CHAPTER 100
Discrete Optimization

RONALD L. RARDIN
Purdue University

Optimization is the process of selecting a solution from among available decision alternatives so that it conforms to all problem constraints and (at least approximately) maximizes or minimizes one or more objective/criterion functions. *Discrete optimization* is the branch confronting the vast array of problems having decisions of a logical or countable nature. Instead of, say, selecting an operating temperature, which is a decision that can pick any value in a *continuous* interval, discrete decisions are those with only specified list of options: turn left or turn right, build or do not build a plant, undertake job *A* before job *B* or *B* before *A*. If all decisions of a problem are discrete, the problem is termed *pure;* otherwise (e.g., if decisions include both whether to setup a process and what temperature to operate it at), the problem is *mixed*. The Appendix to this chapter presents many specific discrete optimization models. Other names for discrete optimization are *combinatorial optimization, integer programming,* and *mixed-integer programming*. A good introduction is provided in Rardin (1998, chaps. 9–12). More advanced books include Wolsey (1988), Parker and Rardin (1988), and Nemhauser and Wolsey (1988). Other standard sources are Schrijver (1986), Papadimitriou and Steiglitz (1982), and Lawler (1976).

1. MODELING

Modeling is the process of mathematically representing a problem in a form conducive to analysis and solution. The logical nature of discrete optimization—especially pure discrete optimization—invites a variety of quite different representations. Many problems can usefully be modeled in terms of logical predicates, objects and sets, graphs, or numerous other constructs.

The format that has proved most useful, and the focus of this chapter, is *numerical representation*—formulation of the problem in terms of numerically valued decision variables. Discrete decision options are encoded as specific numerical variable values. For example, variable $y_j = 1$ may mean plant j is selected for construction and $y_j = 0$ that it is not.

 Handbook of Industrial Engineering: Technology and Operations Management, Third Edition.
Edited by Gavriel Salvendy

TABLE 1 Representing Discrete Phenomena with Numerical Variables

Phenomena	Representation
At least K decisions j of subset J must be taken.	$y_j \triangleq 1$ if j is in the solution, 0 otherwise $\Sigma_{j\in J}\, y_j \geq K$
Exactly K decisions j of subset J must be taken.	$y_j \triangleq 1$ if j is in the solution, 0 otherwise $\Sigma_{j\in J}\, y_j = K$
At most K decisions j of subset J can be taken.	$y_j \triangleq 1$ if j is in the solution, 0 otherwise $\Sigma_{j\in J}\, y_j \leq K$
At most K variables z_j with j in subset J can be positive in a solution.	$y_j \triangleq 1$ if $z_j > 0$, 0 otherwise $\Sigma_{j\in J}\, y_j \leq K$ $z_j \leq M\, y_j$ for all $j \in J$
Decision j is allowed only if all decisions $i_1, \ldots, i_n$ are taken.	$y_{ik} \triangleq 1$ if decision i_k is taken, 0 otherwise $y_j \triangleq 1$ if decision j is taken, 0 otherwise $y_j \leq y_{i_k}$ for all $k = 1, \ldots, n$
Decision j is allowed only if some decision $i_1, \ldots, i_n$ is taken.	$y_{i_k} \triangleq 1$ if decision i_k is taken, 0 otherwise $y_j \triangleq 1$ if decision j is taken, 0 otherwise $y_j \leq \Sigma_{k=1}^{n}\, y_{i_k}$
Decision j is implied if all decisions $i_1, \ldots, i_n$ are taken.	$y_{i_k} \triangleq 1$ if decision i_k is taken, 0 otherwise $y_j \triangleq 1$ if decision j is taken, 0 otherwise $\Sigma_{k=1}^{n}\, y_{i_k} \leq (n-1) + y_j$
Decision j is implied if any decision $i_1, \ldots, i_n$ is taken.	$y_{i_k} \triangleq 1$ if decision i_k is taken, 0 otherwise $y_j \triangleq 1$ if decision j is taken, 0 otherwise $y_{i_k} \leq y_j$ for all $k = 1, \ldots, n$
Nonnegative fixed cost F_j is incurred whenever variable z_j is positive.	$y_j \triangleq 1$ if z_j positive, 0 otherwise min $\ldots + F_j y_j + \ldots$ $z_j \leq M\, y_j$
If decision i is taken and decision j is taken, cost C_{ij} is incurred.	$y_i \triangleq 1$ if decision i is taken, 0 otherwise $y_j \triangleq 1$ if decision j is taken, 0 otherwise min $\ldots + C_{ij} y_i y_j + \ldots$
Either task i of duration T_i done before task j of duration T_j or vice versa.	$z_i \triangleq$ task i start time; $z_j \triangleq$ task j start time $y_{ij} \triangleq 1$ if task i before j, 0 otherwise $0 \leq z_i + T_i \leq z_j - M\,(1 - y_{ij})$ $0 \leq z_j + T_j \leq z_i - M\, y_{ij}$
Function $\theta(p) = \Theta_1, \ldots, \Theta_n$ at $p = P_1 < \ldots < P_n$, with linear interpolation between P_j's.	$y_j \triangleq 1$ if j left interpolation point, 0 otherwise $\theta \triangleq$ interpolated value $z_j \triangleq$ weight on point j $p = \Sigma_{j=1}^{n}\, P_j z_j$; $\theta = \Sigma_{j=1}^{n}\, \Theta_j z_j$ $\Sigma_{j=1}^{n-1}\, y_j = 1$; $\Sigma_{j=1}^{n}\, z_j = 1$ $0 \leq z_1 \leq y_1$; $0 \leq z_n \leq y_{n-1}$ $0 \leq z_j \leq y_j + y_{j-1}$ for all $j = 2, \ldots, n-1$
Quantity q is a nonnegative integer variable with value less than or equal to positive integer U.	$y_j \triangleq 1$ if the 2^j bit in the binary representation of q is "on," 0 otherwise $N \triangleq \lfloor \log_2 U \rfloor$ $q = \Sigma_{j=0}^{N}\, 2^j y_j$ $q \leq U$

Although numerical modeling of discrete problems requires more abstraction than some other approaches, it offers a host of advantages. First, objective and constraint functions involving weighted sums and the like are easily expressed in terms of numerically valued variables. Second, methods and encodings evolved for pure problems extend naturally to mixed cases. Most important, however, is the fact that the available methods for continuous optimization are generally more effective than those for discrete problems (see Chapters 97 and 98). Embedding a discrete problem in a continuous environment makes it possible to exploit continuous approximations in the analysis.

Table 1 shows the standard modeling of a variety of discrete notions in terms of numerical decision variables. Throughout, y denotes discrete variables restricted to 0 and 1 values, z represents continuous variables, and M is a large positive constant.

2. SOLUTIONS

In optimization, a solution (choice of values for decision variables) is termed *feasible* if it satisfies all problem constraints and *optimal* if it is feasible and as good as any other feasible solution in objective function value. Models that have no feasible solutions are *infeasible*.

The goal of all discrete optimization analysis is to find feasible solutions with good objective function values. It is rarely important to know a mathematically optimal solution to a model, since the model is itself only an approximation to the underlying problem. However, it is desirable to have a sharp bound on the objective function value that might be obtained by any feasible solution. Then, for example, if a feasible solution to a maximize problem is known with objective function value \$92, and other analysis establishes that no feasible solution can produce better that \$100, one can accept the \$92 solution with confidence that it is no more $(100 - 92)/100 = 8\%$ suboptimal. The attraction of mathematically optimal solutions is that they provide good feasible solutions with zero-error bounds.

Many search methods for discrete optimization move through a sequence of *partial solutions* that assign specific values to some decision variables in a model, leaving others *free* or undetermined. A *completion* of a partial solution is an assignment of specific values to any remaining free variables. Of course, the definition of a partial solution includes the possibility that there are no free variables, in which case the solution is *complete*.

Each change in solution as a search proceeds is termed a *move,* and the collection of partial solutions reachable in one move from the current solution constitutes its *neighborhood*. Moves may change the value of a decision variable already assigned a value in the partial solution, or they may give a value to a previously free decision variable.

To illustrate, consider a pure discrete model with four decision variables y_1, y_2, y_3, y_4. One partial solution is $(y_1, y_2, y_3, y_4) = (1, 0, \#, \#)$, where # denotes a free value. Here only y_1 and y_2 have been assigned a specific value. Assume moves may either increase a single fixed variable from 0 to 1; or decrease a single fixed variable from 1 to 0; or assign the value 0 to a single free variable; or assign the value 1 to a single free variable. The neighborhood of the present solution is then the six partial solutions $(y_1, y_2, y_3, y_4) = (1, 1, \#, \#)$, $(0, 0, \#, \#)$, $(1, 0, 0, \#)$, $(1, 0, 1, \#)$ $(1, 0, \#, 0)$, and $(1, 0, \#, 1)$, reachable in one move.

3. TOTAL ENUMERATION

When there are only a few discrete-valued decision variables in a model, the most effective method of analysis is usually the most direct one: *total enumeration* of all the possibilities. For example, a model with only eight 0–1 variables could be enumerated by trying all $2^8 = 256$ combinations of values for the different variables. If the model is pure discrete, it is only necessary to check whether each possible assignment of values to discrete variables is feasible and to keep track of the feasible solution with best objective function value. For mixed models the process is more complicated because each choice of discrete values yields a residual optimization problem over the continuous variables. Each such continuous problem must be solved or shown infeasible to establish an optimal solution for the full mixed problem.

Although attractive for problems with only a few discrete decisions, enumeration becomes impractical as the number of discrete variables grows to even modest size. Each new 0–1 variable doubles the number of cases that must be considered. In the range of 100–150 discrete decisions, one can compute that this explosive number of cases could occupy the fastest imaginable computer longer than the estimated life of the universe.

4. RELAXATION

When dealing with difficult discrete optimization problems, it is natural to search for related, but easier optimization models that can aid in the analysis. *Relaxations* are auxiliary optimization problems of this sort formed by weakening either the constraints or the objective function of the main problem. Specifically, an optimization problem $(\tilde{P})$ is said to be a *constraint relaxation* of another optimization problem (P) if every solution feasible to (P) is also feasible for $(\tilde{P})$. Similarly, maximize problem (respectively minimize problem) $(\tilde{P})$ is an *objective relaxation* of another maximize (respectively minimize) problem (P) if the two problems have the same feasible solutions and the objective function value in $(\tilde{P})$ of any feasible solution is $\geq$ (respectively $\leq$) the objective function value of the same solution in (P).

To illustrate, consider the discrete optimization problem

$$
\begin{aligned}
&\min\ -2y_1 + 3y_2 + 6z_1 \\
(P)\ &\text{s.t. } y_1 + y_2 = 1 \\
&\qquad z_1 - 10y_1 \leq 0 \\
&\qquad y_1, y_2 = 0 \text{ or } 1,\ z_1 \geq 0
\end{aligned}
$$

Table 2 shows a variety of relaxations.

TABLE 2 Examples of Relaxations

Relaxation ($\tilde{P}$)	Reason
min $-2y_1$ s.t. $y_1 + y_2 = 1$ $z_1 - 10y_1 \leq 0$ $y_1, y_2 = 0$ or 1, $z_1 \geq 0$	Objective relaxation. New objective underestimates at feasible y_1, y_2, z_1.
min $-2y_1 + 3y_2 + 6z_1$ s.t. $y_1 + y_2 = 1$ $y_1, y_2 = 0$ or 1, $z_1 \geq 0$	Constraint relaxation. Second main constraint deleted.
min $-2y_1 + 3y_2 + 6z_1$ s.t. $y_1 + y_2 = 1$ $z_1 - 10y_1 \leq 0$ $1 \geq y_1, y_2 \geq 0, z_1 \geq 0$	Constraint relaxation. Feasible 0–1 values satisfy $1 \geq y_1, y_2 \geq 0$.
min $-2y_1 + 3y_2 + 6z_1$ $+100(z_1 - 10y_1)$ s.t. $y_1 + y_2 = 1$ $y_1, y_2 = 0$ or 1, $z_1 \geq 0$	Objective relaxation by term $100(z_1 - 10y_1) \leq 0$ at feasible solutions. Then constraint relaxation dropping $z_1 - 10y_1 \leq 0$.

Relaxations of discrete optimization problems aid in both the good solution finding and the bounding tasks of model analysis.

- An optimal solution to a relaxation can often be rounded or otherwise manipulated in a straightforward way to obtain a satisfactory solution for the main problem.
- The objective function value of an optimal solution to a relaxation bounds the optimal objective function value of the main problem. Specifically, relaxation optima provide lower bounds for minimize problems and upper bounds for maximize problems.
- An optimal solution to a relaxation is an optimal solution to the main problem if (1) it is feasible in the main problem and (2) its objective value in the main problem is the same as that in the relaxation.
- If a relaxation is infeasible, the main problem is infeasible.

4.1. Linear Programming Relaxations

An expression is *linear* if it consists of a weighted sum of variables + or − a constant. Most of the discrete modeling illustrated in Table 1 consists of linear objective functions and constraints. In follows that a great many discrete optimization problems can be effectively modeled as *integer linear programs* of the general form

$$\begin{aligned} &\min \text{ or } \max \sum_{j=1}^{n} C_j x_j \\ (ILP)\ \text{s.t.}\quad &\sum_{j=1}^{n} A_{ij} x_j \begin{Bmatrix} \geq B_i \\ = B_i \\ \leq B_i \end{Bmatrix} \quad \text{for all } i = 1, \ldots, m \\ &x_j \geq 0 \qquad\qquad \text{for all } j \notin J \\ &x_j = 0 \text{ or } 1 \qquad \text{for all } j \in J \end{aligned}$$

Here the C_j are given objective function coefficients, the A_{ij} are given coefficients of the m linear constraints, the B_i are constant terms of the linear constraints, and J is the subset of subscripts $j = 1, \ldots n$ indexing the 0–1 variables.

The *linear programming relaxation* of problem (*ILP*), denoted ($\overline{ILP}$), is the linear program obtained when the last 0–1 system of constraints is replaced by

$$1 \geq x_j \geq 0 \text{ for all } j \in J$$

The third relaxation of Table 2 provides an example.

Because linear programs are the best solved of all optimization problems (see Chapter 97), linear programming relaxations can be optimized for very large models. For this reason, (*ILP*)s are by far the most commonly employed relaxations, and they form the core of most commercial software for discrete optimization.

Not all linear programming relaxations are close approximations of the discrete problem of interest. Because very complex criterion functions and constraints can be modeled by the linear part of an *ILP,* however, it is often the case that an optimal solution to the relaxation (*ILP*) provides a sound starting point for construction of a good feasible solution to the discrete problem. Such constructions may loosely be termed *rounding*.

Let an optimal solution to LP relaxation (*ILP*) be denoted by $\bar{x}_j$, $j = 1, \ldots, n$. The easiest case of rounding, yet one that still applies in many models, is where the model permits fractional $\bar{x}_j$ that should be integer (i.e., $j \in J$) to be simply rounded up to the next integer (often denoted $\lceil \bar{x}_j \rceil$) or rounded down to the next integer ($\lfloor \bar{x}_j \rfloor$). General-purpose rounding algorithms are also available, notably Balas and Martin's (1980), pivot and complement procedure. For most cases, however, the methods are dependent on the form of the model.

In a number of very important cases, (*ILP*)s have properties that guarantee there will always be an optimal solution to the corresponding (*ILP*) with integer values for variable x_j with $j \in J$. That is, the full model (*ILP*) can be solved optimally by solving the linear relaxation (*ILP*) because its only relaxed constraints (x_j integer for $j \in J$) are automatically satisfied by an LP optimum.

The most common of these exact cases are optimization problems that can be modeled as single-commodity *network flows* (see Chapter 99). Equivalently, these are the (*ILP*)s that can be written so that for each variable x_j, at most one constraint coefficient A_{ij} equals 1, at most one A_{ij} equals -1, and all other A_{ij} equal 0. Such (*ILP*)'s are *totally unimodular* in that any submatrix formed by the $\{A_{ij}\}$ associated with a collection of rows i and a like-sized collection of variables j has determinant 0, 1 or -1. This is enough to ensure optimal basic solutions to (*ILP*) (produced, for example, by the simplex algorithm for linear programming) are integer whenever right-hand-side coefficients B_i are all integer.

Another important class of (*ILP*)'s with integer optimal solutions to their linear relaxations is those that are *totally dual integer* (TDI). An (*ILP*) with integer constraint coefficients A_{ij} is TDI if its linear programming dual (see Chapter 97) has an integer optimal solution for every integer choice of objective function coefficients C_j. Models that are TDI will have integer optimal solutions to their linear programming relaxations if all right-hand-side constants B_i are integer.

When linear programming relaxations are not exact, it is important to make them as sharp an approximation as possible. Different formulations of discrete models as (*ILP*)s can produce quite different linear programming relaxations, even though the models have the same discrete solutions.

To illustrate, consider a problem to

$$\begin{aligned} \min \quad & 30z_1 + 60z_2 - 100z_3 \\ \text{s.t.} \quad & (z_3 = 1 \textit{ only if } z_1 = z_2 = 1) \\ & z_1, z_2, z_3 = 0 \text{ or } 1 \end{aligned}$$

If the specified logical requirement is modeled as $z_1 + z_2 \geq 2z_3$ the optimal solution to the linear programming relaxation is $z_1 = 1$, $z_2 = 0$, $z_3 = 1/2$ with objective function value -20. An equivalent (*ILP*) with a sharper (*ILP*) comes from modeling the requirement as $z_1 \geq z_3$, $z_2 \geq z_3$. In this format, an optimal solution to the linear programming relaxation is $z_1 = z_2 = z_3 = 1$ with objective value -10. The latter form is stronger because the relaxation solution obtained is both more nearly feasible for the discrete problem and the source of a more exact bound. (In fact, the second modeling yields an optimal solution for this instance.)

Because of the importance of using formulations with sharp linear programming relaxations, a great deal of research in the recent decades has been addressed to various aspects of that issue. One broad area of opportunity for sharpening linear programming relaxation comes in choosing constraint coefficients more carefully. Assume that each linear inequality of the formulation is rearranged to $\geq$ format with only the constant term on the right-hand side. Each inequality is then of the form $\sum_{j=1}^{n} A_{ij}x_j \geq B_i$. Recalling that we also assume all variables are subject to nonnegativity constraints, it is easy to see that this constraint will cut off more LP-feasible points if either B_i can be increased or at least one of the A_{ij} can be decreased.

A classic example relates to large-integer "big M" constants in (*ILP*) formulations. For example, suppose continuous variable z_1 is subject to constraints $0 \leq z_1 \leq 10$ and corresponding 0–1 variable y_1 is to be 1 whenever z_1 is positive. The latter requirement is modeled $My_1 - z_1 \geq 0$, where M is any positive constant at least 10. However, the modeling with the strongest linear programming relaxation will be the one that makes M as small as possible (i.e., $M = 10$).

A second common method of improving linear programming relaxations is to add new valid inequality constraints. An inequality is *valid* for an (*ILP*) if it is satisfied by every (integer) feasible

solution to (*ILP*). Technically all the original constraints of (*ILP*) are valid inequalities. However, the term usually refers to added constraints that are not needed for a correct integer linear programming formulation but do sharpen the corresponding linear programming relaxation. The strongest such valid inequalities—ones that are unavoidable if the sharpened LP relaxation is to be exact—are called *facet-inducing, facetial,* or simply *facets*.

It is easy to find valid inequalities for discrete optimization formulations but difficult to find ones strong enough to materially improve the LP relaxation. Most of the families of such inequalities that have proved useful are highly problem specific.

One fairly general case is the family of valid inequalities employed by Crowder et al. (1983), when the original (*ILP*) formulation includes a constraint $\Sigma_{j\in L} A_{ij}x_j \leq B_i$ where nonzero coefficients occur only on integer subscripts in $L \subseteq J$, all coefficients A_{ij} and B_i are positive, and the A_{ij} are not just ones. For any $K \subseteq L$ with $\Sigma_{j\in K} A_{ij} > B_i$, it is easy to see the following inequality is valid $\Sigma_{j\in K} x_j \leq |K| - 1$ ($|K|$ denotes the number of subscripts in K). Furthermore, the smaller we can make K while satisfying its defining requirement, the sharper the revised linear programming relaxation. Choosing minimal K, that is, K that cannot be further reduced, produces a useful family of valid inequalities.

One difficulty with using families of strong valid inequalities to sharpen linear programming relaxations is that the number of inequalities in such families is usually exponentially large. It would be impossible actually to enumerate all such inequalities and add them to the formulation submitted to a linear programming code. Instead, successful applications have employed *separation* subroutines. Such subroutines heuristically select a small subfamily of inequalities that are likely to improve the current formulation. After solving the linear programming relaxation with the selected inequalities added, if the approximation is still not sharp enough, the separation routine may be reinvoked to generate further inequalities.

A third approach to sharpening linear programming relaxations has been to introduce an extended (larger) set of variables (see Martin 1999). Such extra variables are not necessary for a correct (*ILP*) formulation. Still, their presence in the model makes it possible to write new constraints that sharpen the linear programming relaxation.

To illustrate, consider a fixed charge network flow problem with two candidate locations for \$50 thousand facilities, each with a demand for 10 thousand units of the commodity to be provided by the facilities. If there is a \$1 per unit shipping charge between the facilities, a textbook formulation is

$$\begin{aligned} \min \quad & z_{12} + z_{21} + 50y_1 + 50y_2 \\ \text{s.t.} \quad & z_1 = z_{12} + 10,\ z_2 = z_{21} + 10 \\ & z_1 \leq 20y_1,\ z_2 \leq 20y_2 \\ & z_1, z_2, z_{12}, z_{21} \geq 0;\ y_1, y_2 = 0 \text{ or } 1 \end{aligned}$$

Here z_i is the total supplied at facility i, z_{ij} represents the number of thousands of units shipped to the other facility, and y_i decides whether facility i is built. This model has LP relaxation optimal solution $\bar{z}_1 = \bar{z}_2 = 10$, $\bar{z}_{12} = \bar{z}_{21} = 0$, $\bar{y} = \bar{y}_2 = 1/2$ and value \$50 thousand.

The extended form of Rardin and Wolsey (1993) introduces new variables that subdivide flows z_1 and z_2 according to whether they are directed to the facility site or to its companion site. Specifically,

$$\begin{aligned} z_1 &= w_{11} + w_{12},\ z_2 = w_{21} + w_{22} \\ z_{12} &= w_{12},\ z_{21} = w_{21} \\ 0 &\leq w_{11} \leq 10y_1,\ 0 \leq w_{12} \leq 10y_1 \\ 0 &\leq w_{21} \leq 10y_2,\ 0 \leq w_{22} \leq 10y_2 \end{aligned}$$

With this extended system, the LP relaxation yields a discrete optimum of value \$60 thousand.

4.2. Lagrangean Relaxations

Linear programming relaxations of integer linear programs produce an easier problem by deleting difficult integrality requirements on variables x_j with $j \in J$. This makes them as widely applicable as integer linear programming itself, but they are not always very good approximations.

Lagrangean relaxations are an alternative appropriate where some of the linear constraints are treated as the complications in an otherwise manageable discrete model. Integrality requirements are explicitly retained in relaxations, and complicating linear constraints are *dualized,* that is, taken to the objective function with appropriate Lagrange multipliers.

The fourth relaxation of Table 2 illustrates the principle of Lagrangean relaxation, but most of the success with Lagrangean relaxation has derived from problem-specific structures. Its power is better illustrated by a classic application.

Generalized assignment problems involve optimal arranging of objects $i = 1, \ldots, m$ of known size S_i into locations $j = 1, \ldots, n$ of known capacity K_j. An (*ILP*) formulation is

$$\begin{aligned} \min \quad & \sum_{i=1}^{m}\sum_{j=1}^{n} C_{ij}y_{ij} \\ (GA)\ \text{s.t.} \quad & \sum_{j=1}^{n} y_{ij} = 1 \qquad \text{for all } i = 1, \ldots, m \\ & \sum_{i=1}^{m} S_i y_{ij} \le K \quad \text{for all } j = 1, \ldots, n \\ & y_{ij} = 0 \text{ or } 1 \quad \text{for all } i = 1, \ldots, m;\ j = 1, \ldots, n \end{aligned}$$

where C_{ij} is the cost of assigning object i to location j.

Either of the main systems of constraints in (*GA*) might be thought of as complicating because deletion of either leaves an easier model where each variable appears in only one constraint. Thus, either system might be first dualized and then dropped. Corresponding Lagrangean relaxations are

$$\begin{aligned} \min \quad & \sum_{i=1}^{m}\sum_{j=1}^{n} C_{ij}y_{ij} + \sum_{j=1}^{n} u_j\left(\sum_{i=1}^{m} S_i y_{ij} - K_j\right) \\ (GA_u)\ \text{s.t.} \quad & \sum_{j=1}^{n} y_{ij} = 1 \text{ for all } i = 1, \ldots, m \\ & y_{ij} = 0 \text{ or } 1 \text{ for all } i = 1, \ldots, m;\ j = 1, \ldots, n \end{aligned}$$

and

$$\begin{aligned} \min \quad & \sum_{i=1}^{m}\sum_{j=1}^{n} C_{ij}y_{ij} + \sum_{i=1}^{m} v_i\left(\sum_{j=1}^{n} y_{ij} - 1\right) \\ (GA_v)\ \text{s.t.} \quad & \sum_{i=1}^{m} S_i y_{ij} \le K_j \text{ for all } j = 1, \ldots, n \\ & y_{ij} = 0 \text{ or } 1 \text{ for all } i = 1, \ldots, m;\ j = 1, \ldots, n \end{aligned}$$

The effect of dualizing constraints is to enforce them partially by penalizing violating solutions in the objective function. However, care must be taken to ensure that a valid relaxation results. In (GA_u), the dualized constraints are inequalities. Terms weighted by u_j will be negative or zero at feasible solutions. Thus, in order to have a proper objective relaxation, multipliers must satisfy

$$u_j \ge 0 \text{ for all } j = 1, \ldots, n$$

Furthermore, an optimal solution to a relaxation (GA_u) may not be optimal in (*GA*) even if it satisfies the relaxed constraints. Since the objective function values in (*GA*) and (GA_u) may differ, *complementary slackness* conditions

$$u_j\left(\sum_{i=1}^{m} S_i y_{ij} - K_j\right) = 0 \text{ for all } j = 1, \ldots, n$$

must also be satisfied if a relaxation optimum is to be optimal in (*GA*).

In (GA_v), the situation is much easier because dualized constraints are equalities. Terms weighted by v_i will be zero at feasible solutions. Thus, no sign restrictions on Lagrange multipliers are needed for a proper relaxation, and complementary slackness is not an issue for (*GA*) optimality.

Any valid choice of multipliers on dualized constraints produces a Lagrangean relaxation, and like all relaxations the optimal objective function value in the relaxation bounds the optimal value of the main problem. However, some choices of constraints to dualize may give rather weak bounds; others may yield very strong bounds (see Parker and Rardin 1988, chap. 5 for a full discussion).

For any fixed dualization strategy, good bounds depend on good multipliers. Successful use of Lagrangean relaxation requires a search where a sequence of Lagrangean relaxations is solved with different multipliers. Results of one relaxation suggest ways to change the multipliers for the next.

The simplest, *subgradient search,* is effective in many settings. Assume the given discrete problem has been expressed as a minimize (*ILP*) with all inequalities $\leq$, $I^{=}$ the collection of dualized equality row numbers, $I^{\leq}$ the collection of dualized ($\leq$) inequalities, and $\{u_i : i \in I^{=} \cup I^{\leq}\}$ the Lagrange multipliers. Then subgradient search updates multipliers:

$$u_i \leftarrow u_i + \alpha \left(\sum_{j=1}^{n} A_{ij}\hat{x}_j - B_i \right) \text{ for all } i \in I^{=} \cup I^{\leq}$$

$$u_i \leftarrow \max\{u_i, 0\} \text{ for all } i \in I^{\leq}$$

where $\hat{x}_j$ denotes the most recent relaxation optimum and α is a stepsize.

5. HEURISTIC SEARCH

When a discrete model has too many decisions for total enumeration and no convenient relaxation that is sharp enough to give sound approximations, the principal remaining approach is to organize a search through a series of solutions (or partial solutions) until a satisfactory feasible solution is isolated.

Sometimes a practical method is available for computing an exact optimum (see Section 6). More commonly, however, only a *heuristic* or *approximate optimum* can be obtained in reasonable computation time. Such solutions are feasible, but there is no guarantee they are optimal. Often it is not even possible to bound the error in the heuristic optima produced.

Implementations of heuristic search in discrete optimization are usually classified according to whether they focus on partial or complete solutions. *Constructive searches* begin with an all-free partial solution and fix one or more components at each move until a complete solution is obtained. *Improving searches* work their way through a sequence of complete solutions, striving at each move to find a complete solution in the neighborhood that is either less infeasible, better in objective value than the current solution, or both. Of course, the strategies can be combined by, for example, using a constructive search to build a first complete solution and then applying an improving search to make it better.

5.1. Constructive or Solution-Building Search

Constructive searches are often called *greedy* or *myopic* because they usually choose variables and values to fix on the basis of estimates of immediate or short-term gain. The main issue in design of such procedures is to choose an informative measure of efficiency or gain on which to base selections.

For certain maximizing models on combinatorial structures called *matroids* (see e.g., Parker and Rardin 1988, chap. 3), an exact optimal solution results from the most naive possible greedy choice rule. Moves iteratively make $=1$ the remaining free variable with greatest objective function coefficient, subject only to the requirement that this choice does not produce infeasibility. The *spanning tree problem* (see the Appendix) is the most famous example of this matroid structure where a greedy algorithm is optimal.

Constructive searches based on more complicated efficiency ratios are much more common than the pure greedy notion of considering only objective coefficient magnitudes. One example is Dobson's (1982), heuristic for *generalized covering* problems of the form

$$\begin{aligned} \min \quad & \sum_{j=1}^{n} C_j y_j \\ \text{s.t.} \quad & \sum_{j=1}^{n} A_{ij} y_j \geq B_i \quad \text{for all } i = 1, \ldots, m \\ & y_j = 0 \text{ or } 1 \quad \text{for all } j = 1, \ldots, n \end{aligned}$$

where all A_{ij} and B_i are nonnegative integers. Dobson's algorithm starts with the all free solution and iteratively chooses a new $y_{\tilde{j}}$ to make $= 1$ where $\tilde{j}$ is the free index with $\min\{C_j / \Sigma_{i=1}^{m} \min\{A_{ij}, \tilde{B}_i\}\}$ and $\tilde{B}_i = \max\{0, B_i - \Sigma_{j \text{ fixed } 1} A_{ij}\}$. The denominator of this ratio is the total overall constraints of the amount variable j could contribute toward resolving remaining infeasibility. Thus, the algorithm is choosing on the basis of least cost per unit improvement in infeasibility. The process terminates (making any remaining free variables $= 0$) when the current partial solution is feasible in every main constraint.

The fact that constructive heuristics usually terminate as soon as the first feasible solution is found makes them an attractive choice where solution time is critical, such as in near real-time control. Still, results obviously depend critically on the early decisions taken, so the quality of the heuristic optimum produced tends to deteriorate rapidly with the number of discrete variables in the model.

5.2. Improving or Solution-Enhancing Search

Improving searches begin from a complete solution and apply moves that either improve the objective function value or reduce infeasibility. All or a large part of the complete solutions in the neighborhood are evaluated, and the search advances to the one found most attractive. One example is the *parallel processor* problem:

$$\begin{aligned} \min \quad & z \\ \text{s.t.} \quad & y_{i1} + y_{i2} = 1 \quad \text{for all } i = 1, \ldots, m \\ & \sum_{i=1}^{m} T_i y_{ip} \le z \quad \text{for all } p = 1, 2 \\ & y_{ip} = 0 \text{ or } 1 \quad \text{for all } i = 1, \ldots, m;\ p = 1, 2 \end{aligned}$$

Here tasks $i = 1, \ldots, m$ of time duration T_i must be scheduled on one of two processors p. Decision variables $y_{ip} = 1$ if task i is assigned to processor p and $= 0$ otherwise. Continuous variable z measures the completion time of all tasks.

An improving search would begin from any feasible choice of values for the discrete variables (each task is assigned to some processor). A major algorithm-design question is what other solutions should be considered neighbors, or equivalently, what set of moves to apply.

One obvious family of moves consists of switching one task from its current processor to the other. Whichever of these changes most improved the objective function would provide the move to be taken.

A neighborhood allowing pairs of tasks to be interchanged between the two processors could yield better results. However, the computational effort per step would increase because there are only m reassignments of one task to a different processor, but more like $m^2/4$ pairwise swaps (assuming about half the tasks will be on each processor).

Still another neighborhood might employ moves that delete or add a single task to one of the processors. Such moves could create infeasibility because a task might be assigned to both processors, or it might not be assigned at all. The difficulty in this case is how to balance improvement in solution value with reduction in infeasibility as the next solution is chosen. A common approach is to add a penalty term in the objective function to discourage infeasibility without prohibiting it. In the parallel processor example above, this penalty term would have the form

$$\alpha \sum_{i=1}^{m} |y_{i1} + y_{i2} - 1|$$

where $\alpha > 0$ is the weight applied to infeasibility.

5.3. Local Optima

Although heuristic search has proved useful on some problems, natural definitions of neighborhoods often lead to poor *local optima,* that is, final feasible solutions that cannot be improved in the neighborhood. Of course, a richer family of allowed moves would make it possible to reach stronger local optima, but the cost of examining the neighborhood at each iteration rapidly becomes prohibitive.

One standard solution to this dilemma, known as *multistart,* is to employ a limited neighborhood but restart the search several times. Each time, the search begins with a randomly chosen starting solution and continues to a local optimum. The best of these local optima is kept as an heuristic optimum.

5.4. Tabu, Simulated Annealing, and Genetic Algorithms

An alternative that has generated much recent research interest is to liberate neighborhood search from the obligation to improve at each step. That is, moves are sometimes adopted that do not improve the objective function (or reduce infeasibility). Comprehensive books on the topic include Reeves (1993), Aarts and Lenstra (1997), and Glover and Laguna (1998).

The immediate difficulty with nonimproving moves is that they can make the search loop. For example, suppose that no improving move is available and a nonimproving move is adopted to change y_{27} from $= 1$ to $= 0$. Assuming some symmetry in the move set, there will certainly be an improving

move at the next step: changing y_{27} back from = 0 to = 1. The algorithm could loop infinitely between the two.

Tabu algorithms of Glover and others deal with repeats by keeping list of moves that are temporarily "tabu" or forbidden. The best improving (or least nonimproving) non-tabu move is adopted at each step of the procedure. For example, in the above case where y_{27} is switched from = 1 to = 0, any move involving y_{27} might be placed on the tabu list for say the next 5–10 steps and then freed. Solutions can still repeat under tabu, but computational experience has shown promise in a number of applications.

From this simple beginning, tabu methods have developed in a variety of directions. Once a data structure must be maintained to limit moves, it can be used to guide other aspects of the search in a variety of creative ways. For example, moves that have proved useful in the past may be given some preference, or moves that have not been used in the most recent part of the search might be tried. Glover and Laguna (1998) develop a host of alternatives.

A second approach is the stochastic one of *simulated annealing* (see, e.g., Kirkpatrick et al. (1983); Aarts and Lenstra (1998)). With simulated annealing, a move is selected randomly from the available neighborhood at each iteration. If the selected move would result in an improvement, it is adopted and the search advances to the indicated solution. If the move would degrade the solution, it may or may not be adopted, depending on a probability that decreases with the magnitude of the degradation.

A common rule is to accept a nonimproving move with probability $e^{-d/T}$, where d is the amount by which the solution degrades the objective function value and T is a control parameter called the *temperature*. Typically T is started relatively large so that the search can range widely in the early stages. Then T is slowly decreased as the procedure settles into the region of a good feasible solution.

As with tabu, solutions can repeat, but simulated annealing's use of probabilities ensures the search will advance to better solutions if any exist, although it may take a long time. Experience has shown simulated annealing to be a reliable and easy-to-implement way to compute good heuristic solutions in a wide variety of applications. However, comparatively long running times are often required.

Genetic algorithms (see Holland 1975; Goldberg 1989) offer still another approach to dealing with local optima. Instead of keeping just a single current solution at each move, these methods retain a whole *population* of solutions. At each update or *generation,* some or all of these solutions will be replaced by improved ones.

Any of the normal manipulations of neighborhood search can be employed to construct the new solutions, but *crossover* moves, which interchange parts of solutions in the current population, are the most popular. For example, crossover of "parent" solutions (0,1,0,0,1,1) and (1,1,0,1,0,1) by cutting after the third component would produce "offspring" (0,1,0,1,0,1) and (1,1,0,0,1,1). These new solutions would be evaluated and the better ones preserved in the next generation.

Genetic algorithms have become the method of choice in difficult engineering design circumstances where complex feasibility limitations and massive nonlinearity make it difficult to employ neighborhood-based methods. However, other methods usually give better performance on classic combinatorial optimization problems—especially (*ILP*)'s and cases with linear constraints.

6. BACKTRACKING SEARCH AND BRANCH AND BOUND

Heuristic searches usually guarantee neither a global optimal solution nor a bound on the error when computation stops because they make *preemptive* moves—moves for which there are viable alternatives that are not explored. In order to carry out a more exhaustive search, such moves must be taken as *provisional*. That is, a record must be retained of the alternatives not yet pursued, and search must *backtrack* to those alternatives, pursuing them until they prove incapable of producing an satisfactory solution. As the search encounters a particular partial solution, it either *terminates* that solution, that is, finds it to admit no improving move, or *branches* it, that is, extends it by one applicable move. The best feasible solution encountered is recorded as the *incumbent solution*. Thus, if the search exhausts all open alternatives, the incumbent solution is a global optimum.

6.1. Tree Representation

One essential element of backtracking search is a record of provisional moves and alternatives not yet explored. The most convenient format for such a record is a tree like the one in Figure 1. *Nodes* of that tree represent states of the search. *Branches* show the selected and alternative moves available in a state. Search proceeds from the first node or *root* of the tree toward the bottom. A node is numbered when it is actually visited by the search; ones still to be considered are left unnumbered. A backtrack occurs thenever the search skips to an open alternative instead of an extension of the current state.

The example of Figure 1 represents a search that has already completed 6 nodes. Successive moves from the root=1 fixed y_5, then y_{11}, then y_3 to 1. The last produced a feasible solution and node 4, which is terminated. The search now backtracks to one of the three unexplored alternatives

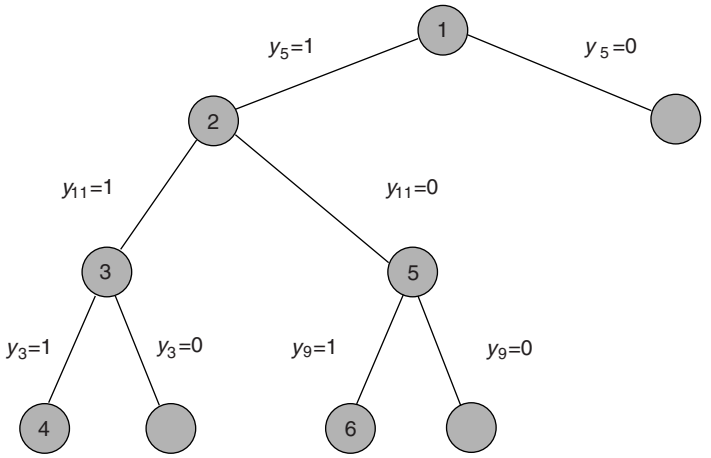

Figure 1 Backtracking Search Tree.

to the moves taken. Here the choice was to adopt move $y_{11} = 0$ to the node marked 5. Moves $y_9 = 1$ and $y_9 = 0$ were now available at this node, so it was branched and the first of these moves was selected.

6.2. Branch and Bound

Branch and bound procedures combine backtracking search with the power of relaxations. Any partial solution in a search that has variables still free defines a *candidate problem,* that is, a discrete optimization problem over the free variables subject to limits imposed by the fixed decisions. Instead of pursuing partial solutions until no further moves are available, branch and bound solves a relaxation of the corresponding candidate problems. If the relaxation optimum satisfies requirements to be optimal for the candidate problem, the partial solution can be terminated immediately; its best completion has been identified. If the relaxation proves infeasible, the partial solution can also be terminated; no completion exists. When neither of these cases occurs, the value of the relaxation optimal solution provides a bound on the value of the candidate problem. That is, it yields a bound on the quality of any completion. If that bound is already worse than the incumbent solution, no completion can improve on the incumbent; the node can be terminated. In any event, the best such bound across all unexplored nodes provides a global bound on the optimal value of the full discrete model.

Any reasonable set of branching moves can be combined with any convenient relaxation to produce a branch and bound procedure. Still, the great majority of successful applications and all commercial codes for discrete optimization use implementations of branch and bound on integer linear programs (*ILP*), with linear programming relaxations. The main ideas of such an algorithm can be outlined as follows:

ILP branch and bound (minimize):
initialize $CAND \leftarrow \{(ILP)\}$; $INCUM \leftarrow \infty$; $k \leftarrow 0$
while $CAND \neq \emptyset$ **do**
  $k \leftarrow k + 1$
  select a member of $CAND$ as (ILP_k)
  attempt to solve relaxation $(\overline{ILP_k})$ for value VAL_k
  if $(\overline{ILP_k})$ is infeasible or $VAL_k \geq INCUM$
    then delete (ILP_k) from $CAND$
    elseif the $(\overline{ILP_k})$ optimum is feasible in (ILP)
      then $INCUM \leftarrow \min\{INCUM, VAL_k\}$
        delete any member of $CAND$ with stored bound $\geq INCUM$
        delete (ILP_k) from $CAND$
    else choose binary free x_p fractional in $(\overline{ILP_k})$
      replace (ILP_k) in $CAND$ by extensions with $x_p = 0$
        and $x_p = 1$, both with stored bound VAL_k
end

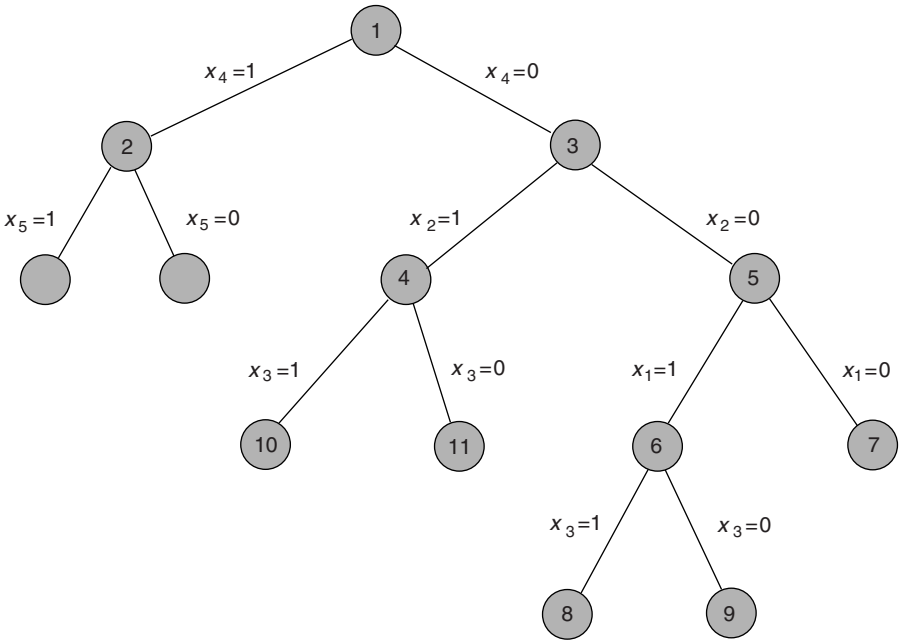

Figure 2 Branch and Bound Example.

In this statement, *CAND* represents a list of the candidate problems associated with active nodes of the search. *Stored bounds* are maintained with members of *CAND* to show the best-known lower bound on the value of an optimal solution to the candidate.

To illustrate this (*ILP*) form of branch and bound, consider the example

$$\begin{aligned} \min \quad & 7x_1 + 12x_2 + 7x_3 + 14x_4 + x_5 + 25x_6 \\ \text{s.t.} \quad & 3x_1 + 6x_2 + 5x_3 + 16x_4 + x_6 \geq 7 \\ & 5x_1 + 4x_2 + 4x_3 + 5x_4 \geq 2 \\ & 3x_1 + 3x_2 + 3x_3 + 10x_5 \leq 3 \\ & x_1, \ldots, x_5 = 0 \text{ or } 1;\ 0 \leq x_6 \leq 3 \end{aligned}$$

Figure 2 shows the search tree for this example, and Table 3 describes processing of each node.

A complete implementation of branch and bound in even the (*ILP*) form given above involves many heuristic rules. The primary ones are (1) which free variable to fix in branching node k and (2) which active node (member of *CAND*) to choose at each iteration k. For the simple example above, corresponding rules were (1) choose the fractional-valued free variable closest to 0.0 and (2) choose the active node with least stored bound, breaking ties in favor of the last fixed variable = 1.

A desirable feature of branch and bound is that a bound on the value of a global optimal solution is always available, so that the algorithm need not be run to termination in order to bound the error of accepting the incumbent solution as approximately optimal. The best stored bound of *CAND* (least for minimize problems, highest for maximize problems) always provides such a bound. Thus, for example, after node 8 is processed in the example of Figure 2, it is certain that any solution to the full (*ILP*) will cost at least $\min\{12.6,13.4\} = 12.6$. Stopping at that point with the incumbent solution of value 14 would produce at most $(14 - 12.6)/12.6 = 11.1\%$ error.

7. GUIDELINES AND LIMITS

Discrete optimization problems abound in all phases of industrial engineering, and ones with important economic implications easily justify formal modeling and systematic analysis. However, there are no general-purpose methods appropriate for dealing with all or even most models. This concluding

TABLE 3 Processing for Branch and Bound Example

Node	Fixed	LP Relaxation	*VAL*	Processing
1	none	(0,0,0,0.44,0.3,0)	6.42	branch on x_4
2	$x_4 = 1$	(0,0,0,1,0.3,0)	14.30	branch on x_5
3	$x_4 = 0$	(0,0.33,1,0,0,0)	11	branch on x_2
4	$x_4 = 0, x_2 = 1$	(0,1,0.2,0,0,0)	13.4	branch on x_3
5	$x_4 = 0, x_2 = 0$	(0.67,0,1,0,0,0)	11.67	branch on x_1
6	$x_4 = 0, x_2 = 0, x_1 = 1$	(1,0,0.8,0,0,0)	12.6	branch on x_3
7	$x_4 = 0, x_2 = 0, x_1 = 0$	(0,0,1,0,0,2)	57	new incumbent; *INCUM* ← 57; terminate solved
8	$x_4 = 0, x_2 = 0, x_1 = 1, x_3 = 1$	(1,0,1,0,0,0)	14	new incumbent; *INCUM* ← 14; delete extensions of Node 2; terminate solved
9	$x_4 = 0, x_2 = 0, x_1 = 1, x_3 = 0$	infeasible	—	terminate infeasible
10	$x_4 = 0, x_2 = 1, x_3 = 1$	(0,1,1,0,0,0)	19	terminate bound $\geq$ 14
11	$x_4 = 0, x_2 = 1, x_3 = 0$	(0.33,1,0,0,0,0)	14.33	terminate bound $\geq$ 14; incumbent solution optimal

section reviews theory and accumulated wisdom delimiting which of the approaches treated in previous sections are appropriate for a given model.

7.1. Computational Complexity Theory

The formal theory of problem difficulty classification is called *computational complexity theory* (see Garey and Johnson 1979; Papadimitriou 1994; Parker and Rardin 1988, chap. 2). Complexity theory terms a *problem* any collection of related *instances* distinguished only by their size and numerical constants. For example, (*ILP*) is a problem with instances distinguished by counts m and n, discrete variable list J, and constants C_j, A_{ij}, and B_i. The *size* of an instance is the length of the symbol string required to encode it for a computer.

The objective of complexity theory is to classify problems according to how efficiently instances can be solved relative to their size. Bounds on the required computation are expressed as *computational orders,* denoted $O(\ )$. For example, if every instance of a problem can be solved in a number of elementary calculations bounded by the square of its size s, the problem is $O(s^2)$ solvable. More generally, a problem is said to be *polynomially solvable* if there is a constant k such that every instance of size s is solvable in $O(s^k)$ effort. Specifically, $O(s^4)$, $O(s\sqrt{s})$, and $O(s^2 \log s)$ computations are polynomial (the last because $s^2 \log s \leq s^3$). $O(2^s)$ and $O(s!)$ are not polynomial.

One of the most important theoretical achievements of discrete optimization research has been to isolate polynomial solvability as the defining characteristic of truly tractable discrete problems. Every one of the discrete models for which a generally effective algorithm or an exact (polynomially solvable) relaxation has been discovered belongs to the polynomially solvable class.

Across what seems to be a cosmic boundary in mathematics lies the alternative *NP-hard* class, to which almost all discrete optimization models belong that lack such tractable characteristics. *NP*-hard problems are not (yet) provably outside the reach of polynomial solvability, but neither are they just problems for which research has so far failed. Class *NP* is a vast collection of ''Does there exist . . .'' problems in discrete mathematics and computer science, some of which have been studied for centuries without much progress. No discrete optimization problem actually belongs to *NP* because none is of this existence form. Still, each *NP*-hard discrete optimization problem (*H*) is as difficult as any in *NP* in the sense that a polynomial algorithm for (*H*) would provide one for every member of *NP*. Thus, to seek such an algorithm for any *NP*-hard problem is simultaneously to attack the enormous variety of challenging problems in *NP*. Success is most unlikely. Similar logic establishes that exact relaxations and strong duality theories are also highly improbable for most *NP*-hard problems, although additional technical issues make general statements more difficult.

The fundamental importance of the polynomially solvable vs. *NP*-hard distinction makes it a high-priority matter to try to determine on which side of the boundary any given discrete optimization application falls (theory indicates there may be problems that belong to neither category, but few realistic candidates are known). Polynomial solvability is almost always established by showing the given model is a special case of one of the classic polynomial time models detailed in the Appendix,

or inventing some simple enumeration over cases solvable as one of those models. On the other hand, one shows a problem is *NP*-hard by demonstrating that some known *NP*-hard problem can be viewed as a special case (several *NP*-hard models are included in the Appendix, and Garey and Johnson 1979 provides a much bigger list). For example, generalized assignment (*GA*) is known to be *NP*-hard. This is enough to establish that integer linear programming (*ILP*) is *NP*-hard because every instance of (*GA*) is an instance of (*ILP*).

7.2. Choosing a Strategy

When a discrete optimization problem is in the polynomially solvable complexity category, it is usually clear how to proceed with its analysis. A clever and efficient algorithm is at hand. Often an exact linear programming formulation is also known, and a strong duality theory is available for sensitivity studies. Very complete analysis should be possible, unless limits on the time available for solution (e.g., in a real-time setting) mandate quicker methods.

In the far more typical case where the given discrete optimization is *NP*-hard, more care should be exercised in choosing avenues of analysis. Figure 3 offers some very approximate guidelines in terms of two critical characteristics: the number of binary decisions in the model vs. the error of the best available relaxation bound.

For small numbers of decisions, say up to 10–15, total enumeration is recommended, regardless of relaxation quality. As the number of binary decisions increases, models subdivide into three regions. In the outermost, relaxations are too poor to assist in analysis. The only practical strategy is some form of heuristic search lacking even a bound on the suboptimality of results obtained.

The innermost region shows where branch and bound methods may be effective. Relatively strong relaxations are required, and the needed sharpness increases rapidly with the number of binary decisions.

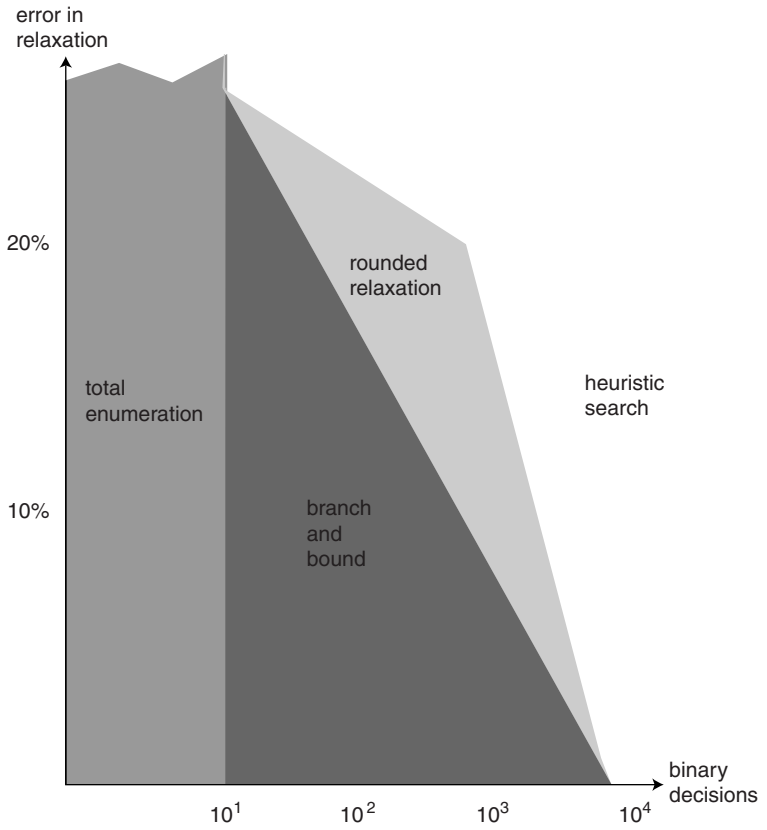

Figure 3 Guidelines for NP-Hard Problems.

Between these two extremes lies a region where relaxations may be helpful but branch and bound is unlikely to be effective. Here, bounds from relaxations at least delimit the suboptimality of solutions found through heuristic search or other means. Relaxation may also provide a good source of feasible discrete solutions when the problem admits easy rounding of relaxation optima.

As with any broad guidelines, the boundaries in Figure 3 are very fuzzy. In particular, it is often too simple to take the number of binary decisions in a model to equal the number of 0–1 variables in an (*ILP*) formulation. A generalized assignment model, for example, with m objects and n locations has mn 0–1 variables. However, each belongs to $\Sigma_j y_{ij} = 1$ multiple-choice set. Thus, there are really only n choices in each of these subsets, or $m \log_2 n$ total binary decisions in the model.

Application of the guidelines in Figure 3 also requires, of course, information about the likely quality of available relaxations. Each family of examples has different behavior, but there are characteristic attributes of relaxations with little promise:

- Nonlinear objective functions or constraints
- Either–or constraints that can only be placed in (*ILP*) format through the use of large positive constants (''big *M*'s'')
- Massive symmetry introducing numerous feasible alternatives of nearly the same objective function value

Any relaxation possessing such attributes is likely to require strengthening if it is to be of practical use with even moderately large applications.

APPENDIX

This appendix briefly describes and formulates a variety of standard discrete optimization models. Throughout, y denotes 0–1 discrete variables, z represents continuous variables, $D(N, A)$ indicates a directed network or graph with nodes in N and arcs in A, and $G(N, E)$ denotes an undirected network or graph with nodes in N and edges in E. Capital letters are used to denote input and lower case to show decision variables. The size of set S is indicated by $|S|$.

Assignment: Maximum utility pairing of objects from two given sets S and T; $\triangleq$ family of allowed pairs (i, j), $i \in S$, $j \in T$; $C_{ij} \triangleq$ value of pair (i, j).

$$
\begin{aligned}
y_{ij} \triangleq\ & 1 \text{ if pair } (i, j) \text{ is chosen, else } 0 \\
\max\ & \Sigma_{(i,j)\in E}\, C_{ij} y_{ij} \\
\text{s.t.}\ & \Sigma_{j\in T}\, y_{ij} = 1,\ i \in S \\
& \Sigma_{i\in S}\, y_{ij} = 1,\ j \in T \\
& y_{ij} = 0 \text{ or } 1,\ (i, j) \in E
\end{aligned}
$$

Polynomially solvable by combinatorial algorithms. Special case of both Matching and Network Flows. LP relaxation is totally unimodular.

Capital Budgeting: Maximum value subset of objects or projects j to pack within given capacity or resource limits B_i; $A_{ij} \triangleq$ (nonnegative) consumption of resource i by project j; $C_j \triangleq$ value of project j.

$$
\begin{aligned}
y_j \triangleq\ & 1 \text{ if project } j \text{ is chosen, else } 0 \\
\max\ & \Sigma_j\, C_j y_j \\
\text{s.t.}\ & \Sigma_j\, A_{ij} \le B_i \quad \text{for all } i \\
& y_j = 0 \text{ or } 1 \quad \text{for all } j
\end{aligned}
$$

NP-hard. LP relaxation rounds down and can be strengthened with the inequalities discussed in the Valid Inequalities section above. Solution-building heuristics based on ratios of C_j to weighted sums of A_{ij} are common.

Facilities Location: Minimum cost subset of facilities i with capacity U_i, considering both construction, setup, etc. cost $F_i \ge 0$ of the facility; plus variable travel, service cost C_{ij} of serving customers j from facility i; $D_j \triangleq$ demand at j.

$z_{ij} \triangleq$ flow from i to j
$y_i \triangleq$ 1 if i is opened, else 0
min $\Sigma_{i,j}\, C_{ij}z_{ij} + \Sigma_i\, F_i y_i$
s.t. $\Sigma_i\, z_{ij} = D_j$ for all j
$\Sigma_j\, z_{ij} \le U_i y_i$ for all i
$0 \le z_{ij} \le D_j y_i$ for all i, j
$y_i = 0$ or 1 for all i

NP-hard. LP relaxation of the stated form is strong, but weak if the third system of constraints is deleted. Special case of Fixed Charge Network Flows.

Fixed Charge Network Flows: Minimum variable plus fixed cost flow in directed graph $D(N, A)$ with net demands D_k at nodes $k \in N$, and capacities U_{ij} on arc $(i, j) \in A$; $C_{ij} \triangleq$ unit cost of (i, j) flow; $F_{ij} \triangleq$ fixed cost (nonnegative) of using arc (i, j) at all.

$z_{ij} \triangleq$ flow in arc (i, j)
$y_{ij} \triangleq$ 1 if $z_{ij} > 0$, else 0
min $\Sigma_{(i,j)\in A}\, C_{ij}z_{ij} + \Sigma_{(i,j)\in A}\, F_{ij}y_{ij}$
s.t. $\Sigma_{(i,k)\in A}\, z_{ik} - \Sigma_{(k,j)\in A}\, z_{kj} = D_k,\ k \in N$
$0 \le z_{ij} \le U_{ij}y_{ij},\ (i, j) \in A$
$y_{ij} = 0$ or 1, $(i, j) \in A$

NP-hard. LP relaxation is poor unless capacities are tight. For other cases much improved *multicommodity* extended LP relaxation is obtained by introducing separate variables z^{st}_{ij} recording the (i, j) flow originating at supply s and bound for demand t.

Generalized Assignment: Minimum cost assignment of objects i to capacitated locations, plants, vehicles, etc.; $S_i \triangleq$ size of object i; $K_j \triangleq$ capacity of location j; C_{ij} cost of assigning i to j.

$y_{ij} \triangleq$ 1 if i assigned to j, else 0
min $\Sigma_{i,j}\, C_{ij}y_{ij}$
s.t. $\Sigma_j\, y_{ij} = 1$ for all i
$\Sigma_i\, S_i y_{ij} \le K_i$ for all j
$y_{ij} = 0$ or 1 for all i, j

NP-hard. Excellent bounds are obtained from Lagrangean relaxations dualizing $=1$ constraints to leave a series of Knapsack problems.

Generalized Covering: Minimum cost subset of patterns j to cover given requirements B_i; $A_{ij} \triangleq$ (nonnegative) contribution to requirement i by pattern j; $C_j \triangleq$ cost of pattern j.

$y_j \triangleq$ 1 if pattern j is chosen, else 0
max $\Sigma_j\, C_j y_j$
s.t. $\Sigma_j\, A_{ij}y_j \ge B_i$ for all i
$y_j = 0$ or 1 for all j

NP-hard. LP relaxation rounds up. Solution building heuristics based on ratios of C_j to weighted sums of A_{ij} are common.

Job Shop Scheduling: Sequence a collection of jobs j with steps $s = 1, \ldots, S_j$ on processors p to minimize the time to complete all jobs; processor sequence for any job is fixed; $T_{js} \triangleq$ duration of step s; $P_{js} \triangleq$ processor of step s.

$z \triangleq$ completion time of all tasks
$z_{js} \triangleq$ start time of job j, step s
min z
s.t. $z_{j1} \ge 0$ for all j
$z_{js} \ge z_{j,s-1} + T_{j,s-1}$,
for all j; $s = 2, \ldots, S_j$
$z \ge z_{jS_j} + T_{jS_j}$ for all j
$z_{js} \ge z_{j's'} + T_{j's'}$ or $z_{j's'} \ge z_{js} + T_{js}$,
for all j, s, j', s' with $P_{js} = P_{j's'}$

NP-hard. LP relaxation formed with big M as in Table 1 is poor. Both solution-building and solution-enhancing heuristics are common.

Knapsack: Maximum value subset of objects or projects j to pack within a given capacity or budget B; $A_j \triangleq$ size of object j; $C_j \triangleq$ value of object j.

$$
\begin{aligned}
y_j \triangleq\ & 1 \text{ if object } j \text{ is chosen, else } 0 \\
\max\ & \Sigma_j C_j y_j \\
\text{s.t.}\ & \Sigma_j A_j y_j \leq B \\
& y_j = 0 \text{ or } 1 \quad \text{for all } j
\end{aligned}
$$

NP-hard. LP relaxation rounds down and is near optimal in many cases. Heuristics can come arbitrarily close to optimal in polynomial time.

B+ Leontief Flows: Minimum cost flow through state nodes in N of directed hyperarcs or composition operators (I, j) combining positive integer multiples A^i_{Ij} of inputs $i \in I \subseteq N$ to produce one unit at node j; $C_{Ij} \triangleq$ unit cost of (I, j) flow; $B_k \geq 0$ is the nonnegative net requirement at node N.

$$
\begin{aligned}
z_{Ij} \triangleq\ & \text{flow in hyperarc } (I, j) \\
\min\ & \Sigma_{(I,j)} C_{Ij} z_{Ij} \\
\text{s.t.}\ & \Sigma_{(I,k)}\ z_{Ik} - \Sigma_{(K,j),K \ni k}\ A^{k\ j}_{K,\ \ } z_{Kj} \\
& \quad = B_k,\ k \in N \\
& z_{Ij} \geq 0, \quad \text{for all } (I, j)
\end{aligned}
$$

Polynomially solvable by combinatorial algorithms. LP relaxation is TDI so has integer optima if all B_k are integer.

Matching: Maximum utility nonoverlapping collection of pairs of objects from a given set N; $E \triangleq$ family of allowed pairs (i, j), $i < j$; $C_{ij} \triangleq$ value of pair (i, j).

$$
\begin{aligned}
y_{ij} \triangleq\ & 1 \text{ if pair } (i, j) \text{ is chosen, else } 0 \\
\max\ & \Sigma_{(i,j)\in E}\ C_{ij} y_{ij} \\
\text{s.t.}\ & \Sigma_{(i,k)\in E}\ y_{ik} + \Sigma_{(k,j)\in E}\ y_{kj} \leq 1,\ k \in N \\
& y_{ij} = 0 \text{ or } 1,\ (i, j) \in E
\end{aligned}
$$

Polynomially solvable by combinatorial algorithms. An exact (TDI) linear programming relaxation is also known.

Network Flows: Minimum cost flow in directed graph $D(N, A)$ with net demands D_k at nodes $k \in N$, and capacities U_{ij} on arc $(i, j) \in A$; $C_{ij} \triangleq$ unit cost of (i, j) flow.

$$
\begin{aligned}
z_{ij} \triangleq\ & \text{flow in arc } (i, j) \\
\min\ & \Sigma_{(i,j)\in A}\ C_{ij} z_{ij} \\
\text{s.t.}\ & \Sigma_{(i,k)\in A}\ z_{ik} - \Sigma_{(k,j)\in A}\ z_{kj} = D_k,\ k \in N \\
& 0 \leq z_{ij} \leq U_{ij},\ (i, j) \in A
\end{aligned}
$$

Polynomially solvable by combinatorial algorithms. LP relaxation is totally unimodular so has integer optima if demands and capacities are integer. Special cases include Assignment, Shortest Path, Maximum Flow, and Minimum Cut (see Chapter 99).

Parallel Processor Scheduling: Assign tasks i of duration T_i to one of several processors p so that the time to complete all tasks (*makespan*) is minimized.

$$
\begin{aligned}
y_{ip} \triangleq\ & 1 \text{ if task } i \text{ assigned to } p, \text{ else } 0 \\
z \triangleq\ & \text{completion time of all tasks} \\
\min\ & z \\
\text{s.t.}\ & \Sigma_i\ T_i y_{ip} \leq z \quad \text{for all } p \\
& \Sigma_p\ y_{ip} = 1 \quad \text{for all } i \\
& y_{ip} = 0 \text{ or } 1 \quad \text{for all } i, p
\end{aligned}
$$

NP-hard. Closely related to Generalized Assignment. Both solution-building and solution-enhancing heuristics are common.

Quadratic Assignment: Minimum cost assignment of objects $i \in S$ to locations, times $j \in T$, where value is measurable only after pairs of assignments; $E \triangleq$ family of allowed pairs (i, j), $i \in S$, $j \in T$; $V_{ik} \triangleq$ the shared activity of i and k; $C_{jl} \triangleq$ unit cost or distance of activity between locations j and l.

$$
\begin{aligned}
y_{ij} &\triangleq 1 \text{ if pair } (i, j) \text{ is chosen, else } 0 \\
\min \quad & \Sigma_{(i,j)\in E} \Sigma_{(k,l)\in E} (V_{ik}C_{jl})y_{kl} \\
\text{s.t.} \quad & \Sigma_{j\in T} y_{ij} = 1,\ i \in S \\
& \Sigma_{i\in S} y_{ij} = 1,\ j \in T \\
& y_{ij} = 0 \text{ or } 1.\ (i, j) \in E
\end{aligned}
$$

NP-hard. No effective relaxations are available for even moderate-sized problems. Local improvement by pairwise exchange of assignments is common.

Set Covering: Finite list of patterns, routes, workers, etc. j that must span a collection of customers, hours, districts, jobs i (duplication allowed); $A_{ij} \triangleq 1$ if pattern j covers i, else 0; $C_j \triangleq$ cost of pattern j.

$$
\begin{aligned}
y_j &\triangleq 1 \text{ if pattern } j \text{ is chosen, else } 0 \\
\min \quad & \Sigma_j C_j y_j \\
\text{s.t.} \quad & \Sigma_j A_{ij} y_j \geq 1 \quad \text{for all } i \\
& y_j = 0 \text{ or } 1 \quad \text{for all } j
\end{aligned}
$$

NP-hard. LP relaxation rounds up and is near optimal in many cases. LP relaxations often numerically difficult.

Set Packing: Finite list of patterns, routes, workers, etc. j that must not overlap in customers, hours, districts, jobs i; $A_{ij} \triangleq 1$ if pattern j uses i, else 0; $C_j \triangleq$ value of pattern j.

$$
\begin{aligned}
y_j &\triangleq 1 \text{ if pattern } j \text{ is chosen, else } 0 \\
\max \quad & \Sigma_j C_j y_j \\
\text{s.t.} \quad & \Sigma_j A_{ij} y_j \leq 1 \quad \text{for all } i \\
& y_j = 0 \text{ or } 1 \quad \text{for all } j
\end{aligned}
$$

NP-hard. LP relaxation rounds down and is near optimal in many cases. For others many valid inequalities are known. LP relaxations often numerically difficult.

Set Partitioning: Finite list of patterns, routes, workers, etc. j that must span a collection of customers, hours, districts, jobs i without duplication; $A_{ij} \triangleq 1$ if pattern j covers i, else 0; $C_j \triangleq$ cost of pattern j.

$$
\begin{aligned}
y_j &\triangleq 1 \text{ if pattern } j \text{ is chosen, else } 0 \\
\min \quad & \Sigma_j C_j y_j \\
\text{s.t.} \quad & \Sigma_j A_{ij} y_j = 1 \quad \text{for all } i \\
& y_j = 0 \text{ or } 1 \quad \text{for all } j
\end{aligned}
$$

NP-hard. LP relaxation gives good bounds in many cases but difficult to round. LP relaxations often numerically difficult.

Spanning Tree: Maximum total weight subset of edges in a connected undirected graph $G(N, E)$ containing exactly one path between each pair of nodes; $C_{ij} \triangleq$ value of edge (i, j), $i < j$.

$$
\begin{aligned}
y_{ij} &\triangleq 1 \text{ if edge } (i, j) \text{ is chosen, else } 0 \\
\max \quad & \sum_{(i,j)\in E} C_{ij} y_{ij} \\
\text{s.t.} \quad & \sum_{(i,j)\in E} C_{ij} y_{ij} \geq |N| - 1 \\
& \sum_{i,j\in S} y_{ij} \leq |S| - 1,\ S \subseteq N \\
& y_{ij} = 0 \text{ or } 1,\ (i, j) \in E
\end{aligned}
$$

Polynomially solvable by the greedy algorithm. LP relaxation is Totally Dual Integer.

Steiner Tree: Minimum cost collection of edges of a graph $G(N, E)$ providing a path between vertices in subset $S \subseteq N$; $C_{ij} \triangleq$ (nonnegative) cost of edge $(i, j) \in E$, $i < j$.

$$
\begin{aligned}
y_{ij} &\triangleq \text{ 1 if edge } (i, j) \text{ is chosen, else 0} \\
\min \quad & \Sigma_{(i,j)\in E}\, C_{ij} y_{ij} \\
\text{s.t.} \quad & \sum_{i\in Q, j\in N\setminus Q} y_{ij} + \sum_{j\in Q, i\in N\setminus Q} y_{ij} \geq 1, \\
& \quad Q \subset N,\ Q \cap S \neq \emptyset,\ S \setminus Q \neq \emptyset \\
& y_{ij} = 0 \text{ or } 1,\ (i, j) \in E
\end{aligned}
$$

NP-hard. LP relaxation easily rounded up then enhanced by deleting unnecessary edges. Numerous valid inequalities are known to strengthen the LP relaxation.

Traveling Salesman: Minimum total weight route or sequence visiting each object, job, customer in N exactly once. Represent on a graph with nodes N and edges for allowed i to j transitions; $C_{ij} \triangleq$ cost of transition (i, j), $i < j$.

$$
\begin{aligned}
y_{ij} &\triangleq \text{ 1 if transition } (i, j) \text{ is chosen, else 0} \\
\max \quad & \Sigma_{i,j}\, C_{ij} y_{ij} \\
\text{s.t.} \quad & \Sigma_{i<k}\, y_{ik} + \Sigma_{j>k}\, y_{kj} = 2,\ k \in N \\
& \Sigma_{i,j\in S}\, y_{ij} \leq |S| - 1,\ S \subset N \\
& y_{ij} = 0 \text{ or } 1,\ (i, j) \in E
\end{aligned}
$$

NP-hard. The best-researched problem in discrete optimization. LP relaxation of the given form is strong but requires a separation procedure. Numerous effective solution building and solution enhancing heuristics exist, especially for the *triangular cost* case with $C_{ik} \leq C_{ik} + C_{kj}$.

REFERENCES

Aarts, E., and Lenstra, J. K. (1997), *Local Search in Combinatorial Optimization,* Wiley-Interscience, New York.

Balas, E., and Martin, C. H. (1980), "Pivot and Complement—A Heuristic for 0–1 Programming," *Management Science,. Vol. 26, pp. 86–96.*

Crowder, H., Johnson, E. L., and Padberg, M. (1983), "Solving Large-Scale Zero–One Linear Programming Problems," *Operations Research,* Vol. 31, pp. 803–834.

Dobson, G. (1982), "Worst-Case Analysis of Greedy Heuristics for Integer Programming with Nonnegative Data," *Mathematics of Operations Research,* Vol. 7, pp. 515–531.

Garey, M. R., and Johnson, D. S. (1979), *Computers and Intractability: A Guide to the Theory of NP-Completeness,* W.H. Freeman, San Francisco.

Glover, F., and Laguna, M. (1998), *Tabu Search,* Kluwer, Norwell, MA.

Goldberg, D. E. (1989), *Genetic Algorithms in Search, Optimization and Machine Learning,* Addison-Wesley, Reading, MA.

Holland, J. H. (1975), *Adaptation in Natural and Artificial Systems,* University of Michigan Press, Ann Arbor, MI.

Kirkpatrick, S., Gelatt, J. R., and Vecchi, M. P. (1983), "Optimization by Simulated Annealing," *Science,* Vol. 220, pp. 671–680.

Lawler, E. L. (1976), *Combinatorial Optimization: Networks and Matroids,* Holt, Rinehart & Winston, New York.

Martin, R. K. (1999), *Large Scale Linear and Integer Optimization: A Unified Approach,* Kluwer, Norwell, MA.

Nemhauser, G. L., and Wolsey, L. A. (1988), *Integer and Combinatorial Optimization,* John Wiley & Sons, New York.

Papadimitriou, C. H. (1994), *Computational Complexity,* Addison-Wesley, Reading, MA.

Papadimitriou, C. H., and Steiglitz, K. (1982), *Combinatorial Optimization: Algorithms and Complexity,* Prentice-Hall, Englewood Cliffs, MJ.

Parker, R. G., and Rardin, R. L. (1988), *Discrete Optimization,* Academic Press, Boston.

Rardin, R. L. (1998), *Optimization in Operations Research,* Prentice Hall, Upper Saddle River, NJ.

Rardin, R. L., and Wolsey, L. A. (1993), "Valid Inequalities and Projecting the Multicommodity Extended Formulation for Uncapacitated Fixed Charge Network Flow Problems," *European Journal of Operational Research,* Vol. 71, 95–109.

Reeves, C. R. (1993), *Modern Heuristic Techniques in Combinatorial Problems,* Halsted Press, New York.

Schrijver, A. (1986), *Theory of Linear and Integer Programming,* John Wiley & Sons, New York.

Wolsey, L. A. (1998), *Integer Programming,* Wiley-Interscience, New York.