Recent developments and future directions in mathematical programming

by E. L. Johnson
G. L. Nemhauser

Recent advances in mathematical programming methodology have included: development of interior methods competing with the simplex method, improved simplex codes, vastly improved performance for mixed-integer programming using strong linear programming formulations, and a renewed interest in decomposition. In addition, use of vector and parallel processing has improved performance and influenced algorithmic developments. Application areas have been expanding from the traditional refinery planning and distribution models to include finance, scheduling, manufacturing, manpower planning, and many others. We see the acceleration of better methods and improved codes moving together with faster, lower-cost, and more interesting hardware into a variety of application areas, thereby opening up new demands for greater function of optimization codes. These new functions might include, for example, more powerful nonlinear codes, decomposition techniques taking advantage of network and other problem-dependent structures, and mixed-integer capability in quadratic and general nonlinear problems. Stochastic scenario programming and multitime-period problems are becoming solvable and open up applications and algorithmic challenges. The IBM Optimization Subroutine Library has helped to accelerate these changes but will have to continue to change and expand in ways that are touched upon in this paper.

Linear programming grew out of work begun during the 1930s: in transportation problems by Kantorovich, game theory by Morgenstern and Von Neumann, and input-output models by Leontief. In the late 1940s, Dantzig defined the general model and proposed the simplex method for its solution. In the 1950s codes were developed at several places, including the Rand Corporation and the U.S. National Bureau of Standards. In the 1960s several commercial codes were developed, mainly for oil companies, and in some cases were even written by oil companies. Linear programming was developed and has grown in parallel with computers. The business of developing and selling linear programming codes has been a commercial success for 40 years and has driven hardware sales, as well. Application areas have become more diverse and today include transportation, distribution, manufacturing, scheduling, finance, and many others. Its importance can be seen from the fact that a Nobel Prize in economics was given to L. V. Kantorovich and T. C. Koopmans for work in linear programming.

Mathematical programming refers to that part of mathematical optimization concerned with optimizing some objective function subject to constraints (maximizing profit or minimizing cost, for

example). The word "program" or "programming" (as in the term linear program), came from the use of the word in the 1940s as a synonym for planning or scheduling and does not refer to computer programming—although mathematical programming growth has paralleled growth in the use of computers. A mathematical programming problem is a *linear program* (LP) when the objective function is linear and the only constraints are linear equations and inequalities. A standard form for linear programs is linear equations in nonnegative variables, i.e., $Ax = b$, $x \geq 0$. *Quadratic programming* usually means a linearly constrained problem with a convex, quadratic objective, whereas *nonlinear programming* is used to cover any problem having linear or nonlinear constraints or objective function. *Integer programs* are typically linear programs with the additional constraint that some or all of the variables of the problem must take on integer values. *Mixed-integer programs* (MIPs) have some integer variables and some variables not constrained to be integer, sometimes called *continuous variables*. *Mixed 0-1 program* refers to a mixed-integer program in which the integer variables are additionally constrained to only take on values of either zero or one. "Pure" in place of "mixed" denotes problems in which there are only integer variables. The most important class of integer programming problems is the mixed 0-1 problem, and the 0-1 variables typically represent choice variables; i.e., some activity is either done or not done.

The easiest of these problems to solve are linear programs, and the simplex method is the classical method. Recently, interior point algorithms have begun to compete with the simplex method. In practice, problems with tens of thousands of equations in hundreds of thousands of variables can be solved in reasonable times, say one or two hours on a mainframe or powerful workstation. Quadratic programs, of the convex type for minimization, are also fairly easy to solve. Mixed-integer programs can take much longer, and running times are much more variable depending on problem type, formulations, and efficiency of codes. In practice, branch-and-bound is the method employed.

The nonlinear area is also difficult in general, and one where specialized codes have been developed for certain classes of problems. Commercial codes have tended to avoid nonlinear programming because of the lack of robust, general-purpose methodology. In practice, linear approximations are frequently employed, sometimes in a sequential manner in order to at least obtain a local optimum.

In the next section, we summarize several application areas that have motivated the development of mathematical programming methodology and computer codes. In turn, recent advances in methodology and improvements in codes have opened up new possibilities for solving some of these problems. We discuss the types of mathematical programs involved in each application area.

In succeeding sections we first present a very brief summary of recent advances in linear programming. We then present the methodology behind the strong linear programming formulations of mixed-integer programs. These recent advances in MIP methodology have yielded very impressive computational results. Finally, we discuss recent developments in decomposition and column generation that are important for solving large-scale LPs and MIPs.

We have chosen to emphasize mixed-integer programming and column generation for three reasons. The first is that in this issue, linear programming is covered in great detail by Forrest and Tomlin,[1,2] and quadratic programming is presented by Jensen and King.[3] The second is that MIP and column generation are our areas of specialization and current interest. Finally, we believe that some of the most significant current and future applications lie in this area.

Application areas

The oldest extensively used large-scale application is found in the petroleum industry for blending crude oils in refinery operations to produce a desired mix of final products. This application is widespread and important enough that virtually every oil company has some linear programming model in use. The problems typically have some nonlinear component, and some type of linear approximations are used. One of the techniques employed is sequentially solving linear programs, and another is piecewise linear functions. Also, distribution problems are sometimes incorporated into the model or are solved separately.

In general, distribution problems are widespread. In some examples, production planning is incorporated into the model. The production-distribution model at General Motors[4] includes a mixed-integer model for planning changeovers in production together with an aggregated distribution model for shipping the cars produced. The 0-1 variables in the mixed-integer model are choice variables having to do with changeovers. The problem, generally speaking, is that demand patterns change across the country, and production must change to meet demand. The optimization problem determines how to meet demand given current production capacity, costs of changeovers, and shipping costs.

The finance area is one where there are great advantages to using optimization software.

Another widely used model is multiproduct planning with single-sourcing restrictions at warehouses or distribution centers for shipments to customers. In this model, 0-1 decision variables are used to decide which warehouse will supply a given customer. The example of using this model at Frito-Lay[5] was an early indication of the computational advantage to be obtained by reformulation to give a stronger linear programming relaxation.

Distribution problems without multiproduct single-sourcing and without any additional constraints result in linear programs that are network flow problems. A network flow problem is a linear program in which every variable is a shipping variable from the node of origin to a destination node. The important fact about network flow problems is that they can be solved by specialized simplex codes much faster than by using a general linear programming code. In network flow problems with additional constraints, decomposition methods can take advantage of the network flow structure by solving subproblems more quickly as network flow problems.

The finance area is one where there are great advantages to using optimization software, and as faster, easier-to-use software becomes available, the usage could increase dramatically. The classical work in this area is the Markowitz model for portfolio selection. Markowitz[6] developed a model for balancing risk and return in selecting investments. The model leads to a convex quadratic problem in which the quadratic part is the matrix of covariances of the returns from the various investments. If risk is ignored, it is best to put all of the available money in the investment with highest expected return, provided that the only constraint is a budget constraint. What the Markowitz model tries to do as an alternative is to keep expected return high, but diversify investments by looking for the ones that move counter to each other or at least independently of each other. The resulting diversified portfolio is less likely to have major deviations from the expected return. Recently, there has been activity to extend the model to multitime periods and multiscenarios[7] in order to take into account dynamic aspects of the problem and make the solutions more robust in responding to various possible market movements.

Another finance model uses liability matching for investing money in fixed-income securities such as high-quality bonds to generate a revenue stream over time that meets scheduled liabilities. This model can be used to invest money generated from a bond issue in order to meet scheduled construction costs, while generating as much revenue as is safely possible from the extra money until it is needed. This model leads to a linear, rather than quadratic, program since the safety factors are generally included as constraints.

Optimization models have been in use in energy planning for many years; examples include models involving crude oil allocation, power generation planning, and coal allocation. Most of the problems are linear programs with integer restrictions in some models. Large problems may be needed, and solving them can benefit from vastly reduced running times using new software and hardware.

Recent work in the airline industry has received much attention. Crew planning[8,9] is an active area. Until recently, all codes used a suboptimization approach, and only recently and only in some codes was any linear or integer program-

ming utilized, and then only for subproblems. The power to be gained from a global approach has only been possible to realize as a result of better linear programming codes to solve the difficult set-partitioning linear programs.

Another airline application of mathematical programming is fleet assignment.[10] The word "fleet" denotes a type of plane, such as a Boeing 747, and the fleet assignment problem is to assign the fleets to flight legs in such a way that profit is maximized and the planes can be efficiently scheduled and maintained. This problem must be solved before crew planning but after scheduling flight legs or routes. Once the fleets have been assigned, the individual planes must be routed and maintenance scheduled. Then the crew pairings can be formed as part of the crew-pairing optimization.

This brief summary of major application areas illustrates the diversity and importance of mathematical programming models. We now turn to computational advances making such models more tractable.

Linear programming

Until recently, the algorithms used in commercial mathematical programming systems had hardly changed from the original systems developed in the late 1950s and in the 1960s. The two basic algorithms—the simplex method for linear programs and branch-and-bound for mixed-integer and piecewise linear programs, solved by the simplex method and partial enumeration—remained the mainstays. Although we still rely on the simplex method and branch-and-bound, in the last five years major advances have taken place in the computational aspects of the simplex method, and substantial progress has been made in avoiding branching rather than just coping with it efficiently. In addition, interior point methods now offer serious competition to simplex methods.

Simplex methods. A paper by Forrest and Tomlin in this issue[1] presents recent computational advances, and one by Goldfarb and Todd[11] found elsewhere provides an in-depth survey. We will only discuss theoretical results on the running time of the simplex method. In the worst case, it can perform quite poorly. Klee and Minty[12] present a family of problems whose feasible region is a distorted hypercube and for which choosing the variable with greatest reduced profit to enter the basis causes the simplex algorithm to visit all of the vertices of the transformed hypercube. These negative results have been extended to other variants of the simplex algorithm and to some special cases of linear programs, including network flow problems. However, it is not known whether there exists some variant of the simplex algorithm that runs in polynomial time for general linear programs, but such polynomial-time variants for network flow problems have been obtained.[13]

On the positive side, the expected behavior of the simplex algorithm is quite good. Under various probabilistic assumptions, it has been shown that the average number of iterations is bounded by a low-degree polynomial in m and n.[14]

Ellipsoid algorithms. Khachian[15] showed that an algorithm developed for nonlinear programming could be adapted to linear programming with a polynomial time bound on its running time. This result was remarkable since the existence of a polynomial time algorithm for linear programming was then considered to be one of the most important unsolved problems in computational complexity. Its announcement stimulated a tidal wave of research and papers, which ended disappointingly with the conclusion that the ellipsoid method was a total failure in practice (see Bland et al.[16] for a survey). The poor performance resulted because the number of iterations is bounded by a polynomial that involves the logarithm of numerical coefficients, and the actual number of iterations frequently is close to the bound. Nevertheless, the ellipsoid algorithm has proved to be a very important tool for proving theorems about polynomiality in combinatorial optimization,[17] since it is capable of dealing with an exponential number of structured constraints.

The essential idea of the ellipsoid algorithm is very simple and most easy to describe geometrically for a problem of determining whether a set of linear inequalities $Ax \leq b$ has a feasible solution, i.e., given $P = \{x \in R^n: Ax \leq b\}$, determine whether P is nonempty. We assume for simplicity that if P is nonempty, then

1. P is bounded and therefore is contained in a hypersphere S_R of radius R centered at x^0, and R and x^0 are given.

2. P is full-dimensional and therefore contains a hypersphere S_r of radius r centered at x^0, and r is given.

Now let $E^0 = S_R$. To find a point in P or to show that none exists, the algorithm produces a sequence of ellipsoids $\{E^k\}$ of shrinking volume such that each ellipsoid contains P. At iteration k, we see if x^k, the center of E^k, is contained in P. This can be done simply by checking all of the inequalities by substitution. Now if all of the inequalities are satisfied, the problem is solved since $x^k \in P$. If not, a violated inequality $a^{i_k}x \leq b_{i_k}$ is identified. Then P is contained in the half-ellipsoid obtained by intersecting E with the inequality $a^{i_k}x \leq a^{i_k}x^k$. The ellipsoid E^{k+1}, which can be approximated with sufficient accuracy in polynomial time, contains the half-ellipsoid and thus also contains P. The important result of this construction is that

$$\text{vol}\,(E^{k+1})/\text{vol}\,(E^k) \leq e^{-\frac{1}{2(n+1)}}$$

This shrinkage rate is sufficiently big to guarantee that after a number of iterations proportional to n and log (vol (S_R)), the volume of the resulting ellipsoid is less than the volume of S_r, in which case the problem must be infeasible.

The ellipsoid algorithm described above can be extended to handle optimization by adding objective function cuts. In particular, once a feasible solution x^0 has been found, we add the constraint $cx > cx^0$ and continue. Now the procedure will terminate with an indication of infeasibility after an optimal solution has been found. But this implies that the worst-case running time is almost always achieved, which explains the poor performance in practice.

Finally, it is important to realize that the ellipsoid algorithm only uses the constraints to test the feasibility of the centers of some ellipsoids. Therefore, if the problem of determining whether the point is feasible can be decided by some oracle, an explicit representation of the constraints may not be necessary. This is very important in combinatorial optimization where we consider linear programs with an exponential number of constraints and make use of the theorem (which is a consequence of the ellipsoid algorithm) that says: for a family of polyhedra, the *separation* problem of testing the feasibility of a point and finding a violated inequality if the point is not feasible can be solved in polynomial time if and only if for any objective function the linear programming optimization can be solved in polynomial time.

Interior point methods. The computational disappointment with ellipsoid algorithms was not to be repeated with the next fundamental development in linear programming—the emergence of interior point methods.[18] In contrast to the ellipsoid algorithm, interior point methods have already demonstrated their computational power as an alternative to the simplex method in solving large linear programs. It is an extremely active area of research, including algorithm development and implementation. The developments from 1984 to 1989 and their antecedents are carefully traced in the Goldfarb and Todd survey.[11] Marsten et al.[19] provide a status report on computational developments and results. Lustig et al.[20] give computational results for the code OB1, which at this time is generally considered to be the best implementation of an interior point method, and compare OB1 to a widely used simplex code. Forrest and Tomlin[2] present a survey that emphasizes primal-dual interior point methods and their IBM Optimization Subroutine Library (OSL) implementation.

As suggested by the name, the basic interior point method begins with and maintains feasible solutions throughout in which all constraints are satisfied with strict inequality. It requires a special end game strategy to find an optimal vertex if such a solution is required. The idea of moving in a good direction through the interior of the feasible set is old. The problem with such an approach is what to do when a boundary is encountered.

Given a feasible point x in the interior of P, the basic idea of Karmarkar's algorithm is a *projective transformation* that maps the feasible region into itself and x into the center of the transformed region. The advantage of the transformation is that x is now far from the boundaries of all of the constraints, and therefore a nontrivial step in a good direction is possible. Hence, rather than trying to determine the active constraints in an optimal solution as in a simplex method, an interior method avoids altogether the problem of determining the active constraints. As a simpler alternative to projective transformations, Barnes[21] proposed affine transformations which gave rise to the primal and dual affine algorithms.

To prove polynomiality, Karmarkar introduced a *potential function* to measure progress toward the solution. This function, which imposes a logarithmic penalty on violating the nonnegativity constraints, closely resembles the logarithmic barrier function for turning a nonlinear program with linear inequality constraints into a sequence of unconstrained optimization problems. Gill et al.[22] studied the relationship between classic barrier algorithms, such as the SUMT algorithm of Fiacco and McCormick,[23] and Karmarkar's algorithm. In particular they show that Karmarkar's algorithm takes a step in the same direction as a barrier method, with steps in a projected Newton direction for some value of the barrier parameter. This connection led to the path-following algorithms over a parameterized family of logarithmic barrier functions[24] and to what are now called the primal-dual interior point methods, which have produced the best computational results.[2,20]

At this writing, there is no definitive conclusion regarding the superiority of interior or simplex methods, and it may very well be the case that the methods are incomparable. On one hand, a real virtue of the interior methods is the constant number, or very slow growth, of iterations as problems grow larger. Therefore, these methods should be better for very large problems. On the other hand, they appear to have an even greater need for sparsity than the simplex method and do not naturally produce basic optimal solutions. However, the biggest current disadvantage of interior point methods is not being able to take advantage of good starting solutions, which is especially disappointing for mixed-integer programming where one solves a sequence of linear programs with slightly modified constraint sets.

Mixed-integer programming

We consider the linear mixed-integer program (MIP)

$$\max \quad \{cx + hy : Ax + Gy \leq b, \quad x \geq 0 \text{ and integer}, y \geq 0\}$$

Here some or all of the variables are required to be integer. In many models, the integer variables are used to represent logical relationships and therefore are constrained to equal zero or one. The MIP model is very robust in the sense that an unusually wide variety of practical problems, including those with nonlinearities and nonconvexities, can be represented by MIPs. But until recently, one could not count on solving MIPs with a few hundred integer variables to optimality. That situation is rapidly changing, and the capability of solving MIPs with thousands of variables has arrived. We present the basic ideas of these advances. For more information see Hoffman and Padberg[25] and Nemhauser and Wolsey.[26,27]

The linear programming relaxation of MIP, denoted by LPR, is the linear program obtained from MIP by omitting all of the integrality requirements. The following elementary connections between LPR and MIP are used in the algorithms given below for solving MIP.

1. If LPR is feasible, its optimal value z (LPR) $\geq z$ (MIP), the optimal value of MIP.
2. If LPR is infeasible, so is MIP.
3. If an optimal solution to LPR is also a feasible solution to MIP (satisfies the integrality constraints), that solution is also an optimal solution to MIP.

Branch-and-bound. Branch-and-bound with linear programming relaxations is the classical approach to solving MIP and forms the basis for all of the general-purpose codes. See Beale[28] and Land and Powell[29] for a survey of the algorithms and software from the 1970s, which have begun to be replaced only recently.

The basic idea is very simple. We first solve LPR. If it is infeasible or has an optimal integral solution, we have also solved MIP. Otherwise, some variable in the optimal solution to LPR is fractional. We pick such a variable and branch by creating two MIP subproblems: In one subproblem the variable is constrained to be equal to or less than its round down (equal to zero in the case of a binary variable); in the second problem the variable is constrained to be equal to or greater than its round up (equal to one in the case of a binary variable). Now the optimal solution to MIP is obtained by solving the two subproblems and taking the better of the two solutions. This process is executed recursively so that there is a danger of having to solve a number of MIP subproblems that grows exponentially with the number of integer variables. But the degree of enumeration is controlled by two factors:

1. If LPR (for a subproblem) is infeasible or has an optimal integer solution, MIP is solved for the subproblem, and no branching is necessary.
2. If LPR (for a subproblem) has an optimal integer solution, that solution is feasible to the original MIP, and therefore gives a lower bound on the optimal value of MIP. We keep the best lower bound $\underline{z}$ on the optimal value and update it whenever an improved feasible solution is found. Then if for some subproblem z (LPR) $\leq \underline{z}$, that subproblem cannot have a better solution to MIP than what is already known, and no branching is necessary. This is known as *pruning by bounds* and is responsible for keeping the number of subproblems manageable where possible.

The basic branch-and-bound algorithm needs to be elaborated by specifying details such as which subproblem should be processed next and which fractional variable should be the one on which to make the decision to branch. One way to view the subproblem list is as a tree in which each subproblem corresponds to a node and in which the immediate descendants of a node (its children) correspond to the subproblems derived from it. In this setting, a natural way to select subproblems is by *depth first search* plus *backtracking*. This means that if a node has children, the next node selected is one of them. Otherwise we backtrack by following the unique path from the current node back to the root node (original problem) until we find a node (if any) with an unprocessed child.

The advantages of depth first search are:

1. Solving LPR for a child, given the optimal solution for a parent, just involves changing a single bound, and therefore, reoptimization by the dual simplex method should be very fast.
2. It helps in finding feasible solutions since nodes deep in the tree have more variables constrained to be integral.

The second advantage disappears once a good feasible solution has been obtained. Then other strategies such as choosing the node with the best bound become more attractive.

Variable selection is based on user priorities reflecting the importance of the integer variables and logical relations among them, on the likelihood of getting an integer solution, and on estimates of the decrease in the optimal value by forcing the variable to round up or round down its current value.

Finally, we note that branching on simple constraints such as the clique or SOS constraint

$$\sum_{j \in Q} x_j = 1$$

can be advantageous in producing a more equitable division of the solution space. This is done by picking a subset Q of the variables in the clique constraint and then requiring $\Sigma_{j \in Q} x_j = 0$ along one branch and $\Sigma_{j \notin Q} x_j = 0$ along the other. Note that if the subset is of size 1, this branching rule is exactly the rule given above for branching on 0-1 variables.

The success of branch-and-bound in solving either large or difficult problems or both types depends heavily on how closely LPR approximates MIP and on finding a good feasible solution quickly. If the linear program approximation is poor, a very large number of subproblems will have to be solved regardless of the strategies that are used for node and variable selection. However, if the approximation is perfect in the sense of giving an optimal solution in which all of the integrality constraints are satisfied, it will suffice to solve only one linear program.

The developments in the 1980s have focused largely on improving linear programming approximations with some work done on the feasibility problem, see, e.g., Balas and Martin.[30] These approaches are discussed below under modeling, preprocessing, cut generation, and column generation. Modeling refers to the initial formulation; preprocessing refers to actions taken before solving an LP; cut and column generation refers to improving the LP relaxation by adding additional rows or columns. However, this classification is somewhat arbitrary since some improvements left out of the initial model may be picked up in preprocessing and some improvement of the LP not done in preprocessing may be done in the cutting phase of the algorithm.

Modeling principles: Good formulations. Many correct formulations generally exist for a MIP problem. In general, we are concerned much less

with the size of the formulation than with the quality of its LP approximation.

For example, there is a classical lot-sizing model which has a formulation with variables that give the amount produced in each period, and another in which there are variables that give the production in period t designated for sale in period $t + k$. The number of variables in the second formulation is the square of the number in the first. But the second formulation yields a network flow problem, which implies that its LP relaxation has an optimal integral solution, and for this reason is the better formulation.

For the traveling salesman problem, there is a formulation with a number of constraints proportional to the number of edges and a second one with a number of constraints that grows exponentially with the number of edges. The second one, however, is far superior because its linear programming relaxation is provably better. Of course, we cannot directly solve a linear program whose number of constraints is exponential in the size of the problem. Instead, as we discuss under cut generation, the constraints are generated sequentially as we need them to achieve feasibility.

Similarly, for a partitioning problem, there is a formulation with a 0-1 variable for each element-subset pair that specifies whether the element belongs to the subset. But this formulation is inferior with respect to its linear programming relaxation to another one in which there is an exponential number of 0-1 variables, one for each feasible subset of elements.

Recently some general concepts have evolved for obtaining good formulations.

Disaggregation is the idea of separating one constraint into many without changing the meaning of the MIP model but improving the LP relaxation. Suppose we have m 0-1 variables x_j, $j = 1 \cdots$, m, each of which must be zero unless another 0-1 variable $x_0 = 1$. This relationship can be modeled with only one constraint, namely

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

Alternatively, we can use m constraints

$$x_j \leq x_0 \text{ for } j = 1 \cdots, m$$

For 0-1 variables both sets of constraints say the same thing. But when x_0 is fractional, the second set of constraints imposes much more on the x_js and therefore gives the better formulation. Note that although disaggregation creates additional constraints, it is not likely to make the LP much harder to solve since the number of nonzeros in the constraint matrix only increases slightly.

Now suppose our model contains the variable upper-bound constraint

$$y \leq M x$$

where x is a 0-1 variable that appears only in this constraint and in the objective function with a negative coefficient $-f$, and y is a continuous variable with an upper bound of M. The constraint says $x = 0$ implies $y = 0$, and if $y > 0$, we incur the fixed cost f. Now in the LP relaxation, if $0 < y < M$, then $0 < x < 1$. Therefore, a good modeling principle is to *make the upper bound M as small as possible*.

Preprocessing. (See Crowder et al.[31] and Hoffman and Padberg.[32]) Given a model, we wish to improve it by tightening the LP relaxation, fixing variables, or, more generally, doing whatever we can do quickly to make the problem easier to solve. The line between good modeling and preprocessing is fuzzy. A sophisticated preprocessor should be able to recognize constraints that need to be disaggregated and to tighten variable upper bounds. But we cannot expect the software to produce a formulation that is radically different from the one it is given. However, there are certain operations for which the computer is better suited.

Consider a single inequality of the form

$$\sum_{j=1}^{n} a_j x_j \leq b$$

in which all of the variables are binary. Without loss of generality, for this constraint in isolation, we can assume that a_js are positive since if $a_j < 0$, we complement the corresponding variable by replacing x_j by $1 - x_j$. Inequalities of this type are called *knapsack* constraints, and a good deal of information can be extracted from them rather painlessly.

The linear programming relaxation of a knapsack inequality can be tightened by *coefficient reduction*. Observe that

$$(a_1 - \lambda)x_1 + \sum_{j=2}^{n} a_j x_j \leq b - \lambda$$

gets progressively tighter as λ increases from 0. Therefore, we would like to make λ as large as possible without eliminating any 0-1 solutions. Determining the largest such λ is a hard problem, but it is easy to find a feasible value of λ, e.g.,

$$\lambda = \max\left(0, b - \sum_{j=2}^{n} a_j\right)$$

The knapsack constraints yield logical relations among the variables. Suppose

$$a_j + a_k > b \tag{1}$$

then

$$x_j + x_k \leq 1 \tag{2}$$

Note that after reversing the complementation, we can obtain four types of constraints, namely

$$x_j + x_k \leq 1,\ x_j \leq x_k,\ x_j \geq x_k, \text{ and } x_j + x_k \geq 1$$

The constraints (2) define a node-packing problem on a graph in which there is a node for each variable and its complement, edges joining complementary nodes and pairs of nodes for which (1) is satisfied. A *node packing* is a subset of nodes such that no pair in the set is joined by an edge. Now let C be a subset of nodes of size at least three such that each pair in the set is joined by an edge. C is called a *clique*. The edge constraints imply that no more than one node can be selected from a clique. After decomplementing the variables we obtain the *generalized clique constraint*

$$\sum_{j \in C^-} x_j - \sum_{j \in C^+} x_j \geq |C^-| - 1$$

where $C^- = \{j \in C: x_j \text{ complemented}\}$ and $C^+ = C/C^-$.

It is important to realize that the edges of the graph can come from different knapsack constraints of the original MIP. Thus, in building clique constraints from the edges of the graph, we capture logical relations that depend on the interactions among the original constraints. For example, one constraint may yield $x_1 + x_2 \leq 1$, another $x_2 + x_3 \leq 1$, and a third $x_1 + x_3 \leq 1$. Then together they yield $x_1 + x_2 + x_3 \leq 1$. Clique constraints can be added to the original formulation or used as cutting planes as explained below.

Cut generation. We now suppose that our original formulation of MIP contains too many constraints to include all of them in the LP relaxation and that in solving LPR some constraints have been ignored. The constraints can be of two types. Type I constraints are needed for the formulation to be correct. If a type I constraint is violated, LPR can give an integral optimal solution that is infeasible. An example of type I constraints is the class of *subtour elimination constraints* (SEC) for the traveling salesman problem (TSP) of finding a minimum cost cycle that contains all the nodes of a graph. They state the feasibility requirement that for every proper subset S of the nodes, there must be at least two edges with one end in S and the other not in S. However, since the number of these constraints is exponential in the size of the graph, it is not possible to include all of them when solving an LP relaxation.

Type II constraints are used only for the purpose of tightening the LP relaxation. An example of these are the cover constraints that are derived from knapsack inequalities. These constraints generalize the edge inequalities given above. 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 cover inequality

$$\sum_{j \in C} x_j \leq |C| - 1$$

which must be satisfied by all feasible solutions.

Now we address the question: Given a solution x' to LPR, does it satisfy all of the ignored con-

straints? This question is called the *separation* problem. We assume that there are too many constraints to check by substitution. Note that if all of the ignored constraints are type II and the LP solution satisfies the integrality requirements, the answer is obviously yes.

To illustrate how separation problems can be solved, we will consider the SECs for the TSP and the cover inequalities for knapsack constraints. Given a solution x' to the LPR of TSP with some or all of the subtour constraints

$$\sum_{i \in S,\, j \notin S} x_{ij} \geq 2$$

ignored, x satisfies all of them if and only if the minimum value of

$$\sum_{i \in S,\, j \notin S} x_{ij}$$

is ≥ 2. But this is the problem of finding a minimum cut or maximum flow in a graph with edge weights x_{ij} and can be solved efficiently. If the minimum cut has weight = 2, all SECs are satisfied; otherwise the set S associated with the minimum cut yields a most violated SEC.

The situation with knapsack inequalities is not quite so nice since the separation problem is much harder. Note that we can rewrite a cover inequality as

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

Then given a solution x to the MIP, it satisfies all of the cover inequalities if and only if

$$\min \left\{ \sum_{j \in N} (1 - x_j) z_j : \sum_{j \in N} a_j z_j \geq b + 1,\ z_j = 0, 1 \right\}$$

is ≥ 1. Since this problem may be too hard to solve exactly, we can use a heuristic procedure to attempt to find a feasible solution with cost less than one. Such a solution yields a violated cover inequality. However, by not optimizing exactly, we may fail to find a violated cover inequality even if one exists. This situation is not a catastrophe for type II constraints since they are only needed for improving the LP relaxation. In other words, separation for type II constraints involves a delicate balance between the work needed to find them versus the help they provide. Moreover, to be useful they must be "strong" constraints in the sense of being facets of the convex hull or faces of high dimension. In general, cover inequalities are not strong enough and must be lifted to inequalities of the form

$$\sum_{j \in C} x_j + \sum_{j \notin C} a_j x_j \leq |C| - 1$$

to be useful. We will not discuss lifting here other than to say that it too must be done quickly, and heuristic procedures are frequently used so long as they guarantee that the resulting inequality is valid.

The difference between the SECs and cover inequalities illustrates another point. Cover inequalities can be used whenever a problem has one or more knapsack inequalities. Therefore, they are useful for general 0-1 problems. So it is reasonable for a general-purpose code to have a separation routine for cover inequalities. In contrast, SECs are rather specific for TSPs and related problems, so it is very unlikely that a general-purpose code would include a separation routine for SECs. This makes it imperative for optimization routines to provide user interfaces for separation routines and other customizing capabilities.

Decomposition and column generation

Large linear programs are almost always sparse, not only in terms of individual coefficients but also in that an overall structure is present that is also sparse. For example, multitime-period problems have linear programs for each time period, together with a small number of linking variables representing storage or shipping of goods from an earlier time period to a later time period. Multicommodity flow problems, as another example, have many single commodity network flow problems linked by joint capacity constraints on some of the arcs. Multiscenario stochastic programs have linking variables representing current stage decision variables. These variables enter into all subsequent linear programs for all outcomes of random events in the model.

Linear programming decomposition work goes back to the Dantzig-Wolfe decomposition meth-

od.[33] There are by now several structures for decomposition methods.[26] A description of available options in OSL release 2 is given by Forrest and Tomlin.[1] We focus on Dantzig-Wolfe decomposition because it is closely related to column generation methods frequently used in combinatorial optimization.

A major reason for using column generation is that easier integer programs can be solved.

An example of a column generation method in combinatorial optimization is given by the method for crew-pairing optimization in Anbil et al.[8] There, many columns are generated, and the linear program over them is solved. A true column generation procedure does not generate large sets of columns but only a few that price out to enter the linear programming basis. These columns are found by solving another, smaller linear program in the usual Dantzig-Wolfe decomposition. In other cases, they might be found by solving shortest path problems, as in multicommodity flows,[34] or by solving an integer programming problem, as in the cutting stock problem[35] where the columns are solutions to easy integer programs, namely knapsack problems. A recent example of the latter approach for solving clustering problems is given by Johnson et al.[36] OSL is used to solve both the small integer programming problems and the master linear program over them.

A major reason for using column generation, as done for cutting stock problems and clustering problems, is that easier integer programs (knapsack problems in the case of the cutting stock problem) can be solved, and in addition, the resulting master problem is a better linear programming formulation. A discussion of this point as well as the cutting stock and clustering examples is given in Johnson.[37] To summarize the major point, the linear program with many columns is actually an integer program: the solution needs to be integer. In some cases, such as the cutting stock problem, the linear program turns out to be close to integer, and a rounding procedure seems to work well enough. In other cases, such as large-scale crew pairing, getting integer solutions seems to be difficult. A major difficulty is that usual branch-and-bound does not work well in a column generation framework, and some branching rule must be found that can be employed in the column generation procedure.

Future needs in mathematical programming

Mathematical programming is in a healthy state. The combination of remarkable advances in algorithms and computers has made it possible to solve linear and integer programs of sizes and with speeds that we did not even dream possible ten years ago. Moreover, most of our computational work can be done on more accessible and less expensive workstations. Codes are being written that take advantage of vector processors, parallel architectures, cache, and large extended memory. These advances have brought about a much greater use of mathematical programming in distribution, the airline and finance industries, and certain areas of manufacturing such as very large-scale integrated (VLSI) design and printed circuit board production.

To summarize, the main algorithmic advances of the past five years have been:

- Interior point algorithms for linear programming
- Improvements in efficiency of simplex methods, which were stimulated by the competition with interior point algorithms
- Important computational advances in numerical linear algebra helping both of the above developments, but especially interior point methods
- Better formulations, preprocessing, and strong cutting plane algorithms for 0-1 integer programming

At the same time, implementation possibilities for these advances became much more accessible through the callable library philosophy of the modern mathematical programming codes.

In linear and integer programming, three main future requirements are:

1. To start interior point algorithms from good solutions
2. To use decomposition and column generation

for massive problems and to take advantage of structure such as network flow subproblems
3. To adapt algorithms, particularly the simplex algorithm, to take advantage of massively parallel computers

For integer programs, improvements in linear programming, especially the first requirement above, can help provide faster algorithms. Decomposition and column generation have even more importance for integer programming in that, in addition to the issue of solving linear programs more quickly, column generation can lead to better linear programming relaxations, making problems vastly more tractable. Use of parallel computers should be easier and offers greater returns in integer programming where large branch-and-bound trees need to be explored and can be processed in parallel. In addition to the above three requirements, we would also list one more for integer programs: extend the success in solving combinatorial 0-1 problems to mixed-integer programming. This requirement is especially important for a wide variety of industrial problems such as production scheduling.

Beyond linear and integer programming, the main new area for codes and applications might well be nonlinear programming. With barrier methods coming from nonlinear programming to help solve linear programs, it makes sense to ask when nonlinear programs might be solved in large-scale commercial applications by other than linear approximations or by sequentially solving linear approximations. At least three large-scale computational codes are available and being used. The oldest and best known is probably MINOS.[38] It linearizes the nonlinear parts and uses a superbasic linear programming approach. Computational efficiency depends on the number of superbasic variables used, so this approach is generally good for problems with mostly linear constraints and objective function. A newer effort is Lancelot,[39] a code that uses a trust region approach and second derivative information, without exactly solving quadratic programs, in order to get a search direction. It uses partial separability to exploit the nonlinear structure and has good convergence properties.[40] Recently, optimization codes have been incorporated into spreadsheets, and Lasdon's GRG2 code[41] is used in this way. It is a reduced gradient code that handles memory requirements efficiently.

Although one hears it said, "problems are all nonlinear with discrete and stochastic elements," finding nonlinear programming applications on the scale that exists for linear, or even integer, programming is not easy. Petroleum and chemical companies have known about and understood their nonlinearities for a long time, but the predominant methodology still seems to be sequential linear programming. With the success of linear programming barrier methods, an increasing demand for nonlinear programming codes seems inevitable. Because of the fact that there is not one single methodology or one single structure for problems, a library approach seems necessary, perhaps with either automatic or interactive problem analysis. This need itself presents a considerable challenge.

Beyond linear and integer programming, the main new area for codes and applications might well be nonlinear programming.

Incorporation of integer variables into nonlinear programs is an even larger challenge. One area fairly easy to address is convex quadratic programs with integer variables. Branch-and-bound only needs a solver for the continuous relaxation, and convex quadratic problems can be easily solved at each node of a branch-and-bound tree. The same remark applies to convex nonlinear programs, where the objective is to minimize a convex function over a convex region, except that optimizing at each node may be inordinately time-consuming. For general nonlinear programs, an integrality restriction introduces another nonconvexity that greatly complicates an already difficult problem.

When we try to introduce the stochastic element, the most promising approach seems to be multiscenario stochastic programming. In order to make effective use of this modeling approach, it needs to be better understood in practice and needs to be solvable either by decomposition, by

column generation, or directly by a linear programming solver. Since the resulting linear programs are very large, introducing integer variables can quickly lead to an intractable problem. The advantage to be gained is the ability to make real-time decisions that will be more robust in adapting to a variety of possible future outcomes without knowing what might happen in the future.

Finally, we will say a few words about directions that an optimization library, such as OSL, might need to develop in order to remain in the forefront of the rapid movement we see in optimization software. There are essentially three important areas: (1) performance, (2) function, and (3) usability.

Performance is relatively straightforward, at least in principal. What is needed is simple: speed, robustness, numerical stability, effective use of hardware, and timely incorporation of algorithmic developments. Function is more difficult to forecast, but our predictions are: decomposition, stochastic, and nonlinear functionality. As function increases, some type of problem analyzer to detect structure and take effective action will become important.

Interactive problem analysis brings us to the third point: usability. However, usability is much more than interactive problem analysis. Everyone wants something better, easier, more friendly, etc., but it is not so easy to know exactly what the right tools are. Mathematical Programming System Extended[42] Version 2 (MPSX.V2) has an interactive front end that includes among other things a picture and view capability that seems to be very popular. However, MPSX.V2 is based on menu-driven and graphic software not available on workstations or personal computers, machines that are ideally suited to the modeling of problems. We list some features that seem desirable for increased usability:

1. Links to the sources of problem data, principally relational databases and spreadsheets
2. Links to existing matrix generators and potential matrix generators based on database and spreadsheet software
3. Links to other applications, simulations, forecasts, etc.
4. Graphical tools for viewing the matrix and data
5. Windows to view various aspects of the model: matrix generators, data, matrices, solutions
6. Example models that can be built on or put together to form larger models
7. Assistance in tracking down solution anomalies
8. Views of the model at aggregate as well as detailed levels

There may well be others of equal importance. What seems clear is that many users would like some higher-level assistance. The subroutine structure of OSL opens up flexible use, but there is always the need for tools for modeling and problem analysis even among sophisticated users. Less sophisticated users may be content with tools to set up database connections to existing models. Embedding optimization software in spreadsheets may be the quickest way to provide access for new users with small problems.

In conclusion, we can say that mathematical programming is making good use of hardware to provide solutions rapidly. However, it needs to move into the mainstream of problem-solving computing, and that means making it easier to use and making solutions more meaningful.

Cited references

1. J. J. H. Forrest and J. A. Tomlin, "Implementing the Simplex Method for the Optimization Subroutine Library," *IBM Systems Journal* **31**, No. 1, 11–25 (1992, this issue).
2. J. J. H. Forrest and J. A. Tomlin, "Implementing Interior Point Linear Programming Methods in the Optimization Subroutine Library," *IBM Systems Journal* **31**, No. 1, 26–38 (1992, this issue).
3. D. L. Jensen and A. J. King, "A Decomposition Method for Quadratic Programming," *IBM Systems Journal* **31**, No. 1, 39–48 (1992, this issue).
4. E. L. Johnson, M. M. Kostreva, and U. H. Suhl, "Solving 0-1 Integer Programming Problems Arising from Large Scale Planning Models," *Operations Research* **33**, 803–819 (1985).
5. T. G. Mairs, G. W. Wakefield, E. L. Johnson, and K. Spielberg, "On a Production Allocation and Distribution Problem," *Management Science* **24**, 1622–1630 (1978).
6. H. Markowitz, Portfolio Selection, *Journal of Finance* VII (1952).
7. D. L. Jensen and A. J. King, "Frontier: A Graphical Interface for Portfolio Optimization in a Piecewise Linear-Quadratic Risk Framework," *IBM Systems Journal* **31**, No. 1, 62–70 (1992, this issue).
8. R. Anbil, R. Tanga, and E. L. Johnson, "A Global Approach to Crew-Pairing Optimization," *IBM Systems Journal* **31**, No. 1, 71–78 (1992, this issue).
9. R. Anbil, E. Gelman, B. Patty, and R. Tanga, "Recent

Advances in Crew-Pairing Optimization at American Airlines," *Interfaces* **21**, No. 1, 62–74 (1991).
10. J. Abara, "Applying Integer Linear Programming to the Fleet Assignment Problem," *Interfaces* **19**, 20–28 (1989).
11. D. Goldfarb and M. J. Todd, "Linear Programming," in Handbooks in Operations Research and Management Science, Vol. 1, *Optimization*, G. L. Nemhauser et al., Editors, North-Holland, Amsterdam (1989), pp. 73–170.
12. V. Klee and G. J. Minty, "How Good Is the Simplex Algorithm?," in *Inequalities III*, O. Shisha, Editor, Academic Press, New York (1972), pp. 159–175.
13. E. Tardos, "A Strongly Polynomial Minimum Cost Circulation Algorithm," *Combinatorica* **5**, 247–255 (1985).
14. K. H. Borgwardt, *The Simplex Method: A Probabilistic Analysis,* Springer-Verlag, New York (1987).
15. L. G. Khachian, "A Polynomial Algorithm in Linear Programming," *Soviet Mathematics Doklady* **20**, 191–194 (1979).
16. R. G. Bland, D. Goldfarb, and M. J. Todd, "The Ellipsoid Method: A Survey," *Operations Research* **29**, 1039–1091 (1983).
17. M. Grotschel, L. Lovasz, and A. Schrijver, *Geometric Algorithms and Combinatorial Optimization,* Springer-Verlag, New York (1988).
18. N. Karmarkar, "A New Polynomial Time Algorithm for Linear Programming," *Combinatorica* **4**, 373–395 (1984).
19. R. Marsten, R. Subramanian, M. Saltzman, I. Lustig, and D. Shanno, "Interior Point Methods for Linear Programming: Just Call Newton, Lagrange, and Fiacco and McCormick!" *Interfaces* **20**, 105–116 (1990).
20. I. J. Lustig, R. E. Marsten, and D. F. Shanno, "Computational Experience with a Primal-Dual Interior Point Method for Linear Programming," *Linear Algebra and Its Applications* **152**, 191–322 (1991).
21. E. R. Barnes, "A Variation of Karmarkar's Algorithm for Solving Linear Programming Problems," *Mathematical Programming* **36**, 174–182 (1986).
22. P. E. Gill, W. Murray, M. A. Saunders, J. A. Tomlin, and M. H. Wright, "On Projected Newton Barrier Methods for Linear Programming and an Equivalence to Karmarkar's Projective Method," *Mathematical Programming* **36**, 183–209 (1986).
23. A. V. Fiacco and G. P. McCormick, *Nonlinear Programming: Sequential Unconstrained Minimization Techniques*, Wiley, New York (1968).
24. N. Megiddo, "Pathways to the Optimal Set in Linear Programming," in *Progress in Mathematical Programming*, N. Megiddo, Editor, Springer-Verlag, New York (1989), pp. 131–158.
25. K. Hoffman and M. W. Padberg, "LP-Based Combinatorial Problem Solving," *Annals of Operations Research* **4**, 145–194 (1985).
26. G. L. Nemhauser and L. A. Wolsey, *Integer and Combinatorial Optimization*, Wiley, New York (1988).
27. G. L. Nemhauser and L. A. Wolsey, "Integer Programming," in Handbooks in Operations Research and Management Science, Vol. 1, *Optimization*, G. L. Nemhauser et al., Editors, North-Holland, Amsterdam (1989), pp. 447–527.
28. E. M. L. Beale, "Branch and Bound Methods for Mathematical Programming Systems," *Annals of Discrete Mathematics* **5**, 201–219 (1979).
29. A. H. Land and S. Powell, "Computer Codes for Problems of Integer Programming," *Annals of Discrete Mathematics* **5**, 221–269 (1979).
30. E. Balas and R. Martin, "Pivot and Complement: A Heuristic for 0-1 Programming," *Management Science* **26**, 86–96 (1980).
31. H. P. Crowder, E. L. Johnson, and M. W. Padberg, "Solving Large-Scale 0-1 Linear Programming Problems," *Operations Research* **31**, 803–834 (1983).
32. K. Hoffman and M. W. Padberg, "Techniques for Improving the LP-Representation of 0-1 Linear Programming Problems," *ORSA Journal on Computing* **3**, 121–134 (1991).
33. G. B. Dantzig and P. Wolfe, "Decomposition Principle for Linear Programs," *Operations Research* **8**, 101–111 (1960).
34. C. Barnhart, E. L. Johnson, R. Anbil, and L. Hatay, *A Column Generation Technique for the Long Haul Crew Assignment Problem*, COC-91-01, Industrial and Systems Engineering, Georgia Institute of Technology, Atlanta, GA (1991).
35. P. C. Gilmore and R. E. Gomory, "A Linear Programming Approach to the Cutting Stock Problem—Part I," *Operations Research* **9**, 849–859 (1961).
36. E. L. Johnson, A. Mehrotra, and G. L. Nemhauser, *Min-Cut Clustering*, COC-91-10, Industrial and Systems Engineering, Georgia Institute of Technology, Atlanta, GA (1991).
37. E. L. Johnson, "Modeling and Strong Linear Programs for Mixed Integer Programming," in *Algorithms and Model Formulation in Mathematical Programming*, S. W. Wallace, Editor, Springer-Verlag, New York (1989), pp. 1–43.
38. B. A. Murtagh and M. A. Saunder, *MINOS 5.1 User's Guide*, Technical Report SOL83-20R, Department of Operations Research, Stanford University, Stanford, CA (1987).
39. A. R. Conn, N. L. M. Gould, and Ph. L. Toint, "An Introduction to the Structure of Large Scale Nonlinear Optimization Problems and the Lancelot Project," in *Computing Methods in Applied Sciences and Engineering*, R. Glowinski and A. Lichnewsky, Editors, SIAM, Philadelphia (1990), pp. 42–54.
40. A. R. Conn, N. L. M. Gould, and Ph. L. Toint, "A Globally Convergent Augmented Lagrangian Algorithm for Optimization with General Constraints and Simple Bounds," *SIAM Journal on Numerical Analysis* **28**, 545–572 (1991).
41. L. Lasdon and S. Smith, "Solving Large, Sparse Non-Linear Programs Using GRG," *ORSA Journal on Computing*, to appear.
42. *Mathematical Programming System Extended/370, General Information Manual*, IBM Corporation; available through IBM branch offices.

Accepted for publication September 25, 1991.

Ellis L. Johnson *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598-0218.* Dr. Johnson is an IBM Fellow in mathematical programming. He is also the Coca-Cola Professor of Industrial and Systems Engineering and Co-director of the Computational Optimization Center at Georgia Institute of Technology. Currently working in the area of mathematical programming, he established and managed the Optimization Center in the Mathematical Sciences Department at IBM Research from 1986 until 1990, during which time he oversaw the technical aspects of the development of the Optimization Subroutine Library (OSL). His work in integer programming has included

projects with General Motors and American Airlines, and he contributed to the codes for MPSX.V2 MIP and OSL MIP. The paper describing the methodology developed to solve the General Motors problems received the Lanchester Prize from ORSA/TIMS in 1983. In 1985, Dr. Johnson was awarded the Dantzig Prize of the Mathematical Programming Society and the Society of Industrial and Applied Mathematics for his mathematical programming research, and in 1987 he was elected to the National Academy of Engineering.

George L. Nemhauser *School of Industrial and Systems Engineering, Georgia Institute of Technology, Atlanta, Georgia 30332-0205*. Dr. Nemhauser is the Russell Chandler Professor of Industrial and Systems Engineering and Co-director of the Computational Optimization Center at the Georgia Institute of Technology. He previously held faculty positions at Cornell University and Johns Hopkins University and also served as Research Director of the Center for Operations Research and Econometrics of the University of Louvain. He received a Ph.D. in operations research from Northwestern University in 1961, an M.S. in chemical engineering from Northwestern in 1959, and a B.Ch.E. from City College of New York in 1958. Dr. Nemhauser's research interests are in discrete optimization. He has published many papers and three books in this field, two of which received the Lanchester Prize of the Operations Research Society of America. He is currently the editor of *Operations Research Letters* and previously served as editor of *Operations Research*. He has been a member of the council and president of ORSA and has received ORSA's Kimball Prize for distinguished services. He is currently chairman of the Mathematical Programming Society and a member of the National Academy of Engineering.

Reprint Order No. G321-5463.