
Annals of Operations Research 140, 67–124, 2005

Integer-Programming Software Systems

ALPER ATAMTÜRK atamturk@ieor.berkeley.edu
Department of Industrial Engineering and Operations Research, University of California, Berkeley, CA 94720–1777, USA

MARTIN W. P. SAVELSBERGH mwps@isye.gatech.edu
School of Industrial and Systems Engineering, Georgia Institute of Technology, Atlanta, GA 30332–0205, USA

Abstract. Recent developments in integer-programming software systems have tremendously improved our ability to solve large-scale instances. We review the major algorithmic components of state-of-the-art solvers and discuss the options available to users for adjusting the behavior of these solvers when default settings do not achieve the desired performance level. Furthermore, we highlight advances towards integrated modeling and solution environments. We conclude with a discussion of model characteristics and substructures that pose challenges for integer-programming software systems and a perspective on features we may expect to see in these systems in the near future.

Keywords: integer programming, algebraic modeling languages, software

In the last decade, the use of integer-programming (IP) models and software has increased dramatically. Twenty years ago, mainframe computers were often required to solve instances with fifty to a hundred integer variables. Today, instances with thousands of integer variables are solved reliably on a personal computer and high quality solutions for instances of structured problems, such as set partitioning, with millions of binary variables can frequently be obtained in a matter of minutes.

The basis of state-of-the-art integer-programming systems is a linear-programming based branch-and-bound algorithm. Today's IP codes, however, have become increasingly complex with the incorporation of sophisticated algorithmic components, such as advanced search strategies, preprocessing and probing techniques, cutting plane algorithms, and primal heuristics. The behavior of the branch-and-bound algorithm can be altered significantly by changing the parameter settings that control these components. Through extensive experimentation, integer-programming software vendors have determined "default" settings that work well for most instances encountered in practice. However, in integer programming there is no "one-size-fits-all" solution that is effective for all problems. Therefore, integer-programming systems allow users to change the parameter settings, and thus the behavior and performance of the optimizer, to handle situations in which the default settings do not achieve the desired performance. A major portion of this paper is dedicated to a discussion of the important components of modern integer-programming solvers and the parameters available to users for controlling these

components in three state-of-the-art systems: CPLEX[1], LINDO[2], and Xpress-MP[3]. We explain how parameter settings change the behavior of the underlying algorithms and in what situations this may be helpful. We illustrate the impact of modified parameter settings on instances from the MIPLIB 2003 library (Martin, Achterberg, and Koch, 2003).

Although many integer programs can be solved well by fine-tuning parameter settings, there remain cases in which this is not sufficient. In such cases, it may be necessary to develop decomposition or iterative approaches, in which several models representing different subproblems are solved in a coordinated manner, and the solution of one model is used as the input for another model. In order to easily develop and experiment with such approaches, a close integration of modeling tools and optimization engines is required.

Modern integer-programming systems support several means for developing customized solution approaches. The user can modify the basic branch-and-bound algorithm to obtain a customized approach that can include specialized techniques such as dynamic cut and column generation, specialized branching, or heuristics to help find feasible solutions more quickly. This kind of solver customization can be accomplished at several levels. At the highest level are the modeling languages that enable the implementation of sophisticated algorithms using looping features and successive calls to a solver. At the lowest level are the application programming interfaces (APIs) or subroutine libraries that enable interaction with a solver through various low level functions within a programming language. More recently, we have seen the emergence of environments that fall between a modeling language and an application programming interface. These environments aim to provide the ease of model development offered by modeling languages, as well as the efficiency and low level controls of an API. We review some of the options available and provide complete implementations of a simple cutting plane algorithm using Dash Optimization's Xpress-Mosel (Dash Optimization, 2004c) and Xpress-BCL (Dash Optimization, 2004b), and ILOG's OPL/OPL script (ILOG, 2002) and Concert Technology (ILOG, 2003) as examples.

We want to emphasize that the purpose of this paper is *not* to compare the performance of integer-programming solvers. For such comparisons we refer the reader to the website "Benchmarks for Optimization Software" (Mittelman, 2002). Our goal is to show what state-of-the-art integer-programming solvers are capable of, and what advanced features and customization options they offer to the users.

Although tremendous progress has been made over the past several years, many challenges still remain for integer-programming systems today. We will discuss model characteristics or substructures that are known to pose difficulties for modern integer-programming solvers. This type of knowledge is useful for practitioners, of course, but it also points to potential research topics in the area of integer programming.

This paper is intended to be accessible and interesting for a variety of readers, from users of integer-programming models and software who would like to understand more about the methodology so that they can, for example, make more knowledgeable choices; to students who are relatively new to the topic of integer programming and would like to

learn more about it; to operations researchers, computer scientists, and mathematicians who are curious about recent developments, trends, and challenges in this active branch of optimization.

The remainder of the paper is organized as follows. In Section 1, we present a few examples that illustrate what modern IP solvers are capable of. In Section 2, we describe the linear-programming based branch-and-bound algorithm that forms the basis of most integer-programming solvers. In Sections 3–7, we describe the important algorithmic components of modern IP solvers: search, cut generation, preprocessing, and primal heuristics. In Sections 3 and 4, we discuss the choices for node selection and branching and their effect on the search tree. In Section 5, we discuss issues related to cut generation and management. In Section 6, we present several relatively simple and computationally efficient techniques for strengthening the formulations and reducing their size. In Section 7, we review techniques available for quickly finding feasible solutions during the search. In each of the Sections 3–7, we present computations that illustrate the impact of different parameter settings that control the corresponding component. We present experiments on MIPLIB 2003 (Martin, Achterberg, and Koch, 2003) instances for which changing the respective parameters have the most noticeable impact. In Section 8, we discuss the options provided by IP solvers for developing customized solution approaches and review the trend towards integrating modeling and optimization more closely. In Section 9, we discuss model characteristics and substructures that pose challenges for modern integer-programming systems, and finally, in Section 10, we conclude with a perspective on features that we may see appearing in integer-programming systems in the near future.

1. Success stories

As mentioned in the introduction, integer programming is rapidly gaining acceptance as a powerful computational tool that can provide optimal or near optimal solutions to real-life strategic and operational planning problems.

In this section, we illustrate the advances made in integer programming in the area of production scheduling, which is key to manufacturing across industries. The reported speed-ups achieved by the commercial integer-programming solvers is the result of effectively integrating many algorithmic components and enhancements (to be discussed in the remainder of this paper), and advances in computer technology.

As linear programming is at the heart of branch-and-bound methods for integer programming, we observe that Bixby et al. (2002) report a 2360 fold speed-up of the CPLEX linear programming code from 1988 to 2002 and that, in the same period of time, an additional 800 fold speed-up is obtained due to advances in hardware.

Our first example (Bixby et al., 2002) is a weekly production planning model using daily buckets. The objective is to minimize the end-of-day inventory. The model involves production (at a single facility), shipping (via a dedicated fleet), and demand (assumed to be deterministic) from wholesale warehouses. Complicating constraints arise due

to consecutive-day production and minimum truck fleet use requirements. When the model was developed a few years ago, real-life instances could not be solved using the integer-programming technology of that time (CPLEX 5.0). A two-phase approach was developed instead. In the first phase, decisions regarding which products to assign to which machines were made. In the second phase, with variables corresponding to product-machines assignments fixed, a restricted version of the model was solved. Solution times of the integer program with fixed variables using CPLEX 5.0 are about 3500 seconds on a 2 GHz Pentium IV computer. With CPLEX 8.0 this integer program with fixed variables can be solved in 1.4 seconds on the same computer. Moreover with CPLEX 8.0 the original model can be solved in less than 2 hours and the resulting solution is about 20% better than the one obtained with the two-phase approach.

Our second example (Dash Optimization, 2004a) is a multi-period production and inventory planning problem arising in Procter & Gamble's snacks business. It involves capacity constraints, minimum lot-sizes and product changeover restrictions. The objective is to keep inventory as low as possible, while maintaining product availability. Out-of-stocks must be avoided. The line produces ∼50 SKUs, which belong to 6 product families. Capacity is limited, and not far in excess of average demand. Day-to-day demand fluctuates; on top, there are two seasonal peaks. Due to the nature of the production process, capacity must be 100% utilized, or planned to be shut down for a prolonged period. Production is scheduled in multiples of full shifts (8 hours each). The process is continuous (24/7), with the exception of planned maintenance periods. Depending on the SKU, a minimum batch can represent a few days or several months of shipments. The problem was formulated as a discrete lot sizing problem with backlogging. (With planning periods defined as one shift, only one product is produced in a "bucket", and production must run during a whole shift.) The use of strong cutting planes resulted in (near-)integer linear-programming relaxations and branch-and-bound times of less than one minute–on some data sets even less than 15 seconds. This represents a speed-up of over 50 times compared to the initial formulation. The optimization model has been integrated in an in-house built production planning tool and the results were excellent: improved customer service at equal or lower inventories, less disruptions (expediting/schedule changes) in the operation. Planners' productivity increased significantly, as a complete planning cycle (including, among others, data transfers, checks on materials availability and warehouse status) reduced to less than 2 hours.

2. Linear-programming based branch-and-bound

The task of an integer-programming (IP) solver is to solve an instance of the mixed-integer program

$$\begin{array}{lll} & \min & cx + dy \\ \text{(MIP)} & \text{s.t.} & Ax + Gy \leq b \\ & & x \in \mathbb{Z}^n,\ y \in \mathbb{R}^m, \end{array}$$

where c, d, A, G, and b are rational matrices with appropriate dimensions. Here x and y are the *variables* of MIP. Each row (α, γ, β) of (A, G, b) defines a *constraint* of MIP. Constraint (α, γ, β) is satisfied by a point $(x, y) \in \mathbb{R}^n \times \mathbb{R}^m$ if $\alpha x + \gamma y \leq \beta$. A point $(x, y) \in \mathbb{Z}^n \times \mathbb{R}^m$ is said to be *feasible* if it satisfies every constraint of MIP. The set of all feasible points defines the *feasible region*. The *objective function* or *objective* of MIP is $cx + dy$. An *optimal solution* is a feasible point with the smallest objective function value. The *linear programming (LP) relaxation* of MIP is the problem obtained from MIP by dropping the integrality restrictions, i.e., replacing $x \in \mathbb{Z}^n_+$ with $x \in \mathbb{R}^n_+$. As the feasible points of MIP form a subset of the feasible points of its LP relaxation, the optimal value of the LP relaxation provides a lower bound on the optimal value of MIP. Therefore, if an optimal solution to the LP relaxation satisfies the integrality restrictions, then that solution is also optimal for MIP. If the LP relaxation is infeasible, then MIP is also infeasible, and if the LP relaxation is unbounded, then MIP is either unbounded or infeasible.

All commercially available IP software packages employ an LP based branch-and-bound algorithm. To be able to use IP software packages effectively, it is imperative to understand the use of lower and upper bounds on the optimal objective function value in an LP based branch-and-bound algorithm. Therefore, we briefly describe the basic LP based branch-and-bound algorithm for solving MIP. For a comprehensive exposition on integer-programming algorithms we refer the reader to Nemhauser and Wolsey (1988) and Wolsey (1998).

An overview of the basic LP based branch-and-bound algorithm is given in Algorithm 1. The branch-and-bound algorithm follows a "divide-and-conquer" strategy by which the feasible region of MIP is partitioned into subregions and then each subregion is explored recursively. The algorithm maintains a list $\mathbf{L}$ of active subproblems which are the optimization problems over these subregions. Let MIP(k) denote subproblem k. The objective value of any feasible solution to MIP(k) provides an upper bound on the global optimal value. The feasible solution with the smallest objective value found so far is called the *incumbent solution* and its objective value is denoted as z^{best}. Let x^k be an optimal solution to the LP relaxation of MIP(k) with objective value z^k. If x^k satisfies the integrality constraints, then it is an optimal solution to MIP(k) and a feasible solution to MIP, and therefore we update z^{best} as $\min\{z^k, z^{\text{best}}\}$. Otherwise, there are two possibilities: if $z^k \geq z^{\text{best}}$, then an optimal solution to MIP(k) cannot improve on z^{best}, hence the subproblem MIP(k) is removed from consideration; on the other hand, if $z^k < z^{\text{best}}$, then MIP(k) requires further exploration, which is done by branching, i.e., by creating $q \geq 2$ new subproblems MIP($k(i)$), $i = 1, 2, \ldots, q$, of MIP(k) by dividing its the feasible region S^k into q subregions $S^{k(i)}$, $i = 1, 2, \ldots, q$. A simple realization, which avoids x^k in the subproblems of MIP(k) is obtained by selecting a variable x_i for which x^k_i is not integer and creating two subproblems; in one subproblem, we add the constraint $x_i \leq \lfloor x^k_i \rfloor$, which is the floor of x^k_i, and in the other $x_i \geq \lceil x^k_i \rceil$, which is the ceiling of x^k_i. MIP(k) is replaced with its subproblems in the list $\mathbf{L}$ of active subproblems. The smallest among all LP relaxation values associated with the active subproblems provides a

global lower bound on the optimal value. The algorithm terminates when the global lower bound and global upper bound are equal, in which case the list **L** of active subproblems vanishes.

Algorithm 1 The Linear-Programming Based Branch-and-Bound Algorithm

0. **Initialize.**
   $\mathbf{L} = \{\text{MIP}\}$. $z^{\text{best}} = \infty$. $x^{\text{best}} = \emptyset$.
1. **Terminate?**
   Is $\mathbf{L} = \emptyset$? If so, the solution x^{best} is optimal.
2. **Select.**
   Choose and delete a problem MIP(k) from **L**.
3. **Evaluate.**
   Solve the linear programming relaxation LP(k) of MIP(k). If LP(k) is infeasible, go to Step 1, else let z^k be its objective function value and x^k be its solution.
4. **Prune.**
   If $z^k \geq z^{\text{best}}$, go to Step 1. If x^k is not integer, go to Step 5, else let $z^{\text{best}} = z^k$, $x^{\text{best}} = x^k$. Go to Step 1.
5. **Branch.**
   Divide the feasible set S^k of MIP(k) into smaller sets $S^{k(i)}$ for $i = 1, \ldots, q$, such that $\cup_{i=1}^{q} S^{k(i)} = S^k$ and add subproblems MIP($k(i)$), $i = 1, \ldots, q$, to **L**. Go to Step 1.

It is convenient to represent the branch-and-bound algorithm as a *search tree*, in which the nodes of the tree represent the subproblems. The top node of the tree is referred to as the *root* or the *root node* and represents the original MIP. Subsequent descendant (or child) nodes represent the subproblems obtained by branching. At any time, the leaf nodes of the tree denote the list of active subproblems yet to be evaluated.

As the branch-and-bound algorithm terminates when the global lower bound and global upper bound are equal, efforts to make the basic algorithm more efficient need to be focused on rapidly improving these global bounds, i.e., on decreasing the global upper bound and on increasing the global lower bound. Decreasing the global upper bound can only be accomplished by finding improved feasible solutions during the search process. Increasing the global lower bound, as it is the minimum over all lower bounds of actives nodes, can only be accomplished by choosing the node with the smallest lower bound and improve that LP bound. LP bounds are improved either by branching, i.e., dividing the feasible region into smaller pieces, or by adding constraints to the subproblem that cut-off the optimal LP solution but keep at least one optimal integer solution (such constraints are often referred to as *cutting planes*).

The above paragraph highlights two fundamental tradeoffs in linear–programming based branch-and-bound algorithms:

- Should we employ a strategy that focuses on improving the upper bound or one that focuses on improving the lower bound?
- Should we employ a strategy that favors improving the lower bound through the identification of cutting planes or one that favors improving the lower bound through branching?

Much of the remainder of this paper is dedicated to providing a more elaborate discussion of these tradeoffs and the options provided in modern IP solvers for tuning the strategy so as to obtain the best performance on a class of integer programs.

Another potentially effective way of improving lower bounds is to reformulate the problem so that the objective value of the LP relaxation of the reformulation is higher. LP relaxations of different formulations of a problem can be vastly different in terms of the quality of the bounds they provide. A discussion on how to generate formulations with strong LP bounds is beyond the scope of this paper. The reader is referred to (Wolsey, 1998) for this important topic. Preprocessing, which is discussed in Section 6, can be viewed as a way of deriving an alternative stronger formulation by simple transformations.

3. Node selection

As mentioned above, a fundamental tradeoff in linear–programming based branch-and-bound algorithms is whether to focus on decreasing the global upper bound or on increasing the global lower bound. The primary control mechanism available for this is the node (subproblem) selection strategy.

A popular method of selecting a node to explore is to choose the one with the lowest value z^k. This is known as *best-bound* search (or best-first search). This node selection strategy focuses the search on increasing the global lower bound, because the only way to increase the global lower bound is to improve the LP relaxation at a node with the lowest LP relaxation value. Another well-known method of selecting a node to explore is to always choose the most recently created node. This is known as *diving* search (or depth-first search). This node selection strategy focuses the search on decreasing the global upper bound, because feasible solutions are typically found deep in the tree.

In addition to a different focus, both methods also have different computational attributes. Diving search has low memory requirements, because only the sibling nodes on the path to the root of the tree have to be stored. Furthermore, the changes in the linear programs from one node to the next are minimal—a single bound of a variable changes, which allows warm-starts in the LP solves. (As the linear programs are solved using a simplex algorithm, the resident optimal basis can be used to warm-start subsequent LP solves.) Best-bound search, on the other hand, favors exploring nodes at the top of the

tree as these are more likely to have low LP relaxation values. This, however, can lead to large list of active subproblems. Furthermore, subsequent linear programs have little relation to each other—leading to longer solution times. (The time to solve LPs at each node can be reduced by warm-starting from the optimal basis of the parent node, but this requires saving basis information at all the nodes. Consequently, memory requirements for best-bound search may become prohibitive.)

Let z^{opt} be the optimal objective value for MIP. We say that node k is superfluous if $z^k > z^{opt}$. Best-bound search ensures that no superfluous nodes will be explored. On the other hand, diving search can lead to the exploration of many superfluous nodes that would have been fathomed, had we known a smaller z^{best}.

Most integer-programming solvers employ a hybrid of best-bound search and diving search, trying to benefit from the strengths of both, and switch regularly between the two strategies during the search. In the beginning the emphasis is usually more on diving, to find high quality solutions quickly, whereas in the later stages of the search, the emphasis is usually more on best-bound, to improve the global lower bound.

For selecting nodes that may lead to good feasible solutions, it would be useful to have an estimate of the value of the best feasible solution at the subproblem of a given node. The *best-projection* criterion, introduced by Hirst (1969) and Mitra (1973) and the *best-estimate* criterion found in Bénichou et al. (1971) and Forrest, Hirst, and Tomlin (1974), incorporate such estimates into a node selection scheme. The best-projection method and the best-estimate method differ in how an estimate is determined. Once estimates of the nodes are computed, both methods select a node with the smallest estimate.

For any node k, let $s^k = \sum_{i \in I} \min(f_i, 1 - f_i)$ denote the sum of its integer infeasibilities. The best projection criterion for node selection is to choose the node with the smallest estimate

$$E^k = z^k + \left(\frac{z^{\text{best}} - z^0}{s^0} \right) s^k.$$

The value $(z^{\text{best}} - z^0)/s^0$ can be interpreted as the change in objective function value per unit decrease in infeasibility. Note that this method requires a known z^{best}.

The estimate of the best-projection method does not take into account which variables are fractional or the individual costs for satisfying the integrality requirements of each variable. A natural extension of the best-projection method is to use estimates P_i^- and P_i^+ of the per unit degradation of the objective function value if we fix x_i to its floor and ceiling, respectively. These estimates are called *down* and *up pseudocosts* and are discussed in more detail in Section 4. This extension is known as the *best-estimate* criterion. Here, the estimate of the best solution obtainable from a node is defined as

$$E^k = z^k + \sum_{i \in I} \min(|P_i^- f_i|, |P_i^+ (1 - f_i)|),$$

which has the advantage that it does not require a known upper bound z^{best}.

Forrest, Hirst, and Tomlin (1974) and Beale (1979) propose a two-phase method that first chooses nodes according to the best-estimate criterion. Once a feasible solution is found, they propose to select nodes that maximize the *percentage error*, defined as

$$PE^k = 100 \times \frac{z^{\text{best}} - E^k}{z^k - z^{\text{best}}}$$

for node k. The percentage error can be thought of as the amount by which the estimate of the solution obtainable from a node must be in error for the current solution to not be optimal.

A typical branching scheme creates two child nodes. Usually the value of the LP relaxation at the parent node is (temporarily) assigned to the node as an (under)estimate of the bound it provides until the node is selected and evaluated. Consequently, a node selection strategy must rank these nodes to establish the evaluation order. For schemes that prioritize nodes based on an estimate of the optimal solution obtainable from that node, the ranking is immediately available, since individual estimates are computed for the newly created nodes. For schemes that do not distinguish between the importance of the two newly created nodes, such as best-bound or diving search, we have to resort to other means. One possibility, if we branch on variable x_i, is to select the down node first if $f_i < 1 - f_i$ and the up node first otherwise (Land and Powell, 1979).

3.1. Software node selection options

3.1.1. CPLEX

One of the most important factors influencing the choice of node selection strategy is whether the goal is to find a good feasible solution quickly or to find a proven optimal solution. CPLEX offers a single parameter `mip emphasis` that allows users to indicate that high level goal. The following options are available:

- balanced (default),
- feasibility,
- optimality,
- bestbound,
- hiddenfeas.

With the setting of `balanced`, CPLEX works towards a proof of optimality, but balances that with an effort towards finding high quality feasible solutions early in the optimization. When set to `feasibility`, CPLEX favors identifying feasible solutions early on as it optimizes the problem, thereby increasing the time is takes to establish a proof of optimality. When set to `optimality`, less effort is spent towards finding feasible solutions early. With the setting `bestbound`, even greater emphasis is placed on proving optimality through moving the best bound value, so that the detection of feasible solutions

along the way becomes almost incidental. When set to `hiddenfeas`, CPLEX works even harder to find high quality feasible solutions.

CPLEX allows more detailed control over the solution process for knowledgeable users. Node selection rules in CPLEX can be set using the parameter `mip strategy nodeselect`. The following options are available

- best-bound search: the node with the best objective function value will be selected,
- best-estimate search: a node's progress towards integer feasibility relative to its degradation of the objective function value will be estimated,
- alternate best-estimate search: a proprietary variant of best-estimate search,
- depth-first search.

When best-estimate node selection is in effect, the parameter *bbinterval* defines the frequency at which backtracking is done by best-bound.

The parameter `mip strategy backtrack` controls how often backtracking is done during the branching process. The decision when to backtrack depends on three values that change during the course of the optimization:

- the objective function value of the best integer feasible solution ("incumbent")
- the best remaining objective function value of any unexplored node ("best node")
- the objective function value of the most recently solved node ("current objective").

CPLEX does not backtrack until the absolute value of the difference between the objective of the current node and the best node is at least as large as the target gap, where "target gap" is defined to be the absolute value of the difference between the incumbent and the best node, multiplied by the backtracking parameter. Low values of the backtracking parameter thus tend to increase the amount of backtracking, which makes the search process more of a pure best-bound search. Higher parameter values tend to decrease backtracking, making the search more of a pure depth-first search.

The parameter `mip strategy dive` controls "probing dives." The tree traversal strategy occasionally performs probing dives, where it looks ahead at both child nodes before deciding which node to choose. The following options are available:

- automatic: choose when to perform a probing dive,
- traditional: never perform probing dives,
- probing dive: always perform probing dives,
- guided dive: spend more time exploring potential solutions that are similar to the current incumbent.

CPLEX allows users to specify the preferred branching direction, either up, down, or leave it to CPLEX' discretion.

For a detailed description of CPLEX user options the reader is referred to ILOG (2003).

3.1.2. LINDO

The node selection rule is set by the parameter `nodeselrule` in LINDO. Available options are:

- depth-first search,
- best-bound search,
- worst-bound search,
- best-estimate search using pseudo-costs,
- a mixture of the above.

For a detailed description of LINDO user options please refer to LINDO Systems (2002).

3.1.3. Xpress-MP

The setting of `nodeselection` determines which nodes will be considered for solution once the current node has been solved. The available options are:

- local-first search: choose from the two child nodes if available; if not, then choose from all active nodes,
- best-first search; all active nodes are always considered,
- local depth-first search: choose from the two child nodes if available; if not, then choose from the deepest nodes,
- best-first, then local-first search: all nodes are, considered for `breadthfirst` nodes, after which the local-first method is used,
- depth-first: choose from the deepest active nodes.

To determine the default setting of `nodeselection` Xpress-MP analyzes the matrix.

Once the set of nodes to consider is determined based on the chosen node selection strategy, the setting of `backtrack` determines the actual selection of the node to be processed next. The available options are:

- If `miptarget` is not set, choose the node with the best estimate. If `miptarget` is set (by user or during the tree search), the choice is based on the Forrest-Hirst-Tomlin criterion.
- Always choose the node with the best estimated solution.
- Always choose the node with the best bound on the solution (default).

For a detailed description of XPRESS-MP user options please refer to Dash Optimization (2004d).

3.2. Sample computations on node selection strategies

To demonstrate the effect of the node selection strategy on the overall solution process, we have conducted the following experiment. We have used Xpress-MP 2004b to solve MIPLIB problems with different settings of the control parameter `nodeselection`. The results of the experiment for eight instances are summarized in Table 1. The maximum CPU time was set to 1800 seconds and we present the value of the best feasible solution, the number of feasible solutions found, and, in case the instance was solved to optimality, the solution time (which happened for only one of the eight instances).

The most striking observation is that with the default settings no feasible solution was found within the given time limit for one of the instances. As expected, the two node selection strategies that focus specifically on finding good feasible solutions, i.e., best project (Hirst-Forrest-Tomlin) search and best estimate search, identify more feasible solutions during the solution process than the other node selection strategies.

It also interesting to observe that the best-bound search strategy takes more than three times as long to prove optimality than local-depth-first search on the one instance that could be solved within the given time limit.

While examining these computational results, it is important to observe that certain parameter settings can exhibit quite different computational behavior on instances from different sources. However, it is generally true to that certain parameter settings exhibit fairly consistent behavior on instances from the same source. Therefore, in practice tuning branching parameters can be extremely important. Even though a combination of parameters may not work well on a wide variety of instances, it may work extremely well on the class of problems that need to be solved.

4. Branching

An obvious way to divide the feasible region of MIP is to choose a variable x_i that is fractional in the current linear programming solution $\bar{x}$ and impose the new bounds of $x_i \leq \lfloor \bar{x}_i \rfloor$ to define one subproblem and $x_i \geq \lceil \bar{x}_i \rceil$ to define another subproblem. This type of branching is referred to as *variable dichotomy*, or simply branching on a variable.

If there are many fractional variables in the current linear programming solution, we must select one variable to define the division. Because the effectiveness of the branch-and-bound method strongly depends on how quickly the upper and lower bounds converge, we would like to "branch on a variable" that will improve these bounds as much as possible. It has proven difficult to devise general rules for selecting a branching variable that will affect the upper bound. However, there are ways to predict which fractional variables will improve the lower bound most when required to be integral.

Table 1
Impact of node selection strategy.

	Default			Local-depth-first			fht			Best-estimate			Best-bound		
Name	Value	#sol	Time	Value	#sol	Time	Value	#sol	Time	Value	#sol	Time	Value	#sol	Time
mzzv11	–	–		−19,460.00	1		−20,350.00	3		−20,072.00	2		–	–	
mzzv42z	−19,200.00	2		−19,550.00	4		−18,920.00	9		−18,940.00	10		−19,200.00	2	
timtab1	801,497.00	10		813,926.00	13		788,306.00	23		786,522.00	16		801,497.00	10	
tr12-30	133,213.00	9		133,910.00	8		133,285.00	13		134,712.00	11		133,213.00	9	
swath	468.63	18		468.48	18		507.84	9		468.63	19		468.63	13	
mkc	−558.29	18		−556.39	20		−527.27	9		−545.53	17		−588.29	18	
a1c1s1	12,271.38	9		12,464.94	14		12,438.31	8		12,821.22	7		12,271.38	9	
harp2	−73,899,739.00	32	687	−73,899,739.00	37	208	−73,899,739.00	253	263	−73,899,739.00	76	219	−73,899,739.00	32	683

Estimation methods work as follows: with each integer variable x_i, we associate two estimates P_i^- and P_i^+ for the per unit degradation of the objective function value if we fix x_i to its rounded down value and rounded up value, respectively. Suppose that $\bar{x}_i = \lfloor \bar{x}_i \rfloor + f_i$, with $f_i > 0$. Then by branching on x_i, we will estimate a change of $D_i^- = P_i^- f_i$ on the down branch and a change of $D_i^+ = P_i^+(1 - f_i)$ on the up branch. The values P_i^- and P_i^+ are often referred to as *down* and *up pseudocosts*.

One way to obtain values for P_i^- and P_i^+ is to simply observe the increase in the LP bound due to branching. That is, letting z_{LP}^- and z_{LP}^+ denote the values of the linear programming relaxation for the down and up branches, then we compute the pseudocosts as

$$P_i^- = \frac{z_{LP}^- - z_{LP}}{f_i} \quad \text{and} \quad P_i^+ = \frac{z_{LP}^+ - z_{LP}}{1 - f_i}.$$

To complement this monitoring strategy an initialization strategy is needed. A popular choice is to use the objective function coefficient as an initial estimate of the expected change.

Once we have computed the estimates on the degradation of the LP objective, given that we branch on a specific variable, we must still decide how to use this information in order to make a branching choice. The goal of variable selection methods is to select a variable that maximizes the difference between the LP objective value of the node and its children. However, since there are two branches at a node, there are different ways to combine degradation estimates of the two branches. The most popular ones include maximizing the sum of the degradations on both branches, i.e., branch on the variable $x_{\hat{\imath}}$ with

$$\hat{\imath} = \arg\max_i \{D_i^+ + D_i^-\},$$

and maximizing the minimum degradation on both branches, i.e., branch on the variable $x_{\hat{\imath}}$ with

$$\hat{\imath} = \arg\max_i \{\min\{D_i^+, D_i^-\}\}.$$

Instead of using an estimate of the change in the objective function based on pseudocosts to choose among possible variables to branch on, it is also possible to tentatively change the bound on a variable and partially solve the resulting LP to observe what happens to the objective function value. (That is, to simply perform a small number of simplex iterations.) This is the idea behind *strong branching*. Not surprisingly, without care, the computational requirements can become prohibitive. Solving LPs, albeit partially, just to identify a branching variable has to be done very carefully. To keep the computational requirements of strong branching under control, typically only the first few iterations of an LP solve are executed for only a small set of "promising" variables.

When the problem has constraints of the form $\sum_{j \in T} x_j = 1$ (or $\sum_{j \in T} x_j \leq 1$) for some $T \subseteq \{1, 2, \ldots, n\}$, another scheme to divide the feasible region of MIP can be used. Here, a subset $T' \subseteq T$ for which the solution of the LP relaxation x^i at node i satisfies $0 < \sum_{j \in T'} x_j^i < 1$ is chosen. The constraint $\sum_{j \in T'} x_j = 0$ is enforced in one subregion, and the constraint $\sum_{j \in T \setminus T'} x_j = 0$ is enforced in the other subregion. Note that these constraints can be enforced by fixing the variables' bounds. When there is a logical ordering of the variables in the set T, this set is sometimes called a *special ordered set* (SOS) and hence this division method is sometimes called *SOS branching*. One advantage of branching on such a constraint instead of on a single variable is that the branch-and-bound tree is more "balanced." Suppose we have a constraint $\sum_{j \in T} x_j = 1$, and we choose to branch on a single variable x_{j^*}. Setting $x_{j^*} = 0$ has little effect on variables $x_i, i \in T \setminus \{j^*\}$. However, setting $x_{j^*} = 1$ fixes $x_i = 0$ for all $i \in T \setminus \{j^*\}$; hence the asymmetry in the two branches.

The discussion above illustrates the ideas and concepts incorporated in branching schemes employed by modern IP solvers. In a sense, most of them can be viewed as attempts to identify the variables representing the most important decisions in the problem and the values to which these variables need to be set to obtain the best possible solution. In many cases, knowledge about the problem being solved can be used to effectively guide these choices. For example, many planning problems involve multiple periods and decisions relating to the first period impact the decisions relating to subsequent periods. Therefore, it is usually important to focus on decisions in the earlier periods before focusing on decisions in the later periods. Thus, it is preferable to branch on variables representing decisions in the earlier periods before branching on variables representing decisions in the later periods. IP software allows user to convey this type of information through variable priorities. Other situations where guiding branching decisions using priorities can be effective involve problems in which there exist a natural hierarchy among the decisions. In facility location problems, for example, two types of decisions have to be made: which facilities to open, and which facility to assign customers to. Clearly, the impact of deciding to open or close a facility on the objective value is much larger than the impact of deciding to assign a customer to a particular (open) facility or not. Thus, it is usually more effective to branch on variables representing the decisions to open or close facilities, before branching on variables representing the decisions to assign a customer to a particular facility.

4.1. Software branching options

4.1.1. CPLEX

Users can set `mip strategy variableselect` to choose one of the following variable selection rules:

- minimum integer infeasibility: branch on a fractional variable whose fraction is closest to being integer,

- maximum integer infeasibility: branch on a fractional variable whose fraction is closest to being 0.5,
- automatic: solver determines strategy,
- pseudo-costs: derive an estimate about the effect of each proposed branch from duality information,
- strong branching: analyze potential branches by performing a small number of simplex iterations,
- pseudo reduced costs: a computationally cheaper version of pseudo-costs.

Users can also prioritize the variables for branching and set limits on the number of candidate variables (`strongcand`) and simplex iterations (`strongit`) for strong branching.

4.1.2. LINDO
The users can specify the variable selection strategies in LINDO by setting the `varselrule` parameter. The following options are available:

- most infeasible (fractional part closest to 0.5),
- smallest index,
- cause large change in the objective function.

The last option is executed by either strong branching or pseudocost. Users can adjust the amount of strong branching application with the parameter `strongbranchlevel`. This parameter specifies the maximum depth of nodes in which strong branching is used (default is 10).

LINDO allows users to specify variable priorities to bias variable selection.

4.1.3. Xpress-MP
The setting of `varselection` determines the formula for calculating a degradation estimate for each integer variable, which in turn is used to select the variable to be branched on at a given node. The variable selected is the one with the minimum estimate. The variable estimates are combined to calculate an overall degradation estimate of the node, which, depending on the `backtrack` setting, may be used to choose an active node. The following options are available:

- determined automatically,
- the minimum of up and down pseudo-cost,
- the sum of up and down pseudo-cost,
- the maximum of up and down pseudo-cost, plus twice the minimum of up and down pseudo-cost,
- the maximum of the up and down pseudo-cost,

- the down pseudo-cost,
- the up pseudo-cost.

Xpress-MP assigns a priority for branching to each integer variable; either the default value of 500 or one set by the user in the so-called *directives* file. A low priority means that the variable is more likely to be selected for branching. Up and down pseudocosts can also be specified for each integer variable. The optimizer selects the variable to branch on from among those with the lowest priority. Of these, it selects the one with the highest estimated cost of being satisfied (degradation).

4.2. Sample computations on node selection strategies

To demonstrate the effect of the variable selection strategy on the overall solution process, we have conducted the following experiment. We have used Xpress-MP 2004b to solve MIPLIB problems with different settings of the control parameter `varselection`. The results of the experiment for eight instances are summarized in Table 2. The maximum CPU time was set to 1800 seconds and we present the value of the best feasible solution, and, in case the instance was solved to optimality, the solution time (which happened for only one of the seven instances).

The most striking observation is that with the default variable selection strategy, as was the case with the default node selection strategy, no feasible solution was found within the given time limit for one of the instances.

Note that the variable selection rule "maximum of the up and down pseudocosts plus twice the minimum of the up and down pseudocosts" does reasonably well on this set of instances. The motivation for this criterion is to accurately balance the impact on both child nodes created.

Table 2
Impact of variable selection strategy.

	Default		min{up, down}		up + down		max{up, down} +2 min{up, down}	
Name	Value	Time	Value	Time	Value	Time	Value	Time
mzzv11	–		–		−15,020.00		−17,074.00	
mzzv42z	−19,200.00		−18,786.00		−19,090.00		−18,180.00	
timtab1	801,497.00		790,118.00		791,058.00		791,962.00	
tr12-30	133,213.00		133,789.00		133,278.00		132,967.00	
swath	468.63		472.06		473.03		471.96	
mkc	−558.29		−556.59		−551.29		−552.24	
a1c1s1	12,271.38		12,326.00		12,224.48		12,223.91	
harp2	−73,899,739.00	687	−73,899,739.00	580	−73,899,739.00	663	−73,899,739.00	446

5. Cutting planes

As mentioned earlier, the key for solving large-scale MIPs successfully in an LP based branch-and-bound framework is to have strong upper and lower bounds on the optimal objective value. Modern solvers generate a host of cutting planes to strengthen the LP relaxations at the nodes of the branch-and-bound tree.

Before discussing the issues related to adding cutting planes, we briefly review the basics of the approach. A *valid inequality* for an MIP is an inequality that is satisfied by all feasible solutions. A *cutting plane*, or simply *cut*, is a valid inequality that is not satisfied by all feasible points of the LP relaxation. If we find a cut violated by a given LP solution, we can add it to the formulation and strengthen the LP relaxation. By doing so, we modify the current formulation in such a way that the relaxed feasible region becomes smaller but the MIP feasible region does not change. Then the LP relaxation of the new formulation is resolved and the cut generation process is repeated as necessary. It is also possible to add cutting planes that may remove part of the MIP feasible region as long as at least one integer optimal solution remains intact. Such cuts are typically referred to as *optimality cuts*.

The branch-and-cut method is a generalization of the branch-and-bound method, in which we add violated cuts to the formulation at the nodes of the search tree. If no violated cuts are found or the effectiveness of the cutting planes in improving the LP bound tails off, we branch. Branch-and-cut algorithms generalize both pure cutting plane algorithms, in which cuts are added until an optimal IP solution is found, i.e., no branching is done, as well as branch-and-bound algorithms, in which no cuts are added.

Naturally, a branch-and-cut algorithm spends more time in solving LP relaxations at the nodes. However, the result of improved LP bounds is usually a significantly smaller search tree. If the cuts improve the LP bounds significantly and the addition of cuts does not increase the LP solution times too much, a carefully implemented branch-and-cut algorithm can be much faster than a branch-and-bound algorithm. Finding a good tradeoff between cutting and branching is essential in reducing the solution time and this is very much problem dependent.

The cutting plane generation algorithms implemented in IP solvers can be classified into two broad categories. The first are general cuts that are valid for any MIP problem; these include Gomory mixed-integer (Balas et al., 1996; Gomory, 1960) and mixed-integer rounding (MIR) cuts (Marchand and Wolsey, 2001; Nemhauser and Wolsey, 1990). The second category includes strong special cuts from knapsack (Balas, 1975; Balas and Zemel, 1978; Crowder, Johnson, and Padberg, 1983; Gu, Nemhauser, and Savelsbergh, 1998; Hammer, Johnson, and Peled, 1975; Padberg, 1979), fixed-charge flow (Gu, Nemhauser, and Savelsbergh, 1999; Padberg, Roy, and Wolsey, 1985) and path (Van Roy and Wolsey, 1987), and vertex packing (Padberg, 1973) relaxations of MIPs. Strong inequalities identified for these simpler substructures are quite effective in improving LP relaxations of more complicated sets. IP solvers automatically identify such substructures by analyzing the constraints of the formulation and add the appropriate cuts if they are found to be violated by LP relaxation solutions in the search tree.

By default, IP solvers look for all of these substructures and try to add appropriate cuts. As with all techniques designed to improve the performance of the basic branch-and-bound algorithm, one should keep in mind that the time spent looking for cuts should not outweigh the speed-up due to improved LP bounds. Solvers provide users with parameters to set the level of aggressiveness in looking for cuts or to disable certain classes of cuts.

Computational experience has shown that knapsack cover cuts are effective for problems with constraints modeling budgets or capacities, fixed-charge flow and path cuts are effective for production and distribution planning problems with fixed-charge network flow substructures, and clique cuts are effective for problems with packing and partitioning substructures modeling conflicts and coverage, such as time-indexed formulations of scheduling problems, crew pairing, staff rostering problems.

When none of the above mentioned special cuts work well, general mixed-integer cuts may help to improve the quality of the LP relaxations. Gomory mixed-integer cuts, which had been overlooked for several decades due to initial unsuccessful implementations in pure cutting plane algorithms, have become standard in state-of-the-art IP solvers after their effectiveness in a branch-and-cut framework was demonstrated by Balas et al. (1996). Caution is in order, though, when using Gomory mixed-integer cuts, especially for instances with large numbers of variables, as the cuts tend to be dense, i.e., contain nonzero coefficients with many digits for many variables, compared to the special cuts. This can cause numerical difficulties for the LP solvers and make the LP relaxations significantly harder to solve. The mixed-integer rounding variants from the original constraints may provide sparse alternatives (Marchand and Wolsey, 2001).

Other related and important issues regarding cutting planes are how often to generate them and how to manage them. The more cuts added to the formulation, the bigger the formulation gets and the harder it becomes to solve the LP relaxations. It is not uncommon to add thousands of cuts to an instance with only a few hundred initial variables and constraints. This is especially true when adding user-defined cutting planes that make use of specific problem structure. Therefore, it may become necessary to remove some of the cuts that are not binding from the formulation to reduce the LP solution times. Removal of cuts is usually done in two phases. In the first phase, an inactive cut is moved from the formulation to a pool, where it is checked for violation before new cuts are generated. If the size of the cut pool increases beyond a limit, to reduce the memory requirements, inactive cuts in the pool may be removed permanently in the second phase. In order to control the size of the formulations, the solvers also provide options for setting the maximum number of cut generation phases, the maximum depth of nodes to generate cuts, number of nodes to skip before generating cuts, etc.

5.1. Software cut generation options

5.1.1. CPLEX

The aggressiveness of cut generation in CPLEX can be adjusted for individual cut classes or for all classes by setting the parameters `mip cuts all`, `cliques`, `covers`, `disjunctive`,

`flowcovers`, `gomory`, `gubcovers`, `implied`, `mircut`, `pathcut` to one of the following values:

- do not generate,
- automatic,
- moderate,
- aggressive.

The aggressiveness of generating a cut class should be increased if the particular cut class is useful for solving the problem of interest. Other options available to users include

- `aggcutlim`: maximum number of constraints that can be aggregated for generating flow cover and mixed-integer rounding cuts,
- `cutpass`: maximum number of passes for generating cuts,
- `cutsfactor`: the maximum number of cuts added to the formulation is (`cutsfactor`-1) times the original number of constraints in the model (no cuts are added if `cutsfactor` is set to at most 1).

5.1.2. LINDO

Cuts generated at higher nodes of the search tree usually have more effect on the solution process than the ones generated at deeper nodes. LINDO provides user options to control the depth of nodes for generating cuts. The following user parameters are available:

- `cutlevel top`: This parameter turns on or off the types of cuts to be applied at the root node of the search tree. Bit settings are used to enable the cuts including knapsack cover, GUB cover, flow cover, lifting cuts, plant location cuts, disaggregation cuts, lattice cuts, coefficient reduction cuts, greatest common divisor cuts, Gomory cuts, objective integrality cuts, basis cuts, cardinality cuts.
- `cutlevel tree`: Parameter for turning on or off the types of cuts to be applied at the nodes other than the root node of the search tree.
- `cuttimelim`: Maximum time in seconds to be spent on cut generation.
- `cutfreq`: Cut generation frequency. Cuts will be generated at every `cutfreq` nodes.
- `cutdepth`: A threshold value for depth of nodes. Cut generation will be less likely for nodes deeper than `cutdepth`.
- `maxcutpass top`: Maximum number of cut generation phases at the root node of the search tree.
- `maxcutpass tree`: Maximum number of cut generation phases at nodes other than the root node of the search tree.

- `maxnonimp_cutpass`: Maximum of cut phases without improving the LP objective (default is 3).
- `addcutobjtol`: Minimum required LP objective improvement in order to continue cut generation.
- `addcutper`: Maximum number of cuts as a factor of the total number of original constraints (default is 0.5).

5.1.3. Xpress-MP

The parameter `cutstrategy` is used to adjust the aggressiveness of cut generation in Xpress-MP. A more aggressive cut strategy, generating a greater number of cuts, will result in fewer nodes to be explored, but with an associated time cost in generating the cuts. The fewer cuts generated, the less time taken, but the greater subsequent number of nodes to be explored. The parameter may be set to one of the following:

- automatic selection of the cut strategy,
- no cuts,
- conservative cut strategy,
- moderate cut strategy,
- aggressive cut strategy.

The parameter `cutdepth` sets the maximum depth in the tree search at which cuts will be generated. Generating cuts can take a lot of time, and is often less important at deeper levels of the tree since tighter bounds on the variables have already reduced the feasible region. A value of 0 signifies that no cuts will be generated.

The parameter `cutfrequency` specifies the frequency at which cuts are generated in the tree search. If the depth of the node modulo `cutfrequency` is zero, then cuts will be generated.

The parameter `covercuts` specifies the number of rounds of lifted cover cuts at the top node. The process of generating these can be carried out a number of times, further reducing the feasible region, albeit at a significant time penalty. There is usually a good payoff from generating these at the top node, since these inequalities then apply to every subsequent node in the tree search.

The parameter `treecovercuts` specifies the number of rounds of lifted cover inequalities generated at nodes other than the top node in the tree.

The parameter `gomcuts` specifies the number of rounds of Gomory cuts at the top node. These can always be generated if the current node does not yield an integral solution. However, Gomory cuts are dense and may slow down the LP solving times.

The parameter `treegomcuts` specifies the number of rounds of Gomory cuts generated at nodes other than the first node in the tree.

Table 3
Impact of cuts.

	with cuts				without cuts		
problem	cuts	zroot	nodes	time	zroot	nodes	time
10teams	19	924	162	9.03	917	128843	1800*
gesa2_o	157	2.5731e+07	1409	3.44	2.5476e+07	67159	84.90
modglob	186	2.0708e+07	136	0.29	2.0431e+07	751117	208.71
mzzv42z	622	−20844.74	439	195.89	−21446.24	14049	1800*
p2756	353	3116.51	28	0.43	2701.14	174131	306.93

The parameter `maxcuttime` specifies the maximum amount of time allowed for generation of cutting planes and re-optimization. The limit is checked during generation and no further cuts are added once this limit has been exceeded.

5.2. Sample computations with cutting planes

Here we present sample computations that illustrate the impact of cut generation on the performance of the solution algorithms. In Table 3 we compare the performance of CPLEX with and without adding cutting planes to the formulation for six problems from the MIPLIB 2003 (Martin, Achterberg, and Koch, 2003). In this table, we present the total number of cuts added to the formulation, the objective value of the LP relaxation immediately before branching, the number of branch-and-bound nodes, and the total CPU time elapsed in seconds. The computations are done with CPLEX version 9.0 on a 3GHz Pentium4/Linux workstation.

Cut generation with default settings reduces the number of nodes in the search tree significantly for all problems in Table 4; and this reduces the computation time for almost all problems. Two of the problems could not be solved without adding cuts within the time limit 1800 seconds (marked with*). Different classes of cutting planes are effective for the listed problem in Table 4. For *10teams* a large number of cliques are generated in preprocessing by probing and clique cuts are the most effective on problem. Problem *gesa2_o* contains general integer variables and therefore Gomory and MIR cuts are more effective. Flow cuts are very effective for *modglob* due to the fixed-charge flow substructure in this problem. Cover and gub cover inequalities are the most effective ones for *p2756* due to the 0-1 knapsack constraints contained in the formulation of *p2756*.

6. Preprocessing

Preprocessing refers to manipulating an MIP formulation in an effort to decrease the overall solution time. These manipulations reduce the feasible region in such a way that at least one optimal integer solution remains feasible.

Table 4
Impact of preprocessing.

	with preprocessing					without preprocessing				
problem	cons	vars	zinit	nodes	time	cons	vars	zinit	nodes	time
aflow30a	455	818	983.17	31165	86.39	479	842	983.17	13903	52.65
fixnet6	477	877	3192.04	50	0.39	478	878	1200.88	9241	41.01
mod011	1535	6872	−6.2082e+07	73	127.95	4480	10958	−6.2122e+07	3904	900.47
mzzv11	8601	9556	−22773.75	4680	736.04	9499	10240	−22945.24	6611	1792.16
nw04	36	46190	16310.67	569	26.55	36	87482	16310.67	1712	96.58

IP solvers employ a number of preprocessing techniques *before* starting the branch-and-bound algorithm. We will briefly discuss a number of these techniques here and mention how they can be controlled in modern IP solvers. We refer the reader to Anderson and Anderson (1995), Gondzio (1997), Guignard and Spielberg (1981), and Savelsbergh (1994) for detailed explanations of preprocessing techniques.

Preprocessing usually alters the given formulation significantly by eliminating and substituting variables and constraints of the formulation, as well as changing the coefficients of the constraints and the objective function. IP solvers then solve this reduced formulation rather than the original one and recover the values of the original variables afterwards. The fact that there are two different formulations becomes an important issue, when the user interacts with the solver during the algorithm via callable libraries to add constraints and variables to the formulation. Modern solvers allow the user to work on either the original or the reduced formulation.

Common preprocessing techniques include detecting inconsistent constraints, eliminating redundant constraints, and strengthening bounds on variables. A constraint is redundant if its removal from the formulation does not change the feasible region. Strengthening bounds refers to increasing the lower bound or decreasing the upper bound of a variable in such a way that at least one optimal integer solution remains feasible. For a 0-1 variable, this is equivalent to fixing it to either 0 or 1 and eliminating it from the formulation. A simple implementation of these techniques is accomplished by considering each constraint in isolation and verifying whether the constraint is inconsistent with the bounds of the variables or whether there is a feasible solution when a variable is fixed to one of its bounds. Techniques that are based on checking infeasibility are called primal reduction techniques. Dual reduction techniques make use of the objective function and attempt to fix variables to values that they take in any optimal solution. Another preprocessing technique that may have a big impact in strengthening the LP relaxation of a MIP formulation is coefficient improvement. This technique updates the coefficients of the formulation so that the constraints define a smaller feasible set for LP relaxation, hence leading to improved the LP bounds.

Probing is a more involved preprocessing technique that considers the implications of fixing a variable to one of its bounds by considering all constraints of the formulation

simultaneously. For instance, if fixing a 0-1 variable (a variable that takes value either 0 or 1) x_1 to one, forces a reduction in the upper bound of x_2, then one can use this information, in all constraints in which x_2 appears and possibly detect further redundancies, bound reductions, and coefficient improvements. One should note, however, that fixing each variable to one of its bounds and analyzing the implications on the whole formulation may become quite time consuming for large instances. Therefore, one should weigh the benefits of probing in reducing the problem size and strengthening the LP relaxation with the time spent performing it, in order to decide whether it is useful for a particular instance.

Finally, we note that all the preprocessing techniques that are applied before solving the LP relaxation of a formulation at the root node of the branch-and-bound tree can be applied in other nodes of the tree as well. However, since the impact of node preprocessing is limited to only the subtree rooted by the node, one should take into consideration whether the time spent on node preprocessing is worthwhile.

Even though it is applied *after* solving the LP relaxation, we mention here another technique that is very effective in eliminating variables from the formulation. Given an optimal LP solution, reduced cost fixing is a method for checking whether increasing or decreasing a nonbasic variable by $\delta > 0$ would lead to an LP solution with worse objective value than that of the best known feasible integer solution. If this is the case, the variable will not change from its nonbasic value by more than δ in any optimal integer solution. For a 0-1 variable, $\delta \leq 1$ implies that the variable can be fixed to its nonbasic value. For other variables, this technique can be used to tighten their bounds without changing the optimal IP value.

Reduced cost fixing is particularly effective for problems with small gap between optimal IP and LP values and with large variations in the objective coefficients. For example, it is not uncommon to eliminate more than half of the variables by reduced cost fixing for set partitioning instances arising in crew scheduling applications.

Coefficient improvement can be very effective in strengthening the LP relaxations of problems with large differences in constraint coefficients. In particular, models with "big M" coefficients, which are frequently used to represent logical implications such as "either or" and "if then" statements, can benefit significantly from coefficient improvement.

Probing tends to be most effective in tightly constrained problems, in which setting a variable to a certain value fixes the value of many others in all feasible solutions. On the other hand, it can be very time consuming for large-scale instances of time-indexed formulations of scheduling problems, crew pairing and staff rostering problems with set packing substructures, since one typically gets an overwhelming number of feasibility implications for such formulations.

6.1. Software preprocessing options

6.1.1. CPLEX

Users can control the presolver of CPLEX by changing the levels of the following preprocessing options from none to aggressive:

- `aggregatorind`: maximum trials of variable substitutions,
- `coeffreduce`: coefficient improvement,
- `boundstrength`: bound strengthening,
- `probe`: probing level,
- `presolvenode`: indicator to perform preprocessing in the tree nodes,
- `numpass`: maximum number of preprocessing passes.

The users can also activate or deactivate primal and/or dual reductions by setting the parameter `reduce`. Primal reductions should be disabled if columns are to be added to a formulation. Relaxing a problem by adding columns renders primal reductions, that are based on infeasibility arguments, invalid. Similarly, dual reductions should be disabled if the user will introduce constraints that are invalid for the original formulation.

6.1.2. LINDO

Users can control the preprocessing operations in LINDO by changing the value of the parameter `prelevel`. Bit settings are used to turn on or off the following operations:

- simple presolve,
- probing,
- coefficient reduction,
- elimination,
- dual reductions,
- use dual information.

6.1.3. Xpress-MP

Users can control preprocessing in Xpress-MP through three parameters: `presolve`, `presolveops`, and `mippresolve`.

The parameter `presolve` determines whether presolving should be performed prior to starting the main algorithm.

- Presolve applied, but a problem will not be declared infeasible if primal infeasibilities are detected. The problem will be solved by the LP optimization algorithm, returning an infeasible solution, which can sometimes be helpful.
- Presolve not applied.
- Presolve applied.
- Presolve applied, but redundant bounds are not removed. This can sometimes increase the efficiency of the barrier algorithm.

The parameter `presolveops` specifies the operations which are performed during the presolve. The following options are available:

- singleton column removal,
- singleton row removal,
- forcing row removal,
- dual reductions,
- redundant row removal,
- duplicate column removal,
- duplicate row removal,
- strong dual reductions,
- variable eliminations,
- no IP reductions,
- linearly dependant row removal.

The parameter `mippresolve` determines the type of integer preprocessing to be performed.

- No preprocessing.
- Reduced cost fixing will be performed at each node. This can simplify the node before it is solved, by deducing that certain variables' values can be fixed based on additional bounds imposed on other variables at this node.
- Logical preprocessing will be performed at each node. This is performed on binary variables, often resulting in fixing their values based on the constraints. It greatly simplifies the problem and may even determine optimality or infeasibility of the node before the simplex method commences.
- Probing of binary variables is performed at the top node. This sets certain binary variables and then deduces effects on other binary variables occurring in the same constraints.

6.2. Sample computations with preprocessing

Now we present sample computations in order to illustrate the impact of preprocessing on the performance of the solution algorithms. In Table 4 we compare the performance of CPLEX with and without preprocessing for six problems from the MIPLIB 2003 (Martin, Achterberg, and Koch, 2003). In this table, we present the number of constraints and variables in the formulation, the objective value of the initial LP relaxation, the

number of branch-and-bound nodes, and the total CPU time elapsed in seconds for both options. The computations are done with CPLEX version 9.0 on a 3 GHz Pentium4/Linux workstation with 1 GB memory.

Default settings of preprocessing have a positive impact on the performance of the algorithm for most of the problems in Table 4. Typically the number of nodes and time to solve the problems reduce significantly after preprocessing. For *fixnet6* the improvement is due to, not problem size reduction, but to the strengthening of the LP relaxation. For *nw04*, however, the improvement in the performance is largely due to the significant number of reduction in the number of variables. For *mod011*, *mzzv11*, the improvement appears to be due to both problem size reduction and LP strengthening. However, preprocessing appears to have a negative effect on *aflow30a* and it may be turned off for this problem.

7. Primal heuristics

Primal heuristics are algorithms that attempt to find feasible solutions to MIP quickly. For a minimization problem, while cutting planes are employed to strengthen the lower bound, primal heuristics are used to improve the upper bound on the optimal objective value. Rather than waiting for an integer feasible LP solution at a node, one attempts to find feasible solutions to MIP early in the search tree by means of simple and quick heuristics.

The motivation for employing primal heuristics is to produce a good upper bound early in the solution process and prune as much of the tree as possible. For instance, if one could find an optimal solution at the root node with a heuristic, branch-and-bound would be used only to prove the optimality of the solution. For a fixed branching rule, any node selection rule would then produce the same tree with the minimum number of nodes. Thus, primal heuristics play a role complementary to cutting plane algorithms in reducing the gap between the bounds and consequently the size of the branch-and-bound tree.

Most solvers apply several heuristics that combine partial enumeration, preprocessing, LP solving, and reduced cost fixing. In order to get a feasible integer solution, a good analysis of the constraints is crucial when deciding which variables to fix and to what values. CPLEX Relaxation Induced Neighborhoods (RINS) heuristic fixes variables that have the same value in the incumbent and LP solution and solves the remaining smaller MIP with limited branching in an effort to obtain a feasible solution quickly (Danna, Rothberg, and Pape, 2003).

As with cutting planes, it seems reasonable to spend more effort on finding good feasible solutions early in the search process, since this would have the most impact on the running time. Solvers provide users with parameters to set the frequency of applying primal heuristics and the amount of enumeration done in the heuristics. Users may find it useful to adjust these parameters during the course of the algorithm according to the node selection rule chosen, depth of the node processed, and the gap between the best known upper and lower bound.

7.1. Software primal heuristic options

7.1.1. CPLEX

Users can adjust the aggressiveness of the periodic heuristic application in CPLEX by setting the parameter `heuristicfreq`. The heuristic is then applied every `heuristicfreq` nodes in the tree. Setting this parameter to -1 disables the heuristic. With the default setting 0, the frequency is determined automatically by CPLEX. A similar parameter `rinsheur` controls the frequency of RINS heuristic calls.

7.1.2. LINDO

Two parameters are available in LINDO to users for controlling the amount of heuristic application. These are

- `heuinttimlim`: Minimum total time in seconds to spend in finding heuristic solutions.
- `heulevel`: This parameter controls the actual heuristic applied. Use higher levels for a heuristic that spends more time to find better solutions.

7.1.3. Xpress-MP

The parameter `heurstrategy` specifies the heuristic strategy.

- Automatic selection of heuristic strategy.
- No heuristics.
- Rounding heuristics.

The parameter `heurdepth` sets the maximum depth in the tree search at which heuristics will be used to find feasible solutions. It may be worth stopping the heuristic search for solutions below a certain depth in the tree.

The parameter `heurfreq` specifies the frequency at which heuristics are executed during the tree search. If the depth of the node modulo `heurfreq` is zero, then heuristics will be used.

The parameter `heurmaxnode` specifies the maximum number of nodes at which heuristics are used in the tree search.

The parameter `heurmaxsol` specifies the maximum number of heuristic solutions that will be found in the tree search.

7.2. Sample computations with primal heuristics

In order to illustrate the impact of employing primal heuristics on the performance of the solution algorithms, we performed an experiment using the heuristic level feature of LINDO that controls the aggressiveness of the primal heuristics applied. If heuristic level is zero, no primal heuristic is applied. The higher the level is, the more time is spent in finding a feasible solution and usually the better is the heuristic solution found. In Table 5

Table 5
Impact of primal heuristics.

	heur level = 0			heur level = 3			heur level = 5			heur level = 10		
problem	htime	time	nodes	htime	time	nodes	htime	time	nodes	htime	time	nodes
enigma	0.00	46.71	10273	1.95	2.00	725	1.93	1.99	725	2.11	2.20	725
p0282	0.00	10.55	308	1.21	8.56	34	2.30	7.14	409	4.94	10.25	165
pp08acuts	0.00	72.57	1145	3.63	67.17	904	4.42	71.89	866	4.95	70.73	857
qnet1	0.00	215.84	764	6.59	196.10	593	11.05	35.75	88	11.19	34.14	174
gesa3_o	0.00	171.05	538	3.41	56.69	103	7.48	84.22	304	7.16	25.84	99

we compare the performance of LINDO with different heuristic levels for five problems from the MIPLIB (Martin, Achterberg, and Koch, 2003). In this table, we present the time spent on the heuristic, the total CPU time elapsed in seconds, and the number of branch-and-bound nodes. The computations are done with LINDO API 2.0 on a 2GHz Pentium4/XP workstation with 512 MB memory.

Better feasible solutions found early in the branch-and-cut algorithm may lead to increased pruning of the tree due to tighter bounds. Also good feasible solutions lead to improved reduced cost fixing, hence reduction in the problem size, which reduces the LP solution times.

In Table 5 we observe that the performance of the algorithm improves when heuristic level is positive, i.e., when a heuristic is used to find a feasible solution. However, the heuristic level that gives the best overall performance varies from problem to problem. If it is difficult to find feasible solutions, it may be worthwhile to spend more time on heuristics. This is a feature users should experiment with to find the right level for their problem in order to achieve the best performance.

8. Integrated modeling and optimization environments

One of the reasons that early mathematical programming solvers found little application in practice was the considerable programming effort required to prepare the data into a format recognizable by the solvers. The solvers typically required the constraint matrix of an instance to be provided in the form of a list of coefficients and associated row and column indices. Creating such a representation from a model and a given data set was time-consuming and error-prone. Furthermore, changing and refining models was cumbersome, to say the least, as it required modifying the input generation programs.

An important step forward in the development of mathematical programming tools was the definition of the Mathematical Programming System (MPS) format by IBM in the early sixties. The MPS format resembles the internal data structures of the algorithms; it represents the components of an instance sequentially. An MPS file is processed in batch mode. In the early eighties, matrix generation languages were developed to facilitate the generation of MPS files. The most popular were DATAFORM (from Ketron), OMNI

(from Haverly Systems), and MGG (from Scion). Even though it is lengthy and rather cryptic to the human eye, the MPS format became a standard for specifying and exchanging mathematical programming problems, and it is still supported by modern commercial mathematical programming systems.

In order to enable practitioners to update and maintain their models more easily, however, more flexible languages were needed. The algebraic formulation became the basis of the next generation of modeling languages. These algebraic modeling languages are declarative in nature, allow a concise representation of the model, and provide for separation of model and data, which is extremely important for maintainability of the models. Examples of algebraic modeling languages include GAMS, AMPL, LINGO, and mp-model. Many large-scale practical applications have been developed using these algebraic modeling languages.

The availability of modeling languages and improved solvers has led to an increase in the use of mathematical programming in practice. As a result, the demand for better and tighter integration of modeling tools and solvers has increased. This demand has been addressed in two ways. One approach has been the development of scripting languages, such as OPL script. Another has been the development of programming language interfaces, such as EMOSL. However, the repeated execution of complete models after small modifications and/or communication of problem matrices via files can be expensive in terms of execution times. Recently completely integrated modeling and solving languages (that avoid intermediate file storage) have been developed, for example Xpress-Mosel.

8.1. Lot sizing application

We illustrate the value of these modern mathematical programming modeling and optimization tools in the development of a customized branch-and-cut program for the solution of the classical *Lot-Sizing Problem*. The lot-sizing problem considers production planning over a horizon of NT time periods. In period $t \in \{1, 2, \ldots, NT\}$, a given demand d_t must be satisfied by production in that period and by inventory carried over from earlier periods. The unit production cost in period t is equal to c_t and there is a set-up cost f_t associated with production in period t. The objective is to determine how much to produce in each period so as to minimize the total costs over the horizon. Let the production in period t be denoted by $y_t \in \mathbb{R}_+$, and let $x_t \in \{0, 1\}$ indicate whether the plant operates during period t. Letting $d_{ik} = \sum_{t=i}^{k} d_t$, the lot-sizing (LS) problem can be formulated as

$$\min \sum_{t=1}^{n} (f_t x_t + c_t y_t)$$

$$\sum_{s=1}^{t} y_s \geq d_{1t}, \qquad t \in 1, 2, \ldots, NT, \tag{1}$$

$$y_t \leq d_{tn} x_t, \qquad t \in 1, 2, \ldots, NT, \tag{2}$$

$$x_t \in \{0, 1\}, \; y_t \in \mathbb{R}_+, \quad t \in 1, 2, \ldots, NT.$$

A well-known class of inequalities for the lot-sizing problem is the (ℓ, S)-inequalities described as

$$\sum_{t\in\{1,2,\ldots,\ell\}\setminus S} y_t + \sum_{t\in S} d_{t\ell}x_t \geq d_{1\ell} \qquad \forall\, S \subseteq \{1,2,\ldots,\ell\}, \quad \ell \in \{1,2,\ldots,NT\}.$$

Adding these inequalities to the basic formulation strengthens the formulation significantly. In fact, it has been shown (Barany, Van Roy, and Wolsey, 1984) that replacing (1) with the more general (ℓ, S)-inequalities will make the LP relaxation integral, i.e., it will always have an integral optimal solution. Thus it suffices to solve just the LP relaxation with (ℓ, S)-inequalities for finding an optimal solution to LS.

Unfortunately, the number of (ℓ, S)-inequalities is very large—an exponential function of NT. Consequently, it is impossible, except for very small instances, to explicitly add all the (ℓ, S)-inequalities to the formulation. Therefore, we have to handle the (ℓ, S)-inequalities implicitly. Fortunately, identifying violated (ℓ, S)-inequalities is easy. Given a solution $(\bar{x}, \bar{y})$, for a fixed $\ell \in \{1, 2, \ldots, n\}$, the left hand side of the (ℓ, S)-inequality

$$\sum_{t\in\{1,2,\ldots,\ell\}\setminus S} \bar{y}_t + \sum_{t\in S} d_{t\ell}\bar{x}_t,$$

is minimized by letting $t \in S$ if $d_{t\ell}\bar{x}_t < \bar{y}_t$ and $t \notin S$, otherwise. Hence for each ℓ, we can easily check whether there is a violated (ℓ, S)-inequality. This leads to the simple Algorithm 2.

Algorithm 2 Cutting plane algorithm for LS

```
Read the model to obtain the current formulation
Solve the LP relaxation of the current formulation and let (x, y) be an optimal solution
for ℓ = 1 to NT do
    S = ∅, lhsval = 0
    for t = 1 to ℓ do
        if d_tℓ x_t < y_t then
            lhsval = lhsval + d_tℓ x_t
            S = S ∪ {t}
        else
            lhsval = lhsval + y_t
        end if
    end for
    if lhsval < d_1ℓ then
        Add Σ_{t∈{1,2,...,ℓ}\S} y_t + Σ_{t∈S} d_ℓt x_t ≥ d_1ℓ to the current formulation
        Go to Step 2
    end if
end for
```

In the next four sections, we show how Algorithm 2 can be implemented using Xpress-Mosel and Xpress-BCL, and using ILOG OPL/OPL script and ILOG Concert Technology. In our discussions, we focus on the key concepts and the relevant portions of the implementation. Complete source codes are provided in the appendices.

The two implementations discussed in Sections 8.2 and 8.3 demonstrate that more sophisticated and involved solution approaches for classes of difficult integer programs can relatively easily be implemented using modeling languages that are tightly coupled with the underlying optimization engines.

Even though modern modeling languages provide data manipulation capabilities, often necessary when setting up the integer program to be solved and when processing the solution produced by the solver, there are situations where lower level control is desired. Lower level control is often available through object oriented libraries that are part of the integer-programming software suite. These object oriented libraries allow users to build integer programs step-by-step within their C/C++ or Java programs, with functions to add variables and constraints. Once the integer program is completely defined, it is solved using the underlying optimizer. The libraries also provide a variety of functions to access the solution directly from within their program.

The two implementations discussed in Sections 8.4 and 8.5 illustrate the relative ease with which sophisticated and involved solution approaches for classes of difficult integer programs can be implemented using such object oriented libraries.

A major advantage of object-oriented libraries, as in XPRESS-BCL and ILOG-Concert, over traditional callable libraries is that variable and constraint indexing is completely taken over by the libraries. This allows the user to refer to model variable and constraint names when implementing an algorithm, just like in Mosel and OPL/OPL script, without having to worry about the indexing scheme of the underlying solver.

Application development and maintenance with modeling and optimization languages is easier than with object-oriented libraries. Therefore, using modeling and optimization languages may be the method choice for users for which time-to-market is critical. On the other hand, even though implementations using object-oriented libraries take a little more time to develop and maintain, they have the advantage of faster run times and the flexibility provided by greater control, such as access to the preprocessed formulation and the LP tableau.

8.2. Lot sizing using Xpress-Mosel

Here we describe the basics of Xpress-Mosel implementation of Algorithm 2. The complete Xpress-Mosel code can be found in Appendix A. As a first step, we need to specify the LS model in the Mosel language. A model specification in the Mosel language starts with a declaration of the model entities.

```
  declarations
    ...
    DEMAND: array(T) of integer      ! Demand per period
    SETUPCOST: array(T) of integer   ! Setup cost per period
    PRODCOST: array(T) of integer    ! Production cost per period
    D: array(T,T) of integer         ! Total demand in periods t1 - t2
    ...
    product: array(T) of mpvar       ! Variables representing production
                                       in period t
    setup: array(T) of mpvar         ! Variables representing a setup in
                                       period t ...
  end-declarations
```

We introduce arrays `DEMAND`, `SETUPCOST`, and `PRODCOST`, to hold the instance data, a 2-dimensional array `D` to contain all partial sums of period demands, and two arrays `product` and `setup` of continuous variables. The instance data arrays can be instantiated in different ways, for example by direct assignment or by importing data from spreadsheets of databases.

A powerful feature of modern modeling languages is that they allow data manipulations to be carried out directly within the model specification. For example, the 2-dimensional array `D` is instantiated using the following statement.

```
forall(s,t in T) D(s,t) := sum(k in s..t) DEMAND(k)
```

Once the declaration and instance data instantiation portions of the model have been completed, we continue with the specification of the objective, the constraints, and the variable types. (Note the separation of model and data.)

```
MinCost := sum(t in T) (SETUPCOST(t) * setup(t) + PRODCOST(t) *
           product(t))
forall(t in T) Demand(t) := sum(s in 1..t) product(s) >= sum
                            (s in 1..t)
DEMAND(s)
forall(t in T) Production(t) := product(t) <= D(t,NT) * setup(t)
forall(t in T) setup(t) is_binary
```

After the model has been specified, we can define the cutting plane procedure. The procedure also starts with a declaration part. In this case, we define two arrays to hold the values of the variables after an LP has been solved, and we define an array to hold the violated cuts that are generated.

```
procedure cutgen
declarations
  ...
```

```
    solprod, solsetup: array(T) of real
    ...
    cut: array(range) of linctr
    ...
  end-declarations
```

The remainder of the procedure is fairly straightforward. After solving the LP relaxation, the solution values are retrieved, and violated cuts, if they exist, are identified.

```
  ...
  repeat
    minimize(...)

    forall(t in T) do
      solprod(t) := getsol(product(t))
      solsetup(t) := getsol(setup(t))
    end-do

    forall(l in T) do

      lhsval := 0
      forall(t in 1..l)
           if (solprod(t) > D(t,l)*solsetup(t) + EPS)
             then lhsval += D(t,l)*solsetup(t)
             else lhsval += solprod(t)
           end-if

      if (lhsval < D(1,l) - EPS) then
           cut(..) := sum(t in 1..l)
             if (solprod(t)<(D(t,l)*solsetup(t))+EPS, product(t), D(t,l)
             *setup(t)) >= D(1,l)
      end-if

      end-do
    until (...)
  ...
```

For a detailed description of Dash Optimization's Xpress-Mosel please refer to Dash Optimization (2004c).

8.3. Lot sizing using ILOG OPL/OPL script

Modeling with ILOG's optimization programming language (OPL) is done in a fashion similar to the Mosel implementation. The first step is the declaration of data and variables. For LS, this is done as follows:

```
int NT = ...;
range T 1..NT;
int DEMAND[T] = ...;
int D[s in T, t in T] = sum (k in [s..t]) DEMAND[k];
...
var float+ product[T];
var float+ setup[T] in 0..1;
```

`NT` and `DEMAND` will be initialized later when specifying data. The variable `product` is declared as a nonnegative array over the range of integers 1..NT, whereas setup is a 0-1 array with the same range. The next step is to declare the constraints as

```
constraint prodsetup[T];
constraint meetdemand[T];
```

Then the model can be completed by defining the objective function and the constraints. For LS, the OPL model is specified as

```
minimize sum(t in T) (SETUPCOST[t]*setup[t] + PRODCOST[t] * product[t]);
subject to {
  forall(t in T)
    meetdemand[t] : sum(s in 1..t) product[s] >= sum (s in 1..t)
    DEMAND[s];
  forall(t in T)
    prodsetup[t] : product[t] <= D[t,NT] * setup[t];
}
```

In order to implement the cutting plane algorithm for LS, we use OPL Script, which is a procedural language that allows combining models and solutions. First we extend the model with the constraints

```
forall(c in 0..ncuts) {
  Cuts[c] : sum (t in pcut[c]) product[t] +
            sum (t in scut[c]) D[t,Lval[c]]*setup[t] >= D[1,Lval[c]];
}
```

The arrays `Lval, pcut`, and `scut` are declared in OPL script as

```
Open int Lval[int+];
Open setof(int) pcut[int+];
```

and imported to the model file as

```
import Open Lval;
import Open pcut;
```

Since the number of cuts that will be added is unknown, these are declared as open arrays, whose size will be incremented during execution using the `addh` function (method). The cut generation loop is straightforward.

```
Model m("ELScuts.mod");
...
repeat {
  m.solve();
  forall (l in 1..NT) {
    value := 0;
  setof(int) Scut:=t |t in 1..l: m.product[t]>m.D[t,l]*m.setup[t]+eps;
    setof(int) Pcut := t |t in 1..l diff Scut;
    value := sum (t in Pcut) m.product[t] + sum(t in Scut) m.D[t,l]
    *m.setup[t];
    if (value < m.D[1,l] - eps) then {
      pcut.addh();
      scut.addh();
      Lval.addh();
      pcut[ncuts-1] := Pcut;
      scut[ncuts-1] := Scut;
      Lval[ncuts-1] := l;
    }
  }
}UNTIL (...);
```

After the model `m` is solved, the optimal LP solution values `m.product` and `m.setup` are checked to see whether an (ℓ, S) inequality is violated. If so, the indices of the integer and continuous variables for the corresponding (ℓ, S)-inequality are stored in `scut[ncuts-1]` and `scut[ncuts-1]` to be used in the model.

The complete OPL script code for Algorithm 2 can be found in Appendix B. For a detailed description of ILOG's OPL/OPL script see (ILOG, 2002).

8.4. Lot sizing using Xpress-BCL

In our presentation, we will focus on the objects and functions that specifically deal with integer-programming concepts, such as variables, objective, and constraints.

The basic objects in Xpress-BCL relating to integer programs are: problem, variable, and linear expression. For the lot-sizing example, we start by creating a problem object

p, creating variable objects for production and setup, and associate the variable objects with the problem object.

```
XPRBprob p("Els");        /* Initialize a new problem in BCL */

XPRBvar prod[T];          /* Production in period t */
XPRBvar setup[T];         /* Setup in period t */

for(t=0;t<T;t++) {
  prod[t]=p.newVar(xbnewname(‘‘prod %d",t+1));
  setup[t]=p.newVar(xbnewname(‘‘setup %d",t+1), XPRB_BV);
}
```

The actual integer program is built by defining linear expressions over the variables. To build the objective, we create a linear expression object `cobj`, define the linear expression representing the objective function, and associate the linear expression object with the problem object.

```
XPRBlinExp cobj;

for(t=0;t<T;t++) {
  cobj += SETUPCOST[t]*setup[t] + PRODCOST[t]*prod[t];
}
p.setObj(cobj);
```

Similarly, to build the constraints, we create a linear expression object `le`, define the linear expression representing the constraint, and associate the linear expression object with the problem object.

```
XPRBlinExp le;

for(t=0;t<T;t++) {
  le=0;
  for(s=0;s<=t;s++) {
    le += prod[s];
  }
  p.newCtr("Demand", le >= D[0][t]);
}
for(t=0;t<T;t++) {
  p.newCtr("Production", prod[t] <= D[t][T-1]*setup[t]);
}
```

Note that it is also possible to associate a linear expression representing a constraint with the problem object without explicitly creating a linear expression object first.

Once the integer program has been completely defined, we can invoke the `solve` method of the problem object to solve the instance. In case of the lot sizing example we want to solve the linear programming relaxation.

```
p.solve("lp");
```

Once a solution has been found, it can be accessed using the methods provided by the objects.

```
objVal = p.getObjVal();

for(t=0;t<T;t++) {
  solProd[t]=prod[t].getSol();
  solSetup[t]=setup[t].getSol();
}
```

The remainder of the solution procedure is straightforward to implement as it involves similar steps.

The complete C++ implementation of Algorithm 2 using Xpress-BCL can be found in Appendix C. For a detailed description of Xpress-BCL refer to Dash Optimization (2004b).

8.5. Lot sizing using ILOG concert technology

Using ILOG Concert Technology is very similar to using Xpress-BCL. The basic objects relating to integer programs are: model, variable, and linear expression.

```
IloModel model(env);
IloNumVarArray product(env, NT, 0, IloInfinity, ILOFLOAT);
IloNumVarArray setup(env, NT, 0, 1, ILOINT);
```

Linear expressions are used to build a model. For example, the objective function `obj` is constructed and added to the model as follows:

```
IloExpr obj(env);
obj = IloScalProd(SETUPCOST,setup) + IloScalProd(PRODCOST,product);
model.add(IloMinimize(env,obj));
```

Here `IloScalProd` is a Concert function to represent the scalar product of two arrays (`sum(t in T) (SETUPCOST[t]*setup[t])` in OPL syntax).

Constraints can be added using a syntax very much like in an algebraic modeling language:

```
for(t = 0; t < NT; t++) {
  model.add(product[t] <= D[t][NT-1]*setup[t]);
}
```

Also, expressions can be built iteratively, which is useful when defining complicated constraints as illustrated below

```
for(t = 0; t < NT; t++){
  IloExpr lhs(env), rhs(env);
  for(s = 0; s <= t; s++){
    lhs += product[s];
    rhs += DEMAND[s];
  }
  model.add( lhs >= rhs );
  lhs.end(); rhs.end();
}
```

Here, expressions `lhs` and `rhs` are augmented conveniently, before they are used to define a constraint.

Once the integer program has been completely defined, we can load it into the solver, solve the instance, and access the solution.

```
IloCplex cplex(model);
cplex.solve();

cplex.getObjValue()
cplex.getValue(product[t])
```

The complete C++ implementation of Algorithm 2 using ILOG's Concert Technology can be found in Appendix D. For a detailed description of ILOG Concert Technology refer to ILOG (2003).

9. Challenges

Even though the algorithmic developments of the past decades have resulted in far more powerful integer-programming solvers, there are still many practical integer programs that cannot be solved in a reasonable amount of time. In this section, we will discuss a few model characteristics and substructures that are known to pose difficulties for modern integer-programming solvers. This knowledge is useful for practitioners, of course, but it also identifies potentially rewarding research topics in the area of integer programming.

9.1. Symmetry

An integer program is considered to be symmetric if its variables can be permuted without changing the structure of the problem (Margot, 2003). Symmetry is a phenomenon

occurring in scheduling problems, graphs problems, and many other classes of optimization problems (Sherali and Smith, 2001). Symmetry poses a problem for integer-programming solvers as there are many assignments of values to variables that represent the same solution, causing branching to become ineffective.

To illustrate, consider the scheduling problem of minimizing the makespan on identical parallel machines. Given m machines and n jobs with processing times p_j, $j = 1, \ldots, n$, the objective is to assign the jobs to the machines so that the latest completion time among all machines is minimized. Letting x_{ij} equal 1 if job j is assigned to machine i, and 0 otherwise, the problem can be formulated as

$$\begin{aligned} \min\ & z \\ \text{s.t.}\ & \sum_{j=1}^{n} p_j x_{ij} \le z \quad i = 1, \ldots, m && (3) \\ & \sum_{i=1}^{m} x_{ij} = 1 \quad j = 1, \ldots, n && (4) \\ & x_{ij} \in \{0, 1\} \quad i = 1, \ldots, m; \quad j = 1, \ldots, n. \end{aligned}$$

Now, since the machines are identical, permuting the machine indices does not affect the structure of the problem. For example, switching the assignment of all jobs assigned to machine 1 and those assigned to machine 2 will result in essentially the same solution. As a consequence, it is no longer possible to improve the linear programming bound by simply fixing a fractional variable to either 0 or 1.

Several methods for dealing with symmetric integer programs have been proposed in the literature. Sherali and Smith (2001) suggest breaking the symmetry by adding constraints to the problem. In the case of our scheduling problem, for example, the symmetry can be broken by imposing an arbitrary order on the machines, which can be accomplished by replacing constraints (3) with

$$\sum_{j=1}^{n} p_j x_{1j} \le \sum_{j=1}^{n} p_j x_{2j} \le \cdots \le \sum_{j=1}^{n} p_j x_{mj} \le z. \tag{5}$$

Note that switching the assignment of all jobs assigned to machine 1 and those assigned to machine 2 will no longer be possible (unless sum of the processing times on both machines happen to be the same), and that the latest completed job will always be on machine m.

Margot proposes tree pruning and variable fixing procedures that exploit isomorphisms in symmetry groups if the symmetry group is given. Promising computations with symmetric set covering problems are given in Margot.

As symmetric integer programs pose significant difficulties for integer-programming solvers, users should try to include symmetry-breaking constraints when modeling a problem. It remains to be seen to what extent automatic detection of symmetry and methods addressing symmetry can be incorporated in integer-programming solvers.

9.2. *Logical statements*

One of the most common uses of binary variables is to represent logical statements such as "if ... is true, then ... holds," "either ... is true or ... is true," "at most/least k activities are positive," "k out of m constraints must hold," etc. If such statements involve continuous variables, one typically introduces auxiliary binary variables to represent the nonconvexities implicit in such statements.

To illustrate, a typical disjunctive constraint that arises in scheduling problems is the following: either the start time of job i is after the completion time of job j or the start time of job j is after the completion time of job i, since both cannot be processed at the same time. If s_i and s_j denote the start times and p_i and p_j the processing times of jobs i and j, then the statement can be written as

$$\text{either } s_i \geq s_j + p_j \quad \text{or} \quad s_j \geq s_i + p_i. \tag{6}$$

Introducing a binary variable x_{ij} that equals 1 if job i precedes job j and 0 otherwise, the disjunctive statement (6) can be put into the form of a MIP as

$$\begin{aligned} s_i &\geq s_j + p_j - M x_{ij}, \\ s_j &\geq s_i + p_i - M(1 - x_{ij}) \end{aligned}$$

where M is a big constant such that $M \geq \max\{\max\{s_j + p_j - s_i\}, \max\{s_i + p_i - s_j\}\}$. The computational difficulty with this "big M" formulation is that the LP relaxation often gives a fractional value for x_{ij} even if the original disjunctive statement (6) is satisfied, which leads to superfluous branching. An alternative approach would be to drop the disjunctive constraints (6) and enforce them by branching on the disjunction only when they are violated.

A similar situation arises in dealing with semi-continuous variables, which are generalizations of 0-1 variables and continuous variables. Variable y is *semi-continuous* if it is required to be either 0 or between two positive bounds $a < b$. A standard way of formulating such non-convex decisions is to introduce an auxiliary binary variable x and specify the domain of variable y with constraints

$$ax \leq y \leq bx. \tag{7}$$

Again, in the LP relaxation even when the requirement $a \leq y \leq b$ is satisfied, x can be fractional, resulting in unnecessary branching. A better approach is to not explicitly model the disjunction, but enforce it by branching, i.e., require only $0 \leq y \leq b$ and enforce the disjunction by branching as $y = 0$ or $a \leq y \leq b$ if it is violated (when $y \in (0, a)$ in the relaxation).

A third example concerns problems with cardinality constraints on continuous variables. Such constraints specify that at least/at most k variables can be positive in any feasible solution. For a set of nonnegative bounded continuous variables, i.e., $0 \leq y_j \leq u_j$, introducing auxiliary binary variables x_j will allow an MIP formulation of a

cardinality constraint as follows

$$\sum_j x_j \geq (\leq)\, k, \quad 0 \leq y_j \leq u_j x_j \ \forall j.$$

Again, the linear programming relaxation of this formulation will tend to be fractional even when the cardinality constraint is satisfied. Effective branching rules that work directly on the continuous variables are given in Bienstock (1996) for this case.

Although small sized instances of formulations involving such auxiliary variables can be solved with the standard IP solvers, specialized branching techniques avoiding the need for auxiliary variables or alternative stronger IP formulations are usually required for solving large-scale instances. Barring a few exceptions[4] the burden of implementing specialized branching schemes and strengthening formulations in these situations is currently on the users of IP solvers.

Recent research on strengthening formulations with logical constraints by means of cutting planes (Farias, Johnson, and Nemhauser, 2001; Farias, Johnson, and Nemhauser, 2003) and automatic reformulations (Codato and Fischetti, 2004) may help to improve the solvability of these problems.

9.3. General integer variables

Most of the theoretical as well as algorithmic research of the past decades has focused on 0-1 and mixed 0-1 programming problems. As a result, many of the cutting planes, specialized search strategies, and preprocessing techniques embedded in integer-programming software are only effective on instances in which integer variables are restricted to values 0 and 1. Methods that work well for mixed 0-1 instances typically do not have generalizations that are as effective for general integer programs. As IP solvers do not have the appropriate tools to attack problems that contain general integer variables effectively, such problems continue to be significantly harder to solve than 0-1 problems. Examples of small integer programs with general integer variables that are very hard to solve with state-of-the-art IP solvers are given in Cornuejols and Dawande (1998), Aardal and Lenstra (2002).

In the hope of overcoming this known shortcoming of IP solvers, users of IP solvers sometimes formulate IPs involving general integer variables as 0-1 programs by using the 0-1 expansion of bounded integer variables. However, this is generally a poor and counterproductive strategy for several reasons, including the symmetry and weak LP bounds of expanded formulations (Owen and Mehrotra, 2002).

Research on polyhedral analysis of general mixed-integer knapsacks (Atamtürk, 2003a) and on lattice–based methods for pure integer programs (Aardal, Hurkens, and Lenstra, 2000; Louveaux and Wolsey, 2002) are ongoing and may lead to more powerful techniques for handling instances with general integer variables.

10. Future

Computational integer programming is a rapidly advancing field. The availability of fast and reliable optimization software systems has made integer programming an effective paradigm for modeling and optimizing strategic, tactical, and operational decisions in organizations in a variety of industries, ranging from finance to health care, from telecommunications to defense.

The adoption of the integer-programming paradigm continues to spread, leading to new applications in areas such as fiber-optic telecommunication network design, radiotherapy, genetics, financial/commodity exchanges. These new applications frequently pose new challenges, which in turn result in improved technology.

The progress of integer-programming software has resulted, to a large extent, from identifying and exploiting problem structure. Automatic classification of the constraints and the effective use of this classification in preprocessing, primal heuristics, and cut generation has significantly increased the power of general purpose integer-programming solvers. (We have seen dramatic improvements for 0-1 and mixed 0-1 problems containing knapsack, set packing, and fixed-charge structures.) We expect to see further exploitation of problem structure by IP solvers in the future.

We anticipate the creation of dedicated integer-programming solvers for common problem classes, such as set-partitioning, fixed-charge network flow, multi–commodity flow, and time-indexed production planning problems. Furthermore, we anticipate the construction of specialized techniques for embedded combinatorial structures that arise frequently, such as disjunctive constraints, cardinality constraints, specially ordered sets, and semi–continuous variables (Farias, Johnson, and Nemhauser, 2001; Farias, Johnson, and Nemhauser, 2003). Some of these structures are already exploited during branching, but we believe they can also be exploited in other components of an integer-programming solver.

As we have seen in the preceding sections, the behavior of the basic branch-and-cut algorithm can be altered dramatically by the parameter settings that control crucial components of the algorithms, such as when to cut, when to branch, how many cuts to add, which node to branch on, etc. It is a rather arduous task to determine the proper parameter settings for a particular problem. Furthermore, what is appropriate at the beginning of the solution process may not be so at later stages. There is a need for integer-programming systems to dynamically adjust parameter settings based on an analysis of the solution process.

Constraint programming has proven to be an effective enumerative paradigm, especially for tightly constrained combinatorial problems. Consequently, the integration of LP based branch-and-cut methods and constraint programming techniques seems desirable. Initial efforts along these lines are promising (Harjunkoski, Jain, and Grossman, 2000; Hooker et al., 2000; Jain and Grossmann, 2001). Partial enumeration at the nodes of branch-and-bound tree using constraint programming and LP reduced-cost information may help to identify good heuristic solutions early in the search process. Constraint

programming may also be useful in column generation subproblems that are modeled as shortest path problems with side constraints.

Today's IP systems allow users to dynamically add constraints to the formulation in the branch-and-bound tree, thus turning the branch-and-bound algorithm into a branch-and-cut algorithm. We expect that in the future they will also allow the users add variables to the formulation in the tree and facilitate easy implementation of branch-and-price algorithms.

In most practical situations, one is usually satisfied with good feasible solutions (preferably provably within a few percent from optimality). Although tremendous amounts of time and effort have been dedicated to developing techniques for finding strong LP relaxations in order to reduce the size of the search tree (and thus solution time), far less work has been done on finding good quality feasible solutions quickly, even though having high quality incumbent solutions early on in the search is equally valuable in pruning the branch-and-bound tree. Most efforts have focused on enhancing search strategies and on relatively simple linear programming based rounding heuristics. Fortunately, several researchers have produced promising computational results using novel ideas, such as local branching (Fischetti and Lodi, 2003) and its variants (Danna, Rothberg, and Pape, 2003; Fischetti, Glover, and Lodi, 2005).

As more powerful IP technology becomes a practical tool to solve real–life optimization problems, there is an increasing demand for approaches that can effectively handle uncertainty of data. The research community is responding to this need by increased efforts in the area of robust and stochastic optimization (Atamtürk, 2003b; Atamtürk and Zhang, 2004; Bertsimas and Sim, 2003; Birge and Louveaux, 1997; van der Vlerk, 2003). These efforts will stimulate advances in integer programming as large-scale structured instances need to be solved efficiently. In the future, integer-programming software may even be enhanced with features specifically designed to handle uncertainty of the objective or technology coefficients.

Appendix A. Lot sizing using Xpress-Mosel

```
model LS                              ! Start a new model

uses ‘‘mmxprs",‘‘mmsystem"            ! Load the optimizer library
forward procedure cutgen              ! Declare a procedure that is
                                      ! defined later
declarations
  EPS=1e-6                            ! Zero tolerance
  NT=6                                ! Number of time periods
  T=1..NT                             ! Range of time

  DEMAND: array(T) of integer         ! Demand per period
  SETUPCOST: array(T) of integer      ! Setup cost per period
```

```
  PRODCOST: array(T) of integer       ! Production cost per period
  D: array(T,T) of integer            ! Total demand in periods t1 - t2

  product: array(T) of mpvar          ! Production in period t
  setup: array(T) of mpvar            ! Setup in period t
end-declarations

DEMAND := [ 1, 3, 5, 3, 4, 2]
SETUPCOST := [17,16,11, 6, 9, 6]
PRODCOST := [ 5, 3, 2, 1, 3, 1]

! Calculate D(.,.) values

forall(s,t in T) D(s,t) := sum(k in s..t) DEMAND(k)

! Objective: minimize total cost

MinCost := sum(t in T) (SETUPCOST(t) * setup(t) + PRODCOST(t) *
product(t))

! Production in periods 0 to t must satisfy the total demand
! during this period of time

forall(t in T) Demand(t) :=
    sum(s in 1..t) product(s) >= sum (s in 1..t) DEMAND(s)

! Production in period t must not exceed the total demand for the
! remaining periods; if there is production during t then there
! is a setup in t

forall(t in T) Production(t) := product(t) <= D(t,NT) * setup(t)

forall(t in T) setup(t) is_binary

! Solve by cut generation

cutgen                                ! Solve by cut generation

! Print solution

forall(t in T)
  writeln("Period ", t,": prod ", getsol(product(t))," (demand: ",
```

```
    DEMAND(t),
        ", cost: ", PRODCOST(t), "), setup ", getsol(setup(t)),
        " (cost: ", SETUPCOST(t), ")")

  !*************************************************************************
  ! Cut generation loop at the top node:
  ! solve the LP and save the basis
  ! get the solution values
  ! identify and set up violated constraints
  ! load the modified problem and load the saved basis
  !*************************************************************************

  procedure cutgen

  declarations
    ncut,npass,npcut: integer           ! Counters for cuts and passes
    solprod,solsetup: array(T) of real  ! Sol. values for var.s product
                                          & setup
    objval,starttime,lhsval: real
    cut: array(range) of linctr
  end-declarations

    starttime := gettime
    setparam("XPRS_CUTSTRATEGY", 0)   ! Disable automatic cuts
    setparam("XPRS_PRESOLVE", 0)      ! Switch presolve off
    ncut := 0
    npass := 0

    repeat
      npass += 1
      npcut := 0
      minimize(XPRS_LIN+XPRS_PRI, MinCost)  ! Solve the LP using primal
                                            simplex
      savebasis(1)                          ! Save the current basis
      objval := getobjval                   ! Get the objective value

      forall(t in T) do                     ! Get the solution values
        solprod(t) := getsol(product(t))
        solsetup(t):= getsol(setup(t))
      end-do

  ! Search for violated constraints:
```

```
    forall(l in T) do
      lhsval := 0
      forall(t in 1..l)
        if(solprod(t) > D(t,l)*solsetup(t) + EPS)
          then lhsval += D(t,l)*solsetup(t)
          else lhsval += solprod(t)

        end-if

      ! Add the violated inequality: the minimum of the actual
        production
      ! prod(t) and the maximum potential production D(t,l)*setup(t)
      ! in periods 1 to l must at least equal the total demand in
        periods
      ! 1 to l.
      !
      ! sum(t=1:l) min(product(t), D(t,l)*setup(t)) >= D(1,l)
      !

      if(lhsval < D(1,l) - EPS) then
        ncut += 1
        npcut += 1
        cut(ncut):= sum(t in 1..l)
          if(solprod(t)<(D(t,l)*solsetup(t))+EPS, product(t), D(t,l)*
            setup(t)) >= D(1,l)
      end-if
    end-do

    if(npcut=0) then
      writeln("Optimal integer solution found:")
    else
      loadprob(MinCost)   ! Reload the problem
      loadbasis(1)        ! Load the saved basis
    end-if
  until (npcut <= 0)

end-procedure
end-model
```

Appendix B. Lot sizing using OPL/OPL Script

```
% OPL model file: ELScuts.mod
```

```
int NT = ...;
range T 1..NT;

int DEMAND[T] =...;
int SETUPCOST[T] = ...;
int PRODCOST[T] = ...;

int D[s in T, t in T] = sum (k in [s..t]) DEMAND[k];

import Open pcut;
import Open scut;
import Open Lval;
int ncuts = pcut.up;

var float+ product[T];
var float+ setup[T] in 0..1;

constraint prodsetup[T];
constraint meetdemand[T];
constraint Cuts[0..ncuts];

minimize sum(t in T) (SETUPCOST[t]*setup[t] + PRODCOST[t] * product[t])
subject to {

  forall(t in T)
    meetdemand[t] : sum(s in 1..t) product[s] >= sum (s in 1..t)
                    DEMAND[s];

  forall(t in T)
    prodsetup[t] : product[t] <= D[t,NT] * setup[t];
  forall(c in 0..ncuts) {
    Cuts[c] : sum (t in pcut[c]) product[t] +
              sum (t in scut[c]) D[t,Lval[c]]*setup[t] >= D[1,Lval[c]];
  };
};

data {
  NT = 6;
  DEMAND = [ 1, 3, 5, 3, 4, 2];
  SETUPCOST = [17,16,11, 6, 9, 6];
  PRODCOST = [ 5, 3, 2, 1, 3, 1];
};
```

```
% OPL script file: ELScuts.osc

int ncuts := 0;
Open setof(int) pcut[int+];
Open setof(int) scut[int+];
Open int Lval[int+];

Model m("ELScuts.mod");

int NT := m.NT;
float eps := 1.0e-6;
int cuts := 0;
float value := 0;
int itcnt := 0;

repeat {
   cuts := 0;
   m.solve();
   cout << " Iter " << itcnt << " Cuts " << ncuts << " Obj " <<
   m.objectiveValue()
         << " Iters " << m.getNumberOfIterations() << endl;
   forall(l in 1..NT) {
      value := 0;
      setof(int) Scut := t |t in 1..l: m.product[t] > m.D[t,l]
      *m.setup[t] + eps;
      setof(int) Pcut := t |t in 1..l  diff Scut;
      value := sum (t in Pcut) m.product[t] + sum(t in Scut) m.D[t,l]
      *m.setup[t];
      if (value < m.D[1,l] - eps) then {
         cuts := cuts + 1;
         ncuts := ncuts + 1;
         pcut.addh();
         scut.addh();
         Lval.addh();
         pcut[ncuts-1] := Pcut;
         scut[ncuts-1] := Scut;
         Lval[ncuts-1] := l;
      }
   }
   Basis b(m);
   if (cuts > 0) then {
      m.reset();
```

```
      m.setBasis(b);
   }
   itcnt := itcnt + 1;
}UNTIL (CUTS = 0);

forall(t in 1..NT) {
   cout << "Time " << t << " product " << m.product[t] << " setup "
   << m.setup[t] << endl;
}
```

Appendix C. Lot sizing using Xpress-BCL

```
#include <stdio.h>
#include "xprb_cpp.h"
#include "xprs.h"

using namespace ::dashoptimization;

#define EPS 1e-6
#define T 6                          /* Number of time periods */

/****DATA****/
int DEMAND[] =  1, 3, 5, 3, 4, 2;    /* Demand per period */
int SETUPCOST[] = 17,16,11, 6, 9, 6; /* Setup cost per period */
int PRODCOST[] =  5, 3, 2, 1, 3, 1;  /* Production cost per period */
int D[T][T];                         /* Total demand in periods t1 -
                                     t2 */
XPRBvar prod[T];                     /* Production in period t */
XPRBvar setup[T];                    /* Setup in period t */
XPRBprob p("Els");                   /* Initialize a new problem in
                                      BCL */

void modEls() {
 int s,t,k;
 XPRBlinExp cobj,le;

 for(s=0;s<T;s++)
  for(t=0;t<T;t++)
   for(k=s;k<=t;k++)
    D[s][t] += DEMAND[k];

/****VARIABLES****/
```

```
 for(t=0;t<T;t++) {
  prod[t]=p.newVar(xbnewname("prod%d",t+1));
  setup[t]=p.newVar(xbnewname("setup%d",t+1), XPRB_BV);
 }

/****OBJECTIVE****/
for(t=0;t<T;t++)
 cobj += SETUPCOST[t]*setup[t] + PRODCOST[t]*prod[t];
p.setObj(cobj);

/****CONSTRAINTS****/
         /* Production in period t must not exceed the total demand for
            the remaining periods; if there is production during t then
            there is a setup in t */
for(t=0;t<T;t++)
p.newCtr("Production", prod[t] <= D[t][T-1]*setup[t]);

         /* Production in periods 0 to t must satisfy the total demand
           during this period of time */
 for(t=0;t<T;t++)
 {
 le=0;
 for(s=0;s<=t;s++) le += prod[s];
 p.newCtr("Demand", le >= D[0][t]);
 }
}

/* Cut generation loop at the top node:
      solve the LP and save the basis
      get the solution values
      identify and set up violated constraints
      load the modified problem and load the saved basis
*/

void solveEls() {
 double objval;                  /* Objective value */
 int t,l;
 int starttime;
 int ncut, npass, npcut;         /* Counters for cuts and passes */
 double solprod[T], solsetup[T]; /* Solution values for var.s prod &
                                    setup */
 double ds;
```

```
 XPRBbasis basis;
 XPRBlinExp le;

 starttime=XPRB::getTime();
 XPRSsetintcontrol(p.getXPRSprob(), XPRS_CUTSTRATEGY, 0);
                  /* Disable automatic cuts - we use our own
*/
 XPRSsetintcontrol(p.getXPRSprob(), XPRS_PRESOLVE, 0);
                  /* Switch presolve off */
 ncut = npass = 0;

 do
 {
 npass++;
 npcut = 0;
 p.solve("lp");          /* Solve the LP */
 basis = p.saveBasis();  /* Save the current basis */
 objval = p.getObjVal(); /* Get the objective value */

/* Get the solution values: */
 for(t=0;t<T;t++)
 {
 solprod[t]=prod[t].getSol();
 solsetup[t]=setup[t].getSol();
 }

/* Search for violated constraints: */
 for(l=0;l<T;l++)
 {
 for (ds=0.0, t=0; t<=l; t++)
 {
  if(solprod[t] < D[t][l]*solsetup[t] + EPS) ds += solprod[t];
  else ds += D[t][l]*solsetup[t];
  }

   /* Add the violated inequality: the minimum of the actual production
   prod[t] and the maximum potential production D[t][l]*setup[t]
   in periods 0 to l must at least equal the total demand in periods
   0 to l.
   sum(t=1:l) min(prod[t], D[t][l]*setup[t]) >= D[0][l]
  */
 if(ds < D[0][l] - EPS)
```

```
 {
 le=0;
 for(t=0;t<=l;t++)
 {
 if (solprod[t] < D[t][l]*solsetup[t] + EPS)
 le += prod[t];
 else
 le += D[t][l]*setup[t];
 }
 p.newCtr(xbnewname("cut%d",ncut+1), le >= D[0][l]);
 ncut++;
 npcut++;
 }
}

printf("Pass %d (%g sec), objective value %g, cuts added: %d
(total %d)\n",
 npass, (XPRB::getTime()-starttime)/1000.0, objval, npcut, ncut);

if(npcut==0)
 printf("Optimal integer solution found:\n");
else
{
 p.loadMat();                /* Reload the problem */
 p.loadBasis(basis);         /* Load the saved basis */
 basis.reset();              /* No need to keep the basis any longer */
 }
}WHILE(NPCUT>0);

/* Print out the solution: */
 for(t=0;t<T;t++)
 printf("Period %d: prod %g (demand: %d, cost: %d), setup %g
 (cost: %d)\n",
     t+1, prod[t].getSol(), DEMAND[t], PRODCOST[t], setup[t].getSol(),
     SETUPCOST[t]);
}

 int main(int argc, char **argv) {
 modEls();                       /* Model the problem */
 solveEls();                     /* Solve the problem */
}
```

Appendix D. Lot sizing using ILOG Concert

```
#include <ilcplex/ilocplex.h>
ILOSTLBEGIN
typedef IloArray<IloNumArray> NumArray2;

int main(int argc, char **argv) {
 IloEnv env;
 try {
  IloInt s,t,k;

  IloInt NT = 6;
  IloNumArray DEMAND(env, NT, 1, 3, 5, 3, 4, 2);
  IloNumArray SETUPCOST(env, NT, 17, 16, 11, 6, 9, 6);
  IloNumArray PRODCOST(env, NT, 5, 3, 2, 1, 3, 1);

  IloNumVarArray product(env, NT, 0, IloInfinity, ILOFLOAT);
  IloNumVarArray setup(env, NT, 0, 1, ILOINT);
  NumArray2 D(env, NT);
  IloModel model(env);

  for(s = 0; s < NT; s++){
   D[s] = IloNumArray(env, NT);
   for(t = 0; t < NT; t++)
    for(k = s; k <= t; k++)
         D[s][t] += DEMAND[k];
  }
  IloExpr obj(env);
  obj = IloScalProd(SETUPCOST,setup) + IloScalProd(PRODCOST,product);
  model.add(IloMinimize(env, obj));

  for(t = 0; t < NT; t++){
   IloExpr lhs(env), rhs(env);
   for( s = 0; s <= t; s++){
    lhs += product[s];
    rhs += DEMAND[s];
 }
 model.add( lhs >= rhs );
 lhs.end(); rhs.end();
}
for(t = 0; t < NT; t++)
 model.add( product[t] <= D[t][NT-1]*setup[t] );
```

```
IloCplex cplex(model);
IloNum eps = cplex.getParam(IloCplex::EpInt);
model.add(IloConversion(env, setup, ILOFLOAT));

ofstream logfile("cplex.log");
cplex.setOut(logfile);
cplex.setWarning(logfile);

IloInt cuts;
do {
 cuts = 0;
 if( !cplex.solve() ){
  env.error() << "Failed" << endl;
  throw(-1);
  }
  env.out() << "Objective: " << cplex.getObjValue() << endl;
  for(IloInt l = 0; l < NT; l++){
   IloExpr lhs(env);
   IloNum value = 0;
   for(t = 0; t <= l; t++){
        if( cplex.getValue(product[t]) > D[t][l] * cplex.getValue
        (setup[t])+eps ){
          lhs += D[t][l] * setup[t];
          value += D[t][l]* cplex.getValue( setup[t] );
        }
        else {
          lhs += product[t];
          value += cplex.getValue( product[t] );
        }
   }
   if( value < D[0][l]-eps ){
        model.add( lhs >= D[0][l] );
        env.out() << "** add cut " << endl;
        cuts++;
   }
   lhs.end();
  }
 }WHILE (CUTS);
 env.out() << endl << "Optimal value: " << cplex.getObjValue() << endl;
 for(t = 0; t < NT; t++)
  if (cplex.getValue(setup[t]) >= 1 - eps)
   env.out() << "At time " << t << " produced " <<
```

```
        cplex.getValue(product[t]) << " " << cplex.getValue(setup[t])
        << endl;
   }
   catch(IloException& e) {
    cerr << " ERROR: " << e << endl;
   }
   catch(...) {
    cerr << " ERROR" << endl;
   }
   env.end();
   return 0;
  }
```

Acknowledgment

We are grateful to Lloyd Clark (ILOG, Inc.), Linus Schrage (LINDO Systems, Inc.), and James Tebboth (Dash Optimization) for their help and to two anonymous referees for their valuable suggestions for improving the original version of the paper.

Notes

1. CPLEX is a trademark of ILOG, Inc.
2. LINDO is a trademark of LINDO Systems, Inc.
3. Xpress-MP is a trademark of Dash Optimization Ltd.
4. XPRESS supports specialized branching for semi-continuous variables.

References

Aardal, K., C.A.J. Hurkens, and A.K. Lenstra. (2000). "Solving a System of Linear Diophantine Equations with Lower and Upper Bounds on the Variables." *Mathematics of Operations Research* 25(3), 427–442.

Aardal, K. and A. Lenstra. (2002). "Hard Equality Constrained Integer Knapsacks." In W. Cook and A. Schultz (eds.), *Proc. 9th International IPCO Conference* Springer-Verlag, pp. 350–366.

Anderson, E.D. and K.D. Anderson. (1995). "Presolving in Linear Programming." *Mathematical Programming* 71, 221–225.

Atamtürk, A. (2003a). "On the Facets of Mixed-Integer Knapsack Polyhedron." *Mathematical Programming* 98, 145–175.

Atamtürk, A. (2003b). "Strong Formulations of Robust Mixed 0-1 Programming." Research Report BCOL.03.04. Available at `http://ieor.berkeley.edu/~atamturk` (To appear in *Mathematical Programming*).

Atamtürk, A. and M. Zhang. (2004). "Two-Stage Robust Network Flow and Design for Demand Uncertainty." Research Report BCOL.04.03. Available at `http://ieor.berkeley.edu/~atamturk`.

Balas, E. (1975). "Facets of the Knapsack Polytope." *Mathematical Programming* 8, 146–164.

Balas, E., S. Ceria, G. Cornuéjols, and N. Natraj. (1996). "Gomory Cuts Revisited." *Operations Research Letters* 19, 1–9.

Balas, E. and E. Zemel. (1978). "Facets of the Knapsack Polytope from Minimal Covers." *SIAM Journal of Applied Mathematics* 34, 119–148.

Barany, I., T.J. Van Roy, and L.A. Wolsey. (1984). "Uncapacitated lot Sizing: The Convex Hull of Solutions." *Mathematical Programming Study* 22, 32–43.

Beale, E.M.L. (1979). "Branch and Bound Methods for Mathematical Programming Systems." In P.L. Hammer, E.L. Johnson, and B.H. Korte (eds.), *Discrete Optimization II*, North Holland Publishing Co, pp. 201–219.

Bénichou, M., J.M. Gauthier, P. Girodet, G. Hentges, G. Ribière, and O. Vincent. (1971). "Experiments in Mixed-Integer Linear Programming." *Mathematical Programming* 1, 76–94.

Bertsimas, D. and M. Sim. (2003). "Robust Discrete Optimization and Network Flows." *Mathematical Programming* 98, 49–71.

Bienstock, D. (1996). "Computational Study of a Family of Mixed-Integer Quadratic Programming Problems." *Mathematical Programming* 74, 121–140.

Birge, J.R. and F. Louveaux. (1997). *Introduction to Stochastic Programming*. New York: Springer Verlag.

Bixby, R.E., M. Fenelon, Z. Gu, E. Rothberg, and R. Wunderling. (2002). Mixed–integer programming: A progress report.

Codato, G. and M. Fischetti. (2004). "Combinatorial Benders' Cuts." In D. Bienstock, and G.L. Nemhauser (eds.), *Proc. 10th International IPCO Conference* Springer-Verlag; pp. 178–195.

Cornuejols, G. and M. Dawande. (1998). "A Class of Hard Small 0-1 Programs." In R.E. Bixby, E.A. Boyd, and R.Z. Rios-Mercado (eds.), *Proc. 6th International IPCO Conference* Springer-Verlag, pp. 284–293.

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

Danna, E., E. Rothberg, and C.L. Pape. (2003). "Exploring Relaxation Induced Neighborhoods to Improve MIP Solutions." Technical report, ILOG, Inc.

Dash Optimization, L. (2004a). *Proctor and Gamble Case Study*.

Dash Optimization, L. (2004b). *XPRESS-BCL Reference Manual—Release 2.6*.

Dash Optimization, L. (2004c). *XPRESS-Mosel Language Reference Manual—Release 1.4*.

Dash Optimization, L. (2004d). *XPRESS-Optimizer Reference Manual—Release 15*.

Farias, I.R.D., E.L. Johnson, and G.L. Nemhauser. (2001). "Branch-and-Cut for Combinatorial Optimisation Problems without Auxiliary Binary Variables." *Knowledge Engineering Review* 16, 25–39.

Farias, I.R.D., E.L. Johnson, and G.L. Nemhauser. (2003). "A Polyhedral Study of the Cardinality Constrained Knapsack Problem." *Mathematical Programming* 96, 439–467.

Fischetti, M. and A. Lodi. (2003). "Local Branching." *Mathematical Programming* 98, 23–47.

Fischetti, M., F. Glover, and A. Lodi. (2005). 'The Feasibility Pump." *Mathematical Programming* 104, 91–104.

Forrest, J.J.H., J.P.H. Hirst, and J.A. Tomlin. (1974). "Practical Solution of Large Scale Mixed Integer Programming Problems with UMPIRE." *Management Science* 20, 736–773.

Gomory, R.E. (1960). "An Algorithm for the Mixed Integer Problem." Technical Report RM-2597, The Rand Corporation.

Gondzio, J. (1997). "Presolve Analysis of Linear Programs Prior to Applying an Interior Point Method." *INFORMS Journal on Computing* 9, 73–91.

Gu, Z., G.L. Nemhauser, and M.W.P. Savelsbergh. (1998). "Lifted Cover Inequalities for 0-1 Integer Programs: Computation." *INFORMS Journal on Computing* 10, 427–437.

Gu, Z., G.L. Nemhauser, and M.W.P. Savelsbergh. (1999). "Lifted Flow Cover Inequalities for Mixed 0-1 Integer Programs." *Mathematical Programming* 85, 439–467.

Guignard, M. and K. Spielberg. (1981). "Logical Reduction Methods in Zero–One Programming." *Operations Research* 29, 49–74.

Hammer, P.L., E.L. Johnson, and U.N. Peled. (1975). "Facets of Regular 0-1 Polytopes." *Mathematical Programming* 8, 179–206.

Harjunkoski, I., V. Jain, and I.E. Grossman. (2000). "Hybrid Mixed-Integer/Constraint Logic Programming Strategies for Solving Scheduling and Combinatorial Optimization Problems." *Computers and Chemical Engineering* 24, 337–343.

Hirst, J.P.H. (1969). "Features Required in Branch and Bound Algorithms for (0-1) Mixed Integer Linear Programming." Privately circulated manuscript.

Hooker, J.N., G. Ottosson, E.S. Thornsteinsson, and H.-J. Kim. (2000). "A Scheme for Unifying Optimization and Constraint Satisfaction Methods." *Knowledge Engineering Review* 15, 11–30.

ILOG, I. (2002). *ILOG OPL User's Manual*.

ILOG, I. (2003). *ILOG CPLEX 9.0 Reference Manual*.

Jain, V. and I.E. Grossmann. (2001). "Algorithms for Hybrid MILP/CP Methods." *INFORMS Journal on Computing* 13, 258–276.

Land, A. and S. Powell. (1979). "Computer Codes for Problems of Integer Programming." In P.L. Hammer, E.L. Johnson, and B.H. Korte (eds.), *Discrete Optimization II* North Holland Publishing Co, pp. 221–269.

LINDO Systems, I. (2002). *LINDO API User's Manual*.

Louveaux, Q. and L.A. Wolsey. (2002). "Combining Problem Structure with Basis Reduction to Solve a Class of Hard Integer Programs." 27, 470–484.

Marchand, H. and L.A. Wolsey. (2001). "Aggregation and Mixed Integer Rounding to Solve MIPs." *Operations Research* 49, 363–371.

Margot, F. (2003). "Exploiting Orbits in Symmetric Ilp." *Mathematical Programming* 98, 3–21.

Martin, A., T. Achterberg, and T. Koch. (2003). MIPLIB 2003. http://miplib.zib.de/.

Mitra, G. (1973). "Investigation of Some Branch and Bound Strategies for the Solution of Mixed Integer Linear Programs." *Mathematical Programming* 4, 155–170.

Mittelman, H. (2002). "Benchmarks for Optimization Software." http://plato.asu.edu/bench.html.

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

Nemhauser, G.L. and L.A. Wolsey. (1990). "A Recursive Procedure for Generating all Cuts for 0-1 Mixed Integer Programs." *Mathematical Programming* 46, 379–390.

Owen, J.H. and S. Mehrotra. (2002). "On the Value of Binary Expansions for General Mixed-Integer Linear Programs." *Operations Research* 50(5), 810–819.

Padberg, M.W. (1973). "On the Facial Structure of Set Packing Polyhedra." *Mathematical Programming* 5, 199–215.

Padberg, M.W. (1979). "Covering, Packing and Knapsack Problems." *Annals of Discrete Mathematics* 4, 265–287.

Padberg, M.W., T.J.V. Roy, and L.A. Wolsey. (1985). "Valid Linear Inequalities for Fixed Charge Problems." *Operations Research* 33, 842–861.

Savelsbergh, M.W.P. (1994). "Preprocessing and Probing Techniques for Mixed Integer Programming Problems." *ORSA Journal on Computing* 6, 445–454.

Sherali, H.D. and J.C. Smith. (2001). "Improving Discrete Model Representations via Symmetry Considerations." *Management Science* 47, 1396–1407.

van der Vlerk, M.H. (1996–2003). "Stochastic Integer Programming Bibliography." `http://mally.eco.rug.nl/biblio/stoprog.html`.

Van Roy, T.J. and L.A. Wolsey. (1987). "Solving Mixed Integer Programming Problems using Automatic Reformulation." *Operations Research* 35, 45–57.

Wolsey, L.A. (1998). *Integer Programming*. New York: John Wiley & Sons.