

Combinatorial Optimization: Current Successes and Directions for the Future

Karla L. Hoffman¹

School of Information Technology and Engineering
George Mason University, Mail Stop 4A6
Fairfax, Virginia 22030

ABSTRACT

Our ability to solve large, important combinatorial optimization problems has improved dramatically in the decade. The availability of reliable software, extremely fast and inexpensive hardware and high-level languages that make the modeling of complex problems much faster have led to a much greater demand for optimization tools. This paper highlights the major breakthroughs and then describes some very exciting future opportunities. Previously, large research projects required major data collection efforts, expensive mainframes and substantial analyst manpower. Now, we can solve much larger problems on personal computers, much of the necessary data is routinely collected and tools exist to speed up both the modeling and the post-optimality analysis. With the information-technology revolution taking place currently, we now have the opportunity to have our tools embedded into supply-chain systems that determine production and distribution schedules, process-design and location-allocation decisions. These tools can be used industry-wide with only minor modifications being done by each user.

1. Introduction²

The versatility of the combinatorial optimization model stems from the fact that in many practical problems, activities and resources, such as machines, airplanes and people are indivisible. Also, many problems (e.g. scheduling) have rules that define a finite number of allowable choices and consequently can appropriately be formulated using procedures that transform the logical alternatives descriptions to linear constraint descriptions where some subset of the variables are required to take on certain discrete values. Such problems are labeled *mixed-integer linear optimization problems*.

This paper will consider problems whereby both the function to be optimized and the functional form of the constraints restricting the possible solutions are linear functions. Although this linear restriction might seem overly constraining, the wealth of real-world problems that either naturally assume this form or can be acceptably transformed, possibly by adding many more variables and constraints, into this mathematical structure is extraordinarily large. Thus, the general linear integer model that we will consider is:

¹ This research has been supported by a grant from the Office of Naval Research.

² We note that each section of this paper will include only a limited number of survey references. These survey papers contain the references to the much larger body of work in each area. We have chosen this approach because of the editorial policy of this volume. An alternative copy of this paper with all of the references detailed in the text can be downloaded from the author's homepage at: <http://iris.gmu.edu/~khoffman>.

$$\max \sum_{j \in B} c_j x_j + \sum_{j \in I} c_j x_j + \sum_{j \in C} c_j x_j$$

subject to:

$$\sum_{j \in B} a_{ij} x_j + \sum_{j \in I} a_{ij} x_j + \sum_{j \in C} a_{ij} x_j \sim b_i \quad (i = 1, \dots, m)$$

$$l_j \leq x_j \leq u_j \quad (j \in I \cup C)$$

$$x_j \in \{0, 1\} \quad (j \in B)$$

$$x_j \in \text{integers} \quad (j \in I)$$

$$x_j \in \text{reals} \quad (j \in C)$$

Where B is the set of zero-one variables, I is the set of integer variables, C is the set of continuous variables, and the \sim symbol in the first set of constraints denotes the fact that the constraint $i=1, \dots, m$ can be either \leq , $=$, or \geq . The data l_j and u_j are the lower and upper bound values, respectively, for variable x_j . As we are discussing the integer case, there must be some variable in $B \cup I$. If $C=I=\emptyset$, then the problem is referred to as a pure 0-1 linear-programming problem; if $C = \emptyset$, the problem is called a pure integer (linear) programming problem. Otherwise, we problem is a mixed integer (linear) programming problem. Throughout this discussion, we will call the set of points satisfying all constraints S , and the set of points satisfying all but the integrality restrictions, S' .

While linear optimization belongs to the class of problems for which provably good algorithms exist – i.e. algorithms for which the running time is bounded by a polynomial in the size of the input – combinatorial optimization belongs to the class of problems (called *NP-hard problems*) for which provably efficient algorithms do not exist. Even so, when one is careful in choosing among *mathematically correct alternative models* and when one takes advantage of the specific structure of the problem, many very large and important combinatorial problems have been solved in reasonable times. Thus, we begin the discussion by highlighting some of the formulation issues that determine the solvability of the problem.

2. Formulation Issues

Since there are often different ways of mathematically representing the same problem, and, since obtaining an optimal solution to a large integer programming in a reasonable amount of computing time may well depend on the way it is “formulated”, much recent research has been directed toward the reformulation of combinatorial optimization problems. In this regard, it is sometimes advantageous to increase (rather than decrease) the number of integer variables, the number of constraints, or both. When we discuss the

notion of a “good” formulation, we normally think about creating an easier problem to solve that approximates well, the objective function value of the original problem. Since it is the integrality restrictions on the decision variables that destroys the convexity of the feasible region, the most widely used approximation removes this restriction; Such an approximation is known as the *Linear-Programming (LP) relaxation*. However, merely removing these integrality restrictions can alter the structure so significantly that the LP solution is far from the integer solution. One might therefore consider adding additional restrictions to the problem so that, at least in the vicinity of the optimal solution, the linear programming polytope closely approximates the polyhedron described by the *convex hull* of all feasible points to the original combinatorial optimization problem. When one considers adding such constraints to the LP-relaxation iteratively within an overall algorithm, the algorithm is called a *cutting plane algorithm*. More will be said about this in the section on solution approaches.

An example of two very different formulations for the same problem is the *machine-shop-scheduling problem*. Early formulations of this problem took a straight-forward approach of defining the decision variables to be the time at which job i started on machine j , while an alternative formulation might consider providing feasible schedules for each machine and then combining these schedules to form feasible solutions for each job. The first of these formulations has as its linear programming relaxation, an objective function value that is far from the true objective value. The second requires the generation of feasible schedules as input to the formulation and the number of such possible schedules can be enormous. However, the second formulation, although appearing far more work for the modeler and far larger, is the one that allows solvability with current computing technologies. For more on how column generation handles this problem, see the section on Column Generation in this paper.

To illustrate the importance of careful formulations, we include some examples of formulation alternatives that make a difference in the length of time it will take to solve a combinatorial problem. One will obtain a better formulation is one performs, for example, *constraint disaggregation*: In this case, one removes the constraint $\sum_{j=1, \dots, m} x_j = mx_0$ and replaces it with the m -constraints: $x_j = x_0$ for $j=1, \dots, m$. Similarly, whenever a collection of variables is indistinguishable (e.g. one has k identical machines in a machine scheduling problems), one must provide a decision hierarchy that prioritizes among the identical objects. Otherwise, the LP relaxation will continue to interchange the identical machines and provide alternative fractional solutions with the same objective function value.

Finally, the user must supply bounds that are as tight as they can be, or have the solution procedure attempt to search for tight bounds by examining individual constraints, by probing – a term used to consider the implications of fixing a single variable on all other variables in the problem, or by solving related optimization problems. Without tight bounds, coefficients in the formulation are likely to be large and the resulting LP relaxation weak.

Recently, reformulating these problems as either set-covering or set-partitioning problems, having an extraordinary number of variables, allowed the solution of a variety of difficult problems. Because, for even small instances of the problem, the problem size cannot be explicitly solved, techniques known as *column generation*, which began with the seminal work of Gilmore and Gomory on the cutting stock problem, are employed.

Bramel and Simchi-Levi have shown that the set-partitioning formulation for the vehicle routing problem with time windows is a tight formulation, i.e. the relative gap between the fractional linear programming solution and the global integer solution are close. Similar results have been obtained for the bin-packing problem and the machine-scheduling problem.

Because formulation has such a significant impact on the solvability of the problem, most software packages now contain “automatic” reformulation or *preprocessing* procedures. For discussions of alternative formulation approaches, see the textbook by [Williams, 1998] and the paper by [Hoffman and Padberg, 1991]. Such preprocessing includes the elimination of variables (by fixing them to their only feasible value deduced through logical implications), the elimination of redundant or non-binding constraints, the tightening of bounds on the variables, coefficient improvement within a row, deducing additional constraint restrictions (adding of cover or clique inequalities), and using ideas from disjunctive programming to strengthen the formulation. Current general-purpose software packages continue to expand the use of such automatic reformulation strategies.

2. Exact Solution Strategies

Solving combinatorial optimization problems, i.e. finding an optimal solution to such problems can be a difficult task. The difficulty arises from the fact that unlike linear programming, the feasible region of the combinatorial problem is not a convex set. Thus, we must, instead, search a lattice of feasible points, or in the case of the mixed integer case, a set of disjoint half-lines or line segments to find an optimal solution. In linear programming, due to the convexity of the problem, we can exploit that fact that any local solution is a global optimum. In integer programming, problems have many local optima and finding a global optimum to the problem requires one to prove that a particular solution dominates all feasible points by arguments other than the calculus-based derivative-approaches of convex programming.

There are a number of quite different approaches for solving integer-programming problems, and currently, they are frequently combined into “hybrid” solutions that try to exploit the benefits of each. We will highlight the attributes of enumerative techniques, relaxation and decompositions approaches, and of cutting planes. We will then indicate how these have been powerfully combined to tackle very difficult problems.

Enumerative approaches: The simplest approach to solving a *pure* integer-programming problem is to enumerate all finitely many possibilities. However, due to the “combinatorial explosion” resulting from the parameter “size,” only the smallest instances could be solved by such an approach. Sometimes one can *implicitly* eliminate many

possibilities by domination or feasibility arguments. Besides straight-forward or implicit enumeration, the most commonly used enumerative approach is called *branch and bound*, where the "branching" refers to the enumeration part of the solution technique and bounding refers to the fathoming of possible solutions by comparison to a known upper or lower bound on the solution value. To obtain an upper bound on the problem (we presume a maximization problem), the problem is relaxed in a way which makes the solution to the relaxed problem, relatively easy to solve.

All commercial branch-and-bound codes relax the problem by dropping the integrality conditions and solve the resultant continuous linear programming problem over the set S' . If the solution to the relaxed linear programming problem satisfies the integrality restrictions, the solution obtained is optimal. If the linear program is infeasible, then so is the integer program. Otherwise, at least one of the integer variables is fractional in the linear programming solution. One chooses one or more such fractional variables and "branches" to create two or more subproblems each of which exclude the prior solution but do not eliminate any feasible integer solutions. These new problems constitute "nodes" on a branching tree, and a linear programming problem is solved for each node created. Nodes can be fathomed if the solution to the subproblem is infeasible, satisfies all of the integrality restrictions, or has an objective function value worse than a known integer solution. A variety of strategies that have been used within the general branch-and-bound framework is described in [Linderoth and Savelsbergh [1998].

Lagrangian Relaxation and Decomposition Methods: Relaxing the integrality restriction is not the only approach to relaxing the problem. An alternative approach to the solution to integer programming problems is to take a set of "complicating" constraints into the objective function in a Lagrangian fashion (with fixed multipliers that are changed iteratively). This approach is known as *Lagrangian relaxation*. By removing the complicating constraints from the constraint set, the resulting sub-problem is frequently considerably easier to solve. The latter is a necessity for the approach to work because the subproblems must be solved repetitively until optimal values for the multipliers are found. The bound found by Lagrangian relaxation can be tighter than that found by linear programming, but only at the expense of solving subproblems in *integers*, i.e., only if the subproblems do not have the *Integrality Property*. (A problem has the integrality property if the solution to the Lagrangian problem is unchanged when the integrality restriction is removed). Lagrangian relaxation requires that one understand the structure of the problem being solved in order to then relax the constraints that are "complicating". A related approach that attempts to strengthen the bounds of Lagrangian relaxation is called *Lagrangian decomposition*. This approach consists of isolating sets of constraints so as to one obtain separate, easy problems to solve over each of the subsets. Creating linking variables, which link the subsets, increases the dimension of the problem. All Lagrangian approaches are problem-structure dependent and no underlying general theory -- applicable to, for example, arbitrary zero-one problems -- has evolved.

Most Lagrangian-based strategies provide approaches, which deal with special row structures. Other problems may possess special column structure, such that when some subset of the variables is assigned specific values, the problem reduces to one that is easy to solve. Benders' decomposition algorithm fixes the complicating variables, and solves the resulting problem iteratively. Based on the problem's associated dual, the

algorithm must then find a cutting plane (i.e. a linear inequality) which "cuts off" the current solution point but no integer feasible points. This cut is added to the collection of inequalities and the problem is re-solved. The texts by Nemhauser and Wolsey [1985] and Martin [1998] provide excellent discussions of relaxation and decomposition methods.

Finally, recent work on algorithms for solving the continuous semi-definite programming problem – a generalization of linear programming -- are leading researchers to formulations that consider a semi-definite relaxation of the combinatorial optimization problem. Specifically, [Goemans and Williamson, 1995] have shown that such relaxations provide very strong bounds for the MAX 2SAT (proven to be within .931 of optimality), MAX 3SAT (proven to be within 7/8 of the optimal solution), and the maximum cut and MAX DICUT problems. The satisfiability problem of propositional logic is to determine whether or not an assignment of truth values (or the negation) to the variables exists such that the conjunction of all clauses in a truth statement can be satisfied by that assignment. One can transform each clause into a string of quadratic inequalities that significantly tighten the formulation. With the appearance of semi-definite programming software, we can expect to see many important graph-theoretic problems being reformulated in this manner.

Since each of the decomposition approaches described above provide a bound on the integer solution, they can be incorporated into a branch and bound algorithm, instead of the more commonly used linear programming relaxation. However, these algorithms are special-purpose algorithms in that they exploit the "constraint pattern" or special structure of the problem.

Cutting Plane algorithms based on polyhedral combinatorics: Significant computational advances in exact optimization have taken place. Both the size and the complexity of the problems solved have been increased considerably when *polyhedral theory*, developed over the past twenty-five years, was applied to numerical problem solving. The underlying idea of polyhedral combinatorics is to replace the constraint set of an integer-programming problem by an alternative convexification of the feasible points and extreme rays of the problem.

In 1935, Weyl established the fact that a convex polyhedron can alternatively be defined as the intersection of finitely many halfspaces *or* as the convex hull plus the conical hull of some finite number of vectors or points. If the data of the original problem formulation are *rational* numbers, then Weyl's theorem implies the existence of a finite system of linear inequalities whose solution set coincides with the *convex hull* of the mixed-integer points in S which we denote $conv(S)$. Thus, if we can list the set of linear inequalities that completely define the *convexification* of S , then we can solve the integer-programming problem by linear programming. Gomory [1958] derived a "cutting plane" algorithm for integer programming problems, which can be viewed as a *constructive* proof of Weyl's theorem, in this context.

Although Gomory's algorithm converges to an optimal solution in finite number of steps, the convergence to an optimum is extraordinarily slow due to the fact that these algebraically-derived cuts are "weak" in the sense that they frequently do not even define supporting hyperplanes to the convex hull of feasible points. Worse yet, when many Gomory cuts are added to a problem, the cuts generated may be nearly parallel and

thereby cause serious ill conditioning in the basis-matrix requiring factorization. Finally, an additional problem with these cutting planes was that, if generated within a branch-and-bound tree, the cut was not valid throughout the tree, since the basis representation used to generate these cuts, assumed that certain variables were fixed. Recent work by Balas, et al. [1993] has suggested approaches to overcome the ill-conditioning problem (by carefully considering when to branch and when to cut). Similarly, they have adopted lifting techniques originally derived for polyhedral-based cuts to force the validity of the cuts throughout the tree. We will first introduce the concepts of polyhedral-based cutting planes and then come back to the promise of Gomory cuts for mixed-integer programming.

Since one is interested in a linear constraint set for $\text{conv}(S)$ which is as small as possible, one is led to consider *minimal* systems of linear inequalities such that each inequality defines a *facet* of the polyhedron $\text{conv}(S)$. When viewed as cutting planes for the original problem then the linear inequalities that define facets of the polyhedron $\text{conv}(S)$ are "best possible" cuts -- they cannot be made "stronger" in any sense of the word without losing some feasible integer or mixed-integer solutions to the problem. Considerable research activity has focused on identifying part (or all) of those linear inequalities for specific combinatorial optimization problems -- problem-dependent implementations, of course, that are however derived from an underlying *general* theme due to Weyl's theorem, which applies generally. Since for most interesting integer-programming problems the minimal number of inequalities necessary to describe this polyhedron is exponential in the number of variables, one is led to wonder whether such an approach could ever be computationally practical. It is therefore all the more remarkable that the implementation of cutting plane algorithms based on polyhedral theory has been successful in solving problems of sizes previously believed intractable. The numerical success of the approach can be explained, in part, by the fact that we are interested in *proving* optimality of a *single extreme point* of $\text{conv}(S)$. We therefore do not require the *complete* description of F but rather only a partial description of F in the *neighborhood* of the optimal solution.

Thus, a general cutting plane approach relaxes in a first step the integrality restrictions on the variables and solves the resulting linear program over the set S' . If the linear program is unbounded or infeasible, so is the integer program. If the solution to the linear program is integer, then one has solved the integer program. If not, then one solves a *facet-identification problem* whose objective is to find a linear inequality that "cuts off" the fractional linear programming solution while assuring that all feasible integer points satisfy the inequality -- i.e. an inequality that "separates" the fractional point from the polyhedron $\text{conv}(S)$.

Most of the polyhedral-theory requires that one to identify specific sub-structures of the original problem and then based on such structures generate polyhedral cuts (or approximations to such cuts) that separate the hyperplane added from the fractional point. Clearly, we want to generate strong cuts -- i.e. cuts that approximate well the convex hull of the integer points around the optimal solution point, and one wishes to generate as few of them as necessary. The separation problem, therefore, is an optimization problem that determines the coefficients of the separating hyperplane such that the distance between this inequality and the fractional point are maximized. Many such formulations have been proposed. Most polyhedral cuts employ algorithms that generate, among all

possible, the one that has the maximum geometric distance. This approach has also been proposed for *Fenchel cuts* [Boyd, 1994], disjunctive cuts [Balas, 1975] and [Jeroslow, 1972], and [Balas, 1979] for general mixed-integer cuts.

Since most of these cuts are based on some substructure of the original problem, the cuts generated will often include only a subset of the entire variable set. The idea is “cut lifting” is quite simple – assume a cutting plane on some subset of the variables has been generated. All other zero-one variables had been assumed to be either at zero or one. We now examine the consequences of having that variable no longer restricted to remain at that bound. It is precisely this lifting procedure that allows one to take Gomory cuts and make them valid throughout the enumeration tree. The ideas related to lifting originated with Padberg [1973] and Wolsey [1975]

A further approach to determining the convex hull of the integer points considered the role that disjunctions play in zero-one optimization problems. Disjunctions are logical conditions involving the operators “and”, “or”, and “not”. Clearly, zero-one variables are natural disjunctions since these variables can only take on the two values, either zero or one. Using disjunctive arguments, Balas [1974] showed that one could incorporate all of the restrictions of a pure zero-one linear programming problem in an equivalent linear programming problem in a much higher dimensional space. Sherali and Adams [1990] provided an alternative formulation that also provides the convex hull of all integer points in a nonlinear programming formulation in a higher dimensional space. Upon first glance, these approaches may seem to provide formulations that are too enormous to be practical to consider. However, when one uses either of these formulations, and projects back into the original space of variables, one can obtain a tighter formulation through both variable substitutions and the addition of tightening cutting planes. The “lift and project” algorithm of Ceria, et al. is based on these ideas and those of “lifting” back variables not in the generated cut. Separation algorithms based on these ideas require the solution of linear programming problems. Since the process is based on a given fractional solution to an LP relaxation – and not based on any specific structure of the problem, a violated inequality can always be found. For textbooks describing in detail polyhedral cuts, as well as disjunctive cuts, Gomory cuts and Fenchel cuts, see Wolsey [1998], Martin [1998], and Nemhauser and Wolsey [1985].

A cutting-plane algorithm terminates when: 1) an integer solution is found (we have successfully solved the problem); 2) the linear program is infeasible and therefore the integer problem is infeasible; or 3) no cut is identified by the facet-identification procedures either because a full description of the facial structure is not known or because the facet-identification procedures are inexact, i.e., one is unable to *algorithmically* generate cuts of a known form, or 4) the last few rounds of cut generation has not improved the objective function value sufficiently to warrant continuing the generation process. If we terminate the cutting plane procedure because of either the third or fourth possibilities, then, in general, the process has “tightened” the linear programming formulation so that the resulting linear programming solution value is much closer to the integer solution value.

Thus, cutting planes can be used as a reformulation technique. However, we consider that the overall cutting plane approach is best if incorporated into a bounding algorithm, that allows one to generate cuts not only at the top of the tree, but also throughout the tree search. This method is called “branch and cut”. However, before we

provide an overall description of such a hybrid algorithm, we must return to our discussions of branch-and-bound. The power of such an algorithm is dependent on the strength of the bounding arguments – when the lower bound equals the upper bound, optimality is proven. Cutting plane procedures provide a mechanism for tightening the bound produced by the relaxation. We must also have another bound – namely, we must have a good feasible solution to the optimization problem. One can wait and hope that one finds this bound within the tree search, or one can use heuristics to generate good bounds early in the process.

3. Heuristics

Operations research analysts have routinely considered using heuristics to obtain good solutions for problems considered too complex to be able to obtain optimal solutions. However, the situation has drastically changed in the past few years. Now, commercial codes whose purpose is to either prove optimality or to terminate once the solution is proven to be within a specified tolerance of optimality, apply heuristic algorithms routinely throughout the procedure so that good bounds are obtained early in the algorithm. Thus, heuristics now serve two very important purposes: to provide good solutions to problems for which current algorithms are incapable of proving optimality within reasonable times and to help in the fathoming efficiency of exact algorithm.

The research in heuristics began with concepts of *local search* whereby one constructs a feasible solution and then iteratively improves that solution by performing local moves, or “swaps”. Constructive algorithms for finding the original feasible solution may be as simple as attempting to construct such a solution greedily, i.e. picking the best single move without any look-ahead, to considering the impact of both rounding up and rounding down a given variable in an linear programming solution. Improving heuristics, similarly, can consider simple neighborhoods of a current solution, or can consider more complicated moves, such as those proposed in the Lin-Kerningham algorithm for the traveling salesman problem.

Alternatives to these construction/improvement procedures became popular in the 1980’s when algorithms were proposed that allowed moves that degraded the solution in an attempt to avoid becoming stuck at local solutions. Much of this research applies techniques based on analogies from the natural world – properties of materials, natural selection, neural processing, or properties of learning found in animals.

Simulated annealing algorithms are based on the properties from statistical mechanics whereby an annealing process requires the slow cooling of metals to improve their strength. The analogy is that one will slowly converge to a feasible solution by inserting a randomization component. With a given probability, the algorithm allows moves that degrade the solution. As the algorithm progresses, however, the probability that such moves will be taken decreases. See Hansen [1986] for an overview and history of such algorithmic applications to combinatorial optimization.

Similarly, genetic or evolutionary algorithms are based on properties of natural mutation. The analogy here is more obvious, in that every feasible solution to the combinatorial optimization problem is equivalent to a DNA string and each such string is given a value. One then chooses to evolve future generations of the population with “good” attributes. The likelihood that two individuals (parents) mate is dependent upon their objective function value and the mating of two individuals creates a new solution

whose attributes are a combination of attributes of each parent. However, an offspring might also contain a mutation – i.e. an attribute that neither parent possessed. One is less likely to generate the same local solutions because the combining process does not center entirely on the best current solution. Goldberg [1989] provides a good overview of the research in this area.

Finally, neural networks are based on models of brain function. Artificial neural network algorithms have, as their essential goal, to recognize patterns and to learn “good” responses to a given pattern. In essence, a neural network consists of a set of nodes (neurons) that are capable of receiving information from neighboring nodes and then responding to such neighbors. Since each of these nodes processes the information it receives simultaneously, the idea is that these nodes serve as a powerful parallel processor of information. Eventually, the neural networks “learn” to identify good and bad attributes. There are many alternative approaches to determining the learning strategy – there are self-organizing maps, elastic nets, back-propagation algorithms, feed-forward algorithms, etc. Also, linear-programming and steepest descent algorithms are being used to help “train” nodes in the network more quickly. At the current time, neural nets have not been shown to be competitive with other heuristics. However, the rapid evolution of neural network technology may well make these algorithms effective in the future. <http://mindit.netmind.com/go/1/14821153/3302555> For a review of research in neural networks, see Smith [1999] and the entire issue of *Journal on Computing*, **5**, number 4.

Glover and Laguna [1992] have generalized many of the attributes of these methods into a method called *tabu-search*. Tabu-search is a meta-heuristic that classifies the attributes that one would wish for in an algorithm. In order to avoid returning to a known local solution too often, the algorithm keeps a list of recent moves and makes such moves forbidden for a given period of time. Thus, at each step, the algorithm must choose among moves that are feasible. The algorithm will choose a move that degrades the solution if no improving moves are possible. Other concepts built into tabu-search, include *diversification* (similar to mutations, these moves force the algorithm into a different parts of the feasible region), *long-term memory* (labeling of moves so that one prevents the repetition of the same series of moves from occurring), and *aspiration rules* (which specify when one can overlook the tabu criteria because, for example, the resulting solution is guaranteed to be better than any solution seen so far). Randomization of algorithms – including randomizing the tabu rules themselves – is easily incorporated into this framework, as is the inclusion of very sophisticated sub-algorithms. For more on meta-heuristics, see *Journal on Computing*, **11**, number 4 (1999).

One approach to obtaining good solutions to combinatorial optimization problems – used often for difficult scheduling problems – has evolved within the computer science community. Constraint programming is a language built around concepts of tree-search and logical implications. Various tools are provided to allow the user to easily explore the search space, thereby allowing users to determine the order in which variables are given specific values and the order in which such variables are specified. One language that supports such tree-search is OPL (Optimization Programming Language) and

descriptions of the language can be found in [Van Hentenryck, 1999 while the underlying strategies can be found in [MacAloon and Trekkoff, 1996].

When considering exact approaches to solving general mixed-integer programming problems, one would like to have a heuristic that employs approaches that are used for other parts of the algorithm, as well. Thus, heuristics that can exploit some or all of the information obtained from the linear-programming relaxation of the problem are most widely used. One can see how to take many of the concepts described above, and apply them to such a heuristic. The simplest approach is to consider a “dive and fix heuristic”, whereby we fix some subset of the integer variables to a fixed integer value, perform all implied fixing and preprocessing, and again solve the resulting linear programming problem. This process continues until either the LP comes back with an integer feasible solution (considered a success) or stops because there are no feasible solutions to the current LP relaxation. If the latter occurs, one can either stop the algorithm and hope to find a solution at some other iteration, or one can try back-tracking (i.e. unfixing the most recently fixed variables, and fix them to their other bound). Similarly, once only a small subset of the variables remains unfixed, one can enumerate that subset thereby allowing more likelihood of finding a feasible solution. One of the first LP-based heuristics implemented into general IP-software packages was the Pivot and Complement heuristic of [Balas and Martin, 1980].

5. Column generation

One of the recurring themes in many of the approaches to solving combinatorial optimization problems is to examine the structure of the problem and find a relaxation or decomposition of the problem that is easier to solve. One then attempts to strengthen this approximation by either adding constraints, columns or by altering the coefficients in either the constraints or the objective function. One decomposition – often referred to as *column-generation* or *branch-and-price* - that has been extraordinary successful in recent years, is that of Dantzig-Wolfe decomposition. The theory rests on the fact that any feasible point can be represented as a linear combination of the extreme points of the feasible region. Thus, if the constraint set can be divided into two segments (one with nice special structure, for example, a set-partitioning structure), and the other with a structure that allows us to generate extreme points feasible to that structure. We write the problem as:

$$\begin{array}{ll} \text{Max} & cx \\ \text{subject to:} & Ax \leq b \\ & x \in S \\ & x \text{ integer} \end{array}$$

The procedure rests on the fact that given a set $S^* = \{x \in S : S \text{ is a bounded set, } x \text{ integer}\}$ then S^* can also be represented as a finite set of points $S^* = \{y_1, y_2, \dots, y_p\}$. Thus, any point $y \in S^*$ can be represented as $y = \sum_{1 \leq k \leq p} \lambda_k y_k$ subject to the convexity constraint $\sum_{1 \leq k \leq p} \lambda_k = 1$ and $\lambda_k \in \{0, 1\}$ $k=1, 2, \dots, p$. Thus, one can formulate the problem as:

$$\text{Max } \sum_{1 \leq k \leq p} (c y_k) \lambda_k$$

subject to:

$$\begin{aligned} \sum_{1 \leq k \leq p} (A y_k) \lambda_k &\leq b \\ \sum_{1 \leq k \leq p} \lambda_k &= 1 \\ \lambda_k &\in \{0,1\}, \quad k=1,2,\dots,p. \end{aligned}$$

For most practical problems the set S^* is too large to enumerate. Instead, one begins by generating sufficient columns so that the “master problem” is guaranteed to have a feasible solution (at least in the LP relaxation to the problem). One then performs a “pricing problem”, to identify additional columns that will improve the LP solution to the master problem. This pricing algorithm uses the dual information from the master problem to generate new columns. The master problem is re-solved and the process continues until no column exists that improves the LP. Branching is performed once the LP optimum is found. This approach is especially useful when the resulting master problem has a structure, such as set partitioning that is well known to have a tight LP objective function value and whose polyhedral structure has been well-studied. In addition, this structure may remove symmetries that existed in the compact formulation, and may allow for branching on constraints, referred to as *strong branching*. Finally, there are problems for which the column-formulation is the only choice (e.g. crew-scheduling problems – For these problems, the rules determining a “feasible” schedule for a crew are so complicated that one cannot write a linear constraint set that describes all the characteristics of the problem.)

Problems that have been successfully solved using this re-formulation include the generalized assignment problem, bin-packing, graph coloring, vehicle routing with time windows, and other complicated delivery problems. For each of these problems, the resulting optimization problem has a set-partitioning, packing or covering structure. Given this structure, one carefully designs the overall algorithm so that symmetries that occur because there are identical machines, trucks or crews can be identified in the generation process. A reformulation is then done that combines the convexity constraints in such a way as to remove the symmetry. Similarly, branching strategies are employed, similar to those in constraint logic, which determines if specific trucks, machines, crews must handle specific types of customers, tasks or flights. Thus, the cutting planes, branching and re-formulation are all strengthened because one better understands the problem characteristics. For an excellent overview of column generation techniques, see Barnhart, et al. [1996] and Sol [1994].

6. Hybrid algorithms

We next explain how much of the research and development of integer programming methods can be incorporated into a super-algorithm, which uses all that is known about the problem. This method is called “branch-and-cut”.

Current software packages include many of the features described above. The major components of these hybrid algorithms consist of automatic reformulation

procedures, heuristics which provide "good" feasible integer solutions, and cutting plane procedures which tighten the linear programming relaxation to the combinatorial problem under consideration -- all of which is embedded into a tree-search framework as in the branch-and-bound approach to integer programming. Whenever possible, the procedure permanently fixes variables (by reduced cost implications and logical implications) and does comparable conditional fixing throughout the search-tree. These four components are combined so as to guarantee optimality of the solution obtained at the end of the calculation. However, the algorithm may also be stopped *early* to produce sub-optimal solutions along with a bound on the remaining error. The cutting planes generated by the algorithm are facets of the convex hull of feasible integer solutions or good polyhedral approximations thereof and as such they are the "tightest cuts" possible. Lifting procedures assure that the cuts generated are valid throughout the search-tree that aids the search process considerably.

Mounting empirical evidence indicates that both pure and mixed integer programming problems can be solved to *proven* optimality in economically feasible computation times by methods based on the polyhedral structure of integer programs. A direct outcome of these research efforts is that similar preprocessing and constraint generation procedures can be found in commercial software packages for combinatorial problems.

Finally, we are now seeing algorithms that expand not only the constraint set but also the column set. These algorithms begin by creating a "master problem" and a "pricing problem". It allows the use of all of that we have learned about constraint generation for set-covering and packing structures, allows strong branching and includes heuristics to be used to both generate columns and find feasible solutions to the master problem. There are many issues, however, that are still little understood. When one designs such algorithms, one must consider when to generate columns, when to generate additional cuts, when to search for a better feasible solution and when to branch. When one generates more columns and more constraints, the resulting LP-relaxations become harder to solve. However, the overall time spent solving the problem is likely to be reduced because the number of nodes on the branching tree is reduced substantially. Similarly, spending time finding good feasible solutions allows greater fathoming of the branching tree. It is also important to realize that the successes of these hybrid algorithms are not due to a single component but rather to the interactions and symbiotic relationship among these components. Good upper and lower bounds allow the fixing of variables. The fixing of variables changes the structure of the overall problem, implying new constraints, allowing the heuristic to find new solutions, and altering the rules for searching the tree or generating new columns. Much more testing needs to be done before we can better understand the interactions among these procedures.

The computational successes for difficult combinatorial optimization problems reflect the intense effort devoted to developing the underlying structure of these problems. These approaches may expand the dimensionality of the problem, expand the size of the constraint-set, and may require sophisticated heuristic procedures to be embedded in such algorithms. A variety of search techniques might be considered within the mega-procedure. It should be stated, however, that we would not have been able to consider applying such complicated strategies had the underlying "engine" – *linear programming* – not been able to solve the subproblems generated so efficiently. Work on

linear programming in the past ten years has substantially altered our strategies toward solving combinatorial problems. See papers in this volume on the changes in this technology. Other breakthroughs may come about because of breakthroughs in our ability to solve efficiently -- to global optimality -- nonlinear programming problems with special structure, such as semi-definite programming problems. We would then begin to use quite different relaxations, which will then alter the cutting-plane and heuristic techniques employed. Thus, successes in one optimization technology naturally bring successes in other very different structures and problems. See Wolsey [1998], Nemhauser and Wolsey [1985] and Martin [1998] for a detailed discussion of branch-and-cut and polyhedral approaches to solving many important classes of 0-1 programming problems.

7. Parallel implementations

A significant amount of research has taken place recently related to parallel implementations of combinatorial and linear programming algorithms. For linear programming, parallel factorization and pricing schemes have proven extraordinarily successful in shortening the time it takes to solve linear programming problems having millions of variables and thousands of constraints. These algorithms will play a very important role as we expand both the constraint set and the column set of the linear programming relaxations. Again, see other papers in this volume that discuss these important breakthroughs.

When considering how to alter an algorithm so that computations are done across a variety of machines, there are many alternative approaches to consider. One can provide each machine a single node of the branching tree and allow that processor to perform all work associated with that node. Alternatively, one can require that a single machine take on all work associated with a collection of branches. Similarly, one can have machines dedicated to column generation, constraint generation (possibly having many machines each devoted to generating cuts of specific type), and machines dedicated to generating feasible solutions through one or more heuristic schemes.

Parallelization of the search tree has, naturally, seen more study than any of the other approaches, since the subproblems associated with each node are completely independent. However, even in such simple approaches to parallelization, one wishes to share information among nodes as quickly as possible. [Cannon and Hoffman, 1990] designed an algorithm whereby, whenever a processor found a feasible solution better than any previously known, that solution was broadcast to all other processors. Since the only information broadcast was the *value* of the objective function value, such broadcasting was easy to perform. Knowing a better solution value allows fathoming and formulation strengthening to take place instantaneously on all nodes. In addition, these authors stored all constraints in a central pool -- a file readable by all processors-- so that many nodes could share structural information and not incur the expense of regeneration. The branch-and-cut algorithm is especially suited to this approach, since the cuts generated are applicable throughout the tree.

However, one must store these cuts in a way that avoids serious contention and latency problems. The Cannon-Hoffman approach stored cuts generated from each given row in a separate file so that various processors could be reading different files simultaneously. The file was only accessed if the processor found that the row in

question identified a fractional variable. Each cut in the file had a unique identifier so that any cuts in the file that were in the existing problem were not re-examined. Each cut also had a key structure that allowed one to also calculate the overlap between that cut and the fractional variables in the current LP solution quickly. In this way, one could examine more closely only cuts likely to be useful to that processor. Having designed a parallel version of a branch-and-cut code to exploit the characteristics of the machines being used (distributed workstations with *no* shared memory, Cannon and Hoffman were capable of achieving superlinear speedups on a set of difficult optimization problems. This approach did not have a master-slave relationship among processors, but rather used a file system again to maintain the list of all tasks still needing work. Whenever a processor completed its work, it would return to this work file and both add new tasks to the file and extract a new task from the file.

Column-generation algorithms have similar challenges to overcome. Decisions about how to share columns among processors are essential. Since each column is generated from some subset of the entire structure. Alternatively, one can store columns in files based on whether that column covers a specific row. In either approach, a processor will only examine files when needing a column having some specific structure.

Much additional research needs to be done to better understand how the many sub-algorithms now existing within an overall hybrid algorithm interact. Parallel optimization algorithms may help us “learn” when alternative approaches work best. Appelgate, et al. used many distributed workstations to prove optimality to traveling salesmen problems having over 10000 variables. The algorithm employed required substantial work at each node of the tree. They therefore wanted to carefully choose the variable to branch on before beginning such work. Such considerations resulted in a pivoting strategy to choose the branching variable that is now incorporated as an option in the single processor version of CPLEX (a widely-used software package for integer linear programming problems).

Parallel processors may also serve another very important role: currently optimization is used mostly in planning situations. Scheduling algorithms are often used to determine the optimal machine to use to accomplish specific tasks, to determine the announced schedules for crews two months prior to flying, or to determine the schedule of machines before the day begins. However, when the situation changes during the day, users require that the schedule be changed in “real-time”. Our algorithms are often not fast enough to supply such answers. Parallel implementations may be able to re-optimize a schedule as complicated as that of an airline when a major storm or maintenance situation causes the existing schedule to no longer be feasible.

7. New developments in modeling and problem generation

Much of this paper has been concerned with the *solution* of difficult and important combinatorial optimization problems. This presumes that the task of correctly modeling the problem and then providing that mathematical model to a solver is a simple task. A major breakthrough in our ability to quickly solve many important problems has been in our ability to model quickly such problems and to provide to other modelers and algorithm developers language that can be quickly understood and whose structure can be readily identified. Modeling languages such as AIMMS [Bisschop & Entriken. 1993], AMPL [Fourer, et al, 1993], GAMS A. Brooke et al., 1988], MIMI, MPL and OPL [Van

Hentenryck, 1999] have allowed analysts to express their problem in languages that directly supports a natural (i.e., more word-like) statement of the problem. All of the above-mentioned languages except MIMI present the problem from a row orientation. MIMI looks at the problem from a process-oriented perspective, and formulates the model in terms of activities (columns). There are also language extensions to many of these that allow one to discuss networks in a natural arc/node descriptive form. Clearly, what is natural for one modeler may not be for another, so flexibility in the ability to describe the model has much value.

A nice attribute of the row-oriented languages is that they allow the user to state the general form of a constraint-set and have the language generate the sequence of constraints that have that form. Since the language allows long naming, as well as constructs such as “while” and “for all” statements, the model is far more readable and changeable quickly. Some of these languages allow the user to separate the model from specific data instances thereby allowing the same model to be used for many alternative instances. Some allow the automatic linking to databases eliminating the need for the extraction of data into new tables solely for the use of an optimization code. Some have Graphical User Interfaces (GUIs) that allow users to present their output in charts and graphs that help explain the results obtained. Some of these languages allow the solving of a string of optimization problems thereby providing a more natural and automatic mechanism for doing sensitivity analysis. Since all data is stored together, the results of this analysis can be displayed in a variety of intuitive, graphical ways.

One of these languages, OPL, has now incorporated language that allows constraint programming to be linked with mathematical optimization tools into a single overall modeling tool. All other languages treat the optimizer as a black box, accessible only through well-defined parameters. OPL, on the other hand, now allows the user to link concepts of user-directed tree search with concepts of optimization relaxation. This new package is a first step in bridging the gap between modelers who treat optimization as black-box solvers and code developers who need to test new algorithmic concepts.

In one sense, MINTO [Nemhauser, et al, 1994] can be considered a pre-cursor (from the optimization-communities’ perspective) to OPL. This software package allowed optimizers to use pieces of a general optimization package, and test their own sub-algorithms within this overall package. However, that package was designed specifically for algorithmic developers and did not have the higher-level modeling language tools of the packages discussed above.

OPL, on the other hand, is the first language to attempt to provide higher-level modeling tools and to link these with language constructs specifically designed to help direct tree-search activities. Specifically, constraint programming provides to the optimization community many of the constraint reasoning tools i.e., provides non-deterministic constructs that relieve a modeler from the many mundane implementation aspects of tree-search procedures. Since constraint programming is mostly concerned with proposing software architectures to simplify search algorithms, such methods are likely to be useful

in quickening the modelers ability to generate feasible solutions to difficult optimization problems.

The real strength of merging concepts of constraint programming with those of combinatorial optimization, is that we may both better understand and preserve the structure of the underlying problem and we may be able to quickly develop hybrid, meta-algorithms far more powerful than any algorithms we employ today.

Currently, the user of combinatorial optimization algorithms must transform many logical restrictions into a set of linear constraints. Such transformations – as presented in textbooks on linear and integer programming – often destroy an underlying structure and, when linear programming is used as the relaxation, often provide bounds that are far from the optimal integer solution. We believe that a better approach is to have the user supply the problem using logical operators and have the optimization procedure determine the best way to approximate the problem. Thus, instead of requiring the user to transform logical constructs, (such as “A not equal to B”, “A only if B”, “always choose A before B”), the user supplies these restrictions in the natural form. The modeling language then makes whatever transformations are best for the algorithm used. Similarly: modeling languages should, in the future, allow the user to supply fixed charges, piecewise-linear approximations and graph-related concepts (such as paths, cycles, etc) in a natural way. The user should also be able to tell the optimizer any information that might help the tree-search or the constraint generation. We do not yet understand how best to perform these tasks, but future versions of modeling languages are likely to allow the user to maintain a transparent descriptions of the underlying problems and allow the optimizer to exploit the underlying structure of such problems far more easily.

With the structure transparent, new algorithms are likely to emerge. Such meta-algorithms will allow all procedures (re-formulation, constraint generation, heuristics, column-generation, and tree-search), to choose sub-algorithms that are most useful for the problem structure exhibited.

8. Understanding the Solution

The discussion so far has concentrated on the issues associated with the initial formulation of the problem and current algorithms for solving the problem. However, users want more than a solution vector or objective function value. Users need an understanding of *why* the problem was infeasible. Much progress has been made recently in determining an *IIS* of constraints (see [Guieu and Chinneck, 1999]). That is, a subset of constraints defining the overall linear program that is itself infeasible, but for which any proper subset is feasible.

Similarly, if the problem is feasible, one wants to understand the set of constraints that drives that solution, and which constraints are redundant (or play little role in the solution obtained). One would also like to know whether bounds on specific variables are most restrictive, and, most importantly, which variables were “driving” the problem – i.e. as soon as the value of these variables is known, the problem becomes “easy” to solve.

Current software has incorporated techniques that inform the optimizer this information. We need to develop ways of presenting this information back to the modelers so that they learn far more about the underlying process than is currently provided by the solution vector itself. One software package that provides some of this information is ANALYZE, developed by Greenberg [1993] for analyzing linear and integer programs and their solutions.

As the demand for more complex modeling increases, the demand for computer-assisted modeling and analysis will increase. New approaches include the use of artificial-intelligence queries to the model and its outputs, visualization tools to understand the structure of the problem, and a variety of model management tools [Jones and Baker, 1994]. One can find a complete bibliographic listing to work on modeling languages, analysis tools and data management tools in [Greenberg,1997 and 1998].

9. Stochastic and Robust Optimization

When our ability to solve large, complex combinatorial optimization problems seemed quite limited, users were satisfied with strictly the solution to the problem posed. But, with our successes has come demands for far more challenging problems to be solved. Although this paper has focused exclusively on the solution of deterministic problems, we acknowledge that demand is growing for solution approaches to the more difficult (but far more realistic) problem – acknowledging that all data is not known with certainty. For such instances, a variety of approaches have been proposed: stochastic integer programming, chance-constrained programming, dynamic programming, and robust optimization.

The simplest approach to handling uncertainty is to estimate the mean value of each parameter and solve a deterministic problem. Then, for those values that have most variability, perform sensitivity analysis on the respective values. Of course, sensitivity analysis in integer programming requires far more effort than for the linear case, so only very small perturbations are usually considered.

Another approach – one commonly used in portfolio optimization and capital budgeting – is to force a diversification of the portfolio (i.e. add constraints that force the portfolio to choose a variety of different types of investments). A second approach adds a penalty to the objective function for the likely that a constraint will be violated because of variability in the data. A third approach adds new constraints that provide a measurement of risk and then enforces that one does not allow more risk than a given amount. In each of these cases, one has transformed the problem to a deterministic problem. Along these same lines, one can evaluate a reward to risk curve by solving a variety of deterministic optimization problems and having the user determine where along the curve he feels most comfortable.

We now present methods that address the stochasticity directly. One such method is called robust optimization. In this case, stochasticities are addressed via a set of discrete scenarios. Here, one needs to not only specify the scenarios that are likely to occur, but also the utility of the outcomes that occur under each scenario. Here, models either incorporate risk by incorporating variance measures into the objective function or by incorporating expected utility functions. However, in either case, the transformed objective function becomes nonlinear, making the problem more difficult to solve, especially when integrality conditions are imposed. Other reasons for their lack of use are

that it is often difficult to obtain the users utility function and/or variance and covariance measures. Also, the resulting solutions are less intuitive to the user. See [Mulvey, et al., 1995] for a discussion of such methods.

Stochastic optimization takes a similar approach to that of robust optimization, but instead of using an expected utility function, it incorporates a penalty for deviation from feasibility for any of the given scenarios (weighted by the expected value of the scenario occurring). The absence of general efficient methods for solving stochastic linear integer problems reflects the fact that, unlike the linear case, very few general properties are known, and what is known is discouraging. One encouraging note is that, when the random variables are appropriately described by a finite distribution, one can obtain approximation algorithms that provide bounds on the solutions obtained (see [Birge, 1997]).

Thus, although currently, there is little commercial software that incorporates these ideas, it is likely that as our technology for solving deterministic integer linear programming problems improves, however, we are far more likely to examine ways of incorporating risk issues into our models. We hope that future research will also address the issue of how to incorporate "fuzzy" data, i.e. data for which even the mean value is not known and for which one only has range estimates of its value. Research in the stochastic optimization must also address mechanisms for explaining the suggested results to users in a far more intuitive and understandable fashion. These are extremely difficult problems, and yet, those of most interest to the industrial community that has so benefited from our successes.

10. Where can these successes take us?

Until recently, only large corporations could afford to use combinatorial optimization because the costs of data collection, expensive computer machinery, analyst's time, and the training of employees to use such sophisticated tools were simply too high. Now, computing costs are no longer an issue (every small company has PCs that are capable of running extraordinarily large optimization problems). The data collection have been mostly eliminated because of sophisticated database technology (automated inventory systems, order fulfillment packages, automatic storage of customer requests, etc). Modeling languages make the time to develop and test models far shorter.

With the growth of the Internet, more people have access to sophisticated tools and information that ever before. Now organizations are faced with an environment marked by increasing complexity, economic pressure and customer expectations. The need for cost reduction and the need for fast product development is imperative. Customers have come to expect high product reliability and sophisticated functionality at a low cost. The need to accomplish these new demands has required that companies focus on their internal business processes and to create relationships with suppliers and their customers so as to achieve maximum efficiency and integration along the entire supply chain. Clearly, optimization can play an important role in these activities. Cost savings can occur by limiting inventory, by continually evaluating all of the logistics costs, and by examining how to minimize the capital tied up in the supply chain. By reducing the cumulative time between product development and delivery to the customer and by elimination of duplication of effort in the supply chain, one can obvious increase long-term profitability.

Enterprise Resource Planning (ERP) information systems are designed to “optimize” across the extended enterprise. Many such systems are in the process of embedding sophisticated combinatorial optimization models within their systems. Once they have been successfully integrated into these systems, entire industries will be using optimization tools routinely for infrastructure design, facility location and sizing, synchronization resources and material flow, resource allocation, transportation and logistics, inventory control and pricing modeling. The impact that such modeling might have on the long-term viability of enterprises could be staggering.

Another exciting challenge for the optimization community is to consider how to provide our tools over the Internet on an as-needed basis. As the software industry moves from having individuals and corporations buy software to individuals leasing software for as short as a few minutes over the Internet, optimization tools can play an important role. Conceivably, someone with a specific scheduling problem would go to a website, provide the data specific to the problem, and nearly-instantly receive a solution. That individual may use such software routinely or only once per year. One of the early entries into this market that allows users to solve optimization applications on the web is the NEOS Project [Czyzyk, et al, 1999].

To achieve these goals a number of issues must be resolved. We must provide intuitive graphical-user-interfaces so that less-technical users will be able to use our tools. We must continue to improve the tools available. Far more research on the mixed-integer problem needs to take place. Considerations of stochasticity, robustness, adjusting of solutions to small data changes (e.g. re-scheduling when something alters the availability of resources), must be considered. Being able to handle simple non-linearities must be addressed. Similarly, as we continue to improve our ability to solve larger and more complex problems, we are likely to be asked to take on even greater challenges. A very interesting collection of papers on the opportunities for optimization on the world wide web can be found in *Journal on Computing*, **10** (1998).

These needs in no way degrade the achievements already made. It is precisely the past successes that have highlighted the need to take on even greater challenges. Happily, our ability to solve more of the real-world problems appears to be accelerating as we have begun to bring divergent lines of research together into mega-algorithms. We must always remember, however, that looming in the shadows is the conjecture that $P \neq NP$, making it unlikely that we will ever be able to solve *all* of the challenges posed. The advances in information technology, make our existing tools much more useful, and provide us with a far greater set of opportunities than we could have hoped for even five years ago. We hope that many in the modeling and algorithmic community will step up to these challenges.

References

1. E.D. Anderson and Anderson, K.D. (1995) “Pre-solving in linear programming”, *Mathematical Programming* **71** 221-245.
2. D. Applegate, Bixby, R. and Cook, W. (1996) “Finding cuts in the traveling salesman problem”, Rice University technical report.
3. E. Balas and R. K. Martin (1980) “Pivot and complement: A heuristic for 0-1 programming” *Management Science* **26** 86-96.
4. E. Balas, Cornuejols, G. and Natraj, N. (1999) "Gomory Cuts Revisited", *Operations Research Letters*, to appear 1999.

5. C. Barnhart, E.L. Johnson, G.L. Nemhauser, M.W.P. Savelsbergh and P.H. Vance (1996) "Branch and price column generation for solving huge integer programs" *Operations Research* **71**, 221-245.
6. Birge, J. R. *Introduction to Stochastic Programming* 1997 Springer-Verlag.
7. J. Bisschop and R. Entriken (1993) AIMMS *The modeling system*. Paragon Decision Technology.
8. A.L. Brearly, G. Mitra and H.P. Williams (1975). "Analysis of mathematical programming problems prior to applying the simplex method," *Mathematical Programming*, **8**, 54-83.
9. A. Brooke, D. Kendrick and A. Meeraus (1988) *GAMS, A User's Guide*, The Scientific Press, Redwood City.
10. T. Cannon and K.L. Hoffman (1990) Large-scale 0-1 programming on Distributed Workstations" *Annals of Operations Research* **22**, 181-217
11. A. Chang, D. Simchi-Levi, and J. Bramel (1998) "Worst-case analysis, linear programming and the bin-packing problem" Technical Report. Dept of Industrial Engineering. Northwestern University.
12. CPLEX Optimization, Inc. (1998) Using the CPLEX Callable Library and CPLEX Mixed Integer Programming Library, Version 5.0.
13. Czyzyk, J., T. Wisniewski, and S. Wright "Optimization Case Studies in the NEOS Guide" *SIAM Review*, **41** 148-163.
14. Dash Associates (1994) XPRESS-MP User's Manual.
15. R. Fourer, D.M. Gay, and B.W. Kernighan (1993) *AMPL A Modeling Language for Mathematical Programming*, The Scientific Press.
16. F. Glover and M. Laguna (1992). "Tabu Search," a chapter in *Modern Heuristic Techniques for Combinatorial Optimization*.
17. M.X. Goemans and D.P. Williamson (1996) "Improved approximation algorithms for maximum cut and satisfiability problems using semi-definite programming" *Journal of the ACM*, **42** (6) 1113-1145.
18. D.E. Goldberg (1989) *Genetic Algorithms in Search Algorithms and Machine Learning* Addison-Wesley, Reading MA.
19. H.J. Greenberg 1993 "A computer Assisted Analysis System for Mathematical Programming Models and Solutions: A User's Guide to ANALYZE. Kluwer Academic Publishers, MA.
20. H. J. Greenberg 1998 "An annotated bibliography for post-solution analysis in mixed integer and combinatorial optimization" *Advances in Computational and Stochastic Optimization, Logic Programming and Heuristic Search*, ed. David L. Woodruff, Kluwer Academic Publishers, MA.
21. O Guieu and J. Chinneck (1999) "Analyzing Infeasible Mixed Integer and Integer Linear Programs" *INFORMS Journal on Computing* **11** 63-77.
22. P. Hansen (1986). "The steepest ascent mildest descent heuristic for combinatorial programming" *Proceedings of Congress on Numerical Methods in Combinatorial Optimization*, Capri, Italy.
23. K.L. Hoffman and M. Padberg (1991). "Improving the LP-representation of zero-one linear programs for branch-and-cut," *ORSA Journal Computing*, **3**, 121-134.
24. J. T. Linderoth and M.W.Savelsbergh (1998) "A computational study of search strategies for mixed integer programming", *INFORMS Journal on Computing* **11** 173-187.
25. R. K. Martin (1998) *Large Scale Linear and Integer Optimization*" Kluwer Academics, MA. 728pp.
26. K. McAloon and C. Tretkoff (1996) *Optimization and Computational Logic*, Wiley Interscience Series in Mathematics and Optimization. John Wiley and Sons, New York.
27. G.L. Nemhauser, M.W.P. Savelsbergh and G. Sigismondi (1994) MINTO: A Mixed Integer Optimizer." *Operations Research Letters*, **15** 47-58.
28. G.L. Nemhauser and L.A. Wolsey (1988). *Integer and Combinatorial Optimization*, John Wiley and Sons, New York.
29. M. Padberg (1995) *Linear Optimization and Extensions*, Springer Verlag, Heidelberg.
30. M. Padberg and G. Rinaldi (1991). "A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems," *SIAM Review*, **33**, 60-100.
31. R.G. Parker and R.L. Rardin (1988). *Discrete Optimization*, Academic Press, San Diego.
32. Peterson, C. and Soderberg, B (1997) "Artificial neural networks", *Local Search in Combinatorial Optimization* Edited by E. Aarts and J.K. Lenstra. John Wiley and Sons Ltd.
33. M.W.P. Savelsbergh (1994) "Preprocessing and Probing for Mixed Integer Programming Problems" *ORSA Journal on Computing* **6**, 445-454.
34. Sherali, H.D. and W.P. Adams 1994 "A hierarchy of relaxation and convex hull characterizations for mixed-integer zero-one programming problems" *Discrete Applied Mathematics* **52** 83-106.
35. A. Schrijver (1986) *Theory of Linear and Integer Programming* John Wiley and Sons, New York.

36. Van Hentenryck, P. 1999 *The OPL Optimization Programming Language* The MIT Press, Cambridge, MA
37. L.A. Wolsey (1998) *Integer Programming* JohnWiley and Sons, New York.
38. H.P. Williams (1998). *Model Building in Mathematical Programming*, 4th ed. Wiley and Sons, New York.