Noncommercial Software for Mixed-Integer Linear Programming

J. T. Linderoth[*] T. K. Ralphs[†]

December, 2004.

Revised: January, 2005.

Abstract

We present an overview of noncommercial software tools for the solution of mixed-integer linear programs (MILPs). We first review solution methodologies for MILPs and then present an overview of the available software, including detailed descriptions of eight software packages available under open source or other noncommercial licenses. Each package is categorized as a black box solver, a callable library, a solver framework, or some combination of these. The distinguishing features of all eight packages are described. The paper concludes with case studies that illustrate the use of two of the solver frameworks to develop custom solvers for specific problem classes and with benchmarking of the six black box solvers.

1 Introduction

A mixed-integer linear program (MILP) is a mathematical program with linear constraints in which a specified subset of the variables are required to take on integer values. Although MILPs are difficult to solve in general, the past ten years has seen a dramatic increase in the quantity and quality of software—both commercial and noncommercial—designed to solve MILPs. Generally speaking, noncommercial MILP software tools can't match the speed or robustness of their commercial counterparts, but they can provide a viable alternative for users who cannot afford the sometimes costly commercial offerings. For certain applications, open source software tools can also be more extensible and easier to customize than their commercial counterparts, whose flexibility may be limited by the interface that is exposed to the user. Because of the large number of open source and noncommercial packages available, it can be difficult for the casual user to determine which of these tools is the best fit for a given task. In this paper, we provide an overview of the features of the available noncommercial and open source codes, compare selected alternatives, and illustrate

[*]Department of Industrial and Systems Engineering, Lehigh University, Bethlehem, PA 18015, `jtl3@lehigh.edu`, `http://www.lehigh.edu/~jtl3`

[†]Department of Industrial and Systems Engineering, Lehigh University, Bethlehem, PA 18015, `tkralphs@lehigh.edu`, `http://www.lehigh.edu/~tkr2`

the use of various tools. For an excellent overview of the major algorithmic components of commercial solvers, especially CPLEX, LINDO, and XPRESS, we refer to reader to the paper of Atamtürk and Savelsbergh [6].

To formally specify a MILP, let a polyhedron

$$\mathcal{P} = \{x \in \mathbb{R}^n \mid Ax = b, x \geq 0\} \tag{1}$$

be represented in standard form by a constraint matrix $A \in \mathbb{Q}^{m\times n}$ and a right-hand side vector $b \in \mathbb{Q}^m$. Without loss of generality, we assume that the variables indexed 1 through $p \leq n$ are the integer-constrained variables (the *integer variables*), so that the feasible region of the MILP is $\mathcal{P}^I = \mathcal{P} \cap \mathbb{Z}^p \times \mathbb{R}^{n-p}$. In contrast, the variables indexed $p+1$ through n are called the *continuous variables*. A subset of the integer variables, called *binary variables*, may additionally be constrained to take on only values in the set $\{0,1\}$. We will denote the set of indices of binary variables by $B \subseteq \{1,2,\ldots,p\}$. The mixed-integer linear programming problem is then to compute the optimal value

$$z_{IP} = \min_{x\in\mathcal{P}^I} c^\top x, \tag{2}$$

where $c \in \mathbb{Q}^n$ is a vector defining the objective function. The case in which all variables are continuous ($p = 0$) is called a *linear program* (LP). Associated with each MILP is an LP called the *LP relaxation* with feasible region $\mathcal{P}$, obtained by relaxing the integer restrictions on the variables. For the remainder of the paper, we will use this standard notation to refer to the data associated with a given MILP and its LP relaxation.

In what follows, we review the relevant notions from the theory and practice of integer programming, referring to other sources when necessary for the full details of the techniques described. This paper is largely self-contained, though we do assume that the reader is familiar with concepts from the theory of linear programming (see [18]). We also assume that the reader has at least a high-level knowledge of object-oriented programming and functional programming interfaces. For an in-depth treatment of the theory of integer programming, we direct the reader to the works of Schrijver [77], Nemhauser and Wolsey [62], and Wolsey [86].

The paper is organized as follows. In Section 2, we sketch the branch-and-cut algorithm, which is the basic method implemented by the solvers we highlight herein, and we describe in some detail the advanced bound improvement techniques employed by these solvers. In Section 3, we discuss the various categories of MILP software systems and describe how they are typically used. In Section 4, we describe the use and algorithmic features of eight different noncommercial MILP software systems: ABACUS, BCP, BonsaiG, CBC, GLPK, lp_solve, MINTO, and SYMPHONY. Section 5 illustrates the use of two solver frameworks to develop specialized algorithms for solving specific MILP problems. In Section 6, the six noncommercial solvers that can be used as "black box" solvers are benchmarked on a suite of over 100 MILP instances. We conclude by assessing the current state of the art and trends for the future.

2 Branch and Bound

Branch and bound is a broad class of algorithms that is the basis for virtually all modern software for solving MILPs. Here, we focus specifically on *LP-based branch and bound*, in which LP relaxations of the original problem are solved to obtain bounds on the objective function value of an optimal solution. Roughly speaking, branch and bound is a divide and conquer approach that reduces the original problem to a series of smaller *subproblems* and then recursively solves each subproblem. More formally, recall that $\mathcal{P}^I$ is the set of feasible solutions to a given MILP. Our goal is to determine a least cost member of $\mathcal{P}^I$ (or prove $\mathcal{P}^I = \emptyset$). To do so, we first attempt to find a "good" solution $\bar{x} \in \mathcal{P}^I$ (called the *incumbent*) by a heuristic procedure or otherwise. If we succeed, then $\beta = c^\top \bar{x}$ serves as an initial upper bound on z_{IP}. If no such solution is found, then we set $\beta = \infty$. We initially consider the entire feasible region $\mathcal{P}^I$. In the *processing* or *bounding* phase, we solve the LP relaxation $\min_{x \in \mathcal{P}} c^\top x$ of the original problem in order to obtain a *fractional solution* $\hat{x} \in \mathbb{R}^n$ and a lower bound $c^\top \hat{x}$ on the optimal value z_{IP}. We assume this LP relaxation is bounded, or else the original MILP is itself unbounded.

After solving the LP relaxation, we consider $\hat{x}$. If $\hat{x} \in \mathcal{P}^I$, then $\hat{x}$ is an optimal solution to the MILP. Otherwise, we identify k disjoint polyhedral subsets of $\mathcal{P}$, $\mathcal{P}_1, \ldots, \mathcal{P}_k$, such that $\cup_{i=1}^k \mathcal{P}_k \cap \mathbb{Z}^p \times \mathbb{R}^{n-p} = \mathcal{P}^I$. Each of these subsets defines a new MILP with the same objective function as the original, called a *subproblem*. Based on this partitioning of $\mathcal{P}^I$, we have

$$\min_{x \in \mathcal{P}^I} c^\top x = \min_{i \in 1..k} \left(\min_{x \in \mathcal{P}_i \cap \mathbb{Z}^p \times \mathbb{R}^{n-p}} c^\top x \right), \tag{3}$$

so we have reduced the original MILP to a family of smaller MILPs. The subproblems associated with $\mathcal{P}_1, \ldots, \mathcal{P}_k$ are called the *children* of the original MILP, which is itself called the *root subproblem*. Similarly, a MILP is called the *parent* of each of its children. It is common to associate the set of subproblems with a tree, called the *search tree*, in which each node corresponds to a subproblem and is connected to both its children and its parent. We therefore use the term *search tree node* or simply *node* interchangeably with the term subproblem and refer to the original MILP as the *root node* or *root* of this tree.

After partitioning, we add the children of the root subproblem to the list of *candidate subproblems* (those that await processing) and associate with each candidate a lower bound either inherited from the parent or computed during the partitioning procedure. This process is called *branching*. To continue the algorithm, we select one of the candidate subproblems and process it, i.e., solve the associated LP relaxation to obtain a fractional solution $\hat{x} \in \mathbb{R}^n$, if one exists. Let the feasible region of the subproblem be $\mathcal{S} \subseteq \mathcal{P} \cap \mathbb{Z}^p \times \mathbb{R}^{n-p}$. There are four possible results.

1. If the subproblem has no solutions, then we discard, or *fathom* it.

2. If $c^\top \hat{x} \geq \beta$, then $\mathcal{S}$ cannot contain a solution strictly better than $\bar{x}$ and we may again fathom the subproblem.

3. If $\hat{x} \in \mathcal{S}$ and $c^\top \hat{x} < \beta$, then $\hat{x} \in \mathcal{P}^I$ and is the best solution found so far. We set $\bar{x} \leftarrow \hat{x}$ and $\beta \leftarrow c^\top \bar{x}$, and again fathom the subproblem.

4. If none of the above three conditions hold, we are forced to branch and add the children of this subproblem to the list of candidate subproblems.

We continue selecting subproblems in a prescribed order (called the *search order*) and processing them until the list of candidate subproblems is empty, at which point the current incumbent must be the optimal solution. If no incumbent exists, then $\mathcal{P}^I = \emptyset$.

This procedure can be seen as an iterative scheme for improving the difference between the current upper bound, which is the objective function value of the current incumbent, and the current lower bound, which is the minimum of the lower bounds of the candidate subproblems. The difference between these two bounds is called the *optimality gap*. We will see later that there is a tradeoff between improving the upper bound and improving the lower bound during the course of the algorithm.

The above description highlights the four essential elements of a branch-and-bound algorithm:

- *Lower bounding method*: A method for determining a lower bound on the objective function value of an optimal solution to a given subproblem.
- *Upper bounding method*: A method for determining an upper bound on the optimal solution value z_{IP}.
- *Branching method*: A procedure for partitioning a subproblem to obtain two or more children.
- *Search strategy*: A procedure for determining the search order.

With specific implementations of these elements, many different versions of the basic algorithm can be obtained. So far, we have described only the most straightforward implementation. In the sections that follow, we discuss a number of the most common improvements to these basic techniques. Even further improvements may be possible by exploiting the structure of a particular problem class.

2.1 Lower Bounding Methods

The effectiveness of the branch and bound algorithm depends critically on the ability to compute bounds that are close to the optimal solution value. In an LP-based branch-and-bound algorithm, the lower bound is obtained by solving an LP relaxation, as we have indicated. There are a number of ways in which this lower bound can be potentially improved using advanced techniques. In the remainder of this section, we describe those that are implemented in the software packages reviewed in Section 4.

2.1.1 Logical Preprocessing

One method for improving the lower bound that can be applied even before the solution algorithm is invoked is *logical preprocessing.* Using simple logical rules, preprocessing methods attempt to tighten the initial formulation, thereby improving the bound that will be produced when solving the LP relaxation. Formally, preprocessing techniques attempt to determine a polyhedron $\mathcal{R}$ such that $\mathcal{P}^I \subseteq \mathcal{R} \subset \mathcal{P}$. The bound obtained by minimizing over $\mathcal{R}$ is still valid, but may be better than that obtained over $\mathcal{P}$.

Preprocessing techniques are generally limited to incremental improvements of the existing constraint system. Although they are frequently designed to be applied to the original formulation before the solution algorithm is invoked, they can also be applied to individual subproblems during the search process if desired. Preprocessing methods include procedures for identifying obviously infeasible instances, removing redundant constraints, tightening the bounds on variables by analyzing the constraints, improving matrix coefficients, and improving the right-hand side value of constraints. For example, a constraint

$$\sum_{j=1}^{n} a_{ij}x_j \leq b_i$$

can be replaced by the improved constraint

$$\sum_{j=1, j\neq k}^{n} a_{ij}x_j + (a_k - \delta)x_k \leq b_i - \delta$$

if $k \in B$ and we can show that $\sum_{j=1,j\neq k}^{n} a_{ij}x_j \leq b_i - \delta, \forall x \in \mathcal{P}^I$. This technique is called *coefficient reduction.* Another technique, called *probing*, can be used to determine the logical implications of constraints involving binary variables. These implications are used in a variety of ways, including the generation of valid inequalities, as described next in Section 2.1.2. An extended discussion of advanced preprocessing and probing techniques, can be found in the paper of Savelsbergh [75].

2.1.2 Valid Inequalities

The preprocessing concept can be extended by dynamically generating entirely new constraints that can be added to the original formulation without excluding members of $\mathcal{P}^I$. To introduce this concept formally, we define an *inequality* as a pair (a, a_0) consisting of a coefficient vector $a \in \mathbb{R}^n$ and a right-hand side $a_0 \in \mathbb{R}$. Any member of the half-space $\{x \in \mathbb{R}^n \mid a^\top x \leq a_0\}$ is said to *satisfy* the inequality and all other points are said to *violate* it. An inequality is *valid for* $\mathcal{P}^I$ if all members of $\mathcal{P}^I$ satisfy it.

A valid inequality (a, a_0) is called *improving* for a given MILP if

$$\min\{c^\top x \mid x \in \mathcal{P}, ax \leq a_0\} > \min\{c^\top x \mid x \in \mathcal{P}\}.$$

A necessary and sufficient condition for an inequality to be improving is that is be violated by all optimal solutions to the LP relaxation, so violation of the fractional solution $\hat{x} \in \mathbb{R}^n$ generated by solving the LP relaxation of a MILP is necessary for a valid inequality to be improving. If a given valid inequality violated by $\hat{x}$ is not improving, adding it to the current LP relaxation will still result in the generation of a new fractional solution, however, and may in turn result in the generation of additional candidate inequalities. By repeatedly searching for violated valid inequalities and using them to augment the LP relaxation, the bound may be improved significantly. If such an iterative scheme for improving the bound is utilized during the processing of each search tree node, the overall method is called *branch and cut*. Generating valid inequalities in the root node only is called *cut and branch*. Branch and cut is the method implemented by the vast majority of solvers today.

Valid inequalities that are necessary to the description of $\text{conv}(\mathcal{P}^{\text{I}})$ are called *facet-defining inequalities*. Because they provide the closest possible approximation of $\text{conv}(\mathcal{P}^{\text{I}})$, facet-defining inequalities are typically very effective at improving the lower bound. They are, however, difficult to generate in general. For an arbitrary vector $\hat{x} \in \mathbb{R}^n$ and polyhedron $\mathcal{R} \subseteq \mathbb{R}^n$, the problem of either finding a facet-defining inequality (a, a_0) violated by $\hat{x}$ or proving that $\hat{x} \in \mathcal{R}$ is called the *facet identification problem*. The facet identification problem for a given polyhedron is polynomially equivalent to optimizing over the same polyhedron [37], so finding a facet-defining inequality violated by an arbitrary vector is in general as hard as solving the MILP itself. The problem of generating a valid inequality violated by a given fractional solution, whether facet-defining or not, is called the *separation problem*.

To deal with the difficulty of the facet identification problem, a common approach is to generate valid inequalities (possibly facet-defining) for the convex hull of solutions to a relaxation of the instance. In the following paragraphs, we describe relaxations of general MILPs and classes of valid inequalities arising from them. The relaxations are often created from certain instance-specific substructures, with the exception of the Gomory and MIR inequalities, which can be generated for all MILP instances. Generally, the solver determines automatically which substructures exist, but if it is known a priori that certain substructures do *not* exist, then it is worthwhile to turn off generation of the corresponding classes of inequalities. For example, many of the classes of inequalities discussed below can only be derived in the presence of binary variables. If such variables are not present, searching for such inequalities is inefficient. On the other hand, if a given class of problem instances is known to contain a substructure from which one of these classes of valid inequalities *is* derived, a solver that has the option of generating that particular class of valid inequalities should be used, if possible. The classes of valid inequalities covered here are those employed by at least one of the noncommercial solvers that we describe in Section 4.

Although subroutines for generating valid inequalities are generally integrated into the solver itself, the Cut Generation Library (CGL) is a library of open-source, free-standing subroutines for generating inequalities valid for generic MILPs. The CGL is available for download as part of the

Computational Infrastructure for Operations Research (COIN-OR) software suite [19] provided by the COIN-OR Foundation and includes separation routines for most of the classes of valid inequalities reviewed here. Three of the MILP solvers discussed in Section 4 take advantage of generic separation routines that are part of the CGL.

Knapsack Cover Inequalities. Often, a MILP has a row i of the form

$$\sum_{j \in B} a_{ij} x_j \leq b_i. \tag{4}$$

We assume without loss of generality that $a_{ij} > 0$ for all $j \in B$ (if not, we can complement the variables for which $a_{ij} < 0$). Considering only (4), we have a relaxation of the MILP, called the $0-1$ *knapsack problem*, with the feasible region

$$\mathcal{P}^{\text{knap}} = \left\{ x \in \{0,1\}^B \mid \sum_{j \in B} a_j x_j \leq b \right\}.$$

Note that in all of the relaxations we consider, the feasible region is implicitly assumed to be contained in the space $\mathbb{Z}^p \times \mathbb{R}^{n-p}$, and the variables not explicitly present in the relaxation are treated as free. Many researchers have studied the structure of the knapsack problem and have derived classes of facet-defining inequalities for it [7, 42, 84].

A set $C \subseteq B$ is called a *cover* if $\sum_{j \in C} a_j > b$. A cover C is *minimal* if there does not exist a $k \in C$ such that $C \setminus \{k\}$ is also a cover. For any cover C, we must have

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

for all $x \in \mathcal{P}^{\text{knap}}$. This class of valid inequalities are called *cover inequalities.* In general, these inequalities are not facet-defining for $\mathcal{P}^{\text{knap}}$, but they can be strengthened through a procedure called *lifting.* The interested reader is referred to the paper by Gu, Nemhauser, and Savelsbergh [40].

GUB Cover Inequalities. A *generalized upper bound* (GUB) inequality is an inequality of the form

$$\sum_{j \in Q} x_j \leq 1,$$

where $Q \subseteq B$. When MILP contains a knapsack row i of the form (4) and a set of GUB inequalities defined by disjoint sets $Q_k \subset B$ for $k \in K$, we obtain a relaxation of MILP with the feasible region

$$\mathcal{P}^{\text{GUB}} = \left\{ x \in \{0,1\}^B \mid \sum_{j \in B} a_{ij} x_j \leq b_i, \sum_{j \in Q_k} x_j \leq 1 \quad \forall k \in K \right\}.$$

This class of inequalities models a situation in which a subset of items not violating the knapsack inequality (4) must be selected, and within that subset of items, at most one element from each of the subsets Q_i can be selected. A *GUB cover* C_G is a cover that obeys the GUB constraints, (i.e., no two elements of the cover belong to the same Q_i). For any GUB cover C_G, the inequality

$$\sum_{j \in C_G} x_j \le |C_G| - 1$$

is valid for $\mathcal{P}^{\text{GUB}}$. Again, a lifting procedure, in this case taking into account the GUB constraints, can lead to significant strengthening of these inequalities. For more details of the inequalities and lifting procedure, the reader is referred to the paper of Wolsey [85] and the paper of Gu, Nemhauser, and Savelsbergh [38].

Flow Cover Inequalities. Another important type of inequality commonly found in MILP problems is a *variable upper bound*. A variable upper bound is an inequality of the form

$$x_j \le U x_k,$$

where x_j is a continuous variable ($j > p$), x_k is a binary variable ($k \in B$), and U is an upper bound on variable x_j. Such inequalities model the implication $x_k = 0 \Rightarrow x_j = 0$. Variable upper bound inequalities are often used to model a fixed charge associated with assigning a positive value to variable x_j, and they are particularly prevalent in network flow models. In such models, we can often identify a relaxation with the following feasible region:

$$\mathcal{P}^{\text{flow}} = \left\{ (x^1, x^2) \in \mathbb{R}^{n'}_{+} \times \{0,1\}^{n'} \mid \sum_{j \in N^+} x^1_j - \sum_{j \in N^-} x^1_j \le d,\ x^1_j \le U_j x^2_j,\ j \in N \right\},$$

where N^+ and N^- are appropriate sets of indices, $N = N^+ \cup N^-$, and $n' = |N|$.

A set $C = C^+ \cup C^-$ is called a *flow cover* if $C^+ \subseteq N^+$, $C^- \subseteq N^-$ and $\sum_{j \in C^+} m_j - \sum_{j \in C^-} m_j > d$. For any flow cover C, the inequality

$$\sum_{j \in C^+} x_j + \sum_{j \in C^{++}} (m_j - \lambda)(1 - y_j) \le d + \sum_{j \in C^-} m_j + \sum_{j \in L^-} \lambda y_j + \sum_{j \in L^{--}} x_j,$$

where $\lambda = \sum_{j \in C^+} m_j - \sum_{j \in C^-} m_j - d$, $C^{++} = \{j \in C^+ : m_j > \lambda\}$, $L^- \subseteq (N^- \setminus C^-)$ and $m_j > \lambda$ for $j \in L^-$, and $L^{--} = N^- \setminus (L^- \cup C^-)$, is called a *simple generalized flow cover inequality* and is valid for $\mathcal{P}^{\text{flow}}$. Just as with knapsack cover inequalities, these inequalities can be strengthened through lifting to obtain an inequality called the *lifted simple generalized flow cover inequality*. The full details of obtaining such inequalities are given by Gu, Nemhauser, and Savelsbergh [39].

Clique Inequalities. Many MILPs contain logical restrictions on pairs of binary variables such as $x_i = 1 \Rightarrow x_j = 0$. In such cases, an auxiliary data structure, called a *conflict graph*, can be used to capture these logical conditions and further exploit them [4]. The conflict graph is a graph with vertex set B, and an edge between the nodes corresponding to each pair of variables that cannot simultaneously have the value one in any optimal solution. The logical restrictions from which the conflict graph is derived may be present explicitly in the original model (for example, note that GUB inequalities lead directly to edges in the conflict graph), or may be discovered during preprocessing (see [5, 75]).

Since any feasible solution $x \in \mathcal{P}^I$ must induce a vertex packing in the conflict graph, valid inequalities for the vertex packing polytope of the conflict graph are also valid for the MILP instance from which the conflict graph was derived. Classes of inequalities valid for the vertex packing polytope have been studied by a number of authors [17, 43, 63, 65]. As an example, if C is the set of indices of nodes forming a clique in a conflict graph for a MILP instance, then the *clique inequality*

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

is satisfied by all $x \in \mathcal{P}^I$. If O is a cycle in a conflict graph for a MILP instance, and $|O|$ is odd, then the *odd-hole inequality*

$$\sum_{j \in O} x_j \leq \frac{|O| - 1}{2}$$

is also satisfied by all $x \in \mathcal{P}^I$. Again, these inequalities can be strengthened by lifting [63, 64].

Implication Inequalities. In some case, the logical implications discovered during preprocessing are not between pairs of binary variables (in which case clique and odd-hole inequalities can be derived), but between a binary variable and a continuous variable. These logical implications can be enforced using inequalities known as *implication inequalities.* If x_i is a binary variable and x_j is a continuous variable with upper bound U, the implication

$$x_i = 0 \Rightarrow x_j \leq \alpha$$

yields the implication inequality

$$x_j \leq \alpha + (U - \alpha)x_i.$$

Other implication inequalities can also be derived. For more details, the reader is referred to the paper of Savelsbergh [75].

Gomory Inequalities. In contrast to the classes of inequalities we have reviewed so far, Gomory inequalities are generic, in the sense that they do not require the presence of any particular sub-structure other than integrality and non-negativity, so they can be derived for any MILP. Gomory inequalities are easy to generate in LP-based branch and bound. After solving the current LP relaxation, we obtain an optimal basis matrix $A_B \in \mathbb{R}^{m \times m}$. The vector $A_B^{-1}b$ yields the values of the basic variables in the current fractional solution $\hat{x}$. Assuming $\hat{x} \notin \mathcal{P}^I$, we must have $(A_B^{-1}b)_i \notin \mathbb{Z}$

for some i between 1 and m. Taking u above to be the i^{th} row of A_B^{-1},

$$x_l + \sum_{j \in NB^I} uA_j x_j + \sum_{k \in NB^C} uA_k x_k = ub, \tag{5}$$

for all $x \in \mathcal{P}^I$, where NB^I is the set of nonbasic integer variables, NB^C is the set of nonbasic continuous variables, and A_j is the j^{th} column of A. Let $f_j = uA_j - \lfloor uA_j \rfloor$ for $j \in NB^I \cup NB^C$, and let $f_0 = ub - \lfloor ub \rfloor$; then the inequality

$$\sum_{\substack{j \in NB^I: \\ f_j \leq f_0}} f_j x_j + \sum_{\substack{j \in NB^I: \\ f_j > f_0}} \frac{f_0(1-f_j)}{1-f_0} x_j + \sum_{\substack{j \in NB^C: \\ ua_j > 0}} ua_j x_j - \sum_{\substack{j \in NB^C: \\ ua_j < 0}} \frac{f_0}{1-f_0} ua_j x_j \geq f_0, \tag{6}$$

is called the *Gomory mixed-integer inequality* and is satisfied by all $x \in \mathcal{P}^I$, but not satisfied by the current fractional solution $\hat{x}$. The inequality is first derived by Gomory [36], and also be derived by a simple disjunctive argument, as in Balas *et al.* [8].

Mixed-Integer Rounding Inequalities. Gomory mixed-integer inequalities can be viewed as a special case of a more general class of inequalities known as *mixed-integer rounding inequalities*. Mixed-integer rounding inequalities are obtained as valid inequalities for the relaxed feasible region

$$\mathcal{P}^{\text{MIR}} = \left\{ (x^1, x^2, x^3) \in \mathbb{R}^1_+ \times \mathbb{R}^1_+ \times \mathbb{Z}^{n'}_+ \mid \sum_{j=1}^{n'} a_j x^3_j + x^1 \leq b + x^2 \right\}. \tag{7}$$

The mixed-integer rounding inequality

$$\sum_{j=1}^{n'} \left(\lfloor a_j \rfloor + \frac{\max\{f_j - f, 0\}}{1-f} \right) x^3_j \leq \lfloor b \rfloor + \frac{x^2}{1-f},$$

where $f = b - \lfloor b \rfloor$, $f_j = a_j - \lfloor a_j \rfloor$ for $j = 1, 2, \ldots n'$, is valid for $\mathcal{P}^{\text{MIR}}$ [55, 61]. Marchand [54] established that many class of valid inequalities for structured problem instances are mixed-integer rounding inequalities, including certain subclasses of the lifted flow cover inequalities described above.

The process of generating a mixed-integer rounding inequality is a three-step procedure. First, rows of the constraint matrix are aggregated. Second, bound substitution of the simple or variable upper and lower bounds is performed, and variables are complemented in order to produce a relaxation of the form (7). Third, a heuristic separation procedure is used to find mixed-integer rounding inequalities valid for some such relaxation and violated by the current fractional solution. Marchand and Wolsey [55] discuss the application of mixed-integer rounding inequalities in detail.

2.1.3 Reduced Cost Tightening

After the LP relaxation of MILP is solved, the reduced costs of nonbasic integer variables can be used to tighten bounds on integer variables for the subtree rooted at that node. Although we have assumed a problem given in standard form, upper and lower bounds on variable values are typically present and are handled implicitly. Such bound constraints take the form $l \leq x \leq u$ for $l, u \in \mathbb{R}^n$ for all $x \in \mathcal{P}^I$. Even if no such bound constraints are initially present, they may be introduced during branching. Let $\bar{c}_j$ be the reduced cost of nonbasic integer variable j, obtained after solving the LP relaxation of a given subproblem and let $\hat{x} \in \mathbb{R}^n$ be an optimal fractional solution. If $\hat{x}_j = l_j \in \mathbb{Z}$ and $\gamma \in \mathbb{R}_+$ is such that $c^\top \hat{x} + \gamma \bar{c}_j = \beta$, where β is the objective function value of the current incumbent, then $x_j \leq l_j + \lfloor \gamma \rfloor$ in any optimal solution, so we can replace the previous upper bound u_j with $\min(u_j, l_j + \lfloor \gamma \rfloor)$. The same procedure can be used to potentially improve the lower bounds. This is an elementary form of preprocessing, but can be very effective when combined with other forms of logical preprocessing, especially when the optimality gap is small. Note that if this tightening takes place in the root node, it is valid everywhere and can be considered an improvement of the original model. Some MILP solvers store the reduced costs from the root LP relaxation, and use them to perform this preprocessing whenever a new incumbent is found.

2.1.4 Column Generation

A technique for improving the lower bound that can be seen as "dual" to the dynamic generation of valid inequalities is that of *column generation*. Most column generation algorithms can be viewed as a form of Dantzig-Wolfe decomposition, so we concentrate here on that technique. Consider a relaxation of the original MILP with feasible set $\mathcal{F} \supset \mathcal{P}^I$. We assume that $\mathcal{F}$ is finite and that it is possible to effectively optimize over $\mathcal{F}$, but that a minimal description of the convex hull of $\mathcal{F}$ is of exponential size. Let $\mathcal{Q} = \{x \in \mathbb{R}^n \mid Dx = d, x \geq 0\} \supset \mathcal{P}^I$ be a polyhedron representing the feasible region of a second relaxation whose description is "small" and such that $\mathcal{F} \cap \mathcal{Q} \cap \mathbb{Z}^p \cap \mathbb{R}^{n-p} = \mathcal{P}^I$. We can then reformulate the original integer program as

$$\min \left\{ \sum_{s \in \mathcal{F}} c^\top s \lambda_s \mid \sum_{s \in \mathcal{F}} (Ds) \lambda_s = d, \lambda_s^\top \mathbf{1} = 1, \lambda \geq 0, \lambda \in \mathbb{Z}^{\mathcal{F}} \right\}, \tag{8}$$

where $\mathbf{1}$ is a vector of all ones of conformable dimension. Relaxing the integrality restriction on λ, we obtain a linear program whose optimal solution yields the (possibly) improved bound

$$\min_{x \in \mathcal{F} \cap \mathcal{Q}} c^\top x \geq \min_{x \in \mathcal{P}} c^\top x. \tag{9}$$

Of course, this linear program generally has an exponential number of columns. It is therefore necessary to generate them dynamically in much the same fashion as valid inequalities are generated in branch and cut. If we solve the above linear program with a subset of the full set of columns to obtain a dual solution u, then the problem of finding the column with the smallest reduced cost among those that are not already present is equivalent to solving

$$\min_{s \in \mathcal{F}} (c^\top - uD) s, \tag{10}$$

which is referred to as the *column generation subproblem.* Because this is an optimization problem over the set $\mathcal{F}$, it can be solved effectively, and hence so can the linear program itself. When employed at each search tree node during branch and bound, the overall technique is called *branch and price.*

Due to the often problem-specific nature of the column generation subproblem, branch and price is frequently implemented using a solver framework. Recently two different groups have undertaken efforts to develop generic frameworks for performing column generation. Vanderbeck [83] is developing a framework for branch and price that will take care of many of the generic algorithmic details, allowing the solver to behave essentially as a black box. Galati and Ralphs [70] have undertaken a similar effort in developing DECOMP, a general framework for computing bounds using decomposition within a branch-and-bound procedure. Currently, however, implementing a branch-and-price algorithm is a somewhat involved procedure requiring a certain degree of technical expertise. Section 5.2 also describes the implementation of a branch-and-price algorithm using the BCP framework.

2.2 Upper Bounding Methods

In branch and bound, upper bounds are obtained by discovering feasible solutions to the MILP. Feasible solutions arise naturally if the branch-and-bound algorithm is allowed to run its full course. However, accelerating the process of finding feasible solutions has three potential benefits. First, the solution process may be terminated prematurely and in such a case, we would like to come away with a solution as close to optimal as possible. Second, an improved upper bound β may lead to fewer subproblems being generated due to earlier fathoming (depending on the search strategy being employed). Third, a good upper bound allows the bounds on integer variables to be tightened based on their reduced cost in the current relaxation (see Section 2.1.3). Such tightening can in turn enable additional logical preprocessing and may result in significant improvement to the lower bound as a result.

There are two ways of accelerating the process of finding feasible solutions during the search procedure. The first is to influence the search order, choosing to evaluate and partition nodes that are close to being integer feasible. This technique is further discussed in Section 2.4. The second is to use a heuristic procedure, called a *primal heuristic*, to construct a solution. Primal heuristics are applied during the search process and generally take an infeasible fractional solution as input. A very simple heuristic procedure is to round the fractional components of the infeasible solution in an attempt to produce a feasible solution. There are many ways in which to round the current solution and determining a rounding that maintains feasibility with respect to the constraints $Ax = b$ may be difficult for certain problem classes. A more sophisticated class of primal heuristics, called *pivot and complement*, involves pivoting fractional variables out of the current linear programming basis in order to achieve integrality [9, 59, 11]. Still other classes of primal heuristics use the solution of auxiliary linear programs to construct a solution. One simple, yet effective example of such a heuristic is known as the *diving* heuristic. In the diving heuristic, some integer variables are fixed and the linear program re-solved. The fixing and re-solving is iterated until either the an integral solution is found or the linear program becomes infeasible. Recent successful primal heuristics, such as local branching [29] and RINS [22], combine solving auxiliary linear programs with methods for controlling the neighborhood of feasible solutions that are being searched.

2.3 Branching

Branching is the method by which a MILP is divided into subproblems. In LP-based branch and bound, there are three requirements for the branching method. First, the feasible region must be partitioned in such a way that the resulting subproblem are also MILPs. This means that the are usually defined by imposing additional linear inequalities. Second, the union of the feasible regions of the subproblems must contain at least one optimal solution. Finally, since the primary goal of branching is to improve the overall lower bound, it is desirable that the current fractional solution not be contained in any of the members of the partition. Otherwise, the overall lower bound will not be improved.

Given a fractional solution to the LP relaxation $\hat{x} \in \mathbb{R}^n$, an obvious way to fulfill the above requirements is to choose an index $j < p$ such that $\hat{x}_j \notin \mathbb{Z}$ and to create two subproblems, one by imposing an upper bound of $\lfloor \hat{x}_j \rfloor$ on variable j and a second by imposing a lower bound of $\lceil \hat{x}_j \rceil$. This is a valid partitioning, since any feasible solution must satisfy one of these two linear constraints. Furthermore, $\hat{x}$ is not feasible for either of the resulting subproblems. This partitioning procedure is known as *branching on a variable.*

Typically, there are many integer variables with fractional values, so we must have a method for deciding which one to choose. A primary goal of branching is to improve the lower bound of the resulting relaxations. The most straightforward branching methods are those that choose the branching variable based solely on the current fractional solution and do not use any auxiliary information. Branching on the variable with the largest fractional part, the first variable (by index) that is fractional, or the last variable (by index) that is fractional are examples of such procedures. These rules tend to be too myopic to be effective, so many solvers use more sophisticated approaches. Such approaches fall into two general categories: *forward-looking methods* and *backward-looking methods*. Both types of methods try to choose the best partitioning by predicting, for a given candidate partitioning, how much the lower bound will be improved. Forward-looking methods generate this prediction based solely on locally generated information obtained by "pre-solving" candidate subproblems. Backward-looking methods take into account the results of previous partitionings to predict the effect of future ones. Of course, as one might expect, there are also hybrids that combine these two basic approaches [2].

A simple forward-looking method is the *penalty method* of Driebeek [26], which implicitly performs one dual simplex pivot to generate a lower bound on the bound improvement that could be obtained by branching on a given variable. Tomlin [80] improved on this idea by considering the integrality of the variables. *Strong branching* is an extension of this basic concept in which the solver explicitly performs a fixed and limited number of dual simplex pivots on the LP relaxations in each of the children resulting from branching on a given variable. This is called *presolving* and again provides a bound on the improvement one might see as a result of a given choice. The effectiveness of strong branching was first demonstrated by Applegate *et al.* in their work on the traveling salesman problem [3] and has since become a mainstay for solving difficult combinatorial problems. An important aspect of strong branching is that presolving a given candidate variable is a relatively expensive operation, so it is typically not possible to presolve all candidates. The procedure is therefore usually accomplished in two phases. In the first phase, a small set of candidates is chosen (usually based on one of the simple methods described earlier). In the second phase, each of these candidates is presolved and the final choice is made using one of the selection rules to be described below.

Backward-looking methods generally depend on the computation of *pseudocosts* [14] to maintain a history of the effect of branching on a given variable. Such procedures are based on the notion that each variable may be branched on multiple times during the search and the effect will be similar each time. Pseudocosts are defined as follows. With each integer variable j, we associate two quantities, P_j^- and P_j^+, that estimate the per unit increase in objective function value if we fix variable j to its floor and ceiling, respectively. Suppose that $\hat{x}_j = \lfloor \hat{x}_j \rfloor + f_j$, with $f_j > 0$. Then by branching on variable j, we will estimate an increase of $D_j^- = P_j^- f_j$ on the "down branch" and an increase of $D_j^+ = P_j^+(1 - f_j)$ on the "up branch".

The most important aspect of using pseudo-costs is the method of obtaining the values P_j^- and P_j^+ for variable j. A popular way to obtain these values is to simply observe and record the true increase in objective function value whenever variable j is chosen as the branching variable. For example, if a given subproblem had lower bound z_{LP} and its children had lower bounds z_{LP}^- and z_{LP}^+ after branching on variable j, then the pseudocosts would be computed as

$$P_j^- = \frac{z_{LP}^- - z_{LP}}{f_j} \qquad P_j^+ = \frac{z_{LP}^+ - z_{LP}}{1 - f_j}, \tag{11}$$

where f_j is the fractional part of the value of variable j in the solution to the LP relaxation of the parent. The pseudocosts may be updated using the first observation, the last observation, or by averaging all observations. Because generating pseudo-cost estimates is inexpensive, they are typically calculated for all variables.

Whether using a forward-looking or a backward-looking method, the final step is to select the branching variable. The goal is to maximize the improvement in the lower bound from the parent to each of its children. Because each parent has two (or more) children, however, there is no unique metric for this change. Suggestions in the literature have included maximizing the sum of the changes on both branches [35], maximizing the smaller of the two changes [12], or a combination of the two [27].

More general methods of branching can be obtained by branching on other disjunctions. For any vector $a \in \mathbb{Z}^n$ whose last $n - p$ entries are zero, we must have $a^\top x \in \mathbb{Z}$ for all $x \in \mathcal{P}^I$. Thus, if $a\hat{x} \notin \mathbb{Z}$, a can be used to produce a disjunction by imposing the constraint $a^\top x \leq \lfloor a^\top \hat{x} \rfloor$ in one subproblem and the constraint $a^\top x \geq \lceil a^\top \hat{x} \rceil$ in the other subproblem. This is known as *branching on a hyperplane.* Typically, branching on hyperplanes is a problem-specific method that exploits special structure, but it can be made generic by keeping a pool of inequalities that are slack in the current relaxation as branching candidates.

An example of branching on hyperplanes using special structure is *GUB branching.* If the MILP contains rows of the form

$$\sum_{j \in G} x_j = 1,$$

where $G \subseteq B$, then a valid partitioning is obtained by choosing a nonempty subset G^0 of G, and enforcing the linear constraint $\sum_{j \in G^0} x_j = 0$ in one subproblem and the constraint $\sum_{j \in G \setminus G^0} x_j = 0$ in the other subproblem. These are linear constraints that partition the set of feasible solutions, and the current LP solution $\hat{x}$ will be excluded from both resulting subproblems if G^0 is chosen so that $0 \leq \sum_{j \in G^0} \hat{x}_j < 1$. GUB branching is a special case of branching on *special ordered sets* (SOS)[1] [13]. Special ordered sets of variables can also be used in the minimization of separable piecewise-linear nonconvex functions.

Because of their open nature, noncommercial software packages are often more flexible and extensible than their commercial counterparts. This flexibility is perhaps most evident in the array of advanced branching mechanisms that can be implemented using the open source and noncommercial frameworks we describe in Section 4. Using a solver framework with advanced customized branching options, it is possible, for instance, to branch directly on a disjunction, rather than introducing auxiliary integer variables. An important example of this is semi-continuous variables, in which a variable is constrained to take either the value 0 or a value larger than a parameter K. Additionally, branching frameworks can make it easy for the user to specify prioritization schemes for branching on integer variables or to implement complex partitioning schemes based on multiple disjunctions.

2.4 Search Strategy

As mentioned in Section 2.3, branching decisions are made with the goal of improving the lower bound. In selecting the order in which the candidate subproblems should be processed, however, our focus may be on improving either the upper bound, the lower bound or both. Search strategies, or *node selection methods*, can be categorized as either *static* methods, *estimate-based* methods,

[1]Some authors refer to GUBs as special ordered sets of type 1

two-phase methods, or *hybrid* methods.

Static node selection methods employ a fixed rule for selecting the next subproblem to process. A popular static method is *best-first search*, which chooses the candidate node with the smallest lower bound. Due to the fathoming rule employed in branch and bound, a best-first search strategy ensures that no subproblem with a lower bound above the optimal solution value can ever be processed. Therefore, the best-first strategy minimizes the number of subproblems processed and improves the lower bound quickly. However, this comes at the price of sacrificing improvements to the upper bound. In fact, the upper bound will only change when an optimal solution is located. At the other extreme, *depth-first search* chooses the next candidate to be a node at maximum depth in the tree. In contrast to best-first search, which will produce no suboptimal solutions, depth-first search tends to produce many suboptimal solutions, typically early in the search process, since such solutions tend to occur deep in the tree. This allows the upper bound to be improved quickly. Depth-first search also has the advantage that the change in the relaxation being solved from subproblem to subproblem is very slight, so the relaxations tend to solve more quickly when compared to best-first search. Some solvers also allow the search tree to be explored in a breadth-first fashion, but there is little advantage to this method over best-first search.

Neither best-first search nor depth-first search make any intelligent attempt to select nodes that may lead to improved feasible solutions. Estimate-based methods such as the *best-projection method* [31, 58] and the *best-estimate method* [14] are improvements in this regard. The best-projection method measures the overall "quality" of a node by combining its lower bound with the degree of integer infeasibility of the current solution. Alternatively, the best-estimate method combines a node's lower bound, integer infeasibility, and pseudocost information to rank the desirability of exploring a node.

Since we have two goals in node selection—finding "good" feasible solutions (i.e., improving the upper bound) and proving that the current incumbent is in fact a "good" solution (i.e., improving the lower bound)—it is natural to develop node selection strategies that switch from one goal to the other during the course of the algorithm. This results in a two-phase search strategy. In the first phase, we try to determine "good" feasible solutions, while in the second phase, we are interested in proving this goodness. Perhaps the simplest "two-phase" algorithm is to perform depth-first search until a feasible solution is found, then switch to best-first search. A variant of this two-phase algorithm is used by many of the noncommercial solvers that we describe in Section 4.

Hybrid methods also combine two or more node selection methods, but in a different manner. In a typical hybrid method, the search tree is explored in a depth-first manner until the lower bound of the child subproblem being explored rises above a prescribed level in comparison to the overall lower or upper bounds, after which a new subproblem is selected by a different criterion (e.g., best-first or best-estimate), and the depth-first process is repeated. For an in-depth discussion of search strategies for mixed-integer programming, see the paper of Linderoth and Savelsbergh [50].

3 User Interfaces

An important aspect of the design of software for solving MILPs is the user interface, which determines the way in which the user interacts with the solver and the form in which the MILP instance must be specified. The range of purposes for noncommercial MILP software is quite large, so it stands to reason that the number of user interface types is also large. In this section, we give a broad categorization of the software packages available. The categorization provided here is certainly not a perfect one—some tools may fall between categories or into multiple categories. However, it does represent the typical ways in which software packages for MILP are employed in practice.

3.1 Black Box Solvers

Many users simply want a "black box" that takes a given MILP as input and returns a solution as output. For such black box applications, the user typically interacts with the solver through a command-line interface or an interactive shell, invoking the solver by passing the name of a file containing a description of the instance to be solved. One of the main differences between various black box solvers from the user's perspective is the format in which the user can specify the model to the solver. In the two sections below, we describe two of the most common input styles—raw (uninterpreted) file formats and modeling language (interpreted) file formats. Table 1 in Section 4 lists the packages covered in this paper that function as black box solvers, along with the file formats they accept and modeling languages they support. In Section 6, we provide computational results comparing all of these solvers over a wide range of instances.

3.1.1 Raw File Formats

One of the first interfaces conceived for black box solvers was a standard file format for specifying a single instance of a mathematical program. Such file formats provide a structured way of writing the constraint matrix and rim vectors (objective function vector, right hand side vector, and variable lower and upper bound vectors) to a file in a form that can be easily read by the solver. The oldest and most pervasive file format is the long standing Mathematical Programming System (MPS) format, developed by IBM in the 1970s. In MPS format, the file is divided into sections, each specifying one of the elements of the input, such as the constraint matrix, the right-hand side, the objective function, and upper and lower bounds on the variables. MPS is a column-oriented format, meaning that the constraint matrix is specified column-by-column in the MPS file. Another popular format, LP format, is similar to MPS in that the format consists of a text file divided into different sections, each specifying one of the elements of the input. However, LP format is a row-oriented format, so the constraint matrix is specified one row at a time in the file. This format tends to be slightly more readable by humans.

Since MPS was adopted as the de facto standard several decades ago, there has not been much deviation from this approach. MPS, however, is not an extensible standard, and is only well-suited for specifying integer and linear models. Several replacements have been proposed based on the extensible markup language (XML), a wide-ranging standard for portable data interchange. One of the most well-developed of these is an open standard called LPFML [33]. The biggest advantage of formats based on XML is that they are far more extensible and are based on an established standard with broad support and well-developed tools.

3.1.2 Modeling Languages

Despite their persistent use, raw file formats for specifying instances have many disadvantages. The files can be tedious to generate, cumbersome to work with, extremely difficult to debug, and not easily readable by humans. For these reasons, most users prefer to work with a *modeling language*. Generally, modeling languages allow the user to specify a model in a more intuitive (e.g., algebraic) format. An interface layer then interprets the model file, translating it into a raw format that the underlying solver can interpret directly. Another powerful feature of modeling languages is that they allow for the separation of the model specification from the instance data.

Full-featured modeling languages are similar to generic programming languages such as C and C++, in that they have constructs such as loops and conditional expressions. They also feature constructs designed specifically to allow the user to specify mathematical models in a more natural, human-readable form. Two modeling language systems that are freely available are ZIMPL [88] and Gnu Mathprog (GMPL) [53]. ZIMPL is a stand-alone parser that reads in a file format similar to the popular commercial modeling language AMPL [32] and outputs the specified math program in either MPS or LP format. GMPL is the GNU Math Programming Language, which is again similar to AMPL. The parser for GMPL is included as part of the GLPK package described in Section 4.5, but it can easily be used as a free-standing parser as well.

3.2 Callable Libraries

A more flexible mechanism for invoking a MILP solver is through a callable library interface. Most often, callable libraries are still treated essentially as a "black box," but they can be invoked directly from user code, allowing the development of custom applications capable of generating a model, invoking the solver directly without user intervention, parsing the output and interpreting the results. The use of a callable library also makes it possible to solve more than one instance within a single invocation of an application or to use the solver as a subroutine within a larger implementation. The interfaces to the callable libraries discussed in Section 4 are implemented in either C or C++, with each solver generally having its own Application Programming Interface (API). The column labeled *Callable API* in Table 1 of Section 4 indicates which of the software packages discussed in this paper have callable library interfaces and the type of interface available.

The fact that each solver has its own API makes developing portable code difficult, as there must be an inherent dependence on the use of a particular solver. Recently, however, two open standards for calling solvers have been developed that remove the dependence on a particular solver's API. These are discussed below.

3.2.1 Open Solver Interface

The Open Solver Interface (OSI), part of the COIN-OR software suite mentioned earlier, is a standard C++ interface for invoking solvers for LPs and MILPs [51]. The OSI consists of a C++ base class with containers for storing instance data, as well as a standard set of problem import, export, modification, solution, and query routines. For each supported solver, there is a derived class that implements the methods of the base class, translating the standard calls into native calls to the solver in question. Thus, a code written using only calls from the OSI base class could be easily interfaced with any supported solver without changing any of the code. At the time of this writing, there are eleven commercial and noncommercial solvers with OSI implementations, including several of the solvers reviewed in Section 4.

3.2.2 Object-Oriented Interface

In an object-oriented interface, there is a mapping between the mathematical modeling objects that comprise a MILP instance (variables, constraints, etc.) programming language objects. With this mapping, MILP models can be easily built in a natural way directly within C++ code. The commercial package ILOG Concert Technology [45] was perhaps the first example of such an object-oriented interface, but FLOPC++ [44] is an open source C++ object-oriented interface for algebraic modeling of LPs and MILPs that provides functionality similar to Concert. Using FLOPC++, linear models can be specified in a declarative style, similar to algebraic modeling languages such as GAMS and AMPL, within a C++ program. As a result the traditional strengths of algebraic modeling languages, such as the ability to declare a model in a human-readable format, are preserved, while the user is still able to embed model generation and solution procedures within a larger applications. To achieve solver independence, FLOPC++ uses the OSI to access the underlying solver, and may therefore be linked to any solver with an OSI implementation. Another interesting interface, allowing users to model LP instances in the python language is PuLP [74].

3.3 Solver Frameworks

A solver framework is an implementation of a branch-and-bound, branch-and-cut, or branch-and-price algorithm with hooks that allow the user to provide custom implementations of certain aspects of the algorithm. For instance, the user may wish to provide a custom branching rule or problem-specific valid inequalities. The customization is generally accomplished either through the use of C language callback functions, or through a C++ interface in which the user must derive certain base classes and override default implementations for the desired functions. Not all frameworks are black box solvers. Some frameworks function as black box solvers, but others, such as BCP and ABACUS, do not include default implementations of certain algorithmic components. Table 1 in Section 4 indicates the frameworks available and their style of customization interface.

4 MILP Software

In this section, we summarize the features of the noncommercial software packages available for solving MILPs. Tables 1–3 are a summary of the packages reviewed here. In Table 1, the columns have the following meanings.

- *Version Number*: The version of the software reviewed for this paper. Note that BCP does not assign version numbers.
- *LP Solver*: The LP software used to solve the relaxations arising during the algorithm. The MILP solvers listed as OSI-compliant can use any LP solver with an OSI interface. ABACUS can use either CPLEX, SOPLEX, or XPRESS-MP.
- *File Format*: The file formats accepted by packages containing a black box solver. File formats were discussed in Section 3.1.1.
- *Callable API*: The language in which the callable library interface is implemented (if the package in question has one). Some packages support more than one interface. Two of the solvers can also be called through their own OSI implementation.
- *Framework API*: For those packages that are considered frameworks, this indicates how the callback functions must be implemented—through a C or a C++ interface.
- *User's Manual*: Indicates whether the package has a user's manual.

Table 2 indicates the algorithmic features of each solver, including whether the solver has a pre-processor, whether it can dynamically generate valid inequalities, whether it can perform column generation, whether it includes primal heuristics, what branching strategies are built in, and what search strategies are built in. For the column denoting branching methods, the letters stand for the following methods:

- e: pseudo-cost branching
- f: branching on the variables with the largest fractional part
- h: branching on hyperplanes
- g: GUB branching
- i: branching on first or last fractional variable (by index)

	Version Number	LP Solver	File Format	Callable API	Framework API	User's Manual
ABACUS	2.3	C/S/X	no	none	C++	yes
BCP	11/1/04	OSI	no	none	C++	yes
bonsaiG	2.8	DYLP	MPS	none	none	yes
CBC	0.70	OSI	MPS	C++/C	C++	no
GLPK	4.2	GLPK	MPS/GMPL	OSI/C	none	yes
lp_solve	5.1	lp_solve	MPS/LP/GMPL	C/VB/Java	none	yes
MINTO	3.1	OSI	MPS/AMPL	none	C	yes
SYMPHONY	5.0	OSI	MPS/GMPL	OSI/C	C	yes

Table 1: List of solvers and main features

- p: penalty method
- s: strong branching
- x: SOS(2) branching and branching on semi-continuous variables

For the column denoting search strategies, the codes stand for the following:

- b: best-first
- d: depth-first
- e: best-estimate
- p: best-projection
- r: breadth-first
- h(x,z): a hybrid method switching from strategy 'x' to strategy 'z'
- 2(x,z): a two-phase method switching from strategy 'x' to strategy 'z'

Finally, Table 3 indicates the classes of valid inequalities generated by those solvers that generate valid inequalities. In the following sections, we provide an overview of each solver, then describe the user interface, and finally describe the features of the underlying algorithm in terms of the four categories listed in Section 2.

Solver performance can vary significantly with different parameters settings, and it is unlikely that one set of parameters will work best for all classes of MILP instances. When deciding on a MILP package to use, users are well-advised to consider the ability of a packages to meet their performance requirements through customization and parameter tuning. An additional caveat about performance is that MILP solver performance can be impacted by the speed with which the LP relaxations are solved, so users may need to pay special attention to the parameter tuning of the underlying LP solver as well.

	Preproc	Built-in Cut Generation	Column Generation	Primal Heuristic	Branching Rules	Search Strategy
ABACUS	no	no	yes	no	f,h,s	b,r,d,2(d,b)
BCP	no	no	yes	no	f,h,s	h(d,b)
bonsaiG	no	no	no	no	p	h(d,b)
CBC	yes	yes	no	yes	e,f,g,h,s,x	2(d,p)
GLPK	no	no	no	no	i,p	b,d,p
lp_solve	no	no	no	no	e,f,i,x	d,r,e,2(d,r)
MINTO	yes	yes	yes	yes	e,f,g,p,s	b,d,e,h(d,e)
SYMPHONY	no	yes	yes	no	e,f,h,p,s	b,r,d,h(d,b)

Table 2: Algorithmic features of solvers

Name	Knapsack	GUB	Flow	Clique	Implication	Gomory	MIR
CBC	yes	no	yes	yes	yes	yes	yes
MINTO	yes	yes	yes	yes	yes	no	no
SYMPHONY	yes	no	yes	yes	yes	yes	yes

Table 3: Classes of valid inequalities generated by black box solvers

4.1 ABACUS

4.1.1 Overview

ABACUS [46] is a pure solver framework written in C++. It has a flexible, object-oriented design that supports the implementation of a wide variety of sophisticated and powerful variants of branch and bound. The object-oriented design of the library is similar in concept to BCP, described below. From the user's perspective, the framework is centered around C++ objects representing the basic building blocks of a mathematical model—constraints and variables. The user can dynamically generate variables and valid inequalities by defining classes derived from the library's base classes. ABACUS supports the simultaneous generation of variables and valid inequalities for users requiring this level of sophistication. Another feature of ABACUS worth noting is its very general implementation of *object pools* for storing previously generated constraints and variables for later use.

ABACUS was for some time a commercial code, but has recently been released open source [79] under the GNU Library General Public License (LGPL). Because of the generality of its treatment of dynamically generated classes of constraints and variables, it is one of the most full-featured solver frameworks available. ABACUS does not, however, have a callable library interface and it cannot be used as a black box solver. However, it can be called recursively. It comes with complete documentation and a tutorial that shows how to use the code. Compared to the similar MILP framework BCP, ABACUS has a somewhat cleaner interface, with fewer classes and a more straightforward object-oriented structure. The **target audience** for ABACUS consists of sophisticated users who need a powerful framework for implementing advanced versions of branch and bound, but who do not need a callable library interface.

4.1.2 User Interface

There are four main C++ base classes from which the user may derive problem-specific implementations in order to develop a custom solver. The base classes are the following:

- `ABA_VARIABLE`: The base class for defining problem-specific classes of variables.
- `ABA_CONSTRAINT`: The base class for defining problem-specific classes of constraints.
- `ABA_MASTER`: The base class for storing problem data and initializing the root node.
- `ABA_SUB`: The base class for methods related to the processing of a search tree node.

In addition to defining new template classes of constraints and variables, the latter two C++ classes are used to implement various user callback routines to further customize the algorithm. The methods that can be implemented in these classes are similar to those in other solver frameworks and can be used to customize most aspects of the underlying algorithm.

4.1.3 Algorithm Control

ABACUS does not contain built-in routines for generating valid inequalities, but the user can implement any separation or column generation procedure that is desired in order to improve the **lower bound** in each search tree node. ABACUS does not have a default primal heuristic for improving the **upper bound**, but again, the user can implement one easily. ABACUS has a general notion of **branching** in which one may branch on either a variable or a constraint (hyperplane). Several strategies for selecting a branching variable are provided. In addition, a strong branching capability is also provided, in which case a number of variables or constraints are selected and presolved before the final branching is performed. To select the candidates, a sophisticated mechanism that allows for selection and ranking of candidates using multiple user-defined branching rules is employed. The **search strategies** include depth-first, breadth-first, best-first, and a strategy that switches from depth-first to best-first after the first feasible solution is found.

4.2 BCP

4.2.1 Overview

BCP is a pure solver framework developed by Ladányi. It is a close relative of SYMPHONY, described below. Both frameworks were derived from the earlier COMPSys framework of Ralphs and Ladányi [47, 67]. BCP is implemented in C++ and has a design centered around problem-specific template classes of cuts and variables, like ABACUS, but takes a more "function-oriented" approach that is similar to SYMPHONY. The design is very flexible and supports the implementation of the same variety of sophisticated variants of branch and bound that ABACUS supports, including simultaneous generation of columns and valid inequalities. It is a pure solver framework and does not have a callable library interface. The BCP library provides its own main function, which means that it cannot easily be called recursively or as a subroutine from another code. Nonetheless, it is still one of the most full-featured solver frameworks available, due to the generality with which it handles constraint and variable generation, as well as branching.

Although BCP is not itself a black box solver, two different black box codes [56] have been built using BCP and are available for download along with BCP itself as part of the COIN-OR software suite [48]. BCP is open source software licensed under the Common Public License (CPL). BCP has a user's manual, though it is slightly out of date. However, the code itself contains documentation that can be parsed and formatted using the Doxygen automatic documentation system [82]. A number of applications built using BCP are available for download, including some simple examples that illustrate its use. Tutorials developed by Galati describing the implementation of two problem-specific solvers—one implementing branch and cut and one implementing branch and price—are available for download [34]. The **target audience** for BCP is similar to that of ABACUS—relatively sophisticated users who need a powerful framework for implementing advanced versions of branch and bound without a callable library interface. BCP is also targeted at users who want to solve MILPs in parallel.

4.2.2 User Interface

To use BCP, the user must implement application-specific C++ classes derived from the virtual base classes provided as part of the BCP library. The classes that must be implemented fall broadly into two categories: *modeling object classes*, which describe the variables and constraints associated with the user's application, and *user callback classes*, which control the execution of various specific parts of the algorithm.

From the user's point of view, a subproblem in BCP consists primarily of a *core relaxation* present in every subproblem and *modeling objects*—the sets of extra constraints and variables that augment the core relaxation. To define new template classes of valid inequalities and variables, the user must derive the classes `BCP_cut` and `BCP_var`. The derivation involves defining an abstract data structure for describing a member of the class and providing methods for *expanding* each object, i.e., adding the object to a given LP relaxation.

To enable parallel execution, the internal library and the set of user callback functions are divided along functional lines into five separate computational modules. The modular implementation facilitates code maintenance and allows easy, configurable parallelization. The five modules are *master*, *tree manager*, *node processor*, *cut generator*, and *variable generator*. The master module includes functions that perform problem initialization and input/output. The tree manager is responsible for maintaining the search tree and managing the search process. The node processor is responsible for processing a single search tree node, i.e., producing a bound on the solution to the corresponding subproblem by solving a dynamically generated LP relaxation. Finally, the cut and variable generators are responsible for generating new modeling objects for inclusion in the current LP relaxation.

Associated with each module "`xx`" is a class named `BCP_xx_user` containing the user callbacks for the module. For each module, the user must provide a derived class, overriding those methods for which the user wishes to provide a customized implementation. Note that most, but not all, methods have default implementations. Another important role of the user callback classes is that they can contain data structures for storing the data needed to execute the methods in the class. Such data could include the original input data, problem parameters, and instance specific auxiliary information such as graph data structures.

4.2.3 Algorithm Control

As with ABACUS, BCP does not contain built-in routines for generating valid inequalities, but the user can implement any separation or column generation procedure that is desired in order to improve the **lower bound**. BCP tightens variable bounds by reduced cost and allows the user to tighten bounds based on logical implications arising from the model. BCP does not yet have a built-in integer preprocessor and also has no built-in primal heuristic to improve the **upper bound**. The user can, however, pass an initial upper bound if desired. The default **search strategy** is a hybrid depth-first/best-first approach in which one of the children of the current node is retained for processing as long as the lower bound is not more than a specified percentage higher than the best available. It is also possible for the user to specify a customized search strategy by implementing a new comparison function for sorting the list of candidate nodes in the `BCP_tm_user` class.

BCP has a generalized **branching** mechanism in which the user can specify *branching sets*, consisting of any number of hyperplanes and variables. The hyperplanes and variables in these branching sets do not have to be present in the current subproblem. In other words, it is possible to branch on any arbitrary hyperplane (see Section 2.3), whether or not it corresponds to a known valid inequality. After the desired number of candidate branching sets have been chosen, each one is presolved as usual by performing a specified number of simplex pivots to determine an estimate of the bound improvement resulting from the branching. The final branching candidate can then be chosen by a number of standard built-in rules. The default rule is to select a candidate for which the smallest lower bound among its children is maximized.

4.3 BonsaiG

4.3.1 Overview

BonsaiG is a black box MILP solver available at [41]. BonsaiG comes with complete documentation and descriptions of its algorithms and is available as open source software under the GNU General Public License (GPL). BonsaiG does not have a documented callable library interface or the customization options associated with a solver framework. It does, however, have two unique features worthy of mention. The first is the use of a partial arc consistency algorithm proposed in [78] to help enforce integrality constraints and dynamically tighten bounds on variables. Although the approach is similar to that taken by today's integer programming preprocessors, the arc consistency algorithm can be seen as a constraint programming technique and is applied aggressively for every subproblem. From this perspective, bonsaiG is perhaps one of the earliest examples of integrating constraint programming techniques into an LP-based branch-and-bound algorithm. The integration of constraint programming and traditional mathematical programming techniques has recently become a topic of increased interest among researchers. Achterberg is currently developing a solver called SCIP that will also integrate these two approaches [1].

The second feature worthy of mention is the use of DYLP, an implementation of the dynamic LP algorithm of Padberg [66], as the underlying LP solver. DYLP was designed specifically to be used for solving the LP relaxations arising in LP-based branch and bound. As such, DYLP automatically selects the subsets of the constraints and variables that should be active in a relaxation and manages the process of dynamically updating the active constraints and variables as the problem is solved. This management must be performed by all MILP solvers, but it can be handled more efficiently if kept internal to the LP solver. The **target audience** for bonsaiG consists of users who need a lightweight black box solver capable of solving relatively small MILPs without incurring the overhead associated with advanced bounding techniques and who don't need a callable library interface.

4.3.2 User Interface and Algorithm Control

BonsaiG is a pure black box solver developed by Lou Hafer that can only be called from the command-line. Instances must be specified in MPS format. Algorithm control in bonsaiG is accomplished through the setting of parameters that are specified in a separate file. To improve the **lower bound** for generic MILPs, bonsaiG aggressively applies the arc consistency algorithm discussed earlier in combination with reduced cost tightening of bounds in an iterative loop called the *integrated variable forcing loop*. No generation of valid inequalities or columns is supported. BonsaiG does not have any facility for improving the **upper bound**. The default **search strategy** is a hybrid of depth-first and best-first, but with a slight modification. Upon partitioning of a subproblem, all children are fully processed and among those that are not fathomed, the best one, according to an evaluation function that takes into account both the lower bound and the integer infeasibility, is retained for further partitioning. The others are added to the list of candidates, so that the list is actually one of candidates for branching, rather than for processing. When all children of the current node can be fathomed, then the candidate with the best bound is retrieved from the list and another depth-first search is initiated.

For **branching**, BonsaiG uses a penalty method strengthened by integrality considerations. Only branching on variables or groups of variables is supported. The user can influence branching decisions for a particular instance or develop custom branching strategies through two different mechanisms. First, bonsaiG makes it easy to specify relative branching priorities for groups of variables. This tells the solver which variables the user thinks will have the most effect if branched upon. The solver then attempts to branch on the highest-priority fractional variables first. The second mechanism is for specifying *tours* of variables. These are groups of variables that should be branched on as a whole. The group of child subproblems (called a *tour group*) is generated by adjusting the bounds of each variable in the tour so that the feasible region of the parent is contained in the union of the feasible regions of the children, as usual.

4.4 CBC

4.4.1 Overview

CBC is a black box solver distributed as part of the COIN-OR software suite [30]. CBC was originally developed by John Forrest as a lightweight branch-and-cut code to test CLP, the COIN-OR LP solver. However, CBC has since evolved significantly and is now quite sophisticated, even sporting customization features that allow it to be considered a solver framework. CBC has a native C++ callable library API similar to the Open Solver Interface, as well as a C interface built on top of that native interface. The CBC solver framework consists of a collection of C++ classes whose methods can be overridden to customize the algorithm. CBC does not have a user's manual, but it does come with some well-commented examples and the source code is also well-commented. It is distributed as part of the COIN-OR software suite and is licensed as open source software under the Common Public License (CPL). The **target audience** for CBC consists of users who need a full-featured black box solver with a callable library API and very flexible, yet relatively lightweight, customization options.

4.4.2 User Interface

The user interacts with CBC as a black box solver either by invoking the solver on the command line or through a command-based interactive shell. In either case, instances must be specified in MPS format. The callable library API is a hybrid of the API of the underlying LP solver, which is accessed through the Open Solver Interface, and the methods in the `CbcModel` class. To load a model into CBC, the user creates an OSI object, loads the model into the OSI object and then passes a pointer to that object to the constructor for the `CbcModel` object. CBC uses the OSI object as its LP solver during the algorithm.

To use CBC as a solver framework, there are a number of classes that one can reimplement in order to arrive at problem-specific versions of the basic algorithm. The main classes in the CBC library are:

- `CbcObject`, `CbcBranchingObject`, `CbcBranchDecision`: The classes used to specify new branching rules. CBC has a very general notion of branching that is similar to that of BCP. CBC's branching mechanism is described in more detail in Section 4.4.3.
- `CbcCompare`, `CbcCompareBase`: The classes used to specify new search strategies by specifying the method for sorting the list of candidate search tree nodes.
- `CbcHeuristic`: The class used to specify new primal heuristics.
- `CbcCutGenerator`: The class that interfaces to the Cut Generation Library.

As seen from the above list, it is possible to introduce custom branching rules, custom search strategies, custom primal heuristics, and custom generators for valid inequalities. Cut generator objects must be derived from the Cut Generation Library base class and are created by the user before being passed to CBC. Primal heuristic objects are derived from the `CbcHeuristic` class and are also created before being passed to CBC.

4.4.3 Algorithm Control

CBC is one of the most full-featured black box solver of those reviewed here in terms of available techniques for improving bounds. The **lower bound** can be improved through the generation of valid inequalities using all of the separation algorithms implemented in the CGL. Problem-specific methods for generation of valid inequalities can be implemented, but column generation is not supported. CBC has a logical preprocessor to improve the initial model and tightens variable bounds using reduced cost information. Several primal heuristics to improve the **upper bound** are implemented and provided as part of the distribution, including a rounding heuristic and two different local search heuristics. The default **search strategy** in CBC is to perform depth-first search until the first feasible solution is found and then to select nodes for evaluation based on a combination of bound and number of unsatisfied integer variables. Specifying new search strategies can also be done easily.

CBC has a strong **branching** mechanism similar to that of other solvers, but the type of branching that can be done is more general. An abstract CBC branching object can be anything that has a feasible region whose degree of infeasibility with respect to the current solution can be quantified, that has an associated action that can be taken to improve the degree of infeasibility in the child nodes, and that supports some comparison of the effect of branching. Specifying a CBC branching object involves implementing three methods: `infeasibility()`, `feasibleRegion()`, and `createBranch()`. Using these methods CBC can perform strong branching on any sort of branching objects. Default implementations are provided for branching on integer variables, branching on cliques, and branching on special ordered sets.

4.5 GLPK

4.5.1 Overview

GLPK is the GNU Linear Programming Kit, a set of subroutines comprising a callable library and black box solver for solving linear programming and MILP instances [53]. GLPK also comes equipped with GNU MathProg (GMPL), an algebraic modeling language similar to AMPL. GLPK was developed by Andrew Makhorin and is distributed as part of the GNU Project, under the GNU General Public License (GPL). Because GLPK is distributed through the Free Software Foundation (FSF), it closely follows the guidelines of the FSF with respect to documentation and automatic build tools. The build system relies on autoconf, which ensures that users can easily build the library and executable on a wide variety of platforms. The documentation consists of the Reference Manual and the description of the GNU MathProg language. The distribution includes examples of using the callable library and models demonstrating the MathProg language.

GLPK is a completely self-contained package—it does not rely on external components to perform any part of the branch-and-bound algorithm. In particular, GLPK includes the following main components:

- revised primal and dual simplex methods for linear programming,
- a primal-dual interior point method for linear programming,
- a branch-and-bound method,
- a parser for GNU MathProg,
- an application program interface (API),
- a black box LP/MILP solver.

The **target audience** for GLPK consists of users who want a lightweight, self-contained MILP solver with both a callable library and modeling language interface.

4.5.2 User Interface

The default build of GLPK yields the callable library and a black box solver. The callable library consists of nearly 100 routines for loading and modifying a problem instance, solving the loaded instance, querying the solver, and getting and setting algorithm parameters. There are also utility routines to read and write files in MPS format, LP format, and GMPL format. The subroutines operate on a data structure for storing the problem instance that is passed explicitly, so the code should be thread safe and it is possible to work on multiple models simultaneously.

4.5.3 Algorithm Control

Since GLPK was first and foremost developed as a solver of linear programs, it does not yet contain advanced techniques for improving the **lower bound**, such as preprocessing techniques and procedures for generating valid inequalities. It also does not include a primal heuristic for improving the **upper bound**. The user can set a parameter (either through the callable library or in the black box solver) to choose from one of three methods for selecting a **branching** variable—the index of the first fractional variable, the index of the last fractional variable, or the penalty method discussed in Section 2.3. The user can also change the **search strategy** to either depth-first-search, breadth-first-search, or the best-projection method described in Section 2.4.

4.6 lp_solve

4.6.1 Overview

Lp_solve is a black box solver and callable library for linear and mixed-integer programming. The original solver was developed by Michel Berkelaar at Eindhoven University of Technology, and the work continued with Jeroen Dirks, who contributed a procedural interface, a built-in MPS reader, and fixes and enhancements to the code. Kjell Eikland and Peter Notebaert took over development starting with version 4, and there is an active group of users. The most recent version bears little resemblance to earlier versions and includes a number of unique features such as a modular LP basis factorization engine and a large number of language interfaces. Lp_solve is distributed as open source under the GNU Library General Public License (LGPL). The main repository for lp_solve information, including a FAQ, examples, the full source, precompiled executables, and a message board, is at the YAHOO lp_solve group [15]. The **target audience** for lp_solve is similar to that of GLPK—users who want a lightweight, self-contained solver with a callable library API implemented in a number of popular programming languages, including C, VB and Java, as well as an AMPL interface.

4.6.2 User Interface

Lp_solve can be used as a black box solver or as a callable library through its native C API. The lp_solve API consists of over 200 functions for reading and writing problem instances, building or querying problem instances, setting algorithm parameters, invoking solution algorithms, and querying the results. The lp_solve API has methods that can read and write MPS files, LP format files, and an XLI (External Language Interface) that allows users to implement their own readers and writers. At present, XLI interfaces are in place for GNU MathProg, LPFML, and for the commercial LP formats of CPLEX and LINDO. An interface to ZIMPL will be available in lp_solve v5.2.

Of the noncommercial MILP software reviewed here, lp_solve has interfaces to the largest number of different programming languages. With lp_solve, there are examples that illustrate how to call its API from within a VB.NET or C# program. Also, there exists a Delphi library and a Java Native Interface (JNI) to lp_solve, so lp_solve can be called directly from Java programs. There are also AMPL, MATLAB, R, S-Plus and Excel driver programs. Lp_solve supports several types of user callbacks and an object-like facility for revealing functionality to external programs.

4.6.3 Algorithm Control

Lp_solve does not have any special procedures for improving the **upper bounds** or **lower bounds**. Users can set parameters either from the command line or through the API to control the branch-and-bound procedure. The **search strategy** is one of depth-first search, breadth-first search, or a two-phase method that initially proceeds depth-first followed by breadth-first. Lp_solve contains a large number of built-in **branching** procedures and can select the branching variable based on the lowest indexed non-integer column (default), the distance from the current bounds, the largest current bound, the most fractional value, a simple, unweighted pseudocost of a variable, a pseudocost strategy for minimizing the number of integer infeasibilities, or an extended pseudocost strategy based on maximizing the normal pseudocost divided by the number of infeasibilities.

The algorithm also allows for GUB branching and for branching on semi-continuous variables (variables that have to take a value of zero or a positive value above some given lower bound). The branching rule and search strategy used by lp_solve is set through a call to `set_bb_rule()`, and there are even ways in which the branching rule can be modified using the parameter values beyond what are listed here. MILP performance can be expected to vary significantly based on parameters settings and model class. The default settings in lp_solve are inherited from v3.2, and tuning is therefore necessary to achieve desirable results. The developers have indicated that MILP performance and more robust general settings will be a focus in lp_solve v5.3.

4.7 MINTO

4.7.1 Overview

MINTO (Mixed INTeger Optimizer) is a black box solver and solver framework for MILP. The chief architects of MINTO were George Nemhauser and Martin Savelsbergh, and a majority of the software development was done by Savelsbergh. MINTO was developed at the Georgia Institute of Technology and is available under terms of an agreement created by the Georgia Tech Research Institute. The current maintainer of MINTO is Jeff Linderoth of Lehigh University. MINTO is available only in library form for a number of platforms [60]. MINTO relies on external software to solve the linear programming relaxations that arise during the algorithm. Since version 3.1, MINTO has been equipped to use the OSI, so any of the linear programming solvers for which there is an OSI interface can be used with MINTO. MINTO can also be built to directly use the commercial LP solvers CPLEX, XPRESS-MP, or OSL. MINTO comes with a user's manual that contains instructions for building an executable, and a description of the API for user callbacks that allow it to be used as a solver framework, along with examples of using each routine. The **target audience** for MINTO are users who require the power of a sophisticated solver framework for implementing advanced version of branch and bound, but with a relatively simple C-style interface, or who need a full-featured black box solver without a callable API.

4.7.2 User Interface

MINTO can be accessed as a black-box solver from the command line with parameters through a number of command-line switches. The most common interface to MINTO is to pass problem instances in MPS file format. However, beginning with version 3.1, MINTO can also be used directly with the AMPL modeling language. MINTO can be customized through the use of "user application" functions (callback functions) that allow MINTO to operate as a solver framework. At well-defined points of the branch-and-cut algorithm, MINTO will call the user application functions, and the user must return a value that signals to MINTO whether or not the algorithm is to be modified from the default. For example, consider the MINTO user application function `appl_constraints()`, which is used for generating user-defined classes of valid inequalities. The input to `appl_constraints()` is the solution to the current LP relaxation. The outputs are arrays describing any valid inequalities that the user wishes to have added to the formulation. The return value from `appl_constraint()` should be `SUCCESS` or `FAILURE`, depending on whether or not the user-supplied routine was able to find inequalities violated by the input solution. If so, MINTO will add these inequalities and pass the new formulation to the linear programming solver.

4.7.3 Algorithm Control

To strengthen the **lower bound** during the course of the algorithm, MINTO relies on advanced preprocessing and probing techniques, as detailed in the paper of Atamtürk, Nemhauser, and Savelsbergh [5], and tightens bounds based on reduced costs. MINTO also has separation routines for a number of classes of valid inequalities, including clique inequalities, implication inequalities, knapsack cover inequalities, GUB cover inequalities, and flow cover inequalities. The user can perform dynamic column generation by implementing the callback function `appl_variables()`. For improving the **upper bound**, MINTO has a primal diving heuristic. There are a number of built-in **branching** methods, such as branching on the most fractional variable, the penalty method strengthened with integrality considerations, strong branching, a pseudocost-based method, a dynamic method that combines both the penalty method and pseudocost-based branching, and a method that favors branching on GUB constraints. For **search strategies**, the user can choose best-bound, depth-first, best-projection, best-estimate, or an adaptive mode that combines depth-first with the best-estimate mode. Of course, by using the solver framework, any of the branching or node selection methods can be overridden by the user.

4.8 SYMPHONY

4.8.1 Overview

SYMPHONY is a black box solver, callable library, and solver framework for MILPs that evolved from the COMPSys framework of Ralphs and Ladányi [47, 67]. The source code for packaged releases, with full documentation and examples, is available for download [68] and is licensed under the Common Public License (CPL). The latest source is also available from the CVS repository of the COIN-OR Foundation [19]. SYMPHONY is fully documented and seven different specialized solvers built with SYMPHONY are available for download as examples of how to use the code. There is a step by step example showing how to build a simple branch-and-cut solver in SYMPHONY for the matching problem [81], which is summarized in Section 5. The core solution methodology of SYMPHONY is a customizable branch, cut, and price algorithm that can be executed sequentially or in parallel [73]. SYMPHONY calls on several other open source libraries for specific functionality, including COIN-OR's Cut Generation Library, Open Solver Interface, and MPS file parser components, GLPK's GMPL file parser, and a third-party solver for linear-programming problems (LPs), such as COIN-OR's LP Solver (CLP).

There are several unique features of SYMPHONY that are worthy of mention. First, SYMPHONY contains a generic implementation of the WCN algorithm described in [72] for solving bicriteria MILPs, and methods for approximating the set of Pareto outcomes. The bicriteria solver can be used to examine tradeoffs between competing objectives, and for solving parametric MILPS, a form of global sensitivity analysis. SYMPHONY also contains functions for local sensitivity analysis based on ideas suggested by Schrage and Wolsey [76]. Second, SYMPHONY has the capability to warm start the branch-and-bound process from a previously calculated branch-and-bound tree, even after modifying the problem data. These capabilities are described in more detail in the paper of Ralphs and Guzelsoy [71]. The **target audience** for SYMPHONY is similar to that of MINTO—users who require the power of a sophisticated solver framework, primarily for implementing custom branch and cut algorithms, with a relatively simple C-style interface, or users who require other advanced features such as parallelism, the ability to solve multi-criteria instances, or the ability to warm start solution procedures.

4.8.2 User Interface

As a black box solver, SYMPHONY can read GMPL files using an interface to GLPK's file parser or MPS files using COIN-OR's MPS file reader class. It can also be used as a callable library through the API described below. SYMPHONY's callable library consists of a complete set of subroutines for loading and modifying problem data, setting parameters, and invoking solution algorithms. The user invokes these subroutines through the native C API, which is exactly the same whether invoking SYMPHONY sequentially or in parallel. The choice between sequential and parallel execution modes is made at compile-time. SYMPHONY has an OSI implementation that allows solvers built with SYMPHONY to be accessed through the OSI.

The user's main avenues for customization are the tuning of parameters and the implementation of SYMPHONY's callback functions. SYMPHONY contains over 50 callback functions allowing the user to override SYMPHONY's default behavior for branching, generation of valid inequalities, management of the cut pool, management of the LP relaxation, search and diving strategies, program output, etc. Each callback function is called from a SYMPHONY *wrapper function* that interprets the user's return value and determines what action should be taken. If the user performs the required function, the wrapper function exits without further action. If the user requests that SYMPHONY perform a certain default action, then this is done. Files containing default function stubs for the callbacks are provided along with the SYMPHONY source code. These can then be modified by the user as desired. Makefiles and Microsoft Visual C++ project files are provided for automatic compilation. A full list of callback functions is contained in the user's manual [69]. For an example of the use of callbacks, see the SYMPHONY case study in Section 5.1.

4.8.3 Algorithm Control

To improve the **lower bound** for generic MILPs, SYMPHONY generates valid inequalities using COIN-OR's Cut Generation Library (CGL) described in Section 2.1.2. The user can easily insert custom separation routines and can perform column generation, though the implementation is not yet fully general and requires that the set of variables be indexed a priori. This limitation makes the column generation in SYMPHONY most appropriate for situations in which the set of columns has a known combinatorial structure and is of relatively small cardinality. In each iteration, SYMPHONY tightens bounds by reduced cost and also allows the user to tighten bounds based on logical implications arising from the model. SYMPHONY does not yet have its own logical preprocessor or primal heuristics to improve the **upper bound**, though it is capable of using CBC's primal heuristic if desired. The user can also pass an initial upper bound.

SYMPHONY uses a strong **branching** approach by default. Branching candidates can be either constraints or variables and are chosen by any one of a number of built-in rules, such as most fractional, or by a customized rule. After the candidates are chosen, each one is presolved to determine an estimate of the bound improvement resulting from the branching. The final branching candidate can then be chosen by a number of standard built-in rules. There is also a naive version of pseudo-cost branching available.

The default **search strategy** is a hybrid depth-first/best-first approach in which one of the children of the current node is retained for processing as long as the lower bound is not more than a specified percentage higher than the best available. Another option is to stop diving when the current node is more than a specified percentage of the gap higher than the best available. By tuning various parameters, one can obtain a number of different search strategies running the gamut between depth-first and best-first.

```
typedef struct MATCH_DATA{
   int              numnodes;
   int              cost[MAXNODES][MAXNODES];
   int              endpoint1[MAXNODES*(MAXNODES-1)/2];
   int              endpoint2[MAXNODES*(MAXNODES-1)/2];
   int              index[MAXNODES][MAXNODES];
}match_data;
```

Figure 1: Data structure for matching solver

5 Case Studies

In this section, we describe two examples that illustrate the power of solver frameworks for developing custom optimization codes. The first is a custom branch-and-cut algorithm for solving the matching problem developed using SYMPHONY. The second is a custom branch-and-price algorithm for the axial assignment problem developed using BCP. Full source code and more detailed descriptions of both solvers are available [34, 81].

5.1 Branch and Cut

The Matching Problem. Given a complete, undirected graph $G = (V, E)$, the *Matching Problem* is that of selecting a set of pairwise disjoint edges of minimum weight. The problem can be formulated as follows:

$$\min \sum_{e \in E} c_e x_e$$

$$\begin{aligned}
\sum_{j \in V: e=\{i,j\} \in E} x_e &= 1 \qquad \forall i \in V, && (12)\\
x_e &\geq 0 \qquad \forall e \in E, && (13)\\
x_e &\in \mathbb{Z} \qquad \forall e \in E,
\end{aligned}$$

where x_e is a binary variable that takes value 1 if edge e is selected and 0 otherwise.

Implementing the Solver. The first thing needed is a data structure to store the description of the problem instance and any other auxiliary information. Such a data structure is shown in Figure 1. We assume a complete graph, so a problem instance can be described simply by the objective function coefficients, stored in the two-dimensional array `cost`. Each primal variable is identified by an index, so we must have a way of mapping the edge $\{i, j\}$ to the index that identifies the corresponding variable and vice versa. Such mappings between problem instance objects and variable indices are a common construct when using solver frameworks. Recent commercial modeling frameworks such as Concert [45] and Mosel [20] and the noncommercial modeling system FLOPC++ [44] have an interface that allows for a more explicit coupling of problem objects and instance variables. In the data structure shown, `endpoint1[k]` returns the first endpoint of the edge with index `k` and `endpoint2[k]` returns the second endpoint. On the other hand `index[i][j]` returns the index of edge $\{i, j\}$.

Next, functions for reading in the problem data and creating the instance are needed. The function `match_read_data()` (not shown) reads the problem instance data (a list of objective function coefficients) in from a file.

The function `match_load_problem()`, shown in Figure 2, constructs an integer program in column-ordered format. In the first part of this routine, a description of the MILP is built, while in the second part, this representation is loaded to SYMPHONY through the subroutine `sym_explicit_load_problem()`.

The `main()` routine is shown in Figure 3. In this routine, a SYMPHONY environment and a user data structure are created, the data is read in, the MILP is created and loaded into SYMPHONY and then solved. Results are automatically printed, but we could also implement a custom subroutine for displaying these if desired.

We next show how to add the ability to generate some simple problem-specific valid inequalities. The odd-set inequalities

$$\sum_{e \in E(O)} x_e \leq \frac{|O|-1}{2} \qquad O \subseteq V,\ |O| \text{ odd}, \tag{14}$$

with $E(O) = \{e = \{i, j\} \in E \mid i \in O,\ j \in O\}$ are satisfied by all matchings. Indeed, Edmonds [28] showed that the inequalities (12), (13), and (14) define the convex hull of matchings, so the matching problem can be solved as a linear program, albeit with an exponential number of constraints.

The textbook of Grötschel, Lovász, and Schrijver [37] describes how to efficiently separate the odd-set inequalities in full generality, but for simplicity, we show how to implement separation for odd-set inequalities for sets of size three. We do this by brute force enumeration of triples, as shown in Figure 4. The function `user_find_cuts()` is the SYMPHONY callback for generating valid inequalities. The user is provided with the current fractional solution in a sparse vector format and asked to generate violated valid inequalities. The call to `cg_add_explicit_cut()`, informs SYMPHONY of any inequalities found. Even this simple separation routine can significantly reduce the number of nodes in the branch-and-cut tree.

5.2 Branch and Price

The Three-Index Assignment Problem. The *Three-Index Assignment Problem* (3AP) is that of finding a minimum-weight clique cover of the complete tri-partite graph $K_{n,n,n}$, where n is redefined here to indicate the size of the underlying graph. Let I, J and K be three disjoint sets

```
int match_load_problem(sym_environment *env, match_data *prob){
   int i, j, index, n, m, nz, *column_starts, *matrix_indices;
   double *matrix_values, *lb, *ub, *obj, *rhs, *rngval;
   char *sense, *is_int;

   n = prob->numnodes*(prob->numnodes-1)/2; /* Number of columns */
   m = 2 * prob->numnodes;                  /* Number of rows */
   nz = 2 * n;                              /* Number of nonzeros */
   /* Normally, allocate memory for the arrays here (left out to save space) */
   for (index = 0, i = 0; i < prob->numnodes; i++) {
      for (j = i+1; j < prob->numnodes; j++) {
         prob->match1[index] = i; /*The 1st component of assignment 'index'*/
         prob->match2[index] = j; /*The 2nd component of assignment 'index'*/
         prob->index[i][j] = prob->index[j][i] = index; /*To recover later*/
         obj[index] = prob->cost[i][j]; /* Cost of assignment (i, j) */
         is_int[index] = TRUE; /* Indicates the variable is integer */
         column_starts[index] = 2*index;
         matrix_indices[2*index] = i;
         matrix_indices[2*index+1] = j;
         matrix_values[2*index] = 1;
         matrix_values[2*index+1] = 1;
         ub[index] = 1.0;
         index++;
      }
   }
   column_starts[n] = 2 * n; /* We have to set the ending position */
   for (i = 0; i < m; i++) { /* Set the right-hand side */
      rhs[i] = 1;
      sense[i] = 'E';
   }
   sym_explicit_load_problem(env, n, m, column_starts, matrix_indices,
                             matrix_values, lb, ub, is_int, obj, 0, sense, rhs,
                             rngval, true);
   return (FUNCTION_TERMINATED_NORMALLY);
}
```

Figure 2: Function to load the problem for matching solver

```
int main(int argc, char **argv)
{
   int termcode;
   char * infile;

   /* Create a SYMPHONY environment */
   sym_environment *env = sym_open_environment();

   /* Create the data structure for storing the problem instance.*/
   user_problem *prob = (user_problem *)calloc(1, sizeof(user_problem));

   sym_set_user_data(env, (void *)prob);
   sym_parse_command_line(env, argc, argv);
   sym_get_str_param(env, "infile_name", &infile);
   match_read_data(prob, infile);
   match_load_problem(env, prob);
   sym_solve(env);
   sym_close_environment(env);
   return(0);
}
```

Figure 3: Main function for the matching solver

with $|I| = |J| = |K| = n$ and set $H = I \times J \times K$. Then, 3AP can be formulated as follows:

$$\min \sum_{(i,j,k)\in H} c_{ijk} x_{ijk},$$

$$\sum_{(j,k)\in J\times K} x_{ijk} = 1 \qquad \forall i \in I, \tag{15}$$

$$\sum_{(i,k)\in I\times K} x_{ijk} = 1 \qquad \forall j \in J,$$

$$\sum_{(i,j)\in I\times J} x_{ijk} = 1 \qquad \forall k \in K, \tag{16}$$

$$x_{ijk} \in \{0,1\} \ \forall (i,j,k), \in H.$$

In [10], Balas and Saltzman consider the use of the classical Assignment Problem (AP) as a relaxation of 3AP in the context of Lagrangian relaxation. We use the same relaxation to

```
int user_find_cuts(void *user, int varnum, int iter_num, int level,
                   int index, double objval, int *indices, double *values,
                   double ub, double etol, int *num_cuts, int *alloc_cuts,
                   cut_data ***cuts)
{
   user_problem *prob = (user_problem *) user;
   double cut_val[3], edge_val[200][200]; /* Matrix of edge values */
   int i, j, k, cutind[3], cutnum = 0;

   /* Allocate the edge_val matrix to zero (we could also just calloc it) */
   memset((char *)edge_val, 0, 200*200*ISIZE);

   for (i = 0; i < varnum; i++) {
      edge_val[prob->node1[indices[i]]][prob->node2[indices[i]]] = values[i];
   }
   for (i = 0; i < prob->nnodes; i++){
      for (j = i+1; j < prob->nnodes; j++){
         for (k = j+1; k < prob->nnodes; k++) {
            if (edge_val[i][j]+edge_val[j][k]+edge_val[i][k] > 1.0 + etol) {
               /* Found violated triangle cut */
               /* Form the cut as a sparse vector */
               cutind[0] = prob->index[i][j];
               cutind[1] = prob->index[j][k];
               cutind[2] = prob->index[i][k];
               cutval[0] = cutval[1] = cutval[2] = 1.0;
               cg_add_explicit_cut(3, cutind, cutval, 1.0, 0, 'L',
                                   TRUE, num_cuts, alloc_cuts, cuts);
               cutnum++;
            }
         }
      }
   }
   return(USER_SUCCESS);
}
```

Figure 4: Cut generator for matching solver

reformulate the 3AP using a Dantzig-Wolfe decomposition (see Section 2.1.4 for a discussion of this technique). The AP is a relaxation of the 3AP obtained by relaxing constraint (15). Let $\mathcal{F}$ be the set of feasible solutions to the AP. The Dantzig Wolfe (DW) reformulation of 3AP is then:

$$\begin{aligned}
\min \quad & \sum_{s\in\mathcal{F}} c_s\lambda_s, \\
\sum_{s\in\mathcal{F}} \left(\sum_{(j,k)\in J\times K} s_{ijk}\lambda_s \right) &= 1 \quad \forall i \in I, \\
\sum_{s\in\mathcal{F}} \lambda_s &= 1, \\
\lambda_s &\in \mathbb{Z}_+ \ \forall s \in \mathcal{F},
\end{aligned}$$

where $c_s = \sum_{(i,j,k)\in H} c_{ijk}s_{ijk}$ for each $s \in \mathcal{F}$. Relaxing the integrality constraints of this reformulation, we obtain a relaxation of 3AP suitable for use in an LP-based branch-and-bound algorithm. Since there are an exponential number of columns in this linear program, we use a standard column generation approach to solve it.

Implementing the Solver. The main classes to be implemented are the `BCP_xx_user` classes mentioned earlier and a few problem-specific classes. We describe the problem-specific classes first.

- `AAP`: This class is a container for holding an instance of 3AP. Data members include the dimension of the problem n and the objective function vector.
- `AAP_user_data`: This class is derived from `BCP_user_data` and is used to store problem-specific information in the individual nodes of the search tree. Since we branch on the original variables x_{ijk} and not the master problem variables λ_s, we need to keep track of which variables have been fixed to 0 or 1 at a particular node, so that we can enforce these conditions in our column generation subproblem.
- `AAP_var`: Each variable in the Dantzig-Wolfe reformulation represents an assignment between members of sets J and K. In each instance of the class, the corresponding assignment is stored using a vector containing the indices of the assignment along with its cost.

Note that because `BCP` is a parallel code, every data structure must be accompanied by a subroutine that can both pack it into and unpack it from a character buffer for the purposes of communicating the contents to other parallel processes. For most built-in types, default pack and unpack routines are predefined. For user-defined data structures, however, they must be provided. Typically, such routines consist simply of a collection of calls to either `BCP_buffer::pack()` or

`BCP_buffer::unpack()`, as appropriate, packing or unpacking each data member of the class in turn. The user callback classes that must be modified are as follows.

- `AAP_tm`: Derived from `BCP_tm_user`, this class contains the callbacks associated with initialization and tree management. The main callbacks implemented are `initialize_core()` and `create_root()`. These methods define the core relaxation and the initial LP relaxation in the root node.

- `AAP_lp`: Derived from `BCP_lp_user`, this class contains the callbacks associated with the solution of the LP relaxations in each search tree node. The main methods implemented are

  `generate_vars_in_lp()`: the subroutine that generates new variables,

  `compute_lower_bound()`: returns a true lower bound in each iteration (the LP relaxation does not yield a true lower bound unless there exist no variables with negative reduced cost),

  `restore_feasibility()`: a subroutine that tries to generate columns that can be used to restore the feasibility of a relaxation that is currently infeasible,

  `vars_to_cols()`: a subroutine that generates the columns corresponding to a set of variables, so that they can be added to the current relaxation,

  `select_branching_candidates()`: a subroutine that selects candidates for strong branching. We branch on the original variables x_{ijk}. Candidates are selected by the usual "most fractional" rule using the helper function `branch_close_to_half()`. A second function, `append_branching_vars()`, is called to create the branching objects, and

  `set_user_data_for_children()`: Stores the appropriate data regarding what original variables were branched on in each child node.

In addition to defining these classes, there are a few important parameters to be set. We have glossed over a few details here, but the source code and a more thorough description of this example are available for download [34].

6 Benchmarking

In this section, we present computational results showing the relative performance of the black box solvers reviewed in Section 4. Each solver was built with gcc 3.3 using the default build parameters. The experiments were run under Linux RedHat v7.3 on an Intel Pentium III with an 1133MHz clock speed and 512MB of memory. The maximum CPU time allowed for each solver and each instance was two hours. For each of the codes, the default parameter settings were used on the

Name	m	n	$n-p$	$\|B\|$	$p-\|B\|$	Name	m	n	$n-p$	$\|B\|$	$p-\|B\|$
22433	198	429	198	231	0	23588	137	368	137	231	0
aligninq	340	1831	1	1830	0	bc1	1913	1751	1499	252	0
bienst1	576	505	477	28	0	bienst2	576	505	470	35	0
dano3_3	3202	13873	13804	69	0	dano3_4	3202	13873	13781	92	0
dano3_5	3202	13873	13758	115	0	fiball	3707	34219	1	33960	258
mcsched	2107	1747	2	1731	14	mkc1	3411	5325	2238	3087	0
neos10	46793	23489	0	23484	5	neos11	2706	1220	320	900	0
neos12	8317	3983	847	3136	0	neos13	20852	1827	12	1815	0
neos14	552	792	656	136	0	neos15	552	792	632	160	0
neos16	1018	377	0	336	41	neos17	486	535	235	300	0
neos18	11402	3312	0	3312	0	neos1	5020	2112	0	2112	0
neos20	2446	1165	198	937	30	neos2	1103	2101	1061	1040	0
neos3	1442	2747	1387	1360	0	neos4	38577	22884	5712	17172	0
neos5	63	63	10	53	0	neos6	1036	8786	446	8340	0
neos7	1994	1556	1102	434	20	neos8	46324	23228	0	23224	4
neos9	31600	81408	79309	2099	0	npmv07	76342	220686	218806	1880	0
nsa	1297	388	352	36	0	nug08	912	1632	0	1632	0
pg5_34	225	2600	2500	100	0	pg	125	2700	2600	100	0
qap10	1820	4150	0	4150	0	ramos3	2187	2187	0	2187	0
ran14x18_1	284	504	252	252	0	roy	162	149	99	50	0
sp97ic	2086	1662	0	718	944	sp98ar	4680	5478	0	2796	2682
sp98ic	2311	2508	0	1139	1369	sp98ir	1531	1680	0	871	809
Test3	50680	72215	39072	7174	25969						

Table 4: Characteristics of (non MIPLIB) problem instances

instances, the sole exception being lp_solve, in which the default branching and node selection rule was changed to a pseudocost-based rule[2].

There were 122 problem instances included in the test: the instances of miplib3[3][16], miplib2003 [57], and 45 instances collected by the authors from various sources. The instances collected by the authors are available from the Computational Optimization Research at Lehigh (COR@L) Web site [49]. Table 4 shows the number of rows m, number of variables n, number of continuous variables $n-p$, number of binary variables $|B|$, and number of general integer variables $p-|B|$ for each of the instances of the test suite that are not already available through MIPLIB.

In order to succinctly present the results of this extensive computational study, we use performance profiles, as introduced by Dolan and Moré [24]. A performance profile is a relative measure of the effectiveness of a solver s when compared to a group of solvers $\mathcal{S}$ on a set of problem instances P. To completely specify a performance profile, we need the following definitions:

- γ_{ps} is a quality measure of solver s when solving problem p,
- $r_{ps} = \gamma_{ps}/(\min_{s\in\mathcal{S}} \gamma_{ps})$, and

[2]The lp_solve command line was: `lp_solve -mps name.mps -bfp ./bfp_LUSOL -timeout 7200 -time -presolve -presolvel -piva -pivla -piv2 -ca -B5 -Bd -Bg -si -s5 -se -degen -S1 -v4.`

[3]Save for the simple instances air03, blend2, egout, enigma, flugpl, gen, khb05250 lseu misc03 misc06, mod008, mod010, p0033, p0201, p0282, rgn, stein27, vpm1, which at least 5 of the six solvers were able to solve in less than 2 minutes.

- $\rho_s(\tau) = |\{p \in P \mid r_{ps} \leq \tau\}| / |P|$.

Hence, $\rho_s(\tau)$ is the fraction of instances for which the performance of solver s was within a factor of τ of the best. A performance profile for solver s is the graph of $\rho_s(\tau)$. In general, the "higher" the graph of a solver, the better the relative performance.

Comparing MILP solvers directly on the basis of performance is problematic in a number of ways. By compiling these codes with the same compiler on the same platform and running them under identical conditions, we have eliminated some of the usual confounding variables, but some remain. An important consideration is the feasibility, optimality, and integrality tolerances used by the solver. Dolan, Moré, and Munson [25] performed a careful study of the tolerances used in nonlinear programming software and concluded that trends of the performance profile tend to remain the same when tolerances are varied. The differences in solver tolerances for these tests were relatively minor, but it is possible that these minor difference could lead to large differences in runtime performance. Another potential difficulty is the verification of solvers' claims with respect to optimality and feasibility of solutions. The authors made little attempt to verify *a posteriori* that the solutions claimed as optimal or feasible were indeed optimal or feasible. The conclusions drawn here about the relative effectiveness of the MILP solvers must be considered with these caveats in mind.

For instances that were solved to provable optimality by one of the six solvers, the solution time was used as the quality measure γ_{ps}. Under this measure, $\rho_s(1)$ is the fraction of instances for which solver s was the fastest solver, and $\rho_s(\infty)$ is the fraction of instances for which solver s found a provably optimal solution. Figure 6 shows a performance profile of the instances that were solved in two CPU hours by at least one of the solvers. The graph shows that bonsaiG and MINTO were able to solve the largest fraction of the instances the fastest. The solvers MINTO and CBC were able to find a provably optimal solution within the time limit for the largest largest fraction of instances, most likely because these two solvers contain the largest array of specialized MILP solution techniques.

For instances that were not solved to optimality by any of the six solvers in the study, we used the value of the best solution found as the quality measure. Under this measure, $\rho_s(1)$ is the fraction of instances for which solver s found the best solution among all the solvers, and $\rho_s(\infty)$ is the fraction of instances for which solver s found at least one feasible solution. In Figure 6, we give the performance profile of the six MILP solvers on the instances for which no solver was able to prove the optimality of the solution. SYMPHONY was able to obtain the largest percentage of good feasible solutions, and the performance of GLPK was also laudable in this regard. This is a somewhat surprising conclusion in that neither SYMPHONY nor GLPK contain a specialized primal heuristic designed for finding feasible solutions. This seems to indicate that the primal heuristics existing in these noncommercial codes are relatively ineffective. Implementation of a state-of-the-art primal heuristic in a noncommercial code would be a significant contribution.

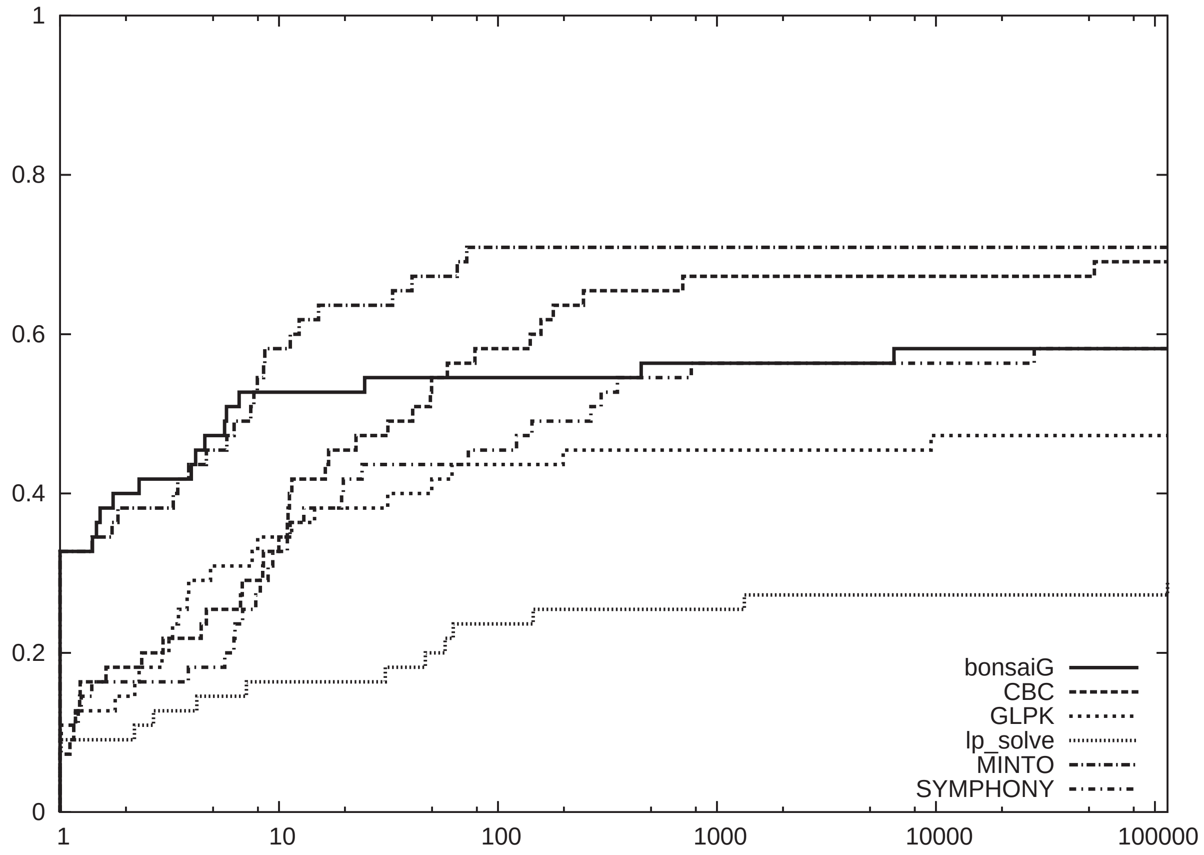

Figure 5: Performance Profile of MILP Solvers on Solved Instances

7 Future Trends and Conclusions

In closing, we want to mention the work of two groups that have been strong supporters and developers of open code and are well-positioned to support such development for many years into the future. The first is the COIN-OR Foundation, a non-profit foundation mentioned several times in this paper, part of whose mission it is to promote the development of open source software for operations research [52]. This foundation, originally a loose consortium of researchers from industry and academia founded by IBM, has become a major provider of open source optimization software and is poised to have a large impact on the field over the coming decade. The second is the NEOS (Network Enabled Optimization System) project [21, 23]. NEOS provides users with the ability to solve optimization problems on a remote server using any of a wide range of available solvers. At current count, there are 55 different solvers for a variety of different optimization problem types available for use. Interfaces to the noncommercial MILP solvers CBC, GLPK, and MINTO are available on NEOS. To use NEOS, the user submits a problem represented in a specified input format (e.g., MPS, AMPL, GMPL, or LP) through either an e-mail interface, a web interface, or a specialized client program running on the user's local machine. The instance is sent

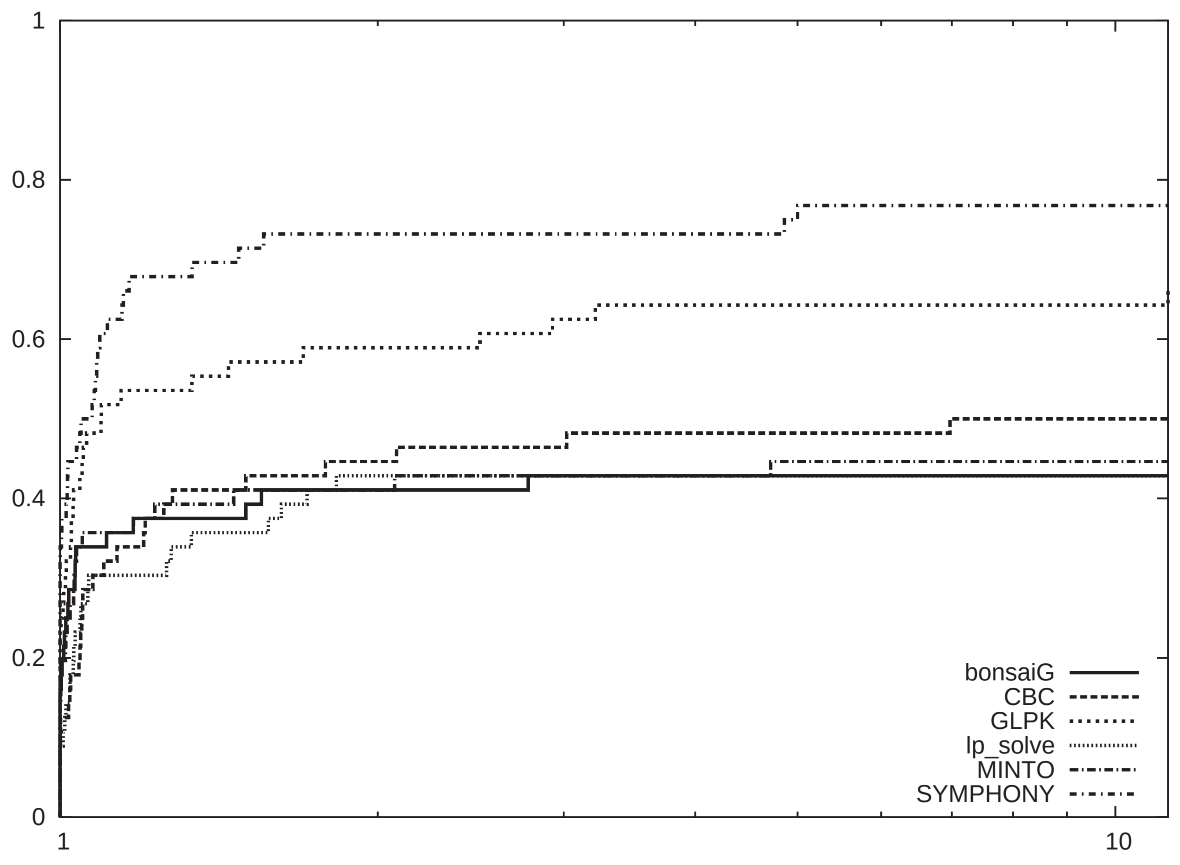

Figure 6: Performance Profile of MILP Solver on Unsolved Instances

to the NEOS server, which locates resources to run the instance and sends the results back to the user. Because the job runs remotely, this provides the user with ability to test multiple solvers without downloading and installing each of them individually. The NEOS project has been solving optimization problems on-line since 1991, and currently handles over 10,000 optimization instance per month.

As for the future, we see no slowing of the current trend toward the development of competitive noncommercial software for solving MILPs. A number of exciting new open source projects are currently under development and poised to further expand the offerings to users of optimization software. One of these is the Abstract Library for Parallel Search (ALPS), a C++ class library for implementing parallel search algorithms planned that is the planned successor to BCP [87]. ALPS will further generalize many of the concepts present in BCP, providing the ability to implement branch-and-bound algorithms for which the bounding procedure is not necessarily LP-based. A second framework, called DECOMP, will provide the ability to automate the solution of decomposition-based bounding problems, i.e., those based on Lagrangian relaxation or Dantzig-Wolfe decomposition [70]. Both of these frameworks will be available as part of the COIN-OR software suite. A new generation of the Open Solver Interface supporting a much wider range of problem types and with better model building features is under development by COIN-OR, along with a new open standard based on LPFML for describing mathematical programming instances [33]. Finally, a MILP solver integrating techniques from constraint programming with those described here is also under development and due out soon [1].

On the whole, we were impressed by the vast array of packages and auxiliary tools available, as well as the wide variety of features exhibited by these codes. The most significant features still missing in open codes are effective logical preprocessors and primal heuristics. More effort is needed in developing tools to fill this gap. While noncommercial codes will most likely continue to lag behind commercial codes when it comes to raw speed in solving generic MILPs out of the box, they generally exhibit a much greater degree of flexibility and extensibility. This is especially true of the solver frameworks, which are designed specifically to allow the development of customized solvers. A number of features appearing in noncommercial codes, such as parallelism, the ability to support column generation, and the ability to solve multi-criteria MILPs, simply do not exist in most commercial codes. Although the noncommercial codes are in general slower than the best commercial codes, we believe that many users will be genuinely satisfied with the features and performance of the codes reviewed here and we look forward to future developments in this fast-growing area of software development.

Acknowledgment

The authors would like to thank Kjell Eikland, John Forrest, Matthew Galati, Lou Hafer, and Matthew Saltzman for their insightful comments that helped to improve the presentation.

References

[1] Achterberg, T. SCIP—a framework to integrate constraint and mixed integer programming. Tech. Rep. ZIB-Report 04-19, Konrad-Zuse-Zentrum für Informationstechnik Berlin, Takustr. 7, Berlin, 2004.

[2] Achterberg, T., Koch, T., and Martin, A. Branching rules revisited. *Operations Research Letters 33* (2004), 42–54.

[3] Applegate, D., Bixby, R., Cook, W., and Chvátal, V., 1996. Personal communication.

[4] Atamtürk, A., Nemhauser, G., and Savelsbergh, M. W. P. Conflict graphs in integer programming. Technical Report LEC-98-03, Georgia Institute of Technology, 1998.

[5] Atamtürk, A., Nemhauser, G., and Savelsbergh, M. W. P. Conflict graphs in solving integer programming problems. *European J. Operational Research 121* (2000), 40–55.

[6] Atamtürk, A., and Savelsbergh, M. Integer-programming software systems. *Annals of Operations Research* (2004). Forthcoming, available at `http://www.isye.gatech.edu/faculty/Martin_Savelsbergh/publications/ipsoftware-final.pdf`.

[7] Balas, E. Facets of the knapsack polytope. *Mathematical Programming 8* (1975), 146–164.

[8] Balas, E., Ceria, S., Cornuejols, G., and Natraj, N. Gomory cuts revisited. *Operations Research Letters 19* (1999).

[9] Balas, E., and Martin, R. Pivot and complement: a heuristic for 0-1 programming. *Management Science* (1980), 86–96.

[10] Balas, E., and Saltzman, M. An algorithm for the three-index assignment problem. *Operations Research 39* (1991), 150–161.

[11] Balas, E., Schmieta, S., and Wallace, C. Pivot and shift—A mixed integer programming heuristic. *Discrete Optimization 1* (2004), 3–12.

[12] Beale, E. M. L. Branch and bound methods for mathematical programming systems. In *Discrete Optimization II* (1979), P. L. Hammer, E. L. Johnson, and B. H. Korte, Eds., North Holland Publishing Co., pp. 201–219.

[13] Beale, E. W. L., and Tomlin, J. A. Special facilities in a general mathematical programming system for non-convex problems using ordered sets of variables. In *Proceedings of the 5th International Conference on Operations Research* (1969), J. Lawrence, Ed., pp. 447–454.

[14] Bénichou, M., Gauthier, J. M., Girodet, P., Hentges, G., Ribière, G., and Vincent, O. Experiments in mixed-integer linear programming. *Mathematical Programming 1* (1971), 76–94.

[15] Berkelaar, M. lp_solve 5.1, 2004. Available from `http://groups.yahoo.com/group/lp_solve/`.

[16] Bixby, R. E., Ceria, S., McZeal, C. M., and Savelsbergh, M. W. P. An updated mixed integer programming library: MIPLIB 3.0. *Optima 58* (1998), 12–15.

[17] Borndörfer, R., and Weismantel, R. Set packing relaxations of some integer programs. *Mathematical Programming 88* (2000), 425 – 450.

[18] Chvátal, V. *Linear Programming.* W. H. Freeman and Co., New York, 1983.

[19] COIN-OR: Computational Infrastructure for Operations Research, 2004. `http://www.coin-or.org`.

[20] Colombani, Y., and Heipcke, S. Mosel: An extensible environment for modeling and programming solutions. In *Proceedings of the Fourth International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimisation Problems (CP-AI-OR'02)* (2002), N. Jussien and F. Laburthe, Eds., pp. 277–290.

[21] Czyzyk, J., Mesnier, M., and Moré, J. The NEOS server. *IEEE Journal on Computational Science and Engineering 5* (1998), 68–75.

[22] Danna, E., Rothberg, E., and LePape, C. Exploring relaxation induced neighborhoods to improve MIP solutions. *Mathematical Programming* (2004). To appear.

[23] Dolan, E., Fourer, R., Moré, J., and Munson, T. Optimization on the NEOS server. *SIAM News 35* (2002), 8–9.

[24] Dolan, E., and Moré, J. Benchmarking optimization software with performance profiles. *Mathematical Programming 91* (2002), 201–213.

[25] Dolan, E., Moré, J., and Munson, T. Optimality measures for performance profiles. Preprint ANL/MCS-P1155-0504, Mathematics and Computer Science Division, Argonne National Lab, 2004.

[26] Driebeek, N. J. An algorithm for the solution of mixed integer programming problems. *Management Science 12* (1966), 576–587.

[27] Eckstein, J. Parallel branch-and-bound methods for mixed integer programming. *SIAM News 27* (1994), 12–15.

[28] Edmonds, J. Maximum matching and a polyhedron with 0-1 vertices. *Journal of Research of the National Bureau of Standards 69B* (1965), 125–130.

[29] Fischetti, M., and Lodi, A. Local branching. *Mathematical Programming 98* (2002), 23–47.

[30] Forrest, J. CBC, 2004. Available from `http://www.coin-or.org/`.

[31] Forrest, J. J. H., Hirst, J. P. H., and Tomlin, J. A. Practical solution of large scale mixed integer programming problems with UMPIRE. *Management Science 20* (1974), 736–773.

[32] Fourer, R., Gay, D. M., and Kernighan, B. W. *AMPL. A Modeling Language for Mathematical Programming.* The Scientific Press, 1993.

[33] Fourer, R., Lopes, L., and Martin, K. LPFML: A W3C XML schema for linear programming, 2004. Available from `http://www.optimization-online.org/DB_HTML/2004/02/817.html`.

[34] Galati, M. COIN-OR tutorials, 2004. Available from `http://coral.ie.lehigh.edu/~coin/`.

[35] Gauthier, J. M., and Ribière, G. Experiments in mixed-integer linear programming using pseudocosts. *Mathematical Programming 12* (1977), 26–47.

[36] Gomory, R. E. An algorithm for the mixed integer problem. Tech. Rep. RM-2597, The RAND Corporation, 1960.

[37] Grötschel, M., Lovász, L., and Schrijver, A. *Geometric Algorithms and Combinatorial Optimization.* Springer-Verlag, New York, 1988.

[38] Gu, Z., Nemhauser, G. L., and Savelsbergh, M. W. P. Cover inequalities for 0-1 linear programs: Computation. *INFORMS Journal on Computing 10* (1998), 427–437.

[39] Gu, Z., Nemhauser, G. L., and Savelsbergh, M. W. P. Lifted flow covers for mixed 0-1 integer programs. *Mathematical Programming 85* (1999), 439–467.

[40] Gu, Z., Nemhauser, G. L., and Savelsbergh, M. W. P. Sequence independent lifting. *Journal of Combinatorial Optimization 4* (2000), 109–129.

[41] Hafer, L. bonsaiG 2.8, 2004. Available from `http://www.cs.sfu.ca/~lou/BonsaiG/dwnldreq.html`.

[42] Hammer, P. L., Johnson, E. L., and Peled, U. N. Facets of regular 0-1 polytopes. *Mathematical Programming 8* (1975), 179–206.

[43] Hoffman, K., and Padberg, M. Solving airline crew-scheduling problems by branch-and-cut. *Management Science 39* (1993), 667–682.

[44] Hultberg, T. FlopC++, 2004. Available from `http://www.mat.ua.pt/thh/flopc/`.

[45] ILOG concert technology. `http://www.ilog.com/products/optimization/tech/concert.cfm`.

[46] Jünger, M., and Thienel, S. The ABACUS system for branch and cut and price algorithms in integer programming and combinatorial optimization. *Software Practice and Experience 30* (2001), 1325–1352.

[47] Ladányi, L. *Parallel Branch and Cut and Its Application to the Traveling Salesman Problem.* PhD thesis, Cornell University, May 1996.

[48] Ladányi, L. BCP, 2004. Available from `http://www.coin-or.org/`.

[49] Linderoth, J. Mip instances, 2004. Available from `coral.ie.lehigh.edu/mip-instances`.

[50] Linderoth, J. T., and Savelsbergh, M. W. P. A computational study of search strategies in mixed integer programming. *INFORMS Journal on Computing 11* (1999), 173–187.

[51] Lougee-Heimer, R. The Common Optimization INterface for Operations Research. *IBM Journal of Research and Development 47* (2003), 57–66.

[52] Lougee-Heimer, R., Saltzman, M., and Ralphs, T. 'COIN' of the OR Realm. *OR/MS Today* (October 2004).

[53] Makhorin, A. GLPK 4.2, 2004. Available from `http://www.gnu.org/software/glpk/glpk.html`.

[54] Marchand, H. *A Study of the Mixed Knapsack Set and its Use to Solve Mixed Integer Programs*. PhD thesis, Facult'e des SciencesAppliquées, Université Catholique de Louvain, 1998.

[55] Marchand, H., and Wolsey, L. Aggregation and mixed integer rounding to solve MIPs. *Operations Research 49* (2001), 363–371.

[56] Margot, F. BAC: A BCP based branch-and-cut example. Report RC22799, IBM Research, 2004.

[57] Martin, A., Achterberg, T., and Koch, T. MIPLIB 2003. `http://miplib.zib.de`.

[58] Mitra, G. Investigation of some branch and bound strategies for the solution of mixed integer linear programs. *Mathematical Programming 4* (1973), 155–170.

[59] Nediak, M., and Eckstein, J. Pivot, cut, and dive: A heuristic for mixed 0-1 integer programming. Tech. Rep. RUTCOR Research Report RRR 53-2001, Rutgers University, 2001.

[60] Nemhauser, G., and Savelsbergh, M. MINTO 3.1, 2004. Available from `http://coral.ie.lehigh.edu/minto/`.

[61] Nemhauser, G., and Wolsey, L. A recursive procedure for generating all cuts for 0-1 mixed integer programs. *Mathematical Programming 46* (1990), 379–390.

[62] Nemhauser, G., and Wolsey, L. A. *Integer and Combinatorial Optimization*. John Wiley and Sons, New York, 1988.

[63] Nemhauser, G. L., and Sigismondi, G. A strong cutting plane/branch-and-bound algorithm for node packing. *Journal of the Operational Research Society 43* (1992), 443–457.

[64] Nemhauser, G. L., and Trotter Jr., L. E. Properties of vertex packing and independence system polyhedra. *Mathematical Programming 6* (1974), 48–61.

[65] Padberg, M. On the facial structure of set packing polyhedra. *Mathematical Programming 5* (1973), 199–215.

[66] Padberg, M. *Linear Optimization and Extensions.* Springer-Verlag, New York, 1995.

[67] Ralphs, T. *Parallel Branch and Cut for Vehicle Routing.* PhD thesis, Cornell University, May 1995.

[68] Ralphs, T. SYMPHONY 5.0, 2004. Available from `http://www.branchandcut.org/SYMPHONY/`.

[69] Ralphs, T. SYMPHONY Version 5.0 User's Manual. Technical Report 04T-020, Lehigh University Industrial and Systems Engineering, 2004.

[70] Ralphs, T., and Galati, M. Decomposition in integer programming. Industrial and Systems Engineering Technical Report 04T-019, Lehigh University, 2004.

[71] Ralphs, T., and Guzelsoy, M. The SYMPHONY callable library for mixed integer programming. To appear in the Proceedings of the Ninth Conference of the INFORMS Computing Society, 2004.

[72] Ralphs, T., Saltzman, M., and Wiecek, M. An improved algorithm for biobjective integer programming and its application to network routing problems. To appear in Annals of Operations Research, 2004.

[73] Ralphs, T. K., Ladányi, L., and Saltzman, M. J. Parallel branch, cut, and price for large-scale discrete optimization. *Mathematical Programming 98* (2003), 253–280.

[74] Roy, J.-S. PuLP : A linear programming modeler in Python. `http://www.jeannot.org/~js/code/index.en.html#PuLP`.

[75] Savelsbergh, M. W. P. Preprocessing and probing techniques for mixed integer programming problems. *ORSA Journal on Computing 6* (1994), 445–454.

[76] Schrage, L., and Wolsey, L. A. Sensitivity analysis for branch and bound linear programming. *Operations Research 33* (1985), 1008–1023.

[77] Schrijver, A. *Theory of Linear and Integer Programming.* Wiley, Chichester, 1986.

[78] Sidebottom, G. Satisfaction of constraints on non-negative integer arithmetic expressions. Open File Report 1990-15, Alberta Research Council, 6815 8th Street, Calgary, Alberta, CA T2E 7H7, 1990.

[79] Thienel, S. ABACUS 2.3, 2004. Available from `http://www.informatik.uni-koeln.de/abacus/`.

[80] Tomlin, J. A. An improved branch-and-bound method for integer programming. *Operations Research 19* (1971), 1070–1075.

[81] Trick, M., and Guzelsoy, M. Simple walkthrough of building a solver with symphony, 2004. Available from `ftp://branchandcut.org/pub/reference/SYMPHONY-Walkthrough.pdf`.

[82] van Heesch, D. Doxygen documentation system, 2004. Available from `http://www.doxygen.org/`.

[83] Vanderbeck, F. A generic view of Dantzig-Wolfe decomposition in mixed integer programming. Working paper, Laboratoire de Mathématiques Appliquées Bordeaux, Université Bordeaux, 2003. Available at `http://www.math.u-bordeaux.fr/~fv/papers/dwcPap.pdf`.

[84] Wolsey, L. A. Faces for a linear inequality in 0-1 variables. *Mathematical Programming 8* (1975), 165–178.

[85] Wolsey, L. A. Valid inequalities for mixed integer programs with generalized and variable upper bound constraints. *Discrete Applied Mathematics 25* (1990), 251–261.

[86] Wolsey, L. A. *Integer Programming.* John Wiley and Sons, New York, 1998.

[87] Xu, Y., Ralphs, T., Ladányi, L., and Saltzman, M. ALPS: A framework for implementing parallel search algorithms. to appear in the Proceedings of the Ninth Conference of the INFORMS Computing Society, 2004.

[88] Zimpl. `http://www.zib.de/koch/zimpl/`.