantage of virtual running time is that it is particularly well suited for situations in which the testing is carried out on more than one computer. This situation might arise for several reasons: For example, several users might be conducting computational experiments at different sites; or the same user might wish to conduct additional tests after upgrading from an old computer. This situation is very common in the research literature because authors often conduct additional experiments at the suggestion of a referee or editor. When we move from one computer system to another, the representative operation counts remain unchanged. Using the representative operation counts of the previous study and the constants of the new computer system (obtained through the regression estimate for the virtual running time), we can obtain the virtual running times for all problems of the previous study measured in terms of the new computer system.

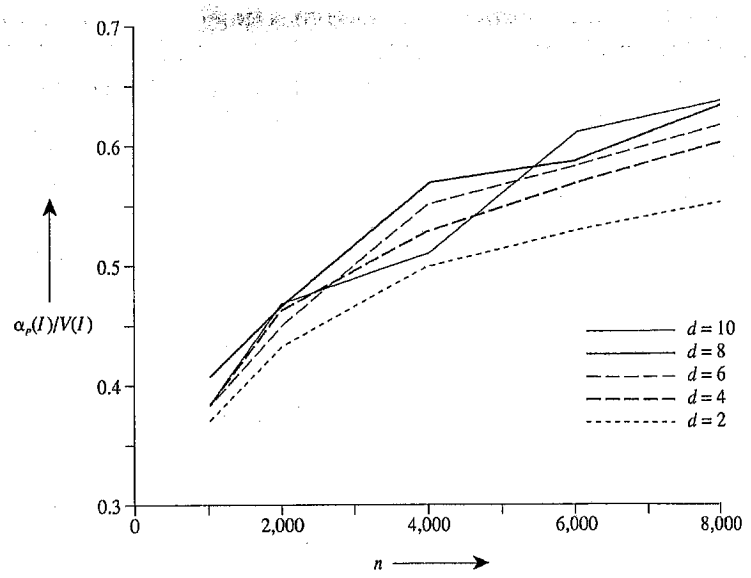


Figure 18.5 Identifying the percentage of the virtual running time accounted for by potential updates.

A third advantage of virtual running time is that it permits us to eliminate the effect of "paging" and "caching" in determining the running times. When a computer program executes a large program whose data do not fully fit into the computer's primary memory (e.g., RAM), it stores part of the data in the secondary memory (e.g., disk) with higher retrieval times. As a result, large programs run more slowly. In the virtual running time analysis, if we evaluate the constants using small problems (that fully execute in primary memory), the virtual running times for large problems would provide running time information as though the program were entirely run in primary memory. Thus we can use virtual running times to estimate the running time of an algorithm on a computer with sufficiently large primary memory.

A natural question is whether the constants used in the virtual running time are robust (i.e., do they give an accurate estimate of the CPU times for all possible problem inputs). Again, our use of virtual CPU times is not fully rigorous, but our limited experience so far has suggested that they do appear to be robust. In practice, we can also measure the robustness of the constants using statistical analysis.

Additional Insight into Algorithms

Computation counts have the potential to provide additional insight concerning an algorithm. For example, we can use the preceding analysis to estimate the number of pivots performed by the network simplex algorithm. Consider the following widely held belief in the linear programming literature: for most pivoting rules, the number of pivots is typically proportional to the number of constraints and rarely more than 3 times the number of constraints. In our case, the number of constraints is $n + m$ since we assume that each variable has an associated upper bound. Let us try to verify whether this assertion is valid for the first eligible pivot rule as applied to our

randomly generated problems. Notice that this rule would imply that if we plot the ratio of (number of pivots)/n versus n for any specific network density d, we should obtain nonincreasing functions. The plots given in Figure 18.6(a) indicate that this conjecture is not true: the number of pivots, for a fixed value of d, is not bounded by a linear function of n. However, if we plot (number of pivots)/n^2 for different network densities, then, as indicated by Figure 18.6(b), we do obtain nondecreasing functions. Therefore, the growth rate of the number of pivots is bounded from above by n^2 and bounded from below by n.

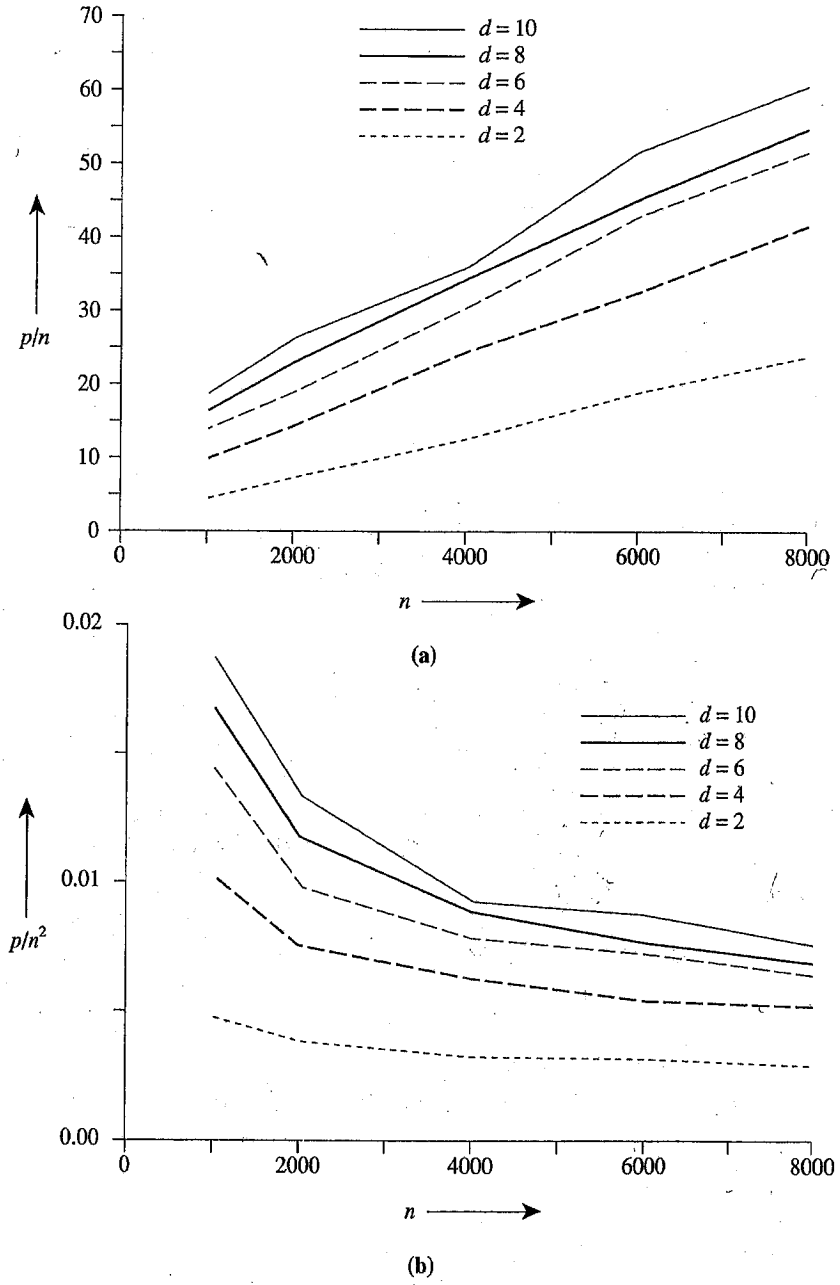


Figure 18.6 Determining lower and upper bounds on the growth rate of the number of pivots.

We might also be interested in determining what percentages of pivots are degenerate as the network size grows. The plots in Figure 18.7, which show the ratio of the number of degenerate pivots to the total number of pivots as a function of n, indicate that the ratio of degenerate pivots to total pivots varies between 70 and 90 percent.

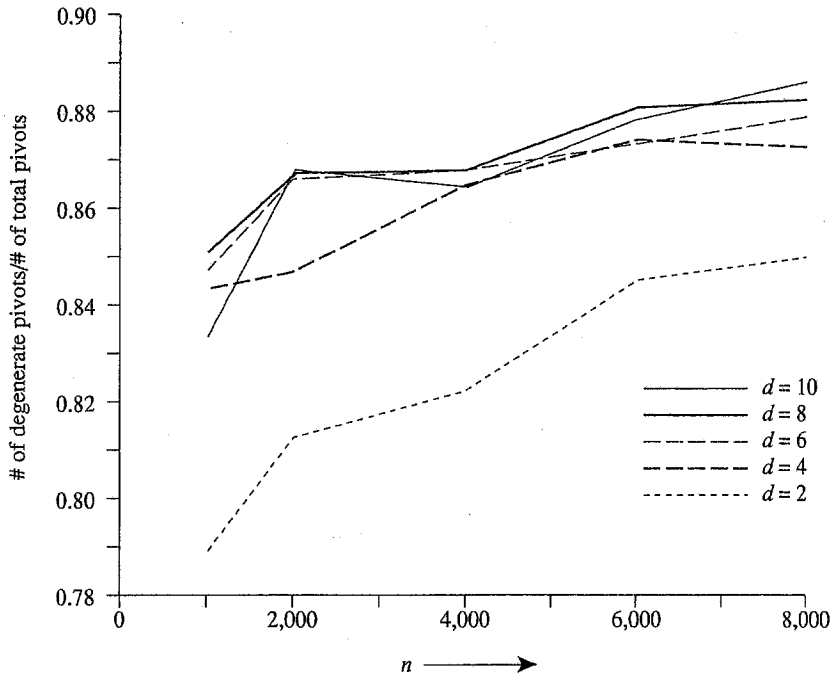


Figure 18.7 Occurance of degenerate pivots.

We might plot a few more graphs to gain additional insight about the network simplex algorithm. For example, we might plot the average size of the tree whose node potentials change during the pivot operation [i.e., $\alpha_P(I)/p$]. This plot would give us a good estimate of the running time per pivot since the potential update is the bottleneck operation, at least for the first entering pivot rule.

In addition, we might be interested in convergence results for the network simplex algorithm, i.e., how quickly does the network algorithm converge to the optimal objective function value? Does the algorithm quickly obtain a solution with a near-optimal objective function value and then slows down, or does it slowly approach the optimal objective function value and then converges rapidly? We could, in principle, provide a partial answer to this question by selecting a few sufficiently large instances and plotting the objective function values as a function of the number of pivots.

Limitations

We emphasize that the methodology we have described is suggestive and provides both insight and guidance, but it does not provide guarantees. In certain circumstances it can lead to incorrect conclusions. Let us illustrate a situation in which

this approach would underestimate the asymptotic growth rate of the bottleneck operation. Suppose that the actual growth rate of a bottleneck operation is $h(n) = n^2 + 1000n$ and we have data only for instances $n \leq 2000$. Suppose we conjecture that the growth rate of the function is $n^{1.7}$. When we plot the ratio $(n^2 + 1000n)/n^{1.7}$ for various values of n, we obtain the plot shown in Figure 18.8, which is a decreasing function of n. Therefore, using the methodology suggested earlier, we would incorrectly reach the conclusion that $n^{1.7}$ has an upper bound on $h(n)$.

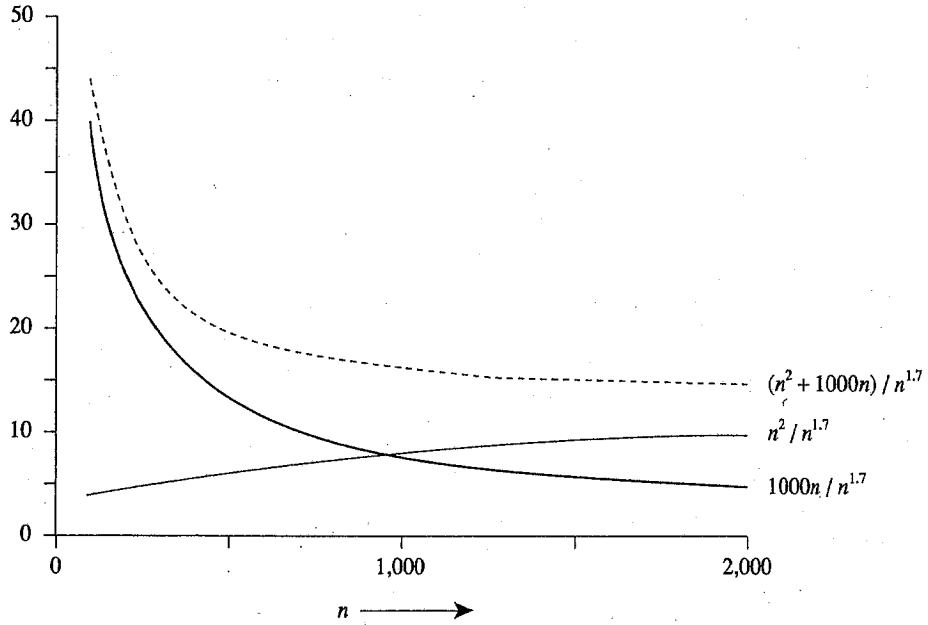


Figure 18.8 Limitation of the analysis using representative operation counts.

What went wrong in the previous example that would lead us to make such a significant mistake in estimating the running time? In our illustration, the growth rate had two components with very dissimilar constant terms and our test problems weren't sufficiently large so that the effect of the constant terms became insignificant. Had the growth function been $n^2 + 10n$ or had we solved problems of size $n = 100{,}000$, we would not have assessed the running time incorrectly, except possibly by a minimal amount. We anticipate that errors of this type would occur rather infrequently in practice.

We have suggested that we can determine an expression for the virtual running time using regression analysis. In general, regression analysis will misestimate the constant terms in the expression if the representative operation counts are highly correlated. In this case we might prefer to use more sophisticated methods of estimating the constants, possibly including computer timings of the basic operations.

We further point out that the methodology we have described is not useful for identifying improvements that affect only the constant factors. The use of computational counts will not identify improvements in coding or in data structures that improve the running time by a constant factor; the analysis of CPU times would be able to identify the effects of these improvements.

18.4 SUMMARY

Most iterative algorithms for solving network flow problems repetitively perform some basic operations. For almost all algorithms, we can decompose these basic operations into fundamental operations so that the algorithm executes each operation in $\Theta(1)$ time. An algorithm typically performs a large number of fundamental operations. We refer to a subset of fundamental operations as a set of representative operations if for every possible problem instance, the sum of representative operations provides an upper bound (to within a multiplicative constant) on the sum of all operations that an algorithm performs. We have shown that these representative operation counts might provide valuable information about an algorithm's behavior that is not captured by CPU time. For example, the representative operation counts allow us (1) to determine the asymptotic growth rate of the running time of an algorithm independent of its computing environment, (2) to assess the time an algorithm spends on different basic operations, (3) to compare two algorithms executed on different computers, and (4) to estimate the running time of an algorithm on a computer different from the one carrying out the experiments.

The simple methodologies that we have presented for conducting empirical analysis of an algorithm using representative operation counts do not provide rigorous guarantees; nevertheless, they often provide considerable insight about an algorithm's behavior, and they typically yield far more insight than is obtainable by analyzing only CPU times. The ideas we have outlined in this chapter apply to most network algorithms and should, as well, apply to optimization algorithms for problems arising in several other application domains.

REFERENCE NOTES

This chapter has been largely excerpted from the paper of Ahuja and Orlin [1992b]. We illustrated our ideas on computational testing of algorithms using a network simplex code developed by these authors. Researchers have tested several other codes of the network simplex algorithm; some notable computational studies are due to Glover, Karney, and Klingman [1974], Mulvey [1978], Bradley, Brown, and Graves [1977], Grigoriadis [1986], and Chang and Chen [1989]. In the computational results reported in this chapter, we used NETGEN to generate random network flow problems; Klingman, Napier, and Stutz [1974] developed this network generator.

In our discussion we have emphasized the value of operation counts in computational testing. The idea of using operation counts is quite old and probably dates back to the origins of computational experiments. Nevertheless, the literature on computational testing for mathematical programming has historically used CPU time as its primary measure of computational effort and has used operation counts in a rather limited way. (For example, most computational experiments on the network simplex algorithm have counted the number of pivots, but have not kept track in any systematic way of the work per pivot.) The thesis of McGeoch [1986] is an excellent reference that deemphasizes CPU time in favor of other measures of performance. The computational studies by Johnson [1990] and Bentley [1990] provide excellent illustrations of how to use both CPU times and representative operation

counts to analyze empirical behavior of algorithms. The term "virtual time" appears in the thesis of Brown [1988] and is used in a similar way in this chapter.

Although we have focused on the use of operation counts in this chapter, the following references offer insight about other very important aspects of computational testing.

Performance measures. In this chapter we have focused on empirical running time as measured both by representative operation counts and by CPU times. Other important measures of performance for an algorithm include (1) ease of implementation, (2) robustness, (3) reliability, and (4) accuracy of the solutions. The papers by Crowder and Saunders [1980], Hoffman and Jackson [1982], and Greenberg [1990] discuss these measures of performance.

Reporting computational experiments. The most comprehensive references on the reporting of computational experiments in mathematical programming is Crowder, Dembo, and Mulvey [1978, 1979]. The authors provide guidelines for what should be reported in a research paper and offer advice on how to conduct appropriate computational experiments. Jackson and Mulvey [1978] have summarized the reporting of computational experiments within the mathematical programming literature, largely detailing how poor the reporting had been up to that time. More recently, Jackson, Boggs, Nash, and Powell [1989] have provided updated guidelines for conducting computational experiments.

Analysis and evaluation of test results. As emphasized in this chapter, a large part of data analysis can be carried out without the formal use of statistical methodology. Graphs, charts, and elementary statistics such as the computation of means and standard deviations often provide significant insights into the performance of algorithms. McGeoch [1986], Bentley [1990], and Johnson [1990] present several case studies to show the power of these basic analytical tools.

Statistical methodologies. Often, statistical methodologies can provide analysis and insight that is unavailable through other means. For some papers on computational testing that provide details on the use of statistical methodologies, see Bland and Jensen [1985] and Golden, Assad, Wasil, and Baker [1986]. Moreover, statistical methodologies such as *variance reduction* can often increase the power of the analysis and reduce the number of experiments needed to obtain conclusive results. We refer the reader to McGeoch [1992] for excellent illustrations of variance as well as for pointers to the literature.

Algorithm animation. Algorithm animation is another technique for gaining insight about an algorithm. In his dissertation, Brown [1988] provides an excellent treatment of this topic. Algorithm animation techniques view the progress of an algorithm on a single instance as a sequence of snapshots. For example, consider the greedy algorithm for finding the minimum spanning tree joining n points in the plane. This algorithm creates the minimum spanning tree one arc at a time, maintaining a forest at each intermediate stage until it ultimately creates the minimum

cost spanning tree. Using animation, we could construct a sequence of figures showing the forest at intermediate stages. Viewing a motion picture consisting of a series of these snapshots might reveal additional insight about an algorithm. Alternatively, we could see how the total length of the forest increases as a function of computer time, or we could see how the size of the components decreases as a function of computer time. The papers by Bentley and Kernighan [1990] and Bentley [1990] discuss a simple language that facilitates the construction of these animations.

EXERCISES

18.1. An algorithm is said to have a *predictable running time* if its empirical running time is guaranteed to be within a constant factor of its worst-case running time. State which of the following algorithms have a predictable running time: (1) the breadth-first search algorithm discussed in Section 3.4; (2) the original implementation of Dijkstra's algorithm discussed in Section 4.5; (3) Dial's implementation of Dijkstra's algorithm discussed in Section 4.6; (4) the radix heap implementation of Dijkstra's algorithm discussed in Section 4.8; (5) the $O(nm)$-time cycle detection algorithm discussed in Section 5.5; and (6) the minimum mean cycle algorithm discussed in Section 5.7. Justify your answers by theoretical arguments without doing the computational testing.

18.2. Specify a set of representative operations for each of the following algorithms: (1) the topological sorting algorithm discussed in Section 3.4; (2) the binary heap implementation of Dijkstra's algorithm discussed in Section 4.7; and (3) the radix heap algorithm described in Section 4.8. Justify your answers.

18.3. Give a set of representative operations for the following maximum flow algorithms discussed in Chapter 7: (1) the shortest augmenting path algorithm; (2) the highest-label preflow-push algorithm; and (3) the excess scaling algorithm. For each algorithm, obtain a set of representative operations with the fewest possible number of operations and justify your answer.

18.4. Specify a set of representative operations for the following minimum cost flow algorithms: (1) the relaxation algorithm described in Section 9.10; (2) the cost scaling algorithm described in Section 10.3; and (3) the double scaling algorithm described in Section 10.4.

18.5. Give a set of representative operations for the following minimum spanning tree algorithms: (1) the $O(m + n \log n)$ time implementation of Kruskal's algorithm (assume that arcs are already sorted); and (2) the $O(m \log n)$ time implementation of Sollin's algorithm. Justify your answers.

18.6. What are a set of representative operations for the generalized network simplex algorithm discussed in Chapter 15 and for the Dantzig–Wolfe decomposition algorithm discussed in Chapter 17?

18.7. Let operation i and operation j be two operations that require $\Theta(1)$ time in a particular algorithm. Suppose that when this algorithm is applied to an instance I, it executes operation i $\alpha_i(I)$ times and operation j $\alpha_j(I)$ times. We say that operation j dominates operation i if for every possible instance I, $\alpha_i(I) \leq c_1\alpha_j(I)$ for some known constant c_1. For each of the algorithms mentioned in Exercises 18.1 and 18.2, specify representative operations as well as a nonrepresentative operation that it dominates.

18.8. Design a computational experiment for comparing the following implementations of the shortest path problem discussed in Chapter 4: (1) the original implementation; (2) Dial's implementation; and (3) the radix heap implementation. Which set of representative operations would you collect for each algorithm? Write a computer code for each of these algorithms and test them using the methodology described in this chapter.

18.9. Write computer programs for the following maximum flow algorithms and compare them using the methodology described in this chapter: (**1**) the labeling algorithm; (**2**) the capacity scaling algorithm; and (**3**) the shortest augmenting path algorithm.

18.10. Write computer programs for the following minimum cost flow algorithms and compare them using the methodology described in this chapter: (**1**) the cycle-canceling algorithm; (**2**) the successive shortest path algorithm; and (**3**) the relaxation algorithm.

19

ADDITIONAL APPLICATIONS

Mens et Manus (Mind and Hand)
—The MIT Motto

Chapter Outline

19.1 Introduction
19.2 Maximum Weight Closure of a Graph
19.3 Data Scaling
19.4 Science Applications
19.5 Project Management
19.6 Dynamic Flows
19.7 Arc Routing Problems
19.8 Facility Layout and Location
19.9 Production and Inventory Planning
19.10 Summary

19.1 INTRODUCTION

As we have noted throughout this book, network flows is a topic that has evolved in the best tradition of applied mathematics: It is a subject matter that poses considerable challenges for modeling and algorithm development and it has a core of substantial theory and scholarly content. It unites ideas from the abstract world of mathematics and concrete world of computation, and so draws its intellectual heritage from several disciplines, including applied mathematics, computer science, engineering, management science, and operations research. But perhaps as important, it is a subject that has had numerous applications in a wide variety of practical problem settings.

As indicated by the title of this book, our coverage has attempted to emphasize three critical ingredients of network flows: theory, algorithms, and applications. Although we have organized most of our discussion in the previous chapters around core network models and algorithmic approaches, as a key element of our discussion, we have described approximately 150 applications and mentioned dozens more. We have introduced these applications in the context of core network flow models—shortest path problems, maximum flow problems, minimum cost flows—and other network optimization models such as minimal spanning trees. In this chapter we adopt a somewhat different approach. We discuss 24 applications, organized around application type. Therefore, within most of the topics that we introduce, we consider

several different network optimization models. Adopting this approach permits us to see a number of important applications in a somewhat different light: at times, building from simple to more complex models all within the same application context.

At the end of Chapters 4, 6, 9, and 12 to 17 we have listed, with references, a great many applications, including both those that we have considered in the text and in the exercises and others from the literature. At the end of this chapter, we offer another view of these applications, including those that we have considered in this chapter, organized in the following categories:

Applied mathematics
Computer science and communication systems
Defense
Distribution systems and transportation
Engineering
Management science
Manufacturing, production, and inventory planning
Physical and medical sciences
Scheduling
Social sciences and public policy

Although we could have adopted many alternative ways to categorize these various applications, this topography provides one useful view of network flows in practice. Some of the categories, such as manufacturing and transportation, refer to specific industries; in these categories, the applications typically are models of direct relevance to practitioners. Applied mathematics represents another type of category; the applications in this case are mathematical problems of some interest to the applied mathematics community (e.g., finding solutions to certain systems of equations and inequalities) and which often are generic models with rich end applications of their own. When viewed in its entirety, this list of applications attests to the remarkable robustness of network flows as a practical modeling tool; it suggests that we might revise the opening sentence in this book and state: "Everywhere we look, not only in our daily lives, but also in the worlds of commerce, science, social systems, and technology, networks are apparent."

Figure 19.1 summarizes the applications that we consider in this chapter as well as the network problems used to model these applications. As indicated by this table, just the applications in this chapter attest to the richness of network flows in practice. As shown by this table, this collection of applications uses many of the network models that we have developed in previous chapters, including the core shortest path, maximum flow, and minimum cost flow models, as well as minimum spanning trees, matchings, and multicommodity flows. The applications also use more specialized models that are transformable into network flow models (duals of minimum cost flow models).

Application	Problem type
19.1 Open pit mining	Minimum cut problem
19.2 Selecting freight handling terminals	Minimum cut problem
19.3 Optimal destruction of military targets	Minimum cut problem
19.4 Flyaway kit problem	Minimum cut problem
19.5 Asymmetric data scaling with lower and upper bounds	Shortest path problem
19.6 Minimum ratio asymmetric data scaling	Minimum mean cycle problem
19.7 DNA sequence alignment	Shortest path problem
19.8 Automatic karyotyping of chromosomes	Transportation problem
19.9 Determining minimum project duration	Longest path problem
19.10 Just-in-time scheduling	Longest path problem, minimum cost flow problem
19.11 Time–cost trade-off in project management	Dual of minimum cost flow problem
19.12 Maximum dynamic flows	Maximum flow problem
19.13 Models for building evacuation	Minimum cost flow problem
19.14 Directed Chinese postman problem	Minimum cost flow problem
19.15 Undirected Chinese postman problem	Shortest path problem and nonbipartite matching problem
19.16 Discrete location problems	Assignment problem
19.17 Warehouse layout	Transportation problem
19.18 Rectilinear distance facility location	Dual of minimum cost flow problem
19.19 Dynamic lot sizing	Shortest path problem, minimum cost flow problem
19.20 Dynamic lot sizing with concave costs	Shortest path problem
19.21 Dynamic lot sizing with backorders	Shortest path problem
19.22 Multistage production–inventory planning	Minimum cost flow problem, mixed integer programs
19.23 Multiproduct multistage production–inventory planning	Integer multicommodity flow problem
19.24 Mold allocation	Minimum cost flow problem

Figure 19.1 Applications considered in this chapter.

19.2 MAXIMUM WEIGHT CLOSURE OF A GRAPH

One of the primary purposes of scientific investigation is to structure the world around us, discovering patterns that cut across and therefore help to unify varied applied contexts. To begin our discussion of applications, in this section we examine one such generic model that has applications as varied as designing mining operations, scheduling freight handling terminals, developing a strategy for destroying military targets, and designing optimal kits of parts and tools for field repair crews.

A *closure* of a directed network $G = (N, A)$ is a subset of nodes without any outgoing arcs, that is, a subset $N_1 \subseteq N$ satisfying the property that if i belongs to N_1 and $(i, j) \in A$, then j also belongs to N_1. A closure might have more than one

component. Suppose that we associate a node weight w_i (of arbitrary sign) with each node i of G. In the *maximum weight closure problem*, we wish to find a closure N_1 with the largest possible weight $w(N_1)$ defined as $w(N_1) = \sum_{i \in N_1} w_i$. As an example, the network shown in Figure 19.2(a) has the closures {3, 4, 5}, {4, 5}, {5}, {2, 5}, and {1, 2, 4, 5}; the maximum weight closure for this network is {3, 4, 5}.

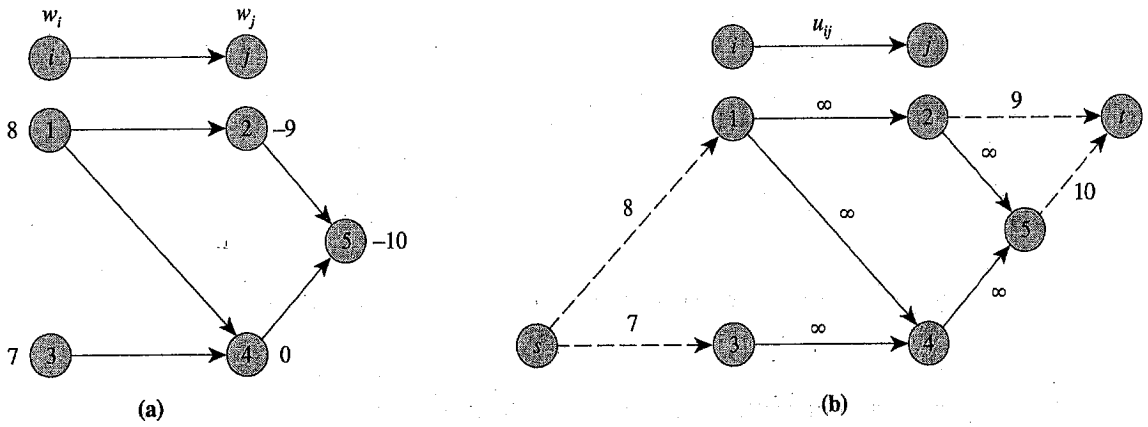


Figure 19.2 (a) Maximum weight closure problem; (b) transformed network G'.

As we will see in a moment, the maximum weight closure problem arises in a variety of applications. Before discussing these applications, let us show how to transform the maximum weight closure problem defined on the network $G = (N, A)$ into a maximum flow problem on a slightly augmented network $G' = (N', A')$. To define G', we introduce a source node s and for each node $i \in N$ with $w_i > 0$, we create an arc (s, i) with capacity w_i. We also introduce a sink node t and for each node $i \in N$ with $w_i < 0$, we create an arc (i, t) with capacity $-w_i$. We then set the capacity of every original arc $(i, j) \in A$ equal to ∞ (any integer greater than $\sum_{i \in N} | w_i |$ would suffice). Figure 19.2(b) shows the transformed network for the maximum weight closure problem shown in Figure 19.2(a).

We refer to an s–t cut in the transformed network $G' = (N', A')$ as a *simple cut* if all its forward arcs are source and sink arcs (i.e., arcs incident to the source and sink nodes). We claim that there is a one-to-one correspondence between closures of G and simple cuts in G'. To establish this result, note that if N_1 is a closure of G, its corresponding cut is $[S, \bar{S}]$ with $S = \{s\} \cup N_1$. The fact that N_1 is a closure of G implies that no arc in A is a forward arc of the cut $[S, \bar{S}]$. Consequently, all of the forward arcs in the cut $[S, \bar{S}]$ will be either source arcs or sink arcs, so this cut will be a simple cut. Similarly, if $[S, \bar{S}]$ is a simple cut of G', the subset of nodes N_1 defined by $N_1 = S - \{s\}$ is a closure of G.

To relate the weight of a closure to the capacity of the corresponding cut, let N_1 be a closure of G and $N_2 = N - N_1$. In addition, let N_1^+ denote the nodes with nonnegative weights in N_1, and N_1^- denote the nodes with negative weights in N_1. We define N_2^+ and N_2^- similarly. By definition, the weight of the closure N_1 is

$$w(N_1) = \sum_{i \in N_1^+} w_i - \sum_{i \in N_1^-} | w_i |. \tag{19.1}$$

Now consider the simple cut $[S, \bar{S}]$ corresponding to the closure N_1. Each forward arc in the cut is a source or a sink arc. The construction of the network G' implies that this cut would have a forward arc (i, t) for every $i \in N_1^-$ and a forward arc (s, i) for every $i \in N_2^+$. Therefore, the capacity of this cut is

$$u[S, \bar{S}] = \sum_{i \in N_2^+} w_i + \sum_{i \in N_1^-} |w_i|. \tag{19.2}$$

Adding (19.1) and (19.2), we find that

$$w(N_1) + u[S, \bar{S}] = \sum_{i \in N_1^+} w_i + \sum_{i \in N_2^+} w_i = \bar{w}.$$

The constant $\bar{w}$ in this expression is the total weight of all nodes. Consequently, if $[S, \bar{S}]$ is a minimum capacity simple cut, the corresponding closure N_1 is a maximum weight closure.

To obtain a minimum capacity simple cut, we simply need to find a minimum capacity cut in the network since in the transformed network G', no arc in A will be a forward arc in any minimum cut because the arcs in A all have an infinite capacity. Therefore, a minimum capacity cut of G' will automatically be a simple cut.

To conclude this introductory discussion of the maximum weight closure problem, we note that we can also derive the network formulation of the problem using minimum cost flow duality, since we can formulate the maximum weight closure problem as a linear program with at most one $+1$ and at most one -1 in each row, which is the dual of a minimum cost flow problem (as shown in Theorem 9.9).

We next describe four different applications of the maximum weight closure problem.

Application 19.1 Open Pit Mining

In mining operations, a problem of considerable importance is the determination of the optimal contour of an open pit mine. In an open pit mine, we might divide the potential mining region into blocks. The provisions of any given mining technology, and perhaps the geography of the mine, impose restrictions on how we can remove the blocks. For example, we can never remove a block until we have removed every block that lies immediately above it (see Figure 19.3); restrictions on the "angle" of mining the blocks might impose similar precedence conditions. Moreover, every block i has an economic measure w_i representing the net profit obtained from removing that block (value of the ore contained in the block minus the cost of exploiting and processing the block). In the open pit mining problem, we wish to identify a set of blocks that maximizes the net profit. We model this problem as a maximum weight closure problem by representing each block as a node; if we must remove block j before removing block i, we include the arc (i, j) in the network. If we want to remove a contour B of blocks, every block that we need to remove before removing a block in B must also lie in B. That is, the nodes defined by B have no outgoing arcs and therefore define a closure of the network.

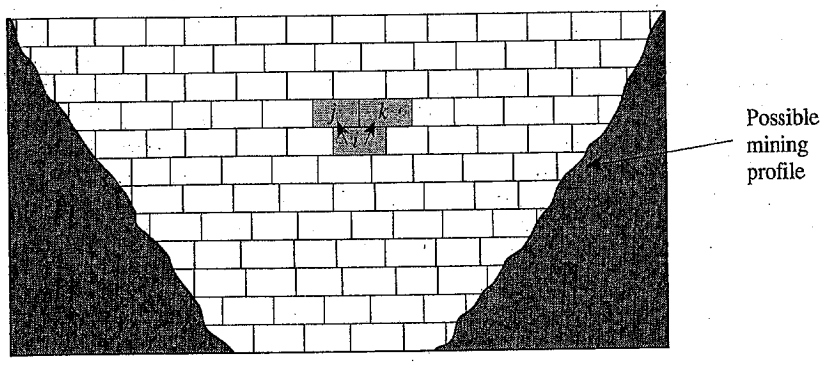


Figure 19.3 Open pit mine; we must remove blocks j and k before removing block i.

Application 19.2 Selecting Freight Handling Terminals

A transport company is considering the installation of a number of freight handling terminals. It wants to choose from a set S of possible locations for the terminals. The company has the potential to attract market share (which is a given amount of demand) between some of the pairs of terminals. To satisfy the demand between locations i and j, the company must locate terminals at both of these locations. Suppose that c_j is the cost of installing a terminal at location j and that p_{ij} is the profit obtained by satisfying the demand between locations i and j. The transport company would like to determine where to install terminals in order to maximize its net profit (i.e., the revenue obtained from satisfying the demands minus the cost of installing the terminals).

Consider, for example, the network shown in Figure 19.4(a). Each node in this

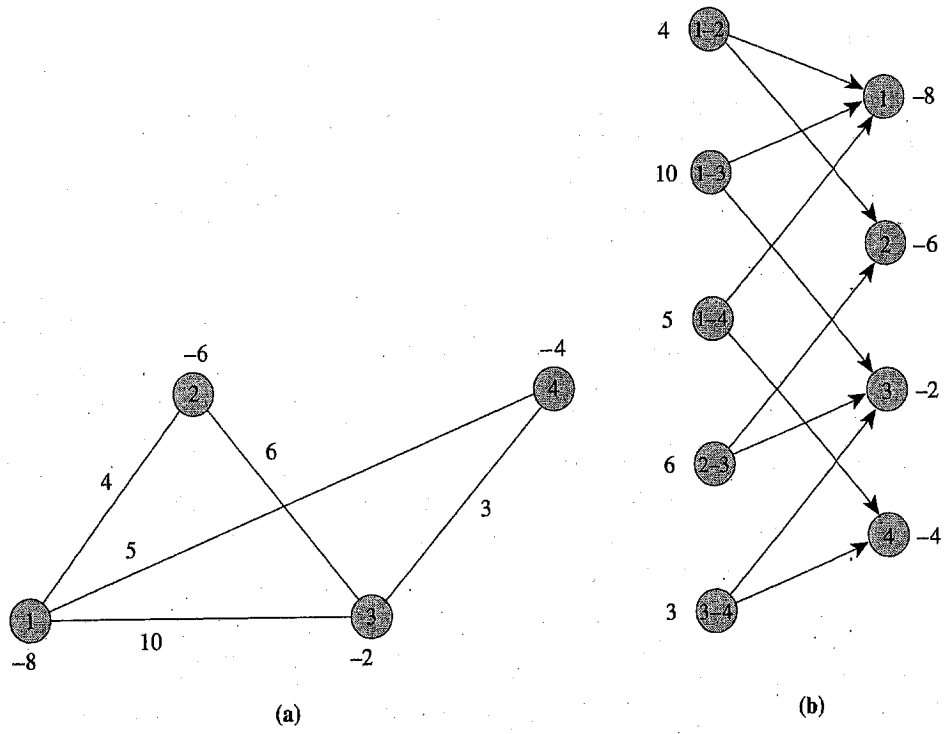


Figure 19.4 (a) Selection problem; (b) corresponding maximum closure problem.

network represents a potential terminal; the number next to it represents the negative of the cost of installing that terminal. Each arc (i, j) represents a service that can be operated only if both the terminals i and j are operating; the number next to the arc represents the profit obtained by operating that service. For example, if we decide to operate terminals 1, 2, and 3, we can operate services only between the following pairs of terminals: (1, 2), (1, 3), and (2, 3). The net profit for this selection is 4, which is the difference of the total revenues (of value 20) and the total installation cost (which is 16).

To reduce the selection problem to the maximum weight closure problem, we define a bipartite network $G = (N_1 \cup N_2, A)$ with a node in N_1 for every service and a node in N_2 for every terminal. The service node representing the service between nodes i and j has two outgoing arcs entering the nodes representing terminals i and j, implying that whenever we decide to provide the service between the nodes i and j and accrue the profit p_{ij}, we must install the terminals at these two nodes and incur the installation cost c_i and c_j. Figure 19.4(b) shows the resulting maximum weight closure problem for our example.

Application 19.3 Optimal Destruction of Military Targets

A military commander has identified a set S of military targets that he wants to destroy. These targets are heavily defended by four different layers of defense. The first layer consists of *forward air defense sites* (FADS), the second layer consists of *band surface to air missiles* (BSAM), the third layer consists of *airborne interceptors* (AI), and the fourth layer consists of *terminal surface to air missiles* (TSAM); a fifth layer contains the military targets themselves. Let $\overline{S}$ denote the set of all defense sites. Each military target is protected by some, but not necessarily all, of these defense sites. A defense site might also provide protection to other defense sites in lower-numbered layers. Let $D(i)$ denote the set of defense sites that protect the target or defense site $i \in S \cup \overline{S}$.

Based on his past experience, the military commander feels that while it might be possible for missiles to pass through all the defenses to reach the military targets, the probability of such a "leakage" is quite small. Instead, he believes that to destroy a target, he must first destroy all the defense sites that protect it. Therefore, he must destroy defense sites as well as targets. Destroying the ith target or defense site has a certain military benefit but also incurs some loss. Let w_i denote the benefit minus the loss, which is the (net) value, of destroying the ith target or defense site. The military commander wants to identify a set of targets and defense sites with the largest possible total value.

To formulate this problem as a maximum weight closure problem, we associate a node i with a weight of w_i with the ith target or defense site. For each $i \in S \cup \overline{S}$, we introduce an arc (i, j) for every $j \in D(i)$. Figure 19.5 gives an example of the resulting network. As is easy to see, every feasible destruction of targets and defense sites corresponds to a closure of the network. Therefore, the military commander's problem is a maximum weight closure problem.

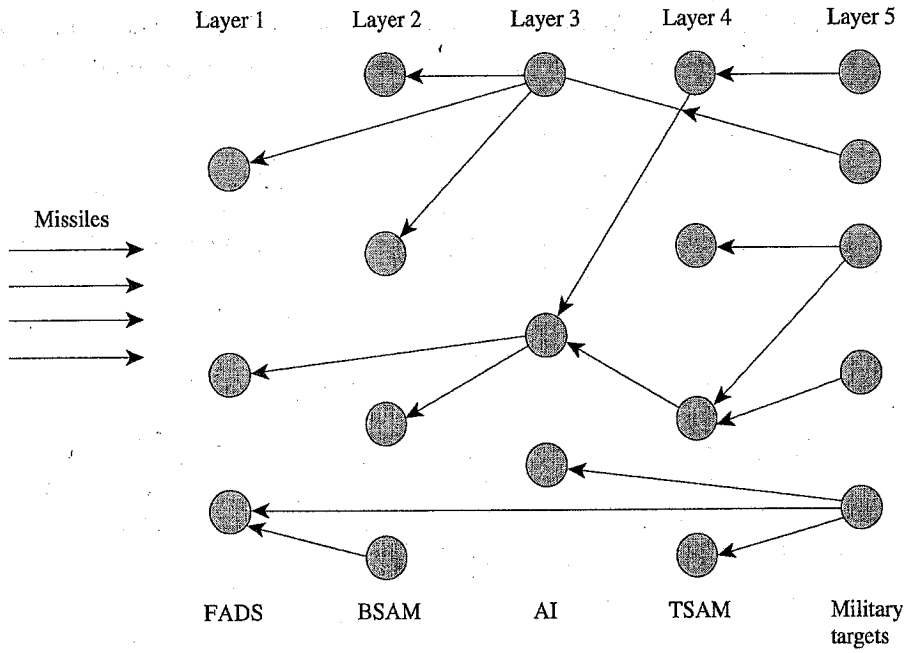


Figure 19.5 Formulation for the optimal destruction of military targets.

Application 19.4 Flyaway Kit Problem

Many companies (e.g., computer companies or telephone companies) own, lease, or warrantee a wide range of equipment that they must maintain at geographically dispersed field locations. In performing a given job, the repair crew often require various types of parts (and tools). In many cases the crews carry some replacement parts in a *kit* rather than storing them at the equipment site. If all the required parts are in the kit, the crew member can repair the equipment. But if any of these items is not available, the service call is incomplete and the job is a "broken job." Broken jobs are costly for several reasons: (1) they increase equipment downtime, (2) the repair crew must make an extra trip for parts, and (3) partially repaired equipment might be unsafe or vulnerable to damage. On the other hand, carrying more items in the kit increases handling and inventory costs. In the *flyaway kit problem*, we need to obtain the optimal kit of parts (and tools) that minimizes the sum of the handling and inventory costs and the costs of broken jobs.

Suppose that we number the parts required for servicing the jobs as $1, 2, \ldots, r$. We assume that the repairman restocks the kit between jobs, but with a fixed and specified content. For our purposes we define a job by the set of parts (and tools) that it requires. Making this association defines a collection of job types $J_1, J_2, \ldots, J_l$, that encompasses all the known possibilities that a repairman might encounter. The job type J_j is defined by the set B_j of the parts required by that job. Let l_j denote the expected number of job types J_j serviced in one year, and V_j denote the penalty cost we incur whenever job j is a broken job.

A *stocking policy* of a kit consists of a fixed set of parts $M \subseteq \{1, 2, \ldots, r\}$ that a crew would carry. Let H_i denote the yearly handling and inventory cost for carrying part i in the kit. Then the total handling cost is $\sum_{i \in M} H_i$. Moreover,

the total expected cost of broken jobs per kit per year for policy M would be $\sum_{\{j:B_j \not\subseteq M\}} V_j l_j$. Therefore, policy M incurs a total expected yearly cost per kit of

$$z(M) = \sum_{i \in M} H_i + \sum_{\{j:B_j \not\subseteq M\}} L_j.$$

In this expression, $L_j = V_j l_j$. The optimal policy would, of course, be a set $M \subseteq \{1, 2, \ldots, r\}$ that minimizes $z(M)$. Notice that minimizing $z(M)$ is equivalent to maximizing $-z(M)$, which we can restate as

$$-z(M) = \sum_{\{j:B_j \subseteq M\}} L_j - \sum_{i \in M} H_i - L,$$

by letting $L = \sum_{j=1}^{l} L_j$, a constant. Consequently, our objective is to identify a policy M that maximizes $\sum_{\{j:B_j \subseteq M\}} L_j - \sum_{i \in M} H_i$. This problem is a special case of the maximum weight closure problem on the bipartite network shown in Figure 19.6. This network contains two types of nodes: those representing parts and those representing jobs. It also contains an arc from a node representing job type J_j to each part node in B_j. Notice that a node J_j can be in the maximum weight closure only if the closure also contains each part (and tool) in B_j.

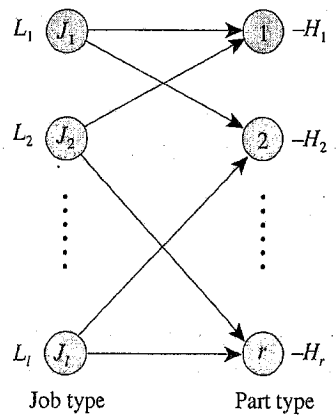


Figure 19.6 Network for the fly away kit problem.

19.3 DATA SCALING

In many applied problem contexts, we wish to modify matrix data to achieve some desired objective. In Application 6.3 we examined one instance of this generic problem: the rounding of entries in census tables to preserve the basic nature of the tabular entries and yet disguise confidential information. In this section we consider two variations of a model of data scaling: We would now like to scale the rows and columns of a matrix so that the resulting entries of the matrix are "close" to each other; one variant of this model requires that we solve a shortest path problem and the other requires that we solve a minimum mean cycle problem. Before discussing these generic models, we first consider one important application context for data scaling.

When we solve a linear programming problem of the form minimize cx, subject to $Ax = b$ and $x \geq 0$ by the simplex method, we incur round-off errors because computers perform arithmetic operations in floating-point arithmetic. The magnitude of these errors depends on the relative sizes of the numbers. If the numbers being

manipulated are comparable, the round-off errors are relatively small; otherwise, they are larger. Thus, given a linear program, it is often desirable to transform it into an equivalent linear program whose constraint matrix $\mathbf{A} = \{a_{ij}\}$ has elements that are as close as possible to each other. We can achieve this objective by *data scaling*: that is, multiplying each row i by a positive constant α_i and dividing each column j by a positive constant β_j. Notice that multiplying the equality (row) i by the constant α_i does not affect the feasibility of any solution, and dividing a column j by a constant β_j is equivalent to replacing the variable x_j by the variable $x'_j = x_j\beta_j$. Consequently, other than rescaling the variables, this transformation does not affect the optimal solutions of the linear program.

We now study two data scaling problems and, in each case, reduce the problem to a network flow model.

Application 19.5 Asymmetric Data Scaling with Lower and Upper Bounds

In the asymmetric data scaling problem with lower and upper bounds, we want to determine whether we can find row multipliers α_i and column divisors β_j so that every scaled entry of a matrix has a value between the prescribed lower and upper bounds l and u, that is,

$$l \le \alpha_i \mid a_{ij} \mid /\beta_j \le u \qquad \text{for each } i = 1, \ldots, p \text{ and each } j = 1, \ldots, q. \quad (19.3)$$

If we take the logarithms of both sides of the inequalities in (19.3), they become

$$\log l \le \log \alpha_i + \log \mid a_{ij} \mid - \log \beta_j \le \log u \qquad (19.4)$$

$$\text{for each } i = 1, \ldots, p, \text{ and each } j = 1, \ldots, q.$$

For notational convenience, let us index the rows from 1 to p and index the columns from $p + 1$ to $p + q$. Because of this numbering convention, we subsequently refer to β_j by β_{p+j} and to a matrix element a_{ij} by $a_{i,p+j}$. Let

$$\pi(i) = \log \alpha_i \qquad \text{for each } i = 1, \ldots, p,$$

$$\pi(j) = \log \beta_j \qquad \text{for each } j = p + 1, \ldots, p + q.$$

Also, let $l' = \log l$, $u' = \log u$, and $a'_{ij} = \log \mid a_{ij} \mid$. Then we can rewrite the two inequalities in (19.4) as

$$\pi(i) - \pi(j) \le u' - a'_{ij} \text{ for each } i = 1, \ldots, p \text{ and } j = p + 1, \ldots, p + q, \quad (19.5a)$$

$$-\pi(i) + \pi(j) \le a'_{ij} - l' \text{ for each } i = 1, \ldots, p \text{ and } j = p + 1, \ldots, p + q. \quad (19.5b)$$

We have thus reduced the data scaling problem into the problem of identifying whether some vector π satisfies the inequalities in (19.5). This problem is known as a *system of difference constraints* and, as described in Application 4.5, we can solve it as a shortest path problem on the network shown in Figure 19.7. This network is a complete bipartite network $G = (N_1 \cup N_2, A)$ with node sets $N_1 = \{1, \ldots, p\}$ and $N_2 = \{p + 1, \ldots, p + q\}$. The network contains an arc (i, j) of cost $c_{ij} = u' - a'_{ij}$ for every node pair $[i, j] \in N_1 \times N_2$, and an arc (j, i) of cost $c_{ji} = a'_{ij} - l'$

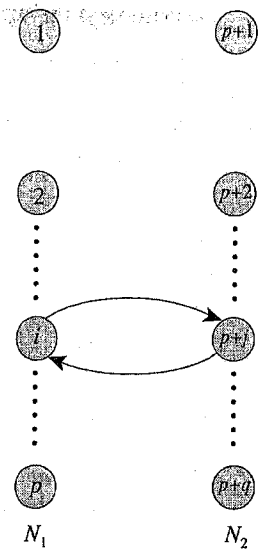


Figure 19.7 Network for data scaling problems. The network contains an arc for each $[i, j] \in N_1 \times N_2$ and an arc for each $[j, i] \in N_2 \times N_1$.

for every node pair $[j, i] \in N_2 \times N_1$. If we set $d(j) = -\pi(j)$ for each j, then the inequalities (19.5) reduce to

$$d(j) \leq d(i) + c_{ij} \qquad \text{for each arc } (i, j) \in A, \tag{19.6}$$

which are the optimality conditions of a shortest path problem (see Section 5.2). We have shown in Section 5.2 that (19.6) has a feasible solution [i.e., a set of shortest path distances $d(\cdot)$] if and only if the network contains no negative cycle. We can resolve this question in $O(nm)$ time by using the negative cycle detection algorithm described in Section 5.5.

Application 19.6 Minimum Ratio Asymmetric Data Scaling

In the minimum ratio asymmetric data scaling problem, we want to identify bounds l and u, row multipliers α_i, and column divisors β_j so that

$$l \leq \alpha_i \mid a_{ij} \mid / \beta_j \leq u \qquad \text{for each } i = 1, \ldots, p \text{ and } j = 1, \ldots, q, \tag{19.7}$$

with the smallest possible ratio u/l. We will show how to solve this data scaling problem by solving a minimum mean cycle problem.

We first observe that without any loss of generality we can fix $l = 1$ and identify the minimum value of u that, together with some row and column multipliers, satisfies the inequalities (19.7) (because we can transform any solution with l different than 1 to a solution with l equal to 1 by multiplying u and each α_i by l). Our discussion of the preceding application shows that the following linear program is an alternative formulation of the minimum ratio asymmetric data scaling problem:

$$\text{Minimize} \quad u' \tag{19.8a}$$

subject to

$$\pi(i) - \pi(j) - u' \leq -a'_{ij} \text{ for every } i = 1, \ldots, p, \text{ and } j = p + 1, \ldots, p + q, \tag{19.8b}$$

$$-\pi(i) + \pi(j) \leq a'_{ij} \qquad \text{for every } i = 1, \ldots, p \text{ and } j = p + 1, \ldots, p + q. \tag{19.8c}$$

In this model the variables $\pi(i)$ and constants u' and a'_{ij} are defined as in Application 19.5. Notice that $l' = \log_2 1 = 0$. To simplify (19.8), we again redefine the variables. Let $\theta = -u'/2$, $\pi'(i) = \pi(i) - u/2$ for each $i = 1, \ldots, p$, and $\pi'(j) = \pi(j)$ for each $j = p + 1, \ldots, p + q$. Moreover, for each $i = 1, \ldots, p$ and $j = p + 1, \ldots, p + q$, define $c_{ij} = -a'_{ij}$ and $c_{ji} = a'_{ij}$. In this notation, the problem (19.8) becomes the following linear program:

$$\text{Minimize} \quad \theta \tag{19.9a}$$

subject to

$$\pi'(i) - \pi'(j) + \theta \leq c_{ij} \quad \text{for every } i = 1, \ldots, p \text{ and } j = p + 1, \ldots, p + q, \tag{19.9b}$$

$$-\pi'(i) + \pi'(j) + \theta \leq c_{ji} \text{ for every } i = 1, \ldots, p \text{ and } j = p + 1, \ldots, p + q. \tag{19.9c}$$

This problem is similar to the one addressed in the preceding application; in this case we want to find the minimum value of θ for which the system of difference constraints (19.9b) and (19.9c) has a feasible solution. Our discussion in the last application implies that the system of inequalities (19.9b) and (19.9c) has a feasible solution if and only if the network shown in Figure 19.7, with $c_{ij} - \theta$ as the length of each arc (i, j), does not contain a negative cycle. The latter statement is equivalent to saying that the network does not contain a negative cycle with mean θ. Therefore, determining the minimum value of θ for which the network contains no negative cycle corresponds to determining the minimum mean cycle of the graph in Figure 19.7. In Section 5.7, we showed how to use dynamic programming to solve the problem efficiently. If u^* denotes the minimum cycle mean and $d(\cdot)$ represents shortest path distances with $c_{ij} - \theta$ as arc lengths, then $\pi'(i) = -d(i)$ solves (19.9). We can use these $\pi'(i)$ values to obtain row multipliers and column divisors.

19.4 SCIENCE APPLICATIONS

In previous chapters we have considered several applications in the physical and medical sciences, for example, reconstructing the left ventricle from x-ray projections and determining chemical bonds. To illustrate other possibilities in the science arena, we next consider two applications in the field of biology: DNA sequencing and automatic karyotyping of chromosomes. We solve the first of these problems as a shortest path problem and the second one as a transportation problem.

Application 19.7 DNA Sequence Alignment

Scientists model strands of DNA as a sequence of letters drawn from the alphabet $\{A, C, G, T\}$. Given two sequences of letters, say $B = b_1 b_2 \cdots b_p$ and $D = d_1 d_2 \cdots d_q$ of possibly different lengths, molecular biologists are interested in determining how similar or dissimilar these sequences are to each other. (These sequences are subsequences of a genome and typically contain several thousand letters.) A natural way of measuring the dissimilarity between the two sequences B and D is to determine the minimum "cost" required to transform sequence B into sequence D. To transform B into D, we can perform the following operations: (1) insert an element in B (at any place in the sequence) at a "cost" of α units; (2) delete an element from

B (at any place in the sequence) at a "cost" of β units; and (3) mutate an element b_i into an element d_j at a "cost" of $g(b_i, d_j)$ units. Needless to say, it is possible to transform the sequence B into the sequence D in many ways, so identifying a minimum cost transformation is a nontrivial task. We show how we can solve this problem using dynamic programming, which we can also view as solving a shortest path problem on an appropriately defined network.

Suppose that we conceive of the process of transforming the sequence B into the sequence D as follows. Add or delete elements from the sequence B so that the modified sequence, say B', has the same number of elements as D. Next "align" the sequences B' and D to create a one-to-one alignment between their elements. Finally, mutate the elements in the sequence B' so that this sequence becomes identical with the sequence D. As an example, suppose that we wish to transform the sequence B = AGTT into the sequence D = CTAGC. One possible transformation is to delete one T from B and add two new elements at the beginning, giving the sequence B' = $\oplus\oplus$AGT (we denote the new element by the placeholder $\oplus$ and later assign a letter to this placeholder). We then align B' with D, as shown in Figure 19.8, and mutate the element T into C so that the sequences become identical. Notice that because we are free to assign values to the newly added elements, they do not incur any mutation cost. The cost of this transformation is $\beta + 2\alpha + g(\mathrm{T}, \mathrm{C})$.

B' = $\oplus\oplus$AGT $\Rightarrow$ CTAGC
D = CTAGC CTAGC

Figure 19.8 Transforming the sequence B into the sequence D.

We now describe a dynamic programming formulation of this problem. Let $f(i, j)$ denote the minimum cost of transforming the subsequence $b_1 b_2 \cdots b_i$ into the subsequence $d_1 d_2 \cdots d_j$. We are interested in the value $f(p, q)$, which is the minimum cost of transforming B into D. To determine $f(p, q)$, we determine $f(i, j)$ for all $i = 0, 1, \ldots, p$, and for all $j = 0, 1, \ldots, q$. We can determine these intermediate quantities $f(i, j)$ using the following recursive relationships:

$$f(i, 0) = \beta_i \quad \text{for all } i, \tag{19.10a}$$

$$f(0, j) = \alpha_j \quad \text{for all } j, \tag{19.10b}$$

$$f(i, j) = \min\{f(i-1, j-1) + g(b_i, d_j), f(i, j-1) + \alpha, f(i-1, j) + \beta\}. \tag{19.10c}$$

We now justify this recursion. The cost $f(i, 0)$ of transforming a sequence of i elements into a null sequence is the cost of deleting i elements. The cost $f(0, j)$ of transforming a null sequence into a sequence of j elements is the cost of adding j elements. Next consider $f(i, j)$. Let B' denote the optimal aligned sequence of B (i.e., the sequence we create just before the mutation of B' to transform it into D). At this point, B' satisfies exactly one of the following three cases:

Case 1. *B' contains the letter b_i, which is aligned with the letter d_j of D* [as shown in Figure 19.9(a)].
In this case, $f(i, j)$ equals the optimal cost of transforming the subsequence $b_1 b_2 \cdots$

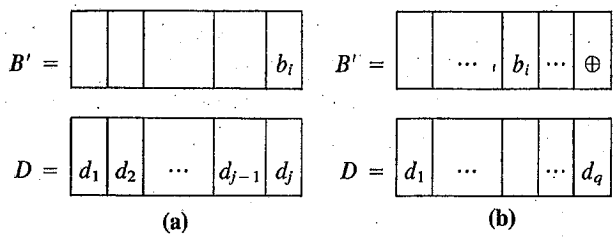


Figure 19.9 Explaining the dynamic programming recursion.

b_{i-1} into $d_1 d_2 \cdots d_{j-1}$ and the cost of transforming the element b_i into d_j. Therefore, $f(i, j) = f(i - 1, j - 1) + g(b_i, d_j)$.

Case 2. *B' contains the letter b_i, which is not aligned with the d_j* [as shown in Figure 19.9(b)].
In this case, b_i is to the left of d_j, so a newly added element must be aligned with b_j. In this case $f(i, j)$ equals the optimal cost of transforming the subsequence $b_1 b_2 \cdots b_i$ into $d_1 d_2 \cdots d_{j-1}$ plus the cost of adding a new element to B. Therefore, $f(i, j) = f(i, j - 1) + \alpha$.

Case 3. *B' does not contain the letter b_i.*
In this case we must have deleted b_i from B, so the optimal cost of the transformation equals the cost of deleting this element and transforming the remaining sequence into D. Therefore, $f(i, j) = f(i - 1, j) + \beta$.

The preceding discussion justifies the recursive relationships specified in (19.10). We can use these relationships to compute $f(i, j)$ for increasing values of i and, for a fixed value of i, for increasing values of j. This method allows us to compute $f(p, q)$ in $O(pq)$ time.

We can alternatively formulate the DNA sequence alignment problem as a shortest path problem. In Figure 19.10 we show the shortest path network for this

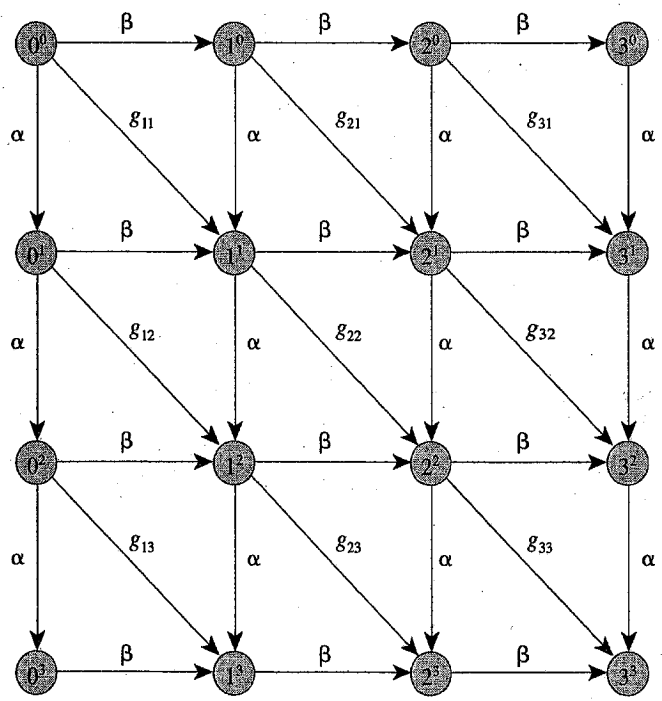


Figure 19.10 Sequence alignment problem as a shortest path problem.

formulation for a situation with $p = 3$ and $q = 3$. For simplicity, in this network we denote $g(b_i, d_j)$ by g_{ij}. We can establish the correctness of this formulation by applying an induction argument based on the induction hypothesis that the shortest path length from node 0^0 to node i^j equals $f(i, j)$. The shortest path from node 0^0 to node i^j must contain one of the following arcs as the last arc in the path: (1) arc $(i - 1^{j-1}, i^j)$, (2) arc (i^{j-1}, i^j), or (3) arc $(i - 1^j, i^j)$. In these three cases, the lengths of these paths will be $f(i - 1, j - 1) + g(b_i, d_j)$, $f(i, j - 1) + \alpha$, and $f(i - 1, j) + \beta$. Clearly, the shortest path length $f(i, j)$ will equal the minimum of these three numbers, which is consistent with the dynamic programming relationships stated in (19.10).

Application 19.8 Automatic Karyotyping of Chromosomes

A normal human cell has 46 chromosomes, usually subdivided into 23 groups with two identical (homologous) chromosomes per group, except for one male chromosome. For males, the group with the sex chromosomes has one X chromosome and one Y chromosome; these two chromosomes are not homologous. Each of these 23 groups of chromosomes has its own characteristic features and serves different biological functions. In certain clinical tests, such as amniocentesis, it is necessary to identify each of the 46 chromosomes of the cell. The medical community refers to the process of identifying which chromosomes belong to which of the 23 chromosome classes as *karyotyping*.

We consider the process of karyotyping a female cell. In this case each of the 23 chromosome classes consists of two homologous chromosomes. (Karyotyping of male cells is identical except for the treatment of the sex chromosomes.) The chromosomes of a suitably stained cell, when viewed under a microscope, exhibit a series of characteristic bands along the length of the cell. For each chromosome i of a cell and for each chromosome class j, we can assign a measure p_{ij} which is the probability that chromosome i is a member of class j. In some settings, a clinician measures the characteristic bands used to determine the p_{ij}'s. In other settings, a commercially available imaging machine, such as the Cytoscan system, uses a mechanical scanning device (known as a linear CCD array) to measure the bands.

The assignment of chromosomes to classes is easy if p_{ij} is nearly 1 for all correct assignments; however, in practice, the values of p_{ij}'s might be far from 1 for some correct assignments. An important task in karyotyping is to assign the chromosomes to classes in order to maximize the expected number of correct assignments, or, equivalently, to minimize the number of incorrect assignments.

We let $x_{ij} = 1$ if we assign chromosome i to class j, and $x_{ij} = 0$ otherwise. Using this notation we can formulate the problem of minimizing the expected number of incorrect assignments of chromosomes to chromosome classes as the following transportation problem:

$$\text{Minimize} \quad \sum_{i=1}^{46} \sum_{j=1}^{23} (1 - p_{ij}) x_{ij}$$

subject to

$$\sum_{j=1}^{23} x_{ij} = 1 \qquad \text{for } i = 1 \text{ to } 46,$$

$$\sum_{i=1}^{46} x_{ij} = 2 \qquad \text{for } j = 1 \text{ to } 23,$$

$x_{ij} \geq 0$ and integer.

19.5 PROJECT MANAGEMENT

An important class of network problems centers around the planning and scheduling of large projects, such as constructing a building or a highway, planning and launching a new product, installing and debugging a computer system, or developing and implementing a space exploration program. This application context was among the earliest successes of network optimization, and the network flow models of project management continue to be an important management tool used in numerous industries every day. In this section we consider three basic models of project management: a shortest path technique for scheduling projects to achieve the earliest possible completion and two network flow models for (1) just-in-time scheduling of jobs in a project and (2) deciding where to allocate additional resources to reduce a project's overall duration.

Application 19.9 Determining Minimum Project Duration

For the purpose of modeling, we envision a project as a set of jobs and a set A of precedence relations between the jobs. If $(i, j) \in A$, we need to complete job i before beginning job j. In addition, each job j has a known duration τ_j. The problem is to identify the project schedule (i.e., the start time of each job) that will satisfy the precedence relations between the jobs and complete the project in the least possible amount of time (alternatively, that gives the least possible *project duration*). Consider, for example, the project planning problem given in Figure 19.11.

We can formulate this project planning problem as a shortest path problem; in

Job	Duration	Immediate predecessors
a	14	—
b	3	—
c	3	*a, b*
d	7	*a*
e	4	*d*
f	10	*c, e*

Figure 19.11 Project planning problem.

fact, there are two alternative methods for doing so. In the first method we represent jobs by arcs, and in the second method we represent jobs by nodes. Although the first of these approaches is more popular in the literature, we adopt the latter approach in this discussion because it is conceptually simpler.

To formulate the project planning problem as a shortest path problem, we define a project network by associating a node j with each job j, and by including arc (i, j) whenever job i is an immediate predecessor of job j. We set the length c_{ij} of arc (i, j) equal to τ_i, the duration of job i. We also introduce a source node s, denoting the beginning of the project, and connect it to every node that has no incoming arc (corresponding to jobs without any predecessors) by zero-length arcs. Similarly, we introduce a sink node t, denoting the end of the project, and connect every node i with no outgoing arc to this sink node by an arc (i, t) whose length equals the duration of job i. Figure 19.12 gives the network corresponding to the project planning example shown in Figure 19.12. Note that the network corresponding to any project planning model must be acyclic because we could never complete a network containing a cycle (why?).

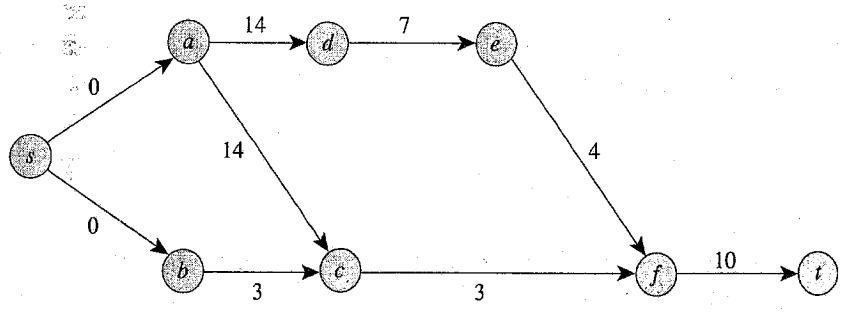


Figure 19.12 Shortest path formulation of the project planning problem.

Let $u(j)$ denote the earliest possible start time of job j in a project planning schedule that satisfies the precedence constraints. Notice that with respect to quantities $u(j)$, the project duration is $u(t) - u(s)$. We can state the project planning problem as the following optimization model:

$$\text{Minimize} \quad u(t) - u(s), \tag{19.11a}$$

subject to

$$u(j) - u(i) \geq c_{ij}, \text{ for all } (i, j) \in A, \tag{19.11b}$$

$$u(j) \text{ unrestricted.} \tag{19.11c}$$

The inequalities (19.11b) model the precedence constraints by stating that if job i is an immediate predecessor of job j, then job j can start only after $c_{ij} = u_i$ units of time have elapsed since the start of job i.

To bring (19.11) into a familiar network flow form, we take its dual. If we associate the dual variables x_{ij} with the constraints (19.11b), the dual linear program is

$$\text{Maximize} \quad \sum_{\{j:(i,j)\in A\}} c_{ij}x_{ij}$$

subject to

$$\sum_{\{j:(j,i)\in A\}} x_{ji} - \sum_{\{j:(i,j)\in A\}} x_{ij} = \begin{cases} -1 & \text{for } i = s, \\ 0 & \text{for all } i \in N - \{s, t\}, \\ 1 & \text{for } i = t, \end{cases}$$

$$x_{ij} \geq 0 \qquad \text{for all } (i, j) \in A.$$

Clearly, this is a longest path problem with c_{ij} as the length of arc (i, j); we wish to send 1 unit of flow from node s to node t along the longest path. To solve the problem we multiply each arc length by -1 and then solve a shortest path problem. Note that since the project planning network is acyclic, by multiplying the arc lengths by -1, we do not create any negative cycle. Let $d(\cdot)$ denote the vector of shortest path distances for this problem. Then by setting $u(i) = -d(i)$ for each $i \in N$, we obtain an optimal solution of (19.11).

Application 19.10 Just-in-Time Scheduling

The *just-in-time scheduling problem* is an extension of the project planning problem that we discussed in the last application. In the just-in-time scheduling problem, we determine the minimum project duration subject to both the precedence constraints and some additional "just-in-time constraints." In this problem we are given a subset $S \subseteq A$ and a number α_{ij} for each $(i, j) \in S$. The just-in-time constraints state that for each $(i, j) \in S$, job j must start within α_{ij} units of time from the start of job i. Notice that if $\alpha_{ij} = c_{ij}$, the just-in-time scheduling constraint for arc (i, j) says that job i must start exactly c_{ij} units before the start of job j, which is the latest possible start time for this job. Just-in-time is a management philosophy that has become very popular in recent years; it attempts to eliminate waste by reducing slack times and buffers, such as inventory between distribution, production, and scheduling activities.

If $u(\cdot)$ denotes the earliest start times of the jobs, the just-in-time constraints require that

$$u(j) \leq u(i) + \alpha_{ij} \qquad \text{for all } (i, j) \in S,$$

or, equivalently,

$$u(i) - u(j) \geq -\alpha_{ij} \qquad \text{for all } (i, j) \in S. \tag{19.12}$$

The start times must also satisfy the usual precedence constraints:

$$u(j) - u(i) \geq c_{ij} \qquad \text{for all } (i, j) \in A. \tag{19.13}$$

In the just-in-time scheduling problem, we wish to minimize $[u(t) - u(s)]$ subject to the inequalities (19.12) and (19.13). Just as we noted in our discussion in the last application, we can solve this problem as a longest path problem, in this case on an augmented network $G' = (N, A')$ whose arc set A' includes an arc (i, j) of cost c_{ij} for each $(i, j) \in A$ and an arc (j, i) of cost $-\alpha_{ij}$ for each $(i, j) \in S$. To transform this longest path problem into a shortest path problem, we multiply each arc cost by -1. Notice that in this case the augmented network G' might not

be acyclic, and the resulting shortest path problem might contain a negative cycle. The presence of a negative cycle indicates that the just-in-time scheduling problem has no feasible solution (why?). When the resulting shortest path problem has no negative cycle, the negative of the shortest path distances provide optimal start times for the jobs.

As a variant of the just-in-time scheduling problem, suppose that instead of imposing an upper bound on when job j should start after the start of job i, we penalize the time difference between the completion of job i and the start of job j using a penalty factor of d_{ij}. We wish to determine start times of jobs that will minimize this penalty and yet satisfy the restriction that the project duration is at most λ (a specified constant). The following linear program models this problem:

$$\text{Minimize} \sum_{(i,j)\in A} (u(j) - u(i) - c_{ij})d_{ij}, \tag{19.14a}$$

subject to

$$-u(t) + u(s) \geq -\lambda, \tag{19.14b}$$

$$u(j) - u(i) \geq c_{ij} \text{ for all } (i, j) \in A, \tag{19.14c}$$

$$u(j) \text{ unrestricted for all } j \in N. \tag{19.14d}$$

Let $D_i = \sum_{\{j:(j,i)\in A\}} d_{ji} - \sum_{\{j:(i,j)\in A\}} d_{ij}$. The following linear program is the dual of this model.

$$\text{Maximize} \sum_{\{j:(i,j)\in A\}} c_{ij}x_{ij} - \lambda x_{ts}$$

subject to

$$\sum_{\{j:(j,s)\in A\}} x_{js} - \sum_{\{j:(s,j)\in A\}} x_{sj} + x_{ts} = D_s,$$

$$\sum_{\{j:(j,i)\in A\}} x_{ji} - \sum_{\{j:(i,j)\in A\}} x_{ij} = D_i \qquad \text{for all } i \neq s \text{ or } t,$$

$$\sum_{\{j:(j,t)\in A\}} x_{jt} - \sum_{\{j:(t,j)\in A\}} x_{tj} - x_{ts} = D_t,$$

$$x_{ij} \geq 0 \qquad \text{for all } (i, j) \in A.$$

Note that this problem is a minimum cost network flow problem with an arc (t, s) from the end node t to the start node s.

Application 19.11 Time–Cost Trade-off in Project Management

Project scheduling problems provide celebrated applications of minimum cost flow duality theory. In the basic project scheduling problem that we examined in the preceding applications, we showed how we could represent an interrelated set of jobs as a network. As we noted in that discussion, whenever the time to perform each job is fixed, we can determine the minimum project duration by solving a shortest path problem. Many project managers would argue that they could reduce

the duration of most jobs by allocating extra resources (people, machines, or money) to them. Although the use of the added resources typically increases the cost of carrying out any particular job, expediting or *crashing* the job might permit the project team to complete the entire project more quickly. There might, however, be no reason to shorten the length of some jobs if they have a generous amount of *slack* (i.e., the time between the earliest and latest possible start times for the job that would not delay the project); the project team should perform this job at its *normal* pace, without any added resources. Thus we need not crash all jobs to complete a project faster; only certain "critical" jobs that have no slack need to be crashed. As we will see, we can formulate the problem of identifying which jobs to expedite, and by what amount, to complete the project within a specified time λ, as a minimum cost flow problem.

With job (i) in the network we associate a *time–cost trade-off curve*, which represents the cost of performing the job as a function of its duration. We assuxe that the time–cost trade-off curve is linear, as shown in Figure 19.13. We use three numbers to specify the trade-off curve for any job (i): Its "normal" completion time a_i, its "crash" completion time b_i, and the cost d_i of shortening the job duration by 1 unit. To assess the overall project cost, we would also need to know the normal cost of performing a job. But since we must perform every job, the normal cost contributes a constant to the objective function value; therefore, we ignore it from the analysis.

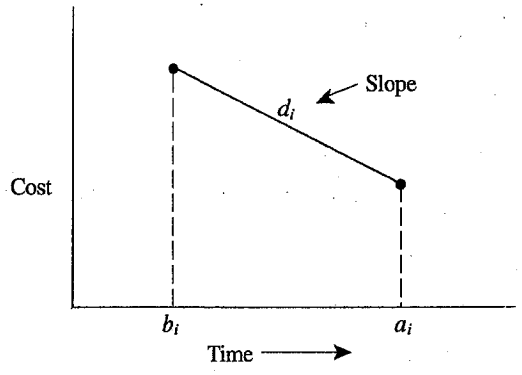


Figure 19.13 Time–cost trade-off curve of job i.

We would like to determine the optimal reduction, if any, in each job time that would permit us to reduce the project duration to a given value λ while incurring the least possible cost in reducing one or more of the job durations. To formulate this problem as a network flow model, we introduce some notation. As in the last application, we let node s designate the beginning of the project and node t designate its completion. The formulation uses two decision variables for each job: (1) $u(i)$, which denotes the start time of job i (i.e., the start time of all the jobs emanating from node i) and (2) β_i, which represents the reduction in time required to complete job i. After we reduce the duration of the jobs, the time required to complete job i will be $a_i - \beta_i$. Using this notation, we state the project scheduling problem mathematically as follows:

$$\text{Minimize} \quad \sum_{i \in N} d_i \beta_i \tag{19.15a}$$

subject to

$$u(t) - u(s) \leq \lambda, \tag{19.15b}$$

$$u(j) \geq u(i) + (a_i - \beta_i) \quad \text{for all } (i, j) \in A, \tag{19.15c}$$

$$0 \leq \beta_i \leq a_i - b_i \quad \text{for all } i \in N, \tag{19.15d}$$

$$u(i) \text{ unrestricted for all } i \in N. \tag{19.15e}$$

In this formulation, the constraint (19.15a) assures that the project duration is at most λ, constraint (19.15b) ensures that the jobs satisfy their precedence constraints, and (19.15c) imposes lower and upper bounds on the reductions in job times. We next make a substitution of variables to simplify (19.15). Let $\beta_i' = u(i) - \beta_i$ for all $i \in N$. Substituting β_i' in (19.15) in the place of β_i, we obtain the following modified formulation:

$$\text{Minimize} \quad \sum_{i \in N} d_i u(i) - \sum_{i \in N} d_i \beta_i' \tag{19.16a}$$

subject to

$$u(t) - u(s) \leq \lambda, \tag{19.16b}$$

$$u(j) - \beta_i' \geq a_i \quad \text{for all } (i, j) \in A, \tag{19.16c}$$

$$u(i) - \beta_i' \geq 0 \quad \text{for all } i \in N, \tag{19.16d}$$

$$u(i) - \beta_i' \leq a_i - b_i \quad \text{for all } i \in N, \tag{19.16e}$$

$$u(i) \text{ and } \beta_i' \text{ are unrestricted for all } i \in N. \tag{19.16f}$$

In this formulation, each row has at most one $+1$ and at most one -1; consequently, our discussion in Section 9.4 shows that this model is the dual of a minimum cost flow problem. Consequently, we can obtain an optimal solution of (19.16) using any minimum cost flow algorithm.

19.6 DYNAMIC FLOWS

Much of our discussion in this book has focused on static models, that is, problems that have no underlying temporal dimensions. As we have seen, static network models provide good mathematical representations of a great many applications. In some other applications, however, such as scheduling of people, jobs, or projects, or the carryover of inventory of a product from one time period to another, time is an essential ingredient. In these instances, to account properly for the evolution of the underlying system over time, we need to use dynamic network flow models. We might view these models as being composed of multiple copies of an underlying network, one at each point in time: arcs that link these static "snapshots" of the underlying network describe temporal linkages in the system. Since each node in the network now has an associated time, we refer to the dynamic networks as *time-expanded networks*. Dynamic network models arise in many problem settings, including production-distribution systems, economic planning, energy systems, traffic systems, and building evacuation systems. In the following discussion we describe two applications of dynamic network flow models.

Application 19.12 Maximum Dynamic Flows

The maximum dynamic flow problem is a variant of the maximum flow problem that arises, for example, in the following scenario. In a war between two countries, A and B, suppose that the generals of army A have decided to launch a major offensive in the next 24 hours using their major infantry units based at location s against enemy troops at location t. The generals would like to send a maximum number of units from location s to location t within 24 hours while honoring the arc capacities and traversal times of the arcs.

In the maximum flow problem, we maximize the number of flow units that can pass through the network from node s to node t *per unit time* while satisfying the arc capacities u_{ij}. In the dynamic flow problem, we maximize the total number of flow units that can be sent from node s to node t in p time periods while satisfying arc capacities u_{ij} and the *arc traversal times* τ_{ij}. In other words, the maximum flow problem determines the maximum *steady state* flow per unit time between two nodes, so we might refer to this problem as the *static flow problem*. On the other hand, the maximum dynamic flow problem maximizes the total (transient) flow that we can send between two nodes within a given period. We will show how to transform the maximum dynamic flow problem into a maximum flow problem on a new network $G^p = (N^p, A^p)$, called the *time-expanded replica* of G. We illustrate our transformation on the example shown in Figure 19.14(a) with the time-expanded replica for $p = 6$ shown in Figure 19.14(b).

For a given network $G = (N, A)$, we form the network G^p as follows. We make p copies $i_1, i_2, \ldots, i_p$ of each node i. Node i_k in the time-expanded network represents node i of the original network at time k. We include arc (i_k, j_l) of capacity u_{ij} in the time-expanded network whenever $(i, j) \in A$ and $l - k = \tau_{ij}$; the arc (i_k, j_l) in the time-expanded network represents the potential movement of a commodity from node i to node j in time τ_{ij}. It is easy to see that any static flow in G^p from the source nodes $s_1, s_2, \ldots, s_p$ to the sink nodes $t_1, t_2, \ldots, t_p$ is equivalent to a dynamic flow in G, and vice versa. We can further reduce the multiple-source, multiple-sink problem in the time-expanded network to the single-source, single-sink problem by introducing a *super-source node* s^* and a *super-sink node* t^*. Consequently, we can solve the maximum dynamic flow problem in G by solving an (ordinary) maximum flow problem in G^p.

Application 19.13 Models for Building Evacuation

In large metropolitan areas, among the criteria used to design large buildings, architects must ensure sufficient capabilities to evacuate buildings quickly: to respond, for example, to a fire, an earthquake, a toxic or natural gas leak, a power blackout, a bomb threat, or a civil defense emergency. As an aid to their design efforts, the architects would like to be able to develop an evacuation plan and assess the evacuation time for any particular design. We show how to model this building evacuation problem as a dynamic flow problem and solve it using a minimum cost flow algorithm. In our description we present a highly simplified version of the building evacuation problem. The references cited at the end of this chapter provide a more realistic description of the problem.

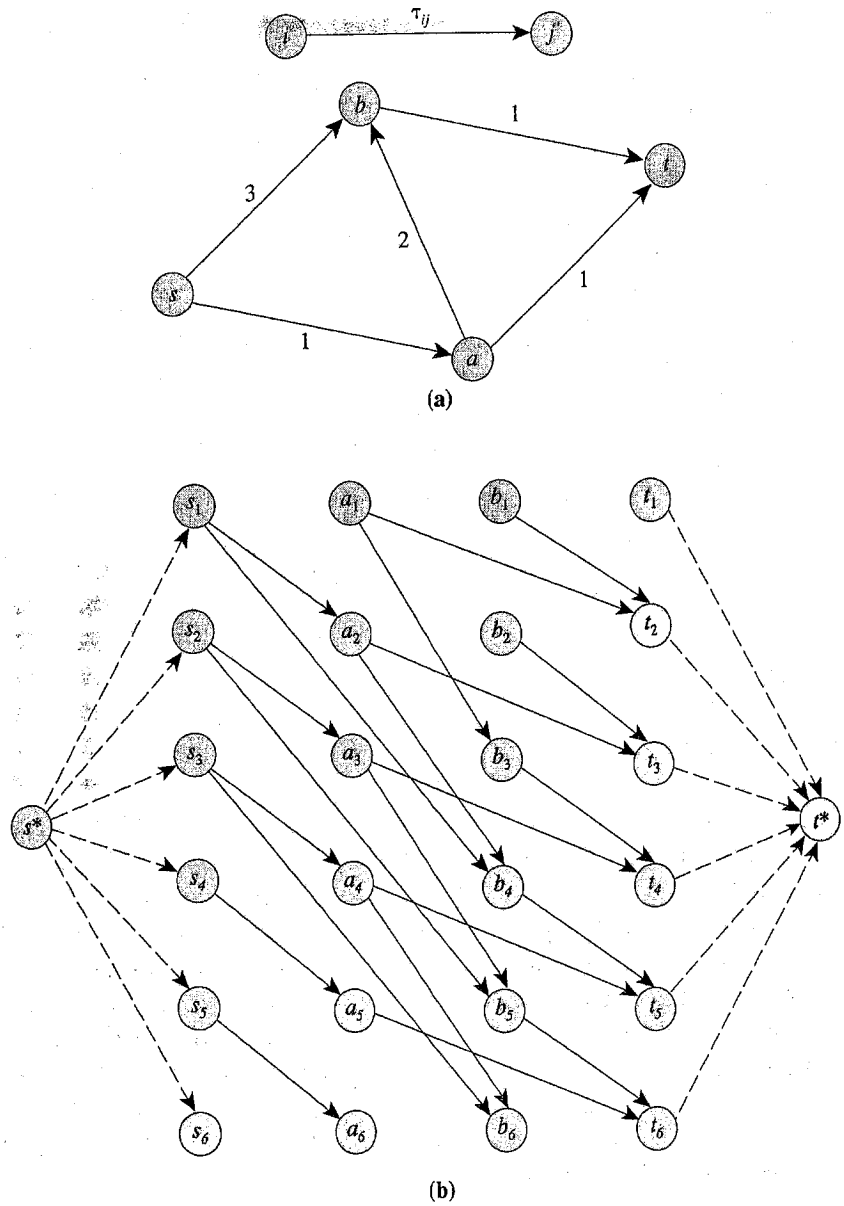


Figure 19.14 (a) Network G; (b) time-expanded replica of G. All capacities equal 2 in both the networks.

We first construct a static network for the building. The nodes of this network represent locations of the building such as work centers, offices, hallways, elevator stops, staircases, and building exits; the arcs represent passages between locations. Those locations of the building that house a significant number of people are source nodes in the network, and the building exits are the sink nodes. The supply of a source node equals an estimate of the number of people in the location that the node represents. The capacity of an arc is the number of people that can pass through the associated passage way per unit time. For example, if we anticipate that at most 60 people per minute can pass by every point in a stairwell, and the length of the time period in our model is 10 seconds, the capacity of the stairwell is $(60)(\frac{1}{6}) = 10$

units. We estimate the travel time of an arc and represent it as an integral number of time periods. For example, specifying two time periods for descending one floor in a stairwell means that we allow 20 seconds.

Although the static model might have multiple sources and multiple sinks (exits), we can transform it into a model with single source and single sink by using the standard technique of introducing a *super-source* node and a *super-sink* node. Therefore, we assume that the network contains a single source s with a given supply, say B, and a single sink t.

We construct the time-expanded replica of the static model as described in the last application. The time-expanded replica of the network contains p nodes i^1, i^2, i^3, . . . , i^p for each node i in the static network; we choose p to be suitably large to ensure that we can evacuate the building within p time periods. The minimum time required to evacuate the building is the smallest index r satisfying the property that the maximum flow from the nodes s^1, s^2, . . . , s^r to the nodes t^1, t^2, . . . , t^r is at least B people (B is given data). As shown in Exercise 19.11, we can solve this problem by a single application of any minimum cost flow algorithm or repeated application of any maximum flow algorithm.

19.7 ARC ROUTING PROBLEMS

Leaving from his or her home post office, a postal carrier needs to visit the households on each block in the carrier's route, delivering and collecting letters, and then returning to the post office. The carrier would like to cover this route by traveling the minimum possible distance. Mathematically, this problem has the following form: Given a network $G = (N, A)$ whose arcs (i, j) have an associated nonnegative length c_{ij}, we wish to identify a walk of minimum length that starts at some node (the post office), visits each arc of the network at least once, and returns to the starting node. This problem has become known as the *Chinese postman problem* because it was first discussed by a Chinese mathematician, K. Mei-Ko. The Chinese postman problem arises in other application settings as well; for instance, in patrolling streets by police, routing of street sweepers and household refuse collection vehicles, fuel oil delivery to households, and the spraying of roads with sand during snowstorms.

We discuss the Chinese postman problem on directed and undirected networks separately. We will show how to reduce the Chinese postman problem on a directed network to a minimum cost flow problem and how to solve the Chinese postman problem on an undirected network as a nonbipartite weighted matching problem. Interestingly, the Chinese postman problem on a mixed network (i.e., some arcs are directed and the others are undirected) is $\mathcal{NP}$-complete.

Application 19.14 Directed Chinese Postman Problem

In the Chinese postman problem on directed networks, we are interested in a closed (directed) walk that traverses each arc of the network at least once. The network need not contain any such walk! Figure 19.15 shows an example. The network contains the desired walk if and only if every node in the network is reachable from every other node; that is, it is *strongly connected*. In Section 3.4 we showed how

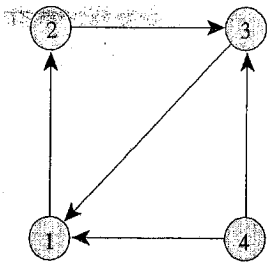


Figure 19.15 Network with no feasible solution for the Chinese postman problem.

to determine whether or not the graph is strongly connected within $O(m)$ time. In the remainder of our discussion we assume that the network G is strongly connected.

In an optimal walk, a postal carrier might traverse arcs more than once. The minimum length walk minimizes the sum of lengths of the repeated arc traversals. Let x_{ij} denote the number of times the postal carrier traverses arc (i, j) in a walk. Any carrier walk must satisfy the following conditions:

$$\sum_{\{j:(i,j)\in A\}} x_{ij} - \sum_{\{j:(j,i)\in A\}} x_{ji} = 0 \qquad \text{for all } i \in N, \tag{19.17a}$$

$$x_{ij} \geq 1 \qquad \text{for all } (i, j) \in A. \tag{19.17b}$$

The constraints (19.17a) state that the carrier enters a node the same number of times that he or she leaves it. The constraints (19.17b) state that the carrier must visit each arc at least once. Any solution x satisfying (19.17a) and (19.17b) defines a carrier's walk. We can construct such a walk in the following manner. We replace each arc (i, j) with flow x_{ij} with x_{ij} copies of the arc, each carrying a unit flow. Let A' denote the resulting arc set. Since the outflow equals inflow for each node in the flow x, once we have transformed the network, the outdegree of each node will equal its indegree. The flow decomposition theorem (i.e., Theorem 3.5) implies that we can decompose the arc set A into a set of at most m directed cycles. We can connect these cycles together to form a closed walk as follows. The carrier starts at some node in one of the cycles, say W_1, and visits the nodes (and arcs) of W_1 in order until returning to his or her starting node, or encountering a node that also lies in a directed cycle not yet visited, say W_2. In the former case, the walk is complete; and in the latter case, the carrier visits cycle W_2 first before resuming his or her visit of the nodes in W_1. While visiting nodes in W_2, the carrier follows the same policy, i.e., if he/she encounters a node lying on another directed cycle W_3 not yet visited, he/she visits W_3 first before visiting the remaining nodes in W_2, and so on. We illustrate this method on a numerical example. Let A' be as indicated in Figure 19.16(a). This solution decomposes into three directed cycles, W_1, W_2, and W_3. As shown in Figure 19.16(b), the carrier starts at node a and visits the nodes in the following order: a–b–d–g–h–c–d–e–b–c–f–a.

This discussion shows that the solution x defined by a feasible walk for the carrier satisfies (19.17), and, conversely, every feasible solution of (19.17) defines a walk of the postman. The length of a walk equals $\sum_{(i,j)\in A} c_{ij}x_{ij}$. Therefore, the Chinese postman problem seeks a solution x that minimizes $\sum_{(i,j)\in A} c_{ij}x_{ij}$, subject to the set of constraints (19.17). This problem is clearly an instance of the minimum cost flow problem.

One particular instance of the directed Chinese postman problem deserves

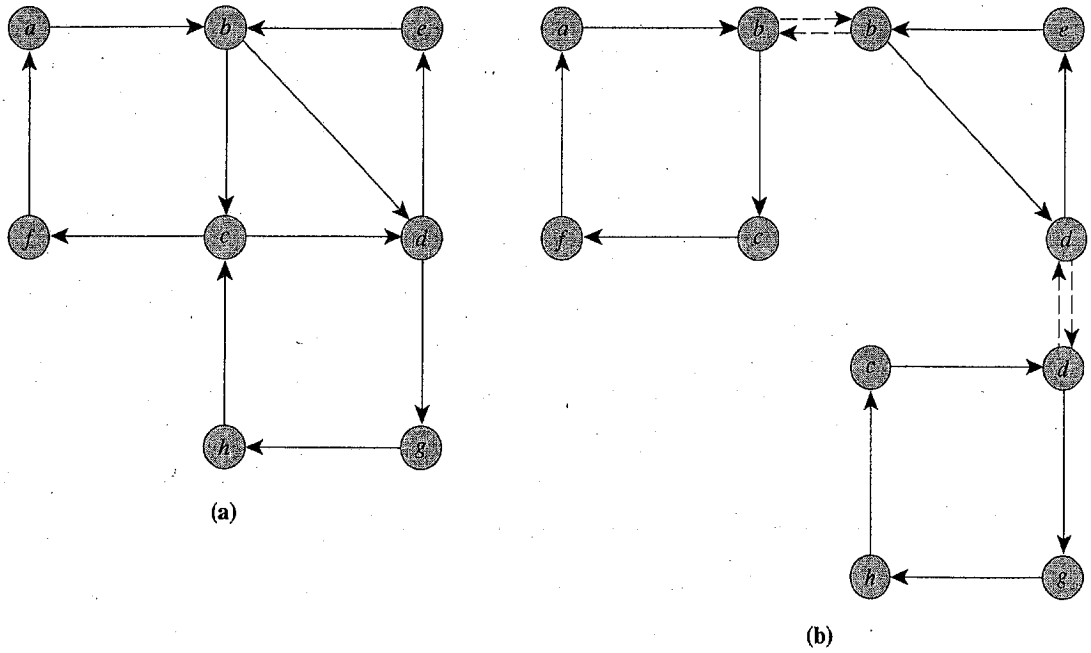


Figure 19.16 Constructing a closed walk for the postal carrier.

special mention. Suppose that the indegree of each node in G equals its outdegree. In that case, by setting $x_{ij} = 1$ for each arc $(i, j) \in A$, we define a feasible solution of (19.17) that is also a minimum cost solution (why?). Consequently, whenever the indegree of each node in a network equals its outdegree, the network contains a walk that traverses each arc exactly once (i.e., the walk has no arc repetitions). If the network contains a node whose indegree differs from its outdegree, the walk must necessarily repeat some arcs (why?). This result is the directed version of the well-known Euler's theorem for undirected graphs that we discussed in Exercise 3.45.

Application 19.15 Undirected Chinese Postman Problem

The undirected version of the Chinese postman problem is more complicated than the directed version; as we will see, rather than solving this problem as a minimum cost flow problem, we will solve it by using both an all-pairs shortest path algorithm and a nonbipartite weighted matching algorithm.

As we observed in the directed case, a network contains a directed walk visiting each arc exactly once if and only if the indegree of each node equals its outdegree. For the undirected case, we have an analogous result, which is known as Euler's theorem. This theorem states that a graph contains a closed walk that visits each arc exactly once if and only if every node in the graph has an even degree (i.e., the graph is *even*). In Euler's honor, the research community often refers to even graphs as Eulerian graphs and to closed walks that visit each arc exactly once as Eulerian tours.

Euler's theorem is very easy to establish. Note that since any closed walk in an undirected network enters and leaves any node the same number of times, the subgraph composed of the arcs in any closed walk is even. Therefore, if the network contains a closed walk visiting each arc exactly once, the graph must be even. The converse statement—that is, if the graph is even, then it contains an Eulerian tour—is an easy consequence of a simple probing algorithm that we discussed in the previous application. Since all Eulerian tours have the same cost (the sum of the arc costs), solving the Chinese postman problem on an even graph does not involve any optimization. We simply need to find any Eulerian tour.

If the graph is not even, a closed walk on a graph must repeat some arcs. Consequently, identifying a minimum length closed walk requires an optimization technique; we must determine which arcs to repeat. Suppose that a feasible carrier tour is a closed walk W that traverses arc (i, j), x_{ij} times. Suppose that we replace each arc (i, j) by x_{ij} copies of the same arc and refer to the resulting graph as the *traversal graph*. Notice that since the degree of each node in the traversal graph is even, it contains a closed walk visiting each arc exactly once. We now delete one copy of each arc (i, j) in the traversal graph and refer to the remaining graph as the *repetition graph*, which we represent as $G'' = (N, A'')$. Observe that the arc set $A \cup A''$ defines the traversal graph.

We now focus on the optimal repetition graph $G'' = (N, A'')$. A node i in $G'' = (N, A'')$ has an odd degree if and only if i has an odd degree in $G = (N, A)$, because $A \cup A''$ defines even degrees for all nodes. We claim that the repetition graph can be decomposed into an arc-disjoint union of cycles and paths satisfying the following properties: (1) each path starts and ends at odd-degree nodes in G'', and (2) each odd-degree node in G'' is the start node or the end node of exactly one path. We can establish this claim by using an argument similar to the one we used in the proof of the flow decomposition theorem (i.e., Theorem 3.5).

Consider now the *optimal repetition graph* G^* (i.e., the repetition graph corresponding to an optimal solution of the Chinese postman problem). We can assume without any loss of generality that G^* does not contain any cycle because each such cycle must have zero length (because arc lengths are nonnegative, any cycle length is nonnegative; hence, G^* would not be optimal if it contained a positive length cycle). We can eliminate any zero length cycle without affecting the objective function value. Therefore, we can assume that the optimal repetition graph G^* consists of arc-disjoint paths between odd-degree nodes. Each odd-degree node is paired to another odd-degree node by a path, and collectively these paths contain all the arcs in G^*. Notice that for a given pairing of odd-degree nodes, the path between them must be a shortest path; otherwise, we could improve the solution.

The preceding discussion implies the following procedure for solving the Chinese postman problem. We first solve an all-pairs shortest path problem in $G = (N, A)$ to compute shortest path distances d_{ij} between all pairs of nodes. We then construct a graph $G' = (N', A')$ whose node set consists of all odd-degree nodes in N and whose arc set contains arcs between every pair of nodes in N'. We set the cost of the arc $(i, j) \in A'$ to the shortest path distance d_{ij}. The network G' is nonbipartite. Next we identify a minimum weight (nonbipartite) matching M of G' to identify an optimal pairing of the nodes. For each arc $(i, j) \in M$ in the optimal

matching, we add one copy of every arc in the shortest path from node i to node j to A. We can then decompose the resulting even graph into the carrier's closed walk by the procedure described in the previous application.

19.8 FACILITY LAYOUT AND LOCATION

Facility layout and location problems offer two perspectives on a similar set of issues. In facility layout problems we would like to determine the best internal configuration of an office building, plant, or warehouse. Where should we put offices, equipment, storage locations, and any support facilities, such as material handling equipment, heating and ventilation, a mailroom, dining rooms, and rest rooms? In answering this question, we would like to configure the facility for maximum benefit (e.g., most efficient interactions between the occupants) at the least possible cost, while recognizing building codes and other externally imposed regulations (e.g., evacuation codes). Facility location models typically are more macroscopic and consider the interactions between, rather than within, facilities: for example, between plants, warehouses, and retail outlets; or between libraries, fire houses, service facilities (e.g., drugstores, fast-food restaurants), and the populations that they serve. Many facility layout and facility location problems are integer programming models with embedded network flow structure since we either locate a facility at a particular location or not (a zero–one variable) and once we have decided on the location of the facilities, we wish to route people or goods between them at the least possible cost. Some facility location and layout problems, however, such as the two that we consider in this section, are pure network flow models.

Application 19.16 Discrete Location Problems

This application is concerned with the optimal location of p new facilities to be selected from an available set of $q \geq p$ sites. The new facilities interact with the r existing facilities. The objective is to locate the facilities to minimize the total transportation cost between the new and existing facilities. One example of this problem is the location of hospitals, fire stations, or libraries in a metropolitan area; in this setting, we can treat population concentrations as the existing facilities. Other examples of this problem are the locations of new airfields used to provide supplies for a number of military bases, new components on a control panel, and water fountains in an office building.

The data of this problem consist of (1) d_{kj}, the distance between existing facility k and site j; and (2) w_{ik}, the total transportation cost per unit distance (for some given time period) incurred between the new facility i and the existing facility k. If we use the same medium of transportation between all the facilities, we can let w_{ik} be the number of trips per unit time (day, week, month) made between the new facility i and the existing facility k. The objective is to assign each new facility i to an available site j, denoted by a binary variable x_{ij}, to minimize the total cost of transportation between new and existing facilities. For a given assignment x, $w_{ik} \sum_{j=1}^{q} d_{kj} x_{ij}$ is the cost of transportation between the new facility i and the existing facility k. Thus the total transportation cost is given by

$$\sum_{i=1}^{p}\sum_{k=1}^{r}\left(w_{ik}\sum_{j=1}^{q}d_{kj}x_{ij}\right),$$

or, equivalently,

$$\sum_{i=1}^{p}\sum_{j=1}^{q}\left(\sum_{k=1}^{r}w_{ik}d_{kj}\right)x_{ij}=\sum_{i=1}^{p}\sum_{j=1}^{q}c_{ij}x_{ij}.$$

In this expression $c_{ij} = \sum_{k=1}^{r} w_{ik}d_{kj}$ is the cost of locating the new facility i at site j. This model is an instance of the assignment problem.

Notice that in this model, the new facilities are independent of each other in the sense that the transportation cost (as measured by number of trips) between new facility i and old facility k does not depend upon the location of other new facilities. When the demand (and the transportation cost) between facilities depends on the location of other facilities (e.g., when the new facilities provide the same service and any old facility will travel to the least cost or closest facility), the models become more complicated, and typically are integer programming models.

Application 19.17 Warehouse Layout

A warehouse typically is configured with docks for loading and unloading goods and with open areas (or bins) for storing the goods. Trucks that deliver or pick up goods arrive at any one of the loading docks (typically, at random). The warehouse operators must collect or deliver the required items from their storage locations. In managing the warehouse, the operating staff must decide where to locate each of the goods—the closer a good is to any dock, the lower is the cost of (1) accessing that good and transferring it to the dock for loading, or (2) unloading the good from the dock and transporting it to its storage area. Consequently, the items "compete" for the storage areas that are closest to the docks. Suppose that a company is using a warehouse to store p items. The warehouse has r loading and unloading docks. Let w_{ik} be a known total cost per foot incurred in transporting item i between dock k and its storage region (this cost might be just the cost of the time of a forklift operator—the transport time might depend on the item being moved, because some items are heavier or bulkier than others). Typically, the warehouse will store items on pallets (small wooden platforms) and w_{ik} is directly proportional to the number of pallet loads of item i moving between dock k and the storage region of the item. The warehouse layout problem is to determine the region(s) for storing each of the p items that will minimize the total transportation cost between the items and the docks.

We first discretize this problem by subdividing the floor area into q square grids of equal size, numbered in any convenient manner from 1 to q. Since we represent the distance between each grid and each dock by a single (average) number, we will be approximating the travel distances and associated loading and unloading costs; the accuracy of the approximation depends on the size of the grids. Choosing a smaller grid size improves the accuracy of the approximation but increases the problem size; therefore, we must make a trade-off between accuracy and solution cost. Let F_i be the total number of grids required to store item i. For simplicity, we

assume that $\sum_{i=1}^{p} F_i = q$. Let d_{kj} denote the distance between dock k and the center of grid j. Multiplying this quantity by w_{ik}, we obtain the transportation cost $w_{ik}d_{kj}$ for item i between grid location j and dock k. Since we are assuming that each item is equally likely to be loaded and unloaded from each dock, the average cost for locating item i in grid j is

$$c_{ij} = \frac{1}{F_i} \sum_{k=1}^{r} w_{ik}d_{kj}.$$

We convert the warehousing problem to a standard transportation problem by defining the variables x_{ij}, for all $i = 1, \ldots, p$ and $j = 1, 2, \ldots, q$, as follows:

$$x_{ij} = \begin{cases} 1 & \text{if we store item } i \text{ in grid square } j, \\ 0 & \text{if we do not store item } i \text{ in grid square } j. \end{cases}$$

In terms of the variables x_{ij}, the warehouse layout problem becomes

$$\text{Minimize} \quad \sum_{i=1}^{p} \sum_{j=1}^{q} c_{ij}x_{ij}$$

subject to

$$\sum_{j=1}^{q} x_{ij} = F_i \qquad \text{for } i = 1, \ldots, p,$$

$$\sum_{i=1}^{p} x_{ij} = 1 \qquad \text{for } j = 1, \ldots, q,$$

$$x_{ij} \geq 0, \qquad \text{for all } i = 1, \ldots, p, j = 1, \ldots, q,$$

which is clearly an instance of the transportation problem.

Application 19.18 Rectilinear Distance Facility Location

In this application we model a rectilinear distance facility location as the dual of a minimum cost flow problem. Suppose that r existing facilities (e.g., hospitals that offer different medical expertise) are located at distinct points in a two-dimensional plane: The ith facility has the coordinates (u_i, v_i). We wish to locate p new facilities. Let (x_j, y_j), which are the decision variables, denote the coordinates of the jth new facility. For each set of locations of the new facilities, we incur transportation costs that are proportional to the rectilinear distances between the new facilities and between the new and existing facilities. The *rectilinear distance* between any two points in the xy-plane is the shortest distance between the points if we restrict travel to be parallel to the two axes (i.e., x-axis and y-axis). Therefore, the rectilinear distance between the two points (x_1, y_1) and (x_2, y_2) is $(|x_1 - x_2| + |y_1 - y_2|)$. Because rectilinear distances model distances on many urban street networks, they are sometimes also called *manhattan distances*.

In our location problem, we want to determine the coordinates (x_j, y_j) for all $j = 1, \ldots, r$ that optimize the following objective function:

$$\text{Minimize} \sum_{j=1}^{p} \sum_{k=1}^{p} a_{jk}(|x_j - x_k| + |y_j - y_k|) + \sum_{i=1}^{r} \sum_{j=1}^{p} d_{ij}(|x_j - u_i| + |y_j - v_i|). \tag{19.18}$$

In this expression, a_{jk} and d_{ij} are constants that represent the cost (per unit time) per unit distance traveled between the two facilities. It might be useful to think of d_{ij} as the number of trips made between the existing facility i and the new facility j. With this interpretation, the term a_{jk} (as well as a_{kj}) would represent half of the number of trips made between the new facilities j and k (because the summation counts the interaction between the new facilities j and k twice). In the first term, we should, in fact, be summing over all indices j and k with $j \neq k$; since $|x_j - x_j| = |y_j - y_j| = 0$, for each j we can set a_{jj} to any value, e.g., $a_{jj} = 0$ for all j.

First, we notice that the objective function (19.18) consists of two parts; the first part contains the x-coordinates of the new facilities and the second part contains the y-coordinates. Since these two parts are independent of each other, we can optimize them separately. In view of this observation, we assume that our location problem has the following form:

$$\text{Minimize} \sum_{j=1}^{p} \sum_{k=1}^{p} a_{jk} |x_j - x_k| + \sum_{i=1}^{r} \sum_{j=1}^{p} d_{ij} |x_j - u_i|. \tag{19.19}$$

We have a similar problem for the y variables. To solve the location problem, we first convert it into a linear program. To do so, we use a standard technique for converting a function with absolute value functions to a linear programming problem [i.e., we replace the term "minimize $|\alpha - \beta|$" by the term "minimize $(c + d)$, subject to the constraints $\alpha - \beta = c - d$, $c \geq 0$, and $d \geq 0$"] (we discuss this transformation in Exercise 19.1). Using this technique, we convert (19.19) into the following equivalent problem, which we state in the form of a maximization problem:

$$\text{Maximize} \sum_{j=1}^{p} \sum_{k=1}^{p} (-a_{jk})(e_{jk} + f_{jk}) + \sum_{i=1}^{r} \sum_{j=1}^{p} (-d_{ij})(g_{ij} + h_{ij}) \tag{19.20a}$$

subject to

$$x_j - x_k - e_{jk} + f_{jk} = 0 \quad \text{for all } j \text{ and all } k, \tag{19.20b}$$

$$x_j - g_{ij} + h_{ij} = u_i \quad \text{for all } i \text{ and all } j, \tag{19.20c}$$

$$e_{jk} \geq 0,\ f_{jk} \geq 0,\ g_{ij} \geq 0,\ h_{ij} \geq 0, \text{ and } x_j \text{ unrestricted.} \tag{19.20d}$$

We next make a transformation of variables to simplify (19.20). Let $e'_{jk} = e_{jk} + x_k$, $f'_{jk} = f_{jk} + x_j$, and $h'_{ij} = h_{ij} + x_j$. Substituting these variables in (19.20) gives

$$f'_{jk} - e'_{jk} = 0 \qquad \text{for all } j \text{ and all } k, \tag{19.21a}$$

$$h'_{ij} - g_{ij} = u_{ij} \qquad \text{for all } i \text{ and all } j, \tag{19.21b}$$

$$e'_{jk} - x_k \geq 0 \qquad \text{for all } j \text{ and all } k, \tag{19.21c}$$

$$f'_{jk} - x_j \geq 0 \qquad \text{for all } j \text{ and all } k, \tag{19.21d}$$

$$h'_{ij} - x_j \geq 0 \qquad \text{for all } i \text{ and all } j, \tag{19.21e}$$

$$g_{ij} \geq 0;\ e'_{jk},\ f'_{jk},\ h'_{ij}, \text{ and } x_j \text{ are unrestricted.} \tag{19.21f}$$

Now notice that (19.21a) implies that $f'_{jk} = e'_{jk}$ and (19.21b) implies that $g_{ij} = h'_{ij} - u_{ij}$. Making this substitution in (19.21c) to (19.21f) shows that

$$e'_{jk} - x_k \geq 0 \qquad \text{for all } j \text{ and all } k, \tag{19.22a}$$

$$f'_{jk} - x_j \geq 0 \qquad \text{for all } j \text{ and all } k, \tag{19.22b}$$

$$h'_{ij} - x_j \geq 0 \qquad \text{for all } i \text{ and all } j, \tag{19.22c}$$

$$h'_{ij} \geq u_{ij} \qquad \text{for all } i \text{ and all } j, \tag{19.22d}$$

$$e'_{jk},\ f'_{jk},\ h'_{ij}, \text{ and } x_j \text{ are unrestricted.} \tag{19.22e}$$

Observe that in (19.22), each constraint has at most one $+1$ and at most one -1. In Section 9.4 (see Theorem 9.9) we have shown that a linear program with at most $+1$ and at most one -1 in each constraint is the linear programming dual of a minimum cost flow problem. Consequently, we can solve the rectilinear distance location problem by taking the dual of (19.20a) and (19.22) and solving the resulting minimum cost flow problem. The optimal node potentials of this minimum cost flow problem will be an optimal solution of the (transformed) location problem.

19.9 PRODUCTION AND INVENTORY PLANNING

Many optimization problems in production and inventory planning are network flow problems. In fact, many of the earliest contributions of operations research to the field of management were network flow models for this class of applications. To illustrate the range of applications of network flow models in this problem domain, we consider several deterministic production-inventory planning problems involving multiple periods, single or multiple stages, and single or multiple products. All of these models address a basic *economic order quantity* issue: When we plan a production run of any particular product, how much should we produce? Producing in large quantities reduces the time and cost required to set up equipment for the individual production runs; on the other hand, producing in large quantities also means that we will carry many items in inventory awaiting purchase by customers. The economic order quantity strikes a balance between the setup and inventory costs to find the production plan that achieves the lowest overall costs. The models that we consider in this section all attempt to balance the production and inventory carrying costs while meeting known demands that vary throughout a given planning horizon. We first study the simplest model of this genre: a single-product, single-

stage model without backordering, which is known as the *dynamic lot-size problem.*

Application 19.19 Dynamic Lot Sizing

In the dynamic lot-size problem, we wish to meet prescribed demand d_j for each of K periods $j = 1, 2, \ldots, K$ by either producing an amount x_j in period j and/or by drawing upon the inventory I_{j-1} carried from the previous period. Figure 19.17 shows the network for modeling this problem. The network has $K + 1$ nodes: the jth node, for $j = 1, 2, \ldots, K$, represents the jth planning period; node 0 represents the "source" of all production. The flow on the *production arc* $(0, j)$ prescribes the production level x_j in period j, and the flow on *inventory carrying arc* $(j, j + 1)$ prescribes the inventory level I_j to be carried from period j to period $j + 1$. The mass balance equation for each period j models the basic accounting equation: Incoming inventory plus production in that period must equal the period's demand plus the final inventory at the end of the period. The mass balance equation for node 0 indicates that during the planning periods $1, 2, \ldots, K$, we must produce all of the demand (we are assuming zero beginning and zero final inventory over the planning horizon).

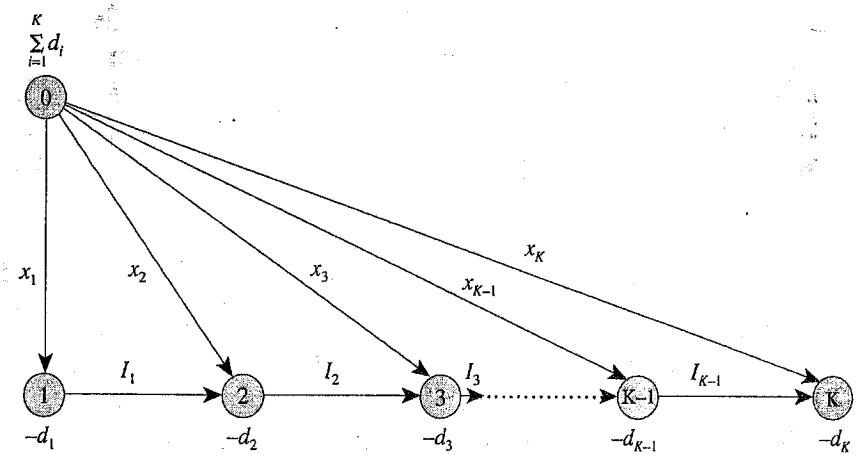


Figure 19.17 Network flow model of the dynamic lot-size problem.

If the production and inventory carrying costs are linear, we can easily solve the dynamic lot-size problem as a shortest path problem as follows. We determine a shortest path from node 0 to each node j and send d_j units of flow along that path; this path gives the minimum cost production-inventory schedule that satisfies the demand in period j. If we impose capacities on the production and inventory in each period, the dynamic lot-size problem no longer separates into independent shortest path problems. For example, if the costs are linear and we impose production or inventory capacities, the problem becomes a minimum cost flow model. We next consider situations with nonlinear production costs.

Application 19.20 Dynamic Lot Sizing with Concave Costs

In practice, the production x_j in the jth period frequently incurs a fixed cost F_j (independent of the level of production) and a per unit production cost c_j. Therefore, for each period j, the flow cost on the production arc $(0, j)$ is 0 if $x_j = 0$ and $F_j + c_j x_j$ if $x_j > 0$, which is a concave function of the production level x_j. (The production cost might also be concave, due to other economies of scale in production.) We assume that the inventory carrying cost is linear and that we have no capacity imposed upon production and inventory. So this problem is an instance of the concave cost flow problem.

In Exercise 14.13 we showed that any concave cost flow problem always has an optimal spanning tree solution (i.e., for some spanning tree, the tree arcs may carry nonzero flows and nontree arcs have zero flow). As we saw in Chapter 11, the fact that the spanning tree arcs must carry nonnegative flows, restricts our choice of spanning trees: Only some of them define a feasible solution of the dynamic lot-size problem. Any such feasible spanning tree decomposes into disjoint directed paths from the source node 0; the first arc on each path is a production arc and every other arc is an inventory carrying arc (see Figure 19.18). This observation implies the following *production property*: In the solution, each time we produce, we produce enough to meet the demand for an integral number of contiguous periods. Moreover, in no period do we both carry inventory from the previous period and produce.

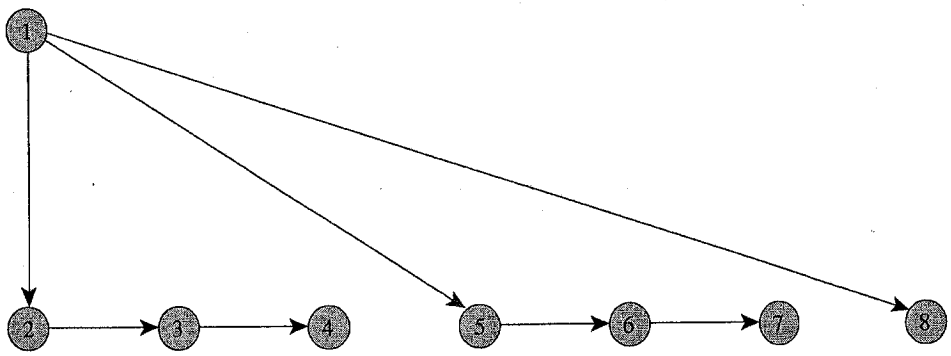


Figure 19.18 Illustrating the production property.

The production property permits us to solve the problem very efficiently as a shortest path problem on an auxiliary network G' defined as follows. The network G' consists of nodes 1 through $K + 1$ and contains an arc (i, j) for every pair of nodes i and j satisfying $i < j$. We set the cost of each arc (i, j) equal to the production and inventory carrying costs incurred in satisfying the demands of periods i, $i + 1$, $\ldots$, $j - 1$ by the production in period i. Observe that for every production schedule satisfying the production property, G' contains a directed path from node 1 to node $K + 1$ with the same objective function value, and vice versa. Therefore, we can obtain the optimal production schedule by solving a shortest path problem.

Application 19.21 Dynamic Lot Sizing with Backorders

In the preceding model we obtained the optimal production-inventory schedule assuming that we must satisfy the demands exactly in each period. In a more general version of this model, we permit backordering, which implies that we might not fully satisfy the demand of any period from the production in that period or from current inventory, but could fulfill the demand from production in future periods. In this model we assume that we do not lose any customer whose demand is not satisfied on time and who must wait until his or her order materializes. Instead, we assume that we incur a penalty cost for backordering any item. Figure 19.19 shows the network flow model for this situation. This model is the same as that of the dynamic lot-size model, except that we have additional *backorder carrying arcs* $(j, j-1)$ for each $j = 2, 3, \ldots, K$, which represent the flow of backorders.

As in the preceding model, we assume that production costs are concave, and inventory carrying costs are linear, as are the costs of backordering. (Backordering does not affect the demand but incurs intangible cost in the form of lost goodwill of the customer.) We also assume that production, inventory, and backordering (and so the corresponding arcs) have no capacity limitations. This problem, again, is a

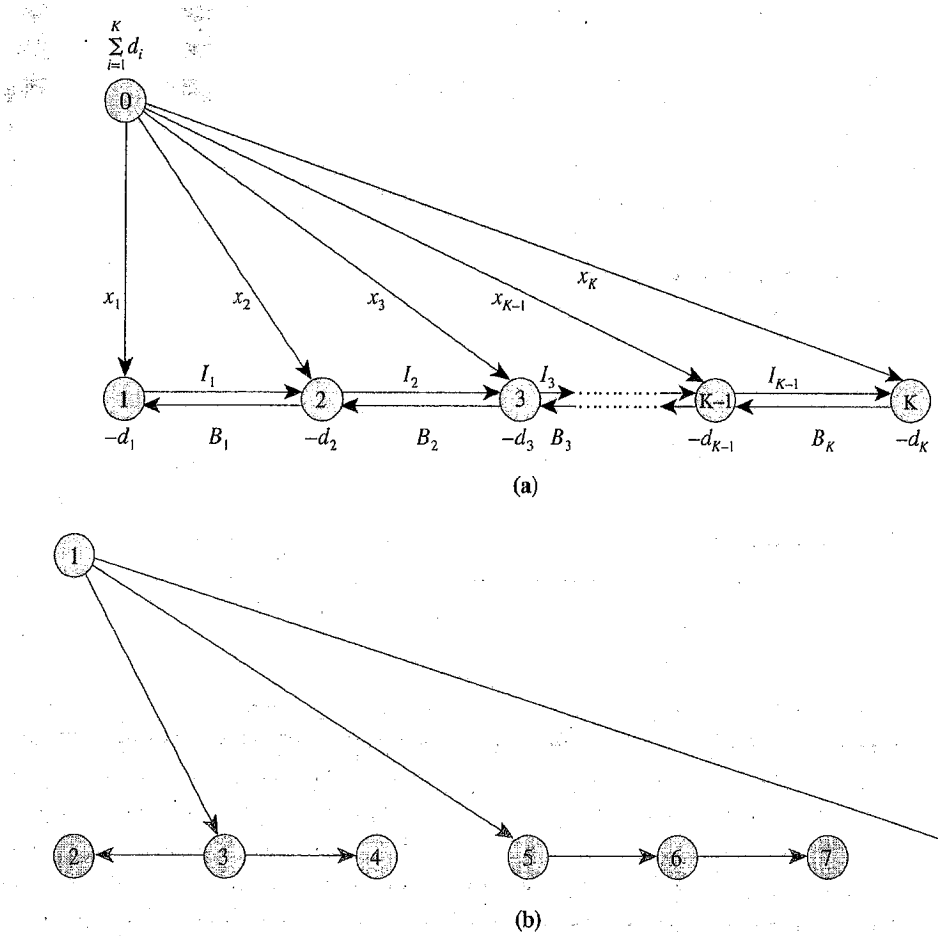


Figure 19.19 Dynamic lot-size problem with backorders: (a) network for the dynamic lot-size problem; (b) graphical structure of a spanning tree solution.

concave cost flow problem and has an optimal spanning tree solution. A feasible spanning tree solution in this case could, however, be different from the no-backorder case that we discussed in the last application. Although the optimal solution still satisfies the production property, we can use the production in period j to satisfy previous as well as future demands. Figure 19.19(b) shows an instance of the spanning tree solution.

The shortest path network G' for this problem is the same as the case without backorders except that we define the cost of arc (i, j) differently. In the case with backorders, we set the cost of arc (i, j) equal to the sum of the production, inventory carrying, and backorder carrying costs incurred in satisfying the demands of periods $i, i + 1, \ldots, j - 1$ by producing in some period k between i and $j - 1$; we select the period k that gives the least possible cost. In other words, we vary k from i to $j - 1$, and for each k, we compute the cost incurred in satisfying the demands of periods i through $j - 1$ by the production in period k; the minimum of these values defines the cost of arc (i, j) in the auxiliary network G'. (Recall that in the situation without backordering, we always chose period i as the production period for meeting the demand in the periods between i and $j - 1$.)

Application 19.22 Multistage Production-Inventory Planning

In the dynamic lot-size model, we considered a simple production process with a single stage of production. Often to produce a product, we must perform a sequence of operations, possibly performed on different machines at different times. If these machines have different production capacities that possibly vary from period to period, a multistage model would be a better approximation of this planning situation. In this case we treat each production operation as a separate stage and require that the product pass through each of the stages before its production is complete. We will use the following notation:

d_j: demand of the product in period j
x_{kj}: amount of the product produced at stage k in period j
I_{kj}: inventory of the product carried from period j to $j + 1$ at stage k
P_{kj}: production capacity at stage k in period j
u_{kj}: upper limit on inventory holding from period j to $j + 1$ at stage k
c_{kj}: unit production cost at stage k in period j
h_{kj}: inventory carrying cost from period j to $j + 1$ at stage k

Figure 19.20 shows a minimum cost flow model of this problem. The network has $KT + 1$ nodes: For each $k = 1, \ldots, K$, and for each $j = 1, \ldots, T$, the node j^k represents the kth stage in period j. The decision variable next to each arc in the network models the flow on that arc; c_{kj} and P_{kj} are the cost and capacity for the arc representing the production x_{kj} at stage k in period j; h_{kj} and u_{kj} are the cost and capacity for the arc representing the inventory I_{kj} carried from period j to $j + 1$ at stage k. As is evident from the construction of the network, there is one-to-one

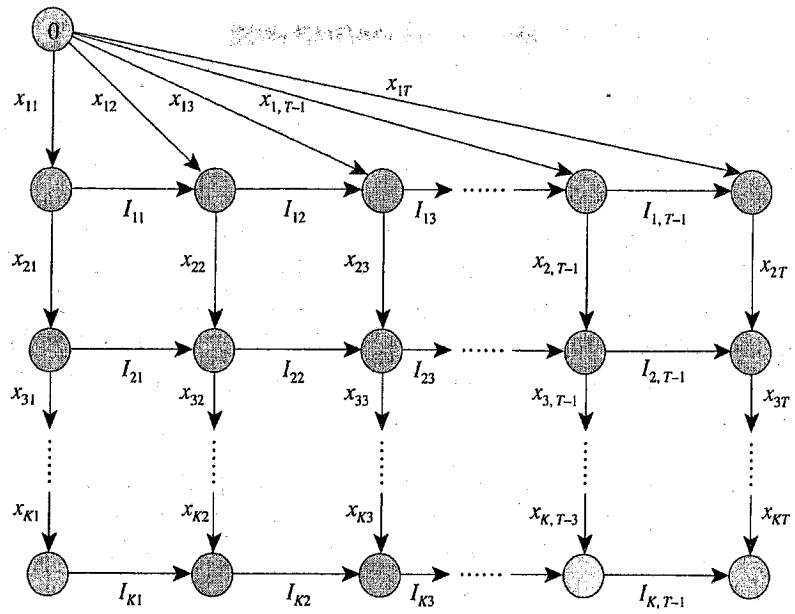


Figure 19.20 Network model for the multistage production-inventory planning problem.

correspondence between feasible flows in the network and production schedules. Consequently, we can obtain a minimum cost production-inventory schedule by finding a minimum cost flow in the network.

The model we have considered assumes that the production cost in each period is linear in the production quantity. If these costs are concave (e.g., because of fixed production costs for producing any quantity in any period), the model becomes a more complex (NP-hard) mixed integer programming model.

Application 19.23 Multiproduct Multistage Production-Inventory Planning

The multistage model that we just discussed modeled the production and inventory operations of only a single item. In more general situations we would need to consider not only multiple stages but also multiple products. Suppose that r different products share the common manufacturing facilities in p stages, and we want to meet the prescribed demands of the products in each period. The underlying network model is the same as that shown in Figure 19.20 with the exception that H distinct commodities flow on each arc. Consequently, the resulting model is an integer minimum cost multicommodity flow model. Clearly, the multiproduct multistage model is significantly more difficult than the single-product multistage model. Accordingly, to solve the model, we might use heuristic procedures that exploit the underlying network structure, or we might use general-purpose solution methods from integer programming and combinatorial optimization. For example, as shown in Chapter 16, where we examine a multiproduct, single-stage version of this problem, we might use Lagrangian relaxation and solve a sequence of single-product subproblems as shortest path problems.

Application 19.24 Mold Allocation

This application concerns a mold allocation problem that arises in the tire industry. A manufacturing plant in this industry consists of w cavities (or presses). At the beginning of each scheduling period, the manufacturing plant will insert different mold types (assume that the plant has p different mold types, one for each type of tire) into the cavities and subsequently use each mold to produce a fixed number of tires in that scheduling period. In the next scheduling period, depending on the demand requirements and cost structure, the plant might change some mold types in the cavities. In the mold allocation problem, we would like to assign molds to cavities over a scheduling horizon of T scheduling periods in order to satisfy the following conditions:

1. *Changeover restrictions*. We incur a setup cost whenever we place a mold in a cavity. This cost includes the cost of setting the mold in the cavity and performing quality control testing of the initial output of the mold. Highly skilled personnel perform these operations using specialized equipment; the limited availability of these personnel and of the specialized equipment normally restricts the number of setups that we can perform at the beginning of any scheduling period. Let R_j denote the maximum number of setups that we can perform at the beginning of the jth scheduling period. Let c_i denote the cost of placing a mold of type i in a cavity. In the mold allocation problem, our objective is to assign molds to cavities in a way that minimizes the total cost of placing the molds in the cavities.
2. *Minimum tire requirements*. The number of molds of a given type in place in the cavities during each scheduling period must meet a management-specified minimum. These limits originate from agreements assuring independent distributors with minimum production quantities in each period. Let r_{ij} denote the minimum number of molds of type i required in the jth scheduling period.
3. *Mold availability constraints*. The number of molds of a given type in place in the cavities during a given scheduling period is limited by the number of molds of that type available during that scheduling period. Mold availability might vary from period to period because of the planned arrival of new molds and the "reworking" of molds to convert them from one type to another. Let a_{ij} denote the molds of type i available in the jth scheduling period.
4. *Plant capacity limitations*. The total number of molds in place in the cavities during any scheduling period cannot exceed the number w of available cavities.

Figure 19.21 shows the minimum cost flow formulation of the mold allocation problem for $p = 2$ and $T = 2$. Next to each arc, we give the cost, lower bound, and capacity for the arc; for those arcs without any accompanying data, $(c_{ij}, l_{ij}, u_{ij}) = (0, 0, \infty)$. This formulation has T layers of nodes, one for each scheduling period. In our example, the first layer has nodes 1^0, 1^1, 1^2, 1^*, and the second layer has nodes 2^0, 2^1, 2^2, 2^*. In general, the layer i has $p + 2$ nodes numbered i^0, i^1, i^2, i^3, $\ldots$, i^p, i^*. The network essentially represents the flow of cavities. Initially, assume that V_i cavities contain mold i, and V_0 of the cavities are empty. Notice that $V_0 + V_1 + \cdots + V_p = w$. We represent this relationship by introducing for each mold

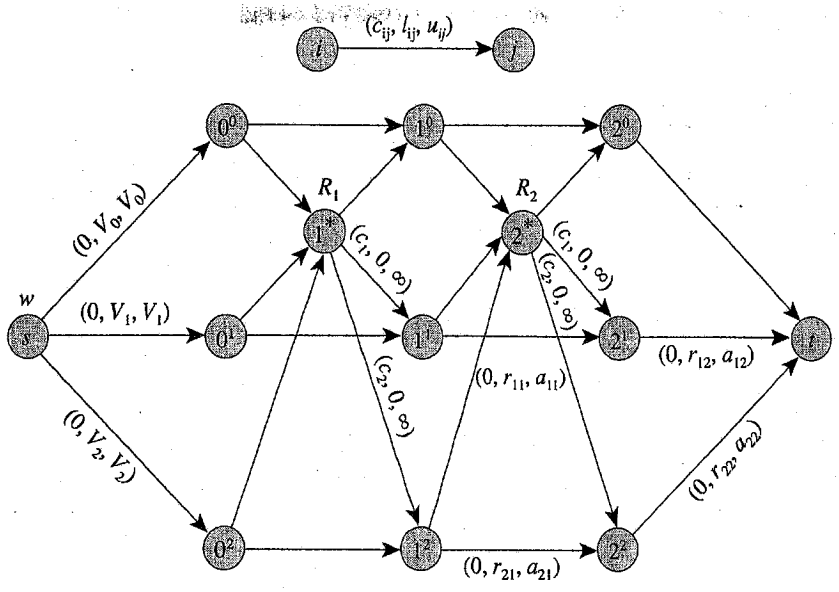


Figure 19.21 Minimum cost flow formulation of the mold allocation problem.

type i, arcs $(s, 1^i)$ whose lower and upper bounds are equal to V_i; node s has a throughput capacity of w. At the beginning of the first scheduling period, we might change molds in the existing cavities. We accomplish this by using the node 1*, which collects all such cavities (including the empty ones) through the arcs $(0^0, 1^*)$, $(0^1, 1^*)$, and $(0^2, 1^*)$, retains some of these cavities as empty [by sending flow on arc $(1^*, 1^0)$], and changes the molds in the remaining cavities [by sending flow on arcs $(1^*, 1^1)$ and $(1^*, 1^2)$]. We allow a maximum of R_1 changes in the first scheduling period. We impose this restriction by setting the capacity of the node 1* equal to R_1. The arc $(1^*, 1^1)$ carries the cavities in which we place mold 1; therefore, this arc has cost c_1. Similarly, the arc $(1^*, 1^2)$ carries the cavities in which we place mold 2, and so has cost c_2. Since the flow on the arc $(1^1, 2^1)$ is the total number of cavities that contain mold 1 in the first scheduling period, we impose a lower bound of r_{11} and an upper bound of a_{11} on this arc. Similarly, the flow on the arc $(1^2, 2^2)$ represents the number of cavities that contain mold 2, and flow on the arc $(1^0, 2^0)$ represents the number of empty cavities. This network has capacities on some nodes; we can transform node capacities into arc capacities by performing the node splitting transformation described in Section 2.4.

This discussion shows that there is one-to-one correspondence between feasible flows in the network and feasible mold allocations. Consequently, a minimum cost flow would prescribe a minimum cost mold allocation plan.

19.10 SUMMARY

This chapter complements our discussion in previous chapters by introducing a number of additional applications of network flows. The applications are of two types: (1) broad generic problem types (the maximum weight closure of a graph, data scaling, dynamic flows, and arc routing), and (2) contextual applications (DNA sequencing, automatic karyotyping of chromosomes, facility layout and location, project

management, and production and inventory planning). Within the generic problem types, we have considered a number of more specific applications: open pit mining, selecting freight handling terminals, optimal destruction of military targets, the flyaway kit problem, scaling linear programming constraint matrices; in addition, building evacuations, patrolling streets by police cars, routing of street sweepers and household refuse collection vehicles, fuel oil delivery, and spraying roads during snowstorms. As part of this discussion we have also examined some classical network optimization models, including the directed and undirected Chinese postman problems, critical path scheduling, and various dynamic models of economic lot sizing.

As shown by the following tables, these applications, when combined with those that we have examined in previous chapters, demonstrate the ability of network flows to model an extraordinarily wide range of problems met in practice. Our choice of the categories in these tables is somewhat arbitrary, yet does provide at least one

Engineering
1. Leveling mountainous terrain (Application 1.4)
2. Rewiring of typewriters (Application 1.5)
3. Measuring homogeneity of bimetallic objects (Application 1.7)
4. Electrical networks (Application 1.8)
5. The paragraph problem (Exercise 1.7)
6. Network reliability testing (Application 8.2)
7. Equipment replacement (Application 9.6)
8. Phasing out capital equipment (Exercise 9.6)
9. Terminal assignment problem (Exercise 9.7)
10. Capacity expansion of a network (Exercise 14.4)
11. Dual completion of oil wells (Application 12.4)
12. Cabling electrical panels (Application 13.1)
13. Constructing pipeline networks (Application 13.1)
14. Managing energy or mineral networks (Application 15.1)
15. Stick percolation problem (Application 14.4)
16. Machine loading (Application 15.2)
17. Network design (Application 16.4)
18. Open pit mining (Application 19.1)
19. Flyaway kit problem (Application 19.4)

Manufacturing, production, and inventory planning
1. Assortment of structural steel beams (Application 1.2)
2. Allocating inspection effort on a production line (Application 4.2)
3. Machine setup problem (Exercise 6.19)
4. Entrepreneur's problem (Exercise 9.1)
5. Caterer problem (Exercises 11.2 and 15.1)
6. Optimal depletion of inventory (Application 12.8)
7. Two-duty operator scheduling (Application 16.5)
8. Multi-item production planning (Application 16.7)
9. Warehousing of seasonal products (Application 17.2)
10. Warehouse layout (Application 19.17)
11. Determining minimum project duration (Application 19.9)
12. Just-in-time scheduling (Application 19.10)
13. Time–cost trade-off in project management (Application 19.11)
14. Dynamic lot sizing (Application 19.19)
15. Dynamic lot sizing with concave costs (Application 19.20)
16. Dynamic lot sizing with backorders (Application 19.21)
17. Multistage production-inventory planning (Application 19.22)
18. Multiproduct multistage production-inventory planning (Application 19.23)
19. Mold allocation (Application 19.24)

Scheduling

1. Police patrol problem (Exercise 1.9)
2. Telephone operator scheduling (Application 4.6)
3. Single-duty crew scheduling (Exercise 4.13)
4. Scheduling on uniform parallel machines (Application 6.4)
5. Tanker scheduling problem (Application 6.6)
6. Nurse scheduling problem (Exercises 6.2 and 11.1)
7. Airline scheduling problem (Exercise 6.32)
8. Scheduling with deferral costs (Application 9.5)
9. Optimal capacity scheduling (Application 9.6)
10. Employment scheduling (Application 9.6)
11. Scheduling on parallel machines (Application 12.9)
12. School bus driver assignment (Exercise 12.1)
13. Multivehicle tanker scheduling (Application 17.3)

Management science

1. Compact book storage in libraries (Exercise 4.3)
2. Personnel planning (Exercise 4.9)
3. Optimal storage policy for libraries (Exercise 9.3)
4. Zoned warehousing (Exercise 9.4)
5. Project assignment (Exercise 11.3)
6. Faculty–course assignment (Exercise 11.6)
7. Bipartite personnel assignment (Application 12.1)
8. Nonbipartite personnel assignment (Application 12.2)
9. Managing financial investments (Application 15.1)
10. Managing warehousing goods and funds flows (Application 15.3)
11. Managing foodgrain imports–exports (Application 17.1)

Physical and medical sciences

1. Disease categorization (Exercise 4.6 and Application 13.5)
2. Reconstructing the left ventricle from X-ray projections (Application 9.2)
3. Determining chemical bonds (Application 12.5)
4. Storing sequences of amino acids in proteins (Application 13.4)
5. DNA sequence alignment (Application 19.7)
6. Automatic karyotyping of chromosomes (Application 19.8)

Defense

1. Locating objects in space (Application 12.6)
2. Matching moving objects (Application 12.7)
3. Optimal destruction of military targets (Application 19.3)
4. Network interdiction problem (Exercise 19.18)

Computer science and communication systems

1. Concentrator location on a line (Exercises 4.7 and 4.8)
2. Distributed computing on a two-processor computer (Application 6.5)
3. Allocating receivers to transmitters (Exercise 11.5)
4. Constructing digital computer systems (Application 13.1)
5. Designing computer networks (Application 13.1)
6. Area transfers in communication networks (Application 14.2)
7. Message routing in computer and communication networks (Application 17.1)
8. Reducing data storage (Application 13.4)

Social sciences and public policy

1. Reallocation of housing (Application 1.1)
2. Determining an optimal energy policy (Application 1.9)
3. Racial balancing of schools (Applications 1.10 and 9.3)
4. Police patrol problem (Exercise 1.9)
5. Forest scheduling problem (Exercise 1.10)
6. Large-scale personnel assignment (Exercise 1.4)
7. Problem of representatives (Application 6.2)
8. Matrix rounding problem (Application 6.3)
9. Statistical security of data (Exercise 6.5 and Application 8.3)
10. Allocation of contractors to public works (Exercise 9.5)
11. Assigning medical school graduates to hospitals (Application 12.3)
12. Optimal message passing (Application 13.2)
13. Designing physical systems (Application 13.1)
14. Matrix balancing problems (Application 14.3)
15. Land management (Application 15.4)
16. Optimal deployment of resources (Exercise 17.1)
17. Models for building evacuation (Application 19.13)
18. Parking model (Exercise 19.17)
19. Optimal deployment of firefighting companies (Exercise 19.21)

Applied mathematics

1. Approximating piecewise linear functions (Application 4.1)
2. Knapsack problem (Application 4.3)
3. Cluster analysis (Exercise 4.6 and Application 13.5)
4. System of difference constraints (Application 4.5)
5. Finding feasible network flows (Application 6.1)
6. Matrix rounding problem (Application 6.3)
7. Solving a system of equations (Exercise 6.4)
8. Linear programs with consecutive ones in columns (Application 9.6)
9. Capacitated minimum spanning tree problem (Exercise 9.54)
10. Fractional b-matching problem (Exercise 9.55)
11. Bottleneck transportation problem (Exercise 9.56)
12. Linear programs with consecutive 1's in rows (Exercise 9.8)
13. Linear programs with circular 1's in rows (Exercise 9.9)
14. Optimal rounding of a matrix (Exercise 11.7)
15. All pairs minimax path problem (Application 13.3)
16. Traveling salesman problem (Application 16.2)
17. Degree-constrained minimum spanning trees (Application 16.6)
18. Asymmetric data scaling with lower and upper bounds (Application 19.5)
19. Minimum ratio asymmetric data scaling (Application 19.6)
20. Symmetric data scaling problems (Exercise 19.6)
21. Maximum dynamic flows (Application 19.12)

plausible way to group the various applications. Several of the applications in these tables fit into several of the categories; rather than attempting to capture all such overlaps, with only a few exceptions we have placed each application into a single table, usually reflecting the application's primary application context. (In a few cases, an application combines several problems contexts; to help us obtain a better picture of the application contexts, we have included citations to several of these applications more than once.) We encourage readers to scan all of these tables to obtain a broad picture of the applications we have considered and to discover connections between the various categories.

We note that these tables underestimate the full range of applications that we have considered in our discussion throughout the text. For example, the tables con-

Distribution and transportation
1. Tramp steamer problem (Application 4.4)
2. Minimax transportation problem (Exercise 6.6)
3. Optimal loading of a hopping airplane (Application 9.4)
4. Distribution problems (Application 9.1)
5. Vehicle fleet planning (Exercise 9.2)
6. Passenger routing (Exercise 11.4)
7. Pilot assignments (Application 12.2)
8. Designing rural road networks (Application 13.1)
9. Aircraft assignment (Application 15.2)
10. Vehicle routing problem (Application 16.3)
11. Routing of railcars (Application 17.1)
12. Distribution systems planning for multiple products (Application 17.1)
13. Selecting freight handling terminals (Application 19.2)
14. Directed Chinese postman problem (Application 19.14)
15. Undirected Chinese postman problem (Application 19.15)
16. Discrete location problems (Application 19.16)
17. Rectilinear distance facility location (Application 19.18)
18. Truck scheduling problem (Exercises 19.19 and 19.20)
19. Dynamic facility location (Exercise 19.22)

Miscellaneous
1. Pruned chessboard problem (Exercise 1.6)
2. Dating problem (Exercise 1.5)
3. Seat-sharing problem (Exercise 1.8)
4. Bridges of Königsberg (Exercise 2.6)
5. Knight's tour problem (Exercise 3.25)
6. Maze problem (Exercise 3.26)
7. Wine division problem (Exercise 3.27)
8. Money-changing problem (Exercise 4.5)
9. Dining problem (Exercise 6.1)
10. Ski instructor's problem (Exercise 12.2)
11. Dancing problem (Exercise 12.12)
12. Tournament problem (Application 1.3)
13. Optimal coverage of sports events (Exercise 6.41)
14. Baseball elimination problem (Application 8.1)
15. Balanced assignment problem (Exercise 12.24)
16. Optimal message passing (Application 13.2)
17. Urban traffic flows (Application 14.1)

tain a single entry for discrete location problems, even though, as pointed out in this chapter, this application class applies to problem contexts in (1) the public and service sectors such as hospitals, fire stations, libraries, and air field locations, (2) technical arenas such as the layout of the component of a control panel, and (3) architectural design, such as the location of water fountains in an office building.

We also note that the coverage of applications in this book, as broad as it is, is not intended to be exhaustive. The applications we have discussed typically are classical, are easy to describe, or are representative of a broad class of problems. The literature contains many other applications, some that amplify on the themes we have presented and some that treat new problem domains. We hope that when viewed in its entirety, our coverage in this book gives an appreciation for the power of network flows as a field that not only has substantial intellectual content, but also has an important impact on practice.

REFERENCE NOTES

In this chapter we described many applications of network flow problems. We have adapted these applications from the following papers:

1. Open pit mining (Johnson [1968])
2. Selecting freight handling terminals (Rhys [1970])
3. Optimal destruction of military targets (Orlin [1987])
4. Flyaway kit problem (Mamer and Smith [1982])
5. Asymmetric data scaling with lower and upper bounds (Orlin and Rothblum [1985])
6. Minimum ratio asymmetric data scaling (Orlin and Rothblum [1985])
7. DNA sequence alignment (Waterman [1988])
8. Automatic karyotyping of chromosomes (Tso, Kleinschmidt, Mitterreiter, and Graham [1991])
9. Determining minimum project duration (Elmaghraby [1978])
10. Just-in-time scheduling (Elmaghraby [1978], Levner and Nemirovsky [1991])
11. Time–cost trade-off in project management (Fulkerson [1961a], Kelley [1961])
12. Maximum dynamic flows (Ford and Fulkerson [1958a])
13. Models for building evacuation (Chalmet, Francis, and Saunders [1982])
14. Directed Chinese postman problem (Edmonds and Johnson [1973])
15. Undirected Chinese postman problem (Edmonds and Johnson [1973])
16. Discrete location problem (Francis and White [1976])
17. Warehouse layout (Francis and White [1976])
18. Rectilinear distance facility location (Cabot, Francis, and Stary [1970])
19. Dynamic lot sizing (Veinott and Wagner [1962])
20. Dynamic lot sizing with concave costs (Zangwill [1969])
21. Dynamic lot sizing with backorders (Zangwill [1969])
22. Multistage production-inventory planning (Zangwill [1969])
23. Multiproduct multistage production-inventory planning (Evans [1977])
24. Mold allocation (Love and Vemuganti [1978])

Additional applications of network flow problems can be found throughout this book, in the text and in the exercises. We provide numerous citations to the literature in the reference notes of Chapters 1, 4, 6, 9, 12, 13, 14, 15, 16, and 17. We have adapted many applications in this book from the paper of Ahuja, Magnanti, Orlin, and Reddy [1992]. Since the applications of network flow models are so pervasive, no single source provides a comprehensive account of network flow models and their impact on practice. Several researchers have prepared general surveys of selected application areas. Notable among these are the papers by Bennington [1974], Glover and Klingman [1976], Bodin, Golden, Assad, and Ball [1983], Aronson [1989], Bazaraa, Jarvis, and Sherali [1990], and Glover, Klingman, and Phillips [1990]. The book by Gondran and Minoux [1984] also describes a variety of applications of network flow problems.

EXERCISES

19.1. Show that minimizing $|\alpha - \beta|$ is equivalent to minimizing $(c + d)$, subject to the conditions $\alpha - \beta = c - d$, $c \geq 0$, and $d \geq 0$. (*Hint*: Consider the cases when $\alpha \geq \beta$ and $\alpha \leq \beta$.)

19.2. Write a linear programming formulation of the maximum weight closure problem. Show that the dual of this linear program is a minimum cost flow problem.

19.3. This exercise concerns the least cost interdiction of a physical transportation network for carrying supplies, troops, or arms during a large-scale conventional war. Let $G = (N, A)$ denote a transportation network with two distinguished nodes s and t and let c_{ij} denote the cost of destroying arc (i, j) in this network.

(a) Suggest a method for identifying the cheapest way to destroy a subset of arcs that will disconnect nodes s and t.

(b) Suppose that besides destroying arcs of the transportation network, we could also destroy nodes of the network. Destroying a node in the network is equivalent to destroying all the arcs incident to that node. Let c_i denote the cost of destroying a node i in N. Identify the cheapest way to disconnect node s from node t.

19.4. This exercise concern the flyaway kit problem that we discussed in Application 19.4.

(a) Suppose that the penalty cost for all jobs is the same; that is, for all j, $L_j = \alpha$ for some constant α. Then what is the optimal policy when $\alpha = 0$ or when $\alpha = \infty$?

(b) Our formulation assumes that each job requires at most one item of each part type. Modify the formulation so that it allows multiple parts of the same type. Justify your modification.

19.5. This exercise concerns the asymmetric data scaling problem with lower and upper flow bounds that we discussed in Application 19.5. Suppose that you want to find the row multipliers α_i and the column multipliers β_j so that $l \le \alpha_i \mid a_{ij} \mid / \beta_j \le u$ for all matrix elements and, moreover, among all such multipliers, you want the ones with the smallest possible product $\Pi_{\{(i,j):a_{ij}\neq 0\}} \alpha_i \mid a_{ij} \mid / \beta_j$. How would you solve this problem as a network flow problem?

19.6. In this exercise we study a restriction of the asymmetric data scaling problem (discussed in Section 19.3) known as the *symmetric data scaling problem*. In this problem the matrix **A** is a $p \times p$ square matrix and we want to scale the data so that the diagonal elements of the matrix do not change. This implies that if the multiplier of row i is α_i, the multiplier of column i is $1/\alpha_i$.

(a) Describe how you would solve the symmetric data scaling problem with lower and upper flow bounds.

(b) Describe how you would solve the minimum ratio symmetric data scaling problem.

19.7. In the project scheduling problem that we discussed in Application 19.11, we assumed that the cost of each job (i, j) is a linear function of its duration. Suppose, instead, that the cost of job (i, j) is a piecewise linear convex function of its duration, with breakpoints at the job durations $0 = d_{ij}^0 \le d_{ij}^1 \le \cdots \le d_{ij}^p$; assume that the cost function has a slope of c_{ij}^k in the interval $[d_{ij}^{k-1}, d_{ij}^k]$. Specify a linear programming formulation of this problem and show that its dual is a convex cost flow problem.

19.8. Battle of the Marne (Berge and Ghouila-Houri [1962]). From the different towns a_1, a_2, . . . , a_n, buses leave to go to a single destination b. For any direct road joining town a_i and town a_j, we are given the number c_{ij} of buses that can leave town a_i for a_j in a unit of time; we also know the time t_{ij} required to make this journey. Given the number s_i of buses available at town a_i at the start of our planning horizon and the number c_i of buses that can be parked at town a_i, we want to organize traffic routes so that in a given interval of time T, the number of buses arriving at b will be as large as possible. Formulate this problem as a network flow problem.

19.9. This exercise is related to the maximum dynamic flow problem discussed in Application 19.12.

(a) Can the time-expanded replica of a network contain a directed cycle?

(b) Can you solve the maximum dynamic flow problem if arc capacities change from one time period to another? If yes, how?

(c) Show that the maximum dynamic flow problem can have multiple optimal solutions.

(d) For any optimal solution of the maximum dynamic flow problem, let v be a p-element vector whose kth element denotes the flow reaching the sink node at

time k. Suggest a method for identifying a solution with the lexicographically maximum vector v from among all solutions.

19.10. In a maximum dynamic flow x, let $v(k)$, for every $k = 1, 2, \ldots, p$, denote the amount of flow reaching the sink up to time period k. Clearly, $v(1) \leq v(2) \leq \cdots \leq v(p)$. Show that if p is sufficiently large, then for some index l, $v(i) - v(i-1) = \alpha$ for every $l \leq i \leq p$. What is α?

19.11. Consider the time-expanded replica G^p of a network G. Let $s^1, s^2, \ldots, s^p$ denote the sources and $t^1, t^2, \ldots, t^p$ denote the sinks in different time periods $1, 2, \ldots, p$. In this expanded network G^p, we wish to determine the smallest index r so that the maximum flow from the nodes in $s^1, s^2, \ldots, s^p$ to the nodes in $t^1, t^2, \ldots, t^p$ is at least B.

(a) Show how you can solve this problem in polynomial time using any maximum flow algorithm.

(b) Show how you can solve this problem by a single application of any minimum cost flow algorithm.

19.12. In our study of the dynamic maximum flow problem in Application 19.12, we assumed that capacities do not change over time. Explain how to model situations when the capacities are functions of time and change from period to period.

19.13. In a transportation network, let c_{ij} represent the time required to traverse arc (i, j) and assume, therefore, that $c_{ij} \geq 0$ for all arcs $(i, j) \in A$. In this network we wish to identify a directed path with the least possible traversal time from node s to node t. In the shortest path algorithms studied in Chapter 4, we assumed that all arc traversal times are fixed and do not change with time. But often in practice traversal times are functions of time: They are higher during rush hours and lower at other times. Explain how you could model the situation in which the traversal times vary with the time of the day. Assume that you want to obtain a shortest directed walk from node s to node t (the walk might sometimes be shorter than the directed path). State your assumptions.

19.14. Constrained shortest path problem. In a network G we associate two numbers with each arc: its length c_{ij} and its traversal time τ_{ij}. We would like to determine a shortest-length path from the source node s to the sink node t with the additional constraint that the traversal time of the path does not exceed τ. Formulate this problem as a shortest path problem in a time-expanded network.

19.15. Consider an undirected graph $G = (N, A)$ with arc costs c_{ij}. For any given subset $S \subseteq N$, we want to find a minimum cost subgraph of G that has an odd number of arcs incident to the nodes in S and an even number (possibly, zero) of arcs incident to the nodes in $N - S$. Describe a method for solving this problem. (*Hint*: Use the results in Application 19.15.)

19.16. As we discussed in Application 19.14, the optimal Chinese postman tour for directed network might traverse some arcs several times. Suppose that we can traverse no arc more than k times for some fixed $k \geq 2$. How would you solve this problem? Will this problem always have a feasible carrier tour?

19.17. Parking model (Dirickx and Jennergren [1975]). Develop a network model to solve the following parking problem that arises in the downtown area of a busy district. You are given a city district consisting of I blocks. Every block i has a daily demand D_{ik} for parking places of class $k = 1, \ldots, K$ and each class k has its own time length (0–1 hours, 1–2 hours, etc.). We measure the demand D_{ik} in physical parking places, which should be understood to mean that somewhere we must reserve D_{ik} parking places for vehicles whose drivers and passengers have their final destination in block i. Drivers might have to park in some other block and then walk by foot to block i. Suppose that the downtown area contains several public parking facilities (offstreet parking, garages, etc.). Let S_j denote the capacity of the jth parking facility. Our objective is to assign users to the parking facilities in order to minimize the total societal cost. The societal cost has two elements: walking costs, and the cost of maintaining

the parking facilities. Formulate this problem as a minimum cost flow problem. Describe your notation and state your assumptions.

19.18. Network interdiction problem (Fulkerson and Harding [1977]). Let $G = (N, A)$ represent the transportation network of a military opponent. Suppose that node s is our opponent's supply point, that node t is his demand point, and the length of arc $(i, j) \in A$ is c_{ij}. We can increase the length of any arc $(i, j) \in A$ by any positive amount $y_{ij} \geq 0$ units by spending $d_{ij}y_{ij}$ units of resources. Our task is to increase the difficulty of our opponent's supply task by increasing the length of the shortest path from node s to node t.

(a) Suppose that our objective is to increase the length of the shortest path from node s to node t to a value of at least λ_t while spending the least possible amount of resources. Show that we can solve this problem by solving a minimum cost flow problem. (*Hint*: Write the linear programming formulation and take its dual.)

(b) Suppose that our task requires that we design the network so that the length of the shortest path from node s to node i be λ_i, for all $i \in N - \{s\}$. How could we achieve this objective while spending the least amount of resources?

19.19. Truck scheduling problem (Gavish and Schweitzer [1974]). In this exercise we study a generalization of the tanker scheduling problem that we discussed in Application 6.6. A large trucking company must decide on a weekly basis how best to meet the demands for truck trips from its customers during the next week. Each customer demand is specified by the following information: (1) a quantity of cargo (in terms of the numbers of trucks), (2) a starting place and time for the job, and (3) the terminating place and time for completing the job. The company has a truck depot within its area of operation and a fleet of trucks adequate to meet the demands. The decision problem is to assign trucks from the depot to meet the demand at minimum cost. Explain how you would formulate this problem as a minimum cost flow problem. Use the following data in your formulation: (1) job j begins at time t_j^b, ends at time t_j^f, and requires r_j trucks; (2) c_j is the cost of (one-way) driving between the depot and the origin of job j; (3) d_j is the cost of performing job j; and (4) f_{ij} is the cost of driving from the destination of job i to the origin of job j; this trip requires τ_{ij} time. Assume that all trucks start at the depot and return to it after completing their jobs.

19.20. This exercise studies some generalizations of the truck scheduling problem that we considered in Exercise 19.19. Explain how you would incorporate the following additional constraints in the model. Consider each of these constraints independently.

(a) The trucking company has several, say k, depots for the supply of trucks. Let s^k denote the number of trucks available at the kth depot at the beginning of the week. You can assume that the trucks need not return to their own depot.

(b) Allow a "safety margin" of h_{ij} time for the direct driving time from the destination of job i to the origin of job j.

(c) The same truck cannot perform jobs i and j if doing so would mean that the truck would arrive too early at job j. We forbid any truck to wait more than L units of time to prevent the drivers from becoming accustomed to too much idle time.

(d) Certain jobs cannot be combined. For example, we cannot carry food directly after we have transported garbage.

19.21. Optimal deployment of firefighting companies (Denardo, Rothblum, and Swersey [1988]). This exercise studies an application of the transportation problem that arises in the deployment of firefighting companies. Its special nature results from the fact that the benefit obtained from the firefighting companies depends on their order of arrival. The company arriving at the scene first offers the greatest benefit, and as further companies arrive their benefits decrease.

We assume that q incidents are in hand and we want to assign firefighting companies located at p company locations to them. Let u_i denote the number of companies available at location i and v_j denote the number of companies required at incident j. We assume that $\sum_{i=1}^{p} u_i \geq \sum_{j=1}^{q} v_j$. Let τ_{ij} denote the travel time from company

location i to incident j. Furthermore, let α_{jk} denote the cost per unit time delay in the arrival of the kth company at incident j; assume that $\alpha_{jl} > \alpha_{jr}$ for $l < r$. If, for instance, the first two arrivals at incident j have identical travel times, one is given a cost of α_{j1} and the other is given a cost of α_{j2}. The objective is to assign firefighting companies to fire incidents to minimize the total cost. Formulate this problem as a minimum cost flow problem. Justify your formulation.

19.22. Dynamic facility location problem. A manufacturing facility supplies K consumer regions. The demand of each consumer region changes from period to period and is known in advance for each period in a planning horizon consisting of T periods. The company transports goods produced at its manufacturing facility to the consumer regions. The manufacturing facility is always located at one of the consumer regions, and due to fluctuating demands, the company might profitably relocate this facility occasionally. We wish to determine where to locate the manufacturing facility in each period of the planning horizon to minimize the total cost of transporting the goods and relocating the facility. Formulate this problem as a shortest path problem in a time-expanded network for the data shown in Figure 19.22 for which $T = 4$ and $K = 4$. Figure 19.22(a) gives the demand of the product in the next four periods, and Figure 19.22(b) gives the cost of transporting 1 unit of the product between consumer regions. Assume that the cost of relocating the facility is 300 times the cost of transporting 1 unit of the product. Moreover, assume that at the beginning of period 1, the facility is located in region 1.

Time period

Consumer region	1	2	3	4
1	100	50	50	80
2	200	300	250	150
3	100	50	200	250
4	75	125	125	200

(a)

Consumer region

Consumer region	1	2	3	4
1	—	5	2	3
2	5	—	1	6
3	2	1	—	4
4	3	6	4	—

(b)

Figure 19.22 Dynamic facility location problem.

Appendix A

DATA STRUCTURES

Knowledge is of two kinds. We know a subject ourselves or we know where we can find information on it.
—Samuel Johnson

Chapter Outline

A.1 Introduction
A.2 Elementary Data Structures
A.3 *d*-Heaps
A.4 Fibonacci Heaps

A.1 INTRODUCTION

Most network algorithms require the manipulation of data, particularly sets representing arc and node information, or representing trees or other network structures. Over the years, analysts have studied many different ways to store and manipulate data within computer memory; these investigations have shown, both empirically and in theory, that the choice of storage schemes often has a considerable effect on algorithmic performance. Indeed, by storing and manipulating sets more cleverly, we can often improve the worst-case complexity of an algorithm. In this appendix we describe some of the more common ways of storing and manipulating sets.

Often, elements of a set are ordered; following customary practice, we refer to such sets as *lists*. We typically perform several basic operations on lists: for example, inserting elements, deleting elements, determining whether a list contains a certain element. In Section A.2 we describe some of the more popular ways for storing lists and for performing these operations on them. The discussion focuses on data structures known as arrays, singly linked lists, doubly linked lists, queues, and stacks.

Sometimes we associate a number, or *key*, with each element of a set. We often need to perform some of the following operations on a set and its associated keys: finding an element in the set with the minimum key, inserting an element in the set, deleting an element, decreasing the key of an element. A *heap* is a data structure for sets that allows us to perform these operations efficiently. In Section A.3 we discuss binary and *d*-heaps. In Section A.4 we describe a more efficient (and, also, more sophisticated) heap known as the *Fibonacci heap*.

In this section we discuss some of the most popular ways of storing lists (i.e., ordered sets).

Arrays

An array is the simplest data structure used to store an ordered set. This representation uses an array of size n, called *list*; the ith position (or *index*) of the array contains the ith element of the set, which we denote by list[i]. Figure A.1 shows an array representation of the ordered set {10, 5, 8, 9, 7}. To keep track of the size of the array that contains data elements, we also use a variable *last* which we set equal to the number of elements in the list. For our example, last = 5, indicating that list contains no data elements in the positions 6, 7, . . . , n.

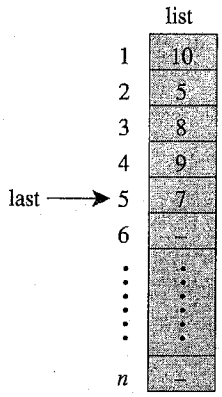


Figure A.1 Storing a set as an array.

Some operations are very convenient to perform using the array representation of a list. To determine the kth element of the list, we simply access the data element list(k). To establish whether list contains a given element α, we vary the index i from 1 to last and check whether list(i) = α. This operation requires time proportional to the number of elements in the set. In fact, all the storage methods described in this section require time proportional to the number of elements in a list to check for membership of any element. If we wish to insert an element at the end of the list, we increment last by one and store the new element as list(last).

An array is not very well suited for performing some other set operations. For example, suppose that we wish to delete the kth element of the set and $k <$ last. Deleting the kth element from the list changes the position of the elements stored at indices $k + 1, k + 2, \ldots$, last; we must shift each element back by one position. In the worst case, deleting the element and revising the list requires as many operations as the number of elements in the list, which is quite unattractive. Inserting an element in the middle of the list requires a similar amount of work. The linked list representations, discussed next, perform these operations much more efficiently but at the expense of using more storage.

Singly Linked Lists

Rather than store the elements of an ordered set sequentially as in an array, a singly linked list may store elements in an arbitrary order; however, it then requires additional information that permits us to access the data in the order specified in the ordered set.

A *cell* is the basic building block of linked lists. We can picture a cell as a box that is capable of holding several values, called *fields*. A singly linked list, then, is a collection of cells that are linked together. Each cell in the singly linked list has two fields: a *data* field and a *link* field. The data field holds the element of the list and the link field stores a pointer to the location of the next element in the list. Figure A.2(a) shows a geometric representation of a linked list for the set LIST = {5, 8, 9, 10}. The computer science community uses two popular methods for implementing linked lists: a pointer-based implementation and an array-based implementation. In this section we describe array-based implementations; in several references the reference notes of this appendix describe pointer-based implementations.

We can store a singly linked list by defining two arrays of size n, data and link. These two arrays define n cells, indexed from 1 through n: the kth cell consists of the fields *data*(k) and *link*(k). The data field of a cell contains an element of the set LIST and the link field contains the index of the cell containing the next element of the set. We also maintain a scalar called *first* that stores the index of the cell containing the first element of LIST. We set first to zero if the set is empty. Figure A.2(b) shows the array form of the linked list, LIST = {5, 8, 9, 10}. Because the list contains the element 9 in position 4 and next element 10 in position 2, data(4) = 9, link(4) = 2, and data(link(4)) = data(2) = 10.

It is fairly easy to manipulate singly linked lists. Suppose that we wish to scan all the elements of a set to determine membership of an element. We define a variable next = first. If next = 0, the set is empty and we stop. Otherwise, the first element of the set is data(next). Notice that link(next) gives the index of the following element in the set. Therefore, we update next = link(next) and check whether next equals

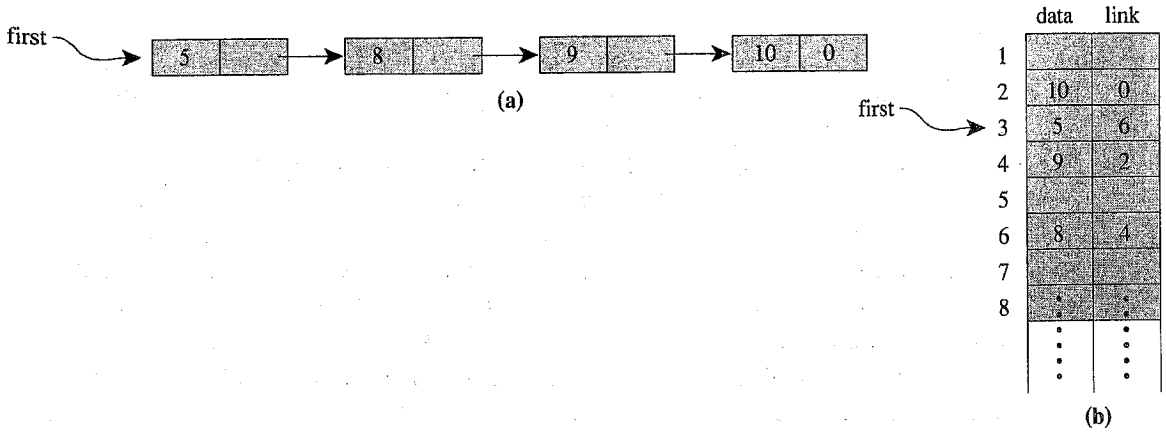


Figure A.2 Example of a singly linked list: (a) pointer representation; (b) array representation.

zero. If so, we stop; otherwise, the second element of the set is data(next). We repeat this process until next becomes zero. We suggest that the reader use this method to scan through the elements of the linked list shown in Figure A.2(b).

Now consider the insertion of an element into a linked list. Suppose, for example, that we wish to insert the element 6 into the linked list shown in Figure A.3(a). We first identify an unused cell with index *new*, and set data(new) = 6. If we wish to insert the element at the beginning of the list, we set link(new) = first and set first = new [as depicted in Figure A.3(a)]. If we wish to insert the element after the cell with index *prev*, we set link(new) = link(prev), link(prev) = new [as depicted in Figure A.3(b)]. This discussion shows that we can add an element to a linked list in $O(1)$ time.

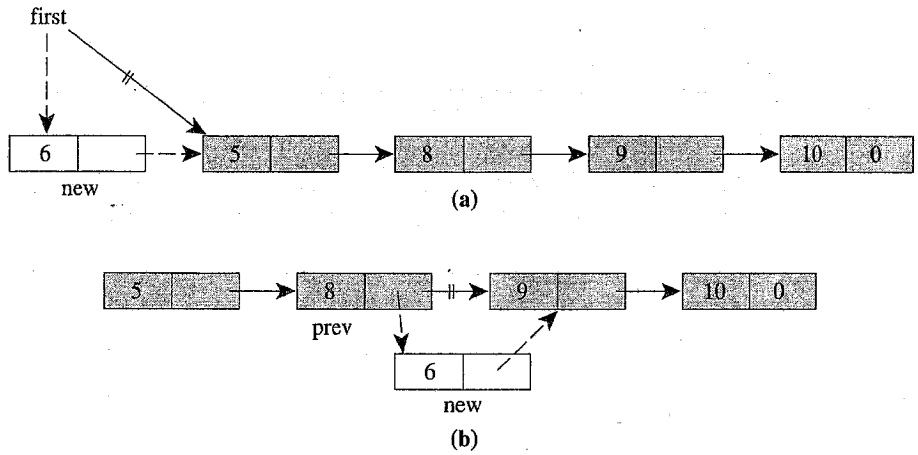


Figure A.3 Inserting an element in a linked list: (a) insertion at the beginning of the list; (b) insertion in the middle of the list.

Now consider the deletion of an element from the linked list; let us illustrate the process using the same example. As shown in Figure A.4(a), the deletion of the first element in the list is rather easy: We simply set next = first, and redefine first =

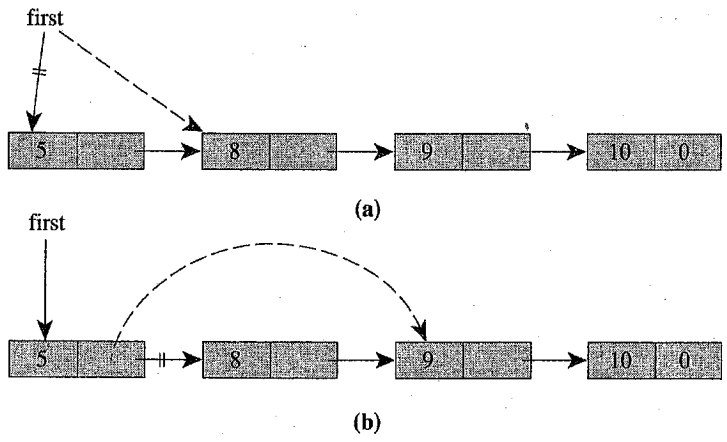


Figure A.4 Deleting an element from a linked list: (a) deletion from the beginning of the list; (b) deletion from the middle of the list.

link(next); therefore, first now points to the position of the second element of the original list. The deletion of an element from the middle of the list is more difficult. As shown in Figure A.4(b), to delete the element 8, we must modify the link field of the preceding cell so that its link array now points to the index of the cell following the element 8. So if we knew the preceding cell of the element to be deleted, we could easily perform the deletion in $O(1)$ time; otherwise, we will need to scan the linked list starting at the first element until we reach the element preceding the one to be deleted. The later operation in the worst case would require $O(n)$ time. To make deletion more efficient, we need to represent LIST as a doubly linked list, a data structure that we will discuss in the next subsection.

Some algorithms described in this book require the storage of sets LIST(1), LIST(2), . . . , LIST(n) of disjoint elements, each with a value between 1 and n. For example, for $n = 6$, we might have LIST(1) = {5, 4}, LIST(3) = {1, 2, 3}, LIST(4) = {6}, and LIST(2) = LIST(5) = LIST(6) = $\varnothing$. In these situations we will always add or delete elements from the front of any list. One plausible way to store this information would be by maintaining n different singly linked lists or n different arrays, but this method is not space efficient. In fact, we can store these n different sets using two n-dimensional arrays: *first* and *link*. Figure A.5 illustrates this storage scheme on this example data.

	first	link
1	5	2
2	0	3
3	1	0
4	6	0
5	0	4
6	0	0

Figure A.5 Storing multiple singly linked lists of disjoint elements.

In this storage scheme, first(k) stores the first element in the set LIST(k). If first(k) = 0; then LIST(k) is empty; otherwise, LIST(k) contains one or more elements. For example, first(1) = 5. Therefore, the first element in LIST(1) is 5. We then look at link(5), which is 4. Consequently, the second element in LIST(1) is 4. We then look at link(4), which is 0, indicating the end of the list. Therefore, LIST(1) = {5, 4}.

Suppose that we wish to insert an element p into LIST(k). Observe that because these lists are mutually disjoint, none of them can contain p. We insert element p to the beginning of LIST(k) by executing the statements link(p) = first(k) and first (k) = p. To delete the first element from LIST(k), we execute the statement first (k) = first(link(k)).

Doubly Linked Lists

A doubly linked list is the same as a singly linked list except that each cell has two links, one to the preceding cell and the other to the succeeding cell. Maintaining two links allows us to traverse the list easily in both directions and perform an arbitrary deletion of an element in $O(1)$ time.

A cell of the doubly linked list consists of three fields: *data, llink* (denoting left link), and *rlink* (denoting right link), which store the data element, the index of the preceding cell, and the index of the succeeding cell. For the set LIST = {5, 8, 9, 10}, Figure A.6(a) and (b) shows the pointer form and array form representations of the doubly linked lists.

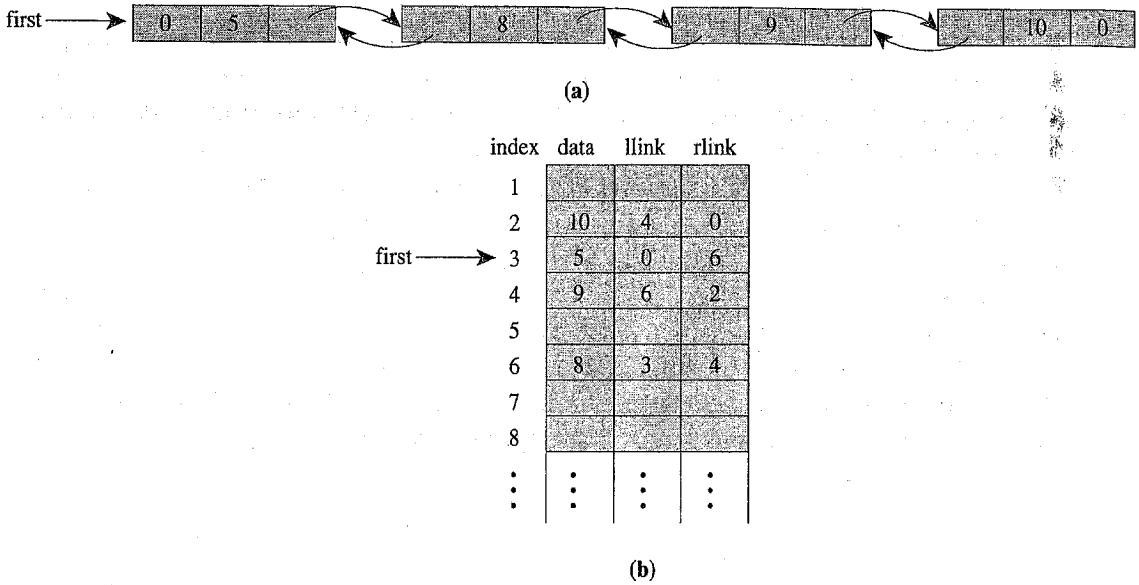


Figure A.6 Example of a doubly linked list: (a) geometric representation; (b) array representation.

We manipulate a doubly linked list in almost the same way in which we manipulate a singly linked list. We scan elements of the list in the same fashion, except that we can traverse the list in two directions: forward as well as backward. Using rlinks, we traverse the list in the forward direction (i.e., from left to right), and using llinks, we traverse the list in the backward direction (i.e., from right to left). Insertions and deletions from the doubly linked lists are easy to perform. Figure A.7

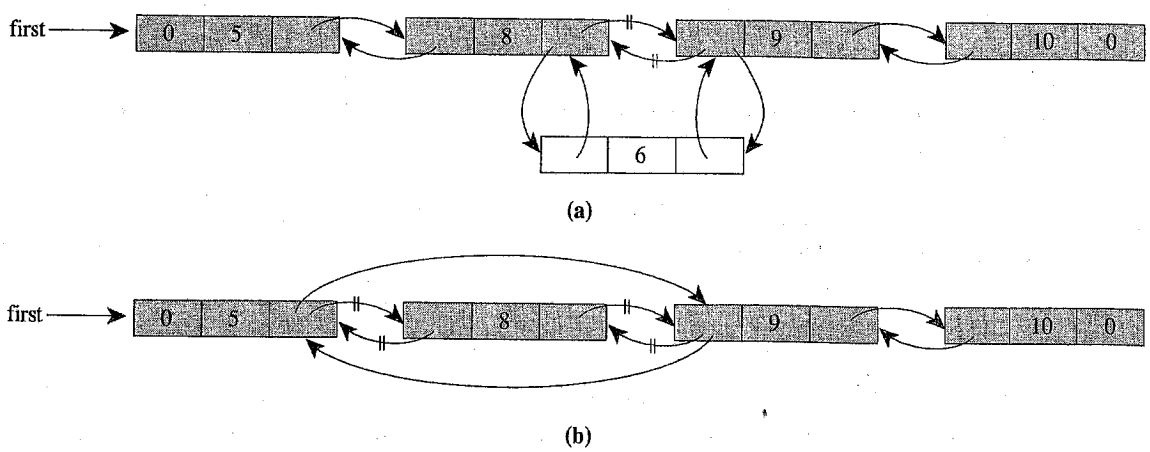


Figure A.7 Addition and deletion from a doubly linked list: (a) adding an element in the middle of the list; (b) deleting an element from the middle of the list.

shows how to apply these operations to the middle of a linked list. We ask the reader to show how to perform insertions and deletions from the beginning of the list. In each case, an insertion or deletion requires $O(1)$ time.

The doubly linked list allows us to traverse the list in either direction. Starting at some element i, by traversing rlinks, we can identify all the subsequent elements in the set. Traversing llinks allows us to identify all the elements preceding any element i. However, in some applications of doubly linked lists, we need to examine all the elements of the set starting at some arbitrary element i. We can do so by storing the elements in wraparound fashion (i.e., we set the rlink of the last element of the set equal to the index of the cell storing the first element and we set the llink of the first element of the set equal to the index of the cell storing the last element). We refer to this modification of the doubly linked list as a *circular doubly linked list*.

Finally, consider the storage of multiple doubly linked lists of disjoint elements, namely LIST(1), LIST(2), . . . , LIST(n). As in our earlier discussion, the value of each element lies between 1 and n. We can store these n different sets using three n-dimensional arrays: first, rlink, and llink. Figure A.8 illustrates this storage scheme for the previous example for which $n = 6$, LIST(1) = {5, 4}, LIST(3) = {1, 2, 3}, LIST(4) = {6}, and LIST(2) = LIST(5) = LIST(6) = $\varnothing$.

index	first	llink	rlink
1	5	2	0
2	0	3	1
3	1	0	2
4	6	0	5
5	0	4	0
6	0	0	0

Figure A.8 Storing multiple doubly linked list of disjoint elements.

Stacks

A *stack* is a special kind of ordered list (or set) in which all insertions and deletions take place at one end, called the *top*. The intuitive model of a stack is a pile of poker chips or a pile of dishes on a table; accordingly, we can conveniently remove only the top object on the pile or add a new one to the top. We can store a stack as an array or as a linked list. We shall discuss the array implementation of a stack.

A stack is represented by an n-dimensional array, *list*, that stores the element of a set, and a scalar *top* that denotes the index of the last entry to the array list. If the stack is empty, then top = 0. For example, suppose that we start with an empty list, and insert elements in the order 5, 4, 8, 7, 10; then at the end of the last step, the stack will appear as shown in Figure A.9. Since we have inserted five elements, when we have completed these operations, top = 5.

The most frequent operations performed on a stack are insertions and deletions. To insert a new element i on the stack, we increment top by 1 and set list(top) = i. To delete the topmost element i from the stack, we set i = list(top) and decrease top by 1. Clearly, both of these steps require $O(1)$ time. Notice that in a stack, we always remove the element that we added last. Consequently, if we store elements

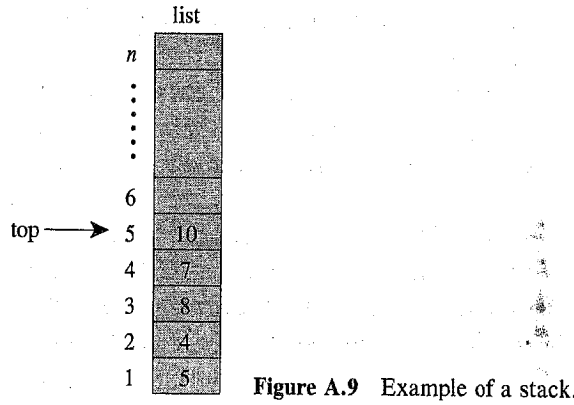


Figure A.9 Example of a stack.

using a stack and examine them one by one, we inspect the elements in a last-in, first-out (LIFO) order. Therefore, whenever we need to examine the elements of a dynamically changing set in the LIFO order, it is very natural to store the set as a stack.

Queues

A *queue* is another special kind of list, with elements inserted at one end (the *rear*) and deleted from the other end (the *front*). The operations for a queue are similar to those for a stack, the substantial difference being that insertions take place at the end of the list rather than the beginning. We see physical examples of queues everywhere since they are an integral part of contemporary society. Examples include lines at banks and at grocery stores, or items in a manufacturing plant waiting to be processed by a machine. A less apparent example is telephone calls waiting in a buffer for a telephone trunk to become available.

We represent a queue by an array *list* of size n that contains elements of the set. Figure A.10(a) gives an example of the queue. As shown in this example, we maintain a pointer called *front*, which is the index of the first position of the array

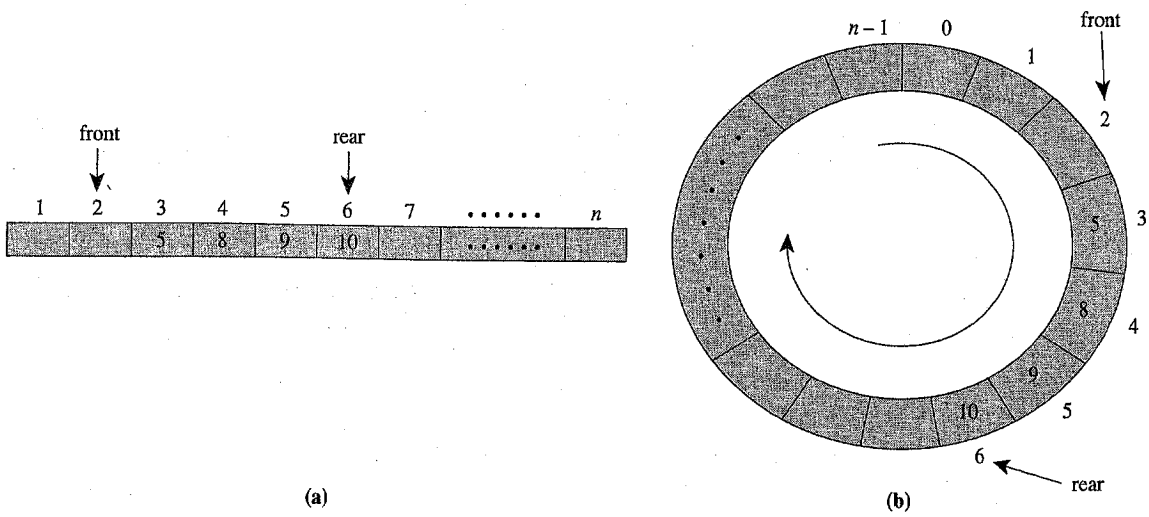


Figure A.10 Example of a queue: (a) sequential representation; (b) circular representation.

minus one, and maintain another pointer called *rear*, which is the index of the last position of the array.

We perform operations on a queue as follows. To check if the queue is empty, we simply check whether front equals rear. If so, the queue is empty; otherwise, it is nonempty. The frontmost element in the queue is list(front + 1). To insert a new element i in the queue, we increment rear by 1 and set list(rear) = i. To delete an element i from the queue, we set front = front + 1 and set i = list(front). Clearly, both the steps require $O(1)$ time. Notice that in a queue, the first element we remove is the one we added first. Consequently, if we store elements using a queue and examine the elements one by one, then we inspect the elements in a first-in, first-out (FIFO) order. Consequently, whenever we want to examine the elements of a dynamically changing set in the FIFO order, it is very natural to store the set as a queue.

Notice that if we keep adding elements to the rear of the queue and keep deleting elements from the front, eventually rear becomes equal to n and we cannot add any more elements to the queue. The queue at this point might not be "full" because the earlier part of the queue might be empty. We can overcome this difficulty by representing the elements of the list in the wraparound fashion, as shown in Figure A.10(a). In this representation, we perform the operations in exactly the same way as earlier except that while adding an element to the list we increment rear as (rear + 1) mod n, and while deleting an element from the list we increment front as (front + 1) mod n.

Summary

In this appendix we studied three important ways to store and manipulate sets: arrays, singly linked lists, and doubly linked lists. Figure A.11 summarizes the time complexity for performing various set operations on these data structures. Stacks and queues are special types of sets that perform operations either at the beginning or at the end of a list. To implement stacks and queues, we can use either arrays or linked lists; we have described only the array implementation.

	Array	Singly linked list	Doubly linked list
1. Inserting element at the end	$O(1)$	$O(1)$	$O(1)$
2. Inserting element at an arbitrary place	$O(n)$	$O(1)$	$O(1)$
3. Deleting element from the end	$O(1)$	$O(1)$	$O(1)$
4. Deleting element from an arbitrary place	$O(n)$	$O(n)$	$O(1)$
5. Determining kth element of the list	$O(1)$	$O(k)$	$O(k)$
6. Determining membership of an element in the set	$O(n)$	$O(n)$	$O(n)$

Figure A.11 Time complexity of various set operations for arrays and linked lists.

A.3 d-HEAPS

A *heap* (or, a *priority queue*) is a data structure for efficiently storing and manipulating a collection H of elements (or objects) when each element $i \in H$ has an associated real number, denoted by $key(i)$. We want to perform the following operations on the elements in the heap H:

create(H). Create an empty heap H.

insert(i, H). Insert an element i in the heap.

find-min(i, H). Find an element i with the minimum key in the heap.

delete-min(i, H). Delete the element i with the minimum key from the heap.

delete(i, H). Delete an arbitrary element i from the heap.

decrease-key(i, value, H). Decrease the key of element i to a smaller value, denoted by *value*.

increase-key(i, value, H). Increase the key of element i to a larger value, denoted by *value*.

In this section we discuss the d-heap and binary heap data structures (the binary heap is a well-known special case of the d-heap with $d = 2$). In the next section we describe a more efficient (and also more complex) heap known as the *Fibonacci heap*.

In most applications of heaps to network flow algorithms, the elements are nodes and their keys are node labels. Therefore, in our discussion of heaps, we shall use the word "element" and "node" interchangeably. Moreover, to be consistent with the conventions we have adopted for networks, we shall assume that the heap stores a maximum of n nodes.

Heaps find a variety of applications in network flow algorithms. Two such applications are Dijkstra's algorithm for the shortest path problem discussed in Section 4.7, and Prim's algorithm for the minimum spanning tree problem described in Section 13.5. Another important application of heaps is the sorting of n numbers in a nondecreasing order. We can sort n numbers using a heap as follows. First, we create an empty heap. Then, one by one, we add n numbers to the heap by performing n insert operations, letting the key for the ith entry be one of the numbers we wish to sort. Next, we repeat the following step iteratively: Select an element i with the minimum key using the operation find-min and then delete it from the heap using the operation delete-min. We terminate this procedure when the heap is empty. It is easy to see that we delete the elements from the heap in a nondecreasing order of their values.

Definition and Properties of a d*-Heap*

In a d-heap, we store the nodes of the heap as a rooted tree whose arcs represent a predecessor–successor (or parent–child) relationship. We store the rooted tree using predecessor indices and sets of successors, as follows:

pred(i): the predecessor (or the parent) of node i in the d-heap. The root node has no predecessor, so we set its predecessor equal to zero.

SUCC(i): the set of successors (or children) of node i in the d-heap.

In the d-heap we define the *depth* of a node i as the number of arcs in the unique path from node i to the root. For example, in the d-heap shown in Figure A.12, node 5 has a depth of 0 and nodes 9, 8, and 15 have a depth of 1.

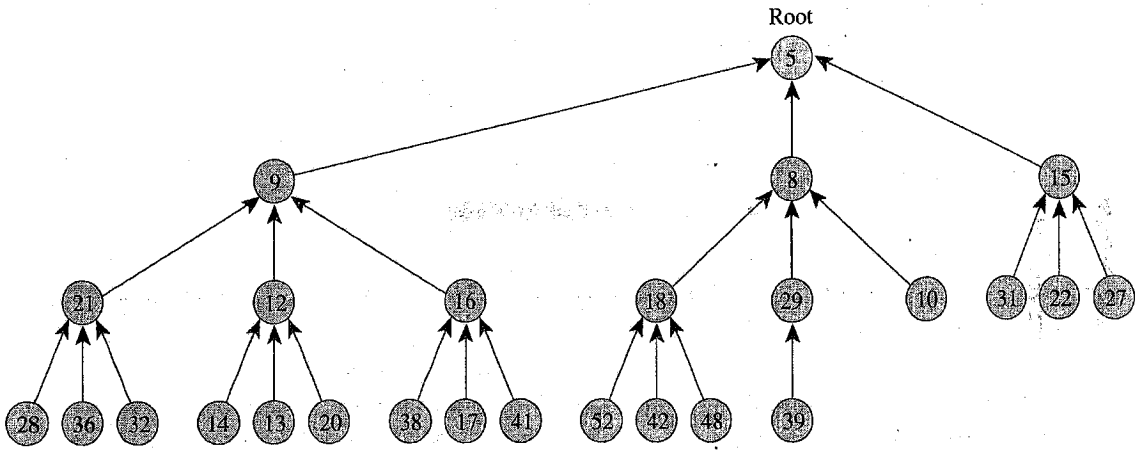


Figure A.12 Example of a d-heap for $d = 3$.

Each node in the d-heap has at most d successors, which we assume to be ordered from left to right. We refer to the successors of a node as *siblings* (of each other). The d-heap always satisfies the following property that we maintain inductively. We add nodes to the heap in an increasing order of depth values, and for the same depth value we add nodes from left to right. We refer to this property as the *contiguity property*. Figure A.12 gives an example of the d-heap for $d = 3$. In this example, we assume for convenience that key(i) = i for each $i = 1$ to n (in this case $n = 50$). In this particular example, we have stored only a subset of the nodes in the heap. Note that nodes 12, 29, and 15 have the predecessors 9, 8, and 5, respectively.

The contiguity property implies the following results:

Property A.1

(*a*) *At most* d^k *nodes have depth* k.
(*b*) *At most* $(d^{k+1} - 1)/(d - 1)$ *nodes have depth between* 0 *and* k.
(*c*) *The depth of a d-heap containing* n *nodes is at most* $\lfloor \log_d n \rfloor$.

We leave the proof of this property as an exercise to the reader.

Storing a d*-Heap*

The structure of a d-heap permits us to store it as an array and manipulate it quite efficiently. We order the nodes in the increasing values of their depths, and we order the nodes with the same depth from left to right. We then store the nodes, in order, in an array DHEAP. For example, if we apply this method to the d-heap shown in Figure A.12, then DHEAP = {5, 9, 8, 15, 21, 12, 16, 18, 29, 10, 31, 22, 27, 28, 36, 32, 14, 13, 20, 38, 17, 41, 52, 42, 48, 39}. We also maintain an array position that contains the position of each node. For this example, position(9) = 2 and position(15) = 4. We maintain an additional parameter *last* that specifies the number of nodes stored in the array DHEAP. For this example, last = 26. This storage

scheme has one rather nice property that permits us to easily access the predecessors and successors of any node:

Property A.2

(*a*) *The predecessor of the node in position i is contained in position* $\lceil (i - 1)/d \rceil$.

(*b*) *The successors of the node in position i are contained in positions* $id - d + 2, \ldots, id + 1$.

We leave the proof of this property as an exercise to the reader; it is instructive to verify this result on our numerical example. For example, node 18 is in position 8, so its predecessor is in position $\lceil (8 - 1)/3 \rceil = 3$ and its successors are in positions $3(8) - 3 + 2 = 23$ to $3(8) + 1 = 25$. This property implies that if we maintain the array DHEAP, we need not explicitly maintain the predecessor index and the set of successor indices of a node. We can compute these when required during the course of an algorithm. For the sake of exposition, we continue using predecessors and successors, but ignore the time required to update these data structures whenever the *d*-heap changes.

Heap Order Property

A heap always satisfies the following invariant, which we subsequently refer to as the *heap order property*.

Property A.3 (Invariant 1). *The key of node i in the heap is less than or equal to the key of each of its successors. That is, for each node i,* $key(i) \leq key(j)$ *for every* $j \in SUCC(i)$.

We note that we might violate Invariant 1 while performing a heap operation but will always satisfy it at the end of any heap operation. The reader can verify that the example shown in Figure A.12 satisfies the heap order property if for every node in the heap we assume that $key(i) = i$.

The following result is an immediate consequence of the heap order property.

Property A.4. *The root node of the d-heap has the smallest key.*

Swapping

In the *d*-heap data structure, we reduce each heap operation into a sequence of a fundamental operation, each called *swap*(i, j). The operation swap(i, j) swaps (or interchanges) nodes i and j. Figure A.13 gives an example of a swap. In terms of the array used to store a *d*-heap, as a result of applying swap(i, j), we store node i at the position where node j was stored, and store node j at the position where node i was stored. For example, if we perform swap(4,6) in the DHEAP = {5, 6, 7, 4, 8, 11, 12, 9}, as shown in Figure A.13, then the new array representation of the *d*-heap becomes DHEAP = {5, 4, 7, 6, 8, 11, 12, 9}. Clearly, the swap operation requires $O(1)$ time.

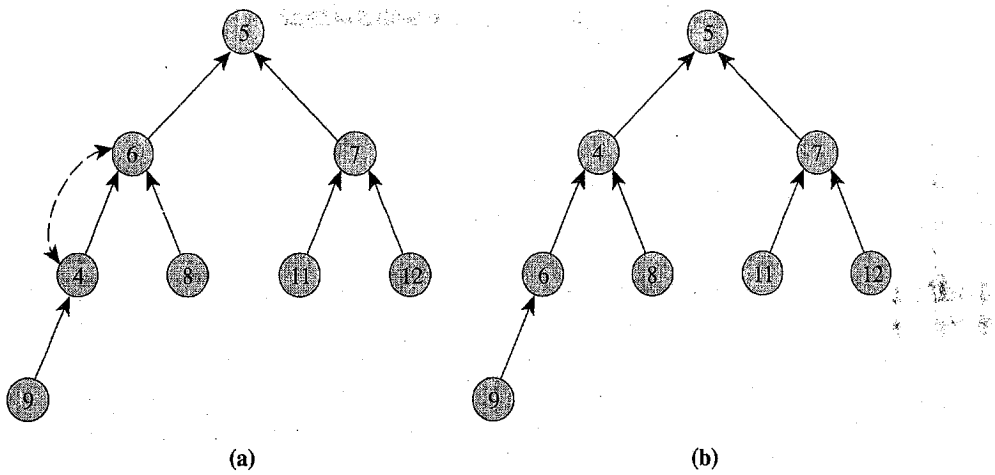


Figure A.13 Example of swap (4, 6): (a) heap before the swap; (b) heap after the swap.

Restoring the Heap Order Property

In the course of applying an algorithm, we will frequently change the value of some key and so temporarily violate the heap order property. How can we restore this property? Suppose that we decrease the key of some node i. Let j = pred(i). If after the change in the value of key(i), key (j) $\leq$ key(i), the heap still satisfies the heap order property and we are done. However, if key(j) > key(i), we need to restore the heap order property. The procedure *siftup*(i) described in Figure A.14 accomplishes this task.

```
procedure siftup(i);
begin
    while i is not a root node and key(i) < key(pred(i)) do swap(i, pred(i));
end;
```

Figure A.14 Procedure siftup(i).

Inductive arguments show that at the termination of the siftup procedure, the heap satisfies the heap order property. The procedure siftup requires $O(\log_d n)$ time because each execution of the while loop decreases the depth of node i by one unit and, by Property A.1, its original depth is $O(\log_d n)$.

Suppose next that we increase the key of some node i. If after the change in the value of key(i), key(i) $\leq$ key(j) for all $j \in$ SUCC(i), the heap still satisfies the heap order property and we are done; otherwise, we need to restore the heap order property. The procedure *siftdown*(i) described in Figure A.15 accomplishes this task. In the description we let *minchild*(i) denote the node with smallest key in SUCC(i).

```
procedure siftdown(i);
begin
    while i is not a leaf node and key(i) > key(minchild(i)) do swap(i, minchild(i));
end;
```

Figure A.15 Procedure siftdown(i).

An inductive argument will again show that at the termination of the siftdown procedure, the heap satisfies the heap order property. The procedure requires $O(d \log_d n)$ time because each execution of the while loop increases the depth of node i by one unit and each execution requires $O(d)$ time to compute minchild(i).

Performing Heap Operations

We are now in a position to describe how we can perform various operations in the d-heap.

find-min(i, H). The root node of the heap is the node with the minimum key and it is located at the first position of the array DHEAP. Therefore, this operation requires $O(1)$ time.

insert(i, H). We increment last by one and store the new node i at the last position of the array DHEAP. Then we execute the procedure siftup(i) to restore the heap order property. Clearly, this operation requires $O(\log_d n)$ time.

decrease-key(i, value, H). We decrease the key of node i and execute the procedure siftup(i) to restore the heap order property. This operation requires $O(\log_d n)$ time.

delete-min(i, H). Clearly, node i is the root node of the heap. Let node j be the node stored at the last position of the array DHEAP. We first perform swap(i, j) and then decrease last by 1. Next, we perform siftdown(j) to restore the heap order property. Clearly, this heap operation requires $O(d \log_d n)$ time.

We ask the reader to show as an exercise how to perform the remaining two heap operations, delete(i, H) and increase-key(i, value, H), in $O(d \log_d n)$ time. We summarize our discussion as follows:

Theorem A.5. *The d-heap data structure requires $O(1)$ time to perform the operation find-min, $O(\log_d n)$ time to perform the operations insert and decrease-key, and $O(d \log_d n)$ time to perform the operations delete-min, delete, and increase-key.* ◆

Recall that a binary heap is a d-heap with $d = 2$. For binary heaps, this theorem assumes the following special form.

Theorem A.6. *The binary heap data structure requires $O(1)$ time to perform the operation find-min, and $O(\log n)$ time to perform each of the operations insert, delete, delete-min, decrease-key, and increase-key.* ◆

As an example of applying heaps, consider a sorting algorithm. Recall from our prior discussion that while sorting n numbers, we perform n inserts, n find-mins, and n delete-mins. Consequently, the running time of the sorting algorithm using d-heaps is $O(nd \log_d n)$, which is $O(n \log n)$ for any fixed value of d.

A.4 FIBONACCI HEAPS

The Fibonacci heap is a novel data structure that allows the heap operations to be performed more efficiently than d-heaps. This data structure performs the operations insert, find-min, and decrease-key in $O(1)$ amortized time and the operations delete-min, delete, and increase-key in $O(\log n)$ amortized time. Recall from Section 3.2 that the amortized complexity of an operation is the *average* worst-case complexity of performing that operation. In other words, the amortized complexity of an operation is $O(g(n))$ if for a sequence of k (sufficiently large) operations, the total time required by these operations is $O(kg(n))$. For our purpose, $k \geq n$ is sufficiently large.

Properties of Fibonacci Numbers

Researchers have given the Fibonacci heap data structure its name because the proof of its time bounds uses properties of the well-known Fibonacci numbers. Before discussing the data structure, we first discuss these properties. The Fibonacci numbers are defined recursively as $F(1) = 1$, $F(2) = 1$, and $F(k) = F(k-1) + F(k-2)$, for all $k \geq 3$. These numbers satisfy the following properties:

Property A.7

(a) For $k \geq 3$, $F(k) \geq 2^{(k-1)/2}$.

(b) $F(k) = 1 + F(1) + F(2) + \cdots + F(k-2)$.

Proof. The facts that $F(k) = F(k-1) + F(k-2)$ and $F(k-1) \geq F(k-2)$ imply that $F(k) \geq 2F(k-2)$. Consequently, if k is odd, $F(k) \geq 2F(k-2) \geq 2^2 F(k-4) \geq 2^3 F(k-6) \geq 2^{(k-1)/2} F(1) = 2^{(k-1)/2}$. If k is even, we argue by induction. The claim is true if $k = 4$. Suppose it is true for even numbers less than k. The $F(k) \geq F(k-1) + F(k-2) \geq 2^{(k-2)/2} + 2^{(k-3)/2}$ by the result for k odd and the induction hypothesis. But then $F(k) \geq 2^{(k-3)/2}[2^{1/2} + 1] \geq 2^{(k-1)/2}$ and so by induction the conclusion is true for all $k \geq 3$.

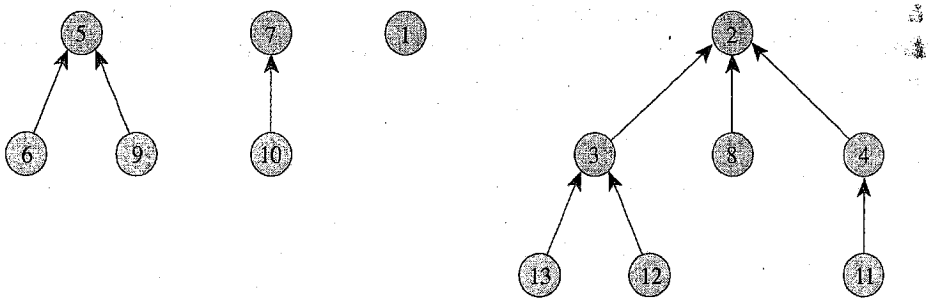
To prove part (b), let us define a series of numbers $G'(\cdot)$ as $G'(1) = 1$, $G'(2) = 1$, and $G'(k) = 1 + G'(1) + G'(2) + \cdots + G'(k-2)$ for all $k \geq 3$. Then $G'(k-1) = 1 + G'(1) + G'(2) + \cdots + G'(k-3)$ and $G'(k) - G'(k-1) = G'(k-2)$. Alternatively, $G'(k) = G'(k-1) + G'(k-2)$, which is the same manner in which Fibonacci numbers are defined. Therefore, $G'(k) = F(k)$ for all k. ◆

Property A.8. *Suppose that a series of numbers $G(\cdot)$ satisfies the properties that $G(1) = 1$, $G(2) = 1$, and $G(k) \geq 1 + G(1) + G(2) + \cdots + G(k-2)$ for all $k \geq 3$. Then $G(k) \geq F(k)$.*

Proof. We prove inductively that $G(k) \geq F(k)$ for all k. This claim certainly is true for $k = 1$ and $k = 2$. Let us assume that it is true for all values of k from 1 through $q - 1$. Then $G(q) \geq 1 + G(1) + G(2) + \cdots + G(q-2) \geq 1 + F(1) + F(2) + \cdots + F(q-2) = F(q)$, the equality following from Property A.7(b). ◆

Defining and Storing a Fibonacci Heap

As we noted earlier, a heap stores a set of elements, each with a real-valued key. A Fibonacci heap is a collection of directed rooted in-trees: each node i in the tree represents an element i and each arc (i, j) represents a predecessor–successor (parent–child) relationship: node j is the predecessor (parent) of node i. Figure A.16 gives an example of a Fibonacci heap.



(a)

i	1	2	3	4	5	6	7	8	9	10	11	12	13
pred(i)	0	0	2	2	0	5	0	2	5	7	4	3	3
SUCC(i)	$\varnothing$	{3, 8, 4}	{13, 12}	{11}	{6, 9}	$\varnothing$	{10}	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$
rank(i)	0	3	2	1	2	0	1	0	0	0	0	0	0

(b)

Figure A.16 Fibonacci heap: (a) rooted trees; (b) corresponding data structure.

To represent a Fibonacci heap numerically (i.e., in a computer) and to manipulate it effectively, we need the following data structure:

pred(i): the predecessor (or the parent) of node i in the Fibonacci heap.

We refer to a node with no parent as a root node and we set its predecessor to zero. This convention permits us to determine whether a node is a root node or a nonroot node by looking at the node's predecessors index. We also need the following data structures:

SUCC(i): the set of successors (or children) of node i. We maintain this set as a doubly linked list.

rank(i): the number of successors of node i [i.e., rank(i) = | SUCC(i) |].

minkey: the node with the minimum key.

Figure A.16(b) shows this data structure for the rooted trees given in Figure A.16(a). We need additional data structures to support various heap operations; we will introduce these data structures later, when we require them.

We need one additional piece of notation. A subtree *hanging* at any node i of any rooted tree contains the node i, its successors, successors of its successors, and

so on. For example, in Figure A.16, the subtree hanging at node 5 contains the nodes 5, 6, and 9, and the subtree hanging at node 7 contains the nodes 7 and 10.

Linking and Cutting

In using the Fibonacci heap data structure, we reduce each heap operation into a sequence of two fundamental operations: *link*(i,j) and *cut*(i). We apply the operation link(i, j) to two (distinct) root nodes i and j of equal rank; it merges the two trees rooted at these nodes into a single tree. The operation cut(i) cuts node i from its predecessor and makes i a root node.

link(i, j). If key$(j) \leq$ key(i), then add arc (i, j) to the Fibonacci heap (thus making node i the predecessor of node j). If key$(j) >$ key(i), then add arc (j, i) to the heap.

cut(i). Delete arc $(i,$ pred$(i))$ from the heap (thus making node i a root node).

We illustrate these two operations on the examples shown in Figure A.17. For simplicity, we assume that for every node i, key$(i) = i$. Notice that the link operation increases the rank of node i or of node j by 1. Moreover, each of these operations

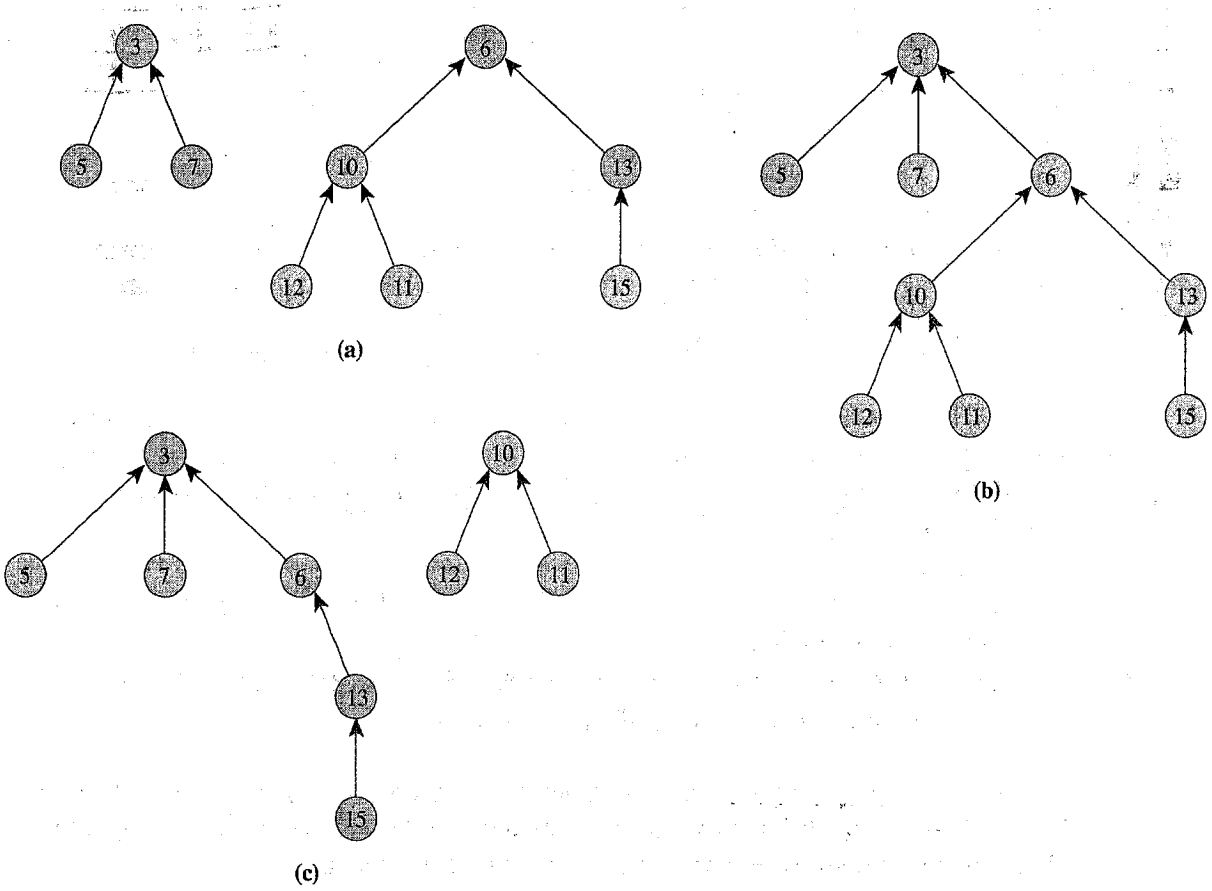


Figure A.17 Illustrating link and cut operations: (a) original heap; (b) heap after the operation link (3, 6); (c) heap after the operation cut(10).

changes the pred and SUCC and rank information for at most two nodes; consequently, we can perform them in $O(1)$ time. Later in this section we describe the additional data structures that we maintain to manipulate the Fibonacci heap effectively; we can also modify them in constant time as we perform a link and a cut operation. We record this result formally for ease of future reference.

Property A.9. *The operations link(i, j) and cut(i) require $O(1)$ time to execute.*

While manipulating the Fibonacci heap data structure, we perform a sequence of links and cuts. There is a close relationship between the number of links and cuts. To observe this relationship, consider a potential function Φ defined as the number of rooted trees. Each link operation decreases Φ by 1 and each cut operation increases Φ by 1. The total decrease in Φ is bounded by its initial value (which is n) plus the total increase in Φ. The following result is now evident.

Property A.10. *The number of links is at most n plus the number of cuts.*

Invariants in Fibonacci Heaps

The Fibonacci heap data structure maintains a set of rooted trees that change dynamically as we perform various linking and cutting operations. These rooted trees satisfy certain invariants that are essential for deriving the claimed time bounds for the heap operations. The nodes of the Fibonacci heap always satisfy the heap order property (i.e., Invariant 1), which states that the key of a node is less than or equal to the keys of its successors. The Fibonacci heap also satisfies the following two invariants:

Property A.11 (Invariant 2). *Each nonroot node has lost at most one successor after becoming a nonroot node.*

Property A.12 (Invariant 3). *No two root nodes have the same rank.*

As before, although we might violate these invariants at intermediate steps of some heap operations, the heap will satisfy them at the conclusion of each heap operation. One important consequence of Invariants 2 and 3 is that the maximum possible rank of any node is $2 \log n + 1$. We establish this result next.

Lemma A.13. *Any node in the Fibonacci heap has rank at most* $2 \log n + 1$.

Proof. Let $G(k)$ denote the minimum number of nodes contained in a subtree hanging at a node of rank k in a Fibonacci heap. We shall prove that $G(k) \geq F(k)$. Since no subtree can contain more than n nodes, Properties A.7 and A.8 imply that $n \geq G(k) \geq F(k) \geq 2^{(k-1)/2}$, which implies that $k \leq 2 \log n + 1$.

Let w be a node in a Fibonacci heap with rank k. Arrange the successors of node w in the same order in which the previous operations linked them to w, from the earliest to the latest. We claim that the rank of the ith successor of w is at least $i - 2$. To establish this result, let y be the ith successor of node w and consider the moment when y was linked to w. Just before this link operation, w had at least

$i - 1$ successors. (It might have had more than $i - 1$ successors at that time, some having been cut since then.) Since at the time of this link operation, nodes y and w both have the same rank, node y had at least $i - 1$ successors just before we performed this link operation. Furthermore, notice that since that time node y has lost at most one successor (from Invariant 2). Therefore, node y (which is the ith successor of node w) has rank at least $i - 2$. As a result, the subtree hanging at node w contained at least $1 + G(1) + G(2) + \cdots + G(k - 2)$ nodes. To summarize, we have shown that $G(k) \geq 1 + G(1) + G(2) + \cdots + G(k - 2)$, which in view of Property A.8 implies that $G(k) \geq F(k)$. From our prior observation, this conclusion establishes the lemma. ◆

The following property follows directly from Invariant 3 and Lemma A.13.

Property A.14. *A Fibonacci heap contains at most* $1 + 2 \log n$ *rooted trees.*

We next discuss how we restore Invariants 2 and 3 if they become violated at intermediate steps of a heap operation.

Restoring Invariant 2

To restore Invariant 2, we maintain an additional index $lost(i)$ for every node i, defined as follows.

lost(i): For a nonroot i, lost(i) represents the number of successors the node has lost after it became a nonroot node. For a root node i, lost(i) = 0.

Suppose that while manipulating a Fibonacci heap, we perform the operation cut(i). We refer to this cut as the *actual cut*. Let j = pred(i). In this operation, node j loses a successor. If node j is a nonroot node, we increment lost(j) by 1. If lost(j) becomes two, Invariant 2 requires that we make node j a root node. In that case we perform cut(j) and make j a root node. Let k = pred(j). This cut increases lost(k) by 1. If k is a nonroot node and lost(k) = 2, we must make it a root node as well, and so on. Thus an actual cut might lead to several cuts due to a *cascading* effect: We keep performing these cuts until we reach a node that has not lost any successor so far or is a root node. We refer to these additional cuts that are triggered by an actual cut as *cascading cuts*, and the entire sequence of steps following an actual cut as *multicascading*.

We illustrate this process on the Fibonacci heap shown in Figure A.18(a). In this figure we represent nonroot nodes with lost(i) = 1 by shaded circles. Suppose that we cut node 17 from its predecessor. This operation also requires that we also cut nodes 11 and 8 from their predecessors. Figure A.18(b) shows the resulting Fibonacci heap that satisfies Invariant 2. We now summarize the preceding discussion.

Property A.15. *If we perform an actual cut, we might also need to perform several cascading cuts so that the heap again satisfies Invariant 2; the time needed for these operations is proportional to the total number of cuts performed.*

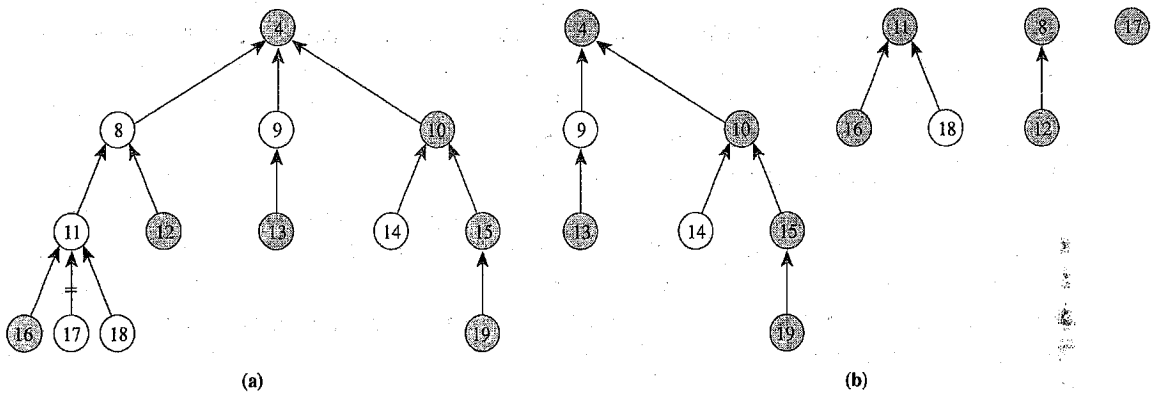


Figure A.18 Illustrating how we satisfy invariant 2. (Unshaded nodes have already lost a child.)

Suppose that we perform a number of actual cuts at different times while manipulating a Fibonacci heap and that these cuts cause additional cascading cuts. What is the relationship between the total number of actual cuts and the total number of cascading cuts? We shall show that the total number of cascading cuts cannot exceed the total number of actual cuts. To prove this result, consider the potential function $\Phi = \sum_{i \text{ in heap}} \text{lost}(i)$. Suppose that we perform cut(i) and $j = \text{pred}(i)$. This operation sets lost(i) to zero and increases lost(j) by one if j is a nonroot node. If the cut is an actual cut, lost(i) equals 0 or 1 before the cut, and if it is a cascading cut, lost(i) equals 2 before the cut. Therefore, an actual cut increases lost(i) + lost(j), and hence the value of the potential function Φ by at most one, and a cascading cut decreases lost(i) + lost(j) by at least one. If we start with a potential value of zero, the total decreases in the potential function are bounded by the total increases. The following property is now apparent.

Property A.16. *The total number of cascading cuts is less than or equal to the total number of actual cuts.*

Restoring Invariant 3

The Invariant 3 requires that no two root nodes have the same rank. To maintain this property, we need the following index for every possible rank $k = 1, \ldots, K = 2 \log n + 1$.

bucket(k). If the Fibonacci heap contains no root node with rank equal to k, then bucket(k) = 0; and if some root node i has a rank equal to k, then bucket(k) = i.

Suppose that while manipulating a Fibonacci heap, we create a root node j of rank k and the heap already contains another root node i with the same rank. Then we repeat the following procedure to restore Invariant 3. We perform the operation link(i, j), which merges the two rooted trees into a new tree of rank $k + 1$. Suppose that node l is the root of the new tree. Then by looking at bucket($k + 1$), we check to see whether the heap already contains a root node of rank $k + 1$. If not, we are

done. Otherwise, we perform another link operation to create another rooted tree of rank $k + 2$ and check whether the heap already contains a root node of rank $k + 2$. We repeat this process until we satisfy Invariant 3. We refer to this sequence of steps following the addition of a new root as *multilinking*.

We illustrate this process of re-establishing Invariant 3 on a numerical example. Consider the Fibonacci heap shown in Figure A.19(a), assuming that the key of node i equals i. Suppose that we add a new rooted tree containing a singleton node 10. The heap already contains another root node of rank 0, namely node 9. Thus we perform a link operation on nodes 9 and 10, obtaining the rooted trees shown in Figure A.19(b). Now two trees in the heap contains, with roots 7 and 9, have rank 1. We perform another link operation, producing the structure shown in Figure A.19(c). But now two trees, with roots 1 and 7, have rank 2. We perform another link operation, producing the structure shown in Figure A.19(d). At this point, the rooted trees satisfy Invariant 3 and we terminate.

We summarize the preceding discussion in the form of a property.

Property A.17. *If we add a new rooted tree to the Fibonacci heap, we might need to perform several links to restore Invariant* 3; *the time needed for these operations is proportional to the total number of links.*

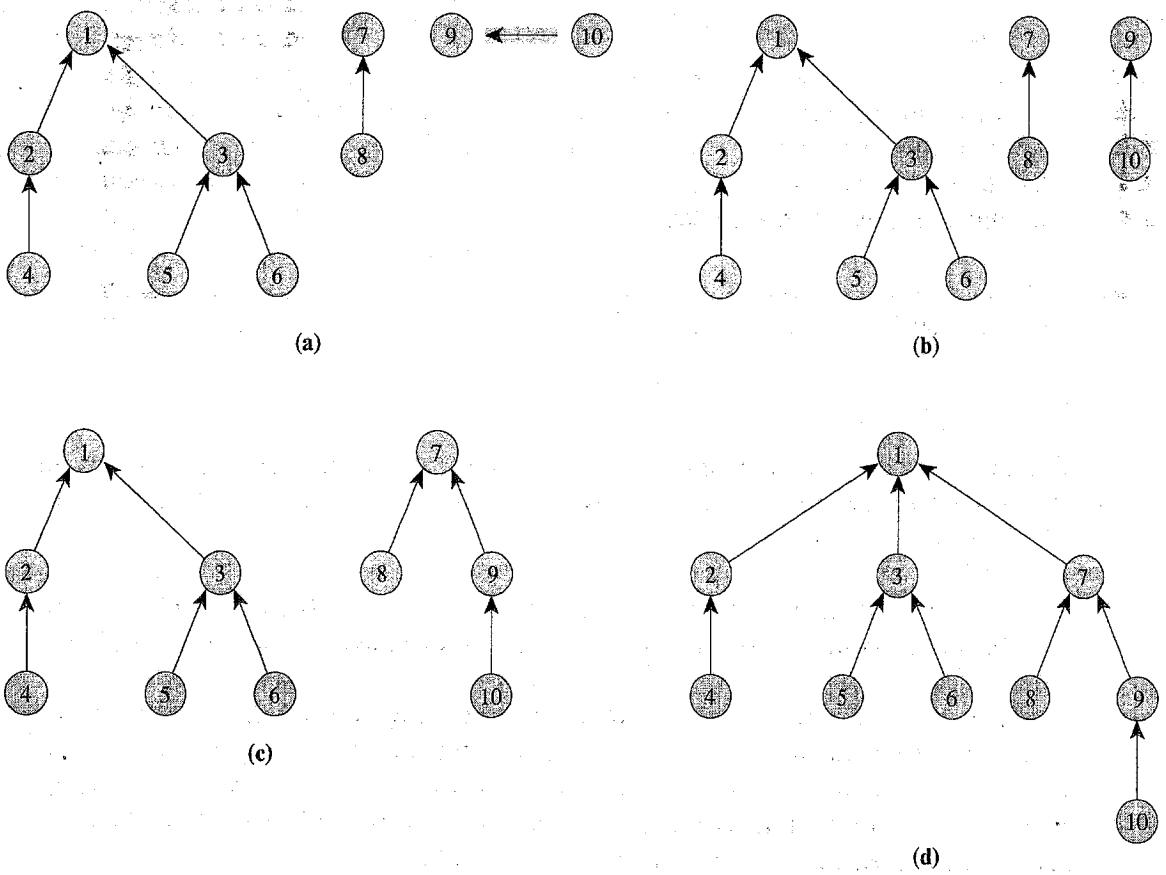


Figure A.19 Illustrating how we satisfy invariant 3.

Heap Operations

Finally, we show how we perform various heap operations using the Fibonacci heap data structure and indicate the amount of time they take.

find-min(i, H). We simply return i = minkey, since the variable minkey contains the node with the minimum key.

insert(i, H). We create a new singleton root node i and add it to H. After we have performed this operation, the heap might violate Invariant 3, in which case we perform multilinking to restore the invariant.

decrease-key(i, value, H). We first decrease the key of node i and set it equal to value. After we have decreased the key of node i, every node in the subtree hanging at node i still satisfies the heap order property; the predecessor of node i might, however, violate this property. Let j = pred(i). If key(j) $\leq$ value, we are done. Otherwise, we perform an actual cut, cut(i), make node i a root node, and update minkey. After we have performed the cut, the heap might violate Invariant 2, so we perform multicascading to restore this invariant. The resulting cascading cuts generate new rooted trees whose roots we store in a list, LIST. Then one by one, we remove a root node from LIST, add it to the previous set of roots, and perform multilinking to satisfy Invariant 3. We terminate when LIST becomes empty.

delete-min(i, H). We first set i = minkey. Then one by one, we scan each node $l \in$ SUCC(i), perform an actual cut, cut(l), and update minkey. We apply multilinking after performing each such actual cut. When we have cut each node in SUCC(i), we scan through all root nodes [which are stored in bucket(k), for $k = 0, 1, \ldots, 2 \log n + 1$], identify the root node h with minimum key, and set minkey = h. Recall that $|\text{SUCC}(i)| \leq 2 \log n + 1$, because Lemma A.13 implies that each node has at most $2 \log n + 1$ successors. Therefore, the delete-min operation performs $O(\log n)$ actual cuts, followed by a number of cascading cuts and links. Then we scan through $O(\log n)$ root nodes to identify the root with the minimum key.

Figure A.20, which lists the sequence of steps and the associated running time for each heap operation, summarizes the preceding discussion. Since we do not, as yet, know the time required for multicascading and multilinking, we write "?" for the times of these steps.

We now consider the time required for multicascading and multilinking. We claim that we can ignore the time taken by these two steps. To establish this claim, we use the following facts: (1) Property A.16, which states that the number of cascading cuts is no more than the number of actual cuts; and (2) Property A.10, which states that the number of links is no more than n plus the number of actual and cascading cuts. Consequently, if we perform a sufficiently large number of operations (i.e., more than n), the number of actual cuts will count the number of links and cascading cuts within a constant factor; therefore, we can ignore the time required for the latter operations.

We illustrate this idea further by considering Dijkstra's algorithm for the short-

Heap operation	Sequence of steps	Time taken
find-min(i, H)	(a) Return i = minkey.	$O(1)$
insert(i, H)	(a) Add a new singleton node i. (b) Perform multilinking.	$O(1)$?
decrease-key(i, *value*, H)	(a) Decrease the key of node i. (b) If node i violates Invariant 1 then (b.1) Perform cut(i) and update minkey. (b.2) Perform multicascading. (b.3) Perform multilinking.	$O(1)$ $O(1)$? ?
delete-min(i, H)	(a) For each node $l \in SUCC(i)$ do (a.1) Perform cut(l). (a.2) Perform multilinking. (b) Compute minkey by scanning all root nodes.	 $O(\log n)$? $O(\log n)$

Figure A.20 Summary of heap operations in a Fibonacci heap.

est path problem as described in Section 4.7. In that discussion we showed that Dijkstra's algorithm performs n inserts, n find-mins, n delete-mins, and at most m decrease-key operations. The time requirements of the heap operations listed in Figure A.20 imply that the algorithm requires $O(m + n \log n)$ time, plus the time for $O(n \log n)$ actual cuts, plus the time for multicascading and multilinking. Using the facts that the number of cascading cuts and links are no more than twice the number of actual cuts, and that each actual cut requires $O(1)$ time, we immediately see that the shortest path algorithm runs in $O(m + n \log n)$ time.

So, if we ignore the time for multicascading and multilinking, it is clear from Figure A.20 that the operations find-min, insert, and decrease-key require $O(1)$ amortized time, and the operation delete-min requires $O(\log n)$ amortized time. We ask the reader to prove that the operation delete and increase-key also require $O(\log n)$ amortized time. We summarize the discussion in this section as follows:

Theorem A.18. *The Fibonacci heap data structure requires $O(1)$ amortized time to perform each of the operations insert, find-min, and decrease-key, and $O(\log n)$ amortized time to perform each of the operations delete-min, delete, and increase-key.* ◆

REFERENCE NOTES

The role of data structure is critical in designing efficient algorithms and in writing computer programs for implementing algorithms. In this appendix we have presented some of the most elementary data structures and the ones that we use frequently in network flow algorithms. The following books provide much additional information on data structures: Knuth [1973a, 1973b], Aho, Hopcroft, and Ullman [1983], Mehlhorn [1984], and Cormen, Leiserson, and Rivest [1990].

Appendix B

$\mathcal{NP}$-COMPLETENESS

Seek not out the things that are too hard for thee, neither search the things that are above thy strength.
—The Apocrypha (The Hidden Books)

Chapter Outline

B.1 Introduction
B.2 Problem Reductions and Transformations
B.3 Problem Classes $\mathcal{P}$, $\mathcal{NP}$, $\mathcal{NP}$-Complete, and $\mathcal{NP}$-Hard
B.4 Proving $\mathcal{NP}$-Completeness Results
B.5 Concluding Remarks

B.1 INTRODUCTION

One of the primary purposes of scientific inquiry is to help structure the world around us so that we can better understand it. For example, in physics and chemistry, the periodic chart of the elements helps us to understand and categorize the relationship between the basic elements of the universe; in biology, the genus/species nomenclature helps us to understand the commonalties and differences among animals and plants. Computer science, mathematics, and operations research are no different; we often classify these fields in a variety of ways—for example, simply into subspecialties such as discrete and continuous mathematics—that helps us to discern their underlying structure. This idea of using classification as an organizing tool prompts the following basic question: Is there a way to develop a structural understanding of algorithms or for the problems to which we wish to apply them? Perhaps surprisingly, the research community had not proposed an approach for resolving this question until the early 1970s, when the field began in earnest to develop a topic known as computational complexity theory which attempts to categorize the computational requirements of both algorithms and important classes of problems met in practice. In this appendix, we discuss one cornerstone of this development, a topic known as $\mathcal{NP}$-completeness.

We call a class of optimization problems *easy* if we can develop an algorithm to solve every instance of the problem class in polynomial time (i.e., by an algorithm that requires a number of operations that is polynomial in the size of the input data for the problem). We also refer to a polynomial-time algorithm as an *efficient* algorithm. A majority of the network flow problems studied in this book—the shortest path problem, the maximum flow problem, the minimum cost flow problem, to name a few—are easy. Despite the best efforts of thousands of researchers across the

globe spanning several decades, the research community has been unable to show that many other network and combinatorial optimization problems (e.g., the knapsack problem and the traveling salesman problem) are easy because no one has been able to develop any efficient algorithm for solving these problems. These unsuccessful attempts have led some researchers to question whether these problems are *inherently hard* in the sense that no efficient algorithm could possibly ever solve these problems. The theory of $\mathcal{NP}$-completeness is an outgrowth of these inquiries. Although this theory has been unable to prove that these difficult problems admit no efficient algorithms, the theory has shown that the majority of these problems are equivalent to each other in the sense that if we could develop an efficient algorithm for one problem in this class, we would then be able to develop an efficient algorithm for *every* other problem in this class. We refer to this broad class of "computationally equivalent" problems as *$\mathcal{NP}$-complete problems* (later in this appendix we give a formal definition of this class of problems). This class now includes thousands of problems and possesses the remarkable property (which is somewhat difficult to believe initially) that each problem in this class can be transformed to every other problem in polynomial time; as a consequence, each problem is "just as hard" as every other problem. This relationship suggests that $\mathcal{NP}$-complete problems share some generic difficulty that is beyond the reach of polynomial-time algorithms. Indeed, the research community widely believes that $\mathcal{NP}$-complete problems *cannot* be solved efficiently. This is the bad news about difficult problems.

The theory of $\mathcal{NP}$-completeness also has its positive aspects. To capture its usefulness, consider the following story. Suppose that your boss asks you to develop an algorithm for a complex design problem. Despite weeks of sincere efforts you do not succeed in developing an efficient algorithm for solving this problem. Every algorithm that you are able to construct is substantially no better than searching through all possible designs: There are so many of them that this enumeration would require several years of computer time on the fastest computers owned by your company. Surely, you are intelligent enough not to return to your boss's office and report, *"I can't find an efficient algorithm, I guess I am just too dumb."*

Although you did not succeed in developing an efficient algorithm, you were convinced that the design problem is *inherently difficult* and that no one, no matter how smart and creative, could possibly develop an efficient algorithm for this design problem. However, you cannot prove your conjecture, because proving it could be as difficult as finding an efficient algorithm. Therefore, you cannot walk into your boss's office and declare, *"I can't find an efficient algorithm because no such algorithm is possible!"*

The theory of $\mathcal{NP}$-completeness provides many techniques for proving that a given problem is just as hard as a large number of other problems that have defied solution by an efficient algorithm despite decades of efforts of the brightest researchers. Using these techniques, you might be able to show that your design problem is $\mathcal{NP}$-complete. Then you can confidently march into your boss's office and announce, *"I can't find an efficient algorithm, but neither can these famous people."* This statement might be sufficient to save your job.

As illustrated by this story, the theory of $\mathcal{NP}$-completeness has the following utility in practice. Whenever we encounter a new problem of some practical or theoretical interest, we try to develop an efficient algorithm for solving it. If we do

succeed, clearly the problem is easy and we and others might make further attempts to develop an even more efficient algorithm. However, if we do not succeed in developing an efficient algorithm for the problem, we might begin to wonder whether our problem is an $\mathcal{NP}$-complete problem. The theory of $\mathcal{NP}$-completeness provides us with several tools for establishing that a problem is $\mathcal{NP}$-complete. If we do succeed in showing the problem is $\mathcal{NP}$-complete, we have sufficient reason to believe that the problem is hard and no efficient algorithm can ever be developed to solve it. We should thus abandon our quest for an efficient algorithm and direct our efforts at developing efficient heuristics (i.e., algorithms that give solutions that are not guaranteed to be optimal), or at developing various types of enumeration algorithms or other algorithms that will generally run in exponential time. If we cannot prove that our problem is $\mathcal{NP}$-complete, the status of the problem is inconclusive and remains so until someone settles it either way. Indeed, many interesting problems live in this never-never land: for example, the important problem of recognizing whether two graphs are isomorphic.

This appendix provides the basic tools for carrying out this program and is organized as follows. In Section B.2 we describe problem reductions and transformations, an important construct in the theory of $\mathcal{NP}$-completeness. In Section B.3 we describe several problem classes, such as $\mathcal{P}$, $\mathcal{NP}$, and $\mathcal{NP}$-complete. In Section B.4 we study a method for showing that a problem is $\mathcal{NP}$-complete and illustrate this approach on a variety of simple optimization problems. Needless to say, our discussion of the theory of $\mathcal{NP}$-completeness is intended to be very elementary. For a deeper study of this topic, we refer the reader to the literature discussed in the reference notes.

B.2 PROBLEM REDUCTIONS AND TRANSFORMATIONS

The theory of $\mathcal{NP}$-completeness helps us to classify a given problem into two broad classes: (1) easy problems that can be solved by polynomial-time algorithms, and (2) hard problems that are not likely to be solved in polynomial time and for which all known algorithms require exponential running time. Notice that in this classification we want to determine only whether a problem can or cannot be solved in polynomial time; the order of the polynomial is irrelevant. We should keep this point in mind throughout our subsequent discussion.

In almost all the problems studied in this book, we have been concerned with determining some type of an optimal solution. The theory of $\mathcal{NP}$-completeness requires that problems be stated so that we can answer them with a yes or no; we refer to this yes–no version of a problem as a *recognition version* of it. We illustrate this notion using the traveling salesman problem (TSP).

TSP-optimization. Given a directed graph $G = (N, A)$ and an integer arc length c_{ij} associated with every arc $(i, j) \in A$, determine a tour W (i.e., a directed cycle that visits each node in the network exactly once) with the smallest possible value of the tour length $\sum_{(i,j)\in W} c_{ij}$.

TSP-recognition-I. Given a directed graph $G = (N, A)$, an integer arc length c_{ij} associated with every arc $(i, j) \in A$, and an integer k^*, does the network contain a tour W satisfying the condition $\sum_{(i,j)\in W} c_{ij} \leq k^*$?

It is easy to see that if we have a polynomial-time algorithm for TSP-optimization, we can use it to solve TSP-recognition-I. To do so, we use an algorithm for the TSP-optimization to determine an optimal tour W^*, and then we check to see whether $\sum_{(i,j)\in W^*} c_{ij}$ is less than or equal to k^*. The answer to this question is an answer of TSP-recognition-I. Interestingly, the converse is also true: If we have a polynomial-time algorithm for TSP-recognition-I, we can use it, although applied several times, to solve TSP-optimization in polynomial time. First, we perform binary search (see Section 3.3 for details of binary search methods) on the possible tour lengths and solve TSP-recognition-I at each search point to identify the optimal tour length, say k^*, of TSP-optimization. If C denotes the largest arc length in the network, then using the binary search, we require $O(\log(nC))$ executions of TSP-recognition-I to identify the optimal tour length k^*. Next we determine the optimal tour, again by executing the TSP-recognition-I algorithm repeatedly. We consider every arc $(i, j) \in A$, and one by one, apply the TSP-recognition-I algorithm to find whether the network $G(N, A - \{(i, j)\})$ contains a tour with length less than or equal to k^*. If the answer is yes, we delete the arc. After we have considered all the arcs, the remaining graph is an optimal tour of TSP-optimization.

The preceding discussion showed that we could solve TSP-optimization in polynomial time if we could solve TSP-recognition-I in polynomial time. Thus the optimization and recognition versions are equivalent in terms of whether or not they can be solved in polynomial time. Alternatively, we say that these two problems are *polynomially equivalent*. We point out that recognition problems are not always polynomially equivalent to the corresponding optimization problems. To illustrate this point, consider the following two alternative recognition problems for the traveling salesman problem:

TSP-recognition-II. Given a directed graph $G = (N, A)$, an integer k^*, and an integer arc length c_{ij} associated with every arc $(i, j) \in A$, does the network contain a tour W for which $\sum_{(i,j)\in W} c_{ij} \geq k^*$?

TSP-recognition-III. Given a directed graph $G = (N, A)$, an integer k^*, and an integer arc length c_{ij} associated with every arc $(i, j) \in A$, does every tour W in G satisfy the condition $\sum_{(i,j)\in W} c_{ij} \geq k^*$?

Note that TSP-recognition-III has a no instance if and only TSP-recognition-I has a yes instance, so we can use TSP-recognition-III, like TSP-recognition-I, to solve TSP-optimization. Can we solve the TSP-optimization by solving a polynomial number of instances of TSP-recognition-II? Perhaps. But the problem has no obvious solution. In general, each optimization problem has several associated recognition problems and we want to select a recognition problem that is polynomially equivalent to the optimization problem. For all optimization problems considered in this book, and for most problems that ever arise, some recognition version is polynomially equivalent to the optimization version. For this reason the theory of $\mathcal{NP}$-completeness, even though it applies formally only to recognition problems, is also suitable for assessing the complexity of optimization problems.

The preceding discussion also illustrates an important technique known as *problem reduction*. We say that a problem P_1 reduces to problem P_2 if we can solve problem P_1 using an algorithm for P_2 as a subroutine. We have shown in the preceding

discussion that the TSP-optimization reduces to TSP-recognition. We say that the problem P_1 *polynomially reduces* to problem P_2 if some polynomial-time algorithm that solves P_1 uses the algorithm for solving P_2 at *unit cost*. A point of central importance in this definition is the unit cost clause, which implies that the algorithm for P_2 requires unit time to execute. Naturally, in almost all cases, this assumption will be very unrealistic. Its usefulness, however, is apparent because of the following property, whose proof is left to the reader.

Property B.1. *If a problem P_1 polynomially reduces to problem P_2 and some polynomial-time algorithm solves P_2, some polynomial-time algorithm solves P_1.*

We refer to an instance of the recognition problem as a *yes instance* if the answer to this problem instance is yes, and a *no instance* otherwise. A special type of problem reduction is of significant interest, which we call *problem transformation*. We say that a problem P_1 polynomially transforms to another problem P_2 if for every instance I_1 of problem P_1 we can construct in polynomial time (e.g., polynomial in terms of the size of I_1) an instance I_2 of problem P_2 so that I_1 is a yes instance of P_1 if and only if I_2 is a yes instance of P_2. In the subsequent discussion, we consider several examples of polynomial-time transformations.

Polynomial reductions and polynomial transformations are useful in the following sense. If problem P_1 polynomially transforms to problem P_2, problem P_2 is at least as hard as P_1: Given an algorithm for problem P_2, we can always use it to solve problem P_1 with comparable (i.e., polynomial or not) running times. If the algorithm for P_2 is polynomial-time, then by using it (and the polynomial transformation), we can also solve P_1 in polynomial time. If P_1 is polynomially transformable to P_2, then P_2 is at least as hard as P_1. The possibility of making this transformation does not imply that P_1 is as hard as P_2. In fact, P_1 might be easy, while P_2 is hard. As an example, we can transform the minimum cost flow problem P_1 to an integer linear programming problem P_2. Although no known algorithm will solve the integer linear programming problem in polynomial time, as we have seen in the text, several algorithms will solve the minimum cost flow problem in polynomial time. As shown by this example, even though we might be able to polynomially reduce (or transform) an easy problem to a more difficult problem, this transformation does not imply that the easy problem is difficult. Whenever we can polynomially reduce a given problem to an easy problem, though, we can be assured that the given problem is easy.

B.3 PROBLEM CLASSES $\mathcal{P}$, $\mathcal{NP}$, $\mathcal{NP}$-COMPLETE, AND $\mathcal{NP}$-HARD

In this section we study the problem classes $\mathcal{P}$, $\mathcal{NP}$, and $\mathcal{NP}$-complete and discuss relationships among these classes.

Class $\mathcal{P}$

We say that a recognition problem P_1 belongs to class $\mathcal{P}$ if some polynomial-time algorithm solves problem P_1. In this book we have seen several examples of problems that belong to class $\mathcal{P}$; the recognition versions of the following problems belong to

class $\mathcal{P}$: the shortest path problem, the maximum flow problem, the minimum cost flow problem, assignment and matching problems, and the minimum spanning tree problem.

Class $\mathcal{NP}$

Roughly speaking, we say that a recognition problem P_1 is in the class $\mathcal{NP}$, if for every yes instance I of P_1, there is a short (i.e., polynomial length) verification that the instance is a yes instance.

Throughout this book, our standard measure of complexity of a problem has been the difficulty of solving the problem. However, the class $\mathcal{NP}$ deals with another measure of complexity that is more closely related to the idea of a proof. Consider, for example, the TSP-recognition-I. If someone hands you a yes instance and a tour W of length at most k^* and asks you to verify whether the problem instance is a yes instance, you can do so rapidly in $O(n)$ time by examining the tour and checking whether it passes through every node exactly once and has a length of no more than k^*.

Next, suppose that you are handed a no instance of TSP-recognition-I and you are asked to prove that it is no instance; then you are in serious trouble. There is no obvious proof other than (1) enumerating all possible tours and verifying that each tour length is greater than k^*, or (2) using any algorithm for TSP-optimization to determine the optimal tour length L and verifying that $L > k^*$. Unfortunately, the time to implement either of these approaches is not polynomial in the size of the instance I. Indeed, a proof of a no instance might (in the worst case) require an exponential amount of time. We therefore see a peculiar asymmetry in TSP-recognition-I. Although we might require the same amount of time to determine whether a given instance is a yes instance or a no instance, and need only polynomial time to prove the correctness of a given yes instance, we might require exponential time to prove that an instance is a no instance.

This situation is somewhat akin to proving theorems. When we ask a student to prove or disprove a conjecture, she focuses on how long it takes to find a proof. Thus we might view a conjecture to be quite difficult if the student requires a very long time to settle it either way. In contrast, suppose that we hand her a theorem along with its proof and ask her to verify the proof. Here the student's task is easy if she can quickly verify the proof. Thus the student's measure of difficulty of the theorem is how long it takes to verify the correctness of the given proof, not how long it takes her to develop the proof on her own. Needless to say, verifying a given proof of a difficult theorem is substantially easier than developing its proof.

We now make these notions more formal. Let P_1 be a recognition problem. For an instance I of P_1, let $| I |$ denote the *size* of the instance (i.e., the number of digits or bits needed to represent I; see Section 3.2 for a discussion of how to measure problem sizes). We refer to a proof that an instance is a yes instance as its *certificate*. For example, for the TSP-optimization-I, the certificate of a yes instance is a tour W whose length is at most k^*. Let CR(I) denote the certificate of a yes instance I, and let $|\text{CR}(I)|$ denote its size. We refer to an algorithm that can verify the correctness of a given certificate, that is, that the certificate establishes the instance as a yes instance, as a *certificate checking algorithm*. For example, for TSP-optimi-

zation-I, the certificate checking algorithm might be an algorithm that scans the nodes in a given certificate and verifies that each node is visited exactly once and that the length of the tour is at most k^*. We can now give a formal definition of the class $\mathcal{NP}$.

We say that a recognition problem P_1 is in the class $\mathcal{NP}$ if some certificate checking algorithm $\mathcal{AL}$ and polynomial $p(\cdot)$ satisfy the following properties:

1. Every yes instance I of P_1 has a certificate CR(I).
2. The algorithm $\mathcal{AL}$ can verify the correctness of CR(I) in at most $p(|\ I\ |)$ steps.

We say that a certificate is *succinct* if it, together with some polynomial $p(\cdot)$, satisfies these conditions. Note that since the time to verify the correctness of the certificate CR(I) must be polynomial in $|\ I\ |$, $|\text{CR}(I)\ |$ must also be polynomial in $|\ I\ |$.

Observe that the definition of the class $\mathcal{NP}$ implies that every problem in the class $\mathcal{P}$ is also in $\mathcal{NP}$. Let P_1 be a problem in the class $\mathcal{NP}$ and let $\mathcal{AL}_1$ be a polynomial-time algorithm for solving P_1. In this case we could choose the null set as the certificate and choose the algorithm $\mathcal{AL}_1$ itself as the certificate checking algorithm. Then for any given yes instance I of P_1, the algorithm $\mathcal{AL}_1$ can verify the correctness of I in polynomial time.

We next introduce some additional problems that are in the class $\mathcal{NP}$.

1. *Hamiltonian cycle problem.* Does a given directed network $G = (N, A)$ contain a directed cycle that visits each node in the network exactly once?
2. *Partition problem.* Given a finite set N of elements with element values $w(\cdot)$, does some subset $S \subseteq N$ satisfy the property that $\sum_{i \in S} w(i) = \sum_{i \in N\text{-}S} w(i)$?
3. *3-cover problem.* Given a collection of m (possibly overlapping) sets $S_1, S_2, \ldots, S_m$, each with three elements in a ground set $\{1, 2, \ldots, n\}$, do $n/3$ pairwise disjoint sets from this collection span the entire ground set (i.e., the union of the sets is $\{1, 2, \ldots, n\}$)?
4. *Integer programming feasibility problem.* Given a $p \times q$ constraint matrix $\mathcal{A}$ and a p-vector b, does some nonnegative integer q-vector x satisfy the equations $\mathcal{A}x = b$?

We now prove that some of these problems are in the class $\mathcal{NP}$.

TSP-recognition-II. In this case the certificate of a yes instance is a tour W of length at least k^*. The certificate checking algorithm can easily verify the correctness of W in $O(n)$ steps. Consequently, TSP-recognition-II is in the class $\mathcal{NP}$.

TSP-recognition-III. In this case there is no obvious succinct certificate of a yes instance (enumerating all the tours of the network G requires an exponential amount of space and time), although there is a succinct certificate for a no instance. Therefore, we cannot conclude that TSP-recognition-III is in the class $\mathcal{NP}$. (It probably is not in the class $\mathcal{NP}$, although no one has yet been able to prove this fact.)

Integer programming feasibility problem. At first glance this problem appears to be in the class $\mathcal{NP}$. If the integer programming problem has a feasible solution x, then x is a certificate and we can easily verify the correctness of this certificate in time that is polynomial in $|I|$ and x. The potential difficulty is that the size of x might not be polynomial in $|I|$ and the definition of the class $\mathcal{NP}$ requires that the certificate be polynomially bounded in $|I|$. Nevertheless, the integer programming feasibility problem is in the class $\mathcal{NP}$, because a deep theorem of integer programming states that if an instance I is feasible, some feasible solution has a size that is polynomially bounded in the size of $|I|$.

We ask the reader to prove that the remaining problems we have introduced in this appendix are in the class $\mathcal{NP}$.

Class $\mathcal{NP}$-Complete

A recognition problem P_1 is said to be *$\mathcal{NP}$-complete* if (1) $P_1 \in \mathcal{NP}$, and (2) all other problems in the class $\mathcal{NP}$ polynomially transform to P_1.

Property B.1 implies that if there is an efficient algorithm for some $\mathcal{NP}$-complete problem P_1, there is an efficient algorithm for *every* problem in the class $\mathcal{NP}$. As a result, an $\mathcal{NP}$-complete problem is (in a certain technical sense) at least as hard as every other problem in the class $\mathcal{NP}$. The class $\mathcal{NP}$ is a broad class of problems that includes all the "hard nuts" of combinatorial optimization such as the TSP, the 3-cover problem, and the Hamiltonian cycle problem. At first glance, the definition of $\mathcal{NP}$-complete problems might appear to be so restrictive that we might be tempted to think that very few problems are $\mathcal{NP}$-complete, and that proving a problem to be $\mathcal{NP}$-complete would be a remarkable feat. Nevertheless, researchers have shown that many problems are $\mathcal{NP}$-complete, including all three problems in $\mathcal{NP}$ that we listed earlier: the Hamiltonian cycle problem, the partition problem, and the 3-cover problem. Since we will be using these $\mathcal{NP}$-completeness results in our subsequent discussion, we state these results as a theorem (which we do not prove).

Theorem B.2. *Each of the following problems is $\mathcal{NP}$-complete:*

(*a*) *The Hamiltonian cycle problem*
(*b*) *The partition problem*
(*c*) *The 3-cover problem* ◆

Consider two $\mathcal{NP}$-complete problems P_1 and P_2. By definition, P_1 polynomially transforms to P_2 and P_2 polynomially transforms to P_1. This observation implies that each $\mathcal{NP}$-complete problem polynomially transforms to every other $\mathcal{NP}$-complete problem. Therefore, all $\mathcal{NP}$-complete problems are in some sense comparable in their computational difficulty. If we succeed in developing an efficient algorithm for one $\mathcal{NP}$-complete problem, we know that every problem in the class $\mathcal{NP}$ is polynomially solvable, and $\mathcal{P} = \mathcal{NP}$. Most researchers today conjecture that no polynomial-time algorithm could possibly solve any $\mathcal{NP}$-complete problem. Equivalently, most researchers believe that $\mathcal{P} \neq \mathcal{NP}$, but the issue remains unresolved.

Figure B.1 gives a schematic representation of relationships between different problem classes studied in the section.

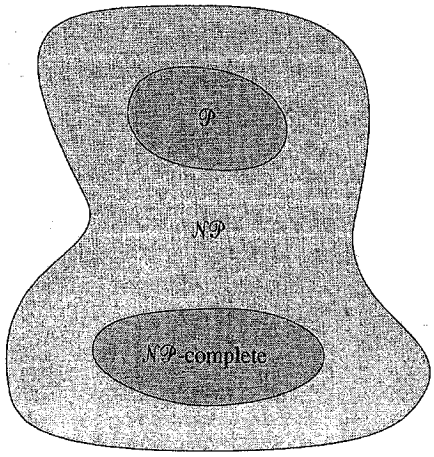


Figure B.1 Relationship between various problem classes.

Class $\mathcal{NP}$-Hard

A recognition problem P_1 is said to be *$\mathcal{NP}$-hard* if all other problems in the class $\mathcal{NP}$ polynomially reduce to P_1.

The class $\mathcal{NP}$-hard is broader than the class $\mathcal{NP}$-complete because it includes the class $\mathcal{NP}$ as well as problems that are not in class $\mathcal{NP}$.

B.4 PROVING $\mathcal{NP}$-COMPLETENESS RESULTS

Proving the existence of the first $\mathcal{NP}$-complete problem was indeed quite challenging, but once researchers discovered one (or a few) $\mathcal{NP}$-complete problems, showing that some other problems are $\mathcal{NP}$-complete was much easier. To prove that a problem P_2 is $\mathcal{NP}$-complete, we must establish two facts:

1. P_2 is in $\mathcal{NP}$.
2. All other problems in $\mathcal{NP}$ polynomially transform to P_2.

In practice we establish part (b) by showing that a known $\mathcal{NP}$-complete problem, say P_1, polynomially transforms to P_2. Since P_1 is a $\mathcal{NP}$-complete problem, every other problem in $\mathcal{NP}$ polynomially transforms to P_1. Moreover, since P_1 polynomially transforms to P_2, every problem in the class $\mathcal{NP}$ polynomially transforms to P_2.

For most problems encountered in practice, we can find recognition versions that are in $\mathcal{NP}$. Showing that problem P_2 is in $\mathcal{NP}$ is generally (but not always) straightforward. However, identifying a known $\mathcal{NP}$-complete problem P_1 that would polynomially transform to P_2 frequently is more challenging. The selection of the problem P_1, for which this transformation is simple and direct, requires some skill, experience, and insight. Indeed, proving $\mathcal{NP}$-completeness results is more of an art than a science. In the following discussion we illustrate briefly how to establish $\mathcal{NP}$-completeness results by selecting a few problems and proving that they are $\mathcal{NP}$-complete. We have selected problems related to the network flow problems discussed in this book and for which the resulting transformations are direct. In this discussion, we use the fact that the Hamiltonian cycle, partition, and 3-cover prob-

lems that we have discussed earlier in this appendix are $\mathcal{NP}$-complete. We do not prove that these problems are in $\mathcal{NP}$, because doing so is straightforward in each case.

Traveling salesman problem. We show that the Hamiltonian cycle problem polynomially transforms to the TSP. Suppose that we wish to solve the Hamiltonian cycle problem in the graph $G = (N, A)$. We construct a complete directed graph $G' = (N, A')$ with arc lengths defined as follows: $c_{ij} = 1$ if $(i, j) \in A$ and $c_{ij} = 2$ otherwise. We define $k^* = n$. With this definition of arc lengths, the tour constraint $\sum_{(i,j)\in W} c_{ij} \leq k^*$ is satisfied if and only if W belongs to A and thus W is a Hamiltonian cycle of the original problem. Consequently, TSP has a yes instance if and only if G contains some tour of length n, and this occurs if and only if G has a Hamiltonian cycle. Notice that for every triple i, j, k, $c_{ik} \leq c_{ij} + c_{jk}$, and thus the arc costs satisfy the *triangle inequality*. We have therefore shown that the traveling salesman problem is $\mathcal{NP}$-complete even if arc costs satisfy the triangle inequality.

Hamiltonian path problem. Does a given directed network $G = (N, A)$ contain a directed path that visits every node exactly once (the path can start at any node and can end at any other node)?

We transform the Hamiltonian cycle problem to the Hamiltonian path problem. Suppose that we want to determine whether the graph G contains a Hamiltonian cycle. From G we construct a new graph G' as follows. We add a new node $n + 1$ and redirect every incoming arc at node 1 to node $n + 1$ (i.e., we replace arc $(i, 1)$ by the arc $(i, n + 1)$. We prove that G contains a Hamiltonian cycle if and only if G' contains a Hamiltonian path. Consider a Hamiltonian cycle $1 = i_1 - i_2 - \cdots - i_n - i_1$ in G; this cycle corresponds to a Hamiltonian path $1 = i_1 - i_2 - \cdots - i_n - (n + 1)$ in G'. To see the converse, notice that every Hamiltonian path in G' must begin at node 1 (because this node has no incoming arc) and must end at node $n + 1$ (because this node has no outgoing arc). Moreover, every Hamiltonian path of the form $1 = j_1 - j_2 - \cdots - j_n - (n + 1)$ in G' corresponds to the Hamiltonian cycle $1 = j_1 - j_2 - \cdots - j_n - j_1$ in G. This conclusion establishes that we can solve the Hamiltonian cycle problem in G by solving a Hamiltonian path problem in G'.

Longest path problem. Does a given network $G = (N, A)$ contain a (simple) path from node s to node t with at least L arcs?

If $L = n - 1$, a path of length L from node s to node t is a Hamiltonian path. We have already seen that this problem is $\mathcal{NP}$-complete.

Knapsack problem. Given a finite set N of elements, the integers v^* and w^*, and a *value* $v(i)$ and a *weight* $w(i)$ associated with every element $i \in N$, does some subset $S \subseteq N$ satisfy the property that $\sum_{i\in S} v(i) \geq v^*$ and $\sum_{i\in S} w(i) \leq w^*$?

We show that the knapsack problem is $\mathcal{NP}$-complete using a transformation from the partition problem. Given a partition problem on a set N with element values $s(\cdot)$, we construct a knapsack problem on the same set as follows. We define $v(i) = w(i) = s(i)$ for every element $i \in N$, and $v^* = w^* = K = (\sum_{i\in S} s(i))/2$. The knapsack problem then finds a set S for which $\sum_{i\in S} s(i) \leq K$ and $\sum_{i\in S} s(i) \geq K$,

so $\sum_{i \in S} s(i) = K$. Now notice that $\sum_{i \in N-S} s(i) = \sum_{i \in N} s(i) - \sum_{i \in S} s(i) = 2K - K = K$. Therefore, the set S is a partition.

Constrained shortest path problem. Given a graph $G = (N, A)$, integers c and τ, and an arc length c_{ij} and a traversal time τ_{ij} associated with every arc $(i, j) \in A$, does the graph contain a directed path from node s to node t whose length is at most c and whose traversal time is at most τ?

We show that the constrained shortest path problem is $\mathcal{NP}$-complete by transforming the knapsack problem to it. For a given knapsack problem, consider the constrained shortest path problem shown in Figure B.2 with $c = -v$ and $\tau = w$. As is easy to verify, the knapsack problem has a feasible solution if and only if the graph in Figure B.2 contains a path whose length is at most $-v = c$ and whose traversal time is at most w. Therefore, we can solve the knapsack problem by solving a constrained shortest path problem.

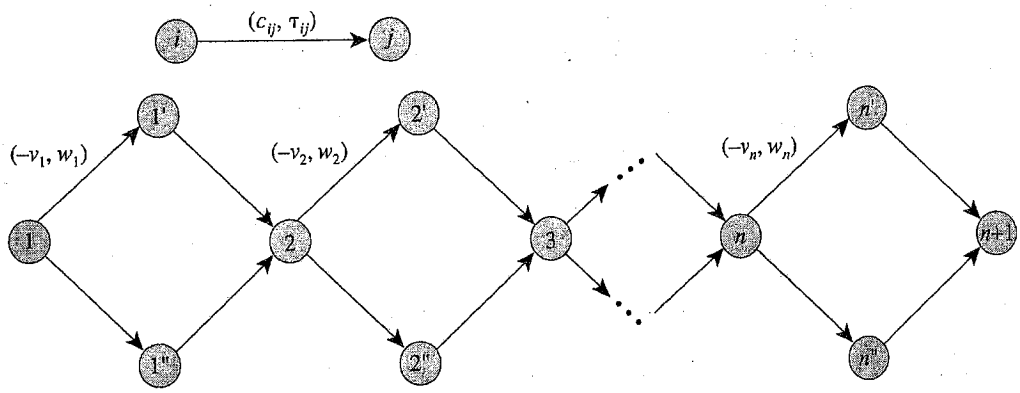


Figure B.2 Transforming the knapsack problem into the constrained shortest path problems. (Arcs without any data have zero cost and zero traversal time.)

Integer generalized flow problem. Given a network $G = (N, A)$, a number v^*, and an arc multiplier μ_{ij} and an arc capacity u_{ij} associated with every arc $(i, j) \in A$ (the network has no associated arc costs), does the network contain an integer generalized flow with a total flow of value of at least v^* into the sink?

We will show that 3-cover problem reduces to an integer generalized flow problem in an appropriately defined network. For a given 3-cover problem, we consider the integer generalized flow problem shown in Figure B.3. In this network a node representing the set S_i has three outgoing arcs directed toward the nodes representing the three elements of the set S_i; these arcs have unit multipliers and unit capacities. Now notice that if we send a unit flow on the arc (s, S_i) from node s, 3 units arrive at node S_i, which in turn sends 1 unit along each of the outgoing arcs. Using this observation, we can easily establish a one-to-one correspondence between 3-covers and integer generalized flows. This correspondence implies that the 3-cover problem has a yes instance if and only if the integer generalized flow problem has a yes instance.

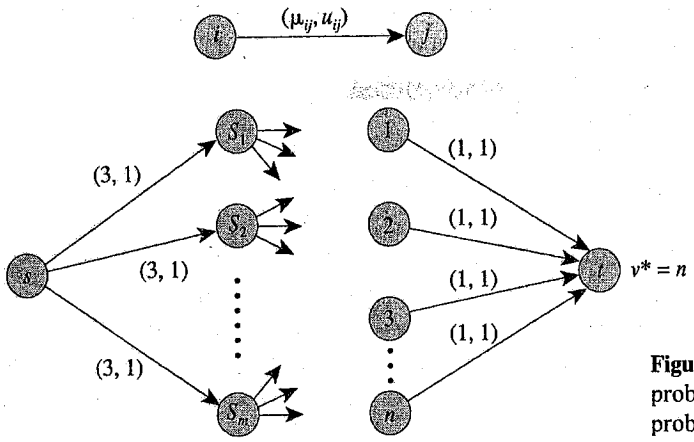


Figure B.3 Transforming a 3-cover problem into an integer generalized flow problem.

0–1 integer programming problem. Given an $p \times q$ matrix $\mathcal{A}$ and an integer p-vector b, does some integer q-vector x, whose components are all 0 or 1, satisfy the inequality system $\mathcal{A}x \leq b$?

The fact that the 0–1 integer programming problem is $\mathcal{NP}$-complete is fairly straightforward to establish because we can formulate most of the problems discussed previously as 0–1 integer programs. Consider, for instance, the knapsack problem which can be formulated as determining binary variables x_j's satisfying the following constraints:

$$-\sum_{j=1}^{n} v_j x_j \leq -v^*$$

and

$$\sum_{j=1}^{n} w_j x_j \leq w^*,$$

which is a 0–1 integer programming problem.

Weak and Strong $\mathcal{NP}$-Completeness and the Similarity Assumption

Throughout much of this book, we have emphasized that data for network flow problems typically satisfies the similarity assumption in practice, i.e., for some fixed integer k, $C = O(n^k)$ and $U = O(n^k)$. We have also pointed out that algorithms with running times involving C or U rather than $\log C$ or $\log U$ should be considered pseudo-polynomial and not polynomial. For example, Dial's algorithm for the shortest path problem requires time $O(m + nC)$ and is thus a pseudo-polynomial algorithm. However, if the data is known to satisfy the similarity assumption, any pseudo-polynomial time algorithm becomes a polynomial time algorithm. For example, if $C = O(n^k)$ then the running time for Dial's algorithm would become $O(m + n^{k+1})$.

If a problem is $\mathcal{NP}$-complete even when the similarity assumption is satisfied, we say that the problem is *strongly $\mathcal{NP}$-complete*. If a problem is $\mathcal{NP}$-complete but

it fails to be $\mathcal{NP}$-complete when the similarity assumption is satisfied, then we call it *weakly $\mathcal{NP}$-complete*. For example, the knapsack problem and the constrained shortest path problem are both weakly $\mathcal{NP}$-complete. The traveling salesman problem, the Hamiltonian path problem, the longest path problem, the integer generalized flow problem, and 0–1 integer programming are all strongly $\mathcal{NP}$-complete.

An an aside, we showed that integer programming is $\mathcal{NP}$-complete by showing that the knapsack problem is a special case. While this proof is very simple, it has the disadvantage of showing only weak $\mathcal{NP}$-completeness of integer programming (why?). In fact, we could also have showed that the traveling salesman problem is a special case of 0–1 integer programming, and this transformation would show the strong $\mathcal{NP}$-completeness of integer programming (assuming that we had already established the strong $\mathcal{NP}$-completeness of the traveling salesman problem).

B.5 CONCLUDING REMARKS

In this book we have discussed a variety of network flow problems and developed many polynomial-time algorithms for solving these problems. It is interesting to observe that simple generalizations of these problems often are $\mathcal{NP}$-complete. The shortest path problem is polynomially solvable, but the constrained shortest path problem is $\mathcal{NP}$-complete. The maximum flow problem in directed networks with nonnegative lower bounds is polynomially solvable (see Section 6.5), but the maximum flow problem in undirected networks with nonnegative lower bounds on arc flows is $\mathcal{NP}$-complete. We have seen how to find a minimum cut in a network efficiently, but the maximum cut problem is $\mathcal{NP}$-complete. Several efficient algorithms will solve the two-dimensional matching problem (i.e., the matching of objects two at a time), but a three-dimensional version of the problem is $\mathcal{NP}$-hard. In Chapter 19 we have shown how to solve the Chinese postman problem efficiently in directed as well as undirected networks; in mixed networks (i.e., those whose arcs can both be directed and undirected) this problem is $\mathcal{NP}$-complete. The generalized network flow problem and the multicommodity flow problem are polynomially solvable, since they are special cases of the linear programming problem, which has several polynomial-time algorithms. Unfortunately, integer versions of both the problems are $\mathcal{NP}$-complete. The literature cited in the reference notes contains proof of some of these results. We do not intend to imply that *every* single generalization of network flow problems studied in this book is $\mathcal{NP}$-complete. Nevertheless, most generalizations are $\mathcal{NP}$-complete, except some generalizations that happen to be linear programming problems.

Like worst-case complexity theory, the theory of $\mathcal{NP}$-completeness is pessimistic: It always focuses on what happens in the worst case. The worst-case behavior of an algorithm might be markedly different than its behavior in practice. For example, from a worst-case perspective a problem whose best available algorithm runs in time $O(n^{100})$ is an easy problem, despite the fact that the $O(n^{100})$ time is terrible running time in practice. Similarly, the theory will regard an $\mathcal{NP}$-complete problem with an $O(n^{0.01 \log n})$ time bound as a hard problem, even though for $n \leq 2^{100}$, the running time is better than linear. Indeed, several $\mathcal{NP}$-complete problems can be solved very efficiently in practice, possibly faster than some problems in class $\mathcal{P}$ of comparable size. However, we can safely say that $\mathcal{NP}$-complete problems "some-

times'' do not have algorithms that can solve large practical instances in reasonable time, whereas problems in class $\mathcal{P}$ ''often'' have.

In concluding this discussion of $\mathcal{NP}$-completeness proofs, we might note that we have considered just one set of issues within the very broad field of computational complexity. Many other issues and refinements arise when we try to understand the structure of algorithms and of computers and the computations they perform. In particular, it is possible to classify algorithms in terms of the space they require and it is possible to distinguish algorithms and problems by imposing more structure on the underlying data.

REFERENCE NOTES

The field of $\mathcal{NP}$-completeness is vast and replete with deep results. In this appendix we have discussed only some of the most elementary results. Cobham [1964] and, independently, Edmonds [1965a] introduced the class $\mathcal{P}$. Cook [1971] introduced the notion of $\mathcal{NP}$-completeness and proved that the satisfiability problem is $\mathcal{NP}$-complete. Independently, Levin [1973] developed this notion. Karp [1972] showed that a rich class of problems, including the traveling salesman problem and the knapsack problem, is $\mathcal{NP}$-complete. The set of problems known to be in the class $\mathcal{NP}$-complete grew at a phenomenal pace; this set now contains thousands of problems. The book by Garey and Johnson [1979] is still the best guide to $\mathcal{NP}$-completeness results. The story given in Section B.1 has also been adapted from this book. Books by Papadimitriou and Steiglitz [1982] and by Cormen, Leiserson, and Rivest [1990] are good additional references on this topic.

Appendix C

LINEAR PROGRAMMING

Inequality is the cause of all local movements.
—Leonardo da Vinci

Chapter Outline

C.1 Introduction
C.2 Graphical Solution Procedure
C.3 Basic Feasible Solutions
C.4 Simplex Method
C.5 Bounded Variable Simplex Method
C.6 Linear Programming Duality

C.1 INTRODUCTION

Linear programming is perhaps *the* core model of constrained optimization; and the simplex method for solving linear programming has been one of the most significant algorithmic discoveries of this century. Developed in 1947, the simplex method has stood the test of time, having been applied to thousands of applications in fields as diverse as agriculture, communications, computer science, engineering design, finance, industrial and military logistics, manufacturing, transportation, and urban planning. Moreover, methods and concepts developed for linear programming—such as duality theory, decomposition methods, and sensitivity analysis—have served as important base methodologies for stimulating discoveries in many other fields within the sphere of optimization. For these reasons, linear programming rightly deserves its position as one of the basic cornerstones of applied mathematics, computer science, and operations research.

In this appendix we summarize some of the basic ideas of linear programming and the simplex method. We do so for at least two reasons. First, a great majority of the models that we have developed in this book, and indeed most of the models encountered in the field of network optimization, are either linear programs or integer programming extensions of linear programs. Therefore, a firm understanding of linear programming is valuable for understanding the structure of the models we have been studying. Second, although we have attempted to develop much of network flows from first principles and to use basic combinatorial ideas rather than more general methodologies and concepts of linear programming, we have, by necessity, needed to invoke ideas from linear programming on many occasions, sometimes as a basic tool in our development and at other times to make appropriate connections between the ideas we have been developing and more general concepts. Therefore,

we have needed to rely on several central ideas of linear programming. This appendix serves to make our coverage as complete as possible; it functions both (1) as an introduction to linear programming for those who have only passing familiarity with this topic, and (2) as a review of linear programming for those readers who are already conversant with this topic.

A linear program is an optimization problem with a linear objective function, a set of linear constraints, and a set of nonnegativity restrictions imposed upon the underlying decision variables; that is, it is an optimization model of the form

$$\text{Minimize} \quad \sum_{j=1}^{q} c_j x_j \tag{C.1a}$$

subject to

$$\sum_{j=1}^{q} a_{ij} x_j = b(i) \qquad \text{for all } i = 1, \ldots, p, \tag{C.1b}$$

$$x_j \geq 0 \qquad \text{for all } j = 1, \ldots, q. \tag{C.1c}$$

This problem has q nonnegative decision variables x_j and p equality constraints (C.1b). (In many texts, m denotes the number of equality constraints and n denotes the number of decision variables. This notation, unfortunately, is the reverse of the convention in network flows, since network flow systems contain one constraint per node and one variable per arc. For this reason we do not use the notation of m and n to denote the number of constraints and variables of a linear program.)

We assume, by multiplying the ith equation by -1, if necessary, that the right-hand-side coefficient $b(i)$ of each constraint $i = 1, \ldots, p$ is nonnegative. We might note that we could formulate a linear program in several alternative ways; for example, the objective function could be stated in maximization form, or the constraints could be in a less than or equal to or greater than or equal to form. The linear programming literature frequently refers to the formulation (C.1)—a model with equality constraints, nonnegative variables, and a minimization form of the objective function—as the *standard form* of a linear program.

In economic planning, each decision variable models one particular production activity (x_j is the level of that activity), each constraint corresponds to a scarce resource, and the coefficient a_{ij} indicates the amount of the ith resource consumed per unit of the jth production activity. In this instance the model seeks the "best" use of the scarce resources, that is, the production plan that uses the available resources to produce the maximum possible revenue (assuming a maximization form of the objective function).

In matrix notation, the linear programming model has the following form:

$$\text{Minimize} \quad cx \tag{C.2a}$$

subject to

$$\mathcal{A}x = b, \tag{C.2b}$$

$$x \geq 0. \tag{C.2c}$$

In this formulation the matrix $\mathcal{A} = (a_{ij})$ has p rows and q columns, the vector $c = (c_j)$ is a q-dimensional row vector, the vectors $x = (x_j)$ and $b = (b(i))$ are q-

and p-dimensional column vectors, respectively. We let $\mathcal{A}_j$ denote the column of $\mathcal{A}$ corresponding to the variable x_j. We assume that the rows of the matrix are linearly independent; thus the system $\mathcal{A}x = b$ contains no redundant equations. In terms of linear and matrix algebra (we assume modest background concerning these topics), this assumption states that the rows of the matrix $\mathcal{A}$ are linearly independent; that is, the matrix $\mathcal{A}$ has full row rank.

For the special case of the minimum cost network flow problem, each component of the decision variable x corresponds to the flow on an arc and the matrix $\mathcal{A}$ has one row for each node of the underlying network. In this case the matrix $\mathcal{A}$ is the node–arc incident matrix N that we introduced in Chapter 1.

In the following sections we describe the rudiments of linear programming theory. We begin by illustrating the underlying geometry of linear programs and by introducing the fundamental concepts of extreme points and basic feasible solutions. Then we describe the key features of the simplex method and of a variant that permits us to efficiently handle upper bounds on the decision variables. We conclude this appendix by introducing the basic features and key results of linear programming duality theory.

C.2 GRAPHICAL SOLUTION PROCEDURE

Linear programs involving only two or three variables have a convenient graphical representation that helps in understanding the nature of linear programming and of the simplex method. We illustrate this procedure using the following example, stated in inequality form with a maximization objective:

$$\text{Maximize} \quad z = x_1 + x_2 \tag{C.3a}$$

subject to

$$2x_1 + 3x_2 \leq 12, \tag{C.3b}$$

$$x_1 \leq 4, \tag{C.3c}$$

$$x_2 \leq 3, \tag{C.3d}$$

$$x_1, x_2 \geq 0. \tag{C.3e}$$

The shaded region in Figure C.1 is the set of feasible solutions for this problem. The set of feasible solutions for the linear programming problem is generally referred to as a *polyhedron*. The points A, B, C, D, and E are the *extreme points* of the polyhedron; these points are formed by the intersection of the lines corresponding to various constraints. Note that extreme points do not lie on any line segment joining two other points in the polyhedron. More formally, a vector x is a strict convex combination of distinct vectors $x^1, x^2, \ldots, x^k$ if $x = \theta^1 x^1 + \theta^2 x^2 + \cdots + \theta^k x^k$ for a set of weights $\theta^i > 0$ satisfying the condition $\sum_{i=1}^{k} \theta^i = 1$. A vector x is an extreme point of a polyhedron if it is not a strict convex combination of two distinct points in the polyhedron, that is, cannot be represented as $x = \theta x^1 + (1 - \theta)x^2$ for some weight $0 < \theta < 1$ and two distinct points x^1 and x^2 of the polyhedron. It is an easy exercise to show that if x is a strict convex combination of $k > 2$ distinct points in the polyhedron, it is not an extreme point.

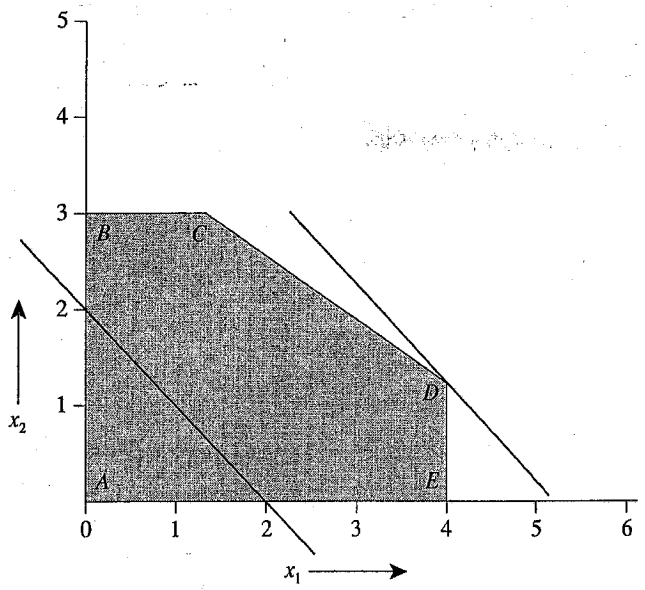


Figure C.1 Set of feasible solutions for a linear program.

The linear programming problem seeks a point (x_1, x_2) in the polyhedron *ABCDEA* that achieves the maximum possible value of $x_1 + x_2$. Equivalently, we wish to determine the largest value of w for which the line $x_1 + x_2 = z$ has at least one point in common with the polyhedron *ABCDEA*. The lines obtained for different values of z are parallel to each other. Since these lines move farther away from the origin as z becomes larger, to maximize $x_1 + x_2$ we need to slide the line $x_1 + x_2 = z$ away from the origin as far as possible so that it has some point in common with the polyhedron *ABCDEA*. We can do so until we reach some extreme point, point D in our case, at which point the line has only one point in common with the polyhedron; at this point, for any further translation of the line away from the origin, no matter how small, the line has no point in common with the polyhedron. Thus point D is an optimal solution of our linear programming problem; its objective function value equals 16/3.

This graphical solution procedure illustrates an important property of linear programs, namely that every linear program always has an extreme point solution as one of its optimal solutions. Therefore, to solve a linear programming problem, we can focus only on the extreme point solutions. Consequently, we need to consider only a finite number of solutions. The simplex method, described in the next section, makes use of this extreme point property. It starts at some feasible extreme point and visits "adjacent" extreme points, improving the objective function values of the solution at each step, until it reaches an optimal extreme point. For instance, in our example, if the simplex method starts at point A, it might visit the points B and C before reaching the optimal extreme point D. Alternatively, it might follow the path A, E, D.

C.3 BASIC FEASIBLE SOLUTIONS

Our description of the simplex method requires that the linear program to be solved be stated in the standard form (C.1); that is, the objective function is in the minimization form; each constraint, except for the nonnegative condition imposed on

the decision variables, is an equality; and each right-hand-side coefficient $b(i)$ is nonnegative. Any linear program not in the standard form can be brought into standard form through simple transformations. For example, maximizing $\sum_{j=1}^{q} c_j x_j$ is equivalent to minimizing $-\sum_{j=1}^{q} c_j x_j$. To model an inequality constraint $\sum_{j=1}^{q} a_{ij} x_j \leq b(i)$ as an equality constraint, we could add a new nonnegative "slack variable" y_i, with zero cost, and writing the inequality as $\sum_{j=1}^{q} a_{ij} x_j + y_i = b(i)$.

During its execution, the simplex method modifies the original linear program stated in the standard form by performing a series of one or more of the following elementary row operations:

1. Multiplying a row (i.e., constraint) by a constant, or
2. Adding one row to another row or to the objective function.

Since we have stated all the constraints in the equality form, row operations do not affect the set of feasible solutions of the linear program. As an illustration, consider the following linear program:

$$\text{Minimize} \quad z(x) = x_1 + x_2 - 8x_3 + 6x_4 \tag{C.4a}$$

subject to

$$2x_1 + x_2 - 14x_3 + 10x_4 = 16, \tag{C.4b}$$

$$x_1 + x_2 - 11x_3 + 7x_4 = 10, \tag{C.4c}$$

$$x_1, x_2, x_3, x_4 \geq 0. \tag{C.4d}$$

Subtracting (C.4c) from (C.4a) and (C.4b) gives the following equivalent linear program:

$$\text{Minimize} \quad z(x) = 0x_1 + 0x_2 + 3x_3 - x_4 + 10 \tag{C.5a}$$

subject to

$$x_1 - 3x_3 + 3x_4 = 6, \tag{C.5b}$$

$$x_1 + x_2 - 11x_3 + 7x_4 = 10, \tag{C.5c}$$

$$x_1, x_2, x_3, x_4 \geq 0. \tag{C.5d}$$

Subtracting (C.5b) from (C.5c) gives another equivalent formulation of this linear program:

$$\text{Minimize} \quad z(x) = 0x_1 + 0x_2 - 3x_3 + x_4 + 10 \tag{C.6a}$$

subject to

$$x_1 - 3x_3 + 3x_4 = 6, \tag{C.6b}$$

$$x_2 - 8x_3 + 4x_4 = 4, \tag{C.6c}$$

$$x_1, x_2, x_3, x_4 \geq 0. \tag{C.6d}$$

Since the linear program (C.6) is equivalent to (C.4), both have the same optimal solutions. A linear program like (C.6) is said to be in the *canonical form* if it satisfies the following *canonical property*.

Canonical Property. *The formulation has one decision variable isolated in each constraint; the variable isolated in a given constraint has a coefficient of* $+1$ *in that constraint and does not appear in any other constraint, nor does it appear in the objective function.*

The previous formulation satisfies the canonical property because x_1 is isolated in constraint (C.6b) and x_2 in constraint (C.6c).

A linear program typically has a large number of canonical forms since there are many ways to isolate decision variables in the constraints. The canonical form in (C.6) has the following attractive feature. Assigning any values to x_3 and x_4 uniquely determines the values of x_1 and x_2. In fact, setting $x_3 = x_4 = 0$ immediately gives the solution $x_1 = 6$ and $x_2 = 4$. Solutions such as these, known as *basic feasible solutions*, play a central role in the simplex method. In general, given a canonical form for any linear program, we obtain a basic solution by setting the variable isolated in constraint i, called the ith *basic variable*, equal to the right-hand side of the ith constraint, and by setting the remaining variables, called *nonbasic*, all to value zero. Collectively, the basic variables are known as a *basis*.

In general, we obtain a basic feasible solution as follows. We isolate a variable in each constraint. For simplicity, assume that we have isolated the variable x_i in the ith constraint. Let $\mathbf{B} = \{1, 2, \ldots, p\}$ denote the index set of basic variables and let $\mathbf{L} = \{p + 1, p + 2, \ldots, q\}$ denote the index set of nonbasic variables (we choose the mnemonic **L** for the index set of nonbasic variables because we set the value of the corresponding variables to their lower bound—that is, value 0—in the basic solution defined by this index set). We refer to the pair $(\mathbf{B}, \mathbf{L})$ as a *basis structure* of the linear problem. For a given basis structure $(\mathbf{B}, \mathbf{L})$, we can compatibly partition the columns of the constraint matrix $\mathcal{A}$. Let $\mathcal{B} = [\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_p]$ and $\mathcal{L} = [\mathcal{A}_{p+1}, \mathcal{A}_{p+2}, \ldots, \mathcal{A}_q]$. We refer to the $p \times p$ matrix $\mathcal{B}$ as a *basis matrix*. We also let $x_{\mathbf{B}} = [x_i : i \in \mathbf{B}]$ and $x_{\mathbf{L}} = [x_j : j \in \mathbf{L}]$ be a partitioning of the variables into subvectors corresponding to the index sets **B** and **L**. With this notation, we can rewrite the constraint matrix $\mathcal{A}x = b$ as

$$\mathcal{B}x_{\mathbf{B}} + \mathcal{L}x_{\mathbf{L}} = b. \tag{C.7}$$

We can convert (C.7) to the canonical form by premultiplying each term by $\mathcal{B}^{-1}$, the inverse of the basis matrix, giving

$$x_{\mathbf{B}} + \mathcal{B}^{-1}\mathcal{L}x_{\mathbf{L}} = \mathcal{B}^{-1}b. \tag{C.8}$$

We obtain a *basic solution* from (C.8) by setting each nonbasic variable to value zero. The resulting solution is

$$x_{\mathbf{B}} = \mathcal{B}^{-1}b \quad \text{and} \quad x_{\mathbf{L}} = 0.$$

We refer to this solution as a *basic feasible solution* if the value of each basic variable is nonnegative (i.e., $x_{\mathbf{B}} \geq 0$). We also say that the basis structure $(\mathbf{B}, \mathbf{L})$ is *feasible* if its associated basic solution is feasible. For some choices of the basis matrix, the corresponding basic solution will be feasible, and for some other choices it will not be feasible.

Converting a linear program to a canonical form requires that we invert the basis matrix, which is possible only if the columns associated with the basic variables

are linearly independent. If the associated columns are linearly dependent, the basis matrix is singular (i.e., its determinant is zero) and we cannot invert it. We shall therefore henceforth refer to a basis as a subset of p variables whose corresponding columns are linearly independent.

During its execution, the simplex method requires information about $\mathcal{B}^{-1}\mathcal{A}_j$, the updated column corresponding to the nonbasic variable x_j. We let $\bar{\mathcal{A}}_j = \mathcal{B}^{-1}\mathcal{A}_j$ and call this vector the *representation* of $\mathcal{A}_j$ with respect to the basis matrix $\mathcal{B}^{-1}$ (or, alternatively, basis **B**). We refer to this column vector as a representation because by premultiplying both sides of the equations $\bar{\mathcal{A}}_j = \mathcal{B}^{-1}\mathcal{A}_j$ by $\mathcal{B}$, we obtain $\mathcal{B}\bar{\mathcal{A}}_j = \mathcal{A}_j$, which implies that we can interpret the elements in the vector $\bar{\mathcal{A}}_j$ as weights that we use to multiply the columns of the basis matrix $\mathcal{B}$ in order to obtain the column $\mathcal{A}_j$. For notational convenience, we let $\bar{\mathcal{A}}_{\mathbf{L}}$ denote the matrix $\mathcal{B}^{-1}\mathcal{L}$ containing the column representations of all the nonbasic variables and let $\bar{b} = \mathcal{B}^{-1}b$; we refer to the vector $\bar{b}$ as the modified right-hand side. Finally, let $c_{\mathbf{B}} = (c_1, c_2, \ldots, c_p)$ and $c_{\mathbf{L}} = (c_{p+1}, c_{p+2}, \ldots, c_q)$ denote the cost vectors associated with the basic and nonbasic variables, respectively.

In a canonical form of a linear program, each basic variable has a zero coefficient in the objective function. We can obtain this special form of the objective function by performing a sequence of elementary row operations (i.e., multiplying constraints by some multipliers and subtracting them from the objective function). Any sequence of elementary row operations is equivalent to the following: Multiply each constraint i by a number $\pi(i)$ and subtract it from the objective function. This operation gives the equivalent objective function $\sum_{j=1}^{q} c_j x_j - \sum_{i=1}^{p} \pi(i)[\sum_{j=1}^{q} a_{ij}x_j - b(i)]$, or, collecting terms and letting $z_0 = \sum_{i=1}^{p} \pi(i)b(i)$,

$$z(x) = \sum_{j=1}^{p} [c_j - \sum_{i=1}^{p} \pi(i)a_{ij}]x_j + \sum_{j=p+1}^{q} [c_j - \sum_{i=1}^{p} \pi(i)a_{ij}]x_j + z_0. \qquad \text{(C.9)}$$

To obtain a canonical form, we select the vector π so that

$$c_j - \sum_{i=1}^{p} \pi(i)a_{ij} = 0 \qquad \text{for each } j \in \mathbf{B}. \qquad \text{(C.10)}$$

In matrix notation we select π so that

$$\pi\mathcal{B} = c_{\mathbf{B}}.$$

In this expression, $c_{\mathbf{B}} = (c_1, c_2, \ldots, c_p)$ is the cost vector associated with the basic variables. We refer to

$$\pi = c_{\mathbf{B}}\mathcal{B}^{-1}$$

as the *simplex multipliers* associated with the basis **B** and refer to $c_j^{\pi} = c_j - \sum_{i=1}^{p} \pi(i)a_{ij}$ as the *reduced cost* of the variable x_j. Note that $z_0 = \pi b = c_{\mathbf{B}}\mathcal{B}^{-1}b = c_{\mathbf{B}}x_{\mathbf{B}}$, which since $(x_{\mathbf{L}} = 0)$ is the value of the objective function corresponding to the basis **B**.

In Section C.2 we observed that linear programs always have extreme-point solutions and in this section we have focused on basic solutions. Are these concepts related? Indeed, they are. To conclude this section we show that extreme points and basic solutions are geometric and algebraic manifestations of the same concept.

Therefore, by finding an optimal basic solution, we are obtaining an optimal extreme point solution, and vice versa. Recall that columns of the constraint matrix $\mathcal{A}$ corresponding to the basis **B** are linearly independent.

Theorem C.1 (Extreme Points and Basic Feasible Solutions). *A feasible solution x to a linear program (C.1) is an extreme point if and only if the columns $\{\mathcal{A}_j : x_j > 0\}$ of the constraint matrix corresponding to the strictly positive variables are linearly independent.*

Proof. To simplify our notation, suppose that $x_1 > 0, x_2 > 0, \ldots, x_k > 0$ and $x_{k+1} = x_{k+2} = \cdots = x_q = 0$. Suppose that the columns $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_k$ are linearly dependent. Then there exist constants $\omega_1, \omega_2, \ldots, \omega_k$ (not all zero), satisfying the condition

$$\mathcal{A}_1\omega_1 + \mathcal{A}_2\omega_2 + \cdots + \mathcal{A}_k\omega_k = 0.$$

Let us define $\omega_{k+1} = \omega_{k+2} \cdots = \omega_q = 0$ and let ω denote the vector $(\omega_1, \omega_2, \ldots, \omega_q)$. Then since each of the first k components of the solution x are positive, for some sufficiently small value of the scalar θ,

$$x + \theta\omega \geq 0 \quad \text{and} \quad x - \theta\omega \geq 0.$$

Also, since $\mathcal{A}\omega = 0$, $\mathcal{A}(x + \theta\omega) = \mathcal{A}(x - \theta\omega) = \mathcal{A}x = b$. Therefore, both $x + \theta\omega$ and $x - \theta\omega$ are feasible for the linear program. But then $x = \frac{1}{2}(x + \theta\omega) + \frac{1}{2}(x - \theta\omega)$, which implies that x is not an extreme point (because it lies on the line joining the points $x + \theta\omega$ and $x - \theta\omega$). Therefore, we have shown that if x is an extreme point, the columns $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_k$ must be linearly independent.

To establish the converse, suppose that $x = \theta x^1 + (1 - \theta)x^2$ for some scalar $0 < \theta < 1$ and two feasible points x^1 and x^2 to the linear program (C.1). Since $x_j = \theta x_j^1 + (1 - \theta)x_j^2$ for any $j \geq k + 1$, and both x_j^1 and x_j^2 are nonnegative, both x_j^1 and x_j^2 must have value zero. Therefore, since both these points are feasible solutions to the linear program, they satisfy

$$\mathcal{A}_1x_1^1 + \mathcal{A}_2x_2^1 + \cdots + \mathcal{A}_kx_k^1 = b,$$

and

$$\mathcal{A}_1x_1^2 + \mathcal{A}_2x_2^2 + \cdots + \mathcal{A}_kx_k^2 = b.$$

Subtracting these equations from each other shows that

$$\mathcal{A}_1(x_1^1 - x_1^2) + \mathcal{A}_2(x_2^1 - x_2^2) + \cdots + \mathcal{A}_k(x_k^1 - x_k^2) = 0.$$

But since the columns $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_k$ are linearly independent, each component of x^1 and x^2 must be the same. But then we cannot represent $x = \theta x^1 + (1 - \theta)x^2$ in terms of two distinct feasible points of the linear program and therefore x is an extreme point. ◆

Although we have not stated this theorem as "x is a basic feasible solution if and only if it is an extreme point," the theorem easily implies this statement. Notice that if $k = p$ in this proof, the columns $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_k$ form a basis of the linear program, so x is a basic feasible solution. On the other hand, if $k < p$, we can add

$p - k$ other linearly independent columns to $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_k$ to form a basis $\mathcal{B}$. In this case x still is a basic feasible solution, but with some basic variables at value zero (those corresponding to the columns we have added). Therefore, these observations and Theorem C.1 imply that basic feasible solutions and extreme points are identical.

C.4 SIMPLEX METHOD

The simplex method maintains a basic feasible solution at every step. Given a basic feasible solution, the method first applies the optimality criteria to test the optimality of the current solution. If the current solution does not fulfill this condition, the algorithm performs an operation, known as a *pivot operation*, to obtain another basis structure with a lower or identical cost. The simplex method repeats this process until the current basic feasible solution satisfies the optimality criteria. Before describing the details of these steps, we consider some preliminary issues.

Obtaining an Initial Basis Structure

Recall from our preceding discussion that a basic solution might not be feasible. Therefore, identifying a basic feasible solution requires specifying a set of basic variables so that the columns associated with these variables are linearly independent and the solution x_B, x_L obtained by setting $x_B = \mathcal{B}^{-1}b$ and $x_L = 0$ is nonnegative. Unfortunately, there is no simple method for identifying any such collection of basic variables. In fact, finding a feasible solution of a linear program (or a basic feasible solution) is almost as difficult as finding an optimal solution. Nevertheless, using a simple technique, we can find a basic feasible solution of a related linear program and use it to initiate the simplex method. This technique consists of introducing an *artificial variable* x_{q+i} with a sufficiently large cost M for each constraint i and defining an augmented linear program as:

$$\text{Minimize} \quad \sum_{j=1}^{q} c_j x_j + \sum_{j=q+1}^{q+p} M x_j \tag{C.11a}$$

subject to

$$\sum_{j=1}^{q} a_{ij} x_j + x_{q+i} = b(i) \qquad \text{for all } i = 1, \ldots, p, \tag{C.11b}$$

$$x_j \geq 0 \qquad \text{for all } j = 1, \ldots, q. \tag{C.11c}$$

It is easy to show that the original linear program (C.1) has a feasible solution if and only if each artificial variable has value zero in every optimal solution of the augmented linear program (C.11). (The large cost coefficient associated with each artificial variable ensures that it has value zero in an optimal solution if the original problem has a feasible solution.) As a consequence, solving the augmented linear program solves the original linear program. For the augmented linear program, the set of all artificial variables constitutes an initial basis and the value of the ith basic variable is $b(i) \geq 0$.

Optimality Criteria

Let (**B**, **L**) denote a feasible basis structure of the linear program. Assume, for simplicity, that $\mathbf{B} = \{1, 2, \ldots, p\}$. Consider the canonical form associated with this basis structure. In this canonical form the objective function is

$$\text{Minimize} \quad z(x) = z_0 + \sum_{j=p+1}^{q} c_j^{\pi} x_j. \tag{C.12}$$

The coefficient $c_j^{\pi} = c_j - \sum_{i=1}^{p} \pi(i) a_{ij}$ is the reduced cost of the nonbasic variable x_j with respect to the current simplex multipliers π. We claim that if $c_j^{\pi} \geq 0$ for each nonbasic variables x_j, the current basic feasible solution x is an optimal solution of the linear program. To see this, observe that in any feasible solution of the linear program, $x_j \geq 0$ for all $j \in \mathbf{L}$. Therefore, if $c_j^{\pi} \geq 0$ for each nonbasic variables x_j, then z_0 is a lower bound on the optimal objective value. As we noted in the preceding section, the current solution x, which sets $x_j = 0$ for all $j \in \mathbf{L}$, achieves this lower bound and, therefore it must be optimal.

Pivot Operation

If $c_j^{\pi} < 0$ for some nonbasic variable x_j, the current basic feasible solution might not be optimal. The expression (C.12) implies that c_j^{π} is the rate of decrease in the objective function value per unit increase in the value of x_j. The simplex method selects one such nonbasic variable, say x_s, as the *entering variable* and tries to increase its value. As we will see, when the simplex method increases x_s as much as possible while keeping all the other nonbasic variable at value zero, some basic variable, say x_r, reaches value zero. The simplex method replaces the basic variable x_r by x_s, defining a new basis structure. It then updates the inverse of the basis and repeats the computations.

If we increase the value of the entering variable x_s to value θ and keep all other nonbasic variables at zero value, expression (C.8) implies that the basic variables $x_{\mathbf{B}}$ change in the following manner:

$$x_{\mathbf{B}} + \theta \bar{\mathcal{A}}_s = \bar{b}. \tag{C.13}$$

In this expression $\bar{b} = \mathcal{B}^{-1} b \geq 0$, $\bar{\mathcal{A}}_s = \mathcal{B}^{-1} \mathcal{A}_s$, and $x_{\mathbf{B}} = [x_1, x_2, \ldots, x_p]$. Let $\bar{\mathcal{A}}_s = [\bar{a}_{1s}, \bar{a}_{2s}, \ldots, \bar{a}_{ps}]$. We can restate (C.13) as

$$x_i = \bar{b}(i) - \theta \bar{a}_{is} \qquad \text{for all } i = 1, \ldots, p. \tag{C.14}$$

If $\bar{a}_{is} \leq 0$ and we increase θ, then x_i either remains unchanged or increases. If $\bar{a}_{is} > 0$ and we increase θ, then x_i decreases and eventually it will become zero. Consequently, (C.14) implies that if $\bar{a}_{is} > 0$, then the scalar θ must satisfy the condition $\theta \leq \bar{b}(i)/\bar{a}_{is}$ in order for x_i to remain nonnegative. As a result,

$$\theta = \min_{1 \leq i \leq p} \{\bar{b}(i)/\bar{a}_{is} : \bar{a}_{is} > 0\},$$

is the largest value of θ that we can assign to x_s while remaining feasible. What if $\bar{a}_{is} \leq 0$ for each $i = 1, \ldots, p$? Then we can assign an arbitrarily large value to the entering variable x_s and the solution remains feasible. Since $c_s^{\pi} < 0$, by setting x_s as large as we like, we can make the objective function arbitrarily small, and make

it approach $-\infty$. In this instance we say that the linear program has an *unbounded solution.*

We next focus on situations in which θ is finite. If we set $x_s = \theta$, one of the basic variables, say x_r, becomes zero. Note that $\bar{b}(r)/\bar{a}_{rs} = \theta$. We refer to x_r as the *leaving variable* and refer to the rule we have described for identifying θ and the corresponding leaving variable as the *minimum ratio rule.* Next, we designate x_s a basic variable, x_r a nonbasic variable, and update the canonical form of the linear program so that it satisfies the canonical property with respect to the new basis. We do so by performing a sequence of elementary row operations.

Updating the Simplex Tableau

Recall from Section C.3 that in the canonical form with respect to the basis **B**, the equations of the linear program assume the form

$$x_{\mathbf{B}} + \mathcal{B}^{-1}\mathcal{L}x_{\mathbf{L}} = \mathcal{B}^{-1}b. \tag{C.15}$$

In the new basis, the entering variable x_s becomes the basic variable for the rth row, which requires that in the new canonical form the variable x_s should have a coefficient $+1$ in the rth row, and a coefficient 0 in all other rows. We achieve this new canonical form by first dividing the rth row by $\bar{a}_{rs}$; the variable x_s then has a $+1$ coefficient in this row. Then, for each $1 \leq i \leq p$, $i \neq r$, we multiply the rth row by the constant $-\bar{a}_{is}$ and add it to the ith row so that the updated value of $\bar{a}_{is}$ becomes zero. We also multiply the rth row by a constant $-c_s^{\pi}$ and add it to the objective function so that the objective function coefficient of x_s becomes zero. We refer to this set of computations as a *pivot operation.*

To illustrate these steps of the simplex method, consider the linear programming example given in (C.6). For convenience, we state the example again.

$$\text{Minimize} \quad z(x) = 0x_1 + 0x_2 + 3x_3 - x_4 + 10 \tag{C.16a}$$

subject to

$$x_1 \qquad -3x_3 + 3x_4 = 6, \tag{C.16b}$$

$$x_2 - 8x_3 + 4x_4 = 4, \tag{C.16c}$$

$$x_1, x_2, x_3, x_4 \geq 0. \tag{C.16d}$$

The nonbasic variable x_4 has a negative reduced cost, and we select it as the entering variable. Applying the minimum ratio rule, we find that $\theta = \min\{6/3, 4/4\} = 1$, the minimum being achieved in the second row, which contains the basic variable x_2. The variable x_2 is the leaving variable. In the next basis, x_1 and x_4 are the basic variables, so we need to modify the canonical form. Since x_4 is the new basic variable for the constraint (C.16c), we divide this constraint by 4 so that x_4 has a $+1$ coefficient. We then multiply the modified constraint by -3 and add it to (C.16b), and multiply it by $+1$ and add it to (C.16a). These operations produce the following (equivalent) formulation of the linear program:

$$\text{Minimize} \quad z(x) = 0x_1 + 0x_4 + \tfrac{1}{4}x_2 + x_3 + 9$$

subject to

$$
\begin{aligned}
x_1 \qquad & -\tfrac{3}{4}x_2 + 3x_3 = 3, \\
x_4 + \tfrac{1}{4}x_2 & - 2x_3 = 1, \\
x_1, x_2, x_3, x_4 & \geq 0.
\end{aligned}
$$

In this canonical form, the reduced costs of both nonbasic variables x_2 and x_3 are nonnegative, so the current basic feasible solution, $x_1 = 3$, $x_4 = 1$, and $x_2 = x_3 = 0$, is optimal. This solution has an objective function value of 9.

One way to perform the pivot operation is by updating the full matrix $\bar{\mathcal{A}}$, that is, by performing the explicit set of pivot computations iteratively on the matrix $\mathcal{A}$. This set of computations can be very expensive for linear programs that contain many variables (as is typical in practice). The *revised simplex method* is a particular implementation of the simplex method that permits us to avoid many of these computations. To describe the basic approach of the revised simplex method, suppose that we (conceptually) append a set of fictitious variables y to the original linear program and form a new linear program with the constraints:

$$
\begin{aligned}
\mathcal{A}x + \mathcal{I}y &= b, \\
x \geq 0, \quad y &\geq 0.
\end{aligned}
$$

In this formulation, $\mathcal{I}$ is an identity matrix; that is, its diagonal elements all have value 1 and its elements off the diagonal all have value zero. Then to obtain the canonical form (C.13) with respect to the basis $\mathcal{B}$, we premultiply this system by the basis inverse $\mathcal{B}^{-1}$. With the fictitious variables y, the system becomes

$$
x_{\mathbf{B}} + \mathcal{B}^{-1}\mathcal{L}x_{\mathbf{L}} + \mathcal{B}^{-1}y = \mathcal{B}^{-1}b.
$$

As shown by this expression, if we were to perform the pivot operation on the entire matrix $\mathcal{A}$ from step to step, the coefficients of fictitious variables y would be the basis inverse. This observation shows that we need not carry out the pivot operations on the entire matrix $\mathcal{A}$. Instead, we can perform these operations on the columns associated with the initial identity matrix $\mathcal{I}$ (we do not formally introduce the variables y). Since the resulting computations give us the basis inverse $\mathcal{B}^{-1}$, we can use this matrix to compute the simplex multipliers $\pi = c_{\mathbf{B}}\mathcal{B}^{-1}$ and then use them to compute the reduced cost of each variable. Once we have determined the variable x_s to introduce into the basis, we compute its representation $\bar{\mathcal{A}}_s = \mathcal{B}^{-1}\mathcal{A}_s$ and then use this information to perform the ratio test to identify the variable x_r to leave the basis. We next perform the row elementary operations on the current $\mathcal{B}^{-1}$ matrix and obtain the updated basis inverse.

The advantage of this approach is that we use only the original data $\mathcal{A}$ in computing the reduced costs [using the formula $c_j^{\pi} = c_j - \sum_{i=1}^{p} \pi(i)a_{ij}$] and determine the modified data $\bar{\mathcal{A}}$ for only one column s of $\mathcal{A}$ and perform row elementary operations only on the basis inverse matrix. This apparently modest change in the algorithm often has dramatic effects on its efficiency because the original data $\mathcal{A}$ for most problems met in practice is very sparse in the sense that most (90 percent or more) of its coefficients are zero. On the other hand, the modified matrix $\bar{\mathcal{A}}$ typically becomes very dense as we perform the iterations of the simplex algorithm. In the revised simplex method, by using appropriate data structures, we can avoid all the

computations corresponding to the zero elements. As a consequence, by implementing the revised simplex method, we usually achieve great economies in our computations. The vast linear programming literature and many standard texts specify more details about this approach and about other ways to improve the empirical efficiency of the simplex method.

Finite Termination

The simplex method moves from one basic feasible solution to another by performing a pivot operation. In each iteration, the solution value improves by the amount $c_s^{\pi}\theta$. If $\theta > 0$, we say that the pivot is *nondegenerate*; otherwise, we refer to it as *degenerate*. A nondegenerate pivot strictly decreases the value of the objective function associated with the basic feasible solution. If every pivot is nondegenerate, the simplex method will terminate finitely because a linear program has at most ${}^{q}C_{p}$ distinct basic feasible solutions and the algorithm can never repeat any basic feasible solution (since the objective function value is strictly decreasing). However, the simplex method might perform degenerate pivots, and without further modifications it might repeat a basic feasible solution and therefore not terminate finitely. Fortunately, researchers have developed several ways for implementing the simplex method so that it converges finitely. We refer the reader to linear programming textbooks for a discussion of these techniques.

The simplex method terminates with one of the following three outcomes:

Case 1. The method terminates with an unbounded solution. In this case the linear program has no optimal solution with a finite solution value.

Case 2. The method terminates with a finite solution in which some artificial variable has a positive value. In this case the original linear program has no feasible solution.

Case 3. The method terminates with a finite solution in which all the artificial variables have zero value. This solution is an optimal solution of the original linear program.

C.5 BOUNDED VARIABLE SIMPLEX METHOD

Often linear programming problems have upper as well as lower bounds imposed on the variables. Network flow problems with arc capacities on the arc flows belong to this class. We refer to this class of linear programs, formulated as follows, as bounded variable linear programs.

$$\text{Minimize} \quad \sum_{j=1}^{q} c_j x_j \tag{C.17a}$$

subject to

$$\sum_{j=1}^{q} a_{ij} x_j = b(i) \qquad \text{for all } i = 1, \ldots, p, \tag{C.17b}$$

$$x_j \leq u_j \qquad \text{for all } j = 1, \ldots, q, \tag{C.17c}$$

$$x_j \geq 0 \qquad \text{for all } j = 1, \ldots, q. \tag{C.17d}$$

The simplest way to handle the upper bound constraints (C.17c) is to treat them like the other constraints (by adding slack variables to convert them into equality constraints). However, doing so increases the size of the linear program substantially and is therefore undesirable. In this section we describe a generalization of the simplex method, called the *bounded variable simplex method*, that treats these upper bound constraints implicitly, very much like the lower bound constraints (C.17d).

In our previous discussion we defined a basic feasible solution in the simplex method by a pair (**B**, **L**), consisting of a set **B** of basic variables and a set **L** of nonbasic variables at their lower bounds. We define a basic feasible solution in the bounded variable simplex method by a triplet (**B**, **L**, **U**), consisting of a set **B** of basic variables, a set **L** of nonbasic variables at their lower bounds, and a set **U** of nonbasic variables at their upper bounds. Let $\mathcal{B}$, $\mathcal{L}$, and $\mathcal{U}$ denote a compatible partitioning of the constraint matrix **A** corresponding to the sets **B**, **L**, and **U**. Then

$$\mathcal{B}x_{\mathbf{B}} + \mathcal{L}x_{\mathbf{L}} + \mathcal{U}x_{\mathbf{U}} = b.$$

In a basic feasible solution we set the value of each nonbasic variable to its appropriate bound, depending on whether the variable is contained in **L** or **U**. Let $u_{\mathbf{U}}$ denote the vector of upper bounds for variables in **U**. Then, setting $x_{\mathbf{L}} = 0$ and $x_{\mathbf{U}} = u_{\mathbf{U}}$, we find that

$$\mathcal{B}x_{\mathbf{B}} = b - \mathcal{U}u_{\mathbf{U}},$$

or

$$x_{\mathbf{B}} = \mathcal{B}^{-1}b - \mathcal{B}^{-1}[\mathcal{U}u_{\mathbf{U}}].$$

As before, $\mathcal{B}^{-1}$ is the inverse of the basis $\mathcal{B}$. For the bounded variable simplex method, we define the simplex multipliers π in the same way as in the simplex method (i.e., as the multipliers to apply to the constraints to create the reduced cost of zero for each basic variable). We now discuss, one by one, various steps of the simplex method and point out any changes required in the bounded variable simplex method.

Optimality Criteria

In a canonical form of the linear program with respect to the basis **B**, we can write the objective function as

$$z(x) = \sum_{j \in \mathbf{L}} c_j^{\pi} x_j + \sum_{j \in \mathbf{U}} c_j^{\pi} x_j + z_0.$$

Using arguments similar to those used earlier for the case without upper bounds, we can easily show that if $c_j^{\pi} \geq 0$ for each $j \in \mathbf{L}$ and $c_j^{\pi} \leq 0$ for each $j \in \mathbf{U}$, then $z_0 + \sum_{j \in \mathbf{U}} c_j^{\pi} u_j$ is a lower bound on the optimal value of the objective function. We refer to these conditions as the *optimality* conditions for bounded variable linear programs. If the basic feasible solution satisfies these optimality conditions, the objective function achieves this lower bound, and hence it must be optimal. Observe that the optimality conditions imply that it is not cost-effective to

increase the value of any nonbasic variable at its lower bound or decrease the value of any nonbasic variable at its upper bound.

Entering Arc Criteria

The bounded variable simplex method selects any nonbasic variable violating its optimality condition as the entering arc. In other words, a nonbasic variable x_j is eligible to be an entering arc if (1) $j \in \mathbf{L}$ and $c_j^\pi < 0$, or (2) $j \in \mathbf{U}$ and $c_j^\pi > 0$.

Leaving Arc Criteria

If the entering variable x_s is a nonbasic variable at its lower bound, the method attempts to increase the value of x_s by the largest possible amount, and if the entering variable x_s is at its upper bound, the method attempts to decrease its value by the largest possible amount. The maximum change is determined by the requirement that the value of every basic variable and of the nonbasic variable x_s remains between (or at) its lower and upper bounds. The expression (C.14) implies that if $\bar{a}_{is} < 0$, then by increasing θ, we increase the value of the basic variable x_i. In this case the maximum change allowed by the upper bound constraint of x_i is $\theta_i = (u_i - \bar{b}(i))/(-\bar{a}_{is})$. If $\bar{a}_{is} > 0$, then by increasing θ, we decrease the value of the basic variable x_i. In this case the maximum change allowed by the lower bound constraint of x_i is $\theta_i = \bar{b}(i)/\bar{a}_{is}$. We let $\theta_i = +\infty$ if $\bar{a}_{is} = 0$. Finally, the maximum change allowed by the upper and lower bound constraints of x_s is $|u_s|$. The maximum value of θ that we can assign to x_s will be the minimum of $|u_s|$ and $\min\{\theta_i : 1 \le i \le p\}$. In the former case the basis remains unchanged; x_s simply moves from the set **L** to **U** or from the set **U** to **L**. In the latter case the basis changes and so does the canonical form. The method of updating the canonical form is same as for the case without upper bounds. Conceivably, we might find that $\theta = \infty$, in which case the linear programming problem has an unbounded solution. The case, when the entering variable x_s is a nonbasic variable at its upper bound, is left as an exercise to the reader.

Termination of the Method

Under the nondegeneracy assumption, the bounded variable simplex method terminates finitely because it strictly decreases the objective function value of the basic feasible solution at each step and any linear program has a finite number of basic feasible solutions. Without the nondegeneracy assumption, we need some additional technique to ensure finite termination of the method. For a discussion of these techniques, we refer the reader to linear programming textbooks.

C.6 LINEAR PROGRAMMING DUALITY

Each linear programming problem, which for the purposes of this discussion we call the *primal problem*, has a closely related associated linear programming problem, called the *dual problem*, and together these two problems define a duality theory that lies at the heart of linear programming and many other areas in the field of constrained optimization. In this section we review briefly some of the most salient features of duality theory.

While discussing duality theory, we assume that the linear program has been stated in the following form:

$$\text{Minimize} \quad \sum_{j=1}^{q} c_j x_j \tag{C.18a}$$

subject to

$$\sum_{j=1}^{q} a_{ij} x_j \geq b(i) \qquad \text{for all } i = 1, \ldots, p, \tag{C.18b}$$

$$x_j \geq 0 \qquad \text{for all } j = 1, \ldots, q. \tag{C.18c}$$

In this formulation, we permit $b(i)$ to have an arbitrary sign. We refer to this form as the *symmetric form* of a linear program. It is possible to show that we can convert any linear program in a nonsymmetric form into the symmetric form. To define the dual problem, we associate a dual variable $\pi(i)$ with the ith constraint in (C.18b). With respect to these variables, the dual problem is:

$$\text{Maximize} \quad \sum_{i=1}^{p} b(i)\pi(i) \tag{C.19a}$$

subject to

$$\sum_{i=1}^{p} a_{ij}\pi(i) \leq c_j \qquad \text{for all } j = 1, \ldots, q, \tag{C.19b}$$

$$\pi(i) \geq 0 \qquad \text{for all } i = 1, \ldots, p. \tag{C.19c}$$

Notice that each constraint in the primal has an associated variable in the dual and that the right-hand side of this constraint becomes the cost coefficient of the associated variable. Moreover, each variable in the primal has an associated constraint in the dual and the cost coefficient of this variable becomes the right-hand side of the associated constraint. It is easy to verify that the dual of the dual is the primal.

In the linear programming problem, whose dual we want to form, if some constraint $\sum_{j=1}^{q} a_{ij}x_j = b(i)$ is in the equality form, we form the dual exactly as described earlier, except that the dual variable $\pi(i)$ becomes unrestricted in sign. Similarly, in the primal, if some primal variable x_i is unrestricted in sign, then in the dual the constraint corresponding to x_i is an equality constraint. Our first result concerning the primal–dual pair is known as the *weak duality theorem.*

Theorem C.2 (Weak Duality Theorem). *If x is any feasible solution of the primal problem and π is any feasible solution of the dual problem, then $\sum_{i=1}^{p} b(i)\pi(i) \leq \sum_{j=1}^{q} c_j x_j$.*

Proof. Multiplying the ith constraint in (C.18b) by $\pi(i)$ and adding yields

$$\sum_{i=1}^{p} b(i)\pi(i) \leq \sum_{i=1}^{p} \pi(i) \left[\sum_{j=1}^{q} a_{ij} x_j \right], \tag{C.20}$$

while multiplying the jth constraint in (C.19b) by x_j and adding yields

$$\sum_{j=1}^{q} x_j \left[\sum_{i=1}^{p} a_{ij}\pi(i) \right] \leq \sum_{j=1}^{q} c_j x_j. \tag{C.21}$$

Since the right-hand side of (C.20) equals the left-hand side of (C.21), together these two constraints imply that

$$\sum_{i=1}^{p} b(i)\pi(i) \leq \sum_{j=1}^{q} c_j x_j,$$

which is the conclusion of the lemma. ◆

The weak duality theorem has a number of immediate consequences, which we state next.

Property C.3

(*a*) *The objective function value of any feasible dual solution is a lower bound on the objective function value of every feasible primal solution.*

(*b*) *If the primal problem has an unbounded solution, the dual problem is infeasible.*

(*c*) *If the dual problem has an unbounded solution, the primal problem is infeasible.*

(*d*) *If the primal problem has a feasible solution x and the dual problem has a feasible solution π and $\sum_{i=1}^{p} b(i)\pi(i) = \sum_{j=1}^{q} c_j x_j$, then x is an optimal solution of the primal problem and π is an optimal solution of the dual problem.*

Theorem C.4 (Strong Duality Theorem). *If any one of the pair of primal and dual problems has a finite optimal solution, so does the other one and both have the same objective function values.*

Proof. Assume without any loss of generality that the primal problem has a finite optimal solution. We can convert the primal problem into standard form (in which all constraints have equalities instead of inequalities) by adding slack variables. Suppose that we apply the simplex method to this standard form and let (**B**, **L**) be the optimal basis structure. Let x be the optimal solution and π be the simplex multipliers associated with the optimal basis structure. Recall from our previous discussion that the simplex multipliers π are the multipliers associated with the constraints (C.1b), which when subtracted from the original form of the objective function yield the objective function in the final canonical form. Therefore,

$$z(x^*) = \sum_{j \in \mathbf{B}} c_j^{\pi} x_j + \sum_{j \in \mathbf{L}} c_j^{\pi} x_j + \sum_{i=1}^{p} b(i)\pi(i),$$

and

$$c_j^{\pi} = c_j - \sum_{i=1}^{p} a_{ij}\pi(i). \tag{C.22}$$

Since x is a basic feasible solution, $c_j^{\pi} = 0$ for all $j \in \mathbf{B}$, and $x_j = 0$ for all $j \in \mathbf{L}$. Consequently, $z(x^*) = \sum_{i=1}^{p} b(i)\pi(i)$. Moreover, since the basis structure (**B**, **L**) satisfies the optimality criteria (i.e., $c_j^{\pi} \geq 0$ for all $j \in \mathbf{B} \cup \mathbf{L}$), the expressions (C.22) and (C.19b) imply that π is a feasible solution to the dual problem. The objective function value of this dual solution is $\sum_{i=1}^{p} b(i)\pi(i)$, which is same as that

of the primal solution x. Property C.3(d) implies that π is an optimal dual solution, completing the proof of the theorem. ◆

The weak and strong duality theorems imply several fundamental results concerning relationships between the primal and dual problems. The following complementary slackness property, which is another way of relating the two problems, makes some of these relationships more explicit.

Complementary Slackness Property. *A pair (x, π) of the primal and dual feasible solutions is said to satisfy the complementary slackness property if*

$$\pi(i)\left[\sum_{j=1}^{q} a_{ij}x_j - b(i)\right] = 0 \qquad \text{for each } i = 1, \ldots, p, \tag{C.23a}$$

and

$$x_j\left[c_j - \sum_{i=1}^{p} a_{ij}\pi(i)\right] = 0 \qquad \text{for each } j = 1, \ldots, q. \tag{C.23b}$$

Observe from the formulation (C.18) of the primal problem that $[\sum_{j=1}^{q} a_{ij}x_j - b(i)]$ is the amount of slack in the ith primal constraint and $\pi(i)$ is the dual variable associated with this constraint. Similarly, $[c_j - \sum_{i=1}^{p} a_{ij}\pi(i)]$ is the amount of slack in the jth dual constraint and x_j is the primal variable associated with this constraint. The complementary slackness property states that for every primal and dual constraint, the product of the slack in the constraint and its associated primal or dual variable is zero. In other words, if a constraint has a positive slack, the associated primal or dual variable must have value zero, and alternatively, if a primal or dual variable has a positive value, the dual solution must satisfy the corresponding constraint as an equality.

Theorem C.5 (Complementary Slackness Optimality Conditions). *A primal feasible solution x and a dual feasible solution π are optimal solutions of the primal and dual problems if and only if they satisfy the complementary slackness property.*

Proof. We first prove that if x and π are optimal primal and dual solutions, they must satisfy the complementary slackness property. While proving the weak duality theorem, we observed that

$$\sum_{i=1}^{p} b(i)\pi(i) \leq \sum_{i=1}^{p}\sum_{j=1}^{q} a_{ij}\pi(i)x_j \leq \sum_{j=1}^{q} c_j x_j. \tag{C.24}$$

Since x and π are optimal primal and dual solutions, the strong duality theorem implies that both of the inequalities in (C.24) must be satisfied as equalities. We can rewrite the first equality in (C.23) as

$$\sum_{i=1}^{p} \pi(i)\left[\sum_{j=1}^{q} a_{ij}x_j - b(i)\right] = 0. \tag{C.25}$$

Since x and π are feasible in the primal and dual problems, each term in (C.25)

is nonnegative. Consequently, the sum of these terms can be zero only if each term is individually zero. This observation shows that the solutions satisfy (C.23a). Similarly, the second equality in (C.24) implies that the solutions satisfy (C.23b).

We next prove the converse result: namely, if the solutions x and π satisfy the complementary slackness property, they must be optimal in the primal and dual problems. Adding (C.23a) for all i yields

$$\sum_{j=1}^{q} \sum_{i=1}^{p} a_{ij}\pi(i)x_j = \sum_{i=1}^{p} b(i)\pi(i).$$

Similarly, adding (C.23b) for all j yields

$$\sum_{j=1}^{q} \sum_{i=1}^{p} a_{ij}\pi(i)x_j = \sum_{j=1}^{q} c_j x_j.$$

Therefore, $\sum_{i=1}^{p} b(i)\pi(i) = \sum_{j=1}^{q} c_j x_j$, and by Property C.3(c), x is an optimal primal solution and π is an optimal dual solution, concluding the proof of the theorem. ◆

REFERENCE NOTES

Dantzig, who conducted the pioneering work in linear programming, developed the simplex method in 1947 to solve several military planning problems. The optimization community introduced a steady stream of developments since then and now the theory of linear programming includes a vast body of knowledge. Books by Dantzig [1962], Bradley, Hax, and Magnanti [1977], Chvátal [1983], Schrijver [1986], and Winston [1991] are excellent references on the history, applications, and theory of this topic.

REFERENCES

AASHTIANI, H. A., and T. L. MAGNANTI. 1976. Implementing primal–dual network flow algorithms. Technical Report OR 055-76, Operations Research Center, MIT, Cambridge, MA.

ABDALLAOUI, G. 1987. Maintainability of a grade structure as a transportation problem. *Journal of the Operational Research Society* **38,** 367–369.

ADEL'SON-VEL'SKI, G. M., E. A. DINIC, and E. V. KARZANOV. 1975. *Flow Algorithms.* Science, Moscow. (In Russian.)

AHLFELD, D. P., R. S. DEMBO, J. M. MULVEY, and S. A. ZENIOS. 1987. Nonlinear programming on generalized networks. *ACM Transactions on Mathematical Software* **13,** 350–367.

AHO, A. V., J. E. HOPCROFT, and J. D. ULLMAN. 1974. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, MA.

AHO, A. V., J. E. HOPCROFT, and J. D. ULLMAN. 1983. *Data Structures and Algorithms.* Addison-Wesley, Reading, MA.

AHUJA, R. K. 1986. Algorithms for the minimax transportation problem. *Naval Research Logistics Quarterly* **33,** 725–740.

AHUJA, R. K., and J. B. ORLIN. 1989. A fast and simple algorithm for the maximum flow problem. *Operations Research* **37,** 748–759.

AHUJA, R. K., and J. B. ORLIN. 1991. Distance-directed augmenting path algorithms for maximum flow and parametric maximum flow problems. *Naval Research Logistics Quarterly* **38,** 413–430.

AHUJA, R. K., and J. B. ORLIN. 1992a. The scaling network simplex algorithm. *Operations Research* **40,** Supplement 1, S5–S13.

AHUJA, R. K., and J. B. ORLIN. 1992b. Use of representative counts in computational testings of algorithms. Sloan Working Paper, Sloan School of Management, MIT, Cambridge, MA.

AHUJA, R. K., J. L. BATRA, and S. K. GUPTA. 1984. A parametric algorithm for the convex cost network flow and related problems. *European Journal of Operational Research* **16,** 222–235.

AHUJA, R. K., A. V. GOLDBERG, J. B. ORLIN, and R. E. TARJAN. 1992. Finding minimum-cost flows by double scaling. *Mathematical Programming* **53,** 243–266.

AHUJA, R. K., M. KODIALAM, A. K. MISHRA, and J. B. ORLIN. 1992. Computational testing of maximum flow algorithms. Sloan Working Paper, Sloan School of Management, MIT, Cambridge, MA.

AHUJA, R. K., T. L. MAGNANTI, and J. B. ORLIN. 1989. Network flows. In *Handbooks in Operations Research and Management Science.* Vol. 1: *Optimization,* edited by G. L. Nemhauser, A. H. G. Rinnooy Kan, and M. J. Todd. North-Holland, Amsterdam, pp. 211–369.

AHUJA, R. K., T. L. MAGNANTI, and J. B. ORLIN. 1991. Some recent advances in network flows. *SIAM Review* **33,** 175–219.

AHUJA, R. K., T. L. MAGNANTI, J. B. ORLIN, and M. R. REDDY. 1992. Applications of network optimization. Sloan Working Paper, Sloan School of Management, MIT, Cambridge, MA.

AHUJA, R. K., K. MEHLHORN, J. B. ORLIN, and R. E. TARJAN. 1990. Faster algorithms for the shortest path problem. *Journal of ACM* **37,** 213–223.

AHUJA, R. K., J. B. ORLIN, C. STEIN, and R. E. TARJAN. 1990. Improved algorithms for bipartite network flow problems. Technical Report, Sloan School of Management, MIT, Cambridge, MA. Submitted to *SIAM Journal on Computing.*

AHUJA, R. K., J. B. ORLIN, and R. E. TARJAN. 1989. Improved time bounds for the maximum flow problem. *SIAM Journal on Computing* **18,** 939–954.

AKGÜL, M. 1985a. Shortest path and simplex method. Research Report, Department of Computer Science and Operations Research, North Carolina State University, Raleigh, NC.

AKGÜL, M. 1985b. A genuinely polynomial primal simplex algorithm for the assignment problem. Research Report, Department of Computer Science and Operations Research, North Carolina State University, Raleigh, NC.

ALI, A. I., E. P. ALLEN, R. S. BARR, and J. L. KENNINGTON. 1986. Reoptimization procedures for bounded variable primal simplex network algorithms. *European Journal of Operational Research* **23**, 256–263.

ALI, A. I., D. BARNETT, K. FARHANGIAN, J. L. KENNINGTON, B. PATTY, B. SHETTY, B. MCCARL, and P. WONG. 1984. Multicommodity network problems: Applications and computations. *IIE Transactions* **16**, 127–134.

ALI, A. I., R. V. HELGASON, and J. L. KENNINGTON. 1978. The convex cost network flow problem: A state-of-the-art survey. Technical Report OREM 78001, Southern Methodist University, Dallas, TX.

ALI, A. I., R. PADMAN, and H. THIAGARAJAN. 1989. Dual algorithms for pure network problems. *Operations Research* **37**, 159–171.

ALON, N. 1990. Generating pseudo-random permutations and maximum flow algorithms. *Information Processing Letters* **35**, 201–204.

ANDERSON, W. N. 1975. Maximum matching and the rank of a matrix. *SIAM Journal on Applied Mathematics* **28**, 114–123.

ARISAWA, S., and S. E. ELMAGHRABY. 1977. The "hub" and "wheel" scheduling problems. *Transportation Science* **11**, 124–146.

ARONSON, J. E. 1989. A survey of dynamic network flows. *Annals of Operations Research* **20**, 1–66.

ASSAD, A. A. 1978. Multicommodity network flows: A survey. *Networks* **8**, 37–91.

ASSAD, A. A. 1980a. Models for rail transportation. *Transportation Research* **14A**, 205–220.

ASSAD, A. A. 1980b. Solving linear multicommodity flow problems. *Proceedings of the IEEE International Conference on Circuits and Computers*, pp. 157–161.

BACHARACH, M. 1966. Matrix rounding problems. *Management Science* **9**, 732–742.

BALACHANDRAN, V., and G. L. THOMPSON. 1975. An operator theory of parametric programming for the generalized transportation problems. Parts I–IV. *Naval Research Logistics Quarterly* **22**, 79–125, 297–340.

BALAKRISHNAN, A., T. L. MAGNANTI, and R. T. WONG. 1989. A dual-ascent procedure for large scale uncapacitated network design. *Operations Research* **37**, 716–740.

BALAKRISHNAN, A., T. L. MAGNANTI, A. SHULMAN, and R. T. WONG. 1991. Models for capacity expansion in local access telecommunication networks. *Annals of Operations Research* **33**, 239–284.

BALAKRISHNAN, A., T. L. MAGNANTI, and R. T. WONG. 1991. A decomposition algorithm for local access telecommunications network expansion planning. Working Paper, Operations Research Center, MIT, Cambridge, MA.

BALINSKI, M. L. 1986. A competitive (dual) simplex method for the assignment problem. *Mathematical Programming* **34**, 125–141.

BALL, M. O., and U. DERIGS. 1983. An analysis of alternative strategies for implementing matching algorithms. *Networks* **13**, 517–549.

BARAHONA, F., and É. TARDOS. 1989. Note on Weintraub's minimum cost circulation algorithm. *SIAM Journal on Computing* **18**, 579–583.

BARNHART, C. 1988. A network-based primal–dual solution methodology for the multicommodity network flow problem. Ph.D. dissertation, Department of Civil Engineering, MIT, Cambridge, MA.

BARR, R. S., F. GLOVER, and D. KLINGMAN. 1977. The alternating path basis algorithm for the assignment problem. *Mathematical Programming* **13**, 1–13.

BARR, R. S., and J. S. TURNER. 1981. Microdata file merging through large scale network technology. *Mathematical Programming Study* **15**, 1–22.

BARROS, O., and A. WEINTRAUB. 1986. Spatial market equilibrium problems as network models. *Discrete Applied Mathematics* **13**, 109–130.

BARTHOLDI, J. J., J. B. ORLIN, and H. D. RATLIFF. 1980. Cyclic scheduling via integer programs with circular ones. *Operations Research* **28**, 1074–1085.

BARZILAI, J., W. D. COOK, and M. KRESS. 1986. A generalized network formulation of the pairwise comparison consensus ranking model. *Management Science* **32**, 1007–1014.

BAZARAA, M. S., J. J. JARVIS, and H. D. SHERALI. 1990. *Linear Programming and Network Flows*, 2nd ed. Wiley, New York.

BELFORD, P. C., and H. D. RATLIFF. 1972. A network-flow model for racially balancing schools. *Operations Research* **20,** 619–628.

BELLMAN, R. E. 1957. *Dynamic Programming.* Princeton University Press, Princeton, NJ.

BELLMAN, R. 1958. On a routing problem. *Quarterly of Applied Mathematics* **16,** 87–90.

BELLMORE, M., G. BENNINGTON, and S. LUBORE. 1971. A multivehicle tanker scheduling problem. *Transportation Science* **5,** 36–47.

BENNINGTON, G. E. 1974. Applying network analysis. *Industrial Engineering* **6,** 17–25.

BENTLEY, J. L. 1990. Experiments on geometric traveling salesman heuristics. Computing Science Technical Report 151, AT&T Bell Laboratories, Holmdel, NY.

BENTLEY, J. L., and B. W. KERNIGHAN. 1990. A system for algorithm animation: Tutorial and algorithm animation. Unix Research System Paper, 10th ed., Vol. II. Saunders College Publishing, Philadelphia, pp. 451–475.

BERGE, C. 1957. Two theorems in graph theory. *Proceedings of the National Academy of Sciences USA* **43,** 842–844.

BERGE, C., and A. GHOUILA-HOURI. 1962. *Programming, Games and Transportation Networks.* Wiley, New York.

BERRISFORD, H. G. 1960. The economic distribution of coal supplies in the gas industry: An application of the linear programming transport theory. *Operations Research Quarterly* **11,** 139–150.

BERTSEKAS, D. P. 1976. *Dynamic Programming and Stochastic Control.* Academic Press, New York.

BERTSEKAS, D. P. 1979. A distributed algorithm for the assignment problem. Working Paper, Laboratory for Information and Decision Systems, MIT, Cambridge, MA.

BERTSEKAS, D. P. 1988. The auction algorithm: A distributed relaxation method for the assignment problem. *Annals of Operations Research* **14,** 105–123.

BERTSEKAS, D. P., and D. E. BAZ. 1987. Distributed asynchronous relaxation methods for convex network flow problems. *SIAM Journal on Control and Optimization* **25,** 74–85.

BERTSEKAS, D. P., and J. ECKSTEIN. 1988. Dual coordinate step methods for linear network flow problems. *Mathematical Programming B* **42,** 203–243.

BERTSEKAS, D. P., P. A. HOSEIN, and P. TSENG. 1987. Relaxation methods for network flow problems with convex arc costs. *SIAM Journal on Control and Optimization* **25,** 1219–1243.

BERTSEKAS, D. P., and P. TSENG. 1988a. The relax codes for linear minimum cost network flow problems. In *FORTRAN Codes for Network Optimization,* edited by B. Simeone, P. Toth, G. Gallo, F. Maffioli, and S. Pallottino. *Annals of Operations Research* **13,** 125–190.

BERTSEKAS, D. P., and P. TSENG. 1988b. Relaxation methods for minimum cost ordinary and generalized network flow problems. *Operations Research* **36,** 93–114.

BERTSIMAS, D., and J. B. ORLIN. 1991. A technique for speeding up the solution of the Lagrangian dual. Working Paper OR 248-91, Operations Research Center, MIT, Cambridge, MA.

BIXBY, R. E. 1982. Matroids and operations research. In *Advanced Techniques in the Practice of Operations Research,* edited by H. J. Greenberg, F. H. Murphy, and S. H. Shaw. North-Holland, Amsterdam, pp. 433–458.

BIXBY, R. E. 1991. The simplex method: It keeps getting better. Presented at the *14th International Symposium on Mathematical Programming,* Amsterdam, The Netherlands.

BLAND, R. G., and D. L. JENSEN. 1992. On the computational behavior of a polynomial-time network flow algorithm. *Mathematical Programming* **54,** 1–39.

BOAS, P. V. E., R. KAAS, and E. ZIJLSTRA. 1977. Design and implementation of an efficient priority queue. *Mathematical Systems Theory* **10,** 99–127.

BODIN, L. D., B. L. GOLDEN, A. D. SCHUSTER, and W. ROWING. 1980. A model for the blockings of trains. *Transportation Research* **14B,** 115–120.

BODIN, L. D., B. L. GOLDEN, A. A. ASSAD, and M. O. BALL. 1983. Routing and scheduling of vehicles and crews: The state of the art. *Computers and Operations Research* **10,** 69–211.

BONDY, J. A., and U. S. R. MURTY. 1976. *Graph Theory with Applications.* American Elsevier, New York.

BORŮVKA, O. 1926, Příspevěk k řešení otázky ekonomické stavby elektrovodních sítí. *Elektrotechnicky Obzor* **15,** 153–154.

BRADLEY, G., G. BROWN, and G. GRAVES. 1977. Design and implementation of large scale primal transshipment algorithms. *Management Science* **21,** 1–38.

BRADLEY, S. P., A. C. HAX, and T. L. MAGNANTI. 1977. *Applied Mathematical Programming*. Addison-Wesley, Reading, MA.

BROGAN, W. L. 1989. Algorithm for ranked assignments with application to multiobject tracking. *Journal of Guidance*, 357–364.

BROWN, M. H. 1988. *Algorithm Animation*. MIT Press, Cambridge, MA.

BROWN, G. G., and R. D. MCBRIDE. 1984. Solving generalized networks. *Management Science* **30**, 1497–1523.

BRUYNOOGHE, M., A. GIBERT, and M. SAKAROVITCH. 1968. Une méthode d'affection du traffic. In: *Fourth International Symposium on the Theory of Traffic Flow*, Karlsruhe, 1968, W. Lentzback and P. Barons (eds.), Beiträge Theorie des Verkehrsflusses Strassenbau und Strassenkehrstechnik Heft 86, Herausgeben von Bunderesminister fur Verkehr, Abteilung Strassenbau, Bonn, Germany.

BUSAKER, R. G., and P. J. GOWEN. 1961. A procedure for determining minimal-cost network flow patterns. ORO Technical Report 15, Operational Research Office, Johns Hopkins University, Baltimore, MD.

BUSACKER, R. G., and T. L. SAATY. 1965. *Finite Graphs and Networks*. McGraw-Hill, New York.

CABOT, A. V., R. L. FRANCIS, and M. A. STARY. 1970. A network flow solution to a rectilinear distance facility location problem. *AIIE Transactions* **2**, 132–141.

CAHN, A. S. 1948. The warehouse problem (Abstract). *Bulletin of the American Mathematical Society* **54**, 1073.

CARPENTO, G., S. MARTELLO, and P. TOTH. 1988. Algorithms and codes for the assignment problem. In *FORTRAN Codes for Network Optimization*, edited by B. Simeone, P. Toth, G. Gallo, F. Maffioli, and S. Pallottino. *Annals of Operations Research* **13**, 193–224.

CARRARESI, P., and G. GALLO. 1984. Network models for vehicle and crew scheduling. *European Journal of Operational Research* **16**, 139–151.

CHALMET, L. G., R. L. FRANCIS, and P. B. SAUNDERS. 1982. Network models for building evacuation. *Management Science* **28**, 86–105.

CHANDRASEKARAN, R. 1977. Minimum ratio spanning trees. *Networks* **7**, 335–342.

CHANG, M. D., and C. J. CHEN. 1989. An improved primal simplex variant for pure processing networks. *ACM Transactions on Mathematical Software* **15**, 64–78.

CHARNES, A., and D. KLINGMAN. 1971. The "more for less" paradox in the distribution model. *Cahiers du Centre D'Etudes de Recherche Operationnelle* **13**, 11–22.

CHEN, H., and C. G. DEWALD. 1974. A generalized chain labeling algorithm for solving multicommodity flow problems. *Computers and Operations Research* **1**, 437–465.

CHENG, C. K., and T. C. HU. 1990. Ancestor tree for arbitrary multi-terminal cut functions. *Proceedings of a Conference on "Integer Programming and Combinatorial Optimization,"* edited by R. Kannan and W. R. Pulleyblank. University of Waterloo, Waterloo, Canada.

CHERIYAN, J., and T. HAGERUP. 1989. A randomized maximum-flow algorithm. *Proceedings of the 30th IEEE Conference on the Foundations of Computer Science*, pp. 118–123.

CHERIYAN, J., T. HAGERUP, and K. MEHLHORN. 1990. Can a maximum flow be computed in $O(nm)$ time? *Proceedings of the 17th International Colloquium on Automata, Languages and Programming*, pp. 235–248.

CHERIYAN, J., and S. N. MAHESHWARI. 1989. Analysis of preflow push algorithms for maximum network flow. *SIAM Journal on Computing* **18**, 1057–1086.

CHESHIRE, M., K. I. M. MCKINNON, and H. P. WILLIAMS. 1984. The efficient allocation of private contractors to public works. *Journal of the Operational Research Quarterly* **35**, 705–709.

CHIN, F., and D. HOUCH. 1978. Algorithms for updating spanning trees. *Journal of Computer and System Sciences* **16**, 333–344.

CHRISTOPHIDES, N. 1975. *Graph Theory: An Algorithmic Approach*. Academic Press, New York.

CHVÁTAL, V. 1983. *Linear Programming*. W. H. Freeman, New York.

CLARK, J. A., and N. A. J. HASTINGS. 1977. Decision networks. *Operational Research Quarterly* **20**, 51–68.

CLARKE, S., and J. SURKIS. 1968. An operations research approach to racial desegregation of school systems. *Socio-Economic Planning Sciences* **1**, 259–272.

COBHAM, A. 1964. The intrinsic computational difficulty of functions. *Proceedings of the 1964 Congress for Logic, Methodology, and the Philosophy of Science*, North-Holland, Amsterdam, pp. 24–30.

COLLINS, M., L. COOPER, R. HELGASON, J. KENNINGTON, and L. LEBLANC. 1978. Solving the pipe network analysis problem using optimization techniques. *Management Science* **24**, 747–760.

COOK, S. 1971. The complexity of theorem proving procedures. *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing*, pp. 151–158.

CORMEN, T. H., C. L. LEISERSON, and R. L. RIVEST. 1990. *Introduction to Algorithms*. MIT Press and McGraw-Hill, New York.

COX, L. H., and L. R. ERNST. 1982. Controlled rounding. *INFOR* **20**, 423–432.

CRAINIC, T., J. A. FERLAND, and J. M. ROUSSEAU. 1984. A tactical planning model for rail freight transportation. *Transportation Science* **18**, 165–184.

CREMEANS, J. E., R. A. SMITH, and G. R. TYNDALL. 1970. Optimal multicommodity network flows with resource allocation. *Naval Research Logistics Quarterly* **17**, 269–280.

CROWDER, H. P., R. S. DEMBO, and J. M. MULVEY. 1978. Reporting computational experiments in mathematical programming. *Mathematical Programming* **15**, 316–329.

CROWDER, H. P., R. S. DEMBO, and J. M. MULVEY. 1979. On reporting computational experiments with mathematical software. *ACM Transactions on Mathematical Software* **5**, 193–203.

CROWDER, H. P., and P. B. SAUNDERS. 1980. Results of a survey on MP performance indicators. *COAL Newsletter*, January, pp. 2–6.

CRUM, R. L., and D. J. NYE. 1981. A network model of insurance company cash flow management. *Mathematical Programming* **15**, 86–101.

CUNNINGHAM, W. H. 1976. A network simplex method. *Mathematical Programming* **11**, 105–116.

CUNNINGHAM, W. H. 1979. Theoretical properties of the network simplex method. *Mathematics of Operations Research* **4**, 196–208.

DAFERMOS, S., and A. NAGURNEY. 1984. A network formulation of market equilibrium problems and variational inequalities. *Operations Research Letters* **5**, 247–250.

DANIEL, R. C. 1973. Phasing out capital equipment. *Operations Research Quarterly* **24**, 113–116.

DANTZIG, G. B. 1951. Application of the simplex method to a transportation problem. In *Activity Analysis and Production and Allocation*, edited by T. C. Koopmans. Wiley, New York, pp. 359–373.

DANTZIG, G. B. 1960. On the shortest route through a network. *Management Science* **6**, 187–190.

DANTZIG, G. B. 1962. *Linear Programming and Extensions*. Princeton University Press, Princeton, NJ.

DANTZIG, G. B., W. BLATTNER, and M. R. RAO. 1966. Finding a cycle in a graph with minimum cost to time ratio with application to a ship routing problem. In *Theory of Graphs. International Symposium*. Dunod, Paris, and Gordon and Breach, New York, pp. 209–213.

DANTZIG, G. B., and D. R. FULKERSON. 1954. Minimizing the number of tankers to meet a fixed schedule. *Naval Research Logistics Quarterly* **1**, 217–222.

DANTZIG, G. B., and P. WOLFE. 1960. Decomposition principle for linear programs. *Operations Research* **8**, 101–111.

DANTZIG, G. B., and P. WOLFE. 1961. The decomposition method for linear programming. *Econometrica* **29**, 767–778.

DEARING, P. M., and R. L. FRANCIS. 1974. A network flow solution to a multifacility minimax location problem involving rectilinear distances. *Transportation Science* **8**, 126–141.

DEMBO, R. S., J. M. MULVEY, and S. A. ZENIOS. 1989. Large-scale nonlinear network models and their applications. *Operations Research* **37**, 353–372.

DENARDO, E. V. 1982. *Dynamic Programming: Models and Applications*. Prentice Hall, Englewood Cliffs, NJ.

DENARDO, E. V., and B. L. FOX. 1979. Shortest-route methods: 1. Reaching, pruning and buckets. *Operations Research* **27**, 161–186.

DENARDO, E. V., U. G. ROTHBLUM, and A. J. SWERSEY. 1988. A transportation problem in which costs depend on the order of arrival. *Management Science* **34**, 774–783.

DEO, N., and C. PANG. 1984. Shortest path algorithms: Taxonomy and annotation. *Networks* **14**, 275–323.

DERIGS, U. 1988. *Programming in Networks and Graphs*. Lecture Notes in Economics and Mathematical Systems, Vol. 300. Springer-Verlag, New York.

DERIGS, U., and W. MEIER. 1989. Implementing Goldberg's max-flow algorithm: A computational investigation. *Zeitschrift für Operations Research* **33**, 383–403.

DERMAN, C., and M. KLEIN. 1959. A note on the optimal depletion of inventory. *Management Science* **5**, 210–214.

DEVINE, M. V. 1973. A model for minimizing the cost of drilling dual completion oil wells. *Management Science* **20**, 532–535.

DEWAR, M. S. J., and H. C. LONGUET-HIGGINS. 1952. The correspondence between the resonance and molecular orbital theories. *Proceedings of the Royal Society of London* **A214**, 482–493.

DIAL, R. 1969. Algorithm 360: Shortest path forest with topological ordering. *Communications of ACM* **12**, 632–633.

DIAL, R., F. GLOVER, D. KARNEY, and D. KLINGMAN. 1979. A computational analysis of alternative algorithms and labeling techniques for finding shortest path trees. *Networks* **9**, 215–248.

DIJKSTRA, E. 1959. A note on two problems in connexion with graphs. *Numeriche Mathematics* **1**, 269–271.

DINIC, E. A. 1970. Algorithm for solution of a problem of maximum flow in networks with power estimation. *Soviet Mathematics Doklady* **11**, 1277–1280.

DINIC, E. A. 1973. The method of scaling and transportation problems. *Issled. Diskret. Mat. Science*, Moscow. (In Russian.)

DIRICKX, Y. M. I., and L. P. JENNERGREN. 1975. An analysis of the parking situation in the downtown area of West Berlin. *Transportation Research* **9**, 1–11.

DIVOKY, J. J., and M. S. HUNG. 1990. Performance of shortest path algorithms in network flow problems. *Management Science* **36**, 661–673.

DORSEY, R. C., T. J. HODGSON, and H. D. RATLIFF. 1974. A production scheduling problem with batch processing. *Operations Research* **22**, 1271–1279.

DORSEY, R. C., T. J. HODGSON, and H. D. RATLIFF. 1975. A network approach to a multi-facility, multi-product production scheduling problem without backordering. *Management Science* **21**, 813–822.

DRESS, A. W. M., and T. F. HAVEL. 1988. Shortest path problems and molecular conformation. *Discrete Applied Mathematics* **19**, 129–144.

DROR, M., P. TRUDEAU, and S. P. LADANY. 1988. Network models for seat allocation on flights. *Transportation Research* **22B**, 239–250.

DUDE, R. O., and P. E. HART. 1973. *Pattern Classification and Science Analysis*. Wiley, New York.

EDMONDS, J. 1965a. Paths, trees, and flowers. *Canadian Journal of Mathematics* **17**, 449–467.

EDMONDS, J. 1965b. Maximum matchings and a polyhedran with 0, 1 vertices. *Journal of Research of the National Bureau of Standards* **69B**, 125–130.

EDMONDS, J. 1965c. Minimum partition of a matroid into independent subsets. *Journal of Research of the National Bureau of Standards* **69B**, 67–72.

EDMONDS, J. 1967. An introduction to matching. Mimeographed notes, Engineering Summer Conference, The University of Michigan, Ann Arbor, MI.

EDMONDS, J. 1971. Matroids and the greedy algorithm. *Mathematical Programming* **1**, 127–136.

EDMONDS, J., and E. L. JOHNSON. 1973. Matching, Euler tours and the Chinese postman. *Mathematical Programming* **5**, 88–124.

EDMONDS, J., and R. M. KARP. 1972. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of ACM* **19**, 248–264.

ELAM, J., F. GLOVER, and D. KLINGMAN. 1979. A strongly convergent primal simplex algorithm for generalized networks. *Mathematics of Operations Research* **4**, 39–59.

ELIAS, P., A. FEINSTEIN, and C. E. SHANNON. 1956. Note on maximum flow through a network. *IRE Transactions on Information Theory* **IT-2**, 117–119.

ELMAGHRABY, S. E. 1978. *Activity Networks: Project Planning and Control by Network Models*. Wiley–Interscience, New York.

ERLENKOTTER, D. 1978. A dual-based procedure for uncapacitated facility location. *Operations Research* **26**, 992–1009.

ERVOLINA, T. R., and S. T. MCCORMICK. 1990a. Cancelling most helpful cuts for minimum cost network flow. Faculty of Commerce Working Paper 90-MSC-018, University of British Columbia, Vancouver, Canada.

ERVOLINA, T. R., and S. T. MCCORMICK. 1990b. Two strongly polynomial cut cancelling algorithms for minimum cost network flow. Technical Report, Faculty of Commerce and Business Administration, University of British Columbia, Vancouver, Canada.

ESAU, L. R., and K. C. WILLIAMS. 1966. On teleprocessing system design II. *IBM Systems Journal* **5**, 142–147.

EVANS, J. R. 1977. Some network flow models and heuristics for multiproduct production and inventory planning. *AIIE Transactions* **9**, 75–81.

EVANS, J. R. 1984. The factored transportation problem. *Management Science* **30**, 1021–1024.

EVEN, S. 1979. *Graph Algorithms*. Computer Science Press, Rockville, MD.

EVEN, S., and O. KARIV. 1975. An $O(n^{2.5})$ algorithm for maximum matching in general graphs. *Proceedings of the 16th Annual Symposium on Foundations of Computer Science*, pp. 100–112.

EVEN, S., and R. E. TARJAN. 1975. Network flow and testing graph connectivity. *SIAM Journal on Computing* **4**, 507–518.

EVERETT, H., III. 1963. Generalized Lagrange multiplier method for solving problems of optimal allocation of resources. *Operations Research* **11**, 399–417.

EWASHKO, T. A., and R. C. DUDDING. 1971. Application of Kuhn's Hungarian assignment algorithm to posting servicemen. *Operations Research* **19**, 991.

FARINA, R. F., and F. W. GLOVER. 1983. The application of generalized networks to choice of raw materials for fuels and petrochemicals. In *Energy Models and Studies*, edited by B. Lev. North-Holland, Amsterdam.

FARLEY, A. R. 1980. Levelling terrain trees: A transshipment problem. *Information Processing Letters* **10**, 189–192.

FARVOLDEN, J. M., and W. B. POWELL. 1990. A primal partitioning solution for multicommodity network flow problems. Working Paper 90-04, Department of Industrial Engineering, University of Toronto, Toronto, Canada.

FEDERGRUEN, A., and H. GROENEVELT. 1986. Preemptive scheduling of uniform machines by ordinary network flow techniques. *Management Science* **32**, 341–349.

FERNANDEZ-BACA, D., and C. U. MARTEL. 1989. On the efficiency of maximum flow algorithms on networks with small integer capacities. *Algorithmica* **4**, 173–189.

FILLIBEN, J. J., K. KAFADAR, and D. R. SHIER. 1983. Testing for homogeneity of two-dimensional surfaces. *Mathematical Modelling* **4**, 167–189.

FISHER, M. L. 1981. The Lagrangian relaxation methods for solving integer programming problems. *Management Science* **27**, 1–18.

FISHER, M. L. 1985. An applications oriented guide to Lagrangian relaxation. *Interfaces* **15**, 10–21.

FLORIAN, M. 1986. Nonlinear cost network models in transportation analysis. *Mathematical Programming Study* **26**, 167–196.

FLOYD, R. W. 1962. Algorithm 97: Shortest path. *Communications of ACM* **5**, 345.

FORD, L. R. 1956. Network flow theory. Report P-923, Rand Corp., Santa Monica, CA.

FORD, L. R., and D. R. FULKERSON. 1956a. Maximal flow through a network. *Canadian Journal of Mathematics* **8**, 399–404.

FORD, L. R., and D. R. FULKERSON. 1956b. Solving the transportation problem. *Management Science* **3**, 24–32.

FORD, L. R., and D. R. FULKERSON. 1957. A primal–dual algorithm for the capacitated Hitchcock problem. *Naval Research Logistics Quarterly* **4**, 47–54.

FORD, L. R., and D. R. FULKERSON. 1958a. Constructing maximum dynamic flows from static flows. *Operations Research* **6**, 419–433.

FORD, L. R., and D. R. FULKERSON. 1958b. A suggested computation for maximal multicommodity network flows. *Management Science* **5**, 97–101.

FORD, L. R., and D. R. FULKERSON. 1962. *Flows in Networks*. Princeton University Press, Princeton, NJ.

FORD, L. R., and S. M. JOHNSON. 1959. A tournament problem. *The American Mathematical Monthly* **66**, 387–389.

FRANCIS, R. L., and J. A. WHITE. 1976. *Facility Layout and Location*. Prentice Hall, Englewood Cliffs, NJ.

FRANK, C. R. 1965. A note on the assortment problem. *Management Science* **11**, 724–726.

FRANK, H., and I. T. FRISCH. 1971. *Communication, Transmission, and Transportation Networks*. Addison-Wesley, Reading, MA.

FREDMAN, M. L., and R. E. TARJAN. 1984. Fibonacci heaps and their uses in improved network optimization algorithms. *Proceedings of the 25th Annual IEEE Symposium on Foundations of Computer Science*, pp. 338–346. Full paper in *Journal of ACM* **34**(1987), 596–615.

FUJII, M., T. KASAMI, and K. NINOMIYA. 1969. Optimal sequencing of two equivalent processors. *SIAM Journal on Applied Mathematics* **17**, 784–789. Erratum, same journal **18**, 141.

FUJISHIGE, S. 1986. A capacity-rounding algorithm for the minimum cost circulation problem: A dual framework of Tardos' algorithm. *Mathematical Programming* **35**, 298–308.

FULKERSON, D. R. 1961a. A network flow computation for project cost curve. *Management Science* **7**, 167–178.

FULKERSON, D. R. 1961b. An out-of-kilter method for minimal cost flow problems. *SIAM Journal on Applied Mathematics* **9**, 18–27.

FULKERSON, D. R. 1963. Flows in networks. In *Recent Advances in Mathematical Programming*, edited by R. L. Graves and P. Wolfe. McGraw-Hill, New York, pp. 319–332.

FULKERSON, D. R. 1965. Upsets in a round robin tournament. *Canadian Journal of Mathematics* **17**, 957–969.

FULKERSON, D. R. 1966. Flow networks and combinatorial operations research. *American Mathematical Monthly* **73**, 115–138.

FULKERSON, D. R., and G. B. DANTZIG. 1955. Computation of maximum flow in networks. *Naval Research Logistics Quarterly* **2**, 277–283.

FULKERSON, D. R., and G. C. HARDING. 1977. Maximizing the minimum source–sink path subject to a budget constraint. *Mathematical Programming* **13**, 116–118.

GABOW, H. N. 1975. An efficient implementation of Edmond's algorithm for maximum matchings on graphs. *Journal of ACM* **23**, 221–234.

GABOW, H. N. 1985. Scaling algorithms for network problems. *Journal of Computer and System Sciences* **31**, 148–168.

GABOW, H. N. 1990. Data structures for weighted matching and nearest common ancestors with linking. *Proceedings of the First Annual ACM–SIAM Symposium on Discrete Algorithms*. SIAM, Philadelphia, pp. 434–443.

GABOW, H. N., Z. GALIL, T. SPENCER, and R. E. TARJAN. 1986. Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. *Combinatorica* **6**, 109–122.

GABOW, H. N., and R. E. TARJAN. 1989a. Faster scaling algorithms for network problems. *SIAM Journal on Computing* **18**, 1013–1036.

GABOW, H. N., and R. E. TARJAN. 1989b. Faster scaling algorithms for general graph matching problems. Technical Report CU-CS-432-89, Department of Computer Science, University of Colorado, Boulder, CO.

GALE, D. 1957. A theorem on flows in networks. *Pacific Journal of Mathematics* **7**, 1073–1082.

GALE, D., and L. S. SHAPLEY. 1962. College admissions and the stability of marriage. *American Mathematical Monthly* **69**, 9–14.

GALIL, Z. 1981. On the theoretical efficiency of various network flow algorithms. *Theoretical Computer Science* **14**, 103–111.

GALIL, Z., and É. TARDOS. 1986. An $O(n^2(m + n \log n) \log n)$ min-cost flow algorithm. *Proceedings of the 27th Annual Symposium on the Foundations of Computer Science*, pp. 136–146. Full paper in *Journal of ACM* **35**(1987), 374–386.

GALLO, G., M. D. GRIGORIADIS, and R. E. TARJAN. 1989. A fast parametric maximum flow algorithm and applications. *SIAM Journal on Computing* **18**, 30–55.

GALLO, G., and S. PALLOTTINO. 1984. Shortest path methods in transportation models. In *Transportation Planning Models*, edited by M. Florian. Elsevier/North-Holland, Amsterdam.

GALLO, G., and S. PALLOTTINO. 1986. Shortest path methods: A unifying approach. *Mathematical Programming Study* **26**, 38–64.

GALLO, G., and S. PALLOTTINO. 1988. Shortest path algorithms. In *Fortran Codes for Network Optimization*, edited by B. Simeone, P. Toth, G. Gallo, F. Maffioli, and S. Pallottino. *Annals of Operations Research* **13**, 3–79.

GAREY, M. S., and D. S. JOHNSON. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, New York.

GAVISH, B. 1985. Augmented Lagrangian based algorithms for centralized network design. *IEEE Transactions on Communications* **COM-33**, 1247–1257.

GAVISH, B., and P. SCHWEITZER. 1974. An algorithm for combining truck trips. *Transportation Science* **8**, 13–23.

GAVISH, B., and K. N. SRIKANTH. 1979. $O(n^2)$ algorithms for sensitivity analysis of minimal spanning trees and related subgraphs. Working Paper 8003, Graduate School of Management, University of Rochester, Rochester, NY.

GEOFFRION, A. 1974. Lagrangian relaxations for integer programming. *Mathematical Programming Study* **2**, 82–114.

GEOFFRION, A. M., and G. W. GRAVES. 1974. Multicommodity distribution system design by Benders decomposition. *Management Science* **20,** 822–844.

GERSHT, A., and A. SHULMAN. 1987. A new algorithm for the solution of the minimum cost multicommodity flow problem. *Proceedings of the IEEE Conference on Decision and Control* **26,** 748–758.

GILMORE, P. C., and R. E. GOMORY. 1964. Sequencing a one state-variable machine: A solvable case of the travelling salesman problem. *Operations Research* **12,** 655–679.

GLOVER, F., R. GLOVER, and F. K. MARTINSON. 1984. A netform system for resource planning in the U.S. Bureau of Land Management. *Journal of the Operational Research Society* **35,** 605–616.

GLOVER, F., R. GLOVER, and D. J. SHIELDS. 1988. Microcomputer-based model of international mineral market. In *Operational Research '87*, edited by G. K. Rand. Elsevier, Amsterdam.

GLOVER, F., J. HULTZ, D. KLINGMAN, and J. STUTZ. 1978. Generalized networks: A fundamental computer based planning tool. *Management Science* **24,** 1209–1220.

GLOVER, F., D. KARNEY, and D. KLINGMAN. 1974. Implementation and computational comparisons of primal, dual and primal–dual computer codes for minimum cost network flow problem. *Networks* **4,** 191–212.

GLOVER, F., D. KARNEY, D. KLINGMAN, and A. NAPIER. 1974. A computational study on start procedures, basis change criteria, and solution algorithms for transportation problem. *Management Science* **20,** 793–813.

GLOVER, F., and D. KLINGMAN. 1977. Network applications in industry and government. *AIIE Transactions* **9,** 363–376.

GLOVER, F., D. KLINGMAN, J. MOTE, and D. WHITMAN. 1984. A primal simplex variant for the maximum flow problem. *Naval Research Logistics Quarterly* **31,** 41–61.

GLOVER, F., D. KLINGMAN, and N. PHILLIPS. 1985. A new polynomially bounded shortest path algorithm. *Operations Research* **33,** 65–73.

GLOVER, F., D. KLINGMAN, and N. PHILLIPS. 1990. Netform modeling and applications. *Interfaces* **20,** 7–27.

GLOVER, F., D. KLINGMAN, N. PHILLIPS, and R. F. SCHNEIDER. 1985. New polynomial shortest path algorithms and their computational attributes. *Management Science* **31,** 1106–1128.

GLOVER, F., and J. ROGOZINSKI. 1982. Resort development: A network-related model for optimizing sites and visits. *Journal of Leisure Research*, 235–247.

GOETSCHALCKX, M., and H. D. RATLIFF. 1988. Order picking in an aisle. *IIE Transactions* **20,** 53–62.

GOLDBERG, A. V. 1985. A new max-flow algorithm. Technical Report MIT/LCS/TM-291, Laboratory for Computer Science, MIT, Cambridge, MA.

GOLDBERG, A. V., M. D. GRIGORIADIS, and R. E. TARJAN. 1988. Efficiency of the network simplex algorithm for the maximum flow problem. Technical Report, Department of Computer Science, Stanford University, Stanford, CA.

GOLDBERG, A. V., S. A. PLOTKIN, and É. TARDOS. 1991. Combinatorial algorithms for the generalized circulation problem. *Mathematics of Operations Research* **16,** 351–381.

GOLDBERG, A. V., É. TARDOS, and R. E. TARJAN. 1989. Network flow algorithms. Technical Report 860, School of Operations Research and Industrial Engineering, Cornell University, Ithaca, NY.

GOLDBERG, A. V., and R. E. TARJAN. 1986. A new approach to the maximum flow problem. *Proceedings of the 18th ACM Symposium on the Theory of Computing*, pp. 136–146. Full paper in *Journal of ACM* **35**(1988), 921–940.

GOLDBERG, A. V., and R. E. TARJAN. 1987. Solving minimum cost flow problem by successive approximation. *Proceedings of the 19th ACM Symposium on the Theory of Computing*, pp. 7–18. Full paper in *Mathematics of Operations Research* **15**(1990), 430–466.

GOLDBERG, A. V., and R. E. TARJAN. 1988. Finding minimum-cost circulations by cancelling negative cycles. *Proceedings of the 20th ACM Symposium on the Theory of Computing*, pp. 388–397. Full paper in *Journal of ACM* **36**(1989), 873–886.

GOLDEN, B. L. 1975. A minimum cost multicommodity network flow problem concerning imports and exports. *Networks* **5,** 331–356.

GOLDEN, B. L., A. A. ASSAD, E. A. WASIL, and E. BAKER. 1986. Experiments in optimization. Working Paper Series MS/S 86-004, University of Maryland, College Park, MD.

GOLDEN, B. L., M. LIBERATORE, and C. LIEBERMAN. 1979. Models and solution techniques for cash flow management. *Computers and Operations Research* **6,** 13–20.

GOLDEN, B. L., and T. L. MAGNANTI. 1977. Deterministic network optimization: A bibliography. *Networks* **7,** 149–183.

GOLDFARB, D. 1985. Efficient dual simplex algorithms for the assignment problem. *Mathematical Programming* **33,** 187–203.

GOLDFARB, D., and J. HAO. 1988. Polynomial-time primal simplex algorithms for the minimum cost network flow problem. Technical Report, Department of Industrial Engineering and Operations Research, Columbia University, New York.

GOLDFARB, D., and J. HAO. 1990. A primal simplex algorithm that solves the maximum flow problem in at most nm pivots and $O(n^2m)$ time. *Mathematical Programming* **47,** 353–365.

GOLDFARB, D., J. HAO, and S. KAI. 1990a. Efficient shortest path simplex algorithms. *Operations Research* **38,** 624–628.

GOLDFARB, D., J. HAO, and S. KAI. 1990b. Anti-stalling pivot rules for the network simplex algorithm. *Networks* **20,** 79–91.

GOLDMAN, A. J., and G. L. NEMHAUSER. 1967. A transport improvement problem transformable to a best-path problem. *Transportation Science* **1,** 295–307.

GOLITSCHEK, M. V., and H. SCHNEIDER. 1984. Applications of shortest path algorithms to matrix scalings. *Numerische Mathematik* **44,** 111–126.

GOMORY, R. E., and T. C. HU. 1961. Multi-terminal network flows. *Journal of SIAM* **9,** 551–570.

GONDRAN, M., and M. MINOUX. 1984. *Graphs and Algorithms*. Wiley-Interscience, New York.

GORHAM, W. 1963. An application of a network flow model to personnel planning. *IEEE Transactions on Engineering Management* **10,** 113–123.

GOWER, J. C., and G. J. S. ROSS. 1969. Minimum spanning trees and single linkage cluster analysis. *Applied Statistics* **18,** 54–64.

GRAHAM, R. L., and P. HELL. 1985. On the history of minimum spanning tree problem. *Annals of the History of Computing* **7,** 43–57.

GRAVES, S. C. 1982. Using Lagrangian techniques to solve hierarchical production planning problems. *Management Science* **28,** 260–275.

GRAVES, G. W., and R. D. MCBRIDE. 1976. The factorization approach to large scale linear programming. *Mathematical Programming* **10,** 91–110.

GREENBERG, H. 1990. Computational testing: Why, how, and how much. *ORSA Journal of Computing* **2,** 94–97.

GRIGORIADIS, M. D. 1986. An efficient implementation of the network simplex method. *Mathematical Programming Study* **26,** 83–111.

GRIGORIADIS, M. D., and Y. HSU. 1979. The Rutgers minimum cost network flow subroutines. *SIGMAP Bulletin of the ACM* **26,** 17–18.

GRÖTSCHEL, M., and O. HOLLAND. 1985. Solving matching problems with linear programming. *Mathematical Programming* **33,** 243–259.

GUIGNARD, M., and S. KIM. 1987a. Lagrangian decomposition: A model yielding stronger Lagrangian bounds. *Mathematical Programming* **39,** 215–228.

GUIGNARD, M., and S. KIM. 1987b. Lagrangian decomposition for integer programming: Theory and applications. Technical Report 93, Department of Statistics, The Wharton School, University of Pennsylvania, Philadelphia, PA.

GUPTA, S. K. 1985. *Linear Programming and Network Models*. Affiliated East–West Press, New Delhi, India.

GUSFIELD, D. 1988. A graph theoretic approach to statistical data security. *SIAM Journal on Computing* **17,** 552–571.

GUSFIELD, D. 1990. Very simple methods for all pairs network flow analysis. *SIAM Journal on Computing* **19,** 143–155.

GUSFIELD, D., and R. W. IRVING. 1989. *The Stable Marriage Problem: Structure and Algorithms*. MIT Press, Cambridge, MA.

GUSFIELD, D., and C. MARTEL. 1989. A fast algorithm for the generalized parametric minimum cut problem and applications. Technical Report CSE-89-21, Computer Science Division, University of California, Davis, CA.

GUSFIELD, D., C. MARTEL, and D. FERNANDEZ-BACA. 1987. Fast algorithms for bipartite network flow. *SIAM Journal on Computing* **16,** 237–251.

GUTJAHR, A. L., and G. L. NEMHAUSER. 1964. An algorithm for the line balancing problem. *Management Science* **11,** 308–315.

HALL, M. 1956. An algorithm for distinct representatives. *American Mathematical Monthly* **63,** 716–717.

HAMACHER, H. W., and S. TUFEKCI. 1987. On the use of lexicographic min cost flows in evacuation modelling. *Naval Research Logistics Quarterly* **34**, 487–504.

HANDLER, G. Y. 1973. Minimax location of a facility in an undirected graph. *Transportation Science* **7**, 287–293.

HANDLER, G., and I. ZANG. 1980. A dual algorithm for the constrained shortest path problem. *Networks* **10**, 293–309.

HASSIN, R. 1981. Maximum flow in (s, t)-planar networks. *Information Processing Letters* **13**, 107.

HASSIN, R., and D. B. JOHNSON. 1985. An $O(n \log^2 n)$ algorithm for maximum flow in undirected planar networks. *SIAM Journal on Computing* **14**, 612–624.

HAUSMAN, H. 1978. *Integer Programming and Related Areas: A Classified Bibliography.* Lecture Notes in Economics and Mathematical Systems, Vol. 160. Springer-Verlag, Berlin.

HAX, A. C., and C. CANDEA. 1984. *Production and Inventory Management.* Prentice Hall, Englewood Cliffs, NJ.

HAYMOND, R. E., J. P. JARVIS, and D. R. SHIER. 1980. Computational methods for minimum spanning tree problems. Technical Report 354, Department of Mathematical Sciences, Clemson University, Clemson, SC.

HAYMOND, R. E., J. R. THORNTON, and D. D. WARNER. 1988. A shortest path algorithm in robotics and its implementation on the FPS T-20 hypercube. *Annals of Operations Research* **14**, 305–320.

HELD, M., and R. KARP. 1970. The traveling salesman problem and minimum spanning trees. *Operations Research* **18**, 1138–1162.

HELD, M., and R. KARP. 1971. The traveling salesman problem and minimum spanning trees, Part II. *Mathematical Programming* **6**, 62–88.

HELGASON, R. V., J. L. KENNINGTON, and B. D. STEWART. 1988. Dijkstra's two-tree shortest path algorithm. Technical Report, Department of Operations Research and Engineering Management, Southern Methodist University, Dallas, TX.

HITCHCOCK, F. L. 1941. The distribution of a product from several sources to numerous facilities. *Journal of Mathematical Physics* **20**, 224–230.

HOCHBAUM, D. S., and J. G. SHANTHIKUMAR. 1990. Convex separable optimization is not much harder than linear optimization. *Journal of ACM* **37**, 843–862.

HOFFMAN. A. J. 1960. Some recent applications of the theory of linear inequalities to extremal combinatorial analysis. In *Combinatorial Analysis*, edited by R. Bellman and M. Hall. American Mathematical Society, Providence, RI, pp. 113–128.

HOFFMAN, K. L., and R. H. F. JACKSON. 1982. In pursuit of a methodology for testing mathematical programming software. In *Evaluating Mathematical Programming Techniques*, Lecture Notes in Economics and Mathematical Systems, Vol. 199, edited by J. M. Mulvey et al., Springer-Verlag, New York.

HOFFMAN, A. J., and J. B. KRUSKAL. 1956. Integral boundary points of convex polyhedra. In *Linear Inequalities and Related Systems*, edited by H. W. Kuhn and A. W. Tucker. Princeton University Press, Princeton, NJ, pp. 233–246.

HOFFMAN, A. J., and H. M. MARKOWITZ. 1963. A note on shortest path, assignment and transportation problems. *Naval Research Logistics Quarterly* **10**, 375–379.

HOFFMAN, A. J., and S. T. MCCORMICK. 1984. A fast algorithm that makes matrices optimally sparse. In *Progress in Combinatorial Optimization.* Academic Press Canada, Don Mills, Ontario, Canada.

HOPCROFT, J. E., and R. M. KARP. 1973. A $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing* **2**, 225–231.

HOPCROFT, J. E., and R. E. TARJAN. 1974. Efficient planarity testing. *Journal of ACM* **21**, 549–568.

HORN, W. A. 1971. Determining optimal container inventory and routing. *Transportation Science* **5**, 225–231.

HORN, W. A. 1973. Minimizing average flow time with parallel machines. *Operations Research* **21**, 846–847.

HU, T. C. 1961. The maximum capacity route problem. *Operations Research* **9**, 898–900.

HU, T. C. 1963. Multi-commodity network flows. *Operations Research* **11**, 344–360.

HU, T. C. 1966. Minimum cost flows in convex cost networks. *Naval Research Logistics Quarterly* **13**, 1–9.

HU, T. C. 1967. Laplace's equation and network flows. *Operations Research* **15**, 348–354.

HU, T. C. 1969. *Integer Programming and Network Flows.* Addison-Wesley, Reading, MA.

HU, T. C. 1974. Optimum communication spanning trees. *SIAM Journal on Computing* **3,** 188–195.

HUNG, M. S. 1983. A polynomial simplex method for the assignment problem. *Operations Research* **31,** 595–600.

HUNG, M. S., and J. J. DIVOKY. 1988. A computational study of efficient shortest path algorithms. *Computers and Operations Research* **15,** 567–576.

IMAI, H. 1983. On the practical efficiency of various maximum flow algorithms. *Journal of the Operations Research Society of Japan* **26,** 61–82.

IMAI, H., and M. IRI. 1986. Computational-geometric methods for polygonal approximations of a curve. *Computer Vision, Graphics and Image Processing* **36,** 31–41.

IRI, M. 1960. A new method of solving transportation-network problems. *Journal of the Operations Research Society of Japan* **3,** 27–87.

IRI, M. 1969. *Network Flow, Transportation and Scheduling*. Academic Press, New York.

ITAI, A., and Y. SHILOACH. 1979. Maximum flow in planar networks. *SIAM Journal on Computing* **8,** 135–150.

JACKSON, R. H. B., P. T. BOGGS, S. G. NASH, and S. POWELL. 1989. Report of the ad hoc committee to revise the guidelines for reporting computational experiments in mathematical programming. *COAL Newsletter* **18,** 3–14.

JACKSON, R. H. B., and J. M. MULVEY. 1978. A critical review of comparisons of mathematical programming algorithms and software (1953–1977). *Journal of Research of the National Bureau of Standards* **83,** 563–584.

JACOBS, W. W. 1954. The caterer problem. *Naval Research Logistics Quarterly* **1,** 154–165.

JARNÍCK, V. 1930. O jistém problému minimálním. *Acta Societatis Scientiarum Natur. Moravicae* **6,** 57–63.

JARVIS, J. P., and D. E. WHITE. 1983. Computational experience with minimum spanning tree algorithms. *Operations Research Letters* **2,** 36–41.

JENSEN, P. A., and W. BARNES. 1980. *Network Flow Programming*. Wiley, New York.

JENSEN, P., and G. BHAUMIK. 1977. A flow augmentation approach to the network with gains minimum cost flow problem. *Management Science* **23,** 631–643.

JEWELL, W. S. 1957. Warehousing and distribution of a seasonal product. *Naval Research Logistics Quarterly* **4,** 29–34.

JEWELL, W. S. 1958. Optimal flow through networks. Interim Technical Report 8, Operations Research Center, MIT, Cambridge, MA.

JEWELL, W. S. 1962. Optimal flow through networks with gains. *Operations Research* **10,** 476–499.

JOHNSON, E. L. 1966. Networks and basic solutions. *Operations Research* **14,** 619–624.

JOHNSON, T. B. 1968. Optimum pit mine production scheduling. Technical Report, University of California, Berkeley, CA.

JOHNSON, D. B. 1982. A priority queue in which initialization and queue operations take $O(\log \log D)$ time. *Mathematical Systems Theory* **15,** 295–309.

JOHNSON, D. S. 1990. Local optimization and the traveling salesman problem. *Proceedings of the 17th Colloquium on Automata, Languages, and Programming*. Springer-Verlag, New York, pp. 446–461.

JOHNSON, D. B., and S. VENKATESAN. 1982. Using divide and conquer to find flows in directed planar networks in $O(n^{3/2} \log n)$ time. *Proceedings of the 20th Annual Allerton Conference on Communication, Control, and Computing*, University of Illinois, Urbana-Champaign, IL, pp. 898–905.

KAMEDA, T., and I. MUNRO. 1974. A $O(|V| \cdot |E|)$ algorithm for maximum matching of graphs. *Computing* **12,** 91–98.

KANG, A. N. C., R. C. T. LEE, C. L. CHANG, and S. K. CHANG. 1977. Storage reduction through minimal spanning trees and spanning forests. *IEEE Transactions on Computers* **C-26,** 425–434.

KANTOROVICH, L. V. 1939. Mathematical methods in the organization and planning of production. Publication House of the Leningrad University. Translated in *Management Science* 6(1960), 366–422.

KAPLAN, S. 1973. Readiness and the optimal redeployment of resources. *Naval Research Logistics Quarterly* **20,** 625–638.

KARP, R. M. 1972. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, edited by R. E. Miller and J. W. Thacher. Plenum Press, New York, pp. 83–103.

KARP, R. M. 1978. A characterization of the minimum cycle mean in a diagraph. *Discrete Mathematics* **23,** 309–311.

KARP, R. M., and J. B. ORLIN. 1981. Parametric shortest path algorithms with an application to cyclic staffing. *Discrete Applied Mathematics* **3**, 37–45.

KARZANOV, A. V. 1974. Determining the maximal flow in a network by the method of preflows. *Soviet Mathematics Doklady* **15**, 434–437.

KASTNING, C. 1976. *Integer Programming and Related Areas: A Classified Bibliography*. Lecture Notes in Economics and Mathematical Systems, Vol. 128, Springer-Verlag, Berlin.

KELLY, J. R. 1961. Critical path planning and scheduling: Mathematical basis. *Operations Research* **9**, 296–320.

KELLY, J. P., B. L. GOLDEN, and A. A. ASSAD. 1992. Cell suppression: Disclosure protection for sensitive tabular data. *Networks* **22**, 397–412.

KENNINGTON, J. L. 1978. A survey of linear cost multicommodity network flows. *Operations Research* **26**, 209–236.

KENNINGTON, J. L., and R. V. HELGASON. 1980. *Algorithms for Network Programming*. Wiley-Interscience, New York.

KENNINGTON, J. L., and M. SHALABY. 1977. An effective subgradient procedure for minimal cost multicommodity flow problems. *Management Science* **23**, 994–1004.

KENNINGTON, J. L., and Z. WANG. 1990. The shortest augmenting path algorithm for the transportation problem. Technical Report 90-CSE-10, Southern Methodist University, Dallas, TX.

KHAN, M. R. 1979. A capacitated network formulation for manpower scheduling. *Industrial Management* **21**, 24–28.

KHAN, M. R., and D. A. LEWIS. 1987. A network model for nursing staff scheduling. *Zeitschrift für Operations Research* **31**, B161–B171.

KLEIN, M. 1967. A primal method for minimal cost flows with application to the assignment and transportation problems. *Management Science* **14**, 205–220.

KLINCEWICZ, J. G. 1983. A Newton method for convex separable network flow problems. *Networks* **13**, 427–442.

KLINGMAN, D., A. NAPIER, and J. STUTZ. 1974. NETGEN: A program for generating large scale capacitated assignment, transportation, and minimum cost flow network problems. *Management Science* **20**, 814–821.

KNUTH, D. E. 1973a. *The Art of Computer Programming*. Vol. 1: *Fundamental Algorithms*, 2nd ed. Addison-Wesley, Reading, MA.

KNUTH, D. E. 1973b. *The Art of Computer Programming*. Vol. III: *Sorting and Searching*. Addison-Wesley, Reading, MA.

KOLITZ, S. 1991. Personal communication.

KOOPMANS, T. C. 1947. Optimum utilization of the transportation system. *Proceedings of the International Statistical Conference*, Washington, DC. Also in *Econometrica* **17**(1949).

KORTE, B. 1988. Applications of combinatorial optimization. Technical Report 88541-OR, Institute für Okonometrie und Operations Research, Bonn, Germany.

KOURTZ, P. 1984. A network approach to least cost daily transfers of forest fire control resources. *INFOR* **22**, 283–290.

KRUSKAL, J. B. 1956. On the shortest spanning tree of graph and the traveling salesman problem. *Proceedings of the American Mathematical Society* **7**, 48–50.

KUHN, H. W. 1955. The Hungarian method for the assignment problem. *Naval Research Logistics Quarterly* **2**, 83–97.

LAPORTE, G., and Y. NOBERT. 1987. Exact algorithms for the vehicle routing problem. In *Surveys in Combinatorial Optimization*, edited by S. Martello, G. Laporte, M. Minoux, and C. Ribeiro. North-Holland, Amsterdam.

LARSON, R. C., and A. R. ODONI. 1981. *Urban Operations Research*. Prentice Hall, Englewood Cliffs, NJ.

LAWANIA, A. K. 1990. Personal communication.

LAWLER, E. L. 1964. On scheduling problems with deferral costs. *Management Science* **11**, 280–287.

LAWLER, E. L. 1966. Optimal cycles in doubly weighted linear graphs. In *Theory of Graphs: International Symposium*, Dunod, Paris, and Gordon and Breach, New York, pp. 209–213.

LAWLER, E. L. 1976. *Combinatorial Optimization: Networks and Matroids*. Holt, Rinehart and Winston, New York.

LAWLER, E. L., J. K. LENSTRA, A. H. G. RINNOOY KAN, and D. B. SHMOYS (eds.). 1985. *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization.* Wiley, New York.

LEUNG, J., T. L. MAGNANTI, and V. SINGHAL. 1990. Routing in point to point delivery systems. *Transportation Science* **24,** 245–260.

LEVIN, L. A. 1973. Universal sorting problems. *Problemy Peredachi Informatsii* **9,** 265–266. (In Russian.)

LEVNER, E. V., and A. S. NEMIROVSKY. 1991. A network flow algorithm for just-in-time project scheduling. Memorandum COSOR 91-21, Department of Mathematics and Computing Science, Eindhoven University of Technology, Eindhoven, The Netherlands.

LIN, T. F. 1986. A system of linear equations related to the transportation problem with application to probability theory. *Discrete Applied Mathematics* **14,** 47–56.

LOBERMAN, H., and A. WEINBERGER. 1957. Formal procedures for connecting terminals with a minimum total wire length. *Journal of ACM* **4,** 428–437.

LOVÁSZ, L., and M. D. PLUMMER. 1986. *Matching Theory.* North-Holland, Amsterdam.

LOVE, R. R., and R. R. VEMUGANTI. 1978. The single-plant mold allocation problem with capacity and changeover restriction. *Operations Research* **26,** 159–165.

LOWE, T. J., R. L. FRANCIS, and E. W. REINHARDT. 1979. A greedy network flow algorithm for a warehouse leasing problem. *AIIE Transactions* **11,** 170–182.

LUSS, H. 1979. A capacity expansion model for two facilities. *Naval Research Logistics Quarterly* **26,** 291–303.

MACHOL, R. E. 1961. An application of the assignment problem. *Operations Research* **9,** 585–586.

MACHOL, R. E. 1970. An application of the assignment problem. *Operations Research* **18,** 745–746.

MAGNANTI, T. L. 1981. Combinatorial optimization and vehicle fleet planning: Perspectives and prospects. *Networks* **11,** 179–214.

MAGNANTI, T. L. 1984. Models and algorithms for predicting urban traffic equilibria. In *Transportation Planning Models*, edited by M. Florian. North-Holland, Amsterdam, pp. 153–186.

MAGNANTI, T. L., P. MIRCHANDANI, and R. VACHANI. 1991. Modeling and solving the capacitated network loading problem. Working Paper, Operations Research Center, MIT, Cambridge, MA.

MAGNANTI, T. L., J. SHAPIRO, and M. WAGNER. 1976. Generalized linear programming solves the dual. *Management Science* **22,** 1195–1203.

MAGNANTI, T. L., L. A. WOLSEY, and R. T. WONG. 1992. Optimal Trees. To appear in *Handbooks in Operations Research and Management Science*. Vol. 6: *Networks*, edited by M. Ball, T. L. Magnanti, C. L. Monma, and G. L. Nemhauser. North-Holland, Amsterdam.

MAGNANTI, T. L., L. A. WOLSEY, and R. T. WONG. 1992. Network design. To appear in *Handbooks in Operations Research and Management Science*, Vol. 6: *Networks*, edited by M. Ball, T. L. Magnanti, C. L. Monma, and G. L. Nemhauser. North-Holland, Amsterdam.

MAGNANTI, T. L., and R. T. WONG. 1984. Network design and transportation planning: Models and algorithms. *Transportation Science* **18,** 1–55.

MALIK, K., A. K. MITTAL, and S. K. GUPTA. 1989. The k most vital arcs in the shortest path problem. *Operations Research Letters* **8,** 223–227. Erratum: Same journal **9**(1990) 283.

MAMER, J. W., and S. A. SMITH. 1982. Optimizing field repair kits based on job completion rate. *Management Science* **28,** 1328–1334.

MANNE, A. S. 1958. A target-assignment problem. *Operations Research* **6,** 346–351.

MANSOUR, Y., and B. SCHIEBER. 1988. Finding the edge connectivity of directed graphs. Research Report RC 13556, IBM Thomas J. Watson Research Center, Yorktown Heights, NY.

MARTEL, C. 1982. Preemptive scheduling with release times, deadlines, and due times. *Journal of ACM* **29,** 812–829.

MARTELLO, S., W. R. PULLEYBLANK, P. TOTH, and D. DE WERRA. 1984. Balanced optimization problems. *Operations Research Letters* **3,** 275–278.

MASON, A. J., and A. B. PHILPOTT. 1988. Pairing stereo speakers using matching algorithms. *Asia-Pacific Journal of Operational Research* **5,** 101–116.

MATSUMOTO, K., T. NISHIZEKI, and N. SAITO. 1985. An efficient algorithm for finding multicommodity flows in planar networks. *SIAM Journal on Computing* **14,** 289–302.

MATULA, D. W. 1987. Determining edge connectivity in $O(nm)$. *Proceedings of the 28th Symposium on Foundations of Computer Science*, pp. 249–251.

MAXWELL, W. L., and R. C. WILSON. 1981. Dynamic network flow modelling of fixed path material handling systems. *AIIE Transactions* **13,** 12–21.

McGeoch, C. C. 1986. *Experimental Analysis of Algorithms*. Unpublished Ph.D. dissertation, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA.

McGeoch, C. C. 1992. Analysis of algorithms by simulation: Variance reduction techniques and simulation speedups. *Computing Surveys* **24,** June issue.

McGinnis, L. F., and H. L. W. Nuttle. 1978. The project coordinators problem. *OMEGA* **6,** 325–330.

Meggido, N. 1979. Combinatorial optimization with rational objective functions. *Mathematics of Operations Research* **4,** 414–424.

Meggido, N., and A. Tamir. 1978. An $O(n \log n)$ algorithm for a class of matching problems. *SIAM Journal on Computing* **7,** 154–157.

Mehlhorn, K. 1984. *Data Structures and Algorithms*, Vol. I: *Searching and Sorting*. Springer-Verlag, New York.

Micali, S., and V. V. Vazirani. 1980. An $O(\sqrt{|V|} \cdot |E|)$ algorithm for finding maximum matching in general graphs. *Proceedings of the 21st Annual Symposium on the Foundations of Computer Science*, pp. 17–27.

Minieka, E. 1978. *Optimization Algorithms for Networks and Graphs*. Marcel Dekker, New York.

Minoux, M. 1984. A polynomial algorithm for minimum quadratic cost flow problems. *European Journal of Operational Research* **18,** 377–387.

Minoux, M. 1986. Solving integer minimum cost flows with separable convex cost objective polynomially. *Mathematical Programming Study* **26,** 237–239.

Minoux, M. 1989. Network synthesis and optimal network design problems: Models, solution methods, and applications. *Networks* **19,** 313–360.

Minty, G. J. 1960. Monotone networks. *Proceedings of the Royal Society of London* **257A,** 194–212.

Monma, C. L., and M. Segal. 1982. A primal algorithm for finding minimum-cost flows in capacitated networks with applications. *Bell System Technical Journal* **61,** 449–468.

Moore, E. F. 1957. The shortest path through a maze. In *Proceedings of the International Symposium on the Theory of Switching, Part II;* The Annals of the Computation Laboratory of Harvard University **30,** Harvard University Press, pp. 285–292.

Mulvey, J. 1978. Pivot strategies for primal–simplex network codes. *Journal of ACM* **25,** 266–270.

Mulvey, J. M. 1979. Strategies in modeling: A personal scheduling example. *Interfaces* **9,** 66–76.

Nemhauser, G. L., and L. A. Wolsey. 1988. *Integer and Combinatorial Optimization*. Wiley, New York.

Orlin, D. 1987. Optimal weapons allocation against layered defenses. *Naval Research Logistics Quarterly* **34,** 605–617.

Orlin, J. B. 1984. Genuinely polynomial simplex and non-simplex algorithms for the minimum cost flow problem. Technical Report 1615-84, Sloan School of Management, MIT, Cambridge, MA.

Orlin, J. B. 1985. On the simplex algorithm for networks and generalized networks. *Mathematical Programming Study* **24,** 166–178.

Orlin, J. B. 1988. A faster strongly polynomial minimum cost flow algorithm. *Proceedings of the 20th ACM Symposium on the Theory of Computing*, pp. 377–387. Full paper to appear in *Operations Research*.

Orlin, J. B., and R. K. Ahuja. 1992. New scaling algorithms for the assignment and minimum cycle mean problems. *Mathematical Programming* **54,** 41–56.

Orlin, J. B., and U. G. Rothblum. 1985. Computing optimal scalings by parametric network algorithms. *Mathematical Programming* **32,** 1–10.

Osteen, R. E., and P. P. Lin. 1974. Picture skeletons based on eccentricities of points of minimum spanning trees. *SIAM Journal on Computing* **3,** 23–40.

Pallottino, S. 1991. Personal communications.

Papadimitriou, C. H., and K. Steiglitz. 1982. *Combinatorial Optimization: Algorithms and Complexity*. Prentice Hall, Englewood Cliffs, NJ.

Pape, U. 1974. Implementation and efficiency of Moore-algorithms for the shortest route problem. *Mathematical Programming* **7,** 212–222.

Pape, U. 1980. Algorithm 562: Shortest path lengths. *ACM Transactions on Mathematical Software* **6,** 450–455.

Phillips, D. T., and A. Garcia-Diaz. 1981. *Fundamentals of Network Analysis*. Prentice Hall, Englewood Cliffs, NJ.

PICARD, J. C., and M. QUEYRANNE. 1982. Selected applications of minimum cuts in networks. *INFOR* **20,** 394–422.

PICARD, J. C., and H. D. RATLIFF. 1973. Minimal cost cut equivalent networks. *Management Science* **19,** 1087–1092.

PICARD, J. C., and H. D. RATLIFF. 1978. A cut approach to the rectilinear distance facility location problem. *Operations Research* **26,** 422–433.

PINAR, M. C., and S. A. ZENIOS. 1990. Parallel decomposition of multicommodity network flows using smooth penalty functions. Technical Report 90-12-06, Department of Decision Sciences, Wharton School, University of Pennsylvania, Philadelphia, PA.

PINTO, Y., and R. SHAMIR. 1990. Efficient algorithms for minimum cost flow problems with convex costs. Technical Report, Department of Computer Science, Tel Aviv University, Tel Aviv, Israel.

PLOTKIN, S., and É. TARDOS. 1990. Improved dual network simplex. *Proceedings of the First ACM–SIAM Symposium on Discrete Algorithms*, pp. 367–376.

POTTS, R. B., and R. M. OLIVER. 1972. *Flows in Transportation Networks*. Academic Press, New York.

PRAGER, W. 1957. On warehousing problems. *Operations Research* **5,** 504–512.

PRIM, R. C. 1957. Shortest connection networks and some generalizations. *Bell System Technical Journal* **36,** 1389–1401.

RATLIFF, H. D. 1978. Network models for production scheduling problems with convex cost and batch processing. *AIIE Transactions* **10,** 104–108.

RAVINDRAN, A. 1971. On compact book storage in libraries. *Opsearch* **8,** 245–252.

RECSKI, A. 1988. *Matroid Theory and Its Applications*. Springer-Verlag, New York.

RHYS, J. M. W. 1970. A selection problem of shared fixed costs and network flows. *Management Science* **17,** 200–207.

RÖCK, H. 1980. Scaling techniques for minimal cost network flows. In *Discrete Structures and Algorithms*. Edited by V. Page. Carl Hanser, Munich, pp. 181–191.

ROCKAFELLAR, R. T. 1970. *Convex Analysis*. Princeton University Press, Princeton, NJ.

ROCKAFELLAR, R. T. 1984. *Network Flows and Monotropic Optimization*. John Wiley & Sons, New York.

ROOHY-LALEH, E. 1980. Improvements to the Theoretical Efficiency of the Network Simplex Method. Unpublished Ph.D. dissertation, Carleton University, Ottawa, Canada.

ROSENTHAL, R. E. 1981. A nonlinear network flow algorithm for maximization of benefits in a hydroelectric power system. *Operations Research* **29,** 763–786.

ROSS, G. T., and R. M. SOLAND. 1975. A branch and bound algorithm for the generalized assignment problem. *Mathematical Programming* **8,** 91–103.

ROTH, A. E., U. G. ROTHBLUM, and J. H. VANDE VATE. 1990. Stable matchings, optimal assignments and linear programming. Rutcor Research Report 23-90, The State University of New Jersey, Rutgers, NJ.

ROTHFARB, B., N. P. SHEIN, and I. T. FRISCH. 1968. Common terminal multicommodity flow. *Operations Research* **16,** 202–205.

SAKAROVITCH, M. 1973. Two commodity network flows and linear programming. *Mathematical Programming* **4,** 1–20.

SAPOUNTZIS, C. 1984. Allocating blood to hospitals from a central blood bank. *European Journal of Operational Research* **16,** 157–162.

SCHMIDT, S. R., P. A. JENSEN, and J. W. BARNES. 1982. An advanced dual incremental network algorithm. *Networks* **12,** 475–492.

SCHNEIDER, M. H., and S. A. ZENIOS. 1990. A comparative study of algorithms for matrix balancing. *Operations Research* **38,** 439–455.

SCHNEUR, R. 1991. Scaling algorithms for multicommodity flow problems and network flow problems with side constraints. Ph.D. dissertation, Department of Civil Engineering, MIT, Cambridge, MA.

SCHNORR, C. P. 1979. Bottlenecks and edge connectivity in unsymmetrical networks. *SIAM Journal on Computing* **8,** 265–274.

SCHRIJVER, A. 1986. *Theory of Linear and Integer Programming*. Wiley, New York.

SCHWARTZ, B. L. 1966. Possible winners in partially completed tournaments. *SIAM Review* **8,** 302–308.

SCHWARTZ, M., and T. E. STERN. 1980. Routing techniques used in computer communication networks. *IEEE Transactions on Communications* **COM-28,** 539–552.

SEGAL, M. 1974. The operator-scheduling problem: A network flow approach. *Operations Research* **22,** 808–823.

SERVI, L. D. 1989. A network flow approach to a satellite scheduling problem. Research Report, GTE Laboratories, Waltham, MA.

SHAPIRO, J. F. 1979. *Mathematical Programming: Structures and Algorithms.* Wiley, New York.

SHAPIRO, J. F. 1992. Mathematical programming models and methods for production planning and scheduling. To appear in *Handbooks in Operations Research and Management Science*, Vol. 4: *Logistics of Production and Inventory*, edited by S. C. Graves, A. H. G. Rinnooy Kan, and P. Zipkin. North-Holland, Amsterdam.

SHEPARDSON, F., and R. E. MARSTEN. 1980. A Lagrangian relaxation algorithm for the two-duty scheduling problem. *Management Science* **26,** 274–281.

SHIER, D. R. 1982. Testing for homogeneity using minimum spanning trees. *The UMAP Journal* **3,** 273–283.

SHILOACH, Y., and U. VISHKIN. 1982. An $O(n^2 \log n)$ parallel max-flow algorithm. *Journal of Algorithms* **3,** 128–146.

SLEATOR, D. D., and R. E. TARJAN. 1983. A data structure for dynamic trees. *Journal of Computer and System Sciences* **24,** 362–391.

SLUMP, C. H., and J. J. GERBRANDS. 1982. A network flow approach to reconstruction of the left ventricle from two projections. *Computer Graphics and Image Processing* **18,** 18–36.

SRINIVASAN, V. 1974. A transshipment model for cash management decisions. *Management Science* **20,** 1350–1363.

SRINIVASAN, V. 1979. Network models for estimating brand-specific effects in multiattribute marketing models. *Management Science* **25,** 11–21.

SRINIVASAN, V., and G. L. THOMPSON. 1972. An operator theory of parametric programming for the transportation problem. *Naval Research Logistics Quarterly* **19,** 205–252.

SRINIVASAN, V., and G. L. THOMPSON. 1973. Benefit–cost analysis of coding techniques for primal transportation algorithm. *Journal of ACM* **20,** 194–213.

STILLINGER, F. H. 1967. Physical clusters, surface tension, and critical phenomenon. *Journal of Chemical Physics* **47,** 2513–2533.

STOER, J., and C. WITZGALL. 1970. *Convexity and Optimization in Finite Dimensions.* Springer-Verlag, New York.

STONE, H. S. 1977. Multiprocessor scheduling with the aid of network flow algorithms. *IEEE Transactions on Software Engineering* **3,** 85–93.

SWOVELAND, C. 1971. Decomposition algorithms for the multi-commodity distribution problem. Working Paper 184, Western Management Science Institute, University of California, Los Angeles, CA.

SYSLO, M. M., N. DEO, and J. S. KOWALIK. 1983. *Discrete Optimization Algorithms.* Prentice Hall, Englewood Cliffs, NJ.

SZADKOWSKI, S. 1970. An approach to machining process optimization. *International Journal of Production Research* **9,** 371–376.

TALLURI, K. T. 1991. Issues in the design of survivable networks. Ph.D. dissertation, Operations Research Center, MIT, Cambridge, MA.

TARDOS, É. 1985. A strongly polynomial minimum cost circulation algorithm. *Combinatorica* **5,** 247–255.

TARDOS, É. 1986. A strongly polynomial algorithm to solve combinatorial linear programs. *Operations Research* **34,** 250–256.

TARJAN, R. E. 1982. Sensitivity analysis of minimum spanning trees and shortest path trees. *Information Processing Letters* **14,** 30–33.

TARJAN, R. E. 1983. *Data Structures and Network Algorithms.* SIAM, Philadelphia, PA.

TARJAN, R. E. 1984. A simple version of Karzanov's blocking flow algorithm. *Operations Research Letters* **2,** 265–268.

TARJAN, R. E. 1991. Efficiency of the primal network simplex algorithm for the minimum-cost circulation problem. *Mathematics of Operations Research* **16,** 272–291.

TOMIZAVA, N. 1972. On some techniques useful for solution of transportation network problems. *Networks* **1,** 173–194.

TOMLIN, J. A. 1966. A linear programming model for the assignment of traffic. *Proceedings of the 3rd Conference of the Australian Road Research Board* **3,** 263–271.

TRUEMPER, K. 1977. On max flow with gains and pure min-cost flows. *SIAM Journal on Applied Mathematics* **32,** 450–456.

TSO, M. 1986. Network flow models in image processing. *Journal of the Operational Research Society* **37,** 31–34.

TSO, M., P. KLEINSCHMIDT, I. MITTERREITER, and J. GRAHAM. 1991. An efficient transportation algorithm for automatic chromosome karotyping. *Pattern Recognition Letters* **12,** 117–126.

TUTTE, W. T. 1971. *Introduction to the Theory of Matroids*. American Elsevier, New York.

VAIDYA, P. M. 1989. Speeding up linear programming using fast matrix multiplication. *Proceedings of the 30th Annual Symposium on the Foundations of Computer Science*, pp. 332–337.

VAN SLYKE, R., and H. FRANK. 1972. Network reliability analysis: Part I. *Networks* **1,** 279–290.

VAZIRANI, V. V. 1989. A theory of alternating paths and blossoms for proving correctness of the $O(n^{1/2}m)$ general graph matching algorithm. Technical Report 89-1035, Department of Computer Science, Cornell University, Ithaca, NY.

VEINOTT, A. F., and G. B. DANTZIG. 1968. Integer extreme points. *SIAM Review* **10,** 371–372.

VEINOTT, A. F., and H. M. WAGNER. 1962. Optimal capacity scheduling: Parts I and II. *Operations Research* **10,** 518–547.

VOLGENANT, A. 1989. A Lagrangian approach to the degree-constrained minimum spanning tree problem. *European Journal of Operational Research* **39,** 325–331.

VON RANDOW, R. 1982. *Integer Programming and Related Areas: A Classified Bibliography 1978–1981*. Lecture Notes in Economics and Mathematical Systems, Vol. 197. Springer-Verlag, Berlin.

VON RANDOW, R. 1985. *Integer Programming and Related Areas: A Classified Bibliography 1981–1984*. Lecture Notes in Economics and Mathematical Systems, Vol. 243. Springer-Verlag, Berlin.

WAGNER, R. A. 1976. A shortest path algorithm for edge-sparse graphs. *Journal of ACM* **23,** 50–57.

WAGNER, D. K. 1990. Disjoint (s, t)-cuts in a network. *Networks* **20,** 361–371.

WALLACHER, C., and U. T. ZIMMERMANN. 1991. A combinatorial interior point method for network flow problems. Presented at the *14th International Symposium on Mathematical Programming*, Amsterdam, The Netherlands.

WARSHALL, S. 1962. A theorem on boolean matrices. *Journal of ACM* **9,** 11–12.

WATERMAN, M. S. 1988. *Mathematical Methods for DNA Sequences*. CRC Press, Boca Raton, FL.

WEINTRAUB, A. 1974. A primal algorithm to solve network flow problems with convex costs. *Management Science* **21,** 87–97.

WELSH, D. J. A. 1976. *Matroid Theory*. Academic Press, New York.

WHITE, L. S. 1969. Shortest route models for the allocation of inspection effort on a production line. *Management Science* **15,** 249–259.

WHITE, W. W. 1972. Dynamic transshipment networks: An algorithm and its application to the distribution of empty containers. *Networks* **2,** 211–230.

WHITING, P. D., and J. A. HILLIER. 1960. A method for finding the shortest route through a road network. *Operations Research Quarterly* **11,** 37–40.

WHITNEY, H. 1935. On the abstract properties of linear dependence. *American Journal of Mathematics* **57,** 509–533.

WINSTON, W. L. 1991. *Operations Research: Applications and Algorithms*. PWS-Kent, Boston, MA.

WITZGALL, C., and C. T. ZAHN. 1965. Modification of Edmonds maximum matching algorithm. *Journal of Research of the National Bureau of Standards* **69B,** 91–98.

WONG, R. T. 1980. Integer programming formulations of the traveling salesman problem. *Proceedings of the 1980 IEEE International Conference on Circuits and Computers*, pp. 149–152.

WRIGHT, J. W. 1975. Reallocation of housing by use of network analysis. *Operational Research Quarterly* **26,** 253–258.

YAO, A. 1975. An $O(|E| \log \log |V|)$ algorithm for finding minimum spanning trees. *Information Processing Letters* **4,** 21–23.

YOUNG, N. E., R. E. TARJAN, and J. B. ORLIN. 1990. Faster parametric shortest path and minimum balance algorithms. Working Paper 3112-90-MS, Sloan School of Management, MIT, Cambridge, MA.

ZADEH, N. 1973a. A bad network problem for the simplex method and other minimum cost flow algorithms. *Mathematical Programming* **5,** 255–266.

ZADEH, N. 1973b. More pathological examples for network flow problems. *Mathematical Programming* **5,** 217–224.

ZADEH, N. 1979. Near equivalence of network flow algorithms. Technical Report 26, Department of Operations Research, Stanford University, Stanford, CA.

ZAHN, C. T. 1971. Graph-theoretical methods for detecting and describing gestalt clusters. *IEEE Transactions on Computing* **C20,** 68–86.

ZAKI, H. 1990. A comparison of two algorithms for the assignment problem. Technical Report ORL 90-002, Department of Mechanical and Industrial Engineering, University of Illinois at Urbana-Champaign, Urbana, IL.

ZANGWILL, W. I. 1969. A backlogging model and a multi-echelon model of a dynamic economic lot size production system: A network approach. *Management Science* **15,** 506–527.

ZAWACK, D. J., and G. L. THOMPSON. 1987. A dynamic space–time network flow model for city traffic congestion. *Transportation Science* **21,** 153–162.

ZENIOS, S. A., and J. M. MULVEY. 1986. Relaxation techniques for strictly convex network problems. *Annals of Operations Research* **5,** 517–538.

INDEX

1-forest, 541
1-tree, 541
3-cover problem, 794–95
Δ-residual network, 211–12, 557–58
Δ-scaling phase, 211, 238, 360, 373, 557–60
ϵ-optimality conditions, 363

A* algorithm, 130
Active node, 224
Acyclic networks, 51
  applications of, 368–69, 444
  definition, 27
  determination of, 77–79
  properties, 51
Adjacency lists, 25, 34–35, 46
Adjacency matrix representations, 33–34, 46
Admissible arcs, 74, 210, 364
Admissible networks, 324, 368
Admissible paths, 210
Aircraft assignment, 570
Airline scheduling problem, 204
Algorithms
  animation, 714–15
  bad, 54
  easy, 788
  efficient, 54, 788
  good, 54
All pairs label correcting algorithms, 146–50
All-pairs minimum value cut problem, 277–86
All-pairs shortest path problem, 144–50, 155–56
Allocating contractors to public works, 345
Allocating receivers to transmitters, 454
Alternating paths, 476
Alternating tree, 479–80
Amortized complexity, 63–65
Analog solution of shortest paths, 96
Applications
  of convex cost flows, 562
  of generalized flow problems, 592
  of maximum flow problem, 197–98
  of matchings and assignments, 501
  of minimum cost flow problem, 342–44
  of minimum cut problem, 197–98
  of minimum spanning trees, 536
  of multicommodity flow problems, 685–86
  of shortest path problem, 123–24
Applications of network models
  in computer science and communication systems, 757
  in defense, 757
  in distribution and transportation, 759
  in engineering, 756
  in management science, 757
  in manufacturing, production and inventory planning, 756
  in physical and medical sciences, 757
  in scheduling, 757
  in social sciences and public policy, 758
Approximate optimality, 362–63
Approximating piecewise linear functions, 98–99, 131
Arborescence, 511
Arc adjacency list, 25, 34
Arc coloring problem, 504
Arc connectivity, 273–74, 292–93
Arc reversal transformation, 40
Arc routing problems, 740–44
Arc tolerances, 130–31
Assigning medical school graduates to hospitals, 465
Assignment problem, 7, 470–73
Assortment of structural steel beams, 11, 21
Asymptotic bottleneck operations, 702–07
Augmented forest structures, 572–90
Augmented tree, 575
Augmenting cycle theorem, 83
Augmenting path algorithm, 180–84, 223
Augmenting path theorem, 185, 478
Average-case analysis, 56–57

Backward arc, 26
Balanced assignment problem, 505–06
Balanced nodes, 80, 320
Balanced spanning tree problem, 540
Baseball elimination problem, 258–59, 289
Basic feasible solutions
  for generalized flows, 582–83
  for linear programs, 805–10
  for minimum cost flows, 445
  for multicommodity flows, 679–82
Basis property
  for generalized flows, 582–83
  for minimum cost flows, 442–46
Bellman's equations, 158
Berge's theorem, 508
Bicycles, 587–89
Big Ω notation, 59–60
Big θ notation, 63
Bin packing problem, 87
Binary heaps
  applications, 116, 525
  data structure, 778
Binary search
  applications, 88, 152, 791
  technique, 72–73
Binomial coefficients, 70

Bipartite matching algorithm, 469–73, 478–81
Bipartite networks, 49, 51, 288
  applications, 41
  definition, 31
  in matching algorithms, 189–91
  in maximum flow algorithms, 255–59, 286
  in minimum cost flow algorithms, 373, 399–400
  in shortest path algorithms, 159
  properties, 31, 49, 51
Bit-scaling algorithms
  basic approach, 68–70
  for maximum flow problem, 246
  for minimum cost flow problem, 400
  for shortest path problem, 164
Blocking arc, 418
Blocking flows, 221–22
Blossoms, 483–94
Book storage in libraries, 124–25
Bottleneck assignment problem, 505
Bottleneck operations, 704–07
Bottleneck spanning tree problem, 540
Bottleneck transportation problem, 355
Branch and bound technique, 602–04
Breadth-first search, 76, 90, 107
Breakeven cycle, 574
Bridges of Königsberg, 48
Bubble sort algorithm, 86
Buckets, 113–14, 116–21
Building evacuation models, 738–39

Candidate list pivot rule, 417
Capacitated minimum spanning tree problem, 354, 647
Capacity expansion problems, 562–64, 641
Capacity of a cut, 178
Capacity scaling algorithms
  for convex cost flows, 555–60
  for maximum flows, 210–12, 220, 240, 246
  for minimum cost flows, 360–62, 373–76, 382–95, 400
Caterer problem, 453, 593
Certificate checking algorithm, 793–94
Chinese postman problems
  undirected version, 742–44
Circulation problem, 7, 20, 81, 92
  feasibility conditions, 195
  for multicommodity flows, 687–88
Class $\mathcal{NP}$, 793–96
Class $\mathcal{NP}$-complete, 795–801
  strong $\mathcal{NP}$-completeness, 799–800
Class $\mathcal{NP}$-hard, 796
Class $\mathcal{P}$, 792–95
Clique, 50
Cluster analysis, 125, 515–16
Coloring problems, 49, 504
Column generation approach, 665–70
Comparison of algorithms, 707
Complementary slackness conditions
  for generalized flows, 576–77
  for Lagrangian relaxation, 607
  for linear programs, 819–20
  for minimum cost flows, 309–14, 330
  for minimum spanning trees, 531–32
  for multicommodity flows, 658, 667–68
Complexity analysis, 56–66
Components of a graph, 27
Computational testing of algorithms, 695–716
Concentrator location problem, 125–26
Concurrent flow problem, 691
Connectivity
  algorithms, 273–77, 286, 292–93
  arc, 273–74, 292–93
  biconnectivity, 288
  definition, 27
  node, 273, 293
Constrained maximum flows, 400–01
Constrained minimum cost flows, 460, 621–22
Constrained minimum spanning trees, 631–33
Constrained shortest paths, 599–600, 762, 798
Contractions, 384–85, 492
Conversion of physical entities, 568–69
Convex cost flow problem, 7, 543–65
Convexification, 618, 646
Cost scaling algorithms
  for assignment problem, 472–73
  for convex cost flows, 565
  for minimum cost flows, 362–72, 399
Coverage of sporting events, 205
Crew scheduling, 127
Currency conversion problem, 593
Current-arc data structure, 75, 82, 216–17, 365–72
Current forest, 234
Cuts, 27
  s-t cuts, 28
Cycle-canceling algorithms
  for convex cost flows, 555–56
  for minimum cost flows, 317–19, 340, 376–82
  specific implementations, 319, 376–82
Cycle free solutions, 405–09
Cycles, 26
Cyclic scheduling problem, 622
Cyclic staff scheduling problem, 346–47

d-heaps
  applications, 116, 525
  data structure, 773–78
Dancing problem, 504
Dantzig-Wolfe decomposition, 652–53, 671–73
Data scaling, 725–28
Data structures
  arrays, 766
  binary heaps, 778
  current-arc, 75
  d-heaps, 773–78
  Fibonacci heaps, 779–87
  linked lists, 767–71
Dating problem, 21
Deficit of a node, 80, 320
Degrees, 25
Degeneracy
  in dual network simplex method, 438–39
  in network simplex method, 418, 420–25
  in simplex method, 814
Deployment of firefighting companies, 763–64
Deployment of resources, 686
Depth-first search
  applications, 410
  technique, 76, 90
Depth index, 410
Dequeue, 143
Descendants, 29

Designing physical systems, 512–13
Destruction of military targets, 723
Determining an optimal energy policy, 16
Determining chemical bonds, 466
Dial's implementation
  algorithm, 113–14, 129
  applications, 122, 700
Dijkstra's algorithm, 108–22
  bidirectional implementation, 112–13, 132
  Dial's implementation, 113–14, 122, 129, 700
  Johnson's implementation, 116
  original implementation, 108–12, 122
  radix heap implementation, 116–22
  relationship to label correcting algorithms, 141
  reverse implementation, 112
Dinic's algorithm, 221–23
Dining problem, 198
Directed cycles, 27
Directed in-trees, 30
Directed networks
  definitions, 24–31
  representation, 31–38
Directed out-trees, 29
Directed path, 26
Directed walk, 26
Distance labels, 209–10, 221–23
Distribution problems, 298–99, 654
DNA sequence alignment, 728–31
Double scaling algorithm, 373–76, 399
Doubly linked lists
  applications, 113, 229, 372, 527
  data structure, 769–71, 773
Doubly stochastic matrix, 504
Distributed computing on computers, 174–75
Dual completion of oil wells, 465
Dual integrality property, 413
Dual networks, 262–65, 291
Duality gap, 614, 620
Duality theory
  for linear programming, 816–20
  for minimum cost flows, 310–15, 384
  for multicommodity flows, 657–58
Dynamic flow problems, 737–40
Dynamic lot sizing, 749–52
Dynamic programming
  applications, 88, 102, 107, 148, 153, 162, 729–31
  technique, 70–72
Dynamic trees
  applications, 372
  data structure, 265–73, 286

Economic order quantity, 748
Electrical networks, 15
Eligible arcs, 416, 585–86
  in dual network simplex, 438
  in network simplex, 416
  in parametric network simplex, 434
Empirical analysis of algorithms, 56–57, 695–716
Employment scheduling, 306
Endpoints, 25
Equipment replacement problem, 306, 347
Euler's formula, 261
Euler's theorem, 742–43
Euler's tour, 91
Excess dominator, 237
Excess of a node, 80, 224, 320
Exponential-time algorithms, 60–62
Extreme point solutions, 533
Extreme points, 804–05, 809–10

Factored assignment problems, 505
Factored minimum spanning tree problem, 539–40
Factored transportation problem, 345
Faculty-course assignment, 454
Feasible flow problem
  algorithm, 169–70
  applications, 170–74, 194, 258–59, 563
  feasibility conditions, 196, 205
  max and min arc flows, 283–85
Feasibility of perfect matchings, 503
Fibonacci heaps
  applications, 116, 122, 525
  data structure, 779–87
Fibonacci numbers, 779
FIFO label correcting algorithm, 142–44, 155, 159, 429, 700
FIFO preflow-push algorithm, 231–32, 696
Flow across a cut, 179
Flow bound constraints, 5
Flow decomposition
  applications of, 92, 183–84, 188–90, 228, 308, 398, 470, 596–97, 666, 741, 743
  theory, 79–83
Flow property, 388
Flowers, 483
Floyd-Warshall algorithm, 147–50, 156, 162
Flyaway kit problem, 724–25
Forest, 28–29, 49
Forest scheduling problem, 22
Forward arc, 26
Forward star representation, 35–37, 46
Fractional b-matching problem, 354
Fundamental cuts, 30
Fundamental cycles, 30

Gainy arc, 8, 568, 574
Gainy cycle, 574
Generalized assignment problem, 639–40
Generalized flow problems, 8, 566–97, 800, 596
Generalized upper bounding simplex method, 666–67
Geometric improvement approach
  applications, 211, 377
  basic ideas, 67–68
Good algorithm, 54
Greedy algorithm, 528–30, 541

Hamiltonian cycle problem, 794–95
Hamiltonian path problem, 797
Hard problems, 789
Head nodes, 25
Heaps, 773–87
Hungarian algorithm, 471–72

Imbalance of a node, 80, 320
Imbalance property, 388
Incidence matrix, 5, 32–33, 46
Incoming arc, 25
Indegree, 25
Independent arcs, 190, 205
Independent nodes, 50
Insights into algorithms, 709–11
Inspection of a production line, 99–100
Instance of a problem, 56
Integer programming, 531–33, 598–648, 794–95, 799

Integrality assumption, 6
Integrality property
  for Lagrangian relaxation, 619–20
  for maximum flows, 186
  for minimum cost flows, 318, 413, 415, 447–49
Isomorphic graphs, 49, 790

Just-in-time scheduling, 734–35

Karyotyping of chromosomes, 731
Kilter diagram, 327
Kilter number, 327–31
Knapsack problem, 71–72, 88, 100–02, 127, 131, 697
Knight's tour problem, 89
Kruskal's algorithm, 520–23, 530–34

Label correcting algorithms, 136–65
  and network simplex algorithm, 427–28
  dequeue implementation, 143, 155, 161
  FIFO implementation, 142–44, 155, 159, 317
  for finding negative cycles, 143–44, 159
  generic implementation, 136–41, 155, 159, 161
Labeling algorithm, 70, 184–87, 240, 252–55, 274, 700
  pathological example, 205–06
Lagrangian decomposition, 647–48
Lagrangian multiplier problem, 607–15
Lagrangian relaxation, 598–648
  for minimum cost flows, 332
  for multicommodity flows, 660–65
Land management, 571–72
Layered networks, 88, 221–23
Leaving arc rule, 423
Leveling mountainous terrain, 12
Linear programming, 802–20
  and assignment problem, 471
  and convex cost flows, 552–53
  and generalized flow, 567–68, 582–83
  and greedy algorithm, 541
  and Lagrangian relaxation, 615–20, 638–39
  and matroids, 541–42
  and maximum flows, 168
  and minimum cost flows, 296, 304–06, 310–15
  and minimum ratio cycle problem, 163–64
  and multicommodity flows, 649–50, 666
  and primal-dual algorithm, 326
  and shortest paths, 94, 136
  and spanning trees, 530–33
Linear programs
  canocial form, 806
  standard form, 803
  symmetric form, 817
  with consecutive 1's in columns, 304–06, 314–15, 344
  with consecutive 1's in rows, 314–15, 346–47, 737, 748
Linked lists
  applications, 34–35, 233, 239, 521, 527
  data structure, 767–69, 773
Loading of a hopping airplane, 302
Locating objects in space, 466
Location and layout problems, 163, 640–41, 744–48, 764
Longest path problem, 91, 102, 129, 797
Loops, 25
Lossy arc, 8, 568
Lossy cycle, 574

Machine loading problem, 569–70
Machine scheduling, 172–74, 303–04, 468–69
Mass balance constraints, 5
Matching problems, 9, 461–509
  and Chinese Postman, 743–44
  and maximum flows, 189–191
  and shortest paths, 494–98
  three-dimensional, 800
Matrix balancing, 548–49
Matrix manipulation algorithms, 150
Matrix rounding problems, 171–72, 454–55
Matroids, 528–30, 533, 541–42
Max-flow min-cut theorem, 184–85
  combinatorial implications, 188–191
  for nonzero lower bounds, 193
  linear programming proof, 432
Maximum capacity augmenting path algorithm, 210–11
Maximum capacity path problem, 129
Maximum cut problem, 800
Maximum dynamic flow problem, 738
Maximum flow problem, 6, 69–70, 166–293
Maximum flows
  and minimum cost flows, 324–26, 339
  and primal-dual algorithm, 324–26
  in bipartite networks, 255–59
  in planar networks, 260–65
  in unit capacity networks, 252–55
  with nonzero lower bounds, 191–96
Maximum preflow, 245
Maximum spanning tree problem, 278, 519–20
Maximum weight closure, 719–25
Maze problem, 89
Measuring homogeneity of bimetallic objects, 14
Min-cost max-flow problem, 352
Min-value max-cut theorem, 202
Minimax path problem, 513–14
Minimax transportation problem, 199
Minimum cost flow problem, 4–5, 52, 83, 294–460
Minimum cost flows
  and assignment problem, 470–73
  and convex cost flow problem, 552–53
  and maximum flow problem, 324–26, 339
  and shortest path problem, 316, 320–32, 360–62, 382–94
Minimum cut problem, 167, 178, 184–85, 204
  all-pairs, 277–86
  applications, 174–76, 283–85
  in planar networks, 262–63
  with fewest arcs, 247
Minimum disconnecting set, 273–77
Minimum flow problem
  algorithm, 202
  min-value max-cut theorem, 202
Minimum mean cycle problem, 152–54
  application to data scaling, 728
  application to minimum cost flow algorithms, 319, 376–82

Minimum ratio cycle problem, 150–54, 163–64
Minimum ratio rule, 812
Minimum ratio spanning trees, 541
Minimum spanning trees, 8, 510–42
  and all-pairs min cut problem, 278
  applications, 536
Minimum value problem. See Minimum flow problem
Mold allocation, 754
Money-changing problem, 125
More-for-less paradox, 354
Multiarcs, 25
Multicommodity flows, 8, 649–94
  funnel problem, 688–89
  in two-commodity networks, 690
  in undirected networks, 689–90
  maximum flow version, 690–91
  multisink problem, 688
  multisource problem, 688
Multidrop terminal layout problem, 632
Multipliers of arcs, 568
Multipliers of paths and cycles, 573–74

Negative cycle detection
  algorithms, 136, 143–44, 149, 162, 428, 495
  applications, 103–04, 151–52, 317, 727
Negative cycle optimality conditions, 307–08
Negative cycle optimality theorem, 83
Network connectivity, 188–91, 273–77
Network decomposition algorithms, 79–83
Network design problems, 627–28, 642
Network flow books, 19–20
Network interdiction problem, 763
Network reliability testing, 259
Network representations, 31–38, 46
Network simplex algorithms, 402–60
  degeneracy in, 421
  empirical analysis, 702–12
Network transformations, 38–46
Network types
  communication, 10, 654
  computer, 10, 654
  energy, 569
  financial, 568
  hydraulic, 10
  mechanical, 10
  transportation, 10
Node-arc incidence matrix, 5, 32, 46, 50, 449
Node adjacency list, 25, 34
Node capacities, 42, 203
Node coloring, 49
Node connectivity, 273, 279
Node cover, 50, 189–91
Node-node adjacency matrix, 33, 46, 50, 51
Node potentials, 308
Node splitting transformation
  applications, 189, 497–98
  technique, 41
Nonbipartite matching problem, 475–494, 498
Nonsaturating push, 225, 364
Nontree arcs, 30
NP-completeness, 788–801
Nurse scheduling problem, 453
Nurse staff scheduling, 198

Open pit mining, 721–23
Operator scheduling, 628–31
Optimal capacity scheduling, 306
Optimal depletion of inventory, 468
Optimal message passing, 513
Optimality conditions
  for all-pairs shortest paths, 146
  for generalized flows, 576–77, 597
  for Lagrangian relaxation, 606
  for minimum cost flows, 306–10, 408–09
  for minimum spanning trees, 516–19, 531–32
  for multicommodity flows, 657–58, 667–68
  for shortest paths, 135–36, 306–07
Out-of-kilter algorithm, 326–31, 340
Outdegree, 25
Outgoing arc, 25

Painted network theorem, 203
Pairing stereo speakers, 14
Paragraph problem, 21
Parallel arcs, 25, 37–38, 128, 203
  representation, 37–38
Parameter balancing
  applications, 87, 116, 525
  technique, 65–66, 87
Parametric analysis
  for maximum flows, 248
  for minimum cost flows, 459–60
  for minimum spanning trees, 540
  for shortest paths, 164–65, 433–37
Parking model, 762–63
Partition problem, 794–95
Partitioning algorithm
  for shortest paths, 160–61
Partitioning methods
  for multicommodity flows, 653, 678–88
Passenger routing, 454
Path and cycle flow, 80–83
Path flow formulation, 665–66
Path optimality conditions, 519
Path problems
  maximum capacity, 129, 162
  maximum multiplier, 160, 162
  maximum reliability, 130
  minimax, 513–14
  with additional constraints, 131
  with resource constraints, 131
  with turn penalties, 130
Pathological examples, 161, 205–06
Paths, 26
Penalty approach, 692–93
Perfect b-matching, 496–97
Performance measures, 714
Permanently labeled node, 109
Permutation matrix, 504
Personnel assignment, 21, 463–64
  bipartite, 463–64
  nonbipartite, 464–65
Personnel planning problem, 126
Perturbation
  and strongly feasible solutions, 457
  for generalized flows, 590
  for minimum cost flows, 457–59
Phasing out capital equipment, 345
Physical networks, 9–10
Pivot operations
  for dual network simplex method, 434–35
  for linear programming, 811–13
  for network simplex method, 418–20, 711

Pivot rules
for generalized flows, 585
for minimum cost flows, 416–17
for shortest paths, 428–29
Planar networks, 260–65, 286–87
Police patrol problem, 21–22
Policemen's problem, 509
Polyhedron, 804–05
Polynomial reductions, 790–92
Polynomial-time algorithms, 60–62
pseudopolynomial, 61
strongly polynomial, 61
Polynomial-time algorithms for
all-pairs shortest paths, 147–48
assignment problem, 470–73
bipartite matchings, 469–70, 478–80
convex cost flows, 556–60
maximum flows, 210–40
minimum cost flows, 360–95
nonbipartite matchings, 475–94
shortest paths, 108–112, 115–22, 141–43, 429–30
spanning trees, 520–28
Polynomial transformations, 792–801
Polynomially equivalent, 791
Potential functions
applications, 165, 228–29, 232, 235–37, 239, 257–58, 369–70, 430, 782, 784
technique, 63–65
Potential of a node, 43
Practical improvements
for cost scaling algorithms, 365–66
for Dial's implementation, 129
for preflow-push algorithms, 229–30
for shortest augmenting path algorithm, 219–20
for successive shortest path algorithm, 323–24
Predecessor graph, 137–39
Predecessor index, 26, 29, 410
Preflow, 224
Preflow push algorithms
empirical testing, 700
excess scaling implementation, 237–40, 247
FIFO implementation, 230–34, 240, 246
for bipartite networks, 255–58, 290
generic version, 223–31, 240, 255–58
highest label implementation, 233–36, 240, 246
Preorder traversal, 76
Price-directive decomposition, 652
Prim's algorithm, 523–26, 534
Primal-dual algorithm, 324–26, 340
Priority queue, 773–87
Problem of queens, 50
Problem of representatives, 170–71
Problem size, 57–58
Production-inventory planning models, 748–53
Production planning, 633–35
Production property, 750
Production scheduling problem, 593
Project assignment, 453–54
Project management, 732–37
Pseudoflow, 320
Pseudopolynomial-time algorithms, 113–14, 140, 143, 136, 317–37, 554–56
Pushes, 223

Queues
applications, 142, 231
data structure, 772–73

Racial balancing of schools, 17, 301–02, 347, 563
Radix heaps, 116–21
Reallocation of housing, 10, 163
Recognition problems, 790–91
Reconstructing left ventricle from X-ray projections, 299–300
Reduced cost optimality conditions, 308–09
Reduced costs, 43–44, 308, 808
Reducing data storage, 514
Relabel operation, 213, 225, 364
Relaxation algorithm, 332–37, 340, 472
Repeated shortest path algorithm, 144–45, 156
Reporting computational experiments, 714
Representative operation counts, 698–716
Residual capacity, 44
Residual capacity of a cut, 178
Residual networks, 44–46, 51, 83, 177, 298, 554–55
Resource-directive decomposition, 652, 674–78
Reverse search algorithm, 76
Reverse star representation, 35–37
Revised simplex method, 813–14
Rewiring of typewriters, 13
Rooted trees, 29
Routing multiple commodities, 653
Running time of algorithms, 58–66
Ryser's theorem, 248–49

s-t cut, 177–78
s-t planar networks, 263–65
Saturating push, 225, 364
Scaling algorithms
basic ideas, 68–70
for convex cost flows, 556–61
for maximum flows, 210–12, 237–39, 246
for minimum cost flows, 360–94, 400
for shortest paths, 164
Scheduling problems, 172–74, 303–04, 468–69
School bus driver assignment, 501
Search algorithms
basic approaches, 73–79
Search trees, 74, 76, 90, 107, 479
Seat-sharing problem, 21
Selecting freight terminals, 722
Semi-bipartite networks, 132, 290
Sensitivity analysis
for maximum flows, 204
for minimum cost flows, 337–39, 353, 439–40
for minimum spanning trees, 539
for shortest paths, 159–60, 163
Separable functions, 544
Separator tree, 279–83
Sequencial search algorithm, 151–52
Sharp distance labels, 160
Shortest augmenting path algorithm, 213–23, 240, 252–55, 265–73
Shortest path tree, 106–07, 139
Shortest paths, 6, 93–165
application to min cost flows, 262–63, 320–56, 360–62, 382–94
enumerating all paths, 160
in acyclic networks, 107–08
in bipartite networks, 132, 159
in layered networks, 88
Similarity assumption, 60, 799–800

Simplex method,
for bounded variables, 814–15
for generalized flows, 583–89
for linear programming, 810–19
for maximum flows, 430–33
for minimum cost flows, 415–21
for shortest paths, 425–30
generalized upper bounding, 666–67
revised, 813–14
Simplex multipliers
for linear programs, 808
for minimum cost flows, 445–46
Ski instructor's problem, 501
Small-capacity networks, 289
Sollin's algorithm, 526–28, 534
Solving systems of equations, 199
Sorting, 86, 521, 774, 778
Spanning subgraph, 26
Spanning tree, 30
Spanning tree solutions, 405–09
Spanning tree structures, 408–09
Stable marriage problem, 473–75
Stable matchings, 475
Stable university admissions, 507
Stacks
applications, 64–65
Statistical security of data, 199, 283–85
Steiner tree problem, 642
Stick percolation problem, 550–51
Storage policy for libraries, 344–45
Strong connectivity
algorithm, 77
definition, 27
Strong duality theorem
for linear programs, 818–19
for minimum cost flows, 312–13
Strongly feasible solutions, 421–25, 432, 457, 590
and perturbation, 457
Subgradient optimization
application to multicommodity flows, 663–65
technique, 611–15
Subgraph, 26
Subset systems, 528–30
Subtour breaking constraints, 626
Successive shortest path algorithm
applications, 360, 437, 471, 556, 639, 701
basic approach, 320–24, 340
Succint certificate, 794
Symmetric difference, 477
System of difference constraints, 103–05, 127, 726–28

Tail nodes, 25
Tanker scheduling problems, 176–77, 347, 656
Telephone operator scheduling, 105–06, 127
Teleprocessing design problem, 632
Temporarily labeled nodes, 109
Terminal assignment problem, 346
Thread index, 410–14, 443–46
Threshold algorithm, 161
Time complexity function, 58
Time-cost trade-off problem, 735–37
Time-expanded networks, 737–40
Topological ordering
algorithm, 77–79
applications, 11, 107–08, 371–72
Totally unimodular matrices, 448–49
Tournament problem, 12
Traffic flows, 547
Tramp steamer problem, 103, 150
Transfers in communication networks, 547–48
Transformations
for removing arc capacities, 40
for removing nonzero lower bounds, 39
for removing undirected arcs, 39
node splitting, 41–43
Transitive closure, 90, 91
Transportation problem, 7, 9, 20, 294
Travelling salesman problem. See TSP
Tree arcs, 30
Tree indices, 410–14, 419, 576
Tree of shortest paths, 106, 139
Trees, 28–30
Triangularity property, 443–47
Triple operation, 147
Truck scheduling problem, 763
TSP, 623–25, 643–44, 790–91, 794, 797

Uncapacitated networks, 40–41
Undirected networks
definitions, 25, 31
representations, 38
transformation, 39
Unimodular matrices, 447–49
Unimodularity property, 447–49
Union-find operation, 522
Unique label property, 481–82
Unit capacity networks
and bipartite matchings, 469–70
and minimum cost flows, 399
and network connectivity, 188–91, 274
maximum flows in, 252–55, 285, 289
Unstable roommates, 507

Validity conditions, 209
Variable splitting, 630
Variational principle, 16, 547
Vehicle fleet planning, 344
Vehicle routing, 625–27, 645–47
Virtual running times, 707–09
Vital arcs, 128–29, 244

Walk, 26
Warehousing problem, 570, 655
Wave algorithm, 246
Weak duality theorem
for Lagrangian relaxation, 606
for linear programs, 817–18
for minimum cost flow, 312
Wine division problem, 90
Worst-case complexity, 56–66

Zero length cycle, 151, 160
Zoned warehousing, 345