INTEGER PROGRAMMING SOFTWARE SYSTEMS

ALPER ATAMTÜRK AND MARTIN W. P. SAVELSBERGH

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 to adjust the behavior of these solvers when default settings do not achieve the desired performance level. Furthermore, we highlight advances towards integrated modeling and solving environments. We conclude with a discussion of model characteristics and substructures that pose challenges for modern integer programming software systems and a perspective on features we may expect to see in these systems in the near future.

1. Introduction

In the last decade, the use of integer programming 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 for any state–of–the–art integer programming system is a linear–programming based branch–and–bound algorithm. Today's 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 changing parameter settings on instances from the MIPLIB library [13].

Although many integer programs can be solved well by finetuning parameter settings, there remain situation in which this is not sufficient. In such situations, it may be necessary to develop decomposition or iterative approaches, in which several models representing different subproblems

Date: January 2003, October 2003.

[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.

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 users can turn the basic branch–and–bound algorithm into a customized solver that, for instance, performs specialized branching, dynamically adds problem–specific cutting planes or columns, incorporates specialized heuristics to find feasible solutions.

Customized problem solving with integer programming solvers can be done at several levels of detail. 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 to users of integer programming systems and provide complete implementations of a simple cutting plane algorithm using Dash Optimization's Xpress–Mosel [18] and Xpress–BCL [17], and ILOG's OPL/OPL script [33] and Concert Technology [32] as examples.

At this point, we want to point out and 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" (http://plato.asu.edu/bench.html) maintained by Hans Mittelman. 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.

Even though tremendous progress has been made over the past 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 also points to potential research topics in the area of integer programming.

The 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 knowledgable choices, to students who are relatively new to the topic of integer programming and 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 2, we describe the linear–programming based branch–and–bound algorithm that forms the basis of integer programming solvers. In Sections 3–6, we describe Important algorithmic components of modern IP solvers: search, preprocessing, cut generation, and primal heuristics. In Section 3, we discuss the choices for branching and their effect on the search. In Section 4, we present several relatively simple and computationally efficient techniques for strengthening the formulations and reducing their size. In Section 5, we discuss issues related to cut generation and management. In Section 6, we review techniques available for quickly finding feasible solutions during the search. In each of the sections 3–6, we present computations that illustrate the impact of different parameter settings that control the corresponding component. We present experiments on MIPLIB [13] instances for which changing the respective parameters have the most noticeable impact. In Section 7, we discuss the options provided by IP solvers for developing customized solution approaches and review the trend to integrate modeling and optimization more closely. In Section 8, we discuss model characteristics and substructures that pose challenges for modern integer programming systems, and finally, in Section 9, we conclude with a perspective on features that we may see appearing in integer programming systems in the near future.

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

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

where c, d, A, G, and b are rational matrices with appropriate dimensions. Here $cx + dy$ is the *objective function* and $Ax + Gy \leq b$ are the *constraints* of MIP. A point $(x, y) \in \mathbb{Z}^n \times \mathbb{R}^m$ is said to be *feasible* if it satisfies the constraints. An *optimal solution* is a feasible point for which the objective function achieves the smallest 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 $x \in \mathbb{R}^n_+$. 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. These statements are true also for *subproblems* of MIP, that are obtained by restricting the feasible set to a subset by adding additional constraints.

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 a branch–and–bound algorithm. Therefore, we briefly describe the basic linear–programming based branch–and–bound algorithm for solving MIP. For a comprehensive exposition on integer programming algorithms we refer the reader to [41, 50].

The branch–and–bound algorithm builds a search tree, in which the nodes of the tree represent subproblems defined over subsets of the feasible set. The objective function value of any feasible solution to MIP provides an upper bound on the optimal value. The feasible solution with the smallest objective value found so far is called the *incumbent solution.* Let z^{best} be the objective value of the incumbent solution. Let MIP(k) denote the subproblem at node k of the search tree. The root node of the search tree is associated with the original problem MIP (or MIP(0)). At node k of the tree we solve the LP relaxation of MIP(k). For ease of presentation, we assume that the LP relaxation is feasible and bounded and has an optimal solution x^k with objective value z^k. If x^k satisfies the integrality constraints, then we have found a feasible solution to MIP and we update z^{best} as $\min\{z^k, z^{best}\}$. Otherwise, there are two possibilities: if $z^k \geq z^{best}$, then an optimal solution to MIP(k) cannot improve on z^{best}, hence the subproblem MIP(k) can be removed from consideration; on the other hand, if $z^k < z^{best}$, then MIP(k) requires further exploration, which is done by branching, i.e., by creating $q \geq 2$ new subproblems (children) MIP($k(i)$), $i = 1, 2, \ldots, q$, of MIP(k). Each feasible solution to MIP(k) must be feasible to at least one child and, conversely, each feasible solution to a child must be feasible to MIP(k). Moreover, the solution x^k must not be feasible to the LP relaxations of any child. A simple realization of these requirements is obtained by selecting a variable x_i with fractional x^k_i and creating two children; in one child 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. The children of node k corresponding to these subproblems are added to the tree, and the branch–and–bound tree grows.

An overview of the basic linear–programming based branch–and–bound algorithm is given in Algorithm 1.

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

```
0. Initialize.
   $\mathbf{L} = \{\text{MIP}\}$. $z^{best} = \infty$. $x^{best} = \emptyset$.
1. Terminate?
   Is $\mathbf{L} = \emptyset$? If so, the solution $x^{best}$ is optimal.
2. Select.
   Choose and delete a problem MIP($k$) from $\mathbf{L}$.
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 $z^k \geq z^{best}$, go to step 1. If $x^k$ does not satisfy the integrality restriction for MIP,
   go to step 5, else let $z^{best} = z^k$, $x^{best} = x^k$, and delete from $\mathbf{L}$ all problems $i$ with
   $z^i \geq z^{best}$. Go to Step 1.
5. Branch.
   Divide the feasible set $S^k$ of MIP($k$) into a number of smaller set $S^{k(i)}$ for $i = 1, \ldots, q$,
   such that $\cup_{i=1}^{q} S^{k(i)} = S^k$. For each $i = 1, \ldots, q$, let $z^{k(i)} = z^k$ and add the problem
   MIP($k(i)$) to $\mathbf{L}$. Go to Step 1.
```

The branch–and–bound algorithm is rendered inefficient if many branchings lead to excessively large search trees. Branching at some node k can be avoided if $z^k \geq z^{best}$. What control do we have here? For one, we can apply fast heuristics to try to find feasible solutions that reduce z^{best}. A complementary approach is to add a constraint, called a *cut*, to LP(k) that has the property that x^k is not satisfied by the cut, but every feasible solution to MIP(k) is. Adding such cuts may increase the LP bound z^k and lead to significantly smaller trees. Finally, the LP relaxations of different formulations of a problem can be vastly different in terms of the quality of the bounds they provide. One may control the size of the search tree by formulating problems in such a way that corresponding LP relaxations provide strong bounds.

One of the most important algorithmic advances of the past decade in solving MIPs is the incorporation of cutting planes in integer programming solvers. The effect of cutting planes is to increase LP bounds and consequently to reduce the size of the branch–and–bound trees. Although the time required to process a node increases with the introduction of cutting planes, the overall computing time can be reduced significantly when cutting planes are added judiciously.

In a survey written more than thirty years ago, Geoffrion and Marsten [23] already predicted the current state of the art and recognized the importance of incorporating cutting planes into the branch–and–bound algorithm:

> "... Enumerative algorithms have received the lion's share of attention in recent years, especially measured by new implementations. This is due partly to the disillusionment with the erratic computational performance of the early cutting–plane algorithms... Recent computational experience seems to vindicate this emphasis ..."
>
> "... the synthesis of the cutting–plane approach with enumeration, a topic that has thus far received less attention than it deserves ..."
>
> "... for each candidate problem, persist in generating cuts up to a certain number of times in succession, or until the rate of improvement due to adding new cuts falls below some threshold value. This rule is appealing because cutting–plane algorithms typically make the greatest rate of progress during the early iterations. This means that there is a chance of actually solving within a few

cuts a candidate problem unfathomable by standard means, and in any case a better bound on its optimal value is generated each time a cut is added—which improves the chances of fathoming the candidate problem based on bounds ..."

3. Search strategies

As discussed above, at the heart of a linear–programming based branch–and–bound is a scheme to divide the feasible region into smaller regions such that (1) the current linear programming solution does not belong to any one of the new regions, and (2) an optimal integer solutions belongs to one of the regions. This is typically accomplished by changing the bounds of the variables.

3.1. **Variable Dichotomy.** 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 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 select 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.

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 use 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

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

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 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 pseudo–costs to choose among possible variables to branch on, it is also possible to simply perform a small number of pivots and observe what happens to the objective function. This is the idea behind *strong branching*. To keep the computational requirements of strong branching under control, typically only a number of dual simplex pivots is performed for only a small set of “promising” variables. Strong branching has been shown to be an effective branching rule for large set partitioning problems, traveling salesman problems, and some general integer programs.

3.2. **Node Selection.** Next, we discuss the node (subproblem) selection component of a branch–and–bound algorithm. When we make a branching decision, we are concerned with maximizing the increase in the objective value of the LP relaxation between a node and its children. However, in selecting a node, our purpose is either to find good integer feasible solutions, or to prove that there are no solutions with a smaller objective value than of the incumbent solution. Therefore, the objective value of the incumbent solution (z^{best}) is an important factor in determining which node to select for evaluation.

A popular method of selecting a node to explore is to choose the one with the lowest z^k, known as the *best–first* or *best–bound* search. For a fixed branching rule, bestfirst search minimizes the number of evaluated nodes before completing the search. At the other extreme is a selection method called *depth–first* search. As the name suggests, the nodes (subproblems) are ordered according to their depth in the tree, where depth of a node is the number of its predecessors up to the root node, and a deepest node is processed first.

Both methods have inherent strengths and weaknesses. Best–first search will result in exploration of the smallest possible number of nodes. At any point during the search, best–first attempts to improve the global lower bound on the optimal value. However, the search tree tends to be explored in a breadth–first fashion, so subsequent linear programs have little relation to each other—leading to longer evaluation times. The time to solve LPs at each node can be reduced by starting from the optimal basis of the parent node, but this requires saving basis information at all the nodes. Consequently, memory requirements for searching the tree in a best–first manner may become prohibitive.

Depth–first search overcomes the shortcomings of best–first search. Searching the tree in a depth–first manner has low memory requirement, and the changes in the linear program from one node to the next are minimal—just the bound of a variable. Depth–first search also tends to find feasible solutions more quickly, as feasible solutions are typically found deep in the search tree. Despite its advantages, depth–first search can lead to the evaluation of many nodes that would have been fathomed, had we known a smaller z^{best}.

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

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 [30] and Mitra [40] and the *best–estimate criterion* found in Bénichou *et al.* [9] and Forrest *et al.* [22], 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

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

The value $(z^{best} - z^0)/s^0$ 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 pseudocosts in obtaining an estimate. This extension is known as the best–estimate criterion. Here, the estimate of the best solution obtainable from a node is defined as

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

This estimate has the advantage that it does not require an upper bound z^{best}.

Forrest, Hirst and Tomlin [22] and Beale [8] 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

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

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.

Let z^{opt} be the optimal objective value for MIP. We say that node k is *superfluous* if $z^k > z^{opt}$. Searching the tree in a best–first manner ensures that no superfluous nodes will be evaluated. If, however, one can be assured that all (or most) of the superfluous nodes will be fathomed, the memory and speed advantages of depth–first search make this method the most preferable.

As branching on a variable creates two nodes for evaluation, a node selection must also rank these nodes for 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 depth–first search, we have to resort to other means. If we branch on variable x_i, then we select the down node first if $f_i < 1 - f_i$ and the up node first otherwise [35].

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 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.

3.3. **Software branching options.**

3.3.1. *CPLEX.* One of the most important factors influencing the choice of branching 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,
- favor feasibility,
- favor optimality,
- best bound (just optimality).

If desirable, CPLEX allows more detailed control over the solution process. 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 pivots,
- pseudo reduced costs: a computationally less–intensive version of pseudo–costs.

Users can also prioritize the variables for branching. Node selection rules in CPLEX can be set using the parameter `mip strategy nodeselect`. Allowed options are

- best–bound search: the node with the best objective function value will be selected
- best–estimate search: a node's progress toward 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.

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

For an extended description of CPLEX user options the reader is referred to [32].

3.3.2. *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.

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 [36].

3.3.3. *Xpress–Optimizer.* The tree search in Xpress–Optimizer is controlled by three parameters: `nodeselection`, `backtrack`, and `varselection`.

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 nodes active nodes.

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

The setting of `backtrack` determines how the next node in the tree search is selected for processing. Backtracking is done in any one of the following ways:

- 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).

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:

- the minimum of up and down pseudo–cost (default),
- 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–Optimizer 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).

3.4. **Sample computations on search strategies.** To demonstrate the effect of the search strategy on the overall solution process, we have conducted the following experiments. We have used Xpress–Optimizer 14.24 to solve MIPLIB problems with different settings of control parameters.

In the first experiment, we compare different settings of the `backtrack` parameter. The results of the experiments for 10 instances are summarized in Table 1. The maximum CPU time was set to 1800 seconds and we present the time taken to find the first feasible solution, the value of the first feasible solution, the time taken to find the best feasible solution, the value of the best feasible solution, and the number of feasible solutions found. Note that some instances were solved to optimality within the given time limit, but that others were terminated prematurely. The most striking observation is that with best–estimate search many more feasible solutions are found during the search. This is most clearly seen with the instance *harp2* where 95 improving feasible solutions are found in the first 60 seconds with best–estimate search. On the other hand, both the default search strategy and the Forrest–Hirst–Tomlin search strategy are able to find the optimal solution for instance *harp2* and prove its optimality within the time lime of 1800 seconds, whereas best–estimate search is unable to do so. Best–estimate search was able to find a better feasible solution for instance *seymour* than default search and Forrest–Hirst–Tomlin search. For the instances where all three search strategies find the same best solution, the default strategy finds it first three times, the best–estimate search strategy finds it first four times, and the Forrest–Hirst–Tomlin search strategy finds it first once.

In the second experiment, we compare different settings of the `nodeselection` parameter. The results of the experiments are summarized in Table 2. The most striking observation is that depth-first search is unable to find a single feasible solution for two instances (*seymour* and *harp2*). There does not seem to be a significant difference in performance between the default node selection strategy and the local depth first node selection strategy, except on instance *seymour* where we find a better solution when employing local depth first node selection.

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 the combination of parameters may not work well on a wide variety of instances, they may work extremely well on the class of problems that need to be solved.

	default–bound					best–estimate					Forrest–Hirst–Tomlin				
problem	time first	value first	time best	value best	#	time first	value first	time best	value best	#	time first	value first	time best	value best	#
misc07	0	3,625.00	1	2,810.00	4	0	3,395.00	6	2,810.00	9	0	3,625.00	4	2,810.00	7
cap6000	12	-2,449,548.00	53	-2,451,377.00	9	12	-2,449,548.00	108	-2,451,377.00	24	12	-2,449,548.00	158	-2,451,377.00	17
danoint	3	78.25	17	65.67	3	3	78.25	63	65.67	4	3	78.25	140	65.67	4
seymour	80	432.00	138	427.00	5	84	434.00	996	426.00	9	81	432.00	976	430.00	4
pk1	0	24.00	47	11.00	7	0	80.00	10	11.00	10	0	24.00	134	11.00	7
mod011	3	-51,776,143.50	34	-54,558,535.01	3	3	-50,225,405.58	32	-54,558,535.01	12	3	-51,776,143.50	107	-54,422,236.85	9
qiu	1	1,577.11	48	-132.87	6	2	1,577.11	12	-132.87	12	2	1,577.11	6	-132.87	8
10teams	32	1,016.00	63	924.00	3	11	984.00	15	924.00	3	8	976.00	28	924.00	6
bell5	0	9,062,387.62	17	8,966,406.49	7	0	9,085,845.67	440	8,966,406.49	6	0	9,062,387.62	142	8,966,406.49	8
harp2	1	-70,886,031.00	435	-73,899,778.00	35	1	-70,768,682.00	52	-73,855,566.00	95	1	-70,886,031.00	1318	-73,899,780.00	575

Table 1. Impact of backtracking strategy

	default					best–first					local depth–first				
problem	time first	value first	time best	value best	#	time first	value first	time best	value best	#	time first	value first	time best	value best	#
misc07	0	3,625.00	1	2,810.00	4	14	2,995.00	20	2,810.00	3	0	3,625.00	10	2,810.00	6
cap6000	12	-2,449,548.00	53	-2,451,377.00	9	12	-2,449,548.00	192	-2,451,377.00	7	12	-2,449,548.00	48	-2,451,377.00	8
danoint	3	78.25	17	65.67	3	1146	70.50	1585	70.00	2	3	78.25	122	65.67	4
seymour	80	432.00	138	427.00	5	-	-	-	-	0	81	432.00	922	426.00	5
pk1	0	24.00	47	11.00	7	946	11.00	946	11.00	1	0	24.00	59	11.00	7
mod011	3	-51,776,143.50	34	-54,558,535.01	3	524	-54,422,236.85	524	-54,422,236.85	1	3	-51,776,143.50	50	-54,558,535.01	4
qiu	1	1,577.11	48	-132.87	6	5	1,691.14	153	-132.87	3	2	1,577.10	10	-132.87	6
10teams	32	1,016.00	63	924.00	3	55	924.00	55	924.00	1	8	928.00	99	924.00	2
bell5	0	9,062,387.62	17	8,966,406.49	7	2	8,966,933.66	72	8,966,406.49	2	0	9,062,387.62	15	8,966,406.49	12
harp2	1	-70,886,031.00	435	-73,899,778.00	35	-	-	-	-	0	1	-70,886,031.00	415	-73,899,774.00	33

TABLE 2. Impact of node selection strategy

4. Preprocessing

In order to reduce the size of an instance and to strengthen its LP bound, IP solvers employ a number of techniques *before* the LP relaxation of an instance is solved. This initial step, which is often crucial in reducing the overall solution time for large–scale instances, is referred as preprocessing [27, 47].

Preprocessing usually alters the given formulation significantly by fixing, aggregating, and/or substituting variables and constraints of the problem, 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 checking for infeasibilities, eliminating redundant constraints, and strengthening bounds on variables. Strengthening bounds refers to increasing the lower bound or decreasing the upper bound of a variable; 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 done by considering each constraint in isolation and verifying whether the constraint is inconsistent with the bounds of the variables, dominated by variable bounds, 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 will 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 LP relaxation, hence leading to improved 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 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, especially, 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 on 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 defined 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 dual LP solution, reduced cost fixing is a method for checking whether increasing or decreasing a nonbasic variable by a certain amount $\delta > 0$ 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 vary from its nonbasic value by more than δ in any optimal solution. For a 0–1 variable, $\delta < 1$ implies that the variable can be fixed to its nonbasic value. For other variables, this technique can be used to tighten their bounds.

Reduced cost fixing is particularly effective for problems with small duality gaps and 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 many others due to feasibility. On the other hand, it can be very time consuming for large–scale problems with set packing substructures due to overwhelming number of implications one tends to get for such problems.

4.1. **Software preprocessing options.**

4.1.1. *CPLEX.*

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

- `aggind`: 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.

4.1.2. *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.

4.1.3. *Xpress–Optimizer.*

Users can control preprocessing in Xpress-Optimizer 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. This 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.

4.2. **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 3 we compare the performance of CPLEX with and without preprocessing for six problems from the MIPLIB library [13]. 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 7.5 on a 2MHz Pentium4/Linux workstation with 1GB memory.

Default settings of preprocessing have a positive impact on the performance of the algorithm for most of the problems in Table 3. However, preprocessing appears to have a negative effect on gesa2 and gesa3 and it may be turned off for these problems. A closer look at p0548 shows that preprocessing strengthens the initial LP value significantly, but we don't see an impact on the number of nodes and solution time. Default settings of preprocessing improves the performance for vpm2 significantly. One may expect to gain further benefit from more aggressive preprocessing on these two problems. Indeed, increasing the level of probing, improves the performance further as shown in the last two rows in Table 3.

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. For a minimization problem, strong lower bounds are obtained by ensuring that the LP relaxation of a formulation is as close as possible to the convex hull of the integer feasible solutions. Apart from the preprocessing techniques, modern solvers employ 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

	with preprocessing					without preprocessing				
problem	`cons`	`vars`	`zinit`	`nodes`	`time`	`cons`	`vars`	`zinit`	`nodes`	`time`
fixnet6	477	877	3192.04	83	0.64	478	878	1200.88	6257	53.05
gesa2	1344	1176	2.5493e+07	148	1.90	1392	1224	2.5476e+07	52	0.92
gesa3	1296	1080	2.7846e+07	109	3.27	1368	1152	2.7834e+07	108	2.27
p0548	151	477	3126.38	28	0.25	176	548	315.25	28	0.30
p2756	729	2683	2701.14	27	1.09	755	2756	2688.75	380	6.67
vpm2	128	188	10.27	8812	7.69	234	378	9.89	18492	22.57
p0548+	151	477	4789.81	3	0.13					
vpm2+	128	188	11.34	1274	1.75					

Table 3. Impact of preprocessing

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 its LP relaxation becomes smaller but the MIP feasible region does not change. Then we can resolve the LP and repeat the process, if necessary, so long as we continue to find violated 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, we branch. Branch–and–cut algorithms generalize both pure cutting plane algorithms, in which cuts are added at the root node 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 the 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.

The cutting planes 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 [5, 24] and mixed–integer rounding cuts [38, 42]. The second category includes strong polyhedral cuts from knapsack [4, 6, 25, 28, 45], fixed–charge flow [26, 46] and path [49], and vertex packing [44] 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 lifted 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 timeindexed formulations of scheduling problems, crew pairing, staff rostering problems.

When none of the above mentioned polyhedral cuts work well, general mixed–integer cuts may help to improve the quality of the LP relaxations and thus performance. Gomory mixed–integer cuts, which had been overlooked for several decades due to initial unsuccessful implementations in pure cutting plane algorithms, have become standard technology in state–of–the–art IP solvers after their effectiveness in a branch–and–cut framework was demonstrated by Balas et al. [5]. 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 compared to the polyhedral cuts. This can make the LPs significantly harder to solve. The mixed–integer rounding variants from the original constraints may provide sparse alternatives [38].

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–Optimizer.* The parameter `cutstrategy` is used to adjust the aggressiveness of cut generation in Xpress–Optimizer. 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 incurring a 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.

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 4 we compare the performance of CPLEX with and without adding cutting planes to the formulation for six problems from the MIPLIB library [13]. 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 7.5 on a 2MHz Pentium4/Linux workstation.

Cut generation with default settings reduces the number of nodes in the search tree significantly for all problems in Table 3; and this reduces the computation time for almost all problems. Problems gesa2 and gesa3 contain general integer variables and therefore Gomory and MIR cuts are the most effective on these problems. For gesa2 the cuts are very effective and reduce the search tree and the computation by several orders of magnitude. Generating these cuts more aggressively by increasing the relevant parameter leads to even further improvement as shown in the row for gesa2+. On the other hand, for gesa3, even though cuts reduce the number of nodes, the additional time spent on cut generation and solving larger LPs appears to slow down the computations. This suggests adding a smaller number of cuts may be a better option in this case. Indeed, decreasing the aggressiveness of cut generation leads to a better performance than the default as shown in the row gesa3–.

	with cuts				without cuts		
problem	`cuts`	`zroot`	`nodes`	`time`	`zroot`	`nodes`	`time`
fixnet6	49	3529.79	83	0.64	3192.04	522	0.96
gesa2	103	2.5730e+07	148	1.90	2.5493e+07	101852	264.14
gesa3	103	2.7943e+07	109	3.27	2.7846e+07	613	2.92
p0548	135	8677.00	28	0.25	3126.38	16038	10.56
p2756	478	3107.21	27	1.09	2701.14	163074	525.68
vpm2	66	11.60	8812	7.69	10.27	222698	104.60
gesa2+	153	2.5776e+07	95	1.49			
gesa3–	85	2.7945e+07	70	2.25			

Table 4. Impact of cuts

6. Primal heuristics

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 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; and for a fixed branching rule, any node selection rule would 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.

As with cutting planes, it seems reasonable to spend more effort on finding good feasible solutions early in the search tree, since this would have the most impact on the solution process. 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.

6.1. **Software primal heuristic options.**

6.1.1. *CPLEX.* Users can adjust the aggressiveness of the heuristic application in CPLEX by setting the parameter `heurfreq`. The heuristic is then applied every `heurfreq` nodes in the tree. Setting this parameter to -1 disables the heuristic. With the default setting 0, the frequency is determined automatically by CPLEX.

6.1.2. *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.

6.1.3. *Xpress–Optimizer.* 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.

6.2. **Sample computations on 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 we compare the performance of LINDO with different heuristic levels for five problems from the MIPLIB library [13]. 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 2MHz 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.

	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

TABLE 5. Impact of primal heuristics

7. 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 the 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 use of mathematical programming in practice. As a result, the demand for better and tighter integration of modeling tools and solvers 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.

7.1. 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 lotsizing 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 LS 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 [7] 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 find 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.

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. These examples demonstrate that relatively sophisticated solution approaches can be implemented fairly easily with today's modeling and optimization software systems.

7.2. **Lot sizing using Xpress–Mosel.**

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
  ...
```

Algorithm 2 Cutting plane algorithm for LS

```
1: Read the model to obtain the current formulation
2: Solve the LP relaxation of the current formulation and let $(x, y)$ be an optimal solution
3: for $\ell = 1$ to $NT$ do
4:    $S = \emptyset$, lhsval $= 0$
5:    for $t = 1$ to $\ell$ do
6:       if $d_{t\ell} x_t < y_t$ then
7:          lhsval $=$ lhsval $+ d_{t\ell} x_t$
8:          $S = S \cup \{t\}$
9:       else
10:         lhsval $=$ lhsval $+ y_t$
11:      end if
12:   end for
13:   if lhsval $< d_{1\ell}$ then
14:      Add $\sum_{t \in \{1,2,\ldots,\ell\} \setminus S} y_t + \sum_{t \in S} d_{\ell t} x_t \geq d_{1\ell}$ to the current formulation
15:      Go to Step 2
16:   end if
17: end for
```

```
   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 (...)
...
```

A complete Xpress–Mosel code of Algorithm 2 can be found in Appendix A. For a detailed description of Dash Optimization's Xpress–Mosel please refer to [18].

7.3. **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 ELS, 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 ELS, 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 ELS, 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 [33].

The two implementations discussed above 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. We illustrate this approach using Xpress–BCL and ILOG Concert Technology.

7.4. **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 [17].

7.5. **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 [32].

A major advantage of object–oriented libraries 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 script 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 more time to develop and maintain, they have the advantage of faster run times and the flexibility provided by greater control, for example access to the preprocessed formulation and the LP tableau.

8. Challenges

Even though the many 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 also identifies potentially rewarding research topics in the area of integer programming.

8.1. **Symmetry.**

An integer program is considered to be symmetric if its variables can be permuted without changing the structure of the problem [39]. Symmetry is a phenomenon occurring in scheduling problems, graphs problems, and many other classes of optimization problems [48]. 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, $j = 1, \dots, n$, 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

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

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 [48] 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

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

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 [39] 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 [39].

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.

8.2. **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

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

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., only require $0 \leq y \leq b$ and enforce the disjunction by branching as $y = 0$ or $a \leq y \leq b$ when it is violated (when $y \in (0, a)$ in the linear programming 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 [11] 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 [19, 20] and automatic reformulations [14] may help to improve the solvability of these problems.

8.3. **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 integer 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. (Note that 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 [15, 2].

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 IPs by using the 0–1 expansion of bounded integer variables. However, this is generally a poor and counter productive strategy for several reasons including the symmetry and weak LP bounds of expanded formulations [43].

Research on polyhedral analysis of general mixed–integer knapsacks [3] and on lattice–based methods for pure integer programs [1, 37] are ongoing and may result in more powerful techniques for handling instances with general integer variables.

9. 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.

[4]XPRESS supports specialized branching for semi-continuous variables.

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 program 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 [19, 20]. 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 [29, 31, 34]. 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 [21] and its variants [16].

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 [10, 12]. 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.

Acknowledgement

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

References

[1] K. Aardal, C. A. J. Hurkens, and A. K. Lenstra. Solving a system of linear diophantine equations with lower and upper bounds on the variables. *Mathematics of Operations Research*, 25(3):427–442, 2000.

[2] K. Aardal and A.K. Lenstra. Hard equality constrained integer knapsacks. In W.J. Cook and A.S. Schultz, editors, *Proc. 9th International IPCO Conference*, pages 350–366. Springer-Verlag, 2002.

[3] A. Atamtürk. On the facets of mixed–integer knapsack polyhedron. *Mathematical Programming*, 98:145–175, 2003.

[4] E. Balas. Facets of the knapsack polytope. *Mathematical Programming*, 8:146–164, 1975.

[5] E. Balas, S. Ceria, G. Cornuéjols, and N.R. Natraj. Gomory cuts revisited. *Operations Research Letters*, 19:1–9, 1996.

[6] E. Balas and E. Zemel. Facets of the knapsack polytope from minimal covers. *SIAM Journal of Applied Mathematics*, 34:119–148, 1978.

[7] I. Barany, T. J. Van Roy, and L. A. Wolsey. Uncapacitated lot sizing: The convex hull of solutions. *Mathematical Programming Study*, 22:32–43, 1984.

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

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

[10] D. Bertsimas and M. Sim. Robust discrete optimization and network flows. *Mathematical Programming*, 98:49–71, 2003.

[11] D. Bienstock. Computational study of a family of mixed–integer quadratic programming problems. *Mathematical Programming*, 74:121–140, 1996.

[12] J. R. Birge and F. Louveaux. *Introduction to Stochastic Programming.* Springer Verlag, New York, 1997.

[13] R. E. Bixby, S. Ceria, C. M. McZeal, and M. W. P. Savelsbergh. An updated mixed integer programming library: MIPLIB 3.0. *Optima*, 54, 1998. URL `http://www.caam.rice.edu/∼bixby/miplib/miplib.html`.

[14] G. Codato and M. Fischetti. Combinatorial benders' cuts. First draft, September 2003.

[15] G. Cornuejols and M. Dawande. A class of hard small 0-1 programs. In R. E. Bixby, E. A. Boyd, and R. Z. Rios-Mercado, editors, *Proc. 6th International IPCO Conference*, pages 284–293. Springer-Verlag, 1998.

[16] E. Danna, E. Rothberg, and C Le Pape. Exploring relaxation induced neighborhoods to improve mip solutions. Technical report, ILOG, Inc., 2003.

[17] Dash Optimization, Ltd. *XPRESS BCL User Guide – Release 1.2*, 2003.

[18] Dash Optimization, Ltd. *XPRESS Mosel User Guide – Release 1.2*, 2003.

[19] I. R. de Farias, Ellis L. Johnson, and G. L. Nemhauser. Branch-and-cut for combinatorial optimisation problems without auxiliary binary variables. *Knowledge Engineering Review*, 16:25–39, 2001.

[20] I. R. de Farias, Ellis L. Johnson, and G. L. Nemhauser. A polyhedral study of the cardinality constrained knapsack problem. *Mathematical Programming*, 96:439–467, 2003.

[21] M. Fischetti and A. Lodi. Local branching. *Mathematical Programming*, 98:23–47, 2003.

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

[23] A. M. Geoffrion and R. E. Martsen. Integer programming algorithms: A framework and state-of-the-art survey. *Management Science*, 18(9):465–491, 1972.

[24] R. E. Gomory. An algorithm for the mixed integer problem. Technical Report RM-2597, The Rand Corporation, 1960.

[25] Z. Gu, G. L. Nemhauser, and M. W. P. Savelsbergh. Lifted cover inequalities for 0–1 integer programs: Computation. *INFORMS Journal on Computing*, 10:427–437, 1998.

[26] Z. Gu, G. L. Nemhauser, and M. W. P. Savelsbergh. Lifted flow cover inequalities for mixed 0–1 integer programs. *Mathematical Programming*, 85:439–467, 1999.

[27] M. Guignard and K. Spielberg. Logical reduction methods in zero–one programming. *Operations Research*, 29:49–74, 1981.

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

[29] I. Harjunkoski, V. Jain, and I. E. Grossman. Hybrid mixed-integer/constraint logic programming strategies for solving scheduling and combinatorial optimization problems. *Computers and Chemical Engineering*, 24:337–343, 2000.

[30] J. P. H. Hirst. Features required in branch and bound algorithms for (0-1) mixed integer linear programming. Privately circulated manuscript, December 1969.

[31] J. N. Hooker, G. Ottosson, E. S. Thornsteinsson, and H.-J. Kim. A scheme for unifying optimization and constraint satisfaction methods. *Knowledge Engineering Review*, 15:11–30, 2000.

[32] ILOG, Inc. *ILOG CPLEX 8.0 Reference Manual*, 2002.

[33] ILOG, Inc. *ILOG OPL User's Manual*, 2002.

[34] V. Jain and I. E. Grossmann. Algorithms for hybrid MILP/CP methods. *INFORMS Journal on Computing*, 13:258–276, 2001.

[35] A. Land and S. Powell. Computer codes for problems of integer programming. In P. L. Hammer, E. L. Johnson, and B. H. Korte, editors, *Discrete Optimization II*, pages 221–269. North Holland Publishing Co., 1979.

[36] LINDO Systems, Inc. *LINDO API User's Manual*, 2002.

[37] Q. Louveaux and L. A. Wolsey. Combining problem structure with basis reduction to solve a class of hard integer programs. Technical Report 00/51, CORE, Catholic University of Louvain-la-Neuve, 2000.

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

[39] F. Margot. Exploiting orbits in symmetric ilp. *Mathematical Programming*, 98:3–21, 2003.

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

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

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

[43] J. H. Owen and S. Mehrotra. On the value of binary expansions for general mixed-integer linear programs. *Operations Research*, 50(5):810–819, 2002.

[44] M. W. Padberg. On the facial structure of set packing polyhedra. *Mathematical Programming*, 5:199–215, 1973.

[45] M. W. Padberg. Covering, packing and knapsack problems. *Annals of Discrete Mathematics*, 4:265–287, 1979.

[46] M. W. Padberg, T. J. Van Roy, and L. A. Wolsey. Valid linear inequalities for fixed charge problems. *Operations Research*, 33:842–861, 1985.

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

[48] H. D. Sherali and J. C. Smith. Improving discrete model representations via symmetry considerations. *Management Science*, 47:1396–1407, 2001.

[49] T. J. Van Roy and L. A. Wolsey. Solving mixed integer programming problems using automatic reformulation. *Operations Research*, 35:45–57, 1987.

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

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;
}
```

Alper Atamtürk: Industrial Engineering and Operations Research, University of California, Berkeley, CA 94720–1777 USA, email: atamturk@ieor.berkeley.edu.

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