INFORMS Journal on Computing
Vol. 12, No. 1, Winter 2000

0899-1499/100/1201-0002 $05.00

Progress in Linear Programming-Based Algorithms for Integer Programming: An Exposition

ELLIS L. JOHNSON, GEORGE L. NEMHAUSER, AND MARTIN W.P. SAVELSBERGH / *Georgia Institute of Technology, School of Industrial and Systems Engineering, Atlanta, GA 30332-0205, Email: {ellis.johnson,george.nemhauser,martin.savelsbergh}@isye.gatech.edu*

(Received: January 1997; revised: November 1998, August 1999; accepted: November 1999)

This paper is about modeling and solving mixed integer programming (MIP) problems. In the last decade, the use of mixed integer programming models has increased dramatically. Fifteen years ago, mainframe computers were required to solve problems with a hundred integer variables. Now it is possible to solve problems with thousands of integer variables on a personal computer and obtain provably good approximate solutions to problems such as set partitioning with millions of binary variables. These advances have been made possible by developments in modeling, algorithms, software, and hardware. This paper focuses on effective modeling, preprocessing, and the methodologies of branch-and-cut and branch-and-price, which are the techniques that make it possible to treat problems with either a very large number of constraints or a very large number of variables. We show how these techniques are useful in important application areas such as network design and crew scheduling. Finally, we discuss the relatively new research areas of parallel integer programming and stochastic integer programming.

This paper is about solving the model of maximizing a linear function subject to linear inequality and equality constraints, where some or all of the variables are required to be integral, by a methodology known as linear programming-based branch-and-bound. The model

$$\begin{aligned} \max \quad & cx \\ & Ax \leqslant b \\ & l \leqslant x \leqslant u \\ & x_j \text{ integral}, \ j = 1, \ldots, p \end{aligned}$$

is called a *mixed integer program* (MIP). The input data are the matrices c $(1 \times n)$, A $(m \times n)$, b $(m \times 1)$, l $(1 \times n)$, and u $(1 \times n)$, and the n-vector x is to be determined. We assume $1 \leqslant p \leqslant n$, otherwise the problem is a *linear program* (LP). If $p = n$, the problem is a *pure integer program* (PIP). A PIP in which all variables have to be equal to 0 or 1 is called a *binary integer program* (BIP), and a MIP in which all integer variables have to be equal to 0 or 1 is called a *mixed binary integer program* (MBIP). Binary integer variables occur very frequently in MIP models of real problems.

We consider problems in which the matrix A has no special structure and also some structured problems that are natural to formulate as MIPs. The latter class includes the traveling salesman problem, fixed-charge network flow problems, facility location problems, and numerous other combinatorial optimization problems, many of which are defined on graphs or networks.

LP-based branch-and-bound is a very general algorithm that does not require any structure other than linearity, and linearity is not very restrictive in problem formulation. However, structure can be used to great advantage in LP-based branch-and-bound to obtain vastly improved algorithms. We will discuss this use of structure.

There are several other significant approaches to discrete optimization that are beyond the scope of this paper. These include such important topics as constraint programming, Lagrangian duality, semidefinite programming, basis reduction, approximation algorithms, and heuristics such as simulated annealing, TABU search, and genetic algorithms.

Our intention is for this paper to be accessible to readers who are not already specialists in integer programming including students who are exploring the possibility of doing research in integer programming; users of integer programming models and software who would like to understand more about the methodology so that they might, for example, make more knowledgeable choices; and operations researchers, management scientists, computer scientists, and mathematicians who are just curious about recent developments in this very active branch of optimization. The integer programming specialist is unlikely to find unfamiliar material here but might be interested in comparing his or her opinions on recent and future important developments with ours. Limited space requires that we do no more than introduce and sketch the numerous developments that we have chosen to present. We do, however, point the reader to the papers that contain the detailed results.

In the last decade, the use of MIP models has changed and increased dramatically. The ability to solve relatively large problems in relatively small amounts of time has made integer programming much more widespread. Fifteen years ago, mainframe computers were required to solve problems with a hundred integer variables. Now we solve problems with thousands of binary variables on a personal computer or a workstation, and we also have the capability of getting provably good approximate solutions to problems such as set partitioning with millions of variables. These advances

Subject classifications: Combinatorial optimization.
Other key words: Mixed integer programming, branch-and-cut, branch-and-price, lifting.

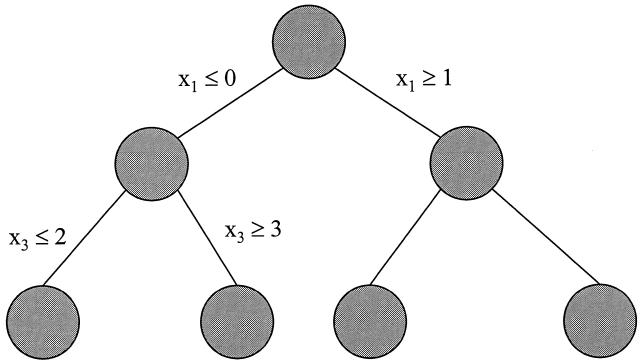

Figure 1. Branch-and-bound tree.

have been made possible by developments in hardware, software, and algorithms. This paper focuses on the algorithmic advances and implementation issues. A cautionary note is appropriate. There exist very small binary integer programs that are currently beyond the reach of modern solvers[29] and it also appears that problems with general integer variables are much more difficult than problems with binary variables.

The *LP relaxation* of a MIP is obtained by dropping the integrality restrictions. The main use of the LP relaxation in solving a MIP is that the optimal value of the LP relaxation provides an upper bound on the optimal value of the corresponding MIP. In addition, if an optimal solution to the LP relaxation is found that satisfies the integrality restrictions, then that solution is also optimal to the MIP. If the LP relaxation is infeasible, then the MIP is also infeasible.

While other relaxations such as those based on Lagrangian duality and semidefinite programming are undoubtedly useful in special situations, LP is still the champion upper-bound producer for MIPs. LP generally gives reasonably tight bounds, and the methodology for solving LPs is very efficient and reliable. Since a very large fraction of the time consumed by a branch-and-bound algorithm is spent on LP solving, the orders of magnitude speed improvements in simplex algorithm codes that have taken place in the last decade have been extremely important. Moreover, modern simplex codes easily deal with the addition of rows (constraints) and columns (variables) during the solution process, which is essential for the new branch-and-bound algorithms called *branch-and-cut* and *branch-and-price*. Interior point methods are also quite useful for solving large-scale LPs, but they are not widely employed in branch-and-bound algorithms because it is cumbersome to reoptimize an LP with an interior point code after rows or columns have been added.

The basic structure of branch-and-bound is an enumeration tree (Fig. 1). The root node of the tree corresponds to the original problem. As the algorithm progresses, the tree grows by a process called *branching*, which creates two or more child nodes of the parent node. Each of the problems at the child nodes is formed by adding constraints to the problem at the parent node. Typically, the new constraint is obtained by simply adding a bound on a single integer variable, where one child gets an upper bound of some integer d, and the other child gets a lower bound of $d + 1$. An essential requirement is that each feasible solution to the parent node problem is feasible to at least one of the child node problems.

The fundamental paradigm of branch-and-bound with LP relaxations for solving MIPs has not really changed. The difference is that advances have made it possible to significantly curtail the enumeration. In other words, the branch-and-bound trees have many fewer nodes, and although the time required to process a node can be higher, there can be a substantial decrease in the overall computing time.

To motivate the methodological advances, it is essential to understand the use of lower and upper bounds on the optimal value of the objective function. To achieve this understanding, we briefly sketch a basic LP-based branch-and-bound algorithm for MIP.

Any feasible solution to MIP provides a lower bound. Let z_{best} be the value of the greatest lower bound over all available feasible solutions.

Let MIP(0) be the original MIP and let MIP(k) be the problem at node k. At node k of the tree we solve the LP relaxation LP(k) of MIP(k). At this stage of our presentation, we assume that LP(k) is feasible and bounded and that its optimal value is $z(k)$. If an optimal solution $x(k)$ to LP(k) is found that happens to satisfy the integrality constraints, then $x(k)$ is also optimal to MIP(k), in which case if $z(k) > z_{best}$, we update z_{best}. We can forget about node k. Otherwise, if $x(k)$ does not satisfy all of the integrality constraints, there are two possibilities. If $z(k) \leqslant z_{best}$, an optimal solution to MIP(k) cannot improve on z_{best}. Again, we can forget about node k. On the other hand, if $z(k) > z_{best}$, MIP(k) requires further exploration, which is done by branching as explained above. Create q new subproblems (children) MIP($k(i)$), $i = 1, \ldots, q$, $q \geqslant 2$ of MIP(k). Each feasible solution to MIP(k) must be feasible to at least one child and, conversely, each feasible solution to a child must be feasible to MIP(k). Moreover, the solution $x(k)$ must not be feasible to any of the LP relaxations of the children. A simple realization of these requirements is to select a variable x_j, $j \leqslant p$, such that $x_j(k)$ is not integral and to create two children; in one of these we add the constraint $x_j \leqslant \lfloor x_j(k) \rfloor$, which is the round down of $x_j(k)$, and in the other we add $x_j \geqslant \lceil x_j(k) \rceil$, which is the round up of $x_j(k)$. The child nodes of node k corresponding to these subproblems are added to the tree, and the tree grows.

Large trees are a result of many branchings, and to avoid branching insofar as is possible is equivalent to avoiding the case $z(k) > z_{best}$. What control do we have here? For one, we can apply fast heuristics to try to obtain high values of z_{best}. Another complementary approach is to add a constraint, called a *cut*, to LP(k) that has the property that $x(k)$ is not satisfied by the cut but that every feasible solution to MIP(k) is satisfied by the cut. Adding such cuts will increase the possibility that no branching is required from node k because it may result in a decrease of $z(k)$ or an integer solution. Finally, most MIPs have many different formulations in terms of variables and constraints. The LP relaxations of

different formulations can be vastly different in terms of the quality of the bounds they provide. So we can control the size of the tree by providing a good initial formulation.

We have now set the stage for what follows. In the next section, we will discuss good formulations. Section 2 discusses preprocessing. The objectives are to fix variables, to eliminate redundant constraints, and to derive logical inequalities. Section 3 presents branching rules for searching the tree efficiently. In Section 4, we give primal heuristics for finding feasible solutions. The two fundamental algorithms presented are branch-and-cut in Section 5 and branch-and-price in Section 6. In branch-and-cut, we add cuts to the LP relaxations at nodes of the tree. In branch-and-price, we have a formulation with a huge number of variables, largely because such a formulation provides good bounds, and then we begin with a small subset of the variables and generate the others on an "as needed only" basis throughout the tree. Section 7 deals with some important implementation issues of branch-and-cut and branch-and-price, and in Sections 8 and 9 we discuss software, and testing and computing, respectively. Section 10 presents two important applications—network design, which is solved by branch-and-cut, and crew scheduling, which is solved by branch-and-price. In Section 11, we present two new research areas of integer programming—stochastic integer programming and parallel integer programming.

A wealth of information and references on many aspects of integer programming can be found in the books by Schrijver,[84] Nemhauser and Wolsey,[79] Williams,[89] and Wolsey.[91] In this paper we only give a small sample of references with emphasis on recent publications.

1. Formulation Principles

The LP relaxations of two different formulations of the same MIP may provide different upper bounds. Everything else being equal, we prefer the formulation whose LP relaxation provides the better bound. The lower the bound, the stronger or tighter the formulation is said to be. Good formulations are characterized by LP relaxations that provide strong or tight bounds.

We begin with an example that illustrates how modified constraints can provide a better formulation, and then we present another example to show how properly selected variables can improve a formulation. We then discuss some general principles of good formulations. Finally, we illustrate the significance of using both an exponential number of constraints and an exponential number of variables to obtain good formulations.

Consider the "pigeon hole" problem. There are $n + 1$ pigeons and n holes. We want to place all of the pigeons in the holes but in such a way that no pair of pigeons is in the same hole. Obviously there is no solution since there are more pigeons than holes. But how do you prove this? Let $x_{ij} = 1(0)$ if pigeon i is in (not in) hole j. We want to assign every pigeon to a hole, so we have the pigeon constraints

$$\sum_{j=1}^{n} x_{ij} = 1 \quad i = 1, \ldots, n + 1.$$

But pigeons i and k cannot go into the same hole so we have the hole constraints

$$x_{ij} + x_{kj} \leq 1 \quad i = 1, \ldots, n, k = i + 1, \ldots, n + 1, \quad j = 1, \ldots, n.$$

This system is infeasible in binary variables, but when the integrality restrictions are relaxed, $x_{ij} = 1/n$ for all i and j is a feasible solution to the relaxation. Moreover, after many branchings, the LP relaxation remains feasible so that nearly total enumeration is required and the LP relaxation is of no use.

Now observe that the hole constraints are equivalent to

$$\sum_{i=1}^{n+1} x_{ij} \leq 1 \quad j = 1, \ldots, n,$$

i.e., no more than one pigeon per hole. The LP relaxation with these n constraints, each containing $n + 1$ variables replacing about n^3 constraints and each containing two variables, is much tighter. In fact, it is easy to show that it is infeasible. The second set of hole constraints are known as clique constraints. Even if they are not present in the original formulation, they can be derived in preprocessing (Section 2).

Now we consider a problem where the right choice of variables is important. Consider an uncapacitated, discrete time, finite horizon, economic lot-sizing model. There is a given demand d_t in period t for $t = 1, \ldots, T$, which can be satisfied by production in any of the periods $k = 1, \ldots, t$. In period t, there is a per unit production cost, a per unit holding cost, and a setup cost that is incurred only if the production in period t is positive. By uncapacitated we mean that there is no explicit capacity on the amount y_t produced in period t. However, there is the implicit bound

$$y_t \leq \sum_{k=t}^{T} d_k = D_t \quad t = 1, \ldots, T.$$

The periods are linked by nonnegative inventory variables

$$s_t = s_{t-1} + y_t - d_t \quad t = 1, \ldots, T,$$

where we assume $s_0 = 0$. The setup costs are modeled with binary variables x_t and the variable upper bound constraints

$$y_t \leq D_t x_t \quad t = 1, \ldots, T.$$

Assuming positive setup costs, in the LP relaxation with $0 \leq x_t \leq 1$ we will obtain

$$x_t = y_t / D_t \quad t = 1, \ldots, T.$$

Hence, if y_t is positive and not at its upper bound, then x_t will be fractional. Since it would not be optimal to produce all current and future demand in more than one period, all but one of the positive x_t will be fractional. We conclude that this LP relaxation is not very good.

It is possible to eliminate the inventory variables by using variables y_{tk} representing the quantity produced in period t

to meet demand in period k where $k \geq t$. Now the demand constraints are

$$\sum_{t=1}^{k} y_{tk} = d_k \quad k = 1, \ldots, T,$$

and the variable upper bound constraints are

$$y_{tk} \leq d_k x_t \quad k = 1, \ldots, T, \; t = 1, \ldots, k.$$

These variable upper bound constraints are much better for the LP relaxation. In particular, it can be shown that there are optimal solutions to the MIP and to the LP relaxation such that y_{tk} positive implies $y_{tk} = d_k$. This result implies that by solving the LP relaxation, we also solve the MIP.

The two examples illustrate general principles that are useful in obtaining good formulations. The pigeon hole problem showed how we can get clique constraints that are stronger than the original constraints with respect to the LP relaxation. This idea will be pursued further in the discussion of preprocessing. The lot-sizing example illustrated the importance of tight bounds in variable upper bound constraints. Again, we will elaborate on this under preprocessing.

Constraint disaggregation is another general principle for obtaining good formulations. In *constraint disaggregation* for 0-1 variables,

$$x_j \leq x_0, \; j = 1, \ldots, m$$

is used to represent the constraints $x_0 = 0$ implies $x_j = 0$, $j = 1, \ldots, m$, rather than the single constraint

$$x_1 + \cdots + x_m \leq m x_0,$$

which gives a weaker LP relaxation.

An experienced modeler should use these principles in developing an initial formulation. However, although it will take some time, preprocessing can automatically execute many of these transformations.

To obtain strong bounds, it may be necessary to have a formulation in which either the number of constraints or the number of variables (or possibly both) is exponential in the size of the natural description of the problem. These kinds of formulations are practical now because we have LP algorithms that are flexible enough to add variables and constraints on an "as needed" basis. Techniques for handling these kind of formulations within a branch-and-bound framework are presented in Sections 5, 6, and 7.

The traveling salesman problem (TSP) provides a well-known example of the use of an exponential number of constraints. Given a graph $G = (V, E)$, where V is the node set and E is the edge set, a feasible solution or tour is a set of edges forming a simple cycle that contains all of the nodes. An optimal solution is a tour in which the sum of the edge costs is minimum. We introduce 0-1 variables to represent the selection of edges, with $x_e = 1$ if edge e is in the tour and $x_e = 0$ otherwise. Feasible solutions can be described by constraints stating that:

1. Each node is met by exactly two edges. Let $\delta(v)$ be the set of edges that are incident to node v, then

$$\sum_{e \in \delta(v)} x_e = 2 \quad \forall v \in V.$$

2. The subgraph consisting of the selected edges is connected.

Connectivity can be enforced by requiring that, for each set $U \subset V$, the number of edges with one node in U and the other in $V \backslash U$ be at least two, i.e.,

$$\sum_{e \in \delta(U)} x_e \geq 2 \quad \forall U \subset V, \; 2 \leq |U| \leq \left\lfloor \left| \frac{V}{2} \right| \right\rfloor,$$

where $\delta(U)$ is the set of edges that are incident to precisely one node in U.

The number of these subtour elimination constraints is exponential in $|V|$, so it is not possible to include all of them in the initial LP relaxation. Nevertheless, this formulation is preferred to some others that preserve connectivity with a small number of constraints since the quality of the LP relaxation is superior.

In many situations, good formulations of set partitioning problems require an exponential number of columns. Suppose a given set $M = \{1, \ldots, m\}$ is to be partitioned into subsets. The subsets have required characteristics, e.g., on their number of elements, types of elements, etc., and costs. The problem is to find a partition of minimum total cost. An example is a political districting problem[76] in which the counties of a state are to be partitioned into congressional districts. In one type of formulation, there is a variable $x_{ij} = 1$ if element $i \in M$ is put in subset j, and $x_{ij} = 0$ otherwise. This type of formulation is compact but generally inferior in modeling capability and quality of LP relaxation to the formulation in which each feasible subset is (implicitly) enumerated, and there is a variable $x_i = 1$ if subset i is selected and $x_i = 0$ otherwise. The latter formulation allows one to capture complex restrictions on the structure and costs of the subsets. However, in this model, the number of variables typically grows exponentially with M. For example, instances with $|M| = 100$ can have millions of variables.

Finally, we observe that there are some parts of a model that may be captured better in branching than by adding 0-1 variables. For example, in a production scheduling problem, consider a job whose processing on a machine cannot be interrupted but, in a discrete time model, if production is started late in period t, then production may not end until some time in period $t + 1$. Let x_{kt} be the fraction of job k done in period t. Then $0 \leq x_{kt} \leq 1$, $\Sigma_t \, x_{kt} = 1$, x_{kt} may be positive for at most two values of t and equal to 2 only if the two periods are consecutive. The last condition can be modeled with binary variables and linear constraints, but it is better to leave the condition out of the LP relaxation and, instead, to enforce it through special branching rules, as explained in Section 3. See Gue et al.[53] for a practical application with these characteristics.

2. Preprocessing

We have stressed the importance of tight linear programming relaxations. Preprocessing applies simple logic to reformulate the problem and tighten the linear programming relaxation. In the process, preprocessing may also reduce the size of an instance by fixing variables and eliminating constraints. Sometimes preprocessing may detect infeasibility.

The simplest logical testing is based on bounds. Let L_i be any lower bound on the value of the ith row $A^i x$ subject only to $l \leqslant x \leqslant u$, and let U_i be any upper bound on the value of the ith row $A^i x$ subject only to $l \leqslant x \leqslant u$. A constraint $A^i x \leqslant b_i$ is redundant if $U_i \leqslant b_i$ and is infeasible if $L_i > b_i$. A bound on a variable may be tightened by recognizing that a constraint becomes infeasible if the variable is set at that bound. These considerations apply to linear programs as well as to integer programs. However, such tests may be unnecessary for linear programs. For mixed integer programs, one is willing to spend more time initially in order to reduce the possibility of long solution times. In the case of 0-1 variables, the initial lower and upper bounds on the variables are 0 and 1, respectively. If these bounds are improved, then the variable can be fixed. For example, if the upper bound of a 0-1 variable is less than 1, then it can be fixed to 0. Of course, if the lower bound is positive and the upper bound is less than 1, then the problem is integer infeasible. In summary, considering one row together with lower and upper bounds may lead to dropping the row if it is redundant, declaring the problem infeasible if that one row is infeasible, or tightening the bounds on the variables.

These relatively simple logical testing methods can become powerful when combined with *probing*. Probing means temporarily setting a 0-1 variable to 0 or 1 and then redoing the logical testing. If the logical testing shows that the problem has become infeasible, then the variable on which we probe can be fixed to its other bound. For example, $5x_1 + 3x_2 \geqslant 4$ becomes infeasible when x_1 is set to 0. We conclude that x_1 must be 1 in every feasible solution. If the logical testing results in another 0-1 variable being fixed, then a logical implication has been found. Consider $5x_1 + 4x_2 \times x_3 \leqslant 8$. If x_1 is set to 1, then subsequent bound reduction will fix x_2 to 0. Thus, we have found the logical implication $x_1 = 1$ implies $x_2 = 0$, which can be represented by the inequality $x_1 + x_2 \leqslant 1$. Adding this inequality tightens the LP relaxation since (1, 0.75, 0) is feasible for the original inequality but is infeasible for the implication inequality. If the logical testing shows that a constraint has become redundant, then it can be tightened by what is called coefficient reduction or coefficient improvement. For example, $2x_1 + x_2 + x_3 \geqslant 1$ becomes strictly redundant when x_1 is set to 1. Whenever a variable being set to 1 leads to a strictly redundant $\geqslant$ constraint, then the coefficient of the variable can be reduced by the amount that the constraint became redundant. Therefore, $2x_1 + x_2 + x_3 \geqslant 1$ can be tightened to $x_1 + x_2 + x_3 \geqslant 1$. Note that (0.5, 0, 0) is no longer feasible to the LP relaxation of the tightened constraint. Less obvious coefficient improvements can also be found during probing. For example, if there are inequalities

$$x_1 \leqslant x_2,\ x_1 \leqslant x_3, \text{ and } x_2 + x_3 \geqslant 1, \tag{1}$$

then setting x_1 to 1 leads to a strictly redundant inequality $x_2 + x_3 \geqslant 1$, and the coefficient of x_1, namely 0, can be lowered to -1. Clearly, (1) and

$$x_1 \leqslant x_2,\ x_1 \leqslant x_3 \text{ and } -x_1 + x_2 + x_3 \geqslant 1$$

have the same set of 0-1 solutions. However, the fractional solution (0.5, 0.5, 0.5) is allowed by the first set of inequalities but not by the strengthened second set. One important aspect of probing, and especially of coefficient improvement, is that it applies to the important mixed 0-1 case and not just to pure 0-1 problems.

An inequality of the form $\Sigma_{j \in S} x_j \leqslant 1$ is called a *clique* inequality. These inequalities can be derived as strengthened implication inequalities. For example, the three implication inequalities $x_1 + x_2 \leqslant 1$, $x_2 + x_3 \leqslant 1$, and $x_1 + x_3 \leqslant 1$ together imply the stronger clique inequality $x_1 + x_2 + x_3 \leqslant 1$. (Note that (0.5, 0.5, 0.5) is feasible for the initial set of implication inequalities but not for the clique inequality.) A more general form of these inequalities, called a generalized clique inequality, possibly involves -1 coefficients and is derived from implication inequalities of the form $x_1 + x_2 \leqslant 1$ ($x_1 = 1$ implies $x_2 = 0$), $x_1 \leqslant x_2$ ($x_1 = 1$ implies $x_2 = 1$), and $x_1 + x_2 \geqslant 1$ ($x_1 = 0$ implies $x_2 = 1$). A generalized clique inequality has the form

$$\sum_{j \in S^+} x_j - \sum_{j \in S^-} x_j \leqslant 1 - |S^-|.$$

These inequalities have the property that for $j \in S^+$, if x_j is set to 1, then all other x_j can be fixed to 0 for $j \in S^+$ and to 1 for $j \in S^-$. Similarly, for $j \in S^-$, if x_j is set to 0, then all the other x_j can be fixed as before. These inequalities are generated from smaller such inequalities. For example, from $x_1 + x_2 \leqslant 1$, $x_2 - x_3 \leqslant 0$, and $x_1 - x_3 \leqslant 0$, we can derive the stronger inequality $x_1 + x_2 - x_3 \leqslant 0$. The generalized clique inequalities can be derived from coefficient improvement as a result of probing. In the small example just given, if x_3 is set to 0, then both x_1 and x_2 can be fixed to 0 and the inequality $x_1 + x_2 \leqslant 1$ becomes strictly slack and, therefore, can be strengthened to $x_1 + x_2 - x_3 \leqslant 0$.

Another class of inequalities based on implications, which may be found by probing, is of the form

$$y_j \leqslant u_j x_k.$$

This inequality is valid whenever $x_k = 0$ forces the continuous variable $y_j \leqslant u_j$ to be 0. Similar inequalities may be derived whenever probing on a 0-1 variable x_k forces a continuous variable to its lower or upper bound. Therefore, probing will automatically disaggregate constraints. In the previous section, we argued that we should use

$$x_j \leqslant x_0,\ j = 1, \ldots, m$$

rather than the single constraint

$$x_1 + \cdots + x_m \leqslant m x_0$$

because it gives a stronger LP relaxation. It is easy to see that probing on the variable x_0 will automatically perform this reformulation.

In summary, preprocessing may identify infeasibility and

redundant constraints, improve bounds, fix variables, and generate new valid inequalities. Coefficient improvement leads to simply replacing an inequality with a stronger one. Thus, it can be done in a preprocessing phase independent of solving the linear program. Clique inequalities on 0-1 variables and implication inequalities linking 0-1 and continuous variables may be found by probing and, in the case of clique inequalities, may be strengthened by probing on the clique inequalities themselves. Since there may be many clique and implication inequalities, one strategy is to generate clique and implication inequalities, store them in separate tables, and only add them to the linear program when they are violated by a current LP solution.

To give some idea of the relative computational difficulty of the various preprocessing techniques, any of the techniques applied to only one row is not likely to take much time and will almost always be worth doing. This remark applies to fixing, identifying redundant rows, bound reduction, coefficient improvement, and finding implication inequalities.

Saving and preprocessing separate implication and clique tables is the next level of difficulty. Especially if the clique table becomes large, probing on the clique table may require much more time. For example, if the original problem is set partitioning, then the resulting clique table includes the original problem.

Full probing on the matrix is the most time consuming part of preprocessing. The clique table gives a way to speed up probing in that whenever a 0-1 variable is set, the implications from the clique tables can be fixed before returning to the relatively more complex probing on the matrix. Also, probing may be restricted to variables that are fractional in the current LP solution. Despite efficient implementations, probing remains a possibly effective but sometimes too time consuming preprocessing technique that requires care in practice.

The papers by Brearley et al.,[22] Dietrich et al.,[36] Escudero et al.,[39] Guignard and Spielberg,[54] Hoffman and Padberg,[57] Johnson,[62] Johnson and Padberg,[63] and Savelsbergh[82] discuss preprocessing and probing.

3. Branching

In the normal course of a branch-and-bound algorithm, an unevaluated node is chosen (initially MIP(0)), the LP relaxation is solved, and a fractional variable (if there is one) is chosen on which to branch. If the variable is a 0-1 variable, one branch sets the variable to 0 (the *down* branch) and the other sets it to 1 (the *up* branch). If it is a general integer variable and the LP value is, for instance, 3½, then one branch constrains it to be ≤ 3 and the other constrains it to be ≥ 4. Even in this simple framework, there are two choices to be made: 1) which active node to evaluate and 2) the fractional variable on which to branch.

The variable choice can be critical in keeping the tree size small. A simple rule is to select a variable whose fractional value is closest to $\frac{1}{2}$, i.e., with the maximum integer infeasibility. More sophisticated rules, which are used by many commercial solvers, try to choose a variable that causes the LP objective function to deteriorate quickly in order to make the LP values at the child nodes as small as possible. This is done in an attempt to prune one or both of the child nodes using the fact that a node can be pruned if its LP value is less than or equal to the best known IP solution. Early branching on variables that cause big changes can be critical in keeping the tree small.

It would be too expensive to compute the actual objective function changes for all candidate variables every time that branching was required. Instead, estimates of the rate of objective function change on both the down and up branches, called *pseudocosts*, are used. The pseudocosts can be obtained from dual information at the node, or they can be based on actual changes from previous branchings involving the variable. Instead of using an estimate of the change, we can also explicitly perform one or more dual simplex iterations to obtain a lower bound on the change and use this lower bound to choose among variables. A recent variant of this idea has become known as *strong branching*. In strong branching, a fairly large number of dual simplex iterations is carried out but only for a small set of candidate branching variables. Strong branching has been shown to be very effective for large traveling salesman problems and large set partitioning problems.[3, 68]

The variable choice rules discussed above do not depend on knowledge of the problem structure and, therefore, are well suited for general purpose optimizers. However, when we do know something about the problem structure, it is often possible to do better by specifying a priority order on the variables. The idea here is to select more important variables first, i.e., variables that correspond to more important decisions in the underlying problem.

Consider a capacitated facility location problem in which each customer requires single sourcing, i.e., each customer's demand has to be fully satisfied from a single facility. A set of potential sites for the facilities has been identified and the company wants to decide which of these sites to use as well as the set of customers that will be served by each of the facilities. Integer programs to model these situations involve two types of binary variables: y_i to indicate whether a facility will be located at potential site i ($y_i = 1$) or not ($y_i = 0$) and x_{ij} to indicate whether customer j is served from a facility at site i ($x_{ij} = 1$) or not ($x_{ij} = 0$). Because the decision of whether or not to locate a facility at a certain site affects the overall solution more than the decision to serve a particular customer from one site or another, it is better to branch on the y variables before branching on the x variables.

Node choice rules are partially motivated by the desire to find good feasible solutions early in the search. This is important for two reasons: if time runs out, then a good answer may be all one gets, and a good lower bound inhibits growth of the tree. A common node choice rule is to choose one of the two newly created nodes, i.e., go depth-first, as long as the objective function value of one of the children is not too low. However, if the current bound becomes too low, then choose from among all active nodes either one with the highest bound or one with highest estimate of the value of a feasible solution. Such an estimate can be obtained using pseudocosts.

It is difficult to balance the relative advantages and disadvantages of selecting nodes near the top or bottom of the tree. In general, the number of active nodes may rapidly increase if the active node is always chosen high up in the tree. On the other hand, if the node is always chosen from down low in the tree, the number of active nodes stays small, but it may take a long time to improve the upper bound.

More complicated branching involves sets of variables. An example of branching on a set of variables is the use of special ordered sets of type 1 (SOS1). It consists of splitting an ordered set into two parts, one for each branch. That is, for a constraint such as

$$x_1 + x_2 + x_3 + x_4 + x_5 = 1,$$

one branch could be

$$x_1 + x_2 = 1 \text{ (with } x_3 = 0,\ x_4 = 0, \text{ and } x_5 = 0),$$

and the other branch could be

$$x_3 + x_4 + x_5 = 1 \text{ (with } x_1 = 0 \text{ and } x_2 = 0).$$

The same type of branching applies to generalized cliques: the constraint

$$x_1 + x_2 + x_3 - x_4 - x_5 = -1$$

can be branched on by forming two branches having $x_1 - x_4 = 0$ (with $x_2 = 0$, $x_3 = 0$, and $x_5 = 1$) and $x_2 + x_3 - x_5 = 0$ (with $x_1 = 0$ and $x_4 = 1$). When the clique constraints are inequalities, the slack variable has to be included in the branching but need not be explicitly introduced, e.g., $x_1 + x_2 + x_3 + x_4 + x_5 \leqslant 1$ can be branched on with two branches, $x_1 + x_2 \leqslant 1$ (with $x_3 = 0$, $x_4 = 0$, and $x_5 = 0$) and $x_3 + x_4 + x_5 \leqslant 1$ (with $x_1 = 0$ and $x_2 = 0$). Most commercial codes require special ordered sets to be disjoint. However, in terms of the branching rules outlined here, there is no reason to impose this restriction.

The "ordered" in special ordered sets means that the set of variables must be given an order and that the split must be done in that order. However, in some cases there may be no natural order. Then the split has to be determined in some way "on the fly" rather than using a predetermined order. In problems where the order makes sense, it can provide for more powerful branching.

Consider the fleet assignment problem,[56] in which aircraft types have to be assigned to flight segments based on forecasted demands. Since exactly one aircraft type has to be assigned to each flight segment, the model contains constraints

$$\sum_{t \in T} x_{st} = 1 \quad \forall s \in S,$$

where T is the set of aircraft types and S is the set of flight segments, and $x_{st} = 1$ if aircraft type t is assigned to flight s. The set of aircraft types is naturally ordered based on passenger capacities. A split now corresponds to deciding that the capacity on a flight segment has to be less than or greater than a certain number of seats. Such a number may be determined from the solution to the LP relaxation.

The above branching scheme can also be viewed as branching on constraints rather than branching on variables. Another example is branching on the connectivity constraints in the TSP formulation given earlier. Recall that connectivity in the TSP can be ensured by the constraints

$$\sum_{e \in \delta(U)} x_e \geqslant 2 \quad \forall U \subset V,\ 2 \leqslant |U| \leqslant \left\lfloor \left| \frac{V}{2} \right| \right\rfloor.$$

In fact, in any tour $z = \Sigma_{e \in \delta(U)} x_e$ will be equal to $2k$ for some $k \geqslant 1$. Therefore, if this is not the case, for instance, for U', we have $z = 2.5$; then we can branch on a connectivity constraint by defining one branch to be the set of tours that satisfy

$$\sum_{e \in \delta(U')} x_e = 2$$

and defining the other branch to be the set of tours that satisfy

$$\sum_{e \in \delta(U')} x_e \geqslant 4.$$

Finally, we return to the example mentioned in the formulation section with the constraint: *no more than two variables from an ordered set can be positive, and if there are two positive variables, then their indices must be consecutive*. These are special ordered sets of type 2 (SOS2). Here we show how to enforce SOS2 constraints by branching.

Again, consider

$$x_1 + x_2 + x_3 + x_4 + x_5 = 1$$

and suppose {1, 2, 3, 4, 5} is an SOS2 set. The LP solution $x_1 = 0$, $x_2 = 0.5$, $x_3 = 0$, $x_4 = 0.5$, and $x_5 = 0$ does not satisfy SOS2. Note that x_2 positive implies $x_4 = 0$ and $x_5 = 0$ and that x_4 positive implies $x_1 = 0$ and $x_2 = 0$. Thus, we can branch by imposing $x_1 = 0$ and $x_2 = 0$ on one child node and $x_4 = 0$ and $x_5 = 0$ on the other. The current LP solution is infeasible to the problems at both child nodes. This is known as SOS2 branching. Note that $x_3 = 0$ does not appear on either branch, which is the difference between SOS1 and SOS2 branching.

Many other logical relationships can be handled by branching. For example, a variable x with domain 0, $[a, b]$ with $a > 0$ is called *semi-continuous*. Excluding an LP solution with $0 < x < a$ obviously can be done by branching.

The LP-based branch-and-bound algorithm for integer programming was developed by Land and Doig.[69] The following papers are representative of the research on branching: Beale,[12] Bénichou et al.,[13] Driebeek,[37] Forrest et al.,[40] and Tomlin.[86] A recent survey of branching techniques is presented by Linderoth and Savelsbergh.[73]

4. Primal Heuristics

As mentioned in the introduction, large trees are the result of many branchings, and avoiding branching, insofar as is possible, is equivalent to avoiding the case $z(k) > z_{best}$. Recall that the variable choice rule based on pseudocosts tries to accomplish that by choosing a variable that causes the objective function to deteriorate quickly, i.e., that results in low values of $z(k)$. In this section, we focus on trying to avoid the

case $z(k) > z_{best}$ by improving the value z_{best}, i.e., by finding better feasible solutions. For large-scale instances there is yet another maybe even more important reason to put some effort in finding good feasible solutions: good feasible solutions may be all we can hope for!

Obviously, if we know the structure of the problem, then any known heuristic for it can be used to provide an initial value z_{best}. For example, branch-and-cut algorithms for the TSP may use the iterated Lin-Kernighan heuristic[75] to get an initial, usually very good, feasible solution. Such heuristics do not use any LP information. In this section, we concentrate on LP-based heuristics that can be used throughout the solution process.

Machine scheduling contains several examples of optimization problems for which effective LP-based heuristics have been developed.[55] The basic idea of the LP-based heuristics for machine scheduling problems is to infer an ordering of the jobs from the solution to the LP relaxation of some IP formulation of the problem. Recently, there has been theoretical as well as empirical evidence that LP-based heuristics are among the best available approximation algorithms for machine scheduling. For example, for the problem of minimizing the total weighted completion time on a single machine subject to release dates, it has recently been shown by Goemans[44] that an LP-based approximation algorithm has a competitive performance ratio (worst case performance ratio) of 2, which makes it the best-known approximation algorithm for this problem. In fact, LP-based approximation algorithms are the only known approximation algorithms that give a constant competitive performance ratio for many scheduling problems. Furthermore, empirical studies have revealed that LP-based heuristics also perform well in practice.[83]

Only a few LP-based heuristics for general integer programs have been developed. The pivot-and-complement heuristic for BIPs[8] is based on the observation that in any feasible LP solution in which all the slack variables are basic, all the regular variables must be nonbasic at one of their bounds. This means that all regular variables are either 0 or 1 and, thus, the solution is integer feasible. The idea now is to pivot slack variables into the basis to obtain such LP solutions.

A completely different approach to finding feasible solutions to BIPs is based on enumerating 0-1 vectors in the neighborhood of the current LP solution and checking whether they are feasible. To make this work, the 0-1 vectors must be enumerated in some sensible order. The OCTANE heuristic[7] generates 0-1 vectors by enumerating extended facets of the octahedron. More precisely, given a point x in the hypercube (for example, x is the current LP solution) and a search direction d, the algorithm generates the extended facets of the octahedron in the order in which they are intersected by the ray originating at point x with direction d. The main drawback of this approach is that it seems to be very difficult to define a search direction that has a high probability of leading to feasible solutions. An alternative enumeration scheme[43] examines the rays of the cone generated by the LP basic solution and looks at their intersections with the hyperplanes of the unit cube to determine coordinates of candidate solutions.

Successive rounding heuristics use a simple iterative scheme in which fractional variables are rounded one after the other until either an integral solution is found or infeasibility is detected. Obviously, the order in which the fractional variables are rounded can have a significant impact on the chance of finding an integral solution. Successive rounding heuristics are often referred to as diving heuristics because rounding of a fractional variable can be viewed as branching on that variable and exploring only one of the child nodes. Unfortunately, the performance of successive rounding heuristics varies greatly from instance to instance.

5. Branch-and-Cut

A *valid inequality* for a MIP is an inequality that is satisfied by all feasible solutions. Here we are interested in valid inequalities that are not part of the current formulation and that are not satisfied by all feasible points to the LP relaxation. Such valid inequalities are called *cuts*. A cut that is not satisfied by the given optimal solution to the LP relaxation is called a *violated cut*. If we have a violated cut, we can add it to the LP relaxation and tighten it. By doing so, we modify the current formulation in such a way that the LP feasible region becomes smaller but the MIP feasible region does not change. Then we can resolve the LP and repeat the process, if necessary, so long as we can continue to find violated cuts.

Branch-and-cut is simply a generalization of branch-and-bound where, after solving the LP relaxation and having not been successful in pruning the node on the basis of the LP solution, we try to find a violated cut. If one or more violated cuts are found, they are added to the formulation and the LP is solved again. If no violated cuts are found, we branch. Branch-and-cut generalizes both pure cutting plane algorithms in which cuts are added at the root node until an optimal MIP solution is found as well as branch-and-bound.

Consider the IP

$$\max 115x_1 + 60x_2 + 50x_3 + 30x_4$$

subject to

$$93x_1 + 49x_2 + 37x_3 + 29x_4 \leqslant 111$$

$$x_j = 0 \text{ or } 1,\ j = 1, \ldots, 4.$$

The optimal LP solution is $x^* = (0.796, 0, 1, 0)$. Since $x_1 = 1$ implies that all other variables have to be equal to 0 and when $x_1 = 0$ at most two of the other variables can be equal to 1, the constraint

$$2x_1 + x_2 + x_3 + x_4 \leqslant 2$$

is a valid inequality. Moreover, it is a violated cut since it is not satisfied by x^*. By adding this constraint to the LP relaxation, we tighten the LP bound.

Given a solution to the LP relaxation of a MIP that does not satisfy all of the integrality constraints, the *separation* problem is to find a violated cut. It can be shown that a violated cut must exist. In particular, there must be a vio-

lated cut among the finite set of linear inequalities that define the set of all convex combinations of feasible solutions. For general MIPs or IPs, these so-called *facet defining* inequalities are extremely difficult to find and to separate. Moreover, there can be a tradeoff between the strength of cuts and the time it takes to find them. Since when no cut is found in reasonable time we can always branch, separation routines are frequently based on fast heuristics that may fail to find a cut of a certain type even though such cuts exist.

At a very high level, there are three types of valid inequalities that can be used to achieve integrality.

Type I—No Structure. These inequalities are based only on variables being integral or binary. Therefore, they can always be used to separate a fractional point. However, they may not be very strong.

Type II—Relaxed Structure. These inequalities are derived from relaxations of the problem, for example, by considering a single row of the constraint set. Therefore, they can at best only separate fractional points that are infeasible to the convex hull of the relaxation. However, these inequalities are frequently facets of the convex hull of the relaxation and therefore may be stronger than Type I inequalities.

Type III—Problem-Specific Structure. These inequalities are typically derived from the full problem structure or from a substantial part it. They are usually very strong in that they may come from known classes of facets of the convex hull of feasible solutions. Their application is limited to the particular problem class and to the known classes of inequalities for that problem class.

We now give examples of the three types of cuts.

The earliest and best known class of Type I inequalities are the Gomory-Chvátal (GC) inequalities for PIPs introduced by Gomory in the 1950s.[46, 27] If all the variables in the inequality

$$\sum_j a_{ij}x_j \leqslant b_i$$

are required to be nonnegative integers, then a GC cut is given by

$$\sum_j \lfloor a_{ij} \rfloor x_j \leqslant \lfloor b_i \rfloor,$$

where $\lfloor \alpha \rfloor$ is the integer part or round down of the real number α. When the a_{ij} are integers and b_i is not an integer, the GC inequality tightens the LP relaxation. If there is a fractional variable in an LP-optimal basic solution, it is well-known that a GC cut can be derived easily from a row with a fractional basic variable. Moreover, a PIP can be solved solely by adding GC cuts to optimal LP solutions that do not satisfy the integrality conditions. This approach is easily extended to inequalities with both integer and continuous variables.[45]

Early computational experiments with pure cutting plane algorithms using GC cuts yielded very slow convergence. However, recent empirical results with GC cuts in a branch-and-cut algorithm have been more positive.[6]

A different class of Type I inequalities that has yielded some positive computational results in a branch-and-cut algorithm for both BIPs and MBIPs is the class of *lift-and-project* inequalities.[5] These cuts are derived from the following basic ideas from disjunctive programming:

1. x_j binary is equivalent to $x_j = x_j^2$.
2. The convex hull of feasible solutions to a MBIP can be obtained by taking the convex hull with respect to one binary variable and then iterating the process.

After selecting a binary variable x_j, the original constraint system $Ax \leqslant b$ is multiplied by x_j and separately by $(1 - x_j)$. Then x_j^2 is replaced by x_j and, for $k \neq j$, $x_k x_j$ is replaced by y_k, where y_k satisfies $y_k \leqslant x_k$, $y_k \leqslant x_j$, and $y_k \geqslant x_k + x_j - 1$. The cuts are obtained in the (x, y)-space and then projected back to the x-space. Separation for these inequalities requires the solution of an LP problem and, therefore, can be very expensive. However, given any fractional solution, a violated inequality can always be found.

The first Type II inequalities used in the solution of MBIPs were derived from knapsack relaxations.[31] Consider a knapsack inequality

$$\sum_{j \in N} a_j x_j \leqslant b,\ x_j \in \{0, 1\} \text{ for } j \in N.$$

Every row of a BIP is of this form and MBIP rows without continuous variables are as well. Also, we can relax a MBIP row with bounded continuous variables to a pure 0-1 row by fixing continuous variables to a bound depending on the sign of its coefficient. Without loss of generality, for a single knapsack inequality, we can assume that $a_j > 0$ since if $a_j < 0$, we can replace x_j by the binary variable $1 - x_j$. Suppose C is a minimal set such that

$$\sum_{j \in C} a_j > b.$$

Then C is called a *minimal cover* and we have the valid *cover inequality*

$$\sum_{j \in C} x_j \leqslant |C| - 1, \tag{2}$$

which gives a facet for the convex hull of solutions to

$$\sum_{j \in C} a_j x_j \leqslant b,\ x_j \in \{0, 1\} \text{ for } j \in C,\ x_j = 0 \text{ for } j \in N \backslash C.$$

To separate on cover inequalities, rewrite (2) as

$$\sum_{j \in C} (1 - x_j) \geqslant 1.$$

Then an LP solution x^* satisfies all of the cover inequalities with respect to the given knapsack if and only if

$$\min\left\{ \sum_{j \in N} (1 - x_j^*) z_j : \sum_{j \in N} a_j z_j > b,\ z_j \in \{0, 1\} \text{ for } j \in N \right\} \geqslant 1.$$

If the minimum is less than 1, then $C = \{j \in N : z_j = 1\}$ defines a cover inequality that is most violated by x^*. This separation problem is a knapsack problem, which is typically solved heuristically using a greedy algorithm. The greedy solution

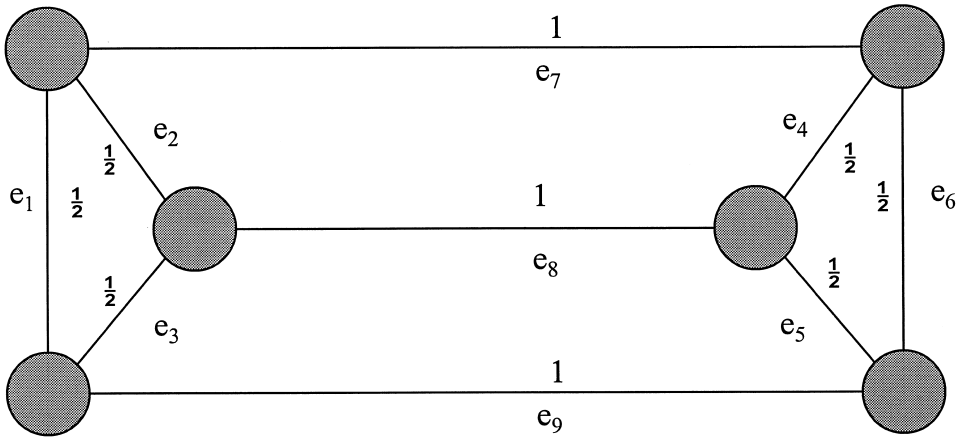

Figure 2. TSP instance with LP solution.

defines a minimal cover that may or may not be violated by x^*.

To make this inequality strong, it must be *lifted* to include the variables in $N\backslash C$. However, before lifting, it is advantageous to fix the variables with $x_j^* = 1$ from the cover inequality. This yields the lower dimensional inequality

$$\sum_{j \in C_1} x_j \leq |C_1| - 1, \tag{3}$$

where $C_1 = \{j \in C : x_j < 1\}$, which is valid if $x_j = 1$ for $j \in C_2 = C\backslash C_1$.

Now (3) can be lifted to a valid inequality for the original knapsack problem of the form

$$\sum_{j \in C_1} x_j + \sum_{j \in N\backslash C} \alpha_j x_j + \sum_{j \in C_2} \gamma_j x_j \leq |C_1| - 1 + \sum_{j \in C_2} \gamma_j.$$

This inequality, called a *lifted cover inequality*, is at least as strong and generally stronger than the initial cover inequality, and if the coefficients α_j for $j \in N\backslash C$ and γ_j for $j \in C_2$ are as large as possible, the inequality gives a facet of the convex hull of solutions to the original knapsack problem. The coefficients can be computed by dynamic programming. Lifted cover inequalities have proven to be very effective in branch-and-cut algorithms for BIPs and MBIPs.[52]

Type II cuts can be improved by tightening the relaxation from which they are derived. For example, for problems with clique (sometimes called GUB) inequalities, the knapsack relaxation can be tightened by considering the relaxation formed by the knapsack inequality and the clique inequalities. From this relaxation, we can derive *lifted GUB cover inequalities* that, when applicable, are stronger than lifted cover inequalities. However, they are more expensive to compute. Generally, tighter relaxations yield stronger inequalities but are more expensive to compute.

Cover inequalities can only be derived from rows with continuous variables by fixing these variables at bounds. To obtain stronger inequalities from rows with both binary and continuous variables, different relaxations are needed. Two kinds of relaxations have been used to derive classes of Type II inequalities that have been used successfully in branch-and-cut algorithms.

Flow cover inequalities[81] are derived from a system consisting of a flow balance equation

$$\sum_{j \in N^+} y_j - \sum_{j \in N^-} y_j = d$$

and variable upper bound constraints on the flows

$$y_j \leq m_j x_j \quad j \in N,$$

where $N = N^+ \cup N^-$. The system can be viewed as a single node-capacitated network design model. The y variables are arc flows that are constrained by the flow balance equation, with an external demand of d, and the variable upper bound constraints, where m_j is the capacity of arc j and x_j is a 0-1 variable that indicates whether arc j is open or not. The usefulness of this system for general MBIPs arises from the fact that a general inequality in a MBIP that contains both continuous and binary variables can be relaxed to this form.

Recently, classes of Type II inequalities have been obtained from a relaxation of the form

$$\sum_{j \in N} a_j x_j \leq b + s,\ x_j \in \{0, 1\} \text{ for } j \in N,\ s \geq 0,$$

which is called a *continuous knapsack set*.[74]

To illustrate the Type III inequalities, we again consider the TSP and the example shown in Fig. 2. Suppose we solve the LP relaxation consisting of the degree, connectivity, and nonnegativity constraints. The solution $x = (\frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}, 1, 1, 1)$ satisfies these constraints and could be an optimal solution to the LP relaxation since it is an extreme point. This fractional solution is cut off by the valid inequality

$$x_1 + x_2 + x_3 + x_7 + x_8 + x_9 \leq 4.$$

The inequality is valid since, if we look at the subgraph generated by these edges and try to put as many of them as possible in a tour, we could include e_7, e_8, and e_9, but then the degree constraints would forbid us from taking more than one edge from e_1, e_2, and e_3. Similarly, if we take two

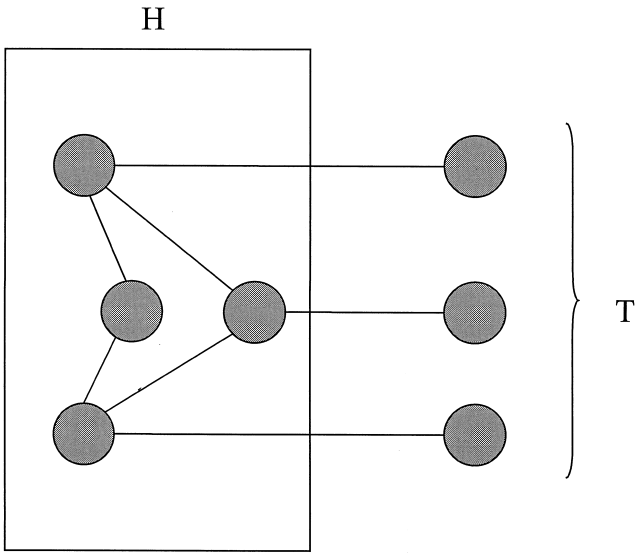

Figure 3. Simple comb.

edges from e_1, e_2, and e_3, then we can take no more than two edges from e_7, e_8, and e_9.

In general, consider a subset of nodes H and an odd set of node disjoint edges T, each having one end in H as shown in Fig. 3. The figure resembles a comb, with handle H and teeth T. Let $E(H)$ be the set of edges with both ends in H. How many edges can we take from $E(H) \cup T$ in a tour? Arguing as above, we see that we should take all of T, but then a simple counting argument shows that the maximum number of edges we can take from $E(H)$ is $|H| - \lceil |T|/2 \rceil$. Hence, we get the *simple comb* inequality[28]

$$\sum_{e \in E(H)} x_e + \sum_{e \in T} x_e \leq |H| + \lfloor |T|/2 \rfloor.$$

These simple comb inequalities can be separated in polynomial time, and in some instances they define facets of the convex hull of tours. It is interesting to note that they can be derived as Gomory-Chvátal cuts. Moreover, they can be generalized by also using the subtour elimination constraints.[50, 51]

The TSP also demonstrates where cuts can be used to exclude integer solutions that are not feasible. The initial formulation of the TSP cannot include all of the connectivity constraints since the number of them is exponential in the size of the TSP input. Without all of the connectivity constraints, an optimal solution to the LP relaxation can be integral but not feasible, i.e., a set of two or more subtours that contain all of the nodes. When this occurs, at least one connectivity constraint must be a violated cut. Connectivity constraints can also be used to cut off fractional solutions.

There is a vast literature on specialized branch-and-cut algorithms for many combinatorial optimization problems. A network design problem will be discussed in Section 11. For some families of cuts, such as connectivity constraints and combs, the separation problem can be solved exactly and rapidly. For more complex families of cuts, especially when the separation problem is NP-hard, we use heuristics that may fail to find an existing violated cut. The problem of how much time should be spent on finding violated cuts, with failure meaning wasted computing time, must be resolved empirically on a case-by-case basis.

The use of cutting planes to solve combinatorial optimization problems dates back to the early years of the field when Dantzig et al.[32, 33] were able to solve a 49-city traveling salesman problem. Caprara and Fischetti[26] provide numerous references on branch-and-cut algorithms. A survey of the developments specifically for the TSP can be found in Jünger et al.[64] Among the first papers to describe the use of cutting planes throughout the branch-and-bound tree are Grötschel et al.[47] and Padberg and Rinaldi.[80]

6. Branch-and-price

The *generalized assignment problem* (GAP) is the problem of finding a maximum profit assignment of m tasks to n machines such that each task is assigned to precisely one machine, subject to capacity restrictions on the machines. The standard integer programming formulation of GAP is

$$\begin{aligned} \max \sum_{1 \leq i \leq m} \sum_{1 \leq j \leq n} p_{ij} z_{ij} & \\ \sum_{1 \leq j \leq n} z_{ij} &= 1 \quad i = 1, \ldots, m \\ \sum_{1 \leq i \leq m} w_{ij} z_{ij} &\leq d_j \quad j = 1, \ldots, n \\ z_{ij} &\in \{0, 1\} \quad i = 1, \ldots, m, \\ & \quad\; j = 1, \ldots, n, \end{aligned} \tag{4}$$

where p_{ij} is the profit associated with assigning task i to machine j, w_{ij} is the amount of the capacity of machine j used by task i, d_j is the capacity of machine j, and z_{ij} is a 0-1 variable indicating whether task i is assigned to machine j.

Alternatively, the problem can be viewed as that of partitioning the set of tasks into subsets that can be performed on a specific machine. This yields the following (re)formulation:

$$\begin{aligned} \max \sum_{1 \leq j \leq n} \sum_{1 \leq k \leq K_j} \left(\sum_{1 \leq i \leq m} p_{ij} y^j_{ik} \right) \lambda^j_k & \\ \sum_{1 \leq j \leq n} \sum_{1 \leq k \leq K_j} y^j_{ik} \lambda^j_k &= 1 \quad i = 1, \ldots, m \\ \sum_{1 \leq k \leq K_j} \lambda^j_k &= 1 \quad j = 1, \ldots, n \\ \lambda^j_k &\in \{0, 1\} \quad j = 1, \ldots, n, \\ & \quad\; k = 1, \ldots, K_j, \end{aligned} \tag{5}$$

where the first m entries of a column, given by $y^j_k = (y^j_{1k}, y^j_{2k}, \ldots, y^j_{mk})$, satisfy the knapsack constraint

$$\sum_{1 \leq i \leq m} w_{ij} x_i \leq d_j$$

$$x_i \in \{0, 1\} \quad i = 1, \ldots, m,$$

and K_j denotes the number of feasible solutions to the knapsack constraint. In other words, the first m entries of a column represent a feasible assignment of tasks to a machine.

The LP relaxation of (5) can be obtained from a Dantzig-Wolfe decomposition of the LP relaxation of (4). The LP relaxation of (5) is tighter than the LP relaxation of (4) since fractional solutions that are not convex combinations of 0-1 solutions to the knapsack constraints are not feasible to (5). However, the number of columns in (5) is exponential in the size of the input. Fortunately, it is possible to handle the huge number of columns implicitly rather than explicitly. The basic idea is simple. Leave most columns out of the LP relaxation because there are too many columns to handle efficiently; most of them will have their associated variable equal to zero in an optimal solution anyway. Then, as in column generation for linear programming, to check the optimality of an LP solution, a subproblem, called the *pricing problem*, is solved to try to identify columns to enter the basis. If such columns are found, the LP is reoptimized; if not, we are done. To solve the LP relaxation of the set-partitioning formulation of the GAP, pricing or column generation is done by solving n knapsack problems.

Obviously, the LP relaxation may not have an integral optimal solution and then we have to branch. However, applying a standard branch-and-bound procedure over the existing columns is unlikely to find an optimal (or even good or even feasible) solution to the original problem. Therefore, it may be necessary to generate additional columns in order to solve the LP relaxations at non-root nodes of the search tree. Branch-and-bound algorithms in which the LP relaxations at nodes of the search tree are solved by column generation are called *branch-and-price* algorithms.

There are two fundamental difficulties in applying column generation techniques to solve the linear programs occurring at the nodes of the search tree.

- Conventional integer programming branching on variables may not be effective because fixing variables can destroy the structure of the pricing problem.
- Solving these LPs and the subproblems to optimality may not be efficient, in which case different rules will apply for managing the search tree.

Consider the set-partitioning formulation of the GAP. Standard branching on the λ-variables creates a problem along a branch where a variable has been set to zero. Recall that y_k^j represents a particular solution to the jth knapsack problem. Thus, $\lambda_k^j = 0$ means that this solution is excluded. However, it is possible (and quite likely) that the next time the jth knapsack problem is solved, the optimal knapsack solution is precisely the one represented by y_k^j. In that case, it would be necessary to find the second-best solution to the knapsack problem. At depth l in the branch-and-bound tree, we may need to find the lth-best solution. Fortunately, there is a simple remedy to this difficulty. Instead of branching on the λs in the master problem, we use a branching rule that corresponds to branching on the original variables z_{ij}. When $z_{ij} = 1$, all existing columns in the master problem that do not assign task i to machine j are deleted and task i is permanently assigned to machine j, i.e., variable x_i is fixed to 1 in the jth knapsack. When $z_{ij} = 0$, all existing columns in the master problem that assign job i to machine j are deleted, and task i cannot be assigned to machine j, i.e., variable x_i is removed from the jth knapsack. Note that each of the knapsack problems contains one fewer variable after the branching has been done.

One reason for considering formulations with a huge number of variables is that they may provide tighter bounds than compact formulations. Another reason for considering such formulations is that when a compact formulation has a symmetric structure, it causes branch-and-bound algorithms to perform poorly because the problem barely changes after branching. A formulation with a huge number of variables may eliminate this symmetry.

Consider the GAP in which all machines are identical. With identical machines, there is an exponential number of solutions that differ only by the names of the machines, i.e., by swapping the assignments of two machines we get two solutions that are the same but have different values for the variables. This statement is true for fractional as well as for 0-1 solutions. The implication is that when a fractional solution is excluded at some node of the tree, it pops up again with different variable values somewhere else in the tree. In addition, the large number of alternate optima dispersed throughout the tree renders pruning by bounds nearly useless.

Here, the set-partitioning reformulation provides an alternative in which it is easier to handle the symmetry. First, observe that in the set-partitioning formulation of the GAP with identical machines we do not need a separate subproblem for each machine. This implies that the λ_k^j can be aggregated by defining $\lambda_k = \Sigma_j \lambda_k^j$ and that the convexity constraints can be combined into a single constraint $\Sigma_{1 \leq k \leq K} \lambda_k = n$ where λ_k is restricted to be integer.

In some cases the aggregated constraint will become redundant and can be deleted altogether. An example of this is when the objective is to minimize $\Sigma \lambda_k$, i.e., the number of machines needed to process all the tasks. Note that this special case of GAP is equivalent to a 0-1 cutting stock problem.

To handle the symmetry, we apply a branching scheme that focuses on pairs of tasks. In particular, we consider rows corresponding to tasks r and s. Branching is done by dividing the solution space into one set in which r and s appear together, in which case they can be combined into one task when solving the knapsack and into another set in which they must appear separately, in which case a constraint $x_r + x_s \leq 1$ is added to the knapsack.

In the reformulation of the GAP, the original formulation has been decomposed into a set-partitioning master problem and knapsack pricing problem. Such a decomposition frequently occurs in formulations with a huge number of variables and often has a natural interpretation in the contextual setting allowing for the incorporation of additional important constraints. As an example, consider a distribution problem in which a set of trucks, each with a certain capacity, has to make deliveries to a set of customers, each with a

certain demand. In the set-partitioning formulation of this problem, rows represent customers and columns represent feasible delivery routes. Additional constraints, such as delivery time windows for the customers, only affect the pricing problem and, therefore, are easier to incorporate than in a compact formulation of the problem.

Johnson[62] was one of the first to realize the potential and complexity of branch-and-price algorithms. The ideas and methodology of branch-and-price as well as a survey of some of its application areas can be found in Barnhart et al.[10] Routing and scheduling has been a particularly fruitful application area of branch-and-price. Desrosiers et al.[35] survey these results.

7. Row and Column Management

Although the theoretical foundations of branch-and-cut and branch-and-price algorithms are now well-known and well-documented, there are still many algorithmic and implementation issues that need to be resolved or require more thorough investigation. In this section, we focus on some of these computational issues.

In branch-and-cut algorithms, the search for violated cuts to tighten the LP relaxation affects computation time in two ways. First, there is the additional time spent on trying to generate violated cuts, and this time is spent regardless of whether any violated cuts are found. Second, if one or more violated cuts are found, they are added to the active linear program, which is then resolved. Therefore, we may be solving several linear programs per node. Moreover, the linear programs become larger and typically harder to solve. Consequently, we have to be careful and try to make sure that the extra time spent on evaluating nodes of the search tree does result in a smaller search tree and, more importantly, does result in a smaller overall solution time.

Cut management refers to strategies embedded in branch-and-cut algorithms to ensure effective and efficient use of cuts in an LP-based branch-and-bound algorithm. These strategies decide when to generate cuts, which of the generated cuts (if any) to add to the active linear program, and when to delete previously generated cuts from the active linear program.

Several strategies have been investigated that try to decrease the time spent on generating cuts without reducing the effectiveness of the branch-and-cut algorithm. This can be done in two ways: 1) by limiting the number of times cuts are generated during the evaluation of a node, or 2) by not generating cuts at every node of the search tree.

The number of times violated cuts are generated during the evaluation of a node is often referred to as the *rounds of cut generation*. A popular strategy is to limit the rounds of cut generation to k_1 in the root node and to k_2 in all the other nodes with $k_1 > k_2$. Limiting the number of times violated cuts are generated during the evaluation of a node is not only important for reducing time spent on cut generation, but it also prevents so-called *tailing-off*. Tailing-off refers to the phenomenon observed frequently that, after several rounds of cut generation, the objective function value hardly changes, i.e., even though violated cuts are identified and added to the active linear program, their addition does not lead to a substantial bound improvement. When tailing-off occurs, it may be beneficial to branch rather than generate cuts, since we may be spending a significant amount of time on cut generation without producing stronger bounds and, thus, without decreasing the size of the search tree.

Tailing-off raises the important issue of how to measure the quality of a cut. One measure of quality that has been used and appears to be reasonably effective is the orthogonal distance from the current optimal LP point to the cut. This measure is used in the obvious way: only add a violated cut if the orthogonal distance from the current optimal LP point to the cut is larger than a certain threshold value.

Several strategies have been developed that generate violated cuts at only a subset of the nodes of the search tree. The simplest such strategy is to generate violated cuts only at the root node. The resulting branch-and-cut algorithm is often called a *cut-and-branch* algorithm. A generalization of this strategy generates cuts only in the top part of the search tree, i.e., at all nodes with depth less than or equal to some given parameter t. The rationale behind these strategies is that generating cuts in the top part of the search tree is more important since it affects all the nodes in the search tree. Another strategy is to generate violated cuts at every tth node. The rationale behind this strategy is that after evaluating t nodes we have hopefully entered a different part of the search space and we will be able to generate violated cuts relevant for it.

Another reason for the increase in time to evaluate nodes is that the addition of cuts leads to larger and more complex linear programs. To minimize the negative effects of larger and more complex linear programs, a strategy is required that controls the size of the active linear program. (Note that we have already seen one possible way to accomplish this—only adding some of the generated cuts, e.g., only adding the k cuts with the highest orthogonal distance from the current optimal LP point to the cut.) The basic idea behind most control strategies is to have only a limited number of all the available cuts in the active linear program. Since most of the cuts will not be binding in an optimal solution anyway, this seems reasonable. The cuts that are not part of the active linear program are kept in a so-called *cut pool*.

Combining these ideas, we obtain the following basic cut management scheme:

1. Solve the active linear program.
2. Identify inactive cuts. If successful, delete them from the active linear program and move them to the cut pool.
3. Search the cut pool for violated cuts. If successful, select a subset of them, add them to the active linear program, and go to 1.
4. Generate violated cuts. If successful, add them to the cut pool and go to 3.

Many implementation issues need to be addressed relating to this scheme. First, we want to minimize the administrative overhead as much as possible. For example, we do not want to move cuts back and forth between the active linear program and the cut pool all the time. Therefore, we have to be careful in defining when a cut becomes inactive.

In the typical definition, a cut is called inactive if the dual variable associated with it has been equal to zero for k consecutive iterations. Second, we want to minimize the time it takes to search the cut pool for violated cuts. Therefore, a cut pool typically has a fixed size: the *cut pool size*. A small cut pool size obviously results in faster search times.

Most of the issues discussed in this section in the context of branch-and-cut algorithms apply to branch-and-price algorithms as well.

8. Software

Anyone who has ever attempted to use integer programming in practice knows that the road from a real-life decision problem to a satisfactory solution can be quite long and full of complications. The process involves developing a model (which typically involves making simplifying assumptions), generating an instance of the model (which may involve gathering huge amounts of data), solving the instance (which involves transforming the instance data into a machine-readable form), verifying the solution and the model (which involves validating the appropriateness of the simplifying assumptions), and, if the need arises, repeating these steps. In addition, models may have to be modified when changes occur in the problem or when user needs become different. This iterative process represents the *modeling life cycle* in which a model evolves over time.

Although this paper concentrates on how to solve integer programs, it is important to realize that this is only a single step in the overall process. Ideally, a computer-based integer-programming modeling environment has to nurture the entire modeling life cycle and has to support the management of all resources used in the modeling life cycle, such as data, models, solvers, and solutions.

Two components of this ideal modeling environment have received the most attention and are now widely available—the modeling module and the solver module. A modeling module provides an easy to use yet powerful language for describing a decision problem. Most modeling modules available today are based on an algebraic paradigm to conceptualize the model. A solver module provides an optimizer that reads an instance produced by the modeling module and then solves it. Obviously, the modeling module and the solver module need to be able to communicate with each other. The simplest but least efficient way to accomplish this is through files. The MPS format is a generally accepted format for the specification of an instance of an integer linear program. All modeling modules are able to produce an MPS input file and to process an MPS output file. All solver modules are able to process an MPS input file and to produce an MPS output file. Since reading and writing files are slow processes, other more efficient interfaces have also been developed. There are also some integrated systems that provide both a modeling module and a solver module.

As we have emphasized throughout the paper, it is often essential to use the structure of the underlying problem to enhance the performance of general-purpose integer-programming software. Such enhancements range from specifying a priority order to be used for branching to developing primal heuristics and cut generation routines. To support such special-purpose functions, software developers have started to provide either "hooks" into the software that allow users to add on their own routines or a callable library version of the software so that users can invoke the integer-programming optimizer from within their own programs.

Although the availability of hooks and callable libraries greatly facilitates the development of special-purpose optimizers, it still remains time consuming because it usually requires a deep and often low-level understanding of the software. To overcome these difficulties and, thus, to speed up the development process, systems such as ABACUS[66] and MINTO[78] have been built that run on top of the commercial optimizers and provide an easier-to-use interface for the development of a special-purpose optimizer. Using these systems, a full-blown special-purpose branch-and-cut algorithm incorporating primal heuristics, special branching schemes, and specific cut generation routines typically can be developed in a matter of weeks rather than months.

Widely used modeling systems include GAMS,[23] AMPL,[41] and AIMMS.[16] Solvers include CPLEX,[30] OSL,[60] and XPRESS-MP.[34]

9. Testing and Computation

What makes a good integer-programming optimizer? How do you test the performance of an integer-programming optimizer? What features should a good integer-programming optimizer have? There do not seem to be easy answers to any of these questions.

A common approach to finding some partial answers is to compare the performance of optimizers on a standard set of test problems. For integer programming, MIPLIB is the only standard set of test problems. MIPLIB is a collection of integer programs in the form of MPS files that contains BIPs, MBIPs, and MIPs. It was created in 1992[17] and the latest version (3.0) is described in Bixby et al.[18] However, as indicated earlier, comparisons provide only partial answers. What is important? Total CPU time? Number of nodes in the search tree? Quality of the bound at the root node? This issue is complicated even more by the fact that the performance of optimizers with respect to any of the mentioned measures varies widely from instance to instance. Although the availability of MIPLIB does not allow us to identify the best integer-programming optimizer, it has stimulated the improvement and development of many of the more sophisticated integer-programming techniques discussed in this paper.

There is, however, one important deficiency with MIPLIB. Because MIPLIB contains instances of integer programs in the form of MPS files, the underlying model is completely lost. As indicated earlier, it is often essential to use the underlying problem structure to be able to solve integer linear programs. Therefore, it is important to create a library that contains models as well as instances of the models. This would not only allow the evaluation of general-purpose techniques but also the evaluation of special-purpose techniques for specific types of integer linear-programming

Table I. Instance Characteristics

Name	Number of Constraints	Number of Variables	Number of Integer Variables	Number of 0-1 Variables
EGOUT	98	141	55	all
BLEND2	274	353	264	231
FIBER	363	1298	1254	all
GT2	29	188	188	24
HARP2	112	2993	2993	all
MISC07	212	260	259	all
P2756	755	2756	2756	all
VPM1	234	378	168	all

models, i.e., the evaluation of specific branching rules, primal heuristics, and classes of valid inequalities.

We present a small computational experiment to show the effects of enhancing a linear programming-based branch-and-bound algorithm with some of the techniques discussed in the previous sections. We have run MINTO (version 3.0) with six different settings and a time limit of 30 minutes on a Pentium II processor with 64 Mb internal memory running at 200 Mhz on a subset of the MIPLIB 3.0 instances. The six different settings are:

1. Plain LP-based branch-and-bound selecting the fractional variable with fraction closest to 0.5 for branching and selecting the unevaluated node with the best bound for processing.
2. Add pseudocost branching to 1.
3. Add preprocessing to 2.
4. Add a primal heuristic to 3.
5. Add cut generation to 3.
6. Add a primal heuristic as well as cut generation to 3.

In Table I, we present the number of constraints, the number of variables, the number of integer variables, and the number of 0-1 variables for each of the instances used in our computational experiment. In Table II, we report for each instance its name, the value of the best solution found, the value of the LP relaxation at the root node, the number of evaluated nodes, and the elapsed CPU time. Table III contains summary statistics on the performance for each of the different settings.

The results presented clearly show that these enhancements significantly improve the performance. In particular, the setting that includes all of the enhancements is best (solves most instances) and plain branch-and-bound is worst (solves fewest instances). However, for some instances, the extra computational effort is not worthwhile and even degrades the performance.

In instance *blend2*, we see that cut generation increases the solution time even though it reduces the number of evaluated nodes. Generating cuts takes time and unless it significantly reduces the size of the search tree, this may lead to an increase in the overall solution time. On the other hand, in instances *fiber* and *p2756*, we see that cut generation is essential. Without cut generation, these instances cannot be solved. Cut generation is also very important in solving *vpm1* efficiently. In instance *gt2*, we see that preprocessing is the key to success. Even though cut generation may not always allow you to solve an instance, it may help in finding good feasible solutions. This can be observed in instance *harp2*, where we find a feasible solution within the given time limit only when cuts are generated. In instance *vpm1*, we see that the use of a primal heuristic may sometimes increase the solution time. The optimal solution is found very quickly (although, of course, this is not known at the time), so all the time spent in the primal heuristic after that is wasted.

10. Applications

In this section we present two applications—one that illustrates branch-and-cut and another that illustrates branch-and-price. Moreover, these applications have had a great deal of impact in their respective industries because the solutions obtained have yielded substantial cost savings. The significance of integer-programming models and applications is demonstrated by the number of recent Edelman prize papers that use integer programming, e.g., Arntzen et al.,[4] Bertsimas et al.,[14] Botha et al.,[21] Camm et al.,[24] Hueter and Swart,[59] and Sinha et al.[85]

10.1 Network Design

Fiber-optic technology is rapidly being deployed in communication networks throughout the world because of its nearly unlimited capacity, reliability, and cost-effectiveness. The high capacity provided by fiber optics results in much sparser network designs with larger amounts of traffic carried by each link than is the case with traditional bandwidth-limited technologies. This raises the possibility of significant service disruptions due to the failure of a single link or single node in the network. Hence, it is vital to take into account such failure scenarios and their potential negative consequences when designing fiber communication networks.

The major function of a communication network is to provide connectivity between users in order to provide a desired service. The term survivability refers to the ability to restore network service in the event of a failure, such as the complete loss of a communication link or a switching node. Service could be restored by means of routing traffic around the damage, if this contingency is provided in the network

Table II. Computational Results on a Subset of MIPLIB 3.0 Instances

Name	Setting	z_{best}*	z_{root}	Number of Nodes	CPU (sec.)
EGOUT	1	568.10	149.59	60779	561.97
EGOUT	2	568.10	149.59	8457	39.14
EGOUT	3	568.10	511.62	198	1.13
EGOUT	4	568.10	511.62	218	1.22
EGOUT	5	568.10	565.95	3	0.33
EGOUT	6	568.10	565.95	3	0.30
BLEND2	1	7.60	6.92	31283	928.92
BLEND2	2	7.60	6.92	17090	226.66
BLEND2	3	7.60	6.92	14787	141.39
BLEND2	4	7.60	6.92	12257	126.50
BLEND2	5	7.60	6.98	10640	381.56
BLEND2	6	7.60	6.98	8359	359.11
FIBER	1		156082.52	30619	1800.00
FIBER	2	405935.18	156082.52	83522	1800.00
FIBER	3	405935.18	156082.52	81568	1800.00
FIBER	4	413622.87	156082.52	74286	1800.00
FIBER	5	405935.18	384805.50	677	39.89
FIBER	6	405935.18	384805.50	379	34.75
GT2	1		13460.23	38528	1800.00
GT2	2		13460.23	39917	1800.00
GT2	3	21166.00	20146.76	1405	8.80
GT2	4	21166.00	20146.76	1912	12.48
GT2	5	21166.00	20146.76	1405	7.16
GT2	6	21166.00	20146.76	1912	12.53
HARP2	1		−74353341.50	24735	1800.00
HARP2	2		−74353341.50	37749	1800.00
HARP2	3		−74325169.35	37813	1800.00
HARP2	4		−74325169.35	33527	1800.00
HARP2	5	−73899781.00	−74207104.67	26644	1800.00
HARP2	6	−73899727.00	−74207104.67	21631	1800.00
MISC07	1	2810.00	1415.00	12278	1800.00
MISC07	2	2810.00	1415.00	43583	1244.67
MISC07	3	2810.00	1415.00	61681	1635.69
MISC07	4	2810.00	1415.00	66344	1800.00
MISC07	5	2810.00	1415.00	62198	1800.00
MISC07	6	2810.00	1415.00	43927	1300.52
P2756	1		2688.75	33119	1800.00
P2756	2		2688.75	35383	1800.00
P2756	3		2701.14	37707	1800.00
P2756	4	3334.00	2701.14	33807	1800.00
P2756	5	3124.00	3117.56	461	140.66
P2756	6	3124.00	3117.56	137	74.56
VPM1	1		15.42	42270	1800.00
VPM1	2	20.00	15.42	179402	1631.84
VPM1	3	20.00	16.43	77343	624.14
VPM1	4	20.00	16.43	146538	1339.64
VPM1	5	20.00	19.25	16	0.72
VPM1	6	20.00	19.25	1	0.84

*No value means that no feasible solution was found.

Table III. Summary Statistics

Settings	1	2	3	4	5	6
Number of problems solved	2	4	5	4	6	7
Number of times with the fewest number of nodes	0	1	1	0	2	5
Number of times with the fastest solution time	0	1	0	1	2	3

architecture. A network topology could provide protection against a single link failure if it remains connected after the failure of any link. Such a network is called 2-edge connected since at least two edges have to be removed to disconnect the network. Similarly, protection against a single node failure can be provided by a 2-node connected network. In the case of fiber-optic communication networks for telephone companies, two-connected topologies provide an adequate level of survivability since most failures typically can be repaired quickly, and a statistical analysis has shown that it is very unlikely that a second failure will occur while the first failure is being repaired.

The survivability conditions are expressed is terms of edge or node connectivity requirements of the graph $G = (V, E)$ representing the network. For each node $s \in V$, a nonnegative integer r_s is specified. We say that the network $N = (V, F)$ to be designed, with $F \subseteq E$, satisfies the node survivability conditions if, for each pair of nodes $s, t \in V$ of distinct nodes, N contains at least $r_{st} = \min\{r_s, r_t\}$ node-disjoint s, t-paths. Similarly, we say that the network $N = (V, F)$ satisfies the edge survivability conditions if, for each pair of nodes $s, t \in V$ of distinct nodes, N contains at least r_{st} edge-disjoint s, t-paths.

The design of a survivable network now can be formulated as the integer program

$$\begin{aligned} \min \quad & cx \\ & x(\delta_G(W)) \geqslant con(W) \quad \text{for all } W \subseteq V,\ \emptyset \neq W \neq V \end{aligned} \tag{6}$$

$$\begin{aligned} & x(\delta_{G-Z}(W)) \geqslant con(W) - |Z| \quad \text{for all pairs } s, t \in V,\ s \neq t, \text{ and} \\ & \qquad \text{for all } \emptyset \neq Z \subseteq V\backslash\{s, t\} \\ & \qquad\quad \text{with } |Z| \leqslant r_{st} - 1 \\ & \qquad \text{and for all } W \subseteq V\backslash Z \\ & \qquad\quad \text{with } s \in W,\ t \notin W \\ & x_{ij} \in \{0, 1\} \qquad \text{for all } ij \in E, \end{aligned} \tag{7}$$

where the variable x_{ij} indicates whether edge $\{i, j\}$ is selected to be part of the network ($x_{ij} = 1$) or not ($x_{ij} = 0$), $x(F) = \Sigma_{e \in F} x_e$, $\delta_G(W)$ denotes the set of edges with exactly one endpoint in W, and $con(W) = \max_{s \in W, t \notin W} r_{st}$. Constraints (6) ensure the edge survivability conditions by requiring that for every pair $s, t \in V$ and any subset $W \subseteq V$ with $s \in W$ and $t \in V\backslash W$, the number of edges in the network with one endpoint in W is at least as large as r_{st}. Constraints (7) ensure the node survivability conditions by requiring that for any pair $s, t \in V$, any subset $Z \subseteq V$ with $s, t \notin Z$ and $|Z| \leqslant r_{st} - 1$, and any subset $W \subseteq V\backslash Z$ with $s \in W$ and $t \in V\backslash(W \cup Z)$, the number of edges in the network with one endpoint in W and one endpoint not in W is at least as large as $r_{st} - |Z|$ when we drop the nodes Z and all edges incident with nodes in Z from the network. Note that (6) is a special case of (7) with $Z = \emptyset$.

The structure of the convex hull of the set of feasible solutions to this integer program has been studied extensively and a branch-and-cut algorithm, based on these results, to compute a minimum cost network satisfying the survivability conditions has been developed (see Grötschel et al.[48, 49]). Their algorithm has been tested on several real-life instances with sizes ranging from 36 nodes and 65 edges to 116 nodes and 173 edges.

Many real-world instances have sparse graphs that can be decomposed and, thus, simplify the solution. These decompositions are based on the observation that some edges have to be used by any feasible solution. If such edges exist, then it is possible to decompose the problem into several subproblems that can be solved independently. Good heuristics are also available,[77] and the results of the branch-and-cut algorithm showed that these heuristics indeed produce high-quality solutions.

10.2 Crew Scheduling

In a crew-scheduling or -pairing problem, sequences of flights, called pairings, are assigned to crews so that each flight segment is assigned to exactly one crew. The problem is defined by a flight schedule, time period, legality rules, and a cost structure. The time period is dictated by the periodic property of the schedule. Domestic schedules for the major U.S. carriers repeat almost daily, with some exceptions on the weekends. Thus, the period is generally taken to be one day. For international flights, a weekly period is more appropriate. For simplicity, we will consider daily problems. The first segment in a daily pairing must depart from the crew's base, each subsequent segment departs from the station where the previous one arrived, and the last segment must return to the base. A sequence can represent several days of flying, typically up to three to five days for a domestic daily problem.

Good pairings have the crews spending as much of the elapsed time of the pairing as is legally possible flying. Bad pairings have a lot of wasted time between flights. For example, a crew arrives at 11 pm, but because of required rest cannot leave with the airplane that departs at 7 am. Since no other planes of this type arrive at this airport, the crew wastes the day and flies the plane departing at 7 am, 32 hours after its arrival. When this unfortunate situation occurs, there is always an unoccupied crew at the station and two crews are there overnight, even though there is only one arrival and one departure each day.

Pairings are subject to a number of constraints resulting from safety regulations and contract terms. These constraints dictate restrictions such as the maximum number of hours a pilot can fly in a day, the maximum number of days before returning to the base, and minimum overnight rest times. In addition, the cost of a pairing is a messy function of several of its attributes. Wasted time is the main one.

For these reasons, it is not desirable to formulate a crew-scheduling problem with variables z_{ij} where $z_{ij} = 1$ if crew i

is assigned to segment j since the constraints on z_{ij} and the cost are highly nonlinear and difficult to express. The alternative approach is to enumerate (implicitly or explicitly) feasible pairings and then to formulate a set partitioning problems in which each column or variable corresponds to a pairing, each row corresponds to a flight, and the objective is to partition all of the flights into a set of minimum cost pairings.

Although enumerating feasible pairings is complex because of all of the rules that must be observed, it can be accomplished by first enumerating all feasible possibilities for one day of flying and then combining the one-day schedules to form pairings. The major drawback is the total number of pairings, which grows exponentially with the number of flights. Problems with 1,000 flights are likely to have billions of pairings.

One approach for solving a 1,000-flight problem, which is a typical large fleet problem for a U.S. domestic carrier, is to enumerate in advance about 100,000 low-cost pairings, a very small fraction of the total. Then, with this fixed set of columns, attempt to solve or get a good feasible solution to the set-partitioning problem defined by these 100,000 pairings, which itself is a very difficult task. Since only a tiny fraction of all of the pairings are available to the integer program or its linear programming relaxation, this methodology may not produce a solution that is close to being optimal to the problem in which all the pairings are considered.

Theoretically, branch-and-price can implicitly consider all of the pairings. It is possible to represent pairings as suitably constrained paths in a network and then to evaluate their costs, i.e., price out nonbasic columns, using a multilabel shortest path or multistate dynamic programming algorithm. However, in practice, the memory requirements of the full multilabel shortest path and multistate dynamic programming algorithms are prohibitive, and scaled-down versions have to be used.

It may be important to generate columns during the solution of LPs throughout the tree. For example, in a small 400-segment instance, 9,400 pairings were generated to solve the initial LP relaxation. The LP bound was 1,600. The only IP solution that could be found using this small set of candidate pairings had cost 13,000. (The run was terminated after two days of CPU time). By generating columns in the tree, an IP solution with cost 800 was found! This cost is less than the cost of the LP relaxation at the root node because of approximations in the pricing algorithm that prevented fully optimizing the LP relaxation (see Vance et al.[87]).

Crew-pairing problems also illustrate the special kind of branching that is required when column generation is used. Instead of branching on whether a particular pairing is in the solution or not, since the not branch would mess up the shortest-path calculation needed in the column generation, branching is done on whether or not segment i follows segment j in some pairing. With this dichotomy, the network can be reconfigured so that the shortest-path paradigm is preserved.

A survey of recent work on crew pairing is contained in Barnhart et al.[11] Other recent papers include Anbil et al.,[2] Hoffman and Padberg,[58] Barnhart et al.,[9] Vance et al.,[88] and Klabjan et al.[68] Dynamic column generation approaches to crew pairing are presented in Vance et al.[87] and Anbil et al.[1]

11. Promising Research Areas

One might be tempted to conclude that mixed-integer programming is a mature field and that further research is unlikely to yield large payoffs. On the contrary, recent successes have fostered industry's wishes to solve larger models and to get solutions in real time. We will present one example of this phenomenon from our experience, but many others exist.

The new generation of crew-scheduling algorithms discussed in the previous section have typically reduced excess pay (above pay from flying time) from around 10% to 1%. However, these results are from planning models and could only be achieved if the flight schedule was actually flown. Because of adverse weather and mechanical difficulties, some flights must be delayed or canceled and the crews must be rescheduled. Consequently, excess pay may jump back up to 5–8%.

The need to reschedule raises the issue of the robustness of the original schedule determined from the planning model. In fact, the planning model should have been formulated as a stochastic integer program that allows for consideration of many scenarios. However, a k-scenario model is at least k times the size of a deterministic, i.e., 1-scenario, model. The need to solve such a model justifies the need for algorithms that are capable of solving problems an order of magnitude larger than our current capabilities.

The crew-scheduling problem exemplifies the past, present, and future of integer programming in logistics and manufacturing. Through the 1980s, integer programming could only be used for small planning models. Presently, it is successful on larger planning models. In the future, integer programming needs to have the capability of dealing with at least two new types of models.

1. Robust models that integrate planning and operations. These will be massively large, multi-scenario stochastic models for which several hours of solution time may be acceptable.
2. Recourse models whose solutions provide the real-time operational corrections to the planning solutions. These models will have to be solved in seconds or minutes depending on the application.

These observations lead us to two promising directions of the field—stochastic integer programming and parallel integer programming.

11.1 Stochastic Integer Programming

The linear stochastic scenario programming model for a two-stage problem is

$$\begin{array}{lll} \max & c_0x + \sum_k p_k c_k y_k & \\ & A_0x & = b_0 \\ & A_kx + B_ky_k & = b_k,\ k = 1, \ldots, K \\ & x,\ y_k & \geqslant 0,\ k = 1, \ldots, K, \end{array}$$

where K is the number of scenarios and p_k is the probability of the kth scenario. This model finds a solution x to use in the current time period that is robust with respect to several outcomes for the next time period. If we knew which scenario would hold for the next time period, then we would solve the problem with only that scenario present. When there are several possible scenarios, this formulation forces a single solution x to be used and solves for what the optimum y_k would be for that x in one model whose objective is to maximize expected two-stage profit. Expositions of multi-scenario, multi-stage stochastic programming are given in the books by Birge and Louveau,[15] Infanger,[61] and Kall and Wallace.[67]

Benders' decomposition is a well-known approach to solving the two-stage linear model (often called the L-shaped solution method in the literature on stochastic linear programs). The idea is that fixing the linking variables x breaks up the problem into separate subproblems for each scenario. The subproblem solutions generate objective function cuts for the master problem that has the constraints $A_0x = b_0$ and all of the cuts from previous iterations. The algorithm cycles between solving the master problem and the subproblems. The fact that these models and algorithms do not enjoy wider use can only be attributed to the complexity of the models and the difficulty of solving them. The main difficulties with Benders' decomposition are that convergence may be slow and many possibly dense constraints are being added to the master.

Integer constraints on the variables may be needed in order for the models to be usable. Many situations have some degree of discreteness and some amount of stochasticity. Although one might assume that any model adequately capturing these two aspects of problems will be computationally intractable, large improvements in efficiency of optimization methodology and resulting software combined with cheaper, faster computers offering parallel processing capabilities should make it possible to reduce solution times of these large models to practical levels.

Recently, several researchers have considered solving the stochastic scenario problem with integrality restrictions on some or all of the x in the first phase using Benders' decomposition.[70] The complication is that the master problem is now an integer program. Several ways can be suggested for adapting the general Benders' method to this stochastic MIP model. The key algorithmic issue is how to enforce the integrality of the x variables. At one extreme, the integrality conditions are fully imposed each time the master problem is solved, i.e., Benders' cuts are only obtained after getting an optimal solution to the master problem. At the other extreme, we solve only the LP relaxation at each node of the tree and derive Benders' cuts from these possibly fractional solutions. In the first approach, we may get very good cuts, and convergence to an optimal solution is obvious, but we pay the extravagant price of having to solve an integer program at each major iteration. It has been proved that, under mild assumptions, the second approach also converges to an optimal solution. However, the cuts will be much weaker and, therefore, more of them may be needed, which could make the LP relaxations difficult to solve and may cause slow convergence. Seeing the extremes from this unified point of view suggests many intermediate alternatives that may be computationally far superior to the extremes.

Benders' decomposition lends itself well to parallellization. This brings us to the next promising direction of integer programming.

11.2 Parallel Integer Programming

Tremendous progress in the size of problems that can be solved and in the time it takes to solve them will, undoubtedly, come from parallel computing. Research in parallel computing has matured to the point where there is an understanding of its potential and limitations. There have been numerous studies in the domain of scientific computation that show that the promise of parallel computing is realizable. Whether and to what extent this promise can be fulfilled in the domain of integer programming is still a matter of speculation. However, initial experiences with parallel computing have been encouraging.

Tree search algorithms, such as the LP-based branch-and-bound algorithm that forms the core of mixed-integer optimizers, are natural candidates for parallellization. Because the subproblems associated with each node of the search tree are completely independent, parallelism can be exploited by evaluating multiple nodes of the search tree simultaneously. Due to the exponential growth of the search tree, parallel search algorithms can lead to substantial improvements. Parallelization of tree search algorithms has attracted considerable attention over the last few years. The bulk of this work has come from the computer science community and has focused on scalable parallel task scheduling and distribution strategies.

There seems to be great potential for parallel branch-and-cut and parallel branch-and-price for two reasons. First, they both have weak synchronization and consistency requirements, which provides the flexibility that is necessary to be able to exploit parallelism successfully. For example, the correctness of a branch-and-cut algorithm does not depend on whether cuts are generated or when cuts are generated. Second, the search trees in these algorithms are still too large to be manageable in a serial context.

The fundamental issue in parallelizing a branch-and-cut algorithm is the management of the set of cuts. A good distributed cut management scheme should aim to achieve the following goals simultaneously:

- Minimize total cut generation time over all processors. A careful design may allow communication to beat cut generation, thus reducing overall cut generation time.
- Maximize "useful sharing" of cuts. Since sharing of cuts involves communication, we would like to share only relevant cuts.
- Minimize latency of access to cuts. If there exists a relevant cut, the processes should be able to readily find it.
- Minimize bandwidth. We do not want to flood the communication system with cuts.

These goals are interrelated, and tradeoffs have to be exploited by varying the degrees to which each is satisfied.

One straightforward implementation for distributed cut management consists of a central processor that holds all the cuts and acts as a server to all the other processors. The advantage of such a centralized scheme is the sharing of information. This sharing results in minimal duplication of effort in cut generation and potentially in some synergism because a cut generated at one processor may be found to be effective for nodes of another processor as well—this second processor benefits from the cut, which it may never have been able to generate itself. The disadvantage of this approach is high contention and latency; the central process can become a bottleneck and communication costs can be high. Clearly, this scheme does not scale well with the number of processors.

Another approach to distributed cut management is to make the cut pools fully distributed and independent, where every process maintains its own cut pool locally and no sharing takes place. The advantages of the independent distributed sets are that the latency of access to a cut is almost constant and it is bounded by the maximum cut generation time. However, at least two undesirable phenomena occur. First, duplication of effort takes place in terms of cut generation and memory requirements for storing the cuts. Second, since the cuts that are generated are typically globally valid, irrespective of the processor at which they are generated, the advantage of sharing the global information is lost in a distributed implementation. Hybrid cut management schemes may be necessary to have a proper balance between the two extremes.

Branch-and-price poses many challenges to the parallel implementor as well. Should there be a single central pool of generated columns? Is there an advantage to sharing generated column information between processors? If so, how? If information is not shared, should the task-scheduling algorithm be adjusted to take into account that some processors may be able to solve some problems faster than others, having already generated most of the applicable columns? Because columns are typically not globally valid, it may be better to have a subtree as a unit of work instead of a node, which is typically the case in branch-and-cut algorithms.

A survey and synthesis of parallel branch-and-bound algorithms can be found in Gendron and Crainic.[42] Parallel algorithms for integer programming are presented in Bixby et al.,[19] Boehning et al.,[20] Cannon and Hoffman,[25] Eckstein,[38] Jünger and Störmer,[65] and Linderoth.[72]

12. Conclusions

We have tried to convey the message that integer programming is a vibrant subject with respect to both research and applications. The increased and widespread use of integer-programming software in the last decade has resulted from faster algorithms that are capable of solving much larger problems, reliable and easy to use software, and inexpensive hardware whose bottleneck is memory rather than speed. This success is proven, for example, in the significant number of Edelman prize papers that use integer programming models. The computational improvements have arisen from better modeling, preprocessing, primal heuristics, branch-and-cut, and branch-and-price.

Current research in integer programming is, of course, dedicated to further improvements. Users are interested in being able to solve even larger problems and with greater reliability. There is a significant need for methods to deal with stochastic and nonlinear models. The successful algorithms of the 1990s that have integrated heuristics, branch-and-bound, cutting planes, and column generation, which were considered to be independent approaches in the 1970s, suggest that even more integration is needed. For example, efforts are underway to integrate constraint-logic programming with LP-based branch-and-bound.[90] Whether the next breakthrough will come from integration, as was the case with branch-and-cut, or an entirely new development like the polynomial time algorithm for integer programming with a fixed number of variables,[71] is anyone's guess.

Acknowledgments

This research was supported by National Science Foundation Grant No. DDM-9700285. We are grateful to anonymous referees whose comments led to significant improvements in the exposition.

References

1. R. ANBIL, J.J. FORREST, and W.R. PULLEYBLANK, 1998. Column Generation and the Airline Crew Pairing Problem, *Documenta Mathematica III*, 677–686.
2. R. ANBIL, R. TANGA, and E. JOHNSON, 1991. A Global Optimization Approach to Crew Scheduling, *IBM Systems Journal 31*, 71–78.
3. D. APPLEGATE, R.E. BIXBY, V. CHVATAL, and W.J. COOK, 1997. ABCC TSP, 16th International Symposium on Mathematical Programming, Lausanne, Switserland.
4. B.C. ARNTZEN, G.B. BROWN, T.P. HARRISON, and L.L. TRAFTON, 1995. Global Supply Chain Management at Digital Equipment Corporation, *Interfaces 25*, 69–93.
5. E. BALAS, S. CERIA, and G. CORNUEJOLS, 1993. A Lift-and-Project Cutting Plane Algorithm for Mixed 0-1 Programs, *Mathematical Programming 58*, 295–324.
6. E. BALAS, S. CERIA, G. CORNUEJOLS, and N. NATRAJ, 1996. Gomory Cuts Revisited, *Operations Research Letters 19*, 1–9.
7. E. BALAS, S. CERIA, M. DAWANDE, F. MARGOT, and G. PATAKI. OCTANE: A New Heuristic for Pure 0-1 Programs, *Operations Research*, in press.
8. E. BALAS and R. MARTIN, 1980. Pivot and Complement: A Heuristic for 0-1 Programming, *Management Science 26*, 86–96.
9. C. BARNHART, E.L. JOHNSON, R. ANBIL, and L. HATAY, 1994. A Column Generation Technique for the Long-Haul Crew Assignment Problem, in *Optimization in Industry II*, T.A. Ciriani and R.C. Leachman, (eds.), Wiley, Chichester, 7–24.
10. C. BARNHART, E.L. JOHNSON, G.L. NEMHAUSER, M.W.P. SAVELSBERGH, and P.H. VANCE, 1998. Branch-and-Price: Column Generation for Solving Integer Programs, *Operations Research 46*, 316–329.
11. C. BARNHART, E.L. JOHNSON, G.L. NEMHAUSER, and P.H. VANCE, 1999. Crew Scheduling, in *Handbook of Transportation Science*, R.W. Hall, (ed.), Kluwer, Boston, 493–521.
12. E.M.L. BEALE, 1979. Branch and Bound Methods for Mathematical Programming Systems, in *Discrete Optimization II*, P.L. Hammer, E.L. Johnson, and B.H. Korte, (eds.), North-Holland, Amsterdam, 201–219.
13. M. BÉNICHOU, J.M. GAUTHIER, P. GIRODET, G. HENTGES, G.

RIBIERE, and O. VINCENT, 1971. Experiments in Mixed-Integer Linear Programming, *Mathematical Programming 1*, 76–94.
14. D. BERTSIMAS, C. DARNELL, and R. SOUCY, 1999. Portfolio Construction Through Mixed-Integer Programming at Grantham, Mayo, Van Otterloo and Company, *Interfaces 29*, 49–66.
15. J. BIRGE and F. LOUVEAUX, 1997. *Introduction to Stochastic Programming*, Springer-Verlag, New York.
16. J. BISSCHOP and R. ENTRIKEN, 1993. *AIMMS The Modeling System*, Paragon Decision Technology.
17. R.E. BIXBY, E.A. BOYD, S.S. DADMEHR, and R.R. INDOVINA, 1992. The MIPLIB Mixed Integer Programming Library, Technical Report R92-36, Rice University, Houston, Texas.
18. R.E. BIXBY, S. CERIA, C. MCZEAL, and M.W.P. SAVELSBERGH, 1998. An Updated Mixed Integer Programming Library: MIPLIB 3.0, *Optima 58*, 12–15.
19. R.E. BIXBY, W. COOK, A. COX, and E. LEE, 1995. Parallel Mixed Integer Programming, Technical Report CRPC-TR5554, Rice University, Houston, Texas.
20. R.L. BOEHNING, R.M. BUTLER, and B.E. GILLETT, 1988. A Parallel Integer Linear Programming Algorithm, *European Journal of Operations Research 34*, 393–398.
21. S. BOTHA, I. GRYFFENBERG, F.R. HOFMEYR, J.L. LAUSBERG, R.P. NICOLAY, W.J. SMIT, S. UYS, W.L. VAN DER MERWE, and G.J. WESSELS, 1997. Guns or Butter: Decision Support for Determining the Size and Shape of the South African National Defense Force, *Interfaces 27*, 7–28.
22. A.L. BREARLEY, G. MITRA, and H.P. WILLIAMS, 1975. Analysis of Mathematical Programming Problems Prior to Applying the Simplex Algorithm, *Mathematical Programming 8*, 54–83.
23. A. BROOKE, D. KENDRICK, and A. MEERAUS, 1988. *GAMS, A User's Guide*, Scientific Press, Redwood City, CA.
24. J.D. CAMM, T.E. CHORMAN, F.A. DILL, J.R. EVANS, D.J. SWEENEY, and G.W. WEGRYN, 1997. Blending OR/MS, Judgement, and GIS: Restructuring P&G's Supply Chain, *Interfaces 27*, 128–142.
25. T.L. CANNON and K.L. HOFFMAN, 1990. Large-Scale 0-1 Programming on Distributed Workstations, *Annals of Operations Research 22*, 181–217.
26. A. CAPRARA and M. FISCHETTI, 1997. Branch-and-Cut Algorithms, in *Annotated Bibliographies in Combinatorial Optimization*, M. Dell'Amico, F. Maffioli, and S. Martello, (eds.), Wiley, Chichester, 45–63.
27. V. CHVÁTAL, 1973. Edmonds Polytopes and a Hierarchy of Combinatorial Problems, *Discrete Mathematics 4*, 305–327.
28. V. CHVÁTAL, 1973. Edmonds Polytopes and Weakly Hamiltonian Graphs, *Mathematical Programming 5*, 29–40.
29. G. CORNUEJOLS and M. DAWANDE, 1998. A Class of Hard Small 0-1 Programs, in *Proceedings of the 6th International IPCO Conference*, R.E. Bixby, E.A. Boyd, and R.Z. Rios-Mercado, (eds.), Springer-Verlag, Berlin, 284–293.
30. CPLEX OPTIMIZATION, INC., 1994. *Using the CPLEX Callable Library and CPLEX Mixed Integer Library, Version 3.0*.
31. H. CROWDER, E. JOHNSON, and M. PADBERG, 1983. Solving Large-Scale Zero-One Linear Programming Problems, *Operations Research 31*, 803–834.
32. G.B. DANTZIG, D.R. FULKERSON, and S.M. JOHNSON, 1954. Solution of a Large-Scale Traveling Salesman Problem, *Operations Research 2*, 393–410.
33. G.B. DANTZIG, D.R. FULKERSON, and S.M. JOHNSON, 1959. On a Linear Programming, Combinatorial Approach to the Traveling Salesman Problem, *Operations Research 7*, 58–66.
34. DASH ASSOCIATES, 1994. *XPRESS-MP User Manual*.
35. J. DESROSIERS, Y. DUMAS, M.M. SOLOMON, and F. SOUMIS, 1995. Time Constrained Routing and Scheduling, in *Handbooks in Operations Research and Management Science, Volume 8: Network Routing*, M.O. Ball, T.L. Magnanti, C. Monma, and G.L. Nemhauser, (eds.), Elsevier, Amsterdam, 35–140.
36. B.L. DIETRICH, L.F. ESCUDERO, and F. CHANCE, 1993. Efficient Reformulation for 0-1 Programs: Methods and Computational Results, *Discrete Applied Mathematics 42*, 147–175.
37. N.J. DRIEBEEK, 1966. An Algorithm for the Solution of Mixed Integer Programming Problems, *Management Science 12*, 576–587.
38. J. ECKSTEIN, 1994. Parallel Branch-and-Bound Algorithms for General Mixed Integer Programming on the CM-5, *SIAM Journal on Optimization 4*, 794–814.
39. L.F. ESCUDERO, S. MARTELLO, and P. TOTH, 1996. A Framework for Tightening 0-1 Programs Based on Extensions of Pure 0-1 KP and SS Problems, in *Proceedings of the 4th International IPCO Conference*, E. Balas and J. Clausen, (eds.), Springer, Berlin, 110–123.
40. J.J.H. FORREST, J.P.H. HIRST, and J.A. TOMLIN, 1974. Practical Solution of Large Scale Mixed Integer Programming Problems with UMPIRE, *Management Science 20*, 736–773.
41. R. FOURER, D.M. GAY, and B.W. KERNIGHAN, 1993. *AMPL: A Modeling Language for Mathematical Programming*, Scientific Press, Redwood City, CA.
42. B. GENDRON and T. CRAINIC, 1994. Parallel Branch-and-Bound Algorithms: Survey and Synthesis, *Operations Research 42*, 1042–1066.
43. F. GLOVER, 1996, personal communication.
44. M.X. GOEMANS, 1997. Improved Approximation Algorithms for Scheduling with Release Dates, *Proceedings of the 8th ACM-SIAM Symposium on Discrete Algorithms*, 591–598.
45. R. GOMORY, 1960. An Algorithm for the Mixed Integer Problem, Technical Report RM-2597, The Rand Corporation, Santa Monica, CA.
46. R.E. GOMORY, 1958. Outline of an Algorithm for Integer Solutions to Linear Programs, *Bulletin of the American Mathematical Society 64*, 275–278.
47. M. GRÖTSCHEL, M. JÜNGER, and G. REINELT, 1984. A Cutting Plane Algorithm for the Linear Ordering Problem, *Operations Research 32*, 1195–1220.
48. M. GRÖTSCHEL, C.L. MONMA, and M. STOER, 1992. Computational Results with a Cutting Plane Algorithm for Designing Communication Networks with Low-Connectivity Constraints, *Operations Research 40*, 309–330.
49. M. GRÖTSCHEL, C.L. MONMA, and M. STOER, 1995. Design of Survivable Networks, in *Handbooks in Operations Research and Management Science, Volume 7: Network Models*, M.O. Ball, T.L. Magnanti, C. Monma, and G.L. Nemhauser, (eds.), Elsevier, Amsterdam, 617–672.
50. M. GRÖTSCHEL and M.W. PADBERG, 1979. On the Symmetric Traveling Salesman Problem II: Lifting Theorems and Facets, *Mathematical Programming 16*, 281–302.
51. M. GRÖTSCHEL and W.R. PULLEYBLANK, 1986. Clique Tree Inequalities and the Symmetric Traveling Salesman Problem, *Mathematics of Operations Research 11*, 537–569.
52. Z. GU, G.L. NEMHAUSER, and M.W.P. SAVELSBERGH, 1998. Cover Inequalities for 0-1 Integer Programs: Computation, *INFORMS Journal on Computing 10*, 427–437.
53. K. GUE, G.L. NEMHAUSER, and M. PADRON, 1997. Production Scheduling in Almost Continuous Time, *IIE Transactions 29*, 341–358.
54. M. GUIGNARD and K. SPIELBERG, 1981. Logical Reduction Methods in Zero-One Programming, *Operations Research 29*, 49–74.
55. L.A. HALL, A.S. SCHULZ, D.B. SHMOYS, and J. WEIN, 1997. Scheduling to Minimize Average Completion Time: Off-Line and

On-Line Approximation Algorithms, *Mathematics of Operations Research 22*, 513–544.
56. C. HANE, C. BARNHART, E.L. JOHNSON, R. MARSTEN, G.L. NEMHAUSER, and G.C. SIGISMONDI, 1995. The Fleet Assignment Problem: Solving a Large Integer Program, *Mathematical Programming 70*, 211–232.
57. K. HOFFMAN and M. PADBERG, 1991. Improving LP-Representations of Zero-One Linear Programs for Branch-and-Cut, *ORSA Journal of Computing 3*, 121–134.
58. K. HOFFMAN and M. PADBERG, 1993. Solving Airline Crew-Scheduling Problems by Branch-and-Cut, *Management Science 39*, 667–682.
59. J. HUETER and W. SWART, 1998. An Integrated Labor-Management System for Taco Bell, *Interfaces 28*, 75–91.
60. IBM CORPORATION, 1990. *Optimization Subroutine Library, Guide and Reference*.
61. G. INFANGER, 1994. *Planning Under Uncertainty*, Scientific Press, Redwood City, CA.
62. E.L. JOHNSON, 1989. Modeling and Strong Linear Programs for Mixed Integer Programming, in *Algorithms and Model Formulations in Mathematical Programming, NATO ASI Series 51*, S.W. Wallace (ed.), 1–41.
63. E.L. JOHNSON and M. PADBERG, 1982. Degree-Two Inequalities, Clique Facets and Biperfect Graphs, *Annals of Discrete Mathematics 16*, 169–187.
64. M. JÜNGER, G. REINELT, and G. RINALDI, 1995. The Traveling Salesman Problem, in *Handbooks in Operations Research and Management Science, Volume 7: Network Models*, M.O. Ball, T.L. Magnanti, C. Monma, and G.L. Nemhauser, (eds.), Elsevier, Amsterdam, 225–330.
65. M. JÜNGER and P. STÖRMER, 1995. Solving Large-Scale Traveling Salesman Problems with Parallel Branch-and-Cut, Technical Report 95.191, Universität zu Köln, Cologne, Germany.
66. M. JÜNGER and S. THIENEL, 1998. Introduction to ABACUS: A Branch-And-Cut System, *Operations Research Letters 22*, 83–95.
67. P. KALL and S.W. WALLACE, 1994. *Stochastic programming*, Wiley, New York.
68. D. KLABJAN, E.L. JOHNSON, and G.L. NEMHAUSER, 1999. Solving Large Airline Crew Scheduling Problems, Technical Report TLI-99-11, Georgia Institute of Technology, Atlanta, GA.
69. H.A. LAND and A.G. DOIG, 1960. An Automatic Method for Solving Discrete Programming Problems, *Econometrica 28*, 497–520.
70. G. LAPORTE and F.V. LOUVEAUX, 1993. The Integer L-Shaped Method for Stochastic Integer Programs, *Operations Research Letters 13*, 133–142.
71. H.W. LENSTRA, JR., 1983. Integer Programming with a Fixed Number of Variables, *Mathematics of Operations Research 8*, 538–547.
72. J. LINDEROTH, 1998. *Topics in Parallel Integer Programming*, Ph.D. Thesis, Georgia Institute of Technology, Atlanta, GA.
73. J. LINDEROTH and M.W.P. SAVELSBERGH, 1999. Search Strategies for Mixed Integer Programming, *INFORMS Journal on Computing 11*, 173–187.
74. H. MARCHAND and L. A. WOLSEY, 1999. The 0–1 Knapsack Problem with a Single Continuous Variable, *Mathematical Programming 85*, 15–33.
75. O. MARTIN, S.W. OTTO, and E.W. FELTEN, 1992. Large Step Markov Chains for the TSP Incorporating Local Search Heuristics, *Operations Research Letters 11*, 219–224.
76. A. MEHROTRA, E.L. JOHNSON, and G.L. NEMHAUSER, 1998. An Optimization Based Heuristic for Political Districting, *Management Science 44*, 1100–1114.
77. C.L. MONMA and D.F. SHALLCROSS, 1989. Methods for Designing Communication Networks with Certain Two-Connected Survivability Constraints, *Operations Research 37*, 531–541.
78. G.L. NEMHAUSER, M.W.P. SAVELSBERGH, and G.C. SIGISMONDI, 1993. MINTO: A Mixed INTeger Optimizer, *Operations Research Letters 15*, 47–58.
79. G.L. NEMHAUSER and L.A. WOLSEY, 1988. *Integer and Combinatorial Optimization*, Wiley, New York.
80. M.W. PADBERG and G. RINALDI, 1987. Optimization of a 532 City Symmetric Traveling Salesman Problem by Branch and Cut, *Operations Research Letters 6*, 1–7.
81. M.W. PADBERG, T.J.V. ROY, and L.A. WOLSEY, 1985. Valid Linear Inequalities for Fixed Charge Problems, *Operations Research 33*, 842–861.
82. M.W.P. SAVELSBERGH, 1994. Preprocessing and Probing Techniques for Mixed Integer Programming Problems, *ORSA Journal on Computing 6*, 445–454.
83. M.W.P. SAVELSBERGH, R.M. UMA, and J. WEIN, 1998. An Experimental Study of LP-Based Approximation Algorithms for Scheduling Problems, *Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms*, 453–462.
84. A. SCHRIJVER, 1986. *Theory of Linear and Integer Programming*, Wiley, Chichester.
85. G.P. SINHA, B.S. CHANDRASEKARAN, N. MITTER, G. DUTTA, S.B. SINGH, A.R. CHOUDHURY, and P.N. ROY, 1995. Strategic and Operational Management with Optimization at Tata Steel, *Interfaces 25*, 6–19.
86. J.A. TOMLIN, 1971. An Improved Branch-and-Bound Method for Integer Programming, *Operations Research 19*, 1070–1075.
87. P.H. VANCE, A. ATAMTURK, C. BARNHART, E. GELMAN, E.L. JOHNSON, A. KRISHNA, D. MAHIDHARA, G.L. NEMHAUSER, and R. REBELLO, 1997. A Heuristic Branch-and-Price Approach for the Airline Crew Pairing Problem, Technical Report LEC-97-06, Georgia Institute of Technology, Atlanta, GA.
88. P.H. VANCE, C. BARNHART, E.L. JOHNSON, and G.L. NEMHAUSER, 1997. Airline Crew Scheduling: A New Formulation and Decomposition Algorithm, *Operations Research 45*, 188–200.
89. H.P. WILLIAMS, 1978. *Model Building in Mathematical Programming*, Wiley, Chichester.
90. H.P. WILLIAMS and J.M. WILSON, 1998. Connections Between Integer Linear Programming and Constraint Logic Programming: An Overview and Introduction to the Cluster of Articles, *INFORMS Journal on Computing 10*, 261–264.
91. L.A. WOLSEY, 1998. *Integer Programming*, Wiley, New York.